GTEngine: Arbitrary Precision Arithmetic

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

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Created: November 25, 2014
Last Modified: December 9, 2017

Contents

1 Introduction **3**

2 IEEE 754-2008 Binary Representations **6**

2.1 Representation of 32-bit Floating-Point Numbers 6

2.2 Representation of 64-bit Floating-Point Numbers 7

3 Binary Scientific Numbers **9**

3.1 Multiplication 9

3.2 Addition 10

3.2.1 The Case $p - n > q - m$ 10

3.2.2 The Case $p - n < q - m$ 10

3.2.3 The Case $p - n = q - m$ 11

3.2.4 Determining the Maximum Number of Bits for Addition 11

3.3 Subtraction 12

3.4 Unsigned Integer Arithmetic 13

3.4.1 Addition 13

3.4.2 Subtraction 14

3.4.3 Multiplication 15

3.4.4 Shift Left 17

3.4.5 Shift Right to Odd Number 18

3.4.6 Comparisons 20

3.5 Conversion of Floating-Point Numbers to Binary Scientific Numbers 21

3.6 Conversion of Binary Scientific Numbers to Floating-Point Numbers 23

4 Binary Scientific Rationals **29**

4.1 Arithmetic Operations 29

4.2 Conversion of Floating-Point Numbers to Binary Scientific Rationals 30

4.3 Conversion of Binary Scientific Rationals to Floating-Point Numbers 30

5 Implementation of Binary Scientific Numbers **32**

6 Implementation of Binary Scientific Rationals **38**

7 Exact Representation of Expressions Involving Square Roots **39**

7.1 Real Quadratic Fields 40

7.2 Allowing for Rational d 41

7.3 Detecting Whether the Length of a Vector is the Square of a Rational 41

7.4 Implementation of Arithmetic for Real Quadratic Fields 42

7.5 Expressions with Multiple Square Roots 46

7.5.1 Arithmetic Operations 46

7.5.2 Uniqueness of the Representation of Zero 47

7.5.3 Comparisons of Numbers with Multiple Square Roots 48

8 Performance Considerations **48**

8.1 Static Computation of the Maximum Bits of Precision 48

8.2 Dynamic Computation of the Maximum Bits of Precision 51

8.3 Memory Management 51

9 Alternatives for Computing Signs of Determinants **52**

9.1 Interval Arithmetic 52

9.2 Alternate Logic for Signs of Determinants 54

10 Miscellaneous Items **55**

1 Introduction

Deriving the mathematical details for an algorithm that solves a geometric problem is a well understood process. A direct implementation of the algorithm, however, does not always work out so well in practice. It is difficult to achieve robustness when computing using floating-point arithmetic. The IEEE 754-2008 Standard for floating-point numbers defines a 32-bit binary representation, float, that has 24 bits of precision and a 64-bit binary representation, double, that has 53 bits of precision. For many geometric problems, the number of bits of precision to obtain robustness is larger than those available from the standard floating-point types.

Example 1. Consider the 2D geometric query for testing whether a point $\mathbf{P}$ is inside or outside a convex polygon with counterclockwise ordered vertices $\mathbf{V}_i$ for $0 \le i < n$. The point is strictly inside the polygon when the point triples $\{\mathbf{P}, \mathbf{V}_i, \mathbf{V}_{i+1}\}$ are counterclockwise ordered for all i with the understanding that $\mathbf{V}_n = \mathbf{V}_0$. If at least one of the triples is clockwise ordered, the point is outside the polygon. It is possible that the point is exactly on a polygon edge. An implementation of the algorithm is

```
// Return:  +1 (P strictly inside), -1 (P strictly outside), 0 (P on edge)
template <typename Real>
int GetClassification(Vector2<Real> const& P, int n, Vector2<Real> const* V)
{
    int numPositive = 0, numNegative = 0, numZero = 0;
    for (int i0 = n-1, i1 = 0; i1 < n; i0 = i1++)
    {
        Vector2<Real> diff0 = P - V[i0], diff1 = P - V[i1];
        float dotPerp = diff0[0] * diff1[1] - diff0[1] * diff1[0];
        if (dotPerp > (Real)0)
        {
            ++numPositive;
        }
        else if (dotPerp < (Real)0)
        {
            ++numNegative;
        }
        else
        {
            ++numZero;
        }
    }

    return (numZero == 0 ? ((numPositive == n ? +1 : -1) : 0);
}

Vector2<float> P(0.5f, 0.5f), V[3];
V[0] = Vector2<float>(-7.29045947e-013f, 6.29447341e-013f);
V[1] = Vector2<float>(1.0f, 8.11583873e-013f);
V[2] = Vector2<float>(9.37735566e-013f, 1.0f);

// The return value is 0, so P is determined to be on an edge of the
// triangle.  The function computes numPositive = 2, numNegative = 0,
// and numZero = 1.
int classifyUsingFloat = GetClassification<float>(P, 3, V);

// If rP and rV[i] are rational representations of P and V[i] and you
// have type Rational that supports exact rational arithmetic, then
// the return value is +1, so P is actually strictly inside the triangle.
// The function computes numPositive = 3, numNegative = 0, and numZero = 0.
int classifyUsingExact = GetClassification<Rational>(rP, 3, rV);
```

The values of the sign counters are sensitive to numerical rounding errors when $\mathbf{P}$ is nearly on a polygon edge. A geometric algorithm that depends on a theoretically correct classification can suffer when the floating-point results are not correct. For example, the convex hull of a set of 2D points involves this query. Incorrect

classification can cause havoc with a convex hull finder.

Example 2. Compute the distance between two lines in n dimensions. The lines are specified using two points. The first line contains points $\mathbf{P}_0$ and $\mathbf{P}_1$ and is parameterized by $\mathbf{P}(s) = (1-s)\mathbf{P}_0 + s\mathbf{P}_1$ for all real values s. The second line contains points $\mathbf{Q}_0$ and $\mathbf{Q}_1$ and is parameterized by $\mathbf{Q}(t) = (1-t)\mathbf{Q}_0 + t\mathbf{Q}_1$ for all real values t. The squared distance between a pair of points, one on each line, is

$$F(s,t) = |\mathbf{P}(s) - \mathbf{Q}(t)|^2 = as^2 - 2bst + ct^2 + 2ds - 2et + f$$

where

$$\begin{array}{lll} a = |\mathbf{P}_1 - \mathbf{P}_0|^2, & b = (\mathbf{P}_1 - \mathbf{P}_0) \cdot (\mathbf{Q}_1 - \mathbf{Q}_0), & c = |\mathbf{Q}_1 - \mathbf{Q}_0|^2, \\ d = (\mathbf{P}_1 - \mathbf{P}_0) \cdot (\mathbf{P}_0 - \mathbf{Q}_0), & e = (\mathbf{Q}_1 - \mathbf{Q}_0) \cdot (\mathbf{P}_0 - \mathbf{Q}_0), & f = |\mathbf{P}_0 - \mathbf{Q}_0|^2 \end{array}$$

Observe that $ac - b^2 = |(\mathbf{P}_1 - \mathbf{P}_0) \times (\mathbf{Q}_1 - \mathbf{Q}_0)|^2 \geq 0$. If $ac - b^2 > 0$, the lines are not parallel and the graph of $F(s,t)$ is a paraboloid with a unique global minimum that occurs when the gradient is the zero vector, $\nabla F(s,t) = (0,0)$. If the solution is $(\bar{s}, \bar{t})$, the closest points on the lines are $\mathbf{P}(\bar{s})$ and $\mathbf{Q}(\bar{t})$ and the distance between them is $\sqrt{F(\bar{s}, \bar{t})}$.

If $ac - b^2 = 0$, the lines are parallel and the graph of $F(s,t)$ is a parabolic cylinder with a global minimum that occurs along an entire line. This line contains all the vertices of the parabolas that form the cylinder. Each point $(\bar{s}, \bar{t})$ on this line produces a pair of closest points on the original lines, $\mathbf{P}(\bar{s})$ and $\mathbf{Q}(\bar{t})$, and the distance between them is $\sqrt{F(\bar{s}, \bar{t})}$.

Mathematically, the problem is straightforward to solve. The gradient equation is $\nabla F(s,t) = 2(as - bt + d, -bs + ct - e) = (0,0)$, which is a linear system of two equations in two unknowns,

$$\begin{bmatrix} a & -b \\ -b & c \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} -d \\ e \end{bmatrix}$$

The determinant of the coefficient matrix is $ac - b^2$, so the equations have a unique solution $(\bar{s}, \bar{t})$ when $ac - b^2 > 0$,

$$\begin{bmatrix} \bar{s} \\ \bar{t} \end{bmatrix} = \frac{1}{ac - b^2} \begin{bmatrix} be - cd \\ ae - bd \end{bmatrix}$$

This is the case when the two lines are not parallel. Let us look at a particular example using floating-point arithmetic to solve the system,

```
Vector3<double> P0(-1.0896217473782599,    9.7236145595088601e-007,  0.0);
Vector3<double> P1( 0.91220578597858548,  -9.4369829432107506e-007,  0.0);
Vector3<double> Q0(-0.90010447502136237,   9.0671446351334441e-007,  0.0);
Vector3<double> Q1( 1.0730877178721130,   -9.8185787633992740e-007,  0.0);
Vector3<double> P1mP0 = P1 - P0, Q1mQ0 = Q1 - Q0, P0mQ0 = P0 - Q0;
double a = Dot(P1mP0, P1mP0);    // 4.0073134733092228
double b = Dot(P1mP0, Q1mQ0);    // 3.9499904603425491
double c = Dot(Q1mQ0, Q1mQ0);    // 3.8934874300993294
double d = Dot(P1mP0, P0mQ0);    // -0.37938089385085144
double e = Dot(Q1mQ0, P0mQ0);    // -0.37395400223322062
double det = a * c - b * b;      // 1.7763568394002505e-015
```

```
double sNumer = b * e - c * d;   // 2.2204460492503131e-016
double tNumer = a * e - b * d;   // 4.4408920985006262e-016
double s = sNumer / det;         // 0.125
double t = tNumer / det;         // 0.25
Vector3<double> PClosest = (1.0 - s) * P0 + s * P1;
Vector3<double> QClosest = (1.0 - t) * Q0 + t * Q1;
double distance = Length(PClosest - QClosest);  // 0.43258687891076358
```

Now repeat the experiment using a type Rational that supports exact rational arithmetic.

```
Vector3<Rational> rP0(P0[0], P0[1], P0[2]);
Vector3<Rational> rP1(P1[0], P1[1], P1[2]);
Vector3<Rational> rQ0(Q0[0], Q0[1], Q0[2]);
Vector3<Rational> rQ1(Q1[0], Q1[1], Q1[2]);
Vector3<Rational> rP1mP0 = rP1 - rP0, rQ1mQ0 = rQ1 - rQ0, rP0mQ0 = rP0 - rQ0;
Rational ra = Dot(rP1mP0, rP1mP0);
Rational rb = Dot(rP1mP0, rQ1mQ0);
Rational rc = Dot(rQ1mQ0, rQ1mQ0);
Rational rd = Dot(rP1mP0, rP0mQ0);
Rational re = Dot(rQ1mQ0, rP0mQ0);
Rational rdet = ra * rc - rb * rb;       // 2.4974018083084524e-020
Rational rsNumer = rb * re - rc * rd;    // -3.6091745045569584e-017
Rational rtNumer = ra * re - rb * rd;    // -3.6617913970635857e-017
Rational rs = rsNumer / rdet;            // -1445.1717351007826
Rational rt = rtNumer / rdet;            // -1466.2403882632732
Rational one(1);
Vector3<Rational> rPClosest = (one - rs) * rP0 + rs * rP1;
Vector3<Rational> rQClosest = (one - rt) * rQ0 + rt * rQ1;
Vector3<Rational> rdiff = rPClosest - rQClosest;  // Rational(0)
Rational rsqrDistance = Dot(rdiff, rdiff);
double converted = sqrt((double)rsqrDistance);  // 0.0
```

The values of rdet, rsNumer, rtNumer, rs, and rs were converted back to double values with some loss in precision. Clearly, the double-precision computations suffer greatly from the numerical rounding errors. In fact, the distance must be zero because the two lines are not parallel and lie in the xy-plane, so they must intersect.

Example 3. A common subproblem in geometric algorithms is computing the real-valued roots of a quadratic polynomial $q(x) = a_0 + a_1 x + a_2 x^2$, where $a_2 \neq 0$. The quadratic formula provides the roots,

$$r = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_0 a_2}}{2a_2}$$

The discriminant is $\Delta = a_1^2 - 4a_0 a_2$. If $\Delta > 0$, the polynomial has 2 distinct real-valued roots. If $\Delta = 0$, the polynomial has 1 repeated real-valued root. If $\Delta < 0$, the polynomial has no real-valued roots (both are complex-valued). The numerical problems arise when Δ is nearly zero. Observe that Δ is in the form of a determinant, so it can suffer from the numerical problems in determining its theoretically correct sign.

To obtain a correct classification, you can use exact rational arithmetic to compute Δ. When it is nonnegative, then convert the number back to a floating-point quantity and compute the roots using floating-point arithmetic. The conversion can lose precision, and so the square root operation can magnify the errors just as if you used floating-point arithmetic entirely for the computations. Alternatively, you can implement an algorithm for approximating the square root, compute it using exact rational arithmetic, and then convert back to floating-point, hopefully producing a more accurate result than just the conversion-first-sqrt-second approach.

---

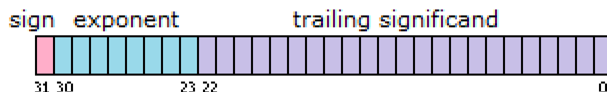
2 IEEE 754-2008 Binary Representations

The 32-bit floating-point type float and the 64-bit floating-point type double are designed according to the IEEE 754-2008 standard for floating-point numbers. The binary scientific notation used to represent such numbers is the basis for the arbitrary precision arithmetic implemented in GTEngine.

2.1 Representation of 32-bit Floating-Point Numbers

The layout of float is shown in Figure 1.

Figure 1. The layout of the floating-point type float.



The sign of the number is stored in bit 31. A 0-valued bit is used for a nonnegative number and a 1-valued bit is used for a negative number. Bits 23 through 30 form an 8-bit unsigned integer β that is a biased representation of the exponent e; the biased exponent satisfies $0 \leq \beta \leq 255$ and the exponent satisfies $-127 \leq e \leq 128$. The trailing significand is stored as a 23-bit unsigned integer in bits 0 through 22. It represents the fractional part of a number when written in binary scientific notation. The integer part of the notation is implied by the representation and not stored explicitly. It is 1 for a normal floating-point number or 0 for a subnormal floating-point number. Listing 1 shows how the binary encoding represents 32-bit floating-point numbers.

Listing 1. Decoding a number of type float.

```
binary32 x = <some 32-bit floating-point number>;
uint32_t s = (0x80000000 & x.encoding) >> 31;  // sign
uint32_t e = (0x7f800000 & x.encoding) >> 23;  // biased exponent
uint32_t t = (0x007fffff & x.encoding);        // trailing significand

if (e == 0)
{
    if (t == 0)  // zeros
    {
        // x = (-1)^s * 0  [allows for +0 and -0]
    }
    else  // subnormal numbers
    {
        // x = (-1)^s * 0.t * 2^{-126}
    }
}
else if (e < 255)  // normal numbers
{
    // x = (-1)^s * 1.t * 2^{e-127}
}
else // special numbers
{
    if (t == 0)
    {
```

```
            // x = (-1)^s * infinity
        }
        else
        {
            if (t & 0x00400000)
            {
                // x = quiet NaN
            }
            else
            {
                // x = signaling NaN
            }
            // payload = t & 0x003fffff
        }
    }
```

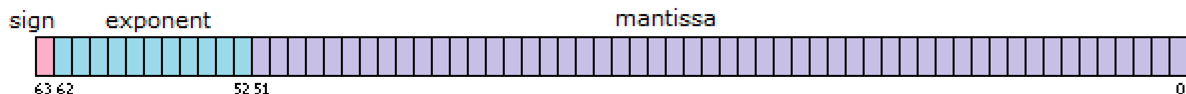
The encoding has signed zeros, $+0$ (encoding `0x00000000`) and -0 (encoding `0x80000000`). There are numerical applications for which it is important to have both representations. The encoding also has signed infinities, $+\infty$ (encoding `0x7f800000`) and $-\infty$ (encoding `0xff800000`). Infinities have special rules applied to them during arithmetic operations. Finally, the encoding has special values, each called *Not-a-Number* (NaN). Some of these are called *quiet NaNs* and are used to provide diagnostic information when unexpected conditions occur during floating-point computations. The others are called *signaling NaNs* and also provide diagnostic information but might also be used to support the needs of specialized applications. A NaN has an associated *payload* whose meaning is at the discretion of the implementer. The IEEE 754-2008 Standard has many requirements regarding the handling of NaNs in numerical computations.

The smallest positive subnormal number occurs when $e = 0$ and $t = 1$, which is $2^{e_{\min}+1-p} = 2^{-149}$. All finite floating-point numbers are integral multiples of this number. The largest positive subnormal number occurs when $e = 0$ and t has all 1-valued bits, which is $2^{e_{\min}}(1 - 2^{1-p}) = 2^{-126}(1 - 2^{-23})$. The smallest positive normal number occurs when $e = 1$ and $t = 0$, which is $2^{e_{\min}} = 2^{-126}$. The largest positive normal number occurs when $e = 254$ and t has all 1-valued bits, which is $2^{e_{\max}}(2 - 2^{1-p}) = 2^{127}(2 - 2^{-23})$.

2.2 Representation of 64-bit Floating-Point Numbers

The layout of `double` is shown in Figure 2.

Figure 2. The layout of the floating-point type `double`.



The sign of the number is stored in bit 63. A 0-valued bit is used for a nonnegative number and a 1-valued bit is used for a negative number. Bits 52 through 62 form an 11-bit unsigned integer β that is a biased representation of the exponent e; the biased exponent satisfies $0 \leq \beta \leq 2047$ and the exponent satisfies $-1023 \leq e \leq 1024$. The trailing significand is stored as a 52-bit unsigned integer in bits 0 through 51. It

represents the fractional part of a number when written in binary scientific notation. The integer part of the notation is implied by the representation and not stored explicitly. It is 1 for a normal floating-point number or 0 for a subnormal floating-point number. Listing 2 shows how the binary encoding represents 64-bit floating-point numbers.

Listing 2. Decoding a number of type double.

```
binary64 x = <some 64-bit floating-point number>;
uint64_t s = (0x8000000000000000 & x.encoding) >> 63;  // sign
uint64_t e = (0x7ff0000000000000 & x.encoding) >> 52;  // biased exponent
uint64_t t = (0x000fffffffffffff & x.encoding);  // trailing significand

if (e == 0)
{
    if (t == 0)  // zeros
    {
        // x = (-1)^s * 0  [allows for +0 and -0]
    }
    else  // subnormal numbers
    {
        // x = (-1)^s * 0.t * 2^{-1022}
    }
}
else if (e < 2047)  // normal numbers
{
    // x = (-1)^s * 1.t * 2^{e-1023}
}
else // special numbers
{
    if (t == 0)
    {
        // x = (-1)^s * infinity
    }
    else
    {
        if (t & 0x0008000000000000)
        {
            // x = quiet NaN
        }
        else
        {
            // x = signaling NaN
        }
        // payload = t & 0x0007ffffffffffff
    }
}
```

The encoding has signed zeros, $+0$ (encoding `0x0000000000000000`) and -0 (encoding `0x8000000000000000`). There are numerical applications for which it is important to have both representations. The encoding also has signed infinities, $+\infty$ (encoding `0x7ff0000000000000`) and $-\infty$ (encoding `0xfff0000000000000`). Infinities have special rules applied to them during arithmetic operations. Finally, the encoding has special values, each called *Not-a-Number* (NaN). Some of these are called *quiet NaNs* and are used to provide diagnostic information when unexpected conditions occur during floating-point computations. The others are called *signaling NaNs* and also provide diagnostic information but might also be used to support the needs of specialized applications. A NaN has an associated *payload* whose meaning is at the discretion of the implementer. The IEEE 754-2008 Standard has many requirements regarding the handling of NaNs in numerical computations.

The smallest positive subnormal number occurs when $e = 0$ and $t = 1$, which is $2^{e_{\min}+1-p} = 2^{-1074}$. All finite floating-point numbers are integral multiples of this number. The largest positive subnormal number occurs when $e = 0$ and t has all 1-valued bits, which is $2^{e_{\min}}(1 - 2^{1-p}) = 2^{-1022}(1 - 2^{-52})$. The smallest positive normal number occurs when $e = 1$ and $t = 0$, which is $2^{e_{\min}} = 2^{-1022}$. The largest positive normal number occurs when $e = 2046$ and t has all 1-valued bits, which is $2^{e_{\max}}(2 - 2^{1-p}) = 2^{1023}(2 - 2^{-52})$.

3 Binary Scientific Numbers

All positive numbers r of type float and double can be written in the binary scientific notation as $r = 1.c * 2^p$, where c is a finite sequence of bits c_i that are either 0 or 1 with the last bit required to be 1. The shorthand notation expands to a summation

$$r = 1.c * 2^p = \left(1 + \sum_{i=0}^{n-1} c_i 2^{-(i+1)}\right) 2^p \tag{1}$$

where $c = c_0 \ldots c_{n-1}$ has n bits and where p is an integer-valued power. If we allow for an infinite sum, replacing $n-1$ by ∞, we can represent all positive real-valued numbers. However, rational numbers require either a finite number of bits or a repeating pattern of bits. For example, in base-10 scientific notation the rational number $51/4 = 1.275 * 10^1$. The rational number $1/3 = 3.\overline{3}^{\infty} * 10^{-1}$ where the notation $\overline{3}^{\infty}$ means that the overlined block of numbers repeats ad infinitum. In binary scientific notation, $51/4 = 1.10011 * 2^3$ and $1/3 = 1.\overline{01}^{\infty} * 2^{-2}$. Irrational numbers such as $\sqrt{2}$ or π always require an infinite number of bits without a repeating block with a finite number of bits. We may keep track of an integer $\sigma \in \{-1, 0, 1\}$ that represents the sign of the number, $+1$ for a positive number, -1 for a negative number, or 0 for the number zero. When the number is zero, in which case the sign is 0, the values c and p are irrelevant and ignored.

The set B of numbers of the form $\sigma(1.c) * 2^p$ in Equation (1), and including the number zero ($\sigma = 0$ but no bits c or power p assigned to the representation) are referred to as *binary scientific numbers*. Addition, subtraction, and multiplication of numbers in B are naturally defined based on the arithmetic of the real numbers. If x and y are in B, then so are $x + y$, $x - y$, and $x * y$. However, division is not allowed. The ratio x/y of numbers in B for nonzero y is a real number of the form $1.c * 2^p$, but that number is not necessarily representable with a finite number of bits. For example, $1 = 1.0 * 2^0$ and $3 = 1.1 * 2^1$, but the ratio $1/3$ requires an infinite number of bits in its representation.

The binary scientific numbers are the basis for arbitrary precision arithmetic in GTEngine. In an implementation, we may store the quantities described next. Let $r = 1.c * 2^p$ be a binary scientific number where c has n bits c_0 through c_{n-1} with the last bit $c_{n-1} = 1$. Rewrite $r = \hat{c} * 2^{p-n} = \hat{c} * 2^{\beta}$, where $\hat{c}$ is an $(n+1)$-bit integer whose first and last bits are 1. The power β is referred to as a biased exponent. We may represent r using an integer-valued sign σ, an $(n+1)$-bit unsigned integer $\hat{c}$, and an integer biased exponent β. It is convenient to store also the number n. Addition, subtraction, or multiplication applied to two binary scientific numbers reduce to the same arithmetic operation of two unsigned integers. Manipulation of the number of bits and the biased exponents is relatively simple.

3.1 Multiplication

The product of $x = 1.u * 2^p$ and $y = 1.v * 2^q$ is $z = 1.w * 2^r$, where we need to determine the values of w and r. If $u = 0$, then $x * y = 1.v * 2^{p+q}$. If $v = 0$, then $x * y = 1.v * 2^{p+q}$. Otherwise, $u > 0$ and $v > 0$, so at least

one bit of u is not zero and at least one bit of v is not zero. Let u have $n > 0$ bits and v have $m > 0$ bits. The product is written as

$$x * y = 1.u * 2^p * 1.v * 2^q = \hat{u} * 2^{p-n} * \hat{v} * 2^{q-m} = \hat{u} * \hat{v} * 2^{p-n+q-m} = \hat{w} * 2^{p-n+q-m} \tag{2}$$

where $\hat{w} = \hat{u} * \hat{v}$ is the product of odd integers. The integer $\hat{u}$ has $n+1$ bits and the integer $\hat{v}$ has $m+1$ bits, so $\hat{w}$ is an odd integer with either $n+m+1$ or $n+m+2$ bits. For example, consider the case $n = 4$ and $m = 3$. The product of the two smallest odd integers is $10001 * 1001 = 10011001$, which has $n+m+1 = 8$ bits. The product of the two largest odd integers is $11111 * 1111 = 111010001$, which has $n+m+2 = 9$ bits.

Define $c = 0$ when the leading bit of $\hat{w}$ is at index $n+m$ or $c = 1$ when the leading bit is at index $n+m+1$, and define $\ell = n+m+c$. The integer $\hat{w}$ is an $(\ell+1)$-bit odd integer of the form $\hat{w} = 1w_0 \ldots w_{\ell-1} = 1.w_0 \ldots w_{\ell-1} * 2^\ell = 1.w * 2^\ell$ with $w_{\ell-1} = 1$. The product is

$$x * y = \hat{w} * 2^{p-n+q-m} = 1.w * 2^{p-n+q-m+\ell} = 1.w * 2^{p+q+c} = 1.w * 2^r = z \tag{3}$$

which implies $r = p+q+c$. The implementation of multiplication for BSNumber is based on performing the integer multiplication $\hat{w} = \hat{u} * \hat{v}$, selecting c by determining the index ℓ of the leading 1-bit of $\hat{w}$, and computing $r = p+q+c$, finally representing $z = 1.w * 2^r$ as the unsigned integer $\hat{w}$ and the biased exponent $\beta = r - \ell = (p-n) + (q-m)$.

3.2 Addition

The sum of $x = 1.u * 2^p$ and $y = 1.v * 2^q$ is $z = 1.w * 2^r$, where we need to determine the values of w and r. The cases $x = 0$ or $y = 0$ are trivial to handle, so assume $x > 0$ and $y > 0$ where u has n bits and v has m bits. The sum is

$$x + y = 1.u * 2^p + 1.v * 2^q = \hat{u} * 2^{p-n} + \hat{v} * 2^{q-m} \tag{4}$$

The computation depends on the relative values of $p-n$ and $q-m$.

3.2.1 The Case $p-n > q-m$

Let $p-n > q-m$. The sum is written as

$$\hat{u} * 2^{p-n} + \hat{v} * 2^{q-m} = \left(\hat{u} * 2^d + \hat{v}\right) * 2^{q-m} = \hat{w} * 2^{q-m} \tag{5}$$

where $d = (p-n) - (q-m)$. The integer $\hat{u} * 2^d$ is even because $\hat{u}$ is odd and $d > 0$. The integer $\hat{v}$ is odd. Therefore, $\hat{w}$ is an odd number. We can determine the index ℓ of the leading 1-bit of $\hat{w}$,

$$x + y = \hat{w} * 2^{q-m} = 1.w * 2^{q-m+\ell} = 1.w * 2^r = z \tag{6}$$

where $r = q-m+\ell$. The implementation of addition for BSNumber in this case involves computing the integer $\hat{u} * 2^d$ by a shift-left operation, adding the result to $\hat{v}$, determining the index ℓ of the leading 1-bit of $\hat{w}$, and finally representing $z = 1.w * 2^r$ as the unsigned integer $\hat{w}$ and the biased exponent $\beta = q-m$.

3.2.2 The Case $p-n < q-m$

Let $p-n < q-m$. The sum is written as

$$\hat{u} * 2^{p-n} + \hat{v} * 2^{q-m} = \left(\hat{u} + \hat{v} * 2^d\right) * 2^{p-n} = \hat{w} * 2^{p-n} \tag{7}$$

where $d = (q - m) - (p - n)$. The integer $\hat{v} * 2^d$ is even because $\hat{v}$ is odd and $d > 0$. The integer $\hat{u}$ is odd. Therefore, $\hat{w}$ is an odd number. We can determine the index ℓ of the leading 1-bit of $\hat{w}$,

$$x + y = \hat{w} * 2^{p-n} = 1.w * 2^{p-n+\ell} = 1.w * 2^r = z \tag{8}$$

where $r = p - n + \ell$. The implementation of addition for BSNumber in this case involves computing the integer $\hat{v} * 2^d$ by a shift-left operation, adding the result to $\hat{u}$, determining the index ℓ of the leading 1-bit of $\hat{w}$, and finally representing $z = 1.w * 2^r$ as the unsigned integer $\hat{w}$ and the biased exponent $\beta = p - n$.

3.2.3 The Case $p - n = q - m$

Let $p - n = q - m$. The sum is written as

$$\hat{u} * 2^{p-n} + \hat{v} * 2^{q-m} = (\hat{u} + \hat{v}) * 2^{q-m} = \tilde{w} * 2^{q-m} \tag{9}$$

The integers $\hat{u}$ and $\hat{v}$ are odd, so $\tilde{w}$ is even. Let ℓ be the index of the leading 1-bit of $\tilde{w}$ and let t be the index of the trailing 1-bit of $\tilde{w}$. It is necessary that $\ell \geq t > 0$. We can shift-right $\tilde{w}$ by t bits to obtain an odd integer $\hat{w}$ whose leading 1-bit is at index $f = \ell - t$; then

$$x + y = \tilde{w} * 2^{q-m} = \hat{w} * 2^{q-m+t} = 1.w * 2^{q-m+t+f} = 1.w * 2^{q-m+\ell} = 1.w * 2^r = z \tag{10}$$

where $r = q - m + \ell$. The fractional part w (as an integer) has $\ell - t$ bits. The implementation of addition for BSNumber in this case involves adding the integers $\hat{u}$ and $\hat{v}$ to obtain $\tilde{w}$, determining the indices ℓ and t, shifting right $\tilde{w}$ by t bits to obtain $\hat{w}$, and finally representing $z = 1.w * 2^r$ as the unsigned integer $\hat{w}$ and the biased exponent $\beta = q - m + t$.

3.2.4 Determining the Maximum Number of Bits for Addition

We may concisely write the sum $x + y = z = 1.w * 2^r$. The implementation of BSNumber stores the biased exponent β, the integer $\hat{w}$, and the number of bits in $\hat{w}$. In all three cases, we have a summation of integers. When adding a k_0-bit positive integer and a k_1-bit positive integer, the maximum number of bits required for the sum is $\max\{k_0, k_1\} + 1$. The extra bit occurs only when there is a carry-out by the addition. For example, 11100 is a 5-bit integer and 11 is a 2-bit integer. The sum is $11100 + 11 = 11111$, which is a 5-bit integer. However, 111 is a 3-bit integer, so the sum $11100 + 111 = 100011$, which is a 6-bit integer; there is a carry-out in this example.

By definition we know that $\hat{u}$ has $n+1$ bits and $\hat{v}$ has $m+1$ bits. In the case $p-n > q-m$, $\hat{u} * 2^{(p-n)-(q-m)}$ is an even integer with $n + 1 + (p - n) - (q - m)$ bits. Adding this to $\hat{v}$ to produce $\hat{w}$, the maximum number of bits required is $\max\{n+1+(p-n)-(q-m), m+1\}+1$. Similarly in the case $p-n < q-m$, the number of bits for $\hat{v} * 2^{(q-m)-(p-n)}$ is $m + 1 + (q - m) - (p - n)$. The integer $\hat{w}$ requires at most $\max\{n+1, m+1+(q-m)-(p-n)\}+1$ bits. Finally, in the case $p-n = q-m$ we must first compute $\tilde{w} = \hat{u} + \hat{v}$ before we can determine ℓ and t. The maximum number of bits for $\tilde{w}$ is $\max\{n+1, m+1\}+1$.

The maximum number of bits for the three cases was determined algebraically. A more intuitive argument is the following. Imagine marking the integer points on the real line at which the consecutivie bits of $x = 1.u*2^p$ are located. The largest marked integer is p, which corresponds to $1.0 * 2^p$, the highest-order bit of x. The smallest marked integer is $p - n$, where u is an odd integer with n bits, and which corresponds to $1 * 2^{p-n}$, the lowest-order bit of x (and of u). The same argument applies to $y = 1.v * 2^q$. The highest-order bit occurs

at q and the lowest-order bit occurs at $q-m$, where v is an odd integer with m bits. The bit locations of $x+y = 1.w * 2^r$ are in the union of the bit locations for x and y, plus one more at the highest-order end in case the addition has a carry-out. Therefore, the maximum number of bits required for the sum is

$$\ell = \max\{p, q\} - \min\{p-n, q-m\} + 2 \tag{11}$$

The right-hand side involves the maximum of exponents and the minimum of biased exponents. When $p - n > q - m$,

$$\begin{aligned}
\ell &= \max\{p, q\} - (q-m) + 2 \\
&= \max\{p - q + m + 1, m + 1\} + 1 \\
&= \max\{n + 1 + (p-n) - (q-m), m+1\} + 1
\end{aligned}$$

which agrees with the previous construction. When $p - n \leq q - m$,

$$\begin{aligned}
\ell &= \max\{p, q\} - (p-n) + 2 \\
&= \max\{n+1, q - p + n + 1\} + 1 \\
&= \max\{n+1, m + 1 + (q-m) - (p-n)\} + 1
\end{aligned}$$

which agrees with the previous construction. In particular, when $p - n = q - m$,

$$\begin{aligned}
\ell &= \max\{p, q\} - (p-n) + 2 \\
&= \max\{n+1, m + 1 + (q-m) - (p-n)\} + 1 \\
&= \max\{n+1, m+1\} + 1
\end{aligned}$$

which agrees with the previous construction.

3.3 Subtraction

We wish to compute $x - y$. The cases $x = 0$ and $y = 0$ are trivial to handle. It is sufficient to analyze the case $x > y > 0$; when $0 < x < y$, we know $x - y = -(y - x)$, which leads to the case $y - x$ with $y > x > 0$. The difference of $x = 1.u * 2^p$ and $y = 1.v * 2^q$ is $z = 1.w * 2^r$, where we need to determine the values of w and r. Let u have n bits and v have m bits. The difference is

$$x - y = 1.u * 2^p - 1.v * 2^q = \hat{u} * 2^{p-n} - \hat{v} * 2^{q-m} \tag{12}$$

The factoring depends on the relative values of $p - n$ and $q - m$, just as it did for addition.

Handling the various cases is the same as for addition, but we need to subtract two integers, the first larger than the second. Our choice for subtraction is to use two's-complement, the same as for native integers on the CPU. For example, consider the subtraction $19 = 29 - 10 = 29 + (-10)$. In binary notation, $29 = 11101$ and $10 = 01010$, using the same number of bits for both numbers. To obtain the two's complement of a number, you negate the bits and add one to the number. For the example at hand, $-10 = \overline{01010} + 1 = 10101 + 1 = 10110$ and $29 + (-10) = 11101 + 10110 = 110011$. Because we always choose the same number of bits for both arguments, there will always be a carry-out. This bit must be discarded, so $29 + (-10) = 10011 = 19$.

The subtraction result is no larger than x, so it has at most the number of bits of x. However, to support two's complement and the carry-out bit, we need an extra bit; thus, the number of required bits is $n + 2$ where x has $n + 1$ bits.

3.4 Unsigned Integer Arithmetic

In each of multiplication, addition, and subtraction of binary scientific numbers, determining the sign and biased exponent is simple. The heart of the computation reduces to the same operations applied to unsigned integers with an arbitrary number of bits.

One of the primary concerns when implementing arbitrary precision integer arithmetic is dealing with carries and overflow. A typical way to handle this is to embedded the N-bit unsigned integer operands in a native type with $2N$ bits. For now, we assume that the CPU supports a 64-bit unsigned integer type uint64_t, which is defined in the header <cstdint>. We may write the unsigned integers

$$\hat{u} = \sum_{i=0}^{i_{\max}-1} u_i (2^{32})^i, \quad \hat{v} = \sum_{j=0}^{j_{\max}-1} v_j (2^{32})^j \tag{13}$$

where u_i and v_j are 32-bit unsigned integers.

3.4.1 Addition

The straightforward algorithm for addition is illustrated for $i_{\max} = 2$ and $j_{\max} = 1$, computing $\hat{w} = \hat{u} + \hat{v}$ using the old-style notation of stacking the numbers and notating it with carry values. A specific example is shown on the left, the general layout on the right.

$$\begin{array}{cccc} {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & \\ \hdashline & 9 & 2 & 5 \\ & & 8 & 3 \\ \hline 1 & 0 & 0 & 8 \end{array} \qquad \begin{array}{cccc} {\scriptstyle c_3} & {\scriptstyle c_2} & {\scriptstyle c_1} & \\ \hdashline & u_2 & u_1 & u_0 \\ & & v_1 & v_0 \\ \hline w_3 & w_2 & w_1 & w_0 \end{array} \qquad \begin{array}{l} \text{carry bits} \\ \hat{u} \\ \hat{v} \\ \hat{u} + \hat{v} \end{array} \tag{14}$$

There is no carry-in value for the first term, but we can think of $c_0 = 0$. Add u_0, v_0, and the carry-in, say, $p_0 = u_0 + v_0 + c_0$. The operands are of type uint32_t and, in worst case, the output p_0 must be stored in uint64_t because its value is larger than 2^{32}. The low-order 32 bits is extracted to obtain $w_0 = \text{Low}(p_0)$ and the high-order 32 bits is extracted to obtain the carry-out $c_1 = \text{High}(p_0)$. The process is repeated for the next u-term, but now the carry value can be positive: $p_1 = u_1 + v_1 + c_1$, $w_1 = \text{Low}(p_1)$, and $c_2 = \text{High}(p_1)$. There are no more v-terms. Repeating the algorithm produces $p_2 = u_2 + c_2$, $w_2 = \text{Low}(p_2)$, and $c_3 = \text{High}(p_2)$. There are no more u-terms, so $w_3 = c_3$.

Pseudocode for the general algorithm is shown in Listing 3, where the assumption is that u[imax] and v[jmax] each contain at least one 1-bit. Also, let $i_{\max} \geq j_{\max}$.

Listing 3. Standard addition of unsigned integers.

```
// inputs
int32_t imax, jmax;
uint32_t u[imax], v[jmax];

// outputs
int32_t kmax = imax + 1; // = max(imax, jmax) + 1;
uint32_t upv[kmax];   // u + v
```

```
uint64_t carry = 0, sum;
int32_t i, j;
for (j = 0; j < jmax; ++j)
{
    sum = u[j] + (v[j] + carry);
    upv[j] = (uint32_t)(sum & 0x00000000FFFFFFFF);
    carry = (sum >> 32);
}

// We have no more v-blocks. Propagate the carry-out if there is one or
// copy the remaining blocks if there is not.
if (carry > 0)
{
    for (/**/; i < imax; ++i)
    {
        sum = u0[i] + carry;
        upv[i] = (uint32_t)(sum & 0x00000000FFFFFFFF);
        carry = (sum >> 32);
    }
    if (carry > 0)
    {
        upv[i] = (uint32_t)(carry & 0x00000000FFFFFFFF);
    }
}
else
{
    for (/**/; i < imax; ++i)
    {
        upv[i] = u[i];
    }
}
```

3.4.2 Subtraction

Subtraction of two unsigned integers uses two's-complement arithmetic, $\hat{u}-\hat{v}$, and we assume that $\hat{u} > \hat{v} > 0$. The result is computed as $\hat{u} + (\sim \hat{w} + 1)$, where $\hat{w}$ equal to $\hat{v}$ but has the same number of bits as $\hat{u}$ (think of paddding $\hat{v}$ on the left by the appropriate number of zeros) and where $\sim \hat{w}$ is the number obtained by negating the bits of $\hat{w}$—a 0-bit is changed to a 1-bit and vice versa. We know that $\hat{v} \neq 0$; thus, $\hat{w} \neq 0$, $\sim \hat{w}$ cannot have all bits equal to 1, and the addition $\sim \hat{w} + 1$ cannot have a carry-out. Once that sum is computed, the sum $\hat{u} + (\sim \hat{w} + 1)$ is computed using addition of unsigned integers. This sum can have a carry-out, but that bit is irrelevant to the result and can be ignored.

Pseudocode for the general algorithm is shown in Listing 4, where the assumption is that u[imax] and v[jmax] each contain at least one 1-bit. Also, let $i_{\max} \geq j_{\max}$.

Listing 4. Two's-complement subtraction of unsigned integers.

```
// inputs
int32_t imax, jmax;
uint32_t u[imax], v[jmax];

// outputs
int32_t kmax = imax + 1; // = max(imax, jmax) + 1;
uint32_t umv[imax];   // u - v

// Create the two's-complement number compV[] from v[].  Firstly, negate
```

```
// the bits; secondly, add 1.
uint32_t compV[imax];
uint64_t carry, sum;
int32_t i;
for (i = 0; i < jmax; ++i)
{
    compV[i] = ~v[i];
}
for (/**/; i < imax; ++i)
{
    compV[i] = 0xFFFFFFFF;
}
carry = 1;
for (i = 0; i < jmax; ++i)
{
    sum = compV[i] + carry;
    compV[i] = (uint32_t)(sum & 0x00000000FFFFFFFF);
    carry = (sum >> 32);
}

// Add the numbers as positive integers.  Set the last block to zero in
// case no carry-out occurs.
umv[imax - 1] = 0;
carry = 0;
for (i = 0; i < imax; ++i)
{
    sum = compV[i] + (u[i] + carry);
    umv[i] = (uint32_t)(sum & 0x00000000FFFFFFFF);
    carry = (sum >> 32);
}
```

The subtraction can lead to umv[] having leading zero-valued blocks. The implementation enforces an invariant that the leading block contains at least one 1-bit. The array umv[] is resized to eliminate the leading zero-valued blocks.

3.4.3 Multiplication

The straightforward algorithm for multiplication is illustrated for $i_{\max} = 0$ and $j_{\max} = 2$, computing $\hat{w} = \hat{u} * \hat{v}$ using the old-style notation of stacking the numbers and notating it with carry values. A specific example is shown on the left, the general layout on the right.

$$
\begin{array}{cccc}
{\scriptstyle 2} & {\scriptstyle 1} & {\scriptstyle 3} & \\
\hdashline
 & 3 & 2 & 5 \\
 & & & 7 \\
\hline
2 & 2 & 7 & 5
\end{array}
\qquad
\begin{array}{cccc}
{\scriptstyle c_3} & {\scriptstyle c_2} & {\scriptstyle c_1} & \\
\hdashline
 & v_2 & v_1 & v_0 \\
 & & & u_0 \\
\hline
w_3 & w_2 & w_1 & w_0
\end{array}
\qquad
\begin{array}{l}
\text{carry bits} \\
\hat{v} \\
\hat{u} \\
\hat{u} * \hat{v}
\end{array}
\tag{15}
$$

There is no carry-in value for the first term, but we can think of $c_0 = 0$. Multiply v_0 by u_0 and add the carry-in, say, $p_0 = u_0 * v_0 + c_0$. The operands are of type uint32_t and, in worst case, the output p_0 must be stored in uint64_t because its value is larger than 2^{32}. The low-order 32 bits is extracted to obtain $w_0 = \text{Low}(p_0)$ and the high-order 32 bits is extracted to obtain the carry-out $c_1 = \text{High}(p_0)$. The process is repeated for the next v-term, but now the carry value can be positive: $p_1 = u_0 * v_1 + c_1$, $w_1 = \text{Low}(p_1)$, and $c_2 = \text{High}(p_1)$. One more repetition produces $p_2 = u_0 * v_2 + c_2$, $w_2 = \text{Low}(p_2)$, and $c_3 = \text{High}(p_2)$. There are no more v-terms, so $w_3 = c_3$.

Now consider the case $i_{\max} = 1$. A specific example is shown on the left, the general layout on the right.

$$
\begin{array}{ccccccccccccl}
 & 1 & 1 & 2 & & \quad & & d_3 & d_2 & d_1 & & \quad & \text{Carry bits for } \hat{q} = u_1 * \hat{v} \\
\hdashline
 & 2 & 1 & 3 & & & & c_3 & c_2 & c_1 & & & \text{Carry bits for } \hat{w} = u_0 * \hat{v} \\
\hdashline
 & & 3 & 2 & 5 & & & & v_2 & v_1 & v_0 & & \hat{v} \\
 & & & 5 & 7 & & & & & u_1 & u_0 & & \hat{u} \\
\hline
0 & 0 & 1 & & & & e_3 & e_2 & e_1 & & & & \text{Carry bits for } \hat{w} + \hat{q} * 2^{32} \\
\hdashline
 & 2 & 2 & 7 & 5 & & & w_3 & w_2 & w_1 & w_0 & & \hat{w} = u_0 * \hat{v} \\
1 & 6 & 2 & 5 & & & q_3 & q_2 & q_1 & q_0 & & & \hat{q} = u_1 * \hat{v} \\
\hline
1 & 8 & 5 & 2 & 5 & & s_4 & s_3 & s_2 & s_1 & s_0 & & \hat{u} * \hat{v}
\end{array}
\tag{16}
$$

As in the previous example, the initial carry-in values are $c_0 = 0$, $d_0 = 0$, and $e_0 = 0$. The product of u_0 with v is computed by $p_j = u_0 * v_j + c_j$, $w_j = \text{Low}(p_j)$, and $c_{j+1} = \text{High}(p_j)$. The product of u_1 with v is computed by $p_j = u_1 * v_j + d_j$, $q_j = \text{Low}(p_j)$, and $d_{j+1} = \text{High}(p_j)$. The sum of products is computed by $w_4 = 0$, $s_0 = w_0$, $p_j = w_{j+1} + q_j + e_j$, $s_{j+1} = \text{Low}(p_j)$, and $e_{j+1} = \text{High}(p_j)$.

Pseudocode for the general algorithm is shown in Listing 5, where the assumption is that u[imax] and v[jmax] each contain at least one 1-bit.

Listing 5. Standard multiplication of unsigned integers.

```
// inputs
int32_t imax, jmax;
uint32_t u[imax], v[jmax];

// outputs
int32_t kmax = imax + jmax;
uint32_t utv[kmax+1];  // u * v

SetToZero(utv, kmax);  // uv is the accumulator of u[i]*v
uint32_t product[kmax];
for (i = 0; i < imax; ++i)
{
    // Compute the product p = u[i]*v.
    uint64_t uBlock = u[i], carry = 0;
    int32_t i1, i2;
    for (j = 0, k = i; j < jmax; ++j, ++k)
    {
        uint64_t vBlock = v[j];
        uint64_t term = uBlock * vBlock + carry;
        product[k] = (uint32_t)(term & 0x00000000FFFFFFFF);
        carry = (term >> 32);
    }
    if (k < kmax)
    {
        product[k] = (uint32_t)carry;
    }

    // Add p to the accumulator uv.
    uint64_t sum;
    carry = 0;
    for (j = 0, k = i; j < jmax; ++j, ++k)
    {
        sum = product[k] + (utv[k] + carry);
        utv[k] = (uint32_t)(sum & 0x00000000FFFFFFFF);
```

```
        carry = (sum >> 32);
    }
    if (k < kmax)
    {
        sum = product[k] + carry;  // utv[k] == 0 is guaranteed
        utv[k] = (uint32_t)(sum & 0x00000000FFFFFFFF);
    }
}
```

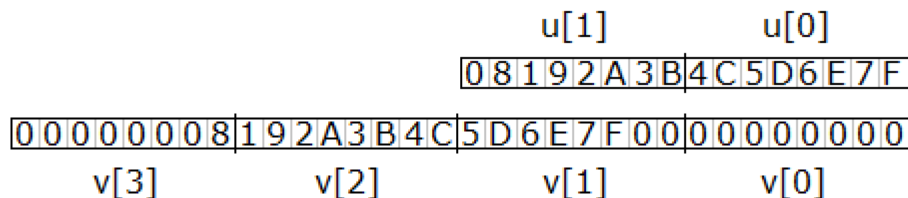
For inputs with a large number of bits, the initialization of utv[] to zero is expensive (based on analysis with a profiler). The GTEngine implementation avoids this with some extra logic, setting the minimum number of required elements of utv[] to u[0]*v on the first pass and then accumulating the remaining u[i]*v products on subsequent passes.

3.4.4 Shift Left

The algorithms for addition and subtraction involved computing $\hat{v} = \hat{u} * 2^{\sigma}$, where $\hat{u}$ is an odd integer and $\sigma > 0$. This is equivalent to applying a left shift of the bits of $\hat{u}$ by σ bits.

To illustrate, let $\hat{u} =$ 0x08192A3B4C5D6E7F and $\sigma = 40$. The input requires 2 32-bit blocks, is odd, and has 60 bits starting with the leading 1-bit at index 59. The trailing 1-bit, of course, is at index 0. The left shift by σ moves the leading 1-bit to index 99, so $\hat{v}$ requires 4 32-bit blocks. Also, $\hat{v}$ has 1 trailing zero-valued block because $\sigma/32 = 1$ (using integer division); thus, v[0] = 0. The trailing 1-bit of $\hat{v}$ is at index $\sigma\%32 = 8$ in v[1]. The leading 1-bit is in v[3]. Figure 3 illustrates the input and output.

Figure 3. Left shift of an unsigned integer by 40.



Pseudocode for the general algorithm is shown in Listing 6.

Listing 6. Shift left of unsigned integers.

```
// inputs
int32_t numUBits = <number of bits of u>;
int32_t numShiftedUBits = numUBits + shift;
int32_t imax = 1 + (numUBits - 1) / 32;
uint32_t u[imax];

// outputs
int32_t jmax = 1 + (numShiftedUBits - 1) / 32;
```

```
uint32_t shiftedU[jmax];  // (u << shift)

// Set the low-order bits to zero.
int32_t shiftBlock = shift / 32;
int32_t i, j;
for (j = 0; j < shiftBlock; ++j)
{
    shiftedU[j] = 0;
}

// Get the index of the trailing 1-bit within the result.
int32_t leftShift = shift % 32;
if (leftShift > 0)
{
    // The trailing 1-bits for source and target are at different relative
    // indices.  Each shifted source block straddles a boundary between two
    // target blocks, so we must extract the subblocks and copy accordingly.
    int32_t rightShift = 32 - leftShift;
    uint32_t previous = 0;
    for (j = shiftBlock, i = 0; i < imax; ++j, ++i)
    {
        uint32_t current = u[i];
        shiftedU[j] = (current << leftShift) | (previous >> rightShift);
        prev = current;
    }
    if (j < jmax)
    {
        // The leading 1-bit of the source is at a relative index such that
        // when you left-shift, that bit occurs in a new block.
        shiftedU[j] = (previous >> rightShift);
    }
}
else
{
    // The trailing 1-bits for source and target are at the same relative
    // index.  The shift reduces to a block copy.
    for (j = shiftBlock, i = 0; i < imax; ++j, ++i)
    {
        shiftedU[j] = u[i];
    }
}
```

In the example of Figure 3, $i_{\max} = 2$ and $j_{\max} = 4$. The left shift is 8, which is positive. In the block of code handling this case, the loop terminates with $j = 7$. The if statement after that loop is executed. This indicates that the leading 1-bit of $\hat{u}$ was relatively shifted out of the 32-bit block that contained it into the left-adjacent block. If instead $\sigma = 36$, we would have found that $j_{\max} = 3$. The leading 1-bit would have stayed in the block and the loop would exit with $j = 3$, in which case the if statement would not be executed. When the left shift is 0, the relative location of the source's leading 1-bit and the target's leading 1-bit are the same, in which case no subblock shifting needs to be performed.

3.4.5 Shift Right to Odd Number

A special case in the algorithms for addition and subtraction involved computing the sum of two odd integers or the difference of two odd integers where the first integer is larger than the second. In both cases the result is a positive even integer. We need to apply a shift right of the bits to obtain an odd number, and we need the shift amount for computing the biased exponents of the BSNumber.

The example in the section on left shifting can be used to illustrate right shifting, but with $\hat{v}$ as the input

and $\hat{u}$ as the output. The algorithm is similar to that of left shifting, except that the direction of subblock shifts is swapped and we determine the actual shift amount knowing that we need to obtain an odd integer.

Pseudocode for the general algorithm is shown in Listing 7.

Listing 7. Shift right of an unsigned integer to create an odd number.

```
// inputs
int32_t numUBits = <number of bits of u>;
int32_t imax = 1 + (numUBits - 1) / 32;
uint32_t u[imax];

// Get the leading 1-bit.
int32_t firstBitIndex = 32 * (imax - 1) + GetLeadingBit(u[imax - 1]);

// Get the trailing 1-bit.
int32_t i, j, lastBitIndex;
for (i = 0; i < imax; ++i)
{
    if (u[i] > 0)
    {
        lastBitIndex = 32 * i + GetTrailingBit(u[i]);
        break;
    }
}

// outputs
int32_t numShiftedUBits = firstBitIndex - lastBitIndex + 1;
int32_t jmax = 1 + (numShiftedUBits - 1) / 32;
uint32_t shiftedU[jmax];  // (u << shift)
int32_t shift;

self.SetNumBits(firstBitIndex - lastBitIndex + 1);
auto& bits = self.GetBits();
int32_t const numBlocks = self.GetSize();

// Get the location of the low-order 1-bit within the result.
int32_t shiftBlock = lastBitIndex / 32;
int32_t rshift = lastBitIndex % 32;
if (rshift > 0)
{
    int32_t lshift = 32 - rshift;
    i = shiftBlock;
    uint32_t curr = u[i++];
    for (j = 0; i < imax; ++j, ++i)
    {
        uint32_t next = u[i];
        shiftedU[j] = (curr >> rshift) | (next << lshift);
        curr = next;
    }
    if (j < jmax)
    {
        shiftedU[j] = (curr >> rshift);
    }
}
else
{
    for (j = 0, i = shiftBlock; j < jmax; ++j, ++i)
    {
        shiftedU[j] = u[i];
    }
}

shift = rshift + 32 * shiftBlock;
```

3.4.6 Comparisons

The comparison operations are based on the unsigned integers being members of the binary scientific representation. They are called only when the two BSNumber arguments to BSNumber::operatorX are of the form $1.u * 2^p$ and $1.v * 2^p$. The less-than comparison applies to $1.u$ and $1.v$ as unsigned integers with their leading 1-bits aligned.

Pseudocode for the equality comparison is shown in Listing 8.

Listing 8. Equality comparison of two UInteger objects.

```
bool Equal(UInteger u[imax], UInteger v[jmax])
{
    if (u.numBits != v.numBits)
    {
        return false;
    }

    if (u.numBits > 0)
    {
        for (int32_t i = imax-1; i >= 0; --i)
        {
            if (u[i] != v[i])
            {
                return false;
            }
        }
    }
    return true;
}
```

Pseudocode for the less-than comparison is shown in Listing 9.

Listing 9. Less-than comparison of two UInteger objects.

```
bool LessThan(UInteger u[imax], UInteger v[jmax]) const
{
    if (u.numBits > 0 && v.numBits > 0)
    {
        // The numbers must be compared as if their leading 1-bits are aligned.
        int bitIndex0 = u.numBits - 1;
        int bitIndex1 = v.umBits - 1;
        int block0 = bitIndex0 / 32;
        int block1 = bitIndex1 / 32;
        int numBlockBits0 = 1 + (bitIndex0 % 32);
        int numBlockBits1 = 1 + (bitIndex1 % 32);
        uint64_t n0shift = u[block0];
        uint64_t n1shift = v[block1];
        while (block0 >= 0 && block1 >= 0)
        {
            // Shift the bits in the leading blocks to the high-order bit.
            uint32_t value0 =
                (uint32_t)((n0shift << (32 - numBlockBits0)) & 0x00000000FFFFFFFF);
            uint32_t value1 =
                (uint32_t)((n1shift << (32 - numBlockBits1)) & 0x00000000FFFFFFFF);
```

```
            // Shift bits in the next block (if any) to fill the current
            // block.
            if (--block0 >= 0)
            {
                n0shift = bits[block0];
                value0 |=
                    (uint32_t)((n0shift >> numBlockBits0) & 0x00000000FFFFFFFF);
            }
            if (--block1 >= 0)
            {
                n1shift = nBits[block1];
                value1 |=
                 (uint32_t)((n1shift >> numBlockBits1) & 0x00000000FFFFFFFF);
            }
            if (value0 < value1)
            {
                return true;
            }
            if (value0 > value1)
            {
                return false;
            }
        }
        return block0 < block1;
    }
    else
    {
        // One or both numbers are negative.  The only time 'less than' is
        // 'true' is when v is positive.
        return (v.numBits > 0);
    }
}
```

The other comparisons may be implemented based on equality and less-than comparisons.

3.5 Conversion of Floating-Point Numbers to Binary Scientific Numbers

The conversion algorithm is described for type float. The algorithm for type double is similar. In fact, the implementation uses a template function that handles either type.

Listing 1 contains a skeleton for the various flavors of float numbers. The zeros $+0$ and -0 both map to the same BSNumber, the one whose sign is zero, whose biased exponent is zero, and whose unsigned integer is zero. In the following discussion, it is clear how to extract the sign from a floating-point number and set the binary scientific number sign, so we will consider only positive numbers.

The subnormals are of the form $0.t * 2^{-126}$, where $t > 0$ is the trailing significand with 23 bits. Let the first 1-bit of t occur at index f and let the last 1-bit occur at index ℓ; then t has $f - \ell + 1$ bits. The biased exponent is $\beta = -149 + \ell$. Thus, $0.t * 2^{-126} = \hat{u} * 2^p$, where $u = t_f t_{f-1} \cdots t_\ell$ and $p = -149 + f$. Pseudocode is shown in Listing 10.

Listing 10. Conversion of a subnormal float to a BSNumber.

```
float subnormal;
BSNumber bsn;
uint32_t s = subnormal.GetSign();
```

```
uint32_t e = subnormal.GetBiasedExponent();  // e = 0 for subnormals
uint32_t t = subnormal.GetTrailingSignificand();
int32_t last = GetTrailingBit(t);
int32_t diff = 23 - last;
bsn.sign = (s > 0 ? -1 : 1);
bsn.biasedExponent = -126 - diff;
bsn.uinteger = (t >> last);
```

The normals are of the form $1.t * 2^{e-127}$, where $t \geq 0$ is the trailing significand with 23 bits. If $t = 0$, the biased exponent is $\beta = e - 127$, so $1.t * 2^{e-127} = 1 * 2^{e-127} = \hat{u} * 2^p$, where $\hat{u} = 1$ and $p = e - 127$. If $t > 0$, let the last 1-bit of t occur at index ℓ. The leading 1-bit of the floating-point number is implicitly 1; that is, when you extract the trailing significand from the encoding, you will obtain t as an integer with 23 bits. You must append a 1 using an OR-operation at index 23. The first 1-bit of $1t$ occurs at index $f = 23$. The number of bits is $f - \ell + 1 = 24 - \ell$. The biased exponent is $e - 149 + \ell$. Thus, $1.t * 2^{e-127} = \hat{u} * 2^p$, where $u = 1t_{22} \cdots t_\ell$ and $p = e - 150 + \ell$. Pseudocode is shown in Listing 11.

Listing 11. Conversion of a normal float to a BSNumber.

```
float normal;
BSNumber bsn;
uint32_t s = normal.GetSign();
uint32_t e = normal.GetBiasedExponent();  // 0 < e < 255 for normals
uint32_t t = normal.GetTrailingSignificand();
if (t > 0)
{
    int32_t last = GetTrailingBit(t);
    int32_t diff = 23 - last;
    bsn.sign = (s > 0 ? -1 : 1);
    bsn.biasedExponent = e - 127 - diff;
    bsn.uinteger = ((t | (1 << 23)) >> last);
}
else
{
    bsn.sign = (s > 0 ? -1 : 1);
    bsn.biasedExponent = e - 127;
    bsn.uinteger = 1;
}
```

In practice you will use the class BSNumber only for finite floating-point numbers, so you should ensure that your inputs you manipulate are not infinities or NaNs. In our implementation, if the floating-point number is an infinity, we convert it to $\pm 2^{128}$; the sign is chosen to be that of the floating-point number. However, we do have a warning assertion that is triggered in our logging system that lets you know when such a number is encountered. If we encounter a NaN, we convert the number to zero and issue an error assertion. Pseudocode is shown in Listing 12.

Listing 12. Conversion of an infinity or NaN float to a BSNumber.

```
float special;  // infinity, NaN
BSNumber bsn;
uint32_t s = special.GetSign();
uint32_t e = special.GetBiasedExponent();  // e = 255 for specials
uint32_t t = special.GetTrailingSignificand();
if (t == 0)  // infinities
{
    // Warning: Input is an infinity.
    bsn.sign = (s > 0 ? -1 : 1);
    bsn.biasedExponent = 128;
    bsn.uinteger = 1;
}
else
{
    // Error: Input is a NaN (quiet or silent).
    bsn.sign = 0;
    bsn.biasedExponent = 0;
    bsn.uinteger = 0;
}
```

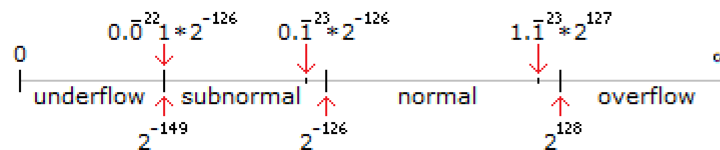
3.6 Conversion of Binary Scientific Numbers to Floating-Point Numbers

The conversion algorithm is described for type float. The algorithm for type double is similar. In fact, the implementation uses a template function that handles either type.

The conversion from a BSNumber to a float is generally not exact. Typically, a sequence of operations with binary scientific numbers will produce a result with more bits than the 24 bits of precision for float. We choose to convert the number using the default rounding mode for floating-point arithmetic: round-to-nearest, ties-to-even. Once again it is easy to handle the sign information, so the discussion is about positive binary scientific numbers.

Figure 4 illustrates the real number line and the location of floating-point numbers on it.

Figure 4. The real number line and locations of some float numbers.



Underflow occurs in the interval of real numbers $(0, 2^{-149})$, which cannot be represented by float numbers. Subnormals occur in the interval of real numbers $[2^{-149}, 2^{-126})$. Normals occur in the interval of real numbers $[2^{-126}, 2^{128})$. Overflow occurs in the interval of real numbers $[2^{128}, \infty)$, which cannot be represented by float numbers. The floating-point number $0.\overline{0}^{22}1 * 2^{-126} = 2^{-149}$ is the smallest subnormal. The notation $\overline{0}^{22}$ means that 0 is repeated 22 times. The largest subnormal is $0.\overline{1}^{23} * 2^{-126}$. The smallest normal is 2^{-126} and the largest normal is $1.\overline{1}^{23} * 2^{127}$.

Let x be a BSNumber in the interval $[0, 2^{-149})$. The two closest float numbers are the interval endpoints.

The midpoint of the interval is 2^{-150}. If $x \leq 2^{-150}$, we round to zero. If $x > 2^{-150}$, we round to 2^{-149}. Pseudocode is shown in Listing 13.

Listing 13. Conversion of a very small positive BSNumber to float.

```
BSNumber bsn;  // assume bsn is not zero
uint32_t s = (bsn.sign < 0 ? 1 : 0);
uint32_t e;  // biased exponent for float
uint32_t t;  // trailing significand for float
int32_t p = bsn.GetExponent();  // biasedExponent + numBits − 1
if (p < −149)
{
    if (p < −150 || bsn.uinteger == 1)
    {
        // round to 0 (when bsn.uinteger is 1, this is a tie, so round to even)
        e = 0;
        t = 0;
    }
    else
    {
        // round to 2^{−249}
        e = 0;
        t = 1;
    }
}
// else: other cases discussed later

float result = CreateIEEEFloat(s, e, t);  // as uint32_t, (s << 31) | (e << 23) | t
```

Let $x = 1.u * 2^p$ be a BSNumber in the interval $[2^{-149}, 2^{-126})$, so $-149 \leq p < -126$. Using the precision of x, we can write

$$x = \left(0.1u * 2^{-\sigma}\right) * 2^{-126} = 0.\overline{0}^{|\sigma|}1u * 2^{-126} = 0.\hat{t} * 2^{-126} \tag{17}$$

where $\sigma = -(p + 127) \in \{0, \ldots, 22\}$ and where $\hat{t} = \overline{0}^{|\sigma|}1u$ is an integer starting with zero or more 0-bits and ending with the bits of $\hat{u} = 1u$.

We need to generate a 23-bit trailing significand t from $\hat{t}$ in order to create a float subnormal $y = 0.t * 2^{-126}$. We can do so by extracting the first 24 bits of $\hat{t}$. If you choose only the first 23, the subnormal y is obtained by truncation. However, we need to apply the mode round-to-nearest-ties-to-even. Generally, consider applying the rounding mode to the nonnegative number $i.f$, where i is the integer part and f is the fractional part. If $f < 1/2$, round to i. If $f > 1/2$, round to $i + 1$. When $f = 1/2$, round to i when i is even or round to $i + 1$ when i is odd. The heart of the rule is the comparison of f to $1/2$. In our algorithm the first 23 bits form the integer i. We choose only 1 additional bit to guide the rounding. At first glance this might seem insufficient to determine information about the f. In fact it is enough because we know that the last bit of u is a 1-bit. If the additional bit is 0, we round down because $f < 1/2$. If the additional bit is 1, $f > 1/2$ when that bit is not the last bit of u, in which case we round up. If that bit is the last bit of u, then $f = 1/2$ and we round down when the previous bit is 0 or round up when the previous bit is 1.

Table 1 shows the information that is computed in the code to determine the trailing significand and rounding. The bits extracted from $\hat{u}$ are stored in a uint32_t *prefix* and are left-aligned to start at bit-index 31 of the prefix.

Table 1. Layouts for the bits of a BSNumber that produces a float subnormal.

σ	$0.\hat{t}$	bits to extract from $\hat{u}$	prefix index of first bit of f
0	$0.1u$	24	8
1	$0.01u$	23	9
2	$0.001u$	22	10
$\vdots$	$\vdots$	$\vdots$	$\vdots$
22	$0.00000000000000000000001u$	2	30

Pseudocode is shown in Listing 14.

Listing 14. Conversion of a BSNumber to a subnormal float.

```
BSNumber bsn;  // assume bsn is not zero
uint32_t s = (bsn.sign < 0 ? 1 : 0);
uint32_t e;  // biased exponent for float
uint32_t t;  // trailing significand for float
int32_t p = bsn.GetExponent();  // biasedExponent + numBits - 1
if (-149 <= p && p < -126)
{
    int32_t sigma = -(p + 127);
    t = bsn.GetSubnormalTrailing(sigma);
    if (t & (1 << 23))
    {
        // The first 23 bits of hat{u} were all 1 and the remaining bits
        // led to rounding up, so round up to the minimum normal.
        e = 1;
        t = 0;
    }
    else
    {
        e = 0;
    }
}
// else: other cases discussed later

float result = CreateIEEEFloat(s, e, t);  // as uint32_t, (s << 31) | (e << 23) | t
```

The construction of the 23-bit trailing significand is shown in Listing 15.

Listing 15. Construction of the trailing significand for the subnormal.

```
uint32_t BSNumber::GetSubnormalTrailing(int32_t sigma)
{
    // Extract leading bits from hat{u}, shifted so first one is at index 31 of prefix.
    int32_t numRequested = 24 - sigma;
```

```
    uint32_t prefix = uinteger.GetPrefix(numRequested);

    // The index into 'prefix' of the first bit of f, used for rounding.
    int32_t roundBitIndex = 32 - numRequested;
    uint32_t mask = (1 << roundBitIndex);

    uint32_t round;
    if (prefix & mask)
    {
        // The first bit of the remainder is 1.
        if (uinteger.GetNumBits() == numRequested)
        {
            // The first bit of the remainder is the lowest-order bit of
            // hat{u}.  Apply the ties-to-even rule.
            if (prefix & (mask << 1))
            {
                // The last bit of the trailing significand is odd, so round up.
                round = 1;
            }
            else
            {
                // The last bit of the trailing significand is even, so round down.
                round = 0;
            }
        }
        else
        {
            // The first bit of the remainder is not the lowest-order bit of hat{u}.
            // The remainder as a fraction is larger than 1/2, so round up.
            round = 1;
        }
    }
    else
    {
        // The first bit of the remainder is 0, so round down.
        round = 0;
    }

    // Get the unrounded trailing significand.
    uint32_t trailing = (prefix >> (roundBitIndex + 1));

    // Apply the rounding.
    trailing += round;
    return trailing;
}
```

The extraction of the prefix is shown in Listing 16.

Listing 16. Extraction of the requested bits from $\hat{u}$ and stored in a prefix.

```
uint32_t UInteger::GetPrefix(int numRequested)
{
    // The UInteger is an array u[imax] with u[0] odd and u[imax-1]
    // nonzero.  The number of bits starting with the leading 1-bit is
    // numBits.

    // Copy to 'prefix' the leading 32-bit block that is nonzero.
    int32_t bitIndex = numBits - 1;
    int32_t blockIndex = bitIndex / 32;
    uint32_t prefix = u[blockIndex];

    // Get the number of bits in the block starting with the leading 1-bit.
    int32_t firstBitIndex = bitIndex % 32;
```

```
    int32_t numBlockBits = firstBitIndex + 1;

    // Shift the leading 1-bit to index 31 of prefix.  We have consumed
    // numBlockBits, which might not be the entire budget.
    int32_t targetIndex = 31;
    prefix <<= (targetIndex - firstBitIndex);
    numRequested -= numBlockBits;

    if (numRequested > 0 && --blockIndex >= 0)
    {
        // More bits are available.  Copy and shift the entire 32-bit next
        // block and OR it into the 'prefix'.
        uint32_t nextBlock = bits[blockIndex];
        targetIndex -= numBlockBits;
        nextBlock <<= targetIndex - 31;  // Shift amount is positive.
        prefix |= nextBlock;
    }

    return prefix;
}
```

Let $x = 1.u * 2^p$ be a BSNumber in the interval $[2^{-126}, 2^{128})$, so $-126 \leq p < 128$. We need to generate a 23-bit trailing significand t so that $y = 1.t * 2^p$ is the closest floating-point number to x. The generation is similar to that for subnormals, but without having to work with the σ parameter. Pseudocode is shown in Listing 17.

Listing 17. Conversion of a BSNumber to a normal float.

```
BSNumber bsn;  // assume bsn is not zero
uint32_t s = (bsn.sign < 0 ? 1 : 0);
uint32_t e;  // biased exponent for float
uint32_t t;  // trailing significand for float
int32_t p = bsn.GetExponent();  // biasedExponent + numBits - 1
if (-126 <= p && p < 128)
{
    t = bsn.GetNormalTrailing();
    if (t & (1 << 24))
    {
        // The first 24 bits of hat{u} were all 1 and the remaining bits
        // led to rounding up, so round up to the closest normal that is
        // larger.
        ++e;
        t >>= 1;
    }
    // Eliminate the leading 1 (implied for IEEE normal floats)
    t &= ~(1 << 24);
}
// else: other cases discussed later

float result = CreateIEEEFloat(s, e, t);  // as uint32_t, (s << 31) | (e << 23) | t
```

The block of code that increments e and shifts-right t is not intuitive. To get to this block, we have $x = 1.\overline{1}^{23} f * 2^p$, where the first bit of f is 1. The float biased exponent is $e = p + 127$ and the t returned from GetNormalTrailing is 2^{24}. To round up, we add 1 to $1.\overline{2}^{23}$ to obtain $10.0 * 2^p = 1.0 * 2^{p+1}$. To obtain the

equivalent for the float result, the biased exponent must be incremented and the trailing significand must be zero. The shift-right produces $t = 2^{23}$, but regardless of rounding, the returned t had an explicit 1 stored at index 23 because we are converting to normal form. That bit is cleared (so in the case at hand we get $t = 0$).

The construction of the 24-bit significand is shown in Listing 18.

Listing 18. Construction of the significand for the normal.

```
uint32_t BSNumber::GetNormalTrailing()
{
    // Extract leading bits from hat{u}, shifted so first one is at index 31 of prefix.
    int32_t numRequested = 25;  // Get 1 extra bit for rounding.
    uint32_t prefix = uinteger.GetPrefix(numRequested);

    // The index into 'prefix' of the first bit of f, used for rounding.
    int32_t roundBitIndex = 32 - numRequested;
    uint32_t mask = (1 << roundBitIndex);

    uint32_t round;
    if (prefix & mask)
    {
        // The first bit of the remainder is 1.
        if (uinteger.GetNumBits() == numRequested)
        {
            // The first bit of the remainder is the lowest-order bit of
            // hat{u}.  Apply the ties-to-even rule.
            if (prefix & (mask << 1))
            {
                // The last bit of the trailing significand is odd, so round up.
                round = 1;
            }
            else
            {
                // The last bit of the trailing significand is even, so round down.
                round = 0;
            }
        }
        else
        {
            // The first bit of the remainder is not the lowest-order bit of hat{u}.
            // The remainder as a fraction is larger than 1/2, so round up.
            round = 1;
        }
    }
    else
    {
        // The first bit of the remainder is 0, so round down.
        round = 0;
    }

    // Get the unrounded trailing significand.
    uint32_t trailing = (prefix >> (roundBitIndex + 1));

    // Apply the rounding.
    trailing += round;
    return trailing;
}
```

The code has only minor differences compared to GetSubnormalTrailing. The actual code combines these and uses a template to handle both float and double. The consolidation then requires the prefix to be stored in a

64-bit unsigned integer because the worst case for double precision is a prefix storing 54 bits. The GetPrefix function also requires some modification to support double because you might have to extract the bits from 3 blocks of $\hat{u}$ rather than the 2 blocks required by float.

When x is in the interval $[2^{128}, \infty)$, we have overflow and the BSNumber is mapped to the floating-point infinity.

4 Binary Scientific Rationals

As stated previously, a binary scientific number has a finite number of bits in its representation. Addition, subtraction, and multiplication of two binary scientific numbers themselves have a finite number of bits. However, division of two binary scientific numbers might require an infinite number of bits. To support arbitrary precision arithmetic for algorithms that require divisions, we can construct ratios of numbers in the same way that the rational numbers are constructed from integers.

Specifically, let B be the set of binary scientific numbers. Let $x \in B$ and $y \in B$ where $y \neq 0$. Ratios x/y can be represented as 2-tuples

$$R = \{(x, y) : x \in B, y \in B, y \neq 0\} \tag{18}$$

As with rational numbers, multiple representations can occur. For example, 1/3 and 2/6 represent the same rational number. Also, 0/1 and 0/2 are both representations for zero. It is possible to reduce a ratio to a canonical form by cancelling factors common to both numbers and arranging for the denominators to be positive. For example, the ratio 2/6 can be reduced by cancelling the common factor 2 in the numerator and denominator. The ratio $2/(-6)$ is reduced to $-1/3$. The rational number zero has canonical form 0/1. We will refer to R as the *binary scientific rationals*.

4.1 Arithmetic Operations

Arithmetic operations for rational numbers are defined as follows.

$$\begin{array}{ccccll} \frac{x_0}{y_0} & + & \frac{x_1}{y_1} & = & \frac{x_0*y_1+x_1*y_0}{y_0*y_1}, & \text{addition} \\ \frac{x_0}{y_0} & - & \frac{x_1}{y_1} & = & \frac{x_0*y_1-x_1*y_0}{y_0*y_1}, & \text{subtraction} \\ \frac{x_0}{y_0} & * & \frac{x_1}{y_1} & = & \frac{x_0*x_1}{y_0*y_1}, & \text{multiplication} \\ \frac{x_0}{y_0} & / & \frac{x_1}{y_1} & = & \frac{x_0*y_1}{x_1*y_0}, & \text{division} \end{array} \tag{19}$$

Division has the additional constraint $x_1 \neq 0$. Equivalent arithmetic operations can be defined for the 2-tuples of R. Let $r_0 = (x_0, y_0)$ and $r_1 = (x_1, y_1)$; then

$$\begin{array}{ccccll} r_0 & + & r_1 & = & (x_0 * y_1 + x_1 * y_0, y_0 * y_1), & \text{addition} \\ r_0 & - & r_1 & = & (x_0 * y_1 - x_1 * y_0, y_0 * y_1), & \text{subtraction} \\ r_0 & * & r_1 & = & (x_0 * x_1, y_0 * y_1), & \text{multiplication} \\ r_0 & / & r_1 & = & (x_0 * y_1, x_1 * y_0), & \text{division} \end{array} \tag{20}$$

In all cases, the components of the 2-tuples are expressions involving addition, subtraction, and multiplication of binary scientific numbers.

4.2 Conversion of Floating-Point Numbers to Binary Scientific Rationals

Because a binary scientific rational is represented as a pair of binary scientific numbers, the floating-point number x is converted to a binary scientific number for the numerator and the floating-point number 1 is converted to a binary scientific number for the denominator.

If you want to represent a floating-point ratio x/y (with $y/\neq 0$) as a binary scientific rational, you may construct a binary scientific number n for the numerator x and a binary scientific number d for the denominator y. The pair (n, d) represents the ratio. In GTEngine, we use a canonical form for such ratios. If $n = 1.u * 2^p$ and $d = 1.v * 2^q$, we represent the ratio as $n' = 1.u * 2^{p-q}$ and $d' = 1.v$; therefore $d' \in [1, 2)$.

4.3 Conversion of Binary Scientific Rationals to Floating-Point Numbers

Let the binary scientific rational have numerator $n = 1.u * 2^p$ and denominator $d = 1.v * 2^q \neq 0$. The ratio is abstractly of the form

$$\frac{n}{d} = \begin{cases} \frac{1.u}{1.v} * 2^{p-q}, & 1.u \geq 1.v \\ \frac{2*(1.u)}{1.v} * 2^{p-q-1}, & 1.u < 1.v \end{cases} \tag{21}$$

The ratios on the right-hand side are rational numbers in the interval $[1, 2)$, so we may represent the right-hand side as $1.w * 2^r$ with the understanding that w is potentially an infinite sequence of bits.

To compute the exponent r for the result, we can use the comparison operator for binary scientific numbers. Compute $r = p - q$ and modify $n = 1.u$ and $d = 1.v$. If $n < d$, subtract 1 from r (it is now $p - q - 1$) and add 1 to the exponent for n (it is now $2 * (1.u)$ and $n \geq d$). At this time we have $n/d = 1.w$ and we need to compute enough bits of w to obtain the floating-point number closest to n/d using the mode of round-to-nearest-ties-to-even.

The conversion algorithm uses binary scientific rational arithmetic. We know that $n/d = 1.w_0 w_1 \ldots$. Subtracting 1 and multiplying by 2, we have $2(n - d)/d = w_0.w_1 \ldots$. The new numerator is $n_0 = 2(n - d)$, so $n_0/d = w_0.w_1 \ldots$. If $n_0 \geq d$, then $w_0 = 1$ and we can repeat the algorithm to move w_1 before the binary point. If $n_0 < d$, then $w_0 = 0$; the next numerator is simply $n_1 = 2n_0$ because no subtraction by 1 is necessary. The algorithm is repeated until we have 23 w_i bits for float or 52 w_i bits for double. As the bits are discovered, they are OR-ed into an unsigned integer to be used for the trailing significand for the floating-point number.

We need to examine the remaining bits of w to determine in which direction to round. At this time in the algorithm, $n/d = w_k.w_{k+1} \ldots \in [0, 2)$ and the trailing significand is the unsigned integer t (with either 23 or 52 bits). We must round the number $t.f$, where $f = 0.w_k w_{k+1} \ldots$. If $n' = n - d$, the classification of the fraction is $f < 1/2$ when $n' < 0$, $f = 1/2$ when $n' = 0$, and $f > 1/2$ when $n' > 0$. Rounding down uses t as is. Rounding up occurs when $n' > 0$ or when $n' = 0$ and the last 1-bit of the trailing significand is $t_0 = 1$.

We can take advantage of the conversions of binary scientific numbers to floating-point numbers by generating a binary scientific number as the approximation to the binary scientific rational n/d. Thus, we must shift-right t to obtain an odd integer and set the biased exponent accordingly.

Pseudocode is shown in Listing 19.

Listing 19. Conversion of a BSRational to a floating-point number.

```
// inputs are BSNumber n and d for BSRational n/d
// output is floating-point of the Real (float or double)
if (n.sign == 0)
{
    return (Real)0;
}

int32_t sign = n.sign * d.sign;
n.sign = 1;
d.sign = 1;
int32_t pmq = n.exponent - d.exponent;
n.biasedExponent = 1 - n.uinteger.numBits;
d.biasedExponent = 1 - d.uinteger.numBits;
if (n < d)
{
    ++n.biasedExponent;
    --pmq;
}

int precision = (24 for float, 52 for double);
int imax = precision - 1;
UIntType w = 0;  // UIntType is uint32_t for float, uint64_t for double
UIntType mask = (1 << imax);
for (int i = imax; i >= 0; --i, mask >>= 1)
{
    if (n < d)
    {
        n = 2 * n;
    }
    else
    {
        n = 2 * (n - d);
        w |= mask;
    }
}

n = n - d;
if (n.sign > 0 || (n.sign == 0 && (w & 1) == 1))
{
    // round up
    ++w;
}
// else round down (nothing to do)

if (w > 0)
{
    int32_t trailing = GetTrailingBit(w);
    w >>= trailing;  // w is now odd
    pmq += trailing;
    BSNumber result;
    result.uinteger = w;
    result.biasedExponent = pmq - imax;
    Real converted = (Real)result;
    if (sign < 0)
    {
        converted = -converted;
    }
    return converted;
}
else
{
    return (Real)0;
}
```

5 Implementation of Binary Scientific Numbers

The GTEngine class that encapsulates the details of binary scientific numbers is BSNumber. The public interface and data members is shown in Listing 20.

Listing 20. The class BSNumber that represents a binary scientific number.

```
template <typename UIntegerType>
class BSNumber
{
public:
    // Construction. The default constructor generates the zero BSNumber.
    BSNumber();
    BSNumber(BSNumber const& number);
    BSNumber(float number);
    BSNumber(double number);
    BSNumber(int32_t number);
    BSNumber(uint32_t number);
    BSNumber(int64_t number);
    BSNumber(uint64_t number);

    // Implicit conversions.
    operator float() const;
    operator double() const;

    // Assignment.
    BSNumber& operator=(BSNumber const& number);

    // Support for std::move.
    BSNumber(BSNumber&& number);
    BSNumber& operator=(BSNumber&& number);

    // Member access.
    int32_t GetSign() const;
    int32_t GetBiasedExponent() const;
    int32_t GetExponent() const;
    UIntegerType const& GetUInteger() const;

    // Comparisons.
    bool operator==(BSNumber const& number) const;
    bool operator!=(BSNumber const& number) const;
    bool operator< (BSNumber const& number) const;
    bool operator<=(BSNumber const& number) const;
    bool operator> (BSNumber const& number) const;
    bool operator>=(BSNumber const& number) const;

    // Unary operations.
    BSNumber operator+() const;
    BSNumber operator-() const;

    // Arithmetic.
    BSNumber operator+(BSNumber const& number) const;
    BSNumber operator-(BSNumber const& number) const;
    BSNumber operator*(BSNumber const& number) const;
    BSNumber& operator+=(BSNumber const& number);
    BSNumber& operator-=(BSNumber const& number);
    BSNumber& operator*=(BSNumber const& number);

private:
    // Helpers for comparisons.
    static bool EqualIgnoreSign(BSNumber const& n0, BSNumber const& n1);
    static bool LessThanIgnoreSign(BSNumber const& n0, BSNumber const& n1);

    // Helpers for arithmetic.
    static BSNumber AddIgnoreSign(BSNumber const& n0, BSNumber const& n1,
        int32_t resultSign);
```

```
    static BSNumber SubIgnoreSign(BSNumber const& n0, BSNumber const& n1,
        int32_t resultSign);

    // Helpers for conversions between BSNumber and float/double.
    template <typename IEEE>
    void ConvertFrom(typename IEEE::FloatType number);

    template <typename IEEE>
    typename IEEE::FloatType ConvertTo() const;

    template <typename IEEE>
    typename IEEE::UIntType GetTrailing(int32_t normal, int32_t sigma) const;

    // The number 0 is represented by: mSign = 0, mBiasedExponent = 0, and
    // mUInteger = 0.  For nonzero numbers, mSign != 0 and mUInteger > 0.
    int32_t mSign;
    int32_t mBiasedExponent;
    UIntegerType mUInteger;

    friend class BSRational<UIntegerType>;
};
```

The class is template based, allowing the user to select the underlying representation of unsigned integers. Effectively, the heart of the system is the manipulation of unsigned integers of arbitrary size, stored as an array. The template parameter UIntegerType must have the minimal interface shown in Listing 21.

Listing 21. The minimal interface required for UIntegerType.

```
class UIntegerType
{
public:
    // Construction.  The default constructor generates 0.
    UIntegerType();
    UIntegerType(UIntegerType const& number);
    UIntegerType(uint32_t number);
    UIntegerType(uint64_t number);
    UIntegerType(int numBits);

    // Assignment.
    UIntegerType& operator=(UIntegerType const& number);

    // Support for std::move.
    UIntegerType(UIntegerType&& number);
    UIntegerType& operator=(UIntegerType&& number);

    // Member access.
    int32_t GetNumBits() const;

    // Comparison.
    bool operator==(UIntegerType const& number) const;
    bool operator< (UIntegerType const& number) const;

    // Arithmetic.
    void Add(UIntegerType const& n0, UIntegerType const& n1);
    void Sub(UIntegerType const& n0, UIntegerType const& n1);
    void Mul(UIntegerType const& n0, UIntegerType const& n1);
    void ShiftLeft(UIntegerType const& number, int shift);
    int32_t ShiftRightToOdd(UIntegerType const& number);

    // Support for conversions to float/double.
    uint64_t GetPrefix(int numRequested) const;
};
```

The standard type used in GTEngine for arbitrary size unsigned integers is std::vector<uint32_t>.

The class functions are built on top of the framework discussed previously in this document. The comparisons are simple, as shown in Listing 22.

Listing 22. The low-level comparisons for BSNumber.

```
template <typename UIntegerType>
bool BSNumber<UIntegerType>::EqualIgnoreSign(BSNumber const& n0, BSNumber const& n1)
{
    return n0.mBiasedExponent == n1.mBiasedExponent
        && n0.mUInteger == n1.mUInteger;
}

template <typename UIntegerType>
bool BSNumber<UIntegerType>::LessThanIgnoreSign(BSNumber const& n0, BSNumber const& n1)
{
    int32_t e0 = n0.GetExponent(), e1 = n1.GetExponent();
    if (e0 < e1)
    {
        return true;
    }
    if (e0 > e1)
    {
        return false;
    }
    return n0.mUInteger < n1.mUInteger;
}

template <typename UIntegerType>
bool BSNumber<UIntegerType>::operator==(BSNumber const& number) const
{
    return (mSign == number.mSign ? EqualIgnoreSign(*this, number) : false);
}

template <typename UIntegerType>
bool BSNumber<UIntegerType>::operator<(BSNumber const& number) const
{
    if (mSign > 0)
    {
        if (number.mSign <= 0)
        {
            return false;
        }

        // Both numbers are positive.
        return LessThanIgnoreSign(*this, number);
    }
    else if (mSign < 0)
    {
        if (number.mSign >= 0)
        {
            return true;
        }

        // Both numbers are negative.
        return LessThanIgnoreSign(number, *this);
    }
    else
    {
        return number.mSign > 0;
    }
}
```

The low-level addition and subtraction are shown in Listing 23. These are direct implementations of the logic described in Section 3.2.

Listing 23. The low-level comparisons for BSNumber.

```
template <typename UIntegerType>
BSNumber<UIntegerType> BSNumber<UIntegerType>::AddIgnoreSign(
    BSNumber const& n0, BSNumber const& n1, int32_t resultSign)
{
    BSNumber result, temp;
    int32_t diff = n0.mBiasedExponent - n1.mBiasedExponent;
    if (diff > 0)
    {
        temp.mUInteger.ShiftLeft(n0.mUInteger, diff);
        result.mUInteger.Add(temp.mUInteger, n1.mUInteger);
        result.mBiasedExponent = n1.mBiasedExponent;
    }
    else if (diff < 0)
    {
        temp.mUInteger.ShiftLeft(n1.mUInteger, -diff);
        result.mUInteger.Add(n0.mUInteger, temp.mUInteger);
        result.mBiasedExponent = n0.mBiasedExponent;
    }
    else
    {
        temp.mUInteger.Add(n0.mUInteger, n1.mUInteger);
        int32_t shift = result.mUInteger.ShiftRightToOdd(temp.mUInteger);
        result.mBiasedExponent = n0.mBiasedExponent + shift;
    }
    result.mSign = resultSign;
    return result;
}

template <typename UIntegerType>
BSNumber<UIntegerType> BSNumber<UIntegerType>::SubIgnoreSign(
    BSNumber const& n0, BSNumber const& n1, int32_t resultSign)
{
    BSNumber result, temp;
    int32_t diff = n0.mBiasedExponent - n1.mBiasedExponent;
    if (diff > 0)
    {
        temp.mUInteger.ShiftLeft(n0.mUInteger, diff);
        result.mUInteger.Sub(temp.mUInteger, n1.mUInteger);
        result.mBiasedExponent = n1.mBiasedExponent;
    }
    else if (diff < 0)
    {
        temp.mUInteger.ShiftLeft(n1.mUInteger, -diff);
        result.mUInteger.Sub(n0.mUInteger, temp.mUInteger);
        result.mBiasedExponent = n0.mBiasedExponent;
    }
    else
    {
        temp.mUInteger.Sub(n0.mUInteger, n1.mUInteger);
        int32_t shift = result.mUInteger.ShiftRightToOdd(temp.mUInteger);
        result.mBiasedExponent = n0.mBiasedExponent + shift;
    }
    result.mSign = resultSign;
    return result;
}
```

The high-level arithmetic operations call the low-level ones, as shown in Listing 24.

Listing 24. The high-level arithmetic for BSNumber.

```
template <typename UIntegerType>
BSNumber<UIntegerType> BSNumber<UIntegerType>::operator+(BSNumber const& n1) const
{
    BSNumber const& n0 = *this;
    if (n0.mSign == 0)
    {
        return n1;
    }
    if (n1.mSign == 0)
    {
        return n0;
    }
    if (n0.mSign > 0)
    {
        if (n1.mSign > 0) // n0 + n1 = |n0| + |n1|
        {
            return AddIgnoreSign(n0, n1, +1);
        }
        else // n1.mSign < 0
        {
            if (!EqualIgnoreSign(n0, n1))
            {
                if (LessThanIgnoreSign(n1, n0)) // n0 + n1 = |n0| - |n1| > 0
                {
                    return SubIgnoreSign(n0, n1, +1);
                }
                else // n0 + n1 = -(|n1| - |n0|) < 0
                {
                    return SubIgnoreSign(n1, n0, -1);
                }
            }
            // else n0 + n1 = 0
        }
    }
    else // n0.mSign < 0
    {
        if (n1.mSign < 0) // n0 + n1 = -(|n0| + |n1|)
        {
            return AddIgnoreSign(n0, n1, -1);
        }
        else // n1.mSign > 0
        {
            if (!EqualIgnoreSign(n0, n1))
            {
                if (LessThanIgnoreSign(n1, n0)) // n0 + n1 = -(|n0| - |n1|) < 0
                {
                    return SubIgnoreSign(n0, n1, -1);
                }
                else // n0 + n1 = |n1| - |n0| > 0
                {
                    return SubIgnoreSign(n1, n0, +1);
                }
            }
            // else n0 + n1 = 0
        }
    }
    return BSNumber();  // = 0
}

template <typename UIntegerType>
BSNumber<UIntegerType> BSNumber<UIntegerType>::operator-(BSNumber const& n1) const
{
    BSNumber const& n0 = *this;
    if (n0.mSign == 0)
```

```
    {
        return -n1;
    }
    if (n1.mSign == 0)
    {
        return n0;
    }
    if (n0.mSign > 0)
    {
        if (n1.mSign < 0) // n0 - n1 = |n0| + |n1|
        {
            return AddIgnoreSign(n0, n1, +1);
        }
        else // n1.mSign > 0
        {
            if (!EqualIgnoreSign(n0, n1))
            {
                if (LessThanIgnoreSign(n1, n0)) // n0 - n1 = |n0| - |n1| > 0
                {
                    return SubIgnoreSign(n0, n1, +1);
                }
                else // n0 - n1 = -(|n1| - |n0|) < 0
                {
                    return SubIgnoreSign(n1, n0, -1);
                }
            }
            // else n0 - n1 = 0
        }
    }
    else // n0.mSign < 0
    {
        if (n1.mSign > 0) // n0 - n1 = -(|n0| + |n1|)
        {
            return AddIgnoreSign(n0, n1, -1);
        }
        else // n1.mSign < 0
        {
            if (!EqualIgnoreSign(n0, n1))
            {
                if (LessThanIgnoreSign(n1, n0)) // n0 - n1 = -(|n0| - |n1|) < 0
                {
                    return SubIgnoreSign(n0, n1, -1);
                }
                else // n0 - n1 = |n1| - |n0| > 0
                {
                    return SubIgnoreSign(n1, n0, +1);
                }
            }
            // else n0 - n1 = 0
        }
    }
    return BSNumber();  // = 0
}

template <typename UIntegerType>
BSNumber<UIntegerType> BSNumber<UIntegerType>::operator*(BSNumber const& number) const
{
    BSNumber result;  // = 0
    int sign = mSign * number.mSign;
    if (sign != 0)
    {
        result.mSign = sign;
        result.mBiasedExponent = mBiasedExponent + number.mBiasedExponent;
        result.mUInteger.Mul(mUInteger, number.mUInteger);
    }
    return result;
}
```

6 Implementation of Binary Scientific Rationals

The GTEngine class that encapsulates the details of binary scientific rationals is BSRational. The public interface and data members are shown in Listing 25.

Listing 25. The class BSRational that represents a binary scientific rational.

```
template <typename UIntegerType>
class BSRational
{
public:
    // Construction.  The default constructor generates the zero BSRational.
    // The constructors that take only numerators set the denominators to one.
    BSRational();
    BSRational(BSRational const& rational);
    BSRational(float numerator);
    BSRational(double numerator);
    BSRational(int32_t numerator);
    BSRational(uint32_t numerator);
    BSRational(int64_t numerator);
    BSRational(uint64_t numerator);
    BSRational(BSNumber<UIntegerType> const& numerator);
    BSRational(float numerator, float denominator);
    BSRational(double numerator, double denominator);
    BSRational(BSNumber<UIntegerType> const& numerator,
        BSNumber<UIntegerType> const& denominator);

    // Implicit conversions.
    operator float() const;
    operator double() const;

    // Assignment.
    BSRational& operator=(BSRational const& rational);

    // Support for std::move.
    BSRational(BSRational&& rational);
    BSRational& operator=(BSRational&& rational);

    // Member access.
    inline int GetSign() const;
    inline BSNumber<UIntegerType> const& GetNumerator() const;
    inline BSNumber<UIntegerType> const& GetDenomator() const;

    // Comparisons.
    bool operator==(BSRational const& rational) const;
    bool operator!=(BSRational const& rational) const;
    bool operator< (BSRational const& rational) const;
    bool operator<=(BSRational const& rational) const;
    bool operator> (BSRational const& rational) const;
    bool operator>=(BSRational const& rational) const;

    // Unary operations.
    BSRational operator+() const;
    BSRational operator-() const;

    // Arithmetic.
    BSRational operator+(BSRational const& rational) const;
    BSRational operator-(BSRational const& rational) const;
    BSRational operator*(BSRational const& rational) const;
    BSRational operator/(BSRational const& rational) const;
    BSRational& operator+=(BSRational const& rational);
    BSRational& operator-=(BSRational const& rational);
    BSRational& operator*=(BSRational const& rational);
    BSRational& operator/=(BSRational const& rational);

private:
```

```
    // Generic conversion code that converts to the correctly
    // rounded result using round-to-nearest-ties-to-even.
    template <typename UIntType, typename RealType>
    RealType Convert() const;

    BSNumber<UIntegerType> mNumerator, mDenominator;
};
```

The implementations of the member functions are straightforward according to the definitions of Section 4.

7 Exact Representation of Expressions Involving Square Roots

Typical geometric applications involving normalizing vectors for use as unit-length direction vectors. For example, consider a line represented by $\mathbf{X} = \mathbf{P} + t\mathbf{U}$, where $\mathbf{P}$ is a point on the line and $\mathbf{U}$ is a unit-length direction vector for the line. The scalar t is any real number. Given a point $\mathbf{X}$ on the line, the corresponding t-value is

$$t = (\mathbf{X} - \mathbf{P}) \cdot \mathbf{U} \tag{22}$$

In practice, the line direction might be generated from a vector $\mathbf{V}$ that is not necessarily unit length. The normalization is $\mathbf{U} = \mathbf{V}/|\mathbf{V}|$, where $|\mathbf{V}| = \sqrt{\mathbf{V} \cdot \mathbf{V}}$. Treating the inputs $\mathbf{X}$, $\mathbf{P}$ and $\mathbf{V}$ as exact rational tuples, equation (22) is an approximation to the true value of t because the normalization introduces rounding errors and it is not possible to represent irrational numbers exactly. For example, let $\mathbf{V} = (1, 2, 3)$. The normalized vector is $(1, 2, 3)/\sqrt{14}$, where $\sqrt{14}$ is irrational. The normalization code is shown in Listing 26.

Listing 26. Normalization of a vector that does not have exactly length 1 when considered as a tuple of rationals.

```
Vector3<double> V = { 1.0, 2.0, 3.0 };
double length = sqrt(V[0] * V[0] + V[1] * V[1] + V[2] * V[2]);  // estimate for sqrt(14.0)
V /= length;
// V = (0.267261241912424440, 0.534522483824848879, 0.801783725737273319)

typedef BSRational<UIntegerAP32> Rational;
Vector3<Rational> rV = { V[0], V[1], V[2] };
Rational rSqrLength = Dot(rV, rV);
// rSqrLength.biasedExponent = -105
// rSqrLength.bits = 0x00000200 0x00000000 0x000cc8b2 0xff10b80f
// Moving the binary point from the right-most bit 105 units to the left,
//   rSqrLength = 1.0^{53}110011001000101100101111111110001000010111000000001111
// where 0^{53} denotes the occurrence of 53 0-valued bits.  Therefore,
//   rSqrLength = 1.t where t > 0
// so V is not truly a unit-length vector.
```

If we introduce a symbolic term $\sqrt{d}$ where $d = 1/\mathbf{V} \cdot \mathbf{V}$ is rational, we may represent the t-value for the point $\mathbf{X}$ on the line by

$$t = (\mathbf{X} - \mathbf{P}) \cdot \mathbf{V}\sqrt{d} \tag{23}$$

Generally, numbers of the form $x + y\sqrt{d}$ for rational x, y and d can be represented exactly. This is the topic of *real quadratic fields*. In our current example, $t = t_0 + t_1\sqrt{d}$, where $t_0 = 0$ and $t_1 = (\mathbf{X} - \mathbf{P}) \cdot \mathbf{V}$ are both rational when $\mathbf{X}$, $\mathbf{P}$ and $\mathbf{V}$ are rational.

Having an exact computation as shown in equation (23), an application can choose to estimate $\sqrt{d}$ to a desired precision rather than converting d to a fixed precision floating-point number followed by a call to the math library function `sqrt`.

7.1 Real Quadratic Fields

It is possible to avoid the approximation of the length of a vector by using a concept from abstract algebra. A *square-free integer* d is an integer (not 0 or 1) whose prime factorization involves primes raised only to the first power. The prime numbers themselves are square free. The number 6 is square free because its factorization is $6 = 2^1 3^1$ which has only exponents 1 for the prime factors. The number 18 is not square free because its factorization is $18 = 2^1 3^2$ which has an exponent larger than 2 for a prime factor. If d is a square-free integer and $\mathbb{Q}$ is the field of rational numbers, a *real quadratic field* $\mathbb{Q}(\sqrt{d})$ is the set of numbers $x + y\sqrt{d}$, where $x, y \in \mathbb{Q}$.

The addition of two numbers is

$$(x_0 + y_0\sqrt{d}) + (x_1 + y_1\sqrt{d}) = (x_0 + x_1) + (y_0 + y_1)\sqrt{d} \tag{24}$$

and the subtraction of two numbers is

$$(x_0 + y_0\sqrt{d}) - (x_1 + y_1\sqrt{d}) = (x_0 - x_1) + (y_0 - y_1)\sqrt{d} \tag{25}$$

The additive identity is $0 + 0\sqrt{d}$. The multiplication of two numbers is

$$(x_0 + y_0\sqrt{d}) * (x_1 + y_1\sqrt{d}) = (x_0 x_1 + y_0 y_1 d) + (x_0 y_1 + x_1 y_0)\sqrt{d} \tag{26}$$

The multiplicative identity is $1 + 0\sqrt{d}$. The division of two numbers is the following, where the denominator is not zero,

$$(x_0 + y_0\sqrt{d})/(x_1 + y_1\sqrt{d}) = \frac{x_0 + y_0\sqrt{d}}{x_1 + y_1\sqrt{d}} * \frac{x_1 - y_1\sqrt{d}}{x_1 - y_1\sqrt{d}} = \frac{x_0 x_1 - y_0 y_1 d}{x_1^2 - y_1^2 d} + \frac{x_1 y_0 - x_0 y_1}{x_1^2 - y_1^2 d}\sqrt{d} \tag{27}$$

Although we have assumed $x_1 + y_1\sqrt{d} \neq 0 + 0\sqrt{d}$, it appears that we have a division by zero when $x_1^2 - y_1^2 d = 0$. In fact, this is not possible when x_1 and y_1 are rational numbers. The classical number-theoretic argument is proof by contradiction.

Let x and y be integers with greatest common divisor of 1 such that $x/y = x_1/y_1$. This is always possible. x_1 is a rational number n_x/d_x and y_1 is a rational number n_y/d_y, where n_x, d_x, n_y and d_y are integers. We have $x_1/y_1 = (n_x/d_x)/(n_y/d_y) = (n_x d_y)/(n_y d_x)$, where the numerator and denominator are integers. Now eliminate the common factors between numerator and denominator to obtain x/y.

We now have $dy^2 = x^2$. Let p be a prime factor of d, in which case $d = pu$ for some integer u and $puy^2 = x^2$. We chose d to be square free, so p is not a prime factor of u. The right-hand side x^2 is an integer that must have a prime factor p; in fact, this can arise only because x has a prime factor p, say $x = pv$. Therefore, $puy^2 = p^2 v^2$. Divide by p: $uy^2 = pv^2$. The right-hand side has a prime factor p, which means the left-hand side must have the factor p. We know that p is not a factor of u, so it must be that y has a factor of p.

But this means that p is a prime factor of x and of y, which is a contradiction to x and y having greatest common divisor of 1. The conclusion is that $x^2 - y^2 d \neq 0$ for any rational x and y.

In summary, $x + y\sqrt{d} = 0$ only when $x = 0$ and $y = 0$. If a real number is representable as an element $\mathbb{Q}(\sqrt{d})$, it has a unique representation. For if $r = x_0 + y_0\sqrt{d} = x_1 + y_1\sqrt{d}$, we have $(x_0 - x_1) + (y_0 - y_1)\sqrt{d} = 0 + 0\sqrt{d}$. The argument of the previous paragraph shows that $x_0 - x_1 = 0$ and $y_0 - y_1 = 0$, so the two representations must in fact be the same.

7.2 Allowing for Rational d

If d is not square free and not the square of an integer, the division operation for numbers of the form $x + y\sqrt{d}$ is still well defined for $(x, y) \neq (0, 0)$. The argument depends on proving that $x^2 - y^2 d \neq 0$ for any nonzero rational numbers x and y. As shown previously, the proof by contradiction is applicable. For example, suppose that $d = 12$. If there exist nonzero rational x and y for which $x^2 - y^2 12 = 0$, then x and $2y$ are nonzero rational numbers for which $x^2 - (2y)^2 3 = 0$, which we proved cannot happen for square-free integers such as 3.

The division is also well defined for $(x, y) \neq (0, 0)$ when we choose d to be rational. Firstly, choose $d = 1/q$ for prime integer q. We need to prove that $x^2 - y^2(1/q) \neq 0$ for any nonzero rational numbers x and y. The argument using proof by contradition applies yet again by considering divisor properties of $qx^2 = y^2$, Secondly, let $d = p/q > 0$ be a rational number where the numerator p and denominator q are prime integers. For the division to be well defined, we need to prove that $x^2 - y^2(p/q) \neq 0$ for nonzero rational x and y. The argument using proof by contradiction applied. Choose integers x and y whose greatest common divisor is 1 and for which $qx^2 = py^2$. It must be that x has a factor p and y has a factor q, say $x = pu$ and $y = qv$. Consequently, $p^2 qu^2 = pq^2 v^2$. Dividing by pq: $pu^2 = qv^2$. This in turn implies u has a factor q and v has a factor p, say, $u = qa$ and $v = pb$. Consequently, $x = pu = pqa$ and $y = qv = qpb$, which imply x and y both have the factor pq. This contradicts the assumption that the greatest common divisor of x and y is 1.

Generally the proof by contradiction methodology applied to rational d that is not the square of a rational number. The division for $x + y\sqrt{d}$ is well defined in the sense that $x^2 - y^2 d \neq 0$ for any nonzero x and y. In practice, this means we can implement the arithmetic operations of equations (24) through (27) for any rational number d that is not the square of a rational.

7.3 Detecting Whether the Length of a Vector is the Square of a Rational

The goal is to use real quadratic fields when $d = |\mathbf{V}|$ for a vector whose components are rational numbers. Let $\mathbf{V} = (a_0/b_0, \ldots, a_{n-1}/b_{n-1})$, where a_i and b_i are integers. The squared length is

$$d^2 = \sum_{i=0}^{n-1} \left(\frac{a_i}{b_i}\right)^2 = \frac{\sum_{i=0}^{n-1} \left(a_i^2 \prod_{j=0, j\neq i}^{n-1} b_j^2\right)}{\prod_{j=0}^{n-1} b_j^2} \tag{28}$$

The numerator and denominator are integers. The length is

$$d = \frac{\sqrt{\sum_{i=0}^{n-1} \left(a_i^2 \prod_{j=0, j\neq i}^{n-1} b_j^2\right)}}{\prod_{j=0}^{n-1} |b_j|} = \frac{\sqrt{A}}{B} \tag{29}$$

where the last equality defines the positive integers A and B.

The length d is a rational number when the integer A is the square of an integer.

Algorithms for estimating square roots of integers are found at [7]. An estimate for the square root of $A = a \times 2^{2n}$ with $1/2 \leq a < 2$ is $x_0 = 2^n$. The iteration scheme $x_{k+1} = (x_k + A/x_k)/2$ converges to $\sqrt{A}$ with quadratic convergence. $\lim_{k\to\infty} x_k = \bar{x}$, so $\bar{x} = (\bar{x} + A/\bar{x})/2$ which implies $\bar{x}^2 = A$. Iterate until x_k and A/x_k are in an interval $[j, j+2]$ with integer endpoints j and $j+2$. If $j^2 = A$ or $(j+1)^2 = A$ or $(j+2)^2 = A$, then d is the square of a rational number: $d = j/B$. Otherwise, $\sqrt{A}$ is irrational and d is not the square of a rational number.

When d is the square of a rational number, there is no need to switch the arithmetic to that of real quadratic fields.

7.4 Implementation of Arithmetic for Real Quadratic Fields

We can implement the arithmetic operations for a real quadratic field. The basic concepts are shown in Listing 27. The arithmetic operations are straightforward implementations of equations (24) through (27). The only technical issue is declaring the static member for d^2. The use of an anonymous namespace is to make QFElement private to the source file in which the code is included. This is necessary if you want to have multiple queries using the same class yet have different values assigned to the static member dSqr.

The less-than comparison operator has logic that is not trivial because of the need to square terms to eliminate $\sqrt{d}$. The direction of the inequality after squaring depends on various signed terms. To compare $x_0 + y_0\sqrt{d} < x_1 + y_1\sqrt{d}$, consider instead $(x_0 - x_1) < (y_1 - y_0)\sqrt{d}$. If $x_0 = x_1$, the comparison is true when $y_1 - y_0 > 0$. If $y_0 = y_1$, the comparison is true when $x_0 - x_1 < 0$. Otherwise, $x_0 \neq x_1$ and $y_0 \neq y_1$. When squaring the expression, the direction of the inequality depends on the signs of $a_0 = x_0 - x_1$ and $b_0 y_0 - y_1$. If $a_0 > 0$ and $b_0 < 0$, the comparison is false. If $a_0 < 0$ and $b_0 > 0$, the comparison is true. If $a_0 > 0$ and $b_0 > 0$, the comparison is true when $a_0^2 < b_0^2 d$. If $a_0 < 0$ and $b_0 < 0$, the comparison is true when $a_0^2 > b_0^2 d$; in this case, note that the inequality has changed direction.

Listing 27. The implementation of the arithmetic operations for a real quadratic field.

```
#include <array>

// The class QFElement represents an element of a real quadratic field
// Q(sqrt(d)), where QFElement<Rational, ID>::DSqr stores d*d.  The template
// parameter ID is used to allow multiple QFElement classes, each sharing a
// DSqr value.  A typical preamble to a source file is
//
// #include <GTEngine.h>
// #include "GteQFElement.h"
// using namespace gte;
//
// typedef BSRational<UIntegerAP32> Rational;
//
// // The source file has a need for two different real quadratic fields, each
// // having its own d value.
// Rational QFElement<Rational, 0>::DSqr;
// Rational QFElement<Rational, 1>::DSqr;
//
// The application must set QFElement<Rational, ID>::DSqr before executing any
// of the QFElement operations that use it.
```

```
namespace gte
{
    template <typename Rational, size_t ID>
    class QFElement
    {
    public:
        // The identifier allows for multiple QFElement classes used by
        // a single algorithm, each class having its own shared DSqr.
        // Set the DSqr value before using arithmetic operations.
        enum { IDENTIFIER = ID };
        static Rational DSqr;

        // The default-constructed element is uninitialized.
        QFElement() {}

        // Create an element of the form x0 + 0 * sqrt(d).
        QFElement(Rational const& x0)
        {
            mTuple[0] = x0;
            mTuple[1] = (Rational)0;
        }

        // Create an element of the form x0 + x1 * sqrt(d).
        QFElement(Rational const& x0, Rational const& x1)
        {
            mTuple[0] = x0;
            mTuple[1] = x1;
        }

        // Access the components of an element e = e[0] + e[1] * sqrt(d).
        Rational& operator[] (size_t i)
        {
            return mTuple[i];
        }

        Rational const& operator[] (size_t i) const
        {
            return mTuple[i];
        }

    private:
        std::array<Rational, 2> mTuple;
    };


    // Unary operations.
    template <typename Rational, size_t ID>
    QFElement<Rational, ID> operator+(
        QFElement<Rational, ID> const& q)
    {
        return q;
    }

    template <typename Rational, size_t ID>
    QFElement<Rational, ID> operator-(
        QFElement<Rational, ID> const& q)
    {
        return QFElement<Rational, ID>(-q[0], -q[1]);
    }

    // Arithmetic operations.
    template <typename Rational, size_t ID>
    QFElement<Rational, ID> operator+(
        QFElement<Rational, ID> const& q0,
        QFElement<Rational, ID> const& q1)
    {
        return QFElement<Rational, ID>(q0[0] + q1[0], q0[1] + q1[1]);
    }

    template <typename Rational, size_t ID>
    QFElement<Rational, ID> operator-(
        QFElement<Rational, ID> const& q0,
```

```
    QFElement<Rational, ID> const& q1)
{
    return QFElement<Rational, ID>(q0[0] - q1[0], q0[1] - q1[1]);
}

template <typename Rational, size_t ID>
QFElement<Rational, ID> operator*(
    QFElement<Rational, ID> const& q0,
    QFElement<Rational, ID> const& q1)
{
    QFElement<Rational, ID> result;
    result[0] = q0[0] * q1[0] + q0[1] * q1[1] * QFElement<Rational, ID>::DSqr;
    result[1] = q0[0] * q1[1] + q0[1] * q1[0];
    return result;
}

template <typename Rational, size_t ID>
QFElement<Rational, ID> operator/(
    QFElement<Rational, ID> const& q0,
    QFElement<Rational, ID> const& q1)
{
    QFElement<Rational, ID> result;
    Rational denom = q1[0] * q1[0] - q1[1] * q1[1] * QFElement<Rational, ID>::DSqr;
    Rational invDenom = (Rational)1 / denom;
    result[0] = (q0[0] * q1[0] - q0[1] * q1[1] * QFElement<Rational, ID>::DSqr) * invDenom;
    result[1] = (q1[0] * q0[1] - q0[0] * q1[1]) * invDenom;
    return result;
}

// Arithmetic updates.
template <typename Rational, size_t ID>
QFElement<Rational, ID>& operator+=(
    QFElement<Rational, ID>& q0,
    QFElement<Rational, ID> const& q1)
{
    q0 = q0 + q1;
    return q0;
}

template <typename Rational, size_t ID>
QFElement<Rational, ID>& operator-=(
    QFElement<Rational, ID>& q0,
    QFElement<Rational, ID> const& q1)
{
    q0 = q0 - q1;
    return q0;
}

template <typename Rational, size_t ID>
QFElement<Rational, ID>& operator*=(
    QFElement<Rational, ID>& q0,
    QFElement<Rational, ID> const& q1)
{
    q0 = q0 * q1;
    return q0;
}

template <typename Rational, size_t ID>
QFElement<Rational, ID>& operator/=(
    QFElement<Rational, ID>& q0,
    QFElement<Rational, ID> const& q1)
{
    q0 = q0 / q1;
    return q0;
}

// Comparisons for sorting.
template <typename Rational, size_t ID>
bool operator==(
    QFElement<Rational, ID> const& q0,
    QFElement<Rational, ID> const& q1)
{
```

```
    return q0[0] == q1[0] && q0[1] == q1[1];
}

template <typename Rational, size_t ID>
bool operator!=(
    QFElement<Rational, ID> const& q0,
    QFElement<Rational, ID> const& q1)
{
    return !operator==(q0, q1);
}

template <typename Rational, size_t ID>
bool operator< (
    QFElement<Rational, ID> const& q0,
    QFElement<Rational, ID> const& q1)
{
    // Analyze x0+y0*sqrt(d) < x1+y1*sqrt(d) using the expression
    // (x0-x1) < (y1-y0)*sqrt(d).
    Rational const zero(0);
    Rational xdiff = q0[0] - q1[0];  // x0 - x1
    Rational ydiff = q1[1] - q0[1];  // y1 - y0
    if (ydiff > zero)
    {
        if (xdiff <= zero)
        {
            return true;
        }
        else  // x0 > x1
        {
            return xdiff * xdiff < ydiff * ydiff * QFElement<Rational, ID>::DSqr;
        }
    }
    else if (ydiff < zero)  // y0 > y1
    {
        if (xdiff >= zero)
        {
            return false;
        }
        else
        {
            return xdiff * xdiff > ydiff * ydiff * QFElement<Rational, ID>::DSqr;
        }
    }
    else  // y0 == y1
    {
        return xdiff < zero;
    }
}

template <typename Rational, size_t ID>
bool operator> (
    QFElement<Rational, ID> const& q0,
    QFElement<Rational, ID> const& q1)
{
    return operator<(q1, q0);
}

template <typename Rational, size_t ID>
bool operator<= (
    QFElement<Rational, ID> const& q0,
    QFElement<Rational, ID> const& q1)
{
    return !operator>(q0, q1);
}

template <typename Rational, size_t ID>
bool operator>= (
    QFElement<Rational, ID> const& q0,
    QFElement<Rational, ID> const& q1)
{
    return !operator<(q0, q1);
}
```

}

7.5 Expressions with Multiple Square Roots

Geometric algorithms in 3 dimensions typically involve a coordinate system with orthonormal basis of vectors. For example, a point in 3D can be written as

$$\mathbf{P} = \mathbf{C} + \sum_{i=1}^{3} x_i \mathbf{U}_i \tag{30}$$

where $\mathbf{C}$ is the origin of the coordinate system and $\{\mathbf{U}_1, \mathbf{U}_2, \mathbf{U}_3\}$ is a right-handed orthonormal basis; that is, the vectors are unit length, mutually perpendicular and $\mathbf{U}_3 = \mathbf{U}_1 \times \mathbf{U}_2$. The extents e_i are the half-lengths of the edges of the box.

Normalization of vectors $\mathbf{V}_i$ that are not generally unit length can lead to non-unit-length vectors when computing with floating-point arithmetic. The concept was discussed previously in this document for a single normalized vector and extends to multiple vectors.

Specifically, let $\mathbf{V}_1 = (v_{11}, v_{12}, v_{13})$ be a nonzero vector that is not necessarily unit length. For the sake of argument, let $|v_{13}|$ be the maximum absolute value of the components of $\mathbf{V}_1$. A vector perpendicular to $\mathbf{V}_1$ is $\mathbf{V}_2 = (v_{21}, v_{22}, v_{23}) = (v_{13}, 0, -v_{11})$. A vector perpendicular to $\mathbf{V}_1$ and $\mathbf{V}_2$ is $\mathbf{V}_3 = \mathbf{V}_1 \times \mathbf{V}_2 = (v_{31}, v_{32}, v_{33}) = (-v_{11}v_{12}, v_{11}^2 + v_{13}^2, -v_{12}v_{13})$. Let $\mathbf{U}_i = \mathbf{V}_i / |\mathbf{V}_i|$. Define $d_j = 1/\mathbf{V}_j \cdot \mathbf{V}_j$ for $j = 1, 2$ and note that $d_1 d_2 = 1/\mathbf{V}_3 \cdot \mathbf{V}_3$. The orthonormal basis vectors are $\mathbf{U}_1 = \mathbf{V}_1\sqrt{d_1}$, $\mathbf{U}_2 = \mathbf{V}_2\sqrt{d_2}$ and $\mathbf{U}_3 = \mathbf{V}_1 \times \mathbf{V}_2\sqrt{d_1 d_2}$.

Referring back to equation (30),

$$\mathbf{P} = \mathbf{C} + x_1 \mathbf{V}_1 \sqrt{d_1} + x_1 \mathbf{V}_2 \sqrt{d_2} + x_3 \mathbf{V}_3 \sqrt{d_1 d_2} \tag{31}$$

where each component equation involves a linear combination of $\sqrt{d_1}$, $\sqrt{d_2}$ and $\sqrt{d_1 d_2}$.

7.5.1 Arithmetic Operations

Generally, we can implement an arithmetic system where the elements are of the form

$$x_0 + x_1\sqrt{d_1} + x_2\sqrt{d_2} + x_3\sqrt{d_1 d_2} \tag{32}$$

where all three square roots are irrational numbers and the coefficients x_i are rational numbers. Addition and subtraction are defined by

$$\begin{aligned} &(x_0 + x_1\sqrt{d_1} + x_2\sqrt{d_2} + x_3\sqrt{d_1 d_2}) \pm (y_0 + y_1\sqrt{d_1} + y_2\sqrt{d_2} + y_3\sqrt{d_1 d_2}) \\ &= (x_0 \pm y_0) + (x_1 \pm y_1)\sqrt{d_1} + (x_2 \pm y_2)\sqrt{d_2} + (x_3 \pm y_3)\sqrt{d_1 d_2} \end{aligned} \tag{33}$$

Multiplication is defined by

$$(x_0 + x_1\sqrt{d_1} + x_2\sqrt{d_2} + x_3\sqrt{d_1 d_2}) * (y_0 + y_1\sqrt{d_1} + y_2\sqrt{d_2} + y_3\sqrt{d_1 d_2})$$
$$= \begin{array}{l} (x_0y_0 + x_1y_1d_1 + x_2y_2d_2 + x_3y_3d_1d_2)+ \\ (x_0y_1 + x_1y_0 + (x_2y_3 + x_3y_2)d_2)\sqrt{d_1}+ \\ (x_0y_2 + x_2y_0 + (x_1y_3 + x_3y_1)d_1)\sqrt{d_2}+ \\ (x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0)\sqrt{d_1d_2} \end{array} \tag{34}$$

Division is defined by solving $x * y = 1$ for the y_i-components to obtain $y = 1/x$. To show some algebraic patterns in the components, define $d_3 = d_1 d_2$,

$$\begin{array}{lcl} y_0 & = & (+x_0(x_0^2 - d_1x_1^2 - d_2x_2^2 - d_3x_3^2) + 2d_3x_1x_2x_3)/\Delta \\ y_1 & = & (-x_1(x_0^2 - d_1x_1^2 + d_2x_2^2 + d_3x_3^2) + 2d_2x_0x_2x_3)/\Delta \\ y_2 & = & (-x_2(x_0^2 + d_1x_1^2 - d_2x_2^2 + d_3x_3^2) + 2d_1x_0x_1x_3)/\Delta \\ y_3 & = & (-x_3(x_0^2 + d_1x_1^2 + d_2x_2^2 - d_3x_3^2) + 2x_0x_1x_2)/\Delta \end{array} \tag{35}$$

where

$$\Delta = (x_0^2 - (d_1x_1^2 + d_2x_2^2 + d_3x_3^2))^2 - 4(d_1x_1^2d_3x_3^2 + d_1x_1^2d_2x_2^2 + d_2x_2^2d_3x_3^2) + 8d_3x_0x_1x_2x_3 \tag{36}$$

7.5.2 Uniqueness of the Representation of Zero

For a real quadratic field $\mathbb{Q}(\sqrt{d})$ where d is square free, the only way $x_0 + x_1\sqrt{d}$ can be zero is when $x_0 = 0$ and $x_1 = 0$. The same is true for elements of the form of equation (32); the number zero is uniquely represented by $x_0 = x_1 = x_2 = x_3 = 0$. Observe that $x = (x_0 + x_1\sqrt{d_1}) + (x_2 + x_3\sqrt{d_1})\sqrt{d_2}$.

If $x_0 = 0$ and $x_1 = 0$, then it is necessary that $x_2 + x_3\sqrt{d_1} = 0$. But then $x_2 = 0$ and $x_3 = 0$ because of the uniqueness of the representation of zero in $\mathbb{Q}(\sqrt{d_1})$. Similarly, if $x_2 = 0$ and $x_3 = 0$, then $x_0 + x_1\sqrt{d_1} = 0$ which in turn implies $x_0 = 0$ and $x_1 = 0$.

The remaining cases are defined by $x_0^2 + x_1^2 \neq 0$ and $x_2^2 + x_3^2 \neq 0$. Solve $x = 0$ for

$$-\sqrt{d_2} = \frac{x_0 + x_1\sqrt{d_1}}{x_2 + x_3\sqrt{d_1}} = \frac{(x_0x_2 - x_1x_3d_1) + (x_1x_2 - x_0x_3)\sqrt{d_1}}{x_2^2 - x_3^2d_1} = y_0 + y_1\sqrt{d_1} \tag{37}$$

where the last equality defines the rational numbers y_0 and y_1. Squaring the equation leads to

$$d_2 = (y_0 + y_1\sqrt{d_1})^2 = (y_0^2 + y_1^2d_1) + (2y_0y_1)\sqrt{d_1} \tag{38}$$

which in turn implies

$$\sqrt{d_1} = \frac{d_2 - (y_0^2 + y_1^2d_1)}{2y_0y_1} \tag{39}$$

This last equation is not possible, because d_1 is square free which makes $\sqrt{d_1}$ irrational. The right-hand side of the equation is a rational number. Therefore, none of these cases can happen and we know that zero has a unique representation.

7.5.3 Comparisons of Numbers with Multiple Square Roots

The comparison $x = x_0 + x_1\sqrt{d_1} + x_2\sqrt{d_2} + x_3\sqrt{d_1 d_2} > 0$ must be computed using a mixture of rational arithmetic and symbolic manipulation; that is, we want not to estimate the square roots. As before, write $x = (x_0 + x_1\sqrt{d_1}) + (x_2 + x_3\sqrt{d_1})\sqrt{d_2}$.

The comparison operators for $\mathbb{Q}(\sqrt{d_1})$ and $\mathbb{Q}(\sqrt{d_2})$ may be jointly used to create comparison operators for $\mathbb{Q}(\sqrt{d_1}, \sqrt{d_2}, \sqrt{d_1 d_2})$. Let $u = x_0 + x_1\sqrt{d_1}$ and $v = x_2 + x_3\sqrt{d_1}$ so that $x = u + v\sqrt{d_2}$. The implementation of the comparison operators for $\mathbb{Q}(\sqrt{d_2})$ are provided in Listing 27. In the less-than operator, the arithmetic uses type Rational. For the multiple square root case, that type becomes QFElement<Rational> with QFElement<Rational>::DSqr set to d_2. Thus, the differences of u and v live in $\mathbb{Q}(\sqrt{d_2})$ and are compared within $\mathbb{Q}(\sqrt{d_2})$. The comparisons

8 Performance Considerations

GTEngine provides a class UIntegerAP32 that supports the unsigned integer storage and logic for arbitrary precision arithmetic (AP stands for Arbitrary Precision). The storage is of type std::vector<uint32_t>. The arithmetic logic unit (ALU) is implemented in a template base class UIntegerALU<UInteger>. We use the Curiously Recurring Template Paradigm to allow other classes to share the ALU.

Simple examples to use the arbitrary precision is shown in the next listing.

```
float x = 1.2345f, y = 6.789f;
BSNumber<UIntegerAPI32> nx(x), ny(y);
BSNumber<UIntegerAPI32> nsum = nx + ny;
float sum = (float)nsum;

BSRational<UIntegerAPI32> rx(x), ry(y);
BSRational rdiv = rx / ry;
float div = (float)rdiv;
```

Many of the template classes in GTEngine that support float or double through a template parameter Real will work when Real is replaced with BSNumber<UIntegerAP32> (code has no divisions) or BSRational<UIntegerAP32> (code has divisions).

For a large number of computations and for a large required number of bits of precision to produce an exact result, a profiler will show that the main bottleneck is allocation and deallocation of the std::vector arrays together with the copies of data. To remedy this, we provide support for an unsigned integer type that allows you to specify the maximum number of bits of precision. You specify the maximum number of 32-bit words, say, N, to store the bits. The maximum number of bits of precision is $4N$. The class that implements this is UIntegerFP32<int N> (FP stands for Fixed Precision) and the storage is std::array<uint32_t,N>. We have taken care to optimize the code to compute quantities in-place, avoiding as many copies of arrays as possible. Without this, the cost of copying can be nearly as expensive as std::vector allocations and deallocations. The fixed-precision class shares the ALU of the arbitrary precision class.

8.1 Static Computation of the Maximum Bits of Precision

The technical challenge for using UIntegerFP32<int N> is to determine how large N must be for your sequence of computations. The analysis for computing N can be tedious. The summary of bit counting in Section 3.4

is listed next. Let $\text{bits}(x)$ denote the number of bits of precision for x; this is 24 for float and 53 for double. Let $p_{\max}(x)$ denote the maximum exponent for x; this is 127 for float and 1023 for double. Let $\beta_{\min}(x)$ denote the minimum biased exponent for x; this is -149 for float and -1074 for double. For multiplication,

$$\begin{aligned} \text{bits}(x*y) &= \text{bits}(x)+\text{bits}(y) \\ p_{\max}(x*y) &= p_{\max}(x)+p_{\max}(y)+1 \\ \beta_{\min}(x*y) &= \beta_{\min}(x)+\beta_{\min}(y) \end{aligned} \tag{40}$$

The exponent for a product is either the sum of exponents of the inputs or one more than the sum of exponents. The formula for $p_{\max}$ uses the worst-case behavior. For addition,

$$\begin{aligned} \beta_{\min}(x+y) &= \min\{\beta_{\min}(x),\beta_{\min}(y)\} \\ p_{\max}(x+y) &= \max\{p_{\max}(x),p_{\max}(y)\}+1 \\ \text{bits}(x+y) &= p_{\max}(x+y)-\beta_{\min}(x+y) \end{aligned} \tag{41}$$

We have provided a class, BSPrecision, that allows you to specify a sequence of expressions and compute N. The class interface is shown in Listing 28.

Listing 28. The class interface for BSPrecision.

```
class BSPrecision
{
public:
    // This constructor is used for 'float' or 'double'.  The floating-point
    // inputs for the expressions have no restrictions; that is, the inputs
    // can be any finite floating-point numbers (normal or subnormal).
    BSPrecision(bool isFloat);

    // If you know that your inputs are limited in magnitude, use this
    // constructor.  For example, if you know that your inputs x satisfy
    // |x| <= 8, you can specify maxExponent of 3.  The minimum power will
    // still be that for the smallest positive subnormal.
    BSPrecision(bool isFloat, int32_t maxExponent);

    // You must use this constructor carefully based on knowledge of your
    // expressions.  For example, if you know that your inputs are 'float'
    // and in the interval [1,2), you would choose 24 for the number of bits
    // of precision, a minimum biased exponent of -23 because the largest
    // 'float' smaller than 2 is 1.1^{23}*2^0 = 1^{24}*2^{-23}, and a maximum
    // exponent of 0.  These numbers work to determine bits of precision to
    // compute x*y+z*w.  However, if you then compute an expression such as
    // x-y for x and y in [1,2) and multiply by powers of 1/2, the bit
    // counting will not be correct because the results can be subnormals
    // where the minimum biased exponent is -149, not -23.
    BSPrecision(int32_t numBits, int32_t minBiasedExponent, int32_t maxExponent);

    // Member access.
    int32_t GetNumWords() const;
    int32_t GetNumBits() const;
    int32_t GetMinBiasedExponent() const;
    int32_t GetMinExponent() const;
    int32_t GetMaxExponent() const;

    // Support for determining the number of bits of precision required to
    // compute an expression.
    BSPrecision operator*(BSPrecision const& precision);
```

```
    BSPrecision operator+(BSPrecision const& precision);
    BSPrecision operator-(BSPrecision const& precision);

private:
    int32_t mNumBits, mMinBiasedExponent, mMaxExponent;
};
```

The computed values are conservative, assuming worst-case inputs. The first two constructors have a bool input that let's you specify the expressions involve float (input is true) or double (input is false).

Suppose you want to compute exactly the sign of the expression $d = x * y - z * w$.

```
// Determine N for 'float'.
BSPrecision px(true); // numbits = 24, minbiasexp = -149, maxexp = 127
BSPrecision py = px, pz = px, pw = px;
BSPrecision pxy = px * py, pzw = pxy; // numbits = 48, minpow = -298, maxexp = 255
BSPrecision pd = pxy - pzw; // or pd = px * py - pz * pw;
int numWords, numBits;
numBits = pd.GetNumBits(); // 554, minbiasexp = -298, maxexp = 256
numWords = pd.GetNumWords(); // 18

// Determine N for 'double'.
px = BSPrecision(false); // numbits = 53, minbiasexp = -1074, maxexp = 1023
py = px; pz = px; pw = px;
pxy = px * py; pzw = pxy; // numbits = 106, minbiasexp = -2148, maxexp = 2047
pd = pxy - pzw;
numBits = pd.GetNumBits(); // 4196, minbiasexp = -2148, maxexp = 2048
numWords = pd.GetNumWords(); // 132
```

The computed numWords allow you to have any finite floating-point input.

Suppose that you know x, y, z, and w are in the interval $[-1, 1]$. The bit counting is then

```
// Compute N for an exact 'float' computation when you know that your inputs
// are in the interval [-1,1] and the maximum power is 0.
px = BSPrecision(true, 0); // numbits = 24, minbiasexp = -149, maxexp = 0
py = px; pz = px; pw = px;
pxy = px * py; pzw = pxy; // numbits = 48, minbiasexp = -298, maxexp = 1
pd = pxy - pzw;
numBits = pd.GetNumBits(); // 300, minbiasexp = -298, maxexp = 2
numWords = pd.GetNumWords(); // 10

// Compute N for an exact 'double' computation when you know that your inputs
// are in the interval [-1,1] and the maximum power is 0.
px = BSPrecision(false, 0); // numbits = 53, minbiasexp = -1074, maxexp = 0
py = px; pz = px; pw = px;
pxy = px * py; pzw = pxy; // numbits = 106, minbiasexp = -2148, maxexp = 1
pd = pxy - pzw;
numBits = pd.GetNumBits(); // 2150, minbiasexp = -2148, maxexp = 2
numWords = pd.GetNumWords(); // 68
```

The maximum number of bits and words are smaller for the restricted domain.

Finally, suppose that you want to compute $x * y + z * w$ where the inputs are in the interval $[1, 2)$. It is safe to use the third constructor because the expression does not have subtraction, in which case the choice of minimum biased exponent is valid.

```
// Compute N for an exact 'float' computation of d = x*y + z*w, where
// x, y, z, and w are in [1,2). The minimum and maximum powers are 0.
// The result d is in [1,8), so there is no chance of subtractions
// causing negative powers to occur. The number 1.1^{23} *2^0 is the
// largest 'float' smaller than, and 1.1^{23} * 2^0 = 1^{24} * 2^{-23},
```

```
// so the minimum biased exponent is -23.
px = BSPrecision(24, -23, 0);  // numbits = 24, minbiasexp = -23, maxexp = 0
py = px;  pz = px;  pw = px;
pxy = px * py; pzw = pxy;  // numbits = 48, minbiasexp = -46, maxexp = 1
pd = pxy + pzw;
numBits = pd.GetNumBits();  // 48, minbiasexp = -46, maxexp = 2
numWords = pd.GetNumWords();  // 2
IEEEBinary32 tmp;
tmp.number = 2.0f;
tmp.encoding = tmp.GetNextDown();  // tmp.number = 1.7FFFFF * 2^0
BSNumber<UIntegerAP32> bx(tmp.number);
BSNumber<UIntegerAP32> bd = bx*bx + bx*bx;  // numbits = 48, pow = 2
IEEEBinary64 dexact;
dexact.number = (double)bd;  // 7.9999990463257120 = 1.FFFFFC000002 * 2^2
double dinput = tmp.number;
double exact = dinput * dinput + dinput * dinput;  // = dexact.number
```

In fact, this example shows that for float inputs to $x * y + z * w$ in the interval $[1, 2)$, if you convert them to double and compute the expression, you get an exact result; no floating-point rounding occurs.

8.2 Dynamic Computation of the Maximum Bits of Precision

A conditional define is disabled in the UIntegerAP32 header file, GTE_COLLECT_UINTEGERAP32_STATISTICS. You can enable the flag to expose a static class member in UIntegerAP32 that is used to count the maximum number of bits used in a sequence of computations. You can use this as a rough estimate for N given your input data. You are responsible for deciding whether this N is large enough no matter what your input data—the value of N measured this way is specific to the input data set. The following code shows how to measure N assuming you have compiled the engine with the flag enabled.

```
float x, y, z;  // all initialized to finite floating-point numbers
UIntegerAP32::SetMaxSizeToZero();
BSNumber<UIntegerAP32> nx(x), ny(y), nz(z);
BSNumber<UIntegerAP32> temp0 = nx + ny * nz;
BSNumber<UIntegreAP32> temp1 = nx * ny + nz;
BSNumber<UIntegerAP32> result = temp0 * temp1;
size_t maxWords = UIntegerAP32::GetMaxSize();
```

A similar system exists for UIntegerFP32, even though the storage is of fixed size (template parameter N). The system tracks mSize, the number of array elements used, to determine whether you might be able to use a smaller N.

8.3 Memory Management

The class UIntegerAP32 uses std::vector for dynamically resizable storage. Once these objects become sufficiently large, memory allocation, memory deallocation, and memory copies become a noticeable bottleneck. UIntegerAP32 is designed to compute arithmetic operations in-place to minimize creation of new objects and copies. However, they are not entirely unavoidable in application code. In order to avoid copies, the classes UIntegerAP32, BSNumber, and BSRational have support for C++ 11 move semantics (move construction, move operator). When creating temporary BSNumber or BSRational objects that are assigned to other objects, consider whether you can replace an assignment x = y by x = std::move(y).

Although std::move may be passed a std::array, there are no pointers to steal, so the operation becomes a copy. The class UIntegerFP32 uses std::array for fixed-size storage, but its move operator does not apply a move to the array. The reason is that the array has the maximum storage required for the computations, but member

mSize keeps track of the number of array elements that are used. Typically, this is smaller than the template parameter N, so both the assignment and move operators for UIntegerFP32 copy only mSize elements, which minimizes copy time.

If you choose a large N for UIntegerFP32, you must deal with the problem of call stack size. The default call stack for MSVS 2013 C++ 11 is 1 MB. For large N and even a moderate number of UIntegerFP32 objects, you can exceed the stack size. You must set the call stack size to a larger number. In the project property pages, navigate in the left pane to

```
Configuration Properties | Linker | System
```

In the right pane you will see the option `Stack Reserve Size`. Edit this and provide a sufficiently large number. For example, the `DistanceSegments3` sample application uses the template class UIntegerFP32<128>, and the test code involves quite a few of these objects. We set the stack reserve size to 16 MB by entering in the right pane the number 16777216.

Assuming N is chosen large enough to allow any finite floating-point input, a lot of memory is never used in the std::array<uint32_t,N> objects. In most applications, speed is the most important goal, so the memory usage is a necessary evil. That said, the arbitrary precision library is structured that you can write your own UIntegerType class and instantiate BSNumber or BSRational classes built on it. Such a class might be designed to be more efficient about memory consumption. It could also use the same std::array storage but include a memory manager that manages a heap of such storage objects. Each UIntegerType object can have a pointer member to a std::array rather than its own array, and the memory manager assigns pointers to the members during the arithmetic operations. The memory manager might even partition its heap into subheaps of various maximum sizes, assigning pointers to available blocks that do not waste too much space.

9 Alternatives for Computing Signs of Determinants

The determinant of a 2×2 matrix $A = [a_{ij}]$ is $d = a_{00}a_{11} - a_{10}a_{01}$. The goal is to compute the sign of d accurately. The magnitude of d is irrelevant. In this sense, computing d using exact rational arithmetic is conservative and relatively slow compared to floating-point arithmetic. An alternative that tries to reduce the computational costs is *interval arithmetic*.

9.1 Interval Arithmetic

Interval arithmetic is a system in which you use the floating-point rounding modes to compute an interval that is known to contain the theoretically correct result. In the example for computing the sign of the determinant, if the computed interval does not contain the floating-point zero, then we know the correct sign. If the interval does contain floating-point zero, we must recompute using exact arithmetic. The goal is that the number of recomputations is minimized, as compared to computing everything using exact arithmetic (which is slower).

To illustrate, let $x > 0$ and $y > 0$ be floating-point numbers which are necessarily rational numbers. The theoretically correcct product is $p = xy$, which is also a rational number. However, the number of bits of precision to represent p can be greater than the number provided by the floating-point type, so the computed result $\bar{p}$ will be only an approximation to p. Of course, there are cases where the result is exact. The IEEE

754-2007 floating-point standard requires *correctly rounded* results for multiplication and addition. The floating-point number $\bar{p}$ must be the floating-point number that is closest to the correct result p, where 'closest' depends on the rounding mode. The default rounding mode is round-to-nearest with ties-to-even. We can compute the product twice, each time using a different rounding mode, to obtain an interval that contains p.

```
float x = <a positive number>, y = <a positive number>;
// theoretically correct value of the product is p
set rounding mode to round-down (round towards -infinity);
float pmin = x * y;
set rounding mode to round-up (round towards infinity);
float pmax = x * y;
restore rounding mode to default (round-to-nearest-ties-to-even);
// The interval [pmin,pmax] is guaranteed to contain p.

// theoretically correct value of the sum is s
set rounding mode to round-down (round towards -infinity);
float smin = x + y;
set rounding mode to round-up (round towards infinity);
float smax = x + y;
restore rounding mode to default (round-to-nearest-ties-to-even);
// The interval [smin,smax] is guaranteed to contain s.
```

The determinant involves multiple operations. The two multiplications lead to two intervals. The subtraction requires another interval computed from the first two intervals. we can actually compute the floating-point numbers p_ℓ and p_u as follows. The function for manipulating the floating-point control word is specific to the Microsoft Windows platform.

```
float a00, a01, a10, a11, det;
float minA00A11, maxA00A11, minA10A01, maxA10A01, minDet, maxDet;
unsigned int controlWord;

// Example where theoretical(det) > 0 and floatingpoint(det) > 0.
a00 = 1.0f + ldexp(1.0f, -13);
a01 = 1.0f;
a10 = 1.0f;
a11 = 1.0f + ldexp(1.0f, -14);
det = a00 * a11 - a10 * a01;  // 0.000183105469
_controlfp_s(&controlWord, _RC_DOWN, _MCW_RC);
minA00A11 = a00 * a11;  // 1.00018311
minA10A01 = a10 * a01;  // 1.00000000
_controlfp_s(&controlWord, _RC_UP, _MCW_RC);
maxA00A11 = a00 * a11;  // 1.00018322
maxA10A01 = a10 * a01;  // 1.00000000
_controlfp_s(&controlWord, _RC_DOWN, _MCW_RC);
minDet = minA00A11 - maxA10A01;  // 0.000183105469
_controlfp_s(&controlWord, _RC_UP, _MCW_RC);
maxDet = maxA00A11 - minA10A01;  // 0.000183224678
_controlfp_s(&controlWord, _RC_NEAR, _MCW_RC);
// 0 is not in [minDet,maxDet], so sign(theoretical(det)) = sign(floatingpoint(det)) = +1

// Example where theoretical(det) > 0 and floatingpoint(det) = 0.
a00 = 1.0f + ldexp(1.0f, -13);
a01 = 1.0f;
a10 = 1.0f + ldexp(1.0f, -13) + ldexp(1.0f, -14);
a11 = 1.0f + ldexp(1.0f, -14);
det = a00 * a11 - a10 * a01;  // 0.0, sign(floatingpoint(det)) = 0
_controlfp_s(&controlWord, _RC_DOWN, _MCW_RC);
minA00A11 = a00 * a11;  // 1.00018311
minA10A01 = a10 * a01;  // 1.00018311
_controlfp_s(&controlWord, _RC_UP, _MCW_RC);
maxA00A11 = a00 * a11;  // 1.00018322
maxA10A01 = a10 * a01;  // 1.00018311
_controlfp_s(&controlWord, _RC_DOWN, _MCW_RC);
minDet = minA00A11 - maxA10A01;  // -0.0
_controlfp_s(&controlWord, _RC_UP, _MCW_RC);
maxDet = maxA00A11 - minA10A01;  // 1.19209290e-007
_controlfp_s(&controlWord, _RC_NEAR, _MCW_RC);
```

```
if (minDet <= 0.0f && 0.0f <= maxDet)
{
    // 0 is in [minDet,maxDet], so we do not know whether theoretical(det)
    // is positive or zero.  Recompute using rational arithmetic.
    Rational ra00 = a00, ra01 = a01, ra10 = a10, ra11 = a11;
    Rational rdet = ra00 * ra11 - ra10 * ra01;
    int sign = rdet.GetSign();  // sign(theoretical(det)) = +1
}
```

GTEngine does not yet have an implementation of interval arithmetic. It is on our TODO list.

9.2 Alternate Logic for Signs of Determinants

When using exact rational arithmetic, one may directly compute a determinant for a 2×2 matrix $A = [a_{ij}]$, $d = a_{00}a_{11} - a_{10}a_{01}$ and test the sign. Instead, the alternate logic shown next defers the computation until necessary.

```
int ComputeSignDeterminant2x2(Rational const& a00, Rational const& a10,
    Rational const& a01, Rational const& a11)
{
    int s00s11 = a00.GetSign() * a11.GetSign();
    int s10s01 = a10.GetSign() * a01.GetSign();
    if (s00s11 == 0) { return s10s01; }
    if (s10s01 == 0 || s00s11 != s10s01) { return s00s11; }

    // Simplest response.
    Rational det = a00 * a11 - a10 * a01;
    return det.GetSign();
}
```

With slightly more effort, it is possible to improve the performance. Let $|a_{ij}| = 1.u_{ij} * 2^{e_{ij}}$ with $1.u_{ij} \in [1, 2)$ and define $\sigma = s_{00}s_{11} = s_{10}s_{01}$. The determinant is

$$
\begin{aligned}
d &= \sigma^2(a_{00}a_{11} - a_{10}a_{01}) \\
&= \sigma\left[(1.u_{00} * 1.u_{11}) * 2^{e_{00}+e_{11}} - (1.u_{10} * 1.u_{01}) * 2^{e_{10}+e_{01}}\right] \\
&= \sigma\left[1.v_0 * 2^{p_0} - 1.v_1 * 2^{p_1}\right]
\end{aligned}
$$

where $1.v_i \in [1, 2)$ and

$$
p_0 = \left\{ \begin{array}{ll} e_{00} + e_{11}, & 1.t_{00} * 1.t_{11} < 2 \\ e_{00} + e_{11} + 1, & 1.t_{00} * 1.t_{11} \geq 2 \end{array} \right\}, \quad p_1 = \left\{ \begin{array}{ll} e_{01} + e_{10}, & 1.t_{01} * 1.t_{10} < 2 \\ e_{01} + e_{10} + 1, & 1.t_{01} * 1.t_{10} \geq 2 \end{array} \right\}
$$

The sign is computed as

$$
\mathrm{Sign}(d) = \left\{ \begin{array}{rl} +\sigma, & (p_0 > p_1) \vee ((p_0 = p_1) \wedge (1.v_0 > 1.v_1)) \\ 0, & (p_0 = p_1) \wedge (1.v_0 = 1.v_1) \\ -\sigma, & (p_0 < p_1) \vee ((p_0 = p_1) \wedge (1.v_0 < 1.v_1)) \end{array} \right\}
$$

We can use the exponents to try for an early exit that avoids the multiplications.

```
int ComputeSignDeterminant2x2(Rational const& a00, Rational const& a10,
    Rational const& a01, Rational const& a11)
{
```

```
    int s00s11 = a00.GetSign() * a11.GetSign();
    int s10s01 = a10.GetSign() * a01.GetSign();
    if (s00s11 == 0) { return s10s01; }
    if (s10s01 == 0 || s00s11 != s10s01) { return s00s11; }

    int q0 = a00.GetExponent() + a11.GetExponent();  // p0 = q0 or q0+1
    int q1 = a10.GetExponent() + a01.GetExponent();  // p1 = q1 or q1+1
    if (q0 > q1 + 1) { return s00s11; }
    if (q0 + 1 < q1) { return -s00s11; }

    Rational det = a00 * a11 - a10 * a01;
    return det.GetSign();
}
```

10 Miscellaneous Items

The following two items are of interest. We have not yet tried to implement these.

For a large number of bits of precision, multiplication is typically the bottleneck in computations. For two unsigned integers, each with N words, the product is order $O(N^2)$; that is, multiplication is asymptotically quadratic. Alternatives are discussed in [3, Section 4.3]. Several algorithms are order $O(N^p)$ for $1 < p < 2$. One such algorithm is recursive and order $(N^{\log_2 3})$. If u and v are $2n$-bit numbers, define U_0 and V_0 to be the n low-order bits and U_1 and V_1 to be the n high-order bits, so $u = 2^n U_1 + U_0$ and $v = 2^n V_1 + V_0$. The product of the numbers is

$$uv = \left(2^{2n} + 2^n\right) U_1 V_1 + 2^n \left(U_1 - U_0\right) \left(V_0 - V_1\right) + \left(2^n + 1\right) U_0 V_0$$

We must now compute two differences of n-bit numbers and three products of n-bit numbers. The results must be combined with some shifting and adding. The three products may themselves be computed in the same way; this is the recursive step. Another algorithm uses fast Fourier transforms to compute the product viewed as a convolution of polynomials; it is $O(N \log N)$. Of course, with multiple cores there are other possibilities to use multithreading but the challenge is keeping the threads busy and communicating the results back to the main thread.

Another arbitrary precision system is described in [6]. The BSNumber and BSRational classes uses a *multiple-digit* format. Shewchuk describes a *multiple-component* format that has the advantage that sums of floating-point numbers that require a large number of bits in a multiple-digit format instead can be stored in a small amount of memory. However, the multiple-component format requires more storage per number that is represented. The speed, though, is invariably the bottleneck rather than memory usage. The main focus of the paper is on exact computation of signs for geometric algorithms.

Some books and papers of interest are listed next. The paper most referenced about floating-point arithmetic is [1]. A good book about floating-point arithmetic is [5]. An amazing discussion about floating-point arithmetic in the context of how Java (at that time) chose the wrong approach to it is [2] William Kahan is a Turing Award winner (1989) and the primary architect behind the IEEE 754-2008 standard. A book about computational geometry with a focus on robustness is [4].

References

[1] David Goldberg.
What every computer scientist should know about floating-point arithmetic.

ACM Computing Surveys, 23(1):5–48, March 1991.
https://math.iastate.edu/alex/502/doc/p5-goldberg.pdf.

[2] W. Kahan and J. D. Darcy.
How Java's Floating-Point Hurts Everyone Everywhere.
`http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf`, 1998.

[3] Donald E. Knuth.
The Art of Computer Programming: Seminumerical Algorithms, volume 2.
Addison-Wesley, Reading, Massachusetts, 3rd edition, 1997.

[4] Elmar Langetepe and Gabriel Zachman.
Geometric Data Structures for Computer Graphics.
AK Peters, Ltd, Wellesley, Massachusetts, 2006.

[5] Michael L. Overton.
Numerical Computing with IEEE Floating Point Arithmetic.
Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 2001.

[6] J.R. Shewchuk.
Adaptive precision floating-point arithmetic and fast robust geometric predicates.
Discrete and Computational Geometry, 18:305–363, 10 1997.
http://www.cs.berkeley.edu/∼jrs/papers/robustr.pdf.

[7] Wikipedia.
Methods of Computing Square Roots.
`https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylonian_method`.
accessed November 26, 2017.