Algorithms for integer multiplication and division, and floating point arithmetic

Adam Boult (www.bou.lt)

July 11, 2025

Contents

Preface

This is a live document, and is full of gaps, mistakes, typos etc.

Part I

Algorithms for multiplication and division of integers with just addition and subtraction operations

Part II

Simulation of integers outside the register range

Part III

Representation and arithmetic of real numbers

Chapter 1

Representation of real numbers, and addition and subtraction on them

1.1 Introduction

1.1.1 Introduction

Chapter 2

Multiplication and division of real numbers

2.1 Introduction

2.1.1 Introduction

Part IV

Numerical methods for real numbers

Chapter 3

Arithmetic on real numbers

3.1 Representing real numbers

3.1.1 Binary floating point

store as two integers (x, y) evaluate as $x * 2^y$ this is binary floating point this means you get inaccuracies

eg $(0.1 + 0.2 - 0.3) * 10^{20}$ is not zero

3.1.2 Decimal floating point

alternative is decimal floating point store as $x * 10^y$

3.2 Operations on real numbers

3.2.1 Floor and ceiling

3.2.2 Powers, logarithms and exponentials

3.2.3 Overflow and underflow

The need to approximate real operations with pseudo real numbers. If we round small values to 0, then ln 0, x/0 break. This is underflow

Chapter 4

Calculating square roots of real numbers

4.1 Introduction

4.1.1 Introduction

Chapter 5

Calculating π and e

5.1 Calculting π

5.2 Calculating e

Chapter 6

Numerical integration

6.1 Numerical integration

Chapter 7

Trigonometric functions

7.1 Trigonomic functions

Chapter 8

Root-finding algorithms

Part V

Numerical methods for complex numbers

Chapter 9

Constructing the Mandelbrot set

9.1 Introduction

Part VI

Linear algebra

Chapter 10

Identifying roots of linear equations

10.1 Finding roots of linear equations

Chapter 11

Identifying roots of non-linear equations

11.1 Finding roots of non-linear equations

Chapter 12

Discrete linear algebra

12.1 Linear programming with integers

Chapter 13

Linear algebra

13.1 Linear operations

13.1.1 Representing matrices

13.1.2 Vectors and matrices

13.1.3 Addition

13.1.4 Multiplication

13.1.5 Inverse

13.1.6 Transpose

13.1.7 Scalar multiplication

13.1.8 Matrix decomposition

13.1.9 Broadcasting

Loosen standards, can do addition subtraction if one matrix is $1 \times n$.

13.2 Linear programming

Chapter 14

Calculating convex hulls

Chapter 15

Numerical methods for Ordinary Differential Equations (ODEs), including Euler's method

Chapter 16

Numerical methods for Partial Differentiable Equations (PDEs)

Chapter 17

Solving elliptic curves

Part VII

Unconstrained optimisation

Chapter 18

Optimising smooth functions with gradient descent

18.1 Gradient descent

18.1.1 Gradient descent

18.1.2 What is gradient descent?

Rather than solve a normal equation, gradient descent takes the loss function, and takes the derivative of the loss function with respect to each parameter.

Small adjustments are then made to the parameters, in the direction of the steepest derivative, resulting in better parameters.

As derivative term gets smaller, convergance happens. The largest changes to the parametres occurs early on in the algorithm.

Can stop if not lowering by much

18.1.3 Local minima

Gradient descent is not guaratneed to arrive at a global minimum. For some loss functions, there will be multiple local minima, and gradient descent can end up in the wrong one.

Linear regression does not have this issue.

As a result, when we create functions with loss functions, convextity is very important. If the loss space is convex, then we will not get stuck in a local minima.

18.1.4 Momentum gradient descent

Batch gradient descent

:= used to denote an update of variable. Used in programming, eg x=x+1.

$\theta_j := \alpha \frac{\delta}{\delta \theta_j} J(\theta_0, \theta_1)$

α sets rate of descent.

$\theta 0 := \theta 0 - \alpha / m \sum (h0(x) - y)$

$\theta j := \theta j - \alpha / m \sum (h0(x) - y) x j$

Can check if j theta increasing, means bad methodology, lower alpha

Get run for x iterations,evaluate j(theta)

Can use matrices to do each step

Can check convergence by checking cost over last 1000 or so, rather than all

Smaller learning rate can get to better solution, as can circle drain for small samples

Slowly decreasing learning rate can get better solutions

$\alpha = const1/(i + cost2)$

Do gradient descent on all samples

The standard gradient descent algorithm above is also known as batch gradient descent. There are other implementations.

Mini-batch gradient descent

Use b samples on each iteration, b is parameter, between stochastic and batch

$b = 2 - 100$ for example

Stochastic gradient descent

Do gradient descent on one (?!) sample only

Not guaranteed for each step to go towards minimum, but each step much faster

Stochastic gradient descent with momentum

The gradient we use is not just determined by the single sample, it is a moving average of past samples.

Epochs

This refers to the number of times the whole dataset has been run.

18.1.5 Adaptive learning rates (Adagrad, Adadelta, RMSProp, ADAptive Momentum (ADAM))

18.1.6 Adagrad

18.1.7 Adadelta

18.1.8 RMSProp

18.1.9 ADAptive Momentum (ADAM)

18.2 Differentiable on a single axis

18.2.1 Coordinate descent

18.3 Twice-differentiable functions

18.3.1 Algorithmic efficiency

An algorithm takes memory and time to run. Analysing these characteristics of algorithms can enable effective choice of algorithms.

Complexity is described using big-O notation. So an algorithm with parameters θ would have a time efficiency of $O(f(\theta)$ where $f(\theta)$ is a function of θ.

Generally we expect $f(\theta)$ to be weakly increasing for all θ. As we add additional inputs, these would not decrease the time or space requirements of the algorithm.

An algorithm which did not change complexity with inputs would have a constant as the largest term. So we would write $O(c)$.

An algorithm which increase linearly with inputs could be written $O(\theta)$.

An algorithm which increased exponentially could be written $O(e^\theta)$.

Complexity can differ between worst-case scenarios, best-case scenarios and average case scenarios.

We can describe logical systems by completeness (all true statements are theorems) and soundness (all theorems are true). We have similar definitions for algorithms.

An algorithm which returns outputs for all possible inputs is complete. An algorithm which never returns an incorrect output is optimal.

18.3.2 Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm

18.4 Sort

18.4.1 Floating point

18.4.2 Integer

18.4.3 Tree path

18.4.4 Array

Eg 8 queens

18.4.5 Search tree

in search tree, each node has state which is used for test. could be ID of node (for path finding), path history and cost (for trav salesman)

frontier (not open list)

backward search. only possible if end state is clearly defined. eg maze. not clear if don't know eg 8 queens.

can do breadth first on them simultaneously?

problem has: initial state. actions, transition model

model $T(s,a)->s_n+1$. as in , given state and action, we have new state

goal test on each state

path cost for each sucessor

search tree. we expand when testing action.

open lists in unexplored notes.

loopy paths. if we go $a->b$ don't need to go $b->a$ because if goal, not any closer, if util, higher cost.

redunant paths. if we've already been to c, no need to explore going there from somewhere else in goal

if already been to c at lower cost, no point for util

actions is function on state.

keep explored states in open list

Chapter 19

Extensions to gradient descent

Chapter 20

Unconstrained optimisation of discrete functions

Chapter 21

Unconstrained optimisation of non-differentiable real functions

21.1 Non-differentiable functions

21.1.1 Subgradient descent

21.1.2 Hill climbing

We initialise at some point in the parameter space.

We identifify nearby alternative points in parameter space, and move to the one with the most improvement.

Movement only occurs in one parameter at a time.

Part VIII

Constrained optimisation

Chapter 22

Constrained optimisation: Linear, quadtratic and convex programming

22.1 Linear programming

22.2 Quadratic programming

22.3 Convex programming