RAEFGC Book Corrections

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

Created: September 5, 2020
Last Modified: December 23, 2020

Contents

This document contains book corrections for *Robust and Error-Free Geometric Computations*. It also contains comments and thoughts about topics in the book.

1 Book Corrections Organized by Date of Change

2020 December 3, pages 83-84. The paragraph before equation (4.6) has the reversed descriptions of projected point location. If **P** is inside the circle, the projection is **inside** the bowl of the paraboloid. If **P** is outside the circle, the projection is **outside** the bowl of the paraboloid. At the top of page 84, if **P** is **inside** the circle, the sign of S is $+1$. If **P** is **outside** the circle, the sign of S is -1.

2020 December 3, page 84. The pseudocode ToCircumcirleHelper has typographical errors. difv01p1 should be dv01p1 and sumv01p1 should be sv01p1.

2020 November 21, page 80. The right-hand side of equation for u2 should be a00 * a13 - a03 * a10.

2020 September 8, page 56. In the next-to-last paragraph of the page, there is $\hat{t} = 1hatt_{22} \cdots \hat{t}_0\hat{w}$. The LaTeX command is malformed; the expression should be $\hat{t} = 1\hat{t}_{22} \cdots \hat{t}_0\hat{w}$.

2020 September 8, page 85. Listing 4.11 has two functions, each missing an input Vector3<FPType> const& V2.

2020 September 5, pages 349-354. Just after the book appeared in print, I received a bug report that the minimum-volume box code for a data set was actually not the smallest. A box orientation was provided that I verified had smaller volume. It turns out that I incorrectly stated the theorem about conditions a minimum-volume box must satisfy. I had said that the box is supported by a polyhedron face or by 3 mutually perpendicular edges, and I recall getting this information from a post in the Usenet group *comp.graphics.algorithms* many years ago. The mutually perpendicular edge conditions are not correct. In fact, the theorem states that the minimum-volume box must be supported by two polyhedron edges, and these edges are flush with 2 box faces that are perpendicular. In many cases, a face-supported box has minimal volume, but the bug report was about a convex polyhedron for which this is not the case. The theorem is in a paper by Joseph O'Rourke, where he mentions a simple example for the minimum-volume box not supported by a face—a regular tetrahedron. I rewrote my online PDF, Minimum-Volume Box Containing a Set of Points, and I reimplemented the algorithm, posted with Geometric Tools Engine 4.9. The sample application for the minimum-volume box has also been updated. Moreover, the 3D convex hull code used to obtain the polyhedron from the input points has also been updated. The old version required the user to specify the computation type to be rational, and all computations for exact signs of determinants were computed with rational. The new version has a mixture of floating-point interval arithmetic and rational arithmetic to improve the performance. The minimum-volume box code is itself multithreaded for performance using rational arithmetic, but you can execute the code using double-precision arithmetic for fast computations that are reasonably accurate.

2020 September 5, pages 66-67. When porting the GTE4 code to GTL for the class BSPrecision, I had introduced a bug in the bit counting when multiplying two BSRational numbers. I then backported this code to GTE4 to make it available when the book shipped. I fixed the bug in both GTE4 and GTL. Some numbers in Listing 3.10 must be modified. The listing shown next has the comment lines that need modifying.

```
BSPrecision fd3 = fx * fd2 - fx * fd2 + fx * fd2;
// bsr: emin = -2235, emax = 1924, b = 4160, w = 130
BSPrecision dd3 = dx * dd2 - dx * dd2 + dx * dd2;
// bsr: emin = -16110, emax = 15364, b = 31475, w = 984
```

```
BSPrecision fd4 = fx * fd3 - fx * fd3 + fx * fd3 - fx * fd3;
// bsr: emin = -9536, emax = 8214, b = 17751, w = 555
BSPrecision dd4 = dx * dd3 - dx * dd3 + dx * dd3 - dx * dd3;
// bsr: emin = -68736, emax = 65558, b = 134295, w = 4197

fd4 = fd2 * fd2 - fd2 * fd2 + fd2 * fd2 + fd2 * fd2 - fd2 * fd2 + fd2 * fd2;
// bsr: emin = -7152, emax = 6160, b = 13313, w = 417
dd4 = dd2 * dd2 - dd2 * dd2 + dd2 * dd2 + dd2 * dd2 - dd2 * dd2 + dd2 * dd2;
// bsr: emin = -51552, emax = 49168, b = 100721, w = 3148}
```

2020 September 5, page 82-88. The aforementioned BSPrecision bug fix leads to modified N-values for BSRational arithmetic. In ToLine: $N = 70$ for float and $N = 525$ for double. In ToTriangle: $N = 70$ for float and $N = 525$ for double. In ToCircumCircle: $N = 573$ for float and $N = 4329$ for double. In ToPlane: $N = 261$ for float and $N = 1968$ for double. In ToTetrahedron: $N = 261$ for float and $N = 1968$ for double. In ToCircumsphere: $N = 1875$ for float and $N = 14167$ for double.

2020 September 5, pages 128-131. The section has a discussion about the unexpected inaccuracies of floating-point evaluation for $(1 - \cos(t))/t^2$. Figure 6.2 contains the graph of this function over a small interval. I suggested Listing 6.2 as a way to fix the problems, but as it turns out the inaccurate behavior occurs for a much larger domain than indicated. I performed extensive experiments to understand the extent of the problems. The summary is in Approximations to Rotation Matrices and Their Derivatives and it includes remedies. For the function at hand, an equivalent one to implement is $(\sin(t/2)/(t/2))^2/2$, which has more desirable behavior when using floating-point arithmetic. A detail discussion is also provided for minimax polynomial approximations using the Remez algorithm. This discussion includes how to use rational arithmetic to determine the exact signs of alternating power series with decreasing-magnitude terms. Source code was added to the Geometric Tools distribution for the minimax approximations. A tool for generating these is in the folder GTE/Tools/RotationApproximation.

2 Book Corrections Organized by Page Number

Section 3.4.2, page 56. In the next-to-last paragraph of the page, there is $\hat{t} = 1hatt_{22} \cdots \hat{t}_0\hat{w}$. The LaTeX command is malformed; the expression should be $\hat{t} = 1\hat{t}_{22} \cdots \hat{t}_0\hat{w}$.

Section 3.5.1, pages 66-67. When porting the GTE4 code to GTL for the class BSPrecision, I had introduced a bug in the bit counting when multiplying two BSRational numbers. I then backported this code to GTE4 to make it available when the book shipped. I fixed the bug in both GTE4 and GTL. Some numbers in Listing 3.10 must be modified. The listing shown next has the comment lines that need modifying.

```
BSPrecision fd3 = fx * fd2 - fx * fd2 + fx * fd2;
// bsr: emin = -2235, emax = 1924, b = 4160, w = 130
BSPrecision dd3 = dx * dd2 - dx * dd2 + dx * dd2;
// bsr: emin = -16110, emax = 15364, b = 31475, w = 984

BSPrecision fd4 = fx * fd3 - fx * fd3 + fx * fd3 - fx * fd3;
// bsr: emin = -9536, emax = 8214, b = 17751, w = 555
BSPrecision dd4 = dx * dd3 - dx * dd3 + dx * dd3 - dx * dd3;
// bsr: emin = -68736, emax = 65558, b = 134295, w = 4197

fd4 = fd2 * fd2 - fd2 * fd2 + fd2 * fd2 + fd2 * fd2 - fd2 * fd2 + fd2 * fd2;
// bsr: emin = -7152, emax = 6160, b = 13313, w = 417
dd4 = dd2 * dd2 - dd2 * dd2 + dd2 * dd2 + dd2 * dd2 - dd2 * dd2 + dd2 * dd2;
// bsr: emin = -51552, emax = 49168, b = 100721, w = 3148
```

Section 4.2, page 80. The right-hand side of equation for u2 should be a00 * a13 - a03 * a10.

Section 4.3, page 82-88. The aforementioned BSPrecision bug fix leads to modified N-values for BSRational arithmetic. In ToLine: $N = 70$ for float and $N = 525$ for double. In ToTriangle: $N = 70$ for float and $N = 525$ for double. In ToCircumCircle: $N = 573$ for float and $N = 4329$ for double. In ToPlane: $N = 261$ for float and $N = 1968$ for double. In ToTetrahedron: $N = 261$ for float and $N = 1968$ for double. In ToCircumsphere: $N = 1875$ for float and $N = 14167$ for double.

Section 4.3.1, pages 83-84. The paragraph before equation (4.6) has the reversed descriptions of projected point location. If **P** is inside the circle, the projection is **inside** the bowl of the paraboloid. If **P** is outside the circle, the projection is **outside** the bowl of the paraboloid. At the top of page 84, if **P** is **inside** the circle, the sign of S is $+1$. If **P** is **outside** the circle, the sign of S is -1.

Section 4.3.1, page 84. The pseudocode ToCircumcirleHelper has typographical errors. difv01p1 should be dv01p1 and sumv01p1 should be sv01p1.

Section 4.3.2, page 85. Listing 4.11 has two functions, each missing an input Vector3<FPType> const& V2.

Section 6.1.1, pages 128-131. The section has a discussion about the unexpected inaccuracies of floating-point evaluation for $(1 - \cos(t))/t^2$. Figure 6.2 contains the graph of this function over a small interval. I suggested Listing 6.2 as a way to fix the problems, but as it turns out the inaccurate behavior occurs for a much larger domain than indicated. I performed extensive experiments to understand the extent of the problems. The summary is in Approximations to Rotation Matrices and Their Derivatives and it includes remedies. For the function at hand, an equivalent one to implement is $(\sin(t/2)/(t/2))^2/2$, which has more desirable behavior when using floating-point arithmetic. A detail discussion is also provided for minimax polynomial approximations using the Remez algorithm. This discussion includes how to use rational arithmetic to determine the exact signs of alternating power series with decreasing-magnitude terms. Source code was added to the Geometric Tools distribution for the minimax approximations. A tool for generating these is in the folder GTE/Tools/RotationApproximation.

Section 9.7, pages 349-354. Just after the book appeared in print, I received a bug report that the minimum-volume box code for a data set was actually not the smallest. A box orientation was provided that I verified had smaller volume. It turns out that I incorrectly stated the theorem about conditions a minimum-volume box must satisfy. I had said that the box is supported by a polyhedron face or by 3 mutually perpendicular edges, and I recall getting this information from a post in the Usenet group *comp.graphics.algorithms* many years ago. The mutually perpendicular edge conditions are not correct. In fact, the theorem states that the minimum-volume box must be supported by two polyhedron edges, and these edges are flush with 2 box faces that are perpendicular. In many cases, a face-supported box has minimal volume, but the bug report was about a convex polyhedron for which this is not the case. The theorem is in a paper by Joseph O'Rourke, where he mentions a simple example for the minimum-volume box not supported by a face—a regular tetrahedron. I rewrote my online PDF, Minimum-Volume Box Containing a Set of Points, and I reimplemented the algorithm, posted with Geometric Tools Engine 4.9. The sample application for the minimum-volume box has also been updated. Moreover, the 3D convex hull code used to obtain the polyhedron from the input points has also been updated. The old version required the user to specify the computation type to be rational, and all computations for exact signs of determinants were computed with rational. The new version has a mixture of floating-point interval arithmetic and rational arithmetic to improve the performance. The minimum-volume box code is itself multithreaded for performance using rational arithmetic, but you can execute the code using double-precision arithmetic for fast computations that are reasonably accurate.

3 Comments and Thoughts on Various Topics

3.1 Dyadic Rationals

Someone was kind enough to point out that my class BSNumber is an implemention of the *dyadic rationals*. I had forgotten that term from my undergraduate mathematics days back in the mid 1970s.

3.2 Bit Counting with BSPrecision

The BSPrecision class allows you to determine the maximum number of bits required for BSNumber or BSRational when computing a sequence of expressions. The bit count is conservative for BSNumber, but the number is within reason for many applications. The bit count is extremely conservative for BSRational because an addition operation $x_0/y_0 + x_1/y_1 = (x_0y_1 + x_1y_0)/(y_0y_1)$ requires multiplication of BSNumber objects, causing the bit count to become large very quickly as the depths of the expression trees increases. Moreover, if the denominators are 1 for many of the operations, the bit count is much larger than required for the size numbers in an application. Sometimes it is better to try to rewrite expressions involving BSRational to use only BSNumber, with possibly divisions as the final step in an expression tree. The bit counts for BSNumber can be a lot smaller. An example where I had to do this is MinimumVolumeBox3 to compute exact volumes of boxes. This reduced a naive BSRational bit count on the order of $100K$ to a BSNumber bit count on the order of $2K$ followed by a single BSRational divison at the end.

3.3 Performance Issue with FPU State Changes

Listings 4.1 and 4.2 have pseudocode with commands to get and set the floating-point rounding modes. The Geometric Tools class FPInterval is an implementation of floating-point interval arithmetic and uses std::fegetround and std::fesetround, provided by header <cfenv>, for controlling the rounding modes. The code works correctly, but if the interval arithmetic is used heavily in an application, the get/set of the rounding mode shows up as a significant bottleneck when profiling. State changes on the floating-point hardware are expensive.

One such example is in the recently modified class ConvexHull3, where the exact signs of 3×3 determinants are computed using a mixture of floating-point interval arithmetic and rational arithmetic. The PrimalQuery::ToPlane query of Listing 4.11 can be used for this, but the performance turned out to be worse than using only rational arithmetic, which is what the previous convex hull code used. To improve the performance, it is necessary to minimize the number of rounding mode changes. The ToPlane query without interval arithmetic is

```
template <typename T>
class ToPlaneQuery
{
public:
    int operator()(Vector3<T> const& P, Vector3<T> const& V0,
        Vector3<T> const& V1, Vector3<T> const& V2)
    {
        Vector3<T> diff0 = P - V0;
        Vector3<T> diff1 = V1 - V0;
        Vector3<T> diff2 = V2 - V0;
        Vector3<T> cross = Cross(diff1, diff2);
        T det = Dot(diff0, cross);
        T const zero = static_cast<T>(0);
        return det > zero ? +1 : (det < zero ? -1 : 0);
```

```
    }
};
```

Naturally, when the theoretical determinant is nearly zero, floating-point rounding errors can lead to a misclassification of the sign of the determinant.

An exact-sign implementation using FPInterval is

```
template <typename T>
class ToPlaneQueryMaxStateChange
{
public:
    // Choose Rational to be the fixed-precision arithmetic class with
    // enough bits to compute the determinant exactly. No divisions are
    // required, so we do not need to use BSRational.
    using Rational = BSNumber<UIntegerFP32<27>>;

    int operator()(Vector3<T> const& P, Vector3<T> const& V0,
        Vector3<T> const& V1, Vector3<T> const& V2)
    {
        T const zero(0);
        FPInterval<T> fpiDet;
        Evaluate<FPInterval<T>>(P, V0, V1, V2, fpiDet);
        if (fpiDet[0] > zero)
        {
            return +1;
        }
        else if (fpiDet[1] < zero)
        {
            return -1;
        }
        else
        {
            Rational rDet;
            Evaluate<Rational>(P, V0, V1, V2, rDet);
            return rDet.GetSign();
        }
    }

private:
    template <typename ResultType>
    void Evaluate(Vector3<T> const& P, Vector3<T> const& V0,
        Vector3<T> const& V1, Vector3<T> const& V2, ResultType& det)
    {
        // No FPU state changes occur in this loop.
        Vector3<ResultType> p, v0, v1, v2;
        for (int i = 0; i < 3; ++i)
        {
            p[i] = ResultType(P[i]);
            v0[i] = ResultType(V0[i]);
            v1[i] = ResultType(V1[i]);
            v2[i] = ResultType(V2[i]);
        }

        // There are 3 FPU state changes per component of diff0, per
        // component of diff1 and per component of diff2 for a total of
        // 27 FPU state changes (9 per diff* expression).
        Vector3<ResultType> diff0 = p - v0;
        Vector3<ResultType> diff1 = v1 - v0;
        Vector3<ResultType> diff2 = v2 - v0;

        // The cross product has 3 components, each of the form a*b-c*d.
        // The products a*b and c*d each have 3 FPU state changes. The
        // difference a*b-c*d has 3 FPU state changes. The total number
        // of FPU state changes is 27.
        Vector3<ResultType> cross = Cross(diff1, diff2);

        // The dot product is a sum a*b+c*d+e*f. Each product term has
        // 3 FPU state changes. Each sum has 3 fpu state changes. The total
        // number of FPU state changes is 15.
```

```
        //det = Dot(diff0, cross);
        ResultType x0c0 = diff0[0] * cross[0];
        ResultType y0c1 = diff0[1] * cross[1];
        ResultType z0c2 = diff0[2] * cross[2];
        det = x0c0 + y0c1 + z0c2;

        // The total number of FPU state changes is 69.
    }
};
```

The interval arithmetic code can be factored so that for a single rounding mode, the largest block of floating-point expressions is evaluated before having to consume the results for a different rounding mode. An implementation for the ToPlaneQuery is

```
template <typename T>
class ToPlaneQueryMinStateChange
{
public:
    using Rational = BSNumber<UIntegerFP32<27>>;

    int operator()(Vector3<T> const& P, Vector3<T> const& V0,
        Vector3<T> const& V1, Vector3<T> const& V2)
    {
        T const zero(0);
        std::array<T, 2> fpiDet;
        EvaluateInterval(P, V0, V1, V2, fpiDet);
        if (fpiDet[0] > zero)
        {
            return +1;
        }
        else if (fpiDet[1] < zero)
        {
            return -1;
        }
        else
        {
            Rational rDet;
            EvaluateRational(P, V0, V1, V2, rDet);
            return rDet.GetSign();
        }
    }

private:
    void EvaluateInterval(Vector3<T> const& P, Vector3<T> const& V0,
        Vector3<T> const& V1, Vector3<T> const& V2, std::array<T, 2>& fpiDet)
    {
        int saveMode = std::fegetround();

        // The intervals for P-V0 = (x0,y0,z0), V1-V0 = (x1,y1,z1) and
        // V2-V0 = (x2,y2,z2).
        std::array<T, 2> x0, y0, z0, x1, y1, z1, x2, y2, z2;
        std::fesetround(FE_DOWNWARD);
        x0[0] = P[0] - V0[0];
        y0[0] = P[1] - V0[1];
        z0[0] = P[2] - V0[2];
        x1[0] = V1[0] - V0[0];
        y1[0] = V1[1] - V0[1];
        z1[0] = V1[2] - V0[2];
        x2[0] = V2[0] - V0[0];
        y2[0] = V2[1] - V0[1];
        z2[0] = V2[2] - V0[2];
        std::fesetround(FE_UPWARD);
        x0[1] = P[0] - V0[0];
        y0[1] = P[1] - V0[1];
        z0[1] = P[2] - V0[2];
        x1[1] = V1[0] - V0[0];
        y1[1] = V1[1] - V0[1];
        z1[1] = V1[2] - V0[2];
        x2[1] = V2[0] - V0[0];
        y2[1] = V2[1] - V0[1];
```

```
    z2[1] = V2[2] - V0[2];

    // Compute Cross(V1-V0, V2-V0)
    //   = (y1 * z2 - y2 * z1, x2 * z1 - x1 * z2, x1 * y2 - x2 * y1)
    std::array<T, 2> y1z2, y2z1, x2z1, x1z2, x1y2, x2y1;
    std::fesetround(FE_DOWNWARD);
    y1z2[0] = IntervalProductDown(y1, z2);
    y2z1[0] = IntervalProductDown(y2, z1);
    x2z1[0] = IntervalProductDown(x2, z1);
    x1z2[0] = IntervalProductDown(x1, z2);
    x1y2[0] = IntervalProductDown(x1, y2);
    x2y1[0] = IntervalProductDown(x2, y1);
    std::fesetround(FE_UPWARD);
    y1z2[1] = IntervalProductUp(y1, z2);
    y2z1[1] = IntervalProductUp(y2, z1);
    x2z1[1] = IntervalProductUp(x2, z1);
    x1z2[1] = IntervalProductUp(x1, z2);
    x1y2[1] = IntervalProductUp(x1, y2);
    x2y1[1] = IntervalProductUp(x2, y1);

    // cross = (c0, c1, c2)
    std::array<T, 2> c0, c1, c2;
    std::fesetround(FE_DOWNWARD);
    c0[0] = y1z2[0] - y2z1[1];
    c1[0] = x2z1[0] - x1z2[1];
    c2[0] = x1y2[0] - x2y1[1];
    std::fesetround(FE_UPWARD);
    c0[1] = y1z2[1] - y2z1[0];
    c1[1] = x2z1[1] - x1z2[0];
    c2[1] = x1y2[1] - x2y1[0];

    // fpiDet = Dot(diff0, cross) = x0 * c0 + y0 * c1 + z0 * c2
    std::array<T, 2> x0c0, y0c1, z0c2;
    std::fesetround(FE_DOWNWARD);
    x0c0[0] = IntervalProductDown(x0, c0);
    y0c1[0] = IntervalProductDown(y0, c1);
    z0c2[0] = IntervalProductDown(z0, c2);
    fpiDet[0] = x0c0[0] + y0c1[0] + z0c2[0];
    std::fesetround(FE_UPWARD);
    x0c0[1] = IntervalProductUp(x0, c0);
    y0c1[1] = IntervalProductUp(y0, c1);
    z0c2[1] = IntervalProductUp(z0, c2);
    fpiDet[1] = x0c0[1] + y0c1[1] + z0c2[1];

    std::fesetround(saveMode);

    // The number of FPU state changes is 9.
}

void EvaluateRational(Vector3<T> const& P, Vector3<T> const& V0,
    Vector3<T> const& V1, Vector3<T> const& V2, Rational& rResult)
{
    Vector3<Rational> rP = { P[0], P[1], P[2] };
    Vector3<Rational> rV0 = { V0[0], V0[1], V0[2] };
    Vector3<Rational> rV1 = { V1[0], V1[1], V1[2] };
    Vector3<Rational> rV2 = { V2[0], V2[1], V2[2] };
    Vector3<Rational> rDiff0 = rP - rV0;
    Vector3<Rational> rDiff1 = rV1 - rV0;
    Vector3<Rational> rDiff2 = rV2 - rV0;
    Vector3<Rational> rCross = Cross(rDiff1, rDiff2);
    rResult = Dot(rDiff0, rCross);
}

T IntervalProductDown(std::array<T, 2> const& u, std::array<T, 2> const& v)
{
    T const zero(0);
    T w0;
    if (u[0] >= zero)
    {
        if (v[0] >= zero)
        {
            w0 = u[0] * v[0];
```

```
        }
        else if (v[1] <= zero)
        {
            w0 = u[1] * v[0];
        }
        else
        {
            w0 = u[1] * v[0];
        }
    }
    else if (u[1] <= zero)
    {
        if (v[0] >= zero)
        {
            w0 = u[0] * v[1];
        }
        else if (v[1] <= zero)
        {
            w0 = u[1] * v[1];
        }
        else
        {
            w0 = u[0] * v[1];
        }
    }
    else
    {
        if (v[0] >= zero)
        {
            w0 = u[0] * v[1];
        }
        else if (v[1] <= zero)
        {
            w0 = u[1] * v[0];
        }
        else
        {
            w0 = u[0] * v[0];
        }
    }
    return w0;
}

T IntervalProductUp(std::array<T, 2> const& u, std::array<T, 2> const& v)
{
    T const zero(0);
    T w1;
    if (u[0] >= zero)
    {
        if (v[0] >= zero)
        {
            w1 = u[1] * v[1];
        }
        else if (v[1] <= zero)
        {
            w1 = u[0] * v[1];
        }
        else
        {
            w1 = u[1] * v[1];
        }
    }
    else if (u[1] <= zero)
    {
        if (v[0] >= zero)
        {
            w1 = u[1] * v[0];
        }
        else if (v[1] <= zero)
        {
            w1 = u[0] * v[0];
        }
```

```
            else
            {
                w1 = u[0] * v[0];
            }
        }
        else
        {
            if (v[0] >= zero)
            {
                w1 = u[1] * v[1];
            }
            else if (v[1] <= zero)
            {
                w1 = u[0] * v[0];
            }
            else
            {
                w1 = u[1] * v[1];
            }
        }
        return w1;
    }
};
```

The most complicated part of the factoring involves the multiplication of intervals. The functions IntervalProductDown and IntervalProductUp are a factoring of the operator* function for FPInterval to encapsulate operations that occur only when rounding down or that occur only when rounding up.

Each maximal block of expressions involvs a rounding-down and a rounding-up phase. The next maximal block consumes the results of the previous maximal block, so the rounding down and rounding up must be applied again.

Observe that the direct implementation using FPInterval has 69 FPU state changes but the refactored implementation uses only 9 FPU state changes.

3.4 C++ Compiler Support for FPU State Changes

The C++11 (and later) language has support for controlling the floating-point environment. The support can be found at the C++ Reference website via the webpage Floating-Point Environment. In particular, this webpage mentions that access and modification of the floating-point environment is meaningful only when #pragma STDC FENV_ACCESS is supported, and with parameter ON or OFF.

Such control is required for a floating-point implementation of interval arithmetic. The main issue is that in a release build of the code, an optimizing compiler can (and usually does) change the order of an expression tree. In doing so, it ignores any calls to std::fesetround and assumes that the default rounding mode is in effect (round-to-nearest-ties-to-even).

Unfortunately, the IEEE 754-2008 Standard for Floating-Point Arithmetic can only tell the hardware architects how to support IEEE format numbers and the associated arithmetic, environment and exception handling. Programming languages and compilers have other constraints that might make it difficult to support all of the IEEE Standard; for example, a compiler might be written for a computer with floating-point hardware that has a model of computing different from IEEE floating-point arithmetic.

The following is a quote from the introduction section of the IEEE Standard.

> This standard is a product of the Floating-Point Working Group of, and sponsored by, the

Microprocessor Standards Committee of the IEEE Computer Society.

This standard provides a discipline for performing floating-point computation that yields results independent of whether the processing is done in hardware, software, or a combination of the two. For operations specified in the normative part of this standard, numerical results and exceptions are uniquely determined by the values of the input data, the operation, and the destination, all under user control.

This standard defines a family of commercially feasible ways for systems to perform binary and decimal floating-point arithmetic. Among the desiderata that guided the formulation of this standard were:

a) Facilitate movement of existing programs from diverse computers to those that adhere to this standard as well as among those that adhere to this standard.

b) Enhance the capabilities and safety available to users and programmers who, although not expert in numerical methods, might well be attempting to produce numerically sophisticated programs.

c) Encourage experts to develop and distribute robust and efficient numerical programs that are portable, by way of minor editing and recompilation, onto any computer that conforms to this standard and possesses adequate capacity. Together with language controls it should be possible to write programs that produce identical results on all conforming systems.

d) Provide direct support for
   — execution-time diagnosis of anomalies
   — smoother handling of exceptions
   — interval arithmetic at a reasonable cost.

e) Provide for development of
   — standard elementary functions such as exp and cos
   — high precision (multiword) arithmetic
   — coupled numerical and symbolic algebraic computation.

f) Enable rather than preclude further refinements and extensions.

In programming environments, this standard is also intended to form the basis for a dialog between the numerical community and programming language designers. It is hoped that language-defined methods for the control of expression evaluation and exceptions might be defined in coming years, so that it will be possible to write programs that produce identical results on all conforming systems. However, it is recognized that utility and safety in languages are sometimes antagonists, as are efficiency and portability. Therefore, it is hoped that language designers will look on the full set of operation, precision, and exception controls described here as a guide to providing the programmer with the ability to portably control expressions and exceptions. It is also hoped that designers will be guided by this standard to provide extensions in a completely portable way.

I added the red highlighting to stress that I had the same hope for access to *the full set of operation, precision, and exception controls*. I expected that with current-day computer technology, all compilers for processors with IEEE floating-point implementations would support changes in rounding mode and in handling floating-point exceptions. Sadly, that is not the case.

In the following discussion, consider trying to compile the block of code

```
    #include <cfenv>

    void TryIntervalArithmetic(float x, float y, float& lowerBound, float& upperBound)
    {
#pragma STDC FENV_ACCESS ON
        auto saveMode = std::fegetround();
        std::fesetround(FE_DOWNWARD);
        lowerBound = x * y;
        std::fesetround(FE_UPWARD);
        upperBound = x * y;
        std::fesetround(saveMode);
#pragma STDC FENV_ACCESS OFF
    }
```

The compilers I tried warned that the pragma is unknown. In a debug build, the compiler optimizations are turned off and the code produces the correct outputs. In a release build with the standard floating-point model enabled in the compilers, the outputs are incorrect.

3.4.1 Microsoft Visual Studio C++ Compiler

The Microsoft Visual Studio C++ compiler has support for accessing and modifying the floating-point environment. The default floating-point model is /fp:precise, which appears in the project settings dialog

Configuration Properties | C/C++ | Code Generation | Floating Point Model

This model allows the compiler to optimize floating-point expressions, so interval arithmetic will not always work correctly with it. You must modify the model to /fp:strict. You can set the model for the entire project, which is probably not desired because other code not using interval arithmetic should be optimized. However, you can set this for each source file that requires it. The Microsoft compiler warns that it does not recognize #pragma STDC FENV_ACCESS. It has a Microsoft-specific pragma, #pragma fenv_access (on,off) that works when the model is /fp:precise. However, this can be used only at global scope or file scope; it cannot be used to control a block of code with a function. The previous listing can be modified to

```
    // The floating-point model is /fp:precise and the build configuration
    // is Release.
    #include <cfenv>

#pragma fenv_access (on)
    // The compiler does not optimize the floating-point expressions
    // in this code, so the rounding works.
    void TryIntervalArithmetic0(float x, float y, float& lowerBound, float& upperBound)
    {
        auto saveMode = std::fegetround();
        std::fesetround(FE_DOWNWARD);
        lowerBound = x * y;
        std::fesetround(FE_UPWARD);
        upperBound = x * y;
        std::fesetround(saveMode);
    }
#pragma fenv_access (off)

    // The compiler does optimize the floating-point expressions
    // in this code. The rounding does not work because the compiler
    // optimizes out the upperBound computation and assigns to it
    // the lowerBound computation.
    void TryIntervalArithmetic1(float x, float y, float& lowerBound, float& upperBound)
    {
        auto saveMode = std::fegetround();
        std::fesetround(FE_DOWNWARD);
        lowerBound = x * y;
        std::fesetround(FE_UPWARD);
```

```
        upperBound = x * y;
        std::fesetround(saveMode);
    }
```

If you choose the model /fp:strict, every source file in the project effectively has #pragma fenv_access (on) at file scope, which is typically not desirable.

If you compile the code in a Debug configuration, optimization is turned off and the rounding modes work. The issue is centered on controlling the optimizations the compiler applies in a Release configuration.

3.4.2 Intel C++ Compiler

The Intel $C++$ compiler, version 19.1 or later, also has support for accessing and modifying the floating-point environment. Their options are described Floating-Point Options. In particular, -fp-model precise corresponds to Microsoft's /fp:precise and -fp-model strict corresponds to Microsoft's /fp:strict.

I tried to verify this with my Intel C++ compiler, version 19.1, through Microsoft Visual Studio 2019 16.8.3 by right-clicking on the solution folder in Solution Explorer and selecting the pop-up menu item "Intel Compiler". The submenu has "Use Intel C++". An attempt to compile failed, but not because of source-code compilation. The MSVS 2019 16.8 release broke the Intel $C++$ front-end. Intel forums has a post about this. Visual Studio developer forums also has a post. The Visual Studio post mentions the Intel forums post.

As soon as I am able to, I will try the Intel compiler from a command line.

3.4.3 GCC Does Not Support FPU State Changes

Searching online, the GCC support for accessing and modifying the floating-point environment is supposed to be enabled by the option -frounding-math. I tried this with GCC 10.2.1 on Fedora 33. The compiler complains that it does not recognize #pragma STDC FENV_ACCESS, so I commented out those lines. The release-build outputs lowerBound and upperBound are different from those of the debug-build outputs. In fact, they are the same number, which is not correct because the real-valued product is not representable by a 32-bit floating-point number.

Based on my online reading, it appears that the GCC optimizing compiler simply cannot be told NOT to optimize the expressions. If you try to use the GTE or GTL interval arithmetic code on a Linux machine, generally the results will not be correct. You have to resort to using the GTE or GTL rational arithmetic support.

Relevant pages I found are FloatingPointMath at the GNU wiki. In particular, the Compliance column of the first table has first-row entry of Full with the following statement: *Warning! GCC currently assumes that the same rounding mode is in effect everywhere.* It has a link to a Bugzilla bug report, an interesting discussion with some bickering between people who wish to have control over the floating-point rounding mode and people who argue why providing that option is too complicated because of other issues (such as hardware that does not even use the IEEE floating-point model). My stance is: Would you please not bicker for over 10 years and actually try to solve the problem? At least provide the ability when the target architecture is a CPU with IEEE 754-2008 compliant hardware, and ignore it on non-IEEE-compliant hardware. Interval arithmetic is an important concept in geometric computing, and I would rather not have to add assembly code to implement it. Thank you.

The Geometric Tools Library (GTL) code will have conditional compilation that is controlled by a define GTL_COMPILER_HAS_FP_STRICT which exposes code that sets the floating-point rounding mode but falls back to only GTL rational arithmetic when the compiler ignores setting of the rounding mode.