Boost.Geometry: A C++ library for geometric computations

Vissarion Fisikopoulos

Athens C++ Meetup

18 Sept. 2025

Outline

Introduction

Boost Geometry design

Using Boost Geometry

Spatial computations

Boost.Geometry

- Open source
- Part of Boost C++ Libraries
- Header-only
- Stable interface
- C++14 support
- Static polymorphism, Metaprogramming, Tag dispatching
- Primitives, Algorithms, Spatial Index
- Standards conformant (OGC Simple Feature Access)

boost GEOMETRY

Open Geospatial Consortium

Documentation and Resources

- `https://www.boost.org/libs/geometry`
- Mailing list: `lists.boost.org/geometry`
- GitHub: `https://github.com/boostorg/geometry`

Main contributors:

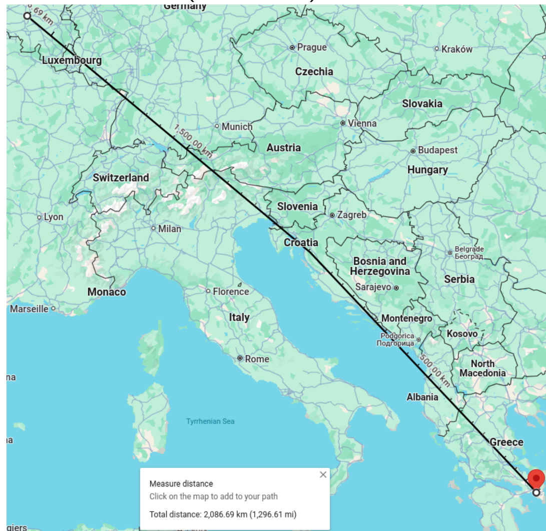
- Barend Gehrels
- Bruno Lalande
- Mateusz Loskot
- Adam Wulkiewicz
- Menelaos Karavelas
- Vissarion Fisikopoulos
- Tinko Bartels
- and several other contributors

Hello World!

```
#include <boost/geometry.hpp>
#include <iostream>
namespace bg = boost::geometry;

int main() {
  using point = bg::model::point
        <
          double,
          2,
          bg::cs::geographic<bg::degree>
        >;
  // Athens -> Brussels
  std::cout << bg::distance(point(23.727539, 37.983810),
                            point(4.350000, 50.833333));
}
```

Result

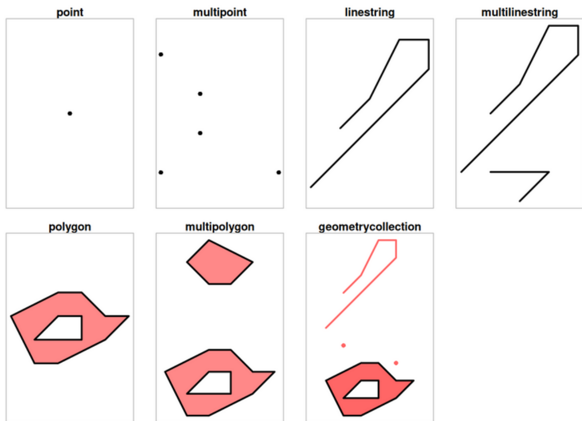
2.09071e+06 (in meters)



google maps

Primitives (Geometries)

- Point, MultiPoint
- Segment, Linestring, MultiLinestring
- Ring, Polygon, MultiPolygon
- Box
- GeometryCollection



Algorithms

Algorithms

Geometry Constructors

make
make_inverse
make_zero

Predicates

crosses
covered_by
disjoint
equals
intersects
is_empty
is_simple
is_valid
overlaps
touches
within

Append

append

Area

area

Assign

assign
assign_inverse
assign_zero
assign_points
assign_values (2 3 4 coordinate values)

Azimuth

azimuth

Buffer

buffer

Centroid

centroid

Clear

clear

Closest Points

closest_points

Convert

convert

Convex Hull

convex_hull

Correct

correct

Densify

densify

Distance

distance

Difference

difference
sym_difference

Envelope

envelope

Expand

expand

For Each

for each (point, segment)

Intersection

intersection

Length

length

Line Interpolate

line_interpolate

Num_ (counting)

num_interior_rings
num_geometries
num_points
num_segments

Perimeter

perimeter

Relate

relate
relation

Reverse

reverse

Similarity

discrete_frechet_d
discrete_hausdorff

Simplify

simplify

Transform

transform

Union

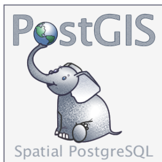
union

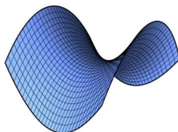
Unique

unique

Applications

- GIS, spatial databases
- Robotics and autonomous systems
- Computer graphics and gaming
- Computer vision and image analysis



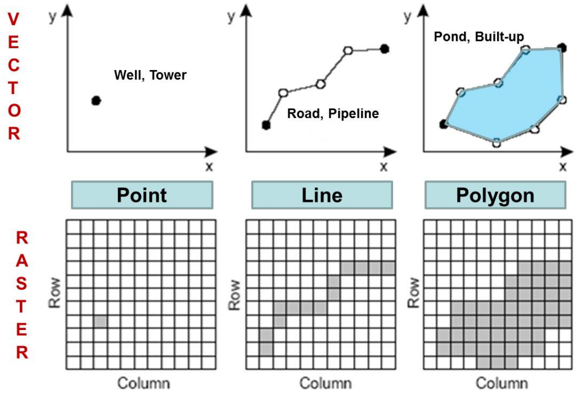


What is GIS?

- Geographic Information System (GIS)
- Capture, store, manipulate, analyze, and visualize spatial data
- Examples: navigation apps, urban planning, disaster response

Vector and Raster Data

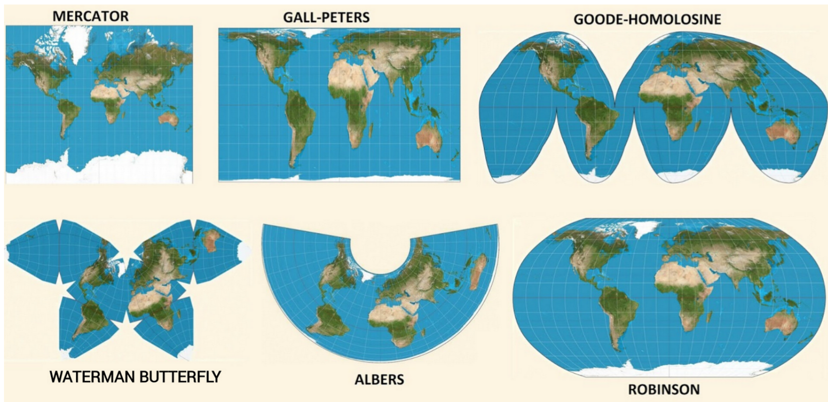
- Vector data: Points, Lines, Polygons
- Raster data: grid of pixels (e.g., satellite images, DEMs)
- Vector good for discrete features; raster for continuous phenomena



Vector and Raster Models

Map Projections

- Earth's surface is curved; maps are flat
- Projection = method to transform coordinates to 2D
- Common types: Mercator, Lambert, UTM
- Tradeoffs: area vs shape vs distance



Spatial Databases

Extend traditional db with the ability to store, index, and query spatial data such as geometries (points, lines, polygons) and rasters.

Core Features:

- Support for spatial data types: `POINT`, `LINESTRING`, `POLYGON`, `MULTIPOLYGON`
- Advanced spatial indexing: R-trees, QuadTrees
- Efficient spatial queries: containment, intersection, proximity, NN
- Built-in geometry functions: `ST_Intersects`, `ST_Contains`, `ST_Distance`, `ST_Union`, etc.

Popular Implementations:

- **PostGIS** (PostgreSQL): most feature-rich, supports topology, raster, 3D/4D, CRS transforms
- **MySQL Spatial**: since v5.7, native support, R-tree on InnoDB
- **SpatiaLite**: spatial extension for lightweight SQLite setups

Outline

Introduction

Boost Geometry design

Using Boost Geometry

Spatial computations

Three-layer design

- **Algorithms layer**: Namespace `boost::geometry` constains user-facing API (e.g. `boost::geometry::distance`).
- **Dispatch layer**: Namespace `boost::geometry::dispatch` contains compile-time selection using traits, tags, concepts.
- **Implementation layer**: Namespace `boost::geometry::detail` contains implementation details.

Algorithms

- Free functions (e.g., `distance`, `area`, `buffer`).
- Input: geometries, strategies (optional)
- Algorithms are independent of geometry representation
- Work across cartesian, spherical, geographic coordinate systems.
- Internally resolved via dispatching and strategy selection.

Strategies

- Encapsulate mathematical models and formulas.
- Examples: Vincenty, Andoyer, Thomas for geographic distance.
- Strategies chosen explicitly by the user or defaulted by the library.
- Provide accuracy/performance flexibility without changing algorithm signatures.

Traits

Traits connect user-defined geometry types with Boost.Geometry algorithms. (left:library, right:user)

```
namespace traits
{
    template <typename P>
    struct access
    {
        static_assert(...);

        // Implement two methods:
        // static auto x(const P& p);
        // static auto y(const P& p);
    };
}

template <typename P>
double comparable_distance(const P& p1,
                           const P& p2)
{
    auto sqr = [](auto v) { return v * v; };
    return sqr(traits::access<P>::x(p1) -
               traits::access<P>::x(p2))
         + sqr(traits::access<P>::y(p1) -
               traits::access<P>::y(p2));
}
```

```
struct Point
{
    double x, y;
};

namespace traits
{
    template<>
    struct access<Point>
    {
        static auto x(const Point& p)
               { return p.x; }
        static auto y(const Point& p)
               { return p.y; }
    };
}
```

Metafunctions

- Used in template meta-programming.
- Result known at compile time.
- By convention, they define just either one type, called ‘type’, or one (usually integer or enum) value, called ‘value’.
- Used a lot in the standard library.
- Sample gives a very simple version of type promotion - a more sophisticated version is used in Boost.Geometry.

```
template <typename A, typename B>
struct most_precise
{
        using type = std::conditional_t<sizeof(A) > sizeof(B), A, B>;
};
```

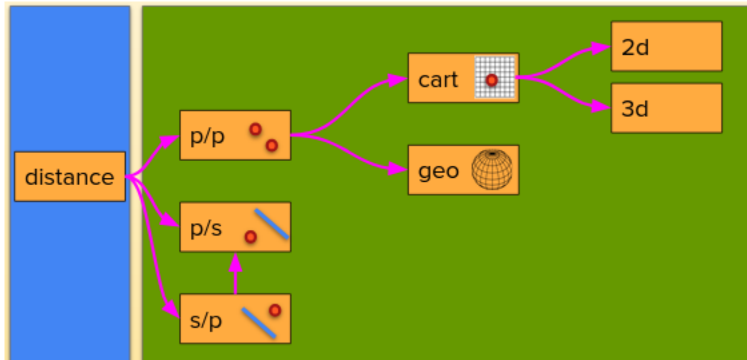
Tag dispatching

- Tags for: geometries, strategies, coordinate systems
- Tag types available through meta-functions
- Dispatching is done w.r.t.
  - Geometry type (point, linestring, polygon, etc.)
  - Coordinate system (cartesian, geographic, etc.)
  - Dimension (2d, 3d, etc.)
  - Strategy (e.g. different distance strategies for geographic cs)
- Provides type-safe, zero-overhead static polymorphism.

Concepts

- Define the compile-time interface a geometry type must satisfy.
- Provide requirements (similar to C++20 concepts, originally implemented with traits + static assertions).
- Examples:
  - `PointConcept`: requires coordinate access, dimension, coordinate type, coordiante system and tag.
  - `BoxConcept`: requires min/max corner points, tag and point type.
  - `PolygonConcept`: requires outer ring and optional inner rings.
  - `RingConcept`: must behave like a Boost.Range Random Access Range
- Future/C++20:
  - with the use of C++20 concepts we can avoid tag dispatching/sfinae
  - e.g. define a concept to verify if some point has a point_tag ?

Distance example flow



Outline

Introduction

Boost Geometry design

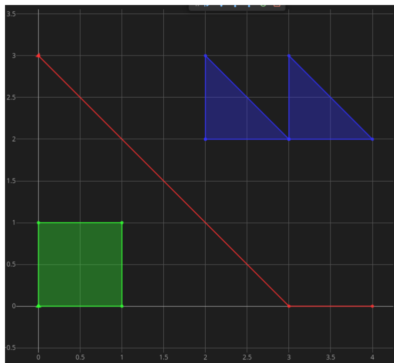
Using Boost Geometry

Spatial computations

Using Geometries

```
using point = bg::model::point<double, 2, bg::cs::cartesian>;
using linestring = bg::model::linestring<point>;
using polygon = bg::model::polygon<point>;
using multi_polygon = bg::model::multi_polygon<polygon>;

linestring ls;
polygon poly;
multi_polygon mpoly;

bg::read_wkt("LINESTRING(0 3,3 0,4 0)", ls);
bg::read_wkt("POLYGON((0 0,0 1,1 1,1 0,0 0))", poly);
bg::read_wkt("MULTIPOLYGON(((2 2,2 3,3 2,2 2)),
                           ((3 2,3 3,4 2,3 2)))", mpoly);

std::cout << bg::distance(ls, poly) << '\n'
          << bg::distance(ls, mpoly);
```

Distance Output




```
0.707107 (to polygon)
0.707107 (to multipolygon)
```

Adapting Custom Types

```
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/register/point.hpp>
#include <boost/geometry/geometries/register/linestring.hpp>
#include <iostream>
namespace bg = boost::geometry;

struct my_point { double x, y; };

BOOST_GEOMETRY_REGISTER_POINT_2D(my_point, double,
        bg::cs::cartesian, x, y)
BOOST_GEOMETRY_REGISTER_LINESTRING(std::vector<my_point>)

int main()
{
  my_point pt{0, 0};
  std::vector<my_point> ls{{0, 1}, {1, 0}, {1, 1}, {0.5, 1}};
  std::cout << bg::distance(pt, ls);
}
```

Result: 0.707107

Algorithms Overview

Measure

- distance, area, length, perimeter
- discrete_frechet_distance, discrete_hausdorff_distance

Geometric

- centroid, convex_hull, envelope, buffer, simplify

Relational and set operations

- crosses, disjoint, equals, intersects, overlaps, relate, within
- difference, intersection, union, sym_difference

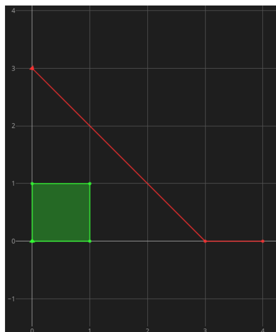
More ...

`https://www.boost.org/doc/libs/latest/libs/geometry/doc/html/geometry/reference/algorithms.html`

Area, Length, Perimeter

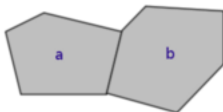
```
linestring ls;
ls.push_back(point(0.0, 3.0));
ls.push_back(point(3.0, 0.0));
ls.push_back(point(4.0, 0.0));

polygon poly = {{{0,0},{0,1},{1,1},{1,0},{0,0}}};

std::cout << "length    " << bg::length(ls) << '\n'
          << "area      " << bg::area(poly) << '\n'
          << "perimeter " << bg::perimeter(poly);
```

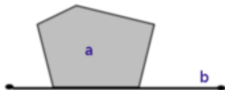
Algorithm Output

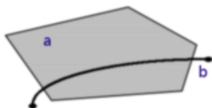


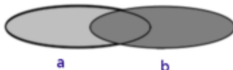
length: `5.242641`
area: `1.000000`
perimeter: `4.000000`

Geospatial topology









DE-9IM intersection matrix

	Interior	Boundary	Exterior
Interior	$\dim[I(a) \cap I(b)] = 2$	$\dim[I(a) \cap B(b)] = 1$	$\dim[I(a) \cap E(b)] = 2$
Boundary	$\dim[B(a) \cap I(b)] = 1$	$\dim[B(a) \cap B(b)] = 0$	$\dim[B(a) \cap E(b)] = 1$
Exterior	$\dim[E(a) \cap I(b)] = 2$	$\dim[E(a) \cap B(b)] = 1$	$\dim[E(a) \cap E(b)] = 2$

within = T*F**F***

Intersects, Within

```
linestring ls;
polygon poly;

bg::read_wkt("LINESTRING(0 1.5, 1.5 0)", ls);
bg::read_wkt("POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))", poly);

std::cout << "intersects " << bg::intersects(ls, poly) << '\n'
          << "within     " << bg::within(ls, poly);
```

Relation Output

intersects: 1

within: 0

Intersects, Relation, Within

```
polygon poly1;
polygon poly2;

bg::read_wkt("POLYGON((0.5 1, 2 2, 1 0.5, 0.5 1))", poly1);
bg::read_wkt("POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))", poly2);

std::cout << "intersects " << bg::intersects(poly1, poly2) << '\n'
          << "within     " << bg::within(poly1, poly2) << '\n'
          << "relation   " << bg::relation(poly1, poly2).str();
```

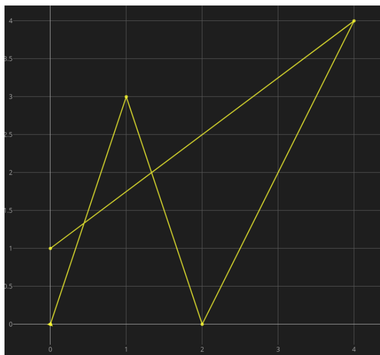
Relation Output

```
intersects 1          (T********)or(*T*******)or...
within     0          (T*F**F***)
relation   212101212
```

Centroid, Envelope, Convex Hull

```
point p;
linestring ls;
box b;
polygon poly;

bg::read_wkt("LINESTRING(0 0, 1 3, 2 0, 4 4, 0 1)", ls);
bg::centroid(ls, p);
bg::envelope(ls, b);
bg::convex_hull(ls, poly);

std::cout << "centroid " << bg::wkt(p) << '\n'
          << "envelope " << bg::wkt(b) << '\n'
          << "hull     " << bg::wkt(poly);
```

Centroid/convex hull output

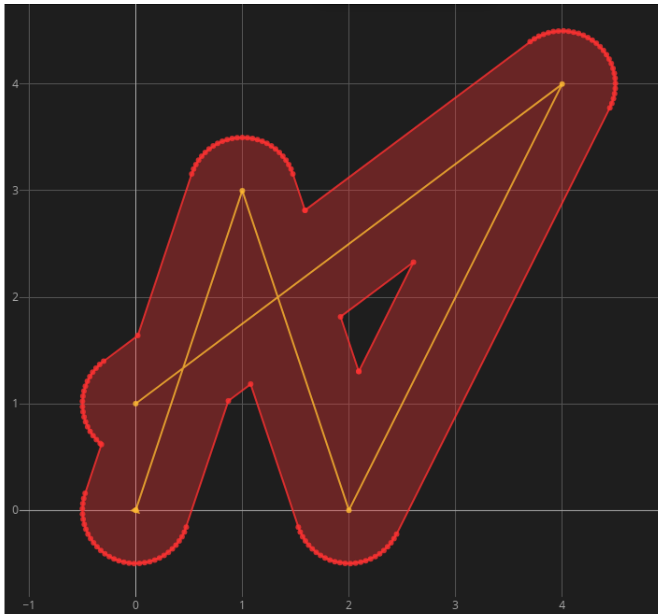


```
centroid POINT(1.7451 1.80392)
envelope POLYGON((0 0,0 4,4 4,4 0,0 0))
hull     POLYGON((0 0,0 1,1 3,4 4,2 0,0 0))
```

Geometric Buffer

```
linestring ls;
multi_polygon mpoly;

bg::read_wkt("LINESTRING(0 0, 1 3, 2 0, 4 4, 0 1)", ls);
namespace bgsb = bg::strategy::buffer;

bg::buffer(ls, mpoly,
           bgsb::distance_symmetric<double>(0.5),
           bgsb::side_straight(),
           bgsb::join_round(64),
           bgsb::end_round(64),
           bgsb::point_circle(64));
```

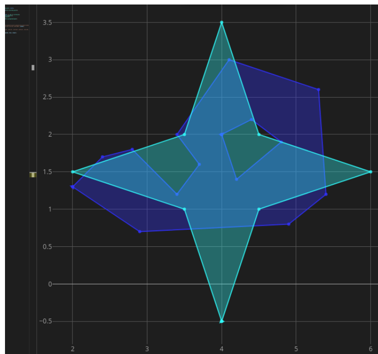
Buffer result

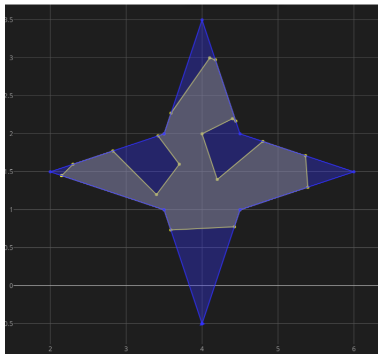


Intersection and Symmetric Difference

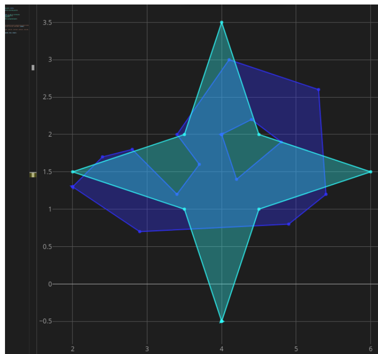
```
polygon green, blue;

boost::geometry::read_wkt(
   "POLYGON((2 1.3,2.4 1.7,2.8 1.8,3.4 1.2,3.7 1.6,3.4 2,4.1 3,5.3
    "(4.0 2.0, 4.2 1.4, 4.8 1.9, 4.4 2.2, 4.0 2.0))", green);

boost::geometry::read_wkt(
   "POLYGON((4.0 -0.5 , 3.5 1.0 , 2.0 1.5 , 3.5 2.0 ,
   4.0 3.5 , 4.5 2.0 , 6.0 1.5 , 4.5 1.0 , 4.0 -0.5))", blue);

multi_polygon mpoly1, mpoly2;

boost::geometry::intersection(green, blue, mpoly1);
boost::geometry::sym_difference(green, blue, mpoly2);
```

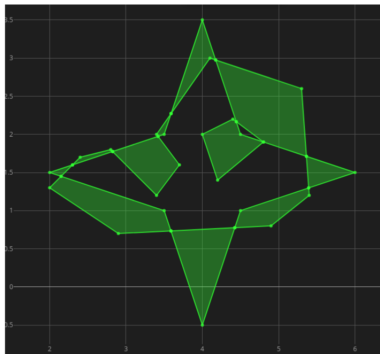
Intersection





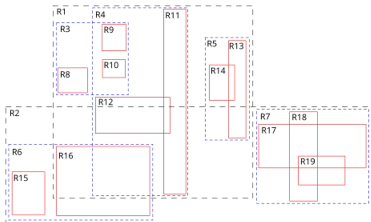
Symmetric Difference (XOR)

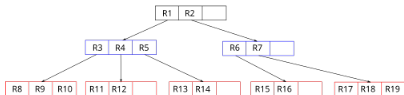




Spatial Index Basics

- R-tree
- Split heuristics: linear, quadratic or r*-tree
- bulk-loading
- user-defined value type
- various spatial and knn query





Spatial Index Basics

```
#include <boost/geometry/index/rtree.hpp>

using point = bg::model::point<double, 2, bg::cs::cartesian>;
using polygon = bg::model::polygon<point>;
using box = bg::model::box<point>;
using value = std::pair<box, std::size_t>;
using rtree = bg::index::rtree<value, bg::index::rstar<16>>;
```

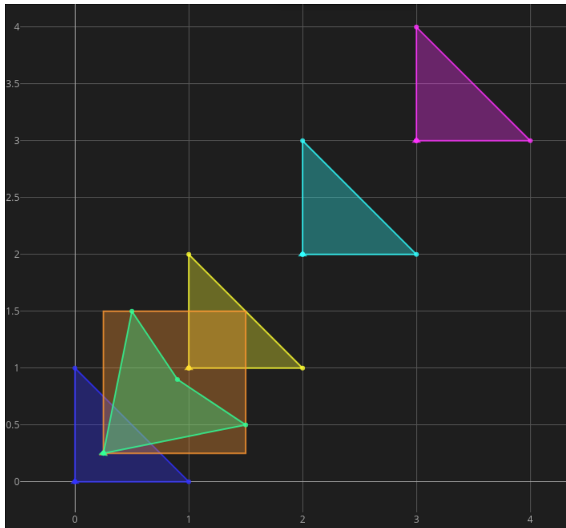
Spatial Index Example

```
std::vector<polygon> polys(4);
bg::read_wkt("POLYGON((0 0, 0 1, 1 0, 0 0))", polys[0]);
bg::read_wkt("POLYGON((1 1, 1 2, 2 1, 1 1))", polys[1]);
bg::read_wkt("POLYGON((2 2, 2 3, 3 2, 2 2))", polys[2]);
bg::read_wkt("POLYGON((3 3, 3 4, 4 3, 3 3))", polys[3]);

rtree rt;
for (std::size_t i = 0; i < polys.size(); ++i) {
  box b = bg::return_envelope<box>(polys[i]);
  rt.insert(std::make_pair(b, i));
}
```

Spatial Index Query

```
polygon qpoly;
bg::read_wkt("POLYGON((0.25 0.25,0.5 1.5,0.9 0.9,
  1.5 0.5,0.25 0.25))", qpoly);
box qbox = bg::return_buffer<box>(bg::return_envelope<box>(qpoly),
  0.0001);

std::vector<value> result;
rt.query(bg::index::intersects(qbox), std::back_inserter(result));

for (const auto& v : result) {
  std::cout << bg::wkt(polys[v.second])
            << (bg::intersects(polys[v.second], qpoly)
              ? " intersects" : " not intersects") << std::endl;
}
```

Result




```
POLYGON((0 0,0 1,1 0,0 0)) intersects
POLYGON((1 1,1 2,2 1,1 1)) not intersects
```

Outline

Introduction

Boost Geometry design

Using Boost Geometry

Spatial computations

Models of the earth and coordinate systems

- Flat
  Accurate only locally. Euclidean geometry. Very fast and simple algorithms.

Models of the earth and coordinate systems

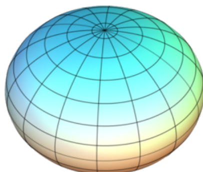
- Flat
  Accurate only locally. Euclidean geometry. Very fast and simple algorithms.
- Sphere
  Widely used (e.g. `google.maps`*). Not very accurate. Fast algorithms.*

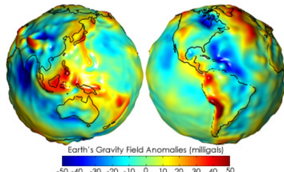
Models of the earth and coordinate systems

- Flat
  Accurate only locally. Euclidean geometry. Very fast and simple algorithms.
- Sphere
  Widely used (e.g. `google.maps`*). Not very accurate. Fast algorithms.*
- Ellipsoid of revolution (or shperoid or ellipsoid)
  State-of-the-art in GIS. More involved algorithms.

Models of the earth and coordinate systems

- Flat
  Accurate only locally. Euclidean geometry. Very fast and simple algorithms.
- Sphere
  Widely used (e.g. `google.maps`*). Not very accurate. Fast algorithms.*
- Ellipsoid of revolution (or shperoid or ellipsoid)
  State-of-the-art in GIS. More involved algorithms.
- Geoid
  Special applications, geophysics etc





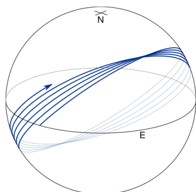
Coordinate systems in Boost.Geometry

```
namespace bg = boost::geometry;

bg::cs::cartesian

bg::cs::spherical_equatorial<bg::degree>
bg::cs::spherical_equatorial<bg::radian>

bg::cs::geographic<bg::degree>
bg::cs::geographic<bg::radian>
```

Geodesics

Definition: Geodesic = shortest path between a pair of points

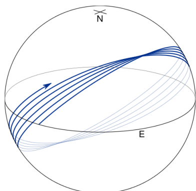
- flat: geodesic = straight line
- sphere: geodesic = great circle
- ellipsoid: geodesic = not closed curve (*except meridians and equator*)

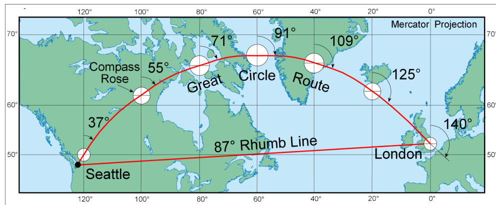


Geodesics

Definition: Geodesic = shortest path between a pair of points

- flat: geodesic = straight line
- sphere: geodesic = great circle
- ellipsoid: geodesic = not closed curve (*except meridians and equator*)





Note: loxodrome or rhump line is an arc crossing all meridians at the same angle (=azimuth). These are straight lines in Mercator projection and not shortest paths.

Geographic algorithms

Two main geodesic problems

- direct: given point p, azimuth a and distance s compute point q and distance s from p on the geodesic defined by p, a
- inverse: given two points compute their distance and corresponding azimuths

Geographic algorithms

Two main geodesic problems

- direct: given point p, azimuth a and distance s compute point q and distance s from p on the geodesic defined by p, a
- inverse: given two points compute their distance and corresponding azimuths

Algorithms:

- 4 different algorithms for distance on ellipsoid implemented as strategies: andoyer (default), thomas, vincenty, series-expansion
  $\rightarrow$ time-accuracy trade-offs

Distance example (revisited)

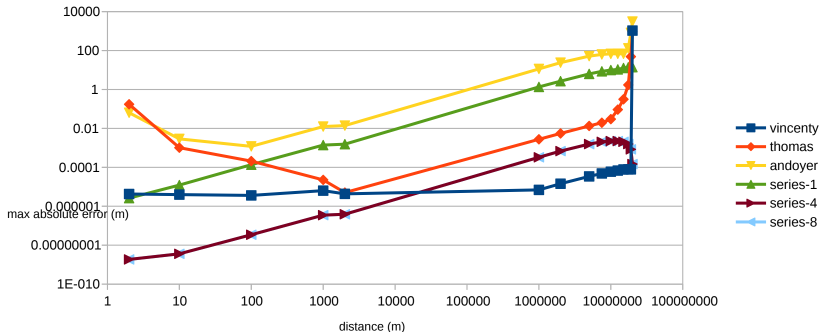
```
namespace bg = boost::geometry;
typedef bg::model::point<double, 2,
                          bg::cs::geographic<bg::degree> > point;

typedef bg::srs::spheroid<double> stype;
typedef bg::strategy::distance::thomas<stype> thomas_type;

std::cout << bg::distance(
               point(23.725750, 37.971536), //Athens, Acropolis
               point(4.3826169, 50.8119483),//Brussels, ULB
               thomas_type())
          << std::endl;
```

Distance example results

spherical	2,085.993 km *
spherical	2,088.327 km **
geographic (andoyer)	2,088.389 km
geographic (thomas)	2,088.384 km
geographic (vincenty)	2,088.385 km
google maps	2,085.99 km

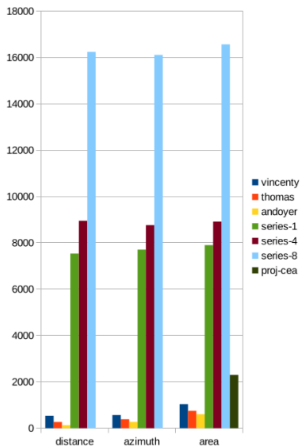
* radius = 6371008.8 (mean Earth radius)
** radius = 6378137 (WGS84 major axis)

Distance strategies benchmark (accuracy)



Data: `https://zenodo.org/records/12510796`

Distance strategies benchmark (performance)



Data: `https://zenodo.org/records/12510796`

Transformations between coordinate systems

- WGS 84 (4326): GPS lat/lon on Earth's surface.
- British National Grid (27700): flat map, units in meters.
- Same place $\Rightarrow$ different numbers in each system.
- To overlay cities on hiking trails: **project** 4326 $\rightarrow$ 27700.

```
namespace bg = boost::geometry;
namespace srs = bg::srs;

using point_car = bg::model::point<double, 2, bg::cs::cartesian>;
using point_geo = bg::model::point<double, 2, bg::cs::geographic<bg::degree>>;

// Create projection from EPSG:4326 to EPSG:27700
srs::transformation<> tr1(srs::epsg(4326), srs::epsg(27700));
srs::transformation<srs::static_epsg<4326>,
                    srs::static_epsg<27700> > tr2;

point_geo pt(0.1278, 51.5074); // Longitude, Latitude for London
point_car pt_out1, pt_out2;

tr1.forward(pt, pt_out1);
tr2.forward(pt, pt_out2);
```

Projected coordinates: 547766, 180865

Projected Hiking trails

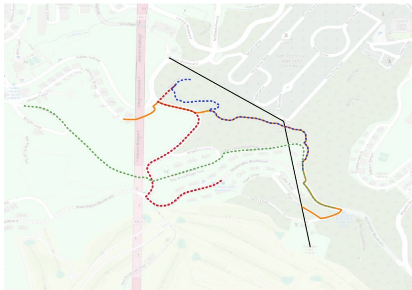
Hiking trails (orange) in Lake District (northwest England). Map data from OpenStreetMap; visualization with QGIS

Analyzing trajectories with Boost.Geometry

- Each walker's path represented as a `linestring` of GPS coordinates (lon, lat).
- Fréchet distance: captures similarity of two curves while respecting traversal order.
- Preprocess: simplify algorithm and R-tree
- No projections: same algorithm for geographic and cartesian CS

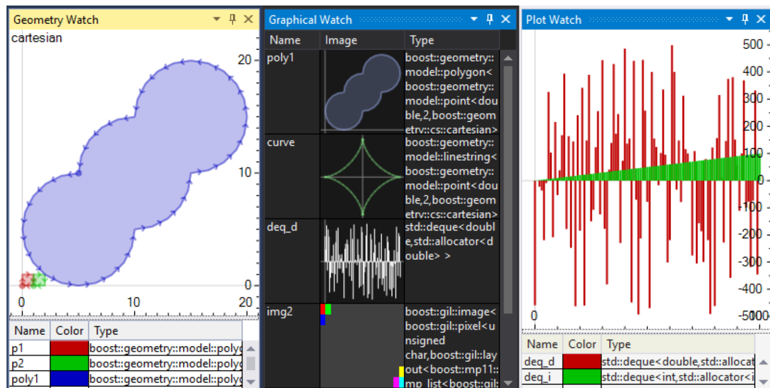
Walkers' trajectories example

```
using point_t = bg::model::point<double, 2, bg::cs::geographic<bg::degree>>;
using linestring_t = bg::model::linestring<point_t>;

linestring_t red, black;

bg::read_wkt("LINESTRING(-122.46428847312926 37.79360088412314,...)", black);

bg::read_wkt("LINESTRING(-122.46702694852982 37.79562087383118,...)", red);

std::cout << "d: " << bg::discrete_frechet_distance(black, red) << std::endl;
```



red : 453.59273618286403 orange : 241.65190882342154 blue : 261.68893003364565 green : 429.19109867209164

Graphical debugging



https://github.com/awulkiew/graphical-debugging

Future steps

- 3D support (recent: PolyhedralSurface limited support)
- Support for curved geometries
- Robustness (predicates and constructions)
- Integrate newer standard features C++17/20

Thank you! Questions?