Quantifying the Impact of Single Bit Flips on Floating Point Arithmetic

***Abstract*—In high-end computing, the collective surface area, smaller fabrication sizes, and increasing density of components have led to an increase in the number of observed bit flips. Such flips result in silent errors, i.e., a potentially incorrect result, if mechanisms are not in place to detect them. These phenomena are believed to occur more frequently in DRAM, but logic gates, arithmetic units, and other circuits are candidates for bit flips as well. Previous work has focused on algorithmic techniques for detecting and correcting bit flips in specific data structures.**

This work takes a novel approach to this problem. We focus on quantifying the impact of a single bit flip on specific floating-point operations. We analyze the error induced by flipping specific bits in the IEEE floating-point representation in an architecture-agnostic manner, i.e., without requiring proprietary information such as bit flip rates and the vendor-specific circuit designs.

We initially study dot products of vectors and demonstrate that not all bit flips create a large error and, more importantly, the relative magnitude of the vectors and vector length can be exploited to minimize the error caused by a bit flip. We also construct an analytic model for the expected relative error caused by a bit flip in the dot product and compare this model against empirical data generated by Monte Carlo sampling. We then extend our analysis to stationary iterative methods and prove that these methods converge to the correct solution in the presence of faulty arithmetic.

In general, this effort presents the first step towards rigorously quantifying the impact of bit flips on numerical methods. Our eventual goal is to utilize these results to provide insight into the vulnerability of leadership-class computing systems to silent faults and, ultimately, to provide a theoretical basis for future silent data corruption research.

I. Introduction

Supercomputers have become an essential instrument to push the limits of complex simulations and large-scale data analysis for sciences, industry, and government. High-Performance Computing (HPC) systems have reached multi-petascale capabilities with exascale on the horizon with ever more compute cores. But recent work shows that faults are becoming the norm rather than the exception when running applications at such large scales [1, 2]. Existing fault tolerance schemes (checkpoint/restart [3, 4, 5, 6]) are reaching their scalability limitations [7, 8, 9, 10] while others (e.g., redundant execution [11]), while scalable, may only become feasible at extreme scale and only via capacity computing (increasing job throughput) rather than capability computing (exploiting all resources for an application) [12].

More significant for this work, hardware protection for memory (ECC) detects and corrects the vast majority of bit flips due to radiation from space and decreasing fabrication sizes of semiconductors [1]. Recent work has shown that multi-bit upsets are rare events and chipkill functionality is extremely effective in reducing node failure rates due to DRAM errors [13], while processing cores remain largely unprotected [14, 15]. Thus, novel approaches for scalable resilience, not only in hardware but also in software, are required when component failures and soft errors become the norm rather than the exception for HPC [16, 17, 18].

Contributions: This work calls into question an assumption that the field of computational science has taken for granted for quite some time, namely the assumption that computer arithmetic is reliable. As systems continue to grow in size and density, and as fabrication sizes continue to shrink, silent faults in the hardware may present an increasing uncertainty never before anticipated by users. The goal of this work is to understand the implications of faulty arithmetic and to do so in manner that is theoretically sound and experimentally reproducible. We seek to quantify the impact of a single bit flip on specific numerical methods. From this analysis, we take a first step in characterizing numerical methods by their resilience to silent data corruption (SDC) caused by bit flips.

Imagine a scenario where a user runs a numerical code that converges. The user believes the solution is correct because no errors were observed. Unknown to the user, a single faulty arithmetic operation slightly perturbed their numerical method. *While the method converged, it converged to the wrong solution.* It is precisely this scenario that we are attempting to address. More specifically, given some numerical method and assuming that a single bit flip silently impacts the arithmetic of this method, we seek to assess the implications on the overall solution.

This work approaches the problem of SDC using both analytic modeling and empirical sampling via Monte Carlo. We present findings demonstrating that the impact of SDC in arithmetic can be rigorously analyzed. We also show that it may be possible to exploit the

binary representation of IEEE 754 double precision numbers to minimize the impact of SDC. We further show that convergence for certain algorithms can be guaranteed in the presence of an SDC. Only the number of iterations to convergence is affected by the bit position flipped.

This document is structured into three major themes: 1) Fault characterization; 2) modeling and sampling the impact of a single bit flip in dot products; 3) analyzing the impact of a silent bit flip on the Jacobi iterative method, which leads to a generalization that all stationary iterative methods are resilient to a silent bit flip in arithmetic.

II. Related Work

A number of researchers have approached the problem of SDC in numerical algorithms in various ways; most research takes the approach of treating an algorithm as a black-box and observes the behavior of these codes when run with soft errors injected. Recently, [19, 20] analyzed the behavior of various Krylov methods and observed the variance in iteration count based on the data structure that experiences the bit flip. Shantharam et al.[21] analyzed how bit flips in a sparse matrix vector multiply (SpMV) impact the L^2 norm and observe the error as CG is run. Bronevetsky et al. [22, 23] analyzed several iterative methods documenting the impact of randomly injected bit flips into specific data structures in the algorithms and evaluated several detection/correction schemes in terms of overhead and accuracy. Malkowski et al. [24] analyzed SDC from the perspective of the L1 and L2 caches and proposed an eviction and prefetching scheme that minimizes the amount of time data sits in the unprotected cache. Hoemmen and Heroux proposed a fault tolerant GMRES algorithm based on the principles of flexible preconditioners and demonstrated that their method is resilient to soft errors [25]. Exemplifying the concept of black-box analysis of bit flips, [26] presents BIFIT for characterizing applications based on their vulnerability to bit flips.

Algorithm-based fault tolerance (ABFT) provides an approach to detect (and optionally correct) faults, which comes at the cost of increased memory consumption and reduced performance [27, 28]. The ABFT work by Huang et al. [27] was proven by Anfinson et al. [29] to work for several matrix operations, and the checksum relationship in the input checksum matrices is preserved at the end of the computation. Consequently, by verifying this checksum relationship in the final computation results, errors can be detected at the end of the computation. Costs in terms of extra memory and computation required for ABFT may be amortized for dense linear algebra, and such overheads have been analyzed by many (e.g., [30, 31, 32]). The more subtle problem is that algorithms have to be manually redesigned for ABFT support taking numerical properties (e.g., invariants) into account. Recent work has looked at extending ABFT to additional matrix factorization algorithms [28] and as an alternative to traditional checkpoint/restart techniques for tolerating fail-stop failures [33, 34, 35].

III. Fault Characterization

Let us first establish a clear definition of terminology for bit flips and faults in general. According to Hoemmen's abbreviated taxonomy [25], our focus is best described as one on *transient silent faults*. We extend this taxonomy by introducing a classification called **silent** and present the following definitions:

(1) **Silent faults** are a subset of soft faults, meaning they do not cause immediate program interruption and are **not** detected. These silent faults may occur in hardware units that do not have any safeguards in place to detect bit flips, such as the Arithmetic Logic Unit or registers (and occasionally L1 caches, such as in BlueGene/L). We do not consider *not a number* (NaN) or *infinity* (Inf) to be silent faults because 1) NaN and Inf may be trapped using floating point exceptions, and, more importantly, 2) for the methods presented in this paper NaN or Inf will propagate to the solution where they are a clear indicator that something is incorrect.

(2) **Transient silent faults** are faults that do not persist in the data, i.e., they corrupt data but do persist in the output of the operation that used the data. For instance, a flip in an adder can be modeled as a corrupt input, but the corrupted input is never saved. Hence, the corruption only manifests itself in the output of the adder, which may be saved or used in another operation.

A. Single Bit Flips vs. Multiple Bit Flips

Bit flips are commonly thought to only occur extremely rarely during arithmetic operations inside of arithmetic and logic units (ALUs) and floating-point units (FPUs). This belief is corroborated by years of experience with ALUs/FPUs units on systems providing solutions that match analytic solutions. However, recent work indicates there may be higher bit flip rates than previously thought, not just in memory but also in ALUs/FPUs [15, 14]. In this study, we analyze a single bit flip on the input to some numerical methods, but this single flip on the input may be viewed as multiple flips on the output. Consider the multiplication of two integers, $9 \times 3 = 27$, and assume a bit is flipped, e.g., $9 \rightarrow 13$. This results in the following output:

$$9 \times 3 = 27 = 1001 \times 0011 = 011011,$$
$$13 \times 3 = 39 = 1101 \times 0011 = 100111.$$

Note how a single flip on the input results in multiple incorrect bits in the output. It is for this reason that it may not be appropriate (or even necessary) to model an

excessive number of flips on the input, as the resulting number of bits differing in the output, particularly with floating point numbers, can be drastic.

IV. Background: IEEE 754 Specification

The IEEE 754 specification describes various floating point standards, from which we chose to analyze the 64-bit *double precision* specification called *Binary64* that is widely used in scientific computing today. Consider a number a expressed in scientific notation as

$$a = m \times d^b, \tag{1}$$

where m is the *mantissa*, d^b is the *magnitude*, and b is the exponent. We begin with the analytic model for the Binary64 format

$$v = (-1)^{sign}(1 + \sum_{i=0}^{51} b_i 2^{i-52} \times 2^{e-1023}), \tag{2}$$

where b_i is the i-th bit in the mantissa, *sign* is the sign bit, and $e - 1023$ is the exponent, stored using a bias of 1023, which removes the need for an explicit sign bit for representing negative exponents. The Binary64 format is depicted graphically in Figure 1, where the 0-th bit is the *least significant* bit of the mantissa and the 51-th bit is the *most significant* bit of the mantissa. Bits 52 and 62 represent the least and most significant bits of the exponent. In summary, this specification describes a format that utilizes 52 bits to represent the mantissa, 11 bits to represent the exponent, and one bit to represent the sign. The exponent is stored using a bias of 1023. This bias may be exploited to enhance resiliency as demonstrated in Section V.

Figure 1. Representation of the Binary64 IEEE format.

V. Case Study: Vector Dot Products

To begin our investigation, we chose to assess the susceptibility of the dot product, or inner product, of two N-dimensional vectors to a silent bit flip in arithmetic. We make this choice since many linear algebra operations can be decomposed into dot products and many numerical methods are built on the concept of evaluating inner products, typically in the form matrix-vector products.

A. Analytic Model

Let us present an analytic model for a bit flip in a double precision floating point number. From this model, we then compose a dot product. Consider a number a written in normalized scientific notation following Eq. (1). Let us define a notation where

- m_a is the mantissa of a,
- d_a is the magnitude of a,

and $a = d_a m_a$. Assuming a is represented using Eq. (2), we may express a perturbed floating point number created by a single bit flip as

$$\hat{a} = \begin{cases} a \pm 2^{j-52} d_a & \text{flip in the mantissa bits,} \quad (3a) \\ a 2^{\pm 2^j} & \text{flip in the exponent bits,} \quad (3b) \\ -a & \text{flip in the sign.} \quad (3c) \end{cases}$$

Given two N-dimensional vectors $\mathbf{u}$ and $\mathbf{v}$, the dot product is defined as

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^\mathsf{T} \mathbf{v} = \sum_{i=1}^{N} c_i, \text{ where } c_i = u_i v_i. \tag{4}$$

The **absolute error** between the correct and perturbed dot product, assuming a perturbation in $\mathbf{u}$, is

$$\begin{aligned} error_{abs} &= \left| \overbrace{\sum_{i=1}^{N} c_i}^{\text{correct}} - \left(\overbrace{\sum_{i=1}^{N} c_i - c_k + \hat{u}_k v_k}^{\text{perturbed}} \right) \right|, \\ &= \left| c_k - \hat{u}_k v_k \right|, \end{aligned}$$

where c_k is a perturbed product from Eq. (4) and $\hat{u}_k$ follows either Eq. (3a), Eq. (3b), or Eq. (3c).

The relative error resulting from a single bit flip in $\mathbf{u}$ or $\mathbf{v}$ may be expressed as

$$error = \frac{error_{abs}}{\left| \sum_{i=1}^{N} c_i \right|} = \frac{|c_k - \hat{u}_k v_k|}{\left| \sum_{i=1}^{N} c_i \right|}. \tag{5}$$

Assuming that all elements in both vectors have similar relative magnitudes, then we get

$$d_{u_k} \approx d_{v_k} \text{ for all } k = 1, \ldots, N. \tag{6}$$

This assumption is encouraged by numerical analysts (but may not always be feasible). This assumption allows us to construct a theoretically sound model where we rigorously model the effects of a bit flip for vectors of similar magnitude. From this, we may correctly and accurately express a bound on the error produced by a bit flip. place.

$$d = \max\{d_{u_i} \times d_{v_i}\}, \text{ for } i = 1, \ldots, N,$$

and the product of two elements may be approximated as

$$c_i = u_i v_i \leq d. \tag{7}$$

Eq. (4) may then be approximated by

$$\sum_{i=1}^{N} c_i \approx d \times N. \tag{8}$$

Note that Eq. (8) is not the 1-norm, $||\mathbf{x}||_1 = \sum_{i=1}^{N} |x_i|$.

We intentionally avoid the use of norms in the analytic model for two reasons: 1) the norms may be assumed to be real numbers, when we actually have approximations

based on the IEEE standard, and 2) we wish to create a model that is based on magnitudes and mantissas, such that we may understand the impact of a bit flip in each location. For these reasons, we avoid the use of norms in our dot product analysis because they are the incorrect tool with respect to the our modeling effort.

B. Relative Error by Bit Location

Substituting Eq. (7) and Eq. (8) into Eq. (5), we obtain a closed form for the relative error given a bit flip in the exponent ($error_e$), sign ($error_s$), and mantissa ($error_m$) under the assumption from Eq. (6).

Assume a bit flip happens to an element of $\mathbf{u}$, and the flip affects a bit in the **mantissa**:

$$error_m \approx \frac{m_{v_k} 2^{j-52}}{N}, \text{ for } j = 0, \ldots, 51. \tag{9}$$

We may bound m_{v_k} by recognizing that the mantissa bits, independent of the magnitude, may range from 2^{-51} to 2^{-1}. Selecting the maximum, we know that $m_a \leq 2^{-1}$. Hence, Eq. (9) may be further simplified to

$$error_m \approx \frac{2^{j-53}}{N}, \text{ for } j = 0, \ldots, 51. \tag{10}$$

Assuming a bit flip happens to an **exponent** bit of $\mathbf{u}$, we have

$$error_e \approx \frac{\left|1 - 2^{\pm 2^j}\right|}{|N|}, \text{ for } j = 0, \ldots, 10. \tag{11}$$

Eq. (11) may be further decomposed based on whether the flip is a 1 to 0, or 0 to 1. Flipping a 0 to 1 in the exponent implies multiplying the number by some power of two. This *magnifies* the error caused by the bit flip. Conversely, a 1 to 0 implies dividing the number by a power of two, which *minimizes* the error caused by a bit flip.

$$error_e \approx \begin{cases} \dfrac{1 - 2^{-2^j}}{N} \text{for } j = 0, \ldots, 10;\ bit_{j+52} = 1 & \text{(12a)} \\ \dfrac{2^{2^j} - 1}{N}, \text{ for } j = 0, \ldots, 10;\ bit_{j+52} = 0 & \text{(12b)} \end{cases}$$

Let us next assume a flip impacts the **sign** bit of an element in the vector $\mathbf{u}$. By substituting Eq. (3c) into Eq. (5), we arrive at the relative error for a flip in the sign bit

$$error_s \approx \frac{2}{N}. \tag{13}$$

To summarize, the relative error can be generalized based on the location of the bit leading to three fundamental forms for the relative error. We summarize our findings thus far in Table I. Our next goal is to express the expected relative error, which will provide a theoretical basis that is later used to assess the quality of a Monte Carlo approximation of the expected error.

Table I
RELATIVE ERROR OF A SINGLE BIT FLIP IN DOT PRODUCT OF TWO N-DIMENSIONAL VECTORS WITH SIMILAR RELATIVE MAGNITUDES.

Bit Location	Relative Error		# Bits	Eq.
Mantissa	$\frac{2^{j-53}}{N}$,	for $j = 0, \ldots, 51$	52	(10)
$\text{Exponent}_{1\to 0}$	$\frac{1 - 2^{-2^j}}{N}$,	for $j = 0, \ldots, 10$; and $bit_{j+52} = 1$	11	(12a)
$\text{Exponent}_{0\to 1}$	$\frac{2^{2^j} - 1}{N}$,	for $j = 0, \ldots, 10$; and $bit_{j+52} = 0$	11	(12b)
Sign	$\frac{2}{N}$		1	(13)

C. Expected Relative Error per Random Flip

Next, we assume a single bit randomly flips while performing the dot product of two vectors. To this end, we introduce a discrete random variable ω that represents the error induced given a single bit flip. We may now construct the expected value $\mathbb{E}$ of the relative error given a bit flip in a specific location using Table I.

The expected relative error given a bit flip in the mantissa is

$$\mathbb{E}[\omega|\text{bit flip in the mantissa}] = \frac{1}{52} \sum_{j=0}^{51} \frac{2^{j-53}}{N},$$

and for the sign bit we have

$$\mathbb{E}[\omega|\text{bit flip in the sign}] = \frac{2}{N}.$$

The expected value of the relative error given a flip in the exponent is slightly more complicated, as we must account for whether the flip was a $1 \to 0$ or $0 \to 1$.

$$\mathbb{E}[\omega|\text{bit flip in the exponent}] = \frac{1}{11} \sum_{j=0}^{10} \mathbb{E}[\omega|\text{bit}_j], \tag{14}$$

$$\mathbb{E}[\omega|\text{bit}_j] = \begin{cases} \dfrac{1 - 2^{-2^j}}{N}, \text{ if } bit_{j+52} = 1; \\ \dfrac{2^{2^j} - 1}{N}, \text{ if } bit_{j+52} = 0. \end{cases}$$

D. Monte Carlo Sampling

We next develop a better understanding of how vector magnitudes and size impact the expected relative error given that the expected value derivations are based on the assumption that the two vectors have approximately the same magnitudes. To conduct Monte Carlo sampling, we must first determine a mechanism for tallying success, and we must define what *success or failure* mean.

- **Vector Creation**

1) Generated randomly using C stdlib rand().
2) For each vector, we fix each element's magnitude to the bit pattern 2^{-50} to 2^{50} (101 bit patterns). This corresponds to the base ten numbers in the range 8.8×10^{-16} to $1.1 \times 10^{+15}$. This range was chosen because 2^{-50} is roughly machine precision, which is 2^{-53} for the smallest number. The numbers

in this range are utilizing the highest *precision* that Binary64 offers. The further distant from 2^0 a number is, the lower its precision becomes.

- **Sample definition and Error Calculation**

1) A random sample is defined by generating two random N length vectors and computing the relative error considering all possible $2 \times 64 \times N$ bit flips.
2) A tally is defined by failure, which we define to be any relative error that is greater than 1×10^{-4}.
3) An empirical estimate of the expected relative error is computed by dividing the number of failures by the number of bits considered times the vector length times 2 times the number of random samples (M) taken for a given magnitude combination, i.e., $failures/(2 \times 64 \times N \times M)$.

- **Visualization**

1) To visualize the expected relative error, we construct tallies for each magnitude combination, i.e., 101×101 unique combinations, and each combination is sampled M times.
2) We summarize this information in a surface plot, where the x- and y-axes denote the $\log_2$ of the relative magnitude of the vector **u** and **v**, respectively. The height of the surface plot indicates the probability of seeing a relative error larger than 1×10^{-4}.

Figure 2 presents four surface plots as described in the Visualization bullet. To interpret a graph, the x-axis indicates the magnitude that all elements of the vector **u** were forced to have while the mantissa was randomly generated. Likewise, the y-axis indicates the magnitude that all elements of the vector **v** were forced to have. Each x-y intersection represents 1,000,000 random vector samples, where the dot product was computed and failures tallied. The height of the surface at an [x,y] location indicates the probability of observing an error larger than 1×10^{-4} given a single bit flip. The height of all plots is fixed to the range [0,.4] to facilitate the comparison between surface plots. From these surfaces, one may immediately recognized the unusual structure of these graphs: When both vectors have magnitudes larger than 2^0, the probability of failure is noticeably higher; yet, when both vectors have magnitudes less than 2^1, the probability of failure is approaching zero as the vector size increases.

E. Per Bit Analysis of Surface Plots

To better understand their structure and to illustrate why these surface plots look the way they do, we take two slices of the surface and look at the per-bit probability of a failure. The slices chosen feature vectors with the *same* relative magnitudes and *inverse* relative magnitudes. Intuitively, these figures slice from the back-most corner of the plot to the front for similar magnitudes, and they slice from the left-most corner to right-most corner for inverse magnitudes. Comparing Figure 3(a) to Figure 3(b) reveals that bits immediately left of the 2^0 axis produce significantly fewer failures than bits on the right side of the 2^0 center axis. These variations about the magnitude 2^0 can be explained in two ways: First, by examining the bias used to store the exponent, and second, by looking at the analytic model we constructed for a flip in the exponent. Reconsider Eq. (2) where the bias of 1023 is used to store the exponents. For example, to represent 2^0, 1023 is stored because $2^0 = 2^{1023-1023}$. A brief analysis of the binary pattern of the stored exponent is shown in Table II. The emphasis here is on the number of ones versus zeros, which is drastically different when the stored exponent is less than 1024. This corresponds to magnitudes slightly less than 2^1.

Table II
BINARY PATTERNS IN THE EXPONENT.

	Exponent Bits		
Base10	$b_{62} \quad \ldots \quad b_{52}$	Bias Relation	Effective Exp.
5	10000000001	$2^{1025-1023}$	2^2
2	10000000000	$2^{1024-1023}$	2^1
1	01111111111	$2^{1023-1023}$	2^0
.2	01111111100	$2^{1020-1023}$	2^{-3}
.5	01111111110	$2^{1022-1023}$	2^{-1}

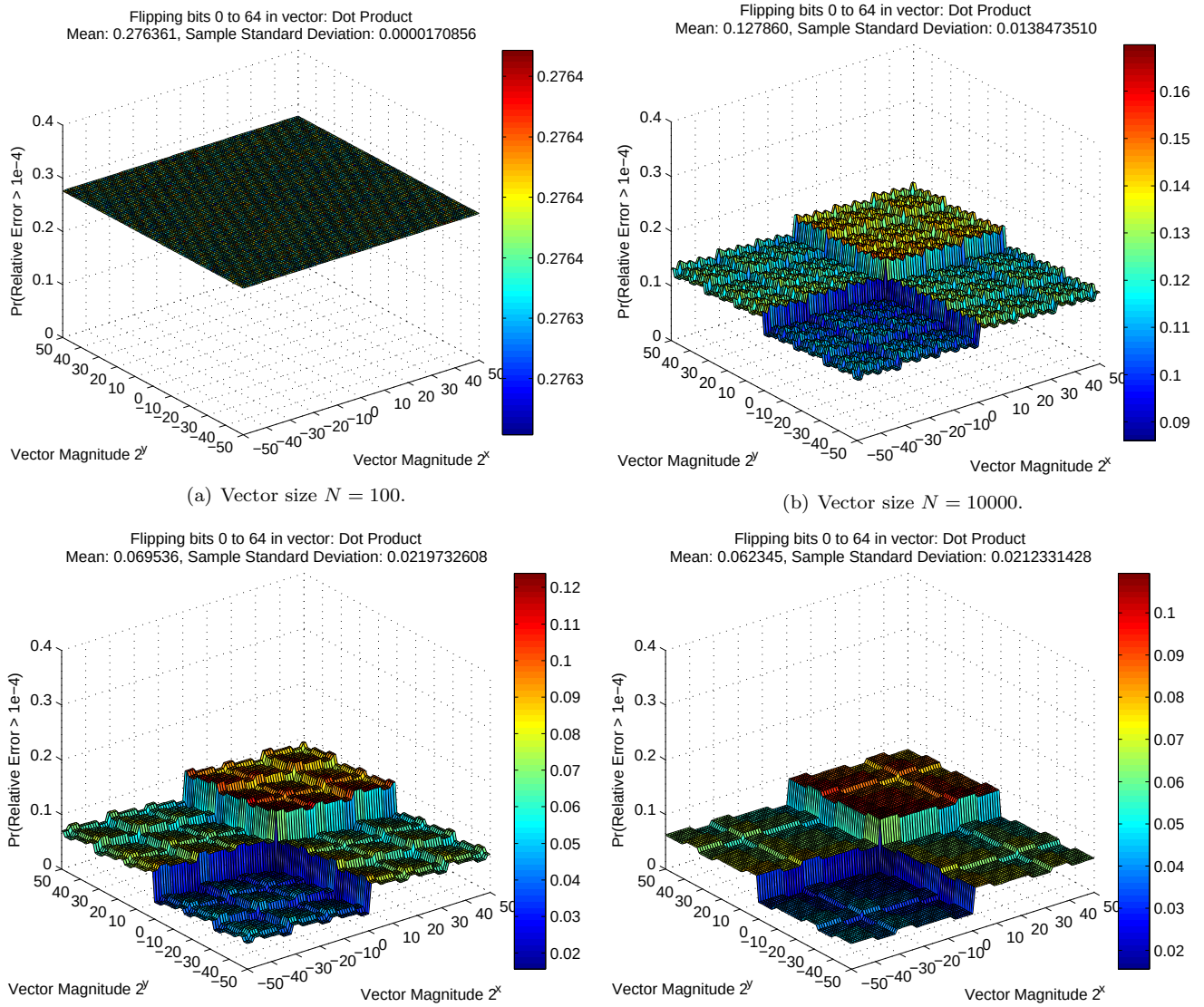
We also previously presented an analytic model that rigorously explains this (see Table I): When the exponent flips from $1 \rightarrow 0$, we have $\lim_{N\rightarrow\infty} \frac{1-2^{-2^j}}{N} = 0$. When a bit in the exponent flips $0 \rightarrow 1$, the error is magnified. The limit still converges to zero, but it requires a much larger N.

VI. Comparison of the Analytic Model and Monte Carlo Sampling

We now compare the error observed through Monte Carlo samples with the expected relative errors computed in Section V-A.

Figure 4 shows the shape of the analytic expected relative error given a flip in the exponent, but excluding the exponent's most significant bit. The reason for this exclusion is that, regardless of scaling, a flip in the most significant exponent bit can produce an extremely large error. From Section V-E, we know that, given vectors with relative magnitudes greater than 2^0, the error will be large, even up to 1 million length vectors.

In Figure 5, we compare the error observed while performing Monte Carlo sampling with the expected error from the model constructed in Section V-A. We sampled up to $M = 1$ million random vectors per data point, which implies a Monte Carlo error of $error_{MC} = 1/\sqrt{M} \approx 0.001$. These results confirm the validity of our model. Hence, the model presented in Eq. (14) may



(a) Vector size $N = 100$.

(b) Vector size $N = 10000$.

(c) Vector size $N = 100000$.

(d) Vector size $N = 1000000$.

Figure 2. Probability of observing an error larger than 1×10^{-4}, given four different vector sizes.

be used to determine a sufficient vector size that will mitigate the impact of a bit flip when performing dot products, assuming similar relative magnitudes.

VII. Stationary Iterative Methods

A. The Jacobi Method

The Jacobi method (see Algorithm 1) is a stationary iterative method that may be used to solve systems of linear equations. While this method is rarely used as the *sole* means for solving a linear system of equations in modern codes, it is still used as a preconditioner and, more recently, as a means to exploit massive parallelism on accelerators. In this section, we demonstrate that the Jacobi method is resilient to single bit flips silently occurring within its arithmetic.

The Jacobi method is derived by splitting the $N \times N$ matrix $\mathbf{A}$ into its diagonal and off-diagonal components

$$\mathbf{A} = \mathbf{D} - \mathbf{R}, \tag{15}$$

where $\mathbf{D}$ is the diagonal of $\mathbf{A}$ and $\mathbf{R}$ contains the off-diagonal elements of $\mathbf{A}$. Suppose we seek the solution to the linear system

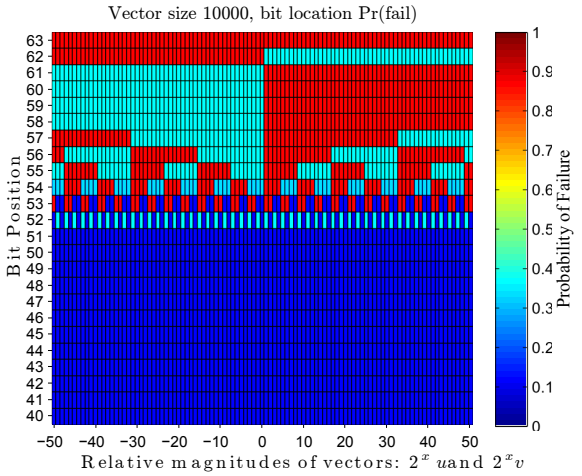
$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{16}$$

By substituting the split version of $\mathbf{A}$ from Eq. (15) into Eq. (16) and solving for $\mathbf{x}$, we get

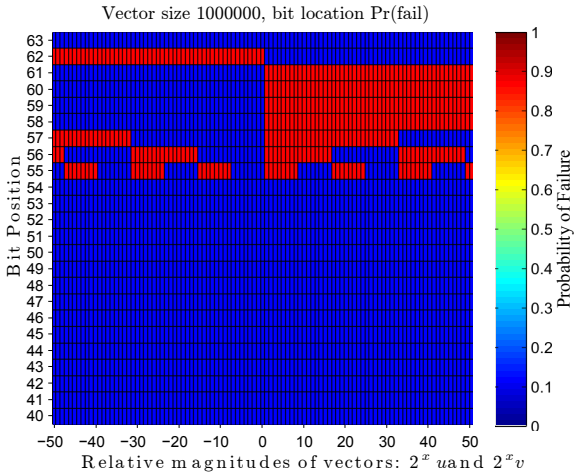
$$\mathbf{x} = \mathbf{D}^{-1}\mathbf{R}\mathbf{x} + \mathbf{D}^{-1}\mathbf{b}.$$

We now obtain the Jacobi method by defining the iterates

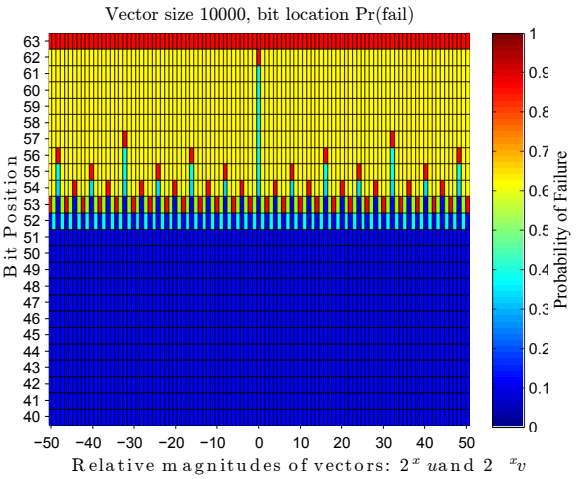
$$\begin{aligned}\mathbf{x}^{(m+1)} &= \mathbf{D}^{-1}\mathbf{R}\mathbf{x}^{(m)} + \mathbf{D}^{-1}\mathbf{b}, \\ &= \mathbf{G}\mathbf{x}^{(m)} + \mathbf{f}.\end{aligned}$$



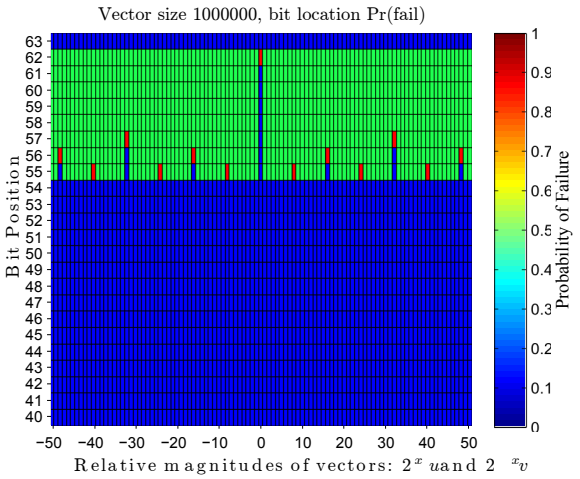
(a) Vector size $N = 10000$, same relative magnitudes.



(b) Vector size $N = 1000000$, same relative magnitudes.



(c) Vector size $N = 10000$, inverse relative magnitudes.



(d) Vector size $N = 1000000$, inverse relative magnitudes.

Figure 3. Probability of observing an error larger than 1×10^{-4}, by bit position, with bits 39 to 0 never yielding an error larger than 1×10^{-4}. The left column contains slices from Figure 2(b), and the right column contains slices from Figure 2(d).

In this form, $\mathbf{G} = \mathbf{D}^{-1}\mathbf{R}$ is known as the iteration matrix, and convergence analysis is based solely on the spectral properties of $\mathbf{G}$.

In the context of a bit flipping silently, we have some sequence of steps where the m-th iterate, $\mathbf{x}^{(m)}$, is computed correctly, but a silent arithmetic error perturbs the $m+1$-th iterate. More precisely, we have

$$\begin{aligned}
\mathbf{x}^{(1)} &= \mathbf{G}\mathbf{x}^{(0)} + \mathbf{f}, \\
&\vdots \\
\mathbf{x}^{(m)} &= \mathbf{G}\mathbf{x}^{(m-1)} + \mathbf{f}, \\
\hat{\mathbf{x}}^{(m+1)} &= \begin{cases} \hat{\mathbf{G}}\mathbf{x}^{(m)} + \mathbf{f} \\ \mathbf{G}\mathbf{x}^{(m)} + \hat{\mathbf{f}} \end{cases} \\
&\vdots \\
\hat{\mathbf{x}}^{(z)} &= \mathbf{G}\hat{\mathbf{x}}^{(z-1)} + \mathbf{f}
\end{aligned}$$

where some arithmetic operation on the right-hand side produces an incorrect result. We assume that the bit flip is not persistent, i.e., the flip may perturb an element of $\mathbf{G}$ leading to $\hat{\mathbf{G}}$, but for all subsequent calculations $\mathbf{G}$ is correct, and likewise for $\mathbf{f}$. This assumption is valid because we assume that the bit flip impacts a single primitive arithmetic operation, *i.e.*, addition, subtraction, multiplication, and division. We assume that since $\mathbf{G}$, $\mathbf{f}$, and $\mathbf{x}^{(m)}$ are not being written to, any perturbation that impacts these data structures will not corrupt the structures themselves — instead, the flip corrupts the output of the operation ($\mathbf{x}^{(m+1)}$) that used the perturbed data. This implies that once the bit flip has occurred, the error from the flip may or may not manifest itself immediately. It is precisely this silent error that we seek to understand and this scenario that we wish to analyze in terms of its impact on convergence of the Jacobi method.

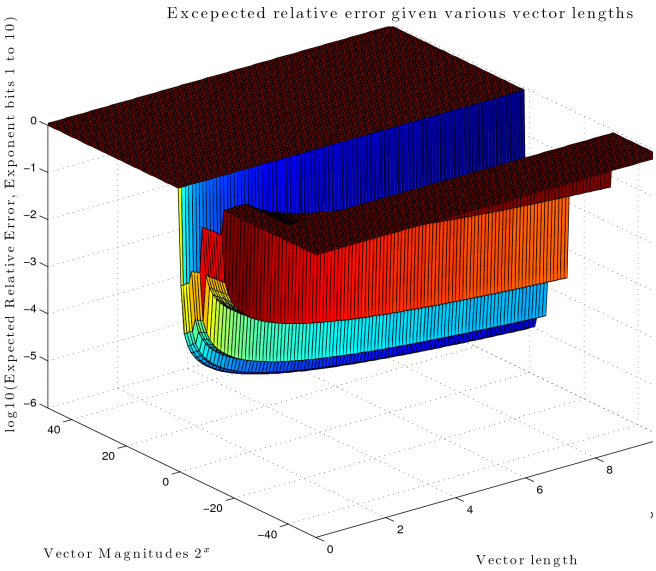


Figure 4. Expected relative error for a flip in the exponent, excluding the most significant exponent bit, assuming vectors have similar relative magnitudes, and various vector lengths.

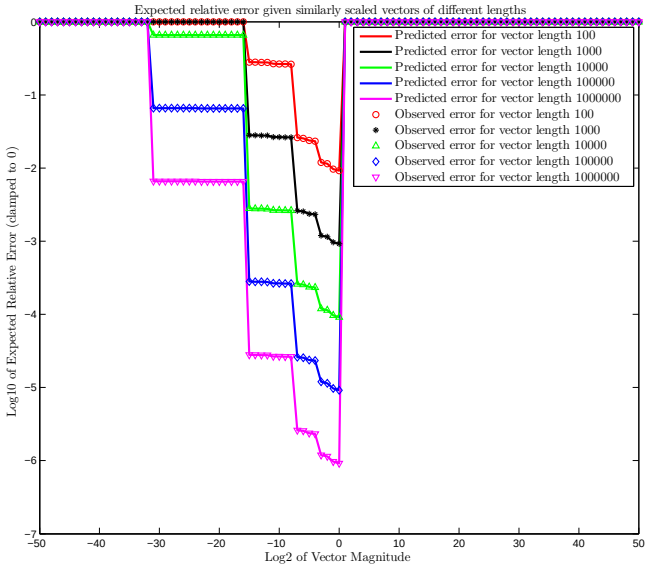


Figure 5. Comparison of observed error caused by a flip in the exponent, excluding the most significant bit, for sampled vector sizes having similar relative magnitudes.

B. Proof of Jacobi Convergence

Suppose we are solving the system $\mathbf{Ax} = \mathbf{b}$. The Jacobi method is expressed in the general form

$$\mathbf{x}^{(k+1)} = \mathbf{Gx}^{(k)} + \mathbf{f}, \tag{17}$$

where $\mathbf{G} = \mathbf{D}^{-1}\mathbf{R}$ is the iteration matrix and $\mathbf{f} = \mathbf{D}^{-1}\mathbf{b}$, $\mathbf{D}$ contains the diagonal entries and $\mathbf{R}$ the off-diagonal entries of $\mathbf{A}$. We may also express the solution $\mathbf{x}$ implicitly as

$$\mathbf{x} = \mathbf{Gx} + \mathbf{f}.$$

Algorithm 1: Pseudocode for the Jacobi Method.

```
// Use the jacobi method to solve the
// linear system Ax = b
```
Input: $N \times N$ matrix $\mathbf{A}$
Input: $N \times 1$ vector $\mathbf{b}$
Input: $N \times 1$ vector $\mathbf{x}^{(0)}$ initial guess or zero
```
// Loop until convergence
```
for $m = 0$ **to** *Convergence* **do**
  for $i = 0$ **to** N **do**

$$x_i^{(m+1)} := \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{N} a_{ij}x_j^{(m)}\right);$$

  end
```
  // Compare residual against stopping
     criteria
```
  if $\frac{\left|\left|\mathbf{b} - \mathbf{Ax}^{(m+1)}\right|\right|_2}{||\mathbf{b}||_2} <$ *Stopping Criteria* **then**
```
     // The Jacobi method has converged
```
     break;
end

To prove that the Jacobi method will converge if a bit flip impacts an arithmetic operation, we introduce the following lemma. The proof can be found in a number of texts that we point the reader to [36, pp 163].

Lemma 1. *For an arbitrary square matrix* $\mathbf{Q}$*, the* $\lim_{k\to\infty} \mathbf{Q}^k = 0 \iff \rho(\mathbf{Q}) < 1$*, where* $\rho(\mathbf{Q})$ *is the spectral radius of* $\mathbf{Q}$ *(the absolute largest eigenvalue).*

We also must be able to express the error of each iterate with the solution.

Lemma 2. *The error between an iterate* $\mathbf{x}^{(k)}$ *and the exact solution* $\mathbf{x}$ *may be expressed recursively as*

$$\mathbf{e}^{(k)} = \mathbf{Ge}^{(k-1)}.$$

Proof: The error between an iterate and the exact solution is $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$, substituting $\mathbf{x} = \mathbf{Gx} + \mathbf{f}$ and $\mathbf{x}^{(k)} = \mathbf{Gx}^{(k-1)} + \mathbf{f}$ in this form we obtain

$$\begin{aligned}\mathbf{e}^{(k)} &= \mathbf{G}[\mathbf{x} - \mathbf{x}^{(k-1)}] \\ &= \mathbf{Ge}^{(k-1)}\end{aligned}$$

□

Theorem 1. *The sequence of vectors* $\mathbf{x}^{(m)}$ *from Eq.* (17) *converges to the solution of* $\mathbf{Ax} = \mathbf{b}$ *given a silent bit flip in arithmetic, if* $\rho(\mathbf{G}) < 1$*.*

Proof: Let a silent bit flip in arithmetic happen on the k-th iteration of Eq. (17). We then have some perturbed iterate $\hat{\mathbf{x}}^{(k)} = \mathbf{x}^{(k)} + \boldsymbol{\xi}$, where $\boldsymbol{\xi}$ is an N-dimensional vector representing the error caused by a bit flip. We now must propagate this error $\boldsymbol{\xi}$ from the k-th iteration to some iteration z, and determine if the error

increases or decrease. First, we express the error that is introduced at each iteration due to a bit flip. Note that the error introduced by the bit flip will be propagated using recursion to latter iterations, and so we must not incorrectly add any additional *new* error at subsequent iterations.

$$\mathbf{e}^{(z)} = \begin{cases} \mathbf{x} - \mathbf{x}^{(z)} = \mathbf{G}\mathbf{e}^{(z-1)} & \text{for } z \neq k, z > 0; \text{(18a)} \\ \mathbf{x} - \hat{\mathbf{x}}^{(z)} = \mathbf{G}\mathbf{e}^{(z-1)} + \boldsymbol{\xi}_k & \text{for } z = k; \text{(18b)} \\ \mathbf{e}^{(0)} & \text{for } z = 0. \text{(18c)} \end{cases}$$

Recursively applying Eq. (18) from the z-th iteration to the first iteration, we obtain

$$\mathbf{e}^{(z)} = \mathbf{G}^{z}\mathbf{e}^{(0)} + \mathbf{G}^{z-k+1}\boldsymbol{\xi}_k.$$

By Lemma 1, as $z - k + 1$ approaches infinity, both terms will become zero. Therefore, the Jacobi method will converge to the true solution $\mathbf{x}$ in spite of a faulty arithmetic operation. □

C. Error Analysis

The preceding proof shows that the Jacobi method will always converge given some ***numeric*** perturbation, but when dealing with computer arithmetic the perturbation may not be numeric. In general, IEEE floating point numbers can either be numeric, or they may be infinity (Inf) or *not a number* (NaN). These special cases are represented by having all ones in the exponent, and if the mantissa is zero, then it is by definition a Inf, but if the mantissa is non-zero, it is NaN.

This could pose a problem, but **NaN and Inf are *not* silent**, both special cases have the property that any operation on one of them produces its identity, e.g., $NaN + 1 = NaN$. This means that once a NaN or Inf begins propagating through the solution, it will remain in the solution. Hence, convergence will be impossible. Because of these special cases, we only need to enhance the Jacobi algorithm (Algorithm 1) to be aware of these special cases so that the code will not remain stuck in an infinite loop. We can do this by adding a simple test on Line 4 and appending the condition that the residual is not a NaN or Inf. This will cause the Jacobi method to halt, or, alternatively, its iteration count may be reset and $\mathbf{x}^{(m+1)}$ set to $\mathbf{x}^{(0)}$, effectively restarting the algorithm from the beginning. The choice to halt the algorithm or restart is beyond the scope of this work. Instead, this work specifically focuses on the problem that a method may give no indication of a silent fault and present an incorrect result, which we might (falsely) believe to be correct. Our aim is to ensure that the method either eventually converges to the correct solution, or in the worst case, present a clear sign that an error occurred (indicated by NaN or Inf).

D. Magnitude Scaling

We now explore the option of using our findings regarding vector scaling to demonstrate that, if possible, vector scaling can enable the Jacobi method to converge to the correct solution in the presence of a bit flip, at a rate similar to the algorithm executing without the bit flip. While Figure 5 shows that scaling can minimize error, the requirement that all values have similar magnitudes is hard to satisfy, particularly in iterative methods, where the vector $\mathbf{x}$ changes every iteration. We leave it to future work to develop a robust scaling technique, but we will present results demonstrating that, if such a method was found, the findings in this work allow Jacobi to not only converge to the correct solution in the presence of a bit flip, but also to converge at a rate close to the bit flip-free rate.

We start with an important fact: Given some scalar α, the scaled linear system $(\alpha\mathbf{A})\mathbf{x} = \alpha\mathbf{b}$ will produce the same solution $\mathbf{x}$ as if no scaling took place. In Algorithm 2, we present a modified Jacobi method that implements scaling. Of particular interest is Line 1, where we introduce a scaling function that determines the correct scaling factor. We experimented with two choices for this function: 1) computing the average magnitude from all elements in $\mathbf{A}$ and $\mathbf{b}$, and 2) taking the maximum magnitude from all elements in $\mathbf{A}$ and $\mathbf{b}$. Note that we can determine the average and maximum by inspecting the bias stored in the exponent bits, and we want this bit pattern to be 1023 or slightly smaller. We now demonstrate the calculation of α:

1) **Maximum** scaling: select the largest bias, which is the largest magnitude, $b_{\max}$, and then the correction should be
$$\alpha = 2^{(1023 - b_{\max})}.$$
2) **Average** scaling: compute the average of all biases in $\mathbf{A}$ and $\mathbf{b}$, b_{avg}, and then the correction should be
$$\alpha = 2^{(1023 - b_{\text{avg}})}.$$

We now present the results of injecting a bit flip into the Jacobi algorithm and show how scaling can impact the number of iterations for Jacobi to converge to the correct solution. We first generate a random system of equations that satisfies the convergence criteria for Jacobi (Diagonal Dominance). Solving this system using the Jacobi method, we find that Jacobi converges after **96 iterations**. We then solve the same system and flip the 61-st bit (an exponent bit) and observe the number of iterations required to converge. Figure 6 shows the result of Jacobi solving this random system **96** times, where we inject the bit flip on different iterations up to iteration 96, which is when Jacobi converges given no bit flips. The x-axis indicates the iteration where

Algorithm 2: Pseudocode for a flip tolerant, scaled Jacobi Method.

```
   // Use the jacobi method to solve the
   // linear system Ax = b
   Input: N × N matrix A
   Input: N × 1 vector b
   Input: N × 1 vector x^(0) initial guess or zero
   // Compute scaling factor
1  α := compute_scaling_factor(A, b);
   // Scale A and b
2  A := αA;
3  b := αb;
   // Loop until convergence
4  for m = 0 to Convergence do
5      for i = 0 to N do
6          x_i^(m+1) := (1/a_ii) ( b_i − Σ_{j=1, j≠i}^{N} a_ij x_j^(m) );
7      end
8      r := ||b − Ax^(m+1)||_2 / ||b||_2 ;
       // Check for Inf or NaN
9      if r is NaN or Inf then
           // Restart or Halt with error message
10         break;
11
       // Compare against stopping criteria
12     if r < Stopping Criteria then
           // The Jacobi method has converged
13         break;
14
15 end
```

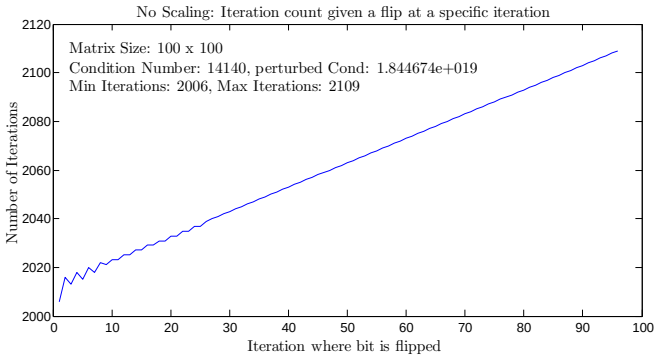


Figure 6. Jacobi **without** scaling, bit flip in the exponent.

the bit was flipped, and the y-axis plots the iteration count. We repeat this experiment again using **average** scaling and **max** scaling. The results are depicted in Figures 7 and 8, respectively. One can immediately tell that scaling can drastically impact the number of

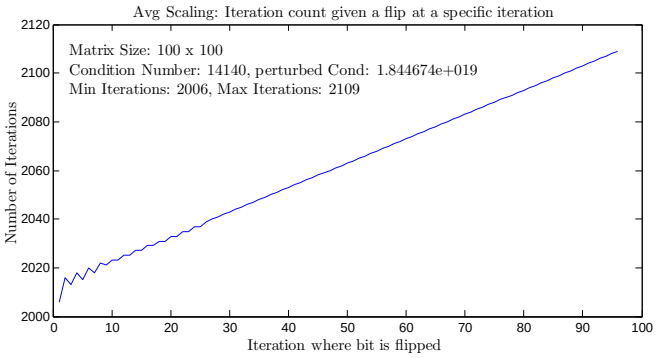


Figure 7. Jacobi+**average** scaling, bit flip in the exponent.

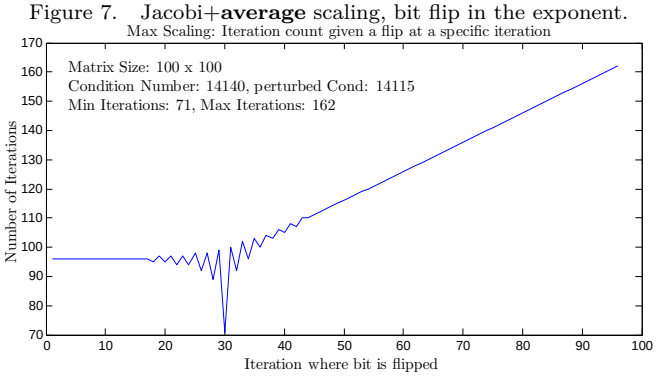


Figure 8. Jacobi+**max** scaling, bit flip in the exponent.

iterations. It is also evident that the choice of the scaling algorithm is critical. If the bit flip happens to an item that has near maximum magnitude, then max scaling works well, but it is not guaranteed to work. In the case that the flip occurs in a value larger than the average, then average scaling does not offer a benefit. We feel that exploiting scaling deserves more investigation but leave this to future work as it is likely that a correct scaling strategy depends on the problem or class of problems being solved.

Also of interest is that, in the case of max scaling, the bit flip *accelerated* convergence when flipped on the 30th iteration. We believe this occurrence can be explained — and possibly exploited — by merging our scaling requirement with the theory backing the Successive Over Relaxation (SOR) method, which we leave as future work.

E. Reliability and Complexity Analysis

It should be noted that on Line 1 of Algorithm 2, we violate our assumption that $\mathbf{A}$ and $\mathbf{b}$ are read-only inside of the algorithm. **This means that if a bit flip impacted the scaling or scaling factor calculation, then this bit flip would not be *transient***, and we could potentially converge to the wrong solution, since we would solve the wrong linear system $\hat{\mathbf{A}}$ or $\hat{\mathbf{b}}$. Because of this, we must expend additional resources to ensure that when $\mathbf{A}$ and $\mathbf{b}$ are written to, the values stored in

these structures is correct. To guarantee that the values are correct, we need to redundantly compute lines 1–3. In general, the computational cost of computing α requires at least $\Omega(N^2 + N)$, since all values in the $N \times N$ matrix $\mathbf{A}$ must be inspected and all N values in $\mathbf{b}$ must be inspected. Likewise, the cost of actually scaling $\mathbf{A}$ requires $\Theta(N^2)$ computation, and scaling $\mathbf{b}$ requires $\Theta(N)$ computation. If we assume that Jacobi is solved using the point-wise algorithm presented in Algorithms 1 and 2, i.e., the iteration matrix $\mathbf{G}$ is not explicitly formed, then the Jacobi method requires $\Theta(N^2)$ calculations per iteration.

To ensure that both $\mathbf{A}$ and $\mathbf{b}$ are written correctly, assume we compute each twice, and that we perform the arithmetic in the same order, e.g., the results should be identical. In the event of a discrepancy we compute an additional third tie breaker to resolve the correct value. This task could also be accomplished by using recovery blocks (containment domains) [37, 38]. Computing the scaling factor will take $\Theta(N^2 + N)$, and applying the scaling will take $\Theta(N^2)$ for $\mathbf{A}$ and $\Theta(N)$ for $\mathbf{b}$.

The overhead for computing the scaling factor and applying the scaling is easily amortized. Recognizing that computing α with no redundancy is asymptotically equivalent to computing an iteration of Jacobi, then computing the scaling factor k redundant times is equivalent to adding k iterations. Similarly, scaling both $\mathbf{A}$ and $\mathbf{b}$ is equivalent to running one iteration, so computing these structures k redundant times is equivalent to adding k additional iterations. In summary, the scaling algorithm using redundant computation is equivalent to adding $2k$ iterations, where k is the degree of redundancy. Clearly, in the case of triple redundancy, 6 additional iterations are minute in comparison to the total number of iterations.

F. Conclusion and Extension to other Iterative Solvers

Theorem 1 was constructed intentionally to be generic. It proves that *any* iterative method that can be written as Eq. (17) will converge given a single bit flip in arithmetic. For example, if we split the matrix into its strict upper and lower triangles, $\mathbf{A} = -\mathbf{L} + \mathbf{D} - \mathbf{U}$, and let $\mathbf{G} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}$ and $\mathbf{f} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{b}$, then we obtain the Gauss-Seidel method, which will also converge given a bit flip in arithmetic. The general form presented in Eq. (17) defines a class of methods called stationary iterative methods. These types of methods are often used in various ways, for instance as preconditioners for other iterative methods or as smoothers in Multigrid methods. We did not present sampling results for the Jacobi and other stationary iterative methods because we have proven analytically that these methods will always converge.

VIII. Conclusion and Future Work

This work contributes an analysis of how a silent bit flip in floating point arithmetic impacts the elementary linear algebra constructs of dot products, present a rigorous analytic model and verify it using Monte Carlo sampling. A unique property of the IEEE-754 floating point specification is uncovered, namely that the bias used to store the exponent may be exploited to minimize the impact of a bit flip in the exponent. The result is subsequently extended from dot products to the composition of a basic iterative method consisting of a mat-vec operation.

In composing the Jacobi method, a rigorous proof was presented showing that stationary iterative methods will converge in spite of a single bit flip in arithmetic. It was further shown how to harden these methods such that they can tolerate bit flips producing non-numeric entries. Furthermore, a sample algorithm was developed that combines the results of this work to construct a version of Jacobi tolerating a silent bit flip in arithmetic, and, in some circumstances, converging to the correct solution in roughly the same number of iterations than without bit flips.

References

[1] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," in *SIGMETRICS Conference on Measurement and Modeling ofComputer Systems*, 2009, pp. 193–204.

[2] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *USENIX Conference on File and Storage Technologies*, 2007.

[3] J. Duell, "The design and implementation of berkeley lab's linux checkpoint/restart," Lawrence Berkeley National Laboratory, TR, 2000.

[4] C. Wang, F. Mueller, C. Engelmann, and S. Scott, "A job pause service under LAM/MPI+BLCR for transparent fault tolerance," in *International Parallel and Distributed Processing Symposium*, Apr. 2007.

[5] ——, "Proactive process-level live migration in hpc environments," in *Supercomputing*, 2008.

[6] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Supercomputing*, Nov. 2010.

[7] I. Philp, "Software failures and the road to a petaflop machine," in *Workshop on High Performance Computing Reliability Issues*. IEEE Computer Society, 2005.

[8] R. Riesen, K. Ferreira, D. D. Silva, P. Lemarinier, D. Arnold, and P. G. Bridges, "Alleviating scalability issues of checkpointing protocols," in *Supercomputing*, Nov. 2012.

[9] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann, "Mcrengine - a scalable checkpointing system using data-aware aggregation and compression," in *Supercomputing*, Nov. 2012.

[10] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. de Supinski, N. Maruyama, and S. Matsuoka, "Design and modeling of a non-blocking checkpointing system," in *Supercomputing*, Nov. 2012.

[11] K. Ferreira, J. Stearley, J. H. L. III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Supercomputing*, nov 2011.

[12] J. Elliot, K. Kharbas, D. Fiala, F. Mueller, C. Engelmann, and K. Ferreira, "Combining partial redundancy and checkpointing for HPC," in *International Conference on Distributed Computing Systems*, 2012.

[13] V. Sridharan and D. Liberty, "A study of dram failures in the field," in *Supercomputing*, Nov. 2012.

[14] A. Geist, "What is the monster in the closet?" Aug. 2011, invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking.

[15] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design," in *Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 111–122.

[16] E. N. M. E. et al., "System resilience at extreme scale," DARPA, Tech. Rep., 2008. [Online]. Available: http://institutes.lanl.gov/resilience/docs/IBM Mootaz White Paper System Resilience.pdf

[17] P. Kogge and et al., "ExaScale computing study: Technology challenges in achieving exascale systems," DARPA IPTO, Tech. Rep., 2008. [Online]. Available: http://users.ece.gatech.edu/ mrichard/-ExascaleComputingStudyReports/exascale_final-_report_100208.pdf

[18] F. Cappello and et al., "Toward exascale resilience," UIUC / INRIA Joint Laboratory on PetaScale Computing, Tech. Rep. TR-JLPC-09-01, Jun. 2009. [Online]. Available: http://institutes.lanl.gov/resilience/docs/Toward Exascale Resilience.pdf

[19] V. Howle and P. Hough, "The effects of soft errors on krylov methods," Feb. 2012, invited Talk. SIAM Parallel Processing.

[20] V. Howle, P. Hough, M. Heroux, and E. Durant, "Soft errors in linear solvers as integrated components of a simulation," Apr. 2010, invited Talk.

[21] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Characterizing the impact of soft errors on iterative methods in scientific computing," in *International Conference on Supercomputing*, 2011, pp. 152–161.

[22] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *International Conference on Supercomputing*, 2008, pp. 155–164.

[23] J. Sloan, R. Kumar, G. Bronevetsky, and T. Kolev, "Algorithmic approaches to low overhead fault detection for sparse linear algebra," *Dependable Systems and Networks*, 2012.

[24] K. Malkowski, P. Raghavan, and M. Kandemir, "Analyzing the soft error resilience of linear solvers on multicore multiprocessors," in *International Symposium on Parallel Distributed Processing*, Apr. 2010, pp. 1 –12.

[25] M. Hoemmen and M. A. Heroux, "Fault-tolerant iterative methods via selective reliability," http://www.sandia.gov/ maherou/docs/FTGMRES.pdf.

[26] D. Li, J. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *Supercomputing*, Nov. 2012.

[27] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[28] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," *SIGPLAN Not.*, vol. 47, no. 8, pp. 225–234, Feb. 2012.

[29] C. J. Anfinson and F. T. Luk, "Linear algebraic model of algorithm-based fault tolerance," *IEEE Transactions on Computers*, vol. 37, no. 12, pp. 1599–1604, 1988.

[30] A. Al-Yamani, N. Oh, and E. J. McCluskey, "Performance evaluation of checksum-based abft," in *Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2001)*, Oct. 2001, pp. 461–466.

[31] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham, "Algorithm-based fault tolerance on a hypercube multiprocessor," *Computers, IEEE Transactions on*, vol. 39, no. 9, pp. 1132–1145, 1990.

[32] Y. Kim, J. S. Plank, and J. J. Dongarra, "Fault tolerant matrix operations using checksum and reverse computation," in *Symposium on the Frontiers of Massively Parallel Computing*, Oct. 1996, pp. 70–77.

[33] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen, "High performance linpack benchmark: A fault tolerant implementation without checkpointing," in *International Conference on Supercomputing*, May 2011, pp. 162–171.

[34] Z. Chen, "Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments," in *International Parallel and Distributed Processing Symposium*, Apr. 2008.

[35] ——, "Algorithm-based recovery for iterative methods without checkpointing," in *Symposium on High-Performance Parallel and Distributed Computing*, Jun. 2011, pp. 73–84.

[36] O. Axelsson, *Iterative solution methods*. Cambridge University Press, 1994.

[37] B. Randell, "System structure for software fault tolerance," *SIGPLAN Not.*, vol. 10, no. 6, pp. 437–449, Apr. 1975.

[38] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: A scalable, efficient, and flexible resiliency scheme for exascale systems," in *Supercomputing*, Nov. 2012.