curvepy: Bézier Curves, Delaunay Triangulations and Voronoi Diagrams

NumerikGang
Lars Quentin, Maximilian Winkler, Timo Specht

July 11, 2022

Contents

1 Introduction

1.1 Motivation

Computer Aided Geometrical Design, or CAGD, is a branch of numerical mathematics which has a variety of practical applications.
Bézier curves are generally utilized to model smooth curves, especially quadratic or cubic ones. They are defined by control polygons and manipulating these polygons results in a change in the original curve. Thus it is quite simple to adjust Bézier curves, which is why they are widely used in computer design, animations, font design, and the car industry. Indeed, Piere Bézier originally invented them to design curves for the bodywork of Renault cars [Far01].
Constraint Delaunay triangulations are applied in path planning in automated driving and topographic surveying [AKI12] and Voronoi diagrams, which are closely related to Delaunay triangulation, are used in fields like biology to model cell [Boc+09] or bone structures [Li+12]. More on that later.

1.2 Scope of this Project

The main goal of this project was to implement the various algorithms from chapter 3, 4, and 5 of Gerald Farin's book "Curves and Surfaces for CAGD, a pracitcal guide" using Python. That means each mathematical result and formula is taken from this book unless stated differently.
The content of chapter 3 focuses on more fundamental topics like linear interpolation or blossoms and served as a great introduction. The most challenging part of not only this chapter but also the entire project was the implementation of an algorithm for the Delaunay triangulation and also the Voronoi diagram.
Chapter 4 serves as an introduction to Bézier curves and describes the de Casteljau algorithm. Furthermore it describes properties of Bézier curves and the technique of blossoming.
Chapter 5 expands on these topics as it proposes more variations on how one can compute Bézier curves, for instance subdivision and Bernstein polynomials. Moreover different forms of Bézier curves are introduced, namely the monomial and matrix form from which we only focused on the monomial form. At last more properties like the derivative of Bézier curves are established. We decided to not use the matrix form because of its bad conditioning.
Since the book didn't contain any definitions on Voronoi diagrams and Delaunay triangulations, this scope extended. We looked at the current literature,

evaluated different algorithmic concepts and implemented a [GS78] based algorithm.

The project structure can be seen in Figure 1.

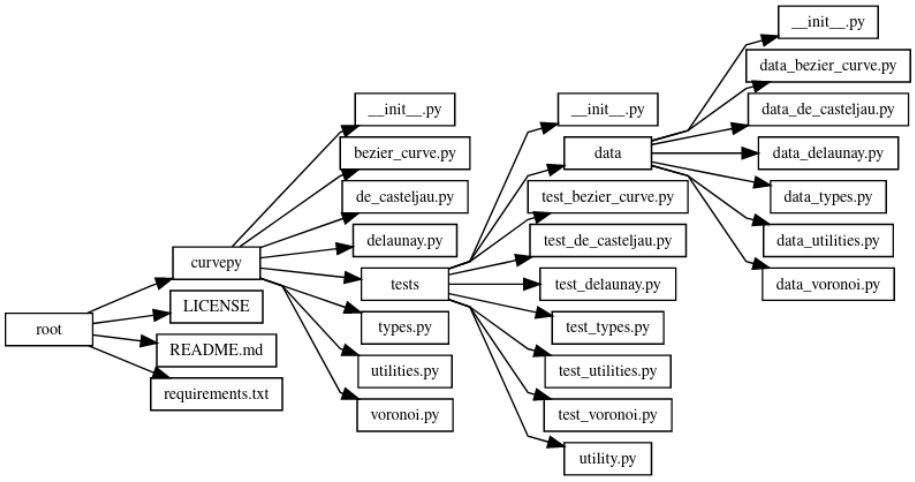


Figure 1: Project Structure

Let us briefly go over the most important architectural design decisions:

`requirements.txt` To keep overhead as minimal as possible we decided to track dependencies via `requirements.txt` instead of using a more sophisticated build system like `poetry`.

`curvepy` All Python code is contained within the `curvepy` top level module, even the tests. This is essential; otherwise any deployment tool (like `setuptools`) would declare `tests` as a top level module as well, thus overshadowing any other test module installed within the python environment. Each python file has a has a matching test and data file.

`tests` Our 1000+ test cases, mostly against precomputed and manually verified values contained in `data`.

`__init__.py` Although most `__init__.py` files contain no code, they have two specific usages: On the one hand, they contain docstrings which are processed by `pdoc3` for HTML-documentation. On the other hand, they are required by Python so that the folder is recognized as a module.

`LICENSE` To allow the least restrictive usage of our algorithms, while still keeping ownership of it's interlectual property (unlike public domain licenses like CC-0), we provide all our code as MIT licensed.

`README.md` To provide a simple overview, our library contains a `README.md`, which can be rendered by Gitlab.

The python files are organized as follows:

`bezier_curve.py` contains multiple Bézier curve implementations. They all have the same methods; this is done via abstract class inheritance.

`de_casteljau.py` contains an implementation of the De Casteljau algorithm for computing Bézier curves as well as some utility methods.

`delaunay.py` contains an implementation of a Delaunay triangulation algorithm. It is also used to compute it's dual graph, the Voronoi diagram.

`types.py/utilities.py` contain various helper methods and classes.

`voronoi.py` computes the voronoi diagram through it's underlying Delaunay triangulation.

1.3 Technical Decisions

Here are some of our technical decisions with their reasoning:

- Since we see ourselves as an mostly educational implementation, we formally only support Python 3.8+, although we use `__future__` imports to keep as much backwards compatibility as possible.
  We do not require any complicated version testing tool like `tox` or building tool like `poetry`.

- Our code style is mostly PEP-8 conform, but with a few exceptions. Here are the according `flake8`-IDs:

  E731 *"Do not assign a lambda expression, use a def"*
  For single usage methods, we sometimes want to assign a variable name to offer more semantics.

  E125 *"Continuation line with same indent as next logical line"*
  This is simply not compatible with PyCharm, which we use as our formatter.

  - For easier readability, we allow line lengths with up to 120 characters.

- For docstrings, we use the `numpy` docstring format, as we heavily depend on `numpy` and `scipy`.
- For docstring-based HTML documentation we use `pdoc3`, since it covers all our use cases while being simpler than sphinx, which requires additional reStructuredText files.
- We use `pytest` for unit tests because it is the most commonly used mature python unit testing framework.
- For formatting, we use Jetbrains default formatter, which is shipped with PyCharm.
  Additionally, we use `isort` for import ordering.
- Once we publish our library on PyPI, we will use version numbers according to PEP 440.
  Note that this is *not* compatible with Semantic Versioning (SemVer).
  Further note that this prohibits git-hashes as version numbers since they have no formal ordering.

We also use the continuous integration (CI) provided by Gitlab. Specifically, we use the default Gitlab runner provided by the GWDG. Our pipeline is split into 3 stages:

`static-analysis` We use `flake8` as our linter. `flake8` is a combination of `pep8`, which checks conformity against the PEP 8 design standard, as well as `pyflakes`, which provides a substantial amount of plugin functionality.
We use the following extensions:

  - `flake8-bugbear` "A plugin for Flake8 finding likely bugs and design problems in your program."
  - `pep8-naming` "Check your code against PEP 8 naming conventions."
  - `flake8-builtins` "Check for python builtins being used as variables or parameters."
  - `flake8-comprehensions` "A flake8 plugin that helps you write better list/set/dict comprehensions."

`test` On every commit to `master` as well as merge requests we run our full `pytest` based test suite.

`deploy` After every commit to `master`, we automatically update and redeploy our HTML based documentation [Tim].

1.4 Acknowledgements

Firstly, we want to thank Dr. Jochen Schulz and Prof. Dr. Gerlind Plonka-Hoch for their monumental patience, waiting around 2 years for this project to complete.
Secondly, we want to thank Gerald E. Farin for his detailed overview of computer aided graphic design [Far01] and Franz Aurenhammer and Steven Fortune for their great meta analysis of Voronoi and Delaunay research [Aur91] [FOR95].
Lastly, we want to thank the now defunct "Coffeebar ins Grüne" for all the great Currywursts which were soaked in a comically large amount of oil that would make any government jealous.

2 Bézier Curves

First we will go over some preliminaries that are useful to know before we get to actual Bézier curves. Then we introduce the general concept of Bézier curves and give an overview how we implemented the different methods to calculate Bézier curves followed by detailed sections about these methods and their implementation.

2.1 Linear Interpolation

Definition 1. *(Straight Line) Let $a, b \in \mathbb{R}^2$ be 2 points. A **straight line** is defined as the set of all possible combinations*

$$x = x(t) = (1 - t) \cdot a + t \cdot b$$

where $t \in [0, 1]$.

But we can also express t as $t = (1-t) \cdot 0 + t \cdot 1$. Hence t is also just a point lying on a straight line between 0 and 1 and therefore the points a, x, b are an **affine map** on $0, t, 1$.

Definition 2. *(Linear Interpoaltion) **Linear interpolation** is an affine map of the real line onto a straight line.*

Since we see linear interpolation as an affine map we get the following property.

Definition 3. *Linear interpolation is **affine invariant**. Thus the following holds*

$$\phi(x) = \phi((1 - t) \cdot a + t \cdot b) = (1 - t) \cdot \phi(a) + t \cdot \phi(b)$$

where ϕ is an affine map, $t \in [0, 1]$ and a, b are two points in $\mathbb{R}^2$.

2.2 Piecewise Linear Interpolation

Definition 4. *(Polygon) A **Polygon** is defined as a finite sequence of points $b_0, \ldots, b_n \in \mathbb{R}^m$ where each two consecutive points b_i and b_{i+1} are connected by a straight line.*

With this we can define the piecewise linear interpolant of a curve c.

Definition 5. *(piecewise linear interpolant) Let c be a curve. The polygon interpolating the b_i is the* ***piecewise linear interpolant*** *PL of c if and only if all points lie on c.*

Remark 2.1. *One can easily show that the piecewise linear interpolant is* ***affinely invariant****, which means that for any curve c and affine map ϕ*

$$PL(\phi(c)) = \phi(PL(c))$$

2.3 Menelaos' Theorem

Theorem 1. *Let b be an open polygonal chain of $b_0, b_1, b_2 \in \mathbb{R}^2$. We can define any points on the straight lines as convex combinations of those points.*

$$\begin{aligned} b[0,t] &= (1-t)\cdot b_0 + t \cdot b_1 \\ b[s,0] &= (1-s)\cdot b_0 + s \cdot b_1 \\ b[t,1] &= (1-t)\cdot b_1 + t \cdot b_2 \\ b[s,1] &= (1-s)\cdot b_1 + s \cdot b_2 \end{aligned}$$

Where $s,t \in [0,1]$ and $b[*,*]$ is a two-dimensional blossom (we will talk more about blossoms in the next section). With this we define $b[s,t],\ b[t,s]$ as:

$$\begin{aligned} b[s,t] &= (1-t)\cdot b[s,0] + t \cdot b[s,1] \\ b[t,s] &= (1-s)\cdot b[0,t] + s \cdot b[t,1] \end{aligned}$$

Menelaos' theorem now simply states

$$b[s,t] = b[t,s]$$

Proof.

$$\begin{aligned}
b[s,t] &= (1-t)\cdot b[s,0] + t\cdot b[s,1] \\
&= (1-t)\cdot((1-s)\cdot b_0 + s\cdot b_1) + t\cdot((1-s)\cdot b_1 + s\cdot b_2) \\
&= (1-t)\cdot(1-s)\cdot b_0 + (1-t)\cdot s\cdot b_1 + t\cdot(1-s)\cdot b_1 + t\cdot s\cdot b_2 \\
&= (1-t)\cdot(1-s)\cdot b_0 + t\cdot(1-s)\cdot b_1 + (1-t)\cdot s\cdot b_1 + t\cdot s\cdot b_2 \\
&= (1-s)\cdot((1-t)\cdot b_0 + t\cdot b_1) + s\cdot((1-t)\cdot b_1 + t\cdot b_2) \\
&= (1-s)\cdot b[0,t] + s\cdot b[t,1] \\
&= b[t,s]
\end{aligned}$$

□

This means that the resulting point is continuous linear interpolation is invariant to the order of weights used.

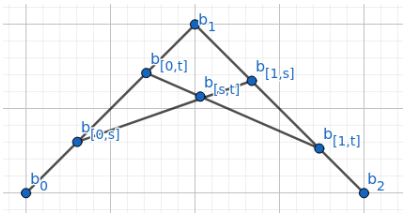


Figure 2: Points defined on the straight line of our previously defined points

2.4 Blossoming

Definition 6. *(Blossoms)* ***Blossoms*** *are a multivariate function. Denoted* $b[t_1,\ldots,t_n]$*, where* $t_i \in \mathbb{R}$ *for* $i \in \{1,\ldots,n\}$*.*

So in each interpolation step a different t can be used. Blossoms have some very interesting properties that are useful for types of Bézier curves:

Lemma 1. ***(Symmetry)*** *In the case of Blossoms the order of the arguments does not change anything; in fact this is an application of theorem 1:*

$$b[t_1,\ldots,t_n] = b[\pi(t_1,\ldots,t_n)]$$

where $\pi(t_1,\ldots,t_n)$ *denotes some permutation matrix and* $t_i \in \mathbb{R}$ *for* $i \in \{1,\ldots,n\}$*.*

Before we can look at the next property let us fist define **Multi Affinity** and **barycentric combinations**. First we will define multi affinity.

Definition 7. *(Multi Affinity) A function is* ***multi affine*** *if it is affine in regards to every parameter.*

Now let us introduce barycentric combinations.

Definition 8. *Let $P_1, \ldots, P_n$ be points in $\mathbb{R}^m$ and $\alpha_1, \ldots, \alpha_n$ be weights in $\mathbb{R}$. A* ***barycentric combination*** *of the points $P_1, \ldots P_n$ is a weighted sum*

$$\sum_{i=1}^{n} \alpha_i \cdot P_i$$

with the affinity constraint

$$\sum_{i=1}^{n} \alpha_i = 1$$

Before we look at the general case, let us examine a simple example:

$$b[\alpha \cdot r + \beta \cdot s, t_2] = \alpha \cdot b[r, t_2] + \beta \cdot b[s, t_2]$$

So if the first argument is a barycentric combination, we can compute the blossom values separately and then build the barycentric combination. Of course we are not restricted to the case for the first argument, since we have symmetry. In general we have:

Lemma 2. ***(Multi Affinity)***

$$b[\alpha \cdot r + \beta \cdot s, *] = \alpha \cdot b[r, *] + \beta \cdot b[s, *]$$

where $\alpha + \beta = 1$, $r, s \in \mathbb{R}$ and $*$ denotes the same arguments. However the order of the parameters does not play a role, as shown in lemma 1.

Remark 2.2. ***(Diagonality)*** *Let $t = t_1 = \cdots = t_n$ and $t_i \in \mathbb{R}$ for $i \in \{1, \ldots, n\}$. In this case we obtain a polynomial curve or Bézier curve. We use the notation:*

$$b[t, \ldots, t] = b[t^n]$$

if the argument t is repeated n times.

2.4.1 Leibniz Formula

If we use the properties we just defined we can consider a special case where we have $(\alpha \cdot r + \beta \cdot s)$ n times as argument. We get:

Lemma 3. ***(Leibniz Formula)***

$$b[(\alpha \cdot r + \beta \cdot s)^n] = \sum_{i=0}^{n} \binom{n}{i} \alpha^i \cdot \beta^{n-i} \cdot b[r^i, s^{n-i}]$$

2.5 Definition of Bézier Curves

Before we will have a look at different types of Bézier Curves and how they can be calculated. Let us first define what a Bézier Curve is.

Definition 9. *A **Bézier curve** is a parametric curve described by control points $b_0, \ldots, b_n \in \mathbb{R}^2$. These points form the so called Bézier polygon or control polygon the b_i are called **Bézier points** or **control points**, and we use these terms interchangeably. The degree of the curve described by the polygons is $n - 1$.*

The actual Bézier curve can be formed using a variety of different formulas that all use some weight t, which is normally on the unit interval, on the Bézier points. So we write $b_0^n(t)$ for an actual point on the curve and $b_i^0(t) := b_i$ for the Bézier points, similar to the notion we already used in the blossom section.

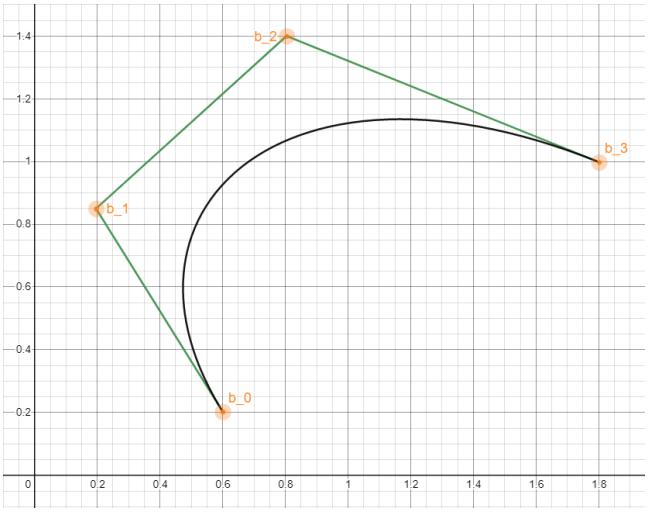


Figure 3: Example Bézier curve with control polygon

2.6 General Implementation of Types of Bézier Curves

To implement the different types of Bézier curves we use a clever class hierarchy.

We use an abstract class as our base. This one implements most of the methods all of its children use. Like the derivative, addition, multiplication and so on.

The most fundamental function is the curve function, which is a cached property. Calling this method generates the preferred amount of evaluated points for the given Bézier points. The curve function supports either parallel or serial execution.

The curve is based on the abstract `init_func`, which is the only method that needs to be provided by any subclass. This makes inheritance very elegant.

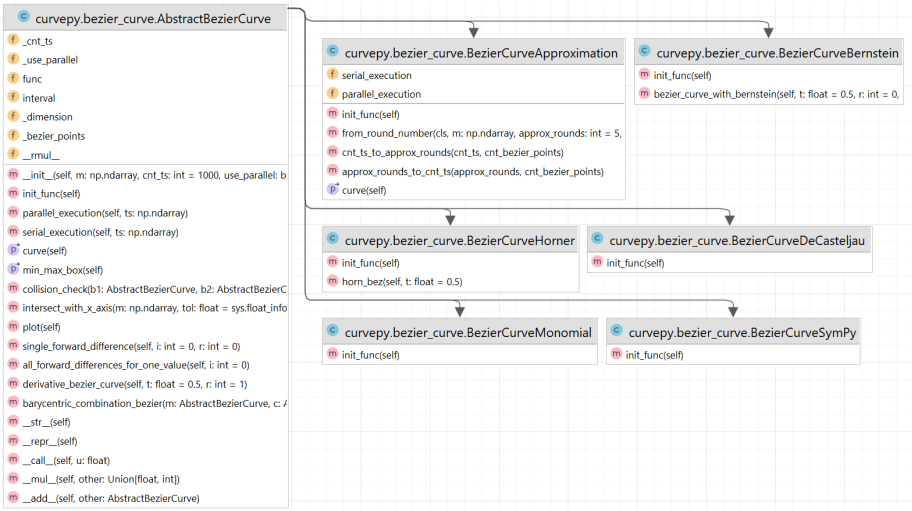


Figure 4: UML class diagram of all Bézier curves

2.7 De-Casteljau Algorithm

In this section we will first introduce the De-Casteljau algorithm which can calculate points lying on a Bézier curve. Following that we explain some interesting properties of Bézier curves and then present our different implementations.

Definition 10. *Let $b_0, \ldots, b_n \in \mathbb{R}^2$ and $t \in \mathbb{R}$, then the following formula is used for the **De-Casteljau** algorithm:*

$$b_i^r(t) = (1-t) \cdot b_i^{r-1}(t) + t \cdot b_{i+1}^{r-1}(t)$$

for $r = 1, \ldots, n$ and $i = 0, \ldots, n-r$.

2.7.1 Endpoint Interpolation

If t is set to 0 we get b_0 (i.e. the beginning) and if it is set to 1 we get b_n (i.e. the end), which allows for easier usage, especially in design-oriented problems. This is also very handy for composing Bézier curves.

2.7.2 Invariance under Affine Maps

Bézier curves are invariant under affine maps, which follows from the De-Casteljau algorithm, since it uses repeated linear interpolation, which is invariant under affine maps. This property is quite handy. For instance if we have four control points and want to evaluate 100 points on the curve described by them and additionally we want to rotate these points, we can just rotate the control points and then compute the 100 points instead of first computing all 100 points and then rotating everyone of them.

2.7.3 Parameter Transformation

Another interesting property is that the curve is blind to the actual interval, that the curve is defined over. This is possible, because the algorithm works on ratios. Furthermore we can map a parameter u from an arbitrarily chosen interval $[a, b]$ on our unit interval $[0, 1]$ by applying the following transformation:

$$t = \frac{u-1}{b-a}$$

2.7.4 Blossoms and De-Casteljau

The concept of blossoming can also be used in the De-Casteljau algorithm. One can use a new parameter value $t \in [0, 1]$ in each step of the algorithm. Looking at the cubic case we obtain:

$$\begin{array}{llll} b_0 & & & \\ b_1 & b_0^1[t_1] & & \\ b_2 & b_1^1[t_1] & b_0^2[t_1, t_2] & \\ b_3 & b_2^1[t_1] & b_1^2[t_1, t_2] & b_0^3[t_1, t_2, t_3] \end{array}$$

This traces out a region because we use in each of our r steps a different value t_r, to recover the original point on the curve we need to set $t = t_1 = t_2 = t_3$ and compute again.

2.7.5 Implementation

We divided the De-Casteljau algorithm in three methods:

1 `de_casteljau_one_step`. This method computes one step of the De-Casteljau algorithm using the following formula:

$$b_i^r(t) = (1 - t) \cdot b_i^{r-1}(t) + t \cdot b_{i+1}^{r-1}(t)$$

Because of numpy arrays the implementation is straightforward. We utilise slicing and element wise operations on numpy arrays to compute one step without the use of a loop. Additionally we perform most of the heavy work in C++, which makes our implementation significantly faster than only using Python.

2 `de_casteljau_n_steps`. This method calls (1) n times, where n lies between `0` and number of control points minus one and is given by the user. So one can calculate intermediate points or resume calculation from intermediate points.

3 `de_casteljau`: This method computes $b_0^n(t)$ by calling (2) with the proper value for n, which is the number of control points minus one.

One drawback is, that we have to explicitly copy the array in certain steps to avoid mistakes, since all python objects, including numpy arrays, are called by reference. We also implemented a parallel version of (3).

```python
def de_casteljau_one_step(m: np.ndarray, t: float = 0.5, interval: Tuple[int, int] = (0, 1),
                          make_copy: bool = True) -> np.ndarray:
    """Method computing only one step of the de Casteljau algorithm...."""
    if make_copy:
        m = m.copy()
    l, r = interval
    t2 = (t - l) / (r - l) if interval != (0, 1) else t
    t1 = 1 - t2

    m[:, :-1] = t1 * m[:, :-1] + t2 * m[:, 1:]

    return m[:, :-1] if m.shape != (2, 1) else m


def de_casteljau_n_steps(m: np.ndarray, t: float = 0.5, r: int = 1, interval: Tuple[int, int] = (0, 1)) -> np.ndarray:
    """..."""

    for _ in range(r):
        m = de_casteljau_one_step(m, t, interval, make_copy=False)
    return m


def de_casteljau(m: np.ndarray, t: float = 0.5, make_copy: bool = True,
                 interval: Tuple[int, int] = (0, 1)) -> np.ndarray:
    """Method computing n Iterations of de Castelljau, n is defined by the amount of given points...."""

    _, n = m.shape
    return de_casteljau_n_steps(m.copy() if make_copy else m, t, n, interval)
```

Figure 5: De-Casteljau Code

There is also a sympy version of the De-Casteljau algorithm, which handels t as a symbolic variable and computes the algorithm without knowing a specific value for t, hence we end up with a function representing our curve. At the end we use the lambdify routine to transform our symbolic function to a lambda function, so that we can calculate numerical values faster. Thus for every value of t we just evaluate our lambda function for this value. Sadly, this implementation is limited by the maximum recursion depth allowed by Python.

Our blossoming method calls `de_casteljau_one_step` for different values of t which are defined by the user in a given list.

2.8 Bernstein

Another way to calculate Bézier curves is to use Bernstein polynomials. We define them as follows.

Definition 11. *For $t \in \mathbb{R}$ we define:*

$$B_i^n(t) = \binom{n}{i} \cdot t^i \cdot (1-t)^{n-i}$$

*as the **Bernstein Polynomial** with parameter t, where*

$$\binom{n}{i} = \begin{cases} \frac{n!}{i! \cdot (n-i)!} & : 0 \leq i \leq n \\ 0 & : otherwise \end{cases}$$

There is also a recursive formula for Bernstein polynomials:

Definition 12. *The **Bernstein polynomial** with parameter t is defined as follows:*

$$B_i^n(t) = (1-t) \cdot B_i^{n-i}(t) + t \cdot B_{i-1}^{n-1}(t)$$

where $B_0^0(t) = 1$ and $B_j^n(t) = 0$ for $j \notin \{0, \ldots, n\}$.

We implemented both the recursive and the iterative variant.

Remark 2.3. *From the blossoming section we know that we can write Bézier curve as $b[t^n]$, moreover t can be expressed as $t = (1-t) \cdot 0 + t \cdot 1$. By applying lemma 3 we get*

$$b(t) = b[t^n] = \sum_{i=0}^{n} b_i \cdot B_i^n(t)$$

since $b_i = b[0^{n-i}, 1^i]$.

This results in the following remark:

Remark 2.4. *We can express the intermediate points b_i^r in the form of Bernstein polynomials of degree r:*

$$b_i^r = \sum_{j=0}^{r} b_{i+j} \cdot B_j^r(t)$$

This follows because of $b_i^r(t) = b[0^{n-r-i}, t^r, 1^i]$ and lemma 3.

With this at our hands we get:

Theorem 2. *Given control points $b_0, \ldots, b_n \in \mathbb{R}^2$ and a parameter $t \in [0, 1]$. Further let $b_i^r, i \in \{1, \ldots, n\}, r < n$ be precomputed. Then we can extend our b^r values as follows:*

$$b^n(t) = \sum_{i=0}^{n-r} b_i^r(t) \cdot B_i^{n-r}(t)$$

This means that r semantically describes the highest degree of the previously computed Bézier polygon. Thus $r = 0$ if and only if no precomputation was made beforehand.

Additionally we have the property of invariance under barycentric combinations:

$$\sum_{i=0}^{n} (\alpha \cdot b_i + \beta \cdot c_i) \cdot B_i^n(t) = \sum_{i=0}^{n} \alpha \cdot b_i \cdot B_i^n(t) + \sum_{i=0}^{n} \beta \cdot c_i \cdot B_i^n(t)$$

Therefore we implemented two magic methods namely the `__add__` and `__mul__` function. `__add__` just adds up the given Bézier points of both curves and `__mul__` multiplies all control points by a scalar value.

```
def bernstein_polynomial(n: int, i: int, t: float = 1) -> float:
    """Method using 5.1 to calculate a bernstein polynomial..."""
    return scs.binom(n, i) * (t ** i) * ((1 - t) ** (n - i))
```

Figure 6: Bernstein Polynomial Code

```
def bezier_curve_with_bernstein(self, t: float = 0.5, r: int = 0,
                                interval: Tuple[float, float] = (0, 1)) -> np.ndarray:
    """Method using teh equation below to calculate a point with given bezier points...."""
    m = self._bezier_points
    _, n = m.shape
    t = (t - interval[0]) / (interval[1] - interval[0])
    return np.sum([m[:, i] * bernstein_polynomial(n - r - 1, i, t) for i in range(n - r)], axis=0)
```

Figure 7: Bézier with Bernstein Code

2.9 Horner Scheme

We can also develop an Horner Scheme for the Bézier curves. First we will look at a cubic example.

Example 2.1.

$$c_0 + t \cdot c_1 + t^2 \cdot c_2 + t^3 \cdot c_3 = c_0 + t \cdot (c_1 + t \cdot (c_2 + t \cdot c_3))$$

This is the case for a normal polygon on the monomial base, where $c_0, \ldots, c_4 \in \mathbb{R}$ *and* $t \in \mathbb{R}$*. We can transform this to Bézier curves and get*

$$b^3(t) = \left(\left(\binom{3}{0} \cdot s \cdot b_0 + \binom{3}{1} \cdot t \cdot b_1 \right) \cdot s + \binom{3}{2} \cdot t^2 \cdot b_2 \right) \cdot s + \binom{3}{3} \cdot t^3 \cdot b_3$$

where $s = (1 - t)$*, with* $t \in \mathbb{R}$ *and* $b_0, \ldots, b_3 \in \mathbb{R}^2$*.*

Now we can generalize this in the following way:

Definition 13. *Given* $n \in \mathbb{N}$ *points* $b_0, \ldots, b_n \in \mathbb{R}^2$ *and a parameter* $t \in \mathbb{R}$ *we can describe* $b^n(t)$ *in the following way:*

$$b^n(t) = \left(\ldots \left(\left(\left(\binom{n}{0} \cdot s \cdot b_0 + \binom{n}{1} \cdot t \cdot b_1 \right) \cdot s + \binom{n}{2} \cdot t^2 \cdot b_2 \right) \cdot s + \binom{n}{3} \cdot t^3 \cdot b_3 \right) \ldots \right) \cdot s + \binom{n}{n} \cdot t^n \cdot b_n$$

where $s = (1 - t)$*.*

This method is faster than the normal De-Casteljau algorithm, since the Horner scheme is of order $\mathcal{O}(n)$ while De-Casteljau is of order $\mathcal{O}(n^2)$. Additionally we do not need to save the control polygon in auxiliary array which saves space and further boosts the performance.

```python
def horn_bez(self, t: float = 0.5) -> np.ndarray:
    """Method using equation below to calculate point on curve given some t...."""
    m = self._bezier_points
    n = m.shape[1] - 1  # need degree of curve (n points means degree = n-1)
    res = m[:, 0] * (1 - t)
    for i in range(1, n):
        res = (res + t ** i * scs.binom(n, i) * m[:, i]) * (1 - t)

    res += t ** n * m[:, n]

    return res
```

Figure 8: Bézier with Horner Scheme Code

Note that `m` is just a reference to the control polygon for simplicity's sake.

2.10 Monomial Form of Bézier Curve

Before we start with the monomial form we need to know how to calculate the derivatives of Bézier curves.

Definition 14. *(Forward Differences and Iterated Forward Differences) Given some points $b_0, \ldots, b_n$ over $\mathbb{R}^2$. The* ***forward difference*** *is defined as follows:*

$$\Delta b_j = b_{j+1} - b_j$$

We can go one step further and define the ***iterative forward differences****:*

$$\Delta^r b_i = \sum_{j=0}^{r} \binom{n}{j} \cdot (-1)^{r-j} \cdot b_{i+j}$$

These will become handy when we inspect the derivatives and the monomial form.

2.10.1 Derivative of Bézier Curves

Before we will have a look at the general derivative, let us first examine the first derivative.

Theorem 3. *For the first **derivative** we first compute one step of De-Casteljau with $t = 1$ and then with an arbitrary $t \in [0, 1]$ the next $n - 1$ steps.*

$$b'(t) = n \cdot \sum_{j=0}^{n-1} \Delta b_j B_j^{n-1}(t)$$

Alternatively we can perform $n - 1$ steps of De-Casteljau with respect to some $t \in [0, 1]$ and then exactly one step with $t = 1$.

$$b'(t) = n \cdot (b_1^{n-1}(t) - b_0^{n-1}(t))$$

This leads to an interesting property.

Remark 2.5. *The derivative of a Bézier curve is just another Bézier curve, hence the derivative is a byproduct of the De-Casteljau algorithm. Though the De-Casteljau algorithm is not the fastest way of computing a Bézier curve, it is useful whenever we also need the first derivative.*

With the help of definition 14 we get a formula for the r'th derivative.

Theorem 4. *For a Bézier curve described by $b_0, \ldots, b_n \in \mathbb{R}^2$ we get that the rth derivative is described by*

$$\frac{d^r}{dt^r} b^n(t) = \frac{n!}{(n-r)!} \cdot \sum_{j=0}^{n-r} \Delta^r b_j \cdot B_j^{n-r}(t)$$

where $\Delta^r b_j$ is the iterated forward difference of b_j for $j \in \{1, \ldots, n\}$.

Since we can know easily compute the derivatives we can generate the Taylor series, which is our monomial form, since we are in the polynomial case.

$$x(t) = \sum_{j=0}^{n} \frac{1}{j!} \cdot x^{(j)}(0) \cdot t^j$$

When we use theorem 4 we get:

Theorem 5. *The monomial Form of a Bézier curve with the control points* $b_0, \dots b_n \in \mathbb{R}^2$ *is decribed by the formula*

$$b^n(t) = \sum_{j=0}^{n} \binom{n}{j} \cdot \Delta^j b_0 \cdot t^j$$

where $\Delta^j b_0$ *is the iterated forward difference of* b_0*.*

2.10.2 Implementation

We again use sympy to first calculate the polynom as a symbolic function. This is done by computing the coefficient $\binom{n}{j} \cdot \Delta^j b_0$ and multiplying it with the symbol t^j. Finally we use lambdify to fasten up the computation of the function. Hence we end up with a lambda function which takes a t as input and calculates the point. Alternatively one could use the presented Horner scheme by first constructing a list of the coefficients, that has to be done just once, and after that just call the method with the list and every value for t.

However this form is **numerical very unstable** and should be avoided whenever possible.

2.11 Subdivision

Let us turn our attention to a different approach of computing Bézier curves.

Definition 15. *(Subdivision) Given* $b_0, \dots, b_n \in \mathbb{R}^2$ *we run the De-Casteljau algorithm for some given parameter* $t \in \mathbb{R}$*. After that, we divide our set of points into two. First set is over the interval* $[0, t]$*, we will denote this points* $l_0, \dots, l_n \in \mathbb{R}^2$*. The second set is over the interval* $[t, 1]$*, we call these points* $r_0, \dots, r_n \in \mathbb{R}^2$*. We repeat this procedure for some given amount of* $k \in \mathbb{N}$ *rounds.*

So subdivision can be seen as an **approximation** of the curve described by our initial Bézier polygon since we get $2 \cdot (n+1)$ points per iteration, which can again be subdivided. Obviously, the more points we evaluate, the better the approximation becomes.

Remark 2.6. *After* $k \in \mathbb{N}$ *rounds of subdivision, one gets* 2^k *Bézier polygons with* $n + 1$ *points each.*

This can easily be seen as per round of subdivision we get two sets of points and we apply subdivision on them in the next step. This leads to a relatively fast approximation of our Bézier curve.

Example 2.2. *After* 10 *rounds of subdivision with* $b_0, \dots, b_5 \in \mathbb{R}^2$ *we get* 1024 *Bézier polygons each consisting of* 6 *points.*

Now we will have a look at which points one should pick for the left and right set of points.

Example 2.3. *If we look at the cubic example it is also easy to see which points to pick.*

$$
\begin{array}{cccc}
b_0 & & & \\
b_1 & b_0^1 & & \\
b_2 & b_1^1 & b_0^2 & \\
b_3 & b_2^1 & b_1^2 & b_0^3
\end{array}
$$

We just take the hypotenuse and the bottom line of the scheme. However the bottom line is taken in ***reversed*** *order and not from left to right.*

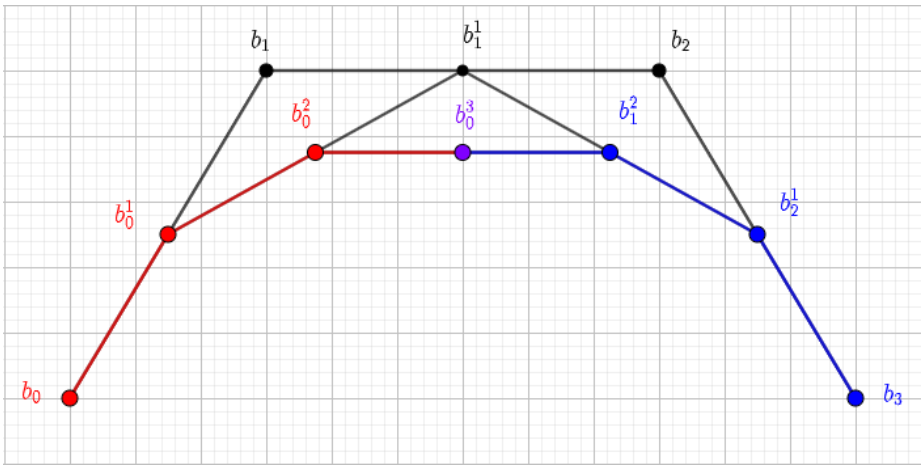


Figure 9: Subdivision Example with $t = 0,5$

In the end we get two new Bézier polygons, which both have four points like the one we had in the beginning, the left one in red with the points b_0, b_0^1, b_0^2 *and* b_0^3 *which is shared with the blue Bézier polygon on the right, consisting of* b_0^3, b_1^2, b_2^1 *and* b_3.

2.11.1 Implementation

We implemented the described procedure in the following way. The `current` variable represents one column of the triangular scheme we presented. By setting the *ith* element of `left` to the fist element of `current` in every step we always ensure that only elements lying on the hypotenuse are saved in `left`. The array `right` works in a quite similar manner but as we want the reverse order we begin at the end of the array and always look at the last element of current which is an element from the bottom line of the scheme. At last in every iteration we call `de_casteljau_one_step` which returns the next column saved in `current`.

```
def subdivision(m: np.ndarray, t: float = 0.5) -> Tuple[np.ndarray, np.ndarray]:
    """..."""
    left, right = np.zeros(m.shape), np.zeros(m.shape)
    current = m
    for i in range(m.shape[1]):
        left[::, i] = current.copy()[::, 0]
        right[::, -i - 1] = current.copy()[::, -1]
        current = de_casteljau_one_step(current, t, make_copy=True)

    return left, right
```

Figure 10: Subdivision Code

This implmenentation differs a bit; since we double the number of points each time, we have to compute the number of rounds needed for the expected accuracy. We use the following formula:

$$approx\_rounds = \left\lceil \log_2 \frac{|\text{points to be calculated}|}{|\text{bezier points}|} \right\rceil$$

```
@cached_property
def curve(self) -> Union[Tuple[List[float], List[float]], Tuple[List[float], List[float], List[float]]]:
    """..."""
    approx_rounds = self.cnt_ts_to_approx_rounds(self._cnt_ts, self._bezier_points.shape[1])
    current = [self._bezier_points]
    for _ in range(approx_rounds):
        queue = []
        for c in current:
            queue.extend(subdivision(c))
        current = queue
```

Figure 11: Subdivision Code

With `approx_rounds` determined we start a nested loop in which we subdivide for each array contained in `current`. The resulting `left` and `right` are saved in `queue` which is a temporary storage. At the end we overwrite `current` with `queue`. This is done `approx_round` times. So for each `c` in `current` we get two new arrays. This results in the 2^k Bézier polygons where k equals `approx_rounds`.

2.12 Intersection

In this last section we do not explain another calculation method but instead how one can check whether two Bézier curves intersect thanks to two concepts.

2.12.1 MinMaxBox

The MinMaxBox, which is the smallest axis parallel box containing all control points, uses the property of the convex hull. Since all intermediate points b_i^r are computed by a convex barycentric combination of previous points, we never get points outside the convex hull of the b_i. Therefore the hole curve described by the control polygon lies within the convex hull. So when we want to perform a fast check of intersection we can construct the MinMaxBox of the two curves, which is just a box containing the control polygon and therefore the curve itself as well. To check if these two boxes intersect can easily be done. If we get a false we can return `False`. In the other case we have to be more precise. However we can very fast determine if two curves could intersect, which is pretty useful, because we maybe want to check the intersection of multiple curves and with the help of these boxes we can maybe eliminate some curves without taking much effort.

2.12.2 Approximate Intersection

If the two boxes indicate an intersection we have to be more precise, since the intersection of boxes is very coarse. In this case we use subdivision. We split our original curve into sub polygons and check for these ones if their boxes intersect or not, one might call this "divide and conquer". If they do not intersect we can return `False`. However if they intersect and the areas of the boxes are very small (is defined by a tolerance parameter), we can return `True`, because our boxes intersect and the box itself can be seen as the part of the curve we are looking at. This is reasonable as we are inherently limited by the floating point precision.

In the other cases we just split up again. At the end we check if one of our calls has returned $True$, because then we can say, that the curves intersect with some given tolerance.

```python
@staticmethod
def collision_check(b1: AbstractBezierCurve, b2: AbstractBezierCurve, tol: float = 0.01) -> bool:
    """..."""
    if not b1.min_max_box & b2.min_max_box:
        return False

    if b1.min_max_box.area + b2.min_max_box.area < tol:
        return True

    b1s = subdivision(b1._bezier_points, 0.5)
    b2s = subdivision(b2._bezier_points, 0.5)

    return any(
        AbstractBezierCurve.collision_check(BezierCurveDeCasteljau(left), BezierCurveDeCasteljau(right), tol)
        for left, right in itt.product(b1s, b2s)
    )
```

Figure 12: Approximate Intersection Code

2.13 Benchmarks

In order to compare the different algorithms for computing Bézier curves, we ran benchmarks using the same control points.

2.13.1 Benchmark Configuration

We used the following configuration:

- Dell OptiPlex 5090 Micro
- 11th Gen Intel(R) Core(TM) i5-11500T @ 1.50GHz
- 1x16 GB non-ECC DDR4 RAM
- 256GB NVME M.2 SSD
- Ubuntu 20.04.4 LTS (deployed via PXE)
- Python 3.8.10

The PC was completely idling while running the benchmarks; the idle CPU workload was significantly below 1%.

2.13.2 Results

At first, we compared all viable Bézier curves with each other:

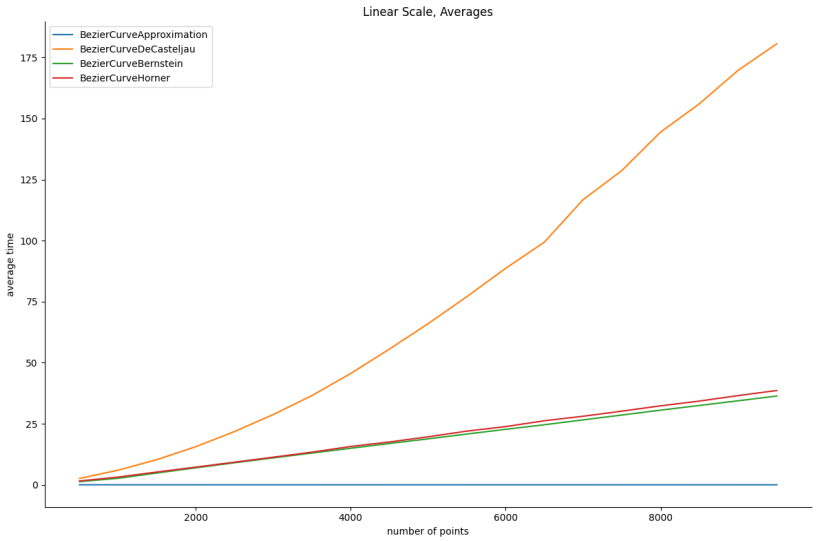


Figure 13: Average runtime in s

To make it more readable, let's look at a logarithmic y-axis:

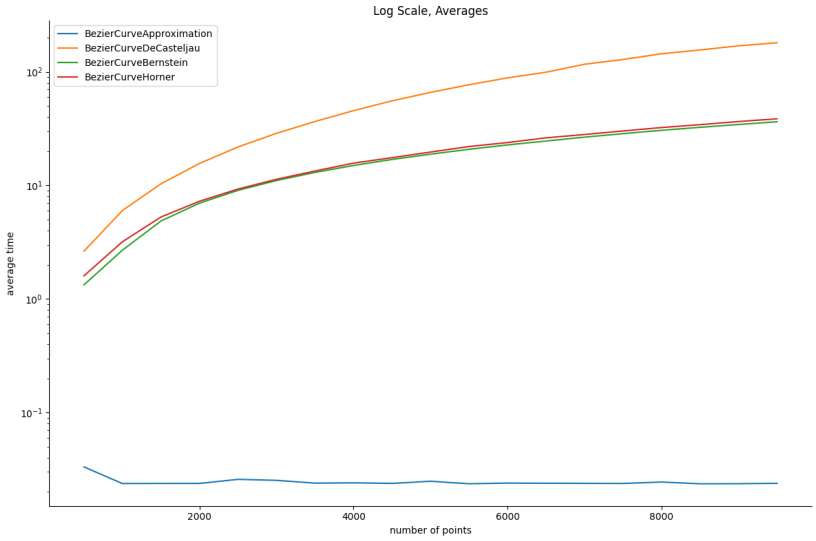


Figure 14: Average runtime in s; logarithmic scale

The approximation algorithm seems to have constant runtime. This is not true; as we increase the problem size the real complexity emerges:

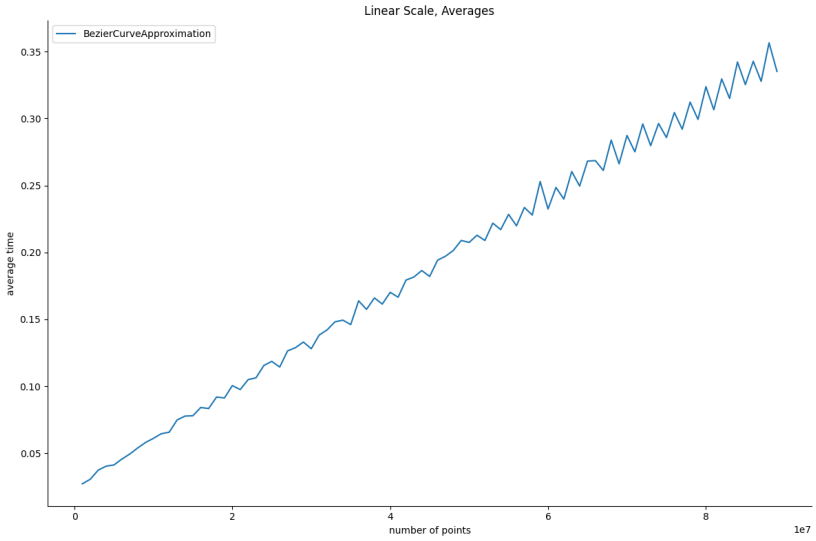


Figure 15: Average runtime in s

Again, in logarithmic scale:

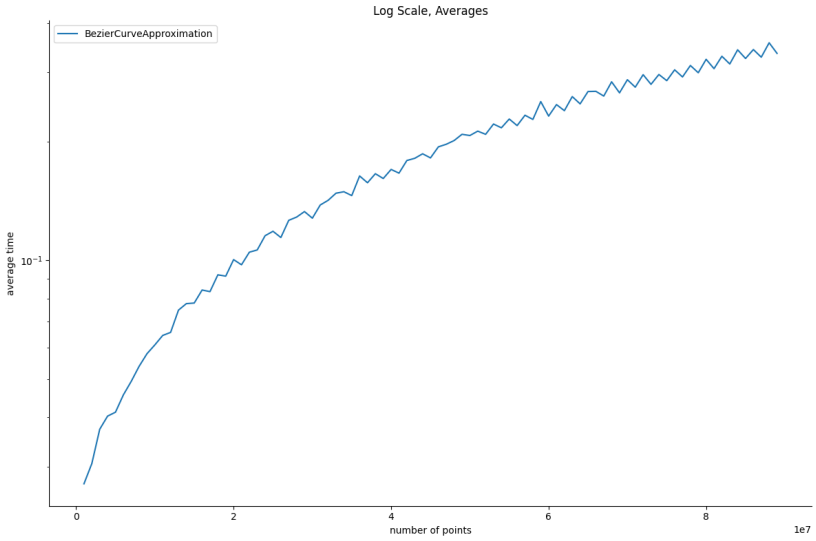


Figure 16: Average runtime in s; logarithmic scale

Those are great results, since the computed Bézier curve is, from a visual perspective, basically indistinguishable.

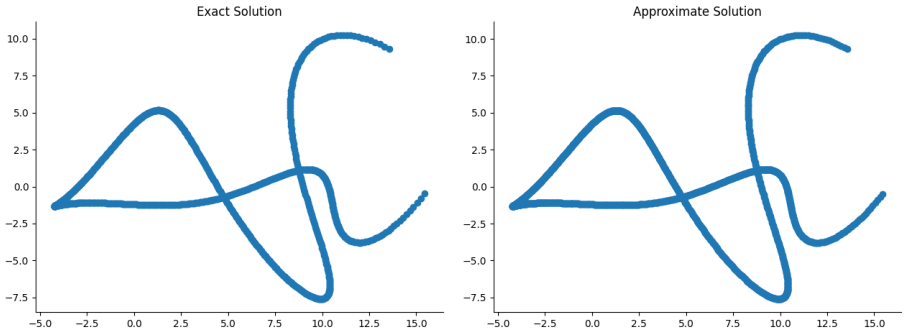


Figure 17: Both algorithms resulting in the approximately same curve

2.13.3 Problems

While benchmarking, we encountered two problems:

1. As mentioned in the literature [Far01], the monomial algorithm is very badly conditioned. Thus, the problem is numerically instable. This would usually not be a problem as Python supports dynamically sized integers. But for large numbers, any floating point values will overflow as they comply with the IEE754 floating point standard. Since $t \in [0, 1]$, this can't be avoided.

2. The Sympy solution does not work for large numbers. This has a rather practical reason. For historical design reasons, Python is designed in such a way that it does not, and will never, support tail call optimization. Therefore the stack space increases linearly with the recursion depth. In order to not get terminated by a segmentation error, Python limits the recursion depth by the variable `sys.setrecursionlimit`. This also limits the solvable problem size of our Sympy based solution.

3 Voronoi Regions, Dirichlet Tessellations and Delaunay Triangulations

3.1 Motivation

Given an arbitrary set of points in the 2D plane, a *Voronoi Diagram* or *Dirichlet Tessellation* divides the plane into tiles; each point is associated with the region of the plane closest to it.

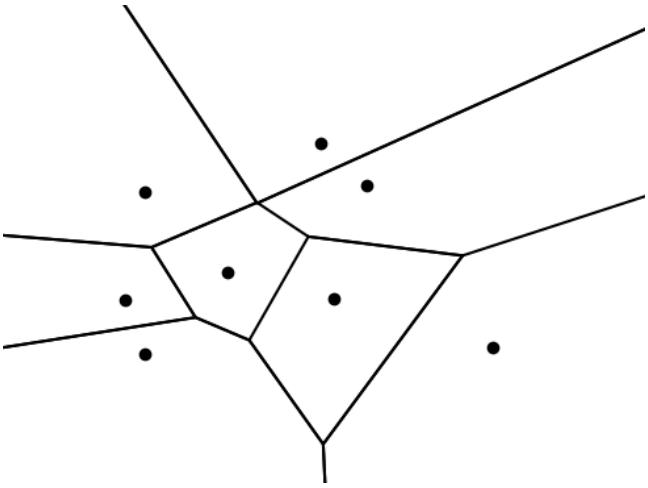


Figure 18: Points and their Voronoi regions

Voronoi diagrams are of monumental importance in many different branches of science and were rediscovered numerous times. Geographers speak of Thiessen polygons, which are used as models for crystal growth, while Metallurgists speak of Wigner-Seitz zones for analyzing equilibrium properties of alloys. As we see later, its dual graph, the *Delaunay Triangulation*, also has various applications in other fields, for example as an automatic mesh generator used in finite element methods. Voronoi Diagrams are especially useful for many problems in the field of computational geometry.

[Aur91] summarizes the importance as follows:

> Accordingly, Voronoi diagrams are useful in three respects: As a structure per se that makes explicit natural processes, as an auxiliary structure for investigating and calculating related mathematical objects, and as a data structure for algorithmic problems that are inherently geometric.

For a more exhaustive list of applications, see [Aur91].

3.2 Definitions and Theorems

We use the formal definitions provided by [GS78].

Definition 16 (Voronoi tile)**.** *Let $\mathcal{P}_N := \{P_1, P_2, \ldots, P_N\}$ be finitely many points in the plane such that $P_i \neq P_j, i \neq j$. The **Voronoi tile** of P_i is the set T_i defined by*

$$T_i = \{x : d(x, P_i) < d(x, P_j), \forall j \neq i\}$$

where d is the euclidean distance.

Note that tiles are infinite if and only if they are not entirely surrounded by other tiles. Further note that all tiles are trivially convex.
Not every point is part of a tile. Let P_n and P_m be 2 points and T_n, T_m be their neighbouring tiles. If x is a point such that $d(x, P_n) = d(x, P_m)$ then x lies in neither T_n nor T_m. This is why tiles are open, all points neighbouring 2 or more tile centers form the **boundary segments** of that tile.

Definition 17 (Voronoi Diagram, Dirichlet Tessellations)**.** *A **Voronoi Diagram**, also called a **Dirichlet Tessellation**, is the set of all tiles and boundaries subdividing a given space.*

Although intuitively understandable, let us formally define what a neighbour is:

Definition 18 (Neighbour)**.** *Two Voronoi Tiles are **neighbours** if and only if they share one boundary segment.*

Through Voronoi diagrams, we can easily get another important geometric structure.

Definition 19 (Triangulation)**.** *A **triangulation** is a subdivision of an object into simplicies. Note that in 2D, this is a subdivision into triangles.*

Definition 20 (Delaunay Triangulation)**.** *Let $\mathcal{P}_N$ be finitely many points in the plane, no two of which coincide. Further let $T_1, \ldots, T_n$ be the Voronoi tiles of those points. The* ***Delaunay Triangulation*** *is a triangulation of these points with the following property:*

$$\forall i, j \in \{1, \ldots, n\}, i \neq j : (P_i, P_j) \text{ are connected } \Leftrightarrow (T_i, T_j) \text{ are neighbours}$$

This makes the Delaunay Triangulation the ***dual graph*** *of the Voronoi Diagram (See [FOR95], Theorem 2.1.3)*

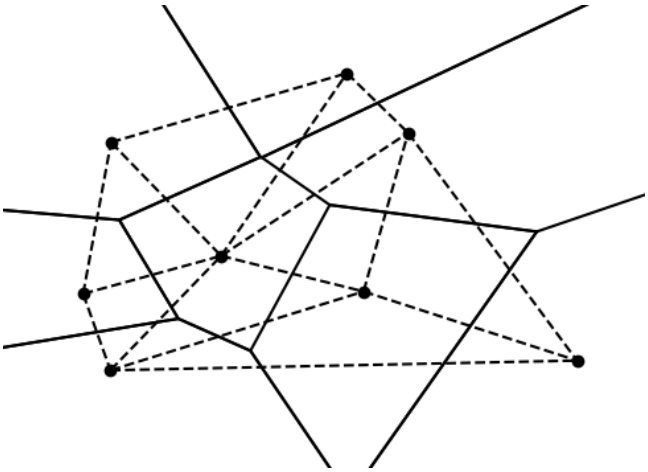


Figure 19: Voronoi Regions and Delaunay triangulations as it's dual graph

The Delaunay Triangulation has a few useful properties, a few of which we list here. For a more in depth discussion, see [Aur91].

Theorem 6 (Existance of a Delaunay Triangulation)**.** *Let $\mathcal{P}_N$ be finitely many points. If the euclidian metric is used, a Delaunay triangulation always exists.*

This follows from the definition of the Voronoi diagram, which always exists.

Theorem 7 (Circumcircle property)**.** *For any Delaunay triangulation, the circumcircle of any triangle does precisely only contain the 3 points defining the triangle.*

Note that this theorem was used by Delaunay as the triangulation definition [Aur91].

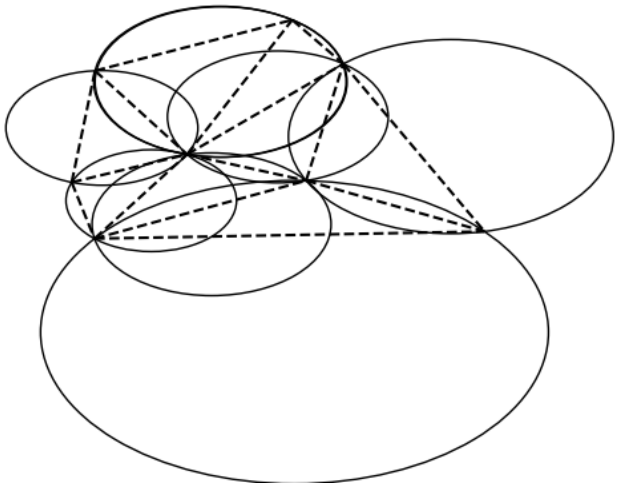


Figure 20: Delaunay triangulations and their Circumcircles

Theorem 8 (Optimality)**.** *Over all proper* 2*D triangulations, the Delaunay triangulation maximizes the minimum angle of any triangle.*

Proof. See [FOR95] Theorem 3.1 □

This implies that the Delaunay triangulation is the most equiangular one, which makes it computationally more desirable in finite element and interpolation methods [Aur91] [Reb93].

3.3 Possible Algorithms

Here follows a short overview on different types on algorithms. For convenience reasons we will present all algorithms in terms of the Delaunay triangulation instead of the Voronoi diagram, which is easily obtained from any graph-based Delaunay representation. For a more detailed discussion, see [FOR95].
The following bounds are known for computing Delaunay triangulations [FOR95]:

- For 2 dimensional Delaunay triangulations, a worst case lower bound of $\Omega(n \log n)$ is known.
- For $d > 2$ dimensional Delaunay triangulations, an lower bound of $\Omega(n^{\lceil d/2 \rceil})$ is known.

3.3.1 Flip Algorithm

Flipping algorithms are the most simple type of algorithm. At first, you start with some kind of triangulation, which doesn't have any further constraints. From there on, we can just "flip" edges until every edge furfills the circumcircle property. The mapping from the starting triangulation to the ending one is called a **transformation**. Formally:

Theorem 9 (flips and flippable edges)**.** *Let Δ_1, Δ_2 be two neighbouring triangles with e being the edge contained in both boundaries. If the union of both triangles $C := \Delta_1 \cup \Delta_2$ forms a convex quadrilateral, e is **flippable**. **Flipping** e means replacing e with it's diagonal in C.*

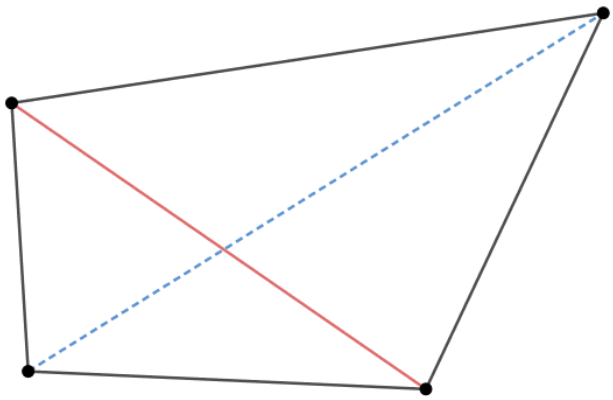


Figure 21: A flippable edge (red), it's alternative (blue) and the required convex quadrilateral (black)

While this class of algorithms is conceptually very simple, it has very poor performance, with some transformations requiring at least $\Omega(n^2)$ flips [HNU99].

3.3.2 Incremental Algorithm

The incremental algorithm works by using the circumcircle property inductively. It was first introduced for two dimensions by [GS78], which was later generalized to n-dimensions by both [Bow81] and [Wat81] simultaneously.
We will inspect this algorithm in more rigorously in section 3.4, but the basic idea works as follows:

```
func IncDelaunay(pts) {
    let delaunay = new EmptyDelaunay()
    for pt in pts {
        // Note: Inductively, we assume that we currently
        // have a correct delaunay triangulation to which
        // we add our new point

        // Step 1: Add each point
        delaunay.add(pt)
        // Step 2: Check which triangles are broken now
        // A triangle D is broken iff the new point is in D's
        // circumcircle
        let broken_triangles = delaunay.find_all_broken_triangles()


        // Step 2.5: Remove those triangles, as they can't be fixed
        delaunay.remove_triangles(broken_triangles)

        // Step 3: Connect the edge which is still part of one healthy triangle
        // to our new point
        // This means that we visually "fill" the hole we created in Step 2.5
        for broken_t in broken_triangles {
            let healthy_edge = extract_healthy_edge(broken_t)
            let triangle = join_triangle(healthy_edge, pt)
            delaunay.add_triangle(triangle)
        }

        // Now we have a valid triangulation again, thus we can add the next point
    }
}
```

Figure 22: Pseudocode Green-Sibson/Bowyer/Watson

The average case time complexity is $O(n^{1+1/d})$ where d is the number of dimensions [Bow81].

3.3.3 Divide and Conquer Algorithm

A **Divide and Conquer** algorithm can be broken down into 3 steps:

1. **Divide** the problem into smaller subproblems
2. **Conquer** the subproblems by solving them recursively until they are small enough to be solvable straightforward.

3. **Merge** the solutions of those smaller subproblems into the original problem

This results in the following cost function

$$T(n) := a \cdot T(n/b) + c(n)$$

where

- a are the number of subproblems,
- n/b is the size of the subproblem,
- $c(n)$ is the cost function matching the cost of the combine step.

This is equivalent to the following Pseudocode

```
func DivideAndConquer(xs) {
  let middle := len(xs)/2
  if (xs is too big to solve) {
    let left := DivideAndConquer(xs[:middle])
    let right := DivideAndConquer(xs[middle:])
    return merge(left, right),
  }
  return solve(xs)
}
```

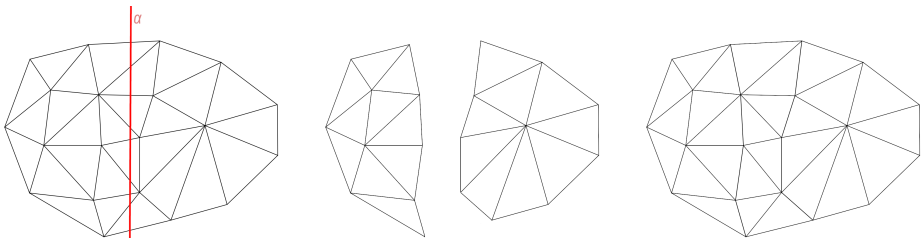
Figure 23: Pseudocode D&C

There are 2 ways to use the D&C paradigm for constructing Delaunay Triangulations.

The **naive way** would work as follows:

```
func NaiveDNC(pts) {
    // Step 0: If it is trivially, solve it directly
    // This is the recursive base case
    if (len(pts) < trivial_threshold) {
        return solve(pts)
    }

    // Step 1: Choose a plane to split
    // Try to make both resulting sets be of roughly same size
    let splitting_plane := generate_splitting_plane(pts)
    let l, r := divide_by_plane(pts, splitting_plane)

    // Step 2: solve both sides recursively
    let delaunay_l = NaiveDNC(l)
    let delaunay_r = NaiveDNC(r)

    // Step 3: Triangulize the edges crossing our splitting_plane
    return merge_along_plane(delaunay_l, delaunay_r, splitting_plane)
}
```

Figure 24: Pseudocode D&C for naively creating a Delaunay Triangulation



(i) Divide into 2 subproblems

(ii) Solve the subproblems recursively

(iii) Merge the subproblems

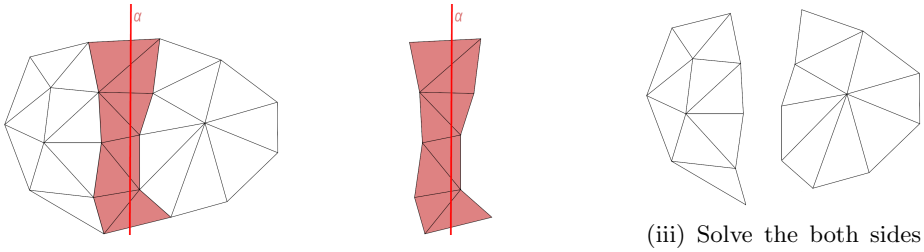
Figure 25: Naive D&C Algorithm, Visually

Note that, while merging both subproblems, this algorithm can still require additional changes to the already solved triangulations.
The better and more generalizable approach was first introduced by [CMS98]. The so-called **DeWall** algorithm can be simplified to the following idea:

```
func DeWallDNC(pts) {
    // Step 0: If we reached the base case where it is already
    // solved, return
    if (len(pts) <= 1) {
        return pts
    }

    // Step 1: Choose a plane to split
    let splitting_plane := generate_splitting_plane(pts)
    let l, r := divide_by_plane(pts, splitting_plane)

    // Step 1.5: Get all points that have lines crossing it
    let relevant_pts := filter(
        lambda x -> contains_edge_crossing(x,splitting_plane),
        pts
    )

    // Step 2: Build a simplex wall along the splitting plane
    let wall := build_delaunay_trig(relevant_pts)

    // Step 3: Join the delaunay wall with the subproblems
    let left_delaunay := DeWallDNC(l)
    let right_delaunay := DeWallDNC(r)
    return merge(left_delaunay, wall, right_delaunay)
}
```

Figure 26: Pseudocode D&C DeWall

Visually:



(i) Cut into 2 problems (ii) Solve the Middle first (iii) Solve the both sides recursively like (ii)

Figure 27: DeWall Algorithm, Visually

The worst case time complexity of DeWall is known to be $O(n^{\lceil d/2 \rceil + 1})$, with d being the number of dimensions

3.3.4 S-hull Algorithm

The **S-hull** [Sin10] algorithm belongs to the family of the so-called line sweep algorithms, which describes solving a problem geometrically in an space by traversing it and working with distances. Note that this requires an euclidean norm.
[Sin10] contains a great pseudocode summary:

> S-hull operates as follows: For a set of unique points $x_i \in \mathbb{R}^2$:
>
> 1. select a seed point x_i from x_i.
> 2. sort according to $|x_i - x_0|^2$.
> 3. find the point x_j closest to x_0.
> 4. find the point x_k that creates the smallest circum-circle with x_0 and x_j and record the center of the circumcircle C.
> 5. order points $[x_0, x_j, x_k]$ to give a right handed system: this is the initial seed convex hull
> 6. resort the remaining points according to $|x_i - C|^2$ to give points s_i.
> 7. sequentially add the points s_i to the porpagating 2D convex hull that is seeded with the triangle formed from $[x_0, x_j, x_k]$. As a new point is added the facets of the 2D-hull that are visible to it form new triangles.
> 8. a non-overlapping triangulation of the set of points is created. (This is an extremly fast method for creating an non-overlapping triangulation of a 2D point set).

9. adjacent pairs of triangles of this triangulation must be 'flipped' to create a Delaunay triangulation from the initial non-overlapping triangulation.

This algorithm just works on 2D and it's worst case time complexity is $O(n \log n)$.

3.3.5 q-hull Algorithm

Definition 21 (Convex Set)**.** *A Set S is **convex** if, for all $x, y \in S$, the line segment $L_{x,y} := \{tx + (1-t)y : t \in [0,1]\}$ is in S.*

Definition 22 (Convex Hull)**.** *A **Convex Hull** of a set of N points is defined as the smallest convex set which contains all of the points.*

In $\mathbb{R}^2$, this is a convex polygon of at most N sides.

Theorem 10 (Delaunay Triangulation/Convex Hulls)**.** *A Delaunay triangulation in $\mathbb{R}^d$ can be created by computing the convex hull in R^{d+1}, followed by then extracting the set of ridges of the lower convex hull.*

Proof. Constructive proof, see [Bro79]. □

This is the most common way for computing Delaunay triangulation, as most numerical libraries like Scipy [**Scipy2022**] use the qhull library [**QHull**] based on the Quickhull Algorithm [BDH96].

3.4 Our Algorithm and Implementation

We chose to implement an incremental algorithm which is a simplified version of [GS78], which in itself is a 2 dimensional version of [Bow81] and [Wat81], in which finding the triangle containing a new point takes $O(n)$, instead of the $O(\sqrt{n})$ neighbour walk alleged by [Bow81].

3.4.1 Basic Idea

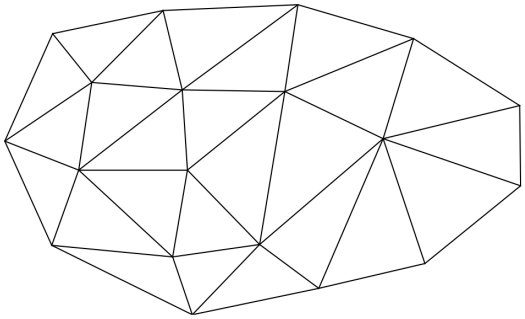
As mentioned before, all incremental algorithms work inductively by assuming a valid Delaunay triangulation of $n-1$ points, adding the n-th point, then repairing the triangulation afterwards.

[Reb93] summarizes the algorithm as follows:

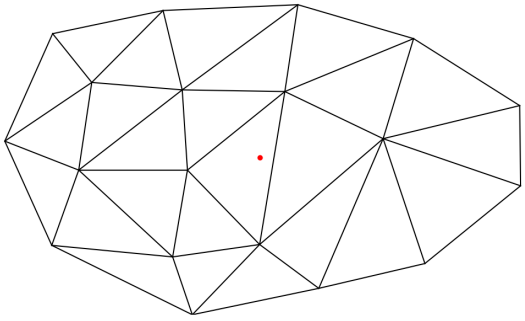
> The method is based on the so-called *circumcircle property* which guarantees that no point of a Delaunay triangulation can lie within the circle circumscribed to any triangle. The Bowyer-Watson algorithm is essentially a "reconnection" method, since it computes how an existing Delaunay triangulation is to be modified because of a new point.

3.4.2 A visual example

Assume we have generated the following valid Delaunay triangulation of $n-1$ points:

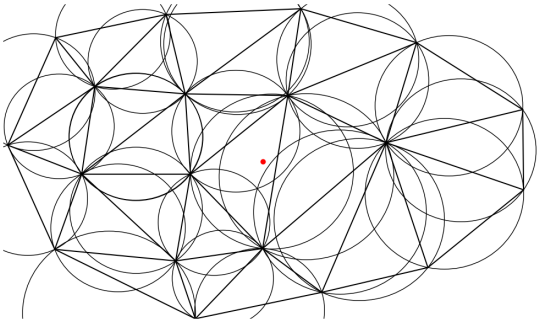


Now we want to add a new point into this triangulation.

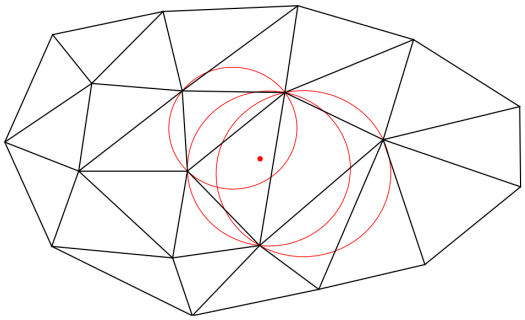


The circumcircle property shows us that any valid Delaunay triangulation has no other points in the circumcircles spanned by all triangles. This also means that any triangle that contains our new point can't be valid!
Note that we just have to focus on our newly inserted point since we know that we started with an already valid Delaunay triangulation.

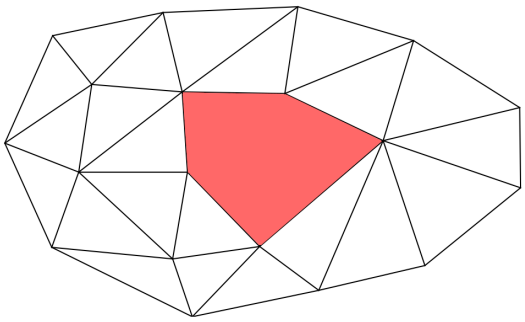
Let's look at the circumcircles:



Now just show the invalid ones:

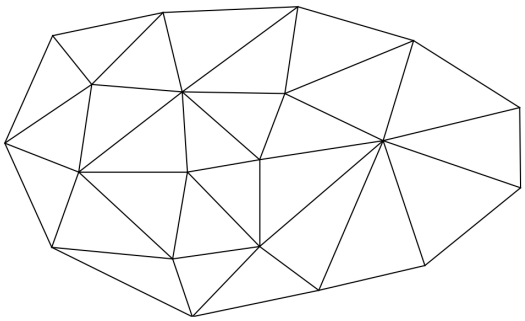


We now know that those triangles are broken. Thus we remove them.

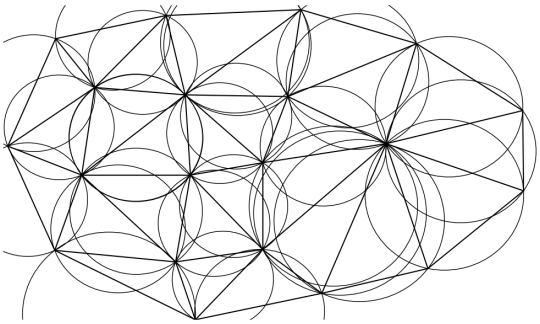


Now we only have one valid way to fill the hole: Connecting all leftover edges to our new point. This makes sense intuitively, since this way results in the most equilateral triangles, thus furfilling Delaunay's optimality criteria. Formally, we know that a triangulation is Delaunay if and only if it is locally Delaunay, which

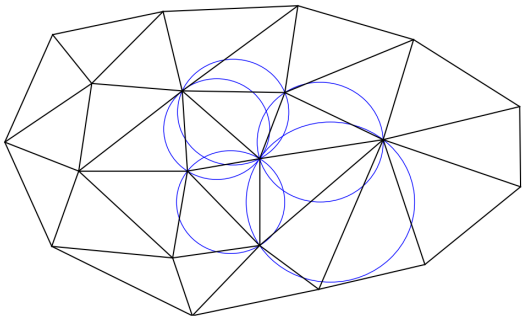
means that any non-optimal solution wouldn't be possible ([FOR95], Lemma 2.2).
So let us reconnect them:



Now we have a valid triangulation of n points. To verify, here are all circumcircles:



And for clarity, just the new circumcircles:



Now all are valid again, resulting in a new Delaunay triangulation of n points.

3.4.3 Initial State

The incremental method works inductively and as we have just seen our induction step is correct. Now we have to handle the base case. This is more difficult because any set with less than 3 points is not triangularizable.
In order to make it easier, we use Green-Sibson's window [GS78] and extend it with a supertriangle [Che].

Definition 23 (Window)**.** *A **window** is a axially parallel rectangle which contains all points of the Delaunay triangulation.*

Now we also have to extend our Voronoi tile definition:

Definition 24 (Window-aware Voronoi tiles)**.** *Let $\mathcal{P}_N := \{P_1, \ldots, P_N\}$ be finitely many points in the plane such that $P_i \neq P_j, i \neq j$. Further let E be a window such that all points fit. The **window-aware Voronoi tile** of P_i is the set T_i^* defined by*

$$T_n^* := \{x \in E : d(x, P_i) < d(x, P_j) \forall i \neq j, P_j \in E\}$$

In our algorithm, all Voronoi tiles are window-aware.

The window used in our algorithm is square. This window allows us to have a boundary, in which all points are contained. In order to make the window a valid triangulation we extend it to a super-triangle:

Definition 25 (Supertriangle)**.** *A **supertriangle** is a window with an additional diagonal edge, dividing it into 2 triangles.*

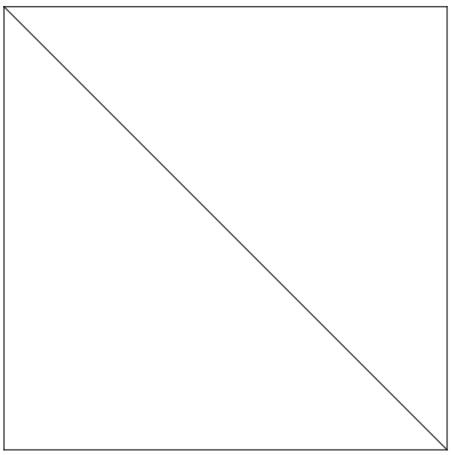


Figure 28: Supertriangle, all points outside of it are invalid

This supertriangle is only used internally; all edges and vertices are not part of the Delaunay triangulation.

3.4.4 Mesh Data Structure

The most important part of this algorithm is fast graph traversal, thus we heavily rely on a efficient mesh data structure. Our data structures are defined as follows:

```
struct Circle {
    center: Point2D
    radius: float
    func contains(pt) {
        return euclid_norm(pt-center) <= radius
    }
    // other funcs
}

struct Triangle {
    // Stored in counterclockwise order
    points: (Point2D, Point2D, Point2D)

    // Actually lazily computed and cached
    circumcircle: Circle
}

struct NeighbourMesh {
    // Stored in counterclockwise order
    mesh: HashMap<Triangle, (Triangle, Triangle, Triangle)>
}
```

The Mesh gets updated while adding new points.
This Data structure has multiple great properties:

- Since we only have 2 dimensions and have the circle as a member variable, detecting whether a point lies in a triangles circumcircle is $\Theta(1)$.
- Although Python dictionaries (i.e. Hashmaps) use open hashing, our hash function is chosen in a way that the hash seed of two triangles is only equal if and only if they share all 3 points. Thus our triangle lookup results in $\Theta(1)$.
- Since any triangle only has 3 neighbours, we can traverse n triangles in $\Theta(n)$, which is obviously optimal.

3.4.5 Triangle Replacement Optimization

In order to optimize the triangle replacement we use 2 useful properties from [Reb93]:

> The algorithm removes from the existing grid all the triangles which violate the circumcircle property because of the insertion of the new point. It can be shown that
>
> 1. all these triangles are always contiguous, thus forming a connected cavity surrounding the newly inserted point, and that
> 2. by joining the vertices of the cavity with the internal new point, a Delaunay triangulation is always obtained.

Our optimization thus consists of three parts:

- The aforementioned $O(1)$ traversal neighbour mesh data structure
- Remembering which edges need to be reconnected while deleting the broken triangles
- Only traversing along the hole edges, starting at a random edge. This works because we know that the hole is contiguous.

3.4.6 Triangle Replacement Implementation

This is the code for adding a new point to the Delaunay triangulation:

```
def add_point(self, p: Point2D):
    """The main method of adding a new point...."""
    bad_triangles = [bad for bad in self._neighbours.keys() if p in bad.circumcircle]

    boundary = self.do_boundary_walk(p, bad_triangles)

    for bad in bad_triangles:
        del self._neighbours[bad]

    # Add new TupleTriangle Entries
    n = len(boundary)
    for i, b in enumerate(boundary):
        triangle_before, triangle_after = boundary[(i - 1) % n][0], boundary[(i + 1) % n][0]
        # other way around to ensure CCW
        self._neighbours[b.new_triangle] = [b.opposite_triangle, triangle_after, triangle_before]

    # Add new triangles to the opposite side
    for b in boundary:
        if b.opposite_triangle is None:
            continue

        for i, neigh in enumerate(self._neighbours[b.opposite_triangle]):
            if neigh is not None and set(b.connecting_edge).issubset(set(neigh.points)):
                self._neighbours[b.opposite_triangle][i] = b.new_triangle
```

```
def do_boundary_walk(self, p: Point2D, bad_triangles: List[TupleTriangle]) -> List[
    DelaunayTriangulation2D._BoundaryNode]:
    """The boundary walk which finds walks along the cavity of the removed (bad) triangles...."""
    boundary = []
    current_triangle, i = bad_triangles[0], 0

    while True:
        opposite_triangle = self._neighbours[current_triangle][i]
        if opposite_triangle in bad_triangles:
            i = (self._neighbours[opposite_triangle].index(current_triangle) + 1) % 3
            current_triangle = opposite_triangle
            continue

        # remember, CCW
        edge = (current_triangle.points[(i + 1) % 3], current_triangle.points[(i - 1) % 3])
        self._BoundaryNode(TupleTriangle(p, *edge), edge, opposite_triangle)
        boundary.append(self._BoundaryNode(TupleTriangle(p, *edge), edge, opposite_triangle))
        i = (i + 1) % 3
        if boundary[0].connecting_edge[0] == boundary[-1].connecting_edge[1]:
            return boundary
```

It works as follows:

- At first, we detect all broken triangles by checking their circumcircles
- Now we do the **boundary walk** that works as follows:

We start at any triangle. Note that we cannot yet know whether it is entirely surrounded by broken triangles or not. We choose any edge of it. Then we rotate along the edges counter clock wise (CCW) and go to the triangle that is connected to our triangle through this edge. One can see that through this motion we go in a more or less specific direction.
Eventually (in reality it's pretty fast, [GS78] shows that through the Euler-Poincare formula) we reach an edge that is connected to a valid triangle (i.e. a triangle which circumcircle contains only it's 3 points). This is relevant because we have to reconnect the edge connecting the broken and valid triangle to our new point.
We call this edge-path of valid-to-invalid triangles our **boundary**.
Now, by moving counter clock wise, we stay along the boundary. We record not only every boundary edge, but also to which boundary edges it is connected to. This allows for linear time insertion of the new triangles since we do not have to search the neighbour (which, combinatorically, would result in $O(n^2)$ where n is the number of boundary edges).

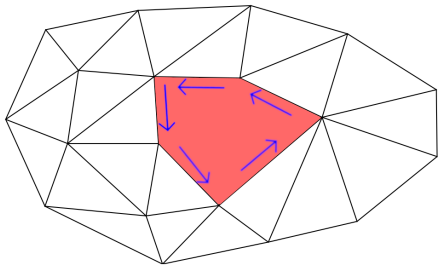


Figure 29: CCW walk. We always check that we are connected to one valid and one broken triangle

Note: the neighbours do not have to be saved explicitly. Since it is a list, and we add it one by one while walking along the neighbourship, it is implicitly ordered.
We stop once we meet our first boundary triangle again (we ran a circle). *This is the main algorithm optimization.*

- After collecting the boundary, we can delete all broken triangles.
- Lastly, we walk along the boundary and create new triangles, consisting the boundary edge connected to our new point. Note that we can add

the triangle neighbours in $O(1)$ because our boundary is ordered (see the traversal).

3.5 Runtime Analysis

The initial data structure creation is $\Theta(1)$. For each point, we have the following costs for adding the n-th point:

1. To detect all broken triangles, we look at all triangles. As explained in the data structure section, this lookup is $\Theta(1)$, resulting in $\Theta(n)$
2. The boundary walk takes $\Theta(1)$ per triangle. Thus we can choose an upper bound of $O(n)$, although this bound is not very tight.
3. Deleting a bad triangle takes $\Theta(1)$. Thus $\Theta(1) \cdot |\texttt{bad\_triangles}| \in O(n)$.
4. Connecting one boundary edge with the new point, creating a new triangle and updating our neighbour mesh all takes $\Theta(1)$. Again, its upper bound is $O(n)$

Resulting in a runtime of $\Theta(n) + 3 \cdot O(n) = O(n)$ per point.
If we have m points, this results in $O(m^2)$.

3.5.1 Possible Improvements and Beyond

As shown by Bowyer-Watson [Bow81] [Wat81], this algorithm could theoretically be generalized to n dimensions.
Both Green-Sibson [GS78], Bowyer [Bow81] and [Reb93] alledge that the nearest-neighbour search (i.e. finding the triangle our new point is contained in) can be implemented in $O(n^{1/d})$ with d being the dimension. Sadly, this was never proven rigorously [FOR95]. Also, no algorithm is mentioned in the literature. For now, the question on whether this is actually possible remains open [Que].

References

[AKI12] Sterling J Anderson, Sisir B. Karumanchi, and Karl Iagnemma. "Constraint-based planning and control for safe, semi-autonomous operation of vehicles". In: *2012 IEEE Intelligent Vehicles Symposium*. 2012, pp. 383–388. DOI: 10.1109/IVS.2012.6232153.

[Aur91] Franz Aurenhammer. "Voronoi diagrams—a survey of a fundamental geometric data structure". In: *ACM Computing Surveys* 23.3 (Sept. 1991), pp. 345–405. DOI: 10.1145/116873.116880. URL: https://doi.org/10.1145/116873.116880.

[BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. "The quickhull algorithm for convex hulls". In: *ACM Transactions on Mathematical Software* 22.4 (Dec. 1996), pp. 469–483. DOI: 10.1145/235815.235821. URL: https://doi.org/10.1145%2F235815.235821.

[Boc+09] Martin Bock et al. "Generalized Voronoi Tessellation as a Model of Two-dimensional Cell Tissue Dynamics". In: *arXiv e-prints*, arXiv:0901.4469 (Jan. 2009), arXiv:0901.4469. arXiv: 0901.4469 [physics.bio-ph].

[Bow81] A. Bowyer. "Computing Dirichlet tessellations". In: *The Computer Journal* 24.2 (Feb. 1981), pp. 162–166. DOI: 10.1093/comjnl/24.2.162. URL: https://doi.org/10.1093/comjnl/24.2.162.

[Bro79] Kevin Q. Brown. "Voronoi diagrams from convex hulls". In: *Information Processing Letters* 9.5 (Dec. 1979), pp. 223–228. DOI: 10.1016/0020-0190(79)90074-7. URL: https://doi.org/10.1016%2F0020-0190%2879%2990074-7.

[Che] Benny Cheung. *PyDelaunay*. URL: https://github.com/bennycheung/PyDelaunay (visited on 07/07/2022).

[CMS98] P Cignoni, C Montani, and R Scopigno. "DeWall: A fast divide and conquer Delaunay triangulation algorithm in Ed". In: *Computer-Aided Design* 30.5 (Apr. 1998), pp. 333–341. DOI: 10.1016/s0010-4485(97)00082-1. URL: https://doi.org/10.1016/s0010-4485(97)00082-1.

[Far01] Gerald Farin. *Curves and Surfaces for CAGD: A Practical Guide*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1558607374.

[FOR95] STEVEN FORTUNE. "VORONOI DIAGRAMS and DELAUNAY TRIANGULATIONS". In: *Lecture Notes Series on Computing*. WORLD SCIENTIFIC, Jan. 1995, pp. 225–265. DOI: 10.1142/9789812831699_0007. URL: https://doi.org/10.1142/9789812831699_0007.

[GS78] P. J. Green and R. Sibson. "Computing Dirichlet Tessellations in the Plane". In: *The Computer Journal* 21.2 (May 1978), pp. 168–173. DOI: 10.1093/comjnl/21.2.168. URL: https://doi.org/10.1093/comjnl/21.2.168.

[HNU99] F. Hurtado, M. Noy, and J. Urrutia. "Flipping Edges in Triangulations". In: *Discrete and Computational Geometry* 22.3 (Oct. 1999), pp. 333–346. DOI: 10.1007/pl00009464. URL: https://doi.org/10.1007/pl00009464.

[Li+12] Hui Li et al. "Spatial modeling of bone microarchitecture". In: *Three-Dimensional Image Processing (3DIP) and Applications II*. Ed. by Atilla M. Baskurt and Robert Sitnik. Vol. 8290. Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series. Mar. 2012, 82900P. DOI: 10.1117/12.907371.

[Que] Lars Quentin. *Bowyer-Watson Delaunay Triangulation neighbour walk in $O(n^{1/d})$*. URL: https://cs.stackexchange.com/questions/148452/bowyer-watson-delaunay-triangulation-neighbour-walk-in-on1-d (visited on 07/07/2022).

[Reb93] S. Rebay. "Efficient Unstructured Mesh Generation by Means of Delaunay Triangulation and Bowyer-Watson Algorithm". In: *Journal of Computational Physics* 106.1 (May 1993), pp. 125–138. DOI: 10.1006/jcph.1993.1097. URL: https://doi.org/10.1006/jcph.1993.1097.

[Sin10] D. A. Sinclair. *S-hull: a fast radial sweep-hull routine for Delaunay triangulation*. 2010. URL: http://www.s-hull.org/paper/s_hull.pdf (visited on 07/06/2022).

[Tim] Maximilian Winkler Timo Specht Lars Quentin. *NumerikGang: Curvepy*. URL: https://numerikgang.pages.gwdg.de/ (visited on 07/07/2022).

[Wat81] D. F. Watson. "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes". In: *The Computer Journal* 24.2 (Feb. 1981), pp. 167–172. DOI: 10.1093/comjnl/24.2.167. URL: https://doi.org/10.1093/comjnl/24.2.167.