Design and Development of Global Optimization Methods with Applications

Doctoral Dissertation

Balázs L. Lévai

Supervised by
Balázs Bánhelyi, PhD

Doctoral School of Computer Science
Department of Computational Optimization
Institute of Informatics
University of Szeged

Szeged, 2019

Acknowledgement

First and foremost, I would like to express my thanks and gratitude to my supervisor, Balázs Bánhelyi, for helping and guiding me through my doctoral studies and for always having another interesting problem. I wish to thank professor Tibor Csendes for finding time to share his thoughts and opinion on my work, even when his hands were full, and for having interesting problems for me when my supervisor momentarily ran out of them. I also thank my co-authors, Endre Palatinus, László Pál, and Dániel Zombori for the great work together and outstanding professionalism.

Last but not least, I would like to thank my family and friends for their patience, understanding, and inexhaustible support that helped me through when I felt stuck in the trenches.

Preamble

Optimization is an inevitable part of our life not just on the individual level but on the larger scale of society, economy, and technology as well. New problems are formulated every day that need to be solved. Global optimization deals with situations when we are curious about the best option we have or get as close to it as it is computationally possible. Global optimization problems come in all shapes and sizes, therefore researchers developed a wide range of algorithms tailored to specific problem classes to be able to handle them efficiently or just to be able to have an approach on them at all, the repertoire of theoretical and practical tools has become extensive.

Mathematically formulating a simple natural task can be challenging, not to mention the selection of the most appropriate algorithm or theoretical concept for the solution. In my dissertation, I discuss the design and development process of solutions to global optimization problems to demonstrate the application of main concepts and approaches, furthermore I also present the improvement of an existing optimization algorithm.

In Chapter 1, we briefly describe the main classes of global optimization algorithms along some of the frequently used representatives, and we introduce interval arithmetic, a technique that apply later to implement mathematically reliable computation.

In Chapter 2, we build an algorithm to determine the optimal cover of arbitrary polygons by circles with fixed centers but variable radii that can be used to optimize e.g. the required power of different towers in a network that broadcasts terrestrial signal for telecommunication. We develop the complete solution in two major steps, first we introduce a method to verify

the correctness of given configuration of circles, then we build a deterministic optimization algorithm that systematically searches the space of allowed circle configurations until it finds a cover that is within the predefined margin of error.

In Chapter 3, we look for a way to design LED streetlights with a better quality light pattern, energy consumption, and complexity. We take a stochastic approach and create a genetic algorithm with a special, geometric crossover operation.

Chapter 4 is about a more theoretical topic, the chaotic trajectories of the forced damped pendulum. We search for initial states of this dynamic system from which it will execute a sequence of prescribed motions. We design a special objective function capable of expressing the difference between trajectories and the motion prescriptions, and we find initial states by optimizing this function with GLOBAL, a stochastic algorithm based on clustering.

In Chapter 5, our focus changes from optimization problems to optimization tools. We refine GLOBAL, the algorithm used in Chapter 4, and create a new implementation of it. Algorithmically, we improve the clustering strategy that is responsible for the identification of promising new starting points for local searches by better leveraging the clustering information and avoiding the initiation of most likely unnecessary local searches. In the second part of the chapter, we present the new JAVA implementation whose modular structure allows users to replace the default clustering and local searching logic of GLOBAL if they can take advantage of the characteristics of a given problem with their own, custom algorithms.

Contents

Chapter 1

Introduction

1.1 Algorithm classification

Global optimization is a vast field of expertise and the range of applications might be even more numerous. In its most general form, it aims to find the globally best solution of models that are expected or proven to have multiple local optima. We are going to discuss the topic using the following formal but not too strict definition.

Definition 1.1.1. *A global optimization problem is finding an element x^* of a search space $P \subseteq \mathbb{R}^n$, or a subset S of P determined by constraints, where a given real-valued objective function $f(x)$ has its global minimum value, formally $\forall x \in P$ (or S) $f(x^*) \leq f(x) : \mathbb{R}^n \to \mathbb{R}$ holds.*

Based on the attributes of the objective function, like continuity, differentiability, or dimension, and the attributes of the search space, like the presence of equality or inequality constraints, particular optimization methods better fit a problem than others. The goal of this dissertation is to present and demonstrate how we can approach global optimization problems and build solutions for them using different types of approaches. Algorithms can be separated into classes depending on multiple attributes. We chose to categorize them based on the application of random variables into three main classes, deterministic, stochastic, and hybrid algorithms as we illustrated in Figure 1.1.

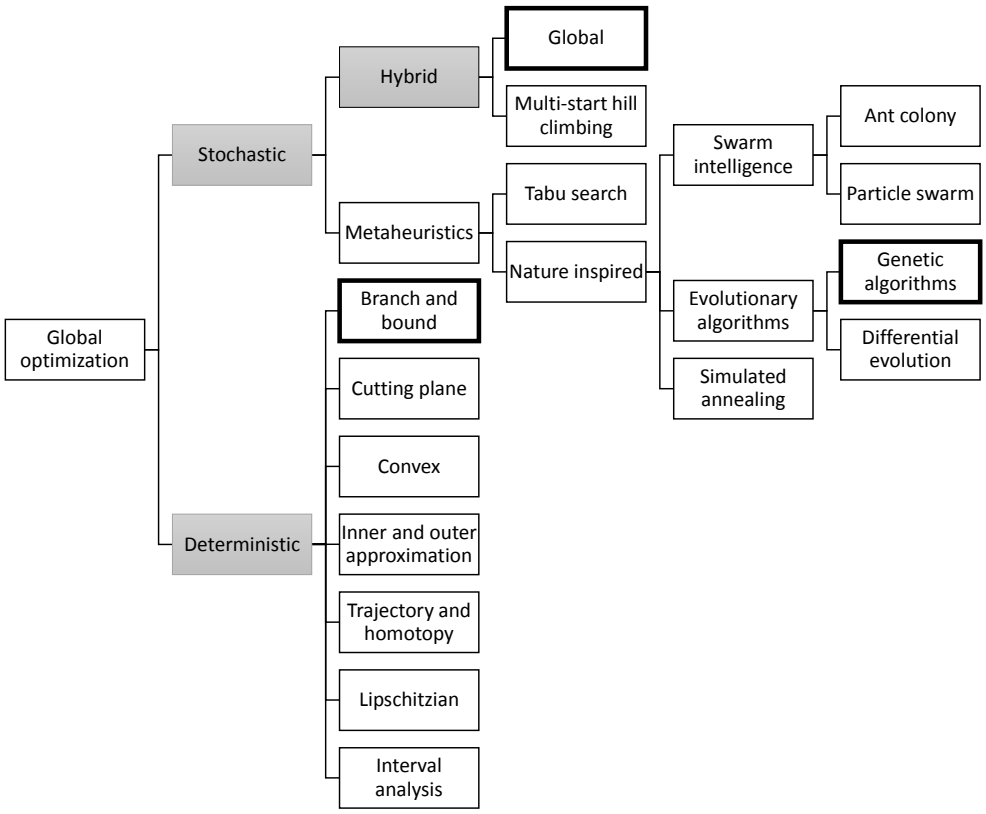


Figure 1.1: The informal classification of global optimization methods with some of the major categories and well-known examples. Grey rectangles denote the main subcategories, the rectangles with thick black contour are the algorithms we apply in our dissertation.

1.2 Deterministic algorithms

Deterministic methods [26] use no random variables and often provide theoretical guarantees at a predefined level of tolerance about the precision of found optima, the main reasons that we use such algorithms. According to Neumaier's classification of global optimization methods in the paper [57], this guarantee can be the completeness or, moving one step further, the rigorousness of the method.

Completeness means that the algorithm would certainly find a global

minimum if it had an indefinitely long run time and error-free numerical operations, and such approaches practically provide approximations of the global minimizer after a finite time within a predefined tolerance. Rigorous methods offer even more. They find global minima within a given tolerance even in the presence of rounding errors with the exception of near-degenerate cases depending on the exact value of tolerance. Rigorousness is achieved by replacing floating-point operations with interval arithmetic, a technique we will present in more detail later in the chapter.

Deterministic algorithms are mainly applicable when our problem has a given, well-defined structure that we can take advantage of. Naive approaches, like the complete, exhaustive search for certain combinatorial problems, can only work in the simplest cases.

Linear programming and integer programming in operations research are classic and extensively studied fields of global optimization. A lot of deterministic solvers have been developed over the decades, like the simplex algorithm [20] and the interior point methods [69], that are able to efficiently solve problems of even hundreds of variables.

A general and widely used optimization concept is the branch-and-bound strategy [41], that iteratively partitions the search space and excludes the partitions from the search that would not provide better solutions than the best solution that we found so far, based on some a prior knowledge about the problem space. We will have a detailed discussion about the branch-and-bound strategy in Chapter 2.

1.3 Stochastic algorithms

Stochastic methods [89] excel when we face complex problems that scale exponentially resulting in an exceptionally high-dimensional objective function or search space what deterministic methods can only optimize at an unacceptably slow pace or cannot handle at all. Stochastic methods rely on random variables, hence the name, and they provide asymptotic or no guarantees about the quality of found optima.

The Monte Carlo simulation [49] is a frequently used, purely statistical approach, a simulation technique that assigns scores to different solution candidates, the elements of the search space. First, it randomly generates samples from a predefined probability distribution depending on the task at hand that represent the possible states of the environment of the candidates, then the algorithm evaluates the candidates against the generated random samples and aggregates the results to provide scores as a final step.

Metaheuristics form a large class of global optimization algorithms that use decisions based on random variables during the global search, and their inventors often drew inspiration and concepts from other natural sciences, like chemistry or biology. This can mean the application of some probability distribution for generating samples or selecting the neighbor or direction that the search moves towards. These algorithms are often modified versions of local search methods or show some resemblance to certain deterministic techniques where the main purpose of the addition of random variables is to avoid getting stuck in local optima. A few notable metaheuristics are the tabu search, the simulated annealing, the ant colony optimization, and the genetic algorithms.

The tabu search [29] and the simulated annealing [83] can be considered the stochastic variants of hill climbing. The tabu search temporarily excludes already visited points from the search for short-term using a so-called tabu list, for medium-term by bias rules favoring the promising neighborhoods, for long-term using diversification rules that always drive the search towards new areas, or using a mix of these three approaches. The simulated annealing randomly generate a neighbor in each iteration and allows the search to step into inferior neighbors with a continuously decreasing probability during the optimization.

Ant colony optimization [22] is a swarm intelligence method for problems that can be modeled as optimal path finding in a graph. This technique simulates ants that walk in the graph laying pheromones in their wake, incentives for other ants to choose the same route instead of other ones. These pheromones evaporate in time, thus shorter paths, that are more likely to be visited by more ants, will have a greater pheromone density. In the end of

simulation, the paths that the ants follow will converge to a single path.

The methodology we will use in Chapter 3 is the concept of genetic, or evolutionary algorithms [23, 30, 44] that simulate another biological phenomenon, the natural selection. This technique models the search space as an ever-changing population of candidate solutions. In each iteration, the so-called genetic operators, that mimic the genetic changes of crossover and mutation, alter candidates or create new ones from existing ones while other operators delete candidates from the population depending on their fitness value, the objective function in this context. Candidates are selected for genetic operations and deletion by stochastic methods that favor the candidates having a greater fitness value.

1.4 Hybrid algorithms

Hybrid methods can be defined as the global extension of not necessarily deterministic local optimization algorithms that start local searches from stochastically produced starting points [81]. Hybrid algorithms try to combine the scalability of stochastic approaches and the relatively fast optimum improvements of local search algorithms like GLOBAL [15], that we are going to discuss in Chapter 5 and present an application of the algorithm in Chapter 4. These approaches are stochastic methods, but we decided to treat them as a separate category due to their conceptual difference as you follow our line of thought in Figure 1.1. Papers often call the combination or chaining of two or more algorithms a hybrid optimization solution [76] like when they run a stochastic method and start another one from the result of the first algorithm to further refine the final solution. Such loosely coupled algorithm pipelines do not fall into the hybrid category in our interpretation.

The simplest hybrid method is the multi-start hill climbing, also called as random restart hill climbing, that simply reruns the hill climbing algorithm multiple times from randomly generated points. It is easy to implement, even as a multithreaded search method, and the actual direction and step selection strategy can easily be switched between problems to take advantage

of any special property.

Genetic algorithms are often combined with local search methods to replace or extend the ordinary concepts of genetic operators like these in the papers [50, 66]. There are algorithms that are built the other way around borrowing the concept of genetic operators to introduce new types of search steps like the modified gravitational search algorithm in the paper [88]. These approaches do not fall under our hybrid algorithm definition either.

1.5 Interval arithmetic

Rigorous methods must provide results with mathematical rigor. A solution to counter the rounding error is to replace floating point arithmetic with interval arithmetic in computations. Instead of real numbers, interval arithmetic uses compact, closed intervals with borders that are computer representable and a redefined set of arithmetic operators. The application of interval arithmetic ensures the correctness of numeric results and forms the basis to provide computer aided proofs of validity.

Definition 1.5.1. *A one dimensional, compact interval, or simply an interval, is spanned by two real numbers:*

$$X = [\underline{X}, \overline{X}] = \{x \in \mathbb{R} : \underline{X} \leq x \leq \overline{X}\},$$

where $\underline{X}$ is the lower bound and $\overline{X}$ is the upper bound of the interval.

Definition 1.5.2. *An n dimensional interval is a vector*

$$X = (X_1, X_2, \ldots, X_n),$$

where every X_i $(i = 1, \ldots, n)$ is a one dimensional interval. Multidimensional intervals are also called boxes.

Notation 1.5.3. *Let $\mathbb{I}$ be the set of compact intervals over $\mathbb{R}$ and $\mathbb{I}^n$ be the set of n-dimensional intervals.*

Definition 1.5.4. *The width of a one dimensional interval is:*

$$\text{width}(X) = \overline{X} - \underline{X}.$$

The concept can naturally be generalized to n dimensions:

$$\text{maximal width(X)} = \max_{\text{i=1,...,n}} (\overline{\text{X}}_\text{i} - \underline{\text{X}}_\text{i}).$$

Because we will not use any other kind of width concept in the dissertation, we are going to simply refer to the extended concept as *width* instead of *maximal width.*

Sometimes, we discuss sets of n-dimensional points, but for special purposes, it is advantageous to use the bounding interval of these sets.

Definition 1.5.5. *Let $[S]$ be the n-dimensional bounding interval of the set of n-dimensional points S which fulfills that*

$$\forall s \in S \; \forall i \in \{1, 2, \ldots, n\} \; s_i \in [S]_i$$

and $\text{width}\left([S]_i\right)$ *is minimal for all $i = 1, 2, \ldots, n$.*

Interval computations use intervals instead of real numbers, therefore we have to extend the well-known arithmetical operators to intervals.

Definition 1.5.6. *Let Ω be the set of the elementary real operators. These operators can be extended to be interval operators by*

$$A \circ B = \{a \circ b : a \in A \text{ and } b \in B\},$$

where $A, B \in \mathbb{I}, \circ \in \Omega$.

By the above definition, computing the result of any interval operation would require an infinite number of execution of real operations when both operands have non zero width. Fortunately, a finite number of operations are sufficient in certain cases if we take advantage of the monotonicity of the operators. More information is available about interval arithmetic in [2, 3]. During our work, we used the *C-XSC* [19] and *PROFILBIAS* [42] libraries for implementation.

Chapter 2

Circle covering with fixed centers

Circle packing and circle covering are two sides of the same coin, they form one of the many dual problem pairs in mathematics.

In circle packing, we place circles without overlapping in a given set, most of the time in some special geometrical shape like a circle, square, or triangle, in a way that the circles can be tangent. The goal is to find the densest packing, an arrangement of the circles which covers the largest portion of the given set. This problem can be generalized into higher dimensions, it is called sphere packing. Researchers have intensively studied the field in the 20th century. Special variants of this problem, for example packing equal circles in a unit square, have attracted a fair share of attention of the scientific community even in the last decades, for example see the papers [43, 82].

The circle covering turns the circle packing problem inside out. We place overlapping circles to entirely cover a given set aiming to find the thinnest covering, an arrangement of circles whose sum of areas is minimal. Circle covering turned out to be more difficult than circle packing. There are less publications, and while we know the proved optimal configuration of circles for a wide range of packing problems, papers about covering usually present improvements on the best known boundaries for specific problems or prove

the optimality of a particular configuration of circles, but rarely do both, often due to the application of unreliable computation that provides numerical results of questionable validity and prevents any proof of optimality like in [60]. A considerable selection of papers about this topic is available online [27].

Although some special packing and covering problems interest researchers for mundane scientific curiosity, there are a lot of practical applications for real situations. Packing solutions provide optimal arrangements for packaging cylinder shaped containers in boxes, like paint canisters, or food cans, maximizing the amount of transportable material in one run. Covering solutions play a role in designing tower arrays for various purposes ranging from broadcasting radio and television signals to deploying radar defense systems as the region covered by a single station can be considered more or less of circular shape. At the scale of covering whole states, a solution better by only 1 percent can save a significant amount of construction and operational costs.

In this chapter, we construct a deterministic algorithm applying rigorous computation and optimization strategies. The algorithm will compute the optimal solution of special circle covering problem within an arbitrary but given precision, the cover of polygons with circles of fixed centers while the circles do not have to be necessarily congruent. Optimality of a covering means that it is the thinnest covering in the sense that we already mentioned above. Similar problems have already been discussed in the paper [21].

We are going to design the whole algorithm in two steps. First, we create a branch-and-bound method that is capable of deciding if a particular set of circles completely covers the target polygon. Then, we construct a second branch-and-bound algorithm responsible for the global optimization that systematically searches for the optimal configuration of radii. We use interval arithmetic instead of floating point arithmetic in all computation because we aim total correctness. We are going to prove the correctness of each algorithm that will imply the correctness of any numerical result they provide.

Some of the theoretical and algorithmic techniques in the following secti-

ons have already been successfully applied in the field of circle covering (e.g. [61]).

We are going to demonstrate the operation of the optimization approach and the quality of calculated results on either artificial test problems and real world scenarios both highlighting key aspects of our solution. We published the theoretical and numerical results of the chapter with my co-authors, Balázs Bánhelyi, and Endre Palatinus in the paper [12].

My supervisor, Balázs Bánhelyi has been conducting research about verified computer assisted proofs of mathematical problems using interval arithmetic since the start of his doctoral studies. He suggested to turn my focus on this sub-field of circle covering that became the topic of this chapter. Balázs Bánhelyi regularly reviewed the formulation and description of the theoretical background of the problem, and also introduced me to the theoretical results of Tibor Csendes that provided an analogy to prove the correctness of the cover verifier algorithm. Beside that, structuring, formulating, and proving the lemmas and theorems about the algorithmic correctness, and the development of the cover verifier and optimization algorithms in this chapter are my results. Implementing both algorithms and calculating the numerical results were the indivisible contribution of Balázs Bánhelyi, Endre Palatinus and myself.

2.1 Definitions

We consider a special circle covering problem. We assume that the centers of the circles are fixed, only the radii are allowed to be changed. This section introduces several definitions and notations that are necessary to properly discuss the algorithms and their proofs of correctness.

Definition 2.1.1. *$O_i = (x_i, y_i, r_i)$ denotes the open circle i, where x_i, y_i are the coordinates of its center and r_i is its radius.*

Definition 2.1.2. *$\mathcal{O} = \{O_i : i = 1, 2, \ldots, n\}$ is a set of n arbitrary circles given as it is in the Definition 2.1.1.*

Definition 2.1.3. *The configuration $r_{\mathcal{O}}$ belonging to the set of circles $\mathcal{O}$ is the vector $(r_1, r_2, \ldots, r_n)$, where r_i is the radius of $O_i \in \mathcal{O}$.*

Definition 2.1.4. *R refers the target region we would like to cover that is closed, bounded, and connected.*

R is a square in most cases, not just in covering but also in packing, although the cover of rectangles [33], circles [32], regular triangles [60], and other shapes have been studied as well.

After clarifying our objects of interest, we define when we consider a set of circles covering a two-dimensional shape.

Definition 2.1.5. *We say that the configuration $r_{\mathcal{O}}$ of the circle set $\mathcal{O}$ is good regarding the region R, that means it covers R, if and only if the following condition is satisfied:*

$$\forall (x, y) \in R \; \exists \; O_i \in \mathcal{O} : \; d((x, y), (x_i, y_i)) = \sqrt{(x - x_i)^2 + (y - y_i)^2} < r_i,$$

where function $d(.,.)$ is the Euclidean distance. Otherwise, we say that the configuration $r_{\mathcal{O}}$ is bad.

It is important to emphasize that the distance of a point and a center of circle has to be strictly lower and not lower or equal than the radius because the circles are open that is required for the correctness of our solution that we discuss in detail later in the this chapter.

We are going to need to handle the concept of goodness at the level of points, thus we derive a goodness property for points from Definition 2.1.5.

Definition 2.1.6. *The point $p = (x, y)$ of the region R has the property $\mathcal{P}$ regarding the configuration $r_{\mathcal{O}}$ of a circle set $\mathcal{O}$ if and only if at least one of the circles of $\mathcal{O}$ covers p. Formally:*

$$\exists \; O_i \in \mathcal{O} : \; d((x, y), (x_i, y_i)) = \sqrt{(x - x_i)^2 + (y - y_i)^2} < r_i,$$

where function $d(.,.)$ is the Euclidean distance.

This property can be extended from points to two-dimensional intervals naturally.

Definition 2.1.7. *If an interval $I \in \mathbb{I}^2$ satisfies that $\forall p \in I$ has the property $\mathcal{P}$ regarding a given $r_{\mathcal{O}}$, then we say that I also have the property $\mathcal{P}$.*

Badness of two-dimensional intervals are extended implicitly through Definitions 2.1.6 and 2.1.7.

Henceforth in this chapter, we are going to refer the concept of goodness and badness in the sense of Definitions 2.1.5, 2.1.6, and 2.1.7 when we are discussing the goodness of configurations, points, or intervals.

The key algorithms including the global optimization algorithm itself use the branch-and-bound optimization strategy. The basic idea is to represent the search space of candidate solutions as a rooted tree where the root represents the whole search space. A branch-and-bound algorithm systematically constructs and explores the tree using a branching and a bounding method, hence the name. The former creates a partition for any tree node while the latter calculates a lower bound of the objective function for them. This strategy is different from the brute-force exhaustive searches in that it does not partition further, or closes in other words, tree leaves whose lower bound returned by the bounding method is greater than or equal to the best objective function value of a known candidate solution. You can see the general, problem agnostic description of the branch-and-bound optimization strategy in Algorithm 2.1.

For a detailed and comprehensive analysis of this algorithmic concept including the proof of correctness, see the book [41]. The paper of N. V. Shilov [74] also provides a good analysis but it tackles these algorithms from a different angle using a formal approach.

ALGORITHM 2.1. Branch-and-bound

```
1: input solution-set: search-space
2: input bounding-method: bound, branching-method: branch
3: input objective-function: F
4: optional-input heuristic: h
5: output float: optimum-value, point: optimum-place

6: if h = null then
7:     optimum-place := null
8:     optimum-value := ∞
9: else
10:    optimum-place := h(from: search-space)
11:    optimum-value := F(optimum-place)
12: end if
13: queue := create-queue(type: solution-set, value: empty, strategy: LIFO)
14: push(value: search-space, into: queue)
15: while size-of(que) > 0 do
16:    node := pop(from: queue)
17:    if size-of(node) = 1 then
18:        if optimum-value > F(node) then
19:            optimum-place := node, optimum-value := F(node)
20:        end if
21:    else
22:        if optimum-value > bound(value: node) then
23:            for all solution-set: part in branch(from: node) do
24:                push(value: part, into: queue)
25:            end for
26:        end if
27:    end if
28: end while
29: return optimum-place, optimum-value
```

2.2 Cover verification

The first step towards constructing the optimization algorithm is to decide whether a particular configuration is good or bad. The branch-and-bound

ALGORITHM 2.2. Verify-goodness

```
1: input region: R
2: input configuration: r_O
3: input precision: ε
4: output interval: counter-example
5: output boolean: goodness

6: I := create-interval(bounding: R)
7: stack := create-queue(type: interval, value: empty, strategy: FIFO)
8: push(value: I, into: stack)
9: while size-of(stack) > 0 do
10:     V := pop(from: stack)
11:     if intersect(polygon: R, interval: V) = true and
            verify-P(configuration: r_O, interval: V) = false then
12:         if width-of(V) < ε then
13:             return goodness := false, counter-example := V
14:         else
15:             left, right := cut(value: V, into: halves,
                    along: largest-dimension)
16:             push(value: left, into: stack), push(value: right, into: stack)
17:         end if
18:     end if
19: end while
20: return goodness := true, counter-example := null
```

algorithm called *Verify-goodness* executes this check, see Algorithm 2.2.

If a set of circles $\mathcal{O}$ covers all points of a region, then it trivially covers the region as well. More formally, if all points of the region R have the property $\mathcal{P}$ regarding $r_{\mathcal{O}}$, this equals to that $r_{\mathcal{O}}$ is good. Therefore we check the property $\mathcal{P}$ point by point to evaluate the goodness of configurations. The algorithm starts from the bounding interval of R, that is allowed to be larger than R, and it is not have to be necessarily the convex hull of R. The algorithm systematically partitions the intervals, cutting them to halves, until a counterexample emerges to the cover, or each generated sub-interval, which overlaps with R, has the property $\mathcal{P}$. An interval is a counterexample if and only if its width is less than the predefined value of the precision variable

ε, it overlaps with R, and it does not have the property $\mathcal{P}$. *Verify-goodness* branches based on two key properties of intervals, the possible overlap with R, and having the property $\mathcal{P}$.

The algorithm *Intersect*, see Algorithm 2.3, handles the question of overlapping. Basically, it can be considered as the interval extension of Shimrat's algorithm, *Point in Polygon* [75], since it uses the same idea. Let the region P be a polygon and V be a two-dimensional interval. All we need to do is to count how many side sequences of P are left of V. If the number is odd, then P and V overlap. If it is even, then the two objects are disjoint. This is a simple trick commonly applied in computer graphics to check whether some object contains partially or entirely another one.

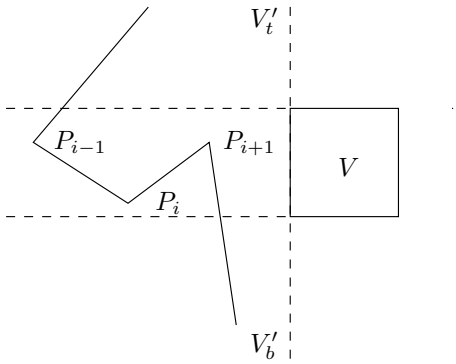


Figure 2.1: Illustration of the special regions used in Algorithm 2.3.

The algorithm starts with testing two straightforward conditions illustrated in Figure 2.1. First, P and V clearly overlap if any sides of P and V overlap. This can easily be checked. Second, if there is no such side, and the polygon is located entirely left and above of V (in the region V_t'), or left and below of V (in the region V_b'), then V and the polygon must be also disjoint.

After the initial tests, we start counting the side sequences that are left to V. We determine where the sequences start, where they end, and which direction the currently followed sequence is heading to. There are some algorithmic challenges in the concrete implementation as a two-dimensional interval might have inner and boundary points with respect to a particular

side, and an interval can even contain a whole sequence of polygon sides. Such situations and corner cases cannot occur with the classic *Point in Polygon* algorithm.

ALGORITHM 2.3. Intersect

```
 1: input polygon: P
 2: input interval: I
 3: output boolean: intersection

 4: if any-of(values: sides-of(P), holds: Intersect(with: I)) then
 5:     return intersection := true
 6: end if
 7: V'_b := create-interval(x1: −∞, x2: x1-of(I), y1: −∞, y2: y1-of(I))
 8: V'_t := create-interval(x1: −∞, x2: x1-of(I), y1: y2-of(I), y2: ∞)
 9: if contains(value: P, in: V'_b) or contains(value: P, in: V'_t) then
10:     return intersection := false
11: end if
12: sides := create-list(type: polygon-side, values: sides-of(P))
13: side := find-first(in: sides, holds: intersect(with: V'_t ∪ V'_b))
14: m := 0, state := out, visited := 1
15: while visited ≤ n do
16:     if intersect(value: side, with: V'_b) = false
          and intersect(value: side, with: V'_t) = false then
17:         m := m + 1
18:     else if state = out and intersect(value: side, with: V'_b) = false then
19:         state := top
20:     else if state = out and intersect(value: side, with: V'_t) = false then
21:         state := bottom
22:     else if state = top and intersect(value: side, with: V'_t) = false then
23:         m := m + 1, state := out
24:     else if state = bottom and intersect(value: side, with: V'_b) = false then
25:         m := m + 1, state := out
26:     end if
27:     side := find-first(in: sides, holds: come-after(in: P, direction: clockwise))
28:     visited := visited + 1
29: end while
30: return intersection := parity-of(m) = even
```

The verification of the property $\mathcal{P}$ is much simpler. We only need to find a circle for each point of V whose center is at least as close to the point of V as the radius of that circle. Formalizing this line of thought,

$$\forall(x,y) \in V, \exists \mathcal{O}_i \in \mathcal{O}, \sqrt{(x-x_i)^2 + (y-y_i)^2} < r_i$$

must be true.

The verification is implemented in Algorithm 2.4, *Verify-*$\mathcal{P}$. Although the idea is plain and clear, additional discussion is required. First, you may recognize that *Verify-*$\mathcal{P}$ is unable to handle situations when only more than one circle realize the total cover of an interval. In other words, the interval is entirely covered but none of the circles contain it. Second, if the interval is uncovered, it will be unknown that exactly which part of the interval violates the cover.

Algorithm 2.4 could be extended to determine the intersections of the intervals and circles and identify which parts are covered and which are not, or instead of a simple boolean value, we could return how many circles cover the interval, but such extra information is not necessary for the cover optimization as you are going to see in the second half if this chapter.

ALGORITHM 2.4. Verify-$\mathcal{P}$

```
1: input configuration: r_O
2: input interval: V
3: output boolean: verification

4: for all circle: O in create-list(values: circles-of(r_O)) do
5:     max-distance := maximum-of(points-of(V) | transformed(to:
         distance(from: center(O), type: Euclidean)))
6:     if max-distance < radius-of(O) then
7:         return verification := true
8:     end if
9: end for
10: return verification := false
```

However, it might seem troubling that Algorithm 2.3 and 2.4 only indicate which interval needs further examination, Algorithm 2.2 will continue

precisely the partitioning of these cases where the property $\mathcal{P}$ may hold, thus Algorithm 2.3 and 2.4 perfectly fit their roles.

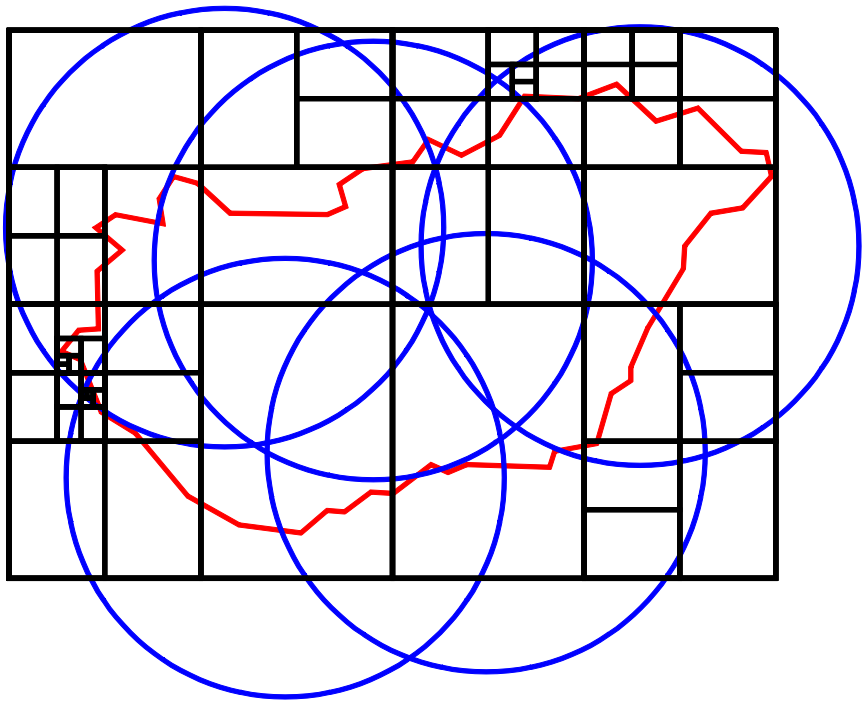


Figure 2.2: A good configuration covering the polygon approximation of Hungary including the final partitioning that the algorithm Verify-goodness generated.

Let us study the cover of the polygon approximation of the land of Hungary to have a better understanding of how the cover checking operates. The initial bounding intervals were the large rectangles containing the whole polygon.

In Figure 2.2, you can see a complete cover. The final partition of R consists of intervals that are either completely covered by at least one circle, see the rectangles in the middle, or do not overlap with R, see the rectang-

les near to the corners. Knowing the branching rule, it is possible to even reconstruct by hand the order of interval cuts.

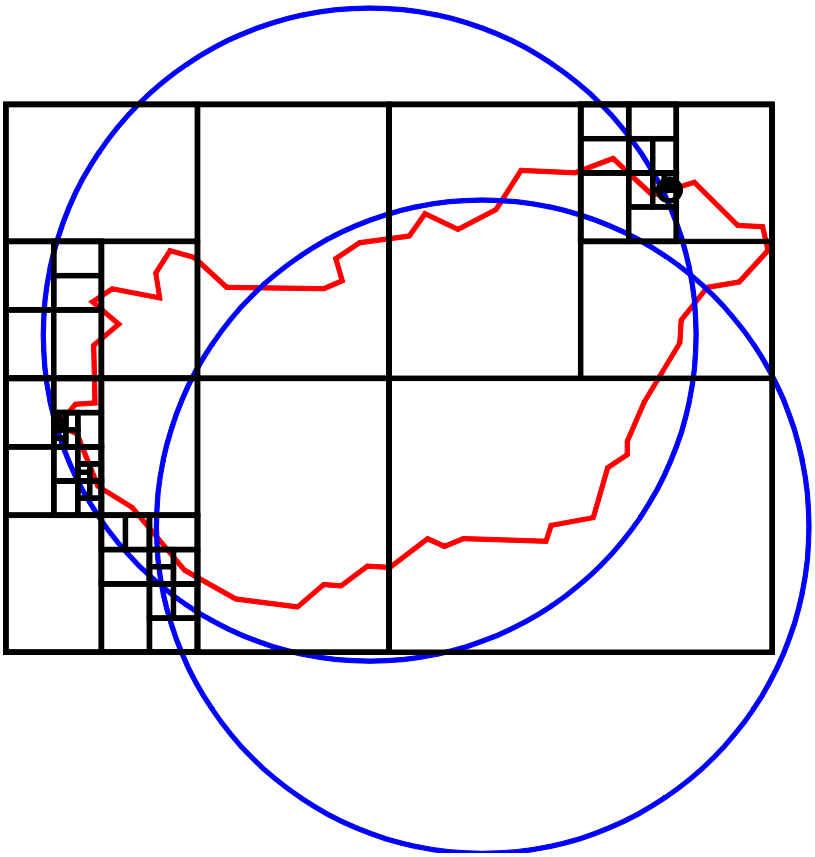


Figure 2.3: A bad configuration only partially covering the polygon approximation of Hungary.

Figure 2.3 shows a case when the algorithm found a counterexample to the cover located in the small black circle in the top right corner. Its neighborhood is illustrated in more detail on Figure 2.4. The width of this interval is smaller than the given ε, and no circle covers it. Two interesting aspects are visualized in this figure. The first is that quite a lot of iterations had to be made to find this interval. The smaller the ε, the later the algorithm terminates. If ε had been set to zero, then Verify-goodness would never terminate continuously creating smaller and smaller intervals whose width is

converging to zero.

Second, you might notice after a scrutiny that some intervals were subdivided despite they were clearly out of the polygon visually, see the rectangles in the bottom left corner of Figure 2.3. This is not a printing error but a side effect of interval arithmetic, more precisely, the result of the over-estimation of interval expressions.The bounding intervals of these rectangles got larger in the chain of interval operations than the rectangles themselves reaching a size large enough to indicate an overlap with R, hence triggering theoretically unnecessary partitioning.

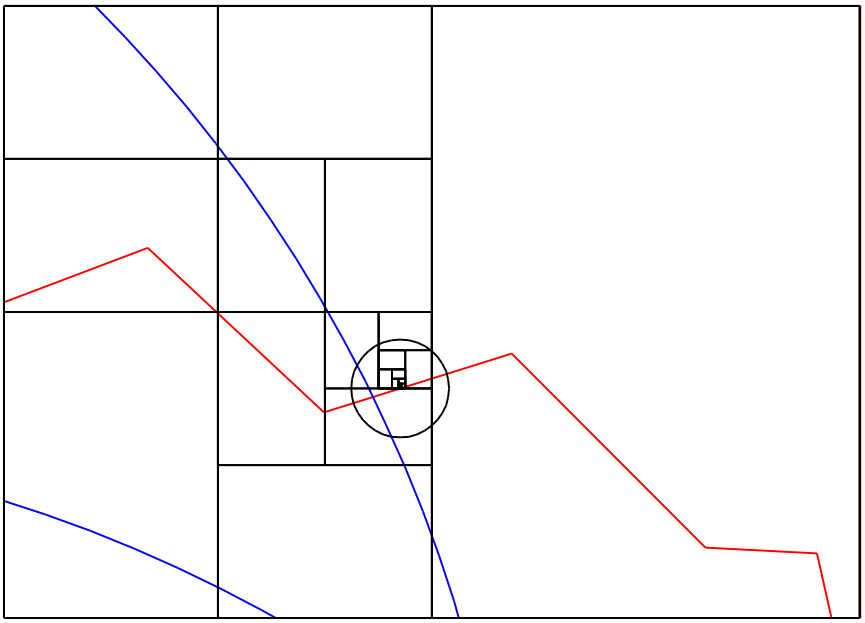


Figure 2.4: The neighborhood of the counter example showed in Figure 2.3

Before moving towards algorithmic correctness, some words must be said about the performance of this approach of the problem. Browsing the literature, you may find more efficient strategies to check the cover of polygons than ours. If special conditions are met, for example the centers of circles are located at the vertices of the polygon, more efficient algorithms may be applied like that in [14], We have never intended to optimize for such special

cases, our goal was to approach this cover checking problem as general as possible. If improved performance is required, a parallel version of the algorithm can be implemented naturally. The experiments of Balázs Bánhelyi and Endre Palatinus with a parallel implementation of Verify-goodness have been successful beyond expectations. They have reached a linear speed-up with respect to the number of used CPU cores.

After clarifying the technical details, let us analyze the correctness of the Algorithm 2.2. We are going to study the question of termination and the validity of computed results.

The first theorem captures the structure of interval partition that Verify-goodness generates.

Theorem 2.2.1. *Assume that we use Algorithm 2.2, an interval extension of a verifier algorithm to the property $\mathcal{P}$, and it returns that every point of R has the property $\mathcal{P}$. In this case, Algorithm 2.2 creates a partition of the initial bounding interval of R in which for every subinterval at least one of the following conditions holds:*

1. *the subinterval does not overlap with the studied region R, or*

2. *at least one circle contains all the points of the subinterval.*

Proof. We prove the theorem analogously to Theorem 2 in [16].

The algorithm must have stopped in Line 20, and not in Line 13, as it returned that every point of R has the property $\mathcal{P}$. Line 20 can only be reached if all the subintervals generated during the execution of Algorithm 2.2 were either further subdivided in Line 15, or skipped from further subdivision since the condition in Line 11 was not fulfilled.

We can assume that the subdivision in Line 15 happens in a way that the result intervals fully cover the original subdivided interval. This is an implementation detail, see the documentation of the applied programming library[19] for further information.

Moving one step further, none of the subintervals in the final partition fulfilled the condition in Line 11 (the Algorithm *Intersect* returned false, or Algorithm *Verify-$\mathcal{P}$* returned true), since Line 20 was reached, and the

algorithm stopped. This means that the starting interval I was fully covered by subintervals that fit the above mentioned cases *1* and *2*, and therefore Algorithm 2.2 is correct in this sense. □

Theorem 2.2.1 guarantees that the branch-and-bound strategy always produces a partition of R whose goodness is recognizable for us in case of good configurations.

Moving forward, the next objective is to validate that the algorithm recognizes total cover and terminates in that case.

Theorem 2.2.2. *Assume that we use Algorithm 2.2, an interval extension of a verifier algorithm to the property $\mathcal{P}$, that is capable of determining whether a point p and some neighborhood of it have the property $\mathcal{P}$; furthermore, the parameter ε of Algorithm 2.2 is set to zero, and every point of R has the property $\mathcal{P}$. In this case, Algorithm 2.2 will terminate with a positive result after a finite number of iterations.*

Proof. We prove the theorem analogously to Theorem 3 in [16].

Assume that the theorem is false, and the algorithm never stops and generates an infinite number of subintervals if the parameter ε is set to zero, and every point of R has the property $\mathcal{P}$. In this case, there must be at least one infinite sequence of subintervals that converges to some point p due to compactness. Based on whether $p \in R$, there are two cases.

Let us consider the case of $p \in R$ first. This means that p has the property $\mathcal{P}$, which means that some circle o covers p. Therefore there exists a given threshold index after all subintervals in this infinite sequence converging to p are also covered by o because the circles are open, but the algorithm will not make further subdivision for such subintervals in Line 15 as the Algorithm *Verify-$\mathcal{P}$* will return true for these subintervals in the second part of the condition in Line 11, whereas it should return false to let the algorithm reach Line 15. This contradicts that such infinite sequences of subintervals are exist.

The case of $p \notin R$ is similar. This means that there exists a given threshold index after all subintervals in this infinite sequence converging to p will

not intersect with R leading us to the fact that the algorithm cannot make further subdivision for such subintervals in Line 15 as the Algorithm *Intersect* will return false for these subintervals in the first part of the condition in Line 11, whereas it should return true to let the algorithm reach Line 15. Again, this contradicts that such infinite sequences of subintervals are exist.

As the parameter ε is set to zero, then the algorithm can only terminate in Line 20 that completes the proof of the theorem. □

Theorem 2.2.2 highlights the importance of using open circles, or more precisely, why it is crucial that their borders do not take part in the covering. Consider two circles that do not overlap except in one point of contact. Let region R contain this point and we want to check whether these closed circles cover R. Then our verifier will fail in that particular point as the point itself is covered but this is not true for any of its neighborhood, though it would be necessary as Theorem 2.2.2 states.

After discussing good configurations, we still need to study how the algorithm operates for bad configurations.

Theorem 2.2.3. *Assume that we use Algorithm 2.2, an interval extension of a verifier algorithm to the property $\mathcal{P}$, the parameter ε of Algorithm 2.2 is set to zero, and there is a point $p \in R$ which violates the property $\mathcal{P}$. In this case, Algorithm 2.2 cannot terminate after a finite number of steps.*

Proof. We prove the theorem analogously to Theorem 4 in [16].

Consider the subintervals generated by Algorithm 2.2 that contain the point p. Such subintervals have two properties:

1. they overlap R, thus Algorithm *Intersect* will return true for them,
2. they do not have the property $\mathcal{P}$, thus Algorithm *Verify-$\mathcal{P}$* will return false for them.

This implies that the condition in Line 11 will always be fulfilled. As the parameter ε is set to zero and these subintervals have always a positive width, the condition of Line 12 will never be true meaning that such subintervals

will always be subdivided into further subintervals and one of them will also have the properties 1 and 2.

As a consequence, while at least one subinterval described above is in the stack, the stack cannot get empty. As the starting interval that bounds R has the properties 1 and 2, the algorithm cannot terminate after a finite number of steps that concludes the proof. □

Theorem 2.2.3 means that the verifier only fits positive cases, it does not terminate when the polygon is just partially covered. We must also remarked that Verify-goodness might return *false* even for covering sets of circles. For example in case of a positive ε, we might found a counterexample interval whose cover is realized by two or more circles at the level of granularity that the value of ε allows. Thus, a negative answer does not mean undoubtedly that a given configuration fails to cover a studied region.

Theorem 2.2.4. *If a polygon P is covered by circles, then Algorithms 2.2, 2.3, and 2.4 terminate after a finite number of iterations with a positive outcome when ε is set to zero.*

Theorem 2.2.4 directly follows from Theorem 2.2.2, and therefore the correctness of our cover verification has been proven. Our next step is to build an optimization algorithm over this functionality.

Proved the correctness, the technical and theoretical analysis of our cover verifier tool is finished. Now, we are going to build an optimization method over this functionality.

2.3 Configuration optimization

In many practical applications, devices require power proportional to the covered area. More precisely, the power requirement is proportional to the square of the operational radius. Our goal is to reduce the energy usage of a set of such devices as much as possible by minimizing the sum of squares of the operational radii. Formally, we have to find the minimum of the following objective function.

Definition 2.3.1. *The objective function of the circle covering problem, in which we try to cover the target region R with configurations of n circles with prescribed centers, is the following:*

$$f(r_1, r_2, \ldots, r_n) = \sum_{i=1}^{n} r_i^2, \tag{2.1}$$

where $r_1, r_2, \ldots, r_n$ form a good configuration. Henceforth in this chapter, f will always refer the objective function

We discuss the algorithmic correctness of our optimization approach to facilitate the understanding the algorithm later. Let us start with the analysis of relationships of configurations.

Lemma 2.3.2. *If a configuration $r = (r_1, r_2, \ldots, r_n)$ is good, then every configuration $r' = (r'_1, r'_2, \ldots, r'_n)$ will also be good, where $r'_i \geq r_i$ for $i = 1, 2, \ldots, n$.*

Proof. The initial configuration r is good meaning that it covers the aimed region. If we enlarge the radius of a circle, it will not cover less area. Modifying this way more than one circle, even all of them, will keep the goodness in all the resulting configurations. □

Bad configurations have a similar connection.

Lemma 2.3.3. *If a configuration $r = (r_1, r_2, \ldots, r_n)$ is bad, then every configuration $r' = (r'_1, r'_2, \ldots, r'_n)$ configuration will also be bad, where $r'_i \leq r_i$ for $i = 1, 2, \ldots, n$.*

Proof. Analogous to the proof of Lemma 2.3.2. □

Although these lemmas are trivial, they are highlighting an important fact. From the goodness point of view, a single configuration can represent a whole set of configurations whose elements are created through monotonic alteration of the radii. This reduces the difficulty of handling configuration sets of infinite elements if we consider just two configurations instead as the following definition formulates this.

Definition 2.3.4. *Let* $C = \left[[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_n}, \overline{c_n}]\right]$ *be a set of configurations whose* i^{th} *component is in the interval* $[\underline{c_i}, \overline{c_i}]$ *for every* $i = 1, 2, \ldots, n$. *We call* $\underline{C} = (\underline{c_1}, \underline{c_2}, \ldots, \underline{c_n})$ *and* $\overline{C} = (\overline{c_1}, \overline{c_2}, \ldots, \overline{c_n})$ *the extremal configurations of* C.

In Figures 2.5(a) 2.5(b), and 2.5(c), you can see a good visual summary of what Definition 2.3.4, and the Lemmas 2.3.2 and 2.3.3 formalize.

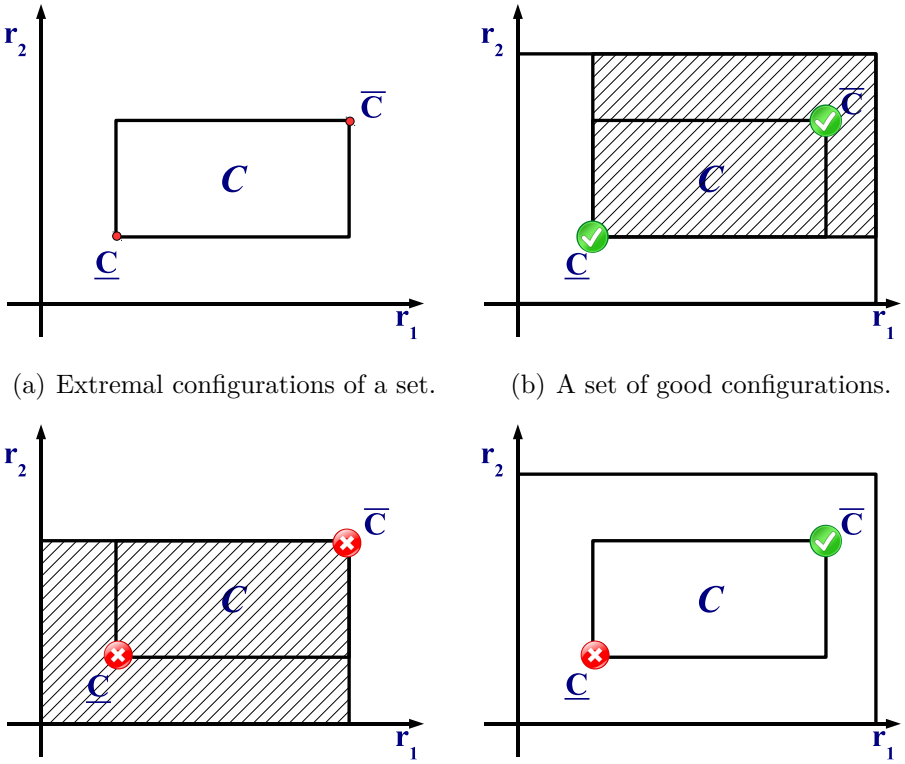
We gathered the necessary relations and notations to discuss where we should look for optimal configurations in order to synthesize an effective search strategy for our algorithm.

Theorem 2.3.5. *Any set of configurations* C *contains an optimal configuration in its interior if and only if* $\underline{C}$ *is a bad configuration and* $\overline{C}$ *is a good one.*

Proof. We are proving the theorem by contradiction in two parts. First, assume that the extremal configuration $\overline{C}$ of the configuration set C is bad, but C contains optima. All the elements of C can be created from $\overline{C}$ if we shrink appropriately the radii of $\overline{C}$. Such alterations preserve the badness of configurations according to Lemma 2.3.3. Therefore the entire set C must be made of bad configurations. This is a contradiction since we asserted at the beginning that C has at least one optimal configuration, and optima are good configurations. The first claim of the theorem is thus proved.

The second claim of the theorem is more challenging. Assume that the extremal configuration $\underline{C}$ is good, but C still contains optima. First, it follows from the structure of the objective function f, see the equation (2.1), and from Definition 2.3.4 that $\underline{C}$ has the lowest objection function value in C yielding that $\underline{C}$ is the one and only one optimum in C.

Let $\underline{c_i} = (\underline{x_i}, \underline{y_i}, \underline{r_i})$ be the i^{th} circle of $\underline{C}$ such that some points of the target region R are only covered by $\underline{c_i}$. If there is no such circle, then at least two circles cover every point of R. In that case, reducing the radius of a circle to zero in $\underline{C}$ results in a configuration that is still good and has an objective function value less than $\underline{C}$, thus $\underline{C}$ cannot be optimal. Henceforth, assume that there exists a compliant $\underline{c_i}$.



(a) Extremal configurations of a set.

(b) A set of good configurations.

(c) A set of bad configurations.

(d) A configuration set which may contain an optimum.

Figure 2.5: Illustration of connections between configuration sets and their extremal configurations from the goodness point of view considering two circles with radii r_1 and r_2.

Let A denote the region that $\underline{c_i}$ exclusively covers. The circles are open and R is closed by definition, therefore A is closed. This implies that there must be a sufficiently small, positive ε value for which the circle $c_\varepsilon = (\underline{x_i}, \underline{y_i}, \underline{r_i} - \varepsilon)$ still covers A. We create a new configuration from $\underline{C}$ by replacing $\underline{c_i}$ with c_ε in it. The resulting configuration,

$$C_\varepsilon = (\underline{c_1}, \ldots, \underline{c_{i-1}}, c_\varepsilon, \underline{c_{i+1}}, \ldots, \underline{c_n})$$

remains good. The radii are equal in C_ε and $\underline{C}$ except that $r_\varepsilon < c_i$. Therefore, $f(C_\varepsilon) < f(\underline{C})$ holds meaning that $\underline{C}$ cannot be optimal.

The proof has been completed, we showed that $\overline{C}$ must be good and $\underline{C}$ must be bad, otherwise C cannot contain optimal configurations. For illustration see Figure 2.5(d). □

Additionally, the proof of Theorem 2.3.5 points out that optimal configurations only exist as limits. We would also like to note that the above theorem can be generalized. If the objective function and the concept of goodness and badness change in a way that preserves the monotony, then everything we relied on in the proof remain true. For example, it is easy to understand that our result would hold if we moved the problem from the two-dimensional plane into the three-dimensional space using spheres instead of circles and replace the quadratic objective function with a cubic one.

Lemma 2.3.6 is the finishing touch of the theoretic background of the optimization algorithm.

Lemma 2.3.6. *There is no such configuration set C for which $\underline{C}$ is a good configuration and $\overline{C}$ is a bad one.*

Proof. This lemma is a trivial consequence of Lemma 2.3.2. □

We continue with *Optimize-cover*, see Algorithm 2.5. The following theorem states the global optimality of the returned configurations.

Theorem 2.3.7. *Assume that we have a verifier algorithm capable of deciding the goodness of configurations. In this case, Algorithm 2.5 always returns a good configuration if it is started from a configuration set C with $\overline{C}$ being a good configuration and $\underline{C}$ being a bad configuration, and there is no other good configuration whose objective function value would be better than the returned one by more than $\varepsilon\%$.*

Proof. Assume that the algorithm stopped and returned the configuration r_o, but the real global optimum, r_g is better than it by more than $\varepsilon\%$. Let C_o and C_g denote the sets which contain r_o, r_g in that order. The algorithm discarded precisely those configuration sets which could not contain global optima according to Theorem 2.3.5. We pushed every other configuration set into the priority queue including C_g.

Remember that $f(r)$ denotes the objective function value at the configuration r. Let us extend this notion to configuration sets and $f(C)$ denote the set of $\{f(r) : r \in C\}$ where C is an arbitrary configuration set. The algorithm stopped because C_o satisfied the condition

$$\frac{\text{width}([f(C_o)])}{\overline{[f(C_o)]}} \cdot 100 < \varepsilon. \tag{2.2}$$

Applying Definition 1.5.4 in the inequality (2.2) and rearranging the two sides, we obtain

$$\frac{\overline{[f(C_o)]}}{\underline{[f(C_o)]}} < \frac{100}{100 - \varepsilon}. \tag{2.3}$$

Since r_o is the upper extremal configuration of C_o, and the priority queue pops the configuration C which has the lowest $\underline{[f(C)]}$ value, the following inequalities hold:

$$f(r_o) \leq \overline{[f(C_o)]}, \tag{2.4}$$

$$f(r_g) \geq \underline{[f(C_g)]} \geq \underline{[f(C_o)]}. \tag{2.5}$$

Substituting $f(r_g)$ in the denominator and $f(r_o)$ in the nominator in the left hand side in the inequality (2.3),

$$\frac{f(r_o)}{f(r_g)} < \frac{100}{100 - \varepsilon} \tag{2.6}$$

holds according to the inequalities (2.4) and (2.5). This contradicts what we initially assumed. □

The proof remains valid if more than one globally optimal configurations exists. The corner stones of the algorithm are that we add only the sets which potentially contain optimal configurations based on Theorem 2.3.5, and we always take out the set with the best lower bound on the objective function to ensure the global optimality of the found configuration, see Theorem 2.3.7.

Additionally, we could use an absolute limit at the start of the loop phase to terminate, in which case the returned solution would differ from the real global optimum by at most ε. We decided to use the relative term because the relative difference and the global optimum are invariant to scaling while the absolute difference is not.

ALGORITHM 2.5. Optimize-cover

```
1: input configuration-set: C
2: input region: R
3: input precision: ε
4: input objective-function: f
5: output configuration: optimum

6: queue := create-queue(type: priority, order-by: minimum-of(function-value(f)))
7: if Verify-goodness(configuration: C̄, region: R, precision: ε) = true then
8:       push(value: C into: que)
9: end if
10: while size-of(queue) > 0 do
11:       C := pop(from: queue)
12:       F := create-interval(bounding: C | transformed(to: function-value(f)))
13:       if width-of(F)/F̄ · 100 < ε then
14:             return optimum := C̄
15:       else
16:             left, right := cut(value: C, into: halves, along: largest-dimension)
17:             C_1 := find(values: {left, right}, holds: contains(value: C_1))
18:             C_2 := find(values: {left, right}, holds: contains(value: C̄))
19:             goodness := Verify-goodness(configuration: C̄_1, region: R, precision: ε)
20:             if goodness = true then
21:                   push(value: C_1, into: queue)
22:             end if
23:             goodness := Verify-goodness(configuration: C_2, region: R, precision: ε)
24:             if goodness = false then
25:                   push(value: C_2, into: queue)
26:             end if
27:       end if
28: end while
29: return optimum := null
```

Figures 2.6, and 2.7 provide illustrations of how the algorithm works. We would like to cover the unit square from two opposite corners in this simple scenario. You can see the result of the optimization on Figure 2.8(a).

Points in Figure 2.6 represent the configurations of the two circles with radii r_1 and r_2, both are at most 2. The shaded rectangles are the generated

configuration sets. The dark gray and light gray sets were discarded during the search because the former had bad upper extremal configurations while the latter had good lower extremal configurations. The resulted optimum is located in the small circle, also shown in more detail on Figure 2.7.

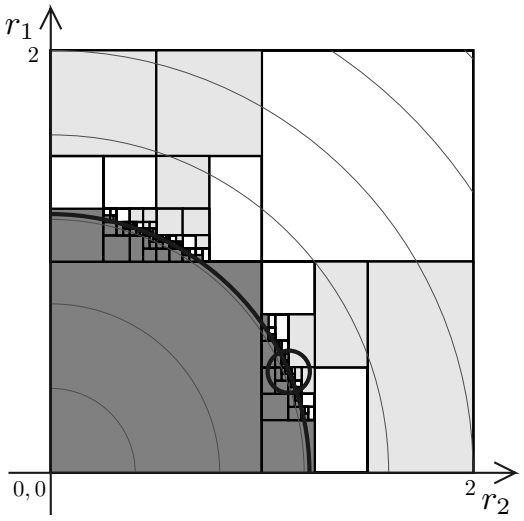


Figure 2.6: Sub-intervals generated by the optimization algorithm when we aimed to cover the unit square from two opposite corners.

The gray curves represent different levels of objective function values. The thick one denotes configurations having the same objective function value as the optimum has. As we do not aim to find all optima, the search stopped as soon as an approximation emerged at the user-defined level of ε. It is possible when some symmetry is present in the problem that other optima are in the priority queue when we finish the search, but none of them is better than the found one by more than $\varepsilon\%$, just like in this case.

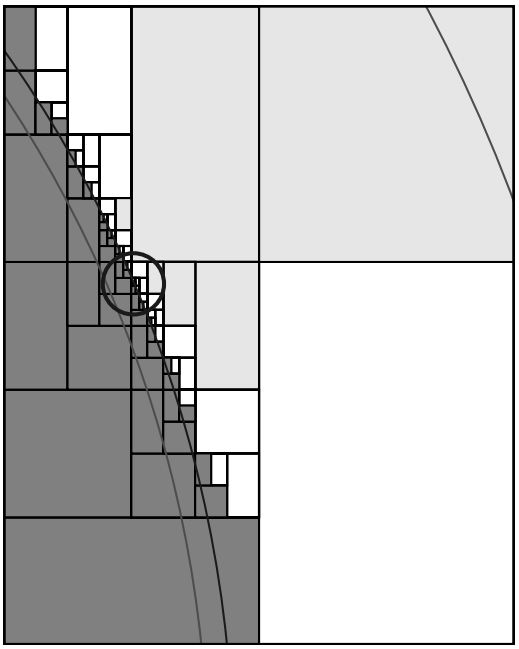
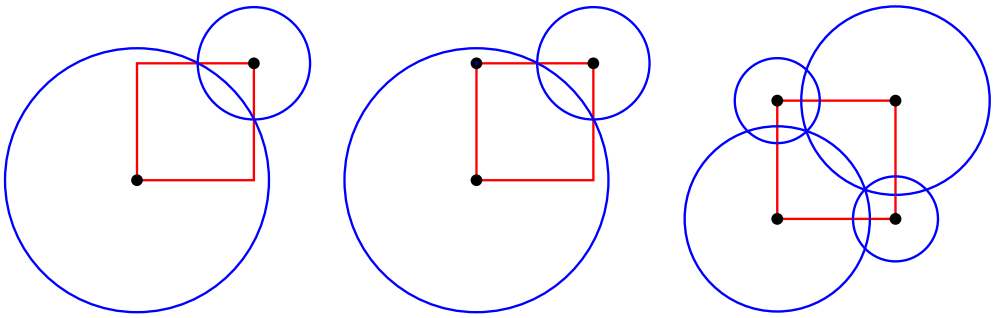


Figure 2.7: Zoomed in around the optimum of 2.6

2.4 Examples

In the final part of this chapter, we show some simpler cases, that provide an opportunity for the numerical analysis of the calculated optima, and some more complex examples to demonstrate the capabilities of the algorithm. The precision parameter ε was set to 1% during all searches that means the value of the objective function at any found optima will be at most $1,01$ times the value of the objective function at any of the real global optima.

Our first set of tasks are about covering the unit square. Figures 2.8 and 2.9 show the centers and the radii of the found optimal configurations.



(a) Covering from opposite corners.

(b) Covering from three corners.

(c) Covering from all corners

Figure 2.8: Simple scenarios when we aim to cover the unit square. Black dots denote the centers of the circles.

The case in Figure 2.8(a) was already discussed before from the search process point of view, now we focus on the objective function value of the optimum. The radii of returned configuration are 1.12890625 and 0.48046875 with an objective function value of $\approx$ 1.50527954. If closed circles were considered, the theoretical optimum would be the minimum of the expression

$$1 + (1 - x)^2 + x^2,$$

where x denotes the radius of the smaller circle. After rearranging the coefficients, we obtain the form:

$$2(x - 0.5)^2 + 1.5,$$

from which it is easy to calculate that the term is minimal at $x = 0.5$. 1.50527954 is clearly within the 1% error bound compared to theoretical optimum.

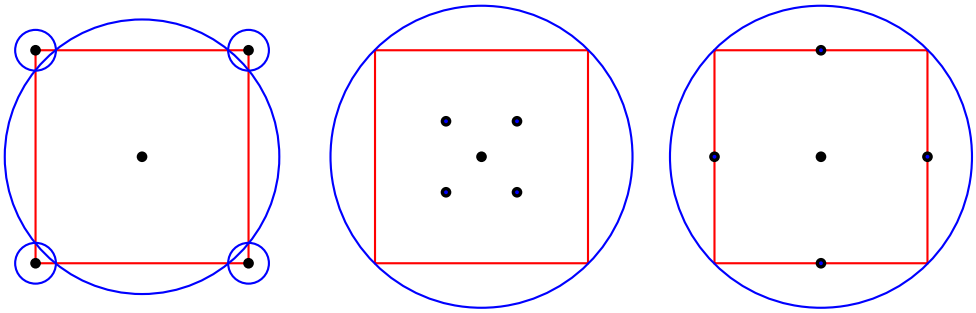
In Figure 2.8(b), a third corner of the square is added beside the previous two. The returned configuration remains approximately the same. The radius of the third circle is 0.00390625, which highlights an interesting characteristic of our method. By definition, upper extremal configurations consist

of upper bounds of radii. These bounds are initially positive, and partitioning the intervals preserve this condition. This forces every circle in the optimum to have a positive radius even if the circle would not be necessary at all to obtain complete cover.

Presented in Figure 2.8(c), adding the fourth corner to the centers of circles brings no improvement. Moreover, the found cover is even worse than in the previous scenarios. The radii of the larger circles are 0.796875000 and 0.783203125, and the smaller circles have an equal radius of 0.359375000 providing a cover with an objective function value of 1.50671. The theoretically optimal cover would be the one that minimizes the expression

$$2(\sqrt{2}/2 - x)^2 + (\sqrt{2}/2)^2,$$

where x is the radius of the smaller circles. Rearranging the coefficients leads us to the optimal value of x as 1.5 again. Had set a lower ε value, this exotic configuration would certainly have been discarded.



(a) Covering from the corners and the center of the square.

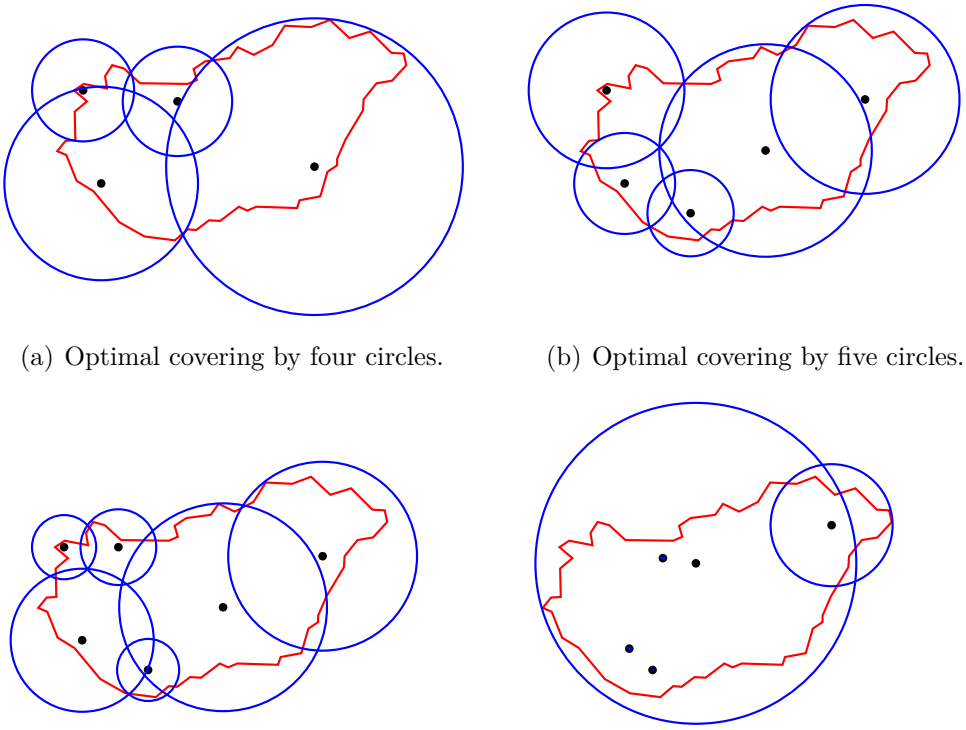
(b) Covering from five inner points including the center of the square.

(c) Covering from all side midpoints and the center of the square.

Figure 2.9: More complex scenarios covering the unit square.

Figures 2.9(a), 2.9(b), and 2.9(c) show more complex cases, but every one of them has some symmetry. Although it might seem that the optimal cover would be a single large circle at the midpoint of the square in all scenarios, this conjecture has not been confirmed all the time. Surprisingly, the optimal

cover consists of five circles with radii 0.095703125 and 0.64453125 having an objective function value of 0.4520569, see Figure 2.9(a).



(a) Optimal covering by four circles.

(b) Optimal covering by five circles.

(c) Optimal covering by six circles.

(d) Optimal covering by five circles, using only two effectively.

Figure 2.10: Some optimal covers of Hungary. Black dots denote the centers of the circles.

In our last set of examples, we tried to cover a more complex polygon, the representation of the country of Hungary. The randomly generated centers come from a uniform distribution of the positions of larger Hungarian cities. Figures 2.10(a), 2.10(b), and 2.10(c) illustrate the optimal covers by four, five, and six circles respectively. Figure 2.10(d) shows an unexpected result when the algorithm was given 5 centers, but the returned cover used only

two of them. Despite the numerical optimality of these solutions, realizing them would be probably impossible due to the range constraints of real TV and radio towers; however, we can get feasible covers if we slightly modify the optimization by maximizing the allowed radii. These examples show the strength of the optimization algorithm. It provides solutions that are counter-intuitive and it would be hard or even impossible to come up manually or by other means. These examples also highlight that real applications need careful parametrization and tweaking tailored to the task at hand.

2.5 Summary

This chapter studied a special type of circle covering problem by reliable, rigorous numerical tools. The centers of circles were considered to be fixed allowing only the radii to be changed, and we were interested in the thinnest cover of polygons. Two important algorithms were presented, a verifier and an optimization algorithm. The former is capable of validating whether a given set of circles covers a polygon, while the latter aims to find the cover closest to the global optimum with arbitrary precision. Both algorithms are based on interval arithmetic, thus the provided results are mathematically reliable, free from numerical errors. To demonstrate their operation and effectiveness, we calculated several optimal covers of the unit square and the polygon approximation of the land of Hungary using various number of circles and different centers.

Chapter 3

Designing LED based streetlights

The practical issues of everyday life lead sometimes to surprising solutions. Light pollution, a symptom of our modern age, offers an interesting problem and an even more interesting solution as well. The term of light pollution means the unnecessary lighting of outdoor areas that came into focus in the last twenty years.

Light pollution is the reason why astronomers and astrophysicists do not recommend the construction of observatories in highly populated areas, nevertheless light pollution is not just the exclusive problem of a few scientists. It affects our life on a much larger scale disturbing the wild life all around the globe, especially the insects. Such species as the fireflies, whose mating ritual essentially involves light signals, suffered a heavy drop in their numbers. Confused by artificial light, males and females cannot find each other. On a much more global scale, light is also an important factor in animals' navigation and migration. Not just the timing of human created light is the problem, but its polarization too, because many animals use the natural polarization of sun light as information. The list of malicious effects on plants and animals could be continued, see [48]. Energy consumption is another relevant aspect. A rough estimation of 25 percent of our total energy needs is required for lighting purposes. More about this topic among other harmful

effects of light pollution are in the papers [25, 28, 67].

All the undesirable effects mainly come from the imperfection of incandescent light bulbs and other light sources used in street lights whose radiation cannot be controlled enough to light only what is needed to be enlighted, see Figure 3.1 for illustration of the light pattern of a traditional streetlight. Nowadays, LED technology matured to the level where it is capable to offer a solution to design and build better lighting instruments that save energy compared to the widely used incandescent light bulbs, not to mention the longer lifetime and optional smart features like the adaptivity to the traffic flow.

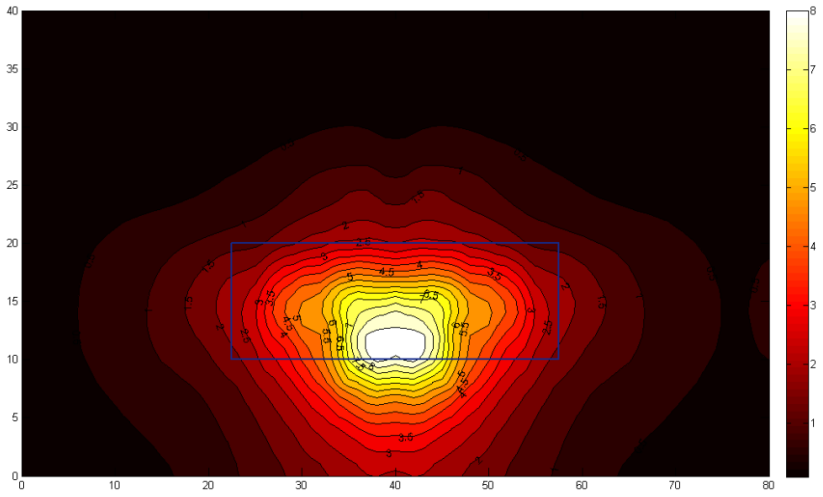


Figure 3.1: The light pattern of a regular incandescent streetlight. The rectangle represents the street surface that should be lighted. Colors denote the different levels of illuminance measured in LUX. Such street lights lack the necessary control to only light what should be visible at night and leave in the dark what should not be.

The production of a LED streetlight, that has all the attributes to replace an incandescent street light, is not as easy and straightforward as it sounds for the first time. LED lights have a focused light cone, the very reason of why we aim to use them, and thus they cover a much smaller area forcing the

engineers to put multiple LED packages into a single housing of a LED street light. This implies that the angle of each light source must be carefully set to distribute the light emission more or less equally on the target surface. Laws regulating public lighting are very strict about the quality of lighting in protection of motorists. Dim, bright, or uneven lighting equivalently obstructs the motorists' perception.

This chapter discusses an automatic designing methodology that calculates a complete LED streetlight design ready for production based on the parameters of the target street section and available light sources. The challenge is to determine the right type and angle of every single LED light to comply the various state regulations and provide an affordable and competitive product in terms of cost and energy efficiency. This is a very high-dimensional global optimization problem, a complex mathematical task that is impossible to do manually. Surprisingly, it is connected to the field of covering problems. The intersection of the target surface and a light cone of a LED light is an ellipse. As the intensities of different light sources simply add up, we can interpret this designing problem as covering a rectangle shaped area with ellipses while overlapping is allowed. The task is even more sophisticated as the light intensity within the same ellipse varies depending on the lighting characteristics and the direction of the source LED. Altering the direction of a light source does not only change the position and shape of the ellipse but its extent of contribution to the coverage too that probably reveals the depth of complexity even better.

The complexity of designing LED streetlights exceeds the capabilities of deterministic approaches. Let us build up the general problem from the simplest scenario. There is a single LED light that lights the street section, we may only alter the angle of the light source, no other modification is allowed. The objective function is similarly simple, it is the average illuminance of the measuring points. The gradient can be calculated, the function is continuous, even convexity can be exploited. As this model is not very realistic unless we wish to light a very small area, let us reduce the restrictions and use multiple but a fixed number of LED lights. The problem remains continuous, however the dimension, and the number of optima are significantly increased while the

objective function has no convexity property anymore. Let us generalize the problem further allowing the optimization algorithm to change the LED type in the sockets or vary the number of applied LED lights in the housing. The model has lost continuity and the chance that we can calculate a formidable solution with deterministic algorithms. Including the cost of LED lights and their energy consumption in the objective function gives us the final form of this problem that only stochastic algorithms might be able to handle. We faced a similar level of complexity later in another project whose aim was the development of the software component of a failure detection and monitoring system. Again, our solution was a stochastic approach that automatically constructed a classifier capable of identifying certain types of malfunctions based on the incoming sensor data, see the paper [11] for details.

Similar to our solution in Chapter 2, we present a two-step approach. First, we develop a method to describe and evaluate the light pattern of different lamp designs, then we discuss the actual optimization algorithm. The cover optimization algorithm of the previous chapter was a rigorous method providing results of proven correctness. In this chapter, we study the other end of the reliability axis by building a stochastic solution based on a metaheuristic using simple, floating point arithmetic. The nature of the problem does not require the rigorousness of interval computation, and the number of dimensions renders the deterministic methodologies impossible to apply with success.

I worked on the topic in relation to an industrial project led by Balázs Bánhelyi for Wemont Kft. Beside giving me some initial direction and regularly reviewing my work, Balázs let me face the challenge of researching and developing the algorithmic core of our solution alone, thus I can consider the theoretical results of this chapter, the adaption of the concept of genetic algorithm, to be my own. Calculating the numerical results and implementing the algorithm were also mostly my effort during the software development. We published the results with Balázs Bánhelyi as a chapter in a book [46]. The software component we created became the part of an international patent about LED streetlight construction, for details see the patent [64].

3.1 Light pattern calculation

Design evaluation requires the scoring of the emitted light pattern. Its calculation happens at the vertices of a two-dimensional grid on the target surface. The granularity of this grid is prescribed by law. The physical quantity that we aim to determine is illuminance, the measure of how much luminous flux is spread over a specific area. As this chapter does not aim to reveal the beauty and depths of photometry or radiometry, we will not use precise physical definitions. It is enough to understand the computation model if you imagine luminous flux, whose unit is lumens, as the measure of how much visible light is present on the surface while illuminance, whose unit is LUX, is the measure of the intensity of illumination on the surface. For an exact discussion of these concepts, see the textbook [1].

Illuminance is inversely proportional to the respective area if the emitted luminous flux is held constant. If the intensity of illumination is known in a given direction and distance from the light source, then the illuminance will change with the distance in the same directions following the formula

$$E(d) = E_{d_k} \cdot \left(\frac{d_k}{d}\right)^2,$$

where E_{d_k} is the known illuminance measured in LUX in d_k meter distance from the light source. The lighting characteristic of a specific LED light is given as an illuminance distribution. It is measured in equiangular directions both vertically and horizontally in a fixed distance from the light source. These angle pairs also create a rectangular, two-dimensional grid just as the measuring points. For actual data formats, see EULUMDAT [24] or IES [36].

The directions of measuring points from the light source almost never coincide with any premeasured angle of the lighting characteristic, therefore we need to interpolate the illuminance towards the measuring points based on the closest directions of known values. Bilinear interpolation extends linear interpolation for functions of two variables whose known values are on a rectangular, two-dimensional grid, just as in our case. The function is simple to calculate while it provides the necessary precision for our purpose. This method performs linear interpolations separately along the different

axes using the formula

$$f(x,y) \approx \frac{1}{(x_2-x_1)\cdot(y_2-y_1)} \cdot \begin{bmatrix} x_2-x & x-x_1 \end{bmatrix} \cdot \begin{bmatrix} f(x_1,y_1) & f(x_1,y_2) \\ f(x_2,y_1) & f(x_2,y_2) \end{bmatrix} \cdot \begin{bmatrix} y_2-y \\ y-y_1 \end{bmatrix},$$

where (x_1, y_1), (x_1, y_2), (x_2, y_1), and (x_2, y_2) are the four grid points closest to the point in which we want to interpolate f. Figure 3.2 illustrates how the interpolation works. First, we execute two linear interpolations along the axis x in the points R_1, using the values at $Q_{1,1}$, $Q_{2,1}$, and R_2, using the values at $Q_{1,2}$, $Q_{2,2}$, that have the same x coordinate as P.

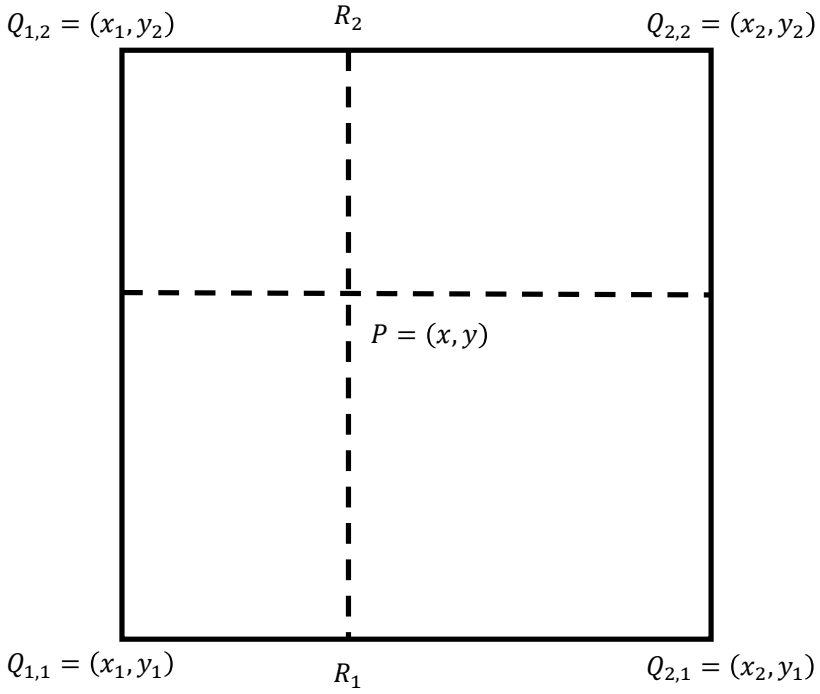


Figure 3.2: The key points used in a bilinear interpolation aiming to approximate a function's value at P based on the known values at $Q_{1,1}$, $Q_{1,2}$, $Q_{2,1}$, and $Q_{2,2}$.

The value of f at P is approximated using a third linear interpolation based on R_1 and R_2 along the y axis. The order of axes used in the process

is invertible leading to the same result what the formula reflects as well. This method much resembles a quadratic interpolation though the name suggests otherwise. When the computation cost has low priority, not like in our case, we only wish to visualize a single light pattern for example, other, more sophisticated methods can replace this interpolation technique like the bicubic or spline interpolation [87].

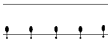
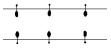
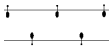
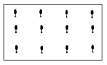
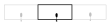
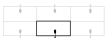
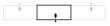
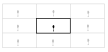
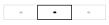
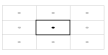
					
					
			Not possible	Not possible	
			Not possible	Not possible	

Table 3.1: The 11 different light pattern scenarios based on the number of axes of symmetry and the deployment of lampposts. Ellipses and circles denote the position of lamppost, the rectangles represent the target surfaces.

The effect of multiple light sources simply add up at a point, thus the essential problem of light pattern determination is the computation of illuminance at a single point caused by a single light source. The amount of required operations to calculate the light pattern can be drastically reduced if we consider the symmetries present in the problem. Looking over the common scenarios of public lighting, see Table 3.1, the target surface may have 1, 2, or 4 axes of symmetry. This enables us to only consider the one half,

one fourth, or one eighth of the target surface whose light pattern should be calculated. This reduces the number of measuring points we have to handle accordingly. The complete light pattern can easily be obtained by mapping the known illuminance of measuring points to the corresponding measuring points.

Let us take a look at Figure 3.3 that shows a light pattern of a LED streetlight designed by our algorithm. This figure highlights an important factor that further shapes the light pattern, the light coming from the neighboring streetlights. Even the light pattern of LED streetlights has positive illuminance extending beyond the target surface that increases the illuminance of surfaces belonging to the neighboring streetlights. This effect must be considered with respect to the deployment of lampposts. A possible solution is to determine the illuminance in grid points located outside of the target surface that can be mapped to the inner measuring points of the neighboring target surfaces.

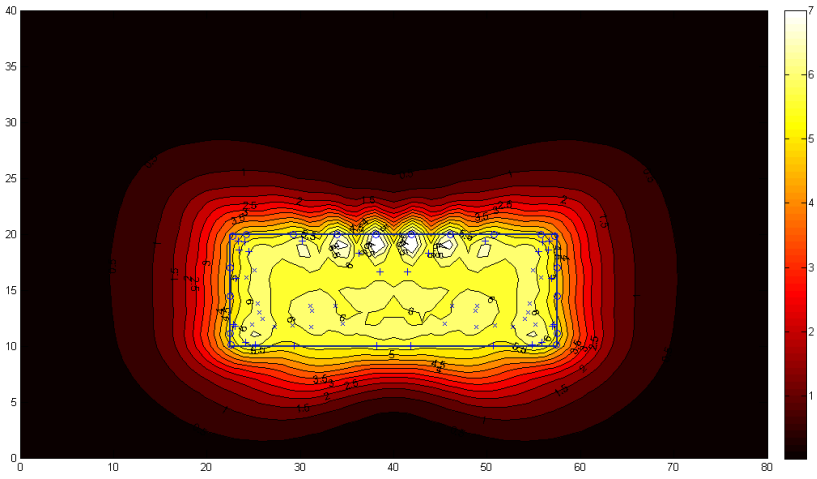


Figure 3.3: The light pattern of a LED streetlight designed by our algorithm. The rectangle represents the street surface that should be lighted. Different shades of colors denote the different levels of illuminance measured in LUX.

Calculate-light-pattern, see Algorithm 3.1, is a higher level description of

light pattern handling based on the points that we discussed. Lower level details are omitted as they entirely depend on the input data format of the lighting characteristics and are marginal from the algorithmic point of view.

ALGORITHM 3.1. Calculate-light-pattern

```
 1: input surface: R
 2: input light-source: lamp
 3: output light-pattern: pattern

 4: target := scale(value: R, ratio: 2)
 5: grid := create-grid(over: target, center: lamp, step: 1 m)
 6: grid-parts := cut(value: grid, along: symmetries-of(lamp))
 7: partial-grid := first-element-of(grid-parts)
 8: measure-points := create-list(type: point,
      values: points-of(partial-grid))
 9: for all point: point in measure-points do
10:       illuminance-of(point) := 0
11:       for all light-source: led in light-sources-of(lamp) do
12:             direction := create-vector(from: position-of(lamp), to: point)
13:             nearest := create-condition(closest(to:
                  scale(value: direction, to: unit)))
14:             Q := find(values: lighting-characteristics-of(led),
                  holds: nearest, count: 4)
15:             I := interpolate(in: direction, base: Q, strategy: bilinear)
16:             illuminance-of(point) := illuminance-of(point)
                  +I / (length-of(direction))^2
17:       end for
18:       copies := reflect(value: point, strategy: axial, along: symmetries-of(lamp))
19:       add(values: {point, copies}, to: pattern)
20: end for
21: neighbor := pattern
22: pattern := clip(value: pattern, area: R)
23: correct(pattern: pattern, neighbor: neighbor,
      along: symmetries-of(lamp)))
24: return pattern
```

Without the loss of generality, we can consider the housing of LED lights as a dimensionless point for simplicity. Assigning values to all measu-

ring points is not necessary if we track their multiplicity with regard to the symmetries of the lighting scenario. The measuring point generation must carefully be designed to support the exploitation of symmetries and the last, correction step that incorporates the illuminance coming from the neighboring streetlights into the final result. Considering twice as long and wide target surface as the original one was sufficient from this point of view.

3.2 The concept of genetic algorithms

Stochastic optimization provides a wide range of tools. Choosing the right method is difficult, but examining the problem structure helps. A design can be locally optimal, not just a local optimum due to LED packages are capable of lighting only relatively small areas. A mediocre design can be partially good, lighting a given street section perfectly, thus the combination of several less good designs might result in a suitable one for production. An ideal solution for this type of optimization problem should be able to use partial solutions of different designs to create even better ones. Genetic algorithms have this capability.

The concept of genetic algorithms [30] is a member of a larger class of metaheuristics, the evolutionary algorithms [23]. The basic idea comes form another discipline, evolutionary biology. If we interpret evolution, that shapes the biological entities, in our abstract space, the optimization process will be transformed into a simulation of natural selection. A genetic algorithm is an iterative approach. The search for an optimum starts from a group of objects, the *population* whose elements are called the *candidate solutions*, or candidates for short, that represent the entire problem space. The elements constituting the population at the start of each iteration are the called the *generation* of this iteration. The optimality of candidate solutions are measured via the evaluations of a *fitness function* that results in a fitness score, or just fitness. The members of the first generation, usually at least hundreds or thousands of candidate solutions, are randomly selected from the problem space. A portion of the generation is selected in each iteration con-

sidering the fitness scores of the candidates, but this is not mandatory. The current generation is modified by applying the genetic operators of *mutation* and *crossover* to the selection of candidates.

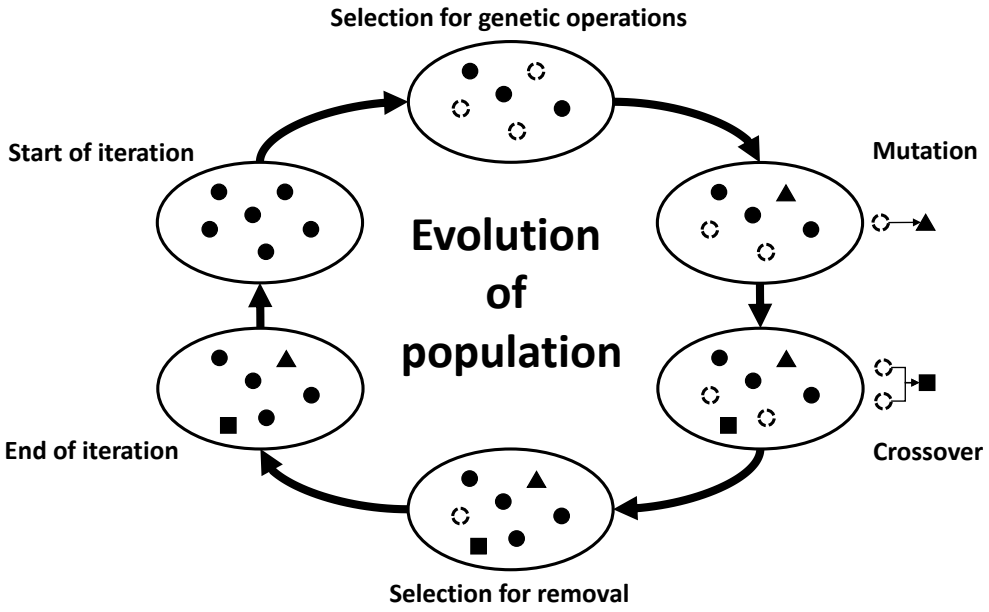


Figure 3.4: One iteration of a genetic algorithm. Ellipses represent the different states of generation in the stages of the iteration. Circles, triangles, and squares denote the candidate solutions.

The final step of each iteration is the removal of some candidate solutions based on fitness to keep the population size constant across iterations as the crossover operators add new candidates to each generation. This process is illustrated in Figure 3.4. The optimization continues until a combination of the classic termination criteria are not met like the runtime, the number of fitness function evaluations, or the number of iterations reach a limit, or the change of the best fitness score is less than a predefined ε threshold.

3.3 Candidate solutions and fitness

We start building our genetic algorithm by defining the candidate solutions and the fitness function as the other algorithmic components depend on their

structures. The textbook representation of candidates are fixed length strings of zeros and ones, each binary number means the existence or absence of some property. The motivation of this choice is the simplicity and intuitiveness of the definition of genetic operators.

On the analogy of binary strings, let a candidate solution be a list of directed LED lights. Each entry in this list consists of a direction in which the LED light points to and a LED type that refers the price, the energy consumption, and the lighting characteristic required for the light pattern. Holding individual position information is not necessary as the production technology of the lamp housing enables us to consider all the LED lights as if they were in the same position, see Figure 3.5 for illustration.

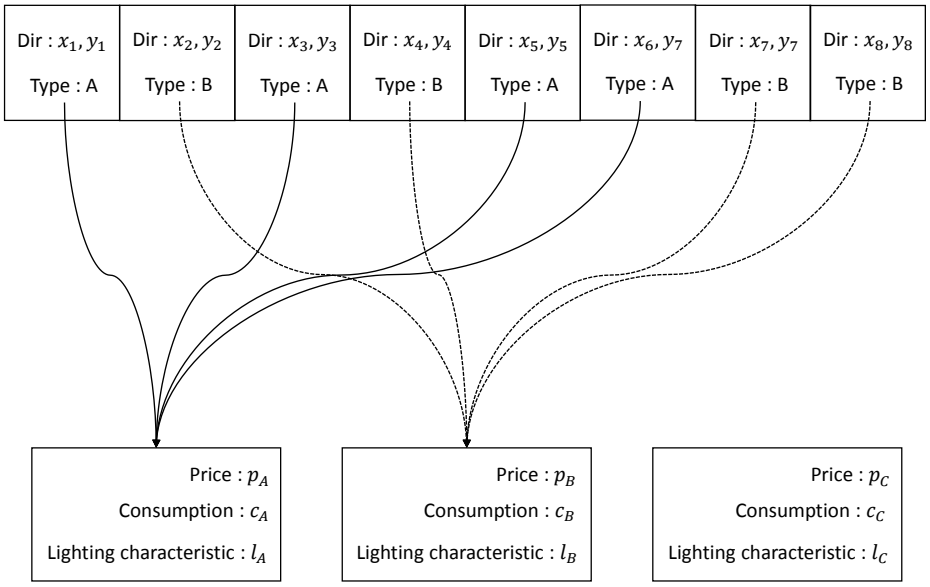


Figure 3.5: Example candidate solution using only 2 of the 3 available LED types, a variable sized list of 8 entries, each one containing the direction and type of the represented LED light.

As we need the ability to remove lights or add new ones to the design, we can use variable sized lists or fixed sized lists introducing an auxiliary LED type, an *empty* type, that represents the deleted LED packages from the design. We chose the first one as the relative position of lights within the

lists does not matter.

The next step is the fitness function that measures how good the different candidates are, and will affect the selection for removal from the population and being the operand of genetic operations. In our case, we have a multi-objective optimization problem, we have to simultaneously consider different, not equally important requirements that are thus categorized into hard and soft requirements. Hard ones are production blocking factors incorporating the state regulations that may forbid the deployment of the final product. The soft requirements measure the quality and competitiveness and allow some room to maneuver.

Our fitness function, $f(x)$ is the normalized, weighted sum of different terms, each one of them representing a requirement.

$$
\begin{aligned}
f(x) &= f_{md}(x) + f_m(x) + f_v(x) + f_p(x) + f_c(x) + p_I(x) + p_{I_m}(x), \\
1 &= w_{md} + w_m + w_v + w_p + w_c + w_I + w_{I_m}, \\
\overline{f}(x) &= \frac{1}{1+x}, \\
f_{md}(x) &= w_{md}\overline{f}\left(\frac{1}{\text{size-of}(I(x))}\sum_{i \ in \ I(x)}(|I_m - i|)\right), \\
f_m(x) &= w_m\overline{f}\left(|I_m - \text{mean}(I(x))|\right), \\
f_v(x) &= w_v\overline{f}\left(|I_{Var} - \max(I_{Var}, \text{Var}(I(x)))|\right), \\
f_p(x) &= w_p\overline{f}\left(\max\left(\overline{P}, \text{sum}(P(x))\right) - \overline{P}\right), \\
f_c(x) &= w_c\overline{f}\left(\max\left(\overline{C}, \text{sum}(C(x))\right) - \overline{P}\right), \\
p(\underline{x}, x, \overline{x}) &= \overline{f}(\underline{x} - \min(\underline{x}, x) + \max(\overline{x}, x) - \overline{x}), \\
p_I(x) &= w_I p(\underline{I}, I(x), \overline{I}), \\
p_{I_m}(x) &= w_{I_m} p(\underline{I}_m, \text{mean}(I(x)), \overline{I}_m),
\end{aligned}
$$

where $I(x)$ is the set of illuminances calculated at the measuring points of the light pattern that belongs to the candidate solution x, $P(x)$ denotes the set of prices of the applied LED lights in the candidate solution x with multiplicity, and $C(x)$ is the set of energy consumptions on the analogy of $P(x)$. $\overline{I}$, $\underline{I}$, are the expected maximum and minimum of illuminances at the measuring

points, I_m, $\overline{I}_m$, and $\underline{I}_m$ are the expected optimum, maximum, and minimum of the mean of illuminances, I_{Var} is the expected variance of illuminances, and last but not least, $\overline{P}$, $\overline{C}$ are the expected maximum price and energy consumption of the candidate solution.

Now f_{md}, f_m, and f_v are the measures of how far a candidate solution is from lighting the target surface evenly in terms difference between the absolute mean deviation, mean, variance, and their expected values. Setting the function weights appropriately, we can adapt the fitness function to the regulation differences between countries, as governments can define the smoothness of the light pattern differently. Here p_I and p_{I_m} further penalize the unevenness of the light pattern if individual illuminance values, or the mean of the illuminances leave an expected interval. f_{md}, f_m, and f_v are typically soft requirements while p_I and p_{I_m} are hard ones. $f_p(x)$ and $f_c(x)$ score the price and energy consumption. Design scenarios can be configured to have $f_p(x)$ measure the difference between the number of used LEDs and the number of expected maximum LEDs in streetlights.

3.4 Genetic operators

The first applied operator is the selection in each iteration. It determines the portion of candidate solutions that will participate in the crossover operations. A good selection process serves two goals, the improvement of fitness and the preservation of genetic diversity in the population. If we apply truncation selection for example, always a fixed number of the best candidates will be promoted whose attributes will dominate and spread in the entire population after several iterations making all the candidates more or less the same. This likely results in a suboptimal pool of candidates from which the genetic algorithm cannot step out due to the loss of diversity. This phenomenon is called *premature convergence*. A selection method that favors the candidates with better fitness but does not exclude the worse ones fits both of our goals.

Unlike fitness functions, they are completely problem-independent as they

solely depend on the fitness value and do not regard the inner structure of candidate solutions. The *fitness proportionate selection*, or *roulette wheel* selection is widely used in genetic algorithms. The idea is to assign the following probability to the i^{th} candidate in the population:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j},$$

where n is the population size and f_i denotes the fitness of the i^{th} candidate solution. The roulette wheel selection then generates a random number r from the uniform distribution over $[0, 1)$. The operator selects the j^{th} candidate that fulfills the condition

$$\sum_{i=1}^{j} p_i \leq r < \sum_{i=1}^{j+1} p_i.$$

We can illustrate this process with a roulette wheel that has n bins. The size of each bin is proportional to the fitness value of a candidate. Selecting a candidate is like rotating the roulette wheel and picking the candidate whose bin contains the ball.

An enhanced version of the fitness proportionate selection is the *stochastic acceptance* method. It is based on the same probabilities as in the case of fitness proportionate selection. First it uniformly selects a random candidate, let it be the i^{th} one, and generates a random number r from the uniform distribution over $[0, 1)$ just as before. The i^{th} candidate is promoted for genetic modification if $r < p_i$, we repeat the process otherwise. After considering several methods like ranking or tournament selection [23] beside the discussed ones, we chose the stochastic acceptance due to its lower computation cost.

Crossover introduces members into the population that can be structurally farther from the other candidates. This helps the algorithm to leave local optima and progress faster. These operators create new candidates from two different parent solutions in most cases, although combining three or even more candidates is also allowed if it better fits the problem. Continuing the classic example, the crossover operator may swap the same section of two strings that results two offspring, or it can take the value from one string or

the other for each bit with a probability proportional to their fitness. The number of offspring does not have to be two in this case.

The crossover concept for binary strings could work for our candidates exchanging LED lights instead of bits, but we chose a more clever way to combine two candidates that suits our design task better. Projecting an acceptable light pattern is the most important requirement. Let us put together the LED lights in the offspring based on their direction instead of their position in the list of the candidates. First, we generate a random rectangle on the target surface and determine which LED lights' direction vectors point into the rectangle in both candidates. The offspring candidate solutions will contain the LED lights that point into the rectangle from one parent, and the LED lights that point outside of the rectangle from the other parent. We swap light pattern parts instead of candidate parts this way.

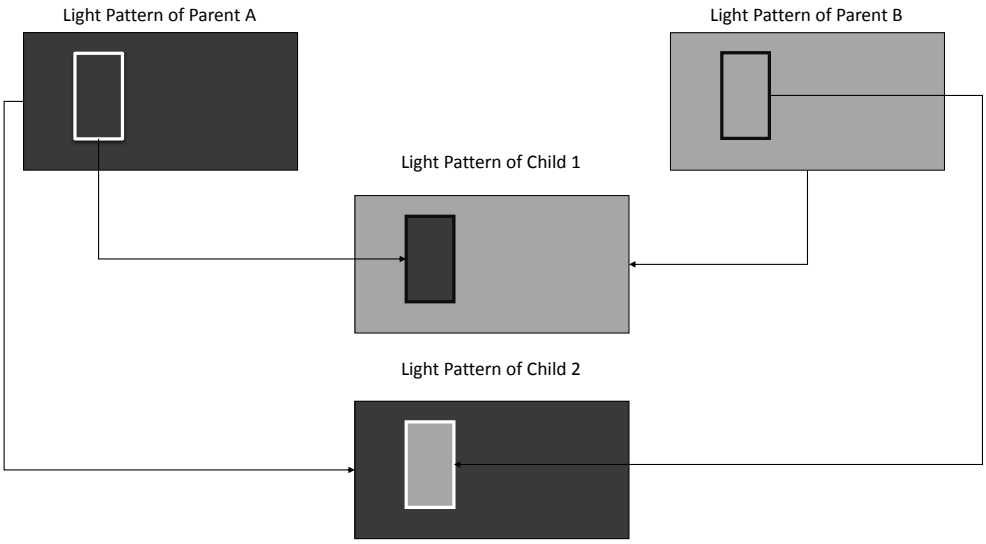


Figure 3.6: Illustration of recombining two candidate solutions based on light patterns.

It is essential how we generate the rectangle. It can be neither too large nor too small or the crossover will simply copy the parents. We also have to decide what kind of randomness we stride towards. There is two distinct approaches conceptually. We can focus on the randomness of the rectangle

as a geometric shape, or we can focus on the randomness of which part of the surface we select.

ALGORITHM 3.2. Random-rectangle

```
1: input rectangle: target-surface
2: output rectangle: random

3: height := height-of(target-surface)
4: width := width-of(target-surface)
5: points := create-distribution(type: uniform, values: [0, width] [0, height])
6: P, Q := generate-points(from: points, count: 2))
7: random := create-rectangle(diagonal: {P, Q})
8: area-ratio := area-of(random) / area-of(target-surface)
9: while 0.9 <= area-ratio or area-ratio <= 0.1 do
10:     Q := generate-points(from: points, count: 1))
11:     random := create-rectangle(diagonal: {P, Q})
12:     area-ratio := area-of(random) / area-of(target-surface)
13: end while
14: return random
```

Algorithm 3.2 is a possible implementation for the first approach. Although the algorithm can generate a rectangle over any part of the target surface, sections closer to the center are more likely to be covered. We will denote this crossover operator by $c_r r$ later in the final algorithm.

There are several ways to ensure uniform selection of surface parts. The bisection approach iteratively cuts the whole surface into halves along its longer side and repeats the process with one half selected randomly. The partition approach defines a grid like partition over the surface and uniformly selects a partition. We can control the size of the result rectangle through the number of iterations and the granularity of the partition. As being able to position the rectangle anywhere within the target surface is very important we decided to implement the *Random-rectangle* algorithm.

Mutation operators are capable of fine tuning the candidate solutions. Continuing the textbook example, a mutation operator can flip a randomly selected bit to its complement in the binary strings, or we can let the operator flip any number of bits but with a given, appropriately low probability.

Mutation is very similar to taking a step from one solution to a neighbor in the hill climbing algorithm, or other search heuristics like the simulated annealing, or taboo searching.

We defined multiple mutation operators to be able to modify the list of LED lights itself and single LEDs within them as well. These operators are the following:

- $m_{dr}(x)$ slightly changes the direction of the LED light x,
- $m_{tp}(x)$ replaces the type of the LED light x with another one,
- $m_{ar}(X)$ removes a LED light from the candidate or adds a randomly generated one to it with equal chance.

All operators generate random values from the uniform distribution over the range of available LED types, the range of existing LED lights in the candidate solutions, and predefined angle intervals.

There are further, more advanced concepts for operators that mimic other further genetic and population dynamics in case a problem requires non-standard approaches, see [30] for examples.

There are multiple strategies to apply the mutation and crossover operators on the selected candidates. Some genetic algorithms apply only one operator per candidate solution, other ones may use more than one for the same candidate. Another option is elitism that guaranties the unaltered survival of the best candidates into the next generation. In this case, the candidates having the best fitness values may only participate in crossover operations and cannot be removed. We decided to apply more than one operator for the same candidate. Candidate solutions with the lowest fitness scores are removed to keep the population size constant at the end of each iteration. The genetic algorithm runs until one of the termination criteria is met, the predefined maximum of executed iterations or runtime is reached, the best fitness value has not changed significantly in the last few iterations. We put together the details of our genetic algorithm that we discussed in Algorithm 3.3, the optimization algorithm.

ALGORITHM 3.3. Optimize-street-light

```
1: input criteria: termination
2: input fitness-function: f
3: output candidate-solution: optimum

4: N := 1000, PN := 0.1 · N
5: mutations := create-set(type: operator, values: {m_dr, m_tp, m_ar})
6: crossover := c_rr
7: population := create-list(type: candidate-solution,
       values: generate-candidates(strategy: random, count: N))
8: update-fitness(values: population, function: f)
9: while termination(for: population) = false do
10:     pool := select(from: population strategy: stochastic-acceptance, count: PN)
11:     while size-of(pool) > 0 do
12:         p_1, p_2 := remove(from: pool, strategy: uniform-selection, count: 2)
13:         o_1, o_2 := apply(operator: crossover, to: {p_1, p_2})
14:         update-fitness(values: {o_1, o_2}, function: f)
15:         add(values: {o_1, o_2}, to: population)
16:     end while
17:     pool := select(from: population strategy: uniform-selection, count: 3 · PN)
18:     while size-of(pool) > 0 do
19:         for all operator: mutator in mutations do
20:             candidate-solution := remove-first(from: pool)
21:             o := apply(operator: mutator, to: candidate-solution)
22:             update-fitness(values: candidate-solution, function: f)
23:             add(values: {o}, to: population)
24:         end for
25:     end while
26:     sort(values: population, order: descending, by: fitness)
27:     remove(from: population index-range: [N + 1, last-index-of(population)])
28: end while
29: return optimum := select-first(from: population)
```

3.5 Application

A partner firm, Wemont Kft., requested us to develop and implement an algorithm for the purpose of automatic LED streetlight design. The genetic algorithm configured for customer's specific needs, that we presented in this chapter, was the designing engine in the software solution we handed over at the end of the development project.

Streetlight product	Pole height	Average illuminance	Illuminance uniformity
Aledin72	9	20	0.79
Aledin64	9	19.4	0.76
Aledin56	9	17.5	0.80
Aledin48	9	14.6	0.81
Aledin40	9	12	0.82
LB-RM370-W036	10	10	>0.5
LB-RM500-W056	10	18	>0.5
LB-RM630-W098	10	20	>0.5
LB-RS660-W100	10	18	>0.5
LB-RS710-W060	4	12	>0.5

Table 3.2: The first five products are designed by our algorithm, the second five products are from Lighting Best International Limited. The column of illuminance uniformity shows the ratio of the minimum and average illuminance of the light patterns. Source: `http://lightingbest.gmc.GlobalMarket.com`, and `http://www.wemont.hu`, accessed 19 May 2017.

The evaluation of our algorithm was difficult. Companies and academic research groups frequently publish design improvements but never design tools. Design methodologies are kept secret, the available commercial software in the field of public lighting focuses on the visualization of light plans,

therefore we were not able to compare our algorithm to other known methods. Another way to measure the quality of our programmatic designs could be the comparison with commercial LED streetlight products. Unfortunately, published specifications usually do not contain light pattern quality metrics, only power consumption data. We have found only one company that provided a common uniformity metric for their light patterns. Figure 3.2 shows some products for comparison.

#	Width	Length	LED	Illuminance	Runtime
1.	30m	10m	27	6 LUX	269 sec
2.	30m	10m	48	10 LUX	391 sec
3.	35m	20m	38	6 LUX	651 sec
4.	40m	20m	68	8 LUX	1,410 sec
5.	50m	25m	100	8 LUX	2,673 sec

Table 3.3: Five design test cases for street sections with different width and length. The column LED displays the number of LED lights in the optimized designs while the column illuminance contains the expected average illuminance values for the different scenarios. Runtime is the time that the sofware spent running the genetic algorithm.

The other motivator to develop the designer tool was to shorten the design time of LED streetlights. Table 3.3 shows some simple lighting scenarios. We ran the software solution for each test case on an average laptop with Intel Core I3-370M processor. The whole design process, including the setup of the lighting scenario at the start and the design export at the end, never took more time than an hour while the partner's engineers spent at least 4 hours on average to manually create designs with the help of a light modeling and visualization software. Moreover, our genetic algorithm designed light patterns of at least twice better quality than the manual ones. A few prototypes were produced and installed. Their light patterns are more homogeneous lighting the target surface and only the target surface. You can

see in Figure 3.7 that private properties are left in the dark, only the road and the sidewalk are lighted.

Figure 3.7: The genetic algorithm designed streetlights that light only the targeted public area. The light pattern is almost perfectly cut off along the edges. Source: `http://www.wemont.hu/activity/led-es-kozvilagitas`, accessed 19 May 2017.

Another common problem of public lighting is the zebra pattern, or zebra effect shown in Figure 3.8. The significantly dominant illuminance difference between the border sections and the centers of light patterns results in an alternating series of brighter and darker stripes on the road surface. Our genetic algorithm with the right parametrization of the fitness function can dampen this phenomenon as you can see an example in Figure 3.9.

Figure 3.8: An example for the zebra effect. Source: `http://www.klightled.com/Proinfo.php?id=126&pid=16&sid=0`, accessed 19 May 2017.

Figure 3.9: Reduced zebra effect with rigorous lighting uniformity requirements. Source: `http://www.wemont.hu/activity/led-es-kozvilagitas`, accessed 19 May 2017.

We tested the capabilities of the algorithm on more difficult scenarios

when the expected average illuminance was ranged from 10 to 30 LUX. Only considering the quality threshold for the fitness function, hard conditions were fulfilled after half an hour in all test cases, high quality solutions were provided after roughly 3 or 4 hours. Reimplementing the light pattern calculation and moving it to the GPU from the CPU, we managed to decrease the runtime drastically and turned the hours into minutes making even these unrealistic scenarios viable for live demonstrations in front of possible customers. For details, see [46].

3.6 Summary

We studied a very special covering problem and a stochastic solution in this chapter. We were searching for an answer to a very practical question, how to design streetlights with LED technology that provide better lighting than incandescent streetlights or other LED streetlights already available in the market. This is a multi-objective, global optimization problem with high dimensionality. Our proposal was a stochastic approach, a genetic algorithm. The essence of our approach was a geometric crossover concept being able to combine partially good parts of different designs based on the light pattern. We used a configurable fitness function that allowed us to set the relative importance of the different requirements. The computationally most expensive part of the fitness function was a grid-based, custom light pattern calculation method that took advantage of any symmetries present in the pattern in order to reduce the number of grid vertices that must be handled.

The genetic algorithm was the engine of a stand-alone designer software that completely automated the creation of streetlight designs providing much better light pattern quality than the commercial competitors demonstrating the main advantage of our solution. After the completion of the software solution, we moved the light pattern calculation to the GPU as we were experimenting on unrealistic lighting scenarios. We managed to reduce the optimization time of 3-4 hours to 10-20 minutes even for such extreme test cases.

Chapter 4

Searching chaotic trajectories

Modeling is a common practice to study natural phenomena and different processes. Researchers have to consider two, conflicting goals when they are building a model. It must be complex enough to express the problem in the required detail, but this complexity may not go beyond the capabilities of the solution methods and tools that are at their disposal.

Modeling dynamic systems demonstrates this difficulty well. We usually model them by differential equation systems whose solutions are the possible trajectories of the studied system. These solutions are numeric approximations in most cases, and their precision depends heavily on the numeric stability of the equations, the possible presence of chaotic solutions in the mathematical sense. Although even the simple models can provide answers to interesting questions of such systems, we must always be careful when we approximate solutions or carry out simulations due to the accumulated rounding errors and the nature of mathematical chaos.

Mathematical chaos is hard to recognize and to give actual chaotic solutions is even more difficult. Researchers have only managed to provide the theoretical proof of existence in the great majority of publications. Computer assisted proofs in chaos detection were early adopted with success [58, 65] and have brought more and more results since then, (computer-assisted proofs for embedded horseshoes [52], strange attractors [79], homoclinic tangencies [86], Hénon systems [7], etc.), though a lot of them did not apply rigorous com-

putations like [80] on chaos indicators (fractal dimension, recurrence plot, Lyapunov exponents, etc.).

The forced damped pendulum is a simple dynamic system, not just in construction but in the describing differential equation system as well, and differs exclusively from its normal version in a constantly changing but deterministic, external force affecting the pendulum. This little addition ensures interesting behavior to the system that is worth to study. John H. Hubbard published a thorough research about the forced damped pendulum and ended the paper [35] with his conjecture that chaotic trajectories might exist beside the periodic ones. More papers followed Hubbard's work about this topic, however the conjecture remained open. The paper of Bánhelyi *et al.* [8] brought the break-through in this matter after almost a decade, in which he proved the existence of chaotic trajectories with his colleagues, but the starting state of these trajectories were still missing.

In this chapter, we are going to present an optimization method that is able to locate the chaotic trajectories with prescribed, finite geometrical behavior. It means that we can find the starting state of the pendulum from which it will go through a specified, finite sequence of gyrations. This behavior prescription can be arbitrary long theoretically, but the precision of the applied computations set an actual limit to its length. The optimization algorithm is a hybrid algorithm from the reliability point of view. We use *GLOBAL*, that we are going to discuss in detail in Chapter 5, to optimize a special objective function.

Studying the forced damped pendulum was one of the many interesting problems that Balázs Bánhelyi, my supervisor, showed me. Locating given chaotic trajectories was left as an open problem from his own thesis. I got familiar with the topic walking in his steps. We had been discussing the problem a lot before I was able to conceptualize a working approach by myself, but the solution I visioned turned out to be feasible later, therefore I consider the optimization algorithm presented in this chapter and its proof of correctness to be my own results. Balázs Bánhelyi helped me in the implementation of the algorithm and the generation of numerical data as he had great experience with the related programming libraries but most of the

coding was done by me. We published the algorithm and results in the paper [45].

4.1 Basic definitions

First, we establish the theoretical common ground to be able to discuss our algorithm, most importantly, we have to clarify what chaos, and chaotic behavior are as they do not mean randomness unlike what we usually mean by them on a day to day basis. They rather mean unpredictability. Chaos is difficult to define in a practical way. As we primarily study dynamic systems from this point of view, it is much more beneficial, and thus widespread among researchers, to look for certain properties of dynamic systems that indicate the presence of chaotic behavior.

First, we define chaos in general. Let H be an arbitrary set and f a mapping $H \mapsto H$.

Definition 4.1.1. *f is **sensitive to its initial state** if there exists a $\delta > 0$ such that for all $x \in H$ and for all N_x neighborhood of x, there is an $y \in N_x$ and an $n > 0$ such that $|f^n(x) - f^n(y)| > \delta$.*

The sensitivity of the mapping f to its initial state informally means that any two points of H can be arbitrary close to each other, but iterating the mapping for these points might result in significantly different values.

Definition 4.1.2. *f is **topologically mixing** if for all open sets $I, J \subset H$, there is an $N > 0$ such that for all $n > N$ $f^n(I) \cap J \neq \emptyset$.*

This property means that iterating enough times the mapping f for any real, open subset of H, the result will overlap, or mix in other words, with any other real open set of H.

Definition 4.1.3. *A set I is **dense** in a set J if $\bar{I} \subset J$, where $\bar{I}$ is the closure of I, $\bar{I} = I \cup \{$limit-points-of$(I)\}$.*

The density of the set I in the set J implies that every neighborhood of every point in J contains at least one point from I.

Definition 4.1.4. *An $x \in H$ point is a* ***periodic point of the mapping*** *f if there is an $n > 0$ integer such that $f^n(x) = x$.*

Simply put, a point is periodic if applying the mapping repeatedly to the point and the returned values, the point itself will appear in the resulting sequence regularly.

Based on the Definitions 4.1.1, 4.1.2, 4.1.3, and 4.1.4, the general definition of chaos is the following.

Definition 4.1.5. *A mapping $f : H \mapsto H$ is chaotic if f is sensitive to its initial state, it is topologically mixing, and the set of periodic points of f is dense in H.*

There are also other interpretations of chaos and chaotic behavior beyond the above definition. We are going to use the definition of combinatorial chaos instead that fits much more our discussion.

Definition 4.1.6. *Let P be a plane, and L and R two regions of P, (open, connected, and non-empty subsets). Additionally, let us denote the bi-infinite L/R sequences by*

$$\ldots, \epsilon_{-2}, \epsilon_{-1}, \epsilon_0, \epsilon_1, \epsilon_2, \ldots,$$

where $\varepsilon_k \in \{L, R\}$. A continuous $\varphi : P \mapsto P$ mapping is chaotic in the region A of P if there is a $p \in A$ point for any arbitrary bi-infinite, L/R sequence such that

$$\ldots, \varphi^{-1}(p) \in \epsilon_{-1}, \varphi^0(p) \in \epsilon_0, \varphi^1(p) \in \epsilon_1, \ldots,$$

holds.

The classic, combinatorial chaos definition, that we introduced above, uses two regions, but it can be extended for any finite number of regions.

Now that we established what we mean by chaos for the rest of the chapter, we continue with some key definitions, and concepts that we will refer later to discuss the chaotic behavior of the forced damped pendulum.

Let φ be a continuous, differentiable, $\mathbb{R}^2 \mapsto \mathbb{R}^2$ mapping such that

$$\varphi(x_1, x_2) = (f(x_1, x_2), g(x_1, x_2))$$

holds for some functions f and g.

Definition 4.1.7. *A point $\overline{x} = (x_1, x_2)$ is the fixed point of φ if $\varphi(\overline{x}) = \overline{x}$ holds.*

Definition 4.1.8. *The Jacobian matrix of φ at $\overline{x}$ is a matrix constructed from all the first-order partial derivatives of φ in the following way:*

$$J_{\varphi}(\overline{x}) = \begin{pmatrix} \frac{\delta f}{\delta x_1}(\overline{x}) & \frac{\delta f}{\delta x_2}(\overline{x}) \\ \frac{\delta g}{\delta x_1}(\overline{x}) & \frac{\delta g}{\delta x_2}(\overline{x}) \end{pmatrix}.$$

Statement 4.1.9. *A fixed point $\overline{x}$ of φ is stable if the absolute values of the eigenvalues of the Jacobian matrix $J_{\varphi}(\overline{x})$ are strictly less than one.*

This means that a stable fixed point $\overline{x}$ has a neighborhood of $\varepsilon > 0$ radius such that

$$\lim_{n \to \infty} \varphi^n(\hat{x}) = \overline{x}$$

holds for all $\hat{x}$ in the neighborhood.

Statement 4.1.10. *A fixed point $\overline{x}$ of φ is unstable if the absolute value of at least one of the eigenvalues of the Jacobian matrix $J_{\varphi}(\overline{x})$ is strictly greater than one.*

On the analogy of the stable fixed points, an unstable fixed point $\overline{x}$ has a neighborhood of $\varepsilon > 0$ radius, and a threshold $\delta > 0$ such that for all $\hat{x}$ in the neighborhood there is some n that

$$|\varphi^n(\hat{x}) - \overline{x}| > \delta$$

holds.

Another important concept, that will be essential, is the Poincaré map of a dynamic system. Let us consider an n-dimensional, deterministic, dynamic system that can be described with a solvable differential equation system.

Definition 4.1.11. *A trajectory belonging to a starting point is the collection of all the solutions of the differential equation system using the starting point as initial condition.*

Definition 4.1.12. *Let S be an $(n-1)$-dimensional hyperplane, called a Poincaré section, such that no trajectory of our dynamic system is parallel to it. The Poincaré map, or first recurrence map, of our dynamic system will be a mapping that maps every point $s \in S$ to the first intersection point with S of the trajectory starting from s.*

It can be proved that a chaotic Poincaré map implies that the dynamic system for which it is defined is chaotic as well, see the book [85], therefore we can study the Poincaré map instead of the original system to detect chaos. We will follow the same course of action to locate the starting points of trajectories. Instead of using the original differential equation system, we will search for them based on their Poincaré mappings using the appropriate Poincaré sections.

4.2 The forced damped pendulum

The forced damped pendulum is a simple dynamic system of one degree of freedom. It consists of a mass point of mass m hung with a weightless solid rod of length l whose other end point is fixed. The mass point is forced to move along a vertical circle of radius l as illustrated in Figure 4.1. The motion of the pendulum, and the name of the system, comes from three forces, the gravitational force of acceleration g, an external, periodic force with intensity $A\cos t$ where A is constant and t is the time, and the dampening friction proportional to the velocity.

Considering these forces, the following second order differential equation describes the behavior of the forced damped pendulum:

$$mlx''(t) = -mg\sin x(t) - \gamma l x'(t) + A\cos t,$$

where $x(t)$, $x'(t)$, and $x''(t)$ are the angle, angular speed, and angular velocity in that order, and γ denotes the damping coefficient. The angle of the pendulum is defined to be 0 at the lowest possible position of the mass point, and it increases counterclockwise.

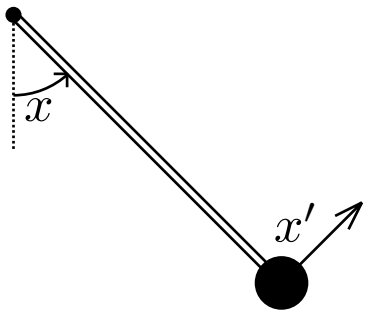


Figure 4.1: Illustration of the forced damped pendulum. x denotes the angle of the pendulum, and x' is angle velocity.

Let us use the same values for the parameters that Hubbard used in his paper [35]:

$$A = g, \quad l = g, \quad m = 1, \quad \gamma = 0.1$$

simplifying the previous equation to

$$x''(t) = \sin x(t) - 0.1x'(t) + \cos t.$$

As a last step, let us introduce a simple substitution of notions replacing x, the angle, with x_1 and x', the angular velocity, with x_2. The result will be the differential equation system that we will use in the rest of the chapter, and it can be easily studied with Poincaré maps:

$$\begin{aligned} x_1^{'}(t) &= x_2(t), \\ x_2^{'}(t) &= \sin(x_1(t)) - 0,1x_2(t) + \cos(t). \end{aligned} \tag{4.1}$$

The state of the pendulum is the vector (x_1, x_2), the angle and angular speed, in any time t. The trajectory of the pendulum is the path of the mass point during the motion of the pendulum started from a given initial state. Formally, a trajectory is the set of all solutions of this system on the initial conditions of $x_1(0) = \hat{x_1}$ and $x_2(0) = \hat{x_2}$ where $(\hat{x_1}, \hat{x_2})$ is the initial state of the pendulum.

We are going to study these trajectories based on the Poincaré sections of the x_1-x_2 plane at the moments of $t = 2n\pi$ where n is an integer in sync

with the period of the external force. We only have to iterate the following Poincaré map to achieve this:

$$P : R^2 \to R^2, \quad (x_1(0), x_2(0)) \mapsto (x_1(2\pi), x_2(2\pi)).$$

The Poincaré mapping for all $2n\pi$ moments can be calculated from the previous Poincaré mapping of $2(n-1)\pi$ started from the initial condition. The fixed points of this map are the 2π periodic solutions of the differential equations. The forced damped pendulum has two periodic orbits. The fixed point that has $x_1 = 0$ is called the stable fixed point or lower equilibrium point, and the other one that has $x_1 = \pi$ is called the unstable fixed point or upper equilibrium point. The names come from the simple mathematical pendulum because it can be stable exclusively in these states for an arbitrary long time.

Let us divide time into fix-length intervals denoting the time interval $[2k\pi, 2(k+1)\pi]$ by I_k. We will study the trajectories based on what happens in these intervals focusing on three specific motion:

- motion $\ominus$: the pendulum goes through the lower equilibrium point clockwise exactly once.
- motion $\otimes$: the pendulum does not go through the lower equilibrium point.
- motion $\oplus$: the pendulum goes through the lower equilibrium point counterclockwise exactly once.

Of course, the forced damped pendulum can move also other ways beyond the previous three motions during these 2π long time intervals, but these trajectories will not be important from the chaos point of view.

After we covered the necessary concepts, we can interpret Definition 4.1.6 for the forced damped pendulum. Let us consider the motions as sets of points on the $x_1 - x_2$ plane. Each set consists of exactly those points, or states in other words, at which the pendulum can be after performing the defining motion of the set. What Hubbard conjectured, that Bánhelyi et al.

later proved in [8], is that for every bi-infinite sequence $\ldots, e_-1, e_0, e_1, \ldots$, where $e_k \in \{\ominus, \otimes, \oplus\}$ for all integer k, there is an initial state $\hat{x} = (x_1, x_2)$ of the pendulum from which the pendulum will execute exactly the e_k motion in the I_k time interval. That means

$$\ldots, P^{-1}(\hat{x}) \in e_{-1}, P^0(\hat{x}) \in e_0, P^1(\hat{x}) \in e_1, \ldots$$

holds, therefore the forced damped pendulum has chaotic trajectories, see an illustration in Figure 4.2. A more tangible interpretation of the chaotic behavior of the pendulum is that every ϵ neighborhood of the initial state of any chaotic trajectory has at least one state from which the trajectory of the pendulum will differ arbitrary long after some time.

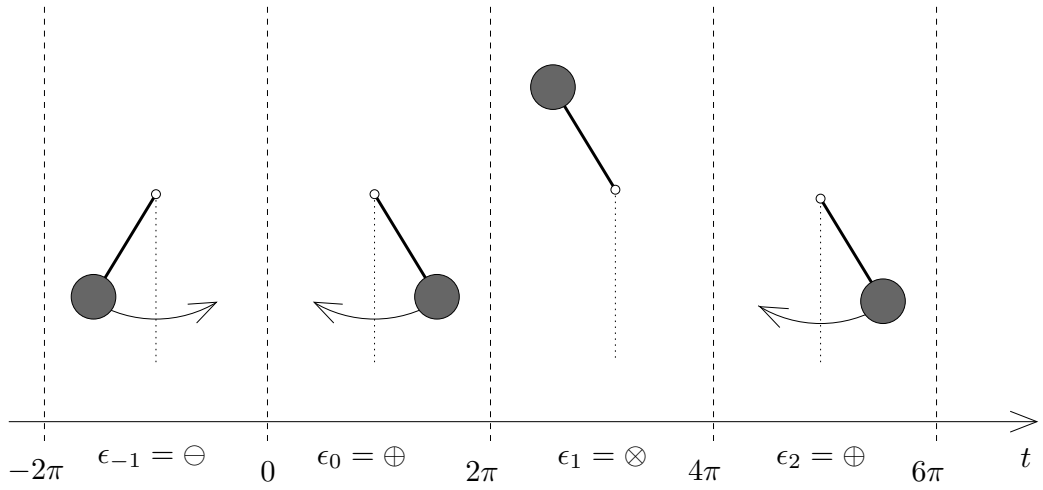


Figure 4.2: A section of a chaotic trajectory of the forced damped pendulum that contains four consecutive allowed motions.

Our goal is to find those $\hat{x} = (x_1, x_2)$ initial states of these chaotic trajectories; more precisely, for any given finite, but arbitrary long prescription of motions, we want to find initial states from which the pendulum would move as it is given. In this context, the prescription of motions means fixing the values of ϵ_k elements. It is important to remark that we can prescribe *past* motions too, not just future motions as we can calculate the inverse of the Poincaré map as well.

4.3 Searching algorithm

First, it is beneficial to dedicate some time to understand and study the structure of the state plane before constructing any optimization model. The plane (x_1, x_2) is similar to the Cantor ternary set that is defined by the following iteration. Let C_0 be the $[0, 1]$ interval, and the C_k interval is determined by the following formula for all positive integer:

$$C_k = \frac{C_{k-1}}{3} \cup \frac{2 + C_{k-1}}{3}.$$

The Cantor ternary set $\mathcal{C}$ is the intersection of these sets,

$$\mathcal{C} = \bigcap_{k=1}^{\infty} C_k.$$

The set can be constructed geometrically as well when we delete the open, middle third part of the $[0, 1]$ interval, and we repeat the process with the parts of the result continuing ad infinitum, see Figure 4.3 for illustration.

Figure 4.3: The first six iteration of the geometrical generation of the Cantor ternary set.

The topological relation of the initial states of the different chaotic trajectories are the same as the relation of the different iterations of the geometric creation of the Cantor ternary set. The set of trajectory initial states from which the first motion is an $\otimes$ motion for example contains all the initial states from which the pendulum will first perform a $\otimes$ motion and then an $\ominus$, or $\oplus$, or another $\otimes$ as it is presented on Figure 4.4. If we repeatedly add further expected motions to a prescription, the sets of corresponding initial states will form a continuously shrinking series of sets whose any element is contained by all the previous ones. The limit of the iteration is a set of Lebesgue measure 0. We can take advantage of this property when we

are searching for trajectories with longer prescriptions of motions. First, we can consider only the first motion. Then, we can narrow the search for the vicinity of the first result.

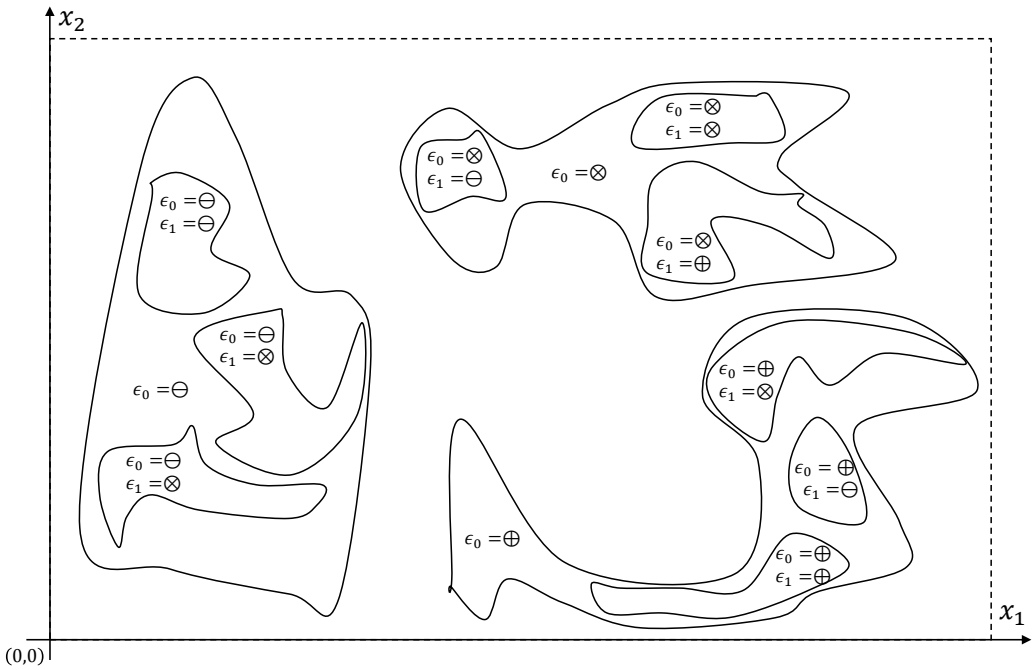


Figure 4.4: An illustration of how the initial states of trajectories could be located on the $x_1 - x_2$ space.

Our approach to find the initial states of trajectories performing given motions is a constrained global optimization model based on an objective function that measures how far the path of a given trajectory is from the expectations. The algorithm itself can be any global optimization method capable of minimizing this objective function over the x_1-x_2 plane.

Reliable computation is a cross-cutting requirement during the whole search. As we are handling chaotic initial states, results affected by rounding errors are quite useless. Interval arithmetic counters this problem if we use intervals with computer representable bounds for the inclusion of floating point numbers similarly how we did in Chapter 2. The challenge is that we must compute the interval inclusion of the whole fixed section of the studied trajectories to be able to check whether it follows the expected path.

The same rigorousness, using computer representable bounds, applies to this inclusion as well. We based our implementation on the VNODE algorithm [56] and on the PROFIL/BIAS interval environment [42], you can find some details in [45].

The list of expected motions is checked element by element to evaluate a given initial state and the corresponding trajectory. We calculate the inclusion of the trajectory for each I_k interval and examine the angle and angular speed values in each interval composing the inclusion. Naturally, the Poincaré map after the I_1 interval is always applied to the result of the previous mapping. It can happen that the angle of a mapping is out of the interval $[0, 2\pi]$. We always transform the angle back into this interval in such cases to keep things simple at the end when checking each I_k interval.

All three motions can be distilled into requirements that define the types of regions of the plane x_1-x_2-t where the trajectory is expected to go, hence we will refer them as expected region types and denote them as $E_\otimes$, $E_\ominus$, and $E_\oplus$ for the motions $\otimes$, $\ominus$, and $\oplus$, respectively.

Definition 4.3.1. *The expected region type $E_\otimes$ of the motion $\otimes$ consists of the regions having the largest possible area for which $0 < x_1 < 2\pi$ holds.*

Definition 4.3.2. *The expected region type $E_\ominus$ of the motion $\ominus$ consists of the regions having the largest possible area for which $-2\pi < x_1 < 2\pi$ and $x_2 < 0$ hold for some time period T_{int}, when the trajectory first and last intersects the plane $x = 0$, and $0 < x_1 < 2\pi$ holds for the time period before T_{int} while $-2\pi < x_1 < 0$ holds for the time period after T_{int}.*

Definition 4.3.3. *The expected region type $E_\oplus$ of the motion $\oplus$ consists of the regions having the largest possible area for which $0 < x_1 < 4\pi$ and $x_2 > 0$ hold for some time period T_{int}, when the trajectory first and last intersects the plane $x = 2\pi$, and $0 < x_1 < 2\pi$ holds for the time period before T_{int} while $2\pi < x_1 < 4\pi$ holds for the time period after T_{int}.*

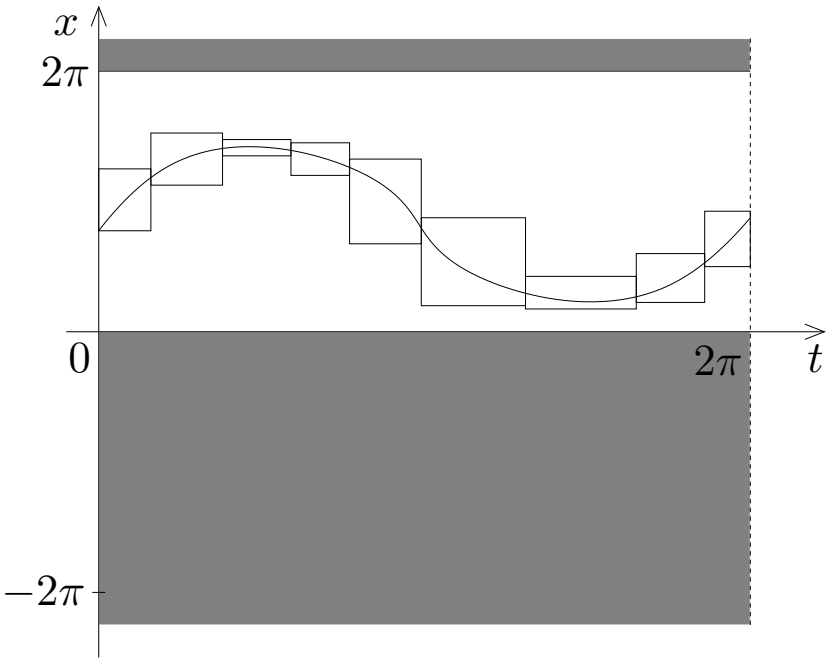


Figure 4.5: Illustration of the expected region where the inclusion of trajectory must fit the requirements of the motion $\otimes$. x denotes the angle of the pendulum, and t denotes the time.

The definition of $E_{\otimes}$ regions is straightforward and simple as you can see it in Figure 4.5. The case of $E_{\ominus}$ and $E_{\oplus}$ regions are a bit more complicated. These expected regions have three logically different parts like that in Figure 4.6. They have an intersection region that contains the inclusion intervals that overlap with the $x = 0$ plane, when the pendulum goes through the lower equilibrium point, and parts that come before and after the intersection. The $x_2 < 0$ and $x_2 > 0$ conditions are necessary to ensure that the pendulum does not turn back and crosses the lower equilibrium point multiple times, only once. We might discard valid trajectories this way during our search, but that will not cause any problems. Trajectories may go through the lower equilibrium point in the same direction more than once during an I_k time interval. The definition of expected region types $E_{\ominus}$ and $E_{\oplus}$ consider only the first valid crossing and the others as the violation of the prescribed motion.

The following lemmas connect the prescribed motions, expected regions,

and trajectories, that trivially follow from the previous definitions.

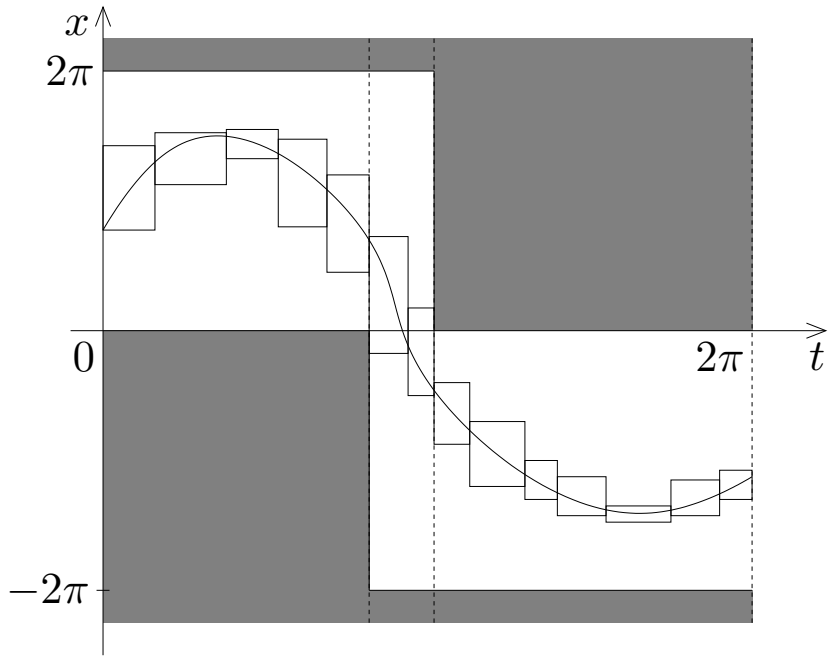


Figure 4.6: Illustration of the expected region where the inclusion of trajectory must fit the requirements of the motion $\ominus$. x denotes the angle of the pendulum, and t denotes the time.

Lemma 4.3.4. *If all inclusion intervals of a trajectory belonging to the time interval I_k are in an $E_\otimes$ type region, and the Poincaré mapping of the trajectory entry point of I_k has an angle in the interval $(0, 2\pi)$, then the trajectory performs the motion $\otimes$ during I_k.*

Lemma 4.3.5. *If all inclusion intervals of a trajectory belonging to the time interval I_k are in an $E_\ominus$ type region, and the Poincaré mapping of the trajectory entry point of I_k has an angle in the interval $(-2\pi, 0)$, then the trajectory performs the motion $\ominus$ during I_k.*

Lemma 4.3.6. *If all inclusion intervals of a trajectory belonging to the time interval I_k are in an $E_\oplus$ type region, and the Poincaré mapping of the trajectory entry point of I_k has an angle in the interval $(2\pi, 4\pi)$, then the trajectory performs the motion $\oplus$ during I_k.*

We defined the cases when we accept a trajectory section of an I_k interval as one the three expected motions. The next step is to create a measure for the extent of differing from the prescribed motions. We found the *Hausdorff distance* to be the best for the purpose from the distance alternatives. The Hausdorff distance defines the distance of two, non-empty subset of an arbitrary metric space as the maximum of the distances of a point from one subset measured from the closest point to it from the other subset. This concept is symmetric to its operands, but we only wish to describe how far the trajectory inclusions reach out from expected regions, thus we will use the following, slightly modified, asymmetric version of Hausdorff distance.

Definition 4.3.7. *The Hausdorff distance of the inclusion of the trajectory during the time interval I_k, denoted by T_k, and the expected region of any type, denoted by E_k, is*

$$H(T_k, E_k) = \max_{x \in T_k} \inf_{y \in E_k} d(x, y),$$

where d is the Euclidean distance.

Henceforth in the chapter, we always mean the above definition whenever we refer to the Hausdorff distance. Its minimal value is 0, thus the future global optimum can be verified with ease, and it proportionally describes the extent of the difference between the expectations and the given trajectory. The only tradeoff is the complexity of computation compared to other, simpler distance concepts like the plane Euclidean distance, though the actual calculation of it is not difficult as we measure the distance of rectangles and composites of rectangles.

ALGORITHM 4.1. Overhang

1: **input** trajectory: T
2: **input** float: *upper-bound*, float: *lower-bound*
3: **output** float: *overhang*

4: *upper* := maximum-of(upper-bound-of(x_1-of(inclusion-of(T)))) − *upper-bound*
5: *lower* := *lower-bound*−minimum-of(lower-bound-of(x_1-of(inclusion-of(T))))
6: **return** *overhang* := maximum-of($\{upper, lower, 0\}$)

Our basic problem is to calculate the extent of how much the trajectory inclusion hangs over a rectangle. Algorithm 4.1 solves this task in a few steps.

ALGORITHM 4.2. Hausdorff-$\otimes$

1: **input** trajectory: T
2: **output** float: *distance*

3: **return** *distance* := Overhang(trajectory: T, upper-bound: 2π, lower-bound: 0)

The Hausdorff distance of the trajectory inclusion and the expected region in case of motion $\otimes$ is an overhang calculation using the upper and lower bounds of 2π and 0 for the pendulum angle, see Algorithm 4.2.

ALGORITHM 4.3. Hausdorff-$\ominus$

1: **input** trajectory: T
2: **output** float: *distance*

3: $first$:= find-first(in: x_1-of(inclusion-of(T)), holds: contains(value: 0))
4: *second* := find-first(in: x_1-of(inclusion-of(T)), holds: **not** contains(values: $[0, \infty)$))
5: $before$, *intersection*, *end* := cut(value: T, into: *parts*,
along: $\{first, second\}$)
6: h_b := Overhang(trajectory: $before$, upper bound: 2π, lower bound: 0)
7: h_i := Overhang(trajectory: *intersection*, upper bound: 2π, lower bound: -2π)
8: h_a := Overhang(trajectory: $after$, upper bound: 0, lower bound: -2π)
9: h_{x_2} := maximum-of({maximum-of(x_2-of(inclusion-of(h_i))), 0})
10: **return** *distance* := maximum-of(h_b, h_i, h_a, h_{x_2})

Determining the Hausdorff distance in case of motion $\ominus$ is not too difficult either. Basically, three overhang calculation and a possible penalty term for a positive angular velocity compose this case. Algorithm 4.3 summarizes the necessary operations. We handle the motion $\oplus$ on the analogy of the $\ominus$ case.

We optimize the following objective function in search for trajectories that comply the prescribed sequence of motions $\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_n$ for an arbitrary integer n.

Definition 4.3.8. *Henceforth, the function F denotes the objective function*

of trajectory searches, and it is defined as

$$F(x_1, x_2) = f(x_1, x_2) + c \cdot \mathrm{sgn}(f(x_1, x_2)),$$
$$f(x_1, x_2) = \sum_{k=0}^{n} H(T_k(x_1, x_2), E_k),$$

where $T_k(x_1, x_2)$ and E_k are respectively the inclusion of the trajectory started from the initial state (x_1, x_2) and the expected region during the time interval I_k while c is a fixed penalty term we use when the requirements of any of the prescribed motions are violated.

F is actually a penalty function whose value of 0 means the full compliance of the trajectory with the prescribed motions.

The theorem below phrases the correctness of the bound, constrained, global optimization model we defined, see the applied methods in [16] for comparison.

Theorem 4.3.9. *In case when any global optimization algorithm finds an initial state for which the function F has a value below c, then the trajectory started from this state will perform the entire prescription of expected motion, and meanwhile the algorithm provides a reliable computational proof of the respective properties for the trajectory.*

Proof. The first statement of the theorem directly follows from the definition of F, Definition 4.3.8, and the Lemmas 4.3.4, 4.3.5, and 4.3.6.

The second statement is the result of checking the interval inclusion of trajectories and constructing the interval bounds rounding outward to the nearest computer representable numbers. □

We chose the C implementation of the clustering, stochastic, global optimization method, GLOBAL [15] from the possible alternatives [70, 78] to optimize our objective function. The strength of GLOBAL is the ability to find the global optimum points of problems with moderate dimensions when the region of attraction of global optima are relatively not too small. We will discuss this method in the next chapter.

Theoretically, we are able to search for initial states of trajectories with arbitrary long series of prescribed motions with our model, but the number of expected motions are bound by the actual implementation. We used double-precision floating-point numbers that have their limits. If we recall the structural similarity of the search space and the Cantor ternary set, it becomes obvious that concatenating additional motions to the end of a series will continuously reduce the set of eligible trajectories reaching a point after the dimensions of the solution set will be smaller than the smallest gap between two representable double-precision floating-point numbers that means a practical limit to the theoretic, arbitrary length. This limitation can be countered if we use some kind of symbolic number representation that allows us to extend the length of expected series of motions to the value we deem necessary albeit at the cost of much more computation and thus longer optimization time.

4.4 Results

We started exploring the space of initial states by drawing the general layout of chaotic regions, how large are these areas, where are they located compared to each other. We set the prescribed behavior to $\varepsilon_0 = \ominus$, $\varepsilon_0 = \oplus$, and finally $\varepsilon_0 = \otimes$, and only evaluated the objective functions. Figures 4.7, 4.8, and 4.9 show the resulted graphs. The graph parts having $F(x_1, x_2) = 0$ are highlighted with red contour. Some areas are illustrated as collections of disconnected regions that is the result of the insufficient precision of the figures and the size of the area over F is illustrated.

We discussed that the state space of the forced damped pendulum has a structure like the Cantor ternary set. This means for example that the region containing the trajectories for the prescription $\varepsilon_0 = \ominus$ also contains the trajectories for the prescription $\varepsilon_0 = \ominus, \varepsilon_1 = \otimes$, and $\varepsilon_0 = \ominus, \varepsilon_1 = \otimes, \varepsilon_2 = \oplus$, etc. Therefore the chaotic starting of the three different motions can most likely be found in S-shaped areas very close to each other and in addition, larger areas in case of motions $\ominus$, and $\oplus$.

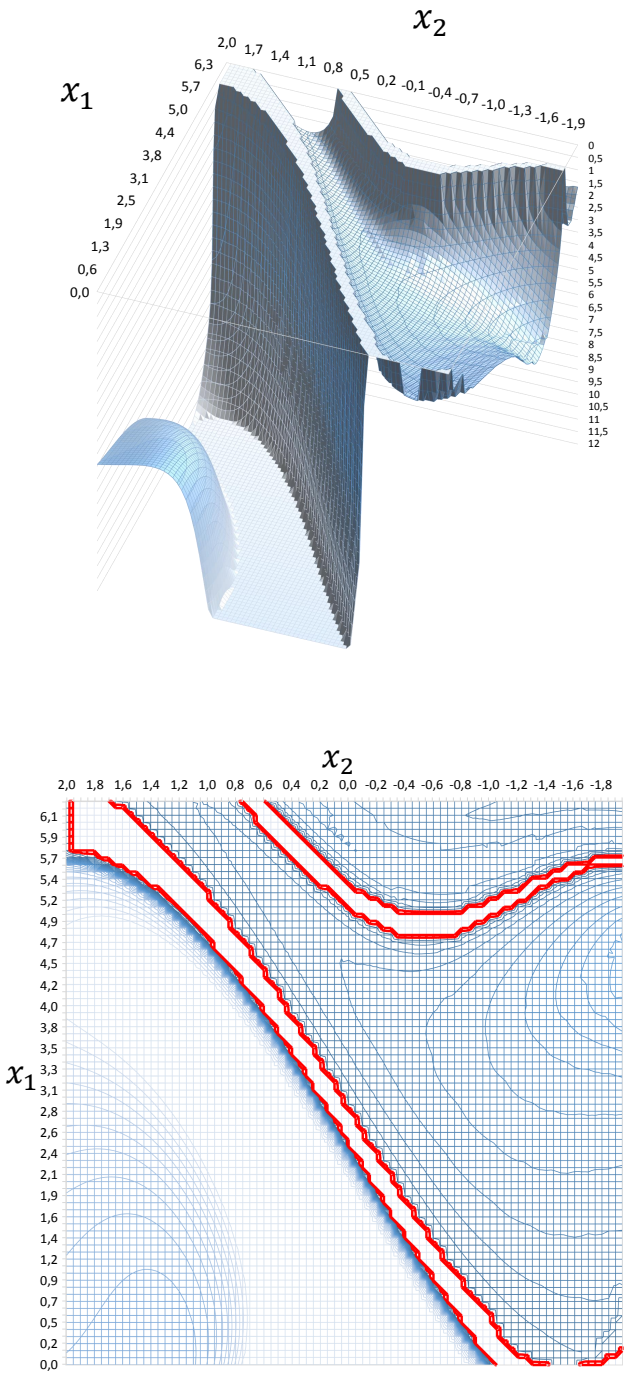


Figure 4.7: The graph of function F for the prescribed motion $\varepsilon_0 = \ominus$ in the area $x_1 \in [0, 2\pi]$, $x_2 \in [-2, 2]$ over a 40×40 grid.

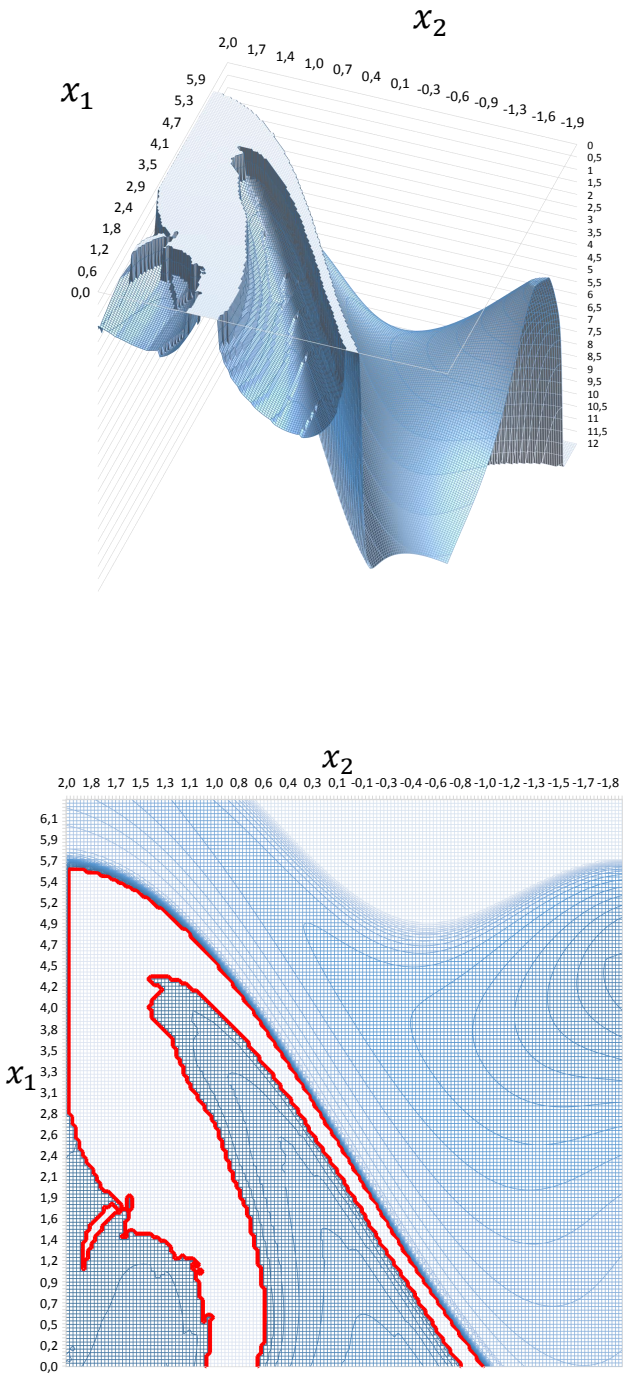


Figure 4.8: The graph of function F for the prescribed motion $\varepsilon_0 = \oplus$ in the area $x_1 \in [0, 2\pi]$, $x_2 \in [-2, 2]$ over a 40×40 grid.

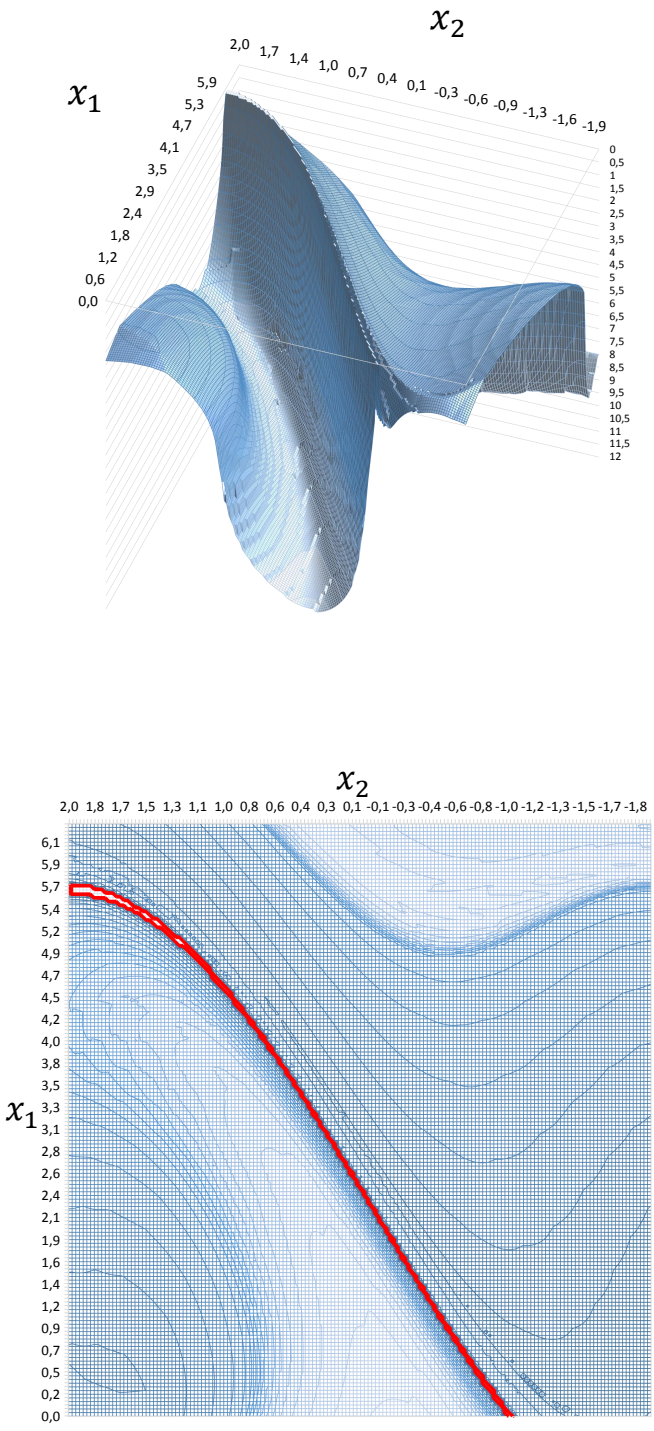


Figure 4.9: The graph of function F for the prescribed motion $\varepsilon_0 = \otimes$ in the area $x_1 \in [0, 2\pi]$, $x_2 \in [-2, 2]$ over a 80×80 grid.

It is important to remember that non chaotic trajectories are there as well. The relative sizes of these areas also suggest that the effort required to locate given trajectories have significant differences depending on the prescribed motions, the motion $\otimes$ seeming to be the greatest challenge based on Figure 4.9.

After having a brief look over the structure of the state space, we moved towards trajectories that fulfill more complex prescriptions. First, we tried to find initial states for every possible prescribed motion series of length three. We positioned the search areas for the optimization method based on the rectangles that were used in the existential proof of chaos in the paper [8]. The researchers proved that there is a rectangle shaped region on the plane x_1-x_2 for all studied motions that contains initial states for every chaotic trajectory that starts with the given motion. We were running the optimization over the interval inclusion of the appropriate rectangle, shown in Table 4.1. Depending on the first prescribed motion, we restricted the search to the relevant area to improve the possibility of finding trajectories for all kind of prescribed motion series.

First motion	inclusion of x_1	inclusion of x_2
$\varepsilon_0 = \ominus$	$[1.000, 2.226]$	$[-1.350, -0.208]$
$\varepsilon_0 = \otimes$	$[2.436, 2.796]$	$[-0.123, 0.201]$
$\varepsilon_0 = \oplus$	$[3.197, 4.412]$	$[0.389, 1.258]$

Table 4.1: The three search area of chaotic trajectories based on the first expected motion.

Table 4.2 presents the numerical results. Beside an example initial state for each prescription, the table contains the total number of found trajectory initial states, the number of executed objective function evaluations, and the total runtime of the searches in seconds. Some trajectories we found were outside of their search area. This happened due to the way how the algorithm GLOBAL works. It starts local searches from the search area, but these searches might end outside of bounds.

Prescribed motions	Example initial state	Found trajectories	Function evaluations	Run time
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \oplus\oplus\oplus$	$(3.5145566; 1.1854134)$	3	2 305	666
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \oplus\oplus\otimes$	$(3.541253; 1.1780008)$	1	3 356	965
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \oplus\oplus\ominus$	$(4.1354217; 1.1146838)$	9	1 489	431
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \oplus\otimes\oplus$	$(3.4500625; 1.2046848)$	1	3 057	862
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \oplus\otimes\otimes$	$(3.6355882; 1.1519576)$	1	7 229	2 089
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \oplus\otimes\ominus$	$(4.1873482; 1.1159454)$	2	2 477	723
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \oplus\ominus\oplus$	$(4.3271325; 1.1040739)$	3	2 968	858
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \oplus\ominus\otimes$	$(3.9656183; 1.0787189)$	2	3 212	931
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \oplus\ominus\ominus$	$(3.7628911; 1.096835)$	7	1 863	540
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \otimes\oplus\oplus$	$(2.6045829; 0.056101674)$	2	3 680	1 061
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \otimes\oplus\otimes$	$(2.6558599; 0.004679824)$	1	11 882	3 439
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \otimes\oplus\ominus$	$(2.5851486; 0.081902247)$	6	2 054	594
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \otimes\otimes\oplus$	$(2.6840309; -0.024118557)$	1	8 940	2 582
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \otimes\otimes\otimes$	$-$	0	8 885	2 573
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \otimes\otimes\ominus$	$(2.4871575; 0.17213042)$	1	2 782	803
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \otimes\ominus\oplus$	$(2.6099677; 0.043887032)$	1	2 347	678
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \otimes\ominus\otimes$	$(2.7034849; -0.050274764)$	2	5 313	1 537
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \otimes\ominus\ominus$	$(2.7717467; -0.11932383)$	5	2 078	600
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \ominus\oplus\oplus$	$(1.3103648; -0.45392754)$	8	1 568	451
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \ominus\oplus\otimes$	$(1.3957671; -0.73793841)$	1	5 063	1 464
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \ominus\oplus\ominus$	$(1.4709218; -0.63161865)$	12	1 055	305
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \ominus\otimes\oplus$	$(1.0277603; -0.20910021)$	1	2 294	669
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \ominus\otimes\otimes$	$(1.4672718; -0.61087661)$	1	9 873	2 869
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \ominus\otimes\ominus$	$(1.5116331; -0.66239224)$	2	2 404	695
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \ominus\ominus\oplus$	$(1.6396628; -0.62997909)$	7	1 527	447
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \ominus\ominus\otimes$	$(1.4479849; -0.5786194)$	2	2 303	665
$\varepsilon_0, \varepsilon_1, \varepsilon_2 = \ominus\ominus\ominus$	$(1.3920852; -0.37132957)$	12	1 131	327

Table 4.2: The optimization results for all possible combinations of three prescribed motions.

The results reflect what the previous function graphs implied. The optimization method revealed significantly more trajectories for the series containing only the motions $\ominus$ and $\ominus$ while it became more and more difficult to find trajectories, note the fewer initial states and the greater number of function

evaluations, as the number of $\otimes$ motions increased in the prescription. This manifested the most significantly for the prescribed motion series of $\otimes\otimes\otimes$ for which we found no complying trajectory.

The outcome of these searches drew our attention to the motion $\otimes$. What the searches revealed are not completely surprising. The Poincaré mapping of the forced damped pendulum might have an unstable fixed point at approximately $(2.634\ldots, 0.026\ldots)$. This is the only known region that is conjectured to contain the initial state of a chaotic trajectory for the bi-infinite $\otimes$ series, its verification is still an open question. The approximation of the eigenvalues of the Jacobian matrix at this point are $\mu_1 \approx 321.836\ldots$ and $\mu_2 \approx 0.001\ldots$. Such large eigenvalues as μ_1 are the indicators of outstandingly large numerical errors in the mapping approximations. This implies that the inclusion intervals of mapping approximations around this initial state grow extremely fast, so quick that the trajectory inclusion simply cannot stay in the expected region, as it happened and resulted in an unsuccessful attempt to find an initial state for the $\otimes\otimes\otimes$ prescription. The optimization algorithm, GLOBAL also sets us back a bit in handling the prescription series of motions $\otimes$ because this optimization solution works best when the region of attraction of local optimum points are relatively large. Unfortunately, the region of attraction around this particular initial state is very small to make things more difficult.

We repeated the search multiple times without success. As our last option, we focused the search region around the saddle point in order to increase the chance of finding trajectories for the three consecutive $\otimes$ motion, or for even longer $\otimes$ series. We decided to completely restart our study of this motion and systematically searched for trajectories that comply the prescribed series of as many consecutive $\otimes$ motions as we can handle started with the series of length one. We narrowed down the search area by one order of magnitude after each unsuccessful search using the the rigorous inclusion box of the saddle point

$$x_1 \in [2.634272, 2.634274]\,, x_2 \in [0.02604294, 0.02604485]\,.$$

Table 4.3 shows what we achieved by the new approach. Each row of the

Length of series	Search area	Found trajectories (example initial state)	Function evaluations	Run time
1	$\left[2.^{8}_{4}; 0.^{2}_{0}\right]$	12 (2.7108515; −0.030099507)	1 055	303
2	$\left[2.^{7}_{6}; 0.^{1}_{0}\right]$	4 (2.6469962; 0.013297356)	3 254	940
3	$\left[2.63^{5}_{4}; 0.02^{7}_{6}\right]$	1 (2.6342106; 0.026105974)	2 225	643
4	$\left[2.634^{3}_{2}; 0.026^{1}_{0}\right]$	1 (2.634273; 0.026043388)	2 491	721
5	$\left[2.63427^{4}_{2}; 0.02604^{4}_{2}\right]$	0 -	7 620	2 206

Table 4.3: Search results of prescribed motion series consisting of increasing number of motion $\otimes$.

table contains the search area, where the shorthand notions like $2.^{8}_{4}$ should be interpreted as the interval $[2.4, 2.8]$, how many trajectories we found with an example, and the number of objective function evaluations and runtime as in the previous table. This time, we managed to locate trajectories for the series of length three and even for a length four. The new limit of our capabilities was five consecutive $\otimes$ motions.

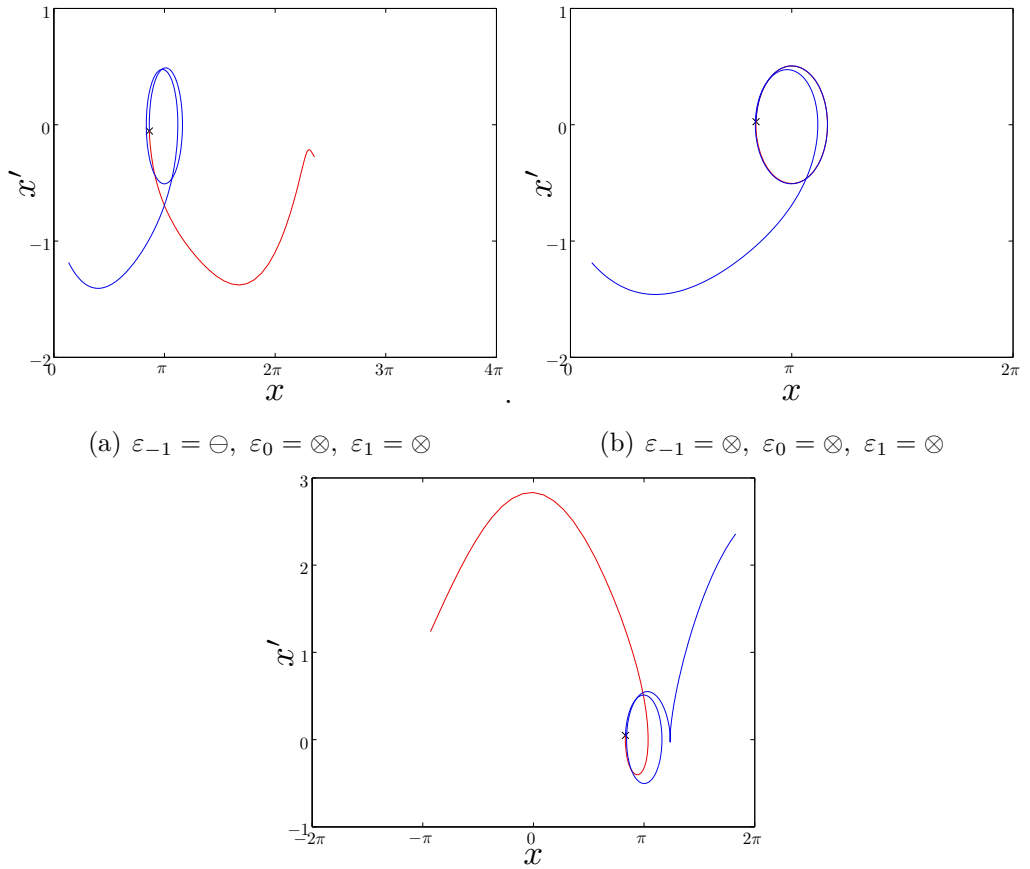
Our optimization based search method can examine the *past* motions of the forced damped pendulum as well, not just the future motions, as we can calculate the inverse Poincaré mapping and the trajectory inclusion that belongs to it as well. This means that we can add $\varepsilon_{-1}, \varepsilon_{-2}, \ldots$ prescription to the series too. As a final demonstration, we searched trajectories for the prescribed motion series

$$\varepsilon_{-1} = \ominus,\ \varepsilon_0 = \otimes,\ \varepsilon_1 = \otimes$$

$$\varepsilon_{-1} = \otimes,\ \varepsilon_0 = \otimes,\ \varepsilon_1 = \otimes$$

$$\varepsilon_{-1} = \oplus,\ \varepsilon_0 = \otimes,\ \varepsilon_1 = \otimes.$$

The result initial states were $(2.713676; -0.05424629)$, $(2.634245; 0.02601851)$, and $(2.612663; 0.04779419)$, respectively. Figure 4.10 shows the found trajectories in the state space.



(a) $\varepsilon_{-1} = \ominus,\ \varepsilon_0 = \otimes,\ \varepsilon_1 = \otimes$

(b) $\varepsilon_{-1} = \otimes,\ \varepsilon_0 = \otimes,\ \varepsilon_1 = \otimes$

(c) $\varepsilon_{-1} = \oplus,\ \varepsilon_0 = \otimes,\ \varepsilon_1 = \otimes$.

Figure 4.10: Trajectories with prescribed past motions on the x_1-x_2 plane. Red trajectory sections belong to the time interval I_{-1}, the crosses mark the initial states.

4.5 Summary

This chapter studied the forced damped pendulum, a chaotic dynamic system where the reliable representation of numeric values are essential. We built a global optimization based search method able to locate trajectories that moves according to a given prescription. These so-called chaotic trajectories of this system are the bi-infinite series of three types of motions. We achieved that through the minimization of an objective function uniquely defined for

the different prescriptions that expresses the measure of difference between where the trajectories actually go and where they should. Theoretically, our approach can find trajectories for arbitrary fixed length series of motions. In practice, the precision of the numeric representation used in the implementation limits the manageable length of such series, we experienced ourselves this during the trajectory searches.

Adequate tools can supplement the theoretical results well. In our case, the optimization method helped us to demonstrate a few preliminary results, to visualize and better understand the state space of the forced damped pendulum. With proper parametrization, we were able to reveal whole chaotic regions and the location of specific trajectories as well.

Chapter 5

Improvement of a stochastic global optimization method

In our modern days, conducting global optimization [31, 34, 40, 47, 62, 73] has become the part of the daily operation of almost all natural sciences. The scale and variety of problems are larger than ever ranging from planning public transportation to biological and chemical engineering [5, 6, 53, 54]. For a long time, the development and improvement of tools for these tasks are not just a simple challenge for researchers but an ever-growing expectation from industrial stakeholders and indirectly from the information society.

As more and more problems are yielded, good optimization tools are simultaneously valued higher and higher especially if they are flexible and capable of handling multiple problem types. The algorithm GLOBAL is such a tool, a stochastic optimization method aiming to solve non-linear, constrained optimization problems [15, 17]. It is versatile and proven to be competitive in multiple comparisons, recently in [17]. The usefulness of an optimization approach depends on implementation.

GLOBAL was not improved in the last decade. The algorithmic design and the first implementation [13] are coming from an era when the hardware meant a serious limitation that researchers had to constantly consider. The idea of reworking the algorithm came around during an industrial, precision designing task [18]. GLOBAL was the best tool to use, but we had to over-

come a lot of obstacles just to integrate its implementation into the software ecosystem, and though enough computing capacity was available to solve the optimization problem at hand within acceptable time, we became curious if we could improve the algorithm to be ready in case the parameters of the assignment changed. We have thus decided to revisit this optimization method and upgrade it in order to provide a new, better implementation to the scientific community that offers easier customization with more options and better performance than its FORTRAN, C, and MATLAB based predecessors.

I learned about GLOBAL first when I studied the forced damped pendulum. I considered it a black box back then and only gathered enough knowledge about the algorithm to be able to use it. I started to analyze GLOBAL in detail much more later after I talked to Balázs Bánhelyi about the increasing need and possible advantages of reimplementing the algorithm. First, I studied more thoroughly the work of Tibor Csendes and the latest implementation of the algorithm by László Pál. Based on the requirements of Balázs Bánhelyi, I modularized GLOBAL and reimplemented it in JAVA along a few improvements in the algorithm itself, therefore I count both the new implementation and the theoretical results presented in this chapter to be my own. I and Balázs Bánhelyi worked together on the evaluation of my work comparing the new implementation with the previous one of László Pál. We published my work as a part of a comprehensive book about GLOBAL [9] and also presented it in a conference [10] with Balázs Bánhelyi, Tibor Csendes, László Pál, and Dániel Zombori, who continued the work on the algorithm and created a parallel implementation of it also in JAVA.

5.1 The GLOBAL algorithm

GLOBAL is designed to solve constrained, non-linear optimization problems that have the following form. Formally, we consider the following global

minimization problem:

$$\begin{aligned} h_i(\overline{x}) &= 0 \qquad & i \in E \\ g_j(\overline{x}) &\leq 0 \qquad & j \in I \\ a \leq \overline{x} &\leq b \\ \hline \min_{\overline{x}} f(\overline{x}), & \end{aligned}$$

where we are searching for the global minimizer points of the n-dimensional, real function f. The equality and inequality constraints, h_i and g_j, and the lower and upper bounds of the argument x, vectors a and b, determine together the feasible set of points. If constraints are present, we shift to another optimization problem with a new objective function like

$$\min_{a \leq \overline{x} \leq b} F(\overline{x}),$$

where F has the same optimum points as f does, but it embeds a continuous penalty term that adds an increasing value to the original function proportional to the extent of violation of the equality and inequality constraints, for details see [72]. F is much more simple to handle as the remaining lower and upper bounds can easily be satisfied if candidate optimum points are generated within the allowed interval during the optimization.

GLOBAL can find optimum points in two ways, rarely when it generates possible starting points local searches, and generally during it is actually running local searches, that are computationally expensive operations, therefore it is important to start the searches from good candidates that promise relatively greater chance to find better optima.

The region of attraction of a local minimum point x^* is the set of points from which the local search will lead to x^*. As other stochastic algorithms, GLOBAL assumes that the relative size of the region of attraction of the global minimizer points is not too small, thus we have a chance to find a starting point within these regions of attraction that can be improved further by a good local search method.

We can reduce the number of unproductive searches if we identify which points belong to the same region of attraction. GLOBAL achieves this through continuous clustering. The algorithm iterates three steps, the stochastic sample generation from the interval $[a, b]$, the clustering of new samples, and running local searches from samples that could not be added to any of the existing clusters. The middle step is a heuristic that aims to determine the regions of attraction, the points belonging to the same cluster. The algorithm only starts searches from points that remained unclustered after this step. We emphasize that this clustering is not definitive in the sense that local searches from unclustered points can lead to already found optima, and points assigned to the same cluster might belong to different regions of attraction.

Algorithm 5.1 is the high-level description of GLOBAL. Sample points are generated from a uniform distribution over the interval $[a, b]$. The parameter N controls the number of new samples added to the pool in each iteration. The parameter λ is the so called reduction ratio that controls how many sample points, having a better objective function value than the others, should be kept and carried over to the next iteration.

GLOBAL clusters the reduced sample set using a modified single-linkage clustering. The original single-linkage clustering concept is an agglomerative, hierarchical approach. It starts considering every sample is a cluster on its own, then it iteratively joins the two clusters having the closest pair of elements in each round. As this is a local criterion, it does not take into account the overall shape and characteristics of the clusters, only the distance of their closest members matters. The GLOBAL single-linkage interpretation follows this line of thought. An unclustered point x is added to the first cluster that has a point with a lower objective function than what x has, and the point is at least as close to x as a predefined critical distance d_c determined by the formula

$$d_c = \left(1 - \alpha^{\frac{1}{N-1}}\right)^{\frac{1}{n}},$$

where n is the dimension of $xinF(x)$, and $0 < \alpha < 1$ is a parameter of the

clustering procedure. The distance is measured by the infinity norm. You can observe that d_c is adaptive meaning that it becomes smaller and smaller as more and more samples are generated.

ALGORITHM 5.1. GLOBAL

```
 1: input objective-function: F
 2: input search-space: [a, b]
 3: input criteria: termination
 4: input integer: N := 100, float: λ := 0.5
 5: input function: local-search
 6: output sample: opt-point := null, float: opt-value := ∞

 7: reduced := create-list(type: sample, values: empty)
 8: clusters := cluster-set(type: cluster, values: empty);
 9: samples := create-distribution(type: uniform, values: [a, b])
10: i := 1
11: while evaluate(condition: termination) = false do
12:     new := generate-samples(from: samples, count: N)
13:     sort(values: union-of({new, reduced}, based-on: F, order: ascending)
14:     reduced := select(from: reduced, index-range: [1, ⌊i · N · λ⌋])
15:     remove(from: elements-of(clusters), holds: not-in(reduced))
16:     clusters, unclustered := try-to-cluster(values: reduced, into: clusters)
17:     while size-of(unclustered) > 0 do
18:         x := pop(from: unclustered)
19:         x* := local-search(function: F, start-from: x, over: [a, b])
20:         clusters, unclustered := try-to-cluster(value: x*, into: clusters)
21:         if none-of(values: clusters, holds: contains(value: x*)) then
22:             add(value: create-cluster({x*, x}), to: clusters)
23:         else
24:             add(value: x, to: cluster-of(x*))
25:         end if
26:         add(value: x*, to: reduced)
27:     end while
28:     i := i + 1
29: end while
30: sort(values: reduced)
31: return opt-point := first-of(reduced), opt-value := F(first-of(reduced))
```

The algorithm starts a local search from every sample point that could

not join any of the existing clusters. If a search ends up in a point that can be clustered, than this point and the starting one are added to that cluster, or the algorithm creates a new cluster from these points otherwise.

ALGORITHM 5.2. UNIRANDI

```
1: input objective-function: F
2: input search-space: [a, b]
3: input criteria: termination
4: input x_0 := x, step-length := h
5: input function: line-search
6: output sample: opt-point := null, float: opt-value := ∞

7: while evaluate(condition: termination) = false do
8:     direction := vector(length: unit, from: uniform-distribution)
9:     x^+ := bound(value: x_0 + step-length · direction, to: [a, b])
10:    x^- := bound(value: x_0 − step-length · direction, to: [a, b])
11:    if F(x^+) < F(x_0) then
12:        x_0 := x^+
13:        opt-point, steplength := Line-search(function: F, x: x_0,
               direction: direction, step-length: step-length, over: [a, b])
14:    else if F(x^-) < F(x_0) then
15:        x_0 := x^-
16:        opt-point, steplength := Line-search(function: F, x: x_0,
               direction: −direction, step-length: step-length, over: [a, b])
17:    else
18:        step-length := step-length/2
19:    end if
20: end while
21: return opt-point, opt-value := F(opt-point)
```

The previous implementations of GLOBAL used the algorithm UNIRANDI, Algorithm 5.2, for local search by default. UNIRANDI is an iterative, random walk method that executes line searches, Algorithm 5.3, in random directions using an adaptive step length. Each iteration starts with a direction vector generation. Then, the algorithm moves in that direction by a previously defined step length. If the objective function has a lower value in the resulting new point, a line search is executed in that direction.

If the first step brings no improvement in the objective function, then the algorithm probes the opposite direction in the same way. If both attempts fails to discover a better point, then we decrease the step-length and try again in the next iteration. If UNIRANDI manages to run a line search in a direction, then the next iteration will continue from the resulted local optimum candidate ans the last used the step length.

ALGORITHM 5.3. Line-search

1: **input** objective-function: F
2: **input** vector: x
3: **input** vector: *direction*
4: **input** search-space: *over*
5: **input-output** vector: *step-length*
6: **output** sample: *opt-point* := *null*

7: $x_1 := \infty$, $x_2 := \infty$, $x_0 := x$
8: **repeat**
9: $x_2 := x_1$, $x_1 := x_0$
10: $x_0 :=$ bound(value: $x_0 + \textit{step-length} \cdot \textit{direction}$, to: *over*)
11: *step-length* := *step-length* $\cdot 2$
12: **until** $F(x_0) < F(x_1)$
13: *step-length* := *step-length*/2
14: **return** *opt-point* := x_1, *step-length*

GLOBAL repeats the iterations until any of the stopping criteria is fulfilled, the algorithm runs out of the allowed maximum of CPU time, iterations, function evaluations, local searches, or number of found local minimum points.

5.2 Algorithmic improvements

We aimed to modernize GLOBAL not just on the implementation level but algorithmically too by improving it to avoid even more unnecessary local searches with better organized clustering and local search.

The original single-linkage clustering strategy, see Algorithm 5.4, is executed at two points in each iteration of GLOBAL, after the sample generations

and the local searches. Let us focus on the latter and examine how the algorithm continues after a local search ends.

ALGORITHM 5.4. Single-linkage-clustering

```
 1: input objective-function: F
 2: input-output sample-set: unclustered
 3: input-output cluster-set: clusters

 4: N := count(values: samples-of(clusters)) + size-of(unclustered)
 5: critical-distance := calculate-critical-distance(sample-count: N)
 6: clustered-samples := create-set(type: sample, values: empty)
 7: for all cluster: cluster in clusters do
 8:       for all sample: sample in cluster do
 9:             add(value: sample, to: clustered-samples)
10:       end for
11: end for
12: for all sample: cs in clustered-samples do
13:       for all sample: us in unclustered do
14:             if distance(from: us, to: cs, type: ∞-norm) ≤ critical-distance
                  and F(cs) < F(us) then
15:                   move(value: us, from: unclustered, to: cluster-of(cs))
16:             end if
17:       end for
18: end for
19: return clusters, unclustered
```

Consider the following situation that we illustrate on Figure 5.1. There are three new samples, A, B, and C, that remained unclustered after the main clustering phase of an iteration, therefore we continue with local searches. First, we start a local search from A, and we find a cluster, the large one in the center, which has an element that is within the critical distance of A', the result point of the local search, therefore we add A and A' to this cluster. We run a clustering step according to Algorithm 5.4 in order to look for potential cluster members within the critical distance of A and A'. As a result B also joins the center cluster. We follow the same process with C, and add the two other sample points, C and C', to the same cluster again. What we missed is that we could have avoided the second local search if we

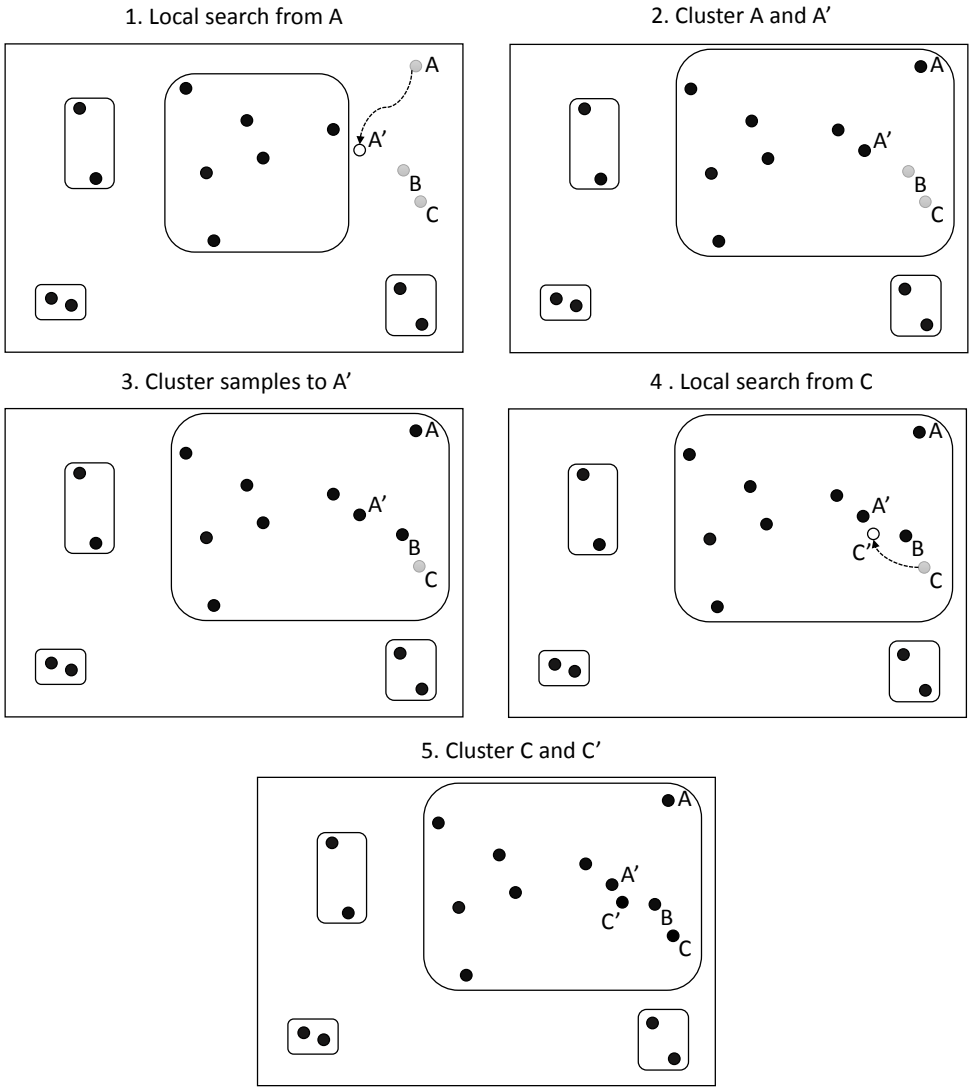


Figure 5.1: An example scenario when the original single-linkage clustering strategy of GLOBAL fails to recognize cluster membership in time and makes an unnecessary local search as a result. Black points denote clustered samples, grey points are unclustered samples, and white points represent the result of local searches.

had realized that C is in the critical distance of B indicating that it belongs to the same cluster which A just joined, thus the second local search was

unnecessary.

The solution is to leverage the complete clustering information all the time, as soon as it is available. The upgraded clustering algorithm for the new implementation of GLOBAL achieves this through a revised clustering strategy that you can see in Algorithm 5.5. Samples are divided into three sets, the clustered, the newly clustered, and the unclustered sets. Clustering cycles start having elements only in the clustered and unclustered sets. If a sample fulfills the joining condition for a cluster, then it moves to the newly clustered set instead of the clustered one. After all the samples of the unclustered set were tried to be added to the existing clusters for the first time, we retry to cluster the remaining unclustered set but checking the joining condition for only the elements of the newly clustered samples, if there were any. After such a follow-up attempt, newly clustered samples become clustered, and the samples added to a cluster in this iteration fill the newly clustered set. We continue these iterations until sets do not change anymore. This recursive-like clustering compares each pair of samples exactly once without requiring further objective function evaluations; moreover, a portion of these comparisons would probably happen anyway during the clustering steps after the later local searches according to the old clustering strategy.

We saw another opportunity for improvement in the sample reduction logic. GLOBAL was designed and implemented in FORTRAN when the available physical memory was far more limited, compared to what we can have today, and the internet and affordable virtual machines in clouds did not exist. The reduction step discards samples that have worse objective function values, thus they are presumably located in less promising areas of the search space, to provide control, beside the sample size, over how much memory our program will use. This limitation was useful back then but not necessary nowadays, though every later implementation carried it over. The problem is that the reduction step affects the frequency of cluster creation, not just the memory usage. It can throw away already clustered samples continuously leaking the already gathered cluster information.

ALGORITHM 5.5. Recursive-single-linkage-clustering

```
1: input objective-function: F
2: input-output sample-set: unclustered
3: input-output cluster-set: clusters

4: newly-clustered := create-set(type: sample, values: empty)
5: N := count(values: samples-of(clusters)) + size-of(unclustered)
6: critical-distance := calculate-critical-distance(sample-count: N)
7: clustered-samples := create-set(type: sample, values: empty)
8: for all cluster: cluster in clusters do
9:       for all sample: sample in cluster do
10:            add(value: sample, to: clustered-samples)
11:      end for
12: end for
13: for all sample: cs in clustered-samples do
14:      for all sample: us in unclustered do
15:            if distance(from: us, to: cs, type: ∞-norm) ≤ critical-distance
                 and F(cs) < F(us) then
16:                move(value: us, from: unclustered, to: newly-clustered)
17:                add(value: us, to: cluster-of(cs))
18:           end if
19:      end for
20: end for
21: while size-of(newly-clustered) > 0 do
22:      buffer := create-set(type: sample, values: empty)
23:      for all sample: us in unclustered do
24:            for all sample: cs in newly-clustered do
25:                  if distance(from: us, to: cs, type: ∞-norm) ≤ critical-distance
                       and F(cs) < F(us) then
26:                      move(value: us, from: unclustered, to: buffer)
27:                      add(value: us, to: cluster-of(value: cs)
28:                 end if
29:           end for
30:      end for
31:      newly-clustered := buffer
32: end while
33: return clusters, unclustered
```

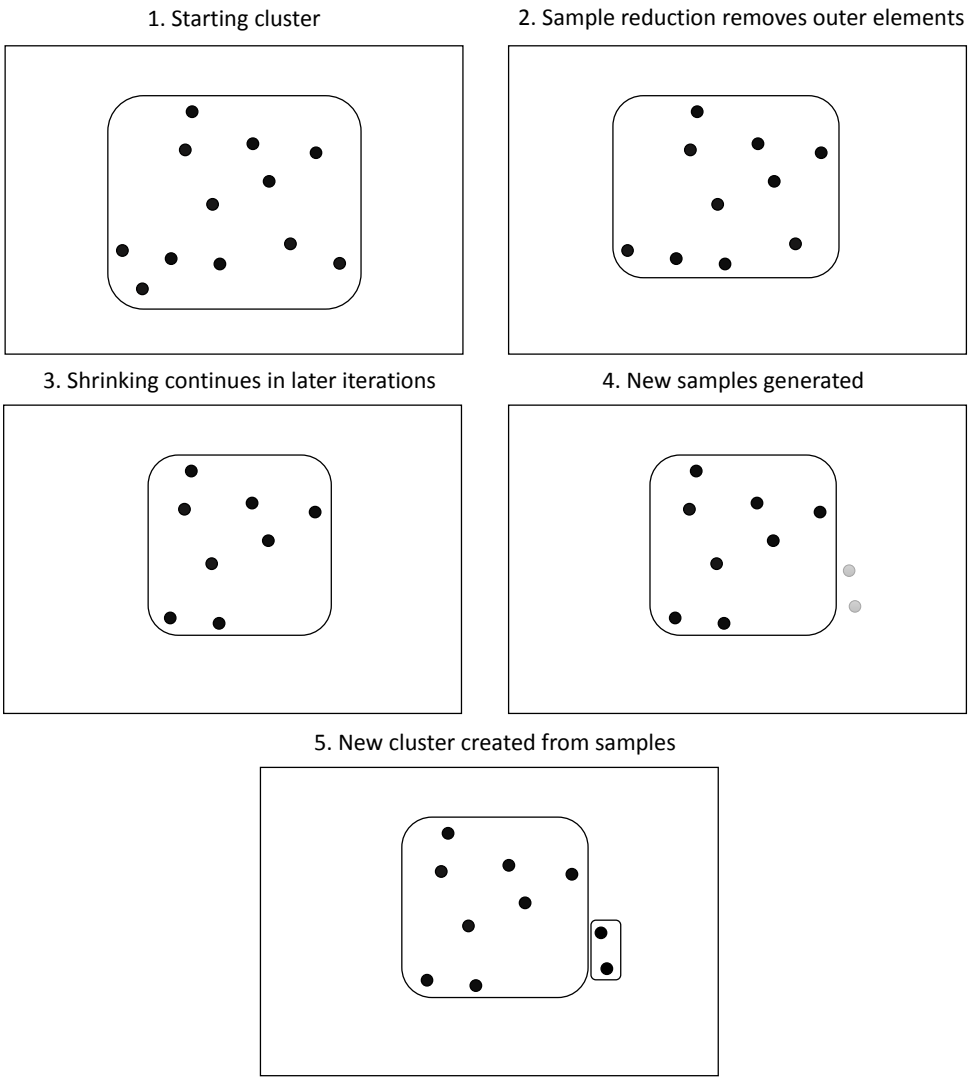


Figure 5.2: An example of cluster erosion and fragmentation. The removal of samples with greater objective function values in the reduction step transforms clusters to be more dense and concentrated around the local minima playing the role of cluster centers in this context. This process and the decreasing critical distance over time may create multiple clusters in place of former larger ones.

The portion of samples having worse objective function values are mainly

located far from the local optima that act like cluster centers from the algorithm's point of view. Clusters become more dense over the iterations because only the samples closer to the centers are carried over from one iteration to the next one. As the critical distance simultaneously decreases with the total number of sample points, new clusters might appear in the place where discarded samples were located previously. This cluster fragmentation or erosion, illustrated in Figure 5.2, has two different interpretations. The first one is that the algorithm discovers the finer granularity of the search space this way. The other is that the algorithm creates false clusters in the sense that they do not represent new, distinct regions of attractions. In our experience, the latter proved to be true in the great majority of the examined cases. More clusters mean more local searches as well forcing the algorithm to reach the stopping criterion of maximum allowed local optima earlier than it should be. Beside that, exploring the same region of the search space multiple times is redundant and thus inefficient.

In our upgraded version of GLOBAL, our strategy is in between keeping the old reduction step, using some reduction ratio λ less than 1, and completely leaving it out from the algorithm. The new reduction step still discards the worst portion of samples in each iteration from the merged set, but it holds on to the cluster membership information of these samples that were clustered in the previous iterations, therefore inferior samples will not participate in sorting operations and will not be clustered if they are newly generated samples in the current iteration. This modification is a tradeoff between runtime and memory usage. GLOBAL will use more memory in this way, storing more cluster information, and it will spend more time for clustering, but less local searches will be executed, that generally need much more computation. This will result in a shorter runtime all in all. You can see the above discussed modifications in the upgraded GLOBAL, named GLOBALJ, in Algorithm 5.6.

ALGORITHM 5.6. GLOBALJ

```
1: input objective-function: F
2: input search-space: [a, b]
3: input criteria: termination
4: input integer: N := 100, float: λ := 0.5
5: input modules: clusterizer, local-search
6: output sample: opt-point := null, float: opt-value := ∞

7: reduced := create-list(type: sample, values: empty)
8: unclustered := reduced, i := 1
9: clusters := create-set(type: cluster, values: empty)
10: while evaluate(condition: termination) = false do
11:     new := create-list(type: sample, values: generate-samples(
          from: create-distribution(type: uniform, values: [a, b]), count: N))
12:     reduced := sort(values: union-of({new, reduced}), by: F, order: ascending)
13:     reduced := select(from: reduced, index-range: [1, ⌊i · N · λ⌋])
14:     add(values: intersection-of({reduced, new}), to: unclustered)
15:     clusters, unclustered := clusterizer.cluster(objective-function: F,
          unclustered: unclustered, clusters: clusters)
16:     while size-of(unclustered) > 0 do
17:         x := first-of(unclustered)
18:         x* := local-search.optimize(function: F, start: x, over: [a, b])
19:         add(value: x*, to: reduced)
20:         clusters, unclustered := clusterizer.cluster(objective-function: F,
              unclustered: {x*, x}, clusters: clusters)
21:         if cluster-of(x*) = null then
22:             cluster := create-cluster(type: sample, values: {x*, x})
23:             add(value: cluster, to: clusters)
24:         end if
25:         clusters, unclustered := clusterizer.cluster(objective-function: F,
              unclustered: unclustered, clusters: clusters)
26:     end while
27:     i := i + 1
28: end while
29: sort(values: reduced)
30: return opt-point := first-of(reduced), opt-value := F(first-of(reduced))
```

5.3 Redesigned implementation

To avoid confusion, we are going to refer the previous MATLAB implementation of the original algorithm as *GLOBALM* [17], the new implementation as *GLOBALJ*, and we are going to use the plain term *GLOBAL* for the algorithm itself in the theoretic sense [15].

The previous implementation of GLOBAL was written in MATLAB [51], a great software for many purposes. It is like a Swiss army knife for computer science having a tool box for almost every area of expertise. It provides a wide range of well-known algorithms as built-in functions optimized for vector operations to handle large data efficiently. MATLAB is great for fast creation of proof of concepts and for conducting research in general. MATHWORKS commits tremendous effort to maintain and improve MATLAB from version to version that costs a lot. Performance is also an issue. MATLAB uses its own interpreted coding language that is significantly slower than compiled languages. Integration with other software is problematic too. Commercial software rarely have a MATLAB compatible interface, and though an array of tools provided to transform our code to another software platform, the main problem remains untouched. We have to learn to use a lot of auxiliary tools that might further degrade the performance of our solution.

The last problem is the implementation itself. GLOBALM is highly expert-friendly. Only a few parameters can be modified out of the box. The monolithic design makes any alteration under the hood exceptionally difficult. Although we can use multiple built-in functions of MATLAB for differentiable objective functions, like FMINCON, a sequential, quadratic programming method relying on the BFGS formula, or SQLNP for solving linearly constrained problems with LP and SQP approaches. For details of these algorithms and their implementation, visit the homepage of MATLAB [51]. The creators added UNIRANDI to the repertoire for non-differentiable cases. We might wish to use our own special algorithms sometimes, unfortunately integrating a custom local search algorithm is also not straightforward at all.

Planning the new implementation, three programming platforms, Python,

C++, and JAVA offered viable options that matched our goals, a software solution that is widely available and easy to customize. They are amongst the most popular choices for software engineering [77] and a lot of great libraries in computer science [4, 37, 38, 59, 71, 84] and financial economics [39, 63] have been engineered using them. Considering our first goal, we decided to create the new implementation in JAVA, hence the name of GLOBALJ, as most of the new optimization libraries in the scientific community are nowadays published in JAVA, and industrial software often provides JAVA APIs for integration.

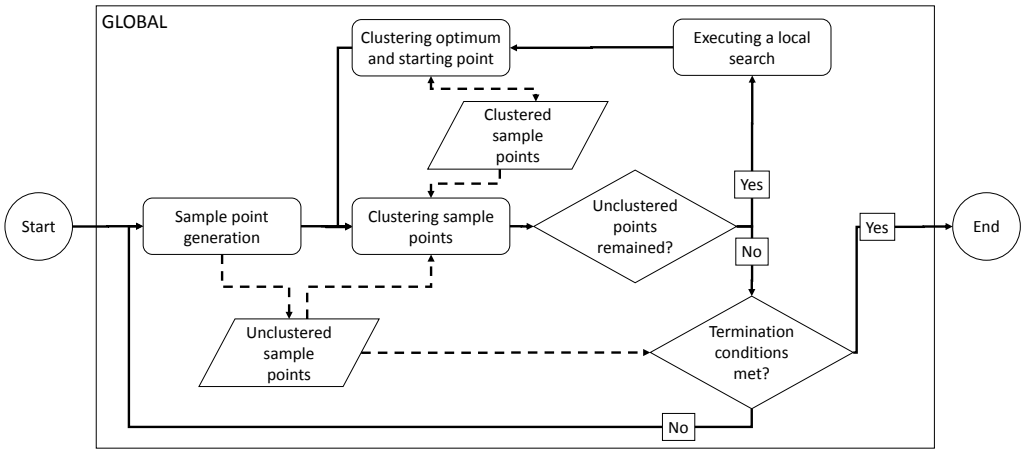


Figure 5.3: High level illustration of how GLOBALM operates showing the optimization cycle. Continuous lines denote control flow, dashed lines denote data flow.

The key of easy customization is a good decomposition, the identification and separation of the main processes. GLOBAL is fundamentally built up from two operations, local search and starting point generation. The uniform random sampling and clustering together serve as the starting point generation, local search is straightforwardly covered by the chosen local search method. The high level abstraction of old GLOBAL is illustrated on Figure 5.3. Based on this observation, we decoupled the previously monolithic design into three modules, the clustering module, the local search module, and

the GLOBAL module that provides a frame for the first two. Figure 5.4 presents the interaction of modules in GLOBALJ. We decided to leave the sample generation in the main module, reduction phase included because, it is a very small functionality, and there is no smarter option than choosing random samples from a uniform distribution over the feasible set anyways. An iteration can be considered as a pipeline, a sequence of clustering and local search operations. The task of this module is to create samples for the pipeline and direct the data through it.

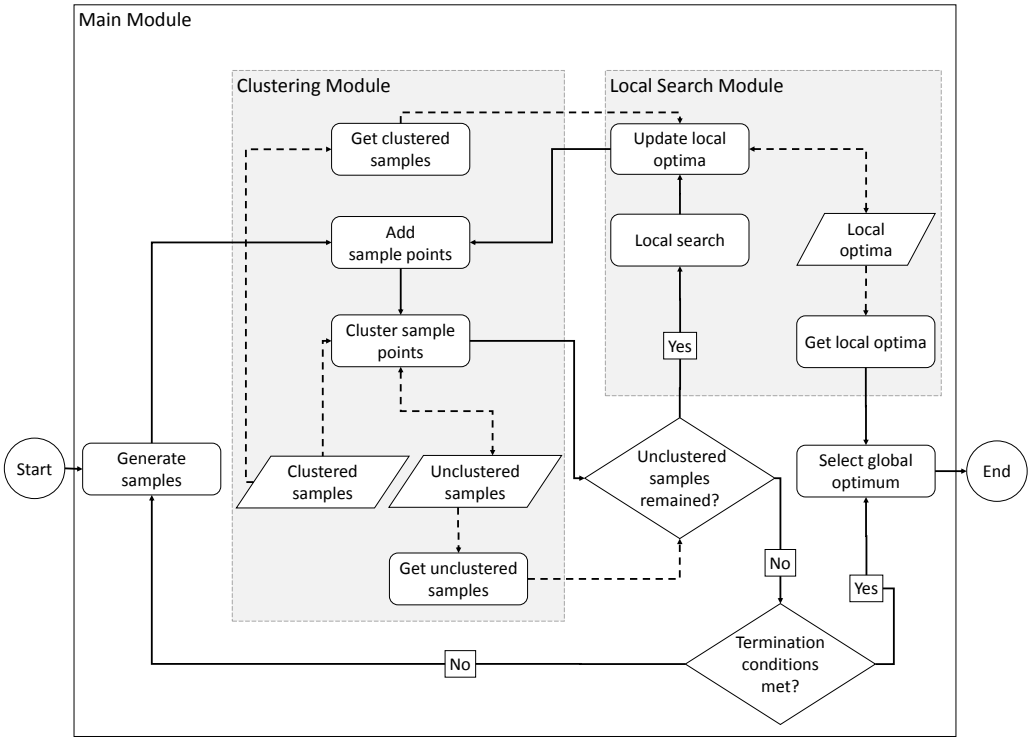


Figure 5.4: High level illustration of the modularized implementation, GLOBALJ. Continuous lines denote control flow, dashed lines denote data flow as in Figure 5.3.

The default clustering module of GLOBALJ implements the improved single-linkage clustering that was previously presented. Consequently, the module interface is richer supporting the addition and removal of samples

and clusters as well beside clustering. This module holds the clusters, their elements, and the unclustered samples. In general, clustering algorithms run until every sample joins a cluster. As GLOBAL does not use a priori knowledge about the regions of attraction, implementers must pay attention to that GLOBAL needs an incomplete clustering that identifies and leaves outliers alone. Hierarchical algorithms [55, 68] are well suited for this task. It is also wise to choose clustering strategies that do not tend to prefer and create certain shapes of clusters, for example spherical ones, as such behaviors can easily mislead the search, the regions of attractions may have arbitrary forms.

The local search module is straightforward. It is responsible for several bookkeeping operations, counting the number of executed objective function evaluations, and storing the new local optima found during the optimization, and it should be provided the upper and lower bounds of the feasible set of sample points in order to make it able to restrict the searches to this region. GLOBALJ provides the implementation of the local search algorithm UNIRANDI by default.

5.4 Results

We compared GLOBALJ and GLOBALM on a test suite consisting of 63 function frequently used for performance studies of optimization methods to examine the effects of the algorithmic improvements and the platform switch in terms of executed objective function evaluations and runtime. Both implementations were provided the same memory limit and computing capacity using an average desktop PC and the same parameter setting listed in Table 5.1. We turned off all stopping criteria except the maximum allowed number of function evaluations and the relative convergence threshold.

New samples generated in a single iteration:	400
Sample reduction factor:	5%
Maximum number of allowed function evaluations:	10^8
Relative convergence threshold:	10^{-8}
The α parameter of the critical distance:	optimal
Applied local search algorithm:	UNIRANDI

Table 5.1: The applied parameter values for the comparison of GLOBALJ with GLOBALM. For both algorithms, we ran a preliminary parameter sweep for the clustering parameter α for every function, and we used the attained value for which the algorithm performed the best.

We ran the algorithms 100 times independently for the entire test suite, see the book [9] for the details. First, we concentrated on the executed function evaluations in our analysis. We discarded the functions for which only one of the algorithms were able to locate an optimum narrowing down the study to the 25 functions for which both GLOBALJ and GLOBALM found the global optimum in all the runs in order to ensure the comparison of results of equal quality. We measured the change of the average number of executed function evaluations using the following formula:

$$change = \frac{average\ of\ GLOBALJ - average\ of\ GLOBALM}{average\ of\ GLOBALM}.$$

The results are presented in Figure 5.5. As it was predicted, the algorithmic improvements of GLOBALJ came up to our expectation in the great majority of the cases scoring a 27% overall improvement, and only performing relatively a bit worse in a few cases.

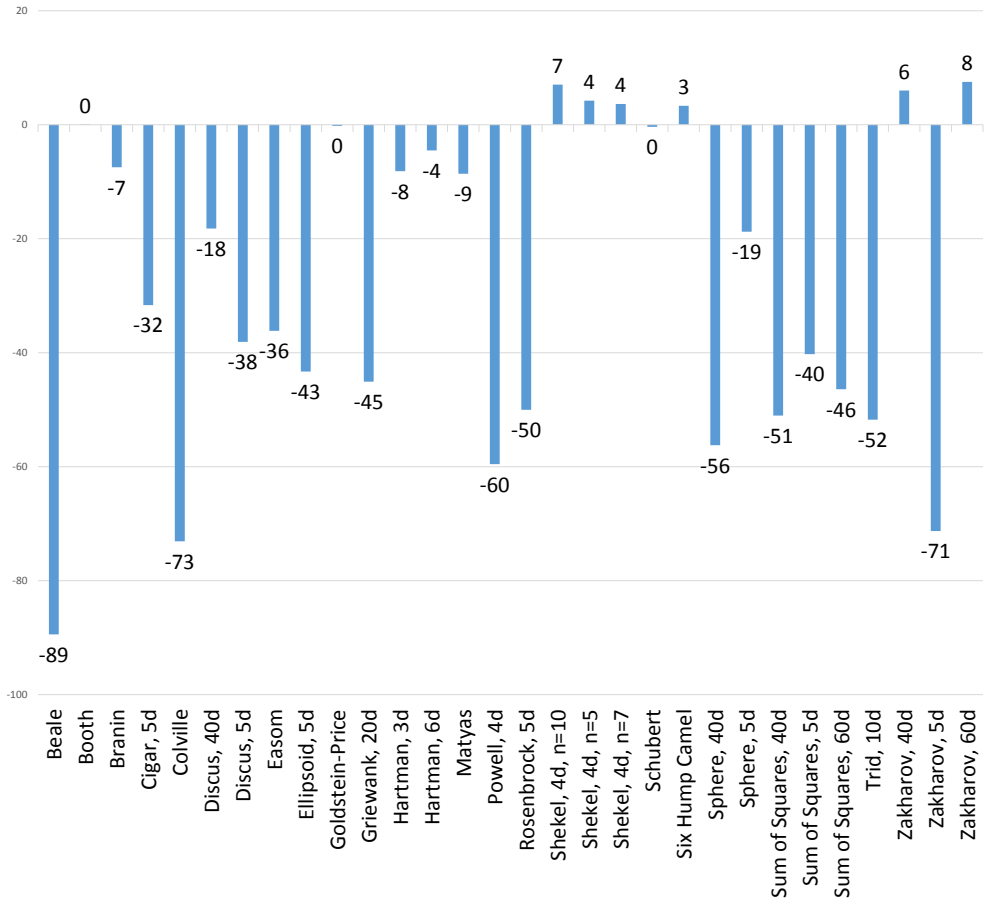


Figure 5.5: The relative change measured in percent in the average of executed function evaluations by GLOBALJ compared to GLOBALM.

Mainly two sets of functions, the *Shekel* and the *Zakharov* functions required a little more effort from GLOBALJ compared to GLOBALM. The *Shekel* family of functions has a lot of local optima that definitely require a higher number of local searches, and it is an exception to our general observation about cluster fragmentation. A cluster created in the early iterations of the optimization may have more than one local optima in reality in case of these functions. As an opposite, the Zakharov functions have only one global optimizer point, but its region of attraction resembles much more to a hypersphere. Pairing this fact with high dimensionality, running more local searches leads to the optimum earlier, and thus it is a better strategy than a

decreased number of local searches.

Regarding the technical aspect of runtime, GLOBALJ was at least 10 times faster than GLOBALM in all cases due to the efficiency difference of compiled and interpreted languages.

5.5 Summary

This chapter focused on the core structure of GLOBAL, a stochastic, global optimization algorithm, and presented our work of making it anew. We highlighted the key points of possible improvements and introduced a solution in form of a new clustering strategy. We kept the basic approach of single-linkage clustering but modified it to incorporate every available information about the search space as soon as possible. We decided to hold onto all the clustering information during the whole run in order to prevent the redundant discovery of the search space. We implemented the modified GLOBAL algorithm using a modularized structure in JAVA to provide the option of customizing the applied local solver and clustering algorithm. We compared the new JAVA and the old MATLAB implementations, and we experienced a significant improvement in the number of necessary function evaluations to find the global optimum on a wide range of optimization test functions.

Publications

Bánhelyi, B., Csendes, T., Lévai, B., L., Pál, L., Zombori, D.: The GLOBAL Optimization Algorithm, Newly Updated with Java Implementation and Parallelization. Springer Briefs in Optimization, Springer, 2018.

Bánhelyi, B., Csendes, T., Lévai, B., L., Zombori, D., Pál, L.: Improved versions of the GLOBAL optimization algorithm and the globalJ modularized toolbox. AIP Conference Proceedings, 2070(1), 2019.

Bánhelyi, B., Lévai, B., L.: Automatic failure detection and monitoring of ventilation and cooling systems. Proceedings of the 9^{th} International Conference on Applied Informatics, Eger, 1:27–32, 2014.

Bánhelyi, B., Palatinus, E., Lévai, B., L.: Optimal circle covering problems and their applications. Central European Journal of Operations Research, 23:815–832, 2015.

Lévai, B., L., Bánhelyi B.: An optimization technique for verified location of trajectories with prescribed geometrical behaviour in the chaotic forced damped pendulum. Central European Journal of Operations Research, 41:757–767, 2013.

Lévai, B., L., Bánhelyi, B.: Automatic Design of Optimal LED Street Lights. Optimized Packings with Applications, Springer Optimization and Its Applications, vol. 105, Springer, 2015.

Summary

Global optimization is a vast field of expertise. Based on the attributes of the objective function and the search space, particular optimization methods better fit a problem than others, selecting the most appropriate algorithm or theoretical concept to solve a problem can be challenging. Optimization algorithms can be separated into classes depending on multiple attributes. We categorized them based on the application of random variables into three main classes, deterministic, stochastic, and hybrid algorithms.

In my dissertation, I discussed the design and development of solutions to global optimization problems to demonstrate the application of the three main approaches, combined with interval arithmetic when mathematical rigorousness was required, furthermore I also presented the improvement of an existing optimization algorithm.

Circle covering with fixed centers

We aimed to determine the optimal cover of arbitrary polygons by circles with fixed centers but variable radii using reliable, rigorous numerical tools. Optimality means that the sum of squares of radii is minimal. A solution to this type of circle covering problem can be used to optimize the required power of different towers in a network that broadcasts terrestrial signal for telecommunication.

First, we introduced an algorithm to verify whether a given configuration of circles covers a polygon, then we constructed an optimization algorithm that systematically searches the space of allowed circle configurations until it

finds a cover that is closest to the global optimum with arbitrary but given precision. Both algorithms are deterministic using the branch-and-bound optimization strategy. The computation is based on interval arithmetic, thus the provided results are free from numerical errors.

The cover verification algorithm recognizes complete covers when the margin of error is set to zero, but it will not terminate for partial covers with this precision setting. The optimization algorithm is also guaranteed to terminate if the search space meets some requirements that can be easily ensured, and the returned configuration will cover the target polygon having an objective function within the predefined precision compared to any global optima.

To demonstrate the operation and effectiveness of our solution, we calculated several optimal covers of the unit square and the polygon approximation of the land of Hungary using various number of circles and different centers.

Designing LED based street lights

We were searching for an answer to a practical question, how to design streetlights with LED technology that provide better lighting than incandescent streetlights or other LED streetlights already available in the market. As the intersection of the target surface and a light cone of a LED light is an ellipse and the intensities of different light sources simply add up, we can interpret this designing problem as covering a rectangle shaped area with ellipses while overlapping is allowed. This is a multi-objective, global optimization problem with high dimensionality.

Our approach was a stochastic solution, we constructed a genetic algorithm whose essence was a geometric crossover concept being able to combine partially good parts of different designs based on the light pattern. We used a configurable fitness function that allowed us to set the relative importance of the different requirements. The computationally most expensive part of the fitness function was a grid-based, custom light pattern calculation method that took advantage of any symmetries present in the pattern in order to reduce the number of grid vertices that must be handled.

The genetic algorithm provided designs whose light patterns had much better quality than the commercial competitors. After the completion of optimization algorithm, we experimentally moved the light pattern calculation to the GPU and managed to drastically reduce the optimization time even for extreme test cases.

Searching chaotic trajectories

We studied the forced damped pendulum, a mass point hung with a weightless solid rod whose other end point is fixed. The pendulum is affected by the gravitation, a dampening friction, and an external periodic force. We created a method to locate regions of initial states from which this dynamic system will execute a sequence of prescribed motions. These so-called chaotic trajectories of this system are the bi-infinite series of three types of motions.

We designed an objective function expressing the measure of difference between trajectories and the motion prescriptions based on the concept of Hausdorff distance, and we located initial states by optimizing this function with GLOBAL, a stochastic algorithm based on clustering. We used interval arithmetic to ensure reliable numeric results. Theoretically, our approach can find trajectories for arbitrary fixed length series of motions, but in practice, the precision of the numeric representation used in the implementation limits the manageable length of such series.

We were able to reveal regions of trajectories for all possible series of prescribed motions of length 3 and for a few other trajectories that we studied to demonstrate the capabilities of our solution.

Improvement of a stochastic global optimization method

We reimplemented and algorithmically improved the algorithm GLOBAL, a stochastic optimization method aiming to solve non-linear, constrained optimization problems. It is a versatile tool that clusters randomly generated

points to already found local optima and starts local searches only from the points that cannot be added to any of the existing clusters.

We kept the basic approach of single-linkage clustering but modified it to better leverage the available clustering information in order to avoid the initiation of most likely unnecessary local searches. We also decided to hold onto all the clustering information during the whole optimization, that was partially dropped in each iteration of the algorithm previously, thus we prevent the redundant discovery of the search space. We created a new modularized JAVA implementation of the improved GLOBAL that provides an easy way to customize not just the applied local search method but the clustering algorithm as well.

We compared the new JAVA and the old MATLAB implementations, and we experienced a significant improvement in the number of necessary function evaluations to find the global optimum on a wide range of optimization test functions.

Összefoglaló

A globális optimalizálás széles szakterület. A célfüggvények és keresési terek tulajdonságai alapján egyes optimalizáló metódusok jobban illeszkednek egy probléma megoldásához, mint mások, pusztán a megfelelő algoritmus vagy elméleti megközelítés kiválasztása egy probléma megoldásához is kihívást jelenthet. Az optimalizáló algoritmusok különböző osztályokba sorolhatók tulajdonságaik alapján. Mi a véletlen változók alkalmazása szerint kategorizáltuk őket három fő osztályba, a determinisztikus, sztochasztikus és hibrid algoritmusok osztályaiba.

Disszertációmban globális optimalizálási feladatok megoldásainak tervezését és fejlesztését tárgyaltam, hogy bemutassam a fentebb említett három fő megközelítés alkalmazását, alkalmanként intervallum aritmetikával kombinálva, amikor szükséges volt a matematikai értelemben vett szigor és pontosság. Mindezen felül egy létező optimalizáló algoritmus továbbfejlesztését is ismertettem.

Körfedés rögzített középpontokkal

Célunk tetszőleges poligonok körökkel történő optimális fedésének meghatározása volt úgy, hogy a körök középpontjai rögzítettek, csak a sugaraik változtathatóak. Ehhez numerikusan megbízható pontos módszereket használtunk. Az optimalitás ebben a kontextusban azt jelenti, hogy a körök sugarainak négyzetösszege minimális. Egy ilyen típusú körfedési probléma megoldása felhasználható például a telekommunikációs hálózatokban lévő földfelszíni sugárázású adótornyok szükséges teljesítményének optimalizálásához.

Először, bevezettünk egy algoritmust annak ellenőrzésére, hogy egy adott kör konfiguráció lefed-e egy adott poligont. Ezután készítettünk egy optimalizáló algoritmust, ami szisztematikusan átkutatja a megengedett kör konfigurációk keresési terét, amíg nem talál egy olyan fedést, amely egy előre rögzített, de tetszőleges pontosságon belül van bármely globális optimumhoz hasonlítva. Mindkét algoritmus determinisztikus és a branch-and-bound optimalizáló stratégiát használja. A számításokban intervallum aritmetikát használtunk, így a kiszámított eredmények numerikus hibáktól mentesek.

A fedés ellenőrző algoritmus felismeri a teljes fedéseket, ha az előre beállítható hibahatár 0, de nem áll meg részleges fedések esetén ezzel a beállítással. Ha a keresési tér teljesít pár könnyen biztosítható feltételt, akkor az optimalizáló algoritmus szintén garantáltan leáll és az általa visszaadott konfiguráció fedni fogja a cél poligont, a célfüggvény értéke pedig erre a konfigurációra az előre megadható pontosságon belül lesz bármely globális optimumhoz hasonlítva.

Megoldásunk működésének és hatékonyságának bemutatásához kiszámítottuk az egység négyzetre és Magyarország területének poligon közelítésére az optimális fedést különböző körülmények között mind a felhasználható körök számát, mind a körök középpontjait változtatva.

LED technológiájú utcai lámpák tervezése

Egy gyakorlatias kérdésre kerestük a választ, hogyan tervezzünk utcai lámpákat a LED technológiát felhasználva úgy, hogy a lámpák, jobban világítsanak, mint az izzós vagy más LED alapú társaik, amelyek már elérhetőek a piacon. Mivel a megvilágítandó felület és egy LED lámpa fénykúpjának metszete egy ellipszis, továbbá a különböző fényforrások fényerőssége egy adott ponton egyszerűen összeadódik, értelmezhetjük ezt a tervezési problémát egy speciális fedési feladatként is, amikor egy téglalap alakú felületet akarunk fedni ellipszisekkel úgy, hogy az ellipszisek egymást is átfedhetik. Ez egy többcélú, magas dimenziószámú, globális optimalizálási feladat.

A sztochasztikus probléma megközelítést választottuk. Készítettünk egy

genetikus algoritmust, amelynek esszenciája egy geometrikus koncepciójú keresztező operátor volt, amely képes a parciálisan jó megvilágítási képet adó részeket kombinálni a különböző lámpa tervekből. Egy konfigurálható fitness függvényt használtunk, ami állíthatóvá tette a különböző követelmények relatív fontosságát. A megoldásunk legtöbb számítást igénylő része egy egyedi rácsháló alapú megvilágítási kép számító eljárás volt, amely a hatékonyság érdekében figyelembe vette a feladatban lévő esetleges szimmetriákat, hogy csökkentse azon rácsháló pontok számát, amelyekre a lámpa fényerőssége meghatározandó.

A genetikus algoritmusunk által készített lámpa tervek sokkal jobb minőségű megvilágítási képpel rendelkeztek, mint a versenytársak. Az optimalizáló algoritmus befejezése után kísérleti jelleggel átmozgattuk a megvilágítási kép meghatározását a CPU-ról a GPU-ra drasztikusan lecsökkentve az optimalizáláshoz szükséges időt még a legextrémebb tesztesetekre is.

Kaotikus trajektóriák keresése

A kényszerrezgéses fékezett ingát tanulmányoztuk, amely egy súlytalan, szilárd rúdra felfüggesztett tömeg pontból áll, a rúd másik végpontja rögzített. Az ingára a gravitáció, a fékező súrlódás, és egy periodikus külső erő van hatással. Készítettünk egy eljárást e dinamikai rendszer kezdőállapot régióinak megtalálására, amelyből az inga egy megadott mozgás sorozatot fog végrehajtani. Ennek a rendszernek ezek az úgynevezett kaotikus trajektóriái olyan mindkét irányban végtelen sorozatok, amelyeknek elemei három különböző mozgásból kerülhetnek ki.

Terveztünk egy célfüggvényt, ami képes leírni az egyes trajektóriák eltérésének mértékét az előírt mozgásoktól egy Hausdorff távolságon alapuló távolság koncepció segítségével. A GLOBAL klaszterezésen alapuló sztochasztikus algoritmust felhasználva optimalizáltuk ezt a célfüggvényt, hogy kezdő állapotokat lokalizáljunk. Az eredmények numerikus megbízhatósága végett intervallum aritmetikát használtunk. Elméleti síkon a megoldásunk képes tetszőleges hosszúságú rögzített mozgás sorozathoz kezdő állapotokat

találni, de a gyakorlatban e sorozatok kezelhető hosszúságát limitálja a számábrázolás implementációjának a pontossága.

Képesek voltunk trajektória régiókat találni az összes három hosszú mozgás sorozat előíráshoz és néhány további sorozathoz is, amelyeket abból a célból vizsgáltunk, hogy bemutassuk a módszerünk képességeit.

Egy sztochasztikus globális optimalizáló eljárás továbbfejlesztése

Újraimplementáltuk és algoritmikusan továbbfejlesztettük a GLOBAL algoritmust, egy sztochasztikus optimalizáló eljárást, amelynek célja a nem lineáris, korlátos optimalizálási problémák megoldása. Ez egy sokoldalú eszköz, amely véletlenszerűen generált pontok már megtalált lokális optimumokhoz klaszterez, és csak olyan pontokból indít lokális kereséseket, amelyeket nem tudott egyik már létező klaszterhez sem hozzáadni.

Megtartottuk a korábban használt single-linkage klaszterezési megközelítést, de úgy módosítottuk, hogy jobban kihasználja a rendelkezésre álló klaszterezési információt annak érdekében, hogy elkerüljük a legnagyobb valószínűség szerint felesleges lokális kereséseket, továbbá megtartottuk az összes klaszterezési információt az optimalizálás teljes idejére, amelyet korábban részlegesen eldobott az algoritmus minden iterációban, így megelőzzük a keresési tér redundáns bejárását. Elkészítettük a továbbfejlesztett GLOBAL új JAVA nyelvű modularizált implementációját, amely megkönnyíti nem csak a lokális keresőeljárás, hanem a klaszterező eljárás személyre szabását is.

Összehasonlítottuk az új JAVA és a régi MATLAB megvalósítást, és szignifikáns javulást tapasztaltunk optimalizálási teszt függvények széles körében a globális optimum megtalálásához szükséges célfüggvény kiértékelések számában.

Bibliography

[1] Ábrahám, Gy., Wenzelné Gerőfy, K., Antal, Á., Kovács, G.: Műszaki optika. BME-MOGI, 2014.

[2] Alefeld, G., Herzberger, J.: Introduction to Interval Computations. Academic Press New York, 1983.

[3] Alefeld G., Mayer G.: Interval Analysis: Theory and Applications. Journal of Computational and Applied Mathematics, 121:421–464, 2000.

[4] Apache Commons Math: accessed 11 Aug 2017. http://commons.apache.org/proper/commons-math

[5] Balogh, J., Csendes, T., Stateva, R., P.: Application of a stochastic method to the solution of the phase stability problem: cubic equations of state. Fluid Phase Equilibria 212:257–267, 2003.

[6] Banga, J. R., Moles, C., G., Alonso, A., A.: Global Optimization of Bioprocesses using Stochastic and Hybrid Methods. C.A. Floudas and P.M. Pardalos (eds.): Frontiers in Global Optimization, 45–70, Springer, Berlin, 2003.

[7] Bánhelyi, B., Csendes, T., Garay, B., M.: Optimization and the Miranda approach in detecting horseshoe-type chaos by computer. International Journal of Bifurcation and Chaos, 17:735–747, 2007.

[8] Bánhelyi, B., Csendes, T., Garay, B. M., Hatvani L.: A computer-assisted proof of Σ_3-chaos in the forced damped pendulum equation. SIAM Journal on Applied Dynamical Systems, 7(3):843–867, 2008.

[9] Bánhelyi, B., Csendes, T., Lévai, B., L., Pál, L., Zombori, D.: The GLOBAL Optimization Algorithm, Newly Updated with Java Implementation and Parallelization. Springer Briefs in Optimization, Springer, 2018.

[10] Bánhelyi, B., Csendes, T., Lévai, B., L., Zombori, D., Pál, L.: Improved versions of the GLOBAL optimization algorithm and the globalJ modularized toolbox. AIP Conference Proceedings, 2070(1), 2019.

[11] Bánhelyi, B., Lévai, B., L.: Automatic failure detection and monitoring of ventilation and cooling systems. Proceedings of the 9^{th} International Conference on Applied Informatics, Eger, 1:27–32, 2014.

[12] Bánhelyi, B., Palatinus, E., Lévai, B., L.: Optimal circle covering problems and their applications. Central European Journal of Operations Research, 23:815–832, 2015.

[13] Boender, C., G., E., Rinnooy Kan, A., H., G., Timmer, G., Stougie, L.: A stochastic method for global optimization. Mathematical Programming, 22:125–140, 1982.

[14] Casado, L., G., García, G.-Tóth, B., Hendrix, E., M., T.: On determining the cover of a simplex by spheres centered at its vertices. Journal of Global Optimization, 50:645–655, 2011.

[15] Csendes, T.: Nonlinear parameter estimation by global optimization – efficiency and reliability. Acta Cybernetica, 8:361–370, 1988.

[16] Csendes, T., Garay, B., M., Bánhelyi, B.: A verified optimization technique to locate chaotic regions of a Hénon system. Journal of Global Optimization, 35:145–160, 2006.

[17] Csendes, T., Pál, L., Sendin, J., O., H., Banga, J., R.: The GLOBAL Optimization Method Revisited. Optimization Letters, 2:445–454, 2008.

[18] Csete, M., Szekeres, G., Bánhelyi, B., Szenes, A., Csendes, T., Szabo, G.: Optimization of Plasmonic structure integrated single-photon detector designs to enhance absorptance. Advanced Photonics 2015, JM3A.30, 2015.

[19] C-XSC Language home page: accessed 5 Jan 2014.
http://www.xsc.de

[20] Dantzig, G., Orden, A., Wolfe, P.: The generalized simplex method for minimizing a linear form under linear inequality restraints. Pacific Journal of Mathematics, 5(2):183–195, 1955.

[21] Das, G., K., Das, S., Nandy, S., C., Shina, B., S.: Efficient algorithm for placing a given number of base station to cover a convex region. Journal of Parallel Distributing Computing, 66:1353–1358, 2006.

[22] Dorigo, M., Stützle T.: Ant Colony Optimization. A Bradford Book, 2004.

[23] Eiben, A., E., Smith, J., E.: Introduction to Evolutionary Computing. Springer, Berlin, 2015.

[24] EULUMDAT specification: accessed 12 Nov 2014.
http://www.helios32.com/Eulumdat.htm

[25] Falchia, F., Cinzanoa, P., Elvidgeb, C., D., Keithc, D., M., Haimd A.: Limiting the impact of light pollution on human health, environment and stellar visibility. Journal of Environmental Management 92:2714-2722, 2011.

[26] Floudas, C. A.: Deterministic Global Optimization, Theory, Methods and Applications. Nonconvex Optimization and Its Applications, vol. 37, Springer, 2000.

[27] Friedman. E.: Circles covering squares web page, accessed 12 Jan 2017.
http://www2.stetson.edu/∼efriedma/circovsqu

[28] Gallaway, T., Olsen, R., N., Mitchell, D., M.: The economics of global light pollution. Ecological Economics, 69:658-665, 2010.

[29] Glover, F., Laguna, M.: Tabu Search. Handbook of Combinatorial Optimization, 3:621–757, Springer, Boston, 1999.

[30] Goldberg, D., E: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Boston, 1989.

[31] Hendrix, E., M., T., G.-Tóth, B.: Introduction to Nonlinear and Global Optimization. Springer Optimization and its Application, Berlin, 2010.

[32] Heppes., A.: Covering a planar domain with sets of small diameter. Periodica Mathematica Hungarica, 53:157–168, 2006.

[33] Heppes, A., Melissen, H.: Covering a rectangle with equal circles. Periodica Mathematica Hungarica, 34:65–81, 1997.

[34] Horst, R., Pardalos, P., M., (Eds.): Handbook of Global Optimization. Kluwer, Dordrecht, 1995.

[35] Hubbard, J., H.: The forced damped pendulum: chaos, complication and control. The American Mathematical Monthly, 8:741–758. 1999.

[36] Illuminating Engineering Society standards: accessed 6 Jun 2018. https://www.ies.org/standards

[37] JScience: accessed 11 Aug 2017. http://jscience.org

[38] JGSL: accessed 11 Aug 2017. http://jgsl.sourceforge.net

[39] JQuantLib: accessed 11 Aug 2017. http://www.jquantlib.org

[40] Kearfott, R., B.: Rigorous global search: continuous problems. Kluwer, Dordrecht, 1996.

[41] Khan, M.; Anisiu, M.-C., Domoszali, L., Iványi, A., Kasa, Z., Pirzada, S., Szécsi, L., Szidarovszky, F., Szirmay-Kalos, L., Vizvári, B.: Algorithms of Informatics, Volume III. AnTonCom, Budapest (electronic), Mondat Kft. Budapest, 2013.

[42] Knüppel, O.: PROFIL/BIAS - A fast interval library. Computing, 53:277–287, 1994.

[43] Kubach, T., Bortfeldt, A., Gehring, H.: Parallel greedy algorithms for packing unequal circles into a strip or a rectangle. Central European Journal of Operations Research, 17:461–477, 2009.

[44] Lawrence, D.: Handbook of genetic algorithms. Van Nostrand Reinhold, 1991.

[45] Lévai, B., L., Bánhelyi B.: An optimization technique for verified location of trajectories with prescribed geometrical behaviour in the chaotic forced damped pendulum. Central European Journal of Operations Research, 41:757–767, 2013.

[46] Lévai, B., L., Bánhelyi, B.: Automatic Design of Optimal LED Street Lights. Optimized Packings with Applications, Springer Optimization and Its Applications, vol. 105, Springer, 2015.

[47] Locatelli, M., Schoen, F.: Global Optimization: Theory, Algorithms, and Applications. MOS-SIAM Series on Optimization, 2013.

[48] Longcore, T., Rich, C.: Ecological light pollution. Frontiers in Ecology and the Environment, 2:191-198, 2004.

[49] Manno, I.: Introduction to the Monte-Carlo method. Akadémiai Kiadó, Budapest, 1999.

[50] Mathar, R.: A Hybrid Global Optimization Algorithm for Multidimensional Scaling. Classification and Knowledge Organization: Proceedings of the 20th Annual Conference of the Gesellschaft für Klassifikation e. V., 63–71, 1997.

[51] The MathWorks Inc.: accessed 11 Aug 2017.
https://www.mathworks.com

[52] Mischaikow, K., Mrozek, M.: Chaos in the Lorenz equations: a computer-assisted proof. Bulletin of the American Mathematical Society, 2:66–72, 1995.

[53] Moles, C., G., Banga, J.R., Keller, K.: Solving nonconvex climate control problems: pitfalls and algorithm performances. Applied Soft Computing, 5:35–44, 2004.

[54] Moles, C., G., Gutierrez, G., Alonso, A., A, Banga, J., R.: Integrated process design and control via global optimization – A wastewater treatment plant case study. Chemical Engineering Research & Design, 81:507–517, 2003.

[55] Murtagh, F., Contreras, P.: Algorithms for hierarchical clustering: An overview. Data Mining and Knowledge Discovery, 2:86–97, Wiley, 2012.

[56] Nedialkov, N., S.: VNODE – A validated solver for initial value problems for ordinary differential equations, accessed 21 Jun 2015. http://www.cas.mcmaster.ca/~nedialk/Software/VNODE/VNODE.shtml

[57] Neumaier, A.: Complete Search in Continuous Global Optimization and Constraint Satisfaction. Acta Numerica, 13:271–369, 2004.

[58] Neumaier, A., Rage, T.: Rigorous chaos verification in discrete dynamical systems. Physica D: Nonlinear Phenomea, 67:327–346, 1993.

[59] NumPy: accessed 11 Aug 2017. http://www.numpy.org

[60] Nurmela, K., J.: Conjecturally optimal coverings of an equilateral triangle with up to 36 equal circles. Experimental Mathematics, 9:241–250, 2000.

[61] Palatinus, E.: Optimal and reliable covering of planar objects with circles. Proceedings of the 2010 mini-conference on applied theoretical computer science, 2010.

[62] Pintér, J., D.: Global Optimization in Action. Kluwer, Dordrecht, 1996.

[63] PyQL: accessed 11 Aug 2017.
https://github.com/enthought/pyql

[64] Radnai T., Vőneki R.: Method for constructing a lighting device with discrete light sources and thus obtained lighting device. World Intellectual Property Organization, International Publication Number, WO 2011/154756 A2, 2011.

[65] Rage, T., Neumaier, A., Schlier, C.: Rigorous verification of chaos in a molecular model. Physical review. E, Statistical physics, plasmas, fluids, and related interdisciplinary topics, 50:2682–2688, 1994.

[66] Renders, J., M., Flasse, S., P.: Hybrid methods using genetic algorithms for global optimization. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 26(2):243–258, 1996.

[67] Riegel, K., W.: Light Pollution: Outdoor lighting is a growing threat to astronomy. Science, 179:1285-1291, 1973.

[68] Rokach, L., Maimon, O.: Clustering Methods. Data Mining and Knowledge Discovery Handbook, 321–352, Springer, 2005.

[69] Roos C., Terlaky, T., Vial, J.-Ph.: Interior Point Methods for Linear Optimization. Springer, 2005.

[70] Salahi, M.: Convex optimization approach to a single quadratically constrained quadratic minimization problem, Central European Journal of Operations Research, 18:181–187, 2010.

[71] SciPy: accessed 11 Aug 2017.
https://www.scipy.org

[72] Sendín, J., O., H., Banga, J., R., Csendes, T.: Extensions of a Multistart Clustering Algorithm for Constrained Global Optimization Problems. Industrial & Engineering Chemistry Research, 48:3014–3023, 2009.

[73] Sergeyev, Y., D., Strongin, R., G., Lera, D.: Introduction to Global Optimization Exploiting Space-Filling Curves. SpringerBriefs in Optimization, Springer, New York, 2013.

[74] Shilov, N., V.: Verification of backtracking and branch and bound design templates. Automatic Control and Computer Sciences, 46(7):402–409, 2012.

[75] Shimrat, M.: Algorithm 112, Position of Point Relative to Polygon. Communications of the ACM, 5(8):434, 1962.

[76] Sotiropoulos, D., G., Stavropoulos, E., C., Vrahatis, M., N.: A new hybrid genetic algorithm for global optimization. Nonlinear Analysis: Theory, Methods, & Applications, 30(7):4529–4538, 1997.

[77] TIOBE Index: accessed 11 Aug 2017.
https://www.tiobe.com/tiobe-index

[78] Torma, B., G.-Tóth, B.: An efficient descent direction method with cutting planes, Central European Journal of Operations Research, 18:105–130, 2010.

[79] Tucker, W.: The Lorenz attractor exists. Comptes rendus de l'Académie des sciences paris - Série I - Mathématique, 328:1197–1202, 1999.

[80] Tykierko, M.: Using invariants to determine change detection in dynamical system with chaos. Central European Journal of Operations Research, 15:223–233, 2007.

[81] Yang, D., Li, G., Cheng, G.: A Hybrid Global Optimization Algorithm for Multidimensional Scaling. Chaos, Solitons & Fractals, 34:1366–1375, 2007.

[82] Ye, T., Huang, W.: Quasi-physical global optimization method for solving the equal circle packing problem. Science China Information Sciences, 54:1333–1339, 2011.

[83] van Laarhoven, P., J., Aarts, E., H.: Simulated Annealing: Theory and Applications. Mathematics and Its Applications, vol. 37, Springer, 1987.

[84] WEKA: accessed 11 Aug 2017.
http://www.cs.waikato.ac.nz/ml/weka/index.html

[85] Wiggins, S.: Introduction to Applied Nonlinear Dynamical Systems and Chaos. Springer, Berlin, 2003.

[86] Wilczak, D., Zgliczynski, P.: Computer assisted proof of the existence of homoclinic tangency for the Henon map and for the forced-damped pendulum. SIAM Journal on Applied Dynamical Systems, 8:1632–1663 (2009).

[87] Wolfram MathWorld, Interpolation: accessed 12 June 2018.
http://mathworld.wolfram.com/topics/Interpolation.html

[88] Zhang, A., Sun, G., Wang, Z., Yao, Y.: A Hybrid Genetic Algorithm and Gravitational Search Algorithm for Global Optimization. Neural Network World, 25(1):53–73, 2015.

[89] Zhigljavsky, A, A.: Theory of Global Random Search. Mathematics and its Applications, vol. 65, Springer, 1991.