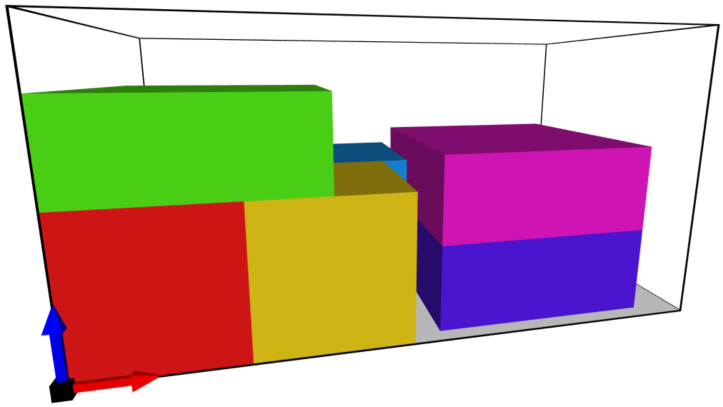
Solution Specification for Loading Problems

May 23, 2018



1 Introduction

This document describes the structure of the solutions that can be handed in on the website. As was the case with the instances, solutions can be in any of the JSON, XML, or YAML file formats, with minimal differences in formatting. Simple examples of solutions in all file formats are given in Section 2. Section 3 contains a complete description of the solution structure. Solution files are textual representations of `ThreeDsolution` Python objects, and solution validation is implemented as a method of this class of objects. Solution validation consists of multiple parts: correctness of formatting and consistency with the corresponding instance (which are explained in the document 'Tools for Loading Problems'), validation of constraints, and the evaluation of objective values. Section 4 describes how each constraint and objective is modeled, and will be continually updated when new objectives and/or constraints become available.

Figure 1: A small JSON example

2 File Formats

Solutions can be specified in three different file formats: JSON, XML, and YAML. The structure used in all formats is the same, and will be explained in Section **??**. There are only a few types of constructs in these files:

- An object with properties, e.g., **solution**, **layout**, **container**, **placement**, etc.
- A property can be of different types:
  - A value, e.g., **id**, **kindid**, **palletid**, **boxid**, **itemid**, **position**, **orientation**, etc.
  - A group of properties, e.g., **description**, **layout**, **placement**, **loadingspace**, etc.
  - A list of similar objects, e.g., **containers**, **pallets**, **boxes**, **placements**, etc.

2.1 JSON

JSON is a lightweight data-interchange format. The mapping between the solution structure and JSON is very straightforward:

- objects are objects
- properties are pairs
- a group of properties is an object
- a list is a list

The biggest difference with the other file formats is that the objects in a list do not have an own name, e.g., **container**, **pallet**, **box**, **placement**, and **loadingspace**.

2.2 XML

XML is an eXtensible Markup Language. The mapping between the solution structure and XML is again straightforward:

- objects are tags containing tags with different names
- properties are tags containing text
- a group of properties is a tag containing tags with different names
- a list is a tag containing tags with that same name

Exceptions to this are the properties **id**, **kindid**, **palletid**, **boxid**, **itemid**, and **quantity** which are provided as attributes.

2.3 YAML

YAML is a human friendly data format. The mapping between the solution structure and YAML is once more straightforward:

- objects are represented by a mapping
- properties are an entry in such a mapping
- a group of properties is also represented by a mapping
- a list is represented by a sequence of single entry mappings with equal names

```
<solution>
    <description>
        <set>Example</set>
        <name>001</name>
    </description>
    <layout>
        <containers>
            <container id="1" kindid="1">
                <loadingspaces>
                    <loadingspace id="1">
                        <placements>
                            <placement itemid="1" id="1">
                                <position>0,0,0</position>
                                <orientation>LWH</orientation>
                            </placement>
                            <placement itemid="2" id="3">
                                <position>1000,100,0</position>
                                <orientation>LWH</orientation>
                            </placement>
                            <placement itemid="1" id="4">
                                <position>0,0,500</position>
                                <orientation>LWH</orientation>
                            </placement>
                            <placement itemid="1" id="5">
                                <position>1000,100,500</position>
                                <orientation>LWH</orientation>
                            </placement>
                        </placements>
                    </loadingspace>
                </loadingspaces>
            </container>
        </containers>
        <pallets />
        <boxes />
        <unplaced>
                <placement id="6" itemid="2" quantity="3" />
        </unplaced>
    </layout>
</solution>
```

Figure 2: A small XML example

Figure 3: A small YAML example

3 Solution Structure

The structure of the solution is similar to that of the instance. The verification of solutions requires an accompanying instance file, and so only a small amount of new data is required to be specified in the solution (namely where each item, box, and pallet is placed). It is, however, possible to copy the data, objectives, and constraints objects from the instance file into the solution file. These objects will not be used for validation, except that an extra check is performed to ensure that the instance specifications match one another.

3.1 Solution

desciption information about the solution associating it with an instance with the same set and name.

required

Type: group

Fields:

set name of a set of instances.

required

Type: string

name name of the instance.

required

Type: string

layout information about the physical placement of each pallet, box, and item within loadingspaces of containers, pallets, and boxes (or whether they are unplaced). Each item must be accounted for in the sense that for each item kind the occurences within the solution file must be equal to the quantity attribute from the corresponding instance file.

required

Type: group

Fields:

containers multiple containers with their associated placements. The total number of appearing containers of each kind may not exceed the quantity specified in the instance file. The description of a container can be found in Section 3.2.

required

Type: list of containers

pallets multiple pallets with their associated placements. The total number of appearing pallets of each kind may not exceed the quantity specified in the instance file. The description of a pallet can be found in Section 3.3.

required

Type: list of pallets

boxes multiple boxes with their associated placements. The total number of appearing boxes of each kind may not exceed the quantity specified in the instance file. The description of a box can be found in Section 3.4.

required

Type: list of boxes

unplaced all pallets, boxes, and items need to be either placed in a container or they need to be unplaced. There are some minor alterations to the unplaced placement object, they are explained in Section 3.5.

required

Type: list of placements

3.2 Container

id unique id among containers.

required

Type: int

kindid kind id of the container, corresponding to the id of a container kind in the instance file.

required

Type: int

loadingspaces list of loadingspaces where any placement type is allowed.

required

Type: list of loadingspaces

3.3 Pallet

id unique id among pallets.

required

Type: int

kindid kind id of the pallet, corresponding to the id of a pallet kind in the instance file.

required

Type: int

loadingspaces list of loadingspaces where pallets are disallowed.

required

Type: list of loadingspaces

3.4 Box

id unique id among boxes.

required

Type: int

kindid kind id of the box, corresponding to the id of a box kind in the instance file.

required

Type: int

loadingspaces list of loadingspaces where only items are allowed.

required

Type: list of loadingspaces

3.5 Unplaced

placements placements where items can have an additional quantity property.

required

Type: list of placements

3.6 Loadingspace

id unique id among loadingspaces in the containing space. Must correspond to the definition in the instance file.

required

Type: string

placements *required*

Type: list of placements

3.7 Placement

id unique id among placements (pallets, boxes, and items).

required

Type: int

palletid exactly one of palletid, boxid, and itemid must be defined (and uniquely so). If defined, it corresponds to the id attribute of a pallet. Also, if defined, the placement may not occur inside a pallet or box.

optional

Type: int

boxid exactly one of palletid, boxid, and itemid must be defined (and uniquely so). If defined, it corresponds to the id attribute of a box. Also, if defined, the placement may not occur inside a box.

optional

Type: int

itemid exactly one of palletid, boxid, and itemid must be defined (and uniquely so). If defined, it corresponds to the id attribute of an item.

optional

Type: int

position x, y, and z coordinates of the position of the placement in the containing loadingspace.

required

Type: int,int,int

orientation string indicating the orientation of the placement, see Section 4.1 for more information.

required

Type: string

quantity may only be set when placement is not placed, in this case it indicates how many placements of the given type are not placed.

optional

Type: int

color a color code that is used for this specific placement when the solution viewer is invoked with the file coloring option. The string is formatted as either: #rrggbb; or #rrggbbaa;, where rr, gg, bb, and aa respectively indicate the hexadecimal red, green, blue and alpha values.

optional

Type: string

4 Constraints and Objectives

In this section we will explain some of the modeling choices made with regards to the constraints and objectives, but first we will give a short general description of how we consider constraints and objectives.

Constraints are traditionally considered hard, and as such can not be violated. However we consider each problem set to in fact describe a family of possible problem sets, where each member of the family differs in what set of constraints are allowed/ignored. Solutions that ignore certain constraints will be flagged as such on the webpage, and may only compete with other solutions that consider (at least) their specific subset of constraints.

The final objective value is a list of real numbers, and comparison is done using the lexicographic ordering. The final objective value is aggregated from separate objectives, which we will describe below. Each objective has an associated priority and weight. The priority indicates its place in the lexicographic ordering. Priorities do not have to be unique, when there is overlap the weighting factors of the corresponding objectives are used to combine their values into a single value. In this manner we support both traditional ways of multi-objective optimization. Lower objective values are considered better, and so for objectives where the opposite is true, we flip the sign of the objective value (most notably of the `item_count` objective).

Solution comparison is done in the following manner: given a subset of constraints, show only the solutions that satisfy at least those constraints, and order them first by objective value, next by the number of constraints met, and finally by date.

4.1 `orientation` constraint

In many cases certain orientations of placements are not allowed. A few good examples of this are 'this side up'-boxes, and long boxes that may not rest on a specific face (typically we would want a long side of said box to support it, otherwise it might get damaged). In short: we constrain each placement to have an orientation that belongs to a predefined subset of all orientations. We have devised a notation for orientations that allows for describing all 24 orientations of a cuboid object placed orthogonally in a cuboid region. Note that in most cases 6 orientations will suffice, but for correct calculation of, for example, the center of mass or the axle weight distributions we need all 24 orientations once we load placements into boxes or pallets and then finally into containers (we want to know exactly where inside the box or pallet the placement resides). When the more intricate 24 orientation system is not needed, aliases are available to work in the more simple 6 orientation system.

We use three character strings of text to describe orientations. These strings are built up in the following way: each of L, W, and H must occur exactly once in either upper or lower case and the characters may occur in different orders. When a letter occurs as upper case it indicates that the local axis of the placement points in the same direction as the global axis corresponding to.

The order of the letters indicates the correspondence of local axes to global axes: the orientations LWH, Lwh, lWh, and lwH all indicate that the local axes are aligned with the global axes. The permuted orientations WLh, WlH, wLH, and wlh all indicate that the local Width component now aligns with the global Length component (and similar for the local Length component).

Consider the following example: WlH is the orientation where the local Width component now points in the direction of the positive global L component, the local L component points in the direction of the negative global W component, and the local H component points in the direction of the positive global H component. In short: WlH is the orientation obtained by rotating the standard orientation LWH 90 degrees clockwise around the H axis. We may also refer to this specific orientation by its alias: WLH. All aliases are as follows:

$$LWH \Rightarrow LWH \qquad LHW \Rightarrow LhW$$
$$WHL \Rightarrow WHL \qquad WLH \Rightarrow WlH$$
$$HLW \Rightarrow HLW \qquad HWL \Rightarrow HwL$$

See Figure 4 for a visualization of all possible orientations. The aliases are as follows:

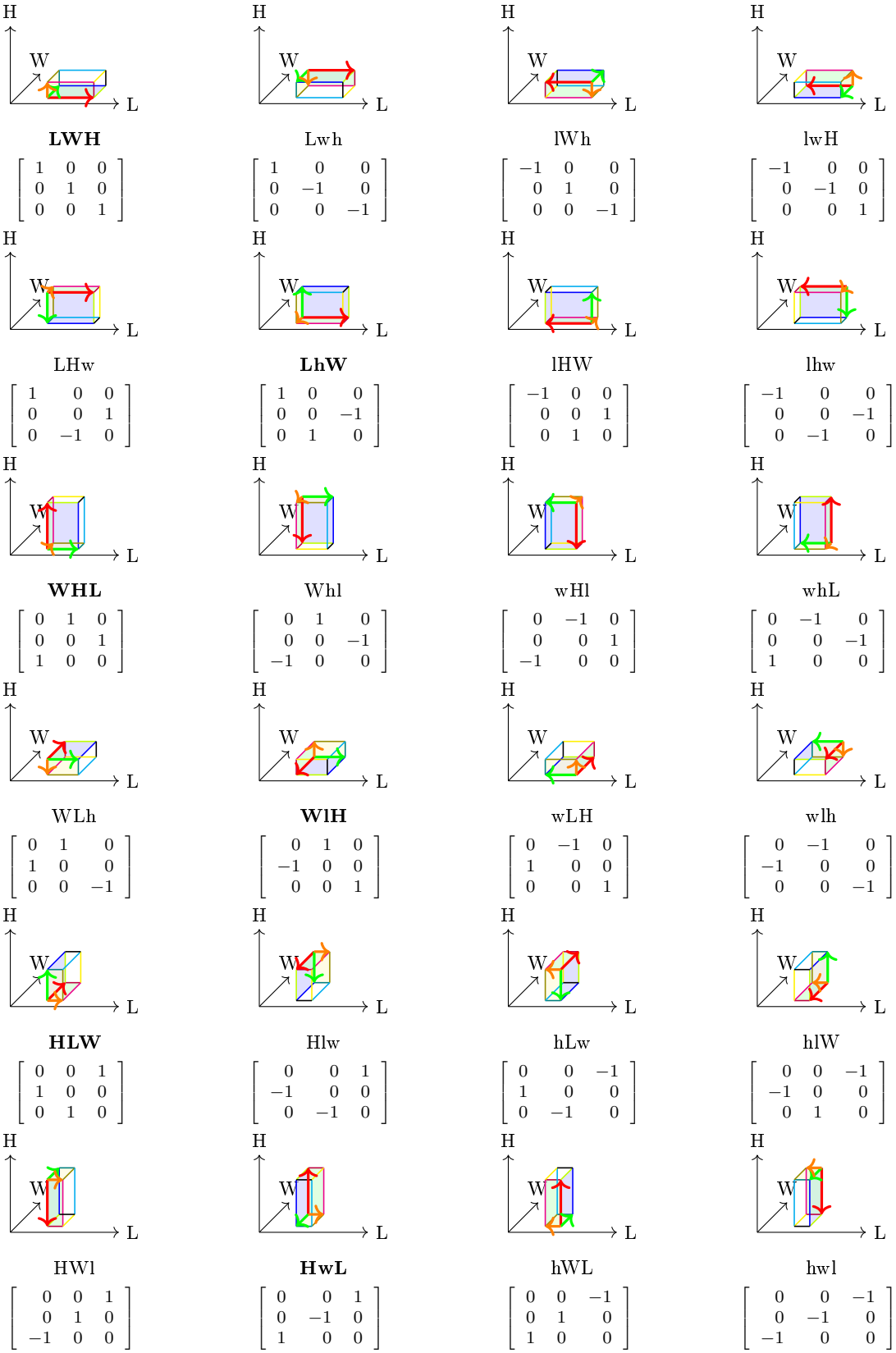


Figure 4: The visual representation, our notation, and the corresponding linear transform of all 24 orientations of orthogonally aligned cuboids. Each row contains all four orientations with identical bounding box dimensions, and so each row corresponds to one of the six simplified orientations. The simplified orientations can be used by referring to the alias of the row. The orientations written in bold are those that are implicitly used when using the alias. The alias of each row is simply the upper case version of any of the orientations of that row.

4.2 Spatial Representation

The used spatial representation is akin to that of Ngoi[2]. However we use a single matrix to keep track of what height levels are attained at each position of a loading space, aswell as the stacking order. The matrix stores tuples containing the current height values and stacking order of a rectilinear partition of the topview of a loading space. The partitioning is updated each time a placement is introduced into the loading space. The matrix can be used to check for a given set of n placements (items, boxes, or pallets) whether they overlap, lie within the loading space, whether each placement has enough support, and whether certain stackability constraints hold. New placements should be inserted in order of increasing z-value. In addition to this matrix, we explicitly store the x and y-partitionings of the loading space, corresponding to the places where the matrix cells start and end.

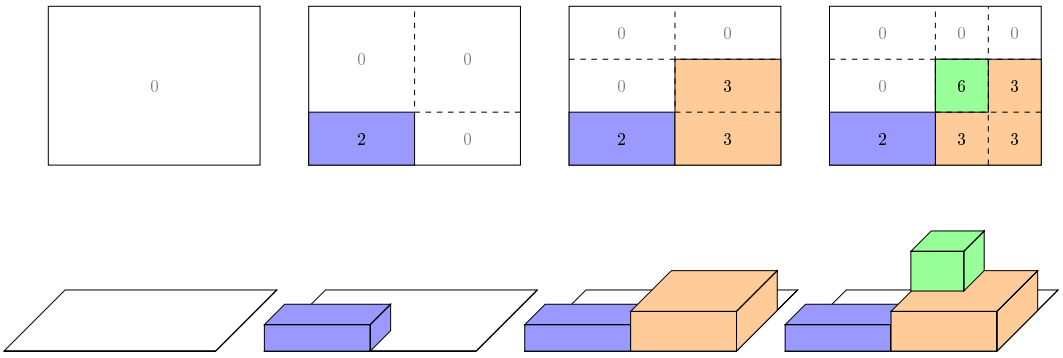


Figure 5: Height matrices of three consecutive placements in an initially empty loading space

In Figure 5 three placements are introduced into a loading space that is empty initially. Each time a placement is introduced, the loading space is partitioned in the x and y-directions at the corner points of the placement. The number in each cell indicates the height level of the highest placement in that cell.

Assume the placements are processed in increasing z-order, with an arbitrary tie-breaking procedure for equal z-values. Each time we want to place a cuboid orthogonally-aligned inside the loading space we go through the following steps:

- if any of the corner points of the cuboid do not fall on a crossing of the current partition, add new partitions for those corner points, and copy columns and rows accordingly to effectively refine the grid without losing the information of current placements;
- compute the set $\mathcal{I}$ that contains precisely those pairs of indices of the Ngoi matrix wherein the new cuboid is to be placed;
- *to determine whether the cuboid is supported:* for each pair of indices in $\mathcal{I}$, check whether the current z-level equals the minimum z-level of the new cuboid;
- *to determine whether the cuboid does not intersect any of the other cuboids:* for each pair of indices in $\mathcal{I}$, check whether the current z-level is less than or equal to the minimum z-level of the new cuboid.

4.2.1 `support` constraint

Using the method outlined above, we can determine what percentage of the bottom side of a placement is supported. The `support` constraint then simply states a lower bound for the support per palletkind, boxkind, and itemkind. Whenever this bound is exactly equal to one we refer to it as full support, otherwise we refer to it as partial support. Note that we have no guarantee of stability in the presence of partial support, though for practical purposes it might still be good to do so; slight overhang should not be penalized by ruling out solutions altogether, as this might exclude a lot of otherwise acceptable solutions.

4.2.2 `exact_stacking` constraint

A very simple constraint on the stacking of placements is to require them to stack 'exactly', which is to say that every placement that is not placed on the ground is placed on top of exactly one other

placement; the bottom and top faces overlap exactly. This constraint is checked by checking all stacking orders of the final Ngoi structure.

4.2.3 `readable_item_labels` constraint

In some cases pallets should be packed such that item labels are readable from the outside. Different modelling choices are possible to achieve this desired effect. We have chosen for a robust approach; we consider item labels to be visible only when the cubic region obtained from extruding the labeled face outward is completely empty. Verifying whether this constraint holds can then be done relatively easily by using the spatial representation from Section 4.2. We add a dummy placement for each label that represents the region for that label that should be empty. Next we simply check for overlap of a dummy and a non-dummy placement.

4.3 `must_place` constraint

In presence of the `must_place` constraint, specific placements need to be placed. The placements that need to be placed have their `place` attribute set to `True` in their instance file. This constraint typically occurs in problems where the goal is to minimize the number of containers or pallets, to avoid trivial solutions.

4.4 Counting objectives

Very often simple counting objectives are good indicators of efficient or cost effective loads. We support the following objectives: `container_count`, `pallet_count`, `box_count`, and `item_count`. These simply count, respectively, how many `container`, `pallet`, `box`, and `item` objects occur in the solution file. The `item_count` behaves differently from the others in that its sign is flipped, and that occurrences from within `unplaced` are not counted.

4.5 Axle Weight Distribution

The axle weight objective and constraint require the calculation of the axle weight distributions. We consider a two axle model, where we have the following variables: position of front axle x_f, and rear axle x_r (with $x_f < x_r$), and their respective axle weights W_f, W_r, the total weight W, and the lengthwise component of the center of mass x. The following formulas then hold for the axle weights:

$$W_f = W\tfrac{x_r - x}{x_r - x_f},$$
$$W_r = W\tfrac{x - x_f}{x_r - x_f}.$$

4.5.1 `axle_weight` constraint

The `axle_weight` constraint is simply the hard constraint that: $W_f \in [W_{f\,\min}, W_{f\,\max}]$ for constants $W_{f\,\min} \leq W_{f\,\max}$, and similarly $W_r \in [W_{r\,\min}, W_{r\,\max}]$ for constants $W_{r\,\min} \leq W_{r\,\max}$.

4.5.2 `axle_weight` objective

From the previous constraints we can deduce the requirement $x \in [C_{\min}, C_{\max}]$, where:

$$C_{\min} = \max\left\{x_f + \tfrac{W_{r\,\min}}{W}(x_r - x_f), x_r - \tfrac{W_{f\,\max}}{W}(x_r - x_f)\right\},$$
$$C_{\max} = \min\left\{x_f + \tfrac{W_{r\,\max}}{W}(x_r - x_f), x_r - \tfrac{W_{f\,\min}}{W}(x_r - x_f)\right\},$$

and we construct the `axle_weight` objective by incurring a linear penalty for lying outside this interval. When $[C_{\min}, C_{\max}]$ is not a valid interval, the objective reduces to minimizing the distance of the center of gravity to $\frac{1}{2}(C_{\min} + C_{\max})$. The objective takes the following closed form (where $x^+ = x$ (when $x \geq 0$), and $x^+ = 0$ (otherwise)):

$$x \mapsto \left(\left|x - \tfrac{1}{2}(C_{\min} + C_{\max})\right| - \tfrac{1}{2}(C_{\max} - C_{\min})\right)^+.$$

A visual representation is presented in Figure 6.

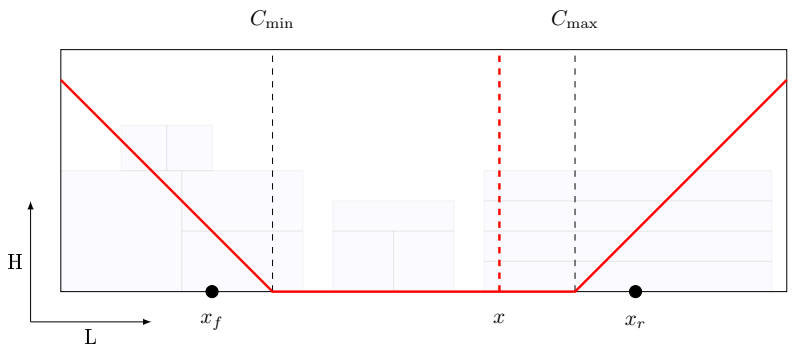


Figure 6: Axle weight violation as a function of the center of mass in two axle model (axle placements and parameter choices are not realistic for this diagram)

References

[1] ORTEC. (2018). ORTEC Scientific Benchmarks - Loadbuilding. *https://benchmarks.ortec.com/*.

[2] Ngoi, B. K. A., & Whybrew, K. (1993). A fast spatial representation method (applied to fixture design). *The International Journal of Advanced Manufacturing Technology, 8*(2), 71-77.