BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING

FAKULTA STROJNÍHO INŽENÝRSTVÍ

INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

ÚSTAV AUTOMATIZACE A INFORMATIKY

SOFTWARE FOR EFFICIENT USE OF MATERIAL IN 2D MACHINING

SOFTWARE PRO EFEKTIVNÍ VYUŽITÍ MATERIÁLU PŘI 2D OBRÁBĚNÍ

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. ONDŘEJ ŠVANDA
SUPERVISOR VEDOUCÍ PRÁCE	Ing. et Ing. STANISLAV LANG, Ph.D.

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

Ústav:	Ústav automatizace a informatiky
Student:	**Bc. Ondřej Švanda**
Studijní program:	Strojní inženýrství
Studijní obor:	Aplikovaná informatika a řízení
Vedoucí práce:	**Ing. et Ing. Stanislav Lang, Ph.D.**
Akademický rok:	2019/20

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Software pro efektivní využití materiálu při 2D obrábění

Stručná charakteristika problematiky úkolu:

Práce je zaměřena na problematiku efektivního využití materiálu při vytváření obrobků z plechu (technologií pálení laserem či plasmou, vyjiskřování či frézování). Snahou je docílit vhodného rozložení výrobních plánů tak, aby využití zdrojového plechu bylo co nejvyšší. Uvažován přitom má být i plech, který již byl použit a nachází se v něm zakázané oblasti, tj. díry po výrobě předchozích obrobků. Problematikou využití zbytků plechu se má zabývat tato práce.

Cíle diplomové práce:

Proveďte stručnou rešerši v oblasti 2D obrábění.
Nastudujte metody SW reprezentace objektů (v 2D prostoru).
Proveďte průzkum dostupných nástrojů řešících uvedenou problematiku.
Vyberte či navrhněte metodu pro umisťování dvourozměrného objektu od ohraničené plochy zahrnující zakázané oblasti.
Zvolte si vhodný framework pro tvorbu vlastní aplikace, implementujte vybrané algoritmy.
Zhodnoťte dosažené výsledky a diskutujte možnosti využití vytvořeného softwaru, případně možnosti dalšího rozšíření jeho funkcionality.

Seznam doporučené literatury:

ERICSON, Christer. Real-time collision detection. Boston: Elsevier, 2005. ISBN 978-155-8607-323.

YANG, Fangkai. Collision Detection between Dynamic Rigid Objects and Static Displacement Mapped Surfaces in Computer Games [online]. Stockholm: Royal Institute of Technology School of Engineering Sciences, 2015 [cit. 2019-10-26]. Dostupné z: http://www.diva-portal.org/smash/get/diva2:839269/FULLTEXT01.pdf

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2019/20

V Brně, dne

L. S.

doc. Ing. Radomil Matoušek, Ph.D. ředitel ústavu	doc. Ing. Jaroslav Katolický, Ph.D. děkan fakulty

Summary

This thesis aims to familiarize the reader with 2D CNC machining and various methods of describing and manipulating planar geometry. Later, the Nesting problem is introduced, and different methods of approaching its solution are described. The practical part covers an implementation of a custom nesting software, that attempts to place selected irregular shapes into an irregular container in an effective way. Finally, an evaluation of said software is conducted and its possible practical applications are also mentioned.

Abstrakt

Tato práce si klade za cíl obeznámit čtenáře s 2D CNC obráběním a rozličnými metodami popisu a manipulace s dvojrozměrnými objekty. Poté je představen tzv. Nesting problém a jsou popsány různé metody přístupu k jeho vyřešení. Praktická část pokrývá implementaci vlastního nesting softwaru, jenž je schopný efektivně rozmístit nepravidelné tvary do nepravidelné ohraničené plochy. Na závěr je provedeno zhodnocení řečeného softwaru a také je zmíněno jeho možné využití v praxi.

Keywords

2D machining, G-Code, material utilisation, Nesting problem, optimisation, spatial analysis, No-Fit Polygon

Klíčová slova

2D obrábění, G-Code, využití materiálu, Nesting problem, optimalizace, prostorová analýza, No-Fit Polygon

ŠVANDA, O. *Software for efficient use of material in 2D machining.* Brno University of Technology, Faculty of Mechanical Engineering, 2020. 61 p. Supervisor Ing. et Ing. Stanislav Lang, Ph.D.

Affidavit
I hereby declare that this Master's thesis is my original work and has been written by me under the guidance of my supervisor Ing. et Ing. Stanislav Lang, Ph.D., using the resources listed in the bibliography.

Brno, 26. 6. 2020 Bc. Ondřej Švanda

Acknowledgements
I would like to express my sincerest thanks to my supervisor Ing. Stanislav Lang for his guidance and helpful remarks on the topic, and to both him and his colleague Ing. Kamil Růžička from the company B+R for the opportunity to work on this interesting task.

I would also like to thank my girlfriend, Veronika Baršová, for her much needed emotional support during these times.

Bc. Ondřej Švanda

Contents

1. Introduction

An effective use of material is a big topic in any industry. More so in CNC machining, where computers control the manufacturing processes of a large quantity of components. It is desirable to minimise the amount of unused source material, because, naturally, the costs add up in the long run.

The aim of this Master's thesis is to create such program, that would enable re-use of material, from which parts could has been already cut out, using 2D CNC machining processes.

In the theoretical part will be covered a basic introduction into planar machining, and ways to represent two-dimensional objects in software. It will be further discussed how these representations can be used for a spatial analysis and how they can be applied in various methods.

Several methods and approaches to the so called *Nesting problem* will be explored and some existing solutions for this problem will be listed.

The practical part will describe an approach to an implementation of said software, using the programming language Python. A thorough evaluation of the program's performance will be carried out and its strengths and shortcomings will be discussed, along with possible enhancements and practical applications.

2. 2D Machining

The term machining refers to any process where a piece of a raw material (called a workpiece) is formed into the desired shape by a controlled gradual removal of material. These machining processes are also collectively referred to as *subtractive manufacturing*, as opposed to forging or casting, where the source material is deformed into the target shape, or *additive manufacturing* as in 3D printers.

Despite many machining power tools still being manually controlled by the operator, in the modern times, most of them are controlled programmatically by a computer (Computer Numerical Control – CNC). The first NC machines were built in the 1940s and the program was "stored" on a punched tape, that was fed into the controller. [34] With the rise of digital computers and microcontrollers, true CNC machines soon emerged. Most CNC machines today are programmed by a language called G-Code (see Chapter 4).

While CNC machines can have an arbitrary number of axes of motion, this thesis is only concerned about 2D (or planar) machining, where just the X and Y dimensions are used. There are multiple types of 2D machining tools but the essential principle under which they operate is as follows: The source workpiece[1] is laid flat on the machine bed, and a cutting tool head that moves relative[2] to the bed frame cuts out pieces of the material. Due to the nature of this method, it is apparent that there is a need to arrange the cutting plan in such way, that the residual material after the parts are cut out is minimised, which is the aim of this thesis.

In the following sections are described and compared different machines used for 2D machining. [40]

2.1. CNC Routing

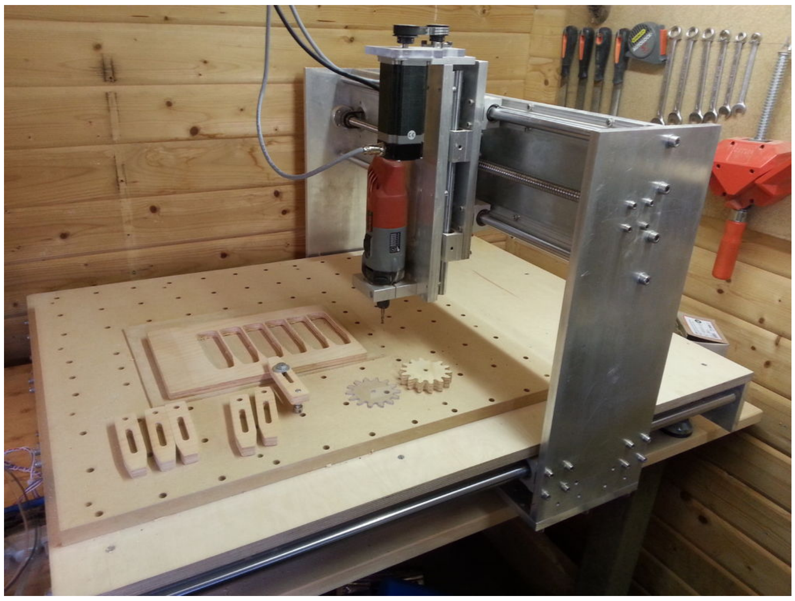


Figure 2.1: CNC wood router [5]

[1]Usually a flat sheet of material like metal or wood

[2]Either the head or the bed can move, or both

While the CNC router is essentially a 3-axis machining tool, it is often used to cut flat shapes, so it is listed here. The machine consists of a moveable head that holds a spindle on which various routing or milling bits are to be attached. The head moves in the horizontal X-Y plane, while the spindle can be lifted or lowered on the Z axis. It can be used to cut and engrave various hard materials like wood, plastics or some composites, and depending on the strength of the device, even aluminium or steel.

2.2. Laser Cutting

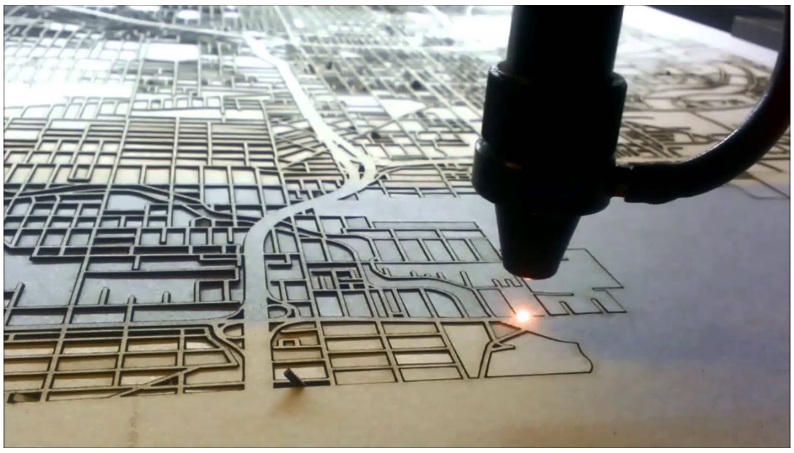


Figure 2.2: Laser cutting a city map [11]

Laser cutting is one of the most precise methods used to cut materials. It works by focusing a high powered laser beam onto the workpiece, which almost instantly melts or evaporates the material in the affected area. As the width of the beam is generally less than half a millimetre, it is used where tight tolerances are required. The disadvantage of this method is that for a clean cut, the material must be placed within the depth of focus of the laser, which is generally relatively small, making this method unsuitable for thicker materials.

2.3. Plasma cutting

This technique requires a electrically conductive material as the work piece, so it is best suited to cut metals like steel, aluminium, brass or copper. It works by creating an electrical arc from the cutting head to the workpiece, completing an electrical circuit with a grounding clamp attached to the metal. This arc ionizes the superheated gas blown under a high pressure from a focused nozzle (creating what is called a *plasma*) towards the work piece. As electrical current flows through the plasma, it delivers enough energy to melt through the workpiece, while the molten metal is being blown away. While not as precise as laser cutting, it can be used to cut through up to 15 cm thick metals. Despite being the least precise method, it is the fastest one with respect to workpiece thickness.

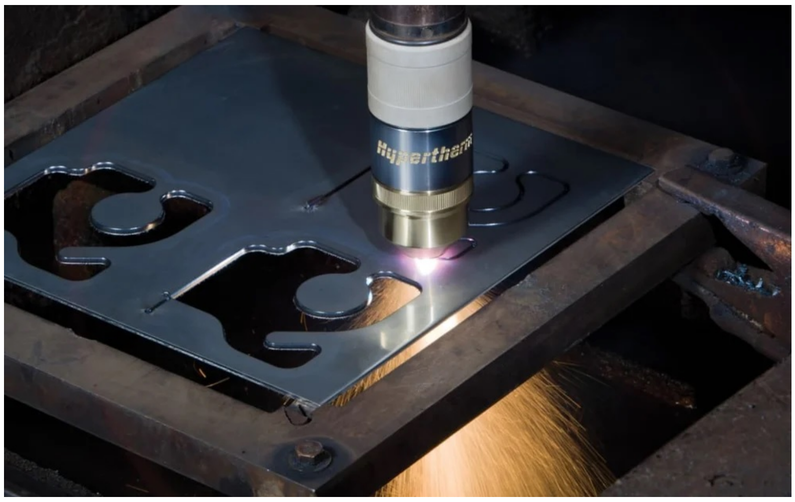


Figure 2.3: Plasma cutter in operation [1]

2.4. Water Jet Cutting

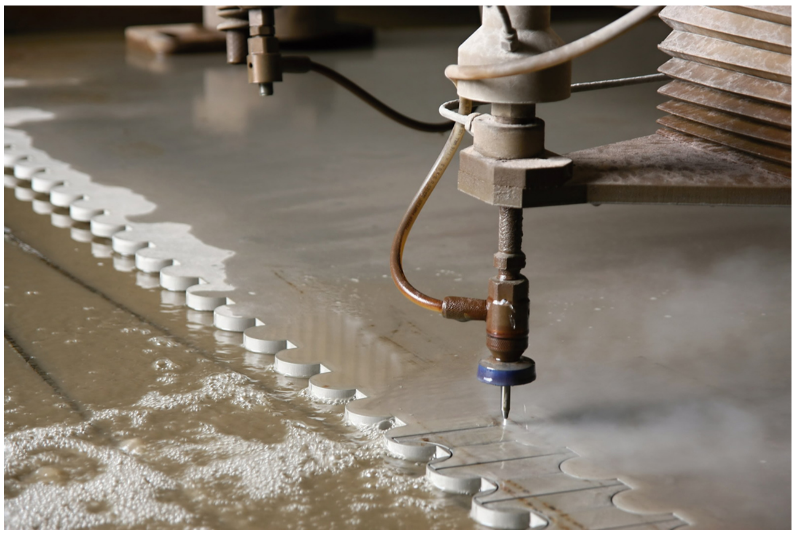


Figure 2.4: Detail of a waterjet cutting stainless steel [35]

Of all the other methods, a waterjet cutter is capable of cutting through the widest range of materials. The cutting is done by using a very high pressure jet of water, which erodes the material while also washing it out. With the addition of an abrasive substance into the stream, it is able to cut very hard materials like granite. The greatest advantage of this method is that it does not add any heat into the system, making it suitable for applications, where the materials being cut are sensitive to high temperatures.

2.5. Wire EDM

EDM stands for Electrical Discharge Machining and like plasma cutting, it works on the principle of creating a electrical circuit through the work piece, and is therefore only applicable to conductive materials. Also known as spark machining, the cutting is done by creating high voltage difference between an electrode and the work piece, while being

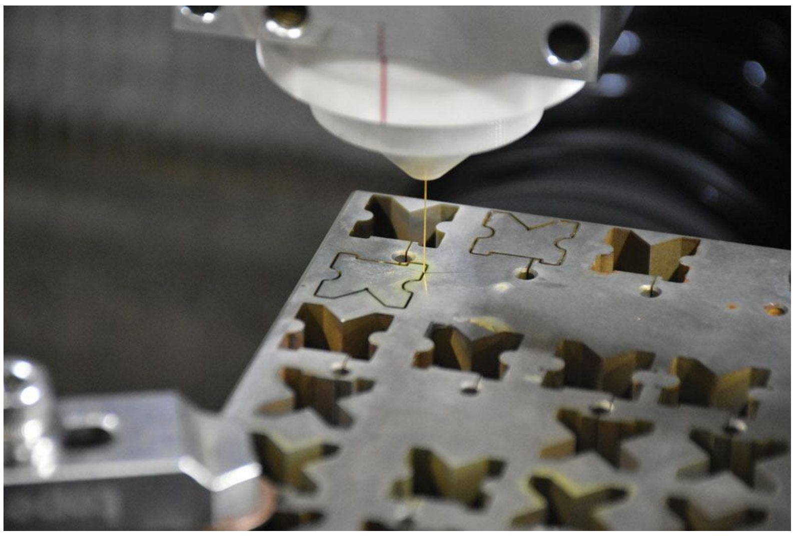


Figure 2.5: Wire EDM cutting a thick piece of material [26]

submerged in an dielectric cooling liquid, and closing the distance between the two until a spark (electrical discharge) is created. A rapid sequence of discharges is what removes the material. A Wire EDM is a version of this machine where the electrode is a wire that is reeled between two spools to avoid excessive wear on a single segment. The spools are positioned on opposite sides of the workpiece, which is placed on a bed that moves it around. While being the most precise method, it is also the slowest, but the maximum thickness of the workpiece can reach almost up to 100 cm.

3. Representation of Objects in 2D

There exist multiple ways of representing 2D objects, all of which can be divided into two well known categories – *vector* and *raster* – both of which approaches have been used in 2D shape placement optimisation problems. This chapter outlines the differences between them and mentions some of their use and application in literature.

3.1. Raster representation

The earliest packing and nesting algorithms were concerned about squares and rectangles [4]; in this case it is useful to describe the object either simply as *width* and *height* or in the form of a *grid-like structure*. This representation is however not limited only to those, actually, an arbitrary shape can be approximated using a rectilinear grid.

Figure 3.1 shows a raster approximation of an irregular shape. It can be seen that when the sides are not aligned to the grid the discretization effect makes the object larger than it in reality is. It is also apparent that among other disadvantages there only exist 4 rotations of a shape. This approach is further discussed in Section 6.2.2.

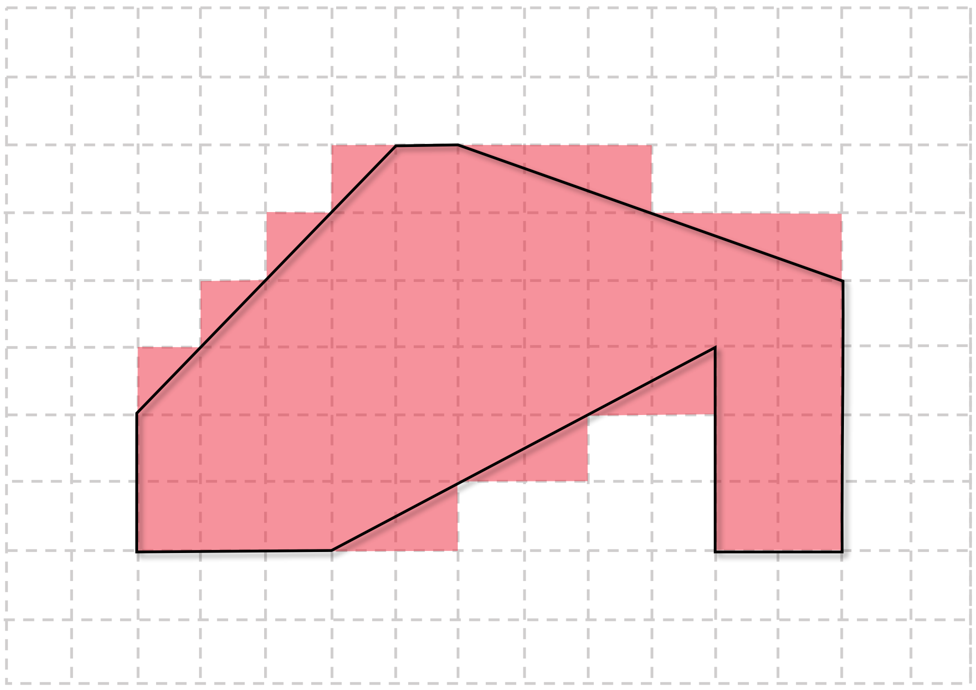


Figure 3.1: Raster representation of a 2D shape

3.2. Vector representation

Vector, or rather mathematical, representation is used to store the description of how to construct the shape rather than storing every "pixel" of it. Each shape can be represented as a combination of basic geometrical primitives such as circles, rectangles, arcs, line segments or Bezier curves. Storing only the mathematical description greatly reduces the memory needed, but there is a trade-off with more computational power needed to analyse (or draw) them.

- **SVG**
  The most pronounced and widespread representative is the Scalable Vector Graphics format. It stores the graphical representation in the form of a XML markup file in a tree structure. Similar (often proprietary) formats with the same idea exist. The SVG format is also used for the input data presented to the open-source nesting software SVGnest [32], more in Chapter 7 (Existing software).

- **Polygons**
  Polygons can be thought of as a subset of the vector representation methods because they only use straight line segments to represent the boundary of shapes. They are also a great compromise between computing power and precision as it is much faster to compute a linear interpolation between two points than it is to find an arbitrary point on a curve.

  Given that polygons are capable of approximating arbitrary shapes to some finite precision, they are almost exclusively used for spatial analysis. The usual format in which this kind of data is exchanged is called a Well-Known Text (WKT). Chapter 5 (Spatial Analysis) goes more into detail on this.

- **G-Code**
  G-Code is language used to control computer numerical control (CNC) machines. It is used to describe the paths the machine has to make to create the target shape. But in a way, it also contains a vector representation of the shapes that are to be machined. The next chapter (4) goes a bit more in depth on this topic.

4. G-Code

G-Code (also known as the NC-Code or ISO Code) is the most widespread programming language used to control CNC machines in computer-aided manufacturing (CAM). The name comes from the fact that most program instructions are prefixed by the letter G (see 4.3). While the language's structure is standardised by the ISO, there exist many different flavours and implementations of the language with subsets or extensions to the standard instruction set that were adapted to suit various machining tools: from lathes, mills, laser or waterjet cutters to 3D printers.

4.1. History

The first version of a numerical control programming language was developed at the MIT in the late 1950s. In the following years many implementations that used a variant of G-Code were developed by various organisations. The final official version was standardised in the United States in 1980 under the name RS-274-D by the EIA. Other countries either adapted or extended the standard and so the G-Code is also known as the ISO 6983 or DIN 66025.

The diversity of standards naturally caused some incompatibilities and there was a movement to standardise the industry on machine controllers manufactured by FANUC group [42]. The diversity has never really been resolved but in the modern days, G-Code is rarely written by hand but is most of the time generated using a CAM software and used as a translation layer between the computer and the machine with the use of a machine-specific post processor[1]. Although some manufacturers hide the G-Code generation in their control software from the user [39], it is often being used under the hood.

The early G-Code was a rigid list of instruction that directly translated into movements of the machine, but has since evolved into a full Turing-complete programming language with most of the functions of modern higher level languages with user defined variables, loops or conditional operators. Manufacturers also often allow the G-Code program to access some of the controller data via predefined variables. [38]

4.2. Syntax

The code is stored as a plain text file and the instructions are interpreted by the order they appear in. A command consists of a single letter and a number identifier followed by one or more parameters called *addresses* that also typically consist of a letter and a number, if some address is not specified [2], the default or the previously used value will be substituted. A typical move instruction is presented in figure 4.1.

It is a common practise to use the `N` command to mark the line number. These commands are ignored by the machine, but can be used by the controller to reference a line at which some error occurred. The `G` indicates a command from the *G* set, in this case number `1` which corresponds to a linear interpolation. The addresses `X` and `Y` mark the target position of the machine tool head, either in global coordinates or relative to

[1]Analogous to a printer driver for personal computers

[2]Typically the feed rate

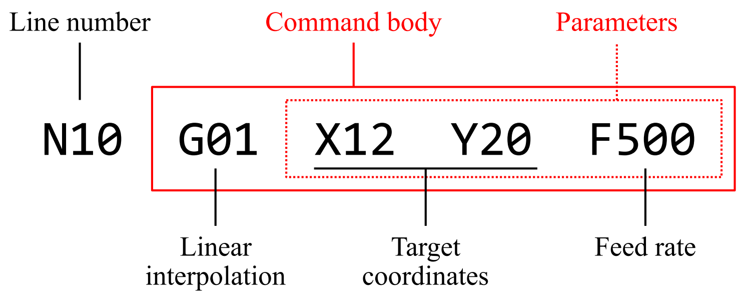
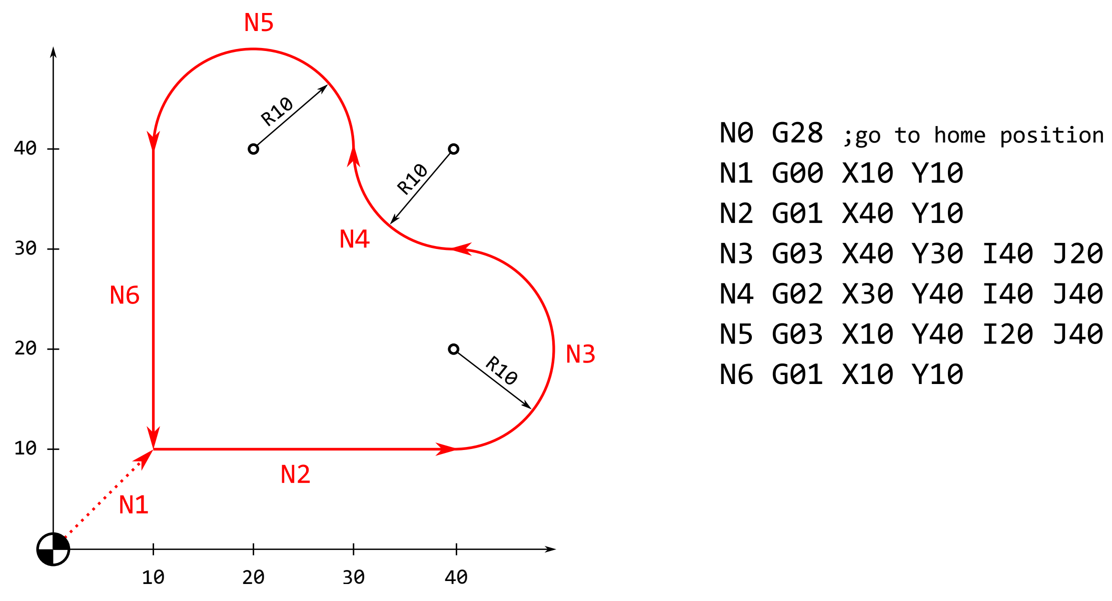


Figure 4.1: G-Code command example

the current position. Finally, the `F` address sets the feed rate (or the speed) in units per minute at which the tool will approach the target. The units can be either millimetres or inches and can be toggled programmatically.

Comments are delimited either by a semicolon ‘`;`’ until the end of the line and multiline comments can be alternatively enclosed in parentheses “`()`”.

Although it is not required, for a better readability, a single line commonly contains a single command and the addresses are delimited by a space.



```
N0 G28 ;go to home position
N1 G00 X10 Y10
N2 G01 X40 Y10
N3 G03 X40 Y30 I40 J20
N4 G02 X30 Y40 I40 J40
N5 G03 X10 Y40 I20 J40
N6 G01 X10 Y10
```

Figure 4.2: Machine path from G-Code

Another common travel move is the arc interpolation (`G02` and `G03`). In addition to the target coordinates, it also needs another set of parameters (`I` and `J`) that mark the center of the arc [33]. Figure 4.2 shows the machine path translated from a source G-Code.

4.3. Commands

There are two groups of commands (or functions) in G-Code: The `G` (Go) commands, that are used to control the motion and function of the system, and `M` (Machine) commands, that controls the machine operation outside of movements. Table 4.1 lists some basic commands. While the commands `G68` and `G92` are not that common, they will play an important role in Section 8.6 (Export).

Table 4.1: Common G commands

Command	Description	Accepted addresses
`G00`	Rapid linear travel move	X, Y, F
`G01`	Linear interpolation	X, Y, F
`G02`	Clockwise circular interpolation	X, Y, I, J, F
`G03`	Anti-clockwise circular interpolation	X, Y, I, J, F
`G20`	Select inches as the unit	–
`G21`	Select millimetres as the unit	–
`G28`	Return to home position	X, Y
`G68`	Rotate coordinate system	X, Y, R
`G90`	Set absolute positioning mode	–
`G91`	Set incremental positioning mode	–
`G92`	Move coordinate system	X, Y

M-commands are often machine-specific, they are used to set the speed of the spindle, or in case of a laser cutter, set the intensity of the laser; turn on or off the cooling liquid or a fan and so on. A sample of M-commands can be seen in Table 4.2.

Table 4.2: Sample of M commands

Command	Description	Accepted addresses
`M00`	Machine stop	–
`M02`	End of program	–
`M03`	Start spindle (CW)	S
`M04`	Start spindle (CCW)	S
`M05`	Stop spindle	–
`M06`	Automatic tool change	T
`M07`	Coolant on	-
`M09`	Coolant off	-
`M30`	End of program	-

5. Spatial Analysis

In the most general sense, spatial analysis is a collection of methods of describing objects and their relation in space. It has many applications in geography, topology and geometry. Geospatial data are commonly exchanged in a WKT (Well-Known Text) format, which is a text markup language designed to be readable by humans, standardised by the Open Geospatial Consortium [25].

As has been said earlier, these methods almost exclusively work with polygons (or polyherda in higher dimensions), and while also circles and other "curves" can be expressed using WKT, they are rarely used for computation performance purposes and many software libraries implement them as a polygonal approximation or not at all. The popular software libraries used for spatial analysis include:

- **CGAL** (Computational Geometry Algorithms Library) is the most robust and complete C++ library for geometric operations in both 2D and 3D, including convex hulls, meshes, triangulations, and many more. [9]
- **GEOS** (Geometry Engine, Open Source) is an open source C/C++ library for 2D spatial analysis including validation and topology functions. It is used in many GIS software applications and there exist many bindings that enable its use within different programming languages like Python (Shapely), PHP (GeoPHP) or Node.js (node-geos). [18]
- **Boost.geometry** is a part of the *Boost* collection of C++ libraries. It is the youngest of these three, so its feature set is more limited, but it is being regarded to as one of the most performant. [7]

5.1. Shapes

The terminology for different shape names varies slightly between implementations, so throughout this chapter the terminology of the GEOS library will be used.

Any shape (or object in space for that matter) is associated with three sets: the *boundary*, which is the set of all points laying on its boundary, the *interior*, which contains all the points inside the object, and *exterior*, which contains every other point not in the shape. They are all mutually exclusive and their union gives the entire space (or a plane in 2D). Every object can be also characterised by its *length* and *area*.

The following list lists the commonly used geometric objects [17]. Their corresponding WKT representation can be found in Figure 5.1.

- **Point** - The interior of a point object consists of a single point, while the boundary is an empty set. Both the length and area of a point are equal to zero.
- **LineString** - Line string, or a polygonal chain, is a series of connected line segments. The boundary contains the two extreme points, while the interior consist of all the points along its length (not just vertices). While it has zero area, the length is defined as the combined length of the line segments. An extension of a line string is a linear ring in which the two extreme points coincide[1].

[1]Linear rings also do not have a boundary

- **Polygon** - A polygon consists of an exterior linear ring and zero or more interior rings (i.e. a polygon can have polygonal "holes"), which together make its boundary and the interior is defined by the area between these. Polygon also the only object with non-zero area.

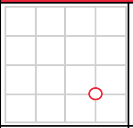
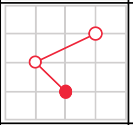
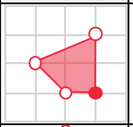
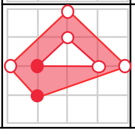
Type	WKT Example	
Point		`POINT (30 10)`
LineString		`LINESTRING (20 10, 10 20, 30 30)`
Polygon		`POLYGON ((30 10, 40 40, 10 20, 20 10))`
		`POLYGON ((10 10, 40 20, 20 40, 0 20), (10 20, 30 20, 20 30))`

Figure 5.1: Example of WKT representation

It is also useful to define other constructs that act as "containers" for various geometry types, such as for example a *MultiPolygon* which describes a collection of polygons or a *GeometryCollection* as a generic container for arbitrary objects.

Another important property of objects it their *validity*: a object is only valid if its interior does not cross itself or that the interior rings of a polygon do not cross its boundary. Figure 5.2 shows an example of invalid self-intersecting objects.

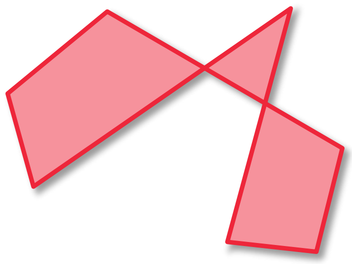


Figure 5.2: Example of invalid geometry

5.2. Spatial Operations

There are various operations that can be applied to the objects, this chapter discusses the majority of them, according to [12].

5.2.1. Relationships

Relationships between two objects can be defined as spatial predicates[1] that are applied to one of these object with respect to the other. Consider two objects A and B, Table 5.1 lists different spatial predicates applied to[2] object A in relation to B.

Table 5.1: Spatial relationships predicates

Predicate	Is true, when
Equals	The object are topologically equal
Disjoint	The objects have no common point
Intersects	The inverse of disjoint (i.e. A and B have at least one common point)
Touches	A and B have at least one *boundary* point in common, but their interiors are disjoint
Overlaps	It the objects share some *but no all* interior points
Within	All points of A lie within the interior of B
Contains	The inverse of within (i.e. All points of B lie within the interior of A)

5.2.2. Simplification and Bounding volumes

To improve the performance of computation, it is often beneficial to simplify the geometry, or approximate it with a simpler object. When testing for an intersection, it is also useful to do this in two phases - the *broad phase* when a rough, simplified (or approximate) version of the geometry is used to avoid unnecessary computation when there is no chance of the two intersecting, and then in a *narrow phase*, actually test against the true representation. This simplified representation can be achieved in two ways – by reducing the number of vertices along the boundary of the shape or by constructing a *bounding volume* around it.

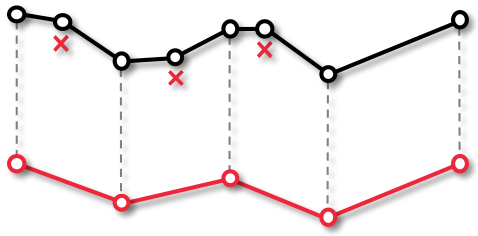


Figure 5.3: Example of a simplified polyline

[1]Predicate is a Boolean function, i.e. a function that returns either *true* or *false*
[2]Often denoted as A.predicate(B)

Polyline simplification

If the desired tolerance permits it, it is preferable to reduce the vertex count, as this allows for significantly faster computation. The popular fast Douglas-Peucker algorithm [14] is most often used. A disadvantage of this approach is that, as opposed to the bounding volumes method, it does allow for false negatives[1] when checking for overlap, so it must only be used when this would not cause significant issues.

Bounding volumes

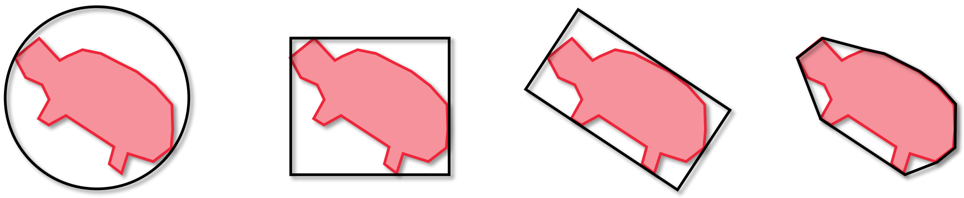


Figure 5.4: Bounding volumes (Circle, AABB, OBB, Convex hull)

- **Axis-aligned Bounding Box** - Being fastest to construct and very easy to test for overlap, the AABB is almost always used as the first test to check intersection of objects in the broad phase. It is defined by a pair of points specifying the top left and bottom right corner.
- **Minimum Rotated Rectangle** - Also known a an *oriented bounding box* or OBB, it is the smallest rectangle, that encloses all the points. The intersection tests are also relatively computationally cheap, but its construction is harder than it seems. The exact algorithm has a cubic asymptotical time complexity, so an approximate linear time algorithm is used instead.
- **Smallest Enclosing Circle** - The intersection test of two circles is the simplest one - only the distance between their centers and a sum of their radii have to be compared. Though the calculation of the circle is not a trivial task, it also possess the property of rotational invariance, meaning that once a smallest enclosing circle is constructed for an object, id does not need to be recalculated when it rotates. Most often, the Welzl's algorithm [41] is used for its construction.
- **Convex Hull** - The convex hull is defined as the smallest polygon (with the shortest circumference) which encloses a set of points. As convex shapes are best suited for intersection testing, the convex hull is best suited for fast test without sacrificing a lot of precision.

5.2.3. Operations

Affine transform

Affine transformations are a collection of linear operations that can be applied to an object to *transform* its shape either by translation, rotation, scaling or shearing. This

[1]e.g. simplified objects no longer intersect even though they did before

section is concerned only with the first two. Any affine transformation can be described using a *transformation matrix*, which when multiplied by a coordinate vector, gives the new transformed coordinates. Let x and y be the original coordinates of point, then:

- **Translation** by a required offset t_x and t_y is defined as

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- **Rotation** about the origin by an angle θ is defined as

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Multiple matrices can be also combined into one to achieve different transformations at once. An object is transformed by multiplying the matrix with all its points.

Set operations

As the objects are just sets of points, all the common binary set operations can be also applied to them. Figure 5.5 shows demonstrations of *union*, *difference*, and *intersection* of two objects A and B.

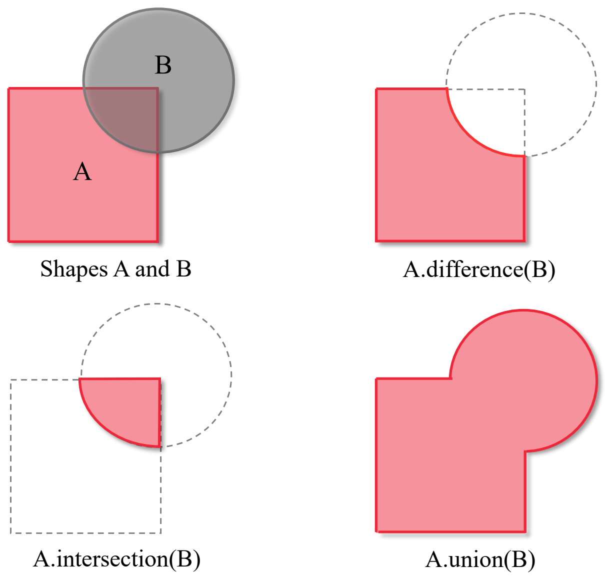


Figure 5.5: Example of binary operations on objects

Minkowski Addition

Also knows as the Minkowski sum, it is a binary operation that can be applied to a pair of sets of vectors. Given objects A and B, a Minkowski addition is defined as

$$A \oplus B = \{\mathbf{a} + \mathbf{b} \mid \mathbf{a} \in A, \mathbf{b} \in B\},$$

where $\mathbf{a} + \mathbf{b}$ is a vector sum of all the vectors that represent points from the objects A and B. Analogously, the Minkowski difference is defined as

$$A \ominus B = \{\mathbf{a} - \mathbf{b} \mid \mathbf{a} \in A, \mathbf{b} \in B\},$$

or in terms of Minkowski addition as $A \oplus (-B)$. An important property of the Minkowski difference is that if and only if the objects A and B intersect (share a common point), their Minkowski difference contains the origin[1].

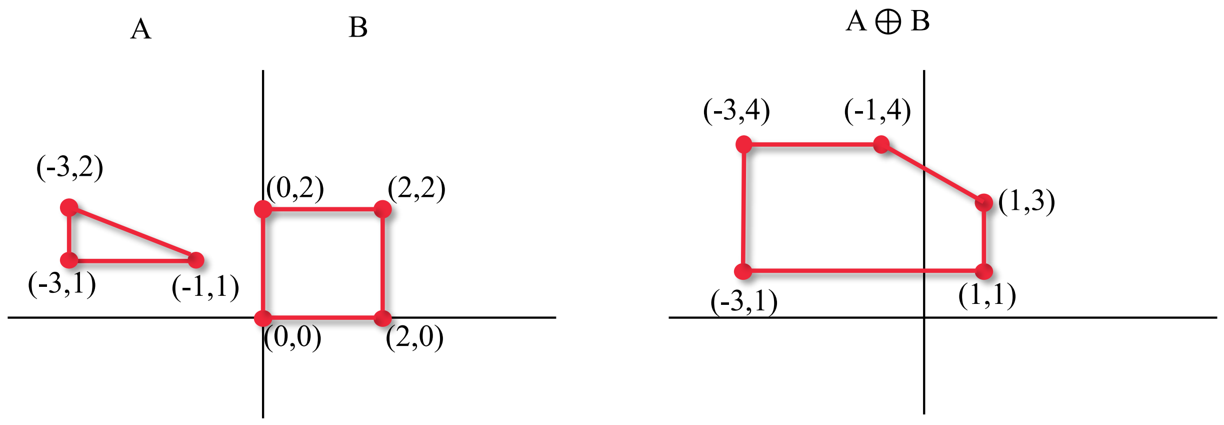


Figure 5.6: Minkowski sum

Buffer

The buffer of an object can be thought of a set of all points *within* a certain distance from it. In computer graphics, a positive buffer is also referred to as a *dilation* whereas a negative buffer is called an *erosion*. In mathematical terms, a positive buffer is defined as a Minkowski sum of the object and a disc centred at the origin with a radius equal to the size of the buffer, while a negative buffer is a difference of the shape with a positive buffer of its boundary.

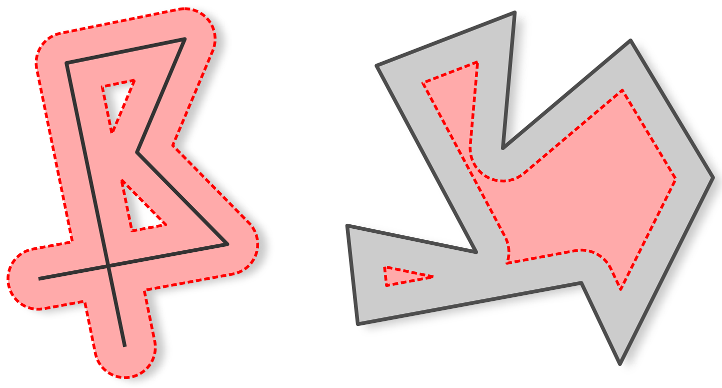


Figure 5.7: Positive and negative buffer

Figure 5.7 shows a positive buffer applied to a polyline and a negative buffer applied to a polygon.

[1]Because there would exist a pair of points whose difference is 0.

6. The Nesting Problem

6.1. Definition

The *Nesting problem* is a collective term for a number of cutting and packing problems, which in general deal with finding an optimal placement of parts in a container without overlap, minimising the unused area with respect to some criteria. These can be generally divided into three categories:

- **Bin packing problem** - Given a set of shapes and a container of a fixed size, arrange the shapes in such way that minimises the number of said containers needed to pack all of them.
- **Strip packing problem** - Given a set of shapes and an “infinite” strip of material of fixed width, arrange the shapes in such way that minimises the length of the material needed.
- **Knapsack problem** - Given a set of shapes and a fixed container, find the optimal arrangement of a subset of shapes that maximises the used area.

Various additional constraints are often needed to be defined, such as a minimal clearance between shapes and the container's boundary or a fixed rotation of shapes. This is because it may be necessary to account for the cutting tool width or the *kerf* of a laser; The fixed rotation constraint applies for example in the textile industry where the pieces need to be aligned to the grain of the fabric. The latter is actually a major simplification of the problem as opposed to the general definition where the rotation of shapes is allowed and thus the solution space is much bigger. [24]

It is not surprising that the Nesting problem is *NP-hard*, [29] meaning that there is no known algorithm that finds the optimal solution in a polynomial time, nor is there one that determines that the found solution is, indeed, optimal. That means one can only rely on a good heuristic to provide an objectively good solution. The next section pinpoints some existing approaches to solving different variants of the problem. [16]

6.2. Existing Solution Methods

In general, the methods can be divided into two categories: *Single pass*, where the “optimal” solution is found by placing all the object one by one to their respective local optima, and *iterative* where the said process is often used to generate a first good solution and then iteratively try to enhance it. These can be further divided into ones that produce a first *legal*[1] solution, and ones that allow for some overlap and try to minimise it with further iterations. The common theme is, however, finding the order of placement.

As has been noted, a good heuristic is needed to ensure a good solution. According to the empirical results in literature, a *First fit decreasing* is the best heuristic [15]; It says that the best initial results are obtained by sorting the shapes by their size[2] and placing

[1]A legal solution is one where the shapes are all placed according to the constraints (e.g. do not overlap each other)

[2]This can mean the area, width, height, etc...

them in a decreasing order. This section outlines some of the developed solution methods and different applicable heuristics.

6.2.1. Binary Tree Bin Packing

The easiest variant of the problem is one where both the shapes and the container are rectangular. While this can often be a gross simplification, the algorithm for this problem is "fairly simple" with an asymptotical complexity of $\mathcal{O}(n \log n)$

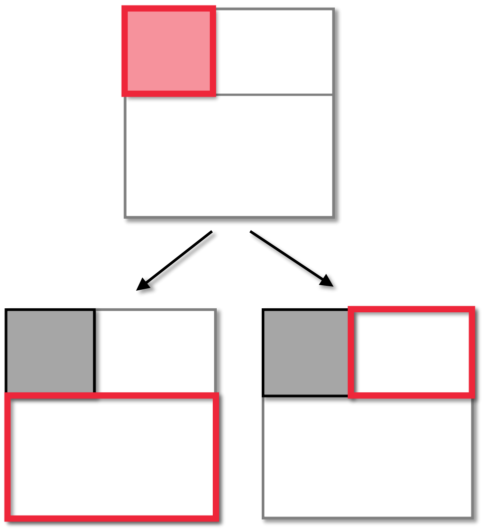


Figure 6.1: Binary tree bin packing

Every shape is represented by its *width* and *height*, and so is the container. The container can also have only one of its dimensions fixed, making it a *strip packing problem*, the only difference is that this way all shapes are always placed, but the algorithm stays the same.

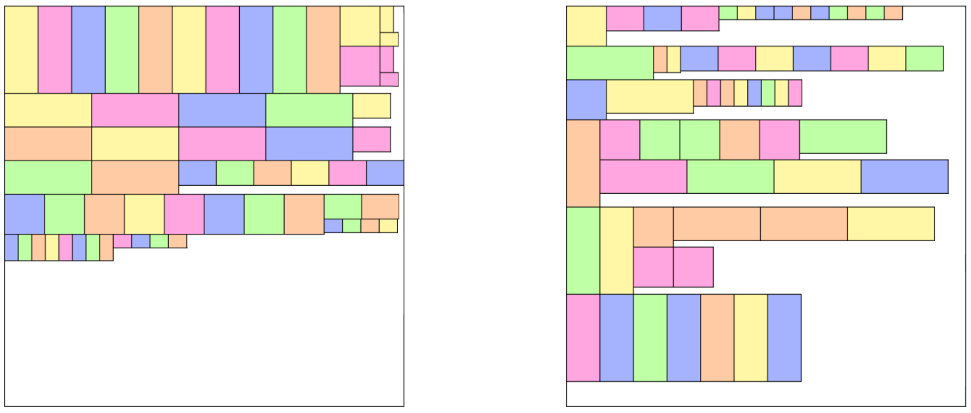


Figure 6.2: Comparison of different heuristics for binary tree bin packing [20]

The container itself acts as the root node of the binary tree. When the first shape is placed, the left node represents the empty space below it and the right node represents the empty space to the right of it. When a new shape is placed to either node, the remaining space is split in a similar fashion and the algorithm proceeds recursively until all shapes are placed or no shape fits the remaining space. See Algorithm 1. [20]

In Figure 6.2 can be seen the difference in the quality of the result between a well chosen heuristic on the left (sorted by descending area) and poorly chosen one on the right (random order).

Algorithm 1 The Binary Tree Bin Packing Algorithm (according to [20])

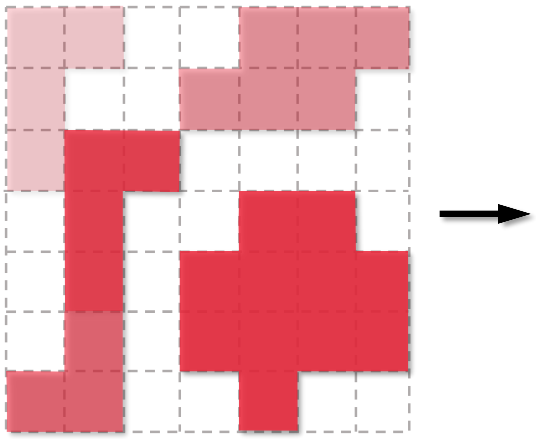
```
Require: shapes is a sorted list of shapes
Require: root is a instance of Node with variables $x$, $y$, $w$, $h$, down, right and shape

 1: procedure PACK
 2:     for all $s$ from shapes do
 3:         node ←FINDNODE($root, s.w, w.h$)
 4:         if node is not $NULL$ then
 5:             $node.shape \leftarrow s$
 6:             $node.down \leftarrow$ new $Node\{x \leftarrow node.x,$
                                       $y \leftarrow node.y + s.h,$
                                       $w \leftarrow node.w$
                                       $h \leftarrow node.h - s.h\}$
 7:             $node.right \leftarrow$ new $Node\{x \leftarrow node.x + s.w,$
                                       $y \leftarrow node.y,$
                                       $w \leftarrow node.w - s.w$
                                       $h \leftarrow s.h\}$
 8:         end if
 9:     end for
10: end procedure

11: function FINDNODE($node, w, h$)
12:     if $node.shape$ is not $NULL$ then                ▷ Node is already occupied
13:         return FINDNODE($node.down, w, h$) or FINDNODE($node.right, w, h$)
14:     else if $w \leq node.w$ and $h \leq node.h$ then          ▷ Shape fits
15:         return $node$
16:     else                                            ▷ Shape does not fit
17:         return $NULL$
18:     end if
19: end function
```

While the assumption of a rectangular container can be mostly satisfied in real applications, for the shapes, it is rarely the case. Therefore, more advanced algorithms that deal with irregular shapes had to be developed.

6.2.2. Rectilinear Grid Genetic Algorithm

Many methods make use of approximating the shapes using a rectilinear grid, as has been noted in Chapter 3. The rationale behind this representation is that it allows for relatively fast checks for overlap, which are in the worst case linear with respect to the area of the object. On the other hand, the area grows quadratically with the size of the object, so it is only useful with *relatively small* shapes that do not suffer much from the discretization imposed by the granularity of the grid.

The authors of [10] proposed a *genetic algorithm* for solving this version of the problem. Due to the fact that the shapes can be represented using a discrete grid, the whole area can be simply converted to 2D matrix chromosome representation, as can be seen in Figure 6.3.



1	1	0	0	2	2	2
1	0	0	2	2	2	0
1	4	4	0	0	0	0
0	4	0	0	5	5	0
0	4	0	5	5	5	5
0	3	0	5	5	5	5
3	3	0	0	5	0	0

Figure 6.3: Shape arrangement (left) vs. 2D chromosome representation (right)

Native to a genetic algorithm are the operations of *mutation* and *crossover*. The mutation operation, in this case, creates a new arrangement by randomly changing the position and/or rotation of a single shape, while the crossover operation is more complicated. The idea is to randomly select two parents and find the largest common area[1] that can be swapped between these two, producing 2 children. This also means that not all pairs of chromosomes are compatible for crossover.

At the end of each phase, a number of the fittest arrangements are selected. The objective fitness function takes into account the number of edges E adjacent to an unoccupied cell as well as the total wasted area A and looks as follows: $\alpha \cdot E + (1 - \alpha) \cdot A$, where the parameter α controls the balance between the two metrics. The rationale is that more closely packed sets of objects have fewer exposed edges than loosely packed ones.

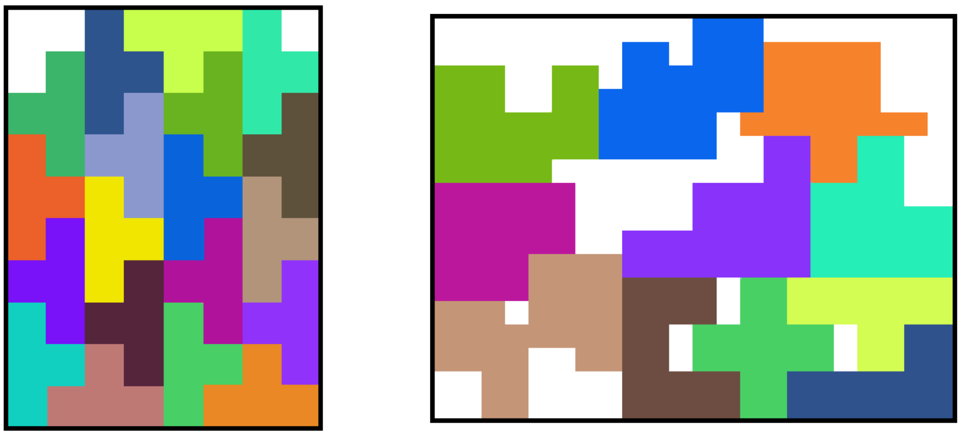


Figure 6.4: Difference between simple and complex shapes [10]

[1] An area of the same shape that does not cross any object in either of the two arrangements

According to the authors, this process takes a significant amount of time and is only applicable to smaller objects that can be well approximated by the grid. Figure 6.4 demonstrates the difference in packing quality between simple and complex shapes.

6.2.3. Bottom Left Greedy Heuristic

In many packing algorithms, the idea is to place the objects one by one to the lowest, leftmost position in the container. The bottom left greedy heuristic was introduced in [3] for packing arbitrary collection of rectangles, in a variant of strip packing problem, to minimise the extra height added by any shape. It was later extended to also work with arbitrary shaped objects.

The algorithm works thusly: first an object is “dropped” to its lowest possible location in the container, the technique of achieving this is an issue of its own and depends on the representation of objects; it can be as simple as finding the lowest non-occupied space in a rectangular grid, or when working with polygons, using either some simpler bounding shape or the No-Fit Polygon. The authors of [10] used a binary search to find the lowest point of no intersection. Then, in a similar fashion, the shape is moved all the way to the left and then down again and the process is repeated until it can no longer be moved. A simple illustration can be seen in figure 6.5.

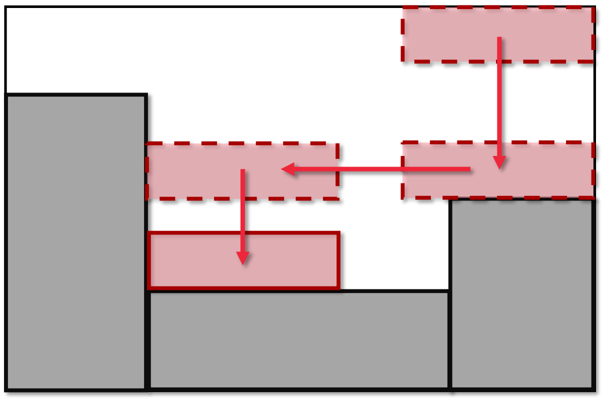


Figure 6.5: Bottom Left Greedy algorithm

There are versions of this method where the shapes are allowed to “tunnel” through already placed pieces if there exists a legal placement position beneath them. This is useful when the smallest shapes are placed last and fill the holes between already placed bigger shapes. [10]

An advantage of this approach is its speed and simplicity, though naturally, in comparison to more sophisticated methods, solutions found using this method generally tend to be of a worse quality.

6.2.4. The No-Fit Polygon

The most advanced and accurate nesting algorithms utilise what is called a *No-Fit Polygon* or NFP. It is a powerful data structure used for testing shapes for overlap and can described as follows: Given a fixed polygon A, free polygon B and a reference point on polygon B, the No-Fit Polygon of A with respect to B (denoted as NFP$_{AB}$) is constructed by “sliding” polygon B around A without overlap as closely as possible, tracing the position of the reference point. The final path of the reference point, once the polygon B has completed a whole revolution, is the resulting NFP, as is demonstrated in Figure 6.6

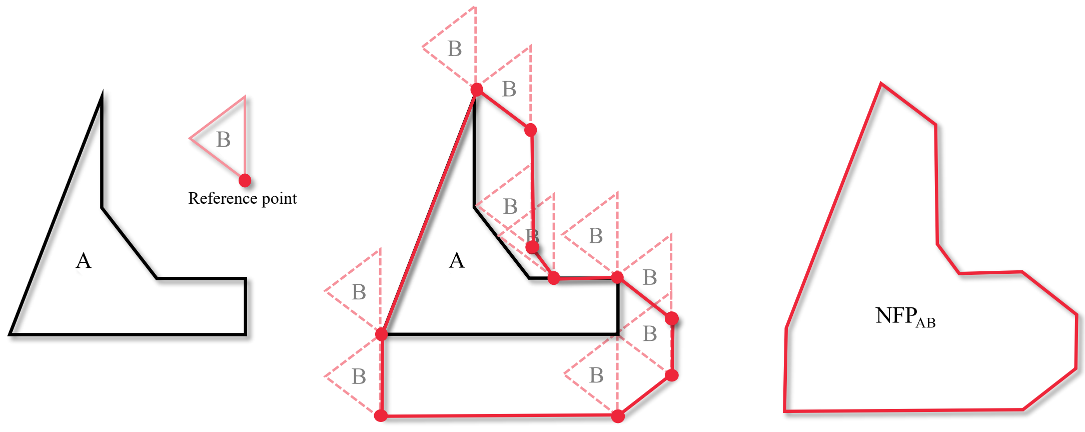


Figure 6.6: No-Fit Polygon

Checking for overlap of the two polygons is then reduced to checking if a the reference point lies within the No-Fit Polygon. If it does, the two shapes overlap; if the reference point is on the boundary of NFP$_{AB}$, then polygon B touches polygon A and finally when it is outside, the two do not overlap or touch.

It can be shown that when both A and B are convex, the NFP can be simply constructed as a Minkowski difference ($A \oplus -B$) of both shapes. For more complicated shapes, however, the resulting No-Fit Polygon can consist of more than one shape (consider a concave shape with a large inner cavity, where the orbiting shape can be placed, and a thin opening in which the orbiting shape does not fit), or there could be an arrangement where a *single additional point* (consider a puzzle piece) is a part of the NFP. For this reason advanced methods were developed, for example the *Sliding approach* [8] that literally “slides” the shapes around, or a more recent one that still uses Minkowski sums but accounts for the concavities. [6]

One disadvantage of using NFP is that it has to be generated for every pair of objects, plus for every possible rotation of them. The former is not such an issue when there are many copies of the same object, but the rotations are often limited to 4 or 8.

6.2.5. A Note on Advanced Methods

Given that most of the advanced nesting software is proprietary (see Chapter 7), there is not much information about the exact methods that are being used, as they are mostly referend to in marketing terms such as “The most powerful and advanced optimisation algorithms”, but it is clearly more often than not a combination of the *No-Fit Polygon* along with some global optimisation algorithm that iteratively searches for better solutions. *Genetic algorithms* are often used to generate new solutions, there are also mentions of *Simulated annealing* or *Tabu search* to allow for worse solution along the way to escape from local minima. [24]

7. Existing software

In the CNC world, nesting software is most of the time proprietary and targeted primarily at large industrial customers. It is also often prohibitively expensive for hobbyists or small businesses. Luckily, in the recent years, friendlier and open source alternatives have been emerging. This chapter lists some of the existing nesting solutions.

7.1. SigmaNEST

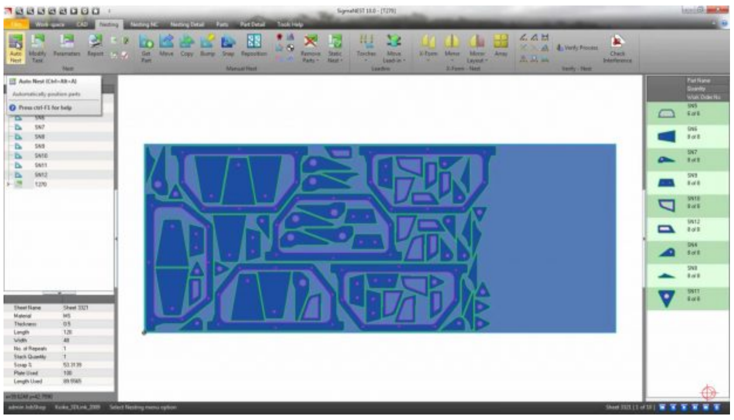
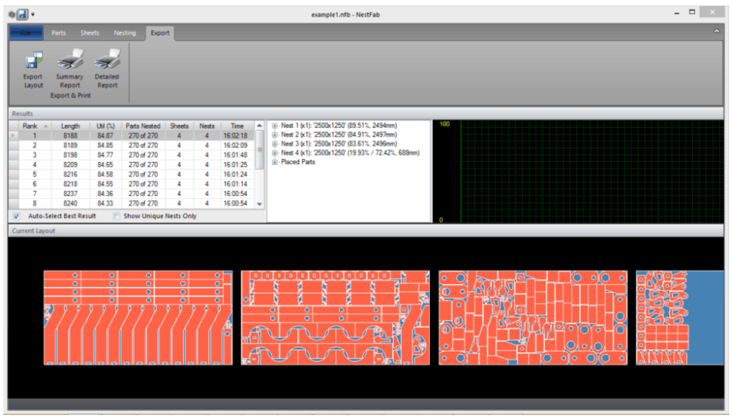


Figure 7.1: SigmaNEST [36]

SigmaNEST is an industrial-grade nesting software developed by SigmaTEK. It is intended for large factories owning tens of machines and is priced accordingly. It claims to be the *best nesting software* out there and to support every cutting machine imaginable. It has support for all the conventional machines like plasma or laser cutters, but it has also options for tube cutting or metal bending. [36]

7.2. NestFab



Rank	Length	Util (%)	Parts Nested	Sheets	Nests	Time
1	8188	84.87	270 of 270	4	4	16:02:18
2	8189	84.85	270 of 270	4	4	16:02:09
3	8198	84.77	270 of 270	4	4	16:01:48
4	8209	84.65	270 of 270	4	4	16:01:25
5	8216	84.58	270 of 270	4	4	16:01:24
6	8218	84.55	270 of 270	4	4	16:01:14
7	8237	84.36	270 of 270	4	4	16:00:54
8	8240	84.33	270 of 270	4	4	16:00:54

[✓ Auto-Select Best Result] Show Unique Nests Only

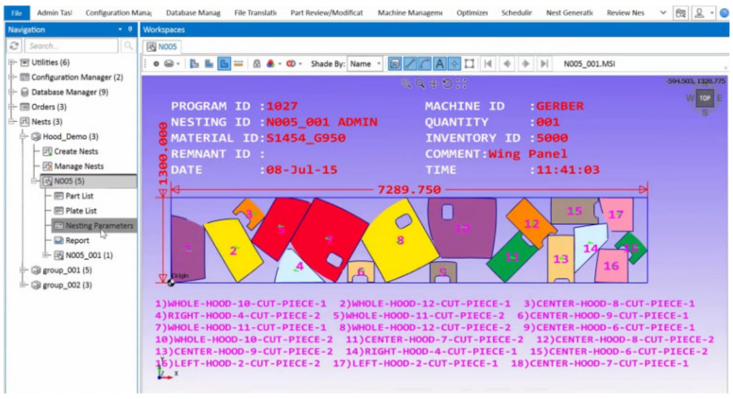
- Nest 1 (x1): '2500x1250' (89.51%, 2494mm)
- Nest 2 (x1): '2500x1250' (84.91%, 2497mm)
- Nest 3 (x1): '2500x1250' (83.61%, 2496mm)
- Nest 4 (x1): '2500x1250' (19.93% / 72.42%, 688mm)
- Placed Parts

Figure 7.2: NestFab [28]

Not being as much versatile as the aforementioned tool, NestFab also claims to be the world's most effective nesting software with the use of the *Ultra-Performance option.*

It has a simpler interface and not as many functions, but that makes it easier to work with. It also supports most of the industry standard file formats like DXF and DWG. [28]

7.3. TruNest

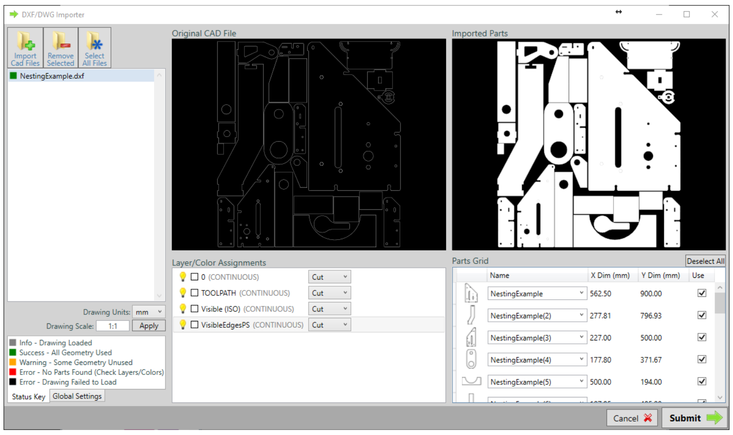


```
PROGRAM ID :1027                 MACHINE ID   :GERBER
NESTING ID :N005_001 ADMIN       QUANTITY     :001
MATERIAL ID:S1454_G950           INVENTORY ID :5000
REMNANT ID :                     COMMENT:Wing Panel
DATE       :08-Jul-15            TIME         :11:41:03

1)WHOLE-HOOD-10-CUT-PIECE-1   2)WHOLE-HOOD-12-CUT-PIECE-1   3)CENTER-HOOD-8-CUT-PIECE-1
4)RIGHT-HOOD-4-CUT-PIECE-2   5)WHOLE-HOOD-11-CUT-PIECE-2   6)CENTER-HOOD-9-CUT-PIECE-1
7)WHOLE-HOOD-11-CUT-PIECE-1   8)WHOLE-HOOD-12-CUT-PIECE-2   9)CENTER-HOOD-6-CUT-PIECE-1
10)WHOLE-HOOD-10-CUT-PIECE-2   11)CENTER-HOOD-7-CUT-PIECE-2   12)CENTER-HOOD-8-CUT-PIECE-2
13)CENTER-HOOD-9-CUT-PIECE-2   14)RIGHT-HOOD-4-CUT-PIECE-1   15)CENTER-HOOD-6-CUT-PIECE-2
16)LEFT-HOOD-2-CUT-PIECE-2   17)LEFT-HOOD-2-CUT-PIECE-1   18)CENTER-HOOD-7-CUT-PIECE-1
```

Figure 7.3: TruNest [2]

TruNest comes from the family of CAD products developed by Autodesk. As such, its biggest advantage is the possibility to seamlessly integrate it with their other CAD software like AutoCAD or Inventor. As with most industrial software, the pricing depends to the customer's needs. [2]

7.4. MyNesting



NestingExample.dxf

Drawing Units: mm
Drawing Scale: 1:1 Apply

⬛	Info - Drawing Loaded
🟩	Success - All Geometry Used
🟧	Warning - Some Geometry Unused
🟥	Error - No Parts Found (Check Layers/Colors)
⬛	Error - Drawing Failed to Load

Status Key | Global Settings

Layer	
☐ 0 (CONTINUOUS)	Cut
☐ TOOLPATH (CONTINUOUS)	Cut
☐ Visible (ISO) (CONTINUOUS)	Cut
☐ VisibleEdgesPS (CONTINUOUS)	Cut

Name	X Dim (mm)	Y Dim (mm)	Use
NestingExample	562.50	900.00	✓
NestingExample(2)	277.81	796.93	✓
NestingExample(3)	227.00	500.00	✓
NestingExample(4)	177.80	371.67	✓
NestingExample(5)	500.00	194.00	✓

Figure 7.4: MyNesting [27]

MyNesting is a unique nesting service, that is provided by the NestFab engine. It lets users freely use the optimisation engine, but the final arrangement can only be exported to a file upon paying a certain fee. This pay-per–use model makes it accessible to individuals and smaller businesses. [27]

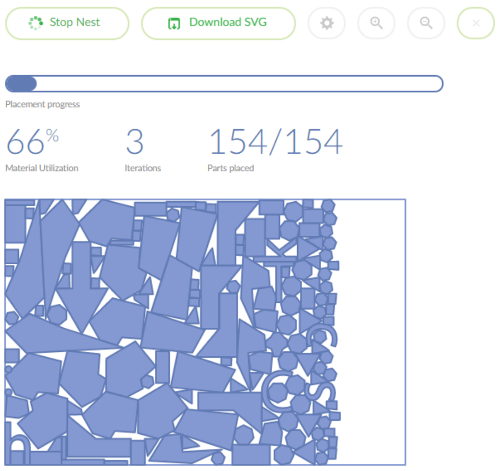


Figure 7.5: SVGnest [32]

7.5. SVGnest

As an answer to the proprietary tools, SVGnest was developed to be an accessible open-source variant. It is written in JavaScript and runs entirely in the web browser. Due to this, the performance is not quite as fast, and also only the SVG format is supported as the input, but quality-wise the produced solutions are competent with those of commercial software. [32]

7.6. Deepnest

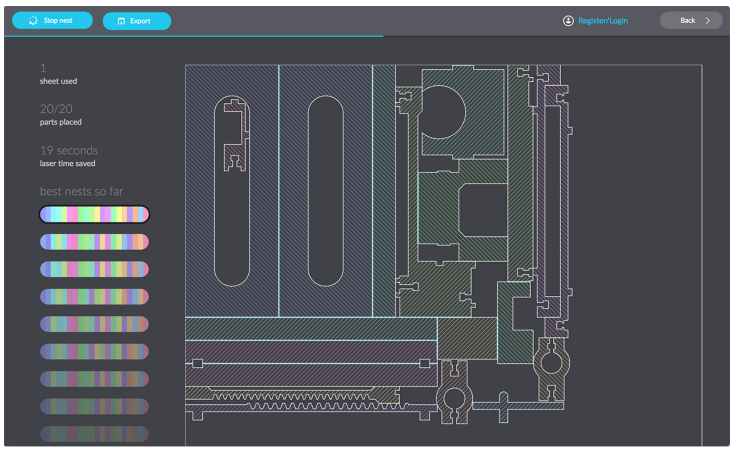


Figure 7.6: Deepnest [13]

Deepnest is a multi-platform desktop application based on SVGnest. It was developed by the same authors and the speed-critical code was rewritten in C++, still being open-source. In addition to this improvement, it can read and export DXF files and it can also merge paths of neighbouring parts to save on redundant cutting passes. [13]

8. Implementation

The task at hand arose from a loose cooperation with the company B+R[1] in an initiative to effectively reuse scrap material from CNC metal cutting. This chapter describes the implementation details of a custom nesting solution that attempts to solve the problem.

8.1. Task Specification

This software is intended to find a placement of provided cutting plans inside of the available material, also considering that the provided material might have been used and parts of it might have been cut out. With this in mind, the problem can be placed into the general category of a *Knapsack problem*, as discussed in Chapter 6.

As such, the solution should provide a set of shapes (with their respective positions) that utilises the most of the available space. Both the shapes and the container can be both convex and non-convex, with the addition that the container can also include “holes” or forbidden regions.

Given that the material used is metal, a free rotation of the shapes is allowed and there is a need for a user-definable clearance between them, preferably in a form of a simple graphical user interface.

The input and output of the program should be able to read and produce G-Codes, as the result is intended to be used by a CNC machine.

8.2. Software Tools

The programming language of choice was *Python* [31], mainly because of its interpreted nature, which allows for testing and benchmarking pieces of code quickly without the need for a compilation in an interactive shell or a using a *Jupyter Notebook*. Another reason was the wide range of available libraries and the ease of use. In real production, though, it would be wiser to use a compiled language instead.

A number of additional libraries was also used, namely *Shapely* [19] that provides a Python interface to the GEOS spatial analysis library, and *numpy* [30] to simplify vector algebra operations. *PyQt5* (a Python binding for the Qt framework [37]) was used to create the graphical front end with the help of *matplotlib* [22] to draw the workspace. An additional C++ library *libnfporb* [21] was leveraged to create the No-Fit Polygons (which also depends on the *Boost.geometry* [7] library) and thus a Python–C++ interface had to be created using the *pybind11* [23] library.

The table 8.1 lists all the software tools needed and their versions, along with their respective license. Some of them are optional if the whole functionality is not needed; the essential required tools are marked with an asterisk.

Additionally, the program Qt Designer was used to help layout the GUI (see Section 8.7). Furthermore, a C++ compiler is needed to build the optional supporting NFP library. A one-click build script for Windows users (for Microsoft Visual Studio) is provided and a guide can be found in the attached archive.

[1]B + R Automatizace, s.r.o

Table 8.1: Software tools

SW	Version	License
Python*	3.7.3	PSFL (BSD, GPL Compatible)
numpy*	1.16.2	BSD
matplotlib	3.0.3	BSD Compatible
Shapely*	1.6.4	BSD
PyQt5	5.9.2	GPL
pybind11	2.4.3	BSD Compatible
libnfporb	2018-5-22	GPL
Boost	1.65	Boost license (permissive)

* Essential

8.3. Program Architecture

The program is presented as a set of Python modules using an object-oriented approach. There are basically two key components which are tied together by an unified API, upon which a graphical user interface is built.

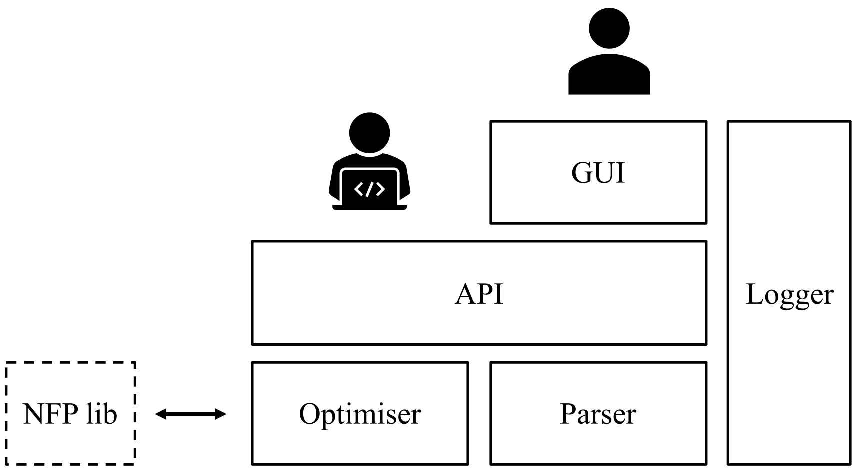


Figure 8.1: Program architecture

The architecture is best described using the scheme in Figure 8.1. On the lowest level, there is the module *Parser*, which is used to translate the input data into an usable format (more on that in the next section). Next is the module *Optimiser*, which is the brains of the whole application and is described in detail in Section 8.5. It can optionally (and preferably) use the NFP generation algorithm provided by *libnfporb* using a Python–C++ interface.

On top of these sits an API that provides access to their function and also implements some helper routines that simplify calling the modules' methods. The API module can be itself included in any other program to leverage the underlying algorithms.

Above all that is a GUI built in Qt, that wraps everything into a ready-to-use package with all necessary tools and options. The GUI is further described in Section 8.7.

For debugging and progress review purposes, a logger module is accessible by all other modules. It provides an interface for logging different types of messages (debug, info, warning, error) and can be configured which of these to show to the user.

8.4. G-Code Parser

Due to the fact, that the sample input files were presented in the form of a G-Code, it was necessary to process them into an usable format. This was achieved by a custom G-Code parser that extracts the shapes and turns them into polygons.

Only a small subset of the G-Code instructions was needed, namely `G0`, `G1` for travel moves and straight lines and `G2`, `G3` for arcs and finally `G90` and `G91` for toggling between a relative and an absolute coordinate system. All other instructions, along with the `Z` parameter of the move instructions, can be disregarded, as they are not important for the conversion. User variables were also not considered in the implementation and the units are assumed to be millimetres[1].

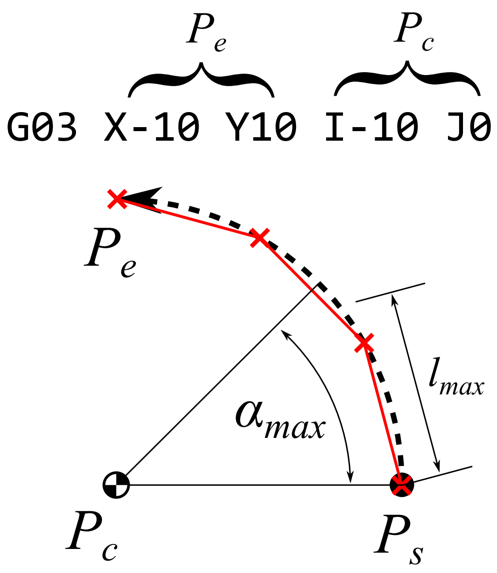


Figure 8.2: Approximating a counter-clockwise arc (`G03`) using a polyline

Because the target shape is a polygon, all arc moves (`G2`, `G3`) must be first broken down into poly-lines. See Figure 8.2 for reference. To achieve this, the number of segments must be determined first. This is decided by two parameters: the maximal segment length l_{max} and maximal angle α_{max}. The smallest integer that satisfies both[2] is used and the arc is broken down to that number of points, evenly distributed along its length. The whole procedure can be found in Algorithm 2.

In machining, it is a convention to cut out the inner holes first to ensure that the part stays in place, and to cut the outer circumference last. Because of this, it is safe to assume that only the last outline is accepted as the representation and all the previous ones can be discarded. It is worth noting that this implementation relies on the fact that only a single part is ever present in the input file.

The parser was implemented as a very simple finite state machine (Figure 8.3) with two states: `pen_up` and `pen_down` with an internal "pen" and "canvas" where the shape is drawn. The parser then scans through the input file and depending on the commands, moves the pen according to the coordinates and puts it "up" or "down" depending on the

[1]But it does not really matter as long as the units are the same in all input files.

[2]For long arcs with large radii, the limiting factor would be the segment length, for smaller arcs, which can be less accurate, it would be the maximal angle.

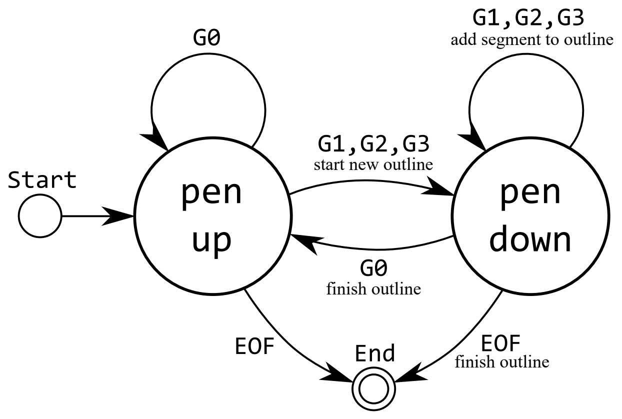


Figure 8.3: The parser as a FSM

type of the move instruction. Whenever the state changes to `pen_down`, a new outline is created, and whenever the state changes to `pen_up`, the currently drawn outline is assumed to be complete. When the parser reaches the end of a file, the last outline is used as the shape's representation.

Algorithm 2 Arc Approximation (assuming CCW rotation)

```
Require: starting point $P_s = [x_s, y_s]$, end point $P_e = [x_e, y_e]$ and center point $P_c = [x_c, y_c]$
Require: parameters $l_{max}$ and $\alpha_{max}$.
Require: function Rec($\rho, \varphi$) that convert polar coordinates to rectangular (Cartesian)

1: procedure ApproximateArc
2:     let points be empty list of points
3:     let $R \leftarrow |P_s P_c|$                              ▷ the radius
4:     let $\phi_{start} \leftarrow \text{arctan2}(y_s - y_c,\ x_s - y_c)$
5:     let $\phi_{end} \leftarrow \text{arctan2}(y_e - y_c,\ x_e - y_c)$
6:     let $\alpha \leftarrow \phi_{end} - \phi_{start}$                    ▷ the total angle
7:     let $l \leftarrow \alpha \cdot R$                             ▷ the lenght of the arc
8:     let $N \leftarrow \lfloor \max\{l/l_{max},\ \alpha/\alpha_{max}\} \rfloor$     ▷ the number of segments
9:     let angles be an evenly distributed set of $N$ numbers from 0 to $\alpha$
10:    for all angle from angles do
11:        let $P \leftarrow$ Rec($R,\ \phi_{start} + angle$)           ▷ point along the arc
12:        append $[P + P_c]$ to points   ▷ account for center of circle not being in origin
13:    end for return points
14: end procedure
```

8.5. Optimiser

The optimisation engine applies the ideas of a First-fit decreasing heuristics, bounding volumes and No-Fit Polygons, and extends them with a custom local placement optimisation method.

Given a container with possible forbidden areas (called *holes* from now on) and a set of shapes, the first task is to determine the order in which to place the shapes. This is decided easily by sorting them by their area in a decreasing manner. This way the smallest shapes are placed last and ideally act like “sand” filling the gaps between previously placed larger shapes.

8.5.1. Initial Placement

The next problem that arises is where and how to place the first shape (and any subsequent one for that matter) and how to determine if the placement is *good*. There is no benefit of placing it in the middle of the available space, so it is obvious that it would be best to place the shape as close to the existing holes or edges of the container as possible. This is where the *hole envelope* and *startpolygon* come into play.

Hole Envelope

The term *hole envelope* is not to be confused with a *bounding volume*; in the context of this chapter, a *hole envelope* refers to a generalized form of the NFP and is a function of two shapes A (fixed) and B (free). If a reference point of the shape B lies outside of the hole envelope, then the two shapes definitely do not overlap, but the opposite implication does not necessarily hold.

The *reference point* of an object was chosen to be the centre of the smallest enclosing circle[1]. This has two benefits:

- Rotating the shape about the reference point produces very small total translation[2] of the rest of its points, which makes *local optimisation* more effective (see Section 8.5.2).
- If a *positive buffer* of the size of the smallest enclosing circle’s radius is applied to a hole, the resulting polygon is guaranteed to be a valid *hole envelope* of the hole with respect to the shape. Furthermore, the hole envelope stays invariant for an arbitrary rotation of the shape.

Although being fast, the method of finding the hole envelope using the smallest enclosing circle is quite inaccurate, especially for shapes that are far from being circular. The best hole envelope of two shapes possible is the No-Fit Polygon (as follows from Chapter 6.2.4) and for this reason, the aforementioned open source C++ library *libnfporb* was leveraged. It implements an orbiting approach algorithm to the creation of a NFP.

This is a complex operation, especially for complicated shapes, but since it is written in a compiled language, the computation is relatively fast. There are some limitations though and the NFP creation sometimes unexpectedly fails for certain (complex) shape combinations. In the case the algorithm falls back to the enclosing circle method. Figure 8.4 highlights the difference between these two methods using a triangle as the reference shape[3].

Due to the complexity of finding NFPs of highly non-convex shapes[4], it is sometimes beneficial to use only the *convex hull* of them. This way the computation is fast, while still being more precise than the smallest enclosing circle method. Two examples of shapes for

[1]Contrary to the first vertex, which is often used in literature

[2]i.e. it stays more “in place” as opposed to when a boundary point was used

[3]Note that the positions of the triangle were chosen arbitrarily for demonstration purposes

[4]The complexity highly depends on the total number of edges and non-convex regions of both shapes

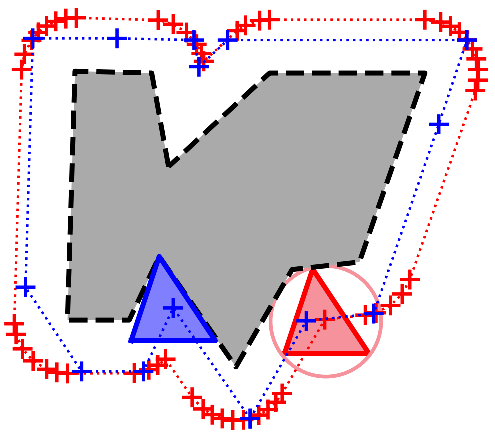


Figure 8.4: Hole envelope using smallest enclosing circle (red) and a NFP (blue)

which the computation of a NFP would be much faster when their convex hull was used can be seen in Figure 8.5.

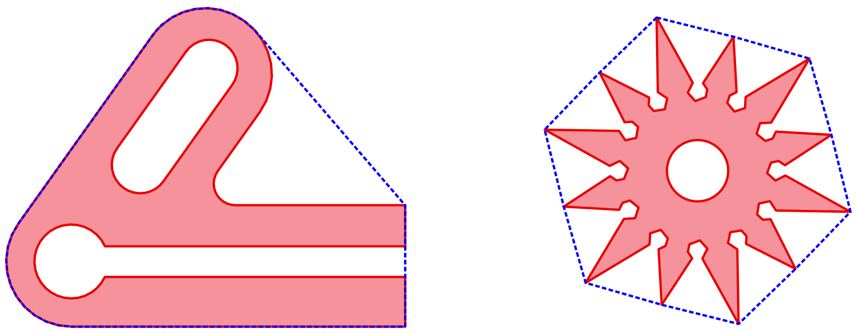


Figure 8.5: Convex hull (· · ·) applied to irregular shapes

As was also stated in Chapter 6.2.4, the resulting NFP is different for any mutual rotation of the shapes. It would be unfeasible to generate NFPs for every hole with respect to all possible rotations of a shape, therefore only a few cases are tested for and the fine-tuning is left to local optimisation.

Given there are often multiple copies of the same shape, the hole envelope does not have to be generated again every time. Each hole holds a cache of all its generated envelopes so far, so when there already exists a hole envelope for the next shape with the corresponding rotation, it is retrieved from the cache, which greatly improves performance.

Startpolygon and Placement Policy

The *startpolygon* is a set of one or more polygons, such that their boundaries represent all viable placement positions of the reference point of the shape in question, considering the whole available area. It is the *union* of all the hole envelopes *subtracted* from the *boundary envelope*.

The boundary envelope is essentially an inverse of the hole envelope. If a reference point of a shape lies within the boundary envelope, it is guaranteed that it is positioned wholly within the container's bounds. In the case of using the smallest enclosing circle as the shape representation, it can be easily constructed by applying a *negative buffer* of the circle's radius to the container's boundary. In the case of using a NFP, the boundary envelope is constructed by shrinking the container on each side by a length equal to the distance between the shape's reference point and an extreme point in the corresponding direction[1].

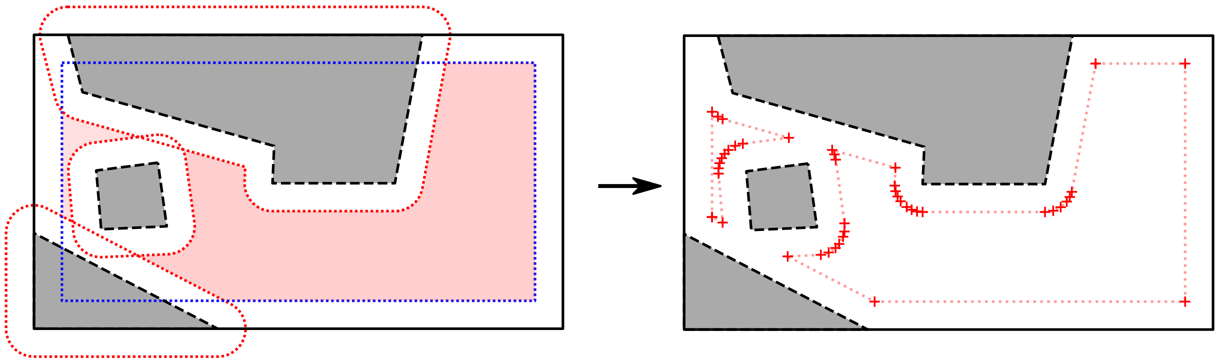


Figure 8.6: The process of creating a startpolygon

Using hole envelopes (red $\cdots$) and a boundary envelope (blue $\cdots$) to obtain all the viable points (red +). The grey areas represent holes; the actual shape of available material is the remaining white region.

The simplest case is when there are no holes in the container, then the startpolygon contains only the boundary envelope. This way the startpolygon consists of four vertices near each corner of the container, these are the *viable* points[2]. Figure 8.6 demonstrates the general case of creating a startpolygon by combining envelopes to obtain the viable points. At which of these the shape will be placed is decided by the *placement policy* function.

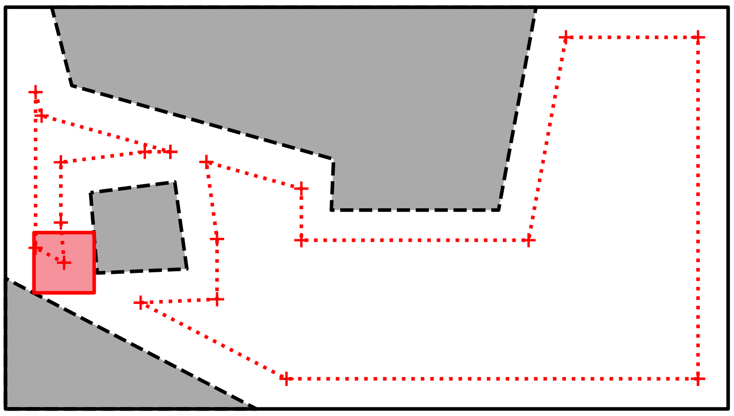


Figure 8.7: Square placed using lower-right placement policy, preferring smaller regions

Most of the time, the placement policy would be set to favour the left-most (or lower-left-most) point, but it can be any function that takes the startpolygon as an argument. It could for example choose the sharpest corner, or in case of multiple starting polygons, choose a point from the smallest ones first, et cetera. Figure 8.7 shows a square placed according to a lower-right placement policy with the preference of filling smaller regions first.

[1]e.g. the right side of the container is shrunk by the difference of the shape's x coordinate of the rightmost point and the x coordinate of its reference point

[2]There is no use in placing the shape along the edges, either

In the case that there is a requirement for a minimal clearance between shapes, holes and the boundary, an additional negative buffer of the required size is applied to the startpolygon prior to applying the placement policy.

When the starting point is selected, the shape's reference point is placed on it and a local optimisation can start (more in Section 8.5.2). When a final location for a shape is found, it is further considered as a hole and a next shape is ready to be placed. Algorithm 3 closely describes the process of placing a single shape.

Algorithm 3 Initial Shape Placement

```
Require: The shape to be placed
Require: Set of holes $H$ and container boundary $B$
Require: Placement policy function $pp(x)$

 1: procedure PlaceShape
 2:     let startpoints be an empty set of all viable starting points
 3:     for all possible rotations of shape do
 4:         let startpolygon be the boundary envelope for the shape
 5:         for all hole from $H$ do
 6:             find the hole envelope for hole with respect to shape
 7:             startpolygon $\leftarrow$difference of startpolygon and hole envelope
 8:         end for
 9:         select a vertex $v$ from startpolygon that minimises $pp(v)$
10:         append $v$ to startpoints, noting the current rotation
11:     end for
12:     from startpoints select again a vertex $v$ that minimises $pp(v)$
13:     return translated and rotated shape according to $v$
14: end procedure
```

8.5.2. Local Placement Optimisation

It is often the case that when a shape is placed in the *initial placement phase*, its sides do not align perfectly with the holes, which wastes the available space, especially when the number of tested rotations is low. To account for the limited number of rotations that can be tested for, a local placement optimisation method was implemented. It trades speed of the calculation for a more precise placement of the shape.

In principle, it works as follows: The goal is to apply small translations and rotations to the shape, such that it maximises the alignment of its sides with the surrounding geometry. To achieve this, it is necessary to determine the metric used to measure the quality of the alignment. Therefore a *near zone* was introduced. Intuitively, a near zone is essentially a buffer around a shape and the *goodness* of a placement can be then measured by how much the near zone overlaps with the nearby holes[1].

This premise alone cannot provide a satisfying result, as the intersection area of the near zone is maximised when the shape is fully enclosed *in a hole*, which is highly undesirable. To counter this, the intersection with holes and the shape itself has to be taken into account. Given that the function $A(s)$ represents the total area of intersection

[1]The bigger the area of intersection, the closer to the surrounding geometry the shape is

of shape s with the surrounding geometry and s_{NZ} is the near zone of shape s. The quality of the alignment of shape s is then defined as

$$Q(s) = \begin{cases} A(s_{NZ}), & \text{if } A(s) = 0, \\ -A(s), & \text{otherwise.} \end{cases}$$

As can be seen, when the shape itself overlaps just a little with any hole, the quality function becomes negative and the solution is rejected.

The position of an object is fully defined by its position in a plane and its rotation (x, y, α), therefore the optimal placement can be found by maximising the *quality of alignment* function over these and becomes a three dimensional maximisation problem. With a slight change in notation, the problem can be defined as

$$(x'_s, y'_s, \alpha'_s) = \underset{x,y,\alpha}{\arg\max}\ Q_s(x, y, \alpha),$$

where x'_s, y'_s, α'_s are the optimal coordinates of the shape and Q_s is the quality function of shape s. In figure 8.8 can be seen an illustrative example of the course of the function along a single axis.

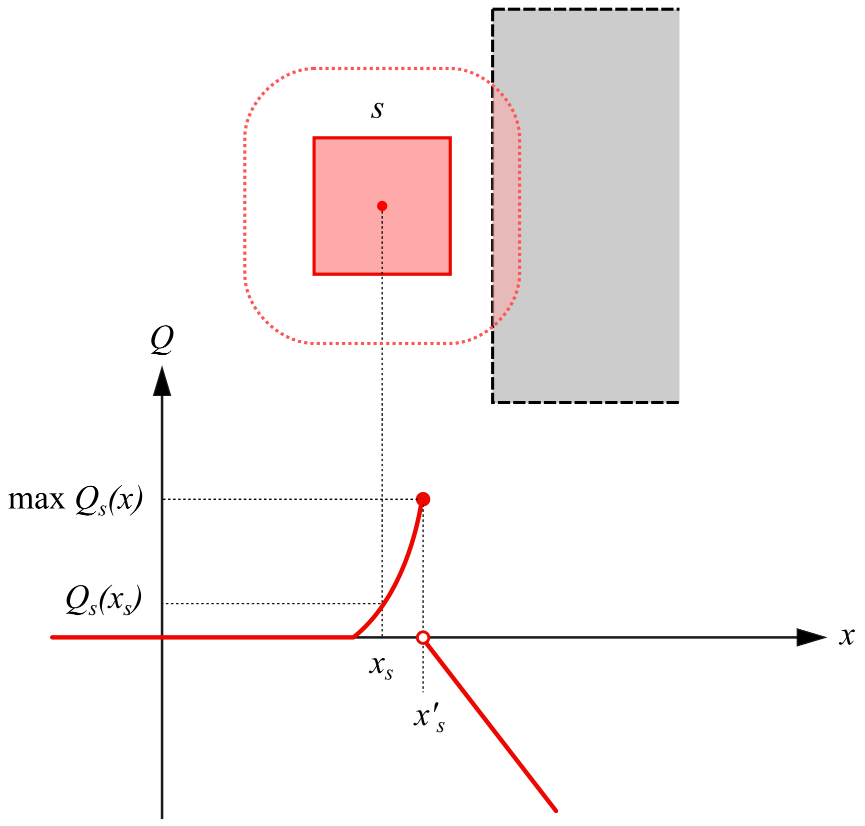


Figure 8.8: 1D demonstration of the quality of alignment function

To solve this problem, a simple greedy hill climbing algorithm was implemented. In each step a number of random neighbouring points is generated around the current position and the one that maximises Q_s is selected. This continues until no better solution can be generated thrice in a row.

As has been mentioned before, the center of the smallest enclosing circle of the shape is selected as the center of rotation. This way, solutions where the shape needs to be only slightly rotated without much translation are “closer” in the solution space to the starting point as opposed to a situation when a boundary point would be used. In other words, this way the solution is found by rotating the shape a bit, whereas otherwise it would have to be rotated by the same amount *and* translated to accommodate for the unwanted shift. Figure 8.9 demonstrates this.

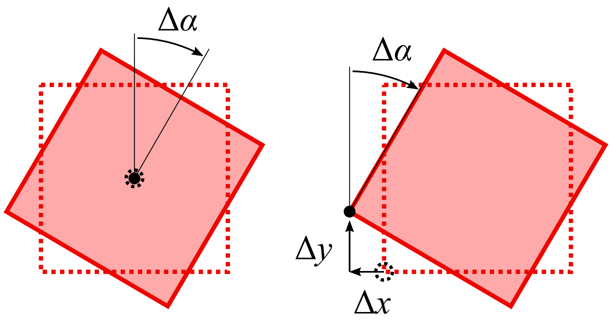


Figure 8.9: The effect of rotating around different reference points

In figure 8.10 can be seen the desired effect of the local optimisation. On the left is the best position found by the initial placement phase and on the right is the optimal position found by local optimisation. Notice how the area of overlap of the near zone gradually increases.

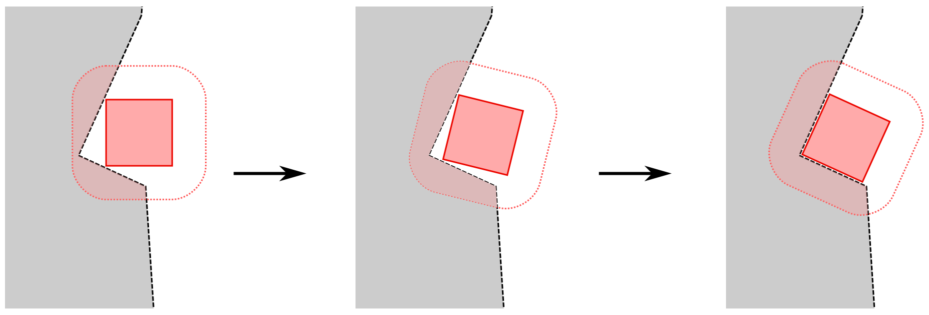


Figure 8.10: The desired effect of local optimisation

8.6. Export

Since the solution is intended to be ready to use, the output format should be in G-Code. The fact that the input files are also G-Codes makes it easy for them to be “glued” together in the resulting file. Each placed shape remembers its original file name, origin point, its position and rotation. When read from left to right as the shapes appear in the container, a rather efficient sequence of cutting plans is created and the original G-Code files are then processed (and duplicated) in that order.

As has been described in Chapter 4 (G-Code), the codes `G68` and `G92` are used for offsetting and rotating of the coordinate system of the machine, this means that the paths in the original files do not need to be altered at all.

The export procedure works as follows: The first G-Code file of shape s is read from the beginning up until the *first* move sequence[1]. Then right before that, the following sequence of commands is inserted

`G92` Xx_{so} Yy_{so}	move the coordinate system to the origin point of shape s
`G00 X0 Y0`	move the tool head there
`G68 X0 Y0` Rα_s	rotate the coordinate system by angle α_s around the origin

Then the file is read normally up until the *last* cutting move. Then the next file in a row is opened and is *skipped* up until the first cutting move and the above command sequence is inserted.

This repeats until the last file in the sequence, which is not disregarded after the last cutting move. Instead the above sequence is added once more, filled with zeros to reset all offsets and rotations. Then the file is read until the end[2]. The resulting combined G-Code file is then ready to be fed to the CNC machine.

8.7. GUI

As has been already noted, the graphical user interface was built using the cross-platform framework *Qt*. Most of the static parts of the interface were laid out using a WYSIWYG editor called *Qt Designer* from which the IU can be exported in the form of a XML markup and then loaded from within Python with the *PyQt5* library. The workspace and preview panes use the *matplotlib* graphical backend.

The GUI consists of several components that provide access to all the key parts of the program. The whole application window can be seen in Figure 8.11.

Input

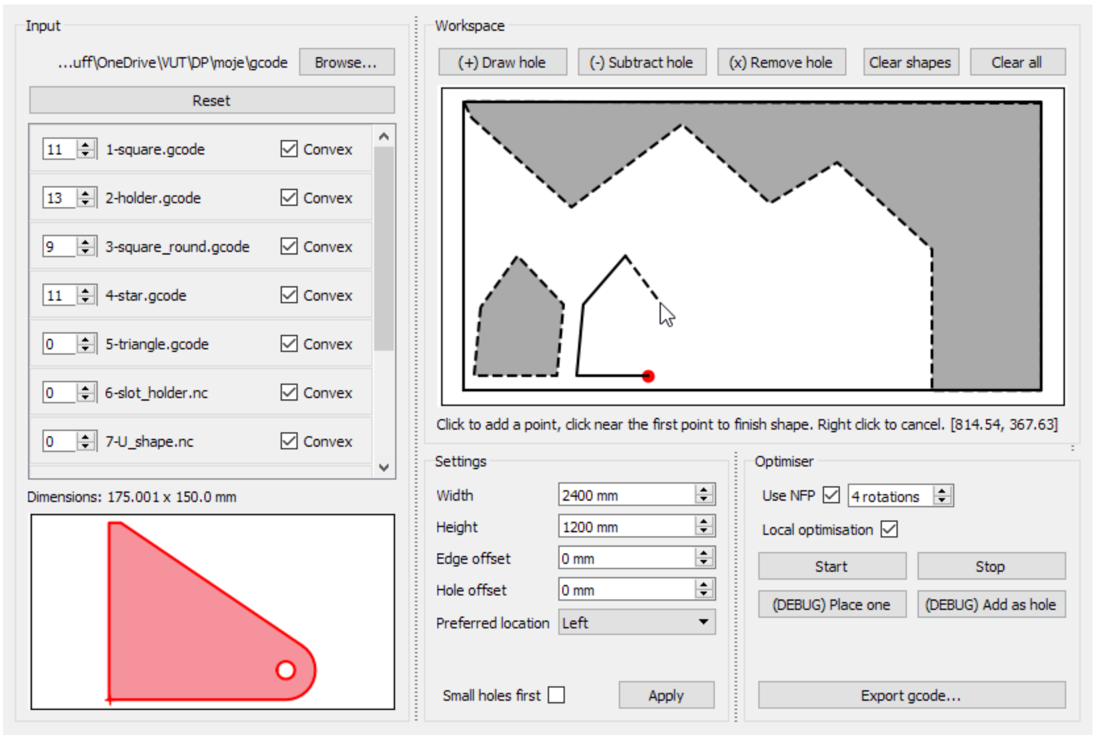
The *input* section allows the user to browse to the folder where the G-Code files are stored, it automatically parses them and displays them in the list below. Upon clicking on any entry, a preview of the selected shape is displayed on the bottom along with its dimensions. Next to each shape in the list is a spin box that allows to set how many copies of the shape should be placed. There is also a check box indicating whether the shape's *convex hull* should be used for reference instead of the true shape (Implications of this were discussed in Section 8.5 (Optimiser)).

Workspace

In the *Workspace*, the user can draw and erase *holes* (forbidden regions, see Figure 8.11) in the material using the provided tools. Once a solution is found, individual shapes can also be removed from it or the whole area can be cleared. The whole workspace can also be exported or imported from a JSON file using the workspace menu item.

[1]To reflect codes needed to setup the machine

[2]To follow the finishing procedure (turn off coolant, etc.)



...uff\OneDrive\VUT\DP\moje\gcode

11	1-square.gcode	☑ Convex
13	2-holder.gcode	☑ Convex
9	3-square_round.gcode	☑ Convex
11	4-star.gcode	☑ Convex
0	5-triangle.gcode	☑ Convex
0	6-slot_holder.nc	☑ Convex
0	7-U_shape.nc	☑ Convex

Dimensions: 175.001 x 150.0 mm

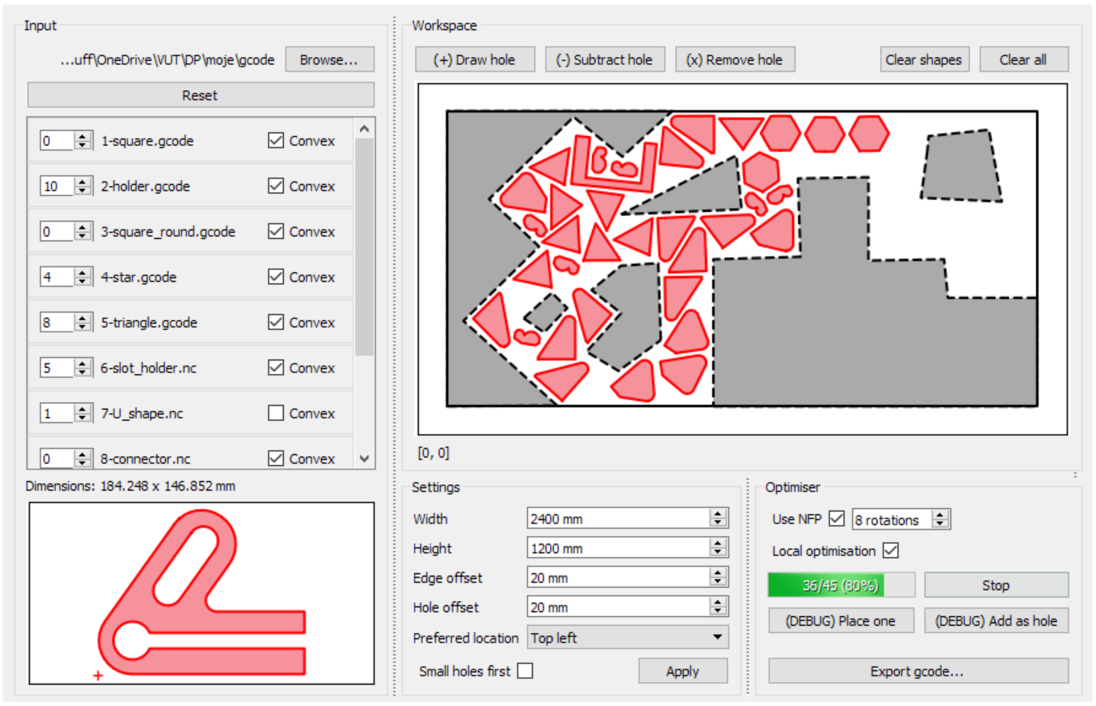
Click to add a point, click near the first point to finish shape. Right click to cancel. [814.54, 367.63]

Width	2400 mm
Height	1200 mm
Edge offset	0 mm
Hole offset	0 mm
Preferred location	Left
Small holes first	☐

Use NFP ☑ 4 rotations

Local optimisation ☑

Figure 8.11: The application window, process of drawing a hole



...uff\OneDrive\VUT\DP\moje\gcode

0	1-square.gcode	☑ Convex
10	2-holder.gcode	☑ Convex
0	3-square_round.gcode	☑ Convex
4	4-star.gcode	☑ Convex
8	5-triangle.gcode	☑ Convex
5	6-slot_holder.nc	☑ Convex
1	7-U_shape.nc	☐ Convex
0	8-connector.nc	☑ Convex

Dimensions: 184.248 x 146.852 mm

[0, 0]

Width	2400 mm
Height	1200 mm
Edge offset	20 mm
Hole offset	20 mm
Preferred location	Top left
Small holes first	☐

Use NFP ☑ 8 rotations

Local optimisation ☑

36/45 (80%)

Figure 8.12: The application window, nesting with clearances in progress

Settings

This section is used to specify the base dimensions of a rectangular working area and the required minimal clearances between shapes and holes or the edges of the workplace. The clearances in effect are illustrated in Figure 8.12. The user can also select the placement policy in a form of the preferred location where to place the shapes, along with the option to first try to fill smaller empty regions prior to placing shapes to the preferred location.

Optimiser

The *optimiser* section provides options to select whether to use the NFP and if so, how many rotations of each shape to test for. A local optimisation can be also enabled here. Upon clicking on the *Start* button, the optimisation algorithm is started in a separate background thread (as to not block the I/O-bound thread and make the GUI unresponsive) and a progress bar is shown indicating the progress, while the shapes are gradually drawn into the workspace. The user can stop the operation anytime. When an arrangement is found the G-Code of it can be exported using the corresponding button.

9. Evaluation

For evaluation purposes, two extensive performance tests were carried out. The goal was to determine relationships between different settings and conditions. Results of all the benchmarks can be found in the attached archive.

In literature, the nesting quality is often measured as the ratio between the total area of the container and the total area of shapes (container utilization). This is only applicable in case of strip-packing problem, as the container's size dynamically changes. Some implementations "virtually" shrink the container from one side or use the x-coordinate of the rightmost placed shape to get around this limitation.

This is also not applicable, as the placement policy could position the shapes quite anywhere. Therefore, after all shapes have been placed, the quality of nesting is measured as the ratio of the area of a startpolygon of the last placed shape[1] and the total available are at the beginning. This is demonstrated in Figure 9.1.

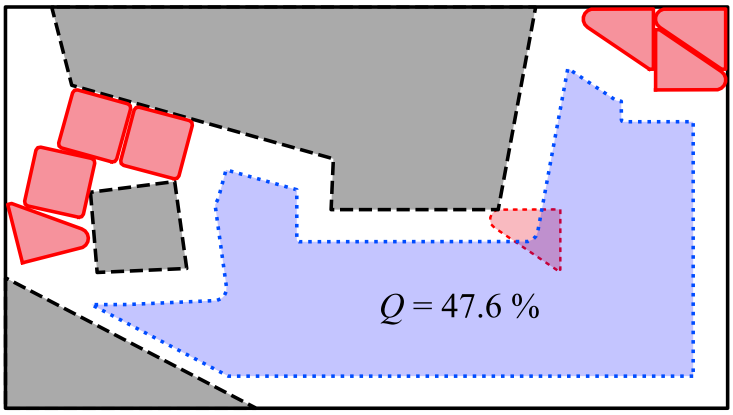


Figure 9.1: Startpolygon as a measure of nesting quality

9.1. Benchmark 1

The first benchmark consisted of 31 relatively simple shapes with small number of edges, as seen in Figure 9.2. The goal was to determine the relationship between the nesting quality and the speed, with respect to the method of creating a startpolygon and the use of local optimisation (LO). The placement policy was set to place the shapes as much to the left as possible. The shape of the container was the same as seen in Figure 9.1 and was used for both benchmarks.

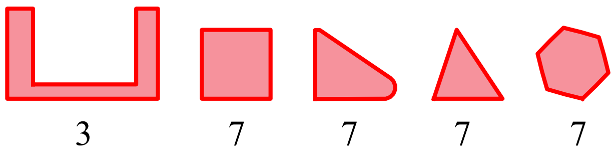


Figure 9.2: Benchmark set 1

As shown in Table 9.1, six cases of startpolygon generation were examined, each one with local optimisation disabled and enabled. The first case (denoted "-") was the

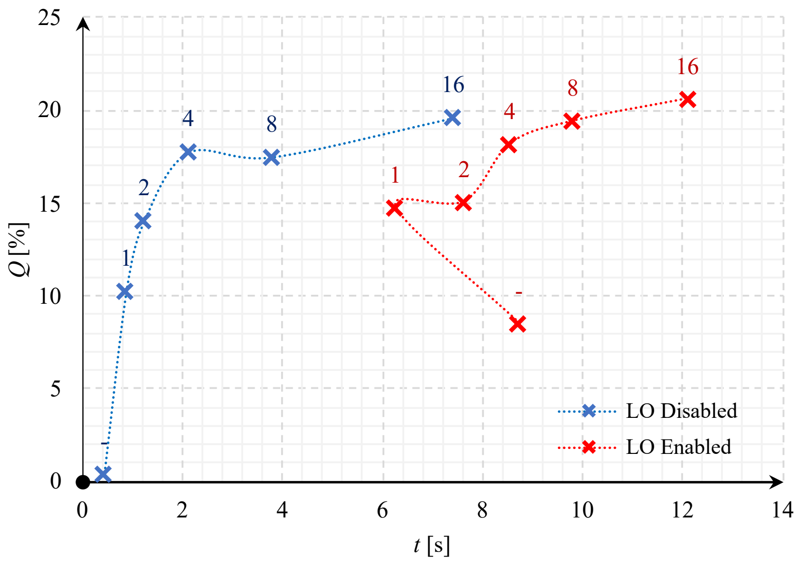
[1]As if one more copy of it would be placed

smallest enclosing circle method without the use of NFP. The other are cases of different numbers of NFP rotations. Each case has a corresponding nesting quality (denoted Q) and time to completion (t), the best and worst values are in bold. The values are visualized in Figure 9.3[1]. The last two columns indicate the relative increase in quality (I_Q) and computation time (I_t) between cases without and with the help of LO.

Table 9.1: Benchmark 1 results

NFP	Local optimisation				Increase	
	Disabled		Enabled			
rotations	Q [%]	t [s]	Q [%]	t [s]	I_Q [%]	I_t [%]
-	**0.4**	**0.4**	8.5	8.7	1914.0	1903.1
1	10.3	0.9	14.7	6.3	43.4	615.8
2	14.1	1.2	15.1	7.6	7.1	517.3
4	17.7	2.1	18.2	8.5	2.5	303.4
8	17.5	3.8	19.4	9.8	10.9	159.0
16	19.6	7.4	**20.6**	**12.1**	5.0	63.2

Figure 9.3: Benchmark 1 results



As can be seen, with the use of LO, the total computation time is longer by approximately the same amount in all cases, but the relative increase is large. This is especially apparent with cases where the initial Q was low[2]. Notably, in the first case, the computation time grew almost twenty-fold, for the price of a 20 times better solution. With more shape rotations to test for, more places become available for the next shape and the increase in nesting quality by using LO is less visible, but the extra time is less noticeable, too. Some notable examples can be seen in Figure 9.4

[1]Note that the connecting lines carry no information value. They are included only for a better readability of the graph.

[2]A bad solution was found very fast.

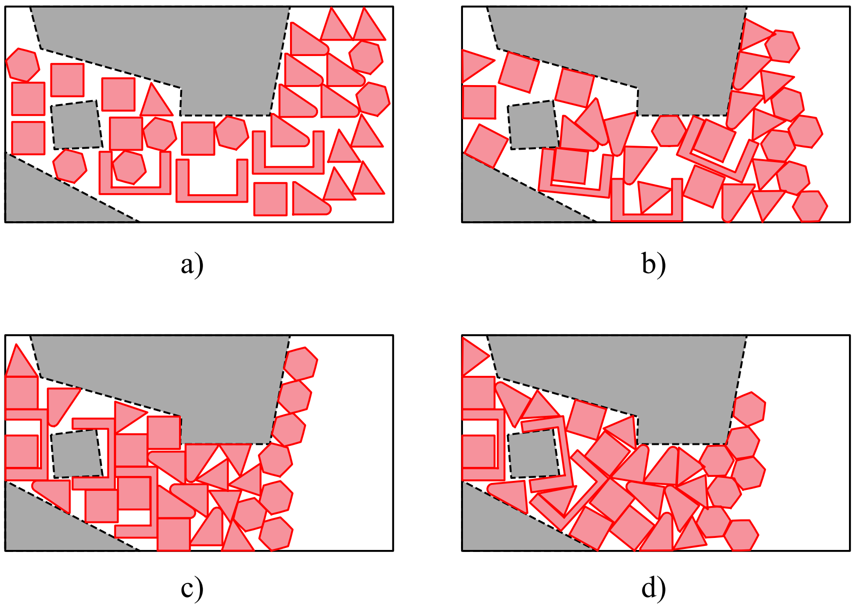


Figure 9.4: Notable examples of benchmark 1

a) smallest enclosing circle method *without* LO (worst),
b) smallest enclosing circle method *with* LO,
c) 4 NFP rotations *without* LO,
d) 16 NFP rotations *with* LO (best)

It is apparent from the figure that LO helps especially when the initial placement is bad, but it still cannot improve it that much. Also, if the shapes are somewhat regular and fit together, the solution without the use of LO looks neater, even though the measured quality is lower.

9.2. Benchmark 2

The second benchmark focused on the difference in complexity of finding the NFP for irregular shapes. The set of the shapes can be seen in Figure 9.5. This time, the square was replaced with rounded square[1] and a new non-convex shape with an inner cavity was added. The point of this benchmark was to find out how this affects the performance of finding the FNPs. There are 2 main scenarios: In the first, the shapes are used as shown and in the second, the complex shape (second from right) is replaced by its convex hull. Furthermore, the placement policy was changed to favour the top-right corner with the addition of placing shapes in smaller regions first.

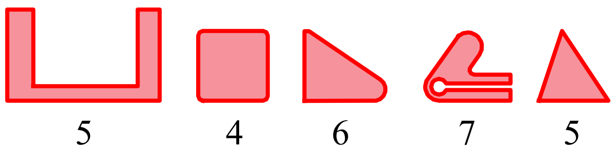


Figure 9.5: Benchmark set 2

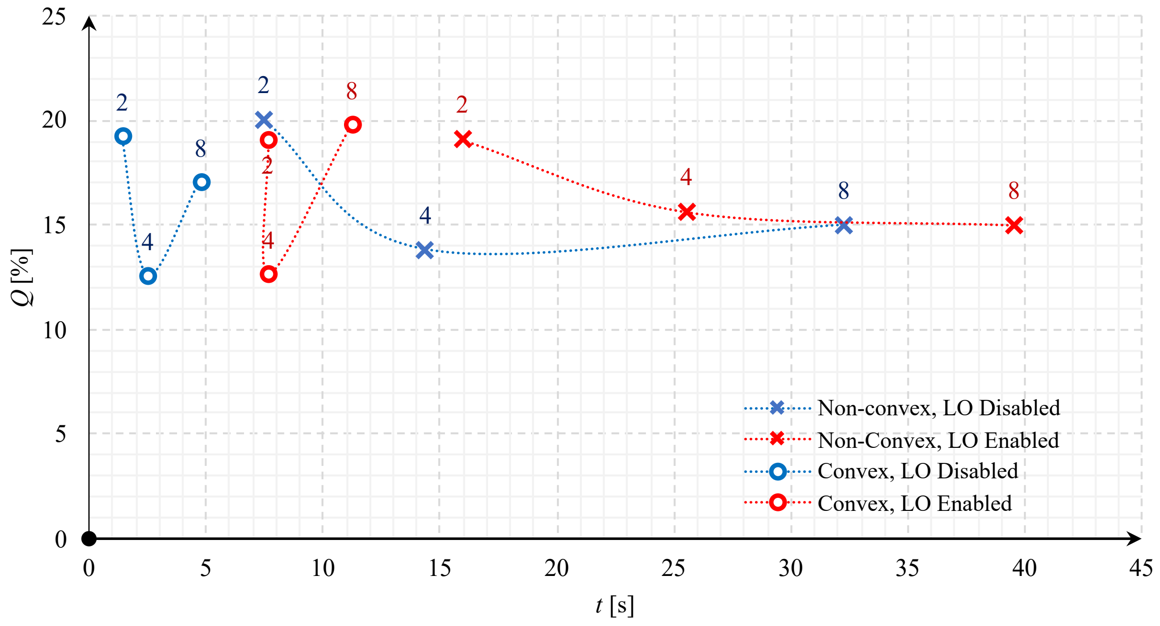
[1]This increases the edge count

Table 9.2: Benchmark 2 results

	NFP rotations	Local optimisation Disabled Q [%]	Disabled t [s]	Enabled Q [%]	Enabled t [s]	Increase I_Q [%]	Increase I_t [%]
Non-convex	2	**20.0**	7.5	19.1	16.0	-4.7	113.6
	4	13.8	14.4	15.6	25.5	13.2	77.5
	8	15.0	32.3	15.0	**39.5**	0.2	22.6
Convex	2	19.3	**1.4**	19.1	7.6	-1.1	437.5
	4	**12.6**	2.5	12.7	7.7	0.7	205.0
	8	17.1	4.8	19.8	11.2	15.7	135.5

The results of this can be found in Table 9.2 and they are quite surprising. Not only are the best results achieved by using only two rotations, but also in this case the use of LO leads to a *worse* nesting quality. In addition, the fastest solution was also the third-to-best (without a major difference), taking only 1.4 s, while the slowest one was on the worse side.

Figure 9.6: Benchmark 2 results



Looking at Figure 9.6, it is apparent that there is no significant difference between the nesting quality with and without local optimisation. What is most noticeable, is the increase of computation times in the non-convex scenario as the number of rotations increase.

What is odd, though, is that most of the time 4 rotations perform the worst of all. Looking at some of the results in Figure 9.7, this phenomenon can be explained by the greedy nature of the algorithm. When the shapes themselves fit well together, using only 2 rotations forces the algorithm to place the shape to a locally worse position, but the next shape, rotated 180°, will fit perfectly, leaving almost no gap. Furthermore, given the shapes fit together, there is little to no benefit of adjusting their final position just a bit, because it lowers the change that the next shape will fit well.

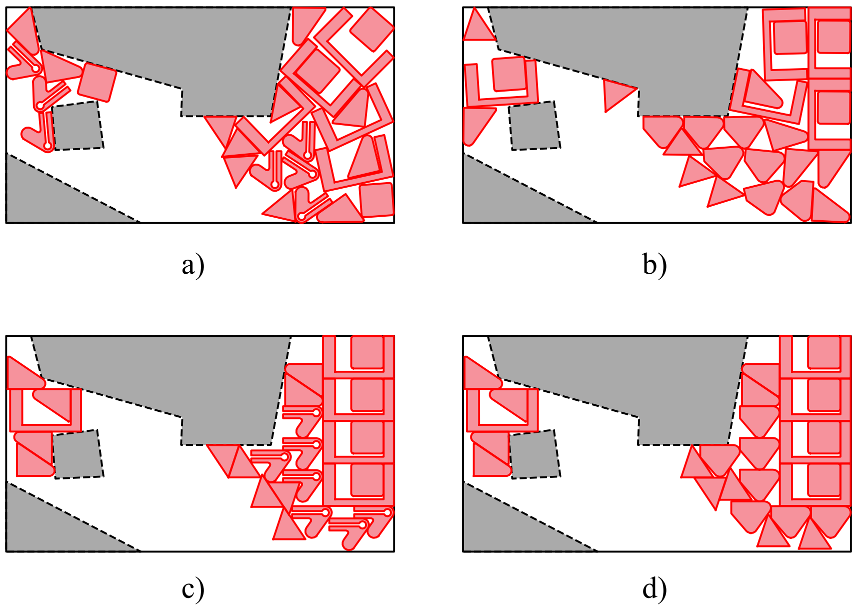


Figure 9.7: Notable examples of benchmark 2
a) 8 NFP rotations, non-convex, with LO (slowest),
b) 4 NFP rotations, convex, with LO (second-to-worst),
c) 2 NFP rotations, non-convex, without LO, (best)
d) 2 NFP rotations, convex, with LO (fastest, third-to-best)

It appears that the placement policy also plays an important role in the nesting quality. In the second benchmark, the shapes were preferably placed near a right-angled corner where the holes did not affect the placement much. Whereas in the first benchmark, they were placed preferably in the irregular part of the container, making local optimisation much more useful. Therefore, when operating the program, the user should use their best judgement and setup the process accordingly. In general, more regular containers allow for lower number of rotations and a faster solution, whereas irregular ones do benefit from the nesting quality added by local optimisation for the price of a longer computation.

9.3. Improvements and Future Work

It is clear that at this point, the software is capable of arranging irregular shapes into an irregular container in a somewhat satisfying manner. However, the algorithm is still far from perfect. Due to the greedy (first fit) nature of the algorithm, much of the solution space remains unexplored. It would greatly benefit from a global optimisation method, where the order of placement was iteratively altered to find even better solutions. This could be achieved for example by implementing a genetic algorithm where the order of placement was the genome.

Also the local placement optimisation could be improved by applying for example the simulated annealing method, to avoid local optima (e.g. if a square, rotated by 45°, is placed in a corner, it lies in a local optimum and the current LO is unable to align it properly). This is especially noticeable in Figure 9.7 a).

Currently, the only method of specifying the container's shape is either by drawing the holes by hand or by importing the board dimensions and holes from a JSON file. To maximise the functionality, the program could be paired with a computer vision software that is able to recognize the shape of the material directly on the machine. That way, the operator could just select the shapes to cut and "throw" a piece of material into the machine. The program would automatically extract its geometry, arrange the shapes accordingly, and generate the appropriate tool paths.

A less automatic but still a great solution would be to display the image from the camera in the background of the workspace area, so that the operator could easily trace the holes by hand.

10. Conclusion

This Master's thesis covered a basic theoretical background needed to understand the Nesting problem. It outlined the different methods of 2D shape representation and spatial analysis and gave an insight on various methods of approaching the Nesting problem, along with examples of the existing solutions.

The practical part of this thesis describes the implementation of a nesting software, capable of effectively placing arbitrary two-dimensional objects into a bounded region with respect to prohibited areas and other already placed objects. This was achieved by using a combination of both pre-existing solutions and own invention.

The program is presented as a module of the programming language Python and can be used either as a graphical user interface application, or incorporated as an API into any other software. Due to the nature of Python, it is directly runnable on any platform that supports this programming language. The additional open-source library and its Python interface can be also easily compiled for the target machine.

The software takes advantage of the No-Fit Polygon that is used to find regions of no overlap between shapes, with the addition of a local placement optimisation that furthermore fine-tunes the placement of a shape.

An extensive performance test was carried out, investigating different scenarios and placement policies, and evaluating the results. It has been discovered, that for irregular shapes and container, the local optimisation method does enhance the solution, whereas more regular shapes benefit from being constrained to a small number of possible rotations.

Although the solutions produced by the program cannot be considered optimal, they are objectively not bad, and at this point, the program is absolutely usable by the end user.

Bibliography

[1] All3DP. *CNC Plasma Cutting – The Basics* [online]. Jan 2019. [cit. 11. 6. 2020]. Available at: < `www.all3dp.com/2/cnc-plasma-cutting-all-you-need-to-know/` >.

[2] Autodesk. *TruNest: Automated Nesting Software* [online]. Available at: < `https://www.autodesk.com/products/trunest/overview` >.

[3] Baker, B. – Coffman, E. – Rivest, R. Orthogonal Packings in Two Dimensions. *SIAM J. Comput.* 11 1980, 9, p. 846–855. doi: 10.1137/0209064.

[4] Bengtsson, B.-E. Packing Rectangular Pieces—A Heuristic Approach. *The Computer Journal.* 08 1982, 25, 3, p. 353–357. ISSN 0010-4620. doi: 10.1093/comjnl/25.3.353. Available at: < `https://doi.org/10.1093/comjnl/25.3.353` >.

[5] Benne. *Building a CNC Router* [online]. Instructables, May 2019. [cit. 11. 6. 2020]. Available at: < `www.instructables.com/id/Building-a-CNC-router/` >.

[6] Bennell, J. A comprehensive and robust procedure for obtaining the nofit polygon using Minkowski sums. *Computers & OR.* 01 2008, 35, p. 267–281. doi: 10.2139/ssrn.766146.

[7] Boost C++ Libraries. *Boost.geometry* [online]. Aug 2017. [cit. 15. 6. 2020]. Available at: < `www.boost.org/doc/libs/1_65_1/libs/geometry/doc/html/index.html` >.

[8] Burke, E. et al. Complete and robust no-fit polygon generation for the irregular stock cutting problem. *European Journal of Operational Research.* 05 2007, 179, p. 27–49. doi: 10.1016/j.ejor.2006.03.011.

[9] CGAL Editorial Board. *The Computational Geometry Algorithms Library* [online]. Feb 2020. [cit. 15. 6. 2020]. Available at: < `https://www.cgal.org/` >.

[10] Chen, P. et al. Two-dimensional packing for irregular shaped objects. *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the.* 2003, p. 10 pp.–.

[11] CutMaps. *Laser Cut Street Map by CutMaps* [online]. Youtube, December 2012. [cit. 11. 6. 2020]. Available at: < `www.youtube.com/watch?v=WdJ4KranBcw` >.

[12] Davis M., A. J. Java Topology Suite, Technical Specifications. [online], March 2003. Available at: < `https://github.com/locationtech/jts/blob/master/doc/JTS%20Technical%20Specs.pdf` >.

[13] Deepnest. *Open source nesting software* [online]. Available at: < `https://deepnest.io/` >.

[14] Douglas, D. H. – Peucker, T. K. Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*. Dec 1973, 10, 2, p. 112–122. Available at: < `https://doi.org/10.3138%2Ffm57-6770-u75u-7727` >.

[15] Dósa, G. – Sgall, J. First Fit bin packing: A tight analysis. *LIPIcs*. 01 2013, 20, p. 538–549. doi: 10.4230/LIPIcs.STACS.2013.538.

[16] Egeblad, J. – Nielsen, B. K. – Odgaard, A. Fast neighborhood search for two- and three-dimensional nesting problems. *European Journal of Operational Research*. December 2007, 183, 3, p. 1249–1266. doi: 10.1016/j.ejor.2005.11.063. Available at: < `doi.org/10.1016/j.ejor.2005.11.063` >.

[17] Elastic. *Geo-Shape datatype: Elasticsearch Reference [6.2]* [online]. [cit. 15. 6. 2020]. Available at: < `https://www.elastic.co/guide/en/elasticsearch/reference/6.2/geo-shape.html` >.

[18] GEOS. *Geometry Engine, Open Source* [online]. Mar 2020. [cit. 15. 6. 2020]. Available at: < `https://trac.osgeo.org/geos/` >.

[19] Gillies, S., Jun 2020. Available at: < `https://shapely.readthedocs.io/en/latest/manual.html` >.

[20] Gordon, J. *Binary Tree Bin Packing Algorithm* [online]. May 2011. [cit. 17. 6. 2020]. Available at: < `www.codeincomplete.com/articles/bin-packing/` >.

[21] Hassan, A. libnfporb, May 2018. Available at: < `https://github.com/kallaballa/libnfporb` >.

[22] Hunter, J. D. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*. 2007, 9, 3, p. 90–95. doi: 10.1109/mcse.2007.55.

[23] Jakob, W. – Rhinelander, J. – Moldovan, D. pybind11 — Seamless operability between C++11 and Python, 2017. Available at: < `https://github.com/pybind/pybind11` >.

[24] Junior, B. A. – Pinheiro, P. R. – Saraiva, R. D. A Hybrid Methodology for Nesting Irregular Shapes: Case Study on a Textile Industry∗. *IFAC Proceedings Volumes*. September 2013, 46, 24, p. 15–20. doi: 10.3182/20130911-3-br-3021.00056. Available at: < `https://doi.org/10.3182/20130911-3-br-3021.00056` >.

[25] Lott, R. *Geographic information - Well-known text representation of coordinate reference systems* [online]. Open Geospatial Consortium, Jul 2013. [cit. 15. 6. 2020]. Available at: < `http://docs.opengeospatial.org/is/12-063r5/12-063r5.html` >.

[26] Luminoso, L. *Understanding the power of wire EDM* [online]. Mar 2019. [cit. 11. 6. 2020]. Available at: < `www.canadianmetalworking.com/canadianmetalworking/article/metalworking/understanding-the-power-of-wire-edm` >.

[27] MyNesting. *Powerful Low-Cost Nesting* [online]. Available at: < https://www.mynesting.com/ >.

[28] NestFab. *Powerful Automatic Nesting Software* [online]. Available at: < https://www.nestfab.com/ >.

[29] Nielsen, B. K. – Odgaard, A. *Fast neighborhood search for the nesting problem.* Technical report, Department of Computer Science, University of Copenhagen, 2003. Technical Report 03/03.

[30] Oliphant, T. E. A guide to NumPy. *Trelgol Publishing, USA*. 2006.

[31] Python Software Foundation. Python Language Reference. Available at: < http://www.python.org/ >.

[32] Qiao, J. *SVGnest* [online]. 2015. [cit. 9. 6. 2020]. Available at: < www.svgnest.com >.

[33] Roland DG. *NC Code Reference Manual* [online]. Roland DG Corporation. [cit. 11. 6. 2020]. Available at: < http://docplayer.net/34529735-Nc-code-reference-manual.html >.

[34] SAKHARE, U. *The History of Computer Numerical Control (CNC)* [online]. Jun 2020. [cit. 11. 6. 2020]. Available at: < www.cnc.com/the-history-of-computer-numerical-control-cnc/ >.

[35] Schmidt, M. *Water Jet Cutting* [online]. 2016. [cit. 11. 6. 2020]. Available at: < www.michaelschmidtstudios.com/services/water-jet-cutting >.

[36] SigmaTEK. *SigmaNEST Nesting Software: CAD/CAM: Automation* [online]. Jun 2020. Available at: < https://www.sigmanest.com/ >.

[37] The Qt Company. Qt Documentation, 2019. Available at: < https://doc.qt.io/ >.

[38] Tools, B. *History of CNC Machining, Part 2:* [online]. CNC Life, Apr 2019. [cit. 11. 6. 2020]. Available at: < https://medium.com/cnc-life/history-of-cnc-machining-part-2-the-evolution-from-nc-to-cnc-4b9fe1653536 >.

[39] TROTEC. *JobControl laser software with maximum operator comfort* [online]. May 2020. [cit. 10. 6. 2020]. Available at: < www.troteclaser.com/en/laser-machines/laser-software/jobcontrol/ >.

[40] WARDJet. *Waterjet vs Laser vs Plasma vs EDM Cutting System Comparison* [online]. May 2016. [cit. 11. 6. 2020]. Available at: < www.wardjet.com/news/waterjet-laser-plasma-wire-edm >.

[41] Welzl, E. Smallest Enclosing Disks (balls and Ellipsoids). In *Results and New Trends in Computer Science*, p. 359–370. Springer-Verlag, 1991.

[42] Wikipedia. *G-code* [online]. Wikimedia Foundation, Jun 2020. [cit. 10. 6. 2020]. Available at: < en.wikipedia.org/wiki/G-code >.

List of Figures

List of Abbreviations and Symbols

API	Application programming interface
EOF	End of file
FSM	Finite state machine
GIS	Geographic information system
GUI	Graphical user interface
CAD	Computer-aided design
CAM	Computer-aided manufacturing
CNC	Computer numerical control
I/O	Input/Output
LO	Local optimisation
NFP	No-Fit Polygon
WKT	Well-Knows Text
WYSIWYG	What you see is what you get

Contents of Electronic Attachment

The attached ZIP archive contains the following folders and files:

Item	Content
`benchmarks/`	All results and images from the benchmarks
`gcodes/`	Sample G-Code files
`screenshots/`	Images of the GUI
`wasteoptimiser/`	The actual nesting software implementation
`README.txt`	Guide to using the software
`README.pdf`	PDF version of the guide
`run.py`	Start-up Python script