# Improvement on the Precision of QR Factorization

James Guo, Tina Shen, and Anna Dai

Johns Hopkins University

March 30, 2025

**Supervisor:** Dr. Mario Micheli

## Table of Contents

We will start from the basic background and motivations, and gradually move into the motivation of our project and its analyses.

## Preliminaries

- QR Factorization
- Machine Representation
- Issues with Machine Representation
    - Catastrophic Cancellation
- Current Work

## QR Factorization

QR factorization is an important algorithm in computational mathematics.

**QR Factorization**

Let $A \in \mathbb{R}^{n \times n}$ be a matrix, its QR factorization factors $A$ into:

$$A = QR,$$

where $Q \in \mathbb{R}^{n \times n}$ is orthogonal $\left(\textit{i.e.}, \ Q^\intercal Q = I\right)$ and $R \in \mathbb{R}^{n \times n}$ is upper triangular.

- Note that $A$ does not need to square matrix, but having it as a square matrix could be helpful to present various applications.
- Also note that the field does not have to be $\mathbb{R}$, it can be any field, such as $\mathbb{C}$.

## Machine Representation

Theoretically, the QR factorization is precise when computed by hand, but this becomes a problem when the computation is on a *computer*.

- It is notable that $\mathbb{R}$ is *uncountable*, and the machine can at most represent a *countable* set of numbers, so the real numbers **cannot** be fully represented by machine.
- The current machine standard uses the IEEE standards.

### IEEE Representation Standard

For single precision, there are 32 bits (or 4 bytes).

| | | |
|---|---|---|
| 1 sign bit | #e 8 exponent bits | #f 23 mantissa bits |

For double precision, there are 64 bits (or 8 bytes).

| | | |
|---|---|---|
| 1 sign bit | #e 11 exponent bits | #f 52 mantissa bits |

## Machine Representation (Continued)

For the sake of the presentation, we consider the double precision.

| | | |
|---|---|---|
| 1 sign bit | #e 11 exponent bits | #f 52 mantissa bits |

The sign bit is intuitive, either positive (technically *nonnegative*) or negative, and the exponential represents the power of 2 subtracted by 127, and the mantissa were the numbers, represented by:

$$\pm(\#\texttt{f})^{(\#\texttt{e})-127}.$$

### Example of Single Precision Float

Consider the number stored as:

| 1 sign bit | #e 11 exponent bits | #f 52 mantissa bits |
|---|---|---|
| $+$ | 00000111111 | $000\cdots0001$ |

This number has the exponent as 127 and the mantissa as 1, so it is:

$$+1\times 2^{127-127}=1.$$

## Issues with Machine Representation

Note that with this representation, the smallest positive representable number is:

| 1 sign bit | #e 11 exponent bits | #f 52 mantissa bits |
|---|---|---|
| $+$ | 00000000000 | $000\cdots0001$ |

This number is then:

$$+1 \times 2^{-127} = 2^{-127}.$$

Hence the machine will be unable to present a number like $2^{-128}$ precisely.

- Note that there is also a upper bound with the machine representable number of floating point, but we will not discuss that yet.

## Issues with Machine Representation (Continued)

In the realm of computational mathematics, there is a definition on machine representable numbers (MRN) as the numbers that can be expressed as above, and there is a consequence as the machine epsilon.

**Machine Epsilon**

The machine epsilon ($\epsilon_{\text{machine}}$) is defined as the distance between 1 and the next larger MRN.

For our single precision floating point number, the next larger number than 1 is:

| 1 sign bit | #e 11 exponent bits | #f 52 mantissa bits |
|---|---|---|
| $+$ | 00001001100 | $100 \cdots 0001$ |

Hence, we have:

$$\epsilon_{\text{machine}} = 2^{-52}.$$

## Catastrophic Cancellation

The machine representation could cause problems, and the most major one of them is the **catastrophic cancellation**.

### Catastrophic Cancellation

**Catastrophic cancellation** occurs when subtraction between two close, large numbers result in precision loss for the differences.

To better illustrate, we give an example in `python`.

```
» dev = 0.12
» x = 2 ** 50 + 0.0
» y = 2 ** 50 + dev
» y - x
0.0
```

This is especially an issue when we attempt to divide something by the difference $y - x$ later on.

## Gram Schmidt Algorithm and Modified Gram Schmidt

The typical algorithm for the QR factorization is the Gram Schmidt algorithm, decomposing $V$ into $Q \cdot R$.

**Classical Gram-Schmidt Algorithm**

```
- for $j \leftarrow 1, \cdots, n$:
    - Let $v_j \leftarrow a_j$.
    - for $i \leftarrow 1, \cdots, j-1$:
        - $r_{i,j} \leftarrow q_i^\intercal a_j$.
        - $v_j \leftarrow v_j - r_{i,j} q_i$
    - $r_{j,j} \leftarrow \|v_j\|_2$
    - $q_j \leftarrow v_j / r_{j,j}$
```

The smallest representable result is $\sqrt{\epsilon_{\text{machine}}}$.

**Modified Gram-Schmidt Algorithm**

```
- for $i \leftarrow 1, \cdots, n$:
    - Let $v_i \leftarrow a_i$.
- for $i \leftarrow 1, \cdots, n$:
    - $r_{i,i} \leftarrow \|v_i\|_2$.
    - $q_i \leftarrow v_i / r_{i,i}$
    - for $j \leftarrow i+1, \cdots, n$:
        - $r_{i,j} \leftarrow q_i^\intercal v_j$.
        - $v_j \leftarrow v_j - r_{i,j} q_i$
```

The smallest representable result is $\epsilon_{\text{machine}}$.

## Applications Requiring Exact or High-Precision QR Factorization

- **Computational Geometry**: requires consistent decisions about incidence and intersection, such as in determining linear independence of vectors without floating-point error.
- **Control Theory**: exact rational QR eliminates false unstable cases or missed unstable poles due to numerical rounding (in filter design, for example).
- **Optimization**: solvers for linear/quadratic programs and certain iterative methods benefit from rational QR to correctly identify constraint activations or dependencies.
- **Scientific Computing**: computational biology or astronomy needs to solve a regression or alignment problem to very high precision.

## Improvement on QR Factorization

- Fractional Representation
- Implementation Choices
  - Newton's Method
- Evaluation on Implementation
  - Results on Implementation

## Fractional Representation

The motivation of our project is to use a fractional representation for matrix operations, noting some **properties** of rational numbers:

- $\mathbb{Q}$ is *countable*, so it is (theoretically) representable.
- $\mathbb{Q}$ is **dense** in $\mathbb{R}$, and technically the $\mathbb{Q}[\mathrm{i}]$ lattice is **dense** in $\mathbb{C}$.
    - We can ensure entry-wise completeness for matrix multiplications.

### Remark on `Python` Language

In the `python` language, the programming language has not casted an explicit upper bound for the largest integer.

- We utilize this property to represent all numbers in terms of rationals.

## Fractional Representation (Continued)

We can think any two number being represented (or approximated) as $\alpha = \frac{a}{b}$ and $\beta = \frac{c}{d}$, where $a, c \in \mathbb{Z}$ and $b, d \in \mathbb{Z}^+$, and the basic operations would be similar of how it was done in a fractional field.

- Addition and subtraction:
$$\alpha \pm \beta = \frac{a \times d \pm b \times c}{b \times d}.$$
- Multiplication:
$$\alpha \times \beta = \frac{a \times b}{c \times d}.$$
- Division, when $\beta \neq 0$:
$$\alpha \div \beta = \frac{a \times d}{b \times c}.$$

If we assume addition and multiplication of constant time, the time complexity is asymptotically the same.

## Fractional Representation (Continued)

### Fraction Class in Python

Luckily python library has a class of Fraction, so there are also functions such as taking integer powers or simplify function to automatically simplify the fractions through the implementation process.

The main idea for this project is to implement the **Gram-Schmidt** algorithm on QR factorization using the Fraction class.

### Issues with Pure Fraction Class Conversion

The implementation may seems trivial, but there are some issues with the Fraction class:

- $\mathbb{Q}$ is **not** closed under square root.
- The addition and multiplication of large integers is **not** constant time.

## Newton's Method

$\mathbb{Q}$ is **not** closed under square root operator.

- Technically, we might could extend the field each time with a new variable for the square roots, quadruple roots, $\cdots$, such as:
$$\mathbb{Q}[\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{7}, \cdots][\sqrt[4]{2}, \cdot] \cdots .$$
However, this will be very complicated: Each extension (of $2^k$-th root of prime) could be thought of as a variable (*Galois Theory*), and we also need to handle the interactions of the terms.

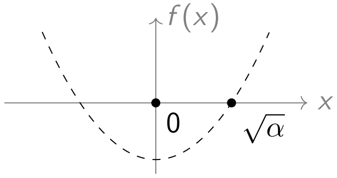
The method we decided to go around is to use **Newton's method**, in which we can turn the square root operator as an equivalent problem.

### Equivalence of Square Root

Let $\alpha \in \mathbb{Q}$ be arbitrary, finding the square root of $\alpha$ is equivalent to finding the positive root of the function $f(x) = x^2 - \alpha$.

## Newton's Method (Continued)

Even better, consider that the function is convex, and as long as we start as a positive value, we will be able to get towards the actual value of $\sqrt{\alpha}$:



- Initialize $x = \alpha$.
- **while** within the maximum iteration:
    - Compute step as $\mathrm{step} = (x^2 - \alpha) \div (2 \times x)$.
    - Update $x$ by subtracting the $\mathrm{step}$, and if the $\mathrm{step}$ is less than the desired precision, break the loop.

Here, we enforced an desired precision for Newton's method so it will terminate when the result is *good enough*.

## Precision and Trade-off

Another thing we need to care about is how complex the algorithm is.

- An assumption of most complexity analysis is that all operations $+$ or $\times$ are of constant time.

However, this is not true.

**Example of Calculation Complexity**

Consider the following two examples of multiplications:

$$12 \times 11 =?$$

$$2,147,483,648 \times 4,398,046,511,104 =?$$

It should be clear that the second multiplication should be slower in general when the integers are too big.

## Precision and Trade-off (Continued)

It is **not** a valid assumption to treat multiplication and addition as the same time when our integers are very large.

- We will analyze the complexity in detail when we analyze the complexity of the algorithm, but here, we can tell that the precision is together with a trade-off with time complexity.

To prevent the algorithm being too complicated, we decided to retrieve our desired precision (as dyadic number) in **Newton's Method**, and use that to prevent explosion of numerators and denominators.

```
• Given a fraction $\frac{p}{q}$.
• while $q >$the denominator of the dyadic desired precision:
    • if $p = \pm 1$, end the simplify procedure.
    • Divide $p$ and $q$ both by 2 (truncate a bit).
```

## Testing Strategy

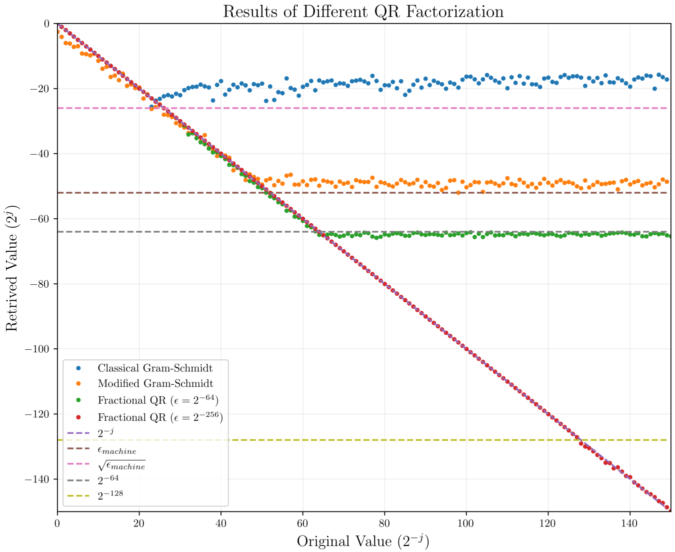
For our improved QR Factorization using Fractions, we have initialized diagonal matrices with dyadic entries, compose it with a orthogonal matrix, and attempt to use QR factorization to retrieve the original dyadic number.

### Experiment Procedure

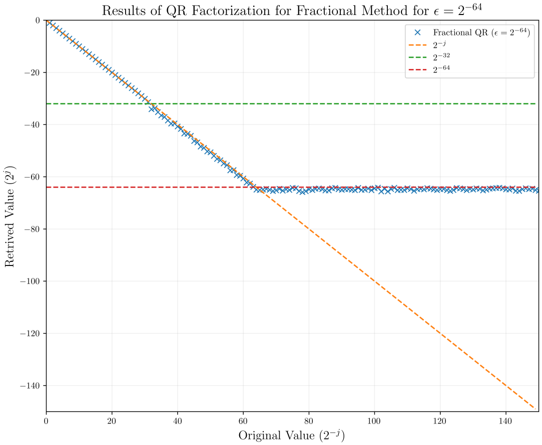
What we aimed to compare are the following models:

- Classical Gram Schmidt algorithm.
- Improved Gram Schmidt algorithm.
- Fractional QR algorithm with $\epsilon = 2^{-64}$.
- Fractional QR algorithm with $\epsilon = 2^{-256}$.

# Results on Implementation

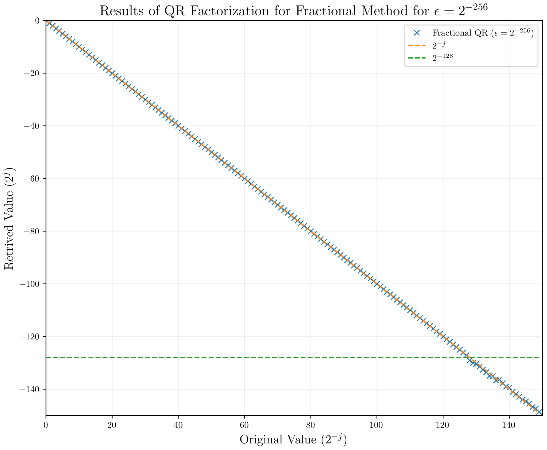


# Detailed Results on Fractional QR with $\epsilon = 2^{-64}$



- High precision when within $2^{-32}$, deviations between $2^{-32}$ and $2^{-64}$, and intractable error after $2^{-64}$.

## Detailed Results on Fractional QR with $\epsilon = 2^{-256}$



- High precision when within $2^{-128}$, deviations after $2^{-128}$.

## Analysis on Fractional Improvement

- Time Complexity
  - Remark on Newton's Method
- Precision
- Extensions

## Worst Time Complexity on Integer Operations

Here, we think about addition and multiplication between two integers of $k$ bits.

- It can be proven that the best runtime of adding/subtraction two integers of $k$ bits will be $\Theta(k)$.
- For the multiplication algorithm, by using the Karatsuba trick, the multiplication of two integers of $k$ bits will be $\Theta(k^{\log_2(3)})$.

### Operation time in Double Precision Float

If we assume the operations in double precision float, $k = 52$:

- Each addition is about 52 times of single bit addition.
- Each multiplication is about 525 times of single bit addition.

## Worst Time Complexity on Integer Operations (Continuous)

Then, we think about our Fraction QR implementation with $\epsilon = 2^{-256}$.

- Each addition is about 256 times of single bit addition.
- Each multiplication is about 6561 times of single bit addition.

Then we need to think about the fractional field operation:

**Operation time in Fractional Field**

- Each addition is composed of three integer multiplications and one integer addition, so it is about 19939 multiples of single bit addition.
- Each multiplication is composed of two multiplications, so it is about 13122 multiples of single bit addition.

Hence, each addition and multiplication is about 25 to 383 times slower than the floating point operation.

## Worst Time Complexity on Integer Operations (Continuous)

Even if we have $\epsilon = 2^{-52}$, which is exactly the machine epsilon, the multiplication time of the fraction class is about 2 times slower, and the addition time of the fraction class is about 31 times slower.

**The Operation could be more Complicated**

Note that the previous arguments are just rough estimates, since:

- There are `simplify` for the `Fraction` class that could be making the fractions into more simplified form.
- At the same time, when simplified, the computation of integer addition and multiplication would not reach the worst case.
- There are truncations of digits through the algorithm, but that runtime should be relatively negligible (as bitwise truncation is fast).

When considering the runtime not as asymptotically runtime, the algorithm will be significantly slower.

## Remark on Newton's Method

Note that the Newton's method is also used for taking the square root operator, so the major difference will still be the same multiple for the number of operations.

The computation of square root for floating point also uses the Newton's method, so the algorithm would be similar for the Fractional QR, with the same constant multiple slower.

As a side note, for our Newton's method on taking the square roots for $\epsilon = 2^{-256}$, the algorithm converges to the desired result for taking the square root of $2^{-200}$ takes about 100 iterations, so the Newton's method is relatively efficient.

## Precision in Operation

However, the Fractional implementation allows some more flexibility in precision:

- We can manually adjust for the degree of preciseness that we want.
- Recall the graph, we can achieve a very good result when the result is larger than $\sqrt{\epsilon}$ and a reasonable result with some deviation when the result is larger than $\epsilon$.
    - This is because the Fractional representation can represent more smaller numbers and preserve the precision of the smaller part of the large numbers, so it effectively avoids the embarrassment of Catastrophic cancellation.

The trade-off is the slower computational time, for higher precision and flexibility of how precise we want.

## Extension of Fractional Algorithms

The `Fractional` adaption of the algorithms is not only limited to QR factorization, and we can apply it to a wider class of algorithms.

- When we need precision, for some computations, there is no room for deviation.
- Especially when all the operations are addition, subtraction, multiplication, and division. This is when all the operations are closed in $\mathbb{Q}$ or $\mathbb{Q}[\mathrm{i}]$.

**Usage**

The Fractional algorithm is most valuable when certain level of precision is required. When precision is not that important, or when we want some randomness generated by noises, the floating-point operations still have its own advantages.

## End of the Talk

We are open to some questions about this project.

Thank you for listening.