The Book Review Column[1]

by Frederic Green

Department of Mathematics and Computer Science
Clark University
Worcester, MA 01610
email: fgreen@clarku.edu

In this column, we review these three books:

1. **The Problem With Software: Why Smart Engineers Write Bad Code**, by Adam Barr. Answers to a question many of us have probably asked; and what we can do about it. Review by Shoshana Marcus.

2. **Elements of Parallel Computing**, by Eric Aubanel. A textbook on the underpinnings of this important and fascinating area of computer science. Review by Michele Amoretti.

3. **Theorems of the 21st Century: Volume I**, by Bogdan Grechuk. The background, context, and statement of numerous important (accessible) mathematical theorems from the past 20 years. Review by William Gasarch.

I for one can't imagine life in the coronaverse without books to read. Perhaps you feel the same; please contact me to write a review! Choose from among the books listed on the next page. Or choose one of your own. The latter is actually preferable in the current circumstances, as I can then ask the publisher to forward it directly to you.

BOOKS THAT NEED REVIEWERS FOR THE SIGACT NEWS COLUMN

Computability, Complexity, Logic

1. *The Foundations of Computability Theory,* by Borut Robič
2. *Applied Logic for Computer Scientists: Computational Deduction and Formal Proofs*, by Mauricio Ayala-Rincón and Flávio L.C. de Moura.
3. *Descriptive Complexity, Canonisation, and Definable Graph Structure Theory*, by Martin Grohe.

Miscellaneous Computer Science

1. *Elements of Causal Inference: Foundations and Learning Algorithms*, by Jonas Peters, Dominik Janzing, and Bernhard Schölkopf.
2. *CoCo: The colorful history of Tandy's Underdog Computer* by Boisy Pitre and Bill Loguidice
3. *Introduction to Reversible Computing*, by Kalyan S. Perumalla
4. *A Short Course in Computational Geometry and Topology*, by Herbert Edelsbrunner
5. *Partially Observed Markov Decision Processes,* by Vikram Krishnamurthy
6. *Statistical Modeling and Machine Learning for Molecular Biology*, by Alan Moses
7. *Language, Cognition, and Computational Models,* Theirry Poibeau and Aline Villavicencio, eds.
8. *Computational Bayesian Statistics, An Introduction,* by M. Antónia Amaral Turkman, Carlos Daniel Paulino, and Peter Müller.
9. *Variational Bayesian Learning Theory,* by Shinichi Nakajima, Kazuho Watanabe, and Masashi Sugiyama.
10. *Programming for the Puzzled: Learn to Program While Solving Puzzles*, by Srini Devadas.
11. *Knowledge Engineering: Building Cognitive Assistants for Evidence-based Reasoning*, by Gheorghe Tecuci, Dorin Marcu, Mihai Boicu, and David A. Schum.

Combinatorics and Graph Theory

1. *Finite Geometry and Combinatorial Applications*, by Simeon Ball
2. *Introduction to Random Graphs*, by Alan Frieze and Michał Karoński
3. *Erdős –Ko–Rado Theorems: Algebraic Approaches*, by Christopher Godsil and Karen Meagher
4. *Combinatorics, Words and Symbolic Dynamics,* Edited by Valérie Berthé and Michel Rigo

Miscellaneous Mathematics

1. *Introduction to Probability*, by David F. Anderson, Timo Seppäläinen, and Benedek Valkó.

Review of[2]
The Problem With Software: Why Smart Engineers Write Bad Code
by Adam Barr
MIT Press, 2018
$30.00, Hardcover, 305 pages

Review by

Shoshana Marcus (shoshana.marcus@kbcc.cuny.edu)
Department of Mathematics and Computer Science
Kingsborough Community College, CUNY

1 Overview

As we become progressively more dependent on computers and the software they run, the threat of bugs in computer systems is more and more concerning. This book begins by describing attacks of different viruses in 1988 and 2014 that exploited similar programming errors. These vulnerabilities in the respective programs resulted from simple mistakes and their fixes seemed trivial. It is astounding to realize that in the intervening twenty five years no standard methodology was implemented to detect this sort of problem and that the programming community did not learn from their collective mistakes made in the recent past. Upon reflection, this seems to be the accepted norm in software engineering. As the author points out, "Software has few of the hallmarks of engineering where a body of knowledge is built up over time based on rigorous experimentation." This book is a quest to understand why that is so and to explore how we can begin to rectify this wrong.

This book was written to raise awareness of fundamental flaws in the field of software engineering. Given his experience at Microsoft, the author is able to recognize systemic flaws in the way that software is developed. These often lead to disastrous results. There seems to be a divide between the way software developers are groomed in the academic setting and the rigor required of the computer science graduates when they join the workforce as software developers. This book weaves a story and provides the reader with a historical backdrop for the current trends in computer programming. The author gives perspective for the reader to realize that computer science professors are not in sync with the software development processes employed in industry and the more general skills that are required of programmers. The goal of this book is to begin to bridge this gap by opening conversation between industry and academia so they can join forces and produce more rigorous and reliable software engineering practices.

2 Summary

Most of this book focuses on painting a historical backdrop to the current trends in computer science. The author participated in the evolution of computer programming from mainframes to PCs, the development of the internet and now cloud computing. Given the author's wide array of software development experiences on a variety of different hardware settings, the personal historical lens prepares the reader to be receptive to the author's commentary on how and why the current state of software engineering developed and how the wrongs can be fixed. Throughout these chapters, it is fun to read about what actually goes on at code reviews and glimpse behind the scenes at Microsoft development.

Throughout this book we are urged to look at the software development life cycle through a critical lens. The author looks back fondly at his undergraduate years at Princeton. As an undergraduate, he became a seasoned coder by hacking away in the lab on his own. During his early career at Microsoft, the author gradually realized that the undergraduate computer science curriculum of the time did not directly prepare him for the skills he needed to succeed in industry. He considers himself and his generation of computer programmers to be self-taught, which has had and continues to have both advantageous and detrimental effects on the way they developed software and have overseen software development.

This book explains in great detail what an application Programmer Interface (API) is and why it is so important. Programs are developed by teams and often libraries of code are reused at a later point in time than they were developed. Bugs are frequently introduced across the layers of code, when they do not communicate well. Although it is important, documentation is not sufficient to prevent misuse of established code. Although unit testing is often employed in software engineering to demonstrate that code is correct, unit testing is not enough because it does not test across the layers of code to make sure that they interact with one another correctly. This book explains common program errors like variable overflow in great detail so that the ideas can be conveyed clearly to any reader regardless of his background in computer science, or lack thereof.

It is widely known that the efficiency of the popular programming language C stems from its flexibility in allowing the programmer to manage memory directly. This very advantage is a vulnerability that makes programs written in C susceptible to certain kinds of hacks. This is particularly true of the popular operating systems widely in use today which rely on kernels written in C. Although difficult to detect once a program is complete since these errors do not occur regularly, programmers need to be vigilant that these low-level errors do not creep into their programs.

This book devotes separate chapters to the development of object oriented programming and to Agile programming. It is interesting to join the author on his tour and to realize that these technologies are not as innovative as they are billed to be. It is also interesting to realize that the object oriented Design Patterns book written by the Gang of Four was produced by a successful partnership between industry and academia.

One observation that is encountered several times throughout the book seems to be a significant impediment to programmers who are trying to keep up with deadlines and produce working code quickly The techniques learned in the classroom for small, short-term projects do not readily scale to an industrial setting in which large teams work on a single programming project over many years. Although this point seems obvious, it seems that thus far this obstacle has not been granted sufficient attention and this book is a cry to rectify this state of affairs.

This book concludes with a set of ideas that the author proposes to rectify the wrongs in software engineering and to enable us to have better programs more of the time. Many of these ideas sound promising and have great merit. Yet we can't lose sight of the fact that these suggestions are merely the cherry on top of this book. The primary goal of this book is to raise awareness of an important issue and to make academia more aware of industry, and vice versa.

Looking to the future, Barr writes that the best prospect for improving software engineering is the move to the cloud. One of the hindrances to continuous improvement of programs is the lack of profitability. Traditionally, software developers sell their product in a package and collect a one-time fee when the software is delivered. When software is a delivered service and not sold as a product, companies should have more incentive to constantly improve their work rather than be satisfied with a product that is deemed "good enough to ship" so that customers continue to subscribe to the service.

We are lulled into complacency. As the author points out, "If you use the number of millionaires as your metric for determining the success of an industry, software engineering appears to be doing great. But that

doesn't mean we don't need to change it." This book serves as a wake-up call.

There is one central theme in this book and that is to explore the question, "Is software development really hard, or are software developers not that good at it?" The author takes us on a tour through the trends and techniques of the various programming languages used in the past sixty years. As we travel from mainframes to PCs to cloud computing, we see common themes emerge at different times from the various software development tools. As the author sums up at the end of the book, "Certainly there are parts of software that are hard, but software developers seem to do everything in their power to make even the easy parts harder by wasting an inordinate amount of time on reinvention and inefficient approaches. A lot of mistakes are in fundamental areas that should be understood by now, and so the software industry needs to push to figure them out and then teach people about them, so we can devote our energy to the parts that really are hard."

3 Opinion

The contents of this book are accessible to anyone with any or no programming background. Actual code, with clear explanations of its execution, and concise diagrams makes the technical content very clear and accessible to all audiences, even to those who are not proficient programmers.

As an academic educating the next generation of programmers, I found it useful to hear about the previous generation in hindsight. The text is sprinkled liberally with quotes from prominent academics as well as famous players in industry. I found it intriguing and validating to hear from Donald Knuth and Edsger Dijkstra throughout the book.

After reading this book, I find it compelling to think about revamping the American computer science education. This book has convinced me that it would make sense to make fundamental changes to the curriculum. The unconventional idea that more technical skills should be taught at the undergraduate level and the more theoretical and specialized courses should be saved for a graduate course of study resonated well with me. Fundamental changes that should occur at the undergraduate curriculum to ensure that the next generation of computer programs will be more robust and less prone to error than we have seen yet. This is especially important as we gradually become more reliant on software in more aspects of our day-to-day life.

I certainly gained a lot from reading this book. I hope that this will translate to a benefit for my students.

Review of[3] Elements of Parallel Computing
by Eric Aubanel
Chapman & Hall/CRC, 2016
238 pages, Hardcover, $172.00

Review by

Michele Amoretti (michele.amoretti@unipr.it)
Department of Engineering and Architecture
University of Parma

1 Introduction

As the title clearly states, this book is about parallel computing. Modern computers are no longer characterized by a single, fully sequential CPU. Instead, they have one or more multicore/manycore processors. The purpose of such parallel architectures is to enable the simultaneous execution of instructions, in order to achieve faster computations. In high performance computing, clusters of parallel processors are used to achieve PFLOPS performance, which is necessary for scientific and Big Data applications.

Mastering parallel computing means having deep knowledge of parallel architectures, parallel programming models, parallel algorithms, parallel design patterns, and performance analysis and optimization techniques. The design of parallel programs requires a lot of creativity, because there is no universal recipe that allows one to achieve the best possible efficiency for any problem.

The book presents the fundamental concepts of parallel computing from the point of view of the algorithmic and implementation patterns. The idea is that, while the hardware keeps changing, the same principles of parallel computing are reused. The book surveys some key algorithmic structures and programming models, together with an abstract representation of the underlying hardware. Parallel programming patterns are purposely not illustrated using the formal design patterns approach, to keep an informal and friendly presentation that is suited to novices.

2 Summary of Contents

The book has two parts. In the first part (Chapters 1–5), the core concepts are presented. In the second part (Chapters 6–8), three case studies are presented with the purpose to reinforce the concepts of the earlier chapters. Every chapter ends with a section about further reading and a section of exercises.

1. Chapter 1: Overview of Parallel Computing. This chapter starts with an introduction to parallel computing, motivating the need for a new approach based on a suitable level of abstraction and the awareness that a limited number of solutions keep being re-used. Then, terminology is made clear, with reference to Flynn's taxonomy. The evolution of parallel computers is shortly presented, with particular emphasis on the cluster-grid-cloud progression and on the emergence of multicore and manycore processors. Then, parallel programming models are divided in three categories (implicit, semi-implicit and explicit) that are shortly presented. The relevance of *thinking in parallel* is outlined. The chapter includes a discussion about the parallel implementation of the high level patterns defined in the OPL (Our Pattern Language) hierarchy. The top two layers, namely Structural Patterns

[3]

and Computational Patterns, are presented. The chapter ends with the presentation of the algorithm that solves the word count problem as an instance of the MapReduce pattern, with a discussion on distributed and shared memory implementations.

2. Chapter 2: Parallel Machine and Execution Models. The first part of this chapter discusses three machine models, i.e., SIMD (Single Instruction Multiple Data), shared memory and distributed memory. SIMD ALUs are illustrated by means of register-level code examples. The difference between conventional SIMD and SIMT (Single Instruction Multiple Threads), characterizing NVIDIA GPUs, is also illustrated. Regarding MIMD (Multiple Instruction Multiple Data) parallel computers, the difference between multicore processors, multiprocessors and multicomputers is discussed. Shared memory execution and distributed memory execution are illustrated by means of simple examples. Regarding the former model, the concepts of data race, sequential consistency and cache coherence are presented. The second part of the chapter presents a task graph execution model defined as a directed acyclic graph, where each vertex represents the execution of a task, and an edge between two vertices represents a real data dependence between the corresponding tasks. Some simple examples of task graphs are explored, to get a feeling for how this execution model represents dependencies between tasks. The particular and highly relevant case of *reduction* is also presented, with reference to the word count example proposed in Chapter 1. The chapter ends with a discussion on mapping a task graph to a given machine.

3. Chapter 3: Parallel Algorithmic Structures. This chapter is about common patterns in parallel algorithm design, whose general guidelines are presented as a premise. The *embarrassingly parallel* pattern is considered, which applies to cases where the decomposition into independent tasks is self-evident. Two examples are proposed: the Monte Carlo estimation of π and the generation of fractal images. The second pattern being considered is *reduction*, which is illustrated by means of the sum reduction example. The corresponding task graph and data structures are given, considering both distributed and shared memory implementations. As a practical application of the pattern, k-means clustering is presented. Then, the *scan* pattern is illustrated, with particular reference to the implementation of the prefix sum case. The fourth pattern is *divide-and-conquer*, which is discussed and compared to reduction, and used to implement a clever parallel version of the merge sort algorithm. The penultimate pattern is *pipeline*, whose presentation is supported by three examples of increasing difficulty (the last one being pipelined merge sort, which is compared to the divide-and-conquer one). Last but not least, the *data decomposition* pattern is illustrated, which is most suitable when the same operation is performed on elements of a data structure. Four applications are presented: Conway's Game of Life, matrix-vector multiplication, bottom-up dynamic programming and pointer jumping.

4. Chapter 4: Parallel Program Structures. Coming back to the three machine models introduced in Chapter 2 (SIMD, shared memory and distributed memory), this chapter examines how they influence parallel algorithm design and implementation. The concept of *load balance* is introduced and used throughout the chapter. SIMD *strict data parallel* programming, where a single stream of instructions executes in lockstep on elements of arrays, is illustrated by means of row-wise matrix-vector multiplication and other algorithms. The section ends with guidelines to consider when relying on strict data parallel execution on SIMD platforms. The *fork-join* pattern, which is well-suited for implementing divide-and-conquer algorithms in shared memory systems, is presented by means of two examples, namely the parallel fork-join estimation of π and the parallel fork-join merge sort. Also for this implementation pattern, useful guidelines are provided. Another important pattern for the shared memory model, *parallel loop* is illustrated by means of two examples, i.e., fractals and subset sum. Particular

attention is devoted to variable scoping, thread synchronization and thread safety. Next section is about parallel language models that enable specification of *task dependencies*, which allows runtime software to schedule tasks more efficiently than a programmer can. The topic is illustrated with the help of an example, namely parallel subset sum with task dependencies. Then, the *single program multiple data (SPMD)* model is presented, which applies to both distributed and shared memory programming. In particular, it is the basis for parallel programming on GPUs. The presentation of this important model is supported by three examples, i.e., round-robin SPMD parallel fractal generation, SPMD merge sort and reduction on GPU. The penultimate section is devoted to the *master-worker* pattern, which is a common pattern for dynamic load balancing of independent tasks, frequently used with SPMD programming. It is very easy to describe and to use, but does not scale well as the number of workers increases. This aspect is discussed at the end of the section. Finally, the last section focuses on important aspects of distributed memory programming, such as distributed arrays, message passing, local vs. global communication, and the MapReduce pattern.

5. Chapter 5: Performance Analysis and Optimization. The first section discusses the *work-depth* analysis of the task graph. Definitions are given for depth, work and parallelism. These concepts are illustrated by means of some of the examples presented in Chapters 3 and 4. Then, the second section illustrates algorithm performance once tasks are mapped to a computing platform. The most common performance metrics are introduced, i.e., *speedup*, *cost*, *efficiency*, as well as the FLOPS and CUPS throughput metrics. The examples used to illustrate the scan pattern, in Chapter 3, are analyzed using these metrics. A specific subsection is devoted to communication analysis, which is presented by means of the following use cases: reduction, grid-based parallel applications like Conway's Game of Life, and distributed arrays. The third section is devoted to barriers to performance, at both the algorithmic and implementation levels: load imbalance, communication overhead and false sharing are discussed. As a viable approach for reducing communication, it is proposed to create hierarchical algorithms to match hierarchical computing platforms. Then, *Amdahl's Law* is presented, which allows one to compute the speedup as a function of the fraction of the execution time of the operations that cannot be done in parallel. The last section is devoted to the use of performance profiling tools.

6. Chapter 6: Single Source Shortest Path. Problems on graphs arise in several different contexts. Solving them efficiently is very important. This chapter focuses on the problem denoted as *single source shortest path* (SSSP), consisting in finding the shortest path between one vertex and all others. The considered algorithms are suitable for all graphs with non-negative real-valued edge weights. In the first section, three sequential algorithms are presented, namely Bellman-Ford, Dijkstra's and delta-stepping. The second section is devoted to parallel design exploration. Parallel versions of the aforementioned algorithms based on task decomposition are illustrated and compared in terms of work, depth and parallelism. Then, it is outlined that distributed memory implementations require the decomposition of the problem graph, and it is suggested to partition the graph into subgraphs of roughly the same size, maximizing the number of boundary vertices and minimizing the maximum path length from boundary vertices and the interfaces between subgraphs. Finally, three parallel algorithms are detailed, namely shared memory delta-stepping, SIMD Bellman-Ford for GPU, and a message passing algorithm.

7. Chapter 7: The Eikonal Equation. Wave propagation is everywhere in nature and has inspired many problem solving tools. Level set methods to track moving fronts have been the most successful. This chapter refers to the case where the problem is stationary. In the first section, a numerical framework is introduced for solving the *eikonal equation* and obtaining the plot of the stationary front. Then, two

sequential algorithms are presented and discussed, namely the Fast Sweeping Method (FSM) and the Fast Marching Method (FMM). In the second section, parallel design is explored. Two approaches for modifying the sequential FSM algorithm to expose independent tasks are presented and analyzed in terms of performance: relaxing dependencies and reordering sweeps. Two parallel designs of the FMM algorithm are presented, namely the Fast Iterative Method (FIM) and the domain decomposed FMM. In the third section, it is discussed how the aforementioned designs can be developed into algorithms for the three machine models that have been used throughout the book, i.e., SIMD, shared memory and distributed memory.

8. Chapter 8: Planar Convex Hull. This last chapter is devoted to parallel algorithms for computing the *convex hull* of a set of points S, i.e., the intersection of all convex sets containing S. In particular, the planar case is considered. Two sequential algorithms are presented, namely QuickHull and MergeHull. Parallel design of these algorithms is discussed in the second section, starting from the most obvious task decomposition, which is divide-and-conquer. A long subsection is devoted to further improving the parallel MergeHull, resulting in an algorithm with optimal parallelism. The third section is devoted to the presentation and discussion of SIMD QuickHull, shared memory SMPD MergeHull, and distributed memory MergeHull.

3 Opinion

The declared goal of *Elements of Parallel Computing* is to guide the reader toward being able to think in parallel, leveraging recent work on patterns in parallel algorithm design to identify the solutions that keep being re-used. The book presents the fundamental concepts of parallel computing not from the point of view of hardware, but from a more abstract view. It takes a language-neutral approach using pseudocode, which can be easily adapted for implementation in relevant languages such as C and C++. The chapters have references to real parallel language models like MPI, OpenMP and CUDA.

I think that students in their final year of bachelor's studies or in their first year of a master's in computer science or computer engineering would benefit from this book. Personally, in the last two years I have recommended it to the students that attend my course in High Performance Computing (MS in computer engineering). The book provides many examples of how to move from the problem statement to the design of an effective task decomposition and to different algorithms, depending on the machine model (SIMD, shared memory or distributed memory). Importantly, there are homework assignments at the ends of the chapters that may help the students to exercise the knowledge they acquire. Moreover, the book encourages further reading on advanced topics, providing high-quality references.

Performance analysis of parallel algorithms is adequately presented, including the most common performance metrics and some nontrivial examples. If I really have to find a defect in this book it is the lack of a discussion on the complexity class NC and on cost optimal NC algorithms, especially those that achieve sublogarithmic time. Maybe in a future edition this gap will be filled.

In conclusion, the book is technically correct and well written, with a fair balance between conciseness and completeness. I think that it may be a constructive and enjoyable experience for the readers that are interested in learning how to design and implement highly efficient parallel programs.

Review of[4]
Theorems of the 21st Century: Volume I
by Bogdan Grechuk
Publisher: Springer-Verlag, 2019
$30.00 hardcover, 446 pages

Review by
William Gasarch (gasarch@umd.edu**)**
Department of Computer Science
University of Maryland, College Park

1 Introduction

In the movie *Oh God! Book II*, God (played by George Burns) says:

Mathematics, that was a mistake. I should have made the whole thing a little easier.

While I am not one to argue with God, George Burns, or George Burns playing God, I do not think Mathematics was a mistake. But I do wish it was easier. Or perhaps it's easy enough to make so much progress in that it becomes hard.

In the 21st century there are many theorems whose statements *cannot* be explained to a good undergraduate math major. Or even a professor of mathematics who is not in that area. But all is not bleak. There are also theorems whose statements *can* be explained to a good undergraduate math major, though you might need to supply some context.

The book under review is about those theorems. Every chapter gives definitions and context for a theorem proven in the 21st century, in 4–6 pages. No proofs are given. The chapters are organized by year. There are 106 chapters total.

In this review I will summarize five of the chapters. I intentionally *do not* summarize any of the chapters that involve (1) theoretical computer science, or (2) you've probably already heard of (e.g., there are arbitrarily long arithmetic sequences of primes) since the biggest benefit of this book is *broaden your horizons.* After the summaries I will have a section that lists all of the chapters in this book that involve TCS. I will then render an opinion. Spoiler Alert: this book is awesome.

2 There is Also a Website!

The author has put up a website which has the theorems in the book, more theorems which will appear in a sequel, along with pointers to the papers where they were proven. The website is here:

https://theorems.home.blog/theorems-list/

3 Counting Integer Solutions of Some Inequalities (2001)

Consider the region

[4]

$$D = \{(x_1, \ldots, x_n) : x_1^2 + \cdots + x_n^2 \leq m\}.$$

The number of integer point in D is well approximated by the volume of D. We generalize this notion.

Definition 3.1 *Let $f(x_1, x_2)$ be a homogenous polynomial. An inequality of the form $|f(x_1, x_2)| \leq m$ is of finite type if (1) the area of its set of real solutions is finite, and (2) the intersection of it with any line with rational coefficients has finite length. Note that the disc is an example. One can easily generalize the definition to higher dimensions. For example, in 3-dimensions it would be (1) the volume of its set of real solutions is finite, (2) the intersection of it with any plane with rational coefficients has finite area, and (3) the intersection of it with any line with rational coefficients has finite length.*

Definition 3.2 *Let $f(x_1, \ldots, x_n)$ be homogeneous of degree d. Then f is decomposable if there exists complex linear polynomials $L_1(x)$, $L_2(x)$, ..., $L_d(x)$ such that*

$$f(x_1, \ldots, x_n) = L_1(x) \cdots L_d(x).$$

ALSO: all of the coefficients of all of the L_i's are non-zero. So you cannot have, for example $x_1 + 3x_4$ as one of the linear factors.

The big theorem says, in essence, that under certain conditions the volume of an n-dim solid is a good approximation for the number of lattice points in it.

Theorem 3.3 *(Jefferey Lin Thunder) Let $F(x_1, \ldots, x_n) \in \mathbb{Z}[x_1, \ldots, x_n]$ be a homogeneous polynomial of degree d. Let F be decomposable. Let $N_F(m)$ be the number of integer solutions to $|F(x_1, \ldots, x_n)| \leq m$.*

1. *$(\forall m)[N_F(m) < \infty]$ iff F is of finite type.*
2. *If F is of finite type then $|N_F(m)| \leq c(n, d)m^{n/d}$.*

4 The Existence of a Field with u-invariant 9

Let F be a field. Solving linear equations over F is easy. We look at quadratic equations of the form

$$\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} x_i x_j = 0$$

and seek non-zero solutions. If $n = 1$ then the equation is $a_{11}x^2 = 0$ so there are no non-trivial solutions. If the number of variables goes up, do we get a solution?

Definition 4.1 *Let F be a field. The u-invariant of F is the number u such that*

1. *There is an quadratic form with n variables that has NO non-trivial solution.*
2. *Every quadratic form of degree $u + 1$ has a non-trivial solution.*

Examples:

1. $F = \mathbb{R}$. For all n, there is a quadratic form on n variables with NO nontrivial solution: $x_1^2 + \cdots + x_n^2 = 0$. So $u(\mathbb{R}) = \infty$.

2. $F = \mathbb{C}$. $x^2 = 0$ only has the trivial solution, but then $ax^2 + bxy + c^2 = 0$ always has a nontrivial solution. So $u(\mathbb{C}) = 1$.

3. Let $F = \mathbb{Z}_p$ where p is prime, $p \geq 3$.

   - Let a be a NON-square in $\mathbb{Z}_p$. Then $x^2 - ay^2 = 0$ has no solution since if it did then $a = (\frac{x}{y})^2$.
   - One can show that every quadratic form in 3 variables has a non-trivial solution.
   - Hence, for all primes $p \geq 3$, $u(\mathbb{Z}_p)) = 2$.

4. It is known that $u(\mathbb{Q}(i)) = 4$.

5. It is known that $u(\mathbb{C}(x_1, \ldots, x_n)) = 2^n$.

Note that all of the $u(F)$ are powers of 2. In 1953 Kaplansky conjectured that, for all fields F, $u(F)$ is a power of 2. In 1989 Merkurev disproved this by showing there was a field F such that $u(F) = 6$. He also showed that for every even number $2k$ there is a field F with $u(F) = 2k$. It was also shown that $u(F)$ cannot be 3, 5, or 7. This naturally leads to the conjecture that the numbers that can be $u(F)$ are exactly the evens.

Alas, in 2003 Izhboldin showed the following:

Theorem 4.2 *There is a field F with $u(F) = 9$.*

5 Counting Rational Functions with Given Critical Points (2002)

We consider $\mathbb{R}(x)$, the set of rational functions over $\mathbb{R}$. Such a function is *of degree d* if both the numerator and denominator are of degree d. We denote the set of such functions $\mathbb{R}_d(x)$. A *critical point* of $f \in \mathbb{R}(x)$ is a place where its derivative is 0. Since $f \in \mathbb{R}_d(x)$ has both numerator and denominator of degree d, there are $\leq 2d - 2$ critical points (counting multiplicities).

Given d and the set of critical points, can you recover the function? The answer is no for two reasons:

1. The question as phrased isn't quite what we want. If $f, g \in \mathbb{R}_d(x)$ but there is a linear rational function L such that $f(x) = L(g(x))$ then we want to consider f and g the same. So we restate the question: does the set of critical points determine the equivalence class of functions?

2. There are sets of critical points so that there is more than one equivalence class with that set. But there are only a finite number of equivalence class.

Theorem 5.1 *(Andrei Gabrielov) For all d, for all sets of $2d - 2$ critical points, there exists exactly*

$$\frac{1}{d}\frac{(2d-2)!}{(d-1)!(d-1)!}$$

equivalence classes of rationals of degree d with those critical points.

6 A Real Number that is Far from All Cubic Algebraic Integers (2003)

If $\zeta \notin \mathbb{Q}$ we seek $\frac{a}{b} \in \mathbb{Q}$ such that $|\zeta - \frac{a}{b}|$ is small. How to measure smallness? Clearly, the larger b is the closer we can make ζ and $\frac{a}{b}$. Hence we want to measure the distance as a function of b. The following are known.

1. For every $\zeta \notin \mathbb{Q}$ there exists infinitely many $\frac{a}{b} \in \mathbb{Q}$ such that $|\zeta - \frac{a}{b}| \leq \frac{1}{\sqrt{5}b^2}$. We would like to improve the 2 to a larger number. But we can't due to the next item.

2. There exists ζ such that, for every $\frac{a}{b} \in \mathbb{Q}$, $|\zeta - \frac{a}{b}| \geq \frac{1}{\sqrt{5}b^2}$. One such ζ is $\frac{1+\sqrt{5}}{2}$, the Golden Ratio..

In other words, you can always get quadratic error, but there are times you can't do any better than that.

The above is about approximating irrationals with rationals. What if we approximate irrationals with numbers that involve square roots of integers? Cube Roots? Let

$$\mathbb{Q}_d = \{x \in \mathbb{R} : \ x \text{ is the root of a degree } d \text{ polynomial with coefficients in } \mathbb{Z}\}.$$

We want to approximate $\zeta \in \mathbb{Q}$ by some $\alpha \in \mathbb{Q}_2$. So we want

$$|\zeta - \alpha| \leq ???$$

We need some measure of how complicated the number α is. For $\frac{a}{b}$ we used b. Note that $\frac{a}{b}$ satisfies $bx - a = 0$. So we could rephrase this as using the highest coefficient in a poly of least degree that b satisfies. This is how we generalize. Let $\alpha \in \mathbb{Q}_2$. Look at all of the quadratic equations over $\mathbb{Z}$ that α solves. Take the one with the smallest largest coefficient. That coefficient is $H(\alpha)$.

1. For every $\zeta \notin \mathbb{Q}$ there exists infinitely many $\alpha \in \mathbb{Q}_2$ such that $|\zeta - \alpha| \leq \frac{C}{H(\alpha)^2}$. This is not impressive. We wanted to improve 2 to a large number. But we can't due to the next item.

2. There exists a constant D and a $\zeta \notin \mathbb{Q}$ such that, for every $\alpha \in \mathbb{Q}_2$, $|\zeta - \alpha| \geq \frac{D}{H(\alpha)^2}$. Darn!

What if we approximate irrationals with numbers in $\mathbb{Q}_3$?

1. For every $\zeta \notin \mathbb{Q}$ there exists infinitely many $\alpha \in \mathbb{Q}_3$ such that $|\zeta - \alpha| \leq \frac{C}{H(\alpha)^{\phi^2}}$ where $\phi = \frac{1+\sqrt{5}}{2}$. Note that $\phi^2 = \frac{3+\sqrt{5}}{2} \sim 2.618$. Yes! We broke he barrier of 2. Well not US, but actually Davenport and Schmidt in 1969. Can we do better? Can they do better? Can anyone do better? In 2003 the answer came: No.

Theorem 6.1 *(Damien Roy) There exists a constant D and a $\zeta \notin \mathbb{Q}$ (in fact, ζ is transcendental) such that, for every $\alpha \in \mathbb{Q}_3$, $|\zeta - \alpha| \geq \frac{D}{H(\alpha)^{\phi^2}}$.*

7 On Divisibility Properties of Dyson's Rank Partition Function (2010)

A *partition* of a natural number is a way to write it as a sum of naturals. For example, $5 = 1 + 2 + 2$ is a partition of 13. Let $p(n)$ be the number of partitions of n. We do NOT care about the order. For example, the following is the set of all partitions of 5 $(1, 1, 1, 1, 1)$, $(1, 1, 1, 2)$, $(1, 1, 3)$, $(1, 2, 2)$, $(1, 4)$, $(2, 3)$. Note that $p(5) = 6$. The following is known.

Theorem 7.1 *For all $n \in \mathbb{N}$:*

1. *(Ramanjuan)* $p(5n+4) \equiv 0 \pmod 5$.
2. *(Ramanjuan)* $p(7n+5) \equiv 0 \pmod 7$.
3. *(Ramanjuan)* $p(11n+6) \equiv 0 \pmod{11}$.
4. *(Atkin and O'Brien)* $p(17303n+237) \equiv 0 \pmod{13}$.
5. *(Ono) For all primes $q \geq 5$ there exists $A, B \in \mathbb{N}$ such that $p(An+B) \equiv 0 \pmod q$.*

By the above $p(9) \equiv 0 \pmod 5$ and in fact $p(9) = 30$. So one can take all of the partitions of 9 and divide them arbitrarily into 5 groups of size 6. Dyson outlined a way to do this non-arbitrarily.

For each partition take the difference between the largest summand and the number of summands. We call this the *rank of the partition*. We list the ranks of some of the partitions of 9:

$(2,2,1,1,1,1)$ has rank $2-7=-5$
$(3,3,3)$ has rank $3-3=0$
$(4,2,2,1)$ has rank $4-4=0$
$(4,3,1,1)$ has rank $4-4=0$
$(5,1,1,1,1)$ has rank $5-5=0$
$(7,2)$ has rank $7-2=5$

So these 6 partitions all have rank $\equiv 0 \pmod 5$. It turns out that no other partitions map to a number $\equiv 0 \pmod 5$. In fact, each of $i = 0,1,2,3,4$ has exactly 6 partitions of rank $\equiv i \pmod 5$. Is this always the case? Dyson conjectured yes. We state this formally.

Let $N(r,q;m)$ be the number of partitions of m which have a rank that is $\equiv r \mod q$. Dyson conjectured that

$$N(0,5;5n+4) = N(1,5;5n+4) = N(2,5;5n+4) = N(3,5;5n+4) = N(4,5;5n+4) = \frac{p(5n+4)}{5}$$

This was proven by Atkin and Swinnerton-Dyer in 1954. They also proved a similar theorem for $p(7n+5)$. Is there a similar theorem for $p(An+B)$? Yes but this result is from 2010.

Theorem 7.2 *(Bringmann and Ono) Let t be a positive odd integer, and let q be a prime such that $6t$ is not divisible by q. If j is a positive integer, then there are infinitely many non-nested arithmetic progressions $An+B$ such that for every $0 \leq r < t$ we have*

$$(\forall n)[N(r,t;An+B) \equiv 0 \pmod{q^j}]$$

8 Theorems from Theoretical Computer Science

The following chapters in the book are about theorems that are from theoretical computer science. I may have missed a few since the line between *theoretical computer science* and *combinatorics* (and sometimes other fields of math) is not that sharp.

1. Explicit expander constructions using the zig-zag product (2002).
2. A finitely presented group with an NP-complete word problem (2002).

3. Finitely generated groups with a word problem in NP (2002).

4. A polynomial time algorithm for primality testing (2004).

5. The NP-hardness of the 1.36. . .-approximation to the minimum vertex cover (2005).

6. The Cayley Graphs of $SL_2(F_p)$ form a family of expanders (2008).

7. A linear time algorithm for the edge-deletion problem (2008).

8. Majority votes are the most stable (2010).

9 Opinion

Every chapter gives just enough definitions and contexts to *understand* the theorem presented. Hence from this book you can learn a lot about many different areas of mathematics as well as the new theorems in those areas.

This is a great book! You can learn a lot *about* mathematics. You are best off reading it slowly and carefully and taking notes on what you read, putting it in your own words, and doing your own examples (the five summaries I gave were from my efforts). The effort is worth it.

If the reader is wondering *how come Theorem X is not in the book?* then check the website given above. It might be in the sequel.