Circular Values Math and Statistics with C++11

Lior Kogan, 2011-2013

1. Introduction

Many scientific and engineering problems involve circular values. Most of the times these values are orientation (angles) or cyclical timing (time-of-day), but it may be any other circular quantity. Such problems are very common in physics, geodesics and navigation, but also appear in other fields such as psychology, criminology (time-of-day statistics), bird-watching and biology (directional statistics and time-of-year statistics).

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

Mathematics and Statistics of circular values is very 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 $[-\pi,\ \pi)$ is used, sometimes $[0, 2\pi)$. Sometimes it is $[-180,\ 180)$ and sometimes $[0, 360)$ - even in the same application. For time-of-day computations, the range may be $[0,24)$, $[0, 24 \cdot 60 \cdot 60)$ or something else. We need a scheme for using several representations in the same application, and to safely perform operations between them.

2. Features

In this article I am going to present:

- Definition of circular value type and its associated operators
- A C++11 infrastructure for doing mathematics with circular values
- C++11 statistical distribution classes for circular values: Wrapped Normal and Wrapped Truncated Normal
- Average, Weighted average and Median of circular values
- Circular parameter estimation based on noisy independent measurements
- Interpolation and average estimation of sampled continuous-time circular signal

This is an original work. To the best of my knowledge, some of the ideas mentioned here were not published before.

3. 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.

4. Some helper functions

We'll define the following helper functions:

- Square function: Sqr(r1). This is simple:

```
// square (x*x)
template <typename T>
T Sqr(const T& x)
{
    return x*x;
}
```

- Floating-point modulo function: $reminder = \ mod(dividend, divisor)$
  The *dividend*, *divisor* and the resulting *reminder* are all floating-point values.

  The *quotient* and the *remainder* must satisfy:
  - *quotient* is integer
  - $|reminder| < |divisor|$
  - $dividend = quotient \cdot divisor + \ reminder$

  Still, many definitions are possible:
  A. The *remainder* has the same sign as the *dividend*
  - Implies that the *quotient* rounds towards zero
  B. The *remainder* has the same sign as the *divisor*
  - Implies that the *quotient* rounds towards negative infinity
  C. The *remainder* has a different sign than the *dividend*
  - Implies that the *quotient rounds away* away from zero
  D. The *remainder* has a different sign than the *divisor*
  - Implies that the quotient rounds towards positive infinity
  E. The sign of the *remainder* is always non-negative
  F. The sign of the *remainder* is always non-positive
  G. The *remainder* is closest to zero
  - Implies that the *quotient* is the integer nearest to the exact value of the division's result
  IEEE 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.
  H. The *remainder* is farthest from zero

  Values for $(quotient, reminder)$ according to the various implementations:

	A	B	C	D	E	F	G	H
$mod(+4,+3)$	$(+1,+1)$	$(+1,+1)$	$(+2,-2)$	$(+2,-2)$	$(+1,+1)$	$(+2,-2)$	$(+1,+1)$	$(+2,-2)$
$mod(-4,+3)$	$(-1,-1)$	$(-2,+2)$	$(-2,+2)$	$(-1,-1)$	$(-2,+2)$	$(-1,-1)$	$(-1,-1)$	$(-2,+2)$
$mod(+4,-3)$	$(-1,+1)$	$(-2,-2)$	$(-2,-2)$	$(-1,+1)$	$(-1,+1)$	$(-2,-2)$	$(-1,+1)$	$(-2,-2)$
$mod(-4,-3)$	$(+1,-1)$	$(+1,-1)$	$(+2,+2)$	$(+2,+2)$	$(+2,+2)$	$(+1,-1)$	$(+1,-1)$	$(+2,+2)$

	A	B	C	D	E	F	G	H
$mod(+5,+3)$	$(+1,+2)$	$(+1,+2)$	$(+2,-1)$	$(+2,-1)$	$(+1,+2)$	$(+2,-1)$	$(+2,-1)$	$(+1,+2)$
$mod(-5,+3)$	$(-1,-2)$	$(-2,+1)$	$(-2,+1)$	$(-1,-2)$	$(-2,+1)$	$(-1,-2)$	$(-2,+1)$	$(-1,-2)$
$mod(+5,-3)$	$(-1,+2)$	$(-2,-1)$	$(-2,-1)$	$(-1,+2)$	$(-1,+2)$	$(-2,-1)$	$(-2,-1)$	$(-1,+2)$
$mod(-5,-3)$	$(+1,-2)$	$(+1,-2)$	$(+2,+1)$	$(+2,+1)$	$(+2,+1)$	$(+1,-2)$	$(+2,+1)$	$(+1,-2)$

  In addition, we should define the behavior when the *divisor* is 0:
  X. When the *divisor* is 0, the *remainder* is undefined
  Y. When the *divisor* is 0, the *remainder* is defined as 0
  Z. 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:

$$mod(x,y) \equiv \; x - y\left\lfloor \frac{x}{y} \right\rfloor, if \; y \neq 0; \qquad mod(x,0) = x$$

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 values:

- Mod(-1e-16, 360.) = 360. (should be 0)
  The value 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:

$$\forall a \in \mathbb{Z}, \forall \mathrm{b}, \mathrm{x}, \mathrm{y} \in \mathbb{R}: mod(x,r) = mod(y,r) \Longrightarrow mod(ax+b,r) = mod(ay+b,r)$$

Proof:

$$mod(x,r) = mod(y,r) \Longrightarrow \exists n \in \mathbb{Z}: y = x + nr$$

$$mod(ay+b,r) = ay + b - r\left\lfloor\frac{ay+b}{r}\right\rfloor = a(x+nr) + b - r\left\lfloor\frac{a(x+nr)+b}{r}\right\rfloor$$

$$mod(ax+b,r) - mod(ay+b,r) = ax + b - r\left\lfloor\frac{ax+b}{r}\right\rfloor - a(x+nr) - b + r\left\lfloor\frac{a(x+nr)+b}{r}\right\rfloor$$

$$= r\left(-an + \left\lfloor\frac{a(x+nr)+b}{r}\right\rfloor - \left\lfloor\frac{ax+b}{r}\right\rfloor\right) = r\left(-an + \left\lfloor\frac{ax+b}{r} + an\right\rfloor - \left\lfloor\frac{ax+b}{r}\right\rfloor\right) = 0$$

- Almost-equal function: IsAlmostEqual(r1, r2)
  When implementing unit-testing for floating-point values computations, it is a common need to verify that 2 computations give equal results. Due to the finite-accuracy of floating-point types, many times the values 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/NAN values.

  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/

5. A circular value type

Definition 2:

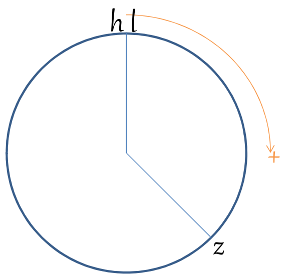
A Circular value type $C = < l, h, z >$ is defined by three constants:

- The range $[l, h)$: $l, h \in \mathbb{R},\ l < h$
- The zero-value $z \in [l, h)$.

Note the use of a right-open interval range.

For example, for angles in the range $[-\pi, \pi)$, it is natural to define the zero-value in the middle, while for angles in the range $[0, 2\pi)$ 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 will be defined below.



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

```
// macro for defining a circular-value 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 value types:

```
// basic circular-value 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 value types - for testing:

```
// some additional circular-value 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)
```

6. Defining a circular value class with a given type

The templated class CircVal is used for storing a single circular value.
The template parameter should be defined using the `CircValTypeDef` macro.

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

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

```
CircVal<UnsignedDegRange> c1;
```

7. Range checking and wrapping

`CircVal::IsInRange` static function is used for testing that a given linear value is within the range of a CircVal class.

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

The $wrap(r)$ function is used to 'wrap around' a linear value to the range of a given CircVal class.
For example: for the range $[0,360)$, the value 360 would be wrapper to 0, and 370 would be wrapped to 10. -350 would be wrapped to 10 as well.

Definition 3:

$$wrap(r) \equiv mod(r-l, h-l) + l = r - l - (h-l)\left\lfloor \frac{r-l}{h-l} \right\rfloor + l = r - (h-l)\left\lfloor \frac{r-l}{h-l} \right\rfloor$$

We can see that

When $2l - h \le r < l$:	$-(h-l) \le r - l < 0$	$\Rightarrow \left\lfloor \frac{r-l}{h-l} \right\rfloor = -1$	$\Rightarrow wrap(r) = r + (h-l)$
When $l \le r < h$:	$0 \le r - l < h - l$	$\Rightarrow \left\lfloor \frac{r-l}{h-l} \right\rfloor = 0$	$\Rightarrow wrap(r) = r$
When $h \le r < 2h - l$:	$(h-l) \le r - l < 2(h-l)$	$\Rightarrow \left\lfloor \frac{r-l}{h-l} \right\rfloor = 1$	$\Rightarrow wrap(r) = r - (h-l)$

We will use this for faster implementation.

The `CircVal::Wrap` static function calculates $wrap(r)$:

```
// 'wraps' circular-value 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;
}
```

8. A Walk

Definition 4:

For linear values, a walk $W = < s, d >$ is defined by:

- A start point $s \in \mathbb{R}$
- A directed length $d \in \mathbb{R}$

A walk is a movement along the linear axis, 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 $e \in \mathbb{R}$ is the point reached at the end of the walk.
We'll call a walk with a start point s and an end point e *"A walk from* s *to* e*"*.

$$e = s + d$$

There is only one possible walk from s to e: $< s, e - s >$

For any circular value type, a walk $W = < s, d >$ is defined by:

- A start point $s \in [l, h)$
- A directed length $d \in \mathbb{R}$

A walk is a movement along the circular axis, 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 $e \in [l, h)$ is the point reached at the end of the walk.
We'll call a walk with a start point s and an end point e *"A walk from* s *to* e*"*.
Since the walk is wrapped around the circle:

$$e = wrap(s + d)$$

There are infinitely many walks from s to e:

$$< s, (e - s) + (h - l) \cdot z >, \qquad z \in \mathbb{Z}$$

The shortest walk from s to e is the one that minimizes $|d|$. The walk may be increasing or decreasing.

9. Directed distance between two values

Definition 5:

The *directed distance* from linear value $r1$ to $r2$:

$$sdist(r1, r2) \equiv r2 - r1,$$

is the directed length of the walk from $r1$ to $r2$.

For example, $sdist(4,10) = 6,\ sdist(10,4) = -6$

Similarly, the *directed distance* from circular value $c1$ to $c2$:

$$sdist(c1, c2) \equiv mod\left(c2 - c1 + \tfrac{h-l}{2}, h - l\right) - \tfrac{h-l}{2},$$

is the directed length of the shortest walk from $c1$ to $c2$.

$sdist(c1, c2)$ is a linear value in the range $\left[-\tfrac{h-l}{2}, \tfrac{h-l}{2}\right)$.

It can easily be deduced that:

$$sdist(c1, c2) = -sdist(c2, c1), except\ for\ when\ |c2 - c1| = \frac{h-l}{2}$$

Since

$$sdist(c1, c2) \equiv mod\left(c2 - c1 + \frac{h-l}{2}, h - l\right) - \frac{h-l}{2} = c2 - c1 + \frac{h-l}{2} - (h-l)\left\lfloor\frac{c2 - c1 + \frac{h-l}{2}}{h-l}\right\rfloor - \frac{h-l}{2}$$

$$= c2 - c1 - (h-l)\left\lfloor\frac{c2 - c1 + \frac{h-l}{2}}{h-l}\right\rfloor = (c2 - c1) - (h-l)\left\lfloor\frac{c2-c1}{h-l} + \frac{1}{2}\right\rfloor$$

And since

$$l \le c1, c2 < h \implies -(h-l) < c2 - c1 < h - l \implies -1 < \frac{c2-c1}{h-l} < 1 \implies \left\lfloor\frac{c2-c1}{h-l} + \frac{1}{2}\right\rfloor \in \{-1,0,1\}$$

We can write:

$$sdist(c1, c2) \equiv \begin{cases} (c2 - c1) + (h - l), & c2 - c1 < -\dfrac{h-l}{2} \\ (c2 - c1), & -\dfrac{h-l}{2} \le c2 - c1 < \dfrac{h-l}{2} \\ (c2 - c1) - (h - l), & \dfrac{h-l}{2} \le c2 - c1 \end{cases}$$

which we will use for faster implementation.

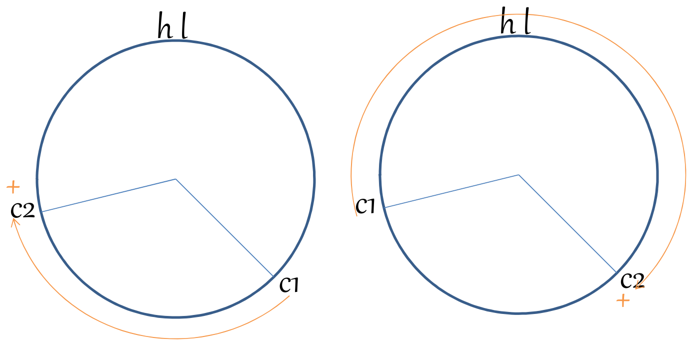
The `CircVal::Sdist` static function calculates $sdist(c1, c2)$:

```
// the length of the direct walk from c1 to c2, with the lowest absolute-value length
// return value 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 $|sdist(c1, c2)|$ as our metric (we won't prove this here)

For circular values, we'll define an additional distance function:

$pdist(c1, c2)$ - The *increasing distance* from circular value $c1$ to $c2$, is the length of the shortest increasing-walk (depicted clockwise in the picture below) from $c1$ to $c2$.
It is always in the range $[0, h-l)$.



There is no equivalent distance function for linear values.

Definition 6:

$$pdist(c1, c2) \equiv mod(c2 - c1, h - l)$$

which may also be expressed as:

$$pdist(c1, c2) = \begin{cases} c2 - c1, & c2 \geq c1 \\ (h - c1) + (c2 - l) = (c2 - c1) + (h - l), & c2 < c1 \end{cases}$$

It can easily be deduced that:

- $pdist(c1, c2) + pdist(c2, c1) = h - l$
- $|pdist(c1, c2)| \neq |pdist(c2, c1)|$ Hence, it cannot be used as a metric

The `CircVal::Pdist` static function calculates $pdist(c1, c2)$:

```
// the length of the increasing walk from c1 to c2 with the lowest length
// return value 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:

$$pdist(c1,c2) \equiv \begin{cases} sdist(c1,c2), & sdist(c1,c2) \geq 0 \\ sdist(c1,c2) + (h-l), & sdist(c1,c2) < 0 \end{cases}$$

Proof:

Rephrasing the definitions:

$$sdist(c1,c2) \equiv \begin{cases} (c2-c1)+(h-l) \quad > \mathbf{0}, & -(h-l) < c2-c1 < -\frac{h-l}{2} \\ (c2-c1) \quad < \mathbf{0}, & -\frac{h-l}{2} \leq c2-c1 < 0 \\ (c2-c1) \quad > \mathbf{0}, & 0 \leq c2-c1 < \frac{h-l}{2} \\ (c2-c1)-(h-l) \quad < \mathbf{0}, & \frac{h-l}{2} \leq c2-c1 < h-l \end{cases}$$

$$pdist(c1,c2) \equiv \begin{cases} (c2-c1)+(h-l) = sdist(c1,c2), & -(h-l) < c2-c1 < -\frac{h-l}{2} \\ (c2-c1)+(h-l) = sdist(c1,c2)+(h-l), & -\frac{h-l}{2} \leq c2-c1 < 0 \\ c2-c1 = sdist(c1,c2), & 0 \leq c2-c1 < \frac{h-l}{2} \\ c2-c1 = sdist(c1,c2)+(h-l), & \frac{h-l}{2} \leq c2-c1 < h-l \end{cases}$$

Theorem 3:

$|sdist(c1,c2)| = min(|c2-c1|, (h-l)-|c2-c1|)$ (Very intuitive)

Proof:

According to the definition above:

$$|sdist(c1,c2)| = \begin{cases} (h-l)-|c2-c1|, & -(h-l) < c2-c1 < -\frac{h-l}{2} \\ |c2-c1|, & -\frac{h-l}{2} \leq c2-c1 < 0 \\ |c2-c1|, & 0 \leq c2-c1 < \frac{h-l}{2} \\ (h-l)-|c2-c1|, & \frac{h-l}{2} \leq c2-c1 < h-l \end{cases}$$

And also

$$min(|c2-c1|, (h-l)-|c1-c2|) = \begin{cases} (h-l)-|c1-c2|, & -(h-l) < c2-c1 < -\frac{h-l}{2} \\ |c2-c1|, & -\frac{h-l}{2} \leq c2-c1 < 0 \\ |c2-c1|, & 0 \leq c2-c1 < \frac{h-l}{2} \\ (h-l)-|c2-c1|, & \frac{h-l}{2} \leq c2-c1 < h-l \end{cases}$$

, which are equal in all cases.

Theorem 4:

For any two circular values $c1, c2$: $wrap\left(sdist(c1, c2)\right) = wrap(c1 - c2)$

Proof:

$$wrap\left(sdist(c1, c2)\right) \equiv \begin{cases} wrap\left((c2 - c1) + (h - l)\right) = wrap(c1 - c2), & c2 - c1 \le -\frac{h - l}{2} \\ wrap\left((c2 - c1)\right), & |c2 - c1| < \frac{h - l}{2} \\ wrap\left((c2 - c1) - (h - l)\right) = wrap(c1 - c2), & c2 - c1 \ge \frac{h - l}{2} \end{cases}$$

Theorem 5

For any two circular values $c1, c2$: $wrap\left(pdist(c1, c2)\right) = wrap(c2 - c1)$

Proof:

$$wrap\left(pdist(c1, c2)\right) \equiv \begin{cases} wrap(c2 - c1), & c2 \ge c1 \\ wrap\left((c2 - c1) + (h - l)\right) = wrap(c2 - c1), & c2 < c1 \end{cases}$$

Theorem 6:

For any two circular values $c1, c2$: $wrap\left(sdist(c1, c2)\right) = wrap\left(pdist(c1, c2)\right)$

Proof:

Trivial, based on theorems 4, 5

10. Conversion between different types of circular values

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

For example:

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

d3= d1+d2;
```

To convert a circular value $c2$ from circular type $C2 < l2, h2, z2 >$ to circular type $C1 < l1, h1, z1 >$, we'll retain the arc-length between $c2$ and $z2$. For that, we can use either of these formulae:

I $\quad c1 = C1.wrap\left(z1 + (h1 - l1) \cdot \frac{C2.sdist(z2, c2)}{h2 - l2}\right)$

II $\quad c1 = C1.wrap\left(z1 + (h1 - l1) \cdot \frac{C2.pdist(z2, c2)}{h2 - l2}\right)$

Theorem 7:

I and **II** are equivalent

Proof:

I $$C1.wrap\left(z1+(h1-l1)\cdot\frac{C2s.sdist(z2,c2)}{h2-l2}\right)$$
$$=z1+(h1-l1)\cdot\frac{C2.sdist(z2,c2)}{h2-l2}-(h1-l1)\left\lfloor\frac{z1+(h1-l1)\cdot\frac{C2.sdist(z2,c2)}{h2-l2}-l1}{h1-l1}\right\rfloor$$
$$=z1+(h1-l1)\left(\frac{C2.sdist(z2,c2)}{h2-l2}-\left\lfloor\frac{z1-l1}{h1-l1}+\frac{C2.sdist(z2,c2)}{h2-l2}\right\rfloor\right)$$

II $$C1.wrap\left(z1+(h1-l1)\cdot\frac{C2.pdist(z2,c2)}{h2-l2}\right)$$
$$=z1+(h1-l1)\cdot\frac{C2.pdist(z2,c2)}{h2-l2}-(h1-l1)\left\lfloor\frac{z1+(h1-l1)\cdot\frac{C2.pdist(z2,c2)}{h2-l2}-l1}{h1-l1}\right\rfloor$$
$$=z1+(h1-l1)\left(\frac{C2.pdist(z2,c2)}{h2-l2}-\left\lfloor\frac{z1-l1}{h1-l1}+\frac{C2.pdist(z2,c2)}{h2-l2}\right\rfloor\right)$$

According to theorem 2, $pdist(z2,c2)\equiv\begin{cases}sdist(z2,c2), & sdist(z2,c2)\ge 0\\ sdist(z2,c2)+(h2-l2), & sdist(z2,c2)<0\end{cases}$

Case 1: $pdist(z2,c2)=\ sdist(z2,c2)$ - **I** =**II**: trivial
Case 2: $pdist(z2,c2)=\ sdist(z2,c2)+(h2-l2)$:

$$=z1+(h1-l1)\left(\frac{C2.sdist(z2,c2)}{h2-l2}+1-\left\lfloor\frac{z1-l1}{h1-l1}+\frac{C2.sdist(z2,c2)}{h2-l2}+1\right\rfloor\right)$$
$$=z1+(h1-l1)\left(\frac{C2.sdist(z2,c2)}{h2-l2}+\left\lfloor\frac{z1-l1}{h1-l1}+\frac{C2.sdist(z2,c2)}{h2-l2}\right\rfloor\right)$$

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

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

```
// construction based on a circular value 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 value
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;
}
```

11. Constructing circular value / Fetching circular value

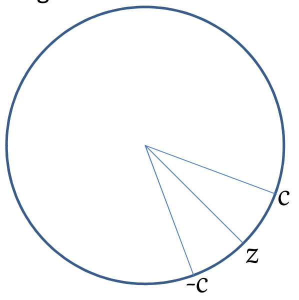
Construction a CircVal objects based on linear value-representation is implemented by the appropriate constructor and assignment operator. Construction of a linear value based on a CircVal object is implemented with `operator double() const` overloading.

```
// construction based on a floating-point value
// should only be called when the floating-point is a value 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 floating-point value
// should only be called when the floating-point is a value 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;
}
```

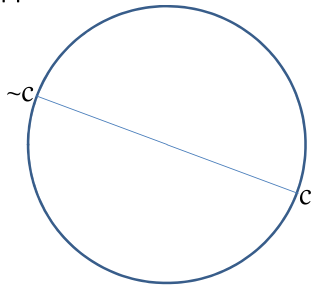
12. Circular value operators – Intuitive description

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

- Negation operator $(-c)$ is defined for any circular value. The result is a circular value. Negation of a circular value is the symmetric value relative to z (the zero-value).

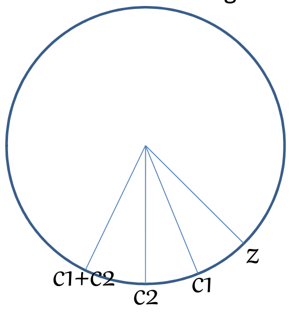


- Opposite operator $(\sim c)$ is defined for any circular value. The result is a circular value. Opposite of a circular value is the circular value pointing to the opposite direction.

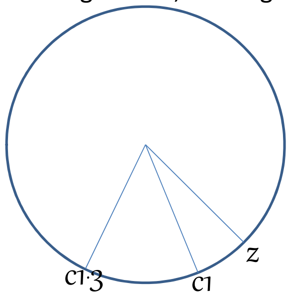


- Addition and subtraction operators $(c1 \pm c2)$ are defined for any two circular values. The result is a circular value - the point reached and the end of the walk:

Addition: Starting from $c1$, walking with length $pdist(z, c2)$.
Subtraction: Starting from $c1$, walking with length $-pdist(z, c2)$.



- Multiplication operator $(c \cdot r)$ is defined for any circular value c and any linear value r.
  The result is a circular value - the point reached and the end of the walk:
  Starting from z, walking distance $r \cdot pdist(c, z)$.



- Division operator (c/r) is defined for any circular value c and any non-zero linear value r.
  The result is a circular value - the point reached and the end of the walk:
  Starting from z, walking $\frac{1}{r}$ times distance $pdist(c, z)$.

- Trigonometric functions sin, cos, tan are defined for any circular value c, such that the result will be a linear value, in accordance with the common functions usage.

- Inverse trigonometric functions $asin, acos, atan, atan2$ shall be defined for any linear number ($-1 \leq r < 1$, for $asin, acos$), such that the result will be a circular value of our class in accordance with the common functions usage.

13. Conversion between circular values and linear values

We want a conversion function $ToR(c)$, such that any circular value c in our range $[l, h)$ can be mapped to a linear value, and a conversion function $ToC(\mathrm{r})$, such that any linear value r can be mapped to a circular value in our range $[l, h)$. Note that these functions are different from the circular value construction/fetching described above.

Requirements to these conversion functions:

- $ToC(0) = z$
- $ToR(z) = 0$
- For any circular value c: $\mathrm{c} = ToC(ToR(\mathrm{c}))$
- For any circular value c: $-\mathrm{c} = ToC(ToR(-\mathrm{c}))$
- For any two circular values $c1, c2$: $c1 + \mathrm{c2} = ToC(ToR(\mathrm{c1}) + ToR(\mathrm{c2}))$
- For any two circular values $c1, c2$: $\mathrm{c1} - \mathrm{c2} = ToC(ToR(\mathrm{c1}) - ToR(\mathrm{c2}))$
- For any circular value c, and any non-negative linear value $0 < r \leq 1$: $\mathrm{c} \cdot \mathrm{r} = ToC(ToR(\mathrm{c}) \cdot \mathrm{r})$
- For any circular value c, and any linear value $r \geq 1$: $\mathrm{c}/\mathrm{r} = ToC(ToR(\mathrm{c})/\mathrm{r})$
- For any trigonometric function $f: f \in \{sin, cos, tan\}$, for any circular value c: $f(c) = f\left(ToR(ToSignedRad(c))\right)$,
  where $ToSignedRad$ converts from the given circular values class to the circular values class $C < -\pi, \pi, 0 >$
- For any linear value $-1 \leq r \leq 1$: $asin(r) = SignedRad.asin(c)$, $acos(r) = SignedRad.acos(r)$
- For any linear value r: $atan(r) = SignedRad.atan(r)$

We'll use these conversion functions to simplify the requirements and the definitions of several circular value operators.

14. Requirements for circular value operators

We want to define the circular value operators $\{-c, c1 + c2, c1 - c2, \sim c, c \cdot r, c/r, \sin(c), \cos(c), \tan(c), \mathrm{C.asin}(r), \mathrm{C.acos}(r), \mathrm{C.atan}(r), C.atan2(r1, r2)\}$ such that the following conditions are satisfied:

- $z = -z$
- For any circular values c: $-(-c) = c$
- For any circular value c: $c - c = z,\ \ c + (-c) = z,\ \ c + z = c,\ \ c - z = c,\ \ z - c = -c$
- For any two circular values $c1, c2$: $c1 + c2 = c2 + c1,\ \ c1 - c2 = -(c2 - c1)$
- For any three circular values $c1, c2, c3$: $c1 + (c2 + c3) = (c1 + c2) + c3$
- For any circular values c: $\sim(\sim c) = c$
- For any circular values c: $c - (\sim c) = ToC\left(\frac{\mathrm{h}-\mathrm{l}}{2}\right)$
- For any circular values c: $c \cdot 0 = z,\ \ c \cdot 1 = c$
- For any circular values c: $c/1 = c$
- For any circular values c, and for any positive linear value $0 < r \leq 1$: $(c \cdot \mathrm{r})/\mathrm{r} = c$
- For any circular values c, and for any positive linear value $r \geq 1$: $(c/\mathrm{r}) \cdot \mathrm{r} = c$
- $sin(z) = 0,\ \ cos(z) = 1,\ \ tan(z) = 0$
- For any circular values c (using circular trigonometric functions):
  $sin(-c) = -sin(c), cos(-c) = cos(c), tan(-c) = tan(c)$
  $$sin\left(c + ToC\left(\frac{h-l}{4}\right)\right) = cos(c)$$
  $$cos\left(c + ToC\left(\frac{h-l}{4}\right)\right) = -sin(c)$$
  $$sin\left(c + ToC\left(\frac{h-l}{2}\right)\right) = -sin(c)$$
  $$cos\left(c + ToC\left(\frac{h-l}{2}\right)\right) = -cos(c)$$
  $sin^2(c) + cos^2(c) = 1$
  $sin(c)/cos(c)\ = tan(c)$
- $asin(0) = z,\ \ acos(1) = z,\ \ atan(0) = z$
- For any linear value $-1 \leq r < 1$ (using circular inverse trigonometric functions):
  $asin(\mathrm{r}) + asin(-\mathrm{r}) = z$
  $$acos(\mathrm{r}) + acos(-\mathrm{r}) = ToC\left(\frac{\mathrm{h}-\mathrm{l}}{2}\right)$$
  $$asin(\mathrm{r}) + acos(-\mathrm{r}) = ToC\left(\frac{\mathrm{h}-\mathrm{l}}{4}\right)$$
- For any linear value r (using circular inverse trigonometric function):
  $atan(\mathrm{r}) + atan(-\mathrm{r}) = z$

The set of real numbers in the range $[l, h)$ with the $+$ binary operator is a bounded group (z is the identity element, and $-c$ is the inverse element of c).

15. Comparison operators

Defining operators $=$ and $\neq$ is straightforward. The other 4 comparison operators $\{<, >, \leq, \geq\}$ can be defined in several different ways. Requirements to these comparison operators (For any circular values $c1, c2, c3$) should satisfy:

- $c1 > c2 \Leftrightarrow c2 < c1$
- $c1 \geq c2 \Leftrightarrow c2 \leq c1$
- $c1 \geq c2 \Leftrightarrow (c1 > c2) \vee (c1 = c2)$
- $\vee \begin{pmatrix} \wedge\big((c1 = c2), \sim(c1 < c2), \sim(c1 > c2)\big) \\ \wedge\big(\sim(c1 = c2), (c1 < c2), \sim(c1 > c2)\big) \\ \wedge\big(\sim(c1 = c2), \sim(c1 < c2), (c1 > c2)\big) \end{pmatrix}$, in other words $\begin{cases} c1 > c2 \Leftrightarrow \sim(c1 = c2) \wedge \sim(c1 < c2) \\ c1 = c2 \Leftrightarrow \sim(c1 > c2) \wedge \sim(c1 < c2) \\ c1 < c2 \Leftrightarrow \sim(c1 = c2) \wedge \sim(c1 > c2) \end{cases}$
- $(c1 > c2) \wedge (c2 > c3) \Longrightarrow c1 > c3$, in other words: $\sim(c1 > c2) \vee \sim(c2 > c3) \vee (c1 > c3)$

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

For linear values we can define:

$$r1 > r2 \equiv \exists s: (r1 - r2)s^2 = 1; \qquad r1, r2, s \in \mathbb{R}$$

Based on this, we can define the circular values $>$ operator in different ways:

- $c1 > c2 \equiv c1 > c2$
- $c1 > c2 \equiv pdist(z, c1) > pdist(z, c2)$
- $c1 > c2 \equiv sdist(z, c1) > sdist(z, c2)$

Hence, for two circular values $c1, c2$, for each comparison operator $\diamond \in \{<, >, \leq, \geq\}$, we can use:

(i) $c1 \diamond c2 \ \equiv ToR(c1) \diamond ToR(c2)$
(ii) $c1 \diamond c2 \ \equiv pdist(z, c1) \diamond pdist(z, c2)$
(iii) $c1 \diamond c2 \ \equiv sdist(z, c1) \diamond sdist(z, c2)$

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

In many practical situations $z = 0$.

For $C = < -180{,}180{,}0 >$ it seems more appropriate to use (i) or (iii) which are equivalent, while for $C = < 0{,}360{,}0 >$, (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 values 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; }
```

16. Circular value operators – Formal definition

Now, here is the definition that satisfies all the requirements

Definition 7:

- $ToR(c) \equiv c - z$
- $ToC(r) \equiv wrap(r + z)$
- For a circular value c: $-c \equiv wrap(2z - c)$
- For a circular value c: $\sim c \equiv wrap\left(c + \frac{h-l}{2}\right)$
- For two circular values $c1, c2$: $c1 + c2 \equiv wrap(c1 + c2 - z)$
- For two circular values $c1, c2$: $c1 - c2 \equiv wrap(c1 - c2 + z)$
- For a circular value c and a non-negative linear value r: $c \cdot r \equiv wrap((c - z) \cdot r + z)$
- For a circular value c and a positive linear value r: $c/r \equiv wrap((c - z)/r + z)$
- For each Trigonometric function f: $f \in \{sin, cos, tan\}$, for any circular value c: $f(c) \equiv f\left(ToR\left(ToSignedRad(c)\right)\right)$
  $ToSignedRad$ Converts from the given circular values class to the circular values class $< -\pi, \pi, 0 >$
- For each Inverse trigonometric function g: $g \in \{asin, acos, atan\}$near value r: $g(r) \equiv FromSignedRad\left(g(r)\right)$
  $FromSignedRad$ Converts from the circular values class $< -\pi, \pi, 0 >$ to the given circular values class.

Theorem 8:

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

- $-c = z - pdist(z, c)$
- $-c = z - sdist(z, c)$

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

- $c1 \pm c2 = wrap\left(z + sdist(z, c1) \pm sdist(z, c2)\right)$
- $c1 \pm c2 = wrap\left(z + pdist(z, c1) \pm pdist(z, c2)\right)$

Proof:

Since $sdist(c1, c2) = (c2 - c1) + n(h - l)$, where $(n = -1) \vee (n = 0) \vee (n = 1)$,
and since $pdist(c1, c2) = (c2 - c1) + n(h - l)$, where $(n = 0) \vee (n = 1)$,

$$z - pdist(z, c) = z - (c - z) + n(h - l) = wrap(2z - c)$$
$$z - sdist(z, c) = z - (c - z) + n(h - l) = wrap(2z - c)$$

$$wrap\left(z + sdist(z, c1) + sdist(z, c2)\right) = wrap\left(z + (c1 - z) + (c2 - z)\right) = wrap(c1 + c2 - z)$$
$$wrap\left(z + pdist(z, c1) + pdist(z, c2)\right) = wrap\left(z + (c1 - z) + (c2 - z)\right) = wrap(c1 + c2 - z)$$
$$wrap\left(z + sdist(z, c1) - sdist(z, c2)\right) = wrap\left(z + (c1 - z) - (c2 - z)\right) = wrap(c1 - c2 + z)$$
$$wrap\left(z + pdist(z, c1) - pdist(z, c2)\right) = wrap\left(z + (c1 - z) - (c2 - z)\right) = wrap(c1 - c2 + z)$$

Here is the implementation of these operators:

```
template <typename Type>
class CircVal
{
    double val; // actual value [L,H)

public:
    // ---------------------------------------------
    // convert circular-value c to real-value [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 value
    const CircVal  operator~ (                ) const { return Wrap(val+Type::R_2             ); } // return opposite circular-value

    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`.

17. Testing for near-equality of circular values

Based on the previously describe test for near-equality for two linear values, we will construct a near-equality test for two circular values:

```
template <typename Type>
class CircValTester
{
    // check if 2 circular-values 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-values are almost equal
    static void AssertCircAlmostEq(const CircVal<Type>& f, const CircVal<Type>& g)
    {
        assert(IsCircAlmostEq(f, g));
    }
    …
}
```

18. Testing correctness of circular value 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 value
            CircVal<Type> c2(c_uni_dist(rand_engine)); // random circular value
            CircVal<Type> c3(c_uni_dist(rand_engine)); // random circular value
            double        r (r_uni_dist(rand_engine)); // random real     value [    0, 1000) - for testing *,/ operators
            double        a1(t_uni_dist(rand_engine)); // random real     value [   -1,    1) - for testing asin,acos
            double        a2(t_uni_dist(rand_engine)); // random real     value [-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 all 8 circular-value defined 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;
```

19. 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.

20. C++11 Distribution classes for circular values

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 values, is the truncated normal distribution.

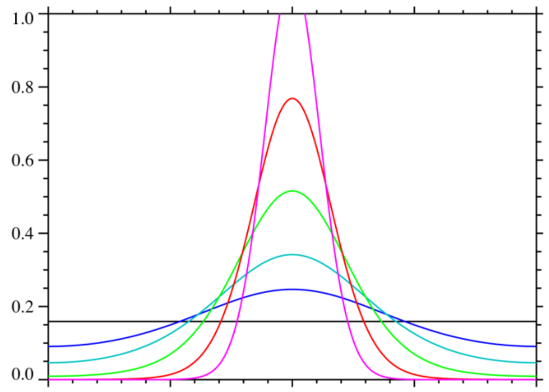
We have implemented three distribution classes: wrapped normal, truncated normal and wrapped truncated normal.

21. Wrapped Normal Distribution class

A wrapped normal distribution results from the "wrapping" of the normal distribution around the range $[\boldsymbol{l}, \boldsymbol{h})$. Therefore, the probability density function $\boldsymbol{p}(\emptyset)$ of the wrapped normal distribution is:

$$\boldsymbol{f}(\boldsymbol{\theta};\ \boldsymbol{\mu}, \boldsymbol{\sigma}, \boldsymbol{l}, \boldsymbol{h}) = \frac{\mathbf{1}}{\boldsymbol{\sigma}\sqrt{\mathbf{2}\boldsymbol{\pi}}} \sum_{\boldsymbol{k}=-\infty}^{\infty} \boldsymbol{e}^{\frac{-(\theta-\mu+(h-l)k)^2}{2\sigma^2}}$$, where $\boldsymbol{l} < \boldsymbol{h}$ and $\boldsymbol{\theta} \in [\boldsymbol{l}, \boldsymbol{h})$

The PDF is described in the following image, for same $\boldsymbol{\mu}, \boldsymbol{a}, \boldsymbol{b}$, but different values of $\boldsymbol{\sigma}$:



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:

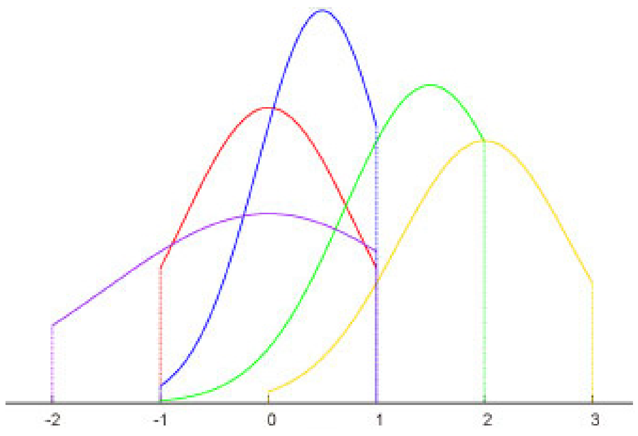
```
#include "WrappedNormalDist.h"

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.

22. Truncated Normal Distribution class

This class has nothing to do with circular values; 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 random variable whose value is bounded to the range $[\boldsymbol{a}, \boldsymbol{b})$.
The PDF is described in the following image, for different values of $\boldsymbol{\mu}, \boldsymbol{\sigma}, \boldsymbol{a}, \boldsymbol{b}$:



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"

23. Wrapped Truncated Normal Distribution class

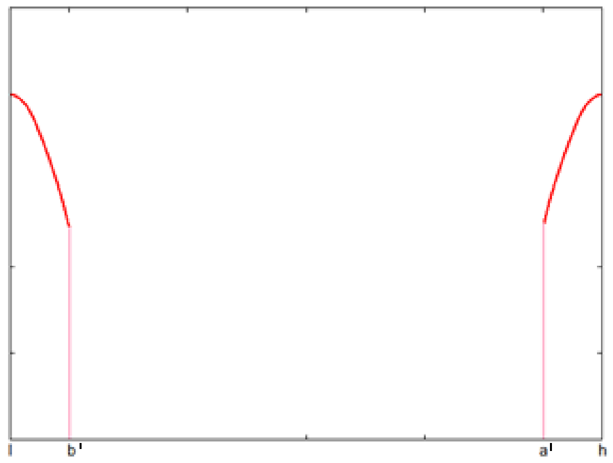
Same as for linear values, circular values wrapped normal distribution may be truncated as well.
Note that the normal distribution is first truncated to the range $[\boldsymbol{a}, \boldsymbol{b})$, and then wrapped around the range $[\boldsymbol{l}, \boldsymbol{h})$.

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 $b - a < h - l$, 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 $[-50,50)$ hours.

After wrapping, it is possible that $wrap(\boldsymbol{b}) < wrap(\boldsymbol{a})$. 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
```

24. 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 values. 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 linear values and for circular values.

Problem Type 1:

Given [circular] values $x_1..x_N$ – calculate their mean
-or-
Given a random sample $x_1..x_N$ from a [circular] random-variable X - calculate the sample mean

Usage: Averaging values

Examples:

- linear: 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: 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:

- Linear: 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: 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.

- Linear: 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: 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 $x_1..x_N$ of a continuous-time [circular] signal, sampled at times $t_1..t_N$ respectively, where the measurement technique/instrument is accurate - Estimate the mean value of the signal at period $[t_1, t_N]$

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

Examples:

- Linear: Given airplane velocity measured each 1 second [MPH] over a period of one hour – Estimate the mean velocity.
- Circular: Given airplane heading measured each 1 second $[0..360)$ over a period of one hour - Estimate the mean heading.

25. Averaging 2 circular values

To average two linear values, $\boldsymbol{r_1, r_2,}$ we can start at $\boldsymbol{r_1}$ and take a walk with length $\frac{r_2-r_1}{2}$. The end-point of the walk is the average. We can, by symmetry, start at $\boldsymbol{r_2}$ and take a walk with length $\frac{r_1-r_2}{2}$.

For linear values $\boldsymbol{average(r_1, r_2) = r1 + \frac{sdist(r_1,r_2)}{2} = r2 + \frac{sdist(r_2,r_1)}{2}}$
$\boldsymbol{r_1 + \frac{sdist(r_1,r_2)}{2} = r_1 + \frac{r_2-r_1}{2} = \frac{r_1+r_2}{2}; \quad r_2 + \frac{sdist(r_2,r_1)}{2} = r_2 + \frac{r_1-r_2}{2} = \frac{r_1+r_2}{2}}$

For circular values, 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 values is a not a single circular value, but a set of circular values.
$average(0, 180) = \{90, 270\}$ - There is no reason to prefer one of these values over the other.

We will use the following formula:
$average(c_1, c_2) = \left\{wrap\left(c_1 + \frac{sdist(c_1,c_2)}{2}\right), wrap\left(c_2 + \frac{sdist(c_2,c_1)}{2}\right)\right\}$,
These two set members are equal, except when $sdist(c_1, c_2) = sdist(c_2, c_1) =- \frac{h-l}{2}$.
In such case, the average is the set $\left\{wrap\left(c_1 - \frac{h-l}{4}\right), wrap\left(c_2 - \frac{h-l}{4}\right)\right\}$
Of course, $average(c_1, c_2) = average(c_2, c_1)$

26. Averaging n circular values

Circular values average calculation is really tricky. Observing algorithms described in many textbooks, and code written by many programmers, I've almost never seen a correct method used.

Let us start with linear values. 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 values $A = \{a_1, \dots, a_n\}$ for any mathematical field F, is the value of $x \in F$ that minimizes $\sum_{i=1}^{n} sdist(a_i, x)^2$. Formally: $average(A) \equiv \underset{x \in F}{\operatorname{argmin}} \sum_{i=1}^{n} sdist(a_i, x)^2$.

For linear values, we can calculate the average as follow:
$sdist(r1, r2) = r2 - r1$
Let y denote the expression we want to minimize.

$y = \sum sdist(a_i, x)^2 = \sum (x - a_i)^2 = \sum (x^2 - 2xa_i + a_i^2) = nx^2 - 2x \sum a_i + \sum a_i^2$
To find the value of x that minimizes y, we need the derivative $\frac{dy}{dx}$ to be 0:

$\frac{dy}{dx} = 2nx - 2\sum a_i$;
for $\frac{dy}{dx} = 0$: $2nx - 2\sum a_i = 0 \rightarrow x = \frac{\sum a_i}{n}$;
We got the well-known arithmetic mean formula for linear values.

Now let's talk about circular values. 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 $[0,360)$.

Note that the true average of 0, 0, and 90 should be 30, and not atan $\left(\frac{\sin 0 + \sin 0 + \sin 90}{\cos 0 + \cos 0 + \cos 90}\right) \cong 25.656$, as suggested in most references.

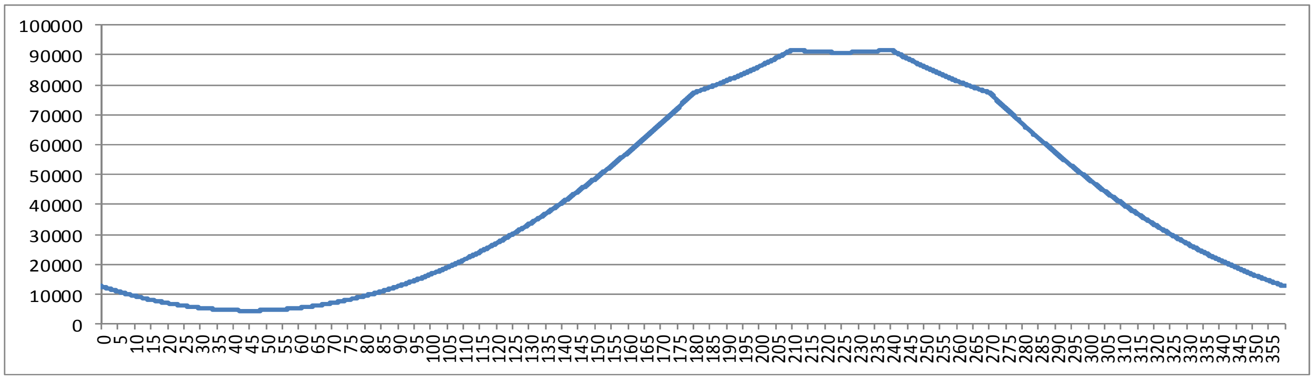
Based on theorem 3, the absolute value of the distance between 2 circular values in the range $[0,360)$ is
$|sdist(c1, c2)| = min(|c2 - c1|, 360 - |c2 - c1|)$

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 y denote the expression we want to minimize.

$y = \sum_{i=1}^{n} sdist(a_i, x)^2 = \sum_{i=1}^{n} (\min(|x - a_i|,\ 360 - |x - a_i|))^2$
The values of x that minimizes y is our average.

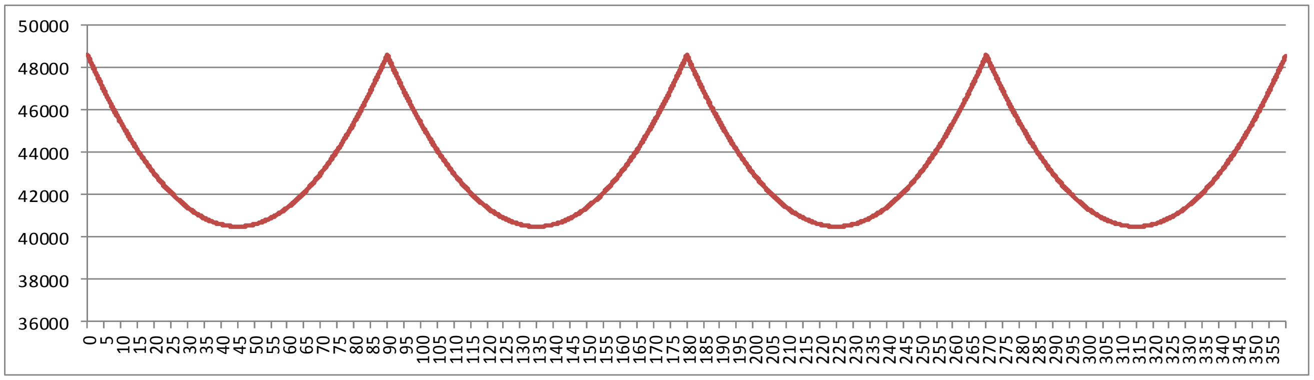
Here are some examples:

Example 1: For the input $\{0,30,60,90\}$, we'll plot $\sum sdist(a_i, x)^2$ as a function of x:



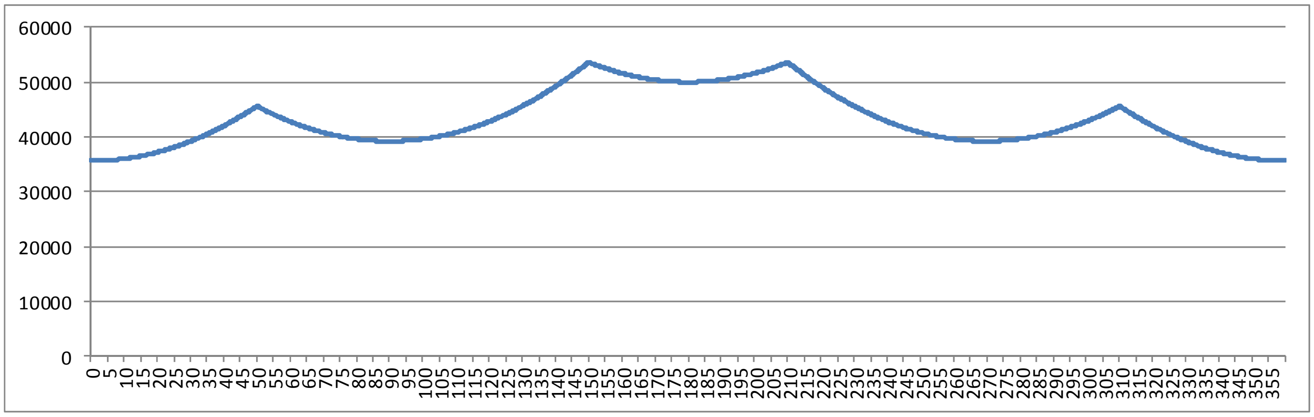
We can see that the average is $\{45\}$ which matches out intuition.

Example 2: For the input $\{0,90,180,270\}$, we'll plot $\sum sdist(a_i, x)^2$ as a function of x:



In this case the average is $\{45, 135, 225, 315\}$
(For all these values, $\sum sdist(a_i, x)^2$ is equal).

Example 3: For the input $\{30,130,230,330\}$, we'll plot $\sum sdist(a_i, x)^2$ as a function of x:



In this case, the average is $\{0\}$. 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 x that minimizes y, we need the derivative $\frac{dy}{dx}$ to be 0.
However, due to the min and abs operators, the expression $y = \sum(\min(|x - a_i|,\ 360 - |x - a_i|))^2$ is not derivable.

Let us try to rephrase the expression.
We'll divide the multiset A into 3 distinct multi-subsets:
B: multi-subset of A, where $sdist(a_i, x)^2 = (\min(|x - a_i|,\ 360 - |x - a_i|))^2 = (x - a_i)^2$
C: multi-subset of A, where $sdist(a_i, x)^2 = (\min(|x - a_i|,\ 360 - |x - a_i|))^2 = (360 - (a_i - x))^2$
D: multi-subset of A, where $sdist(a_i, x)^2 = (\min(|x - a_i|,\ 360 - |x - a_i|))^2 = (360 - (x - a_i))^2$

We'll denote with a, b, c, d – the number of elements in A, B, C, D respectively.
Of course $a = b + c + d$; $\sum a_i = \sum b_i + \sum c_i + \sum d_i$

$$
\begin{aligned}
y &= \sum sdist(a_i, x)^2 = \sum sdist(b_i, x)^2 + \sum sdist(c_i, x)^2 + \sum sdist(d_i, x)^2 \\
&= \sum(x - b_i)^2 + \sum\big(360 - (c_i - x)\big)^2 + \sum\big(\,360 - (x - d_i)\big)^2 \\
&= \sum(x - b_i)^2 + \sum\big((360 - c_i) + x\big)^2 + \sum\big((360 + d_i) - x\big)^2 \\
&= \sum\big(x^2 - 2xb_i + b_i^2\big) + \sum((360 - c_i)^2 + 2x(360 - c_i) + x^2) + \sum((360 + d_i)^2 - 2x(360 + d_i) + x^2) \\
&= bx^2 - 2x\sum b_i + \sum b_i^2 + \sum(360 - c_i)^2 + 2x\sum(360 - c_i) + cx^2 + \sum(360 + d_i)^2 - 2x\sum(360 + d_i) + dx^2 \\
&= x^2(b + c + d) + 2x(-\sum b_i + \sum(360 - c_i) - \sum(360 + d_i)) + \sum b_i^2 - \sum(360 - c_i)^2 + \sum(360 + d_i)^2 \\
&= ax^2 + 2x(-\sum b_i - \sum c_i - \sum d_i + 360c - 360d) + \big(\sum b_i^2 - \sum(360 - c_i)^2 + \sum(360 + d_i)^2\big) \\
&= ax^2 + 2x\big(-\sum a_i + 360(c - d)\big) + \big(\sum b_i^2 - \sum(360 - c_i)^2 + \sum(360 + d_i)^2\big)
\end{aligned}
$$

$$\frac{dy}{dx} = 2ax + 2\big(-\sum a_i + 360(c - d)\big) = 2ax - 2\big(\sum a_i + 360(d - c)\big)$$

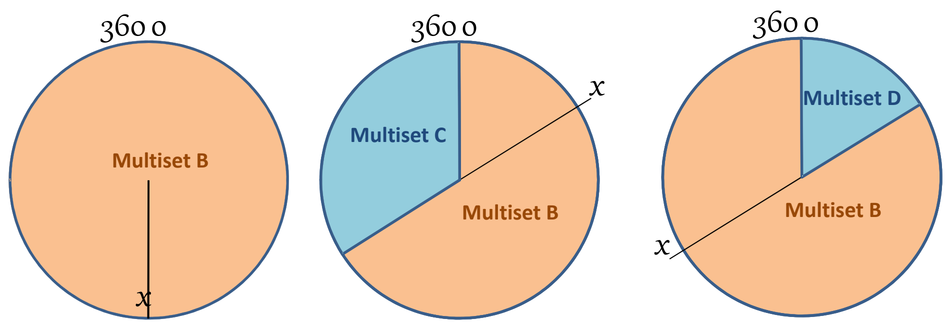
for $\frac{dy}{dx} = 0$: $x = \frac{\sum a_i + 360(d - c)}{a}$

The minimum of y may be where $\frac{dy}{dx} = 0$, or where y is not derivable.

Since in each sector $y(x)$ is a 2nd order polynomial with a positive coefficient for the x^2 term, it has a single minima, which is when $x = \frac{\sum a_i + 360(d - c)}{a}$. In the sample graphs above, it is easy to see the sectors - each sector is a trimmed 2nd order curve. The points where $y(x)$ is not derivable, is on the maximum of each sector.

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

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



If $x = 180$, all values are in multiset B.
If x is in the range $[0,180)$, multiset C range is $(x + 180{,}360)$ and multiset D is empty.
If x is in the range $(180{,}360)$, multiset D range is $[0, x - 180)$ and multiset C is empty.
Multisets C and D cannot coexist.

Here is the average calculation algorithm:

- Divide the circle into sectors, such that if x 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 $x = \frac{\sum a_i + 360(d-c)}{a}$ (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, calculate y as follows:

  (i) Assuming x=180, all values are in multiset B:
$$y = \sum_B sdist(b_i, x)^2 = \sum_A sdist(a_i, 180)^2 = \sum_A (180^2 - 360a_i + {a_i}^2)$$
$$= 32400a - 360 \sum a_i + \sum {a_i}^2$$

  (ii) Assuming x is in the range $[0,180)$, multiset C range is $(x + 180{,}360)$ and multiset D is empty:
$$y = \sum_B sdist(b_i, x)^2 + \sum_C sdist(c_i, x)^2 = \sum_B (x - b_i)^2 + \sum_C \left(360 - (c_i - x)\right)^2$$
$$= \sum_B (x^2 - 2xb_i + {b_i}^2) + \sum_C (360^2 + {c_i}^2 + x^2 - 720c_i + 720x - 2c_i x)$$
$$= bx^2 - 2x \sum b_i + \sum {b_i}^2 + 129600c + \sum {c_i}^2 + cx^2 - 720 \sum c_i + 720cx - 2x \sum c_i$$
$$= \sum {a_i}^2 + ax^2 - 2x \sum a_i - 720 \sum c_i + (129600 + 720x)c$$

  (iii) Assuming x is in the range $(180{,}360)$, multiset D range is $[0, x - 180)$ and multiset C is empty:
$$y = \sum_B sdist(b_i, x)^2 + \sum_D sdist(d_i, x)^2 = \sum_B (x - b_i)^2 + \sum_D \left(360 - (x - d_i)\right)^2$$
$$= \sum_B (x^2 - 2xb_i + {b_i}^2) + \sum_D (360^2 + {d_i}^2 + x^2 + 720d_i - 720x - 2d_i x)$$
$$= bx^2 - 2x \sum b_i + \sum {b_i}^2 + 129600d + \sum {d_i}^2 + dx^2 + 720 \sum d_i - 720dx - 2x \sum d_i$$
$$= \sum {a_i}^2 + ax^2 - 2x \sum a_i + 720 \sum d_i + (129600 - 720x)d$$

- Find the sectors(s) that has the lowest y; return their x.

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

```
// calculate average set of circular-values
// return set of average values
// T is a circular value 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 $O(a \log a)$. Given sorted input, the complexity can be $O(a)$.

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);
```

27. Weighted Average of circular values

The general definition of the weighted mean (arithmetic-average) of values $A = \{a_1, \dots, a_n\}$ with weights $W = \{w_1, \dots, w_n\}$ respectively, for any mathematical field is the value x that minimizes the expression $\sum_{i=1}^{n} w_i dist(a_i, x)^2$.
Formally: $average(A, W) \equiv \underset{x}{\operatorname{argmin}} \sum_{i=1}^{n} w_i dist(a_i, x)^2$.
w_i Is a positive real weight associated with value a_i, and $dist(v_1, v_2)$ is a function that measures the minimal distance between v_1 and v_2.

For linear values, we can calculate the average as follow:
$sdist(r1, r2) = r2 - r1$
Let y denote the expression we want to minimize:

$y = \sum w_i dist(a_i, x)^2 = \sum w_i (x - a_i)^2 = \sum (w_i x^2 - 2x w_i a_i + w_i a_i^2) = x^2 \sum w_i - 2x \sum w_i a_i + \sum w_i a_i^2$
To find the value of x that minimizes y, we need the derivative $\frac{dy}{dx}$ to be 0:

$\frac{dy}{dx} = 2x \sum w_i - 2 \sum w_i a_i$;
for $\frac{dy}{dx} = 0$: $2x \sum w_i - 2 \sum w_i a_i = 0 \rightarrow x = \frac{\sum w_i a_i}{\sum w_i}$;
We got the well-known weighted arithmetic mean formula for linear values.

For circular values, we showed that
$|sdist(c1, c2)| = min(|c2 - c1|, 360 - |c2 - c1|)$

Let y denote the expression we want to minimize:

$y = \sum w_i sdist(a_i, \mathrm{x})^2 = \sum w_i (\min(|x - a_i|,\ 360 - |x - a_i|))^2$

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

$y = \sum_A w_i sdist(a_i, x)^2 = \sum_B w_i sdist(b_i, x)^2 + \sum_C w_i sdist(c_i, x)^2 + \sum_D w_i sdist(d_i, x)^2$
$= \sum_B w_i (x - b_i)^2 + \sum_C w_i \big(360 - (c_i - x)\big)^2 + \sum_D w_i \big(360 - (x - d_i)\big)^2$
$= \sum_B w_i (x - b_i)^2 + \sum_C w_i \big((360 - c_i) + x\big)^2 + \sum_D w_i \big((360 + d_i) - x\big)^2$
$= \sum_B w_i (x^2 - 2x b_i + b_i^2) + \sum_C w_i ((360 - c_i)^2 + 2x(360 - c_i) + x^2) + \sum_D w_i ((360 + d_i)^2 - 2x(360 + d_i) + x^2)$
$= \sum_B w_i x^2 - 2x \sum_B w_i b_i + \sum_B w_i b_i^2 + \sum_C w_i (360 - c_i)^2 + 2x \sum_C w_i (360 - c_i) + \sum_C w_i x^2 + \sum_d w_i (360 + d_i)^2 -$
$2x \sum_D w_i (360 + d_i) + \sum_D w_i x^2$
$= x^2 (\sum_B w_i + \sum_C w_i + \sum_D w_i) + 2x(-\sum_B w_i b_i + \sum_C w_i (360 - c_i) - \sum_D w_i (360 + d_i)) + \sum_B w_i b_i^2 - \sum_C w_i (360 - c_i)^2 +$
$\sum_D w_i (360 + d_i)^2$
$= \sum_A w_i x^2 + 2x(-\sum_B w_i b_i - \sum_C w_i c_i - \sum_D w_i d_i + 360 \sum_C w_i - 360 \sum_D w_i) + (\sum w_i b_i^2 - \sum w_i (360 - c_i)^2 + \sum w_i (360 + d_i)^2)$
$= \sum_A w_i x^2 + 2x\big(-\sum_A w_i a_i + 360(\sum_C w_i - \sum_D w_i)\big) + (\sum w_i b_i^2 - \sum w_i (360 - c_i)^2 + \sum w_i (360 + d_i)^2)$

$\frac{dy}{dx} = 2x \sum_A w_i + 2\big(-\sum_A w_i a_i + 360(\sum_C w_i - \sum_D w_i)\big) = 2x \sum_A w_i - 2\big(\sum_A w_i a_i + 360(\sum_D w_i - \sum_C w_i)\big)$
for $\frac{dy}{dx} = 0$: $x = \frac{\sum_A w_i a_i + 360(\sum_D w_i - \sum_C w_i)}{\sum_A w_i}$

And the algorithm for weighted average is:

- Divide the circle into sectors, such that if x 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 $x = \frac{\sum_A w_i a_i + 360(\sum_D w_i - \sum_C w_i)}{\sum_A w_i}$ (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, calculate y as follows:

  (i) Assuming x=180, all values are in multiset B:

  $$y = \sum_B w_i sdist(b_i, x)^2 = \sum_A w_i sdist(180, a_i)^2 = \sum_A w_i (180^2 - 360a_i + a_i{}^2)$$
  $$= 32400 \sum_A w_i - 360 \sum_A w_i a_i + \sum_A w_i a_i{}^2$$

  (ii) Assuming x is in the range $[0,180)$, multiset C range is $(x + 180,360)$ and multiset D is empty:

  $$y = \sum_B w_i sdist(b_i, x)^2 + \sum_C w_i sdist(c_i, x)^2 = \sum_B w_i (x - b_i)^2 + \sum_C w_i \big(360 - (c_i - x)\big)^2$$
  $$= \sum_B w_i (x^2 - 2xb_i + b_i{}^2) + \sum_C w_i (360^2 + c_i{}^2 + x^2 - 720c_i + 720x - 2c_i x)$$
  $$= x^2 \sum_B w_i - 2x \sum_B w_i b_i + \sum_B w_i b_i{}^2 + 129600 \sum_C w_i + \sum_C w_i c_i{}^2 + x^2 \sum_C w_i - 720 \sum_C w_i c_i$$
  $$+ 720x \sum_C w_i - 2x \sum_C w_i c_i$$
  $$= \sum_A w_i a_i{}^2 + x^2 \sum_A w_i - 2x \sum_A w_i a_i - 720 \sum_C w_i c_i + (129600 + 720x) \sum_C w_i$$

  (iii) Assuming x is in the range $(180,360)$, multiset D range is $[0, x - 180)$ and multiset C is empty:

  $$y = \sum_B w_i sdist(b_i, x)^2 + \sum_D w_i sdist(d_i, x)^2 = \sum_B w_i (x - b_i)^2 + \sum_D w_i \big(360 - (x - d_i)\big)^2$$
  $$= \sum_B w_i (x^2 - 2xb_i + b_i{}^2) + \sum_D w_i (360^2 + d_i{}^2 + x^2 + 720d_i - 720x - 2d_i x)$$
  $$= x^2 \sum_B w_i - 2x \sum_B w_i b_i + \sum_B w_i b_i{}^2 + 129600d + \sum_D w_i d_i{}^2 + x^2 \sum_D w_i + 720 \sum_D w_i d_i$$
  $$- 720x \sum_D w_i - 2x \sum_D w_i d_i$$
  $$= \sum_A w_i a_i{}^2 + x^2 \sum_A w_i - 2x \sum_A w_i a_i + 720 \sum_D w_i d_i + (129600 - 720x) \sum_D w_i$$

- Find the sectors(s) that has the lowest y; return their x.

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

```
// calculate weighted-average set of circular-values
// return set of average values
// T is a circular value 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 $O(a \log a)$. Given sorted input, the complexity can be $O(a)$.

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);
```

28. Median of circular values

Definition 7:

Given the multiset $A = \{a_1, \dots, a_n\}$

Let sequence $S = \{s_1, \dots, s_n\}$ be the non-decreasingly sorted permutation of $\{a_1, \dots, a_n\}$

For linear values:

If n is odd:

$$median = S_{(n+1)/2}$$

If n is even:

$$median = avrg\left(S_{n/2}, S_{n/2+1}\right)$$

For circular values:

Since there is no "first" nor "last" element, we'll evaluate all "rotations" of S:

For each $k \in \{1 \dots n\}$:

If n is odd:

$$median_k = S_{1+mod\left(k+\frac{n-3}{2},n\right)}$$

If n is even:

$$median_k = \left\{avrg\left(S_{1+mod\left(k-2+\frac{n}{2},n\right)}, S_{1+mod\left(k-1+\frac{n}{2},n\right)}\right)\right\}$$ (The avrg set may contain more than one element)

The set of all candidate medians: $M(A) = \bigcup median_k$

Since an optimality property of the median is the minimization of the sum of absolute distances:

$$median(A) = \left\{\min_{m \in M(A)} \sum_A |sdist(a_i, m)|\right\}$$ (The median set may contain more than one value)

Here are some examples for circular values, given the range $[0,360)$:

$A = \{100,200,300\}$

- $median_1 = 100;\ \sum_A |sdist(a_i, 100)| = 100 + 160 = 260$
- $median_2 = 200;\ \sum_A |sdist(a_i, 200)| = 100 + 100 = 200$
- $median_3 = 300;\ \sum_A |sdist(a_i, 300)| = 160 + 100 = 260$

Therefore, $Median(A) = \{200\}$

$A = \{40,100,200,300\}$

- $median_1 = 70;\ \sum_A |sdist(a_i, 70)| = 30 + 30 + 130 + 130 = 320$
- $median_2 = 150;\ \sum_A |sdist(a_i, 150)| = 110 + 50 + 50 + 150 = 360$
- $median_3 = 250;\ \sum_A |sdist(a_i, 250)| = 150 + 150 + 50 + 50 = 400$
- $median_3 = 350;\ \sum_A |sdist(a_i, 350)| = 50 + 110 + 150 + 50 = 360$

Therefore, $Median(A) = \{70\}$

$A = \{0,90,180,270\}$

- $median_1 = 45;\ \sum_A |sdist(a_i, 45)| = 45 + 45 + 135 + 135 = 360$
- $median_2 = 135;\ \sum_A |sdist(a_i, 135)| = 135 + 45 + 45 + 135 = 360$
- $median_3 = 225;\ \sum_A |sdist(a_i, 225)| = 135 + 135 + 45 + 45 = 360$
- $median_3 = 315;\ \sum_A |sdist(a_i, 315)| = 45 + 135 + 135 + 45 = 360$

Therefore, $Median(A) = \{45,135,225,315\}$

$A = \{0,180,270,271\}$
(note that the average of $\{0,180\}$ is $\{90,270\}$)

- $median_{1a} = 90;\ \sum_A |sdist(a_i, 90)| = 90 + 90 + 180 + 179 = 539$
- $median_{1b} = 270;\ \sum_A |sdist(a_i, 270)| = 90 + 90 + 0 + 1 = 181$
- $median_2 = 225;\ \sum_A |sdist(a_i, 225)| = 135 + 45 + 45 + 46 = 271$
- $median_3 = 270.5;\ \sum_A |sdist(a_i, 270.5)| = 89.5 + 90.5 + 0.5 + 0.5 = 181$
- $median_4 = 315.5;\ \sum_A |sdist(a_i, 315.5)| = 44.5 + 135.5 + 45.5 + 44.5 = 270$

Therefore, $Median(A) = \{270, 270.5\}$

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

```
// calculate median set of circular-values
// return set of median values
// T is a circular value 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 values
    {
        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 values
            B.insert(((double)S[m] + d/2.));
            if (d == -CircVal<T>::GetR()/2.)
                B.insert(((double)S[n] + d/2.));
        }
    }
    else                          // odd number of values
        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);
```

29. Circular parameter estimation based on noisy measurements

For linear values:

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 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 value:

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 $\mathrm{atan2}(\sum \sin(a_n), \sum \cos(a_n))$.
   - 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.

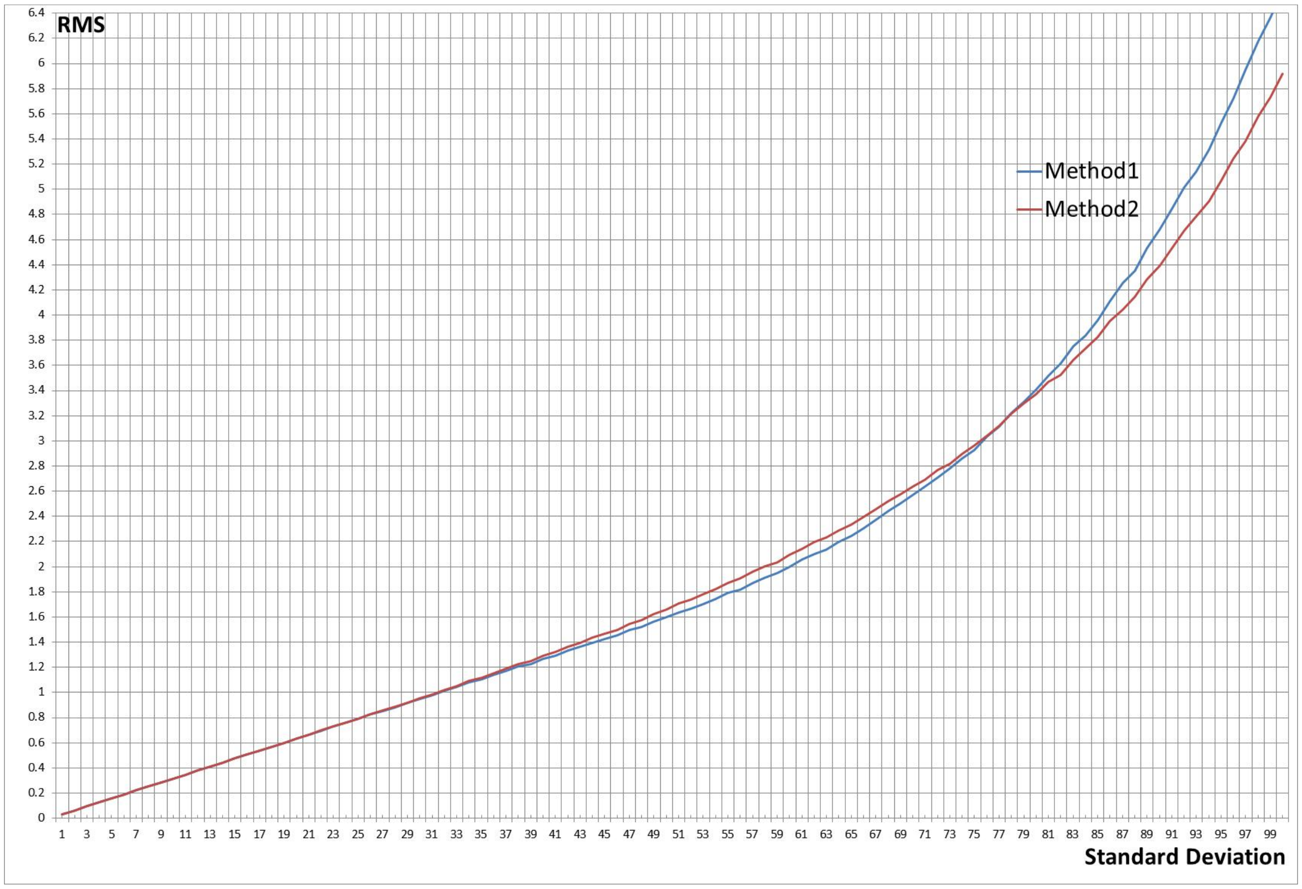
We will repeat the test for different standard deviations – between 1° and 100°.

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
} );
```

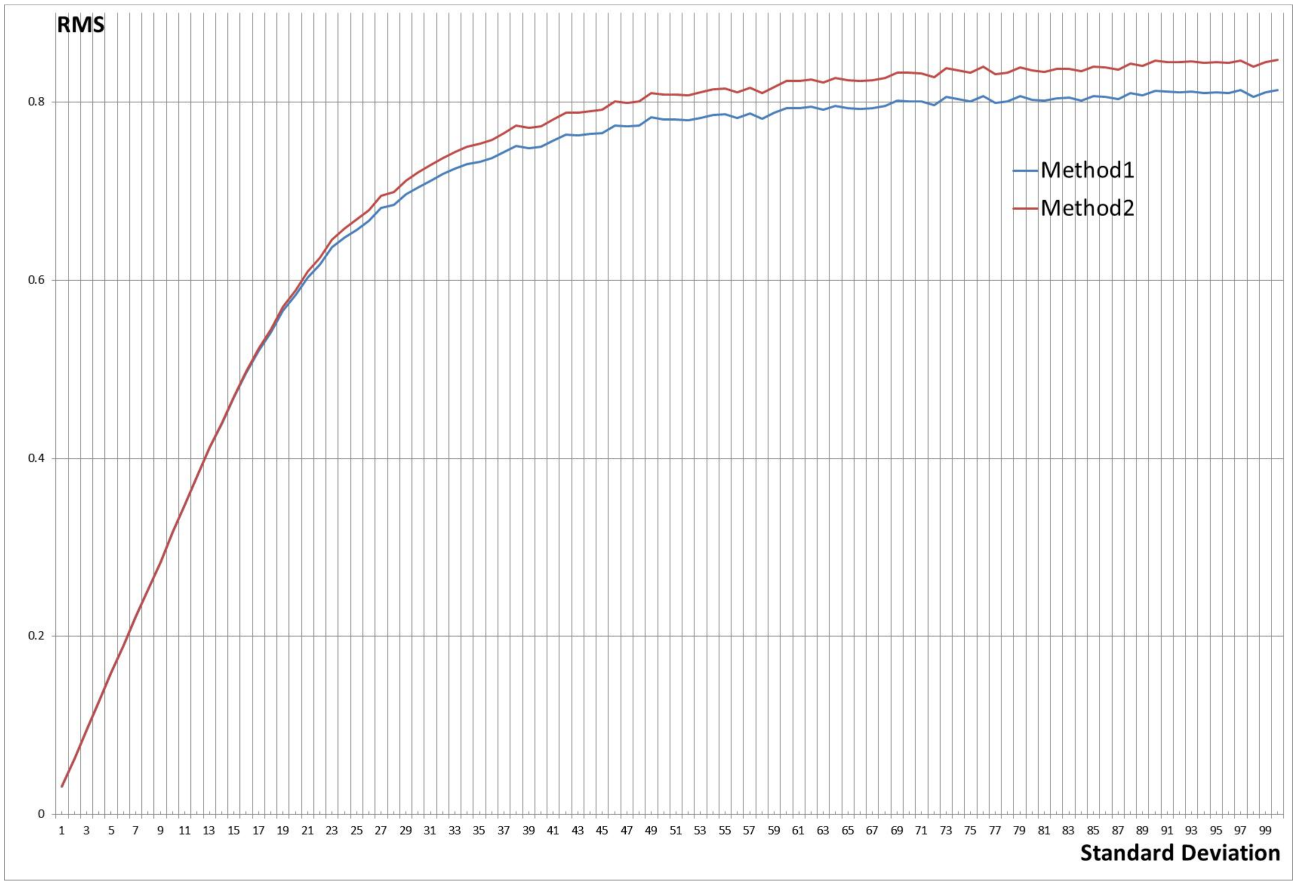
Note that parallel_for is used for parallelizing tests for different standard deviations.

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 value of standard deviation.

30. Interpolation and average estimation of sampled continuous-time circular signal

Given a sequence of samples $y_1, \dots, y_N$ of a continuous-time circular signal, sampled at times $\mathrm{t}_1, \dots, \mathrm{t}_\mathrm{N}$ respectively, where the measurement technique/instrument is accurate - Estimate the mean value of the signal at period $[\mathrm{t}_1, \mathrm{t}_\mathrm{N}]$.

Linear example:

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

Circular example:

- Given airplane heading measured each second $[deg]$ over a period of one hour - Estimate the mean heading

For circular values, it is not obvious in which "direction" the signal changed 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 insert an error into 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 $t_N - t_1$ (average).

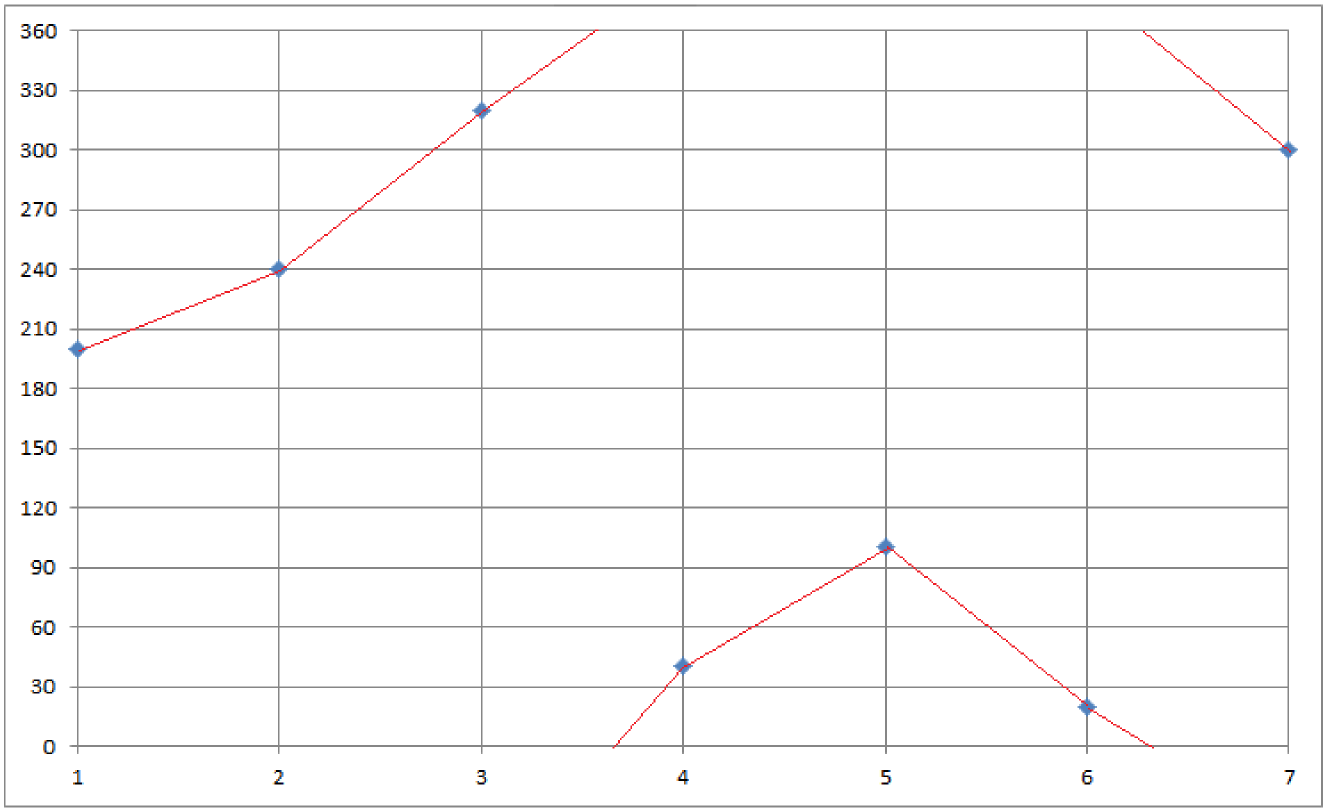
Linear Interpolation

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

For linear values, the average value of a signal, based on linear interpolation, is the weighted average of $\left(y_{k-1} + \frac{sdist(y_{k-1}, y_k)}{2}\right)$ with weight $(t_k - t_{k-1})$, for each k in the range $[2, n]$,
where, as defined previously, $sdist(r1, r2) \equiv r2 - r1$,

For circular values, it is the weighted circular average of
$wrap\left(y_{k-1} + \frac{sdist(y_{k-1}, y_k)}{2}\right)$ with weight $(t_k - t_{k-1})$, for each k in the range $[2, n]$,
where, as defined previously, $sdist(c1, c2) \equiv mod\left(c2 - c1 + \frac{h-l}{2}, h - l\right) - \frac{h-l}{2}$

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 average circular value, and the weight of each interval k in the range $[2, n]$:

  $IntervalAvrg = Wrap\left(x_{k-1} + \frac{sdist(y_{k-1}, y_k)}{2}\right)$

  $IntervalWeight = t_k - t_{k-1}$

  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:

```cpp
// estimate the average of a sampled continuous-time circular signal, using circular linear interpolation
// T is a circular value 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:

```cpp
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);
```