The SIAM 100.0000000-Digit Challenge: A Study in High Accuracy Numerical Computing Using Interval Analysis and Mathematica

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in the
Department of Mathematics at The College of Wooster

by
James Rohal

The College of Wooster
2007

Advised by:

Charles Hampton

Abstract

This Independent Study concerns itself with computational mathematics with an emphasis on three specific problems posed during the SIAM 100-Dollar, 100-Digit Challenge [18]. The difficulty of the challenge is solving the problems with enough precision to get answers accurate to 10 digits. Most solutions put forth by contestants offered no proof of correctness for their answers. By using theory from the field of interval analysis, it will be possible to develop algorithms that will provide a solution within a desired accuracy and that is verifiably correct. To do so, I will be implementing these algorithms in *Mathematica* and using interval analysis to create computer-assisted proofs for the solutions.

Acknowledgments

First, I would like to thank Dr. Charles Hampton for getting me interested in the field of computational mathematics, a field I hope to pursue in the future.

To my friends, thank you for the encouragement and hospitality you provided during this process. Both my friends at home and my friends at school have provided support and acted as stress relievers.

Finally, I wish to express my love and gratitude to all my family. I would particularly like to thank my parents Jim and Rose for providing support for me through the difficulties of my senior year. The amount of confidence they have in my abilities pushed me to succeed and complete my thesis.

Contents

List of Figures

List of Algorithms

CHAPTER 1

Introduction

In the past four years, I have been exposed to many different fields of mathematics. I originally came to Wooster with the intention of becoming a psychology major with an additional concentration in mathematics. However as I began to take more mathematics courses, I became more inclined to the mathematics major. During this same period, I began to take computer science courses as well and began to realize the importance of computers to the field of mathematics. My exposure to computer programming led me to take a numerical analysis course my sophomore year. The numerical analysis course exposed me to topics that required both the use of computers and mathematics and became one of my favorite courses at Wooster. It combined theory and application and allowed me to learn several techniques for approaching a problem. The course also exposed me to *Mathematica* for the first time and I have been an avid user ever since. Although numerical analysis is a topic that can be taught in either a mathematics or computer science department, its topics are relevant for all the physical sciences. Its importance to the scientific world made me want to pursue a topic in numerical analysis for my Independent Study and possibly continue this in graduate school.

1.1 What is Numerical Analysis?

Although the book [4] has subject matter dealing with numerical linear algebra, Lloyd Trefethen has an entire discourse on the definition of numerical analysis. The majority of this section uses new information to summarize his words.

In the field of mathematics, questions like, "What is numerical analysis," do not carry much philosophical significance. Most fields of mathematics can be summarized into several sentences. However, a specific (and correct) definition of numerical analysis relies on having intimate knowledge of the subject at hand and its applications to both theoretical and real-world problems. For example, consider the two following definitions:

Definition 1.1 (American Heritage Dictionary, 2004)**.** Numerical analysis is the study of approximation techniques for solving mathematical problems, taking into account the extent of possible errors.

Definition 1.2 (Dictionary.com Unabridged)**.** Numerical analysis is the branch of mathematics dealing with methods for obtaining approximate numerical solutions of mathematical problems.

It is interesting to note the similarities between these two definitions. Combining both (1.1) and (1.2) into a simple statement could yield, "Numerical analysis is the branch of mathematics dealing with approximating solutions to mathematical problems." In a way, this definition sounds unappealing without some type of context. An extremely uninviting definition could be as follows.

Definition 1.3 ([4])**.** Numerical analysis is the study of rounding errors.

Although rounding errors are inevitable, they are in no way fundamental to numerical analysis nor do they provide any significance of the field. Nevertheless,

very few people hold any of these definitions with high regard and would not give such an unattractive definition to numerical analysis. As with many fields of mathematics, numerical analysis has its own sets of theorems, lemmas, and conjectures. A more exact definition could then be developed by studying both the theory and application of numerical analysis through textbooks. Let us consider the first few chapters of some books related to the field of numerical analysis:

(Prasad, 2005)	1 – Finite Digit Arithmetic and Errors
(Deuflhard & Hohmann, 2003)	1 – Linear Systems
	2 – Error Analysis
(Burden & Faires, 2001)	1.1 – Review of Calculus
	1.2 – Roundoff Errors and Computer Arithmetic
(Ralson & Rabinowitz, 2001)	1.1 – What is Numerical Analysis?
	1.2 – Sources of Error
	1.3 – Error Definitions and Related Matter
	1.4 – Roundoff Error
	1.5 – Computer Arithmetic
(Higham, 1996)	1 – Principles of Finite Precision Computation
	2 – Floating Point Arithmetic
(Hildebrand, 1987)	1.1 – Numerical Analysis
	1.2 – Approximation
	1.3 – Errors
	1.4 – Significant Figures
(Stoer & Bulirsch, 1980)	1.1 – Representation of Numbers
	1.2 – Roundoff Errors and Floating-Point Arithmetic
	1.3 – Error Propagation

There are several important patterns we can see from these chapter titles.

First, the words "error," and "arithmetic" are repeated numerous times. Only in one occurrence do we see the word "approximation," which is interesting since (1.1) and (1.2) both give the definition of numerical analysis as a field that "approximates solutions." With unattractive definitions like (1.1), (1.2), and (1.3) and authors of numerical analysis textbooks conveying similar notions, it is unsurprising to see why scientists in the past tended to hold numerical analysis in low esteem for many years. Furthermore, inquisitive undergraduate students may be deterred from studying numerical analysis simply based on the definitions and introductions to these textbooks. Although these definitions and chapter headings give a negative first impression to numerical analysis, it is unreasonable to say that these notions should not be introduced. With the advent of the 21st century, computers have become increasingly important in mathematics and especially numerical analysis. With the increasing usage of computers, a discussion of computer arithmetic and finite precision is required.

Putting the notions of error, arithmetic, and approximation aside, we would like to formulate a precise definition for numerical analysis. Of course no definition can be perfect, but the definition that follows is a sharp characterization.

Definition 1.4 ([4])**.** Numerical analysis is the study of algorithms for the problems of continuous mathematics.

It is important to notice that numerical analysis is defined in terms of continuous mathematics. The realm of mathematics can be roughly divided into two fields: continuous mathematics and discrete mathematics. As an analogy, consider the differences between an analog and digital watch. As time passes on an analog watch, the hour, minute, and second hands move smoothly. As the second hand traverses, the analog watch shows infinitely many times. Continuous mathematics, similar to the analog watch, is the study of structures that are

"smooth" and are infinite in scope. On the other hand, a digital watch shows a finite number of times using distinct time steps. Unlike continuous mathematics, discrete mathematics is the study of structures that are fundamentally discrete; that is, structures that do not rely on continuity. Discrete problems typically concern computer scientists. [16]

The pivotal word in (1.4) when compared to the earlier definitions is *algorithms*. When I looked at the early chapter titles of our numerical analysis textbooks I did not see this word mentioned once. Although "algorithms" is not mentioned specifically in the rest of the chapter headings in the textbooks, the subject matter is that of the analysis and study of algorithms for various mathematical problems. Numerical analysts are primarily concerned with devising new, more efficient algorithms to a certain class of problems.

Let us consider the implications of (1.4). Continuous mathematics is the study of continuous variables. It is impossible to represent continuous variables including real and complex variables on a computer. Due to this limitation rounding errors are introduced. With computer algebra systems such as *Mathematica* and Maple, attacking problems with exact arithmetic becomes possible. However, most numerical analysis problems cannot be solved using exact arithmetic. "Even if rounding errors vanished, numerical analysis would remain," says Trefethen [4]. Although approximating numbers using floating-point arithmetic is a cumbersome task, it provides benefits to numerical analysts. The goal of numerical analysis is to find algorithms that converge quickly to a solution. Rather than worry about symbolic computations, approximation allows for one to search for unknowns, rather than approximating known values.

In this sense, (1.3) is a corollary of (1.4): numerical analysis must take into account rounding errors and error associated with convergence of algorithms,

however it is not a subject that places error analysis as its main concern. Of course (1.4) is not the most precise definition of numerical analysis because other important matters must be included such as the computer architecture and the programming language. The stability and efficiency of an algorithm is of paramount importance as well. Nevertheless, as computers get faster, speed differences in algorithms become nominal and it would seem that a new definition of numerical analysis is needed. However, this is not the case since algorithms can always be designed more efficiently.

1.2 The 100-Dollar 100-Digit Challenge

The challenge originated in a problem solving course taught by Lloyd N. Trefethen for incoming D.Phil students in numerical analysis at Oxford University. Each week a problem was given with no hints and the students had to find as many digits of the answer as they could. The reason Trefethen made the challenge public was to allow other mathematicians, especially numerical analysts, to have fun solving difficult problems. When Trefethen approached SIAM in 2001 with the the idea of publishing 10 challenging scientific computation problems to the public, they enjoyed the idea so much that that they launched the contest in the January/February issue of *SIAM News*. The full text of Trefethen's challenge is as follows [18]:

> Each October, a few new graduate students arrive in Oxford to begin research for a doctorate in numerical analysis. In their first term, working in pairs, they take an informal course called the "Problem Solving Squad." Each week for six weeks, I give them a problem, stated in a sentence or two, whose answer is a single real number.

> Their mission is to compute that number to as many digits of precision as they can.
>
> Ten of these problems appear below. I would like to offer them as a challenge to the SIAM community. Can you solve them?
>
> I will give $100 to the individual or team that delivers to me the most accurate set of numerical answers to these problems before May 20, 2002. With your solutions, send in a few sentences or programs or plots so I can tell how you got them. Scoring will be simple: You get a point for each correct digit, up to ten for each problem, so the maximum score is 100 points.
>
> Fine print? You are free to get ideas and advice from friends and literature far and wide, but any team that enters the contest should have no more than half a dozen core members. Contestants must assure me that they have received no help from students at Oxford or anyone else who has already seen these problems.
>
> Hint: They're hard! If anyone gets 50 digits in total, I will be impressed. The ten magic numbers will be published in the July/August issue of *SIAM News*, together with the names of winners and strong runners-up.
> —*Nick Trefethen, Oxford University*.

After four months, the deadline arrived and entries were submitted by 94 teams from 25 countries with a total of 180 contestants. Among all the teams, 20 teams had perfect scores of 100 and 5 additional teams with scores of 99. Among the software systems that were used besides *Mathematica*, Maple, and MATLAB include: C, C++, Fortran, Java, Visual Basic, Turbo-Pascal, GMP, GSL, Octave, and Pari/GP [6].

A book by Bornemann, Laruie, Wagon, and Waldvogel [6], four individuals that participated in the challenge, provided the largest source of information used for this Independent Study. The solutions they present are a synthesis of their own solutions and those of other participants. Through their writing, I was able to comprehend the three problems that we will see in Chapters 3, 4, and 5.

In the December 2002 issue of *SIAM News* Joseph Keller of Stanford University published an interesting letter which provides motivation for this Independent Study:

> I found it surprising that no proof of the correctness of the answers was given. Omitting such proofs is the accepted procedure in scientific computing. However, in a contest for calculating precise digits, one might have hoped for more.

Proofs of correctness in numerical analysis take on many forms. An interesting numerical method that provides an algorithm and proof of correctness uses a branch of mathematics called interval analysis. In the next chapter we will briefly introduce interval analysis and its importance to numerical analysis. Using this, we will be able to provide the needed proof of correctness.

CHAPTER 2

Interval Analysis

In [13], Ramon Moore introduced modern concepts and techniques for treating intervals of real numbers as a separate system in which to do numerical computations. This allows programmers to create algorithms to produce sharp upper and lower bounds to numerical computing problems. Using this method eliminates the need for a priori or a posteriori error analysis and as a consequence produces numerical proofs of our solutions. In real world situations, accuracy of solutions is important and a method to generate valid proofs is needed because the consequences of errors can be catastrophic. As we will see later, interval analysis also allows us to solve nonlinear problems that would otherwise be difficult to solve with other numerical methods.

2.1 A Little Background

Although the benefits of using interval analysis are enormous, it is seldom used in practice. There are several reasons for this [10]:

- the slowness of some common commercial interval arithmetic packages,
- the occasional slowness of interval algorithms,

- and the unavoidable difficulty of some interval problems.

In the past, interval arithmetic was not well supported in programming languages or in hardware. For programmers, an interval data type is needed to represent intervals. For an interval $I = [a, b]$, a data type must not only store both a and b but I must also be represented as a single entity. With the introduction of operator overloading and the ability to create user defined types as seen in C++ and Fortran, programming algorithms for interval analysis became easier.

More recently, Sun Microsystems implemented interval arithmetic in its UltraSPARC III processor [12]. By implementing interval-specific hardware instructions for the basic arithmetic operations in single, double, and quadruple precision floating-point, it is possible to eliminate the existing performance deficit in the time required to compute interval versus floating-point expressions. For example, the M77 compiler developed at the University of Minnesota allowed interval arithmetic computations to be roughly five times slower than ordinary floating-point arithmetic [10].

Popular computer algebra systems such as MATLAB, Maple, and *Mathematica* have introduced packages to support interval methods. Specifically, *Mathematica* includes an implementation of basic interval arithmetic on real numbers using the `Interval[]` command.

When measuring time complexity of algorithms, generally it is described in floating-point operations per second (FLOPS). People who are unfamiliar with interval analysis commonly want a basis for comparison and want an estimate of the FLOPS. The effort of implementing interval-specific operations in software and hardware pales in comparison to the decades of research spent on implementing floating-point operations. Therefore, it would be unreasonable to compare the

speed of interval based algorithms to those of floating-point based on how long it takes to compute a solution within a specified bound.

To eliminate this problem, a benchmarking strategy was formulated by Gustafson and Walster to eliminate the ambiguity of FLOPS. In [10] their results are summarized. These give a method to benchmark the performance for *any* standard algorithm on *any* computing system. To illustrate the problem with older benchmarking strategies, we summarize the ideas following from [10]. First consider a problem in which the input is a degenerate (zero width) interval (or intervals) and we wish to perform some sort of sensitivity analysis by bounding the effect of rounding errors. To do so, one would need to compute the time and effort it takes to run both the interval and noninterval programs and to measure the amount of time and effort it takes for the noninterval program to perform rigorous error analysis. This proposed strategy seems like a good method in illustrating the differences between interval and noninterval methods and their efficiency in producing error bounds. However, now consider a problem in which the input is a nondegenerate interval (or intervals). Using the same programs, the interval method will find a set of solutions to our problem whereas the noninterval method may have difficulty, especially if one wanted to do sensitivity analysis.

Depending on the problem, the operation counts in algorithms using interval arithmetic vary from its noninterval counterpart. For example, to get narrow bounds on the solution to a system of linear algebraic equations, interval methods require approximately six times as many operations. On the other hand, using Newton's method to bound a polynomial root to a given accuracy using interval methods uses approximately the same number of FLOPS as the standard arithmetic approach. There are also situations where interval methods perform better. When solving for all the roots of a polynomial, Newton's method gives us a

way to find *one* root, where there can be as many as n roots. In this case, an interval method would be faster than the noninterval one because the noninterval method must use some sort of deflation; that is, if a function $f(x)$ has root r_1, then it is natural to consider the simpler polynomial $g(x) = f(x)/(x - r_1)$ and use Newton's method on that. Interval methods do not suffer from deflation.

2.2 Interval Numbers

We delve into the realm of interval analysis by first considering extensions of other number systems using ordered pairs. For example, the rational numbers can be expressed as a set of ordered pairs of integers where

$$\mathbb{Q} := \Big\{(p, q) : p, q \in \mathbb{Z}\Big\}.$$

Here, the rational number $p/q \in \mathbb{Q}$ can be denoted as (p, q) using this extension. Similarly, the complex numbers can expressed as an ordered pair of real numbers where the real part is the first coordinate and the imaginary part is the second coordinate. We use this idea to extend a real number to that of an interval number as follows.

Definition 2.1. An *interval number* is an ordered pair of real numbers, $[a, b]$ with $a \leq b$. We define the set of real numbers $[a, b]$ to be $[a, b] := \{x \in \mathbb{R} : a \leq x \leq b\}$. An interval of the form $[a, a]$ is a *degenerate interval* where $[a, a] = a$.

When dealing with intervals, it is convenient to define some notation to aid in stating various theorems. Many of these definitions and identities can be found in [10] or [13]. If x denotes a real quantity, then let the capital variant X denote the interval quantity. If the real quantity is denoted by a capital letter then we denote

the corresponding interval quantity by attaching a superscript "I." For example a real matrix denoted by A has a corresponding matrix A^I which has intervals as entries.

Let $\mathscr{I}$ denote the set of closed real intervals and let $I \in \mathscr{I}$ with $I = [a, b]$. The notation $\mathscr{I}_I$ indicates the set of intervals which are contained in I,

$$\mathscr{I}_I := \Big\{[x, y] : a \le x \le y \le b\Big\}.$$

In other words, $\mathscr{I}_I$ is the set of "subintervals" of I. An element of $\mathscr{I} \times \mathscr{I} \times \cdots \times \mathscr{I}$ is an *interval vector*. An interval is said to be *positive* if $a > 0$ and *nonnegative* if $a \ge 0$. It is said to be *negative* if $b < 0$ and *nonpositive* if $b \le 0$. Two intervals $[a, b]$ and $[c, d]$ are *equal* if and only if $a = c$ and $b = d$. We define the following operations:

(i) The width of the interval I is

$$w\Big([a, b]\Big) = b - a.$$

(ii) The magnitude of an interval is

$$\Big|[a, b]\Big| = \max\Big(|a|, |b|\Big).$$

(iii) The midpoint of an interval is

$$m\Big([a, b]\Big) = \frac{a + b}{2}.$$

We define a partial ordering of the elements of $\mathscr{I}$ by

$$[a, b] < [c, d] \qquad \text{if and only if } b < c.$$

2.3 Interval Arithmetic: Notations & Relations

We will begin this section by introducing the arithmetic operations on elements of $\mathscr{I}$. We first make the assumption that these operations are computed with infinite-precision. Later we will account for round-off error in these computations, which is important in our study of high-precision arithmetic.

Given $[a,b],[c,d] \in \mathscr{I}$ we define arithmetic operations on intervals by

$$[a,b] * [c,d] = \{x * y : a \le x \le b, c \le y \le d\} \text{ where } * \in \{+,-,\cdot,/\}. \tag{2.1}$$

Our definition of interval arithmetic in (2.1) is set-theoretic and affirms the fact that the sum, difference, product, or quotient of two intervals is just the set of sums, differences, products, or quotients of the endpoints of the interval. An equivalent set of definitions for the interval operations are as follows.

$$\begin{aligned} [a,b] + [c,d] &= [a+c, b+d], \\ [a,b] - [c,d] &= [a-d, b-c], \\ [a,b] \cdot [c,d] &= \Big[\min\{ac,ad,bc,bd\}, \max\{ac,ad,bc,bd\}\Big], \\ [a,b]/[c,d] &= [a,b] \cdot [1/d, 1/c] \qquad \text{if } 0 \notin [c,d]. \end{aligned} \tag{2.2}$$

If $0 \in [c,d]$, we do not define $[a,b]/[c,d]$.

For a nonnegative integer n we can assign exponents to intervals by defining

$$I^n = \begin{cases} [1,1] & \text{if } n = 0, \\ [a^n, b^n] & \text{if } a \ge 0 \text{ or if } n \text{ is odd}, \\ [b^n, a^n] & \text{if } b \le 0 \text{ and } n \text{ is even}, \\ \Big[0, \max\{a^n, b^n\}\Big] & \text{if } a \le 0 \le b \text{ and } n > 0 \text{ is even}. \end{cases} \tag{2.3}$$

Using degenerate intervals we can see that these operations are the arithmetic operations on the real numbers. It follows easily from (2.2) that interval addition and interval multiplication are both associative and commutative; that is, given $I, J, K \in \mathscr{I}$ then

$$\begin{aligned} I + (J + K) &= (I + J) + K, \\ I \cdot (J \cdot K) &= (I \cdot J) \cdot K, \\ I + J &= J + I, \\ I \cdot J &= J \cdot I. \end{aligned} \tag{2.4}$$

The real numbers $0 = [0, 0]$ and $1 = [1, 1]$ are identities for interval addition and multiplication. For $I \in \mathscr{I}$ we have

$$\begin{aligned} 0 + I = I + 0 = I, \\ 1 \cdot I = I \cdot 1 = I. \end{aligned}$$

We note that the distributive law does not always hold for interval arithmetic. For example

$$\begin{aligned} [1, 2] \cdot \Big([1, 2] - [1, 2]\Big) = [1, 2] \cdot [-1, 1] = [-2, 2], \\ [1, 2][1, 2] - [1, 2][1, 2] = [1, 4] - [1, 4] = [-3, -3]. \end{aligned}$$

Since an interval is also a set of real numbers, we consequently have the following relation amongst interval addition and multiplication for intervals $I, J, K \in \mathscr{I}$:

$$I \cdot (J + K) \subset I \cdot J + I \cdot K. \tag{2.5}$$

We refer to the property of interval arithmetic in (2.5) as *subdistributivity*.

2.4 Rounded-Interval Arithmetic

Recall that in IEEE-754 arithmetic, the range of numbers is limited by the maximum word size of the current instruction set. Consequently, not every real number can be expressed using extended-precision arithmetic on a given computer. To reconcile this problem for a given interval $I = [a, b]$, we round a down to the largest machine-representable number that is less than a and round b up to the smallest machine-representable number that is greater than b. This new interval $[a', b'] \supseteq [a, b]$ and describes the process of *outward rounding*.

Directed rounding is rounding that is specified to be up or down depending on the situation; that is, rather than rounding to the next machine-representable number, we may round to a specified number within some bound.

Now consider what happens when we subtract the interval $I = [a, b]$ from itself. From (2.2) we know

$$[a, b] - [a, b] = [a - b, b - a] \neq [a - a, b - b] = [0, 0].$$

As seen above, you might expect our subtraction operation to yield $[0, 0]$, but it does not. This is due to the fact that each interval is treated as a different variable. Thus, $I - I$ is treated as $I - J$ where I and J are numerically equal but independent from one another. This phenomenon is called *dependence* and causes widening of intervals in complicated expressions. For example, consider the expression $[-1, 2]^2$. Using (2.2)

$$[-1, 2] \cdot [-1, 2] = \Big[\min\{-2, 1, 4\}, \max\{-2, 1, 4\}\Big] = [-2, 4].$$

On the other hand, using (2.3) we have

$$[-1,2]^2 = \Big[0, \max\{1,4\}\Big] = [0,4].$$

Thus using (2.3) for the n-th power of an interval minimizes dependence and creates sharper bounds. Another method for creating sharper bounds on intervals is through cancellation or reduction of the number of occurrences of an interval variable [13]. For example, we can rewrite the expression $I/(I-2)$ as

$$\frac{I}{I-2} = \frac{I-2}{I-2} + \frac{2}{I-2} = 1 + \frac{2}{I-2}.$$

The resulting expression has fewer occurrences of the variable I and would yield narrower intervals. In fact, since I appears only once we know that the interval would be the exact range of values.

We may also make use of subdistributivity (2.5) to reduce interval widths. For example, when dealing with polynomials the nested form of a polynomial expression usually provides a sharper bound on the range. Given intervals $I_0, I_1, \ldots, I_n$ and an interval quantity X then

$$\Big(\cdots\Big((I_n)X + I_{n-1}\Big)X + \cdots I_1\Big)X + I_0 \subset I_n \cdot X \cdot X \cdots X + \cdots + I_2 \cdot X \cdot X + I_1 \cdot X + I_0.$$

For polynomials there are infinitely many combinations of parentheses and elements that define the same polynomial. For example

$$f(X) = X(X-1) = X^2 - X = -6 + (X+2)\Big(5 - (X+2)\Big) = \cdots$$

Then for a given interval X, using the nested form of $f(X)$ would produce a narrower interval than $f(X)$ in its expanded form.

2.5 Functions of Intervals

When dealing with interval numbers, we must describe the functions on which they act. A superscript "I" on the symbol for a function indicates that it is an *interval function*. An interval function is an interval-valued function of one or more interval arguments. An interval function maps the value of one or more intervals onto an interval with $f^{\mathrm{I}}\colon \mathscr{I}_I \to \mathscr{I}$. Consider a real-valued function f with variables $x_1, x_2, \ldots, x_n$ and its corresponding interval function f^{I} of intervals $X_1, X_2, \ldots, X_n$. The interval function f^{I} is said to be an *interval extension* of f if $f^{\mathrm{I}}(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_n)$ for any values of the argument variables; that is, if each of the arguments of f^{I} are degenerate intervals, then $f^{\mathrm{I}}(x_1, x_2, \ldots, x_n)$ is a degenerate interval equal to $f(x_1, x_2, \ldots, x_n)$ [10].

In Section 2.4 we mentioned the concept of rounded-interval arithmetic. In practice, we rarely have the capabilities of performing exact-interval arithmetic. The definition of interval extension presupposes that we are using exact-interval arithmetic. Therefore, for an interval function (or interval extension) f^{I} we are normally able to compute an interval enclosure F with

$$f^{\mathrm{I}}(x_1, x_2, \ldots, x_n) \in F(x_1, x_2, \ldots, x_n).$$

We say an interval function f^{I} is *inclusion monotonic* if $X_i \subset Y_i$ $(i = 1, 2, \ldots, n)$ implies $f^{\mathrm{I}}(X_1, X_2, \ldots, X_n) \subset f^{\mathrm{I}}(Y_1, Y_2, \ldots, Y_n)$. It follows from (2.2) that finite interval arithmetic is inclusion monotonic [13]. Thus for $I, J, K \in \mathscr{I}$ with $I \subset K$ and

$J \subset L$, then

$$I + J \subset K + L$$

$$I - J \subset K - L$$

$$I \cdot J \subset K \cdot L$$

$$I/J \subset K/L \qquad \text{if } 0 \notin L.$$

From this set of relations and the transitivity of inclusion relations we arrive at the following theorem which will aid in the development of our algorithms later.

Theorem 2.2 ([13, Theorem 3.1]). *If $F(X_1, X_2, \ldots, X_n)$ is a rational expression in the interval variables $X_1, X_2, \ldots, X_n$, i.e., a finite combination of $X_1, X_2, \ldots, X_n$ and a finite set of constant intervals with interval arithmetic operations, then*

$$X_1' \subset X_1, \ldots, X_n' \subset X_n \Rightarrow F(X_1', \ldots, X_n') \subset F(X_1, \ldots, X_n)$$

for every set of interval numbers $X_1, \ldots, X_n$ for which the interval arithmetic operations on F are defined.

If $X_1', \ldots, X_n'$ are real numbers, then the value of $F(X_1', \ldots, X_n')$ will be a real number contained in its interval counterpart $F(X_1, \ldots, X_n)$. Therefore, we are able to bound the range of values of a real rational function by performing a finite number of interval arithmetic operations on its corresponding interval function.

As we will see later, an interval enclosure $F(X_1, \ldots, X_n)$ will not produce sharp bounds on the range if a variable X_i appears more than once or appears with a degree greater than one in F. If each interval variable X_i in $F(X_1, \ldots, X_n)$ appears only once and has a degree of one, then $F(X_1, \ldots, X_n)$ will compute the actual

range of values of the function f for each $x_i \in X_i$. Thus the interval enclosure F will be exactly the same as the set of all combinations of real inputs for f; that is,

$$F(X_1, \ldots, X_n) = \Big\{ f(x_1, \ldots, x_n) : x_i \in X_i \text{ for } i = 1, \ldots, n \Big\}.$$

2.6 Importance

Interval methods have many applications to real world problems since they enable one to find solutions to problems that cannot be solved by noninterval methods. One example is the global optimization problem. Although interval algorithms usually run slow, a price is paid to get a reliable algorithm that will not only guarantee error bounds that a noninterval algorithm will not provide, but in some cases provide a proof of uniqueness and existence. Interval methods are also more reliable. For example, the interval Newton method always converges [10]. Another benefit of using interval methods is that stopping criteria can easily be set. A natural stopping criteria would be to stop the algorithm if the interval bounds are no longer decreasing or if the interval bounds are within a required tolerance. To find a stopping criteria for noninterval algorithms is difficult and may be hard to implement.

CHAPTER 3

One Photon, Infinite Mirrors

Problem. A photon moving at speed 1 in the x-y plane starts at time $t = 0$ at $(x, y) = (1/2, 1/10)$ heading due east. Around every integer lattice point (i, j) in the plane, a circular mirror of radius $1/3$ has been erected. How far from $(0, 0)$ is the photon at $t = 10$?

At first glance this would seem like a problem destined for physicists rather than mathematicians. If this problem modeled a real world situation, then we would have to modify the statement. First, a photon is an elementary particle that is a fundamental unit of electromagnetic radiation that travels at the speed of light in empty space [3]. It is a bit out of the ordinary for a photon to be traveling at speed 1. When dealing with reflections off mirrors, one must worry about refraction. In a mirror, a layer of glass sits atop a reflective surface. When light enters the glass it passes from one medium to another and therefore its phase velocity changes as well as its direction. To maintain high accuracy computations, the refractive index of the glass and the space the photon travels through must be considered. Also, the existence of perfectly circular mirrors is absurd.

Putting these considerations aside, assume the magnitude of the photon's velocity does not change and that the mirrors offer a perfect reflection; that is, the

angle of incidence equals the angle of reflection. To determine the final position of the photon, it is necessary to write a program that determines the entire path of the photon. It is important to notice that using typical machine-precision (16 digits) will not yield 10 correct digits of our answer [6]. Knowing that machine-precision is a limiting factor in our program, error will enter our computations quickly because 1/10 is not finitely representable in binary. This also means that software or programming languages that use machine-precision by default (such as C) will not be able to solve this problem without the use of external packages. Fortunately, *Mathematica* can switch from machine-precision to arbitrary-precision arithmetic as necessary. In the following sections, I will be presenting an elegant solution due to Fred Simons that with the aid of interval arithmetic, will allow us a precise answer.

3.1 Estimating the Photon's Path

Since we are given the photon's initial position and its initial velocity and that $t = 10$, the problem at hand can be solved by following this pseudoalgorithm:

while $0 < t \leq 10$
  Find the next mirror of intersection.
  Update the photon's position.
  Update the photon's velocity
  Reduce the travel time of the photon from t.
end while

Using a simple geometric argument we can easily determine (by trial-and-error) which mirror the photon's path will intersect. Consider a ray emanating from a point P in the photon's path in the direction of the unit vector v. If we were to divide $\mathbb{R}^2$ into unit squares centered on the the lattice points in $\mathbb{Z}^2$, then these

squares will divide our ray into segments as seen in Figure 3.1. It is a triviality to find the square the ray is in by rounding its coordinates.

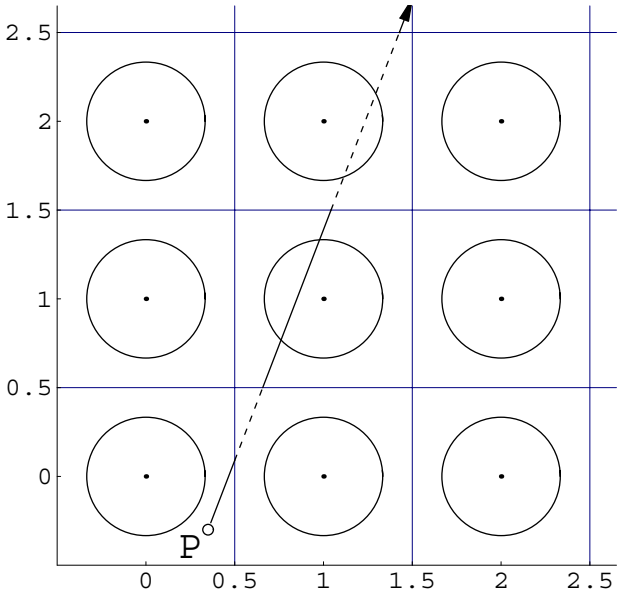


Figure 3.1: Using a lattice gives an efficient way to organize a search to find the next mirror the photon's path will intersect.

Let us take a look at a ray that lies within one of these unit squares. Clearly, this ray will not have length longer than $\sqrt{2}$, but what can we say about its minimum length? For the ray to hit the mirror inside a given square, the shortest length it can have is if it is tangent to the mirror as seen in Figure 3.2. However, since we have a unit square we can easily find the length of this tangent ray to be $\sqrt{2} - 2/3$.

We could use $\sqrt{2} - 2/3$ in our future program, but it is simpler to give a rational lower bound. Fred Simons [1] used $2/3$ as his bound, however $2/3$ is not finitely representable in binary. As an alternative, we could use a rational number that is finitely representable in binary, but through experimentation I found that using such a number does not provide faster computations.

This observation gives us an easy way to determine whether a ray will intersect a mirror. We begin by assuming that the path from P will intersect

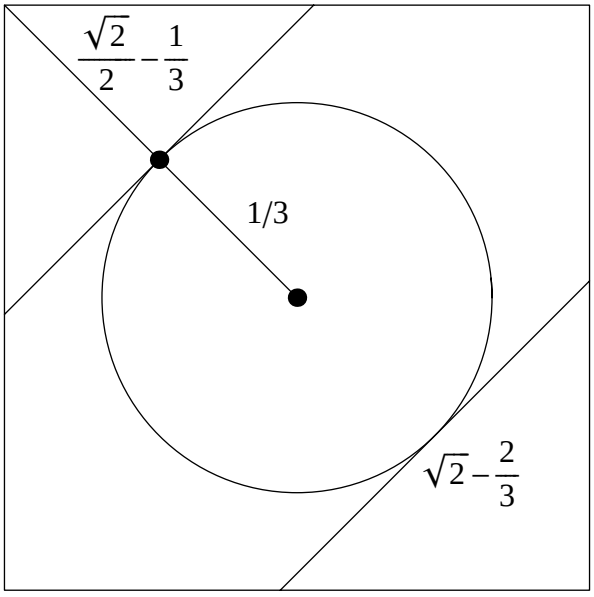


Figure 3.2: If a ray strikes a mirror within a given square it must have length at least $\sqrt{2} - 2/3$.

the mirror corresponding to $P + (2/3)v$. Until we find an intersection, we can consider as long as necessary the sequence of mirrors corresponding to $P + 2 \cdot (2/3)v, P + 3 \cdot (2/3)v, P + 4 \cdot (2/3)v, \ldots$ As soon as we find an intersection; that is, if the segment inside the square has length at least $2/3$, then we replace P by the point of intersection and update its velocity appropriately.

Finding the length of the segment inside of an ambient square is computationally intensive and does not provide an easy way to determine whether the mirror is hit or not hit. If we let m be the center of the circle corresponding to P, then choose the smallest positive root t of the quadratic equation

$$(P + tv - m) \cdot (P + tv - m) = 1/9 \tag{3.1}$$

to find the time it takes for the photon to reach a mirror of intersection. Intuitively, since $P + tv - m$ is a vector and the distance from the intersection point Q to the

center of the mirror is 1/3, we are really solving for $|P + tv - m| = 1/3$. If the photon hits the mirror, it will hit once when it enters and once when it leaves, which explains the quadratic nature. If there is no intersection, then (3.1) will not have any real roots and thus we know that the mirror corresponding to m will not get hit. In this case, we increase P by $(2/3)v$ and then try again.

Now assume that s is the smallest positive root of (3.1). We can find the intersection point Q by just letting $Q = P + sv$, but now we want to find the reflected velocity. Assume for a moment that the center of the mirror is the origin and $Q = (a, b)$. We want to find a linear transformation H that will take our photon's velocity and give us its reflected velocity. The transformation matrix H sends $(-a, -b)$ to (a, b) because it reverses direction, and fixes $(-b, a)$ so the angle of incidence equal the angle of reflection. Therefore, we are trying to find H such that

$$H \cdot \begin{pmatrix} -a & -b \\ -b & a \end{pmatrix} = \begin{pmatrix} a & -b \\ b & a \end{pmatrix}.$$

Using the fact that $a^2 + b^2 = 1/9$, we can easily solve for H:

$$H = 9 \begin{pmatrix} b^2 - a^2 & -2ab \\ -2ab & a^2 - b^2 \end{pmatrix}.$$

However, since we assumed that our circle was centered at the origin, we must have that $(a, b) = Q - m$.

We will now present Algorithm 3.1 which makes use of the above information to get at least ten digits of the answer.

Algorithm 3.1 (Finding the Path of a Reflected Photon)

Assumptions: The photon has unit speed and the mirrors have radius 1/3.

Input: An initial position P, a direction vector v, and a maximum time $t_{\max}$.

Output: The path of the particle from time 0 to $t_{\max}$ where `path` is the set of points of reflection together with the position of P at time 0 and $t_{\max}$. Additionally, return the distance of the photon from the origin at time $t_{\max}$.

Notation: t_{rem} is the amount of time remaining, m is the center of the circle in the square containing the photon, and s is the time the photon strikes the circle, measured from the time of the preceding reflection. For a point Q, H_Q represents the matrix

$$H_Q = 9\begin{pmatrix} b^2 - a^2 & -2ab \\ -2ab & a^2 - b^2 \end{pmatrix}. \tag{3.2}$$

$t_{\text{rem}} = t_{\max}$

`path` $= \{P\}$

while $t_{\text{rem}} > 0$

  // Assume for this iteration that m is the midpoint of the mirror the photon's path intersects with.

  $m = \text{round}(P + 2v/3)$

  // If $s \neq \infty$ then s is equal to the time it takes for the photon to reach the intersection point with the mirror, otherwise the photon does not intersect the mirror.

  $s =$ smallest positive root of $(P + tv - m) \cdot (P + tv - m) = 1/9$

  if $s < t_{\text{rem}}$

    // In this case, the photon intersects a mirror so P is now set to be the point at which it intersects with the mirror.

    $P = P + sv$

    $v = H_{P-m}v$

```
    else
      // The photon does not intersect the mirror centered at m because either (1)
          time has run out and the photon stops in mid-flight or (2) an incorrect
          mirror was chosen.
      s = min(t_rem, 2/3)
      // If (1) then P is the point at time 10. If (2) then P is the point that lies in the
          unit square surrounding the next mirror in its path.
      P = P + sv
    end if
  end while
  // Update time and path
  t_rem = t_rem − s
  Append P to path
  return path and the distance of the photon from the origin
```

Appendix A contains code for Algorithm 3.1. In *Mathematica*, we may force the use of software-arithmetic (high-precision arithmetic) by invoking the `N` command on our initial inputs. By definition, `N[expr, n]` attempts to give a result with n-digit precision. For our case, if we let `P = N[{1/2,1/10},36]`, then *Mathematica* will attempt to use 36 digits of precision using the technique of significance arithmetic. Significance arithmetic not only keeps track of numerical results, but also uses error propagation to track their accuracy. In this way, numerical computations can return in the end a numerical quantity together with its estimated (or worst-case) uncertainty by using calculus-based heuristics. Using 44-digit precision for the initial conditions, the result is believed to have 11.336 correct digits:

```
0.995262919443354160220636126 5 `11.336
```

It seems that by initially using 35-digit values we can get at least 10 digits of our answer but *Mathematica* believes only 4.789 digits are correct. By increasing the software-precision, we can easily get higher precision answers. However, this is a dangerous route, especially since we have no way of verifying that our answer is correct.

Unlike other software packages, *Mathematica*'s approach to high-precision arithmetic does not allow one to use fixed-precision for numerical computations. Although this implementation is good because it gives an error estimate using significance arithmetic, we may use fixed-precision to better understand error propagation. Using *Mathematica*'s software-precision, we can see problems using too low of a precision as in Figure 3.3.

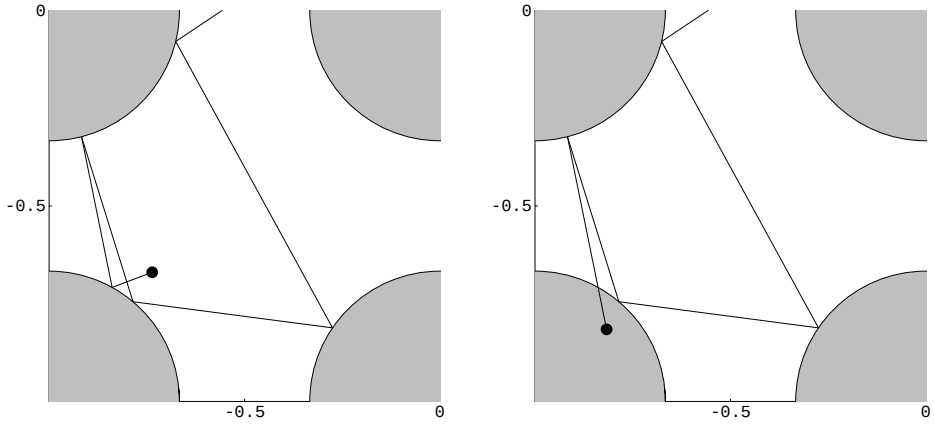


Figure 3.3: Using software-precision, our algorithm is unable to correctly determine the final position of the photon.

Using fixed-precision we are able to determine the actual number of correct digits our algorithm will give us as seen in Figure 3.4. It is safe to conclude that the 10 digits of the correct distance at time $t = 10$ are 0.9952629194. From Figure 3.4 we can estimate that using a precision d we can get about $d - 11$ correct digits. Because our photon reflects 17 times in our stated problem, the precision loss

is about two-thirds of a digit per reflection [6]. Using fixed-precision, we are also able to compare trajectories. Figure 3.5 illustrates that at time 20, using machine-precision may not give an accurate trajectory.

Precision	Computed Distance	Number of Correct Digits
5	3.5923	0
6	0.86569	0
7	2.386914	0
8	0.7815589	0
9	1.74705711	0
10	0.584314018	0
11	0.8272280639	0
12	1.01093541331	0
13	0.993717133054	2
14	0.9952212862076	4
15	0.99525662897655	4
16	0.995262591079377	6
17	0.9952631169165511	5
18	0.99526292565663264	7
19	0.995262918048682059	8
20	0.9952629191087191889	9
21	0.99526291946156616033	10
22	0.995262919441599585251	11
23	0.99526291944435253187805	12
24	0.995262919443369789952 92	13
25	0.9952629194433532618239 51	14
26	0.9952629194433543857853841	15
27	0.995262919443354157814022 73	16
28	0.995262919443354160804882481	19
29	0.9952629194433541607720201289	18
30	0.99526291944335416087109016456	19

Figure 3.4: Results of a fixed-precision approach using precision 5 through 30 [6].

3.2 Reliable Reflections

Although the answer we obtained in Section 3.1 seems correct, we have no way of verifying our solution and must rely on *Mathematica*'s significance arithmetic to determine our precision. To create an algorithm that has a proof of correctness

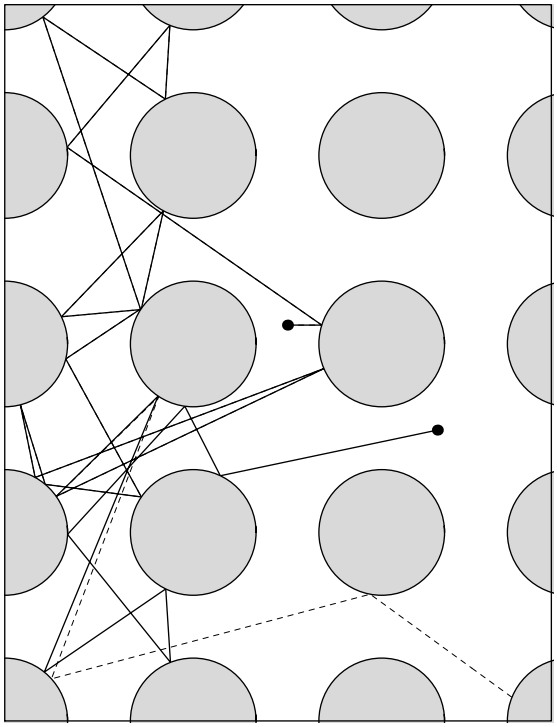


Figure 3.5: At time 20, the machine-precision trajectory (solid) is at a very different point than one following the path (dashed) computed with enough precision to guarantee correctness [6].

we will use interval arithmetic. The interval algorithm we will present is a naive approach because we are simply transforming Algorithm 3.1 by putting a small interval around each of our inputs and replacing each operation by its corresponding interval version. This will give us an interval enclosure of the photon's position at time 10. If the interval is not small enough or we do not have the required precision, then just restart the algorithm and reduce the interval enclosure by a factor of 10 about our initial conditions. Since we are no longer using real arithmetic, we need to consider the following:

(i) Because we are using small interval enclosures, we need to have a high enough working precision to handle the precision required for our intervals. Experimentally, we find that using $s + 2$ digits of working precision when the

initial conditions have radius 10^{-s} appears to be adequate [6]. In *Mathematica*, the working precision can be set by using `N[expr,s + 2]` to give `expr` a working precision of $s + 2$. This also brings up another subtlety: if any of the intermediate results have a precision less than the desired precision, then we must start with a smaller enclosing interval. We must check whether the precision lost when solving the quadratic equation does not build up too much, which can be done by checking the precision of the results using `Precision`.

(ii) When solving the quadratic equation in interval form, one must check that the solution has no expression of the form $\sqrt{[\text{negative value}, \text{positive value}]}$. If this happens, then the resulting solution would not be uniquely determined by the input interval. Therefore, just exit the loop and decrease the size of the initial intervals by a factor of 10.

(iii) When we check to see whether there is time remaining for the photon to strike another mirror, we need to check that the travel time along the current ray is strictly less than the time remaining. Doing so precludes the situation where the travel time overlaps the interval representing the time remaining.

(iv) Since we are using intervals, the photon may end up at time $10 + \delta$ rather than at time 10. However, since the photon travels at unit speed, we may turn this time uncertainty into space uncertainty and thus get an interval containing the answer [6].

We will now present an algorithm that uses similar techniques as Algorithm 3.1 and takes into account the comments listed above.

Algorithm 3.2 (Finding the Path of a Reflected Photon Using Intervals)

Assumptions: The photon has unit speed and the mirrors have radius 1/3. An interval arithmetic package is available and has implemented round(*interval*).

Input: An initial position p, a direction vector v, a maximum time $t_{\max}$, and an absolute error bound ε on the final position.

Output: The path of the particle from time 0 to $t_{\max}$ where `path` is the set of points of reflection together with the position of p at time 0 and $t_{\max}$. Additionally, return the distance of the photon from the origin at time $t_{\max}$. The position of the last point is guaranteed to have absolute error less than ε.

Notation: We have lower-case letters denote real numbers, upper-case letters denote intervals, and script letters for sets of intervals. For an interval Q, H_Q is the same as (3.2). The intervals P, V, M, T_{rem} represent the position, direction, mirror-center, and time remaining, respectively. We let s be the value for the radius of interval precision.

Functions: For an interval $X = [a, b]$, $\min(X)$ and $\max(X)$ represents $\min(a, b)$ and $\max(a, b)$, respectively, $\operatorname{diam}(X) = \max(X) - \min(X)$, and $\operatorname{mid}(X) = \Big(\min(X) + \max(X)\Big)/2$. For a set of intervals $\mathcal{X} = \{X_i\}$, then $\min(\mathcal{X}) = \Big[\min_i\Big(\min(X_i)\Big), \min_i\Big(\max(X_i)\Big)\Big]$ and $\operatorname{diam}(\mathcal{X}) = \max_i\Big(\operatorname{diam}(X_i)\Big)$. For a (interval) vector w, w_x and w_y represent the x and y components, respectively.

1 $T_{\text{rem}} = [t_{\max}, t_{\max}]$

2 `path` $= \{p\}$

3 $s = \lfloor -\log_{10} \varepsilon \rfloor$

4 `error` $= \infty$

5 $\delta = 10^{-s}$

6 `wp` $= s + 2$

```
while error > ε
  Set the working precision to wp digits.
  while min(T_rem) > 0
    // Create small intervals around p and v.
    P = ([p_x − δ, p_x + δ, p_y − δ, p_y + δ])
    V = ([v_x − δ, v_x + δ, v_y − δ, v_y + δ])
    M = round(P + 2V/3)
    Use interval arithmetic to determine S, the set of interval solutions to the
        quadratic equation (P + tV − M) · (P + tV − M) = 1/9
    if S contains an expression of the form √[a, b] with a < 0 < b
      exit inner while loop
    end if
    Let T be those solutions of the form [a, b] with a ≥ 0
    // T = [∞, ∞] if T = ∅
    T = min(T)
    if T ≤ T_rem
      // The photon intersects the mirror.
      P = P + TV
      V = H_{P−M} V
      T_rem = T_rem − T
    else if T > T_rem and T_rem ≥ 2/3
      // The photon does not intersect the mirror in question.
      T_rem = T_rem − 2/3
      P = P + 2/3V
    else if T > T_rem and T_rem < 2/3
      // Time has run out and the photon stops moving.
      P = P + T_rem V
      T_rem = 0
```

34 **else**

35 The intervals are not comparable so **exit** the inner **while** loop

36 **end if**

37 Append mid(P) to `path`

38 **if** the precision of any of T, P, V, T_{rem} is less than $-\log_{10}\varepsilon$

39 **exit** inner **while** loop

40 **end if**

41 **end while**

42 `error` = $\text{diam}\left(\left\{P_x + [-\max(|T_{\text{rem}}|), \max(|T_{\text{rem}}|)], P_y + [-\max(|T_{\text{rem}}|), \max(|T_{\text{rem}}|)]\right\}\right)$

43 $s = s + 1$

44 $\delta = 10^{-s}$

45 `wp` = $s + 2$

46 **end while**

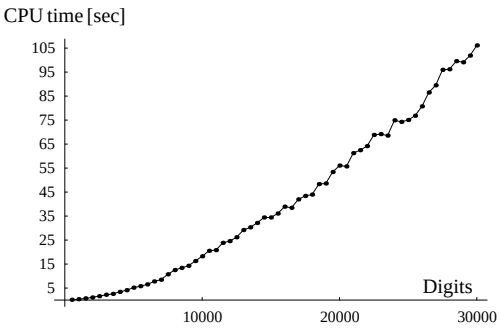
47 **return** `path` and the distance of the photon from the origin

Appendix A contains code for Algorithm 3.2. When we run the algorithm for the first time, we do not know what initial value for s we should use to guarantee a certain accuracy goal. Steps 43 - 45 of Algorithm 3.2 are included so if we have not reached our accuracy goal, then our working precision is increased by a factor of 10. We find that for $s = 40$ we can achieve 10^{-12} accuracy. In the future, we may choose a value of $s > 40$ so the algorithm will not have to decrease the size of the intervals and thus decrease running time. Through experimentation, we can find that an interval radius of 10^{-d+28} for the initial conditions is sufficient to get d digits of the answer [6].

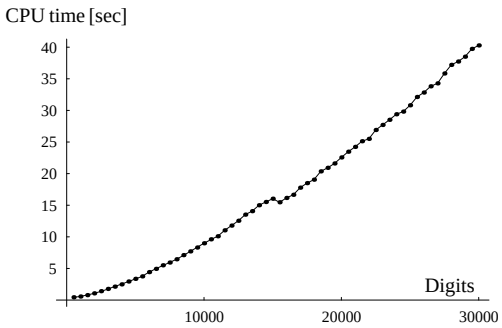
Some additional investigations of this problem would be to consider what happens when the radius of the mirrors is changed and remains constant or if each mirror is given a random radius. It may also be interesting to create an algorithm

to model the reflections of a photon with non-unit speed. In this case, we would have difficulties in determining the photon's final position using space uncertainty. As stated at the beginning of this chapter, knowing the photon's position to such tolerances is absurd and impossible unless given such a simple problem statement.

We have shown that both Algorithm 3.1 and Algorithm 3.2 can both be used to solve the given problem. Using Algorithm 3.2 we are able to eliminate the uncertainty of heuristic error estimates. However, it is strange that the interval algorithm has a faster run time than its noninterval counterpart which can be seen in Figure 3.6. Unfortunately I was unable to duplicate the timing results shown in [6]. This may be due to the fact that the computer I used had a slower processor and less memory available.



(a) Time needed by Algorithm 3.1 to get d digits of the answer.



(b) Time needed by Algorithm 3.2 to get d digits of the answer.

Figure 3.6: Timing results for Algorithm 3.1 and 3.2.

The problem becomes more difficult as the travel times increase. By using Algorithm 3.2 it is possible to find the position of the photon at any specified time. At time 100, starting with a radius of 10^{-265} we are able to get an answer correct to 13 digits. However, for time 2000, a radius of 10^{-5460} is required to get a true ending position for the photon [6]. The trajectory shown for time 2000 is shown in Figure 3.7 and has some interesting characteristics. You would believe that the

photon would exhibit a random walk governed by brownian motion, however due to the fixed positions and radii of the mirrors, the photon takes long steps in both the horizontal and vertical directions. This follows from the fact that the photon cannot travel for a long distance without hitting a mirror unless it travels in a purely horizontal or vertical direction.

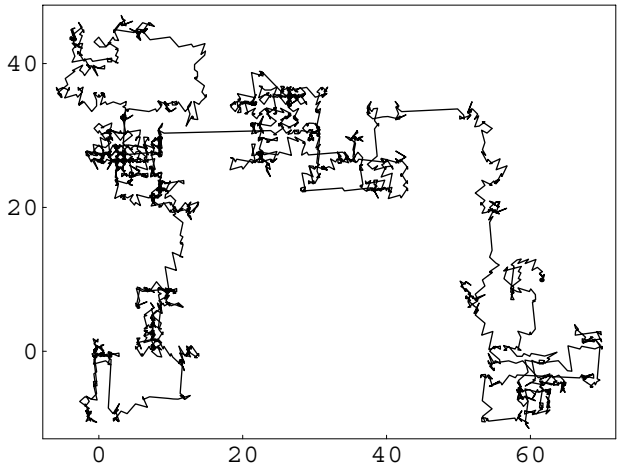


Figure 3.7: Using Algorithm 3.2 and intervals with diameter 10^{-5460} we can find the true path of the photon for time 2000.

CHAPTER 4

Hidden Complexity

Problem. What is the global minimum of the function

$$e^{\sin(50x)} + \sin(60e^y) + \sin(70\sin x) + \sin\Big(\sin(80y)\Big) - \sin\Big(10(x+y)\Big) + \frac{x^2+y^2}{4}?$$

We first let $f(x, y)$ denote the given function in our problem statement. If we take a look at a global plot of $f(x, y)$ as in Figure 4.1 we can see it resembles a paraboloid of one sheet. Since sine is bounded on $[-1, 1]$, we have that the first five summands lie in the intervals $[1/e, e]$, $[-1, 1]$, $[-1, 1]$, $[-\sin 1, \sin 1]$, and $[-1, 1]$, respectively. However, the $(x^2 + y^2)/4$ is unbounded as x and y approach infinity so it dominates the other summands and the result is a quadratic surface which is centered around the origin.

Using this knowledge we may take the largest enclosing region amongst all the other summands and believe that our global minimum lies within the square $[-1, 1] \times [-1, 1]$. Although finding the minimum of a paraboloid is an easy task, we can see that the trigonometric elements introduce added complexity as seen in Figure 4.2.

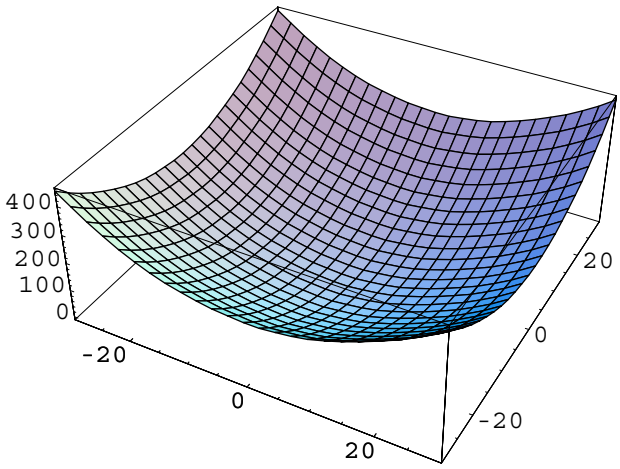
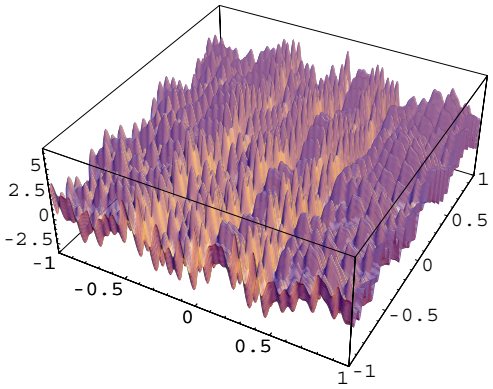
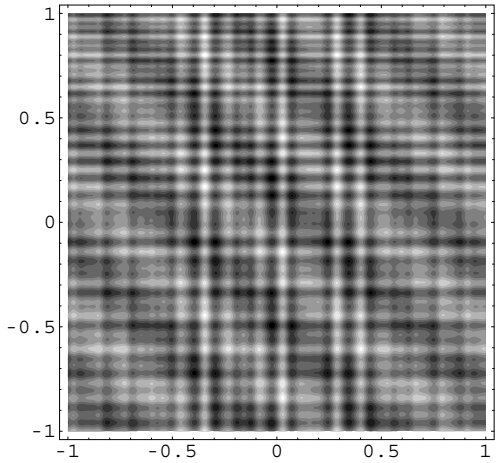


Figure 4.1: A global view of our function $f(x, y)$.



(a) A view of $f(x, y)$ bounded on a fixed domain.



(b) A contour plot of $f(x, y)$ with darker areas denoting smaller values.

Figure 4.2: Two different views of $f(x, y)$.

If this problem were given to a typical calculus student, the first thing he or she would do is take the partial derivatives of f with respect to x and y and then solve the system $\{f_x = 0, f_y = 0\}$. This naive approach would not work quite well since solving this system actually yields 2720 different solutions [6]. It is possible (but difficult) to find all the solutions and determine which critical points are minima,

maxima, and saddle points using contour data. However this is not the approach we will be taking. To solve such a problem we must split our task into several steps:

(1) We must first find a bounded region that contains the minimum. We already believe that the minimum lies within the square $[-1, 1] \times [-1, 1]$ but we would rather have a smaller enclosing interval to guarantee higher accuracy.

(2) The next step requires that we identify a rough location of the lowest point in that region.

(3) By finding a rough location of the minimum, we may use a method with the rough estimate as input to zoom in closer and pinpoint the minimum to high precision.

In *Mathematica* we can easily complete the first two steps by using `Min` and `Table`.

A *Mathematica* Session

For convenience, we define `f` in *Mathematica* to take two real-valued arguments or one `List` argument. By defining `f` in this way, we are able to make use of *Mathematica*'s advanced arbitrary-precision routines. However, we also can use `Compile` to create highly optimized byte-code when we define `f`. The benefits to using optimized code is that, like C or Fortran, we can specify up front what type of variable each input will be. By doing this, *Mathematica* will generate efficient internal code to make numerical expressions evaluate faster. In our case we will be using `Real` inputs to do our initial investigations. There is also a disadvantage to using a technique like this. The types that the `Compile` command handles correspond to the types that computers can handle at the machine-code level. Therefore, for real numbers we can only work with machine-precision, and not

arbitrary-precision. The other disadvantage is that `Compile` does not support `Interval` inputs.

```
f[x_, y_] := e^Sin[50 x] + Sin[60 e^y] + Sin[70 Sin[x]] + Sin[Sin[80 y]] - Sin[10 (x + y)] + (x^2 + y^2)/4;
fc = Compile[{x, y},
   e^Sin[50 x] + Sin[60 e^y] + Sin[70 Sin[x]] + Sin[Sin[80 y]] - Sin[10 (x + y)] + (x^2 + y^2)/4];
f[{x_, y_}] := f[x, y];
fcl = Compile[{{x, _Real, 1}}, e^Sin[50 x[[1]]] + Sin[60 e^x[[2]]] +
    Sin[70 Sin[x[[1]]]] + Sin[Sin[80 x[[2]]]] - Sin[10 (x[[1]] + x[[2]])] + (x[[1]]^2 + x[[2]]^2)/4];
```

Since we believe that the minimum lies within the square $[-1,1] \times [-1,1]$, we may consider a grid of points with boxes of size 0.01×0.01. We are able to find the f-values of all of these points and take the minimum and return its position.

```
grid = Flatten[Table[{x, y}, {x, -1, 1, 0.01}, {y, -1, 1, 0.01}], 1];
fgrid = fcl /@ grid;
{Min[fgrid], Flatten[Extract[grid, Position[fgrid, Min[fgrid]]], 1]}
```

```
{-3.24646, {-0.02, 0.21}}
```

Similarly, we can use a finer grid of points with boxes of size 0.001×0.001 to get an even better approximation.

```
grid = Flatten[Table[{x, y}, {x, -1, 1, 0.001}, {y, -1, 1, 0.001}], 1];
fgrid = fcl /@ grid;
{Min[fgrid], Flatten[Extract[grid, Position[fgrid, Min[fgrid]]], 1]}
```

```
{-3.30563, {-0.024, 0.211}}
```

Due to memory limitations, using a grid finer than 0.001×0.001 becomes difficult and forces us to use different methods to find the global minimum. Although using the grid approach gave an approximate location of the minimum, there is no guarantee that this is the true value. Rather, it serves as a guide and gives an upper bound to the f-value of the global minimum.

Knowing this bound, we can determine that the global minimum lies somewhere within the unit circle centered at the origin. Consider the set of points that lie outside the unit circle centered at the origin: the exponential term is at least $1/e$, the quadratic terms are at least $1/4$, and the sin terms contribute at least $-1, -1, -\sin 1$, and -1, respectively for an approximate total of -3.223. The difficulty now lies in determining a smaller region that contains the correct point and whether that region is small enough so it does not contain any other minimums.

The point $(-0.024, 0.211)$ serves as guide and not as an answer. Although this point was found through a grid search, it is not the approach we will be pursuing. Several teams in the SIAM challenge used a finer grid to search, together with estimates based on the partial derivatives of f to guarantee that the point found is actually the true minimum [6].

4.1 Survival of the Fittest

By using a grid search we were able to get a good approximation of the minimum, however it is easy to devise a routine that uses genetic algorithms to introduce randomness into the process of finding the solution to our problem. Genetic algorithms are inspired by biological evolution such as natural selection and survival of the fittest. A basic genetic algorithm consists of several steps. First, the evolution starts from a population of randomly generated individuals. Then at each generation, the fitness of each individual in the population is evaluated by a fitness function. If a fitness function yields a positive result, then the individual becomes eligible to survive for the next generation. For those that are not eligible,

they die off and do not survive for the next generation. The new population is then used in the next iteration of the algorithm [2].

We can relate this general definition to our problem as follows. Rather than individuals, we begin by using points that lie within $[-1, 1] \times [-1, 1]$. The points that survive at each generation will be called *parents*. We are able to utilize the upper bound to the global minimum in our fitness function. Our fitness function will evaluate each point (x, y) using f and if $f(x, y)$ is less than the current upper bound, then it has a possibility of surviving for the next generation. For each point in the current generation, n new random points are introduced, which we will call *children*. We then evaluate the group of children and parents using the fitness function and take the n best results. This group of n points becomes the parents for the next generation. After each generation, the domain in which the points are randomly chosen from shrinks.

Algorithm 4.1 (Genetic Algorithm to Minimize a Function)

Input: The objective function $f(x, y)$, the search rectangle R, the number of children generated at each generation and the number of total parents n, a bound ε on the absolute error in the location of the minimum of f in R, and a scaling factor s for shrinking the search domain at each generation.

Output: An upper bound to the minimum of f in R and an approximation to its location.

Notation: `parents` is the current generation of sample points, `fvals` is the set of f-values for the current generation, and `newfvals` is the set of f-values of the children formed during the generation.

1 z = center of R

2 `parents` = $\{z\}$

3 `fvals` = $\left\{f(z)\right\}$

4 $\{h_1, h_2\}$ = side-lengths of R

5 **while** $\min(h_1, h_2) > \varepsilon$

6 For each $p \in$ `parents`, let its children consists of n random points in a rectangle around p. We can get these points by randomly choosing x and y from $[-h_1, h_1]$ and $[-h_2, h_2]$, respectively.

7 `newfvals` = f-values on the set of all children.

8 Take the n lowest values from `fvals` $\cup$ `newfvals` and use this to determine the points from the children and previous parents that will survive.

9 Let `parents` be this set of n points and let `fvals` be the corresponding f-values.

10 Shrink our search domain by letting $h_i = s \cdot h_i$ for $i = 1, 2$.

11 **end while**

12 **return** the smallest value in `fvals` and the corresponding value from `parents`.

In problems with sufficient complexity, genetic algorithms like Algorithm 4.1 may have a tendency to converge toward local minima rather than the global minimum of the problem. The likelihood of this occurring depends on the shape of the function [2]. For the problem at hand, this is extremely likely to occur due to the complexity near the origin. This problem may be alleviated by increasing the number of points n introduced at each generation or by increasing the maximum number of generations.

A *Mathematica* Session

With a little experimentation, we can find that a value for n as low as 30 will get correct digits for our answer. However, to increase our confidence we can increase n to whatever value we desire. When $n = 70$, Algorithm 4.1 solves the problem with high probability (992 successes in 1000 runs). Using functional programming, we are able to program our genetic algorithm in a few lines of code [6]:

```
h = 1; gen = {f[#], #} & /@ {{0, 0}};
While[h > 10^-6,
  new = Flatten[Table[#[[2]] + Table[h (2 Random[] - 1), {2}], {50}] & /@ gen, 1];
  gen = Take[Sort[Join[gen, {f[#], #} & /@ new]], 50];
  h = h / 2];
gen[[1]]
```

```
{-3.3068686474668394,{-0.02440311773697685,0.21061239883434543}}
```

Using *Mathematica*, I developed the code in Appendix B to generate Figure 4.3 which illustrates how Algorithm 4.1 finds the true global minimum.

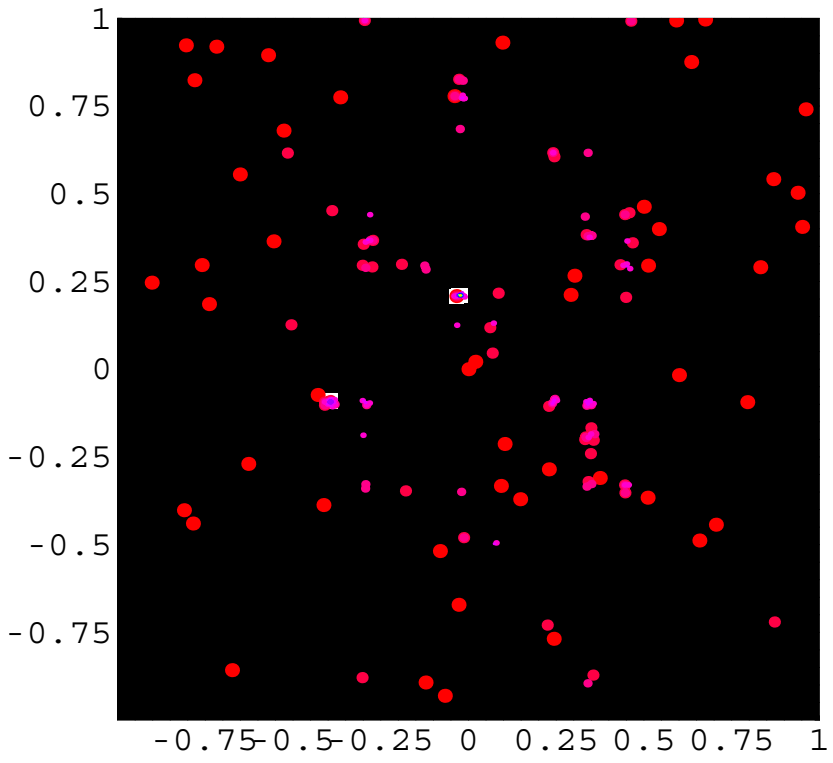


Figure 4.3: The results of Algorithm 4.1. The bound on the absolute error is 10^{-6} and the computation runs through 20 generations with 50 members each. Points of the same color correspond to the same generation. The white squares mark the location of the 20 lowest minima.

If one wanted to use a canned genetic algorithm, the `NMinimize` function minimizes a function f of any number of variables using several different

methods. It seems that both the genetic algorithms `DifferentialEvolution` and `RandomSearch` work quite well given enough starting points.

A *Mathematica* Session

```
NMinimize[{f[x, y], x^2 + y^2 ≤ 1}, {x, y},
 Method → {"DifferentialEvolution", "SearchPoints" → 250}]
```

```
{-3.306868647475238,{x→-0.024403079695523768,y→0.21061242715510411}}
```

Using a genetic algorithm approach has its benefits: it is easy to program and converges quickly. As long as we use a resolution fine enough, our algorithm will converge to a minimum for any number of digits. However, this approach does not guarantee that the minimum found is the true global minimum. Repeated runs of the algorithm will only increase confidence, which makes our algorithm useful for initial investigations but not for getting a complete solution. Next, we will be looking at algorithms devised using interval arithmetic which will not only give us a correct answer, but will provide a proof of correctness as well.

4.2 Interval Arithmetic

Using the ideas from Chapter 2 we are able to devise an algorithm to pinpoint the location of the global minimum. *Mathematica* has interval arithmetic built in and allows for elementary functions (such as f) to be applied to intervals. Using elementary algebra, we previously found that the global minimum must lie within the square $R = [-1, 1] \times [-1, 1]$. Using interval arithmetic we can verify this result since the interval value of f when applied to the half-plane $-\infty < x \leq -1$ is $[-3.223, \infty]$. The same is true if we rotate this half-plane $90°, 180°$, and $270°$ around the origin. What this shows us is that in these four regions (the complement of R) the function is greater than -3.223. Using our grid search we

were able to get an upper bound of −3.24 so we can ascertain that these regions can be ignored. In *Mathematica* this can be done quite easily.

A *Mathematica* Session

```
f[Interval[{-∞, -1.}], Interval[{-∞, ∞}]]
f[Interval[{1., ∞}], Interval[{-∞, ∞}]]
f[Interval[{-∞, ∞}], Interval[{-∞, -1.}]]
f[Interval[{-∞, ∞}], Interval[{1., ∞}]]
```

```
Interval[{-3.22359, ∞}]

Interval[{-3.22359, ∞}]

Interval[{-3.22359, ∞}]

Interval[{-3.22359, ∞}]
```

We will first present a simple subdivision algorithm to get a high-precision enclosing box around the global minimum. Then knowing a precise location of the global minimum, we may use a variation of Newton's method which converges quadratically to the global minimum, giving us whatever number of digits we desire.

4.2.1 Search & Destroy

Since our algorithm will use a subdivision process, the title "Search & Destroy" seems appropriate. In general, the algorithm will be searching the rectangle R and destroying any regions within R that do not have a chance of containing the global minimum. At each iteration of the algorithm, R will be subdivided into smaller rectangles. The candidates for removal can be identified by estimating the size of the function and its gradients. This process was the one used by the Wolfram team during the SIAM challenge and is one of the basic algorithms of interval arithmetic [6]. To make this more succinct, we start with R and the knowledge that the f-value for the global minimum is less than −3.24. We then repeatedly

subdivide R into smaller rectangles and retain only those subrectangles T which have a possibility of containing the global minimum. At each iteration, the subrectangles become candidates to be in T only if they pass the three conditions below. Throughout the rest of this chapter, we will denote $f[T]$ as the enclosing interval for $\{f(t) : t \in T\}$.

(1) $f[T]$ is an interval whose left end is less than or equal to the current upper bound on the absolute minimum.

(2) $f_x[T]$ is an interval whose left end is negative and right end is positive.

(3) $f_y[T]$ is an interval whose left end is negative and right end is positive.

For step (1), we need to keep track of the current upper bound. It is important to improve the upper bound as quickly as possible. In this implementation, the upper bound at each iteration will be determined by taking the min over all the upper bounds of intervals in $f[T]$. A possible improvement to this is called opportunistic evaluation. After subdivision, we can evaluate f at the rectangle centers and gain more information about the function f [6]. More importantly, this pointwise evaluation may give us a better upper bound. Steps (2) and (3) arise from the fact that our global minimum is a critical point and the gradients change from negative to positive for minima. Trying (1) by itself would quickly lead to a region that contains the global minimum. However, without (2) and (3) and due to the flat nature of the function near the minimum, the number of rectangles blows up. For example, after running Algorithm 4.2 for 8 iterations the number of rectangles has decreased to 13 whereas after 20 iterations the number of rectangles remaining has increased to 25757. Using (2) and (3) allows us to design an algorithm that is more aggressive in removing rectangles and will converge to an

answer. For our subdivision process, we have found that uniformly dividing each rectangle into four smaller rectangles is sufficient to get a solution.

Algorithm 4.2 (Using Intervals to Minimize a Function)

Assumptions: An interval arithmetic implementation is available that can evaluate f, f_x and f_y.

Input: The objective function $f(x, y)$, the search rectangle R, a bound ε on the absolute error in the location of the minimum of f in R, an upper bound b on the lowest f-value in R, and a bound $i_{\max}$ on the number of iterations to provide a definite stopping criteria.

Output: Interval approximation to the location of the minimum with interval size less than ε and the f-value at that location.

Notation: Let $\mathcal{R}$ denote the set of candidate rectangles, a_0 and a_1 be the lower and upper bounds, respectively, on the f-value sought, and an interior rectangle to be defined as one that lies within the interior of R.

1 $\mathcal{R} = \{R\}$
2 $i = 0$
3 $a_0 = -\infty$
4 $a_1 = b$
5 **while** $|a_1 - a_0| > \varepsilon$ **and** $i < i_{\max}$
6 $i = i + 1$
7 $\mathcal{R}$ = the set of all rectangles that arise from uniformly dividing each rectangle in $\mathcal{R}$ into 4 smaller rectangles.
8 // Update the current upper bound.
9 $a_1 = \min\left(a_1, \min_{T \in \mathcal{R}}(f[T])\right)$
10 // Perform step (1).
11 Delete any rectangle from $\mathcal{R}$ for which the left end of $f[T]$ is not less than a_1

12 // Perform steps (2) and (3).

13 Delete any rectangle from $\mathcal{R}$ for which $f_x[T]$ does not contain 0 or $f_y[T]$ does not contain 0.

14 // Update the current lower bound.

15 $a_0 = \min_{T \in \mathcal{R}}\big(f[T]\big)$

16 **end while**

17 **return** the approximate location of the minimum and its f-value.

Example code for Algorithm 4.2 can be found in Appendix B. The implementation takes very little time to run. Using a tolerance ε of 10^{-12} yields a result that is correct to 12 digits in a few seconds after running through 47 iterations. The total number of rectangles examined was 1372 and the number of candidate rectangles after each subdivision step are:

$$4, 16, 64, 240, 440, 232, 136, 48, 24, 12, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,$$
$$4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4.$$

We can see that on the third iteration, the algorithm began to discard rectangles. After 11 rounds, only one rectangle remained and was continuously subdivided into 4 rectangles until the given tolerance was met. Figure 4.4 shows the first 12 iterations of our algorithm and visually illustrates which rectangles were removed.

Algorithm 4.2 provides the first example of how we can get an arbitrary amount of digits to our solution using interval arithmetic. However, this method does not lend itself well to extreme precision. As mentioned before, the obvious way to improve our algorithm is to switch to a root finder on the gradient once we have a very good approximation to the location of the global minimum. As in $\mathbb{R}$,

we are able to use Newton's method to quadratically converge to a critical point. Since interval arithmetic is just an extension of $\mathbb{R}$ we will be introducing in the next section an algorithm that utilizes the gradient search method in conjunction with Algorithm 4.2 to get a very accurate solution.

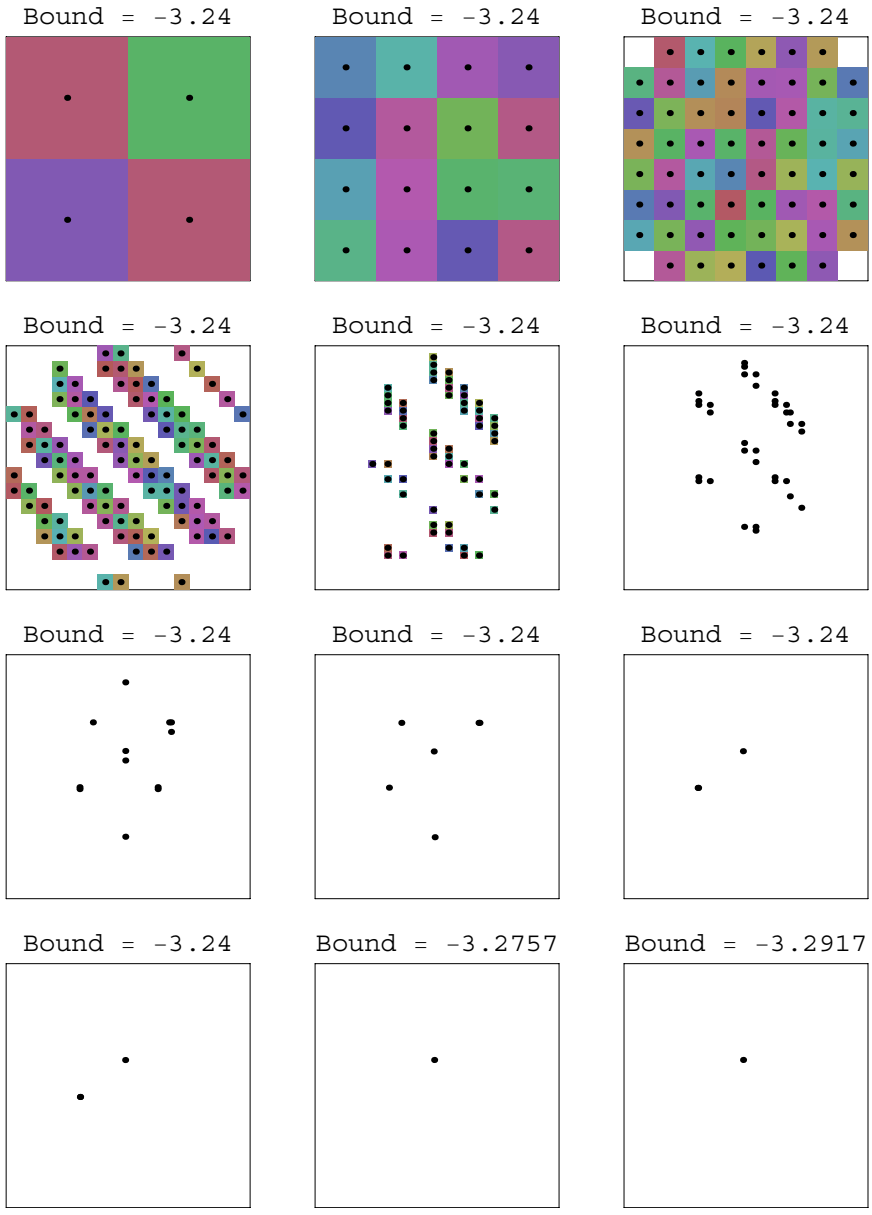


Figure 4.4: The first 12 iterations of Algorithm 4.2.

4.2.2 Newton & Krawczyk

Although Algorithm 4.2 solves our problem to a sufficient amount of digits, one improvement we can make is using a set of techniques called interval Newton methods. This idea originated from Ramon Moore [13] in 1966. There are generally two approaches to interval Newton methods. One is the "process of elimination" method used in Algorithm 4.2 which eliminated intervals that were not zeros of the gradient function. Another technique, involves the use of *interval contractions*; that is, interval functions F with the property that

$$F(X) \subset X. \tag{4.1}$$

If in addition to (4.1) there exists a positive real number $R < 1$ such that

$$w\Big(F(X)\Big) \leq R \cdot w(X)$$

then we call F a *strong interval contraction*. Here, $w(X)$ is the width of the interval X as defined in Section 2.2.

Similar to a contraction mapping in $\mathbb{R}$, iterating an interval contraction F yields a nested sequence of intervals which converges to an interval containing a fixed point based on the restriction imposed by F. If F is a strong interval contraction, then this process will converge to a degenerate interval based on the restriction imposed by F.

The best way to illustrate the interval Newton method is to first introduce it in one-dimension. Suppose that f is a real-valued function bounded on $x \in [a, b]$ and f has a continuous derivative f' on $[a, b]$. Then for $x, y \in [a, b]$ we have from the

mean-value theorem that

$$f(x) = f(y) + f'\Big(y + \theta(x - y)\Big)(x - y) \tag{4.2}$$

for some $\theta \in [0, 1]$. Now if x is a zero of f and if f' has a constant sign on $[a, b]$ (i.e. no zeros) then we may rearrange (4.2) to get

$$x = y + \left(\frac{1}{f'\Big(y + \theta(x - y)\Big)}\right) f(y). \tag{4.3}$$

Thus y is a fixed point of the function on the right-hand side of (4.3) if and only if it is a zero of f. By approximating θ by 0, we may iterate the right side of (4.3) to approximate the root of f.

Now let F' be an interval extension of f'. For $X \subset [a, b]$ we define the interval function N (for *Newton*) by

$$N(X) := m(X) - \left(\frac{1}{F'(X)}\right) f\Big(m(X)\Big) \tag{4.4}$$

where $m(X)$ is the midpoint of the interval X as in Section 2.2. To create an interval Newton method to find roots to our rational function f, we begin by choosing X_0 to be a starting seed and define a sequence of intervals $X_1, X_2, \ldots$ with

$$X_{n+1} := N(X_n) \cap X_n. \tag{4.5}$$

We may then iterate the right side of (4.4) using each X_i to get an interval bound on the roots of f. By using the interval extension F to evaluate the range of values

of f, we may use in place of (4.4) the more general form

$$N(X) := m(X) - \left(\frac{F\Big(m(X)\Big)}{F'(X)} \right). \tag{4.6}$$

This interval Newton method can easily be extended to higher dimensions given certain assumptions. If $f\colon \mathbb{R}^n \to \mathbb{R}^n$ is continuously differentiable and F is the interval extension of f, then we may define the Newton operator [6] on the n-dimensional box X by

$$N(X) := m(X) - J^{-1} \cdot F\Big(m(X)\Big), \tag{4.7}$$

where J is the interval Jacobian

$$F'(X) := \left(\frac{\partial F_i}{\partial x_j}(X) \right)_{ij} \quad \text{for } i, j = 1, 2, \ldots, n. \tag{4.8}$$

Here J^{-1} is obtained by the standard arithmetic interval operations that arise in the computation of a matrix inverse. It is easy to see that if $n = 1$ then (4.7) is simply the statement of (4.6). Note that $N(X) = [-\infty, \infty]$ if $0 \in F'(X)$. Having defined the Newton operator for n-dimensions, we may now begin to devise an algorithm to find roots of our function f. The one-dimensional properties of $N(X)$ are as follows.

Theorem 4.1 ([6],[13])**.** *Suppose $f\colon \mathbb{R} \to \mathbb{R}$ is a continuously differentiable function with interval extension F and $X = [a, b]$ is a finite interval. Then the Newton operator $N(X)$ as defined by* (4.4) *has the following properties:*

(a) If $N(X) \subseteq X$, then f has at most one zero in X.

(b) If r is a simple root of f in X, then r lies in $N(X)$.

(c) If $N(X) \subseteq X$, then f has a zero in X.

Proof. Clearly if f' has a zero in X, then given that

$$\bigcup_{x \in X} f'(x) \subset F'(X),$$

we know $F'(X)$ is an interval containing a zero so $N(X)$ is undefined. So, we will assume that $F'(X)$ does not contain zero and that $F\Big(m(X)\Big)$ is defined.

(a) Consider the case when f has two distinct zeros $x_1, x_2 \in X$. If this is true, then F' must have a zero in between $[x_1, x_2]$ and thus the derivative vanishes on this interval, making $N(X)$ undefined. Thus a necessary condition for $N(X)$ to be defined is that X contain at most one zero of f.

(b) If $r \in X$ is a simple root of F, then $F(r) = 0$. If $m(X) = r$, then using (4.4) we get $N(X) = r \in X$. Otherwise, suppose that $m(X) > r$ and consider the interval $\Big[r, m(X)\Big]$. Since f' is continuous on $\Big[r, m(X)\Big]$, it follows from the mean-value theorem that there exists a $c \in \Big[r, m(X)\Big] \subset X$ such that

$$f'(c) = \frac{f(r) - f\Big(m(X)\Big)}{r - m(X)}.$$

But since r is a zero of f, we have that

$$r = m(X) - \frac{f\Big(m(X)\Big)}{f'(c)}.$$

Since $f'(c) \in F'(X)$, we can conclude from (4.4) that $r \in N(X)$. The case where $m(X) < r$ can be treated similarly to the case $m(X) > r$.

(c) By assumption $F'(x) \neq 0$ for $x \in X$. Since f' is continuous on X, by the mean-value theorem there exists $c_1 \in \Big[a, m(X)\Big]$ and $c_2 \in \Big[m(X), b\Big]$ such that

$$f'(c_1) = \frac{f\Big(m(X)\Big) - f(a)}{m(X) - a}$$

$$\left(m(X) - \frac{f\Big(m(X)\Big)}{f'(c_1)}\right) - a = -\frac{f(a)}{f'(c_1)} \tag{4.9}$$

and

$$f'(c_2) = \frac{f(b) - f\Big(m(X)\Big)}{b - m(X)}$$

$$b - \left(m(X) - \frac{f\Big(m(X)\Big)}{f'(c_2)}\right) = \frac{f(b)}{f'(c_2)}. \tag{4.10}$$

Since $N(X) \subseteq X = [a, b]$, each expression on the left side of (4.9) and (4.10) is positive. Therefore, their product is also positive. However, $f'(c_1)$ and $f'(c_2)$ have the same sign since $F'(X)$ does not contain zero, so the product $f(a)f(b)$ must be negative and therefore by the intermediate value theorem f has a zero in X.

□

In addition to Theorem 4.1 we have the following Lemma which allows us to compute on intervals of nonzeros of f (i.e. to compute on intervals that contain x such that $f(x) \neq 0$) as well as to narrow down intervals which may contain a zero of f.

Lemma 4.2 ([13])**.** *Either $N(X) \cap X$ is empty, in which case X does not contain a zero of f, or else $N(X) \cap X$ is an interval which contains a zero of f if X does.*

Using both Theorem 4.1 and Lemma 4.2 we may devise a simple subdivision

algorithm to find zeros of a given function f. We first subdivide our seed interval into subintervals and for each subinterval check whether the Newton condition, $N(X) \subset X$, holds. If it does, then from (a) we know that f has at most one zero in X. If $N(X) \cap X = \emptyset$ we know from Lemma 4.2 that there are no zeros in X. If neither situation applies, we just subdivide and try again. However, if the Newton condition holds we can iterate the N operator as in (4.5). If we are close enough to a zero of f, this algorithm will converge [10, Theorem 11.15.6] and will do so quadratically [11, Theorem 1.14]. We will continue this subdivision process until the width of the remaining interval is less than a required precision. For an example and more specific statement of this algorithm see [13].

Newton's method for intervals has its share of difficulties. Although we know Newton's method will converge, it may not converge to a root but rather a real number in $N(X) \cap X$ [10]. This is due to the fact that we must approximate the interval value of J^{-1} and that there may be zeros of f for which the Newton condition will never hold. We may think of the Newton algorithm as a queue: in one step we remove intervals from the queue based on some properties of the Newton operator while at another step we subdivide intervals and make the queue larger. It also may be the case that these intervals are added to the list of solutions. If our queue becomes empty we are done since we either have found a solution or no solution exists. On the other hand, we may stop after a certain number of iterations, thus leaving us with unresolved intervals. This case would happen if there are multiple roots such as in $f = (x-2)^2$ or if a root exists at 0 as in $g = x^2$ [6].

Although this algorithm works well in one dimension, when looking at higher dimensions it becomes difficult to guarantee the existence of a zero in as in part (c) of Theorem 4.1. This forces us to use a variation of the Newton operator. One

method is to use a preconditioning matrix and the Hansen-Sengupta operator [15]. The other method makes use of something called the *Krawczyk operator*. The Krawczyk method was introduced in 1969 by R. Krawczyk and first analyzed in 1977 by Moore [14]. It is derived by considering the classical multivariate Newton method as a fixed point iteration. The benefit of this approach over Newton's method is not only that it will computationally verify the existence of a solution in a given region, but it does so without inverting an interval matrix. In general, Krawczyk's method allows us to find a solution to a system of nonlinear equations using interval methods that will converge at least linearly with guaranteed error bounds [14].

Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ be continuously differentiable in the open domain D. We assume that f and f' have continuous, inclusion monotonic interval extensions F and F' defined on interval vectors contained in D. Let $X = (X_1, X_2, \ldots, X_n)$ be contained in D where $X_1, X_2, \ldots, X_n$ are closed bounded real intervals. Finally let J be defined as in (4.8). Now we define the operator $P(X)$ to be

$$P(X) := X - YF(X) \tag{4.11}$$

where Y is some type of approximation to J^{-1}. One natural choice for Y is $J^{-1}\big(m(X)\big)$, and the other is the inverse of the matrix of midpoints of the intervals in the matrix J. The latter is faster since J need to be computed anyway. By choosing this as our approximation for Y, we are able to avoid the inversion of the matrix J as required. The operator P is carefully chosen to help guarantee the existence of solutions and has the following property:

Lemma 4.3 ([14]). *If $P(X)$ maps X into itself, then $f(x) = 0$ has a solution in X.*

We can now define the Krawczyk operator as follows:

$$K(X) := m(X) - YF\Big(m(X)\Big) + (I - YJ)\Big(X - m(X)\Big) \tag{4.12}$$

where I is the $n \times n$ identity matrix. Note that to actually do this computation in *Mathematica* it is required that we put a small interval around $m(X)$. In the simplest sense, the Krawcyzk method is just the iteration $X_{n+1} := K(X_n) \cap X_n$ applied to subintervals of a seed interval X_0. To understand the rationale behind the Krawczyk operator, consider the form the mean-value theorem takes in n-dimensions. Given F as defined on page 57 and x and y in an n-dimensional box X, there are points $c_1, c_2, \ldots, c_n \in X$ so that $F(x) - F(y) = \Big(\nabla F_i(c_i)\Big)(x - y)$, where $\Big(\nabla F_i(c_i)\Big)$ denotes an $n \times n$ matrix with i indexing the rows. This can be stated in the following lemma:

Lemma 4.4 ([14]). *$f(x) - f(y) \in F'(X)(x - y)$ for all $x, y \in X$.*

Now, to show that the definition of $K(X)$ follows directly from the definition of P and the mean-value theorem, let $P'(X)$ denote the interval enclosure of P' on X. By the mean-value theorem, the box $Q = P\Big(m(X)\Big) + P'(X)\Big(X - m(X)\Big)$ contains $P(X)$ where $P(X)$ denotes the exact image, $\{P(x) : x \in X\}$. Since $P'(x) = I - YF'(x)$ and J is an approximation of $F'(x)$, we may take $I - YJ$ to be the enclosure of $P'(x)$. Thus, Q becomes precisely the statement of $K(X)$ from (4.12) and thus $P(X) \subseteq K(X)$. We may now generalize the results of the Krawczyk operator to n-dimensions as follows after the necessary statement of the following theorem.

Theorem 4.5 (Brouwer fixed point theorem, [11]). *Let D be homeomorphic to the closed unit ball in $\mathbb{R}^n$, and suppose P is a continuous mapping such that the range $P^u(D) \subset D$. Then P has a fixed point, i.e. there is an $X \in D$ such that $P(X) = X$.*

Theorem 4.6 ([6]). *Suppose $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ is continuously differentiable in the open domain D and assume that f and f' have continuous, inclusion monotonic interval extensions F and F' defined on interval vectors contained in D. Let $X = (X_1, X_2, \ldots, X_n)$ be a finite box contained in D where $X_1, X_2, \ldots, X_n$ are closed bounded real intervals. Then, $J = F'(X)$ is a component wise interval enclosure, and Y is the inverse of the matrix of midpoints of the intervals in J. We will assume that Y is nonsingular. Let K be the Krawczyk operator as defined in* (4.12)*. Then:*

(a) If r is a root of $F = 0$ in X, then r lies in $K(X)$.

(b) If $K(X) \subseteq X$, then there exists a solution to $f(x) = 0$ in X.

(c) If $K(X) \subset X$, then f has at most one zero in X.

Proof.

(a) Since $P(X) \subseteq K(X)$, then $P(r) \in K(X)$. But since $F(r) = 0$, it follows that $P(r) = r - YF(r) = r$ and thus r lies in $K(X)$.

(b) We will present two ways of proving this claim with the first attributed to [14] and the second to [6]. For the first proof, we know from the definition of $P(x)$ from (4.11) that

$$P(x) = x - Yf(x) = y - Yf(y) + x - y - Y\Big(f(x) - f(y)\Big) \quad \text{for all } x \in X.$$

We may use Lemma 4.4 to obtain

$$P(x) \in y - Yf(y) + \Big(I - YF'(X)\Big)(X - y) \quad \text{for all } x \in X.$$

Thus, $P(x) \in K(X)$ for all $x \in X$. If $K(X) \subseteq X$, then P maps X into itself and from Lemma 4.3 it follows that $f(x) = 0$ has a solution in X.

For the second proof, since we have the containment relation $P(X) \subseteq K(X) \subseteq X$, P is a contraction mapping and therefore by Theorem 4.5, P has a fixed point in X with this fixed point being the solution to $f(x) = 0$ in X.

(c) This is part (iii) of Theorem 5.1.8 in [11].

□

Given Theorem 4.6 we may devise a local root-finding algorithm that will find a root of f given the assumption that our initial box X contains a root. By parts (b) and (c) of Theorem 4.6 we know that if the Krawczyk condition $K(X) \subset X$ holds, then f has one and only one root in X and that root lies in $K(X)$. So we may iterate K until we know the root to the desired accuracy.

The importance of the Krawczyk method to the SIAM problem at hand is that it will speed up the process of finding the global minimum once we are given a box that is known to contain it. After 11 iterations we found that Algorithm 4.2 gave us a box that is guaranteed to contain the global minimum. Using this box, we can use the Krawczyk method to zoom into the unique critical point in the box giving us a speed up of about 10% [6]. To implement this, we need only to add the following after the gradient check (step 13) in Algorithm 4.2:

If $\mathcal{R}$ contains only one rectangle X, compute $K(X)$.

If $K(X) \cap X = \emptyset$, then there is no critical point, and the minimum is on the border. So do nothing.

If $K(X) \subset X$, then iterate the K operator starting with $K(X)$ until the desired tolerance is reached.

Use the last rectangle to set a_0 and a_1 and then end the while loop.

The extended implementation of Algorithm 4.2 using the Krawczyk operator can be found in [6]. There are multiple improvements to this algorithm as seen in [10, 11]. For example, we can use the Hessian to eliminate n-dimensional intervals on which the function is not concave up.

In summary, interval analysis is an important tool for global optimization problems since it gives high precision results that are verifiably correct. Although it is commonplace in the scientific computing community to just get the answer, interval analysis is another method that will provide a solid algorithm that will provide proven results.

CHAPTER 5

A Daunting Matrix

Problem. Let A be the $20,000 \times 20,000$ matrix whose entries are zero everywhere except for the primes $2, 3, 5, 7, \ldots, 224737$ along the main diagonal and the number 1 in all the positions a_{ij} with $|i - j| = 1, 2, 4, 8, \ldots, 16384$. What is the $(1, 1)$ entry of A^{-1}?

Out of all the problems in the SIAM challenge, this problem proved to be one of the easiest with each of the 78 teams getting an average of 8.8 points. However, I found this to be the most difficult out of the problems I studied since it required an extensive knowledge of linear algebra. Unlike the other problems, we will not solve this problem using interval arithmetic, but rather use interval arithmetic to show that round-off errors during our computations will not perturb our answer too much. This problem is unique since we are able to essentially solve it using one command in MATLAB or *Mathematica* using standard IEEE double-precision. It is also the only problem in the SIAM challenge to be solved exactly [6].

In most numerical analysis textbooks, in the chapter on numerical linear algebra it is usually implicitly stated that the problem of finding the inverse of a matrix is itself a task better left undone. The task generally requires a huge amount of operations and any algorithm to find one is usually numerically

unstable, with the chance of introducing approximation errors. A better way to approach this problem is to reformulate it as a linear equation of the form

$$A\mathbf{x} = \mathbf{b} \quad \text{with } \mathbf{b} = (1, 0, 0, \ldots, 0)^T \text{ and } \mathbf{x} = (x_1, \ldots, x_n)^T. \tag{5.1}$$

Observe that since

$$A^{-1} = \left(\begin{array}{c|ccc} x_1 & * & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ x_n & * & \cdots & * \end{array}\right)$$

then our problem is just the first of entry of $\mathbf{x}$ in (5.1), namely, x_1. If we assume the determinant of A is non-zero, then the solution of the linear system can be written as $\mathbf{x} = A^{-1}\mathbf{b}$. As stated before this approach is not adopted in practice.

Before we can discuss another method of solving this linear system, we must figure out a way to store our matrix A on a computer. Having our matrix stored in an efficient data structure will allow us to study why this problem is so daunting.

5.1 A First Look

The given matrix has two defining properties: one relates to its main diagonal and the other deals with the other non-zero entries. This allows us to define A in the following way:

$$\begin{aligned} A_{ii} &= p_i \quad \text{with } p_i \text{ the } i\text{th prime number,} \\ A_{ij} &= 1 \qquad \text{if } |i - j| \text{ is a power of 2, and} \\ A_{ij} &= 0 \qquad \text{otherwise.} \end{aligned} \tag{5.2}$$

Thus for $n = 20000$, A is a highly sparse matrix with $O(n \log n)$ nonzero elements (554,466 nonzero entries to be exact). Out of a total of $n^2 = 4 \cdot 10^8$ entries total, this accounts for about 0.14% [6]. In Figure 5.1 we can visually represent A and see how few nonzero entries there are. To prevent confusion in the future, we will denote the matrix for the problem at hand by A_n.

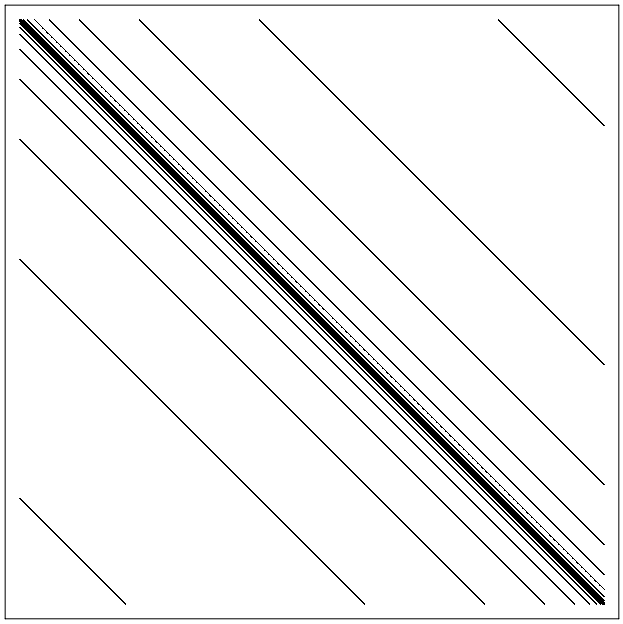


Figure 5.1: The sparsity pattern of A_n. The black represent nonzero entries while the white represents zero entries.

A *Mathematica* Session

Mathematica provides tools for dealing with sparse arrays; that is, we are able to build, store, and operate on them. Using `SparseArray` we can create an efficient container for A as is done in the following code.

```
n = 20000;
b = Table[0, {n}];
b[[1]] = 1;
A = SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &@
   SparseArray[Flatten@Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2., n - i]}], n];
```

Using this method allows us to Generate A in 1.656 seconds and consumes approximately 4.3MB.

When designing the SIAM challenge, Trefethen kept in mind that he needed to make these problems solvable. To make the equation $A\mathbf{x} = \mathbf{b}$ solvable it helps that A has some "nice" properties. Clearly A is symmetric with a positive diagonal and it is also sparse, but what makes A unique is that it is also *positive definite*. A matrix A is positive definite [17] if for every non-zero vector $\mathbf{x}$

$$\mathbf{x}^T A\mathbf{x} > 0. \tag{5.3}$$

This notion is not a very intuitive one, but its relevance will be explained later. From [20] we find the additional property that a real symmetric matrix A is positive definite if and only if there exists a real nonsingular matrix M such that

$$A = MM^T \tag{5.4}$$

There are two algorithms that may be used to prove that A_n is positive definite: the *Cholesky factorization* and the *Reverse Cuthill-Mckee algorithm*. When given A_n as input, both algorithms complete which by construction proves that A_n is positive definite [6]. If we wished to solve the linear system directly we would have to use either of these two in our sparse matrix solvers. Both of these algorithms allow us to solve for x_1 and they only give our solution to 15 digit accuracy. Because of the large memory constraints put on the computer for storing the factorization of A_n using either of these methods, it becomes difficult to get higher precision. Therefore we need to consider other methods for finding our solution. It is important to note that with A_n being positive definite, there are already a handful of algorithms that can deal with finding its inverse [5]. We will be looking at the

Conjugate Gradient method which is an iterative method that lends itself well to sparse symmetric positive definite matrices. Before we can state the algorithm, we have to introduce a lot of mathematical machinery that will illustrate why this method is suitable for a sparse symmetric positive definite matrix like A_n.

The main source of information I used for understanding the Conjugate Gradient method in the following discussion came from a paper [17] written by Jonathan Shewchuk, a student at Carnegie Mellon University. His goal in writing the paper was to create an extensive resource for those who wished to understand the Conjugate Gradient method which was less obscure than the description found in most textbooks. Many of the derivations in the following sections come from this paper as well.

5.2 Quadratic Forms

Since we are working in $\mathbb{R}^n$ we need a way to multiply vectors, so we will define a generalization of the dot product. The *inner product* of two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ is written $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T\mathbf{y}$ which represents the scalar sum $\sum_{i=1}^{n} x_i y_i$. We also have the relation that $\langle \mathbf{x}, \mathbf{y} \rangle = \langle \mathbf{y}, \mathbf{x} \rangle$.

The *quadratic form* is simply a scalar, quadratic function of a vector $\mathbf{x} \in \mathbb{R}^n$ with the form

$$f(\mathbf{x}) := \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x} + c \tag{5.5}$$

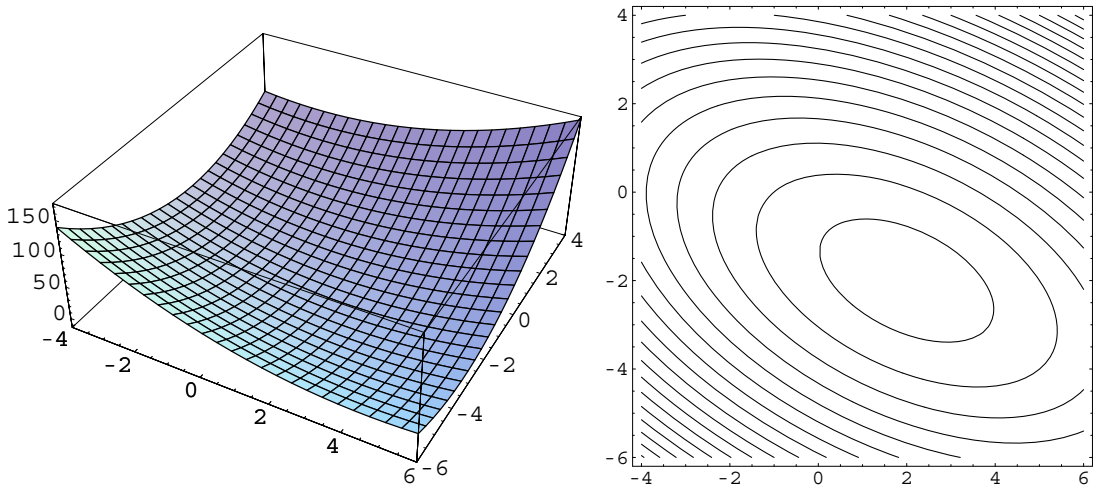
where A is a matrix, $\mathbf{b}$ is a vector in $\mathbb{R}^n$, and c is a scalar constant.

To help illustrate the Conjugate Gradient method, we will demonstrate several

ideas by applying the quadratic form to

$$A = \begin{pmatrix} 3 & 2 \\ 2 & 6 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 2 \\ -8 \end{pmatrix}, \qquad c = 0. \tag{5.6}$$

For this problem, the solution is $\mathbf{x} = (2, -2)^T$. The quadratic form and the contour plot corresponding to the problem in (5.6) are shown in Figure 5.2.



(a) 3D plot of the quadratic form. **(b)** Contour plot of the quadratic form.

Figure 5.2: Two different views of the quadratic form of our sample problem.

One thing we can notice about Figure 5.2 is that the surface determined by $f(\mathbf{x})$ is a paraboloid, with the solution to the linear system $A\mathbf{x} = \mathbf{b}$ being the global minimum. In fact, for every positive definite matrix A and any vector $\mathbf{b}$, the surface defined by $f(\mathbf{x})$ will be a paraboloid bowl and $f(\mathbf{x})$ is minimized by the solution to $A\mathbf{x} = \mathbf{b}$ [17].

Before we prove this, we introduce one additional concept. The *gradient* of a quadratic form is defined to be

$$f'(\mathbf{x}) = \begin{pmatrix} \frac{\partial}{\partial x_1} f(\mathbf{x}) \\ \frac{\partial}{\partial x_2} f(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_n} f(\mathbf{x}) \end{pmatrix}. \tag{5.7}$$

The gradient is a vector field that points in the direction of greatest increase of $f(\mathbf{x})$. Figure 5.3 illustrates the gradient vectors for the example defined in (5.6).

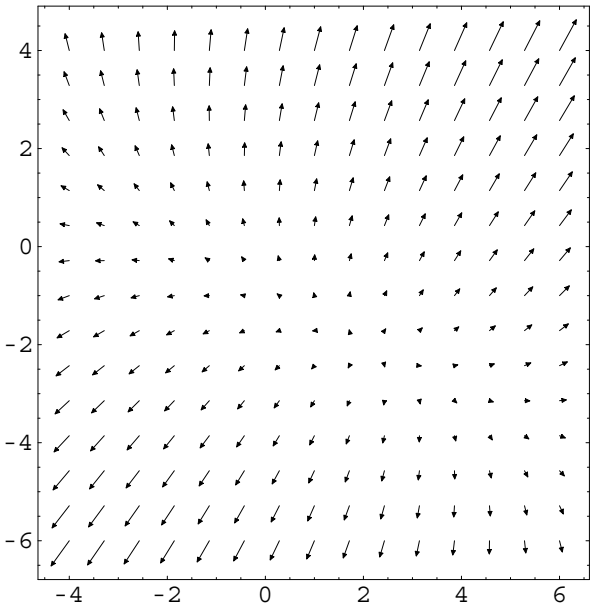


Figure 5.3: The gradient field of $f'(\mathbf{x})$. The arrow indicates a gradient sampled at a $\mathbf{x} \in \mathbb{R}^2$. It points in the direction of steepest increase of $f(\mathbf{x})$ and is orthogonal to the contour lines of Figure 5.2.

Now we will prove what we claimed earlier about the relation between $f(\mathbf{x})$ and $A\mathbf{x} = \mathbf{b}$.

Theorem 5.1 ([17])**.** *If A is a symmetric positive definite matrix, the solution to $A\mathbf{x} = \mathbf{b}$ is a critical point of $f(\mathbf{x})$. In fact, $\mathbf{x}$ is equal to the global minimum of $f(\mathbf{x})$.*

Proof. Given $f(\mathbf{x})$ as in (5.5), we want to find $f'(\mathbf{x})$. Through some tediuous computation, applying (5.7) to $f(\mathbf{x})$ allows us to derive:

$$f'(\mathbf{x}) = \frac{1}{2}A^T\mathbf{x} + \frac{1}{2}A\mathbf{x} - \mathbf{b}. \tag{5.8}$$

Since A is symmetric, $A^T = A$ and (5.8) reduces to

$$f'(\mathbf{x}) = A\mathbf{x} - \mathbf{b}. \tag{5.9}$$

Setting the gradient equal to zero, we obtain the linear system $A\mathbf{x} = \mathbf{b}$ which we wish to solve. Therefore, the solution to $A\mathbf{x} = \mathbf{b}$ is a critical point of $f(\mathbf{x})$. Now let $\mathbf{x}$ be a point that satisfies $A\mathbf{x} = \mathbf{b}$ and minimizes the quadratic form (5.5) and let ε be a vector error term. Then

$$\begin{aligned}
f(\mathbf{x}+\varepsilon) &= \frac{1}{2}(\mathbf{x}+\varepsilon)^T A(\mathbf{x}+\varepsilon) - \mathbf{b}^T(\mathbf{x}+\varepsilon) + c \\
&= \frac{1}{2}(\mathbf{x}^T + \varepsilon^T)A(\mathbf{x}+\varepsilon) - \mathbf{b}^T(\mathbf{x}+\varepsilon) + c \\
&= \frac{1}{2}(\mathbf{x}^T A\mathbf{x} + \mathbf{x}^T A\varepsilon + \varepsilon^T A\mathbf{x} + \varepsilon^T A\varepsilon) - \mathbf{b}^T\mathbf{x} - \mathbf{b}^T\varepsilon + c \\
&= \frac{1}{2}\mathbf{x}^T A\mathbf{x} + \varepsilon^T A\mathbf{x} + \frac{1}{2}\varepsilon^T A\varepsilon - \mathbf{b}^T\mathbf{x} - \mathbf{b}^T\varepsilon + c \qquad \text{(since } A \text{ is symmetric)} \\
&= \left(\frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x} + c\right) + \left(\frac{1}{2}\varepsilon^T A\varepsilon + \varepsilon^T\mathbf{b} - \mathbf{b}^T\varepsilon\right) \\
&= f(\mathbf{x}) + \frac{1}{2}\varepsilon^T A\varepsilon.
\end{aligned}$$

Since A is positive definite, then the latter term is positive for all $\varepsilon \neq 0$. Therefore, $\mathbf{x}$ minimizes f. □

Having proved Theorem 5.1, we have converted the problem of solving the linear system to one of minimizing our quadratic form $f(\mathbf{x})$. By doing so, we are able to use methods like Steepest Descent (which we will discuss in the next

section) and the Conjugate Gradient method to find a solution. Using these methods will provide a more stable algorithm for finding the $(1,1)$ entry of A^{-1}.

5.3 Steepest Descent

For the method of Steepest Descent, we start at an arbitrary point $\mathbf{x}_0$ and slide down to the bottom of the paraboloid defined by $f(\mathbf{x})$. We take a series of steps $\mathbf{x}_1, \mathbf{x}_2, \ldots$ until we come within a reasonable distance from the true solution $\mathbf{x}$. This stopping criteria could be an absolute error bound or we could stop our iteration after a certain period of time. When we take a step, we choose a direction in which f decreases most quickly; that is, the opposite of $f'(\mathbf{x}_i)$. Making use of (5.9), we find that this direction is

$$-f'(\mathbf{x}_i) = \mathbf{b} - A\mathbf{x}_i.$$

Let us introduce some additional definitions that are useful for our study of the Steepest Descent method and the Conjugate Gradient method. The *error* $\mathbf{e}_i = \mathbf{x}_i - \mathbf{x}$ is a vector that indicates the distance we are from the true solution. The *residual* $\mathbf{r}_i = \mathbf{b} - A\mathbf{x}_i$ indicates how far we are from the correct value of $\mathbf{b}$ at each step i. Using the fact that $\mathbf{x} = \mathbf{x}_i - \mathbf{e}_i$, $A\mathbf{x} = \mathbf{b}$ and the definition of residual, we can easily see that $\mathbf{r}_i = -A\mathbf{e}_i$. More importantly, we should think of the residual as the direction of steepest descent [17].

For our first step, we will step along the direction of steepest descent – the residual. In other words, we will be choosing a point

$$\mathbf{x}_1 = \mathbf{x}_0 + \alpha\mathbf{r}_0. \tag{5.10}$$

This leaves us with the question: how do we find an appropriate α so that we

optimize our choice for our next search direction? To do this, we need to use a *line search*. A line search is a procedure that finds an appropriate α to minimize f along a line. From calculus, α minimizes f along a line when the directional derivative $\frac{d}{d\alpha} f(\mathbf{x}_1)$ is equal to zero. Using the chain rule, it follows that

$$\frac{d}{d\alpha} f(\mathbf{x}_1) = f'(\mathbf{x}_1)^T \frac{d}{d\alpha} \mathbf{x}_1 = f'(\mathbf{x}_1)^T \mathbf{r}_0 = \langle f'(\mathbf{x}_1), \mathbf{r}_0 \rangle.$$

Setting this expression to zero, we find that α should be chosen so that $\mathbf{r}_0$ and $f'(\mathbf{x}_1)$ are orthogonal; that is, when the gradient is orthogonal to the search line [17].

To determine α we perform the following set of operations:

$$\begin{aligned}
f'(\mathbf{x}_1)^T \mathbf{r}_0 &= 0 \\
\mathbf{r}_1^T \mathbf{r}_0 &= 0 \qquad \text{(since } f'(\mathbf{x}_1) = -\mathbf{r}_1\text{)} \\
(\mathbf{b} - A\mathbf{x}_1)^T \mathbf{r}_0 &= 0 \\
\Big(\mathbf{b} - A(\mathbf{x}_0 + \alpha \mathbf{r}_0)\Big)^T \mathbf{r}_0 &= 0 \\
(\mathbf{b} - A\mathbf{x}_0)^T \mathbf{r}_0 - \alpha (A\mathbf{r}_0)^T \mathbf{r}_0 &= 0 \\
(\mathbf{b} - A\mathbf{x}_0)^T \mathbf{r}_0 &= \alpha (A\mathbf{r}_0)^T \mathbf{r}_0 \\
\mathbf{r}_0^T \mathbf{r}_0 &= \alpha \mathbf{r}_0^T (A\mathbf{r}_0) \\
\alpha &= \frac{\mathbf{r}_0^T \mathbf{r}_0}{\mathbf{r}_0^T (A\mathbf{r}_0)}.
\end{aligned}$$

Using our previous knowledge, we arrive at a general form for the method of

Steepest Descent:

$$\mathbf{r}_i = \mathbf{b} - A\mathbf{x}_i \tag{5.11a}$$

$$\alpha_i = \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{r}_i^T (A\mathbf{r}_i)} \tag{5.11b}$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{r}_i. \tag{5.11c}$$

The method above runs until it converges. An example run shown in Figure 5.4 uses the values from (5.6) and code from Appendix C. The zig zag path appears because each gradient is orthogonal to the previous gradient.

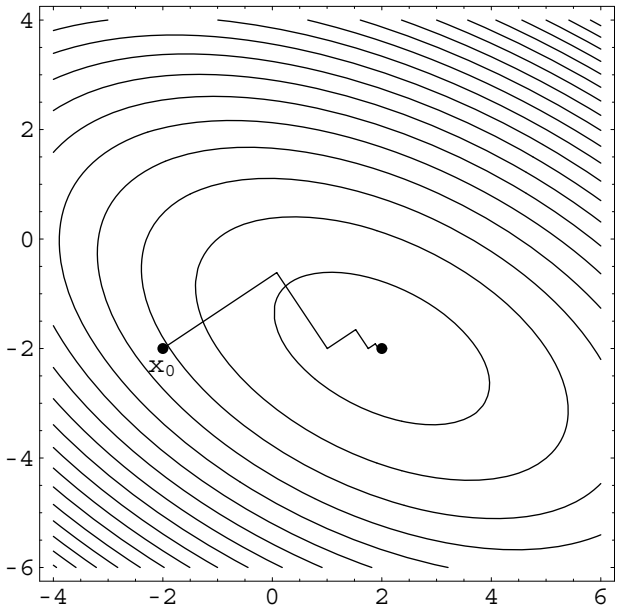


Figure 5.4: For our example, the method of Steepest Descent converges to within a tolerance of 10^{-6} in 22 iterations.

The algorithm above suffers one flaw, there are two matrix-vector multiplications per iteration. Although the example from (5.6) converges quite quickly, a matrix of larger size like in the SIAM problem would run quite slow. To combat

this, notice that by premultiplying both sides of (5.11c) by $-A$ and adding $\mathbf{b}$ we get:

$$\begin{aligned} \mathbf{b} - A\mathbf{x}_{i+1} &= (\mathbf{b} - A\mathbf{x}_i) - \alpha_i A\mathbf{r}_i \\ \mathbf{r}_{i+1} &= \mathbf{r}_i - \alpha_i A\mathbf{r}_i \qquad \big(\text{by (5.11a)}\big). \end{aligned} \tag{5.12}$$

For the first iteration of our algorithm, we need (5.11a) to calculate $\mathbf{r}_0$, but for every other iteration we may use (5.12). The benefit of this approach is that we have eliminated one matrix-vector product and we are left with only the product $A\mathbf{r}_i$ which appears in (5.11b) and (5.12). The disadvantage to this approach is that we are calculating the residual without any input from $\mathbf{x}_i$ which may cause our algorithm to converge to a point near the true solution $\mathbf{x}$ [17]. This is due to roundoff errors in floating-point calculations. For more complex expressions than in our example the error might be even greater. By periodically computing (5.11a), we can avoid this problem. In the code in Appendix C, I recompute the residual every 5 iterations.

5.4 The Method of Conjugate Directions

One disadvantage to the method of Steepest Descent is that we often find ourselves taking steps in the same direction as earlier steps (as seen in Figure 5.4). This shortcoming leads us to develop a new algorithm which utilizes results from the Steepest Descent algorithm.

5.4.1 Conjugacy

To begin, choose a set of orthogonal *search directions* $\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_{n-1}$. In each search direction we will take exactly one step and that one step will be sufficient enough to line up with $\mathbf{x}$. Thus after n steps, we will be done. In general, for each step we will choose a point

$$\begin{aligned} \mathbf{x}_{i+1} &= \mathbf{x}_i + \alpha_i \mathbf{d}_i \\ \mathbf{e}_{i+1} &= \mathbf{e}_i + \alpha_i \mathbf{d}_i. \end{aligned} \tag{5.13}$$

To make sure that we never step in the same direction of $\mathbf{d}_i$ again, we should have the condition that $\mathbf{e}_{i+1}$ be orthogonal to $\mathbf{d}_i$. Using this fact, we can find α_i:

$$\begin{aligned} \mathbf{d}_i^T \mathbf{e}_{i+1} &= 0 \\ \mathbf{d}_i^T (\mathbf{e}_i + \alpha_i \mathbf{d}_i) &= 0 \qquad \big(\text{by (5.13)}\big) \\ \alpha_i &= -\frac{\mathbf{d}_i^T \mathbf{e}_i}{\mathbf{d}_i^T \mathbf{d}_i}. \end{aligned} \tag{5.14}$$

Unfortunately, α_i is defined in terms of $\mathbf{e}_i$ and if we knew the value of $\mathbf{e}_i$, then our problem would already be solved. To fix this, we should make the search directions A-orthogonal instead of orthogonal. We say that two vectors $\mathbf{d}_i$ and $\mathbf{d}_j$, $i \neq j$, are *A-orthogonal* or *conjugate* if

$$\mathbf{d}_i^T A \mathbf{d}_j = 0.$$

Using the new condition that $\mathbf{e}_{i+1}$ be A-orthogonal to $\mathbf{d}_i$, we claim that this is equivalent to finding the minimum point along the search direction $\mathbf{d}_i$, as done for the line search in the method of Steepest Descent. To see this, set the directional

derivative equal to zero:

$$\begin{aligned}
\frac{d}{d\alpha} f(\mathbf{x}_{i+1}) &= 0 \\
f'(\mathbf{x}_{i+1})^T \frac{d}{d\alpha} \mathbf{x}_{i+1} &= 0 \\
-\mathbf{r}_{i+1}^T \mathbf{d}_i &= 0 \qquad \text{(since } f'(\mathbf{x}_{i+1}) = -\mathbf{r}_{i+1}\text{)} \\
\mathbf{d}_i^T A \mathbf{e}_{i+1} &= 0 \qquad \text{(since } \mathbf{r}_{i+1} = -A\mathbf{e}_{i+1}\text{)}.
\end{aligned}$$

By following the same process as the derivation for (5.14) and using the fact that the search directions are A-orthogonal we get the following:

$$\begin{aligned}
\mathbf{d}_i^T A \mathbf{e}_{i+1} &= 0 \\
\mathbf{d}_i^T A(\mathbf{e}_i + \alpha_i \mathbf{d}_i) &= 0 \qquad \Big(\text{by (5.13)}\Big) \\
\alpha_i &= -\frac{\mathbf{d}_i^T A \mathbf{e}_i}{\mathbf{d}_i^T A \mathbf{d}_i} \qquad (5.15) \\
&= \frac{\mathbf{d}_i^T \mathbf{r}_i}{\mathbf{d}_i^T A \mathbf{d}_i}. \qquad (5.16)
\end{aligned}$$

Unlike (5.14), we can actually compute α_i in this instance. By replacing the search direction $\mathbf{d}_i$ with the residual $\mathbf{r}_i$ in (5.16) it is easy to see the relationship between the method of Conjugate Directions and the method of Steepest Descent: (5.16) would be identical to (5.11b).

By choosing the search directions carefully, we will be able to converge to our solution $\mathbf{x}$ in n steps. The idea is that the initial error $\mathbf{e}_0$ can be expressed as a sum of A-orthogonal components. At each step the Conjugate Directions algorithm eliminates one of these components; that is, we eliminate the error component $\alpha_i \mathbf{d}_i$ of $\mathbf{e}_0$ at step i. We finish this section by proving this result as the following theorem.

Theorem 5.2 ([17])**.** *The method of Conjugate Directions converges in n steps.*

Proof. We begin by expressing the error term as a linear combination of search directions:

$$\mathbf{e}_0 = \sum_{j=0}^{n-1} \delta_j \mathbf{d}_j. \tag{5.17}$$

To find values for δ_j we need to employ a mathematical trick. By premultiplying the expression in (5.17) by $\mathbf{d}_k^T A$, we can all but eliminate one δ_k. First,

$$\mathbf{d}_k^T A \mathbf{e}_0 = \sum_{j=0}^{n-1} \delta_j \mathbf{d}_k^T A \mathbf{d}_j,$$

but since each $\mathbf{d}$ vector is A-orthogonal, $\mathbf{d}_i^T A \mathbf{d}_j = 0$ for $i \neq j$. The only case where we do not get zero is when $j = k$ so

$$\begin{aligned} \mathbf{d}_k^T A \mathbf{e}_0 &= \delta_k \mathbf{d}_k^T A \mathbf{d}_k \\ \delta_k &= \frac{\mathbf{d}_k^T A \mathbf{e}_0}{\mathbf{d}_k^T A \mathbf{d}_k}. \end{aligned}$$

Once again, since the $\mathbf{d}$ vectors are A orthogonal we get

$$\delta_k = \frac{\mathbf{d}_k^T A \left(\mathbf{e}_0 + \sum_{i=0}^{k-1} \alpha_i \mathbf{d}_i \right)}{\mathbf{d}_k^T A \mathbf{d}_k}.$$

By applying (5.13) k times, we can arrive at a final value for δ_k:

$$\begin{aligned}
\delta_k &= \frac{\mathbf{d}_k^T A\left(\mathbf{e}_0 + \alpha_0\mathbf{d}_0 + \alpha_1\mathbf{d}_1 + \cdots + \alpha_{k-1}\mathbf{d}_{k-1}\right)}{\mathbf{d}_k^T A\mathbf{d}_k} \\
&= \frac{\mathbf{d}_k^T A\left(\mathbf{e}_1 + \alpha_1\mathbf{d}_1 + \cdots + \alpha_{k-1}\mathbf{d}_{k-1}\right)}{\mathbf{d}_k^T A\mathbf{d}_k} \\
&\vdots \\
&= \frac{\mathbf{d}_k^T A\mathbf{e}_k}{\mathbf{d}_k^T A\mathbf{d}_k}. \qquad (5.18)
\end{aligned}$$

By (5.15) and (5.18), we find that $\alpha_i = -\delta_i$. Using this fact gives us a new view of the error term. As the following will show, the process of building up $\mathbf{x}$ component by component can be seen as eliminating one component from the error term at each step.

$$\begin{aligned}
\mathbf{e}_i &= \mathbf{e}_0 + \sum_{j=0}^{i-1} \alpha_j\mathbf{d}_j \qquad \Big(\text{by (5.13)}\Big) \\
&= \sum_{j=0}^{n-1} \delta_j\mathbf{d}_j - \sum_{j=0}^{i-1} \delta_j\mathbf{d}_j \qquad \Big(\text{by (5.17)}\Big) \\
&= \sum_{j=i}^{n-1} \delta_j\mathbf{d}_j. \qquad (5.19)
\end{aligned}$$

Thus after n iterations, every component is cut away and so $\mathbf{e}_n = 0$ which implies $\mathbf{x}_n = \mathbf{x}$. □

5.4.2 Generating the Search Directions

Although we mentioned that we need a set of A-orthogonal search directions, we never showed how to find those directions. In this section we will be describing a way to find this set of search directions $\{\mathbf{d}_i\}$ using the *conjugate Gram-Schmidt*

process. The Gram-Schmidt process generates conjugate directions from a set of n linearly independent vectors $\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{n-1}$. To construct $\mathbf{d}_i$, we take $\mathbf{u}_i$ and subtract out any components that are not A-orthogonal to the previous $\mathbf{d}$ vectors (as seen in Figure 5.5). In other words, set $\mathbf{d}_0 = \mathbf{u}_0$, and for $i > 0$ set

$$\mathbf{d}_i = \mathbf{u}_i + \sum_{k=0}^{i-1} \beta_{ik} \mathbf{d}_k, \tag{5.20}$$

where the β_{ik} are defined for $i > k$. Similar to how we found δ_j in (5.18), we can find β_{ij} [17]:

$$\beta_{ij} = -\frac{\mathbf{u}_i^T A \mathbf{d}_j}{\mathbf{d}_j^T A \mathbf{d}_j}. \tag{5.21}$$

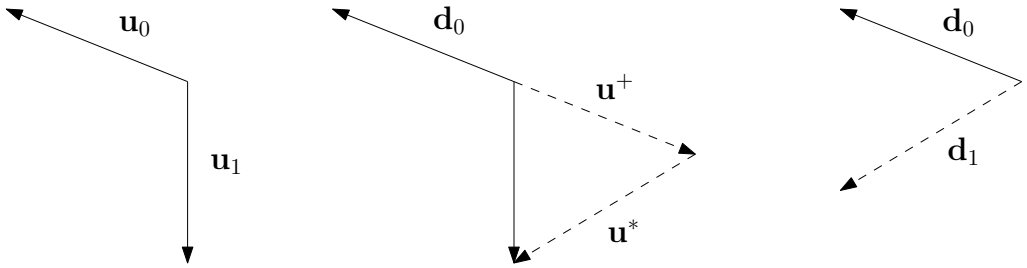


Figure 5.5: An illustration of the conjugate Gram-Schmidt process on two vectors in $\mathbb{R}^2$. We start with two linearly independent vectors $\mathbf{u}_0$ and $\mathbf{u}_1$. Set $\mathbf{d}_0 = \mathbf{u}_0$. Now the vector $\mathbf{u}_1$ is composed of two components: $\mathbf{u}^*$, which is A-orthogonal to $\mathbf{d}_0$, and $\mathbf{u}^+$, which is parallel to $\mathbf{d}_0$. To construct $\mathbf{d}_1$, we subtract out $\mathbf{u}^+$ leaving only the A-orthogonal portion, so $\mathbf{d}_1 = \mathbf{u}^*$.

The main difficulty with using Gram-Schmidt conjugation is that it is computationally intensive. Calculation of β_{ik} takes $O(n^2)$ operations due to the two matrix-vector multiplications. Additionally, previous search directions must be stored to generate the new search directions and a total of $O(n^3)$ operations are required to generate the entire set. Furthermore, β_{ik} and the new search direction $\mathbf{d}_i$ depend on the previous search directions, which means that roundoff errors may be introduced and cause the error vectors $\mathbf{e}_i$ to lose A-orthogonality.

Algorithm 5.1 (Method of Conjugate Directions)

Assumptions: A is a sparse symmetric positive definite matrix.

Input: The matrix A, a vector $\mathbf{b}$, a set of linearly independent vectors $\{\mathbf{u}_i\}$, and an initial guess to the location of the solution $\mathbf{x}_0$.

Output: An approximation to the location of the solution to $A\mathbf{x} = \mathbf{b}$.

1 $\mathbf{d}_0 = \mathbf{u}_0$.
2 **for** $i = 0$ **to** $n - 1$
3 $\quad \mathbf{r}_i = \mathbf{b} - A\mathbf{x}_i$
4 $\quad \alpha_i = (\mathbf{d}_i^T \mathbf{r}_i)/(\mathbf{d}_i^T A\mathbf{d}_i)$
5 $\quad \mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{d}_i$
6 $\quad \mathbf{d}_{i+1} = \mathbf{u}_{i+1} + \sum_{k=0}^{i} \beta_{(i+1)k} \mathbf{d}_k$
7 **end for**
8 **return** $\mathbf{x}_n$

The code for Algorithm 5.1 is available in Appendix C. An interesting property of Conjugate Directions is that if we construct our search directions by conjugation of the axial unit vectors, the method of Conjugate Directions becomes equivalent to that of Gaussian elimination. Thus, it was not until the discovery of the Conjugate Gradient method which made the method of Conjugate Directions become extremely useful [17].

5.5 The Method of Conjugate Gradients

Finally we have the background necessary to begin a discussion on the method of Conjugate Gradients (CG). The CG method aims to improve the method of Conjugate Directions by improving the computation of the search directions. To do this, we will construct the $\mathbf{d}_i$ from residuals $\mathbf{r}_i$ rather than the $\mathbf{u}_i$. This

makes intuitive sense since the residuals worked well for the method of Steepest Descent. Another property that make residuals good to work with is that they are orthogonal to the previous search directions. We can see this by premultiplying (5.19) by $-\mathbf{d}_i^T A$:

$$-\mathbf{d}_i^T A\mathbf{e}_j = -\sum_{k=j}^{n-1} \delta_k \mathbf{d}_i^T A\mathbf{d}_k$$

$$\mathbf{d}_i^T \mathbf{r}_j = 0, \quad \text{for } i < k \quad \text{(by } A\text{-orthogonality of the } \mathbf{d} \text{ vectors).} \tag{5.22}$$

The benefit is that we are always guaranteed a new, linearly independent search direction at each step. If the residual is zero, the problem is solved. With each residual being orthogonal to the previous search direction, it must also be orthogonal to the previous residuals so

$$\mathbf{r}_i^T \mathbf{r}_j = 0, \qquad i \neq j. \tag{5.23}$$

Besides the implications listed above, by choosing to use the residuals, we will show that the CG method will decrease the time complexity of finding the solution $\mathbf{x}$ immensely. To see this, we first notice that we can find the residual using a recurrence:

$$\begin{aligned} \mathbf{r}_{i+1} &= -A\mathbf{e}_{i+1} \\ &= -A(\mathbf{e}_i + \alpha_i \mathbf{d}_i) \qquad \big(\text{by (5.13)}\big) \\ &= \mathbf{r}_i - \alpha_i A\mathbf{d}_i. \end{aligned} \tag{5.24}$$

Let $\mathcal{D}_i$ be the i-dimensional subspace $\text{span}\{\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_{i-1}\}$. Since the search vectors are built from the residuals, the subspace $\text{span}\{\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{i-1}\}$ is equal to

$\mathcal{D}_i$. From (5.24) we can see that each new residual $\mathbf{r}_i$ is just a linear combination of the previous residual and $A\mathbf{d}_{i-1}$. Now given that $\mathbf{d}_{i-1} \in \mathcal{D}_i$, this implies that each new subspace $\mathcal{D}_{i+1}$ is formed from the union of the previous subspace $\mathcal{D}_i$ and the subspace $A\mathcal{D}_i$. Therefore,

$$\begin{aligned}
\mathcal{D}_0 &= \text{span}\{\mathbf{d}_0\} \\
\mathcal{D}_1 &= \mathcal{D}_0 \cup A\mathcal{D}_0 = \text{span}\{\mathbf{d}_0, A\mathbf{d}_0\} \\
\mathcal{D}_2 &= \mathcal{D}_1 \cup A\mathcal{D}_1 = \text{span}\{\mathbf{d}_0, A\mathbf{d}_0, A^2\mathbf{d}_0\} \\
&\vdots \\
\mathcal{D}_i &= \text{span}\{\mathbf{d}_0, A\mathbf{d}_0, A^2\mathbf{d}_0, \ldots, A^{i-1}\mathbf{d}_0\} \\
&= \text{span}\{\mathbf{r}_0, A\mathbf{r}_0, A^2\mathbf{r}_0, \ldots, A^{i-1}\mathbf{r}_0\}.
\end{aligned}$$

The subspace $A\mathcal{D}_i \subseteq \mathcal{D}_i$ is called a *Krylov subspace* since it is formed by repeatedly applying a matrix to a vector. Most modern iterative methods make use of Krylov subspaces and have entire books dedicated to their study [19]. To see why Krylov subspaces makes the CG method efficient, notice that $\mathcal{D}_{i+1}$ contains $A\mathcal{D}_i$. From (5.23) it follows that the next residual $\mathbf{r}_{i+1}$ is orthogonal to $\mathcal{D}_{i+1}$. Using this fact implies $\mathbf{r}_{i+1}$ is A-orthogonal to $\mathcal{D}_i$. The significance of this is that since $\mathbf{r}_{i+1}$ is A-orthogonal to all the previous search directions (besides $\mathbf{d}_i$) and so the time to execute Gram-Schmidt conjugation becomes substantially less due to the fact that we do not need to eliminate the parts that are not A-orthogonal [17].

To see why we get this speed up, first we may use the fact that our linearly independent $\mathbf{u}_i$ are the residuals $\mathbf{r}_i$ and yields,

$$\beta_{ij} = -\frac{\mathbf{r}_i^T A\mathbf{d}_j}{\mathbf{d}_j^T A\mathbf{d}_j}. \tag{5.25}$$

Now consider the inner product of $\mathbf{r}_i$ and $\mathbf{r}_{j+1}$:

$$\mathbf{r}_i^T \mathbf{r}_{j+1} = \mathbf{r}_i^T \mathbf{r}_j + \alpha_j \mathbf{r}_i^T A \mathbf{d}_j \qquad \text{(by (5.24))}$$

$$\alpha_j \mathbf{r}_i^T A \mathbf{d}_j = \mathbf{r}_i^T \mathbf{r}_{j+1} - \mathbf{r}_i^T \mathbf{r}_j$$

$$\mathbf{r}_i^T A \mathbf{d}_j = \begin{cases} \frac{1}{\alpha_i} \mathbf{r}_i^T \mathbf{r}_i, & i = j, \\ -\frac{1}{\alpha_{i-1}} \mathbf{r}_i^T \mathbf{r}_i, & i = j+1, \\ 0, & \text{otherwise.} \end{cases} \qquad \text{(by (5.23))}$$

$$\therefore \beta_{ij} = \begin{cases} \frac{1}{\alpha_{i-1}} \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{d}_{i-1}^T A \mathbf{d}_{i-1}}, & i = j+1, \\ 0, & i > j+1. \end{cases} \qquad \text{(by (5.25))}$$

As seen above, most of the β_{ij} terms have dropped out when compared to the method of Steepest Descent. This is because the search directions $\mathbf{d}_j$ with $j < i-1$ are not needed to make $\mathbf{d}_i$ A-orthogonal to all the previous search directions. Furthermore, it is no longer necessary to store the previous search directions. Thus, we can see the importance of using the CG method with residuals because we have reduced the time and space complexity from $O(n^2)$ to $O(m)$ where m is the number of non-zero entries in the matrix A [17].

Before we continue, recall that we chose the $\mathbf{u}_i$ to be the residuals $\mathbf{r}_i$. Then we have the identity [17]:

$$\mathbf{d}_i^T \mathbf{r}_i = \mathbf{u}_i^T \mathbf{r}_i = \mathbf{r}_i^T \mathbf{r}_i. \tag{5.26}$$

To simplify our notation, we let $\beta_i = \beta_{i,i-1}$ and it follows that

$$\begin{aligned}
\beta_i &= \left(\frac{1}{\alpha_{i-1}}\right)\left(\frac{\mathbf{r}_i^T\mathbf{r}_i}{\mathbf{d}_{i-1}^T A\mathbf{d}_{i-1}}\right) \\
&= \left(\frac{\mathbf{d}_{i-1}^T A\mathbf{d}_{i-1}}{\mathbf{d}_{i-1}^T\mathbf{r}_{i-1}}\right)\left(\frac{\mathbf{r}_i^T\mathbf{r}_i}{\mathbf{d}_{i-1}^T A\mathbf{d}_{i-1}}\right) \qquad \Big(\text{by (5.16)}\Big) \\
&= \frac{\mathbf{r}_i^T\mathbf{r}_i}{\mathbf{d}_{i-1}^T\mathbf{r}_{i-1}} \\
&= \frac{\mathbf{r}_i^T\mathbf{r}_i}{\mathbf{r}_{i-1}^T\mathbf{r}_{i-1}} \qquad \Big(\text{by (5.26)}\Big).
\end{aligned}$$

We conclude this section with the algorithm for the Conjugate Gradient method. There are many different derivations of the method found in literature ([19, Chapter 5],[4, Lecture 38], [8, 6.63]) but the algorithm is essentially the same.

Algorithm 5.2 (Conjugate Gradient Method)

Assumptions: A is a sparse symmetric positive definite matrix.

Input: The matrix A, a vector $\mathbf{b}$, an initial guess to the location of the solution $\mathbf{x}_0$, and an upper bound on the number of iterations $i_{\max}$.

Output: An approximation to the location of the solution to $A\mathbf{x} = \mathbf{b}$.

1 $\mathbf{d}_0 = \mathbf{r}_0 = b - A\mathbf{x}_0$
2 $n = i_{\max}$
3 **for** $i = 0$ **to** n
4 $\quad \alpha_i = \langle \mathbf{r}_i, \mathbf{r}_i\rangle / \langle A\mathbf{d}_i, \mathbf{d}_i\rangle \qquad \Big(\text{from (5.16) and (5.26)}\Big)$
5 $\quad \mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{d}_i$
6 $\quad \mathbf{r}_{i+1} = \mathbf{r}_i - \alpha_i A\mathbf{d}_i$
7 $\quad$ **if** termination criterion is satisfied **then** exit loop
8 $\quad \beta_{i+1} = \langle \mathbf{r}_{i+1}, \mathbf{r}_{i+1}\rangle / \langle \mathbf{r}_i, \mathbf{r}_i\rangle$
9 $\quad \mathbf{d}_{i+1} = \mathbf{r}_{i+1} + \beta_{i+1}\mathbf{d}_i$

10 **end for**

11 **return** $\mathbf{x}_n$

At each round of the **for** loop the matrix-vector product $A\mathbf{d}_i$ needs to only be computed once. Figure 5.6 illustrates the *Mathematica* code we have included in Appendix C applied to our example from (5.6). What remains is some type of termination criterion for Algorithm 5.2

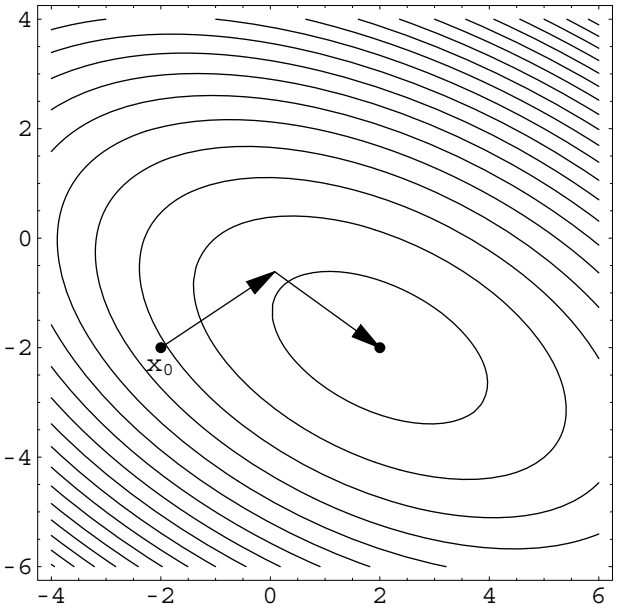


Figure 5.6: The Conjugate Gradient method

5.5.1 Stopping Criteria

Like the method of Conjugate Directions, the CG method converges in n iterations. However for extremely large matrices, it is not feasible to run n iterations due to the large roundoff error that will occur. This error causes the search vectors to lose A-orthogonality. Therefore, we need to do some type of convergence analysis to see how many iterations are necessary to get at least 10 digits of the solution to our

problem correct. Later, we will be using interval analysis to show that the residuals we calculate provide sufficient information to do reliable error estimation.

We begin our analysis by introducing a few new definitions. The *spectral radius* of a matrix A is

$$\rho(A) = \max|\lambda_i|, \qquad \lambda_i \text{ is an eigenvalue of } M.$$

When dealing with positive definite matrices, the *energy norm* of a vector $\mathbf{e}$ defined by $\|\mathbf{e}\|_A = (\mathbf{e}^T A\mathbf{e})^{1/2} = \sqrt{\langle A\mathbf{e}, \mathbf{e}\rangle}$ is much easier to work with than the Euclidean norm. The CG method minimizes the energy norm of the error term, $\|\mathbf{e}_i\|_A$ at each step [4, Theorem 38.2]. Thus, an important characteristic of the energy norm is that minimizing $\|\mathbf{e}_i\|_A$ is equivalent to minimizing the quadratic form $f(\mathbf{x}_i)$. The *spectral condition number* is defined to be

$$\kappa(A) = \lambda_{\max}(A)/\lambda_{\min}(A),$$

the ratio of the largest to smallest eigenvalue. An *ill-conditioned* matrix is one in which the condition number is large. Given a matrix with a large condition number, the CG method will converge slowly. Finally, the eigenvalues of a matrix A are called its *spectrum*.

The properties of convergence of the CG method are determined by the spectrum of A. This is seen through the following Corollary which says that after k iterations, the CG method will converge to a solution within the tolerance ε.

Corollary 5.3 ([9, Cor. 8.18])**.** *In order to reduce the error in the energy norm by a factor of ε, i.e.,*

$$\|\mathbf{x} - \mathbf{x}_k\|_A \leq \varepsilon\|\mathbf{x}\|_A,$$

at most k CG iterations are needed, where k is the smallest integer such that

$$k \geq \left\lceil \frac{\sqrt{\kappa(A)}}{2} \ln(2/\varepsilon) \right\rceil.$$

Let us take a look at our SIAM problem. Since the diagonal of A_n is dominant for at least the lower half, then a rough approximation of the condition number is given by the ratio of the largest to the smallest value of the diagonal,

$$\kappa(A_n) \approx \frac{p_n}{2} = \frac{224737}{2} \approx 112369 \approx 10^5.$$

By using numerical eigenvalue routines, one can improve the approximation of $\kappa(A)$ to be approximately $2 \cdot 10^5$ [6]. Therefore, to obtain an accuracy of 10 digits we must choose $\varepsilon = 10^{-11}$ and apply Corollary 5.3 to find

$$k \geq \left\lceil \frac{\sqrt{2 \cdot 10^5}}{2} \ln\left(2/10^{-11}\right) \right\rceil = 5819.$$

Although choosing $k \geq 5819$ will guarantee us 10 digits of accuracy, it may be an overestimate.

To find a termination criterion, we will put a bound on $\|\mathbf{r}_k\|$. Using the results from [6, Lemma 7.1] which shows $1 \leq \lambda_{\min}(A) \leq 2$, it follows that

$$\langle \mathbf{x}, \mathbf{x} \rangle \leq \langle A\mathbf{x}, \mathbf{x} \rangle \leq \langle A^2\mathbf{x}, \mathbf{x} \rangle = \langle A\mathbf{x}, A\mathbf{x} \rangle.$$

or in other words,

$$\|\mathbf{x}\| \leq \|\mathbf{x}\|_A \leq \|A\mathbf{x}\|.$$

If we let $\hat{\mathbf{x}}$ denote the $(1,1)$ entry of A^{-1}, then for $\mathbf{r}_k = b - A\mathbf{x}_k$ we have

$$|\hat{\mathbf{x}} - \hat{\mathbf{x}}_k| \leq \|\mathbf{x} - \mathbf{x}_k\| \leq \|\mathbf{x} - \mathbf{x}_k\|_A \leq \|A\mathbf{x} - A\mathbf{x}_k\| = \|b - A\mathbf{x}_k\| = \|\mathbf{r}_k\|. \tag{5.27}$$

So by choosing $\|\mathbf{r}_k\| \leq 10^{-11}$ as our termination criterion, we are guaranteed that our approximate solution will have 10 digits of accuracy. It is customary to stop when the norm of the residual falls below a specified value and often this value is some small faction of the initial residue; that is, $\|\mathbf{r}_i\| \leq \varepsilon\|\mathbf{r}_0\|$ [17].

Like the method of Steepest Descent, the CG method *theoretically* converges in n steps. However due to floating-point error, this guarantee is void, making it impossible to stop when the residual is zero. One method of reducing the error is by recomputing the residual using (5.11a) every few iterations. The most efficient method is by using a trick called *preconditioning* [4, Lecture 40].

5.6 Preconditioned Conjugate Gradient

To improve the convergence speed of the CG method, we need to make the condition number for the matrix A smaller. Geometrically, we want to transform the level surfaces of our quadratic form, which in general are ellipsoids (Figure 5.2(b)), to become as close as possible to spheres. By doing so it clusters the eigenvalues closer together. Consider a symmetric positive definite matrix M that is easy to invert with the property that $M^{-1} \approx A^{-1}$; that is, a matrix M such that $\kappa(M^{-1}A) \approx 1$. Then given M, we can indirectly solve $A\mathbf{x} = \mathbf{b}$ by applying the CG method to the system

$$M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}. \tag{5.28}$$

The problem that occurs is that $M^{-1}A$ is generally not symmetric nor positive definite, even if M and A are. To combat this problem, recall that property (5.4) implies there exists a matrix E such that $M = EE^T$. To transform the linear system in (5.28) into a system that we can apply the CG method to, a matrix that has similar eigenvalues to $M^{-1}A$ must be found. The claim below shows that the matrix $E^{-1}AE^{-T}$ is sufficient.

Claim 1. The matrices $M^{-1}A$ and $E^{-1}AE^{-T}$ have the same eigenvalues.

Proof. Let $\mathbf{v}$ be an eigenvector of $M^{-1}A$ with corresponding eigenvalue λ. For the matrices $M^{-1}A$ and $E^{-1}AE^{-T}$ to have the same eigenvalues, $E^T\mathbf{v}$ must be an eigenvector of $E^{-1}AE^{-T}$ with eigenvalue λ:

$$\begin{aligned}(E^{-1}AE^{-T})(E^T\mathbf{v}) &= E^{-1}A\mathbf{v} \\ &= (E^TE^{-T})E^{-1}A\mathbf{v} \\ &= E^TM^{-1}A\mathbf{v} \qquad (\text{since } M^{-1} = E^{-T}E^{-1}) \\ &= \lambda E^T\mathbf{v} \qquad (\text{since } M^{-1}A\mathbf{v} = \lambda\mathbf{v}).\end{aligned}$$

Therefore, the matrices $M^{-1}A$ and $E^{-1}AE^{-T}$ have the same eigenvalues as desired. □

Using Claim 1 the system $A\mathbf{x} = \mathbf{b}$ can be transformed into the problem

$$E^{-1}AE^{-T}\tilde{\mathbf{x}} = E^{-1}\mathbf{b}, \qquad \tilde{\mathbf{x}} = E^T\mathbf{x}, \tag{5.29}$$

which we solve first for $\tilde{\mathbf{x}}$, then for $\mathbf{x}$. Since $E^{-1}AE^{-T}$ is a symmetric positive definite matrix, we may use the CG method on it. The process of using CG to solve (5.29) is called the *Transformed Preconditioned Conjugate Gradient Method* [17]. The algorithm below is just a reformulation of Algorithm 5.2.

Algorithm 5.3 (Transformed Preconditioned Conjugate Gradient Method)

Assumptions: A is a sparse symmetric positive definite matrix.

Input: The matrix A, a vector $\mathbf{b}$, an initial guess to the location of the solution $\tilde{\mathbf{x}}_0$, a factorization of M such that $M = EE^T$, a maximum tolerance ε, and an upper bound on the number of iterations $i_{\max}$.

Output: An approximation to the location of the solution to $A\mathbf{x} = \mathbf{b}$.

1 $\tilde{\mathbf{d}}_0 = \tilde{\mathbf{r}}_0 = E^{-1}\mathbf{b} - E^{-1}AE^{-T}\tilde{\mathbf{x}}_0$
2 $n = i_{\max}$
3 **for** $i = 0$ **to** n
4 $\quad \alpha_i = \langle \tilde{\mathbf{r}}_i, \tilde{\mathbf{r}}_i \rangle / \langle E^{-1}AE^{-T}\tilde{\mathbf{d}}_i, \tilde{\mathbf{d}}_i \rangle$
5 $\quad \tilde{\mathbf{x}}_{i+1} = \tilde{\mathbf{x}}_i + \alpha_i \tilde{\mathbf{d}}_i$
6 $\quad \tilde{\mathbf{r}}_{i+1} = \tilde{\mathbf{r}}_i - \alpha_i E^{-1}AE^{-T}\tilde{\mathbf{d}}_i$
7 $\quad$ **if** $\|\mathbf{r}_{i+1}\| \leq \varepsilon$ **then** exit loop
8 $\quad \beta_{i+1} = \langle \tilde{\mathbf{r}}_{i+1}, \tilde{\mathbf{r}}_{i+1} \rangle / \langle \tilde{\mathbf{r}}_i, \tilde{\mathbf{r}}_i \rangle$
9 $\quad \tilde{\mathbf{d}}_{i+1} = \tilde{\mathbf{r}}_{i+1} + \beta_{i+1}\tilde{\mathbf{d}}_i$
10 **end for**
11 **return** $\mathbf{x} = E^{-T}\tilde{\mathbf{x}}_n$

There is no wrong choice for the starting value for $\tilde{\mathbf{x}}_0$. If you have a rough approximation for the value $\mathbf{x}$, use it as the value for $\tilde{\mathbf{x}}_0$, otherwise let $\tilde{\mathbf{x}}_0 = 0$. When the PCG method is used to solve a linear system, it will always converge.

The drawback of using Algorithm 5.3 is that E must be computed using some type of factorization routine (Cholesky factorization or reverse Cuthill-McKee). By making clever substitutions we are able to eliminate E. By letting $\tilde{\mathbf{r}}_i = E^{-1}\mathbf{r}_i$ and $\tilde{\mathbf{d}}_i = E^T\mathbf{d}_i$ and using the identities $\tilde{\mathbf{x}}_i = E^T\mathbf{x}_i$ and $E^{-T}E^{-1} = M^{-1}$, we derive the *Untransformed Preconditioned Conjugate Gradient Method* [17]:

Algorithm 5.4 (Untransformed Preconditioned Conjugate Gradient Method)

Assumptions: A is a sparse symmetric positive definite matrix.

Input: The matrix A, a vector $\mathbf{b}$, an initial guess to the location of the solution $\mathbf{x}_0$, a preconditioner M, a maximum tolerance ε, and an upper bound on the number of iterations $i_{\max}$.

Output: An approximation to the location of the solution to $A\mathbf{x} = \mathbf{b}$.

1 $\mathbf{r}_0 = b - A\mathbf{x}_0$
2 $\mathbf{d}_0 = M^{-1}\mathbf{r}_0$
3 $n = i_{\max}$
4 **for** $i = 0$ **to** n
5 $\quad \alpha_i = \langle M^{-1}\mathbf{r}_i, \mathbf{r}_i\rangle / \langle A\mathbf{d}_i, \mathbf{d}_i\rangle$
6 $\quad \mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{d}_i$
7 $\quad \mathbf{r}_{i+1} = \mathbf{r}_i - \alpha_i A\mathbf{d}_i$
8 $\quad$ **if** $\|\mathbf{r}_{i+1}\| \leq \varepsilon$ **then** exit loop
9 $\quad \beta_{i+1} = \langle M^{-1}\mathbf{r}_{i+1}, \mathbf{r}_{i+1}\rangle / \langle M^{-1}\mathbf{r}_i, \mathbf{r}_i\rangle$
10 $\quad \mathbf{d}_{i+1} = M^{-1}\mathbf{r}_{i+1} + \beta_{i+1}\mathbf{d}_i$
11 **end for**
12 **return** $\mathbf{x}_n$

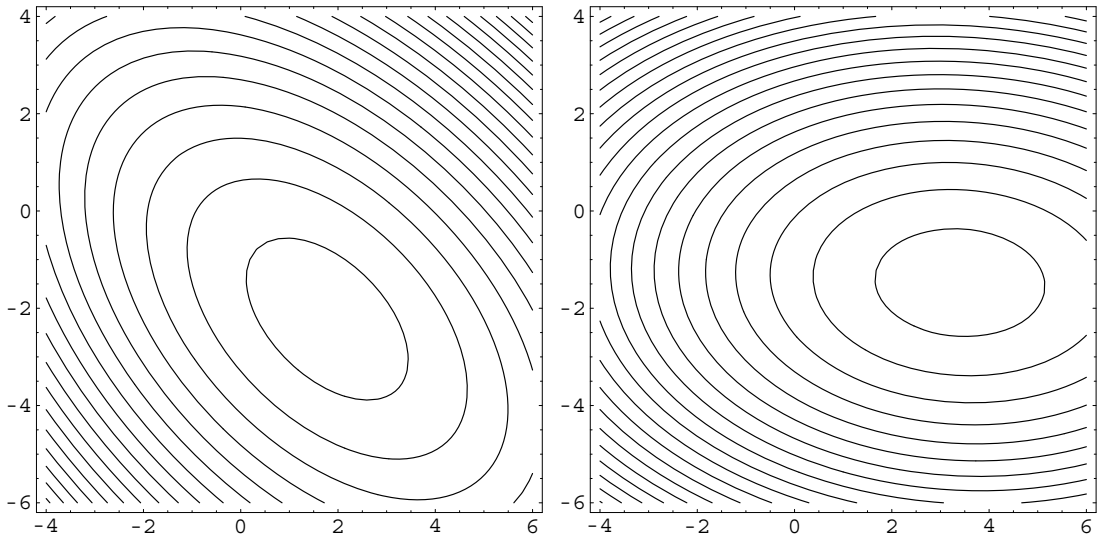
The effectiveness of a preconditioner M is determined by the condition number of $M^{-1}A$. The ideal matrix M will also have the property that the matrix-vector product $M\mathbf{r}_i$ is not too difficult to compute. The problem remains to find an appropriate preconditioner that will make $\kappa(M^{-1}A) \approx 1$. The perfect choice for a preconditioner is $M = A$ which gives us $\kappa(M^{-1}A) = 1$. Unfortunately, the preconditioning step is solving the system $M\mathbf{x} = \mathbf{b}$ which has the same time complexity as the original problem. If $M = I_n$, then nothing has been gained since we are solving the original problem. Between these two extremes lie the useful preconditioners. The simplest choice for a preconditioner is $M = D$ where D is the

matrix of the diagonal entries of A. The act of applying D to the iterative method is called *diagonal scaling* or *Jacobi preconditioning* [4]. The diagonal matrix is trivial to invert, but often is not a good preconditioner.

A more efficient preconditioner uses *incomplete Cholesky factorization* [4]. Normal Cholesky factorization gives an lower-triangular matrix L such that $A = L^T L$. By computing the Cholesky factorization, one runs the risk of eliminating zeros and making L not very sparse. The idea of incomplete Cholesky factorization is that we compute a matrix $\hat{L}$ that minimizes *fill-in*; that is, minimizes the amount of nonzero entries created in the process of calculating $\hat{L}$. One can do so by using Cholesky-like methods with the restriction that the resulting matrix $\hat{L}$ have the same pattern of nonzero elements of A. An example of this would be using Reverse Cuthill-McKee ordering [6]. Unfortunately, Cholesky preconditioning is not always stable.

Figure 5.7 illustrates the results of preconditioning when applied to our sample problem in (5.6). Comparing the results with Figure 5.2a, we can see some improvement in the shape of the ellipses. The condition number for the sample problem was originally 3.5, but by using diagonal and Cholesky preconditioning the condition numbers were reduced to 2.78 and 1.24, respectively.

Let us once again consider the problem at hand. Earlier we found that the condition number for our matrix A was $2 \cdot 10^5$. By using the diagonal preconditioner D, the condition number $\kappa(D^{-1}A)$ reduces to approximately 4.45 [6]. The application of Corollary 5.3 using $\kappa = 4.45$ shows that $k \geq 28$ iterations are sufficient to get 10 digits of accuracy. The code in Appendix C shows sample runs of the PCG method using the diagonal preconditioner.



(a) Diagonal preconditioning. **(b)** Cholesky preconditioning.

Figure 5.7: Contours of the quadratic form of our sample problem after preconditioning.

A *Mathematica* Session

Due to the large memory constraints posed by the PCG method, my personal computer was unable to calculate an approximation to $\mathbf{x}$. Fortunately, *Mathematica* has an efficient PCG solver built into `LinearSolve`. Using the definition of `A` and `b` as shown in the session on page 64, we can calculate the $(1,1)$ entry to 50 digit precision in only a few seconds:

```
diagonal = Table[A[[i, i]], {i, n}];
prec = 50;
b = SetPrecision[b, prec + 5];
x = LinearSolve[A, b, Method → {Krylov, Method → ConjugateGradient,
      Preconditioner → (# / diagonal &), Tolerance → 10^(-prec-1)}];

N[x[[1]], 50]

0.72507834626840116746868771925116096886918059447951
```

5.7 Interval Arithmetic

Using the PCG method we are able to get 10000 digits of accuracy using 2903 iterations and 5.3 days of computing [6]. Like most algorithms in computational mathematics, the PCG method does not provide a proof of correctness. We will be using interval arithmetic to validate the calculated solution. Our general problem is solving the linear system $A\mathbf{x} = \mathbf{b}$. Assume we have a calculated vector $\hat{\mathbf{x}}$ and we want to estimate the error $\|\mathbf{x} - \hat{\mathbf{x}}\|$. By using a residual based estimate, the error can be enclosed using intervals. The reason is because both direct methods and iterative methods such as the PCG method tend to produce small residuals. A fairly simple estimate can be derived just from our problem statement:

$$\|\mathbf{x} - \hat{\mathbf{x}}\| = \|A^{-1} \underbrace{(\mathbf{b} - A\hat{\mathbf{x}})}_{\mathbf{r}} \| \leq \|A^{-1}\| \cdot \|\mathbf{r}\|. \tag{5.30}$$

Using the fact that $1 \leq \lambda_{\min}(A_n) \leq 2$ then

$$\|A_n^{-1}\| = 1/\lambda_{\min}(A_n) \leq 1$$

where $\|\cdot\|$ denotes the spectral norm of A [6]. Using the previous statement, we simplify (5.30) to find:

$$\|\mathbf{x} - \hat{\mathbf{x}}\| \leq \|A_n^{-1}\| \cdot \|\mathbf{r}\| \leq \|\mathbf{r}\|.$$

Therefore, to validate the error we need only to calculate the inclusion of $\|\mathbf{r}\| = \|\mathbf{b} - A_n\hat{\mathbf{x}}\|$ using intervals; that is, $\hat{\mathbf{x}}$ is the midpoint of our interval enclosed by $\|\mathbf{b} - A_n\hat{\mathbf{x}}\|$. Using *Mathematica*'s built in interval arithmetic, one can find the interval enclosure of the result on page 92 in only a few seconds:

A *Mathematica* Session

```
x[[1]] + Interval[{-1, 1}] Norm[b - A.Interval /@ x] // IntervalForm
```

$$0.725078346268401167468687719251160968869180594479\underset{82162}{\overset{96996}{50}}$$

As a concluding note, it is interesting to mention that this SIAM problem is the only one to be solved exactly. The LinBox team used a cluster of 182 processors running in parallel for about 4 days to come up with an exact rational solution that has 97,389 digits in both the numerator and denominator! However, in 2005 Zhendong Wan created an algorithm that substantially reduced the run time to only 25 minutes [7].

CHAPTER 6

Conclusion

The SIAM 100-Digit challenge provides an extremely broad look into the field of numerical analysis with topics like matrix computation, integration strategies, partial differential equations, optimization, error control, interval analysis, and high-precision arithmetic. With the help of *Mathematica* I have successfully solved three of the ten SIAM problems. By making use of interval analysis, the algorithms designed provided verifiably correct solutions without the need for sensitivity analysis. I focused on solutions to these problems using interval analysis specifically to understand what it brings to the world of scientific computing. Interval analysis is a great tool for computational mathematics but can not be used to solve every problem. Like the definitions of numerical analysis introduced in Chapter 1, interval analysis has its share of ambiguous definitions. Although I tried my best to introduce the subject of interval analysis to the reader, there are many more topics to explore. Since interval arithmetic is just an extension of real arithmetic, mathematical fields such as topology and analysis can be used as tools in studying the realm of intervals.

With computer technology becoming more advanced, the problems of the past like memory limitations and rounding error seem to be fading into extinction. On

the other hand, numerical analysis will persevere and continue its study of algorithms in continuous mathematics. A logical step for computer scientists and mathematicians in the future is to make the implementation of interval analysis in hardware more efficient. After decades of study of floating-point arithmetic, interval arithmetic has had relatively little exposure and can be improved upon greatly. Interval analysis can also be improved at the software level as well. The number of built in interval arithmetic routines in *Mathematica* are relatively sparse. In the future, I would like to see Wolfram add more functionality.

Although I only solved three problems for the SIAM 100-Digit challenge, I would like to solve the other seven problems. I hope to continue on to study computational mathematics and by finishing the other problems it would introduce me to new topics in numerical analysis. Through this Independent Study, I have become a more efficient programmer. The three problems I looked at challenged both my reasoning and programming skills and encouraged me to excel as a researcher. The year long experience has prepared me for graduate research and provides a great introduction to the world of scientific computing. In the future I forsee myself studying interval analysis as a possible thesis topic.

APPENDIX A

Chapter 3 Code

This appendix lists the *Mathematica* code for Chapter 3.

Basic Initializations

Pretty Print IntervalForm from The SIAM 100 Digit Challenge Book:

```
DigitsAgreeCount[a_, b_] := (prec = Ceiling@Min[Precision /@ {a, b}];
   {{ad, ae}, {bd, be}} = RealDigits[#, 10, prec] & /@ {a, b};
   If[ae ≠ be ∨ a b ≤ 0, Return[0]]; If[ad == bd, Return@Length[ad]];
   {{com}} = Position[MapThread[Equal, {ad, bd}], False, 1, 1] - 1; com);
DigitsAgreeCount[Interval[{a_, b_}]] := DigitsAgreeCount[a, b];

IntervalForm[Interval[{a_, b_}]] :=
 (If[(com = DigitsAgreeCount[a, b]) == 0, Return@Interval[{a, b}]];
  start = Sign[a] N[FromDigits@{ad[[Range@com]], 1}, com];
  {low, up} = SequenceForm @@ Take[#, {com + 1, prec}] & /@ {ad, bd};
  If[ae == 0, start /= 10; ae++]; SequenceForm[
   DisplayForm@SubsuperscriptBox[NumberForm@start, low, up], If[ae ≠ 1,
    Sequence @@ {" × ", DisplayForm@SuperscriptBox[10, ae - 1]}, ""]])
```

The SIAM book interval solution (not using Newton / Krawczyk)

```
iMin[{Interval[{a_, b_}], Interval[{c_, d_}]}] :=
  Interval[{Min[a, c], Min[b, d]}];
iMin[{}] := ∞;
RealToInterval[r_, d_] := Interval[r + {-1, 1} d];
diam[Interval[{a_, b_}]] := b - a;
diam[{i__Interval}] := Max[diam /@ {i}];
h[{a_, b_}] := 9 {{b^2 - a^2, -2 a b}, {-2 a b, a^2 - b^2}};

ReliableTrajectory::usage =
  "ReliableTrajectory[p, v, targettime, (opts)] gives a set of {time,
    position} intervals that give the reflection points and the times
    they are there. The last position is an interval of diameter
    less than 10^-ag, where ag is the AccuracyGoal setting.";

StartIntervalPrecision::usage =
  "StartIntervalPrecision is an option to ReliableTrajectory that
    sets the starting size of the intervals. A setting of automatic
    means that the accuracy goal is used for the initial size.";

IntervalPrecisionStep::usage =
  "IntervalPrecisionStep is an option to ReliableTrajectory
    that sets the number of powers of 10 by which
    the interval size is decreased after a failure.";

ShowSize::usage = "ShowSize is an option to ReliableTrajectory that asks
    that the initial condition interval size that works be printed.";
```

```
Options[ReliableTrajectory] := {StartIntervalPrecision → Automatic,
   AccuracyGoal → 12, IntervalPrecisionStep → 1, ShowSize → False};

ReliableTrajectory[p_, v_, targettime_, opts___Rule] :=
  Module[{lastpoint = Interval[{-∞, ∞}], stint, ips,
    trQ, pint, intsize, t, resttime, pathdata, S, T},
   {ag, stint, ips, trQ} = {AccuracyGoal, StartIntervalPrecision,
       IntervalPrecisionStep, ShowSize} /.
      {opts} /. Options[ReliableTrajectory];
   If[stint === Automatic, stint = ag];
   intsize = stint; wp = Max[17, 2 + intsize];
   While[diam[lastpoint] > 10^-ag ,
    pint = N[(RealToInterval[#, 10^-intsize] &) /@ p, wp];
    vint = N[(RealToInterval[#, 10^-intsize] &) /@ v, wp];
    pathdata = {N[pint, wp]};
    resttime = Interval[{targettime, targettime}];

    While[resttime > 0, m = Round[pint + 2 / 3 vint];
     S = t /. (Solve[(pint + t vint - m).(pint + t vint - m) == 1 / 9, t]);
     If[FreeQ[S, Power[Interval[{_?Negative, _?Positive}], _]],
          T = iMin[Cases[S, _?Positive]], Break[]];
     Which[
      T ≤ resttime,
       pint += T vint; vint = h[pint - m].vint; resttime -= T,
      T > resttime && resttime ≥ 2 / 3, pint += 2 / 3 vint; resttime -= 2 / 3,
      T > resttime && resttime < 2 / 3, pint += resttime vint; resttime = 0,
      True, Break[]];
     AppendTo[pathdata, pint];
     If[Precision [{resttime, pint, vint, T}] < ag, Break[]]];

    intsize += ips; wp = Max[17, intsize + 2];
    lastpoint = pint + Table[
       RealToInterval[-Max[Abs[resttime]], Max[Abs[resttime]]], {2}]];
   If[trQ, Print[StringForm["Initial condition interval radius is 10^``.",
      intsize]]];
   pathdata];
```

Estimating the Photon's Path

If we consider the lattice of integer points then take unit squares centered at each lattice point, then these squares divide each photon ray into segments. For example:

```
<< Graphics`Arrow`
(* 9 mirrors in [-0.5,2.5]×[-0.5,2.5] *)
mirrors = Table[Circle[{i, j}, 1 / 3], {i, 0, 2}, {j, 0, 2}];
(* 9 centers for each mirror *)
centers = Table[Point[{i, j}], {i, 0, 2}, {j, 0, 2}];
(* Starting point for our example ray and a
  big circular dot to make it visible with a label *)
p = {0.35, -0.3};
point = Circle[p, 0.03];
plabel = Text["P", p - {.1, .1}, TextStyle → {FontSize → 15}];
(* A simple line to illustrate the path of our ray y =
  1.1 x - 0.685. Each segment has a different Dashing *)
line1 = Plot[2.6 x - 1.21, {x, 0.365, 0.5}, Axes → False,
   AxesOrigin → {-1 / 2, -1 / 2}, PlotRange → {{-.6, 2.6}, {-.6, 2.8}},
   AspectRatio → Automatic, DisplayFunction → Identity];
line2 = Plot[2.6 x - 1.21, {x, 0.5, 0.6576923076923077`}, Axes → False,
   AxesOrigin → {-1 / 2, -1 / 2}, PlotRange → {{-.6, 2.6}, {-.6, 2.8}},
   PlotStyle → {Dashing[{0.01, 0.015}]},
   AspectRatio → Automatic, DisplayFunction → Identity];
line3 = Plot[2.6 x - 1.21, {x, 0.6576923076923077`, 1.0423076923076922`},
   Axes → False, AxesOrigin → {-1 / 2, -1 / 2},
   PlotRange → {{-.6, 2.6}, {-.6, 2.8}},
   AspectRatio → Automatic, DisplayFunction → Identity];
line4 = Plot[2.6 x - 1.21, {x, 1.0423076923076922`, 1.426923076923077`},
   Axes → False, AxesOrigin → {-1 / 2, -1 / 2}, PlotRange →
    {{-.6, 2.6}, {-.6, 2.8}}, PlotStyle → {Dashing[{0.01, 0.015}]},
   AspectRatio → Automatic, DisplayFunction → Identity];
(* An arrow to finish it off *)
arrow = Arrow[{1.426923076923077`, 2.5},
   {1.5, 2.69}, HeadWidth → 0.4, HeadLength → 0.04];
(* Display all the items above in a grid with some extra formatting *)
Show[Graphics[{mirrors, centers, point, plabel, arrow},
  AspectRatio → Automatic, Axes → True, AxesOrigin → {-1 / 2, -1 / 2},
  PlotRange → {{-.5, 2.65}, {-.5, 2.65}},
  Ticks → {Range[0, 2.5, .5], Range[0, 2.5, .5]},
  GridLines → {{0.5, 1.5, 2.5}, {0.5, 1.5, 2.5}}], line1, line2, line3, line4]
```

We needed to solve for when our line reaches the top part of the box:

```
Table[Solve[y == 2.6 x - 1.21, x], {y, 0.5, 2.5, 1}]
```

```
{{{x → 0.657692}}, {{x → 1.04231}}, {{x → 1.42692}}}
```

If a ray intersects a mirror inside a square it must have length at least $\sqrt{2} - \frac{2}{3}$:

```
pp = {(-1 / 3 √2 / 2), (1 / 3 √2 / 2)};
Show[Graphics[{Circle[{0, 0}, 1 / 3], PointSize[0.03], Point[{0, 0}],
   Point[√2 / 2 {-1, 1} / 3], Line[{{0, 0}, pp, {-.5, .5}}],
   Text["1/3", {-.05, .11}, {0, -1}],
   Text["√2/2 - 1/3", {-.28, .42}, {0, 0}],
   Text["√2 - 2/3", {.3, -.3}, {0, 0}],
   Line[{{-.5, -.5}, {.5, -.5}, {.5, .5}, {-.5, .5}, {-.5, -.5}}],
   Line[{{-.5, 1/6 (-3 + 2 √2)}, {1/6 (3 - 2 √2), .5}}],
   Line[{{.5, 1/6 (3 - 2 √2)}, {1/6 (-3 + 2 √2), -.5}}],
   PointSize[.03]}], AspectRatio → Automatic,
 TextStyle → {FontFamily → "Times", FontSize → 12}]
```

What would be a better lower bound than $2/3$ since $2/3$ is not finitely representable in binary?

```
BaseForm[2 / 3 // N, 2]
```

0.10101010101010101011_2

Well we need something less than 0.747547

```
Sqrt[2] - 2 / 3 // N
```

```
0.747547
```

Maybe 11/16?

```
11 / 16 // N
BaseForm[%, 2]
```

```
0.6875
```

0.1011_2

■ Algorithm 3.1

This does not use interval arithmetic so we can only get a fixed amount of digits depending on our starting precision:

```
(* Initialize our transformation matrix. Given the photons position p,
 and its velocity vector v, we can determine its new direction *)
H[{a_, b_}] := 9 ( b^2 - a^2   -2 a b
                   -2 a b   a^2 - b^2 );
 (* We begin with n-digit precision *)
 startPrec = 43;
 (* Our initial point is p. We use N to use n-digit precision *)
 p = N[{1/2, 1/10}, startPrec];
 (* Our initial velocity is v. We don't use n-digit precision here
    since our error is dependent on the position of the photon p *)
 v =
   {1,
    0};
```

Begin Algorithm 3.1:

```
(* STEP 1 *)
(* The amount of time remaining is timeRem *)
timeRem = 10;
path = {p};
```

```
(* STEP 2 *)
(* While there is still time left *)
While[timeRem > 0,
     (* Assume for this iteration that m is the
   midpoint of the mirror the photons path intersects with *)
     m = Round[p + 2/3 v];
     (* Find the smallest positive root. If s ≠
   ∞ then s is equal to the time it takes for the photon
    to reach the intersection point with the mirror,
  otherwise the photon does not intersect the mirror *)
     s = Min[Cases[t /. Solve[(p + t v - m).(p + t v - m) == 1/9, t],
    _?Positive]];
     (* If the time it takes to reach the next intersection is less
   than the time remaining then that means there is an intersection *)
     If[s < timeRem,
          (* p is now set to be the point at which it intersects with
    the mirror and v is determined by our transformation matrix H *)
          p += s v;
          v = H[p - m].v,
             (* Else there was no intersection,
   so there are two things that may have
     happened: (1) The photon stops in mid-flight at timeRem =
    10. In this case we let s = timeRem and thus p has reached
      its final destination, or (2) the photon has sufficient time,
   but the mirror we chose initially as determined by m was
    not correct. In this case we let s be 2/3 since we
    will choose the next mirror in the photon's path *)
       s = Min[timeRem, 2/3];
          p += s v
       ];
     (* STEP 3 *)
     (* Reduce timeRem by s *)
     timeRem -= s;
     (* Append p to path *)
     path = Append[path, p];
];
```

```
Print["The final position of our photon = ", Last[path]];
Print["Distance from p to the origin = ", answer = Norm[p]];
Print["Number of digits Mathematica claims to have correct = ",
 Precision[answer]]
```

```
The final position of our photon = {-0.73629269861, -0.66964269636}

Distance from p to the origin = 0.9952629194

Number of digits Mathematica claims to have correct = 10.9179
```

■ Software Precision Trajectories

■ Paths for Various Precisions

```
path29 = {{0.5`29., 0.1`29.},
   {0.68202026619435145028245947134049832193`27.979591477348777,
    0.1`29.},
   {0.13535359952768478361579280473071378559`26.593088199488765,
    0.48157568056677825966104863439140201369`27.7255557365983},
   {-0.41131306713898188305087386187907075075`26.822193737162966,
    0.86315136113355651932209726878280402738`27.680351667890253},
   {-0.66949971878499157782762681272154851253`25.87145755353705,
    1.04336675256358795162454375904746651582`26.24154649457202},
   {-0.12387346586949235788169482574561257965`22.862396033460378,
    1.69053841019507635342824262818397345497`23.92755089295085},
   {-0.14781568220083659098114755501070878814`20.89905486127738,
    1.29876685761076711458137499603183990807`20.650672926520834},
   {-0.79794182115868580085391082 76`18.198970265781032,
    1.73488945046557917484505672 56`18.696147777769255},
   {-0.58660822714765623790765356 63`17.186047877000277,
    1.10260584599142726415899439 42`17.474737144758677},
   {-0.27902661534367222988540814 51`15.221639104612777,
    0.18236024523169924713661687 97`14.576251214372071},
   {-0.69947680477737880311789841 62`13.539836232404737,
    0.14421137349147403254518060 37`13.61049218887457},
   {-0.15806807538136367176485395 69`11.054808921583602,
    0.70652837504054562688937838 53`11.705272862674462},
   {-0.27476420202771379811285635 67`9.626703639138345,
    0.1887213405929091626`8.871445995552044},
   {-0.6762579184348799177`7.690148658284796,
    -0.0793862439909723592`6.917623467769003},
   {-0.2751116434597207771`6.1001956987470605,
    -0.8117855086769284249`6.44055446025527},
   {-0.785422480423465103`4.757515311600611,
    -0.7449164860285702181`5.338176139692664},
   {-0.9170732123424068868`4.028680617531947,
    -0.3228533086347782651`3.310451401049224},
   {-0.8169471476532205183`3.080375344443701,
    -0.8155668962180758346`2.9295005552854834}};

path43 = {{0.5`43., 0.1`43.},
   {0.68202026619435145028245947132559115324845885824983720911 3`41.97959
     147734878, 0.1`43.},
   {0.13535359952768478361579280465892448658179219158324352674 2`40.59308
     819948877,
    0.48157568056677825966104863440929061610184937010019534906 4`41.72555
```

```
  573659831},
{-0.41131306713898188305087386200774218008487447508335015563`40.82219
   3737162955,
 0.8631513611335565193220972688185812322036987402003906981 29`41.68035
  1667890264},
{-0.669499718784991577827626812709919452171917799292233169144`39.8714
   5755353705,
 1.0433667525635879516245437589750101379725702893102553424 43`40.24154
  649457203},
{-0.123873465869492357881694825306121854482473387528`36.8623960334603
 ,
 1.690538410195076353428242628005783837769430685068`37.92755089295085
 },
{-0.147815682200836590981147553202989949197484143534`34.8990548612773
 ,
 1.298766857610767114581374996928560604119684614547`34.65067292652083
 ,
{-0.797941821158685800853910815999514436627142312429`32.1989702657810
   35,
 1.734889450465579174845056734468711141824661872405`32.69614777776926
 ,
{-0.586608227147656237907653489451668390415071410423`31.1860478770002
 ,
 1.102605845991427264158994424807713237365072532152`31.474737144758684
 },
{-0.279026615343672229910973206412386627503133126394`29.2216391046127
 ,
 0.182360245231699247213104912362126159901061734686`28.576251214372075
 }, {-0.699476804777378803145442193238831946 45`26.33653723818892,
 0.144211373491474032748765467180943736 41`26.668386097543287},
{-0.158068075381363673008691766231688092 87`23.602322253496816,
 0.706528375040545627487491881715160748 49`24.24622863451234},
{-0.274764202027713839918741654090346631 21`21.38748966446956,
 0.188721340592909006844547556012182957 68`20.58578036298622},
{-0.676257918434880016389627754 2`19.08105055799066,
 -0.079386243990973088324051595281040814 28`18.317791381376484},
{-0.275111643459723862795531196 2`16.37038286272229,
 -0.811785508676932935218949177 4`16.587479435211847},
{-0.785422480137813736731620928 4`14.510480395232836,
 -0.744916486066025484608142979 8`15.250339536395382},
{-0.917073211418066331476388711 8`13.441226943343025,
 -0.322853308557614733344976498 3`12.56991198952313},
{-0.838752730113357710911055142 5`11.948397813233655,
 -0.708263082443761576436073912 6`11.262054886165851},
{-0.736292698609618308080955099 3`11.039048997422958,
 -0.669642696363571377162059716 7`10.917922704870435}};
```

■ **Trajectory Plot**

It may be hard to see the difference between a software precision of 43 digits versus software precision of 29 digits (43 digits gives us 10 digits of accuracy). Here we will use $t = 10$:

```
mirrors = Table[Disk[{i, j}, 1 / 3], {i, -1, 2}, {j, -1, 2}];
mirroroutline = Table[Circle[{i, j}, 1 / 3], {i, -1, 2}, {j, -1, 2}];
startPoint = Circle[{1/2, 1/10}, 0.015];
p43 = Graphics[{{GrayLevel[0.75], mirrors}, mirroroutline, startPoint,
    Map[Line, Partition[path43, 2, 1]], Disk[Last[path43], 0.015]},
   AspectRatio → Automatic, Axes → True, AxesOrigin → {-1, -1},
   PlotRange → {{-1, 2}, {-1, 2}},
   Ticks → {Range[-1, 2.5, .5], Range[-1, 2.5, .5]}];
p29 = Graphics[{{GrayLevel[0.75], mirrors}, mirroroutline, startPoint,
    Map[Line, Partition[path29, 2, 1]], Disk[Last[path29], 0.015]},
   AspectRatio → Automatic, Axes → True, AxesOrigin → {-1, -1},
   PlotRange → {{-1, 2}, {-1, 2}},
   Ticks → {Range[-1, 2.5, .5], Range[-1, 2.5, .5]}];
Show[GraphicsArray[{p43, p29}]];
```

It might be easier if we zoom in and view what happens near the end of the photons path:

```
mirrors = Table[Disk[{i, j}, 1 / 3], {i, -1, 0}, {j, -1, 0}];
mirroroutline = Table[Circle[{i, j}, 1 / 3], {i, -1, 0}, {j, -1, 0}];
startPoint = Circle[{1/2, 1/10}, 0.015];
p43 = Graphics[{{GrayLevel[0.75], mirrors}, mirroroutline, startPoint,
    Map[Line, Partition[path43, 2, 1]], Disk[Last[path43], 0.015]},
   AspectRatio → Automatic, Axes → True, AxesOrigin → {-1, -1},
   PlotRange → {{-1, 0}, {-1, 0}},
   Ticks → {Range[-1, 2.5, .5], Range[-1, 2.5, .5]}];
p29 = Graphics[{{GrayLevel[0.75], mirrors}, mirroroutline, startPoint,
    Map[Line, Partition[path29, 2, 1]], Disk[Last[path29], 0.015]},
   AspectRatio → Automatic, Axes → True, AxesOrigin → {-1, -1},
   PlotRange → {{-1, 0}, {-1, 0}},
   Ticks → {Range[-1, 0, .5], Range[-1, 0, .5]}];
Show[GraphicsArray[{p43, p29}]];
```

■ Results Using Fixed Precision With Algorithm 3.1

Here is the code from Algorithm 3.1 in the form of a function so I can just put in the fixed precision.

```
calcFixedPrec[fixedPrec_, time_] := Module[{H, p, v, timeRem, m, s, path},
  H[{a_, b_}] := 9 ( b^2 - a^2   -2 a b
                     -2 a b   a^2 - b^2 );
  p = N[{1/2, 1/10}, fixedPrec];
  v = {1, 0};
  timeRem = time;
  path = {p};
  While[timeRem > 0,
       m = Round[p + 2/3 v];
       s =
    Min[Cases[t /. Solve[(p + t v - m).(p + t v - m) == 1/9, t], _?Positive]];
       If[s < timeRem,
            p += s v; v = H[p - m].v,
            s = Min[timeRem, 2/3]; p += s v
        ];
       timeRem -= s;
       path = Append[path, p];
  ];
  Norm[p]
  Print["The final position of our photon = ", Last[path]];
  Print["Distance from p to the origin = ", answer = Norm[p]];
  Print["Number of digits Mathematica claims to have correct = ",
   Precision[answer]]
 ];
```

Testing that it works:

```
calcFixedPrec[56, 10]
```

```
The final position of our photon =
 {-0.736292698609618310776, -0.669642696363571375194}

Distance from p to the origin = 0.995262919443354160890

Number of digits Mathematica claims to have correct = 21.5611
```

We will display a table of the results from 29 to 154 by steps of 5 to illustrate the number of digits *Mathematica* claims to have correct:

```
DisplayForm[StyleBox[GridBox[Prepend[Table[
     {prec, 7
        answer = calcFixedPrec[prec, 10] ,
      Precision[answer]}, {prec, 29, 154, 5}],
    {"Precision", "Computed Distance", "Number of Correct Digits"}],
   GridFrame → 1, ColumnAlignments → {Center, Left, Right},
   RowLines → {1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0,
     0, 0, 0, 1, 0, 0, 0, 0, 1}], FontFamily → Times, FontSize → 9]]
```

Reliable Reflections

■ Algorithm 3.2

Supporting functions for Algorithm 3.2

```
(* Mathematica does not have an interval version of Min *)
iMin[{Interval[{a_, b_}], Interval[{c_, d_}]}] :=
   Interval[{Min[a, c], Min[b, d]}];
 (* Just in case our interval is empty *)
 iMin[{}] := ∞;
 (* Convert an Interval to a Real *)
 IntervalToReal[Interval[{a_, b_}]] := (a + b)/2;
 (* In case IntervalToReal is passed a Real number we have this case *)
 IntervalToReal[a_] := a;
 (* Convert a Real number r to an Interval with diameter d *)
 RealToInterval[r_, d_] := r + Interval[{-1, 1}] d;
 (* The diameter or width of an Interval *)
 diam[Interval[{a_, b_}]] := b - a;
 (* Takes the diameter over a list of Intervals
   and returns the one with the largest diameter *)
 diam[{i__Interval}] := Max[diam /@ {i}];
 (* Once again we defined our transformation matrix *)
 H[{a_, b_}] := 9 {{b^2 - a^2, -2 a b}, {-2 a b, a^2 - b^2}};
```

This algorithm uses interval arithmetic. Each input will therefore be a small interval around the initial value. Begin Algorithm 3.2:

```
(* STEP 1 *)
(* The maximum amount of time the photon will be reflecting *)
tMax = 10;
(* The amount of time remaining begins as a degenerate interval *)
tRem = Interval[{tMax, tMax}];
(* Our starting point *)
p = {1/2, 1/10};
(* Our starting velocity *)
v = {1, 0};
(* Our accuracy goal *)
accuracyGoal = 100;
ϵ = 10^-accuracyGoal;
(* If we know ahead of time what interval
  radius allows us to get the required accuracy goal,
 then sIntervalPrecision eliminates the trial and error of
  increasing the working precision. We will assume that we do know
  the interval radius. In the algorithm this is denoted by s *)
sIntervalPrecision = 127;
(* Our error will eventually decrease,
 so we can just set it to infinity *)
error = ∞;
(* Our working precision, wp *)
workingPrecision = sIntervalPrecision + 2;

(* STEP 2 *)
(* While we have not reached our
  required accuracy goal do the following *)
While[error > ϵ,
  (* Change our point p into an interval
     (if necessary) with workingPrecision-digit accuracy *)
  P = N[RealToInterval[p, 10^-sIntervalPrecision], workingPrecision];
  (* Similarly for our velocity vector *)
  V = N[RealToInterval[v, 10^-sIntervalPrecision], workingPrecision];
  (* A list of the intersection points for our photon,
   these will be changed from intervals to real numbers later *)
  iPath = {P};
  (* This is similar to Algorithm 2.1 *)
  While[tRem > 0,
       (* Assume the mirror we hit has midpoint M (Round might cause
       difficulty in version of Mathematica less than 5.0) *)
       M = Round[P + 2/3 V];
       (* Find the time S,
    at which our photon intersects the mirror. This is a
     set of interval solutions to the following quadratic *)
       S = t /. (Solve[(P + t V - M).(P + t V - M) == 1/9, t]);
```

```
      (* If S DOES NOT contain an expression of the form
     √[a,b] with a < 0 < b then do the first statement. Note
     that FreeQ returns True if NO SUCH expr matches form *)
      If[FreeQ[S, Power[Interval[{_?Negative, _?Positive}], _]],
       (* Let T be those solutions in S of the form [a,b] with a ≥
       0. If T is empty then let T=iMin[T]=[∞,∞]
       (this is carried oiut by the functional definition of iMin) *)
        T = iMin[Cases[S, _?Positive]],
       (* Else there is an expression of the form √[a,b] with a <
     0 < b so exit the inner While loop *)
       Break[]
       ];
      (* Test values of T and timeRem and apply the appropriate case *)
      Which[
           (* The photon intersects this mirror
     so update its position and velocity and reduce tRem *)
           T ≤ tRem,
               P += T V;
               V = H[P - M].V;
               tRem -= T,
           (* The mirror in question does not intersect
     with the photon so try the next one and reduce tRem*)
           T > tRem && tRem ≥ 2 / 3,
               P += 2 V / 3;
                tRem -= 2 / 3,
           (* The time has run out and the photon stops moving
      so update photons final position and set tRem = 0 *)
           T > tRem && tRem < 2 / 3,
               P += tRem V;
               tRem = 0,
           (* Incomparable intervals so exit inner While loop *)
           True,
               Break[]
       ];
      (* Add P to the iPath list *)
      AppendTo[iPath, P];
      (* If any of tRem, P, V,
   or T are less than accuracyGoal then exit the inner While
    loop and thus increase our working precision. This shouldn't
    happen though if we choose sIntervalPrecision correctly! *)
       If[Precision[{tRem, P, V, T}] < ϵ,
           Break[]
       ];
];
(* We didn't reach our accuracyGoal so we need
  to change our working precision and set our error *)
++sIntervalPrecision;
workingPrecision = sIntervalPrecision + 2;
```

```
  error =
   diam[P + Table[RealToInterval[-Max[Abs[tRem]], Max[Abs[tRem]]], {2}]];
 ];

(* STEP 3 *)
(* Return iPath *)
(* This solution uses interval arithmetic,
 so each element of iPath represents a point x =
  (x1,x2) of the path with x1 and x2 both being intervals. We
   use IntervalForm to show the Interval in pretty print *)
Norm[Last[iPath]] // IntervalForm
```

$$0.995262919443354160890311809426721621029466922734154349803208858072986179622830632099174981895^{950}_{574}$$

Let's say we want to graph iPath to see whether it corresponds to the path found in Algorithm 2.1. First we must convert every interval in iPath to a real number. We use the rule x_Interval to check to retrieve, the Intervals from iPath, then we use IntervalToReal to convert it. We use 16 digit precision.

```
path = N[iPath /. x_Interval:> IntervalToReal[x], 16];
```

The standard plotting routine.

```
mirrors = Table[Disk[{i, j}, 1 / 3], {i, -1, 2}, {j, -1, 2}];
mirroroutline = Table[Circle[{i, j}, 1 / 3], {i, -1, 2}, {j, -1, 2}];
startPoint = Circle[{1/2, 1/10}, 0.015];
Show[Graphics[{{GrayLevel[0.75], mirrors}, mirroroutline, startPoint,
   Map[Line, Partition[path, 2, 1]], Disk[Last[path], 0.015]},
  AspectRatio → Automatic, Axes → True, AxesOrigin → {-1, -1},
  PlotRange → {{-1, 2}, {-1, 2}},
  Ticks → {Range[-1, 2.5, .5], Range[-1, 2.5, .5]}]]
```

■ Time Complexity Tests for Both Algorithms

The noninterval version (Algorithm 3.1):

```
H[{a_, b_}] := 9 ( b^2 - a^2   -2 a b
                   -2 a b   a^2 - b^2 );

nonInterval[reqPrecision_] := Module[
   {p, v, tRem, s, m},
   p = N[{1 / 2, 1 / 10}, reqPrecision + 40];
   v = {1, 0};
   tRem = 10;
   While[tRem > 0,
    m = Round[p + 2 v / 3];
    s =
     Min[Cases[t /. Solve[(p + t v - m).(p + t v - m) == 1 / 9, t], _?Positive]];
    If[s < tRem,
         p += s v;
         v = H[p - m].v,
     (* Else *)
         s = Min[tRem, 2 / 3];
         p += s v
    ];
    tRem -= s
   ];
   Norm[p]
  ];

timedata = Table[{i,
    Developer`ClearCache[];
    Timing[{i, nonInterval[i]}][[1, 1]]}, {i, 500, 30000, 500}];
```

- **Cached Data**

```
timedataNonInterval =
  {{500, 0.125`}, {1000, 0.36000000000000004`}, {1500, 0.703`},
   {2000, 1.0779999999999998`}, {2500, 1.6090000000000004`},
   {3000, 2.2189999999999994`}, {3500, 2.578000000000001`},
   {4000, 3.405999999999999`}, {4500, 4.110000000000001`},
   {5000, 5.172000000000001`}, {5500, 5.780999999999999`},
   {6000, 6.530999999999999`}, {6500, 7.765999999999998`},
   {7000, 8.484000000000002`}, {7500, 10.735`}, {8000, 12.515`},
   {8500, 13.406000000000006`}, {9000, 14.281999999999996`},
   {9500, 16.265`}, {10000, 18.250000000000014`},
   {10500, 20.51599999999999`}, {11000, 20.811999999999983`},
   {11500, 23.844000000000023`}, {12000, 24.562999999999988`},
   {12500, 26.171999999999997`}, {13000, 29.187000000000012`},
   {13500, 30.312999999999988`}, {14000, 32.139999999999986`},
   {14500, 34.422000000000025`}, {15000, 34.422000000000025`},
   {15500, 36.061999999999955`}, {16000, 38.922000000000025`},
   {16500, 38.45300000000003`}, {17000, 41.95399999999995`},
   {17500, 43.389999999999986`}, {18000, 43.96900000000005`},
   {18500, 48.34399999999994`}, {19000, 48.593000000000075`},
   {19500, 53.375`}, {20000, 56.047000000000025`},
   {20500, 55.702999999999975`}, {21000, 61.235000000000014`},
   {21500, 62.46899999999994`}, {22000, 64.15599999999995`},
   {22500, 68.81200000000013`}, {23000, 69.17200000000003`},
   {23500, 68.54700000000003`}, {24000, 74.9219999999998`},
   {24500, 74.25`}, {25000, 75.04700000000003`},
   {25500, 76.82799999999997`}, {26000, 80.73400000000015`},
   {26500, 86.51600000000008`}, {27000, 89.5`},
   {27500, 95.95299999999997`}, {28000, 96.18799999999987`},
   {28500, 99.5619999999999`}, {29000, 99.09400000000005`},
   {29500, 101.89000000000033`}, {30000, 106.125`}};
```

```
timedataInterval =
  {{500, 0.43799999999999994`}, {1000, 0.5620000000000003`},
   {1500, 0.7809999999999997`}, {2000, 1.0630000000000002`},
   {2500, 1.4059999999999997`}, {3000, 1.766`}, {3500, 2.125000000000001`},
   {4000, 2.484`}, {4500, 2.937999999999999`}, {5000, 3.3590000000000018`},
   {5500, 3.75`}, {6000, 4.422000000000001`}, {6500, 4.936999999999998`},
   {7000, 5.5`}, {7500, 5.938000000000002`}, {8000, 6.452999999999996`},
   {8500, 7.109000000000002`}, {9000, 7.703000000000003`},
   {9500, 8.312999999999995`}, {10000, 8.984000000000009`}, {10500, 9.61`},
   {11000, 10.09299999999999`}, {11500, 11.03200000000001`},
   {12000, 11.780999999999992`}, {12500, 12.546999999999997`},
   {13000, 13.515000000000015`}, {13500, 14.062999999999988`},
   {14000, 15.`}, {14500, 15.51600000000002`},
   {15000, 16.030999999999977`}, {15500, 15.469000000000023`},
   {16000, 16.155999999999977`}, {16500, 16.656000000000034`},
   {17000, 17.78099999999995`}, {17500, 18.5`},
   {18000, 19.047000000000025`}, {18500, 20.360000000000014`},
   {19000, 20.920999999999992`}, {19500, 21.610000000000014`},
   {20000, 22.562000000000012`}, {20500, 23.468999999999994`},
   {21000, 24.23399999999998`}, {21500, 25.125`}, {22000, 25.5`},
   {22500, 26.906999999999982`}, {23000, 27.703000000000088`},
   {23500, 28.515999999999963`}, {24000, 29.390999999999963`},
   {24500, 29.812000000000012`}, {25000, 30.812000000000012`},
   {25500, 32.125`}, {26000, 32.82800000000009`},
   {26500, 33.81299999999999`}, {27000, 34.28099999999995`},
   {27500, 35.84400000000005`}, {28000, 37.202999999999975`},
   {28500, 37.73400000000004`}, {29000, 38.5`},
   {29500, 39.73400000000004`}, {30000, 40.28200000000004`}};
```

■ The Plots

We may generate a plot of the times needed by the non-interval algorithm to get d digits of the answer using the assumption that it takes $d + 40$ digits of working precision to get d digits of the answer.

```
ListPlot[timedataNonInterval,
  Ticks → {Range[10000, 30000, 10000], Range[5, 110, 10]},
  AxesLabel -> {None, DisplayForm[
     StyleBox["CPU time [sec]", FontFamily → "Times", FontSize → 9]]},
  PlotJoined → True, PlotStyle → Thickness[0.0008], Epilog →
   {PointSize[.01], Point /@ timedata, Text["Digits", {30000, .2},
     {1, -1}, TextStyle → {FontFamily → "Times", FontSize → 9}]},
  TextStyle → {FontFamily → "Times", FontSize → 7}];
```

```
ListPlot[timedataInterval,
  Ticks → {Range[10000, 30000, 10000], Range[5, 45, 5]},
  AxesLabel -> {None, DisplayForm[
     StyleBox["CPU time [sec]", FontFamily → "Times", FontSize → 9]]},
  PlotJoined → True, PlotStyle → Thickness[0.0008],
  Epilog → {PointSize[.01], Point /@ timedataInterval,
    Text["Digits", {30000, .2}, {1, -1},
     TextStyle → {FontFamily → "Times", FontSize → 9}]},
  TextStyle → {FontFamily → "Times", FontSize → 7}];
```

If we really wanted to, we can see what our photon's path looks like after 2000 time steps

```
ipath2000 = ReliableTrajectory[{1 / 2, 1 / 10}, {1, 0},
   2000, AccuracyGoal → 12, StartIntervalPrecision → 5459,
   IntervalPrecisionStep → 1, ShowSize → False];
path = N[ ipath2000 /. x_Interval :→ IntervalToReal[x], 16];

Show[Graphics[{Line[path], Point[Last[path]]}], Frame → True,
  FrameTicks → {Range[0, 60, 20], Range[0, 40, 20], None, None},
  AspectRatio → Automatic];
```

APPENDIX B

Chapter 4 Code

This appendix lists the *Mathematica* code for Chapter 4.

Basic Initializations

Takes two arguments **x** and **y** and its compiled version :

```
f[x_, y_] := e^Sin[50 x] + Sin[60 e^y] +
   Sin[70 Sin[x]] + Sin[Sin[80 y]] - Sin[10 (x + y)] + (x^2 + y^2)/4;
fc = Compile[{x, y}, e^Sin[50 x] + Sin[60 e^y] + Sin[70 Sin[x]] +
    Sin[Sin[80 y]] - Sin[10 (x + y)] + (x^2 + y^2)/4];
```

Takes one argument of the form **{x,y}** and its compiled version**:**

```
f[{x_, y_}] := f[x, y];
fcl = Compile[{{x, _Real, 1}}, e^Sin[50 x[[1]]] + Sin[60 e^x[[2]]] + Sin[70 Sin[x[[1]]]] +
    Sin[Sin[80 x[[2]]]] - Sin[10 (x[[1]] + x[[2]])] + (x[[1]]^2 + x[[2]]^2)/4];
```

Both the compiled and the non-compiled versions yield the same result. The compiled version uses optimized byte-code.

```
f[1., 1.]
fc[1., 1.]
f[{1., 1.}]
fcl[{1., 1.}]
```

```
-0.0362174

-0.0362174

-0.0362174

-0.0362174
```

Yet another version of the above:

```
f[x_] := e^Sin[50 x[[1]]] + Sin[60 e^x[[2]]] + Sin[70 Sin[x[[1]]]] +
   Sin[Sin[80 x[[2]]]] - Sin[10 (x[[1]] + x[[2]])] + (x[[1]]^2 + x[[2]]^2)/4;
```

A First Look

A wide angle view of our function *g*:

```
Plot3D[fc[x, y], {x, -30, 30}, {y, -30, 30}]
```

A view of the unit box centered at the origin using **PlotPoints→37:**

```
Plot3D[fc[x, y], {x, -1, 1}, {y, -1, 1}, {PlotPoints → 37, Mesh → False}]
```

A view of the unit box centered at the origin using **PlotPoints→300:**

```
Plot3D[fc[x, y], {x, -1, 1}, {y, -1, 1}, {PlotPoints → 300, Mesh → False}]
```

A **ContourPlot** of our function *g*. White areas are larger values, dark areas are lower values.

```
ContourPlot[fc[x, y], {x, -1, 1},
  {y, -1, 1}, {PlotPoints → 500, ContourLines → False}];
```

We can get an approximate upper bound on the global minimum by considering a fine grid of points in $[-1, 1]\times[-1, 1]$

```
Min[Table[fc[x, y], {x, -1, 1, 0.01}, {y, -1, 1, 0.01}]]
```

```
-3.24646
```

We can find the minimum value and its point in one swoop

```
grid = Flatten[Table[{x, y}, {x, -1, 1, 0.01}, {y, -1, 1, 0.01}], 1];
fgrid = fcl /@ grid;
{Min[fgrid], Flatten[Extract[grid, Position[fgrid, Min[fgrid]]], 1]}
```

```
{-3.24646, {-0.02, 0.21}}
```

Using a finer grid will give us a better approximation

```
Min[Table[fc[x, y], {x, -1, 1, 0.001}, {y, -1, 1, 0.001}]]
```

```
-3.30563
```

We can use this fine grid to find a better rough estimate

```
grid = Flatten[Table[{x, y}, {x, -1, 1, 0.001}, {y, -1, 1, 0.001}], 1];
fgrid = fcl /@ grid;
{Min[fgrid], Flatten[Extract[grid, Position[fgrid, Min[fgrid]]], 1]}
```

```
{-3.30563, {-0.024, 0.211}}
```

Survival of the Fittest

I have left some of the old code in just to be complete. The "Old Solution" was before I tried using functional programming. At first it was a bit more intuitive to program the genetic algorithm this way, but I found a better way later.

■ Algorithm 4.1 (Without Pictures)

■ Old Solution

Writing a routine using operational programming is tedious.

```
(* g (x), the objective function. *)
g2[x_] := e^Sin[50 x[[1]]] + Sin[60 e^x[[2]]] + Sin[70 Sin[x[[1]]]] +
     Sin[Sin[80 x[[2]]]] - Sin[10 (x[[1]] + x[[2]])] + (x[[1]]^2 + x[[2]]^2)/4;
  (* R, the search rectangle is [-1,1] x [-1,1]. *)
  h1 = 1;
  h2 = 1;
 (* n, the number of children for each parent,
  and the number of points in the new generation. *)
 n = 40;
 (* epsilon, a bound on the absolute
   error in the location of the minimum of g in R. *)
 epsilon = 10^-6;
 (* a counter for the number of generations *)
 gen = 1;

(* Step 1 *)
(* z is the center of R *)
z = {0., 0.};
 parents = {z};
 children = {};
 fvals = {g2[z]};
 (* s, a scaling factor for shrinking the search domain. *)
 s = 1 / 2;
```

```
(* Step 2 *)
While[Min[h1, h2] > epsilon,
     (* For each p ∈ parents,
  let its children consist of n random points in a
    rectangle around p = (p1,p2). Use a uniform random
     x and y chosen from [-h1,h1]+p1 and [-h2,h2]+p2,
  respectively. We store the children for parent i
   in the following way: for parent[[i]],
  its children are located at children[[1 through 50]]. The
   first generation starts with n children,
  the remaining generations each have n*n children total. *)
     For[i = 1, i ≤ Length[parents], i++,
          children = Join[children,
     SetPrecision[Table[{Random[Real, {-h1, h1} + parents[[i]][[1]] ],
        Random[Real, {-h2, h2} + parents[[i]][[2]]]}, {50}], 15]];
      ];
     (* Let newfvals be the f-
   values on the set of all children. We map g2 to each
    element in g2 and then separate the values by braces. *)
     newfvals = Map[g2, children];
     (* Form fvals⋃newfvals,
  and use the n lowest values to determine the points
   from the children and the previous parents that will
   survive. Let parents be this set of n points. *)
     totalcandidates = Join[parents, children];
     totalfvals = Join[fvals, newfvals];
     sortorder = Take[Ordering[totalfvals], n];
     parents = {} ;
     For[i = 1, i ≤ n,
  parents = Append[parents, totalcandidates[[sortorder[[i]]]]]; i++];
     (* Let fvals be the corresponding f-
   values for the parents *)
     fvals = Map[g2, parents];
     (* Shrink the search rectangle by s. *)
     h1 = h1 * s;
     h2 = h2 * s;
     Print[gen, "  ", fvals[[1]], "  ", parents[[1]]];
     gen++;
];
```

```
1  -1.7097164721036  {0.0577690227159755, 0.778113311347590}

2  -3.0196864777856  {-0.387603345785599, -0.0971305675385633}

3  -3.0196864777856  {-0.387603345785599, -0.0971305675385633}

4  -3.0398985053265  {-0.397958059156894, -0.0995745144209791}

5  -3.23899704116767  {-0.0218025425103515, 0.207546887648758}

6  -3.30440566341025  {-0.0252607838684860, 0.210855761091968}
```

```
7  -3.30456933266838  {-0.0235566475804601, 0.210430510069728}

8  -3.30645781845459  {-0.0241481149711172, 0.210819300335705}

9  -3.30677916400846  {-0.0243913203183506, 0.210478259894842}

10  -3.30685943676274  {-0.0243796962936357, 0.210651340168840}

11  -3.30686618746736  {-0.0244020472700103, 0.210590137227259}

12  -3.30686784132445  {-0.0244178268089609, 0.210618185161435}

13  -3.30686851521736  {-0.0243975617425479, 0.210609487282718}

14  -3.30686863912703  {-0.0244045035910734, 0.210611761123014}

15  -3.30686863912703  {-0.0244045035910734, 0.210611761123014}

16  -3.30686864535369  {-0.0244026517274562, 0.210612987026216}

17  -3.30686864743351  {-0.0244031613023877, 0.210612361543736}

18  -3.30686864747280  {-0.0244030610473374, 0.210612410160251}

19  -3.30686864747280  {-0.0244030610473374, 0.210612410160251}

20  -3.30686864747280  {-0.0244030610473374, 0.210612410160251}
```

```
(* Step 3 *)
(* Return the smallest value
  in fvals and the corresponding parent *)
Print["Minimum value = ", fvals[[1]]];
Print["Approximate location = ", parents[[1]]];
```

```
Minimum value = -3.30686864747280

Approximate location = {-0.0244030610473374, 0.210612410160251}
```

■ **New Solution**

Using functional programming gives us a solution similar to that in the SIAM book.

```
(* f(x), the objective function as defined above *)
(* R, the search rectangle is [-1,1] x [-1,1]. I
  make it clear that the bounds of R are h1 and h2
  and are both equal (since it is a square). For
  simplicity we let this value be h *)
h = h1 = h2 = 1;
 (* n, the number of children for each parent,
  and the number of points in the new generation *)
 n = 50;
 (* ∈, a bound on the absolute error
   in the location of the minimum of g in R *)
 ∈ = 10^-6;
 (* counter, a counter for the number of generations *)
 counter = 1;
 (* z, the center of R *)
 z = {0., 0.};
 (* parents, a list of the parents of our
   generation. It begins initially as the center of R *)
 parents = {z};
 (* children, a list of the children of
   each point in List[parents]. It is initially empty *)
 children = {};
 (* fvals, the function values |of the points in List[children] *)
 fvals = {fcl[z]};
 (* s, a scaling factor for shrinking the search domain *)
 s = 1 / 2;
```

```
While[h > ϵ,
  (* For each p ∈ parents, let its children consists of n random
    points in a rectangle around p. (2 Random[]-1) yields a
    point in [-1,1]×[-1,1] and h is the scaling factor. Flatten
    is required since the out Table makes an extra List *)
  children = Flatten[Table[# + Table[h (2 Random[] - 1), {2}], {n}] & /@
     parents, 1];
  (* Let newfvals be the f-values on the set of all
     children. We can just Map g over all of the children *)
  newfvals = fcl /@ children;
  (* For simplicity we join together the f-
    values and the points in a List of the form {g[point],point}. Notice
     in this step we also perform Join[fvals,newfvals] since
     union decides to sort the results as well. We then Sort these
     results and using Take we can take the first n elements of
     this list (the n smallest values) and use those as parents *)
  gen = Take[Sort[Transpose[{Join[fvals, newfvals],
       Join[parents, children]}]], n];
  (* I needed some way of formatting the List gen
    to remove the fvals from it *)
  parents = Drop[Flatten[gen, 1], {1, 2 n, 2}];
  (* Let fvals be the set of corresponding f-
    values of the new parents *)
  fvals = f /@ parents;
  (* Reduce the size of our search rectangle R by a factor of s *)
  h = h * s;
  (* Print out what our current
    minimum is and its approximate location *)
  Print[counter, "  ", fvals[[1]], "  ", parents[[1]]];
  (* Update the gen counter *)
  counter++;
 ];
Print["Minimum value = ", fvals[[1]]];
Print["Approximate location = ", parents[[1]];
```

```
1  -1.43256  {-0.392426, 0.523681}

2  -3.1944  {-0.393275, -0.091631}

3  -3.1944  {-0.393275, -0.091631}

4  -3.29786  {-0.0243328, 0.209275}

5  -3.3052  {-0.0245878, 0.210053}

6  -3.3052  {-0.0245878, 0.210053}

7  -3.30579  {-0.0247195, 0.21022}

8  -3.30685  {-0.02433, 0.210596}

9  -3.30685  {-0.02433, 0.210596}
```

```
10  -3.30686  {-0.0243417, 0.210615}

11  -3.30687  {-0.0244191, 0.210598}

12  -3.30687  {-0.0244192, 0.21061}

13  -3.30687  {-0.0243982, 0.210616}

14  -3.30687  {-0.0244039, 0.210612}

15  -3.30687  {-0.0244039, 0.210612}

16  -3.30687  {-0.0244031, 0.210612}

17  -3.30687  {-0.0244031, 0.210612}

18  -3.30687  {-0.0244031, 0.210612}

19  -3.30687  {-0.0244031, 0.210612}

20  -3.30687  {-0.0244031, 0.210612}

Minimum value = -3.30687

Approximate location = {-0.0244031, 0.210612}
```

■ Algorithm 4.1 (With Pictures)

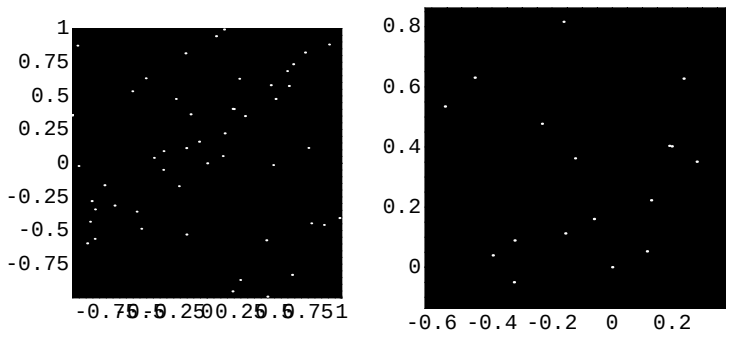
Some of the pictures were removed due to size constraints.

■ New Solution

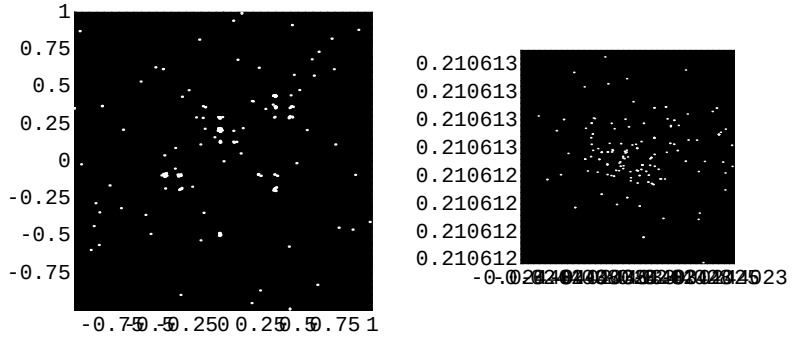
Here is the above code, but with all the comments stripped. This is the non color version.

```
(* A dummy variable for storing all the graphics primitives *)
currentPlotData = {Black, Rectangle[{-1, -1}, {1, 1}], White};
 h = 1;
 n = 50;
 ϵ = 10^-6;
 counter = 1;
 z = {0., 0.};
 parents = {z};
 children = {};
 fvals = {fcl[z]};
 s = 1 / 2;
 While[h > ϵ,
   children =
    Flatten[Table[# + Table[h (2 Random[] - 1), {2}], {n}] & /@ parents, 1];
   newfvals = fcl /@ children;
   gen = Take[Sort[
      Transpose[{Join[fvals, newfvals], Join[parents, children]}]], n];
   parents = Drop[Flatten[gen, 1], {1, 2 n, 2}];
   fvals = fcl /@ parents;
   h = h * s;
   Print[counter, "  ", fvals[[1]], "  ", parents[[1]]];
   counter++;
   AppendTo[currentPlotData, Point /@ parents];
   (* Generates plots that show within original box R *)
   p1 = Graphics[currentPlotData,
     AspectRatio -> Automatic, Axes -> Automatic, Frame → True,
     PlotRange → {{-1, 1}, {-1, 1}}, DisplayFunction → Identity];
   p2 = Graphics[currentPlotData, AspectRatio -> Automatic,
     Axes -> Automatic, Frame → True,
     PlotRange → {{-h, h} + parents[[1, 1]], {-h, h} + parents[[1, 2]]},
     DisplayFunction → Identity];
   Show[GraphicsArray[{p1, p2}]]
  ];
 Print["Minimum value = ", fvals[[1]]];
 Print["Approximate location = ", parents[[1]]];
```

```
1  -1.86657  {-0.123739, 0.362735}
```



```
2  -2.94212  {-0.390521, -0.0858331}

3  -3.16876  {-0.395126, -0.0900202}

4  -3.16876  {-0.395126, -0.0900202}

5  -3.22208  {-0.0248888, 0.214858}

6  -3.30571  {-0.0239211, 0.210913}

7  -3.30589  {-0.0243598, 0.21017}

8  -3.30678  {-0.0245741, 0.210609}

9  -3.30685  {-0.0244841, 0.210595}

10  -3.30687  {-0.0244349, 0.210604}

11  -3.30687  {-0.0244349, 0.210604}

12  -3.30687  {-0.0244016, 0.210609}

13  -3.30687  {-0.0244016, 0.210609}

14  -3.30687  {-0.0244058, 0.210614}

15  -3.30687  {-0.0244023, 0.210613}

16  -3.30687  {-0.0244028, 0.210613}

17  -3.30687  {-0.024403, 0.210612}

18  -3.30687  {-0.024403, 0.210612}

19  -3.30687  {-0.024403, 0.210612}

20  -3.30687  {-0.0244031, 0.210612}
```




```
Minimum value = -3.30687

Approximate location = {-0.0244031, 0.210612}
```

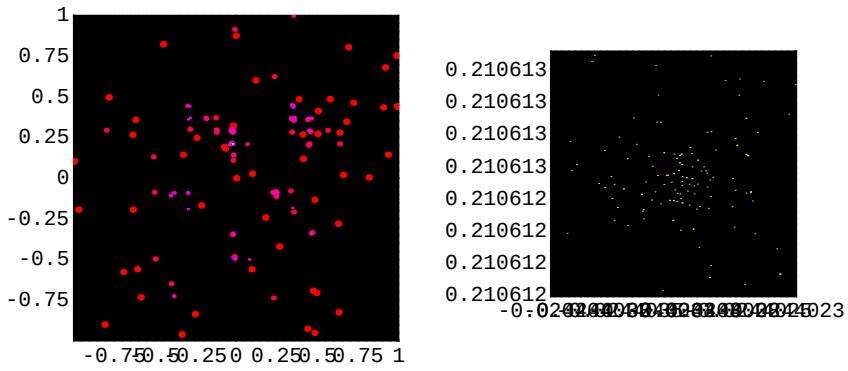
This is the color version.

```
(* A dummy variable for storing all the graphics primitives *)
currentPlotData = {{Black, Rectangle[{-1, -1}, {1, 1}]}};
 (* A dummy variable for storing the white
   rectangle for the approximate answer at each gen *)
 currentRecData = {};
 (* We are going to put white boxes
   behind the approximate answer at each gen *)
 curPointSize = 0.02;
 (* The amount we will reduce
   curPointSize by at each gen from 1 through 5 *)
 redPointSize = 0.004;
 (* The amount we will reduce curPointSize by at each gen from 5 on *)
 redPointSize5 = 0.0001;
 (* The current Hue for the Points in gen *)
 curHue = 1;
 (* The factor to reduce curHue by *)
 redHue = 0.045;
 (* The function for plotting the rectangle *)
 centeredRectangle[{x_, y_}] :=
   {White, Rectangle[{x, y} - {0.02, 0.02}, {x, y} + {0.02, 0.02}]};
 centeredRectangle[{x_, y_}, size_] :=
   {White, Rectangle[{x, y} - {size, size}, {x, y} + {size, size}]};
 h = 1;
 n = 50;
 ϵ = 10^-6;
 counter = 1;
 z = {0., 0.};
 parents = {z};
 children = {};
 fvals = {f[z]};
 s = 1 / 2;
 While[h > ϵ,
   children =
    Flatten[Table[# + Table[h (2 Random[] - 1), {2}], {n}] & /@ parents, 1];
   newfvals = f /@ children;
   gen = Take[Sort[
      Transpose[{Join[fvals, newfvals], Join[parents, children]}]], n];
   parents = Drop[Flatten[gen, 1], {1, 2 n, 2}];
   fvals = f /@ parents;
   h = h * s;
   Print[counter, "  ", fvals[[1]], "  ", parents[[1]]];
   counter++;
   AppendTo[currentRecData, centeredRectangle[parents[[1]]]];
   AppendTo[currentPlotData,
    {Hue[curHue], PointSize[curPointSize], Point /@ parents}];
   (* Reduce the size of the Points and change
     the Color for the next iteration *)
   If[counter < 6, curPointSize -= redPointSize,
```

```
    curPointSize -= redPointSize5];
   curHue -= redHue;
   (* Generates plots that show within original box R *)
   p1 = Graphics[currentPlotData,
     AspectRatio → Automatic, Axes → Automatic, Frame → True,
     PlotRange → {{-1, 1}, {-1, 1}}, DisplayFunction → Identity];
   p2 = Graphics[currentPlotData, AspectRatio → Automatic,
     Axes → Automatic, Frame → True,
     PlotRange → {{-h, h} + parents[[1, 1]], {-h, h} + parents[[1, 2]]},
     DisplayFunction → Identity];
   Show[GraphicsArray[{p1, p2}]];
  ];
 Print["Minimum value = ", fvals[[1]];
 Print["Approximate location = ", parents[[1]];
 Show[Graphics[currentPlotData, AspectRatio → Automatic,
    PlotRange → {{-h, h} + parents[[1, 1]], {-h, h} + parents[[1, 2]]},
    Frame → True, FrameTicks → {{-0.024404, -0.0244035,
       -0.024403, -0.0244025}, Automatic, None, None}]];
 Show[Graphics[Insert[currentPlotData, currentRecData, 2],
    AspectRatio → Automatic, Frame → True, PlotRange → {{-1, 1}, {-1, 1}},
    FrameTicks → {{-1, -0.5, 0, 0.5, 1}, Automatic, None, None}]];
```

```
1  -1.25282  {-0.808323, -0.899458}

2  -3.20409  {-0.0262528, 0.21509}

3  -3.20409  {-0.0262528, 0.21509}

4  -3.20409  {-0.0262528, 0.21509}

5  -3.24355  {-0.0267874, 0.207622}

6  -3.30503  {-0.0237475, 0.210273}

7  -3.30675  {-0.0243902, 0.210768}

8  -3.30683  {-0.024343, 0.210682}

9  -3.30685  {-0.0243836, 0.210667}

10  -3.30687  {-0.0243756, 0.210608}

11  -3.30687  {-0.0243856, 0.210606}

12  -3.30687  {-0.0244066, 0.210615}

13  -3.30687  {-0.0244066, 0.210615}

14  -3.30687  {-0.0244023, 0.210612}

15  -3.30687  {-0.0244023, 0.210612}

16  -3.30687  {-0.0244025, 0.210612}

17  -3.30687  {-0.0244025, 0.210612}
```

```
18  -3.30687  {-0.0244031, 0.210612}

19  -3.30687  {-0.0244031, 0.210612}

20  -3.30687  {-0.0244031, 0.210612}
```




```
Minimum value = -3.30687

Approximate location = {-0.0244031, 0.210612}
```

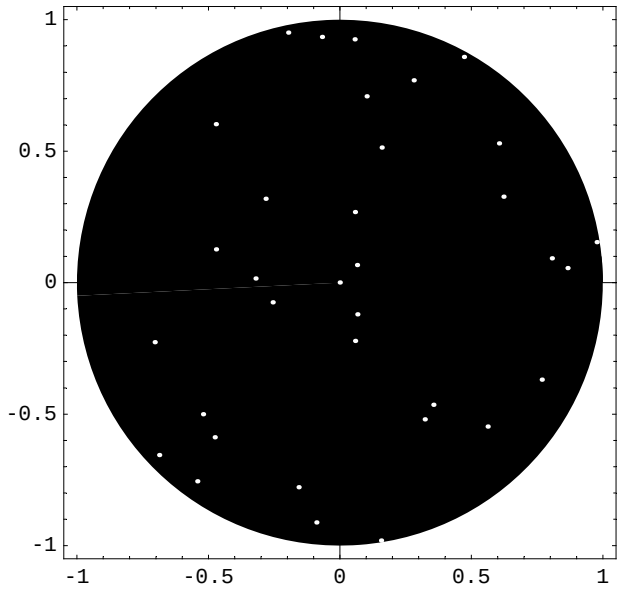
■ Old Solution

```
(* g(x), the objective function. *)
g2[x_] := e^Sin[50 x[[1]]] + Sin[60 e^x[[2]]] + Sin[70 Sin[x[[1]]]] +
     Sin[Sin[80 x[[2]]]] - Sin[10 (x[[1]] + x[[2]])] + (x[[1]]^2 + x[[2]]^2)/4;
  (* R, the search rectangle is [-1,1] x [-1,1]. *)
  h1 = 1;
  h2 = 1;
 (* n, the number of children for each parent,
  and the number of points in the new generation. *)
 n = 40;
 (* epsilon, a bound on the absolute
   error in the location of the minimum of g in R. *)
 epsilon = 10^-6;
 (* a counter for the number of generations *)
 gen = 1;
```

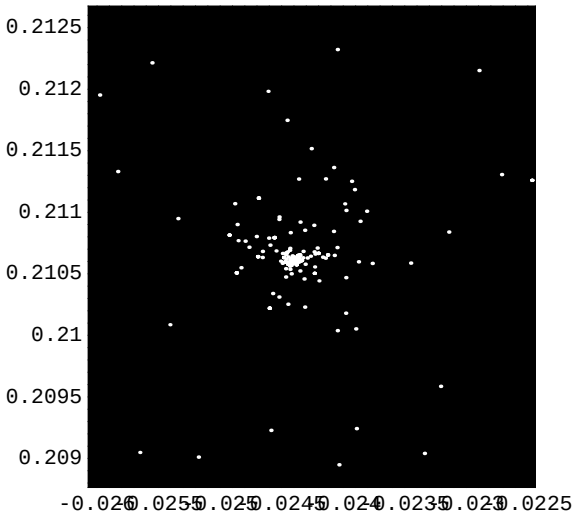
```
(* Step 1 *)
(* z is the center of R *)
z = {0., 0.};
 parents = {z};
 children = {};
 fvals = {g2[z]};
 (* s, a scaling factor for shrinking the search domain. *)
 s = 1 / 2;
 (* A dummy variable for storing all the graphics primitives *)
 currentPlotData = {Disk[{0, 0}, 1], GrayLevel[1]};
```

```
(* Step 2 *)
While[Min[h1, h2] > epsilon,
     (* For each p ∈ parents,
  let its children consist of n random points in a
    rectangle around p = (p1,p2). Use a uniform random
    x and y chosen from [-h1,h1]+p1 and [-h2,h2]+p2,
  respectively. We store the children for parent i
   in the following way: for parent[[i]],
  its children are located at children[[1 through 50]]. The
   first generation starts with n children,
  the remaining generations each have n*n children total. *)
     For[i = 1, i ≤ Length[parents], i++,
          children = Join[children,
     SetPrecision[Table[{Random[Real, {-h1, h1} + parents[[i]][[1]] ],
        Random[Real, {-h2, h2} + parents[[i]][[2]]]}, {50}], 15]];
      ];
     (* Let newfvals be the f-
   values on the set of all children. We map g2 to each
    element in g2 and then separate the values by braces. *)
     newfvals = Map[g2, children];
     (* Form fvals⋃newfvals,
  and use the n lowest values to determine the points
   from the children and the previous parents that will
   survive. Let parents be this set of n points. *)
     totalcandidates = Join[parents, children];
     totalfvals = Join[fvals, newfvals];
     sortorder = Take[Ordering[totalfvals], n];
     parents = {} ;
     For[i = 1, i ≤ n,
  parents = Append[parents, totalcandidates[[sortorder[[i]]]]]; i++];
     (* Let fvals be the corresponding f-
   values for the parents *)
     fvals = Map[g2, parents];
     (* Shrink the search rectangle by s. *)
     h1 = h1 * s;
     h2 = h2 * s;
     gen++;
     (* Plotting our generations
   involves adding to our currentPlotData. *)
     currentPlotData = Join[currentPlotData, Map[Point, parents]];
     Show[Graphics[currentPlotData,
   AspectRatio -> Automatic, Axes -> Automatic, Frame → True]];
];
```

First Iteration



Last Iteration




```
(* Step 3 *)
(* Return the smallest value
  in fvals and the corresponding parent *)
Print["Minimum value = ", fvals[[1]]];
Print["Approximate location = ", parents[[1]]];
```

■ The SIAM Book Solution

This is a concise way to use the genetic algorithm done in functional programming:

```
h = 1; gen = {f[#], #} & /@ {{0, 0}};
While[h > 10^-6,
  new =
   Flatten[Table[#[[2]] + Table[h (2 Random[] - 1), {2}], {50}] & /@ gen, 1];
  gen = Take[Sort[Join[gen, {f[#], #} & /@ new]], 50];
  h = h / 2];
gen[[1]]
```

```
{-3.30687, {-0.0244031, 0.210612}}
```

The *Mathematica* one-liner

```
NMinimize[{f[x, y], x^2 + y^2 ≤ 1}, {x, y},
 Method → {"DifferentialEvolution", "SearchPoints" → 250}]
```

```
{-3.30687, {x → -0.0244031, y → 0.210612}}
```

Interval Arithmetic

Why is $R = [-1, 1] \times [-1, 1]$ the area where our minimum lies? We may use interval arithmetic:

```
f[Interval[{-∞, -1.}], Interval[{-∞, ∞}]]
f[Interval[{1., ∞}], Interval[{-∞, ∞}]]
f[Interval[{-∞, ∞}], Interval[{-∞, -1.}]]
f[Interval[{-∞, ∞}], Interval[{1., ∞}]]
```

```
Interval[{-3.22359, ∞}]

Interval[{-3.22359, ∞}]

Interval[{-3.22359, ∞}]

Interval[{-3.22359, ∞}]
```

■ Search & Destroy

■ Supporting Functions

This is a way to visually plot intervals

```
showIntervals[a_List, opts___] := (Show[Graphics[
      ({PointSize[.025], {{Hue[Random[], .5, .7],
            tt = Rectangle @@ (tt = Transpose[{#1[[1, 1]], #1[[2, 1]]}])},
           Point[.5 Plus @@ tt]}} &) /@ a], opts]) /; Depth[a] == 5;
```

The midpoint of an interval

```
mid[X_] := (Min[X] + Max[X]) / 2;
midl[{X_, Y_}] := {(Min[X] + Max[X]) / 2, (Min[Y] + Max[Y]) / 2}
```

Good implementation of subdivide

```
subdivide1D[X_] := Interval /@ {{Min[X], mid[X]}, {mid[X], Max[X]}};
subdivide2D[{X_, Y_}] := Distribute[{subdivide1D[X], subdivide1D[Y]}, List]
```

■ **Algorithm 4.2 (Using only the upper bound test)**

```
(* STEP 1 *)
(* rects,
 the set of rectangles that might contain the global minimum *)
rects = N[{{Interval[{-1, 1}], Interval[{-1, 1}]}}];
(* i, a counter for the number of iterations *)
i = 0;
(* imax, the maximum number of allowed iterations *)
imax = 20;
(* lwrbnd, lower bound of our estimate to the min value *)
lwrbnd = -∞;
(* uprbnd, upper bound of our estimate to the min value *)
uprbnd = -3.24;
(* ϵ, a bound on the absolute error for
   the final approximation to the lowest f-value *)
ϵ = 10^-12;
(* fvals. the f-values of each rectangle in rect *)
fvals = {};
(* pos,
 a list of positions of rectangles that changes based on our needs *)
pos = {};

(* STEP 2 *)
While[(uprbnd - lwrbnd) > ϵ && i < imax,
  i++;
  Print["Iteration ", i, "\n", "------------"];
  (* Uniformly divide each
    rectangle in rects into 4 smaller rectangles *)
  rects = Join @@ subdivide2D /@ rects;
  Print["The number of rectangles remaining
     before the size and gradient test = ", Length[rects]];
  (* Find the fvals of each of the rectangles *)
  fvals = f /@ rects;
  (* Find the new upper bound *)
  uprbnd = Min[uprbnd, Min[Max /@ fvals]];
  (* Check the
    size: Delete from rects any rectangle T for which the left end
      of f[T] is not less than uprbnd. We can do this by finding
      the positions of the rectangles that the left is of f[T] is
      less than uprbnd, then just reseting rects *)
  pos = Flatten[Position[Min /@ fvals, _?(# ≤ uprbnd &)]];
  (* Now just let rects and fvals be the corresponding
     rectangles and f-values that passed the first test *)
  rects = rects[[pos]];
  fvals = fvals[[pos]];
  Print["The number of rectangles remaining after the size test = ",
   Length[rects]];
```

```
  lwrbnd = Min[fvals];
  (* Normal view *)
  showIntervals[rects, AspectRatio → Automatic,
   Frame → True, PlotRange → {{-1, 1}, {-1, 1}}];
 ];
```

```
Iteration 1
------------

The number of rectangles remaining before the size and gradient test = 4

The number of rectangles remaining after the size test = 4

Iteration 2
------------

The number of rectangles remaining before the size and gradient test = 16

The number of rectangles remaining after the size test = 16

Iteration 3
------------

The number of rectangles remaining before the size and gradient test = 64

The number of rectangles remaining after the size test = 60

Iteration 4
------------

The number of rectangles remaining before the size and gradient test = 240

The number of rectangles remaining after the size test = 110

Iteration 5
------------

The number of rectangles remaining before the size and gradient test = 440

The number of rectangles remaining after the size test = 58

Iteration 6
------------

The number of rectangles remaining before the size and gradient test = 232

The number of rectangles remaining after the size test = 35

Iteration 7
------------

The number of rectangles remaining before the size and gradient test = 140

The number of rectangles remaining after the size test = 14

Iteration 8
------------

The number of rectangles remaining before the size and gradient test = 56

The number of rectangles remaining after the size test = 13
```

```
Iteration 9
------------

The number of rectangles remaining before the size and gradient test = 52

The number of rectangles remaining after the size test = 16

Iteration 10
------------

The number of rectangles remaining before the size and gradient test = 64

The number of rectangles remaining after the size test = 37

Iteration 11
------------

The number of rectangles remaining before the size and gradient test = 148

The number of rectangles remaining after the size test = 55

Iteration 12
------------

The number of rectangles remaining before the size and gradient test = 220

The number of rectangles remaining after the size test = 100

Iteration 13
------------

The number of rectangles remaining before the size and gradient test = 400

The number of rectangles remaining after the size test = 201

Iteration 14
------------

The number of rectangles remaining before the size and gradient test = 804

The number of rectangles remaining after the size test = 400

Iteration 15
------------

The number of rectangles remaining before the size and gradient test = 1600

The number of rectangles remaining after the size test = 805

Iteration 16
------------

The number of rectangles remaining before the size and gradient test = 3220

The number of rectangles remaining after the size test = 1608

Iteration 17
------------

The number of rectangles remaining before the size and gradient test = 6432

The number of rectangles remaining after the size test = 3220
```

```
Iteration 18
------------

The number of rectangles remaining before the size and gradient test = 12880

The number of rectangles remaining after the size test = 6427

Iteration 19
------------

The number of rectangles remaining before the size and gradient test = 25708

The number of rectangles remaining after the size test = 12869

Iteration 20
------------

The number of rectangles remaining before the size and gradient test = 51476

The number of rectangles remaining after the size test = 25757
```

■ **Algorithm 4.2 (Using both the upper bound and gradient test)**

```
(* STEP 1 *)
(* rects,
 the set of rectangles that might contain the global minimum *)
rects = N[{{Interval[{-1.0, 1.0}], Interval[{-1.0, 1.0}]}}];
(* i, a counter for the number of iterations *)
i = 0;
(* imax, the maximum number of allowed iterations *)
imax = 20;
(* lwrbnd, lower bound of our estimate to the min value *)
lwrbnd = -∞;
(* uprbnd, upper bound of our estimate to the min value *)
uprbnd = -3.24;
(* reqDigits, the number of digits required *)
reqDigits = 12;
(* ϵ, a bound on the absolute error for
   the final approximation to the lowest f-value *)
ϵ = 10^-reqDigits;
(* fvals. the f-values of each rectangle in rect *)
fvals = {};
(* pos,
 a list of positions of rectangles that changes based on our needs *)
pos = {};
(* gradient,
 a function that returns both g_x and g_y where g is our function *)
gradient[{x_, y_}] := Evaluate[{D[f[x, y], x], D[f[x, y], y]}];
(* a list to store an array of graphics for presentation purposes *)
ISgraphics = {};
(* examined, the total number of rectangles examined in one run *)
examined = 0;

(* STEP 2 *)
While[i < imax && (uprbnd - lwrbnd) > ϵ,
  i++;
  Print["Iteration ", i, "\n", "------------"];
  (* Uniformly divide each
    rectangle in rects into 4 smaller rectangles *)
  rects = Join @@ subdivide2D /@ rects;
  Print["The number of rectangles remaining
     before the size and gradient test = ", Length[rects]];
  examined += Length[rects];
  (* Find the fvals of each of the rectangles *)
  fvals = f /@ rects;
  (* Find the new upper bound *)
  uprbnd = Min[uprbnd, Min[Max /@ fvals]];
  (* Check the
    size: Delete from rects any rectangle T for which the left end
```

```
      of f[T] is not less than uprbnd. We can do this by finding
      the positions of the rectangles that the left is of f[T] is
      less than uprbnd, then just reseting rects *)
  pos = Flatten[Position[Min /@ fvals, _?(# ≤ uprbnd &)]];
  (* Now just let rects and fvals be the corresponding
     rectangles and f-values that passed the first test *)
  rects = rects[[pos]];
  fvals = fvals[[pos]];
  Print["The number of rectangles remaining after the size test = ",
   Length[rects]];
  (* Check the gradient: delete from rects any interior rectangle T for
      which f_x[T] does not contain 0 or f_y[T] does not contain 0 *)
  pos = Flatten[Position[Apply[And, IntervalMemberQ[
       gradient /@ rects, 0], {1}], True]];
  rects = rects[[pos]];
  lwrbnd = Min[fvals[[pos]]];
  Print["The number of rectangles remaining
     after the size and gradient test = ", Length[rects]];
  Print["Approximate location of the minimum = ", midl /@ rects];
  Print["The current upper bound = ", NumberForm[uprbnd, 20]];
  Print["The current lower bound = ", NumberForm[lwrbnd, 20]];
  (* Normal view *)
  showIntervals[rects, AspectRatio → Automatic,
   Frame → True, PlotRange → {{-1, 1}, {-1, 1}},
   PlotLabel → StringForm["Upper Bound = ``", NumberForm[uprbnd, 5]]];
  (* IS view *)
  (* AppendTo[ISgraphics,
     showIntervals[rects,AspectRatio→Automatic, Frame→True,
      FrameTicks→None,PlotRange→{{-1,1},{-1,1}},PlotLabel→
       StringForm["Upper Bound = ``", NumberForm[uprbnd,5]]]]; *)
 ];
Print["Results", "\n", "-------"];
Print["Total number of rectangles examined = ", examined];
Print["Interval enclosure of the global minimum = ", Flatten[rects]];
Print["Interval enclosure of the f-value = ", f[Flatten[rects]]];
Print["Approximate location of the global minimum = ",
  NumberForm[midl[Flatten[rects]], reqDigits]];
Print["Approximate f-value = ",
  NumberForm[f[midl[Flatten[rects]]], reqDigits]];
```

```
Iteration 1
------------

The number of rectangles remaining before the size and gradient test = 4

The number of rectangles remaining after the size test = 4

The number of rectangles remaining after the size and gradient test = 4

Approximate location of the minimum =
 {{-0.5, -0.5}, {-0.5, 0.5}, {0.5, -0.5}, {0.5, 0.5}}
```

```
The current upper bound = -3.24

The current lower bound = -3 + 1/e - Sin[1]

Iteration 2
------------

The number of rectangles remaining before the size and gradient test = 16

The number of rectangles remaining after the size test = 16

The number of rectangles remaining after the size and gradient test = 16

Approximate location of the minimum =
 {{-0.75, -0.75}, {-0.75, -0.25}, {-0.25, -0.75}, {-0.25, -0.25},
  {-0.75, 0.25}, {-0.75, 0.75}, {-0.25, 0.25}, {-0.25, 0.75},
  {0.25, -0.75}, {0.25, -0.25}, {0.75, -0.75}, {0.75, -0.25},
  {0.25, 0.25}, {0.25, 0.75}, {0.75, 0.25}, {0.75, 0.75}}

The current upper bound = -3.24

The current lower bound = -3 + 1/e - Sin[1]

Iteration 3
------------

The number of rectangles remaining before the size and gradient test = 64

The number of rectangles remaining after the size test = 60

The number of rectangles remaining after the size and gradient test = 60

Approximate location of the minimum =
 {{-0.875, -0.625}, {-0.625, -0.875}, {-0.625, -0.625}, {-0.875, -0.375},
  {-0.875, -0.125}, {-0.625, -0.375}, {-0.625, -0.125}, {-0.375, -0.875},
  {-0.375, -0.625}, {-0.125, -0.875}, {-0.125, -0.625}, {-0.375, -0.375},
  {-0.375, -0.125}, {-0.125, -0.375}, {-0.125, -0.125}, {-0.875, 0.125},
  {-0.875, 0.375}, {-0.625, 0.125}, {-0.625, 0.375}, {-0.875, 0.625},
  {-0.625, 0.625}, {-0.625, 0.875}, {-0.375, 0.125}, {-0.375, 0.375},
  {-0.125, 0.125}, {-0.125, 0.375}, {-0.375, 0.625}, {-0.375, 0.875},
  {-0.125, 0.625}, {-0.125, 0.875}, {0.125, -0.875}, {0.125, -0.625},
  {0.375, -0.875}, {0.375, -0.625}, {0.125, -0.375}, {0.125, -0.125},
  {0.375, -0.375}, {0.375, -0.125}, {0.625, -0.875}, {0.625, -0.625},
  {0.875, -0.625}, {0.625, -0.375}, {0.625, -0.125}, {0.875, -0.375},
  {0.875, -0.125}, {0.125, 0.125}, {0.125, 0.375}, {0.375, 0.125},
  {0.375, 0.375}, {0.125, 0.625}, {0.125, 0.875}, {0.375, 0.625},
  {0.375, 0.875}, {0.625, 0.125}, {0.625, 0.375}, {0.875, 0.125},
  {0.875, 0.375}, {0.625, 0.625}, {0.625, 0.875}, {0.875, 0.625}}

The current upper bound = -3.24

The current lower bound = -3 + 1/e - Sin[1]

Iteration 4
------------

The number of rectangles remaining before the size and gradient test = 240

The number of rectangles remaining after the size test = 110
```

```
The number of rectangles remaining after the size and gradient test = 110

Approximate location of the minimum =
 {{-0.6875, -0.5625}, {-0.5625, -0.6875}, {-0.5625, -0.5625},
  {-0.8125, -0.3125}, {-0.9375, -0.1875}, {-0.9375, -0.0625},
  {-0.8125, -0.1875}, {-0.6875, -0.4375}, {-0.6875, -0.3125}, {-0.5625, -0.4375},
  {-0.5625, -0.0625}, {-0.4375, -0.6875}, {-0.4375, -0.5625}, {-0.3125, -0.6875},
  {-0.1875, -0.9375}, {-0.0625, -0.9375}, {-0.0625, -0.5625}, {-0.3125, -0.3125},
  {-0.4375, -0.1875}, {-0.4375, -0.0625}, {-0.3125, -0.1875}, {-0.3125, -0.0625},
  {-0.1875, -0.4375}, {-0.1875, -0.3125}, {-0.0625, -0.4375}, {-0.0625, -0.3125},
  {-0.1875, -0.1875}, {-0.8125, 0.1875}, {-0.9375, 0.4375}, {-0.8125, 0.3125},
  {-0.8125, 0.4375}, {-0.6875, 0.0625}, {-0.6875, 0.1875}, {-0.5625, 0.0625},
  {-0.5625, 0.1875}, {-0.6875, 0.3125}, {-0.5625, 0.5625}, {-0.5625, 0.6875},
  {-0.5625, 0.8125}, {-0.4375, 0.0625}, {-0.4375, 0.4375}, {-0.3125, 0.3125},
  {-0.3125, 0.4375}, {-0.1875, 0.1875}, {-0.0625, 0.0625}, {-0.0625, 0.1875},
  {-0.1875, 0.3125}, {-0.1875, 0.4375}, {-0.0625, 0.3125}, {-0.4375, 0.5625},
  {-0.4375, 0.6875}, {-0.3125, 0.5625}, {-0.0625, 0.6875}, {-0.1875, 0.8125},
  {-0.1875, 0.9375}, {-0.0625, 0.8125}, {-0.0625, 0.9375}, {0.0625, -0.6875},
  {0.0625, -0.5625}, {0.1875, -0.6875}, {0.1875, -0.5625}, {0.4375, -0.9375},
  {0.3125, -0.6875}, {0.0625, -0.4375}, {0.0625, -0.0625}, {0.1875, -0.1875},
  {0.1875, -0.0625}, {0.3125, -0.3125}, {0.4375, -0.4375}, {0.4375, -0.3125},
  {0.3125, -0.1875}, {0.3125, -0.0625}, {0.4375, -0.1875}, {0.5625, -0.5625},
  {0.6875, -0.5625}, {0.8125, -0.5625}, {0.5625, -0.4375}, {0.5625, -0.3125},
  {0.6875, -0.4375}, {0.6875, -0.0625}, {0.8125, -0.1875}, {0.8125, -0.0625},
  {0.9375, -0.1875}, {0.9375, -0.0625}, {0.0625, 0.0625}, {0.0625, 0.1875},
  {0.1875, 0.0625}, {0.1875, 0.4375}, {0.4375, 0.1875}, {0.3125, 0.3125},
  {0.3125, 0.4375}, {0.4375, 0.3125}, {0.4375, 0.4375}, {0.0625, 0.5625},
  {0.0625, 0.6875}, {0.1875, 0.5625}, {0.1875, 0.6875}, {0.0625, 0.8125},
  {0.3125, 0.5625}, {0.4375, 0.9375}, {0.5625, 0.0625}, {0.5625, 0.1875},
  {0.6875, 0.0625}, {0.6875, 0.1875}, {0.5625, 0.3125}, {0.8125, 0.0625},
  {0.9375, 0.4375}, {0.6875, 0.6875}, {0.5625, 0.8125}, {0.8125, 0.5625}}

The current upper bound = -3.24

The current lower bound = -3 + 1/e - Sin[1]

Iteration 5
-----------

The number of rectangles remaining before the size and gradient test = 440

The number of rectangles remaining after the size test = 58

The number of rectangles remaining after the size and gradient test = 58
```

```
Approximate location of the minimum =
 {{-0.40625, -0.71875}, {-0.40625, -0.65625}, {-0.28125, -0.71875},
  {-0.03125, -0.53125}, {-0.40625, -0.09375}, {-0.28125, -0.21875},
  {-0.28125, -0.09375}, {-0.03125, -0.46875}, {-0.03125, -0.34375},
  {-0.53125, 0.03125}, {-0.40625, 0.03125}, {-0.40625, 0.46875},
  {-0.28125, 0.34375}, {-0.28125, 0.40625}, {-0.28125, 0.46875},
  {-0.03125, 0.09375}, {-0.03125, 0.15625}, {-0.03125, 0.21875},
  {-0.03125, 0.28125}, {-0.40625, 0.53125}, {-0.40625, 0.59375},
  {-0.40625, 0.65625}, {-0.28125, 0.53125}, {-0.03125, 0.71875},
  {-0.03125, 0.78125}, {-0.03125, 0.84375}, {-0.03125, 0.90625},
  {0.09375, -0.65625}, {0.09375, -0.53125}, {0.21875, -0.71875},
  {0.21875, -0.65625}, {0.34375, -0.71875}, {0.09375, -0.46875},
  {0.21875, -0.09375}, {0.46875, -0.34375}, {0.34375, -0.21875},
  {0.34375, -0.09375}, {0.46875, -0.21875}, {0.09375, 0.03125},
  {0.09375, 0.09375}, {0.09375, 0.15625}, {0.21875, 0.03125},
  {0.21875, 0.46875}, {0.46875, 0.21875}, {0.34375, 0.34375}, {0.34375, 0.40625},
  {0.34375, 0.46875}, {0.46875, 0.28125}, {0.46875, 0.34375}, {0.46875, 0.40625},
  {0.09375, 0.59375}, {0.09375, 0.65625}, {0.09375, 0.71875}, {0.21875, 0.53125},
  {0.21875, 0.59375}, {0.21875, 0.65625}, {0.09375, 0.78125}, {0.34375, 0.53125}}

The current upper bound = -3.24

The current lower bound = -3.472614981136455

Iteration 6
------------

The number of rectangles remaining before the size and gradient test = 232

The number of rectangles remaining after the size test = 35

The number of rectangles remaining after the size and gradient test = 34

Approximate location of the minimum =
 {{-0.390625, -0.109375}, {-0.390625, -0.078125},
  {-0.296875, -0.109375}, {-0.015625, -0.484375}, {-0.296875, 0.453125},
  {-0.015625, 0.140625}, {-0.015625, 0.203125}, {-0.390625, 0.515625},
  {-0.390625, 0.546875}, {-0.390625, 0.609375}, {-0.296875, 0.515625},
  {-0.015625, 0.765625}, {-0.015625, 0.828125}, {-0.015625, 0.859375},
  {0.078125, -0.515625}, {0.078125, -0.484375},
  {0.234375, -0.109375}, {0.234375, -0.078125}, {0.453125, -0.328125},
  {0.359375, -0.234375}, {0.328125, -0.109375}, {0.078125, 0.046875},
  {0.078125, 0.140625}, {0.359375, 0.359375}, {0.328125, 0.453125},
  {0.359375, 0.453125}, {0.453125, 0.296875}, {0.453125, 0.359375},
  {0.078125, 0.671875}, {0.234375, 0.515625}, {0.234375, 0.546875},
  {0.234375, 0.609375}, {0.078125, 0.765625}, {0.328125, 0.515625}}

The current upper bound = -3.24

The current lower bound = -3.46478982059715

Iteration 7
------------

The number of rectangles remaining before the size and gradient test = 136

The number of rectangles remaining after the size test = 13

The number of rectangles remaining after the size and gradient test = 12
```

```
Approximate location of the minimum =
 {{-0.398438, -0.101563}, {-0.398438, -0.0859375}, {-0.0234375, -0.492188},
  {-0.289063, 0.445313}, {-0.0234375, 0.132813}, {-0.0234375, 0.210938},
  {-0.0234375, 0.773438}, {0.242188, -0.101563}, {0.242188, -0.0859375},
  {0.351563, 0.367188}, {0.335938, 0.445313}, {0.351563, 0.445313}}

The current upper bound = -3.24

The current lower bound = -3.452277713296274

Iteration 8
------------

The number of rectangles remaining before the size and gradient test = 48

The number of rectangles remaining after the size test = 13

The number of rectangles remaining after the size and gradient test = 6

Approximate location of the minimum =
 {{-0.394531, -0.0898438}, {-0.0195313, -0.496094}, {-0.292969, 0.441406},
  {-0.0273438, 0.207031}, {0.339844, 0.441406}, {0.347656, 0.441406}}

The current upper bound = -3.24

The current lower bound = -3.403657301021024

Iteration 9
------------

The number of rectangles remaining before the size and gradient test = 24

The number of rectangles remaining after the size test = 5

The number of rectangles remaining after the size and gradient test = 3

Approximate location of the minimum =
 {{-0.396484, -0.0917969}, {-0.392578, -0.0917969}, {-0.0253906, 0.208984}}

The current upper bound = -3.24

The current lower bound = -3.381069262602736

Iteration 10
------------

The number of rectangles remaining before the size and gradient test = 12

The number of rectangles remaining after the size test = 9

The number of rectangles remaining after the size and gradient test = 3

Approximate location of the minimum =
 {{-0.395508, -0.0927734}, {-0.393555, -0.0927734}, {-0.0244141, 0.209961}}

The current upper bound = -3.24

The current lower bound = -3.355131895424805

Iteration 11
------------
```

```
The number of rectangles remaining before the size and gradient test = 12

The number of rectangles remaining after the size test = 4

The number of rectangles remaining after the size and gradient test = 1

Approximate location of the minimum = {{-0.0239258, 0.210449}}

The current upper bound = -3.275696679400347

The current lower bound = -3.332572161150516

Iteration 12
------------

The number of rectangles remaining before the size and gradient test = 4

The number of rectangles remaining after the size test = 4

The number of rectangles remaining after the size and gradient test = 1

Approximate location of the minimum = {{-0.0241699, 0.210693}}

The current upper bound = -3.291695461173302

The current lower bound = -3.320691141905372

Iteration 13
------------

The number of rectangles remaining before the size and gradient test = 4

The number of rectangles remaining after the size test = 4

The number of rectangles remaining after the size and gradient test = 1

Approximate location of the minimum = {{-0.024292, 0.210571}}

The current upper bound = -3.29941737519003

The current lower bound = -3.313990411780293

Iteration 14
------------

The number of rectangles remaining before the size and gradient test = 4

The number of rectangles remaining after the size test = 4

The number of rectangles remaining after the size and gradient test = 1

Approximate location of the minimum = {{-0.024353, 0.210632}}

The current upper bound = -3.303168385814956

The current lower bound = -3.310489973558686

Iteration 15
------------

The number of rectangles remaining before the size and gradient test = 4
```

```
The number of rectangles remaining after the size test = 4

The number of rectangles remaining after the size and gradient test = 1

Approximate location of the minimum = {{-0.0243835, 0.210602}}

The current upper bound = -3.305026725933474

The current lower bound = -3.308692240628655

Iteration 16
------------

The number of rectangles remaining before the size and gradient test = 4

The number of rectangles remaining after the size test = 4

The number of rectangles remaining after the size and gradient test = 1

Approximate location of the minimum = {{-0.0243988, 0.210617}}

The current upper bound = -3.305949138174336

The current lower bound = -3.307784090037926

Iteration 17
------------

The number of rectangles remaining before the size and gradient test = 4

The number of rectangles remaining after the size test = 4

The number of rectangles remaining after the size and gradient test = 1

Approximate location of the minimum = {{-0.0244064, 0.210609}}

The current upper bound = -3.306409251616346

The current lower bound = -3.307326986612257

Iteration 18
------------

The number of rectangles remaining before the size and gradient test = 4

The number of rectangles remaining after the size test = 4

The number of rectangles remaining after the size and gradient test = 1

Approximate location of the minimum = {{-0.0244026, 0.210613}}

The current upper bound = -3.306639120660016

The current lower bound = -3.307097932421579

Iteration 19
------------

The number of rectangles remaining before the size and gradient test = 4

The number of rectangles remaining after the size test = 4
```

```
The number of rectangles remaining after the size and gradient test = 1

Approximate location of the minimum = {{-0.0244045, 0.210611}}

The current upper bound = -3.306753910117673

The current lower bound = -3.306983318206755

Iteration 20
------------

The number of rectangles remaining before the size and gradient test = 4

The number of rectangles remaining after the size test = 4

The number of rectangles remaining after the size and gradient test = 1

Approximate location of the minimum = {{-0.0244036, 0.210612}}

The current upper bound = -3.30681128557554

The current lower bound = -3.30692599313178

Results
-------

Total number of rectangles examined = 1264

Interval enclosure of the global minimum =
 {Interval[{-0.0244045, -0.0244026}], Interval[{0.210611, 0.210613}]}

Interval enclosure of the f-value = Interval[{-3.30693, -3.30681}]

Approximate location of the global minimum = {-0.0244035720825, 0.210612297058}

Approximate f-value = -3.30686864666
```

APPENDIX C

Chapter 5 Code

This appendix lists the *Mathematica* code for Chapter 5.

A First Look

We can use **SparseArray** to efficiently store our matrix *A*:

```
n = 20000;
b = Table[0, {n}];
b[[1]] = 1;
A =
  SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &@SparseArray[
    Flatten@Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2., n - i]}], n];
```

We may also test to see how long it takes to generate *A* and how much memory it takes to store (in MB):

```
n = 20000;
{Timing[A =
    SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &@SparseArray[
      Flatten@Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2., n - i]}], n];],
 ByteCount[A] / 1048576 //
  N}
```

```
{{1.656 Second, Null}, 4.30689}
```

Here is the sparsity pattern of *A*:

```
ArrayPlot[A, ColorRules → {0 -> White, _ -> Black}]
```

Quadratic Forms

Example plot of the quadratic form which has the form of a paraboloid:

```
A = {{3, 2}, {2, 6}};
b = {{2}, {-8}};
c = 0;
f[x_] := 1/2 Transpose[x].A.x - Transpose[b].x + c;
Plot3D[f[{{x}, {y}}], {x, -4, 6}, {y, -6, 4}];
```

The contour plot of the region above:

```
ContourPlot[f[{{x}, {y}}], {x, -4, 6}, {y, -6, 4},
 ContourShading → False, Contours → 20, PlotPoints → 45]
```

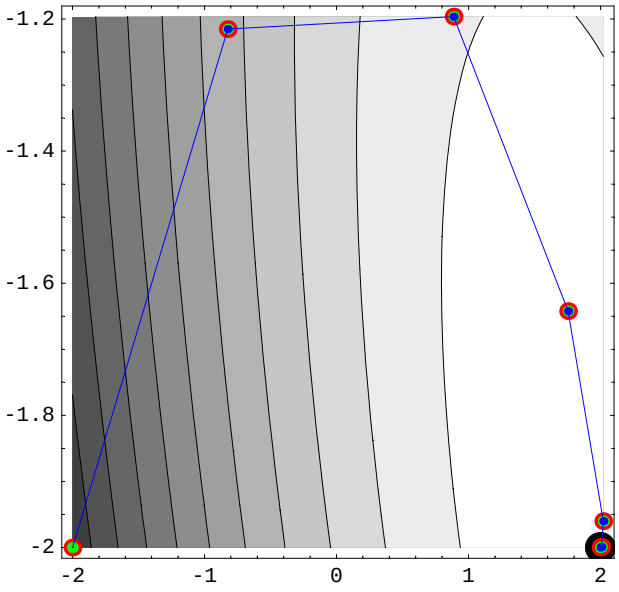
The gradient field for the region above:

```
<< Graphics`PlotField`
PlotGradientField[-2 x + 8 y + 1/2 (x (3 x + 2 y) + y (2 x + 6 y)), {x, -4, 6},
 {y, -6, 4}, Frame → True, HeadLength → 0.008, HeadWidth → 0.9]
```

Steepest Descent

In *Mathematica*, **FindMinimum** performs a type of steepest descent search automatically:

```
<< Optimization`UnconstrainedProblems`
FindMinimumPlot[-2 x + 8 y + 1/2 (x (3 x + 2 y) + y (2 x + 6 y)), {{x, -2}, {y, -2}}]
```




```
{{-10., {x → 2., y → -2.}},
 {Steps → 9, Function → 10, Gradient → 10}, - ContourGraphics -}
```

■ Nonoptimized Version

```
(* maxiters, the maximum number of iterations *)
maxiter = 22;
(* i, our counter *)
i = 0;
(* x0, our initial guess for the solution x in Ax = b. *)
x0 = {{-2.}, {-2.}};
(* xi, the approximate solution to x in Ax = b *)
xi = x0;

Print["Input: ", Subscript[x, 0], " = ", x0 // MatrixForm];
(* perform method of steepest descent
  until maxiter loops have been completed *)
While[i ≤ maxiter,
  (* calculate the residual *)
  ri = b - A.xi;
  (* find the appropriate alpha using line search *)
  alphai = Flatten[ Transpose[ri].ri / Transpose[ri].A.ri ][[1]];
  (* update the approximate answer *)
  xi = xi + alphai * ri;
  Print["Iteration ", i, "\n----------"];
  Print[Subscript[r, i], " = ", ri // MatrixForm];
  Print[Subscript[α, i], " = ", alphai];
  (* increment counter *)
  i++;
  Print[Subscript[x, i], " = ", xi // MatrixForm];
 ];
```

```
Input: x0 = ( -2. )
            ( -2. )

Iteration 0
----------

r0 = ( 12. )
     ( 8.  )

α0 = 0.173333

x1 = ( 0.08      )
     ( -0.613333 )

Iteration 1
----------

r1 = ( 2.98667 )
     ( -4.48   )

α1 = 0.309524
```

$$x_2 = \begin{pmatrix} 1.00444 \\ -2. \end{pmatrix}$$

Iteration 2

$$r_2 = \begin{pmatrix} 2.98667 \\ 1.99111 \end{pmatrix}$$

$$\alpha_2 = 0.173333$$

$$x_3 = \begin{pmatrix} 1.52213 \\ -1.65487 \end{pmatrix}$$

Iteration 3

$$r_3 = \begin{pmatrix} 0.743348 \\ -1.11502 \end{pmatrix}$$

$$\alpha_3 = 0.309524$$

$$x_4 = \begin{pmatrix} 1.75222 \\ -2. \end{pmatrix}$$

Iteration 4

$$r_4 = \begin{pmatrix} 0.743348 \\ 0.495565 \end{pmatrix}$$

$$\alpha_4 = 0.173333$$

$$x_5 = \begin{pmatrix} 1.88106 \\ -1.9141 \end{pmatrix}$$

Iteration 5

$$r_5 = \begin{pmatrix} 0.185011 \\ -0.277517 \end{pmatrix}$$

$$\alpha_5 = 0.309524$$

$$x_6 = \begin{pmatrix} 1.93833 \\ -2. \end{pmatrix}$$

Iteration 6

$$r_6 = \begin{pmatrix} 0.185011 \\ 0.123341 \end{pmatrix}$$

$$\alpha_6 = 0.173333$$

$$x_7 = \begin{pmatrix} 1.9704 \\ -1.97862 \end{pmatrix}$$

Iteration 7

$$r_7 = \begin{pmatrix} 0.0460472 \\ -0.0690708 \end{pmatrix}$$

$$\alpha_7 = 0.309524$$

$$x_8 = \begin{pmatrix} 1.98465 \\ -2. \end{pmatrix}$$

Iteration 8

$$r_8 = \begin{pmatrix} 0.0460472 \\ 0.0306981 \end{pmatrix}$$

$$\alpha_8 = 0.173333$$

$$x_9 = \begin{pmatrix} 1.99263 \\ -1.99468 \end{pmatrix}$$

Iteration 9

$$r_9 = \begin{pmatrix} 0.0114606 \\ -0.017191 \end{pmatrix}$$

$$\alpha_9 = 0.309524$$

$$x_{10} = \begin{pmatrix} 1.99618 \\ -2. \end{pmatrix}$$

Iteration 10

$$r_{10} = \begin{pmatrix} 0.0114606 \\ 0.00764043 \end{pmatrix}$$

$$\alpha_{10} = 0.173333$$

$$x_{11} = \begin{pmatrix} 1.99817 \\ -1.99868 \end{pmatrix}$$

Iteration 11

$$r_{11} = \begin{pmatrix} 0.00285243 \\ -0.00427864 \end{pmatrix}$$

$$\alpha_{11} = 0.309524$$

$$x_{12} = \begin{pmatrix} 1.99905 \\ -2. \end{pmatrix}$$

Iteration 12

$$r_{12} = \begin{pmatrix} 0.00285243 \\ 0.00190162 \end{pmatrix}$$

$$\alpha_{12} = 0.173333$$

$$x_{13} = \begin{pmatrix} 1.99954 \\ -1.99967 \end{pmatrix}$$

Iteration 13

$$r_{13} = \begin{pmatrix} 0.000709937 \\ -0.00106491 \end{pmatrix}$$

$$\alpha_{13} = 0.309524$$

$$x_{14} = \begin{pmatrix} 1.99976 \\ -2. \end{pmatrix}$$

Iteration 14

$$r_{14} = \begin{pmatrix} 0.000709937 \\ 0.000473291 \end{pmatrix}$$

$$\alpha_{14} = 0.173333$$

$$x_{15} = \begin{pmatrix} 1.99989 \\ -1.99992 \end{pmatrix}$$

Iteration 15

$$r_{15} = \begin{pmatrix} 0.000176695 \\ -0.000265043 \end{pmatrix}$$

$$\alpha_{15} = 0.309524$$

$$x_{16} = \begin{pmatrix} 1.99994 \\ -2. \end{pmatrix}$$

Iteration 16

$$r_{16} = \begin{pmatrix} 0.000176695 \\ 0.000117797 \end{pmatrix}$$

$$\alpha_{16} = 0.173333$$

$$x_{17} = \begin{pmatrix} 1.99997 \\ -1.99998 \end{pmatrix}$$

Iteration 17

$$r_{17} = \begin{pmatrix} 0.0000439775 \\ -0.0000659663 \end{pmatrix}$$

$$\alpha_{17} = 0.309524$$

$$x_{18} = \begin{pmatrix} 1.99999 \\ -2. \end{pmatrix}$$

Iteration 18

$$r_{18} = \begin{pmatrix} 0.0000439775 \\ 0.0000293184 \end{pmatrix}$$

$$\alpha_{18} = 0.173333$$

$$x_{19} = \begin{pmatrix} 1.99999 \\ -1.99999 \end{pmatrix}$$

Iteration 19

$$r_{19} = \begin{pmatrix} 0.0000109455 \\ -0.0000164183 \end{pmatrix}$$

$$\alpha_{19} = 0.309524$$

$$x_{20} = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$$

```
Iteration 20
------------
```

$$r_{20} = \begin{pmatrix} 0.0000109455 \\ 7.29701 \times 10^{-6} \end{pmatrix}$$

$$\alpha_{20} = 0.173333$$

$$x_{21} = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$$

```
Iteration 21
------------
```

$$r_{21} = \begin{pmatrix} 2.72422 \times 10^{-6} \\ -4.08633 \times 10^{-6} \end{pmatrix}$$

$$\alpha_{21} = 0.309524$$

$$x_{22} = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$$

```
Iteration 22
------------
```

$$r_{22} = \begin{pmatrix} 2.72422 \times 10^{-6} \\ 1.81615 \times 10^{-6} \end{pmatrix}$$

$$\alpha_{22} = 0.173333$$

$$x_{23} = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$$

■ Optimized Version

This version has one less matrix-vector product.

```
(* maxiters, the maximum number of iterations *)
maxiter = 22;
(* i, our counter *)
i = 0;
(* x0, our initial guess for the solution x in Ax = b. *)
x0 = {{-2.}, {-2.}};
(* xi, the approximate solution to x in Ax = b *)
xi = x0;
(* r0,
 we need to calculate r0 to start our iteration using the equation ri =
  b - Axi. After every 5 iterations, we update r0 to make
  sure floating point roundoff error does not accumulate. *)
r0 = b - A.x0;
ri = r0;

Print["Input: ", Subscript[x, 0], " = ", x0 // MatrixForm];
Print["       ", Subscript[r, 0], " = ", r0 // MatrixForm];
(* perform method of steepest descent
  until maxiter loops have been completed *)
While[i ≤ maxiter,
  (* calculate the matrix-vector product A.ri *)
  Ari = A.ri;
  (* find the appropriate alpha using line search *)
  alphai = Flatten[Transpose[ri].ri / Transpose[ri].Ari][[1]];
  (* update the approximate answer *)
  xi = xi + alphai * ri;
  (* If i is divisible by 5 *)
  If[Mod[i, 5] == 0,
   (* Find the residual the old way *)
   ri = b - A.xi,
   (* Keep using the optimized residual otherwise *)
   ri = ri - alphai * Ari
  ];
  Print["Iteration ", i, "\n------------"];
  Print[Subscript[α, i], " = ", alphai];
  (* increment counter *)
  i++;
  Print[Subscript[x, i], " = ", xi // MatrixForm];
  Print[Subscript[r, i], " = ", ri // MatrixForm];
 ];
```

Input: $x_0 = \begin{pmatrix} -2. \\ -2. \end{pmatrix}$

$r_0 = \begin{pmatrix} 12. \\ 8. \end{pmatrix}$

```
Iteration 0
------------
```

$\alpha_0 = 0.173333$

$x_1 = \begin{pmatrix} 0.08 \\ -0.613333 \end{pmatrix}$

$r_1 = \begin{pmatrix} 2.98667 \\ -4.48 \end{pmatrix}$

Iteration 1

$\alpha_1 = 0.309524$

$x_2 = \begin{pmatrix} 1.00444 \\ -2. \end{pmatrix}$

$r_2 = \begin{pmatrix} 2.98667 \\ 1.99111 \end{pmatrix}$

Iteration 2

$\alpha_2 = 0.173333$

$x_3 = \begin{pmatrix} 1.52213 \\ -1.65487 \end{pmatrix}$

$r_3 = \begin{pmatrix} 0.743348 \\ -1.11502 \end{pmatrix}$

Iteration 3

$\alpha_3 = 0.309524$

$x_4 = \begin{pmatrix} 1.75222 \\ -2. \end{pmatrix}$

$r_4 = \begin{pmatrix} 0.743348 \\ 0.495565 \end{pmatrix}$

Iteration 4

$\alpha_4 = 0.173333$

$x_5 = \begin{pmatrix} 1.88106 \\ -1.9141 \end{pmatrix}$

$r_5 = \begin{pmatrix} 0.185011 \\ -0.277517 \end{pmatrix}$

Iteration 5

$\alpha_5 = 0.309524$

$x_6 = \begin{pmatrix} 1.93833 \\ -2. \end{pmatrix}$

$r_6 = \begin{pmatrix} 0.185011 \\ 0.123341 \end{pmatrix}$

Iteration 6

$\alpha_6 = 0.173333$

$x_7 = \begin{pmatrix} 1.9704 \\ -1.97862 \end{pmatrix}$

$r_7 = \begin{pmatrix} 0.0460472 \\ -0.0690708 \end{pmatrix}$

Iteration 7

$\alpha_7 = 0.309524$

$x_8 = \begin{pmatrix} 1.98465 \\ -2. \end{pmatrix}$

$r_8 = \begin{pmatrix} 0.0460472 \\ 0.0306981 \end{pmatrix}$

Iteration 8

$\alpha_8 = 0.173333$

$x_9 = \begin{pmatrix} 1.99263 \\ -1.99468 \end{pmatrix}$

$r_9 = \begin{pmatrix} 0.0114606 \\ -0.017191 \end{pmatrix}$

Iteration 9

$\alpha_9 = 0.309524$

$x_{10} = \begin{pmatrix} 1.99618 \\ -2. \end{pmatrix}$

$r_{10} = \begin{pmatrix} 0.0114606 \\ 0.00764043 \end{pmatrix}$

Iteration 10

$\alpha_{10} = 0.173333$

$x_{11} = \begin{pmatrix} 1.99817 \\ -1.99868 \end{pmatrix}$

$r_{11} = \begin{pmatrix} 0.00285243 \\ -0.00427864 \end{pmatrix}$

Iteration 11

$\alpha_{11} = 0.309524$

$x_{12} = \begin{pmatrix} 1.99905 \\ -2. \end{pmatrix}$

$r_{12} = \begin{pmatrix} 0.00285243 \\ 0.00190162 \end{pmatrix}$

Iteration 12

$\alpha_{12} = 0.173333$

$x_{13} = \begin{pmatrix} 1.99954 \\ -1.99967 \end{pmatrix}$

$r_{13} = \begin{pmatrix} 0.000709937 \\ -0.00106491 \end{pmatrix}$

Iteration 13

$\alpha_{13} = 0.309524$

$x_{14} = \begin{pmatrix} 1.99976 \\ -2. \end{pmatrix}$

$r_{14} = \begin{pmatrix} 0.000709937 \\ 0.000473291 \end{pmatrix}$

Iteration 14

$\alpha_{14} = 0.173333$

$x_{15} = \begin{pmatrix} 1.99989 \\ -1.99992 \end{pmatrix}$

$r_{15} = \begin{pmatrix} 0.000176695 \\ -0.000265043 \end{pmatrix}$

Iteration 15

$\alpha_{15} = 0.309524$

$x_{16} = \begin{pmatrix} 1.99994 \\ -2. \end{pmatrix}$

$r_{16} = \begin{pmatrix} 0.000176695 \\ 0.000117797 \end{pmatrix}$

Iteration 16

$\alpha_{16} = 0.173333$

$x_{17} = \begin{pmatrix} 1.99997 \\ -1.99998 \end{pmatrix}$

$r_{17} = \begin{pmatrix} 0.0000439775 \\ -0.0000659663 \end{pmatrix}$

Iteration 17

$\alpha_{17} = 0.309524$

$x_{18} = \begin{pmatrix} 1.99999 \\ -2. \end{pmatrix}$

$r_{18} = \begin{pmatrix} 0.0000439775 \\ 0.0000293184 \end{pmatrix}$

Iteration 18

$\alpha_{18} = 0.173333$

$x_{19} = \begin{pmatrix} 1.99999 \\ -1.99999 \end{pmatrix}$

$r_{19} = \begin{pmatrix} 0.0000109455 \\ -0.0000164183 \end{pmatrix}$

Iteration 19

$\alpha_{19} = 0.309524$

$x_{20} = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$

$r_{20} = \begin{pmatrix} 0.0000109455 \\ 7.29701 \times 10^{-6} \end{pmatrix}$

Iteration 20

$\alpha_{20} = 0.173333$

$x_{21} = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$

$r_{21} = \begin{pmatrix} 2.72422 \times 10^{-6} \\ -4.08633 \times 10^{-6} \end{pmatrix}$

Iteration 21

$\alpha_{21} = 0.309524$

$x_{22} = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$

$r_{22} = \begin{pmatrix} 2.72422 \times 10^{-6} \\ 1.81615 \times 10^{-6} \end{pmatrix}$

Iteration 22

$\alpha_{22} = 0.173333$

$x_{23} = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$

$r_{23} = \begin{pmatrix} 6.78028 \times 10^{-7} \\ -1.01704 \times 10^{-6} \end{pmatrix}$

■ Pictures

```
<< Graphics`Arrow`
(* maxiters, the maximum number of iterations *)
maxiter = 22;
(* i, our counter *)
i = 0;
(* x0, our initial guess for the solution x in Ax = b. *)
x0 = {{-2.}, {-2.}};
(* xi, the approximate solution to x in Ax = b *)
xi = x0;
(* graphlist, a list of graphics that illustrate our algorithm *)
graphlist =
  {ContourPlot[f[{{x}, {y}}], {x, -4, 6}, {y, -6, 4}, ContourShading → False,
    Contours → 20, PlotPoints → 45, DisplayFunction → Identity],
   Graphics[{Disk[Flatten[x0], 0.1], Text["x0", Flatten[x0], {0, 1}]}]};

Print["Input: ", Subscript[x, 0], " = ", x0 // MatrixForm];
(* perform method of steepest descent
  until maxiter loops have been completed *)
While[i ≤ maxiter,
  (* calculate the residual *)
  ri = b - A.xi;
  (* find the appropriate alpha using line search *)
  alphai = Flatten[Transpose[ri].ri / Transpose[ri].A.ri][[1]];
  (* keep the old approximate
    answer and update the approximate answer *)
  oldxi = xi;
  xi = xi + alphai * ri;
  Print["Iteration ", i, "\n----------"];
  Print[Subscript[r, i], " = ", ri // MatrixForm];
  Print[Subscript[α, i], " = ", alphai];
  (* increment counter *)
  i++;
  Print[Subscript[x, i], " = ", xi // MatrixForm];
  (*AppendTo[graphlist,Graphics[Arrow[Flatten[oldxi],
       Flatten[xi],HeadWidth→0.3,HeadLength→0.02]]];*)
  AppendTo[graphlist, Graphics[Line[{Flatten[oldxi], Flatten[xi]}]]];
  Show[graphlist, DisplayFunction → $DisplayFunction];
 ];
Print["Final Graphic\n----------"];
AppendTo[graphlist, Graphics[Disk[Flatten[xi], 0.1]]];
Show[graphlist, DisplayFunction → $DisplayFunction];
```

Input: $x_0 = \begin{pmatrix} -2. \\ -2. \end{pmatrix}$

```
Iteration 0
------------
```

$$r_0 = \begin{pmatrix} 12. \\ 8. \end{pmatrix}$$

$$\alpha_0 = 0.173333$$

$$x_1 = \begin{pmatrix} 0.08 \\ -0.613333 \end{pmatrix}$$

```
Iteration 1
------------
```

$$r_1 = \begin{pmatrix} 2.98667 \\ -4.48 \end{pmatrix}$$

$$\alpha_1 = 0.309524$$

$$x_2 = \begin{pmatrix} 1.00444 \\ -2. \end{pmatrix}$$

```
Iteration 2
------------
```

$$r_2 = \begin{pmatrix} 2.98667 \\ 1.99111 \end{pmatrix}$$

$$\alpha_2 = 0.173333$$

$$x_3 = \begin{pmatrix} 1.52213 \\ -1.65487 \end{pmatrix}$$

```
Iteration 3
------------
```

$$r_3 = \begin{pmatrix} 0.743348 \\ -1.11502 \end{pmatrix}$$

$$\alpha_3 = 0.309524$$

$$x_4 = \begin{pmatrix} 1.75222 \\ -2. \end{pmatrix}$$

```
Iteration 4
------------
```

$$r_4 = \begin{pmatrix} 0.743348 \\ 0.495565 \end{pmatrix}$$

$$\alpha_4 = 0.173333$$

$$x_5 = \begin{pmatrix} 1.88106 \\ -1.9141 \end{pmatrix}$$

```
Iteration 5
------------
```

$$r_5 = \begin{pmatrix} 0.185011 \\ -0.277517 \end{pmatrix}$$

$$\alpha_5 = 0.309524$$

$$x_6 = \begin{pmatrix} 1.93833 \\ -2. \end{pmatrix}$$

Iteration 6

$$r_6 = \begin{pmatrix} 0.185011 \\ 0.123341 \end{pmatrix}$$

$$\alpha_6 = 0.173333$$

$$x_7 = \begin{pmatrix} 1.9704 \\ -1.97862 \end{pmatrix}$$

Iteration 7

$$r_7 = \begin{pmatrix} 0.0460472 \\ -0.0690708 \end{pmatrix}$$

$$\alpha_7 = 0.309524$$

$$x_8 = \begin{pmatrix} 1.98465 \\ -2. \end{pmatrix}$$

Iteration 8

$$r_8 = \begin{pmatrix} 0.0460472 \\ 0.0306981 \end{pmatrix}$$

$$\alpha_8 = 0.173333$$

$$x_9 = \begin{pmatrix} 1.99263 \\ -1.99468 \end{pmatrix}$$

Iteration 9

$$r_9 = \begin{pmatrix} 0.0114606 \\ -0.017191 \end{pmatrix}$$

$$\alpha_9 = 0.309524$$

$$x_{10} = \begin{pmatrix} 1.99618 \\ -2. \end{pmatrix}$$

Iteration 10

$$r_{10} = \begin{pmatrix} 0.0114606 \\ 0.00764043 \end{pmatrix}$$

$$\alpha_{10} = 0.173333$$

$$x_{11} = \begin{pmatrix} 1.99817 \\ -1.99868 \end{pmatrix}$$

Iteration 11

$$r_{11} = \begin{pmatrix} 0.00285243 \\ -0.00427864 \end{pmatrix}$$

$$\alpha_{11} = 0.309524$$

$$x_{12} = \begin{pmatrix} 1.99905 \\ -2. \end{pmatrix}$$

Iteration 12

$$r_{12} = \begin{pmatrix} 0.00285243 \\ 0.00190162 \end{pmatrix}$$

$$\alpha_{12} = 0.173333$$

$$x_{13} = \begin{pmatrix} 1.99954 \\ -1.99967 \end{pmatrix}$$

Iteration 13

$$r_{13} = \begin{pmatrix} 0.000709937 \\ -0.00106491 \end{pmatrix}$$

$$\alpha_{13} = 0.309524$$

$$x_{14} = \begin{pmatrix} 1.99976 \\ -2. \end{pmatrix}$$

Iteration 14

$$r_{14} = \begin{pmatrix} 0.000709937 \\ 0.000473291 \end{pmatrix}$$

$$\alpha_{14} = 0.173333$$

$$x_{15} = \begin{pmatrix} 1.99989 \\ -1.99992 \end{pmatrix}$$

Iteration 15

$$r_{15} = \begin{pmatrix} 0.000176695 \\ -0.000265043 \end{pmatrix}$$

$$\alpha_{15} = 0.309524$$

$$x_{16} = \begin{pmatrix} 1.99994 \\ -2. \end{pmatrix}$$

Iteration 16

$$r_{16} = \begin{pmatrix} 0.000176695 \\ 0.000117797 \end{pmatrix}$$

$$\alpha_{16} = 0.173333$$

$$x_{17} = \begin{pmatrix} 1.99997 \\ -1.99998 \end{pmatrix}$$

Iteration 17

$$r_{17} = \begin{pmatrix} 0.0000439775 \\ -0.0000659663 \end{pmatrix}$$

$$\alpha_{17} = 0.309524$$

$$x_{18} = \begin{pmatrix} 1.99999 \\ -2. \end{pmatrix}$$

Iteration 18

$$r_{18} = \begin{pmatrix} 0.0000439775 \\ 0.0000293184 \end{pmatrix}$$

$$\alpha_{18} = 0.173333$$

$$x_{19} = \begin{pmatrix} 1.99999 \\ -1.99999 \end{pmatrix}$$

Iteration 19

$$r_{19} = \begin{pmatrix} 0.0000109455 \\ -0.0000164183 \end{pmatrix}$$

$$\alpha_{19} = 0.309524$$

$$x_{20} = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$$

Iteration 20

$$r_{20} = \begin{pmatrix} 0.0000109455 \\ 7.29701 \times 10^{-6} \end{pmatrix}$$

$$\alpha_{20} = 0.173333$$

$$x_{21} = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$$

Iteration 21

$$r_{21} = \begin{pmatrix} 2.72422 \times 10^{-6} \\ -4.08633 \times 10^{-6} \end{pmatrix}$$

$$\alpha_{21} = 0.309524$$

$$x_{22} = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$$

Iteration 22

$$r_{22} = \begin{pmatrix} 2.72422 \times 10^{-6} \\ 1.81615 \times 10^{-6} \end{pmatrix}$$

$$\alpha_{22} = 0.173333$$

$$x_{23} = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$$

■ Convergence of Steepest Descent

■ Case when e_i is an eigenvector of our system $A\,x = b$

When $x_0 = (-3, 0.5)^T$ we have convergence on the first iteration since:

```
r0 = b - A.{{-3}, {0.5}};
alpha0 = Flatten[ Transpose[r0].r0 / Transpose[r0].A.r0 ][[1]];
x1 = x0 + alpha0 * r0;
Print[Subscript[r, 0], " = ", r0 // MatrixForm];
Print[Subscript[α, 0], " = ", alpha0];
Print[Subscript[x, 1], " = ", x1 // MatrixForm];
```

$$r_0 = \begin{pmatrix} 10. \\ -5. \end{pmatrix}$$

$$\alpha_0 = 0.5$$

$$x_1 = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$$

And for the error term e_i we have:

```
e0 = x0 - {{2}, {-2}};
e1 = e0 + Flatten[ Transpose[r0].r0 / Transpose[r0].A.r0 ][[1]] * r0;
Print[Subscript[e, 0], " = ", e0 // MatrixForm];
Print[Subscript[e, 1], " = ", e1 // MatrixForm];
```

$$e_0 = \begin{pmatrix} -5. \\ 2.5 \end{pmatrix}$$

$$e_1 = \begin{pmatrix} 0. \\ 0. \end{pmatrix}$$

This is illustrated in the following picture:

```
<< Graphics`Arrow`
Show[
  {ContourPlot[f[{{x}, {y}}], {x, -4, 6}, {y, -6, 4}, ContourShading → False,
    Contours → 20, PlotPoints → 45, DisplayFunction → Identity],
   Graphics[{Disk[{-3, 0.5}, 0.1], Arrow[{-3, 0.5}, {2, -2}]}]},
  DisplayFunction → $DisplayFunction];
```

The Method of Conjugate Directions

```
(* n, as in ℝ^n *)
n = 2;
(* x0, our initial guess for the solution x in Ax = b. *)
x0 = {{-2.}, {-2.}};
xi = x0;
(* ui, our set of linearly independent
  vectors. A judicious choice is the axes of ℝ^n *)
ui = Flatten[Partition[IdentityMatrix[n], {2, 1}], 1];
(* di, the list of search directions *)
di = {ui[[1]]};

(* perform method of Conjugate Directions i = n - 1 *)
For[i = 0, i < n - 1, i++,
  (* calculate the residual *)
  ri = b - A.xi;
  (* find the appropriate alpha *)
  alphai = Flatten[ Transpose[di[[i + 1]].ri / Transpose[di[[i + 1]].A.di[[i + 1]] ]][[1]];
  (* update the approximate answer *)
  xi = xi + alphai * di[[i + 1]];
  (* update the search direction *)
  AppendTo[di,
   ui[[i + 1]] + Sum[-Flatten[ Transpose[ui[[i + 1]].A.di[[k + 1]] / Transpose[di[[k + 1]].A.di[[k + 1]] ]][[1]] * di[[k + 1]], {k, 0, i}]];
  Print["Iteration ", i, "\n------------"];
  Print[Subscript[r, i], " = ", ri // MatrixForm];
  Print[Subscript[α, i], " = ", alphai];
  Print[Subscript[x, i + 1], " = ", xi // MatrixForm];
  Print[Subscript[d, i + 1], " = ", di[[i + 1]] // MatrixForm];
 ];
```

```
Iteration 0
------------
```

$$r_0 = \begin{pmatrix} 12. \\ 8. \end{pmatrix}$$

$$\alpha_0 = 4.$$

$$x_1 = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$$

$$d_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

The Method of Conjugate Gradients

■ Without Pictures

```
(* maxiters, the maximum number of iterations *)
imax = 2;
(* i, our counter *)
i = 0;
(* x0, our initial guess for the solution x in Ax = b. *)
x0 = {{-2.}, {-2.}};
(* xi, the approximate solution to x in Ax = b *)
xi = x0;
(* di, the directional vector *)
di = b - A.x0;
(* ri, the residual *)
ri = di;

Print["Input: ", Subscript[x, 0], " = ", x0 // MatrixForm];
Print["Input: ", Subscript[d, 0], " = ", di // MatrixForm];
Print["Input: ", Subscript[r, 0], " = ", ri // MatrixForm];
(* perform method of steepest descent
  until maxiter loops have been completed *)
While[i ≤ imax,
  Print["Iteration ", i, "\n------------"];
  alphai = Flatten[Transpose[ri].ri / Transpose[di].A.di][[1]];
  Print[Subscript[α, i], " = ", alphai];
  xi = xi + alphai * di;
  Print[Subscript[x, i + 1], " = ", xi // MatrixForm];
  riplus1 = ri - alphai * A.di;
  Print[Subscript[r, i + 1], " = ", riplus1 // MatrixForm];
  betai = Flatten[Transpose[riplus1].riplus1 / Transpose[ri].ri][[1]];
  Print[Subscript[β, i + 1], " = ", betai // MatrixForm];
  di = riplus1 + betai * di;
  Print[Subscript[d, i + 1], " = ", di // MatrixForm];
  ri = riplus1;
  (* increment counter *)
  i++;
 ];
```

```
Input: x0 = ( -2. )
            ( -2. )

Input: d0 = ( 12. )
            (  8. )
```

Input: $r_0 = \begin{pmatrix} 12. \\ 8. \end{pmatrix}$

Iteration 0

$\alpha_0 = 0.173333$

$x_1 = \begin{pmatrix} 0.08 \\ -0.613333 \end{pmatrix}$

$r_1 = \begin{pmatrix} 2.98667 \\ -4.48 \end{pmatrix}$

$\beta_1 = 0.139378$

$d_1 = \begin{pmatrix} 4.6592 \\ -3.36498 \end{pmatrix}$

Iteration 1

$\alpha_1 = 0.412088$

$x_2 = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$

$r_2 = \begin{pmatrix} -4.44089 \times 10^{-16} \\ -8.88178 \times 10^{-16} \end{pmatrix}$

$\beta_2 = 3.40137 \times 10^{-32}$

$d_2 = \begin{pmatrix} -4.44089 \times 10^{-16} \\ -8.88178 \times 10^{-16} \end{pmatrix}$

Iteration 2

$\alpha_2 = 0.142857$

$x_3 = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$

$r_3 = \begin{pmatrix} 0. \\ 0. \end{pmatrix}$

$\beta_3 = 0.$

$d_3 = \begin{pmatrix} 0. \\ 0. \end{pmatrix}$

■ With Pictures

```
<< Graphics`Arrow`
(* maxiters, the maximum number of iterations *)
imax = 2;
(* i, our counter *)
i = 0;
(* x0, our initial guess for the solution x in Ax = b. *)
x0 = {{-2.}, {-2.}};
(* xi, the approximate solution to x in Ax = b *)
xi = x0;
(* di, the directional vector *)
di = b - A.x0;
(* ri, the residual *)
ri = di;
(* graphlist, a list of graphics that illustrate our algorithm *)
graphlist =
  {ContourPlot[f[{{x}, {y}}], {x, -4, 6}, {y, -6, 4}, ContourShading → False,
    Contours → 20, PlotPoints → 45, DisplayFunction → Identity],
   Graphics[{Disk[Flatten[x0], 0.1], Text["x0", Flatten[x0], {0, 1}]}]};

Print["Input: ", Subscript[x, 0], " = ", x0 // MatrixForm];
Print["Input: ", Subscript[d, 0], " = ", di // MatrixForm];
Print["Input: ", Subscript[r, 0], " = ", ri // MatrixForm];
(* perform method of steepest descent
  until maxiter loops have been completed *)
While[i ≤ imax,
  (* we need to keep the old value of xi just to graph it *)
  oldxi = xi;
  Print["Iteration ", i, "\n------------"];
  alphai = Flatten[Transpose[ri].ri / Transpose[di].A.di][[1]];
  Print[Subscript[α, i], " = ", alphai];
  xi = xi + alphai * di;
  Print[Subscript[x, i + 1], " = ", xi // MatrixForm];
  riplus1 = ri - alphai * A.di;
  Print[Subscript[r, i + 1], " = ", riplus1 // MatrixForm];
  betai = Flatten[Transpose[riplus1].riplus1 / Transpose[ri].ri][[1]];
  Print[Subscript[β, i + 1], " = ", betai // MatrixForm];
  di = riplus1 + betai * di;
  Print[Subscript[d, i + 1], " = ", di // MatrixForm];
  ri = riplus1;
  (* increment counter *)
  i++;
  (* show the change from oldxi to xi *)
  (*AppendTo[graphlist,
```

```
    Graphics[Line[{Flatten[oldxi],Flatten[xi]}]]];*)
  AppendTo[graphlist, Graphics[Arrow[Flatten[oldxi], Flatten[xi]]]];
  Show[graphlist, DisplayFunction → $DisplayFunction];
 ];
Print["Final Graphic\n-------------"];
AppendTo[graphlist, Graphics[Disk[Flatten[xi], 0.1]]];
Show[graphlist, DisplayFunction → $DisplayFunction];
```

Input: $x_0 = \begin{pmatrix} -2. \\ -2. \end{pmatrix}$

Input: $d_0 = \begin{pmatrix} 12. \\ 8. \end{pmatrix}$

Input: $r_0 = \begin{pmatrix} 12. \\ 8. \end{pmatrix}$

Iteration 0

$\alpha_0 = 0.173333$

$x_1 = \begin{pmatrix} 0.08 \\ -0.613333 \end{pmatrix}$

$r_1 = \begin{pmatrix} 2.98667 \\ -4.48 \end{pmatrix}$

$\beta_1 = 0.139378$

$d_1 = \begin{pmatrix} 4.6592 \\ -3.36498 \end{pmatrix}$

Iteration 1

$\alpha_1 = 0.412088$

$x_2 = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$

$r_2 = \begin{pmatrix} -4.44089 \times 10^{-16} \\ -8.88178 \times 10^{-16} \end{pmatrix}$

$\beta_2 = 3.40137 \times 10^{-32}$

$d_2 = \begin{pmatrix} -4.44089 \times 10^{-16} \\ -8.88178 \times 10^{-16} \end{pmatrix}$

Iteration 2

$\alpha_2 = 0.142857$

$x_3 = \begin{pmatrix} 2. \\ -2. \end{pmatrix}$

$r_3 = \begin{pmatrix} 0. \\ 0. \end{pmatrix}$

$\beta_3 = 0.$

$d_3 = \begin{pmatrix} 0. \\ 0. \end{pmatrix}$

```
Final Graphic
-------------
```

■ Stopping Criteria

An approximation to the condition number for A_n:

$$\frac{224737}{2} \text{ // N}$$

```
112369.
```

The number of iterations (k) that will give us 10 digits of our answer:

$$\text{Ceiling}\left[\frac{\sqrt{2 * 10^5}}{2} \text{ Log}\left[\frac{2}{10^{-11}}\right]\right]$$

```
5819
```

Preconditioned Conjugate Gradient

The matrix in our example problem has eigenvalues shown below:

```
Eigenvalues[A]
```

$\{7, 2\}$

The condition number for A in our sample problem:

```
Max[Eigenvalues[A]] / Min[Eigenvalues[A]] // N
```

```
3.5
```

■ Choices of a Preconditioner

Consider our sample problem preconditioned by the diagonal matrix:

```
M = DiagonalMatrix[Tr[A, List]];
MA = Inverse[M].A
```

$\left\{\left\{1, \frac{2}{3}\right\}, \left\{\frac{1}{3}, 1\right\}\right\}$

The condition number for our preconditioned problem:

```
Max[Eigenvalues[MA]] / Min[Eigenvalues[MA]] // N
```

```
2.78361
```

Plot the quadratic form:

```
g[x_] := 1/2 Transpose[x].MA.x - Transpose[Inverse[M].b].x + c;
Plot3D[g[{{x}, {y}}], {x, -4, 6}, {y, -6, 4}];
```

The contour plot of the above:

```
ContourPlot[g[{{x}, {y}}], {x, -4, 6}, {y, -6, 4},
 ContourShading → False, Contours → 20, PlotPoints → 45]
```

Let's precondition our matrix using Cholesky factorization:

```
M = CholeskyDecomposition[A];
MA = Inverse[M].A
```

$$\left\{\left\{-2\sqrt{\frac{2}{21}+\sqrt{3}},\ -2\sqrt{\frac{6}{7}+\frac{2}{\sqrt{3}}}\right\},\ \left\{\sqrt{\frac{6}{7}},\ 3\sqrt{\frac{6}{7}}\right\}\right\}$$

The condition number for our preconditioned problem:

```
Max[Eigenvalues[MA]] / Min[Eigenvalues[MA]] // N
```

```
1.24722
```

Plot the quadratic form:

```
h[x_] := 1/2 Transpose[x].MA.x - Transpose[Inverse[M].b].x + c;
Plot3D[g[{{x}, {y}}], {x, -4, 6}, {y, -6, 4}];
```

The contour plot of the above:

```
ContourPlot[h[{{x}, {y}}], {x, -4, 6}, {y, -6, 4},
 ContourShading → False, Contours → 20, PlotPoints → 45]
```

By applying the preconditioner *D* to our SIAM problem we have reduced the condition number to approximately 4.45. Using this we can find the number of iterations (*k*) needed to get 10 digits of our answer

```
Ceiling[ Sqrt[4.45]/2 Log[ 2/10^-11 ] ]
```

```
28
```

■ Algorithm 5.4

```
PreconditionedConjugateGradient[A_, b_, M_, x0_, maxiters_,
   prec_, epsilon_] := Module[{b0 = SetPrecision[b, prec + 5],
    i = 0, r0, d0, alphai, xi, ri, riplus1, betai, mInv},
   r0 = b0 - A.x0;
   ri = r0;
   mInv = Inverse[M];
   d0 = mInv.r0;
   di = d0;
   xi = x0;
   While[i < maxiters,
    alphai = Flatten[Transpose[ri].mInv.ri / Transpose[di].A.di][[1]];
    xi = xi + alphai * di;
    riplus1 = ri - alphai * A.di;
    (* termination test *)
    If[Flatten[Transpose[riplus1].riplus1][[1]] ≤ epsilon, Break[]];
    betai = Flatten[Transpose[riplus1].mInv.riplus1 / Transpose[ri].mInv.ri][[1]];
    di = mInv.riplus1 + betai * di;
    ri = riplus1;
    i++;
   ];
   Print["Approximate Solution = ", N[xi[[1, 1]], prec]];
   Print["Number of Iterations = ", i];
  ];
```

■ n = 200

```
n = 200;
b = Table[{0}, {n}];
b[[1]] = {1};
A =
  SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &@SparseArray[
    Flatten@Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2., n - i]}], n];
M = DiagonalMatrix[Table[A[[i, i]], {i, n}]];
(*M=CholeskyDecomposition[A];*)
x0 = Table[{Random[Integer]}, {n}];
PreconditionedConjugateGradient[A, b, M, x0, 5819, 100, 10^-30]
```

```
Approximate Solution =
 0.72441297988280477496291376579305440116371723164431283291385 33

Number of Iterations = 17
```

LinearSolve solves the problem above quickly and easily. This is just to make sure the above solution is correct.

```
N[LinearSolve[A, b][[1, 1]], 30]
```

```
0.724412979882804776010067247620
```

■ Algorithm 5.4 (Optimized)

```
KindaOptimizedPreconditionedConjugateGradient[
   A_, b_, M_, x0_, maxiters_, prec_, epsilon_] :=
  Module[{b0 = SetPrecision[b, prec + 5], i = 0, r0, d0, alphai,
    xi, ri, riplus1, betai, mInv, trmr, ad, mr1},
   r0 = b0 - A.x0;
   ri = r0;
   mInv = Inverse[M];
   d0 = mInv.r0;
   di = d0;
   xi = x0;
   While[i < maxiters,
    (* this saves us some matrix-vector multiplications *)
    trmr = Transpose[ri].mInv.ri;
    ad = A.di;
    alphai = Flatten[trmr / (Transpose[di].ad)][[1]];
    xi = xi + alphai * di;
    riplus1 = ri - alphai * ad;
    (* save us some more work *)
    mr1 = mInv.riplus1;
    tr1 = Transpose[riplus1];
    (* termination test *)
    If[Flatten[tr1.riplus1][[1]] ≤ epsilon, Break[]];
    betai = Flatten[tr1.mr1 / trmr][[1]];
    di = mr1 + betai * di;
    ri = riplus1;
    i++;
   ];
   Print["Approximate Solution = ", N[xi[[1, 1]], prec]];
   Print["Number of Iterations = ", i];
  ];
```

```
OptimizedPreconditionedConjugateGradient[
   A_, b_, M_, x0_, maxiters_, prec_, epsilon_] :=
  Module[{b0 = SetPrecision[b, prec + 5], i = 0, d0, alphai,
    xi = x0, ri, riplus1, betai, mInv, trmr, ad, mr1},
   ri = b0 - A.xi;
   mInv = SparseArray[Inverse[M]];
   di = mInv.ri;
   While[i < maxiters,
    (* this saves us some matrix-vector multiplications *)
    trmr = Transpose[ri].mInv.ri;
    ad = A.di;
    alphai = Flatten[trmr/(Transpose[di].ad)][[1]];
    xi = xi + alphai * di;
    (* remember this is the step that causes the most error;
     for a large number of iterations we have to recalculate
      the residual using the tried and true method *)
    If[Mod[i, 30] == 0, riplus1 = b - A.xi, riplus1 = ri - alphai * ad];
    (* save us some more work *)
    mr1 = mInv.riplus1;
    tr1 = Transpose[riplus1];
    (* termination test *)
    If[Flatten[tr1.riplus1][[1]] ≤ epsilon, Break[]];
    betai = Flatten[tr1.mr1/trmr][[1]];
    di = mr1 + betai * di;
    ri = riplus1;
    i++;
   ];
   Print["Approximate Solution = ", N[xi[[1, 1]], prec]];
   Print["Number of Iterations = ", i];
  ];
```

■ n = 200

```
n = 200;
b = Table[0, {n}];
b[[1]] = 1;
A =
  SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &@SparseArray[
    Flatten@Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2., n - i]}], n];
M = DiagonalMatrix[Table[A[[i, i]], {i, n}]];
x0 = Table[{Random[Integer]}, {n}];
OptimizedPreconditionedConjugateGradient[A, b, M, x0, 5819, 100, 10^-30]
```

```
Approximate Solution = 0.72441297988280477588361353 3

Number of Iterations = 17
```

■ Timing runs

```
n = 20;
b = Table[0, {n}];
b[[1]] = 1;
A =
  SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &@SparseArray[
    Flatten@Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2., n - i]}], n];
M = DiagonalMatrix[Table[A[[i, i]], {i, n}]];
x0 = Table[{Random[Integer]}, {n}];
Timing[
 OptimizedPreconditionedConjugateGradient[A, b, M, x0, 5819, 100, 10^-12]]
```

```
Approximate Solution =
 0.71742892735383445868568560732449950888367917491504717066109018643550226561237 2

Number of Iterations = 9

{0.031 Second, Null}
```

```
n = 200;
b = Table[0, {n}];
b[[1]] = 1;
A =
  SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &@SparseArray[
    Flatten@Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2., n - i]}], n];
M = DiagonalMatrix[Table[A[[i, i]], {i, n}]];
x0 = Table[{Random[Integer]}, {n}];
Timing[
 OptimizedPreconditionedConjugateGradient[A, b, M, x0, 5819, 100, 10^-12]]
```

```
Approximate Solution = 0.72441298183089251997154680795555096474064884558675896

Number of Iterations = 10

{1.719 Second, Null}
```

```
n = 2000;
b = Table[0, {n}];
b[[1]] = 1;
A =
  SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &@SparseArray[
    Flatten@Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2., n - i]}], n];
M = DiagonalMatrix[Table[A[[i, i]], {i, n}]];
x0 = Table[{Random[Integer]}, {n}];
Timing[
 OptimizedPreconditionedConjugateGradient[A, b, M, x0, 5819, 100, 10^-12]]
```

```
{1438.47 Second, 11}
```

I don't have enough memory to run this!

```
n = 20000;
b = Table[0, {n}];
b[[1]] = 1;
A =
  SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &@SparseArray[
    Flatten@Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2., n - i]}], n];
M = DiagonalMatrix[Table[A[[i, i]], {i, n}]];
x0 = Table[{Random[Integer]}, {n}];
Timing[
 OptimizedPreconditionedConjugateGradient[A, b, M, x0, 5819, 100, 10^-12]]
```

```
No more memory available.
Mathematica kernel has shut down.
Try quitting other applications and then retry.
```

■ The *Mathematica* "one-liner"

Mathematica has built in PCG using **LinearSolve**:

```
n = 20000;
b = Table[0, {n}];
b[[1]] = 1;
A =
  SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &@SparseArray[
    Flatten@Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2., n - i]}], n];
diagonal = Table[A[[i, i]], {i, n}];
prec = 50;
b = SetPrecision[b, prec + 5];
x = LinearSolve[A, b, Method → {Krylov, Method → ConjugateGradient,
      Preconditioner → (# / diagonal &), Tolerance → 10^-(prec-1)}];

N[x[[1]], 50]
```

```
0.72507834626840116746868771925116096886918059447951
```

Interval Arithmetic

SIAM Book Pretty Interval Printer

```
DigitsAgreeCount[a_, b_] := (prec = Ceiling@Min[Precision /@ {a, b}];
   {{ad, ae}, {bd, be}} = RealDigits[#, 10, prec] & /@ {a, b};
   If[ae ≠ be ∨ a b ≤ 0, Return[0]]; If[ad == bd, Return@Length[ad]];
   {{com}} = Position[MapThread[Equal, {ad, bd}], False, 1, 1] - 1; com);
DigitsAgreeCount[Interval[{a_, b_}]] := DigitsAgreeCount[a, b];
IntervalForm[Interval[{a_, b_}]] :=
 (If[(com = DigitsAgreeCount[a, b]) == 0, Return@Interval[{a, b}]];
  start = Sign[a] N[FromDigits@{ad⟦Range@com⟧, 1}, com];
  {low, up} = SequenceForm @@ Take[#, {com + 1, prec}] & /@ {ad, bd};
  If[ae == 0, start /= 10; ae++]; SequenceForm[
   DisplayForm@SubsuperscriptBox[NumberForm@start, low, up], If[ae ≠ 1,
    Sequence @@ {" × ", DisplayForm@SuperscriptBox[10, ae - 1]}, ""]])

n = 20000;
b = Table[0, {n}];
b⟦1⟧ = 1;
A =
  SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &@SparseArray[
    Flatten@Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2., n - i]}], n];

diagonal = Table[A⟦i, i⟧, {i, n}];
prec = 100;
b = SetPrecision[b, prec + 5];
x = LinearSolve[A, b, Method → {Krylov, Method → ConjugateGradient,
      Preconditioner → (# / diagonal &), Tolerance → 10^-prec-1}];

N[x⟦1⟧, 100]
```

```
0.725078346268401167468687719251160968869180594479508957878164769207773 1
 899945962835735923927864782020
```

```
x⟦1⟧ + Interval[{-1, 1}] Norm[b - A.Interval /@ x] // IntervalForm
```

```
0.725078346268401167468687719251160968869180594479506 06996
                                                       82162
```

References

1. The SIAM 100-Dollar 100-Digit Challenge. World Wide Web, December 2006. URL http://www.win.tue.nl/casa/meetings/special/siamcontest/. 23

2. Genetic algorithm. World Wide Web, February 2007. URL http://en.wikipedia.org/wiki/Genetic_algorithm. 42, 43

3. Photon. World Wide Web, February 2007. URL http://en.wikipedia.org/wiki/Photon. 21

4. David Bau III and Lloyd N. Trefethen. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, 1997. 2, 4, 5, 83, 85, 87, 91

5. Paolo Bientinesi, Brian Gunter, and Robert A. Van De Geijn. Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix. World Wide Web, August 2006. URL http://www.cs.utexas.edu/users/flame/pubs/toms_spd.pdf. 65

6. Folkmar Bornemann, Dick Laruie, Stan Wagon, and Jörg Waldvogel. *The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing*. SIAM, 2004. vii, 7, 8, 22, 29, 30, 31, 34, 35, 38, 41, 43, 46, 47, 53, 56, 59, 60, 61, 62, 64, 65, 86, 91, 93

7. Folkmar Bornemann, Dick Laruie, Stan Wagon, and Jörg Waldvogel. The SIAM 100-Digit Challenge Book. World Wide Web, 2004. URL http://www-m3.ma.tum.de/m3old/bornemann/challengebook/. 94

8. James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Berkeley, 1997. 83

9. Peter Deuflhard and Andreas Hohmann. *Numerical Analysis in Modern Scientific Computing: An Introduction*. Springer-Verlag, New York, 2nd edition, 2003. 85

10. Eldon Hansen and G. William Walster. *Global Optimization Using Interval Analysis*, volume 264 of *Monograph and Textbooks in Pure and Applied Mathematics*. Marcel Dekker, Inc., New York, 2nd edition, 2004. 9, 10, 11, 12, 18, 20, 56, 61

11. R. Baker Kearfott. *Rigorous Global Search: Continuous Problems*, volume 13 of *Nonconvex Optimization and Its Applications*. Kluwer Academic Publishers, Dordrect, 1996. 56, 58, 60, 61

12. Sun Microsystems. Interval Arithmetic in High Performance Technical Computing. World Wide Web, September 2002. URL `http://www.sun.com/processors/whitepapers/ia12_wp.pdf`. 10

13. Ramon E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, 1966. 9, 12, 17, 18, 19, 51, 53, 55, 56

14. Ramon E. Moore. A Test for Existence of Solutions to Nonlinear Systems. *SIAM Journal on Numerical Analysis*, 14(4):611–615, September 1977. 57, 58, 59

15. Arnold Neumair. *Interval Methods for Systems of Equations*, volume 37 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, Cambridge, U.K., 1990. 57

16. Edward R. Scheinerman. *Mathematics, A Discrete Introduction*. Brooks & Cole, Pacific Grove, 2000. 5

17. Jonathan R. Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. World Wide Web, August 1994. URL `http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf`. 65, 66, 67, 68, 70, 71, 73, 76, 78, 79, 81, 82, 87, 88, 89

18. Lloyd Trefethen. A Hundred-Dollar, Hundred-Digit challenge. *SIAM News*, 35 (1), 2002. URL `http://www.siam.org/news/news.php?id=388`. iii, 6

19. Henk A. van der Vorst. *Iterative Krylov Methods for Large Linear Systems*. Cambridge Monographs on Applied and Computation Mathematics. Cambridge University Press, Cambridge, U.K., 2003. 81, 83

20. Eric W. Weisstein. Positive definite matrix. World Wide Web, February 2007. URL `http://mathworld.wolfram.com/PositiveDefiniteMatrix.html`. 65