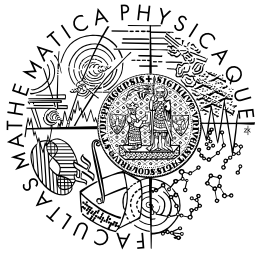

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Jakub Hercík

Detection and Correction of Silent Errors in Pipelined Krylov Subspace Methods

Department of Numerical Mathematics

Supervisor of the master thesis: Erin Claire Carson, Ph.D.
Study programme: Mathematics
Study branch: Computational Mathematics

Prague 2024

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to express my gratitude to my supervisor, Erin Claire Carson, Ph.D., for her invaluable guidance and help as well as to doc. RNDr. Petr Tichý, Ph.D., for being a consultant for this thesis.

Title: Detection and Correction of Silent Errors in Pipelined Krylov Subspace Methods

Author: Bc. Jakub Hercík

Department: Department of Numerical Mathematics

Supervisor: Erin Claire Carson, Ph.D., Department of Numerical Mathematics

Abstract: This thesis focuses on the problem of silent error detection in the pipelined predict-and-recompute conjugate gradient (Pipe-PR-CG) algorithm, a pipelined Krylov subspace method for solving linear systems with a symmetric positive definite matrix. The theory of silent errors and conjugate gradient variants is introduced, and the structure of Pipe-PR-CG is subsequently utilized in rounding error analysis to derive criteria for silent error detection based on bounds of several quantities computed in finite precision arithmetic. The efficacy of the criteria is then tested in a robust numerical experiment, and a fault-tolerant version of the algorithm is introduced. Additionally, the sensitivity of Pipe-PR-CG to silent errors is examined. Codes in the Python programming language which were used for the main experiments and figures presented in this thesis are also provided.

Keywords: fault tolerance, iterative methods, computer science, numerical mathematics, errors, algorithms, high-performance computing, matrix computations, Krylov subspace methods

Contents

Introduction

As computational machines are becoming increasingly large and more complex, and with that rises the probability of failure, the topic of error handling and fault-tolerant algorithms is now more important than ever. Some errors are rather simple to detect, since they, e.g., result in a crash of the computation. However, there is another category of faults, so-called "silent errors". These faults may not be immediately easily apparent, but they can result in significantly altered behavior of the used numerical method. One way of solving the problem of their detection is to perform the computation multiple times, and compare the results. However, this substantially increases the overall computational cost, in terms of both time and energy. Therefore, it is desirable to possess an alternative, more efficient approach.

This thesis examines the subject of silent error detection in the context of pipelined Krylov subspace methods for the solution of linear systems. Pipelined Krylov subspace methods are a class of algorithms designed to be computationally efficient on massively parallel computer architectures. This is achieved by a rearrangement of mathematical expressions in the given method, so that there is less need for communication, i.e., data movement, and certain steps can be overlapped. The focus of this work is a specific variant of the well-known conjugate gradient method, the pipelined predict-and-recompute conjugate gradient algorithm, shortly, Pipe-PR-CG, first introduced in the article "Predict-and-Recompute Conjugate Gradient Variants" by Tyler Chen and Erin Carson in 2020.

The ultimate goal of this thesis is to derive effective and reasonably inexpensive methods for silent error detection in Pipe-PR-CG that can be subsequently incorporated into a modified version of the algorithm which is automatically able the detect and correct the silent faults. These detection methods are based on the comparison of "gaps" between certain quantities which are equal in exact arithmetic and the bounds on their values in finite precision.

The first two chapters contain a summary of the theoretical background concerning the concepts of floating-point arithmetic, silent errors, and various conjugate gradient variants. The third chapter examines the sensitivity of Pipe-PR-CG to silent errors. The fourth chapter then focuses on constructing several criteria for the detection of silent errors in the Pipe-PR-CG algorithm. This involves a mathematical derivation of the necessary relations as well as a demonstration of how the derived expressions behave in the case of a bit flip occurrence. The work is concluded by a numerical experiment investigating the detection performance of the derived criteria for bit flips in all Pipe-PR-CG variables and, finally, a statement of a fault-tolerant modification of the original algorithm. Presented is also an adaptive version of the fault-tolerant algorithm able to dramatically reduce the amount of falsely positive detections.

1. Theoretical background

This chapter focuses on providing theoretical background for various topics from computer science. In the beginning, a short introduction to basic terminology of parallel computing is given. Then, the concept of computer arithmetic and finite precision computations is introduced. In the final part, we delve into the theory and practical matters of silent errors.

1.1 Pipelining and basic parallel terminology

Before we proceed to issues of silent errors and Krylov subspace methods let us explain some terminology of computational science, which is mentioned later in this thesis. However, please note that these concepts are much more complex and contain richness beyond the information presented here. For a gentle introduction into this field please see, e.g., the "Introduction to Parallel Computing Tutorial" [2] assembled by the Lawrence Livermore National Laboratory, which the next paragraphs greatly draw from. It is also possible to find further references and resources there. Now, let us present some terminology in the form of a list.

- *A Task* is a part of the overall computational work that is performed by a processor. During parallel computation multiple tasks are concurrently performed by multiple processors [2].

- *Pipelining* is a type of parallel computing, where our potentially partly sequential work is divided into separate parts, each of which is computed by a different computational unit. The input data go through the pipeline, step by step, like in, e.g., a car assembly line [2]. The advantage of this is that we are able to overlap the operations, and consequently process the data faster.

- *Distributed memory* refers to a model of hardware organization, where physical memory is distributed among computational units. Each task can directly access only data present on the processor it runs on. If it needs data stored in other processors' memory, some form of communication must be performed [2].

- *Communication* is the process of data exchange between tasks [2]. This term can encompass both sequential communication (moving data between levels in the memory hierarchy) and parallel communication (moving data between distributed memory elements). Aside from the obvious "algebraic" cost, each computation also involves some degree of communication costs. This cost can dominate the overall runtime of the computation. Therefore, it is highly desirable to limit the necessary amount of communication in an algorithm, especially if we aim to run it in parallel [3].

- *Synchronization* is the real-time coordination of our parallel tasks. It often involves slowdown of the overall computation, since there is generally the need to wait for one or more tasks to reach some point in the computation [2].

- *Reduction* is a type of collective (involving more than two tasks) parallel communication, which combines data from different computational units into some combined value on one specific unit using a certain operator (e.g., sum, multiplication) [2] [4]. This work will involve the discussion of reductions of inner products, i.e., summing local sums computed from parts of the vectors distributed among multiple computational units into the overall result.

1.2 Finite precision arithmetic

Every computation performed on a computer is subject to so called *finite precision arithmetic*. It is a system designed to represent numbers and to calculate with them. Ideally, we would be able to compute everything using infinite precision arithmetic, i.e., the real numbers, however, that is not possible, since computers are finite machines able to work with no more than finite information [5]. Therefore, we ought to operate with some finite set $\mathbb{F}$ and every $a \in \mathbb{R}$ is to be approximated by some $\hat{a} \in \mathbb{F}$ [5]. There are several ways how to construct this set. The one most commonly used is called *floating-point arithmetic*, where the elements of the set $\mathbb{F}$ are expressed utilizing a *base* $\mathbf{B} \in \mathbb{N} \setminus \{1\}$, an *exponent* $\mathbf{e}$, and a *mantissa* $\mathbf{m}$ [6]. Taking an arbitrary number $\hat{a} \in \mathbb{F}$, we can represented it as [5]:

$$\hat{a} = \pm B^e \times m,$$

where

$$m = d.d \dots d,$$

$$e = d \dots d,$$

$$\text{for } d \in \{0, \dots, B-1\}.$$

This notation was used, e.g., in [7]. Having established how to construct sets $\mathbb{F}$, the next logical step could be to choose one of them, and make it a standard as this would greatly increase software portability. The system currently mostly used is called "the IEEE standard" [5].

Citing from "Scientific Computing: An Introduction using Maple and MATLAB" [5]: *"Since 1985 we have for computer hardware the ANSI/IEEE Standard 754 for Floating Point Numbers. It has been adopted by almost all computer manufacturers. The base is $B = 2$."*

There exist several so called "precisions", which differ by the number of bits allocated for each element of $\mathbb{F}$, respectively, for its exponent and mantissa. We also need to reserve one bit for the sign of the number. The most commonly used precisions are the following. Half precision - *16 bits*, single precision - *32 bits*, double precision - *64 bits* and quadruple precision - *128 bits*. The following Table 1.1 summarizes the main characteristics of each of these formats [8][9]. The machine ϵ is defined as spacing of the elements of $\mathbb{F}$ between 1 and the base $\mathbf{B}$. The symbol $\hat{a}_{\max}$ denotes the largest representable number in the specific precision format [5].

	Half precision	Single precision	Double precision	Quadruple precision
Total bits	16	32	64	128
Sign bits	1	1	1	1
Exponent bits	5	8	11	15
Mantissa bits	10	23	52	112
Machine ϵ (in decimal system)	$2^{-10} \approx 9.77 \times 10^{-4}$	$2^{-23} \approx 1.19 \times 10^{-7}$	$2^{-52} \approx 2.22e \times 10^{-16}$	$2^{-112} \approx 1.93 \times 10^{-34}$
$\hat{a}_{\max}$	$\approx 10^5$	$\approx 10^{38}$	$\approx 10^{308}$	$\approx 10^{4932}$

Table 1.1: Characterizations of IEEE Standard precisions

We are going to be primarily interested in IEEE double precision, since it is the system used in all numerical experiments included in this thesis. On top of that, it is the most commonly used one in general [5].

In IEEE double precision, the first bit represents the sign, then there are 11 bits for the exponent, and the remaining 52 bits are utilized for the mantissa [5]. The value of some $\hat{a} \in \mathbb{F}$ is then represented in the following way [5]:

- **Normal numbers**: In the case that $0 < e < 2047$, we have $\hat{a} = (-1)^s \times 2^{e-1023} \times 1.m$, where we prefix the mantissa m with a binary point and an implicit 1 [5].
- **Subnormal numbers**: In the case that $e = 0$ but $m \neq 0$, we have $\hat{a} = (-1)^s \times 2^{e-1022} \times 0.m$, which is so-called "denormalized number", as the mantissa is not prefixed by an implicit 1 but 0 instead [5]. Moreover:
  - For $e = 0$, $m = 0$, and $s = 0$, it is $\hat{a} = 0$.
  - For $e = 0$, $m = 0$, and $s = 1$, it is $\hat{a} = -0$.
- **Exceptions**: In the case that $e = 2047$, we have [5]:
  - If $m = 0$ and $s = 0$, then $\hat{a} =$ `Inf`.
  - If $m = 0$ and $s = 1$, then $\hat{a} =$ `-Inf`.
  - If $m \neq 0$, then $\hat{a} =$ `NaN` (Not a Number).
- As was mentioned above, the machine epsilon, which is defined as spacing of the elements of $\mathbb{F}$ between 1 and the base $\mathbf{B}$ is for IEEE double precision $\epsilon = 2^{-52} \approx 2.220446049250313 \times 10^{-16}$ [5].
- The largest representable number is $\hat{a}_{max} \approx 1.7976931348623157 \times 10^{308}$. If the result of an operation falls outside the interval $[-\hat{a}_{max}, \hat{a}_{max}]$, we call it an "overflow" [5]. Different programming languages and compilers react to this differently, e.g., Matlab deems such a result as $\pm$`Inf`, while Python might raise an overflow error instead. This is going to be important later in the thesis, as all included experiments are implemented in Python, and we have to account for the occasional overflow errors.

- The smallest representable normalized number (in terms of proximity to zero) is $\pm\hat{a}_{min} \pm 2^{-1022}$. Below this threshold, denormalized numbers are used. They fall in the range $[\epsilon \times \hat{a}_{min}, \hat{a}_{min})$ and its negative counterpart respectively. Numbers smaller in absolute value than $\epsilon \times \hat{a}_{min}$ are said to be in the "underflow range". Once a value gets into this range, it can no longer be represented, and it is deemed as a zero instead [5].

Arithmetic operations also abide by certain rules. Having $\hat{a}, \hat{b} \in \mathbb{F}$ and finite precision arithmetic operator $\star$ (implementation of $+, -, \times, /$), the result of the operation $\hat{a} \star \hat{b}$ will most likely not be an element of $\mathbb{F}$, as this set is rather sparse in $\mathbb{R}$. Instead, it holds that $\hat{a} \star \hat{b} = (\hat{a} * \hat{b})(1 + e)$, where $*$ is the operator in exact arithmetic and e is such that $|e| < \epsilon$. We can also think of finite precision operations as exact arithmetic operations applied to some $\tilde{a}, \tilde{b}$ close to $\hat{a}$ and $\hat{b}$ respectively [5]. This is a similar notion to backward error analysis where one aims to find some perturbed data for which the outcome of an imprecise computation is the precise result [10]. The difference between the exact result $(\hat{a} * \hat{b})$ and the finitely computed result $\hat{a} \star \hat{b}$ is called the "rounding error" [5].

Finite precision arithmetic often has a negative impact on computations due to these errors. For instance, given a symmetric positive definite matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, it should take the conjugate gradient method at most n iterations to converge. In practice, however, it can take many more [10].

1.3 Silent errors

A key concept of this thesis are so-called "silent errors", also known as "soft faults" [11] or "silent data corruption" (SDC) [12]. This section presents the concept of silent errors and tries to motivate why uncovering and fixing them is important.

As was mentioned in the previous section, every computation is subject to finite precision arithmetic, and therefore to rounding errors. However, there are other problems which might arise, be it because of code implementation, hardware defects, or some additional issues. *Silent errors* are faults that do not terminate the computation or raise an error, but rather cause a change in some floating-point number without any apparent indication of a problem [11].

Silent errors may be qualified in several ways from both the low-level point of view (what is their hardware-wise cause) and the high-level point of view (in what way they affect the logical values involved in a computation). At the high-level, a distinction can be made based on whether the silent error permanently affects only the result of the operation or the input quantities as well [13]. The following explanation of this is adapted from the article "On Soft Errors in the Conjugate Gradient Method: Sensitivity and Robust Numerical Detection" [13].

Let us assume there exist floating-point numbers a, b, c such that without any silent error it holds that $a + b = c$. Now, let a get altered $(a \to \tilde{a})$ during this operation by a silent error. This causes c to be altered as well, resulting in some $\tilde{c} = \tilde{a} + b$. Silent errors then can be divided into two categories: *persistent* and *transient* [13].

1. *Transient silent errors* are faults for which only the result $\tilde{c}$ stays altered after the calculation is concluded, meaning, the input variable $\tilde{a}$ "reverts"

back to a. Therefore, the resulting variables stored in memory are $a, b, \tilde{c}$. This can happen if the variable a is altered by the silent error while being in *transient* memory, for instance, cache [13].

2. *Persistent silent errors* are faults for which, in the end, not only the result c stays influenced but the input data a as well. In this case, the resulting quantities after the calculation are $\tilde{a}, b$, and $\tilde{c}$. This can happen if a is altered while being in *persistent* memory, e.g, the main memory [13].

The experiments of this thesis assume the model of *transient* silent errors. The same is done, e.g., in [12] and [13].

Generally speaking, the origin of a silent error need not be known for the purpose of studying its impacts, as it provides no additional information to this end [12]. In the article "Evaluating the Impact of SDC on the GMRES Iterative Solver" [12] the authors argue for this approach as opposed to studying specifically the impact of *bit flips* as one possible origin of silent errors.

A bit flip is an event when a value of a bit in the representation of a floating-point number is reversed, i.e., 0 becomes 1 or 1 becomes 0. For instance, a bit flip occurring in the first bit - the sign - results in a floating point number $\hat{a}$ becoming $-\hat{a}$. Although, as was mentioned above, a general approach to silent errors as a category is possible, this thesis is concerned primarily with bit flips (in the context of numerical linear algebra), since they are commonly studied [12] and easy to model.

The impact of a bit flip may vary. A bit flip in the end of the mantissa may have only negligible effects, whereas a flip of some dominant bit in the exponent can destroy the entire computation. There are several possible situations a bit flip may cause, including, but not necessarily limited to, the following [12]:

- *termination* of the computation, e.g., because of a floating-point overflow;
- *stagnation*, i.e., the stopping criteria are never reached;
- *delay* in convergence;
- *convergence to an incorrect result*, e.g., for an iterative solver, a bit flip in the norm of the residual vector r_k might result in the algorithm prematurely reaching the stopping criteria with a result which does not truly satisfy them;
- *nothing of concern* for the convergence.

As can be seen, these effects can impact the computation in a number of different ways. If the algorithm is terminated or it stagnates we are at least given an indication that something unexpected has happened (because of this, it is, purely speaking, no longer a "silent" error). However, in the case of convergence to an "incorrect" result, there are at first glance no signs that something went wrong. This motivates why the topic of silent errors might be potentially crucial in practice. Especially, as the supercomputers the modern parallel codes run on are becoming rather complex and the number of their parts increases, the risk of a hardware failure increases as well [11]. Moreover, the decrease of transistor feature sizes makes individual components more prone to failure [12].

An interesting observation is the fact that bit flips "from 0 to 1" and "from 1 to 0" are not equally influential in terms of the relative perturbation of the altered floating-point number [13]. Here, the term "relative perturbation" means the ratio $|\tilde{x} - x|/|x|$, where x is the original number and $\tilde{x}$ the changed number after a bit flip.

The following Table 1.2 adapted from the article "On Soft Errors in the Conjugate Gradient Method: Sensitivity and Robust Numerical Detection" [13] depicts values of relative perturbations and their respective bounds for all possible bit flips in IEEE double precision. The bits are numbered from 1 to 64 with the order being: the one sign bit, the twelve exponent bits (with decreasing "importance"), and the fifty two mantissa bits (again, with decreasing "importance"). The constant m is the value of the mantissa (before the bit flip), as it was presented in the section concerning floating-point numbers. Therefore, if we let b_i denote whether the respective bit is 0 or 1, it holds that [13]

$$m = \sum_{i=13}^{64} b_i 2^{12-i}, \quad b_i \in \{0, 1\}.$$

Derivation of the values and their respective bounds (in red) presented in the table can be found in [13], Appendix A.

bit flip type	$i = 1$ (sign)	$2 \leq i \leq 12$ (exponent)	$13 \leq i \leq 64$ (mantissa)
$0 \xrightarrow{\text{flip}} 1$	2	$2^{2^{12-i}}$	$\frac{2^{12-i}}{1+m} \leq \frac{1}{4}$
$1 \xrightarrow{\text{flip}} 0$	2	$\frac{1}{2} \leq 1 - 2^{-2^{12-i}} \leq 1$	$\frac{\lvert -2^{12-i} \rvert}{1+m} \leq \frac{1}{2}$

Table 1.2: Relative perturbations $|\tilde{x} - x|/|x|$ when the i-th bit is flipped in the number x ($x \xrightarrow{\text{flip effect}} \tilde{x}$)

As can be observed from inspecting Table 1.2, the difference in the "type" of the flip is potentially extremely significant for the exponent bits ($2 \leq i \leq 12$). However, it is necessary to keep in mind that this is caused by the fact that the change is being measured in the relative sense of the fraction $|\tilde{x} - x|/|x|$. In the case of a $1 \to 0$ flip, the original value x which is the denominator is larger than the $\tilde{x}$, and thus the ratio is "controlled". Whereas, for a $0 \to 1$ flip, it is the other way around and it may be that the original value x is quite small compared to $\tilde{x}$. Nonetheless, this variability makes it a potentially interesting quantitative factor for future research.

The question now is: How can we recognize that a silent error has occurred? The most straightforward approaches for detecting silent errors are so-called *double modular redundancy* (DMR) and *triple modular redundancy* (TMR) [13]. These methods are based on the idea that we can perform the same computation multiple times, either consecutively on the same hardware unit or simultaneously on different hardware units, and then check whether the results are the same [13]. There might be an objection that it is possible for the exact same silent error to happen multiple times, thus rendering this approach unreliable. However,

it can be argued that silent errors are relatively infrequent events, because, e.g., of multiple precautions build in the hardware to minimize the likelihood of them [12]. Nevertheless, the crucial problem of redundancy approaches is their cost, as they require either multiple computational units or twice/thrice the time. This is especially limiting for large, massively parallel computers because of their energy consumption [12].

Another possible idea for the detection of silent errors is to use information about the numerical method to derive some detection criteria [11]. This approach is called *algorithm-based fault tolerance* (ABFT) [11]. For instance, it is possible to monitor residual norms or to set up bounds for some quantities [11]. In the fourth chapter of this thesis, such bounds are derived for the Pipe-PR-CG algorithm based primarily on monitoring differences between "predicted" and "recomputed" values of two of its variables. Even though ABFT methods do not require the amount of computational resources needed for the redundancy approaches, they still require some. They may also cause delayed convergence if the algorithm is modified to utilize them to correct the found errors during the computation [11].

2. Conjugate gradient and its variants

2.1 Krylov subspace methods

The concept of Krylov subpaces dates back to 1931, when it was introduced by a Russian naval officer and marine scientist Aleksei Nikolaevich Krylov with the aim of analyzing oscillations of mechanical systems via minimal polynomials of matrices [14]. Subsequently, 21 years later, a method utilizing Krylov subspaces for the solution of linear systems with a symmetric positive definite matrix was proposed by Hestenes and Stiefel in their article "Methods of Conjugate Gradients for Solving Linear Systems" [15].

The definition of a Krylov subspace is as follows. Given a matrix $A \in \mathbb{R}^{n \times n}$, a vector $v \in \mathbb{R}^n$, and an integer $k \leq n$, we define the k-dimensional Krylov subspace of the matrix A and the vector v as $\mathcal{K}_k(A, v) \coloneqq span\{v, Av, A^2v, \ldots, A^{k-1}v\}$ [10]. This notion can be naturally generalized to $\mathbb{C}$.

Krylov subspace methods for solving linear algebraic systems are a special case of so-called projection methods. These algorithms construct a sequence of approximations $\{x_k\}_{k=0}^n$ (where k denotes the index of the current iteration as well as dimension of spaces $\mathcal{C}_k, \mathcal{S}_k$ described below) from the initial guess x_0, such that

$$x_k \in x_0 + \mathcal{S}_k, \quad r_k \perp \mathcal{C}_k.$$

The space $\mathcal{S}_k$ used for selecting a set of possible choices for the approximation is called the *Search space* and the space $\mathcal{C}_k$ used for selecting the "best" approximation is called the *Constraint space*. Please note that the meaning of the term "best" differs depending on the particular method. Many different methods can be derived based on the selection of $\mathcal{C}_k$ and $\mathcal{S}_k$ [10].

However, the main focus of this thesis is the conjugate gradient method.

2.2 Conjugate gradient

As was previously mentioned, the original conjugate gradient method (from now on also denoted as HS-CG) for solving linear systems with a symmetric positive definite matrix was first formulated in 1952 by Hestenes and Stiefel in their article [15]. There are several ways how to derive it, e.g., via minimization of a certain quadratic functional or from the Lanczos algorithm [10]. Naturally, it is also a projection method with $\mathcal{S}_k = \mathcal{C}_k = \mathcal{K}_k(A, r_0)$, which minimizes the *energy norm* of the error $||x - x_k||_A$ [10]. The norm is well-defined owing to A being symmetric positive definite, and thus it can be easily proven that it satisfies the necessary conditions to define an inner product. The procedure of the HS-CG algorithm below is, including notation, formulated in the same way as in [1]. From now on, vector variables are written in bold and matrices in bold uppercase. $\mathbf{M}$ denotes a symmetric positive definite "preconditioner" matrix, which is used to improve properties of the system. Using $\mathbf{M}$, we can implicitly solve the system $\mathbf{L^{-T}AL^{-1}y} = \mathbf{L^{-T}b}$, where $\mathbf{y} = \mathbf{Lx}$ and $\mathbf{L}$ is the Cholesky factor of $\mathbf{M}$, utilizing

just solutions of subsystems with $\mathbf{M}$ during the run, e.g., $\tilde{\mathbf{r}}_\mathbf{k} = \mathbf{M}^{-1}\mathbf{r}_\mathbf{k}$ [1]. The symbol $\sim$ denotes extra variables introduced by inclusion of the preconditioner.

A more extensive description of classical CG is beyond the scope of this thesis. In case the reader is interested in this, further information can be found for instance in the book "Krylov Subspace Methods: Principles and Analysis" by Jorg Liesen and Zdeněk Strakoš [16], which contains deep analysis of CG and demonstrates its relation to other mathematical concepts and methods.

The statement of the INITIALIZE() procedure for the following as well as for all other algorithms can be found at the very end of this thesis in Appendix A.1.

Algorithm 1 Hestenes and Stiefel Conjugate Gradient: HS-CG (preconditioned)

1: **procedure** HS-CG($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
2: INITIALIZE()
3: **for** $k = 1, 2, \ldots$ **do**
4: $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_{k-1}\mathbf{p_{k-1}}$
5: $\mathbf{r_k} = \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}},\ \tilde{\mathbf{r}}_\mathbf{k} = \mathbf{M}^{-1}\mathbf{r_k}$
6: $\nu_k = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{r_k} \rangle$
7: $\beta_k = \nu_k / \nu_{k-1}$
8: $\mathbf{p_k} = \tilde{\mathbf{r}}_\mathbf{k} + \beta_k \mathbf{p_{k-1}}$
9: $\mathbf{s_k} = \mathbf{A}\mathbf{p_k}$
10: $\mu_k = \langle \mathbf{p_k}, \mathbf{s_k} \rangle$
11: $\alpha_k = \nu_k / \mu_k$
12: **end for**
13: **end procedure**

Taking a look at Algorithm 1, we can observe that the computation has to be done largely sequentially, since each step directly depends on variables computed in the previous ones. This causes trouble on parallel distributed memory computers, where a communication bottleneck is created because of the inability to overlap computation with expensive global reductions from inner products or with at least some amount of communication from the matrix-vector multiplication [1]. Specifically, there are two so-called global synchronization points at the inner products [17]. Citing from the article "The Numerical Stability Analysis of Pipelined Conjugate Gradient Methods: Historical Context and Methodology" [17]: *"Computing each inner-product requires a global synchronization point; i.e., the computation can not proceed until all processors have finished their local computation and communicated the result to other processors. It is well-known that for large-scale sparse problems on large-scale machines, the cost of synchronization between parallel processors can dominate the run-time."*

The structure of the HS-CG algorithm is illustrated in the diagram (Figure 2.1) below. As was mentioned, there are two global synchronization points - the inner products - in each iteration. The diagram follows the order of operations in the algorithm. First, there are vector updates (lines 4 and 5), then the first inner product (line 6) and its subsequent utilization in a scalar update (line 7), after that, a vector update (line 8) followed by a matrix-vector multiplication (line 9), and finally the second inner product (line 10) and its usage in a scalar update (line 11). For simplicity, neither this nor any other iteration diagrams in this chapter take into account the preconditioning steps.

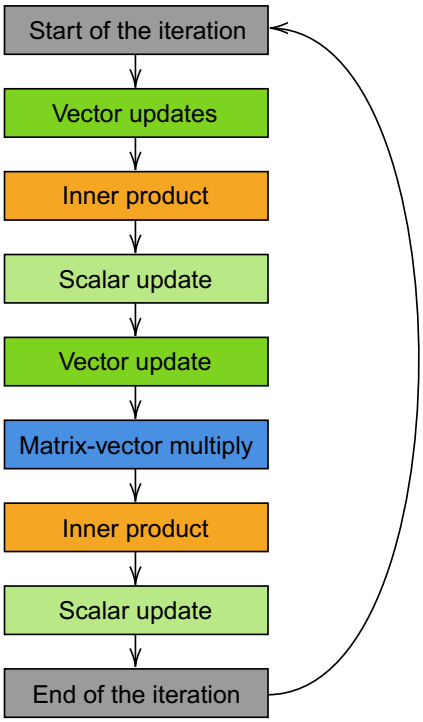

Figure 2.1: Iteration diagram of HS-CG

2.3 Communication-hiding conjugate gradient variants

The high cost of communication in HS-CG motivates the search for mathematically equivalent CG variants better suited for implementation on parallel machines, as with less communication we might be able to speed up the overall computation. One possibility is to rearrange the procedure in a way that reduces the necessary number of synchronization points to only one. However, such *communication-hiding* variants may amplify the numerical problems which already exist in HS-CG such as delayed convergence, or may worsen the maximal attainable accuracy (the level at which the approximation error $||\mathbf{x} - \mathbf{x_k}||$ starts to stagnate [10]) due to the sensitivity of CG to rounding errors [1].

On top of that, we would like to "pipeline" the inner product reductions and matrix operations to occur concurrently, so that the global synchronization points no longer cause a bottleneck. This is explored in Section 2.4. Nonetheless, once we try to include *pipelining*, the above-mentioned issues can become even more grave. In order to modify the recurrences, so that some computations can be performed simultaneously, additional variables are added to the CG algorithm. This can cause further amplification of the numerical rounding errors [1].

To better understand the resulting overall structure of iterations in the *communication-hiding* and *pipelined* variants, it is possible to inspect Figure 2.2 (page 14) and Figure 2.3 (page 20). Although these diagrams are made to describe structures of specific variants (PR-CG and Pipe-PR-CG), they reflect the general idea behind communication hiding and pipelining as well.

2.3.1 PR-CG

Now we shall very closely follow the article "Predict-and-Recompute Conjugate Gradient Variants" by Tyler Chen and Erin Carson [1] in their derivation of a communication-hiding CG variant, which requires only one global synchronization point per iteration. This is done by deriving a mathematically equivalent expression for the variable ν_k (the 6th line in Algorithm 1), utilizing quantities already computed in the previous iteration. This way we avoid the first computation of the inner product $\nu_k = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{r}_\mathbf{k} \rangle$. However, this change could lead to a dramatic loss of accuracy as the value of ν_k could become negative [1], and therefore we employ the new expression just as a "predicted" value, which is then used instead of the original expression to compute the next few steps until it is "recomputed" using the original above written inner product. This idea was proposed by Gérard Meurant in his article "Multitasking the conjugate gradient method on the CRAY X-MP/48" [18] in 1987 to stabilize the algorithm while also retaining the potential for parallelism. Although even this alteration might introduce some instability to the algorithm, it allows us to perform the iteration more efficiently on distributed memory machines as there is less need for data communication, since all the inner products occur at the same time [1]. Moreover, the maximal attainable accuracy is similar as for the original HS-CG. This perhaps somehow surprising result is thoroughly analyzed in [1].

First, as it is done in [1], we realize that by substituting for the $\mathbf{r}_k = \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}}$ and setting $\tilde{\mathbf{s}}_\mathbf{k} \coloneqq \mathbf{M^{-1}}\mathbf{s_k}$, the expression $\tilde{\mathbf{r}}_\mathbf{k} = \mathbf{M^{-1}}\mathbf{r_k}$ can be rewritten as follows [1]:

$$\tilde{\mathbf{r}}_\mathbf{k} = \tilde{\mathbf{r}}_\mathbf{k-1} - \alpha_{k-1}\tilde{\mathbf{s}}_\mathbf{k-1}.$$

Then, we substitute for $\tilde{\mathbf{r}}_k$ and $\mathbf{r_k}$ to obtain the relation

$$\nu_k = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{r_k} \rangle = \langle \tilde{\mathbf{r}}_\mathbf{k-1} - \alpha_{k-1}\tilde{\mathbf{s}}_\mathbf{k-1}, \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}} \rangle = \\ \langle \tilde{\mathbf{r}}_\mathbf{k-1}, \mathbf{r_{k-1}} \rangle - \alpha_{k-1}\langle \tilde{\mathbf{r}}_\mathbf{k-1}, \mathbf{s_{k-1}} \rangle - \alpha_{k-1}\langle \tilde{\mathbf{s}}_\mathbf{k-1}, \mathbf{r_{k-1}} \rangle + \alpha_{k-1}^2 \langle \tilde{\mathbf{s}}_\mathbf{k-1}, \mathbf{s_{k-1}} \rangle$$

which can be simplified, as it holds that

$$\langle \tilde{\mathbf{r}}_\mathbf{k-1}, \mathbf{s_{k-1}} \rangle = \langle \mathbf{M^{-1}}\mathbf{r_{k-1}}, \mathbf{s_{k-1}} \rangle = \langle \mathbf{r_{k-1}}, \mathbf{M^{-T}}\mathbf{s_{k-1}} \rangle = \langle \tilde{\mathbf{s}}_\mathbf{k-1}, \mathbf{r_{k-1}} \rangle,$$

since the preconditioner matrix $\mathbf{M}$ is required to be *symmetric* positive definite, and therefore also $\mathbf{M^{-1}} = \mathbf{M^{-T}}$. Thus

$$\nu_k = \langle \tilde{\mathbf{r}}_\mathbf{k-1}, \mathbf{r_{k-1}} \rangle - 2\alpha_{k-1}\langle \tilde{\mathbf{r}}_\mathbf{k-1}, \mathbf{s_{k-1}} \rangle + \alpha_{k-1}^2 \langle \tilde{\mathbf{s}}_\mathbf{k-1}, \mathbf{s_{k-1}} \rangle.$$

Furthermore, by setting

$$\sigma_k \coloneqq \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{s_k} \rangle, \quad \gamma_k \coloneqq \langle \tilde{\mathbf{s}}_\mathbf{k}, \mathbf{s_k} \rangle,$$

we can write the "predicted" value as

$$\nu'_k \coloneqq \nu_{k-1} - 2\alpha_{k-1}\sigma_{k-1} + \alpha_{k-1}^2\gamma_{k-1}.$$

The above inner products σ_k and γ_k can occur simultaneously as the "recomputing" of ν_k and the computation of μ_k, provided we have already computed $\mathbf{s_k}$ and $\tilde{\mathbf{s}}_\mathbf{k}$ [1].

Synthesis of these ideas leads to a predict-and-recompute variant of CG in Algorithm 2 below. Once again, the steps of the procedure are formulated and ordered the same as in [1].

Algorithm 2 Predict-and-Recompute Conjugate Gradient:
PR-CG (preconditioned)

1: **procedure** PR-CG($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
2: INITIALIZE()
3: **for** $k = 1, 2, \ldots$ **do**
4: $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_{k-1}\mathbf{p_{k-1}}$
5: $\mathbf{r_k} = \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}},\ \tilde{\mathbf{r}}_\mathbf{k} = \tilde{\mathbf{r}}_\mathbf{k-1} - \alpha_{k-1}\tilde{\mathbf{s}}_\mathbf{k-1}$
6: $\nu'_k = \nu_{k-1} - 2\alpha_{k-1}\sigma_{k-1} + \alpha^2_{k-1}\gamma_{k-1}$
7: $\beta_k = \nu'_k/\nu_{k-1}$
8: $\mathbf{p_k} = \tilde{\mathbf{r}}_\mathbf{k} + \beta_k\mathbf{p_{k-1}}$
9: $\mathbf{s_k} = \mathbf{A}\mathbf{p_k},\ \tilde{\mathbf{s}}_\mathbf{k} = \mathbf{M}^{-1}\mathbf{s_k}$
10: $\mu_k = \langle \mathbf{p_k}, \mathbf{s_k} \rangle,\ \sigma_k = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{s_k} \rangle,\ \gamma_k = \langle \tilde{\mathbf{s}}_\mathbf{k}, \mathbf{s_k} \rangle,\ \nu_k = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{r_k} \rangle$
11: $\alpha_k = \nu_k/\mu_k$
12: **end for**
13: **end procedure**

Below is the iteration diagram of PR-CG. As can be seen, there is now only a single synchronization point as all the inner products can occur simultaneously. Although the diagram particularly follows PR-CG, the general idea of coupling inner products together by changing dependencies through rearrangement applies to all other communication-hiding variants as well.

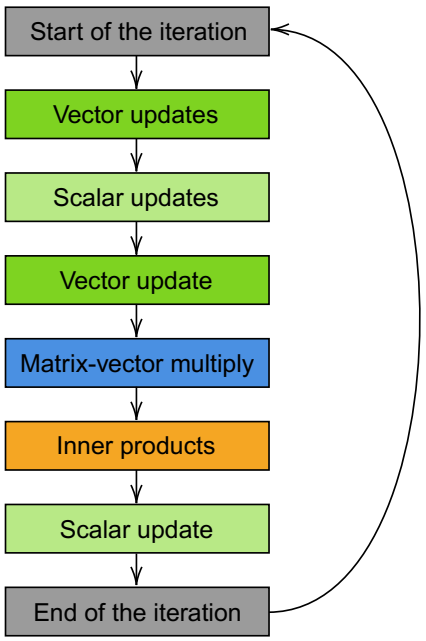

Figure 2.2: Iteration diagram of PR-CG

Different versions of the algorithm can be obtained by expressing the "predicted" value of ν_k in a different way. However, this particular expression seems to function a bit better than the alternatives (e.g., the M-CG described below), both in this form as well as once we compare the pipelined variants of the algorithms [1], which is what this thesis is mainly interested in. Other versions are discussed and compared for instance in [1] and [17].

2.3.2 M-CG

Let us shortly mention here one of the variants. Specifically, the version of conjugate gradient derived by Meurant in [18]. It is a predict-and-recompute version of the original HS-CG, similar to PR-CG. Their difference is that the version proposed by Meurant (here called "M-CG") employs a different expression for the predicted value ν'_k [1].

The algorithm was first introduced by Gérard Meurant in 1987 in the article "Multitasking the conjugate gradient method on the CRAY X-MP/48" [18] with the aim of increasing the level of parallelism in the conjugate gradient method, so that it would be more fitting for implementation on the computer in question. Let us now describe the way it is derived in the article from the standard HS-CG as it is formulated in Algorithm 1.

First, let us realize that, in infinite precision, for integers $i \neq j$ it holds that $\langle \mathbf{r_i}, \tilde{\mathbf{r}}_\mathbf{j} \rangle = 0$ [18]. The reason for this is as follows. Without loss of generality, let us assume that $i > j$, so that the "unpreconditioned" residual has greater index. If not, we can utilize the symmetric property of the preconditioner matrix $\mathbf{M}$, and shift it to the other side of the inner product to receive the desired situation. Next, we shall use line 9 of HS-CG as it is in Algorithm 1 to observe that $\langle \mathbf{r_i}, \tilde{\mathbf{r}}_\mathbf{j} \rangle = \langle \mathbf{r_i}, \mathbf{p_j} - \beta_k \mathbf{p_{j-1}} \rangle$. Now, we use the fact that for the conjugate gradient method the residual vector $\mathbf{r_i}$ is orthogonal to all vectors $\mathbf{p_j}$ for $j < i$ [10]. In our case, this condition is satisfied, so we have proven the desired orthogonal property.

Next, from line 5 of Algorithm 1 we can observe that [18]

$$\mathbf{r_k} - \mathbf{r_{k-1}} = -\alpha_{k-1} \mathbf{A p_{k-1}}. \tag{2.1}$$

Multiplying this by $\mathbf{M^{-1}}$ yields [18]

$$\tilde{\mathbf{r}}_\mathbf{k} - \tilde{\mathbf{r}}_\mathbf{k-1} = -\alpha_{k-1} \mathbf{M^{-1} A p_{k-1}}. \tag{2.2}$$

When we now multiply the respective hand sides of (2.1) and (2.2), by using the proven orthogonal property, we obtain [18]

$$\langle \mathbf{r_k}, \tilde{\mathbf{r}}_\mathbf{k} \rangle + \langle \mathbf{r_{k-1}}, \tilde{\mathbf{r}}_\mathbf{k-1} \rangle = \alpha_{k-1}^2 \langle \mathbf{M^{-1} A p_{k-1}}, \mathbf{A p_{k-1}} \rangle.$$

Utilizing our notation, we can rewrite this as

$$\nu_k = -\nu_{k-1} + \alpha_{k-1}^2 \gamma_{k-1}, \tag{2.3}$$

which can then be used for the predictor ν'_k as Meurant did in [18]. We shall call the algorithm utilizing this predict-and-recompute scheme M-CG (Meurant Conjugate Gradient). The procedure is given below in Algorithm 3.

Alternatively, the algorithm can be derived in the following way. In the PR-CG algorithm (line 6), we have that

$$\nu'_k = \nu_{k-1} - 2\alpha_{k-1}\sigma_{k-1} + \alpha_{k-1}^2\gamma_{k-1}.$$

This expression can be rearranged to obtain the predictor value used in M-CG [1]. First, let us rewrite it as [1]

$$\nu'_k = \nu_{k-1} - 2\alpha_{k-1}\sigma_{k-1} + \alpha_{k-1}^2\gamma_{k-1} = \nu_{k-1} - 2\alpha_{k-1}\langle \tilde{\mathbf{r}}_{\mathbf{k-1}}, \mathbf{s}_{\mathbf{k-1}}\rangle + \alpha_{k-1}^2\gamma_{k-1}.$$

Next, it holds that $\langle \tilde{\mathbf{r}}_{\mathbf{k}}, \mathbf{s}_{\mathbf{k}}\rangle = \langle \mathbf{p}_{\mathbf{k}} - \beta_k \mathbf{p}_{\mathbf{k-1}}, \mathbf{A}\mathbf{p}_{\mathbf{k}}\rangle = \langle \mathbf{p}_{\mathbf{k}}, \mathbf{s}_{\mathbf{k}}\rangle$, owing to the expressions for $\mathbf{p}_{\mathbf{k}}$ and $\mathbf{s}_{\mathbf{k}}$ from lines 8 and 9 of PR-CG and the $\mathbf{A}$-orthogonality between vectors $\mathbf{p}_{\mathbf{k}}$ and $\mathbf{p}_{\mathbf{k-1}}$ [1]. This allows us to rewrite the above expression as [1]

$$\nu'_k = \nu_{k-1} - 2\alpha_{k-1}\langle \mathbf{p}_{\mathbf{k-1}}, \mathbf{s}_{\mathbf{k-1}}\rangle + \alpha_{k-1}^2\gamma_{k-1} = \nu_{k-1} - 2\alpha_{k-1}\mu_{k-1} + \alpha_{k-1}^2\gamma_{k-1}.$$

We can further alter this using the relation $\alpha_k\mu_k = \nu_k$, which follows from $\alpha_k = \nu_k/\mu_k$ (line 5 of Algorithm 1), as [1]

$$\nu'_k = -\nu_{k-1} + \alpha_{k-1}^2\gamma_{k-1},$$

and thus we arrive at the same expression as we have in (2.3). A formulation of the M-CG algorithm can be found below, arranged in the same way as in [1]. As was previously stated, this procedure differs from PR-CG by using a different expression for computing the predicted value of ν_k.

M-CG uses $\nu'_k = -\nu_{k-1} + \alpha_{k-1}^2\gamma_{k-1}$, while PR-CG utilizes the version $\nu'_k = \nu_{k-1} - 2\alpha_{k-1}\sigma_{k-1} + \alpha_{k-1}^2\gamma_{k-1}$. More about M-CG can be found in the original article "Multitasking the conjugate gradient method on the CRAY X-MP/48" [18].

Algorithm 3 Meurant Conjugate Gradient: M-CG (preconditioned)

1: **procedure** M-CG($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
2: INITIALIZE()
3: **for** $k = 1, 2, \ldots$ **do**
4: $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_{k-1}\mathbf{p_{k-1}}$
5: $\mathbf{r_k} = \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}},\ \tilde{\mathbf{r}}_{\mathbf{k}} = \tilde{\mathbf{r}}_{\mathbf{k-1}} - \alpha_{k-1}\tilde{\mathbf{s}}_{\mathbf{k-1}}$
6: $\nu'_k = -\nu_{k-1} + \alpha_{k-1}^2\gamma_{k-1}$
7: $\beta_k = \nu'_k/\nu_{k-1}$
8: $\mathbf{p_k} = \tilde{\mathbf{r}}_{\mathbf{k}} + \beta_k\mathbf{p_{k-1}}$
9: $\mathbf{s_k} = \mathbf{A}\mathbf{p_k},\ \tilde{\mathbf{s}}_{\mathbf{k}} = \mathbf{M^{-1}}\mathbf{s_k}$
10: $\mu_k = \langle \mathbf{p_k}, \mathbf{s_k}\rangle,\ \gamma_k = \langle \tilde{\mathbf{s}}_{\mathbf{k}}, \mathbf{s_k}\rangle,\ \nu_k = \langle \tilde{\mathbf{r}}_{\mathbf{k}}, \mathbf{r_k}\rangle$
11: $\alpha_k = \nu_k/\mu_k$
12: **end for**
13: **end procedure**

In the article "Predict-and-Recompute Conjugate Gradient Variants" by Tyler Chen and Erin Carson [1] and its Appendix C, the authors discuss numerical differences between PR-CG and M-CG in finite precision arithmetic. If the reader is interested in this topic, more information and further references can be found there. For our purposes, it suffices to say that PR-CG seems to work a little bit better then M-CG in practice [1], as was previously stated in the section dedicated to PR-CG.

2.3.3 ChG-CG

Another communication-hiding variant worth at least a brief mention is the so-called ChG-CG first introduced by Chronopoulos and Gear in 1989 in their article "s-step iterative methods for symmetric linear systems" [19]. Once again, the original two synchronization points of HS-CG are reduced to only one by clever expression manipulation. The form of the ChC-CG algorithm presented here is once again, for the sake of consistency, formulated in the same way as in the article [1], Appendix D. It utilizes additional new variables $(\mathbf{w_k}, \eta_k)$ not appearing in the original article [19], to make the procedure more intelligible.

Let us now present how it is possible to derive ChC-CG from HS-CG. First, we shall rewrite the expression for $\mathbf{s_k}$, so that it no longer utilizes a matrix-vector multiplication. Using that $\mathbf{p_k} = \tilde{\mathbf{r}}_\mathbf{k} + \beta_k \mathbf{p_{k-1}}$ and introducing a new variable $\mathbf{w_k} \coloneqq \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k}$ yields

$$\mathbf{s_k} = \mathbf{A}\mathbf{p_k} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k} + \beta_k \mathbf{A}\mathbf{p_{k-1}} = \mathbf{w_k} + \beta_k \mathbf{s_{k-1}}.$$

Next, let us rewrite the expression for $\mu_k (= \langle \mathbf{p_k}, \mathbf{s_k} \rangle)$. Using this alternative form of $\mathbf{s_k}$, evaluating the inner product, and utilizing the knowledge that due to being symmetric positive definite, both the matrix $\mathbf{A}$ and the matrix $\mathbf{M}$ (represented by the preconditioning symbol $\sim$) can be shifted to the other side of inner product gives us

$$\mu_k = \langle \mathbf{p_k}, \mathbf{s_k} \rangle = \langle \mathbf{p_k}, \mathbf{A}\mathbf{p_k} \rangle = \langle \tilde{\mathbf{r}}_\mathbf{k} + \beta_k \mathbf{p_{k-1}}, \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k} + \beta_k \mathbf{A}\mathbf{p_{k-1}} \rangle =$$
$$\langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k} \rangle + \langle \tilde{\mathbf{r}}_\mathbf{k}, \beta_k \mathbf{A}\mathbf{p_{k-1}} \rangle + \langle \beta_k \mathbf{p_{k-1}}, \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k} \rangle + \beta_k^2 \langle \mathbf{p_{k-1}}, \mathbf{A}\mathbf{p_{k-1}} \rangle =$$
$$\langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k} \rangle + 2\langle \tilde{\mathbf{r}}_\mathbf{k}, \beta_k \mathbf{A}\mathbf{p_{k-1}} \rangle + \beta_k^2 \langle \mathbf{p_{k-1}}, \mathbf{A}\mathbf{p_{k-1}} \rangle.$$

For the first term, we introduce yet another new variable η_k, denoting

$$\langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k} \rangle = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{w_k} \rangle =: \eta_k.$$

For the second term, we shall use the facts that $\mathbf{r_k} = \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}}$ $= \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{A}\mathbf{p_{k-1}} \implies \mathbf{A}\mathbf{p_{k-1}} = -(1/\alpha_{k-1})(\mathbf{r_k} - \mathbf{r_{k-1}})$ and that $\tilde{\mathbf{r}}_\mathbf{k} \perp \mathbf{r_{k-1}}$ [18] which yields

$$2\langle \tilde{\mathbf{r}}_\mathbf{k}, \beta_k \mathbf{A}\mathbf{p_{k-1}} \rangle = -2\frac{\beta_k}{\alpha_{k-1}} \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{r_k} - \mathbf{r_{k-1}} \rangle = -2\frac{\beta_k}{\alpha_{k-1}} \nu_k.$$

Finally, for the third term we use that $\mu_{k-1} = \langle \mathbf{p_{k-1}}, \mathbf{s_{k-1}} \rangle$, $\alpha_k = \nu_k/\mu_k$, and $\beta_k = \nu_k/\nu_{k-1}$. These relations result in

$$\beta_k^2 \langle \mathbf{p_{k-1}}, \mathbf{A}\mathbf{p_{k-1}} \rangle = \beta_k^2 \langle \mathbf{p_{k-1}}, \mathbf{s_{k-1}} \rangle = \beta_k^2 \mu_{k-1} = \beta_k^2 \frac{\nu_{k-1}}{\alpha_{k-1}} = \beta_k \frac{\nu_k \nu_{k-1}}{\nu_{k-1}\alpha_{k-1}} = \frac{\beta_k}{\alpha_{k-1}} \nu_k.$$

By combining these three rewritten expressions we obtain that

$$\mu_k = \eta_k - (\beta_k/\alpha_{k-1})\nu_k,$$

which at last allows us to state the procedure of ChC-CG below [1].

Even though the order of operations here is different than in PR-CG and M-CG, the principle of grouping inner products together stays the same.

However, unlike in the case of the previous variants, this is not achieved by using a predict-and-recompute scheme.

Algorithm 4 Chronopoulos and Gear Conjugate Gradient: ChG-CG (preconditioned)

1: **procedure** CHG-CG($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
2: INITIALIZE()
3: **for** $k = 1, 2, \ldots$ **do**
4: $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_{k-1}\mathbf{p_{k-1}}$
5: $\mathbf{r_k} = \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}},\ \tilde{\mathbf{r}}_\mathbf{k} = \mathbf{M^{-1}r_k}$
6: $\mathbf{w_k} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k}$
7: $\nu_k = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{r_k} \rangle,\ \eta_k = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{w_k} \rangle$
8: $\beta_k = \nu_k / \nu_{k-1}$
9: $\mathbf{p_k} = \tilde{\mathbf{r}}_\mathbf{k} + \beta_k \mathbf{p_{k-1}}$
10: $\mathbf{s_k} = \mathbf{w_k} + \beta_k \mathbf{s_{k-1}}$
11: $\mu_k = \eta_k - (\beta_k / \alpha_{k-1})\nu_k$
12: $\alpha_k = \nu_k / \mu_k$
13: **end for**
14: **end procedure**

2.4 Pipelined variants

So far, we have introduced several communication-hiding variants which reduce the number of synchronization points to one. With that we are also a bit closer to the goal of overlapping the most costly operations.

2.4.1 Pipe-PR-CG

Let us now derive the pipelined version of PR-CG. The problem we are facing is that most of the inner products in step 10 of PR-CG are dependent on the $\mathbf{s_k}$ from the previous step. This hinders our ability to perform these computational steps in parallel. Fortunately, we can once more rearrange the algorithm using relations equivalent in exact arithmetic [1].

Once again, the following series of expression manipulations and ideas paraphrases the article [1]. To begin with, we rewrite the current expression for $\mathbf{s_k}$ [1]

$$\mathbf{s_k} = \mathbf{A}\mathbf{p_k} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k} + \beta_k \mathbf{A}\mathbf{p_{k-1}},$$

and set a new variable $\mathbf{w_k}$ as $\mathbf{w_k} \coloneqq \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k}$. Using this, we now have [1]

$$\mathbf{s_k} = \mathbf{w_k} + \beta_k \mathbf{s_{k-1}}.$$

Next, we define $\mathbf{u_k} \coloneqq \mathbf{A}\tilde{\mathbf{s}}_\mathbf{k}$, and utilize it to express an iterative relation for $\mathbf{w_k}$ [1]:

$$\mathbf{w_k} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k-1} - \alpha_{k-1}\mathbf{A}\tilde{\mathbf{s}}_\mathbf{k-1} = \mathbf{w_{k-1}} - \alpha_{k-1}\mathbf{u_{k-1}}.$$

By these manipulations we have shifted where the matrix-vector product occurs, and thus we do not need to compute it to directly express $\mathbf{s_k}$. Instead, we perform

a step $\mathbf{u_k} = \mathbf{A}\tilde{\mathbf{s}}_\mathbf{k}$, which we can overlap in parallel with the inner products [1]. Nonetheless, to derive the complete Algorithm 5 we have to do two more things.

First, we must take care of the preconditioning steps. For this we define $\tilde{\mathbf{w}}_\mathbf{k} \coloneqq \mathbf{M}^{-1}\mathbf{w_k}$ in order to express $\tilde{\mathbf{s}}_\mathbf{k}$ as [1]

$$\tilde{\mathbf{s}}_\mathbf{k} = \mathbf{M}^{-1}\mathbf{s_k} = \mathbf{M}^{-1}\mathbf{w_k} + \beta_k \mathbf{M}^{-1}\mathbf{s_{k-1}} = \tilde{\mathbf{w}}_\mathbf{k} + \beta_k \tilde{\mathbf{s}}_\mathbf{k-1},$$

and similarly defining $\tilde{\mathbf{u}}_\mathbf{k} \coloneqq \mathbf{M}^{-1}\mathbf{u_k}$ we get [1]

$$\tilde{\mathbf{w}}_\mathbf{k} = \mathbf{M}^{-1}\mathbf{w_k} = \mathbf{M}^{-1}\mathbf{w_{k-1}} - \alpha_{k-1}\mathbf{M}^{-1}\mathbf{u_{k-1}} = \mathbf{M}^{-1}\mathbf{w_k} = \tilde{\mathbf{w}}_\mathbf{k-1} - \alpha_{k-1}\tilde{\mathbf{u}}_\mathbf{k-1}.$$

Second, as was mentioned earlier, such a rearrangement as we have done for the $\mathbf{w_k}$ often induces lower accuracy and delays in convergence in finite precision arithmetic [1]. Therefore, we once again employ the predict-and-recompute principle. The above derived expressions for the $\mathbf{w_k}$ and the $\tilde{\mathbf{w}}_\mathbf{k}$ shall be used as predictors, which are going to be recomputed later using the "original" expressions $\mathbf{w_k} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k}$ and $\tilde{\mathbf{w}}_\mathbf{k} = \mathbf{M}^{-1}\mathbf{w_k}$. These recompute steps can be computed concurrently with the inner products [1]. Combining the above variable manipulations, we obtain Algorithm 5 [1].

Algorithm 5 Pipelined Predict-and-Recompute Conjugate Gradient: Pipe-PR-CG (preconditioned)

procedure PIPE-PR-CG($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
  INITIALIZE()
  for $k = 1, 2, \ldots$ **do**
    $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_{k-1}\mathbf{p_{k-1}}$
    $\mathbf{r_k} = \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}},\ \tilde{\mathbf{r}}_\mathbf{k} = \tilde{\mathbf{r}}_\mathbf{k-1} - \alpha_{k-1}\tilde{\mathbf{s}}_\mathbf{k-1}$
    $\mathbf{w}'_\mathbf{k} = \mathbf{w_{k-1}} - \alpha_{k-1}\mathbf{u_{k-1}},\ \tilde{\mathbf{w}}'_\mathbf{k} = \tilde{\mathbf{w}}_\mathbf{k-1} - \alpha_{k-1}\tilde{\mathbf{u}}_\mathbf{k-1}$
    $\nu'_k = \nu_{k-1} - 2\alpha_{k-1}\sigma_{k-1} + \alpha^2_{k-1}\gamma_{k-1}$
    $\beta_k = \nu'_k/\nu_{k-1}$
    $\mathbf{p_k} = \tilde{\mathbf{r}}_\mathbf{k} + \beta_k\mathbf{p_{k-1}}$
    $\mathbf{s_k} = \mathbf{w}'_\mathbf{k} + \beta_k\mathbf{s_{k-1}},\ \tilde{\mathbf{s}}_\mathbf{k} = \tilde{\mathbf{w}}'_\mathbf{k} + \beta_k\tilde{\mathbf{s}}_\mathbf{k-1}$
    $\mathbf{u_k} = \mathbf{A}\tilde{\mathbf{s}}_\mathbf{k},\ \tilde{\mathbf{u}}_\mathbf{k} = \mathbf{M}^{-1}\mathbf{u_k}$
    $\mathbf{w_k} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k},\ \tilde{\mathbf{w}}_\mathbf{k} = \mathbf{M}^{-1}\mathbf{w_k}$
    $\mu_k = \langle \mathbf{p_k}, \mathbf{s_k} \rangle,\ \sigma_k = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{s_k} \rangle,\ \gamma_k = \langle \tilde{\mathbf{s}}_\mathbf{k}, \mathbf{s_k} \rangle,\ \nu_k = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{r_k} \rangle$
    $\alpha_k = \nu_k/\mu_k$
  end for
end procedure

Naturally, we can also derive an unpreconditioned variant of Pipe-PR-CG. This merely requires omitting from Algorithm 5 all parts which involve computation with the preconditioner matrix $\mathbf{M}$ and rewriting "tilded" variables as their unpreconditioned counterparts, e.g., $\tilde{\mathbf{r}}_\mathbf{k}$ to $\mathbf{r_k}$. The procedure is given below as Algorithm 6. Please note that this is the version used in all numerical experiments contained in this thesis. The reason for using an unpreconditioned variant is that the experiments, as they are, already involve a rather extensive range of possible combinations of various parameters and inputs. The choice of a preconditioner matrix and scrutinizing the steps which it induces would introduce another layer of complexity. Therefore, this exploration is left for future research.

Algorithm 6 Pipelined Predict-and-Recompute Conjugate Gradient: Pipe-PR-CG (unpreconditioned)

1: **procedure** PIPE-PR-CG (UNPRECONDITIONED)$(\mathbf{A}, \mathbf{b}, \mathbf{x_0})$
2: INITIALIZE()
3: **for** $k = 1, 2, \ldots$ **do**
4: $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_{k-1}\mathbf{p_{k-1}}$
5: $\mathbf{r_k} = \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}}$
6: $\mathbf{w'_k} = \mathbf{w_{k-1}} - \alpha_{k-1}\mathbf{u_{k-1}}$
7: $\nu'_k = \nu_{k-1} - 2\alpha_{k-1}\sigma_{k-1} + \alpha^2_{k-1}\gamma_{k-1}$
8: $\beta_k = \nu'_k/\nu_{k-1}$
9: $\mathbf{p_k} = \mathbf{r_k} + \beta_k\mathbf{p_{k-1}}$
10: $\mathbf{s_k} = \mathbf{w'_k} + \beta_k\mathbf{s_{k-1}}$
11: $\mathbf{u_k} = \mathbf{A}\mathbf{s_k}$
12: $\mathbf{w_k} = \mathbf{A}\mathbf{r_k}$
13: $\mu_k = \langle \mathbf{p_k}, \mathbf{s_k} \rangle,\ \sigma_k = \langle \mathbf{r_k}, \mathbf{s_k} \rangle,\ \gamma_k = \langle \mathbf{s_k}, \mathbf{s_k} \rangle,\ \nu_k = \langle \mathbf{r_k}, \mathbf{r_k} \rangle$
14: $\alpha_k = \nu_k/\mu_k$
15: **end for**
16: **end procedure**

Since the inner products are no longer dependent on variables resulting from the preceding matrix-vector multiplications, it is now possible to overlap these operations if we compute in parallel. This idea is illustrated in Figure 2.3 which depicts the iteration scheme of Pipe-PR-CG.

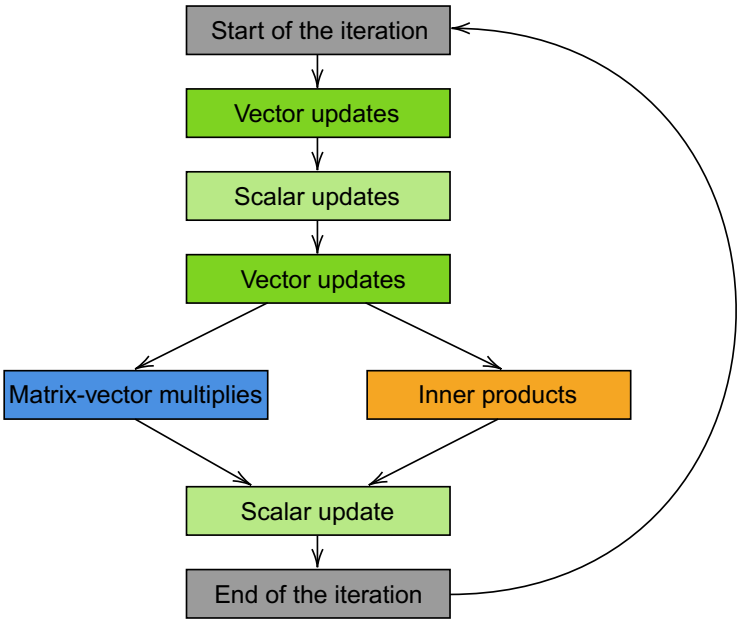

Figure 2.3: Iteration diagram of Pipe-PR-CG

Even though the rearrangements leading to PR-CG and Pipe-PR-CG can help us to hide communication and overlap some operations, they may also lead to increased numerical instability [17]. This is illustrated later in Figures 2.4 and 2.5.

2.4.2 Other pipelined variants

In a similar manner to how we have derived Pipe-PR-CG, we can derive a pipelined version of M-CG as it is formulated in Algorithm 3. The distinction between PR-CG and M-CG is only a different relation for the predicted value ν'_k, which does not come up in the relations used to include pipelining. Therefore, the resulting procedure (Algorithm 7 below) is going to be the same as for Pipe-PR-CG with the only differences being the relation for ν'_k in line 7 and omitted σ_k, which we do not need [1].

Algorithm 7 Pipelined Meurant Conjugate Gradient: Pipe-M-CG (preconditioned)

1: **procedure** PIPE-M-CG($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
2: INITIALIZE()
3: **for** $k = 1, 2, \ldots$ **do**
4: $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_{k-1}\mathbf{p_{k-1}}$
5: $\mathbf{r_k} = \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}},\ \tilde{\mathbf{r}}_\mathbf{k} = \tilde{\mathbf{r}}_\mathbf{k-1} - \alpha_{k-1}\tilde{\mathbf{s}}_\mathbf{k-1}$
6: $\mathbf{w'_k} = \mathbf{w_{k-1}} - \alpha_{k-1}\mathbf{u_{k-1}},\ \tilde{\mathbf{w}}'_\mathbf{k} = \tilde{\mathbf{w}}_\mathbf{k-1} - \alpha_{k-1}\tilde{\mathbf{u}}_\mathbf{k-1}$
7: $\nu'_k = -\nu_{k-1} + \alpha^2_{k-1}\gamma_{k-1}$
8: $\beta_k = \nu'_k/\nu_{k-1}$
9: $\mathbf{p_k} = \tilde{\mathbf{r}}_\mathbf{k} + \beta_k\mathbf{p_{k-1}}$
10: $\mathbf{s_k} = \mathbf{w'_k} + \beta_k\mathbf{s_{k-1}},\ \tilde{\mathbf{s}}_\mathbf{k} = \tilde{\mathbf{w}}'_\mathbf{k} + \beta_k\tilde{\mathbf{s}}_\mathbf{k-1}$
11: $\mathbf{u_k} = \mathbf{A}\tilde{\mathbf{s}}_\mathbf{k},\ \tilde{\mathbf{u}}_\mathbf{k} = \mathbf{M^{-1}u_k}$
12: $\mathbf{w_k} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k},\ \tilde{\mathbf{w}}_\mathbf{k} = \mathbf{M^{-1}w_k}$
13: $\mu_k = \langle \mathbf{p_k}, \mathbf{s_k} \rangle,\ \gamma_k = \langle \tilde{\mathbf{s}}_\mathbf{k}, \mathbf{s_k} \rangle,\ \nu_k = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{r_k} \rangle$
14: $\alpha_k = \nu_k/\mu_k$
15: **end for**
16: **end procedure**

Another possibility is to derive a pipelined version of the Chronopoulos and Gear Conjugate Gradient algorithm. This was originally done by Ghysels and Vanroose in their article "Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm" [20] in 2014. The main idea is the same as for the previously mentioned pipelined variants: to rearrange the iteration so that it is possible to overlap the inner products and the (sparse) matrix-vector multiplication. To this end, we want to rewrite the expression for $\mathbf{w_k}$ from line 6 of ChG-CG (Algorithm 4), since $\mathbf{w_k}$ is needed for the computation of η_k in the next line, thus causing a serial dependence.

The derivation of GV-CG could proceed in the following way [20]. First, we realize the recurrences for $\mathbf{r_k}$ and $\mathbf{s_k}$ can be modified to obtain recurrences for their preconditioned counterparts $\tilde{\mathbf{r}}_\mathbf{k}$ and $\tilde{\mathbf{s}}_\mathbf{k}$, yielding:

$$\tilde{\mathbf{r}}_\mathbf{k} = \tilde{\mathbf{r}}_\mathbf{k-1} - \alpha_{k-1}\tilde{\mathbf{s}}_\mathbf{k-1}$$

$$\tilde{\mathbf{s}}_\mathbf{k} = \tilde{\mathbf{w}}_\mathbf{k} + \beta_k\tilde{\mathbf{s}}_\mathbf{k-1}.$$

Let us now introduce a new variable $\mathbf{u_k} \coloneqq \mathbf{A}\tilde{\mathbf{s}}_\mathbf{k}$. The quantity $\mathbf{w_k}$ can be then rewritten as

$$\mathbf{w_k} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{k-1} - \alpha_{k-1}\mathbf{A}\tilde{\mathbf{s}}_\mathbf{k-1} = \mathbf{w_{k-1}} - \alpha_{k-1}\mathbf{u_{k-1}}.$$

Furthermore, using the expression for $\tilde{\mathbf{s}}_\mathbf{k}$, we obtain

$$\mathbf{u_k} \coloneqq \mathbf{A}\tilde{\mathbf{s}}_\mathbf{k} = \mathbf{A}\tilde{\mathbf{w}}_\mathbf{k} + \beta_k \mathbf{A}\tilde{\mathbf{s}}_\mathbf{k-1}.$$

We have thus modified the algorithm so that the two inner products (line 7 in Algorithm 8) can be computed simultaneously with the matrix-vector multiplication which is no longer required to be calculated prior to the inner products. For the sake of clarity we can introduce another new variable for this: $\mathbf{t_k} \coloneqq \mathbf{A}\tilde{\mathbf{w}}_\mathbf{k}$.

The structure of Algorithm 8 is again of the same form as in the Appendix D of [1].

Algorithm 8 Ghysels and Vanroos Conjugate Gradient: GV-CG (preconditioned)

1: **procedure** GV-CG($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
2: INITIALIZE()
3: **for** $k = 1, 2, \ldots$ **do**
4: $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_{k-1}\mathbf{p_{k-1}}$
5: $\mathbf{r_k} = \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}},\ \tilde{\mathbf{r}}_\mathbf{k} = \tilde{\mathbf{r}}_\mathbf{k-1} - \alpha_{k-1}\tilde{\mathbf{s}}_\mathbf{k-1}$
6: $\mathbf{w_k} = \mathbf{w_{k-1}} - \alpha_{k-1}\mathbf{u_{k-1}},\ \tilde{\mathbf{w}}_\mathbf{k} = \mathbf{M^{-1}}\mathbf{w_k}$
7: $\nu_k = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{r_k} \rangle,\ \eta_k = \langle \tilde{\mathbf{r}}_\mathbf{k}, \mathbf{w_k} \rangle$
8: $\mathbf{t_k} = \mathbf{A}\tilde{\mathbf{w}}_\mathbf{k}$
9: $\beta_k = \nu_k / \nu_{k-1}$
10: $\mathbf{p_k} = \tilde{\mathbf{r}}_\mathbf{k} + \beta_k \mathbf{p_{k-1}}$
11: $\mathbf{s_k} = \mathbf{w_k} + \beta_k \mathbf{s_{k-1}},\ \tilde{\mathbf{s}}_\mathbf{k} = \tilde{\mathbf{w}}_\mathbf{k} + \beta_k \tilde{\mathbf{s}}_\mathbf{k-1}$
12: $\mathbf{u_k} = \mathbf{t_k} + \beta_k \mathbf{u_{k-1}}$
13: $\mu_k = \eta_k - (\beta_k / \alpha_{k-1})\nu_k$
14: $\alpha_k = \nu_k / \mu_k$
15: **end for**
16: **end procedure**

More details about this algorithm can be found in [20]. Additional discussion of ChG-CG and GV-CG as well as some other variants and historical background can be found in [17].

2.4.3 Comparison of variants

Let us now investigate how some of the presented CG variants perform in comparison with each other. According to the theoretical understanding mentioned earlier in this chapter, we would expect that the more a variant deviates from HS-CG the worse its numerical properties become. The following figures illustrate this on a simple example. They show convergence and residual gap graphs for the matrix *bcsstm07* from [21] with the right-hand side $\mathbf{b}$ being all ones and the initial guess $\mathbf{x_0}$ being all zeros. The computations were performed in IEEE double precision. For reference, the convergence graphs contain a line at the level of IEEE double precision machine epsilon ϵ_{mach}.

The most apparent conclusion we can make from the first pair of graphs in Figure 2.4 is that GV-CG is significantly more numerically unstable than Pipe-PR-CG. Another result is that even though the predict-and-recompute based

methods may at first appear to perform almost identically to HS-CG, there might be some hidden differences. When we investigate the relative residual ($||\mathbf{r_k}||/||\mathbf{b}||$) it seems that the algorithms have achieved the same performance, but it can be observed that in terms of the relative true residual ($||\mathbf{b} - \mathbf{Ax_k}||/||\mathbf{b}||$) the methods show slightly different behavior. Specifically, the converge curve of Pipe-PR-CG starts to stagnate earlier then the ones of HS-CG and PR-CG. Although the true residual is more costly to compute, it can allow us to observe what accuracy the methods are truly able to achieve.

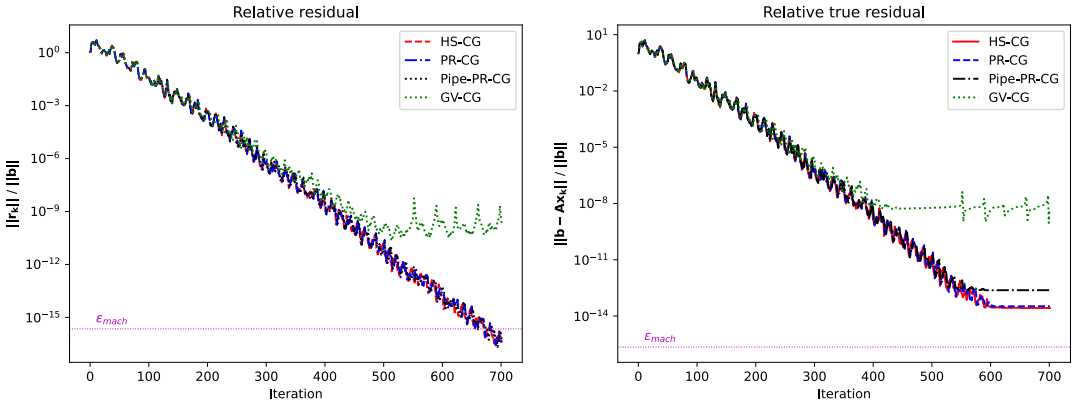

Figure 2.4: Norms of relative residual and relative true residual compared

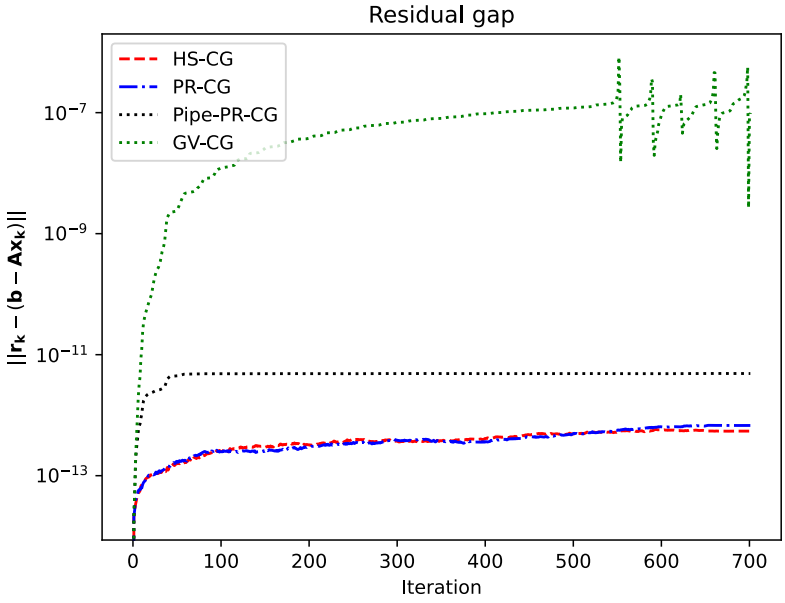

Figure 2.5: Norms of the residual gap compared

Taking a look at the Figure 2.5, we can also observe that the residual gap ($||\mathbf{r_k} - (\mathbf{b} - \mathbf{Ax_k})||$) is for GV-CG notably worse than for the other variants. In addition, the residual gap of Pipe-PR-CG is rather inferior to those of the non-pipelined PR-CG and the original HS-CG. The residual gap is useful as a measure of how reliable any stopping criteria based on the norm $||\mathbf{r_k}||$ might be.

These findings further support the theoretical result that even though the methods are mathematically equivalent in exact arithmetic, their behavior in

finite precision differs [17]. Specifically, the communication-hiding and pipelined algorithms may be more prone to the negative effects of round-off errors.

Let us once again note here that the performance of Pipe-PR-CG is very similar to that of Pipe-M-CG which, however, on average performs slightly worse [1]. This and the numerical instability of GV-CG demonstrated by the figures above are the reasons why Pipe-PR-CG is the primarily investigated pipelined variant and the focus of experiments in this thesis.

3. Effects of silent errors on Pipe-PR-CG

3.1 Sensitivity of convergence

In this section we shall investigate the sensitivity of Unpreconditioned Pipe-PR-CG (Algorithm 6) to silent errors. This is done by investigating the results of numerical experiments designed to analyze how significant is the impact of silent errors on the convergence of the method. First, let us specify the setup and what linear systems were involved in the main experiment. In the following text, the symbol $||\cdot||$ denotes the standard Euclidean 2-norm $||\cdot||_2$. Therefore, the condition number $\kappa(\mathbf{A})$ of a matrix $\mathbf{A}$ is defined as $\kappa(\mathbf{A}) = ||\mathbf{A}||_2 \cdot ||\mathbf{A}^{-1}||_2$.

In all runs, the initial guess $\mathbf{x_0}$ was a vector of all zeros and the right-hand side $\mathbf{b}$ was such that the vector of all ones $\mathbf{e} = (1, 1, \cdots, 1, 1)^T$ was the exact solution of the system, i.e., $\mathbf{b} = \mathbf{Ae}$. No preconditioners were used. The stopping criterion used was that it must hold that $||\mathbf{r_k}||/||\mathbf{b}|| \leq \epsilon_{\text{tol}}$ for the computation to conclude earlier than at the maximal allowed number of iterations, with ϵ_{tol} being 1e$-$10. This threshold can be chosen in many different ways. Here, an inspiration was taken from the article "On Soft Errors in the Conjugate Gradient Method: Sensitivity and Robust Numerical Detection" [13] where the authors utilized two thresholds in their experiments, $\epsilon_{\text{tol}} =$ 1e$-$5 and $\epsilon_{\text{tol}} =$ 1e$-$10. Therefore, our results were computed using the more strict of these, as was mentioned before. The norm of the residual for the stopping criterion did not utilize the already computed $\nu_k = \langle \mathbf{r_k}, \mathbf{r_k} \rangle$ from line 13 of Algorithm 6, so that we can have "illustrative" results for the case of bit flip occurrence in the variable ν_k, without any influence of the silent error on the norm $||\mathbf{r_k}||$ itself.

Additionally, the code was written in such a way that each time an overflow warning occurs, an error is raised instead. This was done to suppress the situations when the Python compiler does not terminate the computation right away, but assigns the result to infinity instead. There were also pure overflow errors, which stopped the computation immediately. All these erroneous cases are counted as "did not converge". In the aforementioned article [13], the same methodology for overflows was used. This is the reason why there were some non-convergent cases for the variable $\mathbf{x_k}$, even though it does not influence any other variables, and therefore we would expect the runs with silent errors in it to be always "convergent".

The injection of silent errors into variables was implemented using the Python module *bitstring* (version 4.1) [22]. Time-wise, the flips always happened after the new value of a variable was computed, i.e., for example, first compute $\alpha_k = \nu_k/\mu_k$, and then insert a bit flip into α_k.

First, for each matrix, a run without any bit flip was performed to determine the number of iterations φ needed to converge. A run "tainted" by a silent error was then deemed as "converged" if it reached the stopping criteria within 1.5φ iterations. The same approach for determining convergence was used in the article [13].

The experiment was performed for each of the 14 variables in Unprecondi-

tioned Pipe-PRCG ($\mathbf{x_k}$, $\mathbf{r_k}$, $\mathbf{w'_k}$, ν'_k, β_k, $\mathbf{p_k}$, $\mathbf{s_k}$, $\mathbf{u_k}$, $\mathbf{w_k}$, μ_k, σ_k, γ_k, ν_k, α_k). There were always 3 different variants of when the bit flip occurred: 0.3φ, 0.6φ, and 0.9φ iterations. This was performed for all 64 bits. In case of scalar variables, one run was performed for each matrix from the dataset, bit number, and flip iteration. For vector variables ($\mathbf{x_k}$, $\mathbf{r_k}$, $\mathbf{w'_k}$, $\mathbf{p_k}$, $\mathbf{s_k}$, $\mathbf{u_k}$, $\mathbf{w_k}$) there was the question of which index to flip the bit in. Generally, no "more important" index exists, so instead of picking a specific one(s), it was chosen randomly. There were 20 trials for each bit, flip iteration, and matrix, so that the randomness in the index choice could be included to a certain degree.

As for the matrix dataset, it consisted of sparse matrices from the SuiteSparse Matrix Collection [21] [23]. Details about it may be found either on the webpage "https://sparse.tamu.edu" [21] itself or in the original article "The University of Florida sparse matrix collection" [23], where the authors describe it in detail. The matrices used were selected to represent various sizes, condition numbers, singular value distributions, structures, as well as problem backgrounds, e.g., from power networks or acoustics. Naturally, all of them satisfy the necessary CG conditions of being symmetric and positive definite. Specifically, the following matrices were used:

- Matrix *1138_bus* $\in \mathbb{R}^{1,138\times 1,138}$, $\kappa(A) = 8.572646e+06$,
- Matrix *bcsstm07* $\in \mathbb{R}^{420\times 420}$, $\kappa(A) = 7.615188e+03$,
- Matrix *bundle1* $\in \mathbb{R}^{10,581\times 10,581}$, $\kappa(A) = 1.004238e+03$,
- Matrix *wathen120* $\in \mathbb{R}^{36,441\times 36,441}$, $\kappa(A) = 2.576962e+03$,
- Matrix *bcsstk05* $\in \mathbb{R}^{153\times 153}$, $\kappa(A) = 1.428114e+04$,
- Matrix *gr_30_30* $\in \mathbb{R}^{900\times 900}$, $\kappa(A) = 1.945739e+02$,
- Matrix *nos7* $\in \mathbb{R}^{729\times 729}$, $\kappa(A) = 2.374510e+09$,
- Matrix *crystm01* $\in \mathbb{R}^{4,875\times 4,875}$, $\kappa(A) = 2.283164e+02$,
- Matrix *aft01* $\in \mathbb{R}^{8,205\times 8,205}$, $\kappa(A) = 4.387086e+18$.

The displayed condition numbers are taken from the information presented in [21]. As was previously stated, the right-hand side $\mathbf{b}$ of the problem $\mathbf{Ax} = \mathbf{b}$ is chosen such that the vector $\mathbf{e} = (1, 1, \cdots, 1, 1)^T$ is the exact solution. Thus, for each matrix the right-hand side differed accordingly.

The output of the experiment is a series of graphs depicting what percentage of runs are "convergent" for each of the 64 bits and each variable. The experiment specifications described in the paragraphs above mean that graphs for scalar variables show how many from 9 trials converged for each bit, as there were 9 matrices used as data. For vector variables there were 180 trials in total for each bit (9 matrices and 20 runs for each bit). The results are presented below, first, as averages over all variables in Figure 3.1 (combined in one big figure) and Figure 3.2 (three separate subfigures for each flip iteration option), and then for all of the 14 variables separately from Figure 3.5 onward. The averages are calculated with each variable having the same weight. Every graph includes thin vertical

lines, which separate the sign (1 bit), exponent (11 bits), and mantissa (52 bits). Additionally, Figure 3.3 and Figure 3.4 depict how the behavior of the algorithm may vary depending on the flip iteration and bit number.

Now, let us take a closer look at the overall outcome of the main sensitivity experiment in Figure 3.1 and Figure 3.2. Some spikes in the curves might be caused by the fact that bit flips "from 0 to 1" and "from 1 to 0" are not equally significant [13], as was mentioned in the section about silent errors. A non-convergent spike can be observed for the second bit, but this might be expected as it is the most significant bit in terms of the absolute value of a number. Therefore, it is most likely to cause an overflow error that inflates the non-convergent cases. The small drops for the 8th bit for flip at 0.9φ and the 9th bit for flips at 0.3φ and 0.6φ might be caused by these bits being significant for some value range our variables often fall into.

Interesting is the fact that seemingly, the earlier a bit flip occurs the greater is its effect on the overall convergence. Figure 3.3 contains convergence curves of the relative residual $||\mathbf{r_k}||/||\mathbf{b}||$ for all three time-wise flip options when the 15th bit is flipped in a computation for matrix *1138_bus*. Dotted purple lines and the solid black line denote when the flips occurred and where the 1.5φ termination point is, respectively. Observing this, we can indeed see that in this case the computation was more heavily influenced by an earlier flip. On the other hand, flipping the 15th bit at a point when the method had almost converged did not have a significant effect on the number of extra iterations necessary.

In general, it seems that silent errors in bits numbered around 25 and higher have no influence on whether the convergence is heavily delayed or not. This is an outcome that might be expected due to the somehow decreasing "significance" of bits as we proceed to those with higher index. A special qualitatively different case is the first bit - the sign, as it, unlike the other bits, does not influence the absolute value of the number.

The concept of decreasing effect of a bit flip as the bit number rises is illustrated in Figure 3.4 which, as an exception in this chapter, has a fixed number of iterations for each run in order to illustrate the behavior more clearly. We can see there that for our data, in terms of both the relative residual and the relative true residual, the computation is indeed impacted more significantly by flips in the sign and exponent bits. The fact that the flip in the 11th bit is in this case more influential than the one in the 6th bit may again be caused by whether it is a "from 0 to 1" or "from 1 to 0" flip.

The individual results for each variable presented in Figures 3.5 - 3.18 generally display no significant divergence from the average. The only notable difference is that the scalar variables seem to be more sensitive to flips in the sign bit. This can have multiple causes, such as that some of the scalar variables, e.g., γ_k, and ν_k are supposed to be always non-negative as they denote inner products of two identical vectors, a property the flip in the sign bit can violate. Another reason might be that sign flips in vector variables are somehow less influential, because vectors have multiple indices. For instance, a reversed sign in α_k - the "step size" for updating $\mathbf{x_k}$, $\mathbf{r_k}$, and $\mathbf{w'_k}$ - leads to taking a step in the opposite "direction", whereas a sign flip in, e.g., $\mathbf{s_k}$ changes the "direction" in only one dimension. Even though this logic can be applied to flips in other bits as well, the sign might posses some unique properties because of its qualitative difference

to other bits.

In conclusion, it seems that bit flips influence the Pipe-PR-CG algorithm more significantly when they occur early in the computation or when they are in bits which are either the sign or have a serious impact on the absolute value of the altered number.

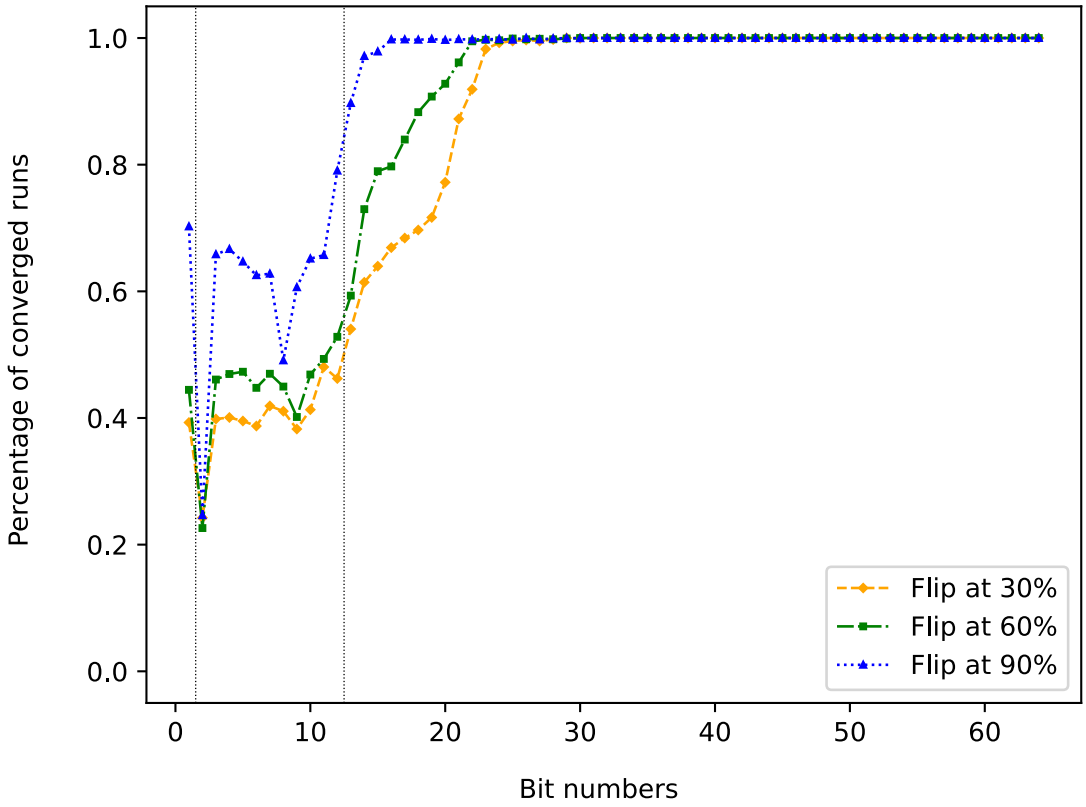

Figure 3.1: Bit flip sensitivity: averages from all variables together

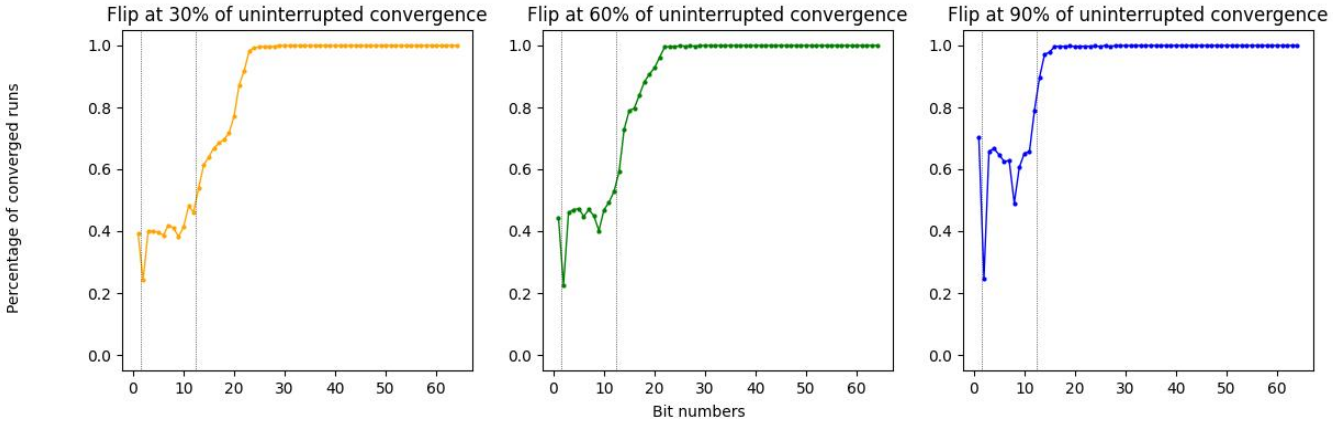

Figure 3.2: Bit flip sensitivity: averages from all variables side by side

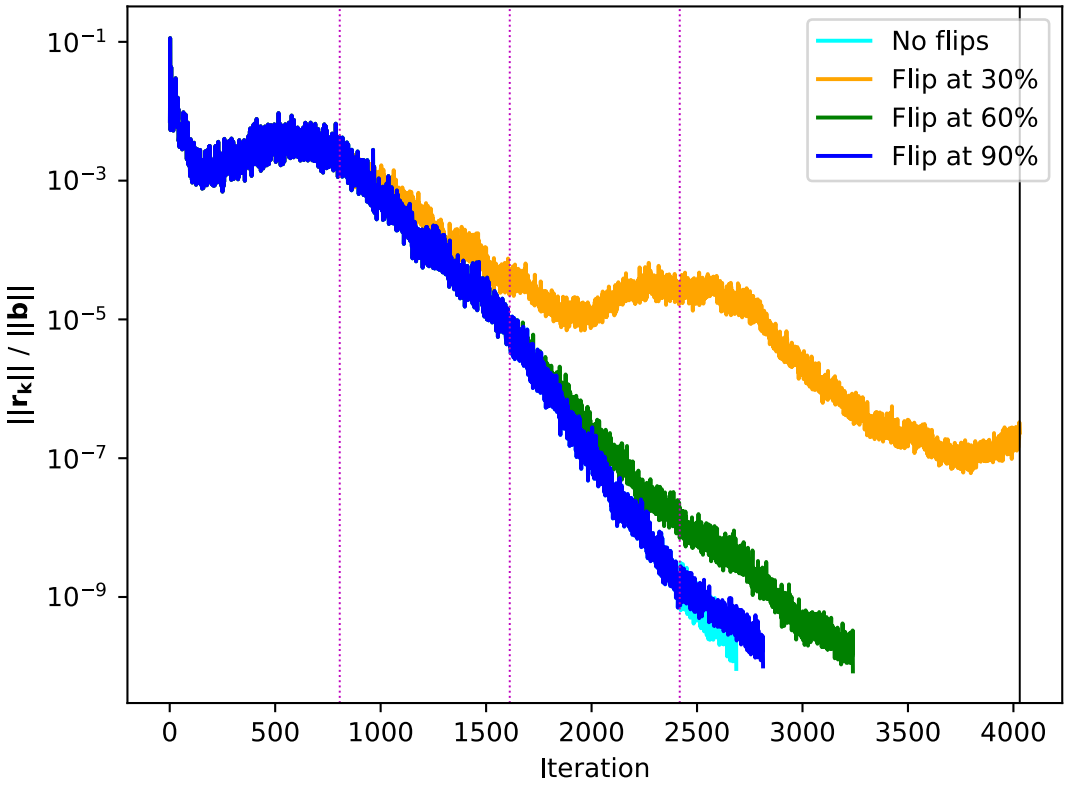

Figure 3.3: Convergence curves of the relative residual $||\mathbf{r_k}||/||\mathbf{b}||$ when the 15th bit of β_k is flipped for the matrix *1138_bus*. The purple dotted lines denote where the flips have occurred.

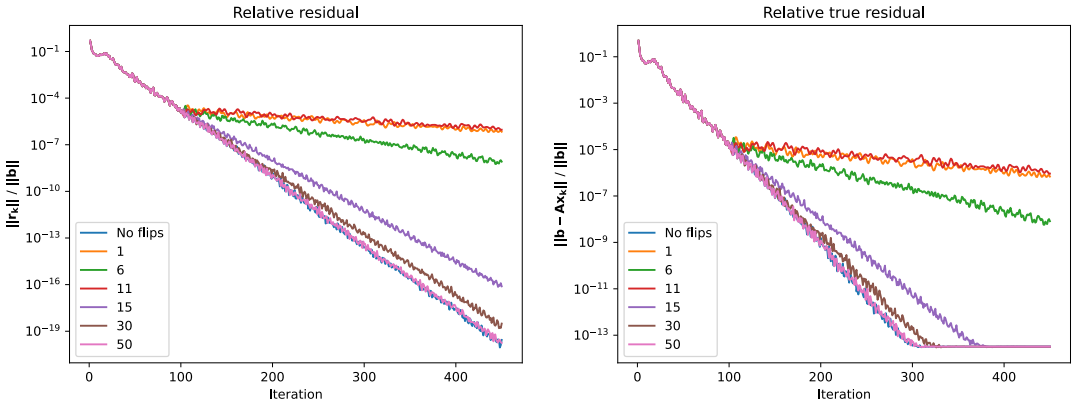

Figure 3.4: Residual convergence curves when various bits of σ_k are flipped in the 100th iteration for the matrix *bundle1*

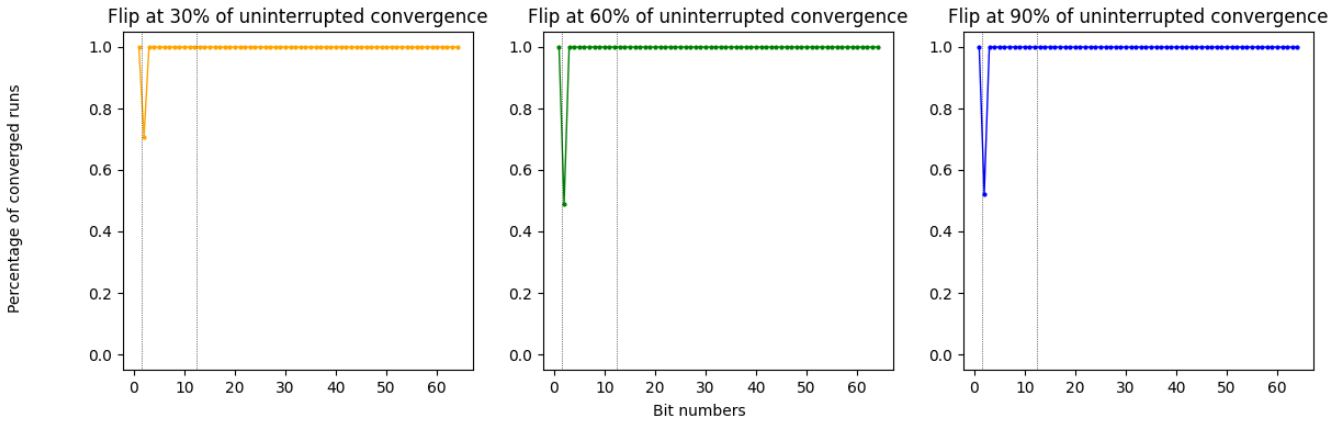

Figure 3.5: Bit flip sensitivity for $\mathbf{x_k}$

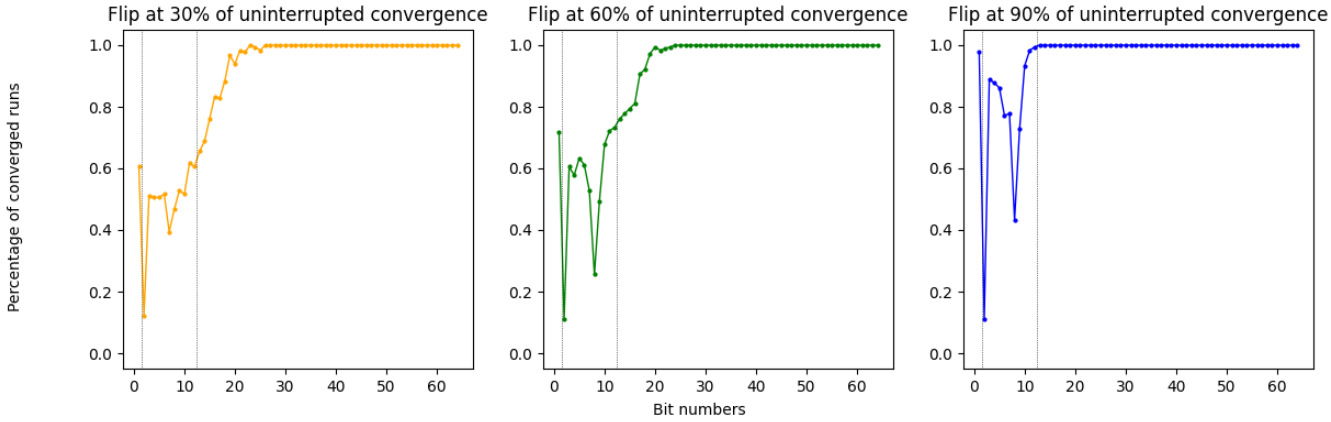

Figure 3.6: Bit flip sensitivity for $\mathbf{r_k}$

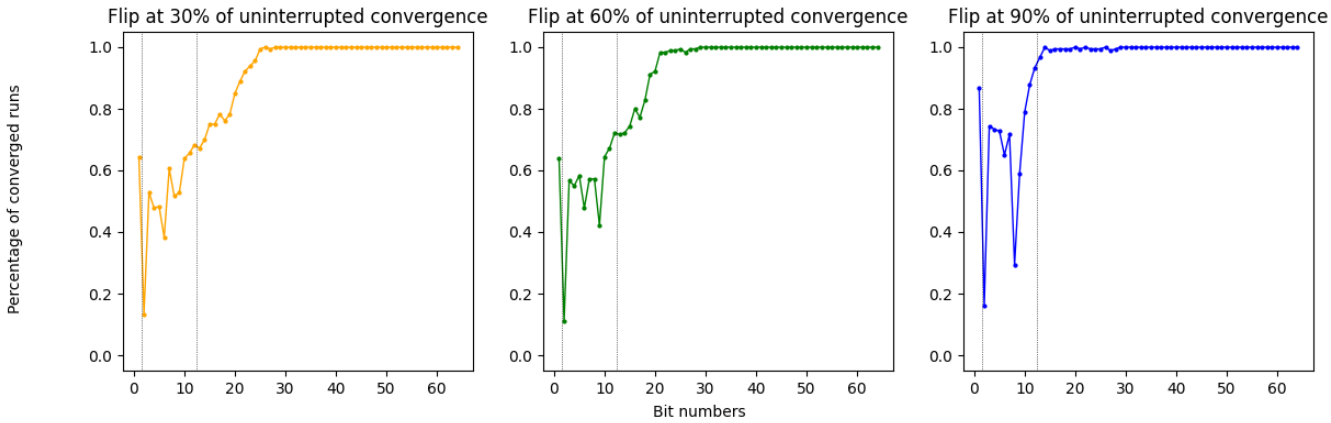

Figure 3.7: Bit flip sensitivity for $\mathbf{w'_k}$

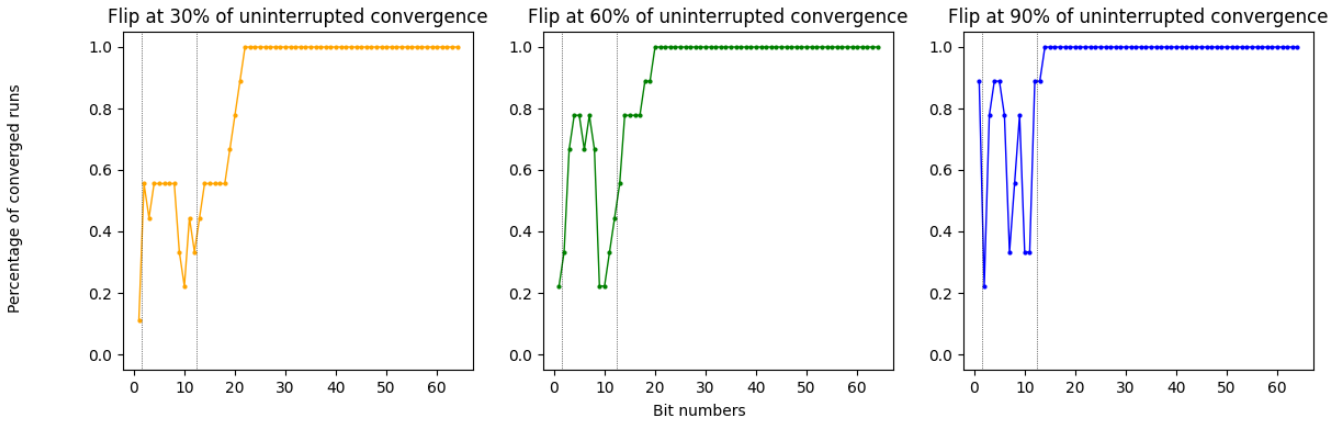

Figure 3.8: Bit flip sensitivity for ν'_k

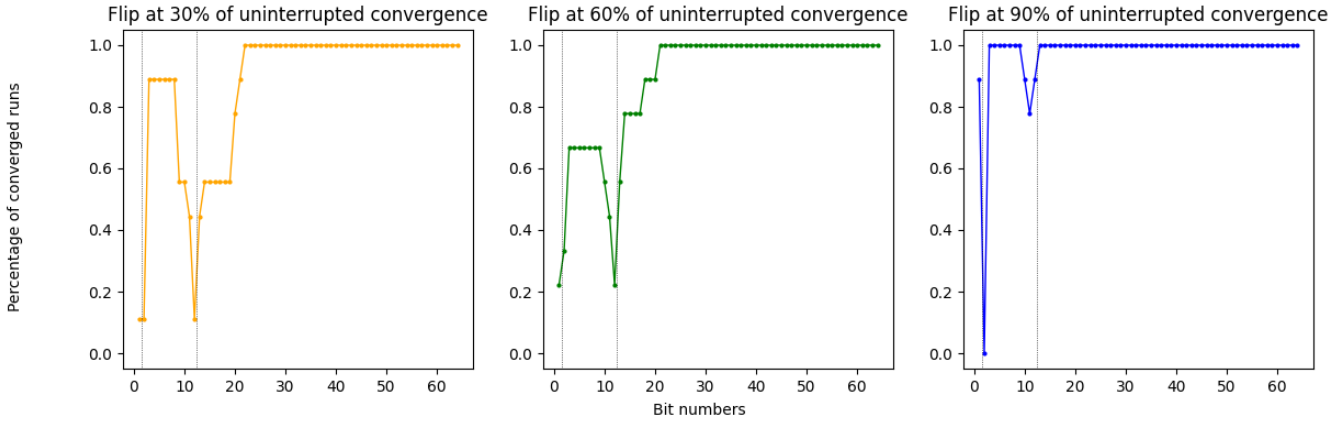

Figure 3.9: Bit flip sensitivity for β_k

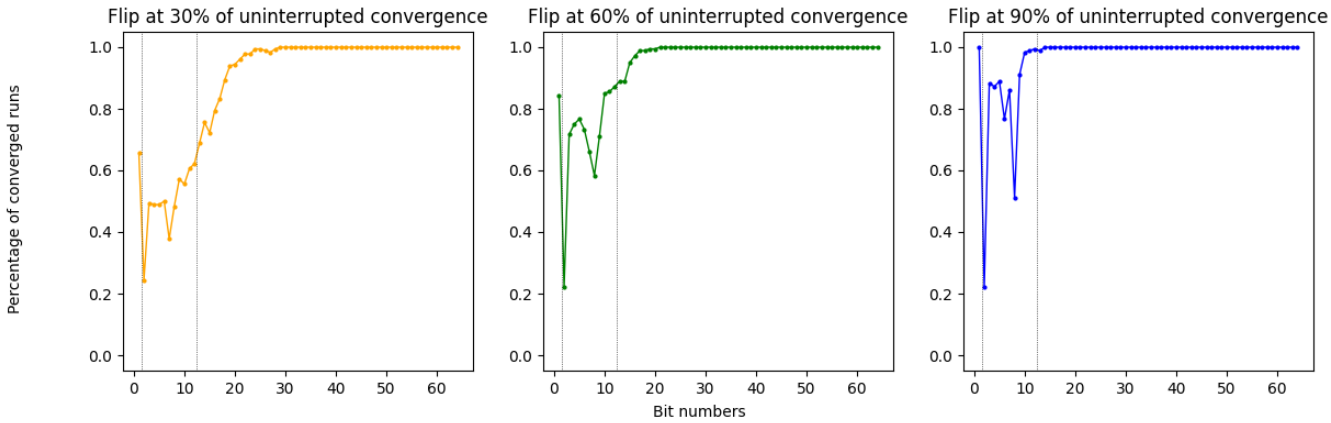

Figure 3.10: Bit flip sensitivity for $\mathbf{p_k}$

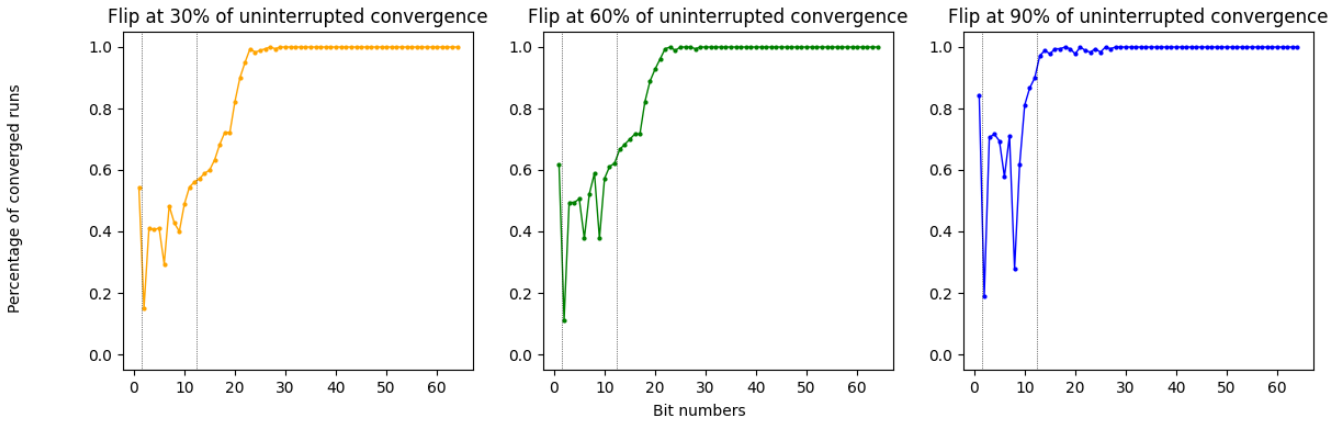

Figure 3.11: Bit flip sensitivity for $\mathbf{s_k}$

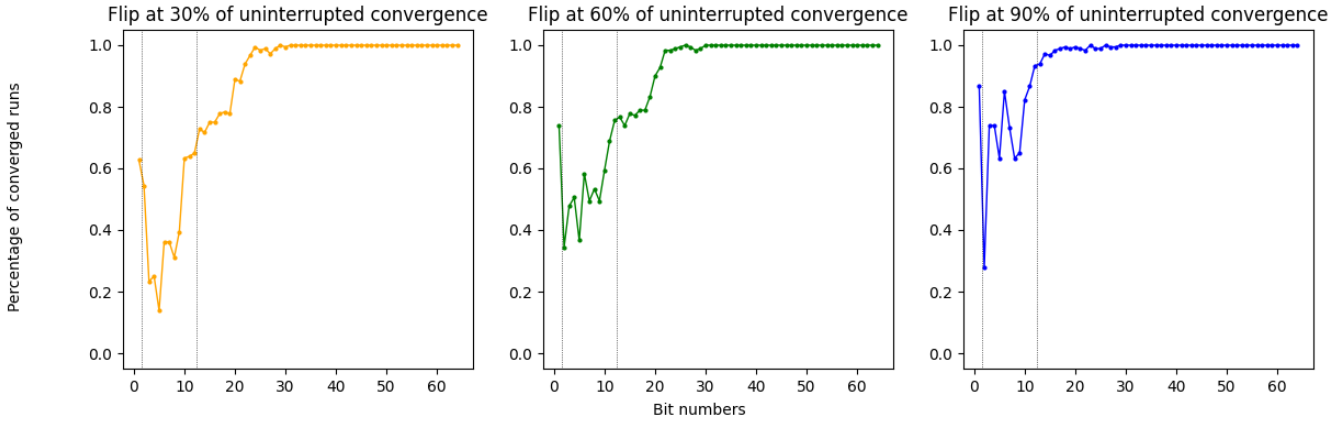

Figure 3.12: Bit flip sensitivity for $\mathbf{u_k}$

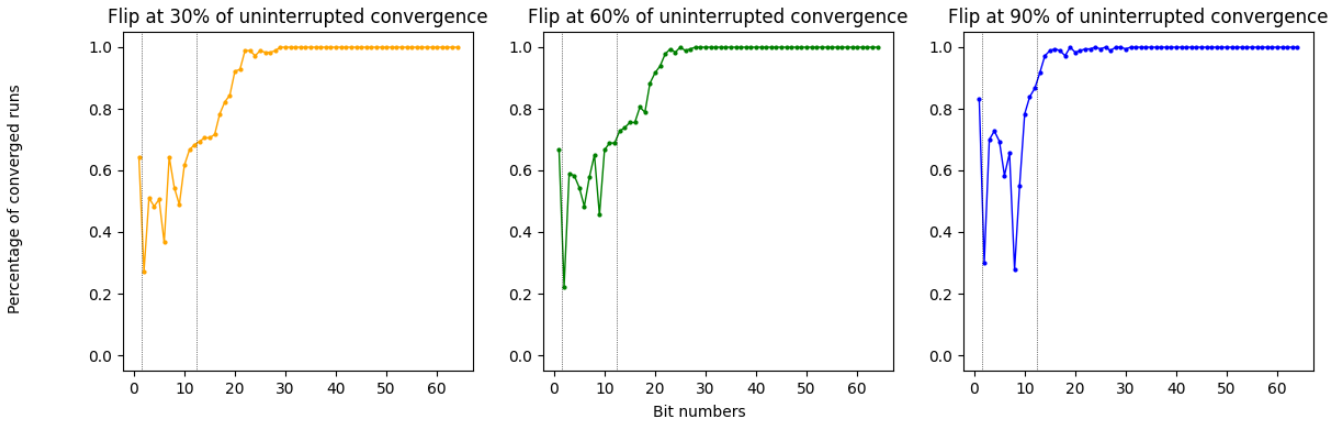

Figure 3.13: Bit flip sensitivity for $\mathbf{w_k}$

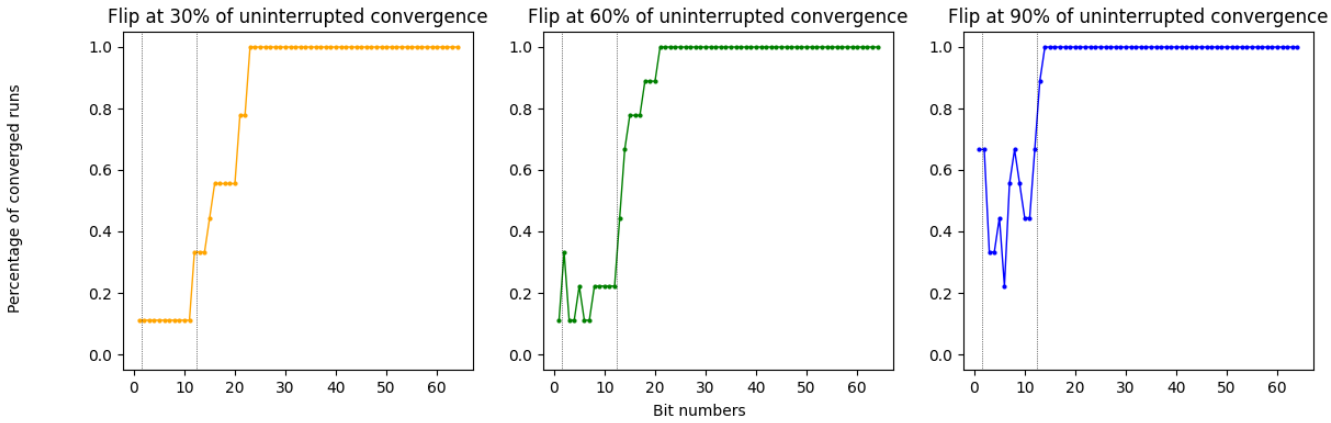

Figure 3.14: Bit flip sensitivity for μ_k

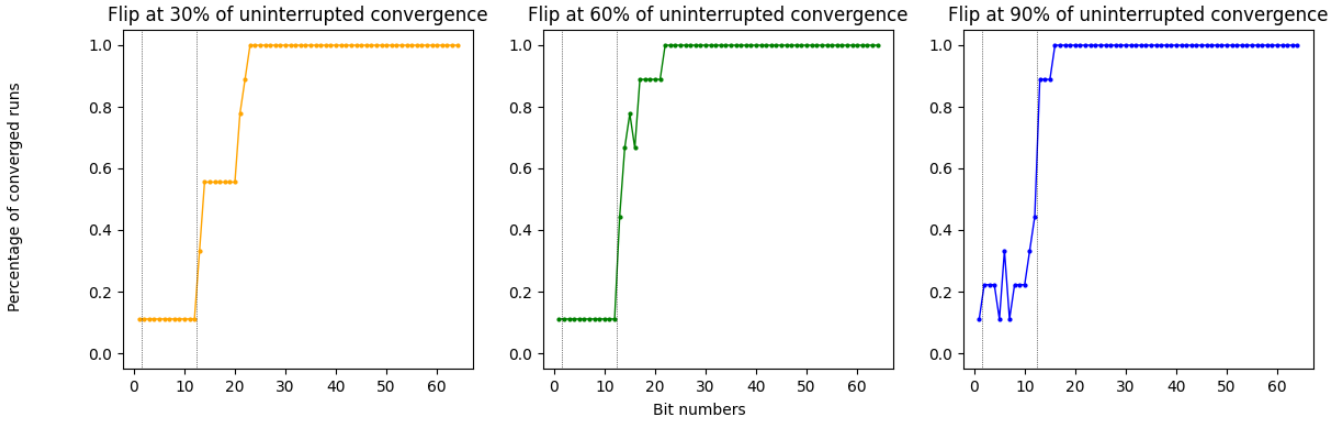

Figure 3.15: Bit flip sensitivity for σ_k

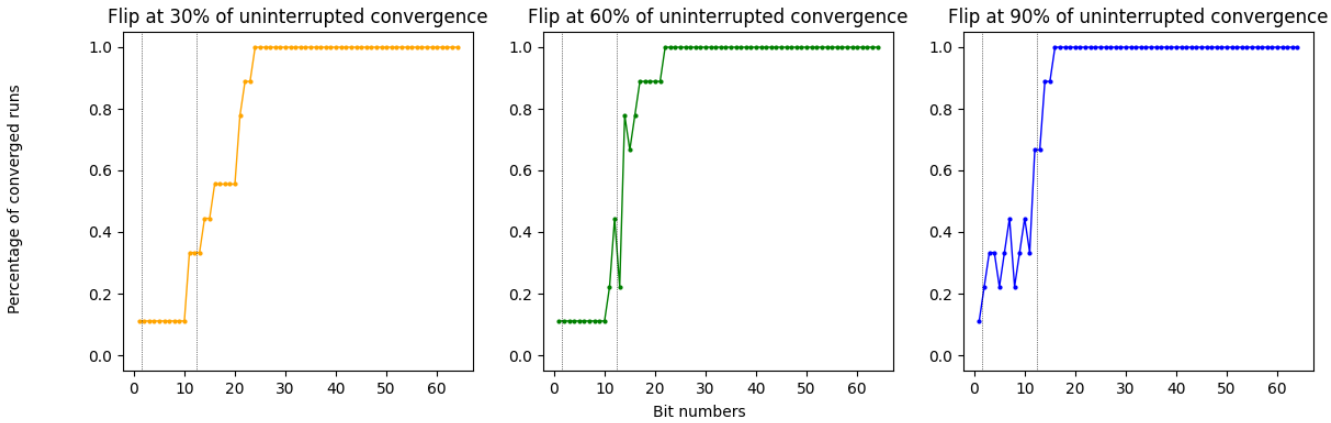

Figure 3.16: Bit flip sensitivity for γ_k

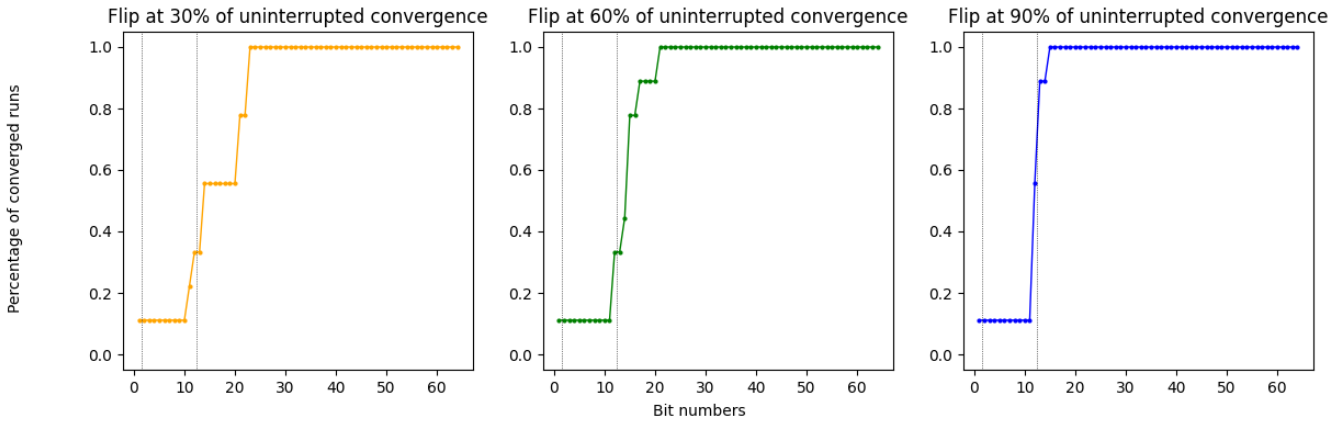

Figure 3.17: Bit flip sensitivity for ν_k

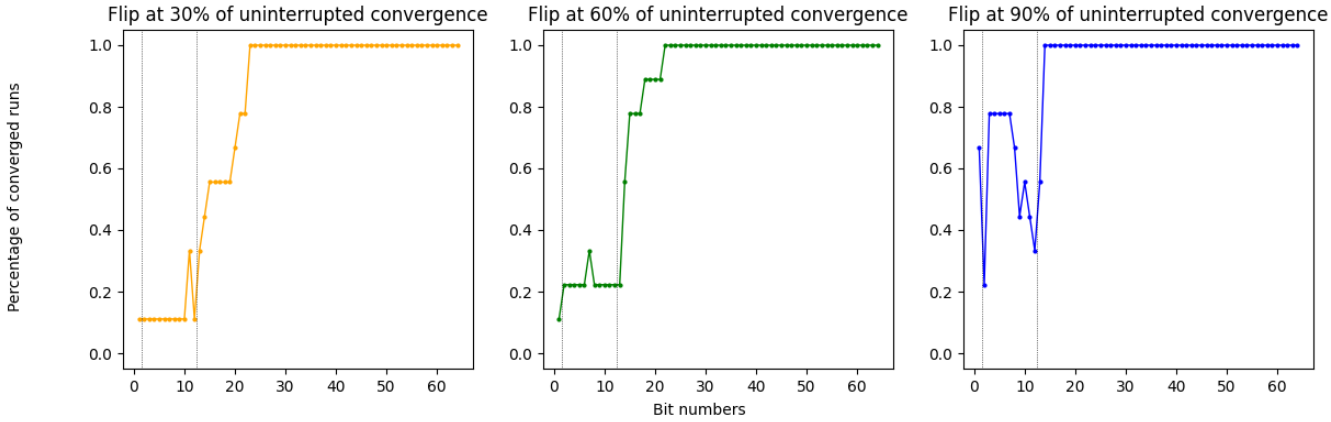

Figure 3.18: Bit flip sensitivity for α_k

3.2 Other effects and error detection

As we have seen in the previous section, silent errors can have significant influence on the convergence of the unpreconditioned Pipe-PR-CG algorithm. Naturally, it can be surmised that other aspects of the procedure might be affected as well. Consequently, some of the effects could be utilized to our ultimate goal - the detection of silent errors. This is the idea of the *algorithm-based fault tolerance methods* which were briefly described in first chapter of this thesis. The fundamental concept of this approach is to derive some criteria of silent error detection from the theoretical or practical knowledge we possess of the algorithm [11]. In our case, we could try to utilize the predict-and-recompute principle which allows us to hide some of the communication.

In Pipe-PR-CG, there are two variables whose value is first predicted using an alternative relation and then recomputed. These are ν_k and $\mathbf{w_k}$. We might try to investigate the "gaps" between their predicted version and their recomputed version, i.e., $|\nu_k - \nu'_k|$ and $||\mathbf{w_k} - \mathbf{w'_k}||$ which are supposed to be zero in exact arithmetic. Let us now shortly investigate the "cheaper" of these two to compute - the *ν-gap*.

The following Figure 3.19 depicts the ν-gap values when the 15th bit of γ_k is flipped in the 100th iteration for the matrix *bcsstm07*. The computation is without preconditioning, with initial guess $\mathbf{x_0}$ being a vector of all zeros, and the right-hand side $\mathbf{b}$ is once again such that it holds that $\mathbf{b} = \mathbf{Ae}$, where $\mathbf{e} = (1, 1, \cdots, 1, 1)^T$. As can be observed, there is a significant outlier among the values in the 101st iteration, i.e., in the very next iteration after the bit flip. This is a promising result which indicates that the ν-gap might be useful for silent error detection and it motivates further exploration into the prospect of utilizing certain quantities for the detection of silent errors in Pipe-PR-CG.

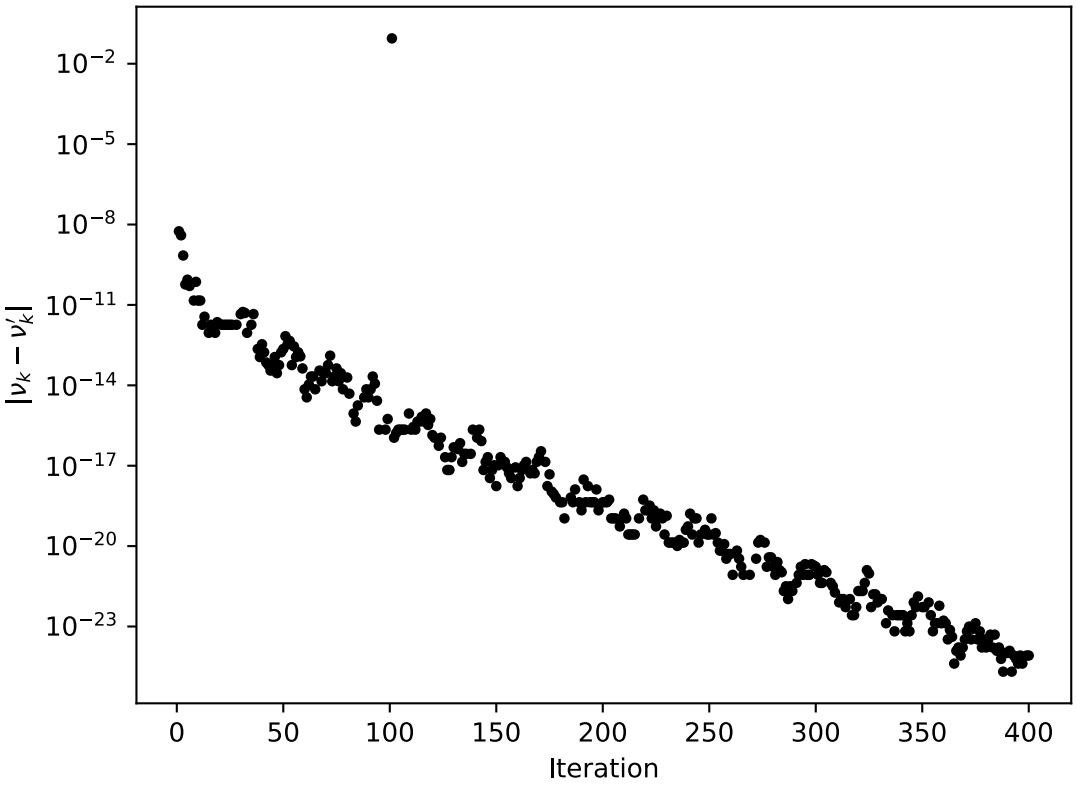

Figure 3.19: The ν-gap ($|\nu_k - \nu'_k|$) when the 15th bit of γ_k is flipped in the 100th iteration for the matrix bcsstm07

The next chapter studies the possibility of using several variable "gaps" to this end. By employing rounding error analysis we derive bounds for the gaps and, subsequently, investigate whether violations of these bounds can be utilized to detect silent errors. A series of graphs resulting from numerical experiments is presented to demonstrate the effectiveness of this approach, and to examine which silent errors each gap is able to detect, as it becomes evident that no studied gap is universally sensitive to flips in all variables. In cases where simple bound violation is not sufficient, alternative techniques are derived.

4. Relations for the detection of silent errors in Pipe-PR-CG

In this chapter, we attempt to derive relations which can be effectively used to detect silent errors in the Pipelined PR-CG algorithm. We present several so-called "gaps", and then employ rounding error analysis to obtain expressions which bound these gaps from above. Subsequently, numerical experiments are performed for each of these gap-bound pairs to judge how effectively they can be utilized to detect silent errors. The idea is that a violation of the bound is a potential indicator of a bit flip having occurred.

This is done for computations without any preconditioning. Therefore, from this point onward, the name "Pipe-PR-CG" refers strictly to Unpreconditioned Pipe-PR-CG. The reason for not including any preconditioner $\mathbf{M}$ is the same as in the case of bit flip sensitivity experiments in the previous chapter. Although preconditioning is widely used to improve properties of linear systems and to speed up computations, it would also introduce another layer of complexity to our experiments. For this reason, the generalization of the detection techniques presented in this thesis to preconditioned systems is left for future research. All computation were performed in IEEE double precision, as was the case in the previous chapter.

4.1 ν-gap

The first quantity for silent error detection we are going to investigate is the ν-gap which was already mentioned at the end of the previous chapter. It is the difference between the predicted value (ν'_k) and the recomputed value of ν_k in Pipe-PR-CG, i.e., $|\nu_k - \nu'_k|$. Let us recall how these variables are defined. It holds that

$$\nu_k = \langle \mathbf{r_k}, \mathbf{r_k} \rangle,$$

and

$$\nu'_k = \nu_{k-1} - 2\alpha_{k-1}\sigma_{k-1} + \alpha_{k-1}^2\gamma_{k-1},$$

which are mathematically equivalent, i.e., they would be equal if the algorithm was executed in exact arithmetic.

4.1.1 Derivation of bounds

As can be observed by investigating Figure 3.19 at the end of the previous chapter, the ν-gap shows promising potential for silent error detection in Pipe-PR-CG. However, an issue is how to determine when the value of the ν-gap signals a potential silent error occurrence. One problem is that, as can be seen in Figure 3.19, the ν-gap can fluctuate. Moreover, even if a silent error influences the ν-gap, nothing guarantees there will always be such a distinct outlier value as in the aforementioned graph. Therefore, it is highly desirable to derive some quantity which bounds the ν-gap from above, and then utilize this bound to determine if

the ν-gap indicates the possibility of a silent error occurring by checking whether the ν-gap exceeded the bound. For the upcoming derivation, let us introduce a new symbol $\Delta_{\nu'_k}$ to denote the difference between ν_k and ν'_k:

$$\Delta_{\nu'_k} \coloneqq \nu_k - \nu'_k = \langle \mathbf{r_k}, \mathbf{r_k} \rangle - \nu'_k.$$

Therefore, the ν-gap can also be denoted by $|\Delta_{\nu'_k}|$.

Now, let us demonstrate how the above-described bound can be obtained. The explanation below follows the article [1], where a similar (slightly less tight) bound was originally derived in the way which is presented here, albeit for a different purpose. Compared to the article, some steps here are explained in more detail; nonetheless, please note that most of the derivation is not an original work of this thesis.

The bound is derived by performing a so-called "rounding error analysis", which is a tool to model how calculations in finite precision arithmetic differ from their exact arithmetic counterparts. In this analysis, we shall work with the standard model of arithmetic with floating-point numbers. Let the symbol $\circ$ denote one of the operations $\{+, -, \times, \div\}$, ϵ the machine precision, a and b arbitrary feasible real numbers, and $\mathrm{fp}(\cdot)$ which operation is performed in finite precision. Then, it holds that [1]:

$$|\mathrm{fp}(a \circ b) - a \circ b| \leq \epsilon \, |a \circ b|. \tag{4.1}$$

Within this framework, it is possible to derive bounds on some of the standard vector operations. Letting $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and $a \in \mathbb{R}$, we have that (see, e.g., [1])

$$||\mathrm{fp}(\mathbf{x} + a\mathbf{y}) - (\mathbf{x} + a\mathbf{y})|| \leq \epsilon \, (||\mathbf{x}|| + 2|a| \, ||\mathbf{y}||), \tag{4.2}$$

$$||\mathrm{fp}(\langle \mathbf{x}, \mathbf{y} \rangle) - \langle \mathbf{x}, \mathbf{y} \rangle|| \leq \epsilon \, n \, ||\mathbf{x}|| \, ||\mathbf{y}||, \tag{4.3}$$

$$||\mathrm{fp}(\mathbf{A}\mathbf{x}) - \mathbf{A}\mathbf{x}|| \leq \epsilon \, c \, ||\mathbf{A}|| \, ||\mathbf{x}||, \tag{4.4}$$

where c is a constant depending on specific properties of the matrix $\mathbf{A}$. For instance, it is frequently taken as $c = mn^{1/2}$, where m is the maximum number of nonzeros over the rows of $\mathbf{A}$ [1].

Before we start the analysis itself, let us recall some of the relations from Pipe-PR-CG which are going to be useful. Here, we introduce a new symbol δ which denotes a round-off error introduced in a calculation, i.e., the difference between the actual finitely computed result and the exact expression for the variable denoted in the subscript of δ. It is either a scalar or a vector, depending on the variable it is associated with, and it can be bounded by using a suitable inequality from (4.1) - (4.4) [1]. Taking for instance δ_{σ_k} as an example, it holds that $\delta_{\sigma_k} = \sigma_k - \langle \mathbf{r_k}, \mathbf{s_k} \rangle \leq \epsilon \, n \, ||\mathbf{r_k}|| \, ||\mathbf{s_k}||$. The above-mentioned relations we are going to use to derive the ν-bound are given below [1]:

$$\begin{aligned}
\mathbf{r_k} &= \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}} + \delta_{\mathbf{r_k}}, \\
\nu'_k &= \nu_{k-1} - 2\alpha_{k-1}\sigma_{k-1} + \alpha_{k-1}^2\gamma_{k-1} + \delta_{\nu'_k}, \\
\sigma_k &= \langle \mathbf{r_k}, \mathbf{s_k} \rangle + \delta_{\sigma_k}, \\
\gamma_k &= \langle \mathbf{s_k}, \mathbf{s_k} \rangle + \delta_{\gamma_k}, \\
\nu_k &= \langle \mathbf{r_k}, \mathbf{r_k} \rangle + \delta_{\nu_k}.
\end{aligned} \tag{4.5}$$

By using the expressions from (4.5) and basic arithmetic manipulations, we can rewrite the equality for $\Delta_{\nu'_k}$ as

$$\begin{aligned}
\Delta_{\nu'_k} &= \langle \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}} + \delta_{\mathbf{r_k}}, \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}} + \delta_{\mathbf{r_k}} \rangle - \nu'_k \\
&= \langle \mathbf{r_{k-1}}, \mathbf{r_{k-1}} \rangle - 2\alpha_{k-1}\langle \mathbf{r_{k-1}}, \mathbf{s_{k-1}} \rangle + \alpha_{k-1}^2 \langle \mathbf{s_{k-1}}, \mathbf{s_{k-1}} \rangle \\
&\quad + 2\langle \delta_{\mathbf{r_k}}, \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}} \rangle + \langle \delta_{\mathbf{r_k}}, \delta_{\mathbf{r_k}} \rangle - \nu'_k \\
&= \nu_{k-1} - \delta_{\nu_{k-1}} - 2\alpha_{k-1}(\sigma_{k-1} - \delta_{\sigma_{k-1}}) + \alpha_{k-1}^2(\gamma_{k-1} - \delta_{\gamma_{k-1}}) \\
&\quad + 2\langle \delta_{\mathbf{r_k}}, \mathbf{r_k} \rangle - \langle \delta_{\mathbf{r_k}}, \delta_{\mathbf{r_k}} \rangle - \nu'_k \\
&= (\nu_{k-1} - 2\alpha_{k-1}\sigma_{k-1} + \alpha_{k-1}^2\gamma_{k-1}) - (\delta_{\nu_{k-1}} - 2\alpha_{k-1}\delta_{\sigma_{k-1}} + \alpha_{k-1}^2\delta_{\gamma_{k-1}}) \\
&\quad + 2\langle \delta_{\mathbf{r_k}}, \mathbf{r_k} \rangle - \langle \delta_{\mathbf{r_k}}, \delta_{\mathbf{r_k}} \rangle - \nu'_k \\
&= \nu'_k - \delta_{\nu'_k} - (\delta_{\nu_{k-1}} - 2\alpha_{k-1}\delta_{\sigma_{k-1}} + \alpha_{k-1}^2\delta_{\gamma_{k-1}}) + 2\langle \delta_{\mathbf{r_k}}, \mathbf{r_k} \rangle - \langle \delta_{\mathbf{r_k}}, \delta_{\mathbf{r_k}} \rangle - \nu'_k \\
&= 2\langle \delta_{\mathbf{r_k}}, \mathbf{r_k} \rangle - \langle \delta_{\mathbf{r_k}}, \delta_{\mathbf{r_k}} \rangle - (\delta_{\nu_{k-1}} - 2\alpha_{k-1}\delta_{\sigma_{k-1}} + \alpha_{k-1}^2\delta_{\gamma_{k-1}}) - \delta_{\nu'_k}.
\end{aligned}$$

Therefore, by taking the norm of both sides, we obtain

$$|\Delta_{\nu'_k}| \leq 2||\delta_{\mathbf{r_k}}||\,||\mathbf{r_k}|| + ||\delta_{\mathbf{r_k}}||^2 + |\delta_{\nu_{k-1}}| + 2|\alpha_{k-1}||\delta_{\sigma_{k-1}}| + |\alpha_{k-1}|^2|\delta_{\gamma_{k-1}}| + |\delta_{\nu'_k}|.$$

Before we derive bounds for each individual δ term on the right-hand side, let us first introduce a useful relation. If we rewrite the expression for $\mathbf{r_k}$ from (4.5) we obtain that

$$\mathbf{s_{k-1}} = \frac{1}{\alpha_{k-1}}(\mathbf{r_{k-1}} - \mathbf{r_k} + \delta_{\mathbf{r_k}}),$$

from which we can bound $\mathbf{s_k}$ from above as

$$||\mathbf{s_{k-1}}|| \leq \frac{1}{|\alpha_{k-1}|}(||\mathbf{r_{k-1}}|| + ||\mathbf{r_k}|| + ||\delta_{\mathbf{r_k}}||). \tag{4.6}$$

Now, let us start by bounding $\delta_{\mathbf{r_k}}$. The first inequality below is obtained by using relations (4.5) and (4.2). Then, we use (4.6), and subsequently rewrite the last term as $O(\epsilon^2)$, owing to the initial inequality. This yields

$$\begin{aligned}
||\delta_{\mathbf{r_k}}|| &\leq \epsilon\,(||\mathbf{r_{k-1}}|| + 2|\alpha_{k-1}|\,||\mathbf{s_{k-1}}||) \\
&\leq \epsilon\left(||\mathbf{r_{k-1}}|| + 2|\alpha_{k-1}|\left(\frac{1}{|\alpha_{k-1}|}(||\mathbf{r_{k-1}}|| + ||\mathbf{r_k}|| + ||\delta_{\mathbf{r_k}}||)\right)\right) \\
&\leq \epsilon\,(2||\mathbf{r_k}|| + 3||\mathbf{r_{k-1}}||) + 2\,\epsilon\,||\delta_{\mathbf{r_k}}|| \\
&\leq 3\,\epsilon\,(||\mathbf{r_k}|| + ||\mathbf{r_{k-1}}||) + O(\epsilon^2).
\end{aligned} \tag{4.7}$$

Next, the bound for $\delta_{\nu_{k-1}}$ can be obtained by a straightforward application of (4.5) and (4.3) as

$$|\delta_{\nu_{k-1}}| \leq \epsilon\,n\,||\mathbf{r_{k-1}}||^2. \tag{4.8}$$

To bound $\delta_{\sigma_{k-1}}$, we once again utilize (4.5) and (4.3), then (4.6), and eventually also (4.7) to rewrite the $\epsilon\,||\delta_{\mathbf{r_k}}||$ term as $O(\epsilon^2)$. By doing this, we obtain

$$\begin{aligned}
|\delta_{\sigma_{k-1}}| &\leq \epsilon\,n\,||\mathbf{r_{k-1}}||\,||\mathbf{s_{k-1}}|| \\
&\leq \epsilon\,n\,||\mathbf{r_{k-1}}||\left(\frac{1}{|\alpha_{k-1}|}(||\mathbf{r_{k-1}}|| + ||\mathbf{r_k}|| + ||\delta_{\mathbf{r_k}}||)\right) \\
&\leq \epsilon\,n\,\frac{1}{|\alpha_{k-1}|}(||\mathbf{r_{k-1}}||^2 + ||\mathbf{r_{k-1}}||\,||\mathbf{r_k}||) + O(\epsilon^2).
\end{aligned} \tag{4.9}$$

To derive bounds for $\delta_{\gamma_{k-1}}$ and $\delta_{\nu'_k}$, we first present an auxiliary alteration of (4.6) by (4.7). This yields

$$||\mathbf{s_{k-1}}|| \leq (1+3\epsilon)\frac{1}{|\alpha_{k-1}|}(||\mathbf{r_{k-1}}|| + ||\mathbf{r_k}||) + O(\epsilon^2), \tag{4.10}$$

which we can now use together with (4.5) and (4.3) to bound $\delta_{\gamma_{k-1}}$ as

$$\begin{aligned} |\delta_{\gamma_{k-1}}| &\leq \epsilon\, n\, ||\mathbf{s_{k-1}}||^2 \\ &\leq \epsilon\, n\, \frac{1}{|\alpha_{k-1}|^2}(||\mathbf{r_{k-1}}|| + ||\mathbf{r_k}||)^2 + O(\epsilon^2). \end{aligned} \tag{4.11}$$

Lastly, to bound $\delta_{\nu'_k}$ we first utilize (4.5) together with triple usage of (4.1) to obtain the first inequality below. Then, we use (4.5) again and rewrite the δ terms as $O(\epsilon^2)$, owing to (4.8), (4.9), and (4.11). Finally, we substitute for $||\mathbf{s_{k-1}}||$ from (4.10), add some terms to $O(\epsilon^2)$, and rewrite the expression:

$$\begin{aligned} |\delta_{\nu'_k}| &\leq 3\,\epsilon\,(|\nu_{k-1}| + 2|\alpha_{k-1}||\sigma_{k-1}| + |\alpha_{k-1}|^2|\gamma_{k-1}|) \\ &\leq 3\,\epsilon\,(||\mathbf{r_{k-1}}||^2 + 2|\alpha_{k-1}|||\mathbf{r_{k-1}}||\,||\mathbf{s_{k-1}}|| + |\alpha_{k-1}|^2||\mathbf{s_{k-1}}||^2) + O(\epsilon^2) \\ &\leq 3\,\epsilon\,(||\mathbf{r_{k-1}}||^2 + 2||\mathbf{r_{k-1}}||(||\mathbf{r_{k-1}}|| + ||\mathbf{r_k}||) + (||\mathbf{r_{k-1}}|| + ||\mathbf{r_k}||)^2) + O(\epsilon^2) \\ &= 3\,\epsilon\,(4||\mathbf{r_{k-1}}||^2 + 4||\mathbf{r_{k-1}}||\,||\mathbf{r_k}|| + ||\mathbf{r_k}||^2) + O(\epsilon^2). \end{aligned} \tag{4.12}$$

Now we can substitute (4.7), (4.8), (4.9), (4.11), and (4.12) into

$$|\Delta_{\nu'_k}| \leq 2||\delta_{\mathbf{r_k}}||\,||\mathbf{r_k}|| + ||\delta_{\mathbf{r_k}}||^2 + |\delta_{\nu_{k-1}}| + 2|\alpha_{k-1}||\delta_{\sigma_{k-1}}| + |\alpha_{k-1}|^2|\delta_{\gamma_{k-1}}| + |\delta_{\nu'_k}|,$$

which yields

$$\begin{aligned} |\Delta_{\nu'_k}| &\leq 6\,\epsilon\,(||\mathbf{r_k}|| + ||\mathbf{r_{k-1}}||)||\mathbf{r_k}|| + O(\epsilon^2) + \epsilon\, n\, ||\mathbf{r_{k-1}}||^2 \\ &\quad + 2\,\epsilon\, n\,(||\mathbf{r_{k-1}}||^2 + ||\mathbf{r_{k-1}}||\,||\mathbf{r_k}||) + \epsilon\, n\,(||\mathbf{r_{k-1}}|| + ||\mathbf{r_k}||)^2 \\ &\quad + 3\,\epsilon\,(4||\mathbf{r_{k-1}}||^2 + 4||\mathbf{r_{k-1}}||\,||\mathbf{r_k}|| + ||\mathbf{r_k}||^2) + O(\epsilon^2) \\ &\leq \epsilon\,((12+4n)||\mathbf{r_{k-1}}||^2 + (18+4n)||\mathbf{r_{k-1}}||\,||\mathbf{r_k}|| + (9+n)||\mathbf{r_k}||^2) + O(\epsilon^2) \\ &\leq \epsilon\,((12+4n)||\mathbf{r_{k-1}}||^2 + (9+2n)(||\mathbf{r_{k-1}}||^2 + ||\mathbf{r_k}||^2) + (9+n)||\mathbf{r_k}||^2) \\ &\quad + O(\epsilon^2). \end{aligned}$$

Finally, by dropping terms of order $O(\epsilon^2)$, we can bound the ν-gap as

$$|\nu_k - \nu'_k| = |\Delta_{\nu'_k}| \lesssim \epsilon\,(21+6n)(||\mathbf{r_{k-1}}||^2 + ||\mathbf{r_k}||^2).$$

As was previously mentioned, a similar bound was derived in [1]. The derivation presented here differs in the very last step, where a different algebraic manipulation is utilized. This results in the bound being slightly tighter by having smaller constants than the one in [1].

4.1.2 Numerical experiments

Now that we have derived the ν-bound, it can be investigated whether it is violated if a bit flip occurs. This is the aim of the multi-graphs (Figure 4.1 - Figure 4.4) presented below which depict the behavior of the ν-gap, $|\nu_k - \nu'_k|$, and the ν-bound, $\epsilon\,(21+6n)(||\mathbf{r_{k-1}}||^2 + ||\mathbf{r_k}||^2)$, when a bit is flipped in each variable. The norms $||\mathbf{r_{k-1}}||^2$ and $||\mathbf{r_k}||^2$ were computed using the already calculated $\nu_k = \langle \mathbf{r_k}, \mathbf{r_k} \rangle$ and $\nu_{k-1} = \langle \mathbf{r_{k-1}}, \mathbf{r_{k-1}} \rangle$.

In case of vector variables ($\mathbf{x_k}$, $\mathbf{r_k}$, $\mathbf{w'_k}$, $\mathbf{p_k}$, $\mathbf{s_k}$, $\mathbf{u_k}$, $\mathbf{w_k}$) the bit was always flipped in the 100th position of the vector. Nonetheless, this choice does not destroy diversity of the experiment, since the matrices, and the vector variables as well, are of different sizes.

As for the data of the linear systems, the matrices used were a subsample of those utilized for the sensitivity experiments in Chapter 3. Specifically, we used the following from [21]:

- Matrix *bundle1* $\in \mathbb{R}^{10,581 \times 10,581}$, $\kappa(A) = 1.004238e+03$, $||A|| = 6.428996e+12$,
- Matrix *bcsstm07* $\in \mathbb{R}^{420 \times 420}$, $\kappa(A) = 7.615188e+03$, $||A|| = 2.510397e+03$,
- Matrix *1138_bus* $\in \mathbb{R}^{1,138 \times 1,138}$, $\kappa(A) = 8.572646e+06$, $||A|| = 3.014879e+04$,
- Matrix *nos7* $\in \mathbb{R}^{729 \times 729}$, $\kappa(A) = 2.374510e+09$, $||A|| = 9.864030e+06$.

For the matrices *bundle1* and *bcsstm07* (Figure 4.1 and Figure 4.2) the right-hand side $\mathbf{b}$ was a vector of all ones, i.e., $\mathbf{b} = (1, \cdots, 1)^T =: \mathbf{e}$. For the matrices *1138_bus* and *nos7* (depicted in Figure 4.3 and Figure 4.4), it was $\mathbf{b} = \mathbf{Ae}$. The right-hand sides differ in order to introduce an additional layer of variety into the sample. The initial guess $\mathbf{x_0}$ was always vector of all zeros.

For each figure, the number of the flipped bit and the flip iteration are always displayed at the top. The iteration range of each multi-graph was chosen to reflect how many iterations are approximately needed to converge for the given problem data. When the ν-gap exceeds the ν-bound, its marker symbol changes to a square. The ν-gap can at times be zero. However, this is not displayed in the figures for the sake of their simplicity. Finally, note that many more experimental runs were performed in order to judge the behavior of the ν-gap and the ν-bound. The four multi-graphs presented here are a representative sample. Nonetheless, they include a varied range of different bit numbers, flip iterations, and problem data.

Let us now comment on what we can deduce by investigating the figures. Generally, it can be concluded that ν-gap/bound detection works well when the silent error occurs in $\nu'_k, \sigma_k, \gamma_k$, or ν_k. It can also detect errors in the residual $\mathbf{r_k}$, but not if the flip occurs in the first (sign) bit. This is only logical, as the sign bit is irrelevant for the value of $\nu_k = \langle \mathbf{r_k}, \mathbf{r_k} \rangle$, and therefore the ν-gap is not influenced by flips in it.

We can also observe that the bound is violated even for flips in bits of higher number. During the sensitivity experiments in the third chapter, it was discovered

that flips in bits of number 25 and higher usually do not destroy convergence, as is illustrated by Figure 3.1. However, Figure 4.4 depicts that the ν-bound is, for the four above-mentioned scalar variables, violated even for flips in the 35th bit. This means that once we employ this criterion in practice, it may raise an alarm even in the case of bit flips which have a negligible effect on convergence. Another noteworthy fact is that for $\mathbf{r_k}$ and ν'_k, the bound is violated at the flip iteration, whereas for σ_k and γ_k, the violation happens one iteration later. In case of ν_k, the bound is violated both in the flip iteration and the following iteration. Monitoring when our criteria raise an alarm that a silent error has likely occurred is important for its correction. The idea for this correction is that we keep variables from a number of previous iterations or make some checkpoints, and if the flip is detected immediately we can roll back to a state which should not yet be influenced by the error.

In conclusion, the ν-gap/bound criterion seems to be able to reliably detect flips in $\nu'_k, \sigma_k, \gamma_k$, and ν_k, as well as in non-sign bits for $\mathbf{r_k}$. For other variables a different detection method must be used.

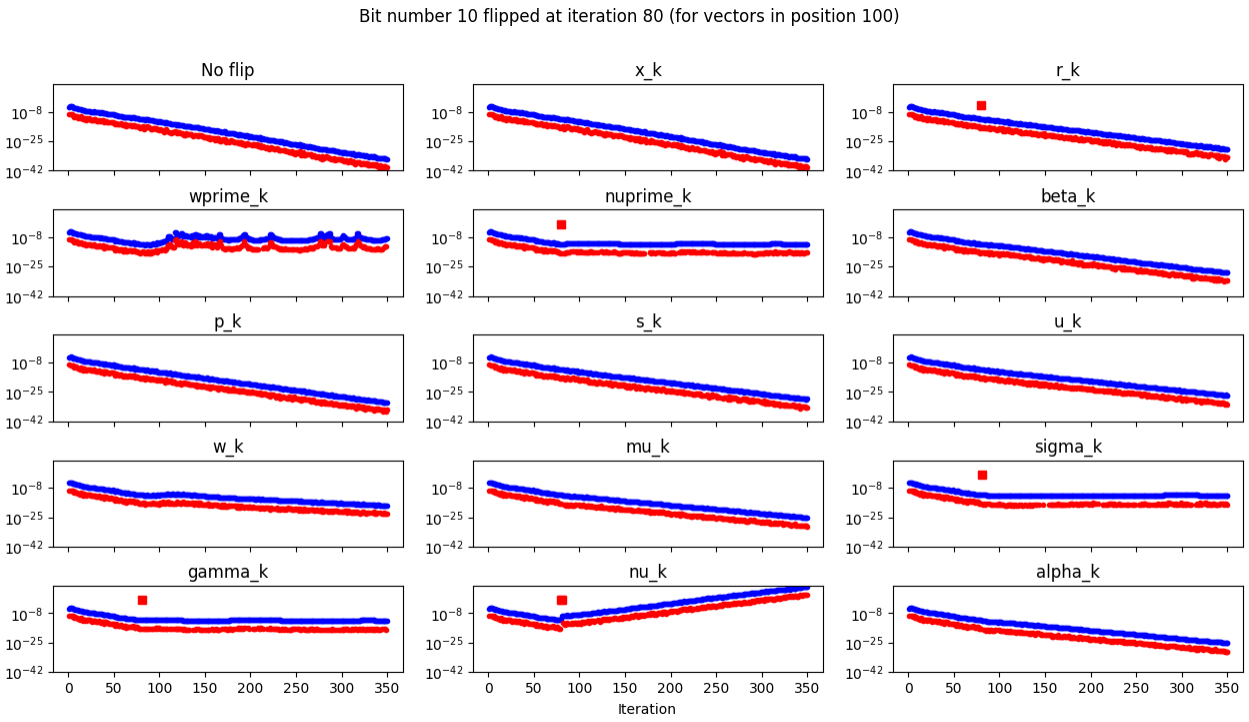

Figure 4.1: ν-gap (red) and ν-bound (blue) graph, matrix *bundle1*

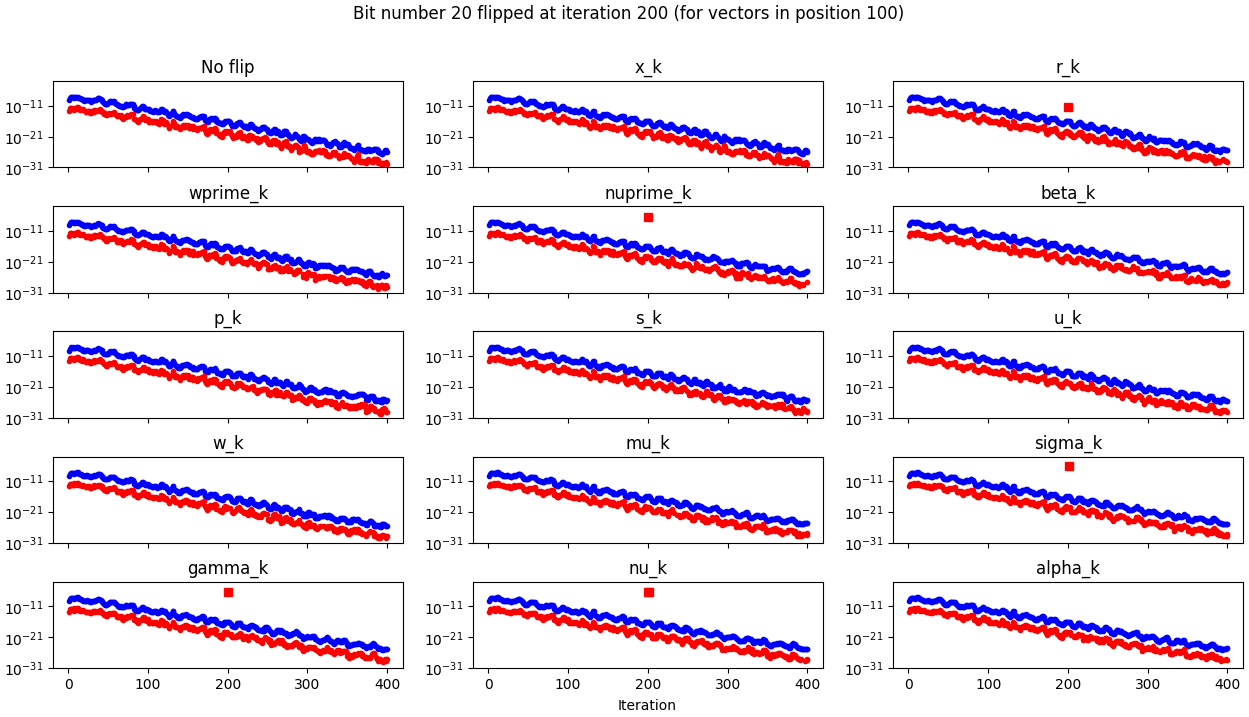

Figure 4.2: ν-gap (red) and ν-bound (blue) graph, matrix *bcsstm07*

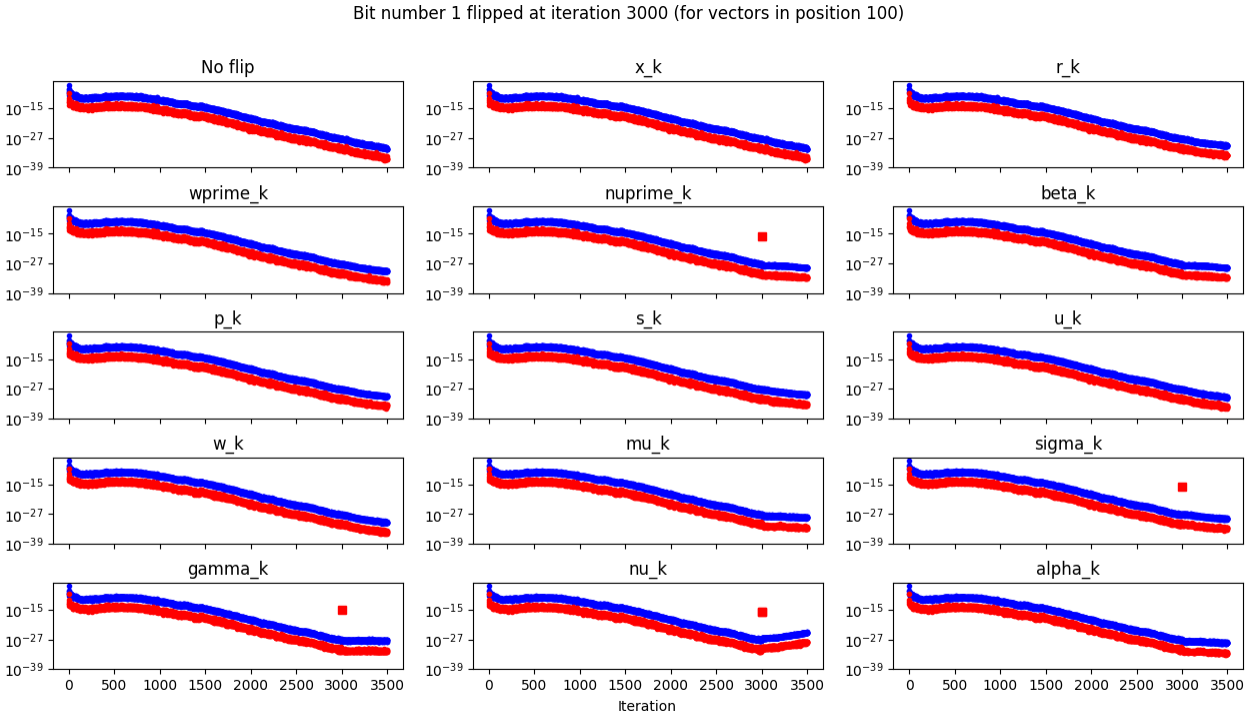

Figure 4.3: ν-gap (red) and ν-bound (blue) graph, matrix *1138_bus*

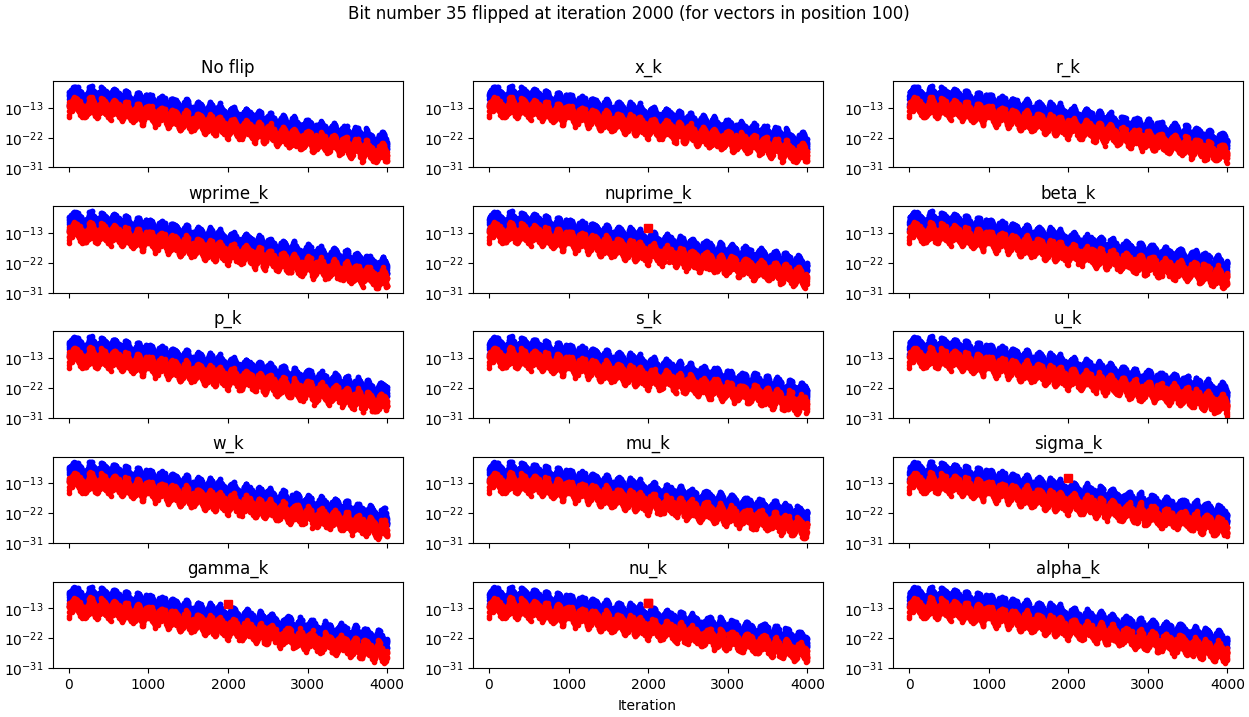

Figure 4.4: ν-gap (red) and ν-bound (blue) graph, matrix *nos7*

4.2 w-gap

In the previous section, we have investigated the efficacy of silent error detection by the ν-gap. It proved to be a useful tool, but, ultimately, it seems to function only for a specific subset of variables. Thus, if we aim to be able to detect flips in all Pipe-PR-CG variables, it is necessary to derive additional detection criteria. At the end of the third chapter, it was mentioned that there are two, first predicted and then recomputed, variables whose gaps we could try to utilize. The first of these was the ν-gap, which we have already thoroughly investigated. The second one was the $\mathbf{w}$-gap, i.e., the difference between the predicted value $\mathbf{w'_k}$ and the recomputed value $\mathbf{w_k}$, which are equal in exact arithmetic. In this section we shall investigate the possibility of using the $\mathbf{w}$-gap for silent error detection in Pipe-PR-CG.

4.2.1 Derivation of bounds

Let us now proceed to deriving a bound for the $\mathbf{w}$-gap. We are going to utilize the same rounding error analysis as we did for the ν-bound. First, we state some relations for variables from Pipe-PR-CG which we shall use in the analysis. With symbols δ once again denoting the rounding errors, it holds that [1]

$$
\begin{aligned}
\mathbf{r_k} &= \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}} + \delta_{\mathbf{r_k}},\\
\mathbf{w'_k} &= \mathbf{w_{k-1}} - \alpha_{k-1}\mathbf{u_{k-1}} + \delta_{\mathbf{w'_k}},\\
\mathbf{u_k} &= \mathbf{A s_k} + \delta_{\mathbf{u_k}},\\
\mathbf{w_k} &= \mathbf{A r_k} + \delta_{\mathbf{w_k}}.
\end{aligned}
\tag{4.13}
$$

Next, we are going to utilize these relations to rewrite the expression for the difference $||\mathbf{w_k} - \mathbf{w'_k}|| =: \Delta_{\mathbf{w'_k}}$, so that we can attempt to bound its terms. This is done in the following way [1]:

$$
\begin{aligned}
\Delta_{\mathbf{w'_k}} &= \mathbf{w_k} - \mathbf{w'_k}\\
&= \mathbf{A r_k} + \delta_{\mathbf{w_k}} - (\mathbf{w_{k-1}} - \alpha_{k-1}\mathbf{u_{k-1}} + \delta_{\mathbf{w'_k}})\\
&= \mathbf{A r_{k-1}} - \alpha_{k-1}\mathbf{A s_{k-1}} + \mathbf{A}\delta_{\mathbf{r_k}} + \delta_{\mathbf{w_k}} - (\mathbf{w_{k-1}} - \alpha_{k-1}\mathbf{u_{k-1}} + \delta_{\mathbf{w'_k}})\\
&= (\mathbf{A r_{k-1}} - \mathbf{w_{k-1}}) - \alpha_{k-1}(\mathbf{A s_{k-1}} - \mathbf{u_{k-1}}) + \mathbf{A}\delta_{\mathbf{r_k}} + \delta_{\mathbf{w_k}} - \delta_{\mathbf{w'_k}}\\
&= -\delta_{\mathbf{w_{k-1}}} + \delta_{\mathbf{w_k}} + \alpha_{k-1}\delta_{\mathbf{u_{k-1}}} + \mathbf{A}\delta_{\mathbf{r_k}} - \delta_{\mathbf{w'_k}}.
\end{aligned}
$$

Subsequently, we can take the norm of both sides to obtain the inequality

$$
||\Delta_{\mathbf{w'_k}}|| \leq ||\delta_{\mathbf{w_{k-1}}}|| + ||\delta_{\mathbf{w_k}}|| + |\alpha_{k-1}|\,||\delta_{\mathbf{u_{k-1}}}|| + ||\mathbf{A}||\,||\delta_{\mathbf{r_k}}|| + ||\delta_{\mathbf{w'_k}}||. \tag{4.14}
$$

Up to this point, the analysis has followed the article [1], but from now on, we proceed independently from it. As was the case for the ν-bound, we are now going to bound individual terms from (4.14), starting with the first three of them, i.e., $\delta_{\mathbf{w_{k-1}}}$, $\delta_{\mathbf{w_k}}$, and $\delta_{\mathbf{u_{k-1}}}$. This can be done by using relations from (4.13) together with (4.4), yielding

$$
||\delta_{\mathbf{w_k}}|| \leq \epsilon\, c\, ||\mathbf{A}||\,||\mathbf{r_k}||, \tag{4.15}
$$

$$
||\delta_{\mathbf{w_{k-1}}}|| \leq \epsilon\, c\, ||\mathbf{A}||\,||\mathbf{r_{k-1}}||, \tag{4.16}
$$

$$
||\delta_{\mathbf{u_{k-1}}}|| \leq \epsilon\, c\, ||\mathbf{A}||\,||\mathbf{s_{k-1}}||, \tag{4.17}
$$

where $c = mn^{1/2}$, for n being the problem dimension, and m being the maximum number of nonzero elements over the rows of $\mathbf{A}$ [1].

Next, from (4.7) we have that:

$$
||\delta_{\mathbf{r_k}}|| \leq 3\,\epsilon\,(||\mathbf{r_k}|| + ||\mathbf{r_{k-1}}||) + O(\epsilon^2). \tag{4.18}
$$

Finally, we shall bound the term $\delta_{\mathbf{w'_k}}$. This can be done by first utilizing (4.13) and (4.2). Then, we use (4.13) again, this time to rewrite the variables inside norms, and, subsequently, we employ the triangle inequality. Finally, we utilize the relations (4.16) and (4.17) to rewrite $\epsilon\,||\delta_{\mathbf{w_{k-1}}}||$ and $\epsilon\,||\delta_{\mathbf{u_{k-1}}}||$ as $O(\epsilon^2)$, and use properties of the 2-norm to factor out $||\mathbf{A}||$:

$$
\begin{aligned}
||\delta_{\mathbf{w'_k}}|| &\leq \epsilon\,(||\mathbf{w_{k-1}}|| + 2|\alpha_{k-1}|\,||\mathbf{u_{k-1}}||)\\
&= \epsilon\,(||\mathbf{A r_{k-1}} + \delta_{\mathbf{w_{k-1}}}|| + 2|\alpha_{k-1}|\,||\mathbf{A s_{k-1}} + \delta_{\mathbf{u_{k-1}}}||)\\
&\leq \epsilon\,(||\mathbf{A r_{k-1}}|| + ||\delta_{\mathbf{w_{k-1}}}|| + 2|\alpha_{k-1}|\,||\mathbf{A s_{k-1}}|| + ||\delta_{\mathbf{u_{k-1}}}||)\\
&\leq \epsilon\,||\mathbf{A}||\,(||\mathbf{r_{k-1}}|| + 2|\alpha_{k-1}|\,||\mathbf{s_{k-1}}||) + O(\epsilon^2).
\end{aligned}
\tag{4.19}
$$

With each term bounded, we can now substitute from (4.15) - (4.19) into (4.14), and then drop terms of order $O(\epsilon^2)$ to obtain the final bound for $||\Delta_{\mathbf{w}'_k}||$ as

$$\begin{aligned}||\Delta_{\mathbf{w}'_k}|| &\leq \epsilon\, c\, ||\mathbf{A}||\, ||\mathbf{r_{k-1}}|| + \epsilon\, c\, ||\mathbf{A}||\, ||\mathbf{r_k}|| + \epsilon\, c\, |\alpha_{k-1}|\, ||\mathbf{A}||\, ||\mathbf{s_{k-1}}|| \\ &\quad + 3\,\epsilon\, ||\mathbf{A}||\, (||\mathbf{r_k}|| + ||\mathbf{r_{k-1}}||) + \epsilon\, ||\mathbf{A}||\, (||\mathbf{r_{k-1}}|| + 2|\alpha_{k-1}|\, ||\mathbf{s_{k-1}}||) + O(\epsilon^2) \\ &\lesssim \epsilon\, ||\mathbf{A}||\, \Big((c+3)||\mathbf{r_k}|| + (c+4)||\mathbf{r_{k-1}}|| + (c+2)\, |\alpha_{k-1}|\, ||\mathbf{s_{k-1}}||\Big).\end{aligned}$$

Unlike the ν-bound, this expression contains two terms, c and $||\mathbf{A}||$, which depend on properties of the matrix $\mathbf{A}$, and which need to be known prior to starting the computation if we wish to utilize this bound for silent error detection. However, this may not always be feasible a priori. Nonetheless, a reasonable estimation of $||\mathbf{A}||$ can be obtained from a few iterations of Pipe-PR-CG itself [24]. Let us also note that an additional inner product is required in the algorithm to compute the norm of the gap, $||\mathbf{w_k} - \mathbf{w'_k}||$.

4.2.2 Numerical experiments

With the $\mathbf{w}$-bound derived, we can now test its efficacy for silent error detection in Pipe-PR-CG as we did for the ν-bound. There is once again a series of multi-graphs, this time depicting the behavior of the $\mathbf{w}$-gap: $||\mathbf{w_k} - \mathbf{w'_k}||$, and the $\mathbf{w}$-bound: $\epsilon\, ||\mathbf{A}||\, ((c+3)||\mathbf{r_k}|| + (c+4)||\mathbf{r_{k-1}}|| + (c+2)\, |\alpha_{k-1}|\, ||\mathbf{s_{k-1}}||)$, when a bit flip occurs in each of the Pipe-PR-CG variables. The norms $||\mathbf{r_k}||$, $||\mathbf{r_{k-1}}||$, and $||\mathbf{s_{k-1}}||$ were computed by taking square roots of $|\nu_k|$ and $|\gamma_k|$, with the absolute value ensuring that the square roots can be taken even if the values of ν_k and γ_k become negative because of the injected bit flip.

The setup of the experiment was the same as for testing the ν-gap/bound. The bit flips in vector variables ($\mathbf{x_k}$, $\mathbf{r_k}$, $\mathbf{w'_k}$, $\mathbf{p_k}$, $\mathbf{s_k}$, $\mathbf{u_k}$, $\mathbf{w_k}$) were always in the 100th index, the initial guess $\mathbf{x_0}$ was always a zero vector, and the right-hand side $\mathbf{b}$ was a vector of all ones for matrices *bundle1* and *bcsstm07* (Figure 4.5, Figure 4.6, Figure 4.10, and Figure 4.11) and $\mathbf{b} = \mathbf{Ae}$ for matrices *1138_bus* and *nos7* (Figure 4.7, Figure 4.8, and Figure 4.9). The first four figures (4.5 - 4.8) depict the exact same systems, flip iterations, and bit numbers which were presented for the ν-bound, so that the results can be compared. However, this time there are also three additional graphs to further illustrate the behavior of the $\mathbf{w}$-gap/bound pair. As in the case of the ν-gap, the marker style of the $\mathbf{w}$-gap changes to a square if it violates the bound.

As for the ν-gap/bound, the presented experiments are only a subsample of a wider test set. Additionally, let us once again repeat what matrices from [21] were used in the experiment, since the norm $||\mathbf{A}||$ appears in the $\mathbf{w}$-bound:

- Matrix *bundle1* $\in \mathbb{R}^{10,581\times 10,581}$, $\kappa(A) = 1.004238e+03$, $||A|| = 6.428996e+12$,
- Matrix *bcsstm07* $\in \mathbb{R}^{420\times 420}$, $\kappa(A) = 7.615188e+03$, $||A|| = 2.510397e+03$,
- Matrix *1138_bus* $\in \mathbb{R}^{1,138\times 1,138}$, $\kappa(A) = 8.572646e+06$, $||A|| = 3.014879e+04$,

- Matrix $nos7 \in \mathbb{R}^{729\times729}$, $\kappa(A) = 2.374510e+09$, $||A|| = 9.864030e+06$.

By inspecting the figures, we observe that the **w**-gap/bound seems to work well for silent error detection in $\mathbf{r_k}$, $\mathbf{w'_k}$, $\mathbf{u_k}$, and $\mathbf{w_k}$. A pleasant surprise is that it is also able to detect flips of the sign bit in the residual vector - something the ν-gap was not able to achieve. However, if we investigate Figure 4.8, we can see that for bits of high number it is not as sensitive as the ν-gap, since there is no change in the **w**-gap behavior after the flip. This was assessed not only by investigating the graphical output, but by comparing the numerical values as well. Nonetheless, as was recalled in the previous section, flips in the 35th bit do not usually hinder convergence.

Moreover, when we test what happens if, for the same data, we flip not the 35th, but the 25th bit, it can be observed that in this case the **w**-gap reacts to it. This is illustrated in Figure 4.9. As it might be a bit difficult to distinguish whether the bound was violated here only by inspecting the graphs, let us add that in this case the **w**-gap exceeded the **w**-bound for $\mathbf{r_k}$, $\mathbf{w'_k}$, and $\mathbf{w_k}$. For $\mathbf{u_k}$, the bound was not violated. On the other hand, for the experiment depicted in Figure 4.10, which has, aside from the bit number, all data same as the run from Figure 4.6, the bound was violated for all four above-mentioned variables, despite the flipped bit being of a high number - 30. The reason for this may be that in the former case the system matrix has a larger norm and condition number. This indicates that the **w**-gap/bound criterion might not function properly for matrices with properties which result in large norm $||\mathbf{A}||$ or the constant c. On the other hand, in Figure 4.11 we can see that, although the matrix *bundle1* has rather large norm, the criterion still works well even for a bit of relatively high number. It is also worth noting that for significant exponent bits, e.g., the 5th bit, the gap and bound can start to act rather wildly, not only for the **w**-gap/bound, but for the already discussed ν-gap/bound as well. Nonetheless, the bound still seems to be violated only for variables where the methods are able to detect a flip. Once again, depending on the variables, the bound is violated either at the flip iteration or in the very next one. Namely, at the flip iteration for $\mathbf{r_k}$ and $\mathbf{w'_k}$, in the next iteration for $\mathbf{u_k}$, and in both for $\mathbf{w_k}$.

In conclusion, the detection method based on monitoring violation of the **w**-bound appears to be functional for $\mathbf{r_k}$, $\mathbf{w'_k}$, $\mathbf{u_k}$, and $\mathbf{w_k}$. However, for $\mathbf{r_k}$ and $\mathbf{u_k}$, it may occasionally be slightly less reliable.

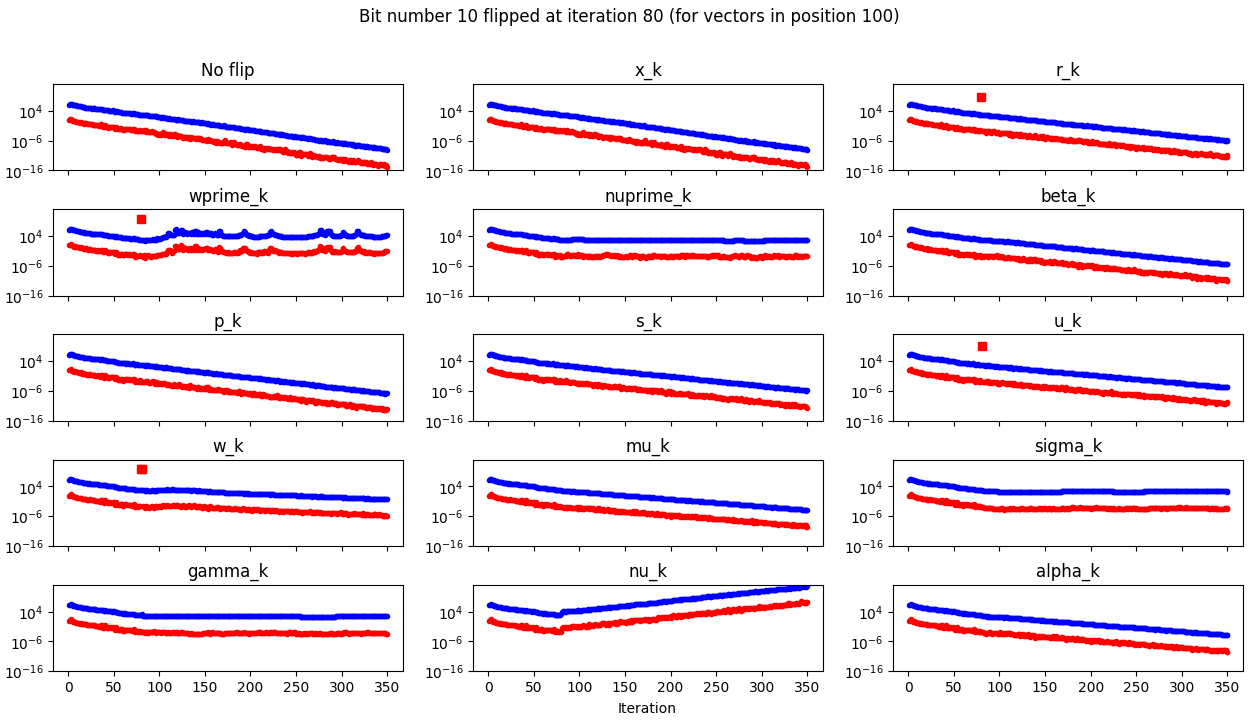

Figure 4.5: **w**-gap (red) and **w**-bound (blue) graph, matrix *bundle1*

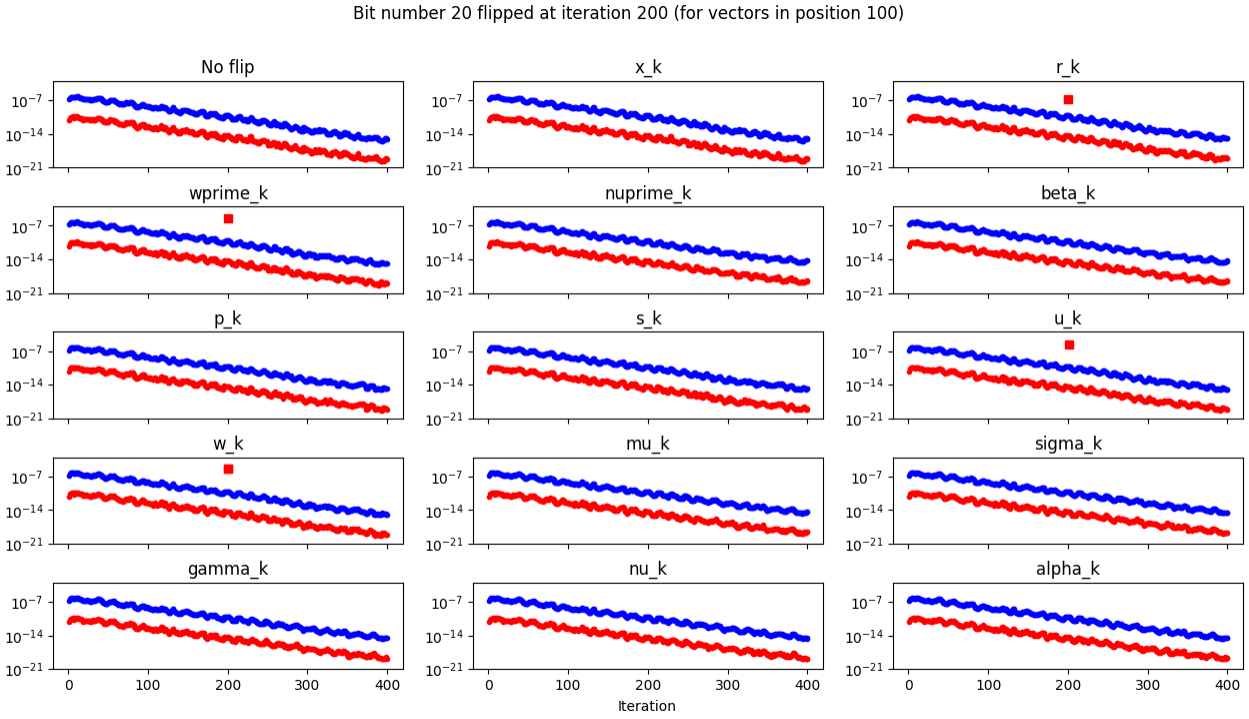

Figure 4.6: **w**-gap (red) and **w**-bound (blue) graph, matrix *bcsstm07*

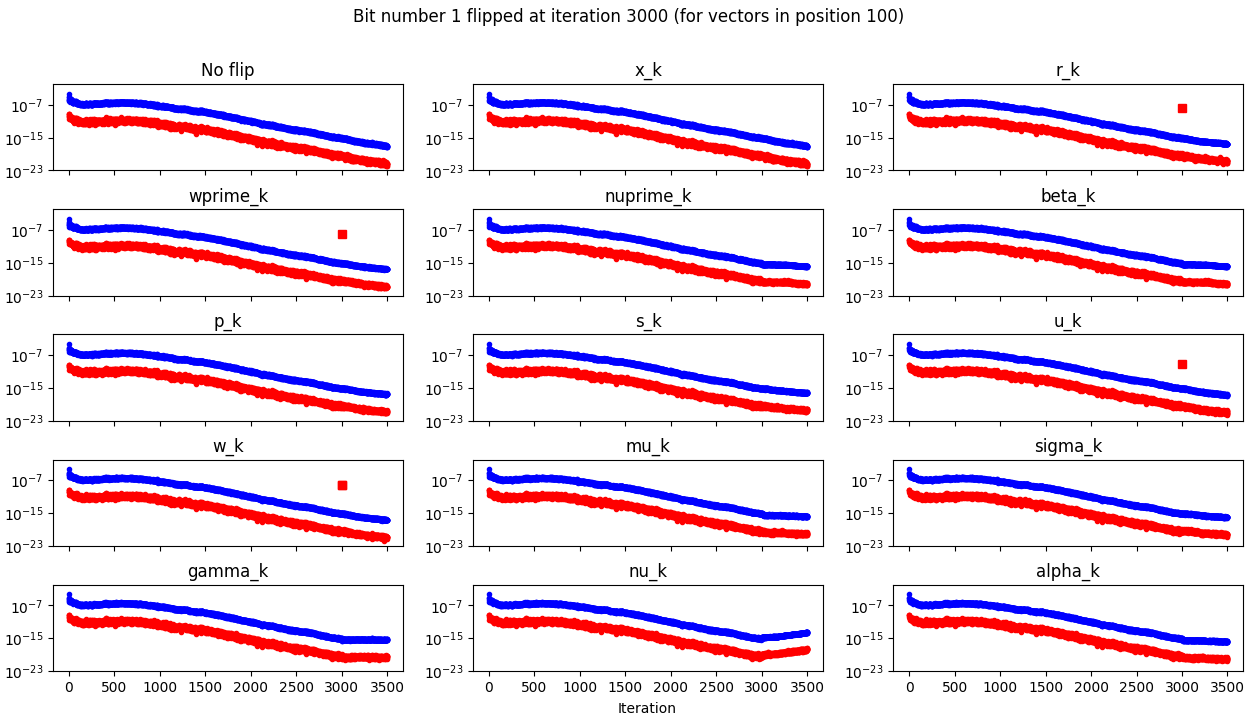

Figure 4.7: **w**-gap (red) and **w**-bound (blue) graph, matrix *1138_bus*

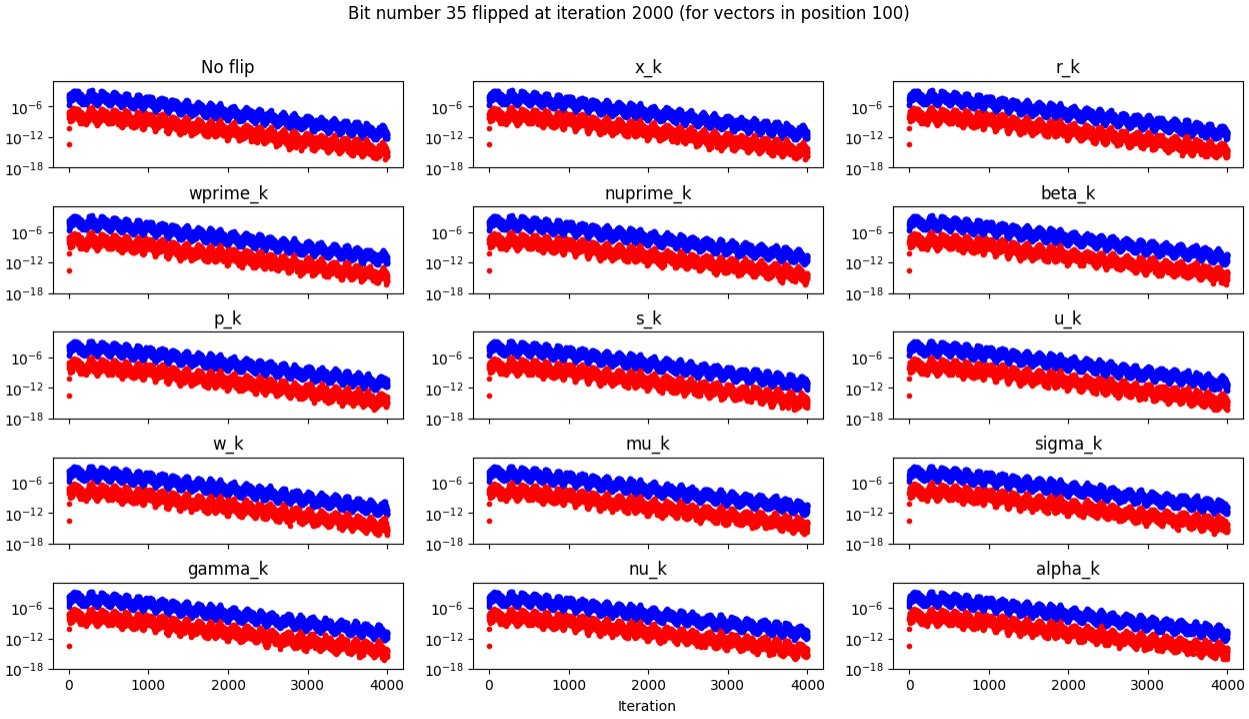

Figure 4.8: **w**-gap (red) and **w**-bound (blue) graph, matrix *nos7*

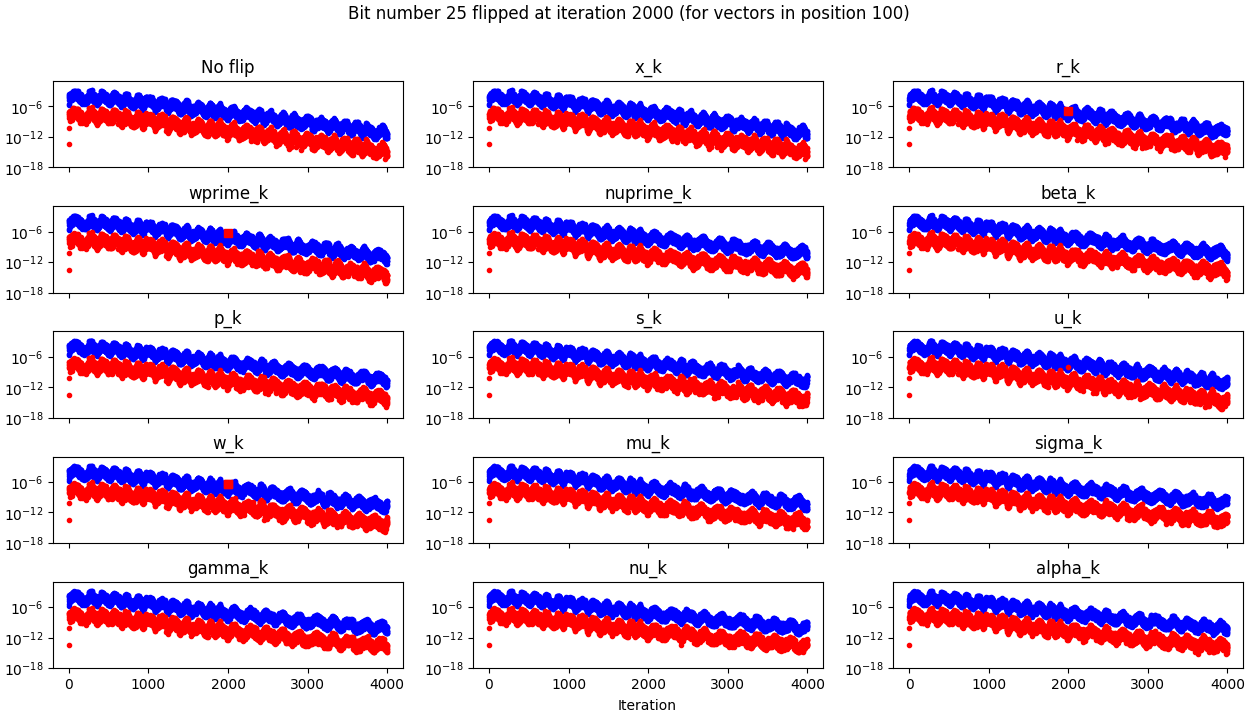

Figure 4.9: **w**-gap (red) and **w**-bound (blue) graph, matrix *nos7*, additional experiment

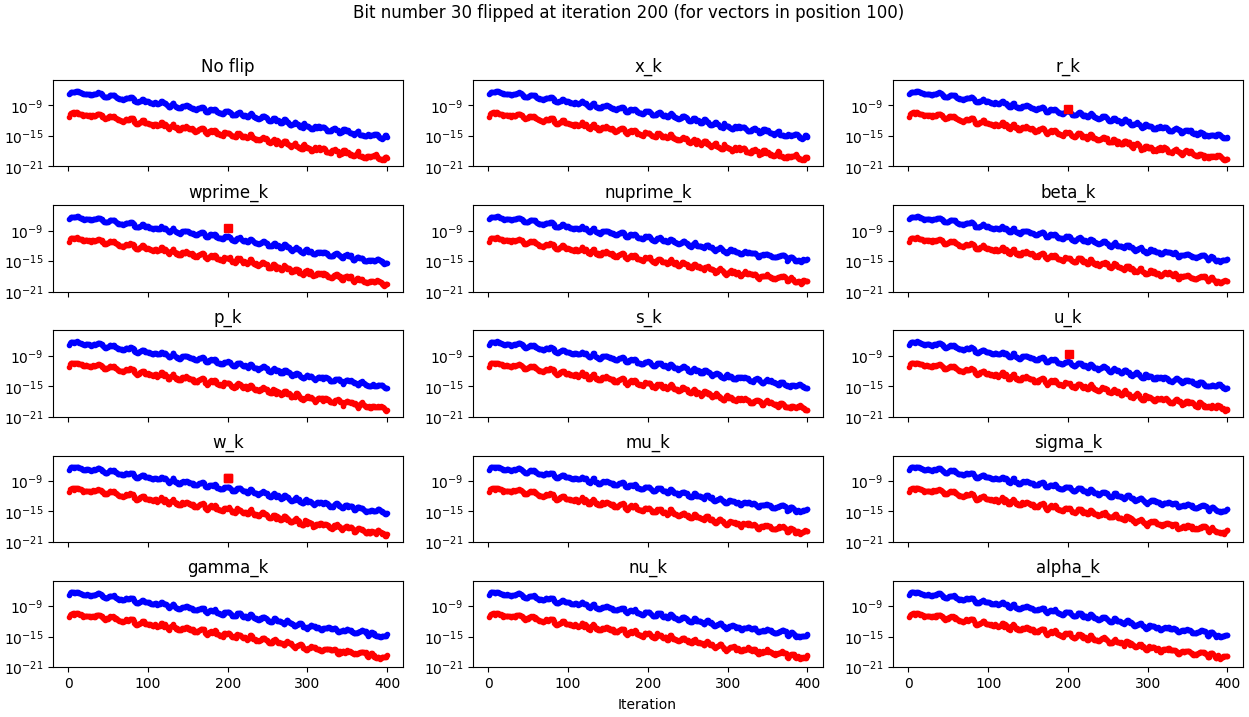

Figure 4.10: **w**-gap (red) and **w**-bound (blue) graph, matrix *bcsstm07*, additional experiment

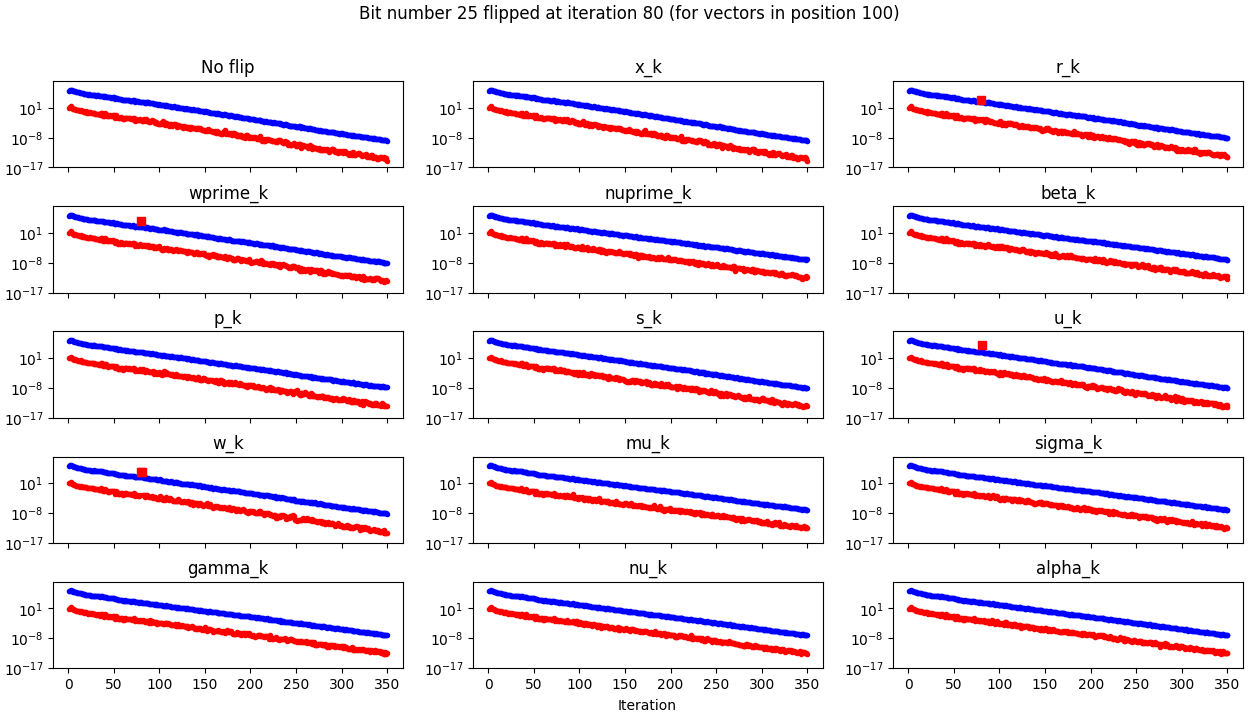

Figure 4.11: **w**-gap (red) and **w**-bound (blue) graph, matrix *bundle1*, additional experiment

4.3 μ-gap

Having investigated both the ν-gap and the **w**-gap, it is apparent that there are still some Pipe-PR-CG variables for which these two detection methods do not work, these are $\mathbf{x_k}$, β_k, $\mathbf{p_k}$, $\mathbf{s_k}$, μ_k, and α_k. Therefore, it is necessary to derive another criterion which is not based on monitoring a difference between the predicted value and the recomputed value of some variable, as at this point we have already utilized all two, respectively four, of them. Fortunately, there are other quantities in the Pipe-PR-CG algorithm which should be equal in exact arithmetic. These are μ_k and σ_k, which are defined as

$$\mu_k = \langle \mathbf{p_k}, \mathbf{s_k} \rangle,$$

and

$$\sigma_k = \langle \mathbf{r_k}, \mathbf{s_k} \rangle,$$

respectively. Their equality in exact arithmetic holds, because, owing to the relation $\mathbf{p_k} = \mathbf{r_k} + \beta_k \mathbf{p_{k-1}}$ from Pipe-PR-CG, we can rewrite μ_k as

$$\mu_k = \langle \mathbf{p_k}, \mathbf{s_k} \rangle = \langle \mathbf{r_k} + \beta_k \mathbf{p_{k-1}}, \mathbf{s_k} \rangle,$$

where the inner product $\langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle$ is equal to zero. This holds because in exact arithmetic we have that $\mathbf{s_k} = \mathbf{A p_k}$, as it originally is in, e.g., HS-CG, and vectors $\mathbf{p_i}$, and $\mathbf{p_j}$ are A-orthogonal for $i \neq j$.

Knowing this, we can define the μ-gap as $|\Delta_{\mu'_k}| \coloneqq |\mu_k - \sigma_k|$, and try to derive a bound for it as we did for the ν-gap and the **w**-gap.

4.3.1 Derivation of bounds

To derive a bound for the μ-gap, we once again utilize rounding error analysis. This time, we are going to use the finite arithmetic relations [1]

$$\begin{aligned} \mathbf{p_k} &= \mathbf{r_k} + \beta_k \mathbf{p_{k-1}} + \delta_{\mathbf{p_k}}, \\ \mu_k &= \langle \mathbf{p_k}, \mathbf{s_k} \rangle + \delta_{\mu_k}, \\ \sigma_k &= \langle \mathbf{r_k}, \mathbf{s_k} \rangle + \delta_{\sigma_k}, \end{aligned} \tag{4.20}$$

where δ once again denotes rounding errors. By utilizing the above expressions, we can rewrite $\Delta_{\mu'_k}$ as

$$\begin{aligned} \Delta_{\mu'_k} = \mu_k - \sigma_k &= \langle \mathbf{p_k}, \mathbf{s_k} \rangle - \langle \mathbf{r_k}, \mathbf{s_k} \rangle + \delta_{\mu_k} - \delta_{\sigma_k} \\ &= \langle \mathbf{r_k} + \beta_k \mathbf{p_{k-1}} + \delta_{\mathbf{p_k}}, \mathbf{s_k} \rangle - \langle \mathbf{r_k}, \mathbf{s_k} \rangle + \delta_{\mu_k} - \delta_{\sigma_k} \\ &= \beta_k \langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle + \langle \delta_{\mathbf{p_k}}, \mathbf{s_k} \rangle + \delta_{\mu_k} - \delta_{\sigma_k}, \end{aligned}$$

which, by taking the norm of both sides, yields that

$$\begin{aligned} |\Delta_{\mu'_k}| &\leq |\beta_k| \, |\langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle| + |\langle \delta_{\mathbf{p_k}}, \mathbf{s_k} \rangle| + |\delta_{\mu_k}| + |\delta_{\sigma_k}| \\ &\leq |\beta_k| \, |\langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle| + ||\delta_{\mathbf{p_k}}|| \, ||\mathbf{s_k}|| + |\delta_{\mu_k}| + |\delta_{\sigma_k}|. \end{aligned} \tag{4.21}$$

Now, we shall bound some of the terms above. First, by using (4.3) and (4.20), $|\delta_{\mu_k}|$ and $|\delta_{\sigma_k}|$ can be bounded as

$$|\delta_{\mu_k}| \leq \epsilon \, n \, (||\mathbf{p_k}|| \, ||\mathbf{s_k}||), \tag{4.22}$$

$$|\delta_{\sigma_k}| \leq \epsilon \, n \, (||\mathbf{r_k}|| \, ||\mathbf{s_k}||), \tag{4.23}$$

and, similarly, owing to (4.2) and (4.20), we can derive that

$$||\delta_{\mathbf{p_k}}|| \leq \epsilon \, (||\mathbf{r_k}|| + 2|\beta_k| \, ||\mathbf{p_{k-1}}||). \tag{4.24}$$

Unfortunately, to the author's knowledge, there is no method of rewriting the term $\langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle$ in a way which does not involve norms of $\mathbf{p_{k-1}}$, $\mathbf{s_k}$, or some other variable, as this would greatly diminish tightness of the bound. Therefore, we must keep it in the overall bound of the μ-gap as it is. Thus, by keeping this term and substituting (4.22), (4.23), and (4.24) into (4.21) we obtain that

$$\begin{aligned} |\Delta_{\mu'_k}| &\leq |\beta_k| \, |\langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle| + \epsilon(||\mathbf{r_k}|| + 2|\beta_k| \, ||\mathbf{p_{k-1}}||)||\mathbf{s_k}|| \\ &\quad + \epsilon \, n \, (||\mathbf{p_k}|| \, ||\mathbf{s_k}||) + \epsilon \, n \, (||\mathbf{r_k}|| \, ||\mathbf{s_k}||) \\ &\leq |\beta_k| \, |\langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle| + \epsilon \, ||\mathbf{s_k}|| \Big(||\mathbf{r_k}|| + 2|\beta_k| \, ||\mathbf{p_{k-1}}|| + n \, (||\mathbf{p_k}|| + ||\mathbf{r_k}||) \Big). \end{aligned}$$

For this expression, two additional inner products are needed in each iteration to compute $\langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle$ and $||\mathbf{p_k}||$. There is no need to compute the norm $||\mathbf{p_{k-1}}||$ as we can simply keep its value from the previous iteration (or initialization). For future usage, let $B_{\mu'_k}$ denote the derived bound.

4.3.2 Numerical experiments

With the μ-bound derived, it is now time to test its efficacy for silent error detection in Pipe-PR-CG. Once again, a series of multi-graphs is presented. However, this time the experimental section has two halves. In the first part, we examine figures depicting the μ-gap:

$$|\Delta_{\mu'_k}| = |\mu_k - \sigma_k|,$$

and the μ-bound:

$$B_{\mu'_k} := |\beta_k|\,|\langle \mathbf{p_{k-1}}, \mathbf{s_k}\rangle| + \epsilon\,||\mathbf{s_k}||\Big(||\mathbf{r_k}|| + 2|\beta_k|\,||\mathbf{p_{k-1}}|| + n\,(||\mathbf{p_k}|| + ||\mathbf{r_k}||)\Big).$$

In the second part, their relative difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, is examined. The analyses and the conclusions are also divided, with the text concerning the first part being below before the first set of figures, and the analysis of the second half dividing the two figure types mentioned above; specifically, it is situated after Figure 4.17.

The norms $||\mathbf{r_k}||$ and $||\mathbf{s_k}||$ in $B_{\mu'_k}$ were again computed by taking square roots of $|\nu_k|$ and $|\gamma_k|$. For $|\langle \mathbf{p_{k-1}}, \mathbf{s_k}\rangle|$ and $||\mathbf{p_k}||$ additional inner products not appearing in the Pipe-PR-CG algorithm had to be computed.

The setup for both of these parts was very similar to what was presented for the ν-gap and the $\mathbf{w}$-gap. For all runs the initial guess was a vector of all zeros, flips for vectors variables ($\mathbf{x_k}$, $\mathbf{r_k}$, $\mathbf{w'_k}$, $\mathbf{p_k}$, $\mathbf{s_k}$, $\mathbf{u_k}$, $\mathbf{w_k}$) always occurred in the 100th index, and the system matrices were the same ones we had utilized throughout this chapter, i.e.,

- Matrix *bundle1* $\in \mathbb{R}^{10,581\times 10,581}$, $\kappa(A) = 1.004238e+03$, $||A|| = 6.428996e+12$,
- Matrix *bcsstm07* $\in \mathbb{R}^{420\times 420}$, $\kappa(A) = 7.615188e+03$, $||A|| = 2.510397e+03$,
- Matrix *1138_bus* $\in \mathbb{R}^{1,138\times 1,138}$, $\kappa(A) = 8.572646e+06$, $||A|| = 3.014879e+04$,
- Matrix *nos7* $\in \mathbb{R}^{729\times 729}$, $\kappa(A) = 2.374510e+09$, $||A|| = 9.864030e+06$,

from the SuiteSparse matrix collection [21]. As for the right-hand side $\mathbf{b}$, it was $\mathbf{b} = \mathbf{e}$ for Figures 4.12, 4.13, 4.17, 4.18, 4.19, 4.20, 4.24, and 4.25. For the other graphs, i.e., Figures 4.14, 4.15, 4.16, 4.21, 4.22, and 4.23, it was $\mathbf{b} = \mathbf{Ae}$. This information is mentioned in the analyses at points where it holds significance. Let us also note that for all gap/bound figures so far, the subplots shared a common y-axis, so that they could be easily compared. However, for Figure 4.12 this is violated as it would, for most variables, disturb visibility of the curve behavior.

The exact same four run sample presented for the ν-gap and the $\mathbf{w}$-gap based methods is here included in Figures 4.12 to 4.15 for the μ-gap and the μ-bound as separate quantities, and in Figures 4.19 to 4.22 for the relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$.

Now, let us comment on what can be deduced by investigating the first group of multi-graphs depicting the μ-gap and the μ-bound side by side. As was done

before, if the bound is violated the gap marker turns into a square. However, this time the square is in addition green in order to be easily visible, since for the μ-gap it is always next to a cluster of other gap values.

The first thing we observe is that the μ-gap and bound are sensitive to flips in almost all variables. Another interesting fact is that the μ-bound for many variables seemingly vanishes after the flip. However, this is caused by it being very close to the μ-gap. The similarity is in most cases so strong that even different visualization styles than the one used here have trouble distinguishing the values. It is also peculiar that neither the gap nor the bound return to their original level, but instead the values are permanently affected by the flip. This was not the case for the ν- and the $\mathbf{w}$-gaps and bounds. The reason for the μ-bound blow-up is the inner product $|\langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle|$, as can be seen by inspecting Figure 4.17, which shows curves for each inner product/norm in $B_{\mu'_k}$. The data of the run presented in this figure are the same as for Figure 4.12.

As for in which variables the μ-gap/bound method is able to detect flips, it seem to be that the only ones are $\mathbf{p_k}$, μ_k, and σ_k. Only in the case of $\mathbf{p_k}$ in Figure 4.15 the flip was not detected, and the μ-gap and the μ-bound were influenced rather moderately. However, this is the 35th bit whose alteration would most likely not prevent convergence. Moreover, as can be seen by inspecting Figure 4.16, once we, for the same data, instead flip the 20th bit, the silent error is detected. For all three above-mentioned variables, the bound is always violated at the flip iteration. It is also interesting that the reason for this is that for these variables the μ-gap "jumps" one iteration earlier than the μ-bound. For all other variables this happens concurrently, either at the flip iteration or at the very next one. As a final remark, let us also add that, as was the case for the ν-gap, the μ-gap can sometimes be zero.

In conclusion, the μ-gap/bound criterion seems to be well applicable for silent error detection in $\mathbf{p_k}$, μ_k, and σ_k. However, the quantities are sensitive to flips for other variables as well.

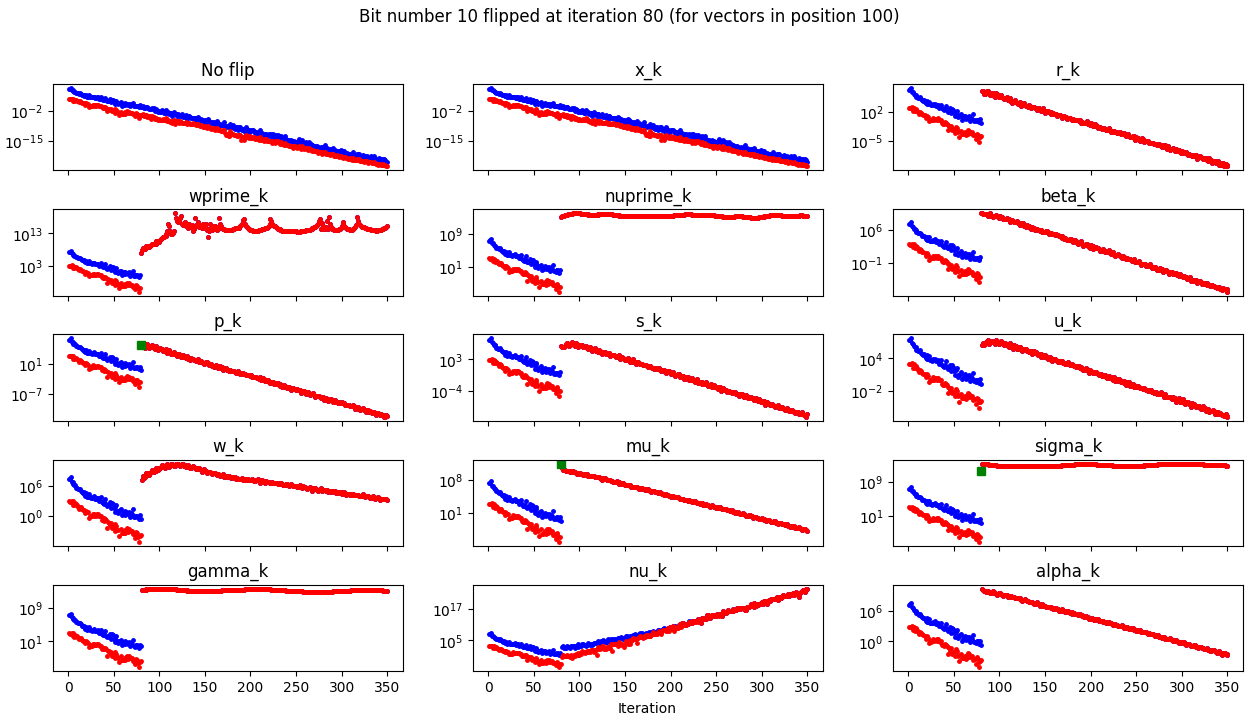

Figure 4.12: μ-gap (red) and μ-bound (blue) graph, matrix *bundle1*

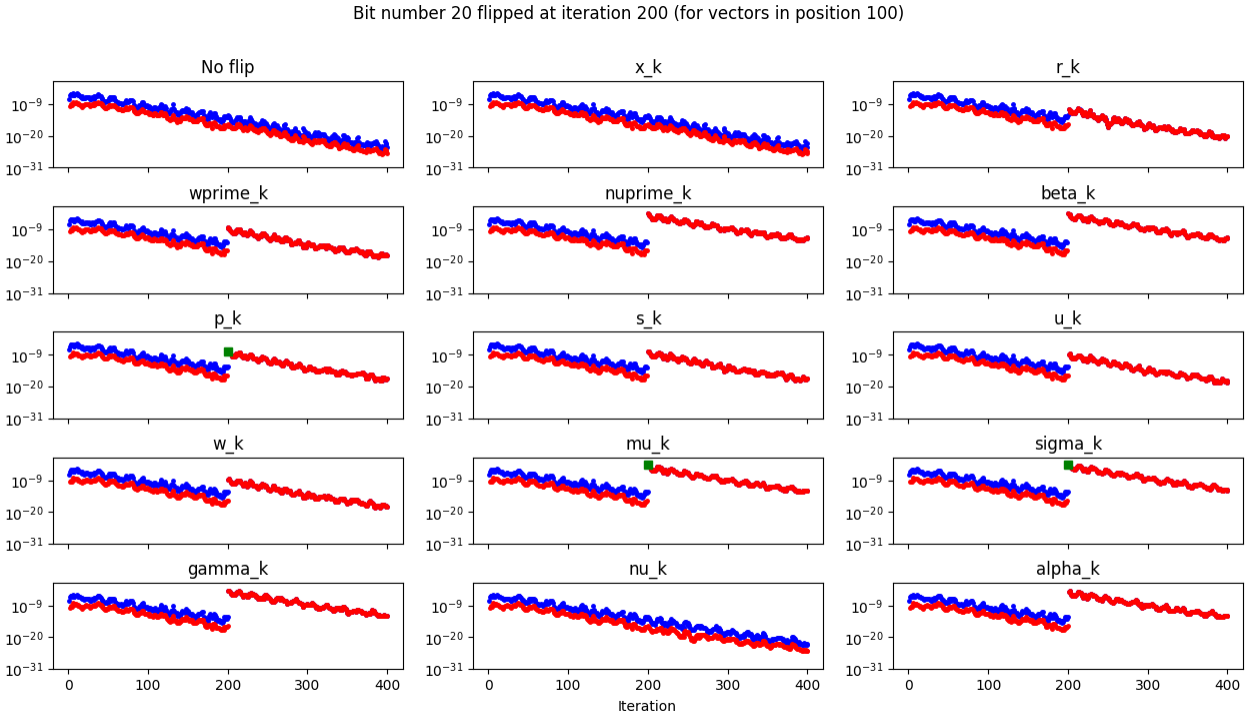

Figure 4.13: μ-gap (red) and μ-bound (blue) graph, matrix *bcsstm07*

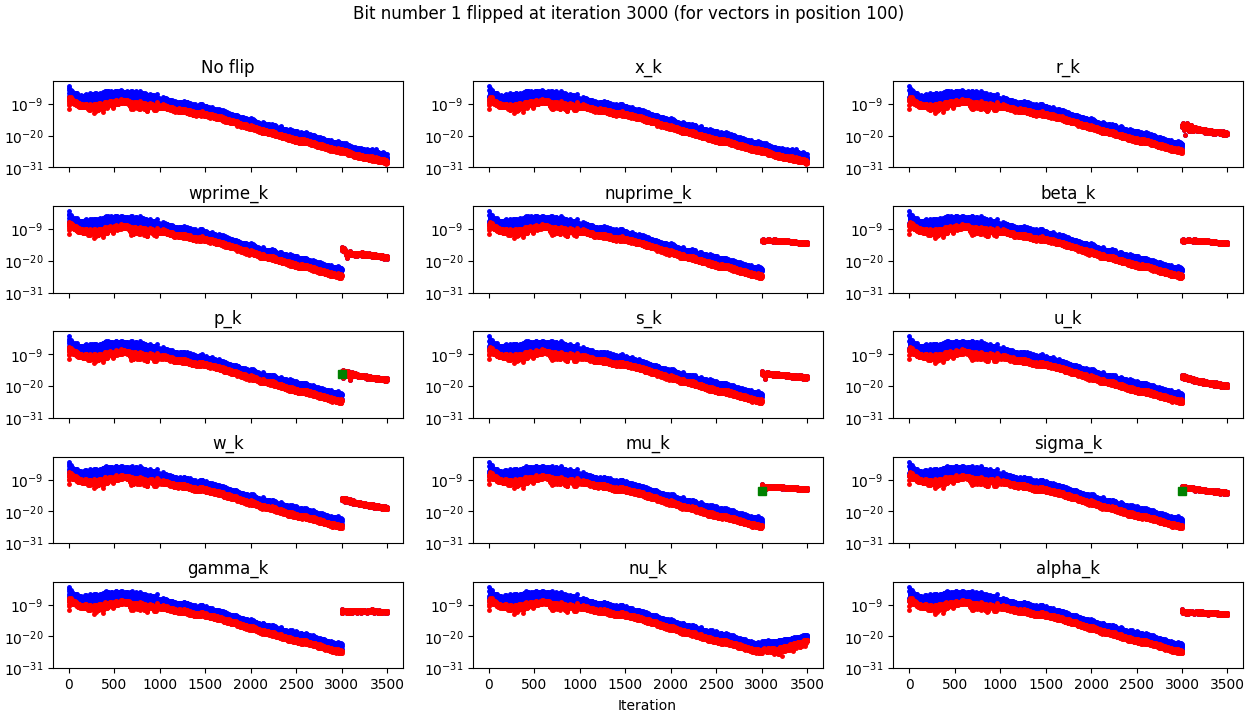

Figure 4.14: μ-gap (red) and μ-bound (blue) graph, matrix *1138_bus*

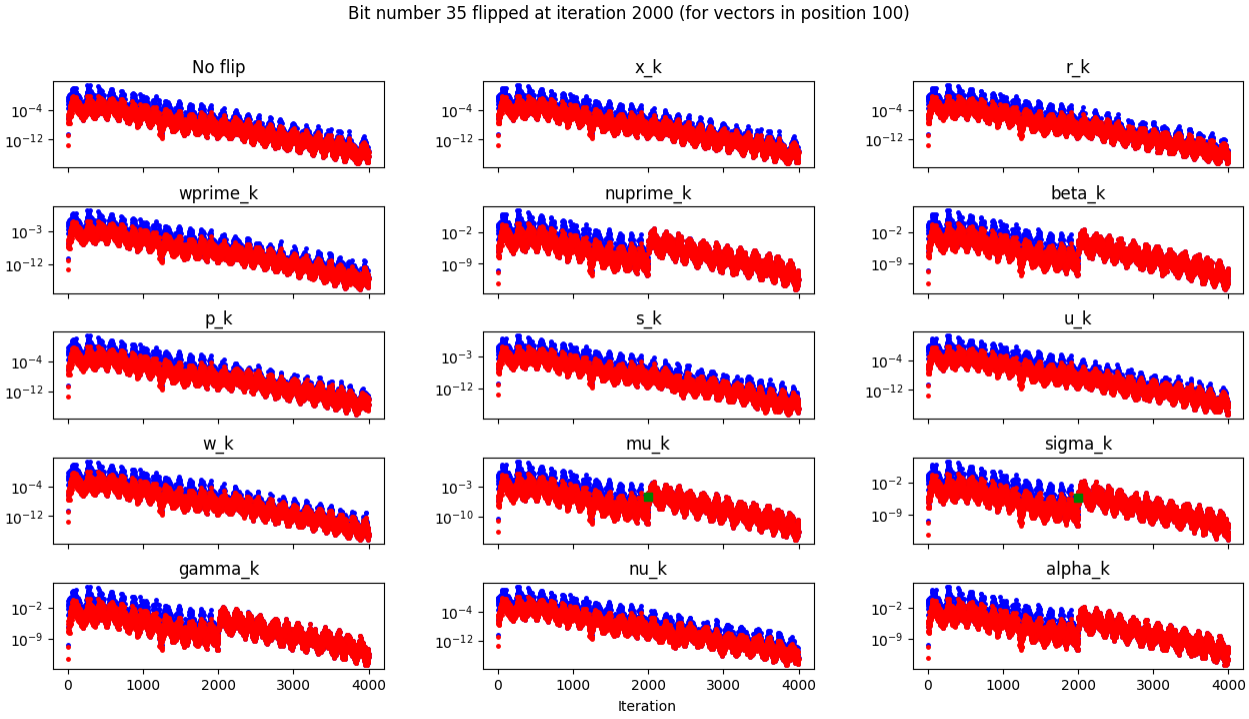

Figure 4.15: μ-gap (red) and μ-bound (blue) graph, matrix *nos7*

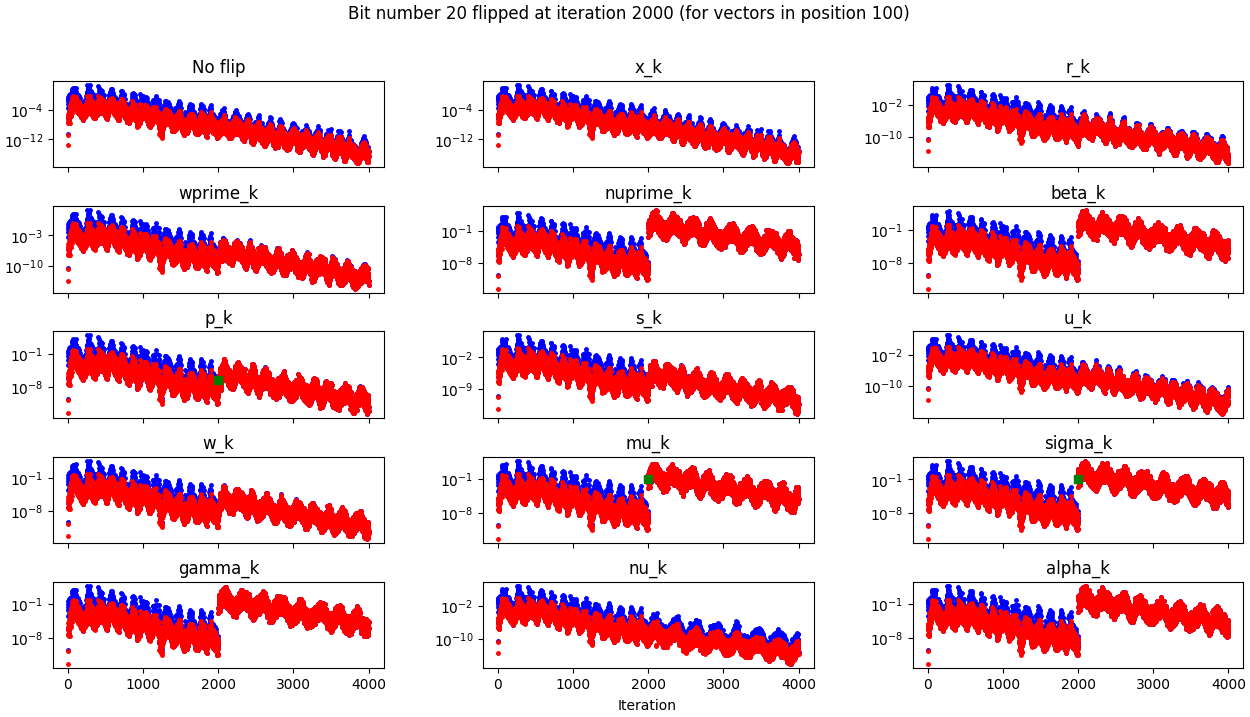

Figure 4.16: μ-gap (red) and μ-bound (blue) graph, matrix *nos7*

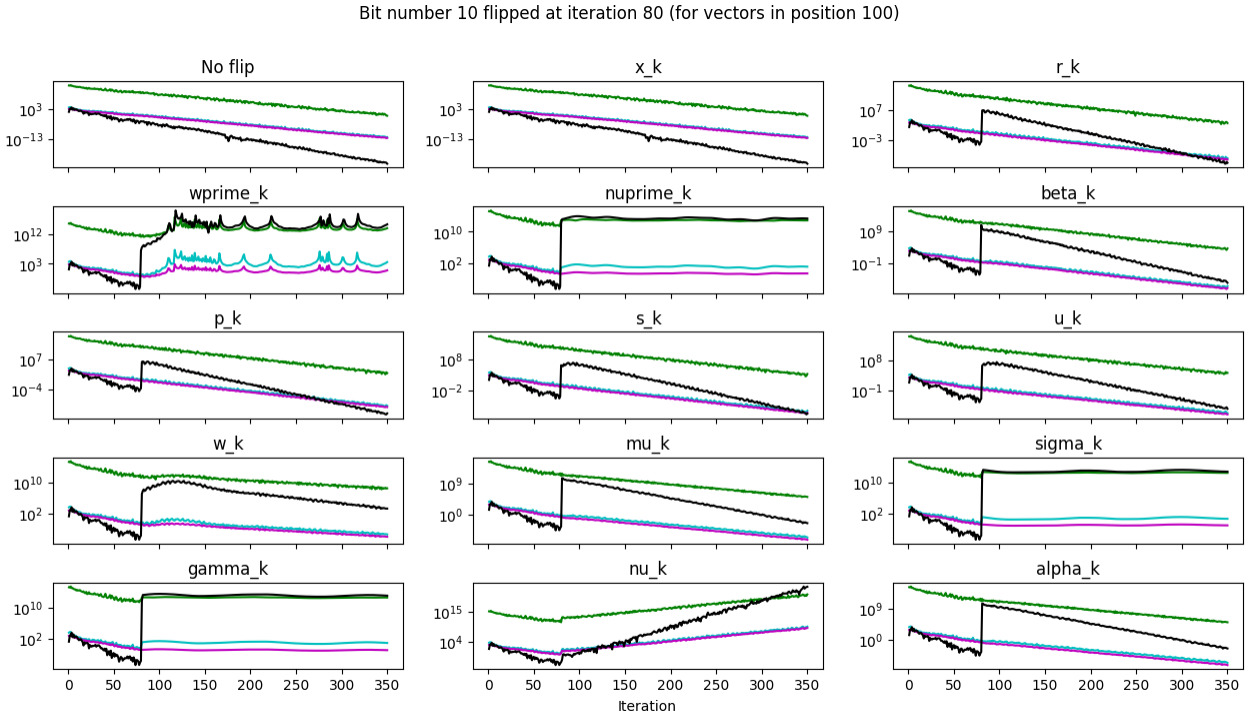

Figure 4.17: Blow-up of the μ-bound, matrix *bundle1*: $||\mathbf{p_k}||$ cyan, $||\mathbf{s_k}||$ green, $||\mathbf{r_k}||$ purple, $|\langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle|$ black

The problem we are now facing is that there are still some variables for which we do not possess a detection method, namely $\mathbf{x_k}$, β_k, $\mathbf{s_k}$, and α_k. However, a straightforward comparison of the gap values and the bound values is not the only way the detection can be done. As was mentioned before, and as can be seen by investigating Figures 4.12 - 4.16, the μ-gap and the μ-bound are influenced by flips in almost all variables. On top of that, their values after the flip become very close. Therefore, we could try to construct a detection method based on the difference of the μ-gap and the μ-bound. However, it somehow surprisingly turns out that their absolute difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|$, steadily decreases despite flips (with the exception of a single jump for $\mathbf{p_k}$, μ_k, and σ_k). This is illustrated in Figure 4.18 which depicts the absolute difference of the μ-bound and the μ-gap from Figure 4.13. Thus, we try to employ the relative difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, instead.

There are two reasons for "normalizing" the difference by the μ-bound and not by the μ-gap. The first one is that the μ-gap can sometimes be zero. The second one is that when no flips occur, the μ-bound is guaranteed to be larger than the μ-gap. Therefore, the ratio is going to be "normalized" better.

Now, we shall investigate the multi-graphs (Figures 4.19 - 4.25) depicting this relative μ-gap/bound difference. The cyan diamond markers show values in the flip iteration and in the one iteration after, i.e., when we would like to be able to detect the flip, so that we can roll back to a close previous state when the variables were still unaffected. The first four figures (4.19 - 4.22) depict our classical problems which we have encountered in the ν-gap and $\mathbf{w}$-gap sections. Then Figure 4.23 has the same data as the graph before it, i.e., $\mathbf{b} = \mathbf{Ae}$, but the 20th bit is flipped instead of the 35th bit. Subsequently, Figure 4.24 depicts flipping of the 20th bit for matrix *nos7* again, but this time with a right-hand side $\mathbf{b} = \mathbf{e}$. The figures are then concluded with a multi-graph of the 20th bit flipped for *1138_bus* with right-hand side $\mathbf{b} = \mathbf{e}$, which is, aside from the chosen matrix, the same problem as in the figure before. (Note that in this case, the flip is in the 2000th iteration instead of the 3000th as was usually done for this matrix).

At first glance, we immediately observe that for all matrices other than *nos7* the effect of the flip is quite significant for all variables besides ν_k. Either at the iteration of the flip, one later, or both, there is a significant jump of the value. On top of that, for the three variables ($\mathbf{p_k}$, μ_k, σ_k), where we were able to detect flips just by the bound violation, the μ-gap/bound relative difference is greater than 1 at the flip iteration, a clear indication that a flip has occurred. Unfortunately, for matrix *nos7* this criterion does not seem to work particularly well for any vector variable other than $\mathbf{p_k}$. Moreover, when the 35th bit is flipped (Figure 4.22) the vector variables, including $\mathbf{p_k}$, are influenced by it only to a very limited degree. For the 20th bit (Figures 4.23 and 4.24) the values are altered, but it can be observed that the diamond markers still largely remain in the same value range as when no bits were flipped. The only exceptions are the aforementioned $\mathbf{p_k}$ and, luckily, partly also $\mathbf{s_k}$. For $\mathbf{s_k}$ we can observe that the diamond markers are, in Figure 4.23, in the "no-flip range", but in Figure 4.24 one of them is slightly below it. This is a crucial observation, since we do not yet posses a detection method for $\mathbf{s_k}$.

When it comes to this approach it is also important to investigate at what level the values of the studied ratio $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$ are when no flips occur, because we need to set some threshold to determine whether to raise an alarm that a silent error has likely appeared or not. If we set the threshold too close to 1 we might get a lot of false positive detections. On the other hand, setting it too low might result in a lot of false negatives. For matrices *bundle1* and *bcsstm07* the values lie very close to 1. For the matrix *1138_bus* (Figures 4.21 and 4.25) the range is a bit more wide, but still quite close to 1. A potential problem is visible if we investigate Figures 4.22 - 4.24, which depict runs for the matrix *nos7*. There, the values are as low as 1e−4. This indicates that choosing a suitable value of the threshold could be a rather complex and data dependent problem. The reason for the values lying in such a wide range, as well as the detection being less reliable, in case of the matrix *nos7* is most likely its high condition number.

In conclusion, the greatest strength of the relative μ-gap/bound difference approach is that it encompasses almost all of the Pipe-PR-CG variables, albeit with some above-mentioned data-related exceptions. However, a disadvantage is that, unlike in the case of the bound violation methods, there is nothing to directly compare the values to, so we have to set some detection threshold.

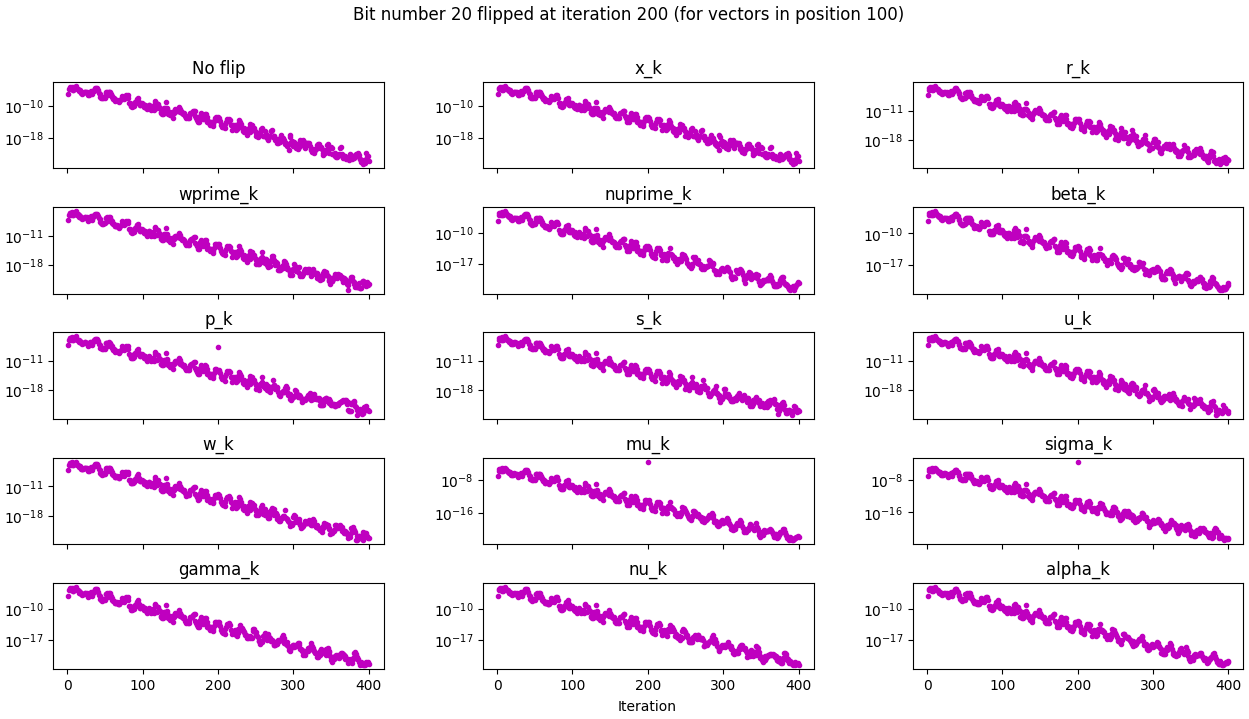

Figure 4.18: Absolute μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|$, matrix *bcsstm07*

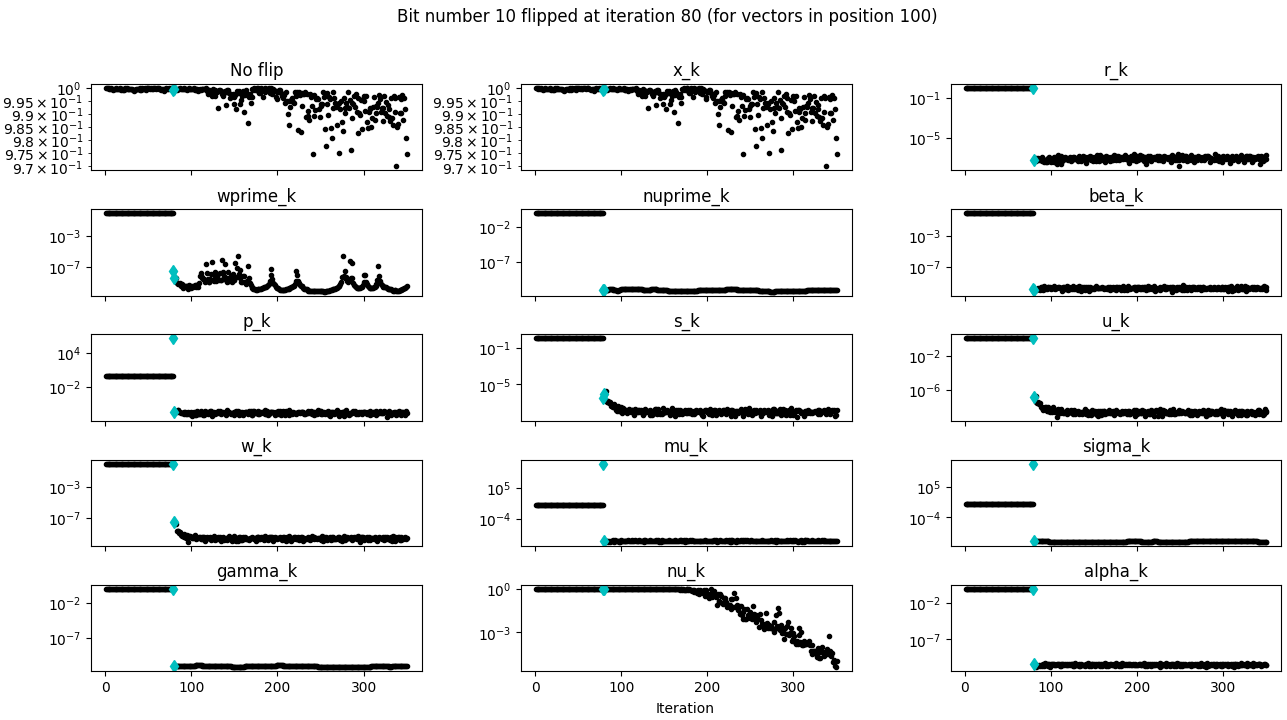

Figure 4.19: Relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, matrix *bundle1*

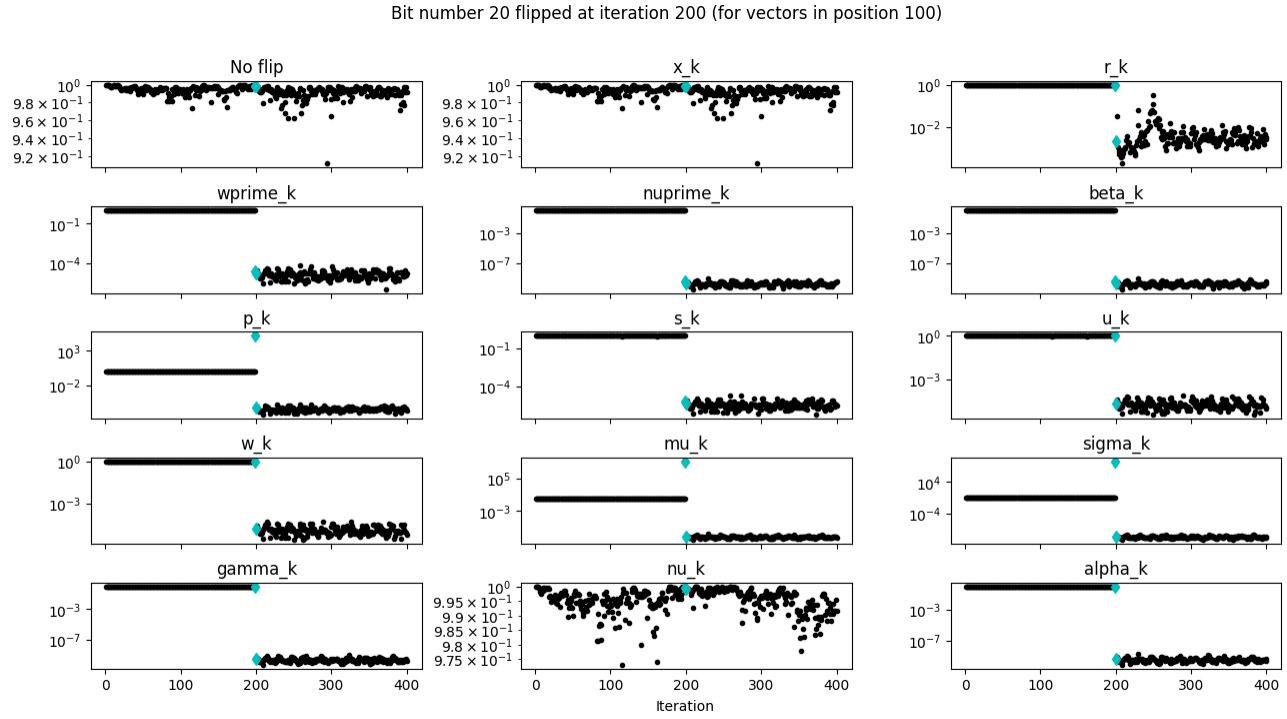

Figure 4.20: Relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, matrix *bcsstm07*

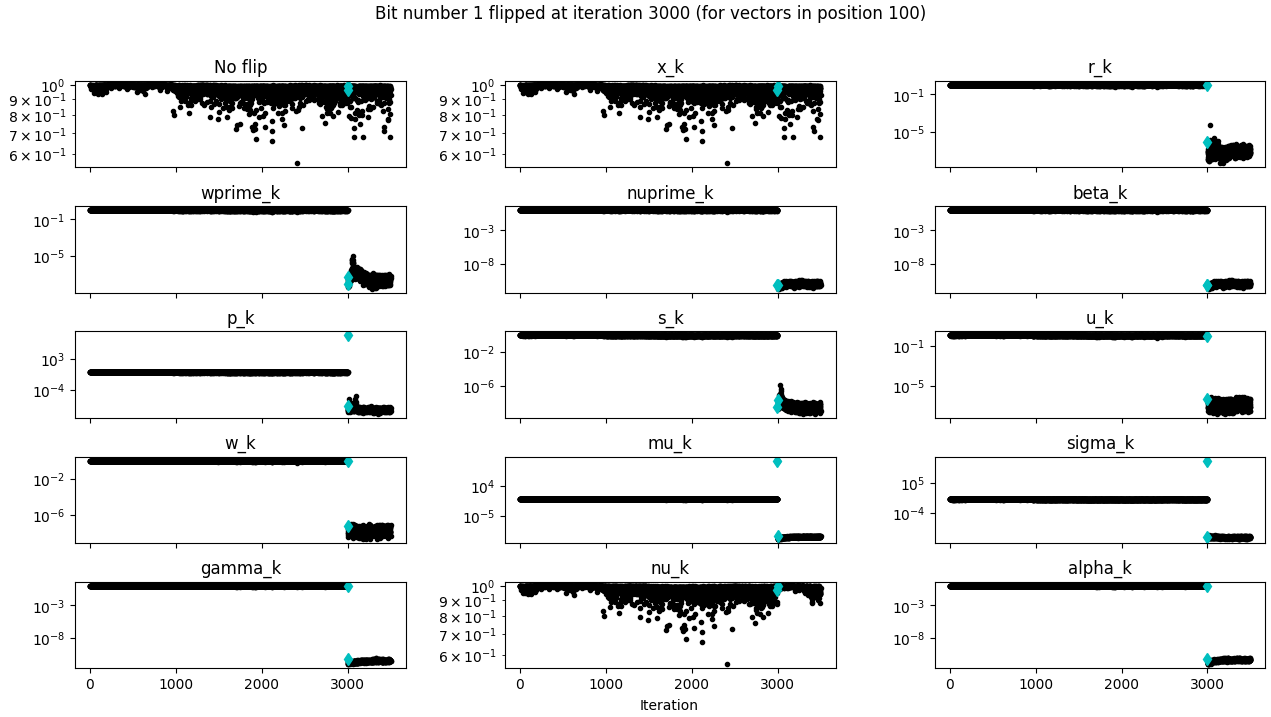

Figure 4.21: Relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, matrix *1138_bus*

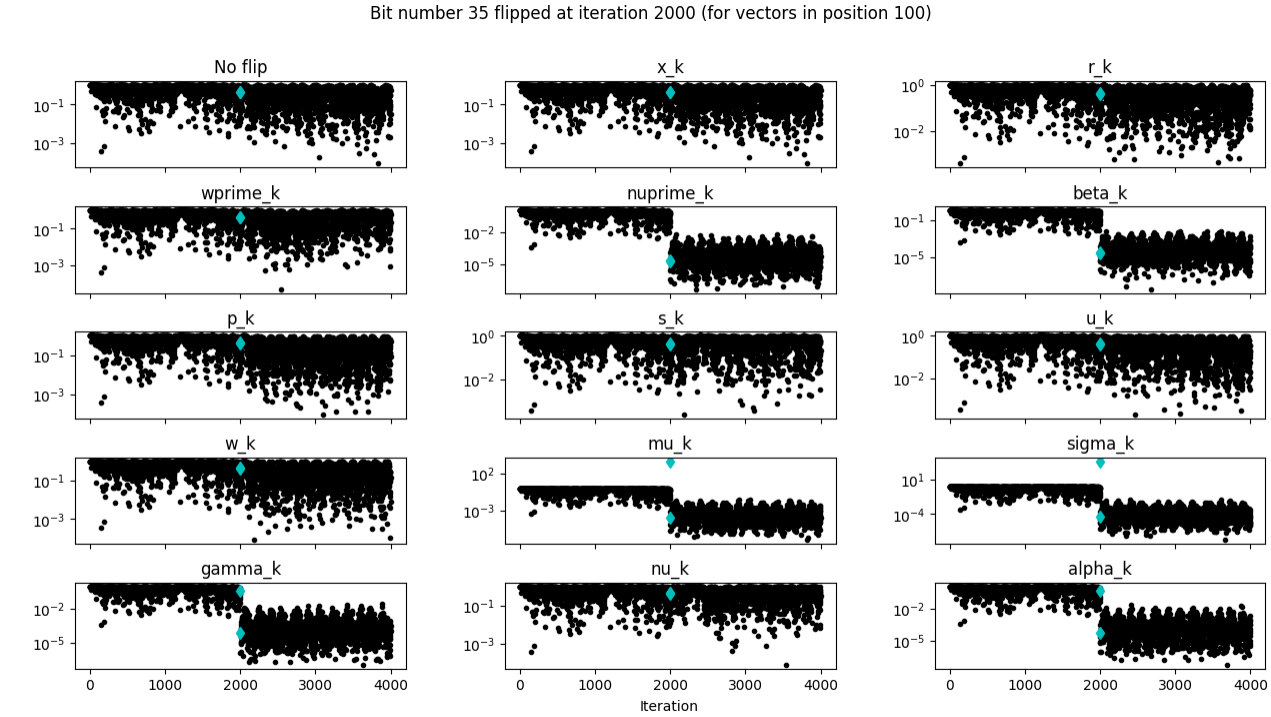

Figure 4.22: Relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, matrix *nos7*

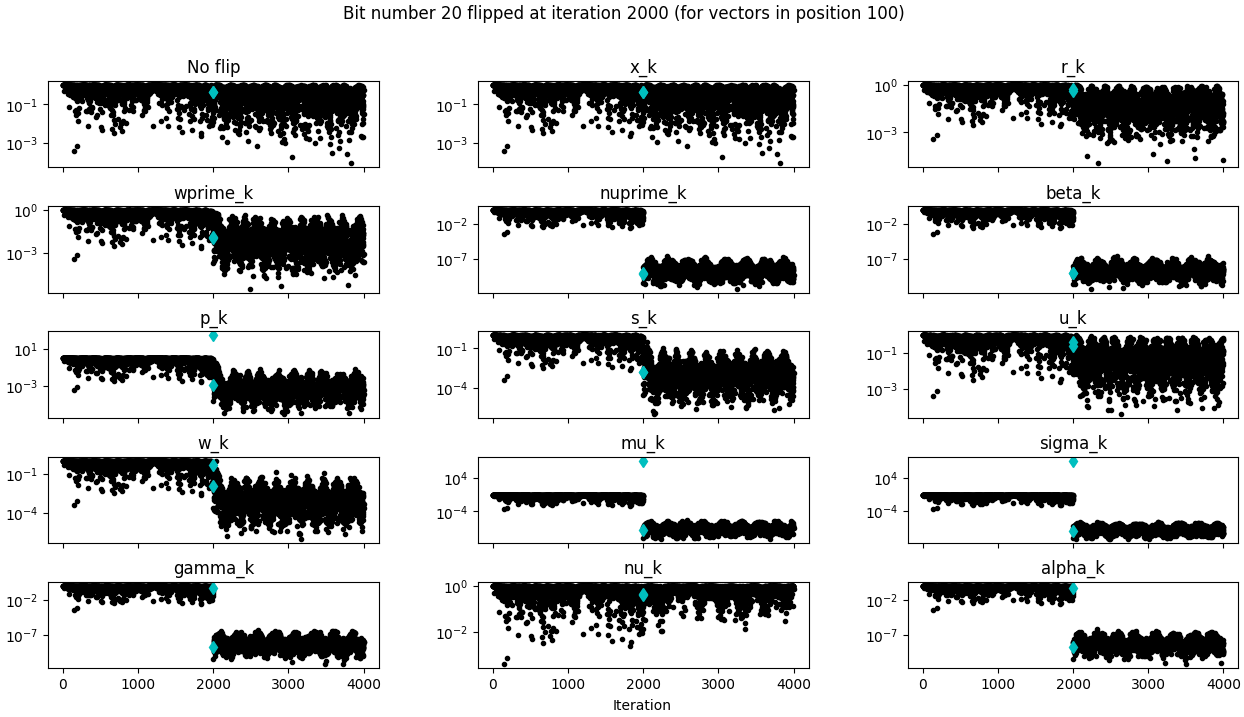

Figure 4.23: Relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, matrix *nos7*

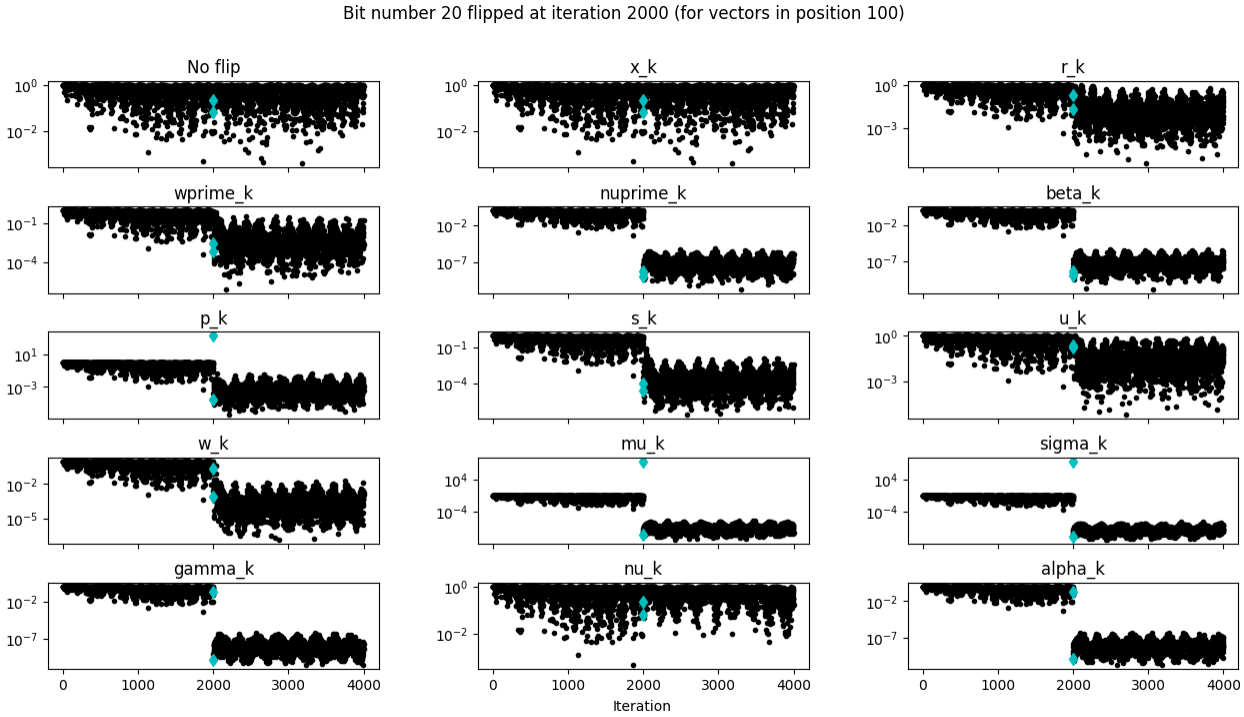

Figure 4.24: Relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, matrix *nos7*

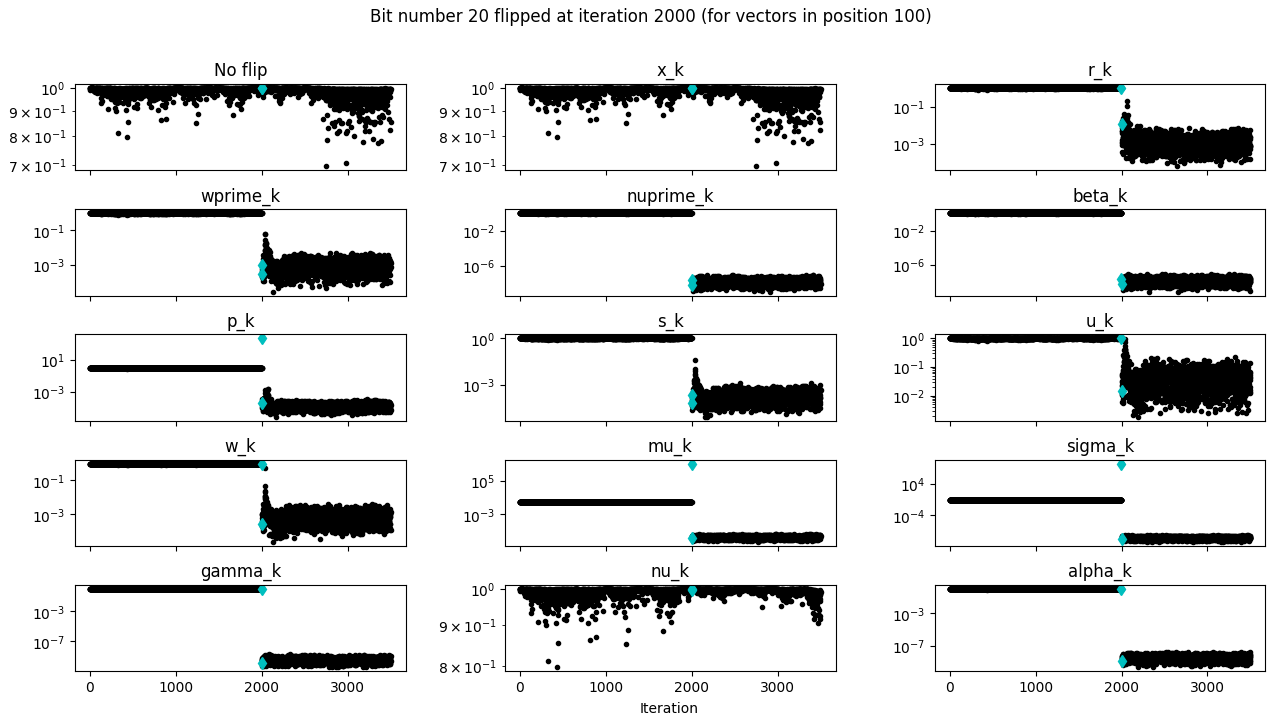

Figure 4.25: Relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, matrix *1138_bus*

4.4 Summary of detection methods

Throughout this chapter, we have investigated four methods for silent error detection in Pipe-PR-CG. Let us now summarize their efficacy in terms of whether or not they have the potential to reliably detect bit flips in a given variable. This is presented in Table 4.1 below, with rows representing each of the Pipe-PR-CG variables. The columns correspond to the detection methods presented in this chapter, i.e., violation of the ν-bound by the ν-gap, violation of the $\mathbf{w}$-bound by the $\mathbf{w}$-gap, violation of the μ-bound by the μ-gap, and finally violation of some preset thresholds by the ratio $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$.

The symbol ✓denotes that the method is able to reliably detect flips in the variable, while ∘ denotes that the method is, for the given variable, somehow functional, but either not in all cases or there are some specific circumstances under which it is not able to detect the injected error, e.g., the ν-gap/bound criterion not working for sign flips in the residual vector $\mathbf{r_k}$.

We can see from the table that the only variable which is not covered by any of the detection methods is the solution vector $\mathbf{x_k}$. The reason for this is that $\mathbf{x_k}$ appears only in its own relation, and thus it does affect any other variable. Therefore, for silent error detention in $\mathbf{x_k}$, a redundancy approach is unfortunately needed. Besides that, we do not posses a robust detection method for flips in $\mathbf{s_k}$. Nonetheless, in the worst case, the redundancy approach can be applied here as well if it turns out that detection by the relative μ-gap/bound difference is truly unreliable. For all other variables we should be able to detect silent errors reasonably well.

Variable	ν-gap/bound	$\mathbf{w}$-gap/bound	μ-gap/bound	$\lvert B_{\mu'_k} - \Delta_{\mu'_k} \rvert / B_{\mu'_k}$
$\mathbf{x_k}$				
$\mathbf{r_k}$	○	✓		○
$\mathbf{w'_k}$		✓		○
ν'_k	✓			✓
β_k				✓
$\mathbf{p_k}$			✓	✓
$\mathbf{s_k}$				○
$\mathbf{u_k}$		✓		○
$\mathbf{w_k}$		✓		○
μ_k			✓	✓
σ_k	✓		✓	✓
γ_k	✓			✓
ν_k	✓			
α_k				✓

Table 4.1: Efficacy of detection methods for each Pipe-PR-CG variable

As a final remark, let us add that if the computation of additional inner products and constants necessary for the bounds would in some case be too expensive, it is also possible to construct a set of detection criteria based just on the values of the gaps alone. For instance, we could monitor a moving average of gap value differences between iterations. Nonetheless, it is important to keep in mind that for this approach to function we must somehow deal with iterations where any of the gaps are zero. It would also require us to set some threshold to determine when to raise an alarm that a silent error has likely occurred.

5. Fault-tolerant Pipe-PR-CG

5.1 Detection testing

In the previous chapter, we have derived several methods for silent error detection in Pipe-PR-CG. For each method we have presented a sample of numerical experiments aimed at deducing for which variables the criterion works. Subsequently, the findings were summarized in Table 4.1. The conclusion was that for each variable, excluding $\mathbf{x_k}$, we seem to possess a detection criterion which, to a certain degree, works.

With this in hand, it is now time to test how well our criteria can detect silent errors in practice. This section contains the results of a large numerical experiment examining the performance of the criteria on a large sample of test runs, both with and without bit flips. The experiment was performed for each of the Pipe-PR-CG variables with the exception of $\mathbf{x_k}$, since none of the methods work for this variable. It is also worth noting that the experiment was performed for each of the variables separately, so that eventual outliers can be identified more easily.

For all variables, the testing was done using eight matrices from [21], which were utilized in the third chapter. We have decided to exclude from this list the matrix *aft01*, since the relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, criterion does not work for it. A figure illustrating this, along with a short note, can be found in Appendix A.2. The other three criteria are sometimes able to detect flips in some of "their" variables for this matrix, but even then the detection is not as all-encompassing as for the other matrices. The reason for this behavior is most likely the quite high condition number of the matrix ($\kappa(A) = 4.387086e+18$). On top of this, the positive definiteness of the matrix is borderline. As a consequence, it was decided not to use it in the experiment. However, this does not necessarily mean that the methods are not functional, since the properties of the matrix *aft01* are truly extreme.

Therefore, as was mentioned, there were eight matrices used in the experiment:

- *1138_bus* $\in \mathbb{R}^{1,138\times1,138}$, $\kappa(A) = 8.572646e+06$,
- *bcsstm07* $\in \mathbb{R}^{420\times420}$, $\kappa(A) = 7.615188e+03$,
- *bundle1* $\in \mathbb{R}^{10,581\times10,581}$, $\kappa(A) = 1.004238e+03$,
- *wathen120* $\in \mathbb{R}^{36,441\times36,441}$, $\kappa(A) = 2.576962e+03$,
- *bcsstk05* $\in \mathbb{R}^{153\times153}$, $\kappa(A) = 1.428114e+04$,
- *gr_30_30* $\in \mathbb{R}^{900\times900}$, $\kappa(A) = 1.945739e+02$,
- *nos7* $\in \mathbb{R}^{729\times729}$, $\kappa(A) = 2.374510e+09$,
- *crystm01* $\in \mathbb{R}^{4,875\times4,875}$, $\kappa(A) = 2.283164e+02$.

For each variable and each of these matrices, 800 runs with a single bit flip and 200 without a bit flip were performed. The bit number was chosen randomly

from 1 to 64. For vectors, the flip occurred in a random index from $[1, n]$, where n is the problem dimension. The flip iteration τ was chosen randomly from 0.1φ to 0.9φ, where φ is the number of iterations needed to converge for the given matrix and right-hand side when no flips occur. This was computed before each tainted run. The stopping criterion was always, for both untainted as well as tainted runs, such that it must hold that $||\mathbf{r_k}||/||\mathbf{b}|| < 1\text{e}{-10}$. The right-hand side $\mathbf{b}$ was a random vector from a uniform distribution over $[0, 1)$. A run tainted by a bit flip was deemed as convergent if it reached the stopping criteria within 1.5φ iterations. The initial guess $\mathbf{x_0}$ was always a vector of all zeros. Runs with an overflow error were not counted, because such errors are no longer silent. For this reason, the total number of runs recorded for each matrix is slightly lower than the above-stated 1000 performed. When possible, the norms appearing in the code were calculated using the already computed quantities, such as ν_k for $||\mathbf{r_k}||$ or γ_k for $||\mathbf{s_k}||$. The only exception was the norm $||\mathbf{r_k}||$ used for the stopping criterion which was not computed utilizing ν_k, so that the convergence can be evaluated more independently from the detection criteria.

Let us now recall what our four detection methods are. The following list summarizes this, with the three "absolute" bound violation criteria coming first, followed by the relative μ-gap/bound difference being lower than some preset threshold. If any of the inequalities

- $|\nu_k - \nu'_k| > \epsilon\,(21 + 6n)(||\mathbf{r_{k-1}}||^2 + ||\mathbf{r_k}||^2)$,
- $||\mathbf{w_k} - \mathbf{w'_k}|| > \epsilon\,||\mathbf{A}||\,\Big((c+3)||\mathbf{r_k}|| + (c+4)||\mathbf{r_{k-1}}|| + (c+2)\,|\alpha_{k-1}|\,||\mathbf{s_{k-1}}||\Big)$,
- $|\Delta_{\mu'_k}| \coloneqq |\mu_k - \sigma_k|$
  $> |\beta_k|\,|\langle \mathbf{p_{k-1}}, \mathbf{s_k}\rangle| + \epsilon\,||\mathbf{s_k}||\Big(||\mathbf{r_k}|| + 2|\beta_k|\,||\mathbf{p_{k-1}}|| + n\,(||\mathbf{p_k}|| + ||\mathbf{r_k}||)\Big) =: B_{\mu'_k}$,
- $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k} <$ threshold,

held, an alarm was raised. Let us from now on call these four criteria working together a *detection set*. In the experiment, there were two detection sets with a different threshold for the relative μ-gap/bound difference criterion, so that it can be evaluated how the detection behavior changes with the threshold. The threshold values were chosen based on experiments from the fourth chapter. Specifically, the levels $5\text{e}{-1}$ and $1\text{e}{-4}$ were chosen, as they seemed to be close to the lower limit of values of the relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, for matrices *1138_bus* and *nos7* when no flips occur, see, e.g., Figure 4.25 and Figure 4.24. The two detection sets with different thresholds for the fourth relative μ-gap/bound difference-based criterion were both evaluated simultaneously during each run. Therefore, we can directly compare how they perform for identical data.

The sequence of steps in the experiment was following:

1. For untainted runs, a right-hand side vector $\mathbf{b}$ is generated, then the computation is performed and it is noted whether an alarm was raised. The two detection sets using the two different thresholds for the $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$ criterion are both checked during the run and they each posses their own alarm.

2. For tainted runs, a right-hand side vector $\mathbf{b}$ is generated, and subsequently an untainted computation is performed to obtain the number of iterations φ needed to converge. Afterwards, the flip iteration τ, bit number, and vector flip index are generated. Then, a tainted run is performed with these inputs. As was the case for the untainted runs, both detection sets are monitored during the computation. Once again, this is done so that they can be better compared against each other, since they examine the same data. If during the run one of the two detection sets raises its alarm, it is noted at what iteration ρ_i, $i \in \{1, 2\}$, that first was. Later alarms are not taken into account. Besides the first alarm iterations, it is also monitored whether the run converged within the 1.5φ iterations or not.

Once the computation concluded and the information about alarms and the number of iterations needed to converge was final, the run (or rather "runs" as we evaluate both detection sets simultaneously during the run) was sorted into one of these six categories:

- true positive (tp): Bit flip occurred, did prevent convergence, and an alarm was raised.
- special positive (sp): Bit flip occurred, did not prevent convergence, and an alarm was raised.
- false positive (fp): An alarm was raised when no flip occurred.
- true negative (tn): If for run with no flips no alarm was raise.
- special negative (sn): If there was a flip which did not prevent convergence and no alarm was raised.
- false negative (fn): A bit flip occurred, did prevent convergence, and no alarm was raised.

The same categorization was used in the article [11]. As was mentioned above, if in a tainted run any of the four criteria within the detection sets raised an alarm it was noted at what iteration (ρ_1 for the first detection set and ρ_2 for the second detection set) this first occurred. The values ρ_i were initialized as ∞. Thus, if the detection set did not raise an alarm during the run its ρ_i value remained so after the computation had concluded. To classify runs, we compared the flip iteration τ and ρ_i, receiving the following options:

1. If $\rho_i < \tau$, we count this as false positive,
2. If $\rho_i \in \{\tau, \tau + 1\}$, we count this as true/special positive based on whether the run converged,
3. If $\rho_i > \tau + 1$, we count this as false/special negative based on whether the run converged.

Runs without a bit flip were categorized either as true negative or false positive based on whether or not an alarm was raised.

The output of the experiment is presented below in Tables 5.2 - 5.14, which contain results for all 13 variables separately, as well as in Table 5.1, which contains the sum of all runs over the individual variables. The rows of these tables correspond to runs for a specific matrix and threshold. The columns contain primarily the sorting of the runs into the six above-mentioned categories. Besides this, the tables for individual variables also contain in the columns information about the lowest numbered bit for each matrix whose run was classified as special negative (*snbit*), so that the interesting aspect of an undetected bit flip which does not destroy convergence can be examined. On top of that, it is also noted what was the highest bit for which an overflow error occurred (*ovbit*). At the bottom of each variable table, a row containing sums of run categories and extrema of the *snbit* and *ovbit* values over all matrices is presented. For the combined Table 5.1, only the sums of the six categories are shown, as taking a maximum/minimum of the above bit numbers over all variables does not, in the author's opinion, make much sense.

With the setup explained, let us now examine the outcomes. The most crucial result is that we were generally able to detect an overwhelming majority of bit flips which would ruin convergence. Important also is the fact that there was no variable which would stick out as seriously problematic for our detection methods. Moreover, the number of false positive runs was (for both thresholds) very close for all variables.

However, there were differences in the overall number of detected errors which would not destroy convergence (sp/sn). The one variable which stands out in this is $\mathbf{s_k}$. In Table 5.7 we observe that the number of special negative runs was for $\mathbf{s_k}$ considerably larger than for any other variable. This was most likely caused by the fact that for $\mathbf{s_k}$ only the relative μ-gap/bound difference criterion works, and even then, it is not fully reliable, as was mentioned at the end of the fourth chapter. Nonetheless, a majority of the undetected flips were special negative, thus they did not destroy convergence, and it can be seen that the number of false negatives is for $\mathbf{s_k}$ quite acceptable. The largest number of false negatives was noted for β_k, but it was still rather small. This variable shows some unique behavior as is mentioned in paragraphs below. Notable also is the fact that in the case of scalar variables for which one of the gap/bound detection methods works (ν'_k, μ_k, σ_k, γ_k, and ν_k) we were able to detect a large portion of the convergence-preserving silent errors.

For most matrices, there were no or almost no false positives. The notable outliers are *nos7* and *1138_bus*. Interestingly, these two matrices along with *bcsstk05* and *bcsstm07* are all in the upper half of our sample when it comes to condition number. This leads to the likely conclusion that the threshold value should be ideally chosen proportional to the condition number of the problem matrix. Let us also note that the false positive detections are caused only by the criterion utilizing the relative difference of the μ-gap and the μ-bound. The three bound violation criteria raise the alarm only when a bit flip truly occurs.

When it comes to the two parameters *snbit* and *ovbit*, it is interesting to observe that in most cases their values correspond to whether the variable is a scalar or a vector. The highest bit number resulting in overflows was generally large for vectors and small for scalars. On the other hand, for vectors, the smallest bit number causing a special negative case was generally lower than for scalars.

However, there are some exceptions to this. For instance, for β_k (Table 5.5), the special negative runs occurred even for bits of a very low number. On the other hand, when it comes to the highest numbered bit which caused an overflow error, $\mathbf{r_k}$ (Table 5.2) stands out as a vector for which this was exceptionally low.

In conclusion, our detection criteria were able to detect silent errors reasonably well, and for the vast majority of the non-detected bit flips the algorithm managed to recover and converge within our set iteration range.

matrix	threshold	tp	sp	fp	tn	sn	fn
1138_bus	5e−1	1532	4125	2681	1652	2844	2
	1e−4	1732	4509	0	2600	3989	6
bcsstm07	5e−1	939	6342	2	2600	2960	3
	1e−4	940	5710	0	2600	3593	3
bundle1	5e−1	833	5213	0	2600	4088	1
	1e−4	833	4554	0	2600	4747	1
wathen120	5e−1	771	5331	0	2600	4113	2
	1e−4	771	4769	0	2600	4675	2
bcsstk05	5e−1	2991	4793	348	2324	2314	5
	1e−4	2999	4186	0	2600	2980	10
gr_30_30	5e−1	964	6479	0	2600	2783	2
	1e−4	964	5900	0	2600	3362	2
nos7	5e−1	0	0	12823	0	0	0
	1e−4	1381	2881	3843	1229	3485	4
crystm01	5e−1	799	5896	0	2600	3564	1
	1e−4	799	5292	0	2600	4168	1
$\sum$	5e−1	8829	38179	15854	16976	22666	16
	1e−4	10419	37801	3843	19429	30999	29

Table 5.1: Detection performance, sum over all variables

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	76	274	217	134	290	0	29	2
	1e−4	88	354	0	200	349	0	29	2
bcsstm07	5e−1	15	457	0	200	319	0	31	2
	1e−4	15	453	0	200	323	0	31	2
bundle1	5e−1	16	336	0	200	441	0	15	2
	1e−4	16	333	0	200	444	0	15	2
wathen120	5e−1	8	393	0	200	386	0	28	2
	1e−4	8	393	0	200	386	0	28	2
bcsstk05	5e−1	218	328	22	181	240	0	42	2
	1e−4	218	318	0	200	253	0	42	2
gr_30_30	5e−1	23	507	0	200	263	0	39	2
	1e−4	23	507	0	200	263	0	39	2
nos7	5e−1	0	0	985	0	0	0	-	2
	1e−4	63	233	307	87	295	0	12	2
crystm01	5e−1	11	417	0	200	361	0	32	2
	1e−4	11	417	0	200	361	0	32	2
$\sum$ or extrema	5e−1	367	2712	1224	1315	2300	0	15	2
	1e−4	442	3008	307	1487	2674	0	12	2

Table 5.2: Detection performance, bit flip in $\mathbf{r_k}$

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	102	233	199	123	327	1	27	5
	1e−4	106	287	0	200	391	1	27	5
bcsstm07	5e−1	30	459	0	200	299	1	33	4
	1e−4	30	445	0	200	313	1	29	4
bundle1	5e−1	38	294	0	200	435	0	24	9
	1e−4	38	270	0	200	459	0	22	9
wathen120	5e−1	7	428	0	200	353	1	30	9
	1e−4	7	428	0	200	353	1	30	9
bcsstk05	5e−1	225	330	27	183	211	0	42	5
	1e−4	230	300	0	200	246	0	39	5
gr_30_30	5e−1	38	472	0	200	272	0	40	4
	1e−4	38	470	0	200	274	0	40	4
nos7	5e−1	0	0	988	0	0	0	-	3
	1e−4	66	173	304	85	360	0	1	3
crystm01	5e−1	25	411	0	200	354	0	33	2
	1e−4	25	411	0	200	354	0	33	2
$\sum$ or extrema	5e−1	465	2627	1214	1306	2251	3	24	9
	1e−4	540	2784	304	1485	2750	3	1	9

Table 5.3: Detection performance, bit flip in $\mathbf{w'_k}$

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	95	428	204	125	139	0	50	3
	1e−4	106	513	0	200	172	0	49	3
bcsstm07	5e−1	50	584	0	200	154	0	51	3
	1e−4	50	569	0	200	169	0	51	3
bundle1	5e−1	45	538	0	200	206	0	47	3
	1e−4	45	518	0	200	226	0	46	3
wathen120	5e−1	40	524	0	200	224	0	46	3
	1e−4	40	506	0	200	242	0	45	3
bcsstk05	5e−1	197	462	19	184	127	0	52	3
	1e−4	197	448	0	200	144	0	51	3
gr_30_30	5e−1	50	553	0	200	181	0	51	2
	1e−4	50	529	0	200	205	0	50	2
nos7	5e−1	0	0	988	0	0	0	-	3
	1e−4	92	390	293	80	133	0	45	3
crystm01	5e−1	46	550	0	200	192	0	49	2
	1e−4	46	532	0	200	210	0	48	2
$\sum$ or extrema	5e−1	523	3639	1211	1309	1223	0	46	3
	1e−4	626	4005	293	1480	1501	0	45	3

Table 5.4: Detection performance, bit flip in ν'_k

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	99	395	208	121	163	1	2	3
	1e−4	112	289	0	200	385	1	2	3
bcsstm07	5e−1	45	540	0	200	205	1	2	2
	1e−4	45	364	0	200	381	1	2	2
bundle1	5e−1	40	478	0	200	264	1	2	3
	1e−4	40	310	0	200	432	1	2	3
wathen120	5e−1	26	470	0	200	293	0	3	2
	1e−4	26	297	0	200	466	0	3	2
bcsstk05	5e−1	205	384	28	177	186	5	2	3
	1e−4	200	240	0	200	335	10	2	3
gr_30_30	5e−1	38	542	0	200	205	0	3	2
	1e−4	38	379	0	200	368	0	3	2
nos7	5e−1	0	0	989	0	0	0	-	3
	1e−4	93	207	294	91	302	2	2	3
crystm01	5e−1	18	511	0	200	262	0	3	2
	1e−4	18	349	0	200	424	0	3	2
$\sum$ or extrema	5e−1	471	3320	1225	1298	1578	8	2	3
	1e−4	572	2435	294	1491	3093	15	2	3

Table 5.5: Detection performance, bit flip in β_k

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	73	282	214	125	297	0	28	2
	1e−4	79	337	0	200	375	0	27	2
bcsstm07	5e−1	16	479	0	200	294	0	34	2
	1e−4	16	460	0	200	313	0	30	2
bundle1	5e−1	2	368	0	200	411	0	25	8
	1e−4	2	347	0	200	432	0	19	8
wathen120	5e−1	9	354	0	200	425	0	26	2
	1e−4	9	342	0	200	437	0	23	2
bcsstk05	5e−1	170	392	27	178	222	0	42	3
	1e−4	170	374	0	200	245	0	39	3
gr_30_30	5e−1	7	517	0	200	265	0	38	2
	1e−4	7	497	0	200	285	0	36	2
nos7	5e−1	0	0	986	0	0	0	-	3
	1e−4	59	179	290	96	362	0	20	3
crystm01	5e−1	4	420	0	200	365	0	32	2
	1e−4	4	405	0	200	380	0	30	2
$\sum$ or extrema	5e−1	281	2812	1227	1303	2279	0	25	8
	1e−4	346	2941	290	1496	2829	0	19	8

Table 5.6: Detection performance, bit flip in $\mathbf{p_k}$

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	144	212	215	131	279	0	31	4
	1e−4	158	121	0	200	498	4	18	4
bcsstm07	5e−1	27	453	0	200	300	1	29	4
	1e−4	27	289	0	200	464	1	18	4
bundle1	5e−1	35	308	0	200	426	0	26	8
	1e−4	35	136	0	200	598	0	1	8
wathen120	5e−1	3	354	0	200	427	0	24	9
	1e−4	3	191	0	200	590	0	14	9
bcsstk05	5e−1	241	326	30	173	212	0	43	4
	1e−4	241	160	0	200	381	0	28	4
gr_30_30	5e−1	22	477	0	200	293	0	40	3
	1e−4	22	320	0	200	450	0	23	3
nos7	5e−1	0	0	983	0	0	0	-	5
	1e−4	106	56	274	106	439	2	4	5
crystm01	5e−1	30	373	0	200	390	0	32	2
	1e−4	30	202	0	200	561	0	19	2
$\sum$ or extrema	5e−1	502	2503	1228	1304	2327	1	24	9
	1e−4	622	1475	274	1506	3981	7	4	9

Table 5.7: Detection performance, bit flip in $\mathbf{s_k}$

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	79	221	218	122	337	0	27	4
	1e−4	87	279	0	200	411	0	26	4
bcsstm07	5e−1	55	422	2	200	307	0	34	4
	1e−4	56	414	0	200	316	0	34	4
bundle1	5e−1	20	308	0	200	442	0	24	7
	1e−4	20	292	0	200	458	0	18	7
wathen120	5e−1	11	380	0	200	385	0	31	8
	1e−4	11	380	0	200	385	0	31	8
bcsstk05	5e−1	240	308	26	181	218	0	43	5
	1e−4	241	292	0	200	240	0	40	5
gr_30_30	5e−1	44	467	0	200	272	1	42	4
	1e−4	44	466	0	200	273	1	40	4
nos7	5e−1	0	0	978	0	0	0	-	4
	1e−4	41	122	298	97	420	0	1	4
crystm01	5e−1	37	422	0	200	335	0	31	2
	1e−4	37	421	0	200	336	0	31	2
$\sum$ or extrema	5e−1	486	2528	1224	1303	2296	1	24	8
	1e−4	537	2666	298	1497	2839	1	1	8

Table 5.8: Detection performance, bit flip in $\mathbf{u_k}$

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	88	274	202	130	297	0	29	3
	1e−4	99	331	0	200	361	0	29	3
bcsstm07	5e−1	29	472	0	200	289	0	31	4
	1e−4	29	461	0	200	300	0	31	4
bundle1	5e−1	39	289	0	200	430	0	23	9
	1e−4	39	272	0	200	447	0	21	9
wathen120	5e−1	3	385	0	200	393	0	29	9
	1e−4	3	385	0	200	393	0	29	9
bcsstk05	5e−1	245	298	33	174	219	0	44	4
	1e−4	246	274	0	200	249	0	43	4
gr_30_30	5e−1	39	485	0	200	271	0	39	4
	1e−4	39	482	0	200	274	0	39	4
nos7	5e−1	0	0	988	0	0	0	-	4
	1e−4	70	181	294	96	347	0	6	4
crystm01	5e−1	20	427	0	200	343	0	34	2
	1e−4	20	427	0	200	343	0	34	2
$\sum$ or extrema	5e−1	463	2630	1223	1304	2242	0	23	9
	1e−4	545	2813	294	1496	2714	0	6	9

Table 5.9: Detection performance, bit flip in $\mathbf{w_k}$

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	144	378	199	122	141	0	50	4
	1e−4	173	430	0	200	181	0	49	4
bcsstm07	5e−1	129	493	0	200	160	0	52	4
	1e−4	129	479	0	200	174	0	50	4
bundle1	5e−1	91	482	0	200	210	0	49	4
	1e−4	91	470	0	200	222	0	48	4
wathen120	5e−1	126	425	0	200	231	0	47	4
	1e−4	126	416	0	200	240	0	46	4
bcsstk05	5e−1	262	409	28	178	112	0	54	4
	1e−4	265	394	0	200	130	0	52	4
gr_30_30	5e−1	117	508	0	200	159	0	53	4
	1e−4	117	496	0	200	171	0	52	4
nos7	5e−1	0	0	986	0	0	0	-	4
	1e−4	143	260	287	104	192	0	43	4
crystm01	5e−1	110	519	0	200	160	0	51	3
	1e−4	110	505	0	200	174	0	49	3
$\sum$ or extrema	5e−1	979	3214	1213	1300	1173	0	47	4
	1e−4	1154	3450	287	1504	1484	0	43	4

Table 5.10: Detection performance, bit flip in μ_k

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	162	361	206	127	133	0	50	3
	1e−4	185	436	0	200	168	0	49	3
bcsstm07	5e−1	118	502	0	200	167	0	52	3
	1e−4	118	493	0	200	176	0	52	3
bundle1	5e−1	141	462	0	200	192	0	49	3
	1e−4	141	441	0	200	213	0	48	3
wathen120	5e−1	134	427	0	200	228	0	47	3
	1e−4	134	416	0	200	239	0	46	3
bcsstk05	5e−1	250	409	27	180	119	0	53	3
	1e−4	252	398	0	200	135	0	52	3
gr_30_30	5e−1	178	472	0	200	130	1	53	3
	1e−4	178	459	0	200	143	1	52	3
nos7	5e−1	0	0	984	0	0	0	-	3
	1e−4	144	332	295	100	113	0	47	3
crystm01	5e−1	136	458	0	200	193	0	50	2
	1e−4	136	438	0	200	213	0	49	2
$\sum$ or extrema	5e−1	1119	3091	1217	1307	1162	1	47	3
	1e−4	1288	3413	295	1500	1400	1	46	3

Table 5.11: Detection performance, bit flip in σ_k

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	169	372	210	128	114	0	51	3
	1e−4	195	445	0	200	153	0	51	3
bcsstm07	5e−1	139	515	0	200	140	0	52	3
	1e−4	139	498	0	200	157	0	52	3
bundle1	5e−1	132	459	0	200	195	0	48	3
	1e−4	132	444	0	200	210	0	48	3
wathen120	5e−1	128	428	0	200	233	0	47	3
	1e−4	128	418	0	200	243	0	46	3
bcsstk05	5e−1	248	408	34	177	121	0	54	3
	1e−4	249	406	0	200	133	0	54	3
gr_30_30	5e−1	153	507	0	200	124	0	53	3
	1e−4	153	483	0	200	148	0	51	3
nos7	5e−1	0	0	983	0	0	0	-	3
	1e−4	165	307	313	100	98	0	51	3
crystm01	5e−1	125	497	0	200	163	0	50	2
	1e−4	125	473	0	200	187	0	49	2
$\sum$ or extrema	5e−1	1094	3186	1227	1305	1090	0	47	3
	1e−4	1286	3474	313	1500	1329	0	46	3

Table 5.12: Detection performance, bit flip in γ_k

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	155	333	197	130	170	0	50	3
	1e−4	181	410	0	200	194	0	49	3
bcsstm07	5e−1	174	462	0	200	152	0	51	3
	1e−4	174	462	0	200	152	0	51	3
bundle1	5e−1	139	413	0	200	228	0	46	3
	1e−4	139	413	0	200	228	0	46	3
wathen120	5e−1	143	393	0	200	249	1	45	4
	1e−4	143	393	0	200	249	1	45	4
bcsstk05	5e−1	261	361	29	175	157	0	52	3
	1e−4	261	361	0	200	161	0	52	3
gr_30_30	5e−1	144	479	0	200	165	0	51	3
	1e−4	144	479	0	200	165	0	51	3
nos7	5e−1	0	0	983	0	0	0	-	3
	1e−4	178	285	299	96	135	0	48	2
crystm01	5e−1	142	423	0	200	223	1	48	3
	1e−4	142	423	0	200	223	1	48	3
$\sum$ or extrema	5e−1	1158	2864	1219	1305	1344	2	45	4
	1e−4	1362	3226	299	1496	1507	2	45	4

Table 5.13: Detection performance, bit flip in ν_k

matrix	threshold	tp	sp	fp	tn	sn	fn	snbit	ovbit
1138_bus	5e−1	146	362	192	134	157	0	46	2
	1e−4	163	277	0	200	351	0	31	2
bcsstm07	5e−1	112	504	0	200	174	0	44	2
	1e−4	112	323	0	200	355	0	34	2
bundle1	5e−1	95	478	0	200	208	0	44	2
	1e−4	95	308	0	200	378	0	31	2
wathen120	5e−1	133	370	0	200	286	0	41	2
	1e−4	133	204	0	200	452	0	27	2
bcsstk05	5e−1	229	378	18	183	170	0	49	2
	1e−4	229	221	0	200	328	0	33	2
gr_30_30	5e−1	111	493	0	200	183	0	50	2
	1e−4	111	333	0	200	343	0	36	2
nos7	5e−1	0	0	992	0	0	0	-	2
	1e−4	161	156	295	91	289	0	27	2
crystm01	5e−1	95	468	0	200	223	0	47	4
	1e−4	95	289	0	200	402	0	33	4
$\sum$ or extrema	5e−1	921	3053	1202	1317	1401	0	41	4
	1e−4	1099	2111	295	1491	2898	0	27	4

Table 5.14: Detection performance, bit flip in α_k

5.2 Correction of silent errors

5.2.1 Fault-tolerant Pipe-PR-CG algorithm

The experiment presented in the previous section has demonstrated that the combination of our detection methods is able to reliably detect majority of silent errors which would destroy convergence of the Pipe-PR-CG algorithm. The problem at hand is now how to correct these errors. The approach we present here is to perform a so-called *rollback* when the alarm is raised. Rolling back essentially means to "return" the computation to an uncorrupted state before the detected silent error has occurred. Therefore, in our case, to *recover* the computation we have to "return" two iterations back, since our detection methods raise the alarm either at the iteration when the error has occurred or one iteration later. This recovery approach was previously proposed in other studies investigating detection and correction of silent errors, e.g., in [11].

Below in Algorithm 10 is a statement of the Fault-Tolerant Pipelined Predict-and-Recompute Conjugate Gradient algorithm (FT-Pipe-PR-CG). Most of the first half of the loop (line 4 and lines 9 to 18) is a standard unpreconditioned Pipe-PR-CG routine, while the second half (lines 19 to 30) is newly added to incorporate into the algorithm the detection and correction of silent faults. Included in lines 5 to 8 is also an utilization of the redundancy detection approach for $\mathbf{x_k}$, so that we can uncover any silent errors in it, since, as was mentioned at the end of the fourth chapter, none of our detection methods work for $\mathbf{x_k}$. Inputs of the algorithm include, aside from the standard problem data $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{x_0}$, the constants $||\mathbf{A}||$, n, ϵ, and c, necessary for evaluation of the bounds, and the

threshold value T used in the $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$ criterion.

As for the implementation of our detection methods, first the gaps (lines 19 to 21) and bounds (lines 22 to 24) are computed, and then in line 25 it is checked whether an occurrence of a silent error is not indicated by any of the criteria. If the alarm is raised we call a Recover() procedure (given in Algorithm 9) to perform the two iteration rollback and we mark the iteration $k+2$ as "corrected", because otherwise, if the detection was falsely positive, the alarm would be raised again indefinitely every two iterations. Although, it may theoretically happen that another silent fault appears in an iteration we have marked as corrected, the probability of this is generally rather low, since silent errors are a rare event [12]. However, in cases where there is an extremely large number of false positive detections, this may be a problem. We propose a way to remedy this later.

In fault-tolerant Pipe-PR-CG it is necessary to compute three additional inner products, $\langle \mathbf{w_k} - \mathbf{w'_k}, \mathbf{w_k} - \mathbf{w'_k} \rangle$, $\langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle$, and $\langle \mathbf{p_k}, \mathbf{p_k} \rangle$, which do not appear in the basic Pipe-PR-CG algorithm, for the sake of our detection criteria. Nonetheless, it is possible to couple their computation with the other inner products in line 17 of Algorithm 10, and, aside from $\langle \mathbf{w_k} - \mathbf{w'_k}, \mathbf{w_k} - \mathbf{w'_k} \rangle$, there is no serial dependence on the matrix-vector multiplications. Therefore, the communication-hiding and pipelining properties of Pipe-PR-CG are, to a certain degree, preserved. Note that the procedure stated below is presented merely as a pseudocode. A practical implementation of it would ideally include an effective utilization of the already calculated variables for the computation of norms in lines 22 to 24.

As was already mentioned, the recover procedure is stated in Algorithm 9. It implements a two iteration rollback. At that point the data should be without any silent errors. We recover not only the data of index k which will be used in the next iteration, but variables with index $k-1$ as well. The reason for this is that the indicated silent error could have occurred either at iteration k or $k-1$, because our detection criteria may raise the alarm one iteration after the error has appeared. Therefore, by recovering variables of index $k-1$ as well we ensure that they are uncorrupted, and hence can be safely utilized to recover at iteration $k+1$ if needed.

Algorithm 9 Recover procedure of FT-Pipe-PR-CG

1: **procedure** RECOVER($\cdot$)
2: $\mathbf{x_k} = \mathbf{x_{k-2}}$, $\mathbf{x_{k-1}} = \mathbf{x_{k-3}}$
3: $\mathbf{r_k} = \mathbf{r_{k-2}}$, $\mathbf{r_{k-1}} = \mathbf{r_{k-3}}$
4: $\mathbf{w'_k} = \mathbf{w'_{k-2}}$, $\mathbf{w'_{k-1}} = \mathbf{w'_{k-3}}$
5: $\nu'_k = \nu'_{k-2}$, $\nu'_{k-1} = \nu'_{k-3}$
6: $\beta_k = \beta_{k-2}$, $\beta_{k-1} = \beta_{k-3}$
7: $\mathbf{p_k} = \mathbf{p_{k-2}}$, $\mathbf{p_{k-1}} = \mathbf{p_{k-3}}$
8: $\mathbf{s_k} = \mathbf{s_{k-2}}$, $\mathbf{s_{k-1}} = \mathbf{s_{k-3}}$
9: $\mathbf{u_k} = \mathbf{u_{k-2}}$, $\mathbf{u_{k-1}} = \mathbf{u_{k-3}}$
10: $\mathbf{w_k} = \mathbf{w_{k-2}}$, $\mathbf{w_{k-1}} = \mathbf{w_{k-3}}$
11: $\mu_k = \mu_{k-2}$, $\mu_{k-1} = \mu_{k-3}$
12: $\sigma_k = \sigma_{k-2}$, $\sigma_{k-1} = \sigma_{k-3}$
13: $\gamma_k = \gamma_{k-2}$, $\gamma_{k-1} = \gamma_{k-3}$
14: $\nu_k = \nu_{k-2}$, $\nu_{k-1} = \nu_{k-3}$
15: $\alpha_k = \alpha_{k-2}$, $\alpha_{k-1} = \alpha_{k-3}$
16: **end procedure**

Algorithm 10 Fault-Tolerant Pipelined Predict-and-Recompute Conjugate Gradient: FT-Pipe-PR-CG

1: **procedure** FT-PIPE-PR-CG($\mathbf{A}, \mathbf{b}, \mathbf{x_0}$, $||\mathbf{A}||$, n, ϵ, c, T)
2: INITIALIZE()
3: **for** $k = 1, 2, \ldots$ **do**
4: $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_{k-1}\mathbf{p_{k-1}}$
5: $\tilde{\mathbf{x}}_\mathbf{k} = \mathbf{x_{k-1}} + \alpha_{k-1}\mathbf{p_{k-1}}$
6: **if** $\mathbf{x_k} \neq \tilde{\mathbf{x}}_\mathbf{k}$
7: **Go To 4**
8: **end if**
9: $\mathbf{r_k} = \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}}$
10: $\mathbf{w}'_\mathbf{k} = \mathbf{w_{k-1}} - \alpha_{k-1}\mathbf{u_{k-1}}$
11: $\nu'_k = \nu_{k-1} - 2\alpha_{k-1}\sigma_{k-1} + \alpha^2_{k-1}\gamma_{k-1}$
12: $\beta_k = \nu'_k/\nu_{k-1}$
13: $\mathbf{p_k} = \mathbf{r_k} + \beta_k\mathbf{p_{k-1}}$
14: $\mathbf{s_k} = \mathbf{w}'_\mathbf{k} + \beta_k\mathbf{s_{k-1}}$
15: $\mathbf{u_k} = \mathbf{A}\mathbf{s_k}$
16: $\mathbf{w_k} = \mathbf{A}\mathbf{r_k}$
17: $\mu_k = \langle \mathbf{p_k}, \mathbf{s_k} \rangle,\ \sigma_k = \langle \mathbf{r_k}, \mathbf{s_k} \rangle,\ \gamma_k = \langle \mathbf{s_k}, \mathbf{s_k} \rangle,\ \nu_k = \langle \mathbf{r_k}, \mathbf{r_k} \rangle$
18: $\alpha_k = \nu_k/\mu_k$
19: $\Delta_{\nu'_k} = |\nu_k - \nu'_k|$
20: $\Delta_{\mathbf{w}'_k} = ||\mathbf{w_k} - \mathbf{w}'_\mathbf{k}||$
21: $\Delta_{\mu'_k} = |\mu_k - \sigma_k|$
22: $B_{\nu'_k} = \epsilon\,(21 + 6n)(||\mathbf{r_{k-1}}||^2 + ||\mathbf{r_k}||^2)$
23: $B_{\mathbf{w}'_k} = \epsilon\,||\mathbf{A}||\,((c+3)||\mathbf{r_k}|| + (c+4)||\mathbf{r_{k-1}}|| + (c+2)\,|\alpha_{k-1}|\,||\mathbf{s_{k-1}}||)$
24: $B_{\mu'_k} = |\beta_k|\,|\langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle| + \epsilon\,||\mathbf{s_k}||(||\mathbf{r_k}|| + 2|\beta_k|\,||\mathbf{p_{k-1}}|| + n\,(||\mathbf{p_k}|| + ||\mathbf{r_k}||))$
25: **if** $\Delta_{\nu'_k} > B_{\nu'_k}$ **or** $\Delta_{\mathbf{w}'_k} > B_{\mathbf{w}'_k}$ **or** $\Delta_{\mu'_k} > B_{\mu'_k}$ **or** $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k} < T$
26: **if** k **is not marked as corrected**
27: **Recover()**
28: **Mark** $k + 2$ **as corrected**
29: **end if**
30: **end if**
31: **end for**
32: **end procedure**

The advantage of the rollback correction approach is that the algorithm is able to universally recover from any detected silent error, no matter what variable it occurred in. The disadvantage is that we need to allocate extra memory for storing the variables from iterations $k - 2$ and $k - 3$.

5.2.2 Adaptive threshold refinement

The fault-tolerant Pipe-PR-CG as it is presented in Algorithm 10 should be able to reliably detect and correct the majority of silent errors significant for convergence. However, in the previous section (e.g., in Table 5.1) we have observed that for some matrices there were many runs which resulted in a false positive detection. Moreover, the experiment was categorizing the runs based only on the *first* raising of the alarm. Hence, in some problematic cases there may potentially be a large number of false positive detections during the computation. As a result of this, we would be performing many extra iterations owing to the rollback recovery. However, we have also seen that the number of false positives decreases with the value of the threshold parameter. This is a fact we could try to utilize.

The idea we propose here is to adapt the value of the threshold T during the run of the algorithm to reflect how many times the alarm was raised. As was mentioned, silent errors are rather rare events, so if the alarm is raised many times we can safely assume that in most cases we did not truly detect a fault. In such a situation it may be beneficial to lower the value of the threshold T to reduce the number of false positive detections by the relative μ-gap/bound difference criterion. This concept is presented below in the adaptive fault-tolerant Pipe-PR-CG, shortly, AFT-Pipe-PR-CG (Algorithm 11), implemented in such a way that we multiply T by an adaptation parameter $a \in (0, 1)$ each time the alarm is raised by the $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$ criterion. Note that it is also possible to increase the threshold when there is a large number of iterations without any alarm. But in that case, it is necessary to set some upper limit for T.

Not only does the adaptive algorithm potentially greatly reduce the number of false positive detections, it also allows us to eliminate the iteration marking. In FT-Pipe-PR-CG, if the alarm was raised at some iteration k we have marked iteration $k + 2$ as corrected, so that the procedure cannot get stuck in a loop. However, this can be caused only by the relative μ-gap/bound difference criterion. As was mentioned earlier, the three bound violation criteria raise the alarm only when a silent error truly occurs, i.e., they do not cause false positive detections. Thus, the procedure cannot get stuck because one of these criteria will indefinitely force a recovery in some iteration. The relative μ-gap/bound difference criterion could do this, but now, each time this method raises the alarm the threshold is lowered. Therefore, sooner or later it will hold that $T < |B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, and the procedure will continue.

Figures 5.1 and 5.2 show the process of threshold adaptation for the matrix *nos7* and right-hand side $\mathbf{b} = \mathbf{e}$. The adaptivity parameter a was in the presented runs set to 0.5 and 0.1, respectively. The initial threshold value was 5e−1, the higher value used in the detection performance experiment earlier in this chapter. Next to the variable names, it is noted how many recoveries, i.e., also detections, there were in total during the computation. This number also includes alarms raised after the bit flip by criteria other than the threshold violation by the relative μ-gap/bound difference. Therefore, for some variables, e.g., $\mathbf{w}'_\mathbf{k}$, the number of threshold adaptations was one less than the number of total detections indicated in the figures. A violation of the threshold by the $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$ ratio in the flip iteration or one iteration later is denoted by the diamond marker turning dark blue.

In the above-mentioned figures, we observe that AFT-Pipe-PR-CG seems to

be able to suitably adapt the threshold, so that the number of false positive detections is reduced, but at the same time the reliability of the detection is not destroyed. Moreover, this holds not only for the more "conservative" choice of the parameter a, but for the more "aggressive" variant $a = 0.1$ as well. Notable also is that the ratio $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$ no longer "jumps down" at the bit flip iteration as was the case for the figures in the fourth chapter, e.g., in Figure 4.24. This is because of the recovery procedure.

Algorithm 11 Adaptive Fault-Tolerant Pipelined Predict-and-Recompute Conjugate Gradient: AFT-Pipe-PR-CG

1: **procedure** AFT-Pipe-PR-CG($\mathbf{A}, \mathbf{b}, \mathbf{x_0}$, $||\mathbf{A}||$, n, ϵ, c, T, a)
2: INITIALIZE()
3: **for** $k = 1, 2, \ldots$ **do**
4: $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_{k-1}\mathbf{p_{k-1}}$
5: $\tilde{\mathbf{x}}_\mathbf{k} = \mathbf{x_{k-1}} + \alpha_{k-1}\mathbf{p_{k-1}}$
6: **if** $\mathbf{x_k} \neq \tilde{\mathbf{x}}_\mathbf{k}$
7: **Go To 4**
8: **end if**
9: $\mathbf{r_k} = \mathbf{r_{k-1}} - \alpha_{k-1}\mathbf{s_{k-1}}$
10: $\mathbf{w'_k} = \mathbf{w_{k-1}} - \alpha_{k-1}\mathbf{u_{k-1}}$
11: $\nu'_k = \nu_{k-1} - 2\alpha_{k-1}\sigma_{k-1} + \alpha^2_{k-1}\gamma_{k-1}$
12: $\beta_k = \nu'_k/\nu_{k-1}$
13: $\mathbf{p_k} = \mathbf{r_k} + \beta_k\mathbf{p_{k-1}}$
14: $\mathbf{s_k} = \mathbf{w'_k} + \beta_k\mathbf{s_{k-1}}$
15: $\mathbf{u_k} = \mathbf{A}\mathbf{s_k}$
16: $\mathbf{w_k} = \mathbf{A}\mathbf{r_k}$
17: $\mu_k = \langle \mathbf{p_k}, \mathbf{s_k} \rangle,\ \sigma_k = \langle \mathbf{r_k}, \mathbf{s_k} \rangle,\ \gamma_k = \langle \mathbf{s_k}, \mathbf{s_k} \rangle,\ \nu_k = \langle \mathbf{r_k}, \mathbf{r_k} \rangle$
18: $\alpha_k = \nu_k/\mu_k$
19: $\Delta_{\nu'_k} = |\nu_k - \nu'_k|$
20: $\Delta_{\mathbf{w}'_k} = ||\mathbf{w_k} - \mathbf{w'_k}||$
21: $\Delta_{\mu'_k} = |\mu_k - \sigma_k|$
22: $B_{\nu'_k} = \epsilon\,(21 + 6n)(||\mathbf{r_{k-1}}||^2 + ||\mathbf{r_k}||^2)$
23: $B_{\mathbf{w}'_k} = \epsilon\,||\mathbf{A}||\,((c+3)||\mathbf{r_k}|| + (c+4)||\mathbf{r_{k-1}}|| + (c+2)\,|\alpha_{k-1}|\,||\mathbf{s_{k-1}}||)$
24: $B_{\mu'_k} = |\beta_k|\,|\langle \mathbf{p_{k-1}}, \mathbf{s_k} \rangle| + \epsilon\,||\mathbf{s_k}||(||\mathbf{r_k}|| + 2|\beta_k|\,||\mathbf{p_{k-1}}|| + n\,(||\mathbf{p_k}|| + ||\mathbf{r_k}||))$
25: **if** $\Delta_{\nu'_k} > B_{\nu'_k}$ **or** $\Delta_{\mathbf{w}'_k} > B_{\mathbf{w}'_k}$ **or** $\Delta_{\mu'_k} > B_{\mu'_k}$ **or** $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k} < T$
26: **Recover()**
27: **if** $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k} < T$
28: $T = a \cdot T$
29: **end if**
30: **end if**
31: **end for**
32: **end procedure**

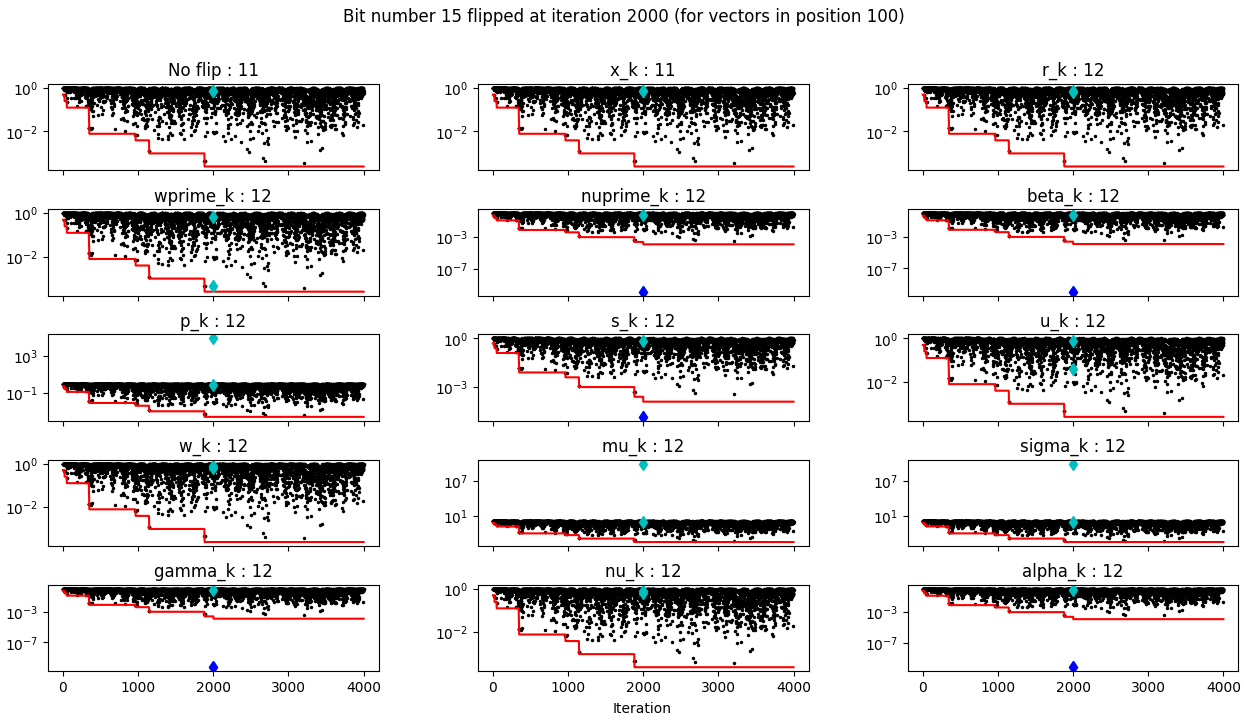

Figure 5.1: Adaptive threshold T for $a = 0.5$ (red) and the relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, (black), matrix *nos7*

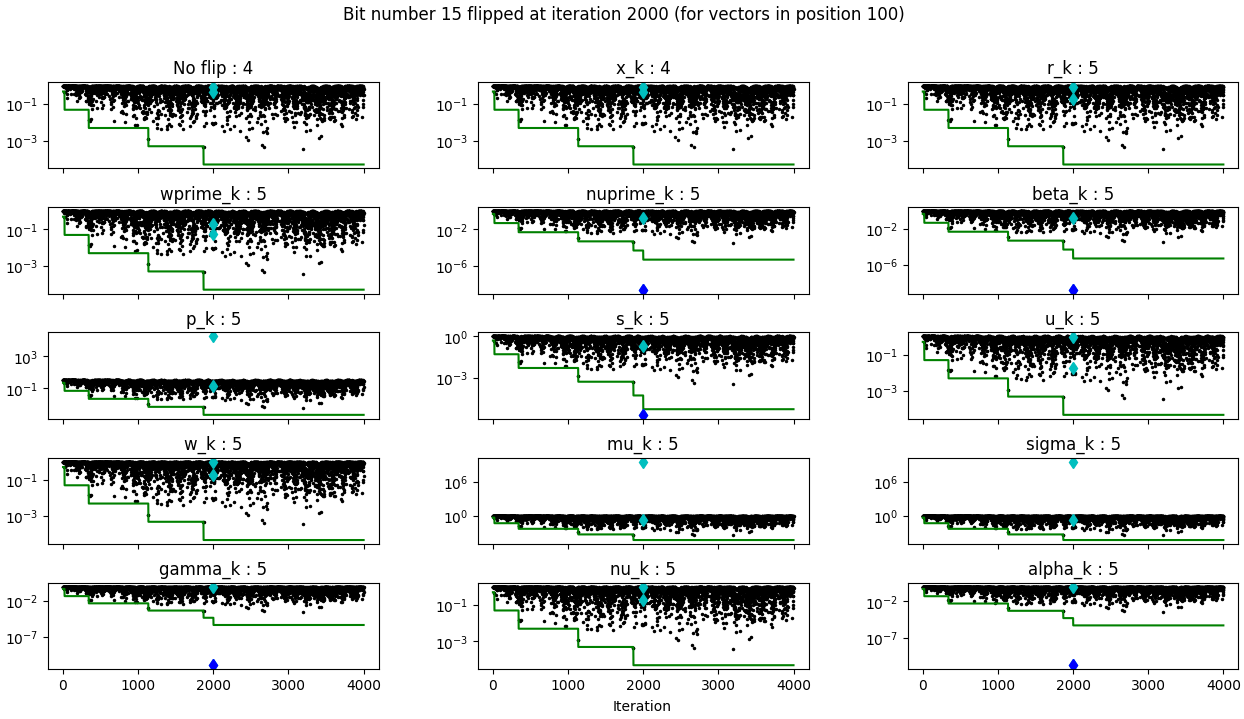

Figure 5.2: Adaptive threshold T for $a = 0.1$ (green) and the relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, (black), matrix *nos7*

Although it seems from the figures above that the adaptive threshold refinement works quite well, it is certainly appropriate to test it more thoroughly. For this purpose, one final numerical experiment was performed. In it, we have investigated the detection reliability of AFT-Pipe-PR-CG and the average number of alarms raised during its runs. The setup of this experiment was very similar to that of the detection experiment at the beginning of this chapter. The choice of the random problem parameters such as right-hand side or flip iteration was the same. Identical also were the convergence criterion, the initial guess, and that the already calculated variables were utilized for computation of the norms in the detection criteria. The initial value of the threshold was set to 5e−1. For each of the Pipe-PR-CG variables, excluding $\mathbf{x_k}$, 500 tainted runs were performed, i.e., there were 6500 runs in total for each matrix and choice of a. We have decided to include in the experiment only the matrices *1138_bus* and *nos7*, since for the other matrices from our sample almost no false positives were indicated in the large detection experiment (see, e.g., Table 5.1).

Please note that runs for the two chosen adaptation parameters a were performed separately, unlike in the case of the experiment in the previous section where the two threshold settings were tested on the same data. The reason for this is that it would be rather difficult to deal with situations when one detection set using the first parameter a does not raise the alarm, but the other detection set using the second parameter a does, and thus it also wants to perform a rollback. Nonetheless, the main purpose of this experiment was not a straight comparison of the two choices of a, but rather to investigate whether the introduction of the adaptive threshold refinement causes additional false negative detections, as well as to get a notion of how many alarms there are in average in each run. Additionally, we also obtain information about the number of extra iterations performed due to the recoveries as this is twice the number of alarms.

Let us now explain the specifics of the experiment and how the runs were classified. As we have done before, first an untainted run to obtain the number of iterations needed to converge for the given right-hand side $\mathbf{b}$ was performed. Then, we executed a tainted run of AFT-Pipe-PR-CG during which it was noted whether the bit flip was detected and corrected (positive cases) or whether the alarm was not raised (special/false negative cases based on the number of iterations to converge) either in the flip iteration or in the following iteration. It was also counted how many alarms, and therefore recoveries, were done during each run. The average number of alarms is presented in Table 5.15 alongside the run categorization. The positive cases were no longer sorted, since successful detection leads to only two additional iterations, and thus the method always converged within the given limit.

Investigating Table 5.15, we observe that the introduction of the adaptive threshold refinement did not increase the number of false negative detections, which remains at the same level as it was for the static threshold approach. Moreover, by utilizing the adaptive strategy, we were able to restrict the number of false positive detections to only a handful per run. This is especially impressive for the matrix *nos7*, because of its rather high condition number and the large number of false positive detections observed for it in the detection performance experiment (see Table 5.1). Interesting also may be that the number of average alarms is for the matrix *1138_bus* very close to one. Nonetheless, this number

also includes the negative detections for which there were no alarms at the flip iterations. In conclusion, this experiment has demonstrated that the AFT-Pipe-PR-CG algorithm is strongly reliable and that it can, in combination with a suitable parameter a, effectively reduce the number of extra iterations performed due to recovery.

matrix	a	positive	sn	fn	#alarms
1138_bus	0.5	4192	2232	4	1.057
	0.1	4310	2117	2	1.010
nos7	0.5	3579	2846	1	13.655
	0.1	3487	2942	1	4.784

Table 5.15: Performance of AFT-Pipe-PRCG

As a final remark, let us note that the threshold for the relative difference of the μ-gap and the μ-bound may also be set based on estimation of the condition number of the matrix $\mathbf{A}$. In the detection experiment we have observed that the higher the condition number of the matrix is, the more likely is a false positive alarm. As was already mentioned, the norm $||\mathbf{A}||$ can be reasonably estimated within few iterations of the Pipe-PR-CG algorithm. Additionally, it is also possible to estimate the condition number [24], and thus, we could use this information for setting the threshold value. However, this investigation is left for future research.

Conclusion

This thesis has explored the problem of the detection and correction of silent errors in the Pipe-PR-CG algorithm. To this end, we have first summarized fundamental information regarding silent errors and various conjugate gradient variants. Subsequently, in the third chapter, we have scrutinized the sensitivity of Pipe-PR-CG to silent errors. The conclusion of this investigation was that bit flips influence the convergence of the method more heavily when they occur early in the computation. Additionally, it was observed that there exists a strong correlation between the number of the flipped bit and the effect the fault has on convergence.

After this, we have shifted our focus to the derivation of methods which would be able to cheaply and reliably detect silent errors in Pipe-PR-CG. By utilizing rounding error analysis, we have managed to bound the values of three so-called "gaps" between variables which are equal in exact arithmetic. We have then shown that the violation of these bounds by the computed gaps can be used to detect silent errors in many, but not all, of the Pipe-PR-CG variables. In order to detect faults in the variables not covered by the three bound violation criteria, a fourth criterion, based on monitoring the relative difference between the μ-gap and the μ-bound, has been constructed. All of these methods are able to detect silent errors either immediately in the iteration in which they occur or one iteration later. In the main experiment of the fifth chapter, it was demonstrated that the union of the four derived criteria is able to reliably detect the vast majority of silent errors which, if left uncorrected, would significantly impact convergence of the method. In cases when the injected errors remained undetected, the algorithm has almost always managed to recover and reach the stopping criterion without serious delay.

In the final part of this work, we have incorporated the derived detection methods along with a simple recovery procedure into the Pipe-PR-CG algorithm. By this, we have constructed a fault-tolerant variant of Pipe-PR-CG, the FT-Pipe-PR-CG method. The recovery in case of an indicated error was performed by returning the computation to a state when the data were still uncorrupted, i.e., two iterations back. However, it was noted that for some matrices the fault-tolerant algorithm could be significantly slowed down by many extra iterations due to recovery caused by a large number false positive detections. To remedy this downside of FT-Pipe-PR-CG we have proposed the idea of adaptive threshold refinement based on the number of detection alarms during the computation. The resulting adaptive fault-tolerant algorithm, AFT-Pipe-PR-CG, has demonstrated a particular ability to quickly adjust itself to the problem, and consequently, to tremendously limit the number of false positive alarms. Moreover, this was accomplished without negatively impacting the high detection reliability.

The main contribution of this thesis is the mathematical derivation of the criteria for silent error detection in Pipe-PR-CG and the creation of AFT-Pipe-PR-CG, a fast yet reliable fault-tolerant algorithm. For future research remains the issue of modifying the presented detection criteria for use in preconditioned systems as well as an efficient implementation of the fault-tolerant algorithms for massively parallel machines.

Bibliography

[1] Tyler Chen and Erin Carson. Predict-and-recompute conjugate gradient variants. *SIAM Journal on Scientific Computing*, 42(5):A3084–A3108, 2020. URL https://epubs.siam.org/doi/10.1137/19M1276856.

[2] Lawrence Livermore National Laboratory. Introduction to parallel computing tutorial. URL https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial. (Last accessed on 2023/11/16).

[3] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011. doi: 10.1137/090769156. URL https://doi.org/10.1137/090769156.

[4] Angskun T. Bosilca G. et al. Pješivac-Grbović, J. Performance analysis of mpi collective operations. *Cluster Computing*, 10:127–143, 2007. URL https://doi.org/10.1007/s10586-007-0012-0.

[5] Walter Gander, Martin J. Gander, and Felix Kwok. *Scientific Computing - An Introduction using Maple and MATLAB*. First Edition. Springer Publishing, New York, 2014. ISBN 978-3-319-04325-8.

[6] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, mar 1991. ISSN 0360-0300. doi: 10.1145/103162.103163. URL https://doi.org/10.1145/103162.103163.

[7] Jakub Hercík. Comparison of iterative matrix methods for information retrieval, 2022. URL http://hdl.handle.net/20.500.11956/173946. Charles University, Faculty of Mathematics and Physics, Department of Numerical Mathematics.

[8] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229. URL https://ieeexplore.ieee.org/document/8766229.

[9] Erin Carson and Nicholas J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing*, 40(2):A817–A847, 2018. doi: 10.1137/17M1140819. URL https://doi.org/10.1137/17M1140819.

[10] Erik Jurjen Duintjer Tebbens, Iveta Hnětynková, Martin Plešinger, Zdeněk Strakoš, and Petr Tichý. *Analýza metod pro maticové výpočty : základní metody*. Matfyzpress, Praha, 2012. ISBN 978-80-7378-201-6.

[11] G. Meurant. Detection and correction of silent errors in the conjugate gradient algorithm. *Numerical Algorithms*, 92:869–891, 2023. URL https://doi.org/10.1007/s11075-022-01380-1.

[12] James Elliott, Mark Hoemmen, and Frank Mueller. Evaluating the impact of SDC on the GMRES iterative solver. pages 1193–1202, 2014. doi: 10.1109/IPDPS.2014.123. URL https://ieeexplore.ieee.org/document/6877347.

[13] Emmanuel Agullo, Siegfried Cools, Emrullah Fatih Yetkin, Luc Giraud, Nick Schenkels, and Wim Vanroose. On soft errors in the conjugate gradient method: Sensitivity and robust numerical detection. *SIAM Journal on Scientific Computing*, 42(6):C336–C358, 2020. URL https://epubs.siam.org/doi/10.1137/18M122858X.

[14] J. Liesen and P. Tichý. Convergence analysis of krylov subspace methods. *Mitteilungen der Gesellschaft für Angewandte Mathematik und Mechanik*, 27 (2):153–173, 2004. URL https://onlinelibrary.wiley.com/doi/epdf/10.1002/gamm.201490008.

[15] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6), 1952. URL https://nvlpubs.nist.gov/nistpubs/jres/049/jresv49n6p409_a1b.pdf.

[16] Zdeněk Strakoš and Jorg Liesen. *Krylov Subspace Methods: Principles and Analysis*. Oxford University Press, 2015. ISBN 978-0198739043.

[17] Erin Carson, Miroslav Rozložník, Zdeněk Strakoš, Petr Tichý, and Miroslav Tůma. The numerical stability analysis of pipelined conjugate gradient methods: Historical context and methodology. *SIAM Journal on Scientific Computing*, 40(5):A3549–A3580, 2018. URL https://epubs.siam.org/doi/10.1137/16M1103361.

[18] Gérard Meurant. Multitasking the conjugate gradient method on the cray x-mp/48. *Parallel Computing*, 5:267–280, 1987. URL https://www.sciencedirect.com/science/article/abs/pii/0167819187900378.

[19] A.T. Chronopoulos and C.W. Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2): 153–168, 1989. ISSN 0377-0427. doi: https://doi.org/10.1016/0377-0427(89)90045-9. URL https://www.sciencedirect.com/science/article/pii/0377042789900459.

[20] P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 40(7): 224–238, 2014. URL https://doi.org/10.1016/j.parco.2013.06.001.

[21] The university of Florida. Suitesparse matrix collection. URL https://sparse.tamu.edu. (Last accessed on 2023/11/25).

[22] Scott Griffiths. Python module bitstring. URL https://pypi.org/project/bitstring/. (Version 4.1).

[23] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663. URL https://doi.org/10.1145/2049662.2049663.

[24] G. Meurant and P. Tichý. Approximating the extreme ritz values and upper bounds for the a-norm of the error in cg. *Numerical Algorithms*, 82:937–968, 2019. doi: https://doi.org/10.1007/s11075-018-0634-8. URL https://doi.org/10.1007/s11075-018-0634-8.

List of Figures

List of Tables

A. Appendices

A.1 Initialization procedures

This appendix contains initialization procedures for all CG variants mentioned in the thesis. The initializations are adapted from a general all-encompassing initialization procedure presented in Appendix D of [1].

Algorithm A.1 Initialize (HS-CG and ChG-CG)

procedure INITIALIZE($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
    $\mathbf{r_0} = \mathbf{b} - \mathbf{A}\mathbf{x_0}$
    $\nu_0 = \langle \tilde{\mathbf{r}}_\mathbf{0}, \mathbf{r_0} \rangle$
    $\mathbf{p_0} = \tilde{\mathbf{r}}_\mathbf{0}$
    $\mathbf{s_0} = \mathbf{A}\mathbf{p_0}$
    $\alpha_0 = \nu_0 / \langle \mathbf{p_0}, \mathbf{s_0} \rangle$
end procedure

Algorithm A.2 Initialize (PR-CG)

procedure INITIALIZE($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
    $\mathbf{r_0} = \mathbf{b} - \mathbf{A}\mathbf{x_0},\ \tilde{\mathbf{r}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{r_0}$
    $\mathbf{p_0} = \tilde{\mathbf{r}}_\mathbf{0}$
    $\mathbf{s_0} = \mathbf{A}\mathbf{p_0},\ \tilde{\mathbf{s}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{s_0}$
    $\sigma_0 = \langle \tilde{\mathbf{r}}_\mathbf{0}, \mathbf{s_0} \rangle$
    $\gamma_0 = \langle \tilde{\mathbf{s}}_\mathbf{0}, \mathbf{s_0} \rangle$
    $\nu_0 = \langle \tilde{\mathbf{r}}_\mathbf{0}, \mathbf{r_0} \rangle$
    $\alpha_0 = \nu_0 / \langle \mathbf{p_0}, \mathbf{s_0} \rangle$
end procedure

Algorithm A.3 Initialize (M-CG)

procedure INITIALIZE($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
    $\mathbf{r_0} = \mathbf{b} - \mathbf{A}\mathbf{x_0},\ \tilde{\mathbf{r}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{r_0}$
    $\mathbf{p_0} = \tilde{\mathbf{r}}_\mathbf{0}$
    $\mathbf{s_0} = \mathbf{A}\mathbf{p_0},\ \tilde{\mathbf{s}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{s_0}$
    $\gamma_0 = \langle \tilde{\mathbf{s}}_\mathbf{0}, \mathbf{s_0} \rangle$
    $\nu_0 = \langle \tilde{\mathbf{r}}_\mathbf{0}, \mathbf{r_0} \rangle$
    $\alpha_0 = \nu_0 / \langle \mathbf{p_0}, \mathbf{s_0} \rangle$
end procedure

Algorithm A.4 Initialize (Pipe-PR-CG)

1: **procedure** INITIALIZE($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
2: $\quad \mathbf{r_0} = \mathbf{b} - \mathbf{A}\mathbf{x_0},\ \tilde{\mathbf{r}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{r_0}$
3: $\quad \mathbf{p_0} = \tilde{\mathbf{r}}_\mathbf{0}$
4: $\quad \mathbf{s_0} = \mathbf{A}\mathbf{p_0},\ \tilde{\mathbf{s}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{s_0}$
5: $\quad \mathbf{u_0} = \mathbf{A}\tilde{\mathbf{s}}_\mathbf{0},\ \tilde{\mathbf{u}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{u_0}$
6: $\quad \mathbf{w_0} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{0},\ \tilde{\mathbf{w}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{w_0}$
7: $\quad \sigma_0 = \langle \tilde{\mathbf{r}}_\mathbf{0}, \mathbf{s_0} \rangle$
8: $\quad \gamma_0 = \langle \tilde{\mathbf{s}}_\mathbf{0}, \mathbf{s_0} \rangle$
9: $\quad \nu_0 = \langle \tilde{\mathbf{r}}_\mathbf{0}, \mathbf{r_0} \rangle$
10: $\quad \alpha_0 = \nu_0 / \langle \mathbf{p_0}, \mathbf{s_0} \rangle$
11: **end procedure**

Algorithm A.5 Initialize (Unpreconditioned Pipe-PR-CG)

1: **procedure** INITIALIZE($\mathbf{A}, \mathbf{b}, \mathbf{x_0}$)
2: $\quad \mathbf{r_0} = \mathbf{b} - \mathbf{A}\mathbf{x_0}$
3: $\quad \mathbf{p_0} = \mathbf{r_0}$
4: $\quad \mathbf{s_0} = \mathbf{A}\mathbf{p_0}$
5: $\quad \mathbf{u_0} = \mathbf{A}\mathbf{s_0}$
6: $\quad \mathbf{w_0} = \mathbf{A}\mathbf{r_0}$
7: $\quad \sigma_0 = \langle \mathbf{r_0}, \mathbf{s_0} \rangle$
8: $\quad \gamma_0 = \langle \mathbf{s_0}, \mathbf{s_0} \rangle$
9: $\quad \nu_0 = \langle \mathbf{r_0}, \mathbf{r_0} \rangle$
10: $\quad \alpha_0 = \nu_0 / \langle \mathbf{p_0}, \mathbf{s_0} \rangle$
11: **end procedure**

Algorithm A.6 Initialize (Pipe-M-CG)

1: **procedure** INITIALIZE($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
2: $\quad \mathbf{r_0} = \mathbf{b} - \mathbf{A}\mathbf{x_0},\ \tilde{\mathbf{r}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{r_0}$
3: $\quad \mathbf{p_0} = \tilde{\mathbf{r}}_\mathbf{0}$
4: $\quad \mathbf{s_0} = \mathbf{A}\mathbf{p_0},\ \tilde{\mathbf{s}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{s_0}$
5: $\quad \mathbf{u_0} = \mathbf{A}\tilde{\mathbf{s}}_\mathbf{0},\ \tilde{\mathbf{u}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{u_0}$
6: $\quad \mathbf{w_0} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{0},\ \tilde{\mathbf{w}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{w_0}$
7: $\quad \gamma_0 = \langle \tilde{\mathbf{s}}_\mathbf{0}, \mathbf{s_0} \rangle$
8: $\quad \nu_0 = \langle \tilde{\mathbf{r}}_\mathbf{0}, \mathbf{r_0} \rangle$
9: $\quad \alpha_0 = \nu_0 / \langle \mathbf{p_0}, \mathbf{s_0} \rangle$
10: **end procedure**

Algorithm A.7 Initialize (GV-CG)

1: **procedure** INITIALIZE($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x_0}$)
2: $\quad \mathbf{r_0} = \mathbf{b} - \mathbf{A}\mathbf{x_0},\ \tilde{\mathbf{r}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{r_0}$
3: $\quad \mathbf{w_0} = \mathbf{A}\tilde{\mathbf{r}}_\mathbf{0}$
4: $\quad \nu_0 = \langle \tilde{\mathbf{r}}_\mathbf{0}, \mathbf{r_0} \rangle$
5: $\quad \mathbf{p_0} = \tilde{\mathbf{r}}_\mathbf{0}$
6: $\quad \mathbf{s_0} = \mathbf{A}\mathbf{p_0},\ \tilde{\mathbf{s}}_\mathbf{0} = \mathbf{M^{-1}}\mathbf{s_0}$
7: $\quad \mathbf{u_0} = \mathbf{A}\tilde{\mathbf{s}}_\mathbf{0}$
8: $\quad \alpha_0 = \nu_0 / \langle \mathbf{p_0}, \mathbf{s_0} \rangle$
9: **end procedure**

Algorithm A.8 Initialize (FT-Pipe-PR-CG and AFT-Pipe-PR-CG)

1: **procedure** INITIALIZE($\mathbf{A}, \mathbf{b}, \mathbf{x_0}$)
2: $\quad \mathbf{x_0} = \mathbf{x_0}$
3: $\quad \mathbf{r_0} = \mathbf{b} - \mathbf{A}\mathbf{x_0}$
4: $\quad \mathbf{p_0} = \mathbf{r_0}$
5: $\quad \mathbf{s_0} = \mathbf{A}\mathbf{p_0}$
6: $\quad \mathbf{u_0} = \mathbf{A}\mathbf{s_0}$
7: $\quad \mathbf{w_0} = \mathbf{A}\mathbf{r_0}$
8: $\quad \mathbf{w'_0} = \mathbf{w_0}$
9: $\quad \sigma_0 = \langle \mathbf{r_0}, \mathbf{s_0} \rangle$
10: $\quad \gamma_0 = \langle \mathbf{s_0}, \mathbf{s_0} \rangle$
11: $\quad \nu_0 = \langle \mathbf{r_0}, \mathbf{r_0} \rangle$
12: $\quad \nu'_0 = \nu_0$
13: $\quad \mu_0 = \langle \mathbf{p_0}, \mathbf{s_0} \rangle$
14: $\quad \alpha_0 = \nu_0 / \mu_0$
15: $\quad \beta_0 = 1$
16: $\quad$ **Copy above variables into their versions with index** -1 **and** -2**, e.g.,** $\mathbf{x_0}$ **to** $\mathbf{x_{-1}}$ **and** $\mathbf{x_{-2}}$**.**
17: $\quad ||\mathbf{r_0}|| = \sqrt{\nu_0}$
18: $\quad ||\mathbf{s_0}|| = \sqrt{\gamma_0}$
19: $\quad ||\mathbf{p_0}|| = ||\mathbf{p_0}||$
20: **end procedure**

Note that lines 17 to 19 in Algorithm A.8 are necessary for the computation of $B_{\nu'_1}$, $B_{\mathbf{w'_1}}$, and $B_{\mu'_1}$.

A.2 Behavior of the relative μ-gap/bound difference for the matrix *aft01*

All runs for this matrix would for both detection sets (union of the four detection criteria with some set threshold) end up as false positive, since the thresholds would be violated immediately at the start of the computation before any bit flip. However, lowering the threshold would not result in a successful detection, because the values of the relative μ-gap/bound difference at the flip iteration and one iteration later are at the same level as when there are no flips at all.

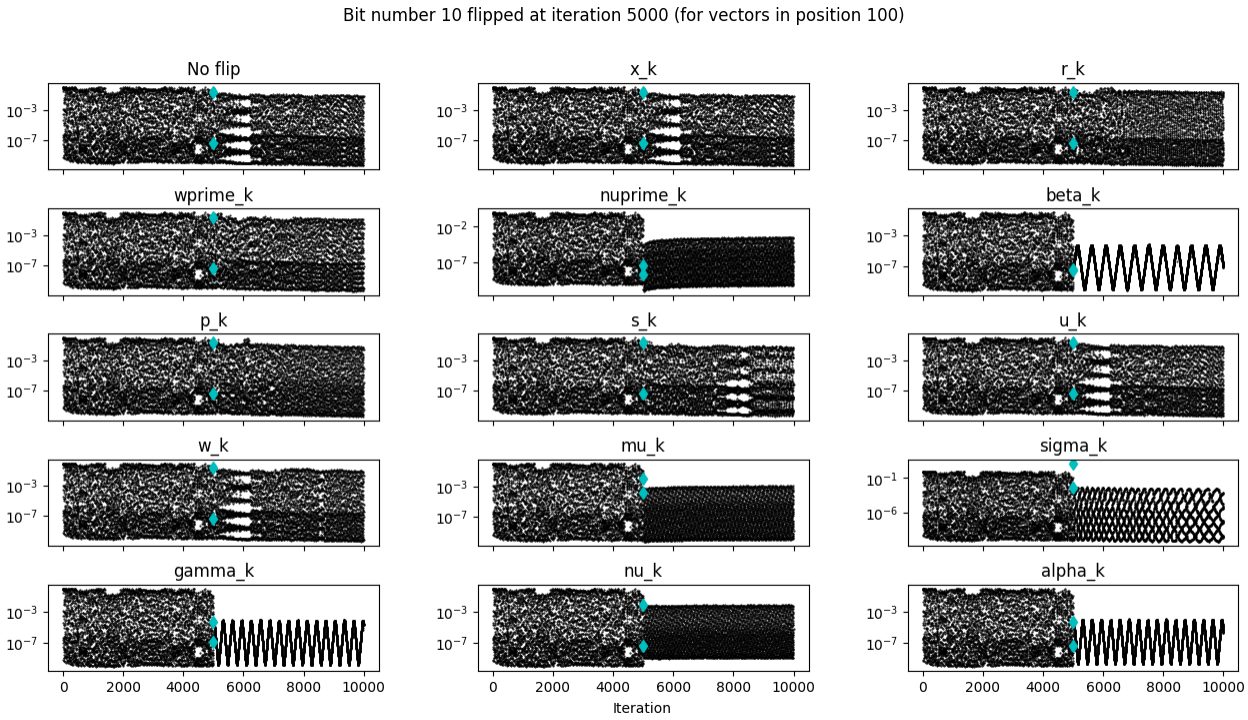

Figure A.1: Relative μ-gap/bound difference, $|B_{\mu'_k} - \Delta_{\mu'_k}|/B_{\mu'_k}$, matrix *aft01*