A New Dispersive Flies Optimisation Algorithm for the Sum of Three Cubes

May 25, 2022

Abstract

By first solving the equation $x^3 + y^3 + z^3 = k$ with fixed k for z and then considering the distance to the nearest integer function of the result, we turn the sum of three cubes problem into an optimisation one. We then present a modification of the dispersive flies optimisation (DFO) algorithm and apply it to this function in the case with $k = 2$. We have two goals: to show the viability of using optimisation when searching for integer solutions and to measure how efficient our modified DFO is. As a comparison we also use two implementations of simulated annealing. The efficiency of the algorithms is measured by their running times. We model the data by assuming two underlying probability distributions – exponential and log-normal, and calculate relevant numerical characteristics for them. Finally, we evaluate the statistical distinguishability of our models with respect to the geodesic distance on the corresponding statistical manifolds.

Keywords: dispersive flies optimisation; particle swarm optimisation; diophantine equations; sum of three cubes; simulated annealing

1 Introduction

The sum of three cubes problem can be stated in the following way: Let k be a positive integer. Is there a solution to the equation

$$x^3 + y^3 + z^3 = k, \tag{1.1}$$

such that $(x, y, z) \in \mathbb{Z}^3$?

As is well known, for $k \equiv 4 \pmod 9$ such a solution does not exist [1]. For $k \not\equiv 4 \pmod 9$ however, it has been conjectured by Heath-Brown that there are infinitely many solutions [2]. A direct search for solutions is one way to support this conjecture. Until recently there were only two numbers below 100 for which a representation as a sum of three integer cubes had not been found. Those were 33 and 42.

Then the sum of three cubes problem gained an unusual amount of fame in the last year after a Numberphile video inspired a solution to the case with $k = 33$ [3]. What followed were a solution for $k = 42$ and a new (third) solution for $k = 3$.

The main problem when directly searching for solutions is the time it takes for a brute force approach. However, there are ways to reduce this time and the latest method by Booker reached a time complexity of $O\left(B^{1+\varepsilon}\right)$, with $\min\{|x|, |y|, |z|\} \leq B$ [3], i.e. an almost linear search.

When thinking about a way to improve this result, a natural step seems to be to gamble a bit and rely on the gods of probability. In other words we may try to guess the solution via

some random search heuristic. One way this can be achieved is by turning the sum of three cubes into an optimisation problem, for which there are a wealth of such heuristics. As it turns out there have been some attempts to solve diophantine equations using particle swarm optimisation (PSO) algorithms [4]. And while we use PSO in some form, our approach is different and permits the use of many other stochastic optimisation methods.

The emerging optimisation problem, however, is highly non-trivial. This leads us to develop a modified version of dispersive flies optimisation (DFO) [5], which falls within the class of swarm optimisation algorithms. This modified DFO (mDFO) can be viewed as the main contribution of the current work.

This paper is organised as follows. In section 2 we define the function to be optimised. Since we've said that our approach was motivated by the desire to use a random search heuristic, in the next sections we apply to our function two such heuristics that look promising, namely mDFO and simulated annealing (SA). Their performance is then evaluated empirically by measuring the time it takes the respective algorithm to obtain a solution to (1.1) for $k = 2$. This choice of k is justified by its high density of solutions, which makes testing much easier. In particular, in section 3 we introduce our mDFO algorithm in detail, while in section 4 we use two versions of SA – one with restarts and one without. In section 5 we conduct a thorough statistical analysis of the performance of our algorithms and of their similarities. We conclude this work with section 6, where we briefly comment on our results and show some possible directions for future research.

2 Our approach

We first start by defining the function, which we will be optimizing. Solving equation (1.1) for z we trivially get

$$z = \left(k - x^3 - y^3\right)^{\frac{1}{3}}. \tag{2.1}$$

We can then define a function $f_k(x, y)$ as

$$f_k(x, y) := \|z\| = \left\| \left(k - x^3 - y^3\right)^{\frac{1}{3}} \right\|, \tag{2.2}$$

where $\|z\|$ denotes the distance to the nearest integer from z.[1]

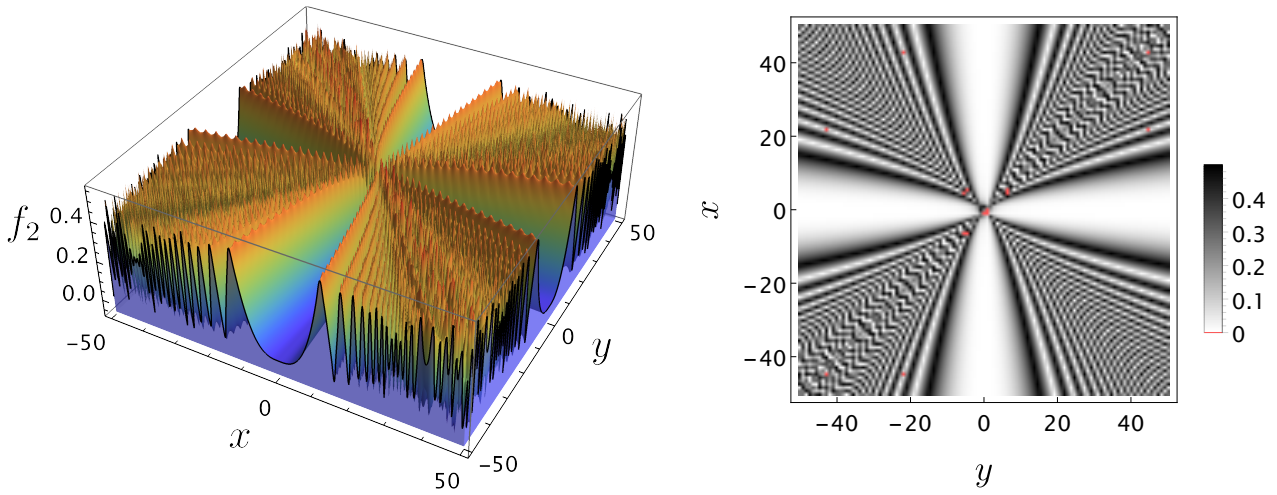
Now let us fix $k = k_0$. If (x_0, y_0, z_0) is an integer solution to Eq. (1.1), we have

$$f_{k_0}(x_0, y_0) = 0 \tag{2.3}$$

and this is a global minimum. Conversely, if $f_{k_0}(x, y)$ has a local minimum at $(x_0, y_0) \in \mathbb{Z}^2$, such that (2.3) is satisfied, it has a global minimum there and this gives the solution (x_0, y_0, z_0) to (1.1), where z_0 is evaluated from Eq. (2.1).

Hence, the problem now is to find a global minimum of $f_k(x, y)$ with $(x, y) \in \mathbb{Z}^2$ and fixed k. As it turns out, this is not an easy problem since our function has no shortage of local optima. As an illustration we show a plot of the function $f_2(x, y)$ with $x, y \in [-50, 50]$ in figure 1.

[1] For a summary of the main properties of the distance to the nearest integer function see `https://www.researchgate.net/publication/308023356_Note_on_the_Distance_to_the_Nearest_Integer`



(a) Many local optima of the function $f_2(x, y)$.

(b) A discrete matrix plot of $f_2(x, y)$.

Figure 1: Sample of the function $f_2(x, y)$ in the range $x, y \in [-50, 50]$. (a) depicts the many local optima of the function $f_2(x, y)$. (b) shows a discrete matrix plot of $f_2(x, y)$ with red pixels corresponding to the global minima, where $f_2(x, y) = 0$.

3 Swarm optimisation

The first algorithm we use to minimise (2.2) is a representative from the family of swarm algorithms. These algorithms try to find extrema of a function by emulating the behaviour of swarms of insects searching for resources. There are many algorithms based on this idea and they may differ significantly in performance, depending on the problem. For the latest development on the subject one may refer to [6].

For our particular function we first tried using standard particle swarm optimisation (SPSO) [7]. However, it turned out to be too slow and, searching for a better alternative, we stumbled upon the DFO algorithm developed by Al-Rifaie [5]. It has the benefit of simplicity and we decided to use it as a base and see where we end up. As it turned out we made some significant changes in order to improve the performace for our particular function.

As the main point we found the breaking of the swarm in DFO to be too limited for our needs, so we opted for simple randomisation of all flies after a certain criterion has been met. This, however, meant that the swarm best would change after every dispersion and may have lead to insufficient exploration around it. To fix this we introduced a memory of the best position of the swarm found so far. We then used this "best swarm best" in the position update formula in addition to the neighbours best and swarm best positions.

This improved the performance of our algorithm when compared with unmodified DFO. However, our function has a lot of local minima and the algorithm tended to get stuck in them often. Furthermore, the closer the best swarm best position's fitness function was to 0, the harder it was to find a better position (including after a dispersion). One way to alleviate this somewhat was to introduce simple restarts after a fixed number of iterations without an improvement in the best swarm best position.

This required including a new parameter in the algorithm however, namely the number of iterations. To avoid it we decided to instead probabilistically change the best swarm best to the current swarm best in the iteration even if the new position was worse. The probability depends on the difference between the fitness function values in both positions.

3.1 Description of the algorithm

Our mDFO algorithm can in principle be used for any discrete optimisation problem and in this subsection we will give a general description, which doesn't refer to specifics such as the dimension of the search space or the explicit form of the fitness function.

Like all PSO algorithms it first starts with the initialisation of the swarm (with s particles). This is done by choosing a random position $\vec{x}_i$ for each particle $i \in \{0, ..., s-1\}$ inside the (discrete) search space and calculating it's fitness function $ff(\vec{x}_i)$. Then the swarm best position $\vec{sb} \in \{\vec{x}_0, ..., \vec{x}_{s-1}\}$ is determined, such that

$$ff(\vec{sb}) = \min_{i \in \{0,...,s-1\}} \{ff(\vec{x}_i)\}. \tag{3.1}$$

As mentioned, the algorithm uses a memory of the best position obtained so far, which we call best swarm best $\vec{bsb}$. At initialisation this is set equal to the swarm best, i.e. $\vec{bsb} \leftarrow \vec{sb}$.

As usual, the particles communicate with their neighbours. We use the standard ring topology for the set of neighbours n_i of particle i [7], i.e.

$$n_i = \{(i-1) \bmod s,\ i,\ (i+1) \bmod s\}. \tag{3.2}$$

Then, knowing the neighbours, the best position among them is found for each particle. We call this the neighbours best $\vec{nb}_i$ and it satisfies the following

$$ff(\vec{nb}_i) = \min_{k \in n_i} \{ff(\vec{x}_k)\}, \quad \vec{nb}_i \in \{\vec{x}_k\}_{k \in n_i}. \tag{3.3}$$

Next follow the iterations. We can limit the number of iterations to get some approximate solution or wait for some condition to be satisfied. An iteration consists of a position update, confinement and a $\vec{sb}$, $\vec{bsb}$ and $\vec{nb}_i$ update.

First is the position update. Since, as we've said, the swarm is periodically dispersed, the position update formula depends on a simple dispersion condition. We first define a dispersion parameter $dp = s/5$. Then the number of particles that have reached the best swarm best position is counted and, if they are no less than dp, the positions of all particles are randomised across the search space.

If the dispersion condition has not been met, the positions of the particles are updated by the formula

$$x_{i,d} \leftarrow \text{round}\left(nb_{i,d} + \frac{r}{2}\left(sb_d + bsb_d - 2x_{i,d}\right)\right), \quad d = 1, ..., D. \tag{3.4}$$

where r is drawn from a uniform distribution on the interval $(0, 1)$ and D is the dimension of the search space.

After the position update there may be some particles outside the search space. We want them confined inside however, so random positions are chosen for such particles. While this is a simple way to implement confinement, it helps with the exploration behaviour that some problems so desperately need.

Finally, $\vec{sb}$, $\vec{bsb}$, and $\vec{nb}_i$ need to be updated. After $\vec{sb}$ is determined as in the initialisation, it is used to update $\vec{bsb}$. If $ff(\vec{sb}) < ff(\vec{bsb})$,

$$\vec{bsb} \leftarrow \vec{sb}, \tag{3.5}$$

as expected. If, however, $ff(\vec{sb}) > ff(\vec{bsb})$, the worse position $\vec{sb}$ is accepted as the new best swarm best with probability

$$p = 1 - \frac{ff(\vec{sb}) - ff(\vec{bsb})}{0.5}. \tag{3.6}$$

As can be seen, the choice of $\vec{bsb}$ borrows it's idea from SA algorithms. In practice the change to a worse $\vec{bsb}$ happens after the particles have dispersed, because this is when $ff(\vec{sb})$ can be less than $ff(\vec{bsb})$.

Next, $\vec{nb}_i$ is determined as in the initialisation. Naturally, after each $\vec{bsb}$ update it needs to be checked whether some suitable condition has been satisfied so the algorithm can be exited.

3.2 Computational results

In order to experiment with mDFO we need to fix some parameters for our particular problem. First of all, we want to search for solutions to the diophantine equation (1.1) with $k = 2$, i.e. global minima of $f_2(x, y)$. This means that our search space has 2 dimensions, i.e. $D = 2$. So we denote the positions in the search space with (x, y) and we fix our fitness function as

$$ff(x, y) = f_2(x, y) = \left\| \left(2 - x^3 - y^3\right)^{\frac{1}{3}} \right\|. \tag{3.7}$$

We also need to determine the exit criterion. As we are searching for integer solutions and thus an approximate one is not good enough, we first choose some threshold thr and the algorithm looks for a pair (x_0, y_0), such that $ff(x_0, y_0) \leq thr$. Then the candidate solution $(x_0, y_0, \text{round}\,(z_0))$, where z_0 is calculated from (2.1), is plugged into (1.1) and the algorithm is exited, if the equation is satisfied.

Additionally, after some experimenting, that is in no way conclusive, we found that a particle swarm size of $s = 50$ works best in our case.

With the above fixed we tested the time performance of our algorithm for different ranges of x and y by recording the time it needs to find a solution. The full code which we used for testing is included in appendix A.1. We wrote the algorithm in C.

We chose to scan three different ranges of values for x and y, namely

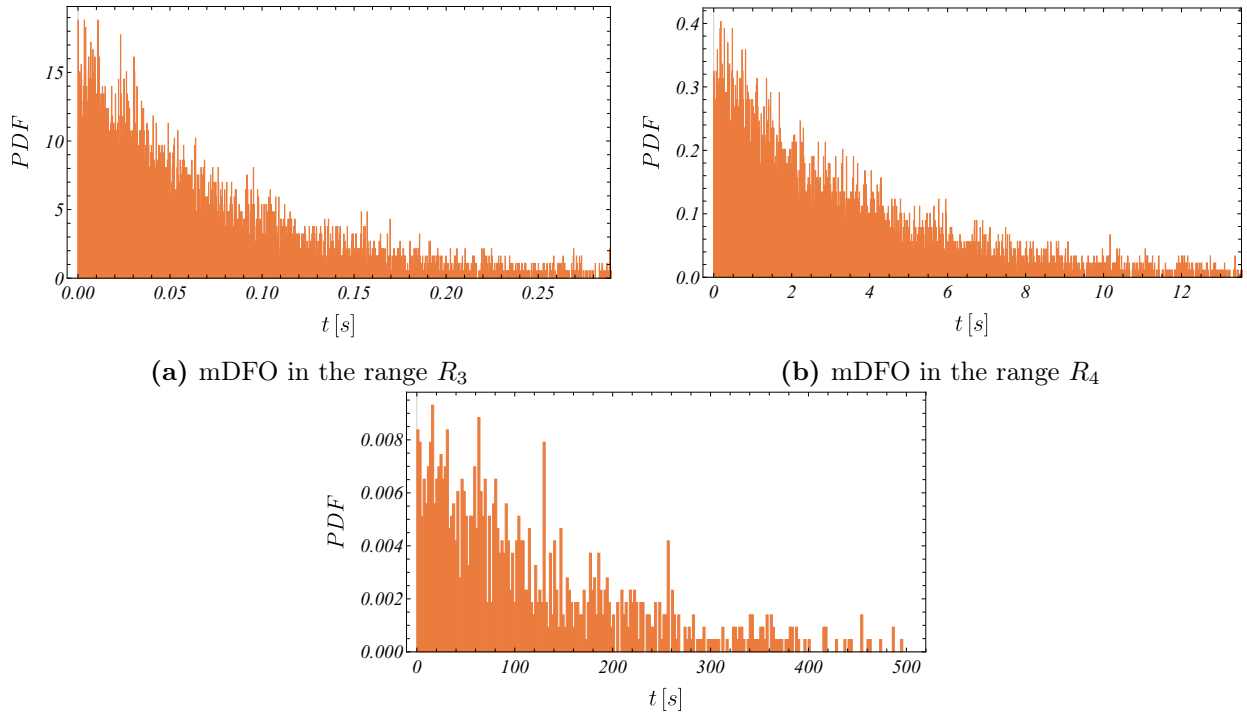
$$\begin{aligned} R_3 &= \{(x, y) :\ x, y \in \mathbb{Z},\ -10^3 \leq x \leq 0 \leq y \leq 10^3\}, \quad N = 10^4, \\ R_4 &= \{(x, y) :\ x, y \in \mathbb{Z},\ -10^4 \leq x \leq 0 \leq y \leq 10^4\}, \quad N = 10^4, \\ R_5 &= \{(x, y) :\ x, y \in \mathbb{Z},\ -10^5 \leq x \leq 0 \leq y \leq 10^5\}, \quad N = 10^3. \end{aligned} \tag{3.8}$$

Here we denote with N the respective number of runs completed by the algorithm in the corresponding range. Hence, N is also the sample size of the time data for a given range.

The results are arranged in histograms based on the data set $\{t_i\}_{i=1}^N$ of running times t_i. The data is gathered into bins with equal widths

$$\Delta t = \frac{\max\{t_i\} - \min\{t_i\}}{\ell}, \tag{3.9}$$

where $\max\{t_i\}$ is the longest time for finding a solution, $\min\{t_i\}$ is the shortest one, and $\ell < N$ is an arbitrary partition. The data is also normalised to show the corresponding probability density function (PDF) (figure 2). In this case, the data is discrete and the PDF function states that the probability of t_i falling within an interval of width Δt is the density in that range times Δt.



(a) mDFO in the range R_3

(b) mDFO in the range R_4

(c) mDFO in the range R_5

Figure 2: Time performance of mDFO. Every histogram gives the distribution of the individual times the mDFO algorithm took to find a solution in the: (a) range R_3 with sample size $N = 10^4$; (b) range R_4 with sample size $N = 10^4$; (c) range R_5 with sample size $N = 10^3$.

4 Simulated annealing

SA is a search heuristic based on the physical process of annealing [8]. It has the advantage of being effective despite its ease of implementation. SA has been extensively studied and has many variations, e.g. in the choice of cooling schedule or neighbours. Those variations of course affect the algorithm's performance.

One particularly interesting modification to SA is the inclusion of restarts. It has been shown that under some conditions restarting the algorithm according to certain criteria results in improved times for finding a desired extremum [9]. More precisely a restarting SA (rSA) algorithm with a local generation matrix and cooling schedule $\text{temp}(m) \sim \frac{1}{m}$ has probabilities that the extremum has not been reached by time m which converge to zero at least geometrically fast in m.

Here we implement two versions of SA – one without restarts and one with restarts. As will be seen, we are using a logarithmic cooling schedule and thus our implementation of the algorithm fails to satisfy all the assumptions of theorem 4.1 of [9]. Nevertheless, those are not necessary conditions and it turns out that restarting significantly improves the running time in our particular case.

4.1 Computational results

As in section 3.2, we again want to test the performance of the algorithm for $f_2(x, y)$, so this is our energy function in the context of SA. A state here is just a point $(x, y) \in \mathbb{Z}^2$. The exit criterion is also the same as in section 3.2. We use the following cooling schedule:

$$\text{temp}(m) = \frac{1}{\ln m + 0.01}, \tag{4.1}$$

where m as usual is the current iteration number.

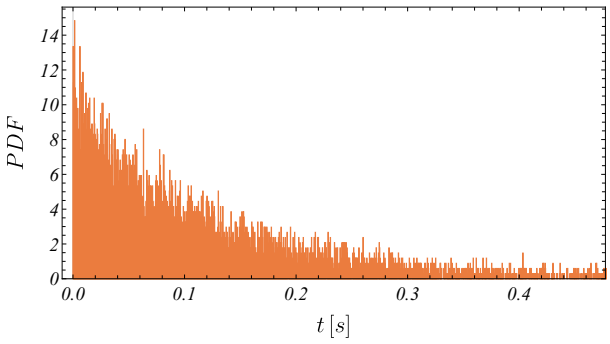
The neighbourhood of a state (x, y) is

$$n(x, y) = \{(x + a, y + b) : a, b \in \mathbb{Z},\ a, b \in [-10, 10]\} \tag{4.2}$$

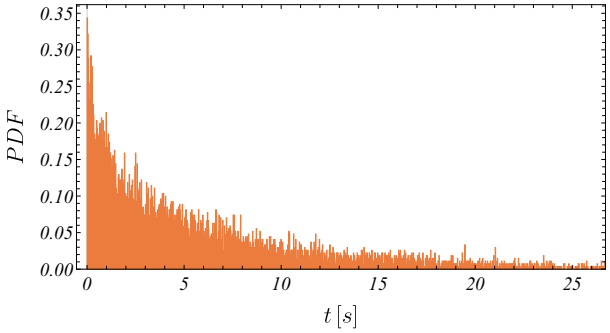
except for the states close to the border of the search space, where we remove the appropriate points so as not to end up outside. The generation matrix allows transitions from (x, y) to all points in $n(x, y)$ with equal probability except to (x, y) itself.

For the rSA algorithm, we use the criterion suggested in [9], namely restarting after rtm consecutive states have the same energy. After some experimenting, we decided to use $rtm = 30$.

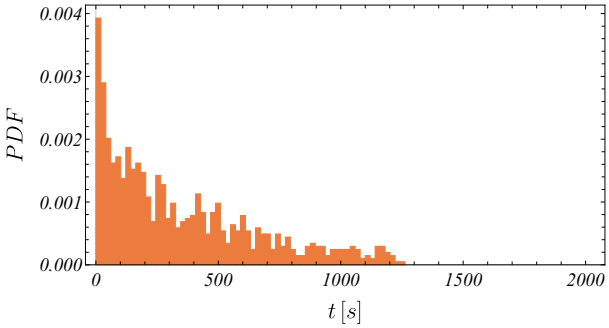
Again, we wrote the algorithms in C and include the full codes in apprendices A.2 and A.3. Figures 3 and 4 show the PDF histograms for the two versions of SA in different ranges.



(a) SA in the range R_3

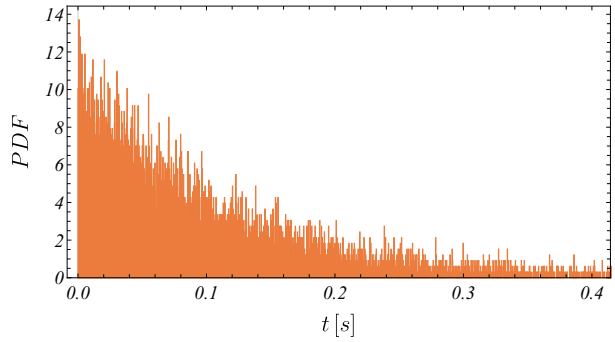


(b) SA in the range R_4

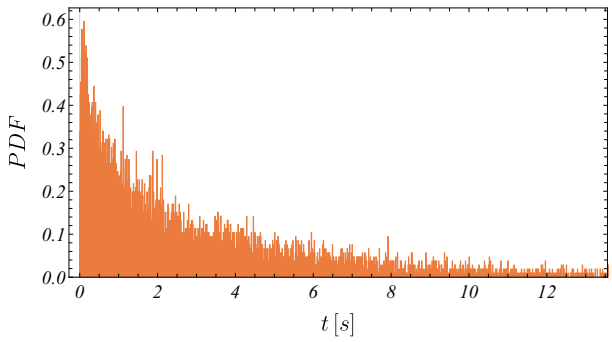


(c) SA in the range R_5

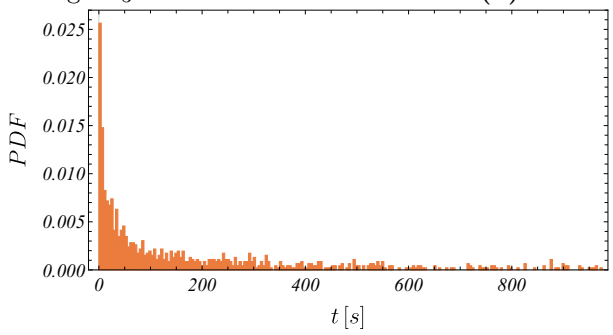
Figure 3: Time performance of SA. Every histogram gives the distribution of the individual times the SA algorithm took to find a solution in the: (a) range R_3 with sample size $N = 10^4$; (b) range R_4 with sample size $N = 10^4$; (c) range R_5 with sample size $N = 10^3$.



(a) rSA in the range R_3



(b) rSA in the range R_4



(c) rSA in the range R_5

Figure 4: Time performance of rSA. Every histogram gives the distribution of the individual times the rSA algorithm took to find a solution in the: (a) range R_3 with sample size $N = 10^4$; (b) range R_4 with sample size $N = 10^4$; (c) range R_5 with sample size $N = 10^3$.

5 Data models and statistical analysis

In this section we conduct a standard statistical analysis by modeling the time data, produced by the given algorithms, with carefully chosen continuous probability distributions. Our goal is to evaluate the relative performance of the three algorithms and furthermore to estimate their similarities. We chose to describe the accumulated time data by two statistical models, namely a simple one-parameter exponential model $f(t;\lambda)$, and a two-parameter log-normal distribution $f(t;\alpha,\beta)$. Here the stochastic variable is the individual time t a given method takes to find an integer solution to $x^3 + y^3 + z^3 = 2$.

5.1 Exponential distribution

The exponential model is a simple one-parameter probability distribution, where one assumes that the underlying statistics models a Poisson process. The PDF of an exponential distribution is given by

$$f(t;\lambda) = \begin{cases} \lambda e^{-\lambda t}, & t \geq 0 \\ 0, & t < 0 \end{cases}, \tag{5.1}$$

where $\lambda > 0$ is the rate parameter.

The expected value, the variance and the median of an exponentially distributed random variable t with rate parameter λ are well known, namely

$$\bar{t} = \mathrm{E}[t] = \frac{1}{\lambda}, \quad \mathrm{Var}[t] = \frac{1}{\lambda^2}, \quad \mathrm{Med}[t] = \frac{\ln 2}{\lambda}. \tag{5.2}$$

When a finite sample data is available the mean time $\bar{t}$ for finding a solution also coincides

with the mean time from the sample data:

$$\bar{t} = \frac{1}{N}\sum_{i=1}^{N} t_i. \tag{5.3}$$

The 95% confidence intervals for λ and t are given by

$$\lambda_{lower} \le \lambda \le \lambda_{upper}, \quad \frac{1}{\lambda_{upper}} \le \bar{t} \le \frac{1}{\lambda_{lower}}, \tag{5.4}$$

where

$$\lambda_{lower} = \lambda\left(1 - \frac{1.96}{\sqrt{N}}\right), \quad \lambda_{upper} = \lambda\left(1 + \frac{1.96}{\sqrt{N}}\right). \tag{5.5}$$

5.2 Log-normal distribution

The two-parameter log-normal distribution $f(t;\alpha,\beta)$, $\alpha \in (-\infty,\infty)$, $\beta > 0$, is a continuous probability distribution of a positive random variable $t > 0$, whose logarithm is normally distributed. There are many different parameterisations of the log-normal distribution, but we prefer the following:

$$f(t;\alpha,\beta) = \frac{1}{\beta\sqrt{2\pi}t}\mathrm{e}^{-\frac{(\ln t-\alpha)^2}{2\beta^2}}, \tag{5.6}$$

where the parameters of the distribution can be obtained directly from the sample data via

$$\alpha = \frac{1}{N}\sum_{i=1}^{N}\ln t_i, \quad \beta = \frac{1}{\sqrt{N}}\sqrt{\sum_{i=1}^{N}(\ln t_i - \alpha)^2}. \tag{5.7}$$

In this case, the mean time $\bar{t}$ for finding a solution and its standard deviation are given by

$$\bar{t} = \mathrm{E}[t] = e^{\alpha+\frac{\beta^2}{2}} \quad \mathrm{SD}[t] = \sqrt{\mathrm{Var}[t]} = e^{\alpha+\frac{\beta^2}{2}}\sqrt{e^{\beta^2}-1}, \tag{5.8}$$

where $\bar{t}$ does not coincide with the mean sample time (5.3). Furthermore, the median and the mode yield

$$\mathrm{Med}[t] = e^{\alpha}, \quad \mathrm{Mode}[t] = e^{\alpha-\beta^2}, \tag{5.9}$$

where the mode defines the point of global maximum of the probability density function.

The standard scatter intervals for the log-normal distribution are written by

$$t_{68\%} \in [e^{\alpha-\beta}, e^{\alpha+\beta}], \quad t_{95\%} \in [e^{\alpha-2\beta}, e^{\alpha+2\beta}]. \tag{5.10}$$

However, these estimates are not very informative for skew-symmetric distributions such as the log-normal one. In this case, we can extract an efficient 95% confidence interval for the log-normal model based on the Cox proposal, namely [10]

$$\bar{t}_{95\%} \in e^{\left[\alpha+\frac{\beta^2}{2}-1.96\sqrt{\frac{\beta^2}{N}+\frac{\beta^4}{2(N-1)}}, \alpha+\frac{\beta^2}{2}+1.96\sqrt{\frac{\beta^2}{N}+\frac{\beta^4}{2(N-1)}}\right]}, \tag{5.11}$$

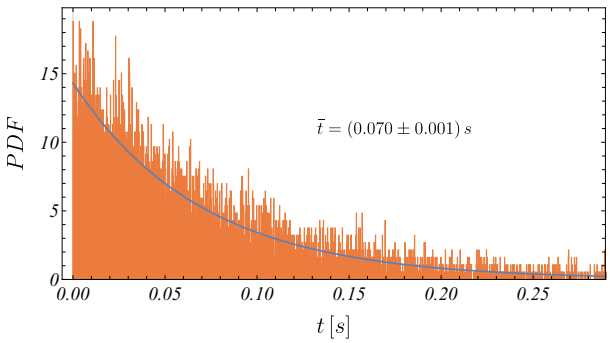
where we can estimate an absolute confidence $\delta t = \max|\bar{t} - \bar{t}_{95\%}|$.

5.3 Statistical models for the mDFO time data

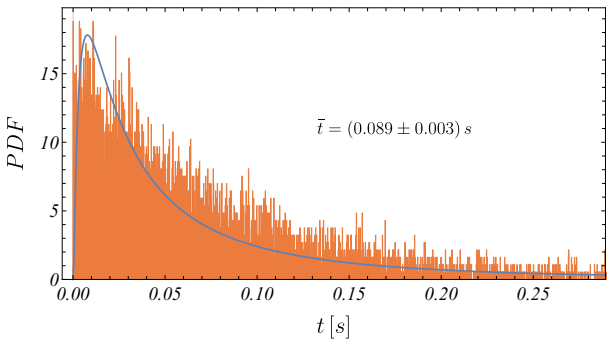
5.3.1 mDFO time data in the range R_3

We begin by analyzing our mDFO method. We looked for solutions to $x^3 + y^3 + z^3 = 2$ in the lowest range R_3. In this case, the method was tested $N = 10^4$ times. The produced set of running times $\{t_i\}_{i=1}^{N}$ is divided into bins with width $\Delta t = 0.00019\,s$ and its PDF histogram is shown in figure 2a.

The two statistical models, describing the mDFO time data, are shown in figure 5. The first one is a simple one-parameter exponential model $f(t;\lambda)$ with $\lambda = 14.301\,s^{-1}$ (figure 5a). The second one is a two-parameter log-normal distribution $f(t;\alpha,\beta)$ with $\alpha = -3.229$, $\beta = 1.275$ (figure 5b).



(a) mDFO time data exponential fit.



(b) mDFO time data log-normal fit.

Figure 5: Statistical models for the mDFO time data in the range R_3. (a) shows a simple one-parameter exponential probability distribution $f(t;\lambda)$ with $\lambda = 14.301\,s^{-1}$ and mean time for finding a solution $\bar{t} = 0.070\,s$. (b) shows a two-parameter log-normal distribution fit $f(t;\alpha,\beta)$ with $\alpha = -3.229$, $\beta = 1.275$, and mean time for finding a solution $\bar{t} = 0.089\,s$.

Exponential model

The expected value, the variance and the median of an exponentially distributed random variable t with rate parameter $\lambda = 14.301\,s^{-1}$ yield

$$\bar{t} = \mathrm{E}[t] = \frac{1}{\lambda} = 0.070\,s, \quad \mathrm{Var}[t] = \frac{1}{\lambda^2} = 0.005\,s^2, \quad \mathrm{Med}[t] = \frac{\ln 2}{\lambda} = 0.048\,s. \tag{5.12}$$

The 95% confidence intervals for λ and $\bar{t}$ are given by

$$\lambda_{lower} \leq \lambda \leq \lambda_{upper}, \quad \lambda_{upper}^{-1} \leq \bar{t} \leq \lambda_{lower}^{-1}, \tag{5.13}$$

where

$$\lambda_{lower} = \lambda\left(1 - \frac{1.96}{\sqrt{N}}\right) \approx 14.021\,s^{-1}, \quad \lambda_{upper} = \lambda\left(1 + \frac{1.96}{\sqrt{N}}\right) \approx 14.581\,s^{-1}, \tag{5.14}$$

$$\lambda_{upper}^{-1} = 0.069\,s, \quad \lambda_{lower}^{-1} = 0.071\,s, \tag{5.15}$$

with absolute confidences $\delta\lambda = |\lambda - \lambda_{upper}| = 0.280\,s^{-1}$ and $\delta t = |\bar{t} - \lambda_{lower}^{-1}| = 0.001\,s$. Therefore, we can write our results for the Poisson distributed mDFO time data as

$$\lambda = (14.301 \pm 0.280)\,s^{-1}, \quad \bar{t} = (0.070\ \pm 0.001)\,s. \tag{5.16}$$

Because λ and $\bar{t}$ are inversely proportional to each other, from now on we will be interested only in $\bar{t}$.

Log-normal model

The two-parameter log-normal distribution $f(t;\alpha,\beta)$ for the mDFO time data has estimated parameters $\alpha = -3.229$, $\beta = 1.275$, which is depicted in figure 5b. The parameters can be obtained from the sample data via

$$\alpha = \frac{1}{N}\sum_{i=1}^{N} \ln t_i = -3.229, \quad \beta = \frac{1}{\sqrt{N}}\sqrt{\sum_{i=1}^{N} (\ln t_i - \alpha)^2} = 1.275. \tag{5.17}$$

Consequently, the mean time $\bar{t}$ for finding a solution and its standard deviation are given by

$$\bar{t} = \mathrm{E}[t] = e^{\alpha+\frac{\beta^2}{2}} = 0.089\,s, \quad \mathrm{SD}[t] = \sqrt{\mathrm{Var}[t]} = e^{\alpha+\frac{\beta^2}{2}}\sqrt{e^{\beta^2}-1} = 0.180\,s. \tag{5.18}$$

Furthermore, the median and the mode yield

$$\mathrm{Med}[t] = e^{\alpha} = 0.040\,s, \quad \mathrm{Mode}[t] = e^{\alpha-\beta^2} = 0.008\,s, \tag{5.19}$$

The standard scatter intervals for the log-normal distribution are given by

$$t_{68\%} \in [e^{\alpha-\beta}, e^{\alpha+\beta}] = [0.011\,s, 0.142\,s], \tag{5.20}$$

for the 68% confidence interval, and

$$t_{95\%} \in [e^{\alpha-2\beta}, e^{\alpha+2\beta}] = [0.003\,s, 0.507\,s], \tag{5.21}$$

for the 95% confidence interval.

The efficient 95% confidence interval for $\bar{t}$ yields

$$\bar{t}_{95\%} \in e^{\left[\alpha+\frac{\beta^2}{2}-1.96\sqrt{\frac{\beta^2}{N}+\frac{\beta^4}{2(N-1)}}, \alpha+\frac{\beta^2}{2}+1.96\sqrt{\frac{\beta^2}{N}+\frac{\beta^4}{2(N-1)}}\right]} = [0.086\,s,\ 0.092\,s], \tag{5.22}$$

with absolute confidence $\delta t = \max|\bar{t} - \bar{t}_{95\%}| = 0.003\,s$, thus

$$\bar{t} = (0.089\ \pm 0.003)\,s. \tag{5.23}$$

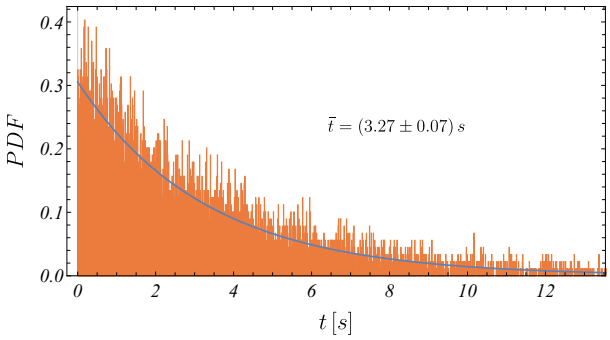
The relevant data is collected in table 1.

Param. / Dist.	α	β	$\lambda\ [s^{-1}]$	$\bar{t} \pm \delta t\ [s]$	$\bar{t}_{95\%}\ [s]$	Med$[t]\ [s]$	Mode$[t]\ [s]$
Exponential	–	–	14.301	0.070 ± 0.001	[0.069,0.071]	0.048	–
Log-normal	-3.229	1.275	–	0.089 ± 0.003	[0.086,0.092]	0.040	0.008

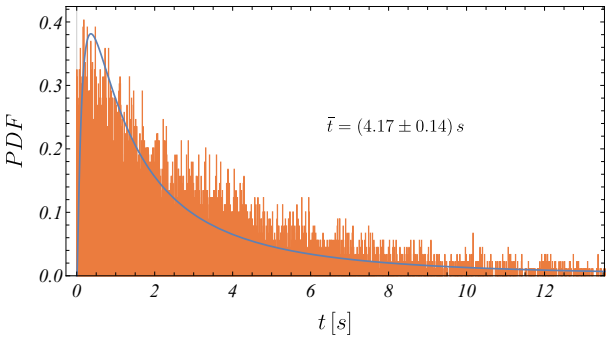
Table 1: The relevant characteristics of the models for mDFO in R_3.

5.3.2 mDFO time data in the range R_4

Next, we analyze the mDFO time data, accumulated when looking for integer solutions to $x^3 + y^3 + z^3 = 2$ in the mid range R_4. In this case, the method was tested $N = 10^4$ times. The produced time set $\{t_i\}_{i=1}^{N}$ is divided into bins with width $\Delta t = 0.01\,s$ and its PDF histogram is shown in figure 2b. As in the previous case, we model the mDFO time data by an exponential model, shown in figure 6a, and a log-normal model, shown in figure 6b, with the relevant characteristics collected in table 2.



(a) mDFO time data exponential fit.



(b) mDFO time data log-normal fit.

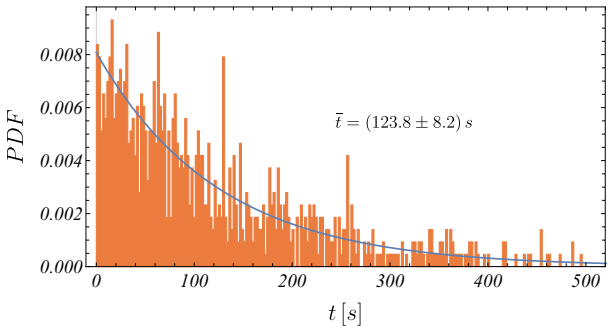
Figure 6: Statistical models for the mDFO time data in the range R_4. (a) shows a one-parameter exponential model fit $f(t;\lambda)$ with $\lambda = 0.31\, s^{-1}$ and mean time for finding a solution $\bar{t} = 3.27\, s$. (b) shows a two-parameter log-normal distribution fit $f(t;\alpha,\beta)$ with $\alpha = 0.62$, $\beta = 1.28$, and mean time for finding a solution $\bar{t} = 4.17\, s$.

Param. / Dist.	α	β	$\lambda\ [s^{-1}]$	$\bar{t} \pm \delta t\ [s]$	$\bar{t}_{95\%}\ [s]$	Med[t] $[s]$	Mode[t] $[s]$
Exponential	–	–	0.31	3.27 $\pm$ 0.07	[3.21,3.34]	2.27	–
Log-normal	0.62	1.28	–	4.17 $\pm$ 0.14	[4.03,4.31]	1.85	0.36

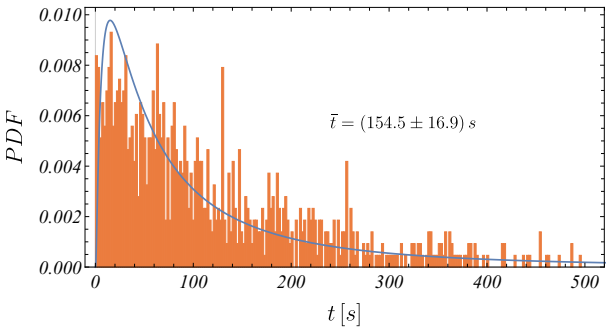
Table 2: The relevant characteristics of the models for mDFO in R_4.

5.3.3 mDFO time data in the range R_5

We continue our analysis by looking for solutions to $x^3 + y^3 + z^3 = 2$ in the range R_5. In this case, the method was tested $N = 10^3$ times. The produced time set $\{t_i\}_{i=1}^{N}$ is divided into bins with width $\Delta t = 2.15\, s$ and its PDF histogram is shown in figure 2c. The statistical models, describing the mDFO time data, are shown in figures 7a and 7b and their characteristics – in table 3.



(a) mDFO time data exponential fit.



(b) mDFO time data log-normal fit.

Figure 7: Statistical models for the mDFO time data the range R_5. (a) shows a one-parameter exponential model fit $f(t;\lambda)$ with $\lambda = 0.008\, s^{-1}$ and mean time for finding a solution $\bar{t} = 123.8\, s$. (b) shows a two-parameter log-normal distribution fit $f(t;\alpha,\beta)$ with $\alpha = 4.27$, $\beta = 1.25$, and mean time for finding a solution $\bar{t} = 154.5\, s$.

Param. / Dist.	α	β	$\lambda\ [s^{-1}]$	$\bar{t} \pm \delta t\ [s]$	$\bar{t}_{95\%}\ [s]$	Med$[t]\ [s]$	Mode$[t]\ [s]$
Exponential	–	–	0.008	123.8 $\pm$ 8.2	[116.6,132.0]	85.8	–
Log-normal	4.27	1.25	–	154.5 $\pm$ 17.0	[139.5,171.5]	71.1	15.1

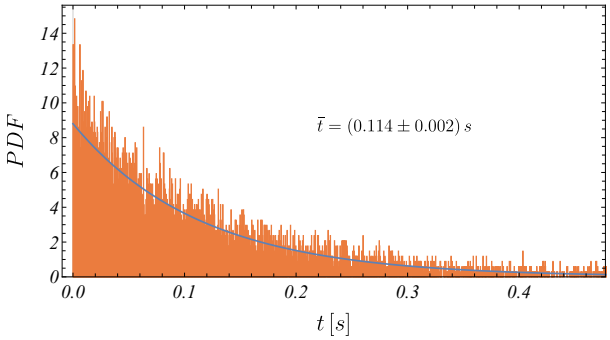
Table 3: The relevant characteristics of the models for mDFO in R_5.

5.4 Statistical models for the SA algorithm time data

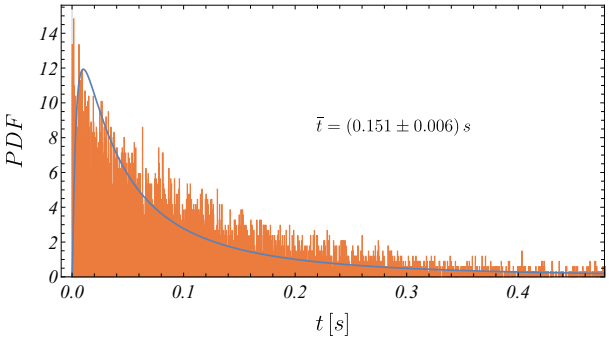
5.4.1 SA time data in the range R_3

In this section we focus on the time data accumulated from the SA algorithm (without restarts). The analysis mimics the one for the mDFO method.

In the lowest range R_3 the produced time data $\{t_i\}_{i=1}^{N}$ is divided into bins with width $\Delta t = 0.00034\,s$ and its PDF histogram is shown in figure 3a. The chosen statistical models, describing the time data of the algorithm, are shown in figures 8a and 8b, correspondingly. Their characteristics are collected in table 4.



(a) Exponential fit for SA.



(b) Log-normal fit for SA.

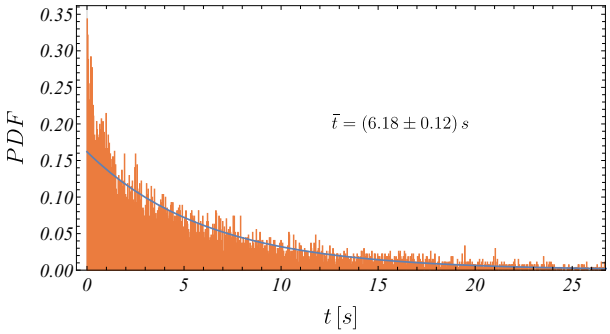
Figure 8: Statistical models for the SA time data in the range R_3. (a) shows an exponential model fit $f(t;\lambda)$ with $\lambda = 8.800\,s^{-1}$ and mean time for finding a solution $\bar{t} = 0.114\,s$. (b) shows a log-normal distribution fit $f(t;\alpha,\beta)$ with $\alpha = -2.791$, $\beta = 1.343$, and mean time for finding a solution $\bar{t} = 0.151\,s$.

Param. / Dist.	α	β	$\lambda\ [s^{-1}]$	$\bar{t} \pm \delta t\ [s]$	$\bar{t}_{95\%}\ [s]$	Med$[t]\ [s]$	Mode$[t]\ [s]$
Exponential	–	–	8.800	0.114 $\pm$ 0.002	[0.111,0.116]	0.079	–
Log-normal	-2.791	1.343	–	0.151 $\pm$ 0.006	[0.146,0.157]	0.061	0.010

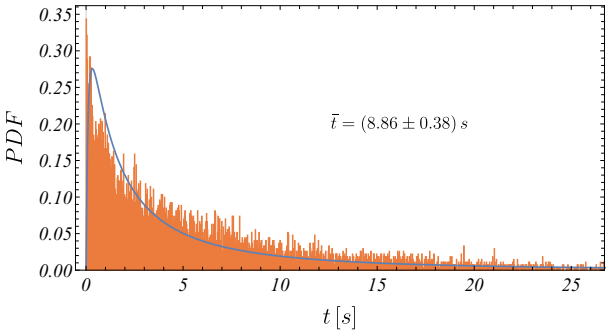
Table 4: The relevant characteristics of the models for SA in R_3.

5.4.2 SA time data in the range R_4

We consider the range R_4 with $N = 10^4$. The produced time set $\{t_i\}_{i=1}^{N}$ is divided into bins with width $\Delta t = 0.027\,s$ and its PDF histogram is shown in figure 3b. The considered statistical models, describing the SA time data, are shown in figures 9a and 9b and their characteristics – in table 5.



(a) Exponential fit for SA.



(b) Log-normal fit for SA.

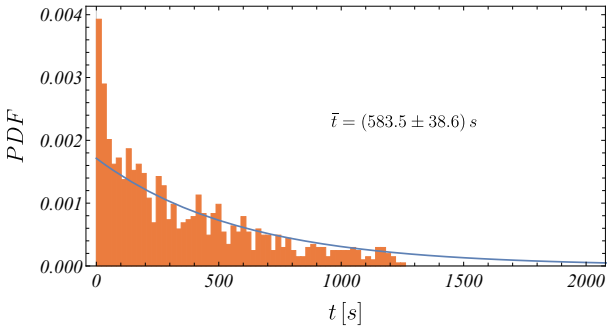
Figure 9: Statistical models for the SA time data in the range R_4. (a) shows an exponential model fit $f(t;\lambda)$ with $\lambda = 0.16\, s^{-1}$ and mean time for finding a solution $\bar{t} = 6.18\, s$. (b) shows a log-normal distribution fit $f(t;\alpha,\beta)$ with $\alpha = 1.08$, $\beta = 1.49$, and mean time for finding a solution $\bar{t} = 8.86\, s$.

Param. / Dist.	α	β	$\lambda\ [s^{-1}]$	$\bar{t} \pm \delta t\ [s]$	$\bar{t}_{95\%}\ [s]$	Med[t] $[s]$	Mode[t] $[s]$
Exponential	–	–	0.16	6.18 $\pm$ 0.12	[6.06,6.30]	4.28	–
Log-normal	1.08	1.49	–	8.86 $\pm$ 0.38	[8.50,9.25]	2.94	0.32

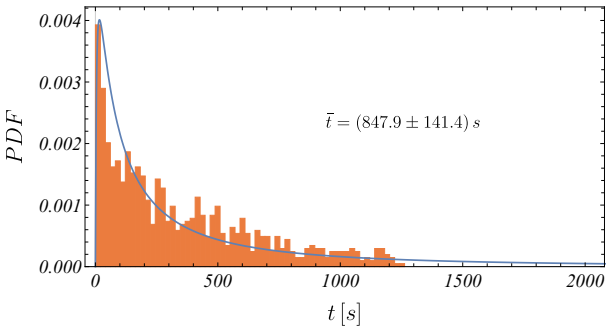
Table 5: The relevant characteristics of the models for SA in R_4.

5.4.3 SA time data in the range R_5

Next, we analyze the SA data in the range R_5 with $N = 10^3$ tests. The produced time set $\{t_i\}_{i=1}^{N}$ is divided into bins with width $\Delta t = 20.4\, s$ and its PDF histogram is shown in figure 3c. The statistical models are shown in figures 10a and 10b, correspondingly. Table 6 shows their characteristics.



(a) Exponential distribution fit for SA.



(b) Log-normal distribution fit for SA.

Figure 10: Statistical models for the SA method in the range R_5. (a) shows an exponential model fit $f(t;\lambda)$ with $\lambda = 0.0017\, s^{-1}$, and mean time for finding a solution $\bar{t} = 583.5\, s$. (b) shows a log-normal distribution fit $f(t;\alpha,\beta)$ with $\alpha = 5.43$, $\beta = 1.62$, and mean time for finding a solution $\bar{t} = 847.9\, s$.

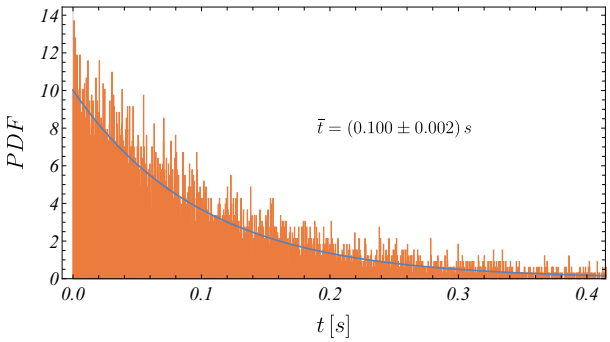
Param. / Dist.	α	β	$\lambda\ [s^{-1}]$	$\bar{t} \pm \delta t\ [s]$	$\bar{t}_{95\%}\ [s]$	Med[t] $[s]$	Mode[t] $[s]$
Exponential	–	–	0.0017	583.5 $\pm$ 38.6	[549.4,622.0]	404.4	–
Log-normal	5.43	1.62	–	847.9 $\pm$ 141.4	[728.6,989.3]	228.2	16.5

Table 6: The relevant characteristics of the models for SA in R_5.

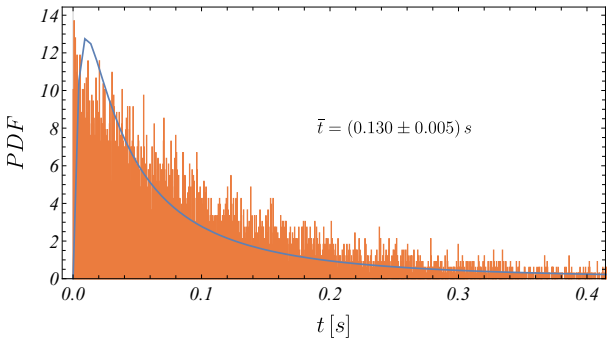
5.5 Statistical models for the rSA algorithm time data

5.5.1 rSA time data in the range R_3

Here, we consider the rSA method in the lowest range R_3 and $N = 10^4$. The produced time set $\{t_i\}_{i=1}^{N}$ is divided into bins with width $\Delta t = 0.00033\,s$ and its PDF histogram is shown in figure 4a. The relevant statistical models are depicted in figures 11a and 11b with their characteristics shown in table 7.



(a) Exponential fit for rSA.



(b) Log-normal fit for rSA.

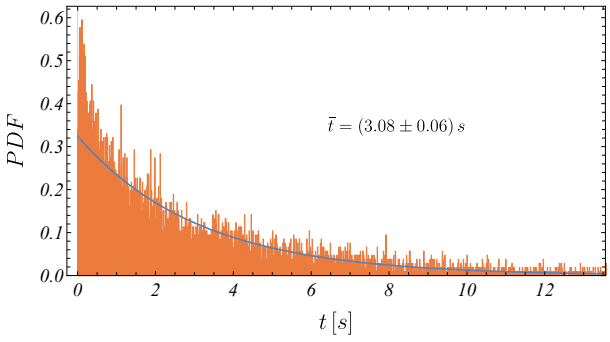
Figure 11: Statistical models for the rSA method in the range R_3. (a) shows an exponential model fit $f(t;\lambda)$ with $\lambda = 10.014\,s^{-1}$ and mean time for finding a solution $\bar{t} = 0.100\,s$. (b) shows a log-normal distribution fit $f(t;\alpha,\beta)$ with $\alpha = -2.886$, $\beta = 1.298$, and mean time for finding a solution $\bar{t} = 0.130\,s$.

Param. / Dist.	α	β	$\lambda\ [s^{-1}]$	$\bar{t} \pm \delta t\ [s]$	$\bar{t}_{95\%}\ [s]$	Med$[t]\ [s]$	Mode$[t]\ [s]$
Exponential	–	–	10.014	0.100 ± 0.002	[0.098,0.102]	0.070	–
Log-normal	-2.886	1.298	–	0.130 ± 0.005	[0.125,0.134]	0.056	0.010

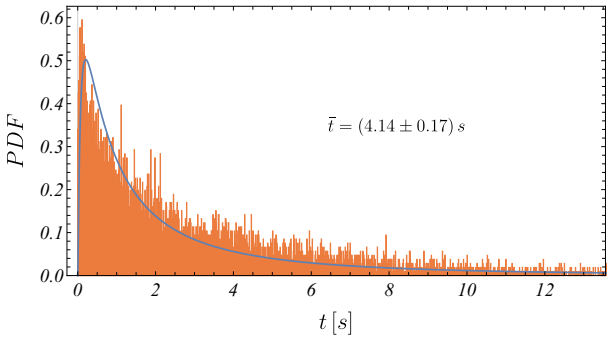
Table 7: The relevant characteristics of the models for rSA in R_3.

5.5.2 rSA time data in the range R_4

We consider the range R_4 with $N = 10^4$. The time data $\{t_i\}_{i=1}^{N}$ is divided into bins with width $\Delta t = 0.011\,s$ and its PDF histogram is shown in figure 4b. The statistical models are also shown in figures 12a and 12b. Table 8 shows the relevant characteristics.



(a) Exponential fit for rSA.



(b) Log-normal fit for rSA.

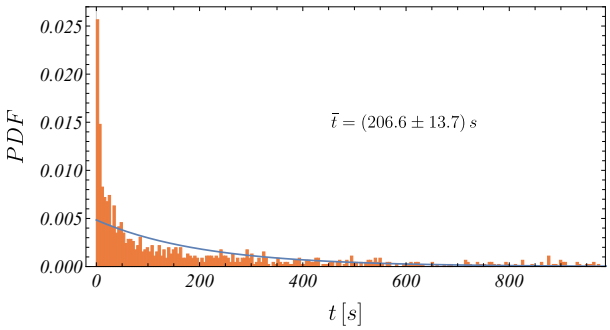
Figure 12: Statistical models for the rSA time data in the range R_4. (a) shows a one-parameter exponential model fit $f(t;\lambda)$ with $\lambda = 0.32\,s^{-1}$ and mean time for finding a solution $\bar{t} = 3.08\,s$. (b) shows a two-parameter log-normal distribution fit $f(t;\alpha,\beta)$ with $\alpha = 0.42$, $\beta = 1.41$, and mean time for finding a solution $\bar{t} = 4.14\,s$.

Param. / Dist.	α	β	$\lambda\ [s^{-1}]$	$\bar{t} \pm \delta t\ [s]$	$\bar{t}_{95\%}\ [s]$	Med$[t]\ [s]$	Mode$[t]\ [s]$
Exponential	–	–	0.32	3.08 $\pm$ 0.06	[3.02,3.14]	2.14	–
Log-normal	0.42	1.41	–	4.14 $\pm$ 0.17	[3.98,4.30]	1.52	0.21

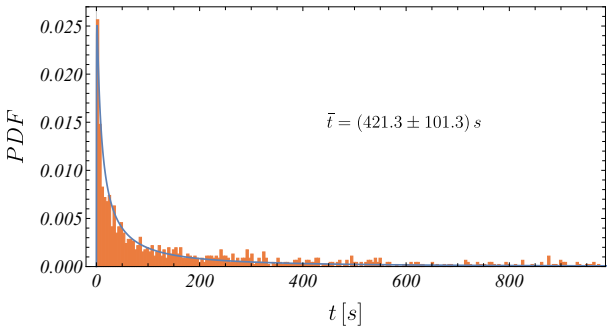
Table 8: The relevant characteristics of the models for rSA in R_4.

5.5.3 rSA time data in the range R_5

The final range is R_5 with $N = 10^3$. The accumulated time data $\{t_i\}_{i=1}^{N}$ is divided into bins with width $\Delta t = 4.6\,s$ and its PDF histogram is shown in figure 4c. The statistical data models are depicted in figures 13a and 13b. Their characteristics are in table 9.



(a) Exponential fit for rSA.



(b) Log-normal fit for rSA.

Figure 13: Statistical data models for rSA in the range R_5. (a) shows an exponential model fit $f(t;\lambda)$ with $\lambda = 0.0048\,s^{-1}$ and mean time for finding a solution $\bar{t} = 206.6\,s$. (b) shows a log-normal distribution fit $f(t;\alpha,\beta)$ with $\alpha = 4.06$, $\beta = 1.99$, and mean time for finding a solution $\bar{t} = 421.3\,s$.

Param. / Dist.	α	β	$\lambda\ [s^{-1}]$	$\bar{t} \pm \delta t\ [s]$	$\bar{t}_{95\%}\ [s]$	Med$[t]\ [s]$	Mode$[t]\ [s]$
Exponential	–	–	0.0048	206.6 $\pm$ 13.7	[194.5,220.2]	143.2	–
Log-normal	4.06	1.99	–	421.3 $\pm$ 101.3	[341.0,522.6]	57.9	1.1

Table 9: The relevant characteristics of the models for rSA in R_5.

5.6 Parametric and non-parametric statistics

5.6.1 Parametric statistics based on the Fisher metric

In order to compare how dissimilar our statistical models are relative to each other, we need the explicit form of the Fisher information metric [11–15] for our distribution functions. Let $f(\vec{u};\vec{\xi})$ be a PDF of some statistical model for a d-dimensional random variable U with parameters $\vec{\xi} = (\xi^1, \xi^2, \ldots, \xi^n)$. The Fisher metric is defined by the following integral over the range of U:

$$g_{ab}(\vec{\xi}) = \int_U \frac{\partial \ln f(\vec{u};\vec{\xi})}{\partial \xi^a} \frac{\partial \ln f(\vec{u};\vec{\xi})}{\partial \xi^b} f(\vec{u};\vec{\xi}) \mathrm{d}^d u\,, \quad a,b = 1,\ldots,n. \tag{5.24}$$

For one-dimensional models, consisting of a single free parameter, the above definition reduces to the so-called Fisher information

$$I(\xi) = \int_U \left(\frac{\partial \ln f(\vec{u};\xi)}{\partial \xi} \right)^2 f(\vec{u};\xi) \mathrm{d}^d u. \tag{5.25}$$

The Fisher metric plays the role of a Riemannian metric on the space of parameters $\vec{\xi} = (\xi^1, \xi^2, \ldots, \xi^n)$, where every point defines a different statistical model (or a PDF). We will not distinguish a given point $\vec{\xi}$ in the parameter space and its associated PDF $f(\vec{u}; \vec{\xi})$. Hence, given two points on the manifold their geodesic distance is interpreted as the statistical distinguishability of the PDFs [16].

The action for the geodesics on the statistical manifold is given by the functional

$$L = \int_{r_1}^{r_2} \sqrt{g_{ab}(\vec{\xi}) \frac{\mathrm{d}\xi^a(r)}{\mathrm{d}r} \frac{\mathrm{d}\xi^b(r)}{\mathrm{d}r}}\, \mathrm{d}r, \tag{5.26}$$

which under variation yields the system of geodesic equations

$$\frac{\mathrm{d}^2\xi^c(r)}{\mathrm{d}r^2} + \Gamma^c_{ab} \frac{\mathrm{d}\xi^a(r)}{\mathrm{d}r} \frac{\mathrm{d}\xi^b(r)}{\mathrm{d}r} = 0, \quad c = 1, ..., n. \tag{5.27}$$

The invariant geodesic length L between statistical models is then obtained from (5.26) after solving (5.27) for the geodesic profiles of the parameters $\xi^a(r)$ as functions of some proper ordering parameter r.

For models with a single parameter one can determine the Fisher distance exactly up to a scale factor. For example, the Fisher information (metric) for the exponential distribution (5.1) is given by

$$g_{\lambda\lambda} = I(\lambda) = \frac{1}{\lambda^2}. \tag{5.28}$$

Therefore, one can compute the distance function for this model directly by solving a single geodesic equation. For this purpose, we find the inverse metric $g^{\lambda\lambda} = \lambda^2$ and the Christoffel symbol $\Gamma^\lambda_{\lambda\lambda} = g^{\lambda\lambda}\partial_\lambda g_{\lambda\lambda}/2 = -1/\lambda$. Thus, the geodesic equation for the model parameter $\lambda(r)$ is

$$\frac{\mathrm{d}^2\lambda}{\mathrm{d}r^2} - \frac{1}{\lambda}\left(\frac{\mathrm{d}\lambda}{\mathrm{d}r}\right)^2 = 0 \tag{5.29}$$

with the simple solution $\lambda(r) = c_2 e^{c_1 r}$. Imposing boundary conditions, $\lambda(0) = \lambda_1$ and $\lambda(1) = \lambda_2$, one finds $c_1 = \ln(\lambda_2/\lambda_1)$ and $c_2 = \lambda_1$. Therefore, the geodesic distance between two statistical exponential models with corresponding parameters λ_1 and λ_2 is written by [17]

$$L_{12} = L(\lambda_1, \lambda_2) = \left| \int_0^1 \sqrt{g_{\lambda\lambda}\left(\frac{\mathrm{d}\lambda}{\mathrm{d}r}\right)^2}\, \mathrm{d}r \right| = \left| \ln \frac{\lambda_2}{\lambda_1} \right|. \tag{5.30}$$

On the other hand, the Fisher metric for the log-normal distribution (5.6) is given by

$$ds^2 = g_{ab}(\vec{\xi})\mathrm{d}\xi^a \mathrm{d}\xi^b = \frac{\mathrm{d}\alpha^2 + 2\mathrm{d}\beta^2}{\beta^2}. \tag{5.31}$$

The geodesic profiles for $\alpha(r)$ and $\beta(r)$ under this metric are given by the coupled system of ordinary second order differential equations

$$\alpha''(r) - \frac{2\beta'(r)}{\beta(r)}\alpha'(r) = 0, \quad \beta''(r) - \frac{\beta'(r)^2}{\beta(r)} + \frac{\alpha'(r)^2}{2\beta(r)} = 0, \tag{5.32}$$

together with the boundary conditions $\alpha(0) = \alpha_1$, $\alpha(1) = \alpha_2$, $\beta(0) = \beta_1$, $\beta(1) = \beta_2$.

In what follows, we will compute the Fisher distances between our models in the given ranges and find out how dissimilar they are from each other. For shortness of notation we will use the following indices: 1 for mDFO, 2 for SA, and 3 for rSA.

We begin by computing the Fisher distances between our exponential distributions for the time data in the range R_3, namely

$$L_{12} = \left|\ln \frac{8.80}{14.30}\right| = 0.49, \quad L_{13} = \left|\ln \frac{10.01}{14.30}\right| = 0.36, \quad L_{23} = \left|\ln \frac{10.01}{8.00}\right| = 0.13. \tag{5.33}$$

With similar computation one finds the Fisher distances in the range R_4:

$$L_{12} = \left|\ln \frac{0.305}{0.162}\right| = 0.635, \quad L_{13} = \left|\ln \frac{0.305}{0.324}\right| = 0.060, \quad L_{23} = \left|\ln \frac{0.324}{0.162}\right| = 0.695. \tag{5.34}$$

Finally, in the range R_5, one finds

$$L_{12} = \left|\ln \frac{0.008}{0.002}\right| = 1.550, \quad L_{13} = \left|\ln \frac{0.008}{0.005}\right| = 0.512, \quad L_{23} = \left|\ln \frac{0.005}{0.002}\right| = 1.038. \tag{5.35}$$

If we want to compare the log-normal models, we can find numerically the functions $\alpha(r)$ and $\beta(r)$ from (5.32) and consequently calculate the following integral:

$$L_{ij} = \int_0^1 \frac{1}{\beta(r)} \sqrt{\alpha'^2 + 2\beta'^2}\, \mathrm{d}r, \quad i, j = 1, 2, 3, \quad i \neq j, \tag{5.36}$$

with the proper boundary conditions, namely $(\alpha(0), \beta(0)) = (\alpha_i, \beta_i)$ and $(\alpha(1), \beta(1)) = (\alpha_j, \beta_j)$. In this case, one can show that in R_3 the Fisher distances between the tree log-normal models are

$$L_{12} = 0.34, \quad L_{13} = 0.27, \quad L_{23} = 0.09. \tag{5.37}$$

For the models in the mid range R_4 we find

$$L_{12} = 0.40, \quad L_{13} = 0.20, \quad L_{23} = 0.46. \tag{5.38}$$

And finally, in R_5, one has

$$L_{12} = 0.88, \quad L_{13} = 0.67, \quad L_{23} = 0.81. \tag{5.39}$$

It is useful to collect the results in tables (table 10).

One can infer that in the lowest range R_3, when considering the exponential distribution, the SA and rSA algorithms are similar relative to each other ($L_{23} = 0.13$, i.e. they are closest), while they are quite dissimilar to mDFO ($L_{12} = 0.49$ and $L_{13} = 0.36$). The same is valid also for the log-normal model in R_3.

On the other hand, in the mid range R_4, the mDFO and rSA algorithms are similar relative to each other, for example $L_{13} = 0.060$, while they are notably dissimilar to SA, i.e. $L_{12} = 0.635$ and $L_{23} = 0.695$. This result persists also in the next range R_5.

R_3			
	L_{12}	L_{13}	L_{23}
Exponential	0.49	0.36	0.13
Log-normal	0.34	0.27	0.09

(a) L_{12}, L_{13} and L_{23} for the exponential and log-normal fits in the range R_3.

R_4			
	L_{12}	L_{13}	L_{23}
Exponential	0.635	0.060	0.695
Log-normal	0.40	0.20	0.46

(b) L_{12}, L_{13} and L_{23} for the exponential and log-normal fits in the range R_4.

R_5			
	L_{12}	L_{13}	L_{23}
Exponential	1.550	0.512	1.038
Log-normal	0.88	0.67	0.81

(c) L_{12}, L_{13} and L_{23} for the exponential and log-normal fits in the range R_5.

Table 10: Geodesic distances in the parameter spaces of the respective distributions between the three algorithms in the tested ranges. We use the indices of L to denote the following: 1 for mDFO, 2 for SA, and 3 for rSA.

5.6.2 Non-parametric statistic based on Friedman test and posthoc Nemenyi test

Range	mDFO, $\bar{t}$, $[s]$	rank	rSA, $\bar{t}$, $[s]$	rank	SA, $\bar{t}$, $[s]$	rank
R_3	0.07	1	0.10	2	0.11	3
R_4	3.27	2	3.08	1	6.18	3
R_5	124	1	202	2	584	3
Total rank r_i		4		5		9

Table 11: The data average times $\bar{t}$ and their ranks according to the Friedman non-parametric test.

In Table 11 we perform the standard non-parametric statistical Friedman test over the average times $\bar{t}$ from the sample data of the three methods. The test requires to rank the measurements by their ascending value. For example: $\bar{t} = 0.07$ is ranked 1, $\bar{t} = 0.1$ has rank 2 and $\bar{t} = 0.07$ has rank 3. We are testing the following hypotheses. The null hypothesis H_0 states that all mean values $\bar{t}$ are equal. The alternative hypothesis H_1: they are different. The Friedman test yields:

$$F_r = \frac{12}{nk(k+1)} \sum_{i=1}^{k} r_i^2 - 3n(k+1) = 4.66(6), \tag{5.40}$$

where $n = 3$ is the number of ranges, $k = 3$ is the number of methods. The number $F_r = 4.66$ is smaller than the corresponding pvalue= 0.05 chi-squared statistics: $\chi^2_{\alpha,k-1} = \chi^2_{0.05,2} = 5.99 > F_r$, thus we cannot confidently reject H_0 at this level of confidence. The rejection of H_0 starts at 9.7% corresponding to $\chi^2_{0.097,2} = F_r$.

After the Friedman test one can apply the posthoc Nemenyi test in order to find out how much the methods differ statistically from each other. The results of the test are shown in Table 12. One can draw the conclusion that mDFO and rSA do not differ significantly, while both differ significantly from SA.

Method	mDFO	rSA	SA
mDFO	1.00	0.90	0.10
rSA	0.90	1.00	0.23
SA	0.10	0.23	1.00

Table 12: Posthoc Nemenyi test.

6 Conclusion

In this paper we adapted the number-theoretic sum of three cubes problem to an optimisation setting. This was motivated by the desire to use a random search algorithm to hopefully improve the time it takes to find a solution. Turning the problem into an optimisation one was not hard and resulted in equation (2.2). However, finding a global minimum to (2.2) with sufficient speed turned out to be a highly non-trivial task (as was expected).

Our attempts in this direction led us to develop a modified version of DFO and to test its performance against two other search heuristics in three ranges for (x, y) when applied to our problem in the special case $k = 2$. These last two algorithms are more or less direct implementations of SA and rSA.

The metric for the performance of all algorithms was the time it takes them to reach a solution to (1.1) (i.e. a global minimum of (2.2)). After a large number of tests, we analysed the results by fitting the respective datasets of running times with two different distributions – exponential and log-normal.

We have analysed two specific aspects of the algorithms, namely their time performance and their similarity. A conclusion about the time performance can be made by looking both at the mean and the mode of the running times (collected in table 13), while the relative similarity between the algorithms can be measured by the Fisher distances between the respective PDFs (table 10).

R_3			
	mDFO	SA	rSA
$\bar{t}_{\text{exp}}$ $[s]$	0.070 ± 0.001	0.114 ± 0.002	0.100 ± 0.002
$\bar{t}_{\text{l-n}}$ $[s]$	0.089 ± 0.003	0.151 ± 0.006	0.130 ± 0.005
$\text{Mode}_{\text{l-n}}[t]$ $[s]$	0.008	0.010	0.010

(a) $\bar{t}_{\text{exp}}$, $\bar{t}_{\text{l-n}}$ and $\text{Mode}_{\text{l-n}}[t]$ for mDFO, SA and rSA in the range R_3.

The main conclusion, when considering the average times, is that for this particular problem mDFO is the fastest of the three algorithms, except in R_4, where mDFO and rSA show similar results. As expected rSA is better than SA in all ranges. The relative performance of the algorithms in the considered ranges, as measured by the ratios of the average times, is shown in table 14.

When looking at the modes of the respective log-normal distributions, we see a slightly different picture in the highest range – rSA is by far the best method, which is not so pronounced in the lower ranges. This is evident from the distribution of its running times, which has

R_4			
	mDFO	SA	rSA
$\bar{t}_{\text{exp}}$ $[s]$	3.27 ± 0.07	6.18 ± 0.12	3.08 ± 0.06
$\bar{t}_{\text{l-n}}$ $[s]$	4.17 ± 0.14	8.86 ± 0.38	4.14 ± 0.17
$\text{Mode}_{\text{l-n}}[t]$ $[s]$	0.36	0.32	0.21

(b) $\bar{t}_{\text{exp}}$, $\bar{t}_{\text{l-n}}$ and $\text{Mode}_{\text{l-n}}[t]$ for mDFO, SA and rSA in the range R_4.

R_5			
	mDFO	SA	rSA
$\bar{t}_{\text{exp}}$ $[s]$	123.8 ± 8.2	583.5 ± 38.6	206.6 ± 13.7
$\bar{t}_{\text{l-n}}$ $[s]$	154.5 ± 17.0	847.9 ± 141.4	421.3 ± 101.3
$\text{Mode}_{\text{l-n}}[t]$ $[s]$	15.1	16.5	1.1

(c) $\bar{t}_{\text{exp}}$, $\bar{t}_{\text{l-n}}$ and $\text{Mode}_{\text{l-n}}[t]$ for mDFO, SA and rSA in the range R_5.

Table 13: Expected values and modes for the respective distribution fits for all the algorithms in the tested ranges.

Exponential	R_3	R_4	R_5		Log-normal	R_3	R_4	R_5
$\bar{t}_{SA}/\bar{t}_{mDFO}$	1.6	1.9	4.7		$\bar{t}_{SA}/\bar{t}_{mDFO}$	1.7	2.1	5.5
$\bar{t}_{rSA}/\bar{t}_{mDFO}$	1.4	0.9	1.7		$\bar{t}_{rSA}/\bar{t}_{mDFO}$	1.5	1.0	2.7
$\bar{t}_{SA}/\bar{t}_{rSA}$	1.1	2.0	2.8		$\bar{t}_{SA}/\bar{t}_{rSA}$	1.1	2.1	2.0

Table 14: Relative performance of the algorithms in the tested ranges. For example, the exponential model fit in R_3 shows that our mDFO is 1.6 times faster on average than SA and 1.4 times faster than rSA. On the other hand, the log-normal fit states that mDFO is 1.7 times faster than SA and 1.5 times faster than rSA on average.

$\text{Mode}[t] = 1.1\,s$ in R_5, compared to $15.1\,s$ and $16.5\,s$ for mDFO and SA, respectively (table 13). In other words, most of the time rSA finds a solution notably more quickly than mDFO and SA.

Finally, we have performed two types of statistical tests over the sample data: parametric and non-parametric. First, we have considered a parametric test based on the Fisher distances between the respective distributions for the different algorithms. The result shows that mDFO and rSA are most similar, while they are significantly dissimilar than SA. This is also confirmed from the non-parametric posthoc Nemenyi test performed on the average times $\bar{t}$ of the three methods.

Now, where can we go from here? mDFO itself can be applied to various discrete optimisation problems, which are not purely theoretical. For this reason, we have given a generic version that does not depend on the particular fitness function. A next step could be to test it on some of the standard benchmark functions.

However, the final goal for the particular problem, that inspired the algorithm's development, is to be able to find solutions to (1.1) for various values of k in ranges above 10^{20} in reasonable time. As is well known, the solution density there is significantly reduced and this means that the search becomes very time consuming. We believe that some stochastic algorithm can be found that could produce solutions in acceptable time.

One way to reduce the search time is parallelisation. Generally, this can be done in many ways. As was already mentioned, the mode of the running times of rSA in the highest range

is peculiarly small. This suggests probably the simplest method to achieve some sort of parallelisation – run the same instance of the algorithm on many cores and just wait for the first one to finish. The probability of achieving a running time in R_5 with rSA of less than $1.5s$ for example is $P_{\mathrm{rSA}}(t \leq 1.5) \approx 0.0333$. Compare that to the probabilities for the same event with the other two algorithms: $P_{\mathrm{SA}}(t \leq 1.5) \approx 0.00096$ and $P_{\mathrm{mDFO}}(t \leq 1.5) \approx 0.00097$. This means that for the above parallelisation to work with mDFO, the algorithm needs to be improved.

Another line of investigation is to search for a better heuristic, not in the class of PSO or SA. In general, stochastic optimisation algorithms are highly specific to the problem and finding a good one isn't easy. A promising new development with regards to this is [18], which may enable us to delegate the task to AI.

A C code implementation

A.1 mDFO

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

double ff(int x, int y, int kk); //The fitness function.
int sort_and_break(double q[], int n); //A function which chooses the number of a best particle breaking ties
      randomly.



int main()
{
//Parameters:
        int kk = 2; //Right side of the diophantine equation.
        int lb[2] = {-pow(10,4),0}; //Lower bounds for the search space.
        int ub[2] = {0,pow(10,4)}; //Upper bounds for the search space.
        double thr = pow(10,-5); //Solution threshold. Used to decide when to check whether a solution has
             been found.
        int s = 50; //Number of particles.
        int dp = (s/5); //Number of particles that need to be at the best swarm best position in order to
             disperse all particles.

//Variables:
        time_t t0 = time(NULL); //Time used to seed the RNG.
        int pos[s][2]; //Particle positions.
        double fitf[s]; //Fitness functions for the respective particles.
        int nbrs[s][3]; //Neighbours of each particle.
        int sb[2],bsb[2],nb[s][2]; //Position of the swarm best particle, best swarm best position and
             position of the neighbours best particles.
        double sbfitf,bsbfitf,nbfitf[s]; //Fitness functions for the above.
        int dc; //Dispersion counter to decide when to disperse the particles.
        double nbrsfitf[3],r; //Auxiliary double variables.
        int i,j,zt; //Auxiliary integer variables.

//Seeding the RNG:
        srand48(t0);

//Calculating the neighbours of each particle:
        i = 0; while(i < s)
        {
                nbrs[i][0] = (s+i-1)%s;
                nbrs[i][1] = (s+i)%s;
                nbrs[i][2] = (s+i+1)%s;
                i = i+1;
        }

//Initialisation:
        i = 0; while(i < s)
        {
                pos[i][0] = lround(drand48()*(ub[0]-lb[0])+lb[0]);
                pos[i][1] = lround(drand48()*(ub[1]-lb[1])+lb[1]);
                fitf[i] = ff(pos[i][0],pos[i][1],kk);
                i = i+1;
        }

//Determining the swarm best and best swarm best particles:
        i = sort_and_break(fitf, s);
        sb[0] = pos[i][0]; sb[1] = pos[i][1]; sbfitf = fitf[i];
        bsb[0] = sb[0]; bsb[1] = sb[1]; bsbfitf = sbfitf;

        if(bsbfitf <= thr) //Solution check.
        {
                zt = lround(cbrt(kk-pow(bsb[0],3)-pow(bsb[1],3)));
                if(pow(bsb[0],3)+pow(bsb[1],3)+pow(zt,3) == kk)
                {
                        printf("{%d,%d,%d}\n",bsb[0],bsb[1],zt);
                        goto end;
                }
        }

//Calculating the best particle among the neighbours:
        i = 0; while(i < s)
        {
                nbrsfitf[0] = fitf[nbrs[i][0]];
                nbrsfitf[1] = fitf[nbrs[i][1]];
                nbrsfitf[2] = fitf[nbrs[i][2]];
                j = sort_and_break(nbrsfitf, 3);
                nb[i][0] = pos[(s+i+j-1)%s][0];
                nb[i][1] = pos[(s+i+j-1)%s][1];
                nbfitf[i] = fitf[(s+i+j-1)%s];
                i = i+1;
        }

//Iterations:
        while(1)
        {
//Position update:
                if(dc >= dp)
                {
                        i = 0; while(i < s)
                        {
                                j = 0; while(j < 2)
                                {
                                        r = drand48();
                                        pos[i][j] = lround(r*(ub[j]-lb[j])+lb[j]);
```

```
                                        j = j+1;
                                }
                                i = i+1;
                        }
                }
                else
                {
                        i = 0; while(i < s)
                        {
                                j = 0; while(j < 2)
                                {
                                        r = drand48();
                                        pos[i][j] = lround(nb[i][j]+(r/2.0)*(sb[j]+bsb[j]-2.0*pos[i][j]));
                                        j = j+1;
                                }
                                i = i+1;
                        }
                }

//Confinement and dispersion condition check:
                i = 0; dc = 0; while(i < s)
                {
                        if((pos[i][0] < lb[0]) || (pos[i][0] > ub[0]) || (pos[i][1] < lb[1]) || (pos[i][1] >
                             ub[1]))
                        {
                                pos[i][0] = lround(drand48()*(ub[0]-lb[0])+lb[0]);
                                pos[i][1] = lround(drand48()*(ub[1]-lb[1])+lb[1]);
                        }
                        fitf[i] = ff(pos[i][0],pos[i][1],kk);
                        if((pos[i][0] == bsb[0]) && (pos[i][1] == bsb[1])) { dc = dc+1; }
                        i = i+1;
                }

//Swarm best, best swarm best and neighbours best updates:
                i = sort_and_break(fitf, s);
                sb[0] = pos[i][0]; sb[1] = pos[i][1]; sbfitf = fitf[i];
                r = drand48();
                if(sbfitf < bsbfitf)
                {
                        bsb[0] = sb[0]; bsb[1] = sb[1]; bsbfitf = sbfitf;

                        if(bsbfitf <= thr) //Solution check.
                        {
                                zt = lround(cbrt(kk-pow(bsb[0],3)-pow(bsb[1],3)));
                                if(pow(bsb[0],3)+pow(bsb[1],3)+pow(zt,3) == kk)
                                {
                                        printf("{%d,%d,%d}\n",bsb[0],bsb[1],zt);
                                        goto end;
                                }
                        }

                }

                if((sbfitf > bsbfitf) && (r <= (1-((sbfitf-bsbfitf)/0.5))))
                {
                        bsb[0] = sb[0]; bsb[1] = sb[1]; bsbfitf = sbfitf;

                        if(bsbfitf <= thr) //Solution check.
                        {
                                zt = lround(cbrt(kk-pow(bsb[0],3)-pow(bsb[1],3)));
                                if(pow(bsb[0],3)+pow(bsb[1],3)+pow(zt,3) == kk)
                                {
                                        printf("{%d,%d,%d}\n",bsb[0],bsb[1],zt);
                                        goto end;
                                }
                        }

                }

//Calculating the best particle among the neighbours:
                i = 0; while(i < s)
                {
                        nbrsfitf[0] = fitf[nbrs[i][0]];
                        nbrsfitf[1] = fitf[nbrs[i][1]];
                        nbrsfitf[2] = fitf[nbrs[i][2]];
                        j = sort_and_break(nbrsfitf, 3);
                        nb[i][0] = pos[(s+i+j-1)%s][0];
                        nb[i][1] = pos[(s+i+j-1)%s][1];
                        nbfitf[i] = fitf[(s+i+j-1)%s];
                        i = i+1;
                }
        }

        end:

        return 0;
}



double ff(int x, int y, int kk)
{
        double z;

        z = fabs(cbrt(kk-pow(x,3)-pow(y,3))-lround(cbrt(kk-pow(x,3)-pow(y,3))));

        return z;
}

int sort_and_break(double q[], int n)
{
        int k,i;
        int c = 0;
```

```
        int b[n];
        double gv = q[0];

        i = 0; while(i < n) { b[i] = 0; i = i+1; }

        i = 1; while(i < n)
        {
                if(q[i] == gv)
                {
                        c = c+1;
                        b[c] = i;
                }
                if(q[i] < gv)
                {
                        gv = q[i];
                        c = 0;
                        b[c] = i;
                }
                i = i+1;
        }

        k = lround(drand48()*c);

        return b[k];
}
```

A.2 SA

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

double ff(int x,int y,int kk); //The energy function.
double temp(int m); //The cooling schedule (temperature).



int main()
{
//Parameters:
        int kk = 2; //Right side of the diophantine equation.
        int lbx = -pow(10,4); //Lower bound for the x coordinate of a state.
        int ubx = 0; //Upper bound for the x coordinate.
        int lby = 0; //Lower bound for the y coordinate of a state.
        int uby = pow(10,4); //Upper bound for the y coordinate.
        double thr = pow(10,-5); //Solution threshold. Used to decide when to check whether a solution has
             been found.

//Variables:
        time_t t0 = time(NULL); //Time used to seed the RNG.
        int x,y,xn,yn; //Coordinates of the current and new states.
        double z,zn; //Energy values for the above.
        int m; //Time.
        int a,b,c,d; //Variables used in the generation of a new state.
        double prob; //Probability for accepting a transition to the already generated state.
        int zt; //Auxiliary integer varialbe.
        double r; //Auxiliary double variable.

//Seeding the RNG:
        srand48(t0);

//Random initial state and its energy:
        x = lround(drand48()*(ubx-lbx)+lbx); y = lround(drand48()*(uby-lby)+lby);
        z = ff(x,y,kk);

        if(z <= thr) //Solution check.
        {
                zt = lround(cbrt(kk-pow(x,3)-pow(y,3)));
                if(pow(x,3)+pow(y,3)+pow(zt,3) == kk)
                {
                        printf("{%d,%d,%d}\n",x,y,zt);
                        goto end;
                }
        }

//Iterations:
        m = 1; while(1)
        {
//Confinement:
                a = -10; b = 10; c = -10; d = 10;
                if(x+a <= lbx) { a = lbx-x; }
                if(x+b >= ubx) { b = ubx-x; }
                if(y+c <= lby) { c = lby-y; }
                if(y+d >= uby) { d = uby-y; }

//State generation:
                roll:
                xn = x+lround(drand48()*(b-a)+a);
                yn = y+lround(drand48()*(d-c)+c);
                if((xn == x) && (yn == y)) { goto roll; }
                zn = ff(xn,yn,kk);

//State transition:
                if(zn <= z)
                {
                        x = xn;
                        y = yn;
                        z = zn;
                }
                else
```

```
                {
                        prob = exp((z-zn)/temp(m));
                        r = drand48();
                        if(r <= prob)
                        {
                                x = xn;
                                y = yn;
                                z = zn;
                        }
                }

                if(z <= thr) //Solution check.
                {
                        zt = lround(cbrt(kk-pow(x,3)-pow(y,3)));
                        if(pow(x,3)+pow(y,3)+pow(zt,3) == kk)
                        {
                                printf("{%d,%d,%d}\n",x,y,zt);
                                goto end;
                        }
                }
                m = m+1;
        }

        end:

        return 0;
}



double ff(int x,int y,int kk)
{
        double z;

        z = fabs(cbrt(kk-pow(x,3)-pow(y,3))-lround(cbrt(kk-pow(x,3)-pow(y,3))));

        return z;
}



double temp(int m)
{
        double z;

        z = 1.0/(log(m)+0.01);

        return z;
}
```

A.3 rSA

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

double ff(int x,int y,int kk); //The energy function.
double temp(int m); //The cooling schedule (temperature).



int main()
{
//Parameters:
        int kk = 2; //Right side of the diophantine equation.
        int lbx = -pow(10,4); //Lower bound for the x coordinate of a state.
        int ubx = 0; //Upper bound for the x coordinate.
        int lby = 0; //Lower bound for the y coordinate of a state.
        int uby = pow(10,4); //Upper bound for the y coordinate.
        double thr = pow(10,-5); //Solution threshold. Used to decide when to check whether a solution has
             been found.
        int rtm = 30; //Number of consecutive states with equal energies needed for a restart.

//Variables:
        time_t t0 = time(NULL); //Time used to seed the RNG.
        int xo,yo,x,y,xn,yn; //Coordinates of the old, current and new states.
        double zo,z,zn; //Energy values for the above.
        int m; //Time.
        int a,b,c,d; //Variables used in the generation of a new state.
        double prob; //Probability for accepting a transition to the already generated state.
        int rt; //Current number of consecutive states with equal energies.
        int zt; //Auxiliary integer variable.
        double r; //Auxiliary double variable.

//Seeding the RNG:
        srand48(t0);

        restart:

//Random initial state and its energy:
        rt = 1;
        x = lround(drand48()*(ubx-lbx)+lbx); y = lround(drand48()*(uby-lby)+lby);
        z = ff(x,y,kk);

        if(z <= thr) //Solution check.
        {
                zt = lround(cbrt(kk-pow(x,3)-pow(y,3)));
                if(pow(x,3)+pow(y,3)+pow(zt,3) == kk)
                {
                        printf("{%d,%d,%d}\n",x,y,zt);
```

```
                        goto end;
                }
        }

//Iterations:
        m = 1; while(1)
        {
//Confinement:
                a = -10; b = 10; c = -10; d = 10;
                if(x+a <= lbx) { a = lbx-x; }
                if(x+b >= ubx) { b = ubx-x; }
                if(y+c <= lby) { c = lby-y; }
                if(y+d >= uby) { d = uby-y; }

//State generation:
                roll:
                xn = x+lround(drand48()*(b-a)+a);
                yn = y+lround(drand48()*(d-c)+c);
                if((xn == x) && (yn == y)) { goto roll; }
                zn = ff(xn,yn,kk);

//State transition:
                xo = x;
                yo = y;
                zo = z;
                if(zn <= z)
                {
                        x = xn;
                        y = yn;
                        z = zn;
                }
                else
                {
                        prob = exp((z-zn)/temp(m));
                        r = drand48();
                        if(r <= prob)
                        {
                                x = xn;
                                y = yn;
                                z = zn;
                        }
                }

//Restart condition check:
                if(zo == z)
                {
                        rt = rt+1;
                        if(rt == rtm) { goto restart; }
                }
                else
                {
                        rt = 1;
                }

                if(z <= thr) //Solution check.
                {
                        zt = lround(cbrt(kk-pow(x,3)-pow(y,3)));
                        if(pow(x,3)+pow(y,3)+pow(zt,3) == kk)
                        {
                                printf("{%d,%d,%d}\n",x,y,zt);
                                goto end;
                        }
                }
                m = m+1;
        }

        end:

        return 0;
}


double ff(int x,int y,int kk)
{
        double z;

        z = fabs(cbrt(kk-pow(x,3)-pow(y,3))-lround(cbrt(kk-pow(x,3)-pow(y,3))));

        return z;
}


double temp(int m)
{
        double z;

        z = 1.0/(log(m)+0.01);

        return z;
}
```

References

[1] H. Davenport, On Waring's problem for cubes, Acta Mathematica, Vol. 71, (1939).

[2] D. R. Heath-Brown, The Density of Zeros of Forms for Which Weak Approximation Fails,

Mathematics of Computation, Vol. 59, Num. 200, 613-623 (1992).

[3] A. R. Booker, Cracking the Problem with 33, Research in Number Theory, 5:26 (2019).

[4] S. Abraham, S. Sanyal, M. Sanglikar, Particle swarm optimisation based Diophantine equation solver, International Journal of Bio-Inspired Computation, Vol.2, No. 2 (2010).

[5] Mohammad Majid Al-Rifaie, Dispersive Flies Optimisation, Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, ACSIS, Vol. 2, (2014).

[6] E. Cuevas, F. Fausto, A. González, New Advancements in Swarm Algorithms: Operators and Applications, Springer Nature Switzerland AG, 2020.

[7] M. Clerc, Standard Particle Swarm Optimisation, hal-00764996, (2012).

[8] D. Bertsimas, J. Tsitsiklis, Simulated Annealing, Statistical Science, Vol. 8, No. 1, 10-15 (1993).

[9] F. Mendivil, R. Shonkwiler, M. C. Spruill, Restarting Search Algorithms with Applications to Simulated Annealing, Advances in Applied Probability, Vol. 33, 242-259 (2001).

[10] Zhou XH, Gao S., Confidence intervals for the log-normal mean, Statistics in Medicine, VOL. 16, 783—790 (1997).

[11] C. R. Rao, Information and the accuracy attainable in the estimation of statistical parameters, Bulletin of the Calcutta Math. Soc. 37:81-91, 1945.

[12] S. Amari, H. Nagaoka, Methods of Information Geometry, Translations of mathematical monographs, (AMS, 2007).

[13] S.-i. Amari, Information Geometry and Its Applications. Springer Publishing Company, Incorporated, 1st ed., 2016.

[14] S. Amari, Differential-Geometrical Methods in Statistics. Lecture Notes in Statistics. Springer New York, 2012

[15] R. Frieden, R. A. Gatenby, Exploratory Data Analysis Using Fisher Information, Applied Mathematical Sciences, (Springer, London, 2010).

[16] S. I. R. Costa, S. A. Santos, J. E. Strapasson, Fisher information distance: A geometrical reading, Discrete Applied Mathematics Vol. 197, 2015.

[17] S. Taylor, Clustering Financial Return Distributions Using the Fisher Information Metric, Entropy 2019, 21, 110, 10.3390/e21020110.

[18] K. Li, J. Malik, Learning to Optimize, arXiv:1606.01885, (2016).