

Full length article

An integrated simulation framework enabling flexible robotic palletizing

Martina Salami, Pietro Bilancia^{ID*}, Marcello Pellicciari

Department of Sciences and Methods for Engineering, University of Modena and Reggio Emilia, Reggio Emilia, Italy

ARTICLE INFO

Keywords:

Engineering method
Bin packing
Robotic palletizing
Robot simulation framework
RoboDK
Instruction streaming

ABSTRACT

Robotic palletizing requires the rapid generation of optimal packing plans and the corresponding robot programs within the tight time constraints imposed by mass customization and factory intralogistics. In this context, while the bin packing problem has been extensively studied in the literature, practical engineering tools that support flexible batch palletizing, from optimal packing definition to validated robot execution, remain limited. In this regard, the present paper proposes an integrated simulation framework that guides engineers in defining an optimal robotic palletizing process through a structured sequence of steps. Starting from the 3D model of an existing palletizing station and its input product mix, the framework, implemented as a set of interconnected Python modules within the RoboDK simulation platform, supports automated generation and assessment of actionable packing layouts, simulation-based verification of the robotic process, and generation of executable robot code that can be automatically transferred to the robotic palletizing cell within a Digital Twin oriented approach. An efficient heuristic method is introduced to solve 3D packing instances for box-type items on standard Europallets. The problem is initially formulated as a sequence of 2D packing problems solved through a Guillotine plus Best Fit strategy, iterating across layers to maximize pallet filling while improving load balancing. The framework is validated across multiple scenarios on an industrially representative case study involving a KUKA palletizing robot, a conveyor feeding system, and two auxiliary buffer pallets. Finally, an instruction streaming module enabling online execution of the validated code on the KUKA controller is presented. Overall, the results confirm the effectiveness of the proposed framework in accelerating process planning, virtual validation and programming activities while maintaining packing quality in realistic industrial scenarios.

1. Introduction

In modern manufacturing environments, driven by increasing product customization and variable demand, industrial robots are expected to deliver high productivity while enabling flexible manufacturing with frequent process changes [1]. In this context, competitiveness increasingly depends on the ability to dynamically program robotic cells to realize flexible, optimized processes that adapt to evolving product mixes, throughput targets, and layout constraints, while maintaining stringent requirements on speed, accuracy, and energy efficiency [2–4]. This, in turn, creates a recurring need to automatically generate optimal processes and the corresponding validated robot programs, while avoiding downtime. To meet these demands, engineering practice is progressively shifting toward Digital Twin oriented workflows [5], where high fidelity virtual prototypes enable rapid evaluation of alternative process configurations, feasibility checks, and offline verification of cycle time, reachability, and collision free paths prior to deploying updated robot code on the shop floor. Achieving this level of operational flexibility requires novel engineering methods and tools that

tightly couple process planning, virtual validation, and code deployment within a repeatable, application driven workflow, rather than treating these steps as separate and largely manual activities.

This need is particularly evident in palletizing and end-of-line logistics stations [6], where newly customized product batches, often in small lots and with heterogeneous sizes, require frequent process adaptation and reprogramming [5]. From an industrial perspective, palletizing performance is largely determined by both the planned pallet layout [7] (i.e., how products are arranged on the pallet) and the execution strategy [8]. The pallet pattern directly affects pallet fill rate and space utilization, load stability [9] and product protection, and downstream handling constraints (e.g., wrapping and transport compliance), while the associated robotic pick-and-place sequence determines cycle time, robot travel distance and motion smoothness, collision risk, and overall process efficiency. Effective palletization offers clear benefits, including increased throughput and reduced material waste and energy consumption, contributing to more sustainable and cost-effective logistics.

* Corresponding author.

E-mail address: pietro.bilancia@unimore.it (P. Bilancia).<https://doi.org/10.1016/j.rcim.2026.103347>

Received 1 June 2026; Accepted 1 June 2026

Available online 11 June 2026

0736-5845/© 2026 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

From a computational standpoint, generating pallet layouts for heterogeneous product mixes while maintaining proper load balance is formulated as a multidimensional Bin Packing Problem (BPP) [10], a combinatorial optimization problem that is NP hard [11] and characterized by multiple interdependent constraints. Selecting an efficient and robust algorithm to solve industrial BPP instances is non-trivial. Although numerous solution strategies exist in the literature [12,13], most are evaluated primarily within academic benchmarks, with limited industrial validation and few actionable guidelines for algorithm selection. Among these, exact methods can yield optimal solutions for restricted instances, but they typically fail to converge within practical time limits once realistic industrial requirements are considered, such as large and heterogeneous item sets, mixed-layer stacking, and additional objectives and constraints related to load balance. The latter is typically introduced to ensure pallet stability, reduce the risk of tilting during handling and transport, and improve operator safety.

Industrial palletizing requires algorithms that are both highly effective and computationally efficient, capable of maintaining robust performance across a wide range of operational scenarios. Over the last decades, many research works have addressed the BPP by proposing a variety of valuable heuristics, metaheuristics, and learning-based approaches, often evaluated on standardized benchmarks. Recent studies have proposed online mixed palletizing solutions based on 3D vision [14] and deep reinforcement learning [15], demonstrating good convergence properties and high solution quality. Nevertheless, adopting such approaches generally entails additional costs in terms of hardware, integration effort, and model training time, which can limit industrial applicability. Despite all these methodological contributions, their translation into ready-to-use engineering tools remains limited [16]. In practice, many companies still rely on in-house packing rules developed through trial-and-error and validated on a small set of representative cases, which can lead to suboptimal pallet utilization and overall logistics performance. On the other hand, existing commercial robotic offline programming platforms typically support only specific stages of the workflow, primarily the robot motion simulation and the subsequent generation of robot programs that are collision free and compliant with kinematic and workspace constraints [17–19]. They rarely provide dedicated tools for BPP specification and for the broader industrial palletizing process planning and validation. For example, ABB RobotStudio [20] or RoboDK [21] offer interfaces that allow users to manually arrange boxes and define the final pallet layout. However, these solutions do not incorporate state-of-the-art algorithms for solving complex BPP in their libraries.

The transition from existing solution methods to their effective industrial deployment remains non-trivial, as structured guidance on how established packing strategies should be selected, combined, parameterized, and adapted to application-specific industrial requirements and constraints is missing. To address this gap, this work introduces a reproducible engineering method, together with an integrated simulation framework, for translating established algorithmic knowledge into an effective solution for industrial robotic palletizing. The framework enables rapid process planning and virtual validation for industrial robotic palletizing systems, by coupling a robust 3D packing algorithm with simulation-based verification and streamlined deployment of executable robot programs to a physical robot. The proposed solution is implemented in the RoboDK platform and exploits its native Python interfaces to integrate dedicated modules that automatically solve the BPP, generate the feed sequencing, and validate the resulting robot program within a 3D parametric environment. This architecture supports flexible process reconfiguration, reduces programming effort, and facilitates systematic performance optimization of existing palletizing cells, while also supporting the definition and evaluation of novel cell designs. The core contributions of this work are as follows:

1. **Overview and comparative benchmark:** a structured review and quantitative comparison of representative BPP solution strategies, with emphasis on their suitability for industrial palletizing instances.

2. **Computationally efficient packing algorithm:** a modular and computationally efficient heuristic workflow for multi-layer three dimensional BPP that seeks efficient space utilization and load balancing. The methodology systematizes and combines literature known packing elements into a coherent procedure that can be selected, parameterized, and adapted to realistic industrial palletizing constraints. The methodology is validated on box-type items packed on Europallets.
3. **Integrated framework:** a simulation framework that integrates the packing algorithm into a Python-based workflow to automatically generate pallet layouts, virtually validate the corresponding palletizing tasks in RoboDK, compute performance indicators to support design decisions, and generate and transfer executable robot programs to the physical system via online instruction streaming. Validation is conducted on an industrial case study comprising a KUKA palletizing robot and a KR C4 controller.
4. **Open source release:** public access to the implemented framework to support reproducibility and future extensions.

The remainder of the paper is organized as follows. Section 2 formalizes the multidimensional BPP and summarizes the state-of-the-art, with emphasis on two dimensional and three dimensional variants and practical solution strategies. Section 3 presents the proposed framework and details the packing algorithm. Section 4 evaluates the packing method and validates the overall framework on an industrial case study. Finally, Section 5 provides the concluding remarks.

2. Bin packing for robotic palletizing

2.1. Problem formulation

The BPP aims to allocate a set of items of different sizes into a finite number of bins (containers), minimizing the total number of bins used. Each bin has a maximum allowable capacity, and the total size of the items assigned to it must not exceed this limit. The problem seeks an arrangement that maximizes space utilization while satisfying non-overlapping and boundary constraints. Depending on the dimensional representation of items and bins, the problem can be formulated in one, two, or three dimensions, as shown in Fig. 1. In the following, each of these cases is introduced and described.

In the one dimensional case (1D-BPP), each item i is characterized by a size w_i , while each bin has a fixed capacity that determines its length W . The goal is to assign every item to a bin such that the sum of item sizes in any bin does not exceed its capacity, i.e.:

$$\sum_{i \in B_j} w_i \leq W, \quad \forall j \quad (1)$$

where B_j denotes the j th bin. In addition, for any pair of items i and k assigned to the same bin, the following non-overlapping condition must hold:

$$(x_i + w_i \leq x_k) \vee (x_k + w_k \leq x_i) \quad (2)$$

where the coordinates x_i and x_k denote the positions of the items along the bin axis, defined as the distance of their left endpoint from the bin origin. The 1D-BPP typically models situations such as allocating files in memory blocks or packing objects along a single length dimension. However, this formulation has limited practical relevance for palletizing applications and is therefore not considered further in this work (the interested reader may refer to [22] for a comprehensive discussion).

Extending to the two dimensional case (2D-BPP), each item i is characterized by its width w_i and height h_i , while the bin capacity is represented by the two dimensions (W, H) , as visible in Fig. 1. Each item must be placed within the bin boundaries:

$$0 \leq x_i + w_i \leq W \quad 0 \leq y_i + h_i \leq H \quad (3)$$

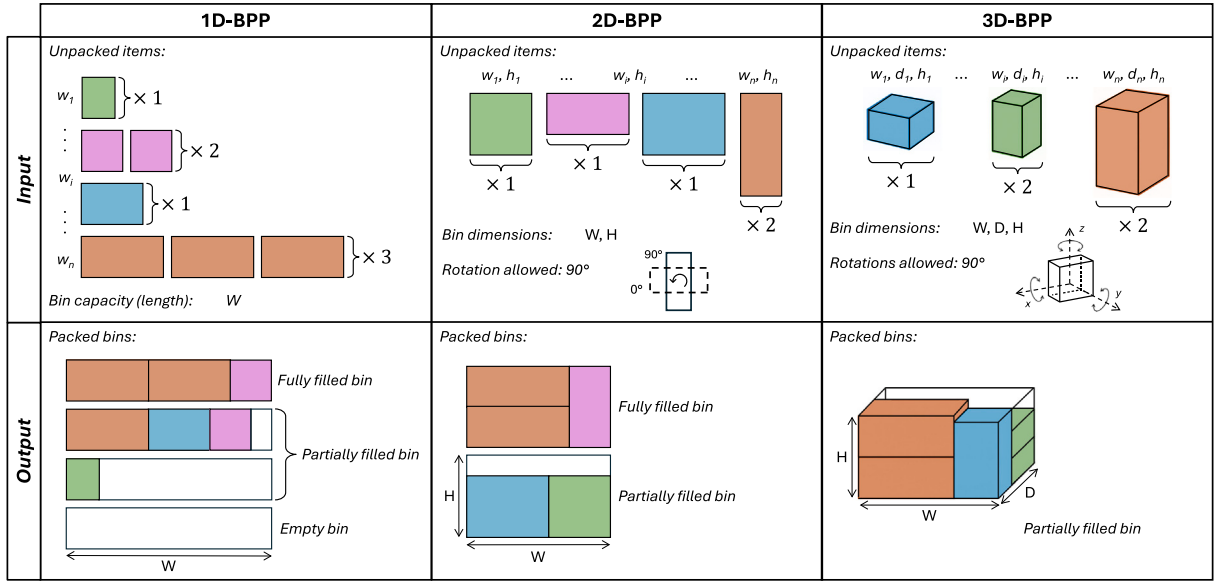


Fig. 1. Visual representation of the 1D, 2D and 3D-BPP: input items, bin geometric characteristics and allowed rotations.

where x_i and y_i denote the coordinates of the lower-left corner of the item i with respect to the lower-left corner of the bin, set as the origin $(0, 0)$. As in the previous case, the following non-overlapping condition must hold for every pair of items i and k packed in the same bin:

$$(x_i + w_i \leq x_k) \vee (x_k + w_k \leq x_i) \vee (y_i + h_i \leq y_k) \vee (y_k + h_k \leq y_i) \quad (4)$$

A relevant extension of the 2D-BPP allows items to be rotated by 90° to improve space utilization, though at the cost of increased combinatorial complexity [23]. In this case, a binary variable $r_i \in \{0, 1\}$ is introduced for each item i , where $r_i = 1$ indicates that item i is rotated. The effective dimensions of each item therefore become:

$$w'_i = (1 - r_i)w_i + r_i h_i \quad h'_i = (1 - r_i)h_i + r_i w_i \quad (5)$$

In this case, the boundary constraints (Eq. (3)) and the non-overlapping constraint (Eq. (4)) must be applied to (w'_i, h'_i) .

Finally, by considering three dimensional items and bins (Fig. 1, third scenario), the problem is further extended to the 3D case (3D-BPP). Each item i is characterized by dimensions (w_i, d_i, h_i) , while the bin capacity is now represented by the three dimensions (W, D, H) . The boundary constraints become:

$$0 \leq x_i + w_i \leq W \quad 0 \leq y_i + d_i \leq D \quad 0 \leq z_i + h_i \leq H \quad (6)$$

where x_i , y_i , and z_i denote the coordinates of the lower-left-rear corner of item i with respect to the lower-left-rear corner of the bin, set as the origin $(0, 0, 0)$. For brevity, the explicit non-overlapping constraints are omitted here, since the previous 2D condition in Eq. (4) can be easily extended to 3D by enforcing separation along all coordinate axes.

In the 3D-BPP, which is more representative of practical industrial applications, each item is additionally characterized by a weight. Accordingly, in addition to the space-efficiency objective shared with the 1D-BPP and 2D-BPP, the evaluation process typically includes a second, complementary objective related to load balancing [24]. While space efficiency seeks to minimize the total number of bins required, load balancing aims to improve the stability of the packed configuration by minimizing the distance between the Center of Gravity (CoG) of the bin ($\mathbf{g}$) and its vertical geometric axis (or centroid on XY plane, $\mathbf{c}$), thereby enabling safer handling during transport and stacking:

$$\min d_{xy} = \|\mathbf{c} - \mathbf{g}\| \quad (7)$$

In 3D scenarios, item rotation can also be considered to improve packing flexibility, by allowing reorientation about one or more axes (x , y , or z). However, in practical applications, allowable rotations are often restricted to rotations about the Z -axis, to preserve product integrity and stability, ensure correct item orientation, and satisfy handling constraints. As in the 2D case, introducing rotation decisions increases the combinatorial complexity of the problem, but yields a more realistic model for pallet loading and container optimization, where both space efficiency and mechanical stability are essential [7].

2.2. Solution approaches

The BPP is a combinatorial optimization problem classified as NP-hard [11], since the number of feasible combinations grows exponentially with the number of items. Consequently, exact methods quickly become computationally impractical as the instance size increases, and practical solution strategies rely on approximations, heuristics, or problem-specific decompositions, with the most suitable choice depending on the considered BPP variant. In the context of industrial palletizing, the most relevant case is the 3D-BPP. As said, in addition to the additional spatial dimension, this variant introduces constraints on the overall pallet stability and load balancing (Eq. (7)), further increasing the problem complexity. To deal with this, recent research works have also explored data-driven and learning-based approaches [25], e.g. adopting neural networks to generate or predict efficient packing strategies [26]. By learning from previously solved instances, these models can generalize to new configurations and provide near optimal layouts. However, they typically require considerable training effort and suitable datasets, and their deployment is not always compatible with industrial response time constraints. Some studies have also explored quantum computing approaches [27], which aim to exploit quantum parallelism to speed up combinatorial optimization tasks, although their practical applicability to real scale palletizing remains limited.

For this reason, an alternative approach is to decompose the 3D problem into a set of 2D sub-problems to simplify computation [28]. This decomposition strategy is commonly used in the literature [29,30], as it enables a reduction in computational effort while maintaining practical solution times for complex packing problems. In practice, the original 3D packing space is partitioned into horizontal layers, and each layer is filled by solving a 2D-BPP instance. This approach is

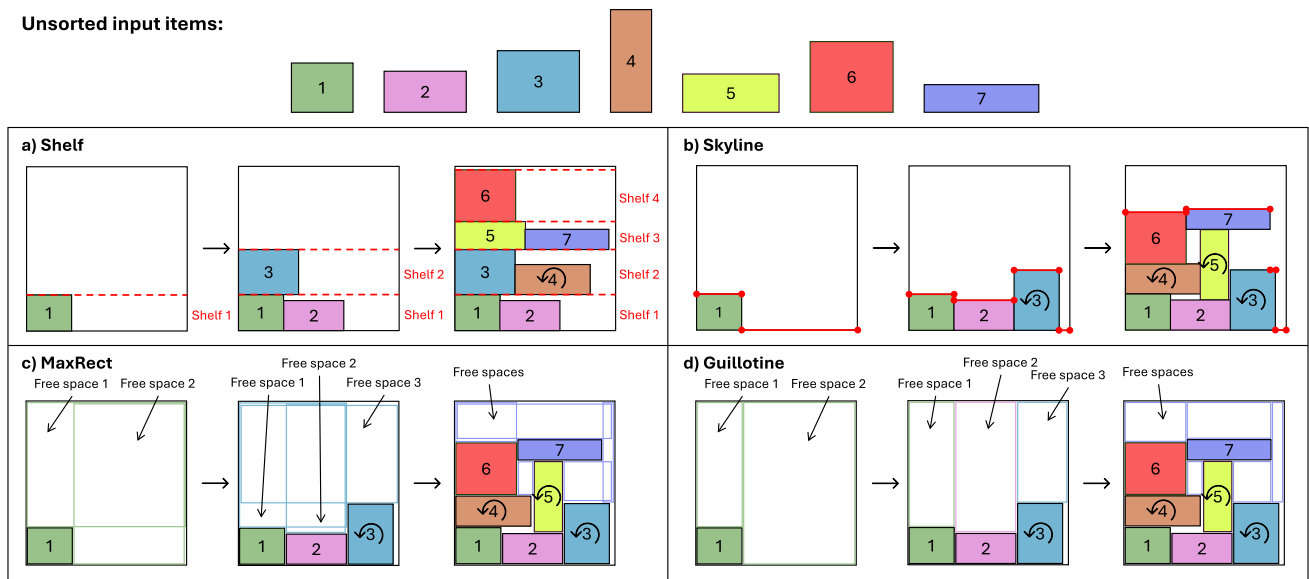


Fig. 2. Comparison of empty space detection algorithms for 2D-BPP: example of space detection results for the same input sequence using (a) Shelf, (b) Skyline, (c) MaxRect, and (d) Guillotine approaches.

consistent with common industrial practice, where pallets are built as stacks of layers composed of items with uniform height, often separated by interleaving elements such as slip sheets. Under this decomposition, the extensive set of well-established algorithms developed for the 2D-BPP can be exploited to solve the layer sub-problems. A brief overview of these methods is provided in the following, while a more extensive survey can be found in Ref. [31]. The overall 3D solution is then refined in a second stage, where additional constraints are enforced to integrate the layer wise layouts within a coherent 3D framework, possibly through iterative evaluation until a satisfactory configuration is obtained.

The 2D-BPP can, in principle, be addressed using exact methods, which can attain the optimal solution while satisfying all constraints, typically through an exhaustive exploration of the solution space. However, as the number of items, degrees of freedom (for example, item rotations), and additional constraints increases, the computational effort grows exponentially. Therefore, in industrial contexts, where thousands of items must be handled and fast decision-making is required, exact methods are rarely applied directly. Common exact approaches include Branch and Bound [32], Mixed-Integer Linear Programming [33,34], and Constraint Programming [35]. Comparative studies of these exact methods can be found in [36].

Since exact methods are impractical, the problem is typically addressed using heuristic algorithms, which aim to find good-quality solutions within reasonable computational times [37]. In practice, heuristics are frequently used as building blocks within larger frameworks, e.g. to generate an initial solution that can later be refined by meta-heuristics or other improvement techniques. Their modular structure also allows different strategies to be combined depending on the problem features. A heuristic method can be described in terms of three main modules (item ordering, empty-space detection, and placement), which are commonly applied in sequence within an iterative procedure, as items are placed one by one on the 2D layer. For each module, various algorithms have been proposed, as reported below:

1. **Item ordering:** this module is executed once to determine how the input item list is organized, i.e. the sequence in which the input items are considered by the iterative placement procedure. Common criteria include sorting items by area, height, width, longest side, perimeter or, when load balancing is considered, by mass. The adopted criterion can significantly affect the final solution, as discussed in Ref. [38].

2. **Placement:** this module is run for every item of the input list obtained at the previous step. At each iteration, it determines where to locate item i by selecting one admissible placement among the currently available spaces. The most common approaches include:

- First Fit, which places the new item in the first available space encountered without searching for a better alternative among the remaining spaces, ensuring simplicity and speed [39].
- Best Fit, which evaluates all possible spaces according to a selected criterion, e.g. maximum contact area or improving load balance, and selects the option that best satisfies it [40].

Additional rules such as Next Fit and Worst Fit have also been considered [41], although they generally lead to less efficient performance, as discussed in [42].

3. **Empty-space detection:** this module is run at each iteration after a new item i is placed to update the representation of the remaining available spaces within the bin (layer) for the placement of the next item ($i + 1$). With reference to Fig. 2, notable methods from literature include:

- Shelf: empty space is identified through a horizontal subdivision of the bin into shelves [43]. For each shelf, the height is set by the tallest item already placed in that shelf, while the residual length within the shelf constitutes the available empty space. If item i does not fit in any existing shelf, the free area above the occupied shelves is regarded as an additional empty space, within which new shelves can be created.
- Skyline: empty space is identified through a one-dimensional skyline profile describing the upper contour of the items already placed in the bin [44]. Whenever a new item i is placed, its upper boundary is incorporated into the profile to reflect the current bin occupancy. The available empty space corresponds to the region above the skyline, where new items can be positioned by aligning them with the contour.
- Maximal Rectangles (MaxRect): empty space is represented as a set of free rectangular regions [45]. After placing item

i , the remaining area of the bin is partitioned into rectangular free spaces, typically one above the item and one adjacent to it laterally. After each subsequent placement, the set of free rectangles is updated by subdividing and removing the regions affected by the new item, thereby maintaining an explicit description of the available empty spaces within the bin.

- Guillotine: empty space is represented as a set of free rectangular regions generated through guillotine cuts applied to the remaining free area of the bin [45]. In practice, after placing item i , the available space is partitioned by full length horizontal or vertical cuts, producing a set of rectangular free regions. These rectangles are updated after each insertion by applying additional guillotine cuts to the affected regions.

For a more complete review on the existent empty space detection algorithms, the reader may refer to [45].

The overall heuristic algorithm can therefore be formulated by combining the modules described above in different ways, prioritizing, for instance, simplicity (Shelf or Guillotine) or compactness (MaxRect). A small comparative study is provided in the next Section, while several broader assessments of these strategies can be found in literature [42, 45, 46]. The resulting bin layout can also be refined through post processing stages, such as Local Search [47], which iteratively improves neighboring solutions, or Try-and-Replace strategies, also referred to as ruin and recreate [48], where a subset of items is removed and reinserted to improve the packing.

The final configuration can also serve as an initial solution for other techniques. In particular, metaheuristics guide or combine heuristics through broader search mechanisms, often inspired by natural or physical processes, to explore the solution space more effectively [49]. Although they do not guarantee optimality, they typically provide a good tradeoff between solution quality and computational effort. Unlike simple heuristics, metaheuristics are designed to escape local optima through adaptive search mechanisms. Representative examples include Genetic Algorithms [50, 51], Simulated Annealing [52, 53], Particle Swarm Optimization [54], Ant Colony Optimization [55], and Tabu Search [56, 57]. These methods can start from an existing solution (warm start) to improve convergence, or generate solutions from scratch (cold start), which is generally more computationally demanding. In BPP applications, metaheuristics are often used to refine heuristic solutions and reach higher quality packing. However, their performance strongly depends on parameter tuning, which is both critical and challenging.

The same methodological classes can be extended to the 3D case by incorporating the additional spatial dimension. Several approaches address the full 3D problem directly using both heuristic [58] and metaheuristic techniques [59]. Other approaches [60] incorporate practical constraints and combine mixed-integer programming, metaheuristics (e.g., Tabu Search), and problem-specific heuristics. Nevertheless, these approaches are often predominantly geometry driven and may provide limited flexibility in accommodating additional objectives relevant to industrial settings, such as load balancing. In this context, the decomposition into 2D sub-problems [29, 30] allows such objectives to be integrated into a more modular decision process. Furthermore, this decomposition supports the treatment of instances with a high number of items while remaining compatible with industrial runtime constraints.

2.3. Preliminary study

As discussed earlier, the 2D and 3D-BPP have been extensively studied, and many solution approaches have been benchmarked and compared using standardized test suites. However, these comparisons

are not always contextualized in palletizing settings, where constraints on stability, handling, and repeatability strongly shape what is practically deployable. Based on the decomposition strategy introduced in Section 2.2, in which the 3D problem is addressed as a sequence of 2D sub-problems, this section focuses on the solution of the resulting 2D-BPP instances. In this respect, exact methods are of limited operational interest for large instances, and metaheuristics with cold starts may exhibit highly variable outcomes under the strict runtime constraints typical of industrial settings. Metaheuristics with warm starts are generally more promising, yet their calibration is nontrivial and often problem dependent. For these reasons, the present work focuses primarily on constructive heuristic methods and on the modular combinations introduced in Section 2.2.

A preliminary study has been conducted in Python by testing several heuristic configurations obtained by combining different item ordering rules, empty space detection, and placement methods. The analyzed configurations are reported in Table 1. A single common benchmark instance is used across all tests to ensure comparability. The items correspond to standardized industrial box sizes, to be packed on one Europallet of size 1200×800 mm [61], but can also be applied to other sizes. The adopted input set is: 2 boxes of 800×600 mm, 4 boxes of 600×400 mm, 8 boxes of 400×300 mm, 16 boxes of 300×200 mm. The considered performance indicators are the algorithm resolution time, the number of layers generated, and the fill ratio. The optimization of the CoG position is not considered in this preliminary study, as it would require inter layer redistribution of boxes to enforce load balancing. This extension will be included in the algorithm development (Section 3).

The results in Table 1 show that all tested configurations achieve very short resolution times. On a personal computer with an Intel(R) Core(TM) i9-14900K vPro @ 3.2 GHz and 64 GB RAM, the measured times range from approximately 0.05 ms to 0.84 ms, confirming the suitability of these constructive heuristics for time constrained industrial contexts. Differences are therefore mainly reflected in solution quality. Among the tested configurations, the best fill ratios are obtained by the Guillotine based strategies coupled with Best Fit criteria. In particular, configurations 5 and 6 attain the highest utilization, with configuration 6 achieving a full fill ratio and requiring fewer layers than the other reported cases. This outcome is consistent with the use of a fragmentation aware Best Fit criterion, which tends to preserve larger contiguous free regions and reduces the likelihood of leaving unusable gaps. Based on these observations, the most effective combinations will be used as baseline building blocks in the subsequent development of the 3D packing algorithm, whose formulation relies on the structured integration of existing and validated methods within a modular engineering framework supporting application-specific adaptation.

3. Framework development

This section presents a framework that supports the optimization of industrial robotic palletizing cells under variable product mixes, where evolving operational scenarios require the generation and virtual validation of novel robot programs prior to deployment on the physical cell. The framework is implemented in RoboDK, which supports the integration of custom programming layers and decision logics, including the proposed bin packing methodologies. In the resulting RoboDK virtual prototype, a set of Python modules exploits the RoboDK Application Programming Interface (API) to generate and validate robotic palletizing sequences [62]. The API provides direct access to the 3D station entities, including the robot, tool, frames, objects (e.g., box items), and mechanisms, thereby extending RoboDK built-in capabilities with user-defined features. A separate Python layer supports direct connections to physical robots through dedicated modules for streaming instructions to the robot controller, enabling tighter coupling between the virtual and real cells and supporting closed-loop execution within a Digital Twin oriented approach [63].

Table 1
Tested 2D-BPP solution approaches obtained by combining different heuristic modules.

N°	Item Ordering	Empty space allocation	Placement	Resolution time [s]	N° of layers	Fill ratio [%]
1	Decreasing height	Shelf	First Fit	5.2×10^{-5}	5	95.33
2	Decreasing area	MaxRect	First Fit	7.9×10^{-5}	5	95.33
3	Decreasing area	Guillotine	First Fit	51×10^{-5}	5	95.33
4	Decreasing area	Skyline	Best Fit (min. height difference)	18.6×10^{-5}	5	93.75
5	Decreasing area	Guillotine	Best Fit (max. touching perimeter)	15.7×10^{-5}	5	98.45
6	Decreasing area	Guillotine	Best Fit (min. space fragmentation)	84×10^{-5}	4	100

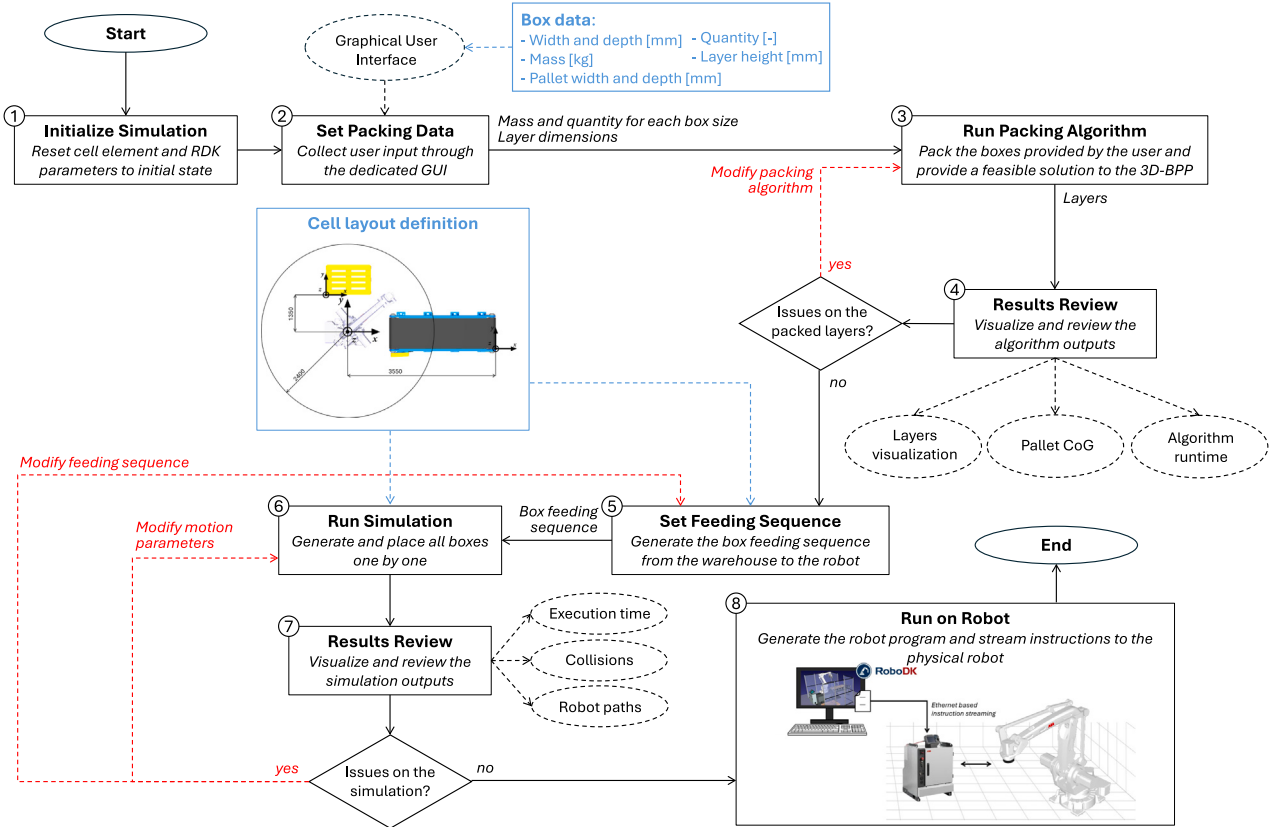


Fig. 3. Flowchart of the proposed framework for robotic palletizing process planning and validation. Required inputs are shown in blue, whereas design fallback options are shown in red.

Fig. 3 summarizes the proposed workflow, including the required inputs (in blue), the eight operational steps, the two decision points, and the designer fallback (in red). The main program orchestrates the execution of the Python scripts dedicated to each step, enabling iterative simulation, results assessment, and parameter tuning until the target requirements are satisfied, after which the solution can be deployed on the physical robot. In the following, all the implemented steps are discussed in detail. The entire framework is made available from the supplementary material Section of this paper to enable further developments.

3.1. Input data

With reference to Fig. 3, the first input consists of the CAD and layout definition required to reconstruct the palletizing cell within a dedicated RoboDK station. This dataset must include the robot, the end-effector (gripping tool), and any auxiliary equipment (e.g., linear tracks, pedestals). It must also specify the material handling equipment, namely the item feeding system (e.g., conveyor), the number and type of buffers for temporary item storage, the pallet format, and any additional fixtures relevant to the operational scenario. In addition to the component geometries, the nominal position of each element

in the workspace is required to replicate the physical arrangement of the real cell. Furthermore, frame definitions and target positions (e.g. for item pick-and-place) can potentially be extracted from robot programs to reproduce the currently adopted reference systems and programming sequences. A basic example of virtual setup is shown in Fig. 4, where the relevant geometric and procedural information is reported. Notably, RoboDK adopts a frame-based station structure that inherently supports parametric layout definition, allowing layout variants to be evaluated directly in simulation. Once the station is generated, the framework interfaces with its elements through the API by referencing item names within the Python scripts. Consequently, station updates, such as geometry replacements or repositioning, can be accommodated without code changes provided that the item naming convention is preserved.

The second input (box data in Fig. 3) specifies the physical characteristics of the boxes (items) and the pallet (bin). For each box type, the user provides the dimensions in millimeters (width and depth), the quantity, and the mass in kilograms. The layer height, which defines the third dimension used by the packing and placement routines, is provided as a single value in millimeters and is assumed valid for all boxes and layers. The pallet dimensions are defined by selecting

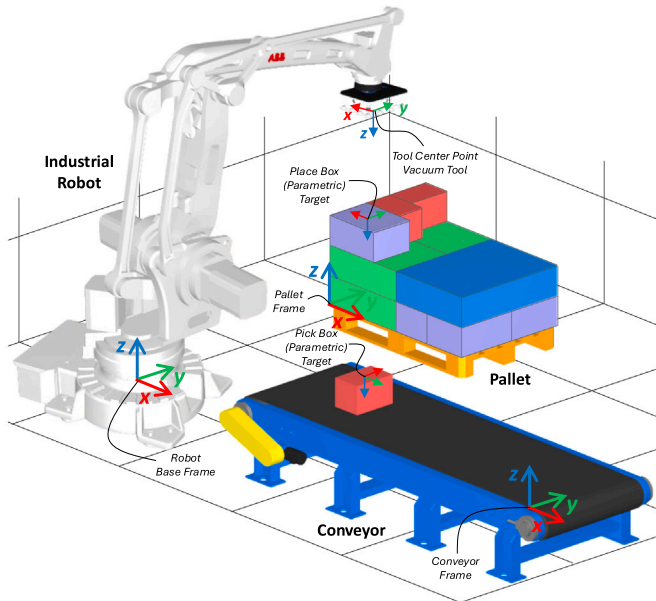


Fig. 4. Virtual palletizing cell in RoboDK, with the reference frames and robot targets required to execute the box pick-and-place tasks.

a standard ISO format from a predefined list (e.g., 1200×1000 , 1200×800 , 1140×1140 , 1100×1100).

3.2. Preliminary steps

Prior to executing the core elements of the framework, namely the packing and palletizing routines, a series of preliminary operations are performed. In particular, **Step 1** in Fig. 3 restores the simulation to a known baseline state to ensure repeatability across consecutive runs by removing any residual changes from the previous execution. Mechanisms that may have moved during the previous simulation, such as the robot and the conveyor, are returned to their initial configuration, and fixed cell objects, including the pallet or pallets, are restored accordingly. Objects previously created or manipulated, such as boxes transferred from the warehouse to the conveyor or the robot workspace, are also cleared from the station. Data consistency across scripts is maintained through RoboDK parameters, which can be read and written via the API by any Python script within the project and are reset to their default values during initialization.

Once the reset is completed, **Step 2** sets the packing data, corresponding to the user inputs described in Section 3.1 and shown in Fig. 3 as Box data. A dedicated routine provides a graphical interface to enter the dimensions, mass, and quantity of each box, together with the layer height and the selected pallet format. After confirmation, the inputs are stored as RoboDK parameters so that downstream steps can access a single consistent dataset and continue the workflow without further user interaction.

3.3. Packing algorithm

Step 3 reads the box data from the RoboDK parameters saved during **Step 2**, executes the proposed 3D packing algorithm, visualizes the obtained results, and writes the packing solution back into the RoboDK parameters. The proposed algorithm implements a 3D-BPP strategy aimed at optimizing the placement of boxes with varying dimensions within the 3D space of a pallet. Although pallet dimensions are selected by the user from a predefined set of standard formats, the algorithm can also handle arbitrary custom pallet dimensions thanks to its parametric formulation. As discussed in Section 2.2, to simplify the mathematical

formulation and improve computational efficiency, the 3D problem is decomposed into a sequence of 2D-BPP instances, one per layer, solved through a combination of heuristic methods. Based on the preliminary study in Section 2.3 (Table 1), items are processed in decreasing area order and placed sequentially using a Best Fit rule within a Guillotine partitioning scheme. In addition to maximizing the usable area of each layer, the algorithm continuously monitors the pallet overall CoG. The in-plane deviation of the CoG from the geometric center of each layer is used as an additional decision criterion for determining the final solution. Among feasible alternatives, the placement that improves balance is selected.

In addition to the boundary and non-overlapping constraints introduced in Section 2.1 (see Eqs. (3) and (4)), no specific gripping or releasing offsets are introduced, since in this work vacuum tools are assumed. This common choice in 2D/3D bin packing avoids unnecessary reductions of usable area, particularly for small box formats. The absence of such offsets does not restrict other end-effector types. Many industrial solutions rely on dedicated mechanisms or positioning strategies to avoid gripper interference. In the literature, specific approaches have been proposed for grasp planning using multi-fingered grippers, confirming the relevance of the problem of grasp pose selection for rigid objects in industrial contexts [64]. These aspects, which depend on the specific robotic cell, lie outside the scope of the packing algorithm and can be handled in the subsequent steps of the framework, where additional modules generate cell-specific pick-and-place robotic sequences.

A visual representation of the algorithm workflow is provided in Fig. 5, while the corresponding script is reported below. A detailed description of the main substeps (A–D) is provided in the following.

```
# Import the RoboDK API and Optimization Modules
from robolink import *
from robodk import *
import MyOptimization as opt

# Establish a link with the RoboDK simulation
RDK = robolink.Robolink()

# Read data from RoboDK parameters
pallet_w, pallet_h, b_sizes, b_quantities, b_masses,
    layer_h = read_RDK_parameters()

# Generate the list of 2D rectangles representing the
    boxes
boxes = generate_list_boxes(b_sizes, b_quantities,
    b_masses)

# Extract single box-type full layers if possible
full_layers, remaining_boxes = opt.initial_check(
    boxes, pallet_w, pallet_h
) # Substep A

# Perform the packing
packed_layers = full_layers + opt.packing(
    remaining_boxes, pallet_w, pallet_h
) # Substep B

# Separate fully and partially filled layers
full_layers, partial_layers = [], []
for pl in packed_layers:
    if pl.is_full(): full_layers.append(pl)
    else: partial_layers.append(pl)

# If partially filled layers are present, execute
    Substep C
if len(partial_layers) != 0:
    new_partial_layers = []
```

```

for pl in partial_layers:

    # Check if the current BB is already optimal
    # If not, try to minimize it
    optimal = is_optimal_bb(pl)
    if not optimal: new_pl, optimal =
        opt.minimize_BB(pl)

    # If an optimal BB is found, store the new layer
    if optimal: new_partial_layers.append(new_pl)
    # Otherwise, perform the removal test and split the
    # layer
    else:
        new_pl_1, optimal_1, new_pl_2, _ =
            opt.split_partial_layer(pl)
        # If the split is successful, store the two new
        # layers
        if optimal_1:
            new_partial_layers.append(new_pl_1)
            new_partial_layers.append(new_pl_2)
        # Otherwise, keep the original layer unchanged
        else: new_partial_layers.append(pl)

# Reorder all the layers to obtain the final packing
# layout
packed_layers = opt.reordering(
    full_layers, new_partial_layers
) # Substep D

# Write the layout data into RoboDK parameters
write_layers_to_RDK(packed_layers)

```

Substep A - Initial check: It verifies whether the available quantity of any single box type (treated as a 2D rectangle for layer packing) is sufficient to form one or more complete homogeneous layers. This is intended to immediately resolve straightforward cases and reduce the search space for the subsequent heuristics. The procedure follows the passages below:

1. For each box type, define a feasible layer layout on the pallet to determine the number of boxes required and their arrangement, allowing rotations if needed.
2. Check whether the available number of boxes of that type is sufficient to fill the identified layer. If so, generate one or more complete layers, assign the corresponding boxes, and remove them from the input list.
3. Return the list of homogeneous full layers and the updated list of remaining (i.e. unused) boxes.

If no full layer can be created due to insufficient quantity, the substep returns the original box list unchanged. Beyond computational benefits, homogeneous layers can facilitate unpacking and improve layout regularity and balance. If required, the designer may also define custom layers consisting of a prescribed set of box types. The algorithm then verifies whether the provided quantities support one or more instances of the custom layer.

Substep B - Packing: It takes as input the remaining boxes (i.e. those not used in **Substep A**) and arranges them into the necessary layers. The packing is implemented with a Best Fit rule, selecting the placement that minimizes fragmentation of the residual space. If fragmentation is equivalent, placements that would create gaps smaller than the minimum box format (based on the smallest side among remaining boxes) are discarded. After each placement, the residual free space is partitioned using a Guillotine strategy and represented as a set of rectangular regions (see Fig. 2), which become candidates for subsequent placements. The procedure follows this logic:

1. Identify the smallest side among all boxes in the list, used to assess unusable fragmented spaces.
2. Sort boxes by decreasing area, placing larger items first.
3. Iterate through the sorted list and attempt to place each box into the existing layers, considering 90° rotations.
4. Insert every box into the first layer where a valid placement is found under the Best Fit rule. If no existing layer can accommodate the box, open a new layer and place it there.
5. Returns the list of packed layers containing all placed boxes.

The layers produced here, combined with those generated in **Substep A**, already define a feasible packing solution.

Substep C - Repacking: The initial solution may include fully filled or partially filled layers, depending on whether the total box area equals the pallet area or not. This substep targets potential improvements for partially filled layers, while fully filled layers are temporarily left unchanged. To this end, the Bounding Box (BB), namely the smallest 2D rectangular region aligned with pallet axes that completely contains all boxes in a layer, is considered. For partially filled layers, the BB is generally smaller than the pallet area, although in some cases it may coincide with it. The objective is to process one partially filled layer at a time and repack its boxes to obtain a new arrangement with a reduced BB area compared to the original one produced in **Substep B**. This yields more compact layouts and facilitates their handling in the subsequent steps. For each partially filled layer, the algorithm first checks whether the current BB is already optimal, which occurs when the sum of the box areas equals the BB area. Otherwise, additional heuristic modules (e.g., MaxRect and Best Fit, see Table 1) are applied, each time verifying whether the resulting packing achieves an optimal BB. If an improved solution is found, the repacked layer is returned, otherwise the original layer is preserved. In the latter case, as a final fallback, the algorithm performs a removal test on the partial layer: boxes are temporarily removed one at a time, and the remaining set is re-evaluated. As soon as a box is found whose removal yields a more compact arrangement with an optimal BB for the remaining items, the algorithm finalizes a layer with those items and assigns the removed box to a new bin, thereby splitting the original set into two layers. This mechanism is intended to handle cases in which a single additional box degrades compactness and prevents an optimal BB, so that the packing is completed by generating two feasible layers instead of forcing a suboptimal partial one. The main drawback is that this strategy can increase the number of layers and, consequently, the overall pallet height. On the other hand, it may facilitate load balancing, since the resulting layers tend to be more compact and easier to weight-distribute consistently.

Substep D - Reordering: This substep determines the final stacking order and in plane orientation of all layers, including homogeneous layers from **Substep A**, packed layers from **Substep B**, and repacked partially filled layers from **Substep C**. The ordering follows placement rules defined to improve stability and reduce the overall pallet imbalance, represented by the vector **R**, which measures the cumulative CoG deviation on the *XY* plane with respect to the pallet centroid. In general terms, the imbalance can be defined as follows:

$$\mathbf{R}_l = \sum_{j=1}^l m_j (\mathbf{g}_j - \mathbf{c}) \quad (8)$$

where l denotes the number of layers considered, m_j is the mass of the j th layer, $\mathbf{g}_j$ its CoG expressed in (x, y) coordinates, and $\mathbf{c}$ the pallet centroid, also expressed in (x, y) coordinates. Naturally, when l equals the total number of layers of the pallet, $\mathbf{R}_l = \mathbf{R}$. With reference to Eq. (7), a practical index is the Euclidean distance d_{xy} , corresponding to the norm of **R**. Along the vertical (*Z*) axis, the objective is to keep the CoG as low as possible to improve stability during pallet transportation.

To achieve these goals, the algorithm places the fully filled layers at the bottom and sorts them by decreasing mass, with the heaviest

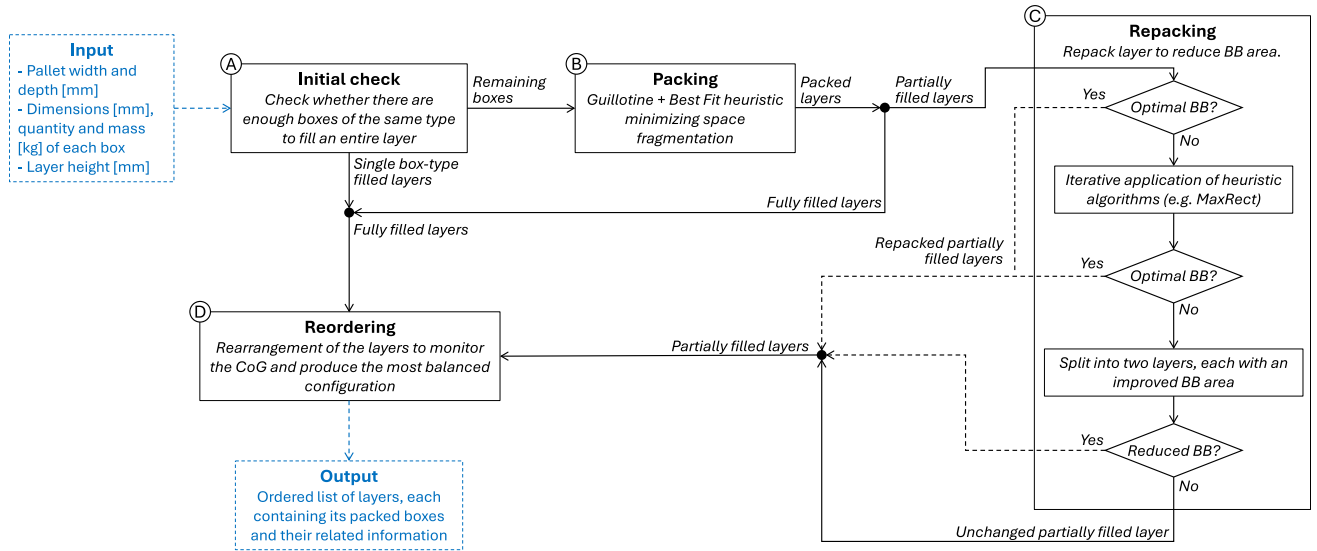


Fig. 5. Flowchart of the proposed packing algorithm.

layer placed first. Starting from the second layer, each layer is evaluated in its current orientation and after a 180° rotation. The option that best compensates for the accumulated CoG imbalance (i.e., reduces the norm of $\mathbf{R}_l$) is selected. Partially filled layers are subsequently placed above the fully filled ones, sorted by decreasing BB area. Each partial layer is positioned so as to counterbalance the residual imbalance, though its items may shift only within the BB of the layer directly below it. Under this constraint, the target CoG position of the j th partially filled layer for canceling the cumulative imbalance generated by the underlying layers is evaluated as:

$$\mathbf{g}_{j,\text{opt}} = \mathbf{c} - \frac{\mathbf{R}_{l-1}}{m_j} \quad (9)$$

where m_j is the mass of the considered layer and $\mathbf{R}_{l-1}$ is the cumulative imbalance generated by the underlying layers. Once computed, a rigid translation is applied to the set of items in the partially filled layer to align its current CoG ($\mathbf{g}_j$) with the target position ($\mathbf{g}_{j,\text{opt}}$). This strategy progressively reduces the global imbalance while ensuring that upper layers rest on a sufficiently wide and stable support. After reordering, the algorithm returns an ordered list of layers, from the lowest to the highest. Each layer contains its packed boxes, and each box includes all the information needed for identification and for its correct placement within the pallet, namely: (i) ID number, (ii) width, (iii) depth, (iv) height, (v) x and y coordinates from the pallet origin (bottom-left corner), (vi) rotation flag; (vii) layer number.

The outputs of **Step 3** are made available to the designer for review in **Step 4**. These include a graphical visualization of the packed layers produced by the algorithm, along with the total mass and CoG deviation for each layer, and the algorithm execution time. After examining these results, the designer reaches the first decision point of the proposed framework. If the packed layers are satisfactory, the solution is forwarded to the next step of the framework. Otherwise, the designer returns to the beginning of **Step 3** to revise the packing logic. Since the algorithm is deterministic, changes occur only when its inputs or internal rules are modified, therefore any adjustment requires adapting the algorithm itself. If the reported issues concern the CoG of an individual layer, **Substep C** and/or **Substep D** must be revised. Conversely, issues involving total mass distribution or box arrangement require updating the packing strategy in **Substep B** and selecting resolution methods better suited to the specific instance (e.g., considering other combinations in Table 1 or from literature). If the issue concerns the z -coordinate of the pallet CoG, it indicates an unsuitable stacking order, and the designer must therefore act on **Substep D**. If the runtime is not

satisfactory, the algorithm must be simplified, accepting a faster but potentially less optimal solution. Once this refinement loop converges to a satisfactory outcome, the packed layer data are stored for subsequent use.

3.4. Feeding sequence definition

In the preliminary setup of the palletizing simulation, **Step 5** generates the sequence in which boxes are delivered by the feeding system to the robot workspace. In an industrial setting, boxes are conveyed to the robot area from a warehouse or from a temporary storage, though this aspect does not affect the sequencing logic implemented in the virtual environment. This step takes as input the packed layers reviewed in **Step 4** and the cell layout definition, which is application dependent and is therefore provided as a framework input, as discussed in Section 2.1. In this paper, the feeding system is represented by a conveyor, shown in Fig. 4, but the same sequencing approach applies if alternative transport devices are adopted. In those cases, the simulation model must be extended to include the additional equipment and to reproduce its specific transport behavior, while the underlying rationale of this step remains unchanged.

Feeding strategies depend on the adopted cell design and layout. With a minimal configuration featuring a single feeding line and no intermediate buffers (Fig. 4), the incoming boxes are pre-ordered according to the pallet packing order obtained in **Step 3**. The sequence therefore follows the layer building order, starting from the boxes assigned to the lowest layer and progressing upward until the pallet is completed. For more complex layouts, different feeding policies may be adopted. For instance, the robot can be supplied with homogeneous batches, where all boxes of a given type are delivered consecutively, followed by the next type, or boxes can arrive in a random order. These policies decouple arrival order from placement order and, as a consequence, typically require temporary buffer stations within the robotic workspace to hold boxes that arrive before they are needed for the layer currently under construction. To manage this, in **Step 5** a dedicated Python script generates the feeding sequence according to the user defined policy and writes the resulting conveyor sequence to RoboDK parameters.

3.5. Robotic simulation

Step 6 executes the simulation in RoboDK through substeps A to G, as summarized in Fig. 6. Each substep is implemented with one or more

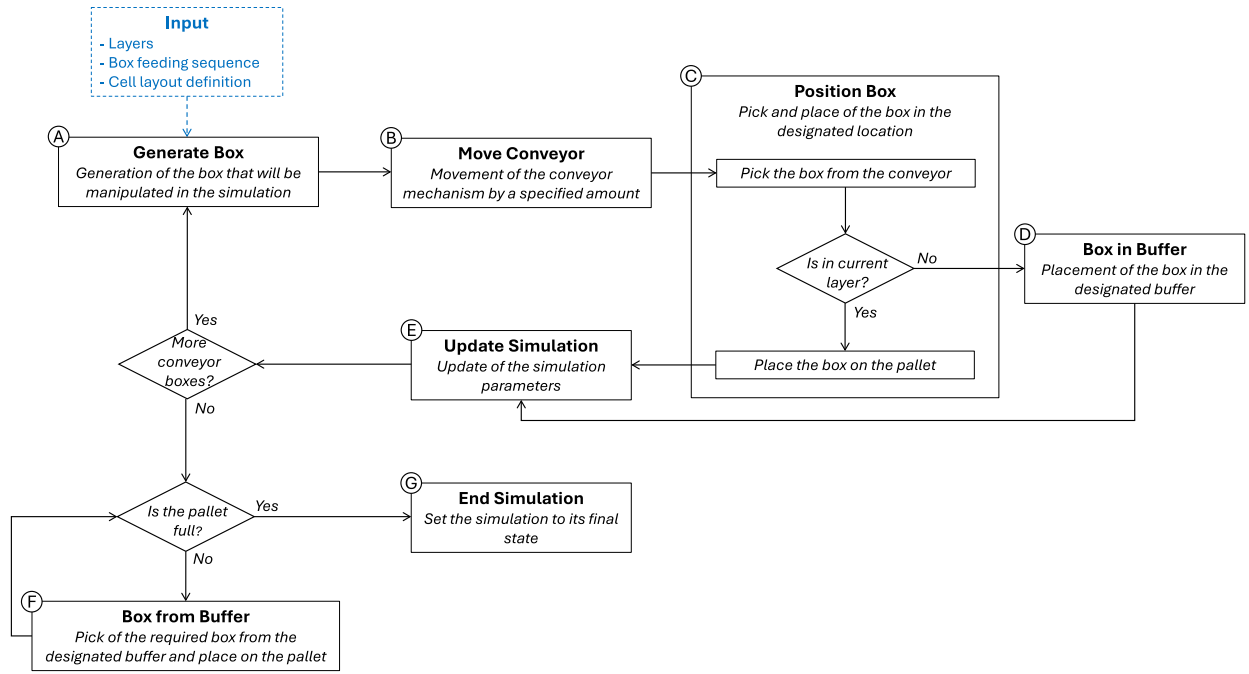


Fig. 6. Flowchart describing the Python-based robotic simulation in RoboDK.

scripts, invoked sequentially for each box to be placed, as illustrated in the script reported below.

```

# Import the RoboDK API
from robolink import *
from robodk import *

# Establish a link with the RoboDK simulation
RDK = robolink.Robolink()

# Read the total number of boxes to process (RoboDK
  ↳ parameter)
tot_box = int(RDK.getParam('Tot_Box'))

# Retrieve the programs involved in the iterative cycle
generate = RDK.Item('Generate_Box.py',
  ↳ robolink.ITEM_TYPE_PROGRAM_PYTHON) # Substep A
move = RDK.Item('Move_Conveyor.py',
  ↳ robolink.ITEM_TYPE_PROGRAM_PYTHON) # Substep B
position = RDK.Item('Position_Box.py',
  ↳ robolink.ITEM_TYPE_PROGRAM_PYTHON) # Substeps C-D
update = RDK.Item('Update_Simulation.py',
  ↳ robolink.ITEM_TYPE_PROGRAM_PYTHON) # Substep E

# Execute Substep A to E for each box
for u in range(1, tot_box + 1):
    for prog in (generate, move, position, update):
        prog.RunCode()
        prog.WaitFinished()

# Count how many boxes are placed on the pallet
all_items = RDK.ItemList(robolink.ITEM_TYPE_OBJECT)
pallet_items = []
frame_item_pallet = RDK.Item('Frame Item Pallet',
  ↳ robolink.ITEM_TYPE_FRAME)
for item in all_items:
    if item.Parent() == frame_item_pallet:
        pallet_items.append(item)

# Read current and total layer numbers (RoboDK
  ↳ parameters)
  
```

```

layer_number = int(RDK.getParam('Layer_Number'))
tot_layer = int(RDK.getParam('Tot_Layer'))

# If the pallet is incomplete, retrieve boxes from the
  ↳ buffer
if len(pallet_items) != int(tot_box):
    for u in range(layer_number, tot_layer + 1):
        buffers = RDK.Item('Box_FromBuffers.py',
          ↳ robolink.ITEM_TYPE_PROGRAM_PYTHON) #
          ↳ Substep F
        buffers.RunCode()
        buffers.WaitFinished()

# Once the pallet is complete, stop the simulation
end=RDK.item('End_Simulation.py',
  ↳ robolink.ITEM_TYPE_PROGRAM_PYTHON) # Substep G
end.RunCode()
end.WaitFinished()
  
```

For a correct interpretation, in the following a detailed explanation of each step is provided:

Substep A - Generate box: The first script is dedicated to generating the next 3D box in the workstation according to the feeding sequence determined during **Step 5**. In RoboDK, this can be achieved in two ways: (i) copying an existing box already present in the station and scaling it to the required dimensions, which is convenient when the box includes specific geometric or visual features, or (ii) generating a new box.

Substep B - Move conveyor: Once the box has been generated, this substep commands the motion of the conveyor mechanism, modeled as a one degree of freedom motion system. The conveyor is advanced by a fixed increment at a prescribed target velocity, selected according to the specific application. Since the generated box is attached to a dedicated frame that follows the conveyor motion, it is transported forward until it reaches the predefined *Pick Box Target* (see Fig. 4).

Substep C - Position box: This substep handles the manipulation of the current box, namely the pick from the conveyor and the subsequent

placement either on the pallet or on the buffer. The Python routine reads the box-specific data stored in RoboDK station parameters, including the ID, width, depth, height, placement coordinates (x_{box}, y_{box}) (expressed with respect to the pallet/buffer origin (0,0) located at the lower-left corner), rotation flag, and layer index. The motion targets used by the robot are then updated parametrically via code, prior to commanding any robot motion. In particular, the *Pick Box Target* is translated along the Z axis according to the box height h , so that the approach and grasp are executed at the correct elevation for the current box. Similarly, the *Place Box Target* is repositioned before the placement motion, either with respect to the pallet reference frame or the buffer reference frame. With reference to the 3D-BPP case in Section 2.1 and to Fig. 4, the *Place Box Target* coordinates are computed as:

$$x_{place} = x_{box} + \frac{w}{2} \quad y_{place} = y_{box} + \frac{d}{2} \quad z_{place} = z_{layer} + h$$

where z_{layer} denotes the vertical offset of the current layer from the pallet base, to be updated once each layer is completed and a new one is started. In the formulas, the in-plane offsets are swapped when the rotation flag is active. Robot motions are executed with a two-stage scheme for both pick and place. First, a point-to-point (e.g., MoveJ) motion brings the robot to a non-interfering approach target obtained by offsetting the corresponding target upward along Z by a safety clearance, i.e., outside the box envelope and potential local obstacles. Then, a fine linear MoveL instruction is used to approach the target and, after actuating the gripper through a digital IO command, to retract along the same direction. The IO command is issued between the approach and retract segments: the gripper is closed to pick the box and opened to release it. If the box is placed on the pallet, the next executed routine is *Substep E*. If the box is instead sent to the buffer, *Substep D* is executed to perform the buffer placement before proceeding to *Substep E*.

Substep D - Box in buffer: The number of available buffers is specified in the input data, while the corresponding utilization logic may be defined within this substep. Typical strategies include assigning one dedicated buffer to each box type or adopting a set of shared buffers across types. In both cases, the designer must define and implement the positioning rules within each buffer, i.e., the internal layout and the placement order. A dedicated counter is used to track the number of boxes already stored in the selected buffer. For each incoming box, its target pose is computed as explained in *Substep C*, by taking into account the current counter value and the predefined buffer layout. After a successful placement, the counter is incremented to update the next available buffer slot.

Substep E - Update simulation: After the box has been placed, the simulation state is updated by incrementing the RoboDK parameter that stores the index of the next box in the conveyor sequence. If additional boxes remain in the sequence, the workflow restarts from *Substep A*. Otherwise, if no further boxes are available, all items have been processed and a final decision is made. If the pallet has been completed without using the buffers, the workflow advances directly to *Substep G* to finalize the simulation. Otherwise, the pallet is still incomplete and *Substep F* is executed to retrieve and place the buffered boxes.

Substep F - Box from buffer: The boxes temporarily stored in the buffer(s) are retrieved one by one and placed in their assigned positions on the pallet. The corresponding Python routine is executed iteratively until all missing boxes have been transferred. Based on the parameters tracking the number of boxes stored in each buffer, the script selects the next item to retrieve and computes the associated parametric target on the buffer, using the same target definition previously adopted for buffer placement, now interpreted as a pick pose. In parallel, by tracking the pallet layer currently being filled and the set of pending placements, it computes the corresponding parametric *Place Box Target* on the pallet. After each placement, the box is marked as completed

and the workflow returns to the second decision point in Fig. 6 to re-evaluate the pallet fill state. If the pallet is still incomplete, the next buffered box is processed. Once the pallet is complete, the workflow advances to *Substep G*, which finalizes the simulation.

Substep G - End simulation: Once the pallet is full, the robot and the remaining mechanisms are moved to a predefined final configuration. The simulation then terminates and returns the full output vectors of the workflow.

In the described logic, buffered items are retrieved only after the conveyor sequence has been fully processed. While this guarantees completion of all layers, it is not necessarily distance- or time-optimal. In some cases, it may be advantageous to interleave conveyor picks with buffer retrievals, for example by temporarily interrupting the conveyor picking loop to place one or more buffered boxes, thereby reducing the overall robot travel distance and, potentially, the total number of pick and place cycles. Additionally, dedicated auxiliary equipment may be integrated to place anti-slip sheets between consecutive layers. These and similar variants can be accommodated within the proposed framework, since the workflow is fully parametric and can be extended by introducing additional devices and associated subroutines.

3.6. Results review

The specific outputs to be inspected depend on the case study and on the design objectives. In general, the most informative indicators for a palletizing system include the total execution time required to complete the process, the robot saturation, traveled distance, and path diagnostics (e.g. collisions, singularities, or joint limit violations). More advanced indicators may include the energy consumption and the motion error committed by the robot, which would aid in the refinement and optimization of the robot motion planning. However, such advanced behavioral modules (see, e.g., Refs. [3,4]) are not available by default in RoboDK and must therefore be integrated upon detailed assessment and modeling.

After reviewing these outputs, the designer reaches the second decision point of the framework (see Fig. 3, after Step 7). If the simulation is satisfactory and the results meet the requirements, the workflow proceeds to Step 8. Otherwise, the designer is redirected to the most appropriate earlier stage, depending on the identified issue. If the execution time is unsatisfactory, the corrective action typically involves tuning motion parameters and, when relevant, revising the buffer management logic. If collisions are detected, parameters such as approach offsets and intermediate way-points should be refined, and additional targets may be introduced to enforce safer paths. If the issue concerns the overall robot path quality, further targets and constraints can be added to shape the motion as required. If buffer utilization is suboptimal, the designer may modify the box feeding sequence, or evaluate the addition or removal of buffer capacity. From an implementation standpoint, adjustments to the box feeding sequence are implemented in Step 5, additional targets can be introduced either as physical elements in the cell layout or directly in code within Step 6, where also motion parameter changes can be applied. Modifications in the number of buffers, and in the corresponding placement and retrieval logic, require acting on *Substep D-F* of Step 6. In all cases, the affected steps of the framework are executed again and the results of the updated simulation are reviewed iteratively until the requirements are met.

3.7. Deployment on real robot

Once a simulation with satisfactory performance has been achieved, Step 8 enables online execution on the physical robot by streaming the validated instructions from the digital model, within a Digital Twin oriented workflow that supports process optimization and continuous feedback monitoring from the real asset. To this end, a communication link must be established between the computer running the

Table 2

Tested scenarios and performance results of the packing algorithm. Type 1, 2, 3, and 4 correspond to box sizes of 800×600 mm, 600×400 mm, 400×300 mm, and 300×200 mm, respectively.

	Type 1	Type 2	Type 3	Type 4	Total	Layers	Full layers	Execution time [s]	Pallet height [mm]	CoG [mm]
Test 1	1	3	7	15	26	4	3	1.048×10^{-2}	1200	(600.0, 400.0, 334.6)
Test 2	2	8	12	16	38	6	5	2.651×10^{-3}	1800	(600.0, 400.0, 592.1)
Test 3	1	10	12	18	41	6	5	1.576×10^{-2}	1800	(600.0, 400.0, 662.2)
Test 4	4	6	12	24	46	7	6	3.329×10^{-2}	2100	(600.0, 408.7, 789.1)
Test 5	2	12	14	20	48	7	7	6.738×10^{-3}	2100	(600.0, 412.5, 687.5)
Test 6	5	10	15	20	50	9	8	1.195×10^{-2}	2700	(600.0, 407.0, 768.0)
Test 7	2	14	16	20	52	8	7	1.231×10^{-2}	2400	(600.0, 400.0, 876.9)
Test 8	1	13	20	20	54	8	7	1.109×10^{-2}	2400	(600.0, 407.4, 916.7)
Test 9	0	8	36	40	84	9	9	6.328×10^{-3}	2700	(604.8, 408.3, 1021.4)
Test 10	0	4	28	64	96	9	8	1.708×10^{-2}	2700	(600.0, 400.0, 1012.5)

virtual station and the industrial robot controller through the Python-based interface provided by RoboDK. The connection setup and data exchange depend on the specific robot brand and controller family, which determine the available communication drivers and supported command sets. Most modern controllers provide Ethernet connectivity, and the link is commonly implemented over TCP/IP sockets, namely bidirectional communication channels between devices on the same network. The resulting architecture follows a client-server structure in which the robot controller acts as the server, exposing a defined interface, while the PC operates as the client, sending commands and receiving status feedback. In this way, the digital model delivers the validated instruction stream and the relevant process parameters to the physical cell, and verifies correct execution through acknowledgments and state monitoring. Unlike offline programming, where the simulated sequence is translated into vendor specific robot code and executed as a complete program, this approach does not require generating the full program in advance.

In the online mode, Python scripts within RoboDK transmit individual instructions, such as MoveJ, MoveL, and I/O activations, to a program running on the controller, which interprets and executes them sequentially. Command transmission typically follows a synchronous handshake: the client issues a new instruction only after the controller has completed the previous one and returned an acknowledgment. This interaction can be modeled as a finite state machine with transitions between *Waiting*, *Action*, and *Exit*, ensuring ordered execution and preventing command overlap. Each transmitted instruction encapsulates all parameters required for correct execution, including motion type, target definition in Cartesian or joint space, speed, and cornering or approximation settings. The approach extends to all supported instruction types, including reading inputs and changing output states (gripper open/close), speed and acceleration overrides, blending or cornering settings, target and tool updates, wait or synchronization commands, and basic flow control signals (start, stop, pause, resume, reset). On the controller side, the server program decodes the received package and maps it to native robot commands. Since the controller executes each instruction without lookahead, short dwell times may occur between segments, particularly when blending is not available or disabled. To improve motion continuity and reduce cycle time, a hybrid strategy can be adopted: optimized subprograms, or precomputed target lists for long paths, are pre-loaded on the controller via the same online channel, either in advance or during pauses, and then triggered for execution when required. On the controller, the server side can be implemented directly in the proprietary robot language, as in the case of ABB, or may rely on external middleware, as in the case of KUKA, where *KUKAVARPROXY* enables TCP/IP read and write access to internal robot variables and provides an interface to a running KRL program.

4. Validation on a case study

This section validates the proposed framework in two stages. First, the novel Python packing module (Section 3.3) is evaluated on multiple

input datasets, including different box mixes, to assess its ability to generate compact and feasible layer layouts. Second, a representative case study validates the overall workflow by executing all previously described steps within the robotic simulation environment and verifying the resulting palletizing sequence and framework consistency.

4.1. Packing algorithm evaluation

To validate the packing algorithm, detailed in Section 3.3 and schematized in Fig. 5, tests were conducted using standard pallet and box sizes. A 1200×800 mm Europallet was considered, together with boxes of four types: (1) 800×600 mm, (2) 600×400 mm, (3) 400×300 mm, and (4) 300×200 mm [61], and a constant layer height of 300 mm. The quantities of these box types were systematically varied, yielding 10 distinct input mixes. Table 2 summarizes the test set and the adopted evaluation metrics, which are intended to capture both computational efficiency and solution quality. For each test, the table reports the number of boxes for each type, the total number of input boxes, the total number of generated layers, the number of fully filled layers, the algorithm execution time, the resulting pallet height (computed by considering only the stacked layers and excluding the pallet base), and the pallet CoG coordinates referenced to a (0, 0, 0) origin at the bottom-left corner of the pallet. In this initial analysis, a uniform mass of 1 kg was assigned to each box, while non uniform mass distributions will be investigated next.

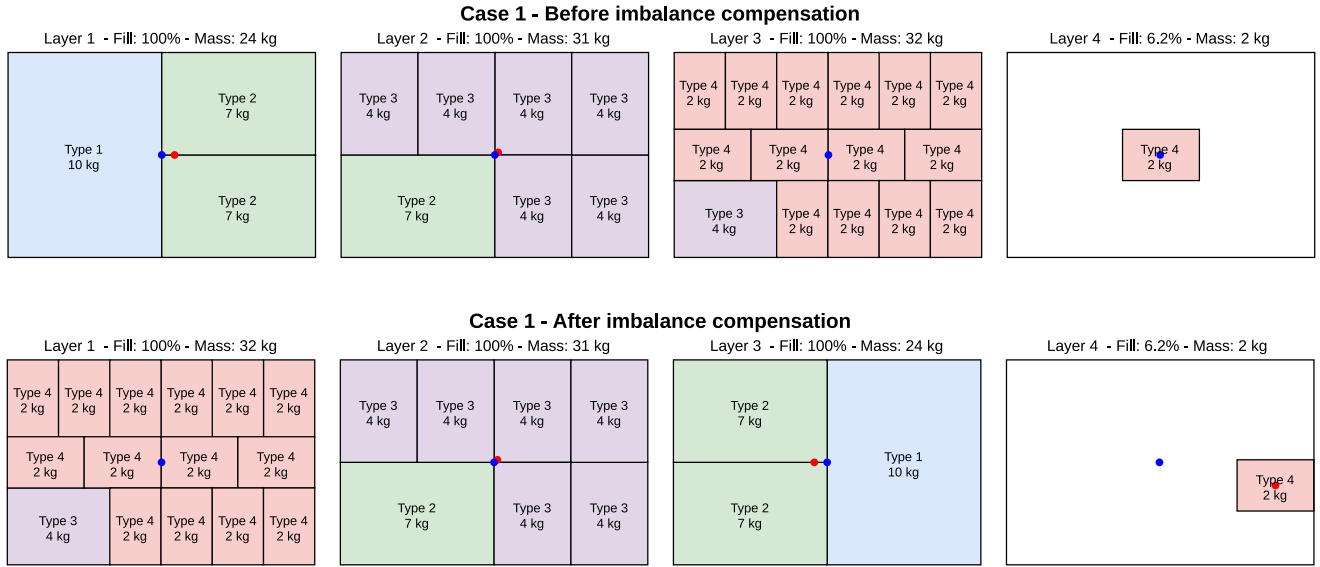
Overall, the algorithm achieves high packing efficiency, as indicated by the large fraction of fully filled layers, together with short execution times across all tested mixes, with a maximum observed value of 33 ms (Test 4). With respect to the ideal CoG, located at $x = 600$ mm, $y = 400$ mm, and within the lower half of the pallet height, the resulting coordinates remain close to the reference values. In particular, the x coordinate coincides with the ideal position for all tests except Test 9, where a deviation of 0.8% is observed. The y coordinate equals 400 mm in five tests, while the remaining cases exhibit small offsets, with a maximum deviation of 3.1% in Test 5. In all cases, the z coordinate remains markedly below half of the corresponding stacked height, indicating a low vertical CoG position.

Test 1 was then selected for further analysis to quantify the effect of box mass distributions on the pallet CoG. To this end, the masses assigned to Types 1, 2, 3, and 4 were varied, resulting in five additional cases, whose results are reported in Table 3. Here, the CoG coordinates, expressed with respect to the bottom-left corner of the pallet, are evaluated before and after imbalance compensation (*Substep D*, see Section 3.3). In addition, the Euclidean distance on the XY plane between the pallet CoG and its centroid ($x = 600$ mm, $y = 400$ mm) is reported before and after compensation, together with the corresponding percentage improvement. The results show a clear reduction in overall pallet imbalance, with the x and y coordinates approaching or matching the reference values in most cases, while the z component remains within the lower half of the pallet, i.e. below the reference threshold of 600 mm defined for Test 1. Fig. 7 illustrates the packed layers corresponding to Table 3, with emphasis on Case 1, and in

Table 3

Effects of varying the mass of the four box types on the pallet CoG coordinates. The product mix of Test 1 is adopted for all cases.

	Type 1 [kg]	Type 2 [kg]	Type 3 [kg]	Type 4 [kg]	CoG Before [mm]	CoG After [mm]	d_{xy} Before [mm]	d_{xy} After [mm]	d_{xy} reduction [%]
Case 1	10.0	7.0	4.0	2.0	(616.9, 402.2, 490.4)	(600.0, 400.0, 436.5)	17.04	0	100%
Case 2	15.0	9.0	4.0	2.0	(606, 398, 459)	(600.0, 400.0, 459.0)	6.32	0	100%
Case 3	5.0	6.0	7.0	8.0	(642.2, 420.1, 634.4)	(600.0, 400.0, 315.6)	46.74	0	100%
Case 4	8.0	5.0	6.0	3.0	(624.5, 412.7, 548.2)	(601.4, 404.5, 384.5)	27.60	4.71	82.93%
Case 5	5.5	4.6	3.2	5.2	(637.8, 418, 629.7)	(600.0, 400.0, 322.4)	41.87	0	100%

**Fig. 7.** Visualization of the packed layers produced by the packing algorithm for Case 1, before and after imbalance compensation.

particular its representation before and after imbalance compensation. Each layer is shown together with the contained boxes, annotated by type and mass, and with key indicators, including fill rate and total layer mass, reported above the layer. The pallet geometric center is marked by a blue dot, while the in-plane CoG of each layer is marked by a red dot. The figure highlights that the algorithm allocates heavier layers at the bottom and applies a 180° rotation when beneficial for load balancing. Moreover, the partially filled layer provides additional flexibility, compensating for residual imbalance accumulated in the underlying layers. These results support the effectiveness of the CoG monitoring and control module implemented in *Substep D*.

4.2. Virtual cell simulation and programming

This section validates the proposed framework, detailed in Section 3 and summarized in Fig. 3, on a representative industrial robotic palletizing cell, shown in Fig. 8. The cell layout (top view) comprises a KUKA KR 180-2 PA robot, with a maximum reach of 3200 mm and a rated payload of 180 kg, operated via a KR C4 controller. Under a reference pick-and-place cycle (400 mm lift, 2000 mm travel, 400 mm lowering), the manufacturer reports a productivity of approximately 30 cycles/min. The robot is placed at the center of the cell and is equipped with a vacuum tool for handling the boxes. The target pallet is a standard Europallet (1200 × 800 mm), positioned in front of the robot to receive the stacked boxes. Temporary buffer systems are also included and modeled as two additional pallets of the same size: *Buffer 1* is located behind the robot, while *Buffer 2* is placed laterally. Finally, a linear conveyor is positioned laterally to the robot and serves as the dedicated feeding line, delivering boxes from the warehouse to the cell. Regarding the input box data, the present validation adopts as reference scenario the Test 1 of Section 4.1 (see Table 2), with box masses set according to Case 1 (Table 3).

Since the packing solution is satisfactory (as discussed in Section 4.1), the workflow proceeds to **Step 5** of Fig. 3. A feeding sequence is defined, which enables the buffer system to be either used or bypassed depending on the actual box arrival order. As an initial benchmark to evaluate the cell behavior without buffers, the arrival sequence was set to match the robot pick order. This packing scenario, shown in Fig. 8(a), resulted in an execution time of approximately 68 s. With 26 handled boxes, this corresponds to approximately 22.9 pick-and-place cycles per minute, namely about 76.5% of the nominal robot productivity reported by the manufacturer, indicating a high level of robot utilization and limited idle time. No collisions are detected and all planned motions are executed correctly, which is consistent with the absence of obstacles between pick and place areas. In addition, the adopted way-point targets and clearance offsets enforce repeatable approach and retreat motions, improving robustness and reducing the likelihood of unsafe trajectories. Finally, the total recorded tool travel distance is 240 m. Such travel distance reflects compact routing between pick and place locations and, consequently, reduced overall joint motion, which is beneficial in terms of mechanical wear and cumulative duty over extended operation.

Overall, this configuration is optimal in terms of execution time and provides clean, collision-free paths. However, it shifts complexity to earlier stages, since the boxes must be pre-sequenced during box handling and feeding before entering the cell. In practice, boxes rarely arrive in the desired order, while it could be beneficial to make the boxes of the same type arrive in batches before switching to another type. For this reason, buffer usage becomes necessary. Recent studies on online palletizing and material handling address the management of randomly arriving boxes and their differentiation through additional perception modules, for instance 3D vision [14], weight estimation [15], or barcode reading. Such modules are application-dependent and can be integrated as additional blocks within the proposed workflow.

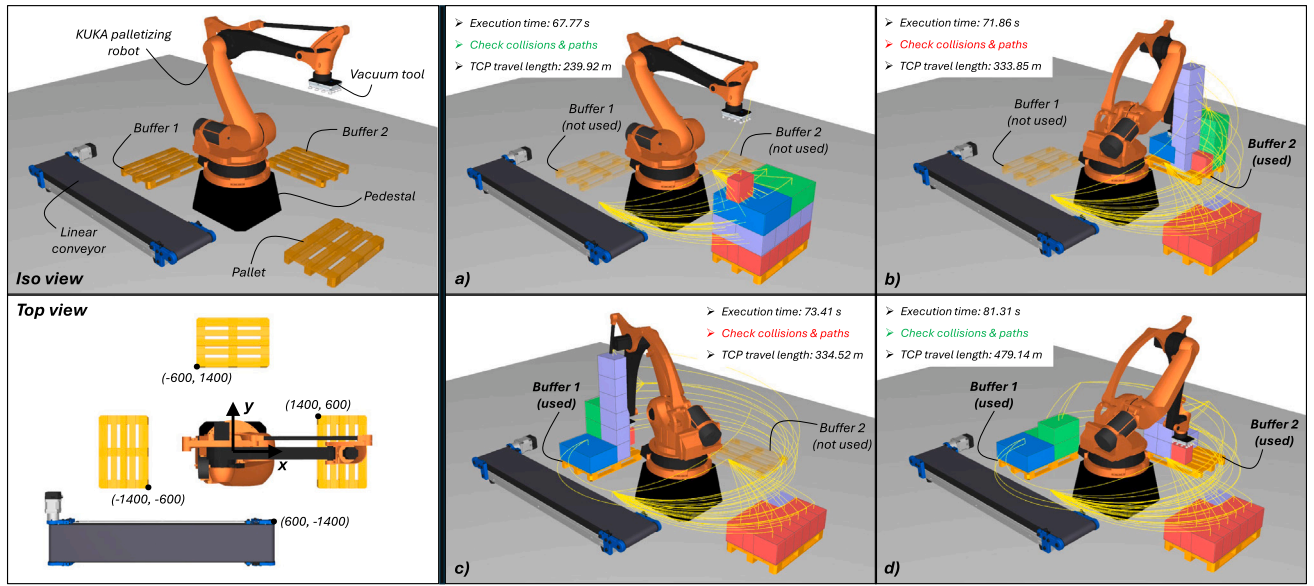


Fig. 8. Virtual robotic palletizing cell. Left: isometric (top) and top (bottom) views of the RoboDK station. Right: simulated buffer utilization configurations, i.e. (a) no buffer, (b) buffer 2 only, (c) buffer 1 only, and (d) both buffers.

To evaluate the system behavior with buffers, additional configurations were tested. The feeding sequence must be modified in **Step 5** (see Section 3.4), while buffer activation and deactivation are handled in **Step 6**, specifically in **Substep D** and **Substep F** (see Section 3.5), where the storage location of each box type is defined by assigning a buffer and an associated space allocation. After applying these changes, the simulation is executed again in **Step 6**, and the results are assessed in **Step 7**. Figs. 8(b), (c), and (d) correspond to the second, third, and fourth tested configurations. Configuration (b), using **Buffer 2** only, increases both cycle time and tool travel length. Moreover, when no additional way-point targets are introduced to constrain the motion, collisions occur, indicating the need for explicit trajectory guidance and/or a revised arrangement within the buffer. Configuration (c), using **Buffer 1** only, exhibits the same issues, with a further increase in execution time and tool travel length due to the buffer position behind the robot. Configuration (d), enabling both buffers, avoids the collision issues observed in the previous cases, at the cost of increased cycle time and tool travel length. Based on these considerations, and given the limited practical feasibility of the no-buffer baseline, configuration (d) was selected as the most promising option for implementation.

4.3. Online deployment on real robot

To run **Step 8** within the proposed Digital Twin oriented workflow and enable bidirectional PC-controller communication for online process execution and monitoring, the general architecture introduced in Section 3.7 has been tailored for the KUKA KR C4 controller adopted in the case study. With reference to Fig. 9a, the implemented approach relies on TCP/IP socket messaging over Ethernet to connect the PC (client) and the KR C4 controller (server). As discussed above, rather than generating and downloading a single monolithic program, the PC streams via Python a sequence of robot instructions that follows the same program flow and operational logic validated in simulation, namely the code established in the Python module of **Step 6** (see Section 3.5) responsible for controlling the virtual robot through the RoboDK API. Since the approach operates at the programming level, the socket communication is used only to exchange command data and acknowledgments, while low-level trajectory generation, inverse kinematics, and closed-loop servo control remain under the controller real-time infrastructure.

On the KR C4 controller side, the link is enabled by a middleware based on *KUKAVARPROXY* [65], running on the Windows partition. This component decodes incoming socket messages and maps their fields to a set of predefined KRL variables. A KRL program runs cyclically on the robot controller and serves as a command execution interface: it continuously reads its internal variables (updated by *KUKAVARPROXY* to activate the received request) and executes the corresponding state within a state machine. Fig. 9b reports an example of instruction streaming considering a *MoveL*. On the Python side, this request is enclosed in the command shown on the left (`kuka.moveL()`). Four different socket packets are then formulated and streamed via TCP/IP, respectively containing the target position and orientation in end effector space with respect to the robot base frame, the desired linear velocity and acceleration, and the command enabling the specific case in the state machine implemented in the KRL program. Each socket packet is structured into seven fields: (1) client ID, (2) total message length (bytes of the body), (3) flag value (0 or 1, respectively for reading or writing a KRL variable), (4) variable name length, (5) variable name, (6) variable value length, and (7) variable value (only for write requests). In terms of communication latency, a single socket packet requires less than 5 ms to be received and processed by the controller within the tested lab prototype, resulting in an overall activation time below 20 ms to trigger the desired *MoveL* command. For commands requiring fewer input data, this interval becomes even shorter. By adopting this communication architecture, any existing KRL variable can be written or read during the robotic process, enabling both command execution and status monitoring. This, in turn, supports a Digital Twin oriented paradigm, in which digital and physical systems are interconnected and can dynamically exchange information during operation, allowing continuous process changes to be assessed in simulation, also based on data collected from the real system, and then deployed to the physical asset without production downtime.

Low-level details on the *KUKAVARPROXY* server operation and TCP/IP message format are available in the online documentation provided in the official repository ([GitHub repository](#)). The validated framework, including the controller connection module with KUKA KR C4, is released as supplementary material.

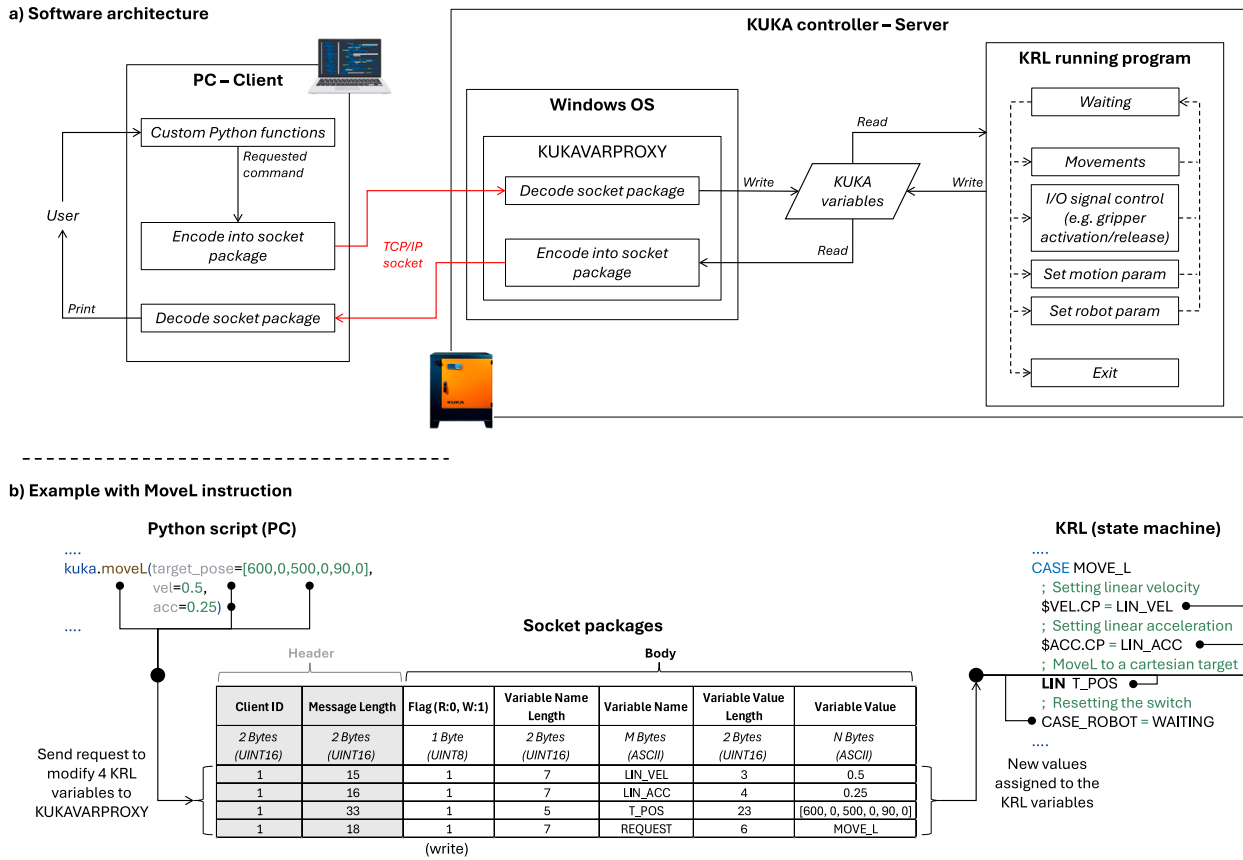


Fig. 9. Implemented online instruction streaming scheme for KUKA KR C4 controllers: (a) client-server software architecture and (b) example of instruction streaming for motion-type robot instructions (MoveL).

4.4. Discussion

The practical relevance of the proposed framework emerges when it is compared with standard industrial workflows, in which packing generation, robot programming, validation, and deployment are handled separately. In conventional practice, the packing solution is often obtained manually or through fixed company-specific rules, with limited flexibility for highly variable orders, and the associated engineering lead time can become significant. With the proposed framework, this stage is reduced to a few minutes when the order is entered through the graphical interface, and can be shortened further when order data is transferred automatically from the upstream information system, since the algorithm execution remains in the order of milliseconds, with a maximum observed value of 33 ms in the tested cases (see Table 2). This reduction is particularly relevant in variable production contexts, where new orders must be translated rapidly into feasible pallet configurations.

A further benefit concerns robot programming and process validation. In conventional approaches, the robot program is typically generated manually and then refined through iterative checks on paths, singularities, and collisions, often directly on the physical system. In the proposed framework, by contrast, instructions are derived automatically from the packing solution and validated in RoboDK before deployment. This makes cycle time estimation directly available, supports rapid comparison of alternative configurations, as illustrated in Section 4.2 by the comparison between no buffer and buffered scenarios, and reduces the need for repeated manual reprogramming and debugging on the real cell. In addition, validated instructions can be streamed directly to the controller with a latency of about 20 ms per command, which does not affect normal cycle execution and is significantly lower than the time typically required to upload a new complete

robot program. Changes in cell configuration, algorithm selection, or input parameters can also be managed within the same workflow, whereas in conventional practice such modifications often lead to prolonged downtime due to manual reprogramming and validation. Overall, the framework improves responsiveness to order variability and setup changes, while reducing engineering effort, downtime, and manual intervention. The reported timings should nevertheless be interpreted as representative of the considered case study, since their absolute values depend on the specific industrial context.

5. Conclusions

This paper presents and validates a novel integrated simulation framework for flexible robotic palletizing planning and validation. Through a structured sequence of steps, the framework generates an actionable optimal pallet configuration and the corresponding robot code, which can be automatically transferred to the robotic palletizing cell within a Digital Twin oriented approach. It aims to support industrial adaptation under changing product mixes by providing simulation modules for evaluating packing layouts, configuring the robotic process, validating the robot program, and enabling direct deployment on physical robots. The framework supports a more efficient and flexible engineering workflow, reducing programming effort and manual intervention while enabling automated validation and rapid evaluation of alternative configurations, thereby facilitating the deployment of optimized solutions in dynamic industrial environments. This enables the effective transfer of existing knowledge into efficient industrial solutions, while supporting designers, especially non-expert users, in selecting and adapting them to specific application scenarios. After a preliminary analysis and comparative benchmark of representative BPP solution strategies, the paper derived and implemented a dedicated

packing algorithm tailored to industrial palletizing instances, selecting existing and validated heuristic components that offer a favorable tradeoff between solution quality and computational cost. The 3D problem is addressed as a sequence of 2D layer packing instances, which are generated, assessed, and iteratively refined to improve both space utilization and load balancing. Within each layer, the routine adopts a Guillotine plus Best Fit scheme to solve the 2D layouts, achieving high filling rates while maintaining computation times in the millisecond range.

Building on this packing core, the proposed simulation workflow is organized as a set of Python modules iteratively executed within a guided step sequence, with explicit checkpoints and fallback actions whenever a requirement is not met. Besides the packing module, the remaining components generate the feeding sequence and automate the robotic simulation in RoboDK through its API, which provides direct access to station elements, motions, and process logic. The configured virtual environment is fully parametric, enabling rapid verification of design variants in terms of both cell configurations and program-level modifications. It supports the automatic generation of robot programs that are first validated in the virtual environment and then deployed to the physical system.

The framework has been validated on a representative industrial case study involving a KUKA palletizing robot equipped with a KR C4 controller, fed by a lateral conveyor and supported by two auxiliary buffer pallets. The simulated scenarios covered multiple product mixes and alternative buffer utilization policies, allowing the impact of input variability and buffering choices to be rapidly assessed within the same virtual workflow. Finally, an instruction streaming mechanism is introduced for the KR C4 controller based on a TCP/IP socket connection over Ethernet. Utilizing a Python client, the robot instructions validated in simulation can be streamed to the real controller with short latency, measured at about 20 ms in the available laboratory prototype, thereby avoiding the generation and download of monolithic robot programs, which inherently introduce downtime and complicate maintenance and updates. Future work is currently directed toward the validation of the proposed framework on fully representative industrial palletizing applications in collaboration with industrial partners, with the aim of assessing the complete workflow on a dedicated industrial setup and deriving more generalizable experimental performance indicators.

CRedit authorship contribution statement

Martina Salami: Formal analysis, Methodology, Software, Validation, Writing – original draft. **Pietro Bilancia:** Conceptualization, Methodology, Supervision, Writing – review & editing. **Marcello Pellicciari:** Conceptualization, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Supplementary material

The research material (scripts and simulation files) related to this work can be freely downloaded from the following Mendeley Data repository <https://data.mendeley.com/datasets/2swjp4vswdw/1>.

Data availability

Data will be made available on request.

References

- [1] E. Oztemel, S. Gursev, Literature review of industry 4.0 and related technologies, *J. Intell. Manuf.* 31 (1) (2020) 127–182.
- [2] S. Asif, M. Bueno, P. Ferreira, P. Anandan, Z. Zhang, Y. Yao, G. Ragunathan, L. Tinkler, M. Sotoodeh-Bahraini, N. Lohse, et al., Rapid and automated configuration of robot manufacturing cells, *Robot. Comput.-Integr. Manuf.* 92 (2025) 102862.
- [3] M. Gadaleta, M. Pellicciari, G. Berselli, Optimization of the energy consumption of industrial robots for automatic code generation, *Robot. Comput.-Integr. Manuf.* 57 (2019) 452–464.
- [4] S. Ferrarini, P. Bilancia, R. Raffaeli, M. Peruzzini, M. Pellicciari, A method for the assessment and compensation of positioning errors in industrial robots, *Robot. Comput.-Integr. Manuf.* 85 (2024) 102622.
- [5] B. Tipary, G. Erdős, Generic development methodology for flexible robotic pick-and-place workcells based on digital twin, *Robot. Comput.-Integr. Manuf.* 71 (2021) 102140.
- [6] A. Daios, N. Kladovasilakis, I. Kostavelis, Mixed palletizing for smart warehouse environments: sustainability review of existing methods, *Sustainability* 16 (3) (2024) 1278.
- [7] S. Elhedhli, F. Gzara, B. Yildiz, Three-dimensional bin packing and mixed-case palletization, *INFORMS J. Optim.* 1 (4) (2019) 323–352.
- [8] G. Pantoja-Benavides, D. Giraldo, A. Montes, A. García, C. Rodríguez, C. Marín, D. Álvarez-Martínez, Comprehensive review of robotized freight packing, *Logistics* 8 (3) (2024) 69.
- [9] R. Morabito, S.R. Morales, J.A. Widmer, Loading optimization of palletized products on trucks, *Transp. Res. Part E: Logist. Transp. Rev.* 36 (4) (2000) 285–296.
- [10] A. Trivella, D. Pisinger, The load-balanced multi-dimensional bin-packing problem, *Comput. Oper. Res.* 74 (2016) 152–164.
- [11] M.R. Garey, D.S. Johnson, et al., A guide to the theory of NP-completeness, *Comput. Intractability* (1990) 37–79.
- [12] J. Valério de Carvalho, Exact solution of bin-packing problems using column generation and branch-and-bound, *Ann. Oper. Res.* 86 (1999) 629–659.
- [13] H.I. Christensen, A. Khan, S. Pokutta, P. Tetali, Approximation and online algorithms for multidimensional bin packing: A survey, *Comput. Sci. Rev.* 24 (2017) 63–79.
- [14] X. Mu, Q. Kan, Y. Jiang, C. Chang, X. Tian, L. Zhou, Y. Zhao, 3D vision robot online packing platform for deep reinforcement learning, *Robot. Comput.-Integr. Manuf.* 94 (2025) 102941.
- [15] S.A. Murdivien, J. Um, Intelligent robot gripper using embedded AI sensor for box re-sequencing system integrated with spatial layout optimization, *Robot. Comput.-Integr. Manuf.* 94 (2025) 102979.
- [16] M. Salami, P. Bilancia, M. Peruzzini, M. Pellicciari, A framework for integrated design of human–robot collaborative assembly workstations, *Robot. Comput.-Integr. Manuf.* 97 (2026) 103108.
- [17] L. Zhang, J. Mei, X. Zhao, J. Gong, Y. Gong, Y. Jiang, J. Sheng, L. Sun, Layout analysis and path planning of a robot palletizing production line, in: 2008 IEEE International Conference on Automation and Logistics, IEEE, 2008, pp. 2420–2425.
- [18] S. Romero, J. Valero, A.V. García, C.F. Rodríguez, A.M. Montes, C. Marín, R. Bolaños, D. Álvarez-Martínez, Trajectory planning for robotic manipulators in automated palletizing: A comprehensive review, *Robotics* 14 (5) (2025).
- [19] Y. Liu, L. Liang, H. Han, S. Zhang, A method of energy-optimal trajectory planning for palletizing robot, *Math. Probl. Eng.* 2017 (1) (2017) 5862457.
- [20] ABB, Palletizing PowerPac for RobotStudio, 2026, <https://www.abb.com/global/en/areas/robotics/products/software/robotstudio-suite>. (Accessed: 10 January 2026).
- [21] RoboDK, Palletizing add-in for RoboDK, 2026, <https://robdk.com/addin/com.robdk.plugin.palletizing>. (Accessed: 10 January 2026).
- [22] C. Munien, A.E. Ezugwu, Metaheuristic algorithms for one-dimensional bin-packing problems: A survey of recent advances and applications, *J. Intell. Syst.* 30 (1) (2021) 636–663.
- [23] M. Francescatto, A.N. Júnior, Two-dimensional bin packing, cutting stock, and open dimension problems: A survey of practical requirements, *Comput. Oper. Res.* (2025) 107209.
- [24] A. Trivellán, D. Pisinger, The load-balanced multi-dimensional bin-packing problem, *Comput. Oper. Res.* Elsevier (2016) (Accessed: May 2025).
- [25] H. Hu, X. Zhang, X. Yan, L. Wang, Y. Xu, Solving a new 3D bin packing problem with deep reinforcement learning method, 2017, arXiv, post=URL <https://arxiv.org/abs/1708.05930>.
- [26] B. Zhang, Y. Yao, H.K. Kan, W. Luo, A GAN-based genetic algorithm for solving the 3D bin packing problem, *Sci. Rep. Springer Nat.* (2024) (Accessed: June, 2025).
- [27] E. Osaba, E. Villar-Rodríguez, S.V. Romero, Benchmark dataset and instance generator for real-world three-dimensional bin packing problems, *Data Brief, Elsevier* (2023) (Accessed: May 2025).
- [28] Y. Harrath, A three-stage layer-based heuristic to solve the 3D bin-packing problem under balancing constraint, *J. King Saud Univ. – Comput. Inf. Sci. Elsevier* (2021) (Accessed: May 2025).

- [29] S. Martello, D. Pisinger, D. Vigo, The three-dimensional bin packing problem, *Oper. Res.* 48 (2) (2000) 256–267.
- [30] A. Lodi, S. Martello, D. Vigo, Heuristic algorithms for the three-dimensional bin packing problem, *European J. Oper. Res.* 141 (2) (2002) 410–420.
- [31] A. Lodi, S. Martello, M. Monaci, Two-dimensional packing problems: A survey, *European J. Oper. Res.* 141 (2) (2002) 241–252.
- [32] S. Martello, D. Vigo, Exact solution of the two-dimensional finite bin packing problem, *Manag. Sci.* 44 (3) (1998) 388–399.
- [33] N.M. Cid-Garcia, Y.A. Rios-Solis, Positions and covering: A two-stage methodology to obtain optimal solutions for the 2d-bin packing problem, *Plos One* 15 (4) (2020) e0229358.
- [34] C. Liu, K. Smith-Miles, T. Wauters, A.M. Costa, Instance space analysis for 2D bin packing mathematical models, *European J. Oper. Res.* (2023) (Accessed: May 2025).
- [35] D. Pisinger, M. Sigurd, Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem, *INFORMS J. Comput.* 19 (1) (2007) 36–51.
- [36] M. Iori, V.L. De Lima, S. Martello, F.K. Miyazawa, M. Monaci, Exact solution techniques for two-dimensional cutting and packing, *European J. Oper. Res.* 289 (2) (2021) 399–415.
- [37] E.A.M. Fernandes, T.F. de Noronha, A.A. Coco, Heuristic solutions for the 2D bin-packing problem with varied size, in: *Brazilian Conference on Intelligent Systems*, Springer, 2024, pp. 65–80.
- [38] G.T. Lam, V.A. Ho, D. Logofătu, Considerations on 2D-bin packing problem: Is the order of placement a relevant factor? *SCITEPRESS– Sci. Technol. Publ.* (2017) (Accessed: May 2025).
- [39] G. Dósa, J. Sgall, First fit bin packing: A tight analysis, in: *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013*, 1998, pp. 538–549.
- [40] C. Kenyon, et al., Best-fit bin-packing with random order., in: *SODA*, vol. 96, 1996, pp. 359–364.
- [41] J. Boyar, L. Epstein, A. Levin, Tight results for next fit and worst fit with resource augmentation, *Theoret. Comput. Sci.* 411 (26–28) (2010) 2572–2580.
- [42] J.O. Berkey, P.Y. Wang, Two-dimensional finite bin-packing algorithms, *J. Oper. Res. Soc.* 38 (5) (1987) 423–429.
- [43] B.S. Baker, J.S. Schwarz, Shelf algorithms for two-dimensional packing problems, *SIAM J. Comput.* 12 (3) (1983) 508–525.
- [44] L. Wei, A. Lim, W. Zhu, A skyline-based heuristic for the 2d rectangular strip packing problem, in: *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, Springer, 2011, pp. 286–295.
- [45] J. Jylänki, A thousand ways to pack the bin—a practical approach to two-dimensional rectangle bin packing, 2010, Retrived from <http://clb.demon.fi/files/RectangleBinPack.pdf>.
- [46] N. Ntene, J.H. Van Vuuren, A survey and comparison of heuristics for the 2D oriented on-line strip packing problem, *ORiON* 24 (2) (2008) 157–183.
- [47] A. Alvim, F. Glover, C. Ribeiro, D. Aloise, Local search for the bin packing problem, in: *Extended Abstracts of the 3rd Metaheuristics International Conference*, 1999, pp. 7–12.
- [48] J. Gardeyn, T. Wauters, A goal-driven ruin and recreate heuristic for the 2D variable-sized bin packing problem with guillotine constraints, *European J. Oper. Res.* 301 (2) (2022) 432–444.
- [49] E. Hopper, B.C. Turton, A review of the application of meta-heuristic algorithms to 2D strip packing problems, *Artif. Intell. Rev.* 16 (4) (2001) 257–300.
- [50] G.T. Lam, V.A. Ho, D. Logofătu, C. Badica, Considerations on using genetic algorithms for the 2D bin packing problem: a general model and detected difficulties, 2017, *IEEE* (Accessed: May 2025).
- [51] J. Junkermeier, A genetic algorithm for the bin packing problem, *Evol. Comput. St. Cloud State Univ. Spring* (2015).
- [52] S.C. Leung, D. Zhang, C. Zhou, T. Wu, A hybrid simulated annealing metaheuristic algorithm for the two-dimensional knapsack packing problem, *Comput. Oper. Res.* 39 (1) (2012) 64–73.
- [53] R. Rao, S. Iyengar, Bin-packing by simulated annealing, *Comput. Math. Appl.* 27 (5) (1994) 71–82.
- [54] D. Liu, K.C. Tan, S. Huang, C.K. Goh, W.K. Ho, On solving multiobjective bin packing problems using evolutionary particle swarm optimization, *European J. Oper. Res.* 190 (2) (2008) 357–382.
- [55] H.-C. Lu, Y.-S. Wan, Y.-T. Shueh, An efficient ant colony optimization for two-dimensional bin packing problem with defect issue, in: *The Fourth International Conference on Informatics & Applications (ICIA2015)*, 2015, p. 83.
- [56] A. Lodi, S. Martello, D. Vigo, TSpack: a unified tabu search code for multi-dimensional bin packing problems, *Ann. Oper. Res.* 131 (1) (2004) 203–213.
- [57] R. Alvarez-Valdés, F. Parreño, J.M. Tamarit, A tabu search algorithm for a two-dimensional non-guillotine cutting problem, *European J. Oper. Res.* 183 (3) (2007) 1167–1182.
- [58] T.G. Crainic, G. Perboli, R. Tadei, Extreme point-based heuristics for three-dimensional bin packing, *Inform. J. Comput.* 20 (3) (2008) 368–384.
- [59] K. Kang, I. Moon, H. Wang, A hybrid genetic algorithm with a new packing strategy for the three-dimensional bin packing problem, *Appl. Math. Comput.* 219 (3) (2012) 1287–1299.
- [60] Z. Jin, T. Ito, K. Ohno, The three-dimensional bin packing problem and its practical algorithm, *JSME Int. J. Ser. C Mech. Syst. Mach. Elements Manuf.* 46 (1) (2003) 60–66.
- [61] International Organization for Standardization, ISO 3676:2012 – packaging — Complete, filled transport packages and unit loads — Unit load dimensions, 2012, <https://www.iso.org/standard/50989.html>. (Accessed: May 2025).
- [62] RoboDK, RoboDK Python API documentation, 2025, URL <https://robodk.com/doc/it/RoboDK-API-Python-API.html>. (Accessed: October 2025).
- [63] F. Mo, H.U. Rehman, J.C. Chaplin, D. Sanderson, S. Ratchev, Digital twin-based self-learning decision-making framework for industrial robots in manufacturing, *Int. J. Adv. Manuf. Technol.* (2025) 1–20.
- [64] J.P.C. de Souza, C.M. Costa, L.F. Rocha, R. Arrais, A.P. Moreira, E.S. Pires, J. Boaventura-Cunha, Reconfigurable grasp planning pipeline with grasp synthesis and selection applied to picking operations in aerospace factories, *Robot. Comput.-Integr. Manuf.* 67 (2021) 102032.
- [65] F. Sanfilippo, L.I. Hatledal, H. Zhang, M. Fago, K.Y. Pettersen, Controlling kuka industrial robots: Flexible communication interface JOpenShowVar, *IEEE Robot. Autom. Mag.* 22 (4) (2015) 96–109.