Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação

DISSERTAÇÃO DE MESTRADO

JFloat: Uma biblioteca de ponto flutuante para a linguagem Java com suporte a arredondamento direcionado

José Frank Viana da Silva

Natal/RN

Novembro/2007

José Frank Viana da Silva

JFloat: Uma biblioteca de ponto flutuante para a linguagem Java com suporte a arredondamento direcionado

Natal, Rio Grande do Norte - Brasil

Novembro de 2007

Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação

JFloat: Uma biblioteca de ponto flutuante para a linguagem Java com suporte a arredondamento direcionado

José Frank Viana da Silva

Orientador Prof. Dr. Benjamín René Callejas Bedregal
Co-orientador Prof. Dr. Regivan Hugo Nunes Santiago

Dissertação submetida e aprovada no Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como parte dos requisitos para a obtenção do grau de Mestre em Sistemas e Computação (MSc.).

Natal, Rio Grande do Norte - Brasil
Novembro de 2007

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial Especializada do Centro de Ciências Exatas e da Terra – CCET.

Silva, José Frank Viana da.

JFloat : uma biblioteca de ponto flutuante para a linguagem Java com suporte a arredondamento direcionado / José Frank Viana da Silva. – Natal, 2007.

106 f. : il.

Orientador: Prof. Dr. Benjamin René Callejas Bedregal.

Co-orientador: Prof. Dr. Regivan Hugo Nunes Santiago.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Inteligência computacional – Dissertação. 2. Matemática Intervalar – Dissertação. I. Bedregal, Benjamín René Callejas. II. Santiago, Regivan Hugo Nunes III. Título

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

Os examinadores abaixo recomendam ao Departamento de Informática e Matemática Aplicada a aceitação da dissertação intitulada **"JFloat: Uma biblioteca de ponto flutuante para a linguagem Java com suporte a arredondamento direcionado"** de autoria de **José Frank Viana da Silva** como requisito para a obtenção do grau de **Mestre em Sistemas e Computação**.

30 de novembro de 2007

Presidente: D.Sc. Benjamín René Callejas Bedregal (UFRN)

_____

Examinadores: D.Sc. Ivan Saraiva Silva (UFRN)

_____

D.Sc. Tiarajú Asmuz Diverio (UFRGS)

_____

Este trabalho é dedicado aos meus pais pela confiança que depositaram em mim e por seu imenso carinho. A Sandra por seu apoio incondicional, por seu amor e compreensão e meu filho Arthur.

AGRADECIMENTOS

No processo para terminar este trabalho, vi como cada pessoa que passou pela minha vida foi importante e contribuiu de alguma forma na realização deste objetivo alcançado.

Quero agradecer primeiramente a Deus, porque sempre esteve a meu lado.

Aos meus pais que são o apoio e força para continuar lutando e alcançar meus objetivos.

Ao meu amor Sandra, por seu carinho, apoio incondicional e incentivo.

Arthur, meu primeiro filho, que me anima todos os dias com seu sorriso.

Ao amigo Osmar pelo incentivo para fazer o mestrado.

Ao professor e orientador Benjamín René Callejas Bedregal pela oportunidade que me deu de poder fazer o mestrado, por sua confiança, por sua orientação ao longo deste trabalho e pela amizade.

Ao professor e co-orientador Regivan Hugo Nunes Santiago por me ajudar a encontrar várias respostas para criação desse trabalho.

Finalmente quero agradecer a todas as pessoas que contribuíram de alguma forma para este trabalho.

JFloat: Uma biblioteca de ponto flutuante para a linguagem Java com suporte a arredondamento direcionado

José Frank Viana da Silva

Orientador: Prof. Dr. Benjamín René Callejas Bedregal
Co-orientador: Prof. Dr. Regivan Hugo Nunes Santiago

RESUMO

Este trabalho apresenta JFloat, uma implementação de software do padrão IEEE-754 de aritmética de ponto flutuante binária. JFloat foi construída para prover algumas características não implementadas em Java, especificamente o suporte ao arredondamento direcionado. Esta característica é de grande importância para o prosseguimento do projeto Java-XSC, em desenvolvimento por esta linha de pesquisa. Apesar de programas escritos em Java, a princípio, serem portáveis para qualquer arquitetura ao usar operações de ponto flutuante, principalmente porque IEEE-754 especifica que programas deveriam ter precisamente o mesmo comportamento em toda configuração, observou-se que programas que usam tipos de ponto flutuantes nativos de Java podem ser dependentes da máquina e do sistema operacional. JFloat também se apresenta como uma possível solução para este problema.

JFloat: A Floating Point Library with Directed Rounding Mode Support for Java language

José Frank Viana da Silva

Advisor: Prof. Dr. Benjamín René Callejas Bedregal

Advisor: Prof. Dr. Regivan Hugo Nunes Santiago

ABSTRACT

This work presents JFLoat, a software implementation of IEEE-754 standard for binary floating point arithmetic. JFloat was built to provide some features not implemented in Java, specifically directed rounding support. That feature is important for Java-XSC, a project developed in this Department. Also, Java programs should have same portability when using floating point operations, mainly because IEEE-754 specifies that programs should have exactly same behavior on every configuration. However, it was noted that programs using Java native floating point types may be machine and operating system dependent. Also, JFloat is a possible solution to that problem.

Sumário

Capítulo 1

Introdução

No princípio da era da computação, o principal objetivo buscado era aumentar o desempenho dos computadores, em especial a velocidade de execução, sem que existisse a preocupação com a qualidade dos resultados. Por isso, os programas numéricos eram freqüentemente afetados por erros de arredondamento e truncamento decorrentes do uso do sistema de ponto flutuante. Tais erros se acumulavam durante os cálculos e produziam resultados inaceitáveis.

Uma das propostas surgidas para o aumento da exatidão dos cálculos numéricos foi a matemática intervalar. Ela se baseia no uso de intervalos fechados para representar dados de entrada inexatos ou que não podem ser representados finitamente na máquina e que causam erros de truncamento ou arredondamento durante a execução do programa (SANTOS, 2001). Baseadas na matemática intervalar surgiram diversas iniciativas para criação de bibliotecas de software que permitissem aos pesquisadores trabalhar com operações definidas por essa matemática.

Uma dessas iniciativas, desenvolvida no Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte, foi Java-XSC (DUTRA, 2000), uma biblioteca intervalar escrita na linguagem de programação Java. Um problema atualmente enfrentado por Java-XSC é que para poder computar operações intervalares com maior exatidão, esta biblioteca requer o controle do arredondamento das operações aritméticas usando ponto flutuante. Embora este controle seja requerido na especificação do padrão de representação e modo de operação de números em ponto flutuante (IEEE-754), a ausência na linguagem de programação Java dessa característica, dificulta o controle da exatidão dos resultados.

O principal motivo para o presente trabalho é dar suporte ao arredondamento direcionado a linguagem Java. JFloat é uma biblioteca de software escrita em Java e independente de máquina que implementa o padrão IEEE-754 e pretende contribuir para a linha de pesquisa e o aperfeiçoamento de Java-XSC. A biblioteca foi projetada para suportar todos os modos de arredondamento exigidos pelo padrão, fornecendo aos desenvolvedores os meios para criação de aplicações em Java que necessitem do controle do tipo de arredondamento.

Outro ponto importante, durante a elaboração desse trabalho foi a detecção de outro problema em Java, relacionado à portabilidade de programas numéricos que usam ponto flutuante. A linguagem de programação Java possui grande portabilidade, uma vez que aplicações escritas nessa linguagem podem executar, sem mudanças no código compilado, em qualquer sistema operacional que disponha de uma máquina virtual Java. Principalmente, porque o padrão IEEE-754 especifica que programas em conformidade com ele deveriam ter precisamente o mesmo comportamento em qualquer configuração de hardware e software (ANSI/IEEE, 1985). Todavia, foi identificado durante os testes que programas usando os tipos de ponto flutuantes nativos da linguagem Java podem ser influenciados pelo hardware e software sobre o qual executam. Outros trabalhos (BOISVERT e MOREIRA, 2001; DOERDELEIN, 2005) também relatam a mesma anomalia.

Uma das causas para o problema é que apesar das representações das operações serem as mesmas em todas as máquinas, pode ocorrer, durante a avaliação de expressões em ponto flutuante, a geração de valores intermediários que acarretem diferenças de resultados entre plataformas. Por exemplo, um programa escrito em Java que esteja executando em um processador Intel, poderia antes de uma divisão de dois operandos de ponto flutuante com precisão dupla (64 bits de largura), converter estes operandos para a precisão estendida interna (80 bits de largura), executar a divisão e então o quociente ser arredondado para dupla precisão. Em outras plataformas, a mesma divisão dos mesmos operandos de precisão dupla poderia acontecer sem a conversão intermediária para precisão estendida e eventualmente gerar resultados com valores ligeiramente diferentes.

Para garantir a portabilidade entre plataformas diferentes, em vez de usar os tipos de ponto flutuantes nativos da linguagem, JFloat usa uma representação interna independente de máquina e um conjunto de operações de ponto flutuante que atuam sobre esse formato interno. Para tanto, JFloat armazena os números em ponto flutuante como inteiros binários e funções aritméticas foram projetadas para trabalhar com esta representação interna.

Essa dissertação está dividida em oito capítulos e anexos:

Capítulo 1. Esta introdução.

Capítulo 2. **Representação numérica**. Este capítulo aborda as formas de representação numérica, em especial dos números em ponto flutuante usando o padrão IEEE-754.

Capítulo 3. Uma introdução à **Matemática Intervalar**.

Capítulo 4. **Linguagens XSC e bibliotecas intervalares**. O capítulo expõe o problema da computação numérica de alta exatidão, destacando as Extensões para Linguagens de

Computação Científica (Languages eXtensions SCientific - XSC), em especial Java-XSC e a motivação para o desenvolvimento de uma biblioteca Java com arredondamento direcionado.

Capítulo 5. **Bibliotecas de ponto flutuante**. São apresentados nesse capítulo bibliotecas de ponto flutuante nas linguagens C e Java que ofereceram subsídios para o planejamento e desenvolvimento deste trabalho.

Capítulo 6. **Biblioteca JFloat**. Neste capítulo, é apresentada a biblioteca JFloat, fruto do trabalho desenvolvido, destacando as decisões tomadas durante o desenvolvimento do projeto de suporte ao uso de números em ponto flutuante com controle direcionado do arredondamento em Java.

Capítulo 7. **Testes** efetuados com a biblioteca.

Capítulo 8. **Considerações finais**. O capítulo expõe as vantagens do uso da biblioteca criada e aponta as melhorias que poderão ser feitas.

Anexo A. Documentação de referência das **classes**.

Anexo B. Código fonte dos testes.

Capítulo 2

Representação numérica

2.1. Introdução

Este capítulo é uma introdução sobre a forma de representação de valores numéricos. Em especial, será mostrado como números são representados e armazenados na memória do computador nos formatos inteiro e ponto flutuante.

2.2. Representação de números inteiros

Basicamente, números inteiros podem ser representados computacionalmente nas formas sinal-magnitude e complemento de base.

2.2.1. Sinal-magnitude

Na forma conhecida como sinal-magnitude, o primeiro bit do conjunto de bytes do número determina seu sinal. O valor 0 (zero), normalmente, indica que o número é positivo e o valor 1 (um) que é negativo. A magnitude é a mesma tanto para números positivos quanto para negativos, variando apenas o bit de sinal.

A representação na base b em sinal e magnitude com n bits (incluindo o bit de sinal) possui b^n representações e permite representar b^n-1 valores, uma vez que há duas representações para o zero. Em especial, para a base 2, a faixa de representação por sinal e magnitude com n bits (incluindo o bit de sinal) possui 2^n representações, representando os valores entre $-(2^{n-1}-1)$ e $+(2^{n-1}-1)$. O maior valor inteiro positivo será então $+(2^{n-1}-1)$ e o menor valor inteiro negativo será $-(2^{n-1}-1)$.

Representação de número inteiro por sinal-magnitude

Sinal	**n-2**	**...**	**0**
±	Bit mais significativo	...	Bit menos significativo

Exemplo com base = 2 e magnitude = 3:

Decimal	Binário	Decimal	Binário
-3	1 | 11	+0	0 | 00
-2	1 | 10	+1	0 | 01
-1	1 | 01	+2	0 | 10
-0	1 | 00	+3	0 | 11

2.2.2. Complemento de base

Complemento é a diferença entre cada algarismo do número e o maior algarismo possível na base. Uma vantagem da utilização da representação em complemento é que a subtração entre dois números pode ser substituída pela sua soma em complemento.

A representação de números inteiros positivos em complemento não tem qualquer alteração, sendo idêntica à representação em sinal e magnitude.

A representação dos números inteiros negativos é obtida efetuando-se: (base-1) menos cada algarismo do número.

Para se obter o complemento a 2 de um número binário, deve-se subtrair cada algarismo de 1. Uma particularidade dos números binários é que, para efetuar esta operação, basta inverter todos os bits e adicionar ao resultado o número 1. A faixa de representação será de -2^{n-1} até $+2^{n-1} - 1$.

Representação de número inteiro por complemento de base

n-1	...	0
Bit mais significativo	...	Bit menos significativo

Exemplo com base = 2 e magnitude = 3:

Decimal	Binário	Decimal	Binário
-1	1 | 11	0	0 | 00
-2	1 | 10	+1	0 | 01
-3	1 | 01	+2	0 | 10
-4	1 | 00	+3	0 | 11

2.3. Representação de números reais

Embora os números reais sejam, essencialmente, objetos infinitos e por isso, não sejam computacionalmente representáveis (BEDREGAL e ACIÓLY, 1993), é possível representá-los finitamente por meio de aproximações.

Por exemplo, o número real π poderia ser aproximado por qualquer um dos números abaixo de acordo com a exatidão desejada:

- 31416×10^{-4},
- $31415926535897932384626433832795 \times 10^{-31}$,
- $31415926535897932384626433832795028841971693993751058209749445923078164062862089 \times 10^{-79}$.

2.3.1. Representação de números em ponto flutuante

O formato de **ponto flutuante**, também conhecido como **vírgula flutuante**, é uma representação computacional dos números reais, na forma

[sinal] significando × base $^{\text{expoente}}$

Nesta representação, a posição da vírgula pode ser movida de uma posição para outra de acordo com a exatidão desejada; por isso o nome flutuante. Em essência, o formato em ponto flutuante é similar à notação científica de números. Especificamente, para base 2 teremos a forma

[sinal] significando × 2 $^{\text{expoente}}$

2.4. Padrão IEEE-754

No início da computação digital, cada fabricante de computadores desenvolveu sua própria representação e métodos de cálculo de números em ponto-flutuante. Estas diferentes representações ocasionavam resultados diferentes nos cálculos, quando feitos em máquinas diferentes.

Em 1985, para encerrar a incompatibilidade entre sistemas, a organização IEEE (**Institute of Electrical and Electronics Engineers**) criou o padrão IEEE-754 que estabeleceu a representação e modo de operação dos números de ponto flutuante em binário (ANSI/IEEE, 1985). O padrão definiu uma família de meios comercialmente possíveis para executar aritmética em ponto flutuante. Ele teve por diretrizes:

- A portabilidade de aplicações já existentes, entre diversos computadores com esse padrão, de tal forma que apresentassem o mesmo resultado,
- Tornar fácil e seguro produzir programas para área matemática, mesmo não requerendo muita experiência do programador,
- Providenciar suporte direto para diagnosticar, em tempo de execução, as anomalias, tratando uniformemente as exceções,
- Prover funções elementares padrão,
- Permitir refinamentos e extensões.

O padrão especifica:

- Formatos básico e estendido para números em ponto flutuante,
- Operações de adição, subtração, multiplicação, divisão, raiz quadrada, resto e comparação,
- Conversões entre inteiros e formatos de ponto flutuante,
- Funções de conversão entre diferentes formatos de ponto flutuante,
- Funções de conversão entre números de ponto flutuante no formato básico para cadeia de caracteres (**strings**) decimais,
- Exceções de ponto flutuante e sua manipulação, incluindo **NaN (Not a Numbers)**.

O padrão não especifica:

- Formatos de cadeias de caracteres decimais e inteiros,
- Interpretação do sinal de campos do significando e NaN,
- Conversões de decimal para binário e vice-versa de formatos estendidos.

Os valores finitos representáveis por um número no formato IEEE-754 podem ser expressos por uma equação com dois parâmetros inteiros k e n, mais duas constantes N e K, dependentes do formato:

$$\text{Valor finito} = n \times 2^{k+1-N}$$

Onde o parâmetro k é o expoente com k + 1 bits. O valor de k varia entre

$$E_{min} = -(2^k-2) \text{ e } E_{max} = 2^k - 1$$

O parâmetro n é o enésimo bit do significando.

2.4.1. Normalização

Representações de números em ponto flutuante não são necessariamente únicas. No caso do bit inicial do significando ser diferente de zero a **representação** é dita **normalizada** (GOLDBERG, 1991). Quando possível, para evitar múltiplas representações para o mesmo valor, este deverá ser normalizado. Para tanto, o expoente k deverá ser minimizado para permitir que o significando tenha o valor 1 (um) em seu primeiro bit.

A normalização em ponto flutuante é a forma na qual exatamente um dígito diferente de 0 (zero) forma a parte inteira do significando. A normalização é a operação de deslocar a parte fracionária de modo a fazer o bit à esquerda da vírgula seja igual a 1. Durante esse deslocamento, o expoente será incrementado. Somente um dígito 1 ficará na parte inteira. Uma vez que esta informação é redundante, somente a parte fracionária precisa ser armazenada e a parte inteira é descartada. A única exceção a essa regra é o número zero, em que todos os bits são iguais a zero. (BELANOVIC, 2002).

Quando este processo de normalização não for possível, será obtido um número **denormalizado**, também conhecido como **subnormalizado**. Números denormalizados possuem o menor expoente possível. No padrão IEEE-754, os números denormalizados são usados para representar valores entre zero e o menor número normalizado.

2.4.2. Formato

Valores de ponto flutuante são codificados em memória usando um modelo de três campos. Do bit mais significativo ao menos significativo, são armazenados os campos de **sinal, expoente e significando,** onde o sinal $\in \{0,1\}$, o expoente é um inteiro sinalizado e o significando é um inteiro positivo.

$$\text{Valor finito} = (-1)^{\text{sinal}} \times 2^{\text{expoente}} \times \text{significando}$$

Armazenamento de um número em ponto flutuante na memória

Sinal	Expoente	Significando

O padrão IEEE-754 interpreta um número em ponto flutuante binário como se o mesmo fosse dividido em quatro grupos: **sinal, expoente, bit implícito, parte fracionária**.

- O sinal indica se o número é negativo (1) ou positivo (0),

- O expoente armazena a potência de 2 para a qual o número será elevado, o mesmo é baseado em uma representação **transladada** ou **polarizada** (**bias exponent**). Isto significa que o valor do expoente do número em ponto flutuante é representado por um número inteiro mais um deslocamento (**bias**). O motivo para uso do expoente polarizado é permitir a representação dos expoentes na faixa de valores E_{min} e E_{max}, inclusive, E_{min-1} para codificar +0 (zero positivo), –0 (zero negativo) e números denormalizados e $E_{max + 1}$ para codificar ±∞ e NaNs. Por exemplo, se **bias** = 127, o expoente 53 é representado pelo expoente polarizado 180 = 127 + 53; o expoente -27 por sua vez, é representado pelo expoente polarizado 100 = 127 + (-27). Se expoente = 0, estará sendo representado o número 0 ou um número denormalizado,
- O **bit implícito** fica à esquerda da parte fracionária. Em regra, este valor será 1 (um) e portanto não será necessário seu armazenamento,
- A **parte fracionária** é o valor que fica à direita do ponto flutuante. Chama-se **significando** a junção do bit implícito mais a parte fracionária,
- A área finita de armazenamento limita o quão perto estarão dois números em ponto flutuante adjacentes.

Para um número normalizado, o bit mais significativo do significando será sempre 1 (um) e por isso, este bit implícito não precisa ser armazenado.

Número real normalizado (32 bits)

Sinal	Expoente	Fração	Valor
x	00000001 a 11111110	00000000000000000000000 a 11111111111111111111111	Número real normalizado

Valores **sub-normalizados** ou **denormalizados** são codificados dentro de um valor especial de expoente fora de faixa, isto é, E_{min} - 1.

Número denormalizado (32 bits)

Sinal	Expoente	Fração	Valor
x	00000000	xxxxxxxxxxxxxxxxxxxxxxx	Número real denormalizado

2.4.3. Valores especiais

Além dos números discutidos acima, o IEEE-754 inclui valores especiais representando **NaN** (**Not a Number**), $+\infty$ e $-\infty$. Valores especiais são codificados com o uso do expoente fora da faixa $E_{max} + 1$. Os valores -0.0 e +0.0 são distinguíveis embora seja verdadeira a expressão +0.0 = -0.0. Juntos, os números normalizados, os denormalizados e os zeros são referidos como números finitos.

Com exceção dos NaNs, se os bits de um número em ponto flutuante IEEE fossem interpretados como inteiros sinalizados, eles terão a mesma ordem lexicográfica que suas contrapartes em inteiro.

Os NaNs são usados no lugar de vários resultados inválidos; tais como: "0/0". Na realidade muitos padrões diferentes de bit podem codificar um valor NaN. A intenção é permitir que sejam repassadas informações adicionais no significando do NaN, tais como o endereço da operação inválida que gerou um NaN. As informações contidas nos NaNs só podem ser obtidas por meios não aritméticos.

Para atribuição e operações aritméticas, se um único NaN é dado como um operando o mesmo NaN deverá ser retornado como resultado; se dois operandos NaN forem fornecidos, um deles deverá ser retornado.

O padrão também define a comparação entre números em ponto flutuante, como exemplo, as relações **maior que** (>), **igual** (=) e **menor que** (<). O padrão também estabelece que não se pode comparar NaNs com as relações primitivas. Um NaN não é nem menor que, nem maior que, nem igual a qualquer valor em ponto flutuante, inclusive ele mesmo. Portanto, as expressões NaN > a, NaN = a, NaN < a são todas falsas, para qualquer valor de a, inclusive a sendo um NaN.

Por último, os NaNs são classificados em **QNaN** (**Quiet NaN**) onde o bit mais significativo da fração é igual a 1 e **SNaN** (**Signaling NaN**) quando o bit mais significativo da fração é igual a 0. QNaNs representam operações indeterminadas (ex: 1/0 = ∞) e SNaNs são decorrentes de operações inválidas (ex: 0/0) e que requerem a sinalização de uma exceção.

NaNs (32 bits)

Sinal	Expoente	Fração	Valor
x	11111111	1xxxxxxxxxxxxxxxxxxxxx	QNaN
x	11111111	0xxxxxxxxxxxxxxxxxxxxx	SNaN

O padrão IEEE-754 especifica ainda formatos especiais para alguns números:

- O zero é representado com todos os bits do expoente e significando iguais a 0 (zero).

Zero (32 bits)

Sinal	Expoente	Fração	Valor
X	00000000	00000000000000000000000	0

- Os valores infinitos (±∞) são representados com todos os bits do expoente igual a 1 e todos os bits do significando iguais a zero.

Infinito (32 bits)

Sinal	Expoente	Fração	Valor
0	11111111	00000000000000000000000	+∞
1	11111111	00000000000000000000000	-∞

2.4.4. Modos de arredondamento

Por padrão, uma operação aritmética IEEE-754 comporta-se como se o resultado fosse computado exatamente e então arredondado para o ponto flutuante mais próximo ao resultado exato. Embora o arredondamento para o número par mais próximo (**rounding nearest even**) seja geralmente o tipo de arredondamento mais adequado, alguns aplicativos requerem outros modos de arredondamento.

Os modos de arredondamento definidos pelo IEEE-754, os quais podem ser definidos e consultados dinamicamente em tempo de execução, são:

- **Float round nearest even (para o número par mais próximo)**. O resultado deve ser o valor, dentro do formato, mais próximo do resultado infinitamente preciso. Se dois valores são igualmente próximos, então deve ser considerado aquele que tem o **bit** menos significativo igual a 0.
- **Float round down (para baixo ou -∞)**. O resultado deve ser o valor, dentro do formato e possivelmente o valor -∞, mais próximo e não maior que o resultado infinitamente preciso.

- **Float round up (para cima ou +∞)**. O resultado deve ser o valor, dentro do formato e possivelmente o valor +∞, mais próximo e não menor que o resultado infinitamente preciso.
- **Float round to zero (para zero)**. O resultado deve ser o valor, dentro do formato e possivelmente o valor 0, mais próximo e não maior em módulo que o resultado infinitamente preciso.

2.4.5. Precisão

O padrão IEEE-754 define um conjunto de precisões baseadas na largura de bits usados para compor o expoente e o significando. As precisões simples e a dupla são as mais comumente usadas. Abaixo são apresentados os formatos definidos no padrão:

Formatos de números em ponto flutuante

Característica/Precisão	Simples	Simples estendida	Dupla	Dupla estendida	Quádrupla
Valor máximo do expoente	+127	+1023	+1023	+16383	+16383
Valor mínimo do expoente	-126	-1022	-1022	-16382	-16382
Deslocamento do expoente (bias)	+127	+1023	+1023	+16383	+16383
Precisão	24	>32	53	>64	113
Total de bits	32	>43	64	80	128
Bit do sinal	1	1	1	1	1
Bits do expoente	8	11	11	15	15
Bits do significando	23	>32	52	64	112

2.4.6. Condições excepcionais

Na avaliação de expressões em ponto flutuante, várias condições excepcionais poderão acontecer. Essas condições indicam eventos que podem exigir providências por parte do programador. Os cinco eventos excepcionais definidos pelo IEEE-754 são em ordem decrescente de severidade:

1. **Operação Inválida.** NaN obtido a partir de operandos diferentes de NaN, por exemplo, 0/0 e √-1,

2. **Overflow.** Resultado largo demais para ser representado. Dependendo do modo de arredondamento e operandos, infinito, ou o número mais positivo ou mais negativo será retornado. A exceção de **operação inexata** será assinalada,
3. **Divisão por zero.** Um dividendo diferente de zero com um divisor zero retorna um 1 exatamente,
4. **Underflow**. Um valor denormalizado foi obtido,
5. **Operação Inexata**. O resultado não é exatamente representável. Algum arredondamento ocorreu, o que na verdade é um evento muito comum. Por exemplo, a divisão de 1/10 é uma operação inexata, pois resultará em uma dízima periódica que não pode ser representada finitamente pela máquina.

2.4.7. Flags e Traps

O padrão usa indicadores (**flags**) e interrupções (**traps**) como mecanismos para lidar com as condições excepcionais. Indicadores são **obrigatórios**, já as interrupções são **opcionais** apesar de largamente estarem disponíveis em processadores que sigam o padrão IEEE-754.

Quando são levantadas exceções (**raise exception**), uma das seguintes ações pode ser tomada:

- Por padrão, a exceção é simplesmente apresentada no indicador (**flag**) de ponto flutuante e o programa continua como se nada tivesse acontecido,
- A operação produz um valor padrão que depende da exceção. Então, o programa pode conferir os indicadores para verificar quais exceções ocorreram,
- Alternativamente, se podem habilitar interrupções (**traps**) para exceções. Neste caso, quando uma exceção é levantada, o programa receberá um sinal de exceção que deverá ser tratado. Normalmente, a ação padrão para este sinal é terminar o programa, porém pode ser modificada para suspender a execução do programa e chamar uma rotina criada pelo usuário para tratar a ocorrência.

2.5. Números dinâmicos

Em contraste ao formato de ponto flutuante apresentado e adotado pela maioria dos computadores e que possui um número fixo de bytes para representar um número, existem alternativas a esse formato. Uma possibilidade é o uso de números dinâmicos. Esse formato é

dito **dinâmico** "quando ele pode variar o espaço utilizado por sua representação interna de modo a se ajustar automaticamente à ordem de grandeza do valor por ele armazenado" (OLIVEIRA JUNIOR, 2004).

Um exemplo de biblioteca amplamente conhecida que suporta números dinâmicos é a **GMP** (**GNU Multiple Precision Arithmetic**)[1]. Ela é uma biblioteca aritmética de precisão arbitrária que opera sobre números inteiros sinalizados, números racionais e em ponto flutuante. Não há limite prático para a precisão, exceto a decorrente da quantidade de memória disponível. A biblioteca GMP é aplicada principalmente em programas de criptografia e sistemas algébricos.

[1] http://www.gnu.org

Capítulo 3

Erros Numéricos e Arredondamento Direcionado

3.1. Introdução

No passado, os projetistas visavam aumentar o desempenho dos computadores, em especial a velocidade de execução, sem que houvesse a preocupação com a qualidade dos resultados. Ocorre que os programas numéricos são freqüentemente afetados por erros de arredondamento e truncamento decorrentes do uso do sistema de ponto flutuante. Tais erros, associados à imprecisão da modelagem, acumulam-se durante os resultados intermediários e podem produzir resultados inaceitáveis. Esse erro de aproximação é a distância entre o irracional e sua aproximação racional. Esse erro não obedece a qualquer regra durante as computações, o que pode levar a distorções em uma computação constituída de muitos passos.

Tomando como exemplo o programa Microsoft Excel (MICROSOFT), observa-se que quando se operam números reais na base decimal, eles são representados internamente pelo computador por um valor binário correspondente. Por exemplo, a fração 1/10 pode ser representada no sistema de numeração decimal por 0.1. Entretanto, o mesmo número em binário não tem uma representação finita e possui o formato 0.00110011001110010011... Como este número não pode ser representado em uma quantidade infinita de espaço, será necessário arredondá-lo.

Além disso, nem todos os números existentes podem ser representados, pois as estruturas de dados que mantém os valores possuem um número mínimo e um número máximo que podem ser representados. Devido ao número de bits da memória ser finito, os valores mínimo e máximo que podem ser representados também são finitos. Novamente, tomando como exemplo o Excel, o menor número possível de ser representado é $2{,}2250738585072 \times 10^{-308}$ e o maior número $1{,}79769313486232 \times 10^{+308}$.

3.2. Falhas causadas por erros numéricos

A prevenção de erros causados pelo arredondamento ou truncamento de resultados possui grande interesse prático. Falhas de programas numéricos são causa de catástrofes e

perda de vidas e recursos materiais. Na literatura, podem-se encontrar exemplos de diversos desastres causados por erros numéricos. Abaixo, são citadas algumas falhas catastróficas descritas em (VUIK):

- **Falha de míssil Patriot**. Em fevereiro de 1991, durante a guerra do Golfo, um míssil americano Patriot falhou em interceptar um míssil Scud iraquiano. O incidente causou a morte de 28 soldados americanos. Um relatório chegou a conclusão que o tempo em décimos de segundo, como medido pelo relógio interno do sistema, era multiplicado por 1/10 para produzir o tempo em segundos. Este cálculo era realizado usando um registrador de ponto fixo de 24 bits. Em particular o valor 1/10, o qual é uma dízima periódica no formato binário, foi arredondado 24 bits após o ponto decimal. Esse pequeno erro acumulado em um grande número de iterações acarretou um erro final significante,
- **Explosão do Ariane V**. Em junho de 1996, um foguete lançado pela Agência Espacial Européia explodiu após quarenta segundos de seu lançamento. O foguete estava fazendo sua primeira viagem após uma década de desenvolvimento ao custo de sete bilhões de dólares. O foguete e sua carga estavam estimados em 500 milhões de dólares. Especificamente, um registro que armazenava um número de ponto flutuante de 64 bits relacionando a velocidade horizontal do foguete em relação à plataforma foi convertido em um inteiro de 16 bits sinalizado. Como o número era maior que 32767, o maior inteiro armazenável em um inteiro de 16 bits sinalizado, o sistema entrou em pane,
- **Bolsa de valores de Vancouver**. No ano de 1982, a bolsa de valores de Vancouver instituiu um novo índice inicializado para um valor de 1.000,000. O índice era atualizado após cada transação. Vinte e dois meses depois, repentinamente ele caiu para o valor de 520. A causa foi que o valor atualizado foi truncado em vez de arredondado. O valor arredondado deveria ter sido 1.098,892. Com o emprego crescente de computadores para fazer operações de **trading** automático baseado nas flutuações de índices, uma queda repentina no índice causou um falso **crash** na bolsa, gerando grandes perdas financeiras.

3.3. Erros numéricos de computação

Basicamente, há três fontes de erros em computação numérica (LEONIDAS e CAMPOS, 1987).

- **Erros advindos dos dados e parâmetros iniciais**. A modelagem de fenômenos do mundo físico, raramente possui uma descrição exata do evento. Normalmente, são necessárias simplificações do mundo físico para chegar a um modelo matemático. Por sua vez, as medidas são obtidas por instrumentos de exatidão limitada, de modo que a incerteza dos parâmetros iniciais acarreta incerteza de resultados.
- **Erro de arredondamento**. Esse erro é causado pelo modo como os cálculos são efetuados, seja manualmente, ou obtidos por computador, porque se utiliza uma aritmética de exatidão finita, ou seja, leva-se em consideração somente um número finito de dígitos. Como exemplo, o cálculo de 1/3 resultaria em 0,3333 se for usado 4 (quatro) casas decimais após a vírgula.
- **Erro de truncamento**. São erros advindos da utilização de processos, que a princípio deveriam ser infinitos ou que envolvam muitos passos, para a determinação de um valor e que por razões práticas, utiliza apenas uma parte finita dele. Um exemplo de processo infinito é a determinação da função trigonométrica seno que corresponde a uma seqüência infinita de soma de produtos.

3.4. Matemática intervalar

A matemática intervalar foi uma das soluções propostas para reduzir o impacto dos erros de uma computação numérica e aumentar a exatidão dos cálculos matemáticos. Criada paralelamente por Moore e Sunaga na década de 50, ela usa intervalos fechados como forma de representar dados de entrada inexatos ou que não podem ser representados finitamente na máquina e que causam erros de truncamento ou arredondamento durante a execução do programa (SANTOS, 2001).

A matemática intervalar emprega um modelo computacional que reflete fielmente o controle e análise dos erros que ocorrem no processo computacional, e escolhe técnicas adequadas de programação para o desenvolvimento de aplicações científicas que minimizem erros numéricos nos resultados (DUTRA, 2000).

Na aritmética intervalar, as operações aritméticas são definidas sobre intervalos fechados de maneira a controlar o erro de aproximação, e o resultado obtido é outro intervalo que contém o resultado exato procurado.

Seja um intervalo fechado $[a_1, a_2]$, como uma aproximação de todos os números reais pertencentes a ele, a distância entre os extremos do intervalo e o número real aproximado não excede $(a_2-a_1)/2$, portanto o erro de aproximação máximo é de $(a_2-a_1)/2$.

Sejam dois números reais r_1 e r_2 e duas aproximações $[a_1, a_2]$ e $[b_1, b_2]$, respectivamente, uma operação aritmética $\oplus$ é tal que

$$r_1 \oplus r_2 \in [a_1, a_2] \oplus [b_1, b_2]$$

Portanto, o controle do erro de aproximação é feito diretamente pela aritmética dispensando a necessidade de um controle de erro externo.

Do ponto de vista computacional, a aritmética intervalar em vez de usar a abordagem tradicional de aproximar os reais por números de ponto flutuante mais próximos, emprega intervalos de números em ponto flutuante; $X = [x_1, x_2]$, onde os limites inferior x_1 e o superior x_2, são números de máquina, tais que $x_1 \leq x \leq x_2$ e x é o número exato desejado.

Um intervalo de máquina ou intervalo de ponto flutuante é um intervalo real em que os extremos são números de máquina. Todos os números entre x_1 e x_2, incluindo estes, pertencem ao intervalo X. Portanto, X contém os números de máquina dos extremos e todos os números reais deste intervalo.

Toda operação intervalar usa em seus cálculos intervalos $[y_1, y_2]$, tal que contenham o resultado exato procurado y, tendo como entrada um intervalo $[x_1, x_2]$ que contém o valor exato de entrada x. Esse princípio é conhecido como **princípio da corretude** (HICKEY, JU *et al.*, 2001; SANTIAGO, BEDREGAL *et al.*, 2006) e pode ser expresso logicamente pela seguinte expressão:

$$x \in [x_1, x_2] \Rightarrow f(x) \in F[x_1,x_2].$$

Neste caso, é dito que F é uma representação intervalar correta da função real f.

3.5. Aritmética de alta exatidão

Segundo (FERNANDES, 1997), a aritmética de alta exatidão possibilita que cálculos sejam efetuados com máxima exatidão, em que o resultado calculado difere do valor exato no máximo em um arredondamento.

São requisitos da aritmética de alta exatidão:

- O uso de arredondamento direcionado (para baixo e para cima),
- As quatro operações aritméticas com alta exatidão e
- O produto escalar ótimo.

Aliada, a matemática intervalar, os resultados são confiáveis e com máxima exatidão, onde o resultado está contido em um intervalo cujos extremos diferem por apenas um arredondamento do valor real, pos os cálculos intermediários são feitos em registradores especiais, de forma a simular a operação nos reais, sendo o arredondamento feito só no final, onde cada extremo é aplicado o arredondamento direcionado para baixo e para cima.

Um exemplo de biblioteca intervalar que possui máxima exatidão é Libavi, descrita em (FERNANDES, 1997).

3.6. Arredondamento direcionado

Para o funcionamento da aritmética intervalar é necessário o emprego de arredondamentos direcionados, em especial o arredondamento para cima e o para baixo.

Arredondamento para cima. É a função que aproxima o número real x para o menor número de máquina, maior que ou igual ao número real x. Este tipo de arredondamento é definido por

$$u(x)= \min\{y \in F \mid y >= x \in R\}, \text{ para todo } x \in R$$

Arredondamento para baixo. É a função que aproxima o número real x para o maior número de máquina, menor que o número real x, definido por

$$d(x)= \max\{y \in F \mid y <= x \in R\}, \text{ para todo } x \in R$$

3.6.1. Arredondamento direcionado e inflacionamento

Uma exigência da matemática intervalar é que as operações sejam realizadas com arredondamentos direcionados nos extremos dos intervalos para cima no limite superior e para baixo no limite inferior. Isso garante que a operação obedece ao **princípio da corretude**.

Como algumas linguagens de programação não dispõem de arredondamentos direcionados, muitas vezes a solução adotada é **inflacionar** os extremos sempre que uma operação seja realizada. Embora atenda o **principio da corretude**, essa solução tem a desvantagem, em relação ao arredondamento direcionado, que em alguns casos provoca o alargamento desnecessário dos limites do intervalo. O **inflacionamento** de um intervalo de

pontos flutuantes $[a_1, a_2]$ é definido como o intervalo cujo limite inferior é o número real pertencente ao conjunto de números representáveis no computador que precede a_1 e o limite superior é o número real pertencente ao conjunto dos intervalos da máquina que sucede a_2, isto é,

$$\text{Inflacionamento } (a_1, a_2) = [\text{prevfp}(a_1), \text{nextfp}(a_2)],$$

Onde prevfp(x) é o maior número em ponto flutuante representável que é menor que, e diferente de x e nextfp(x) é o menor número em ponto flutuante representável que é maior que e diferente de x.

O presente trabalho, como será visto nos próximos capítulos, teve como motivação principal a criação de uma solução para a falta de arredondamento direcionado na linguagem Java.

Capítulo 4

Linguagens XSC

4.1. Introdução

Desde a década de sessenta são desenvolvidas pesquisas e elaborados softwares com o objetivo de possibilitar que os computadores suportem uma aritmética mais poderosa que a aritmética de ponto flutuante. Dentre elas, destacam-se as extensões XSC[2] para algumas linguagens de programação que foram criadas para auxiliar no desenvolvimento de softwares numéricos que exijam resultados verificados automaticamente e com alta exatidão (HOFSCHUSTER, 2004).

Entre 1976 e 1979, as **Universidades de Karlsruhe e Kaiserslautern** cooperaram para desenvolver uma extensão da linguagem de programação Pascal conhecida como Pascal-SC. Nos anos seguintes, em cooperação com a IBM, uma versão SC da linguagem de programação Fortran 77 foi criada.

As linguagens SC (**Scientific Computing**) constituíram a primeira geração das linguagens cientificas, e o nome SC foi escolhido devido a sua capacidade limitada, pois as mesmas tinham a desvantagem de só estarem disponíveis para uma variedade limitada de computadores (KEARFOTT, 1996). As linguagens Pascal-SC e Fortran-SC foram as primeiras a terem suporte intervalar e tipos de dados definidos pelo usuário. Essas linguagens foram disponibilizadas inicialmente para o sistema operacional CP/M e posteriormente para o sistema operacional DOS.

Com o aparecimento de novas linguagens de programação, criou-se uma nova geração de linguagens com suporte à computação intervalar. Essas linguagens receberam a extensão XSC (**Extended Scientific Computing**) e tiveram como característica o aumento da exatidão e o suporte a operações intervalares.

As extensões XSC, desenvolvidas na **Universidade de Wuppertal,** ampliaram as linguagens C, Pascal e Fortran. Como o próprio nome sugere, elas estendem as linguagens para as quais foram criadas com características necessárias ao desenvolvimento de software numérico de alta exatidão.

[2] http://www.xsc.de/

Os compiladores dessas linguagens são escritos na linguagem de programação C ou C++ e essas extensões têm como principais características:

- Sobrecarga de operadores,
- Agrupamento das funções em módulos específicos,
- Vetores dinâmicos,
- Arredondamento direcionado,
- Tipos numéricos primitivos simples e estendidos, como real, complexo e intervalo,
- Operadores aritméticos de maior exatidão para os tipos numéricos e
- Funções elementares de maior exatidão para os tipos numéricos.

4.2. C-XSC

A extensão C-XSC[3] é constituída de um pacote de classes codificadas em C++, e contém um grande número de tipos e operadores numéricos. Esta extensão está disponível para diversas arquiteturas e contém mais de 1.500 operadores de alta exatidão predefinidos.

A extensão possui as seguintes características:

- C-XSC fornece os tipos de dados simples real (**real**), complexo (**complex**), intervalo real (**interval**) e intervalo complexo (**cinterval**),
- O arredondamento dos operadores aritméticos podem ser controlados,
- Cada tipo possui um conjunto de operações apropriadas, operadores relacionais e funções matemáticas padrão,
- Todos os operadores aritméticos predefinidos retornam resultados com uma exatidão de pelo menos uma unidade na última casa decimal, obtendo a máxima exatidão na computação cientifica de seus tipos,
- Suporte a faixas (**subarrays**) de vetores e matrizes, o qual permite operações matriciais sobre somente parte da matriz, por exemplo, a expressão **m[Col(1)]=9**, faz com que todos os elementos da coluna 1 da matriz m assumam o valor 9. Operadores primitivos também podem usar subarrays como operandos. C-XSC fornece uma notação especial para manipular subarrays de vetores e matrizes,
- Tipos com precisão definida após a vírgula,

[3] C-XSC pode ser adquirida gratuitamente em http://www.math.uni-wuppertal.de

- Operadores aritméticos primitivos com grande exatidão. Expressões aritméticas são avaliadas com exatidão matematicamente alta e garantida,
- Funções padrão com grande exatidão,
- C-XSC fornece controle de arredondamento durante operações de entrada e saída para todos os tipos de dado. Isso permite especificar o número de casas decimais que serão apresentadas para o valor a ser impresso, por exemplo, a expressão **cout << SetPrecision(15,10)**, especifica que os próximos números serão impressos com 10 casas decimais após a vírgula, ocupando o string representado o numeral completo no mínimo 15 espaços,
- Aritmética dinâmica de múltipla precisão. Além dos tipos **real** e **interval**, C-XSC possui os tipos dinâmicos de maior precisão denominados reais longos (**l_real**) e intervalos longos (**l_interval**), assim como também são implementados vetores dinâmicos correspondentes a matrizes, incluindo operações vetoriais.
- Rotinas para resolução de problemas. C-XSC contem uma biblioteca com uma coleção de rotinas para problemas corriqueiros de análise numérica e que precisam de resultados de alta exatidão. Entre os problemas cobertos destacam-se avaliação de zeros de polinômios, matriz inversa, sistemas lineares, tranformadas de Fourier, zeros de equações não-lineares, sistemas de equações não lineares e problemas com equações diferenciais ordinárias.
- C-XSC usa a abordagem orientada a objetos e disponibiliza ao programador um grande número de operadores e tipos predefinidos de dados. Tem-se a possibilidade de sobrecarregar os operadores disponíveis em C++, permitindo expressar a adição de variáveis em ponto flutuante por meio da notação infixa, por exemplo, **real_a=real_b+real_c**, a qual é preferível a chamada explícita de funções, como por exemplo, **add(&real_a,&real_b,&real_c)))**.

4.3. Pascal-XSC

A extensão Pascal-XSC [4] é baseada no padrão ISO Pascal e dispõe de um compilador Pascal escrito em C portável para múltiplas plataformas. Essa extensão suporta as principais características necessárias para uma extensão XSC e disponibiliza módulos para resolução de problemas numéricos comuns, tais como:

[4] Pode ser adquirida, gratuitamente, em http://www.math.uni-wuppertal.de/org/WRST/xsc/pxsc.html

- Conceito de operador universal definido pelo usuário,
- Funções e operadores com tipo de resultado arbitrário,
- Sobrecarga de procedimentos,
- Rotinas de leitura e escrita,
- Conceito de módulo,
- Arrays dinâmicos e acesso a subarrays,
- Conceito de string,
- Arredondamento direcionado,
- Produto escalar ótimo,
- Tipo padrão dotprecision e tipos aritméticos adicionais,
- Aritmética de alta-exatidao para todos os tipos,
- Sistema de equações lineares e não lineares,
- Inversão de matriz,
- Autovalores e Autovetores,
- Avaliação de expressões aritméticas,
- Avaliação de polinômios e busca de zeros de polinômios,
- Quadratura numérica,
- Equações diferenciais,
- Equações integrais,
- Diferenciação automática,
- Otimização,
- Aritméticas intervalar, complexa, intervalar complexa e
- Operações aritméticas sobre vetores e matrizes.

Esta linguagem pode ser aplicada na solução dos mais diversos problemas em que se exigem alta exatidão e grande confiabilidade nos resultados obtidos, como cálculos de múltipla precisão com verificação de resultado, computação cientifica e problemas que exijam soluções confiáveis.

4.4. INTLIB90

Este é um pacote criado para linguagem Fortran 77, e visa suportar operações matemáticas elementares. Segundo (KEARFOTT, 1996), suas rotinas são bem documentadas e tidas como de qualidade e portabilidade. As rotinas em Intlib90 são consideradas

significativamente legíveis para os usuários, sendo precisas e sem grande número de funções supérfluas. Quando de seu desenvolvimento, não foram incluídas rotinas que gerassem ambigüidade de interpretação matemática.

A INTLIB é organizada em módulos:

- Rotinas aritméticas elementares,
- Rotinas de funções padrão,
- Rotinas de funções úteis,
- Rotinas de impressão de erro e
- Rotinas de definição de constantes matemáticas de programas que são usadas para inicializar constantes globais e parâmetros.

O primeiro módulo define as quatro operações elementares em tipos de dados intervalares. As operações mistas são permitidas somente entre intervalos e números com dupla precisão, ou entre intervalos e números inteiros. INTLIB suporta operações com números em ponto flutuante com dupla precisão e intervalos.

4.5. Libavi

Libavi é uma biblioteca de aritmética vetorial intervalar, baseada no Fortran 90, desenvolvida no Brasil, no Centro de Supercomputação (CESUP/RS), para operar em um computador Cray. Esta biblioteca providencia rotinas intervalares e álgebra linear usando intervalos (DIVERIO, 1995; HÖLBIG, SAGULA *et al.*, 1996; HÖLBIG, 2004). Ela foi projetada para viabilizar o uso da matemática intervalar em supercomputadores para a solução de problemas físicos, químicos e das engenharias que necessitem de alta exatidão.

Ela é composta por rotinas intervalares organizadas em quatro módulos:

1. Módulo **básico** que inclui o arquivo que contem a definição de intervalo reais e complexos. Nesse modulo, são providas todas as operações entre intervalos reais. Essas operações, por sua vez, servem de base para todos os demais módulos,
2. Módulo de **intervalos complexos** contendo rotinas para tratamento de dados do tipo intervalo complexo,
3. Módulo que provê todas as operações entre vetores de intervalos, matrizes de intervalos e rotinas de diferentes tipos de dados com vetores e matrizes de intervalos e

4. Módulo que provê **operações aritméticas compostas** existentes em outras bibliotecas.

Libavi foi desenvolvida com a finalidade de explorar o alto desempenho dos computadores Cray no cálculo com a aritmética intervalar.

Uma característica importante dessa biblioteca é que os números em ponto flutuante não são representados de acordo com o padrão IEEE-754.

Na versão inicial de Libavi não havia a implementação de arredondamentos direcionados o que tinha como efeito colateral que o diâmetro resultante era maior do que deveria. Posteriormente, em (FERNANDES, 1997) foi adicionado um núcleo de aritmética de alta exatidão que corrigiu esse problema e adicionou a máxima exatidão as operações realizadas.

4.6. INTLAB

IntLab[5] é uma biblioteca escrita para o software MatLab[6] destinada a auto-validação de algoritmos e possui as seguintes características:

- Aritmética intervalar para os tipos de dado real e complexo, incluindo vetores e matrizes,
- Aritmética intervalar para matrizes esparsas do tipo real e complexo,
- Funções padrões para tratamento de intervalos reais e complexos,
- Tratamento rigoroso de entrada e saída,
- Aritmética intervalar de precisão múltipla com limites de erro,
- Diferenciação automática (modo direto, computações vetorizadas),
- Suporte a polinômios univariados e multivariados,
- Suporte a resolução de problemas para sistemas de equações lineares e não lineares,
- Suporte a autovalores,
- O código é completamente escrito em Matlab, o que permite portabilidade entre sistemas operacionais para o qual esse programa esteja disponível,

[5] A biblioteca Intlab está disponível no site http://www.ti3.tu-harburg.de/rump/intlab/

[6] http://www.mathworks.com/products/matlab/

- A aritmética IEEE-754 e o controle do modo de arredondamento é suportado desde a versão 5.3 do Matlab e estão disponíveis para outras plataformas além do Windows.
- Usa extensivamente as rotinas da biblioteca **BLAS** (Basic Linear Álgebra System)[7] que proporciona operações matriciais rápidas e aproveita melhor arquiteturas paralelas.

4.7. Java-XSC

O sucesso das linguagens XSC motivou o desenvolvimento de pesquisas que dessem suporte à matemática intervalar a linguagens mais modernas, como Java. O interesse em Java reside principalmente na questão da neutralidade dessa linguagem em relação a plataforma. Outras linguagens, como C ou Pascal, exigem a recompilação do código fonte toda vez que se precisa migrar de plataforma. No caso de Java, o mesmo código já compilado pode executar em mais de uma plataforma sem a necessidade de alterações do código fonte e recompilação.

Java consegue esse prodígio usando uma máquina virtual independente de plataforma, a qual permite que um sistema desenvolvido, por exemplo, em um computador Intel com Microsoft Windows possa executar sem modificações em uma máquina Sparc usando o sistema operacional Linux.

A linguagem Java tem como características:

- Ser **orientada a objetos**,
- Ser **distribuída**. Java foi uma linguagem projetada desde o inicio para computação distribuída em uma rede heterogênea de computadores. Este é um fato que justifica o grande apelo de Java com o advento da Internet,
- Ser ao mesmo tempo **compilada** e **interpretada**. Os programas fonte em Java (extensão **.java**) são compilados para um formato binário de código conhecido como **bytecode** (extensão **.class**) que é independente de plataforma. O bytecode será então executado em outra máquina de forma interpretada.
- Dispor de compiladores os quais identificam pontos do programa mais executados (**hotspots**) e compilam para código nativo, o que acelera a execução do programa e os torna quase tão rápidos quanto programas escritos na linguagem C/C++.

[7] A biblioteca BLAS está disponível no site http://www.netlib.org/blas/

- Ser uma linguagem **neutra** em relação à arquitetura, o que garante **portabilidade** entre plataformas,
- Ser **multitarefa**. Java suporta o conceito de **threads** permitindo a execução simultânea de diversos segmentos de código,
- Ser **dinâmica**. A ligação (**linking**) do programa com as bibliotecas de terceiros são executadas dinamicamente, ao contrario de outras linguagens que precisam ser ligadas estaticamente,
- Ser **segura**. Devido à característica distribuída de Java, ela foi pensada para incorporar diversos recursos de segurança, raramente encontrada em outras linguagens. Podem se especificar quais permissões de acesso o programa em Java terá em relação à rede ou disco rígido,
- Ser **simples** quando comparada a C++. A linguagem Java apesar de ser derivada da linguagem C++, evitou características problemáticas dessa linguagem que causavam confusão e falta de legibilidade como, por exemplo, herança múltipla,
- Ser **compacta**. A maioria das máquinas virtuais e programas em Java ocupam pouco espaço de memória. Isso permite que Java possa funcionar em máquinas com poucos recursos como celulares e palmtops,
- Ter **alto desempenho**. O emprego de compiladores **just in time (JIT)**, permite a compilação de partes criticas do código (**hotspots**) e alcançar velocidades de execução próximas a linguagens compiladas como C.

Devido as essas vantagens, foi criada a biblioteca Java-XSC (DUTRA, 2000; BEDREGAL e DUTRA, 2006) no Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte, que é uma extensão de suporte à computabilidade de alta exatidão para a linguagem Java.

Java-XSC foi desenvolvida para prover o suporte a intervalos na linguagem Java e é composta de seis módulos de acordo com a natureza de suas operações:

- **Operações básicas**. Contém a definição do intervalo real e as operações aritméticas de adição, subtração, multiplicação e divisão que servem de base aos demais módulos,
- **Funções entre conjuntos**. São métodos disponíveis para operações entre conjuntos como intersecções, união e união convexa entre intervalos,

- **Funções geométricas**. Funções que calculam particularidades geométricas dos intervalos como distancia, diâmetro e ponto médio.
- **Funções elementares**. Funções básicas como valor absoluto, raiz quadrada, exponenciação, potenciação, logaritmo, etc.
- **Funções trigonométricas**. Funções trigonométricas, hiperbólicas e inversas, utilizadas em problemas geométricos.
- **Funções de conversão entre tipos e definição de constantes**. Neste módulo, estão funções de conversão entre os tipos reais, intervalos e inteiros. Neste módulo também estão disponíveis funções para recuperar os limites inferior e superior de um intervalo.

Embora a biblioteca Java-XSC se encontre em estágio avançado, ainda necessita de esforços para tratar do arredondamento dos limites dos intervalos. Da mesma forma que a biblioteca Libavi, ela inflaciona os limites do intervalo após cada operação. Esse inflacionamento incondicional é necessário uma vez que, atualmente, a linguagem Java não provê suporte nativo ao arredondamento direcionado nas operações de ponto flutuante e a matemática intervalar necessita desse arredondamento direcionado para garantir a corretude dos resultados.

Java provê somente o arredondamento ao número par mais próximo (**rounding nearest even**) apesar da biblioteca Java-XSC necessitar dos arredondamentos para cima (**rounding up**) e para baixo (**rounding down**) para aumento da exatidão das operações intervalares. Dessa forma, mesmo que não haja necessidade, ocorre um inflacionamento incondicional dos limites inferior e superior dos intervalos resultantes das operações. A principal motivação para o presente trabalho foi então construir uma biblioteca de ponto flutuante que permitisse a biblioteca Java-XSC executar operações aritméticas usando controle direcionado de arredondamento, como será visto nos próximos capítulos.

4.8. Resumo comparativo entre as bibliotecas

Característica	C-XSC	Pascal-XSC	Intlib90	Libavi	IntLab	Java-XSC
Linguagem	C++	Pascal	Fortran	Fortran	Matlab	Java
Inflacionamento	Não	Não	Sim	Sim	Não	Sim
Orientação a objetos	Sim	Sim	Não	Não	Não	Sim

Arredondamento direcionado	Sim	Sim	Não	Não	Sim	Não
Sobrecarga de operadores	Sim	Sim	Não	Não	Sim	Não
Rotinas para resolução de problemas avançados	Sim	Sim	Sim	Sim	Sim	Não

Capítulo 5

Bibliotecas de ponto flutuante

5.1. Software existente e trabalhos relacionados

Durante o desenvolvimento desse trabalho, além da implementação nativa da linguagem de programação Java para ponto flutuante, foram encontradas poucas bibliotecas alternativas para a linguagem, as quais serão discutidas neste capítulo.

Como será visto, nenhuma delas, inclusive a nativa, implementa todas as características exigidas e especificadas pelo padrão IEEE-754. Como dito anteriormente, o suporte nativo de Java não provê todos os modos de arredondamento exigidos no padrão IEEE-754.

Neste capítulo, será descrita uma lista de alguns pacotes de software disponíveis que habilitam, mesmo que parcialmente, o suporte ao padrão IEEE-754 para a linguagem Java. Além disso, embora seja desejada uma solução desenvolvida completamente em Java para o problema do arredondamento direcionado de Java-XSC, também será descrita a biblioteca Softfloat, a qual é escrita usando a linguagem de programação C.

5.2. Ponto Flutuante em C

5.2.1. Softfloat

SoftFloat é uma biblioteca de ponto flutuante escrita usando a linguagem C e criada por John R. Hauser na Universidade de Berkeley. A versão original foi criada no Instituto Internacional de Ciências da Computação como parte do projeto Spert, um sistema de microprocessamento vetorial projetado para acelerar redes neurais, multimídia e outras tarefas de processamento de sinal digital (HAUSER).

SoftFloat foi escrita usando a linguagem de programação **ISO/ANSI C** e foi compilada e testada usando o compilador **GNU C**[8] (**gcc**). Essa biblioteca está preparada para trabalhar com as plataformas **Intel** 386 (**little endian**) ou **Sun Sparc** (**big endian**), podendo ainda ser

[8] http://gcc.gnu.org

configurada para outras plataformas que suportem o compilador **GNU C**. De acordo com o criador da biblioteca, esta atende fielmente ao padrão IEEE-754 e suporta todos os modos de arredondamento especificados do padrão.

A biblioteca não usa o suporte nativo a ponto flutuante da máquina onde executa. Em vez disso, executa totalmente por software a maioria dos formatos comuns de ponto flutuante: precisão simples (32 bits de largura), precisão dupla (64 bits de largura), precisão dupla estendida (80 bits de largura), e precisão quádrupla (128 bits de largura). Softfloat somente suporta precisão dupla estendida e precisão quádrupla se o compilador C suportar inteiros de 64 bits de largura.

Todas as funções aritméticas do padrão IEEE-754 são suportadas pela biblioteca, com exceção das funções de conversão para decimal. Também, todos os **flags** de sinalização e valores especiais como NaN são suportados.

Somente as operações básicas estão disponíveis. Funções matemáticas avançadas como trigonométricas e exponenciais não estão disponíveis.

Apesar de escrita em uma linguagem diferente de Java, detectamos que Softfloat poderia atender ao requisito de controle direcionado do arredondamento para Java-XSC se pudesse ser convertida para Java ou se fosse possível empregar uma solução híbrida em que Java acessasse C nativamente usando JNI (**Java Native Interface**). Como o objetivo desse trabalho foi encontrar uma solução inteiramente escrita em Java, foi descartada a segunda solução.

5.3. Ponto flutuante em Java

5.3.1. Portabilidade

A linguagem Java foi projetada tendo em vista, dentre outras coisas, a **portabilidade** e a **independência** de plataforma. Programas Java não são traduzidos para a linguagem de máquina nativa como outras linguagens de programação, em vez disso, usa uma representação intermediária (**bytecode**), a qual é interpretada pela máquina virtual Java (**JVM – Java Virtual Machine**). Embora seja uma linguagem em parte interpretada, Java possuiu um desempenho próximo daquelas de linguagens compiladas, principalmente graças aos compiladores mais recentes com suporte a compilação nativa just in time (JIT).

Para garantir a **portabilidade** entre plataformas, Java define exatamente o tamanho de cada tipo e formato de dado numérico, possibilitando o processamento distribuído entre

plataformas diferentes. Outras linguagens, como C e Pascal deixam a critério do compilador a definição da precisão ou do tamanho em bytes da representação interna de números inteiros e de ponto flutuante, o que torna complicada qualquer tentativa de programação portável e a comunicação entre componentes de um sistema distribuído.

5.3.2. Lacunas na implementação do padrão IEEE-754

Apesar da grande portabilidade de programas escritos em Java, existem alguns problemas com a linguagem, em relação ao seu suporte a aritmética de ponto flutuante que não segue fielmente o padrão IEEE-754 (KAHAN, 1998):

- Embora IEEE-754 especifique cinco tipos de flags de exceção (operação inválida, **overflow**, divisão-por-zero, **underflow** e resultado inexato), a especificação de Java para ponto flutuante não usa **flags** de exceção. Em Java, um **overflow** resulta um infinito sinalizado, um **underflow** resulta em um zero sinalizado, e operações matematicamente indefinidas resultam em um **Not a Number (NaN)**, mas nenhuma exceção é levantada.
- IEEE-754 estabelece que todo o hardware/software deveria fornecer ao programador um modo para especificar dinamicamente um dos quatro modos de arredondamento: **arredondamento para o mais próximo** (padrão), **arredondamento para 0**, **arredondamento para cima** e **arredondamento para baixo**. Porém, Java só suporta o modo de arredondamento para o mais próximo. Portanto, para aplicações, como Java-XSC, que requeiram outros modos de arredondamento, o suporte nativo de Java para ponto flutuante não é totalmente adequado.

Também comenta (DOERDELEIN, 2005) que a **Java Language Specification** (JLS) determina que Java implemente a IEEE-754, excluindo-se apenas alguns itens:

- Exceções numéricas e **NaN** com sinal. Essas são limitações importantes, porque não há alternativas precisas para tratar **overflow**, **underflow** ou operação inexata,
- **Opções variadas de arredondamento** para a conversão de números em ponto flutuante para inteiros. A partir da versão 5.0 da **Java Standard Edition** foi adicionada a classe **BigDecimal** com suporte a diversos tipos de arredondamento, sendo possível fazer a conversão entre os tipos de ponto flutuante, BigDecimal e inteiros,

- **Precisões estendidas maiores que dupla precisão**. Esta é uma parte opcional da norma e optou-se por não se implementar, pois não seria portável e eficiente, uma vez que o suporte as diversas plataformas deve ser universal,
- Predicados relacionais sobre valores não-ordenados, exceto != (diferente). Mas é possível escrever código equivalente aos operadores que faltam.

Pelos motivos elencados, Java não pode afirmar que implementa a norma completamente, mas segundo informação do site da IEEE (ANSI/IEEE), não há nenhuma linguagem que faça isso, e a lista de linguagens "quase perfeitas" é minúscula. No site, que não é atualizado desde 2001, são propostas duas extensões à linguagem Java denominadas Borneo e RealJava, mas ambas anteriores a versão Java Standard Edition 2.

Na verdade, não há uma classificação oficial das linguagens que melhor implementam esta norma. Na linguagem C, o padrão ANSI C99 tem aderência à IEEE-754, mas o C++ não suporta esse padrão, e de acordo com (DOERDELEIN, 2005) isso ainda não é implementado corretamente por nenhum compilador popular (como GNU, Microsoft ou Borland).

Em Java, todas as funções geram resultados corretos e dentro da precisão especificada, para todos os valores no seu domínio matemático. Assim, Math.acos() só é válida para o intervalo de 0 a π, pois, para outros valores, a função matemática de arco co-seno não é definida, e a função Java retorna um NaN. Por outro lado, Math.cos() é legal para qualquer valor, sendo que os resultados se repetem a cada 2π.

Segundo (DOERDELEIN, 2005), apesar de óbvio, não é o que acontece necessariamente em todas as linguagens de programação. Como exemplo, ele cita que matematicamente, $\sin(\pi) = 0$, mas o argumento Math.PI é uma aproximação com 17 casas decimais, de forma que mesmo o seno matematicamente exato deste valor é um número próximo a zero, mas não é zero. Em Java, este resultado é $1.2246479914735323 \times 10^{-16}$ preciso até o último dígito. Já em linguagens como Microsoft C/C++ e Visual Basic, o resultado será de $1.22460635382238 \times 10^{-16}$, o que já apresenta um erro a partir da 5ª casa decimal.

Um outro exemplo: a instrução FSIN dos chips Intel só é exata na faixa de $[-\pi/4..+\pi/4]$. Para outros valores, é preciso usar um algoritmo de redução de argumentos, onde qualquer valor será mapeado para a faixa ideal, o que é possível porque a função seno é periódica. Mas a unidade de ponto flutuante da Intel faz a redução de argumentos de forma imprecisa (MICROSYSTEMS), pois os algoritmos que permitem implementar isso de

maneira exata e eficiente no hardware só foram inventados em 1982, após a criação do chip 8087.

Ao que parece, a Intel nunca atualizou seus chips para usar os algoritmos aperfeiçoados e por isso, em vez de Java usar, no processador aritmético da Intel, a instrução de hardware FSIN, ela calcula a função seno por software. O seno não é a única função matemática com tais dificuldades, nem são somente os chips da Intel que não implementam perfeitamente todas as funções da IEEE-754.

Dependendo da plataforma, Java pode usar a biblioteca da Sun, escrita em C, chamada **fdlibm** para executar diferentes funções matemáticas. Mas como as funções da **fdlibm** são executadas por software, elas serão mais lentas que as instruções nativas das **unidades de ponto flutuante** (**FPU**). Os compiladores tentam evitar a **fdlibm** sempre que possível, usando as instruções da FPU somente quando estas instruções produzirem a exatidão e todos os demais requisitos exigidos pelo Java. Similar ao uso que Java faz da fdlibm para algumas funções numéricas, a biblioteca desenvolvida em nosso trabalho também usa software para emular ponto flutuante.

Quanto a questão da portabilidade, Java possui uma regra de reprodutibilidade bit por bit, mas que tem um efeito nocivo sobre o desempenho e a exatidão do cálculo de ponto flutuante. Um problema é que algumas CPUs utilizam registradores de ponto flutuante com precisão maior que os 64 bits exigidos pelo tipo de dupla precisão. Isso é especialmente verdadeiro para os processadores Intel que usam 80 bits internamente para os cálculos. Isso significa que, mesmo que em uma operação os valores de entrada sejam lidos como variáveis de 64 bits, o processador Intel fará o cálculo internamente com 80 bits e produzirá um resultado de 80 bits. Se a e b forem números muito grandes, uma multiplicação produzirá um expoente maior que o máximo permitido e isso exigirá uma renormalização forçando a perda de alguns bits. Mas no Intel, que tem um significando maior que o normal, até 16 bits que deveriam ser perdidos serão preservados, enquanto os valores estiverem presentes nos registradores de ponto flutuante da CPU. Somente quando o resultado de um cálculo for extraído desses registradores e armazenado em uma variável de memória, a conversão para os 64 bits ocorrerá e quaisquer bits de precisão extra serão perdidos.

Segundo a especificação Java 1.0, esta conversão para 64 bits deveria ocorrer a cada operação individual. Mesmo para as variáveis temporárias, que não são necessariamente associadas a nenhum endereço de memória, o compilador deveria ser forçado a gerar código para descarregar os resultados intermediários para memória e recarregá-los logo em seguida (**store-reload**), onde o registrador de ponto flutuante será gravado em uma posição de 64 bits

da pilha de máquina e recarregado no registrador, somente para descartar os bits extras. Este é o único modo de garantir a reprodutibilidade bit por bit nesta plataforma, mas tem o custo de reduzir a velocidade de execução.

A solução criada na segunda versão do Java foi a introdução dos conceitos de matemática relaxada (**loose**) e estrita (**strict**). Somente na matemática estrita, a reprodutibilidade estrita bit por bit é obrigatória. Para trabalhar no modo estrito, deve-se aplicar o modificador **strictfp** aos métodos ou classes cujos cálculos precisem ser estritos.

Por padrão, na ausência deste modificador, o otimizador de código pode evitar as operações de **store-reload** para resultados temporários; os bits extras serão descartados somente quando os resultados finais forem gravados.

Se por um lado, o resultado fica mais preciso, por outro se perde em reprodutibilidade bit por bit. Além do **strictfp**, a **API** do Java fornece duas versões de todas as funções matemáticas: versões relaxadas no pacote **java.lang.Math** e versões estritas no pacote **java.lang.StrictMath**.

Classe e métodos strictfp devem evitar a primeira opção, invocando somente os métodos StrictMath. Também devem evitar a invocação de qualquer método não estrito para fazer parte dos seus cálculos. Pode-se concluir que a matemática estrita é excelente quando são necessários cálculos cujos resultados sejam previsíveis até o último dígito (ou bit) em qualquer plataforma.

Ainda, segundo (DOERDELEIN, 2005), uma ressalva importante é que a única qualidade exclusiva do **strictfp** é a reprodutibilidade bit a bit. Todos os demais critérios: garantias de exatidão, semi-monotonicidade, suporte a todos os casos especiais da IEEE-754, valem também para a matemática relaxada.

5.3.3. Tipos nativos de ponto flutuante

Java, como outras linguagens de programação, possui suporte nativo para tipos de ponto flutuante:

- Tipo “float”. Um tipo de ponto flutuante de precisão simples IEEE-754 com 32-bits de largura. O maior literal positivo finito representável é $3.40282347 \times 10^{38}$ e o menor literal representável em ponto flutuante positivo diferente de zero é $1.40239846 \times 10^{-45}$.
- Tipo “double”. Um tipo de ponto flutuante de precisão dupla IEEE-754 com 64-bits de largura. O maior finito positivo representável em double é

aproximadamente $1.79769313486231570 \times 10^{308}$ e o menor literal positivo diferente de zero representável é aproximadamente $4.94065645841246544 \times 10^{-324}$.

Os tipos nativos de ponto flutuante em Java têm as seguintes características:

- Um programa pode representar números infinitos usando expressões constantes como 1/0 ou as constantes predefinidas **POSITIVE_INFINITY** e **NEGATIVE_INFINITY**, definidas nas classes Float e Double. Exemplo, **double variavel = Double.POSITIVE_INFINITY**,
- Um erro em tempo de compilação acontecerá se um ponto-flutuante literal diferente de zero é pequeno demais, de tal forma que, quando ocorrer conversão arredondada para sua representação interna, este se tornará zero. Um erro em tempo de compilação não acontecerá se um ponto flutuante diferente de zero tiver um valor pequeno que, durante a conversão arredondada para sua representação interna, se tornar um número denormalizado diferente de zero. Quando os bits do campo de expoente forem todos iguais a zero, o número será interpretado como denormalizado.

5.3.4. Tipo float

O primeiro tipo de ponto flutuante em Java que será descrito é o tipo float. Ele corresponde ao formato de precisão simples do padrão IEEE-754:

- O bit 31, isto é o bit selecionado pela máscara 0x80000000, representa o sinal do número em ponto flutuante.
- Os bits 30 a 23, os quais são selecionados pela máscara 0x7f800000 representam o expoente.
- Os bits 22 a 0, selecionados pela máscara 0x007fffff representam o significando.
- Se o argumento é um positivo infinito, seu valor interno será representado pelo valor 0x7f800000.
- Se o argumento é um negativo infinito, seu valor interno será representado por 0xff800000.
- Se o argumento é um NaN, seu valor interno será representado por 0x7fc00000.

5.3.5. Tipo double

O tipo **double** tem como maior valor representável o valor $(2-2^{-52}) \times 2^{1023}$ e o menor valor representável é de 2^{-1074}.

O tipo **double** corresponde ao formato de precisão dupla do padrão IEEE-754:

- O bit 63, isto é o bit selecionado pela máscara 0x8000000000000000L, representa o sinal do número em ponto flutuante.
- Os bits 62 a 52, os quais são selecionados pela máscara 0x7ff0000000000000L representam o expoente.
- Os bits 51 a 0, selecionados pela máscara 0x000fffffffffffffL representam o significando.
- Se o argumento é um positivo infinito, seu valor interno é representado por 0x7ff0000000000000L.
- Se o argumento é um negativo infinito, seu valor interno é representado por 0xfff0000000000000L.
- Se o argumento é um NaN, seu valor interno é representado por 0x7ff8000000000000L.

Tanto o tipo float quanto o tipo double não suportam arredondamento direcionado.

5.3.6. BigDecimal

A partir da versão Java **Standard Edition** 5.0, um novo tipo de ponto flutuante foi adicionado a linguagem Java, a classe **BigDecimal**. Ela tem por base a classe **BigInteger**, a qual representa números inteiros de exatidão ilimitada. A classe **BigDecimal** é formada por um **BigInteger** e um expoente, que é uma potência de 10. O expoente indica quais dígitos são da parte inteira e quais são da parte decimal. A classe **BigDecimal** segue a norma **ANSI** X3.274-1996 e pode representar qualquer número racional. Esta classe combina uma magnitude, representada internamente por um **BigInteger** com uma escala de 32 bits.

A documentação costuma usar a notação **[magnitude, escala]**, onde o valor do número decimal é o produto da magnitude $\times 10^{-escala}$. Precisão é o número de dígitos significativos. Escala é o número de dígitos preservados, antes ou após o separador decimal, com ou sem necessidade.

Antes de ser realizada qualquer operação, os operandos **BigDecimal** são normalizados, então a operação é feita diretamente sobre os valores **BigInteger** armazenados. Em seguida é calculado o expoente do resultado.

A classe BigDecimal tem as seguintes características:

- Especifica o tipo de arredondamento desejado.
- Suporta apenas as operações aritméticas (+,-,x,/).
- Funções trigonométricas, transcendentais ou logarítmicas não são suportadas.
- Na soma e subtração, a escala do resultado é a maior escala entre os parâmetros.
- Na multiplicação, a escala resultante é igual a soma das escalas do multiplicador e do multiplicando.
- Na divisão, a escala é resultante da diferença entre as escalas do dividendo e do divisor. No caso da divisão, a escala preferencial só é respeitada se a divisão for exata.
- Sem levar em conta as divisões inexatas, as regras de escala preferencial produzem um comportamento intuitivo e previsível: a escala resultante é a menor possível para armazenar o resultado no pior caso. No entanto é possível especificar uma escala diversa da preferencial.
- É mais lento que qualquer outro tipo numérico. Devido à exatidão ilimitada obtida internamente por BigInteger, o qual usa um vetor de inteiros para armazenar o número, todas as operações aritméticas precisam fazer laços sobre vetores.
- Cada instância de objeto BigDecimal não pode ser mudada, de forma que um laço envolvendo vários cálculos alocará muitos objetos, aumentando o trabalho da coleta automática de lixo.

Como exemplo, no código abaixo, a cada iteração do laço o valor anterior da variável **a** será descartado e será associada a ela um novo objeto BigDecimal:

```
BigDecimal a = new BigDecimal(20);
BigDecimal b = new BigDecimal(1);
for(int i=1; i<100 ;i++){
        a = a.add(b);
}
```

Apesar das desvantagens de BigDecimal, ele inova em prover suporte ao arredondamento direcionado, característica não suportada pelos tipos nativos **float** e **double**.

5.4. Bibliotecas em Ponto flutuante escritas na linguagem Java por terceiros

5.4.1. Float

Hoje em dia, Java está presente em vários dispositivos eletrônicos que incluem palmtops e celulares que suportem a plataforma Java Micro Edition e os padrões MIDP/CLDC.

A primeira versão MIDP/CLDC não suportava ponto flutuante e, por isso, alguns desenvolvedores Java insatisfeitos com a falta de suporte da linguagem, criaram bibliotecas que emulassem via software o ponto flutuante. Uma delas é Float (KLIMCHUCK), uma biblioteca livre, para uso não comercial, de ponto flutuante para Java. Esta biblioteca de software suporta operadores aritméticos e relacionais e disponibiliza várias funções matemáticas, como por exemplo, funções trigonométricas.

Internamente, ela usa um formato próprio de ponto flutuante com 64-bits de largura para o expoente e 64-bits de largura para o significando. Essa biblioteca não fornece funções de conversão para os tipos de ponto flutuante nativos de Java, os quais estão de acordo com o formato definido no padrão IEEE-754.

Assim como os tipos de ponto flutuante nativo de Java, somente o arredondamento para o número mais próximo está disponível para biblioteca Float. Além disso, ela suporta as seguintes operações:

- Aritméticas (+, -, x, /, - unário)
- Relacionais (>, <, =)
- Funções trigonométricas (seno, co-seno, tangente, arco-seno, arco-cosseno, arco- tangente).
- Outras (exponencial, logaritmo natural e de base 10, potência, teto, piso, parte inteira, parte fracionária.

Float não dispõe de função de conversão para o formato estabelecido pelo padrão IEEE-754, nem arredondamento direcionado.

5.4.2. Real

Outra biblioteca gratuita de ponto flutuante, é Real (LAURITZEN). Assim como Float, ela tem um enorme conjunto de funções para números de ponto flutuante. Os números são armazenados usando um significando com 63-bits de largura e um expoente interno com 31-bits de largura. Diferente da biblioteca Float, a biblioteca tem funções de conversão para os tipos nativos de Java.

Assim como Float, somente o arredondamento para o número mais próximo é suportado.

Suporta um conjunto extenso de operações e funções com ponto flutuante. Diferente de Float dispõe de funções de conversão para o formato IEEE-754.

Apesar das vantagens sobre Float, a biblioteca Real também não tem suporte a arredondamento direcionado.

5.4.3. Considerações

Das bibliotecas escritas em Java analisadas neste capítulo, nenhuma delas resolveu o problema do arredondamento direcionado em Java. Apesar disso, a análise do código fonte das bibliotecas citadas contribuiu com idéias para o desenvolvimento da biblioteca apresentada no próximo capítulo.

Capítulo 6

Biblioteca JFloat

6.1. Características de JFloat

A biblioteca proposta é uma implementação de software do padrão IEEE-754 para aritmética binária de ponto flutuante. JFloat é livremente baseada na biblioteca SoftFloat, e foi construída para prover características não implementadas em Java como arredondamento direcionado.

Características principais de JFloat:

- **Compatibilidade entre versões de Java**. Compatível com a versão 1.4.2 e superiores de Java. Qualquer computador habilitado com Java pode usar a biblioteca de JFloat. A biblioteca foi criada usando o ambiente de desenvolvimento integrado Eclipse versão 3.1, mas qualquer compilador da linguagem Java pode ser usado para compilar o código fonte do projeto,
- **Independência de máquina**. JFloat representa números em ponto flutuante como variáveis inteiras. Esta característica assegura o mesmo comportamento em qualquer plataforma, porque são usadas somente operações sobre tipos inteiros para emular operações em ponto flutuantes,
- **Precisão**. A versão atual da biblioteca JFloat trabalha somente com números de ponto flutuantes em precisão simples armazenados em variáveis inteiras. Uma versão de precisão dupla de JFloat está em desenvolvimento,
- **Operações**. Estão disponíveis as operações de adição, subtração, multiplicação, divisão, raiz quadrada, resto e comparação entre números,
- **Funções de conversão**. A biblioteca dispões de funções de conversão para os formatos de ponto flutuante disponíveis na linguagem Java e números inteiros,
- **Disponíveis todos os modos de arredondamento**. Diferente do suporte nativo de Java a ponto flutuante e das bibliotecas de terceiros desenvolvidas para Java, JFloat oferece todos os modos de arredondamento requeridos pelo padrão IEEE-754,

- **Suporte a NaN e flags de exceção**. JFloat suporta o uso de QNaNs e SNaNs e após cada operação, os flags de exceção correspondentes são marcados de acordo com o resultado.
- **Coleta de lixo mínima**. Para evitar a limpeza de memória (**garbage collection**), as funções de biblioteca evitam a alocação de objetos, o que contribui para velocidade e diminuição dos gastos de memória.

6.2. Arquitetura de JFloat

6.2.1. Representação de dados

JFloat, assim como as bibliotecas de terceiros citadas anteriormente, não faz uso do suporte de hardware nativo para aritmética de ponto flutuante. Ela usa somente software para emular ponto flutuante. Além disso, internamente são armazenados números de ponto flutuantes dentro de variáveis do tipo inteiro.

Um grande desafio enfrentado para criação de JFloat foi a ausência em Java de tipos inteiros não sinalizados e o emprego de algoritmos que necessitavam desse tipo de variáveis.

A primeira solução pensada foi armazenar uma variável inteira não sinalizada em uma variável inteira sinalizada e manipular o bit mais significante para evitar operações cujo resultado fosse um número negativo. Nesta abordagem, seria preciso muito cuidado quando uma operação com operandos inteiros não sinalizados causasse um overflow e mudasse o bit mais significante. Essa solução seria muito difícil de implementar porque seria necessário verificar um possível **overflow**, antes de cada operação aritmética não sinalizada.

A segunda solução para resolver o problema seria evitar o "bit de sinal". Esta solução usaria somente os 31 bits menos-significantes de um tipo "int" Java (32 bits) para representar um número de precisão simples e 63 bits menos-significantes de um tipo "longo" Java (64 bits-largura) para representar números de precisão dupla. Essa solução é semelhante a encontrada na biblioteca Real. Não se verificou uma solução muito boa porque se perderia um bit de precisão.

Ao final, chegou-se a solução proposta neste trabalho, em que para evitar mudanças de sinal, cada variável inteira não sinalizada (32 bits-largura) seriam armazenados nos bits menos significantes de uma variável inteira longa sinalizada (64 bits-largura). Os 32 bits mais significantes da variável longa sinalizada sempre seriam zero.

6.2.2. Classes componentes

As principais classes da biblioteca JFloat são JFloat32 e sua superclasse JFloat32BASE.

A classe JFloat32 esconde a complexidade interna de JFloat32Base. JFloat32 define operações elementares de aritmética como métodos estáticos públicos com parâmetros float. Por exemplo, a assinatura de método de operação é declarada como:

```
public add(float primeiro, float segundo)
```

Por outro lado, JFloat32BASE contém operações de baixo nível para emular operações de ponto flutuante que só usam variáveis longas. As operações elementares (adição, subtração, multiplicação, divisão, raiz quadrada e resto) estão definidas em JFloat32BASE. Operações prévias definidas em JFloa32 usam a operação de adição declarada em JFloat32BASE:

```
protected static long add(long primeiro, long segundo)
```

Cada método da interface da classe JFloat32 é definida para receber parâmetros no formato de precisão simples IEEE-754 (tipo Java float). Primeiramente, no início de cada função definida em JFloat32, parâmetros de precisão IEEE-754 (Java "float"), temporariamente, são convertidos para a representação interna (tipo inteiro longo) e então é executado o código da função correspondente. Após, o resultado é convertido do formato interno de representação para o formato de precisão simples do IEEE-754 e retornado para a função chamadora. Abaixo um fragmento da classe JFloat32 que demonstra essa conversão:

```
public void funcao_chamadora_exemplo(){
float result = JFloat32.add(0.1234, 9899.94);
System.out.println(result);
}
...
public static float JFloat32.add(float a, float b)
{
long float32a = JFloat32Base.floatToFloat32(a); /* conversão do
parametro */
long float32b = JFloat32Base.floatToFloat32(b); /* conversao do
parametro */
long ret = JFloat32Base.add(float32a,float32b);
/* executa método sobrecarregado add(long,long) */
```

```
      return   JFloat32Base.float32ToFloat(ret);   /*   retorna   resultado
convertido */
      }
```

Normalmente, usuários de JFloat biblioteca deveriam usar somente JFloat32 em suas aplicações Java.

6.2.3. Precisão

A biblioteca JFloat, atualmente, só trabalha com o formato de precisão simples (32 bits) definido pelo padrão IEEE-754. Ela se baseia em parte na biblioteca SoftFloat, originalmente escrita na linguagem C e em parte de rotinas presentes na biblioteca Real.

Na versão atual da biblioteca JFloat, apesar do formato de precisão simples requerer apenas 32 bits, eles são armazenados em variáveis do tipo long que possuem 64 bits. Essa decisão de projeto, tida a primeira vista como um desperdício de memória se deveu ao fato da versão original da biblioteca necessitar trabalhar com números inteiros não sinalizados. Ocorre que todo tipo inteiro em Java é sinalizado. A solução encontrada foi usar um inteiro sinalizado de 64 bits como se fosse um inteiro não sinalizado de 32 bits. Isto é conseguido desprezando-se os 32 bits mais significativos do inteiro a cada operação que pudesse alterar esses bits.

Os métodos definidos na classe Float32BASE somente aceitam parâmetros do tipo long que é o formato usado internamente para realizar os cálculos de números em ponto flutuante. A classe Float32BASE foi criada apenas para servir de base a classe Float32 e não pretende ser usada diretamente pelo desenvolvedor externo.

A classe Float32, por outro lado, aceita diretamente parâmetros do tipo float, o que permite códigos em Java do tipo:

```
float a = 0.5f;
float b = 3.14f;
float c = Float32.mul(a,b);
float d = Float32.sin(b);
```

6.3. Descrição da interface de programação de JFloat

A biblioteca proposta tem uma classe principal denominada JFloat32. Essa classe provê operações elementares de ponto flutuantes requeridas pelo padrão IEEE-754: adição, subtração, multiplicação, divisão, raiz quadrada e resto. A presente versão disponibiliza ainda operadores relacionais e funções básicas como seno, co-seno e tangente.

Uma versão futura implementará uma boa parte das funções definidas pelo padrão de biblioteca matemática ISO C99 (ISO, 2003).

6.3.1. Operações aritméticas elementares

Uma vez que Java, diferentemente de outras linguagens como C++, não possibilita a sobrecarga de operadores, as operações aritméticas elementares (adição, subtração, multiplicação, divisão, resto e raiz quadrada) foram definidas como funções (métodos de classe) que recebem operandos de ponto flutuante e retornam os resultados como definidos pelo padrão IEEE-754.

Todos os operadores aritméticos e funções auxiliares estão disponíveis como métodos públicos e estáticos da classe JFloat:

- float add(float x, float y): soma dois números de ponto flutuantes;
- float sub(float x, float y): subtrai dois números de ponto flutuantes;
- float mul(float x, float y): multiplica dois números de ponto flutuantes;
- float div(float x, float y): divide dois números de ponto flutuantes;
- float rem(float x, float y): retorna o resto da divisão de x por y, como definido no padrão IEEE-754,
- float sqrt(float x): retorna a raiz quadrada de x.

A tabela abaixo mostra um exemplo de código Java para somar dois números de ponto flutuante.

```
float a, b, c;
b = 0.5f;
c = 1.34f;
a = JFloat32.add(b,c);
```

A biblioteca JFloat, desenvolvida nesse trabalho, reside basicamente em duas classes Java, uma chamada Float32BASE e a outra Float32.

A classe Float32BASE contém métodos correspondentes aos operadores aritméticos (+, -, X, /, resto, √), operadores relacionais (=, <=, <) e funções de conversão exigidos pela norma IEEE-754.

Por sua vez a classe Float32 é uma subclasse de Float32BASE e por isso herda todos os métodos de sua superclasse. Nesta classe, são adicionados métodos que fazem a conversão entre o formato interno usado por JFloat e o tipo de variável nativo "float" encontrado na linguagem Java.

6.3.2. Funções para tratamento de Not a Numbers

Para identificar se o resultado de uma operação retornou um NaN, JFloat dispõe de duas funções isNaN e isSNaN. A primeira informa se o valor informado é um NaN. A segunda se é um Signaling NaN.

- boolean IsNaN(float x): retorna verdadeiro se x é um NaN e falso caso contrário;
- boolean IsSNaN(float x): retorna verdadeiro se x é um SNaN (Signaling NaN) e falso caso contrário.

6.3.3. Operadores relacionais definidos como funções

As funções seguintes trabalham como operadores relacionais:

- boolean eq(float x, float y): retorna verdadeiro se x é igual a y, e falso caso contrário;
- boolean neq(float x, float y): retorna verdadeiro se x não é igual a y, e falso caso contrário,
- boolean eqSignaling(float x, float y): retorna verdadeiro se x é igual a y, e falso caso contrário. Uma exceção inválida é levantada se qualquer dos operandos for um NaN;
- boolean ge(float x, float y): retorna verdadeiro se x é maior que ou igual a y, e falso caso contrário;
- boolean gt(float x, float y): retorna verdadeiro se x é maior que y, e falso caso contrário,
- boolean le(float x, float y): retorna verdadeiro se x é menor que ou igual a y, e falso caso contrário,

- boolean leQuiet(float x, float y): retorna verdadeiro se x é menor que y, e falso caso contrário. NaNs não causam uma exceção;
- boolean lt(float x, float y): retorna verdadeiro se x é menor que y, e falso caso contrário;
- ltQuiet(float x, float y): retorna verdadeiro se x é menor que y. NaNs não causam uma exceção.

6.3.4. Funções de tratamento de exceção

As operações realizadas pela biblioteca podem gerar exceções indicadas pelas constantes abaixo:

- public static final int FLAG_INVALID = 1;
- public static final int FLAG_DIV_BY_ZERO = 4;
- public static final int FLAG_OVERFLOW = 8;
- public static final int FLAG_UNDERFLOW = 16;
- public static final int FLAG_INEXACT = 32;

É possível mudar ou examinar os flags de exceção usando as funções seguintes:

- void setExceptionFlags(int exceptionFlags): especifica vários flags de exceção ao mesmo tempo, usando sobre constantes, por exemplo, setExceptionFlags(FLAG_OVERFLOW + FLAG_INEXACT)
- void RaiseException(int flag): levanta uma exceção definida pelos flags, por exemplo, raiseException (FLAG_UNDERFLOW).
- int getExceptionFlags (): retorna os flags de exceção.

6.3.5. Funções de conversão e arredondamento

JFloat32 provê funções de conversão de e para inteiros:

- float Int32ToFloat32(int x): converte um inteiro de 32-bits em um número de precisão simples número de ponto flutuante;
- float RoundToInt(float x): arredonda um número em precisão simples para um inteiro e retorna o resultado como uma número de ponto flutuante de precisão simples.

6.3.6. Controle de arredondamento

O padrão IEEE-754 estabelece como toda operação será executada como se fosse primeiro produzido um resultado intermediário correto para exatidão infinita e com faixa ilimitada, e então arredondado o resultado de acordo com um dos quatro modos de arredondamento (ANSI/IEEE, 1985).

A maioria de aplicações numéricas requer operações de ponto flutuantes que usam só o modo padrão de arredondamento do IEEE-754 (arredondamento para o mais próximo). Porém, há aplicações que precisam de outros modos de arredondamento. Por exemplo, este modo não é o bastante para aplicações intervalares.

Devido a somente o modo de arredondamento padrão estar disponível em Java, é comum bibliotecas intervalares implementadas nessa linguagem recorrerem ao inflacionamento dos limites dos resultados, após cada arredondamento, às vezes desnecessariamente. O exemplo abaixo, mostra um exemplo de programa em Java que efetuaria uma adição de intervalos:

```
      float a_low, a_high, b_low, b_high, c_low, c_high;
      a_low = 0.3f ; a_high = 0.5f; /* a = [0.3, 0.5] */
      b_low = 0.25f; b_high = 0.75; /* b = [0.25, 0.75] */
      /* Available rounding mode for Java is 'neareast even'*/
      c_low = a_low + b_low;
      c_high = a_high + b_high;
      /* function prevfp returns previous floating point number of c_low */
      c_low = prevfp(c_low) ;
      /* function nextfp returns next floating point number of c_high */
      c_high = nextfp(c_hight);
      /* c = [0.3 + 0.25 - delta, 0.5+0.75 + epsilon] with unconditional
inflation of limits */
```

JFloat provê todos os quatro modos de arredondamento como definido no padrão de IEEE-754.

Usando a função setRoundingMode(int modo), é possível definir o arredondamento desejado para as operações matemáticas subseqüentes.

A biblioteca JFloat define os quatro modos de arredondamento definidos pela norma, e os deixa disponíveis a partir das constantes públicas:

- public static final int ROUND_NEAREST_EVEN = 0;
- public static final int ROUND_DOWN = 1;

- public static final int ROUND_UP = 2;
- public static final int ROUND_TO_ZERO = 3;

O modo de arredondamento atual afeta resultados de operação como definido no padrão de IEEE-754.

Pode-se especificar e inspecionar o modo de arredondamento atual, respectivamente, usando as funções:

- void setRoundingMode(int roundingMode): especifica o modo de arredondamento atual por exemplo, setRoundingMode(ROUND_UP).
- int getRoundingMode(): retorna o modo de arredondamento atual.

Abaixo, a mesma soma de intervalos usando o arredondamento direcionado de JFloat:

```
float a_low, a_high, b_low, b_high, c_low, c_high;
a_low = 0.3f ; a_high = 0.5f; /* a = [0.3, 0.5] */
b_low = 0.25f; b_high = 0.75; /* b = [0.25, 0.75] */
/* Round down addiction of lower limit */
JFloat32.setRoundingMode(JFloat32.ROUND_DOWN);
c_low = JFloat32.add(a_low, b_low);
/* Round up addiction of upper limit */
JFloat32.setRoundingMode(JFloat32.ROUND_UP);
c_high = JFloat32.add(a_high, b_high);
/* c = [0.3 + 0.25, 0.5+0.75] */
```

6.3.7. Funções avançadas

No momento, poucas funções adicionais foram implementadas. Dentre elas, estão as funções trigonométricas seno, co-seno e tangente. A implementação de outras funções, como hiperbólicas, logaritmo e funções exponenciais estão sendo planejadas.

As funções avançadas presentes em JFloat são construídas usando as funções aritméticas elementares. Por exemplo, na classe JFloat32 está definida uma função chamada sinInternal que computa o seno por meio do polinômio $sen(x) = x - x^3/3! + x^5/5! - x^7/7! + x^9/9! - ...$, onde $x < pi/4$, criada usando somente as funções elementares add() e mul (). Uma vez que as funções são escritas usando somente funções elementares, o controle de modo de arredondamento atual afeta os resultados intermediários e consequentemente o resultado final dessas funções.

Abaixo, é mostrado como exemplo a função sinInternal definida a partir das funções elementares add (adição) e mul (multiplicação).

```
/**
* Compute sin(x) = x - x^3/3! + x^5/5! - ... - x^15/15!, for x < pi/4
*/
protected static long sinInternal(long valor)
{
long accumulator, tmp1, tmp2, tmp3;
tmp1 = valor;
tmp2 = valor;
tmp2 = mul(tmp2, tmp2);
accumulator = toFloatBits(1, 0x3fffffd7, 0x6aa891c4f0eb2713L);
/* -1/15! = -7.5785404094842805756291E-13 */
accumulator = mul(accumulator, tmp2);
tmp3 = toFloatBits(0, 0x3fffffdf, 0x58482311f383326cL);
/* +1/13! = 1.60583631673204432492311E-10 */
accumulator = add(accumulator, tmp3);
accumulator = mul(accumulator, tmp2);
tmp3 = toFloatBits(1, 0x3ffffffe6, 0x6b9914a35f9a00d8L);
/* -1/11! = -2.505210488187086878405 5E-8 */
accumulator = add(accumulator, tmp3);
accumulator = mul(accumulator, tmp2);
tmp3 = toFloatBits(0, 0x3ffffffed, 0x5c778e94cc22e47bL);
/* +1/9! = 2.75573192140649222178 61E-6 */
accumulator = add(accumulator, tmp3);
accumulator = mul(accumulator, tmp2);
tmp3 = toFloatBits(1, 0x3ffffff3, 0x680680680629b28aL);
/* -1/7! = -1.9841269841254799668344E-4 */
accumulator = add(accumulator, tmp3);
accumulator = mul(accumulator, tmp2);
tmp3 = toFloatBits(0, 0x3ffffff9, 0x4444444444442b4dL);
/* +1/5! = 8.3333333333333225058715E-3 */
accumulator = add(accumulator, tmp3);
accumulator = mul(accumulator, tmp2);
tmp3 = toFloatBits(1, 0x3fffffffd, 0x555555555555554cL);
/* -1/3! = -1.6666666666666666402 55E-1 */
accumulator = add(accumulator, tmp3);
accumulator = mul(accumulator, tmp2);
accumulator = mul(accumulator, tmp1);
accumulator = add(accumulator, tmp1);
return accumulator;
}
```

Outras funções presentes em JFloat são:

- float mod(float x, float y): devolve x mod y, onde x mod y = x - y
- abs(float x): retorna valor absoluto de x;
- neg(float x): devolve -x;
- ceil(float x): devolve o teto de x.
- sin(float x): devolve o seno de x;
- cos(float x): devolve o coseno de x;
- tan(float x): devolve a tangente de x.

6.4. Integração de JFloat com Java-XSC

A biblioteca JFloat foi projetada para ser facilmente integrada em aplicações que requeiram controle do arredondamento de operações aritméticas. No caso de Java-XSC, será necessário substituir no código fonte a ocorrência dos operadores aritméticos pela chamada dos métodos estáticos de JFloat. Abaixo, vemos um exemplo de como atualmente é feita a adição de dois intervalos em Java-XSC e como ficaria o código com o uso integrado de JFloat.

Código atual da operação de adição intervalar:

```
public static RealInterval add(RealInterval x, RealInterval y) {
  RealInterval z = new RealInterval();
  z.lo = x.lo + y.lo;
  z.hi = x.hi + y.hi;
  return z;
}
```

Código modificado da adição intervalar usando JFloat:

```
public static RealInterval add(RealInterval x, RealInterval y) {
  RealInterval z = new RealInterval();
  JFloat32.setRoundingMode(JFloat32.ROUND_DOWN);
  z.lo = JFloat32.add(x.lo, y.lo);
  JFloat32.setRoundingMode(JFloat32.ROUND_UP);
  z.hi = JFloat32.add(x.hi, y.hi);
  return z;
}
```

6.5. Considerações

Neste capítulo foi apresentada a arquitetura e interface da biblioteca proposta neste trabalho. No próximo capítulo, serão descritos alguns dos testes realizados para verificar o funcionamento de JFloat em comparação com os tipos nativos de Java.

Capítulo 7

Testes

Neste capítulo serão apresentados alguns testes realizados com a biblioteca JFloat32 comparando-a com o tipo nativo de precisão simples float e o tipo nativo de precisão dupla double. Como padrão de referência foi usado o tipo de dupla precisão da biblioteca C-XSC. Para tanto, foi criada uma biblioteca de comunicação entre C-XSC e a linguagem Java pelo uso da Java Native Interface (JNI), o que permitiu chamar as operações de C-XSC como diretamente de dentro dos testes escritos em Java. Os códigos-fonte dos testes realizados neste capítulo encontram-se em anexo ao final do trabalho. Salvo menção em contrário, os testes foram realizados em um computador Toshiba, modelo Satellite, com processador Intel Centrino Core Duo, de 1,6 GHz com 1Gb de RAM e 60 Gb de disco rígido. A máquina virtual usada nos testes foi a JVM 5.0 da Sun. Para o teste de Rump, foi também usada a JVM da BEA.

7.1. Teste de exatidão de JFloat32, Float e Double em comparação ao tipo de ponto flutuante de dupla precisão da biblioteca C-XSC

O primeiro teste que foi feito verifica a exatidão de cada operação de JFloat32, float e double nativos de Java em relação ao tipo de ponto de flutuante de dupla precisão da biblioteca C-XSC.

7.1.1. Resultado

	Tipo	Erro Máximo	Erro Médio	Variância	Desvio padrão
+	JFloat	1,17578448341E-07	4,22914497210E-08	7,95642395373E-17	8,91987889701E-09
	Float	5,96023462583E-08	4,22760258464E-08	7,80312441087E-17	8,83352953856E-09
-	JFloat	1,78493911216E-07	4,22468955495E-08	8,08546811035E-17	8,99192310374E-09
	Float	5,96019483877E-08	4,22327130101E-08	7,82320483444E-17	8,84488826071E-09
*	JFloat	6,99484153676E-06	4,24226511579E-08	2,52748883353E-14	1,58980779767E-07
	Float	6,99479214434E-06	4,19930603306E-08	2,52752833172E-14	1,58982021994E-07
/	JFloat	7,88687384414E-06	4,41521376403E-08	1,14780128141E-15	3,38792160685E-08
	Float	7,88686760851E-06	4,37273398502E-08	1,14658697079E-15	3,38612901525E-08

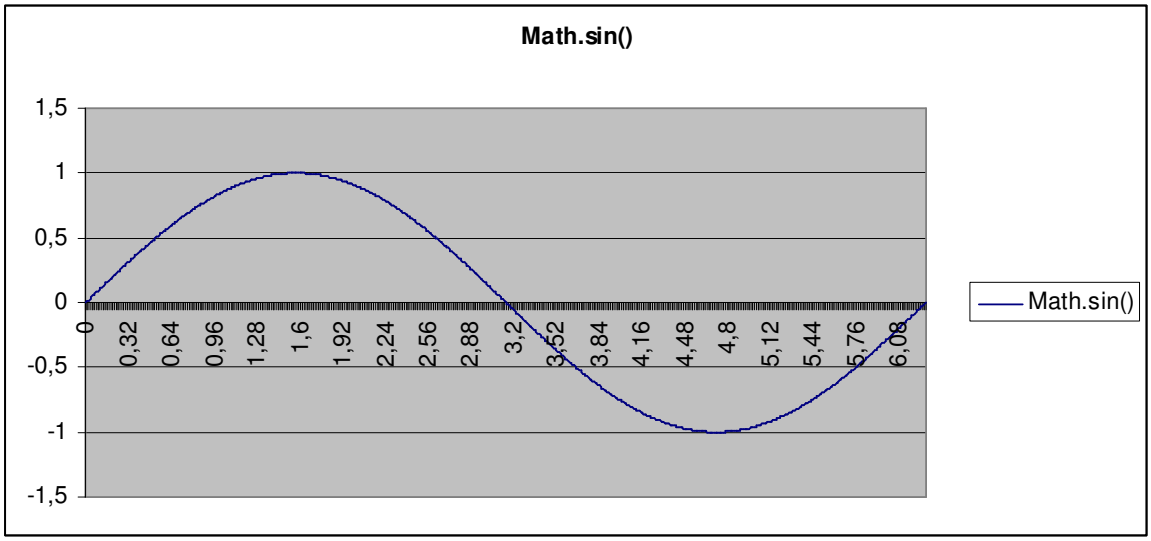
Pela tabela acima, observa-se que a exatidão de JFloat está muito próxima a do tipo de ponto flutuante de precisão simples nativo para cada uma das operações aritméticas.

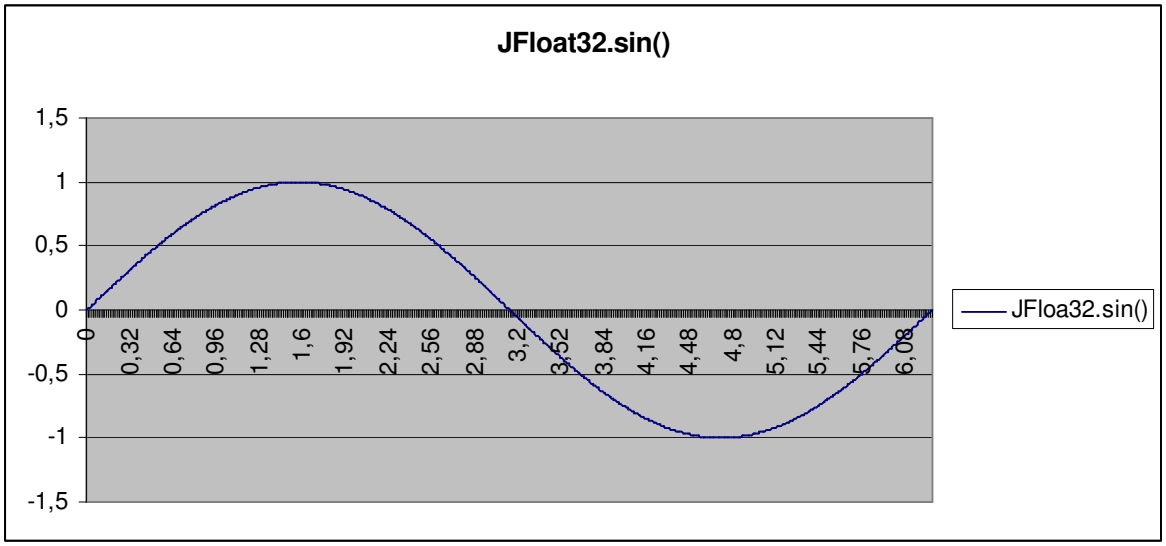
7.2. Teste gráfico da função JFloat32.sin()

Neste teste, é comparado graficamente o comportamento da função seno implementada na biblioteca JFloat32 com a implementação nativa em dupla precisão.

7.2.1. Resultado

Pela inspeção visual dos resultados, observa-se que os dois gráficos têm a mesma forma e comportamento.





7.3. Teste da exatidão da função JFloat32.sin()

A linguagem Java não possui uma função seno de precisão simples. Por isso, a função seno de **precisão simples** de JFloat foi comparada com a função seno nativa de **precisão dupla** e a função seno da biblioteca C-XSC.

7.3.1. Resultado

Tipo	Máximo	Média	Variância	Desvio padrão
Double	1,11022302463E-16	1,06492658440E-17	7,81007722727E-34	2,79465153951E-17
JFloat	1,79375310272E-07	5,79634026398E-08	2,83837232689E-15	5,32763768183E-08
Nativo	2,97911730840E-08	1,11305021116E-08	7,44133134830E-17	8,62631517411E-09

Pelo resultado, observa-se que a precisão nativa simples é melhor que a função seno criada em JFloat, quando comparadas a biblioteca seno de C-XSC. Nota-se que mesmo a função seno nativa de Java para double tem um erro em relação a C-XSC.

O erro médio de JFloat possui a mesma ordem de grandeza do obtido pelo tipo nativo de precisão simples o que sugere que a precisão obtida pelas funções elementares **JFloat32.add()** e **JFloat32.mul()** usadas para calcular a função **JFloat32.sin()** possuem um erro muito pequeno.

7.4. Teste da velocidade da função JFloat32.sin()

Neste teste, foi verificada a velocidade da função seno de JFloat comparada com a função nativa de dupla precisão da linguagem Java e a função seno da biblioteca C-XSC.

Foi usado neste teste um computador Toshiba, modelo Satellite, com processador Intel Centrino Core Duo, de 1,6 GHz com 1Gb de RAM e 60 Gb de disco rígido e a máquina virtual 5.0 da Sun.

7.4.1. Resultado

Biblioteca	Tempo (ms)
Java	1014
CXSC	5397
JFloat32	16036

Como era de se esperar a função nativa foi muito mais rápida que a implementada com Java puro. No teste realizado, a função nativa foi, pelo menos, 20 (vinte) vezes mais rápida

que a implementada por JFloat. A biblioteca C-XSC por usar emulação de software escrita em C foi mais lenta que a função nativa de Java.

7.5. Teste da função de Rump

Neste teste é apresentado um exemplo de erro de arredondamento (RUMP, 1988) computado para os tipos nativos de ponto flutuante de Java (precisão simples e dupla) e a biblioteca JFloat.

7.5.1. Algoritmo

Sejam a = 77617 e b = 33096, a função de teste é definida como

$$(((333,75.b^6)+(a^2.(((((11.a^2)\ .\ b^2)-b^6)-(121.b^4))-2)))+(5,5.b^8))+(a/(2.b))$$

7.5.2. Resultado

O valor correto deveria ser de aproximadamente −0.82739605994682135, mas devido a erros de arredondamento os valores gerados pelos tipos nativos de precisão simples e de dupla precisão de Java divergem significativamente do esperado.

Do quadro, observa-se que durante os testes houve uma grave discrepância no tipo nativo de ponto flutuante de precisão simples quando executando sobre a máquina virtual Java BEA JRockit e o processador AMD Athlon XP. Esse teste demonstra que Java apesar de sua grande portabilidade, **pode estar sujeito a divergências de resultados** quando operando programas de ponto flutuante em arquiteturas distintas.

Processador	**Máquina Virtual**	**Biblioteca**	**Resultado**
Pentium Centrino Duo	Sun JRE 1.6 Win32	double nativo	-1.1805916207174113E21
Pentium Centrino Duo	BEA JRockit JRE 1.6 Win32	double nativo	-1.1805916207174113E21
AMD Athlon XP 2600+	Sun JRE 1.6 Win32	double nativo	-1.1805916207174113E21
AMD Athlon XP 2600+	BEA JRockit JRE 1.6 Win32	double nativo	-1.1805916207174113E21
Pentium Centrino Duo	Sun JRE 1.6 Win32	float nativo	-6.338253E29
Pentium Centrino Duo	BEA JRockit JRE 1.6 Win32	float nativo	-6.338253E29
AMD Athlon XP 2600+	Sun JRE 1.6 Win32	float nativo	-6.338253E29
AMD Athlon XP 2600+	BEA JRockit JRE 1.6 Win32	float nativo	1.5347691E22
Pentium Centrino Duo	Sun JRE 1.6 Win32	JFloat32	-6.338253E29

Pentium Centrino Duo	BEA JRockit JRE 1.6 Win32	JFloat32	-6.338253E29
AMD Athlon XP 2600+	Sun JRE 1.6 Win32	JFloat32	-6.338253E29
AMD Athlon XP 2600+	BEA JRockit JRE 1.6 Win32	JFloat32	-6.338253E29

7.6. Teste da exatidão de JFloat comparada a exatidão de ponto flutuante de C-XSC

Neste teste, foi testado se os cálculos envolvendo JFloat aproximaram-se mais do ponto flutuante de C-XSC que o tipo nativo de Java de precisão simples (float).

7.6.1. Resultado

Para as operações de adição e subtração, verificou-se que o arredondamento da biblioteca JFloat obteve resultados mais próximos do valor obtido pela biblioteca C-XSC. Da tabela, vê-se que a implementação nativa de Java é mais precisa que a presente em JFloat.

Verificamos que o percentual apresentado é irrisório e natural de acontecer, uma vez que internamente Java trabalha com precisão dupla, inclusive para float e ao final ocorre o arredondamento.

Descrição/Operação	**+**	**-**	**x**	/
Número de vezes em que o valor obtido por JFloat e pela precisão nativa simples igual ao valor obtido pela precisão dupla de C-XSC	6364157 (63,64157%)	6359769 (63,59769%)	9 (0,00009%)	9 (0,00009%)
Número de vezes em que o valor obtido por JFloat está **mais perto** do resultado obtido pela precisão dupla de C-XSC que o valor obtido pela precisão nativa simples.	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Número de vezes em que o valor obtido por JFloat e pela precisão nativa simples estão a **mesma distância** do resultado obtido pela precisão dupla de C-XSC.	3635832 (36,35832%)	3640222 (36,40222%)	9980045 (99,80045%)	9980123 (99,80123%)
Número de vezes em que o valor obtido por JFloat está **mais longe** do resultado obtido pela precisão dupla de C-XSC que o valor obtido pela precisão nativa simples.	11 (0,00011%)	9 (0,00009%)	19946 (0,19946%)	19868 (0,19868%)
Total	**10000000 (100%)**	**10000000 (100%)**	**10000000 (100%)**	**10000000 (100%)**

7.7. Teste do arredondamento direcionado em relação ao resultado de precisão dupla

Neste teste, é verificada a exatidão do arredondamento direcionado em relação ao ponto flutuante de precisão dupla de C-XSC.

7.7.1. Resultado

Descrição/Operação	**+**	**-**	**x**	/
Número de vezes, em uma amostra de 10.000.000 de tentativas, que o arredondamento direcionado para baixo de JFloat resultou o número **imediatamente inferior** ao obtido pelo tipo de precisão dupla de C-XSC.	9999999 (99,99999%)	9999998 (99,99998%)	10000000 (100%)	10000000 (100%)
Número de vezes, em uma amostra de 10.000.000 de tentativas, que o arredondamento direcionado para baixo de JFloat **não** resultou o número **imediatamente inferior** ao obtido pelo tipo de precisão dupla de C-XSC.	1 (0,00001%)	2 (0,00002%)	0 (0%)	0 (0%)
Total	**10000000 (100%)**	**10000000 (100%)**	**10000000 (100%)**	**10000000 (100%)**
Número de vezes, em uma amostra de 10.000.000 de tentativas, que o arredondamento direcionado para cima de JFloat resultou o número **imediatamente superior** ao obtido pelo tipo de precisão dupla de C-XSC.	9999999 (99,99999%)	10000000 (100%)	10000000 (100%)	10000000 (100%)
Número de vezes, em uma amostra de 10.000.000 de tentativas, que o arredondamento direcionado para cima de JFloat **não** resultou o número **imediatamente superior** ao obtido pelo tipo de precisão dupla de C-XSC.	1 (0,00001%)	0 (0%)	0 (0%)	0 (0%)
Total	**10000000 (100%)**	**10000000 (100%)**	**10000000 (100%)**	**10000000 (100%)**

Os testes mostraram que, raríssimas vezes, o resultado com arredondamento direcionado resultante do uso de JFloat não resultou o número imediatamente superior ou inferior, de acordo com o tipo de arredondamento usado. Este resultado indica que JFloat está na maior parte dos casos arredondando o resultado para o **melhor** resultado possível. Mais uma vez, verificamos que o percentual apresentado é irrisório e natural de acontecer, uma vez que internamente Java trabalha com precisão dupla, inclusive para float e ao final ocorre o arredondamento.

Capítulo 8

Considerações finais

8.1. Conclusões

As bibliotecas para suporte a números em ponto flutuantes nativas de Java ou de terceiros atualmente não suprem as necessidades de controle direcionado de arredondamento requeridas por Java-XSC. As bibliotecas Float e, em especial, Real, provêem bom suporte a ponto flutuante, mas carecem de arredondamento direcionado. Portanto, a falta de controle direcionado de arredondamento as torna inadequadas para Java-XSC. Se for possível usar uma solução com código escrito em C, a biblioteca Softfloat, poderia atender essa necessidade. No entanto, a intenção do trabalho, desde seu início, seria prover uma solução para o problema usando código escrito totalmente em Java.

Da análise do código fonte das bibliotecas citadas, em especial SoftFloat que parece ser a mais compatível com a norma IEEE-754, foi construída a biblioteca JFloat. Nela foram codificadas as operações básicas exigidas pela norma.

Este trabalho procurou contribuir para a comunidade científica disponibilizando uma biblioteca de ponto flutuante escrita em Java que permite o controle do arredondamento direcionado em operações de ponto flutuante e serve como base para a construção de bibliotecas intervalares escritas em Java.

Devido ao tempo escasso para conclusão da dissertação, somente o formato de ponto flutuante de precisão simples foi codificado. Em edições futuras da biblioteca, esta poderá ser expandida para suportar números de precisão dupla ou de precisão superior.

Como esperado, as operações aritméticas executadas pela biblioteca serão mais lentas que a executada por um hardware dedicado como um processador numérico. Todavia, a ausência do controle nativo direcionado de arredondamento em Java, atualmente, deixa a opção de efetuar esse controle ignorando o hardware e fazendo-o completamente por software.

Um ponto interessante levantado pelos testes, é que houve divergência de resultados para a biblioteca Java nativa da BEA quando rodando em um AMD de arquitetura de 32 bits,

o que pode levar a concluir que Java pode apresentar problemas de portabilidade para ponto flutuante.

8.2. Perspectivas

Pelas possibilidades surgidas desse trabalho, pretende-se dar continuidade ao aperfeiçoamento da biblioteca JFloat, particularmente do aumento da precisão da biblioteca de precisão simples e criação da versão de 64 bits para números em ponto flutuante de dupla precisão.

Outra possibilidade será a adaptação da biblioteca JFloat para suportar exatidão máxima a exemplo da biblioteca Libavi.

Também, se pode continuar a implementação de novas funções trigonométricas e outras de uso corrente em aplicações científicas.

Uma vez concluída essa etapa, poder-se-á adaptar a biblioteca Java-XSC para usar o controle de arredondamento direcionado e assim evitar inflacionamentos incondicionais dos limites dos intervalos, contribuindo para a linha de pesquisa sobre o Java-XSC e matemática intervalar, desenvolvida pelo Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

Referências Bibliográficas

ANSI/IEEE. Supplemental readings for IEEE 754 / 854. http://grouper.ieee.org/groups/754/reading.html.

_____. ANSI/IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE. New York. 1985

BEDREGAL, B. R. C. e B. ACIÓLY. Rational Intervals: An Effectively Given Information Systems. International Conference on Mathematical Modelling and Scientific Computation, p.159-170. 1993.

BEDREGAL, B. R. C. e J. E. M. DUTRA. JAVA-XSC: Estado da arte. XXXII Conferencia Latinoamericana de Informática (CLEI 2006). Santiago: IFIP, CLEI, SCCC e USACH, 2006. 1-12 p.

BELANOVIC, P. Library of parameterized hardware modules for floating-point arithmetic with an example application. (Master thesis). Northeast University, 2002.

BOISVERT, R. F. e J. P. M. P. R. MOREIRA. Java and numerical computing. Computing in Science Engineering, v.3, n.2, p.18-24. 2001.

DIVERIO, T. A. Uso Efetivo da Matemática Intervalar em Supercomputadores Vetoriais. (Tese de Doutorado). CPGCC, UFRGS, Porto Alegre, 1995.

DOERDELEIN, O. Matemática em Java: mastigando números na JVM. Java Magazine, v.19. 2005.

DUTRA, J. E. M. Java-XSC: uma biblioteca java para computações intervalares. (Dissertação de mestrado). PPGSC, UFRN, Natal, 2000.

FERNANDES, U. A. L. Núcleo de Aritmética de Alta Exatidão da Biblioteca Intervalar libavi.a. (Master thesis). CPGCC, UFRGS, Porto Alegre, 1997. 117 p.

GOLDBERG, D. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys, v.23, n.1, p.5-48. 1991.

HAUSER, J. R. Softfloat: http://www.jhauser.us/arithmetic/SoftFloat.html.

HICKEY, T., Q. JU*, et al.* Interval Arithmetic: From principles to implementation. Journal ACM, v.48, n.5, p.1038-1068. 2001.

HOFSCHUSTER, K. C-XSC 2.0: a C++ library for extended scientific computing. Heidelberg: Springer-Verlag, v.2991. 2004

HÖLBIG, C. A. Computação Verificada. 2004 (Série Aritmética Computacional)

HÖLBIG, C. A., R. L. SAGULA, *et al.* High Accuracy and High Performance Environment. Reliable Computing, p.51-52. 1996.

ISO. Rationale for International Standard - Programming Language C. http://www.open-std.org/JTC1/SC22/WG14/www/C99RationaleV5.10.pdf.

KAHAN, W. How java floating-point hurts everyone everywhere. http://cch.loria.fr/documentation/IEEE754/wkahan/JAVAHurt.pdf.

KEARFOTT, R. Rigorous Global Search: Continuous Problems. Dordrecht: Kluwer Academics Publishers. 1996. 285 p.

KLIMCHUCK, N. Float. http://henson.newmail.ru/j2me/Float11.htm.

LAURITZEN, R. Real: Java Floating Point Library for MIDP Devices. http://real-java.sourceforge.net/Real.html.

LEONIDAS, M. e F. CAMPOS. Cálculo numérico com aplicações. 1987.

MICROSOFT. Floating-point arithmetic may give inaccurate results in Excel.: http://support.microsoft.com/default.aspx?scid=kb;en-us;78113.

MICROSYSTEMS, S. Performance regression in trigonometic functions (very slow StrictMath). Http://Bugs.Sun.Com/View_Bug.Do?Bug_Id=4857011.

OLIVEIRA JUNIOR, O. F. Números dinâmicos: uma abordagem computacional orientada a objetos com implementações na linguagem Java. (Dissertação de mestrado). UFRN, 2004.

RUMP, S. M. Reliability in Computing. The Role of Interval Methods in Scientific Computing. Academic Press. 1988.

SANTIAGO, R. H. N., B. R. C. BEDREGAL, *et al.* Formal Aspects of Correctness and Optimality of Interval Computations. Formal Aspects of Computing, n.18, p.231-243. 2006.

SANTOS, J. M. Em direção a uma representação para equações algébricas: uma lógica equacional local. (Dissertação de mestrado). PPGSC, UFRN, Natal, 2001.

VUIK, K. Some disasters caused by numerical erros. Http://Ta.Twi.Tudelft.Nl/Users/Vuik/Wi211/Disasters.Html.

Apêndices

10.1. Documentação de referência da classe JFloat32

br.ufrn.dimap.jxsc
Classe JFloat32
java.lang.Object
  └ br.ufrn.dimap.jxsc.JFloat32BASE
    └ br.ufrn.dimap.jxsc.JFloat32

public class JFloat32
extends JFloat32BASE

Field Summary	
static long	PI Constante PI
static long	PI_2 Constante PI/2
static long	PI_4 Constante PI/4
static long	PI2 Constante 2*PI

Fields inherited from class br.ufrn.dimap.jxsc.JFloat32BASE
COUNT_LEADING_ZEROS_HIGH, currentExceptionFlags, currentRoundingMode, DEFAULT_NAN, detectTininess, FLAG_DIV_BY_ZERO, FLAG_INEXACT, FLAG_INVALID, FLAG_OVERFLOW, FLAG_UNDERFLOW, ONE, ROUND_DOWN, ROUND_NEAREST_EVEN, ROUND_TO_ZERO, ROUND_UP, SQRT_EVEN_ADJUSTMENTS, SQRT_ODD_ADJUSTMENTS, TININESS_AFTER_ROUNDING, TININESS_BEFORE_ROUNDING, ZERO

Constructor Summary
JFloat32()

Method Summary	
static float	abs(float valor) Valor absoluto de um número
protected static long	abs(long valor)
static float	add(float primeiraParcela, float segundaParcela) Adiciona dois números em ponto flutuante
static float	ceil(float valor)
protected static long	ceil(long valor) Função teto
static float	cos(float valor)

protected static long	cos(long valor)
protected static long	cosInternal(long valor) Adapted from: Cephes Math Library Release 2.7: May, 1998 Copyright 1985, 1990, 1998 by Stephen L.
static float	div(float dividendo, float divisor) Divide dois números em ponto flutuante
(package private) static float	float32ToFloat(long float32) Converte JFloat32BASE para float
(package private) static long	floatToFloat32(float floatJava) Converte float para JFloat32BASE
static float	floor(float valor)
protected static long	floor(long valor) Função Piso floor(x) = -ceil(-x)
static float	mod(float dividendo, float divisor) x mod y = x - y * floor(x/y) se y não é 0 x mod 0 = x onde x e y são números reais x mod 0 = x Nota: x rem y = x - y * trunc(x/y) x mod y = x - y * floor(x/y)
protected static long	mod(long dividendo, long divisor)
static float	mul(float multiplicando, float multiplicador) Multiplica dois números em ponto flutuante
static float	neg(float valor) Função inversora de sinal
protected static long	neg(long valor)
static float	nextfp(float valor)
static float	prevfp(float valor)
static float	rem(float dividendo, float divisor) Calcula o a função resto de dois números em ponto flutuante
static float	sin(float valor) Função seno
protected static long	sin(long valor) Função seno
protected static long	sinInternal(long valor) Baseada na função seno da biblioteca Real.java Originalmente adaptada de Cephes Math Library Release 2.7: May, 1998 Copyright 1985, 1990, 1998 by Stephen L.
static float	sqrt(float primeiraParcela) Calcula a raiz quadrada de um número em ponto flutuante
static float	sub(float primeiraParcela, float segundaParcela) Subtrai dois números em ponto flutuante
static float	tan(float valor)

protected static long	tan(long valor) Calculates the trigonometric tangent of this Real.
static int	toFloatBits(int sign, int exponent, long mantissa) Função extraída da classe Real

Methods inherited from class br.ufrn.dimap.jxsc.JFloat32BASE
add, add64, addFloat32Sigs, commonNaNToFloat32, countLeadingZeros32, div, eq, eqSignaling, estimateDiv64To32, estimateSqrt32, extractFloat32Exp, extractFloat32Frac, extractFloat32Sign, float32ToCommonNaN, ge, getRoundingMode, getSinalizadoresExcecao, gt, int32ToFloat32, isNaN, isSNaN, le, leQuiet, lt, ltQuiet, mul, mul32To64, neq, normalizeFloat32Subnormal, normalizeRoundAndPackFloat32, packFloat32, propagateFloat32NaN, raiseException, rem, roundAndPackFloat32, roundToInt, setExceptionFlags, setRoundingMode, shift32RightJamming, shortShift64Left, sqrt, sub, sub64, subFloat32Sigs, toInt32, toInt32RoundToZero

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

PI_4
public static final long PI_4
Constante PI/4

PI_2
public static final long PI_2
Constante PI/2

PI
public static final long PI
Constante PI

PI2
public static final long PI2
Constante 2*PI

Constructor Detail

JFloat32
public JFloat32()

Method Detail

floatToFloat32
static long floatToFloat32(float floatJava)
Converte float para JFloat32BASE
Parameters:
floatJava -

---

float32ToFloat
static float float32ToFloat(long float32)
Converte JFloat32BASE para float
Parameters:
float32 -

---

add
public static float add(float primeiraParcela,
float segundaParcela)
Adiciona dois números em ponto flutuante
Parameters:
primeiraParcela -

segundaParcela -
Returns:
soma de duas parcelas

---

sub
public static float sub(float primeiraParcela,
float segundaParcela)
Subtrai dois números em ponto flutuante
Parameters:
primeiraParcela -
segundaParcela -
Returns:
resto da subtração de dois números

---

mul
public static float mul(float multiplicando,
float multiplicador)
Multiplica dois números em ponto flutuante
Parameters:
multiplicando -
multiplicador -
Returns:
produto de dois termos

---

div
public static float div(float dividendo,
float divisor)
Divide dois números em ponto flutuante
Parameters:
dividendo -
divisor -
Returns:
quociente da divisão de dois números

---

rem
public static float rem(float dividendo,
float divisor)
Calcula o a função resto de dois números em ponto flutuante
Parameters:
dividendo -
divisor -
Returns:
resto da divisão de dois números

---

sqrt
public static float sqrt(float primeiraParcela)
Calcula a raiz quadrada de um número em ponto flutuante
Parameters:
primeiraParcela -
Returns:
raiz quadrada de um número

---

floor
protected static long floor(long valor)
Função Piso floor(x) = -ceil(-x)
Parameters:
valor -
Returns:
Retorna o maior inteiro menor que ou igual a valor

---

floor
public static float floor(float valor)

---

ceil
protected static long ceil(long valor)
Função teto
Parameters:
valor -
Returns:
Retorna o menor inteiro maior que ou igual a valor arredondado

---

ceil
public static float ceil(float valor)

---

mod
protected static long mod(long dividendo,
long divisor)

---

mod
public static float mod(float dividendo,
float divisor)
x mod y = x - y * floor(x/y) se y não é 0 x mod 0 = x onde x e y são números reais x mod 0 = x Nota: x rem y = x - y * trunc(x/y) x mod y = x - y * floor(x/y)
Parameters:
dividendo -
divisor -
Returns:
Retorna o resultado da função módulo

---

neg
protected static long neg(long valor)

---

neg
public static float neg(float valor)
Função inversora de sinal
Parameters:
valor -
Returns:
o valor negado

---

abs
protected static long abs(long valor)

---

abs
public static float abs(float valor)
Valor absoluto de um número
Parameters:
valor -
Returns:
valor absoluto de um número

---

sin
public static float sin(float valor)
Função seno
Parameters:
valor -
Returns:
seno de um número

---

sin

protected static long sin(long valor)
Função seno
Parameters:
valor -
Returns:
seno de um número

---

sinInternal
protected static long sinInternal(long valor)
Baseada na função seno da biblioteca Real.java Originalmente adaptada de Cephes Math Library Release 2.7: May, 1998 Copyright 1985, 1990, 1998 by Stephen L. Moshier X

---

cos
protected static long cos(long valor)

---

cos
public static float cos(float valor)

---

cosInternal
protected static long cosInternal(long valor)
Adapted from: Cephes Math Library Release 2.7: May, 1998 Copyright 1985, 1990, 1998 by Stephen L. Moshier sinl.c long double cosl(long double x); X

---

tan
protected static long tan(long valor)
Calculates the trigonometric tangent of this Real. Replaces the contents of this Real with the result. The input value is treated as an angle measured in radians.
Parameters:
valor -

---

tan
public static float tan(float valor)

---

toFloatBits
public static int toFloatBits(int sign,
int exponent,
long mantissa)
Função extraída da classe Real
Parameters:
sinal -
exponent -
mantissa -

---

nextfp
public static float nextfp(float valor)

---

prevfp
public static float prevfp(float valor)

10.2. Documentação de referência da classe JFloat32Base

br.ufrn.dimap.jxsc
Class JFloat32BASE
java.lang.Object
└ br.ufrn.dimap.jxsc.JFloat32BASE
Direct Known Subclasses:
JFloat32

---

public class JFloat32BASE
extends java.lang.Object

Field Summary	
protected static int[]	COUNT_LEADING_ZEROS_HIGH
protected static int	currentExceptionFlags Flags de exceção
protected static int	currentRoundingMode Modo de arredondamento de ponto flutuante
static long	DEFAULT_NAN Formato de NaN canônico interno
static int	detectTininess Modo de deteção de tininess, estaticamente iniciado para o valor padrão.
static int	FLAG_DIV_BY_ZERO Divisão por zero
static int	FLAG_INEXACT Operação inexata
static int	FLAG_INVALID Operação inválida
static int	FLAG_OVERFLOW Overflow
static int	FLAG_UNDERFLOW Underflow
static long	ONE
static int	ROUND_DOWN Arredondamento para baixo
static int	ROUND_NEAREST_EVEN Arredondamento para o par mais próximo
static int	ROUND_TO_ZERO Arredondamento para zero
static int	ROUND_UP Arredondamento para cima
protected static int[]	SQRT_EVEN_ADJUSTMENTS
protected static int[]	SQRT_ODD_ADJUSTMENTS
static int	TININESS_AFTER_ROUNDING
static int	TININESS_BEFORE_ROUNDING
static long	ZERO

Constructor Summary
JFloat32BASE()

Method Summary	
protected static long	add(long primeiraParcela, long segundaParcela) Retorna o resultado da adição de dois valores em ponto flutuante.
protected static RetLongLong	add64(long a0, long a1, long b0, long b1) Adiciona o valor de 64 bits formado pela concatenação de a0 e a1 ao valor de 64 bits formado por b0 e b1.
protected static long	addFloat32Sigs(long primeiraParcela, long segundaParcela, int sinal) Retorna o resultado da adição de valores absolutos de dois números em ponto flutuante de precisão simples.
protected static long	commonNaNToFloat32(NaNComum a) Retorna o resultado da conversão do NaN canônico valor para o formato de ponto flutuante de precisão simples.
protected static int	countLeadingZeros32(long valor) Conta o número de zeros existentes antes do bit igual a 1 mais significante de valor.
protected static long	div(long dividendo, long divisor) Retorna o resultado da divisão de dois valores em ponto flutuante.
protected static boolean	eq(long primeiroPontoFlutuante, long segundoPontoFlutuante) Retorna verdadeiro se os dois valores são iguais.
protected static boolean	eqSignaling(long primeiroPontoFlutuante, long segundoPontoFlutuante) Retorna verdadeiro se os valores são iguais.
protected static long	estimateDiv64To32(long a0, long a1, long b) Retorna uma aproximação para o quociente de 32 bits obtido pela divisão de um valor de 64 bits formado pela concatenação de a0 e a1 por um valor de 32 bits em b.
protected static long	estimateSqrt32(int aExp, long a) Retorna uma aproximação para a raiz quadrada do significando de 32 bits dado por a.
protected static int	extractFloat32Exp(long pontoFlutuante) Retorna os bits do expoente do número em ponto flutuante
protected static long	extractFloat32Frac(long pontoFlutuante) Retorna os bits da parte fracionária de um número em ponto flutuante.
protected static int	extractFloat32Sign(long pontoFlutuante) Retorna o bit do sinal do número em ponto flutuante
protected static NaNComum	float32ToCommonNaN(long pontoFlutuante) Retorna o resultado da conversão do NaN de ponto flutuante de precisão simples valor para o formato NaN canonico.
protected static boolean	ge(long primeiroPontoFlutuante, long segundoPontoFlutuante) Retorna verdadeiro se o primeiro valor é maior que ou igual ao segundo
static int	getRoundingMode() Retorna o tipo de arredondamento
static int	getSinalizadoresExcecao() Retorna os flags de exceção
protected static boolean	gt(long primeiroPontoFlutuante, long segundoPontoFlutuante) Retorna verdadeiro se o primeiro parâmetro é maior que o segundo
protected static long	int32ToFloat32(int inteiro) Retorna o resultado da conversão de um inteiro de 32 bits no formato de complemento de dois para o formato de ponto flutuante de precisão simples.

protected static boolean	isNaN(long pontoFlutuante) Retorna true se o parâmetro é um NaN
protected static boolean	isSNaN(long pontoFlutuante) Retorna true se o valor em ponto flutuante a é um NaN sinalizante
protected static boolean	le(long primeiroPontoFlutuante, long segundoPontoFlutuante) Retorna verdadeiro se o primeiro valor é menor que ou igual ao segundo.
protected static boolean	leQuiet(long primeiroPontoFlutuante, long segundoPontoFlutuante) Retorna verdadeiro se o primeiro valor é menor que ou igual ao segundo.
protected static boolean	lt(long primeiroPontoFlutuante, long segundoPontoFlutuante) Retorna verdadeiro se o primeiro valor é menor que o primeiro.
protected static boolean	ltQuiet(long primeiroPontoFlutuante, long segundoPontoFlutuante) Retorna verdadeiro se o primeiro valor é menor que o segundo.
protected static long	mul(long multiplicando, long multiplicador) Retorna o resultado da multiplicação de valores em ponto flutuante.
protected static RetLongLong	mul32To64(long multiplicando, long multiplicador) Multiplica a por b para obter um produto de 64 bits.
protected static boolean	neq(long primeiroPontoFlutuante, long segundoPontoFlutuante) Retorna verdadeiro se os dois valores são diferentes
protected static RetIntLong	normalizeFloat32Subnormal(long significando) Normaliza o valor de ponto flutuante de precisão simples subnormal representado pelo significando denormalizado aSig.
protected static long	normalizeRoundAndPackFloat32(int sinal, int expoente, long significando) Recebe um ponto flutuante abstrato representado por sinal, expoente e significando e retorna o valor de ponto flutuante correspondente.
protected static long	packFloat32(int sinal, int expoente, long significando) Empacota o sinal, expoente e significando em um valor de ponto flutuante.
protected static long	propagateFloat32NaN(long primeiroPontoFluante, long segundoPontoFlutuante) Toma dois valores de ponto flutuante a e b, um dos quais é um NaN e retorna o NaN apropriado.
static void	raiseException(int sinalizadores) Levanta as exceções especificados por 'sinalizadores'.
protected static long	rem(long dividendo, long divisor) Retorna o resto de um valor de precisão simples em relação a outro valor
protected static long	roundAndPackFloat32(int sinal, int expoente, long significando) Recebe um valor de ponto flutuante desmembrado em sinal, expoente e significando e retorna o valor apropriado em ponto flutuante correspondente a entrada abstrata.
protected static long	roundToInt(long pontoFlutuante) Arredonda o valor de ponto flutuante de precisão simples para um inteiro e retorna o resultado como um valor de ponto flutuante de precisão simples.
static void	setExceptionFlags(int sinalizadoresExcecao) Especifica os flags de exceção
static void	setRoundingMode(int modoArredondamento) Especifica o tipo de arredondamento
protected static long	shift32RightJamming(long inteiro, int contador) Desloca 'numero' para direita pelo número de bits dado por 'contador'.

protected static RetLongLong	shortShift64Left(long a0, long a1, int contador) Desloca para esquerda o valor de 64 bits formado pela concatenação de a0 e a1, pelo número de bits especificado por 'contador'.
protected static long	sqrt(long pontoFlutuante) Retorna a raiz quadrada de um valor de ponto flutuante de precisão simples.
protected static long	sub(long minuendo, long subtraendo) Retorna o resultado da subtração de dois valores de ponto flutuante de precisão simples.
protected static RetLongLong	sub64(long a0, long a1, long b0, long b1) Subtrai o valor de 64 bits formado pela concatenação de b0 e b1 do valor de 64 bits formado pela concatenação de a0 e a1.
protected static long	subFloat32Sigs(long minuendo, long subtraendo, int sinal) Retorna o resultado da subtração de valores absolutos de dois valores em ponto flutuante.
protected static int	toInt32(long pontoFlutuante) Retorna o resultada da conversão de um valor em ponto flutuante de precisão simples para o formato inteiro de complemento de dois de 32 bits.
protected static int	toInt32RoundToZero(long pontoFlutuante) Retorna o resultado da conversão de um ponto flutuante de precisão simples em um inteiro de 32 bits no formato de complemento de dois.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

TININESS_AFTER_ROUNDING
public static final int TININESS_AFTER_ROUNDING
See Also:
Constant Field Values

TININESS_BEFORE_ROUNDING
public static final int TININESS_BEFORE_ROUNDING
See Also:
Constant Field Values

ROUND_NEAREST_EVEN
public static final int ROUND_NEAREST_EVEN
Arredondamento para o par mais próximo
See Also:
Constant Field Values

ROUND_DOWN
public static final int ROUND_DOWN
Arredondamento para baixo
See Also:
Constant Field Values

ROUND_UP
public static final int ROUND_UP
Arredondamento para cima
See Also:
Constant Field Values

ROUND_TO_ZERO
public static final int ROUND_TO_ZERO

Arredondamento para zero
See Also:
Constant Field Values

FLAG_INVALID
public static final int FLAG_INVALID
Operação inválida
See Also:
Constant Field Values

FLAG_DIV_BY_ZERO
public static final int FLAG_DIV_BY_ZERO
Divisão por zero
See Also:
Constant Field Values

FLAG_OVERFLOW
public static final int FLAG_OVERFLOW
Overflow
See Also:
Constant Field Values

FLAG_UNDERFLOW
public static final int FLAG_UNDERFLOW
Underflow
See Also:
Constant Field Values

FLAG_INEXACT
public static final int FLAG_INEXACT
Operação inexata
See Also:
Constant Field Values

---

currentRoundingMode
protected static int currentRoundingMode
Modo de arredondamento de ponto flutuante

---

currentExceptionFlags
protected static int currentExceptionFlags
Flags de exceção

SQRT_ODD_ADJUSTMENTS
protected static final int[] SQRT_ODD_ADJUSTMENTS

SQRT_EVEN_ADJUSTMENTS
protected static final int[] SQRT_EVEN_ADJUSTMENTS

COUNT_LEADING_ZEROS_HIGH
protected static final int[] COUNT_LEADING_ZEROS_HIGH

ZERO
public static final long ZERO
See Also:
Constant Field Values

ONE
public static final long ONE
See Also:
Constant Field Values

---

detectTininess
public static int detectTininess
Modo de deteção de tininess, estaticamente iniciado para o valor padrão. Relacionado a funções e definições para determinar:
se o teste tininess para underflow deve ser detectado antes ou após o arredondamento por padrão;
que (se algo) acontece quando exceções são levantadas;
como NaNs sinalizantes são distinguidos de NaNs quietos;
o valor padrão gerado para NaNs quietos;
como NaNs são propagados das entradas para a saída.

DEFAULT_NAN
public static long DEFAULT_NAN
Formato de NaN canônico interno

Constructor Detail

JFloat32BASE
public JFloat32BASE()

Method Detail

getSinalizadoresExcecao
public static int getSinalizadoresExcecao()
Retorna os flags de exceção

---

setExceptionFlags
public static void setExceptionFlags(int sinalizadoresExcecao)
Especifica os flags de exceção
Parameters:
currentExceptionFlags -

---

getRoundingMode
public static int getRoundingMode()
Retorna o tipo de arredondamento

---

setRoundingMode
public static void setRoundingMode(int modoArredondamento)
Especifica o tipo de arredondamento
Parameters:
currentRoundingMode -

---

shift32RightJamming
protected static long shift32RightJamming(long inteiro,
int contador)
Desloca 'numero' para direita pelo número de bits dado por 'contador'. Se quaisquer bits diferentes de zero forem deslocados além do bit 0, eles serão 'espremidos' no bit menos significativo do resultado por deixá-lo igual a 1. O valor de 'contador' pode ser arbitrariamente largo, em particular, se 'contador' é maior que 32, o resultado será zero ou 1, dependendo se 'numero' é zero ou um. O resultado será retornado.

---

shortShift64Left
protected static RetLongLong shortShift64Left(long a0,
long a1,
int contador)
Desloca para esquerda o valor de 64 bits formado pela concatenação de a0 e a1, pelo número de bits especificado por 'contador'. Quaisquer bits deslocado para fora da faixa de bits será perdido. O valor do contador deve ser menor que 32. O resultado será retornado quebrado em duas variáveis de 32 bits armazenados em um objeto RetLongLong.

---

add64
protected static RetLongLong add64(long a0,

long a1,
long b0,
long b1)

Adiciona o valor de 64 bits formado pela concatenação de a0 e a1 ao valor de 64 bits formado por b0 e b1. A adição é módulo 2^64, então, qualquer 'vai-um' será perdido'. O resultado será retornado por um objeto RetLongLong.

---

sub64

protected static RetLongLong sub64(long a0,
long a1,
long b0,
long b1)

Subtrai o valor de 64 bits formado pela concatenação de b0 e b1 do valor de 64 bits formado pela concatenação de a0 e a1. A subtração é módulo 2^64, então qualquer 'vem-um' será perdido. O resultado é retornado em um objeto RetLongLong.

---

mul32To64

protected static RetLongLong mul32To64(long multiplicando,
long multiplicador)

Multiplica a por b para obter um produto de 64 bits. O produto é quebrado em dois valores de 32 bits os quais são retornados por um objeto RetLongLong.

---

estimateDiv64To32

protected static long estimateDiv64To32(long a0,
long a1,
long b)

Retorna uma aproximação para o quociente de 32 bits obtido pela divisão de um valor de 64 bits formado pela concatenação de a0 e a1 por um valor de 32 bits em b. O divisor deve ser até 2^31. Se q é o quociente exato truncado em torno de zero, a aproximação retornada ficará entre q e q+2 inclusive. Se o quociente exato é mais largo que 32 bits, o maior inteiro não sinalizado de 32 bits será retornado.

---

estimateSqrt32

protected static long estimateSqrt32(int aExp,
long a)

Retorna uma aproximação para a raiz quadrada do significando de 32 bits dado por a. Considerado como um inteiro, a deve ser até 2^31. Se o bit 0 de aExp (o bit menos significante) é 1, o inteiro retornado aproxima-se de 2^31 * sqrt(a/2^31), onde a é considerado um inteiro. Se o bit 0 de aExp é 0, o inteiro retornado aproxima-se de 2^31 * sqrt(a/2^30). Em outro caso, a aproximação retornado estará na faixa de +/- 2 do valor exato.

---

countLeadingZeros32

protected static int countLeadingZeros32(long valor)

Conta o número de zeros existentes antes do bit igual a 1 mais significante de valor. Se valor for zero, 32 será retornado.

---

raiseException

public static void raiseException(int sinalizadores)

Levanta as exceções especificados por 'sinalizadores'.

Interrupções de ponto flutuante poderiam ser definidas aqui se desejado.

Não é atualmente possível alterar o valor de resultado.

---

isNaN

protected static boolean isNaN(long pontoFlutuante)
Retorna true se o parâmetro é um NaN

---

isSNaN
protected static boolean isSNaN(long pontoFlutuante)
Retorna true se o valor em ponto flutuante a é um NaN sinalizante

---

float32ToCommonNaN
protected static NaNComum float32ToCommonNaN(long pontoFlutuante)
Retorna o resultado da conversão do NaN de ponto flutuante de precisão simples valor para o formato NaN canonico.

Se valor é um NaN sinalizante, a exceção de invalidez será levantada.

---

commonNaNToFloat32
protected static long commonNaNToFloat32(NaNComum a)
Retorna o resultado da conversão do NaN canônico valor para o formato de ponto flutuante de precisão simples.

---

propagateFloat32NaN
protected static long propagateFloat32NaN(long primeiroPontoFluante,
long segundoPontoFlutuante)
Toma dois valores de ponto flutuante a e b, um dos quais é um NaN e retorna o NaN apropriado.

Se um deles for um NaN sinalizante, a exceção de invalidez será levantada.

---

extractFloat32Frac
protected static long extractFloat32Frac(long pontoFlutuante)
Retorna os bits da parte fracionária de um número em ponto flutuante.

---

extractFloat32Exp
protected static int extractFloat32Exp(long pontoFlutuante)
Retorna os bits do expoente do número em ponto flutuante

---

extractFloat32Sign
protected static int extractFloat32Sign(long pontoFlutuante)
Retorna o bit do sinal do número em ponto flutuante

---

normalizeFloat32Subnormal
protected static RetIntLong normalizeFloat32Subnormal(long significando)
Normaliza o valor de ponto flutuante de precisão simples subnormal representado pelo significando denormalizado aSig.

O expoente normalizado e o significando são retornados por um objeto RetIntLong.

---

packFloat32
protected static long packFloat32(int sinal,
int expoente,
long significando)
Empacota o sinal, expoente e significando em um valor de ponto flutuante.

Após serem deslocados para as posições apropriadas, os três campos são simplesmente adicionados para formar

o resultado.

Isto significa que qualquer porção do sinal será adicionado ao expoente.

Desde que o significando normalizado propriamente terá uma porção inteira igual a 1, a entrada do expoente deveria ser 1 a menos que o expoente do resultado desejado sempre que o sinal está completo, significando normalizado.

---

roundAndPackFloat32
protected static long roundAndPackFloat32(int sinal,
int expoente,
long significando)
Recebe um valor de ponto flutuante desmembrado em sinal, expoente e significando e retorna o valor apropriado em ponto flutuante correspondente a entrada abstrata.

Ordinariamente, o valor abstrato é simplesmente arredondado e empacotado em um formato de precisão simples, com a exceção de valor inexato levantada se a entrada abstrata não pode ser representada exatamente.

Entretanto, se o valor abstrato é largo demais, as exceções de overflow e inexato serão levantadas e um valor infinito ou o maior valor finito será retornado.

Se o valor abstrato de entrada é pequeno demais, o valor de entrada é arredondado para um número flutuante subnormal e as exceções de underflow e inexato serão levantadas se a entrada abstrata não puder ser representada exatamente como um número subnormal de ponto flutuante de precisão simples.

O signicando de entrada tem seu ponto binário entre os bits 30 e 29, o qual está 7 bits à esquerda da posição usual. Este significando deslocado deve ser normalizado ou menor. Se o significando não está normalizado, o expoente dever ser 0; naquele caso, o resultado retornado é um número subnormal, e não deve requerer arredondamento.

No caso usual, o significando está normalizado, o expoente deve ser 1 a menos que o expoente verdadeiro do ponto flutuante.

A manipulação de underflow e overflow segue o padrão IEEE para aritmética de ponto flutuante.

---

normalizeRoundAndPackFloat32
protected static long normalizeRoundAndPackFloat32(int sinal,
int expoente,
long significando)
Recebe um ponto flutuante abstrato representado por sinal, expoente e significando e retorna o valor de ponto flutuante correspondente.

Esta rotina é como roundAndPackFloat32 exceto que o significando não tem que estar normalizado.

O bit 31 do significando deve ser zero e o expoente deve ser um a menos que o valor verdadeiro do expoente em ponto flutuante.

---

int32ToFloat32
protected static long int32ToFloat32(int inteiro)
Retorna o resultado da conversão de um inteiro de 32 bits no formato de complemento de dois para o formato de ponto flutuante de precisão simples.

A conversão é realizada de acordo com o padrão IEEE para aritmética de ponto flutuante.

---

toInt32
protected static int toInt32(long pontoFlutuante)

Retorna o resultada da conversão de um valor em ponto flutuante de precisão simples para o formato inteiro de complemento de dois de 32 bits. A conversão é feita de acordo com o padrão IEEE para a aritmética binária de ponto flutuante, o qual significa em particular que a conversão é arredondada de acordo com o modo de arredondamento corrente. Se o valor é um NaN, o maior inteiro positivo será retornado. Por outro lado, se ocorrer overflow na conversão, o maior inteiro com o mesmo sinal do valor será retornado.

---

toInt32RoundToZero
protected static int toInt32RoundToZero(long pontoFlutuante)
Retorna o resultado da conversão de um ponto flutuante de precisão simples em um inteiro de 32 bits no formato de complemento de dois.
A conversão é realizada de acordo como o padrão IEEE para aritmética binária de ponto flutuante, exceto que a conversão é sempre arredondada na direção de zero.
Se o ponto flutuante é um NaN, o maior inteiro positivo será retornado.
Caso contrário, se na conversão ocorre overflow, o inteiro mais largo com mesmo sinal do ponto flutuante será retornado.

---

roundToInt
protected static long roundToInt(long pontoFlutuante)
Arredonda o valor de ponto flutuante de precisão simples para um inteiro e retorna o resultado como um valor de ponto flutuante de precisão simples. A operação é realizada de acordo com o padrão IEEE para aritmética binária de ponto flutuante.

---

addFloat32Sigs
protected static long addFloat32Sigs(long primeiraParcela,
long segundaParcela,
int sinal)
Retorna o resultado da adição de valores absolutos de dois números em ponto flutuante de precisão simples. Se o sinal for igual a 1, a soma é negada antes de ser retornada. O sinal é ignorado se o resultado é um NaN. A adição é realizada de acordo com o padrão IEC/IEE para aritmética binária de ponto flutuante.

---

subFloat32Sigs
protected static long subFloat32Sigs(long minuendo,
long subtraendo,
int sinal)
Retorna o resultado da subtração de valores absolutos de dois valores em ponto flutuante. Se o sinal é 1, a diferença será negada antes de ser retornada. O sinal é ignoradao se o resultado é um NaN. A subtração é realizada de acordo com o padrão IEEE para aritmética binária de ponto flutuante.

---

add
protected static long add(long primeiraParcela,
long segundaParcela)
Retorna o resultado da adição de dois valores em ponto flutuante. A operação é realizada de acordo com o padrão IEEE para aritmética binária de ponto flutuante

---

sub
protected static long sub(long minuendo,
long subtraendo)
Retorna o resultado da subtração de dois valores de ponto flutuante de precisão simples.

---

mul
protected static long mul(long multiplicando,
long multiplicador)

Retorna o resultado da multiplicação de valores em ponto flutuante.

---

div
protected static long div(long dividendo,
long divisor)
Retorna o resultado da divisão de dois valores em ponto flutuante.

---

rem
protected static long rem(long dividendo,
long divisor)
Retorna o resto de um valor de precisão simples em relação a outro valor

---

sqrt
protected static long sqrt(long pontoFlutuante)
Retorna a raiz quadrada de um valor de ponto flutuante de precisão simples.

---

eq
protected static boolean eq(long primeiroPontoFlutuante,
long segundoPontoFlutuante)
Retorna verdadeiro se os dois valores são iguais.

---

neq
protected static boolean neq(long primeiroPontoFlutuante,
long segundoPontoFlutuante)
Retorna verdadeiro se os dois valores são diferentes
Parameters:
primeiroPontoFlutuante -
segundoPontoFlutuante -

---

le
protected static boolean le(long primeiroPontoFlutuante,
long segundoPontoFlutuante)
Retorna verdadeiro se o primeiro valor é menor que ou igual ao segundo.

---

gt
protected static boolean gt(long primeiroPontoFlutuante,
long segundoPontoFlutuante)
Retorna verdadeiro se o primeiro parâmetro é maior que o segundo
Parameters:
primeiroPontoFlutuante -
segundoPontoFlutuante -

---

lt
protected static boolean lt(long primeiroPontoFlutuante,
long segundoPontoFlutuante)
Retorna verdadeiro se o primeiro valor é menor que o primeiro.

---

ge
protected static boolean ge(long primeiroPontoFlutuante,
long segundoPontoFlutuante)
Retorna verdadeiro se o primeiro valor é maior que ou igual ao segundo
Parameters:

primeiroPontoFlutuante -
segundoPontoFlutuante -

---

eqSignaling
protected static boolean eqSignaling(long primeiroPontoFlutuante,
long segundoPontoFlutuante)
Retorna verdadeiro se os valores são iguais.

---

leQuiet
protected static boolean leQuiet(long primeiroPontoFlutuante,
long segundoPontoFlutuante)
Retorna verdadeiro se o primeiro valor é menor que ou igual ao segundo.

---

ltQuiet
protected static boolean ltQuiet(long primeiroPontoFlutuante,
long segundoPontoFlutuante)
Retorna verdadeiro se o primeiro valor é menor que o segundo.

10.3. Teste de exatidão de JFloat32, Float e Double em comparação ao tipo de ponto flutuante de dupla precisão da biblioteca C-XSC

```
package br.ufrn.dimap.jxsc.testes;

import java.util.Random;
import br.ufrn.dimap.cxsc.CXSC;
import br.ufrn.dimap.jxsc.JFloat32;

public class Teste01Precisao extends Teste {

  public void testPrecisao() {
    println("Erro maximo relativo de JFloat em relacao a CXSC");
    print("Operacao");
    print("Tipo");
    print("Maximo");
    print("Medio");
    print("Variancia");
    println("Desvio padrao");
    JFloat32.setRoundingMode(JFloat32.ROUND_NEAREST_EVEN);

    executarOperacao('+');
    executarOperacao('-');
    executarOperacao('*');
    executarOperacao('/');
  }

  private void executarOperacao(char operation) {
    Random rnd1 = new Random();
    Random rnd2 = new Random();

    double maxErrorDouble = 0;
    double maxErrorJFloat = 0;
    double maxErrorNativeFloat = 0;

    float firstFloat;
    float secondFloat;
    double firstDouble;
    double secondDouble;

    double resultDouble = 0;
    double resultNativeFloat = 0;
    double resultJFloat = 0;
    double resultCXSC = 0;

    long n = 0;
    double totalDouble = 0;
    double totalJFloat = 0;
    double totalNativeFloat = 0;

    for (int j = 1; j < 10000000; j++) {
      firstFloat = Float.intBitsToFloat(rnd1.nextInt());
      secondFloat = Float.intBitsToFloat(rnd2.nextInt());

      if (Float.isNaN(firstFloat) || Float.isNaN(secondFloat))
        continue;

      firstDouble = firstFloat;
      secondDouble = secondFloat;
```

```
      switch (operation) {
      case '+':
        resultCXSC = CXSC.add(firstDouble, secondDouble);
        resultDouble = firstDouble + secondDouble;
        resultNativeFloat = (firstFloat + secondFloat);
        resultJFloat = JFloat32.add(firstFloat, secondFloat);
        break;
      case '-':
        resultCXSC = CXSC.sub(firstDouble, secondDouble);
        resultDouble = firstDouble - secondDouble;
        resultNativeFloat = (firstFloat - secondFloat);
        resultJFloat = JFloat32.sub(firstFloat, secondFloat);
        break;
      case '*':
        resultCXSC = CXSC.mul(firstDouble, secondDouble);
        resultDouble = firstDouble * secondDouble;
        resultNativeFloat = (firstFloat * secondFloat);
        resultJFloat = JFloat32.mul(firstFloat, secondFloat);
        break;
      case '/':
        resultCXSC = CXSC.div(firstDouble, secondDouble);
        resultDouble = firstDouble / secondDouble;
        resultNativeFloat = (firstFloat / secondFloat);
        resultJFloat = JFloat32.div(firstFloat, secondFloat);
        break;
      }

      if (Double.isNaN(resultCXSC) || Double.isNaN(resultDouble) ||
Double.isNaN(resultNativeFloat)
          || Double.isNaN(resultJFloat) || Double.isInfinite(resultDouble)
          || Double.isInfinite(resultNativeFloat) ||
Double.isInfinite(resultJFloat))
        continue;

      double errorDouble = Math.abs(resultDouble - resultCXSC) /
Math.abs(resultCXSC);
      double errorJFloat = Math.abs(resultJFloat - resultCXSC) /
Math.abs(resultCXSC);
      double errorNativeFloat = Math.abs(resultNativeFloat - resultCXSC) /
Math.abs(resultCXSC);

      if (resultJFloat == resultNativeFloat) {
        continue;
      }

      if (Double.isNaN(errorDouble) || Double.isNaN(errorJFloat) ||
Double.isNaN(errorNativeFloat)
          || Double.isInfinite(errorDouble) || Double.isInfinite(errorJFloat)
          || Double.isInfinite(errorNativeFloat))
        continue;

      n++;
      totalDouble += errorDouble;
      totalJFloat += errorJFloat;
      totalNativeFloat += errorNativeFloat;

      if (errorDouble > maxErrorDouble)
        maxErrorDouble = errorDouble;

      if (errorJFloat > maxErrorJFloat)
        maxErrorJFloat = errorJFloat;

      if (errorNativeFloat > maxErrorNativeFloat)
        maxErrorNativeFloat = errorNativeFloat;

    }
    double mediaDouble = (totalDouble / n);
    double total2Double = 0;
```

```
    double mediaJFloat = (totalJFloat / n);
    double total2JFloat = 0;

    double mediaNativeFloat = (totalNativeFloat / n);
    double total2NativeFloat = 0;

    n = 0;
    for (int j = 1; j < 10000000; j++) {
      firstFloat = Float.intBitsToFloat(rnd1.nextInt());
      secondFloat = Float.intBitsToFloat(rnd2.nextInt());

      if (Float.isNaN(firstFloat) || Float.isNaN(secondFloat))
        continue;

      firstDouble = firstFloat;
      secondDouble = secondFloat;

      switch (operation) {
      case '+':
        resultCXSC = CXSC.add(firstDouble, secondDouble);
        resultDouble = firstDouble + secondDouble;
        resultNativeFloat = (firstFloat + secondFloat);
        resultJFloat = JFloat32.add(firstFloat, secondFloat);
        break;
      case '-':
        resultCXSC = CXSC.sub(firstDouble, secondDouble);
        resultDouble = firstDouble - secondDouble;
        resultNativeFloat = (firstFloat - secondFloat);
        resultJFloat = JFloat32.sub(firstFloat, secondFloat);
        break;
      case '*':
        resultCXSC = CXSC.mul(firstDouble, secondDouble);
        resultDouble = firstDouble * secondDouble;
        resultNativeFloat = (firstFloat * secondFloat);
        resultJFloat = JFloat32.mul(firstFloat, secondFloat);
        break;
      case '/':
        resultCXSC = CXSC.div(firstDouble, secondDouble);
        resultDouble = firstDouble / secondDouble;
        resultNativeFloat = (firstFloat / secondFloat);
        resultJFloat = JFloat32.div(firstFloat, secondFloat);
        break;
      }

      if (Double.isNaN(resultCXSC) || Double.isNaN(resultDouble) ||
Double.isNaN(resultNativeFloat)
          || Double.isNaN(resultJFloat) || Double.isInfinite(resultCXSC)
          || Double.isInfinite(resultDouble) ||
Double.isInfinite(resultNativeFloat)
          || Double.isInfinite(resultJFloat))
        continue;

      double errorDouble = Math.abs(resultDouble - resultCXSC) /
Math.abs(resultCXSC);
      double errorJFloat = Math.abs(resultJFloat - resultCXSC) /
Math.abs(resultCXSC);
      double errorNativeFloat = Math.abs(resultNativeFloat - resultCXSC) /
Math.abs(resultCXSC);

      if (resultJFloat == resultNativeFloat) {
        continue;
      }

      if (Double.isNaN(errorDouble) || Double.isNaN(errorJFloat) ||
Double.isNaN(errorNativeFloat)
          || Double.isInfinite(errorDouble) || Double.isInfinite(errorJFloat)
          || Double.isInfinite(errorNativeFloat))
```

```
        continue;

      n++;
      total2Double += ((errorDouble - mediaDouble) * (errorDouble - mediaDouble));
      total2JFloat += ((errorJFloat - mediaJFloat) * (errorJFloat - mediaJFloat));
      total2NativeFloat += ((errorNativeFloat - mediaNativeFloat) *
(errorNativeFloat - mediaNativeFloat));
    }
    double varianciaDouble = total2Double / n;
    double desviopadraoDouble = Math.sqrt(varianciaDouble);

    double varianciaJFloat = total2JFloat / n;
    double desviopadraoJFloat = Math.sqrt(varianciaJFloat);

    double varianciaNativeFloat = total2NativeFloat / n;
    double desviopadraoNativeFloat = Math.sqrt(varianciaNativeFloat);

    System.out.println("Operação " + operation);

    println();

    print(operation + "");
    print("Double");
    print(maxErrorDouble);
    print(mediaDouble);
    print(varianciaDouble);
    println(desviopadraoDouble);

    print(operation + "");
    print("JFloat");
    print(maxErrorJFloat);
    print(mediaJFloat);
    print(varianciaJFloat);
    println(desviopadraoJFloat);

    print(operation + "");
    print("Float");
    print(maxErrorNativeFloat);
    print(mediaNativeFloat);
    print(varianciaNativeFloat);
    println(desviopadraoNativeFloat);
  }
}
```

10.4. Teste gráfico da função JFloat32.sin()

```
package br.ufrn.dimap.jxsc.testes;
import org.apache.log4j.Logger;
import br.ufrn.dimap.jxsc.JFloat32;

public class TestarSeno {
  private static Logger log = Logger.getLogger("trace");

  public static void main(String[] args) {
    float i;
    double PI2 = 2f * Math.PI;

    for (i = 0f; i <= PI2; i += 0.01) {
      log.debug(i + ";" + Math.sin(i) + ";" + JFloat32.sin(i));
    }
  }
}
```

10.5. Teste da exatidão da função JFloat32.sin()

```
package br.ufrn.dimap.jxsc.testes;

import br.ufrn.dimap.cxsc.CXSC;
import br.ufrn.dimap.jxsc.JFloat32;

public class Teste02PrecisaoSeno extends Teste {

  final static double PI2 = 2f * Math.PI;

  public void testPrecisaoSeno() {
    float i;
    long n = 0;

    double maxDouble = 0;
    double maxJFloat = 0;
    double maxNativo = 0;

    double totalDouble = 0;
    double totalJFloat = 0;
    double totalNativo = 0;

    double erroDouble;
    double erroJFloat;
    double erroNativo;

    println("Erro maximo da funcao seno de JFloat em relacao a CXSC");

    print("Operacao");
    print("Tipo");
    print("Maximo");
    print("Media");
    print("Variancia");
    println("Desvio padrao");

    for (i = 0f; i <= PI2; i += 0.01) {
```

```
  n++;
  erroDouble = Math.abs(CXSC.sin(i) - Math.sin(i));
  erroJFloat = Math.abs(CXSC.sin(i) - JFloat32.sin(i));
  erroNativo = Math.abs(CXSC.sin(i) - (float) Math.sin(i));

  maxDouble = Math.max(maxDouble, erroDouble);
  maxJFloat = Math.max(maxJFloat, erroJFloat);
  maxNativo = Math.max(maxNativo, erroNativo);

  totalDouble += erroDouble;
  totalJFloat += erroJFloat;
  totalNativo += erroNativo;
}

double mediaDouble = (totalDouble / n);
double total2Double = 0;

double mediaJFloat = (totalJFloat / n);
double total2JFloat = 0;

double mediaNativo = (totalNativo / n);
double total2Nativo = 0;

for (i = 0f; i <= PI2; i += 0.01) {
  erroDouble = Math.abs(CXSC.sin(i) - Math.sin(i));
  total2Double += ((erroDouble - mediaDouble) * (erroDouble - mediaDouble));

  erroJFloat = Math.abs(CXSC.sin(i) - JFloat32.sin(i));
  total2JFloat += ((erroJFloat - mediaJFloat) * (erroJFloat - mediaJFloat));

  erroNativo = Math.abs(CXSC.sin(i) - (float) Math.sin(i));
  total2Nativo += ((erroNativo - mediaNativo) * (erroNativo - mediaNativo));
}

double varianciaDouble = total2Double / n;
double desviopadraoDouble = Math.sqrt(varianciaDouble);

double varianciaJFloat = total2JFloat / n;
double desviopadraoJFloat = Math.sqrt(varianciaJFloat);

double varianciaNativo = total2Nativo / n;
double desviopadraoNativo = Math.sqrt(varianciaNativo);

println();

print("Seno");
print("Double");
print(maxDouble);
print(mediaDouble);
print(varianciaDouble);
println(desviopadraoDouble);

print("Seno");
print("JFloat");
print(maxJFloat);
print(mediaJFloat);
print(varianciaJFloat);
println(desviopadraoJFloat);

print("Seno");
print("Nativo");
print(maxNativo);
print(mediaNativo);
```

```
    print(varianciaNativo);
    println(desviopadraoNativo);
  }

}
```

10.6. Teste da velocidade da função JFloat32.sin()

```
package br.ufrn.dimap.jxsc.testes;

import br.ufrn.dimap.cxsc.CXSC;
import br.ufrn.dimap.jxsc.JFloat32;

public class Teste03VelocidadeSeno extends Teste {
  double PI2 = 2f * Math.PI;

  public void testVelocidadeSeno() {
    println("Comparativo da velocidade das bibliotecas");
    print("Biblioteca");
    print("Tempo (ms)");
    println("Somatorio");

    timerJava();
    timerCXSC();
    timerJFloat32();
  }

  private void timerJava() {
    long time1;
    long time2;
    float i;
    int j;
    double result;
    // Velocidade de execucao da funcao Math.sin()
    result = 0;
    time1 = System.currentTimeMillis();
    for (j = 0; j < 10000; j++)
      for (i = 0f; i <= PI2; i += 0.01) {
        result = result + Math.sin(i);
      }
    time2 = System.currentTimeMillis();

    print("Java");
    print((time2 - time1)+"");
    println(result);
  }

  private void timerJFloat32() {
    long time1;
    long time2;
    float i;
    int j;
    double result;
    // Velocidade de execucao da funcao JFloat32.sin()
    result = 0;
    time1 = System.currentTimeMillis();
    for (j = 0; j < 10000; j++)
      for (i = 0f; i <= PI2; i += 0.01) {
        result = result + JFloat32.sin(i);
```

```
      }
    time2 = System.currentTimeMillis();

    print("JFloat32");
    print((time2 - time1)+"");
    println(result);
  }

  private void timerCXSC() {
    long time1;
    long time2;
    float i;
    int j;
    double result;
    // Velocidade de execucao da funcao CXSC.sin()
    result = 0;
    time1 = System.currentTimeMillis();
    for (j = 0; j < 10000; j++)
      for (i = 0f; i <= PI2; i += 0.01) {
        result = result + CXSC.sin(i);
      }
    time2 = System.currentTimeMillis();

    print("CXSC");
    print((time2 - time1)+"");
    println(result);
  }
}
```

10.7. Teste da função de Rump

```
package br.ufrn.dimap.jxsc.testes;

import br.ufrn.dimap.cxsc.CXSC;
import br.ufrn.dimap.jxsc.Formatar;
import br.ufrn.dimap.jxsc.JFloat32;

public class Teste04Rump extends Teste {

  public void testRump() {
    println("Teste de RUMP executado para as diversas bibliotecas");

    print("");
    print("");
    println("_3210987654321098765432109876543210987654321098765432109876543210");

    print("Tipo");
    print("Resultado");
    println("_SEEEEEEEEEEEEMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM");

    print("Correto");
    println("=\"-0.82739605994682135 +/- 5 E-17\"");

    rumpCXSC();
    rumpDouble();
    rumpFloat();
    rumpJFloat();
  }
```

```
private void rumpCXSC() {
  double a = 77617.0f;
  double b = 33096.0f;

  double a2 = CXSC.mul(a, a);
  double b2 = CXSC.mul(b, b);
  double b3 = CXSC.mul(b2, b);
  double b4 = CXSC.mul(b3, b);
  double b5 = CXSC.mul(b4, b);
  double b6 = CXSC.mul(b5, b);
  double b7 = CXSC.mul(b6, b);
  double b8 = CXSC.mul(b7, b);

  double y = CXSC.mul(333.75f, b6); // y = 333.75f * b6
  double temp = CXSC.mul(11.0f, a2); // temp = 11.0f * a2
  temp = CXSC.mul(temp, b2); // temp = (11.0f * a2) * b2
  temp = CXSC.sub(temp, b6); // temp = ((11.0f * a2) * b2) - b6
  double temp2 = CXSC.mul(121f, b4); // temp2 = 121f * b4
  temp = CXSC.sub(temp, temp2); // temp = (((11.0f * a2) * b2) - b6) -
  // (121f * b4)
  temp = CXSC.sub(temp, 2f); // temp = ((((11.0f * a2) * b2) - b6) -
  // (121f * b4)) - 2f
  temp = CXSC.mul(a2, temp); // temp = a2 * (((((11.0f * a2) * b2) -
  // b6) - (121f * b4)) - 2f)

  y = CXSC.add(y, temp); // y = (333.75f * b6) + (a2 * (((((11.0f * a2)
  // * b2) - b6) - (121f * b4)) - 2f))

  temp = CXSC.mul(5.5f, b8); // temp = 5.5f * b8;

  y = CXSC.add(y, temp); // y = ((333.75f * b6) + (a2 * (((((11.0f *
  // a2) * b2) - b6) - (121f * b4)) - 2f))) +
  // (5.5f * b8)

  temp = CXSC.mul(2f, b); // temp = 2f * b;
  temp = CXSC.div(a, temp); // temp = a / (2f * b)

  y = CXSC.add(y, temp); // y = (((333.75f * b6) + (a2 * (((((11.0f *
  // a2) * b2) - b6) - (121f * b4)) - 2f))) +
  // (5.5f * b8)) + (a / (2f * b))

  print("CXSC");
  print(y);
  println("_" + Formatar.bin(Double.doubleToRawLongBits(y)));
}

private void rumpJFloat() {
  float a = 77617.0f;
  float b = 33096.0f;

  float a2 = JFloat32.mul(a, a);
  float b2 = JFloat32.mul(b, b);
  float b3 = JFloat32.mul(b2, b);
  float b4 = JFloat32.mul(b3, b);
  float b5 = JFloat32.mul(b4, b);
  float b6 = JFloat32.mul(b5, b);
  float b7 = JFloat32.mul(b6, b);
  float b8 = JFloat32.mul(b7, b);

  float y = JFloat32.mul(333.75f, b6); // y = 333.75f * b6
  float temp = JFloat32.mul(11.0f, a2); // temp = 11.0f * a2
  temp = JFloat32.mul(temp, b2); // temp = (11.0f * a2) * b2
  temp = JFloat32.sub(temp, b6); // temp = ((11.0f * a2) * b2) - b6
```

```
    float temp2 = JFloat32.mul(121f, b4); // temp2 = 121f * b4
    temp = JFloat32.sub(temp, temp2); // temp = (((11.0f * a2) * b2) - b6) -
    // (121f * b4)
    temp = JFloat32.sub(temp, 2f); // temp = ((((11.0f * a2) * b2) - b6) -
    // (121f * b4)) - 2f
    temp = JFloat32.mul(a2, temp); // temp = a2 * (((((11.0f * a2) * b2) -
    // b6) - (121f * b4)) - 2f)

    y = JFloat32.add(y, temp); // y = (333.75f * b6) + (a2 * (((((11.0f * a2)
    // * b2) - b6) - (121f * b4)) - 2f))

    temp = JFloat32.mul(5.5f, b8); // temp = 5.5f * b8;

    y = JFloat32.add(y, temp); // y = ((333.75f * b6) + (a2 * (((((11.0f *
    // a2) * b2) - b6) - (121f * b4)) - 2f))) +
    // (5.5f * b8)

    temp = JFloat32.mul(2f, b); // temp = 2f * b;
    temp = JFloat32.div(a, temp); // temp = a / (2f * b)

    y = JFloat32.add(y, temp); // y = (((333.75f * b6) + (a2 * (((((11.0f *
    // a2) * b2) - b6) - (121f * b4)) - 2f))) +
    // (5.5f * b8)) + (a / (2f * b))

    print("JFloat");
    print(y);
    println("_" + Formatar.bin(Double.doubleToRawLongBits(y)));
  }

  private void rumpFloat() {
    float a = 77617.0f;
    float b = 33096.0f;

    float a2 = a * a;
    float b2 = b * b;
    float b3 = b2 * b;
    float b4 = b3 * b;
    float b5 = b4 * b;
    float b6 = b5 * b;
    float b7 = b6 * b;
    float b8 = b7 * b;
    float y = (((333.75f * b6) + (a2 * (((((11.0f * a2) * b2) - b6) - (121f *
b4)) - 2f))) + (5.5f * b8))
        + (a / (2f * b));

    print("Float");
    print(y);
    println("_" + Formatar.bin(Double.doubleToRawLongBits(y)));
  }

  private void rumpDouble() {
    double a = 77617.0f;
    double b = 33096.0f;

    double a2 = a * a;
    double b2 = b * b;
    double b3 = b2 * b;
    double b4 = b3 * b;
    double b5 = b4 * b;
    double b6 = b5 * b;
    double b7 = b6 * b;
    double b8 = b7 * b;
```

```
    double y = (((333.75f * b6) + (a2 * (((((11.0f * a2) * b2) - b6) - (121f *
b4)) - 2.0f))) + (5.5f * b8))
        + (a / (2.0f * b));

    print("Double");
    print(y);
    println("_" + Formatar.bin(Double.doubleToRawLongBits(y)));
  }
}
```

10.8. Teste da exatidão de JFloat comparada a exatidão de ponto flutuante de C-XSC

```
package br.ufrn.dimap.jxsc.testes;

import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.Random;

import br.ufrn.dimap.cxsc.CXSC;
import br.ufrn.dimap.jxsc.JFloat32;

public class Teste06Arredondamento extends Teste {
  private static final int MAXTESTES = 10000000;
  private NumberFormat nf;
  private double doubleCXSC;
  private float jFloatParaBaixo;
  private float jFloatParaCima;

  public Teste06Arredondamento() {
    nf = new DecimalFormat("0.###############E000");
    nf.setMaximumFractionDigits(10);
    nf.setMinimumFractionDigits(10);
  }

  public void testArredondamento() {
    println("Arredondamento direcionado de JFloat comparado a CXSC");
    print("Operacao");
    print("Para baixo melhor");
    print("Para baixo pior");
    print("Para cima melhor");
    println("Para cima pior");

    executar('+');
    executar('-');
    executar('*');
    executar('/');
  }

  public void executar(char operacao) {
    Random r1 = new Random(), r2 = new Random();
    int j = 0;
    int para_baixo_pior = 0;
    int para_baixo_melhor = 0;
    int para_cima_pior = 0;
    int para_cima_melhor = 0;
```

```
  while (j < MAXTESTES) {
    j++;

    float a = Float.intBitsToFloat(r1.nextInt());
    float b = Float.intBitsToFloat(r2.nextInt());

    // Comparações com NaNs são sempre falsas e por isso, operandos
    // iniciais gerados como NaNs serão descartados
    if (Float.isNaN(a)) {
      j--;
      continue;
    }
    if (Float.isNaN(b)) {
      j--;
      continue;
    }

    doubleCXSC = calcularOperacaoCXSC(operacao, a, b);
    jFloatParaBaixo = calcularOperacaoJFLoatParaBaixo(operacao, a, b);
    jFloatParaCima = calcularOperacaoJFloatParaCima(operacao, a, b);

    /*
     * Comparações com NaNs são sempre falsas e por isso operações que
     * resultem em NaNs serão evitadas
     */
    if (Double.isNaN(doubleCXSC)) {
      j--;
      continue;
    }
    if (Float.isNaN(jFloatParaBaixo)) {
      j--;
      continue;
    }
    if (Float.isNaN(jFloatParaCima)) {
      j--;
      continue;
    }

    if (JFloat32.nextfp(jFloatParaBaixo) >= (float) doubleCXSC) {
      para_baixo_melhor++;
    } else {
      para_baixo_pior++;
    }

    if (JFloat32.prevfp(jFloatParaCima) <= (float) doubleCXSC) {
      para_cima_melhor++;
    } else {
      para_cima_pior++;
    }

  }
  print(operacao + "");
  print(para_baixo_melhor + "");
  print(para_baixo_pior + "");
  print(para_cima_melhor + "");
  println(para_cima_pior + "");
}

private float calcularOperacaoJFloatParaCima(char operacao, float a, float b) {
  /* Para cima */
  JFloat32.setExceptionFlags(0);
  JFloat32.setRoundingMode(JFloat32.ROUND_UP);
```

```
    switch (operacao) {
    case '+':
      return JFloat32.add(a, b);
    case '-':
      return JFloat32.sub(a, b);
    case '*':
      return JFloat32.mul(a, b);
    case '/':
      return JFloat32.div(a, b);
    }
    return Float.NaN;
  }

  private float calcularOperacaoJFLoatParaBaixo(char operacao, float a, float b)
{
    /* Para baixo */
    JFloat32.setExceptionFlags(0);
    JFloat32.setRoundingMode(JFloat32.ROUND_DOWN);

    switch (operacao) {
    case '+':
      return JFloat32.add(a, b);
    case '-':
      return JFloat32.sub(a, b);
    case '*':
      return JFloat32.mul(a, b);
    case '/':
      return JFloat32.div(a, b);
    }
    return Float.NaN;
  }

  private double calcularOperacaoCXSC(char operacao, float a, float b) {
    double aa = a, bb = b;
    switch (operacao) {
    case '+':
      return CXSC.add(aa, bb);
    case '-':
      return CXSC.sub(aa, bb);
    case '*':
      return CXSC.mul(aa, bb);
    case '/':
      return CXSC.div(aa, bb);
    }
    return Double.NaN;
  }
}
```

10.9. Teste do arredondamento direcionado em relação ao resultado de precisão dupla

```
package br.ufrn.dimap.jxsc.testes;

import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.Random;
```

```
import br.ufrn.dimap.cxsc.CXSC;
import br.ufrn.dimap.jxsc.JFloat32;

public class Teste07Arredondamento extends Teste {
  private static final int MAXTESTES = 10000000;
  private NumberFormat nf;
  private double doubleCXSCParaBaixo;
  private float jFloatParaBaixo;
  private float jFloatParaCima;
  private double doubleCXSCParaCima;

  public Teste07Arredondamento() {
    nf = new DecimalFormat("0.################E000");
    nf.setMaximumFractionDigits(10);
    nf.setMinimumFractionDigits(10);
  }

  public void testArredondamento() {
    println("Arredondamento direcionado de JFloat comparado a CXSC");
    print("Operacao");
    print("Para baixo melhor");
    print("Para baixo pior");
    print("Para cima melhor");
    println("Para cima pior");

    executar('+');
    executar('-');
    executar('*');
    executar('/');
  }

  public void executar(char operacao) {
    Random r1 = new Random(), r2 = new Random();

    int j = 0;
    int para_baixo_pior = 0;
    int para_baixo_melhor = 0;
    int para_cima_pior = 0;
    int para_cima_melhor = 0;

    while (j < MAXTESTES) {
      j++;

      float a = Float.intBitsToFloat(r1.nextInt());
      float b = Float.intBitsToFloat(r2.nextInt());

      // Comparações com NaNs são sempre falsas e por isso, operandos
      // iniciais gerados como NaNs serão descartados
      if (Float.isNaN(a)) {
        j--;
        continue;
      }
      if (Float.isNaN(b)) {
        j--;
        continue;
      }

      doubleCXSCParaBaixo = calcularOperacaoCXSCParaBaixo(operacao, a, b);
      doubleCXSCParaCima = calcularOperacaoCXSCParaCima(operacao, a, b);

      jFloatParaBaixo = calcularOperacaoJFLoatParaBaixo(operacao, a, b);
      jFloatParaCima = calcularOperacaoJFloatParaCima(operacao, a, b);
```

```
    /*
     * Comparações com NaNs são sempre falsas e por isso operações que
     * resultem em NaNs serão evitadas
     */
    if (Double.isNaN(doubleCXSCParaBaixo)) {
      j--;
      continue;
    }
    if (Float.isNaN(jFloatParaBaixo)) {
      j--;
      continue;
    }
    if (Float.isNaN(jFloatParaCima)) {
      j--;
      continue;
    }

    if (jFloatParaBaixo <= (float) doubleCXSCParaBaixo) {
      if (JFloat32.nextfp(jFloatParaBaixo) >= (float) doubleCXSCParaBaixo) {
        para_baixo_melhor++;
      } else {
        para_baixo_pior++;
      }
    } else
      para_baixo_pior++;

    if (jFloatParaCima >= (float) doubleCXSCParaCima) {
      if (JFloat32.prevfp(jFloatParaCima) <= (float) doubleCXSCParaCima) {
        para_cima_melhor++;
      } else {
        para_cima_pior++;
      }
    } else {
      para_cima_pior++;
    }

  }
  print(operacao + "");
  print(para_baixo_melhor + "");
  print(para_baixo_pior + "");
  print(para_cima_melhor + "");
  println(para_cima_pior + "");
}

private float calcularOperacaoJFloatParaCima(char operacao, float a, float b) {
  /* Para cima */
  JFloat32.setExceptionFlags(0);
  JFloat32.setRoundingMode(JFloat32.ROUND_UP);

  switch (operacao) {
  case '+':
    return JFloat32.add(a, b);
  case '-':
    return JFloat32.sub(a, b);
  case '*':
    return JFloat32.mul(a, b);
  case '/':
    return JFloat32.div(a, b);
  }
  return Float.NaN;
}
```

```
  private float calcularOperacaoJFLoatParaBaixo(char operacao, float a, float b)
{
    /* Para baixo */
    JFloat32.setExceptionFlags(0);
    JFloat32.setRoundingMode(JFloat32.ROUND_DOWN);

    switch (operacao) {
    case '+':
      return JFloat32.add(a, b);
    case '-':
      return JFloat32.sub(a, b);
    case '*':
      return JFloat32.mul(a, b);
    case '/':
      return JFloat32.div(a, b);
    }
    return Float.NaN;
  }

  private double calcularOperacaoCXSCParaBaixo(char operacao, float a, float b) {
    double aa = a, bb = b;
    switch (operacao) {
    case '+':
      return CXSC.add_down(aa, bb);
    case '-':
      return CXSC.sub_down(aa, bb);
    case '*':
      return CXSC.mul_down(aa, bb);
    case '/':
      return CXSC.div_down(aa, bb);
    }
    return Double.NaN;
  }

  private double calcularOperacaoCXSCParaCima(char operacao, float a, float b) {
    double aa = a, bb = b;
    switch (operacao) {
    case '+':
      return CXSC.add_up(aa, bb);
    case '-':
      return CXSC.sub_up(aa, bb);
    case '*':
      return CXSC.mul_up(aa, bb);
    case '/':
      return CXSC.div_up(aa, bb);
    }
    return Double.NaN;
  }
}
```