5 Assignment 3 [Assignment ID: `cpp_arithmetic`]

5.1 Preamble (Please Read Carefully)

Before starting work on this assignment, it is **critically important** that you **carefully** read Section 1 (titled "General Information") which starts on page 1-1 of this document.

5.2 Topics Covered

This assignment covers material primarily related to the following: interval arithmetic, robust geometric predicates, Delaunay triangulations, exceptions.

5.3 Problems — Part A — Nonprogramming Exercises

- 6.1 [exception safety]
- 6.2 [function calls and exceptions]
- 6.6 a b [stack unwinding]
- 6.8 a b c [exception safety]

5.4 Problems — Part B — Robust Geometric Predicates

B.1 *Interval class template for interval arithmetic (*`interval`*).* In this exercise, an interval class template called `interval` will be developed for performing interval arithmetic. Interval arithmetic is useful in many applications where error bounds must be maintained on numerical calculations.

Some interval operations can yield in an indeterminate (i.e., uncertain) result. If such a situation occurs an exception of type `indeterminate_result` should be thrown. This exception type must be defined as follows:

```
namespace ra::math {
    struct indeterminate_result : public std::runtime_error
    {
        using std::runtime_error::runtime_error;
    };
}
```

The `interval` class template has a single template parameter `T`, which is the floating-point type (i.e., **`float`**, **`double`**, or **`long double`**) used to represent each of the lower and upper bounds of an interval. The `interval` class template (with template parameter `T`) should provide the following interface (i.e., public members):

(a) `real_type`. The real number type used to represent each of the lower and upper bounds of the interval. This is simply an alias for the template parameter `T`.

(b) `statistics`. This is a type used to represent various statistics related to the `interval` class template. This type must be defined exactly as follows:

```
struct statistics {
    // The total number of indeterminate results encountered.
    unsigned long indeterminate_result_count;
    // The total number of interval arithmetic operations.
    unsigned long arithmetic_op_count;
};
```

(c) default constructor. This constructor has one parameter of type `real_type`, which has a default value of `real_type(0)`. This constructor should create an interval consisting the single real value given by the constructor parameter (i.e., each of the lower and upper bounds are set to the specified value).

(d) copy constructor and copy assignment operator (which may be compiler-provided defaults if appropriate).

(e) move constructor and move assignment operator (which may be compiler-provided defaults if appropriate).

(f) two-argument constructor. This constructor has two parameters of type `real_type`, which correspond to the lower and upper bounds (in that order) of the interval to be created. The lower bound for the interval must be less than or equal to the upper bound. (If this precondition is violated, the behavior of this constructor is implementation defined.)
(g) destructor (which may be the compiler-provided default if appropriate).
(h) compound assignment operators for addition of, subtraction of, and multiplication by another `interval`. (i.e., **`operator`**`+=`, **`operator`**`-=`, and **`operator`**`*=`). These operators should have the usual semantics for compound-assignment operators. Each of these operators must increment the arithmetic-operation count by exactly one.
(i) `lower`. This (non-static) member function has no parameters and a return type `real_type`. This function returns the lower bound of the interval.
(j) `upper`. This (non-static) member function has no parameters and a return type `real_type`. This function returns the upper bound of the interval.
(k) `is_singleton`. This (non-static) member function has no parameters and a return type **`bool`**. This function is a predicate that tests if an interval contains exactly one real number (i.e., the lower and upper bounds of the interval are equal).
(l) `sign`. This (non-static) member function has no parameters and a return type **`int`**. This function returns the sign of the interval. If all elements in the interval are strictly negative, `-1` is returned. If all elements in the interval are strictly positive, `1` is returned. If the interval consists of only the element zero, `0` is returned. Otherwise, the result is indeterminate and an `indeterminate_result` exception should be thrown.
(m) `clear_statistics`. This static member function takes no parameters and has a return type of **`void`**. The function clears the current statistics for the `interval` class. In particular, all statistics are set to zero.
(n) `get_statistics` This static member function has a single parameter of type `statistics&` and a return type of **`void`**. The function retrieves the current statistics for the `interval` class, passing them back through the single reference parameter.

A number of non-member helper functions are also provided as follows:

(a) binary addition, subtraction, and multiplication operators for `interval` objects (i.e., binary **`operator`**`+`, **`operator`**`-`, and **`operator`**`*`). These operators should be overloaded to allow for the addition, subtraction, and multiplication of two intervals. These operations must take their parameters by reference. Each of these operators must increment the arithmetic-operation count by exactly one.
(b) less-than operator (i.e., **`operator`**`<`). This operator takes two `interval` objects by reference and tests if the value of the first is less than the value of the second. The return type is **`bool`**. For two intervals a and b, the condition $a < b$ is true if every element in a is less than every element in b, false if every element in a is greater than or equal to (i.e., not less than) every element in b, and indeterminate otherwise. If the result is indeterminate, an exception of type `indeterminate_result` is thrown.
(c) stream inserter (i.e., **`operator`**`<<`). A stream inserter should be provided to allow `interval` objects to be written to a stream (i.e., `std::ostream`). This function should output the following items in order with no leading or trailing characters (including newlines):
    i. a left square bracket character;
    ii. the lower interval bound (formatted as a real number);
    iii. a comma character;
    iv. the upper interval bound (formatted as a real number); and
    v. a right square bracket.

It is an invariant of the `interval` class that the lower bound of the interval is always less than or equal to the upper bound.

The source code for the `interval` class and its associated non-member functions should be placed in the file `include/ra/interval.hpp`. Note that all identifiers for the `interval` class template and its helper code (including the `indeterminate_result` class) must be placed in the namespace `ra::math`.

The `interval` class template and its helper code must always preserve the user's rounding mode. That is, if the rounding mode is changed inside the code associated with the `interval` class, the original rounding mode must be restored before leaving this code. This is a perfect candidate for a RAII class, which is explained in more detail in Section 5.6.

The arithmetic-operation count statistic (i.e., `arithmetic_op_count`) counts **interval** operations (not floating-point operations). Each addition, subtraction, and multiplication operation performed for intervals (including those associated with compound-assignment operators) should increment the arithmetic-operation count statistic (i.e., `arithmetic_op_count`) by exactly one. Note that a single arithmetic operation must not be counted more than once. For example, if **operator**+ is implemented in terms of **operator**+=, invoking **operator**+ must only increment the arithmetic-operation count by one, not two. Each operation that yields an indeterminate result must increment the indeterminate result (i.e., `indeterminate_result`) statistic. Since an indeterminate result always causes an `indeterminate_result` exception to be thrown, the indeterminate result statistic also corresponds to the number of `indeterminate_result` exceptions thrown.

Note that division is not supported by the `interval` class template. Also, note that no stream extractor need be provided for this class template.

The code used to test the `interval` class template should be placed in a file called `app/test_interval.cpp`.

Note that, as of the time of this writing, neither GCC nor Clang support the "`STDC FENV_ACCESS ON`" pragma. In the case of GCC, a workaround for this problem is to add the "`-frounding-math`" flag to the compiler when compiling any code that explicitly changes rounding modes (e.g., any code that uses interval arithmetic). This should be done by adding the preceding flag to the `CMAKE_CXX_FLAGS` variable.

B.2 *Geometry kernel class template with robust geometric predicates (*`Kernel`*).* In this exercise, the code for a geometry kernel with robust geometric predicates is developed. This code consists of a class template called `Kernel`.

The functionality of the geometry kernel is encapsulated by a class template called `Kernel`. The `Kernel` class template has one template parameter `R`, which is the real-number type to be used (e.g., for coordinates of points, components of vectors, and so on). The `Kernel` class template has the interface shown in Listing 7. Objects of the `Kernel` type should be stateless (i.e., have no non-static data members). (Private type and function members may be added as well as static data members, however.)

Listing 7: Interface for the `Kernel` class template

```
namespace ra::geometry {

	// A geometry kernel with robust predicates.
	template <class R>
	class Kernel
	{
	public:

		// The type used to represent real numbers.
		using Real = R;

		// The type used to represent points in two dimensions.
		using Point = typename CGAL::Cartesian<R>::Point_2;

		// The type used to represent vectors in two dimensions.
		using Vector = typename CGAL::Cartesian<R>::Vector_2;

		// The possible outcomes of an orientation test.
		enum class Orientation : int {
			right_turn = -1,
			collinear = 0,
			left_turn = 1,
		};

```

```
    // The possible outcomes of an oriented-side-of test.
    enum class Oriented_side : int {
        on_negative_side = -1,
        on_boundary = 0,
        on_positive_side = 1,
    };

    // The set of statistics maintained by the kernel.
    struct Statistics {
        // The total number of orientation tests.
        std::size_t orientation_total_count;
        // The number of orientation tests requiring exact
        // arithmetic.
        std::size_t orientation_exact_count;
        // The total number of preferred-direction tests.
        std::size_t preferred_direction_total_count;
        // The number of preferred-direction tests requiring
        // exact arithmetic.
        std::size_t preferred_direction_exact_count;
        // The total number of side-of-oriented-circle tests.
        std::size_t side_of_oriented_circle_total_count;
        // The number of side-of-oriented-circle tests
        // requiring exact arithmetic.
        std::size_t side_of_oriented_circle_exact_count;
    };

    // Since a kernel object is stateless, construction and
    // destruction are trivial.
    Kernel();
    ~Kernel();

    // The kernel type is both movable and copyable.
    // Since a kernel object is stateless, a copy/move operation
    // is trivial.
    Kernel(const Kernel&);
    Kernel& operator=(const Kernel&);
    Kernel(Kernel&&);
    Kernel& operator=(Kernel&&);

    // Determines how the point c is positioned relative to the
    // directed line through the points a and b (in that order).
    // Precondition: The points a and b have distinct values.
    Orientation orientation(const Point& a, const Point& b,
      const Point& c);

    // Determines how the point d is positioned relative to the
    // oriented circle passing through the points a, b, and c
    // (in that order).
    // Precondition: The points a, b, and c are not collinear.
    Oriented_side side_of_oriented_circle(const Point& a,
      const Point& b, const Point& c, const Point& d);

    // Determines if, compared to the orientation of line
    // segment cd, the orientation of the line segment ab is
    // more close, equally close, or less close to the
    // orientation of the vector v.
    // The value returned is 1, 0, or -1 if, compared to the
```

```
        // orientation of cd, the orientation of ab is more close,
        // equally close, or less close to the orientation of v,
        // respectively.
        // Precondition: The points a and b have distinct values; the
        // points c and d have distinct values; the vector v is not
        // the zero vector.
        int preferred_direction(const Point& a, const Point& b,
          const Point& c, const Point& d, const Vector& v);

        // Tests if the quadrilateral with vertices a, b, c, and d
        // specified in CCW order is strictly convex.
        // Precondition: The vertices a, b, c, and d have distinct
        // values and are specified in CCW order.
        bool is_strictly_convex_quad(const Point& a, const Point& b,
          const Point& c, const Point& d);

        // Tests if the flippable edge, with endpoints a and c and
        // two incident faces abc and acd, is locally Delaunay.
        // Precondition: The points a, b, c, and d have distinct
        // values; the quadrilateral abcd must be strictly convex.
        bool is_locally_delaunay_edge(const Point& a, const Point& b,
          const Point& c, const Point& d);

        // Tests if the flippable edge, with endpoints a and c and
        // two incident faces abc and acd, has the preferred-directions
        // locally-Delaunay property with respect to the first and
        // second directions u and v.
        // Precondition: The points a, b, c, and d have distinct values;
        // the vectors u and v are not zero vectors; the vectors u and
        // v are neither parallel nor orthogonal.
        bool is_locally_pd_delaunay_edge(const Point& a,
          const Point& b, const Point& c, const Point& d,
          const Vector& u, const Vector& v);

        // Clear (i.e., set to zero) all kernel statistics.
        static void clear_statistics();

        // Get the current values of the kernel statistics.
        static void get_statistics(Statistics& statistics);

    };
}
```

(Note that the `Kernel` class template and any supporting code should be placed in the namespace `ra::geometry`.)

All of the geometric predicates must be numerically robust. That is, they must always yield the correct result in spite of effects such as roundoff error. The first attempt to determine the predicate result should be done by using interval arithmetic. Then, only if interval arithmetic fails to produce a conclusive result, should the implementation resort to using exact arithmetic. For interval arithmetic, the `ra::math::interval` class developed in Exercise B.1 must be used. For exact arithmetic, the `CGAL::MP_Float` type should be used.

The `orientation` function performs a two-dimensional orientation test. Given three points a, b, and c in $\mathbb{R}^2$, the function determines to which side of the directed line passing through a and b (in that order) the point c lies (i.e., left of, right of, or collinear with). This result should be determined by testing the sign of the determinant of a 2×2 matrix. For more details on the algorithm to be used, refer to the lecture slides. Each time the `orientation` function is invoked, the `orientation_total_count` statistic should be incremented. If interval arithmetic fails to produce a conclusive result so that exact arithmetic must be used, the `orientation_exact_count` statistic should also be incremented.

The side_of_oriented_circle function performs a side-of-oriented-circle test. Given four points a, b, c, and d in $\mathbb{R}^2$, the function determines to which side of the oriented circle passing through the points a, b, and c (in that order) the point d lies (i.e., left of, right of, or on the boundary of). The left-of and right-of cases correspond to the positive and negative sides, respectively. The points a, b, and c are specified in CCW order (i.e., the circle has a CCW orientation). This result should be determined by testing the sign of the determinant of a 3×3 matrix. For more details on the algorithm to be used, refer to the lecture slides. Each time the side_of_oriented_circle function is invoked, the side_of_oriented_circle_total_count statistic should be incremented. If interval arithmetic fails to produce a conclusive result so that exact arithmetic must be used, the side_of_oriented_circle_exact_count statistic should also be incremented.

The preferred_direction function performs a preferred-directions test. Given four points a, b, c, and d in $\mathbb{R}^2$ and a vector v in $\mathbb{R}^2$, the function determines if the orientation of the line segment ab is more close, equally close, or less close than the orientation of the line segment cd to the preferred direction specified by v. This result should be determined using the algorithm specified in the lecture slides. Each time the preferred_direction function is invoked, the preferred_direction_total_count statistic should be incremented. If interval arithmetic fails to produce a conclusive results requiring exact arithmetic to be used, the preferred_direction_exact_count statistic should also be incremented.

The is_strictly_convex_quad function tests if a quadrilateral is strictly convex. Given a quadrilateral Q with vertices a, b, c, and d in $\mathbb{R}^2$ specified in CCW order, the function determines if Q is strictly convex. This result should be determined by applying four (2-dimensional) orientation tests using only the orientation function. In particular, if a quadrilateral Q is strictly convex, then when moving from one vertex of Q to the next in CCW order, one will encounter only left turns. For more details on the algorithm involved, refer to the lecture slides.

The is_locally_delaunay_edge function tests if a flippable edge in a triangulation is locally Delaunay. Given a flippable edge e (in a triangulation) with the endpoints a and c and two incident faces acd and abc (whose vertices are specified in CCW order), this function tests if e is locally Delaunay. This result should be computed using only the side_of_oriented_circle function. For more details on the algorithm involved, refer to the lecture slides. With regard to the locally-Delaunay condition, a few additional comments are worth making. Applying an edge flip to e (in the triangulation) would yield the new edge e' with endpoints b and d. It is always that case that **at least one** of e and e' must be locally Delaunay. If a, b, c, and d are co-circular (i.e., all four points lie on the same circle), however, **both** of e and e' will be locally Delaunay.

The is_locally_pd_delaunay_edge function tests if a flippable edge in a triangulation has the preferred-directions locally-Delaunay property. Suppose that we are given a flippable edge e (in a triangulation) with the endpoints a and c and two incident faces acd and abc (whose vertices are specified in CCW order). Let e' denote the edge that would be obtained by applying an edge flip to e (i.e., e' has the endpoints b and d and two incident faces abd and bcd). The edge e has the preferred-directions locally-Delaunay property if: 1) e is locally Delaunay and e' is not; or 2) e and e' are both locally Delaunay and e is preferred over e' in terms of the preferred directions u and v. It always the case that **exactly one** of e and e' has the preferred-directions locally-Delaunay property. This result should be determined by using only the predicates side_of_oriented_circle and preferred_direction. For more details on the algorithm involved, refer to the lecture slides.

The source for the Kernel class template (and any supporting code) should be placed in the file include/ra/kernel.hpp. The Kernel class template must work for at least the cases where the template parameter R is chosen as **float** and **double**.

A program called test_kernel should be provided to test the Kernel class template. The source for this program should be placed in a file called app/test_kernel.cpp.

5.5 Problems — Part C — Delaunay Triangulation

C.1 *Delaunay triangulation program (*delaunay_triangulation*).* In this exercise, a program called delaunay_triangulation is developed that computes the preferred-directions Delaunay triangulation of a set P of points, starting from an arbitrary triangulation of P. This is achieved by starting with the given triangulation of P and then applying the Lawson local optimization procedure (LOP) until the preferred-directions Delaunay triangulation is obtained.

The program should read a triangulation from standard input in OFF format. Then, the LOP should be applied to the triangulation to obtain the preferred-directions Delaunay triangulation. The first and second preferred directions for the preferred-directions test should be chosen as $(1,0)$ and $(1,1)$, respectively. Finally, the resulting triangulation should be written to standard output in OFF format.

The LOP algorithm for computing the preferred-directions Delaunay triangulation works as follows. While the triangulation has a flippable edge e that does not have the preferred-directions locally-Delaunay property, apply an edge flip to e. This algorithm terminates when every flippable edge in the triangulation has the preferred-directions locally-Delaunay property. For more details on the algorithm, refer to the lecture slides.

The code must be robust to roundoff error. To achieve this goal, use the `Kernel` class template developed in Exercise B.2.

To reduce the amount of work required in this exercise, a class is provided for representing triangulations. This class is provided in the file `triangulation_2.hpp`. This file must be placed in the `app` directory (without changing the name of the file). The contents of this file must not be modified. The `Triangulation_2` class provides functionality such as the ability to:

- read a triangulation from an input stream;
- write a triangulation to an output stream; and
- perform an edge flip.

The source for the `delaunay_triangulation` program should be placed in the file `app/delaunay_triangulation.cpp`.

5.6 RAII Classes

When an exception is thrown, the normal flow of execution in the code ceases, and control is transferred to the corresponding catch block (which handles the exception) via the stack unwinding process. As each block is exited during the stack unwinding process, the only code that is executed is destructors. (In particular, a destructor is invoked for each variable whose lifetime ends when a block is exited.) This implies that, in code that throws exceptions, **all cleanup must be performed in destructors**. In the interval arithmetic part of this assignment, a critically important type of cleanup is the restoring of the rounding mode. When an exception is thrown, the only way that the rounding mode can be restored is if the code that does this is placed in a destructor. This problem is easily addressed with what is known as an RAII class. An RAII class is simply a class whose sole reason to exist is to perform some kind of cleanup in its destructor.

In what follows, a basic sketch of an RAII class called `rounding_mode_saver` that can be used to restore the rounding mode is described. The class is very simple. It is neither movable nor copyable. It provides a default constructor and destructor and has a single private data member. The default constructor saves the current rounding mode into a private data member. The destructor simply sets the rounding mode to the value that was saved (in the data member) at construction time. In other words, the interface of this class would resemble the following:

```
class rounding_mode_saver {
public:
    // Save the rounding mode.
    rounding_mode_saver();

    // Restore the rounding mode to the value that was saved at
    // the time of construction.
    ~rounding_mode_saver();

    // The type is neither movable nor copyable.
    rounding_mode_saver(rounding_mode_saver&&) = delete;
    rounding_mode_saver(const rounding_mode_saver&) = delete;
    rounding_mode_saver& operator=(rounding_mode_saver&&) = delete;
```

```
    rounding_mode_saver& operator=(const rounding_mode_saver&) = delete;
};
```

A typical usage of the above class would look something like:

```
void some_function_that_changes_rounding_modes()
{
    // Save the rounding mode that should be restored
    // when the enclosing block is exited.
    rounding_mode_saver rms;

    // Set the rounding mode to the desired value.

    // Perform some computation which may throw an exception.

    // Set the rounding mode to another desired value.

    // Perform some computation which may throw an exception.

    // Set the rounding mode to another desired value.

    // Perform some computation which may throw an exception.

} // When this block is exited, rms is destroyed.
// Destroying rms restores the rounding mode that was saved when
// rms was constructed.
```