Math and Statistics of Circular Real Numbers with C++11  
Lior Kogan, 2011-2013

## Introduction

Many scientific and engineering problems involve circular real numbers. Such numbers usually represent angular measurements (e.g. azimuth) or cyclical timing (e.g. time-of-day), but it may represent any other circular quantity. Circual reals are very common in physics, geodesics and navigation, but also appear in other fields such as psychology and criminology (time-of-day statistics), bird-watching and biology (directional statistics and time-of-year statistics).

In contrast to reals (numbers on the real line), circular reals (numbers on a real circle) have a limited range, and a wrap-around property.

Mathematics and Statistics of circular reals is tricky and error prone - both for simple operations such as addition and subtraction, and for more complex operations such as calculating average and median, estimations and interpolations.

For angles, it is even more error prone, since sometimes the range is used, sometimes. Sometimes it is and sometimes - even in the same application. For time-of-day computations, the range may be, or something else. Therefore, we need a scheme for using several representations in the same application, and to safely perform operations between them.

## Features

In this article we are going to present:

* Definitions of circular real type, circular real number, and operators on circular reals
* A C++11 infrastructure for storing and computing with circular reals
* C++11 statistical distribution classes for circular reals: Wrapped Normal and Wrapped Truncated Normal
* Average, Weighted average and Median of circular reals
* Parameter estimation based on noisy independent circular real measurements
* Interpolation and average estimation of sampled continuous circular signal

This is an original work. To the best of our knowledge, some of the ideas mentioned here are novel.

## Requirements

* To compile the code, you’ll need Visual C++ 2012
* The given code relies heavily of C++11 features
* The code may be easily converted to any other C++11 compiler
* The code may be converted to plain old C++. However, it will lose some of its charm.

## A Helper Function: Floating-point modulo

Floating-point modulo function:  
The *dividend*, *divisor* and the resulting *reminder* are all real numbers.  
  
The *quotient* and the *remainder* must satisfy:

* + *quotient* is integer

Still, many definitions are possible:

1. The *remainder* has the same sign as the *dividend*- Implies that the *quotient* rounds towards zero
2. The *remainder* has the same sign as the *divisor*- Implies that the *quotient* rounds towards negative infinity
3. The *remainder* has a different sign than the *dividend*- Implies that the *quotient rounds away* away from zero
4. The *remainder* has a different sign than the *divisor*  
   - Implies that the quotient rounds towards positive infinity
5. The sign of the *remainder* is always non-negative
6. The sign of the *remainder* is always non-positive
7. The *remainder* is closest to zero  
   - Implies that the *quotient* is the integer nearest to the division’s resultIEEE 754 names this *REM* operator, and disambiguate it for a boundary case:  
   When the fractional part of the division’s result is exactly 0.5, the *quotient* is even.
8. The *remainder* is farthest from zero

Values of according to these different implementations:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F | G | H |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F | G | H |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

In addition, we should define the behavior when the *divisor* is 0:

1. When the *divisor* is 0, the *remainder* is undefined
2. When the *divisor* is 0, the *remainder* is defined as 0
3. When the *divisor* is 0, the *remainder* is defined as the *dividend*

Most programming languages, numerical computing environments, and spreadsheets, use definitions A, B or G. For example:

* C (ISO/IEC 9899) and C++ (ISO/IEC 14882) *fmod()* function implements the AX or AY definition (the standard allows either)
* C (starting from ISO/IEC 9899-1999) and C++ (starting from ISO/IEC 14882-2011, but not yet supported by many compilers) defines an additional function - *remainder()*, which implements definition GX or GY (the standard allows either)
* Microsoft Excel’s *MOD()* function implements the BX definition.
* MATLAB’s *mod()* and *rem()* functions implement the BZ and AX definitions respectively.

We’ll use the following definition:

*Definition 1:*

This definition was given by Knuth in *The Art of Computer Programming*, Vol.1 (p.39 of the 3rd edition), and is equivalent to definition BZ.

A straightforward implementation would be:

|  |
| --- |
| template<typename T>  T Mod(T x, T y)  {  static\_assert(!std::numeric\_limits<T>::is\_exact , "Mod: floating-point type expected");  if (0 == y)  return x;  return x - y \* floor(x/y);  } |

However, such implementation is not resilient to boundary cases resulting from the noncontinuity of the modulo function and the rounding behavior of floating-point representation. Here are two examples for double-precision floating-point numbers:

* + Mod(-1e-16, 360.) = 360. (should be 0)  
    The number 360 - 1e-16 = 359.9999999999999999 cannot be represented by double-precision
  + Mod(106.81415022205296 , \_TWO\_PI)= -1.421e-14 (should be \_TWO\_PI - 1.421e-14)

Therefore, we’ll use the following implementation:

|  |
| --- |
| // Floating-point modulo  // The result (the remainder) has the same sign as the divisor.  // Similar to matlab's mod(); Not similar to fmod() - Mod(-3,4)= 1 fmod(-3,4)= -3  template<typename T>  T Mod(T x, T y)  {  static\_assert(!std::numeric\_limits<T>::is\_exact , "Mod: floating-point type expected");  if (0 == y)  return x;  double m= x - y \* floor(x/y);  // handle boundary cases resulting from floating-point limited accuracy:  if (y > 0) // modulo range: [0..y)  {  if (m>=y) // Mod(-1e-16 , 360. ): m= 360.  return 0;  if (m<0 )  {  if (y+m == y)  return 0 ; // just in case...  else  return y+m; // Mod(106.81415022205296 , \_TWO\_PI ): m= -1.421e-14  }  }  else // modulo range: (y..0]  {  if (m<=y) // Mod(1e-16 , -360. ): m= -360.  return 0;  if (m>0 )  {  if (y+m == y)  return 0 ; // just in case...  else  return y+m; // Mod(-106.81415022205296, -\_TWO\_PI): m= 1.421e-14  }  }  return m;  } |

*Theorem 1:*

*Proof:*

## A Helper Function: Floating-point Equality Check

Almost-equal function: IsAlmostEqual(r1, r2)  
When implementing unit-testing for floating-point computations, it is a common need to verify that two computations have equal results. Due to the finite-accuracy of floating-point types, many times the results won’t be equal, but ‘almost equal’. For example double d= 300.; is only almost-equal to double e= exp(log(300.));

To test for almost-equality, we need to consider not the absolute magnitude of the error, but its magnitude relative to the result. Moreover, we need to take care of marginal situations such as INF and NAN.

*Google Test* is a framework for writing C++ tests. I’ve extracted the relevant functionality for testing almost equality into a single file - *FPCompare.h*, which I tweaked a little. Based on it, I implemented the following functions:

|  |
| --- |
| // check if two floating-points are almost equal  template<typename T>  static bool IsAlmostEq(T x, T y)  {  static\_assert(!std::numeric\_limits<T>::is\_exact , "IsAlmostEq: floating-point type expected");  FloatingPoint<T> f(x);  FloatingPoint<T> g(y);  return f.AlmostEquals(g);  }  // assert that 2 floating-points are almost equal  static void AssertAlmostEq(const double f, const double g)  {  assert(IsAlmostEq(f, g));  } |

For more information, look at Google test at <http://code.google.com/p/googletest/>

## Circular real types

*Definition 2:*

A Circular real type is defined by three constants:

* The range
* The zero-value.

Note the use of a right-open interval range.

For example, for angles in the range, it is natural to define the zero-value in the middle, while for angles in the range it is natural to define zero-value on the edge. Generally, the zero-value may be any value in the range – not necessarily 0.

The zero-value has some special properties which we will define below.



We will use the following macro to define a circular real type:

|  |
| --- |
| // macro for defining a circular real type  #define CircValTypeDef(\_Name, \_L, \_H, \_Z) \  struct \_Name \  { \  static const double L ; /\* range: [L,H) \*/ \  static const double H ; \  static const double Z ; /\* zero-value \*/ \  static const double R ; /\* range \*/ \  static const double R\_2; /\* half range \*/ \  \  static\_assert((\_H>\_L) && (\_Z>=\_L) && (\_Z<\_H), \  #\_Name##": Range not valid"); \  }; \  \  const double \_Name::L = (\_L) ; \  const double \_Name::H = (\_H) ; \  const double \_Name::Z = (\_Z) ; \  const double \_Name::R = ((\_H)-(\_L)) ; \  const double \_Name::R\_2= ((\_H)-(\_L))/2.; |

And use it to define the following circular real types:

|  |
| --- |
| // basic circular real types  CircValTypeDef(SignedDegRange , -180., 180., 0. )  CircValTypeDef(UnsignedDegRange, 0., 360., 0. )  CircValTypeDef(SignedRadRange , -M\_PI, M\_PI, 0. )  CircValTypeDef(UnsignedRadRange, 0., 2\*M\_PI, 0. ) |

And some additional circular real types for testing purposes:

|  |
| --- |
| // some additional circular real types - for testing  CircValTypeDef(TestRange0 , 3., 10., 5.3)  CircValTypeDef(TestRange1 , -3., 10., -3.0)  CircValTypeDef(TestRange2 , -3., 10., 9.9)  CircValTypeDef(TestRange3 , -13., -3., -5.3) |

## Defining a circular real with a given type

The templated class CircVal is used for storing a single circular real.  
The template parameter is a circular real type defined using the CircValTypeDef macro.

|  |
| --- |
| // circular real  // Type should be defined using the CircValTypeDef macro  template <typename Type>  class CircVal  {  …  } |

Here is an example of defining a circular real variable with a given type:

|  |
| --- |
| CircVal<UnsignedDegRange> c1; |

## Range checking and wrapping

CircVal::IsInRange static function is used for testing that a given real number is within the range of a circular real type.

|  |
| --- |
| static bool IsInRange(double r)  {  return (r>=Type::L && r<Type::H);  } |

The function is used to ‘wrap around’ a real number to the range of a given circular real type.  
For example: for the range, the number 360 would be wrapper to 0, and 370 would be wrapped to 10. -350 would be wrapped to 10 as well.

*Definition 3:*

We can see that

When:

When:

When: (

We will use this for an optimized implementation.

The CircVal::Wrap static function calculates:

|  |
| --- |
| // 'wraps' a real number to [L,H)  static double Wrap(double r)  {  // the next lines are for optimization and improved accuracy only  if (r>=Type::L)  {  if (r< Type::H ) return r ;  else if (r< Type::H+Type::R) return r-Type::R;  }  else  if (r>=Type::L-Type::R) return r+Type::R;  // general case  return Mod(r - Type::L, Type::R) + Type::L;  } |

## A Walk

*Definition 4:*

For reals, we’ll define a walk by:

* A start point
* A directed length

A walk is a movement along the real line, from the start point, with the given directed length.

When the length is positive, we’ll call it ‘an increasing walk’ (toward positive infinity).  
When the length is negative, we’ll call it ‘a decreasing walk’ (toward negative infinity).

The end point is the point reached at the end of the walk.  
We’ll call a walk with a start point and an end point *“A walk from to”.*

There is only one possible walk from to:

For circular reals, we’ll define a walk by:

* A start point
* A directed length

A walk is a movement along a real circle, from the start point, with the given directed length.

When the length is positive, we’ll call it ‘an increasing walk’ (depicted clockwise in the picture above).  
When the length is negative, we’ll call it ‘a decreasing walk’ (depicted counterclockwise in the picture above).

The end point is the point reached at the end of the walk.  
We’ll call a walk with a start point and an end point *“A walk from to”.*Since the walk is wrapped around the circle:

There are infinitely many walks from to:

*Definition 5:*

The *shortest walk* from  to  is the one with the minimal. The walk may be increasing or decreasing.

## Directed distance between two numbers

*Definition 6:*

The *directed distance* from real number to:

,  
  
is the directed length of the walk from to.

For example,

Similarly, the *directed distance* from circular real to:

,

is the directed length of the shortest walk from to.

is a real number in the range.

It can easily be deduced that:

Since

And since

We can write:

which we will use for faster implementation.

The CircVal::Sdist static function calculates:

|  |
| --- |
| // the length of the direct walk from c1 to c2, with the lowest absolute-value length  // return number is in [-R/2, R/2)  static double Sdist(const CircVal& c1, const CircVal& c2)  {  double d= c2.val-c1.val;  if (d < -Type::R\_2) return d + Type::R;  if (d >= Type::R\_2) return d - Type::R;  return d ;  } |

Note: To define a metric space, we can use as our metric (we won’t prove this here)

For circular reals, we’ll define an additional distance function:

*Definition 7:*

- The *increasing distance* from a circular real to, is the length of the shortest increasing-walk (depicted clockwise in the picture below) from to.

It is always in the range.

There is no equivalent distance function for real numbers.

which may also be expressed as:

It can easily be deduced that:

* + Hence, it cannot be used as a metric

The CircVal::Pdist static function calculates*:*

|  |
| --- |
| // the length of the increasing walk from c1 to c2 with the lowest length  // return number is in [0, R)  static double Pdist(const CircVal& c1, const CircVal& c2)  {  return c2.val>=c1.val ? c2.val-c1.val : Type::R-c1.val+c2.val;  } |

*Theorem 2:*

*Proof:*

Rephrasing the definitions:

*Theorem 3:*

(Very intuitive)

*Proof:*

According to the definition above:

And also

, which are equal in all cases.

*Theorem 4:*

For any two circular reals :

*Proof:*

*Theorem 5*

For any two circular reals :

*Proof:*

*Theorem 6:*

For any two circular reals :

*Proof:*

Trivial, based on theorems 4, 5

## Conversion between different types of circular reals

We want to be able to perform operations between circular reals of different types.

For example:

|  |
| --- |
| CircVal<UnsignedDegRange> d1= 10.;  CircVal<SignedDegRange > d2= -10.;  CircVal<SignedRadRange > d3 ;  d3= d1+d2; |

To convert a circular real from circular real type to circular real type, we’ll retain the arc-length between and. For that, we can use either of these formulae:

**I**

**II**

*Theorem 7:*

**I** and **II** are equivalent

*Proof:*

**I**

**II**

According to theorem 2,

Case 1: - **I** =**II**: trivial  
Case 2:

Therefore, in both cases **I** =**II**

The conversion can be accomplished easily be implementing appropriate constructor and assignment operator:

|  |
| --- |
| // construction based on a circular real of another type  // sample use: CircVal<SignedRadRange> c= c2; -or- CircVal<SignedRadRange> c(c2);  template<typename CircVal2>  CircVal(const CircVal2& c2)  {  double val2= c2.Pdist(c2.GetZ(), c2);  val= Wrap(val2\*Type::R/c2.GetR() + Type::Z);  }  // assignment from another type of circular real  template<typename CircVal2>  CircVal& operator= (const CircVal2& c2)  {  double val2= c2.Pdist(c2.GetZ(), c2);  val= Wrap(val2\*Type::R/c2.GetR() + Type::Z);  return \*this;  } |

## Constructing and fetching a circular real

Construction a CircVal object based on a real number is implemented by the appropriate constructor and assignment operator. Construction of a real number based on a CircVal object is implemented with operator double() const overloading.

|  |
| --- |
| // construction based on a real number  // should only be called when the real number is in the range!  // to translate a floating-point such that 0 is mapped to Type::Z, call ToC()  CircVal(double r)  {  val= Wrap(r);  }  // assignment from a real number  // should only be called when the real number is in the range!  // to translate a floating-point such that 0 is mapped to Type::Z, call ToC()  CircVal& operator= (double r)  {  val= Wrap(r);  return \*this;  }  operator double() const  {  return val;  } |

## Operators on circular reals – intuitive description

First, here is an intuitive description for each operator (formal definition will be given later):

* Negation operator is defined for any circular real. The result is a circular real.  
  Negation of a circular real is the symmetric number relative to (the zero-value).  
  
* Opposite operator is defined for any circular real. The result is a circular real.  
  Opposite of a circular real is the circular real ‘pointing’ to the opposite direction.



* Addition and subtraction operators are defined for any two circular reals.

The result is a circular real - the point reached at the end of the walk:

Addition: Starting from, walking with length.

Subtraction: Starting from, walking with length.



* Multiplication operator is defined for any circular real and any real.  
  The result is a circular real - the point reached and the end of the walk:  
  Starting from, walking distance.
* Division operator is defined for any circular real and any non-zero real.  
  The result is a circular real - the point reached and the end of the walk:  
  Starting from, walking distance.
* Trigonometric functions are defined for any circular real, such that the result is a real, in accordance with the common functions usage.
* Inverse trigonometric functions shall be defined for any real (, for), such that the result is a circular real in accordance with the common functions usage.

## Conversion between circular reals and reals

We want a conversion function, such that any circular real in our range can be mapped to a real, and a conversion function, such that any real can be mapped to a circular real in our range.

Note that these functions are different from the circular real construction/fetching methods described above.

Requirements to these conversion functions:

* For any circular real
* For any circular real
* For any two circular reals
* For any two circular reals
* For any circular real , and any non-negative real
* For any circular real , and any real number
* For any trigonometric function ,   
  for any circular real : *,* where   
  converts from the given circular real type to the circular real type
* For any real : ,
* For any real :

We’ll use these conversion functions to simplify the requirements and the definitions of operators on circular reals.

## Requirements for operators on circular reals

We want to define the following operators on circular reals:   
such that the following conditions are satisfied:

* For any circular real
* For any circular real
* For any two circular reals
* For any three circular reals
* For any circular real
* For any circular real
* For any circular real
* For any circular real
* For any circular real , and for any positive real
* For any circular real , and for any positive real
* For any circular real (using circular trigonometric functions):
* For any real (using circular inverse trigonometric functions):
* For any real (using circular inverse trigonometric function):

The set of real numbers in the range with the binary operator is a bounded group ( is the identity element, and is the inverse element of).

## Comparison operators for circular reals

Defining operators and is straightforward. The other 4 comparison operators can be defined in several different ways. Requirements to these comparison operators (For any circular reals) should satisfy:

* , in other words
* , in other words:

All these requirements, however, tell us nothing about the meaning of the operator.

For reals we can define the operator as:

Based on this, we can define the operator for circular reals in different ways:

Hence, for two circular reals, for each comparison operator, we can use:

Though all definitions satisfy all the requirements, they mean different things.  
 The 1st compares the from , the 2nd compares the from, and the 3rd compares the from .

In many practical situations.

For it seems more appropriate to use (i) or (iii) which are equivalent, while for, (i) and (ii) which are equivalent fits better. This is why our selected definition is (i), but it may be different for specific use cases.

|  |
| --- |
| bool operator==(const CircVal& c) const { return val == c.val; }  bool operator!=(const CircVal& c) const { return val != c.val; }  // note that two circular reals can be compared in several different ways.  // check carefully if this is really what you need!  bool operator> (const CircVal& c) const { return val > c.val; }  bool operator>=(const CircVal& c) const { return val >= c.val; }  bool operator< (const CircVal& c) const { return val < c.val; }  bool operator<=(const CircVal& c) const { return val <= c.val; } |

## Operators for circular reals– Formal definition

Now, here are the definitions that satisfies all the requirements:

*Definition 8:*

* For a circular real
* For a circular real
* For two circular reals
* For two circular reals
* For a circular real and a non-negative real
* For a circular real and a positive real
* For each Trigonometric function , for any circular real :

Converts from the given circular real type - to the circular real type

* For each Inverse trigonometric function and any real :

Converts from the circular reals type to the given circular real type.

*Theorem 8:*

The unary operator can be equivalently defined by each of these two equivalent formulae:

The and the binary operators can be equivalently defined by each of these two equivalent formulae:

*Proof:*

Since, where ,  
and since, where

Here is the implementation of these operators:

|  |
| --- |
| template <typename Type>  class CircVal  {  double val; // actual value [L,H)  public:  // ---------------------------------------------  // convert circular real c to real number [L-Z,H-Z). Z is converted to 0  friend double ToR(const CircVal& c) { return c.val - Type::Z; }  // ---------------------------------------------  const CircVal operator+ ( ) const { return val; }  const CircVal operator- ( ) const { return Wrap(Type::Z-Sdist(Type::Z,val)); } // return negative circular real  const CircVal operator~ ( ) const { return Wrap(val+Type::R\_2 ); } // return opposite circular real  const CircVal operator+ (const CircVal& c) const { return Wrap(val+c.val - Type::Z); }  const CircVal operator- (const CircVal& c) const { return Wrap(val-c.val + Type::Z); }  const CircVal operator\* (const double& r) const { return Wrap((val-Type::Z)\*r + Type::Z); }  const CircVal operator/ (const double& r) const { return Wrap((val-Type::Z)/r + Type::Z); }  CircVal& operator+=(const CircVal& c) { val= Wrap(val+c.val - Type::Z); return \*this; }  CircVal& operator-=(const CircVal& c) { val= Wrap(val-c.val + Type::Z); return \*this; }  CircVal& operator\*=(const double& r) { val= Wrap((val-Type::Z)\*r + Type::Z); return \*this; }  CircVal& operator/=(const double& r) { val= Wrap((val-Type::Z)/r + Type::Z); return \*this; }  CircVal& operator =(const CircVal& c) { val= c.val ; return \*this; }  …  }  // ==========================================================================  template <typename Type> static double sin (const CircVal<Type>& c) { return std::sin(ToR(CircVal<SignedRadRange>(c))); }  template <typename Type> static double cos (const CircVal<Type>& c) { return std::cos(ToR(CircVal<SignedRadRange>(c))); }  template <typename Type> static double tan (const CircVal<Type>& c) { return std::tan(ToR(CircVal<SignedRadRange>(c))); }  template <typename Type> static CircVal<Type> asin (double r ) { return CircVal<SignedRadRange>(std::asin (r )); }  template <typename Type> static CircVal<Type> acos (double r ) { return CircVal<SignedRadRange>(std::acos (r )); }  template <typename Type> static CircVal<Type> atan (double r ) { return CircVal<SignedRadRange>(std::atan (r )); }  template <typename Type> static CircVal<Type> atan2(double r1, double r2 ) { return CircVal<SignedRadRange>(std::atan2(r1,r2)); }  template <typename Type> static CircVal<Type> ToC (double r ) { return CircVal<Type>::Wrap(r + Type::Z); } |

Note that for asin, acos, atan and atan2 functions a template parameter should be used. For example:  
CircVal<SignedDegRange> d1= asin<SignedDegRange>(0.5);

d1= asin(0.5) won’t give us the expected result unless our range is SignedRadRange.

## Testing for near-equality of circular reals

Based on the previously describe test for near-equality for two real numbers, we will construct a near-equality test for two circular reals:

|  |
| --- |
| template <typename Type>  class CircValTester  {  // check if 2 circular reals are almost equal  static bool IsCircAlmostEq(const CircVal<Type>& \_f, const CircVal<Type>& \_g)  {  double f= \_f;  double g= \_g;  if (::IsAlmostEq(f, g))  return true;  if (f < g)  return IsAlmostEq(f, g - Type::R);  else  return IsAlmostEq(f, g + Type::R);  }  // assert that 2 circular reals are almost equal  static void AssertCircAlmostEq(const CircVal<Type>& f, const CircVal<Type>& g)  {  assert(IsCircAlmostEq(f, g));  }  …  } |

## Testing correctness of CircVal class implementation

Using test-driven design, we’ll check that our implementation fulfills our requirements.  
The Test() function generates 10,000 random test-cases, and verifies that all the requirements holds.  
We will use C++11 random generation, which is a very useful addition to the language:

|  |
| --- |
| // tester for CircVal class  template <typename Type>  class CircValTester  {  static void Test()  {  CircVal<Type> ZeroVal= Type::Z;  // --------------------------------------------------------  AssertCircAlmostEq(ZeroVal , -ZeroVal);  AssertAlmostEq (sin(ZeroVal) , 0. );  AssertAlmostEq (cos(ZeroVal) , 1. );  AssertAlmostEq (tan(ZeroVal) , 0. );  AssertCircAlmostEq(asin<Type>(0.), ZeroVal );  AssertCircAlmostEq(acos<Type>(1.), ZeroVal );  AssertCircAlmostEq(atan<Type>(0.), ZeroVal );  AssertCircAlmostEq(ToC<Type>(0) , ZeroVal );  AssertAlmostEq (ToR(ZeroVal) , 0. );  // --------------------------------------------------------  std::default\_random\_engine rand\_engine ;  std::uniform\_real\_distribution<double> c\_uni\_dist(Type::L, Type::H);  std::uniform\_real\_distribution<double> r\_uni\_dist(0. , 1000. ); // for multiplication,division by real-value  std::uniform\_real\_distribution<double> t\_uni\_dist(-1. , 1. ); // for inverse-trigonometric functions    std::random\_device rnd\_device;  rand\_engine.seed(rnd\_device()); // reseed engine  for (unsigned i= 10000; i--;)  {  CircVal<Type> c1(c\_uni\_dist(rand\_engine)); // random circular real  CircVal<Type> c2(c\_uni\_dist(rand\_engine)); // random circular real  CircVal<Type> c3(c\_uni\_dist(rand\_engine)); // random circular real  double r (r\_uni\_dist(rand\_engine)); // random real [ 0, 1000) - for testing \*,/ operators  double a1(t\_uni\_dist(rand\_engine)); // random real [ -1, 1) - for testing asin,acos  double a2(t\_uni\_dist(rand\_engine)); // random real [-1000, 1000) - for testing atan  assert (c1 == CircVal<Type>((double)c1) );  AssertCircAlmostEq(+c1 , c1 ); // +c = c  AssertCircAlmostEq(-(-c1) , c1 ); // -(-c) = c  AssertCircAlmostEq(c1 + c2 , c2 + c1 ); // c1+c2 = c2+c1  AssertCircAlmostEq(c1 + (c2 +c3) , (c1 + c2) + c3 ); // c1+(c2+c3) = (c1+c2)+c3  AssertCircAlmostEq(c1 + -c1 , ZeroVal ); // c+(-c) = z  AssertCircAlmostEq(c1 + ZeroVal , c1 ); // c+z = c  AssertCircAlmostEq(c1 - c1 , ZeroVal ); // c-c = z  AssertCircAlmostEq(c1 - ZeroVal , c1 ); // c-z = c  AssertCircAlmostEq(ZeroVal - c1 , -c1 ); // z-c = -c  AssertCircAlmostEq(c1 - c2 , -(c2 - c1) ); // c1-c2 = -(c2-c1)  AssertCircAlmostEq(c1 \* 0. , ZeroVal ); // c\*0 = 0  AssertCircAlmostEq(c1 \* 1. , c1 ); // c\*1 = c  AssertCircAlmostEq(c1 / 1. , c1 ); // c/1 = c  AssertCircAlmostEq((c1 \* (1./(r+1.))) / (1./(r+1.)) , c1 ); // (c\*r)/r = c, 0<r<=1  AssertCircAlmostEq((c1 / ( r+1.) ) \* ( r+1. ) , c1 ); // (c/r)\*r = c, r>=1  // --------------------------------------------------------  AssertCircAlmostEq(~(~c1) , c1 ); // opposite(opposite(c) = c  AssertCircAlmostEq(c1 - (~c1) , ToC<Type>(Type::R/2.) ); // c - ~c = r/2+z  // --------------------------------------------------------  AssertAlmostEq (sin(ToR(CircVal<SignedRadRange>(c1))), sin(c1) ); // member func sin  AssertAlmostEq (cos(ToR(CircVal<SignedRadRange>(c1))), cos(c1) ); // member func cos  AssertAlmostEq (tan(ToR(CircVal<SignedRadRange>(c1))), tan(c1) ); // member func tan  AssertAlmostEq (sin(-c1) , -sin(c1) ); // sin(-c) = -sin(c)  AssertAlmostEq (cos(-c1) , cos(c1) ); // cos(-c) = cos(c)  AssertAlmostEq (tan(-c1) , -tan(c1) ); // tan(-c1) = -tan(c) error may be large  AssertAlmostEq (sin(c1+ToC<Type>(Type::R/4.)) , cos(c1) ); // sin(c+r/4) = cos(c)  AssertAlmostEq (cos(c1+ToC<Type>(Type::R/4.)) , -sin(c1) ); // cos(c+r/4) = -sin(c)  AssertAlmostEq (sin(c1+ToC<Type>(Type::R/2.)) , -sin(c1) ); // sin(c+r/2) = -sin(c)  AssertAlmostEq (cos(c1+ToC<Type>(Type::R/2.)) , -cos(c1) ); // cos(c+r/2) = -cos(c)  AssertAlmostEq (Sqr(sin(c1))+Sqr(cos(c1)) , 1. ); // sin(x)^2+cos(x)^2 = 1  AssertAlmostEq (sin(c1)/cos(c1) , tan(c1) ); // sin(x)/cos(x) = tan(x)  // --------------------------------------------------------  AssertCircAlmostEq(asin<Type>(a1) , CircVal<SignedRadRange>(asin(a1))); // member func asin  AssertCircAlmostEq(acos<Type>(a1) , CircVal<SignedRadRange>(acos(a1))); // member func acos  AssertCircAlmostEq(atan<Type>(a2) , CircVal<SignedRadRange>(atan(a2))); // member func atan  AssertCircAlmostEq(asin<Type>(a1) + asin<Type>(-a1) , ZeroVal ); // asin(r)+asin(-r) = z  AssertCircAlmostEq(acos<Type>(a1) + acos<Type>(-a1) , ToC<Type>(Type::R/2.) ); // acos(r)+acos(-r) = r/2+z  AssertCircAlmostEq(asin<Type>(a1) + acos<Type>( a1) , ToC<Type>(Type::R/4.) ); // asin(r)+acos( r) = r/4+z  AssertCircAlmostEq(atan<Type>(a2) + atan<Type>(-a2) , ZeroVal ); // atan(r)+atan(-r) = z  // --------------------------------------------------------  assert (c1 > c2 == (c2 < c1) ); // c1> c2 <==> c2< c1  assert (c1 >= c2 == (c2 <= c1) ); // c1>=c2 <==> c2<=c1  assert (c1 >= c2 == ( (c1 > c2) || (c1 == c2)) ); // c1>=c2 <==> (c1> c2)|| (c1==c2)  assert (c1 <= c2 == ( (c1 < c2) || (c1 == c2)) ); // c1<=c2 <==> (c1< c2)|| (c1==c2)  assert (c1 > c2 == (!(c1 == c2) && !(c1 < c2)) ); // c1> c2 <==> !(c1==c2)&&!(c1< c2)  assert (c1 == c2 == (!(c1 > c2) && !(c1 < c2)) ); // c1= c2 <==> !(c1> c2)&&!(c1< c2)  assert (c1 < c2 == (!(c1 == c2) && !(c1 > c2)) ); // c1< c2 <==> !(c1==c2)&&!(c1> c2)  assert (!(c1>c2) || !(c2>c3) || (c1>c3) ); // (c1>c2)&&(c2>c3) ==> c1>c3  // --------------------------------------------------------  AssertCircAlmostEq(c1 , ToC<Type>(ToR( c1) ) ); // c1 = ToC(ToR( c1)  AssertCircAlmostEq(-c1 , ToC<Type>(ToR(-c1) ) ); // -c1 = ToC(ToR(-c1)  AssertCircAlmostEq(c1 + c2 , ToC<Type>(ToR(c1)+ToR(c2)) ); // c1+c2 = ToC(ToR(c1)+ToR(c2))  AssertCircAlmostEq(c1 - c2 , ToC<Type>(ToR(c1)-ToR(c2)) ); // c1-c2 = ToC(ToR(c1)-ToR(c2))  AssertCircAlmostEq(c1 \* r , ToC<Type>(ToR(c1)\*r ) ); // c1\*r = ToC(ToR(c1)\*r )  AssertCircAlmostEq(c1 / r , ToC<Type>(ToR(c1)/r ) ); // c1/r = ToC(ToR(c1)/r )  // --------------------------------------------------------  }  }  public:  CircValTester()  {  Test();  }  }; |

The code below executes these tests for 8 circular real types:

|  |
| --- |
| CircValTester<SignedDegRange > testA;  CircValTester<UnsignedDegRange> testB;  CircValTester<SignedRadRange > testC;  CircValTester<UnsignedRadRange> testD;  CircValTester<TestRange0 > test0;  CircValTester<TestRange1 > test1;  CircValTester<TestRange2 > test2;  CircValTester<TestRange3 > test3; |

## Using the CircVal class

This sample code below speaks for itself:

|  |
| --- |
| CircVal<SignedDegRange > d1= 10. ;  CircVal<UnsignedRadRange> d2= 0.2;  CircVal<SignedDegRange > d3= d1+d2;  d1+= 355.;  double d= d1;  d = sin(d1) / cos(d2) + tan(d3); d1= asin<SignedDegRange>(0.5); |

The code is simple, yet powerful.

## C++11 statistical distribution classes for circular reals

One of the additions to the C++ standard is the <random> header. It is now possible to generate pseudo-random values with a given distribution. Many distributions classes are provided: Bernoulli, Binomial, Cauchy, Chi-Square, Exponential, Extreme Value, Fisher F, Gamma, Geometric, Log-Normal, Negative-Binomial, Normal, Poisson, Student T, Uniform, Weibull, and some interval-related distributions.

Circular distribution classes, however, are not implemented. Some Circular distributions are von Mises, Wrapped Normal and Wrapped Cauchy.

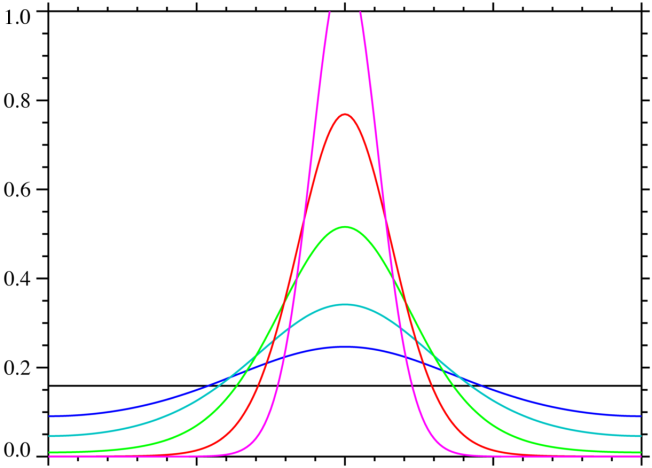
Another distribution, which is very useful when working with circular reals, is the truncated normal distribution.

We have implemented three distribution classes: wrapped normal, truncated normal and wrapped truncated normal.

## Wrapped Normal Distribution class

A wrapped normal distribution results from the "wrapping" of the [normal distribution](http://en.wikipedia.org/wiki/Normal_distribution) around the range.  
Therefore, the probability density function of the wrapped normal distribution is:

, where and  
  
The PDF is described in the following image, for same, but different values of:



This is the distribution of the modulo of a normally-distributed value.

Example: Assume that we have clocks (date + time-of-day), which have a normal-distributed error. It’s time-of-day [00:00, 24:00) distribution would be wrapped-normal.  
Wrapped normal distributed values can be generated as follows:

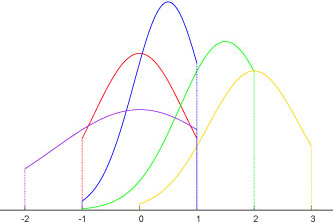
|  |
| --- |
| #include "WrappedNormalDist.h"  std::default\_random\_engine rand\_engine;  std::random\_device rnd\_device ;  rand\_engine.seed(rnd\_device()); // reseed engine  double fAvrg = 0.;  double fSigma= 45.;  double fL = -180.; // wrapping-range lower-bound  double fH = 180.; // wrapping-range upper-bound  wrapped\_normal\_distribution<double> r\_wrp(fAvrg, fSigma, fL, fH);  double r1= r\_wrp(rand\_engine); // random value |

The implementation is based on VC 2012 implementation of the normal distribution, added wrapping.

## Truncated Normal Distribution class

This class has nothing to do with circular reals; however it is a base for the wrapped truncated normal distribution class described in the next section.

A truncated normal distribution is the probability distribution of a [normally distributed](http://en.wikipedia.org/wiki/Normally_distributed) random variable whose value is bounded to the range.  
The PDF is described in the following image, for different values of:



Many phenomena behave, or modeled by a truncated normal distribution. One type of examples is the estimation of quantities by “ordinary” humans, such as today’s temperature: if it is 25° Celsius, no “ordinary” human will estimate it below 10°, or above 40°. Another example is the life span of many species (based on data from Shiro Horiuchi - Interspecies Differences in the Life Span Distribution: Humans versus Invertebrates, The Population Council 2003).

Truncated normal distribution is also useful for generating pseudo-random values such as the described above.

Truncated normal distributed values can be generated as follows:

|  |
| --- |
| #include "TruncNormalDist.h"  std::default\_random\_engine rand\_engine;  std::random\_device rnd\_device ;  rand\_engine.seed(rnd\_device()); // reseed engine  double fAvrg = 0.;  double fSigma= 45.;  double fA = -40.; // truncation-range lower-bound  double fB = 40.; // truncation-range upper-bound  truncated\_normal\_distribution<double> r\_trn(fAvrg, fSigma, fA, fB);  double r2= r\_trn(rand\_engine); // random value |

The implementation is based on VC 2012 implementation of the normal distribution as a skeleton, and on C. H. Jackson's R's implementation of the following paper: “Robert, C. P. - Simulation of truncated normal variables. Statistics and Computing (1995) 5, 121–125”

## Wrapped Truncated Normal Distribution class

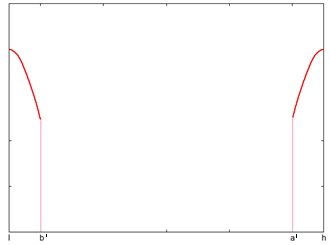
Same as for real numbers, circular reals wrapped normal distribution may be truncated as well.  
Note that the normal distribution is first truncated to the range, and then wrapped around the range

Many phenomena behave, or modeled by a wrapped truncated normal distribution. One type of examples is the estimation of circular quantities by “ordinary” humans, such as the time of sunrise tomorrow, or the azimuth between two drawn points on a piece of paper: If it is 45°, no “ordinary” human will estimate it below 10° or above 80°. Another example may be the exact time-of-day at which John Doe wakes up on an “ordinary” working day.

Wrapped truncated normal distribution is also useful for generating pseudo-random values such as the described above.

Usually, the truncation range is such that, however, this is not a necessity. For example, we may model the time-of-day error distribution of a given clock that was set 1 year ago, and its absolute error now is modeled by a truncated normal distribution of hours.

After wrapping, it is possible that. See picture below, which may depict human estimation of an azimuth between two points, where the real azimuth is 0°.



Wrapped truncated normal distributed values can be generated as follows:

|  |
| --- |
| #include "WrappedTruncNormalDist.h"  std::default\_random\_engine rand\_engine;  std::random\_device rnd\_device ;  rand\_engine.seed(rnd\_device()); // reseed engine  // normal distribution is first truncated, and then wrapped  double fAvrg = 0.;  double fSigma= 100.;  double fA = -500.; // truncation-range lower-bound  double fB = 500.; // truncation-range upper-bound  double fL = 0.; // wrapping -range lower-bound  double fH = 360.; // wrapping -range upper-bound  wrapped\_truncated\_normal\_distribution<double> r\_ wrp\_trn(fAvrg, fSigma, fA, fB, fL, fH);  double d= r\_wrp\_trn(rand\_engine); // random value |

## 3 types of statistical problems

There are 3 distinct types of problems, which tends to be confused one with the other, especially when it comes to circular reals. Although these problems may seem similar, the mathematical methods required to solve each of them should be considered separately.

For each problem type, we’ll give a description, and sample problems for real numbers and for circular reals.

Problem Type 1:

Given [circular] real numbers – calculate their mean  
-or-  
Given a random sample from a [circular] real random-variable - calculate the sample mean

Usage: Averaging numbers

Examples:

* Reals: Given the number of births occurred in the US for each day in the year 2000 -   
  Calculate the mean number of births per day
* Circular reals: Given time-of-day [00:00-24:00) for each birth occurred in US in the year 2000 -   
  Calculate the mean time-of-day

Problem Type 2:

Given a multiset of measurements/observations with a [wrapped-/wrapped-truncated-]normal-distributed error of a [circular] parameter – Estimate the parameter

Usage: Estimation of an unknown constant value, based on a multiset of noisy measurements, where the noise has a [wrapped-/wrapped-truncated-]normal-distribution

Examples:

* Reals: Given a multiset of distance measurements from a stationary transmitter to a stationary receiver, using a measurement technique with a normal distributed error – Estimate the distance.
* Circular reals: Given a multiset of direction measurements from a stationary transmitter to a stationary receiver, using a measurement technique with a wrapped normal distributed error – Estimate the direction.
* Reals: Given a multiset of distance estimates between two points, made by “ordinary” humans (assuming to subject to truncated normal distributed error) - Estimate the distance.
* Circular reals: Given a multiset of azimuth estimates between two points, made by “ordinary” humans (assuming to subject to a wrapped truncated normal distributed error) – Estimate the direction.

Problem Type 3:

Given a multiset of samples of a continuous-time [circular] signal, sampled at times respectively, where the measurement technique/instrument is accurate - Estimate the mean value of the signal at period]

Usage: Estimating the average of a continuous-signal, in a given period, based on a set of time-tagged samples

Examples:

* Reals: Given airplane velocity measured each 1 second [MPH] over a period of one hour – Estimate the mean velocity.
* Circular reals: Given airplane heading measured each 1 second over a period of one hour - Estimate the mean heading.

## Averaging two circular reals

To average two real numbers,, we can start at and take a walk with length. The end-point of the walk is the average. We can, by symmetry, start at and take a walk with length.

For real numbers

For circular reals, given the range [0,360), the average of 330 and 30 is {0} and not {(330+30)/2} = {180}.  
The average is in the middle of the walk with the lowest absolute-value length, from 330 to 30 (or from 30 to 330).

The average of circular reals is a not a single circular real, but a set of circular reals.  
For example, - There is no reason to prefer one of these over the other.

We will use the following formula:  
,  
These two set members are equal, except when .  
In such case, the average is the set  
Of course,

## Averaging *n* circular reals

Calculating average for Circular reals is tricky. Observing algorithms described in many textbooks, and code written by many programmers, I’ve hardly seen a correct method used.

Let us start with reals. Here is our reference problem:

* Given the number of births occurred in the US for each day in the year 2000 -   
  Calculate the mean number of births per day

The solution here is quite simple: Just sum the number of births, and divide it by the number of days.

Why does it work?

The general definition of the mean (arithmetic-average) of numbers for any mathematical field ,  
is the number that minimizes. Formally:.

For real numbers, we can calculate the average as follow:  
Let denote the expression we want to minimize.

To find the value of that minimizes, we need the derivative to be 0:

;  
for 🡪 ;  
We got the well-known arithmetic mean formula for real numbers.

Now let’s consider circular reals. Here is our reference problem:

* Given time-of-day [00:00-24:00) for each birth occurred in US in the year 2000 -   
  Calculate the mean time-of-day

For making the following explanation simple, let us consider the range.

Note that the true average of 0, 0, and 90 should be 30, and not, as suggested in most references.

Based on theorem 3, the absolute value of the distance between two circular reals in the range is

This formula is suitable, since we want to minimize the difference between a value and the average, regardless of which ‘direction’ of the average the value is.  
Let denote the expression we want to minimize.

The values of that minimizes is our average.

Here are some examples:

Example 1: For the input, we’ll plot as a function of:  


We can see that the average is which matches out intuition.

Example 2: For the input, we’ll plot as a function of:  


In this case the average is   
(For all these values, is equal).

Example 3: For the input, we’ll plot as a function of:  


In this case, the average is. Again, it matches our intuition.

The following code was used to collect the data for the above graph:

|  |
| --- |
| vector<CircVal<UnsignedDegRange>> Angles2;  Angles2.push\_back(CircVal<UnsignedDegRange>( 30.));  Angles2.push\_back(CircVal<UnsignedDegRange>(130.));  Angles2.push\_back(CircVal<UnsignedDegRange>(230.));  Angles2.push\_back(CircVal<UnsignedDegRange>(330.));  auto y= CircAverage(Angles2);  ofstream f0("log0.txt");  for (double x= 0.; x<=360.; x+= 0.1)  {  double fSum= 0;  for (const auto& a : Angles2)  fSum+= Sqr(\_\_min(abs(x-a), 360.-abs(x)));  f0 << x << "\t" << fSum << endl;  } |

To find the values of that minimizes, we need the derivative to be 0.  
However, due to the and operators, the expression is not derivable.

Let us try to rephrase the expression.  
We’ll divide the multiset into 3 distinct multi-subsets:  
: multi-subset of , where : multi-subset of , where   
: multi-subset of , where

We’ll denote with – the number of elements in respectively.  
Of course

+ + + + + + + + +   
+  
+  
 + 2= - 2for

The minimum of may be where, or where is not derivable.

Since in each sector is a 2nd order polynomial with a positive coefficient for the term, it has a single minima, which is when. In the sample graphs above, it is easy to see the sectors - each sector is a trimmed 2nd order curve. The points where is not derivable, is on the maximum of each sector.

So, by dividing the input into 3 multisets, we can find the values of that minimizes. This is our average.

However, the division of the input elements into 3 multisets is dependent on the value of.  
Here is a graphic visualization of these multisets, for different values of:

0

x

360

**Multiset B**

0

x

360

**Multiset C**

**Multiset B**

0

x

360

**Multiset D**

**Multiset B**

If, all values are in multiset.  
If is in the range, multiset range is and multiset is empty.  
If is in the range), multiset range is and multiset is empty.  
Multisets and cannot coexist.

Here is the average calculation algorithm:

* Divide the circle into sectors, such that if is within a sector - the elements of multisets *B*, *C* and *D* does not change. The maximal number of sectors equals to the number of elements in *A*.
* For each sector: Assume that (the value of x that minimizes y) is within this sector, calculate its value, and check that the value is indeed within this sector. If so, calculateas follows:  
  1. Assuming x=180, all values are in multiset *B*:
  2. Assuming is in the range, multiset range is and multiset is empty:
  3. Assuming is in the range), multiset range is and multiset is empty:
* Find the sectors(s) that has the lowest ; return their

The Function CircAverage calculates the average set of a given vector of circular reals.   
The return value is a set of all average values:

|  |
| --- |
| // calculate average set of circular reals  // return set of average values  // T is a circular real type defined with the CircValTypeDef macro  template<typename T>  set<CircVal<T>> CircAverage(vector<CircVal<T>> const& A)  {  set<CircVal<T>> MinAvrgVals ; // results set  // ----------------------------------------------  // all vars: UnsignedDegRange [0,360)  double fSum = 0.; // of all elements of A  double fSumSqr = 0.; // of all elements of A  double fMinSumSqrDiff ; // minimal sum of squares of differences  vector<double> LowerAngles ; // ascending [ 0,180)  vector<double> UpperAngles ; // descending (360,180)  double fTestAvrg ;  // ----------------------------------------------  // local functions - implemented as lambdas  // ----------------------------------------------  // calc sum(dist(180, Bi)^2) - all values are in set B  // dist(180,Bi)= |180-Bi|  // sum(dist(x, Bi)^2) = sum((180-Bi)^2) = sum(180^2-2\*180\*Bi + Bi^2) = 180^2\*A.size - 360\*sum(Ai) + sum(Ai^2)  auto SumSqr = [&]() -> double  {  return 32400.\*A.size() - 360.\*fSum + fSumSqr;  };  // calc sum(dist(x, Ai)^2). A=B+C; set D is empty  // dist(x,Bi)= |x-Bi|  // dist(x,Ci)= 360-(Ci-x)  // sum(dist(x, Bi)^2)= sum( (x-Bi) ^2)= sum( Bi^2 + x^2 - 2\*Bi\*x)  // sum(dist(x, Ci)^2)= sum((360-(Ci-x))^2)= sum(360^2 + Ci^2 + x^2 - 2\*360\*Ci + 2\*360\*x - 2\*Ci\*x)  // sum(dist(x, Bi)^2) + sum(dist(x, Ci)^2) = nCountC\*360^2 + sum(Ai^2) + nCountA\*x^2 - 2\*360\*sum(Ci) + nCountC\*2\*360\*x - 2\*x\*sum(Ai)  auto SumSqrC= [&](double x, size\_t nCountC, double fSumC) -> double  {  return x\*(A.size()\*x - 2\*fSum) + fSumSqr - 2\*360.\*fSumC + nCountC\*( 2\*360.\*x + 360.\*360.);  };  // calc sum(dist(x, Ai)^2). A=B+D; set C is empty  // dist(x,Bi)= |x-Bi|  // dist(x,Di)= 360-(x-Di)  // sum(dist(x,Bi)^2)= sum( (x-Bi)^2)= sum( Bi^2 + x^2 - 2\*Bi\*x)  // sum(dist(x,Di)^2)= sum(360-(x-Di)^2)= sum(360^2 + Di^2 + x^2 + 2\*360\*Di - 2\*360\*x - 2\*Di\*x)  // sum(dist(x, Bi)^2) + sum(dist(x, Di)^2) = nCountD\*360^2 + sum(Ai^2) + nCountA\*x^2 + 2\*360\*sum(Di) - nCountD\*2\*360\*x - 2\*x\*sum(Ai)  auto SumSqrD= [&](double x, size\_t nCountD, double fSumD) -> double  {  return x\*(A.size()\*x - 2\*fSum) + fSumSqr + 2\*360.\*fSumD + nCountD\*(-2\*360.\*x + 360.\*360.);  };  // update MinAvrgAngles if lower/equal fMinSumSqrDiff found  auto TestSum= [&](double fTestAvrg, double fTestSumDiffSqr) -> void  {  if (fTestSumDiffSqr < fMinSumSqrDiff)  {  MinAvrgVals.clear();  MinAvrgVals.insert(CircVal<UnsignedDegRange>(fTestAvrg));  fMinSumSqrDiff= fTestSumDiffSqr;  }  else if (fTestSumDiffSqr == fMinSumSqrDiff)  MinAvrgVals.insert(CircVal<UnsignedDegRange>(fTestAvrg));  };  // ----------------------------------------------  for (const auto& a : A)  {  double v= CircVal<UnsignedDegRange>(a); // convert to [0.360)  fSum += v ;  fSumSqr+= Sqr(v);  if (v < 180.) LowerAngles.push\_back(v);  else if (v > 180.) UpperAngles.push\_back(v);  }  sort(LowerAngles.begin(), LowerAngles.end() ); // ascending [ 0,180)  sort(UpperAngles.begin(), UpperAngles.end(), greater<double>()); // descending (360,180)  // ----------------------------------------------  // start with avrg= 180, sets c,d are empty  // ----------------------------------------------  MinAvrgVals.clear();  MinAvrgVals.insert(CircVal<UnsignedDegRange>(180.));  fMinSumSqrDiff= SumSqr();    // ----------------------------------------------  // average in (180,360), set D: values in range [0,avrg-180)  // ----------------------------------------------  double fLowerBound= 0.; // of current sector  double fSumD = 0.; // of elements of set D  auto iter= LowerAngles.begin();  for (size\_t d= 0; d < LowerAngles.size(); ++d)  {  // 1st iteration : average in ( 180, lowerAngles[0]+180]  // next iterations: average in (lowerAngles[i-1]+180, lowerAngles[i]+180]  // set D : lowerAngles[0..d]  fTestAvrg= (fSum + 360.\*d)/A.size(); // average for sector, that minimizes SumDiffSqr  if ((fTestAvrg > fLowerBound+180.) && (fTestAvrg <= \*iter+180.)) // if fTestAvrg is within sector  TestSum(fTestAvrg, SumSqrD(fTestAvrg, d, fSumD)); // if fTestAvrg generates lower SumSqr  fLowerBound= \*iter ;  fSumD += fLowerBound;  ++iter;  }  // last sector : average in [lowerAngles[lastIdx]+180, 360)  fTestAvrg= (fSum + 360.\*LowerAngles.size())/A.size(); // average for sector, that minimizes SumDiffSqr  if ((fTestAvrg < 360.) && (fTestAvrg > fLowerBound)) // if fTestAvrg is within sector  TestSum(fTestAvrg, SumSqrD(fTestAvrg, LowerAngles.size(), fSumD)); // if fTestAvrg generates lower SumSqr    // ----------------------------------------------  // average in [0,180); set C: values in range (avrg+180, 360)  // ----------------------------------------------  double fUpperBound= 360.; // of current sector  double fSumC = 0.; // of elements of set C  iter= UpperAngles.begin();  for (size\_t c= 0; c < UpperAngles.size(); ++c)  {  // 1st iteration : average in [upperAngles[0]-180, 360 )  // next iterations: average in [upperAngles[i]-180, upperAngles[i-1]-180)  // set C : upperAngles[0..c] (descendingly sorted)  fTestAvrg= (fSum - 360.\*c)/A.size(); // average for sector, that minimizes SumDiffSqr  if ((fTestAvrg >= \*iter-180.) && (fTestAvrg < fUpperBound-180.)) // if fTestAvrg is within sector  TestSum(fTestAvrg, SumSqrC(fTestAvrg, c, fSumC)); // if fTestAvrg generates lower SumSqr    fUpperBound= \*iter ;  fSumC += fUpperBound;  ++iter;  }  // last sector : average in [0, upperAngles[lastIdx]-180)  fTestAvrg= (fSum - 360.\*UpperAngles.size())/A.size(); // average for sector, that minimizes SumDiffSqr  if ((fTestAvrg >= 0.) && (fTestAvrg < fUpperBound)) // if fTestAvrg is within sector  TestSum(fTestAvrg, SumSqrC(fTestAvrg, UpperAngles.size(), fSumC)); // if fTestAvrg generates lower SumSqr  // ----------------------------------------------  return MinAvrgVals;  } |

Note that the complexity of this function is the complexity of the sort operation, hence. Given a sorted input, the complexity can be.

Here is a sample use:

|  |
| --- |
| vector<CircVal<UnsignedDegRange>> angles;  angles.push\_back( 90.);  angles.push\_back(180.);  angles.push\_back(270.);  auto avg= CircAverage(angles); |

## Weighted Average of circular reals

The general definition of the weighted mean (arithmetic-average) of numbers with weights respectively, for any mathematical field is the value that minimizes the expression. Formally:.  
 Is a positive real weight associated with value, and is a function that measures the minimal distance between and.

For real numbers, we can calculate the average as follow:   
Let denote the expression we want to minimize:

To find the value of that minimizes, we need the derivative to be 0:

;  
for 🡪 ;  
We got the well-known weighted arithmetic mean formula for real numbers.

For circular reals, we showed that

Let denote the expression we want to minimize:

Same as for non-weighted average, we will break the input into 3 distinct multi-subsets: *B*, *C* and *D*.

+ + + + + + + + +   
+  
+  
 + 2= - 2for

And the algorithm for weighted average is:

* Divide the circle into sectors, such that if is within a sector - the elements of multisets *B*, *C* and *D* does not change. The maximal number of sectors equals to the number of elements we want to average.
* For each sector: Assume that (the value of *x* that minimizes *y*) is within this sector, calculate its value, and check that the value is indeed within this sector. If so, calculateas follows:  
  1. Assuming x=180, all values are in multiset *B*:
  2. Assuming is in the range, multiset range is and multiset is empty:
  3. Assuming is in the range), multiset range is and multiset is empty:
* Find the sectors(s) that has the lowest ; return their

The Function WeightedCircAverage calculates the weighted average set of a given vector of circular reals.   
The return value is a set of all average values:

|  |
| --- |
| // calculate weighted-average set of circular reals  // return set of average values  // T is a circular real type defined with the CircValTypeDef macro  template<typename T>  set<CircVal<T>> WeightedCircAverage(vector<pair<CircVal<T>,double>> const& A) // vector <value,weight>  {  set<CircVal<T>> MinAvrgVals ; // results set  // ----------------------------------------------  // all vars: UnsignedDegRange [0,360)  double fASumW = 0.; // sum(Wi ) of all elements of A  double fASumWA = 0.; // sum(Wi\*Ai ) of all elements of A  double fASumWA2 = 0.; // sum(Wi\*Ai^2) of all elements of A  double fMinSumSqrDiff ; // minimal sum of squares of differences  vector<pair<double, double>> LowerAngles ; // ascending [ 0,180) <angle,weight>  vector<pair<double, double>> UpperAngles ; // descending (360,180) <angle,weight>  double fTestAvrg ;  // ----------------------------------------------  // local functions - implemented as lambdas  // ----------------------------------------------  // calc sum(Wi\*dist(180, Bi)^2) - all values are in set B  // dist(180,Bi)= |180-Bi|  // sum(Wi\*dist(x, Bi)^2) = sum(Wi\*(180-Bi)^2) = sum(Wi\*(180^2-2\*180\*Bi + Bi^2)) = 180^2\*fSumW - 360\*sum(Wi\*Ai) + sum(Wi\*Ai^2)  auto SumSqr = [&]() -> double  {  return 32400.\*fASumW - 360.\*fASumWA + fASumWA2;  };  // calc sum(Wi\*dist(x, Ai)^2). A=B+C; set D is empty  // dist(x,Bi)= |x-Bi|  // dist(x,Ci)= 360-(Ci-x)  // sum(Wi\*dist(x,Bi)^2)= sum(Wi\*( (x-Bi) ^2))= sum(Wi\*( Bi^2 + x^2 - 2\*Bi\*x)) +  // sum(Wi\*dist(x,Ci)^2)= sum(Wi\*((360-(Ci-x))^2))= sum(Wi\*(360^2 + Ci^2 + x^2 - 2\*360\*Ci + 2\*360\*x - 2\*Ci\*x))  // ==========================================================  // sum(Wi\*( Ai^2 + x^2 - 2\*Ai\*x))  auto SumSqrC= [&](double x ,  double fCSumW , // sum(Wi ) of all elements of C  double fCSumWC ) -> double // sum(Wi\*Ci) of all elements of C  {  return fASumWA2 + x\*x\*fASumW -2\*x\*fASumWA - 720\*fCSumWC + (129600+720\*x)\*fCSumW;  };  // calc sum(Wi\*dist(x, Ai)^2). A=B+D; set C is empty  // dist(x,Bi)= |x-Bi|  // dist(x,Di)= 360-(x-Di)  // sum(Wi\*dist(x,Bi)^2)= sum(Wi\*( (x-Bi) ^2))= sum(Wi\*( Bi^2 + x^2 - 2\*Bi\*x))  // sum(Wi\*dist(x,Di)^2)= sum(Wi\*((360-(x-Di))^2))= sum(Wi\*(360^2 + Di^2 + x^2 + 2\*360\*Di - 2\*360\*x - 2\*Di\*x))  // ==========================================================  // sum(Wi\*( Ai^2 + x^2 - 2\*Ai\*x))  auto SumSqrD= [&](double x ,  double fDSumW , // sum(Wi ) of all elements of D  double fDSumWD ) -> double // sum(Wi\*Di) of all elements of D  {  return fASumWA2 + x\*x\*fASumW -2\*x\*fASumWA + 720\*fDSumWD + (129600-720\*x)\*fDSumW;  };  // update MinAvrgAngles if lower/equal fMinSumSqrDiff found  auto TestSum= [&](double fTestAvrg, double fTestSumDiffSqr) -> void  {  if (fTestSumDiffSqr < fMinSumSqrDiff)  {  MinAvrgVals.clear();  MinAvrgVals.insert(CircVal<UnsignedDegRange>(fTestAvrg));  fMinSumSqrDiff= fTestSumDiffSqr;  }  else if (fTestSumDiffSqr == fMinSumSqrDiff)  MinAvrgVals.insert(CircVal<UnsignedDegRange>(fTestAvrg));  };  // ----------------------------------------------  for (const auto& a : A)  {  double v= CircVal<UnsignedDegRange>(a.first); // convert to [0.360)  double w= a.second; // weight  fASumW += w ;  fASumWA+= w\*v ;  fASumWA2= w\*v\*v;  if (v < 180.) LowerAngles.push\_back(pair<double,double>(v,w));  else if (v > 180.) UpperAngles.push\_back(pair<double,double>(v,w));  }  sort(LowerAngles.begin(), LowerAngles.end() ); // ascending [ 0,180)  sort(UpperAngles.begin(), UpperAngles.end(), greater<pair<double,double>>()); // descending (360,180)  // ----------------------------------------------  // start with avrg= 180, sets c,d are empty  // ----------------------------------------------  MinAvrgVals.clear();  MinAvrgVals.insert(CircVal<UnsignedDegRange>(180.));  fMinSumSqrDiff= SumSqr();    // ----------------------------------------------  // average in (180,360), set D: values in range [0,avrg-180)  // ----------------------------------------------  double fLowerBound= 0.; // of current sector  double fDSumW = 0.; // sum(Wi ) of all elements of D  double fDSumWD = 0.; // sum(Wi\*Di) of all elements of D  auto iter= LowerAngles.begin();  for (size\_t d= 0; d < LowerAngles.size(); ++d)  {  // 1st iteration : average in ( 180, lowerAngles[0]+180]  // next iterations: average in (lowerAngles[i-1]+180, lowerAngles[i]+180]  // set D : lowerAngles[0..d]  fTestAvrg= (fASumWA + 360.\*fDSumW)/fASumW; // average for sector, that minimizes SumDiffSqr  if ((fTestAvrg > fLowerBound+180.) && (fTestAvrg <= (\*iter).first+180.)) // if fTestAvrg is within sector  TestSum(fTestAvrg, SumSqrD(fTestAvrg, fDSumW, fDSumWD)); // check if fTestAvrg generates lower SumSqr  fLowerBound= (\*iter).first ;  fDSumW += (\*iter).second ;  fDSumWD += (\*iter).second \* (\*iter).first;  ++iter;  }  // last sector : average in [lowerAngles[lastIdx]+180, 360)  fTestAvrg= (fASumWA + 360.\*fDSumW)/fASumW; // average for sector, that minimizes SumDiffSqr  if ((fTestAvrg < 360.) && (fTestAvrg > fLowerBound)) // if fTestAvrg is within sector  TestSum(fTestAvrg, SumSqrD(fTestAvrg, fDSumW, fDSumWD)); // check if fTestAvrg generates lower SumSqr    // ----------------------------------------------  // average in [0,180); set C: values in range (avrg+180, 360)  // ----------------------------------------------  double fUpperBound= 360.; // of current sector  double fCSumW = 0.; // sum(Wi ) of all elements of C  double fCSumWC = 0.; // sum(Wi\*Ci) of all elements of C  iter= UpperAngles.begin();  for (size\_t c= 0; c < UpperAngles.size(); ++c)  {  // 1st iteration : average in [upperAngles[0]-180, 360 )  // next iterations: average in [upperAngles[i]-180, upperAngles[i-1]-180)  // set C : upperAngles[0..c] (descendingly sorted)  fTestAvrg= (fASumWA - 360.\*fCSumW)/fASumW; // average for sector, that minimizes SumDiffSqr  if ((fTestAvrg >= (\*iter).first-180.) && (fTestAvrg < fUpperBound-180.)) // if fTestAvrg is within sector  TestSum(fTestAvrg, SumSqrC(fTestAvrg, fCSumW, fCSumWC)); // check if fTestAvrg generates lower SumSqr    fUpperBound= (\*iter).first ;  fCSumW += (\*iter).second ;  fCSumWC += (\*iter).second \* (\*iter).first;  ++iter;  }  // last sector : average in [0, upperAngles[lastIdx]-180)  fTestAvrg= (fASumWA - 360.\*fCSumW)/fASumW; // average for sector, that minimizes SumDiffSqr  if ((fTestAvrg >= 0.) && (fTestAvrg < fUpperBound)) // if fTestAvrg is within sector  TestSum(fTestAvrg, SumSqrC(fTestAvrg, fCSumW, fCSumWC)); // check if fTestAvrg generates lower SumSqr  // ----------------------------------------------  return MinAvrgVals;  } |

Note that the complexity of this function is the complexity of the sort operation, hence. Given sorted input, the complexity can be.

Here is a sample use:

|  |
| --- |
| vector<pair<CircVal<UnsignedDegRange>,double>> angles;  angles.push\_back(make\_pair( 90., 0.3));  angles.push\_back(make\_pair(180., 0.5));  angles.push\_back(make\_pair(270., 0.7));  auto avg= WeightedCircAverage(angles); |

## Median of circular reals

*Definition 9:*

Given the multiset  
Let sequence be the non-decreasingly sorted permutation of

For real numbers:

If is odd:   
   
If is even:

For circular reals:

Since there is no “first” nor “last” element, we’ll evaluate all “rotations” of :  
For each :

If is odd:   
   
If is even:   
 (The avrg set may contain more than one element)

The set of all candidate medians:

Since an optimality property of the median is the minimization of the sum of absolute distances:  
 (The median set may contain more than one value)

Here are some examples for circular reals, given the range:

Therefore,

Therefore,

Therefore,

(note that the average of is)

Therefore,

The function CircMedian calculates the median set of a given vector of circular reals.   
The return value is a set of all median values:

|  |
| --- |
| // calculate median set of circular reals  // return set of median values  // T is a circular real type defined with the CircValTypeDef macro  template<typename T>  set<CircVal<T>> CircMedian(vector<CircVal<T>> const& A)  {  set <CircVal<T>> X; // results set  // ----------------------------------------------  set<CircVal<T>> B;  if (A.size() % 2 == 0) // even number of numbers  {  auto S= A;  sort(S.begin(), S.end()); // A, sorted  for (size\_t m= 0; m < S.size(); ++m)  {  size\_t n= m+1; if (n==S.size()) n= 0;  double d= CircVal<T>::Sdist(S[m], S[n]);  // insert average set of each two circular-consecutive numbers  B.insert(((double)S[m] + d/2.));  if (d == -CircVal<T>::GetR()/2.)  B.insert(((double)S[n] + d/2.));  }  }  else // odd number of numbers  for (size\_t m= 0; m < A.size(); ++m)  B.insert(A[m]); // convert vector to set - remove duplicates  // ----------------------------------------------  double fMinSum= numeric\_limits<double>::max();  for (const auto& b : B)  {  double fSum= 0.; // sum(|Sdist(a, b)|)  for (const auto& a : A)  fSum+= abs(CircVal<T>::Sdist(b, a));  if (fSum==fMinSum) X.insert(b);  else if (fSum< fMinSum) { X.clear(); X.insert(b); fMinSum= fSum; }  }  // ----------------------------------------------  return X;  } |

Here is a sample use:

|  |
| --- |
| vector<CircVal<UnsignedDegRange>> angles;  angles.push\_back( 90.);  angles.push\_back(180.);  angles.push\_back(270.);  auto mdn= CircMedian(angles); |

## Circular parameter estimation based on noisy measurements

For reals:

For many observations/measurements techniques/instruments, normal (Gaussian) distribution is often used as a first approximation to model observations/measurements error. As Jonas Ferdinand Gabriel [Lippmann](http://en.wikipedia.org/wiki/Gabriel_Lippmann) said: “Everyone believes in the [normal] law of errors: the mathematicians, because they think it is an experimental fact; and the experimenters, because they suppose it is a theorem of mathematics.”

Hence, for our reference problem:

* Given a multiset of independent distance measurements from a stationary transmitter to a stationary receiver, using a measurement technique with a normal-distributed error – Estimate the distance.

It can be shown that given a multiset of independent measurements of a constant parameter, where the measurement technique/instrument has a normal distributed error, the maximum likelihood estimation (MLE) of the measured parameters equals to the arithmetic average of the measurements.

Hence, for our reference problem, the maximum likelihood estimation of the distance is the arithmetic average of the measurements.

For circular reals:

Here, wrapped normal distribution, or wrapped truncated normal distribution may be used to model a measurement error (according to the physical phenomena we want to model). The maximal measurement error is bounded to half circle.

Given a multiset of measurements of a constant circular parameter, where the measurement technique/instrument has a wrapped/wrapped-truncated normal distributed error.

Here is our reference problem:

* Given a multiset of independent direction measurements from a stationary transmitter to a stationary receiver, using a measurement technique with a wrapped/wrapped-truncated normal-distributed error – Estimate the direction.

We will consider two methods to estimate the circular parameter:

1. The circular average of the samples - as described above
2. Consider each sample as a 2D unit vector, and calculate the weighted vector.   
   - The suggested method in most textbooks

Simulation: As a test case, we will generate 50,000 trails. Each trail will contain 1000 samples of measurements with wrapped/wrapped-truncated normal distributed error (90° truncation span). We will use the two methods to calculate the estimate for each trail, and then calculate the RMS error for each method.

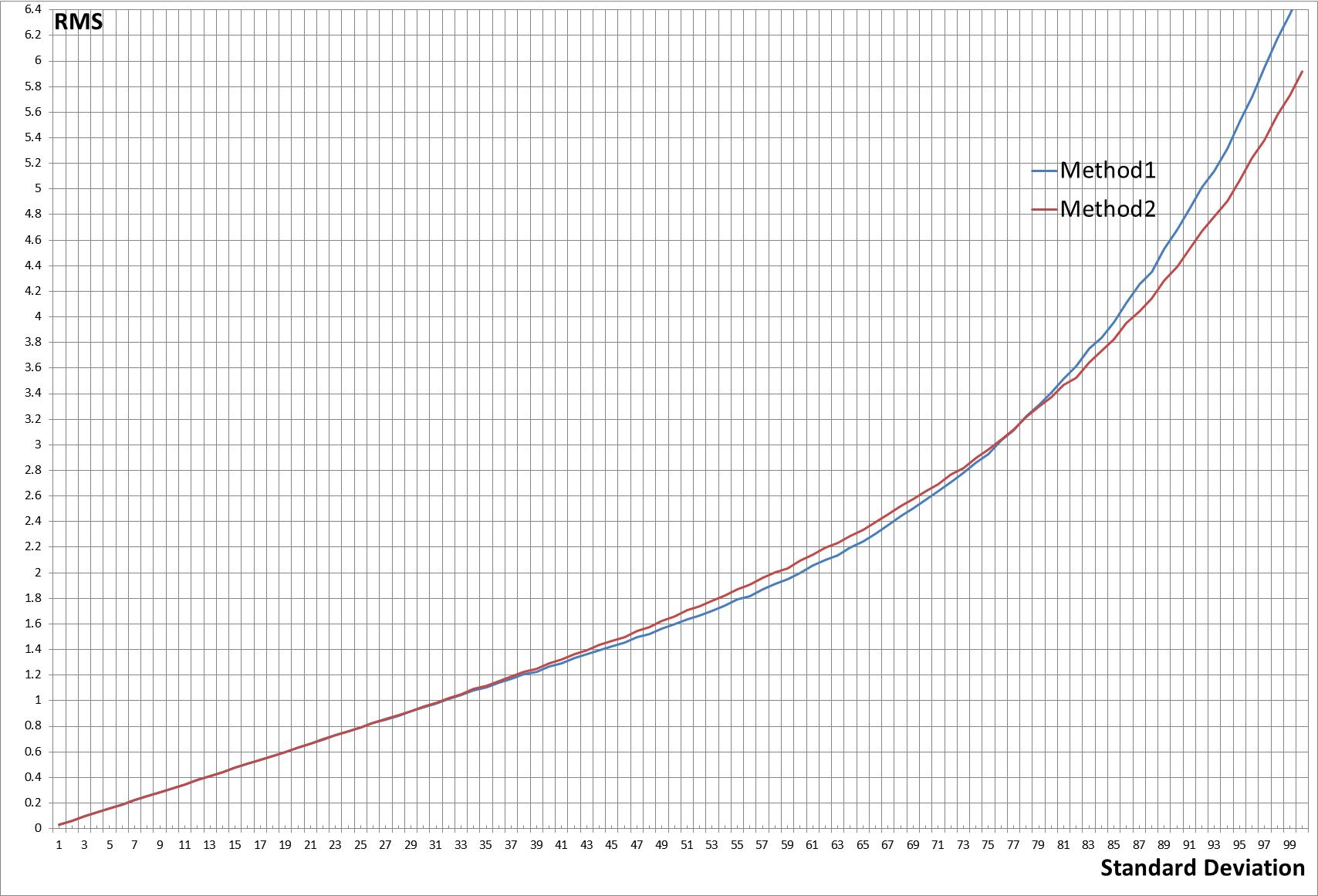
We will repeat the test for different standard deviations – between 1° and 100°.

The following code will collect the data for the two methods:

|  |
| --- |
| std::default\_random\_engine rand\_engine;  std::random\_device rnd\_device ;  rand\_engine.seed(rnd\_device()); // reseed engine  ofstream f1("log1.txt");  Concurrency::parallel\_for(1, 101, [&](int nStdDev) // for each value of standard-deviation  {  uniform\_real<double> ud(0., 360.);  double fSumSqrErr1= 0.;  double fSumSqrErr2= 0.;  const size\_t nTrails = 50000; // number of trails  const size\_t nSamples= 1000; // number of observations per trail    vector<CircVal<UnsignedDegRange>> vInput(nSamples);  const double fAvrg= ud(rand\_engine); // our const parameter for this trail  wrapped\_normal\_distribution <double> r\_wnd1(fAvrg, nStdDev, 0., 360.);  // wrapped\_truncated\_normal\_distribution<double> r\_wnd1(fAvrg, nStdDev, fAvrg-45., fAvrg+45., 0., 360.);  for (size\_t t= 0; t<nTrails; ++t)  {  for (auto& Sample : vInput)  Sample= r\_wnd1(rand\_engine); // generate "noisy" observation  set<CircVal<UnsignedDegRange>> sAvrg1= CircAverage(vInput); // avrg - method 1  double fSigSin= 0.;  double fSigCos= 0.;  for (const auto& Sample : vInput)  {  fSigSin+= sin(Sample);  fSigCos+= cos(Sample);  }  CircVal<UnsignedDegRange> Avrg2= atan2<UnsignedDegRange>(fSigSin, fSigCos); // avrg - method 2  const double fErr1= CircVal<UnsignedDegRange>::Sdist(\*sAvrg1.begin(), fAvrg); // error of estimate - method 1  const double fErr2= CircVal<UnsignedDegRange>::Sdist(Avrg2 , fAvrg); // error of estimate - method 2  fSumSqrErr1+= Sqr(fErr1);  fSumSqrErr2+= Sqr(fErr2);  }  const double fRMS1= sqrt(fSumSqrErr1/ (nTrails-1)); // root mean square error - method 1  const double fRMS2= sqrt(fSumSqrErr2/ (nTrails-1)); // root mean square error - method 2  f1 << nStdDev << "\t" << fRMS1 << "\t" << fRMS2 << endl; // save RMS results to file  } ); |

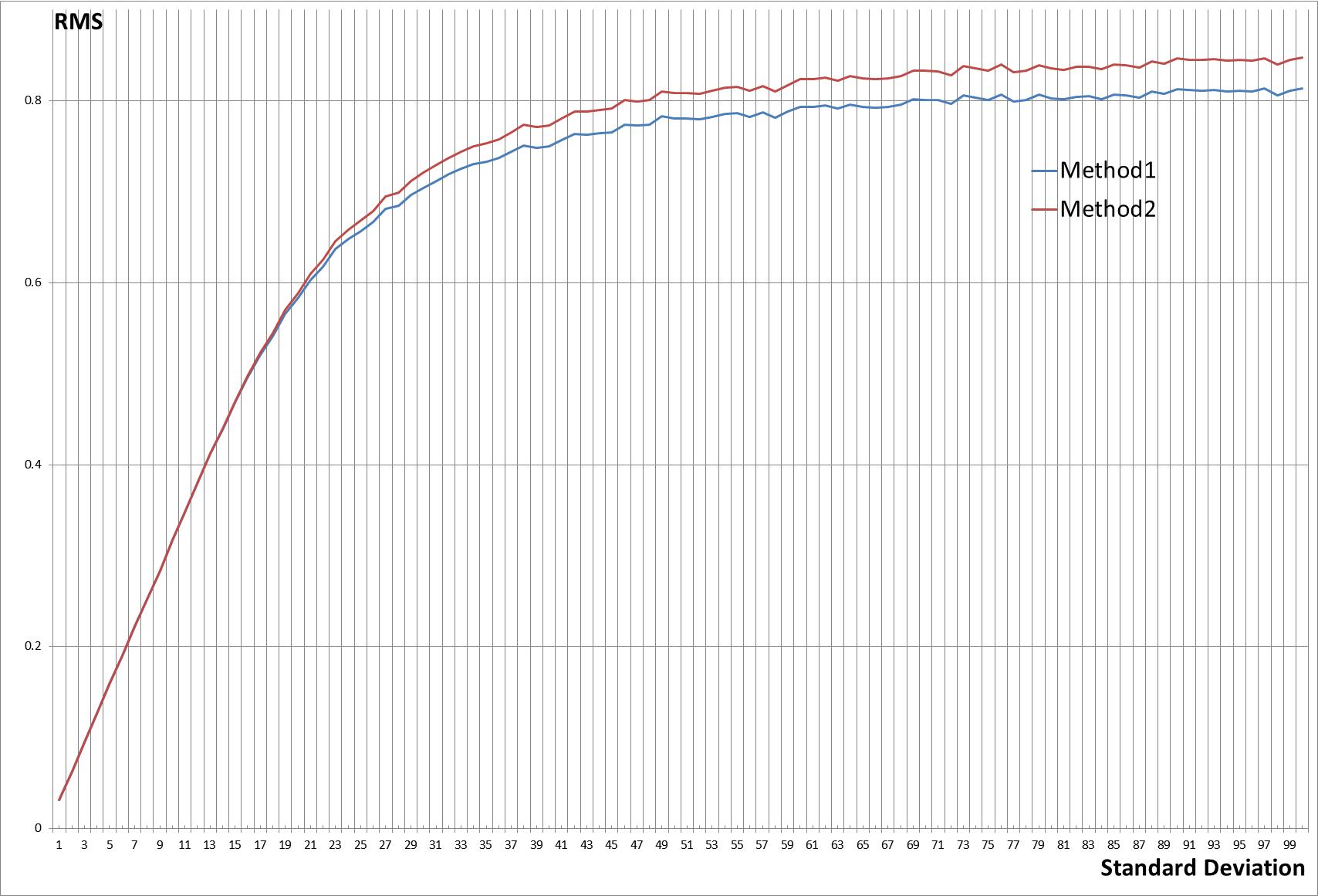
Note that parallel\_for is used for parallelizing tests for different standard deviations.

So which method is better?

****

The graph above shows the RMS error of the average estimation for 50000 trails. Each trail averages 1000 measurements with **wrapped normal** distributed error with standard deviation according to the X axis.

Method 1 turns out to give more accurate estimations when the standard deviation is less than 65°-80° (this value grows when there are more samples per trail: 65° for 10 samples per trail, and 78° for 1000 samples per trail). If the data is noisier - method 2 becomes better. Since the standard deviation in most real scenarios is much less than 65°, method 1 is better for almost any practical purpose.

****

The graph above shows the RMS error of the average estimation for 50000 trails. Each trail averages 1000 measurements with **wrapped truncated normal** distributed error with standard deviation according to the X axis.  
The truncation span is 90° around the parameter value, and the wrapping range is [0,360).

Method 1 proves to be better for wrapped truncated normal standard deviation error, for any standard deviation.

## Interpolation and average estimation of sampled continuous-time circular signal

Given a sequence of samples of a continuous-time circular signal, sampled at times respectively, where the measurement technique/instrument is accurate - Estimate the mean value of the signal at period.

Reals example:

* Given airplane velocity measured each second [*MPH*] over a period of one hour – Estimate the mean velocity

Circular reals example:

* Given airplane heading measured each second over a period of one hour - Estimate the mean heading

For circular reals, it is not obvious in which direction the signal changes between two consecutive samples. For example, if one sample was 10°, and the next sample was 300°– did the signal changed in the increasing direction, or in the decreasing direction?

Since we can’t tell, our only way is to sample fast enough to ensure that the difference between consecutive measurements will always be less than 180°, so we can always be sure about the direction in which the signal changed. If the measure-rate is not high enough, a 330° change in the measured property would be incorrectly interpreted as a 30° change in the other direction, and will add error to the calculation.

Basically, in order to estimate the average, we should reconstruct the signal (interpolation), calculate the area bounded between the signal and the t-axis (integration), and divide it by (average).

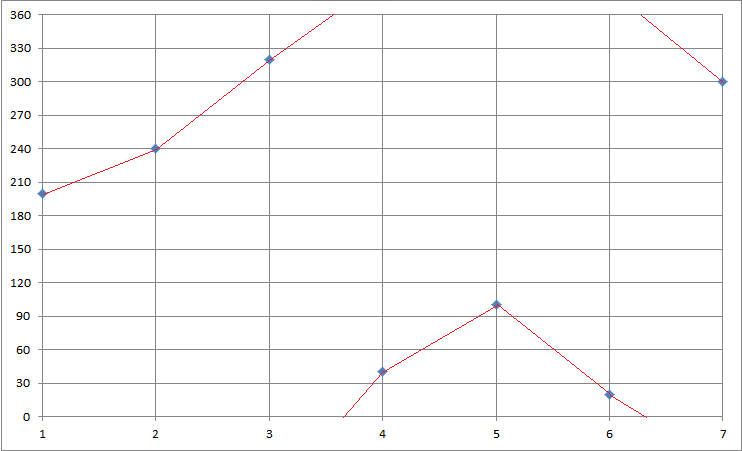
**Linear Interpolation**

The simplest reconstruction technique is linear interpolation (connecting every two consecutive samples by a straight line). We will discuss the circular real numbers equivalent to real numbers linear interpolation.

For real numbers, the average value of a signal, based on linear interpolation, is the weighted average of  
 with weight, for each in the range,   
where, as defined previously,,

For circular reals, it is the weighted circular average of   
 with weight, for each in the range,   
where, as defined previously,

For the above example, assume that the airplane heading measurements vector is {200, 240, 320, 40, 100, 20, 300}.



Here is how to calculate the average:

* Connect the samples such that the difference between consecutive samples will always be less than 180° (see the red line intervals in the figure above).
* Calculate the circular average, and the weight of each interval in the range: In our examples the intervals average values are {220, 280, 0, 70, 60, 340}
* Calculate the weighted circular average of all intervals average values   
  (non-weighted circular average may be used when the samples are equally spaced).  
  In our examples, assuming equally spaced sampling, it is 332.857.

Note that this method is more accurate than the Mitsuta method, described in the U.S. Meteorological Monitoring Guidance for Regulatory Modeling Applications (<http://www.epa.gov/scram001/guidance/met/mmgrma.pdf>), which gives equal weight to all measurements (For interpolation – the 1st and last measurements should have only half-weight) The Mitsuta method is also limited to equally-spaced samples.

The following code calculates the average estimation based on samples:

|  |
| --- |
| // estimate the average of a sampled continuous-time circular signal, using circular linear interpolation  // T is a circular real type defined with the CircValTypeDef macro  template<typename T>  class CAvrgSampledCircSignal  {  size\_t m\_nSamples ;  CircVal<T> m\_fPrevVal ; // previous value  double m\_fPrevTime; // previous time  vector<pair<CircVal<T>, double>> m\_Intervals; // vector of (avrg,weight) for each interval  public:  CAvrgSampledCircSignal()  {  m\_nSamples= 0;  }  void AddMeasurement(CircVal<T> fVal, double fTime)  {  if (m\_nSamples)  {  assert(fTime > m\_fPrevTime);  double fIntervalAvrg = CircVal<T>::Wrap((double)m\_fPrevVal +   CircVal<T>::Sdist(m\_fPrevVal, fVal)/2.) ;  double fIntervalWeight= fTime-m\_fPrevTime ;  m\_Intervals.push\_back(make\_pair(fIntervalAvrg, fIntervalWeight));  }  m\_fPrevVal = fVal ;  m\_fPrevTime= fTime;  ++m\_nSamples;  }  // calculate the weighted average for all intervals  bool GetAvrg(CircVal<T>& fAvrg)  {  switch (m\_nSamples)  {  case 0:  fAvrg= CircVal<T>::GetZ();  return false;  case 1:  fAvrg= m\_fPrevVal;  return true;  default:  fAvrg= \*WeightedCircAverage(m\_Intervals).begin();  return true;  }  }  }; |

Here is a sample use:

|  |
| --- |
| CAvrgSampledCircSignal<UnsignedDegRange> A1;  A1.AddMeasurement(CircVal<UnsignedDegRange>(200.), 1);  A1.AddMeasurement(CircVal<UnsignedDegRange>(300.), 2);  A1.AddMeasurement(CircVal<UnsignedDegRange>( 20.), 6);  CircVal<UnsignedDegRange> ad1;  A1.GetAvrg(ad1); |

**Quadratic Spline Interpolation**

In this reconstruction technique, we connect every two consecutive samples by a 2nd order polynomial. We will discuss the circular real numbers equivalent to real numbers quadratic spline interpolation.

For real numbers, the polynomial used for the interval [ is

, where .   
 is usually set to optimize the initial slope.

For real numbers, the average value of a signal, based on linear interpolation, is the weighted average of  
 with weight, for each in the range,   
where, as defined previously,,

For circular reals, it is the weighted circular average of   
 with weight, for each in the range,   
where, as defined previously,