Vorcyc Mathematics （以下简称 VMath） 是一套基于.NET 8的算法库，其中包含了基础数学、位运算、线性代数、统计学、机器学习、数字信号处理等领域的相关算法，其目的旨在为开发人员提供一套统一的、标准的、高性能的并且能运行在异构设备上的API集。

我司已有一套数学算法库 Vorcyc MathHub ，它在未来将不再继续维护，而 VMath 将会作为一个全新的产品完全替换 MathHub。两者最大的区别是 MathHub 使用C/C++ 编写且面向本机编译，而VMath 则采用C#编写并且面向托管代码和利用AoT技术面向本机代码编译。

VMath 只支持.NET 8或更高版本，目的是为了充分利用.NET的最新特性以优化性能：包括.NET 5引入的 SIMD 指令编译和 .NET 7引入的泛型数学功能。

核心模块

Vorcyc. Mathematics 命名空间

**Vorcyc.Mathematics.ArrayExtension 类**

**方法清单及说明**

**通用方法**

1. **Copy**
   * public static T[] Copy<T>(this T[] source)
     1. 复制整个数组。
   * public static T[] Copy<T>(this T[] source, int length)
     1. 复制数组的指定长度。
2. **InitializeArray**
   * public static T[]? InitializeArray<T>(int length, T initialValue = default)
     1. 初始化一个指定长度的数组，并用初始值填充。
3. **Fill**
   * public static void Fill<T>(this T[] array, T value)
     1. 用指定值填充整个数组。
   * public static void Fill<T>(this T[] array, int start, int end, T value)
     1. 用指定值填充数组的指定范围。
   * public static void Fill<T>(this Span<T> array, Range range, T value)
     1. 用指定值填充数组的指定范围。
4. **FillWithRandomNumber**
   * public static void FillWithRandomNumber(this Span<float> span)
     1. 用随机浮点数填充数组。
   * public static void FillWithRandomNumber<T>(this T[] array) where T : IFloatingPointIeee754<T>
     1. 用随机数填充数组。
   * public static void FillWithRandomNumber<T>(this T[] array, Range range) where T : IFloatingPointIeee754<T>
     1. 用随机数填充数组的指定范围。
   * public static void FillWithRandomNumber(this int[] array, (int max, int min)? limit = null)
     1. 用随机数填充数组。
   * public static void FillWithRandomNumber(this int[] array, Range range)
     1. 用随机数填充数组的指定范围。
   * public static void FillWithRandomNumber(this long[] array)
     1. 用随机数填充数组。
   * public static void FillWithRandomNumber(this long[] array, Range range)
     1. 用随机数填充数组的指定范围。
5. **GetInner**
   * public static IEnumerable<T> GetInner<T>(this T[] array, int start, int length)
     1. 获取数组的内部片段，返回一个可枚举对象。
   * public static T[] GetInnerArray<T>(this T[] array, int start, int length)
     1. 获取数组的内部片段，返回一个新的数组。
6. **RemoveSegment**
   * public static T[] RemoveSegment<T>(this T[] array, int start, int length)
     1. 移除数组中的一部分，并返回移除后的新数组。
7. **Merge**
   * public static T[] Merge<T>(this T[] leading, T[] following)
     1. 合并两个数组，返回合并后的新数组。
8. **ToString**
   * public static string ToString<T>(this IEnumerable<T> collection)
     1. 将集合转换为字符串表示。
9. **FastCopyFragment**
   * public static T[] FastCopyFragment<T>(this T[] source, int size, int sourceOffset = 0, int destinationOffset = 0) where T : unmanaged
     1. 快速复制数组的片段到一个新数组。
10. **FastCopyTo**
    * public static void FastCopyTo<T>(this T[] source, T[] destination, int size, int sourceOffset = 0, int destinationOffset = 0)
      1. 快速复制数组的元素到另一个数组。
11. **Repeat**
    * public static T[] Repeat<T>(this T[] source, int n)
      1. 创建一个包含源数组重复指定次数的新数组。

**单精度方法**

1. **FastCopy**
   * public static float[] FastCopy(this float[] source)
     1. 创建一个单精度浮点数组的快速副本。
2. **FastCopyTo**
   * public static void FastCopyTo(this float[] source, float[] destination, int size, int sourceOffset = 0, int destinationOffset = 0)
     1. 快速复制单精度浮点数组的元素到另一个数组。
3. **FastCopyFragment**
   * public static float[] FastCopyFragment(this float[] source, int size, int sourceOffset = 0, int destinationOffset = 0)
     1. 快速复制单精度浮点数组的片段到一个新数组。
4. **Merge**
   * public static float[] Merge(this float[] source, float[] another)
     1. 合并两个单精度浮点数组，返回合并后的新数组。
5. **Repeat**
   * public static float[] Repeat(this float[] source, int n)
     1. 创建一个包含源单精度浮点数组重复指定次数的新数组。
6. **PadZeros**
   * public static float[] PadZeros(this float[] source, int size)
     1. 创建一个指定大小的零填充单精度浮点数组。
   * public static T[] PadZeros<T>(this T[] source, int size)
     1. 创建一个指定大小的零填充数组。

**双精度方法**

1. **FastCopy**
   * public static double[] FastCopy(this double[] source)
     1. 创建一个双精度浮点数组的快速副本。
2. **FastCopyTo**
   * public static void FastCopyTo(this double[] source, double[] destination, int size, int sourceOffset = 0, int destinationOffset = 0)
     1. 快速复制双精度浮点数组的元素到另一个数组。
3. **FastCopyFragment**
   * public static double[] FastCopyFragment(this double[] source, int size, int sourceOffset = 0, int destinationOffset = 0)
     1. 快速复制双精度浮点数组的片段到一个新数组。
4. **Merge**
   * public static double[] Merge(this double[] source, double[] another)
     1. 合并两个双精度浮点数组，返回合并后的新数组。
5. **Repeat**
   * public static double[] Repeat(this double[] source, int n)
     1. 创建一个包含源双精度浮点数组重复指定次数的新数组。
6. **PadZeros**
   * public static double[] PadZeros(this double[] source, int size)
     1. 创建一个指定大小的零填充双精度浮点数组。

**其他方法**

1. **ToFloats**
   * public static float[] ToFloats(this IEnumerable<double> values)
     1. 将双精度值的可枚举对象转换为单精度数组。
2. **ToDoubles**
   * public static double[] ToDoubles(this IEnumerable<float> values)
     1. 将单精度值的可枚举对象转换为双精度数组。
3. **Last**
   * public static T Last<T>(this T[] array)
     1. 获取数组的最后一个元素。
4. **First**
   * public static T First<T>(this T[] array)
     1. 获取数组的第一个元素。

**代码示例**

using System;

using System.Linq;

using Vorcyc.Mathematics;

public class ArrayExtensionExample

{

public static void Main()

{

// 创建一个整数数组

int[] array = { 1, 2, 3, 4, 5 };

// 复制数组

int[] copiedArray = array.Copy();

Console.WriteLine("Copied Array: " + string.Join(", ", copiedArray));

// 初始化一个长度为5，初始值为10的数组

int[] initializedArray = ArrayExtension.InitializeArray(5, 10);

Console.WriteLine("Initialized Array: " + string.Join(", ", initializedArray));

// 用指定值填充数组

array.Fill(9);

Console.WriteLine("Filled Array: " + string.join(", ", array));

// 用随机数填充数组

array.FillWithRandomNumber((max: 100, min: 1));

Console.WriteLine("Array Filled with Random Numbers: " + string.Join(", ", array));

// 获取数组的内部片段

int[] innerArray = array.GetInnerArray(1, 3);

Console.WriteLine("Inner Array: " + string.Join(", ", innerArray));

// 移除数组中的一部分

int[] removedSegmentArray = array.RemoveSegment(1, 2);

Console.WriteLine("Array after Removing Segment: " + string.Join(", ", removedSegmentArray));

// 合并两个数组

int[] mergedArray = array.Merge(new int[] { 6, 7, 8 });

Console.WriteLine("Merged Array: " + string.Join(", ", mergedArray));

// 将数组转换为字符串

string arrayString = array.ToString();

Console.WriteLine("Array as String: " + arrayString);

// 快速复制数组的片段

int[] fastCopiedFragment = array.FastCopyFragment(3, 1);

Console.WriteLine("Fast Copied Fragment: " + string.Join(", ", fastCopiedFragment));

// 创建一个包含源数组重复3次的新数组

int[] repeatedArray = array.Repeat(3);

Console.WriteLine("Repeated Array: " + string.Join(", ", repeatedArray));

// 将数组分割成指定长度的段

var segments = array.Split(2);

Console.WriteLine("Array Segments:");

foreach (var segment in segments)

{

Console.WriteLine(string.Join(", ", segment));

}

// 压缩数组并生成数组片段序列

var zippedSegments = array.Zip(3);

Console.WriteLine("Zipped Array Segments:");

foreach (var segment in zippedSegments)

{

Console.WriteLine(string.Join(", ", segment));

}

// 通过平均原数组的片段，将单精度浮点数组转换为指定长度的新数组

float[] floatArray = { 1.1f, 2.2f, 3.3f, 4.4f, 5.5f };

float[] transformedArray = floatArray.TransformToArray(3);

Console.WriteLine("Transformed Array: " + string.Join(", ", transformedArray));

}

}

**Vorcyc.Mathematics.BaseConverter 类**

Vorcyc.Mathematics.BaseConverter 是一个静态类，提供了将任意实现了 IBinaryInteger<TSelf> 接口的整数类型转换为指定进制的字符串表示和从指定进制的字符串表示转换为整数的方法。该类包含以下主要功能：

**方法清单及说明**

1. **ToBaseString**
   * public static string ToBaseString<TSelf>(this TSelf integer, TSelf baseNumber) where TSelf : IBinaryInteger<TSelf>
     1. 将任意实现了 IBinaryInteger<TSelf> 接口的整数类型转换为指定进制的字符串表示。进制的范围在2到94之间。
     2. **参数**:
        1. integer: 要转换的整数。
        2. baseNumber: 进制数。要求大于或等于2，小于或等于94。
     3. **返回值**: 字符串形式的进制数。
     4. **异常**:
        1. ArgumentOutOfRangeException: 当整数为负数或进制数不在2到94之间时抛出。
2. **FromBaseString**
   * public static TSelf FromBaseString<TSelf>(this string value, TSelf baseNumber) where TSelf : IBinaryInteger<TSelf>
     1. 将指定进制的字符串表示转换为任意实现了 IBinaryInteger<TSelf> 接口的整数类型。进制的范围在2到94之间。
     2. **参数**:
        1. value: 要转换的字符串。
        2. baseNumber: 进制数。要求大于或等于2，小于或等于94。
     3. **返回值**: 转换后的整数。
     4. **异常**:
        1. ArgumentOutOfRangeException: 当进制数不在2到94之间或字符串包含无效字符时抛出。

**代码示例**

using System;

using Vorcyc.Mathematics;

public class BaseConverterExample

{

public static void Main()

{

// 将整数转换为指定进制的字符串表示

int number = 100;

int baseNumber = 16;

// 使用 ToBaseString 方法将整数 100 转换为 16 进制的字符串表示

string baseString = number.ToBaseString(baseNumber);

Console.WriteLine($"Number {number} in base {baseNumber} is: {baseString}");

// 将指定进制的字符串表示转换为整数

string value = "64";

// 使用 FromBaseString 方法将 16 进制的字符串 "64" 转换为整数

int convertedNumber = value.FromBaseString(baseNumber);

Console.WriteLine($"String {value} in base {baseNumber} is: {convertedNumber}");

}

}

**Vorcyc.Mathematics.BitMathExtension 类**

Vorcyc.Mathematics.BitMathExtension 是一个静态类，提供了多种用于位运算的扩展方法。该类包含以下主要功能：

**方法清单及说明**

1. **IsPowerOf2**
   * public static bool IsPowerOf2(this uint x)
   * public static bool IsPowerOf2(this ulong x)
   * public static bool IsPowerOf2(this int x)
   * public static bool IsPowerOf2(this long x)
   * 验证一个数是否是2的幂。
2. **NextPowerOf2**
   * public static int NextPowerOf2(this int x)
   * public static ulong NextPowerOf2(this ulong value)
   * public static uint NextPowerOf2(this uint value)
   * 获取下一个2的幂。
3. **PreviousPowerOf2**
   * public static int PreviousPowerOf2(this int x)
   * public static ulong PreviousPowerOf2(this ulong value)
   * public static uint PreviousPowerOf2(this uint value)
   * 获取上一个2的幂。
4. **CountBitsSet**
   * public static int CountBitsSet(this uint value)
   * public static int CountBitsSet(this ulong value)
   * 计算设置的位数。
5. **CountBitsCleared**
   * public static int CountBitsCleared(this uint value)
   * public static int CountBitsCleared(this ulong value)
   * 计算未设置的位数。
6. **CreateBitMask**
   * public static ulong CreateBitMask(this int bitCount)
   * 创建一个具有给定位数的位掩码。
7. **CountTrailingZeros**
   * public static int CountTrailingZeros(this uint value)
   * public static int CountTrailingZeros(this ulong value)
   * 计算从最低位开始的连续0的个数。
8. **CountLeadingZeros**
   * public static int CountLeadingZeros(this uint value)
   * public static int CountLeadingZeros(this ulong value)
   * 计算从最高位开始的连续0的个数。
9. **CountTrailingOnes**
   * public static int CountTrailingOnes(this uint value)
   * public static int CountTrailingOnes(this ulong value)
   * 计算从最低位开始的连续1的个数。
10. **CountLeadingOnes**
    * public static int CountLeadingOnes(this uint value)
    * public static int CountLeadingOnes(this ulong value)
    * 计算从最高位开始的连续1的个数。
11. **GetSetBitPositions**
    * public static IEnumerable<int> GetSetBitPositions(this ulong value)
    * public static IEnumerable<int> GetSetBitPositions(this uint value)
    * 返回设置位的位置。
12. **GetClearedBitPositions**
    * public static IEnumerable<int> GetClearedBitPositions(this uint value)
    * public static IEnumerable<int> GetClearedBitPositions(this ulong value)
    * 返回未设置位的位置。
13. **IsOdd**
    * public static bool IsOdd(this long value)
    * public static bool IsOdd(this ulong value)
    * public static bool IsOdd(this int value)
    * public static bool IsOdd(this uint value)
    * 判断是否是奇数。
14. **IsEven**
    * public static bool IsEven(this long value)
    * public static bool IsEven(this ulong value)
    * public static bool IsEven(this int value)
    * public static bool IsEven(this uint value)
    * 判断是否是偶数。

**代码示例**

以下是一个使用 BitMathExtension 类中多个方法的示例，并在示例中加入了注释：

using System;

using Vorcyc.Mathematics;

public class BitMathExtensionExample

{

public static void Main()

{

// 验证一个数是否是2的幂

int number = 16;

bool isPowerOf2 = number.IsPowerOf2();

Console.WriteLine($"{number} is power of 2: {isPowerOf2}");

// 获取下一个2的幂

int nextPowerOf2 = number.NextPowerOf2();

Console.WriteLine($"Next power of 2 after {number} is: {nextPowerOf2}");

// 获取上一个2的幂

int previousPowerOf2 = number.PreviousPowerOf2();

Console.WriteLine($"Previous power of 2 before {number} is: {previousPowerOf2}");

// 计算设置的位数

uint value = 29;

int bitsSet = value.CountBitsSet();

Console.WriteLine($"Number of bits set in {value} is: {bitsSet}");

// 计算未设置的位数

int bitsCleared = value.CountBitsCleared();

Console.WriteLine($"Number of bits cleared in {value} is: {bitsCleared}");

// 创建一个具有给定位数的位掩码

int bitCount = 5;

ulong bitMask = bitCount.CreateBitMask();

Console.WriteLine($"Bit mask with {bitCount} bits is: {bitMask}");

// 计算从最低位开始的连续0的个数

int trailingZeros = value.CountTrailingZeros();

Console.WriteLine($"Number of trailing zeros in {value} is: {trailingZeros}");

// 计算从最高位开始的连续0的个数

int leadingZeros = value.CountLeadingZeros();

Console.WriteLine($"Number of leading zeros in {value} is: {leadingZeros}");

// 判断是否是奇数

bool isOdd = number.IsOdd();

Console.WriteLine($"{number} is odd: {isOdd}");

// 判断是否是偶数

bool isEven = number.IsEven();

Console.WriteLine($"{number} is even: {isEven}");

}

}

**Vorcyc.Mathematics.ConstantsFp32 类**

Vorcyc.Mathematics.ConstantsFp32 是一个静态类，定义了一组常用的单精度浮点数常量。这些常量在数学计算中非常有用。以下是每个常量的详细说明：

**常量清单及说明**

1. **E**
   * public const float E = 2.71828182845904523536f;
   * 自然对数的底数 e。
2. **LOG2E**
   * public const float LOG2E = 1.44269504088896340736f;
   * 以 2 为底的 e 的对数。
3. **LOG10E**
   * public const float LOG10E = 0.434294481903251827651f;
   * 以 10 为底的 e 的对数。
4. **LN2**
   * public const float LN2 = 0.693147180559945309417f;
   * 2 的自然对数。
5. **LN10**
   * public const float LN10 = 2.30258509299404568402f;
   * 10 的自然对数。
6. **PI**
   * public const float PI = 3.1415926535897932384626433832795f;
   * 圆周率 π。
7. **PI\_2**
   * public const float PI\_2 = 1.57079632679489661923f;
   * π 的一半。
8. **PI\_4**
   * public const float PI\_4 = 0.785398163397448309616f;
   * π 的四分之一。
9. **\_1\_PI**
   * public const float \_1\_PI = 0.318309886183790671538f;
   * 1 除以 π。
10. **\_2\_PI**
    * public const float \_2\_PI = 0.636619772367581343076f;
    * 2 除以 π。
11. **\_2\_SQRT\_PI**
    * public const float \_2\_SQRT\_PI = 1.12837916709551257390f;
    * 2 除以 π 的平方根。
12. **\_SQRT2**
    * public const float \_SQRT2 = 1.41421356237309504880f;
    * 2 的平方根。
13. **\_SQRT1\_2**
    * public const float \_SQRT1\_2 = 0.707106781186547524401f;
    * 1 除以 2 的平方根。
14. **TWO\_PI**
    * public const float TWO\_PI = 6.28318530717958647692f;
    * 2 倍的 π。
15. **TOLERANCE**
    * public const float TOLERANCE = 1E-09f;
    * 公差，用于比较浮点数的精度。
16. **RADIANS\_PER\_DEGREE**
    * public const float RADIANS\_PER\_DEGREE = 0.01745329f;
    * 每度的弧度数。
17. **DEGREES\_PER\_RADIAN**
    * public const float DEGREES\_PER\_RADIAN = 57.29578f;
    * 每弧度的度数。
18. **EPSILON**
    * public const float EPSILON = 0.001f;
    * 用于浮点数比较的极小值。

**代码示例**

以下是一个使用 ConstantsFp32 类中常量的示例，并在示例中加入了注释：

using System;

using Vorcyc.Mathematics;

public class ConstantsFp32Example

{

public static void Main()

{

// 使用自然对数的底数 e

float e = ConstantsFp32.E;

Console.WriteLine($"e: {e}");

// 使用圆周率 π

float pi = ConstantsFp32.PI;

Console.WriteLine($"π: {pi}");

// 使用每度的弧度数

float radiansPerDegree = ConstantsFp32.RADIANS\_PER\_DEGREE;

Console.WriteLine($"Radians per degree: {radiansPerDegree}");

// 使用每弧度的度数

float degreesPerRadian = ConstantsFp32.DEGREES\_PER\_RADIAN;

Console.WriteLine($"Degrees per radian: {degreesPerRadian}");

// 使用公差

float tolerance = ConstantsFp32.TOLERANCE;

Console.WriteLine($"Tolerance: {tolerance}");

// 使用 epsilon

float epsilon = ConstantsFp32.EPSILON;

Console.WriteLine($"Epsilon: {epsilon}");

}

}

**Vorcyc.Mathematics.NumberMapper 类**

Vorcyc.Mathematics.NumberMapper 是一个静态类，提供了将一个值从一个范围映射到另一个范围的方法。该类包含以下主要功能：

**方法清单及说明**

1. **Map**
   * public static TNumber Map<TNumber>(this TNumber number, TNumber inMin, TNumber inMax, TNumber outMin, TNumber outMax, InputValueOutOfRangeHandleBehavior handleBehavior = InputValueOutOfRangeHandleBehavior.Saturating) where TNumber : unmanaged, INumber<TNumber>
     1. 将输入值从一个范围映射到另一个范围。
     2. **参数**:
        1. number: 要映射的输入值。
        2. inMin: 输入范围的最小值。
        3. inMax: 输入范围的最大值。
        4. outMin: 输出范围的最小值。
        5. outMax: 输出范围的最大值。
        6. handleBehavior: 指定当输入值超出范围时的处理行为。默认值为 InputValueOutOfRangeHandleBehavior.Saturating。
     3. **返回值**: 映射到输出范围的值。
     4. **异常**:
        1. ArgumentOutOfRangeException: 当 inMin 大于或等于 inMax，或 outMin 大于或等于 outMax 时抛出。
        2. ArgumentException: 当 number 超出输入范围且 handleBehavior 为 InputValueOutOfRangeHandleBehavior.ThrowException 时抛出。
2. **Map**
   * public static bool Map<TFloatingNumber>(TFloatingNumber input, TFloatingNumber inputMin, TFloatingNumber inputMax, TFloatingNumber outputMin, TFloatingNumber outputMax, out TFloatingNumber result) where TFloatingNumber : unmanaged, IFloatingPointIeee754<TFloatingNumber>
     1. 将浮点数从一个范围映射到另一个范围，并返回一个布尔值指示是否成功。
     2. **参数**:
        1. input: 要映射的浮点数。
        2. inputMin: 输入范围的最小值。
        3. inputMax: 输入范围的最大值。
        4. outputMin: 输出范围的最小值。
        5. outputMax: 输出范围的最大值。
        6. result: 如果映射成功，则为映射后的浮点数；否则为 NaN。
     3. **返回值**: 如果映射成功，则为 true；否则为 false。

**枚举**

1. **InputValueOutOfRangeHandleBehavior**
   * 定义当输入值超出指定范围时的处理行为。
   * **值**:
     1. Saturating: 将输入值夹紧到最近的边界值。
     2. ThrowException: 当输入值超出范围时抛出异常。

**代码示例**

以下是一个使用 NumberMapper 类中方法的示例，并在示例中加入了注释：

using System;

using Vorcyc.Mathematics;

public class NumberMapperExample

{

public static void Main()

{

// 将整数从一个范围映射到另一个范围

int number = 5;

int inMin = 0;

int inMax = 10;

int outMin = 0;

int outMax = 100;

int mappedValue = number.Map(inMin, inMax, outMin, outMax);

Console.WriteLine($"Mapped value: {mappedValue}");

// 将浮点数从一个范围映射到另一个范围，并返回一个布尔值指示是否成功

float input = 0.5f;

float inputMin = 0.0f;

float inputMax = 1.0f;

float outputMin = 0.0f;

float outputMax = 100.0f;

bool success = NumberMapper.Map(input, inputMin, inputMax, outputMin, outputMax, out float result);

Console.WriteLine($"Mapping successful: {success}, Mapped value: {result}");

}

}

**Vorcyc.Mathematics.PinnableArray 类**

**介绍**

Vorcyc.Mathematics.PinnableArray 是一个泛型类，表示一个可固定内存的一维数组，用以提供高性能内存访问机制和数组的操作。该类提供了多种方法和属性来操作数组，并且可以将数组固定在内存中，以便与非托管代码进行交互。该类包含以下主要功能：

**方法清单及说明**

1. **构造函数**
   * public PinnableArray(ArraySegment<T> segment, bool pin = false)
     1. 使用 ArraySegment<T> 初始化 PinnableArray 实例。
   * public PinnableArray(Span<T> span, bool pin = false)
     1. 使用 Span<T> 初始化 PinnableArray 实例。
   * public PinnableArray(T[] array, bool pin = false)
     1. 使用数组初始化 PinnableArray 实例。
   * public PinnableArray(T[] array, int offset, int count, bool pin = false)
     1. 使用数组的部分内容初始化 PinnableArray 实例。
   * public PinnableArray(int count, bool pin = false)
     1. 使用指定长度初始化 PinnableArray 实例。
2. **Pin**
   * public void Pin()
     1. 将 PinnableArray 固定在内存中，以便与非托管代码进行交互。
3. **Unpin**
   * public void Unpin()
     1. 取消固定 PinnableArray，使其可以被垃圾回收。
4. **AsSpan**
   * public Span<T> AsSpan()
     1. 返回表示当前数组的 Span<T>。
   * public Span<T> AsSpan(int start, int length)
     1. 返回表示当前数组部分内容的 Span<T>。
   * public Span<T> AsSpan(int start)
     1. 返回表示当前数组从指定位置开始的 Span<T>。
   * public Span<T> AsSpan(System.Index startIndex)
     1. 返回表示当前数组从指定索引开始的 Span<T>。
   * public Span<T> AsSpan(Range range)
     1. 返回表示当前数组指定范围的 Span<T>。
5. **Fill**
   * public void Fill(T value)
     1. 用指定值填充整个数组。
   * public void Fill(T startValue, T step)
     1. 用指定的起始值和步长填充数组。
   * public void FillWith<TNumber>(TNumber number) where TNumber : unmanaged
     1. 用另一个非托管类型的值填充数组。
   * public void FillWithRandomNumber()
     1. 用随机数填充数组。
6. **Each**
   * public void Each(Func<int, T, T> func)
     1. 使用指定的函数遍历数组的每个元素。
   * public void Each(Func<int, T?, T, T?, T> func, Direction direction = Direction.Forward)
     1. 使用指定的函数遍历数组的每个元素，并指定遍历方向。
7. **Max**
   * public T Max()
     1. 返回数组中的最大值。
   * public T Max(int start, int length)
     1. 返回数组指定范围内的最大值。
   * public async Task<T> MaxAsync(int? numberOfWorkers = null, bool useTPL = false)
     1. 异步返回数组中的最大值。
   * public async Task<T> MaxAsync(int start, int length, int? numberOfWorkers = null, bool useTPL = false)
     1. 异步返回数组指定范围内的最大值。
8. **Min**
   * public T Min()
     1. 返回数组中的最小值。
   * public T Min(int start, int length)
     1. 返回数组指定范围内的最小值。
   * public async Task<T> MinAsync(int? numberOfWorkers = null, bool useTPL = false)
     1. 异步返回数组中的最小值。
   * public async Task<T> MinAsync(int start, int length, int? numberOfWorkers = null, bool useTPL = false)
     1. 异步返回数组指定范围内的最小值。
9. **Map**
   * public void MapIn(T fromMin, T fromMax, T toMin, T toMax)
     1. 将数组中的值从一个范围映射到另一个范围。
   * public void MapIn(T toMin, T toMax)
     1. 将数组中的值从当前范围映射到指定范围。
   * public PinnableArray<T> Map(T fromMin, T fromMax, T toMin, T toMax)
     1. 返回一个新的 PinnableArray，其中的值从一个范围映射到另一个范围。
   * public PinnableArray<T> Map(T toMin, T toMax)
     1. 返回一个新的 PinnableArray，其中的值从当前范围映射到指定范围。
10. **Dot**
    * public T Dot(PinnableArray<T> another)
      1. 计算当前数组与另一个 PinnableArray 的点积。
11. **Dispose**
    * public void Dispose()
      1. 释放 PinnableArray 占用的资源。

**枚举**

1. **Direction**
   * 定义遍历数组的方向。
   * **值**:
     1. Forward: 从前向后遍历。
     2. Inverse: 从后向前遍历。

**属性**

1. **IsPinned**
   * public bool IsPinned { get; }
   * 获取当前数组是否已固定在内存中。
2. **Length**
   * public int Length { get; }
   * 获取当前数组的长度。
3. **Values**
   * public T[] Values { get; }
   * 获取当前数组的普通数组形式。

**操作符重载**

1. **+**
   * public static PinnableArray<T> operator +(PinnableArray<T> left, PinnableArray<T> right)
   * public static PinnableArray<T> operator +(PinnableArray<T> left, T right)
   * public static PinnableArray<T> operator +(T left, PinnableArray<T> right)
2. **-**
   * public static PinnableArray<T> operator -(PinnableArray<T> left, PinnableArray<T> right)
   * public static PinnableArray<T> operator -(PinnableArray<T> left, T right)
   * public static PinnableArray<T> operator -(T left, PinnableArray<T> right)
3. **\***
   * public static PinnableArray<T> operator \*(PinnableArray<T> left, PinnableArray<T> right)
   * public static PinnableArray<T> operator \*(PinnableArray<T> left, T right)
   * public static PinnableArray<T> operator \*(T left, PinnableArray<T> right)
4. **/**
   * public static PinnableArray<T> operator /(PinnableArray<T> left, PinnableArray<T> right)
   * public static PinnableArray<T> operator /(PinnableArray<T> left, T right)
   * public static PinnableArray<T> operator /(T left, PinnableArray<T> right)

**代码示例**

以下是一个使用 PinnableArray 类中多个方法的示例，并在示例中加入了注释：

using System;

using System.Threading.Tasks;

using Vorcyc.Mathematics;

public class PinnableArrayExample

{

public static async Task Main()

{

// 使用数组初始化 PinnableArray 实例

int[] array = { 1, 2, 3, 4, 5 };

PinnableArray<int> pinnableArray = new PinnableArray<int>(array);

// 将数组固定在内存中

pinnableArray.Pin();

Console.WriteLine($"IsPinned: {pinnableArray.IsPinned}");

// 获取数组的长度

int length = pinnableArray.Length;

Console.WriteLine($"Length: {length}");

// 获取数组的普通数组形式

int[] values = pinnableArray;

Console.WriteLine($"Values: {string.Join(", ", values)}");

// 用指定值填充整个数组

pinnableArray.Fill(10);

Console.WriteLine($"Filled Values: {string.Join(", ", pinnableArray.Values)}");

// 用指定的起始值和步长填充数组

pinnableArray.Fill(1, 2);

Console.WriteLine($"Filled with step Values: {string.Join(", ", pinnableArray.Values)}");

// 用随机数填充数组

pinnableArray.FillWithRandomNumber();

Console.WriteLine($"Filled with random numbers: {string.Join(", ", pinnableArray.Values)}");

// 遍历数组的每个元素并进行操作

pinnableArray.Each((index, value) => value \* 2);

Console.WriteLine($"Each element multiplied by 2: {string.Join(", ", pinnableArray.Values)}");

// 获取数组中的最大值

int maxValue = pinnableArray.Max();

Console.WriteLine($"Max value: {maxValue}");

// 获取数组中的最小值

int minValue = pinnableArray.Min();

Console.WriteLine($"Min value: {minValue}");

// 异步获取数组中的最大值

int maxAsyncValue = await pinnableArray.MaxAsync();

Console.WriteLine($"Max async value: {maxAsyncValue}");

// 异步获取数组中的最小值

int minAsyncValue = await pinnableArray.MinAsync();

Console.WriteLine($"Min async value: {minAsyncValue}");

// 将数组中的值从一个范围映射到另一个范围

pinnableArray.MapIn(0, 1, 0, 100);

Console.WriteLine($"Mapped values: {string.Join(", ", pinnableArray.Values)}");

// 计算当前数组与另一个 PinnableArray 的点积

PinnableArray<int> anotherArray = new PinnableArray<int>(new int[] { 1, 2, 3, 4, 5 });

int dotProduct = pinnableArray.Dot(anotherArray);

Console.WriteLine($"Dot product: {dotProduct}");

// 使用索引器访问和修改数组元素

pinnableArray[0] = 42;

Console.WriteLine($"First element after modification: {pinnableArray[0]}");

// 获取数组的 Span 表示

Span<int> span = pinnableArray.AsSpan();

Console.WriteLine($"Span values: {string.Join(", ", span.ToArray())}");

// 取消固定数组

pinnableArray.Unpin();

Console.WriteLine($"IsPinned: {pinnableArray.IsPinned}");

// 释放资源

pinnableArray.Dispose();

}

}

**SimpleRNG\_Fp32 和 SimpleRNG\_Fp64 类**

Vorcyc.Mathematics.SimpleRNG\_Fp32 和 Vorcyc.Mathematics.SimpleRNG\_Fp64 是两个静态类，分别用于生成 32 位和 64 位浮点数的随机数。它们基于 George Marsaglia 的 MWC（乘法与进位）算法，能够生成通过 Marsaglia 的 DIEHARD 随机数生成器测试的随机数。

**方法清单及说明**

SimpleRNG\_Fp64

1. **SetSeed**
   * public static void SetSeed(uint u, uint v)
     1. 设置随机数生成器的种子。
   * public static void SetSeed(uint u)
     1. 设置随机数生成器的种子。
   * public static void SetSeedFromSystemTime()
     1. 使用系统时间设置随机数生成器的种子。
2. **GetUniform**
   * public static double GetUniform()
     1. 生成一个 (0, 1) 区间内的均匀分布随机数。
3. **GetNormal**
   * public static double GetNormal()
     1. 生成一个均值为 0，标准差为 1 的正态分布随机数。
   * public static double GetNormal(double mean, double standardDeviation)
     1. 生成一个指定均值和标准差的正态分布随机数。
4. **GetExponential**
   * public static double GetExponential()
     1. 生成一个均值为 1 的指数分布随机数。
   * public static double GetExponential(double mean)
     1. 生成一个指定均值的指数分布随机数。
5. **GetGamma**
   * public static double GetGamma(double shape, double scale)
     1. 生成一个指定形状和尺度参数的伽马分布随机数。
6. **GetChiSquare**
   * public static double GetChiSquare(double degreesOfFreedom)
     1. 生成一个指定自由度的卡方分布随机数。
7. **GetInverseGamma**
   * public static double GetInverseGamma(double shape, double scale)
     1. 生成一个指定形状和尺度参数的逆伽马分布随机数。
8. **GetWeibull**
   * public static double GetWeibull(double shape, double scale)
     1. 生成一个指定形状和尺度参数的威布尔分布随机数。
9. **GetCauchy**
   * public static double GetCauchy(double median, double scale)
     1. 生成一个指定中位数和尺度参数的柯西分布随机数。
10. **GetStudentT**
    * public static double GetStudentT(double degreesOfFreedom)
      1. 生成一个指定自由度的 t 分布随机数。
11. **GetLaplace**
    * public static double GetLaplace(double mean, double scale)
      1. 生成一个指定均值和尺度参数的拉普拉斯分布（双指数分布）随机数。
12. **GetLogNormal**
    * public static double GetLogNormal(double mu, double sigma)
      1. 生成一个对数正态分布随机数。
13. **GetBeta**
    * public static double GetBeta(double a, double b)
      1. 生成一个指定参数的贝塔分布随机数。

SimpleRNG\_Fp32

1. **SetSeed**
   * public static void SetSeed(uint u, uint v)
     1. 设置随机数生成器的种子。
   * public static void SetSeed(uint u)
     1. 设置随机数生成器的种子。
   * public static void SetSeedFromSystemTime()
     1. 使用系统时间设置随机数生成器的种子。
2. **GetUniform**
   * public static float GetUniform()
     1. 生成一个 (0, 1) 区间内的均匀分布随机数。
3. **GetNormal**
   * public static float GetNormal()
     1. 生成一个均值为 0，标准差为 1 的正态分布随机数。
   * public static float GetNormal(float mean, float standardDeviation)
     1. 生成一个指定均值和标准差的正态分布随机数。
4. **GetExponential**
   * public static float GetExponential()
     1. 生成一个均值为 1 的指数分布随机数。
   * public static float GetExponential(float mean)
     1. 生成一个指定均值的指数分布随机数。
5. **GetGamma**
   * public static float GetGamma(float shape, float scale)
     1. 生成一个指定形状和尺度参数的伽马分布随机数。
6. **GetChiSquare**
   * public static float GetChiSquare(float degreesOfFreedom)
     1. 生成一个指定自由度的卡方分布随机数。
7. **GetInverseGamma**
   * public static float GetInverseGamma(float shape, float scale)
     1. 生成一个指定形状和尺度参数的逆伽马分布随机数。
8. **GetWeibull**
   * public static float GetWeibull(float shape, float scale)
     1. 生成一个指定形状和尺度参数的威布尔分布随机数。
9. **GetCauchy**
   * public static float GetCauchy(float median, float scale)
     1. 生成一个指定中位数和尺度参数的柯西分布随机数。
10. **GetStudentT**
    * public static float GetStudentT(float degreesOfFreedom)
      1. 生成一个指定自由度的 t 分布随机数。
11. **GetLaplace**
    * public static float GetLaplace(float mean, float scale)
      1. 生成一个指定均值和尺度参数的拉普拉斯分布（双指数分布）随机数。
12. **GetLogNormal**
    * public static float GetLogNormal(float mu, float sigma)
      1. 生成一个对数正态分布随机数。
13. **GetBeta**
    * public static float GetBeta(float a, float b)
      1. 生成一个指定参数的贝塔分布随机数。

**代码示例**

以下是一个使用 SimpleRNG\_Fp32 和 SimpleRNG\_Fp64 类中多个方法的示例，并在示例中加入了注释：

using System;

using Vorcyc.Mathematics;

public class SimpleRNGExample

{

public static void Main()

{

// 使用 SimpleRNG\_Fp64 生成随机数

SimpleRNG\_Fp64.SetSeedFromSystemTime();

double uniform64 = SimpleRNG\_Fp64.GetUniform();

double normal64 = SimpleRNG\_Fp64.GetNormal();

double exponential64 = SimpleRNG\_Fp64.GetExponential();

double gamma64 = SimpleRNG\_Fp64.GetGamma(2.0, 2.0);

double chiSquare64 = SimpleRNG\_Fp64.GetChiSquare(2.0);

double inverseGamma64 = SimpleRNG\_Fp64.GetInverseGamma(2.0, 2.0);

double weibull64 = SimpleRNG\_Fp64.GetWeibull(2.0, 2.0);

double cauchy64 = SimpleRNG\_Fp64.GetCauchy(0.0, 1.0);

double studentT64 = SimpleRNG\_Fp64.GetStudentT(2.0);

double laplace64 = SimpleRNG\_Fp64.GetLaplace(0.0, 1.0);

double logNormal64 = SimpleRNG\_Fp64.GetLogNormal(0.0, 1.0);

double beta64 = SimpleRNG\_Fp64.GetBeta(2.0, 2.0);

Console.WriteLine("SimpleRNG\_Fp64:");

Console.WriteLine($"Uniform: {uniform64}");

Console.WriteLine($"Normal: {normal64}");

Console.WriteLine($"Exponential: {exponential64}");

Console.WriteLine($"Gamma: {gamma64}");

Console.WriteLine($"ChiSquare: {chiSquare64}");

Console.WriteLine($"InverseGamma: {inverseGamma64}");

Console.WriteLine($"Weibull: {weibull64}");

Console.WriteLine($"Cauchy: {cauchy64}");

Console.WriteLine($"StudentT: {studentT64}");

Console.WriteLine($"Laplace: {laplace64}");

Console.WriteLine($"LogNormal: {logNormal64}");

Console.WriteLine($"Beta: {beta64}");

// 使用 SimpleRNG\_Fp32 生成随机数

SimpleRNG\_Fp32.SetSeedFromSystemTime();

float uniform32 = SimpleRNG\_Fp32.GetUniform();

float normal32 = SimpleRNG\_Fp32.GetNormal();

float exponential32 = SimpleRNG\_Fp32.GetExponential();

float gamma32 = SimpleRNG\_Fp32.GetGamma(2.0f, 2.0f);

float chiSquare32 = SimpleRNG\_Fp32.GetChiSquare(2.0f);

float inverseGamma32 = SimpleRNG\_Fp32.GetInverseGamma(2.0f, 2.0f);

float weibull32 = SimpleRNG\_Fp32.GetWeibull(2.0f, 2.0f);

float cauchy32 = SimpleRNG\_Fp32.GetCauchy(0.0f, 1.0f);

float studentT32 = SimpleRNG\_Fp32.GetStudentT(2.0f);

float laplace32 = SimpleRNG\_Fp32.GetLaplace(0.0f, 1.0f);

float logNormal32 = SimpleRNG\_Fp32.GetLogNormal(0.0f, 1.0f);

float beta32 = SimpleRNG\_Fp32.GetBeta(2.0f, 2.0f);

Console.WriteLine("SimpleRNG\_Fp32:");

Console.WriteLine($"Uniform: {uniform32}");

Console.WriteLine($"Normal: {normal32}");

Console.WriteLine($"Exponential: {exponential32}");

Console.WriteLine($"Gamma: {gamma32}");

Console.WriteLine($"ChiSquare: {chiSquare32}");

Console.WriteLine($"InverseGamma: {inverseGamma32}");

Console.WriteLine($"Weibull: {weibull32}");

Console.WriteLine($"Cauchy: {cauchy32}");

Console.WriteLine($"StudentT: {studentT32}");

Console.WriteLine($"Laplace: {laplace32}");

Console.WriteLine($"LogNormal: {logNormal32}");

Console.WriteLine($"Beta: {beta32}");

}

}

**数值模块**

**Vorcyc.Mathematics.Numerics 命名空间**

1. BigFloat

一个表示高精度浮点数的结构体。

功能：

• 表示高精度浮点数

• 支持加、减、乘、除等运算

• 支持与其他数值类型的转换

• 支持平方根、幂运算等高级数学运算

使用示例：

BigFloat a = new BigFloat(123.456);

BigFloat b = new BigFloat("789.012");

BigFloat resultAdd = a + b;

BigFloat resultSub = a - b;

BigFloat resultMul = a \* b;

BigFloat resultDiv = a / b;

BigFloat resultSqrt = BigFloat.Sqrt(a);

BigFloat resultPow = BigFloat.Pow(a, 2);

Console.WriteLine($"Add: {resultAdd}");

Console.WriteLine($"Subtract: {resultSub}");

Console.WriteLine($"Multiply: {resultMul}");

Console.WriteLine($"Divide: {resultDiv}");

Console.WriteLine($"Sqrt: {resultSqrt}");

Console.WriteLine($"Pow: {resultPow}");

2. Complex<T>

一个表示复数的结构体，具有泛型数值类型。

功能：

• 表示复数

• 支持加、减、乘、除等运算

• 支持极坐标转换

• 支持求模、求共轭等操作

使用示例：

Complex<double> c1 = new Complex<double>(1.0, 2.0);

Complex<double> c2 = new Complex<double>(3.0, 4.0);

Complex<double> resultAdd = c1 + c2;

Complex<double> resultSub = c1 - c2;

Complex<double> resultMul = c1 \* c2;

Complex<double> resultDiv = c1 / c2;

double magnitude = Complex<double>.Abs(c1);

Complex<double> conjugate = Complex<double>.Conjugate(c1);

Console.WriteLine($"Add: {resultAdd}");

Console.WriteLine($"Subtract: {resultSub}");

Console.WriteLine($"Multiply: {resultMul}");

Console.WriteLine($"Divide: {resultDiv}");

Console.WriteLine($"Magnitude: {magnitude}");

Console.WriteLine($"Conjugate: {conjugate}");

3. ComplexFp32

一个表示复数的结构体，具有单精度浮点数分量。

功能：

• 表示复数

• 支持加、减、乘、除等运算

• 支持极坐标转换

• 支持求模、求共轭等操作

使用示例：

ComplexFp32 c1 = new ComplexFp32(1.0f, 2.0f);

ComplexFp32 c2 = new ComplexFp32(3.0f, 4.0f);

ComplexFp32 resultAdd = c1 + c2;

ComplexFp32 resultSub = c1 - c2;

ComplexFp32 resultMul = c1 \* c2;

ComplexFp32 resultDiv = c1 / c2;

float magnitude = ComplexFp32.Abs(c1);

ComplexFp32 conjugate = ComplexFp32.Conjugate(c1);

Console.WriteLine($"Add: {resultAdd}");

Console.WriteLine($"Subtract: {resultSub}");

Console.WriteLine($"Multiply: {resultMul}");

Console.WriteLine($"Divide: {resultDiv}");

Console.WriteLine($"Magnitude: {magnitude}");

Console.WriteLine($"Conjugate: {conjugate}");

4. Int24

一个表示24位有符号整数的结构体。

功能：

• 表示24位有符号整数

• 支持加、减、乘、除等运算

• 支持与其他数值类型的转换

使用示例：

Int24 a = new Int24(123456);

Int24 b = new Int24(654321);

Int24 resultAdd = a + b;

Int24 resultSub = a - b;

Int24 resultMul = a \* b;

Int24 resultDiv = a / b;

Console.WriteLine($"Add: {resultAdd}");

Console.WriteLine($"Subtract: {resultSub}");

Console.WriteLine($"Multiply: {resultMul}");

Console.WriteLine($"Divide: {resultDiv}");

5. IntRange

一个表示整数范围的结构体。

功能：

• 表示整数范围

• 检查值是否在范围内

• 计算范围的交集

使用示例：

IntRange range = new IntRange(1, 10);

bool isInside = range.IsInside(5);

IntRange intersection = range.Intersection(new IntRange(5, 15));

Console.WriteLine($"IsInside: {isInside}");

Console.WriteLine($"Intersection: [{intersection.Min}, {intersection.Max}]");

6. Number

一个表示数字的类，作为数字列表，具有任意精度。

功能：

• 表示任意精度的数字

• 支持加、减、乘、除等运算

• 支持与其他数值类型的转换

使用示例：

Number n1 = new Number("12345678901234567890");

Number n2 = new Number("98765432109876543210");

Number resultAdd = n1 + n2;

Number resultSub = n1 - n2;

Number resultMul = n1 \* n2;

Number resultDiv = n1 / n2;

Console.WriteLine($"Add: {resultAdd}");

Console.WriteLine($"Subtract: {resultSub}");

Console.WriteLine($"Multiply: {resultMul}");

Console.WriteLine($"Divide: {resultDiv}");

7. Point<T>

一个表示二维平面上点的结构体，具有泛型数值类型。

功能：

• 表示二维平面上的点

• 支持加、减运算

• 支持距离计算

使用示例：

Point<int> p1 = new Point<int>(1, 2);

Point<int> p2 = new Point<int>(3, 4);

Point<int> resultAdd = p1 + p2;

Point<int> resultSub = p1 - p2;

double distance = Point<int>.Distance(p1, p2);

Console.WriteLine($"Add: {resultAdd}");

Console.WriteLine($"Subtract: {resultSub}");

Console.WriteLine($"Distance: {distance}");

8. PointFp32

一个表示二维平面上点的结构体，具有单精度浮点数坐标。

功能：

• 表示二维平面上的点

• 支持加、减运算

• 支持距离计算

使用示例：

PointFp32 p1 = new PointFp32(1.0f, 2.0f);

PointFp32 p2 = new PointFp32(3.0f, 4.0f);

PointFp32 resultAdd = p1 + p2;

PointFp32 resultSub = p1 - p2;

float distance = PointFp32.Distance(p1, p2);

Console.WriteLine($"Add: {resultAdd}");

Console.WriteLine($"Subtract: {resultSub}");

Console.WriteLine($"Distance: {distance}");

9. Range<T>

一个表示值范围的类，具有泛型数值类型。

功能：

• 表示值范围

• 检查值是否在范围内

• 计算范围的交集和并集

使用示例：

Range<int> range = new Range<int>(1, 10);

bool isInside = range.IsInside(5);

Range<int> intersection = range.Intersection(new Range<int>(5, 15));

Range<int> union = range.Union(new Range<int>(5, 15));

Console.WriteLine($"IsInside: {isInside}");

Console.WriteLine($"Intersection: [{intersection.Minimum}, {intersection.Maximum}]");

Console.WriteLine($"Union: [{union.Minimum}, {union.Maximum}]");

10. Rational<T>

一个表示有理数的结构体，具有泛型整数类型。

功能：

• 表示有理数

• 支持加、减、乘、除等运算

• 支持与其他数值类型的转换

使用示例：

Rational<int> r1 = new Rational<int>(1, 2);

Rational<int> r2 = new Rational<int>(3, 4);

Rational<int> resultAdd = r1 + r2;

Rational<int> resultSub = r1 - r2;

Rational<int> resultMul = r1 \* r2;

Rational<int> resultDiv = r1 / r2;

Console.WriteLine($"Add: {resultAdd}");

Console.WriteLine($"Subtract: {resultSub}");

Console.WriteLine($"Multiply: {resultMul}");

Console.WriteLine($"Divide: {resultDiv}");

11. RectangleFP32

一个表示具有浮点数坐标和尺寸的矩形结构体。

功能：

• 表示矩形

• 检查点是否在矩形内

• 计算矩形的交集和并集

使用示例：

RectangleFP32 rect = new RectangleFP32(0.0f, 0.0f, 10.0f, 20.0f);

bool contains = rect.Contains(5.0f, 5.0f);

RectangleFP32 intersection = RectangleFP32.Intersect(rect, new RectangleFP32(5.0f, 5.0f, 10.0f, 20.0f));

RectangleFP32 union = RectangleFP32.Union(rect, new RectangleFP32(5.0f, 5.0f, 10.0f, 20.0f));

Console.WriteLine($"Contains: {contains}");

Console.WriteLine($"Intersection: {intersection}");

Console.WriteLine($"Union: {union}");

12. Size<T>

一个表示二维尺寸的结构体，具有泛型数值类型。

功能：

• 表示二维尺寸

• 支持加、减、乘、除等运算

使用示例：

Size<int> size1 = new Size<int>(10, 20);

Size<int> size2 = new Size<int>(5, 5);

Size<int> resultAdd = size1 + size2;

Size<int> resultSub = size1 - size2;

Size<int> resultMul = size1 \* 2;

Size<int> resultDiv = size1 / 2;

Console.WriteLine($"Add: {resultAdd}");

Console.WriteLine($"Subtract: {resultSub}");

Console.WriteLine($"Multiply: {resultMul}");

Console.WriteLine($"Divide: {resultDiv}");

13. SizeFp32

一个表示二维尺寸的结构体，具有单精度浮点数尺寸。

功能：

• 表示二维尺寸

• 支持加、减、乘、除等运算

使用示例：

SizeFp32 size1 = new SizeFp32(10.0f, 20.0f);

SizeFp32 size2 = new SizeFp32(5.0f, 5.0f);

SizeFp32 resultAdd = size1 + size2;

SizeFp32 resultSub = size1 - size2;

SizeFp32 resultMul = size1 \* 2.0f;

SizeFp32 resultDiv = size1 / 2.0f;

Console.WriteLine($"Add: {resultAdd}");

Console.WriteLine($"Subtract: {resultSub}");

Console.WriteLine($"Multiply: {resultMul}");

Console.WriteLine($"Divide: {resultDiv}");

14. UInt24

一个表示24位无符号整数的结构体。

功能：

• 表示24位无符号整数

• 支持加、减、乘、除等运算

• 支持与其他数值类型的转换

使用示例：

UInt24 a = new UInt24(123456);

UInt24 b = new UInt24(654321);

UInt24 resultAdd = a + b;

UInt24 resultSub = a - b;

UInt24 resultMul = a \* b;

UInt24 resultDiv = a / b;

Console.WriteLine($"Add: {resultAdd}");

Console.WriteLine($"Subtract: {resultSub}");

Console.WriteLine($"Multiply: {resultMul}");

Console.WriteLine($"Divide: {resultDiv}");

**统计模块**

Vorcyc.Mathematics.Statistics类。

Vorcyc.Mathematics.Statistics类是一个全面的工具类，用于对数值数据进行统计分析，支持多种数据结构和数值类型。它包括查找极值、计算总和、平均值、方差，以及识别最大值和最小值及其索引的方法。该类在可能的情况下利用硬件加速以优化性能。

**方法**

FindExtremeValue

1. **重载**：
   * (float[] array, int start, int length)
   * (T[] array, int start, int length) where T : INumber<T>
   * (float[] array)
   * (T[] array) where T : INumber<T>
   * (ArraySegment<float> arraySegment)
   * (ArraySegment<T> arraySegment) where T : INumber<T>
   * (Span<T> span) where T : unmanaged, INumber<T>
2. **描述**：在数组、数组段和跨度中查找最大值和最小值。
3. **示例**：

float[] data = { 1.2f, 3.4f, 5.6f, 7.8f };

var (max, min) = data.FindExtremeValue();

Console.WriteLine($"Max: {max}, Min: {min}");

Sum

1. **重载**：
   * (ArraySegment<float> arraySegment)
   * (ArraySegment<double> arraySegment)
   * (Span<float> values)
   * (float[] array, int start, int length)
   * (float[] values)
   * (Span<double> values)
   * (double[] values)
   * (double[] values, int start, int length)
   * (Span<T> span) where T : INumber<T>
2. **描述**：计算数组、数组段和跨度中元素的总和。
3. **示例**：

float[] data = { 1.2f, 3.4f, 5.6f, 7.8f };

float sum = data.Sum();

Console.WriteLine($"Sum: {sum}");

Average

1. **重载**：
   * (float[] array)
   * (float[] array, int start, int length)
   * (byte[] array)
   * (Span<float> values)
   * (Span<double> values)
   * (ArraySegment<float> arraySegment)
   * (IEnumerable<ArraySegment<float>> arraySegments)
   * (Span<T> span) where T : INumber<T>
2. **描述**：计算数组、数组段和跨度中元素的平均值。
3. **示例**：

float[] data = { 1.2f, 3.4f, 5.6f, 7.8f };

float average = data.Average();

Console.WriteLine($"Average: {average}");

Variance

1. **重载**：
   * (float[] array)
   * (float[] array, int start, int length)
   * (Span<float> values)
   * (Span<double> values)
   * (Span<T> array) where T : INumber<T>
2. **描述**：计算数组和跨度中元素的方差。
3. **示例**：

float[] data = { 1.2f, 3.4f, 5.6f, 7.8f };

var (average, variance) = data.Variance();

Console.WriteLine($"Average: {average}, Variance: {variance}");

GetMaximumMinimumMedian

1. **重载**：
   * (float[] array)
2. **描述**：查找浮点数组中的最大值、最小值和中位数。
3. **示例**：

float[] data = { 1.2f, 3.4f, 5.6f, 7.8f };

var (max, min, median) = data.GetMaximumMinimumMedian();

Console.WriteLine($"Max: {max}, Min: {min}, Median: {median}");

Max

1. **重载**：
   * (Span<T> span) where T : struct, INumber<T>
   * (T[] values) where T : struct, INumber<T>
   * (T[] values, int start, int length) where T : struct, INumber<T>
2. **描述**：查找数组和跨度中元素的最大值。
3. **示例**：

float[] data = { 1.2f, 3.4f, 5.6f, 7.8f };

float max = data.Max();

Console.WriteLine($"Max: {max}");

LocateMax

1. **重载**：
   * (Span<T> span) where T : INumber<T>
   * (T[] values) where T : INumber<T>
   * (T[] values, int start, int length) where T : INumber<T>
2. **描述**：查找数组和跨度中元素的最大值及其索引。
3. **示例**：

float[] data = { 1.2f, 3.4f, 5.6f, 7.8f };

var (index, max) = data.LocateMax();

Console.WriteLine($"Index: {index}, Max: {max}");

Min

1. **重载**：
   * (Span<T> span) where T : struct, INumber<T>
   * (T[] values) where T : struct, INumber<T>
   * (T[] values, int start, int length) where T : struct, INumber<T>
2. **描述**：查找数组和跨度中元素的最小值。
3. **示例**：

float[] data = { 1.2f, 3.4f, 5.6f, 7.8f };

float min = data.Min();

Console.WriteLine($"Min: {min}");

LocateMin

1. **重载**：
   * (Span<T> span) where T : INumber<T>
   * (T[] values) where T : INumber<T>
   * (T[] values, int start, int length) where T : INumber<T>
2. **描述**：查找数组和跨度中元素的最小值及其索引。
3. **示例**：

float[] data = { 1.2f, 3.4f, 5.6f, 7.8f };

var (index, min) = data.LocateMin();

Console.WriteLine($"Index: {index}, Min: {min}");