MASTERARBEIT

Titel der Masterarbeit

„Analysis and Evaluation of Binary Cascade Iterative Refinement and Comparison to other Iterative Refinement Algorithms for Solving Linear Systems“

Verfasser

Karl E. Prikopa, BSc

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2011

Studienkennzahl lt. Studienblatt:	A 066 940
Studienrichtung lt. Studienblatt:	Scientific Computing UG2002
Betreuer:	Univ.-Ass. Privatdoz. Dr. Wilfried Gansterer

Abstract

Iterative refinement is a widely used method to improve the round-off errors of a solution of a linear system and is also used in software packets like LAPACK. The cost of the iterative improvement is very low compared to the cost of the factorization of the matrix but results in a solution which can be accurate to machine precision. Many variations on the standard iterative refinement method exist, which use different working precisions to refine the solution. The extra precise iterative refinement can use extended precision to improve the result. The mixed precision iterative refinement tries to exploit the benefits of using lower precisions to compute a solution and then uses iterative refinement to achieve the higher precision accuracy.

The focus of this thesis will be the binary cascade iterative refinement (BCIR), which chooses the working precisions according to the input data. This algorithm depends on arbitrary precision arithmetic to support working precisions outside the IEEE standard data types provided by most hardware vendors. The thesis will analyse the properties of BCIR and conduct experiments which will compare the algorithm to other iterative refinement methods and focus on the numerical accuracy and the convergence.

The arbitrary precision arithmetic will be implemented using the GNU MPFR software library. The simulated arbitrary precision does not provide accurate information about the gains and losses in performance due to the use of the different precisions. Therefore a performance model will be introduced in order to be able to compare the performance of the algorithms and to analyse the possible performance gains, which could be exploited in future works by hardware implementations for example using reconfigurable hardware like FPGAs.

Contents

List of Figures

Listings

Chapter 1

Introduction

The use of iterative refinement can improve the round-off errors of a solution of a linear system. The process computes the residual of the solution, then solves the system for a correction term using the residual as the right hand side of the equation and finally updates the solution with the correction term. These steps are repeated until the requested accuracy is reached. This method was first mentioned by Wilkinson in his book “Rounding Errors in Algebraic Processes” [49] using fixed point arithmetic and later expanded by Moler [35] to cover floating-point arithmetic. The cost of the iterative improvement is very low compared to the cost of the factorization but it results in a solution which can be accurate to machine precision. Iterative refinement is not limited to linear solvers, but can also be used for many other solvers, including eigensolvers and least squares solvers (for example [12, 18]). However, this thesis will focus on iterative refinement using linear solvers.

The standard iterative refinement method (SIR) uses the same precision to compute both, the initial solution and the correction term for the improved result, but other iterative refinement methods exist, which use different precisions for these computation steps. The Extra Precise Iterative Refinement (EPIR) [13] uses a higher precision to compute the residual and the correction term of the solution to compensate for slow convergence and ill-conditioned systems. The Mixed Precision Iterative Refinement (MPIR) [7] takes a different approach and computes the matrix decomposition and the initial solution in single precision and applies iterative refinement using double precision to improve the result and still have a solution which reaches double precision accuracy. This exploits the benefits of using the lower single precision, for example exploiting vector instructions and using less storage which also reduces the amount of data moved through the memory hierarchy, while still achieving a double precision result.

The focus of my master thesis will lay on the *Binary Cascade Iterative Refinement* (BCIR) by Kiełbasiński [26]. The algorithm adapts the precisions for computing the refinement steps depending on the input parameters, the size and condition number of the matrix and the intended target precision. The process can use multiple working precisions throughout the refinement process. This provides the ability to choose the appropriate precision to improve the result and also compensate for ill-conditioned systems. This algorithm has never been implemented and therefore no experimental results are available up until now.

The algorithm depends on arbitrary precision, which is not bound to IEEE standard precision. The hardware support for arbitrary precision is very limited and therefore a software library, the GNU Multiple Precision Floating-Point Reliable Library (GNU MPFR) [15], will be used to implement the iterative refinement methods. This library provides a portable implementation of arbitrary precision and allows the precisions to be set exactly in the number of bits stored in the mantissa of a floating-point number.

The binary cascade iterative refinement method will be compared to other iterative refinement methods and the numerical accuracy and the convergence will be analysed. The numerical behaviour of the binary cascade iterative refinement method will be analysed for different input systems, which will also include extremely ill-conditioned Hilbert matrices. Due to the use of software simulated arbitrary precision, performance measurements would not provide accurate information about the gains and losses in performance due to the use of the different precisions. Therefore a performance model will be introduced in order to be able to compare the performance of the algorithms and to analyse the possible performance gains. These result could be used in future works for hardware implementations for example using reconfigurable hardware like FPGAs.

1.1 Thesis Outline

The binary cascade iterative refinement depends on arbitrary precision. Therefore Chapter 2 will introduce arbitrary precision, describe the differences to the IEEE Standard for Floating-Point Arithmetic (IEEE 754) [10] and introduce the multiple precision floating-point reliable software library (GNU MPFR) [15] which will be used to implement arbitrary precision for the iterative refinement algorithms.

Chapter 3 will describe the different iterative refinement methods which will be compared in this thesis. Beside the already widely used standard iterative refinement (SIR) [49], the extra precise iterative refinement (EPIR) [13] will be introduced, which extends the standard iterative refinement by adding error bounds at a low computational cost and also uses higher precisions than the targeted precision to compute critical steps in the iterative process. The mixed precision iterative refinement (MPIR) [7] is another algorithm which will be used in the experiments. It focuses on the possibility of exploiting performance benefits based on the use of lower working precisions for the computationally expensive tasks, the matrix factorization, while still achieving the target precision accuracy, the same accuracy as the standard iterative refinement. In addition to the description of the available iterative refinement methods, a model to estimate the number of iterations used by the standard and mixed precision iterative refinement is introduced in this chapter, which will later be used as part of the performance models.

The binary cascade iterative refinement (BCIR) is explained and analysed in Chapter 4. The algorithm described by Andrzej Kiełbasiński is introduced, accentuating on the properties which decide which precisions should be used to compute a result with the described accuracy.

Chapter 5 will describe the implementation of the algorithms and their usage. The chapter will also include implementation details concerning the different algorithms.

The experiments are the focus of Chapter 6 and will include the numerical accuracy in terms of accuracy by analysing the relative residual achieved by the different algorithms and the convergence by observing the number of iterations needed to achieve the specified target precision. Included in this chapter is the description of the method used to generate the data and information on the chosen termination criteria for the iterative refinement methods.

A meaningful performance analysis is almost impossible when using arbitrary precision arithmetic which is being emulated using software packages, especially when operating at such low precision levels which are below or just above the standard precision range. Therefore a performance model is used which represents the theoretical performance gains and losses if the algorithms would be implemented using field programmable gate arrays (FPGAs) [16], which provide the ability to set the precision of the data types exactly to the number of bits stored in the mantissa. This ability is also supported by the GNU MPFR software library, which is one of the main reasons it has been chosen for the experiments. The performance model is described in Chapter 7 and the results of the comparison of the performance of the different iterative refinement algorithms will be analysed.

Chapter 2

Precisions of Floating Point Arithmetic

This chapter will introduce arbitrary precision, which will allow computations outside the boundaries set by the standardized floating-point precision formats, which are the predominant precisions implemented in most modern day processors.

Firstly arbitrary precision and the differences, advantages and disadvantages compared to standard precision will be introduced followed by a number of available possibilities of using arbitrary precisions. The arbitrary precision library MPFR [15, 43], which will be used in the implementation of the binary cascade iterative refinement, will be looked at in greater detail. At the end of this chapter, an analysis of the performance of the arbitrary precision package will be shown.

2.1 Standard Precision

2.1.1 History

In modern day computing, one can normally resort to standardised floating-point precisions available on most commodity processors which follow the *IEEE Standard for Floating-Point Arithmetic* (IEEE 754) [10]. This standard was first finalized and released in 1985 and became the leading standard for floating-point arithmetic followed by the majority of processor vendors and is implemented in most modern commodity processors. The standard was revised and extended in 2008 after a seven year review process and still comprises the majority of the definitions from the original document, including the definition of the standard floating-point formats.

Today, programmers can often rely on the IEEE standard being implemented on the target processors and the processor implementations to follow the IEEE 754 specifications. This was not always the case. Before the standard was passed and adopted by the leading processor manufacturers, interoperability among processors from different vendors was almost out of the question. Often this was not limited to the vendors themselves but also true for processors produced by the same firm [8].

The different floating-point formats focused on different aspects of representing a real number and included the range, i.e. the amount of numbers that could be represented, the speed of computations, rounding of results and of course the accuracy of the represented number. The manufacturers tried to sell their format as the "accurate" floating-point implementation, but all representations had their drawbacks and whatever implementation one chose, a compromise had to be made in at least one of the previously mentioned aspects.

This growing issue needed to be addressed because all these incompatible floating-point representations made the life of a developer very difficult and an implementation very dependent on the underlying hardware, which reduced the portability of a program and increased the development time and costs. The behaviour of an algorithm was difficult to predict and the results were inconsistent due to the different formats.

In 1977 the first IEEE 754 standards working group meeting took place with the goal to define a standard for the floating-point formats. The IEEE working group had the benefit of many different floating-point representations being in use and their properties could be analysed in order to avoid their disadvantages. One of the driving forces behind the standardization process was Intel [9]. In 1976, Intel was developing a floating-point co-processor, the i8087, for their new i8086/8 microprocessor and they wanted to use a new floating-point arithmetic, which would be better than any other format used by their competitors and also be applicable to a large market. With the help of William Kahan, who was engaged as a consultant for the new floating-point format and had previously worked for Hewlett Packard and improved their processing capabilities, a specification was formulated for the new microprocessor arithmetic. After the first meeting of the IEEE 754 working group, William Kahan asked for permission from Intel to take part in the standardization process using the newly defined floating-point format developed for the i8087 co-processor. He pointed out in an Interview he gave in 1998 [9] that it was very difficult for him to present the standard because he of course was not allowed to reveal details about the upcoming Intel processor architecture or its transcendental functions (e.g. sine, cosine, logarithms, etc.). He could describe the reasoning behind the proposals, but not how they were going to be implemented. There were still some questions left open, some of which greatly hindered the completion of the standard for many years. One of the disputed aspects of the standard was underflow, where a result is between the smallest normalized number and zero. For many years the working group could not agree on a standard form to handle underflows.

When the standard was finally concluded, eight years had passed since the first meeting. Luckily many manufacturers saw the potential of a standard floating-point format and had already started using some early drafts of the standard before it was officially finalised. The IEEE 754 standard was adopted very quickly by most microprocessor manufacturers. However, many leading high performance computing vendors had to continue to support their proprietary floating-point format for many years after the introduction of the standard due to their large base of customers. Today the IEEE standard has become the dominant floating-point standard and provides the users with a portable floating-point format. [8]

Even though the standard provides a uniform representation of a floating-point number, there are still some issues which have not been clearly defined in the standard and therefore can vary between different architectures leading to different results produced by the same program or algorithm [42, p. D-65]. The main issue is the handling of intermediate results, which are stored in a "destination" variable which does not have to use the same precision as the variables of the expression. By design, the standard only defines that the results must be rounded correctly to the destination's precision but it does not define the precision of the destination variable. This choice is normally made by the system or the programming language without the ability of the user to change it. The same program can therefore return significantly different results depending on the implementation of the IEEE standard.

2.1.2 IEEE Standard

The main focus for the following descriptions will be the IEEE 754-1985 standard because this is the one followed by the GNU MPFR arbitrary precision package which will be used to implement the iterative refinement algorithms in this thesis.

The IEEE 754-1985 standard and its revised version of 2008 (IEEE 754-2008) [10] specifies binary (and decimal) floating-point formats, conversions between different formats, arithmetic operations, rounding modes and floating-point exceptions.

One goal of the IEEE 754-1985 standard was to find a uniform way of rounding floating-point numbers correctly. The standard therefore defines the following rounding modes.

- **Round to nearest, ties to even**: rounds the result to the nearest number. If the result is not representable then the nearest number with the even least significant digit will be returned. For example, if the rounding mode would be applied to decimal numbers, 47.5 would be rounded to 48 and 46.5 would be rounded to 46. This rounding mode is unbiased.
- **Round to nearest, ties away from zero**: rounds the result to the nearest number. If the result is not representable then it is rounded to the nearest number away from zero. For example, -47.5 would be rounded to -48 and 47.5 would be rounded to 48. This rounding mode was included in the revised version of the standard (IEEE 754-2008).
- **Round toward** 0: rounds the result to zero, in other words it truncates the number
- **Round toward** $+\inf$: rounds the number to positive infinity
- **Round toward** $-\inf$: rounds the number to negative infinity

The last three rounding modes are also called directed rounding.

The standard defines four binary floating-point representations:

- **binary16** - Half precision (11 bit mantissa)
- **binary32** - Single precision (24 bit mantissa)
- **binary64** - Double precision (53 bit mantissa)
- **binary128** - Quadruple precision (113 bit mantissa)

In Figure 2.1 the general representation of a floating-point number is shown. The parts of

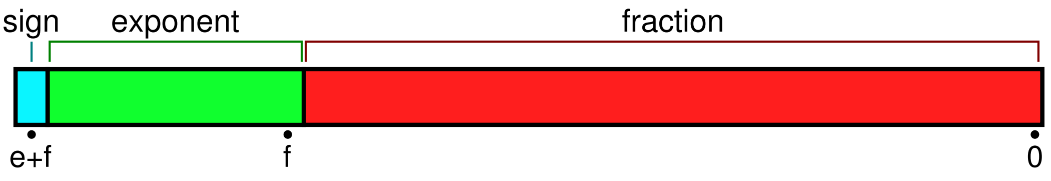

Figure 2.1: General representation of floating-point numbers [48]

the floating-point representation are best described using an example. The *double precision floating-point format* or *binary64*, the new name for double precision since the 2008 revision of the standard, uses 8 bytes to store a number. The 64 bit of available storage is divided into the three sections shown in Figure 2.1 as follows:

- Sign bit (blue): the first bit is used to store the sign of the number.
- Exponent (green): in this case the exponent has a width of 11 bits.
- Significand precision (red): the significand, which is often also called the mantissa due to historical reasons, uses 53 bits to represent the number in binary, although only 52 bits are stored explicitly.

In order to explain why only 52 bits are needed to be stored explicitly, but still can represent 53 bits, it is necessary to take a closer look at the format the numbers are stored in: the normalized numbers. A number is called *normalized* if it has the form

$$\pm d_0.d_1d_2d_3\ldots \times b^n \tag{2.1}$$

b stands for the base of the representation and n is an integer representing the exponent of the base b. The digits d_i are integers between 0 and $b-1$ and $d_0 \neq 0$. Storing the numbers in this representation leads to the most significant bit always being 1. It is unnecessary to store this bit in the standard binary precision formats. Therefore it is often also called the "hidden bit". This beneficial property of the normalized numbers can only be exploited in floating-point representations using two as its base and not with other bases.

The number of bits in the significand can be used to determine how many digits of any other base can be stored in this representation. The following equation is the general form which can be used to convert the number of digits between all bases.

$$d_{b_1} = d_{b_2} \cdot log_{b_1}(b_2) \tag{2.2}$$

b_i stands for the base and d_{b_i} describes the number of digits in base b_i. To calculate the number of digits from binary to decimal the equation from (2.2) would be:

$$d_{10} = d_2 \cdot log_{10}(2) \tag{2.3}$$

For the IEEE standard double precision floating-point format with 53 bits stored in the mantissa this would lead to $15.96\ldots \approx 16$ representable decimal digits.

$$d_{10} = 53 \cdot \log_{10}(2) = 15.95458977 \tag{2.4}$$

A good approximation for the number of decimal digits represented by the binary mantissa can be achieved by multiplying the number of bits in the mantissa by 0.3.

$$d_{10} = d_2 \cdot 0.3 \tag{2.5}$$

2.2 Arbitrary Precision

Arbitrary precision allows the user to choose which precision should be used for a calculation or preferably which precision should be used for each variable to store a value. Arbitrary precision is not bound to machine dependent or IEEE standard types and the precision is only limited by the memory the host system can provide. Therefore arbitrary precision is also often called "infinite-precision arithmetic", which of course is not true in practice because of the finite amount of memory which is available or the limits imposed by index variables and other natural boundaries. Even though limitations exist, the range of precision which can be provided through the use of arbitrary precision is still very large.

Some arbitrary precision libraries allow the user to set the precision exactly to the desired number of bits to store in the mantissa. However, other arbitrary precision systems only provide the ability to increase the precision in steps of the machine word size, which is normally 32 or 64 bits wide. This approach is sometimes called multiple precision, where the significand of a binary number is distributed over multiple machine words [17].

Arbitrary precision has a wide range of applications, some of which are in use in every day life. Arbitrary precision plays a great role in cryptography. The long encryption keys can be integer numbers with hundreds or thousands of digits which could not be represented by standard integer types provided by most programming languages. This is an ideal task for arbitrary precision integer arithmetic and is present in modern web browsers using public-key cryptography. Other common applications of arbitrary precision include calculating mathematical constants like π or the ability to prevent overflows and underflows by increasing the precision of computations. Of course arbitrary precision is used to increase the accuracy of computations as for example in the binary cascade iterative refinement by using arbitrary precision to increase the accuracy of a computed result.

The advantages of using arbitrary precision to increase the available range and gain higher accuracy come with drawbacks. In most cases, arbitrary precision can only be simulated, either in hardware or software, which leads to a significant performance decrease. Any kind of simulation increases the runtime of algorithms compared to using the fast standard data types. Most commodity processors only support the standard IEEE 754 data types and alternatives like field programmable gate arrays (FPGAs), which can be programmed by the user, insert an additional layer of complexity and cannot provide the same speed as the standard processors optimized to operate with the standard data types. Another challenge of arbitrary precision is the special algorithms required by libraries to perform the computations which have to handle large significands. Sometimes arbitrary precision is the only way to compute a result accurately due to the limited precisions provided by the standard data types or the increased complexity of encryption algorithms. Being able to simulate the behaviour of arbitrary precision data types can help us find more efficient ways to compute more accurately and, for example, in the case of iterative refinement also show the theoretical benefits of non standard precisions in terms of runtime performance.

2.2.1 Constant Folding with Arbitrary Precision

Even though most of the time arbitrary precision comes with performance drawbacks, it can also be used in the preprocessing phase to increase the accuracy of constants before they are handled with standard precision data types. A compiler normally replaces constant expressions with their final value in order to reduce the need of recomputing the same result every time the program executes the line containing the constant. This procedure is called *constant folding* [17]. The GNU GCC compiler started to use the GNU MPFR library with version 4.3 to handle constant folding and evaluate mathematical functions applied to constants at compile time at arbitrary precision when optimizations are activated. By using the arbitrary precision library, the result of the mathematical operations does not rely on the underlying architecture and provides reproducibility and correctness.

An example of the effects caused by insufficient precision and inaccurate rounding is provided in [17] and it is also shown how this problem was resolved when the example program was compiled with the latest GCC compiler using the MPFR library for constant folding. The authors provided a simple program to calculate the value of `sin(x)`. Without any optimization (compiler flag `-O0`) the MPFR library was not used and the result was not correctly rounded, but when they increased the level of optimization by adding the compiler flag `-O1`, the result was correctly rounded. In addition, the standard C mathematical library was no longer required to be linked with the program due to the use of the MPFR library.

2.3 Arbitrary-precision Software and Libraries

As mentioned in the last section, there are different ways to achieve arbitrary precision. This section will focus on some of the available methods in hardware and software which provide the ability to use the benefits of arbitrary precision. The methods can be classified into the following three categories:

- Hardware support
- Stand-alone software
- Programming languages and software libraries

2.3.1 Hardware support

The hardware support for arbitrary precision is limited, one example being FPGAs. FPGA stands for field programmable gate array [47] and allows the user to reconfigure the hardware for different applications after the production phase, sometimes even at runtime. The processors contain programmable logic blocks and interconnects and support massive parallelism. A common field for the use of FPGAs is prototype development of application-specific integrated circuits, but the chips are also used for signal processing, cryptography, speech recognition, medical imaging and many other applications. The users can implement a custom instruction set which only consists of instructions which are relevant for the application. Complex functions normally not supported by commodity processors can also be implemented by directly programming the logic gates. The disadvantages of FPGAs include the low speeds for general-purpose arithmetic and the high complexity of designing and programming the chip.

FPGAs are not bound to the IEEE data types and can operate at arbitrary precisions. Lower precisions result in a higher performance due to the increased parallelism and higher throughput, as shown in [36].

2.3.2 Stand-alone Software

There are some applications which include the ability to use arbitrary precision arithmetic. One example is the computer algebra system Maple [31]. Since version 11 the software started using the GNU MPFR library [43] to provide arbitrary precision arithmetic.

Another example is Matlab's Variable Precision Arithmetic (vpa). Matlab provides the ability to perform calculations in arbitrary precision using the `vpa`-command included in the Symbolic Math Toolbox [33]. Matlab can evaluate calculations at variable precision making it possible to increase or decrease the accuracy of a calculation. The `vpa`-command takes two arguments:

- The first argument is the expression to evaluate in the specified precision,
- the second argument specifies the requested precision by defining the number of decimal digits to use.

The return value of the `vpa`-command is a symbolic object, a special data type provided in the Symbolic Math Toolbox software. Many Matlab functions also accept symbolic expressions as input parameters and therefore can facilitate the development of an algorithm using arbitrary precision. Matlab also offers the ability to construct symbolic numbers, which store the numeric values as a symbolic representation using higher precision. A symbolic object can be created passing a numeric scalar or matrix variable as the first parameter to the function `sym`. The newer Matlab versions using Mupad as their symbolic engine show an enormous decrease in performance compared to the performance of older Matlab versions using Maple as their symbolic engine.

2.3.3 Programming languages and software libraries

Many different software libraries exist for many different programming languages and each one has advantages and disadvantages. The libraries differ in the way they store arbitrary precision numbers, in the way they round intermediate results or which data types they provide arbitrary precision for (integer, floating-point, rationals or decimals). Table A.1 shows a list of available packages which provide arbitrary precision arithmetic in different programming languages and with different data types. In the following paragraphs two arbitrary precision packages will be introduced and some of their advantages and disadvantages will be discussed.

The first package is ARPREC which stands for "ARbitrary PRECision Computation Pakackage" [2] and provides support for integers, binary floating-point numbers and complex binary floating-point numbers in arbitrary precision. ARPREC is written in C++ and provides bindings for C++ and Fortran-90. The package uses operator overloading provided by C++ to facilitate development and therefore requires only minor changes to existing source codes. The precision for floating-point numbers can be set by specifying the number of decimal numbers to be represented. One drawback to this package is that the precision can only be set globally for all arbitrary precision variables and it is therefore not possible to use two different precision in the same calculation. The performance is also not very high, as is shown in Section 2.5.

The Multiple Precision Floating-Point Reliable Library (GNU MPFR) is an arbitrary precision library for floating-point numbers written in C. The great advantage of the GNU MPFR library over many other libraries is the ability to set the precision of each variable independently and to set the size of the mantissa to exactly the number of bits required. This also allows computations to be performed with precisions lower than 53 bits. In fact, the lower limit of MPFR is 2 bits for the size of the mantissa. This property is one of the main reasons why this library was chosen for the implementation of the binary cascade iterative refinement algorithm. The GNU MPFR library, its properties and usage will be explained in more detail in Section 2.4.

An external library is not always necessary. Some programming languages provide built-in support for arbitrary precision data types and arithmetic or include them in the standard library of their language. Beginning with .NET Framework 4, a BigInteger structure [34] has been introduced for Visual Basic, C#, C++ and F# which provides the ability to represent an integer with an arbitrary precision. Java also supports a BigInteger class [38] for arbitrarily large integers and has a BigDecimal class [37] which is used to represent decimal numbers using arbitrary precision. A further example of a programming languages which provides built-in support of arbitrary precision is Python, although with Python it is only possible to set all integer variables to use arbitrary precision and not to limit the use of increased precision to a single variable without the help of an external package.

2.4 GNU Multiple Precision Floating-Point Reliable Library (MPFR)

As mentioned before, the GNU MPFR library is used for the implementation of the iterative refinement algorithms and is therefore explained in more detail in this section.

The **M**ultiple **P**recision **F**loating-Point **R**eliable Library (GNU MPFR) [43, 15] is an arbitrary precision package for C/C++ and is based on the GNU Multiple-Precision Library (GMP) [20]. MPFR supports arbitrary precision floating-point variables and provides exact rounding of all implemented operations and mathematical functions. MPFR code is portable which means it will produce the same result regardless of the underlying hardware.

The library allows the user to set the precision of the arbitrary precision variables exactly by specifying the number of bits to use in the mantissa of the floating-point number. The number of bits in the mantissa has to include the hidden bit, which is normally not stored by the standard IEEE 754 standard floating-point formats. For example, to emulate IEEE double precision in MPFR, the precision of the variable has to be set to 53 bits. Due to the design of the library it is possible to work with any precision between 2 bits and the value specified by the constant `MPFR_PREC_MAX`, which can be as high as the maximum value of a `long int`. However, the precision should never be chosen near to the maximum precision, because MPFR has to increase the precision during computations to provide accurate results and correct rounding and a precision exceeding `MPFR_PREC_MAX` would lead to an undefined behaviour or even crash the program. The ability of MPFR to set the precision exactly to the desired precision in bits is one major difference of this library compared to its competitors and the main reason it was chosen for the implementation of the iterative refinement methods.

2.4.1 MPFR Variables

To use the MPFR functions and variables, it is necessary to include the MPFR header file.

```
#include <mpfr.h>
```

The main data type provided by the MPFR library is `mpfr_t`, which is "an arbitrary precision significand (mantissa) with a limited precision exponent." [43]. The precision has its own data type, `mpfr_prec_t`, which is a `typedef` and normally corresponds to an `int` or a `long int`. As mentioned before, the precision has a lower limit of 2 (`MPFR_PREC_MIN`) and an upper limit defined by the data type used for the precision data type. Therefore `MPFR_PREC_MAX` will normally either be the maximum number of an `int` or a `long int`.

`mpfr_t` is a pointer to an `__mpfr_struct`, which is shown in Listing 2.1. The structured type includes the three parts of a floating-point number as described in the IEEE 754 standard (see Figure 2.1). The first field defines the precision of the variable (`mpfr_prec_t _mpfr_prec`). The second field is used for the sign of the number (`mpfr_sign_t _mpfr_sign`) followed by the exponent of the floating-point number (`mpfr_exp_t _mpfr_exp`). The last field is a pointer to the words, called limbs, containing the significand. A limb has the size of a word which is normally 32 or 64 bits wide. The significand is split accross the limbs. If the total length of the limbs is larger than the precision, the remaining bits are filled up with zeros to ensure the user-defined number of bits are being used to represent the floating-point number.

```
/* Definition of the main structure */
typedef struct {
    mpfr_prec_t   _mpfr_prec;
    mpfr_sign_t   _mpfr_sign;
    mpfr_exp_t    _mpfr_exp;
    mp_limb_t    *_mpfr_d;
} __mpfr_struct;
```

Listing 2.1: The definition of the data type `mpfr_t`

Before `mpfr_t` variables can be used, they have to be initialized by calling the function `mpfr_init2`.

`void mpfr_init2 ( mpfr_t x, mpfr_prec_t prec )`

The precision is set to the value specified by `prec` and the value is set to “Not-a-Number”. The precision can be changed after initializing the variable, but should not be done via the `mpfr_init2` function, but rather by calling

`void mpfr_set_prec ( mpfr_t x, mpfr_prec_t prec ) .`

To assign values to the MPFR variables, MPFR provides special assignment functions for a large range of different input types. The following function is an example for assigning the value of a `double` to an MPFR variable.

`int mpfr_set_d ( mpfr_t rop, double op, mpfr_rnd_t rnd )`

This function will assign the value of the `double` variable in `op` to the `mpfr_t` variable passed to `rop` and will round the value using one of the rounding modes specified in the following subsection provided as the last parameter `rnd`. It is also possible to set a value by passing a string to the function `mpfr_set_str`, which is extremely useful for floating-point values, which cannot be represented exactly, e.g. “0.1”. A full list of all available assignment functions can be found in the GNU MPFR documentation [43].

2.4.2 MPFR Rounding Modes

MPFR supports exact rounding in compliance with the IEEE 754-2008 standard (described in Subsection 2.1.2). It implements four of the rounding modes specified by the standard as shown in the following list with their corresponding MPFR keywords. Their MPFR data type is `mpfr_rnd_t`.

- `MPFR_RNDN`: Round to nearest, ties to even
- `MPFR_RNDZ`: Round toward 0
- `MPFR_RNDU`: Round toward $+\inf$
- `MPFR_RNDD`: Round toward $-\inf$
- `MPFR_RNDA`: Round away from 0 (not in the IEEE 754-2008 standard)

2.4.3 MPFR Functions

The GNU MPFR library is written in C and therefore cannot use operator overloading. Consequently MPFR has to offer multiple functions for each operation, one for each supported data type as an input variable. Even the most basic arithmetic operations have do be performed using function calls.

Each function in MPFR starts with the prefix `mpfr_`. The syntax of MPFR functions is designed to resemble the assignment operator. The first parameter of each function is the destination variable, followed by the input values. The last argument is the rounding mode which should be used for the current operations and has to be one of the values described in the previous subsection. Most MPFR functions return a ternary integer value, which provides information about the correctness of the computed result:

- **0**: the value in the destination variable is exact.
- **Positive/Negative**: the value in the destination variable is greater/lower than the exact result.

For example, when using `MPFR_RNDD` as the rounding mode, the returned integer value will always be negative unless the result is exact.

The following list should demonstrate the amount of functions available to perform a basic mathematical operation, in this case for addition.

```c
int mpfr_add ( mpfr t rop, mpfr_t op1, mpfr_t op2, mpfr_rnd_t rnd )
int mpfr_add_ui ( mpfr_t rop, mpfr_t op1, unsigned long int op2, mpfr_rnd_t rnd )
int mpfr_add_si ( mpfr_t rop, mpfr_t op1, long int op2, mpfr_rnd_t rnd )
int mpfr_add_d ( mpfr_t rop, mpfr_t op1, double op2, mpfr_rnd_t rnd )
int mpfr_add_z ( mpfr_t rop, mpfr_t op1, mpz_t op2, mpfr_rnd_t rnd )
int mpfr_add_q ( mpfr_t rop, mpfr_t op1, mpq_t op2, mpfr_rnd_t rnd )
```

Listing 2.2: All available MPFR functions to add two numbers using different input data types

The first function is used to add two MPFR variables, the others are provided to operate directly with other data types without having to explicitly convert them to MPFR variables. The data types `mpz_t` and `mpq_t` are from the GMP library and are included for backward compatibility.

MPFR does not only provide functions for basic arithmetic functions (square root, power, absolute value, ...) but also all mathematical functions implemented in the C99 standard. This includes functions for logarithm, exponential, sine, cosine, gamma function and many others. The library also offers comparison functions to compare MPFR variables with each other and with other data types and conversion functions to convert MPFR variables to other data types or strings. Arbitrary precision floating-point numbers can also be printed with `mpfr_printf`, which works similar to the standard C `printf` function, but is enhanced with additional options for MPFR variables.

The large amount of functions for even the simplest arithmetic operations greatly increases the complexity of the source code, as can be seen in the following subsection.

2.4.4 MPFR Example Source Code

The following implementation of a dot product will be used to demonstrate the usage of MPFR and show the increase of complexity by substituting each operation by a function call and the prerequisites before being able to use a MPFR variable. The first source code in Listing 2.3 shows the dot product implemented in standard C using standard data types, in this case `double` for the input vectors and the output result. The second listing (Listing 2.4) shows the same operation implemented using the GNU MPFR library. The number of code lines necessary for such a simple computation has already doubled. The addtitional parameter `prec` is also required to specify the precision which should be used to compute the dot product in the MPFR implementation.

```
void cDot ( int n, double* a, double* b, double* sum ) {
    int i;
    *sum = 0.0;

    for ( i = 0; i < n; i++ )
        *sum += a[i]*b[i];
}
```

Listing 2.3: Dot product implemented in standard C

The MPFR implementation additionally requires a temporary variable to hold the result of the multiplication before adding the result to the dot product stored in `sum`. As with all MPFR variables it has to be initialized with the correct precision to store the computed result. In order to use the same precision as the dot product, the precision of the variable `sum` can be determined using the function `mpfr_get_prec` and used for the intermediate variable `mult`. In this case this is only used for demonstration purposes, as the precision is already known through the last parameter of the function call.

```
 1 void mpfrDot ( int n, mpfr_t* a, mpfr_t* b, mpfr_t* sum,
       mpfr_prec_t prec ) {
 2     int i;
 3
 4     mpfr_init2(*sum, prec); // set precision in bits
 5     mpfr_set_d(*sum, 0.0, MPFR_RNDN);
 6
 7     mpfr_t mult;
 8     mpfr_init2(mult, mpfr_get_prec(sum));
 9     for ( i = 0; i < n; i++ ) {
10         mpfr_mul(mult, a[i], b[i], MPFR_RNDN);
11         mpfr_add(*sum, *sum, mult, MPFR_RNDN);
12     }
13     mpfr_clear(mult);
14 }
```

Listing 2.4: Dot product implemented using the GNU MPFR library

The main operations in line 6 of Listing 2.3 require two lines in MPFR (lines 10 and 11 in Listing 2.4). Firstly the two values from the vectors have to be multiplied using `mpfr_mul`. Then the value has to be added to the dot product sum. Both operations use the rounding mode `MPFR_RNDN`, which rounds the results to the nearest value. Finally the temporary variable `mult` has to be released by calling `mpfr_clear`.

2.5 Performance Evaluation

Another interesting aspect of arbitrary precision libraries is the analysis of their performance. This was accomplished by using a matrix implementation in standard C and the same implementation being transformed using MPFR function calls. In order to compare the performance of MPFR to other arbitrary precision packages, the matrix multiplication was also implemented using the ARPREC package described in 2.3.3. All implementations were compared and the efficient matrix multiplication provided by the ATLAS BLAS [45] DGEMM function. As a metric for the comparison, the slow down effect was calculated using the number of total instructions (TI) measured by PAPI [6].

$$Slow\ down = \frac{TI_{ATLAS\ BLAS} - TI_{AP\ library}}{TI_{ATLAS\ BLAS}} \tag{2.6}$$

The details of the test system were as follows:

- INTEL Core 2 Quad Q9550 (2,83GHz, 12MB Cache)
- DDR2-RAM 2x2048 MB, PC2-800 MHz
- Ubuntu 9.10 Server with PAPI 3.7.0 and ATLAS BLAS 3.9.17

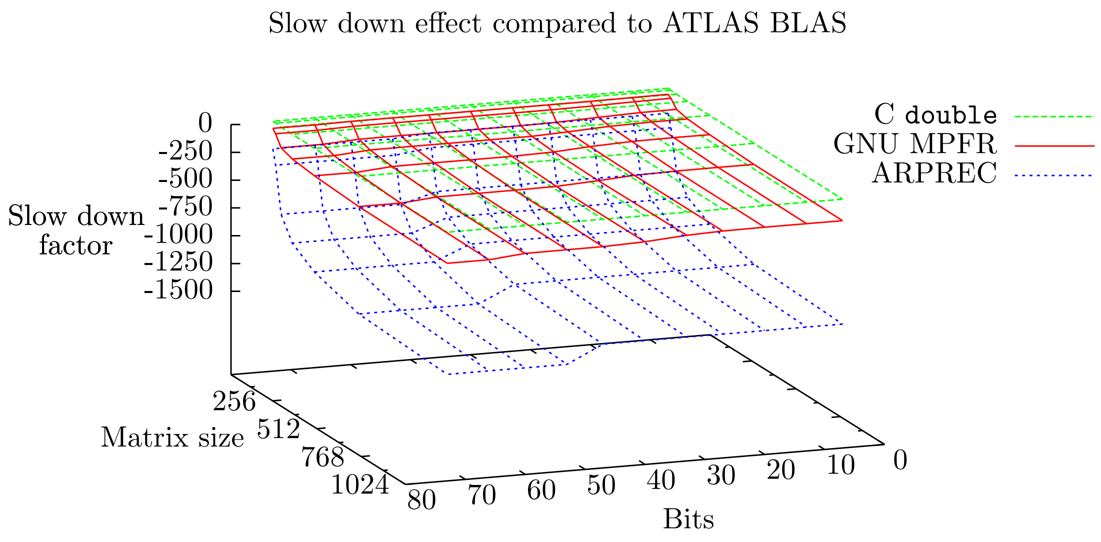

Figure 2.2: Slow down effect of a matrix multiplication using the built-in C variable type `double` and arbitrary precision packages GNU MPFR and ARPREC compared to DGEMM from the ATLAS BLAS library

- Arbitrary Precision: ARPREC 2.2.3, MPFR 2.4.1

In Figure 2.2 the performance described by the slow down effect compared to the ATLAS BLAS implementation can be seen. The different precisions are shown on the x-axis, the matrix dimension is shown on the y-axis. The z-axis shows the slow down factor, where zero refers to the DGEMM function. As can be seen, the performance decrease introduced by the usage of the arbitrary precision packages is very high, the MPFR package being 293 times slower than DGEMM and ARPREC 1291 times slower. Compared to the C double precision implementation MPFR is 21 times slower. ARPREC is 4.4 times slower than MPFR. All slow down factors relative to the different implementations for the maximum matrix size 1024 are shown in Table 1.

BLAS	1.000		
Double	13.776	1.000	
MPFR	293.099	21.275	1.000
ARPREC	1291.649	93.758	4.407

Table 2.1: Average slow down effect

In Figure 2.3 the MPFR implementation is shown alone in order to see another interesting effect. The performance decreases with the number of bits stored in the mantissa which is displayed by a step wise decrease. This indicates the correct treatment of the size of the mantissa with the exact number of bits specified by the user. It proves the statement that MPFR only uses the *specified* precision and does not increase the precision to the next higher word size without excluding the excess number of bits by filling them up with zeros. In Figure 2.2 ARPREC (the blue plane under all other planes) only has one step within the specified precision range, indicating that the precision is increased in steps of multiple machine words and not truncated to the user-defined precision.

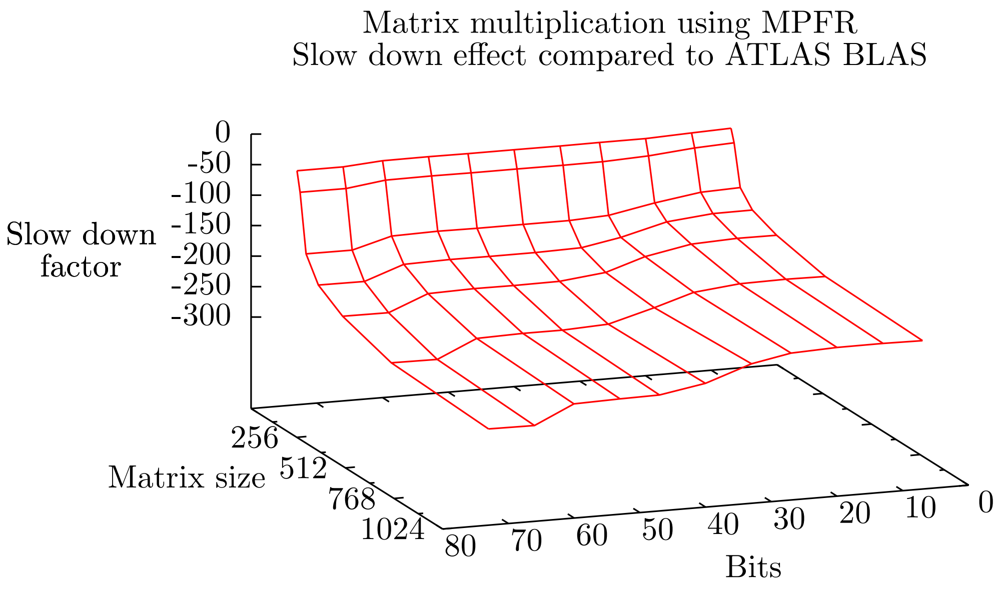

Figure 2.3: Slow down effect of a matrix multiplication using the GNU MPFR library, compared to DGEMM from the ATLAS BLAS library

Chapter 3

Iterative Refinement

In this chapter different iterative refinement methods will be introduced, which will later be compared to the binary cascade iterative refinement.

3.1 Standard Iterative Refinement (SIR)

Iterative refinement is a method used to improve the accuracy of a computed solution by trying to reduce round-off errors. This thesis will focus on linear solvers, which compute the result of linear systems of the form

$$A \cdot x = b \tag{3.1}$$

with $A \in \mathbb{R}^{n \times n}$ and x and $b \in \mathbb{R}^n$, but iterative refinement can be used for many other solvers, including eigensolvers and least squares solvers (for example [12, 18]).

Iterative refinement was first analysed in detail by Wilkinson in [49], but had already been used in desk calculators and computers in the 1940s [24]. The first implementation of iterative refinement was probably written by Wilkinson for the ACE computer built at the National Physical Laboratory. Wilkinson first described the process using a scaled fixed point arithmetic, but the analysis was later expanded by Moler [35] to cover floating-point arithmetic.

The iterative refinement process is described as follows:

1. Solve $A \cdot \widehat{x} = b$ with $\widehat{x}$ being an approximation of x
2. For $i = 0, 1, 2, \ldots$ with $x_0 = \widehat{x}$
   (a) Compute residual $r_i = b - A \cdot x_i$
   (b) Solve $A \cdot \Delta x_i = r_i$
   (c) Update $x_{i+1} = x_i + \Delta x_i$

Firstly an approximate initial solution $\widehat{x}$ is computed using Gaussian elimination with partial pivoting. Subsequently the iterative refinement algorithm tries to increase the accuracy of the solution by computing the residual r_i of the result and using the residual as the right-hand side to solve the linear system for the correction term Δx_i. Finally the correction term is added to the result to correct the solution of the linear system. This process is repeated until the accuracy of the solution is sufficiently improved.

The literature describes many different termination criteria for iterative refinement, which use different measures to check if the convergence is complete. For example, the process can be halted if the norm of the residual $\|r_i\|$ or the norm of the correction term $\|\Delta x_i\|$ is under a described tolerance, which can be the machine epsilon ϵ or a tolerance which also includes the condition number of the input matrix. Other approaches check if the correction term is changing the solution significantly enough. In most cases a limit for the number of iterations is also included to ensure the process does not continue indefinitely or, when applied to extremely ill-conditioned systems, tries to converge to a different solution than the exact solution.

If the initial solution $\widehat{x}$ would already be the exact solution x to the system, then the residual would be zero. However, this is hardly ever the case.

$$A \cdot x_i = b - r_i \tag{3.2}$$

Therefore the correction term Δx_i can be found through

$$A \cdot \Delta x_i = A(x_{i+1} - x_i) = b - (b - r_i) = r_i \tag{3.3}$$

Further it can be shown that the iterative refinement process can produce a better approximation than $\widehat{x}$ [11], since

$$Ax_{i+1} = A\left(x_i + \Delta x_i\right) = Ax_i + A\Delta x_i = (b - r_i) + r_i = b \tag{3.4}$$

The convergence of the iterative process is described in [50] for Gaussian elimination with partial pivoting based on the following factor, where n is the size of the system and α the precision as the number of bits in the mantissa used to store the floating-point number.

$$\Delta = n \cdot 2^{-\alpha} \cdot \left\|A^{-1}\right\|_\infty \tag{3.5}$$

If $\Delta < 2^{-p}$ then the number of correct binary digits of the solution will increase by at least p digits per iteration and the residual will decrease by a factor of 2^p or more. The method will normally not converge if $\Delta > 1/2$. Naturally, this is only a theoretical value as the infinity norm of the inverse of A would be to expensive to compute explicitly, but norm estimators could be used instead [23].

The cost of iterative refinement is very low compared to the matrix factorization, because its complexity is $O(n^2)$ whereas the LU factorization has a complexity of $O(n^3)$. The process also uses the already computed factorization to solve the second system for the correction term in the second step of the iterative process. In [41], it has been shown that for Gaussian elimination a single step of iterative refinement is enough to stabilize the solution sufficiently. One disadvantage of iterative refinement is that it requires more storage than a direct solver without iterative refinement. For the first step of the iterative process, computing the residual, the original matrix A is required in addition to the factorized matrix, which requires double the amount of storage.

The process can be used to recover the accuracy for badly scaled systems to full machine precision [22], but other applications also exist. Iterative refinement has been used to stabilise otherwise unstable solvers, one example being a sparse Gaussian elimination which was performed without pivoting to increase the performance and the result was then stabilised through the use of iterative refinement. [23] The iterative refinement method is extensively used and is also included in software packages like LAPACK [1], where the refinement process is used by the expert drivers for solving linear equations.

The standard iterative refinement method performs all computations using the same floating-point precision, but other iterative refinement methods use different precisions for some steps of the process. To distinguish between the different precisions, the following terminology will be used: *target precision* and *working precision*. The target precision α is the precision to achieve at the end of the computation. The working precision β is the precision used for certain steps during the computation of the solution and is usually higher or lower than the target precision. All steps in the standard iterative refinement use the same precision as the target and working precision. In the next sections, iterative refinement algorithms which use working precisions different than the intended target precision will be introduced.

3.2 Extra Precise Iterative Refinement (EPIR)

The authors in [13] have expanded the standard iterative refinement algorithm to use a higher working precision than the target precision to compute the residual for the iterative improvement. This idea had already been proposed by the original author of iterative refinement, J. H. Wilkinson, in [32] and also by Moler in the same paper where he analysed iterative refinement for floating-point arithmetic [35]. *Extra Precise Iterative Refinement* also includes an error bound for the result and a componentwise error bound, which are both computed at a low additional cost.

There are some differences compared to the standard iterative refinement. Before computing the matrix factorization or a solution of the system, the matrix is equilibrated to try to avoid over- and underflows and to improve ill-conditioned systems which resulted from ill-scaling.

$$A_s = R \cdot A \cdot C,\, b_s = R \cdot b \tag{3.6}$$

R and C are diagonal matrices containing the scaling factors, A and b are the input data of the system which is being solved and A_s and b_s are the scaled system data which will be used by the iterative refinement method. In order not to introduce any additional round-off errors through the equilibration of the matrix, the scaling factors are computed as powers of 2, the same as the standard IEEE floating-point radix.

At the beginning of the extra precise iterative refinement, all computations are computed in the same precision, the target precision α. The higher working precision β is triggered during the iterative refinement and the triggering depends on the convergence rate and the decrease of the error estimate. The higher working precision is chosen to be twice the target precision and is used to store the solution vector to compute the critical stages of iterative refinement, the residual and the update of the solution by the correction factor.

The process terminates when one of the following conditions is fulfilled:

1. The error estimate is not decreasing
2. The correction term does not increase the accuracy of x significantly
3. A predefined maximum number of iterations has been reached

If the error estimate ceases to decrease the process is not immediately terminated unless the solution has already converged. Instead, the first time this termination criteria is encountered, the precision of the solution vector is increased to double the target precision and the process continues until one of the above mentioned termination criteria is met or the solution has converged.

In addition to the solution vector x, which approximates the exact solution of the linear system, the extra precise iterative refinement returns the normwise and componentwise error bounds, which can be approximated as follows:

$$\begin{aligned} \text{Normwise error bound} &\approx \left\| x^{(i)} - x \right\| / \left\| x \right\| \\ \text{Componentwise error bound} &\approx \max_k \left| x_k^{(i)} - x_k \right| / \left| x_k \right| \end{aligned} \tag{3.7}$$

The authors have shown, that the error bounds produce good estimates for the true error, but using ill-conditioned systems the error bounds can underestimate the true error. This is partly compensated through the increase of the working precision during the iterative refinement.

3.3 Mixed Precision Iterative Refinement (MPIR)

In [7], the authors published an iterative refinement algorithm which takes advantage of the availability of single and double precision. The *Mixed Precision Iterative Refinement* takes a different approach compared to the extra precise iterative refinement and focusses on increasing the performance of the linear system solver. It computes the computationally expensive operations, the matrix decomposition and solving the linear systems, in a lower working precision and only performs the critical steps, computing the residual and updating the solution, in a higher target precision, while still achieving the target precision accuracy. Using standard IEEE precisions, the higher target precision α is usually double precision and single precision is used as the lower working precision β. The algorithm for mixed precision iterative refinement is as follows:

1. Solve $A \cdot \widehat{x} = b$ **with variables in precision** β
2. For $i = 0, 1, 2, \ldots$ with $x_0 = \widehat{x}$
    (a) Compute residual $r_i = b - A \cdot x_i$ **with variables in precision** α
    (b) Solve $A \cdot \Delta x_i = r_i$ **with variables in precision** β
    (c) Update $x_{i+1} = x_i + \Delta x_i$ **with variables in precision** α

As stated in [7], mixed precision iterative refinement using single and double precision achieves at least the same and often higher accuracy than a double precision direct solver. Some very ill-conditioned systems may never converge and others may need a high number of iterations until they converge to the correct solution. The number of iterations required for convergence directly relates to the condition number of the input matrix.

Mixed precision iterative refinement requires less storage compared to standard iterative refinement, because the matrix factorization is stored in the lower working precision β. The storage requirements would be 1.5 times more than the storage requirements of a direct solver, but less compared to the standard iterative refinement which uses twice the amount of storage of a direct solver.

Using the lower working precision has many benefits. Modern processors support vector instruction sets, which for example in the case of the SSE2 instruction set enables the processor to compute two double precision operations in one clock cycle. When single precision is being used, the processor can perform four operations in one cycle due to SSE2 instructions, which significantly increases the performance. Single precision data also uses less storage, which results in a lower number of cache misses. Furthermore, moving single precision data through the memory is faster due to the lower storage requirements.

The mixed precision iterative refinement computes the entire solution of the system with increased performance as long as the single and double precision performance is significantly different on the used hardware. On different hardware platforms, for example GPU or Cell processors, there is a much greater difference between the performance of single and double precision computations than on commodity processors. On general purpose GPUs single precision can be more than 8 times faster than double precision [3] and the IBM Cell BE processor can compute single precision roughly 14 times faster than double precision [27].

In [27], mixed precision iterative refinement was implemented for the Cell processor using Cholesky factorization and compared to a direct solver in single precision. One of the results of the performance measurements can be seen in Figure 3.1 for the Cell BE processor used

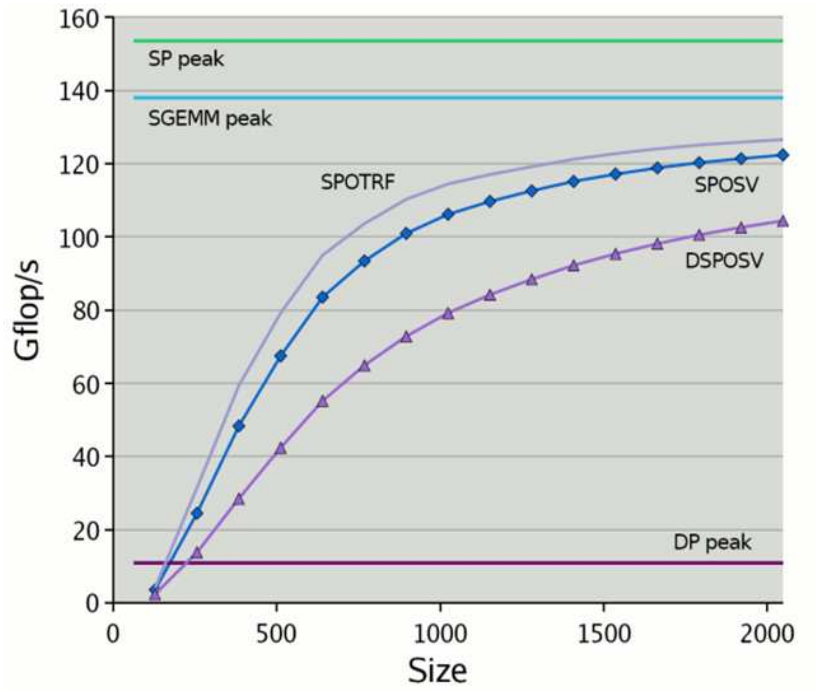

Figure 3.1: Comparison of the performance of the direct solver and the mixed precision iterative refinement using the Cell BE processor on the Sony PlayStation 3 [27].

in the Sony PlayStation 3. The dimension of the linear system is shown on the x-axis and the achieved performance in Gflop/s is plotted on the y-axis.

On the Cell BE processor, the single precision peak performance is 153.6 Gflop/s and the double precision peak performance is only 10.8 Gflop/s. The single precision direct solver, labelled *SPOSV* in the graph, achieves 122 Gflop/s for the maximum system size tested in these experiments ($n = 2048$), but the result is at best only accurate to single precision. By using the mixed precision solver, labelled *DSPOSV* in the graph, the performance for $n = 2048$ is 104 Gflop/s, but the solution of the system is now in double precision accuracy. The use of the mixed precision implementation produces a performance overhead of about 15% compared to the direct solver, but it achieves a solution with double precision accuracy 10 times faster than the peak performance of double precision computations on the Cell processor.

This implementation is a prime example of the great performance benefits of mixed precision iterative refinement using lower precisions to compute the computationally expensive tasks while still achieving the same or better accuracy than a direct solver in the higher target precision, especially on hardware platforms where the performance difference between single and double precision is very high.

3.4 Model Estimating Number of Iterations

The condition number of a matrix provides a means to estimate the accuracy of the solution to a linear system. This property can also be used to estimate the number of iterations required by standard iterative refinement to achieve a given target precision α, as described in [16]. The following is based on the explanations in [40], where the author uses the number of iterations required by iterative refinement to roughly estimate the condition number of the linear system $Ax = b$.

The logarithm to base b of the condition number κ of the matrix A returns an estimate of the number of base-b digits that are lost while solving the linear system, as described in [29]. Let s denote the number of correct base-b digits obtained by solving the linear system, then the accuracy of the solution can be increased by s digits in each iteration. In order to reach the target precision α to base b, the required number of iterations i is therefore defined as

$$i \approx \frac{\alpha_b}{s} \tag{3.8}$$

gaining s digits of accuracy in each iteration. This leads to the following estimate for the condition number κ based on the number of iterations i required by the iterative refinement:

$$\kappa \approx b^{\alpha_b - s} = b^{\alpha_b - \alpha_b / i} \tag{3.9}$$

By using this estimate, it is therefore possible to estimate the number of iterations of the iterative refinement based on the knowledge of the condition number. This results in the following model:

$$i = \frac{\alpha_b}{\alpha_b - log_b(\kappa)} \tag{3.10}$$

This model can be expanded to cover arbitrary precision iterative refinement by setting the precision in the numerator to the target precision and the precision in the denominator to the working precision.

$$i = \frac{\alpha_b}{\beta_b - log_b(\kappa)} \tag{3.11}$$

Chapter 4

BCIR - Binary Cascade Iterative Refinement

4.1 The Algorithm

Binary **C**ascade **I**terative **R**efinement (BCIR) was defined by Andrzej Kiełbasiński in [26]. The main difference between BCIR and other iterative refinement algorithms is the choice of the working precision. In standard iterative refinement the target precision equals the working precision. The extra precise iterative refinement increases the working precision to twice the target precision depending on the progress made by the iterative refinement. The mixed precision iterative refinement chooses a working precision under the target precision.

BCIR takes a different approach in choosing the working precision and does not limit it to a single working precision for the entire process. The algorithm improves the result recursively and chooses a different working precision for each recursion level, making the decision which precision to use based on properties of the input data, more precisely on the dimension and condition number of the input matrix A and taking into account the target precision. This enables the algorithm to dynamically choose the best working precision to achieve an accurate result for the given system and to compensate for ill-conditioned input data.

The algorithm is defined recursively as seen in Equations (4.1)-(4.3).

$$\begin{cases} P(A, \beta_0) \\ x := S_p(b) \end{cases} \tag{4.1}$$

$$d := S_j(f) \textit{ is equivalent to } \begin{cases} z := S_{j-1}(f); \\ u := Az - f; \textit{ in } fl(\beta_j) \\ v := S_{j-1}(u); \\ d := z - v; \textit{ in } fl(\beta_j) \end{cases} \tag{4.2}$$

$$S_0 := \textit{Solves triangular system in } fl(\beta_0) \tag{4.3}$$

First the matrix decomposition P is performed on A in the lowest working precision β_0 chosen by the algorithm. The first call to the solver S_p is executed using the right hand side vector b. p is the number of recursion levels determined by the algorithm.

Each call to the solver S_i performs the computations which correspond to the steps used in any iterative refinement method at the working precision of the current recursion level. The first instruction in Equation (4.2) calls the solver of the next level to solve a system with f as the right hand side. The second instruction computes the result of the previous calculation z and uses the precision β_j of the current level. The correction term v is then computed by again calling the solver of the next lower level, this time using the residual stored in u as the right hand side. These two calls to the solver at the next lower level are the reason why the algorithm is named *binary cascade* iterative refinement. Finally the solution z is updated using the correction term v. d is returned to the previous call of the solver.

Calling the solver S_i to solve the system will cause the algorithm to cascade to the lowest level of the process with the lowest working precision β_0. This means, that any system is always solved at the lowest precision β_0 and all other levels only compute the residual u and update the solution to d. For the LU decomposition, the solver applies the forward and back substitutions at the lowest working precision using the decomposition factors computed before entering the iterative process.

As already mentioned, BCIR uses different working precisions throughout the iterative refinement process. These are chosen adaptively based on the input arguments and are computed before entering the iterative process using the target precision, the dimension n of the system and the condition number of the input matrix A. These properties have to be transformed into the different working precisions which should lead to an accurate solution. This is achieved by computing the four parameters described and analysed in the following sections.

4.2 The Parameters

For the adaptive precisions used during the binary cascading process, some parameters have to be computed before entering the iterative refinement. These parameters will determine the precisions for each level of recursion of the algorithm and take into account the desired target precision and the properties of the input data in order to choose the working precisions to try to compensate for ill conditioned input data.

The first parameter, c, is computed using the dimension n of the $n \times n$ matrix A and two different condition numbers of the matrix.

$$c = \log_2 (\max [K_n \kappa, n(n+6)B/2]) \tag{4.4}$$

Both condition numbers are multiplied by different factors which introduce the matrix dimension n into the equation. The factor K_n should be of the same order of magnitude as the rounding-error accumulation in Gaussian elimination, which is normally of $O(n^2)$. Therefore, K_n was chosen to be n^2.

The first condition number is the standard condition number $\kappa = cond(A) = \|A\| \, \|A^-1\|$. The author does not specify which norm should be used to compute κ, therefore the euclidean norm was chosen. In [26], the author refers to the second condition number B as the Bauer condition number and defines this condition number as

$$B = \sup_{|H| \leq |A|} \left\| A^{-1} H \right\| \tag{4.5}$$

There is neither an explanation what H stands for nor a definition of the norms used in this equation. The author further cites a paper [4] from F. L. Bauer, where the Bauer condition number should originally have been defined, but analysing the specified source did not provide any information on the definition given in Equation (4.5) or a definition for H. In fact, in [4] the condition number is defined as

$$cond(A) = lub(A) lub(A^{-1}) \tag{4.6}$$

which corresponds to the standard condition number κ. The least upper bound lub is used throughout [4] as the maximum norm of the matrices, but can also correspond to other matrix norms, for example to the euclidean norm. The last definition would imply equality between κ and B assuming the same matrix norm is used. The only difference between the condition numbers would be their preceding factor, which under the assumption of $K_n = n^2$ would almost always lead to the first term using the standard condition number κ being chosen as the maximum. Only for matrices with a size $n \leq 6$, the second term would take precedence.

In Kiełbasiński's report [25], which predates the original BCIR paper [26] and also included an earlier form of the binary cascade iterative refinement, the same definition as in Equation (4.5) is given alongside two additional Bauerian condition numbers, C_B and $C_B^{'}$, again citing the same source by F. L. Bauer as before, which again did not include these definitions.

$$\begin{aligned} C_B &= \sup\nolimits_H \frac{\left\|A^{-1} H x^*\right\|}{\|x^*\|}, \; with \;\; |H| \leq |A| \\ C_B^{'} &= \sup\nolimits_m \frac{\left\|A^{-1} m\right\|}{\|x^*\|}, \; with \;\; |m| \leq |b| \end{aligned} \tag{4.7}$$

Additionally, the relation between the different condition numbers is shown in [25, p. 6, (2.2.4)], which result in all Bauer condition numbers always being smaller than or equal to the standard condition number κ.

$$\kappa \geq B \geq C_B \geq C_B^{'} \geq 1 \tag{4.8}$$

This relation again favours the first term of the maximum function in Equation (4.4), leading to the second term only being considered when $n \leq 6$ (with $K_n = n^2$).

The Bauer condition number would only have an impact on the computation, if it were larger than the standard condition number. This can only occur when the definition in

Equation (4.6) would be used and the norm used would be higher than the one chosen for κ. This would lead to a decrease of the number of iterations and to an increase of the working precisions at the different levels of recursion. However, the higher working precisions would also induce a lower performance and would reduce the possibility of any performance benefits of BCIR compared to standard iterative refinement methods. Conversely, the higher working precisions could increase the accuracy of the result, which would be beneficial when solving extremely ill-conditioned linear systems.

All these findings regarding the Bauer condition number show, that the factor hardly has any influence on the choice of the precisions for the iterations of the BCIR algorithm. Due to the ambiguous definitions of the Bauer condition number and the relation provided in Equation (4.8) the definition of c can therefore be reduced to Equation (4.9).

$$c = \log_2 (K_n \kappa), \; \textit{with} \; K_n = n^2 \tag{4.9}$$

Equation (4.10) is the second parameter required to compute the different working precisions of the binary cascade iterative refinement.

$$\tau = 2 - \log_2 \epsilon \tag{4.10}$$

ϵ is the machine epsilon, which is defined in [23, p. 37] by the following equation:

$$\epsilon = b^{1-t} \tag{4.11}$$

b is the base of the floating-point representation and t defines the precision. This definition of the machine epsilon describes the spacing of floating-point numbers by computing the distance between 1.0 and the next larger representable floating-point number. In standard double precision, the number of bits used to store the mantissa would be 53 and have a machine epsilon of $\epsilon \approx 2.220446... \cdot 10^{-16}$. Based on the definition in (4.11) and using $b = 2$ as the base of the floating-point representation, Equation (4.10) can be rewritten to:

$$\tau = 2 - (1 - t) = 1 + t \tag{4.12}$$

The next parameter, p, defines the number of recursion levels used in the binary cascading process. First it chooses the minimum between the relation of the previously defined parameters τ and c and $n/2$. Then the maximum of 0 and the result of the logarithm to base 2 is assigned to p.

$$p = \max(0, \lfloor \log_2(\min(\tau/c, n/2)) \rfloor) \tag{4.13}$$

In order to be able to compare the algorithm to the other iterative refinement methods, an equivalent number of iterations can easily be calculated by

$$\textit{Iterations} = 2^p \tag{4.14}$$

Finally, the precision at each recursion level can be computed using the following equation and the previously defined parameters:

$$\beta_j = c + \tau \cdot 2^{j-p},\ j = 0(1)p \tag{4.15}$$

β_j describes the precision at the j^{th} recursion level and j runs from 0 to the maximum recursion level p.

4.2.1 Analysis of the Levels of Recursion p

The goal of this section is to analyse the behaviour of parameter p (Equation (4.13)) and determine how much influence each factor of the minimum function has on the resulting working precisions for the BCIR process. For the analysis, the target precision α will be chosen to correspond to the IEEE standard double precision with a mantissa width of 53 bit.

$$\tau = 2 + \log_2 \epsilon = 1 + \alpha = 54 \tag{4.16}$$

$$\begin{gathered} \tau/c = n/2 \\ 54/\log_2 (n^2 \cdot \kappa) = n/2 \\ n \cdot \log_2 (n^2 \cdot \kappa) = 108 \\ n \approx 14.1326...\ \mathit{for}\ \kappa = 1 \end{gathered} \tag{4.17}$$

The analysis in (4.17) shows, that for a perfectly conditioned input matrix with $\kappa = 1$ the size n has to be lower than 15 in order for the term $n/2$ to be chosen over τ/c. For higher condition numbers, the factor $n/2$ looses its influence even more and τ/c becomes the dominant factor. For all $n \geq 15$, τ/c is always the dominant factor.

Even though the Bauer condition number has been deemed unnecessary in the definition of c, the same analysis can be performed for the second term of the original definition of parameter c.

$$\begin{gathered} \tau/c = n/2 \\ 54/\log_2 (n \cdot (n+6)/2 \cdot B) = n/2 \\ n \cdot \log_2 (n \cdot (n+6)/2 \cdot B) = 108 \\ n \approx 14.8475...\ \mathit{for}\ B = 1 \end{gathered} \tag{4.18}$$

The results are very close to the previous results in (4.17) and the upper limit for the influence of $n/2$ is again $n < 15$. τ/c becomes the dominant factor for all $n \geq 15$.

The number of recursive levels therefore depends primarily on the value of c. As c approaches infinity, the number of levels reaches 0.

$$\lim_{c \to \infty} p = \frac{\tau}{c} = 0 \tag{4.19}$$

This results in the system only being solved and improved at a single level using only one working precision. The maximum number of recursion levels for $n \geq 15$ and target precision

$\alpha = 53$ is 2.

$$p = \left\lfloor \log_2 \frac{54}{\log_2(15^2 \cdot 1)} \right\rfloor = 2 \ with\ n = 15, \kappa = 1 \tag{4.20}$$

For smaller linear systems with dimension $n < 15$, the equation becomes independent of the condition number and has a maximum of 6 recursive levels for the binary cascade iterative refinement.

$$p = \left\lfloor \log_2 \frac{54}{\log_2(1/2)} \right\rfloor = 6 \ with\ n = 1 \tag{4.21}$$

4.2.2 Analysis of the Precisions β_j

The next factor which is of interest for analysis is the working precisions at the different recursion levels. As we have seen in Chapter 3, there are different approaches as to how to choose the working precision for the iterative refinement. The extra precise iterative refinement increases the precision for the computation of the critical sections. The mixed precision iterative refinement runs the critical sections using the target precision and lets the computationally expensive tasks be computed in the lower working precision. BCIR chooses the working precisions based on the input arguments and tries to find the best precision to accurately solve the linear system. The following figures show the initial and final working precisions used by BCIR for different condition numbers κ between 1 and 10^{16} and different system sizes n from $n = 10$ to $n = 1000$.

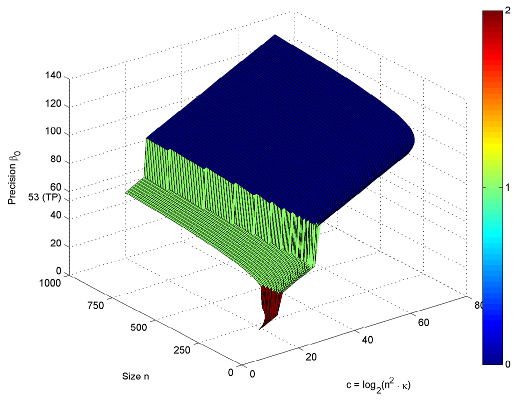

Figure 4.1: Initial working precisions β_0 coloured by the number of recursion levels p

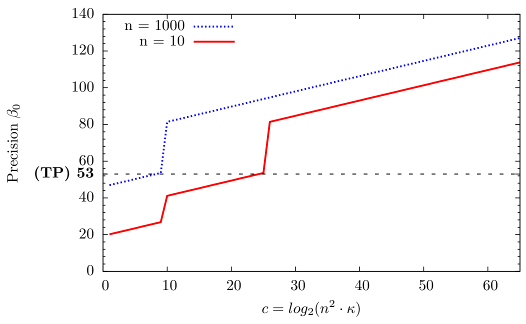

Figure 4.2: Initial working precisions β_0 for the two matrix sizes $n = 10$ and $n = 1000$

Figure 4.1 shows the initial working precisions for the lowest level of the algorithm. The x-axis shows the parameter c, which depends on the dimension n and the condition number κ. The size of the input system is plotted on the y-axis and the initial working precision β_0 is shown on the z-axis. The surface is coloured by the number of recursive levels p. As shown in the last section, the maximum number of recursive levels is $p = 2$. The lowest initial working precision β_0 for well conditioned input data is 21 for $n = 10$ and 47 for $n = 1000$. $\beta_0 = 47$ is already very close to the target precision $\alpha = 53$ and for slightly larger and slightly worse conditioned systems, the working precision rapidly increases and is always larger than the target precision. The highest initial and, due to $p = 0$, also the final working precision in the range computed for these plots is $\beta_0 = 128$ for ill-conditioned matrices with $\kappa = 10^{16}$ and $n = 1000$. The number of recursive levels rapidly decreases to its limit (Equation (4.19)) and results in the binary cascade iterative refinement only performing one iteration at a high working precision for most input data.

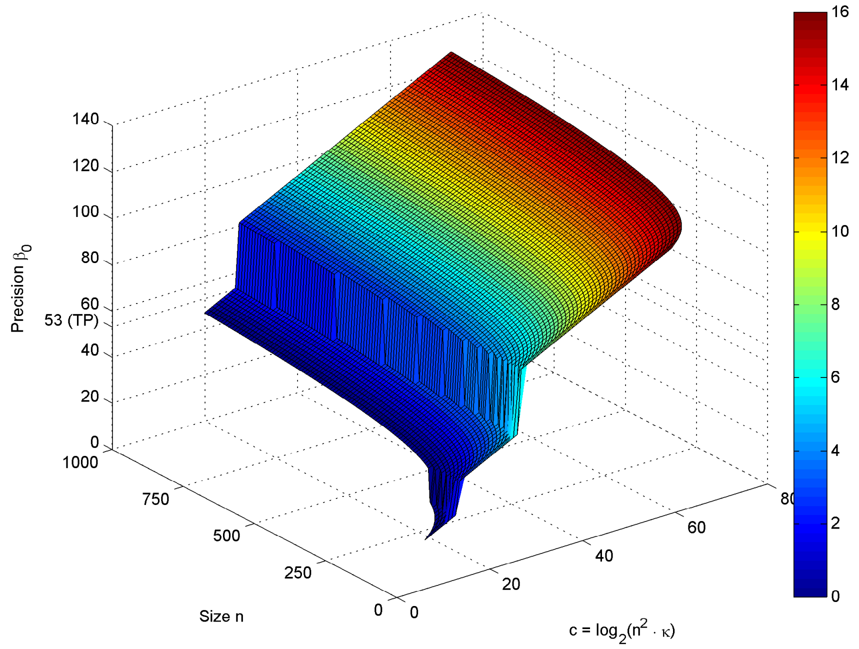

Figure 4.3: Initial working precisions β_0 coloured by the condition number κ

In Figure 4.3 the same values as in Figure 4.1 are shown but in this case, the working precisions are coloured by the exponent of the condition number κ. This shows the influence of κ on the number of recursive levels and the initial working precision. Systems with a condition number $\kappa > 10^4$ already only perform the computations on the lowest level and the working precision rises with the condition number. For larger systems, the boundary of the influence of the condition number decreases and the size of the input data gains influence on determining the number of recursive levels and the working precisions.

Figure 4.4 shows the final working precision β_p, which in case $p = 0$ is the same as the initial working precision β_0. The precision β_p never falls lower than the target precision. The lowest final working precision for well conditioned input matrices is 61, which is higher than the target precision $\alpha = 53$, and increases with the dimension of the system. For the maximum value of n displayed here, $n = 1000$, the working precision $\beta_p = 74$. For large and ill conditioned matrices the maximum precision is the same as before ($\beta_p = 128$) because $p = 0$.

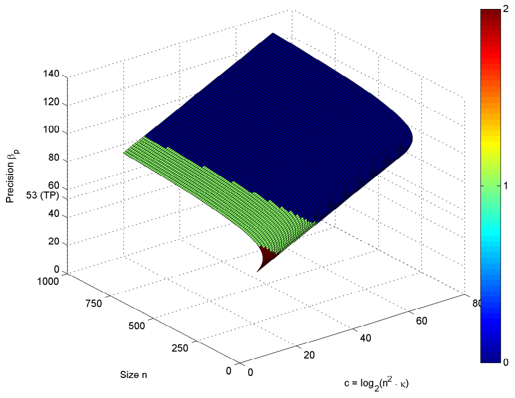

Figure 4.4: Final working precisions β_p coloured by the number of recursion levels p

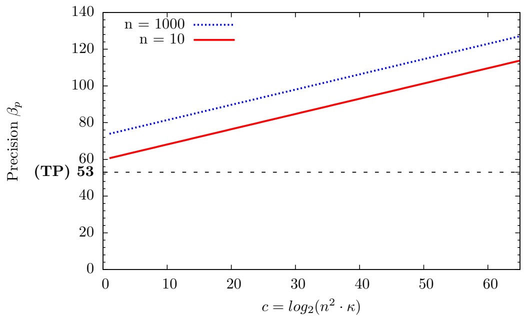

Figure 4.5: Final working precisions β_p for the two matrix sizes $n = 10$ and $n = 1000$

4.3 Refined Version of BCIR

In Section 4.2, it has been shown that the Bauer condition number has hardly any influence on the binary cascade iterative refinement process. Therefore, it was even more intriguing to read a statement made by A. Kiełbasiński in the section "Finale Remarks" of his paper, which describes a "*slightly more refined version of BCIR*" [26] by replacing the previous definition of parameter c (Equation (4.4)) with

$$c = \log_2 B + 5 \tag{4.22}$$

The author further states that this redefinition demonstrates the dependency of c on the Bauer condition number B, which now becomes the dominant term.

This would contradict all findings on the Bauer condition number described in Section 4.2. Due to the ambiguous and incomplete definitions of the Bauer condition number mentioned earlier, it is not possible to verify this statement by computing and comparing the different condition numbers. The definitions in [25] and [26] regarding the Bauer condition number have shown, that it is either equal to or smaller than the standard condition number κ. Furthermore, the analysis of parameter p and the precisions β_j have confirmed these initial findings (see Subsection 4.2.1 and Subsection 4.2.2 respectively).

With the new definition in Equation (4.22), the matrix dimension n is no longer taken into account when calculating the necessary working precisions to achieve an accurate result. n would only occur as the second factor $n/2$ in Equation (4.13). Using the same assumptions as in Subsection 4.2.1, $B = 1$ and the target precision $\alpha = 53$, the factor $n/2$ would affect the working precisions for all $n \leq 41$, but for larger and not perfectly conditioned matrices

τ/c would again become the dominant factor and n would no longer be considered.

$$\begin{aligned} 54/\log_2(1+5) &= n/2 \\ n &= 41.78010318 \end{aligned} \tag{4.23}$$

Assuming that $B = \kappa$ (see relationship in Equation (4.8)), Equation (4.22) would cause the number of recursion levels to rise and the working precisions to decrease. One advantage of this definition is therefore that the performance could be increased due to the lower working precision. In Figure 4.6, the different precisions are shown for the initial precision β_0. The parameter c is shown on the x-axis for κ between 1 and 10^{16}. The y-axis plots the matrix size n for matrices from $n = 10$ to $n = 1000$. The z-axis shows the initial working precisions β_0. The surface plot is coloured by the level of iterations needed for the different condition numbers and starting at the precision β_0.

Compared to Figure 4.1 from the previous analysis of parameter c, the first difference that can be seen is the maximum value of p, which has increased to 4 recursion levels, compared to 2 in the previous analysis. The second important observation is that the values of β_0 have decreased significantly for well conditioned matrices and are still lower for ill-conditioned matrices. The lowest initial working precision is 6 for well conditioned matrices for any size n. In Figure 4.1, the lowest β_0 for well conditioned input data was 21 for $n = 10$ and 47 for $n = 1000$.

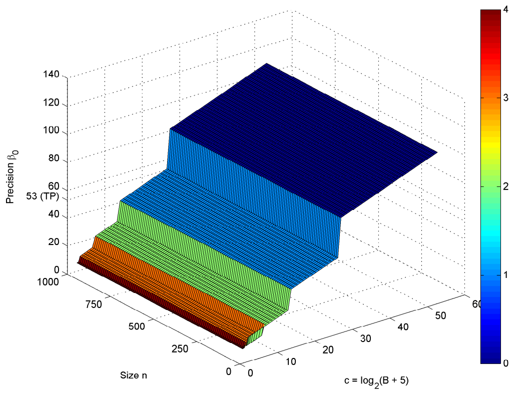

Figure 4.6: Initial working precisions β_0 coloured by the number of recursion levels p using the definition of c in Equation (4.22)

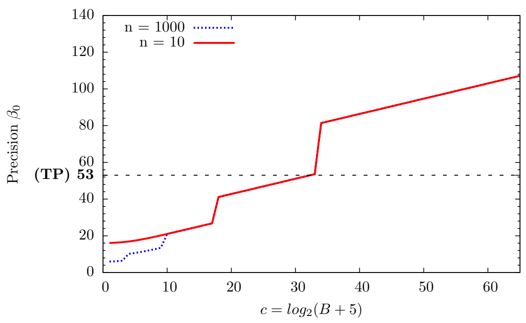

Figure 4.7: Initial working precisions β_0 for the two matrix sizes $n = 10$ and $n = 1000$ using the definition of c in Equation (4.22)

The second figure, Figure 4.8, shows the highest working precisions used for the different condition numbers. It is important to notice that the lowest used precision for well conditioned input matrices is now 57, which is just slighty higher than the target precision $\alpha = 53$. Due to the lack of the matrix dimension in the new definition of c, the precision is independent from n. In Figure 4.4, the lowest precision was 61, but it increased with n ($n = 1000 \rightarrow \beta_p = 74$). The maximum precision shown in Figure 4.4 was $\beta_p = 128$ for large and ill conditioned matrices. In this case, the maximum working precision is 108.

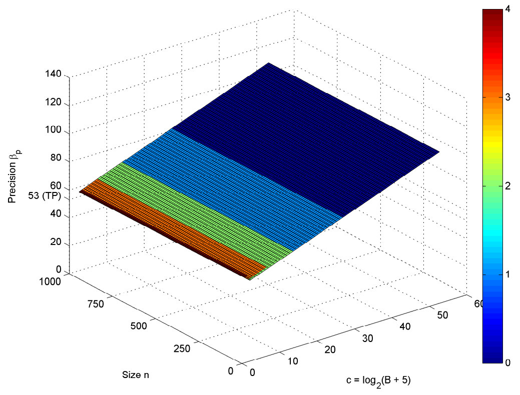

Figure 4.8: Final working precisions β_p coloured by the number of recursion levels p using the definition of c in Equation (4.22)

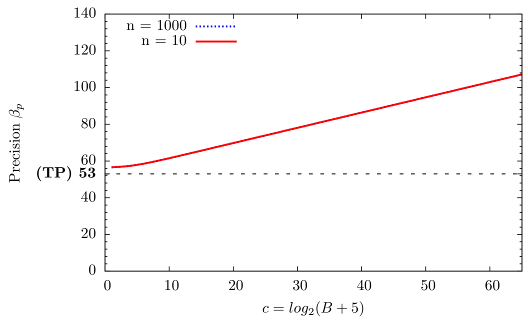

Figure 4.9: Final working precisions β_p for the two matrix sizes $n = 10$ and $n = 1000$ using the definition of c in Equation (4.22)

The alleged refined version of BCIR will be analysed in the experiments (Section 6.6) using $B = \kappa$ as the Bauer condition number and comparing the results to the standard binary cascade iterative refinement.

4.4 Accessing Matrix A and Vector b at Different Precisions

The binary cascade iterative refinement uses different working precisions throughout the refinement process. Each level of recursion requires the input values A and b to be in the corresponding working precision. Even though the amount of recursive levels is limited, as shown in previous sections, this issue still needs to be addressed.

In the mixed precision iterative refinement the input matrix A only has to be available in the working and target precision and in order to improve performance is therefore additionally allocated in the working precision, assuming that the data was provided in the target precision. In the BCIR algorithm this would increase the amount of storage by n^2 for each level of recursion for the conversion to the working precision. Analysing the algorithm reveals that at each level, except for the lowest level using the precision β_0, each element of the matrix A is required and read only once, to be specific when computing the residual. Therefore it is better for the performance of the algorithm and for memory reduction not to preallocate the matrix for all the working precisions, but to convert them on-the-fly to the precision required by the corresponding recursive level. Similarly to the mixed precision iterative refinement, the matrix then only exists twice: once in the input precision and in the lowest working precision β_0, which is accessed 2^p times to solve a linear system.

If the input precision of A and b is lower than the working precision in which the residual is being computed, then the elements of the matrix do not need to be converted at all, because their precision would not increase and instead the excess bits would only be filled up with zeros. Therefore the matrix elements are only converted to lower precisions compared to their input precision. As previously shown, the working precisions are almost always higher than the target precision and therefore the conversion of the input arguments to lower precisions does not occur very often.

4.5 Conclusion

The binary cascade iterative refinement aims to compensate for ill-conditioned system with the use of higher working precisions. This approach resembles more the extra precise iterative refinement than the mixed precision iterative refinement, which reduces the working precision in favour of performance benefits. Furthermore, similarly to the standard iterative refinement, the number of iterations is very often limited to one iteration.

The working precisions of the binary cascade iterative refinement hardly ever fall beneath the target precision α. The binary cascade iterative refinement requires arbitrary precision to compute the result of the linear system with the precisions determined by the algorithm. It therefore relies on either simulating the arbitrary precision in software or being implemented on hardware which supports the use of multiple arbitrary precisions, for example FPGAs.

Chapter 5

Implementation

This chapter will focus on the programs, which were developed for this master thesis, and describe the requirements and the program which generates the input data, which consists of the matrices with the specified condition number and the right hand sides of the equation. The implemented programs for BCIR, EPIR, SIR and MPIR, which both use the same program by setting the working precision β to the target precision α for the standard iterative refinement, and a direct LU solver (program `noir`) to demonstrate the improvement of the accuracy of the solution achieved by iterative refinement will be explained. Important implementation details will also be discussed in this chapter.

All iterative refinement algorithms were implemented using arbitrary precision even though some of the algorithms could have been implemented using the standard single and double precision. The extra precise iterative refinement could have been realised with extended or quadruple precision supported by many hardware vendors and also part of the IEEE 754 standard. GNU MPFR provided a portable, hardware independent implementation of arbitrary precision data types which could avoid any discrepancies which could occur due to differences in the hardware implementation of the IEEE standard data types. Using the same data type implementation for all algorithms facilitates the comparison of the numerical properties of the iterative refinement methods .

A complete list of all project files can be found in the Appendix Section A.2.

5.1 Requirements

The programs developed for this thesis rely on the following libraries:

- The arbitrary precision is achieved by using the GNU MPFR library [15].
- The singular value decomposition included in LAPACK [1] is used to calculate the condition number κ of the input matrix A. Other LAPACK routines are used outside the iterative refinement processes and also for equilibrating the matrix as required by the extra precise iterative refinement.

- The results are written to an XML file using the libxml2 [44] library.
- The performance is measured using PAPI [6].
- The data files are compressed using `tar` and `gzip`.

5.2 Generate Matrix

`generateMatrix` creates a matrix A and vector b with the dimension n and a condition number κ and writes the data to a file using the format described in Section 5.3.

```
Usage: generateMatrix -n <DIM> [-c <CondNr>] -f <FNPrefix> [-d <DIR>] [--date]
    [--rmin <RANDMIN>] [--rmax <RANDMAX>] [--gzip] [--verify] [--type
    <MATRIXTYPE>]
       -n          Matrix dimension n
       -f          Prefix for the data file
       -d          Data directory
   (*) -c          Condition number for matrix A
   (*) --date      Inserts the current date in the file name
   (*) --rmin      Minimum value for the random values
   (*) --rmax      Maximum value for the random values
   (*) --gzip      Compress data file with gzip
   (*) --verify    Verify the generated data and condition number
   (*) --type      Specify the type of matrix to be generated
                    * random  - generates a random matrix (default)
                    * Hilbert - generates a Hilbert matrix

Parameters marked with (*) are optional
```

Apart from the required parameter `-n` for the size of the matrix and vector, a prefix for the output file has to be specified using `-f <PREFIX>`. All other parameters are optional. An output directory can be specified by the parameter `-d <DIR>`. The current date and time can be included in the file name by specifying the parameter `--date`.

The program `generateMatrix` currently supports two different kinds of matrices. The first type, which is also the default type if the parameter `--type` is not explicitly provided, is a random matrix which can be modified to have the condition number specified by parameter `-c`. For details, how the matrix is created with the specified condition number, please refer to Section 6.1. The minimum and maximum of the range of the random values of the matrix can be defined by providing `--rmin` and `--rmax`, respectively. By default the random values will be in the interval $[-1.0; 1.0]$. The second supported matrix type is a Hilbert matrix, which can be specified by `--type=hilbert`.

The data can be compressed to reduce the amount of storage by specifying `--gzip`. `generateMatrix` also provides the ability to verify the created data. The data is loaded from the input file and the condition number is calculated and compared to the condition number provided by `-c`.

The following command shows an example of the usage of the program:

```
./generateMatrix -n 2500 -c 1e3 -f ir -d data --date --gzip
```

This will produce a random matrix with size $n = 2500$ and a condition number $\kappa = 10^3$ and store the data in the compressed file

```
data/ir_20110701_123030000000.random.2500.cn_e3.data.tar.gz
```

using the size and the exponent of the condition number in the file name.

5.3 File Format

The matrix A and the right-hand side vector b are stored together in a plain text file using the following format.

```
TIMESTAMP
DIM1 DIM2
DATA Matrix



DATA Vector
```

The first line of the data file contains an identification number which is the timestamp at the time of creation. The second line defines the dimensions of the matrix A, m and n. The following block of data is the matrix, each row of the matrix being separated by a newline character. The last line of the file holds the data for the vector, which has the size of `DIM1` and is separated from the matrix data by three newline characters.

The functions required to read and write the data files are provided in *ir_io.h*.

5.4 Block LU decomposition

A blocked version of the LU factorization was implemented as described in [19, 14] as an efficient LU factorization. The block LU factorization exploits the benefits of operating on the data that already exists in the local caches and reduces the number of calls to fetch the data from the computationally more expensive entities higher up in the memory hierarchy.

The $n \times n$ matrix $A \in \mathbb{R}$ is partitioned as follows

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \tag{5.1}$$

The blocked LU factorization first computes the factors L_{11} and U_{11} for the upper left block A_{11}. In the next step the triangular system $L_{11}U_{12} = A_{12}$ is solved for the multiple right-hand

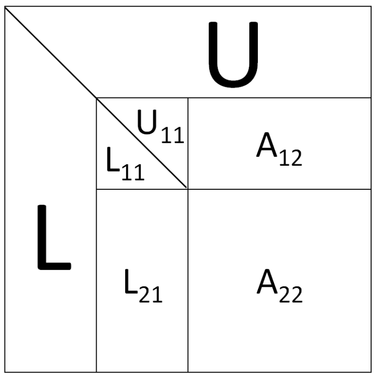

Figure 5.1: Block LU factorization [30]

sides of U_{12} and the triangular system $L_{21}U_{11} = A_{21}$ for the multiple right-hand sides of L_{21}. Finally the last block, the lower right matrix A_{22}, has to be updated by

$$A_{22}^{'} = A_{22} - L_{21}U_{12} \tag{5.2}$$

using a matrix multiplication. This update is called the *Schur complement* of A_{11}. The steps for the blocked algorithm are also shown in Figure 5.1.

The same arithmetic operations are performed for the block LU factorization as for other LU decompositions. They are simply executed in a different order. The majority of the computations are performed in the matrix multiplication. The performance of the blocked algorithm therefore largely depends on an efficient implementation of the matrix multiplication.

5.5 Implemented Programs

The call signature of the programs which prepare and execute the different iterative refinement methods are all very similar. In fact they only differ in the parameter to define the working precision. All other available parameters are identical and will be explained in the following paragraphs.

```
Usage: ir -a <PREC> [--no-bits] (-f <FileName> OR -n <DIM> [--type
    <MatrixType>] [--no-save]) [-o <Directory>]
      -a          target precision alpha
  (*) --no-bits   interpret precisions as number of decimal digits , not number
      of bits
  (*) -o          output directory
  (~) -f          file containing input data
  (~) -n          size of the matrix
  (*) --type      specify the type of matrix to be generated
                      * random  - generates a random matrix (default)
                      * Hilbert - generates a Hilbert matrix
  (*) --no-save  automatically generated input data will not be saved (does not
      affect result files)

Parameters marked with (*) are optional.
One of the parameters marked with (~) has to be provided.
```

The parameter `-a` is used to specify the target precision α for the iterative refinement. If the method supports the choice of a working precisions, an additional parameter `-b` will be available. The target and working precisions can either be interpreted as the number of bits stored in the mantissa of the floating-point number, which is the default behaviour, or as the number of decimal digits by providing the optional parameter `--no-bits` .

The applications can load already existing data for the matrix A and vector b or generate new data for the computation. The user can either provide the size of the $n \times n$ matrix A and the vector b with the parameter `-n` or the file name of a file containing a matrix and vector using the parameter `-f`. The file must conform to the format described in Section 5.3, which can be generated by the program `generateMatrix` as described in Section 5.2. If no input file is specified, the matrix A and the right-hand side vector b are generated automatically using the size n provided by `-n` and stored in the folder `data_autogen`, which will be created if it does not exist. It is also possible to prevent the storage of the automatically generated data by providing the parameter `--no-save`. When the matrix is generated in the program, the type of the matrix can also be specified (`--type <MatrixType>`) using the same types as seen by `generateMatrix`. The parameter `-o` enables the user to specify an output directory for the result files described in Subsection 5.5.1 and will be created by the program if it does not exist.

The following command is an example for the usage of the programs:

```
./ir -a 53 -f data/ir_20110701_123030000000.random.2500.cn_e3.data -o results
```

The target precision α is specified as the number of bits stored in the mantissa and the program uses a previously generated data file to compute the solution of the linear system using the iterative refinement, storing the information about the process in the folder `results`.

5.5.1 File Structure of the Results

The results of the iterative refinement methods are stored in two different files.

The first file is a text file containing the results in the form of a table, which can easily be plotted by programs like gnuplot [1]. The first line is the header of the columns, the second line contains the results. Listings 5.1, 5.3 and 5.5 show examples of this output file. As described in Section 5.5, the target and working precisions can be specified as either the number of bits stored in the mantissa or the number of decimal digits. Regardless of the choice of the representation, the precisions are printed as the number of bits in column 2 and 3 of the output file. In addition, the precision is also printed as the number of decimal digits

[1]http://www.gnuplot.info/

in the column `AlphaDP` and `BetaDP`. The relative residual norm is stored in the last column `RelNormRes`.

The other file is an XML file containing more detailed data about the convergence of the iterative refinement. Examples of this XML file can be seen in Listings 5.2 and 5.4. The XML structure is divided into two sections: the input and output data. The input data includes the name of the file used by the program, the corresponding file id as described in Section 5.3, the size n, the condition number κ and the target and working precisions as the number of bits in the mantissa and the number of decimal digits.

The first node of the output data stores the number of iterations performed by the iterative refinement. The relative residual norm of the solution after the iterative refinement is stored in the node `<rel_norm_res>`. The performance measurements are divided into different sections which depend on the different algorithms. The XML file contains all the defined sections for the execution time (XML node `<times>`), the number of floating-point operations (XML node `<fp_ops>`) and the number of total cycles (XML node `<tot_cyc>`).

The data is stored using a file name that contains the size n of the matrix, the α precision and depending on the iterative refinement also the β precision, the exponent of the condition number and the date and time of the experiment. The XML file uses the extension `*.results`, the text file the extension `*.log`. An example of the file names is shown here:

```
bcir_n002500_a53_condnr3_DT20110701_123030000000.log
bcir_n002500_a53_condnr3_DT20110701_123030000000.results
apir_n002500_a53_b24_condnr3_DT20110701_123030000000.log
apir_n002500_a53_b24_condnr3_DT20110701_123030000000.results
```

5.5.2 Binary Cascade Iterative Refinement (BCIR)

```
Usage: bcir -a <PREC> [--no-bits] (-f <FileName> OR -n <DIM> [--type
    <MatrixType>] [--no-save]) [-o <Directory>]
      -a          target precision alpha
       ...
```

The parameter `-a` is used to specify the target precision for the binary cascade iterative refinement. The parameter for the working precision β is missing, because the working precisions are computed automatically by the binary cascade algorithm for each recursive level and depend on the target precision, the condition number $\kappa(A)$ and the size of the input system n.

Accessing Matrix A and Vector b at Different Precisions

As described in Section 4.4, if the input precision of the matrix is higher than the current working precision, then the elements are converted on-the-fly as they are accessed. They are copied into a temporary variable with the required working precision and the need for converting and storing the entire matrix in memory is eliminated. When computing the residual at higher precisions than the input precision it is not necessary to copy the values to a corresponding working precision variable, because the excess bits would only be filled up with zeros. Therefore the matrix elements can be used directly without the loss of accuracy and the result will still be stored in and rounded to the higher working precision of the destination variable.

Output from `bcir`

The result files are extended by some BCIR specific parameters. The last columns of the table based results file, Listing 5.1, contain the properties computed by the algorithm to determine the different working precisions and a comma separated list of all working precisions used by the binary cascade iterative refinement.

The performance measurements for BCIR are divided into the following sections:

- **bcirprepcondnr** - computing properties required by the binary cascade algorithm and calculating the condition number using the singular value decomposition
- **plu** - the LU decomposition using partial pivoting in the lowest working precision determined by the algorithm
- **bcir** - the entire binary cascade iterative refinement, which is further divided into the following sections
  - ◇ **convert** - converting the data to the required precisions and allocating necessary temporary storage
  - ◇ **iterref** - the iterative refinement
  - ◇ **norm** - computing the norm of the residual $\|r\|$ and the relative residual

The XML file expands the `<output>` node by BCIR specific values. The node `<bcir>` contains the values of the computed parameters c, τ and p required to determine the working precisions. These precisions are then explicitly stored in the node `<precisions>`. The XML file includes the value of $\|r\|$ (XML node `<norm_res>`) for the last step of the iterative refinement.

Listings 5.1 and 5.2 show examples of the different output formats for the same output data.

```
Dimension Alpha Beta Iterations BCIRPrepTime LUFactTime IterRefTime CondNr CondNrLowHigh Accuracy
    ResidualNorm dxNorm BCIRPrepFLOPs BCIRPrepTotCyc LUFactFLOPs LUFactTotCyc IterRefFLOPs
    IterRefTotCyc AlphaDP AlphaDPE BetaDP BetaDPE FileID RelNormRes BCIR_c BCIR_tau BCIR_p
    BCIR_precs
2500 53 0 1 193.1206109999999967 610.7720430000000533 8.6616289999999996 1.00000000e+03 0
    0.00000000e+00 5.42793596e-13 0.00000000e+00 43203280678 439336573020 9996 1400271547370 0
    19825485762 16 15.9546 0 0.0000 1303169780948568 2.09712029e-17 3.25412090e+01 54 0 87
```

Listing 5.1: Example for an output file created by `bcir`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bcir_results date="2011-07-01" time="12:30:30">
   <input>
      <filename>ir_20110701_123030947053.random.2500.cn_e3.data</filename>
      <dataid>1303169780948568</dataid>
      <size>2500</size>
      <alpha bits="53" dps="16" dps_exact="15.9546"/>
      <beta bits="0" dps="0" dps_exact="0.0000"/>
      <condnr>1.000000e+03</condnr>
   </input>
   <output>
      <iterations>1</iterations>
      <bcir>
         <c>3.25412090e+01</c>
         <tau>54</tau>
         <p>0</p>
      </bcir>
      <precisions>
         <precision step="0">87</precision>
      </precisions>
      <times>
         <timesection section="bcirprepcondnr">193.12061100</timesection>
         <timesection section="plu">610.77204300</timesection>
         <timesection section="bcir">11.75304900</timesection>
         <timesection section="convert">1.14143000</timesection>
         <timesection section="iterref">8.66162900</timesection>
         <timesection section="norm">1.05945400</timesection>
      </times>
      <fp_ops>
         <fp_ops_section section="bcirprepcondnr">43203280678</fp_ops_section>
         ...
      </fp_ops>
      <tot_cyc>
         <tot_cyc_section section="bcirprepcondnr">439336573020</tot_cyc_section>
         ...
      </tot_cyc>
      <accuracy/>
      <norm_res>
         <norm_res_step step="1">5.4279359632387033e-13</norm_res_step>
      </norm_res>
      <rel_norm_res>2.0971202862984781e-17</rel_norm_res>
   </output>
</bcir_results>
```

Listing 5.2: Example for an XML file created by `bcir`

5.5.3 Standard IR and MPIR (APIR)

`apir` runs the standard and mixed precision iterative refinement methods and accepts the parameters `-a` and `-b` to specify the target and working precision, respectively. The working precision β can either be 53 bits to match the standard iterative refinement or 24 bits for mixed precision iterative refinement.

```
Usage: apir -a <PREC> -b <PREC> [--no-bits] (-f <FileName> OR -n <DIM> [--type
    <MatrixType>] [--no-save]) [-o <Directory>]
      -a          target precision alpha
      -b          working precision beta
        ...
```

Output from apir

The table based results file stores the working precision in the third column as the number of bits in the mantissa and in one of the last columns as the number of decimal digits

(Listing 5.3).

The XML file contains detailed information about the convergence of the iterative refinement (Listing 5.4). The section containing the input data includes β as the number of bits in the mantissa and the number of decimal digits. The first node of the output data stores the number of iterations required for the convergence and the setting of the maximum number of iterations. It also includes the values of $\|\Delta x\|$ (XML node `<norm_dx>`) and $\|r\|$ (XML node `<norm_res>`) for each step of the iterative refinement.

The performance measurements for SIR and MPIR are divided into the following sections:

- **condnr** - calculating the condition number using the singular value decomposition
- **plu** - the LU decomposition using partial pivoting in β precision
- **apir** - the entire iterative refinement, which is further divided into the following sections
    - ◇ **convert** - converting the data to the required precisions
    - ◇ **initsol** - calculating the initial solution
    - ◇ **iterref** - the iterative refinement

Listings 5.3 and 5.4 show examples of the different output formats for the same output data.

```
Dimension Alpha Beta Iterations LUFactTime InitSolTime IterRefTime CondNr CondNrLowHigh Accuracy
    ResidualNorm dxNorm LUFactFLOPs LUFactTotCyc InitSolFLOPs InitSolTotCyc IterRefFLOPs
    IterRefTotCyc AlphaDP AlphaDPE BetaDP BetaDPE FileID RelNormRes
2500 53 24 3 583.0899170000000140 2.5646000000000000 8.6438039999999994 1.00000000e+03 0
    0.00000000e+00 7.87519820e-13 6.72706868e-11 9996 1336616742278 0 5873981362 83 19797637958 16
    15.9546 8 7.2247 1303169780948568 3.01435415e-17
```

Listing 5.3: Example for an output file created by `apir`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<apir_results date="2011-07-01" time="12:30:30">
  <input>
    <filename>ir_20110701_123030947053.random.2500.cn_e3.data</filename>
    <dataid>1303169780948568</dataid>
    <size>2500</size>
    <alpha bits="53" dps="16" dps_exact="15.9546"/>
    <beta bits="24" dps="8" dps_exact="7.2247"/>
    <condnr>1.000000e+03</condnr>
  </input>
  <output>
    <iterations maxiter="30">3</iterations>
    <times>
      <timesection section="condnr">32.03412200</timesection>
      <timesection section="plu">583.08991700</timesection>
      <timesection section="apir">13.57390400</timesection>
      <timesection section="convert">1.06414500</timesection>
      <timesection section="initsol">2.56460000</timesection>
      <timesection section="iterref">8.64380400</timesection>
    </times>
    <fp_ops>
      <fp_ops_section section="condnr">43203280657</fp_ops_section>
      ...
    </fp_ops>
    <tot_cyc>
      <tot_cyc_section section="condnr">73284036761</tot_cyc_section>
      ...
    </tot_cyc>
    <accuracy/>
    <norm_res>
```

```
         <norm_res_step step="0">1.7685828077005059e-02</norm_res_step>
         <norm_res_step step="1">3.8548877412407704e-06</norm_res_step>
          ...
      </norm_res>
      <rel_norm_res>3.0143541517324874e-17</rel_norm_res>
      <norm_dx>
         <norm_dx_step step="0">3.4291620774316110e-03</norm_dx_step>
         <norm_dx_step step="1">5.2978759174469239e-07</norm_dx_step>
          ...
      </norm_dx>
   </output>
</apir_results>
```

Listing 5.4: Example for an XML file created by `apir`

5.5.4 Extra Precise Iterative Refinement (EPIR)

```
Usage: epir -a <PREC> [-b <PREC>] [--no-bits] (-f <FileName> OR -n <DIM>
    [--type <MatrixType>] [--no-save]) [-o <Directory>]
      -a          target precision alpha
 (*) -b          working precision beta (default: 2*alpha)
       ...
```

`epir` provides the ability to define the β precision explicitly and override the default value of 2α, but it only has to be greater than α to be accepted. This would further allow to test the precision used when extending the target precision and offers a fine grained choice in the working precision. This analysis would exceed the scope of this thesis and will therefore not be analysed, but could be an interesting topic for future experiments. An algorithm that does not necessarily require double the target precision but can solve the same problem with the same accuracy by only slightly increasing the precision and at the same time provide reliable error bounds would probably require the working precision to be chosen according to the properties of the input data.

In [13], the authors suggest to perform the row and column scaling of the equilibration of the input matrix A with multiples of the power of 2 in order not to introduce any additional round-off errors. LAPACK provides the function `DGEEQUB` [28] to meet this requirement.

The extra precise iterative refinement uses the infinity norm to compute the condition number, which influences the triggering of the extended working precision. Based on the definition of the condition number

$$\kappa_\infty = \|A\|_\infty \left\|A^{-1}\right\|_\infty \tag{5.3}$$

the inverse of the scaled matrix A_s was required to compute κ_∞. The inverse can be computed using an LU decomposition and n forward and back substitutions. Due to the LU decomposition already existing for the iterative refinement, only the forward and back substitutions have to be computed. In praxis, a condition number estimator [21, 39, 5] would be used instead of explicitly computing the inverse of the matrix, which would be beneficial for the performance. Furthermore, an estimate of the power of magnitude of the condition number provides sufficient information to decide whether the higher working precision should be used or if the iterative refinement should continue using the target precision.

Output from `epir`

The output of the extra precise iterative refinement is almost identical to the standard and mixed precision iterative refinement. It has been expanded by the number of iterations which were performed in the target precision and the extended working precision.

The only difference in the XML file compared to Listing 5.4 is the `<iterations>` node, which includes the number of iterations performed in α and β precision.

The performance measurements for EPIR are divided into the following sections:

- **condnr** - calculating the condition number using the singular value decomposition
- **epir** - the entire extra precise iterative refinement, which is further divided into the following sections
  - **equilibrate** - equilibrating the input matrix and applying the scaling factors to A and b
  - **kappainf** - computing the condition number κ using the infinity norm
  - **plu** - the LU decomposition using partial pivoting in α precision
  - **initsol** - calculating the initial solution
  - **iterref** - the iterative refinement

```
Dimension Alpha Beta Iterations LUFactTime InitSolTime IterRefTime CondNr CondNrLowHigh Accuracy
    ResidualNorm dxNorm LUFactFLOPs LUFactTotCyc InitSolFLOPs InitSolTotCyc IterRefFLOPs
    IterRefTotCyc AlphaDP AlphaDPE BetaDP BetaDPE FileID RelNormRes IterInAlpha IterInBeta
2500 53 106 4 617.2460290000000214 12.6525449999999999 55.4796710000000033 1.00000000e+03 0
    0.00000000e+00 6.88361891e-12 2.83350419e-13 9996 1414986219429 0 28831631365 340 126341721904
    16 15.9546 32 31.9092 1303169780948568 2.89100157e-17 2 2
```

Listing 5.5: Example for an output file created by `apir`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<epir_results date="2011-07-01" time="12:30:30">
   ...
   <output>
      <iterations maxiter="30" alpha_iter="2" beta_iter="2">4</iterations>
      ...
   </output>
</epir_results>
```

Listing 5.6: Example for an XML file created by `apir`

5.5.5 Direct LU Solver (NoIR)

The direct LU solver only requires one precision, the target precision, and uses no iterative refinement.

```
Usage: noir -a <PREC> [--no-bits] (-f <FileName> OR -n <DIM> [--type
    <MatrixType>] [--no-save]) [-o <Directory>]
      -a            target precision alpha
        ...
```

Chapter 6

Experiments

The aims of the experiments were to compare the different iterative refinement methods based on their numerical behaviour, more precisely the accuracy of the refined solution based on the relative residual

$$r_{rel} = \frac{\|r_{abs}\|_1}{\|A\|_1 \|x\|_1} \tag{6.1}$$

and the rate of convergence based on the number of iterations each method required. It is important to mention, that the number of iterations required by the iterative refinement methods can not be mapped directly to the performance of the algorithms. For example, if a method only requires one iteration to refine the result, but computes the improvement at a very high precision, then the performance for computing this result will normally be low. If a method requires more iterations but can compute these iterations at lower precisions, then it can achieve better or the same performance as one iteration using the higher precision. This point will be described in more detail while examining the results of the experiments.

All algorithms were implemented using the arbitrary precision library GNU MPFR to guarantee the use of the same implementation of the different precisions for all algorithms and not comparing hardware dependent IEEE standard precision implementations to the arbitrary precision library. The performance was not compared based on the experimental data due to the insignificant differences when emulating different mantissa lengths in arbitrary precision when operating in the small range of precisions as in the case of these experiments. To provide information about the performance of the different iterative refinement methods, a performance model was defined taking into account the different precisions and their benefits and losses. The performance model is not part of this chapter, but will be described in Chapter 7.

The experiments were conducted for the following different system dimensions and condition numbers:

- Dimensions: $n = 10, 25, 50, 75, 100, 250, 500, 1000, 1500, 2000, 2500$
- Condition numbers: $\kappa = 10^0 \ldots 10^7$

For each size n and condition number κ, seven square dense random matrices were generated. The results show the average of all the measurements. Based on previously conducted experiments, random matrices with condition numbers higher than $\kappa = 10^7$ did not produce good results with some iterative refinement methods. As already mentioned, the arbitrary precision arithmetic is being emulated using the software library GNU MPFR. Although GNU MPFR is one of the fastest arbitrary precision libraries available, it still slows down the execution of a program considerably. Therefore the dimension of the linear systems was limited to $n = 2500$ in the conducted experiments.

The measurement covered well-conditioned systems as well as ill-conditioned systems. However, in order to also test extremely ill-conditioned systems, Hilbert matrices with the same dimensions as mentioned above were used. Hilbert matrices have entries of the form

$$H(i,j) = \frac{1}{i+j-1} \tag{6.2}$$

and are very ill-conditioned. A small 10×10 Hilbert matrix already has a condition number of $\kappa = 1.6025 \cdot 10^{13}$ and a 1000×1000 Hilbert matrix has $\kappa = 5.0201 \cdot 10^{20}$.

6.1 Generated Data

To generate matrices with a desired condition number, the singular value decomposition is used and the singular values are scaled to the targeted condition number κ_T. The first step is to compute the condition number κ of the randomly generated matrix using the largest and smallest singular value of the diagonal matrix Σ of the singular value decomposition.

$$\kappa = \frac{\sigma_{1,1}}{\sigma_{n,n}} \tag{6.3}$$

If the condition number of the randomly generated matrix is lower than the targeted condition number, then the largest singular value, the first element of the diagonal matrix, is set to be κ_T times the smallest singular value, the last element of the diagonal matrix. Otherwise the diagonal matrix is traversed checking if the following condition has been reached:

$$\frac{\sigma_{i,i}}{\sigma_{n-i,n-i}} \leq \kappa_T \tag{6.4}$$

If the targeted condition number has been found, then all values of the upper part of the diagonal above this position are set to the largest matching value $\sigma_{i,i}$ and all values of the lower diagonal below this position are set to the smallest matching value $\sigma_{n-i,n-i}$. Finally the first element of Σ is set to be κ_T times the smallest singular value. If the condition in Equation (6.3) is not fulfilled, then all singular values are set to 1 and the first singular value is set to the targeted condition number. One special case exists if $\kappa_T = 1$, then all singular values are set to 1.

6.2 Target and Working Precisions

The target precision is represented by α, the working precision by β except in the case of BCIR, where the different target precisions are defined by β_j with $j = 0 \ldots p$, as described in Section 4.2. For all experiments, the target precision will be IEEE double precision with $\alpha = 53$ bits. MPIR will use IEEE single precision $\beta = 24$ bits as the lower working precision.

In the following sections the different algorithms will be addressed by using their abbreviations defined in the previous chapters:

- **LU solver**: This corresponds to a standard LU decomposition with forward and back substitution, but without the use of iterative refinement.
- **SIR**: Standard Iterative Refinement with $\alpha = \beta =$ double precision
- **EPIR**: Extra Precise Iterative Refinement with $\beta = 2\alpha = 106$ bits
- **MPIR**: Mixed Precision Iterative Refinement with $\beta \approx 1/2\alpha =$ single precision
- **BCIR**: Binary Cascade Iterative Refinement

6.3 Termination Criteria

The standard and mixed precision iterative refinement both use the same termination criteria,

$$\|r\|_2 < n\epsilon_\alpha\kappa \text{ or } \|\Delta x\|_2 < n\epsilon_\alpha\kappa \tag{6.5}$$

where n is the system dimension, ϵ_α is the machine epsilon of the target precision and κ is the condition number of the matrix A. Furthermore the iterative process is halted if a maximum number of iterations is reached, which based on heuristic observations was chosen to be 30 iterations.

As described in Section 3.2, the extra precise iterative refinement will terminate if the error estimate is decreasing too slowly or if the correction term does not increase the accuracy of the solution significantly. Again the iterative refinement is also terminated, if the maximum number of iterations is reached. This was chosen to be 30 iterations, the same as for the standard and mixed precision iterative refinement.

The binary cascade iterative refinement behaves differently than the other iterative refinement methods. During the iterative process the algorithm does not check for the convergence of the solution. BCIR computes the number of iterations and the working precision to use in each iteration based on the input data before starting the iterative improvement. This should guarantee that after the precomputed number of iterations the specified target precision will be reached. Therefore the convergence and the accuracy of the result have to be evaluated after the process has terminated.

6.4 Precisions used by BCIR

A brief look at the precisions used by BCIR at the different recursive levels will be provided, before turning the attention to the comparison of the iterative refinement methods.

The following two figures, Figure 6.1 and Figure 6.2, show the different precisions β_j of the binary cascade iterative refinement for different condition numbers, shown on the x-axis. The precision is plotted on the y-axis as the number of bits stored in the mantissa. The dotted black line is a reference for the standard double precision using 53 bits in the mantissa, which is the target precision α. The second y-axis on the right shows the relative residual achieved for each system with the corresponding condition number.

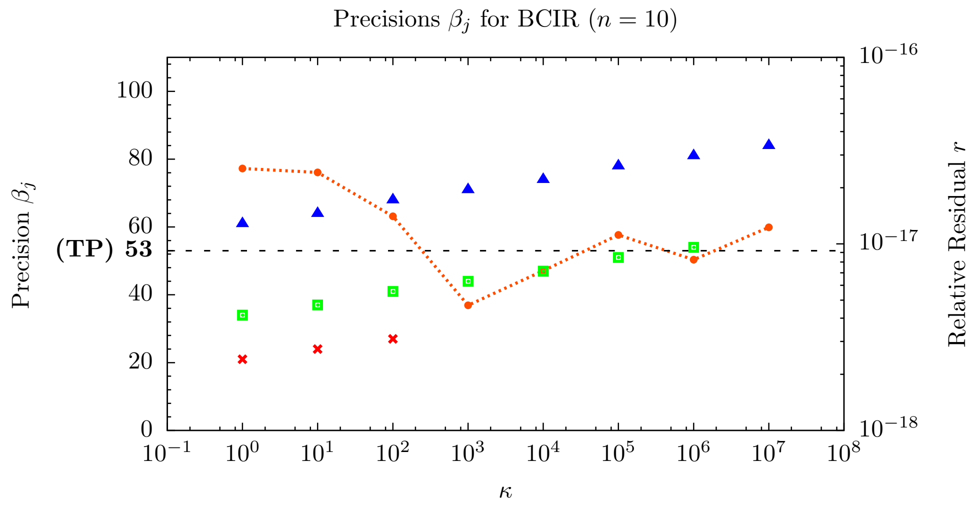

Figure 6.1: Precisions β_j used throughout the iterative process for $n = 10$ plotted on the left y-axis and the relative residual depicted by the orange dotted line on the right y-axis

In Figure 6.1 the precisions β_j are shown for a very small linear system with $n = 10$. For condition numbers $\kappa < 10^3$, the algorithm uses three different recursion levels and therefore three different precisions. For perfectly conditioned systems, the first precision is 21 bits and the second precision uses 34 bits, both significantly lower than the target precision $\alpha = 53$ bits. The last precision used for the perfectly conditioned systems is higher than the target precision and requires 61 bits.

As the condition number rises, the working precisions β_j also increase and the number of iterations diminishes. Between $\kappa = 10^3$ and $\kappa = 10^6$, two precisions are used to improve the result and for $\kappa = 10^6$ all precisions used by BCIR are higher than the target precision, the lowest precision used being just slightly over α with 54 bits. For $\kappa = 10^7$, the iterative method uses only one recursive call and only one precision which is as high as 84 bits.

The last plot only showed the precisions for a very small linear system. In Figure 6.2, the size of the linear system has been increased to $n = 2000$. As seen in the graph, only systems with a condition number $\kappa \leq 10$ use two different precisions for their computations. Otherwise only one precision is used, all of them being significantly higher than the target

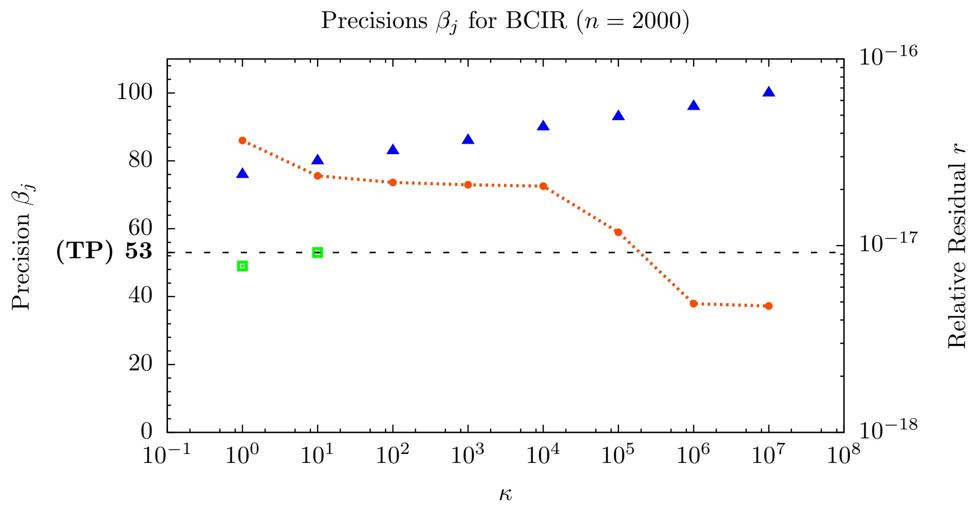

Figure 6.2: Precisions β_j used throughout the iterative process for $n = 2000$ plotted on the left y-axis and the relative residual depicted by the orange dotted line on the right y-axis

precision α with the worst conditioned system of the measurements with $\kappa = 10^7$ using a precision with 100 bits to store the mantissa of the floating-point numbers.

This already shows that the number of iterations is very low for the binary cascade iterative refinement, but that the process always chooses at least one of the working precisions higher than the target precision. This provides a good relative residual for the solution of the linear system, but also incurs a higher computational cost due to the high working precision.

6.5 Comparison of Iterative Refinement Methods

6.5.1 Random Matrices

Number of iterations

The number of iterations provides information about the rate of convergence of the different iterative refinement methods.

Figure 6.3 shows the number of iterations for all iterative refinement methods for a linear system with $n = 100$ for different condition numbers plotted on the x-axis. The number of iterations can not be compared without considering the different working precisions the different methods operate at.

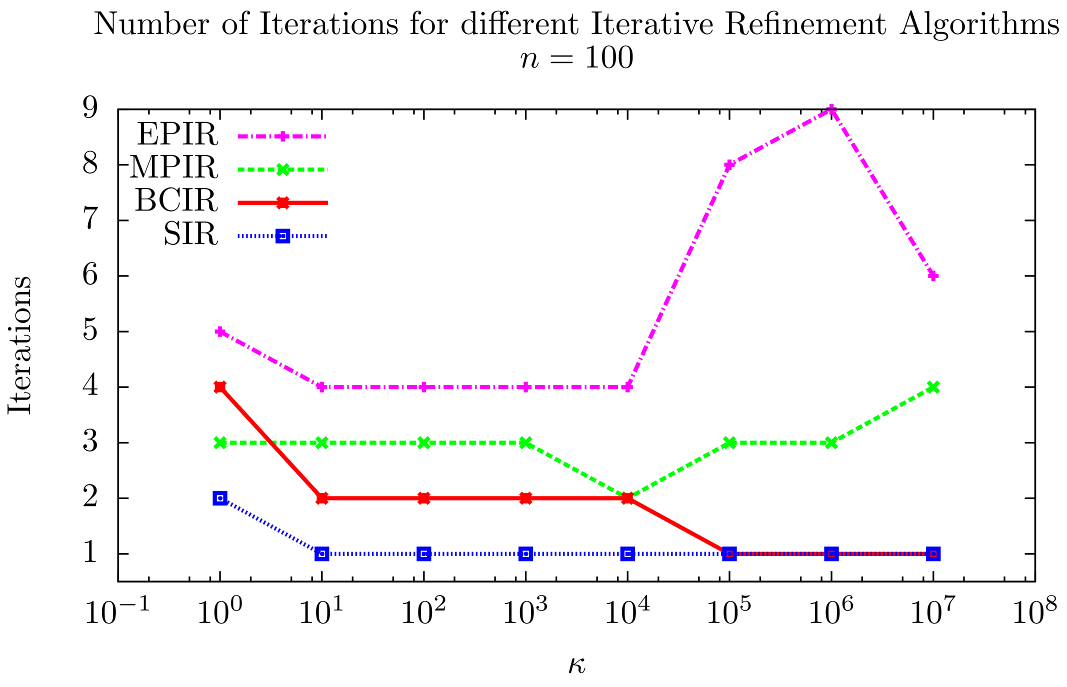

Figure 6.3: Number of iterations for the different iterative refinement methods for $n = 100$

The standard iterative refinement requires the least number of iterations. Except for perfectly conditioned systems, where the process requires two iterations, the algorithm computes the improvement in one iteration using the same precision for all operations. The mixed precision uses more iterations than SIR, but these are performed at the lower working precision, which is single precision. Therefore MPIR achieves a higher performance than SIR while using a higher number of iterations. The extra precise iterative refinement not only requires the highest number of iterations, but also performs some of them at higher working precisions. On average 2 iterations are performed at the target precision α and the other iterations are performed at the extended working precision $\beta = 2\alpha$. For ill-conditioned systems, more iterations are required and these are computed using the extended working precision.

Binary cascade iterative refinement computes the number of iterations to execute before entering the iterative refinement process. It is therefore directly dependent on the condition number of the linear system, which is used to compute p, the number of recursive levels, and corresponds to 2^p iterations. For perfectly conditioned systems, more iterations are executed than for ill-conditioned systems, but for systems with a higher condition number the working precisions used by BCIR are also higher. BCIR does not use the same precision for all iterations and therefore some iterations are computationally cheaper than others. In addition to this, not all iterations perform the same amount of work. The lowest level of the recursion solves two triangular systems in the lowest working precision β_0, whereas the other iterations only compute the residual and update the solution. Therefore it is difficult to directly compare the number of iterations of BCIR with other iterative refinement methods.

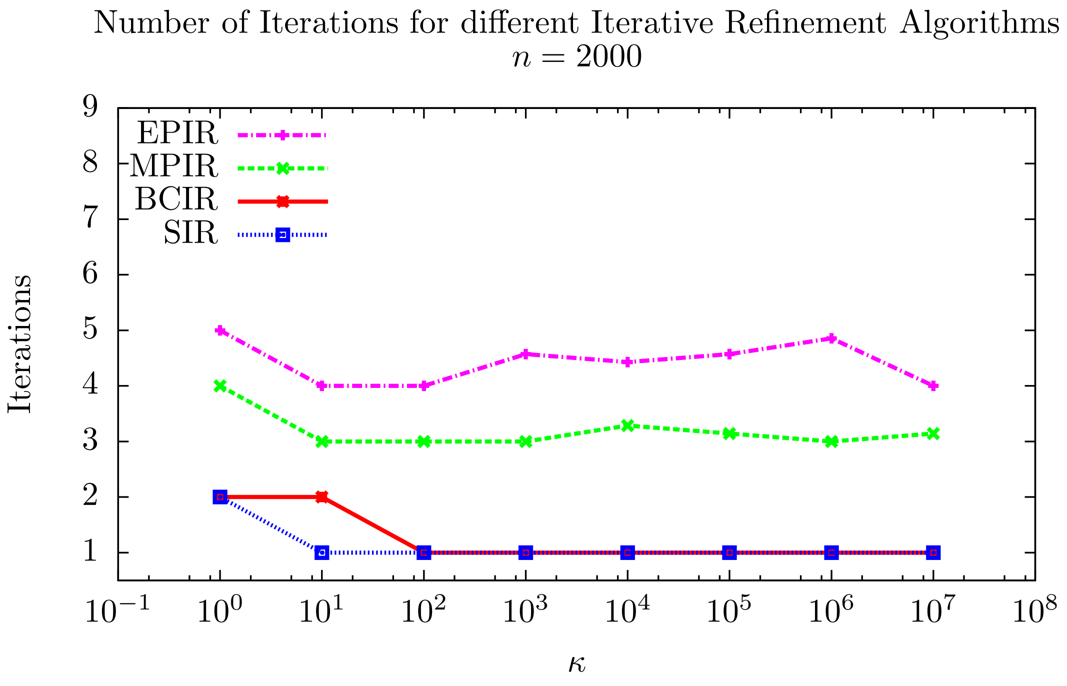

Figure 6.4: Number of iterations for the different iterative refinement methods for $n = 2000$

The second plot, Figure 6.4, shows the number of iterations for different condition numbers for a larger system with $n = 2000$. The number of iterations is more consistent for all different condition numbers. EPIR still requires the most iterations, followed by MPIR which computes the iterations at a lower working precision. The standard iterative refinement still only requires one iteration to improve the result except for the perfectly conditioned linear system. BCIR requires 2 iterations for systems with a condition number of $\kappa = 1$ and $\kappa = 10$, but otherwise also only executes one iteration but at a much higher working precision than the standard iterative refinement.

The last plot, Figure 6.5, shows the number of iterations for different system sizes, plotted on the x-axis, for linear systems with a condition number $\kappa = 10^3$. The standard iterative refinement always uses one iteration to improve the solution and the mixed precision iterative

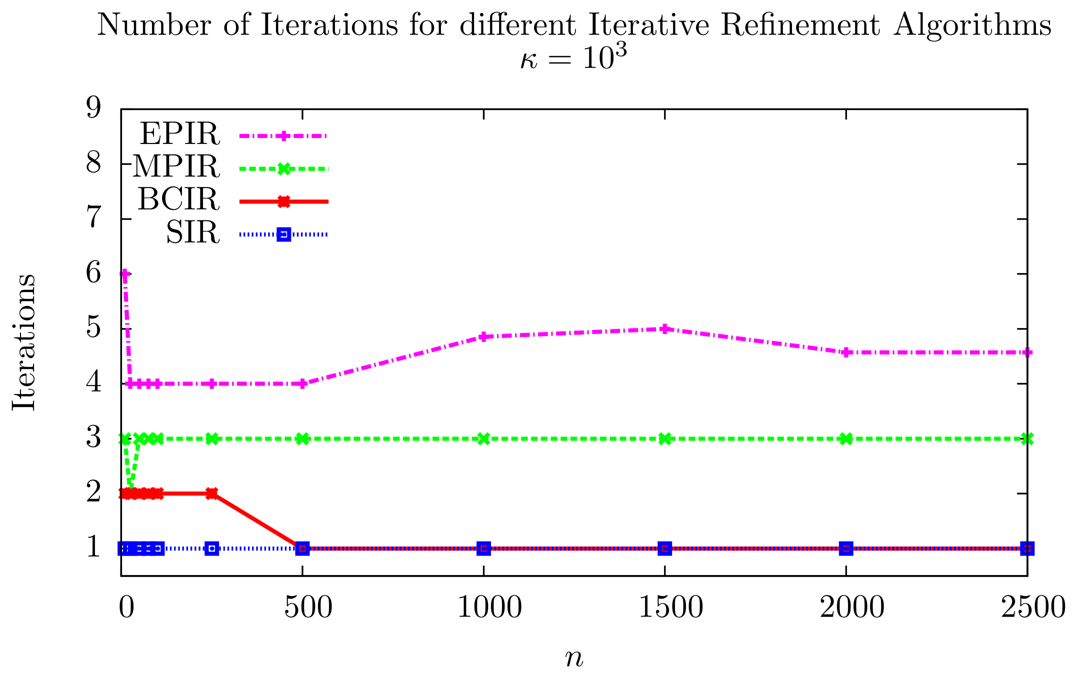

Figure 6.5: Number of iterations for the different iterative refinement methods for $\kappa = 10^3$

refinement is also almost constant and uses 3 iterations at single precision. EPIR requires the most iterations and the number differs depending on the size of the linear system. Binary cascade iterative refinement computes the refinement most of the time in one iteration and only requires more iterations for smaller systems.

Relative Residual

The next numerical aspect to be analysed is the accuracy achieved by the different iterative refinement methods by comparing the relative residual (Equation (6.1)).

In Figure 6.6, the relative residual is plotted for a large linear system with $n = 2000$ for different condition numbers κ shown on the x-axis. The first, the turquoise line is the LU solver without iterative refinement. Through the use of iterative refinement, the relative residual can be improved by almost 2 orders of magnitude. The very ill-conditioned systems do not profit as much from the iterative refinement, but still improve the result by almost 1 order of magnitude. Only the mixed precision iterative refinement fails to find a better result for the worst conditioned system in these experiments and the extra precise iterative refinement is hardly any better than the direct LU solver for systems with $\kappa = 10^7$. In all other cases the iterative refinement methods have significantly improved the result and for systems with a condition number $\kappa \leq 10^4$ they all achieve almost the same results, with the exception of binary cascade iterative refinement. BCIR consistently returns slightly better results than all other methods. Another interesting observation is that the standard iterative refinement performs better with ill conditioned systems than the extra precise iterative refinement, but

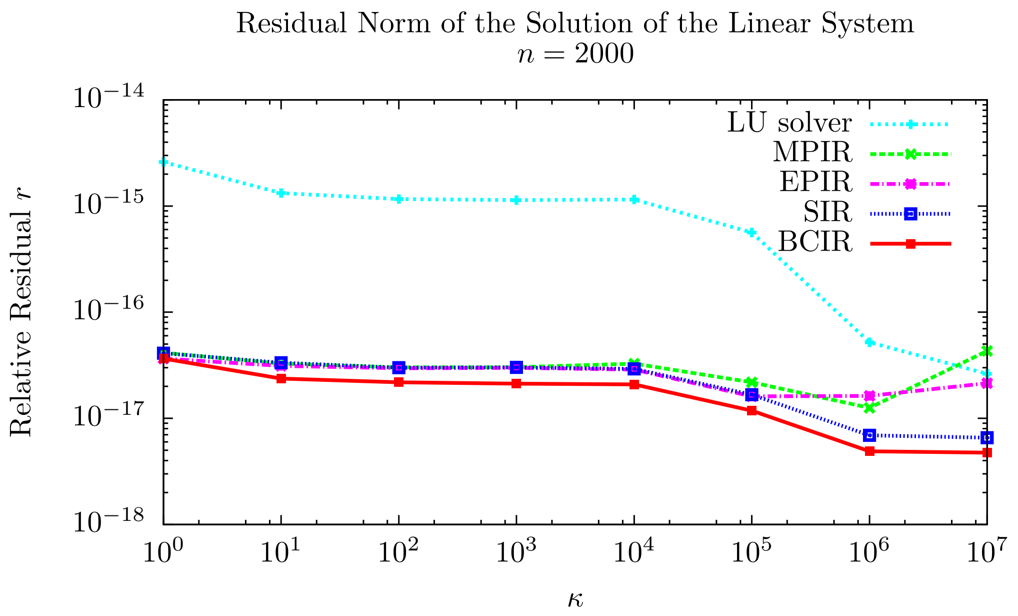

Figure 6.6: Relative residual for different iterative refinement methods for $n = 2000$

only uses one precision for the refinement process whereas EPIR uses higher precisions to improve the result.

Figure 6.7 shows the relative residual for linear systems with a fixed condition number $\kappa = 10^3$ and variable system sizes plotted on the x-axis. The relative residual of the direct LU solver increases with the dimension of the linear system, but the refined solutions remain at the same level of accuracy for all larger systems. All iterative refinement methods achieve again approximately the same improvement for systems larger than $n = 100$ and binary cascade iterative refinement always achieves a slightly better accuracy.

The convergence behaviour of the iterative refinement methods for ill-conditioned linear systems with $\kappa = 10^7$ is displayed in Figure 6.8. Extra precise iterative refinement achieves the worst results for these linear systems and the solution has a lower accuracy than the direct LU solver. Mixed precision iterative refinements can produce worse results for larger systems, but most of the time has similar results as the direct LU solver. Only the standard iterative refinement and the binary cascade iterative refinement have better relative residuals than the direct solver, with BCIR being slightly better than the standard iterative refinement.

All these figures also show that the choice for the termination criteria for the standard and mixed precision iterative refinement methods (Equation (6.5)) is very good, especially for the standard iterative refinement which achieves only slightly worse results than the binary cascade iterative refinement, which computes the necessary number of iterations and the required precisions before entering the iterative process based on the input data to guarantee that the process terminates after achieving the target precision.

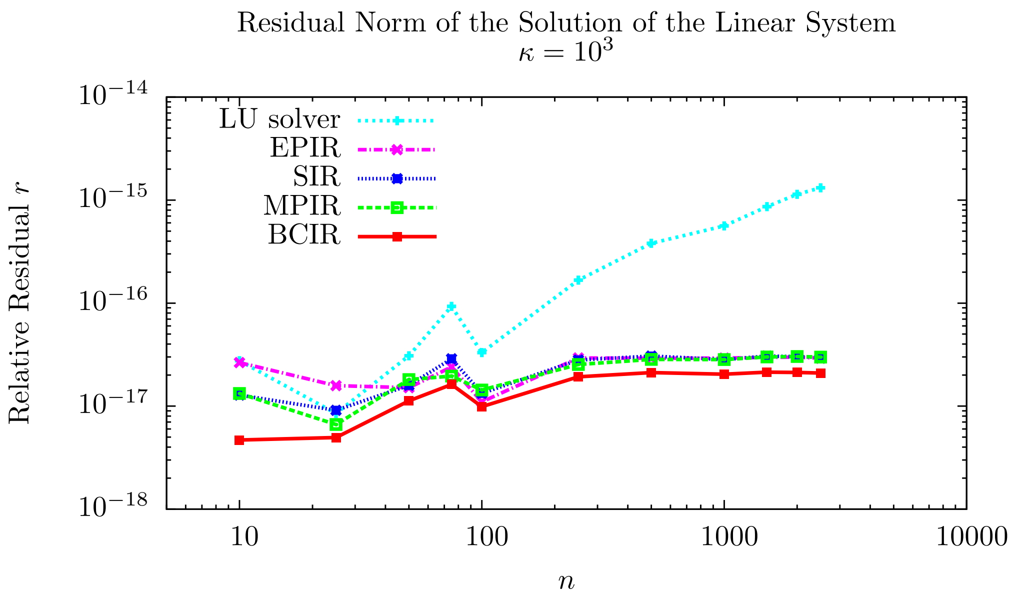

Figure 6.7: Relative residual for different iterative refinement methods for $\kappa = 10^3$

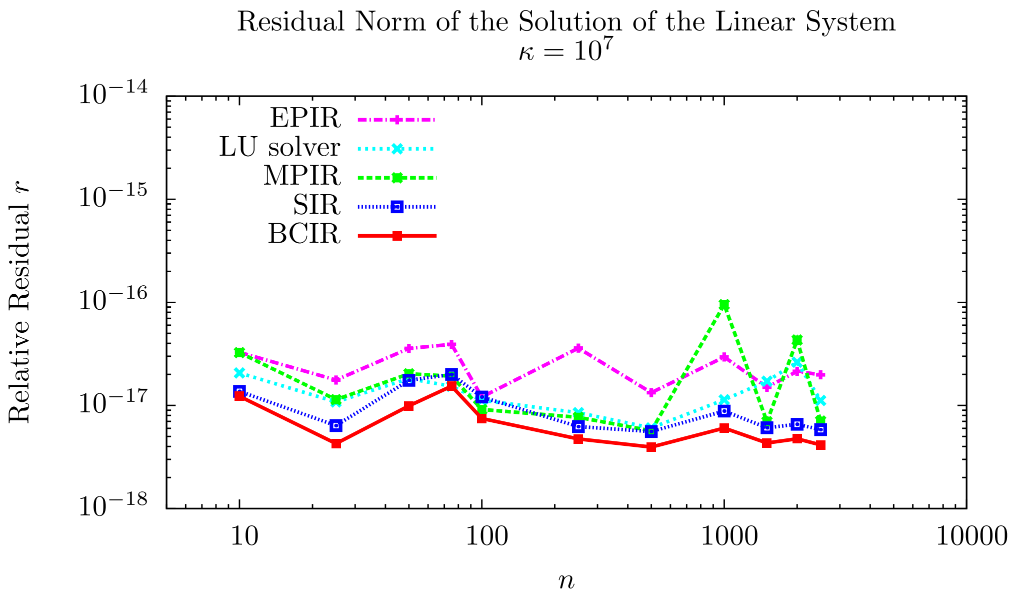

Figure 6.8: Relative residual for different iterative refinement methods for $\kappa = 10^7$

6.5.2 Hilbert Matrices

Hilbert matrices are extremely ill-conditioned by nature and were therefore chosen as an alternate input system to the randomly generated matrices. The main reason was to observe how good the iterative refinement methods can handle the ill-conditioned linear systems. Especially interesting was the behaviour of the binary cascade iterative refinement because it adapts the working precisions based on the input data.

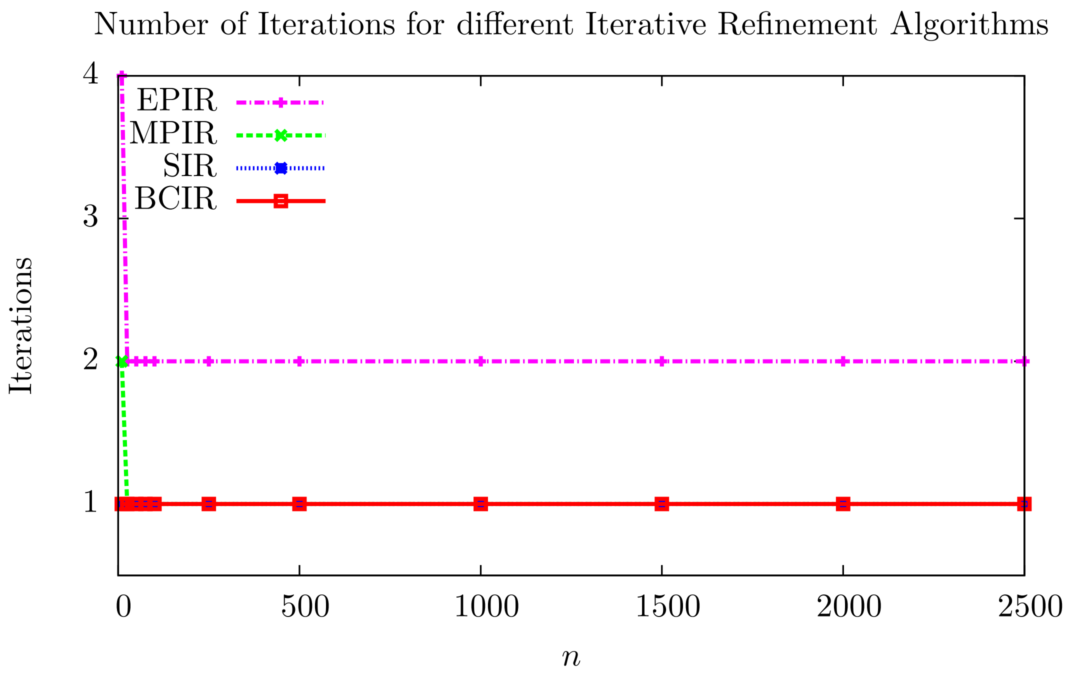

Figure 6.9: Number of iterations for the different iterative refinement methods for Hilbert matrices

The first numerical aspect is the number of iterations required until the iterative improvement converged or a termination criteria was met. As seen in Figure 6.9, all methods except for EPIR required one iteration to converge. For the smallest system in the experimental data, $n = 10$, the mixed precision iterative refinement ran for two iterations, but otherwise also halted the improvement after one iteration. The extra precise iterative refinement always used two iterations. The first iteration computed in the target precision did not converge and the error estimate did not decrease significantly enough. Therefore the working precision of EPIR was doubled and the process continued for a second iteration. The error estimate still did not return a satisfactory result and therefore the process was terminated.

As expected after the analysis and the previous experiments, BCIR used very high precisions to improve the solution of the ill-conditioned linear system with $\beta > 130$ bits. For a large system with $n = 2500$, BCIR used $\beta = 152$ bits as the working precision, almost three times the target precision $\alpha = 53$.

In Figure 6.10, the relative residual is shown on the y-axis for variable system sizes plotted on the x-axis. It is interesting to note that the mixed precision iterative refinement achieved

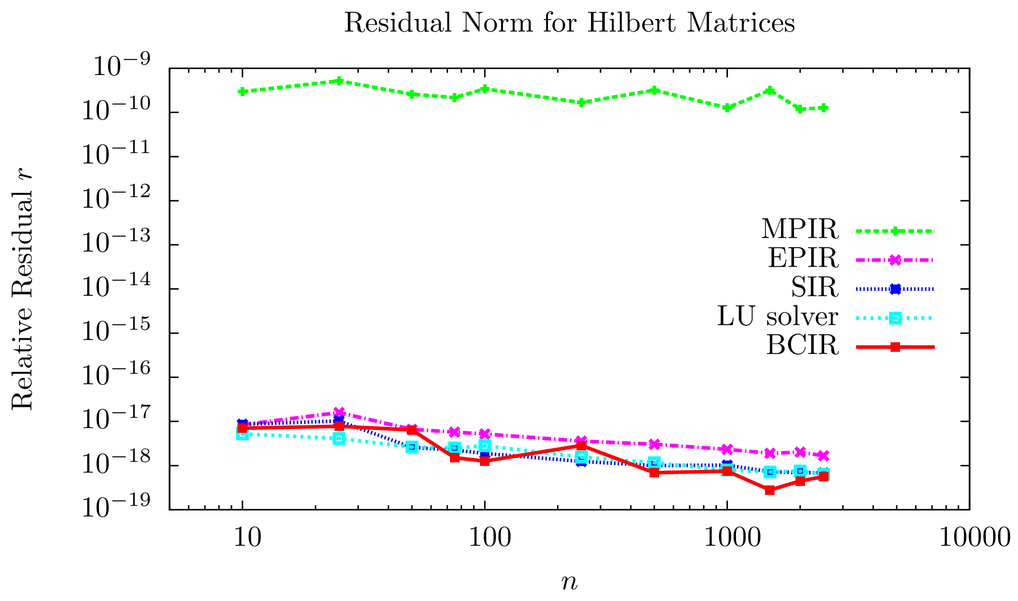

Figure 6.10: Relative residual for different iterative refinement methods for Hilbert matrices

the worst relative residual with a difference of 8 orders of magnitude compared to all other methods, including the direct LU solver. This behaviour is a result of the use of the lower working precision and the iterative refinement being halted after one iteration. The termination criteria (Equation (6.5)) has been met because the residual or the correction term is below $n\epsilon_\alpha\kappa$. The other iterative refinement methods performed hardly better than the direct LU solver, although the binary cascade iterative refinement sometimes produced a slightly better result.

Nevertheless, the solution for the linear system with a Hilbert matrix returns inaccurate solutions for all methods with absolute residuals significantly larger than 1. With such large absolute errors, the difference between MPIR and the other solvers is almost irrelevant, because the result will always be wrong.

6.6 Experiments for Refined Version of BCIR

BCIR does not check for convergence during the iterative, more precisely the recursive, process. Therefore the precomputed parameters are the only controlling mechanism of the iterative refinement. The algorithm computes the number of recursive calls p and the working precisions β_j before entering the refinement phase, both values depending on parameter c.

In Section 4.3, the original definition of parameter c was replaced by $c = \log_2 B + 5$, showing that the BCIR process should mainly depend on the Bauer condition number. The following experiments should help confirm or disprove this claim. The program `bcir` was therefore extended to use the new definition of c under the assumption of $B = \kappa$ based on Equation (4.8). In the following plots, this version of BCIR will be called **Refined BCIR**.

6.6.1 Precisions used by Refined BCIR

Figure 6.11 shows the precisions used by the refined version of BCIR for large linear systems with $n = 2000$ and different condition numbers, which are plotted on the x-axis. The relative residual achieved for the tested systems is plotted on the second y-axis on the right. The dotted horizontal line marks the target precision α, which is the standard double precision with 53 bits to store the significand of the floating-point numbers.

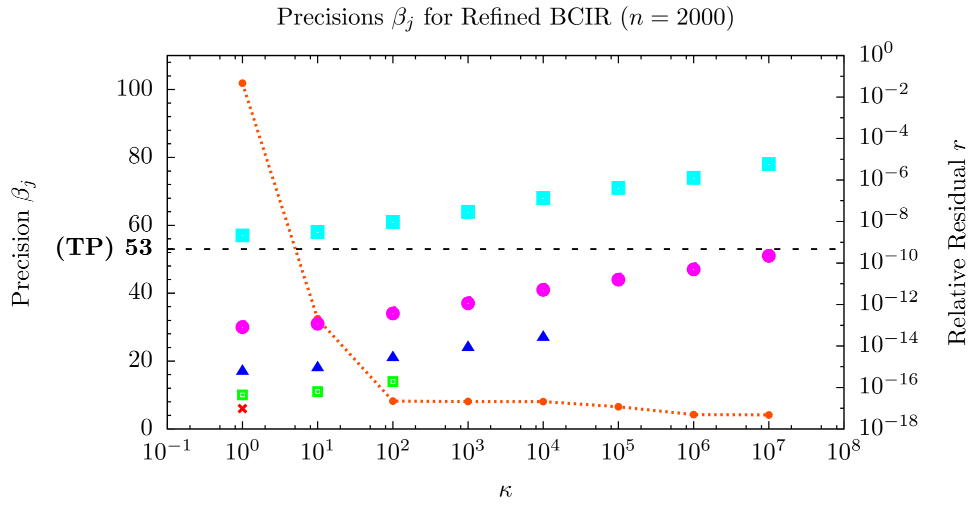

Figure 6.11: Precisions β_j used throughout the iterative process by the refined version of BCIR plotted on the left y-axis and the relative residual depicted by the orange dotted line on the right y-axis

The first notable difference compared to the results of the standard BCIR method (Section 6.4) is that the precisions used by the refined version of BCIR are much lower. The highest precision used for the most ill-conditioned system is 78 bits compared to 100 bits used by the standard BCIR. The number of recursive levels has also increased compared to the standard binary cascade iterative refinement, which only used two different working precisions for systems with a condition number $\kappa = 1$ or $\kappa = 10$ and otherwise only used one recursive call operating at a very high working precision. The refined version of BCIR uses five recursive calls for perfectly conditioned linear systems with the initial working precision being as low as 6 bits. The number of recursive level reduces as the condition number rises, but even for the most ill-conditioned system in the experiments, the process still uses two recursive refinement steps. Except for the last working precision, all working precisions are lower than the target precision, most of the time significantly lower.

This plot is identical for almost all input dimensions because parameter c no longer depends on the size n as part of its calculation. The only difference occurs for well-conditioned systems with $n \leq 41$. As described in Equation (4.23), for these small systems the second term $n/2$ of the parameter p takes precedence and influences the choice of the working precisions. However, this only result in a small change as the lowest working precisions are not used for well-conditioned systems but all other linear systems are improved using the same precisions as seen in Figure 6.11.

The relative residual achieved by the refined version of BCIR for well-conditioned systems is very bad. As the condition number rises, the relative residual is similar to the accuracy achieved by the standard BCIR. The failure of achieving a better relative residual for the perfectly conditioned system is most likely due to the low working precision used by BCIR. Using only 6 bits of accuracy for the factorization of the matrix and solving the linear systems for the solution and the correction terms is too low to return any significant digits. In decimal digits, 6 bits would only be under 2 decimal digits, obviously not enough information to compute the entire solution or to improve the result due to the error accumulation over all matrix elements.

6.6.2 Number of Iterations

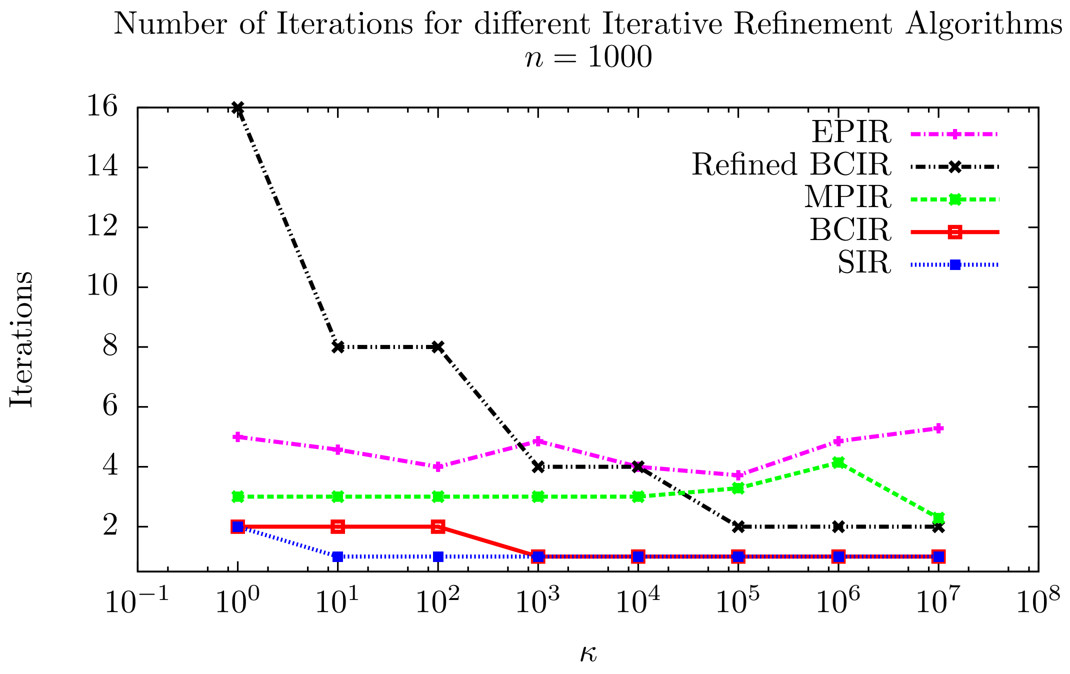

Figure 6.12: Number of iterations for the different iterative refinement methods for $n = 1000$

As seen in the last plot, the number of iterations of the refined version have greatly increased compared to the standard BCIR version. Figure 6.12 shows the number of iterations for all iterative refinement algorithms for a fixed system size of $n = 1000$ and different condition numbers plotted on the x-axis. The maximum number of iterations of the refined version is 16 for perfectly conditioned linear systems. For well-conditioned systems it uses more iterations than the extra precise iterative refinement, but the majority of these iterations are performed at significantly lower working precisions, whereas EPIR uses a working precision higher than the target precision for at least half of its total number of iterations.

For ill-conditioned systems, the refined version of BCIR requires fewer iterations than MPIR, but for these systems the mixed precision iterative refinement computes the improvement at a much lower working precision than the refined BCIR. In all cases, the refined version uses more iterations than the standard BCIR version.

6.6.3 Relative Residual

The relative residual achieved by all iterative refinement methods is plotted in Figure 6.13 for varying system sizes on the x-axis. The plot shows the accuracy of the improved solution for systems with a condition number of $\kappa = 1$. The refined version of BCIR produces the same relative residual up to $n = 100$, but then performs very poorly and fails completely to achieve any acceptable accuracy. The process never converges for $n \geq 500$. This effect can be accounted to the use of the low working precision of only 6 bits for the factorization of the matrix and solving the linear systems for the solution and the correction terms. The number of significant digits is far too low, especially when accumulating the errors over larger matrices.

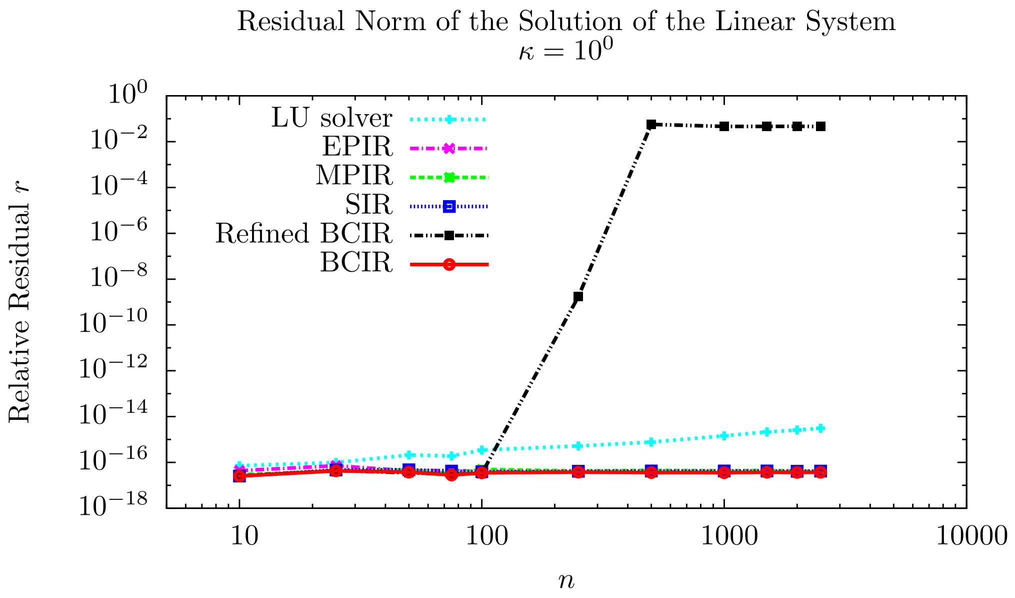

Figure 6.13: Relative residual for different iterative refinement methods for $\kappa = 1$

This effect also occurred for experiments with linear systems having a condition number of $\kappa = 10$, but the results started to deviate from the relative residual achieved by the other iterative refinement methods later than for perfectly conditioned systems. The new definition of c uses only the condition number to compute the working precisions. Therefore the working precisions used for the improvement process have increased compared to systems with $\kappa = 1$ and the lowest working precision is $\beta_0 = 11$ bits instead of $\beta_0 = 6$ bits for perfectly conditioned systems. The accumulated errors therefore influence the accuracy of the solution at a later stage than for lower working precisions.

All other measurements resulted in a very similar accuracy as the standard implementation of BCIR, but this does not mean that the same effect will not occur when the matrix dimensions increase. The effect will more than likely only occur later in the plots when computing the result of larger linear systems with the same condition number using the low working precisions.

6.6.4 Conclusion of Refined BCIR

The dimension of the system in the original definition of c has a greater influence than assumed in the final remark of the author [26] and is recommended to be used to guarantee a convergence of the refined results. This comes at the cost of performance because higher working precisions will be used throughout the process, even though, as can be seen by the results in the previous subsections, they are not always necessary. In the experiments, all matrices with a condition number higher or equal 10^2 and up to a dimension of $n = 2500$, the maximum size measured due to the performance of the simulated arbitrary precision, showed very similar convergence and accuracy compared to the standard BCIR algorithm using higher precisions at each recursive level.

By including the dimension n of the matrix in the refined definition of c (Equation (4.22)) the results could already be stabilised for well-conditioned matrices. However, it cannot be guaranteed that the same effect as before will not occur at a later point using larger input matrices. Including n^2 as the factor before the condition number κ, as in the original definition of c (Equation (4.4)), makes sense in order to cover all input errors for all n^2 elements of the $n \times n$ matrix.

Of course it would be desirable to reduce the working precisions necessary at each recursive level in order to gain performance due to the lower precisions, but the experiments have shown, that this will also cause the iterative refinement to fail when handling certain input data. Modifying the parameters as in Section 4.3 can lead to undesirable results and a loss in accuracy. The dimension of the system is an important factor for the choice of the working precisions and should not be left out of the equation.

Chapter 7

Performance Model

Simulating arbitrary precision with software libraries has a large negative impact on the performance and the time measurements are therefore not conclusive when trying to identify the performance gain through the use of different precisions. The performance benefits of using mixed precision algorithms can only be measured on special hardware implementations, for example FPGAs. Using the software libraries, there is hardly any performance difference when operating within the small range of mantissa widths used here for the iterative refinement methods. Therefore the performance will be compared using performance models, which account for the gains and losses due to the lower and higher working precisions.

Before defining the different performance models for all iterative refinement algorithms tested in this thesis, some basic values used in all models have to be defined. The performance models take into account fused multiply add and subtract, multiplication, division, addition and subtraction operations.

The number of fused multiply subtract operations for both, the standard and blocked LU decomposition, is

$$lu_{ops} := \frac{2n^3}{3} \tag{7.1}$$

The different working precisions have to be included in the models using working precisions which differ from the target precision to simulate the performance gain of the algorithm if these operations were effectively implemented in hardware, e.g. in FPGAs. A valid assumption for FPGAs [16] would be that the performance increases quadratically with the decrease of the precision.

$$speedup(\alpha, \beta) := \frac{\alpha^2}{\beta^2} \tag{7.2}$$

To determine the theoretical number of iterations, the iterations model previously described in Section 3.4 will be used.

$$iterations(\alpha, \beta, \kappa) := \frac{\alpha}{\beta - log_2(\kappa)} \tag{7.3}$$

7.1 Performance Models

7.1.1 Performance Model for Direct LU Solver

The simplest performance model is the one used for the algorithm using no iterative refinement at all, and therefore only consists of the LU decomposition and the forward and back substitution, which each require $n^2/2$ fused multiply subtract operations, and only uses the target precision for all computations.

$$noir_{ops} := \frac{2n^3}{3} + n^2 \tag{7.4}$$

7.1.2 Performance Model for SIR

The standard iterative refinement also only uses the target precision for all computations, but additionally requires a number of iterations to increase the accuracy of the result. In each iteration, the residual is computed using n^2 multiply operations followed by n subtract operations. Then the linear system is solved using the forward and back substitution and the correction term Δx is added to the result x using n addition operations. Finally the norm of the residual and the norm of Δx is calculated to check if the process has converged or if the iterative refinement has to continue. The two norms require n fused multiply add operations. The performance model for the standard iterative refinement is defined by the following equation:

$$sir_{ops} := \frac{2n^3}{3} + n^2 + iterations(\alpha, \alpha, \kappa) \cdot \left(2 \cdot n^2 + 4n\right) \tag{7.5}$$

7.1.3 Performance Model for MPIR

The mixed precision iterative refinement uses exactly the same number of operations as the standard iterative refinement, but most operations are computed using a lower working precision β. The LU decomposition, the initial solution and solving the linear equation in the loops of the iterative refinement are all computed using the working precision β and therefore have to be multiplied by the inverse of the theoretical speed-up gained through the usage of the lower precision. The residual, adding Δx and computing the norms are all performed at the target precision α and do not require any additional term.

$$mpir_{ops} := \left(\frac{2n^3}{3} + n^2\right) \cdot \frac{1}{speedup(\alpha, \beta)} + iterations(\alpha, \beta, \kappa) \cdot \left[n^2 \cdot \frac{1}{speedup(\alpha, \beta)} + \left(n^2 + 4n\right)\right] \tag{7.6}$$

7.1.4 Performance Model for EPIR

The extra precise iterative refinement algorithm requires additional computations before solving the linear system and starting the iterative refinement. This includes equilibrating the input system and computing the infinity norm of the equilibrated matrix.

Equilibrating the system requires $2 \cdot (n^2 + 2n)$ operations for the row and column scaling factors. An additional $2n^2$ multiplication operations for applying the scaling factors to the matrix A and n operations to scale b by the row scaling factors also have to be included in the equation.

The extra precise iterative refinement requires the condition number κ_∞ of the equilibrated matrix A_s to determine whether the working precision should be increased or not. By definition as in Equation (5.3), in order to compute the condition number, the inverse of the matrix A_s would be required. Due to the high computational cost of $O(n^3)$ for computing the inverse, a condition number estimator [21] would be used in praxis. The condition number estimator requires the LU decomposition of the matrix, which already exists for the iterative refinement process. On average the estimator requires a very low number of only 4 or 5 matrix-vector products to compute a reliable estimation of the norm of the matrix [23, p.294]. Therefore, the estimation of the norm of A_s^{-1} will be included in the performance model as using $5n^2$ operations. The infinity norm of the matrix A_s requires n^2 operations for the absolute value and n comparison operations, which are not as expensive as the multiplication or fused-multiply add operations and will therefore not be included in the performance model.

All these operations, as well as the LU decomposition and the initial solution (n^2 operations), are performed in the target precision α. The extended working precision β is only used during the iterative process and only if certain conditions (explained in Section 3.2) are met. In the first iteration of the extra precise iterative refinement all computations except the final update operation are always performed in α precision. During the process the precision can be increased to β precision. This depends on the quality of the input data as well as the rate of convergence. Adding the correction term Δx to x is performed after the precision is increased to the extended β precision and therefore the last α precision iteration will perform this update in β precision. If the working precision is increased, the extra precise iterative refinement by default doubles the target precision α. The speed-up factor for the extended working precision $\beta = 2\alpha$ is therefore reduced to 0.25.

The experiments have shown that in the majority of cases for systems with a condition number between $\kappa = 1$ and $\kappa = 10^7$, the extra precise iterative refinement performs two iterations in α precision before changing the working precision to twice the target precision and then performs on average another two iterations in β precision. Therefore the performance model will use 2 as an estimate for the number of iterations in α and β precision.

The complete performance model for the extra precise iterative refinement method is shown in the following definition, with $iter_\alpha = iter_\beta = 2$.

$$\begin{aligned} epir_{ops} \;:=\; & \left[\tfrac{2n^3}{3} + \left(10n^2 + 5n\right) + iter_\alpha \cdot \left(2n^2 + 5n\right) - n\right] + \\ & + \left[iter_\beta \cdot \left(2n^2 + 5n\right) + n\right] \cdot \tfrac{1}{0.25} \end{aligned} \tag{7.7}$$

7.1.5 Performance Model for BCIR

The binary cascade iterative refinement uses different working precisions throughout the process, which also have to be considered in the performance model. At the final recursion level, twice a linear system is solved using the lowest working precision β_0. The forward and back substitution both require $n^2/2$ fused multiply subtract operations each. In the second step of the recursive function of the BCIR algorithm (see Listing (4.2)), the residual is computed which requires a matrix vector multiplication and a vector-vector subtraction with n^2 multiply and n subtraction operations. Finally, subtracting v from z requires an additional n subtracting operations. This leads to the following number of operations at the lowest level of the recursive algorithm, which is executed 2^p times.

$$2^p \cdot \left(3n^2 + 2n\right) \tag{7.8}$$

On all other levels j of the iterative refinement, only the residual and the subtraction of the correction term is computed, which is executed at each level, a total of 2^{p-j} times, at different working precisions and therefore the number of operations would be:

$$2^{p-j} \cdot \left(n^2 + 2n\right) \tag{7.9}$$

The complete number of operations for BCIR including the LU decomposition and taking into account the speed-up gained through the use of arbitrary precision can therefore be expressed as:

$$bcir_{ops} := \left[\frac{2n^3}{3} + 2^p \cdot \left(3n^2 + 2n\right)\right] \cdot \frac{1}{speedup(\alpha, \beta_0)} + \sum_{j=1}^{p} 2^{p-j} \cdot \left(n^2 + 2n\right) \cdot \frac{1}{speedup(\alpha, \beta_j)} \tag{7.10}$$

7.2 Results for the Modelled Speed-Up

All tables and plots in this section show the modelled speed-up of the different iterative refinement methods based on the previously defined performance models compared to the direct LU solver using no iterative refinement. In the plots, the speed-up is always shown on the y-axis and the direct LU solver is represented as a dotted horizontal line as a reference. The target precision is always double precision with $\alpha = 53$ bits and the working precisions are also the same as described in Section 6.2.

Method	Speed-up
MPIR	3.5887
BCIR	1.0533
SIR	0.9431
EPIR	0.6958

Table 7.1: Modelled speed-up for different working precisions β with $\alpha = 53$, $n = 100$ and $\kappa = 350$

In Table 7.1 the speed-up is shown for a linear system with $n = 100$ and a condition number $\kappa = 350$. The standard iterative refinement, using the same precision as the target and working precision, is naturally slower than not using any iterative refinement. However, the slow-down effect is very small with 0.94 and can be justified because the method can produce better results than not using iterative refinement. The second method being slower than the direct LU solver is the extra precise iterative refinement, which uses higher working precisions to compute the critical sections and has a slow-down effect of 0.70. This is already a high performance decrease, but the process also provides additional information about the quality of the improved result. Binary cascade iterative refinement achieves a speed-up, but the performance increase of 1.05 is very low. The best speed-up is achieved by the mixed precision iterative refinement with 3.59 using standard single precision.

Method	Speed-up
MPIR	4.7103
SIR	0.9940
EPIR	0.9583
BCIR	0.4065

Table 7.2: Modelled speed-up for different working precisions β with $\alpha = 53$, $n = 1000$ and $\kappa = 350$

The performance behaviour of the methods changes when larger systems are solved. In Table 7.2 the same table is shown for a larger linear system with $n = 1000$. The standard iterative refinement is of course still slower than the direct solver, but the slow down effect has decreased to 0.99, which indicates that the extra computational work is insignificant especially compared to the gain of the improved accuracy of the solution. The influence of the extra precise iterative refinement has also dropped significantly and now only has a slow-down effect of 0.96, still providing more information about the error bounds than other iterative refinement methods. Binary cascade iterative refinement has become the slowest method and has a slow-down effect of 0.41, which is explained by the high working precisions used by the process (compare with Section 6.4). BCIR still produces the best relative residual, but this slight improvement over the other iterative refinement methods comes with a high cost at the expense of the performance. The mixed precision iterative refinement achieves again the highest speed-up, which is even higher than for smaller linear systems, with 4.71 for the standard single precision as the working precision.

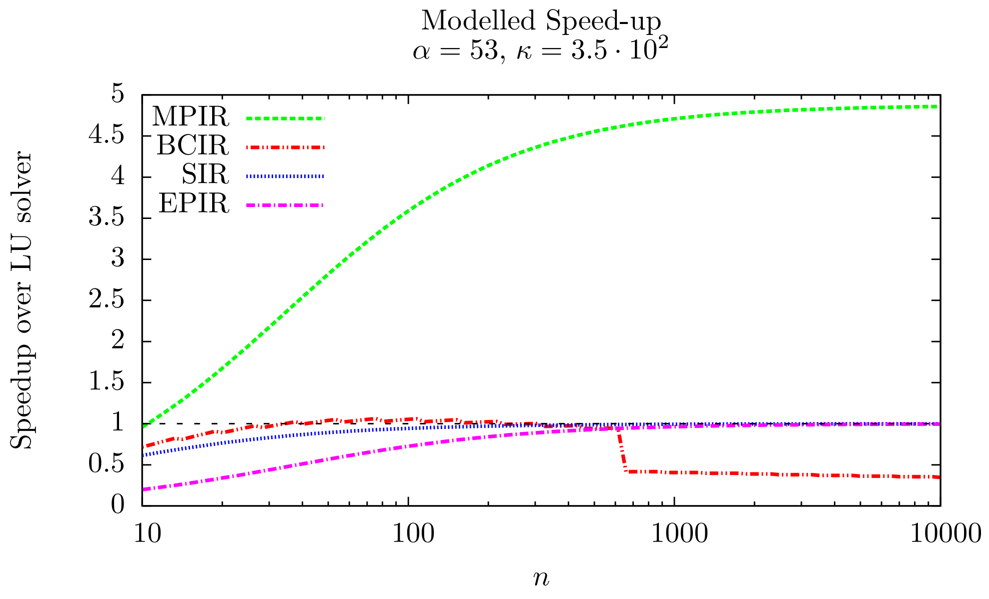

Figure 7.1: Modelled speed-up for different system sizes

Figure 7.1 shows the modelled speed-up for different dimension of linear systems, plotted on the x-axis. For larger problems the effect of the standard and extra precise iterative refinement becomes insignificant and for linear systems over $n = 1000$ both methods only have a small influence on the performance compared to the direct LU solver. The effect of MPIR already seen in the last two tables continues although the increase in performance flattens for larger linear systems. For a large system with $n = 10000$ the process achieves a speed-up of 4.86 when using single precision as the working precision. BCIR also achieves a speed-up but only for a small range of problems and the performance increase compared to the direct solver is extremely low. For larger linear systems, the method has a slow-down factor of under 0.5 due to the higher working precisions. Even though the algorithm produces a slightly better relative residual compared to the other iterative refinement methods, the cost of this improvement is very high.

Finally, Figure 7.2, shows the modelled speed-up for different condition numbers of the input matrix A and a small system size of $n = 100$. Standard and extra precise iterative refinement are not influenced by the condition number and have the same slow-down effect as in Table 7.1. Mixed precision iterative refinement requires more iterations if the condition

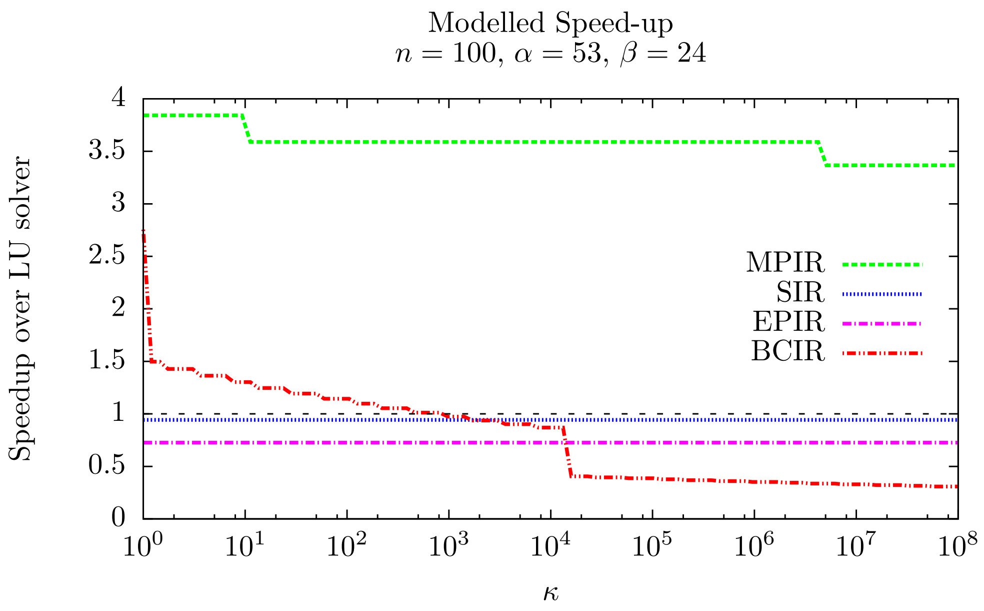

Figure 7.2: Modelled speed-up for different condition numbers

number rises and therefore the speed-up decreases with the rise of the condition number. The speed-up ranges from 3.84 for perfectly conditioned systems to 3.37 for ill-conditioned systems, but for most systems the speed-up is 3.59. Binary cascade iterative refinement is strongly influenced by the condition number due to the influence of κ on the choice of the working precisions β_j. At the beginning, for perfectly conditioned systems, BCIR achieves a high speed-up of 2.76, but this decreases immediately and for systems with a condition number higher than $\kappa = 10^3$ no speed-up is achieved. For the ill-conditioned systems the slow-down factor is 0.31. The large leaps in the graph can be explained by the decrease of the number of recursive levels of BCIR and the simultaneous increase of the working precisions.

Chapter 8

Conclusion

The binary cascade iterative refinement introduced two very good ideas to iterative refinement: the choice of the working precisions should be based on properties of the input data and the working precisions can increase with each iteration.

One problem that arose during the examination of the binary cascade iterative refinement was the ambiguous definition of the Bauer condition number. Based on all available information it was concluded that the Bauer condition number was not vital to the choice of the working precisions and was therefore removed from the equation. However, the author mentioned an alternative approach for the choice of the working precisions which, contradictory to the previous findings, relies completely on the Bauer condition number. The experiments have shown that this variation of the algorithm results in an unreliable process in terms of the accuracy of the solution.

In the experiments, the binary cascade iterative refinement almost always returned the best relative residual for all tested input data. The termination criteria of the standard and mixed precision iterative refinement has proven to be a good choice, because the accuracy was only slightly worse compared to BCIR, which chooses the working precisions to achieve the best accuracy. One goal of the extra precise iterative refinement is to compensate for ill-conditioned systems by increasing the working precision. However, in the experiments, EPIR often returned larger relative residuals than other iterative refinement methods, a result which especially occurred for ill-conditioned systems. MPIR produced very good relative residuals in most cases, but failed to return a good relative residual for the extremely ill-conditioned Hilbert matrices due to the use of the lower working precision. For these problems, even the direct LU solver produced a better relative residual than most other iterative refinement methods.

The experiments have shown that BCIR uses a very low number of iterations but these are performed at very high working precisions which are normally significantly higher than the target precision. As seen in the performance model, for most input data the process will not achieve a speed-up, but instead will be considerably slower than the other tested algorithms and the direct LU solver. BCIR often executes only one iteration of the refinement process, which is executed at a working precision higher than the target precision. The matrix factorization in BCIR is also executed at this higher precision and therefore the process is naturally slower than a standard iterative refinement operating at the target precision.

The extra precise iterative refinement uses the highest number of iterations and most of these iterations are performed at a higher working precision which is double the target precision. The performance of EPIR is not very good for small linear systems, but the model predicts that the influence of the refinement process on the overall performance decreases with the increase of the dimension of the linear system, because the computationally more expensive task of computing the factorization will dominate the performance and the higher working precisions used in the few iterations will no longer be a deciding factor. The accuracy may not always be as good as the standard iterative refinement, but the process also returns error bounds which provide information about the quality of the result. The mixed precision iterative refinement aims on achieving a high performance and the performance model confirms this behaviour. Due to the use of the lower working precision for the computationally expensive steps of the solver, MPIR can achieve a very high performance while still achieving the target precision accuracy.

In terms of accuracy of the solution, the binary cascade iterative refinement is unbeaten by the other iterative refinement methods compared in this thesis, which is largely due to the adaptive choice of the working precisions. However, from a performance point of view, BCIR cannot compete with the other iterative refinement methods. The differences in the relative residual of the results compared to the standard iterative refinement are not significant enough to justify the high computational costs.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen, *LAPACK Users' guide (third ed.)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999.

[2] D. H. Bailey, Y. Hida, X. S. Li, and O. Thompson, "ARPREC: An arbitrary precision computation package," Tech. Rep., 2002.

[3] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, and G. Quintana-Ortí, "Exploiting the capabilities of modern gpus for dense matrix computations," *Concurr. Comput. : Pract. Exper.*, vol. 21, pp. 2457–2477, December 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1656506.1656515

[4] F. Bauer, "Genauigkeitsfragen bei der Lösung linearer Gleichungssysteme," *Z. Angew. Math. Mech*, vol. 46, no. 7, pp. 409–421, 1966.

[5] C. H. Bischof, J. G. Lewis, and D. J. Pierce, "Incremental condition estimation for sparse matrices," vol. 11, no. 4, pp. 644–659, 1990. [Online]. Available: http://dx.doi.org/doi/10.1137/0611047

[6] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The International Journal of High Performance Computing Applications*, vol. 14, pp. 189–204, 2000.

[7] A. Buttari, J. Dongarra, and J. Kurzak, "Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy," *ACM Transactions on*, vol. 34, no. 4, 2008. [Online]. Available: http://doi.acm.org/10.1145/1377596.1377597

[8] S. Charles and K. Dowd, "Floating-Point Numbers - History of IEEE Floating-Point Format," 2010, [accessed 25-April-2011]. [Online]. Available: http://cnx.org/content/m32770/1.3/

[9] Charles Severance, "An Interview with the Old Man of Floating-Point," 1998, [accessed 12-May-2011]. [Online]. Available: http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

[10] Committee, Microprocessor Standards, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 7542008*, vol. 2008, no. August, pp. 1–58, 2008. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.2008.4610935

[11] B. N. Datta, *Numerical linear algebra and applications*, 2nd ed. Society for Industrial Mathematics, 2010.

[12] P. I. Davies, N. J. Higham, and F. Tisseur, "Analysis of the Cholesky Method with Iterative Refinement for Solving the Symmetric Definite Generalized Eigenproblem," *SIAM J. Matrix Anal. Appl.*, vol. 23, pp. 472–493, February 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=587704.587754

[13] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy, "Error bounds from extra-precise iterative refinement," *ACM Transactions on Mathematical Software*, vol. 32, no. 2, pp. 325–351, Jun. 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1141885.1141894

[14] J. Demmel, N. Higham, and R. Schreiber, "Block LU Factorization," *Circulation research*, vol. 109, no. 2, pp. 202–4, Jul 1992.

[15] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, p. 13, 2007.

[16] W. Gansterer, M. Mücke, and K. Prikopa, "Arbitrary Precision Iterative Refinement," 2011, in preparation.

[17] K. Ghazi, V. Lefevre, P. Théveny, and P. Zimmermann, "Why and how to use arbitrary precision," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 5–5, 2010. [Online]. Available: http://perso.ens-lyon.fr/philippe.theveny/cise.pdf

[18] G. Golub and J. Wilkinson, "Note on the iterative refinement of least squares solution," *Numerische Mathematik*, vol. 9, pp. 139–148, 1966, 10.1007/BF02166032. [Online]. Available: http://dx.doi.org/10.1007/BF02166032

[19] G. Golub and C. Loan, *Matrix computations*, ser. Johns Hopkins studies in the mathematical sciences. Johns Hopkins University Press, 1996.

[20] T. Granlund, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 2011. [Online]. Available: http://www.gmplib.org

[21] W. W. Hager, "Condition estimates," vol. 5, no. 2, pp. 311–316, 1984. [Online]. Available: http://dx.doi.org/doi/10.1137/0905023

[22] M. Heath, *Scientific computing: an introductory survey*, ser. McGraw-Hill series in computer science. McGraw-Hill, 2002.

[23] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Society for Industrial and Applied Mathematics Philadelphia, Mar. 2002, vol. 94, no. 445.

[24] N. Higham, "Iterative refinement for linear systems and LAPACK," *IMA Journal of Numerical Analysis*, vol. 17, no. 4, pp. 495–509, Oct. 1997. [Online]. Available: http://imanum.oupjournals.org/cgi/doi/10.1093/imanum/17.4.495

[25] A. Kiełbasiński, "Solving Linear Systems in Unusually High Precisions," Dept. of Math., Linköping University, Report 32, 1979.

[26] ——, "Iterative refinement for linear systems in variable-precision arithmetic," *BIT Numerical Mathematics*, vol. 11, pp. 97–103, 1981. [Online]. Available: http://dx.doi.org/10.1007/BF01934074

[27] J. Kurzak, A. Buttari, and J. Dongarra, "Solving systems of linear equations on the CELL processor using Cholesky factorization," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 19, no. 9, pp. 1175–1186, 2008. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2007.70813

[28] LAPACK, "LAPACK 3.2: DQEEQUB," 2011, [Online; accessed 11-July-2011]. [Online]. Available: http://www.netlib.org/lapack/explore-3.2-html/dgeequb.f.html

[29] D. Lichtblau and E. W. Weisstein, "Condition Number." [Online]. Available: http://mathworld.wolfram.com/ConditionNumber.html

[30] A. Lumsdaine, J. Siek, and L.-Q. Lee, "The Matrix Template Library (MTL) LU Factorization Example," 2006, [Online; accessed 9-July-2011]. [Online]. Available: http://osl.iu.edu/research/mtl/tutorial.php3

[31] Maplesoft, "Maplesoft - Technical Computing Software for Engineers, Mathematicians, Scientists, Instructors and Students," 2011, [Online; accessed 16-May-2011]. [Online]. Available: http://www.maplesoft.com

[32] R. Martin, G. Peters, and J. Wilkinson, "Iterative refinement of the solution of a positive definite system of equations," *Numerische Mathematik*, vol. 8, pp. 203–216, 1966, 10.1007/BF02162558. [Online]. Available: http://dx.doi.org/10.1007/BF02162558

[33] Mathworks, "Variable precision arithmetic - Matlab," Retrieved 3 2010. [Online]. Available: http://www.mathworks.com/access/helpdesk/help/toolbox/symbolic/vpa.html

[34] Microsoft, "BigInteger Structure (System.Numerics)," 2011, [Online; accessed 14-May-2011]. [Online]. Available: http://msdn.microsoft.com/en-us/library/system.numerics.biginteger.aspx

[35] C. B. Moler, "Iterative Refinement in Floating Point," *Journal of the ACM*, vol. 14, no. 2, pp. 316–321, Apr. 1967. [Online]. Available: http://portal.acm.org/citation.cfm?doid=321386.321394

[36] M. Mücke, B. Lesser, and W. N. Gansterer, "Peak Performance Model for a Custom Precision Floating-Point Dot Product on FPGAs," in *Third Workshop on UnConventional High performance Computing 2010 (UCHPC 2010)*, 2010.

[37] Oracle, "Class BigDecimal," 2011, [Online; accessed 14-May-2011]. [Online]. Available: http://download.oracle.com/javase/1.4.2/docs/api/java/math/BigDecimal.html

[38] ——, "Class BigInteger," 2011, [Online; accessed 14-May-2011]. [Online]. Available: http://download.oracle.com/javase/1.4.2/docs/api/java/math/BigInteger.html

[39] D. J. Pierce and R. J. Plemmons, "Fast adaptive condition estimation," vol. 13, no. 1, pp. 274–291, 1992. [Online]. Available: http://dx.doi.org/doi/10.1137/0613021

[40] J. Rice, *Matrix computations and mathematical software.* McGraw-Hill, 1981.

[41] R. Skeel, "Iterative refinement implies numerical stability for Gaussian elimination," *Mathematics of Computation*, vol. 35, no. 151, pp. 817–832, 1980.

[42] Sun Microsystems, *Sun Studio 11: Numerical Computation Guide.* Sun Microsystems, Inc., 2005. [Online]. Available: http://download.oracle.com/docs/cd/E19422-01/819-3693/819-3693.pdf

[43] The MPFR Team, *GNU MPFR: The Multiple Precision FLoating-Point Reliable Library*, 2011. [Online]. Available: http://www.mpfr.org

[44] D. Veillard, "The XML C parser and toolkit of Gnome," Retrieved 7 2010. [Online]. Available: http://xmlsoft.org/

[45] R. C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software," in *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999, cD-ROM Proceedings.

[46] Wikipedia, "Arbitrary-precision arithmetic — Wikipedia, The Free Encyclopedia," 2011, [Online; accessed 13-May-2011]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Arbitrary-precision_arithmetic&oldid=428303523

[47] ——, "Field-programmable gate array — Wikipedia, The Free Encyclopedia," 2011, [Online; accessed 15-May-2011]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Field-programmable_gate_array&oldid=427670193

[48] ——, "General Floating Point Frac — Wikipedia, The Free Encyclopedia," 2011, [accessed 23-April-2011]. [Online]. Available: http://en.wikipedia.org/wiki/File:General_floating_point_frac.svg

[49] J. Wilkinson, *Rounding Errors in Algebraic Processes.* Her Majesty's Stationery Office, London, 1963.

[50] ——, *The algebraic eigenvalue problem.* Oxford University Press, USA, Oct. 1965, vol. 20.

Appendix A

Additional and Extended Information

A.1 List of available Arbitrary Precision Libraries

The following table provides an overview of available arbitrary precision libraries for different programming languages and data types.

Package / Library Name	**Number Type**	**Language**
GNU MPFR	Integers, rationals and floats	C and C++ with bindings
apfloat	Decimal floats, integers, rationals, and complex	Java and C++
BeeCrypt Cryptography Library	Integers	Assembly, C, C++, Java
ARPREC and MPFUN	Integers, binary floats, complex binary floats	C++ with C++ and Fortran bindings
Base One Number Class	Decimal floats	C++
bbnum library	Integers and floats	Assembler and C++
phpseclib	Decimal floats	PHP
BigDigits	Naturals	C
BigFloat	Binary Floats	C++
BigNum	Binary Integers, Floats (with math functions)	C# / .NET
C++ Big Integer Library	Integers	C++
CLN, a Class Library for Numbers	Integers, rationals, floats and complex	C and C++
Computable Real Numbers	Reals	Common Lisp
IMSL		C

decNumber	Decimals	C
FMLIB	Floats	Fortran
GNU Multi-Precision Library (and MPFR)	Integers, rationals and floats	C and C++ with bindings (GMPY,...)
MPCLI	Integers	C# / .NET
C# Bindings for MPIR (MPIR is a fork of the GNU Multi-Precision Library)]	Integers, rationals and floats	C# / .NET
GNU Multi-Precision Library for .NET	Integers	C# / .NET
Eiffel Arbitrary Precision Mathematics Library	Integers	Eiffel
HugeCalc	Integers	C++ and Assembler
IMath	Integers and rationals	C
IntX	Integers	C# / .NET
JScience LargeInteger	Integers	Java
libgcrypt	Integers	C
libmpdec (and cdecimal)	Decimals	C, C++ and Python
LibTomMath	Integers	C and C++
LiDIA	Integers, floats, complex floats and rationals	C and C++
MAPM	Integers and decimal floats	C (bindings for C++ and Lua)
MIRACL	Integers and rationals	C and C++
MPI	Integers	C
MPArith	Integers, floats, and rationals	Pascal / Delphi
mpmath	Floats, complex floats	Python
NTL	Integers, floats	C and C++
bigInteger (and bigRational)	Integers and rationals	C and Seed7
TTMath library	Integers and binary floats	Assembler and C++
vecLib.framework	Integers	C
W3b.Sine	Decimal floats	C# / .NET
Eiffel Arbitrary Precision Mathematics Library (GMP port)	Integers	Eiffel
BigInt	Integers	JavaScript

Table A.1: This table shows a list of available arbitrary precision libraries for different programming languages and number types [46]

A.2 Project Files

The following table provides an overview of the different files created and used in the project and provides a short description of their functions.

File	**Description**
bcir.c	Reads input parameters and input data, calls the BCIR function and saves the results to a file.
apir.c	Reads input parameters and input data, calls the APIR function and saves the results to a file.
extir.c	Reads input parameters and input data, calls the EPIR function and saves the results to a file.
noir.c	Reads input parameters and input data, calls the NoIR function and saves the results to a file.
generateMatrix.c	Generates a matrix and vector and saves the data in a file.
bitdpconverter.c	Program to convert between the different representations (bits and decimal digits).
bcir_mpfr.h	Functions for running binary cascade iterative refinement.
apir_mpfr.h	Functions for running arbitrary precision iterative refinement.
epir_mpfr.h	Functions for running extra precise iterative refinement.
noir_mpfr.h	Functions for running a standard LU decomposition with subsequent forward and back substitution.
mpfr_lu.h	LU decomposition with and without partial pivoting and with and without using the blocked implementation. Versions for double and mpfr are included.
conditionNumber.h	Calculates the condition number and can change the condition number of a matrix.
arbitrary_precision.h	Provides functions for converting precisions between the different representations.
ir_io.h	Provides functions for loading and saving matrices and vectors from and to a file.
iocompression.h	Provides functions for compressing and decompressing data files and determining if a file is compressed or not.

bcir_results.h apir_results.h epir_results.h noir_results.h	Provides functions for printing the results as a table to the screen or to a file and saving detailed results as an XML file.
ir_clapack.h f2c.h	Declarations of the LAPACK functions used by the iterative refinement implementations.
papi_extras.h	Defines which PAPI counters to activate and the function to handle PAPI errors.
timings.h	Includes the functions for measuring the execution time and for processing the measured times.
vector_matrix	The folder contains functions for creating, copying, filling and deleting vector and matrix data structures for double, int and mpfr.

A.3 Test Environment

All tests were conducted on a Sun Fire X4600 M2 Server with the following specifications:

- 8 AMD Opteron 8218 Dual-Core processors with 2.6 GHz resulting in 16 cores, each with 1 MB Level 2 cache
- HyperTransport link used to connect the CPUs to each other (8 GB/s)
- 32GB of main memory
- Operating System: Ubuntu 10.04.2 LTS

The following versions of the libraries were used:

- GNU MPFR 3.0.1 with GNU GMP 5.0.1
- ATLAS BLAS 3.9.17
- LAPACK 3.2.1

Appendix B

Summaries and Curriculum Vitae

B.1 English Summary

Iterative refinement is a widely used method to improve the round-off errors of a solution of a linear system and is also used in software packets like LAPACK. The process computes the residual of the solution, then solves the system for a correction term using the residual as the right hand side of the equation and finally updates the solution with the correction term. These steps are repeated until the requested accuracy is reached. This method was first mentioned by Wilkinson in his book "Rounding Errors in Algebraic Processes" using fixed point arithmetic and later expanded by Moler to cover floating point arithmetic. The cost of the iterative improvement is very low compared to the cost of the factorization of the matrix but results in a solution which can be accurate to machine precision.

The standard iterative refinement (SIR) uses the same precision to compute both, the initial solution and the correction term for the improved result, but other iterative refinement methods exist, which use different precisions for these computation steps. The Extra Precise Iterative Refinement (EPIR) uses a higher precision to compute the residual and the correction term of the solution to compensate for slow convergence and ill-conditioned systems. The Mixed Precision Iterative Refinement (MPIR) takes a different approach and computes the matrix decomposition and the initial solution in single precision and applies iterative refinement using double precision to improve the result and still have a solution which reaches double precision accuracy. This exploits the benefits of using the lower single precision, for example exploiting vector instructions and using less storage which also reduces the amount of data moved through the memory hierarchy, while still achieving a double precision result.

The focus of the master thesis lays on the analysis and evaluation of the Binary Cascade Iterative Refinement (BCIR) by Kiełbasiński. The algorithm adapts the precisions for computing the refinement steps depending on the input parameters, the size and condition number of the matrix and the intended target precision. The process can use multiple working precisions throughout the refinement process. This provides the ability to choose the appropriate precision to improve the result and also compensate for ill-conditioned systems. This algorithm has never been implemented prior to this master thesis and therefore no experimental results were available in the literature. The binary cascade iterative refinement introduced two very good ideas to iterative refinement: the choice of the working precisions should be based on properties of the input data and the working precisions can increase with each iteration.

The algorithm depends on arbitrary precision, which is not bound to IEEE standard precision. The hardware support for arbitrary precision is very limited and therefore a software library, the GNU Multiple Precision Floating-Point Reliable Library (GNU MPFR), is used to implement the iterative refinement methods. This library provides a portable implementation of arbitrary precision and allows the precisions to be set exactly in the number of bits stored in the mantissa of the floating point number.

BCIR was compared to other iterative refinement methods and the numerical accuracy and the convergence have been analysed. The numerical behaviour of BCIR was analysed for different input systems, which also included extremely ill-conditioned Hilbert matrices. Due to the software simulated arbitrary precision, performance measurements do not provide accurate information about the gains and losses in performance due to the use of the different precisions. Therefore a performance model was introduced in order to be able to compare the performance of the algorithms and to analyse the possible performance gains.

In the experiments, BCIR almost always returned the best relative residual for all tested input data. The termination criteria of SIR and MPIR has proven to be a good choice, because the accuracy was only slightly worse compared to BCIR, which chooses the working precisions to achieve the best accuracy. EPIR often returned larger relative residuals than other iterative refinement methods, a result which especially occurred for ill-conditioned systems. MPIR produced very good relative residuals in most cases, but failed to return a good relative residual for the extremely ill-conditioned Hilbert matrices due to the use of the lower working precision. For these problems, even the direct LU solver produced a better relative residual than most other iterative refinement methods.

The experiments have shown that BCIR uses a very low number of iterations but these are performed at very high working precisions which are normally significantly higher than the target precision. The performance model shows that for most input data the process will not achieve a speed-up, but instead will be considerably slower than the other tested algorithms and the direct LU solver. BCIR often executes only one iteration of the refinement process, which is executed at a working precision higher than the target precision. The matrix factorization in BCIR is also executed at this higher precision and therefore the process is naturally slower than a standard iterative refinement operating at the target precision.

EPIR uses the highest number of iterations and most of these iterations are performed at a higher working precision which is double the target precision. The performance of EPIR is not very good for small linear systems, but the model predicts that the influence of the refinement process on the overall performance decreases with the increase of the dimension of the linear system, because the computationally more expensive task of computing the factorization will dominate the performance and the higher working precisions used in the few iterations will no longer be a deciding factor. The accuracy may not always be as good as the standard iterative refinement, but the process also returns error bounds which provide information about the quality of the result. MPIR aims on achieving a high performance and the performance model confirms this behaviour. Due to the use of the lower working precision for the computationally expensive steps of the solver, MPIR can achieve a very high performance while still achieving the target precision accuracy.

In terms of accuracy of the solution, the binary cascade iterative refinement is unbeaten by the other iterative refinement methods compared in this thesis, which is largely due to the adaptive choice of the working precisions. However, from a performance point of view, BCIR cannot compete with the other iterative refinement methods. The differences in the relative residual of the results compared to the standard iterative refinement are not significant enough to justify the high computational costs.

B.2 Deutsche Zusammenfassung

Iterative Refinement ist eine weitverbreitete Methode um die Rundungsfehler einer Lösung eines linearen Gleichungssystems zu verbessern und wird auch in Software Bibliotheken wie LAPACK verwendet. Der Algorithmus berechnet zuerst das Residuum der Lösung des Gleichungssystems und löst das System für einen Korrekturterm unter Verwendung des Residuums als rechte Seite der Gleichung. Diese Schritte werden solange wiederholt bis die gewünschte Genauigkeit erreicht wird. Diese Methode wurde erstmals von Wilkinson in seinem Buch "'Rounding Errors in Algebraic Processes"' für Fixpunktarithmetik erwähnt und wurde später von Moler um Gleitkommaarithmetik erweitert. Die Kosten der iterativen Verbesserung sind sehr gering im Vergleich zu den Kosten der Matrixfaktorisierung. Das Verfahren führt aber zu einem Ergebnisse welches bis zur Maschinengenauigkeit korrekt sein kann.

Standard Iterative Refinement (SIR) verwendet dieselbe Genauigkeit um die erste Lösung des Systems und den Korrekturterm zu berechnen, aber es gibt andere Iterative Refinement Methoden, welche unterschiedliche Genauigkeiten für die jeweiligen Schritte des Verfahrens verwenden. Extra Precise Iterative Refinement (EPIR) verwendet eine höhere Genauigkeit um das Residuum und den Korrekturterm zu berechnen, um für eine langsame Konvergenz oder schlecht konditionierte Systeme zu kompensieren. Mixed Precision Iterative Refinement (MPIR) verwendet einen anderen Ansatz und berechnet die Matrixzerlegung und die erste Lösung des Systems in einfacher Genauigkeit (single precision) und führt das Iterative Refinement in doppelter Genauigkeit (double precision) durch. Das Ergebnis erreicht dadurch eine Genauigkeit von double precision. Dieses Verfahren nutzt die Vorteile, welche die Verwendung der einfachen Genauigkeit mit sich bringt, zum Beispiel die Verwendung von Vektorinstruktionen oder den niedrigeren Speicherverbrauch, welcher auch den Transport der Daten durch die Speicherhierarchie beschleunigt, und erreicht trotzdem ein Ergebnis in doppelter Genauigkeit.

Der Fokus dieser Masterarbeit liegt auf der Analyse und Evaluierung des Binary Cascade Iterative Refinements (BCIR) von Kiełbasiński. Der Algorithmus wählt die Arbeitsgenauigkeiten für die Schritte des Iterative Refinements basierend auf den Eingabedaten, der Dimension und der Konditionszahl der Matrix und die gewünschte Zielgenauigkeit. Des Weiteren ist die Genauigkeit nicht auf eine Arbeitsgenauigkeit beschränkt, sondern kann mehrere Genauigkeiten während des Prozesses verwenden. Dies ermöglicht dem Verfahren, die benötigte Genauigkeit auf das Problem abzustimmen und für schlecht konditionierte Systeme zu kompensieren. Dieser Algorithmus wurde vor dieser Masterarbeit noch nie implementiert und es existierten daher auch keine Daten von Experimenten in der Literatur. BCIR stellt zwei gute Ideen für Iterative Refinement Methoden vor. Die Wahl der Arbeitsgenauigkeit sollte auf den Eigenschaften der Eingangsdaten basieren und sollte während der Iterationen ansteigen.

BCIR beruht auf der Verwendung von beliebigen Genauigkeiten, welche nicht auf die IEEE Standarddatentypen beschränkt sind, welche von den meisten Hardwareherstellern unterstützt werden. Da es kaum Hardwareunterstützung für beliebige Genauigkeiten gibt, wurde eine Softwarebibliothek, die GNU Multiple Precision Floating-Point Reliable Library (GNU MPFR), für die Implementation der Verfahren verwendet. Die Bibliothek ermöglicht eine exakte Angabe über die Anzahl der zu verwendeten Bits zur Speicherung der Mantisse der Gleitkommazahl.

Die Eigenschaften von BCIR wurden analysiert und es wurden Experimente durchgeführt, welche diesen Algorithmus mit anderen Iterative Refinement Methoden vergleichen und besondere Aufmerksamkeit auf die numerische Genauigkeit und die Konvergenz der Verfahren legten. Das numerische Verhalten wurde für unterschiedliche Eingangsdaten, darunter auch extrem schlecht konditionierte Hilbert Matrizen, untersucht. Die verschiedenen Genauigkeiten werden in Software simuliert und liefern daher keine aussagekräftigen Informationen über einen Performancegewinn oder -verlust durch die Verwendung der verschiedenen Genauigkeiten. Daher wurde ein Performancemodel vorgestellt, um die Performance der verschiedenen Methoden miteinander vergleichen zu können und Aufschluss über mögliche Performancegewinne zu erhalten.

In den Experimenten lieferte BCIR fast immer das beste relative Residuum für alle Eingangsdaten zurück. Die Abbruchbedingung für SIR und MPIR erwies sich als eine gute Wahl, da die Genauigkeit nur geringfügig schlechter war im Vergleich zu BCIR. EPIR lieferte oft größere relative Residuen zurück als die anderen Iterative Refinement Methoden, besonders im Fall von schlecht konditionierten Problemen. MPIR erzeugt in den meisten Fällen sehr gute relative Residuen, aber erreichte keine guten Ergebnisse für die extrem schlecht konditionierten Hilbert Matrizen aufgrund der Verwendung der niedrigeren Arbeitsgenauigkeit. Für diese Probleme erzielte sogar der direkte LU Löser ohne Iterative Refinement bessere Ergebnisse als die meisten anderen Methoden.

Die Experimente haben gezeigt, dass BCIR nur eine sehr geringe Anzahl an Iterationen benötigt, diese werden aber mit einer viel höheren Genauigkeit als der Zielgenauigkeit berechnet. Das Performancemodell wies auf, dass für die meisten Eingangsdaten kein Speed-up erreicht werden kann. Stattdessen wird BCIR signifikant langsamer als die anderen getesteten Verfahren sein. In vielen Fällen führt BCIR nur eine einzige Iteration durch, welche noch dazu eine deutlich höhere Genauigkeit als die Zielgenauigkeit verwendet. Da auch die Matrixzerlegung in BCIR auf diesem hohen Genauigkeitsniveau ausgeführt wird ist es leicht nachvollziehbar, dass der Prozess langsamer ist als ein Standard Iterative Refinement Verfahren, welches die Zielgenauigkeit für alle Berechnungen verwendet.

EPIR benötigt die höchste Anzahl an Iterationen und die meisten davon werden auch mit der höheren Arbeitsgenauigkeit ausgeführt, welche der doppelten Zielgenauigkeit entspricht. Für kleinere lineare Systeme ist die Performance nicht sehr gut, aber das Modell zeigt auf, dass der Einfluss des Iterative Refinements auf die Gesamtperformance mit der ansteigenden Problemgröße stark abnimmt. Die rechenintensive Zerlegung der Matrix dominiert die Performance und die Iterationen, welche in der höheren Arbeitsgenauigkeit ausgeführt werden, werden vernachlässigbar. Die relativen Residuen sind vielleicht nicht immer so gut wie bei SIR, aber dafür liefert das Verfahren auch Informationen über die Qualität des Ergebnisses zurück. Das Ziel von MPIR ist das Erreichen einer hohen Performance, welches auch vom Modell bestätigt wird. Durch die Verwendung der niedrigeren Genauigkeit für die aufwendigen Operationen des Gleichungssystemlösers kann MPIR eine sehr hohe Performance erzielen und dabei trotzdem die Zielgenauigkeit erreichen.

Im Hinblick auf die Genauigkeit der Lösung ist Binary Cascade Iterative Refinement ungeschlagen im Vergleich mit den anderen in dieser Arbeit getesteten Verfahren. Dies liegt hauptsächlich an den adaptiv gewählten Arbeitsgenauigkeiten von BCIR. Aus der Perspektive der Performance kann BCIR allerdings nicht mit den anderen Verfahren mithalten. Die Unterschiede in den relativen Residuen der Ergebnisse fallen im Vergleich zum Standard Iterative Refinement zu gering aus, um den hohen rechnerischen Aufwand zu rechtfertigen.

B.3 Curriculum Vitae

Karl Prikopa, BSc

Birth	10th March 1986, Vienna, Austria
Nationality	Austrian and British
E-Mail	karl.prikopa@univie.ac.at

Education

2009-present	**Masterstudium Scientific Computing, University of Vienna**
2006-2009	**Bachelorstudium Informatik mit Schwerpunkt Scientific Computing, University of Vienna** Abschluss mit Auszeichnung bestanden (Bachelor of Science (BSc) with distinction) Bachelor thesis: *"Performance Analysis of MPI-based Distributed Applications on Multicore Computers Using Dynamic Tracing"*
2004-2006	**Diplomstudium Chemie, University of Vienna**
1996-2004	**Gymnasium Sacré Coeur Pressbaum** Gymnasium mit 2. lebender Fremdsprache ab 3. Klasse Abschluss 2004: Matura mit ausgezeichnetem Erfolg (Matura with distinction)
1992-1996	**Volksschule der Dominikanerinnen in Hietzing, Vienna**

Research

10/2010-present	**Wissenschaftlicher Projektmitarbeiter** Fakultät für Informatik / Research Lab Computational Technologies and Applications Univ.-Ass. Privatdoz. Dr. Wilfried Gansterer
01/2010-09/2010	**Research Project** Development of an Arbitrary Precision Iterative Refinement Code Research Lab Computational Technologies and Applications Univ.-Ass. Privatdoz. Dr. Wilfried Gansterer
02/2009-12/2009	**Research Project: CPAMMS** Performance Analysis of CHARMM with OpenSolaris DTrace Research Lab Computational Technologies and Applications Univ.-Ass. Privatdoz. Dr. Wilfried Gansterer, Dipl.-Ing. Dr. Manfred Mücke

Publications

W. Gansterer, M. Mücke, and K. Prikopa, "Arbitrary Precision Iterative Refinement," 2011, in preparation.

Employment

03/2011-06/2011	**Tutor: PR Praktikum aus Computational Drug Design** Univ.-Ass. Privatdoz. Dr. Wilfried Gansterer
03/2011-06/2011	**Tutor: PR Praktikum aus Computational Technologies** Univ.-Ass. Privatdoz. Dr. Wilfried Gansterer
10/2010-02/2011	**Tutor: VU Software Tools and Libraries** Arbitrary Precision with MPFR Univ.-Ass. Privatdoz. Dr. Wilfried Gansterer, Dipl.-Ing. Dr. Dieter Kvasnicka
03/2010-06/2010	**Tutor: VU Algorithmen und Programmierung im Scientific Computing** Univ.-Ass. Privatdoz. Dr. Wilfried Gansterer
03/2010-06/2010	**Tutor: VU Visualisierung** Programming with OpenGL Dipl.-Ing. Dr. Günter Wallner, Dipl.-Ing. Dr. Alexander Wilkie
10/2009-02/2010	**Tutor: VU Software Tools and Libraries** Arbitrary Precision packages (MPFR and ARPREC) Univ.-Ass. Privatdoz. Dr. Wilfried Gansterer, Dipl.-Ing. Dr. Dieter Kvasnicka
03/2009-06/2009	**Tutor: VU Visualisierung** Programming with OpenGL Dipl.-Ing. Simone Kriglstein, Dipl.-Ing. Dr. Günter Wallner

Awards, Distinctions

2008/2009	Best of the Best in the category bachelor studies in computer science ranking position #3
2009	Bachelor with distinction (Bachelor mit Auszeichnung bestanden)
2004	Matura with distinction (Matura mit ausgezeichnetem Erfolg)

Languages German and English (mother tongues), French, Latin