Technische Universiteit Delft

Bachelor Eindproject

Posits als vervanging van floating-points

Een vergelijking van Unum Type III Posits met IEEE 754 Floating Points met Mathematica en Python

(Posits as a replacement for floating points: A comparison of Unum Type III Posits with IEEE 754 Floating Points using Mathematica and Python)

Auteur:
Stan van der Linde
(4299779)

Begeleider:
Matthias Möller

24 september 2018

Samenvatting

In dit verslag worden de resultaten van literatuuronderzoek naar de voor- en nadelen van floats en posits gepresenteerd. Naar aanleiding van de resultaten is de hypothese opgesteld dat posits een goede “drop-in replacement” zouden zijn voor floats vanwege het grotere bereik, de afwezigheid van overflow en underflow en omdat ze enkel één NaN-waarde hebben. Met behulp van Wolfram Mathematica is een omgeving gemaakt welke de prestaties van IEEE 754 floating point standard (floats) vergelijkt met Unum Type III posits in 8-bits. De resultaten hiervan bevestigden de hypothese. Als vervolgonderzoek zijn de 32-bits versies van dezelfde floats en posits vergeleken in Python. Als praktijkvoorbeeld is de methode van Newton-Raphson gekozen met verschillende functies. De resultaten van deze tests lijken de hypothese grotendeels te ontkrachten voor zeer grote en zeer kleine getallen. Gebaseerd op deze onderzoeken lijken posits in de meeste gevallen een geschikte drop-in replacement voor floats, maar zijn ze minder nauwkeurig dan floats wanneer gerekend wordt met zeer grote en kleine getallen. Posits bieden echter meer stabiliteit doordat ze geen overflow en underflow hebben en daarom wel een waarde weergeven. De toepasbaarheid van deze waardes moet nog verder worden onderzocht. Ook kunnen andere numerieke methodes zoals Runge-Kutta gebruikt worden om de hypothese verder te testen.

Inhoudsopgave

1 Inleiding **2**

2 IEEE 754 Floating Points **4**
2.1 Wat zijn floats? 4
2.2 Voor- en nadelen floats 7

3 Unum Type III: Posits **10**
3.1 Type I en Type II 10
3.2 Wat zijn posits? 11
3.3 Verwachte voor- en nadelen posits 13

4 Posits vs Floats 8-bits **16**
4.1 Basisfuncties met één variabele waarde 16
4.1.1 Methode 16
4.1.2 Gedrag rond 0 16
4.1.3 Representaties $f(x) = \frac{1}{x}$ 17
4.1.4 Representaties $f(x) = x^2$ 18
4.1.5 Representaties $f(x) = \sqrt{x}$ 19
4.1.6 Representaties $f(x) = 2^x$ 19
4.2 Basisfuncties met twee variabele waardes 20
4.2.1 Optelling en aftrekken: $x + y$ 20
4.2.2 Vermenigvuldiging: $x \cdot y$ 21
4.3 Samenvatting 22

5 Posits vs Floats 32-bits **24**
5.1 Methode van Newton-Raphson 24
5.1.1 $f(x) = x^3 - 1$ en $x_0 = \frac{1}{4}$ 25
5.1.2 $f(x) = x^3$ en $x_0 = \frac{1}{4}$ 28
5.1.3 $f(x) = x^{120} - 2^{-120}$ en $x_0 = 4$ 30
5.1.4 $f(x) = x^{120} - 2^{-120}$ en $x_0 = 1$ 32
5.2 Samenvatting 34

6 Conclusies **35**

7 Discussie **37**

1 | Inleiding

Op 4 juni 1996 steeg de Ariane 5 op, om vervolgens 39 seconden later een van de duurste vuurpijlen ooit te worden [1]. De Ariane 5 explodeerde, omdat er een overflow optrad bij het omzetten van een integer naar een floating point [1]. Dit betekent dat een getal “afgerond” werd naar oneindig. Door de overflow kon niet langer verder gerekend worden. Dit zorgde ervoor dat het systeem crashte en op deze wijze ontplofte een raket waar 7 miljard in was gestoken aan onderzoek.

Overflows kunnen dus grote problemen veroorzaken. Dit is een van de redenen dat Gustafson onderzoek heeft gedaan naar een datatype waar geen overflow in plaatsvindt. Eind 2017 publiceerde hij in Posit Arithmetic [3] een mogelijke oplossing voor dit probleem: de *Unum Type III Posit*. Dit is een datatype dat geen underflow, overflow en NaN (Not a Number) waardes kent. Gustafson beweert dat posits de huidige IEEE 754 floating points standaard niet alleen kan vervangen, maar ze ook nauwkeuriger rekenen. Dit is een gewaagde uitspraak aangezien dit al ruim 30 jaar de gebruikte standaard is. Om na te gaan of deze uitspraak rechtvaardig is, is het belangrijk de voor- en nadelen van de twee datatypen goed tegen elkaar af te wegen. Dit leidt tot de onderzoeksvraag van dit verslag.

Onderzoeksvraag

Zijn Unum Type III Posits een geschikte drop-in replacement van de huidige IEEE 754 Floating Point standaard?

De gestelde onderzoeksvraag is nog te breed om goed onderzoek te kunnen doen. Om het onderwerp van dit verslag beter af te bakenen en om een heldere structuur aan te houden worden nog enkele deelvragen gesteld.

Deelvragen

- Wat zijn IEEE 754 Floating Points?
- Wat zijn Unum Type III Posits?
- Wat zijn de verwachte verschillen tussen Unum Type III Posits en IEEE 754 Floating Points?
- Wat zijn de verschillen tussen Unum Type III Posits en IEEE 754 Floating Points in praktijk voorbeelden?
- Voor welke doeleinden zijn Unum Type III Posits geschikt?
- Voor welke doeleinden zijn IEEE 754 Floating Points geschikt?

Wat floats en posits zijn wordt behandeld in respectievelijk hoofdstuk (2) en (3). In hoofdstuk (3) wordt ook gekeken naar enkele verwachte voor- en nadelen van posits en daarmee wat verschillend is ten op zichte van floats. Om verder in te gaan op deze verwachte verschillen wordt in hoofdstuk (4) op een 8-bit schaal met behulp van Wolfram Mathematica verder onderzoek gedaan. Posits en floats worden hier op gelijke wijze getest om verschillen in kaart te brengen. In hoofdstuk (5) wordt gekeken naar hoe ze in praktijk verschillen. Met behulp van Python worden 32-bits floats en posits vergeleken in een praktijkomgeving. Door de verkregen data te vergelijken worden in hoofdstuk (6) de doeleinden van posits en floats besproken.

2 | IEEE 754 Floating Points

Op een computer zijn er verschillende manieren om getallen op te slaan, zo zijn er bijvoorbeeld integers, fixed-point en floating-point representaties van getallen elk met hun voor- en nadelen. Integers reken alleen met gehele getallen, wat het rekenwerk snel en precies maakt, maar kunnen geen kommagetallen representeren. Een vroege oplossing voor dit probleem was de fixed-point notatie. Zoals de naam al zegt wordt hier de komma op een vast punt neergezet, waardoor komma getallen mogelijk zijn. Dit houdt het rekenwerk snel en simpel, maar heeft als nadeel dat het slechts een klein bereik heeft. Een andere oplossing voor dit probleem zijn floating-points, waarbij de komma niet op een vaste plek staat. Dit geeft een groter bereik, maar maakt rekenen met de getallen lastiger. In de volgende paragraaf wordt gedetailleerder ingegaan op deze en andere eigenschappen van floats. De beschrijving van floats is gebaseerd op de paper *IEEE Standard 754 Floating Point Numbers* [2].

2.1 Wat zijn floats?

Het idee achter IEEE Standard 754 Floating Point Numbers (floats) is dat de komma geen vaste positie heeft in een getal. Dit wordt verwezenlijkt door een vorm van wetenschappelijke notatie, waar een getal wordt opgedeeld in een mantisse m (ook wel significant of coëfficiënt genoemd), een grondgetal b (basis) en een exponent e. Een getal in wetenschappelijke notatie heeft dan de vorm

$$\pm m \cdot b^e. \tag{2.1}$$

In een decimaal stelsel betekent dit dat een getal zoals $9876,54$ dan omgeschreven wordt naar $9,87654 \cdot 10^3$. In een hexadecimaal stelsel $1a2bc,d$ wordt het getal dan omgeschreven naar $1,a2bcd \cdot 16^4$.

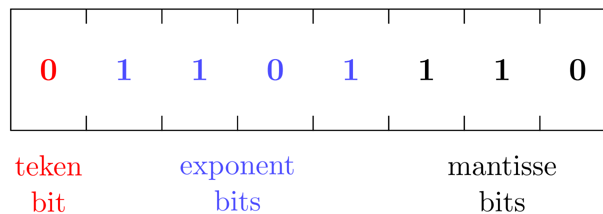
Een computer rekent echter binair, ofwel modulo 2. Ook hier kan de wetenschappelijke notatie toegepast worden. Het getal $100,1110$ wordt dan bijvoorbeeld $1,001110 \cdot 2^3$. Een interessante eigenschap van de wetenschappelijke notatie van getallen in een binair stelsel is dat de representatie van de getallen altijd begint met een 1. De enige uitzondering is het getal 0.

Deze specifieke eigenschap wordt gebruikt in een optimalisatie die later in deze paragraaf beschreven wordt.

In de vorige voorbeelden was te zien dat het grondgetal van te voren vastgesteld wordt. Floats werken binair, het grondgetal is dus 2 en dit geldt voor alle floats, daarom hoeft dit niet opgeslagen te worden in de float. Nu blijven er nog 3 variabelen over uit formule (2.1): het teken ($+$ of $-$), de mantisse en de exponenten. Dit leidt tot de drie soorten bits waaruit een float bestaat, namelijk:

- een teken bit t
- exponent bits e
- mantisse bits m

Hoe dit eruit ziet, is het makkelijkst te begrijpen aan de hand van een voorbeeld, daarom is in figuur (2.1) een voorbeeld van een 8-bits floating point weergeven.



Figuur 2.1: 8-bit floating point visualisatie met exponent van lengte 4.

Teken bit

Het teken bit bepaalt of een getal positief of negatief is. Als het teken bit 0 is zoals in het voorbeeld, dan is het getal positief. Wanneer het teken bit gelijk is aan 1, is het negatief.

Exponent bits

De exponent bits bepalen hoe groot de exponent is die wordt gebruikt. Aangezien er binair gerekend wordt, is het grondgetal altijd 2 en hoeft het dus niet in de float opgeslagen te worden. Gezien floats gemaakt zijn om een groot bereik van getallen te hebben is het belangrijk dat zowel hele kleine als extreem grote getallen berekend kunnen worden. Simpelweg 2^e uitrekenen geeft geen nauwkeurig genoeg bereik rond 0, omdat de kleinst mogelijke exponent dan 1 zou zijn. Daarom zit er in de exponent bits nog meer verborgen informatie, de *offset(e)* welke gelijk is aan

$$offset(e) = 2^{\#bits(e)-1} - 1. \tag{2.2}$$

Dit maakt het mogelijk om ook negatieve exponenten weer te geven zonder een extra tekenbit in de exponent toe te hoeven voegen. In het voorbeeld van figuur (2.1) geldt $\#bits(e) = 4$, dus $offset(e) = 7$. De waarde van de exponent wordt berekend met

$$2^{e-offset(e)}. \tag{2.3}$$

In het gekozen voorbeeld geldt $e = 1101(\text{mod } 2) = 13(\text{mod } 10)$. De waarde van de exponent is in dit geval dus 2^6.

Mantisse bits

Het grondgetal, teken en exponent zijn bekend. Nu resteert alleen de mantisse nog. Hier is zoals eerder vermeld nog een optimalisatie mogelijk, want met uitzondering van 0, kan iedere getal zo geschreven worden dat het begint met een 1, neem als voorbeeld:

$$0,0001011 = 1,011 \cdot 2^{-4}.$$

Aangezien dit zo vaak voorkomt hoeft deze eerste 1 niet opgeslagen te worden in de float. In figuur (2.1) betekent dit dat 110 wordt gelezen als $1,110 \text{ (mod 2)} = 1,75 \text{ (mod 10)}$, ofwel $1,m$.

Alle onderdelen van formule (2.1) zijn nu in de float opgeslagen. Deze kunnen compact in de volgende formule worden weergeven:

$$(-1)^t \cdot 2^{e-offset(e)} \cdot 1,m. \tag{2.4}$$

De float van figuur (2.1) representeert dus het getal $+1,75 \cdot 2^6 = 112$. Tot nu toe zijn enkel de zogenaamde "genormaliseerde floats"besproken. Dit zijn alle floats waarvan de exponent niet geheel uit 0-bits of geheel uit 1-bits bestaat.

Uitzonderingen

Echter, er zijn een aantal belangrijke waardes die niet genoteerd kunnen worden met behulp van formule (2.4), namelijk:

i 0

ii Positieve getallen kleiner dan $2^{-offset(e)}$ (positieve underflow)

iii Negatieve getallen groter dan $-2^{-offset(e)}$ (negatieve underflow)

iv Positieve getallen groter dan $\left(2 - 2^{-\max(m)}\right) \cdot 2^{\max(e)-offset(e)}$ (positieve overflow)

v Negatieve getallen kleiner dan $-\left(2-2^{-\max(m)}\right)\cdot 2^{\max(e)-offset(e)}$ (negatieve overflow)

Om deze uitzonderingen toch uit te drukken in floats worden speciale bit representaties gereserveerd. Om dit uit te voeren wordt gekeken naar de exponent.

Een oplossing voor de eerste drie gevallen (i-iii) kan gevonden worden met zogenaamde "gedenormaliseerde"floats. Dit zijn floats waarvan de exponent alleen uit 0-bits bestaat. In dit geval wordt de formule (2.4) niet gebruikt, maar in plaats daarvan wordt de volgende formule gebruikt om de gedenormaliseerde floats weer te geven:

$$(-1)^{t}\cdot 2^{-offset(e)}\cdot 0,m. \tag{2.5}$$

Merk op dat dit er voor zorgt dat er twee floats zijn die nul representeren. Door het teken bit is er zowel een positieve als negatieve nul. Deze twee representaties zijn echter wel gelijk bij een "compare"functie.

Geval (iv) en (v) zijn echter nog steeds niet uitgedrukt. Dit wordt gedaan met behulp van $+\infty$ en $-\infty$ respectievelijk. Waardes groter dan $(2-2^{-\max(m)})$ worden weergeven met ∞ en waardes kleiner dan $-(2-2^{-\max(m)})$ worden weergeven met $-\infty$. Dit proces wordt overflow genoemd. Als de exponent uitsluitend uit 1-bits bestaat en de mantisse gelijk is aan 0, dan representeert de float afhankelijk van het teken bit $+\infty$ en $-\infty$. Als de exponent uitsluitend uit 1-bits bestaat en de mantisse niet gelijk is aan nul, dan is de float een NaN (Not a Number). Dit wordt gebruikt voor waardes die geen reëel getal voorstellen.

Alles samengevoegd levert dit één algemene formule op om iedere float om te zetten naar zijn bijbehorende waarde:

Algemene formule floats

$$f(t,e,m)=\begin{cases} e=\text{alleen 1 bits}, & \begin{cases} m=0, & (-1)^{t}\infty, \\ m\neq 0 & NaN, \end{cases} \\ e=\text{alleen 0-bits}, & (-1)^{t}\cdot 2^{-offset(e)}\cdot 0,m, \\ \text{anders}, & (-1)^{t}\cdot 2^{e-offset(e)}\cdot 1,m. \end{cases} \tag{2.6}$$

2.2 Voor- en nadelen floats

Bij het maken van de IEEE 754 floating point standaard zijn verschillende keuzes en aannames gemaakt. Zo heeft een standaard 32-bits float bijvoor-

beeld een exponent van 8 bits. Dit soort keuzes zorgen voor verschillende voor- en nadelen die in deze paragraaf besproken zullen worden.

Het grootste voordeel dat floats hebben ten opzichte van data-types als fixed-point notatie en integers is het bereik dat ze hebben. Het bereik van een 32-bit integer zonder tekenbit is bijvoorbeeld van 0 tot $2^{32} - 1 = 4,29 \cdot 10^9$. Aanzienlijk kleiner dan dat van een 32-bit float die ongeveer een bereik heeft van $-3,40 \cdot 10^{38}$ tot $3,40 \cdot 10^{38}$[2]. De exponent zorgt dus voor een groot bereik waardoor grote getallen uitgedrukt kunnen worden met relatief weinig bits.

Dit enorme bereik dat floats hebben is niet gratis. Een nadeel van de notatie is dat, in tegenstelling tot integers, er met afronding wordt gewerkt. Floats geven dus niet altijd exacte waardes. In het algemeen wordt bij het invoeren van een getal al direct een kleine fout gemaakt, omdat dit getal waarschijnlijk afgerond wordt wanneer het opgeslagen wordt in een float. Als er vervolgens een bewerking wordt uitgevoerd kunnen fouten zich opstapelen en op deze wijze een significant verschil maken. Neem bijvoorbeeld een klein getal a en tel dit op bij een groot getal b, het kan dan voorkomen dat a in zijn geheel wegvalt. De data die a had valt dan dus volledig weg.

Als de exponent gelijk is aan 0, dan wordt de float gedenormaliseerd genoemd. Gedenormaliseerde floats zorgen ervoor dat in de buurt van 0 er meer waarden uitgedrukt kunnen worden dan in eerste instantie verwacht zou worden. Het voordeel is dus dat er rondom dit veel voorkomende punt goed gerekend kan worden.

Echter, de implementatie van gedenormaliseerde floats zorgt er ook voor dat er twee verschillende bit-representaties zijn voor het getal 0. Dit veroorzaakt problemen wanneer twee getallen vergeleken moeten worden, omdat -0 gelijk moet zijn aan $+0$ terwijl ze verschillende bit-representaties hebben. Dit zorgt voor een extra rekenstap bij iedere vergelijking en dus extra rekentijd.

Er is gekozen is om $+\infty$ en $-\infty$ uit te drukken in floats. Dit maakt het mogelijk te rekenen met deze waardes. Dit kan er voor zorgen dat runtime-errors voorkomen kunnen worden wanneer overflow of underflow plaatsvindt. Stel er is een berekening waardoor een underflow de bewerking $\frac{1}{+0}$ uitgevoerd wordt, dit geeft dan $+\infty$. Als dit resultaat vervolgens wordt gebruikt om bijvoorbeeld $\frac{1}{+\infty}$ uit te rekenen, dan geeft dit weer $+0$. Op deze manier is een runtime-error voorkomen.

De manier waarop $+\infty$ en $-\infty$ worden gerepresenteerd in floats zorgen voor een ander groot nadeel. Een float is een vorm van oneindigheid, wanneer de

exponent uitsluitend is opgebouwd met 1-bits en de mantisse gelijk is aan 0. Dit lijkt geen probleem, maar alle waardes waar de exponent uit 1-bits bestaat en de mantisse ongelijk is aan 0 is een NaN waarde. Dit zorgt voor erg veel "verspilde" bits. Bij een 32-bit float zijn er bijvoorbeeld ongeveer $8,4$ miljoen verschillende manieren om een NaN aan te geven. NaN waardes kunnen nuttig zijn om een fout in een berekening aan te geven, denk bijvoorbeeld aan een negtatieve wortel trekken. Echter, $8,4$ miljoen manieren om aan te geven dat er iets fout gaat is zonde van de bits.

Een ander nadeel ten opzichte van bijvoorbeeld integers is het feit dat floats opgedeeld zijn in een teken, exponent en mantisse bit. Het kost meer berekeningen om een float weer te geven dan bijvoorbeeld een integer. Dit zorgt ervoor dat floats in vergelijking met integers en fixed-point notatie langzaam zijn. Het verlies in snelheid is klein, maar wel dusdanig groot dat het genoemd mag worden.

Samenvatting voor- en nadelen floats

Voordelen	**Nadelen**
- Rekent met kommagetallen	- Afrondfouten
- Groot bereik door exponent	- Relatief langzaam
- Kan met 0 en ∞ rekenen	- Veel niet gebruikte bit representaties (NaN)
- Nauwkeurig rond 0	- Twee representaties voor 0 met verschillende eigenschappen

3 | Unum Type III: Posits

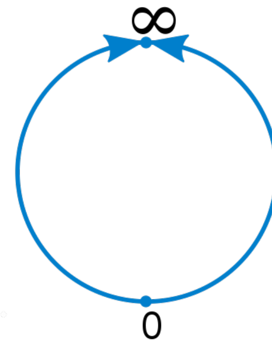
“There ain’t no such thing as a free lunch.”Ofwel gratis lunch bestaat niet is een Amerikaans gezegde welke vaak van toepassing is de wereld van informatica. Het betekent dat “verbetering” niet gratis is, er wordt altijd iets opgeofferd. Verbetering is dus afwegingen maken zodanig dat de winst die gemaakt wordt opweegt tegen de gemaakte opofferingen. In dit hoofdstuk wordt Unum Type III: Posits (posits) beschreven, een mogelijke vervanging van floats die in 2017 door Gustafson [3] is voorgesteld. Om posits beter te begrijpen is het ook handig voorkennis te hebben van Type I en II unums. Daarom worden deze eerst beschreven, waarna posits worden uitgelegd. Als laatste zullen enkele verwachte voor- en nadelen worden besproken.

3.1 Type I en Type II

De belangrijkste afweging die wordt gemaakt bij het ontwerpen van een systeem om getallen te representeren op een computer is de verhouding tussen bereik en nauwkeurigheid. Gustafson keek naar de nauwkeurigheid van floats en probeerde deze met Unum Type I te vergroten. De naam *unum* komt van *u*niversal *num*ber [4]. Het maakt gebruik van dezelfde vorm van teken, exponent en mantisse bits, met als verschil dat de lengte van de mantisse en exponent bits variabel is.

Unums van Type I werken met interval rekenen. Dit betekent dat een getal niet een vaste waarde heeft zoals bij floats, maar een waarde ergens in een interval heeft. Dit is het makkelijkst te begrijpen aan de hand van een voorbeeld. Stel je reist met de Intercity van Delft naar Dordrecht. De reisplanner geeft aan dat dit 31 minuten duurt. Echter, aangezien er onzekerheden meespelen in het openbaar vervoer zal dit niet altijd 31 minuten bedragen, maar je kan wel met grote zekerheid zeggen dat dit ergens in het interval 29 tot 31 minuten zal vallen. Op deze manier ronden Unums van Type I af en dit maakt ze nauwkeuriger dan floats. Implementatie is echter onwaarschijnlijk, omdat het omgaan met de variabele lengtes van de exponent en mantisse lastig is. Dit zorgt ervoor dat unums van Type I te traag worden om ooit te kunnen concurreren met floats.

Bij het ontwerpen van Type II unums werd helemaal opnieuw begonnen. De definities van de teken, exponent en mantisse bits van floats werden niet gebruikt waardoor er geen compatibiliteit was. Dit schept daarentegen wel ruimte om meer vanuit een wiskundig oogpunt te denken. Unums van type II zijn daarom ook gebaseerd op de reëel projectieve lijn, welke ook wel als $\hat{\mathbb{R}} = \mathbb{R} \cup \{\infty\}$ beschreven wordt. Dit is een een lijn met halverwege het punt 0 en op beide uiteinden ∞. Hierdoor wordt de reëele projectieve lijn vaak weergegeven als een cirkel. Het idee van unum type II is dat iedere waarde een multiplicatieve inverse heeft waardoor optellen en vermenigvuldigen heel nauwkeurig uitgevoerd kan worden. Echter is het lastig om floats en andere waardes om te zetten naar deze unums. Hiervoor is namelijk een opzoek tabel nodig [3]. Dit zorgt ervoor dat hoge bit-aantallen te langzaam worden om bruikbaar te zijn. Unums van type III zijn ontworpen om dit probleem op te lossen.



Figuur 3.1: Reëele projectieve lijn in cirkelvorm. [5]

3.2 Wat zijn posits?

Het idee van unums van type III is dus dat ze net als type II gebaseerd zijn op de reëele projectieve lijn, al is dit in mindere mate dan bij type II. De nauwkeurigheid van de projectieve lijn wordt versimpeld, door de harde eis dat ieder getal een multiplicatieve inverse nodig heeft los te laten. Dit maakt een systeem mogelijk waar geen opzoek tabel voor nodig is. Op deze wijze kan sneller gerekend worden met deze unums, dan die van Type II. Er zijn twee versies van deze unums:

- Posits
- Valids

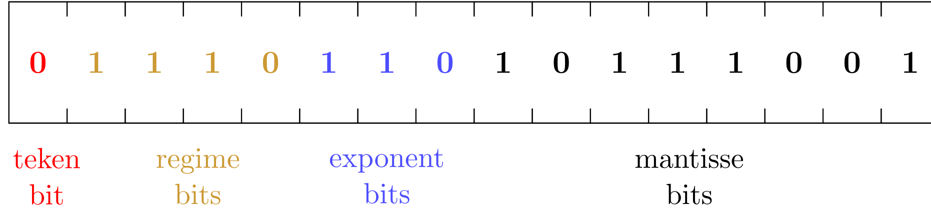
Valids gebruiken één van de bits om aan te geven of een getal is afgerond, zo kan de gebruiker altijd weten of een getal exact is of niet. Posits gebruiken deze bit voor extra nauwkeurigheid, maar kunnen niet meer aangeven of er ergens een afronding heeft plaatsgevonden. In dit verslag wordt alleen ingegaan op de werking van posits, wie genteresseerd is in de werking van valids wordt aangeraden *The End of Error: Unum Computing* [6] te bestuderen.

Allereerst is het belangrijk de structuur van een posit te begrijpen, welke in eerste instantie veel weg lijkt te hebben van een float. Een posit bestaat uit vier delen, namelijk:

– een teken bit t

- exponent bits es
- mantisse bits m
- regime bits r

Ook posits zijn het makkelijkst uit te leggen aan de hand van een voorbeeld. In figuur (3.2) is een posit weergeven met een directe uitwerking. Het teken



Figuur 3.2: Voorbeeld van een 16-bit posit. $+256^2 \cdot 2^6 \cdot (1 + \frac{185}{256}) = 7225344$

bit en de mantisse bits werken op dezelfde manier als bij genormaliseerde floats. Het teken bit in dit voorbeeld geeft dus een + als teken. De mantisse geeft als breuk $\frac{185}{256}$, net als bij floats wordt hier 1 bij opgeteld. Dit geeft in dit geval dus een mantisse van $1 + \frac{185}{256} \approx 1,7227$. De exponent werkt ook op een vergelijkbare manier, met als verschil dat er geen $offset$ is. In het voorbeeld is de exponent dus 2^6.

Het grote verschil met floats zijn de regime bits die worden gebruikt om zowel extreem grote als kleine getallen nauwkeurig weer te geven. De regime bits werken als een soort super exponent die afhangt van een zogenaamde *useed*. De *useed* wordt gegeven door

$$useed = 2^{2^{len(es)}}. \tag{3.1}$$

In het voorbeeld was de exponent es 3 bits lang, dit betekent dat de useed 256 was. Vervolgens wordt gekeken naar de lengte van het regime. Om de lengte van het regime te bepalen wordt gekeken naar het aantal bits dat nodig was voor een omslag van bits, noem dit l. Een omslag van bits is gedefinieerd als een serie van nullen of enen die omslaat naar een tegengesteld getal. In het voorbeeld staan 3 nullen gevolgd door een een 1, dit betekent $l = 3$. Zij $k = m - 1$ dan geldt in het voorbeeld van figuur (3.2) $k = 2$. Als er geen omslag plaatsvindt, dan geldt $k = n - 1$. Waar n de lengte van de posit is. De waarde van het regime wordt gegeven door

$$regime = useed^k. \tag{3.2}$$

De waarde van het $regime$ in het voorbeeld is dus $256^2 = 65536$. Met deze ingrediënten kan nu een formule voor posits worden gemaakt:

$$(-1)^t \cdot useed^k \cdot 2^e \cdot (1, m) \tag{3.3}$$

Om het *regime* te bepalen werd gekeken naar de lengte. De lengte van het *regime* is dus variabel. Dit is een verschil ten opzichte van floats. Net werd gezegd dat de mantisse en exponent van posits op vergelijkbare wijze werken als floats. De uitzondering is echter dat ook de lengte van deze twee onderdelen bij posits niet vast staat. Voor de exponent es wordt wel een maximum lengte gekozen, maar deze kent geen minimum lengte, deze verschilt per posit. Wanneer men een posit ontwerpt kiest men de lengte van de posit n (het aantal bits) en de lengte van de exponent es. Een overzicht van alle mogelijke lengtes is weergeven in tabel (3.1).

Onderdeel	Min. lengte	Max. lengte
Teken	1	1
Regime	$\max\{1, n-2\}$	$n-1$
Exponent	0	$len(es)$
Mantisse	0	$n - len(es) - 3$

Tabel 3.1: Mogelijke lengtes van de verschillende onderdelen van een posit.

Net zoals bij floats zijn er uitzonderingen. Echter kent het posit formaat maar twee uitzonderingen, aanzienlijk minder dan floats. Dit zijn de gevallen 0 en $\pm\infty$ (oneindigheid zonder teken). Als alle onderdelen van een posit alleen bestaan uit 0-bits, dan is de waarde 0. Wanneer het teken-bit een 1-bit is en alle andere onderdelen bestaan uit 0-bits, dan is de waarde $\pm\infty$ Dit leidt tot de volgende algemene formule om de waarde van een posit te berekenen.

Algemene formule posits

$$p(t, k, e, m, n) = \begin{cases} k = n - 1 & \begin{cases} t = 0, & 0, \\ t = 1 & \pm\infty, \end{cases} \\ \text{anders,} & (-1)^t \cdot useed^k \cdot 2^e \cdot (1, m). \end{cases} \tag{3.4}$$

3.3 Verwachte voor- en nadelen posits

Zoals bij ieder datatype dat gemaakt wordt zijn er voor- en nadelen. Echter, omdat de posit zo nieuw is, kan slechts gespeculeerd worden over de voor- en nadelen. Op het moment van schrijven zijn er nog geen hardware implementaties van de posit, dus er moet op dit moment nog genoegen genomen worden met virtuele simulaties. Dit zorgt ervoor dat slechts de nauwkeurigheid getest kan worden.

Net als floats kunnen posits rekenen met kommagetallen. Ook hebben posits net als floats ten opzichte van datatypen als fixed-point notatie en integers een groot bereik. Een voordeel dat posits hebben ten opzichte van floats is echter dat dit bereik nog hoger ligt bij posits dan bij floats. Het regime was een soort super exponent, welke er niet alleen voor zorgt dat het bereik groter worden, maar ook dat getallen beter verspreid worden over de getallenlijn.

Een ander groot voordeel dat posits hebben ten opzichte van floats, is dat posits geen NaN waardes kennen. Eventueel kan $\pm\infty$ gebruikt worden om een waarde aan te geven die niet berekend kan worden. Dit zorgt ervoor dat veel meer bit representaties gebruikt kunnen worden.

Ook kunnen posits net als floats rekenen met zowel 0 als $\pm\infty$. Het voordeel dat posits hebben ten opzichte van floats is dat deze maar één 0 kennen. Daarentegen kennen posits ook maar één vorm van oneindigheid, waar floats een positieve en negatieve ∞ kennen.

Een ander nadeel van posits is dat ze net als floats afronden en dit niet aangeven zoals bijvoorbeeld bij valids. Daarentegen bestaat het concept van overflow niet bij posits. Niet alleen vanuit wiskundig oogpunt is dit fijn -afronden naar ∞ is wiskundig gezien een oneindig grote fout-, maar ook vanuit een informatica oogpunt is dit handig. Programma 's werken stabieler wanneer er met echte getallen wordt gerekend i.p.v. het concept van oneindigheid. Verder kennen posits ook geen underflow, wat ook weer vanuit een wiskundig oogpunt mooi is, de relatieve fout wordt immers aanzienlijk hoger wanneer afgerond wordt naar 0. Men kan zich wel afrvragen of dit in praktijk ook wel zo nuttig is, want bewerkingen als optellen en vermenigvuldigen gaan zeer gemakkelijk met 0.

Gustafson [3] beweert dat de rekensnelheid van posits hoger ligt dan die van floats. Het argument dat hiervoor gegeven wordt is dat, zoals in formule (3.4) te zien is, er maar 3 mogelijkheden zijn voor een posit. Een posit kan 0, $\pm\infty$, of een gestructureerd getal zijn. Dit terwijl voor floats uit formule (2.6) blijkt dat deze 5 mogelijkheden hebben, namelijk: genormaliseerde floats, gedenormaliseerde floats, $+\infty$, $-\infty$ of een NaN. Het aanmaken van een posit is in theorie dus sneller dan het aanmaken van een float.

Gustafson stopt hier echter met de vergelijking van snelheden, maar er zijn meer aspecten waar naar gekeken kan worden. Bij het uitvoeren van verschillende operaties zullen verschillen in rekensnelheid van de computer gemerkt worden. Zo kan men zich inbeelden dat de regime-bits voor extra rekenstappen zullen zorgen bij operaties zoals bijvoorbeeld optellen en vermenig-

vuldigen. Dit is echter alleen pure speculatie, omdat er nog geen hardware implementaties van zijn.

Aan de hand van deze voor- en nadelen lijkt de uitspraak van Gustafson over posits als vervanging van floats terecht te zijn. Ze hebben een groter bereik, minder NaN waardes en geen overflow en knderflow. Dit leidt tot de volgende hypothese:

Unum Type III Posits zijn een geschikte drop-in replacement voor IEEE 754 Floating Point Standard floats.

In het volgende hoofdstuk zal deze hypothese getest worden.

Verwachte voor- en nadelen posits

Voordelen	**Nadelen**
- Rekent met kommagetallen	- Afrondfouten
- Groot bereik door exponent en regime	- Relatief langzaam
- Kan met 0 en ∞ rekenen	- Geen teken bij ∞
- Geen “verspilde” bits door NaN	

4 | Posits vs Floats 8-bits

De manier waarop floats en posits werken is nu besproken. Verder zijn ook enkele verwachte voor- en nadelen bekend van zowel floats als posits. Het wordt nu tijd ze naast elkaar te zetten en de twee datatypen te vergelijken. Eerst zal dit op een kleine schaal gedaan worden, namelijk 8-bits. Door een kleine hoeveelheid bits te kiezen wordt het mogelijk alle waarden die gemaakt kunnen worden met elkaar te vergelijken. In dit hoofdstuk zullen enkele basisoperaties worden vergeleken. Deze worden verkregen met behulp van Wolfram Mathematica en behaalde resultaten uit Posit Arithmetic [3]. De conclusies die uit deze resultaten voorkomen laten zien wat interessant is om verder te testen in de praktijk.

4.1 Basisfuncties met één variabele waarde

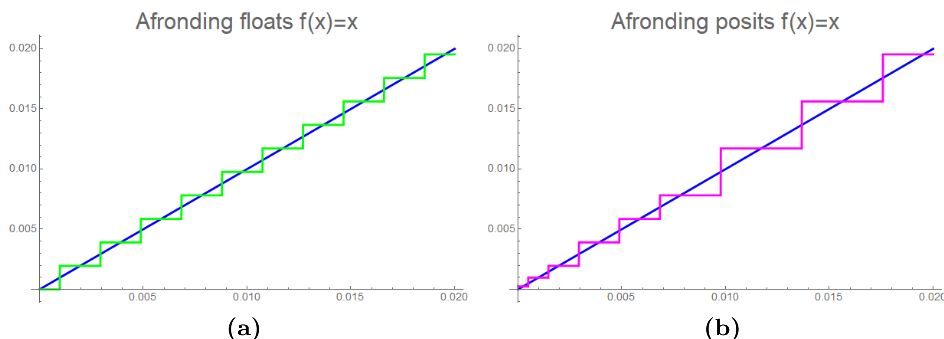
4.1.1 Methode

Met behulp van Wolfram Mathematica wordt een testomgeving gemaakt die enkele basis operaties van floats en posits kan simuleren en deze kan vergelijken met exacte waarden. Er is gekozen voor Mathematica vanwege de hoge nauwkeurigheid die dit programma kan bieden. De omgeving is gebaseerd op de omgeving uit Posits4 [7]. In deze omgeving zullen enkele basisoperaties getest worden. Om deze te testen worden alle mogelijke waarden met elkaar vergeleken. Om deze reden wordt gekozen om 8-bits floats en posits te testen. Als een grotere hoeveelheid bits zoals bijvoorbeeld 16, 32, of 64 getest zou worden, dan zou de rekentijd te groot worden. Voor de 8-bits posits wordt net als in Posit Arithmetic [3] gebruik gemaakt van 2 exponent bits. Voor een 8-bits floats is het gebruikelijk 4 bits voor de exponent te gebruiken[8].

4.1.2 Gedrag rond 0

Allereerst wordt gekeken naar de afrondfout rond 0. Om dit te weergeven wordt gekeken naar de lijn $f(x) = x$. Vervolgens zullen de waarden van de lijn geschat worden met behulp van floats en posits. Dit resulteert in figuur (4.1).

In (a) is een duidelijke trapvorm te zien, dit betekent dat de afstand tussen



Figuur 4.1: Afrondingen die floats (a) en posits (b) maken als ze de lijn $f(x) = x$ weergeven. De blauwe lijn is de exacte waarde, groen en paars geven de afschattingen van floats en posits respectievelijk aan.

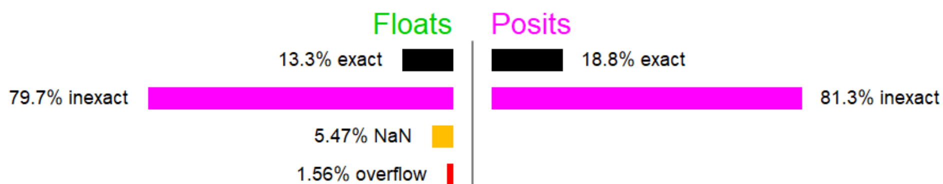
ieder getal hetzelfde is. Verder is te zien dat in de buurt van 0, afgerond wordt naar 0, dus er vind underflow plaats op deze plek. In (b) zien we ook een trapvorm, maar de afstand tussen de getallen wordt kleiner naarmate we dichter bij 0 komen. Dit betekent dat de relatieve fout die gemaakt wordt bij posits kleiner is. Verder wordt er niet afgerond naar 0, een posit representeert alleen 0 als dit ook daadwerkelijk als waarde gegeven wordt.

In de buurt van 0 werken 8-bits posits beter dan floats, omdat de afstand tussen de waarden krimpt naarmate ze dichter bij 0 komen waardoor de relatieve fout klein blijft. Verder is het vanuit wiskundig oogpunt erg mooi dat er niet afgerond wordt naar 0, maar men kan zich afvragen of dit in praktijk ook gewild is.

4.1.3 Representaties $f(x) = \frac{1}{x}$

Posits zijn gebaseerd op het idee van de reëele projectieve lijn. Bij Unum Type II was het noodzakelijk dat ieder getal een multiplicatieve inverse had, dus voor iedere x was er een $\frac{1}{x}$. Unum Type III posits hebben deze harde eis ingeruild voor snelheid. Het is daarom interessant te testen in hoeverre deze wiskundig gezien mooie eis is opgegeven voor de snelheid die posits bieden en of ze dit beter doen dan floats. De verwachting is dat posits dit beter kunnen, omdat de basis is van een van de versies waarop deze gbaseerd is. Zowel posits als floats zullen de functie uitvoeren en de resultaten zijn grafisch weergegeven in figuur (4.2).

Zoals verwacht kunnen posits deze functie vaker exact berekenen dan floats. Verder is hier nog een belangrijk voordeel te zien van de posit, er vindt geen overflow plaats en er zijn geen NaN waardes. Ieder getal wordt dus op zijn minst geschat en er kan dus verder gerekend worden. Dit kan handig zijn



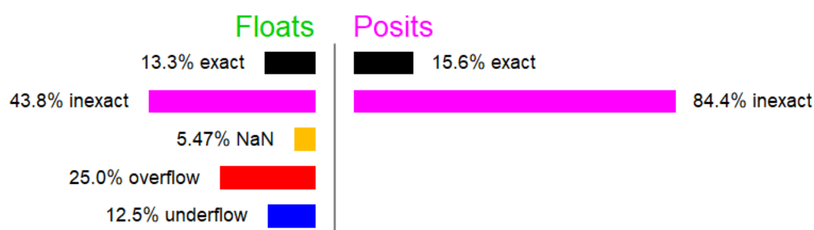
Figuur 4.2: Percentages exacte, inexacte, NaN en overflow waardes voor de functie $f(x) = \frac{1}{x}$.

wanneer een ruimte schatting voldoende informatie geeft.

Dus ook voor de functie $f(x) = \frac{1}{x}$ rekenen posits vaker exact dan floats. Ook geven ze altijd een waarde waarmee verder gerekend kan worden. Dus zoals verwacht zijn posits in dit geval een betere keuze dan floats.

4.1.4 Representaties $f(x) = x^2$

Een andere bewerking die vaak uitgevoerd wordt is het kwadrateren van een getal. Opnieuw is de verwachting dat posits dit beter kunnen vanwege de regime bits. Wanneer een getal een lege mantisse heeft en kleine hoeveelheid regime bits, dan zal het zijn kwadraat ook uitgedrukt kunnen worden in posits. Voor floats geldt ook dat als de mantisse bits leeg zijn het kwadraat uitgedrukt kan worden in floats wanneer de waarde van de exponent klein blijft. Echter heeft de float die getest wordt maar 4 exponent bits en dit zullen dus waarschijnlijk minder waarden zijn. De resultaten van de testen zijn grafisch weergeven in figuur (4.3).



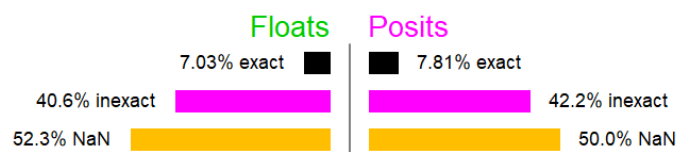
Figuur 4.3: Percentages exacte, inexacte, NaN, overflow en underflow waardes voor de functie $f(x) = x^2$.

Zoals verwacht kunnen posits de waardes vaker exact weergeven dan floats. Verder schatten posits ook veel meer, waardoor met veel meer waardes verder gerekend kan worden. Ongeveer $42,97\%$ van alle waardes gaat verloren aan NaN, overflow of underflow waardes. Waar NaN en overflow bijna geheel

onbruikbaar zijn kan er nog wel beargumenteerd worden dat underflow een goede afronding is. Wanneer underflow gezien wordt als een inexacte waarde blijven floats nog steeds ver achter op posits. Nog steeds wordt $30,47\%$ van de waardes zo goed als onbruikbaar. Dus ook voor de functie $f(x) = x^2$ werken posits aanzienlijk beter dan floats.

4.1.5 Representaties $f(x) = \sqrt{x}$

De volgende functie die vergeleken wordt is $f(x) = \sqrt{x}$. Posits konden de waarde x^2 beter uit drukken dan floats, daarom is te verwachten dat posits ook de wortelfunctie beter uit kunnen drukken. Echter kennen posits geen NaN waarde, hiervoor wordt $\pm\infty$ gebruikt, in dit geval krijgen alle getallen die kleiner zijn dan 0 automatisch een NaN waarde omdat ze de gevraagde waarde niet uit kunnen drukken. De resultaten van de test zijn weergeven in figuur (4.4).



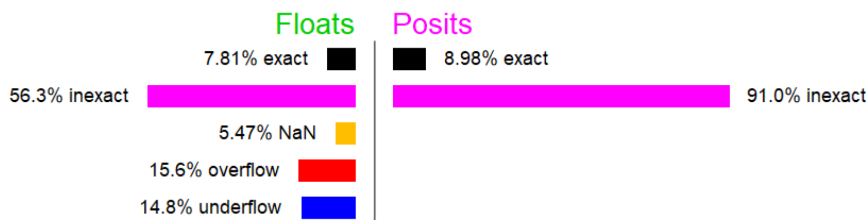
Figuur 4.4: Percentages exacte, inexacte en NaN waardes voor de functie $f(x) = \sqrt{x}$.

Het percentage exacte uitgedrukte waardes ligt opnieuw hoger bij posits. Verder is het ook weer het geval dat hoeveelheid waardes die geschat wordt hoger ligt bij posits en de hoeveelheid NaN waardes lager is. Dus posits geven weer meer waardes waarmee verder gerekend kan worden. Posits zijn dus ook geschikter voor de functie $f(x) = \sqrt{x}$ dan floats.

4.1.6 Representaties $f(x) = 2^x$

Een andere interessante functie om te bekijken is $f(x) = 2^x$. Omdat zowel floats als posits werken met machten van 2 is het interessant om te zien welke dit beter uit kunnen drukken. De resultaten van de vergelijking zijn grafisch weergegeven in figuur (4.5).

Wederom zijn posits de duidelijke winnaars. Ze hebben meer exacte waarden en meer afgeronde waarden waar verder mee gerekend kan worden. Ook hebben ze geen last van NaN, overflow en underflow waardoor meer waardes goed geschat kunnen worden.



Figuur 4.5: Percentages exacte, inexacte, NaN, overflow en underflow waardes voor de functie $f(x) = 2^x$.

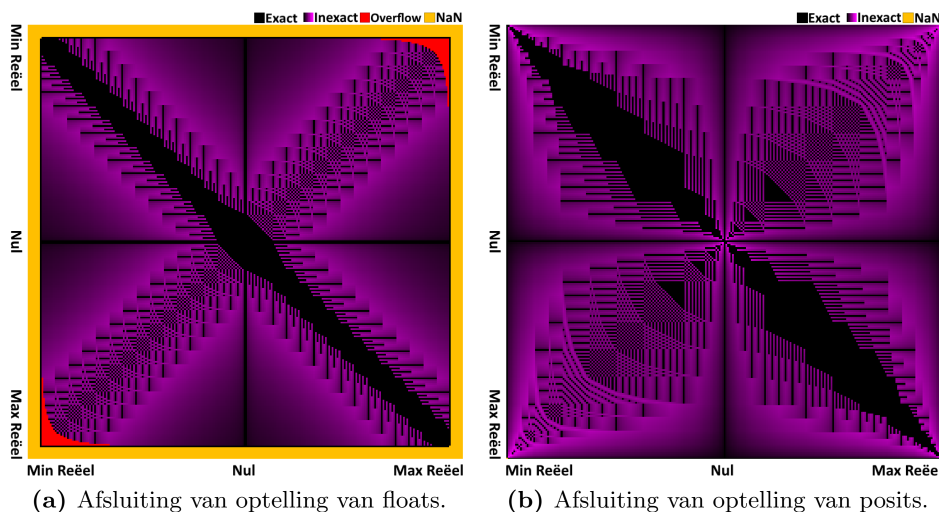
4.2 Basisfuncties met twee variabele waardes

In Posit Arithmetic [3] heeft Gustafson ook enkele basisoperaties met twee variabelen vergeleken. Voor deze vergelijkingen is ook Wolfram Mathematica gebruikt met dezelfde omgeving als in paragraaf (4.1). Gustafson vergelijkt ook 8-bits posits en floats op gelijke wijze. Alle figuren in paragraaf (4.2) worden dus verkregen uit Posit Arithmetic [3].

4.2.1 Optelling en aftrekken: $x + y$

Een van de meeste gebruikte basisoperaties zijn optellen en aftrekken. Omdat voor zowel posits als floats ieder getal een additieve inverse heeft, hoeft slechts één van deze operaties getest te worden. Om inzicht te krijgen in de afsluiting van optelling van posits worden twee posits bij elkaar opgeteld. Deze waarde wordt dan exact berekend en vervolgens afgeschat naar een posit. Het verschil wordt daarna vergeleken. Voor floats wordt op gelijke wijze te werk gegaan. De resultaten zijn grafisch weergeven in figuur (4.6). In zowel (a) als (b) is een duidelijk kruisvorm en een diagonale lijn van de vorm $y = -x$ te zien waar exacte waardes worden weergeven. Die op de x- en y-as kunnen verklaard worden omdat hier 0 opgeteld wordt bij een andere waarde, zowel floats als posits geven in dit geval een exacte waarde. Echter, in (a) is wel te zien dat deze lijn iets dikker is bij floats, dit komt doordat gebruik gemaakt wordt van gedenormaliseerde floats, waardoor de waardes beter weergeven kunnen worden. De diagonale lijn in de vorm $y = -x$ kan verklaard worden doordat hier getallen bij hun additieve inverse opgeteld worden. Dit zorgt voor een makkelijk te genereren antwoord, namelijk 0. Dit is een getal dat zowel floats als posits goed kunnen uitdrukken.

Nu de belangrijkste overeenkomsten bekeken zijn is het tijd om naar de verschillen te kijken. Het eerste wat opvalt is de dikke gele rand van NaN waardes die floats geven. Dit zijn waardes waar niet mee gerekend kan worden. Dit laat zien hoe groot gedeelte van floats een niet bruikbaar antwoord



(a) Afsluiting van optelling van floats.

(b) Afsluiting van optelling van posits.

Figuur 4.6: In (a) is de afsluiting van de operaties optellen weergeven van floats, in (b) is die van posits weergeven.[3]

oplevert, posits hebben hier bijna geen last van. (De ene pixel linksonder in b) Verder gaat er bij floats nog een gedeelte aan resultaten verloren aan overflow. Het is weliswaar mogelijk verder te rekenen met een overflow, maar de functionaliteit is zeer beperkt en kan voor onstabiele systemen zorgen, zoals de explosie van de Ariane 5.

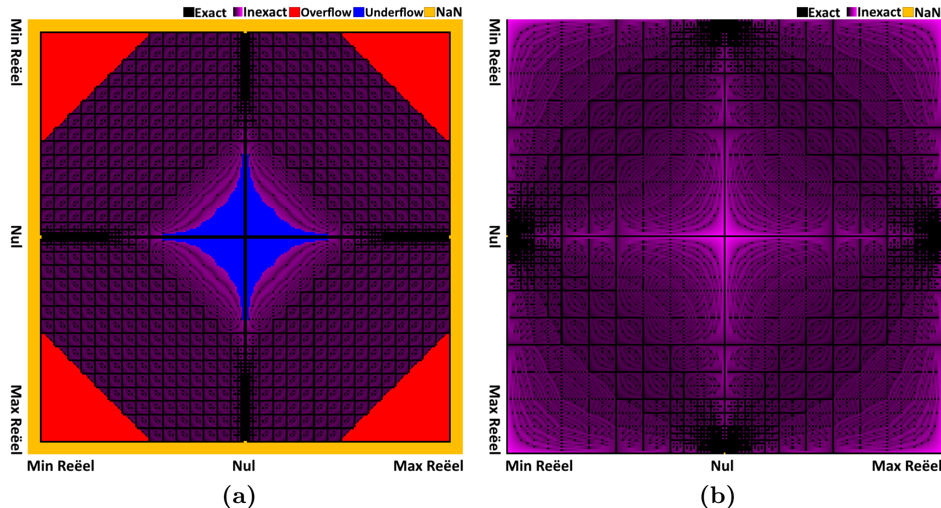
Tegen de verwachting uit paragraaf (4.1.2) in, zijn floats exacter rondom 0. Dit is te zien aan de dikke zwarte band rondom 0 in (a). Dit komt doordat gedenormaliseerde floats hier hun werk zeer goed doen. Echter, er moet wel opgemerkt worden dat het het percentage gedenormaliseerde floats bij een 8-bits float aanzienlijk hoger ligt dan bij grotere floats, zoals een 32-bits float.

Verder is in (b) goed het nut van regime bits te zien. De exact berekende waardes worden bij posits veel beter verspreid dan bij floats. De verwachte afrondfout is bij posits dus in het algemeen kleiner. Posits zijn voor optellen en aftrekken dus de duidelijke winnaar. Ze berekenen meer exact, schatten beter af en hebben bijna geen NaN waardes. Alleen rondom 0 zijn posits zwakker dan floats, maar de verwachting is dat dit verschil bij opschaling naar grote bit waardes wegvalt, omdat het percentage gedenormaliseerde floats hier afneemt.

4.2.2 Vermenigvuldiging: $x \cdot y$

De volgende basisoperatie waar naar gekeken wordt is vermenigvuldiging. Er is weer op gelijke wijze te werk gegaan als bij optelling. Dit resulteert in

figuur (4.7).



Figuur 4.7: In (a) is de afsluiting van de opraties vermenigvuldigen weergeven van floats, in (b) is die van posits weergeven.[3]

Voor zowel posits als floats is er weer een duidelijk kruispatroon om de x- en y-as. Dit komt wederom doordat bij vermenigvuldiging van 0 de exacte waarde 0 wordt gegeven zolang de andere waarde niet een vorm van ∞ is. Een andere overeenkomst is het raster patroon van exacte waardes. Bij floats is dit rooster echter kleiner. Dit betekent dat de hoeveelheid exacte waardes hoger ligt bij floats dan bij posits. De verdeling van getallen is voor een groot gedeelte dus beter dan bij posits, maar hier houden de voordelen van floats op.

Er is weer een rand van NaN waardes bij de floats waar bit representaties verloren gaan. Verder zijn grote hoeken rood afgekapt waar overflow optreed. Deze waardes zijn veel lastiger bruikbaar dan de schattingen die door posits gegeven worden. Verder vindt er ook in een grote ruit in het midden underflow plaats, waar posits een afschatting maken.

Bij vermenigvuldiging is de winnaar niet zo zwart-wit als bij optellen. De dichtheid van exacte waardes die floats bieden is een groot voordeel, maar het is lastig in te schatten of dit opweegt tegen de hoeveelheid slecht of niet bruikbare waardes die voorkomen uit underflow, overflow en NaN waardes.

4.3 Samenvatting

Met behulp van Mathematica zijn verschillende tests gedaan op 8-bits posits en floats. In alle berekeningen waar sprake was van slechts één variabele

waren posits duidelijk de winnaar. Niet alleen wordt er meer exact berekend, ook worden altijd afschattingen gemaakt waarmee verder gerekend kan worden. Dus wanneer bijvoorbeeld een overflow plaats zou vinden bij een float, rekent de posit gewoon door. Dit zal voor stabiliteit in systemen en berekeningen zorgen.

5 | Posits vs Floats 32-bits

In hoofdstuk (4) zijn in Mathematica 8-bits floats en posits grondig getest. Dit riep enkele vragen op over hoe de floats zouden werken in de praktijk. Daarom wordt in dit hoofdstuk wegens snelheid en toepasbaarheid geschakeld van programma. Er wordt gebruik gemaakt van Python in plaats van Mathematica. Verder wordt in dit hoofdstuk gekeken naar een 32-bits omgeving van zowel floats als posits.

5.1 Methode van Newton-Raphson

Enkele vragen die naar boven gekomen zijn: Wat voor effecten hebben underflow en overflow in praktijk? Hoe ernstig is de lage nauwkeurigheid bij optellingen van posits? Hoe goed werken posits met extreem grote en kleine waarden? Om deze vragen te beantwoorden wordt gekozen om te kijken naar bare bones praktijk voorbeeld: de methode van Newton-Raphson. Deze methode is gekozen omdat deze methode nulpunten opzoekt en er dus goed rondom deze waarde gerekend moet worden.

Bij ontwerpen van tests welke gebruik maken van de methode van Newton-Raphson zijn de twee belangrijkste keuzes die gemaakt moeten worden: het beginpunt x_0 en de functie $f(x)$ waar een nulpunt van gevonden moet worden. Als deze bekend zijn, dan geldt voor volgende iteraties

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{5.1}$$

Allereerst wordt de functie $f(x) = x^n - y$ bekeken. Deze functie is gekozen omdat hij relatief eenvoudig te implementeren is, maar ook omdat de toepassing zeer praktisch is. De nulpunten van deze functie zijn namelijk $\sqrt[n]{y}$, de methode wordt dus gebruikt om n-de machtswortels numeriek te berekenen.

Om de posit te testen zal gebruik gemaakt worden van de PCPosit package van Chung, S. Y. [9]. De package kan een posit maken met lengte $nbits$ en exponent lengte es. Verder kent de package de basisoperaties $+, -, \times, \%$.

In de volgende paragrafen worden alleen 32-bits posits en floats vergeleken. Pyhton gebruikt echter standaard 64-bits floats. Om toch gebruik te maken van 32-bits floats wordt de *numpy* package gebruikt. Dit betekent dat alle volgende code zal beginnen met:

```
from sgposit.pcposit import PCPosit
import numpy as np

nbits = 32 #aantal bits van de posits
es = 2     #aantal exponent bits van de posit
```

5.1.1 $f(x) = x^3 - 1$ en $x_0 = \frac{1}{4}$

De eerste test die gedaan wordt zal de 3^e machtswortel van 1 berekenen met startpunt 1/4. De verwachting is dat zowel floats als posits hier geen problemen zullen ervaren, want er wordt niet gerekend met te extreem grote of kleine getallen. Deze waardes worden als volgt geïnitialiseerd.

```
steps = 10 #hoeveelheid stappen die berekend worden

#beginwaarde x0 voor posits en floats
#hier is gekozen voor x0 = 1/4
posit_x0 = PCPosit(3*2**(nbits-4), mode='bits', nbits=nbits, es=es)
float_x0 = np.float32(1.0/4)

#getallen waar de n-de machtswortel van berekendwordt. In dit
#geval is n=3 en het getal waar de wortel van getrokken wordt is 1
n = 3
posit_root = PCPosit(2**(nbits-2), mode='bits', nbits=nbits, es=es)
float_root = np.float32(1)

#exacte antwoord in een 64 bit float
exact = float(1)
```

De gekozen testfunctie was $f(x) = x^n - y$. Omdat de gekozen package geen machtheffing heeft, wordt dit gedaan door n maal x met zichzelf te vermenigvuldigen. Daarna wordt y eraf getrokken.

```
def f(x):
    ans = x
    for i in range(1,n):
        ans *= x
    if isinstance(ans, PCPosit):
        return ans - posit_root
    return ans - float_root
```

Voor de methode van Newton-Raphson is ook een afgeleide nodig. Er geldt

$f'(x) = nx^{n-1}$, waar n een integer is. De package die wordt gebruikt heeft nog geen manier om integers te vermenigvuldigen met posits, deze zal dus zelf ontworpen moeten worden. Bij vermenigvuldiging van een integer n wordt gekozen om een posit n keer bij zichzelf op te tellen. De code is zo ontworpen dat zowel posits als floats gebruik zullen maken van deze implementatie, om te voorkomen dat floats een oneerlijke voorsprong zouden hebben.

```
def mul_int(x, n):
    ans = x
    for i in range(1,n):
        ans += x
    return ans
```

Nu vermenigvuldiging met integers mogelijk is wordt op gelijke wijze als bij $f(x)$ nu ook $f'(x)$ ontworpen.

```
def df(x):
    temp = x
    for i in range(1, n-1):
        temp *= x
    return mul_int(temp, n)
```

Nu zijn alle stukjes van de puzzel aanwezig en kan de daadwerkelijke Newton-Raphson methode worden geïmplementeerd.

```
def NewRap(x):
    return x - f(x) / df(x)
```

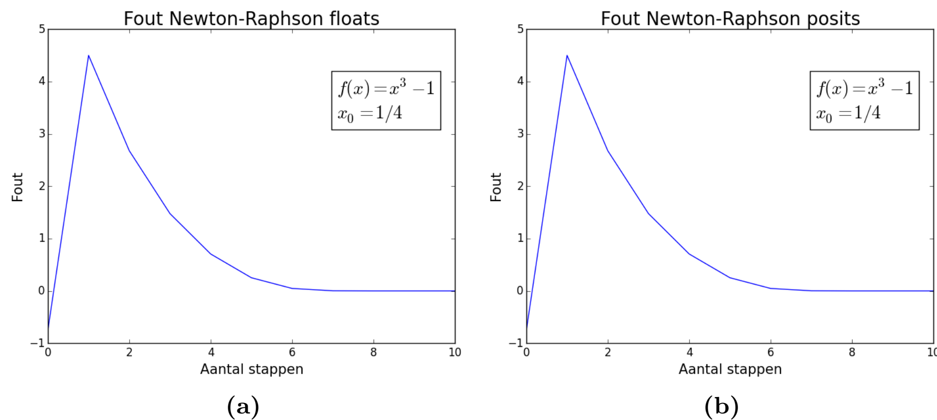
Nu deze methode ontworpen is, wordt het tijd om te vergelijken! Eerst is het echter nog handig om methodes te maken die enkele posits en lijsten posits omzetten naar floats. Dit, omdat op deze manier de data makkelijk vergeleken kan worden.

```
#Zet een posit om in een 64 bit float
def posit_to_float(p):
    (x,m) = p._fixedpoint()
    return x* 2**m

#Zet lijst met posits om naar een lijst met 64 bits floats
def posit_lst_to_float(lst):
    for i in range(0, len(lst)):
        lst[i] = posit_to_float(lst[i])
    return lst
```

Met behulp van deze methodes wordt gekeken naar $f(x) = x^3 - 1$, waar de eerste 10 stappen bekeken zullen worden. De resultaten zijn in tabel (5.1) en figuur (5.1) weergegeven.

Stap	Posit	Fout posit	Float	Fout float
0	$0,25$	$-0,75$	$0,25$	$-0,75$
1	$5,5$	$4,5$	$5,5$	$4,5$
2	$3,677\dots$	$2,677\dots$	$3,677\dots$	$2,677\dots$
3	$2,476\dots$	$1,476\dots$	$2,476\dots$	$1,476\dots$
4	$1,705\dots$	$7,053\dots\cdot 10^{-1}$	$1,705\dots$	$7,053\dots\cdot 10^{-1}$
5	$1,251\dots$	$2,514\dots\cdot 10^{-1}$	$1,251\dots$	$2,514\dots\cdot 10^{-1}$
6	$1,047\dots$	$4,715\dots\cdot 10^{-2}$	$1,047\dots$	$4,715\dots\cdot 10^{-2}$
7	$1,002\dots$	$2,091\dots\cdot 10^{-3}$	$1,002\dots$	$2,091\dots\cdot 10^{-3}$
8	$1,000\dots$	$4,358\dots\cdot 10^{-6}$	$1,000\dots$	$4,410\dots\cdot 10^{-6}$
9	$1,0$	$0,0$	$1,0$	$0,0$
10	$1,0$	$0,0$	$1,0$	$0,0$

Tabel 5.1: Geschatte waarde en gemaakte fout per stap van posits en floats bij de functie $f(x) = x^n$ en $x_0 = \frac{1}{4}$.



Figuur 5.1: Gemaakte fout bij de methode van Newton-Raphson van de functie $f(x) = x^3 - 1$ met beginwaarde $x_0 = \frac{1}{4}$. In (a) is de fout van floats weergegeven en in (b) die van posits.

Zowel posits als floats naderen de exacte oplossing zeer snel. Dit gebeurt bij beide al na 9 stappen. Dit komt omdat het getal dat benaderd wordt 1 is, wat handig rekent. Verder is de gekozen n klein wat zorgt voor relatief kleine afrondfouten. In figuur (5.1) is goed te zien dat posits en floats in deze test gelijk opgaan. Beide maken in de eerste stap een grote fout, waarna ze snel naar een fout van 0 bewegen. Deze fout in de eerste stap heeft niet te maken met de afrondfouten van posits of floats, maar met de methode van

Newton-Raphson zelf. De exacte waarde in stap 1 is namelijk:

$$x_0 - \frac{f(x_0}{f'(x_0)} = 5,5$$

De waarde die wordt gevonden is in beide gevallen dus exact berekend. In dit geval is er geen significant verschil tussen posits en floats.

5.1.2 $f(x) = x^3$ en $x_0 = \frac{1}{4}$

Als de te berekenen waarde 1 is, dan werken posits en floats zoals verwacht. Ze gaan gelijk op in dat geval, maar dit is een goede check dat de code goed ontworpen is. Nu bekijken we een functie die bijna gelijk is aan de vorige, echter is het antwoord dat we nu zoeken de 3^e machtswortel van 0. De functie is dus $f(x) = x^3$, met beginpositie x_0. Voor de test wordt grotendeels dezelfde code gebruikt als in paragraaf (5.1.1), met als verschil de beginwaarden.

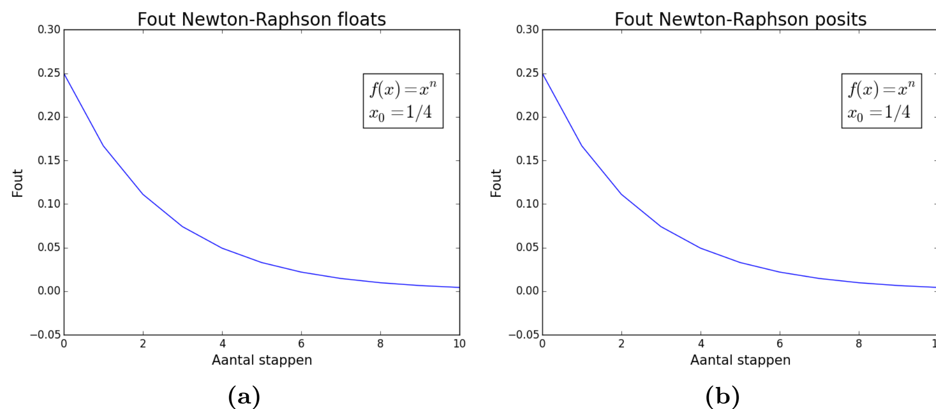
```
steps = 10 #hoeveelheid stappen die berekend worden

#beginwaarde x0 voor posits en floats
#hier is gekozen voor x0 = 1/4
posit_x0 = PCPosit(3*2**(nbits-4), mode='bits', nbits=nbits, es=es)
float_x0 = np.float32(1.0/4)

#getallen waar de n-de machtswortel van berekend wordt. In dit
#geval is n=3 en het getal waar de wortel van getrokken wordt is 0
n = 3
posit_root = PCPosit(0, mode='bits', nbits=nbits, es=es)
float_root = np.float32(0)

#exacte antwoord in een 64 bit float
exact = float(0)
```

Omdat posits rond 0 niet afronden omdat ze geen underflow kennen is de verwachting dat posits tot grotere nauwkeurigheid door kunnen rekenen dan floats. Want floats zullen vanwege underflow geen zeer kleine correcties kunnen maken, waar posits dit nog wel doen. De resultaten van deze test zijn weergeven in tabel (5.2) en figuur (5.2). De gezochte waarde is 0, dus de fout is gelijk aan de gevonden waarde.

Opnieuw gaan de posits en floats nek aan nek. Op een schaal van 10 stappen is nog geen verschil te merken, maar in tegensteling tot benadering van het punt 1 duurt het nu aanzienlijk langer om dicht bij punt 0 te komen. We zullen daarom het aantal stappen verhogen tot een maximale precisie is gehaald. In dit geval is dat 100 stappen. De resultaten zijn weergeven in tabel (5.3)

Stap	Fout posit	Fout float
0	$0,25$	$0,25$
2	$1,111\ldots\cdot 10^{-1}$	$1,111\ldots\cdot 10^{-1}$
4	$4,938\ldots\cdot 10^{-2}$	$4,938\ldots\cdot 10^{-2}$
6	$2,194\cdots\cdot 10^{-2}$	$2,194\ldots\cdot 10^{-2}$
8	$9,754\ldots\cdot 10^{-3}$	$9,754\ldots\cdot 10^{-3}$
10	$4,335\ldots\cdot 10^{-3}$	$4,335\ldots\cdot 10^{-3}$

Tabel 5.2: Geschatte waarde en gemaakte fout per stap van posits en floats bij de functie $f(x) = x^n$ en $x_0 = \frac{1}{4}$. Omdat de te vinden waarde 0 is, is de fout gelijk aan de waarde van de posit.



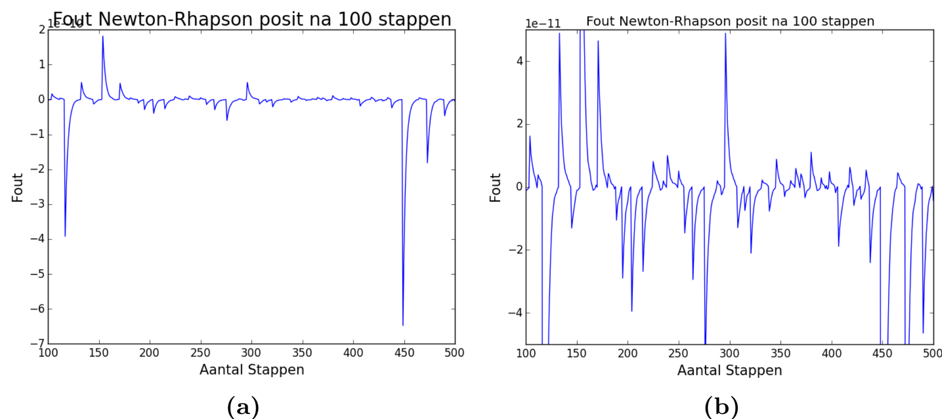
Figuur 5.2: Gemaakte fout bij de methode van Newton-Raphson van de functie $f(x) = x^3$ met beginwaarde $x_0 = \frac{1}{4}$. In (a) is de fout van floats weergegeven en in (b) die van posits.

Uit tabel (5.3) volgt dat floats een maximale nauwkeurigheid bereiken na ongeveer 90 stappen. Na deze hoeveelheid blijft de berekende waarde gelijk. Posits hebben echter meer moeite, niet alleen zijn ze in dit geval minder nauwkeurig, ze blijven ook van waarde veranderen en slaan zelfs van teken om! Dit gebeurt waarschijnlijk omdat posits geen underflow kennen. Hierdoor wordt nooit afgerond naar 0 en blijven er nieuwe waardes berekend worden. Om beter te begrijpen wat er gebeurd worden nog meer stappen bekeken. In figuur (5.3) is het gedrag van de posit test vanaf stap 100 t/m 500 grafisch weergeven. De posits blijven toch zeer sterk schommelen en stabiliseren niet naar een kleine marge. De uitschieters zijn, zoals in (a) te zien is, zeer groot. Verder ingezoomd in (b) kan gezien worden dat zelfs als we deze grote uitschieters buiten beschouwing laten de gevonden waardes sterk schommelen.

In het geval dat we de nulpunten van de functie $f(x) = x^3$ willen benaderen

Stap	Fout posit	Fout float
0	$0,25$	$0,25$
10	$4,335\ldots \cdot 10^{-3}$	$4,335\ldots \cdot 10^{-3}$
20	$7,518\ldots \cdot 10^{-5}$	$7,518\ldots \cdot 10^{-5}$
30	$1,303\ldots \cdot 10^{-6}$	$1,303\ldots \cdot 10^{-6}$
40	$2,259\ldots \cdot 10^{-8}$	$2,260\ldots \cdot 10^{-8}$
50	$3,923\ldots \cdot 10^{-10}$	$3,920\ldots \cdot 10^{-10}$
60	$6,366\ldots \cdot 10^{-12}$	$6,799\ldots \cdot 10^{-12}$
70	$-4,095\ldots \cdot 10^{-12}$	$1,179\ldots \cdot 10^{-13}$
80	$1,728\ldots \cdot 10^{-10}$	$2,035\ldots \cdot 10^{-15}$
90	$2,548\ldots \cdot 10^{-12}$	$8,534\ldots \cdot 10^{-16}$
100	$-1,143\ldots \cdot 10^{-12}$	$8,534\ldots \cdot 10^{-16}$

Tabel 5.3: Geschatte waarde en gemaakte fout per stap van posits en floats bij de functie $f(x) = x^n$ en $x_0 = \frac{1}{4}$. Omdat de te vinden waarde 0 is, is de fout gelijk aan de waarde van de posit.



(a) **(b)**

Figuur 5.3: Gemaakte fout bij de methode van Newton-Raphson van de functie $f(x) = x^3$ met beginwaarde $x_0 = \frac{1}{4}$. In (a) is de fout van posits weergegeven. In (b) wordt hetzelfde weergeven, maar verder ingezoomd.

met de methode van Newton-Raphson zijn floats dus duidelijke winnaars. Niet alleen zijn floats in dit geval nauwkeuriger, ze stabiliseren ook met behulp van underflow.

5.1.3 $f(x) = x^{120} - 2^{-120}$ en $x_0 = 4$

Als de gezochte waarde 0 moet zijn, dan zijn floats dus een betere optie, omdat ze gebruik maken van underflow. De vraag die nu gesteld wordt is dan ook, zijn underflow en overflow altijd zo nuttig? Om dit te onderzoeken wordt gekeken naar de functie $f(x) = x^{120} - 2^{-120}$. Het getal $2^{-120} = 16^{-30}$ is gekozen omdat dit het kleinste getal is dat nog door een posit met 32-

bits en 2 exponent-bits weergeven kan worden. Verder wordt voor $x_0 = 4$ gekozen om te kijken hoe posits en floats werken als ze met zowel grote als kleine getallen moeten rekenen. Het nulpunt van de functie ligt op $x = \frac{1}{2}$. Wederom wordt de code uit paragraaf (5.1.1) gebruikt met de volgende beginwaarden:

```
steps = 160 #hoeveelheid stappen die berekend worden

#beginwaarde x0 voor posits en floats
#hier is gekozen voor x0 = 4
posit_x0 = PCPosit(5*2**(nbits-4), mode='bits', nbits=nbits, es=es)
float_x0 = np.float32(4)


#getallen waar de n-de machtswortel van berekend
#wordt. In dit geval is n=120 en het getal waar de
#wortel van getrokken wordt is 2^(-120)
n = 120
posit_root = PCPosit(1, mode='bits', nbits=nbits, es=es)
float_root = np.float32(2**(-120))

#exacte antwoord in een 64 bit float
exact = float(0.5)
```

Er is gekozen voor 160 stappen, omdat na deze hoeveelheid stappen een duidelijke trend te zien is. De resultaten zijn weergeven in tabel (5.4)

Stap	Posit	Fout posit	Float	Fout float
0	$4,0$	$3,5$	$4,0$	$3,5$
20	$0,8673\ldots$	$0,3673\ldots$	NaN	NaN
40	$0,7337\ldots$	$0,2337\ldots$	NaN	NaN
60	$0,6207\ldots$	$0,1207\ldots$	NaN	NaN
80	$0,7108\ldots$	$0,2108\ldots$	NaN	NaN
100	$0,6009\ldots$	$0,1009\ldots$	NaN	NaN
120	$0,6871\ldots$	$0,1871\ldots$	NaN	NaN
140	$0,5797\ldots$	$0,0797\ldots$	NaN	NaN
160	$0,6644\ldots$	$0,1644\ldots$	NaN	NaN

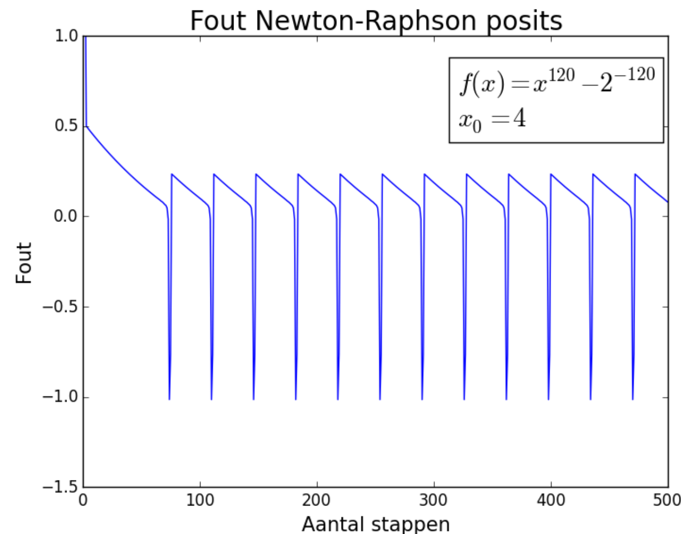
Tabel 5.4: Geschatte waarde en gemaakte fout per stap van posits en floats bij de functie $f(x) = x^{120} - 2^{-120}$ en $x_0 = 4$.

Floats komen na 1 stap een overflow tegen. De enige manier waarop hiermee verder gerekend kan worden op een nuttige manier is als er vervolgens gedeeld wordt door dit getal. Echter wat er gebeurd is dat zowel $f(x)$ als $f'(x)$ een overflow geven, dus de computer rekent hier dan met $\frac{\infty}{\infty}$, wat een NaN oplevert. Vervolgens kan er niet meer verder gerekend worden en worden

alle waarden na de eerste stap dus NaN.

Voor posits geldt dit niet, omdat er geen gebruik gemaakt wordt van overflow blijven ze gewoon doorrekenen. Hierbij worden wel grote fouten gemaakt en komen ze nooit echt in de buurt van het te berekenen getal $\frac{1}{2}$, maar blijven ze er omheen zweven. Om dit gedrag nader te onderzoeken wordt gekeken naar de eerste 500 stappen. Het resultaat is grafisch weergegeven in figuur (5.4).

Hier is te zien dat de fout steeds richting 0 gaat, maar op een gegeven



Figuur 5.4: Gemaakte fout bij de methode van Newton-Raphson van de functie $f(x) = x^{120} - 2^{-120}$ met beginwaarde $x_0 = 4$.

moment slaat het teken om en wordt de fout aanzienlijk groter. Dit proces blijft zich herhalen. In dit geval is bekend dat de exacte waarde $\frac{1}{2}$ moet zijn en kan de methode op een optimaal punt afgesloten worden. In praktijk zal deze waarde echter niet bekend zijn en is het lastig om de beste waarde te vinden. Het is bijvoorbeeld niet te achterhalen of de fout na 100 stappen kleiner is dan na 200 zonder een goede schatting vooraf te hebben. Posits rekenen in dit geval dus beter dan floats, alleen kan men zich afvragen of de gevonden resultaten nog wel nut hebben met een fout van deze omvang.

5.1.4 $f(x) = x^{120} - 2^{-120}$ en $x_0 = 1$

In paragraaf (4.1) konden floats niet verder rekenen wegens overflows. Nu is de vraag of floats een betere benadering kunnen maken als een kleinere startwaarde x_0 gekozen wordt en er geen overflow plaatsvindt. Als startwaarde wordt $x_0 = 1$ gekozen, omdat dit geen overflow oplevert. De resultaten zijn weergeven in tabel (5.5)

Stap	Posit	Fout posit	Float	Fout float
0	$1,0$	$0,5$	$1,0$	$0,5$
10	$0,9197\dots$	$4,197\dots\cdot 10^{-1}$	$0,9197$	$4,197\dots\cdot 10^{-1}$
20	$0,8458\dots$	$3,458\dots\cdot 10^{-1}$	$0,8458$	$3,458\dots\cdot 10^{-1}$
30	$0,7779\dots$	$2,779\dots\cdot 10^{-1}$	$0,7779$	$2,779\dots\cdot 10^{-1}$
40	$0,7155\dots$	$2,155\dots\cdot 10^{-1}$	$0,7155$	$2,155\dots\cdot 10^{-1}$
50	$0,6581\dots$	$1,581\dots\cdot 10^{-1}$	$0,6580$	$1,580\dots\cdot 10^{-1}$
60	$0,6053\dots$	$1,053\dots\cdot 10^{-1}$	$0,6052$	$1,052\dots\cdot 10^{-1}$
70	$0,4850\dots$	$-1,499\dots\cdot 10^{-2}$	$0,5566$	$5,667\dots\cdot 10^{-2}$
80	$0,6931\dots$	$1,931\dots\cdot 10^{-1}$	$0,5121$	$1,212\dots\cdot 10^{-2}$
90	$0,6375\dots$	$1,375\dots\cdot 10^{-1}$	$0,5$	0
100	$0,5842\dots$	$8,4273\dots\cdot 10^{-2}$	$0,5$	0

Tabel 5.5: Geschatte waarde en gemaakte fout per stap van posits en floats bij de functie $f(x) = x^{120} - 2^{-120}$ en $x_0 = 1$.

Net als bij de vorige beginwaarde valt op dat posits nooit echt in de buurt komen. Ze blijven zich hetzelfde gedragen als in het vorige voorbeeld. Floats hebben daarentegen deze keer geen last van overflow en kunnen deze keer dus wel meerekenen. Niet alleen komen floats in de buurt van $\frac{1}{2}$ ze komen er zelfs exact op uit! Dit komt waarschijnlijk doordat floating points gebruik maken van underflow waardoor de correcties die gedaan worden kleiner worden dan bij posits. Het is wel heel opvallend dat dit zo snel gaat als in tabel (5.5) te zien is. In de eerste 80 stappen is de fout nog vrij groot, maar na 90 stappen is deze opeens gelijk aan 0. Om dit verschijnsel nader te onderzoeken wordt nog eens in detail gekeken naar de laatste stappen voordat het antwoord exact gegeven wordt. De resultaten zijn weergeven in tabel (5.6).

Stap	Float	Fout float
80	$0,5121\dots$	$1,212\dots\cdot 10^{-2}$
81	$0,5081\dots$	$8,101\dots\cdot 10^{-3}$
82	$0,5044\dots$	$4,482\dots\cdot 10^{-3}$
83	$0,5017\dots$	$1,718\dots\cdot 10^{-3}$
84	$0,5003\dots$	$3,075\dots\cdot 10^{-4}$
85	$0,5000\dots$	$1,096\dots\cdot 10^{-5}$
86	$0,5$	$0,0$

Tabel 5.6: Geschatte waarde en gemaakte fout per stap van floats bij de functie $f(x) = x^{120} - 2^{-120}$ en $x_0 = 1$.

Zelfs als we verder inzoomen gaat het nog steeds heel snel. De fout gaat van $1,096\dots\cdot 10^{-5}$ opeens naar 0, terwijl het hiervoor veel langzamer ging.

Het is waarschijnlijk dat dit een toevallige afronding is, maar om hier een duidelijke conclusie over te trekken zal meer onderzoek gedaan moeten worden. Wat wel duidelijk is, is dat zelfs zonder deze sprong naar exactheid floats daarvoor ook al een schatting maakten die veel nauwkeuriger is dan bij posits. Posits gaven in beiden gevallen een zeer grove schatting van de waarde die gevonden moest worden. Het kan voor sommige doeleinden zeker nuttig zijn, een schatting is soms beter dan geen. Echter was het in dit geval beter voor floats te kiezen en de startwaarde aan te passen, dit geeft een aanzienlijk beter resultaat.

5.2 Samenvatting

In dit hoofdstuk is gekeken naar een praktijk toepassing van posits. De methode van Newton-Raphson is gebruikt om verschillende nulpunten uit te rekenen. Door de verkregen 8-bit resultaten werd verwacht dat posits hier uitermate geschikt voor zouden zijn. Ze zijn nauwkeuriger rond 0 en kunnen zowel met hele grote als kleine getallen rekenen. Dit is echter niet wat bleek uit de resultaten.

Voor normale gevallen werkten posits en floats niet merkbaar anders qua nauwkeurigheid, echter als we gingen kijken naar extreme gevallen, dan wonnen floats. Deze waren zeer nauwkeurig en kwamen zelfs op exacte waarden uit als het algoritme lang genoeg door bleef rekenen. Daarentegen bleven posits met een enorme fout in een loop zitten. Dit komt waarschijnlijk vanwege underflow die plaatsvindt bij deze extreme gevallen waar met zowel grote als kleine getallen gerekend wordt.

Wat niet vergeten moet worden is dat posits wel altijd een schatting konden maken van een waarde. Ook al zat deze soms ver van het te berekenen getal af, het is nog steeds nauwkeuriger dan de NaN waarde die floats gaven.

6 | Conclusies

Het nieuwe datatype Unum Type III Posits zou een mogelijke vervanging van floating points moeten zijn. De verwachting was dat door middel van de regime bits een groter bereik geleverd kon worden dan bij floats. Verder is er geen verspilling van bits doordat er geen NaN waardes zijn. Hierdoor zijn er maar 3 mogelijke vormen van posits ten opzichte van de 5 van floats. Dit zorgt ervoor dat posits sneller aangemaakt kunnen worden dan floats.

Allereerst zijn 8-bits floats en posits bestudeerd. Dit is gedaan met behulp van Mathematica vanwege de hoge precisie die dit programma te bieden heeft. Hieruit volgde dat posits een duidelijke winnaar zijn ten opzichte van floats. In alle metingen waar een enkele variabele in voorkwam, waren posits exacter en konden ze meer waarden afschatten. Dus niet alleen zijn posits op een 8-bits schaal nauwkeuriger, ook geven ze meer waarden waar verder mee gerekend kan worden. De regime bits zorgen ervoor dat de afstand tussen twee posits meekrimpt met de relatieve fout. Dit zorgt ervoor dat de relatieve fout in de buurt van 0 kleiner is dan die van floats.

Posits hebben geen last van overflow of underflow en dit riep de vraag op of dit in de praktijk ook echt zo nuttig was. Daarom zijn enkele praktijkvoorbeelden getest. Hiervoor zijn 32-bits floats en posits gebruikt en om het zo dicht mogelijk bij de praktijk te houden en om rekentijden klein te houden is gekozen verder te gaan met Python. De verwachting was dat posits uitermate geschikt waren voor deze methode, aangezien het 8-bit geval posits sterk waren rondom 0 en de methode van Newton-Raphson zoekt nulpunten. Dit bleek echter niet het geval. In de niet extreme gevallen was er nagenoeg geen verschil te merken tussen floats en posits. Echter, wanneer met extreem grote en kleine getallen werd gerekend waren floats duidelijke winnaars. Wel moet gezegd worden dat dit alleen geldt zolang als floats de berekeningen konden maken. Overflow bleek een naar onderdeel te zijn van floats, dat verder rekenen vaak onmogelijk maakt. Underflow daarentegen bleek zeer nuttig te zijn. Het is waarschijnlijk dat de hoge precisie die gehaald wordt door floats onder andere te danken is aan underflow, dit zorgt ervoor dat zeer kleine correcties gemaakt konden worden in iedere stap.

Zoals al eerder gezegd hadden floats wel soms last van overflow. Omdat posits nooit afronden naar oneindig hebben ze hier geen last van en kon er doorgerekend worden. Dit zorgt ervoor dat er altijd een schatting gegeven wordt, zelfs als floats na één rekenstap al een NaN geven.

De doeleinden waarvoor posits geschikt lijken te zijn, zijn toepassingen waarin getallen in kleine hoeveelheden bits opgeslagen moeten worden. Ze zijn op dit gebied een stuk krachtiger dan floats, omdat wanneer een float een kleine lengte heeft, het percentage representatie wat een NaN geeft relatief hoog ligt. Voor dit soort applicaties lijken posits dus voor de hand te liggen. Verder is er in het 32-bit model over het algemeen weinig verschil, maar wanneer het mogelijk is lijken floats beter te werken. Floats werken nauwkeuriger onder extreme omstandigheden dan posits, maar kunnen soms wegens overflow of underflow toch geen berekening maken. In deze gevallen, wanneer een afschatting voldoende is, kunnen posits ook gebruikt worden. Doordat ze geen overflow en underflow hebben blijven ze ook onder extreme omstandigheden doorrekenen, al kan er dan niet veel gezegd worden over de nauwkeurigheid van het behaalde resultaat.

De conclusie die getrokken kan worden uit bovenstaande beantwoording van de deelvragen is dat Unum Type III Posits geen geschikte drop-in replacemant zijn van de huidige IEEE 754 Floating Point Standard (floats).

7 | Discussie

In dit verslag is gekeken naar zeer specifieke vormen van floats en posits. De floats zijn in de IEEE 754 standaard gekozen. Voor posits bestaat echter nog geen gestandaardiseerde vorm en is er gekozen voor een exponent van 2 bits voor zowel de 8-bits variant als de 32-bits variant. Er kan nog meer geëxperimenteerd worden met andere groottes van de exponent. Een van de redenen dat posits zo nauwkeurig rondom 0 waren in de 8-bit variant is omdat hier voor een exponent van 2-bits is gekozen. Dit is ten opzichte van 8 erg groot in vergelijking tot 32, waar ook een 2-bits exponent gekozen was. Dit is gedaan om het onderzoek en advies van Gustafson [3] aan te houden. Gustafson pleit voor een 2-bits exponent voor een 32-bits posit, omdat dit in de meeste gevallen tot hogere nauwkeurigheid zal leiden. Echter, er zou geëxperimenteerd kunnen worden met wat de effecten zijn wanneer een grotere exponent wordt gekozen. De nauwkeurigheid rondom 0 zou dan samen met het bereik groter worden. De relatieve fout over de rest van de getallenlijn wordt echter wel groter. Er zal nog nader onderzoek gedaan moeten worden naar de effecten die een andere exponent zal hebben.

Aangezien posits in vergelijking met floats in een 8-bits omgeving aanzienlijk beter konden rekenen dan in een 32-bits omgeving is het ook interessant om vervolgonderzoek te doen naar een 16-bits omgeving. Het zou kunnen zijn dat met een exponentgrootte van 2-bits er een “best of both worlds” -scenario optreedt. Een scenario waar de mogelijkheid van doorrekenen benut kan worden, vanwege het ontbreken van overflow en underflow. De reden hiervoor is dat met 16 bits veel meer nuttige berekeningen gedaan kunnen worden dan met 8 bits. Verder zal de precisie rond 0 waarschijnlijk gelijk zijn aan die van floats. Dit was in de 32-bits variant de voornaamste tegenvaller van posits ten opzichte van floats. Als deze in een 16-bits variant gelijk is, zullen posits beter werken dan floats.

Ook is in het 32-bits geval slechts de methode van Newton-Raphson onderzocht met als functie $f(x) = x^n - y$. De methode van Newton-Raphson was gekozen, omdat dit een veelgebruikte methode is en omdat er gebruik gemaakt wordt van basisoperaties als optellen en delen. De functie $f(x) = x^n$

was gekozen wegens simpliciteit en omdat het makkelijk is heel grote waardes en heel kleine waardes te creëeren door een grote n te kiezen. In een vervolgonderzoek zouden meer n en y waardes getest kunnen worden. Ook zouden andere functies gemaakt en getest kunnen worden om op deze wijze meer praktijkvoorbeelden te testen. Verder zou ook een andere numerieke methode dan die van Newton-Raphson gekozen kunnen worden. Zo kan bijvoorbeeld een numerieke methode als de Runge-Kutta methode geïmplementeerd worden en getest worden. Deze laat waardes snel convergeren en zou verschillen in nauwkeurigheid waarschijnlijk snel aantonen.

Ondank dat posits geen algemene vervanging zijn van floats, lijken ze nog steeds voordelen te hebben ten opzichte van floats. De toepasbaarheid van deze voordelen zal nog verder onderzocht moeten worden.

Bibliografie

[1] A space error: $370 million for an integer overflow. (2016, 5 september). Geraadpleegd op 18 september 2018, van https://hownot2code.com/2016/09/02/a-space-error-370-million-for-an-integer-overflow/

[2] Hollasch, S. (2005). *IEEE Standard 754 Floating Point Numbers*. Geraadpleegd van http://www.dcc.ufrj.br/~luziane/1B_IEEE%20Standard%20754_detalhada.pdf

[3] Gustafson, J. L. (2017). *Posit Arithmetic*. Geraadpleegd van https://posithub.org/docs/Posits4.pdf

[4] Gustafson, J. L., & Yonemoto, I. (2017). *Beating Floating Point at its Own Game: Posit Arithmetic*. Geraadpleegd van http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

[5] Alexandrov, O. (2008, 16 september). [Rele projectieve lijn] [Illustratie]. Geraadpleegd op 8 juli 2018, van https://nl.wikipedia.org/wiki/Re%C3%ABle_projectieve_lijn#/media/File:Real_projective_line.svg

[6] Gustafson, J. L. (2015). *The End of Error: Unum Computing* (24e ed.). Boca Raton, Verenigde Staten: CRC Press.

[7] Gustafson, J. L. (2017, 10 oktober). Posits4 [Wolfram Mathematica Code]. Geraadpleegd op 24 april 2018, van https://posithub.org/docs/Posits4.nb

[8] Munafo, R. (2018, 27 augustus). Survey of Floating-Point Formats at MROB. Geraadpleegd op 14 september 2018, van http://www.mrob.com/pub/math/floatformats.html

[9] Chung, S. Y. (2018, 7 januari). sgpositpy [Python Code]. Geraadpleegd op 17 juli 2018, van https://github.com/xman/sgpositpy