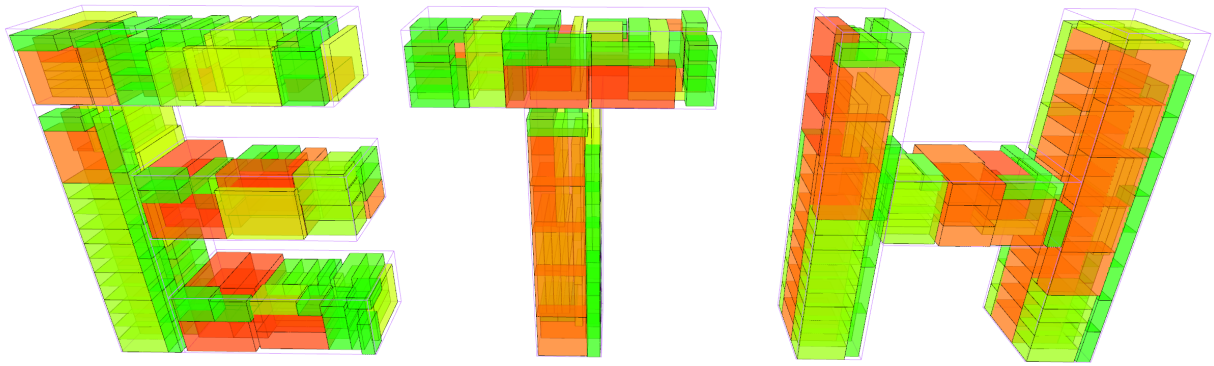
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Advanced algorithms for high performance box packing

A Master Thesis by Damian Heer

HS2023/FS2024

Supervisors: Prof. Dr. Stelian Coros, Dr. Moritz Geilinger, Simon Huber



Meinen Eltern, die immer an mich glaubten.

Meinen Eltern, die mich trotz allen Widrigkeiten immer Unterstützen.

Meinen Eltern, die mich lehrten, dass Weisheit und Verstand ein sicheres Fundament sind, auf dem du dein Haus errichten kannst, und Wissen seine Räume mit wertvollen und schönen Dingen füllt.

Abstract

After some introductory musings a corpus of related work, in both analytical and data-driven bin and box packing methods is reviewed. These papers influenced this thesis considerably or are considered interesting and valuable addition. Next, a class of algorithmic problems are introduced. Specifically focusing on the NP complexity class, the relationship between bin packing and the NP class is explored. An explanation on how box packing relates and is an adaptation of 2D bin packing is provided, along with a brief discussion why dynamic programming cannot effectively solve the bin or box packing problem.

Following this, a first naïve approach is outlined and the reasons why clever approaches, such as spectral convolution, fail are discussed. The concept of tri-level convolution and the cost function heuristic are introduced. A reformulated packing problem is stated, and some analytical costfunctions are presented.

Next, concrete implementations in C++, specifically focusing on the MedianQuadTree (MQT) library, which solves the tri-level convolution in $O(\sqrt{n})$, are delved into. A detailed explanation of the algorithm is provided, and benchmarks demonstrating an overall speedup of approximately 10x are presented.

Following this, the TurboPacker (TP) library, which uses the MQT library as a backend and solves the box packing problem using a cost function heuristic, is presented. The functionality of the library, including its frontend and its three operational modes, is showcased. Additionally, the inner workings of the library and how it employs a fire-and-forget approach to maximize user-friendliness are discussed.

Finally, results from deploying handcrafted analytical costfunctions are presented and their limitations discussed. Furthermore, a first sketch of an unsupervised learning approach using annealing and random gradient ascent are introduced. The achieved results are showcased and reviewed.

The thesis concludes with remarks summarizing the findings and identifying four critical areas where further research is needed: The lack of a distributed and or a CUDA implementation, the lack of critical functionality like environmental information, the lack of understanding of analytic costfunctions and the lack of work done into unsupervised learning.

Contents

1 Preface

Box packing, the process of packing items into a larger bin, is an almost omnipresent task in our daily lives. Whether it's packing presents into a box for Christmas or putting belongings into boxes for moving, the list of applications is nearly endless. It is surprising, then, that a task so intuitive and simple for humans poses such a significant challenge for computers.

In this thesis, I will shed some light on why the box packing problem is so difficult and introduce new ideas and approaches to solve it. I will begin by building up the theoretical foundation from the ground up to explain why box packing is such an algorithmic challenge. Following this, I will present my own considerations and deliberations. Starting with very simple, naïve approaches, I will quickly progress to more solid and useful insights.

The outcome of this process will be the introduction of two high-performance C++ libraries that deploy advanced algorithms: **MedianQuadTree** [8] and **TurboPacker** [9]. Both libraries are open-source, with links to the source code provided in the references.

Finally, I will leverage these libraries to evaluate the effectiveness of algorithmic box packing. This evaluation will include both handcrafted methods and an unsupervised learning approach.

I will conclude this thesis by recapping the work that has been completed and identifying critical areas of research that still need to be addressed.

2 Related Work

In this section, I will present papers that I found most interesting, papers that gave me fresh ideas and papers that that had considerably influence and laid the groundwork for this thesis.

The one-dimensional box packing, or bin packing, is a well-studied combinatorial optimization problem. Since it is an optimization problem, it can generally be solved using various optimization techniques. For instance, *Adriana Alvim et al.* [2] discusses strategies such as the use of lower bounding techniques, the generation of initial solutions by referencing the dual min-max problem, load redistribution based on dominance, differencing, and unbalancing, and an improvement process utilizing tabu search.

Another influential paper proposes a multi-objective variant of the recently introduced fitness-dependent optimizer (FDO). This algorithm, called the Multi-objective Fitness Dependent Optimizer (MOFDO) [1], incorporates five types of knowledge: situational, normative, topographical, domain, and histor-

ical knowledge. Additionally, the concept of weight annealing [13], commonly used in optimization problems, inspired the unsupervised learning approach used in this thesis.

When it comes to box packing, the most influential paper for this thesis is *Qiaodong Cui et al.* [5]. The method presented in this paper leverages a discrete voxel representation. They formulate collisions between objects as correlations of functions computed efficiently using the Fast Fourier Transform (FFT). To determine the best placements, they utilize a novel cost function, which is also computed efficiently using FFT. Additionally, they address interlocking detection and correction within the same framework, resulting in interlocking-free packing.

Another interesting approach is SDF-Pack, which uses a new method based on signed distance field (SDF) to model the geometric condition of objects in a container and compute object placement locations and packing orders for achieving more compact bin packing [14]. Additionally, a space-bounded $O(\frac{d}{\log d})$-competitive hypercube packing algorithm with one active bin only is presented in [7]. This algorithm is particularly noteworthy for its efficiency in online packing scenarios. Finally, a comprehensive survey covers online algorithms for bin packing and strip packing problems, including their generalizations to multidimensional bin packing, multiple strip packing, and packing into strips of different widths [12].

A vast corpus of papers about the application of machine learning in box packing exists. These studies explore learning online packing skills for irregular 3D shapes, arguably the most challenging setting of bin packing problems. The goal is to consecutively move a sequence of 3D objects with arbitrary shapes into a designated container with only partial observations of the object sequence [17]. This task requires picking variable-sized items from piles one by one and packing them into another container immediately without information about the unpicked items, modeled as an Online 3D Bin Packing Problem (3D-BPP) [16]. Another paper addresses the problem of online 2D bin packing, where the objective is to place incoming objects to maximize overall packing density inside the bin. Unlike offline methods, online methods do not use information about the sequence of future objects, making them comparatively difficult to solve. A deep reinforcement learning framework based on Double Q-learning is proposed to solve this problem, taking an image of the current state of the bin as input and outputting the pixel location for placing the incoming object [10]. Lastly, Convolutional Occupancy Networks propose a more flexible implicit representation for detailed reconstruction of objects and 3D scenes by combining convolutional encoders with implicit occupancy decoders, enabling structured reasoning in 3D space [15].

3 Theoretical Considerations

In this chapter I will discuss the thought process and theoretical work which ultimately led to the crafting of both resulting libraries. I will build from the ground up, starting with computational complexity theory, clearly define bin and box packing. Next, I will sketch out the steps that lead to definition of the tri-level convolution and end the chapter with the introduction of the costfunction placing heuristic.

3.1 The NP complexity class

In computational complexity theory a *decision problem* is a *yes* or *no* question as a function of the input values. To reach this decision a algorithm is used and its complexity is categorised in one of the various complexity classes.

First, it is important to understand what the difference between a deterministic and nondeterministic Turing machine is. A *deterministic Turing machine* is a theoretical computing machine that, given a specific input and state, can only make one possible move and transition to one possible new state, following a predefined set of rules. A CPU is maybe the closest practical implementation of a *deterministic Turing machine*.

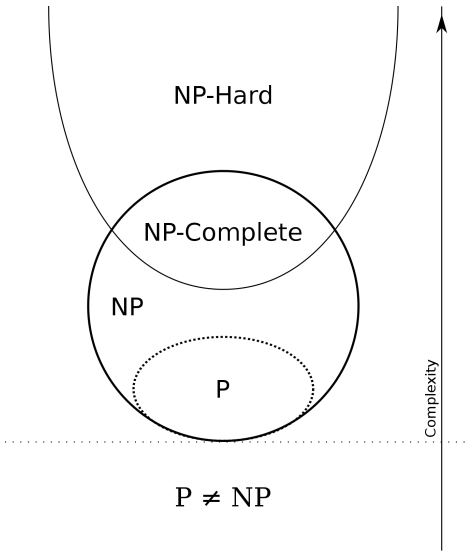


Figure 1: Euler diagram for P, NP, NP-complete, and NP-hard set of problems. [6, 11]

On the other hand a *nondeterministic Turing machine* is a hypothetical computing machine that, for each state and input, can choose from multiple possible moves or transitions, effectively exploring multiple computation paths simultaneously. Probabilistic algorithms, quantum computing or distributed parallel systems are maybe the closest approximation for such a machine.

Now, the class of decision problems that can be solved and verified by a deterministic algorithm in polynomial time are of the category **P** (Polynomial). The next bigger category which contains **P** is **NP** or the class of decision problems that can be **solved** in nondeterministic polynomial time but **verified** in polynomial time.

The two other categories are NP-complete and NP-hard. NP-complete is a subset of NP problems that are both in NP and as difficult as any problem in NP, meaning any NP problem can be reduced to them in polynomial time and NP-hard are Problems to which every NP problem can be reduced in polynomial time, but unlike NP-complete problems, they do not have to be solvable or verifiable in polynomial time themselves.

3.2 Defining 1d bin packing

1D bin packing is a classic combinatorial optimization problem that arises in various practical scenarios such as loading trucks, cutting materials, and allocating resources. The goal is to pack a set of items with given sizes while minimizing the number of fix-sized bins.

Definition 3.1. 1D Bin Packing
Given:

- The list B of m Bins with B_j as the j Bin.
- A list of n items with sizes $s_1, s_2, \ldots, s_n$ where s_i is the size of the i-th item.
- A bin capacity C.

Objective:

$$\min B$$
$$\text{w.r.t.} \sum_{i \in B_j} s_i \leq C \quad \forall B_j$$

Bin packing falls into the category of the previously discussed NP problems. The problem itself is NP-hard and the corresponding decision problem is NP-complete. The decision problem is to decide if a given set of items can fit into a single or multiple bins.
The implication of working with a NP-complete problem is that using a brute-force approach to cover the whole solution space is computational infeasible at worst and unwise at best. The exception being very small 1d bin packing problems. There are only better and worse heuristics to generate a solution that (hopefully) falls into a local maximum. Even the quality of this maximum is hard to gauge.
There already exist various algorithms to find at least near-optimal solutions. Some of the simple and easy to implement solutions are

- **First-Fit (FF):** Places each item in the first bin that has enough space.
- **Best-Fit (BF):** Places each item in the bin that leaves the least remaining space.
- **First-Fit Decreasing (FFD):** Sorts items in decreasing order and applies the First-Fit algorithm.

3.3 Defining box packing

Multi-dimensional bin packing (also known as box packing or d-dimensional bin packing) extends the classic 1D bin packing problem to higher dimensions. This problem is crucial in logistics, manufacturing, and computing, where items must be packed into bins considering multiple attributes such as volume, weight, or different spatial dimensions.

Definition 3.2. Box Packing
Given:

- The list B of m Bins with B_j as the j Bin.
- A set of n items, each represented by a vector of sizes $\mathbf{s}_i = (s_{i1}, s_{i2}, \ldots, s_{id})$ for $i = 1, 2, \ldots, n$, where d is the number of dimensions.
- A bin with capacity $\mathbf{C} = (C_1, C_2, \ldots, C_d)$ in each dimension.

Objective:

$$\begin{aligned} &\min B \\ &\text{w.r.t.} \sum_{i \in B_j} s_{ik} \leq C_k \quad \forall B_j, \quad \forall k \in \{1, 2, \ldots, d\} \end{aligned}$$

3.4 The case against dynamic programming

A first intuition is to use dynamic programming (DP) to solve the bin packing problem. I single out DP specifically because everyone who starts working with bin or box packing will immediately and without exception gravitate towards DP.

Dynamic programming is built on the principal of optimally, as formulated by Richard Bellman [3]. It states that an optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision. In simpler terms, it means that subproblems of an optimal solution are themselves optimal. It is tempting to assume that this would hold for bin packing as well. A simple counter example as proof in **Figure** 2 will show that this is in fact is not the case. With bin size 12 and items $5, 5, 7, 7$ the optimal solution will not be reached. Clearly, another strategy than dynamic programming is needed.

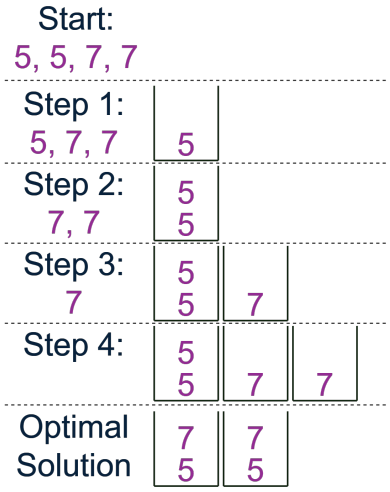


Figure 2: Simple visual proof for DP not reaching the optimal solution for bin size 12

3.5 The first, naïve approach

Returning to the box packing problem at hand, the question arises what kind of heuristic can be deployed to produce good results. It is intuitive to use the time tested method of discreticing the bins with a grid. More specific a voxel grid, as described in *Qiaodong Cui et al.* [5].

Thus, the first naïve algorithm for packing boxes into a grid is the most obvious one: Iterate over all entries of the lattice and check if the testbox fits at that position. If it fits, the box can be put into the grid and the occupancy on the lattice with box support is updated.

A first observation is that this occupancy check is equivalent to the convolution over the grid with a constant kernel the size of the test box. Let's consider the runtime of one occupancy check, which scales linearly with $O(\kappa)$ with respect to the kernel or test box size. It is immediately clear that $\lim_{\kappa \to n} \kappa = n$, where κ is the kernel size and n is the domain size. This implies that the convolution for the full domain has a polynomial runtime of $O(n^2)$. The grid's dimension does not matter; it still has n entries and can be reshaped to any dimension. However, the size of n scales with the grid's dimension, and this quickly becomes impractical as n grows exponentially. Therefore, a reduction in dimensionality is necessary.

Thus, the first change to the most naïve algorithm is to reduce the grid's dimension from 3D to 2D. Instead of storing the occupancy on a 3D lattice, we store the height on a 2D map. This gives rise to the heightmap which is defined in **Definition** 3.3.

While using a heightmap is an excellent choice, it still comes with some downsides. One major limitation is that because the packing direction is always from *above*, no box can ever be packed into an existing recess, hole, or under an overhang. This is a trade-off made to achieve high performance. For a discussion on a method that does not have this limitation, I would like to refer to the work by *Qiaodong Cui et al.* [5].

Definition 3.3. Heightmap

Let H be the height map, where $H : \mathbb{R}^2 \to \mathbb{R}$. $H(x, y)$ gives the maximum permissible height at the coordinates (x, y) on the bin's base.

Definition 3.4. Box Representation

A box B is defined by its dimensions (l, w, h) where:

l is the length along the x-axis,

w is the width along the y-axis,

h is the height along the z-axis.

$l \cdot w = \square(B)$ is its area or support

3.6 Spectral convolution

As previously discussed is the box packing problem in a wider sense the convolution of the test box (as the kernel) with the grid. One solution as introduced by *Qiaodong Cui et al.* [5] is to leverage the Fast Fourier Transform (FFT) for this. The convolution theorem 3.1 states that the convolution of a kernel with a domain can be computed by multiplying the kernel with the domain in the frequency or spectral domain.

Theorem 3.1. Convolution theorem

$$r(x) = \{u * v\}(x) = \mathcal{F}^{-1}\{U \cdot V\}$$

where $r(x)$ denotes the convolution and $\cdot$ the point-wise multiplication.

The runtime of the FFT is $O(nlogn)$. This means that with dimension reduction and by leveraging the convolution theorem the runtime can be reduced considerably.
To check if a box fits at a lattice point one might think that it holds that the height at the current position multiplied by the area of the test box equals the convolution at this point; or more formal $H(x, y) \cdot \square = r(x, y)$. However, because of the linear nature of the convolution false positives can be easily constructed (think of a stair).
Thus, a way to filter out these false positives needs to be found. This can be done by computing the convolution of the heightmap and additionally the convolution of the log of the heightmap. If it holds that $H(x, y) \cdot \square = r(x, y)$ and $log(H(x, y)) \cdot \square = log(r(x, y))$ then the solution is unique, or in other words no solution can be constructed that is not the area of test box multiplied by the current height. A formal proof attached in the appendices shows that this indeed holds.
However, even with a unique solution it is not enough, because the information is only binary: it fits, or it does not. What is needed is more information about the domain: how many lattice point with kernel support are lower, higher or exactly the current height on the heightmap. I will call this more granulated convolution **tri-level convolution**. However, no kernel can be constructed that would yield these information and makes the spectral convolution sadly unsuited. How the tri-level convolution can be computed efficiently will be discussed in the next section in detail.

Definition 3.5. *Tri-Level Convolution*

Let $H(x, y)$ represent the values from the heightmap H, D the domain, $(\hat{x}, \hat{y})$ the coordinates withing the support region $\text{supp}(K)_{(x,y)}$, $K \in \mathbb{R}^2$ the kernel and $h \in \mathbb{R}$ the given height. Then the tri-level convolution $\{l, m, h\}$ is defined as

$$l = \{H(\hat{x}, \hat{y}) \mid (\hat{x}, \hat{y}) \in \text{supp}(K)_{(x,y)}, \forall (x, y) \in D, \text{ and } H(\hat{x}, \hat{y}) < h\}$$
$$m = \{H(\hat{x}, \hat{y}) \mid (\hat{x}, \hat{y}) \in \text{supp}(K)_{(x,y)}, \forall (x, y) \in D, \text{ and } H(\hat{x}, \hat{y}) \equiv h\}$$
$$h = \{H(\hat{x}, \hat{y}) \mid \hat{x}, \hat{y}) \in \text{supp}(K)_{(x,y)}, \forall (x, y) \in D, \text{ and } H(\hat{x}, \hat{y}) > h\}$$

3.7 Placing heuristic

Once the tri-level convolution is computed a heuristic is needed to decide which of the candidate position is the current best pick and will lead to the overall best result. Various heuristics can produce good approximations for the box packing problem. I focus on the application of a costfunction, which is very simple to implement and produces powerful insights and surprising results.

The general idea of the costfunction is to map a tuple to a real valued cost. This tuple, which I will call **Results Tuple** $\mathcal{T}$, is a set that contains all spatial information relating to this very specific lattice point. This information is gathered by the algorithm and fed into the costfunction. A exhaustive list of members can be found in **Definition** 3.6.

With knowledge about the tuple a simple definition of the packing problem can be reformulated as stated in **Definition** 3.7. This expression states that we seek to find the sequence of tuples over n steps that minimizes the total cost summed over all coordinates on the domain. Or simpler, instead of maximising the sum of volumes of the boxes, we seek to minimize the total cost. Additionally, any function that maps the tuple to a real valued value is admissible as a costfunction. Some example costfunctions can be found in **Definition** 3.7. Their performance will be discussed later.

Important: While it is tempting to assume that minimizing the cost is equivalent to maximising pack density, it is not clear that this holds. However, for practical reasons I believe this to be a valid assumption and all the implemented algorithms work under this paradigm.

Definition 3.6. Result Tuple

$\mathcal{T}_{i,j} = (\textit{bin}, \{eX, eY\}, \{x, y, h\}, \{l, m\}, \{\hat{l}, \hat{m}, \hat{h}\})$

with

bin the id of the current bin,

$\{eX, eY\}$ the extends of the test box (depends on rotation of the test box),

$\{x, y, h\}$ the position vector of this tuple on the heightmap,

$\{l, m\}$ the overlap from the tri-convolution (h is omitted because $h > 0$ is an illegal state),

$\{\hat{l}, \hat{m}, \hat{h}\}$ the tri-convolution of the border (grid points surrounding the box).

Definition 3.7. Reformulated Packing problem

Let D the domain, (x, y) the coordinates on the domain, $C : \mathcal{T} \to \mathbb{R}$ the costfunction and $\mathcal{T}$ the result tuple. Then the reformulated packing problem is defined as

$$\min_{\{\mathcal{T}\}_{k=1}^{n}} \sum_{k=1}^{n} \sum_{(x,y)\in D} C\left(\mathcal{T}_k(x, y)\right)$$

Definition 3.8. Example cost functions:

$$\mathcal{C}_{Constant}(\mathcal{T}_{i,j}) = 0 \cong x + y * Width$$
$$\mathcal{C}_{BottomLeft}(\mathcal{T}_{i,j}) = (x + y)^2$$
$$\mathcal{C}_{BottomLeftWidthHeight}(\mathcal{T}_{i,j}) = h^3 + (x + y)^2$$
$$\mathcal{C}_{Krass}(\mathcal{T}_{i,j}) = (x + y)^2 + h^3 - (\hat{l} + \hat{h})^3 + \hat{m}^3 - (eX + eY)^2 + (10 * bin)^3$$

The $\mathcal{C}_{Constant}$ cost function simply returns 0. However, due to the implicit order in which the algorithm iterates over the domain and inserts results into the array, this order approximates a linearized index.

The $\mathcal{C}_{BottomLeft}$ cost function penalizes the quadratic distance or the quadratic radius from the origin of the domain, thereby forcing all boxes into the bottom-left corner. This can be further enhanced with a height term, as seen in $\mathcal{C}_{BottomLeftWidthHeight}$, which encourages boxes to be positioned as close to the ground as possible.

While more sophisticated cost functions can be constructed, the high dimensionality and non-intuitive nature of the results make this challenging. Nevertheless, the $\mathcal{C}_{Krass}$ cost function is an example of a handcrafted approach. The aim is to force all boxes into a corner while penalizing height to encourage layered packing from the ground up. Additionally, it rewards having more border contact around a box, reducing the cost, whereas no border increases the cost. The cost is also reduced by the square of the size to bias towards the largest side of the box. Finally, there is a bias towards placing the box into the bin with the lowest index.

While this may seem straightforward in theory, it is quite complex in practice. It is unclear how to scale the individual components and how they interact, or even if this is an effective cost function. A potentially better approach to this problem is presented later.

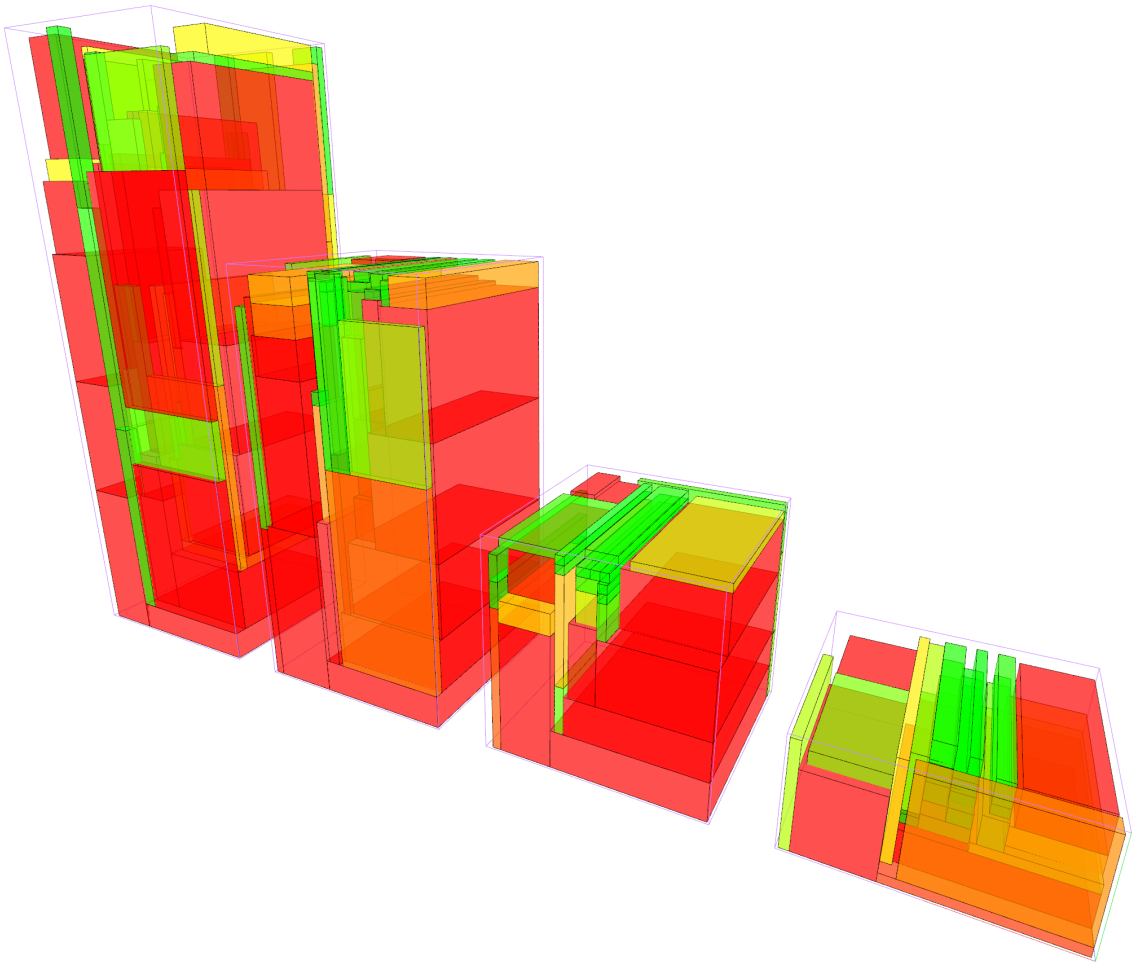


Figure 3: (Visual Break) 182 random boxes in a four bins with overall density of 82%. Color coded by volume (red is larger).

4 Implementation

In this section I will first present the MedianQuadTree library which solves the tri-level convolution in $O(\sqrt{n})$, I will discuss the implementation in great detail and finally, compare the performance to a naïve implementation of the tri-level Convolution. Next, I will present the TurboPacker library which uses the MQT library as backend and provides high-level utility to solve the box-packing problem using the costfunction heuristic.

4.1 MedianQuadTree

A MedianQuadTree or MQT for short is a data structure that allows to compute the tri-level convolution for any height map in a efficient and thread-safe manner. The MQT serves as back-end for the TurboPacker library. The goal of this chapter is that the inner workings and the source code of the MQT can be understood and used.

4.1.1 Algorithm

A initial observation is that computing the tri-level convolution the naïve, or brute force way is in any case memory bound. The operational intensity, or operations per byte is so low with between 0.5 and 2 operations per byte (depending on type used) that the only way to improve performance is to cleverly reduce the memory transferred.

Algorithm 1 Naïve Tri-Convolution

Input: heightmap, kernel size, test height h
Output: tri-convolution
for every value v on the heightmap with kernel support **do**
  if $v == h$ **then**
    $m++$
  else if $v > h$ **then**
    $h++$
  else
    $l++$
  end if
end for
return $\{l, m, h\}$

An idea to reduce memory transfer is to abstract away flat parts of the heightmap using a tree topology. Imagine a tree structure closely related to a quadtree, but instead of a leaf node representing a single cell, it manages a small region. This leaf node is called a *bucket*.

A simple example of how this proposed topology might look is shown in **Figure** 4. Leaf nodes, or buckets, manage small regions, and together these small regions cover the entire domain. Similar to a quadtree, the parent node manages the combined regions of its children, with the root node managing the whole domain.

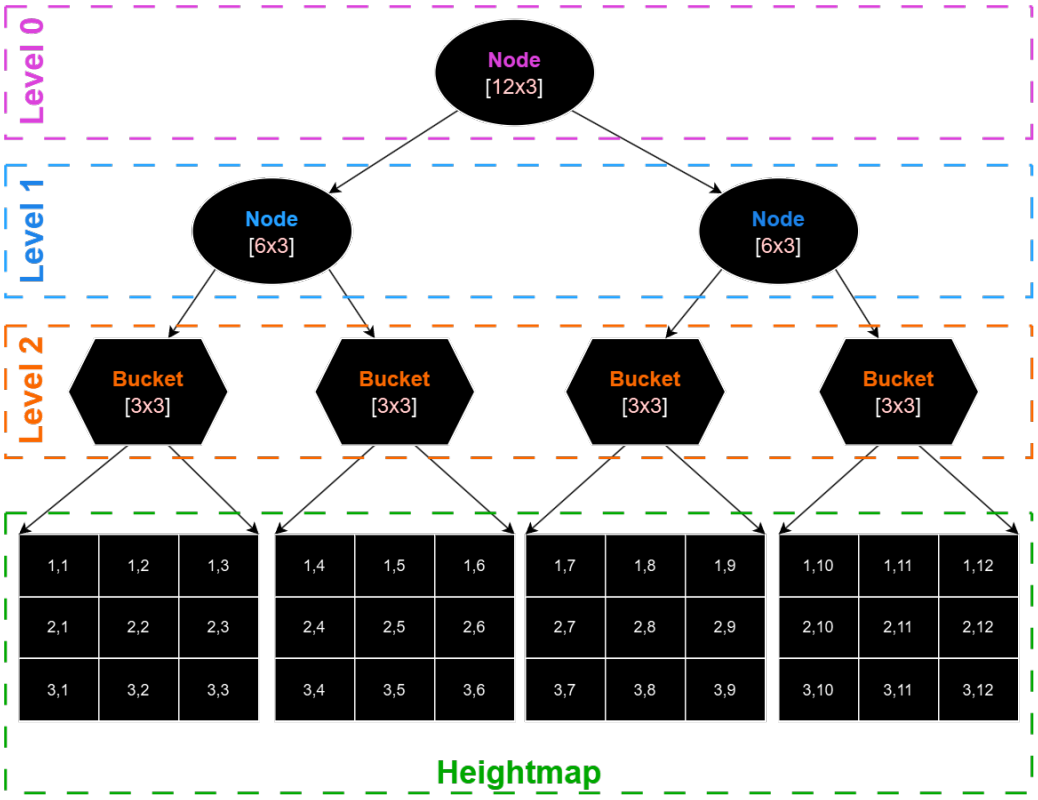


Figure 4: Topology of the MedianQuadTree

The question arises of how exactly this *abstracting away flat regions* can be achieved, how the tree can be implemented, and how functionalities like rebuilding the tree or computing overlaps actually works.

Overlap

The process of abstracting away flat regions is best explained through the overlap mechanism. In **Figure** 4.1.1 and **Figure** 4.1.1, detailed flow chart of the overlap algorithm can be studied and will help understanding the following description.

At its core, the algorithm is recursive, starting at the lowest node level (root). If the heightmap at this level is too noisy, the algorithm moves up a level, increasing the resolution of the heightmap until the minimal resolution needed to compute the tri-level convolution is reached. In the best case, this can be achieved at the root node, and in the worst case, at the highest level within the buckets or leaf nodes. This approach ensures that computational power is only applied where the frequency of the heightmap is sufficiently high.

In more detail, an overlap query starts with the root node. Each node, in addition to its bounds, stores the minimum and maximum values. If the height provided by the query does not fall within this interval and the kernel completely overlaps the node, the tri-level convolution can be computed analytically. Additionally, if the domain managed by the node is flat (i.e., the minimum equals the maximum), the convolution can be computed instantly as well.

If the node is not flat and the height falls within the interval, or if the kernel only partially overlaps the node, the query is handed over to the node's four children. If the children are nodes, the same algorithm is applied to them recursively. If the children are buckets and the maximum level is reached, the computation changes slightly.

Again, if the kernel support overlaps the bucket completely and the height is outside the interval, or if the bucket is flat, the solution becomes trivial. However, if the support is partial, the only option is to brute-force this patch of the heightmap. If the support is equal or larger as the bucket size and the values stored in the bucket are monotone (i.e., sorted), we can compare the height to the median saved in the bucket, reducing the brute-force work by 50%.

Rebuild

Rebuilding the tree is exceptionally straightforward. Unlike the overlap process, rebuilding follows a bottom-up (recursive) approach. Buckets calculate their interval and median, then pass this information up to their parent nodes. This method is extremely efficient because only the parts of the tree that were changed need to be updated or rebuilt.

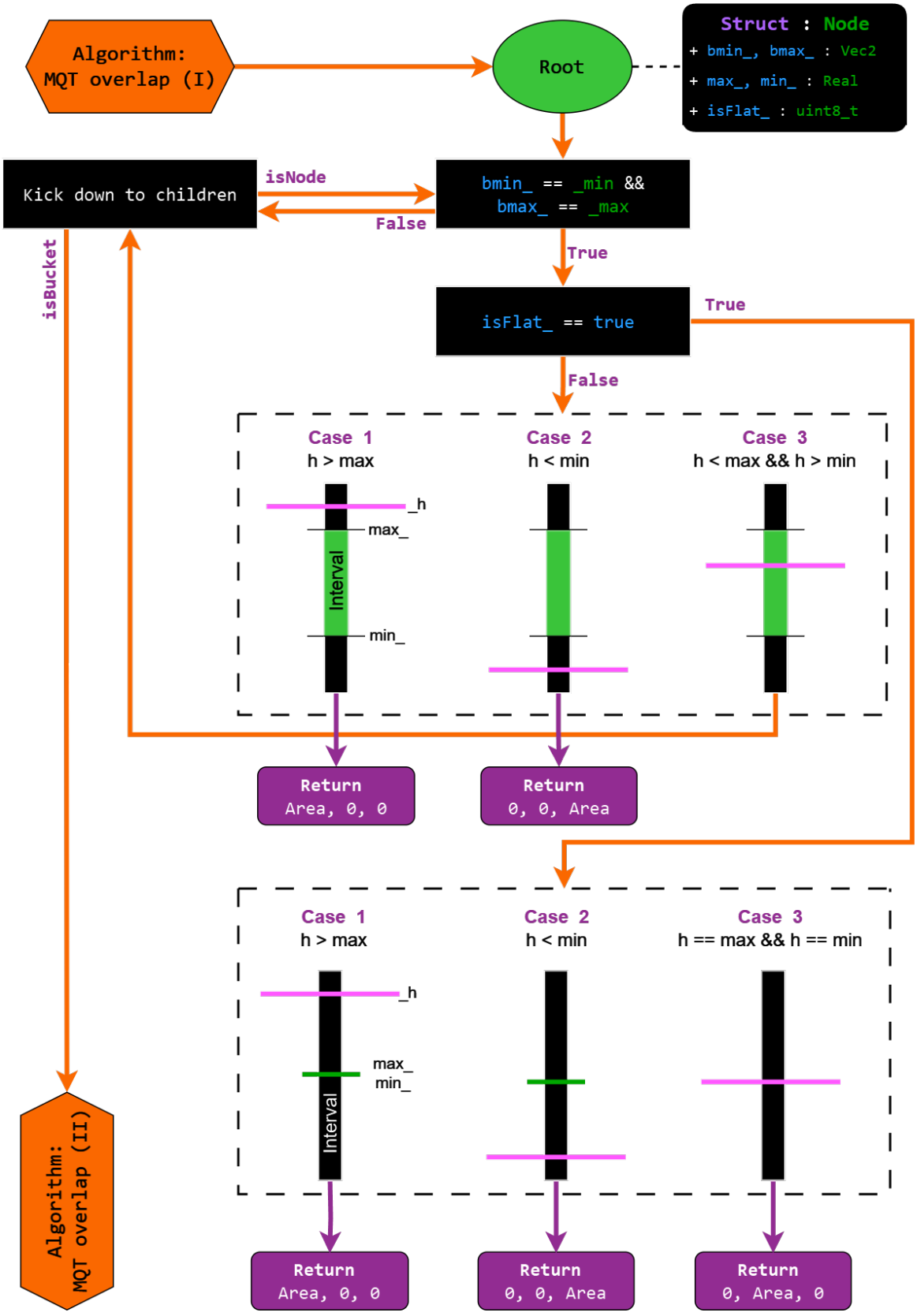


Figure 5: Flow diagram of the overlap algorithm for Nodes

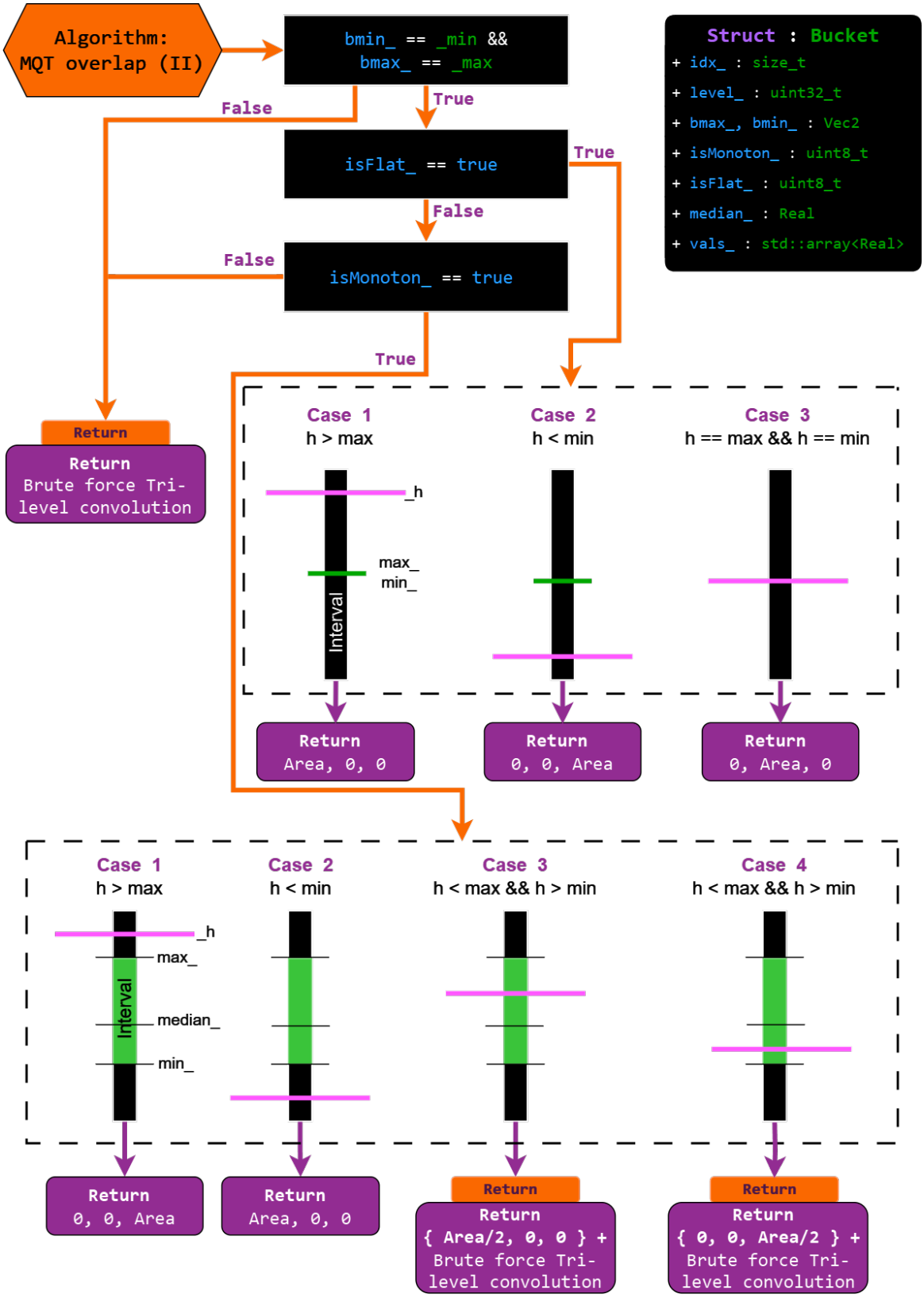


Figure 6: Flow diagram of the overlap algorithm for Buckets

Implementation

The MQT implementation offers a simple and clean interface, as shown in **Figure** 7. Additionally, a minimal C++ example can be found in the Appendix. The functionality of the various template parameters is discussed in more detail in the next TurboPacker section.

A last important thing to note is that the domain managed by the MQT needs to be of size 2^n and $\frac{2^n}{bucketsize} \equiv 0$. While these constraints seem restricting, it is not a real issue because smaller or uneven domains can be managed by just using a subset of a larger domain.

Definition 4.1. *Max depth of the MQT*

For given bucket size β and domain size η the max depth δ is

$$\delta = log_2(\frac{\eta}{\beta}) \quad \text{w.r.t.} \quad \eta \mod \beta \equiv 0$$

4.1.2 Performance and Scaling

All benchmarks are conducted on Arch Linux using a 16-Core AMD Ryzen 9 7950X, which is a Zen4 architecture with a base clock of 4.5 GHz and 1 MB, 16 MB, and 64 MB caches, respectively. Precision Boost Overdrive (PBO) and Core Performance Boost (CPB) are deactivated. The system uses DDR5-6200 memory.

Accurately benchmarking the MQT is not straightforward because its performance scales with tree depth. The bucket size needs to be chosen based on the domain size and the usual kernel size. Testing has shown that a tree depth of 8-10 yields good results. In these benchmarks, the MQT has a bucket size of 15 and a depth of 9. The depth can be easily computed for given domain size and bucket size with **Definition** 4.1. A bucket size of 15 has proven to be an overall good choice.

Furthermore, while the naïve approach does not scale with the data, the MQT does. The more flat areas and structure the data has, the better the MQT will perform. In the benchmark, 500 random boxes with no concern for overlap are written onto the heightmap, resulting in a dataset best described as unstructured or 'wild'. It is theorized that a heightmap could be constructed such that the MQT converges towards the naïve solution in runtime.

In **Figure** 8, we compare the benchmarks of the naïve approach versus the MQT. The upper plot compares smaller kernel sizes, while the lower plot compares larger kernel sizes. Various data types are compared. The first observation is the difference between floating point types and integer types. Additionally, smaller data types, byte-wise, result in faster performance due to reduced memory transfer.

This indicates that both algorithms are indeed memory-bound.

Secondly, it is evident that the MQT is always faster. **Figure** 9 shows that the relative speed depends heavily on the kernel size. Even in the worst case, we observe a speedup of at least 5x, with an overall speedup of around 10x. The speedup seems to flatten towards larger kernel sizes for reasons previously discussed. The outliers showing a 10x speedup should be taken with a grain of salt as they are most likely caused by cache effects and may not be representative.

As mentioned before the naïve algorithm scales linearly with $O(n)$. For the MQT, scaling is not as straightforward. I propose that the scaling is $O(\sqrt{\kappa})$. The scaling observed in the lower plot of **Figure** 8 seems to support this claim. A more theoretical analysis is as follows:

Theorem 4.1. Let κ by the width of a squared sized kernel and let n be the width of a squared size domain. Let the data on the domain be flat.

Then it follows from the nature of the MQT that only values on the edge of the kernel are read. Thus the formula for the scaling becomes $O(\lim_{\kappa \to n} 4 \cdot \sqrt{\kappa} + C)$ which simplifies to $O(\sqrt{n})$.

In conclusion we can observe a $O(\sqrt{n})$ scaling, a 10x speed up and overall outperforming of the naïve solution on all levels.

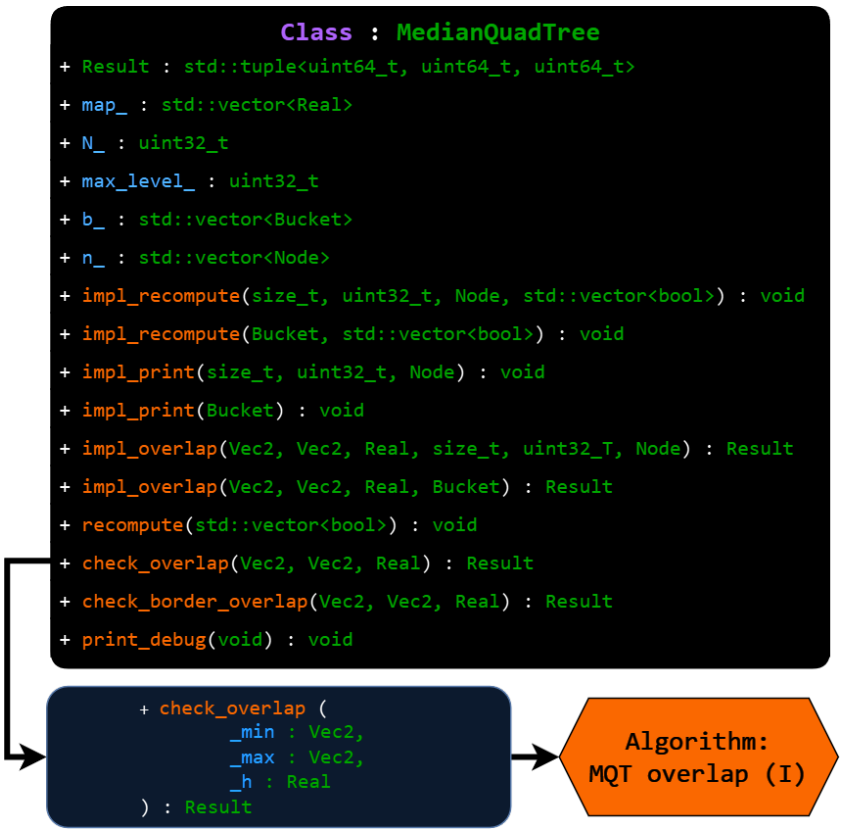


Figure 7: The MQT C++ Interface

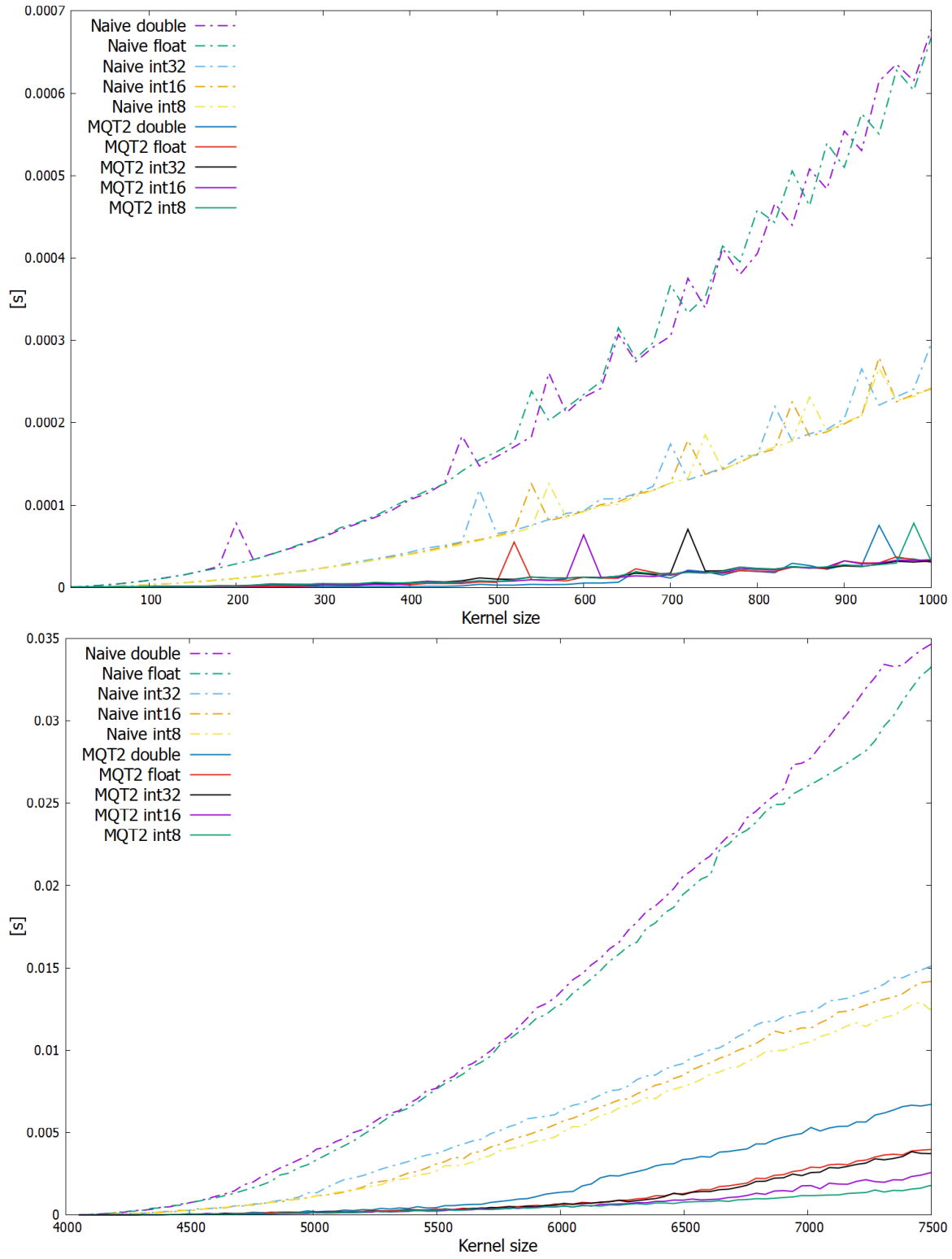


Figure 8: Naïve vs MQT2 Performance. (Lower is better)

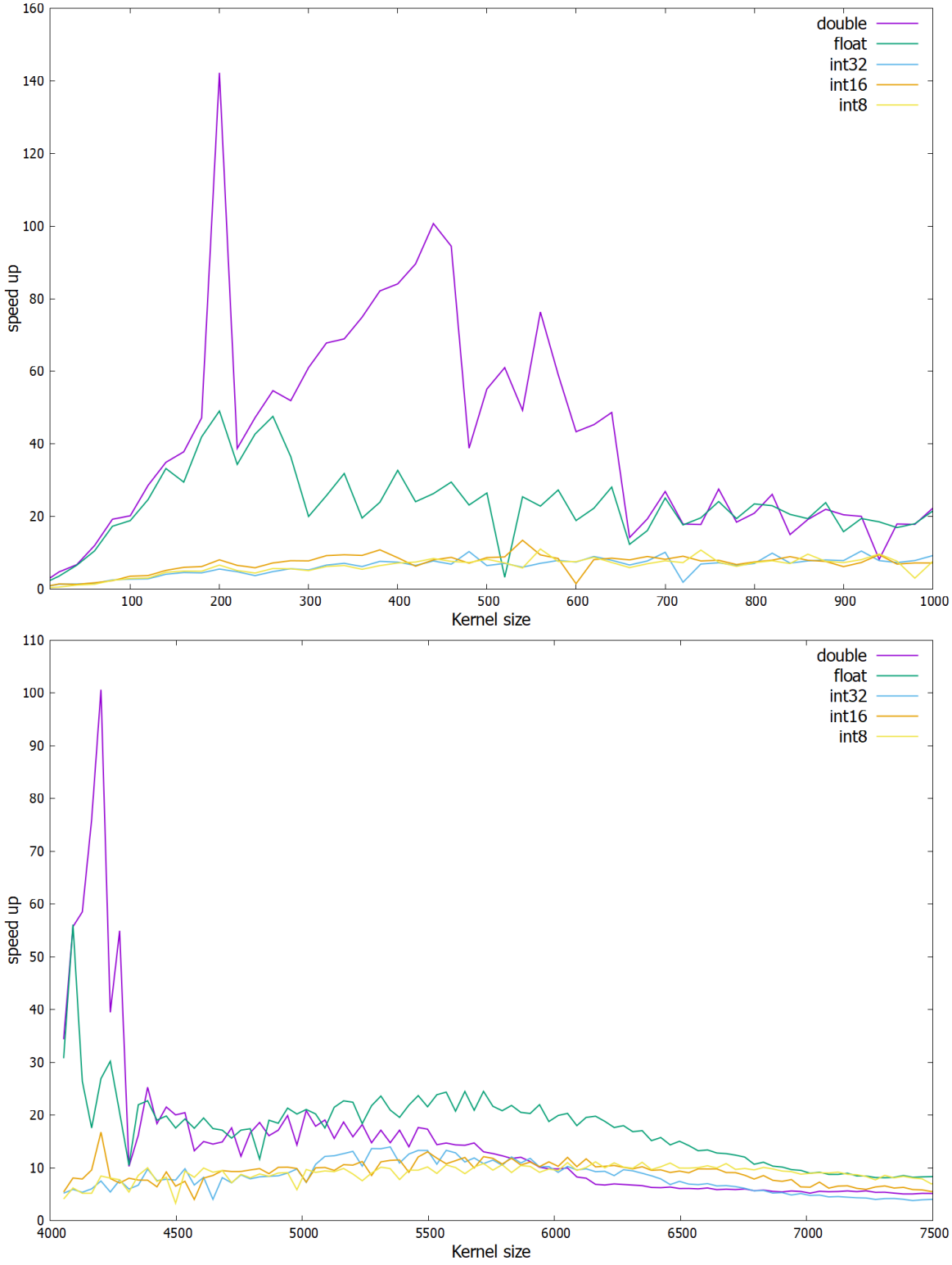


Figure 9: Naïve vs MQT2 speed up. (Higher is better)

4.2 TurboPacker

The TurboPacker is a high level library built upon the the MedianQuadTree (MQT) backend. It allows for simple integration into existing rendering front ends or any other application. It facilitates a rich interface for to pack boxes of various kind into one or multiple bins.

4.2.1 Library Features

The design philosophy of TurboPacker emphasizes ease of use. To achieve this, all configuration options are centralized in the Config struct, and the packing process is initiated in a fire-and-forget manner. The returned Promise provides an interface to query information about the progress and status of the process. Additionally, the underlying algorithm can be fine-tuned and adapted using static template parameters. These template parameters are all used to fine tune the MQT backend and are the same when using the MQT on its own.

Definition 4.2. TP configuration
Static options:

- **Type T:** The underlying floating type used. This allows the use of double precision if so desired.
- **Higher-kinded Type COSTFUNCTION:** The costfunction used.
- **Type HEIGHTMAP_T:** The underlying type used in the MQT backend. For most use cases an unsigned 16bit integer is more than enough.
- **Type R_T:** The return type of the MQT tri-convolution queries. A unsigned 32bit integer type is preferred to avoid overflow for larger queries.
- **unsigned 32bit integer BUCKET_SIZE:** The size of the buckets used in the MQT backend.
- **Type HEIGHTMAP_ALLOCATOR:** The allocator used by the MQT backend. In most cases this will be simply the default allocator, but allows to use an aligned heightmap.

Runtime options:

- **MultiThreading:** If the solver should use worker threads [default: true]
- **NumThreads:** Number of worker threads [default: 4]

- **UseRandomSeed:** If the solver should use a random seed [default: true]
- **Seed:** The seed [default: 1234567890]
- **Bins:** The defined bins.
- **BoxType:** If random boxes or a pre-defined list of boxes should be used [default: Random] (Hint: the VALIDATION option is only used in the app itself and not by the library.)
- **AllowOverlap:** If boxes are allowed to overlap [default: false]
- **EmptryTries:** How many misses in a row before terminating [default: 25]
- **MaxEmptryTries:** How many misses in total before terminating [default: 50]
- **LookAheadSize:** How far the solver can look ahead [default: 1]
- **AllowedPermutations:** Bit flags to control the permutations [default: All]
- **CubeRandomBoxes:** If the random boxes should be cubes [default: true]
- **MinBoxVolume:** The minimum volume of the random boxes [default: 250]
- **MaxBoxVolume:** The maximum volume of the random boxes [default: 1500]
- **BoxList:** The list for predefined boxes
- **EnforceMisses:** If it should enforce emptry tries. if false the solver runs until the list is empty [default: false]
- **EvalBoxCount:** How many boxes the ground truth should contain [default: 12]

The call structure and usage of the Config is shown in **Figure** 10. It also shows highlights the interface of the promise. I believe it to be so straight forward that it does not warrant any further description here.

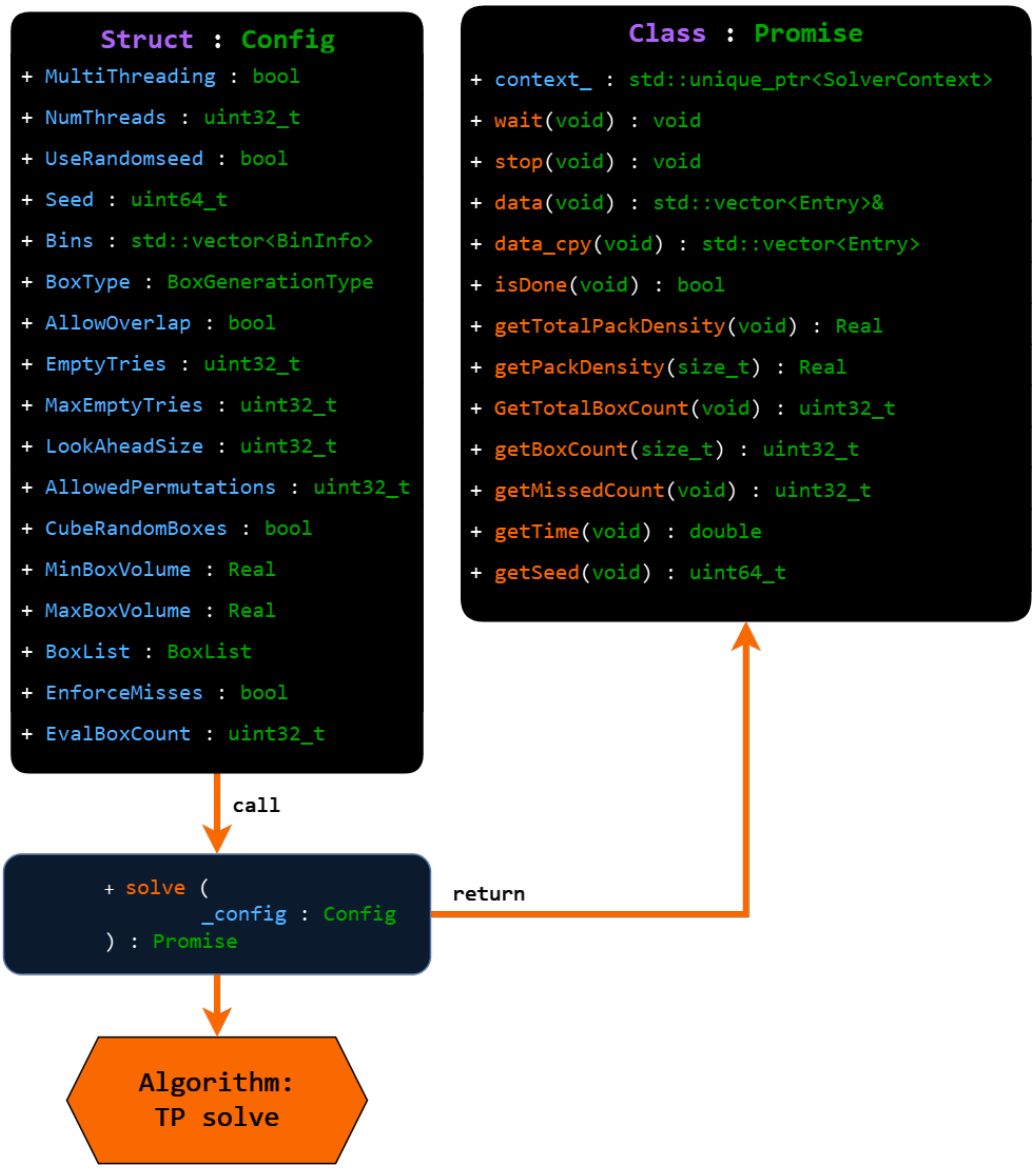


Figure 10: The TP usage flow diagram in C++ syntax

4.2.2 App Features

For every mode the relevant options from the Config are automatically exposed and editable. Additionally it offers feedback about packing density, box count and time. The App features three modes:

Random mode

Random boxes are generated and packed into the bins until full or breaking conditions are reached. It is to note that the PackerWidget exposes all the relevant entries of the Config struct 4.2 for the selected mode.

The application supports multiple bins, each of which can be set to an individual size and will be rendered in a row. Every bin has a unique ID and can be accessed in the Result struct.

While the application supports cubes I deem them only of marginal interest and will not be discussed further. The packing of cubes can be solved analytical in many cases and overall belong to a different category of problem. For the interested reader I want to point to the excellent discussion on this topic by *Grzegorek et al.* [7].

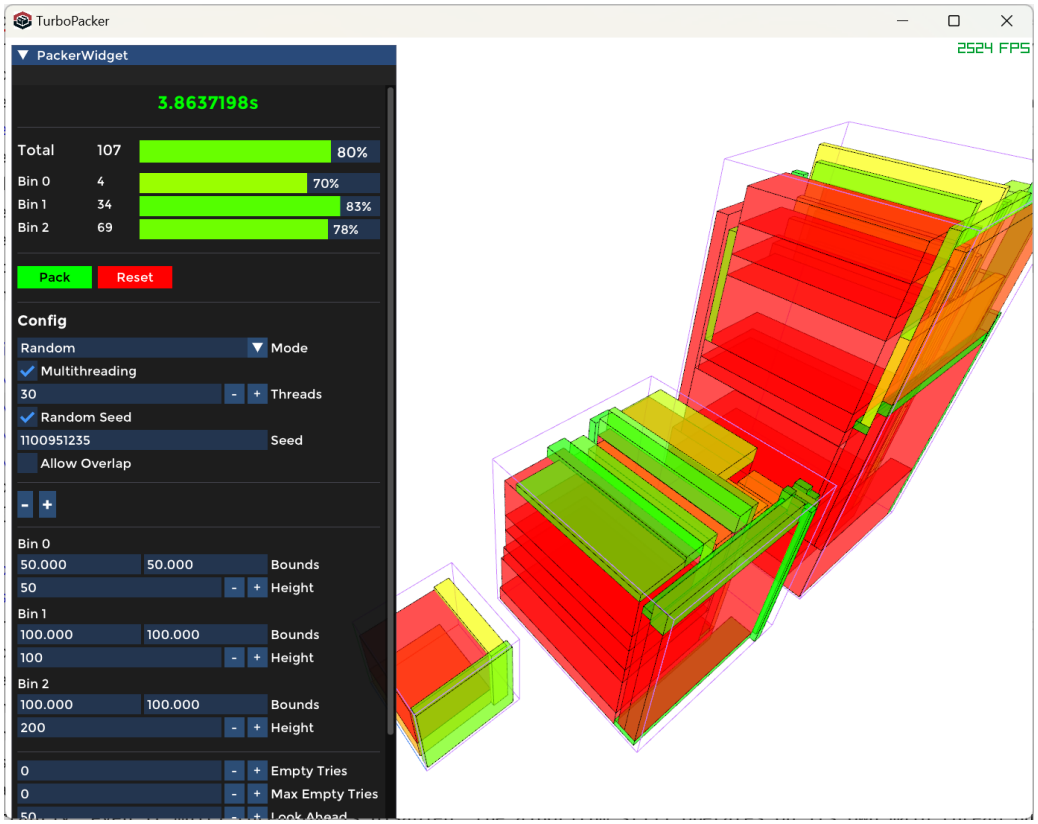


Figure 11: Random boxes packed into 3 bins: 50^3, 100^3 and 100^2x200 with total 80% density.

List mode

A predefined list of boxes is packed into the bins until full or breaking conditions are reached. By default the App features the following list, which is a subset of box sizes used by the Swiss Post.

Definition 4.3. The Postpacs List

All units in grid size. This is mostly centimeters.

- 28 x 17.4 x 10
- 35.5 x 24 x 12.5
- 38 x 35 x 16.9
- 53.5 x 28.5 x 16.5
- 39 x 13 x 11
- 55.5 x 37 x 6

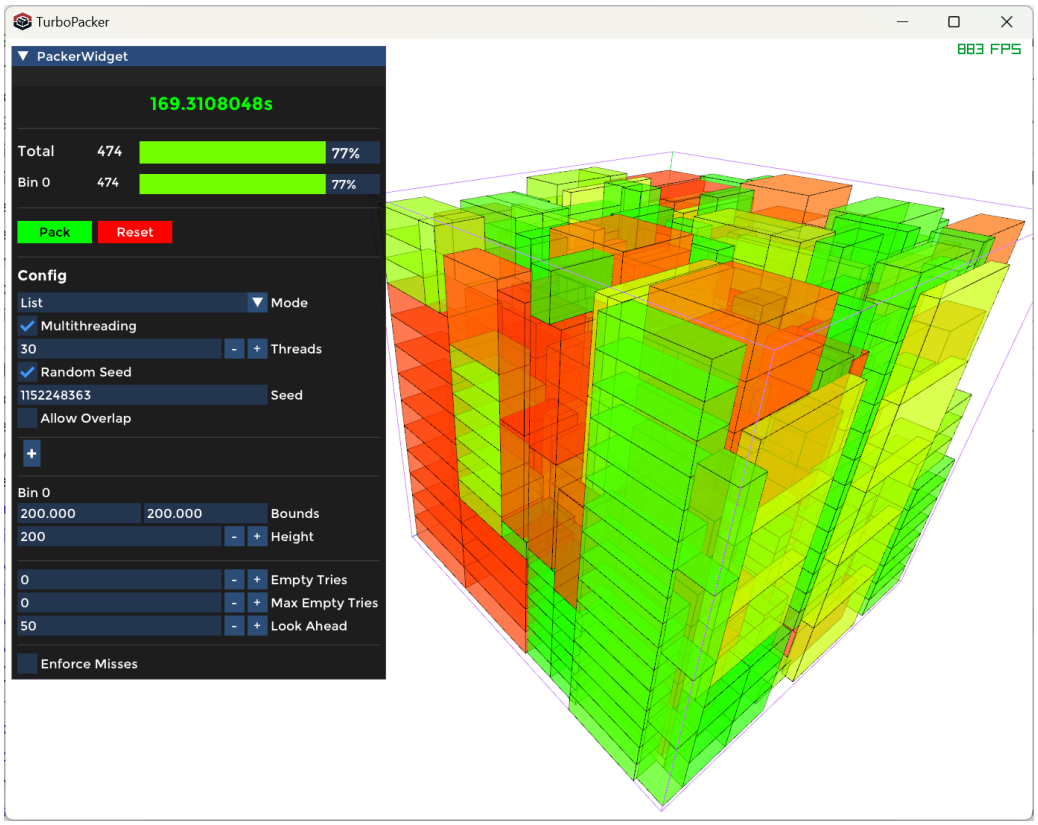


Figure 12: A cube of size 200^3 packed with the Postpacs list achieving 77% density and 474 boxes packed.

Validation mode

A groundtruth is generated and given to the packer. The concept behind the validation mode is to provide the algorithm with a known solution. This enables the testing of cost functions or can be used for data-driven approaches.

To generate a known solution or groundtruth, I have implemented the **squarified treemap** algorithm [4]. This algorithm decomposes given rectangle sizes while maintaining aspect ratios as close to 1 as possible. As a side note, the creation of such a decomposition is classified as an NP-hard problem.

Since validation essentially involves simple list packing, the same options can be used. The primary difference is that a problem must be generated with options to modify the box count and visualize the solution. Each rectangle is assigned a unique color to help identify the packed boxes. This mode only supports a single bin.

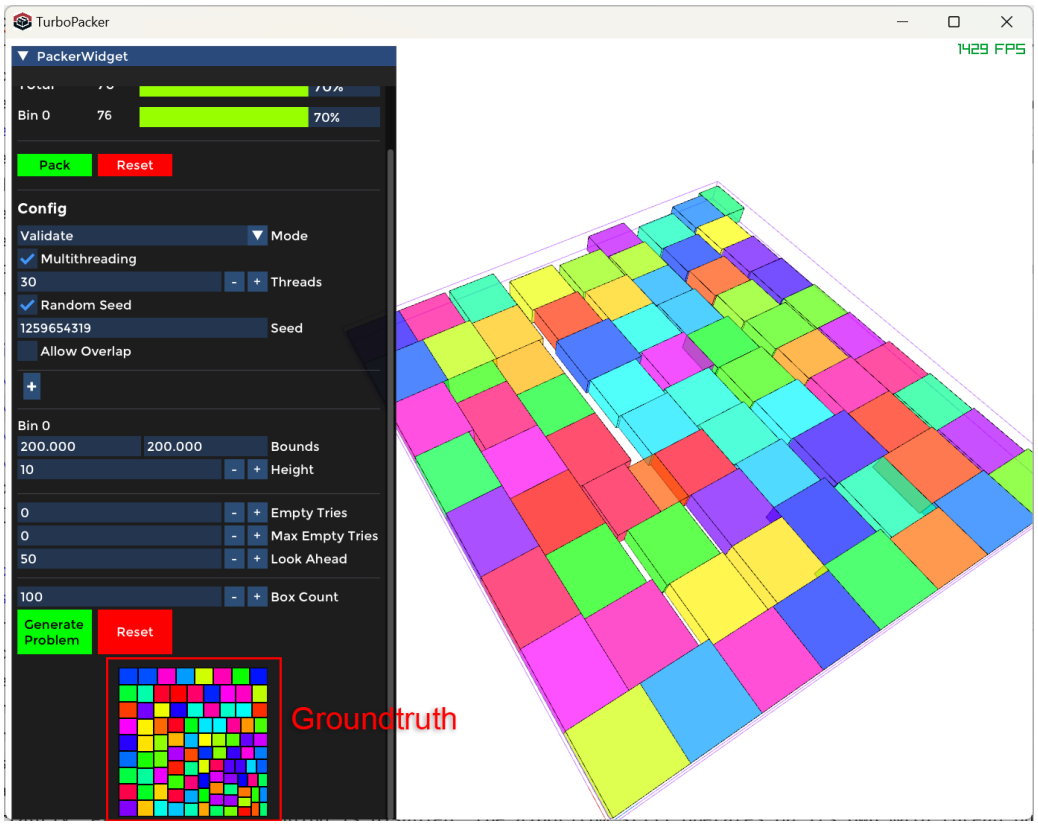


Figure 13: Validation mode packing of 50 rectangles achieving 70% density.

4.2.3 Inner workings

As previously discussed, the library operates as a self-contained, fire-and-forget system. Multiple instances can be initiated simultaneously through multiple calls to the solve function. The configuration (Config) is copied, allowing for safe modification during solver execution. Each instance is associated with a Promise, which must be maintained, as the destruction of the Promise instance results in the solver being halted and deconstructed. The Promise is only movable and not copyable, extremely lightweight, and occupies only 8 bytes.

The algorithm executes asynchronously and manages its own worker threads as specified in the Config. Importantly, even if multi-threading is disabled, the algorithm still operates on its own main thread but does not spawn additional worker threads.

In **Figure** 4.2.3, the algorithm's operation on the main thread is shown. Notably, for the look-ahead process, the algorithm generates a list of boxes and produces Results for each entry. The optimal Result is retained, and the remaining boxes are pushed back to the front of the list. The algorithm terminates once the list is empty.

In **Figure** 4.2.3, the process of generating Results by the algorithm is detailed. It is crucial to note that the cost function is invoked for each Result. While it is possible to create cost functions with states, these states must be synchronized manually.

Using the library has been designed to be as simple and straightforward as possible. To add a custom cost function, follow the example provided in **Definition** 7.2. It is important to note that the template parameter **T** is required and cannot be specialised in the definition. As long as the function is templated and has the correct signature for *eval*, it will work.

In **Definition** 7.3, a minimal example demonstrates how to add and use the TurboPacker library. Most values in the Config have reasonable default settings, as shown in **Definition** 4.2. With just these few lines of code, the library can be up and running. For a more comprehensive example, refer to the *main.cpp* file.

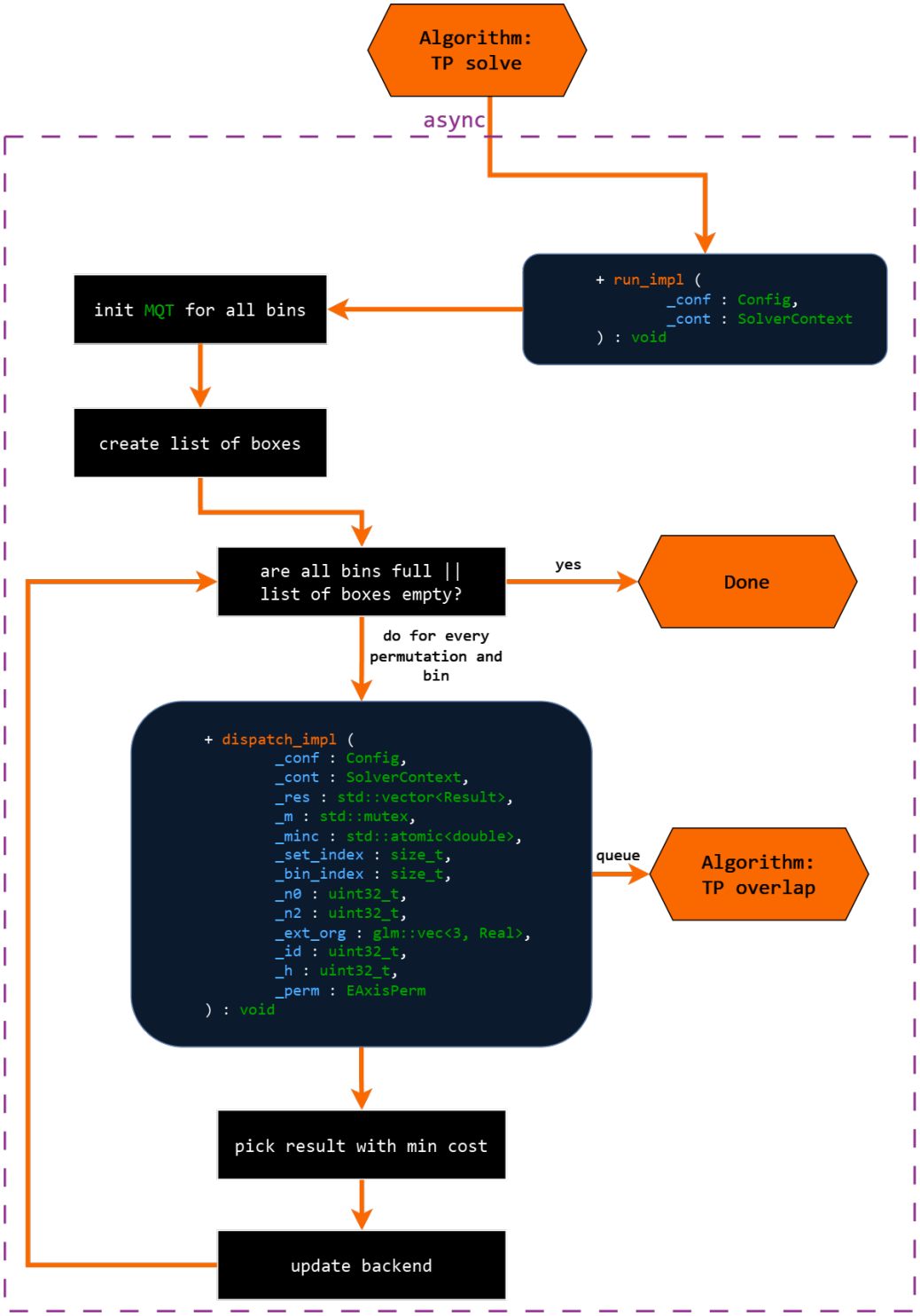


Figure 14: The main loop of the TP packing algorithm.

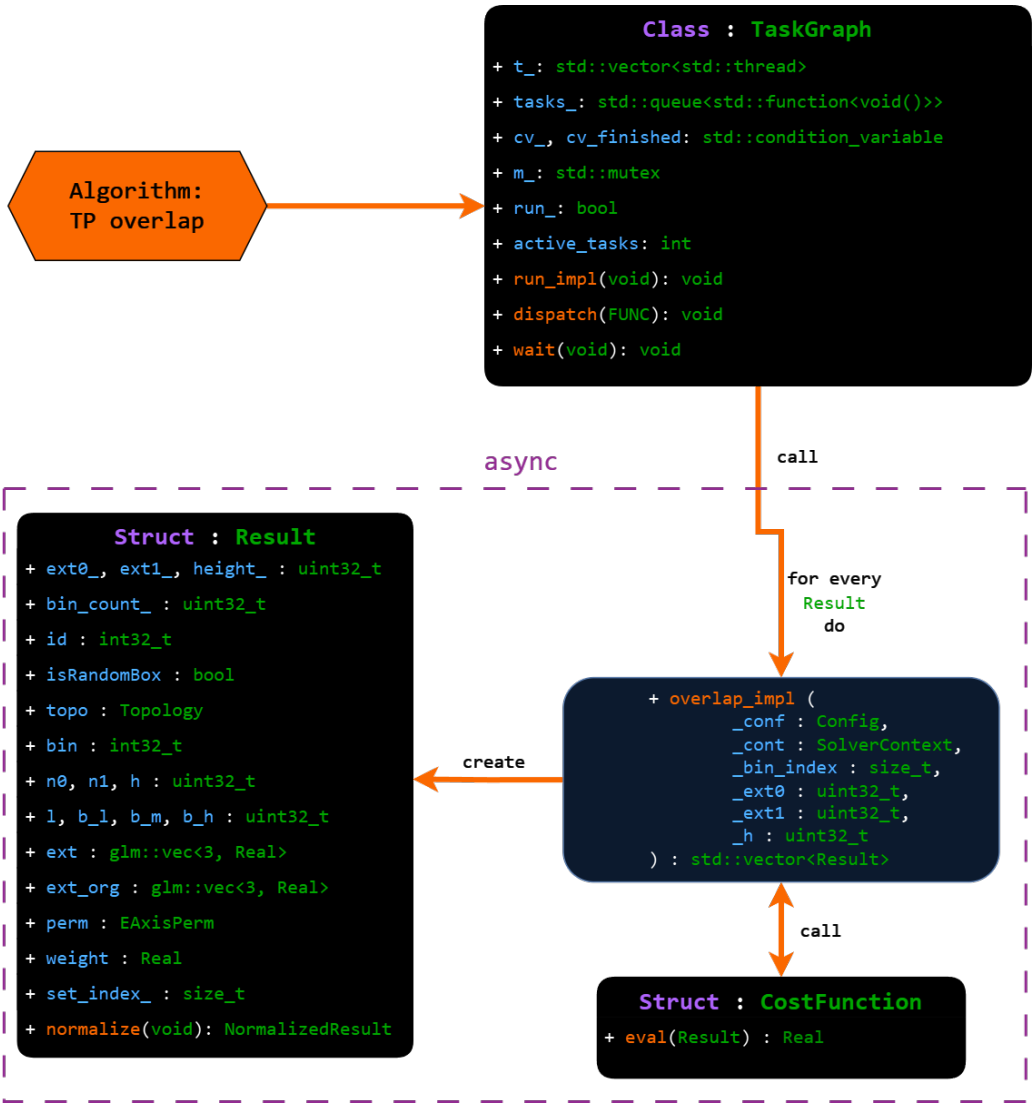


Figure 15: The TP overlap subroutine.

4.2.4 Performance

It is challenging to gauge the performance of the TurboPacker due to the numerous parameters involved, including the OS kernel's thread management. Therefore, I will present some performance numbers in **Figure** 4.2.4 with following configuration: Domain size: 100^3, List: postpacs, random size: $5^3 : 15^3$, look ahead: 1, 5, 25, 50, threads: 8. The same constraints about the machine running the benchmark apply as those laid out in the benchmark section of the MedianQuadTree.

	50		100		150		250	
	Look Ahead	[s]	Look Ahead	[s]	Look Ahead	[s]	Look Ahead	[s]
Random	1	7.56E-05	1	0.2080271	1	3.496468	1	55.1684982
	5	0.0130949	5	2.2373048	5	21.2148059	5	238.60062
	25	0.0670755	25	13.401049	25	115.208105	25	1755.15821
	50	0.0965517	50	24.9608278	50	233.290843	50	3331.2916
List	1	0.1842045	1	10.4798352	1	34.2894181	1	154.0422
	5	0.2448516	5	7.6038228	5	23.4701651	5	188.32654
	25	0.1642297	25	8.1140946	25	55.6550805	25	492.995719
	50	0.2802442	50	10.7056277	50	74.8598612	50	1024.57872

Figure 16: TP packing performance. Domain size: 100^3, list: postpacs, random size: $5^3 : 15^3$

The first interesting observation is that the random boxes seem to be overall faster. Although these boxes are packed more quickly, a greater number of them need to be packed. It appears that packing more smaller boxes is faster than packing fewer larger boxes. This is expected because having more small tasks allows better utilization of worker threads and avoid large, and thus, slow overlap queries. Another observation is that doubling the look-ahead, thereby doubling the work, does not necessarily result in a doubling of the time required. Except for the domain size of 250, the time increase from a look-ahead of 25 to 60 for the list boxes is less than 100%. In some cases, the time even decreases with more look-ahead. This can be explained by the overhead of synchronization when there is insufficient work.

A third observation is that doubling the domain width, which results in four times the domain area, incurs significant performance hits. For domain sizes over 50, the performance degradation is so severe that they are not usable for time-critical applications.

These numbers provide a general sense of the TurboPacker's performance under various configurations and sizes, but are far from conclusive. Indeed, the actual performance can vary significantly depending on specific use cases and system conditions.

5 Experiments with Costfunctions

In this section, I will present the experiments conducted using the TurboPacker library. First, I will discuss the results of the analytical cost functions, as detailed in **Definition** 3.8. Although not exhaustively explored, these experiments highlight several shortcomings and problems. These findings will lead directly to the next step, where I will introduce the initial design and implementation of an unsupervised learning approach using annealing and ascending random walk. Finally, I will leverage the results of the unsupervised learning approach to demonstrate the potential for the existence of a global maximum and discuss its implications.

5.1 Analytical Costfunctions

In this section, I will present the results achieved by the analytical costfunctions presented in **Definition** 3.8 and discuss their performance.

With two different tests, we can gauge the quality of each cost function in terms of mean and variance over 150 packs. The domain size is 100^3 in all cases, with look-ahead values of 1, 5, 25, and 50, respectively.

In **Figure** 5.1, the results of four analytical cost functions and an annealed cost function (to be discussed later) are presented. The default list of boxes is used, with the number of packed boxes ranging from 30 to 90. Green indicates better performance, while red indicates worse performance.

The first observation is that the **Constant** cost function performs poorly overall, with look-ahead providing no benefit. This cost function serves as a baseline for comparison. The **BottomLeft** cost function performs slightly better, but the improvement is negligible. However, it achieves some of the best variance with a look-ahead of 50. The **BottomLeftWidthHeight** cost function performs so poorly that it is the only one to degrade with increased look-ahead, even being outperformed by the **Constant** cost function in online packing. This makes it an exceptionally poor choice and should be avoided.

The **Krass** cost function consistently outperforms all others. While it is mediocre in online packing, only marginally better than the **Constant**, it excels with increased look-ahead, achieving an impressive 81% packing density.

In conclusion, while good cost functions can be crafted through trial and error, the process is not as intuitive as it might seem. The poor performance of the **BottomLeftWidthHeight** cost function

highlights that even seemingly good ideas can fail in practice.

Moving on to the Random tests in **Figure** 5.1, the mean and variance are computed from 150 packing attempts using randomly generated small boxes, with an average of 300 to 900 boxes packed.

The first notable observation is that the **Constant** cost function improves with increased look-ahead, unlike when using the list. Secondly, the **BottomLeft** cost function outperforms all others significantly, both in mean and variance. Conversely, the **BottomLeftWidthHeight** cost function shows no improvement. The **Krass** cost function has the worst mean performance in online packing, with a packing density of only 28%. With more look-ahead, it barely surpasses the **Constant** cost function.

Some initial general conclusions are that the cost function must be tailored to each specific use case, and crafting an effective cost function is neither easy nor straightforward.

	Look Ahead	Mean	Variance	Std
Constant	1	0.6568904	0.00073873	0.02717967
	5	0.6500648	0.0007379	0.02716432
	25	0.6505529	0.00067651	0.0260099
	50	0.6517376	0.00067622	0.02600425
BottomLeft	1	0.65736485	0.00076085	0.02758352
	5	0.66414654	0.00079018	0.02811018
	25	0.67170167	0.00070607	0.02657206
	50	0.68595845	0.00049625	0.02227676
BottomLeftWidthHeight	1	0.65027505	0.00072284	0.02688575
	5	0.6438621	0.00090692	0.03011517
	25	0.6188873	0.00075083	0.02740129
	50	0.62390226	0.00066809	0.02584739
Krass	1	0.6715755	0.0012842	0.03583574
	5	0.70197666	0.00143279	0.03785224
	25	0.7792072	0.00053294	0.0230854
	50	0.8166915	0.00029565	0.01719457
Annealed	1	0.6650972	0.00080724	0.02841194
	5	0.676602	0.00083625	0.02891792
	25	0.7037397	0.00088336	0.02972137
	50	0.7241564	0.00108606	0.03295542

Figure 17: Packing performance of random boxes. Packs: 150, Domain size: 100^3

	Look Ahead	Mean	Variance	Std
Constant	1	0.41695824	0.00635418	0.07971314
	5	0.6373203	0.00460899	0.06788952
	25	0.74225193	0.00057621	0.02400431
	50	0.7650497	0.00052589	0.02293219
BottomLeft	1	0.57275635	0.00169368	0.04115432
	5	0.74622005	0.00018738	0.01368874
	25	0.82186097	3.92E-05	0.00626339
	50	0.8343363	1.01E-05	0.00318145
BottomLeftWidthHeight	1	0.4053268	0.00298626	0.0546467
	5	0.5347498	0.00067242	0.025931
	25	0.6135845	0.00037695	0.0194151
	50	0.6675992	0.00065353	0.02556415
Krass	1	0.28522357	0.0052881	0.07271929
	5	0.64729166	0.00117795	0.03432127
	25	0.75674504	0.00069354	0.02633515
	50	0.7739033	0.00067747	0.02602828
Annealed	1	0.57249945	0.00233574	0.04832954
	5	0.74744517	0.00024511	0.01565589
	25	0.8057124	9.43E-05	0.00971221
	50	0.8128071	7.94E-05	0.00890942

Figure 18: Packing performance of postpac list boxes. Packs: 150, Domain size: 100^3

5.2 Unsupervised learning

Unsupervised learning is when algorithm tries to learn underlying patterns by itself and without human intervention. When using a heuristic like a weighted costfunction the question comes up immediately if its possible to learn the weights on its own. This question is even more interesting in the light how difficult it is to design a good costfunction by hand.

Lets consider following weighted costfunction in the form of a simple linear polynomial:

Definition 5.1.

$$\mathcal{C}_{Annealed}(\mathcal{T}_{i,j}) = \omega_0 * \mathbf{bin} + \omega_1 * \mathbf{x} + \omega_2 * \mathbf{y} + \omega_3 * \mathbf{h} + \omega_4 * \hat{\mathbf{l}} + \omega_5 * \hat{\mathbf{h}} + \omega_6 * \hat{\mathbf{m}} + \omega_7 * \mathbf{eX} + \omega_8 * \mathbf{eY}$$

w.r.t $\quad \overrightarrow{\omega} \in \mathbb{R}^n$

Incorporating this polynomial with the reformulated packing problem in **Definition** 3.7 we can reformulate it again so it maximises the mean and minimizes the variance as stated in **Definition** 5.2. The first part of computing the variance and the mean are straight forward: Create n packs with the same weights ω and use the samples for computing.

Definition 5.2. Reformulated Packing problem for Unsupervised Learning

Let D the domain, (x, y) the coordinates on the domain, $C : \mathcal{T} \to \mathbb{R}$ the costfunction and $\mathcal{T}$ the result tuple and $\hat{\mu}$ as the expected value or mean and $\hat{\sigma^2}$ as the variance over m packs. Then the reformulated packing problem for unsupervised learning is defined as

$$\hat{\mu} = \max E\left[\min_{\{\mathcal{T}\}_{k=1}^{n}} \sum_{k=1}^{n} \sum_{(x,y)\in D} C\left(\mathcal{T}_k(x, y)\right)\right]^m \quad \hat{\sigma^2} = \min \text{Var}\left(\min_{\{\mathcal{T}\}_{k=1}^{n}} \sum_{k=1}^{n} \sum_{(x,y)\in D} C\left(\mathcal{T}_k(x, y)\right)\right)^m$$

What we try to learn is a distribution that yields the best overall result, as in highest mean with lowest variance. However, the distribution may have very high frequency and be not even $\mathcal{C}^0$ smooth, e.g. the first derivative does not exist. If this is the case any algorithm that converges towards a maximum, think of gradient ascent, will not work. Thus, the first step is to sample the distribution and analyse the samples for at least local $\mathcal{C}^0$ smoothness. This can be done with annealing as shown in **Algorithm** 2. It is a very simple algorithm and uses random jitters, or Brownian noise, around a point of the distribution. When it finds a point with an increase in mean and decrease of variance relative to the previous position it adopt this as the new solution. If it gets stuck, as in not finding a better solution it tries to punch through the potential with a big step in a random direction. This may work or get it even more stuck. It terminates after a predefined amount of steps.

Algorithm 2 Annealing

```
Input: Weights w, Steps n
Start at known good position (e.g. 0)
while steps < n do
  if stuck then
    Large step w in random direction
  else
    Small step subset of w in random direction
  end if
end while
```

If we visualise an iso-surface of the x and y weight of annealed samples as in **Figure** 19 we can see that there seems to at least exist local $\mathcal{C}^0$ smoothness, as in, it is continuous. And not just in the mean but as in the variance also. Note: this is by far not an exhaustive analysis or proof for smoothness. The implication of smoothness is that a global maximum exists and even could possibly be found, which would have great practical implications, because once learned an close to ideal costfunction could be deployed at almost zero cost.

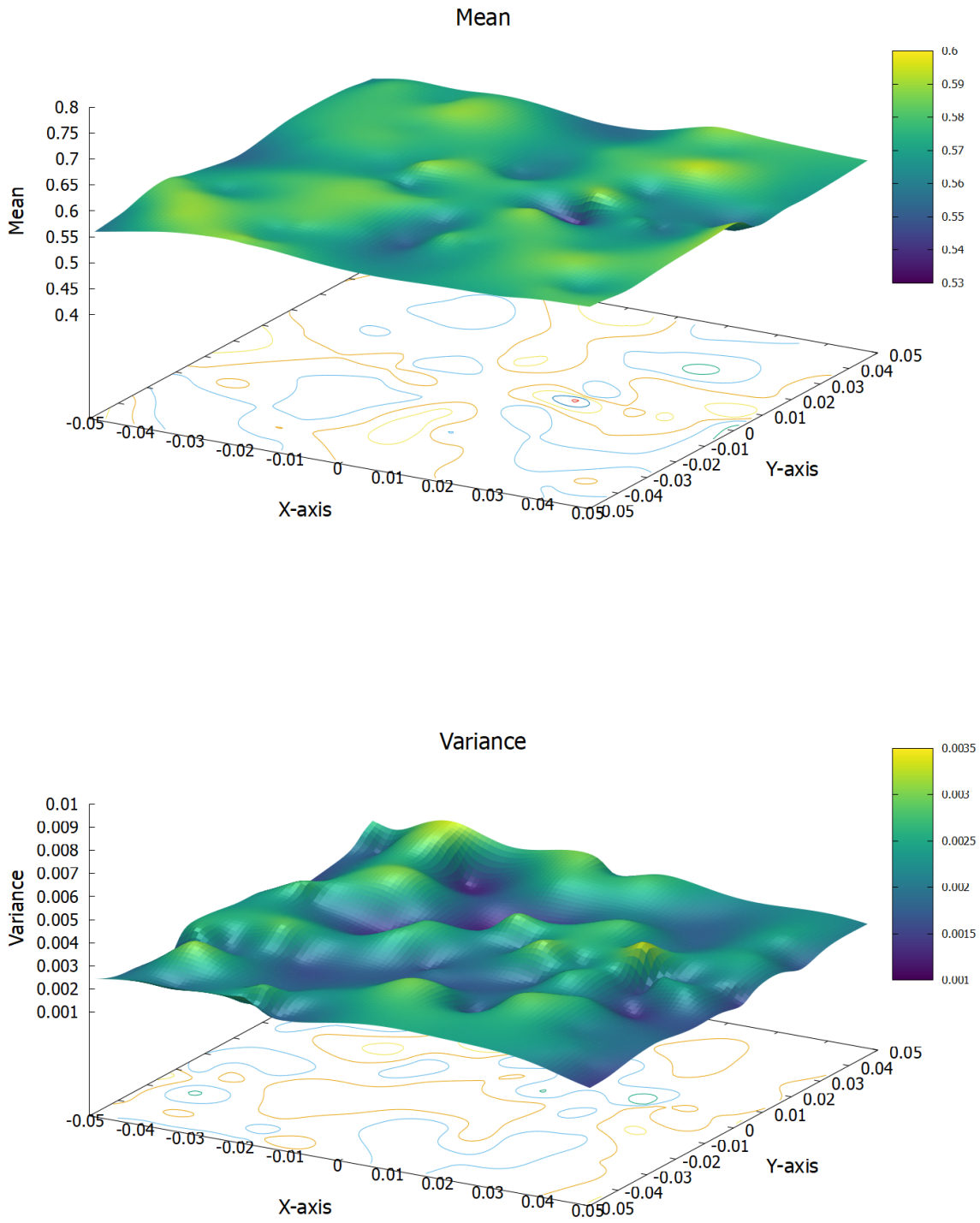


Figure 19: Iso surface of mean and variance of spatial weights in radius < 0.05 around [0, 0]. 150 samples.

With the assumption that the distribution is indeed $\mathcal{C}^0$ smooth we can deploy an adapted annealing algorithm to find potential weights. The idea is to start from a know good starting point, the constant case, e.g. all weights are 0, and randomly sample around the last know good point to find a good weight configuration that would lead to increase in quality of the packing. When such a sample is found the algorithm steps in this direction ascending along the potential. For such a ascending algorithm the first derivative needs to be computed and this is the reason why there is a minimum requirement for smoothness. In **Figure** 20 four steps of ascending random walk are depicted. The large orange points are the new positions assumed after every step. The smaller, blue points are random samples taken around the current best position (orange). The purple is the sample with the highest increase in mean and are used as direction for the next step to a known good position.

Algorithm 3 Ascending random walk

Input: Weights w, Samples m, Steps n
start at known *good* position (e.g. all 0)
while steps $< n$ **do**
  while samples $< m$ **do**
    Sample small radius around current best position
  end while
  Step large step in direction of greatest ∇ Variance and ∇ Mean
end while

Even after only a limited time (few hours) running this algorithm interesting results can be achieved. In **Figure** 5.1 the *Annealed*, which are the learned weights, applied to a domain of 100^3 and box list with a look ahead of 1, 5, 25 and 50 respectively achieves better results than all others except the **Krass**. However, for random boxes in **Figure** 5.1 it performs much better than **Krass** but not as well as **BottomLeft**.

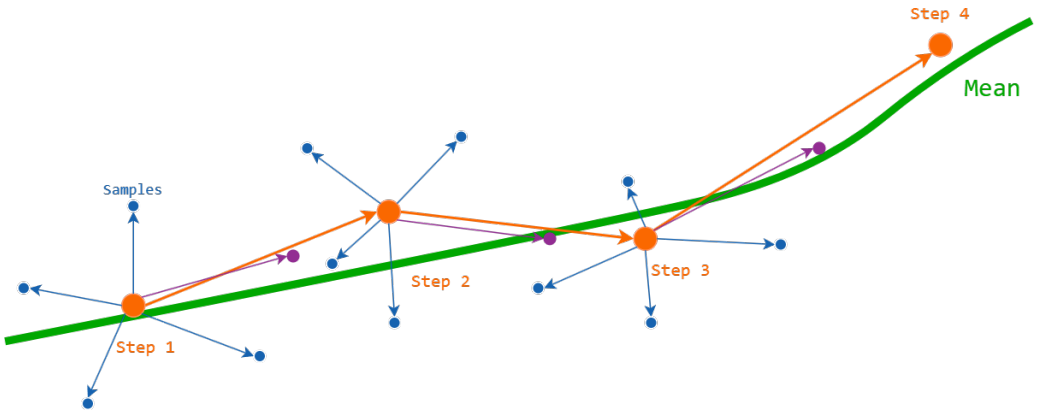


Figure 20: Four steps of an ascending random walk following the potential of the mean of a distribution. Orange: new positions, blue: random samples, purple: best sample

6 Concluding remarks

In this thesis, I explained what an NP problem is, how these kinds of problems relate to box packing, and how to solve them. I introduced the concept of the tri-convolution and demonstrated how to solve it in $O(\sqrt{n})$ with the MQT library. I detailed how the TurboPacker library solves the box packing problem using a cost function heuristic. Finally, I presented experiments with analytical cost functions and provided a preliminary outline on learning the cost function using annealing. From this work, I have identified four critical areas that still need further research and development.

First, while the MedianQuadTree and TurboPacker libraries provide stable and high-performance utility, they lack a CUDA implementation or distributed support. High computational power is essential for larger domain sizes or packing multiple bins. Even with a powerful current CPU, there is a practical limit around a domain size of 100^2 that can be handled without very long waiting times. Although significant improvements have been made in this thesis, more work is still needed.

Second, the application and library are still missing critical functionality. One such functionality is environmental information. For example, it is not possible to query from the costfunction what kind of box is below the current test box, making it impossible to consider weight and box type constraints. Some boxes should not be placed on top of others, etc. Additionally, the app requires further development to fix minor issues and enhance user-friendliness. Features like freely moving bins or displaying information about a box when hovering over it come to mind.

Third, the manual construction of cost functions needs more thorough study. Despite significant time spent on crafting a handcrafted cost function, its performance is questionable at best. With a better understanding of the interplay among the various elements of the Result Tuple, more effective cost functions can be developed. Additionally, stateful cost functions could be investigated and might yield interesting results.

Fourth, more work is needed on unsupervised learning. Further research is necessary to understand the distribution, the continuity and smoothness of the cost functions. Additionally, more advanced annealing algorithms need to be developed and tested. This area is particularly promising due to its practical implications.

In closing, I would like to thank my supervisors, Simon and Moritz, for their tireless support, help, and gentle guidance throughout this thesis.

References

[1] Jaza M. Abdullah et al. “Multi objective Fitness Dependent Optimizer Algorithm”. In: (2023). arXiv: 2302.05519 [cs.NE].

[2] Adriana Alvim et al. “A Hybrid Improvement Heuristic for the One-Dimensional Bin Packing Problem”. In: *Journal of Heuristics* 10 (Mar. 2004), pp. 205–229. DOI: 10.1023/B:HEUR.0000026267.44673.ed.

[3] Richard Bellman. *Dynamic Programming*. Dover Books on Computer Science. Mineola, New York: Dover Publications, 2003. ISBN: 9780486428093.

[4] Mark Bruls, Kees Huizing, and Jarke J. van Wijk. “Squarified Treemaps”. In: *Proceedings of the joint Eurographics and IEEE TCVG Symposium on Visualization*. 2000, pp. 33–42.

[5] Qiaodong Cui et al. “Dense, Interlocking-Free and Scalable Spectral Packing of Generic 3D Objects”. In: *ACM Trans. Graph.* 42.4 (July 2023). ISSN: 0730-0301. DOI: 10.1145/3592126. URL: https://doi.org/10.1145/3592126.

[6] Behnam Esfahbod. *P np np-complete np-hard*. Accessed: 2024-06-07. 2007. URL: https://commons.wikimedia.org/wiki/File:P_np_np-complete_np-hard.svg.

[7] Paulina Grzegorek, Janusz Januszewski, and Łukasz Zielonka. “Efficient 1-Space Bounded Hypercube Packing Algorithm”. In: *Algorithmica* 82 (Nov. 2020). DOI: 10.1007/s00453-020-00723-5.

[8] Damian Heer. *MedianQuadTree*. 2024. URL: https://github.com/Heerdam/MedianQuadtree

[9] Damian Heer. *TurboPacker*. 2024. URL: https://github.com/Heerdam/TurboPacker.

[10] Olyvia Kundu, Samrat Dutta, and Swagat Kumar. “Deep-Pack: A Vision-Based 2D Online Bin Packing Algorithm with Deep Reinforcement Learning”. In: (2019), pp. 1–7. DOI: 10.1109/RO-MAN46459.2019.8956393.

[11] Richard E. Ladner. “On the Structure of Polynomial Time Reducibility”. In: *J. ACM* 22.1 (Jan. 1975), pp. 155–171. ISSN: 0004-5411. DOI: 10.1145/321864.321877. URL: https://doi.org/10.1145/321864.321877.

[12] D. O. Lazarev and Nikolay N. Kuzyurin. “On Online Algorithms for Bin, Strip, and Box Packing, and Their Worst-Case and Average-Case Analysis”. In: *Programming and Computer Software* 45 (2019), pp. 448–457. URL: https://api.semanticscholar.org/CorpusID:210195262.

[13] Kok-Hua Loh, Bruce Golden, and Edward Wasil. “Solving the one-dimensional bin packing problem with a weight annealing heuristic”. In: *Computers Operations Research* 35.7 (2008). Part Special Issue: Includes selected papers presented at the ECCO’04 European Conference on combinatorial Optimization, pp. 2283–2291. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2006.10.021. URL: https://www.sciencedirect.com/science/article/pii/S0305054806002826.

[14] Jia-Hui Pan et al. “SDF-Pack: Towards Compact Bin Packing with Signed-Distance-Field Minimization”. In: (2023). arXiv: 2307.07356 [cs.RO].

[15] Songyou Peng et al. “Convolutional Occupancy Networks”. In: (2020). arXiv: 2003.04618 [cs.CV].

[16] Heng Xiong et al. “Towards reliable robot packing system based on deep reinforcement learning”. In: *Advanced Engineering Informatics* 57 (2023), p. 102028. ISSN: 1474-0346. DOI: https://doi.org/10.1016/j.aei.2023.102028. URL: https://www.sciencedirect.com/science/article/pii/S1474034623001568.

[17] Hang Zhao et al. “Learning Physically Realizable Skills for Online Packing of General 3D Shapes”. In: (2023). arXiv: 2212.02094 [cs.LG].

7 Appendix

Proof. Given:

- A 1D heightmap $\{a, b, c\}$ with $a, b, c > 0$
- A 1D kernel $[1, 1, 1]$
- The convolution at position b: $a + b + c$
- The logarithmic convolution at position b: $\log(a) + \log(b) + \log(c)$

Condition 1:

$$a + b + c = 3b \tag{1}$$

$$a + c = 3b - b \tag{2}$$

$$a + c = 2b \tag{3}$$

Condition 2:

$$\log(a) + \log(b) + \log(c) = 3\log(b) \tag{4}$$

$$\log(abc) = \log(b^3) \tag{5}$$

$$abc = b^3 \tag{6}$$

$$ac = b^2 \tag{7}$$

Solving the System of Equations

$$a + c = 2b \quad ac = b^2 \tag{8}$$

$$c = 2b - a \tag{9}$$

$$a(2b - a) = b^2 \tag{10}$$

$$2ab - a^2 = b^2 \tag{11}$$

$$a^2 - 2ab + b^2 = 0 \tag{12}$$

$$a^2 - 2ab + b^2 = (a - b)^2 = 0 \tag{13}$$

$$\Rightarrow a - b = 0 \tag{14}$$

$$\Rightarrow a = b \tag{15}$$

$$\Rightarrow a + c = 2b \tag{16}$$

$$\Rightarrow b + c = 2b \tag{17}$$

$$\Rightarrow c = b \tag{18}$$

$$\Rightarrow a = b \; and \; c = b \tag{19}$$

Conclusion

The only solution that satisfies both conditions is:

$$a = b = c \tag{20}$$

□

Definition 7.1. MedianQuadTree minimal example

```cpp
#include <MQT2.hpp>
using namespace MQT2;

int main() {
    std::vector<uint32_t> map; //the size n needs to be of size 2^m and n
       %BUCKET_SIZE == 0
    //fill map with values
    //...
    MedianQuadTree<uint32_t, uint16_t, 15> tree(map, 40);

    //overlap
    const auto[l1, m1, h1] = tree.check_overlap(Vec2<BT>{0, 0}, Vec2<BT
       >{40, 40}, 1);

    //recompute
    //this bool map containes a bool for every bucket. if true it will
       recompute this bucket.
    std::vector<bool> mm;
    //do something
    tree.recompute(mm);
}
```

Definition 7.2. Example custom costfunction

```
#include <TP.hpp>
using namespace TP;

template<class T>
struct CF_Example {
    static float eval(const Detail::Result<T>& _r) {
        const auto nrml = _r.normalize();
        return nrml.n0 * nrml.n1;
    }
}

int main() {
    //...
    Config<float, CF_Example> config;
    //...
}
```

Definition 7.3. Minimal example for TurboPacker integration

```cpp
#include <TP.hpp>
using namespace TP;
int main() {
    BinInfo<float> bin;
    bin.Bounds = {100., 100.};
    bin.Height = 100.;

    Config<float, CostFunction::CF_Krass> config;
    config.Bins.push_back(bin);

    auto promise = solve(config);
    promise.wait();
    assert(promise.done());
    const auto result = promise.data();
    //...
    return 0;
}
```