DOWNLOAD PDF THE MATHEMATICAL PROGRAMMER

Chapter 1 : Stevey's Blog Rants: Math For Programmers

Functional programming gives us back that inalienable right to analyze things by using mathematics. Never again need we bear the burden of that foul mutant x = x+1! No novice programmerâ€"nay, not even a mathematician!â€"could comprehend such flabbergastery.

Share on Facebook Computer programming touches almost every aspect of our lives. Software applications for our computers is commonly thought of when computer programming is mentioned. However, programming of embedded devices can be found in cars, cell phones, video games, appliances and door locks. Computer applications are available for education, entertainment and work that use different types of mathematics. Math is a fundamental activity in computer programming. Basic Programming Math Binary math is at the core of how any computer operates. Binary is used to represent each number in the computer. Reading and simple mathematical operations with binary is critical for low-level programming of hardware. Understanding how to work with hexadecimal number system is required for many programming functions such as setting the color of an object. Standard arithmetic is used in many functions of programming. Addition, subtraction, multiplication and division is used in almost every program written. Algebra is used to solve simple problems that many computer programmers will encounter. Video of the Day Advanced Programming Math Obtaining a computer science degree requires completing many math classes. These include college algebra, statistics, calculus I and calculus II. These classes are applied in two different ways for computer programming. The most obvious is using the math taught to solve complex equations. The less obvious is the skills learned to master advanced math is similar to the skills required to build complex applications. These skills include logic and following complicated step-by-step processes. Application-Specific Math The application for the program being created will often dictate the specific type of math techniques required. Linear algebra is often used for transformation of matrices. Matrix transformation is found in both 2D and 3D modeling as seen in computer-aided design and photo editing software. Differential equations can be found in software to simulate traffic or health conditions. Statistics is used in many computer programming applications including polling systems, reports and card games.

Chapter 2 : Mathematical Optimization Society

In mathematics, computer science and operations research, mathematical optimization or mathematical programming, alternatively spelled optimisation, is the selection of a best element (with regard to some criterion) from some set of available alternatives.

I came to it accidentally, in college, when I took an elective programming class because it fit my schedule. Doing Riemann sums in Fortran is about as math-oriented an introduction to programming as you can get. And I loved it. That same quarter, I was taking my first Japanese class. I was learning the mechanics of communicating, while at the same time trying to gain enough cultural knowledge to feel at ease. But I knew â€" and everyone knew â€" that programming was like math. So clearly, I was good at Fortran because I was good at math. It started when I graduated, became a software engineer, and discovered that the vast majority of developer jobs only required middle-school math at the most. I had to keep a bit of math handy to do whiteboard interviews, but once I was on the job, my ability to communicate both with computers and with other humans was much more important. I figured, at the time, that my jobs were the exceptions. Surely most programming was way more about math. Even on the job boards. I taught Ruby on Rails , which is a web programming framework; people came because they wanted to learn how to make websites. Because of those motivations, the curriculum had virtually no math. And this is what finally did for it me. The students I saw â€" all adults â€" came from a wide range of backgrounds. People with a math background did fine, of course, but people with a heavy language background often did better. I saw this curious effect again when I started working with high schoolers, with a similar curriculum. Bilingual kids often took to programming more easily than monolingual kids. Programming is not math. Specifically, learning to program is more like learning a new language than it is like doing math problems. And the experience of programming today, in industry, is more about language than it is about math. Why do we still have this idea that math skills indicate programming potential, while language skills mean you should go into poli sci? So I looked for relevant academic research. I found a lot of opinions, both from computer science educators , and from people in industry. It seems more likely, though, that this research exists, but not under the search terms I tried. Please let me know if you are aware of relevant papers. Here are some things people often say when asserting that people must be good at math to be good developers. Generally, they fall into three categories: You need to know mathâ€¦ 1A. Academically speaking, most computer science departments trace their lineage to the mathematics department. Many computer science degrees are still very math-heavy. However, as many other people have noted, computer science is not programming. At most academic CS schools, the explicit intent is that students learn programming as a byproduct of learning CS. Programming itself is seen as rather pedestrian, a sort of exercise left to the reader. For actual developer jobs, by contrast, the two main skills you need these days are programming and communication. So while CS still does have strong ties to math, the ties between CS and programming are more tenuous. A common variation on this: In fact, they do it so often that the college-dropout-turned-genius-programmer is our primary Silicon Valley archetype of success. And monetarily, their strategy seems be to working out for them, if the fleets of Teslas on are any indication. As an example, consider whiteboard interview staple big-O notation. If you took a dynamic methods class in school, you know that big-O notation is pretty much meaningless in the real world. The only thing that matters is how it operates on your set of data. There are some interesting mathematical ways to model this, but weirdly enough, the people with CS degrees conducting whiteboard interviews never seem to be too interested. You need to learn mathâ€¦ 2A. Abstract thinking is absolutely a skill that every developer needs to hone. In fact, some people say that finding the right level of abstraction for a concept in your code is the root of all hard problems in software. It is also quite true that you can learn abstract thinking by studying math. Learning a new human language is another way to develop that skill. Coming to understand concepts that are literally impossible to express in your native language is pretty damn abstract. When we learn a second language, the way we re-organize concepts based on a higher level of abstraction is structurally quite similar to how we re-organize concepts when we learn to think mathematically. So while math is one way to learn to think about

abstraction, it is not the only way. Indeed, programming is often concerned with logic. But the same logical concepts are embedded in our human languages. Math is just a formalization of the concepts we use every day when we construct sentences and communicate with other humans. The best writers and speakers understand that, and can construct logical statements in human language that are easy for other people to evaluate. Mathematical logic is a notation for concepts we already know, and it is the concepts â€" rather than the notations â€" that are important. But some jobs still do! My developer job uses plenty of math! Of course people still use programming to do math. But programming itself no longer is math. Even in non-math-oriented languages, there are some applications where math will be useful. And then of course there are whole languages oriented around math, such as Fortran, as I mentioned before, and Haskell , a largely academic language now finding favor among some industry developers. But these are the exceptions that prove the rule. The best developers today work on teams, and they do well IFF they know how to communicate â€" via their code, and directly with other people. When programming was just getting started, early in the last century, we used it to solve highly mathematical problems like calculating missile trajectories and decrypting secret messages. At that point, you had to be good at math to even approach programming. Tools, such as programming languages, were designed specifically to solve mathematical problems, because those were the ones we thought it was worth spending money on. Computers were for doing math. Over time, due to lots of different factors, our societal conception of what computers are for has evolved. The shift happened pretty quickly â€" it started gathering steam in earnest in the 90s. And today â€" only 20 years later â€" we find ourselves with math-oriented programming jobs firmly in the minority. So of course they think the path to success looks like theirs. But the rapid evolution of our industry means that math skills are no longer the only indicator of potential developer skill. So please, guidance counselors: July 15th, Category:

Chapter 3 : Math For Programmers | Pluralsight

Mathematical Programming is, therefore, the use of mathematics to assist in these activities. Mathematical Programming is one of a number of OR techniques. Its particular characteristic is that the best solution to a model is found automatically by optimization software.

This may be the dumbest question I have ever posted online. How much math does one actually need to be a good programmer? Math and programming have a somewhat misunderstood relationship. Many people think that you have to be good at math or made good grades in math class before you can even begin to learn programming. But how much math does a person need to know in order to program? Not that much actually. This article will go into detail about the kinds of math you should know for programming. You probably know it already. For general programming, you should know the following: Addition, subtraction, division, and multiplication â€" And really, the computer will be doing the adding, subtracting, dividing, and multiplying for you anyway. You just have to know when you need to do these operations. So 23 divided by 7 is 3 with a remainder of 2. But 23 mod 7 is 2. If the result is 0, the number is even. If the result is 1, the number is odd. If x mod 2 is 0, you know that whatever number is stored in the variable x is even. To get a percentage of a number, multiply that number by the percent number with the decimal point in front of it. This is why 1. Know what negative numbers are. A negative number times a negative number is a positive. A negative times a positive is negative. Know what a Cartesian coordinate system is. In programming, the 0, 0 origin is the top left corner of the screen or window, and the Y axis increases going down. Know the Pythagorean theorem , and that it can be used to find the distance between two points on a Cartesian coordinate system. Know what decimal, binary, and hexadecimal numbering systems are. Computers work with binary data, which is a number system with only two digits: The numbers are still the exact same, but they are written out differently because there are a different number of digits in each system. Because hex has 6 more digits than the numerals can provide, we use the letters A through F for the digits above 9. The easiest way to show these number systems is with an odometer. The following three odometers always show the same number, but they are written out differently in different number systems: See the Odometer Number Systems page in a new window. Every programming language has functions that can do this for you. On a side note, hexadecimal is used because one hexadecimal digit can represent exactly four binary digits. So since 3 in hex represents in binary and A in hex represents This has the nice effect that the hex number 3A which is 58 in decimal is written in binary as Hex is used in programming because it is a shorthand for binary. Nobody likes writing out all those ones and zeros. Other than the number system stuff, you probably already knew all the math you needed to know to do programming. You would need to know math in order to write programs that do, say, earthquake simulators. Other encryption ciphers are mostly moving data around in specific steps. All the steps are basically substituting numbers for other numbers, shifting rows of numbers over, mixing up columns of numbers, and doing basic addition with numbers. If you just want to write a program that encrypts data, there are software libraries that implement encryption and decryption functions already. Just learn to use the libraries. This basically means, how to take some real-world calculation or some data processing, and write out code that makes the computer do it. For example, in the game Dungeons and Dragons the characters and monsters have several different statistics for combat: HP, or hit points, is the amount of damage a person can take before dying. More HP means you can take more damage before dying. AC, or armor class, is a measure of the chance your armor has of blocking an attack. The lower the AC, the more protective the armor is. This means the damage is the amount from rolling 1 six-sided dice, and then adding 2 to it. A damage stat of 2d4 would be rolling 2 four-sided dice and adding them together. Dungeons and Dragons uses 4, 6, 8, 10, 12, and sided dice. To see if an attacker hits a defender, the attacker rolls a twenty-sided die. Otherwise, the defender has either dodged or blocked the attack and takes no damage. But Bob is more likely to make a successful hit remember, lower THAC0 is better and does more damage. So would you bet on Alice or Bob to win in a fight? Even if you knew a lot of statistics, doing all these calculations would be a pain. Bob 12 hp hits Alice 14 hp for 6 points of damage. Alice is reduced to 8 hp. Alice 8 hp hits Bob 12 hp for 5 points of damage. Bob

is reduced to 7 hp. Bob 7 hp hits Alice 8 hp for 2 points of damage. Alice is reduced to 6 hp. Alice 6 hp hits Bob 7 hp for 6 points of damage. Bob is reduced to 1 hp. Alice 6 hp hits Bob 1 hp for 1 points of damage. Bob is reduced to 0 hp. Alice won 1 Bob won 0 0. But maybe Alice just got lucky in this one fight. Alice won Bob won So we can see that with the given stats, Bob is at a slight advantage. The computer just ran 30, simulated fights. If we were to play 30, fights of Dungeons and Dragons with pencil, paper, and physical dice, it would take months to calculate this. But my laptop had the results in less than 8 seconds. Who would win then? We see that those 6 extra hit points turns the tables and gives Alice the advantage. How about if her hit points were only increased to 16 instead of 20? We see that just tweaking the stats by 2 hit points is just enough to even out the advantages that Bob gets from his higher level of damage. And when you look at this program, the only math it uses is addition, subtraction, and multiplication and division to find a percentage. Sure, go ahead and learn more math. It can only help you become a better programmer. But how much math do you need to know to program?

Chapter 4 : An Introduction to Computer Programming and Mathematics : blog.quintoapp.com

the mathematical discipline devoted to the theory and methods of finding the maxima and minima of functions on sets defined by linear and nonlinear constraints (equalities and inequalities). Mathematical programming is a branch of operations research, encompassing a wide class of control problems.

Multi-objective optimization Adding more than one objective to an optimization problem adds complexity. For example, to optimize a structural design, one would desire a design that is both light and rigid. When two objectives conflict, a trade-off must be created. There may be one lightest design, one stiffest design, and an infinite number of designs that are some compromise of weight and rigidity. The set of trade-off designs that cannot be improved upon according to one criterion without hurting another criterion is known as the Pareto set. The curve created plotting weight against stiffness of the best designs is known as the Pareto frontier. A design is judged to be "Pareto optimal" equivalently, "Pareto efficient" or in the Pareto set if it is not dominated by any other design: If it is worse than another design in some respects and no better in any respect, then it is dominated and is not Pareto optimal. The choice among "Pareto optimal" solutions to determine the "favorite solution" is delegated to the decision maker. In other words, defining the problem as multi-objective optimization signals that some information is missing: In some cases, the missing information can be derived by interactive sessions with the decision maker. Multi-objective optimization problems have been generalized further into vector optimization problems where the partial ordering is no longer given by the Pareto ordering. Multi-modal optimization[edit] Optimization problems are often multi-modal; that is, they possess multiple good solutions. They could all be globally good same cost function value or there could be a mix of globally good and locally good solutions. Obtaining all or at least some of the multiple solutions is the goal of a multi-modal optimizer. Classical optimization techniques due to their iterative approach do not perform satisfactorily when they are used to obtain multiple solutions, since it is not guaranteed that different solutions will be obtained even with different starting points in multiple runs of the algorithm. Evolutionary algorithms , however, are a very popular approach to obtain multiple solutions in a multi-modal optimization task. Classification of critical points and extrema[edit] Feasibility problem[edit] The satisfiability problem , also called the feasibility problem, is just the problem of finding any feasible solution at all without regard to objective value. This can be regarded as the special case of mathematical optimization where the objective value is the same for every solution, and thus any solution is optimal. Many optimization algorithms need to start from a feasible point. One way to obtain such a point is to relax the feasibility conditions using a slack variable ; with enough slack, any starting point is feasible. Then, minimize that slack variable until slack is null or negative. Existence[edit] The extreme value theorem of Karl Weierstrass states that a continuous real-valued function on a compact set attains its maximum and minimum value. More generally, a lower semi-continuous function on a compact set attains its minimum; an upper semi-continuous function on a compact set attains its maximum. More generally, they may be found at critical points , where the first derivative or gradient of the objective function is zero or is undefined, or on the boundary of the choice set. Optima of equality-constrained problems can be found by the Lagrange multiplier method. Sufficient conditions for optimality[edit] While the first derivative test identifies points that might be extrema, this test does not distinguish a point that is a minimum from one that is a maximum or one that is neither. When the objective function is twice differentiable, these cases can be distinguished by checking the second derivative or the matrix of second derivatives called the Hessian matrix in unconstrained problems, or the matrix of second derivatives of the objective function and the constraints called the bordered Hessian in constrained problems. If a candidate solution satisfies the first-order conditions, then satisfaction of the second-order conditions as well is sufficient to establish at least local optimality. Sensitivity and continuity of optima[edit] The envelope theorem describes how the value of an optimal solution changes when an underlying parameter changes. The process of computing this change is called comparative statics. The maximum theorem of Claude Berge describes the continuity of an optimal solution as a function of underlying parameters. Calculus of optimization[edit] See also: More generally, a zero subgradient certifies that a local minimum has been

found for minimization problems with convex functions and other locally Lipschitz functions. Further, critical points can be classified using the definiteness of the Hessian matrix: If the Hessian is positive definite at a critical point, then the point is a local minimum; if the Hessian matrix is negative definite, then the point is a local maximum; finally, if indefinite, then the point is some kind of saddle point. Constrained problems can often be transformed into unconstrained problems with the help of Lagrange multipliers. Lagrangian relaxation can also provide approximate solutions to difficult constrained problems. When the objective function is convex , then any local minimum will also be a global minimum. There exist efficient numerical techniques for minimizing convex functions, such as interior-point methods. Computational optimization techniques[edit] To solve problems, researchers may use algorithms that terminate in a finite number of steps, or iterative methods that converge to a solution on some specified class of problems , or heuristics that may provide approximate solutions to some problems although their iterates need not converge.

Chapter 5 : Intro to Math in Game Development & Programming

For programming, it is important to know about mathematics- especially those branches pertaining to, for example, algorithm performance, but the simple fact is that there is no branch of mathematics that will tell you that Singletons are a horrifically bad idea, for example, or when to favour inheritance over composition, or whether or not you.

You can find me at my blog, TechieSimon. Now, the aim of this course is to give you a solid understanding of how your computer, and, therefore, your code, manipulates and stores numerical and logical data, and to make sure you have sufficient background in maths to write good and correct code, wherever mathematical or logical operations are concerned. Types of Data This module is the first of the two background modules. When we are programming, we use data of lots of different types, characters, strings, Booleans, numbers, and then to make things more complicated, we compose these basic types into structs and classes too. However to a computer, at the most fundamental level, there is only one data type, the set of bits in memory. Numbers of values will ask how many unique values can be stored in a given bit of memory. The answer will naturally lead us to revise the mathematical concept of powers. And mapping high-level data types will finish the module by working through the data types available in C to see how all data ultimately becomes a Boolean, an integer, or a floating points number. This is by the way a fairly elementary module, so if you do already understand how data is represented as bits, which a lot of you will do, you may prefer to just skip ahead to the next module on binary arithmetic. Working in Binary Working in Binary. This affects almost every aspect of manipulating data and means that understanding binary arithmetic is an essential prerequisite to being able to deal effectively at a low level with numbers, and indeed with almost any data stored in a computer. Specifically this module will cover number bases. Binary arithmetic, this part of the course aims to familiarize you with the binary form of numbers that computers use, and also with hexadecimal and octal. Special numbers will compare how different bases impact which numbers are round and therefore considered special. And finally converting between bases, the aim of this part of the course is to go over how to convert between the different number bases. This module is about how integers are represented on your computer, and the consequences for you on how you should use them. Floating Point Numbers Floating Point Numbers This module covers floating point numbers and aims to give you the background you need to write code to carry out problem-free floating point operations. As far as your computer is concerned, they constitute a completely different type of number. That all significantly changes how you should use floating point numbers. This requires us to delve a bit into logic theory. Logic theory is a very deep topic, but we only need to skim the surface to get what we need for most programming with Booleans. The really interesting application is for bitwise operations. Errors and Accuracy Errors and Accuracy This final module is about being aware of how big the errors that can build up in your calculations can be.

Chapter 6 : Mathematical Programming Society

Mathematical Programming publishes original articles dealing with every aspect of mathematical optimization; that is, everything of direct or indirect use concerning the problem of optimizing a function of many variables, often subject to a set of constraints. This involves theoretical and.

Early life[edit] Ada, aged four Byron expected his baby to be a "glorious boy" and was disappointed when his wife gave birth to a girl. This set of events made Ada famous in Victorian society. Byron did not have a relationship with his daughter, and never saw her again. He died in when she was eight years old. Her mother was the only significant parental figure in her life. Annabella did not have a close relationship with the young Ada and often left her in the care of her own mother Judith, Hon. Lady Milbanke, who doted on her grandchild. However, because of societal attitudes of the timeâ€"which favoured the husband in any separation, with the welfare of any child acting as mitigationâ€"Annabella had to present herself as a loving mother to the rest of society. Ada dubbed these observers the "Furies" and later complained they exaggerated and invented stories about her. At the age of eight, she experienced headaches that obscured her vision. She was subjected to continuous bed rest for nearly a year, something which may have extended her period of disability. By , she was able to walk with crutches. Despite being ill, Ada developed her mathematical and technological skills. At age 12 this future "Lady Fairy", as Charles Babbage affectionately called her, decided she wanted to fly. Ada went about the project methodically, thoughtfully, with imagination and passion. Her first step, in February , was to construct wings. She investigated different material and sizes. She considered various materials for the wings: She examined the anatomy of birds to determine the right proportion between the wings and the body. She decided to write a book, Flyology, illustrating, with plates, some of her findings. She decided what equipment she would need; for example, a compass, to "cut across the country by the most direct road", so that she could surmount mountains, rivers, and valleys. Her final step was to integrate steam with the "art of flying". Annabella and her friends covered the incident up to prevent a public scandal. Allegra died in at the age of five. She had a strong respect and affection for Somerville, [24] and they corresponded for many years. She was presented at Court at the age of seventeen "and became a popular belle of the season" in part because of her "brilliant mind. This first impression was not to last, and they later became friends. The Manor had been built as a hunting lodge in and was improved by King in preparation for their honeymoon. It later became their summer retreat and was further improved during this time. Immediately after the birth of Annabella, Lady King experienced "a tedious and suffering illness, which took months to cure. When it became clear that Carpenter was trying to start an affair, Ada cut it off. In fact, you merely confirm what I have for years and years felt scarcely a doubt about, but should have considered it most improper in me to hint to you that I in any way suspected. This went disastrously wrong, leaving her thousands of pounds in debt to the syndicate, forcing her to admit it all to her husband. John Crosse destroyed most of their correspondence after her death as part of a legal agreement. She bequeathed him the only heirlooms her father had personally left to her. She was privately schooled in mathematics and science by William Frend , William King , [a] and Mary Somerville , the noted researcher and scientific author of the 19th century. One of her later tutors was the mathematician and logician Augustus De Morgan. From , when she was seventeen, her mathematical abilities began to emerge, [25] and her interest in mathematics dominated the majority of her adult life. While studying differential calculus , she wrote to De Morgan: I may remark that the curious transformations many formulae can undergo, the unsuspected and to a beginner apparently impossible identity of forms exceedingly dissimilar at first sight, is I think one of the chief difficulties in the early part of mathematical studies. She valued metaphysics as much as mathematics, viewing both as tools for exploring "the unseen worlds around us". What she told him is unknown. Mary Magdalene in Hucknall, Nottinghamshire. A memorial plaque in Latin to her and her father is in the chapel attached to Horsley Towers. Work[edit] Throughout her life, Lovelace was strongly interested in scientific developments and fads of the day, including phrenology [50] and mesmerism. In she commented to a friend Woronzow Greig about her desire to create a mathematical model for how the brain gives rise to thoughts and nerves to feelings "a calculus of the nervous system". As

part of her research into this project, she visited the electrical engineer Andrew Crosse in to learn how to carry out electrical experiments. Later that month Babbage invited Lovelace to see the prototype for his Difference Engine. He called her "The Enchantress of Number". Forget this world and all its troubles and if possible its multitudinous Charlatansâ€”every thing in short but the Enchantress of Number. With the article, she appended a set of notes. She wrote that "The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths. This was the first that she knew he was leaving it unsigned, and she wrote back refusing to withdraw the paper. The historian Benjamin Woolley theorised that: On 12 August , when she was dying of cancer, Lovelace wrote to him asking him to be her executor, though this letter did not give him the necessary legal authority. She then augmented the paper with notes, which were added to the translation. Ada Lovelace spent the better part of a year doing this, assisted with input from Babbage. In note G, she describes an algorithm for the Analytical Engine to compute Bernoulli numbers. It is considered the first published algorithm ever specifically tailored for implementation on a computer, and Ada Lovelace has often been cited as the first computer programmer for this reason. A Symposium on Digital Computing Machines. In her notes, she wrote: Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent. Ada saw something that Babbage in some sense failed to see. What Lovelace sawâ€”what Ada Byron sawâ€”was that number could represent entities other than quantity. So once you had a machine for manipulating numbers, if those numbers represented other things, letters, musical notes, then the machine could manipulate symbols of which number was one instance, according to rules. It is this fundamental transition from a machine which is a number cruncher to a machine for manipulating symbols according to rules that is the fundamental transition from calculation to computationâ€”to general-purpose computationâ€”and looking back from the present high ground of modern computing, if we are looking and sifting history for that transition, then that transition was made explicitly by Ada in that paper. Bromley , in the article Difference and Analytical Engines: All but one of the programs cited in her notes had been prepared by Babbage from three to seven years earlier. Not only is there no evidence that Ada ever prepared a program for the Analytical Engine, but her correspondence with Babbage shows that she did not have the knowledge to do so. She was a mathematical genius She made an influential contribution to the analytical engine She was the first computer programmer She was a prophet of the computer age According to him, only the fourth claim had "any substance at all". But he agrees that Ada was the only person to see the potential of the analytical engine as a machine capable of expressing entities other than quantities. The comic features extensive footnotes on the history of Ada Lovelace, and many lines of dialogue are drawn from actual correspondence. Events have included Wikipedia edit-a-thons with the aim of improving the representation of women on Wikipedia in terms of articles and editors to reduce unintended gender bias on Wikipedia. The Ada Initiative was a non-profit organisation dedicated to increasing the involvement of women in the free culture and open source movements. Ada Lovelace House is a council-owned building in Kirkby-in-Ashfield , Nottinghamshire, near where Lovelace spent her infancy; the building was once an internet centre [] She is also the inspiration and influence for the Ada Developers Academy in Seattle, Washington. The academy is a non-profit that seeks to increase diversity in tech by training women, trans and non-binary people to be software engineers.

Chapter 7 : Ada Lovelace - Wikipedia

Basic Programming Math. Binary math is at the core of how any computer operates. Binary is used to represent each number in the computer. Reading and simple mathematical operations with binary is critical for low-level programming of hardware.

Do you really want to get better at mathematics? Remember when you first learned how to program? I spent two years experimenting with Java programs on my own in high school. Those two years collectively contain the worst and most embarrassing code I have ever written. My programs absolutely reeked of programming no-nos. Hundred-line functions and even thousand-line classes, magic numbers, unreachable blocks of code, ridiculous code comments, a complete disregard for sensible object orientation, negligence of nearly all logic, and type-coercion that would make your skin crawl. I committed every naive mistake in the book, and for all my obvious shortcomings I considered myself a hot-shot programmer! At least I was learning a lot, and I was a hot-shot programmer in a crowd of high-school students interested in game programming. This naturally involved a depth-first search and a couple of recursive function calls, and once I had something I was pleased with, I compiled it and ran it on my first non-trivial example. Low and behold even having followed test-driven development! It took hundreds of test cases and more than twenty hours of confusion before I found the error: I was passing a reference when I should have been passing a pointer. This was not a bug in syntax or semantics I understood pointers and references well enough but a design error. And the aggravating part, as most programmers know, was that the fix required the change of about 4 characters. Twenty hours of work for four characters! Once I begrudgingly verified it worked of course it worked, it was so obvious in hindsight , I promptly took the rest of the day off to play Starcraft. Of course, as every code-savvy reader will agree, all of this drama is part of the process of becoming and strong programmer. One must study the topics incrementally, make plentiful mistakes and learn from them, and spend uncountably many hours in a state of stuporous befuddlement before one can be considered an experienced coder. And should you forget one, a crafty use of awk and sed will suffice. We placed Cal Poly third in the Southern California Regionals, and in my opinion our success was due in large part to the dynamics of our team. I center, in blue have since gotten a more stylish haircut. I just want to use applications that others have written, like Chrome and iTunes. This person belongs in some other profession. Perhaps more ironically, all of my real work is done with paper and pencil. Unfortunately this sentiment is mirrored among most programmers who claim to be interested in mathematics. Mathematics is fascinating and useful and doing it makes you smarter and better at problem solving. The appropriate translation of the above quote for mathematics is: The point is that the sentiment is in the wrong place. Mathematics is cousin to programming in terms of the learning curve, obscure culture, and the amount of time one spends confused. Honestly, it sounds ridiculously obvious to say it directly like this, but the fact remains that people feel like they can understand the content of mathematics without being able to write or read proofs. I want to devote the rest of this post to exploring some of the reasons why this misconception exists. My main argument is that the reasons have to do more with the culture of mathematics than the actual difficulty of the subject. I honestly do believe that the struggle and confusion builds mathematical character, just as the arduous bug-hunt builds programming character. If you want to be good at mathematics, there is no other way. And I want to stress that this is not a call for all programmers to learn mathematics. I just happen to notice that, for good reason, the proportion of programmers who are interested in mathematics is larger than in most professions. And as a member of both communities, I want to shed light on why mathematics can be difficult for an otherwise smart and motivated software engineer. So read on, and welcome to the community. Travelling far and wide Perhaps one of the most prominent objections to devoting a lot of time to mathematics is that it can be years before you ever apply mathematics to writing programs. On one hand, this is an extremely valid concern. If you love writing programs and designing software, then mathematics is nothing more than a tool to help you write better programs. Indeed, I provide an extended example of this in my journalesque post on introducing graph theory to high school students: Only then can we see its beauty and wide applicability. Here is a more concrete example. But it took hundreds of years of

number theory to get there, and countless deviations into other fields and dead-ends. Of course there are other examples much closer to contemporary fashionable programming techniques. One such example is boosting. While we have yet to investigate boosting on this blog [update: In a field dominated by practical applications, this result is purely the product of mathematical analysis. And of course boosting in turn relies on the mathematics of probability theory, which in turn relies on set theory and measure theory, which in turn relies on real analysis, and so on. One could get lost for a lifetime in this mathematical landscape! And indeed, the best way to get a good view of it all is to start at the bottom. To learn mathematics from scratch. What is it really, that people have such a hard time learning? Most of the complaints about mathematics come understandably from notation and abstraction. While methods of proof are semantical by nature, in practice they form a scaffolding for all of mathematics, and as such one could better characterize them as syntactical. These are the loops, if statements, pointers, and structs of rigorous argument, and there is simply no way to understand the mathematics without a native fluency in this language. So much of mathematics is built up by chaining together a multitude of absolutely trivial statements which are amendable to proof by the basic four. Of course, there are many sophisticated proofs in mathematics, but an overwhelming majority of very important facts fall in the trivial category. There are certainly works of writing that require a lot more than what it takes to write a shopping list. And as you probably know, there are many many more methods of proof than just the basic four. Proof by construction, by exhaustion, case analysis, and even picture proofs have a place in all fields of mathematics. There are many books dedicated to showcasing such techniques, and rightly so. Clever proofs are what mathematicians strive for above all else, and once a clever proof is discovered, the immediate first step is to try to turn it into a general method for proving other facts. Fully flushing out such a process over many years, showcasing many applications and extensions is what makes one a world-class mathematician. Another difficulty faced by programmers new to mathematics is the inability to check your proof absolutely. With a program, you can always write test cases and run them to ensure they all pass. If your tests are solid and plentiful, the computer will catch your mistakes and you can go fix them. The only way to get feedback is to seek out other people who do mathematics and ask their opinion. It can be really. Mathematical syntax Another major reason programmers are unwilling to give mathematics an honest effort is the culture of mathematical syntax: Let me start with an example of why this is not a problem in programming. The other extreme is the syntax of mathematics. The daunting fact is that there is no bound to what mathematical notation can represent, and much of mathematical notation is inherently ad hoc. Just to give the unmathematical reader a taste: The use of the letter delta could signify a slightly nonstandard way to write the Kronecker delta function , for which is one precisely when and zero otherwise. For example, here is a common difficulty that beginners face in reading math that involves use of the summation operator. The most rigorous way to express this is not far off from programming: Let be a finite set of things. Then their sum is finite: This is really just a left fold of the plus operator over the list. For instance, I could say Let is finite. Things are now more vague. Moreover, the order in which the things are summed which for a left fold is strictly prescribed is arbitrary. In the case of the capital Sigma, there is nothing syntactically stopping a mathematician from writing Though experienced readers may chuckle, they will have no trouble understanding what is meant here. That is, syntactically this expression is unambiguous enough to avoid an outcry: This often shows up in computer science literature, as is a standard letter to denote an alphabet such as the binary alphabet. But programmers must realize: A mathematician would be just as bewildered and confused upon seeing some of the pointer arithmetic hacks C programmers invent, or the always awkward infinite for loop, if they had not had enough experience dealing with the syntax of standard for loops. The upshot of this whole conversation is that the reader of a mathematical proof must hold in mind a vastly larger body of absorbed and often frivolous knowledge than the reader of a computer program. For better or worse, mathematical syntax is just a means to that end, and the more abstract the mathematics becomes, the more flexiblility mathematicians need to keep themselves afloat in a tumultuous sea of notation. The reason for this is once again cultural. There are two parts to understanding how these functions work. The first part is that someone or a code comment explains to you in a high level what they do to an input. The second part is to weed out the finer details. In mathematics there is no unified documentation, just a collective understanding,

scattered references, and spoken folk lore. You are expected to derive the finer details and catch the errors yourself. Is it because they have a hard time getting honest help from rudely abrupt moderators on help websites like stackoverflow? Is it because often when one wants to learn the basics, they are overloaded with the entirety of the documentation and the overwhelming resources of the internet and all its inhabitants?

Chapter 8 : Why there is no Hitchhikerâ€™s Guide to Mathematics for Programmers â€“ Math âˆ© Prograr

Math For Programmers In ProgrammerMathSkills, it is lamented that many programmers lack training (or have forgotten all about it as part of the usual MindWipe that occurs shortly after graduation).

Let me tell you about it. I hear that so often; I hardly know anyone who disagrees. And you know what? You can be a good, solid, professional programmer without knowing much math. Math is a lot easier to pick up after you know how to program. They teach math all wrong in school. Knowing even a little of the right kinds of math can enable you do write some pretty interesting programs that would otherwise be too hard. In other words, math is something you can pick up a little at a time, whenever you have free time. Nobody knows all of math, not even the best mathematicians. The field is constantly expanding, as people invent new formalisms to solve their own problems. You can pick the one you like best. And a basic understanding of Cartesian geometry, too. Those are useful, and you can learn everything you need to know in a few months, give or take. But the rest of them? I think an introduction to the basics might be useful, but spending a whole semester or year on them seems ridiculous. No need to dive right into memorizing geometric proofs and trigonometric identities. For one thing, most of the math you learn in grade school and high school is continuous: Your background as a programmer will help keep you focused on the practical side of things. The math we use for modeling computational problems is, by and large, math on discrete integers. This is a generalization. For programmers, the most useful branch of discrete math is probability theory. How many ways are there to make a Full House in poker? Or a Royal Flush? Whenever you think of a question that starts with "how many ways And as it happens what are the odds? It starts with flipping a coin and goes from there. I still have my discrete math textbook from college. I had to figure out what was important on my own, later, the hard way. A pretty important one, too, but hopefully it needs no introduction. Algebra and Linear Algebra i. They should teach Linear Algebra immediately after algebra. I have a really cool totally unreadable book on the subject by Stephen Kleene, the inventor of the Kleene closure and, as far as I know, Kleenex. If anyone has a recommendation for a better introduction to this field, please post a comment. Information Theory and Kolmogorov Complexity. I bet none of your high schools taught either of those. Information theory is veeery roughly about data compression, and Kolmogorov Complexity is also roughly about algorithmic complexity. There are others, of course, and some of the fields overlap. But it just goes to show: Everyone teaches it, so it must be important, right? Well, calculus is actually pretty easy. Before I learned it, it sounded like one of the hardest things in the universe, right up there with quantum mechanics. Quantum mechanics is still beyond me, but calculus is nothing. After I realized programmers can learn math quickly, I picked up my Calculus textbook and got through the entire thing in about a month, reading for an hour an evening. Calculus is all about continuums â€” rates of change, areas under curves, volumes of solids. Geometry, trigonometry, differentiation, integration, conic sections, differential equations, and their multidimensional and multivariate versions â€” these all have important applications. To put this in perspective, think about long division. Raise your hand if you can do long division on paper, right now. Why do they even teach it to you? And besides, if your life were on the line, you know you could perform long division of any arbitrarily large numbers. How would you do it? If pressed, you could figure out a way to continue using repeated subtraction to estimate the remainder as decimal number in this case, 0. You could figure it out because you know that division is just repeated subtraction. The intuitive notion of division is deeply ingrained now. The right way to learn math is to ignore the actual algorithms and proofs, for the most part, and to start by learning a little bit about all the techniques: Think of it as a Liberal Arts degree in mathematics. Because the first step to applying mathematics is problem identification. If you have a problem to solve, and you have no idea where to start, it could take you a long time to figure it out. There are lots and lots of mathematical techniques and entire sub-disciplines out there now. In school they teach you the Chain Rule, and you can memorize the formula and apply it on exams, but how many students really know what it "means"? The chain rule is just how to take the derivative of "chained" functions â€” meaning, function x calls function g , and you want the derivative of x g. You should learn how to count, and how to program, before you learn how to take derivatives and perform

integration. I think the best way to start learning math is to spend 15 to 30 minutes a day surfing in Wikipedia. You start with pretty much any article that seems interesting e. String theory , say, or the Fourier transform , or Tensors , anything that strikes your fancy. Do this recursively until you get bored or tired. Doing this will give you amazing perspective on mathematics, after a few months. At least for applied math. Metamathematics is the fascinating study of what the limits are on math itself: One great thing that soon falls by the wayside is notation. Mathematical notation is the biggest turn-off to outsiders. For instance, a summation sign capital-sigma or product sign capital-pi will look scary at first, even if you know the basics. It might be a very fancy calculator such as R, Matlab, Mathematica, or a even C library for support vector machines. But almost all useful math is heavily automatable, so you might as well get some automated servants to help you with it. When Are Exercises Useful? Same with logarithms, roots, transcendentals, and other fundamental mathematical representations that appear nearly everywhere. With that said, I still occasionally do math exercises. If an exercise or even a particular article or chapter is starting to bore you, move on. Jump around as much as you need to. Let your intuition guide you. How Will This Help Me? Well, it might not â€” not right away. And learning it is enabling me to code or use in my own code neural networks, genetic algorithms, bayesian classifiers, clustering algorithms, image matching, and other nifty things that will result in cool applications I can show off to my friends. The notation is actually there to make it easier, but like programming-language syntax notation is always a bit tricky and daunting on first contact. Because I know I can figure it out. I have lots of years left, and lots of books, and articles. What a great idea that turned out to be!

Chapter 9 : Mathematical optimization - Wikipedia

Math is not needed for programming, because programming is math. It can be good math, or bad math (like when orangutan's long call is transliterated into Ook language), but whenever a programmer designs an object model of someting, ze is (usually unknowingly) performing an act of mathematics.

We do this using an if statement. Let us demonstrate by an example - try running this code. We can nestle if statements together in this way, for example: The computer will check each expression expr1, expr2, If it is true then the code inside the brackets will be executed and then no further expressions will be checked. If none of the expressions are true then the code in the last brackets is executed lastCode in the above. Any of the else if or the else parts can be eliminated to also produce valid code. The following example program shows them in use. Write a program that defines a variable x with some initial value, and an if-else statement that prints out whether x is odd or even. Loops Looping is the next important concept in programming. The first type of looping is the while loop. When the while loop is encountered by the computer, the behaviour is as follows: Let us see an example - try running the following program: You will need two while loops. The general form is: And of course, we can change 0 to any other value that we want i to have intially. Let us see an example of a for loop summing the numbers from 0 to 10, and printing out the value: Modify the above program: So that it sums the squares from 0 to Add an integer variable called N with some initial value such as , and change the code so that it sums the squares from 0 to N. Collatz conjecture states that the following process always stops for all initial values of n: Take a whole number n greater than 0. If n is even, then halve it. If n now has value 1, then stop. Otherwise go to step 2. Functions At its simplest, a function is a way of grouping together a collection of instructions so that they can be repeatedly executed. Then somewhere in the program, we "define" the function, i. When we want to execute the instructions in the function, we call it be writing its name as will be seen. We write a function that prints out the numbers from 1 to 9, called countToTen. First, we need to include the following line of code somewhere at the top of the program to define the function: The will be explained shortly as well, and of course the semicolon ; is needed as the end of the instruction. And to define the function, we include the following block of code note that this time, we do not include a semicolon at the end of the first line: When we want to call the function, we write countToTen ;. To declare a function with a set of input variables which can be any of the variable types we have seen so far , we list these types inside the brackets in the declaration, separated by commas we can also include names for these variables, which are helpful for descriptive purposes. We declare the function as follows changing its name: To call the function with value, say, 5, we write "countToN 5 ;". If we have an integer variable called x, we can use the value of x as an input by writing "countToN x ;". For example, if we wanted to return a decimal number, we would replace void with float. Inside the function, when we want to return from the function to where the function was called from, we write "return value;", where value is replaced by the value to be returned such as 5, or x. As described above, we declare the function as int gcd int a, int b ; And for the definition of the function, we use a similar format again this time, without the semicolon at the end of the first line: How and why does the algorithm given in the above function work? To call the function, we write, for example, gcd , This is then treated as an integer, whose value is the value returned - so we can treat it like any other integer. Change the code in the gcd function so that it uses Euclids algorithm, which is computationally more efficient. Common maths functions To make use of the mathematical functions, we write " include " somewhere at the top of the program - this tells the compiler about these functions what their names are, what inputs they take, what outputs they give. Most of the mathematical functions have float as their input and output type. The trignometric functions use radians as their input instead of degrees, and the log function is base e. Random numbers To make use of random numbers, we need to include the library "cstdlib", so write include at the top of the program. To turn this into a floating point number between 0 and 1, called x, we would write: This is necessary since if we divide two integers, then the part after the decimal point is discarded. How would you generate a random integer between 1 and 10?