A University of Sussex PhD thesis

Available online via Sussex Research Online:

http://sro.sussex.ac.uk/

This thesis is protected by copyright which belongs to the author.

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the Author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the Author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Please visit Sussex Research Online for more information and further details

Analysing and Bounding Numerical Error in Spiking Neural Network Simulations

James Paul Turner

Ph.D. Computer Science
University of Sussex
April 2019

Acknowledgements

I would first like to dearly thank and dedicate this study to my parents, Sue and Tony. This thesis would not have been possible without them. Thank you!

I would also like to thank my supervisors, Professor Thomas Nowotny and Doctor Luc Berthouze, for their brilliant advice, ad hoc white-board math sessions, admirable patience and continuous enthusiasm throughout the project. Thank you!

This study was funded by a doctoral scholarship from the School of Engineering and Informatics, at the University of Sussex.

Declaration

I hereby declare that this thesis has not been, and will not be, submitted in whole or in part to another University for the award of any other degree.

Signature: .....

JAMES PAUL TURNER
PH.D. COMPUTER SCIENCE
UNIVERSITY OF SUSSEX
ANALYSING AND BOUNDING NUMERICAL ERROR
IN SPIKING NEURAL NETWORK SIMULATIONS

This study explores how numerical error occurs in simulations of spiking neural network models, and also how this error propagates through the simulation, changing its observed behaviour. The issue of non-reproducibility in parallel spiking neural network simulations is illustrated, and a method to bound all possible trajectories is discussed. The base method used in this study is known as mixed interval and affine arithmetic (mixed IA/AA), but some extra modifications are made to improve the tightness of the error bounds.

I introduce Arpra, my new software, which is an arbitrary precision range analysis library, based on the GNU MPFR library. It improves on other implementations by enabling computations in custom floating-point precisions, and reduces the overhead rounding error of mixed IA/AA by computing in extended precision internally. It also implements a new error trimming technique, which reduces the error term whilst preserving correct boundaries. Arpra also implements deviation term condensing functions, which can reduce the number of floating-point operations per function significantly. Arpra is tested by simulating the Hénon map dynamical system, and found to produce tighter ranges than those of INTLAB, an alternative mixed IA/AA implementation.

Arpra is used to bound the trajectories of fan-in spiking neural network simulations. Despite performing better than interval arithmetic, the mixed IA/AA method used by Arpra is shown to be inadequate for bounding the simulation trajectories, due to the highly nonlinear nature of spiking neural networks. A stability analysis of the neural network model is performed, and it is found that error boundaries are moderately tight in non-spiking regions of state space, where linear dynamics dominate, but error boundaries explode in spiking regions of state space, where nonlinear dynamics dominate.

Contents

Chapter 1

Introduction

Computer simulations are valuable tools for understanding the behaviour of complicated natural systems. They make predictions and help us form hypotheses, without the need for expensive or impossible experiments. Computing these models on computers, however, means accepting a small amount of imprecision in our results. This imprecision comes from things like rounding errors when performing floating-point arithmetic on computers, truncation errors when using approximate numerical methods like Euler integration, and even errors in input data from measuring instruments. Although small, when accumulated over time, these errors often influence the overall behaviour of the software, sometimes with dramatic effect. In a famous example, unchecked rounding error caused havoc at the Vancouver Stock exchange. The gradual accumulation of rounding errors, between January 1982 to November 1983, was causing stocks to loose around 25 points per month, according to the Wall Street Journal and the Toronto Star [1]. Another more deadly example of rounding error disaster involved the failure of a Patriot surface-to-air missile defence system [2], allowing a Scud missile to pass through intact, resulting in the death of 28 military personnel. Such occurrences have spurred a lot of interest in analysing how various error sources will propagate through computer programs. In the software tester's case, they need to know how these errors change the behaviour of the software, and determine how reliable the resulting output is. In this study, we are specifically interested in how numerical error changes the behaviour of spiking neural network simulations (henceforth SNN simulations), modelled as non-linear dynamical systems.

Large non-linear dynamical systems are rarely simple enough to solve analytically, so theorists must instead resort to numerical approximation; they test their hypotheses by simulating them using computers. Simulations like these often involve binary representation and arithmetic of real numbers, us-

ing the Floating-Point Unit (FPU) of a processor, implementing the IEEE-754 standard for binary floating-point arithmetic [3], [4]. In such a representation, the arithmetic operations are of an imperfect accuracy, and also the arbitrary-precision input and intermediate values must be transformed in a lossy manner to fit inside a finite size binary word, often leaving a small error in each computed result. This error is cumulative, meaning a simulation's trajectory may be perturbed a little further off-course after each and every successive floating-point operation. The simulation trajectory is also sensitive to the order that the floating-point instructions are executed, since rounding of computed values removes the associativity of operations like addition and multiplication. Loss of associativity is especially problematic in multi-threaded and General Purpose Graphics Processor Unit (GPGPU) code, which utilises multiple FPUs simultaneously, all operating on the same data, with no guarantee on the order which they will begin or finish an operation. To further complicate matters, it is often the case that there are subtle differences in the way floating-point arithmetic is implemented across architectures [5], and additional differences in the way compilers optimise floating-point arithmetic code [6]. For the reasons given above, it can be difficult to, for instance, predict the divergence of a simulation's trajectory from the mathematically 'correct' trajectory, or even to test the validity of a GPGPU simulation's output against that of a reference implementation on another architecture.

This study aims to apply current floating-point error testing techniques to investigate numerical error in realistic SNN simulations, in serial and parallel computer architectures. Specifically, this study will investigate how both rounding and truncation errors propagate through the computation, and will also evaluate the effectiveness of the chosen Affine Arithmetic method for this problem. In the remaining sections of this chapter, the first section introduces SNN simulations, and the results reproducibility problem. The next section reviews how real numbers are represented digitally, and discusses the approximate nature of this representation. The following section reviews numerical error, and what factors can influence it. The final two sections review range analysis methods which can be used to bound the worst-case numerical error in a dynamical systems simulation.

1.1 Spiking Neural Network Simulations

This project started out of necessity to verify the results of an SNN simulation library, called GeNN (GPU enhanced Neural Networks) [7], against the results of a standard CPU implementation. GeNN is a code generation framework which generates C++ / CUDA source code for simulating realistic simulations

of neural circuits. It makes use of the massive pool of processor cores on modern NVIDIA Compute Unified Device Architecture (CUDA) GPU devices [8] to execute simulation code in parallel, much faster than the equivalent serial simulation. One of the difficulties of multi-threaded programming is that there is no guarantee on the order which a pool of threads will finish working in. This is a problem because reordering floating point instructions can influence rounding error propagation, due to the loss of associativity in floating-point arithmetic.

An example of where instruction reordering seems to affect a GeNN simulation's trajectory is in the summation of input currents for neurons. During the neuron model integration phase, each neuron is executed as an independent thread, running on its own core of a CUDA GPU device. Although large, the finite pool of memory and processing resources on a CUDA device often means that not all of the neurons of a complete model network can be simulated at the exact same moment, and must instead be pipelined into the CUDA device in groups called 'blocks' [8]. The neuron CUDA blocks are organised such that the threads of neurons which project and receive synapses from the same source neuron groups are always simulated at, or around, the same time.

Next, after the dynamics of the neuron model have been simulated, each thread makes a check to determine whether or not its neuron has exceeded the spike threshold potential. If that is the case, the thread records its index number of the neuron into an array, which is then transmitted to target neuron groups during the subsequent spike propagation phase. In order to ensure that each thread can execute independently, in parallel, without being bogged down waiting for other threads, it is important to allow each thread to record the index number of its spiking neuron immediately. This means that the index number of neurons is recorded in the same order that each thread finishes its work, rather than having threads waiting patiently for all lower indexed threads to integrate their neurons and record their spikes before recording their own. As a result, the order of the list of neuron indices which is transmitted to postsynaptic neuron groups cannot be predicted, even between runs of the same simulation on the same CUDA device, as figure 1.1 illustrates.

Finally, in the spike propagation phase of the simulation, which transmits presynaptic neuron spikes to postsynaptic neuron targets, each postsynaptic neuron computes a sum of input currents from all spiking presynaptic neurons, weighted by the conductance of each corresponding synapse. The unpredictable order that incoming spike lists are populated, in conjunction with the rounding error incurred by each floating-point addition, means that the total input current for a given neuron could be different with each identical simulation run. The effects of summand ordering are discussed later, but suffice it to say that this variance is enough to cause the actual membrane potential values in identical

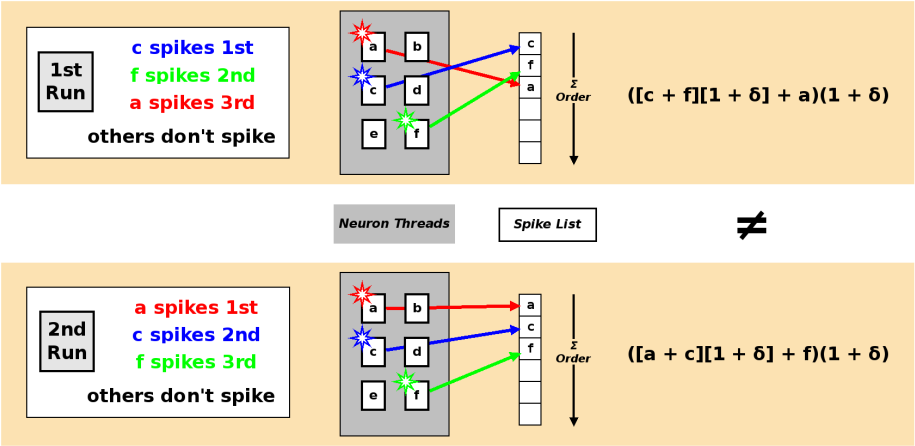

Figure 1.1: Different spike list order for the same simulation on the same GPU, due to unpredictable thread finishing order. A star indicates a GPU core with a spiking neuron. In the first run (above), thread c finishes and writes its index to the spike list first, followed by thread f, then a. On the second run (below), thread a finishes and writes its index to the spike list first, then c, then f.

simulations to vary slightly between runs on identical hardware. This effect is shown empirically in the experiments of chapter 2. This divergence can be worrying for researchers from non-computational backgrounds, who may wonder whether or not this a bug in the software or hardware. How can these researchers be reassured that their simulations are operating within expected boundaries, given their results are varying across simulations in identical environments, let alone the variance from different hardware and software environments?

Before making any claims on the reliability of results in GPU simulations, we must first compute the error bounds of all results in an equivalent simulation using range analysis methods, as discussed later. To determine these error bounds, with or without deterministic summation orders, all rounding error of the computation must be taken into consideration. The following sections describe how real numbers are represented on computers, how the rounding error of an operation is computed, and how these rounding errors propagate through subsequent operations.

1.2 Number Representation

There are a few ways to represent real numbers on computers. We will consider two of them; fixed-point and floating-point representations. Error calculation and floating-point rounding modes are also discussed.

1.2.1 Fixed-Point Numbers

Perhaps the most obvious way of representing real numbers on a computer is to store them using a set of integer variable types, with a different order of magnitude being represented by each variable type. For example, one might have an integer variable x which represents a real number f of the form:

$$f = x \cdot b^k. \tag{1.1}$$

where b is the base of the number system, and k is some constant exponent, which defines the base-b order of magnitude that the variable will represent. This is known as a fixed-point number system. Another way of describing this system is that the constant k defines the location of the radix point inside the integer variable being used to represent the real numbers. Fixed-point arithmetic, as used by the SpiNNaker neuromorphic hardware project [9], can be performed quickly and easily with existing integer arithmetic hardware. However, the low relative range of any given fixed-point variable type makes it unsuitable in simulation software in general, where arithmetic operands can be many orders of magnitude apart.

A possible solution to this problem would be to use multiple fixed-point variable types in numerical code. For instance, one might store variables that are usually of low-magnitude in a fixed-point format with $k = -3$, but store usually higher-magnitude numbers in fixed-point formats with $k = 3$. However, such a strategy does not make efficient use of memory for variables whose magnitude varies greatly; integer overflow occurs when values are large but k is small, and the fixed-point format is not as precise as it could be when values are small but k is large. One could take this strategy further by using structures of multiple fixed-point types to represent each variable, where each type stores digits of different significance. Whilst this approach guarantees certain precisions across a predetermined range of values, it is inflexible and can be memory inefficient, since the number of fixed-point types used to store each variable must be determined in advance, and using more fixed-point types per variable requires more memory. As a reasonable compromise, the floating-point format is often used for real-valued variable representation in numerical simulations instead.

1.2.2 Floating-Point Numbers

The floating-point number system works around these problems by allowing the exponent k to vary as the number changes magnitude. The catch is that there is now more data to store per number, since k now also needs to be recorded, in addition to x. However, if f becomes too large or too small for x, then

k is dynamically incremented or decremented to accommodate. Thus, in the floating-point number system, a number $f \in \mathbb{F} \subset \mathbb{R}$ can be specified with:

$$\begin{aligned} f &= s \cdot b^e \\ s &\in [1, b). \end{aligned} \tag{1.2}$$

where b is the base, and $e \in [e_{\min}, e_{\max}]$ is the bounded variable exponent. s is known as the significand of the floating-point number, which holds exactly p significant digits, and is always normalised. For most floating-point numbers, known as 'normal' numbers, the significand is normalised to within the interval $[1, b)$ to ensure the real number space is represented uniformly at each value of exponent. The 'subnormal' numbers are an exception to this rule; for numbers with the lowest value of exponent, $e_{\min}$, the significand is normalised to within the wider interval $[0, b)$ to fill the space between zero and the lowest representable normal number. Without this so-called gradual underflow, any input number below the smallest representable normal number would be said to have underflowed - i.e. is too small to be represented - and would be flushed to zero, resulting in a large round-off error.

A floating-point number is encoded as a binary word as follows: the first part is a single bit representing the sign of the number, where zero indicates a positive number, and one indicates a negative number. The next part is a sequence of bits, interpreted as a signed integer, representing the variable exponent. The third part is a sequence of bits representing the significand as a normalised fixed-point number. Since the first bit of the significand is always one for normal numbers, a single extra bit of precision can be gained by reading the significand with an implicit leading one bit, rather than storing it explicitly. Subnormal numbers are an exception to this rule because the first bit of a number's significand is not necessarily one. The leading bit, which need not be stored for normal numbers, must now be stored explicitly. Although losing one bit of significand in subnormal numbers costs half of their relative precision, gradual underflow guarantees that the rounding error is still well below that of an actual underflow error.

There have been many floating-point implementations in the past, with varying precision and accuracy. A common problem with running and verifying numerical simulation software was that the data resulting from a simulation on one architecture could be completely unlike the data from the same simulation running on another architecture. Even when run on the very same machine, a simulation compiled with one compiler could produce very different results from the same simulation compiled with another compiler. In an effort to standardise floating-point arithmetic across architectures and software, the

original IEEE-754 standard [3] was drafted in 1985, henceforth referred to as IEEE-754-1985. It defined standard binary data formats for floating-point numbers, including the 32 bit single-precision and 64 bit double-precision formats, respectively corresponding to the **`float`** and **`double`** types in the C-like programming languages. For the single-precision format, p = 24. Not counting the implicit leading one, this means that 23 bits are used for the single-precision significand, 8 bits are used for the signed integer exponent, and the remaining bit determines the number's sign. For the double-precision format, $p = 53$, so the significand uses 52 bits, the exponent uses 11 bits, and the sign is held in the remaining bit.

1.2.3 Approximation Error

Because the representation of real numbers is discrete and approximate, small errors are introduced into the computation whenever the FPU and math library are utilised. These errors can appear both when the simulation input data is converted into its floating-point representation, and also whenever the data is operated on by the arithmetic hardware and software. When storing a real number in computer memory, if the number can not be exactly represented as a floating-point number, it is necessary to round it to another nearby number which can be exactly represented.

There are three widely used ways to measure numerical error [10]; the first of the measurements are absolute error (1.3) and relative error (1.4), given below, where f is the computed floating-point value, and r is the exact value. (1.4) assumes that r is non-zero.

$$\text{error}_{\text{abs}}(f, r) = |f - r| \tag{1.3}$$

$$\text{error}_{\text{rel}}(f, r) = \left| \frac{f - r}{r} \right| \tag{1.4}$$

In words, the absolute error is a measure of how different a floating-point number is from what it is supposed to be, while the relative error is a measure of the absolute error relative to the magnitude of the exact number. Here, the relative error is more useful to us because it recognises that, for example, an error of 0.001 may be more significant when the actual value is 0.005, and less significant when the actual value is 0.5.

The other error measure determines correctness of a floating-point number in terms of units in last place (ULP) of f's significand:

$$\text{error}_{\text{ULP}}(f, r) = \left| s - \frac{r}{b^e} \right| b^{p-1}, \tag{1.5}$$

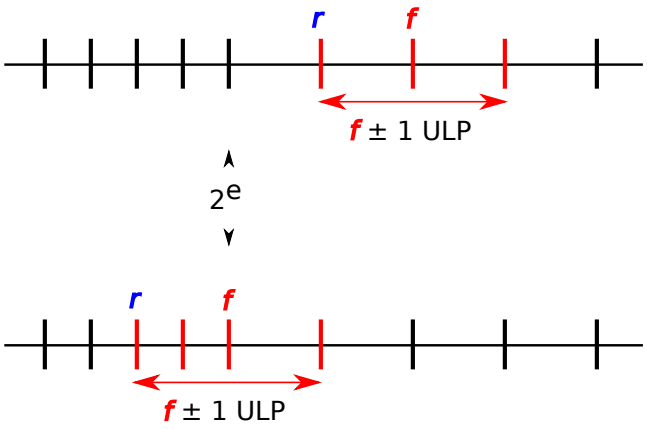

Figure 1.2: Floating-point approximations f of real numbers r, such that $b = 2$. The top shows an example where $\text{error}_{\text{ULP}}(f, r) = 1$ and $|r| \geq 2^e$, in which one better approximation of r exists. The bottom shows an example where $\text{error}_{\text{ULP}}(f, r) = 1$ and $|r| < 2^e$ is a power of two, in which two better approximations exist.

where e and s are respectively the exponent and significand of f, and p is the precision of f, or the number of digits in s. The ULP measurement is helpful because it can tell us, given a floating-point approximation f to a real number r, whether or not a better approximation exists. For example, if $\text{error}_{\text{ULP}}(f, r) = 1$ and $|r| \geq b^e$, then we know that there is one better approximation of r directly adjacent to f, since f is one ULP away from both of its neighbours in this case. This is illustrated in the top of figure 1.2. However, if $\text{error}_{\text{ULP}}(f, r) = 1$ but $|r| < b^e$, there are b better approximations of r, because the spacing between floating-point numbers - thus the distance of one ULP - is multiplied by b at each power of b. Therefore, one must be careful when interpreting $\text{error}_{\text{ULP}}$ if f is at or near a power of b. The bottom of figure 1.2 illustrates this issue.

1.2.4 Rounding Modes

IEEE-754-1985 also introduced the requirement that basic arithmetic operations (addition, subtraction, multiplication, division and square root) be computed to infinite precision, and that the result then be rounded to the nearest representable floating-point number. In the event that the number is equidistant between two valid floating-point numbers, the result would then be rounded to the number with an even least significant bit (LSB), since this introduces less bias, on average, than always rounding upwards or downwards does. The IEEE-754-1985 name for this rounding behaviour is 'round to nearest'. By computing basic arithmetic results to infinite precision, and then rounding using the 'round to nearest' mode, one can guarantee that the error is never greater than

0.5 ULP, which corresponds to a relative error somewhere in the interval:

$$[\frac{b^{-p}}{2}, \frac{b^{1-p}}{2}]. \tag{1.6}$$

The variance in relative error corresponding to 0.5 ULP error is observed due to a phenomenon called 'wobble', where the relative distance from the next highest floating-point number falls slowly from each power of b onwards, before rising sharply at the next power of b. This reflects the change in space between floating-point numbers at different values of e. The upper bound of the relative error for exactly rounded operations is also known as the 'unit roundoff', denoted $\mathbf{u}$, and is often used in rounding error analysis.

$$\mathbf{u} = \frac{b^{1-p}}{2} \tag{1.7}$$

This number is equal to half the distance between one and the next highest floating-point number.

In addition to the 'round to nearest' rounding mode, IEEE-754-1985 defines three other rounding modes. The 'round toward 0' mode, as the name implies, truncates the trailing digit of a non-representable number, rounding it towards zero. The final two modes, named 'round toward $+\infty$' and 'round toward $-\infty$', always round up and down, respectively. These other rounding modes can result in higher relative errors of up to $2\mathbf{u}$, but are useful in certain situations. For example, 'round toward $+\infty$' and 'round toward $-\infty$' are used in the interval arithmetic error bounding technique, discussed later. Note that any form of rounding, including all IEEE-754 rounding modes discussed above, inevitably removes the associative and commutative properties of basic arithmetic, since digits of lower significance are removed or rounded in all intermediate quantities. As a result, the order in which floating-point operations are performed can play a significant role in how error is propagated in an algorithm. With the floating-point number system and the IEEE-754 standard discussed, we can now discuss how the approximate nature of the system, as well as varying levels of IEEE-754 compliance from hardware vendors and software developers, affects the results of realistic neural network simulations.

1.3 Numerical Error

1.3.1 Absorption and Cancellation

When two numbers of very different orders of magnitude are added or subtracted, something peculiar may occur; the operation may simply output the

high-magnitude operand. This is due to a phenomenon called 'absorption', where a low-magnitude operand x is said to be 'absorbed' into a high-magnitude operand y.

For the 'round to nearest' rounding mode, this occurs when x moves y less than halfway towards an adjacent floating-point number. Since the result is closer to y than it is that adjacent number, the result is rounded back to y (remember that the distance between floating-point numbers changes at each power of b, as figure 1.2 illustrates). Furthermore, if the LSB of y is even, absorption will also occur when x moves y exactly halfway towards an adjacent floating-point number, due to the tie-breaking behaviour discussed earlier. As an example of absorption in 'round to nearest' mode, consider a floating-point system with $b = 10$ and $p = 3$. We need to compute $x + y$, where $x = 0.004$ and $y = 1$.

$$\begin{aligned} x &= 0.004 = 4.00 \cdot 10^{-3} \\ y &= 1 = 1.00 \cdot 10^{0} \end{aligned} \tag{1.8}$$

With equation (1.7), we know that half the distance between y and the next highest floating-point number is:

$$\mathbf{u} = \frac{10^{1-p}}{2} = 0.005. \tag{1.9}$$

Since $x < \mathbf{u}$, we know that the result $x + y = 1.004$ is less than halfway towards the next floating-point number 1.01. Thus the result is rounded back down to 1, as illustrated in the top of figure 1.3.

For the directed rounding modes: 'round toward $+\infty$', 'round toward $-\infty$' and 'round toward 0', absorption occurs when x moves y less than the whole distance towards an adjacent floating-point number, and the rounding direction is of opposite sign to x. In this case, the result is always rounded back to y if it is not equal to or greater than that adjacent number, no matter how close it is. For example, using the same floating-point system from above, we will compute $x + y$, rounding towards $-\infty$, where $x = 0.009$ and $y = 1$.

$$\begin{aligned} x &= 0.009 = 9.00 \cdot 10^{-3} \\ y &= 1 = 1.00 \cdot 10^{0} \end{aligned} \tag{1.10}$$

Although x is greater than half the distance towards the next floating-point number 1.01, the addition result 1.009 is still rounded back to y because the rounding direction is towards $-\infty$, as the bottom of figure 1.3 illustrates.

A related source of error is 'cancellation', which is observed when two numbers of similar magnitude are subtracted, resulting in a number at or close to

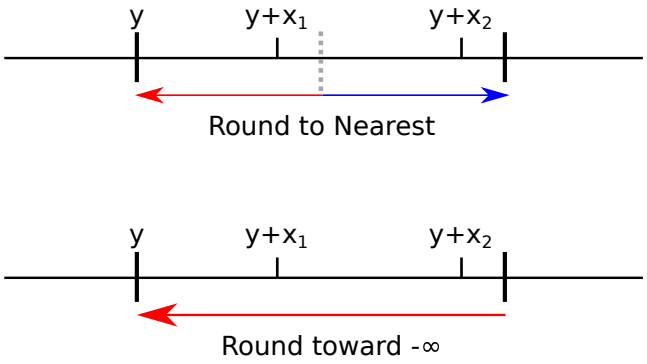

Figure 1.3: The absorption phenomenon. Red and blue arrows respectively indicate numbers that are rounded down and up. For 'round to nearest' (top), $y+x_1$ is rounded back down to y, while $y+x_2$ is rounded up to the next floating-point number. For 'round toward -∞' (bottom), both sums are rounded back down to y.

zero. Goldberg [10] defines two types of cancellation: benign and catastrophic. A benign cancellation is where both operands of the subtraction are known up-front. Since both operands are trusted to be accurate, we can be assured that the result of the subtraction is relatively accurate. On the other hand, a catastrophic cancellation is one where the accuracy of the operands is not known when the subtraction takes place. For a trivial example, take the following matrix:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{1.11}$$

and assume that $ad \approx bc$. Now if we compute the determinant of this matrix, the subtraction $ad - bc$ will be a catastrophic cancellation because of the errors inflicted on the two products ad and bc before the subtraction. Only the most significant figures of each product can be assumed to be correct, and these cancel each other out when subtracted, leaving only the least significant digits – i.e. the digits that were most likely to have been corrupted by the multiplication roundings. Thus, whilst cancellation introduces no more error than any other IEEE-754 subtraction instruction, it makes previous rounding errors in a computation more prominent, relative to the final result. Cancellation is most significant when the absolute sum of all values is much less than the sum of absolute values.

$$\left|\sum_{i=1}^{n} x_{[i]}\right| \ll \sum_{i=1}^{n} \left|x_{[i]}\right| \tag{1.12}$$

Higham [11], [12] explores rounding error in detail, and explains how a summation can be ordered in such a way as to minimise the total rounding error committed. For example, by summing a list of positive numbers in ascending

order, one can alleviate the effect of absorption. This works because the lower magnitude numbers are summed first, allowing them to grow big enough inside a partial sum to not be absorbed by higher numbers later, rather than simply being absorbed early without contributing. The argument is similar for negative number sums, where summing in descending order allows low-magnitude numbers to accumulate without being absorbed early by lower negative numbers.

The benefit of ascending or descending order is lost when both positive and negative numbers are present in the sum. Since both the lowest negative and the highest positive numbers can be high-magnitude, neither ordering prevents absorption. Ordering the summands in ascending absolute value can help reduce rounding error here, since low-magnitude summands, both positive and negative, are always summed before higher magnitude summands. However, this is not always the case. For example, with the following binary floating-point numbers, summing them in ascending absolute value order is actually worse than summing them in descending absolute value order.

$$\begin{aligned} x_1 &= 2^{-p-1} \\ x_2 &= 1 - 2^{-p} \\ x_3 &= -1 \end{aligned} \tag{1.13}$$

With ascending absolute value ordering, $s_1 = x_1 + x_2$ is rounded upwards, while $s_2 = s_1 + x_3$ is exact. However, with descending absolute value ordering, both additions $s_1 = x_3 + x_2$ and $s_2 = s_1 + x_1$ are exact, and no rounding error is incurred. Furthermore, if equation (1.12) holds, ordering arbitrary summands in ascending absolute value may increase the risk of catastrophic cancellation, since each addition is more likely to contain near equal magnitude, but opposite signed, operands.

Another summation method tested by Higham [11] is 'pairwise summation', also known as 'cascade summation', which is a method designed to minimise rounding error. With pairwise summation, all input numbers are paired up together and summed, the results of which are subsequently paired up and summed again, until the base case where only one number remains. The reason this works is that any given summand is only a part of a maximum of $\log_2(n)$ addition operations, as opposed to the recursive summation method, where a given value, including its error, may be a part of up to $n-1$ addition operations. Unlike the summation ordering modifications discussed above, this modification is trivial to implement, and could lower the summation error from a factor of n to a factor of $\log_2 n$. It is worth noting that the pairwise summation scheme is still susceptible the effects of catastrophic cancellation. Adding partial sums of positive values to similar-magnitude partial sums of negative values can still

result in nasty catastrophic cancellations.

1.3.2 The size of δt

Another source of excessive numerical error in simulations emerges with an inappropriate choice of integration step size δt. Numerical integration is about approximating a continuous function by integrating its derivative in discrete steps. Because the integration is discretised, the approximated solution cannot completely follow the curve of the exact solution. The difference between the approximate solution and the exact solution is known as the truncation error. When numerically integrating a model on a computer, the overall error is a combination of rounding error from the computer, and truncation error from the integration method. The combination of these two error sources is called the numerical error.

Intuitively, one might suppose that a simple way to reduce the truncation error of a simulation is to integrate using a smaller step size. For example, one might integrate a model system using a time step $\delta t = 0.1$ ms instead of $\delta t = 0.5$ ms, but this will only get you so far. The more time steps per unit of time, the more floating-point operations, or FLOPS, and the bigger the round-off error can potentially become [13]. Choosing the optimum δt size is a trade-off; too small and the rounding error from floating-point arithmetic dominates the result, yet too big and the truncation error of the method overpowers the result, and it does not make sense to spend hours debugging and fixing bad rounding errors when truncation error is clearly the bigger issue. One must use rounding error evaluation techniques, discussed later, to characterise the error whilst systematically varying δt in order to find the sweet spot, where numerical error is minimised.

On the subject of δt, after integrating a single time step, the naïve user might be tempted to iterate t by doing:

$$t = t + \delta t. \tag{1.14}$$

However, this may cause problems. For example, consider a simulation with $\delta t = 0.1$. The number 0.1 is impossible to represent in a base-2 floating-point system, so repeated accumulation of δt eventually desynchronises t from the actual simulated time. This could cause problems in real-time applications, such as dynamic clamp neuroscience experiments, where precise timing may be essential. Furthermore, a sufficiently long simulation may cause t to overflow if the exponent range of t is too small. If the total simulated time is small, and the floating-point format is large, then these problems will not occur, but they become more likely for longer simulations and smaller floating-point formats.

There is a simple solution to this problem; keep an additional variable i, which holds the integer step number, incremented by one at every iteration, where t can be found by simply calculating:

$$t = i \cdot \delta t. \tag{1.15}$$

Alternatively one could store t as a fixed-point number, which can then be cast to a floating-point type as and when required. However, this approach has problems when adaptive step size integration methods use used, so all approaches should be considered carefully.

1.3.3 Software and Hardware Compliance

As of IEEE-754-1985, the only floating-point operations that are required to be exactly rounded are the basic $(+, -, /, *, \sqrt{})$ operations and floating-point conversion functions. Later, in 2008, the revised IEEE-754 standard [4], henceforth IEEE-754-2008, included the 'fused multiply-add' (FMA) function in this list, which is discussed below. However, many popular mathematical functions, collectively known as the elementary transcendental functions, have yet to be included proper. Instead, IEEE-754-2008 merely recommends that hardware vendors and software developers support correct rounding for these functions. The elementary transcendental functions include the trigonometric, log and exponential functions. Because IEEE-754-2008 only recommends, and does not require, that these functions be implemented with correct rounding, there can be notable differences across architectures, and even across compilers and math libraries on the same architecture. This poses a problem when verifying the results of a simulation, such as one running on a CUDA device with the CUDA runtime library version 6.0, against a reference implementation on another platform, such as x86_64 with the GNU Compiler Collection (GCC) and the GNU Lib C (glibc) standard C library.

Floating-point arithmetic was quite inaccurate during the first few versions of the CUDA architecture. In fact, versions 1.0 through 1.2 of the CUDA architecture barely implemented the IEEE-754 standard at all. Only single-precision floating-point numbers were supported, subnormal numbers were flushed to zero, and very few operations were correctly rounded [5]. Version 1.3 introduced some IEEE-754-2008 compliance for double-precision numbers, but was very slow, and single precision remained inadequate. The results of GeNN [7] simulations for these CUDA architecture versions would be completely incorrect with respect to the results of a simulation on another fully IEEE-754 compliant architecture. Fortunately, recognising the potential of CUDA devices for GPGPU applications, NVIDIA have since fully adopted IEEE-754-2008 compli-

ance for all architectures of version 2.0 and greater. All of the CUDA runtime library's basic arithmetic functions, including fused multiply-add, are correctly rounded in both single-precision and double-precision, and the elementary transcendental functions are all rounded to within around 2 ULP of the correctly rounded result [8]. In addition, CUDA also provides so-called 'device intrinsics', which are functions that correspond to a construct of special high-speed arithmetic circuits inside each CUDA core, collectively called the Special Function Unit (SFU). The intrinsics in the SFU relax the IEEE-754 correct rounding constraints in return for faster operation speed. The CUDA runtime library version 7.0's elementary transcendental functions may very well be more accurate than the corresponding functions in the glibc library, used by GCC on Linux, given that the error bounds for many of the glibc maths functions, such as the exponential function, have not even been entered into the glibc manual [14]. This poses problems for the numerical analyst who wishes to verify CUDA simulation results, such as those from a GeNN simulation, against a reference x86_64 and glibc implementation.

Another verification problem emerges when the other architecture, on which the reference implementation runs, adheres to an older floating-point standard than the target CUDA architecture [6]. For example, many current lab workstation CPUs only adhere to the older IEEE-754-1985 standard, whereas all CUDA devices with architectures of version 2.0, or higher, adhere to the newer IEEE-754-2008 standard. The newer version introduced the requirement that the FMA operation, given below, be included in the group of basic arithmetic operations that are required to be computed as if to infinite precision, and then rounded correctly, according to the selected IEEE-754 rounding mode.

$$\mathrm{FMA}(a, b, c) = ab + c \tag{1.16}$$

On an IEEE-754-2008 FPU, executing the FMA operation is more accurate than executing both the product and sum instructions separately. Since an FMA instruction counts as one operation, it is only rounded once. This means that the relative error is bounded by $\mathbf{u}$, and the absolute error is:

$$|(ab + c)(1 + \delta_1) - \mathrm{FMA}(a, b, c)|, \tag{1.17}$$

where $|\delta_1| \leq \mathbf{u}$. Doing the operations separately, just as an IEEE-754-1985 FPU would need to do, means having two roundoff steps; calculating the product of a and b, rounding, and then adding this intermediate result with c, rounding again. The final result has a potentially larger absolute error of:

$$|(ab(1 + \delta_1) + c)(1 + \delta_2) - \mathrm{FMA}(a, b, c)|, \tag{1.18}$$

where $|\delta_1| \leq \mathbf{u}$ and $|\delta_2| \leq \mathbf{u}$.

FMA instructions have only recently been shipped as standard with Advanced Micro Devices (AMD) and Intel processors, respectively starting with the 'Bulldozer' [15] and 'Haswell' [16] architectures. Even on CPU architectures which support the FMA instruction, there is still no guarantee that a given compiler will make use of them yet. For instance, FMA may not be used by older compiler releases which are not fully compliant to the IEEE-754-2008 standard. As a result, there can be variation in the results of algorithms amenable to the FMA optimisation, such as the dot product, if compiled by different compilers, or for architectures which implement different IEEE-754 revisions. For maximal confusion, the AMD Bulldozer architecture actually implements a different FMA instruction scheme (FMA4) from that of the Intel architectures, from Haswell onwards (FMA3), although the newer AMD 'Piledriver' architecture supports both FMA3 and FMA4 for compatibility reasons. Thankfully, the FMA3 and FMA4 instruction schemes are both faithful to the IEEE-754-2008 standard in the sense that they both round FMA operations to the nearest floating-point number. As a result, an FMA operation should produce identical results on CUDA architecture version 2.0 onwards, AMD Bulldozer architecture onwards, and Intel Haswell architecture onwards, but one should expect results to differ when using architectures and compilers which do not support correctly rounded FMA operations.

A further issue with comparing results against implementations on older architectures is that the precision of the intermediate values in a calculation cannot be guaranteed. In older Intel 32-bit x86 architectures, floating-point operations were performed on x87 coprocessor chips, which were later integrated into x86 CPU chips. The registers inside an x87 floating-point unit are 80 bits wide, corresponding to IEEE-754 'extended-precision', with 15 bits used for the exponent, 64 bits for the significand and the remaining bit used for the sign. The issue is that there are only so many of these extended registers, and so the result of a floating-point operation might be kept in these extended precision registers, rounded once to extended-precision, or it might be flushed to memory to make way for the next x87 instructions, rounded a second time to regular double or single-precision [6]. This double rounding behaviour may or may not happen at the compiler's discretion, depending on how registers are allocated. On the more recent CPUs, starting from the Intel Pentium 4 and the AMD Athlon 64, the x87 arithmetic instructions have been superseded in favour of the Streaming SIMD Extensions 2 (SSE2) floating-point arithmetic instructions, which use standard IEEE-754 single and double-precision registers instead. As before, running the test and reference simulation implementations on sufficiently modern processor architectures should be enough to avoid this problem.

To summarise, there may be many reasons why the trajectory of an SNN simulation may differ from that of a reference implementation on a different processor architecture. Floating-point instructions may be reordered as a result of the non-determinism in the GPU's thread scheduling mechanism, resulting in different results for the same operation on the same operands. The integration step δt, and the way that time is incremented with it, can cause excessive error if not chosen correctly in user-side code. Comparing a dynamical systems simulation to a reference implementation on a somewhat old architecture is likely to give different results than those of the GPU implementation.

There are IEEE-754-2008 compliant mathematical libraries which implement FMA, and even exactly rounded elementary transcendental functions, with floating-point numbers that will always remain in a predictable precision. GNU MPFR [17] is an arbitrary-precision example of such an library. It allows us to verify dynamical systems models containing transcendental functions against an exactly rounded reference CPU simulation, providing the effects of non-deterministic thread scheduling are accounted for. Although this tells us what the simulation trajectory should be in a fully IEEE-754-2008 compliant floating-point implementation with exactly rounded transcendental functions, it does not tell us much about how bad the error could be in the worst-case, and how much it resembles the exact solution, in terms of correct significant digits. It is also unclear exactly how high the precision needs to be raised in order to produce the correct result. Rump's example [18], [19] is a calculation in which the significant digits in the answer are very similar when computed from single-precision up to quadruple-precision, even though all of the computed results has no significant digits in common with the exact result. This would leave an unwitting user content that a solution is correct, given that the significant digits are similar for each of the tested precisions, even though the computed result has no significant digits in common with the exact result. Therefore, we need an easy way to measure the worst-case numerical error bounds of a dynamical systems simulation, in arbitrary precisions. We now explore some options.

1.4 Interval Arithmetic

In order to determine the error bounds of numerical algorithms, such as SNN simulations, one can use a so-called 'range analysis' method, such as Interval Arithmetic (IA). In this method, all real-valued variables are replaced with intervals, represented by a lower and upper bound.

In interval arithmetic (IA), each of the floating-point variables are replaced with an interval variable $\bar{x} = [x_a, x_b]$ containing, at any given moment, the lowest and highest value that the variable can possibly take, assuming a worst-

case rounding error occurs after each floating-point operation. An alternative, but equivalent, representation is the centre-radius form $\bar{x} = [(x_c - x_r), (x_c + x_r)]$. The bounds of each interval variable begin initialised as the same real input value, and gradually widen by plus and minus the absolute error of each successive floating-point operation, such that the exact value of a given variable is guaranteed to be somewhere within its representative interval.

1.4.1 Interval Functions

The basic arithmetic operations $(+, -, *, /)$ for these interval types are defined as follows.

$$\begin{aligned} [a,b] + [c,d] &= [(a+c), (b+d)] \\ [a,b] - [c,d] &= [(a-d), (b-c)] \\ [a,b][c,d] &= [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)] \\ \frac{[a,b]}{[c,d]} &= \left[\min\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right), \max\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right)\right] \end{aligned} \tag{1.19}$$

Note that division by an interval containing zero is undefined. Other interval operators can be defined, as long as they preserve the invariant that the upper and lower interval bounds of a result are always respectively the highest and lowest value that can possibly be returned by the operation. For the floating-point versions of these interval operations, rounding error can be accounted for by ensuring that the upper bound of the result interval is rounded up using the IEEE-754 'round toward $+\infty$' mode, and the lower bound is rounded down with the IEEE-754 'round toward $-\infty$' mode. Other error sources can be accounted for in this way. If this is done consistently for every arithmetic operation, we can keep track of the total rounding error boundaries in a computation. IA can be considered a zeroth-order range analysis method, since the lower and upper bounds of a range are constant values $[x_a, x_b] = [(x_c - x_r), (x_c + x_r)]$.

1.4.2 The Dependency Problem

IA, however, turns out to be a poor choice for long chained computations, such as numerical integration of dynamical systems. For these kinds of computations, the biggest put-off is that the boundaries obtained by IA are often very conservative. In multiple dimensions, IA tends to include regions of state space which are impossible to reach in a simulation run, since the orientation of the bounding box is fixed to that of the axes. The reason for this is the lack of correlation information encoded with this representation. This is known as the dependency problem [20]. If an interval variable appears more than once in an

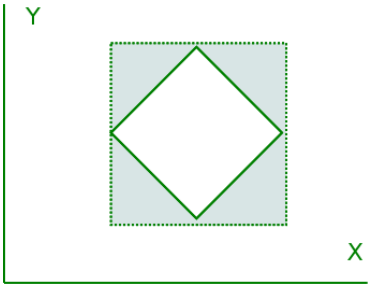

Figure 1.4: A dynamical system of variables X and Y, whose intervals exhibit the wrapping effect, where the computed error box 'wraps' the reachable error box. The dynamics of this system may not allow the trajectory to enter the green area, corresponding to worst-case error bounds of both X and Y according to interval arithmetic.

arithmetic expression, IA treats the second occurrence as if it were a separate independent variable, even though they are in fact the same. Using subtraction as an example, if $\bar{x} = [1, 2]$, we have such things as:

$$\bar{x} - \bar{x} = [(1 - 2), (2 - 1)] = [-1, 1] \tag{1.20}$$

even though the result should clearly be $[0, 0]$. Arithmetic expressions may be rewritten such that a variable only appears once on the right hand side, but this is usually not possible for complicated expressions. Due to the dependency problem, a phenomenon known as the 'wrapping' effect [21], [22] is observed. For instance, in a system of two variables, the error intervals of those variables collectively form a box – or a hypercube, in higher dimensional systems – in which the exact solution lies. In a dynamical system, due to the tight inter-dependency of the system's variables, it may be impossible to reach a point in phase space corresponding to the worst-case rounding error for all variables – shown as the green area in figure 1.4. The true box of reachable states could in fact be rotated in phase space – demonstrated by the white box in figure 1.4 – where the reachable error box's sides are not orthogonal to the axes, in contrast with the computed worst-case error box. For instance, it could be that the point (x, y) lies at the middle of an edge of the green bounding box. However, (x, y) can never lie at two edges due to the linear correlation that exists between x and y. Instead, (x, y) can only ever lie within the white diamond-shaped region. The green bounding box is said to 'wrap' the white diamond-shaped region of actually reachable states.

1.4.3 Discrete Dynamics

Interval arithmetic is made yet more complicated when discrete 'events' are used within a dynamical system, forming so-called hybrid systems. An SNN simulation often uses events to record a spike event to the spike list whenever a threshold criterion is satisfied. In addition to recording the spike, the model neuron's variables are usually reset to some subthreshold baseline level, creating discontinuities in the differential equations. This causes problems when the event threshold criteria is met for only part of an interval variable. If the upper bound of a neuron's membrane potential variable exceeds the threshold potential, but not the lower bound, then should a spike be propagated to the rest of the network? Should the neuron's state variables be reset? One solution would be to hold the upper bounds of the neuron's variables immediately below the threshold values when the threshold criterion is passed, rather than resetting them to the baseline level, and then to set the lower bounds of the neuron variables to the baseline values, if they are not already lower. This ensures that the neuron variables remain single continuous intervals, rather than diverging into two intervals separated by a threshold condition. This still does not deal with the issue of when to propagate the spike to the rest of the network when only the upper bound of a neuron variable is superthreshold. Therefore, IA, though fast and 'correct', is only useful for calculating worst-case error bounds for short numerical algorithms with few correlated variables. To get more accurate bounds in long computations with many correlations, a better interval representation, such as affine arithmetic, can be used instead.

1.5 Affine Arithmetic

Another range analysis method, known by the name Affine Arithmetic (AA) [23], [24], is a variant of IA which represents intervals as first-order polynomials. AA is a variant of IA, in the sense that an AA variable (henceforth an affine range), encodes the minimum and maximum value that a variable may take at any given moment. It avoids the problems of IA by encoding all correlations of one variable with all other variables within its interval representation. Each affine range is a first-order polynomial, such that the constant term $\hat{x}_c$ represents the centre value of the interval, in a similar manner to the centre-radius representation in IA, but with n other terms $\hat{x}_{[i]}\epsilon_{[i]}$ (henceforth deviation terms) instead of one.

$$\begin{aligned} \hat{x} &= \hat{x}_c + \hat{x}_{[1]}\epsilon_{[1]} + ... + \hat{x}_{[n]}\epsilon_{[n]} \\ \epsilon &\in [-1, 1] \end{aligned} \tag{1.21}$$

These n deviation terms each represent a linear correlation with another variable. The values of all noise symbols $\epsilon_{[i]}$ are unknown, and they collectively represent the uncertainty in the interval. If they were known, then the exact error-free solution of a computation can be determined by simply substituting them into the above formula. Affine ranges $\hat{x}$ and $\hat{y}$ are at least partially correlated if $\exists i > 0 : |\hat{x}_{[i]}| > 0 \wedge |\hat{y}_{[i]}| > 0$. In other words, two affine ranges are correlated if they have any non-zero deviation symbols in common. This means that the dependency problem and wrapping effect of standard IA is no longer an issue. For example, returning to the IA subtraction example again (1.20), where $\hat{x} = 1.5 + 0.5\epsilon_{[1]}$ equivalently, we now have:

$$\hat{x} - \hat{x} = (1.5 + 0.5\epsilon_{[1]}) - (1.5 + 0.5\epsilon_{[1]}) = 0 \tag{1.22}$$

as required. Since $\hat{x}$ is the same variable, and entirely correlated with itself, both occurrences of $\hat{x}$ share the deviation term $\hat{x}_{[1]}\epsilon_{[1]}$, which is allowed to cancel in the subtraction. This can happen with any affine forms which share the same deviation terms. Any numerical error from a function is simply appended to the resulting range as a new deviation term $\hat{x}_{[k]}\epsilon_{[k]}$, where k is an unused noise symbol number. The radius of an affine range is the sum of all absolute deviation coefficients in the interval:

$$\hat{x}_r = \sum_{i=1}^{n} |\hat{x}_{[i]}|. \tag{1.23}$$

1.5.1 Affine Functions

Since affine ranges are all first-order polynomials, AA can be considered a first-order range analysis method, where the error bounds are linear in ϵ. Any linear function of one or more affine ranges may be expressed exactly as another affine range; the general form of a linear univariate function in AA is:

$$\begin{aligned} \hat{f}_1(\hat{x}, \alpha, \gamma, \delta) = (\alpha\hat{x}_c + \gamma) + (\alpha\hat{x}_{[1]})\epsilon_{[1]} + ... \\ + (\alpha\hat{x}_{[n]})\epsilon_{[n]} + \delta\epsilon_{[k]}, \end{aligned} \tag{1.24}$$

and the general form of a linear bivariate function is:

$$\begin{aligned} \hat{f}_2(\hat{x}, \hat{y}, \alpha, \beta, \gamma, \delta) = (\alpha\hat{x}_c + \beta\hat{y}_c + \gamma) + (\alpha\hat{x}_{[1]} + \beta\hat{y}_{[1]})\epsilon_{[1]} + ... \\ + (\alpha\hat{x}_{[n]} + \beta\hat{y}_{[n]})\epsilon_{[n]} + \delta\epsilon_{[k]}, \end{aligned} \tag{1.25}$$

where k is an unused noise term number. Multivariate functions can be constructed easily in a similar manner.

Using (1.25), we define the addition, subtraction and negation of affine ranges

as follows.

$$\begin{aligned} \hat{x} + \hat{y} &= \hat{f}_2(\hat{x}, \hat{y}, 1, 1, 0, 0) \\ \hat{x} - \hat{y} &= \hat{f}_2(\hat{x}, \hat{y}, 1, -1, 0, 0) \\ -\hat{x} &= \hat{f}_1(\hat{x}, -1, 0, 0) \end{aligned} \tag{1.26}$$

1.5.2 Multiplication and Division

As a consequence of affine ranges being represented as first-order polynomials, all nonlinear functions must be approximated to the first order to be representable in AA. Addition, subtraction and scalar multiplication are all linear, and thus need no approximation. Multiplication of non-scalar affine ranges, however, is a nonlinear operation, since the multiplication of two first-order polynomials introduces quadratic terms in ϵ.

$$\hat{x}\hat{y} = (\hat{x}_c\hat{y}_c) + \sum_{i=1}^{n}(\hat{x}_c\hat{y}_{[i]} + \hat{y}_c\hat{x}_{[i]})\epsilon_{[i]} + (\sum_{i=1}^{n}\hat{x}_{[i]}\epsilon_{[i]})(\sum_{i=1}^{n}\hat{y}_{[i]}\epsilon_{[i]}) \tag{1.27}$$

Although tighter approximations are known, according to [23], the quadratic final term in (1.27) can be trivially approximated using the product of the radii of $\hat{x}$ and $\hat{y}$:

$$(\hat{x}_r\hat{y}_r)\epsilon_{[k]}, \tag{1.28}$$

where k is an unused noise symbol number and $\hat{x}_r$ is defined as in equation (1.23). This leaves us with the following definition of affine arithmetic multiplication.

$$\hat{x}\hat{y} = (\hat{x}_c\hat{y}_c) + \sum_{i=1}^{n}(\hat{x}_c\hat{y}_{[i]} + \hat{y}_c\hat{x}_{[i]})\epsilon_{[i]} + (\hat{x}_r\hat{y}_r)\epsilon_{[k]} \tag{1.29}$$

Division is defined as the product of $\hat{x}$ and the inverse of $\hat{y}$.

$$\frac{\hat{x}}{\hat{y}} = \hat{x} \cdot \frac{1}{\hat{y}} = \hat{x} \cdot \hat{\text{inv}}(\hat{y}) \tag{1.30}$$

Since the multiplication of two affine ranges needs to be approximated linearly in order to be representable, it is often the case that AA multiplication, and consequently division, produces ranges that are wider than those computed with plain IA. For example, let $\hat{x} = 1+3\epsilon_{[1]}$ and $\hat{y} = 2+5\epsilon_{[2]}$. Converting $\hat{x}$ and $\hat{y}$ into intervals respectively gives $\bar{x} = [-2, 4]$ and $\bar{y} = [-3, 7]$. With IA multiplication, we have $\bar{x}\bar{y} = [-14, 28]$. However, with the trivial AA multiplication defined in equation (1.29), we have $\hat{x}\hat{y} = 2 + 6\epsilon_{[1]} + 5\epsilon_{[2]} + 15\epsilon_{[3]}$. The AA result has a range of $[-24, 28]$, which is noticeably wider than the IA result. This is a known weakness of the AA method, and is most likely to occur when operands

are weakly or not correlated with each other, since AA only has the advantage when operands have shared noise symbols to cancel out. We will return to this topic later.

1.5.3 Transcendental Functions

The affine arithmetic inverse function inv is a univariate nonlinear function, which must be approximated linearly, as does exp, log and all the other elementary transcendental functions. Whilst sqrt is not a transcendental function, it is approximated in an identical manner, and so is mentioned here. There are two methods given in [23] with which one can approximate these functions with; the Chebyshev and Min-Range approximations.

Our goal is to determine the α, γ and δ arguments of function (1.24), corresponding to the best linear approximation to the true function $f(x)$ on the input range $[x_a, x_b] = [(\hat{x}_c - \hat{x}_r), (\hat{x}_c + \hat{x}_r)]$. However, do we mean to say that the best approximation minimises the error, or do we mean to say that it minimises the range of the result? If we are to minimise the error, then the Chebyshev approximation should be used. If we are to minimise the resulting range, then the Min-Range approximation is more appropriate. The following procedures assume that f is twice-differentiable, and that its second derivative does not change sign in $[x_a, x_b]$. Periodic functions, such as sin and cos, require special treatment, and are not discussed here.

Chebyshev Approximation

For the Chebyshev approximation, the idea is to find α, γ and δ, such that the difference between $\hat{f}_1$ and f is minimised in $[x_a, x_b]$. The slope α of the approximation is:

$$\alpha = \frac{f(x_b) - f(x_a)}{x_b - x_a}. \tag{1.31}$$

To find the offset γ and error δ of the approximation, an additional step is necessary; we must first find the point x_p within the input range, where the function's slope is equal to the approximation's slope. To do that, we solve a differential equation for x_p.

$$\left.\frac{df(x)}{dx}\right|_{x_p} = \alpha \tag{1.32}$$

With x_p, we now know the points of maximal difference between αx and $f(x)$. Since f'' does not change sign in $[x_a, x_b]$, these points are x_a, x_b and x_p. We

now compute the differences $f(x) - \alpha x$ for $x \in \{x_a, x_b, x_p\}$.

$$\begin{aligned} d_a &= f(x_a) - \alpha x_a \\ d_b &= f(x_b) - \alpha x_b \\ d_p &= f(x_p) - \alpha x_p \\ d_{\text{lo}} &= \min(d_a, d_b, d_p) \\ d_{\text{hi}} &= \max(d_a, d_b, d_p) \end{aligned} \tag{1.33}$$

Finally, the offset γ and error δ are computed as follows.

$$\begin{aligned} \gamma &= \frac{d_{\text{lo}} + d_{\text{hi}}}{2} \\ \delta &= \frac{d_{\text{hi}} - d_{\text{lo}}}{2} \end{aligned} \tag{1.34}$$

The Chebyshev approximation of the exponential function is illustrated in the top panel of figure 1.5. Note the blue bounding polygon contains $\exp(x)$ completely for all $x \in [x_a, x_b]$.

The Chebyshev approximation is the ideal option, since AA is best when as much correlation information can be preserved, and as little approximation error introduced, as possible. However, Chebyshev approximation suffers from the 'overshoot' and 'undershoot' phenomenon, where the range $[(\hat{y}_c - \hat{y}_r), (\hat{y}_c + \hat{y}_r)]$ of the computed result $\hat{y} = \hat{f}_1(\hat{x}, \alpha, \gamma, \delta)$ is bigger than if it were computed in IA [23]. This is especially problematic when approximating over larger input ranges. You can clearly see undershoot in the top panel of figure 1.5; the approximation evaluated at x_a is negative, however the real exp function should never be negative. To prevent overshoot and undershoot, the Min-Range approximation should be used.

Min-Range Approximation

At the expense of some correlation information, thus a larger independent error term δ, one can find a function $\hat{f}_1$ which approximates f as tightly as plain IA does. Like the Chebyshev approximation, our goal is to find the α, γ and δ arguments of $\hat{f}_1$ in equation (1.24).

To find α, we first find the point $x_q \in [x_a, x_b]$ such that $|f'(x)|$ is minimised. That is to say that we want the point in the input range with the mildest slope, be it positive or negative. Given our earlier assumption that the second derivative does not change sign in the input range, this point must either be x_a

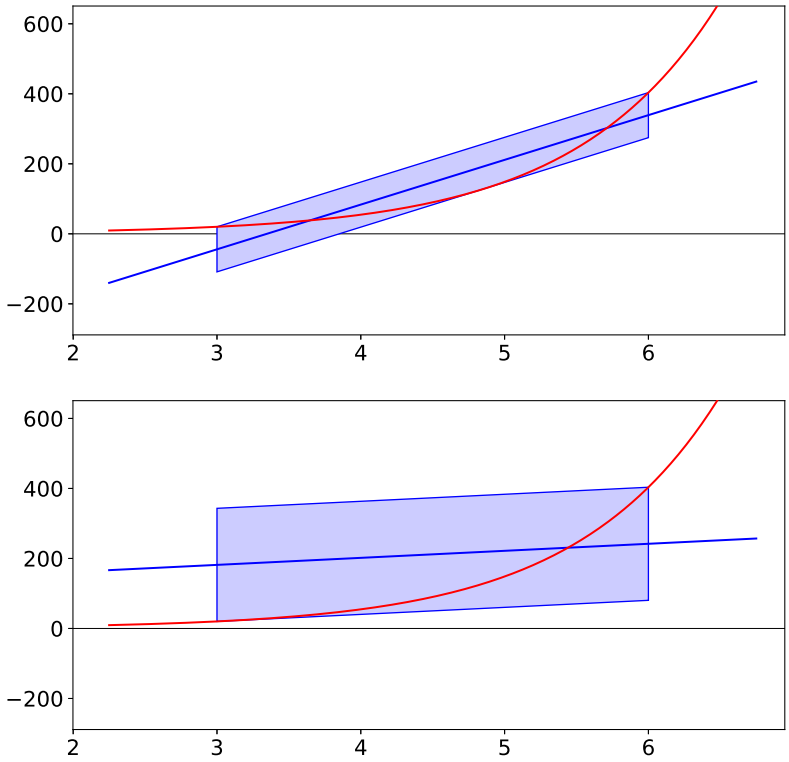

Figure 1.5: Chebyshev (top) and Min-Range (bottom) approximations of $\exp(x)$ over the interval $[x_a, x_b] = [3, 6]$. For both plots, the red line is exp, the blue line is the approximation $\alpha x + \gamma$ and the light blue region represents $\alpha x + \gamma \pm \delta$. Notice how the light blue regions completely cover exp over $[x_a, x_b]$. Also note that, although the bounding region in the Chebyshev approximation is tighter than that of the Min-Range one, its region is partly negative.

or x_b. The slope α of our approximation is the slope at this point:

$$\alpha = \left.\frac{df(x)}{dx}\right|_{x_q}. \tag{1.35}$$

For the exp example, in the bottom panel of figure 1.5, α is the slope at x_a. The points of maximal difference between αx and $f(x)$ are x_a and x_b. We now compute these differences.

$$\begin{aligned} d_a &= f(x_a) - \alpha x_a \\ d_b &= f(x_b) - \alpha x_b \\ d_{\text{lo}} &= \min(d_a, d_b) \\ d_{\text{hi}} &= \max(d_a, d_b) \end{aligned} \tag{1.36}$$

The offset γ and error δ terms are computed in the same manner as in the Chebyshev method, equation (1.34).

As illustrated in figure 1.5, the Min-Range method, in the bottom panel, does not exhibit the same undershoot that the Chebyshev method does, in the top panel. Despite this, the blue approximation region of the Min-Range method is clearly bigger than that of the Chebyshev method. This is because, in this example, much of the correlation information has been lost in the approximation. Consequently, the AA bounds of the Min-Range method begin to look more like the axis-aligned boxes of regular IA. Like undershoot and overshoot in the Chebyshev approximation, this issue is usually only critical for larger input ranges.

With a basic understanding of how floating-point arithmetic works, how numerical error is caused and how it affects numerical simulations, we may now discuss the technical issues which motivate this study. In the following chapter, we see how the non-deterministic thread scheduling of GPU devices is enough to change the behaviour of SNN simulations.

Chapter 2

Motivation

As stated in the introduction, the main motivation behind this study is to determine the worst-case error boundaries of SNN simulations on GPU hardware. This is important because researchers from non-computational backgrounds might expect be able to reproduce their experiment results exactly (bitwise reproducibility), but non-deterministic parallel arithmetic hardware can produce data that is different, even if only slightly, in each simulation run. How reliable are the results if the program is not behaving as expected?

This also has important practical implications, because it implies that one can not use bitwise reproducibility of results as a criterion in software testing. If, after a software update, a result differs slightly from previous runs, it is impossible to tell whether the difference indicates a high likelihood that a bug has been introduced or whether it just originates from the variability of repeated simulation runs. In order to be able to use testing during software development, we need predictions on how much variability to expect between runs and when a deviation becomes significant enough to indicate the presence of a bug.

We now discuss an instance of a spiking neuron model that illustrates this phenomenon - i.e. whose state can vary across different runs on parallel hardware.

2.1 Non-Determinism in Limit Cycles

In order to demonstrate the problem, an instance of the popular Izhikevich neuron model [25] is used. The Izhikevich neuron model is defined as follows, where V is the membrane potential in millivolts and U is the spike recovery

variable.

$$\begin{aligned} \frac{dV}{dt} &= 0.04V^2 + 5V + 140 - U + I \\ \frac{dU}{dt} &= a(bV - U) \\ \text{if } V \geq 30, \text{ then } &\begin{cases} V \leftarrow c \\ U \leftarrow U + d \end{cases} \end{aligned} \tag{2.1}$$

In the above definition, the a parameter determines the time scale of spike recovery, and b determines how sensitive the recovery dynamics are to changes in membrane potential. The remaining parameters affect the discrete spike dynamics, with c being the baseline voltage that V is reset to, and d determining the spike recovery dynamics due to slower rectifying currents. The dynamics of the Izhikevich model are unstable about the spiking threshold. Because of this, there is a possibility that this instability can amplify perturbations in trajectories which pass near the threshold boundary, where small perturbations can push some trajectories superthreshold, whilst other trajectories remain subthreshold.

Such a situation is demonstrated by simulating a single Izhikevich neuron 10 times for 7500 milliseconds, in steps of $h = 0.25$ milliseconds, with forward Euler integration. The parameters of the model are set to the 'typical' values [25] of $a = 0.02$, $b = 0.2$, $c = -65$ and $d = 2.0$. A fixed input current of 3.8 $\mu\text{A/cm}^2$ is applied to push the neuron just above firing threshold, and into a stable limit cycle. An additional 10 constant currents are drawn from a normal distribution with 0 mean and 0.5 standard deviation. These constant currents are summed together in randomised order at the start of each iteration, to simulate parallel input current summation in GeNN simulations, and the result is added to the base input current. The results of the ten runs are plotted in figure 2.1. Although the simulation trajectories start identically, note how the phase and amplitude of spikes has diverged noticeably towards the end of the simulation. This effect is purely a result of non-determinism in the current summation ordering.

Note that the Izhikevich neuron model defined in equation (2.1) will always converge to a stable fixed point in the absence of external input, assuming its parameters are tuned to exhibit quiescent behaviour. When the neuron is not spiking, any prior perturbation to the model's trajectory will immediately begin to disappear. Because the prior experiment set-up is essentially flooding the neuron with input current, and not allowing the trajectories to converge on a stable fixed point, the perturbations quickly become apparent. However, in simulations of real neural circuits, this chronic saturation of neurons is unlikely to occur for long periods. An exception to this rule would be intrinsically spiking

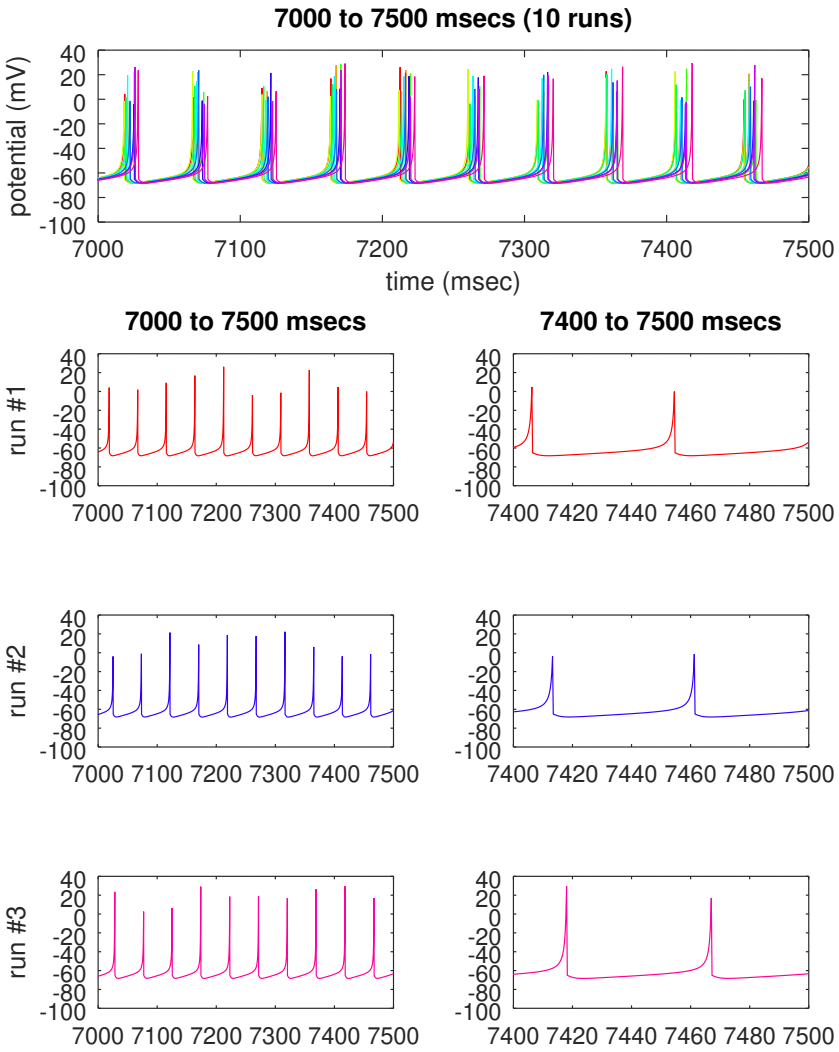

Figure 2.1: This plot shows 10 simulation runs of an identical Izhikevich neuron, receiving 10 fixed input currents that are summed in a randomised order. Note how the phase and amplitude of spiking varies slightly between runs, due to the different input current values obtained by the non-deterministic summation order of the identical set of input currents. Note also that the input current is constantly superthreshold, and the neuron state trajectory is never allowed to converge to a stable fixed point.

neurons, which have been configured to always be superthreshold in the absence of inhibitory input. Although the resulting limit cycle dynamics may be stable, there is nothing driving the limit cycle's phase to convergence.

While this example may appear a little contrived, it is easy to imagine that in simulations with millions of neurons, there is always a neuron that is in this regime. Once spike times have moved noticeably, the effects may or may not propagate further through the network. If they do, this can lead to measurable difference in global results, especially in dynamically rich recurrent neural networks. We now explore trajectory perturbation in a more realistic SNN model.

2.2 Non-Determinism in Realistic Simulations

Earlier in this study, the trajectory perturbation due to non-deterministic summation order was thought to be far greater than it actually turned out to be. However, it was later found that a large part of the observed perturbations was caused by issues in GeNN [7] with the random number generator seed fixing mechanism, and GPU-side memory initialisation, which have since been fixed. In many cases, realistic SNN simulations can be a little more stable than the contrived example given in the previous section, with more stable fixed points, and therefore more opportunities for perturbed simulation trajectories to converge on.

To demonstrate this, a large 'pulse-coupled neural network' (PCNN) of $n = 10000$ neurons is simulated on an NVIDIA Tesla K40c GPU using GeNN. The network is built entirely from Izhikevich neurons, with $0.8n = 8000$ neurons providing excitatory positive current to a random subset of the network, while $0.2n = 2000$ neurons provide an inhibitory negative current to a random subset of the network. Excitatory neurons are subject to a random normally distributed input current at each step, with mean of zero and standard deviation of five. Inhibitory neurons are subject to normally distributed noisy current with a mean of zero and a standard deviation of two. Each neuron randomly projects synapses to at most $0.1n = 1000$ other neurons within the network. The random number generator seed is fixed in all upcoming experiments, to demonstrate purely the effect of non-determinism in parallel simulations alone. The complete network is integrated for 1000 milliseconds using the forward Euler scheme, in steps of $h = 1$ millisecond. Note that it is possible to reduce the total numerical error per unit of simulated time by using higher-order integration methods, which may in turn allow larger time steps to be taken. Euler steps of one millisecond are instead used here, since they are more typical for Izhikevich neuron simulations.

The mean and standard deviation of membrane potential for the first five

excitatory neurons from 50 identical parallel runs of the PCNN simulation are plotted in the left and centre columns of figure 2.2, respectively. From these plots, the divergence between simulation trajectories is no more apparent than that seen in the previous contrived experiment, despite the additional complexity of the model. Although small perturbations appear during high spiking activity, the trajectories immediately converge back during the more stable subthreshold dynamics, since none of the neurons are intrinsic spikers.

In order to confirm that all trajectory perturbation is a consequence of parallel execution alone, and not a hidden software or hardware issue, an identical serialised model is also simulated on the GPU, and the standard deviations of the first five excitatory neurons are plotted in the right hand column of figure 2.2. As discussed previously, GeNN simulations transfer spikes between neuron populations using spike list structures, which are filled in the order that the neuron threads finish in. It follows that if one were to fix the order that spike lists are populated, assuming all random number seeds are fixed, then all perturbations should disappear. Indeed, this is what we see. This means that the problem of bounding numerical error in parallel GPU simulations of SNN models is reduced to bounding the error of the equivalent serial simulation, and adding the worst-case error of the non-deterministic current summations.

The experiments in this chapter have shown that small numerical perturbations are enough to distort both the phase and amplitude of neuronal spiking in SNN simulations. They further show that, while this distortion is more apparent in small specially constructed toy problems, there is still potential for it to occur in larger and more realistic SNN models.

Although rather specific prerequisite conditions must be met before significant trajectory divergence becomes likely, the possibility is ever present in parallel simulations. In theory, any model with sufficient activity, simulated for long enough, can diverge in a significant manner by chance. This could be an issue in networks which require precise spike timing to function correctly. Given the initial conditions of the simulation, how can one predict when such a trajectory divergence will occur, if at all, and what would happen to the network's behaviour if it did? Even if it does not, there are still the smaller perturbations which cause minor mismatches between resulting datasets. We need a way to compute the upper and lower bounds of these simulation trajectories to ensure that researchers know how much divergence should be expected, allowing them to make appropriate adjustments to their simulation and analysis methods. In the following chapter, I introduce my Arpra library for arbitrary-precision range analysis that addresses these issues.

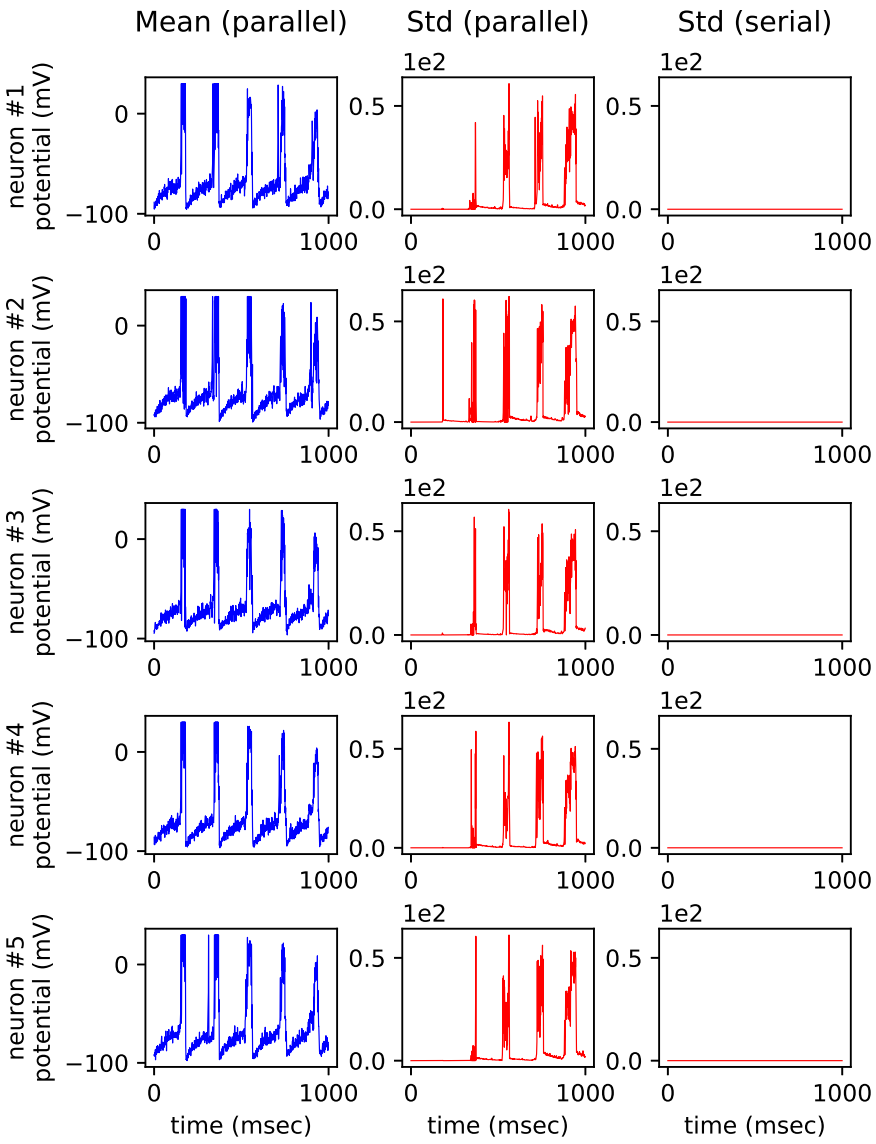

Figure 2.2: This plot shows the mean and standard deviation of membrane potential trajectory for the first five excitatory neurons of an Izhikevich PCNN simulation, computed with parallel and serial current summation order. Respectively, the left and the centre column shows the mean and standard deviation of trajectories computed in the parallel simulation. The right column shows the standard deviation of trajectories computed in the equivalent serial simulation. Note how the system is stable enough such that, although the trajectories diverge from time to time, the perturbations are not enough to cause a significant divergence, and trajectories eventually converge back. Also note how the serialisation of input current summation is enough to completely prevent the trajectory divergence.

Chapter 3

The Arpra Library

In this chapter I will introduce a new software tool kit I have developed, named the Arpra library [26], which stands for arbitrary-precision range analysis. Arpra is a fully open source library, written in C, and is compatible with all UNIX-like systems, including Linux, BSD and Mac. It is licensed under the terms of the GNU Lesser General Public version 3 license (LGPL-3.0), which not only allows Arpra to be used freely for personal and academic reasons, but also allows it to be linked into commercial applications as a shared or static library. It uses the GNU autotools build system, enabling fast and simple configuration and installation. Arpra implements a modified version of mixed IA/AA range analysis, described in [23] and [27], and implemented in INTLAB [28]. Arpra uses GNU MPFR [17] as its floating-point back end, which has many advantages over standard hardware floating-point implementations, as we shall see. Therefore, a basic understanding of MPFR will be required to explain the Arpra library's features.

This chapter will first discuss the MPFR back end, including its internal operation and its advantages. Following this, the Arpra library itself will be discussed, including its internal operation, its improvements over the standard AA method, and its error bounding performance over standard IA.

3.1 The MPFR Library

The GNU MPFR library [17] is a base-2 software implementation of the IEEE-754-2008 standard for floating-point arithmetic [4], with correct rounding for all implemented functions. It allows significands of arbitrary precision, and allows unbounded choice of exponent range, up to the maximum of the signed integer type which MPFR was configured to use. Like the Arpra library, it is free, open source and licensed under the terms of the GNU LGPL-3.0 license. In

addition, it has a comprehensive test suite, ensuring reliability and full IEEE-754 conformance.

Being a software implementation of floating-point arithmetic, it has the advantage of operating identically on any system, no matter which CPU architecture, operating system and compiler is present. Consequently, all computation results are fully reproducible. However, it is important to note that MPFR is an idealised floating-point arithmetic implementation. As discussed in the introduction, hardware and software vendors can be rather inconsistent with their conformance to the IEEE-754 standard, so results obtained with MPFR are not always reflective of floating-point implementations in general. In this section, the internals of MPFR are discussed, as are the advantages of using MPFR over a regular hardware floating-point arithmetic implementation.

3.1.1 Number Representation

As mentioned, MPFR is a software implementation of floating-point arithmetic, outlined in [17]. This means that floating-point numbers are represented and operated on without utilising the processor's FPU. In fact, the FPU is not needed by MPFR, since floating-point numbers are represented purely by integral data types. The representation is similar to that of standard hardware floating-point types, except that each component is stored as a separate field in a C structure named **__mpfr_struct**, aliased as **mpfr_t**.

```
typedef struct {
    int        _mpfr_prec;
    int        _mpfr_sign;
    int        _mpfr_exp;
    unsigned *_mpfr_d;
} __mpfr_struct;

typedef __mpfr_struct mpfr_t[1]
```

Aliasing **__mpfr_struct** as **mpfr_t**`[1]` is a C hack which allows one to define a pointer type **mpfr_t** to a **__mpfr_struct**, such that the pointed memory is pre-allocated on the stack. The precision of each **mpfr_t** variable is stored explicitly in the `_mpfr_prec` field. The sign and exponent are respectively stored in `_mpfr_sign` and `_mpfr_exp` fields, while the significand is stored inside `_mpfr_d`. Positive one in `_mpfr_sign` indicates a positive number, while negative one indicates a negative number.

One of the many strengths of MPFR is its ability to store the significand of each variable with its own unique precision. The significand is stored in

the `_mpfr_d` field as an array of 'limbs'. These limbs are implemented by the GMP arbitrary precision arithmetic library (see [29] and references therein), and are the basis for MPFR's arbitrary precision functionality. The length of a GMP limb array depends on the precision of the MPFR number, found in the `_mpfr_prec` field. For instance, assuming 32-bit unsigned integers, an **`mpfr_t`** variable with 53 bits of precision would need an array of at least two limbs to store the significand in. The remaining 11 bits are disregarded. MPFR significands store the leading bit explicitly, which is always one. Furthermore, MPFR significands are interpreted as being in the interval $[0.5, 1)$, unlike most base-2 hardware floating-point significands, which are interpreted as being in the interval $[1, 2)$. This means that the exponent field of an MPFR number is always one greater than that of the equivalent floating-point number in common hardware implementations. For example, assuming a limb size of four bits, the significand 1.001011010 will be represented in an **`mpfr_t`** with ten bits of precision as follows:

$$\texttt{\_mpfr\_d} = ((1, 0, 0, 1), (0, 1, 1, 0), (1, 0, \square, \square))_2, \tag{3.1}$$

where $\square$ is an unused significand bit.

The ability to individually assign arbitrary precisions to MPFR variables at will is useful in several ways. For instance, it allows us to test a simulation run in multiple precisions, and potentially to adapt to accuracy constraints when numerically integrating a dynamical system through stable or unstable regions of state space. It even allows us to reduce 'overhead' rounding error in range analysis methods such as AA, for example in the computation of the approximation parameters α, γ, and δ for AA transcendental functions. Another strength of the MPFR library is its vast selection of IEEE-754-compliant arithmetic functions, which are discussed next.

3.1.2 Functions

The MPFR library [17] implements all standard arithmetic operations, including FMA, and all of the elementary transcendental functions as suggested by the IEEE-754-2008 floating-point standard [4]. All of the functions implemented in MPFR are rounded correctly, including the transcendental functions. That is to say that all functions behave as if to compute the result in infinite precision, and then round it to the correct floating-point number according to the selected IEEE-754 rounding mode, at some arbitrarily user-defined precision. This is another advantage of using MPFR; most current hardware implementations do not follow the suggestion of correctly rounding their transcendental functions, only their standard arithmetic operators.

Almost all MPFR arithmetic functions follow a common schema, analogous to their equivalent mathematical expressions $z = f(x)$ and $z = g(x, y)$. The following dummy functions illustrate this schema.

```
// Compute z = f(x) (rounding mode rnd)
int mpfr_1 (mpfr_t z, mpfr_t x, int rnd);

// Compute z = g(x, y) (rounding mode rnd)
int mpfr_2 (mpfr_t z, mpfr_t x, mpfr_t y, int rnd);
```

The first argument `z` is a pointer to the MPFR variable in which the result is placed. The following arguments `x` and `y` are pointers to the operands. The precision of the rounding is taken to be the precision of `z` when the function is called, which may differ from that of `x` and `y`. This is useful for computing intermediate variables in the computational overhead of range analysis methods.

The final argument `rnd` is the desired IEEE-754 rounding mode to which the result is rounded. Although standard hardware IEEE-754 implementations also must enable users to switch rounding modes dynamically, in practice this is poorly implemented; the rounding mode FPU register affects floating-point arithmetic globally, and changing it can be computationally expensive. Being able to dynamically set the rounding mode of each MPFR function elegantly avoids this issue.

Finally, each MPFR arithmetic function returns an integer 'ternary value', which indicates whether the result required rounding, and, if so, in which direction. This is also useful for range analysis methods, since it allows us to determine whether or not extra independent error should be added to computed ranges. If the result is exact, then extra error need not be applied.

Fast algorithms for the exact computation of the standard arithmetic operations $(+, -, *, /, \sqrt{})$ are already well known [30], so implementing them in the MPFR library was relatively straightforward. However, the implementation of the transcendental functions was less so, due to a problem known as the 'Table Maker's Dilemma' [31]. We discuss the implications of this problem next.

3.1.3 The Table Maker's Dilemma

In general, computing the correctly rounded result of elementary transcendental functions is non-trivial, due to the Table Maker's Dilemma [31], henceforth TMD. Unlike the standard arithmetic operators, it is difficult to predetermine the amount of precision required to represent intermediate values, in order to produce correctly rounded results. Due to Lefèvre et al. [31] [32], there are efforts to answer this question for the IEEE-754 single and double-precision

floating-point formats, by determining the worst-case values where maximal internal precision is required, helping to design algorithms which correctly handle corner cases. However, these results do not generalise to arbitrary-precision floating-point arithmetic. Firstly, why is the TMD so problematic?

In order to demonstrate an instance of the TMD, consider a base-2 floating-point number with p bits of precision. We need to compute some transcendental function of this number, in some extended precision q, such as to ensure that the final result is correctly rounded according to IEEE-754 'round to nearest'. Assume that the significand of the computed result is one of the following:

$$\begin{aligned} s_\infty &= [\{1.xxxx...xxx0\}10000...0000]xxxx \\ t_\infty &= [\{1.xxxx...xxx0\}01111...1111]xxxx, \end{aligned} \tag{3.2}$$

where digits inside the square brackets [ ] form the extended q-bit significand, digits inside the curly braces { } form the final p-bit significand, and x indicates arbitrarily-valued bits. How does one know whether or not q is enough bits to compute a correct result? In the case of s_∞, the extended q-bit significand could either be rounded up or down, due to the lower significance bits outside the square brackets being respectively high or low. If it is rounded upwards, then s_∞ becomes:

$$s_q = [\{1.xxxx...xxx0\}10000...0001]. \tag{3.3}$$

Then, after the final rounding, s_q becomes:

$$s_p = \{1.xxxx...xxx1\}. \tag{3.4}$$

Conversely, if the extended significand is rounded downwards, then s_∞ becomes:

$$s_q = [\{1.xxxx...xxx0\}10000...0000]. \tag{3.5}$$

After the final rounding, since ties are broken to even zero, s_q becomes:

$$s_p = \{1.xxxx...xxx0\}. \tag{3.6}$$

A similar argument applies to t_∞ in equation (3.2). So which is it then? How do we know how big q would need to be before the tie can be broken?

It turns out, however, that in order to get a correctly rounded result that is fast in the majority of cases, we do not need to know. One can simply increase q until the TMD no longer occurs. This is the gist of a technique discussed by Ziv, known as 'Ziv's strategy' [33], originally proposed in [34]. According to

section 2.5 of [17], Ziv's strategy is used whenever a direct implementation of a function is not possible. The idea is to try the computation with q slightly larger than p, and to evaluate whether or not the TMD occurs in the result. If it does not, we accept the rounded result as correct. If not, we increase q by some factor and try again. Since the TMD does not occur in the majority of cases, the first few loops of Ziv's strategy are usually sufficient, and the average computation time of the function is reasonably low. The occasional nasty corner case is acceptable. The MPFR summation routine also makes use of Ziv's strategy. Having a correctly rounded summation routine is helpful for determining the rounding error bound of recursive and pairwise summation methods, as explained in the following section.

Now that a basic understanding of arbitrary-precision floating-point and the MPFR library has been established, we can begin to discuss the internal workings of the Arpra library itself.

3.2 Features of the Arpra Library

The Arpra library loosely follows the design philosophy of the MPFR library [17], and implements mixed IA/AA [23] [27], with some extra modifications. In this section, we discuss the implementation and various features of the Arpra library, starting with how Arpra represents ranges.

3.2.1 Range Representation

Much like how the MPFR library represents floating-point numbers with C structures, the Arpra library represents number ranges with C structures. The elementary structure of an Arpra computation is known as an **arpra_range**.

```
typedef struct {
    __mpfi_struct  true_range;
    __mpfr_struct  centre;
    __mpfr_struct  radius;
    __mpfr_struct *deviations;
    unsigned      *symbols;
    unsigned       nTerms;
    unsigned       prec;
} arpra_range_struct;

typedef arpra_range_struct arpra_range;
```

The `true_range` field is an MPFI interval representing the actual lower and

upper bounds of the **arpra_range**. MPFI is an implementation of IA, written by Revol and Rouillier [35], which also uses MPFR as its floating-point back end. The `centre` and `radius` fields respectively hold the $\hat{x}_c$ and $\hat{x}_r$ of the affine range $\hat{x}$. Next, the `deviations` and `symbols` fields are respectively pointers to an array of deviation coefficients and a corresponding array of noise symbol numbers, which will be discussed next. The following field `nTerms` is the number of non-zero deviation coefficients in $\hat{x}$. Finally, `prec` is the precision of $\hat{x}$.

The `radius` field is a redundant variable which accumulates the absolute value of all deviation terms in the **arpra_range**. Although the radius must be known when computing the `true_range` field, this is the only time it is used by Arpra internally. This field could in principle be removed, and the computed radii discarded after use, saving the space of one MPFR number per **arpra_range** instance in memory. As the time of writing, it is present in the **arpra_range** structure for convenience.

In an AA implementation that accounts for rounding errors, a new deviation term is typically added after each operation, meaning the number of active noise symbols grows very quickly. Furthermore, noise symbols often only affect a small subset of affine ranges in a computation. In an effort to reduce the memory footprint of AA, Stolfi and de Figueiredo [23] suggested that rather than allocating enough memory to store deviation coefficients for all active noise symbols in a computation, a sparse representation should be used. The idea is to store only the non-zero deviation coefficients inside the `deviations` array. For each deviation coefficient, the corresponding noise symbol number is stored at the same index of the `symbols` array, as in the following example.

$$\begin{aligned} \texttt{deviations} &= (2.45, 1.03, 12.56, 3.12) \\ \texttt{symbols} &= (1, 3, 4, 6) \end{aligned} \tag{3.7}$$

The deviation terms stored in these arrays are sorted in order of increasing noise symbol number, to reduce the complexity of indexing into them. In the above example, note how at least six noise symbol numbers must exist globally: $(1, 2, 3, 4, 5, 6)$. However only the four symbols $(1, 3, 4, 6)$ are actually stored in the deviation term arrays, since the deviation coefficients of symbol numbers $(2, 5)$ are zero. Depending on the number of active noise symbols at a given point in the computation, this could be far less computationally intensive than the equivalent dense representation.

Each **arpra_range** must be initialised before use. This allocates the internal memory of the range, and initialises it to the Arpra equivalent of IEEE-754 not-a-number. When done with a range, the memory should be freed to prevent

memory leaks. The following functions achieve this.

```
// Initialise z with default precision
void arpra_init(arpra_range *z);

// Initialise z with a given precision
void arpra_init2(arpra_range *z, unsigned prec);

// Free the memory used by z
void arpra_clear(arpra_range *z);
```

With the range representation discussed, we now proceed to discuss the general structure of the mathematical functions implemented in Arpra.

3.2.2 Function Structure

As discussed earlier, univariate and bivariate AA functions are often extensions of the generic functions $\hat{f}_1$ in (1.24) and $\hat{f}_2$ in (1.25), and thus have a similar algorithm structure. A small exception is in multiplication, where the centre value calculation is a product, rather than a linear sum. The Arpra library implements two auxiliary functions: `arpra_affine_1`, corresponding to $\hat{f}_1$, and `arpra_affine_2`, corresponding to $\hat{f}_2$.

```
// Compute alpha x + gamma +/- delta
void arpra_affine_1 (arpra_range *z,
                     arpra_range *x, mpfr_t alpha,
                     mpfr_t gamma, mpfr_t delta);

// Compute alpha x + beta y + gamma +/- delta
void arpra_affine_2 (arpra_range *z,
                     arpra_range *x, arpra_range *y,
                     mpfr_t alpha, mpfr_t beta,
                     mpfr_t gamma, mpfr_t delta);
```

As seen above, Arpra mathematical functions use a function schema similar to that used by MPFR, with the result pointer followed by the operand pointers. Algorithm 1 illustrates the structure of `arpra_affine_1`, whilst algorithm 2 illustrates the structure of `arpra_affine_2`. In these listings, the `centre` and `radius` fields of an **`arpra_range`** structure are abbreviated c and r, the `symbols` and `deviations` fields are abbreviated s and d, while the `nTerms` and `true_range` fields are abbreviated n and t, respectively.

As specified in the Affine Arithmetic section, using these generic affine functions, Arpra implements the plus, minus and negation operations, as well as the

Algorithm 1 Auxiliary univariate function

procedure AFFINE_1($\hat{z}, \hat{x}, \alpha, \gamma, \delta$)
  if $\hat{x} = \text{NaN}$ **then** ▷ Check for NaN
    $\hat{z} \leftarrow \text{NaN}$
    return
  else if $\hat{x} = \infty$ **then** ▷ Check for ∞
    $\hat{z} \leftarrow \infty$
    return
  end if
  $\hat{z}.c \leftarrow \alpha \cdot \hat{x}.c + \gamma$ ▷ Compute $\hat{z}_c$
  if $\hat{z}.c$ is inexact **then**
    $\delta \leftarrow \delta + 0.5 \text{ ULP}(\hat{z}.c)$
  end if
  $\hat{z}.r \leftarrow 0$
  $\hat{z}.n \leftarrow 0$
  for $xi \leftarrow 1$ to $\hat{x}.n$ **do** ▷ Compute each $\hat{z}_{[i]}$
    $\hat{z}.d[\hat{z}.n + 1] \leftarrow \alpha \cdot \hat{x}.d[xi]$
    $\hat{z}.s[\hat{z}.n + 1] \leftarrow \hat{x}.s[xi]$
    if $\hat{z}.d[\hat{z}.n + 1]$ is inexact **then**
      $\delta \leftarrow \delta + 0.5 \text{ ULP}(\hat{z}.d[\hat{z}.n + 1])$
    end if
    $\hat{z}.r \leftarrow \hat{z}.r + |\hat{z}.d[\hat{z}.n + 1]|$
    $\hat{z}.n \leftarrow \hat{z}.n + 1$
  end for
  $\hat{z}.d[\hat{z}.n + 1] \leftarrow \delta$ ▷ Append new $\hat{z}_{[i]}$
  $\hat{z}.s[\hat{z}.n + 1] \leftarrow$ a new noise symbol
  $\hat{z}.r \leftarrow \hat{z}.r + \delta$
  $\hat{z}.n \leftarrow \hat{z}.n + 1$
end procedure

Algorithm 2 Auxiliary bivariate function

procedure AFFINE_2($\hat{z}, \hat{x}, \hat{y}, \alpha, \beta, \gamma, \delta$)
  if $(\hat{x} = \text{NaN}) \vee (\hat{y} = \text{NaN})$ **then** ▷ Check for NaN
    $\hat{z} \leftarrow \text{NaN}$
    return
  else if $(\hat{x} = \infty) \vee (\hat{y} = \infty)$ **then** ▷ Check for ∞
    $\hat{z} \leftarrow \infty$
    return
  end if
  $\hat{z}.c \leftarrow \alpha \cdot \hat{x}.c + \beta \cdot \hat{y}.c + \gamma$ ▷ Compute $\hat{z}_c$
  if $\hat{z}.c$ is inexact **then**
    $\delta \leftarrow \delta + 0.5 \text{ ULP}(\hat{z}.c)$
  end if
  $\hat{z}.r \leftarrow 0$
  $\hat{z}.n \leftarrow 0$
  $xi \leftarrow 1$
  $yi \leftarrow 1$
  while $(xi \leq \hat{x}.n) \vee (yi \leq \hat{y}.n)$ **do** ▷ Compute each $\hat{z}_{[i]}$
    if only $\hat{x}$ has next symbol **then**
      $\hat{z}.d[\hat{z}.n + 1] \leftarrow \alpha \cdot \hat{x}.d[xi]$
      $\hat{z}.s[\hat{z}.n + 1] \leftarrow \hat{x}.s[xi]$
    else if only $\hat{y}$ has next symbol **then**
      $\hat{z}.d[\hat{z}.n + 1] \leftarrow \beta \cdot \hat{y}.d[yi]$
      $\hat{z}.s[\hat{z}.n + 1] \leftarrow \hat{y}.s[yi]$
    else
      $\hat{z}.d[\hat{z}.n + 1] \leftarrow \alpha \cdot \hat{x}.d[xi] + \beta \cdot \hat{y}.d[yi]$
      $\hat{z}.s[\hat{z}.n + 1] \leftarrow \hat{x}.s[xi] = \hat{y}.s[yi]$
    end if
    if $\hat{z}.d[\hat{z}.n + 1]$ is inexact **then**
      $\delta \leftarrow \delta + 0.5 \text{ ULP}(\hat{z}.d[\hat{z}.n + 1])$
    end if
    $\hat{z}.r \leftarrow \hat{z}.r + |\hat{z}.d[\hat{z}.n + 1]|$
    $\hat{z}.n \leftarrow \hat{z}.n + 1$
  end while
  $\hat{z}.d[\hat{z}.n + 1] \leftarrow \delta$ ▷ Append new $\hat{z}_{[i]}$
  $\hat{z}.s[\hat{z}.n + 1] \leftarrow$ a new noise symbol
  $\hat{z}.r \leftarrow \hat{z}.r + \delta$
  $\hat{z}.n \leftarrow \hat{z}.n + 1$
end procedure

Chebyshev versions of the square root, natural exponential, natural logarithm and inverse functions. These algorithms are all implemented as in [24]. Arpra also implements Min-Range versions of the natural exponential and inverse functions, with Min-Range square root and natural logarithm left for future work. Multiplication, thus division, is handled specially by an algorithm nearly identical to algorithm 2, in which the centre value and deviation terms are computed as in equation (3.14). With the univariate and bivariate affine auxiliary functions defined, the general structure of an Arpra univariate and bivariate function is respectively listed in algorithm 3 and algorithm 4. The `compute_range` and `trim_range` functions will be discussed later in this section.

Algorithm 3 Univariate function structure

1: **procedure** UNIVARIATE_FUNCTION($\hat{z}, \hat{x}$)
2: $\alpha, \gamma, \delta \leftarrow$ choose affine parameters for function
3: affine_1($\hat{z}, \hat{x}, \alpha, \gamma, \delta$)
4: compute_range($\hat{z}$)
5: trim_range($\hat{z}$)
6: **end procedure**

Algorithm 4 Bivariate function structure

1: **procedure** BIVARIATE_FUNCTION($\hat{z}, \hat{x}, \hat{y}$)
2: $\alpha, \beta, \gamma, \delta \leftarrow$ choose affine parameters for function
3: affine_2($\hat{z}, \hat{x}, \hat{y}, \alpha, \beta, \gamma, \delta$)
4: compute_range($\hat{z}$)
5: trim_range($\hat{z}$)
6: **end procedure**

All Arpra functions check to see if the operand ranges $\hat{x}$ and $\hat{y}$ are real and finite. That is to say that the operands are not the equivalent of IEEE-754 not-a-number (henceforth NaN), and that their range is not infinite. In Arpra, a range is NaN if either or both of the `true_range` bounds are NaN, whereas a range is infinity if either both of the `true_range` bounds are infinity, and neither of them are NaN. If either of the operands are NaN or infinity, then the function immediately sets the result $\hat{z}$ to respectively NaN or infinity, and then returns, allowing many unnecessary instructions to be skipped in such cases. Next, the centre value $\hat{z}_c$ and the deviation terms $\hat{z}_{[i]}\epsilon_{[i]}$ of the result $\hat{z}$ are computed, along with the new numerical error term. The absolute value of the deviation coefficients are accumulated in the radius, rounding upwards. Finally, the `true_range` field is computed and trimmed after the auxiliary affine function calls in algorithms 3 and 4. Arpra implements a few tricks to reduce the overhead error incurred due to the AA method. These tricks will be discussed in the following sections.

3.2.3 Rounding Errors

When computing $\hat{z}_c$ and all $\hat{z}_{[i]}$, any error which occurs in these calculations must be accumulated and appended to $\hat{z}$ as a new deviation term. Therefore, we need these to all be as accurate as possible, such as to keep the overhead rounding error from AA minimal. The error itself is computed differently from how Stolfi and de Figueiredo compute it in [23]. Their method involves computing the terms three times; once with 'round to nearest' mode, again with 'round toward $+\infty$', and once more with round negative. The result computed in 'round to nearest' mode is used as the term, whilst the maximum of the differences between this and the two directed rounding results is taken to be the rounding error. Consequently, the rounding error it incurs may be larger than 0.5 ULP. Using the MPFR library, we need only compute terms once, and we can reduce the incurred rounding error to a maximum of 0.5 ULP.

The Arpra library does this using extended precision intermediate variables, unless it is possible to compute terms with a single correctly rounded MPFR call. For example, if computing

$$\hat{z}_c = \alpha\hat{x}_c + \gamma \tag{3.8}$$

from $\hat{f}_1$, Arpra computes the whole expression with a single call of MPFR's correctly rounded FMA function `mpfr_fma`. However, if computing

$$\hat{z}_c = \alpha\hat{x}_c + \beta\hat{y}_c + \gamma \tag{3.9}$$

from $\hat{f}_2$, high-precision intermediate variables for the products $\alpha\hat{x}_c$ and $\beta\hat{y}_c$ must be used, such that the precision is high enough that the results do not need any rounding. The two products are then summed together with the remaining γ term. In order to compute an exact unrounded floating-point product, the precision of the result must be at least the sum of the operands' precisions. For example, if the precision of x is 25, and the precision of y is 30, then the result $z = xy$ can be represented without rounding if its precision is at least 55. With these exact unrounded products, one then calls the exactly rounded `mpfr_sum` function, which guarantees that the whole expression is evaluated with just one rounding at the end, and thus has a maximum error of 0.5 ULP. We suggested adding a dot product function to the MPFR library at the iRRAM/MPFR/MPC Workshop (2018), in order to simplify deviation term and centre value calculations in Arpra functions of two or more operands. With our thanks to the developers, the `mpfr_dot` function became available in recent versions of MPFR, which combines the above computation into a single function call.

Arpra uses an internal auxiliary function to compute 0.5 ULP of an MPFR number. Since significands in MPFR are normalised to within $[0.5, 1)$, the exponent in an MPFR ULP calculation must be one less than the exponent in a standard IEEE-754 implementation, whose significands are normalised to within $[1, 2)$. As such, 0.5 ULP of an MPFR number $\hat{x}$ is computed as follows, where $\hat{x}_{\text{prec}}$ is the precision and $\hat{x}_{\text{exp}}$ is the exponent.

$$\frac{\text{ULP}(\hat{x})}{2} = 2^{\hat{x}_{\text{exp}} - \hat{x}_{\text{prec}} - 1} \tag{3.10}$$

This function is applied to the result of the term computation, and the resulting 0.5 ULP error is accumulated into a new deviation term, rounding upwards, but only if rounding occurs. As discussed in the MPFR section, all MPFR functions return a ternary value indicating whether or not the result is exact, and, if not, the direction of rounding. We can use this value to conditionally include rounding error, for each term, only if the ternary value indicates that rounding has occurred, and exclude it if not, saving yet more overhead rounding error in Arpra. This corresponds to lines 10-12 and 18-20 of algorithm 1, and also lines 10-12 and 28-30 of algorithm 2.

3.2.4 Arbitrary-Precision

As mentioned in the MPFR section, one of the strengths of MPFR is through the user's ability to dynamically set the precision of **`mpfr_t`** variables. Similar to MPFR, Arpra allows users to dynamically change the precision of the `true_range` field of **`arpra_range`** variables, which is useful for determining the effect of altering floating-point precision in a computation. In the following discussion, the 'working precision', or simply the 'precision', of an **`arpra_range`** shall refer to the precision of that range's `true_range` field.

The working precision of an **`arpra_range`** is set during initialisation. If a range is initialised using the `arpra_init2` function, then its working precision is set to the value of the precision argument. If the range is initialised with the `arpra_init` function, then its working precision is determined by a global 'default precision' variable. The value of this default precision can be retrieved and dynamically set by the user with the following functions.

```
// Retrieve the default precision
unsigned arpra_get_default_precision();

// Set the default precision
void arpra_set_default_precision(unsigned prec);
```

One can also retrieve and dynamically set the working precision of a range that has already been initialised by using the following functions.

```
// Retrieve the precision of a range
unsigned arpra_get_precision(arpra_range *x);

// Set the precision of a range
void arpra_set_precision(arpra_range *z, unsigned prec);
```

Setting the precision of a range using the above setter function is faster than clearing and reinitialising it. Note, however, that setting it in this manner causes the range to be reset to NaN. If one needs to change the precision of a range without invalidating it, one can simply initialise a new range with the desired precision, and then set the new range with the old one using the `arpra_set` function.

Arpra also exploits arbitrary-precision internally when computing Chebyshev and Min-Range approximations for the transcendental functions. The approximation parameters α, γ, δ, and all intermediate quantities, are all computed in a global 'internal precision', separate from the precision of the operand range. We can do this safely because only the `true_range` field is required to be rounded to the specified working precision; all computations up until the final rounding can be done in whichever precision one chooses, so it makes sense to choose a higher one. As with the default precision, and the precision of individual ranges, the user is able to retrieve and dynamically set this internal precision through Arpra function calls.

```
// Retrieve the internal precision
unsigned arpra_get_internal_precision();

// Set the internal precision
void arpra_set_internal_precision(unsigned prec);
```

Arpra also makes heavy use of arbitrary-precision floating-point internally. In addition to the exact multiplication trick mentioned previously, Arpra exploits arbitrary-precision when computing the affine terms of ranges. The centre value and deviation coefficients are all computed and stored in internal precision, rather than the working precision that the final `true_range` is computed in. The global internal precision is not necessarily the same as the precision of a given **`arpra_range`**. In fact, in practice, it should be set much higher, such as to minimise the overhead error of the AA method. Because of this, extra care is required when computing the `true_range` at the end of affine functions, to ensure that the resulting range accurately bounds the computation in the

specified precision.

To compute the `true_range`, we first compute the lower and upper bounds $z_{\text{internal}} = [(\hat{z}_c - \hat{z}_r), (\hat{z}_c + \hat{z}_r)]$ of the range in internal precision, rounding outwards. We then round these high precision bounds into the lower precision `true_range` interval, again rounding outwards. This gives `true_range` in our final working precision. To keep the deviation terms consistent, the error from rounding the internal precision z_{internal} into the working precision of `true_range` must be accounted for. To do this, the maximum of the differences between both lower bounds and between both upper bounds is accumulated with the rounding error in the new deviation term. For instance, if $z_{\text{internal}} = [0.5, 1.5]$ and `true_range` $= [0.45, 1.6]$, then 0.1 rounding error would be added to the new deviation term. The procedure is listed in algorithm 5.

Algorithm 5 Compute the bounds of a range

1: **procedure** COMPUTE_RANGE($\hat{z}$)
2: $T \leftarrow [(\hat{z}_c - \hat{z}_r), (\hat{z}_c + \hat{z}_r)]$ ▷ Internal precision
3: $\hat{z}.t \leftarrow T$ ▷ Working precision
4: $\delta \leftarrow \max((T.lo - \hat{z}.t.lo), (\hat{z}.t.hi - T.hi))$
5: $\hat{z}.d[\hat{z}.n] \leftarrow \hat{z}.d[\hat{z}.n] + \delta$
6: $\hat{z}.r \leftarrow \hat{z}.r + \delta$
7: **end procedure**

We have discussed how arbitrary-precision can help us to minimise the overhead rounding error caused by the AA method. However, since AA ranges are essentially first-order polynomials, often with many deviation terms each, a small amount of overhead rounding error is inevitable when computing the `true_range`. Furthermore, approximation error from multiplication and the transcendental functions can result in ranges that are wider than those computed with plain IA, regardless of rounding error. To reduce the impact of these additional error sources, Arpra implements a modified version of the mixed IA/AA method, which is discussed in the next section.

3.2.5 Mixed Trimmed AA/IA

After the `arpra_affine` functions, the `true_range` of the result $\hat{z}$ is computed. For vanilla AA, the `true_range` is simply the interval $[(\hat{z}_c - \hat{z}_r), (\hat{z}_c + \hat{z}_r)]$. As a consequence, the rounding error from these bound calculations, including that from the radius sum $\hat{z}_r$, and even the error incurred from range linearisation, is included as bloat in the final `true_range`. In order to trim some of this excess error, a method known as 'mixed IA/AA' can be used. This method is described by Stolfi, de Figueiredo [23], and Rump [27], and has been implemented in INTLAB [28].

The idea of mixed IA/AA is to simultaneously compute AA and IA versions of each range, and use the range information from either method to complement that of the other. Specifically, when computing some AA function $\hat{z} = \hat{f}(\hat{x})$, the IA version of that function $\bar{z} = \bar{f}(\bar{x})$ is also computed, where $\bar{x}$ is the `true_range` field of $\hat{x}$. Arpra uses the MPFI library [35] for IA functions. After that, the resulting `true_range` field of $\hat{z}$ is taken to be the intersection of the plain AA version of the `true_range`, described above, and the IA computed range $\bar{z}$.

$$\texttt{true\_range} = [(\hat{z}_c - \hat{z}_r), (\hat{z}_c + \hat{z}_r)] \cap \bar{z} \tag{3.11}$$

Doing this consistently ensures that the `true_range` field of all resulting ranges is never worse than when computing it with either AA or IA on its own. In other words, if variable correlations causes an IA range to expand, the AA range will compensate. Conversely, if approximation error causes an AA range to expand, the IA range compensates. It should be emphasised that only the `true_range` is modified; the centre and deviation terms remain the same. Consequently, this method is only beneficial for computations involving the AA transcendental functions and AA square root, since these are the only functions that considers the `true_range` field of their input range.

However, one can do better than this. The Arpra library implements modified version of mixed IA/AA, which I will refer to as 'mixed trimmed AA/IA'. In this version, the `true_range` field is computed identically as above. However, if the AA result fully contains the $\bar{z}$ range computed in IA, the new error deviation term $\hat{z}_{[k]}\epsilon_{[k]}$ of the AA operation can also be trimmed a little, so long as we maintain the condition that $\bar{z} \subseteq [(\hat{z}_c - \hat{z}_r), (\hat{z}_c + \hat{z}_r)]$, and that $\hat{z}_{[k]}$ can only be reduced to a minimum of zero. It is safe to do this because $\hat{z}_{[k]}\epsilon_{[k]}$ is a brand new and independent deviation term, whose noise symbol number k is not used elsewhere in the computation, so no correlation information is lost. The full Arpra range trimming procedure is listed in algorithm 6.

Algorithm 6 Trim a range with mixed trimmed AA/IA

```
1:  procedure TRIM_RANGE(ẑ)
2:      ia ← compute IA range
3:      if ia ⊆ ẑ.t then                          ▷ Trim error ẑ.d[ẑ.n]
4:          δ ← min((ia.lo − ẑ.t.lo), (ẑ.t.hi − ia.hi))
5:          ẑ.d[ẑ.n] ← ẑ.d[ẑ.n] − δ
6:          if ẑ.d[ẑ.n] < 0 then
7:              ẑ.d[ẑ.n] ← 0
8:          end if
9:      end if
10:     ẑ.t ← ẑ.t ∩ ia_range                      ▷ Intersect AA/IA ranges
11: end procedure
```

3.2.6 Multiplication Linearisation

Another small optimisation can be found in the Arpra multiplication routine. As mentioned earlier, multiplication of two affine ranges produces quadratic deviation terms, which need to be linearised. The trivial linearisation error estimate given in equation (1.28) is up to four times larger than the optimal bound, according to [23], and is optimal only when $\hat{x}$ and $\hat{y}$ share no deviation terms. In order to tighten the range of affine multiplication, a new error estimate was discussed in [27], based on the ideas presented in [36].

Rather than simply multiplying the radii of the operands $\hat{x}$ and $\hat{y}$, as in equation (1.28), the error estimate $\hat{z}_{[k]}$ is computed as follows.

$$\hat{z}_{[k]} = \max\left(\sum_{i=1}^{n}\left|\hat{z}^{-}_{[i]}\right|, \sum_{i=1}^{n}\left|\hat{z}^{+}_{[i]}\right|\right) + \sum_{i<j}\left|\hat{x}_{[i]}\hat{y}_{[j]} + \hat{x}_{[j]}\hat{y}_{[i]}\right| \tag{3.12}$$

where $\hat{z}^{+}$ is the vector of all deviation coefficients $\hat{z}_{[i]} = \hat{x}_{[i]}\hat{y}_{[i]}$, with negative components set to zero, and $\hat{z}^{-}$ is the vector of all $\hat{z}_{[i]}$, with positive components set to zero.

$$\begin{aligned} &\forall i \in \{1, ..., n\}, \\ &\hat{z}^{+}_{[i]} = \begin{cases} \hat{x}_{[i]}\hat{y}_{[i]} & \text{if } \hat{x}_{[i]}\hat{y}_{[i]} > 0 \\ 0 & \text{otherwise} \end{cases} \\ &\hat{z}^{-}_{[i]} = \begin{cases} \hat{x}_{[i]}\hat{y}_{[i]} & \text{if } \hat{x}_{[i]}\hat{y}_{[i]} < 0 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{3.13}$$

This gives our final equation for affine multiplication in Arpra.

$$\begin{aligned} \hat{x}\hat{y} = (\hat{x}_c\hat{y}_c) &+ \sum_{i=1}^{n}(\hat{x}_c\hat{y}_{[i]} + \hat{y}_c\hat{x}_{[i]})\epsilon_{[i]} \\ &+ \left(\max\left(\sum_{i=1}^{n}\left|\hat{z}^{-}_{[i]}\right|, \sum_{i=1}^{n}\left|\hat{z}^{+}_{[i]}\right|\right) + \sum_{i<j}\left|\hat{x}_{[i]}\hat{y}_{[j]} + \hat{x}_{[j]}\hat{y}_{[i]}\right|\right)\epsilon_{[k]} \end{aligned} \tag{3.14}$$

In random experiments by Rump, in [27], this error estimate is found to be around 2.06 times larger than the optimal bound. However, since the error term is only dominant when the operand centres are small relative to the size of their radii, the average error reduction is relatively low. Furthermore, since only the error term $\hat{z}_{[k]}\epsilon_{[k]}$ is reduced by this method, which is trimmed later anyway due to the mixed trimmed AA/IA method, its usefulness appears to be questionable at first glance. It is nonetheless included and tested in Arpra.

3.2.7 Term Reduction Functions

In long chained computations, it is often the case that many deviation terms are accumulated in AA ranges. Since a new deviation term is added after practically all mathematical functions, and are only removed in the unlikely event that the deviation coefficient becomes exactly zero, an AA computation can often grind to a halt after a short while, due to the computational overhead. No other AA implementation handles this eventuality, to the author's knowledge, despite the existence of a solution discussed by Stolfi and de Figueiredo in [23].

In order to contend with this problem, one needs to implement a so-called 'term condensing' function, which sums the absolute value of selected deviation coefficients into a new coefficient, corresponding to a new noise symbol $\epsilon_{[k]}$, and removes the old condensed terms. For example, if one has an AA range $\hat{x}$, with deviation terms $(1.5\epsilon_{[1]}, 8\epsilon_{[2]}, 2\epsilon_{[3]}, -4\epsilon_{[4]}, 1\epsilon_{[5]})$, one can reduce the $\epsilon_{[1]}$, $\epsilon_{[3]}$ and $\epsilon_{[4]}$ terms of $\hat{x}$ in a new range $\hat{z}$, with just three deviation terms.

$$\hat{z} = \hat{x}_c + 8\epsilon_{[2]} + 1\epsilon_{[5]} + (|1.5| + |2| + |-4|)\epsilon_{[k]} \tag{3.15}$$

Although some of the correlation information in $\hat{x}$ is potentially lost in $\hat{z}$, this is a safe operation, since $\epsilon_{[k]}$ is a new and independent noise symbol, and the actual range $\hat{z}_c \pm \hat{z}_r$ of $\hat{z}$ is not smaller than $\hat{x}$.

Arpra provides three variants of term condense function. The first variant condenses the last n deviation terms of an **arpra_range**. The second variant reduces all terms whose deviation coefficient magnitude is smaller than a given threshold. The third variant reduces all terms whose deviation coefficient magnitude is smaller than some given fraction of the range's radius.

```
// Reduce last n terms
void arpra_reduce_last_n(arpra_range *z, unsigned n);

// Reduce terms smaller than absolute threshold
void arpra_reduce_small_abs(arpra_range *z, mpfr_t thr);

// Reduce terms smaller than relative threshold
void arpra_reduce_small_rel(arpra_range *z, mpfr_t thr);
```

The `arpra_reduce_last_n` function, listed in algorithm 7, can be considered a 'lossless' condensing function, if used correctly. That is to say that, if the noise symbols in the last n deviation terms are not present in any other **arpra_range**, this function is guaranteed to preserve all correlation information when condensing terms. There are a number of situations in which the last n terms of a range are independent. For instance, if only a single **arpra_range**

is returned by any given function, then all noise symbols introduced by the intermediate computations in that function are guaranteed to be only present in the returned range. When computing the condensed deviation coefficient, Arpra uses MPFR's correctly rounded sum function, minimising the rounding error incurred by the operation.

Alternatively, one can use either `arpra_reduce_small_abs`, listed in algorithm 8, or `arpra_reduce_small_rel`, listed in algorithm 9, if some loss of correlation information is acceptable. These condensing functions can be considered 'lossy', since there is no direct control over which deviations terms are condensed, and some of these terms may consequently be correlated ones. However, this matters less when the deviation coefficients are small. If the majority of deviation coefficients are close to zero, with just a few coefficients contributing to the majority of the radius, then the loss of correlation information will be minimal when these low magnitude terms are condensed.

Algorithm 7 Condense the last n terms

```
1:  procedure REDUCE_LAST_N(ẑ, n)
2:      if (ẑ.n = 0) ∨ (n = 0) then                    ▷ Sanity check
3:          return
4:      else if n > ẑ.n then
5:          n ← ẑ.n
6:      end if
7:      sum ← 0
8:      i_0 ← ẑ.n − n + 1
9:      for i ← i_0 to ẑ.n do                          ▷ Condense last n terms
10:         sum ← sum + |ẑ.d[i]|                       ▷ Rounding towards +∞
11:     end for
12:     ẑ.d[i_0] ← sum                                 ▷ Append condensed term
13:     ẑ.s[i_0] ← a new noise symbol
14:     ẑ.n ← i_0
15: end procedure
```

In this section, we have discussed the key features of the Arpra library, including its intrinsics, the mixed trimmed AA/IA model it implements, and how arbitrary precision is utilised by it. The following section documents the performance of Arpra, with regards to range correctness and tightness.

3.3 Accuracy of the Arpra Library

In order to test the accuracy of ranges computed by Arpra, compared to those computed with IA, functions are tested on $n = 100000$ randomly generated operand test cases. Tests are performed both for plain AA, and for mixed IA/AA. The precision of all MPFI ranges is 24, corresponding to IEEE-754

Algorithm 8 Condense terms smaller than absolute threshold

```
1: procedure REDUCE_SMALL_ABS(ẑ, thr)
2:     if (ẑ.n = 0) ∨ (thr ≤ 0) then                    ▷ Sanity check
3:         return
4:     end if
5:     n ← 0
6:     for i ← 1 to ẑ.n do                              ▷ Move large terms to front
7:         if ẑ.d[i] ≥ thr then
8:             n ← n + 1
9:             if i > n then
10:                swap ẑ.d[n] and ẑ.d[i]
11:                swap ẑ.s[n] and ẑ.s[i]
12:            end if
13:        end if
14:    end for
15:    reduce_last_n(ẑ, (ẑ.n − n))                      ▷ Condense small terms
16: end procedure
```

Algorithm 9 Condense terms smaller than relative threshold

```
1: procedure REDUCE_SMALL_REL(ẑ, thr)
2:     if (ẑ.n = 0) ∨ (thr ≤ 0) then                    ▷ Sanity check
3:         return
4:     end if
5:     thr ← thr ∗ ẑ.r                                  ▷ Condense small terms
6:     reduce_small_abs(ẑ, thr)
7: end procedure
```

single-precision numbers. Arpra's working precision is set to 24, its internal precision is set to 256, and all transcendental functions use the Chebyshev approximation method. Each operand range has a random centre value, and from five to nine small deviation terms drawn from $[-1, 1]$. The corresponding IA function from the MPFI library [35] is also computed on the `true_range` fields of the operands. In all tests, the diameter of the Arpra result is computed, relative to that of the IA result. For testing purposes, a NaN result in IA is considered equal to a NaN result in Arpra.

Univariate functions are tested once on each test case. Bivariate functions are tested three times on each test case, with different correlation scenarios, in order to determine how the strength of operand correlation affects the resulting range. In the uncorrelated operands scenario, the noise symbol sets of the operands are mutually exclusive. In the partially correlated operands scenario, the operands have three noise symbols in common. In the correlated operands scenario, the operands have p noise symbols in common, where p is the size of the smallest noise symbol set of both operands. Ranges computed by univariate functions, and bivariate functions of uncorrelated operands, are checked to ensure that they are equal to, or fully contain, the computed IA ranges. This is the correct behaviour, since AA should only be able to compute ranges tighter than IA does when function operands are at least partially correlated. The results for plain AA tests will be presented first, followed by the results for mixed IA/AA.

3.3.1 Plain AA Results

The relative diameters of results computed by plain AA univariate functions are plotted in figure 3.1. Note that the ranges computed by these functions are never of smaller diameter than those computed with IA. This is because deviation term cancellation does not occur in univariate AA functions. Instead, as expected, the results of nonlinear AA functions are often slightly larger than the IA results, since the result includes error from its linear approximation.

The relative diameters of results from plain AA bivariate functions are plotted in figure 3.2. These results clearly illustrate how deviation term cancellation improves the resulting range, when the operands of a function are correlated. Note how the distribution of relative diameters progressively spreads towards zero as operand correlation increases. Results from the linear addition and subtraction functions can be represented by an AA range without approximation, and take advantage of deviation term cancellation, leading to huge improvements over IA results. Nonlinear function results incur approximation error, and are often seen to be wider than IA results in the plot. However, deviation term cancellation also can produce better results than IA, providing the

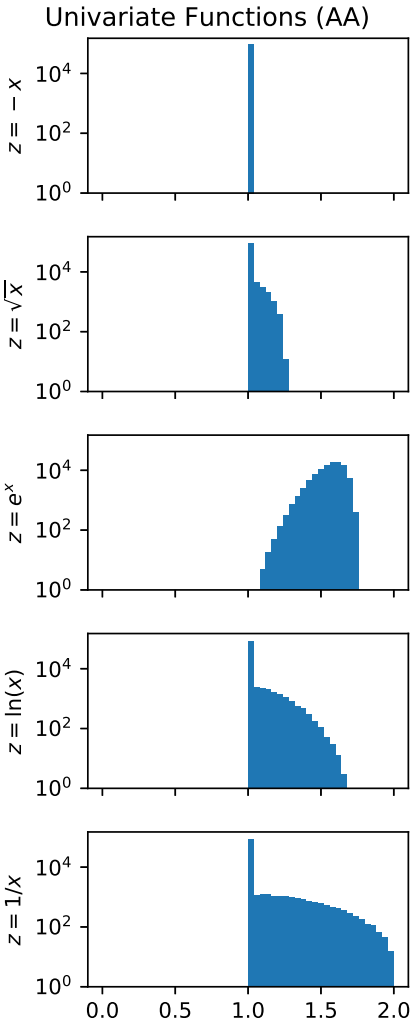

Figure 3.1: Histogram plots of the relative result diameters of univariate AA functions. Shown from the top are results for negation, square root, exponential, logarithm and inverse. Note how nonlinear function results computed with plain AA can often be larger than those computed with IA

operands are correlated. Finally, as expected, the improved linearisation error estimate for multiplication, given in equation (3.12), performs marginally better than the estimate given in equation (1.28). This modest improvement is useful enough to be included in the Arpra library.

3.3.2 Mixed IA/AA Results

The relative diameters of results computed by univariate mixed IA/AA functions are plotted in figure 3.3. Note that these functions all compute ranges with a relative diameter of one, and are thus equal in diameter to those computed in IA. This is expected, since there can be no deviation term cancellation, and the IA/AA range intersection step in algorithm 6 prevents the AA range from being larger than the IA range.

The relative diameters of bivariate mixed IA/AA function results are plotted in figure 3.4, for all three correlation scenarios. As expected, the relative diameters of results computed from uncorrelated operands are all one, for the same reason as in the previous paragraph. Also, as expected, the relative diameters of results are successively lower, on average, in each correlation scenario, since more deviation terms are able to cancel out.

What is perhaps unexpected in figure 3.4 is that multiplication using the alternative linearisation error estimate from equation (3.12) is still better, on average, than multiplication using the trivial error estimate from equation (1.28), for correlated operands. This is in spite of the error term trimming procedure, given in algorithm 6, which one might expect to trim the oversized ranges of both products back down to the same width. This occurs because ranges are only trimmed when the AA range fully contains the IA range. Since deviation terms can cancel out, due to correlation, the resulting range can still be smaller than the IA range, even with the added linearisation error [23]. Consequently, range trimming does not occur, and a small difference in diameter remains between the two products.

In this chapter, we have presented this work's main software contribution, the Arpra library, and discussed its internal operation. We described Arpra's novel methods for computing correct rounding error, and exploiting arbitrary precision floating-point arithmetic for its internal computations using the MPFR library. We were introduced to Arpra's novel mixed trimmed IA/AA method, and deviation term condensing functions. Finally, we saw how ranges computed by the three Arpra methods compare to those computed by IA when computing common univariate and bivariate mathematical functions. The next step is to test how it performs in real problems. We now proceed to testing Arpra on a popular chaotic dynamical system known as the Hénon map in the next chapter.

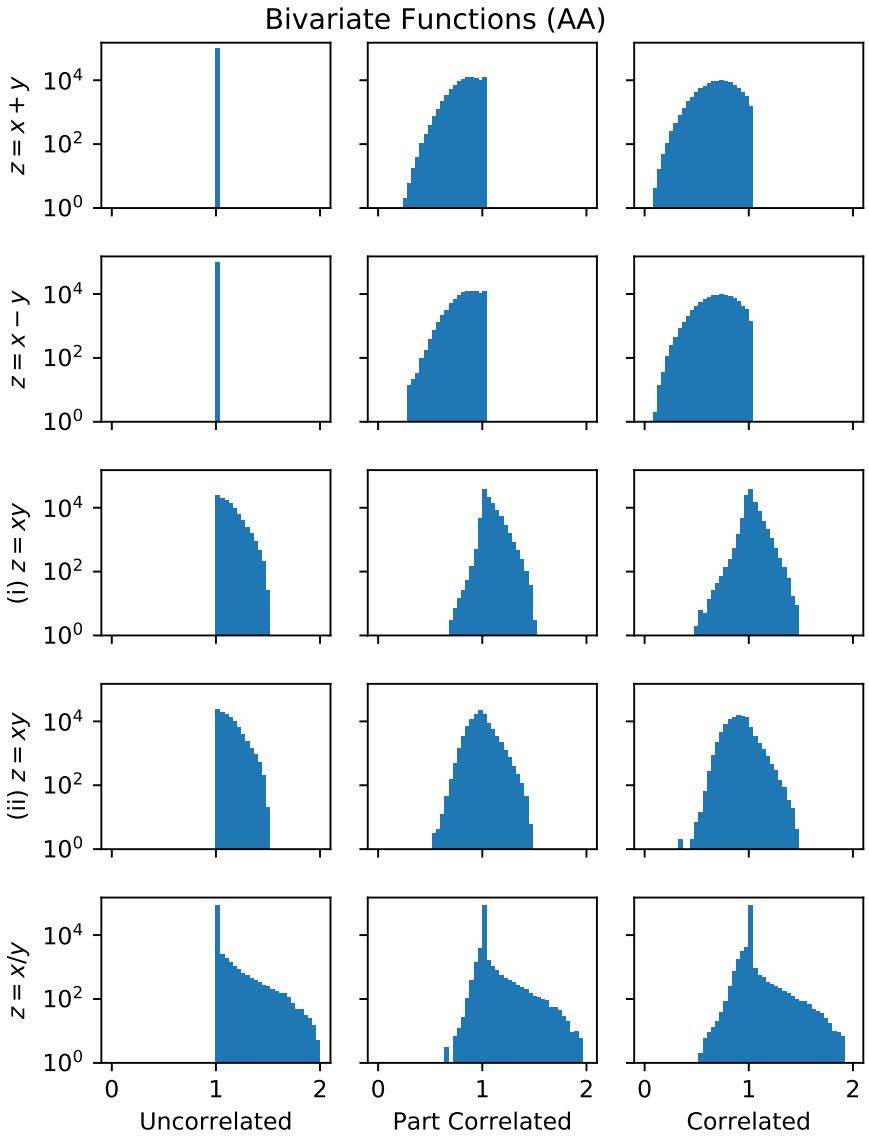

Figure 3.2: Histogram plots of the relative result diameters of bivariate AA functions. Shown from the top are results for addition, subtraction, multiplication (i) with the trivial linearisation error estimate in equation (1.28), multiplication (ii) with improved linearisation error estimate in equation (3.12), and division. Shown from the left are computations with uncorrelated operands, partially correlated operands, and fully correlated operands. Note how nonlinear function results computed with plain AA can often be larger than those computed with IA, but not when operands are correlated or when the AA result representation has zero internal error. Also note how the relative AA result diameter is lower, on average, when operands are more correlated.

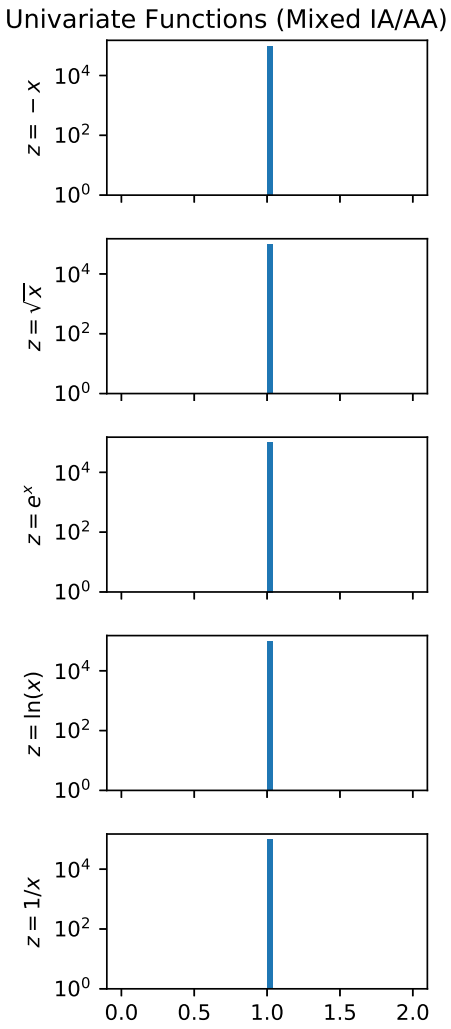

Figure 3.3: Histogram plots of the relative result diameters of univariate mixed IA/AA functions. Shown from the top are results for negation, square root, exponential, logarithm and inverse. Note that the relative diameters for all n tests are equal to one, indicating that the mixed IA/AA and IA results computed by these functions are of identical diameter.

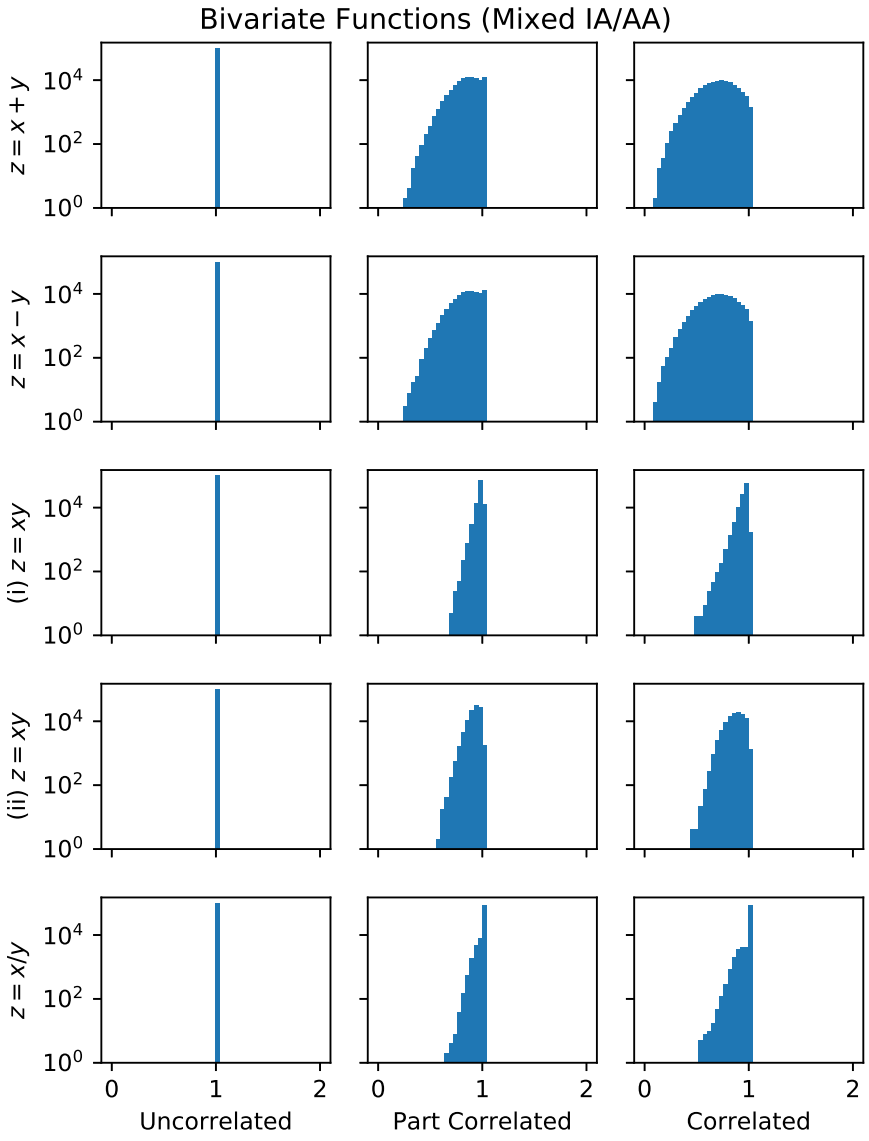

Figure 3.4: Histogram plots of the relative result diameters of bivariate mixed IA/AA functions. Shown from the top are results for addition, subtraction, multiplication (i) with the trivial linearisation error estimate in equation (1.28), multiplication (ii) with improved linearisation error estimate in equation (3.12), and division. Shown from the left are computations with uncorrelated operands, partially correlated operands, and fully correlated operands. Note that, unlike the plain AA results, the mixed IA/AA results are never wider than the results computed with IA.

Chapter 4

The Hénon Map

In this chapter, we test the performance of the Arpra library [26] on the simulation of a dynamical system with known stability properties and different dynamical regimes. The Hénon map [37] is a two-dimensional discrete-time dynamical system, and is a reduced version of the three dimensional Lorenz system [38]. It has trajectories ranging from stable limit cycles to chaotic attractors, depending on the choice of parameters. In addition to evaluating Arpra, this allows us to observe how system stability affects the growth of Arpra ranges.

The model was used by Rump and Kashiwagi in [27] to test the INTLAB range analysis package for MATLAB, making it a good first benchmark to see how the Arpra library compares. Unlike Arpra, the working precision of INTLAB is fixed to IEEE-754 double-precision, and computations do not benefit from extended precision in internal calculations. Although INTLAB implements the mixed IA/AA method discussed in chapter 3, it does not implement the error term trimming technique used in the mixed trimmed IA/AA method. Furthermore, INTLAB does not implement any deviation term condensing functions, so simulations can eventually become bogged down due to excessive memory use.

The Hénon map is defined by the following equations, where x_i and y_i are the state variables at the ith iteration, while α and β are the constant parameters.

$$\begin{aligned} x_{i+1} &= 1 - \alpha x_i^2 + y_i \\ y_{i+1} &= \beta x_i \end{aligned} \tag{4.1}$$

In the 'classical' Hénon map, $\alpha = 1.4$ and $\beta = 0.3$, resulting in a chaotic system. However, the system is also known to have a stable periodic orbit below around $\alpha = 1.06$, and is increasingly stable as α is reduced further. Note that this model does not require transcendental functions to implement. As

a consequence, besides a small amount of error introduced in multiplications, very little approximation error is incurred by the AA method itself, with the majority being the rounding error from floating-point arithmetic.

We begin by testing how well the Arpra library competes with the INTLAB package, with Arpra using either the plain AA, mixed IA/AA or mixed trimmed IA/AA methods. We then test to see how the radius of Arpra ranges change as the internal precision increases. Finally, we see how Arpra performs as the system becomes more chaotic. For the following experiments, β is fixed to 0.3. Both x and y are initialised as ranges centred on zero with small initial radii of 0.00001. All simulations are run for $n = 500$ iterations, using version 0.1 of the Arpra library [26], and version 11 of INTLAB [28].

4.1 Method Evaluation

We first test Arpra's plain AA functionality against the INTLAB implementation of IA. That is to say that Arpra's mixed IA/AA and range trimming functionality is disabled for this experiment, to demonstrate only the improvements of standard AA over IA. The α parameter of the Hénon map is set to 1.057, meaning the model is close to chaotic, but still locally stable. The results of the AA and IA runs are plotted in figure 4.1.

From the plots we see that the ranges computed in IA almost immediately explode to infinite width between iterations $i = 30$ to 40, despite the global stability of the underlying model. In agreement with [27], we also see that ranges computed with AA initially grow for a short while, but then shrink back below their initial width as the trajectory converges to a periodic orbit.

Since Arpra is capable of more precise methods than just plain AA, it is expected that ranges computed with these advanced methods should grow comparatively slower in unstable phase space, and also shrink faster in stable phase space. Arpra implements the AA, mixed IA/AA and mixed trimmed IA/AA range analysis methods. We will now evaluate these implemented methods against the INTLAB implementation of the mixed IA/AA method, in order to compare the tightness of ranges computed by each. For fairness, the working precision and internal precision of Arpra are both set to 53, corresponding to the IEEE-754 double-precision numbers used by INTLAB. As before, $\alpha = 1.057$. The radius differences between the INTLAB mixed IA/AA ranges and the three Arpra methods' ranges are plotted in figure 4.2.

As the left column of figure 4.2 shows, Arpra with the plain AA method performs worse than INTLAB with the mixed IA/AA method, but only marginally. This is because the mixed IA/AA method is only beneficial when transcendental functions are used. Since only the Chebyshev and Min-Range approximations

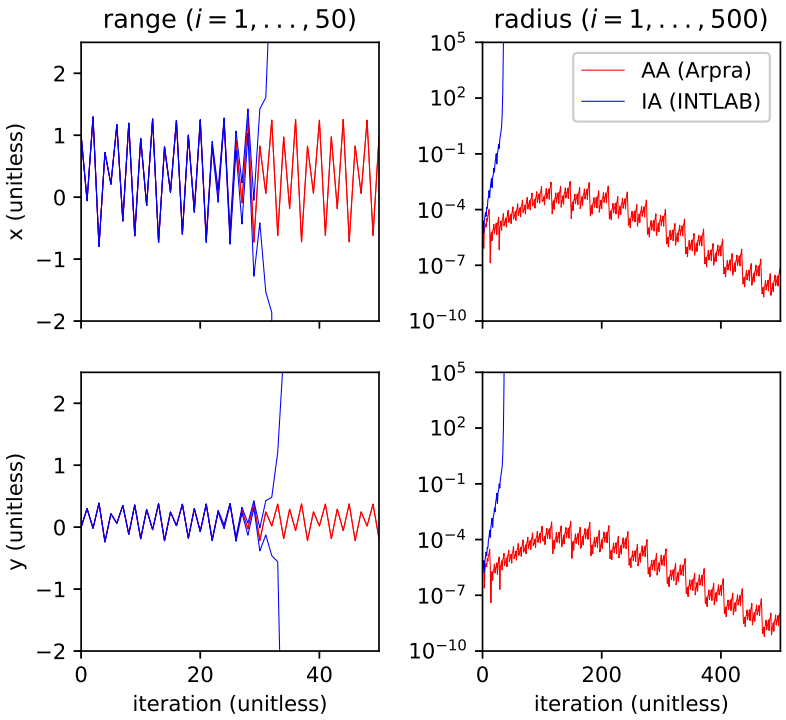

Figure 4.1: Range analysis of the Hénon map, computed by Arpra using plain AA, and by INTLAB using IA. The left column shows the error bounds of the variables x and y for the first 50 iterations. The right column shows the radii of the ranges, in log scale, for all 500 iterations. $\alpha = 1.057$ and $\beta = 0.3$. Note how the IA ranges begin to explode at around $i = 30$, while the AA ranges shrink after a short growth period.

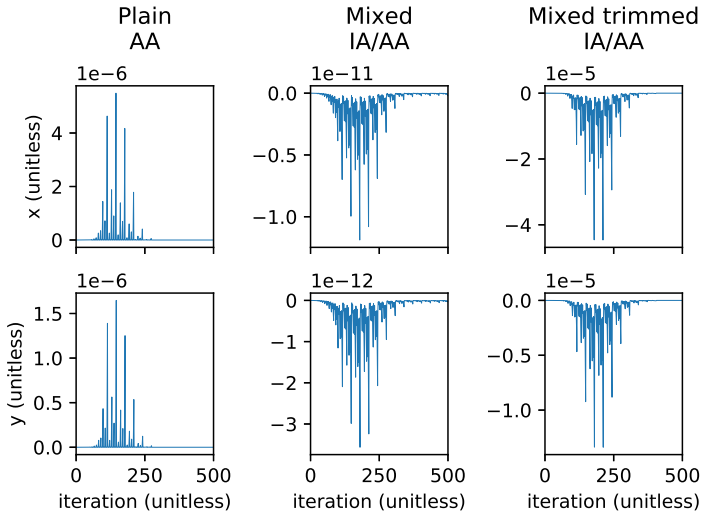

Figure 4.2: The radius difference between Hénon map ranges computed with mixed IA/AA in INTLAB, and ranges computed with AA (left), mixed IA/AA (centre), mixed trimmed IA/AA (right) in Arpra. $\alpha = 1.057$ and $\beta = 0.3$. Negative values indicate that Arpra ranges are slimmer than INTLAB ranges, and vice versa.

of transcendental functions make use of the intersected IA/AA ranges, algebraic functions, such as those used in the Hénon map, are not applicable. The remaining difference is due to the fact that, although the tighter mixed IA/AA ranges do not affect the computation of subsequent ranges in algebraic systems, they make for better quality output data than pure AA ranges do.

The middle column of figure 4.2 compares Arpra's mixed IA/AA method to the mixed IA/AA method of INTLAB. Here, the radius of ranges computed by Arpra qualitatively matches those computed by INTLAB. On close inspection, Arpra ranges have a marginally smaller radius than INTLAB ranges do. This can be attributed to Arpra's use of the improved multiplication error estimate from equation (3.12), and also because Arpra computes correct rounding error bounds as a function of ULP, as seen in equation (3.10).

In the right column of 4.2, we see the difference between Arpra's mixed trimmed IA/AA method and the mixed IA/AA method in INTLAB. Here, Arpra outperforms INTLAB by a modest amount as the trajectory converges to its stable orbit. It should be noted that the 'trimmed' aspect of Arpra's mixed trimmed IA/AA method is only advantageous when the new numerical error terms from affine functions are large. As seen in algorithm 6, the new error term cannot be trimmed to below zero. Therefore, if the new term is the sum of tiny rounding errors alone, with no larger approximation error due to

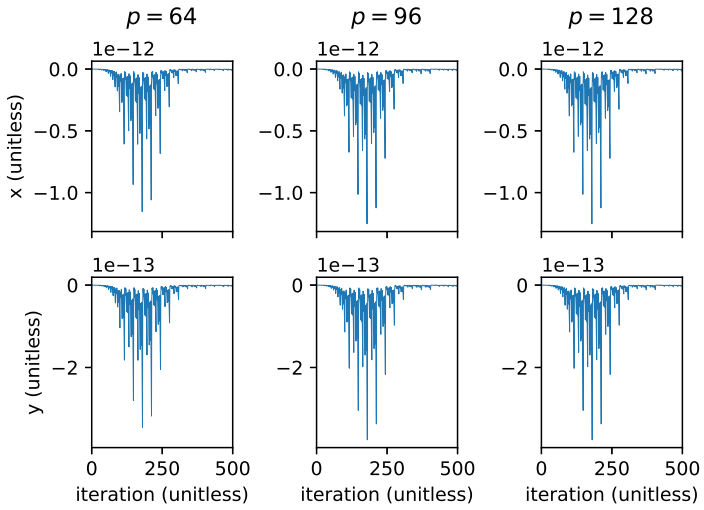

Figure 4.3: The radius difference between Hénon map ranges computed with the internal precision p equal to the working precision of $w = 53$ bits, and ranges computed with $p = 64$ (left), $p = 96$ (centre) and $p = 128$ (right) bits. $\alpha = 1.057$ and $\beta = 0.3$. Negative values indicate that ranges with $p > w$ bits of internal precision are slimmer than ranges with $p = w$ bits of internal precision.

nonlinear functions, the mixed trimmed IA/AA method becomes ineffective. In the next section, we see how increasing Arpra's internal floating-point precision affects the quality of computed ranges.

4.2 Internal Precision

As seen in chapter 3, Arpra also has the advantage of being able to dynamically change the precision of its floating-point numbers to arbitrary values. The internal precision of Arpra is distinct from its externally facing working precision, in order to reduce the overhead AA method error, yet maintain the effective precision of the emulated floating-point type. To test how internal precision affects range tightness, the Hénon map is simulated using Arpra's mixed trimmed IA/AA method. The working precision is set to $w = 53$, as before, but the internal precision p is set to 64, 96 and 128 bits. Once again, $\alpha = 1.057$. The radii of the computed ranges with each internal precision value are compared to those from when the internal precision is equal to the working precision, and the differences are plotted in figure 4.3.

From the plot, we see that the small increase of internal precision to $p = 64$ decreases the radius of computed ranges slightly, but the further increase to $p = 96$ has diminished effect, while the difference is negligible for $p = 128$.

Rump's example [18] [19] implies that it is non-trivial to determine how the accuracy of floating-point arithmetic changes as the precision increases, since the mapping from precision to accuracy is not continuous. It stands to reason that ranges with many deviation terms, and those computed by nonlinear functions, would benefit more from a higher internal precision than the Hénon map does.

Despite this, there is clearly a ceiling where the increases in accuracy begin to plateau. This suggests that a more algorithmic way of finding the optimal internal precision is possible. A potential solution, used by the MPFR library [17] mentioned in chapter 3, is to use Ziv's strategy [33] as a heuristic when setting the internal precision. The idea would be to start at some base precision, such as Arpra's default working precision, and incrementally raise the internal precision until the Table Maker's Dilemma (TMD) [31] does not occur when computing the `true_range` field of affine ranges. However, a problem with this is that affine ranges are constantly changing, with deviation terms being added, and sometimes removed, and the internal precision set by such a method would need constant updating.

In the next section, we investigate how the stability of the Hénon map affects the radius of computed ranges.

4.3 Chaotic Systems

Trajectories in chaotic dynamical systems are, by definition, highly sensitive to perturbations in the initial state, and these perturbations can propagate in unpredictable ways. As a result, ranges representing the state of these systems can grow very wide, very quickly. The Hénon map is known to exhibit chaotic behaviour with $\beta = 0.3$ and α approaching around 1.06. In this experiment, α is set to 1.057, 1.058 and 1.059, to see how changes in the local stability of the Hénon map affect the radius of computed ranges. As before, the mixed trimmed IA/AA method is used. The radii of computed ranges are plotted for these α values in figure 4.4.

In the left column, as we saw earlier, the radii of ranges computed in the stable Hénon map initially grow, as the trajectory converges to its stable orbit, but begin to shrink later once the stable orbit is reached. As α is increased, the Hénon map enters a chaotic regime, and the small perturbations represented by the affine ranges are amplified in unpredictable ways. This results in the runaway growth of the bounding range we see in the centre and right columns of figure 4.4. The rate of range growth is dependent on how sensitive to state perturbations, or rather how chaotic, the system is; higher values of α result in faster interval growth.

This effect poses a problem for analysing systems with singularities. For

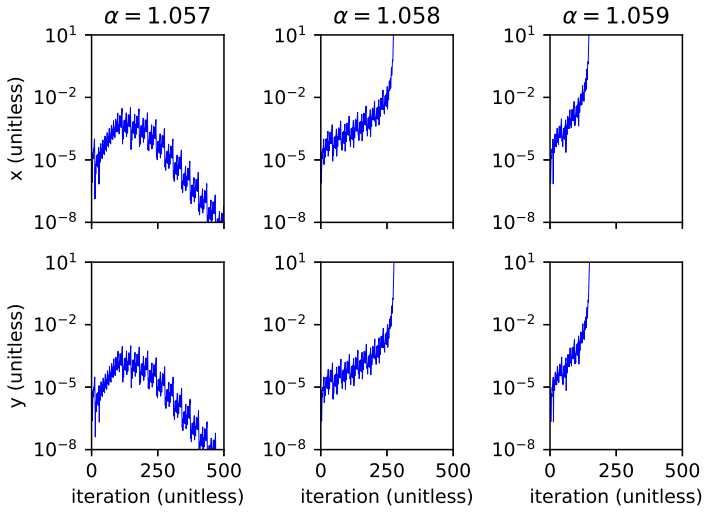

Figure 4.4: The radius of Hénon map ranges for $\alpha = 1.057$ (left), $\alpha = 1.058$ (centre) and $\alpha = 1.059$ (right), with $\beta = 0.3$. Note how the ranges grow to infinite width faster as the Hénon map becomes more unstable.

example, in range analysis methods such as IA and AA, dividing a range or a scalar with a range which straddles zero is an invalid operation, just as division by zero is invalid in real arithmetic. If one analyses a sufficiently chaotic system involving division, it is likely that the computed ranges will quickly grow large enough such that they inevitably straddle zero, and cause an invalid operation. No matter how unlikely a singularity is for a given floating-point computation in practice, the so-called 'fundamental theorem of interval arithmetic' must always be satisfied. That is to say that the range computed by an operation must contain the result for every single combination of operands inside the operand ranges, including a zero divisor.

Another problem with chaotic systems is that, even if a singularity does not occur, there remains the question of how useful the bounding ranges are when they grow so fast. Although the AA method is behaving entirely correctly, bounding all possible trajectories of a simulation, the likelihood that these worst cases will ever occur in practice is vanishingly small. The AA method was chosen early on in the project because the priority was to see exactly how bad the worst cases of error is in a numerical simulation. In this way, any trajectory divergence, no matter how severe, could be explained by this form of analysis. A more conservative analysis of, for example, numerical software libraries or safety-critical systems would see benefit in using this method over IA or plain AA. However, if highly unlikely worst-case behaviour is of less concern, a more statistical approach looking at the most likely deviations instead of the worst-

case bounds might be more informative.

We now test the effectiveness of the deviation term condensing functions, for reducing the computational overhead of the Arpra library.

4.4 Deviation Term Reduction

In Arpra, after 500 iterations of the Hénon map, the x and y affine ranges each contain approximately 3500 deviation terms, which is enough to cause noticeable slowdown, especially in longer iterative computations. Since INTLAB is implemented in MATLAB, it is able to temporarily alleviate this bottleneck by making use of the processor's parallel vector instruction set, although it too eventually becomes bogged down as well. Arpra, on the other hand, uses MPFR arbitrary-precision floating-point numbers, which are implemented in software, and therefore cannot make use of the special vector instructions of the processor. In order to solve this issue, Arpra implements the deviation term condensing functions, as discussed in chapter 3.

Among these functions is the lossless `arpra_reduce_last_n` function, which is used to condense the last n deviation terms of an affine range, with the assumption that their noise symbols are not shared with any other affine range in use. Although it is considered lossless, this function can introduce additional rounding error. To illustrate this, the Hénon map simulation is run for 500 iterations, condensing all independent terms in x and y after each iteration. For this experiment, $\alpha = 1.057$, and the internal precision is set to 128. The difference between the radii of ranges condensed with `arpra_reduce_last_n` and the radii of those computed without condensing is plotted in figure 4.5. We can see that `arpra_reduce_last_n` has a small overhead cost, in terms of radius growth, but the time and memory performance gains are significant. A comparison of Arpra term condensing functions is given in table 4.1.

Despite the performance improvements due to `arpra_reduce_last_n`, the number of deviation terms still grows linearly with each iteration. As a result, the computation will still eventually slow down, but much later. Another possible lossless term condensing method, not implemented in Arpra, would involve scanning affine ranges for deviation coefficients equal to zero, and removing them. Such a coefficient can appear due to deviation term cancellation, or when a new term is added with zero numerical error. Such a method would incur zero overhead rounding error, since zero deviation coefficients do not contribute to the range in any way.

As another solution, Arpra implements the `arpra_reduce_small_abs` and `arpra_reduce_small_rel` functions. They condense all terms whose deviation coefficient is less than some absolute value, or some fraction of the

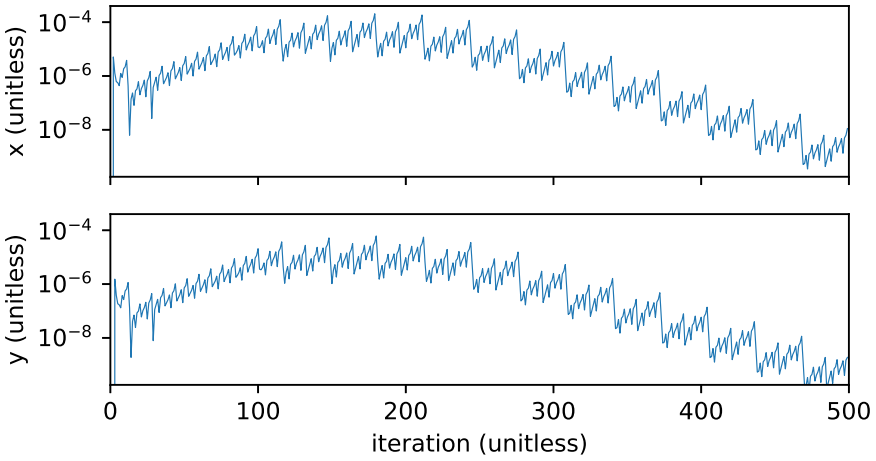

Figure 4.5: The difference between the radius of Hénon map ranges condensed with `arpra_reduce_last_n`, and those computed without term condensing. $\alpha = 1.057$ and $\beta = 0.3$. Deviation term count is reduced considerably, at the expense of minor radius growth.

range's radius, respectively. Although `arpra_reduce_small_abs` allows finer control of term condensing, which may be desirable in some applications, here it makes sense to condense the deviation terms that are most weakly contributing to the radius.

The following experiment tests the `arpra_reduce_small_rel` function with various relative thresholds, calling the condensing routine on x and y after each of the 500 Hénon map iterations, with $\alpha = 1.057$ and 128 bits of internal precision. The differences between the radii of condensed and non-condensed ranges is plotted in figure 4.6.

What is immediately obvious from this plot is that, since correlation information is not maintained by the lossy `arpra_reduce_small` functions, the ranges begin to grow quickly in a manner not dissimilar to IA ranges. While these functions are great for removing lesser deviation terms en masse, it is clear that these functions should be used sparingly. Since earlier experiments have shown that affine ranges shrink in stable system regimes, this suggests that small term condensation should ideally occur when the system trajectory is sufficiently stable, to minimise radius growth. Alternatively, the `arpra_reduce_small` functions can be called at some given epoch in the iteration. In the next experiment, we test the Hénon map with `arpra_reduce_small_rel` as before, with $\alpha = 1.057$ and 128 bits of internal precision, but this time term condensing occurs after 50 iterations. The radii differences from non-condensed ranges are plotted in figure 4.7.

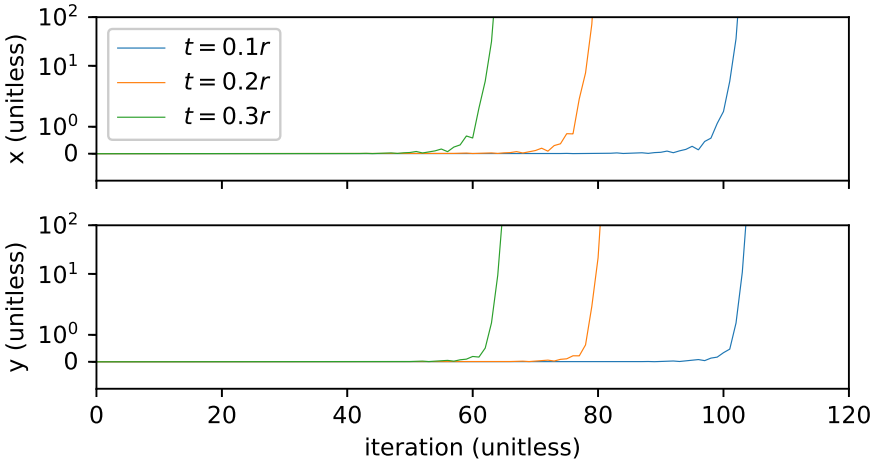

Figure 4.6: The difference between the radius of Hénon map ranges condensed with `arpra_reduce_small_rel` after each iteration, and those computed without term condensing. $\alpha = 1.057$ and $\beta = 0.3$. The thresholds t used above are 0.1 (blue), 0.2 (red) and 0.3 (green) multiplied by the radius r. Deviation terms are condensed more aggressively than in `arpra_reduce_last_n`, but at far greater expense in terms of radius growth.

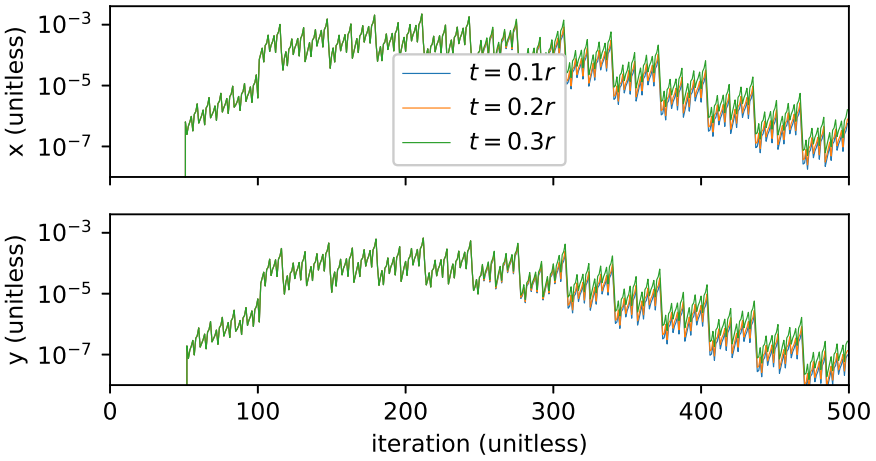

Figure 4.7: The difference between the radius of Hénon map ranges condensed with `arpra_reduce_small_rel` after 50 iteration epochs, and those computed without term condensing. $\alpha = 1.057$ and $\beta = 0.3$. The thresholds t used above are 0.1 (blue), 0.2 (red) and 0.3 (green) multiplied by the radius r. Note that ranges do not explode if the `arpra_reduce_small` functions are used sparingly.

Method	Run Time	x Terms	y Terms	Mallocs	Malloc Bytes
none	1m 36s	3502	3497	9,680,433	589,107,669
last n	3.5s	501	501	1,453,531	86,223,965
small $(0.1r)$	1.3s	7	7	1,033,419	60,897,173
small $(0.2r)$	1.3s	3	3	1,023,563	60,295,141
small $(0.3r)$	1.3s	2	2	1,021,365	60,160,837

Table 4.1: Performance comparison of deviation term condensing functions in Arpra. The condense epoch for `arpra_reduce_small_rel` is 50 iterations. Term counts are taken at the end of the computation. Heap memory allocation information was obtained using Valgrind [39].

These ranges are clearly better than those in figure 4.6. Although the ranges grow larger than the ones computed with `arpra_reduce_last_n` in figure 4.5, the number of active deviation terms is far lower. Table 4.1 shows the time and memory performance of each term reduction strategy in Arpra. From this data, we can see that the vast majority of deviation terms in the Hénon map x and y ranges have magnitudes less than 10% of their radius. We also see from these results that allowing the `arpra_reduce_small` functions to periodically remove lesser deviation terms can greatly improve the time and memory performance of the range analysis, but excessive use deteriorates the quality of ranges. Using even smaller thresholds reduces this deterioration, but less of the range's deviation terms are merged. This may not be a problem in computations which naturally produce many near-zero deviation terms. Therefore, some combination of all term reduction strategies seems desirable, where independent terms are condensed as they appear, but small terms are swept away when appropriate.

In this chapter, we have demonstrated the features of the Arpra library by analysing the popular Hénon map in both the stable orbit and chaotic regimes. We found that AA performs well when analysing stable systems, but its usefulness is debatable for chaotic systems. We found that the mixed trimmed IA/AA method used by Arpra marginally outperforms the mixed IA/AA method INTLAB uses, despite the lack of transcendental functions used in the Hénon map. We showed the effectiveness of the internal precision feature of Arpra on the Hénon map, and suggested a means of algorithmically tuning it to best make the most of it. Finally, we tested the effectiveness of Arpra's deviation term condensing functions, finding that overuse of the `arpra_reduce_small` methods rapidly deteriorates range quality, while combining term reduction strategies significantly improves computational performance. In the next chapter, we begin the analysis of a nonlinear SNN model.

Chapter 5

Spiking Neural Networks

A spiking neural network (SNN) model is distinct from the typical neural network model one might find in the machine learning literature. Rather than being a simple weighted sum of inputs, followed by some abstract activation function, they are modelled realistically as systems of coupled differential equations, which are capable of exhibiting complex spiking dynamics. With these models, computational neuroscientists can simulate anything from tiny peripheral neural circuits to vast cortical networks 'in silico', rather than investing time and resources in empirical experiments. Furthermore, there has been growing interest in the field of neuromorphic computing [9] [40] [41]. Here, realistic SNN models are used for real-time classification problems, with the advantage being that these models are more amenable to implementation on specially designed power-efficient neuromorphic hardware.

The widespread use of SNN simulations in computational neuroscience research and bio-inspired machine learning has prompted interest in the verification of their results, including those run on parallel computers, such as high performance computing clusters, and GPU hardware configurations. In this analysis, we are interested in the upper and lower bounds of numerical error in SNN simulations. As discussed in chapter 2, this problem is reduced to bounding the numerical error of the equivalent serial implementation, and adding the worst-case rounding error of non-deterministic input current summation. Before analysis can begin, we must first define our SNN model. This will be the focus of the following section.

5.1 Model Definitions

There are many different neuron and synapse models with which one can construct a SNN simulation, each with varying degrees of abstraction. Some of

the more popular models include the Izhikevich neuron model [25] and the Hodgkin-Huxley type Traub-Miles [42] neuron model. In this study, we use the Morris-Lecar neuron model [43], which is a reduced version of a Hodgkin-Huxley conductance based model, and a slightly modified version of the Rall synapse model [44].

The models used in this study are fully continuous. If hybrid systems such as the popular integrate-and-fire neuron or the Izhikevich neuron model were to be used, their discretised spiking dynamics (instantaneous spike detection and voltage reset) can cause simulation trajectories to be partitioned into two or more regions. This would require the capability to split affine ranges into smaller sub-ranges, and the ability to merge these ranges as and when the trajectories converge again. This is non-trivial, since modifying the range can invalidate the correlation information. Because the Arpra library does not currently support this, we require continuous models for our analysis. We begin by defining the Morris-Lecar neuron model.

5.1.1 Morris-Lecar Neuron Model

The neuron model of choice in the following experiments will be the Morris-Lecar model [43], due to its fully-continuous dynamics. This is as opposed to models such as the Izhikevich neuron [25], which exhibit discrete spike threshold dynamics. The Morris-Lecar model is a two-dimensional continuous dynamical system, and is defined as follows:

$$\begin{aligned} \frac{dV}{dt} &= \frac{I - G_{\mathrm{Ca}} M_\infty(V)(V - V_{\mathrm{Ca}}) - G_{\mathrm{K}} N (V - V_{\mathrm{K}}) - G_{\mathrm{L}}(V - V_{\mathrm{L}})}{C} \\ \frac{dN}{dt} &= \frac{N_\infty(V) - N}{\tau_N(V)}, \end{aligned} \tag{5.1}$$

using the following auxiliary functions:

$$\begin{aligned} M_\infty(V) &= \frac{1 + \tanh(\frac{V - V_1}{V_2})}{2} \\ N_\infty(V) &= \frac{1 + \tanh(\frac{V - V_3}{V_4})}{2} \\ \tau_N(V) &= \frac{1}{\phi \cosh(\frac{V - V_3}{2V_4})}. \end{aligned} \tag{5.2}$$

Here, V represents the electrical potential across the neuron's membrane, while N represents the fraction of open rectifying K^+ ion channels at a given time. M_∞ and N_∞ are functions of V, and represent the steady state value for M and N, respectively, where M is the fraction of open depolarising Ca^{2+} ion channels. $G_{\mathrm{Ca}}, G_{\mathrm{K}}$ and G_{L} are conductance values for calcium, potassium and

leak channels, respectively, while $V_{\mathrm{Ca}}, V_{\mathrm{K}}$ and V_{L} are their respective reversal potentials. I represents current inputs from external sources, C is the cell membrane capacitance, ϕ is the rate of the recovery process, and the $V_1, ..., V_4$ parameters determine the shape of the steady state activation curves for M and N, and the N time scale.

The model is a reduction of a three-dimensional system describing a neuron with Ca^{2+} spikes, in which dynamics for M are also integrated. Morris and Lecar [43] assume that M reaches its steady-state of $M_\infty(V)$ very fast, and thus can be modelled as instantaneous in equation (5.1). The Morris-Lecar model is a reasonable compromise between realism and computational complexity, making it a practical choice for a first analysis. In addition to the continuous neuron model, we also need a continuously differentiable synapse model to transmit neuronal spikes to other model neurons. This will be discussed in the following section.

5.1.2 Modified Rall Synapse Model

The synapses of the SNN model are simulated using a model similar to the standard Rall synapse [44], but with the additional constraint of being fully continuous. A complete synapse model, constructed in the typical Rall style, is a two-dimensional system that looks similar to the following:

$$\begin{aligned} \frac{dR}{dt} &= \alpha\theta(V_{\mathrm{pre}} - V_{\mathrm{thr}}) - \beta R \\ \frac{dS}{dt} &= \gamma R - \delta S, \end{aligned} \tag{5.3}$$

with θ being the Heaviside step function, defined as follows.

$$\theta(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases} \tag{5.4}$$

In the above definitions, R represents the amount of neurotransmitter released into the synaptic cleft by the presynaptic neuron, while S represents the amount of activated receptors in the postsynaptic neuron. The constant parameters α and β respectively control the rate of presynaptic neurotransmitter release and dispersal, whereas γ and δ respectively control the rate at which neurotransmitter binds to and unbinds from the postsynaptic receptors. In the following experiments, for simplicity, $\alpha = \gamma$ and $\beta = \delta$. Finally, V_{pre} is the membrane potential of the presynaptic neuron, and V_{thr} is the spiking threshold, above which the presynaptic neuron is considered to be spiking.

In order for the modified Rall synapse models to interface with the Morris-

Lecar neuron models, the receptor activation value S must be converted into input current, which is accumulated in the I variable of equation (5.1). For a single synapse, the incoming postsynaptic current I_{syn} is computed as follows.

$$I_{\text{syn}} = G_{\text{syn}} S(V_{\text{syn}} - V) \tag{5.5}$$

where G_{syn} is the conductance of the postsynaptic ion channels, and V_{syn} is the reversal potential for synaptic current.

Note that the model is discontinuous, thanks to its use of θ in equation (5.4). To construct a continuous synapse model, the Heaviside step function θ in equation (5.3) is substituted with the sigmoid function σ, defined as follows.

$$\sigma(x) = \frac{1}{1 + e^{-kx}} \tag{5.6}$$

where k is the steepness of the synapse activation slope. This gives us a fully continuous synapse model, suitable for analysis with Arpra. With the elementary neuron and synapse components of our model defined, we now define the input spike model which will feed randomised input current into the network.

5.1.3 Poisson Input Spike Model

In order to create biologically plausible SNN models, randomised input spikes are generated by dummy neuron models, henceforth Poisson neurons, which are then propagated to the Morris-Lecar neuron models via the modified Rall synapses. The goal is to create randomised input current for the SNN, but in a way that imitates the natural input from synapses originating from upstream neurons outside of the model.

Each Poisson neuron is modelled using a Poisson point process, and is defined as follows.

$$P(N(h) = n) = \frac{(0.001\lambda h)^n}{n!} e^{-0.001\lambda h} \tag{5.7}$$

where $P(N(h) = n)$ is the probability of n spike events occurring within the next h milliseconds, and λ is the desired spike rate in Hertz. To simplify the model, we chose $h = \Delta t$, i.e. equal to the integration time step of the simulation, and the probability of $n > 0$ spikes occurring is interpreted as the probability of a single spike occurring within the next time step. Because neurons cannot produce more than one spike within a small time step, this is a realistic approximation. Since we only care about whether or not a spike occurs, equation (5.7) can be simplified to the following.

$$P(N(h) = 0) = e^{-0.001\lambda h} \tag{5.8}$$

Let r be a uniform random variable in $[0, 1]$, drawn at the start of each integration step for each Poisson neuron. Using equation (5.8), a Poisson neuron's membrane potential V is assumed to take one of two values for the next h milliseconds, depending on whether or not $r > e^{-0.001\lambda h}$. If it is, V is set high to 20 millivolts, otherwise it is set low to -60 millivolts.

We now have the neuron, synapse and input models to construct a complete SNN model. Earlier, we mentioned that performing range analysis on a parallel SNN simulation is the same as analysing the equivalent serial model and adding the maximal error bounds of parallel summation. These summation error bounds are discussed in the following section.

5.1.4 Input Current Summation

As discussed in the introduction, there are a few methods one can use to minimise the rounding error of floating-point summation. One such method is pairwise summation. In this method, all n summands are added in pairs, then the resulting $n/2$ sums are added in pairs, and so on ad infinitum, until only one value remains. This results in a sum with relative error on the order of $\log_2(n)$. Pairwise summation is also a good choice because the procedure is highly amenable to parallel computation.

Despite this, many SNN simulation programs, including GeNN [7], use the recursive summation procedure, in which summands are added one at a time into a single cumulative sum. This variant of summation produces a sum with relative error on the order of n. In the introduction, another method for reducing summation error is discussed, this time using the recursive summation procedure. In this method, summands are reordered before recursive summation is performed, with the intention of reducing the rounding error of each intermediate addition result. However, since the order of input current summation in a parallel SNN simulation is dependent on the order which presynaptic neuron threads finish in, this method is not practical in this instance.

As a consequence, we are forced to assume that summand ordering is arbitrary, and the rounding error bound we use must reflect this. Higham [11] [12] provides a Wilkinson-type error bound E_n for the arbitrary order recursive summation of n floating-point numbers, denoted x, listed below.

$$|\tilde{S} - S| \leq \gamma_{n-1} \sum_{i=1}^{n} |x_i| = (n-1)\mathbf{u} \sum_{i=1}^{n} |x_i| + \mathcal{O}(\mathbf{u}^2) \tag{5.9}$$

where $\gamma_k = k\mathbf{u}/(1 - k\mathbf{u})$ for some constant k, and $\mathbf{u}$ is the unit roundoff.

However this bound is not the strongest we can get, and becomes very weak as $n\mathbf{u}$ approaches 1. Thanks to Rump [45], the quadratic $\mathcal{O}(\mathbf{u})$ term in equation

(5.9) can safely be removed, resulting in an improved error bound for recursive summation.

$$|\tilde{S} - S| \leq (n-1)\mathbf{u}\sum_{i=1}^{n}|x_i| \tag{5.10}$$

This bound is simpler, tighter than the bound in equation (5.9), and is not restricted by n. With the improved error bound, we construct a function which sums n affine ranges and augments the resulting range with the extra recursive summation error.

This recursive summation function works by summing the centre values of the summand ranges using MPFR's correctly rounded `mpfr_sum` function. Next, each deviation coefficient in the result range is computed by summing all deviation coefficients in the operand ranges with the matching noise symbol, again using the `mpfr_sum` function. It then computes the recursive error bound in the right hand side of equation (5.10) by multiplying $(n-1)\mathbf{u}$ with the sum $\sum_{i=1}^{n}|\hat{x}_i|$ of absolute valued operand ranges. The absolute value of an affine range $|\hat{x}_i|$ is taken to be the boundary of $\hat{x}_c \pm \hat{x}_r$ with the highest magnitude. The result is accumulated with other rounding errors in the new deviation term.

With our complete model defined, and our error bounding method for parallel input current summation explained, we can now begin our analysis with a simple fan-in SNN model.

5.2 Results

In these experiments, our first model of choice is a simple fan-in network, in which multiple Poisson neuron inputs project to a single Morris-Lecar neuron via modified Rall synapses. This model is similar to the model in the first experiment of chapter 2, but uses a less contrived input model, and has fully continuous dynamics.

The parameters of the models in these experiments are defined as follows. For the Morris-Lecar model, the parameters are set such that the neurons exhibit class 1 excitability [43], meaning the frequency of generated spikes can be arbitrarily low. The conductance values are $G_{\mathrm{L}} = 2$, $G_{\mathrm{Ca}} = 4$ and $G_{\mathrm{K}} = 8$. The reversal potentials are $V_{\mathrm{L}} = -60$, $V_{\mathrm{Ca}} = 120$ and $V_{\mathrm{K}} = -80$. The remaining parameters are $V_1 = -1.2$, $V_2 = 18$, $V_3 = 12$, $V_4 = 17.4$, $\phi = 1/15$ and $C = 20$.

As for the synapse parameters, the conductance of each synapse is drawn from a normal distribution with standard deviation 1 and mean $10/n_{\mathrm{pre}}$, where n_{pre} is the number of presynaptic neurons. For excitatory synapses, $V_{\mathrm{syn}} = 0$, $V_{\mathrm{thr}} = -50$, $\alpha = 0.25$, $\beta = 0.15$ and $k = 10^6$. For inhibitory synapses, $V_{\mathrm{syn}} = -80$, $V_{\mathrm{thr}} = -50$, $\alpha = 0.075$, $\beta = 0.035$ and $k = 10^6$. Excitatory and inhibitory synapse conductances G_{syn} are normally distributed, with standard deviation

1 and mean $150/n_{\text{pre}}$, where n_{pre} is the number of neurons in the presynaptic neuron group. All remaining parameters are determined on a per-experiment basis.

We begin by testing how the Arpra library's range analysis methods fare on this model, compared to the IA method.

5.2.1 Comparison of Arpra and IA

In the first experiment, the AA, mixed IA/AA and mixed trimmed IA/AA methods implemented by Arpra are compared to the IA method implemented by the MPFI library [35]. 50 Poisson input neurons are used to stimulate the model, each with a firing rate of $\lambda = 20$ Hertz. The modified Rall synapses are all excitatory, and the random number generator seed, used to initialise synaptic conductances, is fixed for all methods. The working precision of Arpra and MPFI ranges is 53 bits, equivalent to IEEE-754 double-precision, while Arpra's internal precision is set to 64 bits. Arpra transcendental functions use the Chebyshev approximation scheme. For Arpra AA methods, independent deviation terms are merged each iteration with `arpra_reduce_last_n`, and terms smaller than 0.3 times the radius are merged after 50 iteration epochs with `arpra_reduce_small_rel`. The model is simulated for 500 milliseconds in steps of $h = 0.5$ milliseconds, using forward Euler integration, and the results for each method are plotted in figure 5.1.

Notice how IA ranges explode almost immediately, whereas Arpra ranges maintain reasonable width for around 300 simulated milliseconds. The radii difference of ranges computed by the three Arpra methods is noticeable, but slight. Whilst these results are not what one might have hoped for, one might argue that this is to be expected. The experiments of chapter 4 showed us that, if a system is sufficiently unstable, even AA ranges eventually explode. But how quickly do the ranges of each method grow in quieter networks?

One might expect ranges of all methods to grow slower in systems with less instability, just as the Hénon map ranges did in chapter 4. In order to investigate this, the same model is simulated for each range analysis method with the 50 Poisson input neurons' firing rate $\lambda = 10$ Hertz. The rest of the model is identical to before. The results are plotted in figure 5.2.

As expected, figure 5.2 illustrates that the ranges computed by all methods do indeed grow slower in quieter networks. Furthermore, the difference between the radii of ranges computed by the three AA methods is more apparent, with mixed trimmed IA/AA ranges maintaining reasonable range for about 60 iterations longer than the plain AA ranges do. However, the ranges computed by Arpra will still explode eventually with this particular model. The ques-

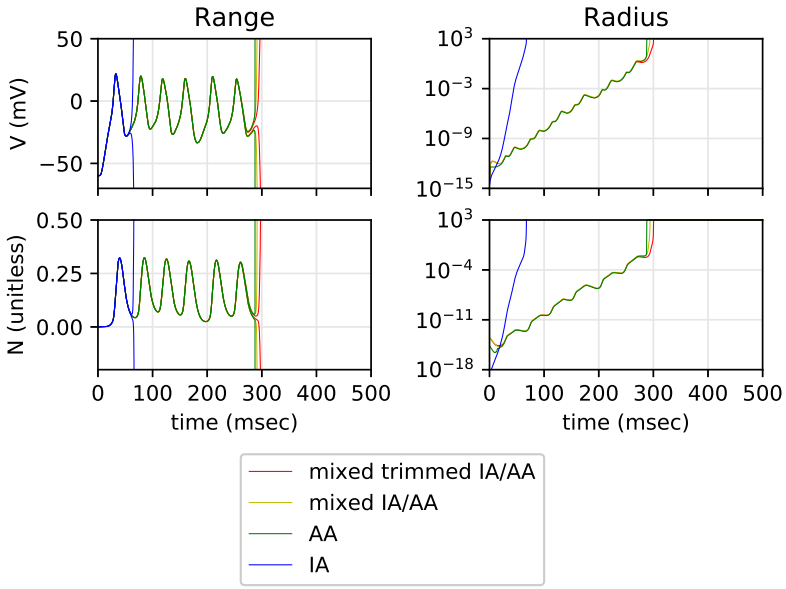

Figure 5.1: Fan in SNN model with high spiking activity. 50 Poisson neurons project to the output Morris-Lecar neuron, each with a firing rate of $\lambda = 20$ Hertz. Arpra methods AA (green), mixed IA/AA (yellow) and mixed Trimmed IA/AA (red) are compared to the MPFI [35] implementation of IA (blue).

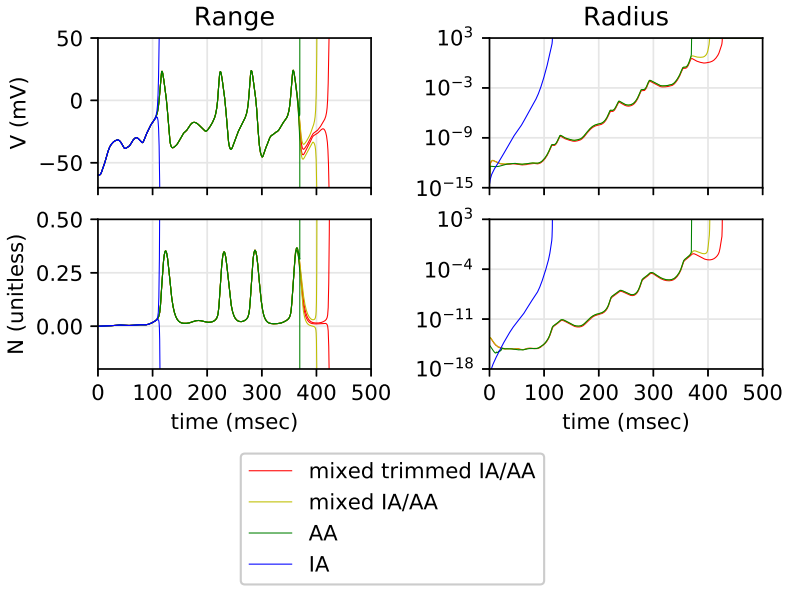

Figure 5.2: Fan in SNN model with moderate spiking activity. Each of the 50 Poisson input neurons has a firing rate of $\lambda = 10$ Hertz. Arpra methods AA (green), mixed IA/AA (yellow) and mixed Trimmed IA/AA (red) are compared to the MPFI [35] implementation of IA (blue).

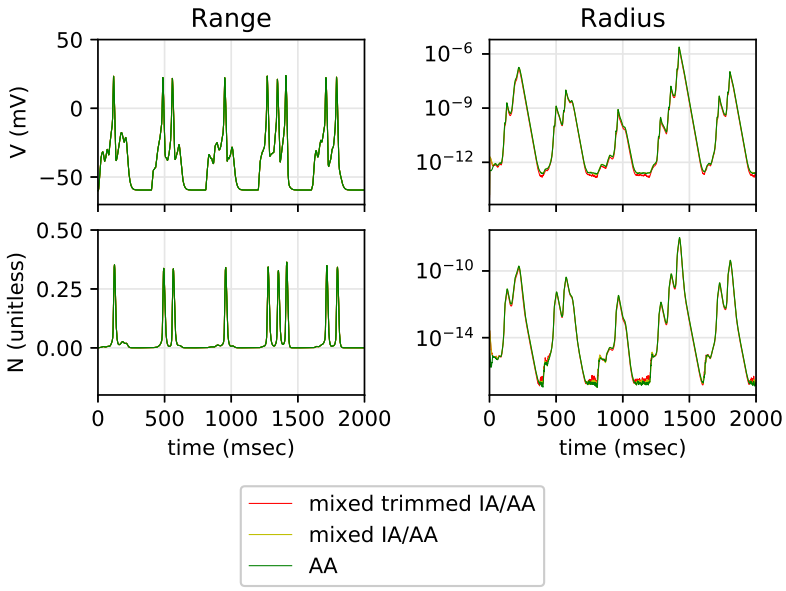

Figure 5.3: Fan in SNN model with repeating 200 milliseconds of input burst and 200 milliseconds of rest. During the bursting input regime, the firing rate of all 50 Poisson input neurons is $\lambda = 10$ Hertz. During the quiescent regime, $\lambda = 0$ Hertz. The radii of ranges computed by AA, mixed IA/AA and mixed trimmed IA/AA all shrink back to a baseline level in the absence of stimulus.

tion then becomes whether or not Arpra ranges can recover in a stable system regime, after a period of growth in an unstable regime.

To test the recovery of Arpra ranges after growth periods in chaotic regimes, the same model is simulated for 2000 milliseconds, and is stimulated with a repeating pattern of 200 millisecond input spike bursts and 200 millisecond rests. For the spike burst regime, the firing rate of the 50 Poisson input neurons is set to $\lambda = 10$ Hertz, while the firing rate is set to $\lambda = 0$ Hertz for the rest regime. The IA method is omitted, since IA ranges explode regardless of the presence of a recovery period in the simulation. The results are plotted for the three Arpra methods in figure 5.3.

These plots illustrate that, although the radii of Arpra ranges grow rapidly during the spike burst regime, they also shrink equally rapidly in the quiescent regime to a baseline of approximately 10^{-13} for V and 10^{-17} for N. This is consistent with the behaviour of Arpra when iterating the stable Hénon map in chapter 4, where range width begins to shrink as the stable limit cycle is approached. Although this demonstrates that Arpra at least has the ability to recover from moderate range explosion in chaotic regimes, the other results in this section suggest that the scope of all three AA variants discussed here may

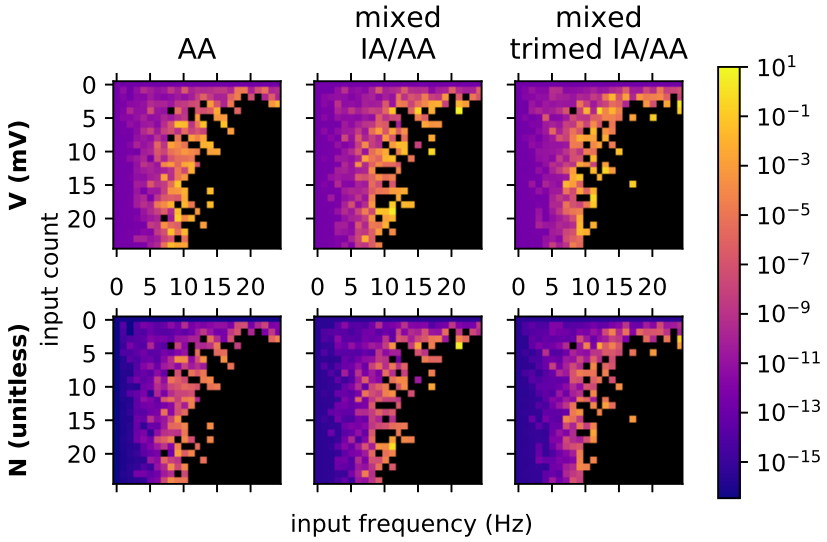

Figure 5.4: Fan-in SNN simulation with number of Poisson input neurons and firing rate varied from 0 to 24 inclusive. Each tile shows the mean radius width of V or N over 500 simulated milliseconds, with warmer tiles indicating wider mean radius. Black tiles indicate that the range explodes at some point during the simulation.

be limited to the analysis of SNN models with relatively low spiking activity.

To demonstrate this, the same fan-in model is simulated a total of 625 times for 500 milliseconds. Both the number of Poisson input neurons and their common firing rate are varied from 0 to 24 inclusive, in steps of 1. Random number seeds for Poisson input generators and synaptic conductance values are *not* fixed in this experiment. The average radius of both V and N is plotted for all three Arpra methods in figure 5.4. Black tiles indicate that a range explodes within the 500 milliseconds of simulated time. Otherwise, warmer tiles indicate a higher average range width.

As these results illustrate, both increasing the number of Poisson input neurons and increasing the spiking frequency of the inputs affects the average width of ranges over the course of the simulation, but in unequal measure. In these simulations, ranges tend to begin exploding when input neuron spiking frequency exceeds around 5 Hertz, while ranges can explode in models with as low as just a single input neuron. This is because changing a shared firing rate for presynaptic neurons has a multiplicative effect on the number of spikes received by the postsynaptic neuron, whereas changing the number of presynaptic neurons only has an additive effect.

It now seems clear that simulated dynamical systems trajectories must have local stability for a sufficiently high proportion of the simulation to be amenable

for analysis using Arpra's three AA methods. So how accurately do these methods bound the trajectories of simulations using real IEEE-754 floating-point arithmetic?

5.2.2 How Tight are Arpra Bounds?

To test how well Arpra bounds the trajectories of floating-point SNN simulations, a chaotic fan-in model is analysed for 500 milliseconds, in steps of $h = 0.5$ milliseconds, using Arpra's mixed trimmed IA/AA method. The firing rate of the $n = 500$ Poisson input neurons is set to $\lambda = 10$ Hertz, all synapses use the excitatory parameters from the Model Definition section, and all random number seeds are fixed. The same model is also simulated 1000 times using IEEE-754 floating-point arithmetic, using the MPFR library [17]. Arpra working precision and MPFR precision is set to 53 while Arpra internal precision is set to 64. In the floating-point simulations, incoming spike lists are randomised, to simulate parallel summation of the input currents. The upper and lower bounds of the floating-point trajectories are compared with the ranges computed by Arpra.

Furthermore, a stability analysis is performed on the Morris-Lecar neuron model to determine whether the growth of Arpra ranges in unstable regimes is reflective of the actual trajectory divergence in the floating-point computations. The tangent space method is used for the stability analysis. In this method, the local Lyapunov exponent is numerically estimated at each time step by evaluating the Jacobian matrix at the average state vector of all 1000 floating-point simulations, diagonalising this matrix, and then taking the largest Eigenvalue as the local Lyapunov exponent. The global Lyapunov exponent of the trajectory is obtained by averaging the local Lyapunov exponent over the complete trajectory. The range analysis, floating-point trajectory bounds, and stability analysis results are potted in figure 5.5.

As these results illustrate, the width of ranges computed with mixed trimmed IA/AA grows consistently towards infinity, whilst the divergence of trajectories computed in floating-point remains relatively constant throughout, diverging slightly when spikes occur and converging back afterwards. While we saw in chapter 2 that the trajectories of floating-point simulations will eventually diverge more visibly in such an unstable model, this did not occur within the time it took for the Arpra ranges to explode in figure 5.5, and certainly not to the same extent.

In chapter 4, we saw that affine ranges inevitably explode when analysing chaotic systems, and this is reflected in these results. The global Lyapunov exponent of the floating-point simulations' average trajectory is 0.027, to three decimal places, indicating that this trajectory is indeed unstable. We see that,

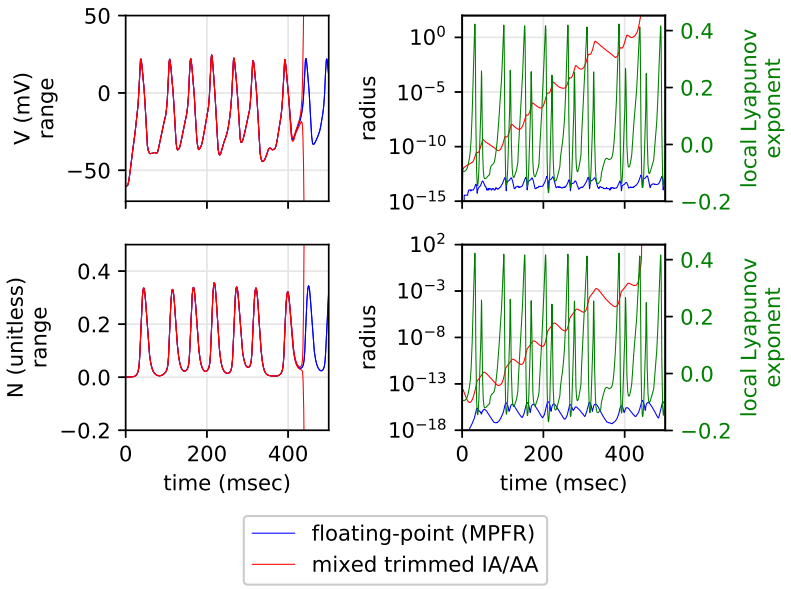

Figure 5.5: Comparison of mixed trimmed IA/AA ranges and the trajectory boundaries computed from 1000 floating-point simulations of a fan-in SNN model, consisting of $n = 500$ Poisson input neurons, with $\lambda = 10$ Hertz spike frequency, and excitatory synapses projecting to a single Morris-Lecar neuron. The local Lyapunov exponent is plotted in the right hand column (in green) at each time step.

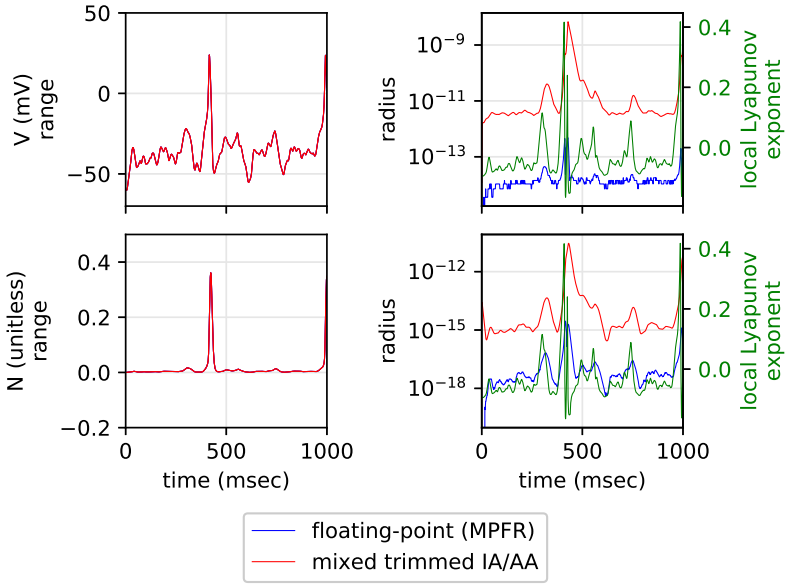

Figure 5.6: Comparison of mixed trimmed IA/AA ranges and the trajectory boundaries computed from 1000 floating-point simulations of a fan-in model with $n = 500$ Poisson input neurons, $\lambda = 5$ Hertz spiking frequency, and excitatory synapses. The local Lyapunov exponent is plotted in the right hand column (in green) at each time step.

although the Arpra ranges have brief recovery periods when the local Lyapunov exponent falls below zero due to the absence of spiking dynamics, the ranges resume growing when the local Lyapunov resurfaces above zero.

Since we already know from chapter 4 that Arpra ranges explode in chaotic systems, what is perhaps of more interest is how tightly they bound floating-point trajectories in stable systems. We will now perform the same analysis on a similar model, where the spike frequency of the $n = 500$ Poisson input neurons is $\lambda = 5$ Hertz. The model is simulated for 1000 simulated milliseconds, in steps of $h = 0.5$ milliseconds. As before, MPFR floating-point is used to simulate the model 1000 times with random input current summation order, Arpra mixed trimmed IA/AA method is used for range analysis, and the tangent space method is used to analyse system stability. The results of this quieter model are plotted in figure 5.6.

Here we see again that the radii of Arpra ranges is allowed to recover fully down to a baseline value in the absence of spiking dynamics. However, the baseline radius of both V and N is still approximately three orders of magnitude higher than the radius of trajectory divergence in the floating-point simulations. A global Lyapunov exponent of -0.033, to three decimal places, confirms that

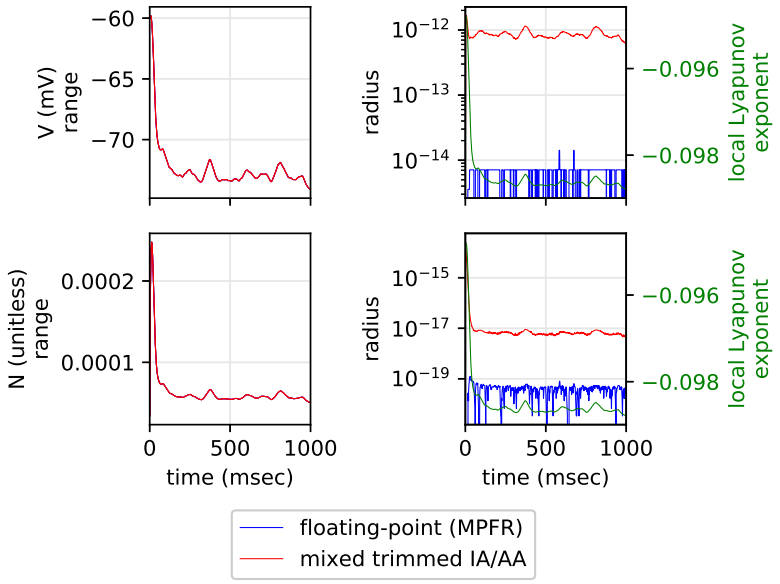

Figure 5.7: Comparison of mixed trimmed IA/AA ranges and the trajectory boundaries computed from 1000 floating-point simulations of an inhibitory fan-in model with $n = 500$ Poisson input neurons, $\lambda = 10$ Hertz spiking frequency, and inhibitory synapses. The local Lyapunov exponent is plotted (in green) at each time step.

this model is stable, and local Lyapunov exponents remain below zero after the Arpra range value bottoms out. Even still, the ranges are conservative.

To test whether near-threshold dynamics are to blame for the conservative bounds, the same analysis is performed on yet another similar model, but this time with inhibitory synapses, whose parameters are given in the Model Definitions section. As before, the model is advanced 1000 milliseconds, in steps of $h = 0.5$. The input neuron firing frequency is set to $\lambda = 10$. By hyperpolarising the output Morris-Lecar neuron in this way, one reduces the probability that an Arpra range straddles the neuron's firing threshold. The results of this analysis are plotted in figure 5.7.

The global Lyapunov exponent of the average floating-point model trajectory is -0.099, to three decimal places, indicating the model is not chaotic. As the plots illustrate, although the Arpra ranges are thinner than those in 5.6, they are still approximately two orders of magnitude wider than the divergence seen empirically. This rules out near-threshold dynamics as the sole culprit of range overestimation. So what else could be the cause? The fact that the AA methods perform worst-case error bounding means that Arpra ranges will always be slightly larger than the MPFR mean $\pm$ standard deviation ranges.

Furthermore, the radius of an Arpra range, which is used when computing the true range, is not rigorous. Since it is computed by summing the absolute value of each deviation term, a small amount of extra rounding error is introduced. Perhaps an even greater source of extra range width, however, comes from function linearisation.

5.2.3 Nonlinear Dynamics Approximation

The difference between Arpra bounds and the trajectory bounds observed in floating-point experiments could still be caused by a number of factors not yet tested, such as rounding error due to low internal precision, or error associated with deviation term condensing (discussed in the next section). First and foremost, it is important to remind ourselves that range analysis, by its very definition, is a method for computing the theoretical worst-case error bounds of a computation, and not necessarily the bounds that one may observe in practice. Range analysis is conservative by design. Having said that, there are different flavours of range analysis. We have already seen how much of an improvement AA is over IA, but AA is a first-order range analysis method, and thus incurs heavy approximation error whenever nonlinear functions are used. One solution, and potential future work in Arpra, is to implement Taylor intervals, in which ranges are represented using Taylor series polynomials. For now, however, we focus on the linear mixed trimmed IA/AA method.

The Morris-Lecar neuron model in equation (5.1) uses the nonlinear functions tanh and cosh, implemented in terms of `arpra_exp`, and division, implemented using `arpra_inv`, which are both susceptible to overshoot and undershoot, as discussed in chapter 1. Because of this, one would expect there to be a noticeable difference in the radius of Arpra ranges when different approximation schemes are used. To determine how the error from nonlinear function approximation affects Arpra ranges in SNN models, a fan-in network with 50 Poisson input neurons, with $\lambda = 10$ Hertz firing rates, is simulated three times. In the first run, `arpra_exp` and `arpra_inv` use the same Chebyshev approximation scheme used up until now. In the second run, these functions use the Min-Range approximation scheme. In the final run, functions use the Min-Range scheme, but the approximation error term δ is set to zero, giving us a crude demonstration of the effect linearisation error has on computed ranges. The random number seeds for synapse conductance and Poisson generators is fixed, and synapses are excitatory. The results are plotted in figure 5.8.

As the plots illustrate, the Min-Range function approximation scheme performs only marginally better than the Chebyshev scheme, with ranges lasting approximately ten simulated milliseconds longer before exploding. Whilst this

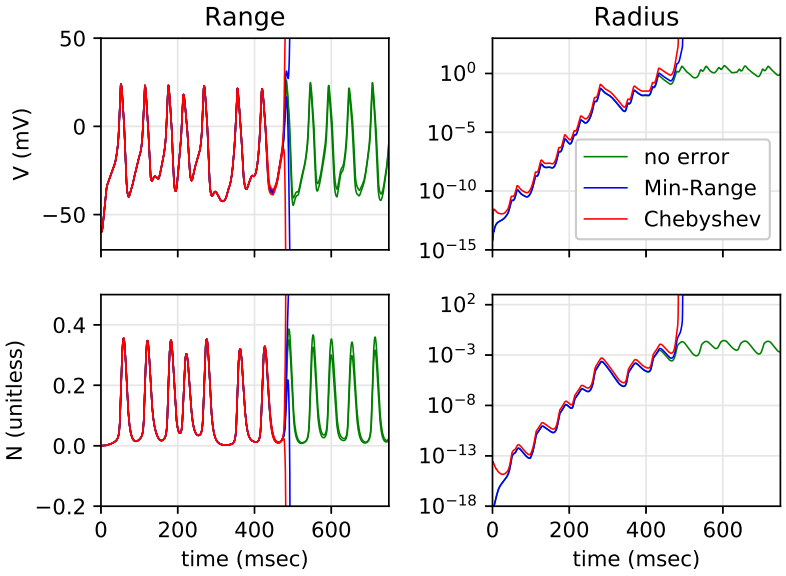

Figure 5.8: Difference in Arpra ranges computed with transcendental functions using Chebyshev (red), Min-Range (blue) and Min-Range with zero approximation error (green). The model is a fan-in SNN with $n = 50$ Poisson inputs firing at $\lambda = 10$ Hertz, connected to a Morris-Lecar neuron with excitatory synapses. Note how the ranges do not explode when approximation error is removed artificially.

may be surprising, this is not an unreasonable outcome. Although the Min-Range approximation does not suffer from the overshoot and undershoot phenomenon of the Chebyshev approximation, the additional range width resulting from loss of correlation information can negate this benefit in certain problems.

What is more interesting, though, is the behaviour of Arpra ranges when no approximation error is added to the new deviation term after a transcendental function. Although the ranges initially grow to have a radius of about 10^{-2}, they subsequently cease growing and remain at this width for the remainder of the simulation. As crude as the zero-error measurement is, it does seem to suggest that one of the biggest cause of range growth in unstable systems is that the range representation is of too low order. One can imagine that a second order range analysis method would better approximate the exp function using a quadratic curve, lowering the error term δ, and successively higher order methods would further reduce δ. Arguably, nonlinear differential equations allow us to model more interesting dynamical systems. While it is important to recognise that the first-order AA range analysis methods may be an ideal choice for first-order dynamical systems and linear algebra computations, we may conclude here that AA is not a suitable choice for nonlinear dynamical systems such as SNN models.

However, there are a few more features of Arpra that we can try, in order to delay range explosion for as long as possible. We test the internal precision and term condensing features of Arpra on SNN simulations in the following section.

5.2.4 Internal Precision and Term Condensing

As discussed in chapter 3, Arpra implements a few more tricks for reducing the overhead rounding error of the AA methods, and minimising the computational resources required for the analysis. The internal precision feature allows one to control how precisely the range is represented internally, and how precisely the intermediate values are computed, while the `arpra_reduce` functions allow one to selectively condense sets of deviation terms, saving memory and boosting the speed of the analysis.

We know from Rump's example [18] [19] that the mapping between floating-point precision and accuracy of the result is not continuous, so it is difficult to determine how high the precision can go before the result stops changing, if at all. However, we found in chapter 4 that moderately increasing the internal precision offered a small improvement in range tightness, but further increases to internal precision offered diminishing returns. We also know, due to the literature on the Table Maker's Dilemma [31] [33], that there exists some precision p for intermediate calculations, such that the final rounding of an Arpra range

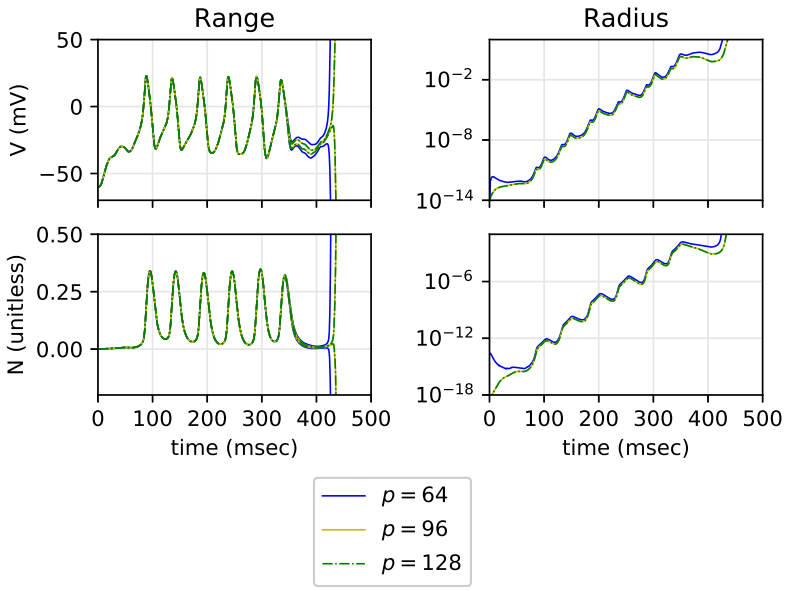

Figure 5.9: Difference in Arpra ranges computed with internal precision $p = 64$ (solid blue), $p = 96$ (solid yellow) and $p = 128$ (dashed green). The model is a fan-in SNN model with $n = 50$ Poisson inputs firing at $\lambda = 10$ Hertz, connected to a Morris-Lecar neuron with excitatory synapses. Note how the improvement in range tightness has plateaued at $p = 96$, and coincides with ranges computed with $p = 128$.

to the working precision is correct when the internal precision is at least p.

In figure 5.9, a fan-in SNN of $n = 50$ Poisson input neurons firing at $\lambda = 10$ Hertz is simulated with Arpra internal precision set to $p = 64$, $p = 96$ and $p = 128$ bits. Synapses are excitatory, and random number seeds are fixed. Just as we saw for the Hénon map in figure 4.3, a small improvement of range tightness is observed as p is raised to 96, but further increases of p have negligible effect. Besides Ziv's strategy [33], there is no known way to determine the minimum p for which the result stops changing as p increases. However, in the experiments of this study, $p = 96$ has been sufficient to prevent the majority of overhead AA rounding error when analysing IEEE-754 double-precision computations. So what is the optimal deviation term condensing strategy?

All Arpra analyses of SNN simulations in this chapter have made heavy use of Arpra's deviation term condensing routines. Without them, the analysis would become intractable. For a single Arpra range, assuming we are integrating for m time steps and that the number of deviation terms grows by some constant k each step, we need to compute up to $k + 2k + ... + (m-1)k + mk$ terms throughout the simulation. Ignoring constants, this gives us an asymptotic

runtime complexity of $\mathcal{O}(km^2)$, which is not ideal in longer simulations.

$$k\sum_{i=1}^{m} i = k\frac{m(m+1)}{2} = \mathcal{O}(km^2) \tag{5.11}$$

The SNN simulations call `arpra_reduce_last_n` on V and N after each integration step, which condenses all k new (independent) deviation terms into one, thus reducing equation (5.11) to $\mathcal{O}(m^2)$. The SNN simulations also call `arpra_reduce_small_rel` every 50 iteration epoch, using a relative threshold of $t_{\text{rel}} = 0.3$. This function condenses all deviation terms smaller than $t_{\text{rel}}r$, with r being the range's radius. No more than $1/t_{\text{rel}}$ (rounded down) terms can remain after this call, since the absolute sum of the remaining high-magnitude terms cannot exceed r. This effectively resets the number of active noise symbols in a range to some threshold-dependent baseline each time it is used.

Given that the number of remaining deviation terms is bounded after the application of `arpra_reduce_small_rel`, one might also consider a term condensing function in which the number of deviation terms is reduced to some given constant. Such a function would reduce terms in order from smallest to largest, until the term number constraint is satisfied. Implementing this method is left for future work, but it would likely have similar effectiveness to that of the relative small term reduction method.

Figure 5.10 illustrates how range quality varies with t_{rel} set to 0.1, 0.2 and 0.3. Besides t_{rel}, The SNN model is identical to that from the last experiment. Arpra's internal precision is set to $p = 64$. Surprisingly, although range fit is generally looser when any of the term condensing strategies are used than when none are used, ranges are generally tighter when a higher relative threshold is used. This is in contrast with the Hénon map example, in figure 4.7, where a lower relative threshold produced tighter ranges. This could be because the majority of important correlation information is lost when using all three relative thresholds, and the overhead rounding error of having slightly more deviation terms outweighs the error from lost correlation information due to the merging of more terms, but this requires further investigation.

This concludes our analysis of SNN models with Arpra. In this chapter, we used Arpra to analyse more realistic SNN models, each consisting of Morris-Lecar neurons, our continuous modified Rall synapses and our simplified Poisson input neurons. We explain how the maximum error bounds of parallel recursive summation can be computed, based on the work of Higham [11] and Rump [45]. We compared the IA method with Arpra's AA, mixed IA/AA and mixed trimmed IA/AA methods when analysing SNN models, and found that the Arpra maintained tighter ranges for longer than IA did in unstable dynamics.

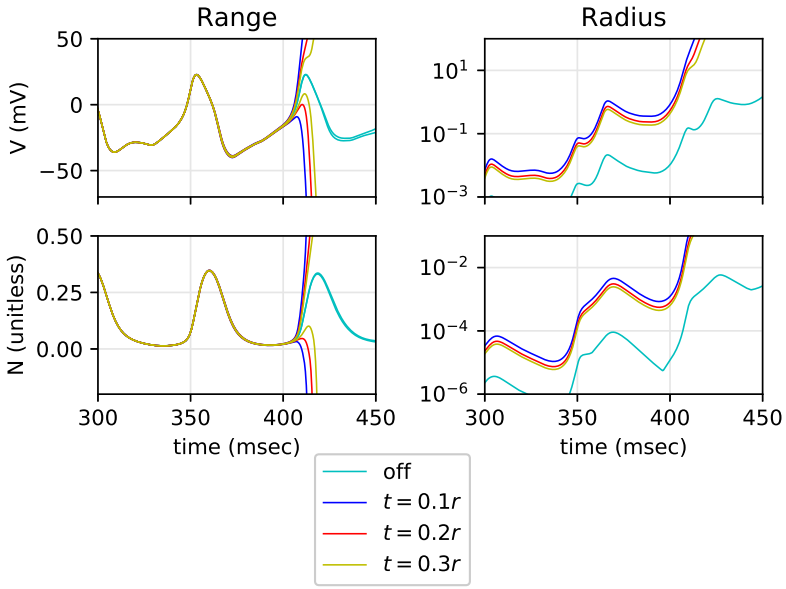

Figure 5.10: The difference between the radius of ranges condensed with `arpra_reduce_small_rel` after 50 iteration epochs, and those computed without term condensing (cyan). The thresholds t used above are 0.1 (blue), 0.2 (red) and 0.3 (green) multiplied by the radius r. The plot shows the ranges between 300 and 450 simulated milliseconds.

We found that mixed trimmed IA/AA performed marginally better than mixed IA/AA, which in turn performed marginally better than plain AA, and ranges from all three Arpra methods converged back to a tight baseline width whenever the simulation dynamics become sufficiently stable, unlike IA ranges. We found that Arpra ranges are predictably quite conservative estimates of actual floating-point computations, and demonstrate how much of the added range width is linearisation error, due the presence of non-linear functions in the SNN model. Finally, we test the internal precision and deviation term condensing features of Arpra, and find comparable results to those from chapter 4. In the next chapter, we summarise our findings, state our conclusions and discuss the implications of this study.

Chapter 6

Discussion

The goal of this project was to compute boundaries for the numerical error of spiking neural networks (SNN) on massively parallel hardware. In the pursuit of this goal, the Arpra library for arbitrary-precision range analysis was developed, which, unlike other AA packages, builds on the standard AA method by exploiting extended internal floating-point precision to produce tighter bounding ranges. It also features the novel mixed trimmed IA/AA method, improved affine multiplication and three novel term reduction functions. This library is tested against an alternative AA implementation, from the INTLAB [28] package, using the Hénon map as a benchmark. After having confirmed the correctness and accuracy of the Arpra library, we proceeded to analyse a realistic fan-in SNN model involving the Morris-Lecar neuron model [43], with Poisson process spike generators and a slightly modified Rall synapse model [44].

In this chapter, we begin by summarising the results of this study, and drawing our conclusions. We then discuss some alternative methods, which perhaps may have been more suitable for this study in retrospect. Finally, we end by discussing some different problem domains which Arpra's mixed trimmed IA/AA method might be more suited to.

6.1 Summary and Conclusions

In chapter 2, we demonstrated the issue of results reproducibility for SNN simulations on parallel hardware using the popular Izhikevich neuron model [25]. In the first experiment, we saw how randomising the summation order of input currents is sufficient to cause noticeable divergence in identical simulations, even when computed on the same software and hardware environment. Despite the simplicity of this experiment, we can clearly see how the phase of spiking is affected, as well as the shape of the spikes themselves. Although flooding

a single neuron to saturation with input current may be a little contrived, it is suggestive that similar effects are likely to appear in the dynamics of more complex SNN models.

In order to see a more realistic example, we simulated a larger pulse-coupled neural network (PCNN), with comparatively less stimulating current per neuron. We again found noticeable divergence in the simulated trajectories, which completely disappeared when the order of input current summation was fixed. Although this divergence was not catastrophic in this particular experiment, the example shows that the risk of catastrophic trajectory divergence is ever present in parallel SNN simulations. The few times that it does occur may very well be the times that cause the most harm.

Accordingly, it is the authors opinion that all researchers who do computational studies should be made aware of this phenomenon, no matter what their background of expertise is. Ideally, researchers should also be provided with a means of computing, or at the very least estimating, the boundaries of the trajectory divergence possible in their computations. This is particularly important today and for the foreseeable future, with the advent of high performance massively parallel compute architectures. The adoption of this computing paradigm is speeding up, but education on the differences to more familiar sequential compute architectures lags behind. Information on the existence and nature of numerical errors is perceived as confusing and inaccessible at best, or boring and inconsequential at worst. This attitude will only change if the tools for analysing the problem - i.e. range analysis - are simple to use, computationally tractable, and compute bounds tight enough to be useful.

In chapter 3, I presented Arpra [26]; my arbitrary-precision range analysis library that aims to meet these requirements. We examined the MPFR library [17] for correctly rounded arbitrary-precision floating-point arithmetic, which is the foundation on which Arpra was built. Towards the ease of use requirement, we described how Arpra adopts a similar function schema to MPFR, in which all functions `function(z, x, y)` are written very similarly to how one would write the corresponding equation $z = x$ op y. Towards the computational tractability requirement, we discussed the core AA method, and discuss several modifications which aim to improve the tightness of computed ranges while minimising the computational complexity. These include the extended internal precision mechanism, a method for computing the exact overhead rounding error in terms of the unit in last place (ULP), the improved multiplication routine mentioned in [27], the mixed IA/AA extension with error term trimming, and finally the three deviation term condensing routines. We tested the main arithmetic functions implemented in Arpra, and found that they compute ranges at least as tight as ranges computed with IA, and potentially far tighter when

operands are correlated.

In chapter 4, we put Arpra through its paces by analysing the Hénon map, as an example of a dynamical system which is known to have regular and chaotic regimes. The Hénon map is a simple two-dimensional discrete dynamical system with no transcendental operations, which can be tuned to exhibit dynamics ranging from chaotic to stable limit cycles, depending on the chosen value of the parameters α and β. Using a stable Hénon map with $\alpha = 1.057$ and $\beta = 0.3$, we compared Arpra to another range analysis package known as INTLAB [28], which implements the IA and mixed IA/AA methods. Here we first saw that the width of Arpra AA ranges eventually collapses towards zero when the system is not chaotic, whereas INTLAB IA ranges will continue to grow regardless due to the dependency problem and the resulting wrapping effect.

We saw that Arpra's plain AA method seemed to perform marginally worse than the mixed IA/AA method of INTLAB, even though the Hénon map does not involve transcendental functions, for which the mixed IA/AA method was specifically designed. This is likely because the tighter INTLAB ranges used in the plot were intersections of AA and IA ranges, whereas the plotted Arpra ranges computed with plain AA had to be derived from the centre and radius, and were thus looser. This suggested that the mixed IA/AA method is still useful even in algebraic computations, since although the tighter IA/AA range intersections are not used in the computation itself, they can still be used for plotting purposes, for example. We found that the performance of the mixed IA/AA methods in Arpra and INTLAB are qualitatively the same, with Arpra very slightly superior likely due to its use of the improved multiplication routine and rounding errors in terms of ULP. Finally, we saw that Arpra ranges computed with the mixed trimmed IA/AA method were modestly slimmer by around 10^{-5} units than those computed with mixed IA/AA by INTLAB just before the ranges converged on the stable limit cycle.

Arpra's mixed trimmed IA/AA method was also tested on the Hénon map with higher α values, and we found that Arpra ranges will eventually explode like the IA ranges did when the system becomes sufficiently unstable. Since trajectories can diverge rapidly in chaotic systems, the ranges representing those conditions can also grow rapidly. Besides the fact that wide ranges are less useful as error bounds, another problem is that an invalid operation such as division by zero is more likely to occur with wider ranges. Even if the range is still tight enough to be useful, if any part of the range becomes invalid, then the entire range becomes invalid.

We also tested the internal precision feature of Arpra and found that increasing it only yielded a modest improvement in range tightness, and further precision increases gave diminished returns. In theory, one could compute the

optimum internal precision algorithmically, using Ziv's strategy [33], for instance. However, given that the number of deviation terms in Arpra ranges is in constant flux, this procedure would need to be performed quite regularly.

Finally, Arpra's deviation term reduction strategies were tested. A small overhead error cost was demonstrated when using the `arpra_reduce_last_n` routine, even though no correlation information was lost. However, the benefits were an approximate 85% reduction of deviation terms and a large decrease in both the runtime and memory usage. We saw even more aggressive reduction of deviation terms using the lossy `arpra_reduce_small_rel` routine, at the cost of correlation information, with a 99.9% reduction of deviation terms using the relative threshold 0.1. However, since so much correlation information is lost, it was found that this routine should be used sparingly, and in combination with other term reduction strategies. In these experiments, increasing the relative threshold used by `arpra_reduce_small` resulted in looser ranges. However, the converse was true in the SNN simulations of chapter 5. Perhaps this was because certain condensed deviation terms, no matter how small, may have been more important than the larger deviation terms at that point in the computation. While this requires more investigation, it would suggest that the best relative threshold to use is highly problem dependent.

In chapter 5, we analysed the numerical error boundaries of a fan-in SNN simulation using the Arpra library for arbitrary-precision range analysis. We specified how parallel SNN simulations can be analysed by proxy, using equivalent serial SNN simulations and adding the worst-case error bounds for arbitrarily ordered input current summation. Since the AA method does not handle discrete range partitioning gracefully, Arpra cannot perform range analysis on computations involving conditional branching without added complexity. Consequently, the SNN model was constructed using the fully continuous Morris-Lecar neuron model, and a slightly modified Rall synapse with continuous neurotransmitter release dynamics. Although the modified neurotransmitter release function from equation (5.6) is continuous, its slope was set very steep in these simulations. As a result, heavy linearisation error can be incurred if the input range straddles the transition from low to high neurotransmitter release. This illustrates a general obstacle for analysing hybrid dynamical systems with AA. One either has to approximate discrete functions with continuous ones, or implement functions for splitting and joining affine ranges while respecting correlation information, where either option has the potential to noticeably increase the overhead error in computed ranges.

For the first few experiments, we compared the AA, mixed IA/AA and mixed trimmed IA/AA methods of Arpra with the plain IA method and found that, whilst Arpra vastly outperforms IA in all cases, AA ranges still tend to

explode when spiking activity is high. We saw successive improvements in range tightness with each AA variant, and saw that these improvements became more pronounced as the spiking activity of the analysed model increases. We also found that AA range width can fully recover after periods of growth due to rapid spiking, providing the network becomes sufficiently quiescent afterwards. These results suggest that the AA variants are only suitable for analysing SNN models in which few spikes are expected to occur, but we can make some predictions of which model modifications will decrease range quality the most. For instance, we showed that when populations of neurons with similar firing rates project excitatory synapses to other neurons in a fan-in manner, the quality of ranges is more sensitive to the mean presynaptic firing rate than it is to the number of presynaptic neurons, since the former has a multiplicative effect on the total spike count whereas the latter has only an additive effect.

The next experiments tested how tightly the Arpra mixed trimmed IA/AA ranges bounded the trajectories of actual floating-point SNN simulations. It also examined how the local stability of the state vector affected range growth in Arpra and trajectory divergence in floating-point. We found that the ranges grow rapidly as the local Lyapunov exponent rises above zero, corresponding to the occurrence of neuronal spikes in the simulation, while floating-point trajectories only diverged slightly. We also saw that recovery of Arpra ranges occurred when the local Lyapunov exponent sank below zero, corresponding to the subthreshold neuronal dynamics, but were still approximately two orders of magnitude wider than the interval which bounds the observed floating-point trajectories. Whilst this does not bode well for the AA methods when analysing such models, one can still use Arpra range growth to infer high-risk points of divergence in actual simulations. The floating-point trajectories diverge when the local Lyapunov exponent is high, and this is reflected in the sharp growth of Arpra ranges, which might be useful to the user in itself. However, given the tendency of Arpra ranges to completely explode in unstable systems, the extent of its utility for bounding simulated trajectories in SNN models is in doubt.

Here, we feel compelled to repeat that, while the computed ranges appear loose compared to the interval of trajectory divergence observed in practice, the Arpra library is behaving correctly. Arpra obeys the fundamental theorem of interval arithmetic. It computes the worst-possible-case error boundaries of all computations, no matter how extreme, and never compromises in the interest of range tightness. It is conservative by design. With that said, why is it that the divergence of actual floating-point computations is so low relative to the bounds computed by Arpra?

Rounding errors are certainly not random. However it stands to reason that floating-point rounding errors incurred using IEEE-754 'round to nearest' mode

are a somewhat even mixture of positive and negative. If one were to make the simplifying assumptions that rounding errors are uniformly distributed in some constant interval $[-k, k]$, and these rounding errors are independent, then the central limit theorem states that a floating-point number containing n rounding errors should be normally distributed about a mean somewhere near the centre of the corresponding Arpra range, with a standard deviation of $k\sqrt{n}$. With such a low likelihood of the worst-case occurring, it now seems reasonable to expect the extra padding in Arpra ranges. Artificially constructing a worst-case for parallel input current summation to test Arpra with is surprisingly difficult. According to Higham [12], constructing the best-case summation ordering involves arranging summands such that the magnitude of each successive partial sum is minimised, which is known to be NP-hard. Presumably, to construct the worst-case summation ordering, one would arrange summands such that the magnitude of each partial sum is maximised, which would also be NP-hard. Instead, the worst-case summation ordering was approximated by arranging summands in order of decreasing magnitude, but preliminary tests showed little additional trajectory deviation.

In our experiments, the internal precision of Arpra was varied to determine the improvement of Arpra range tightness in the SNN simulation. As was the case in the Hénon map experiments in chapter 4, we saw that increasing the internal precision resulted in a moderate improvement in range tightness, with further increases having negligible effect. Although the range improvement from each optimisation in Arpra is modest on its own, when taken together they can noticeably improve the tightness, thus usefulness, of Arpra ranges. One may suppose that these improvements are only relevant in the analysis of short and high accuracy computations, such as in the verification of numerical algorithm packages. However, the effects also tend to become visible when the dynamics of the computation are borderline unstable. For instance, in the method experiments of chapter 5, figure 5.2, we see that successive improvements to the core AA method resulted in ranges that remained tight, and thus useful, for successively longer.

Given the fact that, even in unstable systems, the width of ranges computed with Arpra remains at a usable tightness for a reasonable amount of time before exploding, Arpra is still a useful tool for examining the short-term trajectory divergence in chaotic systems. One could even simply reset Arpra ranges to zero width in known stable regimes, or before points of specific interest. Arpra performs very well in most linear computations. It also performs reasonably well in sufficiently stable nonlinear computations, although performance degrades quickly as range width increases. In figure 5.8 of chapter 5, we saw the dramatic effect that removing the linearisation error had on range growth, suggesting that

a sizeable portion of affine range width in SNN simulations is a consequence of heavy linearisation error in transcendental functions. One solution is to use higher order range representations. We discuss alternative methods and suitable problem domains for Arpra in the following section.

6.2 Reflection and Implications

Although the AA method performs well when analysing linear computations, and reasonably well when analysing certain nonlinear computations, its performance begins to decline as the nonlinear dynamics begin to dominate the computation. Since AA can be considered a first-order range analysis method, consisting of linear functions of the centre and deviation terms, the logical progression for range analysis would be to allow higher-order terms in the range representations. For instance, one might approximate the exponential function with quadratic deviation terms, which would incur third-order approximation error instead of second-order. One could even go further, and allow up to nth-order deviation terms, with $n + 1$ order approximation error. These ideas were proposed by Berz et al [46] [47], under the name 'Taylor methods'. They were subsequently used to successfully model near-Earth object trajectories in space, given intervals of initial conditions [48].

Whilst the computational complexity of such an analysis may become a limiting factor for how high n can go in practice, implementing such functionality would unlock the analysis of a far greater space of problems. Given the accuracy of Arpra in linear computations, it seems reasonable to expect comparable accuracy in the analysis of an nth-order computation with an nth-order Taylor polynomial method in Arpra. For these reasons, a high-priority future milestone for the Arpra project will be the implementation and testing of arbitrary-precision Taylor method range analysis techniques, to a user-determined order. One may suggest an alternate method such as trigonometric polynomials might be suitable, but results due to Nedialkov et al [49] seem to suggest that Taylor polynomials are the optimal approximation choice.

Whilst the calculation of worst-case error bounds may be important when, for instance, predicting if a near-Earth asteroid is on a collision course, a more relaxed 'average-case' error bounding may be sufficient in some cases. To achieve this, one would resort to statistical methods. One such method is known as 'discrete stochastic arithmetic' (DSA) [50] [51] [13], implemented in the CADNA library [52]. This method roughly consists of running the computation n times, usually three, using randomised IEEE-754 rounding modes for each floating-point operation. This is claimed to give a reasonable approximation of the error boundaries one actually observes in practice. It should be emphasised that

CADNA and the DSA method attempt to solve a different problem to what Arpra and the range analysis methods intend to solve. The stochastic nature of the DSA method means that it does not obey the fundamental theorem of interval arithmetic. The DSA boundaries are not guaranteed to contain all possible trajectories of a computation, but instead computes something more like the expected range of trajectories. Whilst this kind of analysis has its uses, it cannot be relied upon when rigorous worst-case boundaries are required. For instance, the analysis of flight control software would not be an appropriate problem for DSA, and would instead be handled by more comprehensive static analysis software using range analysis methods, such as the Astrée static analysis package [53].

These comprehensive analysis packages are often proprietary and expensive, which rules out their use in the analysis of open source numerical software. Arpra [26], on the other hand, is open source and freely available under the terms of the GNU lesser general public license version 3.0. Arpra also has the advantage of being built on top of the arbitrary-precision MPFR library, and benefits from arbitrary floating-point precision and correct rounding for all arithmetic functions in all software and hardware environments. So if it is not suitable for the analysis of SNN models and other highly nonlinear computations, then what else might it be used for?

Stolfi and de Figueiredo [23] give plenty of examples where the AA method is useful, such as function root finding and global optimisation. Besides the many examples listed there, Arpra has many more uses, such as for the verification of serial and parallel numerical libraries, both proprietary and otherwise. Open source libraries often lack tight accuracy bounds for functions in their documentation, with a notable example being the GNU standard C library [14] (glibc). Given the improved performance of Arpra in linear computations, due to the extended internal precision and other features, the analysis of software like glibc and many linear algebra packages could be prime use cases for Arpra. Furthermore, although Arpra is not ideal for analysing realistic chaotic or highly nonlinear SNN simulations, it would likely be perfectly suited for analysing complex artificial neural networks, whose dynamics are usually less nonlinear and less unstable. Even before higher-order Taylor methods are implemented in Arpra, there is still quite a large space of problems that Arpra is very well suited to, and further work by the Arpra project will examine these opportunities in greater detail.

As for simulations that are a little more stable, Arpra provides several ODE steppers to assist in the analysis of these models. Besides the forward Euler method, Arpra also provides steppers for the trapezoidal rule, the Bogacki-Shampine 3(2) method [54], the Dormand-Prince 5(4) method [55] and the

Dormand-Prince 8(7) method. For all Arpra steppers, the user may select which precision in which all of the method's constants are pre-computed. As mentioned in chapter 1, the choice of integration step size and method has non-trivial consequences in terms of incurred numerical error. If either a low step value or a high-order integration method is chosen, then, while truncation error may be reduced, rounding error may be increased due to the higher number of floating-point operations per unit time compared to using a high step size or low-order integration method. Arpra will be useful here in determining a decent trade-off between truncation error and rounding error for a simulation.

In any case, one of the main goals of this study was to highlight the potential for computed trajectories in massively parallel software to diverge in unexpected ways. Some take-home messages to consider for parallel SNN simulations are that, while the probability of trajectory divergence is low in fairly quiet SNN models, the possibility is always there, and one should be prepared for it. This is especially true as models become larger and more chaotic, with longer lists of input currents to sum, and simulation trajectories that lurk ever closer, on average, to the firing threshold. Pairwise input current summation may alleviate the issue somewhat, and the algorithm is highly amenable to parallel computation. As parallel compute architectures continue to increase in popularity, rather than measuring simulation reproducibility in terms of bitwise identicality of results, researchers should instead begin to consider reproducibility in terms of statistical metrics and qualitative simulation behaviour.

Recent studies have begun to take reproducibility seriously in biological simulations [56] [57] [58], which was sparked by an ongoing more general debate on what it should mean for computational results to be called reproducible [59]. The Association for Computing Machinery (ACM) currently define three main classes of verifiability [60]. The result is said to be 'repeatable' if the same team, using the same experimental set-up, can produce the same result. A result is 'replicable' if another team can produce the result with the same experimental set-up. Finally, a result is said to be truly 'reproducible' if another team can produce the result using a different experimental set-up.

While it is great that reproducibility of SNN simulation results is taken more seriously these days, little attention is still paid to reproducibility on parallel compute architectures. Researchers from all disciplines who perform computational studies are often obsessed with the concept of bitwise reproducibility in their results. They expect the exact same bit pattern in their output when simulating models using the exact same compute environment, and presume any variance to be caused by software bugs and faulty hardware. However, with the advent of massively parallel compute architectures, this study has exposed how this is simply no longer a reasonable expectation.

Bibliography

[1] T. Huckle, "The Vancouver Stock Exchange Sources."

[2] R. Skeel, "Roundoff Error and the Patriot Missile."

[3] IEEE, "IEEE-754-1985 Standard for Floating-Point Arithmetic," 1985.

[4] IEEE, "IEEE-754-2008 Standard for Floating-Point Arithmetic," 2008.

[5] N. Whitehead and A. Fit-florea, "Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs," tech. rep., NVIDIA, 2011.

[6] D. Monniaux, "The Pitfalls of Verifying Floating-point Computations," *ACM Trans. Program. Lang. Syst.*, vol. 30, pp. 12:1–12:41, May 2008.

[7] E. Yavuz, J. Turner, and T. Nowotny, "GeNN: a code generation framework for accelerated brain simulations," *Scientific Reports*, vol. 6, p. 18854, Jan. 2016.

[8] NVIDIA, "CUDA C Programming Guide," 2018.

[9] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The SpiNNaker Project," *Proceedings of the IEEE*, vol. 102, pp. 652–665, May 2014.

[10] D. Goldberg, "What Every Computer Scientist Should Know About Floating-point Arithmetic," *ACM Comput. Surv.*, vol. 23, pp. 5–48, Mar. 1991.

[11] N. Higham, "The Accuracy of Floating Point Summation," *SIAM Journal on Scientific Computing*, vol. 14, pp. 783–799, July 1993.

[12] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2nd ed., 2002.

[13] J.-M. Chesneaux, S. Graillat, and F. Jezequel, "Numerical validation and assessment of numerical accuracy," *LIP6*, 2009.

[14] GNU, "The GNU C Library," 2018.

[15] B. Hollingsworth, "New "Bulldozer" and "Piledriver" Instructions," 2012.

[16] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," Nov. 2018.

[17] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding," *ACM Trans. Math. Softw.*, vol. 33, June 2007.

[18] S. M. Rump, "Algorithms for Verified Inclusions: Theory and Practice," in *Reliability in Computing* (R. E. Moore, ed.), pp. 109–126, Academic Press, Jan. 1988.

[19] E. Loh and G. W. Walster, "Rump's Example Revisited," *Reliable Computing*, vol. 8, pp. 245–248, June 2002.

[20] W. Krämer, "Generalized Intervals and the Dependency Problem," *PAMM*, vol. 6, no. 1, pp. 683–684, 2006.

[21] C. Barbăro§ie, "Reducing the wrapping effect," *Computing*, vol. 54, pp. 347–357, Dec. 1995.

[22] A. Neumaier, "The Wrapping Effect, Ellipsoid Arithmetic, Stability and Confidence Regions," in *Validation Numerics: Theory and Applications* (R. Albrecht, G. Alefeld, and H. J. Stetter, eds.), Computing Supplementum, pp. 175–190, Vienna: Springer Vienna, 1993.

[23] J. Stolfi and L. H. de Figueiredo, "Self-Validated Numerical Methods and Applications," *21st Brazilian Mathematics Colloquium*, 1997.

[24] L. H. de Figueiredo and J. Stolfi, "Affine Arithmetic: Concepts and Applications," *Numerical Algorithms*, vol. 37, pp. 147–158, Dec. 2004.

[25] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, pp. 1569–1572, Nov. 2003.

[26] J. P. Turner, "Arpra: Arbitrary-Precision Range Analysis," Mar. 2019.

[27] S. M. Rump and M. Kashiwagi, "Implementation and improvements of affine arithmetic," *Nonlinear Theory and Its Applications, IEICE*, vol. 6, no. 3, pp. 341–359, 2015.

[28] S. M. Rump, "INTLAB — INTerval LABoratory," in *Developments in Reliable Computing* (T. Csendes, ed.), pp. 77–104, Dordrecht: Springer Netherlands, 1999.

[29] Y. Bertot, N. Magaud, and P. Zimmermann, "A Proof of GMP Square Root," *Journal of Automated Reasoning*, vol. 29, pp. 225–252, Sept. 2002.

[30] J.-M. Muller, N. Brunie, F. d. Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Basel, 2 ed., 2018.

[31] V. Lefèvre, J. Muller, and A. Tisserand, "Toward correctly rounded transcendentals," *IEEE Transactions on Computers*, vol. 47, pp. 1235–1243, Nov. 1998.

[32] V. Lefèvre and J. Muller, "Worst cases for correct rounding of the elementary functions in double precision," in *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, pp. 111–118, June 2001.

[33] A. Ziv, "Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit," *ACM Trans. Math. Softw.*, vol. 17, pp. 410–423, Sept. 1991.

[34] S. Gal and B. Bachelis, "An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard," *ACM Trans. Math. Softw.*, vol. 17, pp. 26–45, Mar. 1991.

[35] N. Revol and F. Rouillier, "Motivations for an Arbitrary Precision Interval Arithmetic and the MPFI Library," *Reliable Computing*, vol. 11, pp. 275–290, Aug. 2005.

[36] O. Bouissou, E. Goubault, J. Goubault-Larrecq, and S. Putot, "A generalization of p-boxes to affine arithmetic," *Computing*, vol. 94, pp. 189–201, Mar. 2012.

[37] M. Hénon, "A two-dimensional mapping with a strange attractor," *Communications in Mathematical Physics*, vol. 50, pp. 69–77, Feb. 1976.

[38] E. N. Lorenz, "Deterministic Nonperiodic Flow," *Journal of the Atmospheric Sciences*, vol. 20, pp. 130–141, Mar. 1963.

[39] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 89–100, ACM, 2007. event-place: San Diego, California, USA.

[40] A. Diamond, T. Nowotny, and M. Schmuker, "Comparing Neuromorphic Solutions in Action: Implementing a Bio-Inspired Solution to a Benchmark

Classification Task on Three Parallel-Computing Platforms," *Frontiers in Neuroscience*, vol. 9, 2016.

[41] A. Diamond, M. Schmuker, A. Z. Berna, S. Trowell, and T. Nowotny, "Classifying continuous, real-time e-nose sensor data using a bio-inspired spiking network modelled on the insect olfactory system," *Bioinspiration & Biomimetics*, vol. 11, p. 026002, Feb. 2016.

[42] R. D. Traub and R. Miles, *Neuronal Networks of the Hippocampus*. New York, NY, USA: Cambridge University Press, 1991.

[43] C. Morris and H. Lecar, "Voltage oscillations in the barnacle giant muscle fiber," *Biophysical Journal*, vol. 35, pp. 193–213, July 1981.

[44] W. Rall, "Distinguishing theoretical synaptic potentials computed for different soma-dendritic distributions of synaptic input.," *Journal of Neurophysiology*, vol. 30, pp. 1138–1168, Sept. 1967.

[45] S. M. Rump, "Error estimation of floating-point summation and dot product," *BIT Numerical Mathematics*, vol. 52, pp. 201–220, Mar. 2012.

[46] M. Berz and G. Hoffstätter, "Computation and Application of Taylor Polynomials with Interval Remainder Bounds," *Reliable Computing*, vol. 4, pp. 83–97, Feb. 1998.

[47] M. Berz and K. Makino, "Verified Integration of ODEs and Flows Using Differential Algebraic Methods on High-Order Taylor Models," *Reliable Computing*, vol. 4, pp. 361–369, Nov. 1998.

[48] M. Berz, K. Makino, and J. Hoefkens, "Verified integration of dynamics in the solar system," *Nonlinear Analysis: Theory, Methods & Applications*, vol. 47, pp. 179–190, Aug. 2001.

[49] N. S. Nedialkov, V. Kreinovich, and S. A. Starks, "Interval Arithmetic, Affine Arithmetic, Taylor Series Methods: Why, What Next?," *Numerical Algorithms*, vol. 37, pp. 325–336, Dec. 2004.

[50] J. Vignes, "Review on stochastic approach to round-off error analysis and its applications," *Mathematics and Computers in Simulation*, vol. 30, pp. 481–491, Dec. 1988.

[51] J. Vignes, "Discrete Stochastic Arithmetic for Validating Results of Numerical Software," *Numerical Algorithms*, vol. 37, pp. 377–390, Dec. 2004.

[52] F. Jézéquel, J.-M. Chesneaux, and J.-L. Lamotte, "A new version of the CADNA library for estimating round-off error propagation in Fortran programs," *Computer Physics Communications*, vol. 181, pp. 1927–1928, Nov. 2010.

[53] P. Cousot, R. Cousot, J. Feret, A. Mine, L. Mauborgne, D. Monniaux, and X. Rival, "Varieties of Static Analyzers: A Comparison with ASTREE," in *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE '07)*, pp. 3–20, June 2007.

[54] P. Bogacki and L. F. Shampine, "A 3(2) pair of Runge - Kutta formulas," *Applied Mathematics Letters*, vol. 2, pp. 321–325, Jan. 1989.

[55] J. R. Dormand and P. J. Prince, "A family of embedded Runge-Kutta formulae," *Journal of Computational and Applied Mathematics*, vol. 6, pp. 19–26, Mar. 1980.

[56] L. Mulugeta, A. Drach, A. Erdemir, C. A. Hunt, M. Horner, J. P. Ku, J. G. Myers Jr., R. Vadigepalli, and W. W. Lytton, "Credibility, Replicability, and Reproducibility in Simulation for Biomedicine and Clinical Applications in Neuroscience," *Frontiers in Neuroinformatics*, vol. 12, 2018.

[57] T. Manninen, J. Aćimović, R. Havela, H. Teppola, and M.-L. Linne, "Challenges in Reproducibility, Replicability, and Comparability of Computational Models and Tools for Neuronal and Glial Networks, Cells, and Subcellular Structures," *Frontiers in Neuroinformatics*, vol. 12, 2018.

[58] G. Trensch, R. Gutzen, I. Blundell, M. Denker, and A. Morrison, "Rigorous Neural Network Simulations: A Model Substantiation Methodology for Increasing the Correctness of Simulation Results in the Absence of Experimental Validation Data," *Frontiers in Neuroinformatics*, vol. 12, 2018.

[59] H. E. Plesser, "Reproducibility vs. Replicability: A Brief History of a Confused Terminology," *Frontiers in Neuroinformatics*, vol. 11, 2018.

[60] ACM, "Artifact Review and Badging," 2018.