DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2020

A comparative study of IEEE 754 32-bit Float and Posit 32-bit floating point format on precision.

Using numerical methods.

Johan Besseling
Anders Renström

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Abstract

Posit is a new way of representing floating points in computers. This thesis investigates the precision of the 32-bit Posit floating point format compared to the current standard 32-bit IEEE 754 Float format by conducting tests with numerical methods. Posit was chosen due to its promising results in previous work. The numerical analytical methods that where chosen was the Least Square Method, Gauss Newton Interpolation Method, Trapezoid Method and Newton Raphsons Method. Results from the tests show that Posit32 performs at least as high precision as IEEE 754 Float on computations larger than a range of 1 and above but tends to increase precision up to three significant figures when moving towards a range of 0 - 1.

Abstract

Posit är ett nytt sätt att representera flytande punkter i datorer. Den här avhandlingen undersöker precisionen i 32-bitars Posit-flytpunktsformat jämfört med nuvarande standard 32-bitars IEEE 754 Float-format genom att utföra test med numeriska metoder. Posit valdes på grund av sina lovande resultat i tidigare arbete. De numeriska analysmetoderna som valde var Minstakvadrat metoden, Gauss Newton Interpolation Metod, Trapezoid Metod och Newton Raphsons Metod. Resultaten från testerna visar att Posit32 utför minst lika hög precision som IEEE 754 Float på beräkningar som är större än ett intervall mellan 1 och högre men tenderar att öka precisionen upp till tre värdesiffror när beräkningarna rör sig mot ett intervall mellan 0 - 1.

Abbreviations

- IEEE 754 - Institute of Electrical and Electronics Engineers standard for Floating-Point Arithmetic
- Unum - Universal number
- HPC - High Performance Computing
- NaN - Not a Number

Acknowledgements

We would like to acknowledge our supervisor Stefano Markidis, as well the doctorands Steven Vei Der Chien and Artur Podobas, for all their help they have contributed throughout this study.

Authors

Johan Besseling <johanbes@kth.se>
Anders Renström <renstr@kth.se>
Information and Communication Technology
KTH Royal Institute of Technology

Place for Project

Stockholm, Sweden
KTH Royal Institute of Technology

Examiner

Pawel Herman KTH Royal Institute of Technology

Supervisor

Stefano Markidis
KTH Royal Institute of Technology

Contents

1 Introduction

Computers cannot read numbers like humans do. One way to represent real numbers is to let the computer use something called floating point arithmetic's. A floating point number is not infinitely precise due to a computers limited memory which means that a computer cannot represent infinitely large and small numbers. This causes a trade-off between range and precision. Currently the IEEE 754 floating point format is the industry standard on most computers today and has been used since it was developed in the mid-1980s. Recently new floating point formats has emerged, which can possibly represent a number with a higher precision with the same amount of bits as IEEE 754. One of these formats are the Type III Unum - Posit format developed by John L. Gustafson[5].

1.1 Interest and importance of subject

The reader might ask why this is an important subject, and why it matters if a computer cannot measure an immensely huge or small number correct. If a computer cannot make an accurate approximation of a number it can result in some serious consequences. In 1991 an United States missile battery failed to intercept an incoming missile. The failure was caused by an inaccurate measurement of time by the battery's computer system. The computer had a limit on 24-bit precision and on the 24th bit the time was truncated by $1/10$ causing a rounding error. When the hostile missile were approaching, the computer systems miscalculations caused the missile battery to miss the incoming missile, which killed 28 and injured 100 people[12]. Another common error with precision is the problem with cancellation. It happens when nearly two identical numbers are subtracted and might illustrate the importance of precision better to the reader. If we have the decimal number $x = 0.1234567891234567890$. On a machine with 10 floating point precision the number would be represented as $y = 0.123456789$. If we do the operation $x - y$ the real answer would be $x - y = 0.000000001234567890$, however the machine cannot represent more than 10 digits so when we do the same operation on the machine we get the results $x - y = 0.0000000001$, meaning a lot of data is lost. Therefore, ways of improving the accuracy of floating points is important and Posit could be a contender. Therefore, a lot of testing has to be

done in order to see how these formats compare to each other. This problem has been investigated before by running benchmark tests between the two formats[2] and the results are promising, due to an increase in decimal precision by 0.6 - 1.4. However, calls for better hardware support to even further increase performance. Another study [3] says that Posit beats IEEE 754 at its own game. Stating that Posits could give a better result with the same amount of bits and even reduce energy consumption due to lesser rounding operations. An example of this can be seen from one of the John L Gustafsons seminars, where he showcases a computational evaluation equation[6].

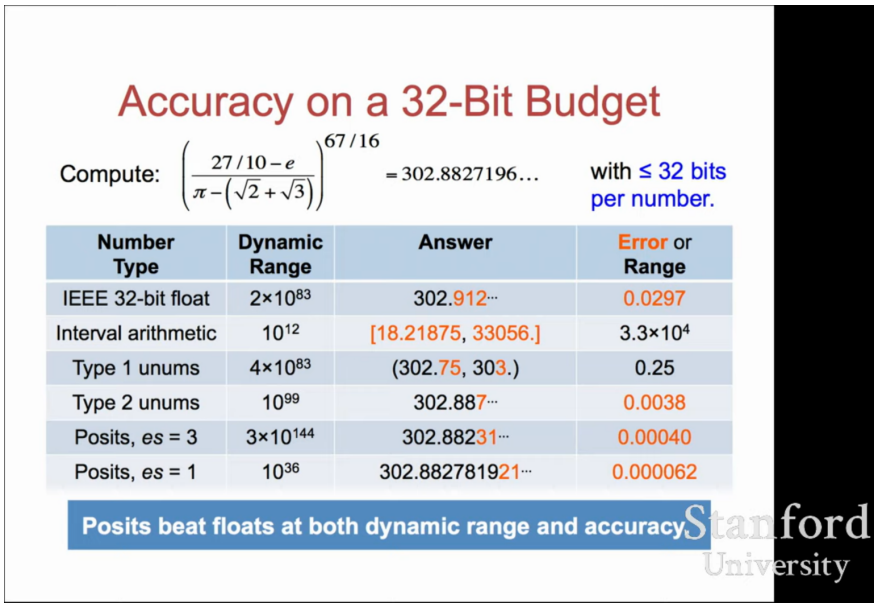

Figure 1.1: *Timestamp: 45:59*

As the above figure states the calculations using Posit as format is more precise in this example compared to the conventional IEEE 754 format. The 'es' value in the picture is an exponent used in a formula to define the precision of a Posit. Even though Posit showcases promising results the Posit format is still in early testing cases, it is still unsure if the precision is better compared to the IEEE 754 format.

1.2 Purpose and research question

Increasing computer performance is something that has always been in high demand. The purpose of this paper is to investigate the effect in floating point

precision in high performance computing using emerging floating point 32-bit Posit32 Posit format compared to the conventional 32-bit IEEE 754 Float format. The study will look more closely at the following:

- How will Posit32 affect the floating point precision compared to the 32-bit IEEE 754 Float format?

1.3 Scope

This study is limited to only compare two floating point formats and one set bit size. There are other new floating point formats next to Posit but because of the limited time, a choice had to be made on which format to evaluate compared to IEEE 754 format. Posit was chosen due to its promising results in previous work and that it supports the C++ programming language. The tests are limited to algorithms using matrix multiplication simply because it is a common operation used in high performance computing and is generally easy to implement. The bit-size chosen for Posit is Posit32 as it is currently the largest implemented size in the software library used in this study. To be compared with the Posit32 is the IEEE 754 Float as it is the corresponding size in the IEEE 754 format.

1.4 Goal

The goal for this project is to get a good understanding of the structure of the Posit and IEEE 754 floating point format, as well as comparing these formats with different benchmark tests to get a better understanding of how these formats compare to each other in terms of precision. Finally, the results and conclusions that can be drawn from these tests, can help further motivate future research regarding Posit replacing the current standard floating point format IEEE 754 in the future.

1.5 Sustainability and energy consumption

Using Posit as a floating point format could be beneficial for a sustainable perspective within energy consumption. For an arbitrarily application which requires a precision which is one or two significant figures higher the limit of

IEEE 754 32-bit Float one would have to increase the precision to a IEEE 754 64-bit Double, resulting in twice the memory use, while if the corresponding 32-bit Posit32 could perform the required precision, no extra memory would have to be used. For example a load or store from a processor register with 64 bits requires 6 pJ compared to loading or storing from the physical memory which consumes 4200 pJ, an increasing factor of 700, which is a huge number when taking into account the number of operations done in a modern computer today[5]. Because of the energy differences is so high of the load and store between the register and physical memory there is an interest of optimizing the memory management and reducing energy consumption.

1.6 Outline

In the following chapters, we will go into further detail about what has already been researched in the field, the structure of the different floating point formats, the method we used in order to solve the problem, the results of that method and finally draw conclusions based on our results.

2 Background

In this chapter information about the following topics will be covered:

- History behind the floating point formats.
- Structure of floating point IEEE 754 format and the Posit floating point format.
- Previous research work revolving the Posit floating point format and it's importance to the project.
- Algorithmic methods within HPC and numerical analysis.

2.1 History

Perhaps the first template of representing a real number in a machine came in the early 1900s by Leonardo Torres y Quevedo. The reader can find this paper "Essays on Automatics"(1914)[13]. Torres y Quevedo constructed a machine called "Electromechanical arithmometer" which was capable, at the time, of accepting advanced statements such as $365 * 256$ and printing $with = 93440$ on the same line. In his paper he also described a method to store real numbers. According to the method the number will be stored in exponential format as $n * 10^m$. Upon this format a machine is required to follow three rules.

- n will always be the same number of digits (six for example)
- The first digit of n will be the order of tenths, the second of hundredths, etc.
- One will write each quantity in the form n;m

The reader might recognize this format because of its similarity with most programming languages way of representing a number. If we for example have 5;4 in Torres format we can easily replace ";" with an "e" and get 5e4 which is the same as $5 * 10^4$. However, the person who is in general given credit for implementing the first floating point number is Konrad Zuse in his Z1 and Z3 computers[14]. Z1 and Z3 were constructed around 1940 and the computers used binary floating-point numbers with one bit for the sign, 14 bits for the significant, and a signed exponent of 7 bits. Zuse claimed the computers supported a number

range between $2^{-63} - 2^{62}$. Numbers was, like in today's modern computers, entered in decimal format and converted into binary format, while output did the opposite. In the coming years a lot of different floating-point formats were used, and it was not until 1985 IEEE released a conventional standard for floating-point arithmetic. Since then the IEEE standard formats has been dominating the computer industry[10].

The Posit format is a young floating point format and derives from its predecessors Type I and II Unum which was introduced by John L. Gustafson [5]. Type I Unum was a superset of IEEE 754 and supported compatibility and had a more efficient bit representation. However, it had problems with values having dynamical sizes and inherited a lot of the disadvantages with IEEE 754. Type II Unum was an overhaul and supported reciprocals of the arithmetic operations (+ - x /) and also supported decimal representations. The disadvantages was that the format was not precise in certain operations like table look-up which was limited to 20 bits or less. In February 2017 Type III Unum also called Posits were introduced. According to John L. Gustafson[4] Posits are possible to directly convert to IEEE 754 floats, it supports hardware implementations, it operates faster, the results are more accurate and is more cost efficient than the IEEE 754 in terms of energy consumption. Posits could essentially act as an drop-in replacement for the IEEE 754.

2.2 Floating Point

In this section we will describe in more detail what an IEEE 754 and Posit floating point number is and how it is constructed in terms of usage of the bit structure depending on the number of bits.

2.2.1 IEEE 754

In today's IEEE standard[7] there are currently 4 different basic binary floating point formats with 16, 32, 64 and 128 bits respectively. The formats are represented in the same way in the following structure.

$$(-1)^{sign} * b^{e} * significand$$

The interval of real numbers a format can represent is determined by the following

integer parameters

- $b = 2$ since it is a binary format
- p = the number of digits in the significand (precision)
- emax = the maximum exponent e
- emin = the minimum exponent e
- emin shall be 1−emax for all formats.
- Signed zero and non-zero floating-point numbers of the form $(-1)^s * b^e * m$, where
  - s is 0 or 1
  - e is any integer $emin \leq e \leq emax$.
  - m is a number represented by a digit string of the form $d_0 * d_1 d_2 ... d_{p-1}$ where d_i is an integer digit $0 \leq d_i \leq b$ (therefore $0 \leq m \leq b$)
- Two infinities, $+\infty - \infty$
- Two NaNs, qNaN(quiet) and sNaN(signaling)

An encoding in this format will create a bit string which can be read by the computer. The infinity signs mean that each format supports a subset of the real numbers including the negative and positive infinity which is used if an encoding leads to a bit string mapping outside the real number subset.

NaN

For all kinds of floating point operations there should be support for two different NaN. Signalling NaNs are used when a variable is not yet initialized and is generated if an operation is invalid. Quiet NaNs is used by the programmer to interpret invalid or undefined results.

The basic binary formats is represented in the following table.

Binary format (b=2)			
parameter	**binary32**	**binary64**	**binary128**
p,digits	24	53	113
emax	+127	+1023	+16383
bias, $E-e$	127	1023	16383

Table 2.1: Basic binary formats structure

Binary format encoding

For each floating point in respective format described above there is a unique interpretation of the floating point into a binary format. A binary representation of a floating point value consists of k bits divided into three fields.

- 1-bit sign S
- w-bit biased exponent E = e + bias
- ($t = p-1$-bit trailing significand field digit string $T = d_1 d_2 ... d_{p-1}$; the leading bit of the significand,d_0, is implicitly encoded in the biased exponent E

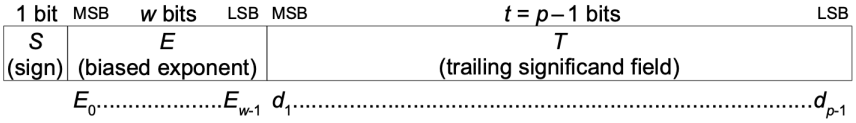

Figure 2.1: Binary fields of a IEEE 754 floating point

Examples

In order to convert a binary machine code value into a float we simply use the instructions described above and interpret the binary number and insert the values into our formula

$$binary = (-1)^S * 2^{E-bias} * 1.significand$$

32-Bit

Smallest positive normal number

$0\ 00000001\ 00000000000000000000000_2 = 2^{-126} \approx 1.175494350810^{-38}$

Largest normal number

$0\ 11111110\ 11111111111111111111111_2 = 2^{127}(2-2^{-23}) \approx 3.402823466410^{38}$

One

$0\ 01111111\ 00000000000000000000000_2 = (-1)^0 2^{127-127} = 1$

binary64(Double) and binary128(Long Double) bit size follows the same pattern but with a lot more numbers involved.

2.2.2 Posit

The Posit standard definition uses a different approach to the construction of the floating point number compared to IEEE 754 [8]. Currently there are four Posit format ranging from 8-, 16-, 32- and 64-bit precision and a special format called Quire. Each format is defined by a set of properties.

property	**posit8**	**posit16**	**posit32**	**posit64**
Max significand bits	6	13	28	59
Max exponent bits, `es`	0	1	2	3
`minpos`	$2^{-6} \approx 1.5 \times 10^{-2}$	$2^{-28} \approx 3.7 \times 10^{-9}$	$2^{-120} \approx 7.5 \times 10^{-37}$	$2^{-496} \approx 4.9 \times 10^{-150}$
`maxpos`	$2^{6} \approx 6.4 \times 10^{1}$	$2^{28} \approx 2.7 \times 10^{8}$	$2^{120} \approx 1.3 \times 10^{36}$	$2^{496} \approx 2.0 \times 10^{149}$
`pintmax`	8	256	2^{22}	2^{52}
quire bits	32	128	512	2048
Exact sum quire limit	32767	$2^{43}-1$	$2^{151}-1$	$2^{559}-1$
Exact dot product quire limit	127	32767	$2^{31}-1$	$2^{63}-1$

Figure 2.2: Posit format properties [8]

The Posit bit-string consists of four bit-fields called Sign, Regime, Exponent and Fraction.

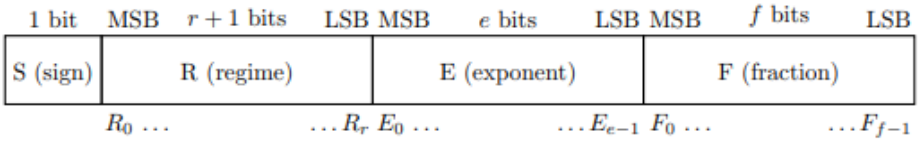

Figure 2.3: Binary fields of a Posit floating point

For each Posit there is a unique encoding between the bit-string and the floating point format. The encoding is given the following parameters combined with the chosen Posit format in the properties table above.

- Sign bit S, which defines if it is a negative or positive value
  - S is either 1 or 0

- Regime R bits includes all bits next to sign until and including one complement bit terminates the regime.
  - $R = R_0 == 1 \; ? \; r-1 \; : \; -r$
- Exponent E bits is a maximum of the number of es bits depending on which format is used and the properties of the regime. Potentially the regime could fill up the entire bit space, leaving no room for E or F bits.
- Fraction F bits is defined by the number of fraction bits and is terminated when the rest of the bits are used.
  - F includes a hidden bit like the significand for IEEE 754 which is always 1. If F is non-zero it is divided by 2^f

An example[3] gives a more concrete way of explaining how this encoding takes place. Define $useed^R$ where $useed = 2^{2^{es}}$. Depending on format and the parameter values given above, a Posit p, will be represented as

$$x = \begin{cases} 0, & p = 0 \\ \pm\infty, & p = -2^{n-1} \\ (-1)^S * useed^R * 2^E * F, & otherwise \end{cases}$$

From choosing a 16 bit long bit-string 0 0001 1 0111011101. Since it is a 16-bit long bit-string we shall set es = 1 according to our table above. We can see that the sign bit is 0 which means the value is positive and $S = 0$. The regime R has to be -3 since R_0 is 0 and $r = 3$. With an es value of 1 we can only leave a single bit for E, assigning $E = 1$. The rest of the bits are left to the fraction F. Evaluating $0111011101_2 = 477_{10}$ results in $F = 1 + 477/2^{10} = 1.4658203125$. The addition of 1 is to include the hidden bit which is always 1. Inserting these values into our formula with $useed = 2^{2^1}$ we get $(-1)^0 * (2^{2^1})^{-3} * 2^1 * 1.4658203125 \approx 4.58068 * 10^{-2}$

2.3 Related Work

John L. Gustafson and Isaac Yonemoto(2017)[3]. Compares the IEEE 754 format to the Posit floating point format using several tests such as the Classic Linpack benchmark, Goldberg's thin triangle problem, Expression evaluation

and standard arithmetic calculations such as exponential, logarithmic, reciprocal, addition subtraction and multiplication. It was found that the Posit format did better with the accuracy on every test except for the multiplication test in exact precision, where the IEEE 754 format had an exact precision accuracy of 22.3% while the Posit floating point format only had 18.0% precision accuracy. They conclude after the following tests that the Posit format has a higher accuracy, larger dynamic range and better closure. And if Posit gets hardware support they estimate that the implementations will give us the equivalent of one or two turns of the Moore's law improvement[11], without any need to reduce transistor size, or cost.

S. W. D. Chien et al.(2019) [2][1] work focused on comparing precision optimization in HPC applications with the Posit 32-bit floating point format compared to the IEEE 754 32-bit format floating point format using widely-adopted benchmark suite NAS Parallel Benchmark (NPB) . Using the NPB suite they managed to reach a higher precision, ranging from 0.6 to 1.4 decimal digits better accuracy with the Posit 32-bit format compared to the IEEE 754 Float baseline. However, due to the limited hardware implementation of the Posit format made the Posit NPB suite quantified 4x-19x overhead that of the IEEE format.

Wan, Sizhen et al.(2018) [16] focuses on recurrent neural networks speech recognition and the difference between a fixed-point-based hardware system, a float system format and a Posit-based hardware system with a limited space. In the paper they concluded that the Posit number system is well more suited for speech recognition inference at 8-bit precision and that the Posit-based hardware would be less expensive in terms of power and area compared to float and fixed-point.

Klöwer et al, (2019)[9] work focused on trying to minimize the bit size of the weather forecast data from 32-bit to 16-bit size. In the report the Float format, the half precision floats (IEEE 754) format and the 16-bit Posit format to was compared see which format had the smallest rounding error. Using standard forecast model the shallow water model, they concluded that a 16-bit Posit with 1 or 2 exponent bits had more accuracy than the half precision floats. With these results they recommend that the 2 exponent bit Posit format should be the

standard Posit format for weather and climate models.

2.4 Numerical algorithmic methods

This section describes suitable methods to conduct comparisons between floating point formats. The methods consist of HPC applications such as operations with matrices but also elementary methods derived from numerical analysis. The common factor of the tests is that they are all used in practical applications within computer science such as linear algebra, approximate solutions to equations and integrals and there is an interest of having an high amount of precision in order to achieve a satisfactory result.

2.4.1 Least Square Method

Least Square Method[15] is derived from the problem of having an over determined system of linear equations and is common when one want to fit data to models.

$$m + b = 2$$
$$m - b = 1$$
$$m + b = 3$$

For this system there is no solution, so the goal is to approximate a solution to find the best fitting vector by using the matrix form of the system.

$$A\vec{x} = \vec{y}$$

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$$

In order to compute this the transpose A^T is used to transform the matrix A and vector y into dimensions which will give an approximated solution to the

system.

$$A^T A\vec{x} = A^T\vec{y}$$

$$A^T A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \quad A^T\vec{y} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \end{bmatrix}$$

Solving this new system yields the approximated solution $m = \frac{7}{4},\ b = \frac{3}{4}$ to the data points.

2.4.2 Gauss Newton Interpolation method

Gauss Newton Interpolation method[15] is a technique used to replace a complex function, to expensive to compute, or if it does not exist a function with a given set of n data points. With interpolation we create a polynomial that approximates the original function with the data points. Given the data points $\vec{x} = [x_0, x_1, ..., x_{n-1}], y = [y_0, y_1, ..., y_{n-1}]$ and a polynomial function of degree at most $n-1$.

$$p_{n-1}(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + ... + c_{n-1}(x - x_0) * ... * (x - x_{n-2})$$

We wish to construct the vector $c = [c_0, c_1, c_2, ..., c_{n-1}]$ populated by the coefficients in the polynomial $p(x)_{n-1}$, by doing the following operation.

$$A\vec{c} = \vec{y}$$

$$c = A^{-1}\vec{y}$$

$$\begin{bmatrix} 1 & 0 & 0 & ... & 0 \\ 1 & (x_1 - x_0) & 0 & ... & 0 \\ 1 & (x_2 - x_0) & (x_2 - x_0)(x_2 - x_1) & ... & 0 \\ . & & & & . \\ . & . & & & . \\ . & & . & & . \\ 1 & (x_{n-1} - x_0) & (x_{n-1} - x_0)(x_{x-1} - x_1) & ... & (x_{n-1} - x_1) * ... * (x_{n-0} - x_{n-2}) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ . \\ . \\ . \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ . \\ . \\ . \\ y_{n-1} \end{bmatrix}$$

Solving this system of equations yields the coefficients required to construct $p(x)_{n-1}$.

2.4.3 Trapezoid method

A common numerical problem is to approximate integrals[15] if it is to complicated or even impossible to calculate the primitive form of a function. An example is the function $f(x) = e^{x^2}$.

For a given function $f(x)$ the integral to be evaluated is $I = \int_a^b f(x)dx$ where a and b is the interval to be integrated. Depending on $h = \frac{b-a}{n}$ where n is the number of partitions and h is the length of one partition the integral is approximated by $I \approx h(\frac{f(a)}{2} + f(a+h) + f(a+2h) + ... + f(a+(n-1)) + \frac{f(b)}{2}$ which can be refactored into the Trapezoid formula.

$$T(h) = h(\sum_{i=1}^{n+1} f_i - \frac{f_1 + f_{n+1}}{2})$$

This will yield an approximated solution to the integral with a precision mostly depending on the length of the sub intervals h but also on the precision of the chosen floating point format used to represent the numbers.

2.4.4 Newton-Raphson method

This method[15] is used to approximate a solution of an equation with one variable or a system of nonlinear equations with multiple variables. Given a single variable polynomial $f(x)$ and a starting guess x_0 the method wants to find the approximated solution at $f(x) = 0$. The iterative method for a single variable is created by expanding and rewriting the definition of the derivative $f'(x)$.

$$f'(x_0) = \frac{f(x) - f(x_0)}{x - x_0}$$
$$(x - x_0)f'(x_0) = f(x) - f(x_0)$$
$$x - x_0 = \frac{0 - f(x_0)}{f'(x_0)}$$
$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

The multivariable case is an expansion of the single variable method and includes the vector $\mathbf{F} = [f_1(x_1,...,x_n), f_2(x_1,...,x_n),...,f_n(x_1,...,x_n)]$, a jacobian matrix **J** and similar to the single variable method a starting guess X_0 = $[x_{1_0}, x_{2_0},...,x_{n_0}]$ is required. The jacobian is a matrix populated by the partial derivates with respect to each variable in respective function of **F**.

$$\mathbf{J} = \begin{bmatrix} f'_{1_{x_1}} & f'_{1_{x_2}} & ... & f'_{1_{x_n}} \\ f'_{2_{x_1}} & . & . & . \\ . & . & . & . \\ . & . & & . \\ . & & . & \\ f'_{n_{x_1}} & f'_{n_{x_2}} & ... & f'_{n_{x_n}} \end{bmatrix}$$

The first iteration of the method is initiated by subtracting X_0 with $\mathbf{J}(X_0)^{-1}$ multiplied by $\mathbf{F}(X_0)$.

$$X_1 = X_0 - \mathbf{J}(X_0)^{-1}\mathbf{F}(X_0)$$

By running several iterations of both methods, x_i and X_i will converge to the sought approximated solution depending on the amount of iterations, chosen for the approximation. With a higher amount of iterations a better approximation is generated but with a higher cost in computing power.

2.4.5 Hypothesis

From looking at other related work to the comparisons, we assume that our result will be similar, meaning that the 32-bit Posit format will have a better or similar precision compared to the 32-bit IEEE 754 format when it comes to computations on figures in the range of 0 and 1. We believe that the IEEE 754 Float format will perform better the higher the values reaches, except for when it hits the overflow limit.

3 Method

In this section details are described in regard to sources used, tools to construct algorithms and restrictions used to conduct the comparison between IEEE 754 and Posit floating point format, such as computer hardware, software implementation language and compiler details in order for the reader to be able to recreate the same methodology and verify the results. Also we motivate the tests chosen to make the comparisons and how to quantify the results given by a particular test.

3.1 Literature study

To gain a solid understanding of the subject a literature study were conducted. The sources were found through Google Scholar and KTH Library. The sources are books, articles and papers written by authors who is working with HPC and conducts research on the subject. The content of the sources are floating point format history, theoretical descriptions and algorithmic testing. From these sources the literature study begins with a brief historical perspective on floating point numbers and continues with the definition of the conventional standard IEEE 754 floating point format and the emerging Posit format. The literature study also found several methods to compare different formats by conducting algorithmic tests.

3.2 Construction of tests

In this section we describe in detail the tools and restrictions used to conduct the comparison between IEEE 754 and Posit floating point format, such as computer hardware, software implementation language and compiler details. Also, motivate the tests chosen to make the comparisons and how to quantify the results given by a particular test.

The general method was to implement a test suite and run tests on both IEEE 754 and Posit floating point format. In order to do so it was necessary to obtain and learn an established library. A promising library was NAS-Posit-benchmark[1]

[1]https://github.com/steven-chien/NAS-Posit-benchmark

which was used before in[2]. The library implements several different bit sizes which we could use to implement our own tests and investigate our problem. It was also required to find a Posit library since it is not a pre-defined type in any language at the moment. The Posit library used was the Cerlane SoftPosit[2]. Because of the amount of special matrix operations in the tests, such as coherent, determinant and inverse of a matrix, a complete template matrix class[3] was implemented to support the operations. The class was not optimized for any particular hardware or compiler, so it is possible that results was tampered in terms of speed and memory used. Each test was first done either with a reference bit-size that acted as the correct answer to the problem or a manual computation by hand if possible, in order to get complete result. Then the IEEE 754 and the Posit format was used to the same tests. The results were then compared to the assumed correct answer to evaluate the difference in precision. Each test described in this section showcases what comparisons will be used and what result value is looked at. The complete implementation of the tests can be found in the footnote of this section. For every test input data was also described and for some tests a visual plot was used to make the results clearer. The following bit-sizes that was used for the tests:

- Long double (binary128) as reference and regarded as the correct answer
- Float (binary32)
- Posit32

3.3 Hardware

- Operative System: Windows 10 Home
- Windows Version: 1903
- System type: 64-bit Operative System, x64-based processor
- Processor: Intel(R) Core(TM) i5-8300H CPU @2.30GHz
- DRAM : 8.00 GB

[2]https://gitlab.com/cerlane/SoftPosit
[3]https://gits-15.sys.kth.se/johanbes/KEXJobb

3.4 Implementation language

To implement the tests and the matrix class the decision fell on C++ due to speed, properties of object-oriented language and that the SoftPosit library offered an C++ extension. To visualize the results gnuplot was used.

3.5 Compiler

As for the compiler, the GNU C++ standard version 11.0 and the g++ MinGw version 9.2.0 was used for implementing the tests. In order to get a clear comparison between the Posit32 format and the IEEE 754 Float format, without having any hidden optimization in precision. The following flags where used:

-mfpmath=sse :

Uses scalar floating point instructions present in the SSE instruction set instead of the i386 set. Which avoids utilization of 80-bit temporaries.

-msse2 :

Enables SSE extensions.

-ffp-contract=off :

Disables floating point expression contraction, which turns off the fused multiple add option for floating point computations.

-ffloat-store :

Sets the compiler to not store floating points variables in registers. This option prevents undesirable excess precision, meaning that it keeps more precision than it is supposed to have.

3.6 Least Square Method

Conduct Least Square Method tests where a set of data points are used to construct a linear function ($y = mx + b$) as a model for the data. The results of the test will be measured by evaluating the residual $r = b - Ax$ and then calculate the norm $||r||_2 = \sqrt{r_1^2 + ... + r_m^2}$. The final result was to compare the norm with the norm of the reference size. The closer to 0 the better the approximation.

3.7 Gauss Newton Interpolation Method

Conduct Gauss Newton Interpolation to approximate a polynomial with a set of n data points. To measure the precision, error vectors was created between the reference format and the format that was tested by taking the absolute value between each function value. Then by calculating plotting the error vector to function summarized error was given for each format. The closer to zero the better approximation.

3.8 Trapezoid Method

Conduct the Trapezoid method to approximate an integral between an interval with a given function f(x) and a set number of partitions. The precision is measured by calculating the absolute error between each function value and then calculate the absolute difference of the correct answer for each format and each set number of partitions. The higher the amount of in order correct numbers to the answer, the better the result.

3.9 Newton-Raphson Method

Conduct tests with Newton-Raphson single and multi-variable method to approximate a solution to a linear single variable equation or a system of multi-variable equations. For both single and multi-variable form the result is interpreted by looking at the correction of the approximated solution in each iteration.

4 Results

This section presents the results of the tests. To study the effects of using Posit32 as floating point format the tests were executed as Float, Long Double and Posit32. The scope of the tests was limited to two different sets of input and output data. One for numbers between $0 - 1$ and one for numbers greater than 1. The reason for this is to see if the formats precision behaves differently depending on the properties of the numbers being represented as a floating point number. The data displayed in table 4.1 and 4.2 is the correction of the current approximated solution. In figure 4.1 the linear model is displayed for all three formats along with the data. Figure 4.2 - 4.3 displays a zoomed in view of figure 4.1 on different intervals to demonstrate the difference in precision while the table 4.3 and table 4.4 showcases the Residual Summary for the three values and its difference to the long double format. Table 4.5 shows the error norm of the residual between the reference format Long Double compared to Float and Posit32. Figure 4.4 is the approximated polynomial function value error of Float and Posit32 compared to the Long Double polynomial. Figure 4.5 - 4.6 demonstrates a zoomed in view on the polynomials to demonstrate the difference in precision. Figure 4.7 - 4.10 demonstrates the number of correct digits between the different formats, approximating the integral of a function, on a given interval depending on the amount of partitions. Table 4.6 and 4.7 is the correction for each iteration approximating a solution for the non-linear equations in figure 4.11 - 4.12.

4.1 Newton-Raphson Single Variable

Two tests conducted with the Newton-Raphson method to approximate a solution to the equations

$$(x-100)^2-\pi^2=0$$
$$(x-3)^2-\pi^2=0$$

The solutions for these equations are $x_1=100-\pi$, $x_2=100+\pi$ and $x_3=3-\pi$,$x_4=3+pi$. The sought solution will be x_1 which was arbitrarily and x_3 to contain the solution within the domain $0-1$ to study if the precision behaves differently.

Corr/It	Float	Posit32
It	x	x
1	0.0418717339634	0.0418716855347
2	0.0002790540456	0.0002790261059
3	0.0000001043081	0.0000000093132
4	0.0000000000000	0.0000000000000

Table 4.1: Float compared to Posit32 on finding a solution to $(x-3)^2-\pi^2$.

Corr/It	Float	Posit32
It	x	x
1	0.0984573364257	0.0984573364257
2	0.0015411376953	0.0015420913696
3	0.0000000000000	0.0000000000000

Table 4.2: Float compared to Posit32 on finding a solution to $(x-100)^2-\pi^2$.

4.2 Least Square Method

For this test the set of points used was

$\vec{x} = [2,3,5,7,9,10,13,15,20,21,23,25,29,30,11,12,8,1,4,16,19,22]$

$\vec{y} = [4,5,7,10,15,17,19,22,26,28,30,32,35,37,14,16,11,3,4,21,25,26]$.

MSM Results		Posit32	Float	Long Double
Variables	m	1.9786309525371	1.9786243438721	1.9786306762266
	b	1.1917053610086	1.1917051076889	1.1917053282722
Residual Summary		6.4559805671679	6.4559799935467	6.4559799935467

Table 4.3: Least Square Method Results

Residual Difference to Long Double Residual Summary	
Float	Posit32
0.0000005736212	0.0000000304338

Table 4.4: Residual summary difference for Least Square Method compared to Long Double format summary residual

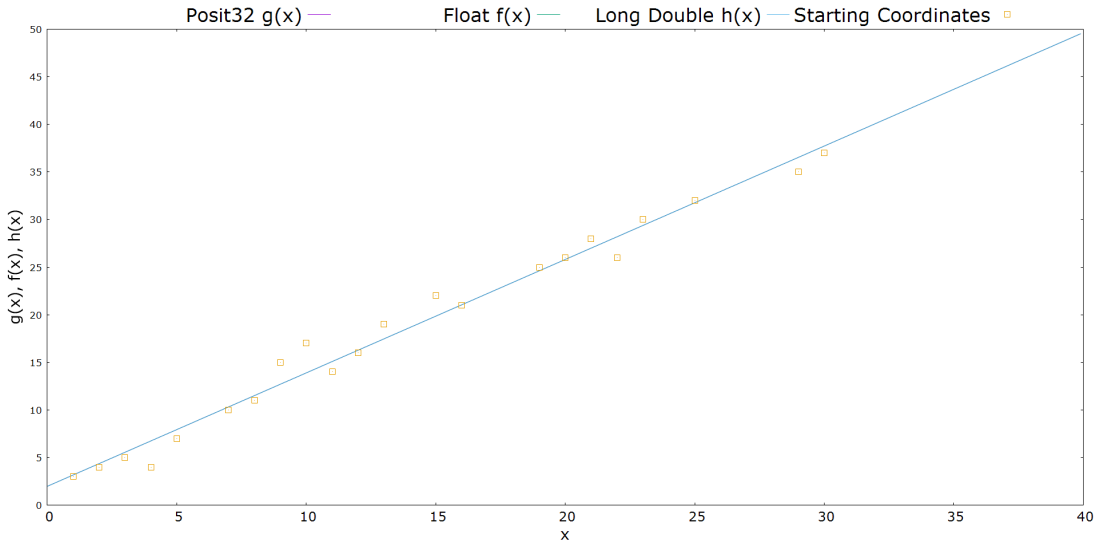

Figure 4.1: Linear model fitted to data.

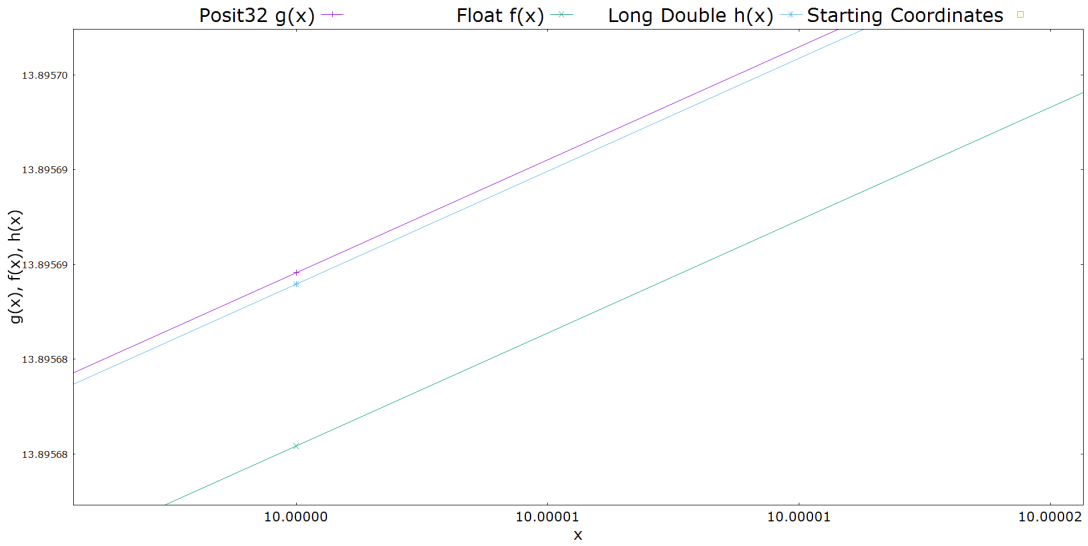

Figure 4.2: Large number interval.

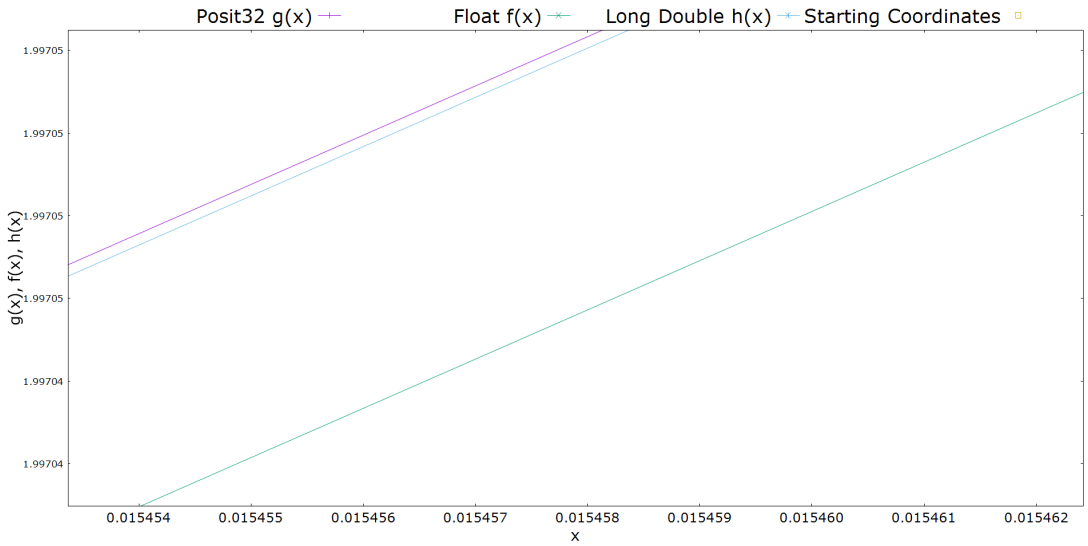

Figure 4.3: Small number interval.

4.3 Gauss Newton Interpolation

Input for this test is $\vec{x} = [1,2,3]$ and $\vec{y} = [0.3,0.5,0.7123]$ which was used to approximate a second degree polynomial with Newtons interpolations method. The test was done for Float, Posit32 and Long Double with the Long Double regarded as the correct approximation.

Float	Posit32
0.711767	0.055157

Table 4.5: Residual norm for the Gauss Newton Interpolation compared to the Long Double solution

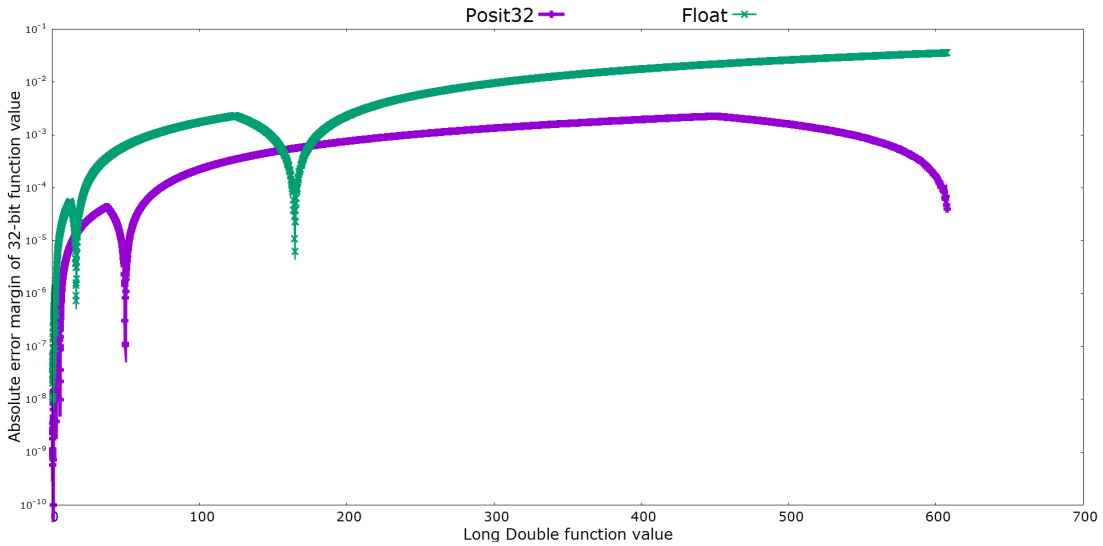

Figure 4.4: Error comparison between polynomial function values.

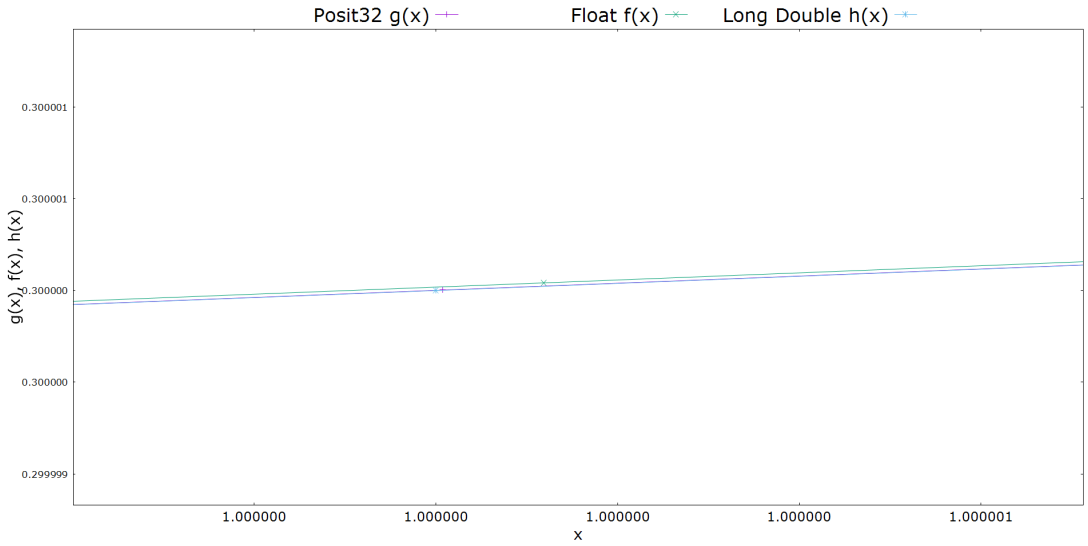

Figure 4.5: Plot of approximated functions on an interval of small numbers.

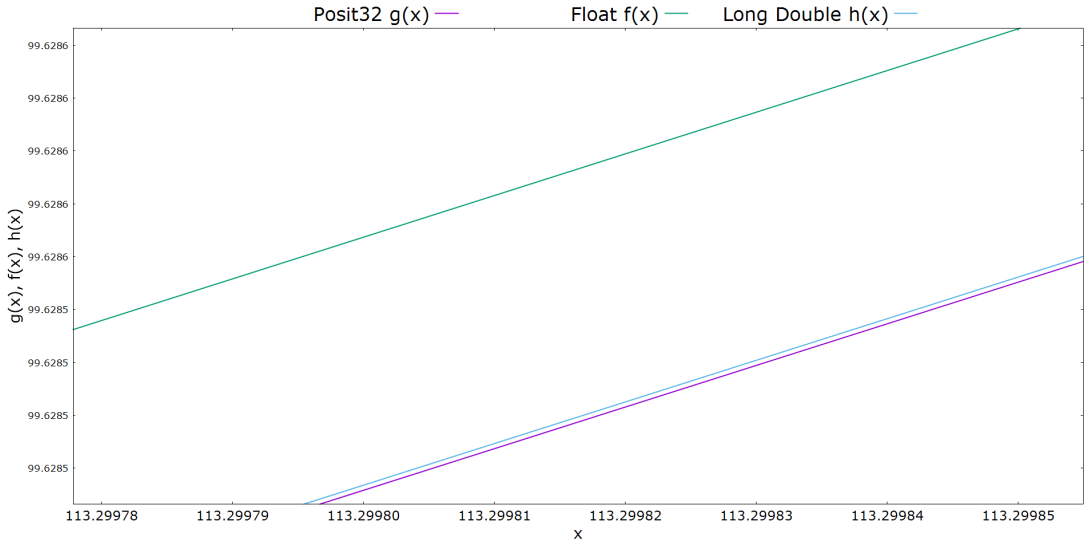

Figure 4.6: Plot of approximated functions on an interval of large numbers.

4.4 Trapezoid method

By conducting the trapezoid method the following approximate integral has been generated.

$$Equation: f(x) = \int_0^2 (x^2 + 4x + 3)dx \qquad Solution: f(x) = \frac{50}{3}$$

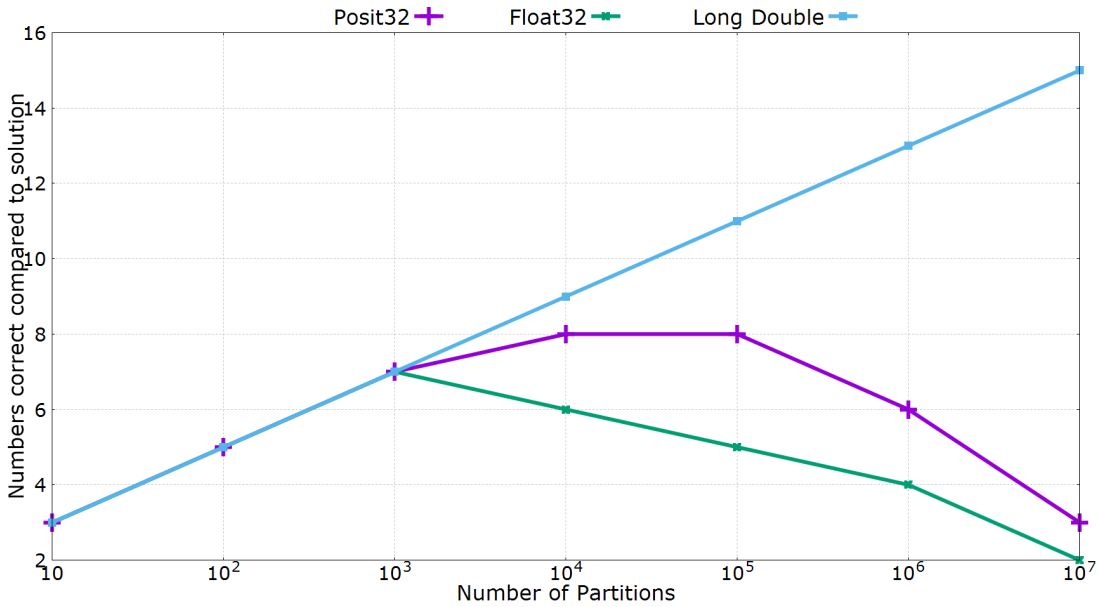

Figure 4.7: Number of correct numbers depending on the number of partitions (**n**) used for the above equation.

$$Equation: f(x) = \int_0^{1024} \frac{x^2 + 4x + 3}{10000}dx \qquad Solution: f(x) = \frac{67502656}{1875}$$

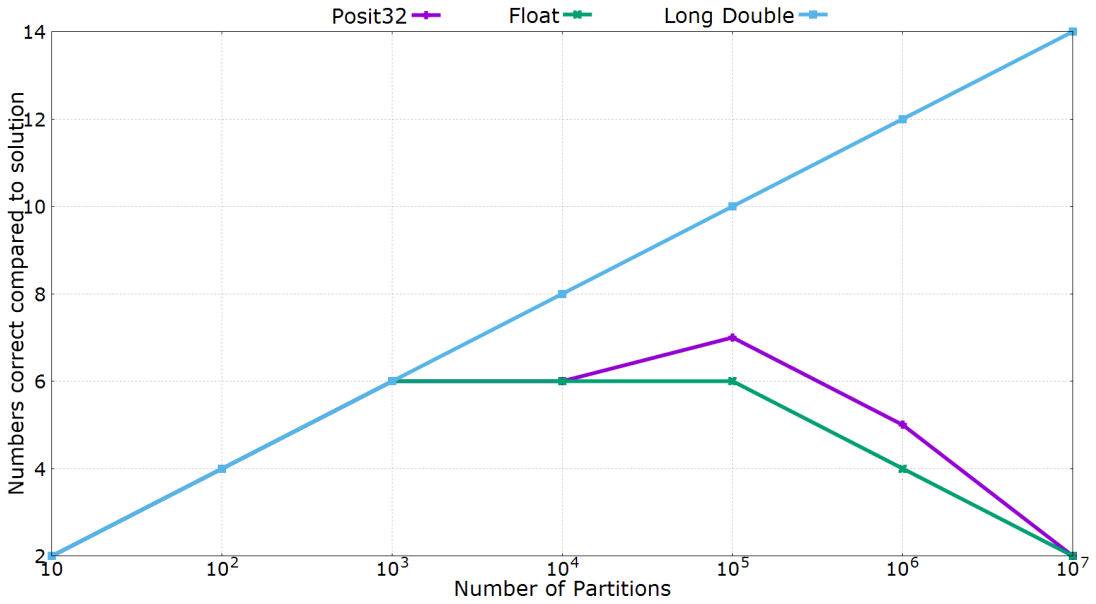

Figure 4.8: Number of correct numbers depending on the number of partitions (**n**) used for the above equation.

$$Equation: f(x) = \int_0^2 \sqrt{2^2 + x^2}\,dx \qquad Solution: f(x) = \pi$$

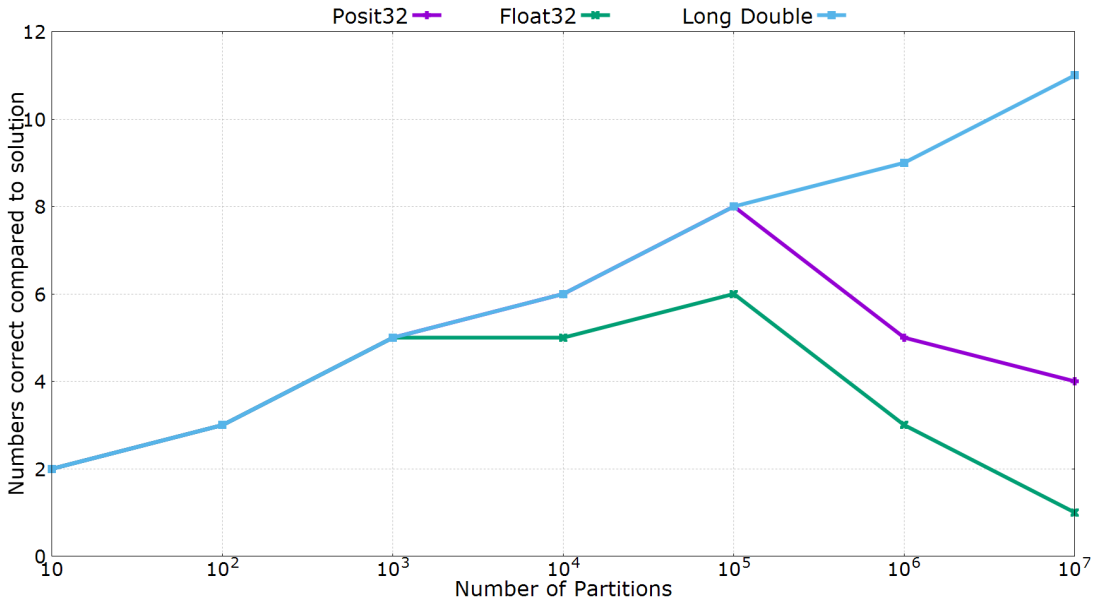

Figure 4.9: Number of correct numbers depending on the number of partitions (**n**) used for the above equation.

$$Equation : f(x) = \int_0^{1024} \sqrt{1024^2 + x^2}\,dx \qquad Solution : f(x) = 262144\pi$$

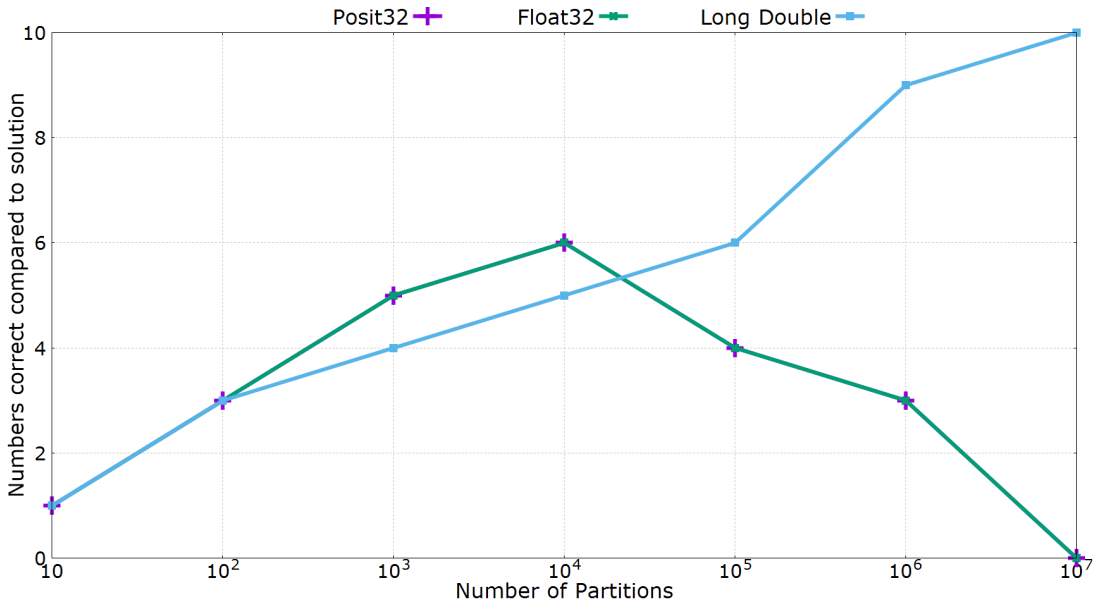

Figure 4.10: Number of correct numbers depending on the number of partitions (**n**) used for the above equation.

4.5 Newton-Raphson Multi-Variable

By using Newton-Raphson Multi-Variable method two tests were concluded. The first one with three circles within a domain on numbers larger than 1.

$$f(x,y) = (x-500)^2 + (y-450)^2 - 20000$$
$$g(x,y) = (x-480)^2 + (y-200)^2 - 20000$$
$$h(x,y) = (x-250)^2 + (y-400)^2 - 30000$$

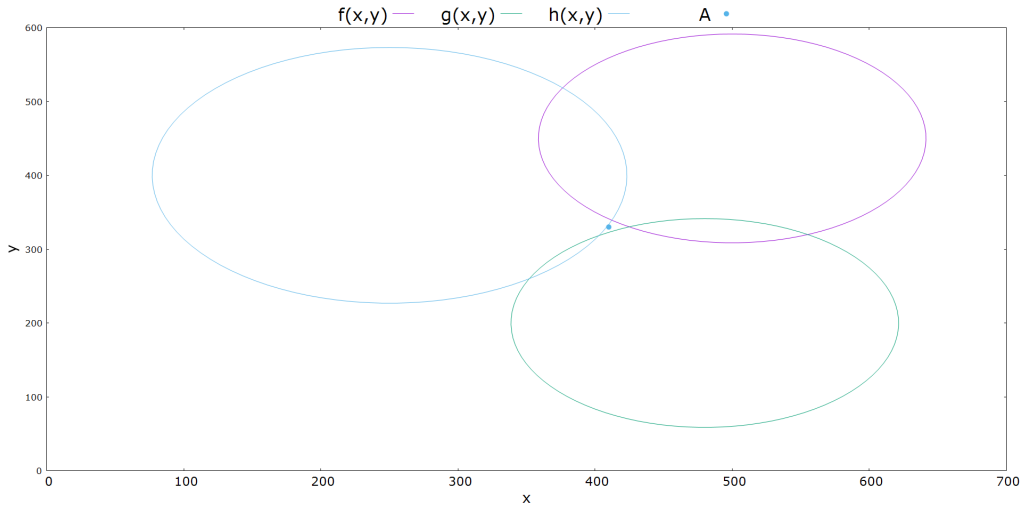

Figure 4.11: Non-linear for larger numbers. Starting guess A = (410,330).

Corr/It	Float		Posit32	
It	x	y	x	y
1	36.375488281250	11.2063903808593	36.3754959106445	11.2063980102539
2	0.2497558593750	1.1450195312500	0.249763488769531	1.14500427246093
3	0.0354614257812	0.0814819335937	0.035453796386718	0.08145904541015
4	0.0036621093750	0.0061950683593	0.003662109375000	0.00619506835937
5	0.0003356933593	0.0004577636718	0.000343322753906	0.00048828125000
6	0.0000305175781	0.0000305175781	0.000030517578125	0.00003814697265
7	0.0000000000000	0.0000000000000	0.000000000000000	0.00000000000000

Table 4.6: Float compared to Posit32 on approximating a solution to q(x,y),w(x,y),r(x,y)

The second test used circles with the same proportions in a smaller scale to contain the domain of numbers within $0-1$.

$$f(x,y) = (x-0.5)^2 + (y-0.45)^2 - 0.02$$
$$g(x,y) = (x-0.48)^2 + (y-0.2)^2 - 0.02$$
$$h(x,y) = (x-0.25)^2 + (y-0.4)^2 - 0.03$$

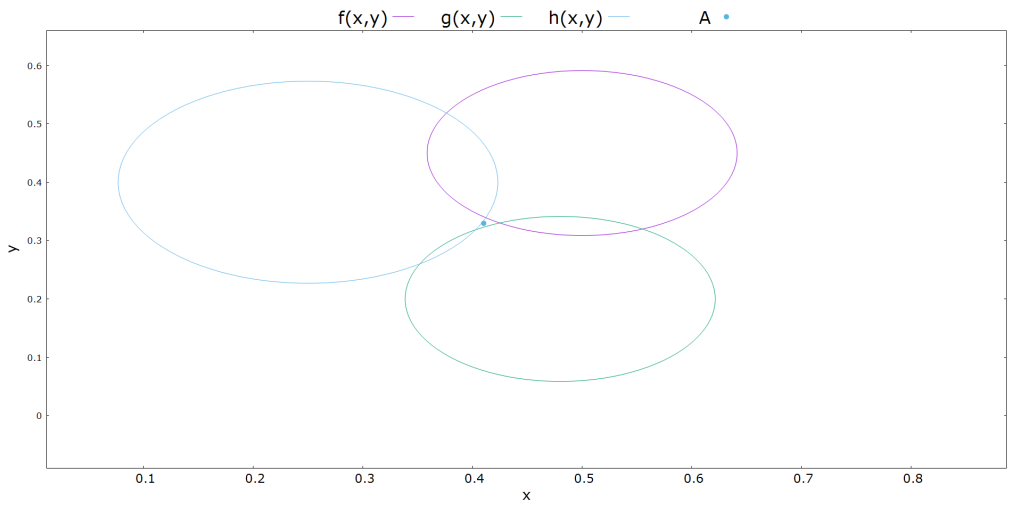

Figure 4.12: Approximation of solution to three circles. Starting guess A = (0.41,0.33).

Corr/It	Float		Posit32	
It	x	y	x	y
1	0.0165068209171	0.0223132967948	0.0165068395435	0.0223132800310
2	0.0003705322742	0.0000343024730	0.0003705602139	0.0000342763960
3	0.0000229775905	0.0000039041042	0.0000229943543	0.0000039190053
4	0.0000015199184	0.0000006556510	0.0000015310943	0.0000006631016
5	0.0000001192092	0.0000000596046	0.0000001098960	0.0000000726431
6	0.0000000298023	0.0000000000000	0.0000000074505	0.0000000074505
7	0.0000000000000	0.0000000000000	0.0000000018626	0.0000000000000

Table 4.7: Float compared to Posit32 on approximating a solution to f(x,y),g(x,y),h(x,y).

5 Discussion

5.1 Results

For the first test table 4.1 shows that Posit32 have a two digit higher precision while in table 4.2 the formats have slightly the same precision. This could be an effect of the Posit32 losing precision on numbers far from $0 - 1$.
In the second test, in figure 4.1-4.3 Posit32 performs better than Float when comparing the results to the Long Double approximation. In figure 4.2-4.3 Posit32 yields a higher precision over Float, disregarding on which interval of numbers. This was surprising as it was expected that Posit32 would lose some precision representing numbers greater than one. While approximating a polynomial Posit32 and Float performs close to the same on small number intervals, but both formats loses precision compared to Long Double when representing larger numbers.
As for the third test figure 4.4 and 4.6 showcases that Posit32 performs better than Float on larger numbers. A surprising result are the large increase of precision on certain intervals on both Float and Posit32. We assume this is caused by some sort of numbers making the format do a rounding error to their benefit, increasing the precision instead. When approximating integrals Posit32 also seems to perform better than Float.
As for approximate integral test, figure 4.7 showcases that Float starts to lose precision on partitions larger 10^3 while the precision of Posit32 keeps increasing until 10^4 partitions and starts losing on 10^5 partitions, while still having higher precision than Float. In figure 4.8 Posit32 keeps the same precision as Float up except for an interval between $10^4 - 10^7$ where Posit32 have one digit higher precision. In figure 4.9 Posit32 outperforms Float after 10^3 partitions Posit32 keeps the same precision as long double up to eight-digit precision but then starts losing precision but still keeps around two digit better precision than Float. In figure 4.10 there was a surprising result where Float and Posit32 performs the same and had a higher precision than Long Double between an interval of $10^2 - 10^4$ partitions. We believe this is caused by rounding error mechanics giving the formats a temporary increasing precision compared to Long double. The precision decreases fast with partitions higher than 10^4. We believe the reason Posit32 does

not perform better than Float in this test is due to the large numbers represented in the integral approximation.
For the last test Float and Posit32 performs the same on numbers greater than 1, by looking at table 4.5, both formats starts to overflow on the seventh iteration while on numbers between $0 - 1$ as can be seen in table 4.6. Posit32 outperforms Float with an eight-digit precision on the sixth iteration of correction, where Float starts to overflow.

5.2 Method

The test was run ten times to make sure the algorithms were working as they should. We did not compute any mean-value since the computations are deterministic. We always had the same output-data for an input-data. The reason we used elementary numerical methods was to keep the algorithms on a complex level but also to use methods that had some sort of connection to real-world applications like computing an area or an airplane trajectory. We had an ambition of conducting more extensive testing with a lot more different tests and a larger set of input data to study how the precision behaves but also look on execution time and memory allocation of the different formats but due to difficulties with implementation of the methods the time allocated for testing was delimited to only look at the precision, which is in the end, the goal of this thesis. There was also an ambition to in some extent find the threshold when Posit starts to lose precision on numbers greater than one but due to the large numbers of tests necessary to find this threshold and the small time interval to conduct tests we decided not to do this. All methods yielded an overall expected result with some surprises. For example in the polynomial approximation where both Float and Posit32 had a high increase of precision on a particular interval and the reason to why Posit32 does not lose precision on numbers greater than one in the test for fitting data to a linear model. It could be an implementation error in the test, or the nature of the test or the reason stated in the discussion of the results.

5.3 Conclusion

This thesis has investigated the effect on using the Posit floating point format compared to the IEEE 754 format by conducting tests with elementary numerical methods. The results show that Posit32, the 32-bit variant of Posit, performs equally or better than Float, the corresponding bit-size of IEEE 754.For the tests done in this paper we found no direct downside using Posit as floating point format in any of results, Posit performed consistently through the entire test-suit. This means that Posit as a floating point format could potentially act as a replacement for the current standard. However, more extensive testing is necessary before this could be realized. The bit-size Posit64 must be implemented and tested compared to Double to investigate if it follows the same pattern as with Posit32 compared to Float. It is also necessary to implement hardware support for Posit to evaluate the impacts of the precision, but also to look on the run-time and memory usage to study if Posit could reduce energy consumption as stated at section 1.5

References

[1] Bailey, D.H. et al. “The Nas Parallel Benchmarks”. In: *The International Journal of Supercomputing Applications* 5.3 (1991), pp. 63–73. DOI: 10.1177/109434209100500306. eprint: https://doi.org/10.1177/109434209100500306. URL: https://doi.org/10.1177/109434209100500306.

[2] Chien, Steven Wei Der, Peng, Ivy Bo, and Markidis, Stefano. “Posit NPB: Assessing the Precision Improvement in HPC Scientific Applications”. In: *CoRR* abs/1907.05917 (2019). arXiv: 1907.05917. URL: http://arxiv.org/abs/1907.05917.

[3] Gustafson, J. L. Yonemoto I. “*Beating floating point at its own game: Posit arithmetic. Supercomputing Frontiers and Innovations*”. In: (2017), pp. 71–86.

[4] Gustafson, John L. *Posit Arithmetic*. https://posithub.org/docs/Posits4.pdf. Accessed: 2020-03-11.

[5] Gustafson, John L. *The end of error: unum computing*. eng.

[6] Gustafson, John L. *Stanford Seminar: Beyond Floationg Point: Next Genereation Arithmetic*. https://youtu.be/aP0Y1uAA-2Y?t=2508. Channel: stanfordonline. Feb. 2017.

[7] *IEEE Std 754-2019 (Revision of IEEE 754-2008): IEEE Standard for Floating-Point Arithmetic*. IEEE, 2019. ISBN: 1-5044-5924-5.

[8] John L. Gustafson S.H. Leong (Cerlane), W.L.Koh (Vanessa). *Posit Standard Documentation*. https://posithub.org/docs/posit_standard.pdf. Accessed: 2020-03-13.

[9] Klöwer, Milan, Düben, Peter D., and Palmer, Tim N. “Posits as an Alternative to Floats for Weather and Climate Models”. In: *Proceedings of the Conference for Next Generation Arithmetic 2019*. CoNGA’19. Singapore, Singapore: Association for Computing Machinery, 2019. ISBN: 9781450371391. DOI: 10.1145/3316279.3316281. URL: https://doi.org/10.1145/3316279.3316281.

[10] Kneusel, Ronald T. *Numbers and Computers*. Springer, 2015. ISBN: 978-3-319-17260-6.

[11] Moore, G. E. “Cramming More Components Onto Integrated Circuits”. In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[12] Office, U.S. Government Accountability. “*PATRIOT MISSILE DEFENSE: Software Problem Led to System Failure at Dhahran, Saudi Arabia*”. In: *Office of Public Affairs* (1992), (1, 5, 15).

[13] Randell B., Springerlink. “*The Origins of Digital Computers: Selected Papers*”. In: Third ed., Texts and Monographs in Computer Science (1982).

[14] Rojas, R. “Konrad Zuse’s legacy: the architecture of the Z1 and Z3”. eng. In: *IEEE Annals of the History of Computing* 19.2 (1997), pp. 5–16. ISSN: 1058-6180.

[15] Sauer, Timothy. *Numerical Analysis*. Ed. by Dierdre Lynch. Pearson Education. Inc., 2012.

[16] Wan, Zishen et al. “Study of Posit Numeric in Speech Recognition Neural Inference”. In: (2018).

TRITA-EECS-EX-2020: 346