PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

ESCOLA POLITÉCNICA

CURSO DE CIÊNCIA DA COMPUTAÇÃO

BERNARDO TARNOWSKI DALLAROSA

LUCAS DA PAZ OLIVEIRA

NICOLAS DUTRA TONDO

PEDRO GONZATTI CHIELLA

RODRIGO MIOTTO SLONGO

PROBABILIDADE E ESTATÍSTICA: IMPLEMENTAÇÃO DE FUNÇÕES NA LINGUAGEM JAVA

PORTO ALEGRE

2025

BERNARDO TARNOWSKI DALLAROSA

LUCAS DA PAZ OLIVEIRA

NICOLAS DUTRA TONDO

PEDRO GONZATTI CHIELLA

RODRIGO MIOTTO SLONGO

PROBABILIDADE E ESTATÍSTICA: IMPLEMENTAÇÃO DE FUNÇÕES NA LINGUAGEM JAVA

Trabalho apresentado no curso de Ciência da Computação e Ciência da Computação da Pontifícia Universidade Católica do Rio Grande do Sul, referente à cadeira de Probabilidade e Estatística

Professor: Felipe Fonseca Salerno

PORTO ALEGRE

2025

SUMÁRIO

I. INTRODUÇÃO

O presente trabalho tem como objetivo detalhar a implementação de funções estatísticas em Java e a posterior comparação com funções nativas. Como a linguagem Java não é amplamente utilizada para áreas relacionadas à ciência de dados, não há funções próprias da linguagem para o cálculo de estatísticas, portanto foi utilizada a biblioteca *Apache Commons Math* na sua versão 3.6.1; mais especificamente, foram utilizados os métodos da classe *StatUtils* para realizar a comparação com as funções implementadas. A documentação de referência para a biblioteca e para a classe pode ser encontrada nas referências.

Para comparar a saída dos métodos é necessário fornecer dados de entrada; foi utilizada a API disponibilizada pela *World Meteorological Organization* para buscar dados históricos mensais sobre o clima de diversas cidades e compilá-los em um arquivo *CSV*, posteriormente lido e interpretado pelo programa. A página da *web* da API utilizada está na seção de referências.

Declaração de autor	
Integrante	**Atividades desenvolvidas**
Bernardo	Implementação dos quartis e *outliers*.
Lucas	Implementação da mediana e da amplitude, obtenção dos dados e escrita do relatório final.
Nicolas	Desenvolvimento da apresentação de *slides*.
Pedro	Revisor.
Rodrigo	Implementação das médias, variância e desvio padrão.

Sobre o uso de IA: Houve utilização de inteligência artificial para auxílio na comunicação com a API para obtenção dos dados e para a listagem de referências.

II. DESENVOLVIMENTO

A abordagem utilizada foi a de criar uma classe Java específica para cada função estatística; elas foram separadas em pacotes conforme a medida calculada, i.e., *dispersion* (dispersão) e *central* (tendência central). É importante ressaltar que, por conveniência, todo o código fonte foi escrito em inglês; a documentação *Javadoc*, no entanto, está em português.

Cada classe foi desenvolvida sob o conceito de classe de utilidades, i.e., uma classe apenas provê métodos estáticos para realizar uma operação, e não deve ser instanciada. Isto é implementado utilizando a palavra-chave *final* ao declarar a classe e sobrescrevendo o construtor padrão com um privado.

Para operações amplamente utilizadas, como ordenar um conjunto de números, encontrar o maior ou o menor valor em um conjunto etc. foi criada uma classe de utilidades chamada *Utils*, seguindo o mesmo padrão apresentado acima.

O tipo numérico escolhido para se trabalhar foi o *BigDecimal* devido à sua maior precisão. Foram criados métodos na classe *Utils* para converter conjuntos de quaisquer tipos *Number* em conjuntos de *BigDecimal*, assim como para converter conjuntos de *BigDecimal* para conjuntos de *double*, formato utilizado pela classe *StatUtils*. Para alguns resultados foi utilizado o método *stripTrailingZeros* da classe *BigDecimal*; ele serve para remover zeros adicionais e, em alguns casos, pode formatar o número com notação científica,

Algumas exceções são tratadas, porém espera-se que os dados de entrada sejam válidos. Por fim, para testar a implementação das funções estatísticas foram criados testes unitários automatizados utilizando *JUnit* na versão 5.12.0. Abaixo são detalhadas cada implementação criada para este projeto.

Para comparar as funções desenvolvidas com as funções da biblioteca foi criada uma interface que define quais métodos serão disponibilizados e duas classes que implementam esta interface, uma delegando a chamada para as funções desenvolvidas e outra para as funções disponíveis na biblioteca.

2.1 Implementação

Abaixo são detalhadas as implementações das funções estatísticas e métodos utilitários. O código fonte completo, incluindo testes e os códigos utilizados para buscar os dados, estará em anexo, bem como o *link* para o repositório remoto estará nas referências.

2.1.1 Métodos Utilitários: Classe *Utils*

A classe *Utils* contém os seguintes métodos públicos:

- *toBigDecimalArray*: converte um conjunto de qualquer tipo numérico para um conjunto de *BigDecimal*;
- *toDoubleArray*: converte um conjunto de *BigDecimal* para um conjunto de *double*;
- *containsNonPositiveValues*: retorna verdadeiro se o conjunto contém valores não positivos, i.e., menores ou iguais a zero;
- *max*: retorna o maior valor do conjunto;
- *min*: retorna o menor valor do conjunto;
- *sort*: retorna uma cópia ordenada do conjunto; utiliza o algoritmo *merge sort*.

Além disso, a classe *Utils* também fornece uma constante *MATH_CONTEXT*, objeto utilizado para realizar algumas operações matemáticas com *BigDecimal*; define a precisão dos resultados para 15 valores significativos e arredondamento *half-up*, i.e., se a fração descartada for maior ou igual a 0.5 arredonda para cima, caso contrário para baixo.

2.1.2 Média Aritmética: Classe *ArithmeticMean*

A implementação da média aritmética consiste simplesmente em um laço de repetição que soma todos os valores do conjunto e posteriormente divide o resultado pela quantidade de números do mesmo. Remove quaisquer zeros adicionais à direita do resultado.

```java
    Realiza o cálculo da média aritmética do conjunto de valores passado como argumento.

    Params: values – Conjunto sobre o qual ocorrerá o cálculo; não pode ser vazio.

    Returns: Valor correspondente à média aritmética do conjunto.

    Throws: ArithmeticException – Se o conjunto de dados informado for vazio.
  public static BigDecimal arithmeticMean(BigDecimal... values) {
    BigDecimal sum = BigDecimal.valueOf(0.0);
    BigDecimal length = BigDecimal.valueOf(values.length);

    for (BigDecimal number : values) {
      sum = sum.add(number);
    }

    return sum.divide(length, MATH_CONTEXT).stripTrailingZeros();
  }
}
```

Figura 1: Implementação do método que calcula a média aritmética.

2.1.3 Média Geométrica: Classe *GeometricMean*

A implementação da média geométrica segue uma abordagem diferente da fórmula utilizada no dia a dia. A multiplicação de vários números pode resultar em um valor muito grande, e a posterior operação de potência com um expoente que é uma fração pode resultar em números muito pequenos. Para evitar estouro de representação, utilizou-se o método de soma de logaritmos: soma-se os logaritmos de todos os números, divide-se o resultado pela

quantidade de elementos do conjunto e, por fim, obtêm-se o valor do expoente em *e* (Euler).

```
Realiza o cálculo da média geométrica do conjunto de valores passado como argumento. Utiliza o
método da soma de logaritmos para evitar estouro de representação ao multiplicar vários valores.

Params: values – Conjunto sobre o qual ocorrerá o cálculo; não pode ser vazio.

Returns: Valor correspondente à média geométrica do conjunto.

Throws: ArithmeticException – Se o conjunto de dados informado for vazio ou se o conjunto
        contiver valores não positivos.
```

```java
public static BigDecimal geometricMean(BigDecimal... values) {
  if (containsNonPositiveValues(values)) {
    throw new ArithmeticException("A média geométrica só é definida para números reais positivos");
  }

  BigDecimal logSum = Arrays.stream(values)
    .map(v -> BigDecimal.valueOf(Math.log(v.doubleValue())))
    .reduce(BigDecimal.valueOf(0.0), BigDecimal::add);

  BigDecimal meanLog = logSum.divide(BigDecimal.valueOf(values.length), MATH_CONTEXT);
  double result = Math.exp(meanLog.doubleValue());
  return BigDecimal.valueOf(result).stripTrailingZeros();
}
```

Figura 2: Implementação do método que calcula a média geométrica.

2.1.4 Média Harmônica: Classe *HarmonicMean*

A implementação da média harmônica realiza o somatório de todos os valores invertidos e retorna o a quantidade de elementos dividida pelo resultado deste somatório. Utiliza uma *stream* (fluxo de dados do conjunto) e faz sua redução (aplica uma função acumuladora para cada elemento) para obter o resultado do somatório, conforme a figura abaxo:

```java
/**
 * Realiza o cálculo da média harmônica do conjunto de valores passado como argumento.
 *
 * Params: values – Conjunto sobre o qual ocorrerá o cálculo; não pode ser vazio.
 *
 * Returns: Valor correspondente à média harmônica do conjunto.
 *
 * Throws: ArithmeticException – Se o conjunto de dados informado for vazio.
 */
public static BigDecimal harmonicMean(BigDecimal... values) {
  if (containsNonPositiveValues(values)) {
    throw new ArithmeticException("A média harmônica só é definida para números reais positivos");
  }

  BigDecimal length = BigDecimal.valueOf(values.length);

  BigDecimal summationResult = inverseNumbersSummation(values);

  return length.divide(summationResult, MATH_CONTEXT).stripTrailingZeros();
}

/**
 * Realiza a somatório dos elementos na potência -1 ; equivalente à ∑ 1 / values[i] ;
 *
 * Params: values – Conjunto sobre o qual ocorrerá o cálculo.
 *
 * Returns: O resultado da soma do somatório.
 */
private static BigDecimal inverseNumbersSummation(BigDecimal[] values) {
  return Arrays.stream(values).reduce(BigDecimal.valueOf(0.0), HarmonicMean::addFraction);
}

/**
 * Adiciona o valor de 1/x ao valor de sum .
 *
 * Params: sum – O valor do somatório.
 *         x – Elemento sendo adicionado.
 *
 * Returns: Resultado de sum + 1/x .
 */
private static BigDecimal addFraction(BigDecimal sum, BigDecimal x) {
  return sum.add(BigDecimal.valueOf(1).divide(x, MATH_CONTEXT));
}
```

Figura 3: Implementação do método que calcula a média harmônica e métodos auxiliares.

2.1.5 Mediana: Classe *Median*

A implementação da mediana ordena o conjunto e define qual método auxiliar — *medianEven* ou *medianOdd* — deve chamar com base na paridade do número de elementos. O método selecionado retornará, no caso *odd*, o valor na posição do meio do conjunto e, no caso *even*, a média aritmética dos dois valores nas posições centrais.

```
Realiza o cálculo da mediana do conjunto de valores passado como argumento.

Params: values – Conjunto sobre o qual ocorrerá o cálculo; não pode ser vazio.

Returns: Valor correspondente à mediana do conjunto.

Throws: IndexOutOfBoundsException – Se o conjunto de dados informado for vazio.
        NullPointerException – Se o conjunto de dados informado for nulo ou contiver valores
        nulos.
```

```java
public static BigDecimal median(BigDecimal... values) {
  int length = values.length;

  BigDecimal[] sorted = sort(values);

  if (length % 2 == 0) {
    return medianEven(sorted);
  }
  return medianOdd(sorted);
}
```

```
Calcula a mediana de um conjunto de valores reais ordenados com número par de elementos.
Dado que arrays, em Java, começam na posição 0, e que o valor de int i = values.length / 2 é
o quociente inteiro da divisão, retorna a média dos valores nas posições i e i - 1 do array.

E.g.: Se o tamanho do array for 8, i será o quociente inteiro de 8 / 2, i.e., 4, que representa o
valor na quinta posição do array, equivalente ao cálculo matemático n / 2 + 1; logo, para
completar o cálculo da média utiliza-se o valor na posição i - 1, que representa o valor na quarta
posição (n / 2).

Params: values – Conjunto de valores reais ordenados com número par de elementos.

Returns: A mediana para o conjunto passado como argumento.
```

```java
private static BigDecimal medianEven(BigDecimal... values) {
  int i = values.length / 2;
  BigDecimal n1 = values[i - 1];
  BigDecimal n2 = values[i];
  return arithmeticMean(n1, n2);
}
```

```
Calcula a mediana de um conjunto de valores reais ordenados com número ímpar de elementos.
Dado que arrays, em Java, começam na posição 0, e que o valor de int i = values.length / 2 é
o quociente inteiro da divisão, retorna o valor na posição i do array.

E.g.: Se o tamanho do array for 15, i será o quociente inteiro de 15 / 2, i.e., 7, que representa
o valor na oitava posição do array, equivalente ao cálculo matemático (n + 1) / 2.

Params: values – Conjunto de valores reais ordenados com número ímpar de elementos.

Returns: A mediana para o conjunto passado como argumento.
```

```java
private static BigDecimal medianOdd(BigDecimal... values) {
  int i = values.length / 2;
  return values[i];
}
```

Figura 4: Implementação do método que calcula a mediana e métodos auxiliares.

2.1.6 Amplitude: Classe *Amplitude*

Para realizar o cálculo da amplitude de um conjunto de valores são utilizados os métodos utilitários *min* e *max* da classe *Utils* e o método retorna o valor de *max* subtraído o valor de *min*:

```
Realiza o cálculo da amplitude do conjunto de valores passado como argumento.

Params:  values – Conjunto sobre o qual ocorrerá o cálculo.

Returns:  Valor correspondente à amplitude do conjunto.

Throws:  IllegalArgumentException – Se o conjunto de dados informado for nulo ou vazio.

See Also:  Utils.min(BigDecimal...),
           Utils.max(BigDecimal...)
```

```java
public static BigDecimal amplitude(BigDecimal... values) {
  BigDecimal max = max(values);
  BigDecimal min = min(values);

  return max.subtract(min);
}
```

Figura 5: Implementação do método que calcula a amplitude usando métodos da classe Utils.

2.1.7 Variância: Classe *Variance*

Para o cálculo da variância foram disponibilizados dois métodos públicos: um para amostra (*sample*) e outro para população (*population*). A parte em comum dos dois cálculos, i.e., o somatório de cada valor subtraído da média aritmética e elevado ao quadrado, é delegada para métodos auxiliares. Utiliza a classe *ArithmeticMean* para o cálculo da média aritmética.

Realiza o cálculo de variância da população para o conjunto de dados passado como argumento.

Fórmula: $\sigma^2 = \sum(xi - \mu)^2 / N$

- σ^2: Variância da população;
- xi: Cada valor individual no conjunto de dados;
- μ: Média da população;
- N: Número total de observações na população.

Params: `values` – Conjunto sobre o qual ocorrerá o cálculo; não pode ser vazio.

Returns: Valor correspondente à variância populacional do conjunto.

Throws: `ArithmeticException` – Se o conjunto de dados informado for vazio ou unitário.
`NullPointerException` – Se o conjunto de dados informado for nulo.

```
public static BigDecimal populationVariance(BigDecimal... values) {
  BigDecimal n = BigDecimal.valueOf(values.length);
  return variance(n, values);
}
```

Realiza o cálculo de variância da amostra para o conjunto de dados passado como argumento.

Fórmula: $s^2 = \sum(xi - \bar{x})^2 / (n - 1)$

- s^2: Variância da amostra;
- xi: Cada valor individual no conjunto de dados;
- $\bar{x}$: Média da amostra;
- n: Número de observações da amostra.

Params: `values` – Conjunto sobre o qual ocorrerá o cálculo; não pode ser vazio.

Returns: Valor correspondente à variância amostral para o conjunto.

Throws: `ArithmeticException` – Se o conjunto de dados informado for vazio ou unitário.
`NullPointerException` – Se o conjunto de dados informado for nulo.

```
public static BigDecimal sampleVariance(BigDecimal... values) {
  BigDecimal nMinusOne = BigDecimal.valueOf(values.length - 1);
  return variance(nMinusOne, values);
}
```

Figura 6: Métodos públicos da classe Variance.

```
Realiza o cálculo da variância do conjunto de valores passado como argumento utilizando o
parâmetro n informado.

Params: n – Valor sobre o qual será dividido o somatório para obter a variância; difere entre as
        variâncias amostral e populacional.
        values – Conjunto sobre o qual ocorrerá o cálculo; não pode ser vazio.

Returns: Valor correspondente à variância calculada.
```

```java
private static BigDecimal variance(BigDecimal n, BigDecimal... values) {
  BigDecimal arithmeticMean = ArithmeticMean.arithmeticMean(values);
  BigDecimal summation = arithmeticMeanSummation(arithmeticMean, values);

  return summation.divide(n, MATH_CONTEXT).stripTrailingZeros();
}
```

```
Realiza o somatório utilizado para calcular a variância para o conjunto de dados informado: subtrai
cada valor pela média aritmética e eleva ao quadrado; o somatório é o resultado da soma destes
valores.

Params: arithmeticMean – A média aritmética do conjunto.
        values – Conjunto sobre o qual ocorrerá o cálculo.

Returns: O resultado do somatório.
```

```java
private static BigDecimal arithmeticMeanSummation(BigDecimal arithmeticMean, BigDecimal... values) {
  return Arrays.stream(values)
    .map(x -> x.subtract(arithmeticMean).pow( n: 2))
    .reduce(BigDecimal.ZERO, BigDecimal::add);
}
```

Figura 7: Métodos auxiliares da classe Variance.

2.1.8 Desvio Padrão: Classe *StandardDeviation*

De modo similar à classe *Variance*, a implementação do desvio padrão fornece dois métodos públicos: um para amostra e outro para população; um terceiro método auxiliar retorna a raiz quadrada da variância calculada, i.e., o desvio padrão:

```
Realiza o cálculo do desvio padrão da população para conjunto de valores passado como argumento.

Params:  values – Conjunto sobre o qual ocorrerá o cálculo; não pode ser vazio.

Returns:  Valor correspondente ao desvio padrão populacional do conjunto.

See Also:  Variance.populationVariance(BigDecimal...)
```

```
public static BigDecimal populationStandardDeviation(BigDecimal... values) {
  return standardDeviation(Variance.populationVariance(values));
}
```

```
Realiza o cálculo do desvio padrão da amostra para o conjunto de valores passado como argumento.

Params:  values – Conjunto sobre o qual ocorrerá o cálculo; não pode ser vazio.

Returns:  Valor correspondente ao desvio padrão amostral do conjunto.

See Also:  Variance.sampleVariance(BigDecimal...)
```

```
public static BigDecimal sampleStandardDeviation(BigDecimal... values) {
  return standardDeviation(Variance.sampleVariance(values));
}
```

```
Realiza o cálculo do desvio padrão para a variância passada como argumento. Este método retorna o
valor de BigDecimal.sqrt(MathContext) utilizando o MathContext da aplicação e chamando
BigDecimal.stripTrailingZeros().

Params:  variance – Valor sobre o qual ocorrerá o cálculo.

Returns: Valor correspondente ao desvio padrão da variância informada.
```

```
private static BigDecimal standardDeviation(BigDecimal variance) {
  return variance.sqrt(MATH_CONTEXT).stripTrailingZeros();
}
```

Figura 8: Implementação dos métodos que calculam o desvio padrão.

2.1.9 Quartis: Classe *Quartiles*

Para calcular os quartis foi utilizada uma abordagem diferente: criou-se, dentro da classe, um *enum* chamado *Quartile* que contém as constantes *Q1*, *Q2* e *Q3*, cada uma com um valor percentual associado. O quartil desejado é informado para o método, que calcula o valor de $n + 1$, sendo n a quantidade de elementos do conjunto, e multiplica o resultado pelo valor do quartil. Se necessário, calcula a interpolação e retorna o valor obtido.

```java
/**
 * Calcula o quartil desejado para o conjunto de valores informado.
 *
 * Params: q – O quartil desejado.
 *         values – Conjunto sobre o qual ocorrerá o cálculo; não pode ser vazio.
 *
 * Returns: O valor correspondente ao quartil calculado.
 */
public static BigDecimal quartile(Quartile q, BigDecimal... values) {
  if (values.length == 0) {
    throw new IllegalArgumentException("O conjunto não pode ser vazio.");
  }

  BigDecimal[] sorted = sort(values);
  BigDecimal position = BigDecimal.valueOf(sorted.length + 1).multiply(q.percentValue);

  return interpolate(position, sorted);
}

/**
 * Cálculo da interpolação a partir dos valores ordenados e do índice informado. Se a posição não for
 * um número real, retorna o valor em index - 1 (pois os conjuntos, em Java, começam em 0).
 *
 * Params: position – Índice calculado.
 *         values – Conjunto de valores ordenados.
 *
 * Returns: Resultado da interpolação.
 */
private static BigDecimal interpolate(BigDecimal position, BigDecimal... values) {
  int index = position.intValue();
  BigDecimal fraction = position.remainder(BigDecimal.valueOf(1.0), MATH_CONTEXT);

  if (fraction.compareTo(BigDecimal.valueOf(0.0)) == 0) {
    return values[index - 1].stripTrailingZeros();
  }

  BigDecimal lower = values[index - 1];
  BigDecimal upper = values[index];

  return lower.add(fraction.multiply(upper.subtract(lower))).stripTrailingZeros();
}
```

Figura 9: Métodos implementados na classe Quartiles.

```java
public enum Quartile {
  Q1("0.25"), Q2("0.5"), Q3("0.75");

  private final BigDecimal percentValue;

  Quartile(String percentValue) {
    this.percentValue = new BigDecimal(percentValue);
  }

  public BigDecimal percentValue() {
    return percentValue;
  }
}
```

Figura 10: Enumeração Quartile.

2.1.10 *Outliers*: Classe *Outliers*

Para calcular os limites superior e inferior de um conjunto utilizou-se abordagem semelhante ao caso anterior: criou-se um *enum* chamado *Bound* com duas constantes — *LOWERBOUND* e *UPPERBOUND*, limite inferior e superior respectivamente. Com base na constante informada, o método calcula o resultado; se for *LOWERBOUND* o resultado é $Q1 - 1{,}5 \times IQR$, se for *UPPERBOUND* é $Q3 + 1{,}5 \times IQR$, sendo IQR (*interquartile range* ou intervalo interquartil) o valor de $Q3 - Q1$ e $Q1$ e $Q3$ os valores do primeiro e terceiro quartil, respectivamente. É utilizada a classe *Quartiles* para calcular os quartis.

```
Identifica os outliers em um conjunto de valores usando o intervalo interquartil (IQR).

Params:  bound – Determina qual limite (inferior ou superior) é analisado.
         values – Conjunto de valores a serem analisados.

Returns: Outlier calculado.

Throws:  IllegalArgumentException – Se o conjunto informado for nulo ou vazio.

See Also: Outliers.Bound
```

```java
public static BigDecimal outlier(Bound bound, BigDecimal... values) {
  if (values == null || values.length == 0) {
    throw new IllegalArgumentException("O conjunto de dados informado não pode ser nulo ou vazio!");
  }

  BigDecimal q1 = Quartiles.quartile(Quartiles.Quartile.Q1, values);
  BigDecimal q3 = Quartiles.quartile(Quartiles.Quartile.Q3, values);
  BigDecimal iqrMultiplied = q3.subtract(q1).multiply(new BigDecimal( val: "1.5"));

  return switch (bound) {
    case LOWERBOUND -> q1.subtract(iqrMultiplied).stripTrailingZeros();
    case UPPERBOUND -> q3.add(iqrMultiplied).stripTrailingZeros();
  };
}

public enum Bound {
  LOWERBOUND, UPPERBOUND
}
```

Figura 11: Implementação da classe Outliers.

2.2 Comparação

No pacote *statistics.calculator* foi criada uma *interface* que define um método para cada uma das operações explicadas acima; a classe *PucrsCalculator* implementa esta interface e delega a chamada de cada método para o equivalente implementado. Já a classe *StdCalculator* delega as chamadas para os métodos da classe *StatUtils*.

Algumas observações dignas de nota: A classe *StatUtils* trabalha com valores *double*, então os valores de entrada foram convertidos de *BigDecimal* para *double* e o resultado convertido novamente para *BigDecimal*. Não foi encontrada implementação existente para a média harmônica, portanto a classe *StdCalculator* não é capaz de realizar esta operação; além disso, também não há métodos para o cálculo do desvio padrão, portanto a classe *StdCalculator* retorna a raiz quadrada da variância para estas operações. Por fim, não há o cálculo dos *outliers*, então é calculado o valor do $IQR \times 1{,}5$ e a subsequente operação de adição ou soma com o quartil correspondente com base na constante *Bound* informada.

Para o cálculo dos quartis e *outliers* foram usadas como referências as enumerações (*Quartile* e *Bound*) criadas; no caso dos quartis, a classe *StatUtils* possui apenas o método para calcular percentis, portanto este foi utilizado multiplicando o valor associado à constante *Quartile* por 100. Para o cálculo dos *outliers*, a constante *Bound* apenas indica qual cálculo deve ser realizado.

Em métodos que poderiam lançar exceções, e.g., para valores não positivos, foi realizado tratamento com *try-catch* na implementação da *interface StatisticsCalculator*, retornando *null* ao invés de interromper a execução do programa.

```
See Also: ArithmeticMean.arithmeticMean(BigDecimal...)
@Override
public BigDecimal arithmeticMean(BigDecimal... values) {
  return ArithmeticMean.arithmeticMean(values);
}

See Also: GeometricMean.geometricMean(BigDecimal...)
@Override
public BigDecimal geometricMean(BigDecimal... values) {
  try {
    return GeometricMean.geometricMean(values);
  } catch (Exception e) {
    return null;
  }
}

See Also: HarmonicMean.harmonicMean(BigDecimal...)
@Override
public BigDecimal harmonicMean(BigDecimal... values) {
  try {
    return HarmonicMean.harmonicMean(values);
  } catch (Exception e) {
    return null;
  }
}
```

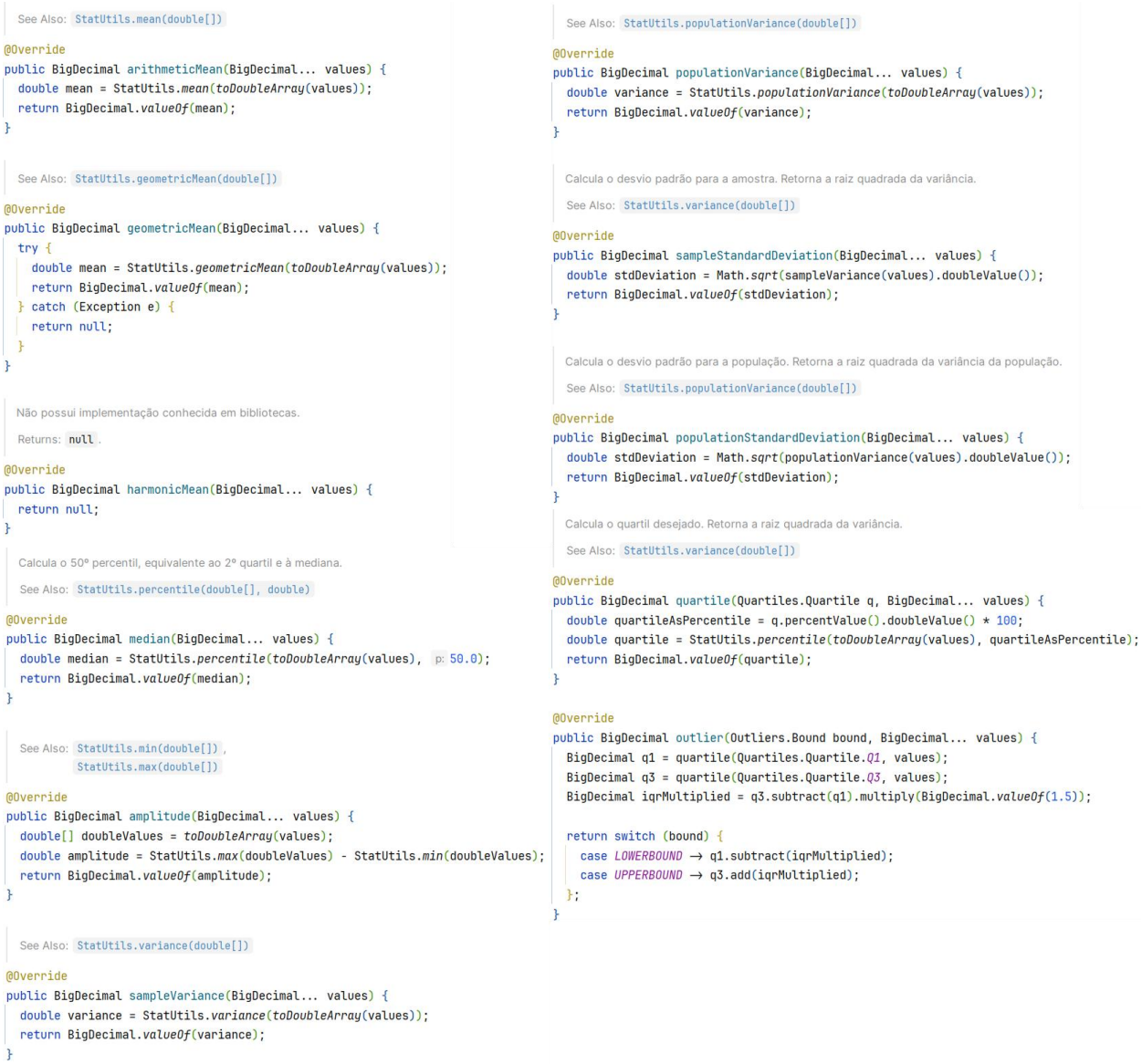
```
See Also: StandardDeviation.sampleStandardDeviation(BigDecimal...)
@Override
public BigDecimal sampleStandardDeviation(BigDecimal... values) {
  return StandardDeviation.sampleStandardDeviation(values);
}

See Also: StandardDeviation.populationStandardDeviation(BigDecimal...)
@Override
public BigDecimal populationStandardDeviation(BigDecimal... values) {
  return StandardDeviation.populationStandardDeviation(values);
}

See Also: Quartiles.quartile(Quartiles.Quartile, BigDecimal...)
@Override
public BigDecimal quartile(Quartiles.Quartile q, BigDecimal... values) {
  return Quartiles.quartile(q, values);
}

See Also: Outliers.outlier(Outliers.Bound, BigDecimal...)
@Override
public BigDecimal outlier(Outliers.Bound bound, BigDecimal... values) {
  return Outliers.outlier(bound, values);
}
```

Figura 12: Detalhes de implementação da classe PucrsCalculator



```
See Also: StatUtils.mean(double[])
@Override
public BigDecimal arithmeticMean(BigDecimal... values) {
  double mean = StatUtils.mean(toDoubleArray(values));
  return BigDecimal.valueOf(mean);
}

See Also: StatUtils.geometricMean(double[])
@Override
public BigDecimal geometricMean(BigDecimal... values) {
  try {
    double mean = StatUtils.geometricMean(toDoubleArray(values));
    return BigDecimal.valueOf(mean);
  } catch (Exception e) {
    return null;
  }
}

Não possui implementação conhecida em bibliotecas.
Returns: null .
@Override
public BigDecimal harmonicMean(BigDecimal... values) {
  return null;
}

Calcula o 50º percentil, equivalente ao 2º quartil e à mediana.
See Also: StatUtils.percentile(double[], double)
@Override
public BigDecimal median(BigDecimal... values) {
  double median = StatUtils.percentile(toDoubleArray(values), p: 50.0);
  return BigDecimal.valueOf(median);
}

See Also: StatUtils.min(double[]),
          StatUtils.max(double[])
@Override
public BigDecimal amplitude(BigDecimal... values) {
  double[] doubleValues = toDoubleArray(values);
  double amplitude = StatUtils.max(doubleValues) - StatUtils.min(doubleValues);
  return BigDecimal.valueOf(amplitude);
}

See Also: StatUtils.variance(double[])
@Override
public BigDecimal sampleVariance(BigDecimal... values) {
  double variance = StatUtils.variance(toDoubleArray(values));
  return BigDecimal.valueOf(variance);
}

See Also: StatUtils.populationVariance(double[])
@Override
public BigDecimal populationVariance(BigDecimal... values) {
  double variance = StatUtils.populationVariance(toDoubleArray(values));
  return BigDecimal.valueOf(variance);
}

Calcula o desvio padrão para a amostra. Retorna a raiz quadrada da variância.
See Also: StatUtils.variance(double[])
@Override
public BigDecimal sampleStandardDeviation(BigDecimal... values) {
  double stdDeviation = Math.sqrt(sampleVariance(values).doubleValue());
  return BigDecimal.valueOf(stdDeviation);
}

Calcula o desvio padrão para a população. Retorna a raiz quadrada da variância da população.
See Also: StatUtils.populationVariance(double[])
@Override
public BigDecimal populationStandardDeviation(BigDecimal... values) {
  double stdDeviation = Math.sqrt(populationVariance(values).doubleValue());
  return BigDecimal.valueOf(stdDeviation);
}

Calcula o quartil desejado. Retorna a raiz quadrada da variância.
See Also: StatUtils.variance(double[])
@Override
public BigDecimal quartile(Quartiles.Quartile q, BigDecimal... values) {
  double quartileAsPercentile = q.percentValue().doubleValue() * 100;
  double quartile = StatUtils.percentile(toDoubleArray(values), quartileAsPercentile);
  return BigDecimal.valueOf(quartile);
}

@Override
public BigDecimal outlier(Outliers.Bound bound, BigDecimal... values) {
  BigDecimal q1 = quartile(Quartiles.Quartile.Q1, values);
  BigDecimal q3 = quartile(Quartiles.Quartile.Q3, values);
  BigDecimal iqrMultiplied = q3.subtract(q1).multiply(BigDecimal.valueOf(1.5));

  return switch (bound) {
    case LOWERBOUND -> q1.subtract(iqrMultiplied);
    case UPPERBOUND -> q3.add(iqrMultiplied);
  };
}
```

Figura 13: Implementação dos métodos em StdCalculator.

2.2.1 Resultado das Comparações

Para poder comparar as funções implementadas com as funções oferecidas pela biblioteca *Apache Commons Math* são necessários dados de entrada. Para este projeto foi utilizado um arquivo CSV montado com dados meteorológicos de 2071 cidades em 178 países diferentes. O arquivo contém, em cada linha, o nome do país, nome da cidade, um valor *booleano* informando se a cidade é capital, o número do mês de referência para os dados, a temperatura mínima, a temperatura máxima, a quantidade de dias de chuva e a precipitação deste mês. Para cada cidade há 12 linhas de dados, uma para cada mês do ano.

O programa realiza a leitura deste arquivo CSV, extraindo todas as informações e classificando-as de acordo: o nome do país e da cidade são qualitativas nominais (*Strings*), o

mês é qualitativa ordinal (enumeração *Month*); as variáveis numéricas temperatura e precipitação são lidas como *Double* (quantitativa contínua) e dias de chuva é lida como *Integer* (quantitativa discreta). Para realizar os cálculos, porém, todas as variáveis numéricas são consideradas quantitativas discretas (*BigDecimal*) por questão de precisão.

Para manipular os dados, decidiu-se por agrupá-los por países, i.e., junta-se os dados meteorológicos disponíveis para cada cidade de um país e calculam-se as funções estatísticas para os conjuntos resultantes. Os cálculos são realizados pelas classes *PucrsCalculator* e *StdCalculator*; ambos os resultados, e o módulo da diferença entre eles, são exibidos na saída. Por conveniência, para reduzir o tamanho da saída, o programa principal só está realizando os cálculos para o Brasil, mostrando a seguinte saída:

```
Brazil - Temperatura mínima
| Função                    | Resultado PUCRS           | Resultado Apache          | Diferença                 |
| ------------------------- | ------------------------- | ------------------------- | ------------------------- |
| Média aritmética          | 20.11875                  | 20.118750000000002        | 0.000000000000002         |
| Média geométrica          | 19.739152973656918        | 19.739152973656847        | 0.000000000000071         |
| Média harmônica           | 19.27034625476322         | N/A                       | N/A                       |
| Mediana                   | 21.6                      | 21.6                      | 0                         |
| Amplitude                 | 16.9                      | 16.9                      | 0                         |
| Variância amostral        | 12.60062282229965         | 12.600622822299657        | 0.000000000000007         |
| Variância populacional    | 12.55687065972222         | 12.556870659722229        | 0.000000000000009         |
| Desvio padrão amostral    | 3.549735598928412         | 3.5497355989284127        | 0.0000000000000007        |
| Desvio padrão populacional | 3.543567504609193         | 3.543567504609194         | 0.000000000000001         |
| Q1 (1° Quartil)           | 18.3                      | 18.3                      | 0                         |
| Q2 (2° Quartil/Mediana)   | 21.6                      | 21.6                      | 0                         |
| Q3 (3° Quartil)           | 22.9                      | 22.9                      | 0                         |
| Outlier inferior          | 11.4                      | 11.4                      | 0                         |
| Outlier supperior         | 29.8                      | 29.8                      | 0                         |

Brazil - Temperatura máxima
| Função                    | Resultado PUCRS           | Resultado Apache          | Diferença                 |
| ------------------------- | ------------------------- | ------------------------- | ------------------------- |
| Média aritmética          | 28.77152777777778         | 28.771527777777777        | 0.000000000000003         |
| Média geométrica          | 28.58813630117541         | 28.58813630117541         | 0                         |
| Média harmônica           | 28.38746721312087         | N/A                       | N/A                       |
| Mediana                   | 29.4                      | 29.4                      | 0                         |
| Amplitude                 | 18.1                      | 18.099999999999998        | 0.000000000000002         |
| Variância amostral        | 9.738001838946961         | 9.73800183894697          | 0.000000000000009         |
| Variância populacional    | 9.704189332561728         | 9.704189332561736         | 0.000000000000008         |
| Desvio padrão amostral    | 3.120577164395548         | 3.1205771643955496        | 0.0000000000000016        |
| Desvio padrão populacional | 3.115154784687549         | 3.11515478468755          | 0.000000000000001         |
| Q1 (1° Quartil)           | 27.125                    | 27.125                    | 0                         |
| Q2 (2° Quartil/Mediana)   | 29.4                      | 29.4                      | 0                         |
| Q3 (3° Quartil)           | 30.7                      | 30.7                      | 0                         |
| Outlier inferior          | 21.7625                   | 21.7625                   | 0                         |
| Outlier supperior         | 36.0625                   | 36.0625                   | 0                         |
```

Figura 14: Cálculos das funções estatísticas para temperaturas máximas e mínimas no Brasil.

Brazil - Dias de chuva

Função	Resultado PUCRS	Resultado Apache	Diferença
Média aritmética	13.27083333333333	13.270833333333334	0.000000000000004
Média geométrica	11.637168680228974	11.637168680228964	0.00000000000001
Média harmônica	9.679008224763117	N/A	N/A
Mediana	13	13	0
Amplitude	26	26	0
Variância amostral	36.18423344947735	36.18423344947738	0.00000000000003
Variância populacional	36.05859375	36.05859375000003	0.00000000000003
Desvio padrão amostral	6.015333195216816	6.015333195216819	0.000000000000003
Desvio padrão populacional	6.004880827293744	6.004880827293746	0.000000000000002
Q1 (1° Quartil)	9	9	0
Q2 (2° Quartil/Mediana)	13	13	0
Q3 (3° Quartil)	18	18	0
Outlier inferior	-4.5	-4.5	0
Outlier supperior	31.5	31.5	0

Brazil - Precipitação

Função	Resultado PUCRS	Resultado Apache	Diferença
Média aritmética	151.1149305555556	151.11493055555556	0.00000000000004
Média geométrica	111.42201319368665	111.42201319368637	0.00000000000028
Média harmônica	67.13471107975656	N/A	N/A
Mediana	125.85	125.85	0
Amplitude	466.4	466.40000000000003	0.00000000000003
Variância amostral	10594.59374842722	10594.593748427229	0.000000000009
Variância populacional	10557.80696457851	10557.806964578522	0.000000000012
Desvio padrão amostral	102.9300429827328	102.93004298273283	0.00000000000003
Desvio padrão populacional	102.7511896017682	102.75118960176822	0.00000000000002
Q1 (1° Quartil)	74.9	74.9	0
Q2 (2° Quartil/Mediana)	125.85	125.85	0
Q3 (3° Quartil)	215.375	215.375	0
Outlier inferior	-135.8125	-135.8125	0
Outlier supperior	426.0875	426.0875	0

Figura 15: Cálculo das funções estatísticas para dias de chuva e precipitação.

A saída completa do programa, com as mesmas variáveis de entrada, pode ser obtida em anexo junto a este documento, no arquivo *output.txt*.

III. CONCLUSÃO

Conforme é possível constatar nas figuras 14 e 15, o resultado de ambas as implementações — *PucrsCalculator* e *StdCalculator* — ficaram iguais em muitos casos; quando houve divergência foi sempre por um valor mínimo, muito provavelmente devido a um erro de representação de números reais. Isso pode ser ocasionado devido ao fato de que alguns valores decimais, e.g. 0.1, não possuem representação finita no sistema de numeração binário. Dessa forma, a computação necessita aproximar o resultado, o que consequentemente gera erros de representação. A utilização de *BigDecimal* ajuda a reduzir a ocorrência destes erros, porém eles não podem ser completamente evitados devido às limitações da máquina.

No que tange à implementação, não foram encontradas grandes dificuldades. Os fatos mais dignos de nota talvez sejam a necessidade de modificar o cálculo da média geométrica para utilizar soma de logaritmos, buscando evitar um estouro de representação, e a implementação dos quartis e *outliers* utilizando enumerações, visando clareza e reutilização de código, boas práticas de programação.

Outro fato interessante é a ausência, no ecossistema Java, de funções e bibliotecas para cálculos estatísticos; na biblioteca *Apache Commons Math*, além da classe *StatUtils* há outras implementações voltadas para esta área, porém em sua maioria para as mesmas funções, visando apenas uma utilização diferente. Não foram encontradas implementações para o cálculo da média harmônica, e as implementações encontradas para desvio padrão da amostra exigem que se trabalhe com os dados dentro da classe, ao invés de um método utilitário que recebe os valores e retorna o resultado.

De modo geral, acredita-se que o código apresentado está bem estruturado e organizado, além de funcional. Foram desenvolvidos testes unitários automatizados para garantir que a execução das funções está de acordo com o esperado, aumentando a qualidade e confiabilidade do projeto. Poderia, se fosse de interesse, ser transformado em uma biblioteca para ser importado em projetos Java que necessitam utilizar funções estatísticas.

REFERÊNCIAS

THE APACHE SOFTWARE FOUNDATION. **Commons Math: The Apache Commons Mathematics Library**, © 2003-2016. Documentação oficial da biblioteca *Apache Commons Math*. Disponível em: https://commons.apache.org/proper/commons-math/index.html. Acesso em: 17 mar. 2025.

THE APACHE SOFTWARE FOUNDATION. **StatUtils (Apache Commons Math 3.6.1 API)**, © 2003-2024. Documentação oficial da classe *StatUtils*. Disponível em: https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/index.html. Acesso em: 17 mar. 2025.

OLIVEIRA, Lucas da Paz et al. *probability-statistics-calculator*: repositório contendo código fonte do projeto. 2025. Disponível em: https://github.com/xLucaspx/probability-statistics-calculator. Acesso em: 19 mar. 2025.

WORLD METEOROLOGICAL ORGANIZATION. **World Weather Information Service - Official Forecasts**, © 2020. Diretrizes para *download* de previsões e informações climatológicas. Disponível em: https://worldweather.wmo.int/en/dataguide.html. Acesso em: 15 mar. 2025.