

# Numeric Computations of Landau’s constant

Matija Derganc

matija.derganc@gmail.com

Faculty of Mathematics and Physics

University of Ljubljana

Jadranska cesta 21

1000 Ljubljana, Slovenia

Luka Urbanč

urbancluka3@gmail.com

Faculty of Mathematics and Physics

University of Ljubljana

Jadranska cesta 21

1000 Ljubljana, Slovenia

## Abstract

Landau’s constant, denoted by  $\lambda$ , arises naturally in complex analysis from the fact that entire nonconstant holomorphic functions are open maps. It is associated with the largest disc that can be contained in the image of any holomorphic function on the unit disc. Despite its theoretical significance, the exact value is unknown. The best known bounds are

$$\frac{1}{2} < \lambda \leq \frac{\Gamma(\frac{1}{3}) \Gamma(\frac{5}{6})}{\Gamma(\frac{1}{6})} = 0.54325 \dots$$

Unlike most mathematical constants such as  $\pi$  or  $e$ , for which we have efficient computation algorithms that yield millions of digits, no such algorithms have been implemented for Landau’s constant until now. We present an implementation of the algorithm proposed by Rettinger in Rust, which allows us to quickly compute Landau’s constant. It is organized around modular components that handle rational arithmetic, covering grids, and evaluation methods. We have also implemented a minimal dashboard that allows users to experiment with complex functions and provides a quick overview of the project’s theme. Rust was chosen for its strong type system, memory safety, and performance, making it well-suited for numerical computation. The modular approach ensures clarity of flow, ease of debugging, and adaptability for future refinements.

## Keywords

Landau’s Constant, Rust, Computation, Complex analysis, Algorithms, Euclidean distance transform

## 1 Introduction

Complex functions that are differentiable are called *holomorphic*. They possess many useful mathematical properties; for example, each holomorphic function is equal to its Taylor series. Another important theorem states that nonconstant holomorphic functions are *open*, which informally means that they map sets with area, such as the unit disc, to other sets with area (they cannot, for example, map a disc into a line, which has no area). A natural question that arises is how large a disc can fit inside the image over a given domain. Does there exist a lower bound on the size of the disc?

The answer is yes, and Landau’s constant determines this lower bound. More precisely, Landau’s constant  $\lambda$  is defined so that for any radius  $r > 0$  and any holomorphic function defined on the disc

$$K_r(z_0) := \{z \in \mathbb{C} : |z - z_0| < r\}$$

with  $f'(z_0) \neq 0$ , the image  $f(K_r(z_0))$  contains a disc of radius  $|f'(z_0)| \cdot r \cdot \lambda$ , where the constant  $\lambda > 0$  is independent of  $f$  [1]. If we restrict ourselves to the unit-disc domain (radius 1) and to *normalised* functions with  $f'(0) = 1$ , then Landau’s constant is

the largest number such that we can fit a disc of radius  $\lambda$  into the image of the unit disc, regardless of the function. We also denote by  $\lambda_f$  the radius of the largest disc that can fit inside the image of a specific function  $f$ , making Landau’s constant the infimum of  $\lambda_f$  as  $f$  ranges over all possible normalised functions. This is also how the implemented algorithm works: by checking  $\lambda_f$  for a sufficient number of carefully chosen functions, we can approximate the constant effectively.

## 2 Methodology

We present the algorithm we used to compute Landau’s constant. The project is structured into distinct modules. The first modules cover the tools used to implement the algorithm: dyadic complex numbers suitable for fast and accurate computation, holomorphic functions and their generation for accurate representation of functions, and covering grids used to measure sizes of complex sets. We then present the Euclidean distance transform (EDT) algorithm and the steps taken to improve performance, such as multithreading, as well as a brief summary of the implemented dashboard. Finally, we use these modules to implement the main algorithm and compute the constant effectively.

### 2.1 Dyadic types

The first step in the implementation was to ensure greater control over numerical accuracy during computation. To achieve this, we implemented dyadic numbers: numbers of the form

$$n \cdot 2^m$$

for integers  $n$  and  $m$ . By defining our own types, we gained precise control over the program’s numerical precision and ensured that we did not lose too much information. The `dyadic.rs` module defines operations such as addition, multiplication, and scaling, ensuring that numerical results are representable without hidden rounding error. This is because types like `f64` have a limited number of bits, which are shared between the exponent, mantissa, and sign. Our implementation addresses this by giving the exponent and mantissa their own allocations. This design choice was crucial for maintaining rigorous bounds. We then represented complex numbers as pairs of dyadics.

### 2.2 Holomorphic function representation and generation

We implemented the type `ComplexFunction`, which represents a function’s Taylor coefficients as well as its bounding sequence, which must be a convergent series. `ComplexFunction` objects were

obtained using the  $\Psi$  function proposed in [3], which generates a sequence of expansion coefficients from words.

Because the constant is defined as a lower bound taken over all possible functions, we require a systematic way to generate a sufficient number of them. For this purpose, we use words that encode the function's expansion coefficients. The alphabet consists of the digits 1, 2, 3, 4. We first split a word into multiple shorter words, which represent the different expansion coefficients. For the  $n$ -th coefficient, we select the letters in the word that occur at the positions of  $n$  in the  $t$ -sequence 0, 0, 1, 0, 1, 2, 0, 1, 2, 3,  $\dots$ . For the word 123412, this splits into the words 124, 31, 2, matching the positions of the numbers 0, 1, and 2 in the  $t$ -sequence.

We then use each of the shorter words to determine a specific coefficient for a function. In our case, this yields three expansion coefficients. For each one, we start with a complex square of a given size. Then, for each letter, we split the square into quarters and recursively move into the sub-square represented by the letter: 1 means NE, 2 NW, 3 SW, 4 SE (the Cartesian quadrants). When we run out of letters, we take the midpoint of the remaining square as the expansion coefficient.

For example, if we start with a  $2 \times 2$  square spanning  $\text{Re}(z) \in [-1, 1]$  and  $\text{Im}z \in [-1, 1]$  about the origin, and apply the first shorter word 124, the first letter moves us to the quadrant  $[0, 1] \times [0, i]$ . The letter 2 refines this to  $[0, \frac{1}{2}] \times [\frac{i}{2}, i]$ . The letter 4 then selects  $[\frac{1}{4}, \frac{1}{2}] \times [\frac{i}{2}, \frac{3i}{4}]$ . Having exhausted the letters, we take the midpoint of this final square as the first coefficient, in this case  $\frac{3}{8} + \frac{7}{8}i$ .

In this way, we can obtain a large number of different functions by generating words of given lengths, with longer words resulting in more functions [3].

## 2.3 Covering Grids

After using a word to create a unique holomorphic function, we evaluate the function on the unit disc. The disc is represented as a grid with spacing  $2^{-n}$  for a given natural number  $n$ . We then compute the value of the function at each grid point of the disc and store the results as an image represented by a vector of complex dyadics in Rust.

The next step is to turn this image—currently a vector of scattered complex numbers—into a compact grid in which each point is classified as inside or outside the image. These grids are called  $\epsilon$ -covering grids. The points in the grid are spaced  $\delta = \frac{\epsilon}{4}$  apart. A point  $x$  in the grid is inside the image if there exists a point  $z$  in the image such that the Euclidean distance  $|z - x|$  is smaller than  $\epsilon$ . By choosing  $\epsilon$  sufficiently small, we obtain an accurate grid representation of the image. For the grid dimensions, we locate the extreme inside points and then add an extra outer layer so that the inside is clearly surrounded by the complement. The grid is stored in a `GridBitmap` structure, which records its height, width, and a bitmap of inside points saved as a vector of 0s and 1s indicating whether a point is inside or outside. This structure provides the fastest computation speed and uses less memory than storing dyadic numbers directly.

## 2.4 Approximation on Grids using EDT

After we have created an  $\epsilon$ -covering grid  $G$  of an image, we can approximate the radius of the largest disc that can fit inside:

$$\max_{z \in G} \min_{x \in G^C} |z - x|.$$

In words, we compute, for each interior point, its distance to the boundary and then take the maximum of those distances.

This yields accurate results, but it must be done efficiently: a naïve loop that, for each interior point, scans all exterior points runs in quadratic time in the grid size. The grid itself scales quadratically with  $1/\epsilon$ , so this would imply a complexity of  $O(n^4)$ , which is not suitable for this project. Therefore, we implement the Euclidean distance transform (EDT) [2]. As input, we take a `GridBitmap` and operate directly on its data. As output, the algorithm assigns to each grid point its squared distance to the nearest point in the complement.

The algorithm first processes the rows. For each point, we consider the parabola

$$\phi_i(x) = (x - i)^2 + g(i),$$

where  $g(i)$  is a distance prior: it is set to 0 for points in the complement and to  $\infty$  for interior points. For each  $x$  in a row we then assign the squared distance to the nearest exterior point,

$$D(x) = \min_{0 \leq i \leq n-1} ((x - i)^2 + g(i)), \quad x \in \{0, 1, \dots, n-1\},$$

which corresponds to the lower envelope of  $\{\phi_i\}$ .

For example, if a row is  $[0, 1, 1, 1, 0]$  (with 1 indicating inside and 0 outside), the EDT result is  $[0, 1, 4, 1, 0]$ , since we record squared distances. A row with no exterior points, such as  $[1, 1, 1, 1, 1]$ , is transformed to  $[\infty, \infty, \infty, \infty, \infty]$  because  $g(i) = \infty$  for all positions in that row.

After processing each row, we run the algorithm on each column. The previously computed values mean interior points no longer have value  $\infty$  but differ according to their horizontal distances to the exterior, which allows the column pass to recover the true minimum distance to the outside for each interior point. Finally, we take the maximum over interior points.

The running time of EDT is linear per row and per column, so it is linear in the total grid size: two sweeps suffice for the whole grid. As a result, we can approximate the largest inscribed disc even on very large and dense grids containing millions of points.

## 2.5 Multithreading

One of Rust's standout features is its excellent support for parallel computing, which further accelerates our final computation. At the coarse level, independent words are evaluated in separate tasks, requiring no synchronization beyond aggregating the running minimum of  $\lambda_f$ . At the fine level, loops over grid rows and over image samples are safe to parallelize with data-parallel iterators because all shared state is read-only. The current binaries rely on asynchronous orchestration, and the code structure allows a drop-in replacement with a data-parallel library if desired. For this, we use Rust's `tokio` library, which supports asynchronous programming and provides excellent concurrency.

## 2.6 User Interface and Dashboard

Alongside the core algorithm, we implemented a minimal but functional dashboard that allows users to interactively explore holomorphic functions and visualize results. Through the dashboard, users can enter words that encode functions via the  $\Psi$  construction, after which the system automatically generates the corresponding holomorphic function and its coefficients. The image of the unit disc under the function is displayed using domain coloring, with the  $\epsilon$ -covering grid overlaid to illustrate the approximation. The largest inscribed disc found by the EDT algorithm is highlighted, providing immediate insight into  $\lambda_f$  for the chosen function. Users can also adjust computation settings such as grid resolution,  $\epsilon$  values, and word lengths to balance accuracy and performance. In addition, the dashboard presents summary metrics, including the computed lower bound  $l$  for each function, along with runtime statistics and memory consumption.

The dashboard also includes a short introduction to the theory behind Landau’s constant and the implementation proposed in [3], as well as a practical guide to reading domain-coloring plots. The main algorithm can also be run from the dashboard, providing users with information regarding the best possible approximation for a given maximum word length and accuracy. We have also added the option to evaluate a number of random words and display the one with  $\lambda_f$  closest to the constant’s precise value, which also allows us to evaluate the program’s performance on words of lengths 1000 or more—cases that are not computationally feasible when approximating the constant directly.

## 3 Results

We now have all of the required implementations for the final algorithm. We start by selecting a natural number  $n$ , which determines the accuracy of a quantity  $l$  satisfying

$$(1 - 2^{-n})\lambda \leq l \leq \lambda.$$

This is achieved by approximating the constant from below, using the methods described above on discs of smaller radius. We generate all words of length  $t$ , for a length  $t$  sufficient for the algorithm [3]. For each word, we construct the corresponding function as described in the function-generation section. We then evaluate the function on a disc of radius  $1 - 2^{-n}$ , create an  $\epsilon$ -covering grid for a theoretically suitable  $\epsilon$  (as explained in [3]), and approximate the radius of the largest inscribed disc using EDT. The parameter  $\epsilon$  must be small enough to ensure that the image of the function is accurately represented by the grid. For the domain spacing, we choose  $\frac{\epsilon}{4}$  so that grid generation leaves no holes. This yields an approximation  $l(\omega)$  for each word  $\omega$ , and we take

$$l = \min_{\omega} \{ l(\omega) \}.$$

Theoretical guarantees ensure that this value is correct and that the algorithm terminates in finite time.

The experiments were conducted on a MacBook Air M2, and computations were parallelized across available cores using the tokio runtime. In each case, we measured:

- the estimated value of  $\lambda$  obtained for a given depth  $n$  and word length  $t$ ,

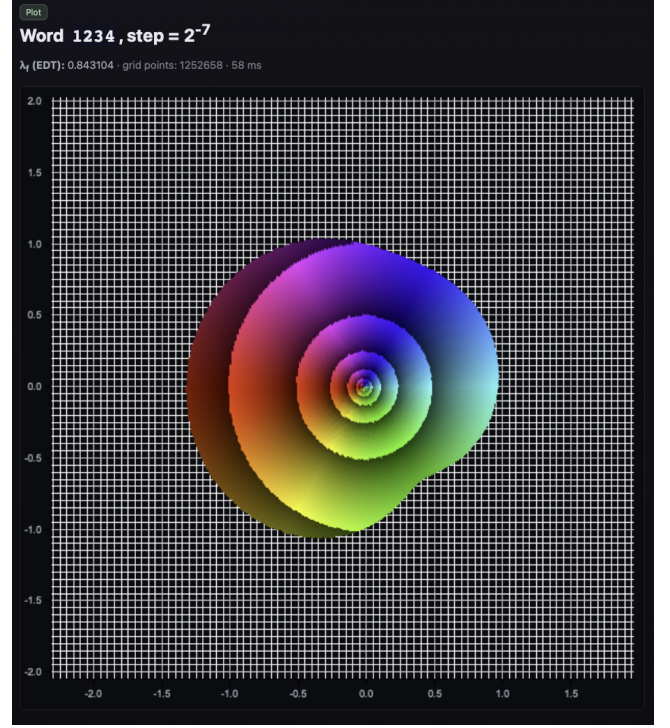


Figure 1: Image of the function generated from the word 1234.

- the execution time for grid creation, EDT, and function evaluation.

The results of the approximation are presented in Table 1. Note that these results were obtained with step = 1, as runs with larger steps could not be fully executed on our computer (see the discussion in the next section). Also, with increasing word length, we generate more functions and consequently a lower minimum, although all of these are valid approximations lying between  $\frac{\lambda}{2}$  and  $\lambda$ .

| Word Length | Computed $\lambda$ |
|-------------|--------------------|
| 4           | 0.4981760          |
| 5           | 0.4832330          |
| 6           | 0.4354750          |

Table 1: Approximations of  $\lambda$  obtained from words of increasing length using step = 1.

Another approach we tried was random word generation, which approximates the constant from above. When evaluating a random batch of words, we specify the word length, the number of words, and the disc accuracy  $n$  (so  $\epsilon = 2^{-n}$  and the domain spacing is  $\frac{\epsilon}{4}$ ). We then evaluate the corresponding functions on the whole unit disc to obtain an upper bound for  $\lambda$ . For words of length 1000 and accuracy  $n = 7$ , evaluating 1000 different words yielded a best  $\lambda_f$  of about 0.6682, which is fairly close to the actual value. The elapsed

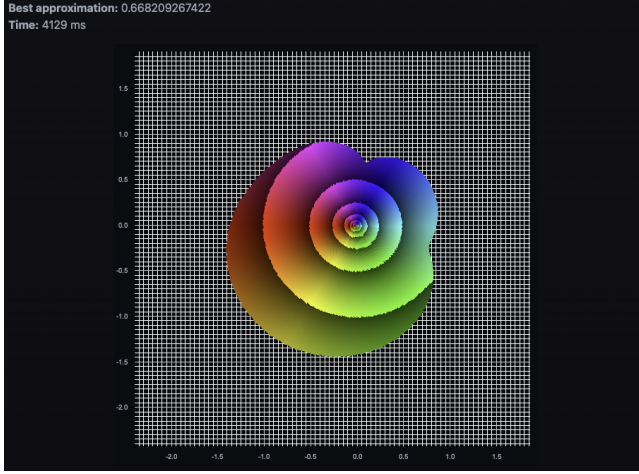


Figure 2: Best  $\lambda_f$  from 1000 random words of length 1000 with  $n = 7$ .

time for this computation was 4129 ms, leaving room to test larger random samples.

### 3.1 Performance

The performance benchmarks show that the EDT-based approximation scales linearly with grid size, enabling efficient computation on grids containing millions of points. For a selected function, the measured times were 100 ms for a grid with 4.8 million points, 471 ms for 20 million points, and 2119 ms for 80 million points. Horner’s algorithm, used for evaluating Taylor expansions of holomorphic functions, significantly reduced polynomial-evaluation overhead compared to naïve summation. Parallelization provided valuable speedups when distributing function evaluations across CPU cores.

### 3.2 Encountered Problems

There were multiple issues encountered in the algorithm. After mapping the disc-domain grid for each function corresponding to specific words, we had to cover the image with an  $\epsilon$ -covering grid. The parameter  $\epsilon$  must satisfy several requirements, as described in [3]. These bounds ensure that the grid gives an accurate representation of the image and are tied to bounds on second derivatives of the functions, which can grow large near the boundary. Increasing  $n$  (corresponding to a disc radius of  $1 - 2^{-n}$ ) increases these bounds by at least a factor of 8 over the previous step. For  $n = 1$ ,  $\epsilon$  is approximately  $2^{-12}$ ; consequently, the grid spacing  $\delta = \frac{\epsilon}{4}$  is about  $6 \cdot 10^{-5}$ . This implies a grid with roughly 1.1 billion points, already posing a significant computational and storage challenge.

For further steps, grid sizes increase to the point where they cannot be stored in system RAM, exceeding 30 GB. At present, this appears to be a practical boundary we cannot cross with our current equipment, as our personal computers lack the memory and compute capacity required for grid generation and the EDT on such large data structures.

**Data:** Word length  $t$ , result-accuracy parameter  $n$ , tolerance  $\epsilon$

**Result:** Approximation  $l$  of Landau’s constant  $\lambda$

```

1 initialization;
2 generate all words  $\omega$  of length  $t$ ;
3 foreach word  $\omega$  do
4   construct holomorphic function  $f_\omega$  using  $\Psi(\omega)$ ;
5   evaluate  $f_\omega$  on a disc of radius  $1 - 2^{-n}$ ;
6   create an  $\epsilon$ -covering grid  $G$  of the image  $f_\omega(D)$ ;
7   apply the Euclidean Distance Transform to  $G$  to
   compute the largest inscribed disc  $l(\omega)$ ;
8 end
9 set  $l = \min_\omega \{ l(\omega) \}$ ;
/* Theoretical guarantee:  $(1 - 2^{-n}) \lambda \leq l \leq \lambda$  */

```

Algorithm 1: General pipeline for approximating Landau’s constant

## 4 Discussion

The results are promising: we were able to lay the foundation for an accurate and efficient algorithm, given the theoretical guarantees of the framework [3]. Several key implementation choices contributed significantly to performance. First, using EDT to reduce the compute time for finding the largest inscribed radius greatly improved the algorithm’s performance. Manual implementations of number types ensured clarity and computational accuracy that would otherwise be difficult to guarantee and could lead to numerical errors, while algorithmic choices such as fast powering reduced time complexity. Although the work presented here is promising, the subject would benefit from additional research aimed at further reducing the time complexity of the proposed algorithm, especially regarding the bounds on  $\epsilon$  required by the final algorithm.

## 5 Conclusions

We are the first to have successfully implemented and improved the algorithm presented in [3], as far as possible given our current computational constraints. We achieved a best approximation of 0.4981760, which was obtained relatively quickly, leading us to believe it is possible to improve our work and develop an algorithm that would efficiently compute Landau’s constant to arbitrary precision.

## Acknowledgments

We would like to express our gratitude to Prof. Andrej Bauer for recommending this theme to us, as well as to Prof. Robert Rettinger for his extensive theoretical work, which formed the basis of our research. We would also like to thank Prof. Matija Pretnar, Gašper Žajdela, and Filip Koprivec for their helpful insight, advice, and support during our work.

## References

- [1] Edmund Landau. 1929. Über die Blochsche Konstante und zwei verwandte Weltkonstanten. *Mathematische Zeitschrift* 30, 1 (1929), 608–634.
- [2] Daniel Huttenlocher Pedro Felzenszwalb. 2004. Distance transforms of sampled functions. *Theory of computing* 8 (2004).
- [3] Robert Rettinger. 2012. On computable approximations of Landau’s constant. *Logical Methods in Computer Science* 8 (2012).