DEGREE PROJECT IN COMPUTER ENGINEERING,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2021

A Comparative Study on the Accuracy of IEEE-754 and Posit for N-body Simulations

CHRISTOFER NOLANDER

AMANDA STRÖMDAHL

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

A Comparative Study on the Accuracy of IEEE-754 and Posit for N-body Simulations

CHRISTOFER NOLANDER

AMANDA STRÖMDAHL

Degree project, first cycle (15 hp)
Date: June 24, 2021
Supervisor: Stefano Markidis
Examiner: Pawel Herman
School of Electrical Engineering and Computer Science
Swedish title: En jämförelse mellan noggrannheten hos IEEE-754 och Posit för N-kroppssimuleringar

Abstract

IEEE-754 has been the standard floating-point format for several decades. However, in recent years multiple new floating-point formats have emerged, one of which is Posit. This study aimed to examine the accuracy of 32-bit IEEE-754 compared to that of 32-bit Posit. Cosmological N-body simulations were chosen as the benchmark due to physics being a field with a need for high-accuracy calculations. Additionally, these calculations tend to challenge floating-point formats by including values of vastly different magnitudes.

Two different methods were used to determine the accuracy of the floating-point formats. They both involved running N-body simulations with different parameters and floating-point formats. In the first method, accuracy was measured by comparing against a high precision 1024-bit floating-point format. For the second method, simulations were first run forward for some number of iterations and then in reverse for that same number of iterations. The accuracy was then determined by comparing the initial state of the simulation to its final state.

The results from both methods implied that 32-bit Posit is slightly more accurate than 32-bit IEEE-754 when the values are approximately within the range $[10^{-3},\ 10^{3}]$. More specifically, it was shown that Posit has the highest accuracy when values are near 1. For all other values, IEEE-754 outperformed Posit, which suggests that IEEE-754 might be a safer choice when the magnitude of the values is unpredictable. Though, when values are known to be close to 1, Posit has clear benefits.

Sammanfattning

IEEE-754 har varit standarden för flyttalsformat i flera årtionden, men under de senaste åren har flera andra format uppkommit, varav ett är Posit. Denna studie syftade till att jämföra noggrannheten hos 32-bitars IEEE-754 med den hos 32-bitars Posit. För åstadkomma detta valdes kosmologiska *N-kroppssimuleringar* som tillämpningsområde. Detta på grund av att fysikaliska simuleringar ofta kräver hög noggranhet och dessutom tenderar att ställa höga krav på flyttalsformat eftersom beräkningarna vanligtvis innefattar både extremt stora och extremt små tal.

För att betsämma nogrannheten hos de två formaten användes två olika metoder som båda gick ut på att köra N-kroppssimuleringar med olika parametrar och flyttalsformat. I den första metoden mättes noggranhet genom att jämföra simuleringar med IEEE-754 eller Posit mot en simulering med ett 1024-bitars flyttalsformat som hade väldigt hög precision. Den andra metoden gick ut på att först köra simuleringen framlänges ett visst antal steg, och sedan beräkna samma antal steg baklänges. Noggrannheten kunde sedan bestämmas genom att jämföra simuleringens startvärden med slutvärdena, som inte var samma på grund av avrundningsfel.

Baserat på resultaten från de två metoderna är 32-bitars Posit aningen mer noggrann än 32-bitars IEEE-754 när värdena ungefär befinner sig inom intervallet $[10^{-3},\ 10^{3}]$. Mer specifikt visade resultaten att Posit är som mest noggrann när värdena är nära 1. För alla andra värden presterade IEEE-754 bättre än Posit, vilket antyder att IEEE-754 är ett säkrare val när storleken på värdena är oförutsägbar. Däremot har Posit tydliga fördelar när det är känt att värdena är nära 1.

Contents

Chapter 1

Introduction

Floating-point formats are the most widely used tool to approximate and work with real numbers in computers today. Out of the available formats, the one with the highest adoption and most hardware support is IEEE-754, as of this writing. IEEE-754 is a floating-point format from 1985, developed by the *Institute of Electrical and Electronics Engineering* (IEEE) (Vinayan and Prof R. Nage 2015). However, in recent years, several new floating-point formats have emerged. John Gustafson developed one such format in 2017, which was named Posit (Gustafson and Yonemoto 2017). This new format makes different trade-offs to IEEE-754 regarding numerical accuracy and memory usage, potentially making it more suitable under certain circumstances.

One area where the choice of floating-point format may be of particular importance is simulation software in physics. This area require both high precision and the ability to work with values of many different magnitudes. Using a format that excels at this use case is thus highly valuable. Cosmological N-body simulations are an example of such an application: these attempt to model the gravitational forces between astral bodies.

Curently, however, no research has gone into the advantages of each floating-point format when applied to N-body simulations. A comparison between IEEE-754 and Posit could give valuable insight into which format is most suitable for this use case and other simulation applications.

1.1 Aim

This study aims to explore the advantages and disadvantages of Posit compared to IEEE-754. To that end, this thesis will answer the following question:

- How do the accuracies of 32-bit Posit and 32-bit IEEE-754 compare when used for N-body simulations?

1.2 Scope

In this thesis, the focus is on the differences in numerical accuracy between the two floating-point formats. Developing a physically accurate simulation is not a goal of this thesis. However, simulations will still be adhering to the fundamental laws of nature on a best-effort basis. Being able to sacrifice physical accuracy, such as varying physical constants, allows more freedom when setting up scenarios to test the floating-point formats.

Chapter 2

Background

2.1 Floating-point

Floating-point numbers are used to approximate real numbers in various computing scenarios. They are represented using three numbers: $sign$, $significand$ and $exponent$:

$$(-1)^{sign} \times significand \times 2^{exponent}$$

Without loss of generality, $sign$ and $exponent$ may be assumed to be integers. However, in order to represent any real value, $significand$ should be able to assume any real number in the range $[1, 2)$. However, that is not possible in practice. Instead $significand$ is commonly approximated using a fraction of the form $significand = 1 + \frac{fraction}{2^k}$, where $fraction$ is a k-bit integer (Gustafson and Yonemoto 2017).

The number of bits used to represent a floating-point number is usually limited to deal with memory, circuitry, and performance constraints. In turn, this introduces an upper and lower bound on the values that $fraction$, $exponent$ and $sign$ may assume, since they all need to fit within the available bits. Because $sign$ only ever requires 1 bit to determine if the number is positive or negative, $fraction$ and $exponent$ may divide the remaining bits among them, which is a balancing act.

On the one hand, using more bits for $exponent$ increases the dynamic range:

the ratio between the largest and smallest representable number. Assuming $exponent$ is an e-bit integer, $exponent$ lies in the range $[0, 2^e - 1]$. The exponential part for the floating-point number thus lies in the range $[2^0, 2^{2^e-1}]$, leading to a dynamic range of $DR_{exponent} = 2^{2^e-1}$. Additionally, $significand$ has fixed dynamic range $DR_{significand} = 2$, since it lies in the range $[1, 2)$. For the entire floating-point number, this gives the dynamic range: $DR_{significand} \times DR_{exponent} = 2 \times 2^{2^e-1} = 2^{2^e}$. As can be seen, the dynamic range directly depends on the number of bits e of the $exponent$.

On the other hand, using more bits for $fraction$ increases the precision. *High precision* means that the difference between two consecutive representable numbers is relatively small. Similarly, *low precision* means that the difference is relatively large. For example, two consecutive numbers may be written as $A(f)$ and $A(f+1)$, given that $A(f) = (1 + \frac{f}{2^k}) \times 2^{exponent}$. Calculating the difference between these gives the following:

$$\begin{aligned} A(f+1) - A(f) &= \left(1 + \frac{f+1}{2^k}\right) \times 2^{exponent} - \left(1 + \frac{f}{2^k}\right) \times 2^{exponent} \\ &= \frac{2^{exponent}}{2^k} \\ &= 2^{exponent-k} \end{aligned} \tag{2.1}$$

As evident from equation 2.1, the difference increases exponentially with the value of $exponent$ but decreases exponentially with k. Thus, increasing the number of bits, k, for the fraction causes the precision to increase.

One important property inherent to floating-point numbers is that a calculation may end up with a value in-between two consecutive representable numbers. If this is the case, the value would have to be rounded to one of the two consecutive representable numbers, which introduces so-called *rounding errors*. This error is at most half the distance to the next consecutive number, since the number is either rounded up or down. As a result from equation 2.1, the upper bound of the rounding-error is exactly $\frac{2^{exponent-k}}{2}$. Notably, when $exponent$ is large, rounding errors are potentially much larger than when $exponent$ is small. Additionally, by increasing k, the rounding error may be reduced further.

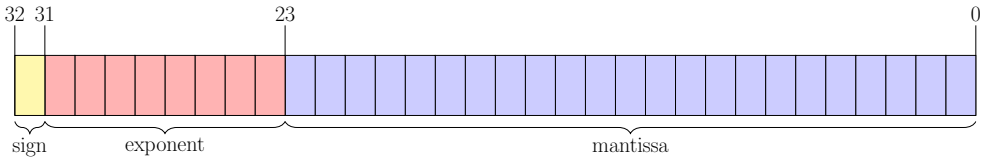


Figure 2.1: Memory layout for a binary 32-bit IEEE-754 floating-point number.

2.1.1 IEEE-754

IEEE-754 is a standard that specifies binary memory layouts for floating-point formats. These formats use the same three components described above to represent a floating-point number: a sign, an exponent, and a fraction (also known as mantissa in the IEEE-754 standard). IEEE-754 specifies that these components get stored in the following order: first the sign, then the exponent, and finally the fraction. The sign always takes up one bit, whereas the exponent and fraction have a fixed number of bits that depend on the specific variant used.

The two variants of IEEE-754 that are most common in practice use 32 and 64 bits. Formally, they are referred to as *single-precision* and *double-precision*, but may informally be called *float* and *double*, respectively. Single-precision IEEE-754 uses 8 bits for the exponent and 23 bits for the fraction (see figure 2.1), while double-precision has an 11-bit exponent and 52-bit fraction (Kahan 1996).

Other than representing finite real numbers, IEEE-754 can also depict positive or negative infinity, as well as NaN:s (Not a Number) (Kahan 1996). A number with only 1-bits in the exponent and 0-bits in the fraction represents an infinity. Although, if such a number instead had a 1-bit anywhere in its fraction, it would represent a NaN (Kahan 1996).

2.1.2 Posit

Posit is a novel floating-point format introduced by Gustafson and Yonemoto (2017). It attempts to improve upon the IEEE-754 formats by increasing precision. Posit achieves this by dynamically adjusting the size of each component in its representation depending on the magnitude of the number. Specifically,

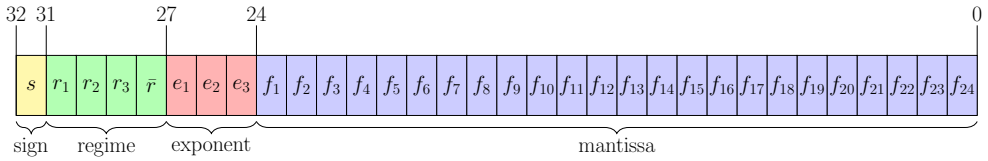


Figure 2.2: Memory layout for a 32-bit Posit ($es = 3$) floating-point number between 2^{16} (inclusive) and 2^{24} (exclusive).

Posits allocate more bits to the significand when representing numbers near 1. Gustafson and Yonemoto (2017) argue that these are the numbers with which most calculations occur. Additionally, Posit has fewer bit-patterns reserved for exceptional values (infinities and NaN:s), which frees up more bit-patterns to use for actual values.

The binary representation of Posit, as presented by Gustafson and Yonemoto (2017), consists of the same three parts as above, but with one additional field: the regime. The number of bits used to represent the regime and the fraction is adjusted dynamically, depending on the number's magnitude. Meanwhile, the sign and the exponent use a fixed number of bits: one bit for the sign, and es (short for *exponent size*) bits for the exponent. For an n-bit Posit, es can be any arbitrary integer in the range $[0, n-3]$. Gustafson and Yonemoto (2017) recommend using $es = 3$ for a 32-bit Posit, and $es = 1$ for a 16-bit Posit.

An n-bit Posit is laid out in memory sequentially as follows:

- 1 bit for the sign (s)
- $m \geq 1$ bits for the regime (r_1, r_2, ..., r_m)
- 1 bit for a regime terminator bit ($\bar{r}$)
- es exponent bits (e_1, e_2, ..., e_{es})
- $k \geq 0$ fraction bits (f_1, f_2, ..., f_k)

Figures 2.2 and 2.3 illustrate the difference between a Posit representing small and large values, respectively. Note how the number of bits allocated to the regime and fraction vary.

Posit achieves its dynamic precision by using a *run-length* encoding of the regime: an m-bit sequence of either only 1's or only 0's. This sequence is either terminated by the end of the binary string or the regime terminator bit, which is a bit of opposite value compared to the other regime bits (a 0 or 1 in the respective case). The value of the regime is then given by the number of

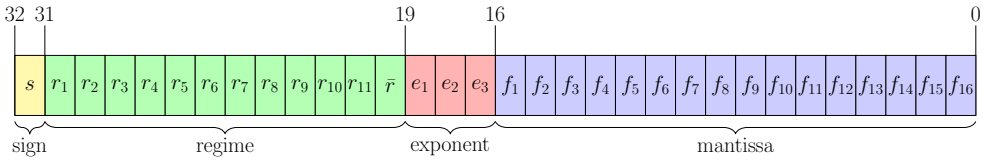


Figure 2.3: Memory layout for a 32-bit Posit ($es = 3$) floating-point number between 2^{80} (inclusive) and 2^{88} (exclusive).

regime bits m, using the following formula (Gustafson and Yonemoto 2017):

$$regime = \begin{cases} -m & if\ r_i = 0,\ \forall i \in [1,\ m] \\ m - 1 & if\ r_i = 1,\ \forall i \in [1,\ m] \end{cases}$$

It might be the case that m is as large as $n - 1$, in which case there are no more available bits for the exponent or fraction. If this happens, the bits of the exponent and fraction that do not fit are treated as if they were 0. Since es is usually a small constant, the only way for a Posit to represent a number greater than 2^{es} is to make the regime larger, which requires more bits. In turn, this causes a loss of precision as fraction bits get excluded from the binary representation to make space for the regime (Gustafson and Yonemoto 2017).

The value represented by a Posit is finally given by the formula:

$$(-1)^{sign} \times useed^{regime} \times 2^{exponent} \times (1 + \frac{fraction}{2^k})$$

where $sign$, $regime$, $exponent$ and $fraction$ are the integer values of the sign bit, regime, exponent and fraction, respectively, and $useed$ is defined by:

$$useed = 2^{2^{es}}$$

There are only two exceptional Posit values defined by Gustafson and Yonemoto (2017): 0 and $\pm\infty$ (note that positive and negative infinity are the same number). These use the reserved bit sequences $000...000$ and $100...000$, respectively. Contrast this to 32-bit IEEE-754, which defines millions of bit sequences reserved for exceptional values.

2.2 Related work

Earlier research in this area has measured the differences in accuracy between IEEE-754 and Posit for various computations, such as matrix-matrix multiplication (De Blasio and Ekstedt Karpers 2020), the Fast Fourier Transform (Chien, Peng, and Markidis 2019), and the LINPACK benchmark (Gustafson and Yonemoto 2017). Gustafson and Yonemoto (2017) also presented results on some contrived problems that were constructed to introduce substantial rounding errors. These included evaluating a complex mathematical expression using cherry-picked constants, and *Goldberg's thin triangle problem*.

All of the previously mentioned studies showed that, when using the same number of bits, Posit consistently achieves higher accuracy than IEEE-754 when the magnitude of the numbers is around 1.[1] De Blasio and Ekstedt Karpers (2020) showed that for numbers outside this range, it is IEEE-754 that achieves higher precision. Additionally, Dinechin et al. (2019) concluded that Posit performs especially poorly when mixing numbers from a large range of magnitudes. They pointed out that this becomes very relevant in physics, where constants range from very small (such as the Planck constant $h \approx 6.626 \times 10^{-34}$) to very large (such as the Avogadro constant $N_A \approx 6.022 \times 10^{23}$).

2.3 Physics Simulations

Physics simulations are used to model behaviors in the real world, such as the movement of particles in currents or gravity in a solar system. When performing simulations on computers, the number of iterations in the simulation becomes significant since rounding errors accumulate over time. If these errors grow too large, they may render the results from the simulation unreliable, which would negate the entire purpose of the simulation.

[1] It should be noted that the measurements presented by Gustafson and Yonemoto (2017) may be biased to favor Posit; John Gustafson is credited to be the inventor of the Posit format and is thus likely to present it in a good light to accelerate adoption.

2.3.1 N-body simulations

Cosmological N-body simulations are used in physics and astrology to simulate forces (usually gravity), acting between a set of N bodies in space. An example use case of this kind of simulation is to predict the movement of astral bodies in different parts of the universe.

The goal of an N-body simulation is to compute the positions and velocities of bodies in space. The computation is divided into multiple discrete iteration steps, with each iteration representing a small step in time Δt. At some simulation step i, it is assumed that the position p_i and velocity v_i of each body is known. In addition to position and velocity, each body also has a mass m, which is kept constant throughout the simulation.

Initially it is assumed that p_0 and v_0 are known for all bodies. The goal of the simulation is then to compute p_{i+1} and v_{i+1} given the values of p_i and v_i.

Assuming that acceleration is uniform during the entire iteration step, the Equations of Instantaneous Motion may be used to approximate[2] p_{i+1} and v_{i+1} for a given body x as follows:

$$v_{i+1} = v_i + a_x \Delta t \tag{2.2}$$

and

$$p_{i+1} = p_i + v_{i+1} \Delta t \tag{2.3}$$

where a_x is the acceleration of the given body x, which is given by Newton's Second Law of Motion:

$$a_x = \frac{F_x}{m_x} \tag{2.4}$$

Here, m_x is the body's mass, and F_x is the sum of all forces acting on it.

The value of F_x depends on which forces are acting on the bodies. If the force in question is gravity, Newton's Law of Universal gravity can be used to calculate the force F_{xy} between two distinct bodies x and y:

$$F_{xy} = G\frac{m_x m_y}{r_{xy}^2} \tag{2.5}$$

[2]There are multiple alternative methods for computing the bodies' positions with higher numerical accuracy, such as the Adaptive Verlet (Huang and Leimkuhler 1997) and Runge-Kutta methods (Dormand 1978).

where G is the gravitational constant, r_{xy} the euclidean distance between the two bodies, and m_x and m_y their respective masses. Note that r_{xy} only depends on the positions of the bodies.

Using equation 2.5 above, the total force exerted on the reference body x can then be calculated by adding all forces acting on that body:

$$F_x = \sum_{y \in B - \{x\}} F_{xy} \tag{2.6}$$

where B is the set of all bodies in the N-body simulation.

Reversability

Using the equations above, it is also possible to infer a procedure for running an N-body simulation in reverse. Assume that p_{i+1} and v_{i+1} are known for a given body x. In order to run the simulation in reverse, the body's position p_i and velocity v_i in the previous iteration need to be computed. Using equation 2.3 immediately yields $p_i = p_{i+1} - v_{i+1}\Delta t$. Using p_i, F_x is then computed as in equation 2.5, which also yields a_x from equation 2.4. Finally, solving for v_i in equation 2.2 gives $v_i = v_{i+1} - a_x \Delta t$.

Running the Simulation

By repeatedly applying the procedures described above, it is possible to compute the positions and velocities of all bodies in an N-body system at any iteration step, both infinitely far forward or backward in time.

Chapter 3

Methodology

The first step of this study was to implement an N-body simulation that supported switching between the different floating-point formats that were examined in this study. Two different methods were developed to determine the accuracy of each floating-point format in various situations. For the first method, the simulations that used Posit and IEEE-754 were compared to a baseline simulation that used a high-precision floating-point format. In the second, the bodies' initial positions were used as a baseline instead, which was then compared to the positions that resulted from running the simulation forward and then in reverse, for an equal number of simulation steps.

3.1 Simulation

Listed below are snippets of pseudo-code that implement the equations described in section 2.3.1.

3.1.1 Gravitational Force

The procedure `gravity_force` implements equation 2.5, which determines the gravitational force between two distinct bodies `a` and `b`:

```
-- Return the gravitational force acting between two bodies
function gravity_force(a, b)
    distance  = distance(a.position, b.position)
    magnitude = G * a.mass * b.mass / (distance * distance)
    direction = (b.position - a.position) / distance
    return magnitude * direction
```

The `distance` function is assumed to return the Euclidean distance between two vectors.

3.1.2 Total Force

The procedure `total_force` implements equation 2.6, which determines the total force acting on a body `i`:

```
-- Returns the total force acting on body i
function total_force(bodies, i)
    force = [0, 0]
    for j = 1 to N do
        if i != j then
            force += gravity_force(bodies[i], bodies[j])
    return force
```

3.1.3 Stepping Forward

Equations 2.2 and 2.3 are implemented in `step_forward`. This function computes a single simulation iteration by updating the bodies' positions and velocities according to their acceleration:

```
-- Update the simulation to the next iteration
function step_forward(bodies, time_step)
    for i = 1 to N do
        force = total_force(bodies, i)
        acceleration = force / bodies[i].mass
        bodies[i].velocity += acceleration * time_step
    for i = 1 to N do
        bodies[i].position += bodies[i].velocity * time_step
```

We update velocities and positions in separate loops to ensure that all forces get calculated using the identical positions of the bodies. It is safe to update the bodies' velocities in the first loop since the force between two bodies only depends on their positions and masses, but the same is not the case for the positions.

3.1.4 Stepping in Reverse

In order to reverse a simulation, as seen in section 2.3.1, code similar to that of `step_forward` was used. The only differences were the flipped order of the two loops and the replacement of additions with subtractions, as can be seen in `step_reverse`:

```
-- Update the simulation to the previous iteration
function step_reverse(bodies, time_step)
    for i = 1 to N do
        bodies[i].position -= bodies[i].velocity * time_step
    for i = 1 to N do
        force = total_force(bodies, i)
        acceleration = force / bodies[i].mass
        bodies[i].velocity -= acceleration * time_step
```

`step_reverse` works as an inverse to `step_forward` due to the following (assuming the arithmetic is without rounding errors):

1. In the first loop of `step_reverse`, the bodies' positions are set to those of the previous iteration. This reverses the effect of the last loop in `step_forward`.
2. Using the positions from the previous iteration, every force will be identical to the corresponding force in `step_forward`. This is due to forces only depending on bodies' positions and masses.
3. Using these forces, the effect of the first loop in `step_forward` is reversed, resetting the bodies' velocities to that of the previous iteration.

3.2 Software Simulation of Posit

As described in section 2.1.1, most modern computers have hardware support for the IEEE-754 floating-point format. However, this is not the case for Posit, which meant that Posit arithmetic had to be emulated in software. We used the SoftPosit library (Leong 2021) for this task. SoftPosit implements software emulation of multiple sizes of Posits, notably the 32-bit Posit ($es = 3$) used in this thesis.

3.3 Measuring Error

In order to compare a given simulation to the baseline, the *mean squared error* (MSE) metric was used. The MSE for two sets of n points A_i and B_i is defined as:

$$\text{MSE} = \frac{1}{Q}\sum_{i=1}^{n} ||A_i - B_i||^2 \tag{3.1}$$

where Q is a normalization constant defined as:

$$Q = \frac{1}{n}\sum_{i=1}^{n} ||I_i||^2 \tag{3.2}$$

where I_i is the set of initial positions of all bodies.

By letting A_i and B_i in equation 3.1 be the bodies' positions from two different simulations, the MSE effectively measures of how much the two simulations deviate from each other. The MSE is weighted toward favoring small differences over large differences. That is, a large difference between two positions results in a much larger contribution to the total error than a small difference does. This is due to calculating the square of the difference in equation 3.1. For our case, this is desirable since, in a simulation, small deviations from the actual value are generally acceptable, but large deviations are not.

The MSE becomes useful when comparing the precision of different floating-point formats. A comparison is performed by first running the simulation using one of the tested formats for an arbitrary number of time-steps. We may then calculate the MSE between that simulation and some baseline simulation. Doing this for multiple formats using the same initial values and baseline allows

us to rank them by their MSE, where a smaller MSE corresponds to higher precision.

Do note that such a ranking does not depend on the normalization constant Q. Since the initial values of all simulations are equal, the value of Q is as well. Therefore, the relative difference between any format's MSE does not change by varying Q.

The purpose of the normalization constant Q is to allow for direct comparison of MSE between simulations whose initial values differ. Without normalization by Q, we would expect the MSE to be greater for a simulation where the bodies' initial positions are large, compared to one where the positions are small. This is due to the fact that errors produced by rounding a floating-point number grow linearly with the magnitude of that number, as shown in equation 2.1. Thus, to cancel this effect, we divide the MSE by a constant roughly on the same order of magnitude as the square of the positions themselves. In other words, the normalization constant Q.

3.3.1 Using a High Precision Baseline

One way to acquire a baseline for use in the comparison procedure described above, is to run a simulation using a high-precision floating-point format. This simulation results in a set of bodies whose positions are closer to the exact value than any of the tested floating-point formats. Using this as a baseline for the MSE gives us an error that approximately corresponds to how far a given simulation is from exact.

Previous studies in this area have used the 64-bit variant of IEEE-754 as their high-precision floating-point format. However, for this paper, we decided to use a floating-point format with even higher precision in order to increase confidence in our results. For this purpose, we used *The GNU Multiple Precision Arithmetic Library* (GMP, Free Software Foundation (2021)). The GMP library implements software emulation of IEEE-754 style floating-point numbers of arbitrary sizes. In this thesis we used their 1024-bit floating-point format.

3.3.2 Reversing the Simulation

The other approach for acquiring a baseline uses the reversibility property of N-body simulations. By running a simulation T iteration steps forward and then T steps backward, one may expect that the simulation returns to the initial state. However, as an inexact floating-point format accumulates errors while running the simulation, the simulation may never get back to the initial state, only close to it. We may then use the original initial state as a baseline to compare against the results from running the simulation forward and then backward.

Unlike the first method, this method does not rely on approximation in order to acquire a baseline, since the bodies' exact initial positions are used. This method is thus included in order to allow for a comparison against the other method that approximates the baseline. Furthermore, including a second method will hopefully add another perspective to the comparison between the floating point formats.

3.4 Test Cases

In an N-body simulation, multiple parameters can be varied depending on the use case. For example, both the gravitational constant and the number of bodies may be changed. For each body, it is also possible to adjust its mass, position, and velocity. The benefit of such a large number of parameters is the freedom it allows when generating test cases, but it also comes with some issues.

Different parameters affect different parts of the simulation, which impacts different aspects of the floating-point formats. For example, to test the accuracy of the formats when manipulating numbers of large magnitudes, a simulation could be run where the bodies' initial positions are large. Therefore, we generated test cases using positions, velocities, masses, and time-steps from a large range of magnitudes. The purpose of this was to see in which ranges the different floating-point formats perform better or worse.

However, increasing the magnitude of the bodies' positions also means that their positions will not change much from iteration to iteration unless their ve-

locities or the time-step are also of large magnitudes. The problem is that, unless the bodies' positions change, the simulations will not have time to diverge, leading to monotonous results. Thus, we only generated test cases where there was noticeable movement.

The number of bodies in a simulation also influences the quality of results. Since a single iteration of the N-body algorithm runs in $\mathcal{O}(N^2)$ time, the number of operations increases quadratically with the number of bodies. Thus, by increasing the number of bodies in the simulation, we drastically increase the number of floating-point operations performed. Hence we expect the accumulated rounding errors to grow at a higher rate as well.

3.4.1 Generating Test Cases

Taking all of the above into account, several test cases were generated based on a simple formula. This formula controlled the bodies' initial positions p, the time-step Δt, and the gravitational constant G. The number of bodies N, their velocities v, and masses m were kept constant throughout all test cases.

The values of p, Δt, and G were then chosen from a uniform random distribution, such that they satisfy the following, for some integer A:

$$\begin{aligned} p_{min} &= 10^A \\ p_{max} &= 2 \times 10^A \\ ||p|| &\in [p_{min},\ p_{max}] \\ \Delta t &= 10^{A-3} \\ G &= 10^A \end{aligned}$$

The formula above was developed through trial and error. This final version of the formula was chosen because it can produce stable simulations where all bodies display movement for all values of A used in this thesis. Additionally, this formula keeps parameters at an approximately equal order of magnitude.

When generating a test case, the bodies were placed within a circle in the cartesian plane, centered around the origin. The bodies' distances from the origin were randomly selected from the range $[p_{min}, p_{max}]$. This effectively distributed the bodies in a donut shape around the origin, resembling the or-

biting planets in a solar system. One additional body was placed at the origin, corresponding to the sun.

The orbiting planets' velocities were initialized to a direction orthogonal to the origin, such that they orbited the sun counter-clockwise. The velocities' magnitudes were randomly set to a value in the range $[4, 5]$. This range produced stable orbits and was chosen by trying out different values. The sun's velocity was initialized to 0.

The masses of the planets were randomly selected from the range $[1, 2]$, while the sun's mass was randomly selected from the range $[10,\ 20]$. These values were arbitrarily chosen to resemble a sun whose mass is greater than those of the planets.

One benefit of computing the positions of the bodies based on A in this manner, is that it makes it relatively easy to approximate the normalization constant Q. Assuming that $||p||$ is chosen from a uniform distribution, we calculate the expected value as:

$$|p| = \frac{p_{min} + p_{max}}{2} = \frac{10^A + 2 \times 10^A}{2} = \frac{3}{2}10^A$$

Given the definition of Q in equation 3.2, and plugging in the expected value of $||p||$, we get:

$$Q = \frac{1}{n}\sum_{i=1}^{n} ||p||^2 = ||p||^2 = \left(\frac{3}{2}10^A\right)^2 = \frac{9}{4}10^{2A} \approx 10^{2A}$$

The reader might want to keep this trick in mind when examining the results below.

3.4.2 List of Test Cases

The list of test cases used to produce the results in this thesis was randomly generated by setting $N = 128$ for all test cases, meaning every simulation used 128 bodies (127 for the planets and 1 for the sun). By then varying the value of A, we produced several test cases where the simulation parameters ranged over multiple orders of magnitudes. The following values were chosen for A: -10, -6, -3, -1, 0, 1, 3, 6, 10 and 16.

Chapter 4

Results

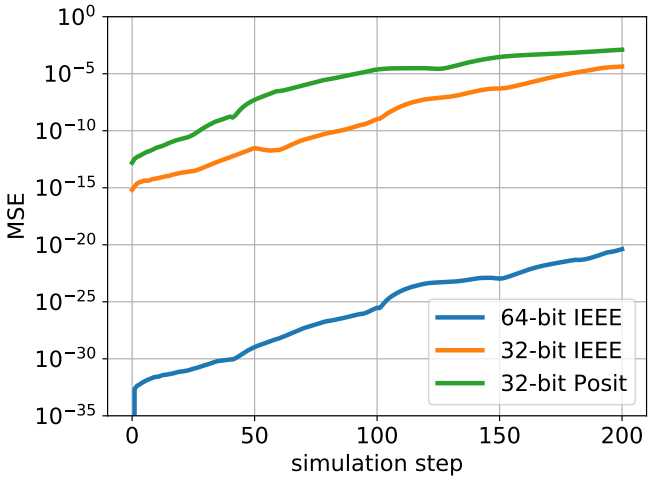
This study aimed to examine the accuracy of Posit and IEEE-754 when used in N-body simulations. In order to accomplish this, we used two different methods to measure the accuracy. The first method calculated the errors (MSE) of all formats relative to a high-precision 1024-bit floating-point format (section 4.1). The second method utilized the reversibility property of the N-body implementation by running the simulation forward and then backward the same number of steps. By comparing the initial positions to the resulting positions, the error (MSE) was calculated (section 4.2).

For every step of each simulation, the MSE was plotted for the 32-bit variants of IEEE-754 and Posit. In addition to the 32-bit variants, the 64-bit variant of IEEE-754 was also included to put the results from other formats in a context.

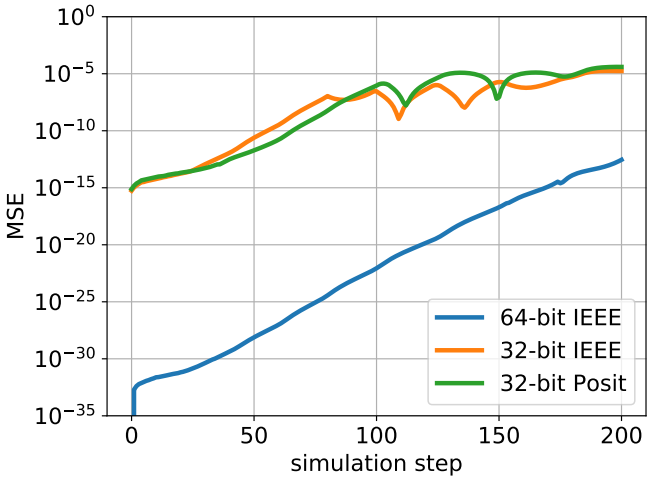
4.1 Using a 1024-bit Baseline

When observing figure 4.2, it becomes apparent that 32-bit IEEE-754 is better with extreme values, while 32-bit Posit is more accurate with numbers near 1 in particular. Figure 4.1a shows the results from the test case where $A = -10$. In this plot, 32-bit IEEE-754 is consistently more accurate than 32-bit Posit. However, when $A = -6$ (figure 4.1b), 32-bit Posit and IEEE-754 end up having similar errors. For $A = -3$, $A = -1$ and $A = 0$ (figures 4.1c, 4.1d and 4.1e, respectively), Posit is consistently more accurate than the corresponding

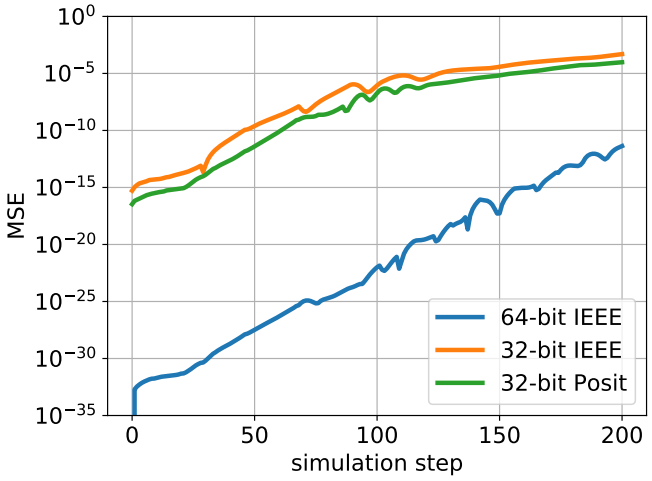
IEEE-754 format. 32-bit Posit reaches its peak in performance compared to 32-bit IEEE-754 when $A = 0$ (figure 4.1e), where the initial positions are near 1. For $A = 1$ and $A = 3$ (figures 4.1f and 4.1g), 32-bit IEEE-754 is better than 32-bit Posit at certain time-steps, and when $A = 6$ (figure 4.1h) 32-bit IEEE-754 is once again consistently more accurate than Posit. The gap between IEEE-754 and Posit only increases throughout the last two test cases where $A = 10$ (figure 4.1i) and $A = 16$ (figure 4.1j).



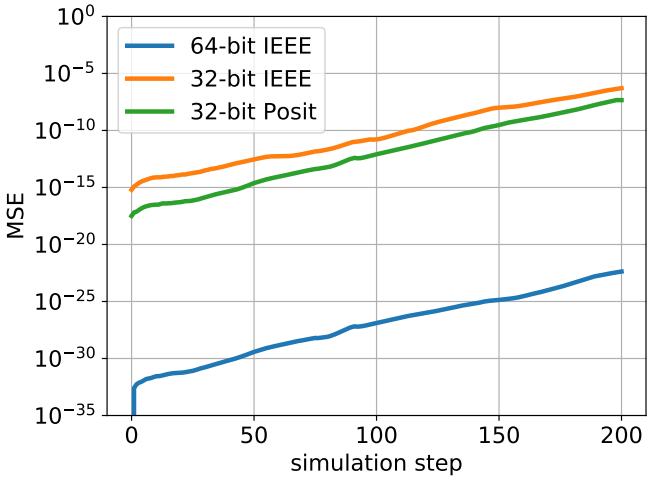
(a) $A = -10$



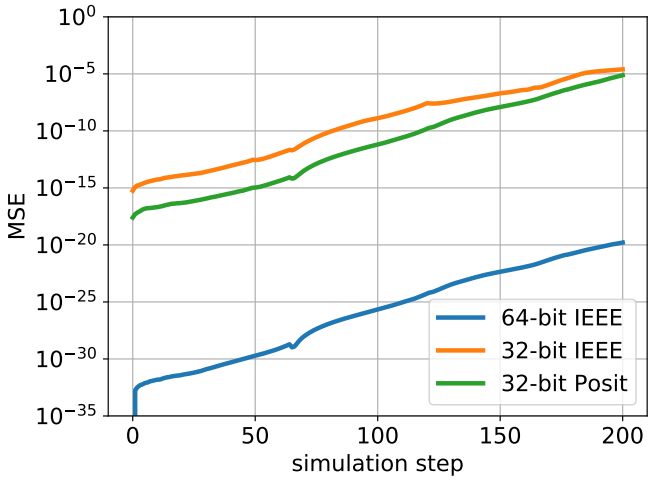
(b) $A = -6$



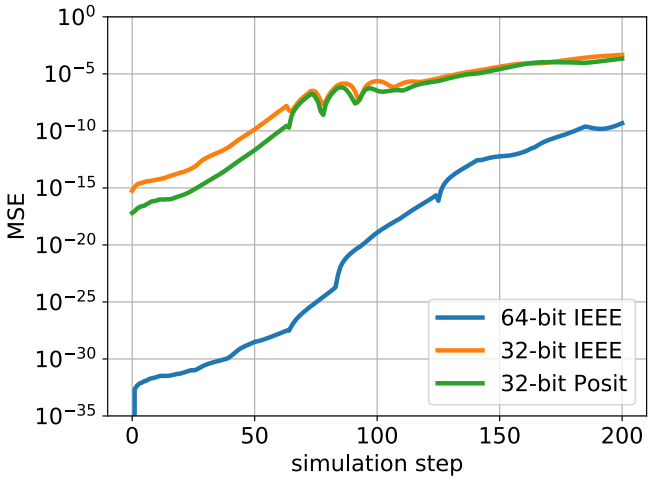
(c) $A = -3$



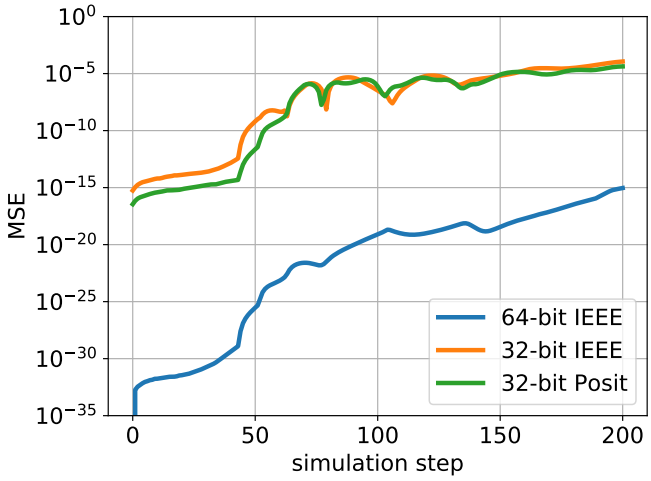
(d) $A = -1$



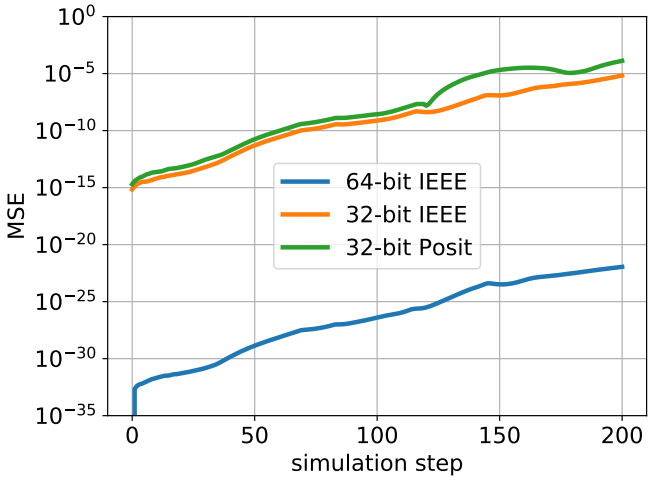
(e) $A = 0$



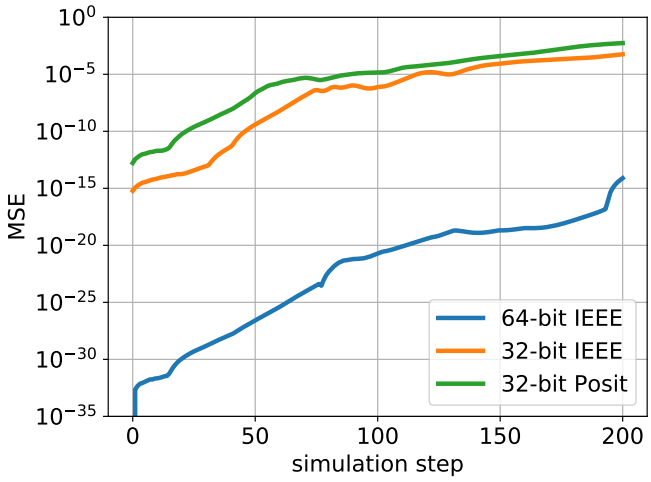
(f) $A = 1$



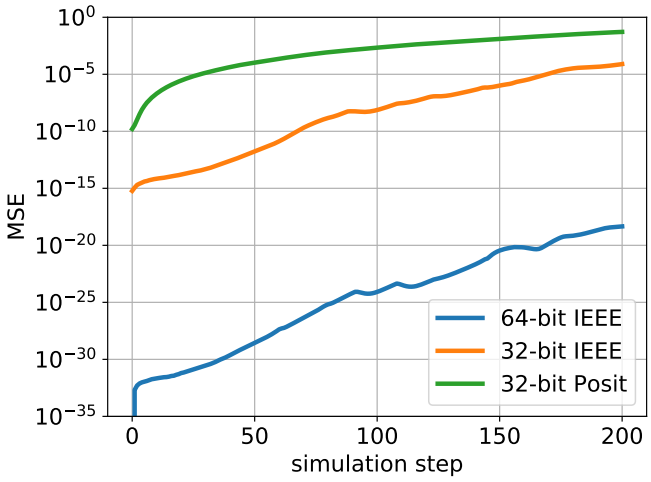
(g) $A = 3$



(h) $A = 6$



(i) $A = 10$



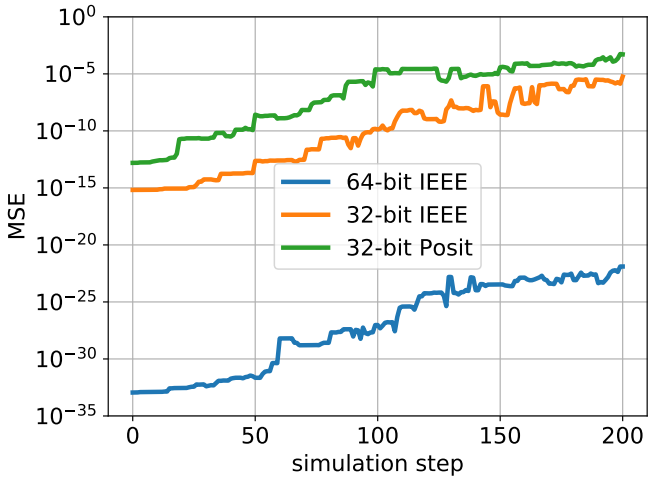
(j) $A = 16$

Figure 4.1: MSE plotted over time, calculated using a 1024-bit arbitrary precision baseline (lower is better). 64-bit IEEE is included for reference.

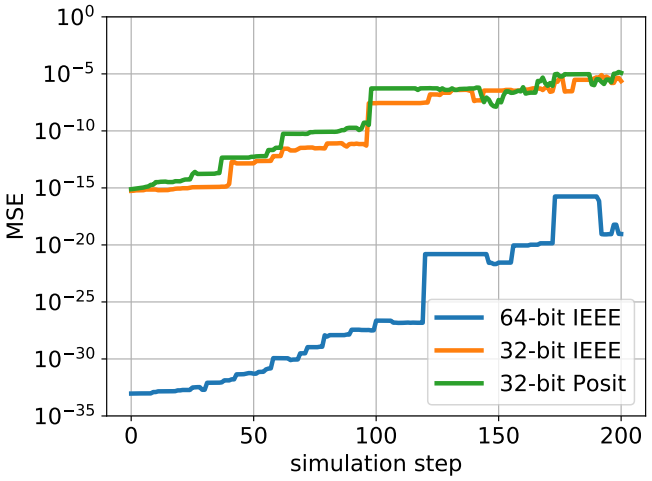
4.2 Using Reversability

Overall, the results from the reversability method are somewhat similar to those from the first method. For instance, in the plots where $A = -10$ (figures 4.1a and 4.2a) and $A = 16$ (figures 4.1j and 4.2j), 32-bit IEEE-754 is consistently better than 32-bit Posit. Similarly, when $A = -1$ (figures 4.1d and 4.2d) and when $A = 0$ (figures 4.1e and 4.2e), 32-bit Posit has a smaller MSE than 32-bit IEEE-754 throughout the entire simulation. Another similarity among the results is that whenever the MSE:s from 32-bit Posit and 32-bit IEEE-754 overlap for the first method (namely in figure 4.1b, figure 4.1f and figure 4.1g), they also overlap in the corresponding plots from the second method (figures 4.2b, 4.2f, and 4.2g).

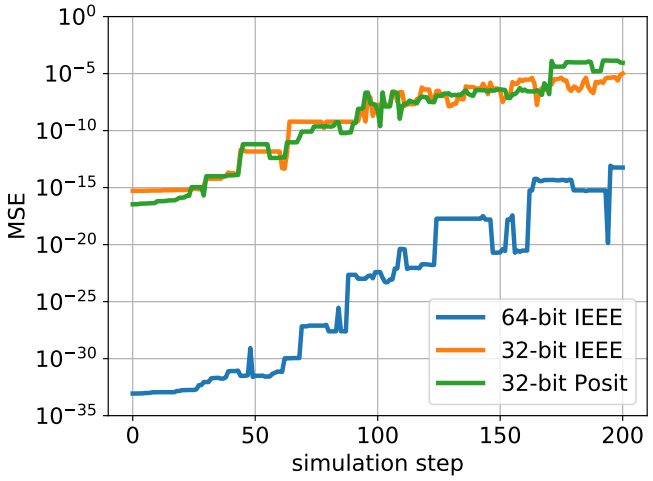
However, the results from the two methods also differ in several ways. To begin with, the last reversibility MSE:s for 32-bit Posit and 32-bit IEEE-754 in every plot are off by roughly a factor of 10 to 1000, compared to the MSE:s from the first method. Additionally, the graphs of 32-bit Posit and 32-bit IEEE-754 intersect much more frequently in the reversibility results. This is made evident by comparing the plots where $A = -3$ (figures 4.1c and 4.2c), $A = 6$ (figures 4.1h and 4.2h) and $A = 10$ (figures 4.1i and 4.2i). The differences in these three comparisons may stem from the overall unstable MSE:s from the reversibility method. At certain time-steps, the severe oscillations in the reversibility plots cause enormous and rapid changes in the MSE. Such is the case in figure 4.2g, where the MSE of 32-bit IEEE-754 decreased with a factor of 10^7 in the span of a single time-step.



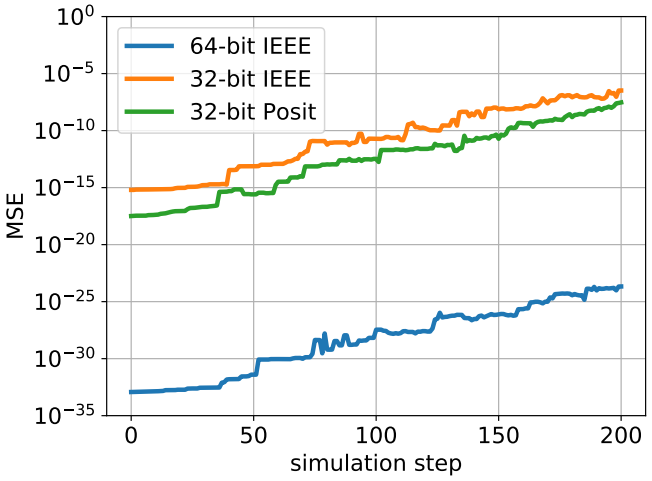
(a) $A = -10$



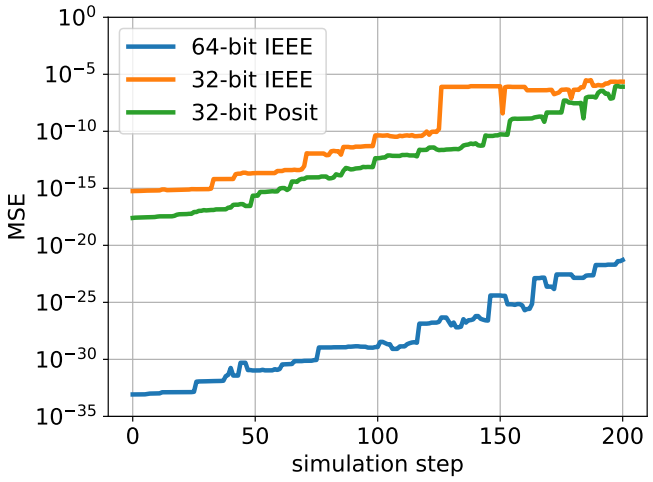
(b) $A = -6$



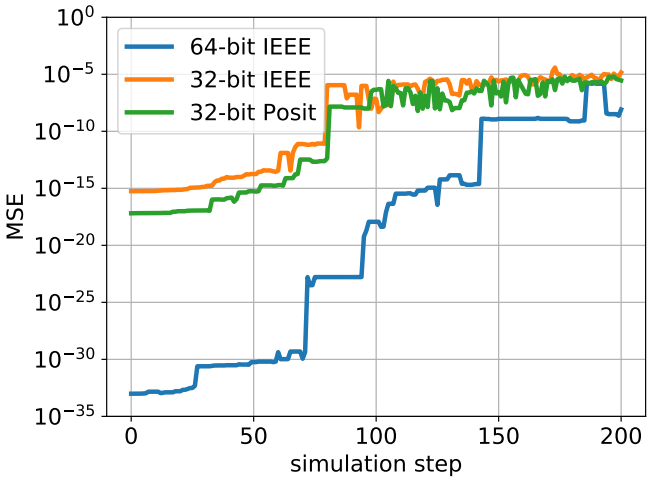
(c) $A = -3$



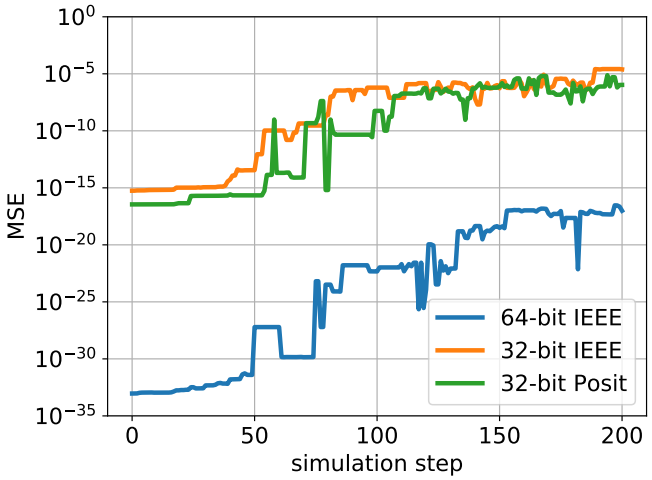
(d) $A = -1$



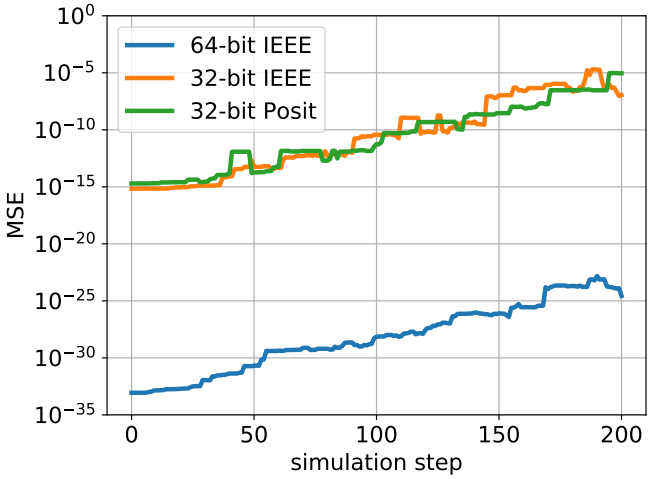
(e) $A = 0$



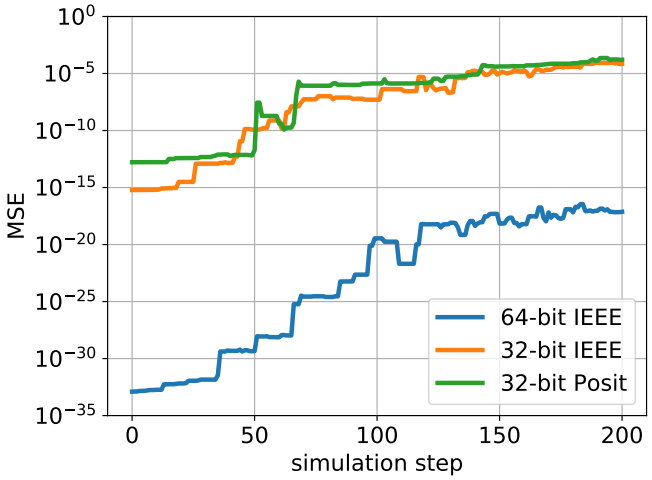
(f) $A = 1$



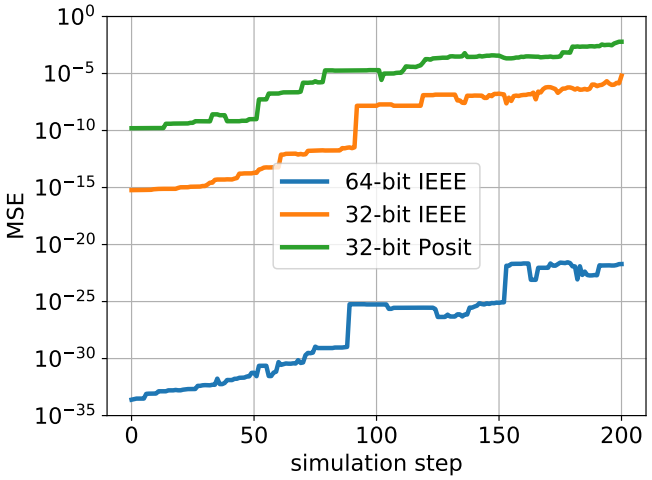
(g) $A = 3$



(h) $A = 6$



(i) $A = 10$



(j) $A = 16$

Figure 4.2: MSE plotted over time, calculated using the reversability method (lower is better). 64-bit IEEE is included for reference.

Chapter 5

Discussion

The results of this study show that 32-bit Posit outperforms or matches the precision of 32-bit IEEE-754 in N-body simulations when the bodies' positions are roughly in the interval $[10^{-3},\ 10^{3}]$. However, as the magnitude of the positions approaches extreme values, in either direction outside this interval, IEEE-754 gains the advantage. Our findings are consistent with the previous research performed in this area by Chien, Peng, and Markidis (2019), and De Blasio and Ekstedt Karpers (2020).

For N-body simulations, it could be hard to fully take advantage of the benefits of Posit since the distance between two bodies might span a relatively large range. For instance, consider a simulation that models the interaction of planets in a solar system as wel as the interaction of solar systems in a galaxy. In such a case, the distances between the bodies could fall outside the interval within which Posit outperforms IEEE-754. If this were to happen, it would nullify any benefit that might have been gained by using Posit. Therefore, making sure that values stay within this interval is paramount.

5.1 Normalizing Units

One solution to keeping values within the optimal interval of Posit, as suggested by Dinechin et al. (2019), is to normalize the units in the simulation. Normalization should be done so that most calculations end up dealing with

numbers near 1. For example, when representing a distance of 1000 meters, Posit would achieve higher accuracy if it were instead interpreted as 1 kilometer. In an N-body simulation, this means one could use *light-years* or *parsecs* to measure distance rather than kilometers or, similarly, measure time in hours instead of seconds. Note that changing the units in this manner is conceptually equivalent to adding a fixed offset to the exponent in the floating-point representation of the value.

However, mixing units in this way comes at the cost of introducing additional complexity to the programmer, who now needs to consider what units to use for different simulation scenarios. For example, using light-years as a unit of measure might work fine on the scale of stars, but not on the scale of galaxies. Thus, the programmer would need to use another unit of measure for simulations on that scale.

One way to address this problem is to determine the unit of measure dynamically. Tracking units dynamically could be achieved by, for instance, storing the unit and value together in memory. Although, this does come at an additional memory cost. At that point, one could equally well use that additional memory to switch to a floating-point format with higher precision, such as the 64-bit IEEE-754 variant.

Nonetheless, if memory usage is a valid concern, and the jump from 32-bits to 64-bits is too costly, 32-bit Posit has clear advantages over IEEE-754 under specific circumstances. However, apart from a few niche cases, we would expect the consistency of IEEE-754 to outweigh the benefits gained by using Posit.

5.2 Future Work

In order to get the most out of Posit, the values in the calculations need to be close to 1. In section 5.1, we described how normalizing units could help increase the likelihood that this happens. However, such an approach is not straightforward to implement and relies on the use case being easy to predict.

Another approach to normalizing units could be to use more sophisticated data structures to track the units used by different values dynamically. For example, one may have multiple lists of numbers, where each list only contains values

with a specific unit. A floating-point number in the software would then simply be an index into one of these lists. Further research into the feasibility of this and alternative approaches to normalizing units is needed.

5.3 Limitations

For this thesis, physical accuracy was not strictly needed. Therefore it was sacrificed to keep the implementation simpler and this thesis more approachable for those without extensive knowledge of physics and numerical methods. Hence, this study is not truly representative of real-world N-body simulations used for practical purposes. A more faithful study would have been implemented using advanced techniques that are more common in practice, such as the Barnes-Hut algorithm and one of the higher-order Runge-Kutta methods.

5.4 Comparing the Methods

When comparing the results from the two methods, it becomes clear that their implications on the accuracy of Posit are more or less the same. Both methods imply that Posit is preferable when the calculations involve numbers that are near 1 or not extreme in either direction. This observation serves to strengthen the reliability of the methods and the credibility of the results.

However, the plotted MSE:s had notable fluctuations for the reversibility method. One explanation for these fluctuations is that the N-body problem is chaotic. Even for small changes in the input, the output may vary drastically after just a few simulation steps. This could, for example, happen when two bodies get close to each other, in which case the forces between them would grow very large (Newton's Law of Universal Gravity, equation 2.5). Given that we have two simulations with initial values that have minor differences, it is possible that in one of the simulations, two bodies get very close to one another, but that in the other, those two bodies do not get as close. This causes vast differences in the bodies' orbits in the first simulation compared to the other because much larger forces are being exerted on the bodies in the former simulation.

When stepping the simulation forward once and then backward again, bodies

may end up a bit translated due to rounding errors. As previously described, this may cause chaotic behavior that places the bodies on slightly different orbits with an error that magnifies over time. It is also possible that stepping the simulation forward twice causes the rounding errors to cancel each other out, which places the bodies on the expected orbits. This would then result in the fluctuating behavior observed when using the reversibility method above.

With regards to this observation, the first method seems like a safer choice as it produced results with fewer fluctuations. The first method also has a shorter execution time than the reversibility method, and its precision can be adapted through the choice of baseline. Regarding the second method, it is also worth noting that it requires that the calculations can be executed in reverse, and therefore this method can only be applied in some cases.

Chapter 6

Conclusion

The goal of this thesis was to compare the accuracy of 32-bit Posit to that of 32-bit IEEE-754 when applied to N-body simulations. We found that for the formats' 32-bit counterparts, Posit's accuracy exceeded that of IEEE-754 when the values in the simulation were near 1. However, for values differing more than three orders of magnitudes from 1, IEEE-754 had higher accuracy. We also found that for values further from 1, Posit's accuracy was greatly diminished while IEEE-754's accuracy remained constant throughout.

Based on these results, we conclude that Posit is preferable when values are known to be near 1. If that is the case, Posit offers the potential for much more accurate calculations, which could be valuable to applications that are sensitive to rounding-errors. Nonetheless, if values are of unknown magnitudes, IEEE-754 has better accuracy overall and would lead to more predictable results. In general, IEEE-754 might therefore be the safer choice of the two formats.

References

Chien, Steven Wei Der, Ivy Bo Peng, and Stefano Markidis. 2019. "Posit NPB: Assessing the Precision Improvement in HPC Scientific Applications." *CoRR* abs/1907.05917. http://arxiv.org/abs/1907.05917.

De Blasio, Simone, and Fredrik Ekstedt Karpers. 2020. "Comparing the Precision in Matrix Multiplication Between Posits and IEEE 754 Floating-Points : Assessing Precision Improvement with Emerging Floating-Point Formats." TRITA-EECS-EX. KTH, School of Electrical Engineering; Computer Science (EECS); KTH, School of Electrical Engineering; Computer Science (EECS).

Dinechin, Florent de, Luc Forget, Jean-Michel Muller, and Yohann Uguen. 2019. "Posits: The Good, the Bad and the Ugly." In *Proceedings of the Conference for Next Generation Arithmetic 2019*. CoNGA'19. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3316279.3316285.

Dormand, Prince, J. R. 1978. "New Runge-Kutta Algorithms for Numerical Simulation in Dynamical Astronomy." *Celestial Mechanics* 18: 223–32.

Free Software Foundation. 2021. "The GNU Multiple Precision Arithmetic Library." 2021. https://gmplib.org.

Gustafson, John, and Isaac Yonemoto. 2017. "Beating Floating Point at Its Own Game: Posit Arithmetic." *Supercomputing Frontiers and Innovations* 4 (2). https://superfri.org/superfri/article/view/137.

Huang, Weizhang, and Benedict Leimkuhler. 1997. "The Adaptive Verlet Method." *SIAM Journal on Scientific Computing* 18 (1): 239–56. https://doi.org/10.1137/S1064827595284658.

Kahan, William. 1996. “IEEE Standard 754 for Binary Floating-Point Arithmetic.” *Lecture Notes on the Status of IEEE* 754 (94720-1776): 11.

Leong, Cerlane. 2021. “SoftPosit.” 2021. `https://gitlab.com/cerlane/SoftPosit`.

Vinayan, Suvina, and Anup Prof R. Nage. 2015. “Design of IEEE 754 Format 32 Bit Complex Floating Point Vedic Multiplier - a Review.” *Journal of Emerging Technologies and Innovative Research* 2: 3.

TRITA -EECS-EX-2021:432