Big Rational Arithmetic and Robust Geometric Primitives

Spatial data is often large and default implementations of datatypes like Integer, Float and Double in C++ aren't equipped to deal with data in this format, since it frequently exceeds their bounds of computation. A new data type is thus required to support computations on spatial data which sometimes needs to be accurate to a precision level which might not be supported by already available datatypes.

'BigRational' will allow us to store and perform operations on the type of spatial data that gives trouble to the default implementations. Such an implementation requires a robust arithmetic operations to be defined, like the mathematical operators "+, - , /, * and %". In addition to these, I/O operations (input and output) for a BigRational number are also required.

Datatypes and Operations: (Big Rational Arithmetic)

The following classes, datatypes and operations will be present in the interface:

Datatypes and Classes

<u>BigInteger</u>: This datatype will be used to represent integers of dynamic length. The length of the integer is defined by the operation that is used to generate that "Big Number.

<u>BigRational</u>: This datatype will be used to represent rational numbers. The BigRational will be of the form p/q which will both be BigIntegers

INTERFACES:

Mathematical Operations:

<u>Addition '+'</u>: A binary operator that performs the addition operation on any of the given 'Big' input types.

<u>Multiplication '*'</u>: A binary operator that performs the multiplication operation on any of the given 'Big' input types.

<u>Subtraction '-'</u>: A binary operator that performs the subtraction operation on any of the given 'Big' input types.

<u>Division '/'</u>: A binary operator that performs the division operation on any of the given 'Big' input types. The return will be a BigRational number.

Mod ‘%’: A binary operator that performs the modular operation on BigInteger types returning a BigInteger.

GCD(BigInteger,BigInteger): Return the greatest common divisor in BigInteger.

LCM(BigInteger, BigInteger): Returns the LCM of two BigIntegers in BigInteger form.

All these operations need to work within different “Big…” types.

Comparison

Greater and Smaller than ‘>’, ‘<’, ‘>=’, ‘<=’: To check if one big number (bigInteger or bigRational) is greater than or smaller than the other.

Equal to and not equal to ‘==’ ‘!=’: To check if one big number (BigRational or BigInteger) is equal to the other or not.

Conversion:

toBigRational: to convert a BigInteger to BigRational for internal computation.

I/O Operations

Obtaining user input: We will have to define an operation that obtains the user input and translates it into a representation that can then be used to store it in memory.

Displaying numbers of that datatype: This would require an operation that translates the representation of the number in memory into a format that can be displayed.

Datatypes and Operations: (Geometric Primitives)

Using the Big Rational Arithmetic developed in the first part of the project, the second part will use the developed datatypes to create some geometric primitives.

Datatypes and Classes:

PoiR2D: A point initialized with and x and y coordinate both of which are BigRationals.

Seg2D: A segment returned from 2 points which can be PoiF2D, with 2 half segments.

HalfSegment: A half segment from a segment with a left/right value and a dominating point.

AttrHalfSegment: An attribute half segment is initialized using a half segment and a Boolean value, which indicates whether the region lies inside or outside

MinBoundingRec: Defines the minimum Bounding rectangle of a segment

INTERFACES:

PoiR2D(BigRational, BigRational): Returns a 2D point with Big Rational x and y coordinates

Seg2D(<PoiR2D/Poi2D>,<PoiR2D/Poi2D>): Returns a segment with 2 points which can be BigRationals or BigIntegers. The constructor will also create the respective half segments.

AttrHalfSegment(bool value, bool direction, bool dominatingPoint): returns an attributed Half Segment.

MinBoungingRec(Seg2D): Takes a segment as an argument and returns the minimum bounding rectangle defined by that segment.

Operations:

ReturnHalfSegment(Seg2D, Boolean): Based on the Boolean value identified as left or right, a half segment is returned.

HalfSegment.Left(HalfSegment): Returns true if one half segment is left to the other.

HalfSegment.Right(HalfSegment): Returns true if one half segment is right to the other.

Comparison: <,>,==,!=, >=, <=: Comparison between two poi2D's or poiR2D's.

PoiR2D.LiesOnSegment(Seg2D): tests if a point lies on a segment or not

Seg2D.Intersects(Seg2D): test if one segment intersects the other

Seg2D.IntersectionPoint(Seg2D): Takes two segments as input and returns the intersection point if the segments intersect

Seg2D.Meet(Seg2D): Determines if a segment meets the other.

Seg2D.MeetingPoint(Seg2D): Determines if two segments intersect, and if they do, return the meeting point of the 2 segments.