Visualizing Global Optimization Algorithms With A Function Derived From Darts

Max Caragozian

April 27, 2025

> *The tap-room fire is alight again and a new stock of darts laid in; serviceable and well-feathered, that fly true and will see us into another spring.*
>
> H. M., *"A Chiltern Autumn"* [M.29]

1 Introcudction

I bought a dartboard for my college house this past January, and darts quickly became a common pastime during what ended up being a long, snowy winter. I started reading into the mathematics of darts and came across the paper "A Statistician Plays Darts [TPT11]" by Tibshirani, Price, and Taylor. The paper quantifies how the optimal place to aim a dart relates to the skill of the thrower.

In the simplest model, the authors assume that a person throws with a symmetrical bivariate normal distribution. They estimate the function $\mu^*(\sigma)$ that takes in a standard deviation σ and outputs a point μ which maximizes the expected score of a dart thrown with distribution $\mathcal{N}(\mu, \sigma^2)$.

The authors found that a person with a very small standard deviation (that is, a very skilled player) should aim for the triple twenty (T20). That shouldn't be surprising as the T20 is the highest point value on the board. The optimal aiming location stays in the T20 until approximately $\sigma = 16.9$mm, where it jumps to the T19, before eventually moving towards the center of the board. (See Figure 5 in [TPT11].)

This project compares four different ways to replicate that result.

2 Integration Brute Force

Before trying to find $\mu^*(\sigma)$, I wrote code that would find the expected score for someone throwing a dart with a given random distribution. We define a *sector* as a continguous area of the dartboard with a single point value. Examples are

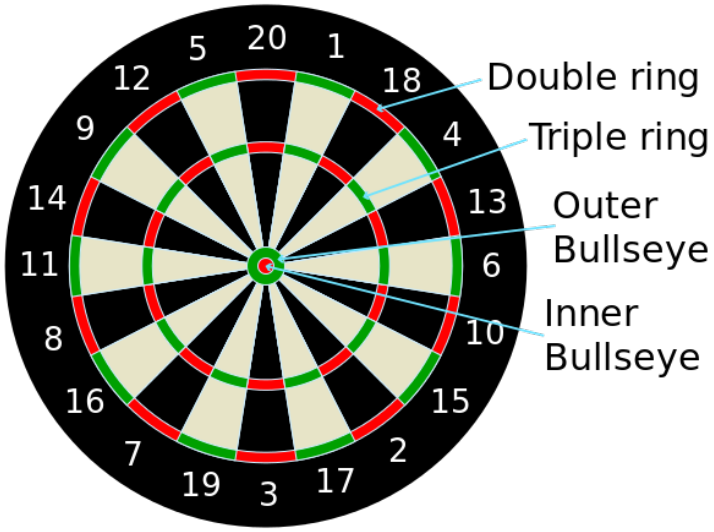

Figure 1: Standard dartboard with point values [Sta14]

the triple 20, the double bullseye, or the portion of single 19 that lies between the outer bullseye ring and the inner triple ring. Let S be the set of all sectors and X be a distribution with PDF f_x. To find the expected score of a distribution we compute:

$$E[score(X)] = \sum_{s \in S} \iint_s score_s \cdot f_x(x, y) dx dy. \tag{1}$$

I used SciPy's *nquad* function to perform the integration. Originally, I used the *dblquad* method, which is specific to double integral, but I ran into a rounding issues when f_x was very small. The *nquad* function allows finer control and fixed the issue.

Turning now to the specific case of symmetrical bivariate normal distributions, let us define the function that takes in a point μ and returns the expected score of a dart thrown with distribution $\mathcal{N}(\mu, \sigma^2)$ as

$$F(\mu|\sigma). \tag{2}$$

With the code implementation of Equation 1, I could calculate $F(\mu|\sigma)$ for a grid of possible μ values across the dartboard. I could then both plot the grid as a heatmap and return the maximum calculated value as a numerical approximation for $\mu^*(\sigma)$.

For each pixel in the heat map (i.e. each evaluation of $F(\mu|\sigma)$), I needed to compute 82 double integrals, and this method was painfully slow (see Section 5 for exact numbers).

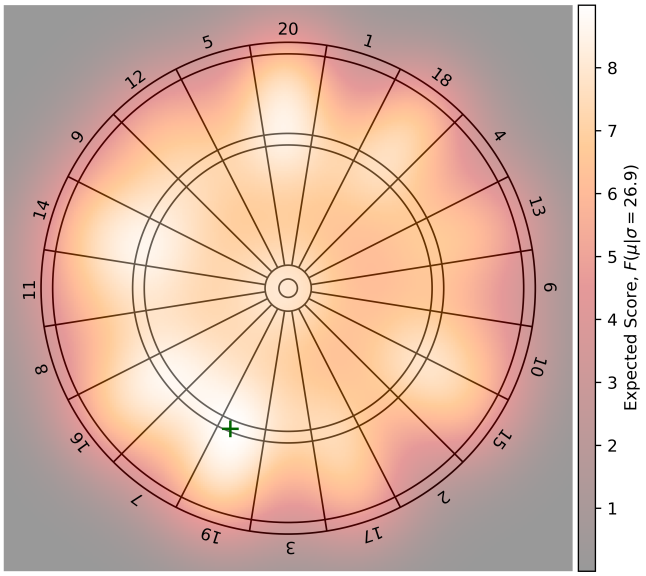

Figure 2: Heatmap for $\sigma = 26.9$. The green + is the location of μ^*

3 Basin-Hopping

I looked for faster ways to compute $\mu^*(\sigma)$. We can think about the computation as a global maximization problem for a function of function of two inputs (the x and y coordinates of μ), that is

$$\mu^*(\sigma) = \max[F(\mu|\sigma)] \tag{3}$$

By convention, global optimization algorithms find the *minimum* of a function, so in practice we compute

$$\min[-F(\mu|\sigma)]. \tag{4}$$

SciPy has bindings for several different global minimization algorithms. I ended up settling on a stochastic algorithm called *basin-hopping*.

Each iteration of basin-hopping starts with a point (the "currently accepted point"). The algorithm then takes a jump of a random size in a random direction. After landing at a new point, the algorithm moves "downhill" to a local minimum and compares the new local minimum with the currently accepted point. If the new minimum is lower than the previous one, it is always accepted. If it is not lower, there is a random process to decide whether or not

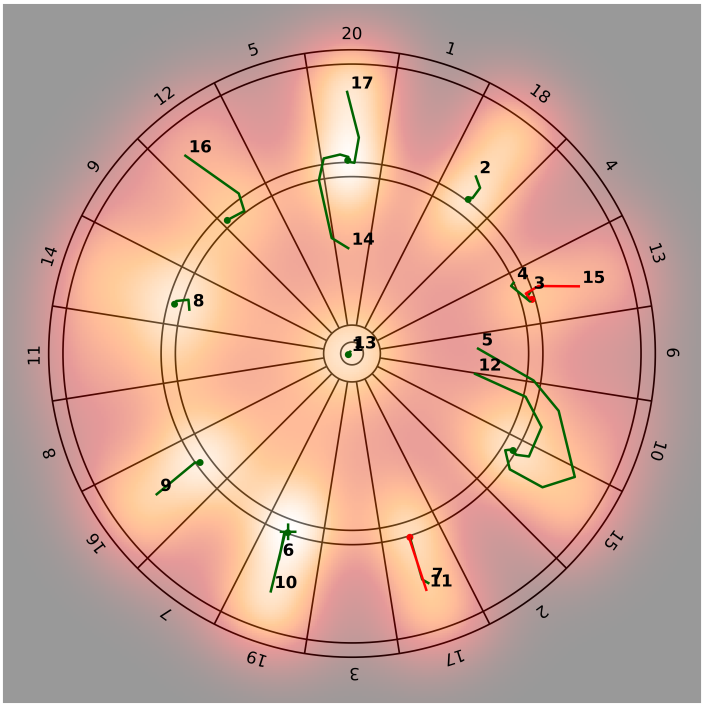

Figure 3: Each run of the local optimizer for a basin-hopping run to find $\mu^*(\sigma = 19mm)$. Points that were *accepted* by the algorithm are green while those that were *rejected* are red. The green + is the true location of μ^*.

to accept it, with lower minima being more likely to be accepted. The process then repeats, with another random perturbation, a local minimization, and a comparison with the most recently accepted point. Over time, random jumps get smaller, and the algorithm ends when a point has remained an unchallenged candidate for the global minimum for some number of cycles. The distribution of jump sizes, the local minimization algorithm, and the acceptance criteria for new points can all be controlled by parameters [Com].

We should note that basin-hopping is not guaranteed to find the global minimum. If the random jumps never find the right "basin", the algorithm will return the wrong answer.

Figure 3 shows a visualization for one run of the basin hopping algorithm. Each line represents one pass of the local optimizer, with the dots being the local minima.

While computing μ^* proved a good way to illustrate basin-hopping, basin-hopping was not a good way to compute μ^*. Besides not being guaranteed to find the correct solution, basin-hopping still relies on the slow process of integration. To be sure, it is still faster than brute force: the basin-hopping run

in Figure 3 needed only 134 evaluations of $F(\mu|\sigma)$, compared to 90,000 for a 300×300 heatmap. But with the added overhead of the algorithm, it still took time on the order of minutes to compute μ^*, and I wanted to find a better way.

4 Don't Let Ten Minutes of Reading Save You From Ten Hours of Coding

I reread "A Statistician Playes Darts [TPT11]" and realized the authors came up with a much faster way to compute $F(\mu|\sigma)$ that I missed the first time I read the paper. The authors derived that

$$F(\mu|\sigma) = (f_{0,\sigma^2} * score)(\mu) \tag{5}$$

where $*$ is the convolution between the PDF of $\mathcal{N}(0, \sigma^2)$ and the score value for each point on the dartboard [TPT11]. One can compute the convolution extremely quickly with an algorithm that uses a Fast Fourier Transform. Generating the data for a 300 by 300 heatmap such as that in Figure 2 took over an hour by integration, but SciPy's *fftconvolve* function gets the job done in seconds.

Tibshirani, Price, and Taylor also note that previous researchers have approximated the same results using Monte-Carlo [TPT11]. To round off my survey of methods for computing μ^*I implemented the Monte-Carlo method. It ended up being considerably faster than direct integration even when taking many samples for each function evaluation.

5 Method Comparison, Timing, and Final Discussion

Let us summarize the four methods in a table:

	Deterministic	**Stochastic**
Compute Whole Grid	Convolution, Integration	Monte Carlo
Compute Single Point	-	Basin-Hopping

As seen in Figure 4, none of my home-grown methods for computing μ^* performed better than the convolution algorithm derived by Tibshirani, Price, and Taylor. Not that I exhausted every option. For example, my basin-hopping code uses only direct integration for its function evaluations, but there is no reason why I could not have used Monte-Carlo, which would be significantly faster. My code would also work for distributions other than a bivariate normal, but I did not have time to pursue that.

The sheer number of paths I *couldn't* investigate hints at the real value in this project. $F(\mu|\sigma)$ is easy to understand but hard to calculate, making it a good test case for a wide variety of global optimization algorithms. It is also

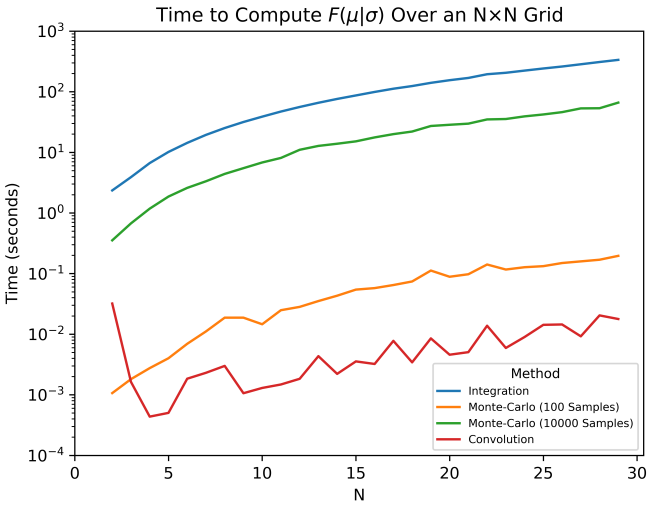

Figure 4: Speed comparison for the methods I tried. I have omitted basinhopping as it does not depend on a grid size.

a function of two inputs, which lends itself to visualizations and real world analogies like "moving downhill."

6 Github

The code for this project can be found here.

References

[Com] The Scipy Community. *scipy.optimize.basinhopping*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.basinhopping.html#scipy.optimize.basinhopping.

[M.29] H. M. "A Chiltern Autumn". In: *The Spectator* 143.5289 (1929). URL: https://books.google.com/books?id=mniapACANGEC.

[Sta14] Tijmen Stam. *Dartboard diagram.svg*. 2014. URL: https://commons.wikimedia.org/wiki/File:Dartboard_diagram.svg.

[TPT11] Ryan Tibshirani, Andrew Price, and Jonathan Taylor. "A Statistician Plays Darts". In: *Journal of the Royal Statistical Society Series A* 174 (2011), pp. 213–226. URL: https://www.stat.cmu.edu/~ryantibs/papers/darts.pdf.