Analysis of Integer Square Root Algorithms in Solidity*

Christopher H. Gorman

August 30, 2023

1 Introduction

We will look at various algorithms for computing integer square roots in Solidity and compare their efficiency (gas cost). This work extends [9] by adding more candidate algorithms and improving the analysis. Although the emphasis will be on efficiency and implementation details, we will also note when algorithms are *provably correct*, which is important for such a fundamental operation. A version of these results may be found online [7].

2 Integer Square Root Algorithm Overview

2.1 Definition

Given $a \in \mathbb{N}$, the *integer square root* of a is the unique integer m such that

$$m^2 \leq a < (m+1)^2 . \tag{2.1}$$

This will be denoted as

$$\textsc{Isqrt}(a) \coloneqq m \tag{2.2}$$

and is equivalent to

$$\textsc{Isqrt}(a) = \left\lfloor \sqrt{a} \right\rfloor , \tag{2.3}$$

the integer part of the square root of a.

2.2 Computation Methods

One method for computing integer square roots is based on Newton's method; see [3, Algorithm 1.7.1], [5, Algorithm 9.2.11], and [2, Algorithm 1.13]. Another method based on approximate square roots may be found in [6]. These algorithms usually come with a proof of correctness. For completeness, we reproduce [3, Algorithm 1.7.1] in Algorithm 1, and include a modified version of the proof [3, Algorithm 1.7.1, Proof] in Appendix D.4.1.

*Permanent ID of this document: `021e7116950e28e158ce35151b0dda3a`. Date: 2023.08.30.

Algorithm 1 Integer Square Root algorithm found in [3, Algorithm 1.7.1]. We include the following from [3, Chapter 1.7, Remarks]: "When actually implementing this algorithm, the initialization step must be modified."

Require: n is a positive integer

procedure IntegerSquareRoot(n)
    $x \coloneqq n$
    $y \coloneqq \lfloor (x + n/x)/2 \rfloor$ ▷ All operations are integer operations
    while $x > y$ **do**
        $x \coloneqq y$
        $y \coloneqq \lfloor (x + n/x)/2 \rfloor$
    end while
    return x
end procedure

The focus of this work is on computation within the Ethereum Virtual Machine. There is no native integer square root function, so this must be computed using only basic integer operations. It is important for this fundamental operation to be efficient *and provably correct*. We look at various algorithms found online and compare them to see which is most efficient. Additional algorithms have been included in the analysis which are *provably correct*; none of the algorithms found online have a correctness proof.

3 Integer Square Root Algorithms

3.1 Algorithm Design Principle

The algorithm design philosophy is straightforward: provably correct algorithms which are efficient within the Ethereum Virtual Machine.

3.2 Specific Algorithms in Solidity

3.2.1 Online Algorithms

The specific Isqrt algorithms we found online were `UniswapV2` (Listing 1) [18], `PRB` (Listing 2) [1], `OpenZeppelin` (Listing 3) [14], and `ABDK` (Listing 4) [4]. Two algorithms (`PRB` and `ABDK`) appear to be exactly the same aside from different integer notation and "unchecked" portions. The `OpenZeppelin` algorithm is mathematically equivalent to `PRB` and `ABDK`.

None of these algorithms reference a proof of correctness. We note that `UniswapV2` is provably correct as it essentially implements [3, Algorithm 1.7.1]; compare Listing 1 and Algorithm 1. Unfortunately, `UniswapV2` does not follow the comment in [3, Chapter 1.7, Remarks] which states that "[w]hen actually implementing this algorithm, the initialization step must be modified". `OpenZeppelin` [14] says that we "need at most 7 iteration[s]" while `ABDK` [4] states that "[s]even iterations should be enough"; `PRB` [1] has similar language.

Listing 1: Method for computing Integer Square Roots from `UniswapV2` [18]

```
// License: GPL-3.0
function sqrt(uint y) internal pure returns (uint z) {
    if (y > 3) {
        z = y;
        uint x = y / 2 + 1;
        while (x < z) {
            z = x;
            x = (y / x + x) / 2;
        }
    } else if (y != 0) {
        z = 1;
    }
}
```

3.2.2 Provably Correct Algorithms

Due to the requirement of *provably correct* integer square roots algorithms, additional algorithms were written. All proofs are relegated to Appendix D.

The first provably correct algorithm converts results from [6] into a Solidity algorithm. We denote this as `Python` (Listing 5); this algorithm was previously presented in [9].

The next algorithms come in two groups of 3. They are `Unrolled1` (Listing 6), `Unrolled2` (Listing 7), `Unrolled3` (Listing 8), `While1` (Listing 9), `While2` (Listing 10), and `While3` (Listing 11). A version of `Unrolled3` was previously presented in [9]. Here, `Unrolled` refers to the fact that a fixed number of Newton iterations are performed; `While` refers to the fact that the Newton iterations are iterated within a `while` loop. Labels 1, 2, and 3 refer to the different initialization values: the largest power-of-two less than or equal to integer square root value; the smallest power-of-two greater than the integer square root value; the arithmetic mean of the two previous values.

Additional algorithms include `BitLength` (Listing 12), `Linear` (Listing 13), `Hyper4` (Listing 14), `Lookup4` (Listing 15), and `Lookup8` (Listing 16). `BitLength` uses the full bitlength of the number. `Linear` uses a linear approximation based on the high bits of the input value. `Hyper4` uses a hyperbolic approximation based on the high bits of the input value; we investigated such approximations based on the discussion in [12]. `Lookup4` and `Lookup8` use lookup tables to obtain 4 bits and 8 bits of accuracy for the initial approximation; the idea to use a lookup table for initialization came from [10, 11], although [10] uses a different method for computing square roots (it computes square roots by first computing the *inverse square root*).

3.3 General Algorithm Discussion

When using Newton iterations, a good initial approximation is critical to ensure a low overall cost. The following algorithms use the largest power-of-two less than or equal to the

Listing 2: Method for computing Integer Square Roots from PRB [1]

```
// SPDX-License-Identifier: MIT
function sqrt(uint256 x) internal pure returns (uint256 result) {
    if (x == 0) { return 0; }
    uint256 xAux = uint256(x);
    result = 1;
    if (xAux >= 2 ** 128) {
        xAux >>= 128;  result <<= 64;
    }
    if (xAux >= 2 ** 64) {
        xAux >>= 64;   result <<= 32;
    }
    if (xAux >= 2 ** 32) {
        xAux >>= 32;   result <<= 16;
    }
    if (xAux >= 2 ** 16) {
        xAux >>= 16;   result <<= 8;
    }
    if (xAux >= 2 ** 8) {
        xAux >>= 8;    result <<= 4;
    }
    if (xAux >= 2 ** 4) {
        xAux >>= 4;    result <<= 2;
    }
    if (xAux >= 2 ** 2) {
                       result <<= 1;
    }
    unchecked {
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        uint256 roundedResult = x / result;
        if (result >= roundedResult) {
            result = roundedResult;
        }
    }
}
```

Listing 3: Method for computing Integer Square Roots from `OpenZeppelin` [14]

```
// SPDX-License-Identifier: MIT
function sqrt(uint256 a) internal pure returns (uint256) {
    if (a == 0) {
        return 0;
    }
    uint256 result = 1 << (log2(a) >> 1);
    // At this point `result` is an estimation with one bit of
    // precision. We know the true value is a uint128, since it is
    // the square root of a uint256. Newton's method converges
    // quadratically (precision doubles at every iteration). We
    // thus need at most 7 iteration to turn our partial result
    // with one bit of precision into the expected uint128 result.
    unchecked {
        result = (result + a / result) >> 1;
        result = (result + a / result) >> 1;
        result = (result + a / result) >> 1;
        result = (result + a / result) >> 1;
        result = (result + a / result) >> 1;
        result = (result + a / result) >> 1;
        result = (result + a / result) >> 1;
        return min(result, a / result);
    }
}

function log2(uint256 value) internal pure returns (uint256) {
    uint256 result = 0;
    unchecked {
        if (value >> 128 > 0) { value >>= 128; result += 128; }
        if (value >>  64 > 0) { value >>=  64; result +=  64; }
        if (value >>  32 > 0) { value >>=  32; result +=  32; }
        if (value >>  16 > 0) { value >>=  16; result +=  16; }
        if (value >>   8 > 0) { value >>=   8; result +=   8; }
        if (value >>   4 > 0) { value >>=   4; result +=   4; }
        if (value >>   2 > 0) { value >>=   2; result +=   2; }
        if (value >>   1 > 0) {                result +=   1; }
    }
    return result;
}

function min(uint256 a, uint256 b) internal pure returns (uint256){
    return a < b ? a : b;
}
```

Listing 4: Method for computing Integer Square Roots from `ABDK` [4]

```
// SPDX-License-Identifier: BSD-4-Clause
function sqrt(uint256 x) private pure returns (uint128) {
    unchecked {
        if (x == 0) return 0;
        else {
            uint256 xx = x;
            uint256 r = 1;
            if (xx >= 0x100000000000000000000000000000000) {
                xx >>= 128; r <<= 64;
            }
            if (xx >= 0x10000000000000000) {
                xx >>= 64;  r <<= 32;
            }
            if (xx >= 0x100000000) {
                xx >>= 32;  r <<= 16;
            }
            if (xx >= 0x10000) {
                xx >>= 16;  r <<= 8;
            }
            if (xx >= 0x100) {
                xx >>= 8;   r <<= 4;
            }
            if (xx >= 0x10) {
                xx >>= 4;   r <<= 2;
            }
            if (xx >= 0x4) {
                                r <<= 1;
            }
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            // Seven iterations should be enough
            uint256 r1 = x / r;
            return uint128 (r < r1 ? r : r1);
        }
    }
}
```

Listing 5: Provably correct method for computing Integer Square Roots based on [6]

```
// SPDX-License-Identifier: OBSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        // Here, e represents the bit length
        uint256 e = 1;

        // Here, result is a copy of x to compute the bit length
        uint256 result = x;
        if (result >= (1 << 128)) { result >>= 128; e = 129; }
        if (result >= (1 <<  64)) { result >>=  64; e +=  64; }
        if (result >= (1 <<  32)) { result >>=  32; e +=  32; }
        if (result >= (1 <<  16)) { result >>=  16; e +=  16; }
        if (result >= (1 <<   8)) { result >>=   8; e +=   8; }
        if (result >= (1 <<   4)) { result >>=   4; e +=   4; }
        if (result >= (1 <<   2)) { result >>=   2; e +=   2; }
        if (result >= (1 <<   1)) {                 e +=   1; }

        // e is currently bit length; we overwrite it to scale x
        e = (256 - e) >> 1;

        // m now satisfies 2**254 <= m < 2**256
        uint256 m = x << (2 * e);

        // result now stores the result
        result = 1 + (m >> 254);
        result = (result <<  1) + (m >> 251) / result;
        result = (result <<  3) + (m >> 245) / result;
        result = (result <<  7) + (m >> 233) / result;
        result = (result << 15) + (m >> 209) / result;
        result = (result << 31) + (m >> 161) / result;
        result = (result << 63) + (m >>  65) / result;
        result >>= e;

        if (result * result <= x) {
            return result;
        }
        return result -1;
    }
}
```

Listing 6: Provably correct method for computing Integer Square Roots (`Unrolled1`)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        uint256 xAux = x;

        uint256 result = 1;

        if (xAux >= (1 << 128)) { xAux >>= 128; result = 1 << 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; result <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; result <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; result <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; result <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; result <<=  2; }
        if (xAux >= (1 <<   2)) {               result <<=  1; }

        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;

        if (result * result <= x) {
            return result;
        }
        return result-1;
    }
}
```

Listing 7: Provably correct method for computing Integer Square Roots (`Unrolled2`)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        uint256 xAux = x;

        uint256 result = 2;

        if (xAux >= (1 << 128)) { xAux >>= 128; result = 2 << 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; result <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; result <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; result <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; result <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; result <<=  2; }
        if (xAux >= (1 <<   2)) {               result <<=  1; }

        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;

        if (result * result <= x) {
            return result;
        }
        return result-1;
    }
}
```

Listing 8: Provably correct method for computing Integer Square Roots (`Unrolled3`)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        uint256 xAux = x;

        uint256 result = 1;

        if (xAux >= (1 << 128)) { xAux >>= 128; result = 1 << 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; result <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; result <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; result <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; result <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; result <<=  2; }
        if (xAux >= (1 <<   2)) {               result <<=  1; }
        result = (3 * result) >> 1;

        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;

        if (result * result <= x) {
            return result;
        }
        return result-1;
    }
}
```

Listing 9: Provably correct method for computing Integer Square Roots (`While1`)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }

        uint256 xAux = x;

        uint256 result = 1;

        if (xAux >= (1 << 128)) { xAux >>= 128; result = 1 << 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; result <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; result <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; result <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; result <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; result <<=  2; }
        if (xAux >= (1 <<   2)) {               result <<=  1; }

        result = (result + x / result) >> 1;
        xAux = (result + x / result) >> 1;
        while (xAux < result) {
            result = xAux;
            xAux = (result + x / result) >> 1;
        }
        return result;
    }
}
```

Listing 10: Provably correct method for computing Integer Square Roots (`While2`)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }

        uint256 xAux = x;

        uint256 result = 2;

        if (xAux >= (1 << 128)) { xAux >>= 128; result = 2 << 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; result <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; result <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; result <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; result <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; result <<=  2; }
        if (xAux >= (1 <<   2)) {               result <<=  1; }

        xAux = (result + x / result) >> 1;
        while (xAux < result) {
            result = xAux;
            xAux = (result + x / result) >> 1;
        }
        return result;
    }
}
```

Listing 11: Provably correct method for computing Integer Square Roots (`While3`)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }

        uint256 xAux = x;

        uint256 result = 1;

        if (xAux >= (1 << 128)) { xAux >>= 128; result = 1 << 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; result <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; result <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; result <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; result <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; result <<=  2; }
        if (xAux >= (1 <<   2)) {               result <<=  1; }
        result = (3 * result) >> 1;

        result = (result + x / result) >> 1;
        xAux = (result + x / result) >> 1;
        while (xAux < result) {
            result = xAux;
            xAux = (result + x / result) >> 1;
        }
        return result;
    }
}
```

Listing 12: Provably correct method for computing Integer Square Roots (`BitLength`)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        // Here, e represents the bit length;
        // its value is at most 256, so it could fit in a uint16.
        uint256 e = 1;
        // Here, result is a copy of x to compute the bit length
        uint256 result = x;
        if (result >= (1 << 128)) { result >>= 128; e =  129; }
        if (result >= (1 <<  64)) { result >>=  64; e +=  64; }
        if (result >= (1 <<  32)) { result >>=  32; e +=  32; }
        if (result >= (1 <<  16)) { result >>=  16; e +=  16; }
        if (result >= (1 <<   8)) { result >>=   8; e +=   8; }
        if (result >= (1 <<   4)) { result >>=   4; e +=   4; }
        if (result >= (1 <<   2)) { result >>=   2; e +=   2; }

        if (result >= (1 <<   1)) {
            result = (27 << (e/2)) >> 4;
        } else {
            result = (39 << (e/2)) >> 5;
        }

        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;

        if (result * result <= x) {
            return result;
        }
        return result -1;
    }
}
```

Listing 13: Provably correct method for computing Integer Square Roots (`Linear`) based on linear approximation

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        // Here, e represents the bit length;
        // its value is at most 256, so it could fit in a uint16.
        uint256 e = 1;
        // Here, result is a copy of x to compute the bit length
        uint256 result = x;
        if (result >= (1 << 128)) { result >>= 128; e =  129; }
        if (result >= (1 <<  64)) { result >>=  64; e +=  64; }
        if (result >= (1 <<  32)) { result >>=  32; e +=  32; }
        if (result >= (1 <<  16)) { result >>=  16; e +=  16; }
        if (result >= (1 <<   8)) { result >>=   8; e +=   8; }
        if (result >= (1 <<   4)) { result >>=   4; e +=   4; }
        if (result >= (1 <<   2)) {                 e +=   2; }
        // e is currently bit length; we overwrite it to scale x
        e = (256 - e) >> 1;
        // m now satisfies 2**254 <= m < 2**256
        uint256 m = x << (2 * e);
        // result now stores the result

        // need to initialize result
        result = m >> 254;
        result = (4 + result) << 125;

        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result >>= e;

        if (result * result <= x) {
            return result;
        }
        return result-1;
    }
}
```

Listing 14: Provably correct method for computing Integer Square Roots (`Hyper4`) based on hyperbolic approximation

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        // Here, e represents the bit length;
        // its value is at most 256, so it could fit in a uint16.
        uint256 e = 1;
        // Here, result is a copy of x to compute the bit length
        uint256 result = x;
        if (result >= (1 << 128)) { result >>= 128; e =  129; }
        if (result >= (1 <<  64)) { result >>=  64; e +=  64; }
        if (result >= (1 <<  32)) { result >>=  32; e +=  32; }
        if (result >= (1 <<  16)) { result >>=  16; e +=  16; }
        if (result >= (1 <<   8)) { result >>=   8; e +=   8; }
        if (result >= (1 <<   4)) { result >>=   4; e +=   4; }
        if (result >= (1 <<   2)) {                 e +=   2; }
        // e is currently bit length; we overwrite it to scale x
        e = (256 - e) >> 1;
        // m now satisfies 2**254 <= m < 2**256
        uint256 m = x << (2 * e);
        // result now stores the result

        // need to initialize result
        result = m >> 252;
        result = (512/(31 - result)) << 123;

        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result >>= e;

        if (result * result <= x) {
            return result;
        }
        return result-1;
    }
}
```

Listing 15: Method for computing Integer Square Roots (`Lookup4`) based on lookup table

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        // Here, e represents the "approximate" bit length
        uint256 e = 1;

        // Here, result is a copy of x to compute the bit length
        uint256 result = x;
        if (result >= (1 << 128)) { result >>= 128; e =  129; }
        if (result >= (1 <<  64)) { result >>=  64; e +=  64; }
        if (result >= (1 <<  32)) { result >>=  32; e +=  32; }
        if (result >= (1 <<  16)) { result >>=  16; e +=  16; }
        if (result >= (1 <<   8)) { result >>=   8; e +=   8; }
        if (result >= (1 <<   4)) { result >>=   4; e +=   4; }
        if (result >= (1 <<   2)) {                 e +=   2; }

        // overwrite e for scaling parameter
        e = (256 - e) >> 1;
        // m now satisfies 2**254 <= m < 2**256
        uint256 m = x << (2 * e);

        // need to initialize result
        result = uint256(uint8(lookup_table_4[(m >> 252)])) << 123;

        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result >>= e;

        if (result * result <= x) {
            return result;
        }
        return result-1;
    }
}

bytes constant lookup_table_4 =
"\x00\x00\x00\x00\x11\x13\x15\x16\x17\x19\x1a\x1b\x1c\x1d\x1e\x1f";
```

Listing 16: Method for computing Integer Square Roots (`Lookup8`) based on lookup table

```
// SPDX-License-Identifier: OBSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        // Here, e represents the "approximate" bit length
        uint256 e = 1;

        // Here, result is a copy of x to compute the bit length
        uint256 result = x;
        if (result >= (1 << 128)) { result >>= 128; e =  129; }
        if (result >= (1 <<  64)) { result >>=  64; e +=  64; }
        if (result >= (1 <<  32)) { result >>=  32; e +=  32; }
        if (result >= (1 <<  16)) { result >>=  16; e +=  16; }
        if (result >= (1 <<   8)) { result >>=   8; e +=   8; }
        if (result >= (1 <<   4)) { result >>=   4; e +=   4; }
        if (result >= (1 <<   2)) {                 e +=   2; }

        // overwrite e for scaling parameter
        e = (256 - e) >> 1;
        // m now satisfies 2**254 <= m < 2**256
        uint256 m = x << (2 * e);

        // need to initialize result
        result = 2**127 +
                 (uint256(uint8(lookup_table_8[(m>>248)])) << 119);

        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result >>= e;

        if (result * result <= x) {
            return result;
        }
        return result-1;
    }
}

bytes constant lookup_table_8 = "...";
```

integer square root: `PRB` (Listing 2), `OpenZeppelin` (Listing 3), `ABDK` (Listing 4), `Unrolled1` (Listing 6), and `While1` (Listing 9). Another good initialization value is shared by `Unrolled2` (Listing 7) and `While2` (Listing 10), which follow the initialization suggestions of [3, Chapter 1.7, Remarks (2)] and [5, Algorithm 9.2.11]). `Unrolled3` (Listing 8) and `While3` (Listing 11) use the new initialization from [9]. `BitLength` (Listing 12), `Linear` (Listing 13), `Hyper4` (Listing 14), `Lookup4` (Listing 15), and `Lookup8` (Listing 16) use initializations specific to each method. `UniswapV2` (Listing 1) has the worst initialization value (the input value itself).

`Python` (Listing 5) is based on approximate square roots [6].

4 Comparison of Gas Costs

Because all operations are performed within the Ethereum Virtual Machine, the primary cost metric is the overall gas cost. Thus, the focus is on minimizing the total gas used. Ideally, the best algorithm will have the lowest maximum, mean, and median gas costs. We note that the results here are different from those in [9] for three reasons: first, some of the algorithms are different (and we chose the most recent version of those found online); second, we ensure that all input values are unique (certain values may have been double counted); third, different input values were chosen.

4.1 Data Point Selection

Any gas cost comparison requires the selection of `uint256` values for input. The values were chosen in this way:

- all numbers of the form 2^k, $2^k - 1$, and $2^k + 1$ for nonnegative values of k;
- $v - 1$, v, and $v + 1$ for $v = (2^{128} - 1)^2$ (these were chosen because of an edge case in the new algorithms);
- random values chosen according to the loguniform distribution [16] on $[1, 2^{256}]$ when the random seed is initialized to 0 [13].

The number of random samples were increased until there were 2048 unique values. While it is clear that the particular distribution affects the results, we believe choosing the input values in this way reduces the risk of bias. There were a total of 768 deterministic values chosen and a total of 1303 random samples determined the remaining 1280 values.

In addition to measuring the gas cost, the result of each integer square root operation was validated using Python's integer square root [6, 15]. There was no instance in which an algorithm ever produced an incorrect result; this is, of course, not a proof that `PRB`, `OpenZeppelin`, and `ABDK` are correct, but rather that there are no known values where these algorithms fail.

4.2 Summary Statistics

A listing of the summary statistics describing the runs may be found in Tables 1, 2, and 3. From here, we see that the algorithm with the smallest maximum gas cost is `Linear` while

	UniswapV2	PRB	OpenZeppelin	ABDK	Python
Max	34 205	878	1021	881	959
Mean	17 734	795	950	803	883
Median	17 639	798	949	803	884
Std	9558	34	30	33	43

Table 1: Here are statistics related to the gas cost data. The `UniswapV2` and `Python` algorithms are provably correct. These results are for the tests in Section 4.

	Unrolled1	Unrolled2	Unrolled3	While1	While2	While3
Max	845	838	817	1202	1154	1132
Mean	769	768	742	817	873	833
Median	773	773	745	860	909	856
Std	41	40	40	175	155	135

Table 2: Here are statistics related to the gas cost data; all of these algorithms are provably correct. These results are for the tests in Section 4.

	BitLength	Linear	Hyper4	Lookup4	Lookup8
Max	841	812	845	922	925
Mean	768	746	778	855	858
Median	770	751	784	861	864
Std	38	37	38	41	41

Table 3: Here are statistics related to the gas cost data. All of these algorithms are provably correct. These results are for the tests in Section 4.

Total	2048
`UniswapV2`	2
`Python`	5
`Unrolled1`	5
`Unrolled2`	5
`Unrolled3`	738
`While1`	383
`While2`	189
`While3`	294
`BitLength`	5
`Linear`	453
`Hyper4`	5
`Lookup4`	5
`Lookup8`	5

Table 4: Here are the number of times each method had minimal gas cost; methods not included were never minimal. These results are for the tests in Section 4.

the smallest mean and median gas costs is `Unrolled3`. The standard deviations of `Linear` and `Unrolled3` are reasonable.

4.3 Detailed Analysis

We now take a closer look at how the gas cost varies with the argument.

For each value we tested, we determined the minimal gas cost and then counted the number of times each algorithm was minimal; the results may be found in Table 4. Additionally, we show a histogram where certain algorithms were minimal in Figure 1. There does not appear to be a particular pattern in the value distribution. The main trend is that `Unrolled3` consistently performs well across the range of arguments; `Linear` performs well, too. The only region where this does not hold is for small arguments, where other algorithms (`While`) performed better; this makes sense because those values require fewer Newton iterations.

Overall, `Unrolled3` has the lowest overall cost for a plurality of points (and close to lowest gas cost), so we declare this to be the algorithm of choice. A more extensive analysis may be found in Appendix A; the results are similar.

5 Conclusion

After comparing various algorithms for total gas cost, we found that the most efficient algorithm based on the sampled values is `Unrolled3` (Listing 8). A proof of correctness may be found Appendix D. `Linear` (Listing 13) is a close second-place algorithm.

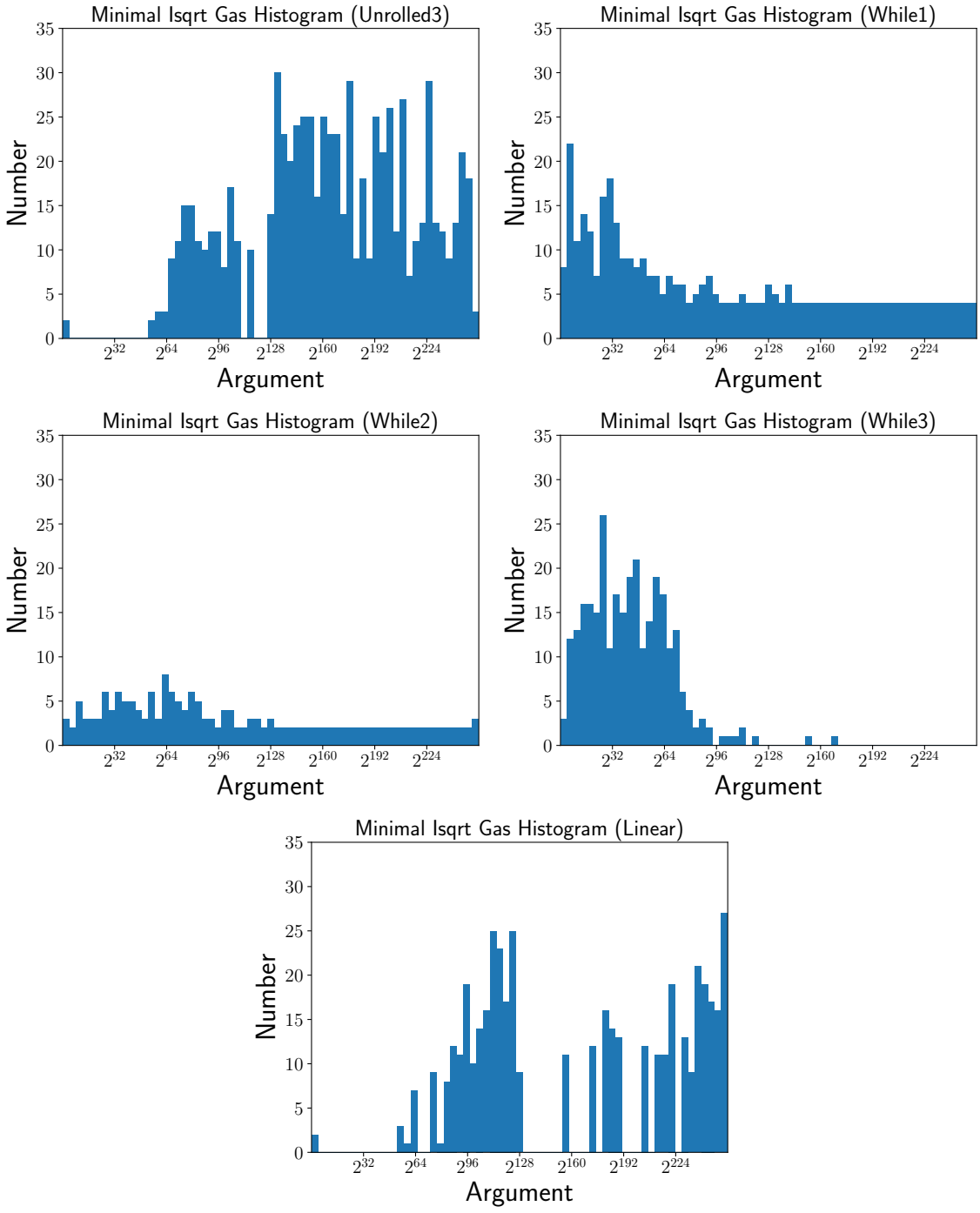

Figure 1: Here we plot a histogram showing where each algorithm is minimal and provides more detail to the results in Table 4. Each bin counts the total number instances where the algorithm's gas cost was minimal. These results are for the tests in Section 4.

References

[1] Paul Razvan Berg. *Integer Square Root Algorithm in Solidity*. June 2023. URL: https://github.com/PaulRBerg/prb-math/blob/28055f6cd9a2367f9ad7ab6c8e01c9ac8e9acc61/src/Common.sol#L595.

[2] Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Version 0.5.1. 2010. DOI: https://doi.org/10.48550/arXiv.1004.4710.

[3] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Graduate Texts in Mathematics 138. Springer, 1993.

[4] ABDK Consulting. *Integer Square Root Algorithm in Solidity*. Sept. 2020. URL: https://github.com/abdk-consulting/abdk-libraries-solidity/blob/9e69beb255e2f87d67b44047dABDKMath64x64.sol#L727.

[5] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. 2nd ed. Springer, 2005.

[6] Mark Dickinson. *Python's integer square root algorithm*. Jan. 2022. URL: https://github.com/mdickinson/snippets/blob/master/papers/isqrt/isqrt.pdf.

[7] Christopher H. Gorman. *Analysis of Integer Square Root Algorithms in Solidity*. 2023. URL: https://github.com/chgorman/isqrt-gas.

[8] Chun-Hua Guo. "On Newton's method and Halley's method for the principal pth root of a matrix". In: *Linear Algebra and its Applications* 432.8 (2010), pp. 1905–1922. DOI: https://doi.org/10.1016/j.laa.2009.02.030.

[9] Christopher H. Gorman. *Efficient Integer Square Roots in Solidity*. Nov. 2022. URL: https://github.com/alicenet/.github/blob/main/docs/efficient_isqrt.pdf.

[10] Guillaume Melquiond and Raphaël Rieu-Helft. "Formal Verification of a State-of-the-Art Integer Square Root". In: *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE. 2019, pp. 183–186. DOI: https://doi.org/10.1109/ARITH.2019.00041. URL: https://inria.hal.science/hal-02092970/file/main.pdf.

[11] *Methods of computing square roots: Binary estimates*. URL: https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Binary_estimates (visited on 08/14/2023).

[12] *Methods of computing square roots: Hyperbolic estimates*. URL: https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Hyperbolic_estimates (visited on 08/14/2023).

[13] Numpy. *numpy.random.seed*. URL: https://numpy.org/doc/stable/reference/random/generated/numpy.random.seed.html (visited on 07/16/2023).

[14] OpenZeppelin. *Integer Square Root Algorithm in Solidity*. July 2023. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/6bf68a41d19f3d6e364d8c207cb7d1acontracts/utils/math/Math.sol#L220.

[15] Python. *Mathematical Functions*. URL: https://docs.python.org/3/library/math.html#math.isqrt (visited on 07/16/2023).

[16] Scipy. *scipy.stats.loguniform*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.loguniform.html (visited on 07/16/2023).

[17] J. Michael Steele. *The Cauchy-Schwarz Master Class: An Introduction to the Art of Mathematical Inequalities*. Cambridge University Press, 2004.

[18] Uniswap. *Integer Square Root Algorithm in Solidity*. Jan. 2020. URL: https://github.com/Uniswap/v2-core/blob/585ee2ef824db671b71e351a91860a003c05431d/contracts/libraries/Math.sol#L11.

[19] Mikhail Vladimirov. *Math in Solidity (Part 3: Percents and Proportions)*. URL: https://medium.com/coinmonks/math-in-solidity-part-3-percents-and-proportions-4db014e080b1 (visited on 08/21/2023).

A Extended Analysis

While the results in Section 4 are good, it is desirable to have a more extensive analysis. The only algorithms tested are `Unrolled3` (Listing 8), `While1` (Listing 9), `While2` (Listing 10), `While3` (Listing 11), and `Linear` (Listing 13), as these performed the best: they had a significant number of values which were minimal.

The deterministic and random values will be separated. All values tested in Section 4 are included in these tests.

A.1 Deterministic Tests

A.1.1 Overview

We test additional deterministic values. First, we include all deterministic values from Section 4. Additional values are of the form $2^k + 2^j$ for $j, k \in \mathbb{N}$ and $j < k$. The total number of values were 33154. The results are shown in Table 5 and Figure 2.

The minimal values have a somewhat different trend than before: `Unrolled3`, `Linear`, and `While1` all perform well. By looking at Tables 6 and 7, we see that `Unrolled3` and `Linear` "steal" minimal values from each other. A direct comparison between `Unrolled3` and `Linear` may be found in Table 8, where `Unrolled3` has the majority of minimum values. The fact that `While1` performs the best in Table 5 may be explained by the initialization value in this test frequently starts *very close* to the correct value; see the discussion in Appendix A.1.2. It is interesting that `Unrolled3` and `Linear` performed as well as they did in this situation which favors `While1`.

The results here show that it is easy to (unwittingly) test values which are biased toward one particular algorithm. This is the reason for testing a large collection of random values in Appendix A.2.

A.1.2 Detailed Look

We now take a closer look at particular values: we look at all values $2^{254} \leq v < 2^{255}$ and then $2^{255} \leq v < 2^{256}$. In this instance, we are just comparing the gas cost of `Unrolled3` and `While1`. We expect these two collections to be characteristic of the remaining gas values.

Values $2^{254} \leq v < 2^{255}$ For `Unrolled3`, there are only two gas costs in this range: 810 and 817. `While1` starts off at a low gas cost of 572; it then jumps to 662, 752, 842, 932, and 1022. We estimate that each Newton iteration costs 90 gas.

Values $2^{255} \leq v < 2^{256}$ For `Unrolled3`, the gas cost is predominantly 810; there is one instance of 817, and three values of 168 (from early exit). Almost all gas cost values for `While1` are 1022, with some larger values.

A.2 Loguniform Tests

The tests in Section 4 ran 1280 values which came from a loguniform distribution [16]. Here, we perform the same test using more samples; in particular, we use $16384 = 2^{14}$ random values from 16808 samples. The results are shown in Table 9 and Figure 3. In this case, both `Unrolled3` and `Linear` perform well. By looking at Tables 10 and 11, we see that `Unrolled3` and `Linear` "steal" minimal values from each other. A direct comparison between `Unrolled3` and `Linear` may be found in Table 12, where `Unrolled3` has the majority of minimum values.

There is no significant difference between the results in Tables 4 and 9 or Figures 1 and 3. `Unrolled3` has a plurality of the minimal cost for random values.

A.3 Conclusion

This extended analysis gives additional insight but with the same result: `Unrolled3` is the best algorithm for a generic `uint256` value.

B Additional Algorithms and Further Optimizations

B.1 Additional Algorithms

We discuss potential lines of algorithmic improvement; see the discussion in Appendix C about a variant of `Unrolled3`.

B.1.1 Polynomial Approximations

The results of `Linear` in Table 3 shows that a linear approximation provides a good initial approximation. It may be possible to extend this linear (affine) approximation to higher degree polynomials; however, the author thinks that will not be productive. This follows from the fact that a more complex initialization algorithm will require additional gas and Newton iterations are pretty cheap (48 gas). Almost 8 bits of accuracy is needed to remove *one* Newton iteration; the author does not think such an accurate approximation is possible with only 48 gas.

B.1.2 Rational Approximations

The results `Hyper4` in Table 3 shows that hyperbolas provide a good initial approximation; however, `Hyper4` is not more efficient than `Unrolled3` (or `Linear`). It seems unlikely that

Total	33 154
`Unrolled3`	10 103
`While1`	15 639
`While2`	130
`While3`	1625
`Linear`	5671

Table 5: Here are the number of times each method had minimal gas cost; these are results for the additional deterministic values based around powers-of-two. These results are for the tests in Appendix A.1.

Total	33 154
`Unrolled3`	15 769
`While1`	15 639
`While2`	130
`While3`	1625

Table 6: Here are the number of times each method had minimal gas cost; these are results for the additional deterministic values based around powers-of-two. These results are for the tests in Appendix A.1 without `Linear`.

Total	33 154
`While1`	15 639
`While2`	130
`While3`	1625
`Linear`	15 769

Table 7: Here are the number of times each method had minimal gas cost; these are results for the additional deterministic values based around powers-of-two. These results are for the tests in Appendix A.1 without `Unrolled3`.

Total	33 154
`Unrolled3`	21 818
`Linear`	11 410

Table 8: Here are the number of times each method had minimal gas cost; these are results for the additional deterministic values based around powers-of-two. These results are for the tests in Appendix A.1 comparing `Unrolled3` and `Linear`.

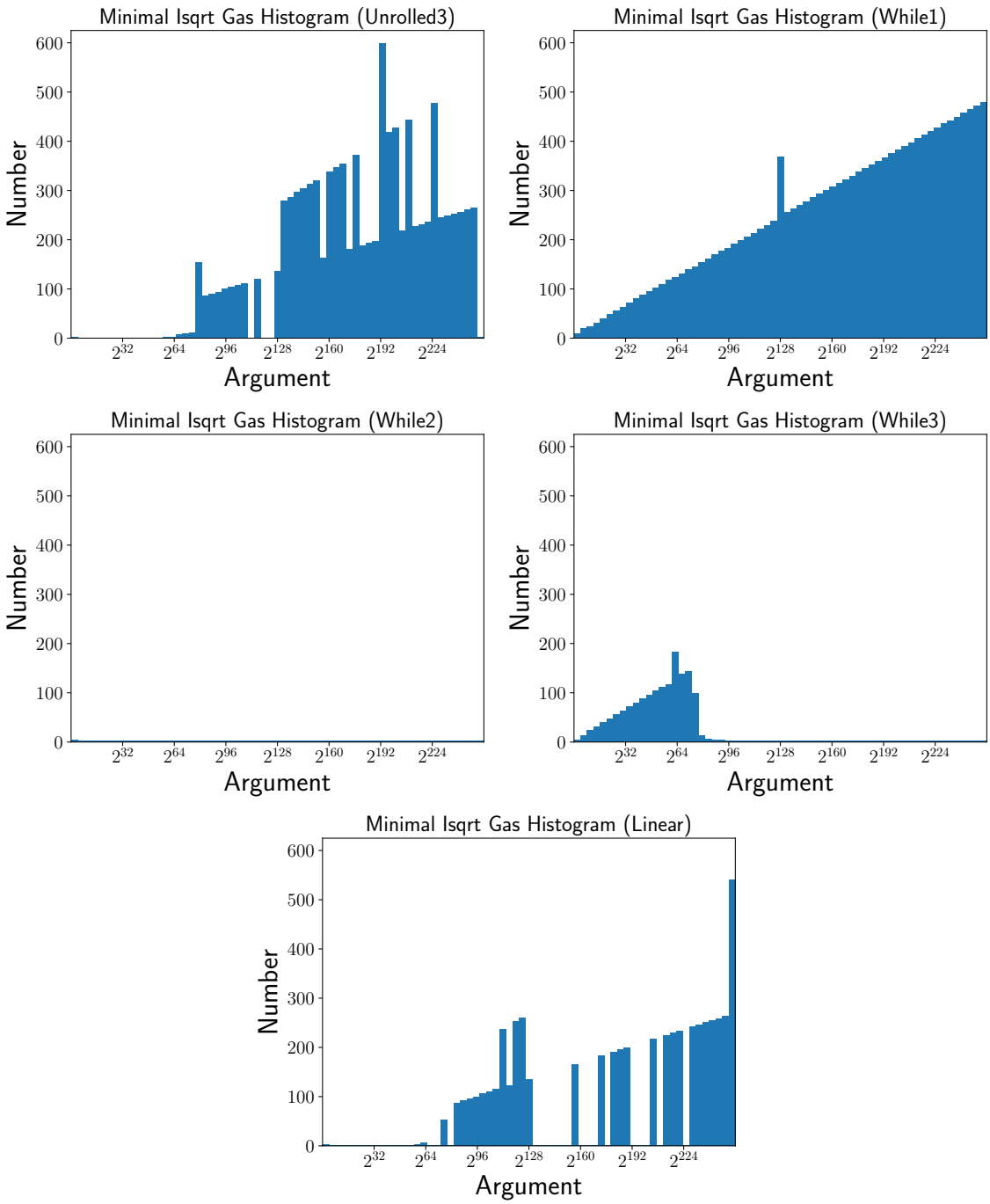

Figure 2: Here we plot a histogram showing where each algorithm is minimal when using a larger sample of deterministic values. Each bin counts the total number instances where the algorithm's gas cost was minimal. These results are for the tests in Appendix A.1.

Total	16 384
`Unrolled3`	7612
`While1`	1537
`While2`	697
`While3`	2347
`Linear`	4195

Table 9: Here are the number of times each method had minimal gas cost; these are results for the additional loguniform random values. These results are for the tests in Appendix A.2.

Total	16 384
`Unrolled3`	11 806
`While1`	1537
`While2`	697
`While3`	2347

Table 10: Here are the number of times each method had minimal gas cost; these are results for the additional loguniform random values. These results are for the tests in Appendix A.2 without `Linear`.

Total	16 384
`While1`	1537
`While2`	697
`While3`	2347
`Linear`	11 806

Table 11: Here are the number of times each method had minimal gas cost; these are results for the additional loguniform random values. These results are for the tests in Appendix A.2 without `Unrolled3`.

Total	16 384
`Unrolled3`	11 291
`Linear`	5094

Table 12: Here are the number of times each method had minimal gas cost; these are results for the additional loguniform random values. These results are for the tests in Appendix A.2 comparing `Unrolled3` and `Linear`.

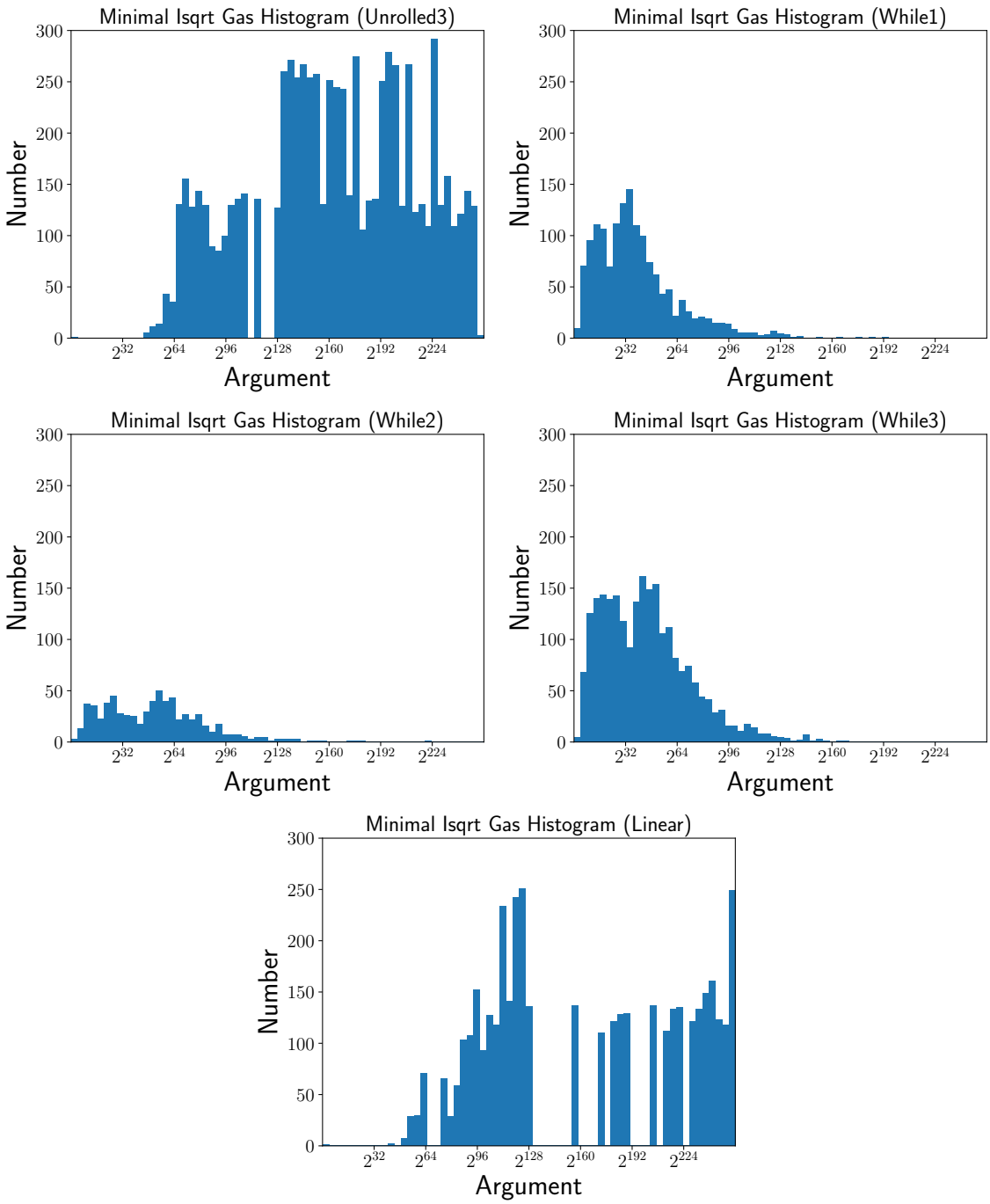

Figure 3: Here we plot a histogram showing where each algorithm is minimal when using a larger sample of loguniform random values. Each bin counts the total number instances where the algorithm's gas cost was minimal. These results are for the tests in Appendix A.2.

extending this method (to higher degree rational functions) would produce a more efficient result: as mentioned in Appendix B.1.1, Newton iterations are cheap and good approximations are expensive.

B.1.3 Lookup Tables

The results in Table 3 show `Lookup4` and `Lookup8` produce decent results; however, the gas costs are essentially the same even though `Lookup8` uses one less Newton iteration than `Lookup4`. This leads us to expect that higher precision lookup tables will not result in lower gas costs.

In [10], the method for computing square roots is actually based around computing *inverse square roots*. We note, though, that the setting and constraints are different as [10] focuses on the inverse square root algorithm in order to not have to perform division. Within the Ethereum Virtual Machine, the cost of integer multiplication and division are the same; thus, although it would be interesting to implement the algorithm, the author does not expect it to produce an efficient algorithm (that is, better than `Unrolled3` or `Linear`).

B.2 Higher-Order Approximations

We have primarily focused on variations of Newton's method. We now briefly look at higher-order methods. In particular, we look at Halley's method for computing square roots [8, Section 2]. We mentioned that Newton's method has *quadratic convergence*: correct digits doubling approximately every iteration. Halley's method has *cubic convergence*: correct digits *tripling* approximately every iteration.

Although the increase in the rate of convergence is desirable, from the discussion which follows it appears that this method (and others like it) will not give us a more efficient algorithm due to the increased complexity of the calculation.

When solving $x^2 - \alpha = 0$ (that is, computing square roots), Newton's method reduces to

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{\alpha}{x_n}\right). \tag{B.1}$$

When solving $x^2 - \alpha = 0$ using Halley's method, we have the recursive formula

$$x_{n+1} = \frac{x_n^3 + 3\alpha x_n}{3x_n^2 + \alpha}. \tag{B.2}$$

If we set $\varepsilon = t - \sqrt{\alpha}$ and

$$h_\alpha(t) = \frac{t^3 + 3\alpha t}{3t^2 + \alpha}, \tag{B.3}$$

then we see

$$h_\alpha(t) - \sqrt{\alpha} = \frac{\varepsilon^3}{3t^2 + \alpha}. \tag{B.4}$$

The "ε^3" shows the cubic convergence of the algorithm.

When attempting to perform this operation within Solidity, the first problem we notice in Eq. (B.2) is that both the numerator and denominator will overflow when α is large: $\alpha \geq 2^{180}$ for the numerator; $\alpha \geq 2^{255}$ for the denominator. While this could be handled, that additional cost alone renders this algorithm impractical. Thus, these methods do not appear relevant. An optimized form of `mulDiv` ($\lfloor (a \cdot b)/c \rfloor$ with no overflow) costs 550 gas per call [19].

B.3 Detailed Gas Cost

To better see where gas is being spent, we include a detailed analysis of the gas costs associated with computing

$$\begin{aligned} \left\lfloor \sqrt{2^{255}} \right\rfloor &= 240615969168004511545033772477625056927 \\ &= \texttt{0xb504f333f9de6484597d89b3754abe9f}. \end{aligned} \tag{B.5}$$

We let $x = 2^{255}$ throughout to simplify our discussion.

B.3.1 Initialization

The main part of the initialization x is to determine the correct order-of-magnitude. In particular, we want to compute e so that

$$2^{e-1} \leq \sqrt{x} < 2^e. \tag{B.6}$$

We note that this is done implicitly in most instances. Computing this estimate for x costs 426 gas; it is unclear if this could be reduced by any significant amount.

B.3.2 Newton Iterations

The cost of each unrolled Newton iteration costs 48 gas. A Newton iteration inside a `while` loop is estimated to be 90 gas from A.1.2. Thus, using a fixed number of Newton iterations is more efficient in general.

B.3.3 Final Check

A final check ensures a valid result is returned (checking if $r^2 \leq x$) is estimated to cost 29 gas, and there does not appear to be a way to make this less expensive. The return logic will be discussed in Appendix D.3.

B.4 Potential Optimizations

Based on the cost of unrolled Newton iterations (48 gas per unrolled iteration and 90 gas within a `while` loop), reducing the number of Newton iterations only reduces the overall gas cost so much.

Additional Approximations As mentioned in Appendix B.1, it is expected that polynomial estimates, rational estimates, and lookup tables will not lead to an overall reduction in gas. It is unclear where additional approximation methods would come from. The best algorithms presented here (`Unrolled3` (Listing 8) followed closely by `Linear` (Listing 13)) produce 2 or 4 bits of initial precision.

Assembly One way to reduce gas is to write all algorithms in assembly. This is not completely separate from algorithm development, as it may be the case that optimized (assembly) versions of algorithms have a different ordering. This requires a thorough knowledge of Solidity assembly and the result should be audited to ensure the algorithm is properly implemented.

C Variants of `Unrolled3`

We include an additional algorithm `Unrolled3v2` (Listing 17), which is a modified version of the `Unrolled3` algorithm.

The results for the same analyses (standard and extended) between `Unrolled3` and `Unrolled3v2` are included in Tables 13, 14, 15, and 16. As we can see, there is no clear-cut winner. `Unrolled3v2` has a lower maximum gas cost while `Unrolled3` has lower mean and median gas cost. Furthermore, although `Unrolled3` has minimal gas cost in the majority of instances (between 55% and 57% of the time), this is not a large margin. It is unclear to the author why exactly this is the case. He chose to use `Unrolled3` as the recommended algorithm over `Unrolled3v2` because of the lower average gas cost and majority of minimal gas cost.

D Proofs

Here we include proofs for the algorithms. This includes some of the material from [9, Appendix B]. A notable omission is the proof of `Python` (Listing 5); this is due to the fact that the emphasis here is on algorithms based on Newton iteration. A proof of correctness for `Python` may be found in [9, Appendix C]. Some of the work here will be similar to that found in [9, Appendix A] but is included for this document to be self-contained.

D.1 Mathematical Review

D.1.1 Newton Iteration over the Real Numbers

Fix $\alpha > 0$. For any $t > 0$, we define

$$f_\alpha(t) \coloneqq \frac{1}{2}\left(t + \frac{\alpha}{t}\right). \tag{D.1}$$

We always have

$$f_\alpha(t) = \frac{1}{2}\left(t + \frac{\alpha}{t}\right) \geq \sqrt{\alpha}. \tag{D.2}$$

	Unrolled3	Unrolled3v2
Max	817	809
Mean	742	743
Median	745	748
Std	40	36

Table 13: Here are statistics related to the gas cost data for `Unrolled3` and `Unrolled3v2` in terms of maximum, mean, and median. These results are for the tests in Section 4.

Total	2048
Unrolled3	1149
Unrolled3v2	904

Table 14: Here are number of times when `Unrolled3` and `Unrolled3v2` had minimal gas cost. These results are for the tests in Section 4.

Total	33 154
Unrolled3	18 276
Unrolled3v2	14 883

Table 15: Here are number of times when `Unrolled3` and `Unrolled3v2` had minimal gas cost. These results are for the tests in Appendix A.1.

Total	16 384
Unrolled3	9321
Unrolled3v2	7064

Table 16: Here are number of times when `Unrolled3` and `Unrolled3v2` had minimal gas cost. These results are for the tests in Appendix A.2.

Listing 17: Provably correct method for computing Integer Square Roots (`Unrolled3v2`), a variant of `Unrolled3`

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        // Here, e represents the "approximate" bit length
        uint256 e = 1;

        // Here, result is a copy of x to compute the bit length
        uint256 result = x;
        if (result >= (1 << 128)) { result >>= 128; e =  129; }
        if (result >= (1 <<  64)) { result >>=  64; e +=  64; }
        if (result >= (1 <<  32)) { result >>=  32; e +=  32; }
        if (result >= (1 <<  16)) { result >>=  16; e +=  16; }
        if (result >= (1 <<   8)) { result >>=   8; e +=   8; }
        if (result >= (1 <<   4)) { result >>=   4; e +=   4; }
        if (result >= (1 <<   2)) {                 e +=   2; }

        result = (3 << (e/2)) >> 1;

        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;

        if (result * result <= x) {
            return result;
        }
        return result-1;
    }
}
```

This follows from the inequality of the arithmetic and geometric means (AM-GM inequality); one reference is [17]. Given $t > 0$, we can look at the sequence $\{t_k\}_{k=0}^{\infty}$ defined by

$$t_k = \begin{cases} t & k = 0 \\ f_\alpha(t_{k-1}) & k > 0 \end{cases}. \tag{D.3}$$

Each application of $f_\alpha(\cdot)$ is called a *Newton iteration*. This sequence converges to $\sqrt{\alpha}$: $\lim_{k\to\infty} t_k = \sqrt{\alpha}$.

In order to see how the error changes with each iteration, we set

$$\varepsilon \coloneqq t - \sqrt{\alpha}. \tag{D.4}$$

From here, we find

$$f_\alpha(t) - \sqrt{\alpha} = \frac{\varepsilon^2}{2t}. \tag{D.5}$$

The "ε^2" in the error term shows what we mean when we say the algorithm has quadratic convergence.

We make another important observation: when $t \geq \sqrt{\alpha}$, we have

$$\begin{aligned} f_\alpha(t) - \sqrt{\alpha} &= \frac{\varepsilon^2}{2t} \\ &\leq \frac{\varepsilon^2}{2\sqrt{\alpha}}. \end{aligned} \tag{D.6}$$

This fact allows us to simplify the error bounds we compute. We note that all values of the sequence in Eq. (D.3) satisfy $t_k \geq \sqrt{\alpha}$ with the possible exception is t_0.

D.1.2 Newton Iteration over the Integers

The fact that we care about *integer operations* within the Ethereum Virtual Machine means we focus on a slightly different function. Fix $n \in \mathbb{N}$ and let $x \in \mathbb{N}$. We define

$$g_n(x) \coloneqq \left\lfloor \frac{1}{2}\left(x + \frac{n}{x}\right) \right\rfloor. \tag{D.7}$$

This represents the integer version of f_α.

D.2 Error Bounds

Based on the above work, we will now compute the error bounds for a number of algorithms. The computations are similar, and we primarily focus on the initial error and the first Newton iteration, as these are most important. Only when greater care is required to prove the desired error bounds do we go into greater detail.

Throughout this section, we are assuming that $\sqrt{n}$ is an e-bit number:

$$2^{e-1} \leq \sqrt{n} < 2^e. \tag{D.8}$$

This allows us to state the initialization value; different initialization *algorithms* may arrive at the same initialization *value*.

Throughout, we will specify the initialization x_0 and then look at

$$x_k = g_n(x_{k-1}), \quad k > 0. \tag{D.9}$$

We are interested in the values

$$\varepsilon_k \coloneqq x_k - \sqrt{n}\,. \tag{D.10}$$

These values will allow us to determine when x_k has reached $\textsc{Isqrt}(n)$.

Technically, we care about these error values:

$$\hat{\varepsilon}_k \coloneqq x_k - \left\lfloor \sqrt{n} \right\rfloor. \tag{D.11}$$

This is because we are computing *integer square roots*, not square roots. At the same time, we note that when $\varepsilon_k \leq 1$,

$$x_k \in \{q, q+1\}, \quad q = \left\lfloor \sqrt{n} \right\rfloor. \tag{D.12}$$

This follows because $x_k \in \mathbb{N}$ and $|\sqrt{n} - \lfloor \sqrt{n} \rfloor| < 1$. Thus, we focus on determining when $\varepsilon_k \leq 1$. Any attempt to get a smaller error bound is pointless because it is less expensive to check if x_k satisfies $x_k^2 \leq n$.

In each error bound calculation, we will assume that $x_k > \sqrt{n}$; if this ever fails, we know that $x_k = \lfloor \sqrt{n} \rfloor$.

D.2.1 `Unrolled1`

The initialization value is chosen to be the largest power-of-two less than or equal to $\sqrt{n}$:

$$x_0 = 2^{e-1}. \tag{D.13}$$

We have the error bound

$$|\varepsilon_0| \leq 2^{e-1}. \tag{D.14}$$

We now use this to bound ε_1:

$$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq 2^{e-2}. \end{aligned} \tag{D.15}$$

Continuing, we find

$$\begin{aligned} \varepsilon_2 &\leq 2^{e-4} \\ \varepsilon_3 &\leq 2^{e-8} \\ &\vdots \end{aligned} \tag{D.16}$$

D.2.2 `Unrolled2`

The initialization value is chosen to be the smallest power-of-two greater than to $\sqrt{n}$:

$$x_0 = 2^e. \tag{D.17}$$

Note that strict inequality: we have $x_0 > \sqrt{n}$. We have the error bound

$$\varepsilon_0 \leq 2^{e-1}. \tag{D.18}$$

We now use this to bound ε_1:

$$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq 2^{e-3}. \end{aligned} \tag{D.19}$$

Continuing, we find

$$\begin{aligned} \varepsilon_2 &\leq 2^{e-6} \\ \varepsilon_3 &\leq 2^{e-12} \\ &\vdots \end{aligned} \tag{D.20}$$

This shows that `Unrolled2` converges slightly faster (has smaller error) than `Unrolled1`. This is due solely to the larger initial starting value.

D.2.3 `Unrolled3`

The initialization value is chosen to be the arithmetic mean of the initializations for `Unrolled1` and `Unrolled2`:

$$x_0 = 2^{e-1} + 2^{e-2}. \tag{D.21}$$

We have the error bound

$$|\varepsilon_0| \leq 2^{e-2}. \tag{D.22}$$

We now use this to bound ε_1:

$$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq 2^{e-4.5}. \end{aligned} \tag{D.23}$$

Here we are using the fact that $\log_2(1/3) < -1.5$. Continuing, we find

$$\begin{aligned} \varepsilon_2 &\leq 2^{e-9} \\ \varepsilon_3 &\leq 2^{e-18} \\ &\vdots \end{aligned} \tag{D.24}$$

D.2.4 `BitLength`

We use the following approximation

$$x_0 = \begin{cases} \lfloor (27 \cdot 2^{e-1})/2^4 \rfloor & \textsc{BitLength}(x) = 0 \mod 2 \\ \lfloor (39 \cdot 2^{e-1})/2^5 \rfloor & \textsc{BitLength}(x) = 1 \mod 2 \end{cases}. \tag{D.25}$$

We assume x is an f-bit number:

$$2^{f-1} \leq x < 2^f, \quad f = 2e - k, \quad k \in \{0, 1\}. \tag{D.26}$$

- Case 0: $\textsc{BitLength}(x)$ is even or $k = 0$.

  In this case, we have

  $$2^{2e-1} \leq x < 2^{2e}, \tag{D.27}$$

  whence it follows that

  $$\sqrt{2} \cdot 2^{e-1} \leq \sqrt{x} < 2^e, \tag{D.28}$$

  Using the fact $45/32 < \sqrt{2}$, we have

  $$45 \cdot 2^{e-6} \leq \sqrt{x} < 64 \cdot 2^{e-6}. \tag{D.29}$$

  Because we are choosing

  $$x_0 \coloneqq 27 \cdot 2^{e-5}, \tag{D.30}$$

  we have

  $$|\varepsilon_0| \leq 5 \cdot 2^{e-5}. \tag{D.31}$$

  It follows that

  $$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq \frac{25}{27} 2^{e-6} \\ &\leq 2^{e-6}. \end{aligned} \tag{D.32}$$

- Case 1: $\textsc{BitLength}(x)$ is odd or $k = 1$.

  In this case, we have

$$2^{2e-2} \leq x < 2^{2e-1}, \tag{D.33}$$

whence it follows that

$$\cdot 2^{e-1} \leq \sqrt{x} < \sqrt{2} \cdot 2^{e-1}, \tag{D.34}$$

Using the fact $\sqrt{2} < 23/16$, we have

$$16 \cdot 2^{e-5} \leq \sqrt{x} < 23 \cdot 2^{e-5}. \tag{D.35}$$

Because we are choosing

$$x_0 \coloneqq 39 \cdot 2^{e-6}, \tag{D.36}$$

we have

$$|\varepsilon_0| \leq 7 \cdot 2^{e-6}. \tag{D.37}$$

It follows that

$$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq \frac{49}{39} 2^{e-7} \\ &\leq 2^{e-6.65}, \end{aligned} \tag{D.38}$$

where we are using the bound $\log_2(49/39) < 0.35$.

In the worst case, we have

$$\begin{aligned} \varepsilon_2 &\leq 2^{e-12} \\ \varepsilon_3 &\leq 2^{e-24} \\ &\vdots \end{aligned} \tag{D.39}$$

We note that it is possible to get better error bounds, but they are not worth the effort.

D.2.5 `Linear`

We start by scaling x (relabeled as m) so that

$$2^{2e-2} \leq m < 2^{2e}. \tag{D.40}$$

We then set

$$v \coloneqq \left\lfloor \frac{m}{2^{2e-4}} \right\rfloor \tag{D.41}$$

so that

$$v \cdot 2^{2e-4} \leq m < (v+1) \cdot 2^{2e-4}, \quad v \in \{4, 5, \ldots, 15\}. \tag{D.42}$$

This allows us to make the reduction to

$$\sqrt{v} \cdot 2^{e-2} \leq \sqrt{m} < \sqrt{v+1} \cdot 2^{e-2}. \tag{D.43}$$

In order to determine the appropriate approximation, we first find bounds for $\sqrt{v}$ and $\sqrt{v+1}$. In particular, we computed values $\nu_{v,0}, \nu_{v,1} \in \mathbb{N}$ with

$$\begin{aligned} \frac{\nu_{v,0}}{16} &\leq \sqrt{v} \\ \frac{\nu_{v,1}}{16} &\geq \sqrt{v+1} \end{aligned} \tag{D.44}$$

Using these values, we let

$$\mu_v \coloneqq 2 \cdot (14 + v). \tag{D.45}$$

The initial approximation is

$$x_0 = \mu_v \cdot 2^{e-6}. \tag{D.46}$$

We separate error calculations into two cases:

- Case 1: $v \in \{4, 5, 6, 7, 8\}$

  In this case, the initial error is

  $$\varepsilon_0 \leq 2^{e-4} \tag{D.47}$$

  and we have the error bounds

  $$\begin{aligned} \varepsilon_1 &\leq 2^{e-8} \\ \varepsilon_2 &\leq 2^{e-16} \\ &\vdots \end{aligned} \tag{D.48}$$

- Case 2: $v \in \{9, 10, 11, 12, 13, 14, 15\}$

  This case is more involved. We focus on the case $v = 9$. We have the bound

$$48 \cdot 2^{e-6} \leq \sqrt{m} < 51 \cdot 2^{e-6} \tag{D.49}$$

  with

$$\mu_9 = 46 \cdot 2^{e-6}. \tag{D.50}$$

  As we can see, $|\varepsilon_0| \leq 5 \cdot 2^{e-6}$. The error bound after the first Newton iteration is

$$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq \frac{25}{23} 2^{e-8} \\ &\leq 2^{e-7.875}. \end{aligned} \tag{D.51}$$

  Here, we are using the bound $\log_2(25/23) < 0.125$. The second iteration bound gives

$$\begin{aligned} \varepsilon_2 &= \frac{\varepsilon_1^2}{2x_1} \\ &\leq \frac{1}{24} 2^{e-11.75} \\ &\leq 2^{e-16.25}. \end{aligned} \tag{D.52}$$

  Here, we are using the bound of $x_1 \geq 48 \cdot 2^{e-6} = 24 \cdot 2^{e-5}$ from Eq. (D.49); we also use the bound $\log_2(1/24) < -4.5$. This bound for x_k continues to be used for the next error bounds:

$$\begin{aligned} \varepsilon_3 &\leq 2^{e-33} \\ \varepsilon_4 &\leq 2^{e-66.5} \\ \varepsilon_5 &\leq 2^{e-133.5} \\ &\vdots \end{aligned} \tag{D.53}$$

  The cases of $v \in \{10, 11, 12, 13, 14, 15\}$ are similar but we have $|\varepsilon_0| \leq 3 \cdot 2^{e-5}$; we do not include the details. The important fact is that, in all cases, $\varepsilon_5 \leq 2^{e-128}$. We note that the case $v = 10$ is shown in Appendix D.2.6.

D.2.6 `Hyper4`

Note that the error bound derivation here is essentially the same as that of `Linear` in Appendix D.2.5. The analysis is the same; the difference is the initialization value.

We start by scaling x (relabeled as m) so that

$$2^{2e-2} \leq m < 2^{2e}. \tag{D.54}$$

We then set

$$v \coloneqq \left\lfloor \frac{m}{2^{2e-4}} \right\rfloor \tag{D.55}$$

so that

$$v \cdot 2^{2e-4} \leq m < (v+1) \cdot 2^{2e-4}, \quad v \in \{4, 5, \ldots, 15\}. \tag{D.56}$$

This allows us to make the reduction to

$$\sqrt{v} \cdot 2^{e-2} \leq \sqrt{m} < \sqrt{v+1} \cdot 2^{e-2}. \tag{D.57}$$

In order to determine the appropriate approximation, we first find bounds for $\sqrt{v}$ and $\sqrt{v+1}$. In particular, we computed values $\nu_{v,0}, \nu_{v,1} \in \mathbb{N}$ with

$$\begin{aligned} \frac{\nu_{v,0}}{16} &\leq \sqrt{v} \\ \frac{\nu_{v,1}}{16} &\geq \sqrt{v+1} \end{aligned} \tag{D.58}$$

Using these values, we set $\mu_v \in \mathbb{N}$ with

$$\mu_v \coloneqq 2 \left\lfloor \frac{512}{31 - v} \right\rfloor. \tag{D.59}$$

The initial approximation is

$$x_0 = \mu_v \cdot 2^{e-6}. \tag{D.60}$$

We separate error calculations into two cases:

- Case 1: $v \in \{4, 5, 6, 7, 8, 13, 14, 15\}$

  In this case, the initial error is

  $$\varepsilon_0 \leq 2^{e-4} \tag{D.61}$$

  and we have the error bounds

  $$\begin{aligned} \varepsilon_1 &\leq 2^{e-8} \\ \varepsilon_2 &\leq 2^{e-16} \\ &\vdots \end{aligned} \tag{D.62}$$

- Case 2: $v \in \{9, 10, 11, 12\}$

  This case is more involved. We focus on the case $v = 10$ so as to not repeat the work from `Linear` in Appendix D.2.5. We have the bound

$$25 \cdot 2^{e-5} \leq \sqrt{m} < 27 \cdot 2^{e-5} \tag{D.63}$$

  with

$$\mu_{10} = 24 \cdot 2^{e-5}. \tag{D.64}$$

  We have absorbed the common factor of 2 present in the previous two equations. As we can see, $|\varepsilon_0| \leq 3 \cdot 2^{e-5}$. The error bound after the first Newton iteration is

$$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq \frac{9}{24} 2^{e-6} \\ &\leq 2^{e-7.4}. \end{aligned} \tag{D.65}$$

  Here, we are using the bound $\log_2(9/24) < -1.4$. The second iteration bound gives

$$\begin{aligned} \varepsilon_2 &= \frac{\varepsilon_1^2}{2x_1} \\ &\leq \frac{1}{25} 2^{e-10.8} \\ &\leq 2^{e-15.44}. \end{aligned} \tag{D.66}$$

  Here, we are using the bound of $x_1 \geq 25 \cdot 2^{e-5}$ from Eq. (D.63); we also use the bound $\log_2(1/25) < -4.64$. This bound for x_k continues to be used for the next error bounds:

$$\begin{aligned} \varepsilon_3 &\leq 2^{e-31.52} \\ \varepsilon_4 &\leq 2^{e-63.68} \\ \varepsilon_5 &\leq 2^{e-128} \\ &\vdots \end{aligned} \tag{D.67}$$

  The case of $v = 9$ was shown in Appendix D.2.5 and the case is similar for $v \in \{11, 12\}$ as $|\varepsilon_0| \leq 3 \cdot 2^{e-5}$; we do not include the details. The important fact is that, in all cases, $\varepsilon_5 \leq 2^{e-128}$.

D.2.7 `Lookup4`

We start by scaling x (relabeled as m) so that

$$2^{2e-2} \leq m < 2^{2e}. \tag{D.68}$$

We then set

$$v \coloneqq \left\lfloor \frac{m}{2^{2e-4}} \right\rfloor \tag{D.69}$$

so that

$$v \cdot 2^{2e-4} \leq m < (v+1) \cdot 2^{2e-4}, \quad v \in \{4, 5, \ldots, 15\}. \tag{D.70}$$

This allows us to make the reduction to

$$\sqrt{v} \cdot 2^{e-2} \leq \sqrt{m} < \sqrt{v+1} \cdot 2^{e-2}. \tag{D.71}$$

In order to determine the appropriate approximation, we first find bounds for $\sqrt{v}$ and $\sqrt{v+1}$. In particular, we computed values $\nu_{v,0}, \nu_{v,1} \in \mathbb{N}$ with

$$\begin{aligned} \frac{\nu_{v,0}}{8} &\leq \sqrt{v} \\ \frac{\nu_{v,1}}{8} &\geq \sqrt{v+1}. \end{aligned} \tag{D.72}$$

Combining Eqs. (D.71) and (D.72), we have

$$\nu_{i,0} \cdot 2^{e-5} \leq \sqrt{m} < \nu_{i,1} \cdot 2^{e-5}. \tag{D.73}$$

We note that

$$\max_v |\nu_{v,1} - \nu_{v,0}| = 3. \tag{D.74}$$

This allows values $\mu_v \in \mathbb{N}$ to be chosen so that

$$|\nu_{v,i} - \mu_v| \leq 2. \tag{D.75}$$

The initial approximation is

$$x_0 = \mu_v \cdot 2^{e-5} \tag{D.76}$$

and the initial error is

$$\varepsilon_0 \leq 2^{e-4}. \tag{D.77}$$

This gives the error bounds

$$\begin{aligned} \varepsilon_1 &\leq 2^{e-8} \\ \varepsilon_2 &\leq 2^{e-16} \\ &\vdots \end{aligned} \tag{D.78}$$

D.2.8 `Lookup8`

We start by scaling x (relabeled as m) so that

$$2^{2e-2} \leq m < 2^{2e}. \tag{D.79}$$

We then set

$$v \coloneqq \left\lfloor \frac{m}{2^{2e-8}} \right\rfloor \tag{D.80}$$

so that

$$v \cdot 2^{2e-8} \leq m < (v+1) \cdot 2^{2e-8}, \quad v \in \{64, 65, \ldots, 255\}. \tag{D.81}$$

This allows us to make the reduction to

$$\sqrt{v} \cdot 2^{e-4} \leq \sqrt{m} < \sqrt{v+1} \cdot 2^{e-4}. \tag{D.82}$$

In order to determine the appropriate approximation, we first find bounds for $\sqrt{v}$ and $\sqrt{v+1}$. In particular, we computed values $\nu_{v,0}, \nu_{v,1} \in \mathbb{N}$ with

$$\begin{aligned} \frac{\nu_{v,0}}{32} &\leq \sqrt{v} \\ \frac{\nu_{v,1}}{32} &\geq \sqrt{v+1}. \end{aligned} \tag{D.83}$$

Combining Eqs. (D.82) and (D.83), we have

$$\nu_{i,0} \cdot 2^{e-9} \leq \sqrt{m} < \nu_{i,1} \cdot 2^{e-9}. \tag{D.84}$$

We note that

$$\max_v |\nu_{v,1} - \nu_{v,0}| = 3. \tag{D.85}$$

This allows values $\mu_v \in \mathbb{N}$ to be chosen so that

$$|\nu_{v,i} - \mu_v| \leq 2. \tag{D.86}$$

The initial approximation is

$$x_0 = \mu_v \cdot 2^{e-9} \tag{D.87}$$

and the initial error is

$$\varepsilon_0 \leq 2^{e-8}. \tag{D.88}$$

This gives the error bounds

$$\begin{aligned} \varepsilon_1 &\leq 2^{e-16} \\ \varepsilon_2 &\leq 2^{e-32} \\ &\vdots \end{aligned} \tag{D.89}$$

D.3 Final Check

We now discuss the final check performed in the new algorithms and discuss a potential issue with checks performed in other algorithms found online.

Possible Values at the end of Newton Iterations In the case of algorithms performing unrolled Newton iterations, all algorithms perform iterations until $\varepsilon_k \leq 1$. This ensures that $r \in \{q, q+1\}$ with respect to error bounds. This is also ensured even if the iteration oscillates. Thus, we must ensure that we return the correct value.

Confirm if Integer Square Root Given that we have $r \in \{q, q+1\}$, we can check if $r^2 \leq n$: if this is true, then $r = q$; otherwise, $r = q + 1$. The return logic is not too expensive, although care must be taken to ensure that no overflow occurs. Thus, in the newly developed algorithms, the case $n \geq (2^{128} - 1)^2$ is handled separately with an `if` statement at the beginning.

Return Minimum We note that in some of the algorithms found online (`PRB` (Listing 2) [1], `OpenZeppelin` (Listing 3) [14], and `ABDK` (Listing 4) [4]) the final check is of the form

$$\textbf{return} \quad \min(r, n/r). \tag{D.90}$$

This is done after applying 7 Newton iterations.

It is not clear that this logic ensures the correct value is returned. These algorithms have the same initialization as in `Unrolled1` and perform 7 Newton iterations, so we know $r \in \{q, q+1\}$.

Suppose we have

$$n = q^2 + \ell, \quad \ell \in \{0, \ldots, q-1\}. \tag{D.91}$$

Then we see

$$n = (q-1)(q+1) + (1+\ell) \tag{D.92}$$

so that

$$\left\lfloor \frac{n}{q+1} \right\rfloor = q - 1. \tag{D.93}$$

This shows that if n is as specified and we have $r = q + 1$ after the final Newton iteration (which is possible with the current analysis), then $\min(r, n/r) = n/r = q - 1$, which is incorrect. Thus, the current analysis will not ensure that this return logic will always give the correct result.

We note that although we have shown that it is *possible* for `PRB`, `OpenZeppelin`, and `ABDK` to return incorrect results (that is, results which are off by 1), we must mention that there is no known value where this error occurs. That is to say, the current analysis is *unable to prove the algorithms are correct*, but that is not the same as *proving they are incorrect*.

With that said, these algorithms operate on `uint256` values and approximately half of all integers are of the form described in Eq. (D.91), so it is not possible to manually check each possible value to ensure a valid result is returned. Thus, if `PRB`, `OpenZeppelin`, and `ABDK` are to be proven correct, additional analysis is required.

D.4 Convergence Proofs

We now combine the results from the previous sections. They are straightforward at this point.

D.4.1 Proof of General Newton Iteration

We start by proving Algorithm 1 (from [3, Algorithm 1.7.1]) produces the correct result. This is included for completeness, and the proof provided here is essentially that of [3, Algorithm 1.7.1, Proof].

First, x decreases at every iteration. Because n is finite, the algorithm must eventually terminate. We set $q = \lfloor \sqrt{n} \rfloor$. From the discussion above, we always have $(t + n/t)/2 \geq \sqrt{n}$ for all $t > 0$, so we always have $x \geq q$. We see

$$y - x = \left\lfloor \frac{n - x^2}{2x} \right\rfloor . \tag{D.94}$$

It follows that $x > \sqrt{n}$ iff $n - x^2 < 0$ iff $y < x$. We also have $x \leq \sqrt{n}$ iff $n - x^2 \geq 0$ iff $y \geq x$. Thus, $y \geq x$ implies that $x \leq \sqrt{n}$. Because $x \in \mathbb{N}$ and $q \leq x \leq \sqrt{n}$, it follows that $x = q$. Thus, the algorithm returns the correct result.

Note: the only place we used the initialization value is to ensure that we initially have $x \geq \lfloor \sqrt{n} \rfloor$. Thus, provided the initialization value always satisfies this, this proof may be applied.

D.4.2 Proof of Newton Iteration Fixed Points

We now prove that for $q = \lfloor \sqrt{n} \rfloor$ we have q is a fixed point of g_n (that is, $g_n(q) = q$) iff $n+1$ is not a square.

We let $y = g_n(x)$ as in Appendix D.4.1. The work there shows that $x > \sqrt{n}$ implies that $y < x$; additionally, for $x < q$ we have $y > x$. This holds for all n. So, we only need to investigate the case $x = q$.

From the definition of $q = \lfloor \sqrt{n} \rfloor$ we have

$$q^2 \leq n < (q+1)^2 . \tag{D.95}$$

This implies

$$q^2 \leq n \leq (q+2)\, q \tag{D.96}$$

so that

$$\left\lfloor \frac{n}{q} \right\rfloor \in \{q, q+1, q+2\} . \tag{D.97}$$

It follows that

$$g_n(q) = \begin{cases} q & \lfloor n/q \rfloor = q \\ q & \lfloor n/q \rfloor = q+1 \\ q+1 & \lfloor n/q \rfloor = q+2 \end{cases}. \tag{D.98}$$

We note that when $n = a^2 - 1$ (that is, when $n+1$ is a square) q is not a fixed point for g_n. We note that the result will oscillate as $q, q+1, q, q+1, \ldots$.

D.4.3 `Unrolled1`

From the error bounds in Appendix D.2.1, we see that $\varepsilon_7 \leq 1$ (we require 7 Newton iterations). Thus, $r \in \{q, q+1\}$, and the return logic discussed in Appendix D.3 ensures the correct result.

D.4.4 `Unrolled2`

From the error bounds in Appendix D.2.2, we see that $\varepsilon_7 \leq 1$ (we require 7 Newton iterations). Thus, $r \in \{q, q+1\}$, and the return logic discussed in Appendix D.3 ensures the correct result.

D.4.5 `Unrolled3`

From the error bounds in Appendix D.2.3, we see that $\varepsilon_6 \leq 1$ (we require 6 Newton iterations). Thus, $r \in \{q, q+1\}$, and the return logic discussed in Appendix D.3 ensures the correct result.

D.4.6 `While1`

After the first Newton iteration, we have $r \geq q$; the proof in Appendix D.4.1 then applies. The first Newton iteration *is required* for this proof to be valid.

D.4.7 `While2`

The proof in Appendix D.4.1 applies because $r > \sqrt{n}$.

D.4.8 `While3`

After the first Newton iteration, we have $r \geq q$; the proof in Appendix D.4.1 then applies. The first Newton iteration *is required* for this proof to be valid.

D.4.9 `BitLength`

We let $k \in \mathbb{N}$ and suppose we have

$$2^{2k-2} \leq x < 2^{2k} \tag{D.99}$$

After the `if (result >= (1 << 2))` statement, we will have $e = 2k - 1$.

If the statement `if (result >= (1 << 1))` is True, then this means that we actually have the bitlength is equal to $2k$. We have

$$r = 27 \cdot 2^{k-5}. \tag{D.100}$$

Otherwise, `if (result >= (1 << 1))` is False and we have the bitlength is equal to $2k-1$, We have

$$r = 39 \cdot 2^{k-6}. \tag{D.101}$$

From the error bounds in Appendix D.2.4, we see that $\varepsilon_6 \leq 1$ (we require 6 Newton iterations). Thus, $r \in \{q, q+1\}$, and the return logic discussed in Appendix D.3 ensures the correct result.

D.4.10 `Linear`

Initial Scaling We show that we properly scale x so that $2^{254} \leq m < 2^{256}$.

We let $k \in \mathbb{N}$ such that

$$2^{2k-2} \leq x < 2^{2k}. \tag{D.102}$$

In this case, we see that the "bitlength" b is

$$b \coloneqq 2k - 1. \tag{D.103}$$

The "bitlength" b is the scaling factor e before it is overwritten. From here, we compute the scaling factor

$$e \coloneqq 128 - k. \tag{D.104}$$

We define

$$m \coloneqq x \cdot 2^{2e}. \tag{D.105}$$

From Eq. (D.102), we see

$$\begin{aligned} m &= 2^{2e}x \\ &\geq 2^{256-2k} \cdot 2^{2k-2} \\ &= 2^{254} \end{aligned} \tag{D.106}$$

and

$$\begin{aligned} m &= 2^{2e}x \\ &\leq 2^{256-2k} \cdot 2^{2k} \\ &= 2^{256}. \end{aligned} \tag{D.107}$$

Together, these imply

$$2^{254} \leq m < 2^{256}, \tag{D.108}$$

as desired.

Error Bounds From the error bounds in Appendix D.2.5, we see that $\varepsilon_5 \leq 1$ (we require 5 Newton iterations). Thus, $r \in \{q, q+1\}$ for $q = \lfloor \sqrt{m} \rfloor$.

Final Scaling and Check At this point, we have $r \in \{q, q+1\}$ for $q = \lfloor \sqrt{m} \rfloor$. This means

$$(r-1)^2 \leq m < (r+1)^2 . \tag{D.109}$$

We set

$$r = s2^e + t, \quad t \in \left\{0, 1, \cdots, 2^{e-1}\right\}. \tag{D.110}$$

This implies that

$$s = \left\lfloor \frac{r}{2^e} \right\rfloor . \tag{D.111}$$

We see

$$\begin{aligned} (r-1)^2 &= (s2^e + t - 1)^2 \geq 2^{2e} (s-1)^2 \\ (r+1)^2 &= (s2^e + t + 1)^2 \leq 2^{2e} (s+1)^2 . \end{aligned} \tag{D.112}$$

After substituting $m = 2^{2e}x$ and using the previous inequalities, Eq. (D.109) reduces to

$$2^{2e} (s-1)^2 \leq 2^{2e} x < 2^{2e} (s+1)^2 \tag{D.113}$$

or

$$(s-1)^2 \leq x < (s+1)^2 . \tag{D.114}$$

Thus, $s \in \{z, z+1\}$ for $z = \lfloor \sqrt{x} \rfloor$. The final check ensures we return the correct result. There is no overflow because we take care of the edge case when $\lfloor \sqrt{x} \rfloor = 2^{128} - 1$; this ensures that $s \leq 2^{128} - 1$, so $s^2 < 2^{256}$ and no overflow occurs.

D.4.11 `Hyper4`

Initial Scaling See the discussion in Appendix D.4.10.

Error Bounds From the error bounds in Appendix D.2.6, we see that $\varepsilon_5 \leq 1$ (we require 5 Newton iterations). Thus, $r \in \{q, q+1\}$ for $q = \lfloor \sqrt{m} \rfloor$.

Final Scaling and Check See the discussion in Appendix D.4.10. This shows the correct result is returned.

D.4.12 `Lookup4`

Initial Scaling See the discussion in Appendix D.4.10.

Error Bounds From the error bounds in Appendix D.2.7, we see that $\varepsilon_5 \leq 1$ (we require 5 Newton iterations). Thus, $r \in \{q, q+1\}$ for $q = \lfloor \sqrt{m} \rfloor$.

Final Scaling and Check See the discussion in Appendix D.4.10. This shows the correct result is returned.

D.4.13 `Lookup8`

Initial Scaling See the discussion in Appendix D.4.10.

Error Bounds From the error bounds in Appendix D.2.8, we see that $\varepsilon_4 \leq 1$ (we require 4 Newton iterations). Thus, $r \in \{q, q+1\}$ for $q = \lfloor \sqrt{m} \rfloor$.

Final Scaling and Check See the discussion in Appendix D.4.10. This shows the correct result is returned.