Lycée classique de Diekirch
32 av. de la gare L-9233 Diekirch

Development of a LabVIEW G-Code generation program for 2.5D CNC contour offset toolpaths

Claude BAUMANN (COMPUTARIUM)

Version 1.0
Last edited: October 31, 2023 (17:28)

Abstract

This lab report keeps track of the development of a NI LabVIEWprogram that should be able to generate G-Code contour (inside, outside, offset) and pocket milling toolpaths starting from 2D **Scalable Vector Graphics (SVG)** exclusively based on cubic **Bézier** curves and simple lines.

History:

- Version 1.0 October 31, 2023

Contents

1 Project description

1.1 Preliminary note

All the program diagrams presented in this document are supposed to evolve over the development time line. Diagrams shown here are reproduced for explanation and documentation. Code will receive version numbers and documentation.

1.2 Introduction

This report is a direct follow-up to the 2021 document entitled *Bringing back to life a Colinbus Profiler CNC-Router* (pp. 63-74) available at the COMPUTARIUM repository.[1] These indicated pages presented a first and naive study of the router toolpath problem with the following aspects:

1. constant distance offset path
2. local toolpath collisions in concave curve parts due to non-zero tool diameter

[1] https://computarium.lcd.lu/literature/COMPUTARIUM_CREW/BAUMANN/COLINBUS_restoration_lab_report3_29MAR21.pdf

3. global toolpath collisions and intersections appearing in complex curve forms

4. necessity of cusp circumvention

The naive approach started from a curve approximation by polygons, which ended in an impasse.

We started scrutinizing a list of miscellaneous software tools that should help us generate the required G-Code for our GRBL controlled CNC router, some of which are available online like https://jscut.org/, a CAM in a browser project, for instance. Non of the programs satisfied us for our specific applications, either being too costly, or way too complex with killer learning curves, some suffering from hidden issues like imprecision, missing or bad functions for the placement of tabs, for example. The most annoying thing is that all of them work like black boxes, so that you are sometimes just guessing, why things don't work as expected, and worse, why they do work, if some magic has been applied.

Finally, we decided to conceive our own program that we could easily adapt to particular challenges. Because of the available powerful and easy-to-use libraries and of the excellent features of the LabVIEW environment, we also decided to create the program with this sophisticated and compelling tool.

The contour offset curve (also called parallel curve) problem, is an astonishingly complex challenge. Regarded the non-trivial mathematical background required, the main question arises, whether the programmatic implementation of a purely mathematical method is worth the effort, given the complexity and the uncertain added-value to execution speed compared to a pragmatic KISS solution that resolves all the encountered issues on the fly during code conversion with minimal math implication.
In order to find out, which method will produce the better result, we need to plunge into the problem and try to understand, which elements are at the root of the difficulties, and how they can be avoided or resolved. We will develop a first program trying to identify *a priori* the critical points and also provide solutions to each category. A second program will follow a direct method, where the offset path is generated while checking if there is a collision somewhere that should be avoided.

1.3 Documentation

Interestingly, our Internet research didn't show up any precise details, how people out there have solved the three enumerated problem aspects. Some principles and hints are well to be found. However, we groped in the dark, as to implementation methods, although we got excellent basic knowledge from:

1. Duncan March, *Applied Geometry for Computer Graphics and CAD*, Springer, London, (2005): Homogeneous coordinates, transformations, curves, Bézier curves, curvature, ...

2. Frank Morley, F. V. Morley, *Inversive Geometry*, Dover edition, US, (2014, original 1933): algebraic geometry, curvature of a path, etc.

3. Dennis G. Zill, Patrick D. Shanahan, *A First Course in Complex Analysis With Applications*, Jones and Bartlett Publishers, Sudbury, MA, (2009): complex numbers, complex plane, functions and mappings.

4. H.S.M. Coxeter, *Unvergängliche Geometrie*, Bikhäuser Verlag, Basel, (1963): affine geometry, ...

5. Max Koecher, *Lineare Algebra und analytische Geometrie*, Springer, Berlin, (1992).

6. J.C. Binz, U. Friedli, *Vektorgeometrie*, Orell Füssli, Zürich, (1981).

7. Molina-Carmona, R., Jimeno, A. & Davia, M., *Contour pocketing computation using mathematical morphology*, Int. J. Adv. Manuf. Technol., Vol. 36, 334–342 (2008). https://doi.org/10.1007/s00170-006-0823-9.

8. C. M. Hoffmann, *Conversion Methods Between Parametric and Implicit Curves and Surfaces*, Perdue University, (1990) https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1827&context=cstech.

9. D. Lasser, *Calculating the Intersections of Bézier Curves*, Computers in Industry, Vol. 12, Issue 3, (1989), pp. 259-268, ISSN 0166-3615,https://doi.org/10.1016/0166-3615(89)90072-9.

10. Gershon Elber, In-Kwon Lee, and Myung-Soo Kim, *Comparing Offset Curve Approximation Methods*, IEEE Comput. Graph. Appl., Vol. 17, Issue 3, (May 1997), pp 62–71, https://doi.org/10.1109/38.586019.

11. Nicholas M. Patrikalakis, Takshi Maekawa, Wonjoon Cho, *Shape Interrogation for Computer Aided Design and Manufacturing*, Springer Berlin, Heidelberg, (2002), Hyperbook Edition, https://web.mit.edu/hyperbook/Patrikalakis-Maekawa-Cho/mathe.html, (2009): Bernstein polynomials, and much more.

12. https://math.stackexchange.com/questions/3776840/2d-cubic-bezier-curve-point-of-self-intersection

13. https://cp-algorithms.com/geometry/circle-line-intersection.html: extraordinary site with plenty of math with C-code examples.

14. L. Roberts, *Machine Perception of Three-Dimensional Solids*, MIT, doctoral thesis, MA, (193), Download thesis: Apparently first use of homogeneous coordinates in computer graphics.

15. H. Henkel, *Calculating the Cubic Bézier Arc Length by Elliptic Integrals*, [2014] http://www.circuitwizard.de/metapost/arclength.pdf, [retrieved Oct. 2023]

2 Minimal svg text

This project will only work with a certain **svg** text structure, as it is generated by Inkscape for example, as shown in Listing 1. It is NOT planned to build or use a complete **svg** analyzer. The user will have to provide the **svg** files strictly under these conditions.

Listing 1: **svg** text structure that must be minimally observed

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<svg height="210" width="500">
<g>
  <path d="M 0,0 L 200,200"/>
</g>
</svg>
```

Processed commands or attributes:

- height, width *with or without unit specification mm or, inch*
- transform
- path d="..." *with the following legal commands*
  - M, L, H, V, C, Z, m, l, h, v, c, z
  - decimal point is *dot*, coordinate separator is *comma*
  - a path may have a single affine transformation only (rotation, translation, scale, skew, or matrix)
  - a group may also have a single transformation

Rules:

- Version 1.0 of CAM software doesn't care about height and width
- There may only be a single path (initial program version 1.0)
- There may only be one M (m) command in a path[2]
- There may only zero or one single Z (z) command at the end of the path

[2]Sometimes Inkscape adds several M (m) commands right at the beginning, causing software version 1.0 to throw an error message. The user can easily merge these commands manually to a single move.

3 Notation

- $\mathbf{p}$ represents a point in the Cartesian plane with coordinates (x, y)
- $\boldsymbol{z} = x + iy$ is the corresponding complex number
- $\vec{p} = (x, y)$: for editing ease we use horizontal vector notation without transpose symbol. (Of course, this leads to an alternate representation of matrix equations.)
- $\mathbf{C}(t) = \{\mathbf{p}(t)\} = \{(x(t), y(t))\}$ represents a parametric curve
- we admit in the whole document that the variable $t \in [0, 1]$
- x' represents the first derivative $\frac{dx}{dt}$ with the parameter t as the independent variable.

4 Cubic Bézier curves

Curved shapes designed for machining are in many cases based on cubic Bézier curves. This special class of parametric curves are easy to handle because of numerous features:

4.1 Properties

Preliminary note: Most of the following properties can be generalized for degree $n \geq 1$. Because this project is based on cubic Bézier curves, we will focus on the study of this particular case. The following property list has been composed and adapted from [1, pp. 137-167].

- *Cubic* Bézier *curves can exhibit loops, sharp corners (called cusps) and inflections* cf. [1, p.138] and Fig.1 & 2.
- Cubic Bézier curves are entirely described by 4 control points $\mathbf{b_0}..\mathbf{b_3}$, where $\mathbf{b_0}$ and $\mathbf{b_3}$ are located on the curve (in fact the starting and ending points of the curve).

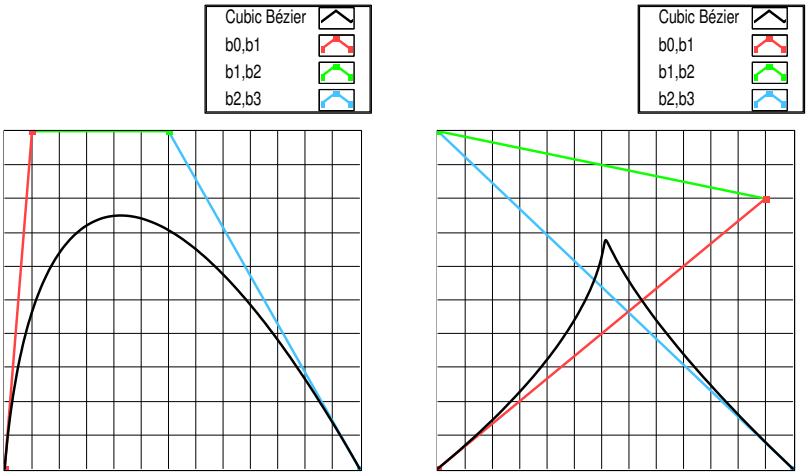


Figure 1: Left: Convex Bézier curve with its 4 control points; Right: crossed control segments → cusp.

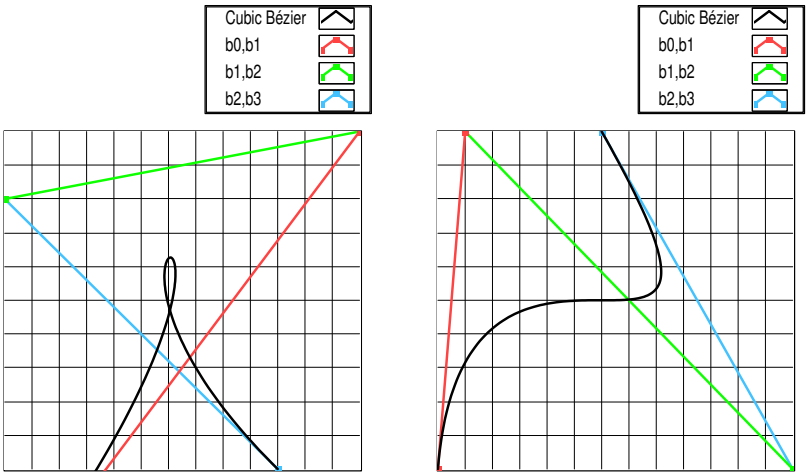


Figure 2: Left: loop representing a curve self-intersection; Right: changing the order of the control points generates opposite concavities.

- Bézier curves are based on the **Bernstein polynomials** with degree n:

$$B_{i,n}(t) = \begin{cases} \frac{n!}{(n-i)!i!}(1-t)^{n-i}t^i, & \text{if } 0 \leq i \leq n \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

  More particularly:

$$B_{0,3}(t) = (1-t)^3,\ B_{1,3} = 3(1-t)^2t,\ B_{2,3} = 3(1-t)t^2,\ B_{3,3} = t^3 \tag{2}$$

- Cubic Bézier curves are **defined** as ($n = 3$):

$$\begin{aligned} \mathbf{C}_3(t) &= \sum_{i=0}^{n} B_{i,n}(t)\mathbf{b}_i \\ &= (1-t)^3\mathbf{b}_0 + 3(1-t)^2t\mathbf{b}_1 + 3(1-t)t^2\mathbf{b}_2 + t^3\mathbf{b}_3 \end{aligned} \tag{3}$$

- **Endpoint Interpolation Property:** $\mathbf{C}_3(0) = \mathbf{b}_0$ and $\mathbf{C}_3(1) = \mathbf{b}_3$, cf. [1, p.147].
- **Endpoint Tangent Property:** $\mathbf{C}_3'(0) = 3(\mathbf{b}_1 - \mathbf{b}_0)$ and $\mathbf{C}_3'(1) = 3(\mathbf{b}_3 - \mathbf{b}_2)$.
- **Convex Hull Property:** $\forall t \in [0,1]$, $\mathbf{C}_3(t) \in CH\{\mathbf{b}_0, ..., \mathbf{b}_3\}$
- **Invariance under Affine Transformations:** (T= rotation, reflection, translation or scaling)

$$T\left(\sum_{i=0}^{n=3} B_{i,n}(t)\mathbf{b}_i\right) = \sum_{i=0}^{n=3} B_{i,n}(t)T\left(\mathbf{b}_i\right) \tag{4}$$

- **Variation Diminishing Property:** The number of intersections of a given line with $\mathbf{C}(t)$ is less than or equal to the number of intersections of that line with the control polygon.
- **The Casteljau algorithm:** For a given $t \in [0,1]$, a cubic Bézier curve defined by its control points $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$ can be expressed as follows ($n = 3$):

$$\begin{aligned}&\mathbf{C}_n(t) = \mathbf{b}_0^n, \textit{ where}\\&\begin{cases}\mathbf{b}_i^0 = \mathbf{b}_i, \textit{ and}\\ \mathbf{b}_i^j = (1-t)\mathbf{b}_i^{j-1} + t\mathbf{b}_{i+1}^{j-1}\end{cases}\\&\text{for } j = 1, ..., n \text{ and } i = 0, .., n-j\end{aligned} \tag{5}$$

This property is based on the recursion property of the BERNSTEIN polynomials. It is especially useful for the rendering and the subdivision of the curve.

- **Subdivision of curve:** The control points of the two curves parts obtained by subdivision at parameter value t are:
$$\begin{cases}\mathbf{C}_{3,left}(t) : \mathbf{b}_0^0, \mathbf{b}_0^1, \mathbf{b}_0^2, \mathbf{b}_0^3\\ \mathbf{C}_{3,right}(t) : \mathbf{b}_0^3, \mathbf{b}_1^2, \mathbf{b}_2^1, \mathbf{b}_3^0\end{cases} \tag{6}$$
according to the CASTELJAU algorithm.

- **Conversion between representations:** All polynomials can be expressed in BÉZIER form. For cubic curves, this means:

$$\begin{aligned}\mathbf{C}_3(t) &= \sum_{i=0}^{n=3} t^i \mathbf{a}_i\\ &= \mathbf{a}_0 + t\mathbf{a}_1 + t^2\mathbf{a}_2 + t^3\mathbf{a}_3\end{aligned} \tag{7}$$

Eq. 3 and 7 can be converted back an forth by applying the matrix operations:

$$\begin{pmatrix}\mathbf{a}_0 & \mathbf{a}_1 & \mathbf{a}_2 & \mathbf{a}_3\end{pmatrix} = \begin{pmatrix}\mathbf{b}_0 & \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3\end{pmatrix}\begin{pmatrix}1 & -3 & 3 & -1\\ 0 & 3 & -6 & 3\\ 0 & 0 & 3 & -3\\ 0 & 0 & 0 & 1\end{pmatrix} \tag{8}$$

and

$$\begin{pmatrix}\mathbf{b}_0 & \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3\end{pmatrix} = \frac{1}{3}\begin{pmatrix}\mathbf{a}_0 & \mathbf{a}_1 & \mathbf{a}_2 & \mathbf{a}_3\end{pmatrix}\begin{pmatrix}3 & 3 & 3 & 3\\ 0 & 1 & 2 & 3\\ 0 & 0 & 1 & 2\\ 0 & 0 & 0 & 3\end{pmatrix} \tag{9}$$

- **Derivatives of a Bézier curve:**[3]

$$\begin{aligned}\mathbf{C}'_\mathbf{n}(t) &= \sum_{i=0}^{n-1} B_{i,n-1}(t)\mathbf{b}_i^{(1)}\\ \text{where } \mathbf{b}_i^{(1)} &= n(\mathbf{b}_{i+1} - \mathbf{b}_i)\\ \mathbf{C}''_\mathbf{n}(t) &= \sum_{i=0}^{n-2} B_{i,n-2}(t)\mathbf{b}_i^{(2)}\\ \text{where } \mathbf{b}_i^{(2)} &= (n-1)(\mathbf{b}_{i+1}^{(1)} - \mathbf{b}_i^{(1)}) = n\cdot(n-1)(\mathbf{b}_{i+2} - 2\mathbf{b}_{i+1} + \mathbf{b}_i)\end{aligned} \tag{10}$$

[3]Although strictly speaking, a BÉZIER curve is defined on the close interval $t \in [0, 1]$, practically the derivatives may well be calculated at the interval extrema, because the underlying cubic polynomial is defined $\forall t \in \mathbb{R}$. Continuity and differentiability should however be evaluated, if necessary on the open interval only $t \in (0, 1)$.

Note however that the derivatives are sometimes easier to handle when using the monomial representation:

$$\begin{aligned}\mathbf{C}'_{\mathbf{n}}(t) &= \sum_{i=0}^{n-1}(i+1)t^i\mathbf{a}_{i+1}\\ \mathbf{C}''_{\mathbf{n}}(t) &= \sum_{i=0}^{n-2}(i+2)(i+1)t^i\mathbf{a}_{i+2}\end{aligned} \tag{11}$$

5 Tools

5.1 Line equation using Homogeneous Coordinates (cf. [1, pp. 38-40])

In the Cartesian plane the general line equation is $ax + by + c = 0$. If we use instead of $\mathbf{p} = (x, y)$, the homogeneous coordinates $\vec{p} = (x, y, 1)$, and the line coefficients vector $\vec{l} = (a, b, c)$, we can write the equation as a dot product:

$$\vec{l} \cdot \vec{p} = ax + by + c = 0 \tag{12}$$

This means that both vectors are orthogonal. As a consequence, if the line is defined by two distinct points $\mathbf{p_1}$ and $\mathbf{p_2}$, the cross product of their corresponding homogeneous point vectors $\vec{p_1}$ and $\vec{p_2}$ yields an orthogonal vector, which in this case is the line vector (cf. Fig. 3):

$$\vec{l} = \vec{p_1} \times \vec{p_2} \tag{13}$$

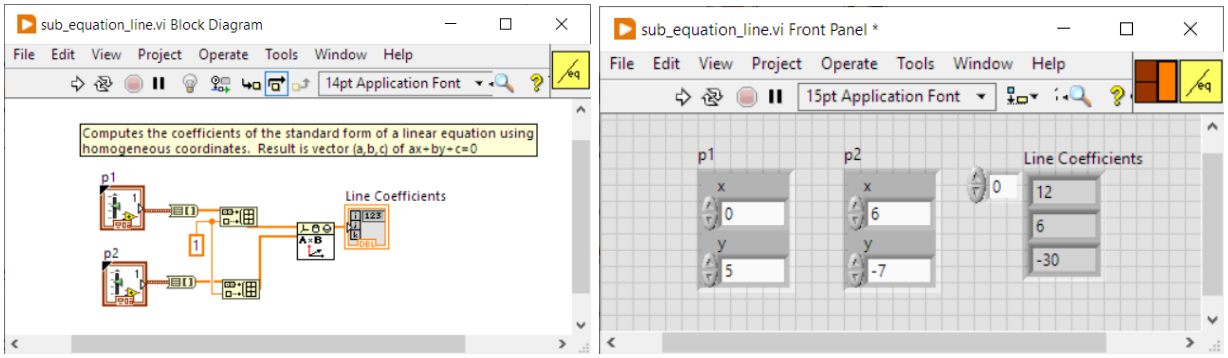


Figure 3: LabVIEW implementation of the line definition by two points.

5.2 Offset or parallel curve, cf. [1, pp. 107-109]

In order to be able to solve the constant distance path problem (1.), one considers the unit normal vector $\vec{n}(t)$ to the curve tangent at $(x(t), y(t))$, the slope of which is represented by the first curve derivative $\mathbf{C}'(t) = (x'(t), y'(t))$. If d is the offset distance, the offset curve is given by:

$$\mathbf{O}_d(t) = \mathbf{C}(t) \pm d \cdot \vec{n}(t) \tag{14}$$

$$\vec{n}(t) = \frac{(-y'(t), x'(t))}{|\ \vec{C}'(t)\ |} \tag{15}$$

where $|\ \vec{C}'(t)\ |$ equals the curve's speed $v(t) = \sqrt{x'(t)^2 + y'(t)^2}$.

Note that speed may not be zero here, otherwise the normal vector cannot be defined due to undefined $\frac{0}{0}$ division.

The sign in Eq. 14 indicates, whether the offset curve is situated this or that side of the curve.

5.3 Using complex numbers and matrices

5.3.1 Cubic Bézier equations

The planar curve equations so far can be expressed using complex numbers $\boldsymbol{z} = x + yi$, t real, with $t \in [0, 1]$:

$$\begin{aligned} \boldsymbol{C_3}(t) &= \begin{pmatrix}(1-t)^3 & 3(1-t)^2 t & 3(1-t)t^2 & t^3\end{pmatrix} \bullet \begin{pmatrix}\boldsymbol{z_0} & \boldsymbol{z_1} & \boldsymbol{z_2} & \boldsymbol{z_3}\end{pmatrix}^T \\ &= \begin{pmatrix}1 & t & t^2 & t^3\end{pmatrix} \bullet \begin{pmatrix}\zeta_0 & \zeta_1 & \zeta_2 & \zeta_3\end{pmatrix}^T \end{aligned} \tag{16}$$

where the $\bullet$ symbol denotes the operation $\boldsymbol{A} \bullet \boldsymbol{B} = \sum_i \boldsymbol{a}_i \cdot \boldsymbol{b}_i$, which differs from the usual complex dot product $\boldsymbol{A} \cdot \boldsymbol{B} = \sum_i \boldsymbol{a}_i \cdot \overline{\boldsymbol{b}_i}$.

$$\begin{pmatrix}\zeta_0 & \zeta_1 & \zeta_2 & \zeta_3\end{pmatrix} = \begin{pmatrix}\boldsymbol{z_0} & \boldsymbol{z_1} & \boldsymbol{z_2} & \boldsymbol{z_3}\end{pmatrix} \begin{pmatrix}1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1\end{pmatrix} \tag{17}$$

or, inversely

$$\begin{pmatrix}\boldsymbol{z_0} & \boldsymbol{z_1} & \boldsymbol{z_2} & \boldsymbol{z_3}\end{pmatrix} = \frac{1}{3}\begin{pmatrix}\zeta_0 & \zeta_1 & \zeta_2 & \zeta_3\end{pmatrix} \begin{pmatrix}3 & 3 & 3 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3\end{pmatrix} \tag{18}$$

$$\boldsymbol{C'_3}(t) = \begin{pmatrix}1 & 2t & 3t^2\end{pmatrix} \bullet \begin{pmatrix}\zeta_1 & \zeta_2 & \zeta_3\end{pmatrix}^T \tag{19}$$

$$\boldsymbol{C''_3}(t) = \begin{pmatrix}2 & 6t\end{pmatrix} \bullet \begin{pmatrix}\zeta_2 & \zeta_3\end{pmatrix}^T \tag{20}$$

5.3.2 2D dot product and cross products

Let $\vec{p}_a = (x_a, y_a)$, $\vec{p}_b = (x_b, y_b)$ and $\boldsymbol{z_a} = x_a + y_a i$, $\boldsymbol{z_b} = x_b + y_b i$.

$$\begin{aligned} \vec{p}_a \cdot \vec{p}_b &= x_a x_b + y_a y_b = \Re(\overline{\boldsymbol{z}}_{\boldsymbol{a}} \boldsymbol{z_b}) = \Re(\boldsymbol{z_a}\overline{\boldsymbol{z}}_{\boldsymbol{b}}) \\ \vec{p}_a \times \vec{p}_b &= (x_a y_b - x_b y_a)\vec{k} = \Im(\overline{\boldsymbol{z}}_{\boldsymbol{a}} \boldsymbol{z_b})\vec{k} = -\Im(\boldsymbol{z_a}\overline{\boldsymbol{z}}_{\boldsymbol{b}})\vec{k} = \begin{vmatrix} x_a & x_b \\ y_a & y_b \end{vmatrix} \vec{k} \end{aligned} \tag{21}$$

where $\vec{k}$ is the unit vector orthogonal to $\vec{i}$ and $\vec{j}$ defining the x, y and z-axes. In 2D only the signed magnitude of the cross product are of interest.

5.3.3 Unit tangent and unit normal vector at $\boldsymbol{C}(t)$

$$\begin{aligned} \boldsymbol{t}(t) &= \frac{\boldsymbol{C'}(t)}{\mid \boldsymbol{C'}(t) \mid} \\ \boldsymbol{n}(t) &= \boldsymbol{t}(t) \cdot i \end{aligned} \tag{22}$$

Note that the product by imaginary i represents a rotation by $\frac{\pi}{2}$. Remind that $\mid \boldsymbol{C'}(t) \mid$ equals the curve speed $v(t)$.

5.3.4 Curvature $\kappa(t)$ and radius of curvature $\rho(t)$ of a parametric curve (cf [1., pp. 267-275]

$$\begin{aligned} \kappa(t) &= \frac{x'(t)y''(t) - y'(t)x''(t)}{(x'(t)^2 + y'(t)^2)^{\frac{3}{2}}} \\ &= \frac{\Im\left(\overline{\boldsymbol{C'}(t)}\boldsymbol{C''}(t)\right)}{\mid \boldsymbol{C'}(t) \mid^3} \\ \rho(t) &= \frac{1}{\kappa(t)} \end{aligned} \tag{23}$$

Important note: Software must check, if there exists a concave curve part with curvature radius that is smaller than the milling tool radius d, which is the distance of the offset curve.

5.4 Finding the orientation of a convex polygon

If the polygon is defined by an ordered array of points $\boldsymbol{z}_i$, the easiest method of finding the winding order (=orientation) by choosing three adjacent points $\boldsymbol{z}_i, \boldsymbol{z}_{i+1}, \boldsymbol{z}_{i+2}$ and calculating the magnitude of the cross product of the corresponding vectors $\boldsymbol{z}_A = (\boldsymbol{z}_{i+2} - \boldsymbol{z}_i)$ and $\boldsymbol{z}_B = (\boldsymbol{z}_{i+1} - \boldsymbol{z}_i)$. The cross product yields a vector that is orthogonal to both the real and the imaginary axes. The resulting vector may have positive or negative direction depending on the sign of w, which is defined by:

$$w = \Im(\overline{\boldsymbol{z}_B}\boldsymbol{z}_A) \tag{24}$$

Important note: Because the piecewise curves that are used in this project are not necessarily convex, we decided to use another method of finding the orientation. We first determine the curve barycenter, operate a translation of the whole curve from this point to the origin. Then we render the curve in the initial array order. We then check, if the arguments of the curve points are increasing or decreasing. Because concave curve parts may exist, the resulting argument function could have rising or decreasing parts. Therefore, we operate a linear fit over the argument function, and consider the sign of the slope of the regression line as the main curve orientation. This method must only be applied once for a specific curve.

5.5 Implementation of the Casteljau algorithm

5.5.1 Recursive implementation

Fig. 4 depicts the straight forward bottom-up implementation of the Casteljau algorithm evaluating the curve at parameter t according to equation Eq. 5. LabVIEW accepts a self-calling function only, if its execution property is changed to *re-entrant*. In the case of a cubic Bézier curve, the recursive depth is 4. If the termination condition is reached with $j = 0$, the result is passed to the next level, and from this to level 2 a.s.o. Behind the scene, the computer must store the intermediate data on the stack, which slows down the execution.

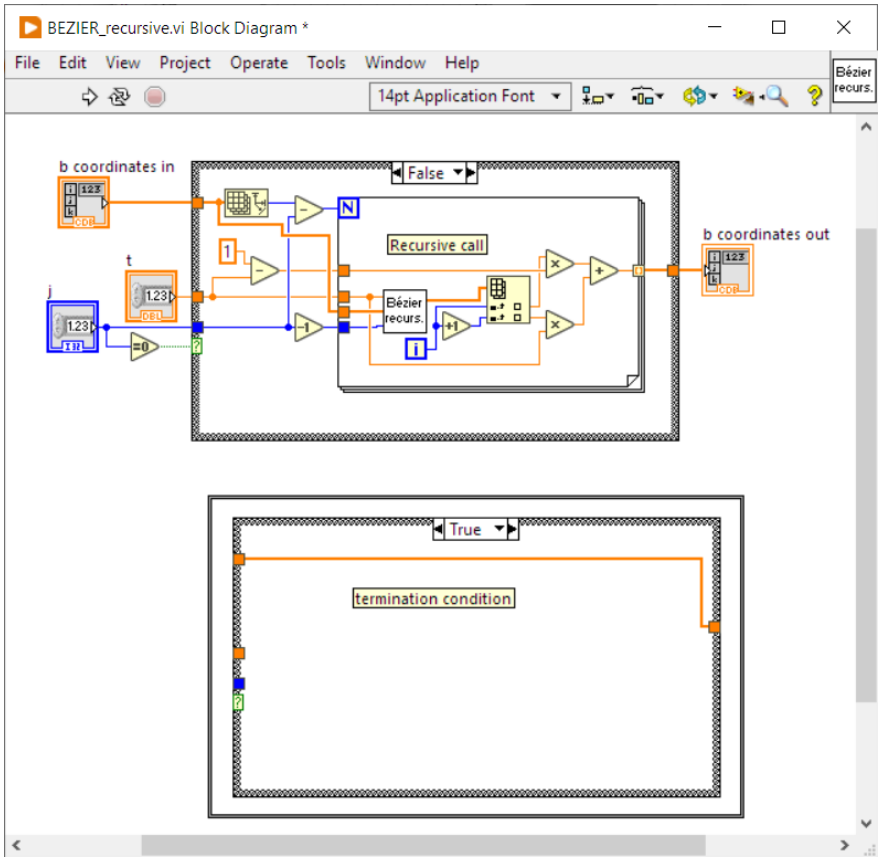


Figure 4: For this LabVIEW self-calling program to work, the vi execution property must be set to *re-entrant*.

5.5.2 Iterative implementation

Execution time is improved if an iterative version is used. Increasing computing speed is possible because Eq. 5 doesn't break the initial problem into two (or more) sub-problems of the same kind and thus doesn't represent a divide-and-conquer solution.

Each iteration reduces the number of points to evaluate by 1 until a single point is left over. Since the number of iterations required is known, the iteration loop can be made using a *for loop*.

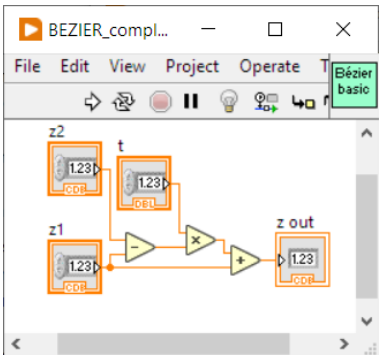


Figure 5: Eq. 5 is slightly altered in order to have a single multiplication.

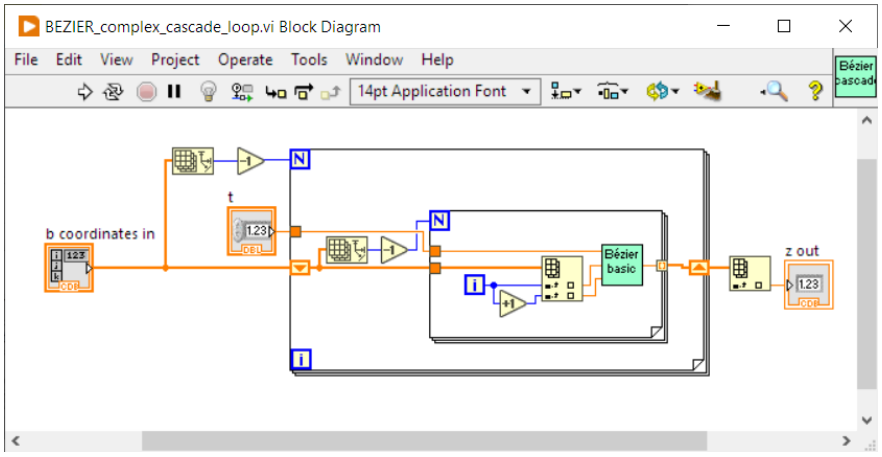


Figure 6: Generally, iterative methods use *loops*. In this case, each loop cycle reduces the number of evaluated points by 1.

5.5.3 Cascaded implementation

If the BÉZIER degree is small, execution time may be increased by avoiding loops, as shown in Fig. 7. Instead, the basic computations are cascaded.

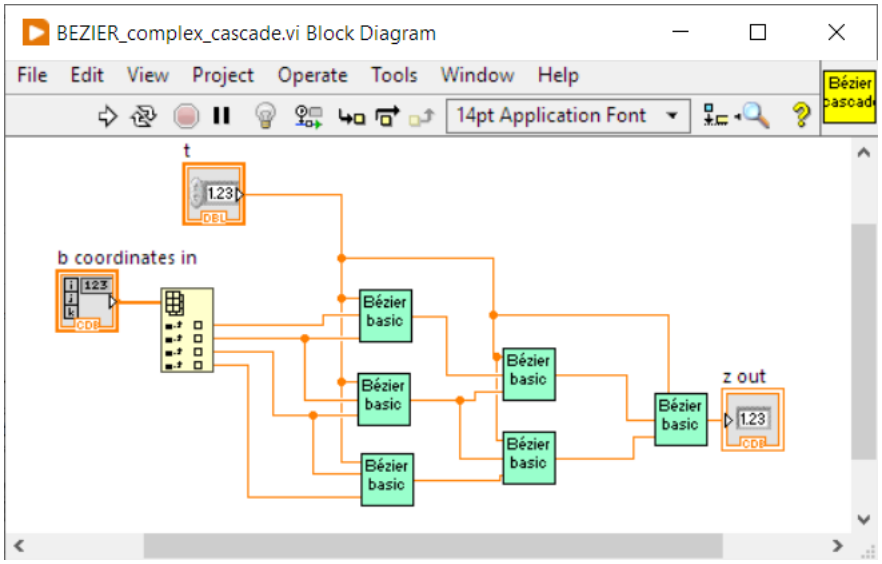


Figure 7: This program doesn't use loops. The basic functions are cascaded.

Note that we evaluated execution speed by comparing the three alternatives. The test program running on a PC (Intel Core i5 650 @ 3.2GHz, 64bit) run 10^6 functions calls, which produced the execution durations exposed in Table 1.

degree	recursive	iterative	cascaded
cubic	2.87s	1.11s	0.58
5th degree	32.94s	3.83s	-

Table 1: Execution time comparison of one million function calls

5.5.4 Splitting a cubic Bézier curve at t

The cascaded Casteljau algorithm is not only the fastest implementation in the case of cubic Bézier curves, it also allows a pretty simple method to subdivide such a curve by applying Eq. 6 in a straight forward manner.

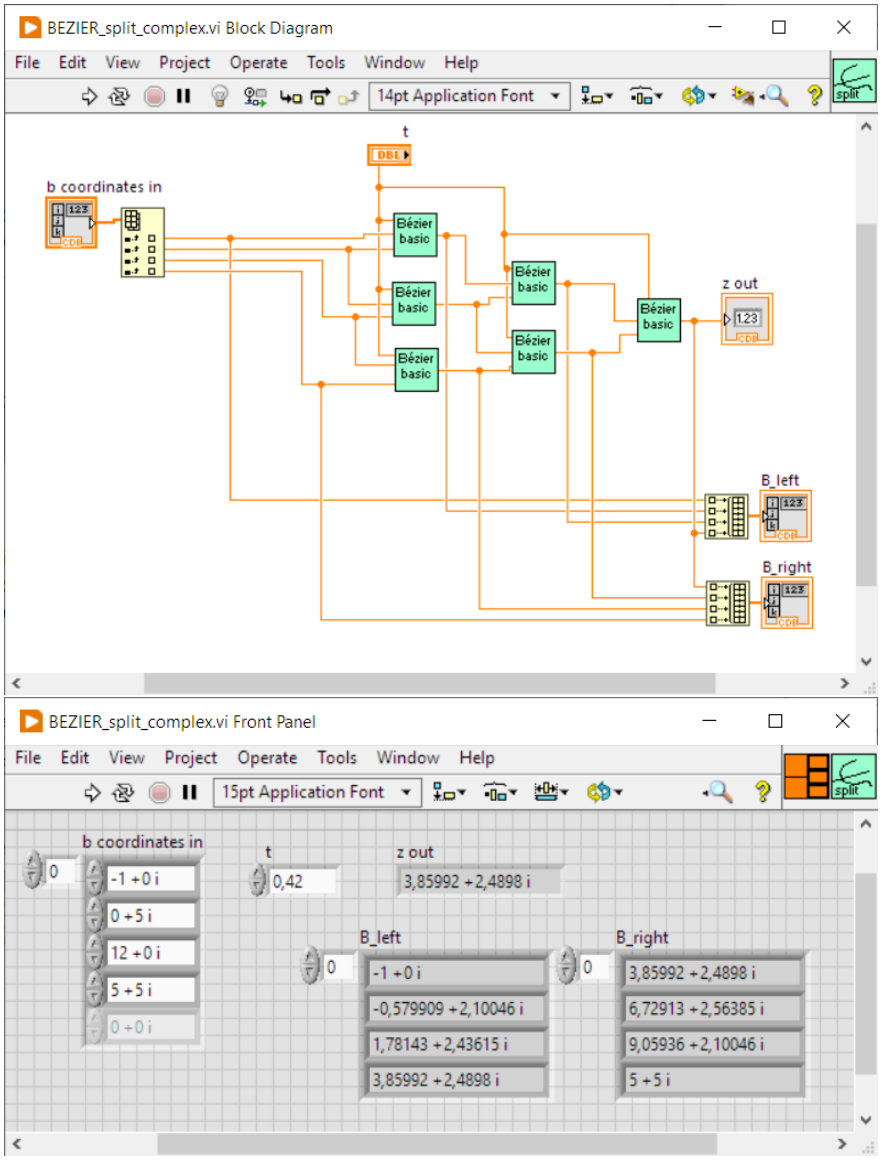


Figure 8: This program splits the input curve into two new sub-curves of the same kind.

5.6 Rendering of a cubic Bézier curve

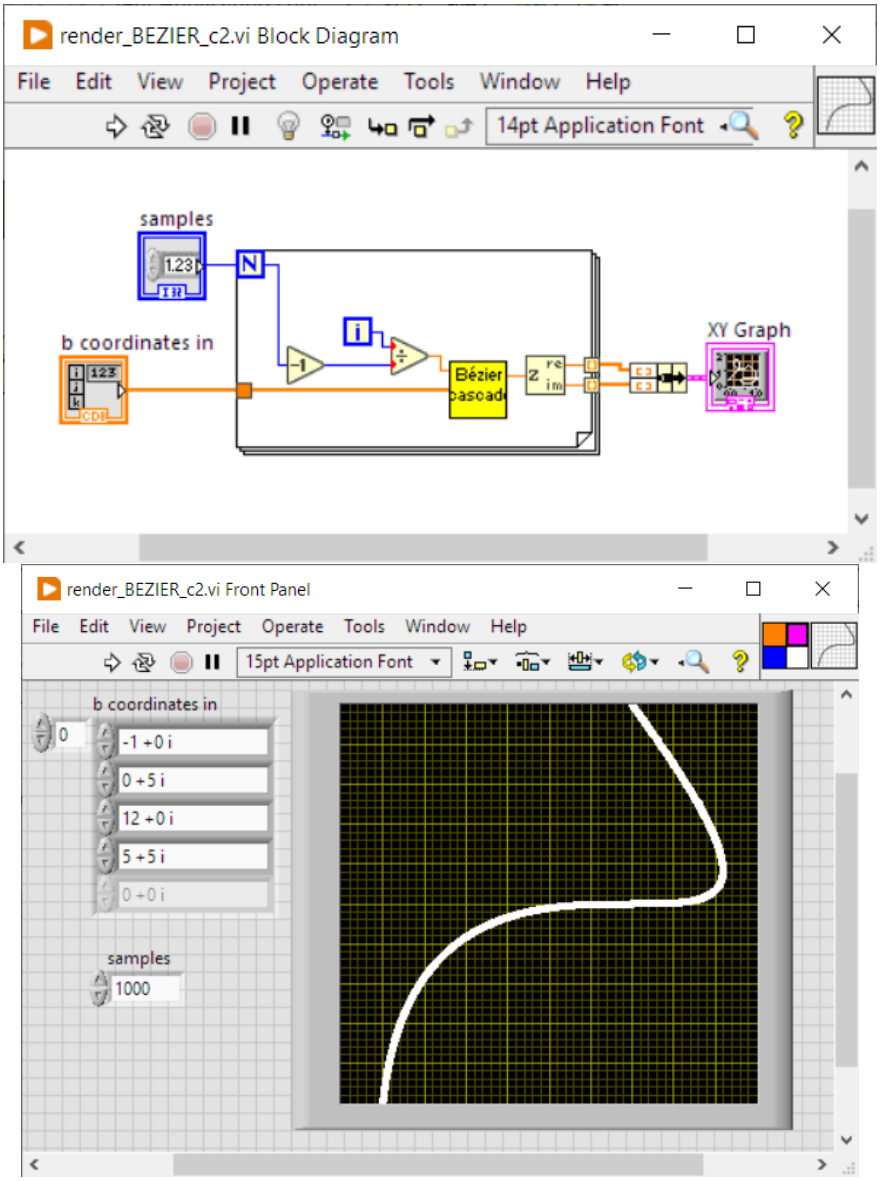


Figure 9: This sub.vi renders a cubic Bézier curve by applying the fast cascading method shown in Fig. 7.

5.7 Intersection of a circle and a line

Traditionally, the problem of yielding the intersection of a circle and a line is solved algebraically in the Cartesian plane. In order to simplify the bulky equations, one can translate the circle at the origin, (cf. [13]). This indirect approach may be extended by applying additional rotations.

We present here a geometrical method in the complex plane for yielding the intersection points of a circle with center z_0, radius r and a line defined by two points z_1, z_2.

The idea is to move the original circle to the origin **O** and use the same translation for the line, then rotate both about the origin by the slope of the line. The circle remains invariant, and the rotated line becomes

parallel to the real axis, and thus gets constant imaginary part for any point on it. Yielding both intersection points is reduced to finding the opposite real coordinates of the two points with identical imaginary coordinates. The resulting point vectors are then rotated and translated back, yielding the searched coordinates.

Considering the line l as defined by the complex vector $\vec{\boldsymbol{l}} = (\boldsymbol{z_1}, \boldsymbol{z_2})$, with $\boldsymbol{z_1} \neq \boldsymbol{z_2}$, the method can be expressed as follows (cf. Fig. 10):

$$\begin{aligned}
\text{Translation: } & \vec{\boldsymbol{l}}' = \vec{\boldsymbol{l}} - (\boldsymbol{z_0}, \boldsymbol{z_0}) \\
\text{Unit rotation vector: } & \boldsymbol{z_R} = \frac{\boldsymbol{z_1} - \boldsymbol{z_2}}{|\, \boldsymbol{z_1} - \boldsymbol{z_2} \,|} \\
\text{Negative Rotation: } & \vec{\boldsymbol{l}}'' = (\boldsymbol{z_1''}, \boldsymbol{z_2''}) = \vec{\boldsymbol{l}}' / \boldsymbol{z_R} \\
\text{let: } & b = \Im(\boldsymbol{z_1''}) = \Im(\boldsymbol{z_2''}) \\
\text{Intersections of } \vec{\boldsymbol{l}}'' \text{ with circle at origin: } & \begin{cases} \text{if } |\, b \,| > r: & \text{No solution} \\ \text{if } |\, b \,| = r: & \text{Single solution } \lambda_{\boldsymbol{1}}'' = (0, b) \\ \text{if } |\, b \,| < r: & \text{Two solutions } \lambda_{\boldsymbol{1,2}}'' = (\pm\sqrt{r^2 - b^2}, b) \end{cases} \\
\text{Inverse rotation and translation: } & \lambda_{\boldsymbol{1,2}} = \lambda_{\boldsymbol{1,2}}'' \cdot \boldsymbol{z_R} + (\boldsymbol{z_0}, \boldsymbol{z_0})
\end{aligned} \tag{25}$$

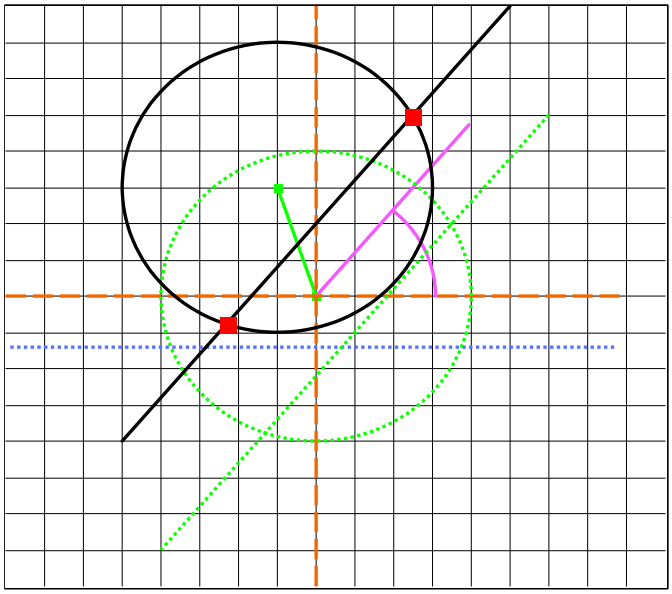


Figure 10: Green: translated circle and line. Dotted blue: negatively rotated line about origin (with constant imaginary part).

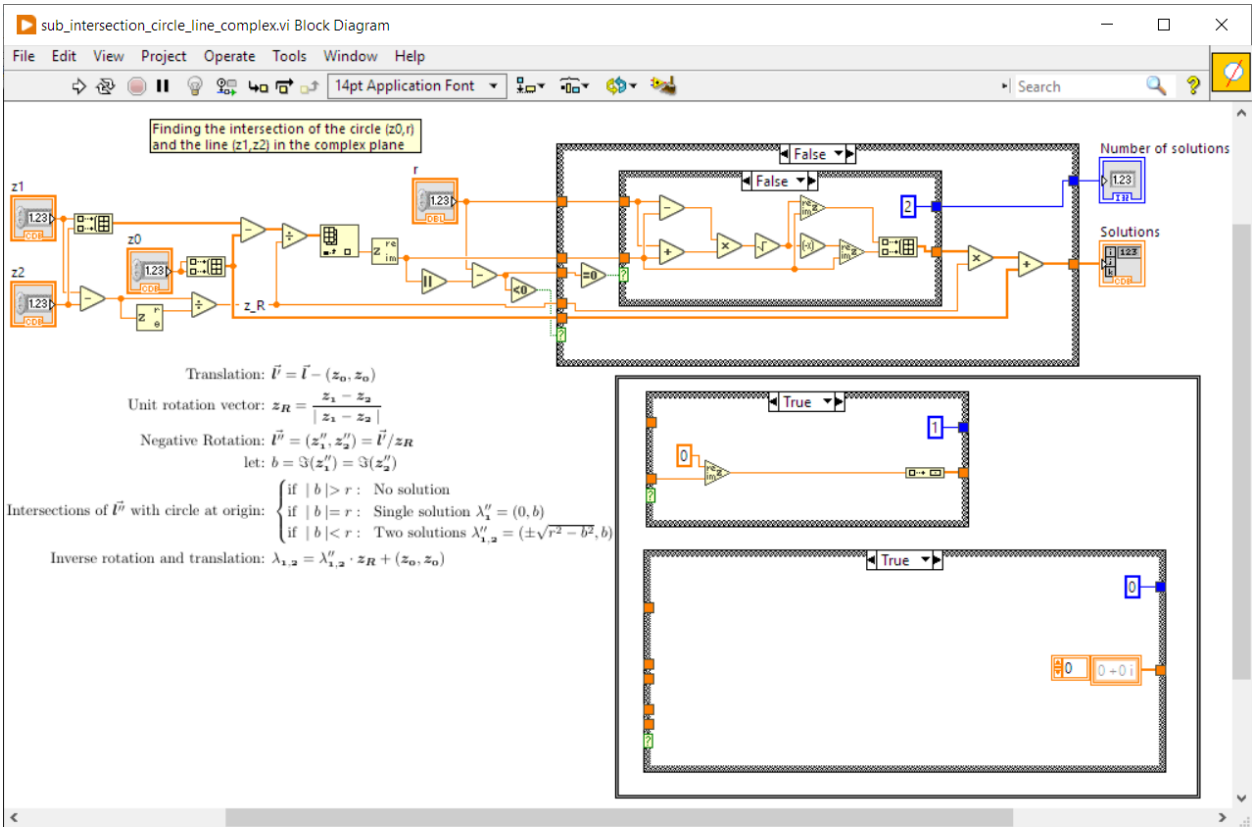


Figure 11: LabVIEW implementation of the circle/line intersection finding.

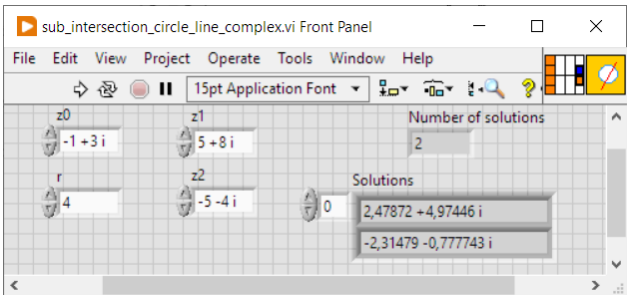


Figure 12: LabVIEW front panel.

5.7.1 Intersection of a circle with a segment

This exercise is very close to the previous line/circle intersection problem. If the segment is defined by its complex end points vector (z_1, z_2), then we only need to verify, whether the real solution(s) $\lambda_{1,2}''$ is included in the interval $\min[\Re(z_1''), \Re(z_2'')], \max[\Re(z_1''), \Re(z_2'')]$, as defined in Eq. 25.

Note that for programming ease reasons, we choose an alternative method, where both real and imaginary parts of the final result(s) $\lambda_{1,2}$ are checked for inclusion in the respective intervals.

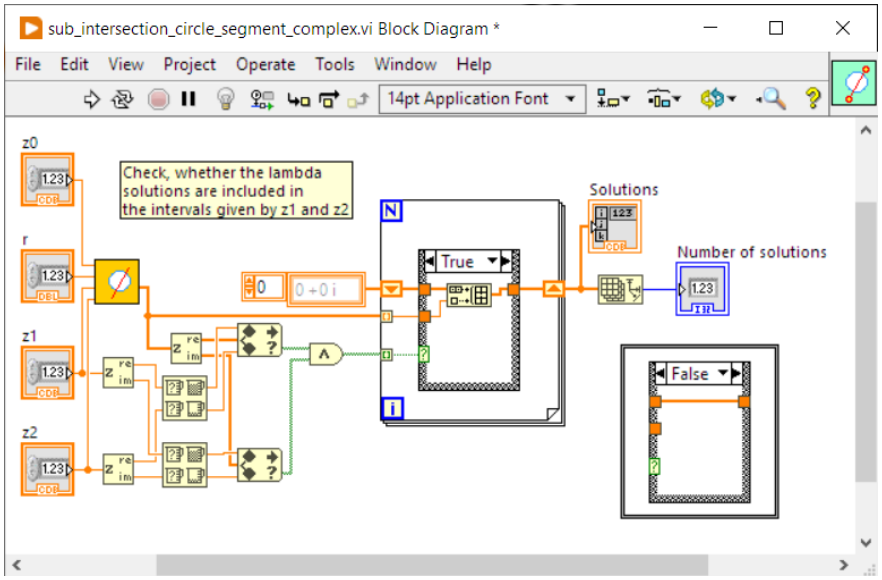


Figure 13: This sub.vi yields intersections between a circle and a segment.

5.7.2 Intersection of a circle and a quadrilateral

Without any further effort, the intersections of a circle with a quadrilateral can be yielded by applying the previous algorithm to the four concerned segments. This can be especially be used with the convex hull or the bounding box of the cubic BÉZIER control points. Note that in this case, the polygon to analyze might be a triangle.

5.8 Self-intersection of a Bézier curve (cf. [12])

As this project focuses on toolpath generation for machining, it is clear that cubic BÉZIER curves with loops should be avoided, because offset toolpaths would cross the curve. Loops can be detected, by searching for curve self-intersections. We consider the monomial form of the BÉZIER polynomial:

$$\boldsymbol{C_3}(t) = \begin{pmatrix} 1 & t & t^2 & t^3 \end{pmatrix} \bullet \begin{pmatrix} \zeta_0 & \zeta_1 & \zeta_2 & \zeta_3 \end{pmatrix}^T \tag{26}$$

We search for two distinct values λ and μ that generate the same location on the curve. Let $\zeta_j = \alpha_j + \beta_j i$.

$$\begin{aligned} \boldsymbol{C_3}(\lambda) - \boldsymbol{C_3}(\mu) &= 0 \\ \zeta_3(\lambda^3 - \mu^3) + \zeta_2(\lambda^2 - \mu^2) + \zeta_1(\lambda - \mu) &= 0 \mid \cdot \frac{1}{\lambda - \mu} \\ \zeta_3(\lambda^2 + \lambda\mu + \mu^2) + \zeta_2(\lambda + \mu) + \zeta_1 &= 0 \end{aligned} \tag{27}$$

Let $A = \lambda^2 + \lambda\mu + \mu^2$ and $B = \lambda + \mu$. We get the linear system:

$$\begin{cases} \alpha_3 A + \alpha_2 B = -\alpha_1 \\ \beta_3 A + \beta_2 B = -\beta_1 \end{cases} \tag{28}$$

A solution exists, if and only if the determinant $\mathrm{DET} = \alpha_3\beta_2 - \beta_3\alpha_2 = \Im(\overline{\zeta_3}\zeta_2) \neq 0$. This is the case, if $\zeta_3 \neq \zeta_2$.

$$A = \frac{\Im(\zeta_1\overline{\zeta_2})}{\mathrm{DET}} \quad B = \frac{\Im(\zeta_3\overline{\zeta_1})}{\mathrm{DET}} \tag{29}$$

We can rewrite:

$$\begin{cases} P = \lambda\mu = (\lambda+\mu)^2 - (\lambda^2 + \lambda\mu + \mu^2) = B^2 - A \\ S = \lambda + \mu = B \end{cases} \tag{30}$$

According to Viète's formula, the sum and the product of two numbers solve the equation:

$$t^2 - St + P = 0 \tag{31}$$

Condition:

$$\begin{aligned} \Delta = S^2 - 4P &\geq 0 \\ -3B^2 + 4A &\geq 0 \\ 4A &\geq 3B^2 \end{aligned} \tag{32}$$

$$\lambda = \frac{B + \sqrt{\Delta}}{2} \quad \mu = \frac{B - \sqrt{\Delta}}{2} \tag{33}$$

A loop exists, if there are two distinct real solutions, which is given, if $\Delta > 0$ and $\lambda, \mu \in [0, 1]$. Curve containing loops cannot be usefully handled in milling machining. So, software must verify that all the processed curves are valuable from this point of view.

5.8.1 Cusp singularities

If $\Delta = 0$, Eq. 31 has a single solution only $\lambda = \frac{B}{2}$, which in fact corresponds to a cusp singularity of the curve, where $\boldsymbol{C}'(\lambda) = 0$, so that the tangent vector is undefined (cf. Eq. 22). The reason for this particularity is that if $\Delta = 0 \Leftrightarrow 4A = 3B^2$ we can calculate the derivatives:

$$\begin{aligned} \frac{dx}{dt} &= 3\alpha_3 \left(\frac{B}{2}\right)^2 + 2\alpha_2 \frac{B}{2} + \alpha_1 \\ &= \alpha_3 A + \alpha_2 B + \alpha_1 \\ &= \frac{\alpha_3(\alpha_2\beta_1 - \alpha_1\beta_2) + \alpha_2(\alpha_1\beta_3 - \alpha_3\beta_1) + \alpha_1 DET}{DET} \\ &= \frac{\alpha_3\alpha_2\beta_1 - \alpha_3\alpha_1\beta_2 + \alpha_2\alpha_1\beta_3 - \alpha_2\alpha_3\beta_1 + \alpha_1\alpha_3\beta_2 - \alpha_1\alpha_2\beta_3}{DET} = 0 \end{aligned} \tag{34}$$

$$\begin{aligned} \frac{dy}{dt} &= \beta_3 A + \beta_2 B + \beta_1 \\ &= \frac{\beta_3(\alpha_2\beta_1 - \alpha_1\beta_2) + \beta_2(\alpha_1\beta_3 - \alpha_3\beta_1) + \beta_1 DET}{DET} \\ &= \frac{\beta_3\alpha_2\beta_1 - \beta_3\alpha_1\beta_2 + \beta_2\alpha_1\beta_3 - \beta_2\alpha_3\beta_1 + \beta_1\alpha_3\beta_2 - \beta_1\alpha_2\beta_3}{DET} = 0 \end{aligned} \tag{35}$$

In the case of a cusp, we choose to subdivide the cubic BÉZIER curve at $t = \lambda$ according to Eq. 6 (cf. Section 5.14).

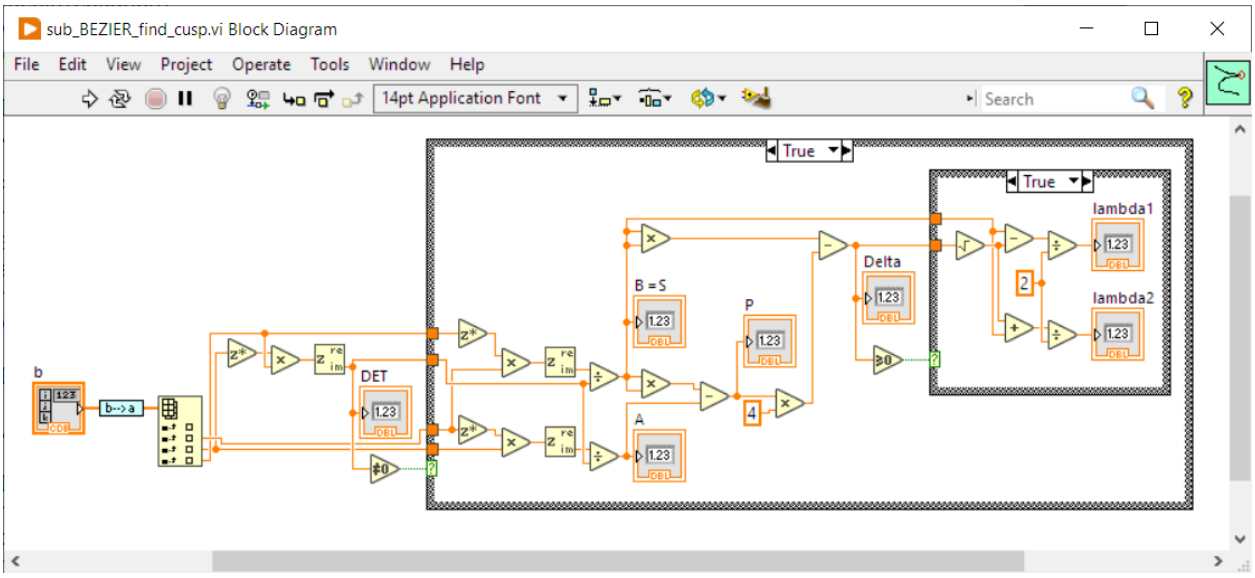


Figure 14: LabVIEW implementation of finding the locus of a cusp.

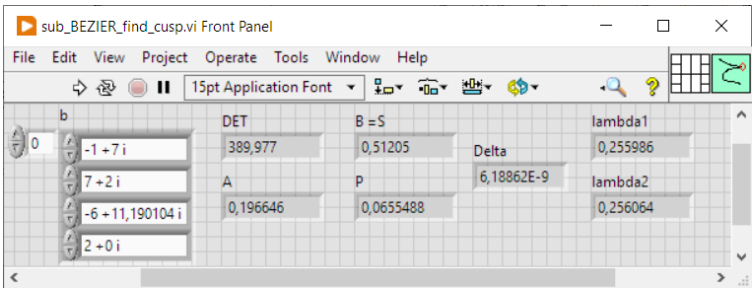


Fig. 14 shows the implementation of the cusp finding process.

Essential note: It is important to add a certain tolerance to this process. In fact, depending on the parameter step *dt* chosen for the real computations, it could be possible that the tool could cross the workpiece! The reason is that, even if no cusp or self-intersection is detected mathematically, only a few offset points or none at all could be generated in the region of the singularity, as shown in Fig. 15.

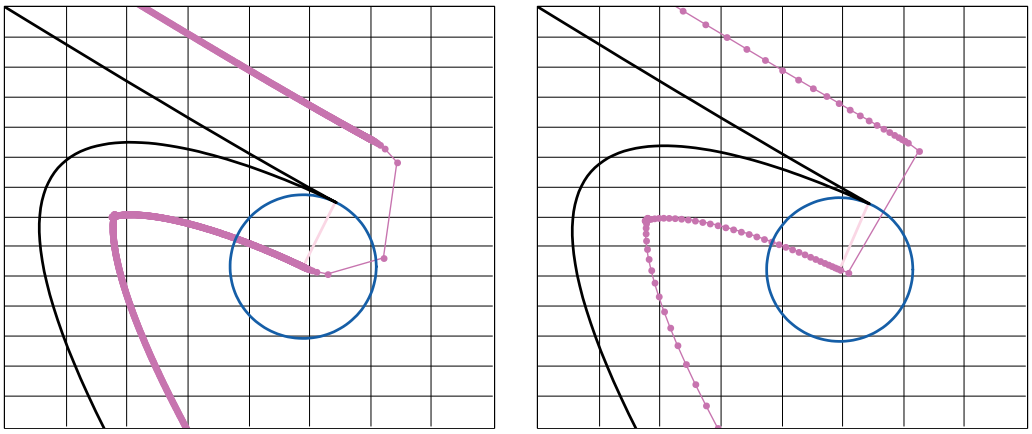


Figure 15: The parameter step dt doesn't produce the toolpath points needed to circumvent the cusp.

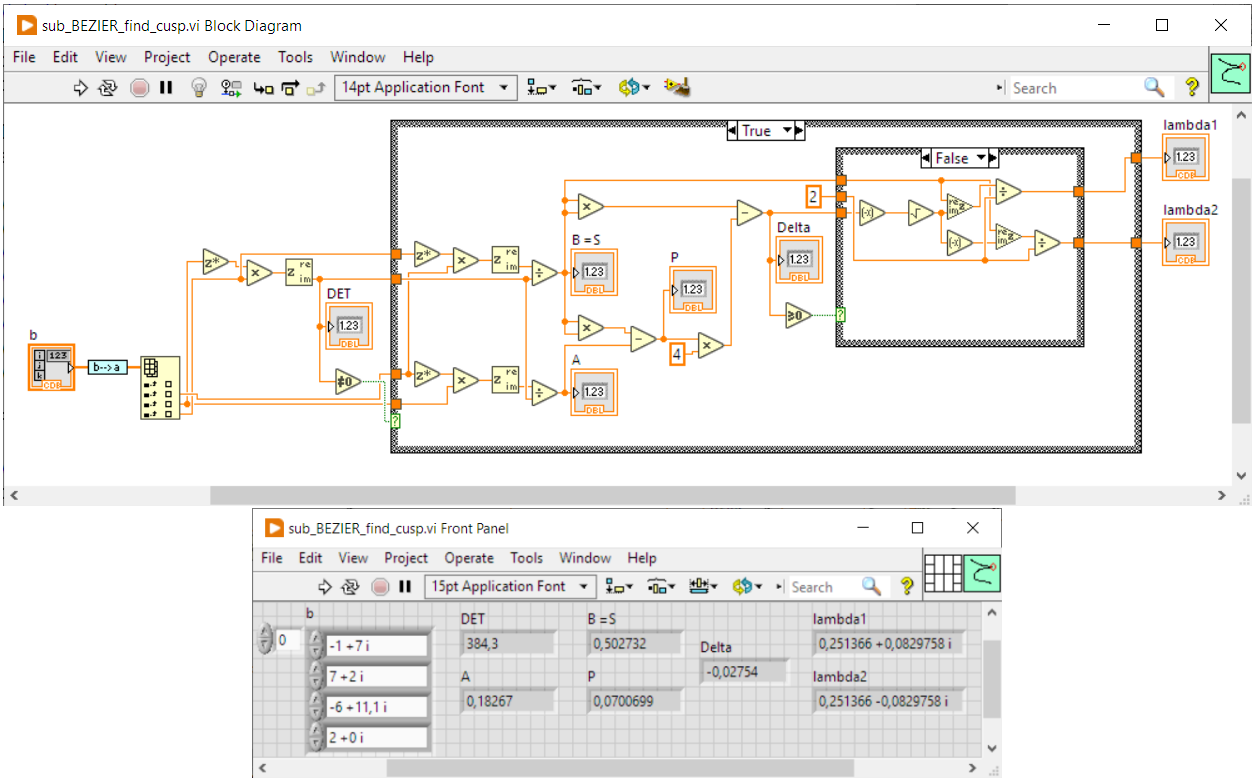


Figure 16: LabVIEW implementation of finding the locus of a **almost a cusp**.

The solution to this issue is to consider **almost cusps** that may be detected, if the real part of $\Re(\lambda) \in [0, 1]$ and the imaginary part $|\ \Im(\lambda)\ | < d$. (Of course a slightly different implementation is required, as shown in Fig. 16.)[4] In this case, and in the case of a real cusp, the Bézier curve should be split at $t = \Re(\lambda) = \Re(\mu)$. In fact in the case of complex solutions, the real part of both solutions is the same. Although mathematically, in the case of an existing cusp, there should be single real solution. However, due to rounding and truncation

[4]We use d as a rule of thumb indicator here.

errors, it might be possible that both values differ minimally. Therefore, we suppose the cusp to be located at the average of both parameter solutions.

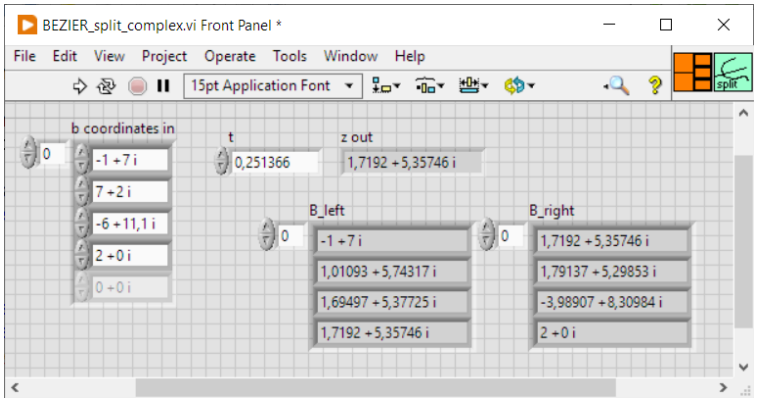


Figure 17: Splitting the curve at a **nearly cusp** location.

5.9 Finding a concavity

For the CAD/CAM (Computer Aided Design / Computer Aided Manufacturing) processes, it is essential to solve problem (2.) of our initially enumerated list. Therefore we must locate concavities with lower radii of curvature than the tool radius.

In order to find the minimal curvature radius loci, we may consider the BÉZIER curve speed function $v(t) =\mid \boldsymbol{C}'(t) \mid$ Fortunately, BÉZIER curves do not depict constant curve speed (sic!). This may sound confusing, because one really expects constant feed speed during the milling process. In fact, CNC machines that are controlled via G-Code generally move by lines or circles, while maintaining constant feed speed. So, even if the points chosen for the machined path have non-constant mathematical curve speed, the CNC will keep the real motion speed constant. Note that this has been explained in the cited COMPUTARIUM paper (cf. Footnote 1). The underlying BÉZIER curves are converted to G-Code by using the parameter t generated curve segments, which are longer, if the local curvature radius is large, and vice-versa smaller, in the other case.

This provides us a useful method for yielding the locus $t_{min(\rho)}$ of the smallest curvature radius, which is given, if the curve speed is minimal. We only need to calculate the roots of the first speed derivative, and choose the locus with the smallest speed, given the desired sign of the curvature radius, which depends on the positive or negative toolpath offset.

$$\begin{aligned}
&\text{Let } \alpha = 3a_{x,3}\ \beta = 2a_{x,2}\ \gamma = a_{x,1} \\
&\text{and } \delta = 3a_{y,3}\ \epsilon = 2a_{y,2}\ \eta = a_{y,1} \\
&v(t) = \sqrt{x'(t)^2 + y'(t)^2} \\
&x'(t) = \alpha t^2 + \beta t + \gamma \\
&y'(t) = \delta t^2 + \epsilon t + \eta
\end{aligned} \tag{36}$$

Because the square root function is monotonic increasing function, it is sufficient for the study to calculate the first derivative of $f(t) = v(t)^2$. With some effort, we get:

$$f'(t) = 4(\alpha^2 + \delta^2)t^3 + 6(\alpha\beta + \delta\epsilon)t^2 + 2[2(\alpha\gamma + \delta\eta) + \beta^2 + \epsilon^2]t + 2(\beta\gamma + \epsilon\eta) \tag{37}$$

The cubic equation $f'(t) = 0$ may be solved using Cardano's formula, for instance. (Note: in our LABVIEW implementation, we use the high-speed built-in root finding sub.vi.)

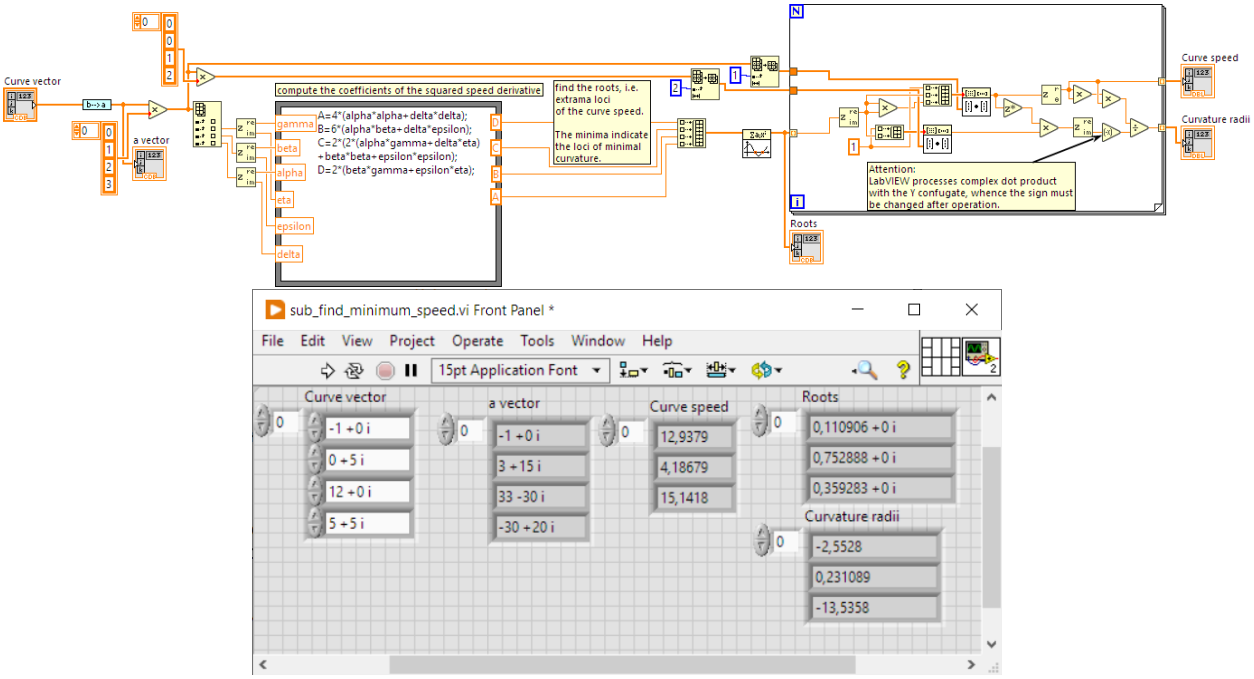


Figure 18: LabVIEW implementation of finding the loci of the minimal curvature radii.

5.10 Finding the minimax bounding box of a cubic Bézier curve.

The minimax bounding box represents the rectangle with parallel sides to the $x-$ and $y-$ axes that includes all the curve points. It is closer to the curve than the convex hull of the control points. Instead of scanning and minimaxing all the rendered points, we can find the bounding rectangle by searching for all the curve points with tangents that are parallel to the $x-$ and $y-$ axes, among which we find the contact points of the curve with the bounding box. In the case of a non-closed curve, the control points $\boldsymbol{b_0}$ and $\boldsymbol{b_3}$ must also be added to the list, because in most of the cases the curve is not closed, so that these points may also define the minimax bounding box.

We yield the points with axis parallel tangents by finding all zeros of the first derivatives for the real and imaginary components of the curve equation (we only write the equation for the real part here):

$$\begin{aligned} 3\Re(\zeta_3)\lambda^2 + 2\Re(\zeta_2)\lambda + \Re(\zeta_1) &= 0 \\ \Delta = 4\Re(\zeta_2)^2 - 12\Re(\zeta_3)\Re(\zeta_1) &\geq 0 \\ \lambda_x = \frac{-2\Re(\zeta_2) \pm \sqrt{\Delta}}{6\Re(\zeta_3)} & \end{aligned} \tag{38}$$

Note that if both real and imaginary components are zero, we have a cusp. The program shown in Fig. 19 uses a complete function for solving polynomial equations of the second degree $ax^2 + bx + c = 0$ in the case of real numbers. The sub.vi also considers the special cases, where $a = 0 \vee b = 0 \vee c = 0$.

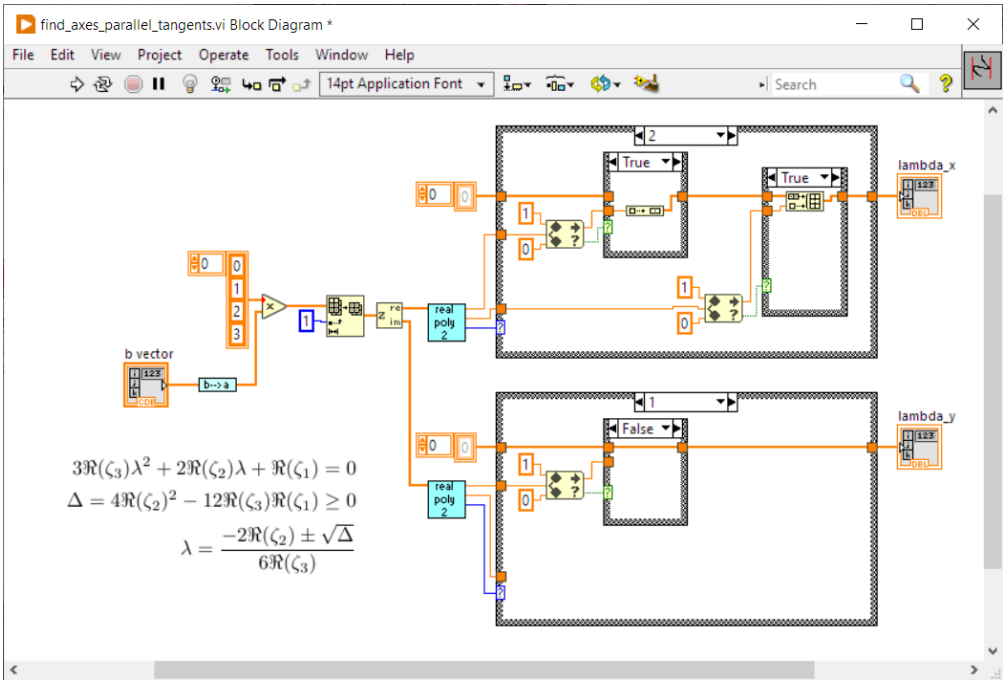


Figure 19: Yielding extremum locations (vi diagram).

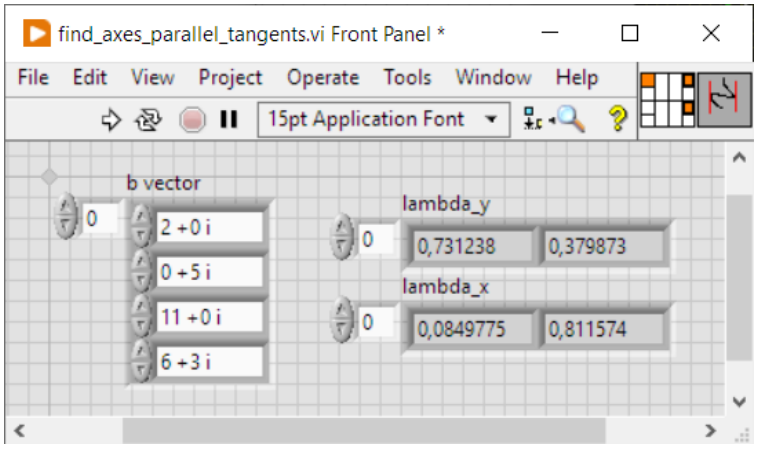


Figure 20: Yielding extremum locations (front panel).

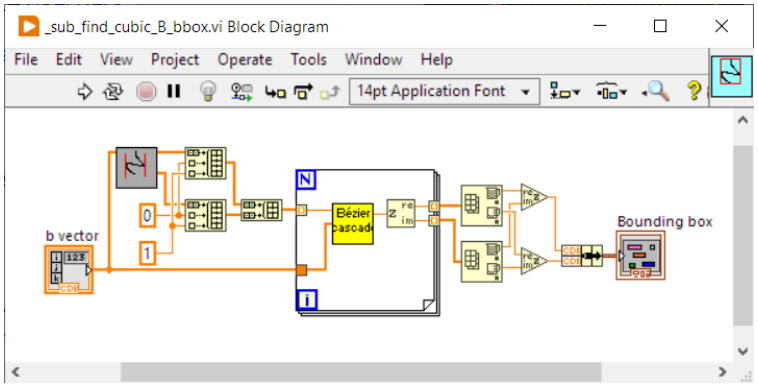


Figure 21: Yielding extremum locations (front panel).

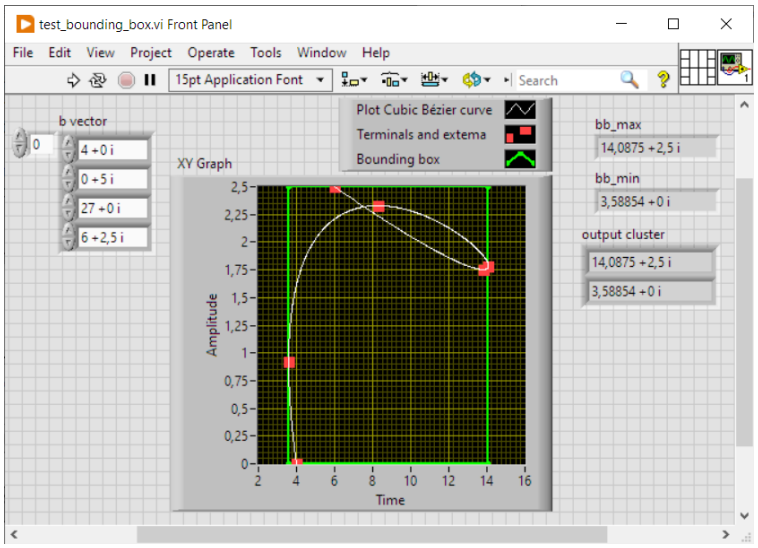


Figure 22: Testing the method.

5.11 Intersection of a circle and a cubic Bézier curve

5.11.1 Algebraic method

In order to be able to solve enumerated problems (2.) an (3.), we will need an efficient computation of the intersection of a circle with a Bézier curve.

The intersection(s) can be defined by a polynomial of degree 6, whose real roots are isolated. Note that for this project, it is not necessary to be absolutely certain that the root candidates are real. It is sufficient to know that the corresponding circle points are located in an epsilon-neighborhood of the curve points.

If a curve point intersects with a circle defined by its center $\boldsymbol{O} = p_x + p_y i$ and the radius r, the curve point $\boldsymbol{C}(t)$ matches the following equation:

$$(a_{x,3}t^3 + a_{x,2}t^2 + a_{x,1}t + a_{x,0} - p_x)^2 + (a_{y,3}t^3 + a_{y,2}t^2 + a_{y,1}t + a_{y,0} - p_y)^2 = r^2 \tag{39}$$

With some effort the equation is expanded to:

$$\begin{aligned}
&a_{x,3}^2 t^6 + 2a_{x,3}a_{x,2}t^5 + (2a_{x,3}a_{x,1} + a_{x,2}^2)t^4 + (2a_{x,3}m_x + 2a_{x,2}a_{x,1})t^3 + (2a_{x,2}m_x + a_{x,1}^2)t^2 + 2a_{x,1}m_x t + m_x^2 \\
&+a_{y,3}^2 t^6 + 2a_{y,3}a_{y,2}t^5 + (2a_{y,3}a_{y,1} + a_{y,2}^2)t^4 + (2a_{y,3}m_y + 2a_{y,2}a_{y,1})t^3 + (2a_{y,2}m_y + a_{y,1}^2)t^2 + 2a_{y,1}m_y t + m_y^2 - r^2 = 0 \\
&= At^6 + Bt^5 + Ct^4 + Dt^3 + Et^2 + Ft + G = 0
\end{aligned} \tag{40}$$

where

$$\begin{aligned}
m_x &= a_{x,0} - p_x \text{ and } m_y = a_{y,0} - p_y \\
A &= a_{x,3} + a_{y,3} \\
B &= 2a_{x,3}a_{x,2} \\
C &= (2a_{x,3}a_{x,1} + a_{x,2}^2) + (2a_{y,3}a_{y,1} + a_{y,2}^2) \\
D &= (2a_{x,3}m_x + 2a_{x,2}a_{x,1}) + (2a_{y,3}m_y + 2a_{y,2}a_{y,1}) \\
E &= (2a_{x,2}m_x + a_{x,1}^2) + (2a_{y,2}m_y + a_{y,1}^2) \\
F &= 2a_{x,1}m_x + 2a_{y,1}m_y \\
G &= m_x^2 + m_y^2 - r^2
\end{aligned} \tag{41}$$

Fig. 23 to 25 depict the LabVIEW implementation of the root finding process for the intersection of a circle with a cubic Bézier curve. The VIs present the option to choose between $\boldsymbol{z}$ or ζ, (here called $\boldsymbol{b}$ and $\boldsymbol{a}$). LabVIEW has a powerful and fast built-in polynomial root finding vi, which is used here.[5] Note that we must still discuss the existence of real roots, which is given, if there is at least one intersection point. Otherwise, the roots are complex.

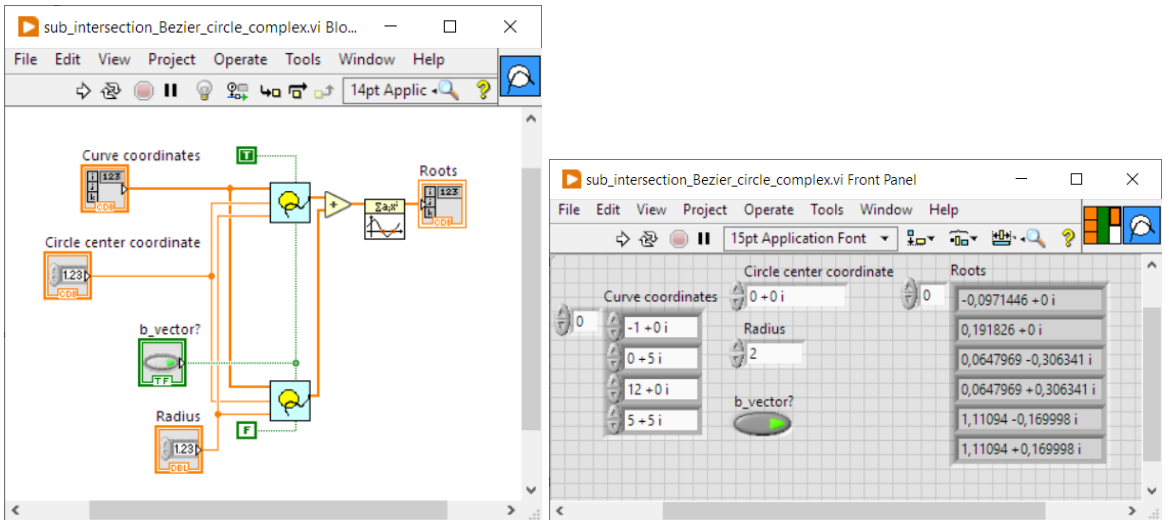


Figure 23: LabVIEW implementation of the root finding process for circle and cubic Bézier curve intersection. In the example, only the second real root is valid, as it is $\in [0, 1]$

[5]Note that in LabVIEW programs are called *Virtual Instruments (VIs)*.

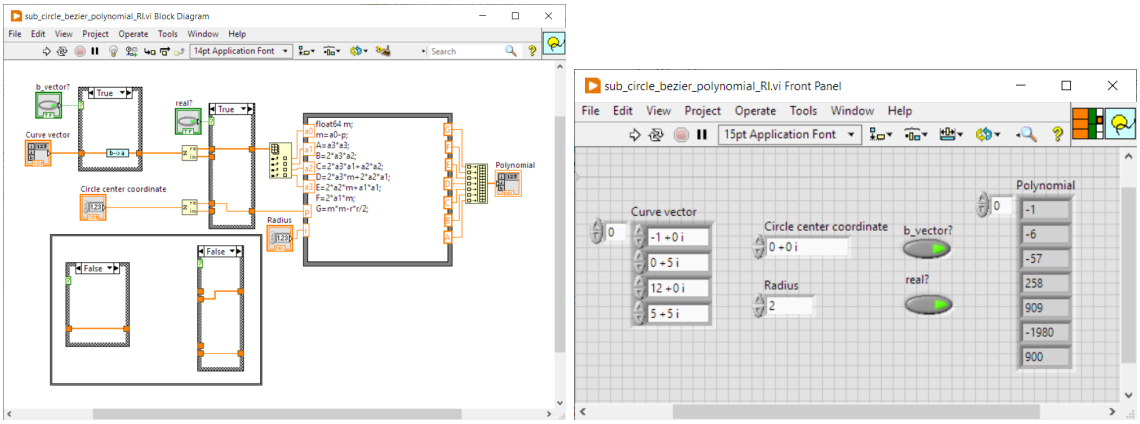


Figure 24: LabVIEW calculation of the polynomial coefficients.

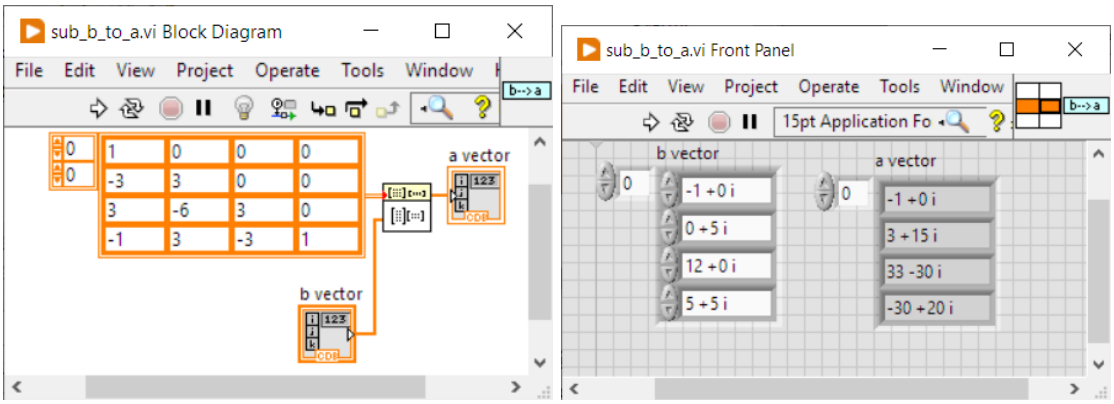


Figure 25: LabVIEW conversion of $\boldsymbol{z}$ to ζ coefficients according to Eq. 17 (here denoted $\boldsymbol{b}$ and $\boldsymbol{a}$).

5.11.2 Divide and conquer method

The idea here is to recursively check for circle intersections with the curve bounding box, while gradually splitting the curves at $t = 0.5$. The recursion is stopped, whenever the absolute value of the difference between the end points is smaller than the precision desired. in this case the result is added to a global array. Recursive calls are only performed, if the circle intersects with the bounding box. Otherwise nothing happens in the calling t segment.

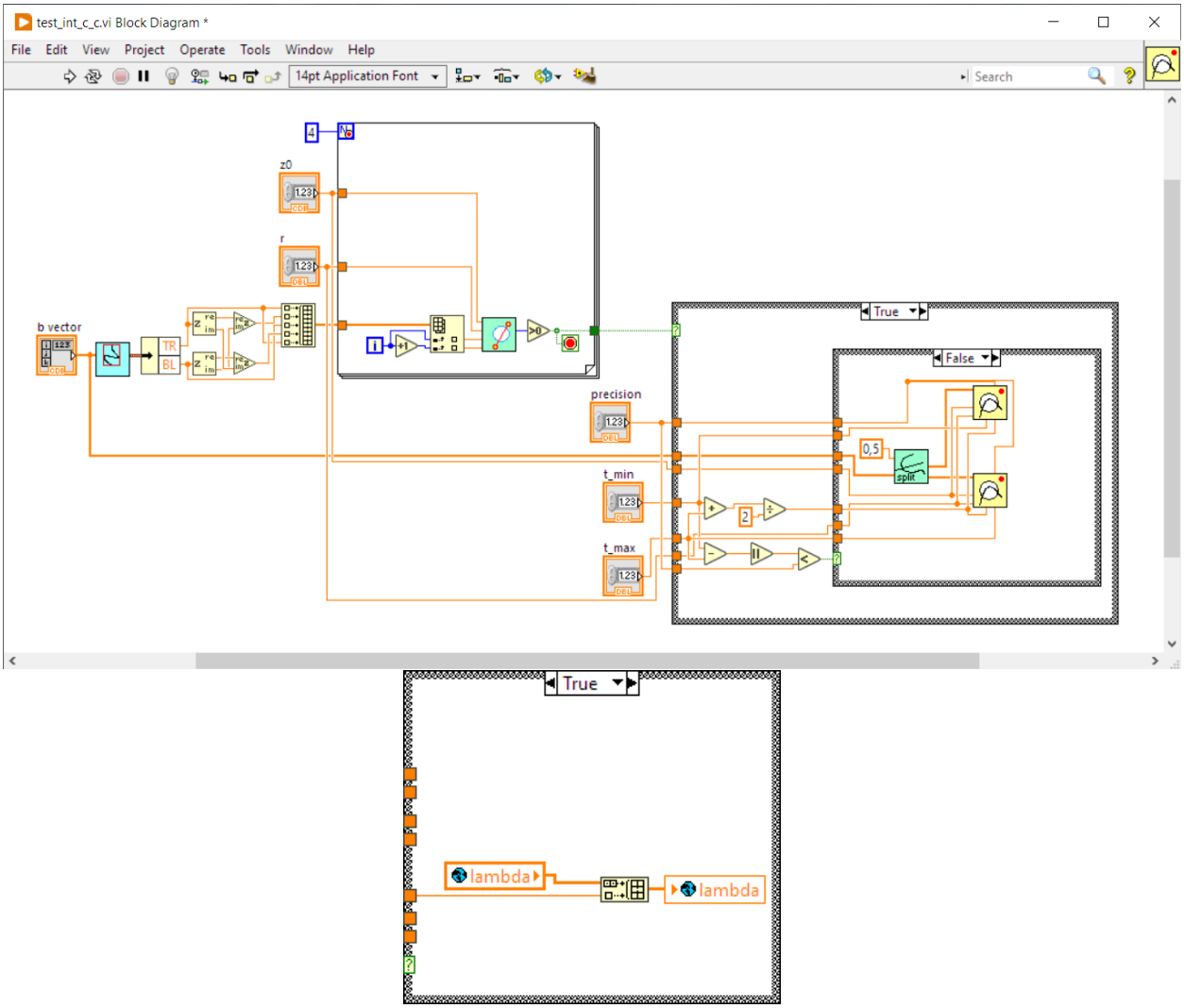


Figure 26: LabVIEW implementation of a recursive intersection method.

5.11.3 Execution speed evaluation

The recursive solution executes within 2.6ms, whereas the algebraic method runs at 47.9μs (!!!) on our 3.2GHz PC. LabVIEW finds the roots of a 6th degree polynomial very rapidly. So, the algebraic method clearly wins here.

5.12 Offset curve intersection issue

Obviously, the CNC machine may not be permitted to cross the curve boundary in the case of a tool diameter which is greater than the minimal curvature radius. in such a case, the untrimmed generated path has singularities and a cross-intersection as shown in Fig. 27.

We present here an efficient method for the yielding of the point on the toolpath curve, where the machine has to switch from one parameter λ to μ, in order to avoid collisions with the concave curve part. In fact, this point is the location of the local offset self-intersection. Now, one issue is that the offset curve describes a polynomial curve including variable square-root parameters, which cannot be solved by simple means. Another issue arises, if the tool radius is greater than a critical value, where the tool circle touches one of the

BÉZIER curve end-points (cf. Fig. 28) In such a case, there is no self-intersection that can help identifying the allowed parameter t values.

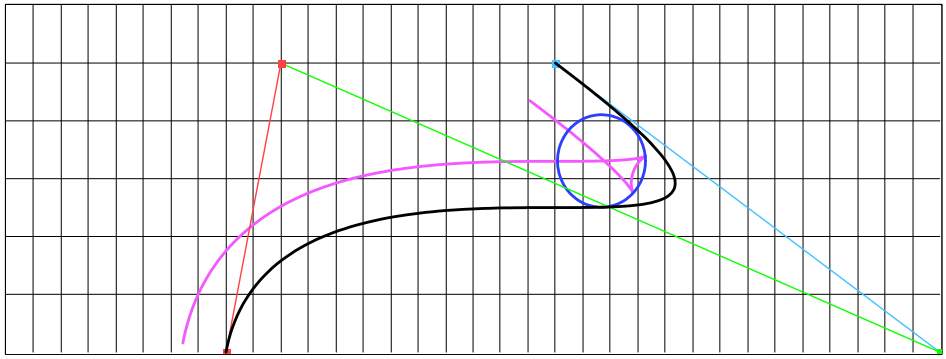


Figure 27: A greater offset distance greater than the minimal curvature radius generates a complex toolpath with cusp singularities and self-intersection that has to trimmed.

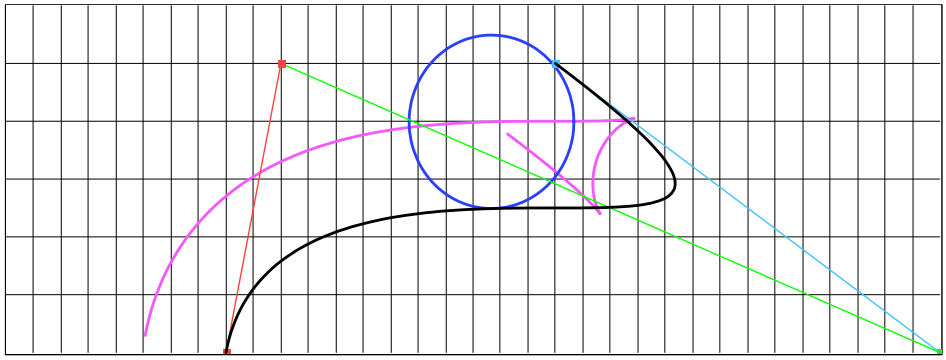


Figure 28: The tool radius generates the collision with curve end-point. The offset curve doesn't present any self-intersection.

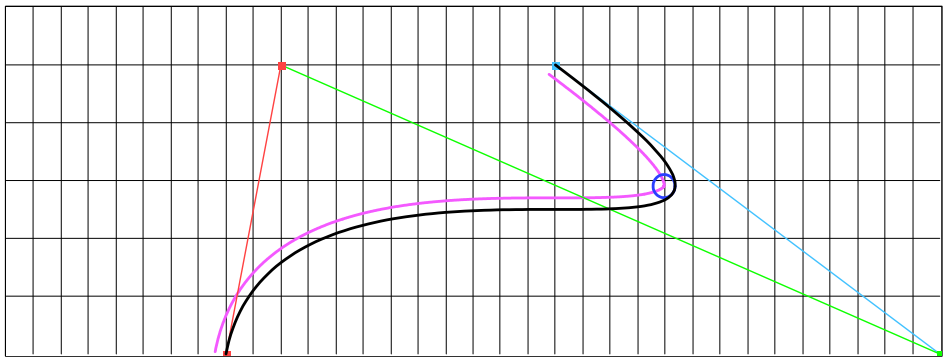


Figure 29: A smaller tool radius doesn't generate any singularity.

Case 1: The offset curve presents a self-intersection.

The self-intersection point has the particularity that the distances to both parametric locations $\boldsymbol{C}(\lambda)$ and $\boldsymbol{C}(\mu)$, which define the self-intersection, are equal. In other words, the self-intersection point must be located on the curve of the centers of all the tool circles with growing radii that are tangent to the curve concavity at two points. Because concave parts of regular cubic BÉZIER curves (without loops) represent almost parabolas the circle center curve can be considered the bisector line, which we admit here without proof.

We can construct that line by taking for center of a circle the point of minimal curvature radius $\boldsymbol{C}(\boldsymbol{t_{min(\rho)}})$. This circle with radius d, for example, cuts the curve concavity at two more points. These points describe a triangle, whose perpendicular bisector at $\boldsymbol{C}(\boldsymbol{t_{min(\rho)}})$ yields represents the searched line. In order to find the line equation $ax + by + c = 0$, as explained in section 5.1, one can choose as second required point on the line the middle of the two constructed circle intersection points (cf. Fig. 30).

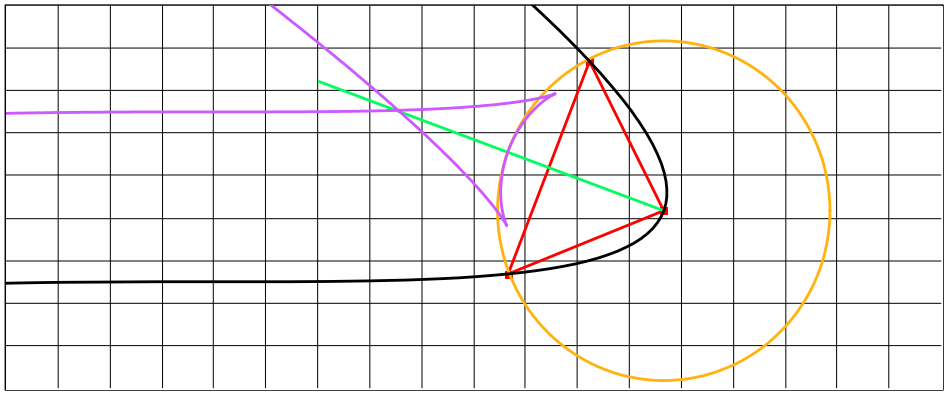


Figure 30: Constructing the perpendicular bisector.

Now the cusp trimming is effectuated on the fly during the G-Code conversion, which requires a try-and-error scan of all the curve points. We have to evaluate:

$$f(t) = ax(t) + by(t) + c \tag{42}$$

If this function is zero at $t = \lambda$, or more practically, if $\mid f(\lambda) \mid < \epsilon$, where ϵ is some very small positive tolerance value, we have located the curve intersection with the bisector. Starting from this point until the

next intersection point at $t = \mu$, while ignoring the point at $t = t_{min(\rho)}$, no G-Code may be generated. The offset curve and the G-Code generation then restart at $t = \mu$.

Case 2: The offset curve doesn't present a self-intersection despite the existence of cusp singularities.

Fig. 31 shows well that in this particular case the tool circle (with radius larger than the distance of the curve end point to the bisector line) collides with the curve end point before reaching the intersection point of the bisector with the offset curve. In other cases tool circle contact with curve is reached only once passed this point. If the tool radius is greater than the distance of the relevant curve terminal point to the bisector line a self-intersection of the offset curve cannot take place.

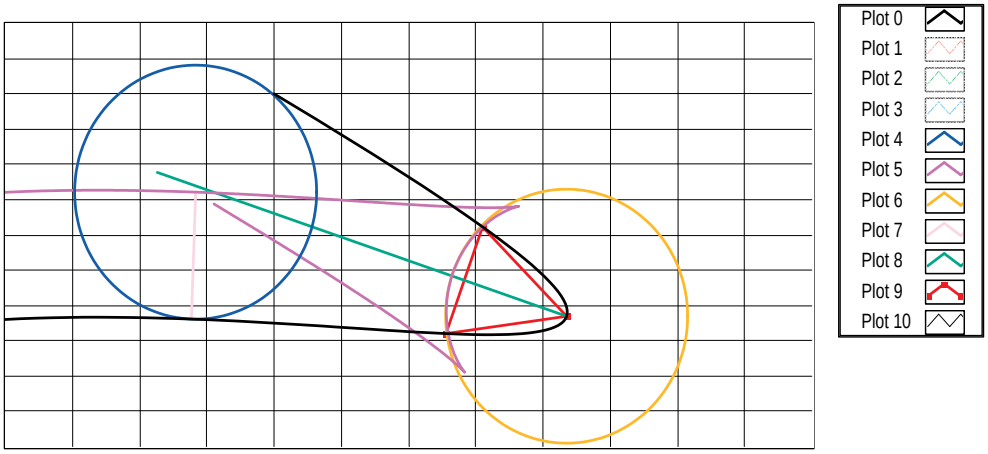


Figure 31: The tool circle collides with the end point of the curve.

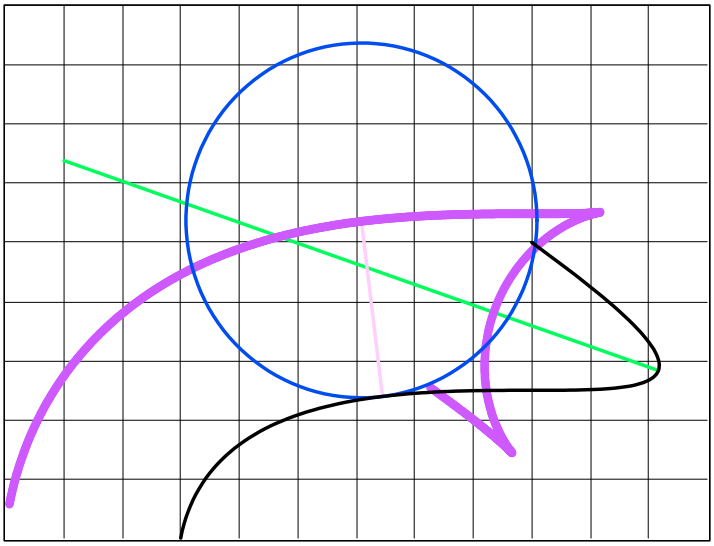


Figure 32: Yielding the tool motion limit at the end point proves to be a mathematically intractable problem.

Fig. 32 shows that we run into a mathematically intractable problem, which consists in finding the parameter

value λ, where the tool motion should stop, in order to avoid curve crossing, which would in reality damage the work piece. The solution would represent the yielding of the intersection of a circle with the non-polynomial offset curve. We must therefore define a numerical solution here. The easiest method seems to verify on the fly, if the distance of the actual, untrimmed offset point to the terminal point is less than the tool radius.

5.13 Closed curves

A BÉZIER curve of degree n is closed, if $z_0 = z_n$. It must be underlined that in such a case, there exist two different curve derivatives for the same point in the Cartesian or complex plane, because it is met by two distinct curve parameters $t_0 = 0$ and $t_n = 1$. This represents a curve discontinuity that has to be handled specially (cf. Section 5.14).

5.14 Piecewise cubic Bézier curves

In this project we do not strictly stick to the definition by [1] of a the piecewise BÉZIER curve, which uses an interval extension for the parameter t. Instead, we build an M sized array of cubic BÉZIER curves with the condition that if $\boldsymbol{C_3}[k](t)$ and $\boldsymbol{C_3}[k+1](t)$, with $k \in [0, M-1]$, are consecutive cubic BÉZIER curves, then the end point of $\boldsymbol{C_3}[k](t)$ and the starting point of $\boldsymbol{C_3}[k+1](t)$ coincide:

$$\boldsymbol{z_0}[k+1] = \boldsymbol{z_3}[k] \tag{43}$$

If $\boldsymbol{z_0}[0] = \boldsymbol{z_3}[M-1]$, the piecewise BÉZIER curve is closed.

During the toolpath generation process developed here, every sub-curve of a piecewise BÉZIER uses parameter $t \in [0, 1]$.

5.14.1 Handling end points

Although the definition of a piecewise BÉZIER curve guarantees continuity at the starting & end points –we will use the term end points here–, the piecewise curve is not necessarily differentiable. In fact, in many cases, a smooth joint of two adjacent cubic curves is given by the C^1 continuity. We define the C^1 differentiabilty at the joint as follows:

$$\arg\left\{\boldsymbol{C'_3}[k+1](t=0^+)\right\} = \arg\left\{\boldsymbol{C'_3}[k](t=1^-)\right\} + 2j\pi \text{ where } j \in \mathbb{N} \tag{44}$$

If this condition is not fulfilled, the piecewise BÉZIER curve has a singularity at the joint, which requires a special handling of the toolpath.

Fig. 33 (left) demonstrates well the cusp issue. Because the cubic polynomials of the real and imaginary parts present a maximum or minimum at the same parameter value $t = \lambda$, both corresponding derivatives are zero, and a tangent line cannot be drawn, as explained in Section 5.8.1. Therefore the convex toolpath curve jumps from one end point to the next. In a real milling process, this would represent an undesired tool collision with the workpiece.

If the BÉZIER control points are slightly changed, the cusp becomes smooth as can be seen in Fig. 33 (right), and the toolpath changes to an arc with radius d (we admit this without proof).

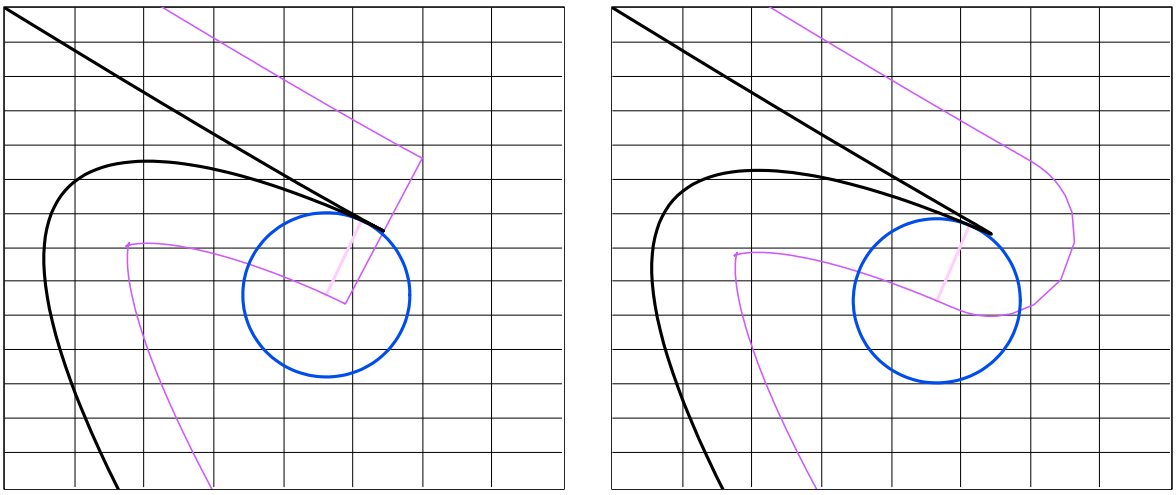


Figure 33: Minimal changes of the control points make the cusp smooth.

If the unsmoothed BÉZIER curve was split into two sub-curves at the cusp location, as recommended in Section 5.8.1, the total curve becomes a piecewise curve, and the toolpath of the first sub-curve can be processed from one end point to the other. After this, the tool should move over an arc with radius d(=tool radius and also offset curve) ending at the offset starting point of the following curve. This procedure may be applied to any singularity joint.

If the offset curve is drawn on the concave side of the curve at an end-point cusp, we must apply an additional algorithm. Unfortunately, it may happen that the left and right tangent lines at the joint are identical (C^1 continuity), so that there is no cusp, but well a critical concavity.

Fig. 34 shows this mathematically intractable issue (yellow circle), which consists in finding the intersection point of the untrimmed offset curve in the case of C^1 continuity at the joint of two different BÉZIER curves. Visibly, the curvatures of both neighboring sub-curves are not identical, indicating that there is no C^2 continuity. The consequence is that the offset curve intersection point is not located on the bisector line, but rather on a bent curve that can hardly be described algebraically.

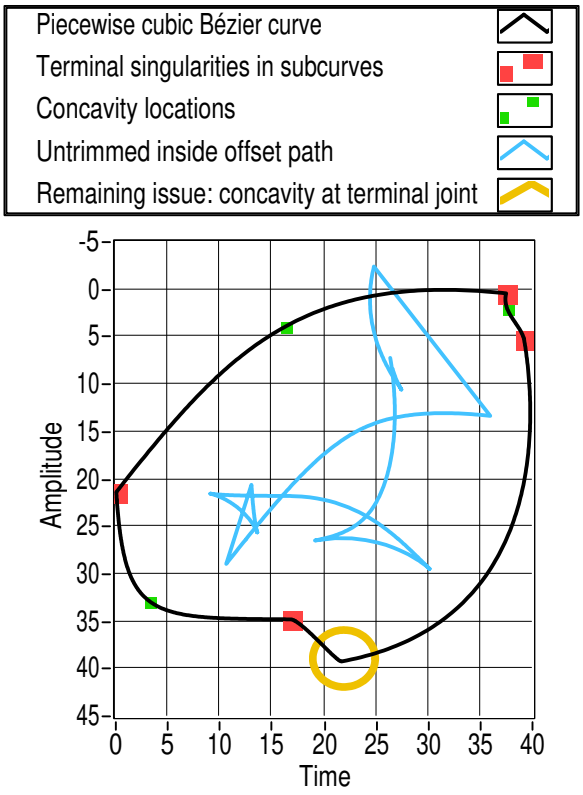


Figure 34: Remaining issue.

5.15 Approximate distance between two Bézier curves

We will need to know, if two BÉZIER curves are closer to each other than the tool radius, in order to identify potential toolpath collisions with other pieces of the curve than the actually machined part.

The idea here is to draw a (normally short) list of potential collision candidates for each curve piece, which are then checked on the run for circle and curve intersection. Because this is an execution acceleration measure only, it is not required to know the exact distance; an approximation will be sufficient.

Therefore, we propose to use the bounding rectangle of the BÉZIER control quadrilateral, and evaluate the distance between the rectangles. The corresponding vi is displayed in Fig. 35. It first checks, if the rectangle points are valid. Rectangles may consist of valid segments, where the end points don't coincide. The sub,vi then verifies, if one of the rectangles is located above or left of the other. It applies the Boolean function table exposed in Table 2.

Left	Above	R=Left or Above
0	0	0=Overlap
0	1	1
1	0	1
1	1	1

Table 2: Possible relative position of two x- and y-axis parallel rectangles.

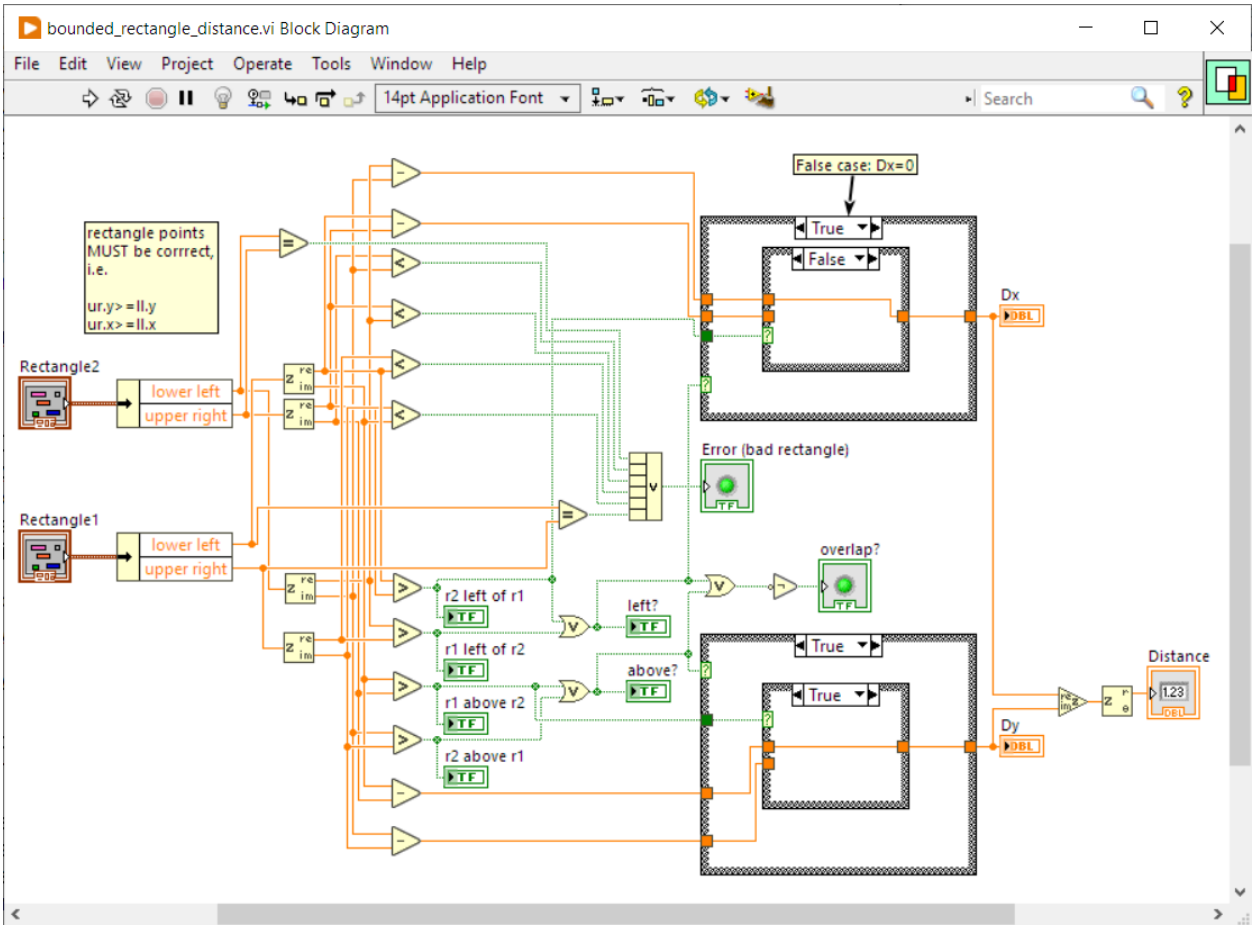


Figure 35: Finding the distance between two bounding rectangles defined by their respective lower left and upper right points. (Such a vi diagram is difficult to read; a text based programming environment would certainly be more appropriate here.)

5.16 Intersection of two cubic Bézier curves

For the same reason explained in Section 5.8 concerning the single Bézier curve, a piecewise curve should not present self-intersections, because the machine cannot validly process such a curve. Catastrophic curve crossings would be the result. Therefore the CAD/CAM program should detect eventual intersections and notify their existence to the user, in order to correct the input curve by eliminating these undesired curve characteristics.

Calculating the intersections of two cubic curves algebraically is an almost intractable task, because the curves could intersect in as many as 9 points (cf. [1, pp. 159-160]). [1] proposes a divide-and-conquer solution to the problem, which consists in gradually sub-dividing the concerned curves while checking for overlap of the bounding rectangles until the sub-curves are nearly linear.

In the present project, we will in a first time only check for overlap without proceeding to a gradual subdivision. Although this will not necessarily prove the presence of self-intersections, the user will be notified about critical curve sections, where he or she has to verify for integrity. Only in a second stage will we complete the program by adding the divide-and-conquer algorithm.

5.17 Optimal step Δt

Digital numerical computations are based on discretization. In the present case, the most sensitive variable is the parameter $t \in [0, 1]$. Generating the offset toolpath curve is operated at small constant steps Δt. In fact, because the resulting G-Code is mainly made of small segments, the quality for the machining process depends on the good choice of the step. A remarkable feature of BÉZIER curves is that, while Δt is considered constant, the corresponding curve step size Δs is not. In fact, the higher the curvature, the smaller the curve step size. The result is that more G-Code rendering points are generated at high curvature than in an almost linear sections. In other words, the milling machine has to process more points around a curvature than on a line. This is essentially important for work precision. The trade-off is that the machine must be able to accelerate or decelerate correctly, in order to keep the effective tool speed constant, which is desired for the cutting quality and the drill wear.

A useful method of determining Δt is by computing the arc length of each sub-curve, which is divided by a general parameter *precision* p (cf. [1, p.103]):

$$\Delta t = \frac{\int_0^1 v(t)dt}{p} \tag{45}$$

5.18 Convert svg file to piecewise cubic Bézier curve

5.18.1 Svg coordinate system

There are a few things to observe, when converting **svg** paths to G-Code. First, the user must pay attention that the **viewBox** (=canvas) coincides with the processed curve. Otherwise, parts of the curve might get lost in the process. Also, he or she has to consider the stroke width used by the graphics editor. Note that the underlying path has zero stroke width being set of mathematical dimensionless points. For instance, in the freeware INKSCAPE editor the underlying path represents the center line of the rendered curve. However, INKSCAPE draws the bounding box outside the shape. Conversion to another CAD/CAM software might therefore be erroneous in width and height.

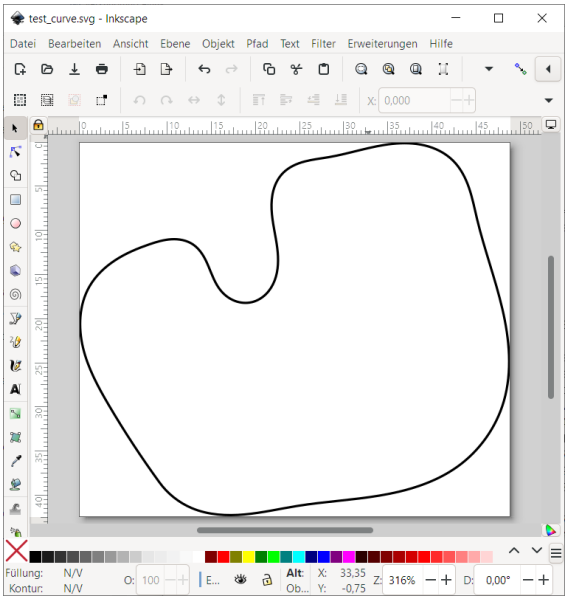


Figure 36: The INKSCAPE vector graphics editor renders a curve. Coordinate system is $\downarrow^{\rightarrow}$. In this case, the unit is mm. The **viewBox** (=canvas) seems to coincide with the bounding rectangle of the curve. In fact, it is increased in two dimensions by the stroke width.

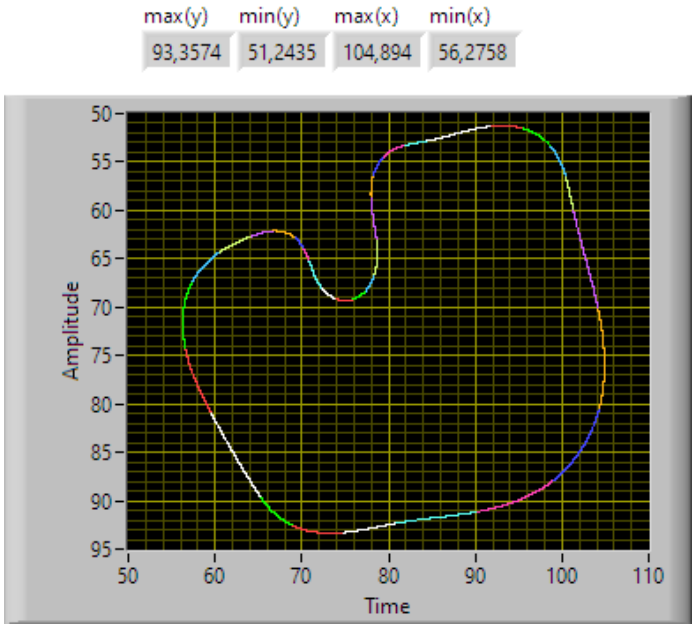


Figure 37: The sub-curves are made visible by using colors. The **svg** path is expressed in user coordinates, here *mm* (see the bounding box values in Listing 2!). Note that the translation has not been operated yet. Also note the minimal differences to the translation vector in Listing 2 due to the stroke-width.

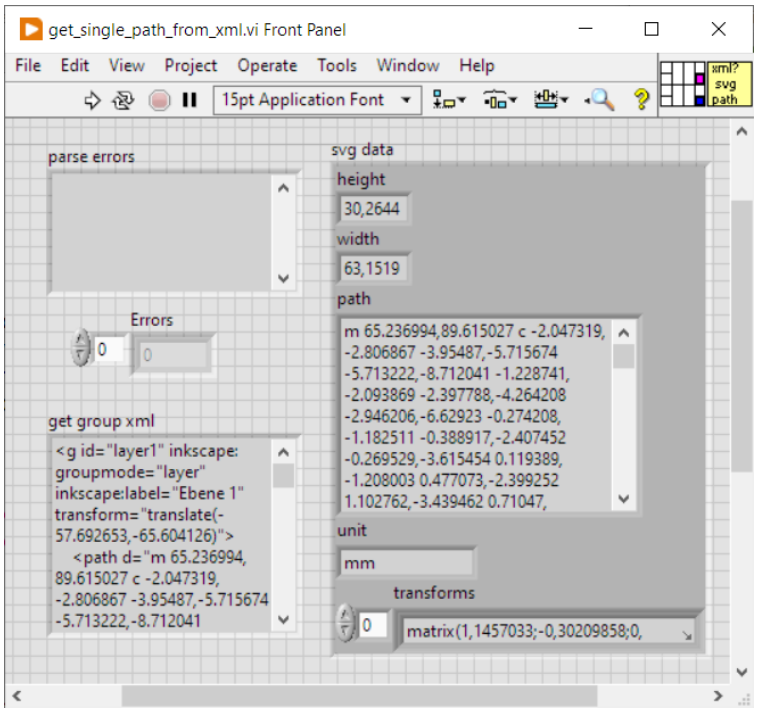
The **svg** format uses the attributes **width** and **height** for the scaling of the figure. It is essential for precise machining that our CAM program takes the stroke width into account. Note the **svg** tag *translate(-56.143201,-51.111133)* in Listing 2. In this example, our program must translate by the indicated values, which are slightly smaller than the bounding box minima. A closer look reveals that the difference is half the stroke width.

Listing 2: Extract of an **svg** file generated in INKSCAPE

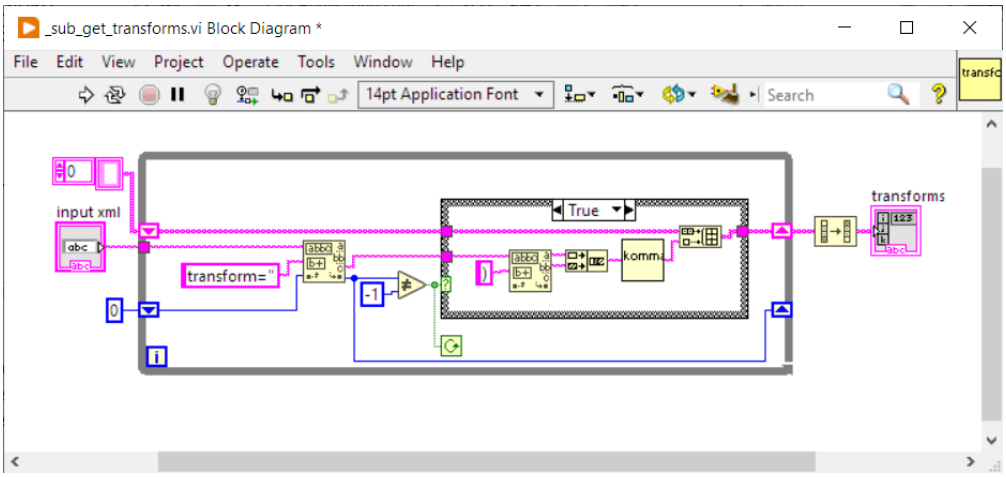
```
<svg
   width="48.883556mm"
   height="42.378716mm"
   viewBox="0 0 48.883556 42.378716"
   version="1.1"
   id="svg1"
   ...
   <g
     inkscape:label="Ebene 1"
     inkscape:groupmode="layer"
     id="layer1"
     transform="translate(-56.143201,-51.111133)">
    <path
       style="fill:none;fill-rule:evenodd;stroke:#000000;stroke-width:0.264583px;stroke-linecap:
            butt;stroke-linejoin:miter;stroke-opacity:1"
       d="m 65.236994,89.615027 c -2.042615,-2.810176 -3.94996,-5.718669 -5.713222,-8.712041
            -1.209652,-2.053546 -2.364169,-4.178997 -2.920492,-6.496498 -0.556322,-2.317501
            -0.4604,-4.875562 0.729276,-6.940744 0.704741,-1.223373 1.760271,-2.220798
            2.944437,-2.989597 1.184166,-0.768799 2.497232,-1.319268 3.832644,-1.777499
            0.81288,-0.278929 1.646376,-0.526985 2.504289,-0.577571 0.857914,-0.05059
            1.750446,0.110482 2.461849,0.592645 0.426771,0.28925 0.774785,0.683896
            1.055977,1.116019 0.281192,0.432123 0.498422,0.902478 0.697811,1.377919
            0.398777,0.950881 0.742144,1.950204 1.400848,2.743492 0.421274,0.507347
            0.966508,0.91553 1.58264,1.15059 0.616132,0.235061 1.302849,0.293085 1.943672,0.137464
            0.597173,-0.145022 1.144132,-0.471843 1.580346,-0.9047 0.436213,-0.432857
            0.763797,-0.969098 0.983676,-1.542945 0.439758,-1.147694 0.448516,-2.417842
            0.300239,-3.637926 -0.179473,-1.476767 -0.579519,-2.926893 -0.646055,-4.413037
            -0.03327,-0.743072 0.01855,-1.495041 0.220304,-2.210973 0.201752,-0.715933
            0.558458,-1.396058 1.082441,-1.923984 0.654434,-0.659359 1.535625,-1.051426
            2.433611,-1.289456 0.897987,-0.238031 1.826182,-0.335455 2.741848,-0.492281
            2.423031,-0.414992 4.770031,-1.248256 7.214924,-1.504764 1.222447,-0.128254
            2.470194,-0.108996 3.665167,0.178867 1.194973,0.287863 2.337659,0.854676
            3.208972,1.721647 0.856596,0.852328 1.42111,1.959301 1.813204,3.102314 0.3921,1.143014
```

```
0.62404,2.33336 0.9052,3.508591 0.8204,3.429223 2.06484,6.746497 2.86757,10.179901
0.80273,3.433403 1.1497,7.073834 0.16385,10.459205 −0.8636,2.965567 −2.75493,5.619362
−5.269789,7.412648 −2.563416,1.827911 −5.673076,2.737427 −8.778194,3.257621
−3.105117,0.520194 −6.261146,0.681911 −9.369631,1.181592 −1.994212,0.320565
−3.96906,0.780205 −5.980775,0.960898 −2.011714,0.180693 −4.094332,0.0691
−5.956242,−0.713801 −1.47341,−0.619542 −2.76965,−1.65418 −3.700395,−2.953596 z"/>
  </g>
</svg>
```

5.18.2 Extract the relevant code from svg file



Svg files follow the **xml** rules. Fortunately LabVIEW has powerful built-in tools for parsing such files. Fig. 38 shows how a single path data can be extracted from an **svg** file.



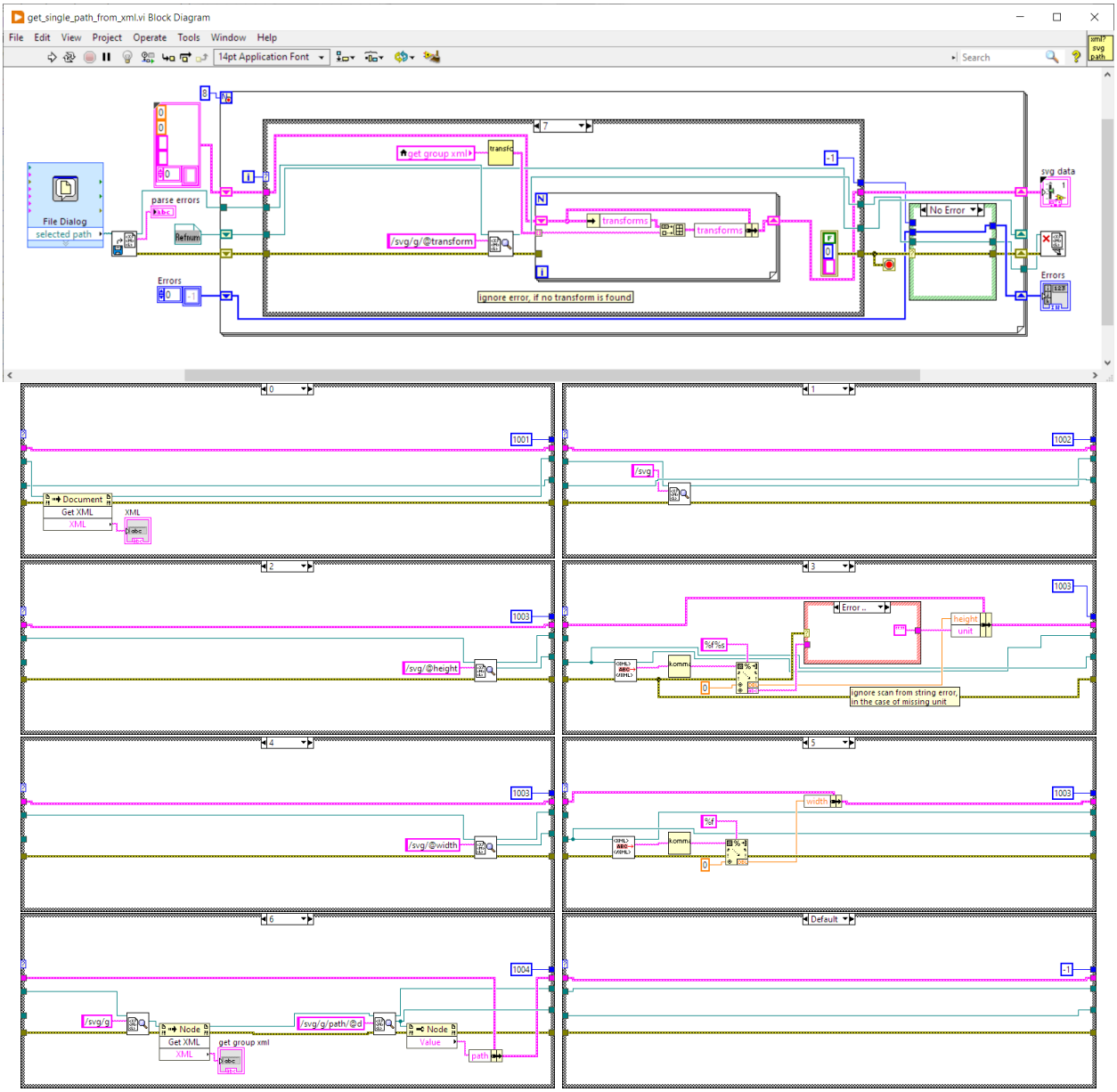


Figure 38: This sub.vis extracts the path data from an **svg** file.

5.18.3 Convert to commands

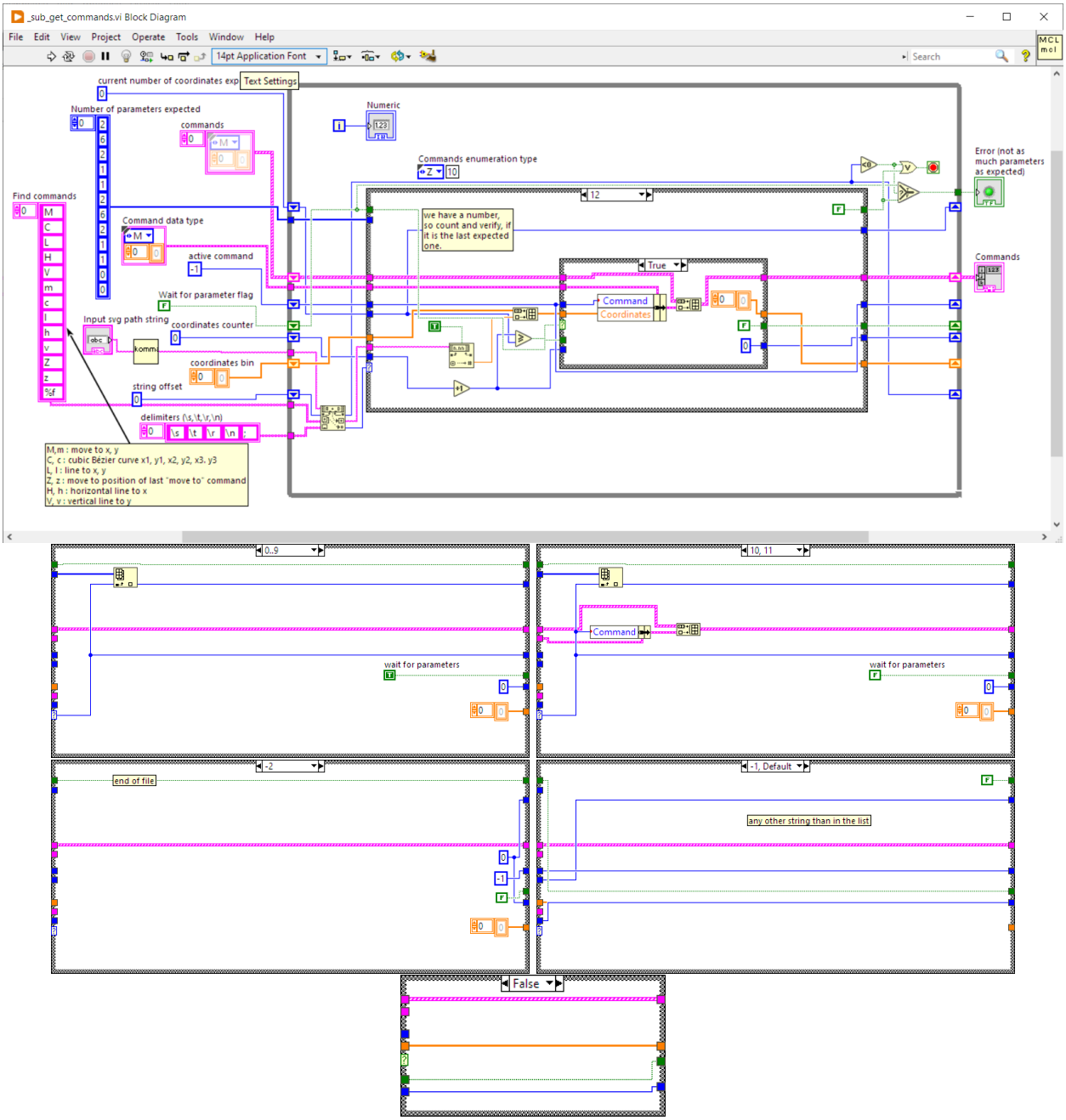


Figure 39: This sub.vis converts the relevant **svg** code to an array of commands.

Important notice: It is essential for this to work in LabVIEW that the decimal point is changed to the system decimal point that LabVIEW uses automatically. If the comma is configured, the code shown in fig. 40 replaces the comma by the semi-colon, and the point by the comma.

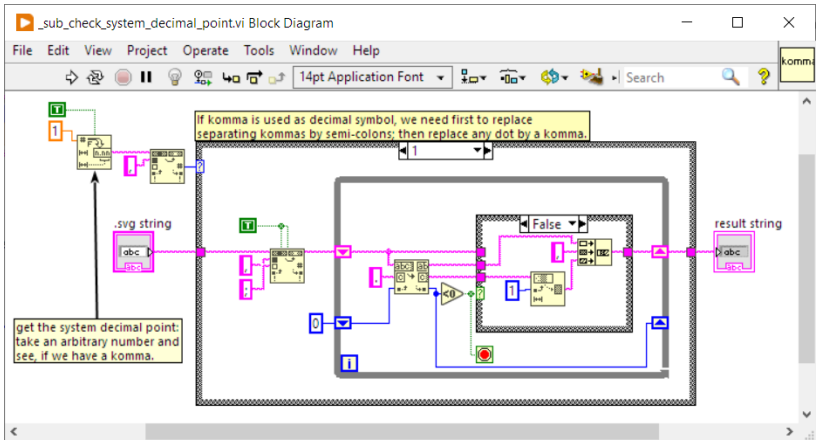


Figure 40: Comma issue.

5.18.4 Changing relative to absolute coordinates and completing control point list

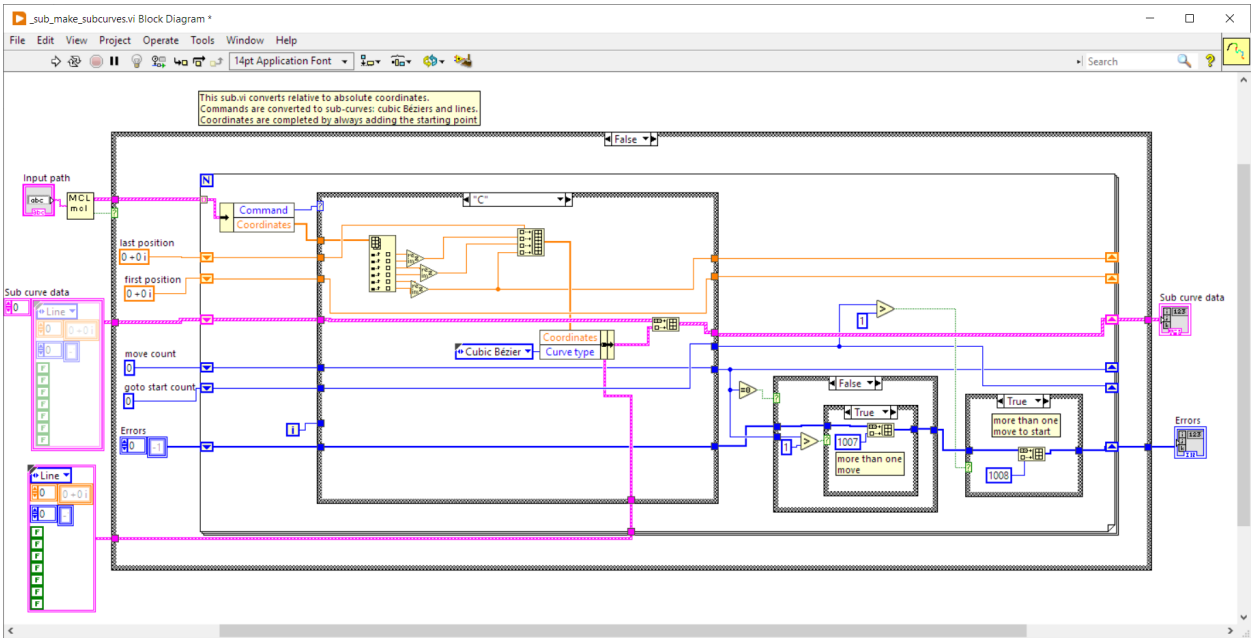


Figure 41: This vi makes complete lines and curves in absolute coordinates. It trows an error message, if discontinuities are detected in the curve.

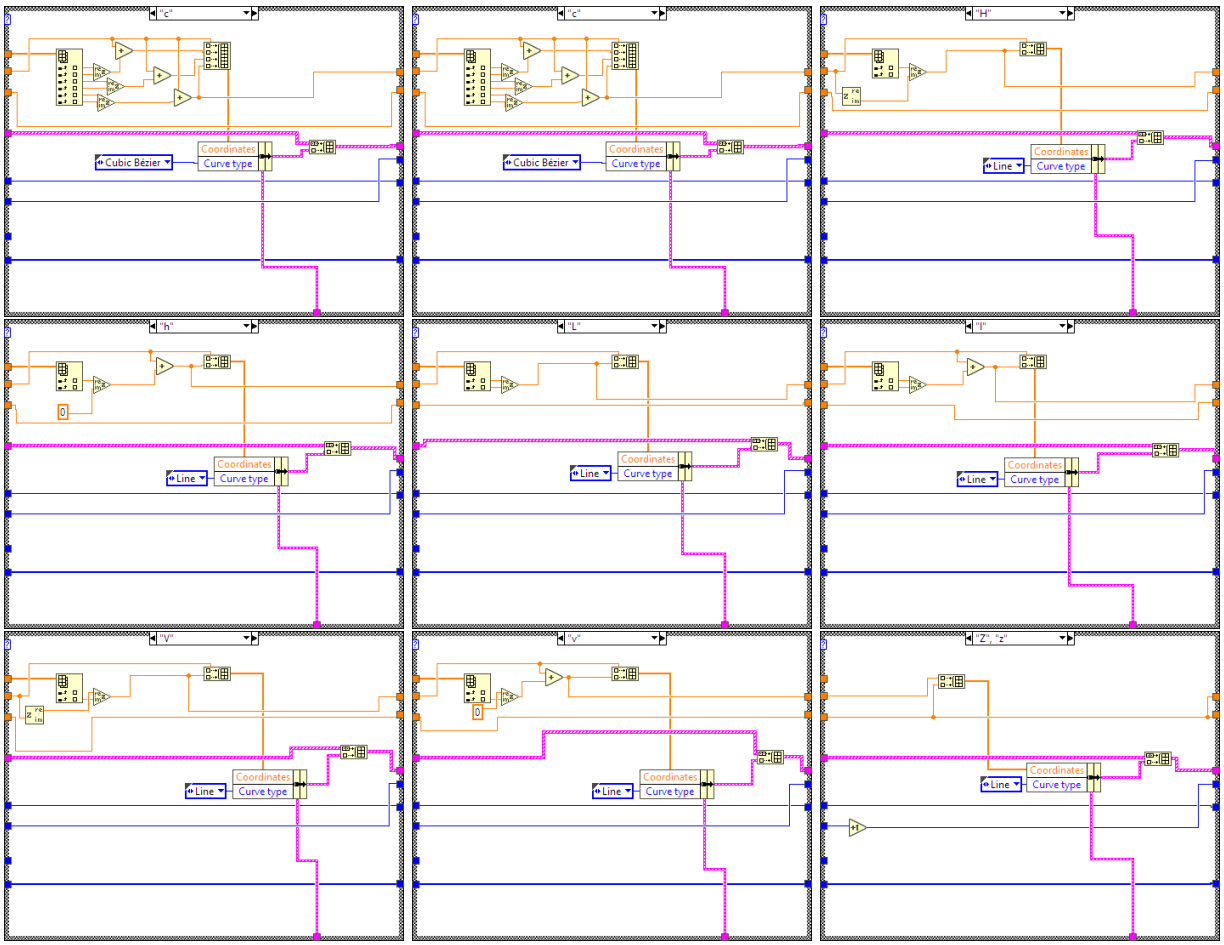


Figure 42: The different cases of diagram shown in Fig. 41.

5.19 Easy processing of affine transformations using homogeneous coordinates (cf. [1], ch.1-2 and [14], appendix A)

Homogeneous coordinates provide a very exciting feature, which is that translations can be calculated with matrix multiplications, whereas in normal Cartesian coordinates this is not possible. The most useful application is that diverse transformations (translation, rotation, scaling, skewing, reflections, shears) can be concatenated by successive matrix multiplications to a single 3×3 matrix that serves as product term for any relevant point expressed in its homogeneous coordinates $\mathbf{p}(x, y, 1)$. (Note that this can be extended without any trouble to 3D). Eq. 46 shows the different transformations used in the **svg** format (1.1):

$$
\begin{aligned}
\mathbf{T}(h,k): &\begin{pmatrix} x_{new} & y_{new} & 1 \end{pmatrix} = \begin{pmatrix} x_{old} & y_{old} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ h & k & 1 \end{pmatrix} \\
\mathbf{R}(\theta): &\begin{pmatrix} x_{new} & y_{new} & 1 \end{pmatrix} = \begin{pmatrix} x_{old} & y_{old} & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
\mathbf{S}(s_x,s_y): &\begin{pmatrix} x_{new} & y_{new} & 1 \end{pmatrix} = \begin{pmatrix} x_{old} & y_{old} & 1 \end{pmatrix} \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
\mathbf{SKx}(\theta): &\begin{pmatrix} x_{new} & y_{new} & 1 \end{pmatrix} = \begin{pmatrix} x_{old} & y_{old} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ \tan\theta & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ Attention if } \theta = \pm\tfrac{\pi}{2} \\
\mathbf{SKy}(\theta): &\begin{pmatrix} x_{new} & y_{new} & 1 \end{pmatrix} = \begin{pmatrix} x_{old} & y_{old} & 1 \end{pmatrix} \begin{pmatrix} 1 & \tan\theta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
\mathbf{M}(a,b,c,d,e,f): &\begin{pmatrix} x_{new} & y_{new} & 1 \end{pmatrix} = \begin{pmatrix} x_{old} & y_{old} & 1 \end{pmatrix} \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{pmatrix}
\end{aligned}
\tag{46}
$$

Importante notices:

- Rotation, scaling, skewing are made about the origin
- Reflections and shears can be produced by combining these modules. For instance, a reflection about the x-axis is a $\mathbf{S}(-1,1)$
- A rotation about the arbitrary point $\mathbf{p_0}(x_0,y_0)$ needs first a translation to the origin by $\mathbf{T}(-x_0,-y_0)$, followed by the rotation $\mathbf{R}(\theta)$. The operation is closed by undoing the negative translation with $\mathbf{T}(x_0,y_0)$. Note that matrix multiplication is associative, but not commutative.

$$
\begin{aligned}
\mathbf{R}_{x_0,y_0}(\theta) &= \mathbf{T}(-x_0,-y_0) \circ \mathbf{R}(\theta) \circ \mathbf{T}(x_0,y_0) \\
&= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_0 & -y_0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_0 & y_0 & 1 \end{pmatrix}
\end{aligned}
\tag{47}
$$

- Similar methods can be used for scaling or skewing about an arbitrary point.
- Note that **svg** allows only a single transformation per path. In other words, if more consecutive transformations –except for the usual group translation– are being applied, then the **svg** generating software must operate the matrix multiplication and provide the overall matrix $\mathbf{M}(a,b,c,d,e,f)$.

5.19.1 Building the matrices

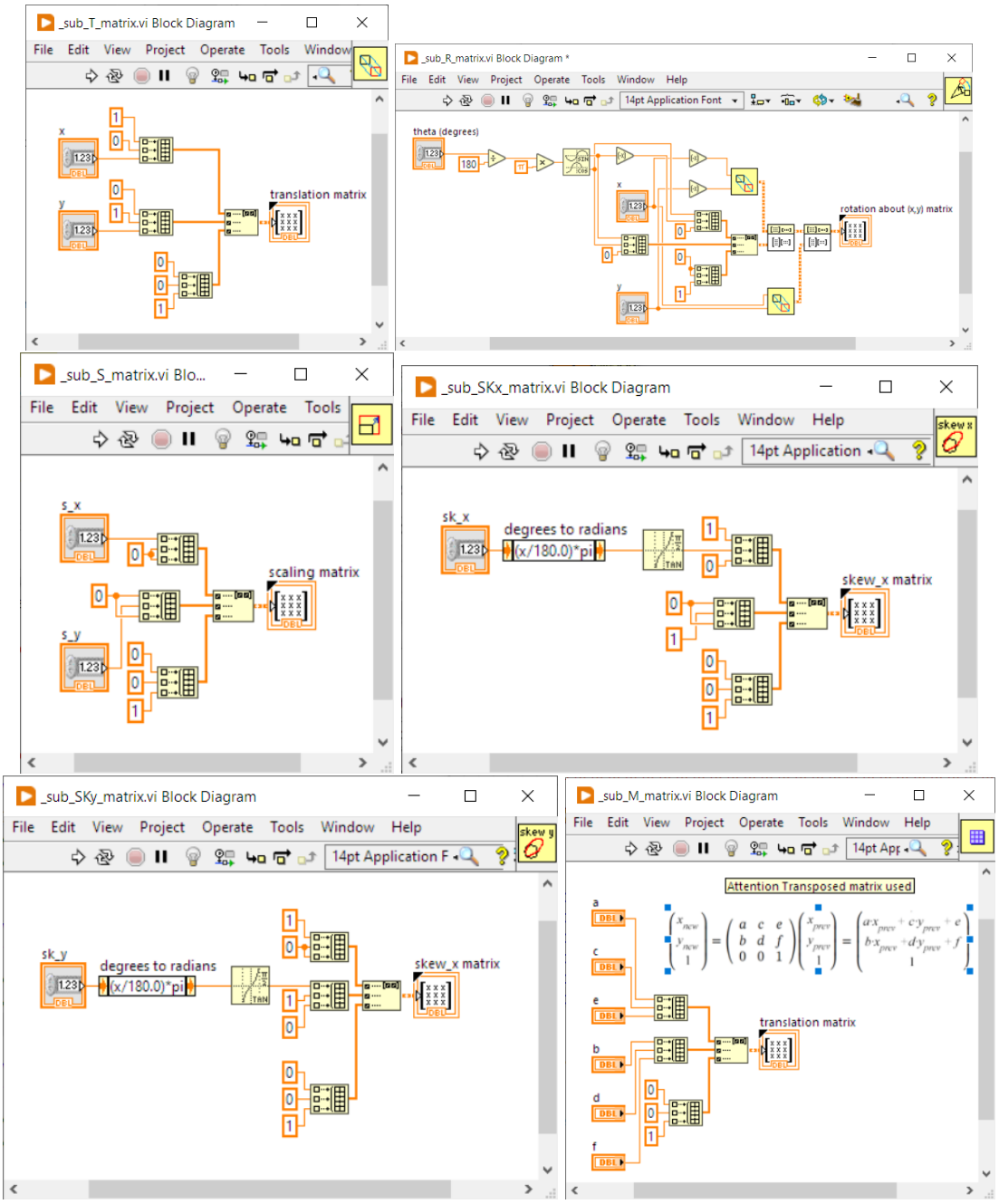


Figure 43: Building the matrices in LabVIEW Note that in order to have the correct screen appearance of the matrices in the front panel, the matrices are transposed.

5.19.2 Getting the transformation matrices from svg code

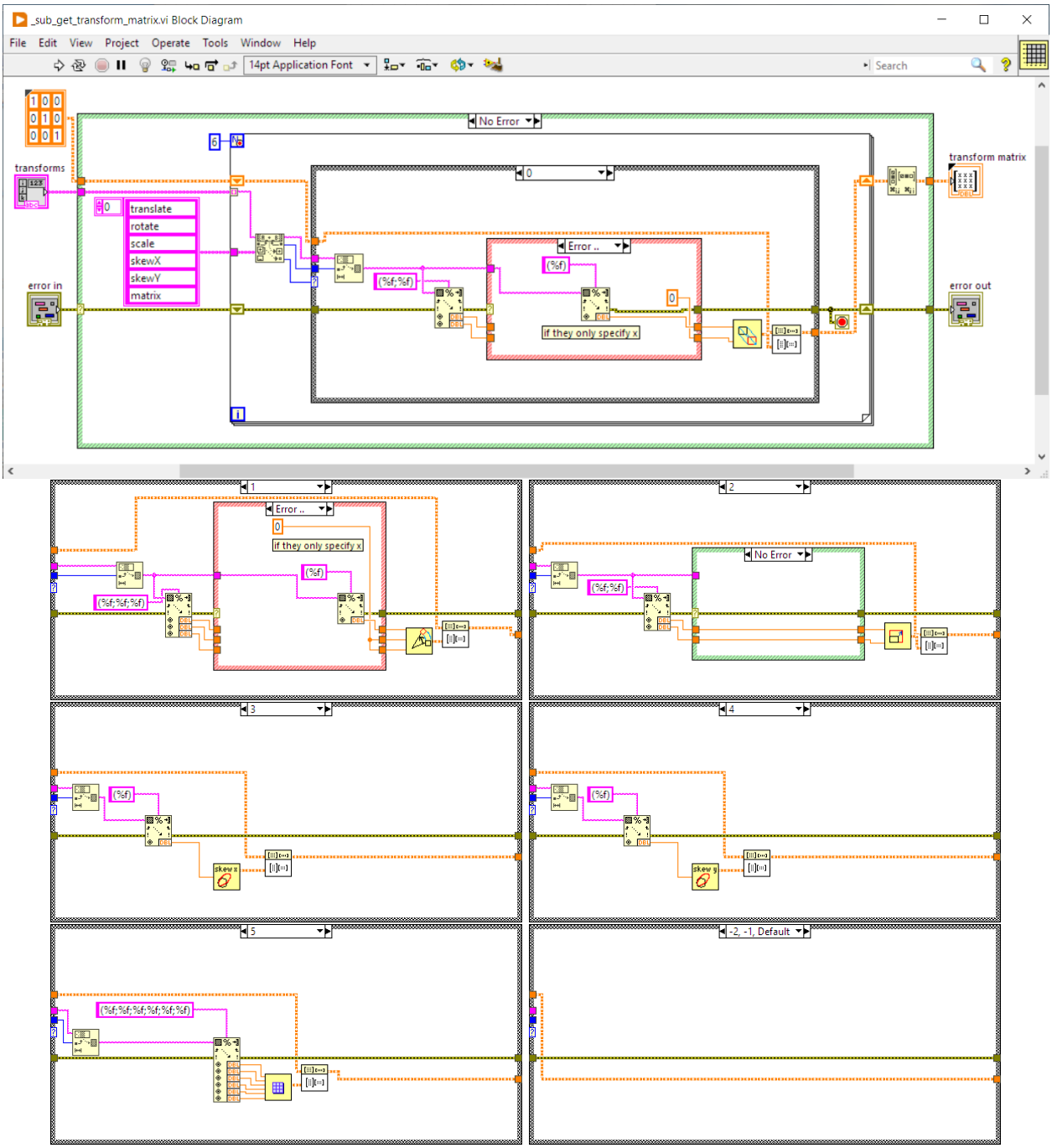


Figure 44: This LabVIEW diagram shows how the transformation matrices are operated. Note the matrix shift register that is consecutively multiplied with the next matrix in the list in order to generate the overall transformation matrix.

5.19.3 Testing the transformations

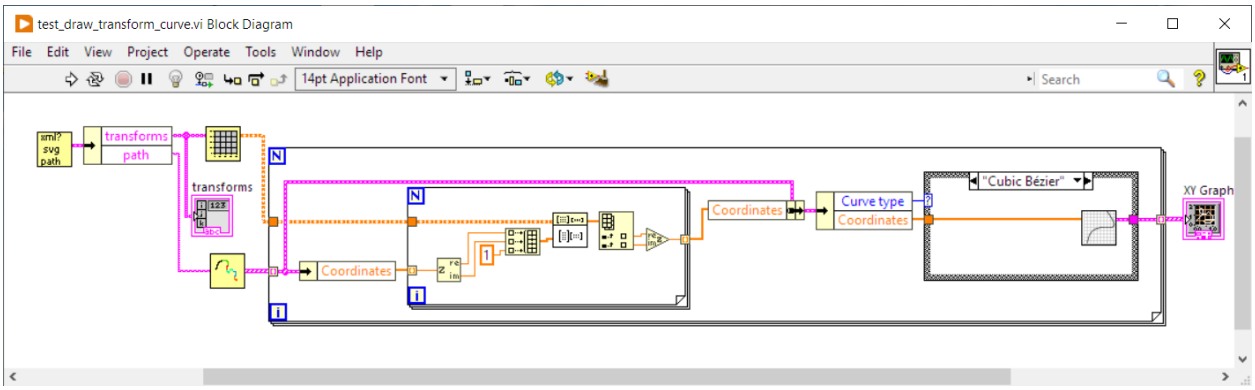


Figure 45: This vi calls various sub.vis to extract the path data from the **svg** file. The transformation matrix is also built and all the relevant curve control points homogeneous coordinates are multiplied with the transform matrix. Then the piecewise curve is rendered.

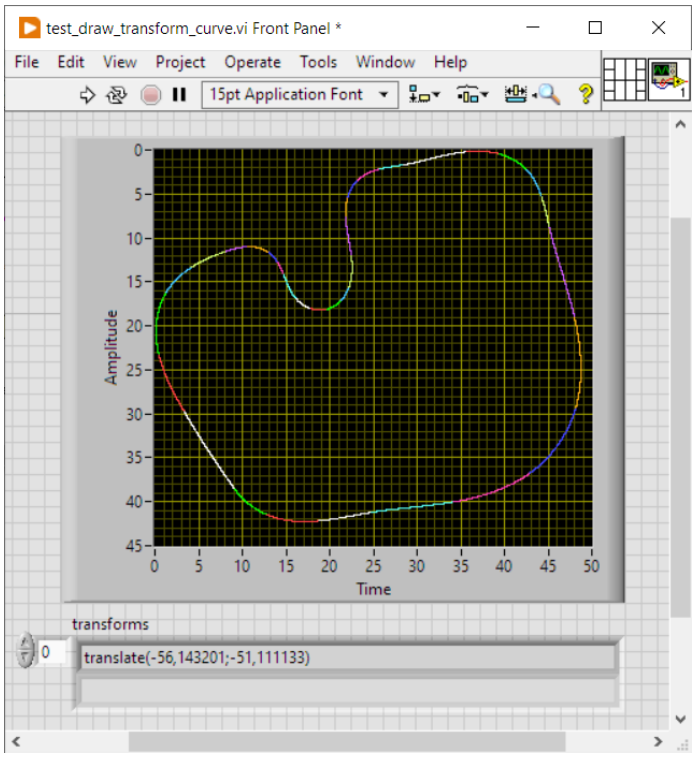


Figure 46: Translating the curve.

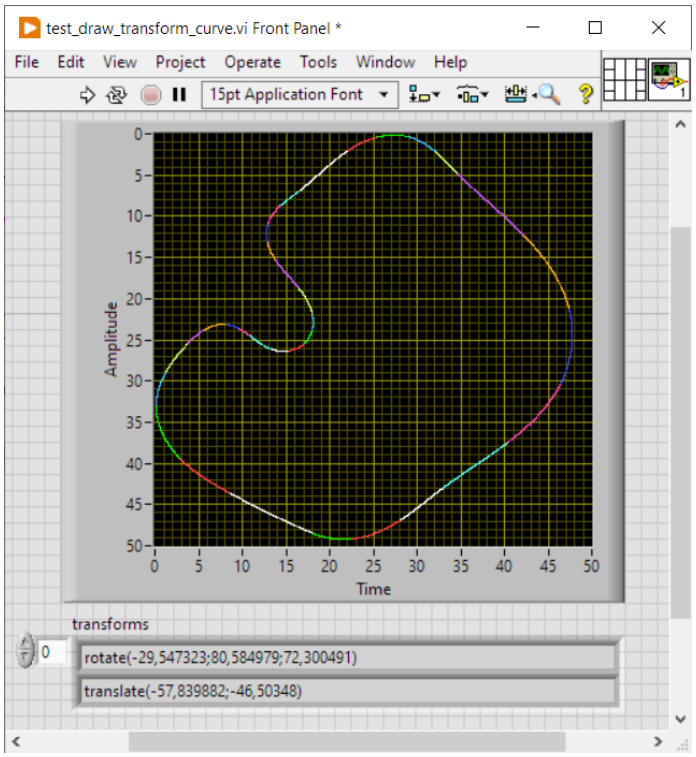


Figure 47: Rotating and translating.

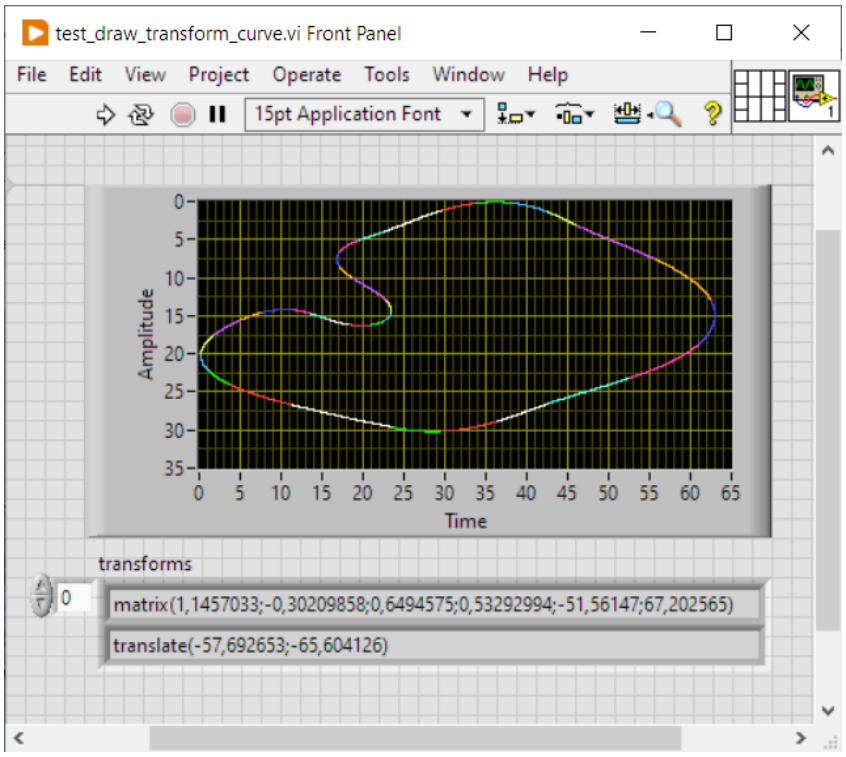


Figure 48: Rotating, scaling and translating.

6 Conclusion so far:

Because of the mathematically untractable issues detected in sections 29 & 5.14.1 (red marked text), we must conclude that tool collision detection can must be processed on the fly and that an *a priori* mathematical solution can only be conceived with unrealistically great effort that would end in very bad execution time.

7 Putting it all together: Converting a connected set of piece-wise cubic Bézier curves to a G-Code offset toolpath

The planned toolpath program should operate the following procedures:

1. Convert data from **.svg** file to piecewise cubic Bézier curve.
2. Piecewise cubic Bézier curve integrity check
   (a) Verify that all sub-curves are free of self-intersections (loops) → Stop and notify if not
   (b) Split sub-curves into more sub-curves at cusps
   (c) Verify that we have end-point continuity → Stop and notify if not[6]
   (d) Verify that there are no intersections between sub-curves → Stop and notify if not
3. Get the offset sign (answers the question: *Is offset curve located cis or trans relatively to the input curve?* or in the case a closed curve: *Is the offset curve inside or outside the closed curve?*) This also determines:
   (a) where the G-Code generation process starts and which way to proceed, i.e. $t : 1 \to 0$ or vice-versa. (Note that in general, **outside** milling paths should run anti-clockwise, whereas **inside** paths should run clockwise, if the cutter rotates clockwise. Program should allow climb milling as an exception only.)
4. According to the offset sign, identify problematic **concavities**:
   If the curvature radius is smaller than the tool radius (or equivalently the offset distance) d:
   → If the offset curve has a self-intersection → mark the curve as **concave critical with intersection** (Later during the G-Code generation process, no G-code may be added between $t = \lambda$ and $t = \mu$. These parameter values are determined by evaluation of Eq. 42 on the fly.)
   → If the offset curve doesn't have a self-intersection despite the existence of cusp singularities, i.e. if the tool radius is greater than the distance of the concerned end-point to the bisector → find the intersection of the circle centered at $\boldsymbol{C}(0)$ or $\boldsymbol{C}(1)$, respectively, with the bisector and start/stop the G-code generation process accordingly; → mark the curve as **concave critical with end-point issue**.
5. Create the **neighbor list** for each sub-curve:
   (a) preceding sub-curve
   (b) following sub-curve (open curve: if empty, this is the end sub-curve)
      i. Check for differentiablilty at the end-points → later during the G-Code generation process, the end-point circumvention must be added, in the case of a two different derivatives (or tangent lines); → mark sub-curve curve and its follower as **end-point cusp critical**
   (c) critical sub-curves (distance of bounding rectangles smaller than d) → on the fly, the program must check for circle/curve intersection with preceding, following and critical curves.
6. Start the G-Code generation process. At each step:

[6] Because of the **svg** structure, this should not be necessary, as consecutive segments and curves always assume their starting point to be the terminal of their predecessors.

(a) Find the next valid point on the offset curve

(b) Resolve concavity issue for those sub-curve marked **concave critical with intersection**

(c) Resolve concavity issue for those sub-curve marked **concave critical with end-point issue**

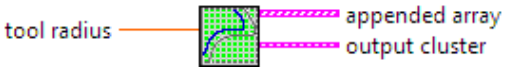
(d) Circumvent critical cusp end-points

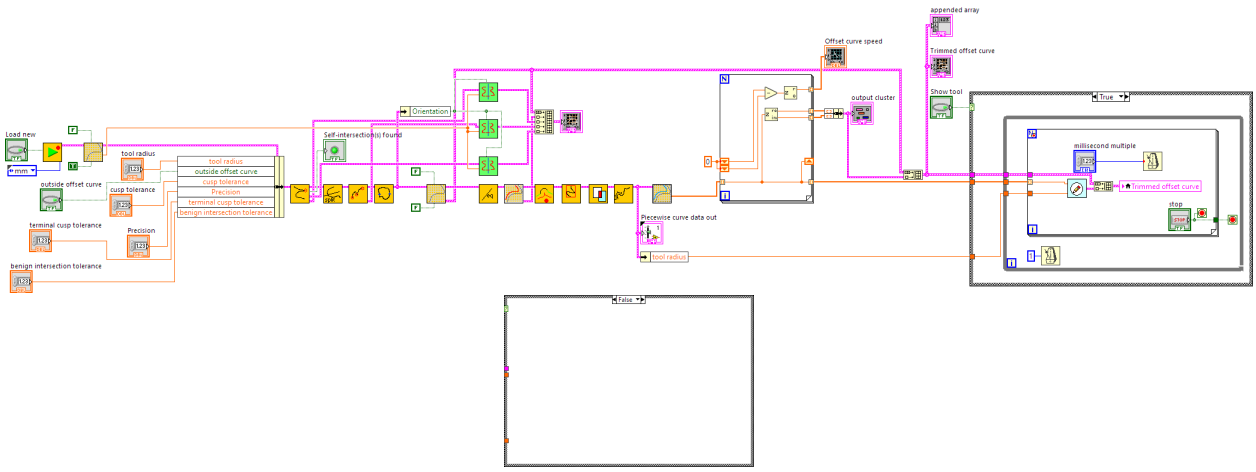
(e) Resolve critical sub-curve issues

7.1 Test program (Version 1.0)

Comments:

1. First load and convert the .svg file, while verifying text integrity
2. Render the piecewise curve in order to determine the curve barycenter
3. Search for self-intersections and cusps
4. Split sub-curves at cusps
5. Detect end-point singularities
6. Determine curve orientation (translate curve to barycenter and check the sense of rotation of the curve points; mirror the curve, if the rotation is negative → **don't forget to mirror the resulting offset curve points.**
7. Render the (mirrored) piecewise curve
8. Find the line equations for the line sub-curves
9. Render the untrimmed offset curve
10. Add circumvention curves at detected end-point singularities
11. Find the sub-curve bounding boxes
12. Yield the critical sub-curves for each sub-curve (distances of bonding boxes are smaller than tool diameter)
13. Find the sub-curve curve lengths $L[i]$ in order to determine the optimal curve step $dt[i] = L[i]/precision$
14. Detect tool circle intersections with sub-curves and store valid offset curve points
15. Draw graph of Offset curve speed in order to detect remaining disconitinuities.
16. Draw tool circle





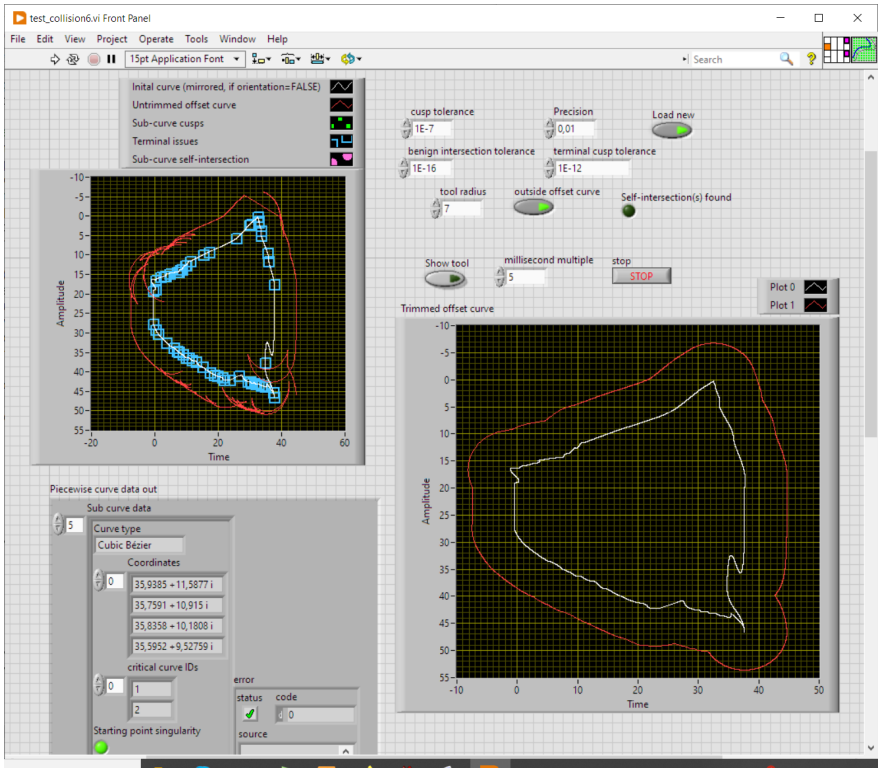
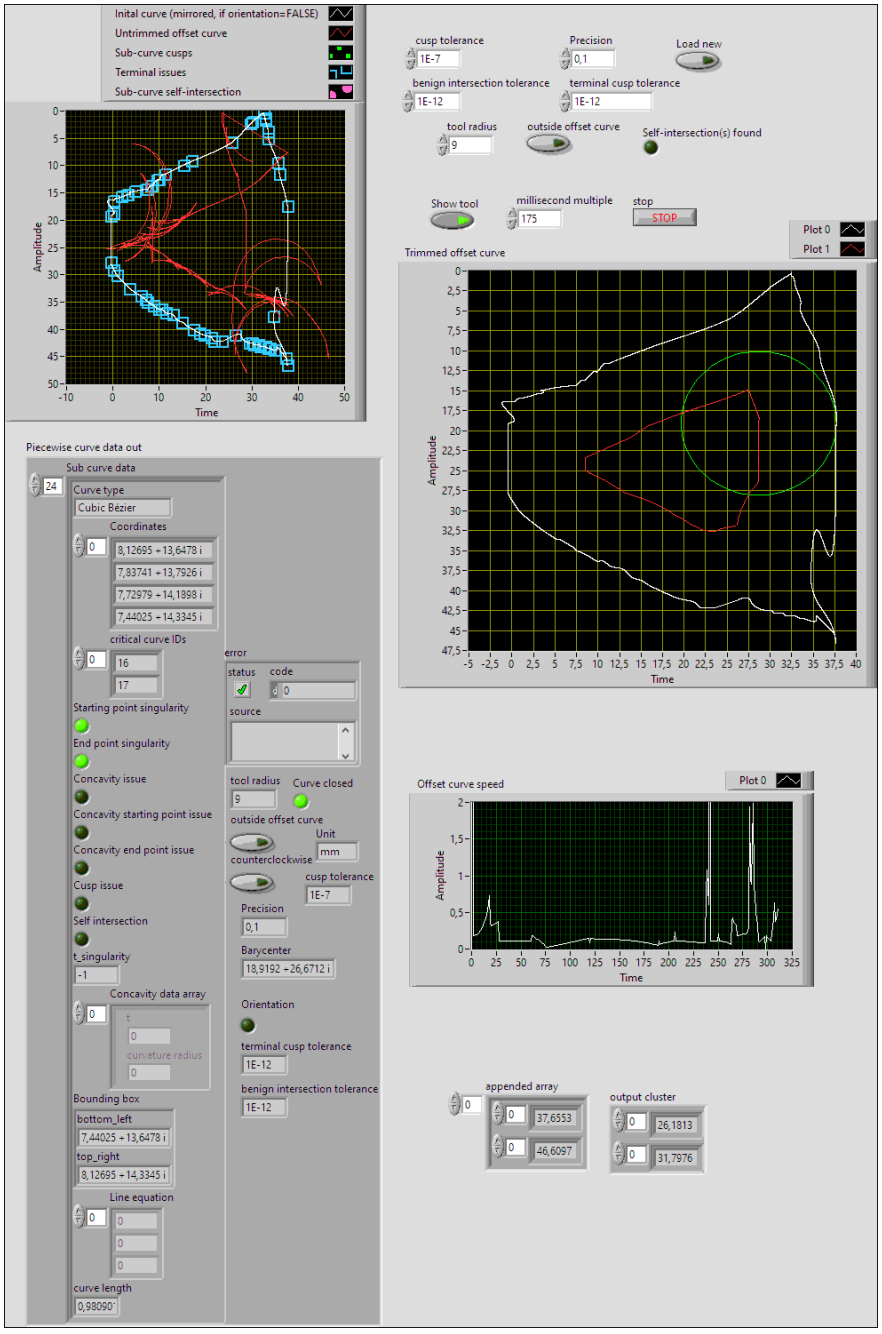


Figure 49: This resulting offset curve demonstrates that the program correctly adds the circumvention arcs about cusps.



7.2 G-Code generation

We use the previous offset curve generator in three different, but similar programs for **inside/outside**, **pocket** and **laser** contour paths, which require particular treatments. For instance, pocketing represents inside paths with growing tool radii. In the case of inside curves, it must be paid attention at large **jumps** that are produced by the offset path generator in the case of concavities with curvature radius greater than the tool radius. These jumps must be translated into valid G-Code tool retraction, moving action and tool re-plunging. Otherwise the workpiece could be damaged during the milling process.

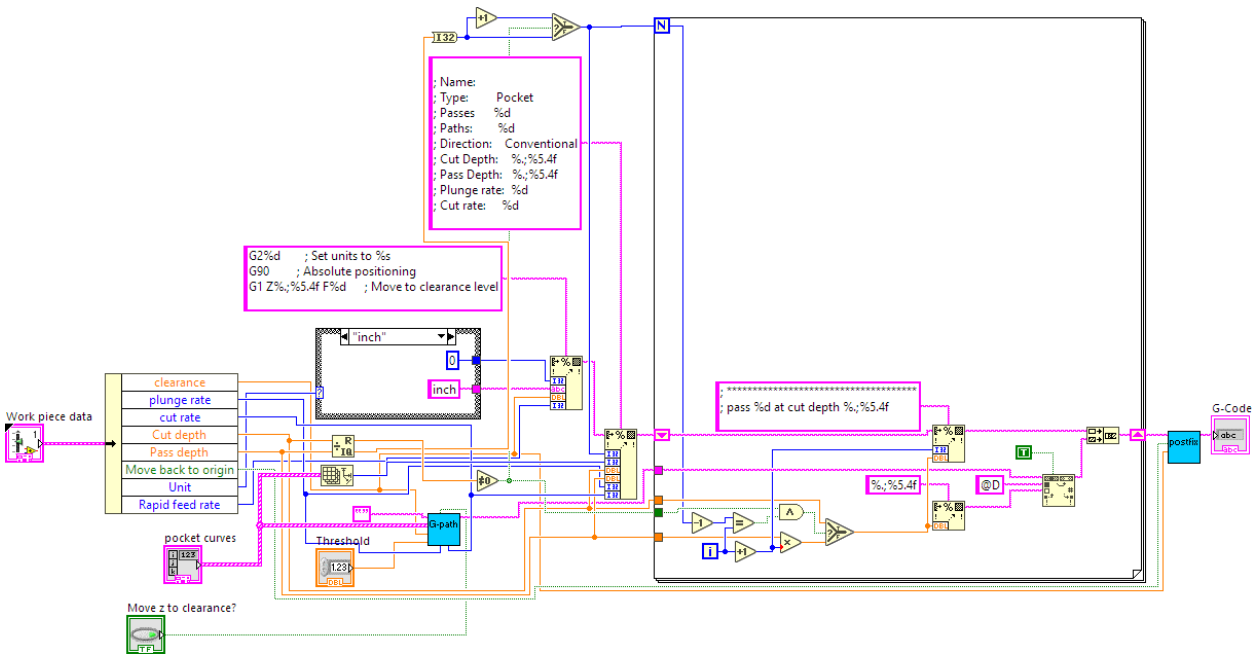
While generating outside G-Code paths, we must have the facility that the tool can jump over tabs, which are small work-piece bridges that should prevent the cut work-piece from loosening.

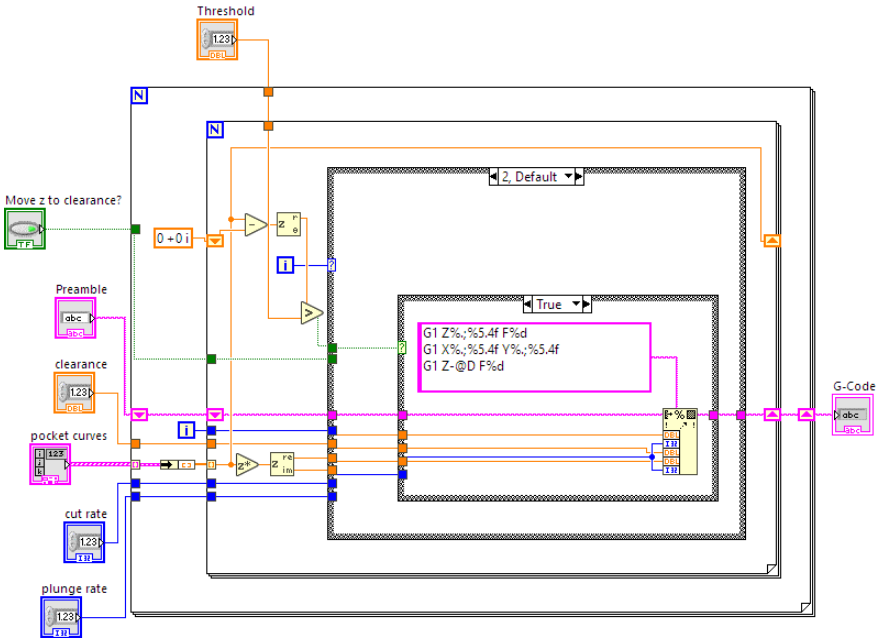
Software also must provide the path code for different plunge depths in the case of multipass milling.

Engraving can be done with choosing tool diameter = 0.

Laser mode must ignore retraction and plunging of the z-axis and replace any jump by switch-off/on of the laser tool.

We reproduce here the pocket G-Code generator diagram.





8 Results

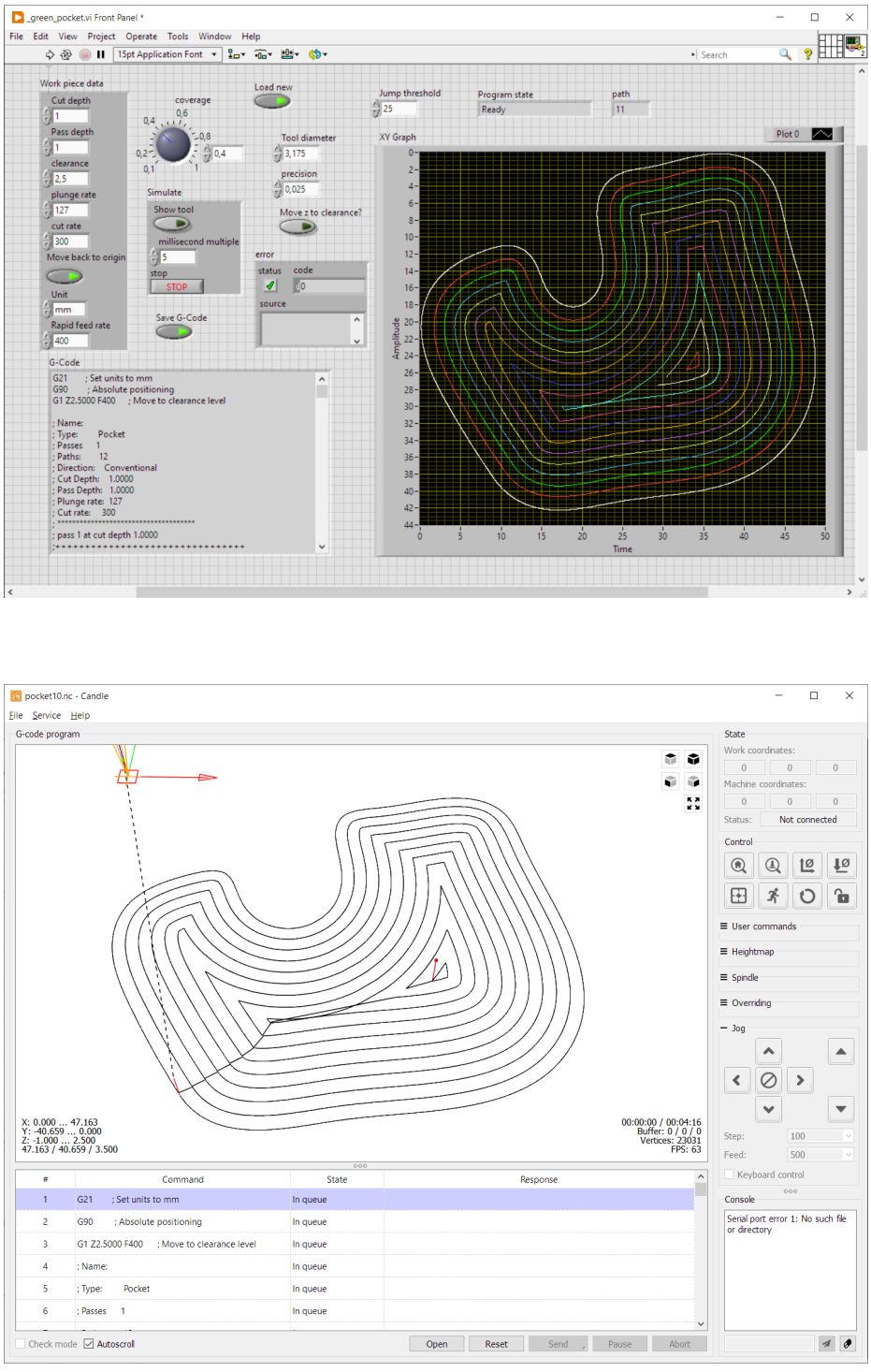
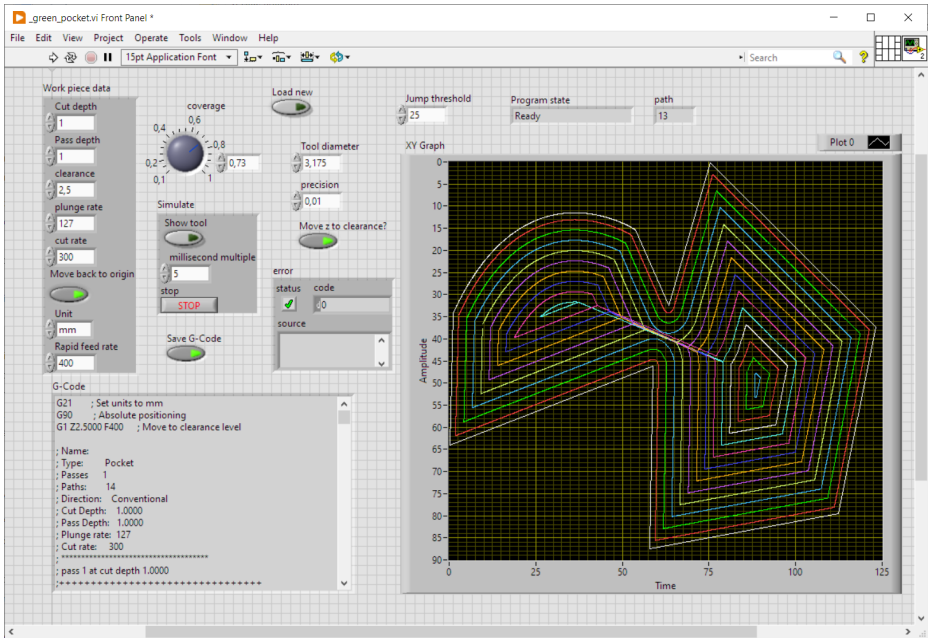


Figure 50: Pocketing without *jumps*. Visibly the generated G-Code is correctly understood by the freeware GRBL-milling CANDLE program.



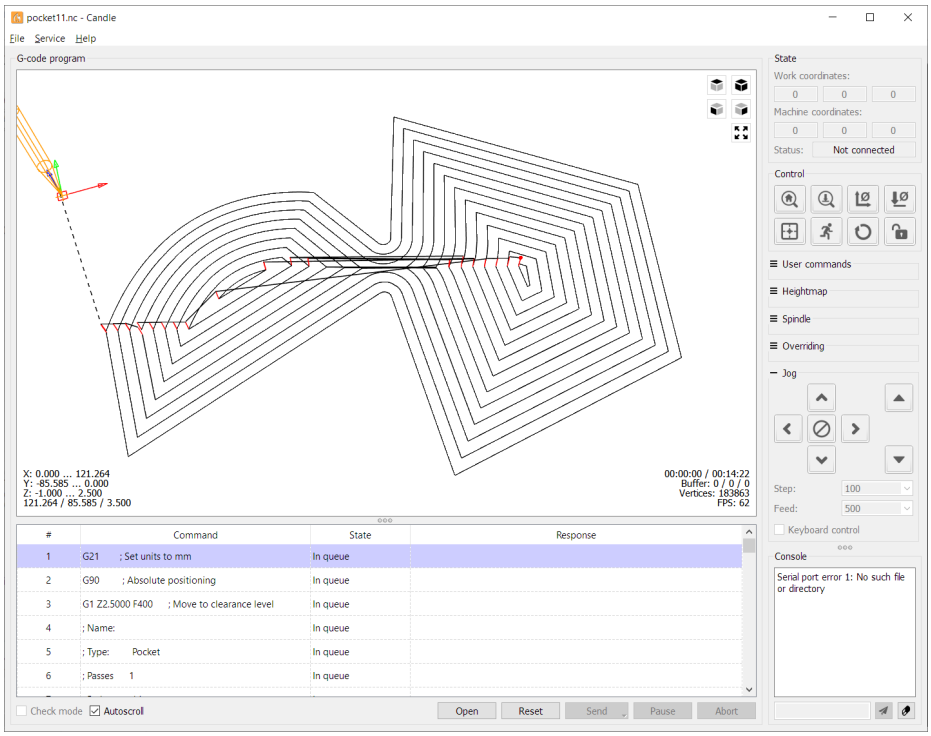
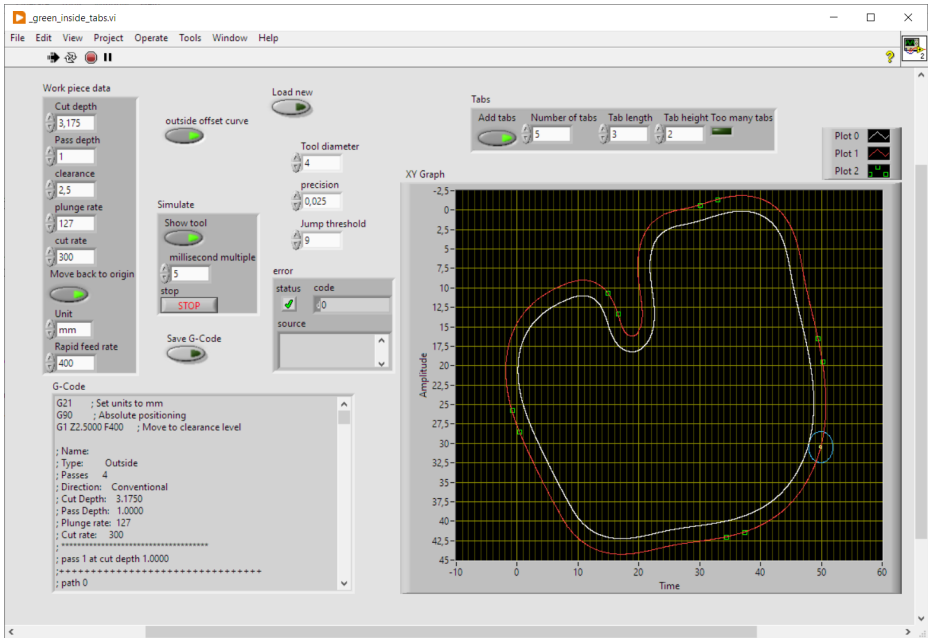


Figure 51: Pocketing with *jumps*.



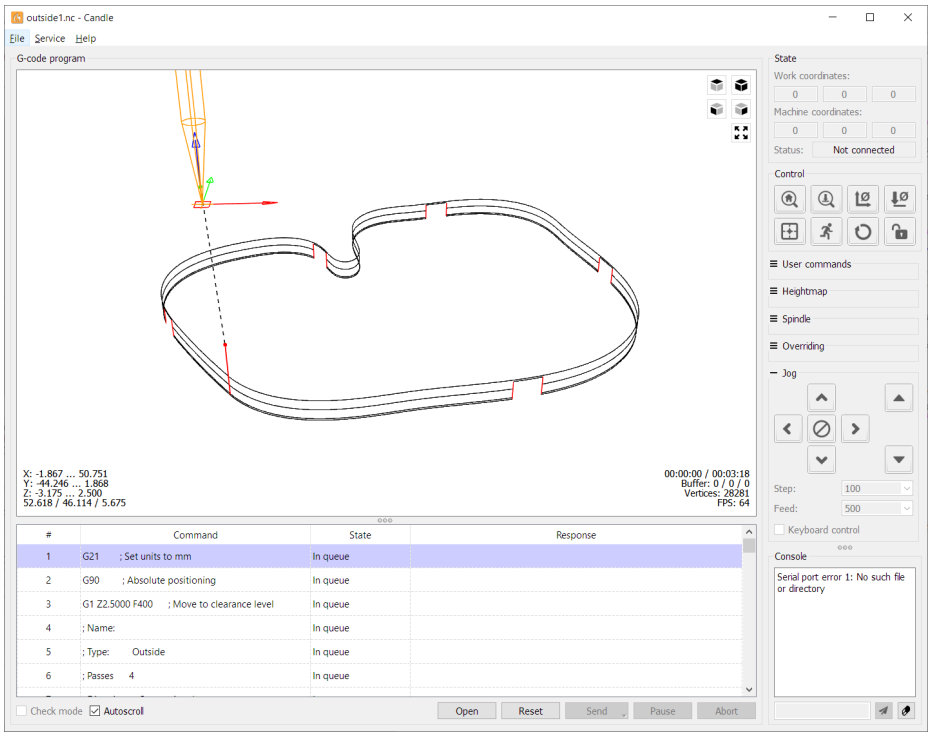


Figure 52: Outside contouring with tabs.

9 Final word

For those readers interested in the LabVIEW code for personal use, please ask at claude.baumann@education.lu.