RAEFGC Book Corrections

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

Created: September 5, 2020
Last Modified: September 8, 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 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 4.3, 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 4.3.2, 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.