Splines in Euclidean Space and Beyond

Release 0.1.0

Matthias Geier

2021-02-28

Contents

1 Polynomial Curves in Euclidean Space **2**
- 1.1 Polynomial Parametric Curves . . . 2
- 1.2 Lagrange Interpolation . . . 4
- 1.3 Hermite Splines . . . 12
- 1.4 Natural Splines . . . 30
- 1.5 Bézier Splines . . . 38
- 1.6 Catmull–Rom Splines . . . 52
- 1.7 Kochanek–Bartels Splines . . . 84
- 1.8 End Conditions . . . 98
- 1.9 Piecewise Monotone Interpolation . . . 103

2 Rotation Splines **117**
- 2.1 Quaternions . . . 117
- 2.2 Spherical Linear Interpolation (Slerp) . . . 120
- 2.3 De Casteljau's Algorithm . . . 125
- 2.4 Uniform Catmull–Rom-Like Quaternion Splines . . . 129
- 2.5 Non-Uniform Catmull–Rom-Like Rotation Splines . . . 131
- 2.6 Kochanek–Bartels-like Rotation Splines . . . 134
- 2.7 "Natural" End Conditions . . . 136
- 2.8 Barry–Goldman Algorithm . . . 138
- 2.9 Cumulative Form . . . 141
- 2.10 Naive 4D Quaternion Interpolation . . . 145
- 2.11 Naive Interpolation of Euler Angles . . . 147

3 Python Module **148**
- 3.1 splines . . . 149
- 3.2 splines.quaternion . . . 153

4 External Resources **156**

5 References **156**

References **156**

. . . with focus on univariate, non-uniform piecewise cubic polynomial curves in one, two and three spatial dimensions, as well as rotation splines.

- **Installation**:

```
python3 -m pip install splines
```

- **Online documentation**:
  https://splines.readthedocs.io/
- **Documentation notebooks on Binder**:
  https://mybinder.org/v2/gh/AudioSceneDescriptionFormat/splines/master?filepath=doc/index.ipynb
- **Source code repository (and issue tracker)**:
  https://github.com/AudioSceneDescriptionFormat/splines
- **License**:
  MIT – see the file `LICENSE` for details.

---

1 Polynomial Curves in Euclidean Space

The following section was generated from `doc/euclidean/polynomials.ipynb`

1.1 Polynomial Parametric Curves

[1]:
```
import sympy as sp
sp.init_printing(order='grevlex')
```

[2]:
```
t = sp.symbols('t')
```

The coefficients are written as bold letters, because they are elements of a vector space (e.g. $\mathbb{R}^3$).

We are using bold symbols because apart from simple scalars (for one-dimensional functions), these symbols can also represent vectors in two- or three-dimensional space.

[3]:
```
coefficients = sp.Matrix(sp.symbols('a:dbm')[::-1])
coefficients
```

[3]:
$$\begin{bmatrix} \boldsymbol{d} \\ \boldsymbol{c} \\ \boldsymbol{b} \\ \boldsymbol{a} \end{bmatrix}$$

Monomial basis functions:

[4]:
```
b_monomial = sp.Matrix([t**3, t**2, t, 1]).T
b_monomial
```

[4]:
$$\begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}$$

[5]:
```
b_monomial.dot(coefficients)
```

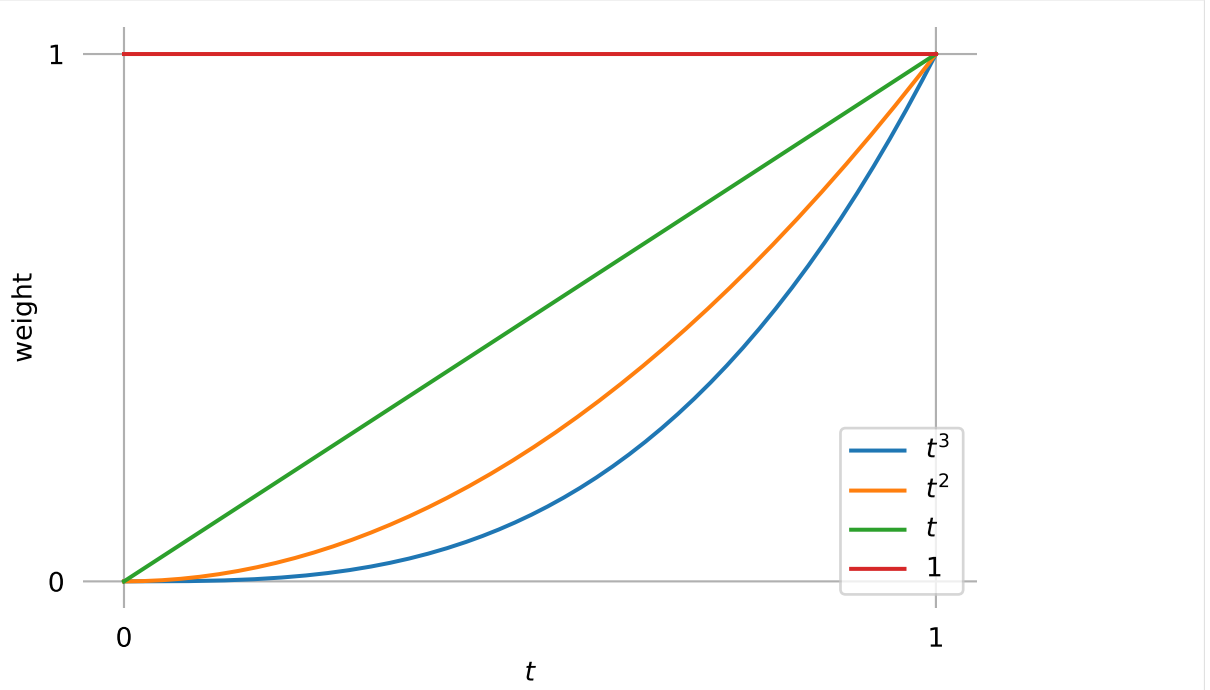
[5]:
$$\boldsymbol{d}t^3 + \boldsymbol{c}t^2 + \boldsymbol{b}t + \boldsymbol{a}$$

This is a cubic polynomial in its canonical form (monomial basis).

Monomial basis functions:

[6]:
```
from helper import plot_basis
```

[7]:
```
plot_basis(*b_monomial, labels=b_monomial)
```



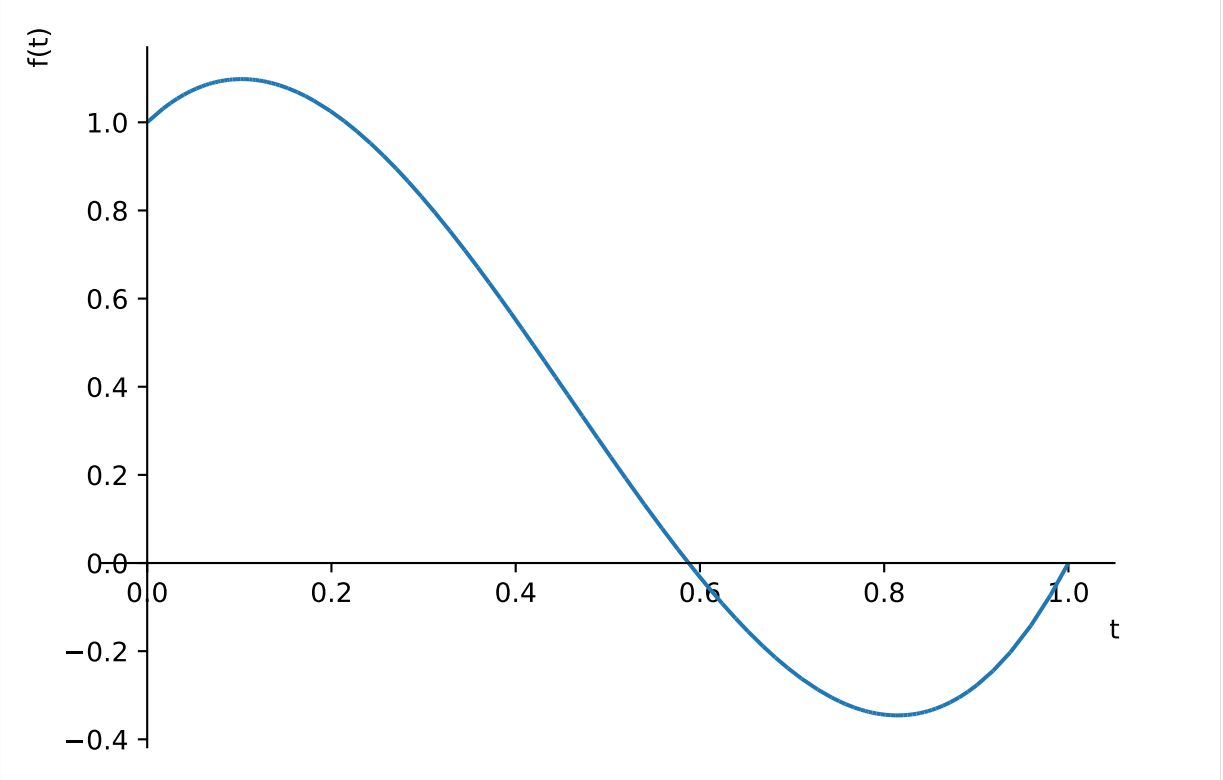
It doesn't look like much, but every conceivable cubic polynomial can be formulated as exactly one linear combination of those basis functions.

Example:

[8]:
```
example_polynomial = (2 * t - 1)**3 + (t + 1)**2 - 6 * t + 1
example_polynomial
```

[8]: $(2t-1)^3+(t+1)^2-6t+1$

[9]:
```
sp.plot(example_polynomial, (t, 0, 1));
```



[9]:
```
<sympy.plotting.plot.Plot at 0x7ff97a689a90>
```

Can be re-written with monomial basis functions:

[10]:
```
example_polynomial.expand()
```

[10]: $8t^3 - 11t^2 + 2t + 1$

[ ]:

doc/euclidean/polynomials.ipynb ends here.

The following section was generated from doc/euclidean/lagrange.ipynb

1.2 Lagrange Interpolation

Before diving into splines, let's have a look at an arguably simpler interpolation method using polynomials: Lagrange interpolation[1].

This is easy to implement, but as we will see, it has quite severe limitations, which will motivate us to look into splines later.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

[1] https://en.wikipedia.org/wiki/Lagrange_polynomial

One-dimensional Example

Assume we have N time instants t_i, with $0 \leq i < N$:

[2]:
```
ts = -1.5, 0.5, 1.7, 3, 4
```

... and for each time instant we are given an associated value x_i:

[3]:
```
xs = 2, -1, 1.3, 3.14, 1
```

Our task is now to find a function that yields the given x_i values for the given times t_i and some "reasonable" interpolated values when evaluated at time values in-between.

The idea of Lagrange interpolation is to create a separate polynomial for each of the N given time instants, which will be weighted by the associated x. The final interpolation function is the weighted sum of these N polynomials.

In order for this to actually work, the polynomials must fulfill the following requirements:

- Each polynomial must yield 1 when evaluated at its associated time t_i.
- Each polynomial must yield 0 at all other instances in the set of given times.

To satisfy the second point, let's create a product with a term for each of the relevant times and make each of those factors vanish when evaluated at their associated time. As an example we look at the basis for $t_3 = 3$:

[4]:
```
def maybe_polynomial_3(t):
    t = np.asarray(t)
    return (t - (-1.5)) * (t - 0.5) * (t - 1.7) * (t - 4)
```

[5]:
```
maybe_polynomial_3(ts)
```

[5]:
```
array([ -0.   ,   0.   ,  -0.   , -14.625,   0.   ])
```

As we can see, this indeed fulfills the second requirement. Note that we were given 5 time instants, but we need only 4 product terms (corresponding to the 4 roots of the polynomial).

Now, for the first requirement, we can divide each term to yield 1 when evaluated at $t = 3$ (luckily, this will not violate the second requirement). If each term is 1, the whole product will also be 1:

[6]:
```
def polynomial_3(t):
    t = np.asarray(t)
    return (
        (t - (-1.5)) / (3 - (-1.5)) *
        (t - 0.5) / (3 - 0.5) *
        (t - 1.7) / (3 - 1.7) *
        (t - 4) / (3 - 4))
```

[7]:
```
polynomial_3(ts)
```

[7]:
```
array([ 0., -0.,  0.,  1., -0.])
```

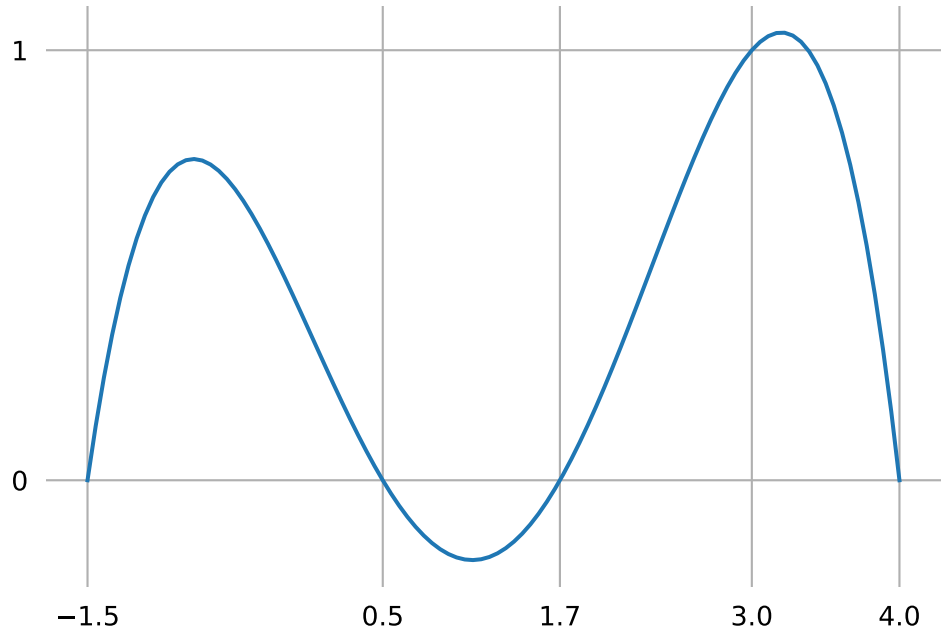
That's it!

To get a better idea what's going on between the given time instances, let's plot this polynomial (with a little help from helper.py):

[8]:
```
from helper import grid_lines
```

[9]:
```
plot_times = np.linspace(ts[0], ts[-1], 100)
```

[10]:
```
plt.plot(plot_times, polynomial_3(plot_times))
grid_lines(ts, [0, 1])
```



We can see from its shape that this is a polynomial of degree 4, which makes sense because the product we are using has 4 terms containing one t each. We can also see that it has the value 0 at each of the initially provided time instances t_i, except for $t_3 = 3$, where it has the value 1.

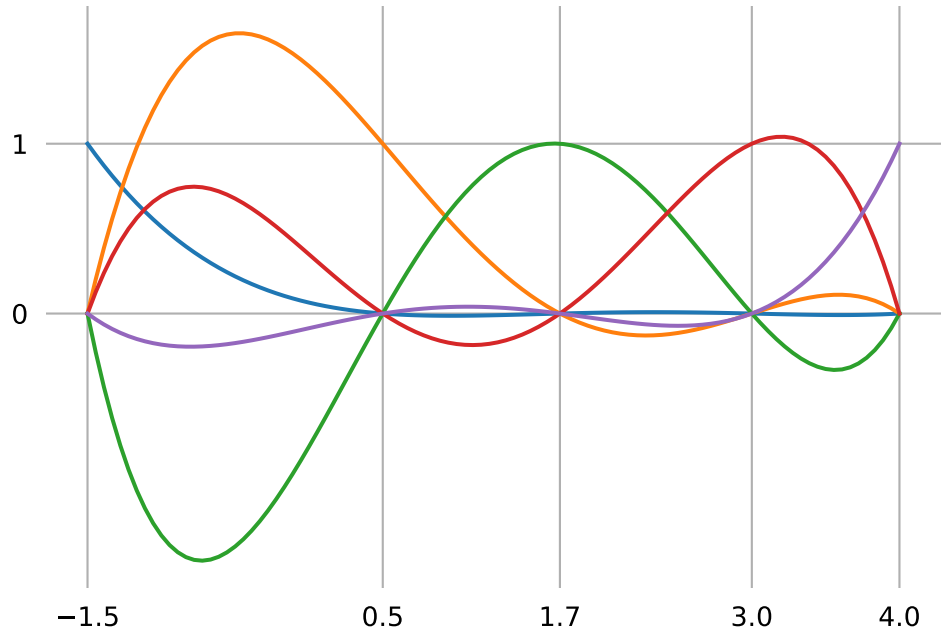
The above calculation can be easily generalized to be able to get any one of the set of polynomials defined by an arbitrary list of time instants:

[11]:
```
def lagrange_polynomial(times, i, t):
    """i-th Lagrange polynomial for the given time values, evaluated at t."""
    t = np.asarray(t)
    product = np.multiply.reduce
    return product([
        (t - times[j]) / (times[i] - times[j])
        for j in range(len(times))
        if i != j
    ])
```

Now we can calculate and visualize all 5 polynomials for our 5 given time instants:

[12]:
```
polys = np.column_stack([lagrange_polynomial(ts, i, plot_times)
                         for i in range(len(ts))])
```

[13]:
```
plt.plot(plot_times, polys)
grid_lines(ts, [0, 1])
```

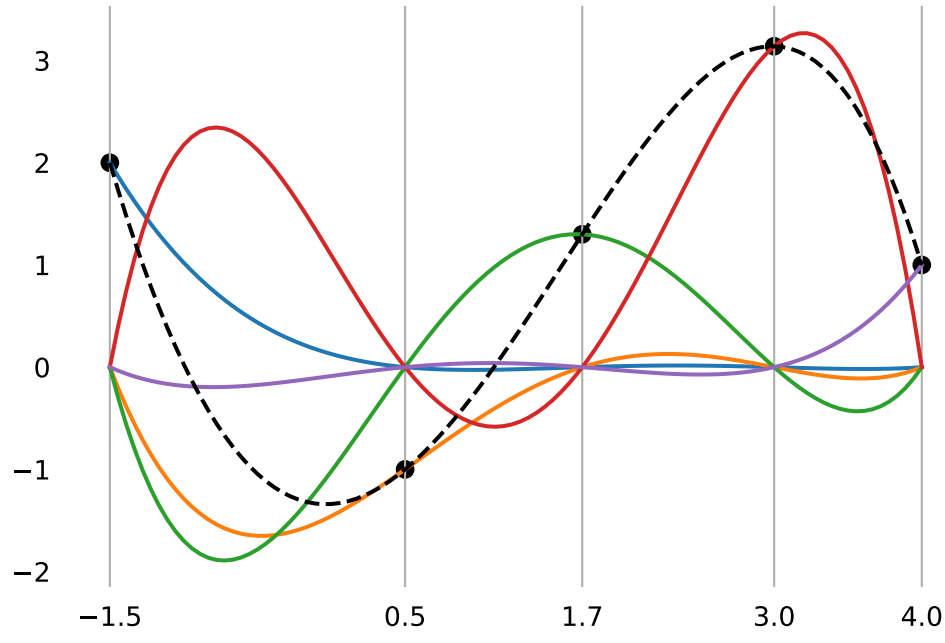


Finally, the interpolated values can be obtained by applying the given x_i values as weights to the polynomials and summing everything together:

[14]:
```
weighted_polys = polys * xs
```

[15]:
```
interpolated = np.sum(weighted_polys, axis=-1)
```

[16]:
```
plt.plot(plot_times, weighted_polys)
plt.plot(plot_times, interpolated, color='black', linestyle='dashed')
plt.scatter(ts, xs, color='black')
grid_lines(ts)
```



Neville's Algorithm

An alternative way to calculate interpolated values is Neville's algorithm[2] (see also [BG88], figure 2). We mention this algorithm mainly because it is referenced in the *derivation of non-uniform Catmull–Rom splines* (page 70) and the *description of the Barry–Goldman algorithm* (page 75).

As main building block, we need a linear interpolation between two values in a given time interval:

[17]:
```
def lerp(xs, ts, t):
    """Linear interpolation.

    Returns the interpolated value at time *t*,
    given the two values *xs* at times *ts*.

    """
    x_begin, x_end = xs
    t_begin, t_end = ts
    return (x_begin * (t_end - t) + x_end * (t - t_begin)) / (t_end - t_begin)
```

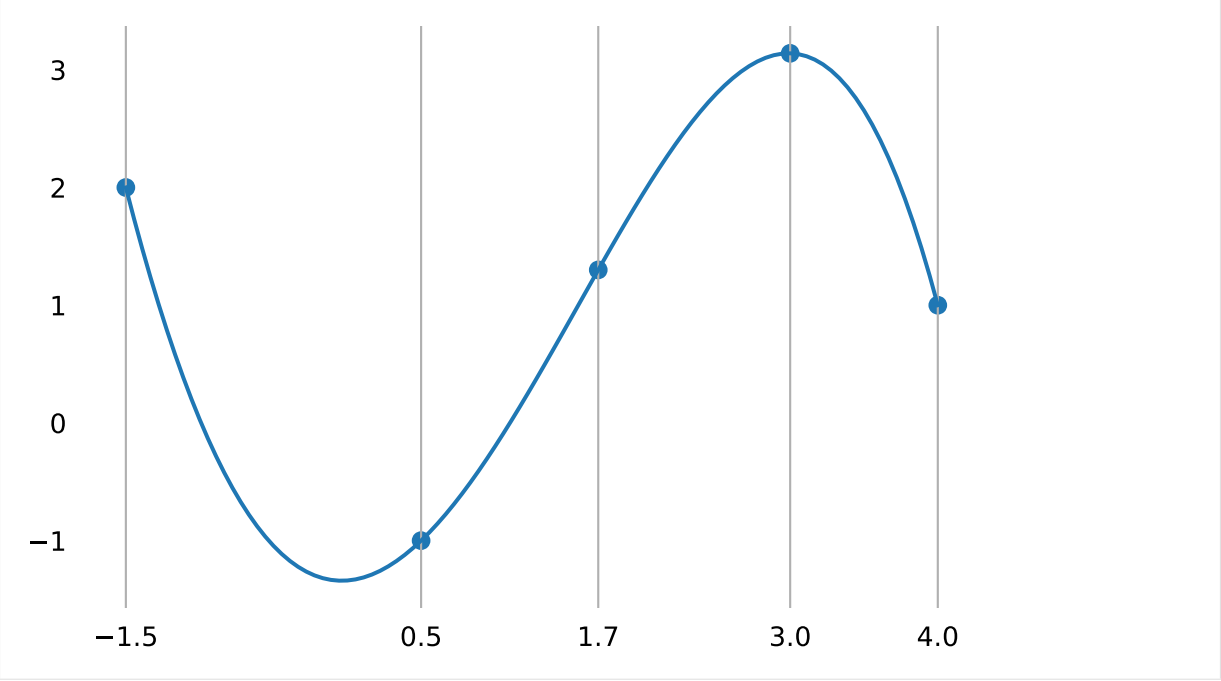
In each stage of the algorithm, linear interpolation is used to interpolate between adjacent values, leading to one fewer value than in the stage before. The new values are used as input to the next stage and so on. When there is only one value left, this value is the result.

The only tricky part is to choose the appropriate time interval for each interpolation. In the first stage, the intervals between the given time values are used. In the second stage, each time interval is combined with the following one, leading to one fewer time intervals in total. In the third stage, each time interval is combined with the following two intervals, and so on until the last stage, where all time intervals are combined into a single large interval.

[18]:
```
def neville(xs, ts, t):
    """Lagrange interpolation using Neville's algorithm.

    Returns the interpolated value at time(s) *t*,
    given the values *xs* at times *ts*.

    """
    assert len(xs) == len(ts)
    if not np.isscalar(t):
        return np.array([neville(xs, ts, time) for time in t])
    while len(xs) > 1:
        step = len(ts) - len(xs) + 1
        xs = [
            lerp(*args, t)
            for args in zip(zip(xs, xs[1:]), zip(ts, ts[step:]))]
    return xs[0]
```

[19]:
```
plt.plot(plot_times, neville(xs, ts, plot_times))
plt.scatter(ts, xs)
grid_lines(ts)
```

[2] https://en.wikipedia.org/wiki/Neville%27s_algorithm



Two-dimensional Example

Lagrange interpolation can of course also be used in higher-dimensional spaces. To show this, let's create a class:

[20]:
```
class Lagrange:

    def __init__(self, vertices, grid):
        assert len(vertices) == len(grid)
        self.vertices = np.array(vertices)
        self.grid = list(grid)

    def evaluate(self, t):
        # Alternatively, we could simply use this one-liner:
        #return neville(self.vertices, self.grid, t)
        if not np.isscalar(t):
            return np.array([self.evaluate(time) for time in t])
        polys = [lagrange_polynomial(self.grid, i, t)
                 for i in range(len(self.grid))]
        weighted_polys = self.vertices.T * polys
        return np.sum(weighted_polys, axis=-1)
```

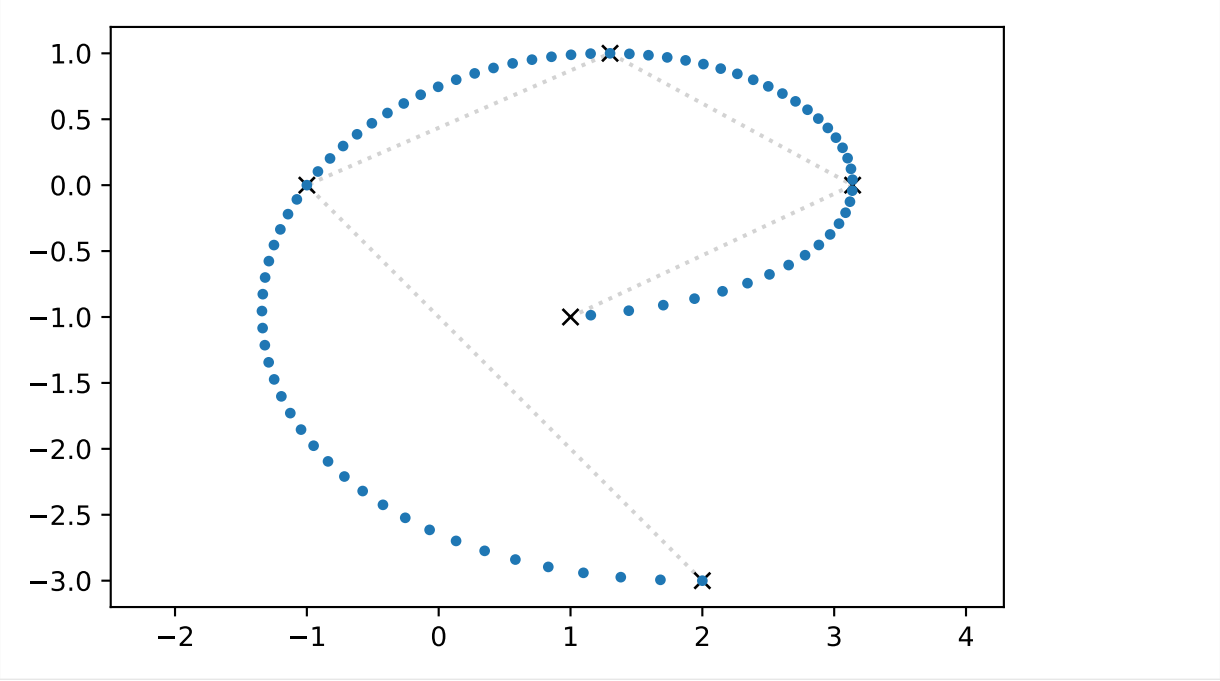
Since this class has the same interface as the splines that are discussed in the following sections, we can use a spline helper function from helper.py for plotting:

[21]:
```
from helper import plot_spline_2d
```

This time, we have a list of two-dimensional vectors and the same list of associated times as before:

[22]:
```
l1 = Lagrange([(2, -3), (-1, 0), (1.3, 1), (3.14, 0), (1, -1)], ts)
```

[23]:
```
plot_spline_2d(l1)
```

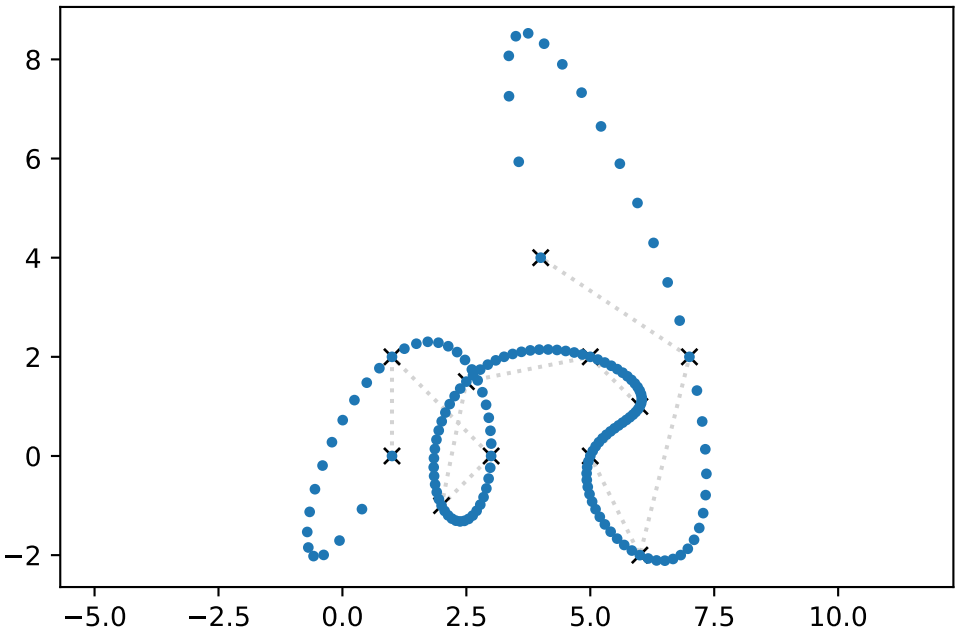


Runge's Phenomenon

This seems to work quite well, but as indicated above, Lagrange implementation has a severe limitation. This limitation gets more apparent when using more vertices, which leads to a higher-degree polynomial.

[24]:
```
vertices = [
    (1, 0),
    (1, 2),
    (3, 0),
    (2, -1),
    (2.5, 1.5),
    (5, 2),
    (6, 1),
    (5, 0),
    (6, -2),
    (7, 2),
    (4, 4),
]
times = range(len(vertices))
```

[25]:
```
l2 = Lagrange(vertices, times)
plot_spline_2d(l2)
```



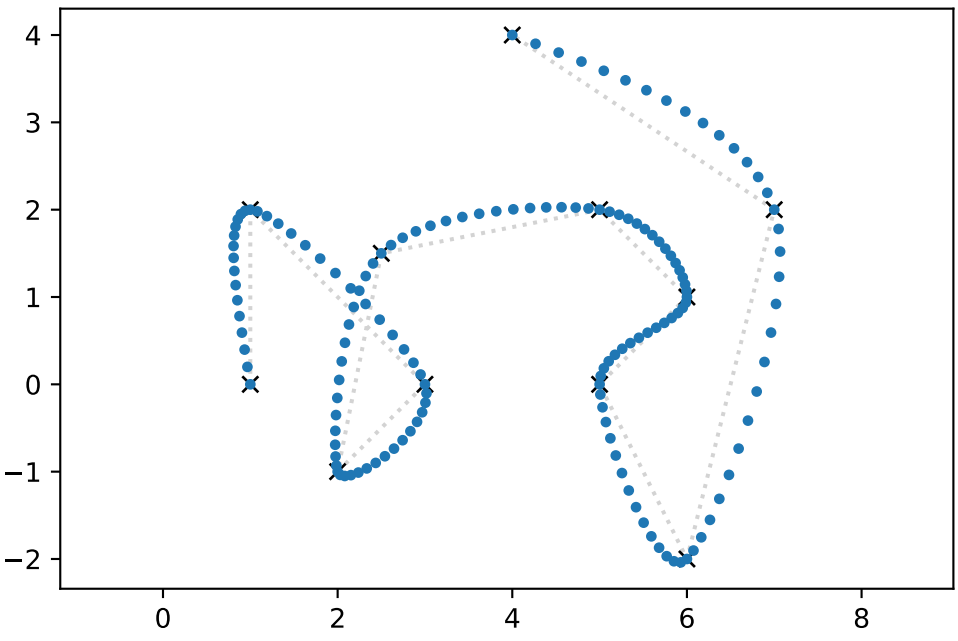
Here we see a severe overshooting effect, most pronounced at the beginning and the end of the curve. This effect is called Runge's phenomenon[3].

Long story short, Lagrange interpolation is typically not usable for drawing curves. For comparison, let's use the same positions and time values and create a *Catmull–Rom spline* (page 52):

[26]:
```
import splines
```

[27]:
```
cr_spline = splines.CatmullRom(vertices, times)
```

[28]:
```
plot_spline_2d(cr_spline)
```



This clearly doesn't have the overshooting problem we saw above.

[3] https://en.wikipedia.org/wiki/Runge's_phenomenon

Note

The *splines.CatmullRom* (page 150) class uses *"natural" end conditions* (page 98) by default.

doc/euclidean/lagrange.ipynb ends here.

1.3 Hermite Splines

Hermite splines[4] (named after Charles Hermite[5]) are the building blocks for many other types of interpolating polynomial splines, for example *natural splines* (page 30) and *Catmull–Rom splines* (page 52).

A Python implementation of (cubic) Hermite splines is available in the *splines.CubicHermite* (page 150) class.

The following section was generated from doc/euclidean/hermite-properties.ipynb

Properties of Hermite Splines

Hermite splines are interpolating polynomial splines, where for each polynomial segment, the desired value at the start and end is given (obviously!), as well as the values of a certain number of derivatives at the start and/or the end.

Most commonly, *cubic* (= degree 3) Hermite splines are used. Cubic polynomials have 4 coefficients to be chosen freely, and those are determined for each segment of a cubic Hermite spline by providing 4 pieces of information: the function value and the first derivative, both at the beginning and the end of the segment.

Other degrees of Hermite splines are possible (but much rarer), for example *quintic* (= degree 5) Hermite splines, which are defined by the second derivatives at the start and end of each segment, on top of the first derivatives and the function values (6 values in total).

Hermite splines with even degrees are probably still rarer. For example, *quadratic* (= degree 2) Hermite splines can be constructed by providing the function values at both beginning and end of each segment, but only one first derivative, either at the beginning or at the end (leading to 3 values in total). Make sure not to confuse them with *quartic* (= degree 4) Hermite splines, which are defined by 5 values per segment: function value and first derivative at both ends, and one of the second derivatives.

However, *cubic Hermite splines* are so overwhelmingly common that they are often simply referred to as *Hermite splines*.

From this point forward, we will only be considering *cubic* Hermite splines.

[1]:
```
import splines
```

[2]:
```
import matplotlib.pyplot as plt
import numpy as np
```

We import a few helper functions from helper.py:

[3]:
```
from helper import plot_slopes_1d, plot_spline_2d, plot_tangents_2d, grid_lines
```

Let's look at a one-dimensional spline first. We provide a list of values (to be interpolated) and a list of associated parameter values (or time instances, if you will).

[4]:
```
values = 2, 4, 3, 3
grid = 5, 7, 8, 10
```

[4] https://en.wikipedia.org/wiki/Cubic_Hermite_spline
[5] https://en.wikipedia.org/wiki/Charles_Hermite

Since (cubic) Hermite splines ask for the first derivative at the beginning and end of each segment, we provide a list of slopes (outgoing, incoming, outgoing, incoming, ...).

[5]:
```
slopes = 0, 0, -1, 0.5, 1, 3
```

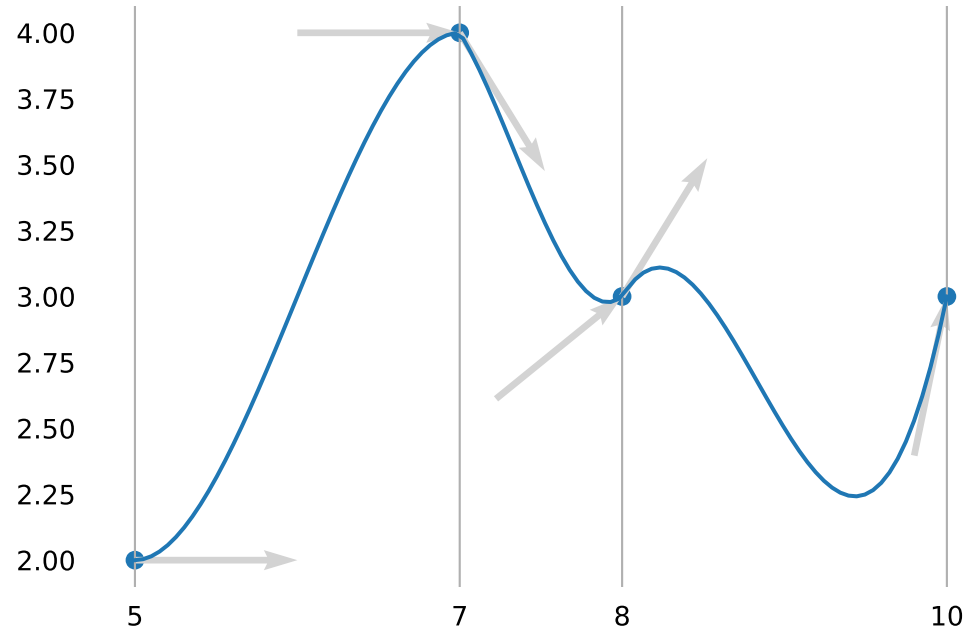
We are using the *splines.CubicHermite* (page 150) class to create the spline:

[6]:
```
s1 = splines.CubicHermite(values, slopes, grid=grid)
```

OK, let's plot this one-dimensional spline, together with the given values and slopes.

[7]:
```
times = np.linspace(grid[0], grid[-1], 100)
```

[8]:
```
plt.plot(times, s1.evaluate(times))
plt.scatter(grid, values)
plot_slopes_1d(slopes, values, grid)
grid_lines(grid)
```



Let's try a two-dimensional curve now (higher dimensions work similarly).

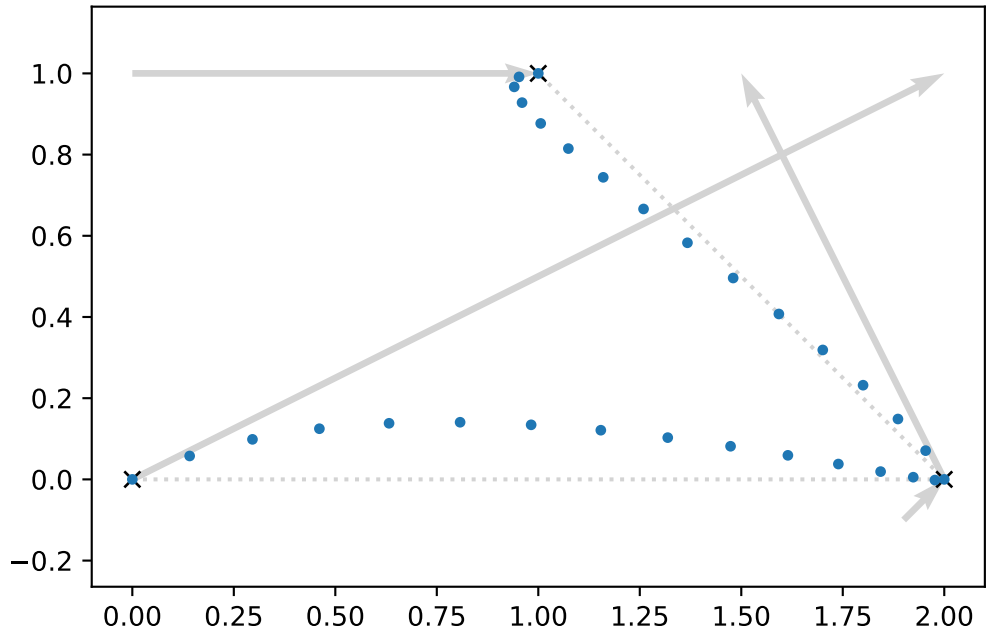
[9]:
```
vertices = [
    (0, 0),
    (2, 0),
    (1, 1),
]
```

The derivative of a curve is its tangent vector, so we provide a list of them (outgoing, incoming, outgoing, incoming, ...):

[10]:
```
tangents = [
    (2, 1),
    (0.1, 0.1),
    (-0.5, 1),
    (1, 0),
]
```

[11]:
```
s2 = splines.CubicHermite(vertices, tangents)
```

[12]:
```
plot_spline_2d(s2)
plot_tangents_2d(tangents, vertices)
```

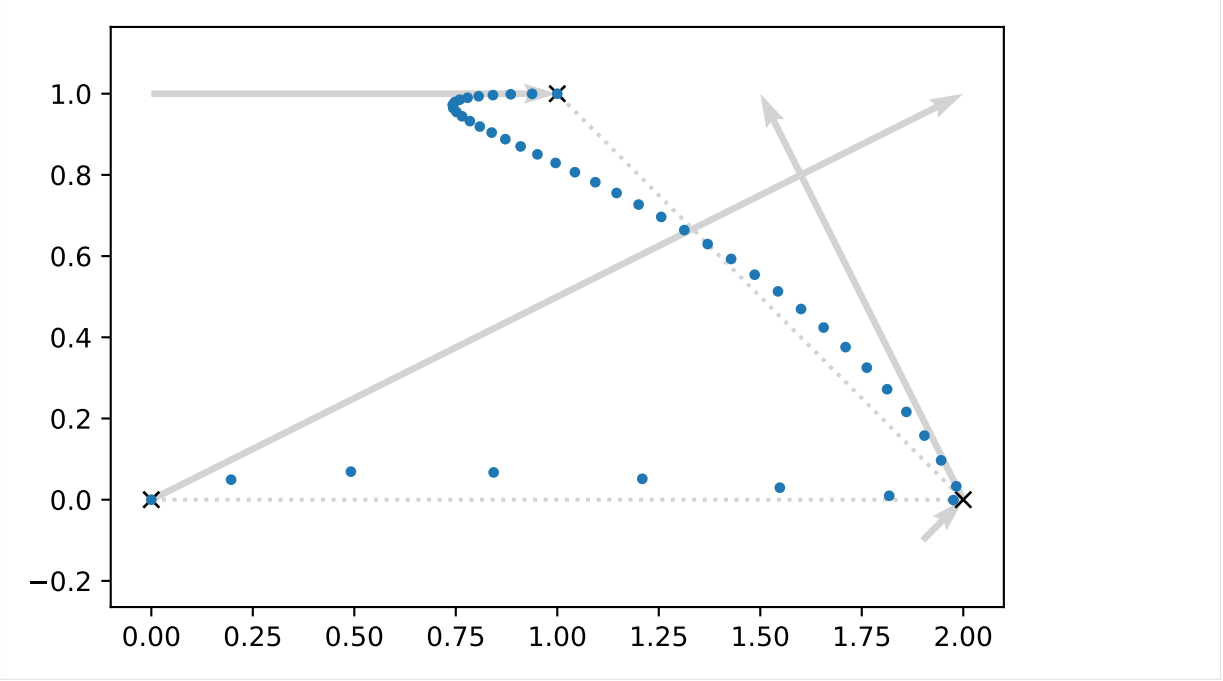


If no parameter values are given (by means of the `grid` argument), the *splines.CubicHermite* (page 150) class creates a *uniform* spline, i.e. all parameter intervals are automatically chosen to be 1. We can create a *non-uniform* spline by providing our own parameter values:

[13]:
```
grid = 0, 0.5, 3
```

Using the same vertices and tangents, we can clearly see how the new parameter values influence the shape and the speed of the curve (the dots are plotted at equal time intervals!):

[14]:
```
s3 = splines.CubicHermite(vertices, tangents, grid=grid)
plot_spline_2d(s3)
plot_tangents_2d(tangents, vertices)
```



Hermite splines are by default C^0 continuous. If adjacent tangents are chosen to point into the same direction, the spline becomes G^1 continuous.

If on top of having the same direction, adjacent tangents are chosen to have the same length, that makes the spline C^1 continuous. An example for that are *Catmull–Rom splines* (page 52). *Kochanek–Bartels splines* (page 84) can also be C^1 continuous, but only if their "continuity" parameter C is 0.

There is one unique choice of all of a cubic Hermite spline's tangents (given certain *end conditions* (page 98)) that leads to continuous second derivatives at all vertices, making the spline C^2 continuous. This is what *natural splines* (page 30) are all about.

doc/euclidean/hermite-properties.ipynb ends here.

The following section was generated from doc/euclidean/hermite-uniform.ipynb

Uniform Cubic Hermite Splines

We derive the basis matrix as well as the basis polynomials for cubic (= degree 3) Hermite splines. The derivations for other degrees is left as an exercise for the reader.

In this notebook, we consider *uniform* spline segments, i.e. the parameter in each segment varies from 0 to 1. The derivation for *non-uniform* cubic Hermite splines can be found in *a separate notebook* (page 24).

[1]:
```
import sympy as sp
sp.init_printing(order='rev-lex')
```

We load a few tools from utility.py:

[2]:
```
from utility import NamedExpression, NamedMatrix
```

[3]:
```
t = sp.symbols('t')
```

We are considering a single cubic polynomial segment of a Hermite spline (which is sometimes called a *Ferguson cubic*).

To simplify the indices in the following derivation, we are looking at the fifth polynomial segment $\boldsymbol{p}_4(t)$ from $\boldsymbol{x}_4$ to $\boldsymbol{x}_5$, where $0 \le t \le 1$. The results will be easily generalizable to an arbitrary polynomial segment $\boldsymbol{p}_i(t)$ from $\boldsymbol{x}_i$ to $\boldsymbol{x}_{i+1}$.

The polynomial has 4 coefficients, a_4 to d_4.

[4]:
```
coefficients = sp.Matrix(sp.symbols('a:dbm4')[::-1])
coefficients
```

[4]:
$$\begin{bmatrix} d_4 \\ c_4 \\ b_4 \\ a_4 \end{bmatrix}$$

Combined with the *monomial basis* ...

[5]:
```
b_monomial = sp.Matrix([t**3, t**2, t, 1]).T
b_monomial
```

[5]:
$$\begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}$$

... the coefficients form an expression for our polynomial segment $p_4(t)$:

[6]:
```
p4 = NamedExpression('pbm4', b_monomial.dot(coefficients))
p4
```

[6]:
$$p_4 = d_4 t^3 + c_4 t^2 + b_4 t + a_4$$

For more information about polynomials, see *Polynomial Parametric Curves* (page 2).

Let's also calculate the first derivative (a.k.a. velocity, a.k.a. tangent vector), while we are at it:

[7]:
```
pd4 = p4.diff(t)
pd4
```

[7]:
$$\frac{d}{dt} p_4 = 3d_4 t^2 + 2c_4 t + b_4$$

To generate a Hermite spline segment, we have to provide the value of the polynomial at the start and end point of the segment (at times $t = 0$ and $t = 1$, respectively). We also have to provide the first derivative at those same points.

$$\begin{aligned} x_4 &= p_4|_{t=0} \\ x_5 &= p_4|_{t=1} \\ \dot{x}_4 &= \frac{d}{dt} p_4 \Big|_{t=0} \\ \dot{x}_5 &= \frac{d}{dt} p_4 \Big|_{t=1} \end{aligned}$$

We call those 4 values the *control values* of the segment.

Evaluating the polynomial and its derivative at times 0 and 1 leads to 4 expressions for our 4 control values:

[8]:
```
x4 = p4.evaluated_at(t, 0).with_name('xbm4')
x5 = p4.evaluated_at(t, 1).with_name('xbm5')
xd4 = pd4.evaluated_at(t, 0).with_name('xdotbm4')
xd5 = pd4.evaluated_at(t, 1).with_name('xdotbm5')
```

[9]:
```
display(x4, x5, xd4, xd5)
```

$$x_4 = a_4$$

$$x_5 = d_4 + c_4 + b_4 + a_4$$

$$\dot{x}_4 = b_4$$
$$\dot{x}_5 = 3d_4 + 2c_4 + b_4$$

Given an input vector of control values ...

[10]:
```
control_values_H = NamedMatrix(sp.Matrix([x4.name,
                                          x5.name,
                                          xd4.name,
                                          xd5.name]))
control_values_H.name
```

[10]:
$$\begin{bmatrix} x_4 \\ x_5 \\ \dot{x}_4 \\ \dot{x}_5 \end{bmatrix}$$

... we want to find a way to transform those into the coefficients of our cubic polynomial.

[11]:
```
M_H = NamedMatrix(r'{M_\text{H}}', 4, 4)
```

[12]:
```
coefficients_H = NamedMatrix(coefficients, M_H.name * control_values_H.name)
coefficients_H
```

[12]:
$$\begin{bmatrix} d_4 \\ c_4 \\ b_4 \\ a_4 \end{bmatrix} = M_{\mathrm{H}} \begin{bmatrix} x_4 \\ x_5 \\ \dot{x}_4 \\ \dot{x}_5 \end{bmatrix}$$

This way, we can express our previously unknown coefficients in terms of the given control values.

However, in order to make it easy to determine the coefficients of the *basis matrix* M_H, we need the equation the other way around (by left-multiplying by the inverse):

[13]:
```
control_values_H.expr = M_H.name.I * coefficients
control_values_H
```

[13]:
$$\begin{bmatrix} x_4 \\ x_5 \\ \dot{x}_4 \\ \dot{x}_5 \end{bmatrix} = {M_{\mathrm{H}}}^{-1} \begin{bmatrix} d_4 \\ c_4 \\ b_4 \\ a_4 \end{bmatrix}$$

We can now insert the expressions for the control values that we obtained above ...

[14]:
```
substitutions = x4, x5, xd4, xd5
```

[15]:
```
control_values_H.subs_symbols(*substitutions)
```

[15]:
$$\begin{bmatrix} a_4 \\ d_4 + c_4 + b_4 + a_4 \\ b_4 \\ 3d_4 + 2c_4 + b_4 \end{bmatrix} = {M_{\mathrm{H}}}^{-1} \begin{bmatrix} d_4 \\ c_4 \\ b_4 \\ a_4 \end{bmatrix}$$

... and from this equation we can directly read off the matrix coefficients of ${M_H}^{-1}$:

[16]:
```
M_H.I = sp.Matrix(
    [[expr.coeff(cv) for cv in coefficients]
     for expr in control_values_H.subs_symbols(*substitutions).name])
M_H.I
```

[16]:
$$M_{\mathrm{H}}{}^{-1} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

The same thing for copy & paste purposes:

[17]:
```
print(_.expr)
```
```
Matrix([[0, 0, 0, 1], [1, 1, 1, 1], [0, 0, 1, 0], [3, 2, 1, 0]])
```

This transforms the coefficients of the polynomial into our control values, but we need it the other way round, which we can simply get by inverting the matrix:

[18]:
```
M_H
```
[18]:
$$M_{\mathrm{H}} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Again, for copy & paste:

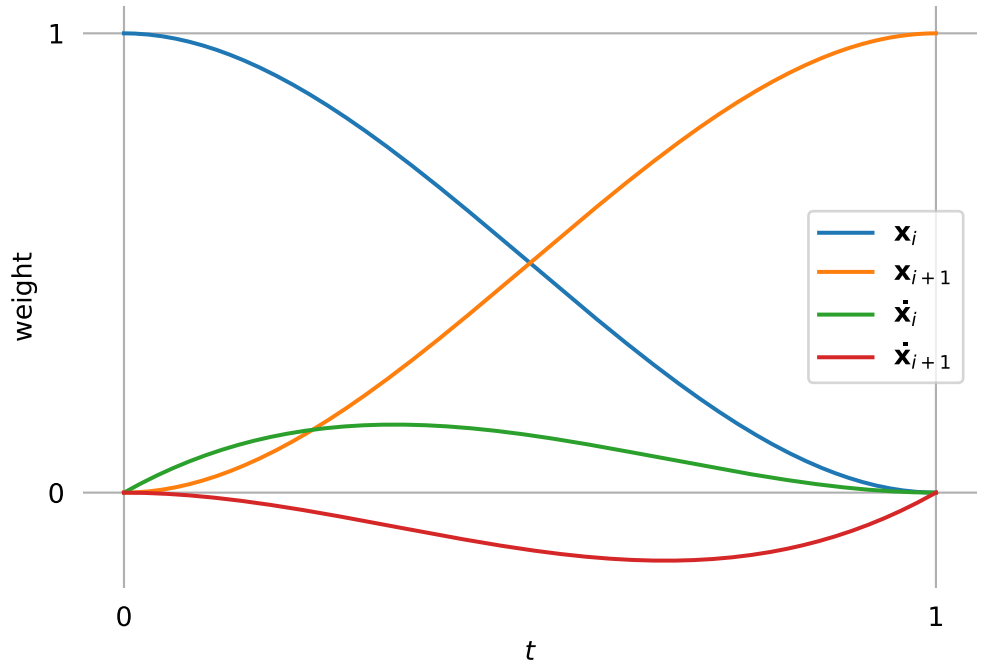
[19]:
```
print(_.expr)
```
```
Matrix([[2, -2, 1, 1], [-3, 3, -2, -1], [0, 0, 1, 0], [1, 0, 0, 0]])
```

Multiplying the monomial basis with this matrix yields the *Hermite basis polynomials*:

[20]:
```
b_H = NamedMatrix(r'{b_\text{H}}', b_monomial * M_H.expr)
b_H.factor().simplify().T
```
[20]:
$$b_{\mathrm{H}}{}^{T} = \begin{bmatrix} (-1+t)^2 (1+2t) \\ t^2 (3-2t) \\ t (-1+t)^2 \\ t^2 (-1+t) \end{bmatrix}$$

[21]:
```
from helper import plot_basis
```

[22]:
```
plot_basis(*b_H.expr, labels=sp.symbols('xbm_i xbm_i+1 xdotbm_i xdotbm_i+1'))
```



Note that the basis function associated with x_i has the value 1 at the beginning, while all others are 0 at that point. For this reason, the linear combination of all basis functions at $t = 0$ simply adds up to the value x_i (which is exactly what we wanted to happen!).

Similarly, the basis function associated with $\dot{x}_i$ has a first derivative of $+1$ at the beginning, while all others have a first derivative of 0. Therefore, the linear combination of all basis functions at $t = 0$ turns out to have a first derivative of $\dot{x}_i$ (what a coincidence!).

While t progresses towards 1, both functions must relinquish their influence to the other two basis functions.

At the end (when $t = 1$), the basis function associated with x_{i+1} is the only one that has a non-zero value. More concretely, it has the value 1. Finally, the basis function associated with $\dot{x}_{i+1}$ is the only one with a non-zero first derivative. In fact, it has a first derivative of exactly $+1$ (the function values leading up to that have to be negative because the final function value has to be 0).

This can be summarized by:

[23]:
```
sp.Matrix([[
    b.subs(t, 0),
    b.subs(t, 1),
    b.diff(t).subs(t, 0),
    b.diff(t).subs(t, 1),
] for b in b_H.expr])
```

[23]:
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To quickly check whether the matrix M_H does what we expect, let's plot an example segment:

[24]:
```
import numpy as np
```

If we use the same API as for the other splines, we can reuse the helper functions for plotting from helper.py.

[25]:
```
from helper import plot_spline_2d, plot_tangents_2d
```

[26]:
```
class UniformHermiteSegment:

    grid = 0, 1

    def __init__(self, control_values):
        self.coeffs = sp.lambdify([], M_H.expr)() @ control_values

    def evaluate(self, t):
        t = np.expand_dims(t, -1)
        return t**[3, 2, 1, 0] @ self.coeffs
```

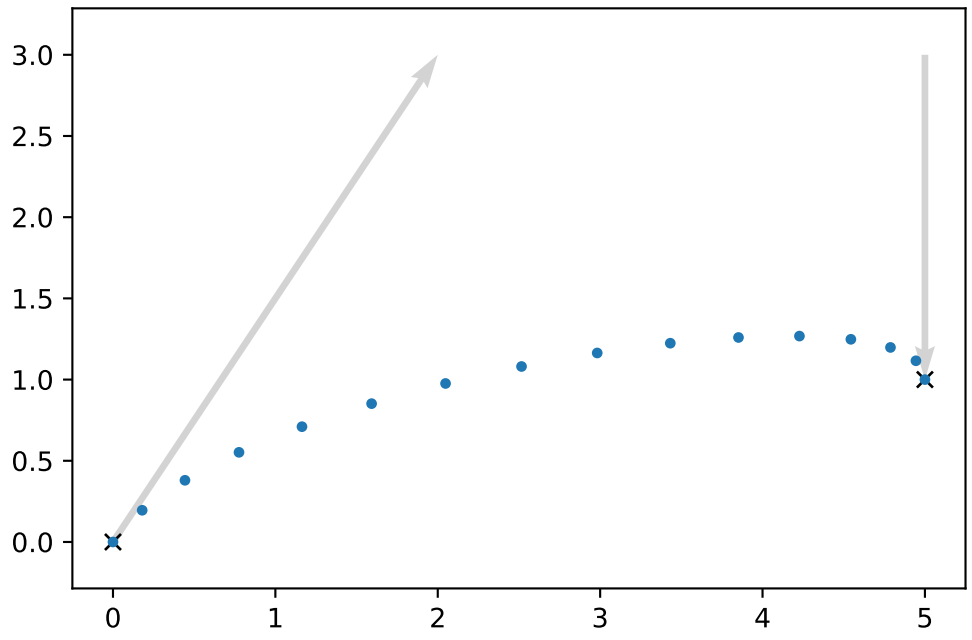
Note

The `@` operator is used here to do NumPy's matrix multiplication[6].

[27]:
```
vertices = [0, 0], [5, 1]
tangents = [2, 3], [0, -2]
```

[28]:
```
s = UniformHermiteSegment([*vertices, *tangents])
```

[29]:
```
plot_spline_2d(s, chords=False)
plot_tangents_2d(tangents, vertices)
```



[6] https://numpy.org/doc/stable/reference/generated/numpy.matmul.html

Relation to Bézier Splines

Above, we were using two positions (start and end) and two tangent vectors (at those same two positions) as control values:

[30]:
```
control_values_H.name
```

[30]:
$$\begin{bmatrix} x_4 \\ x_5 \\ \dot{x}_4 \\ \dot{x}_5 \end{bmatrix}$$

What about using four positions (and no tangent vectors) instead?

Let's use the point $\tilde{x}_4$ as a "handle" (connected to x_4) that controls the tangent vector. Same for $\tilde{x}_5$ (connected to x_5).

And since the tangents looked unwieldily long in the plot above (compared to the effect they have on the shape of the curve), let's put the handles only at a third of the length of the tangents, shall we?

$$\tilde{x}_4 = x_4 + \frac{\dot{x}_4}{3}$$
$$\tilde{x}_5 = x_5 - \frac{\dot{x}_5}{3}$$

[31]:
```
control_values_B = NamedMatrix(sp.Matrix([
    x4.name,
    sp.Symbol('xtildebm4'),
    sp.Symbol('xtildebm5'),
    x5.name,
]), sp.Matrix([
    x4.name,
    x4.name + xd4.name / 3,
    x5.name - xd5.name / 3,
    x5.name,
]))
control_values_B
```

[31]:
$$\begin{bmatrix} x_4 \\ \tilde{x}_4 \\ \tilde{x}_5 \\ x_5 \end{bmatrix} = \begin{bmatrix} x_4 \\ \frac{\dot{x}_4}{3} + x_4 \\ -\frac{\dot{x}_5}{3} + x_5 \\ x_5 \end{bmatrix}$$

Now let's try to come up with a matrix that transforms our good old Hermite control values into our new control points.

[32]:
```
M_HtoB = NamedMatrix(r'{M_\text{H$\to$B}}', 4, 4)
```

[33]:
```
NamedMatrix(control_values_B.name, M_HtoB.name * control_values_H.name)
```

[33]:
$$\begin{bmatrix} x_4 \\ \tilde{x}_4 \\ \tilde{x}_5 \\ x_5 \end{bmatrix} = M_{\mathrm{H}\to\mathrm{B}} \begin{bmatrix} x_4 \\ x_5 \\ \dot{x}_4 \\ \dot{x}_5 \end{bmatrix}$$

We can immediately read the matrix coefficients off the previous expression.

[34]:
```
M_HtoB.expr = sp.Matrix([[expr.coeff(cv) for cv in control_values_H.name]
                         for expr in control_values_B.expr])
M_HtoB.pull_out(sp.S.One / 3)
```

[34]:
$$M_{\mathrm{H}\to\mathrm{B}} = \frac{1}{3}\begin{bmatrix} 3 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 0 & 3 & 0 & -1 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

[35]:
```
print(_.expr)
```

```
(1/3)*Matrix([
[3, 0, 0,  0],
[3, 0, 1,  0],
[0, 3, 0, -1],
[0, 3, 0,  0]])
```

The inverse of this matrix transforms our new control points into Hermite control values:

[36]:
```
M_BtoH = NamedMatrix(r'{M_\text{B$\to$H}}', M_HtoB.I.expr)
M_BtoH
```

[36]:
$$M_{\mathrm{B}\to\mathrm{H}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix}$$

[37]:
```
print(_.expr)
```

```
Matrix([[1, 0, 0, 0], [0, 0, 0, 1], [-3, 3, 0, 0], [0, 0, -3, 3]])
```

When we combine M_H with this new matrix, we get a matrix which leads us to a new set of basis polynomials associated with the 4 control points.

[38]:
```
M_B = NamedMatrix(r'{M_\text{B}}', M_H.name * M_BtoH.name)
M_B
```

[38]:
$$M_{\mathrm{B}} = M_{\mathrm{H}} M_{\mathrm{B}\to\mathrm{H}}$$

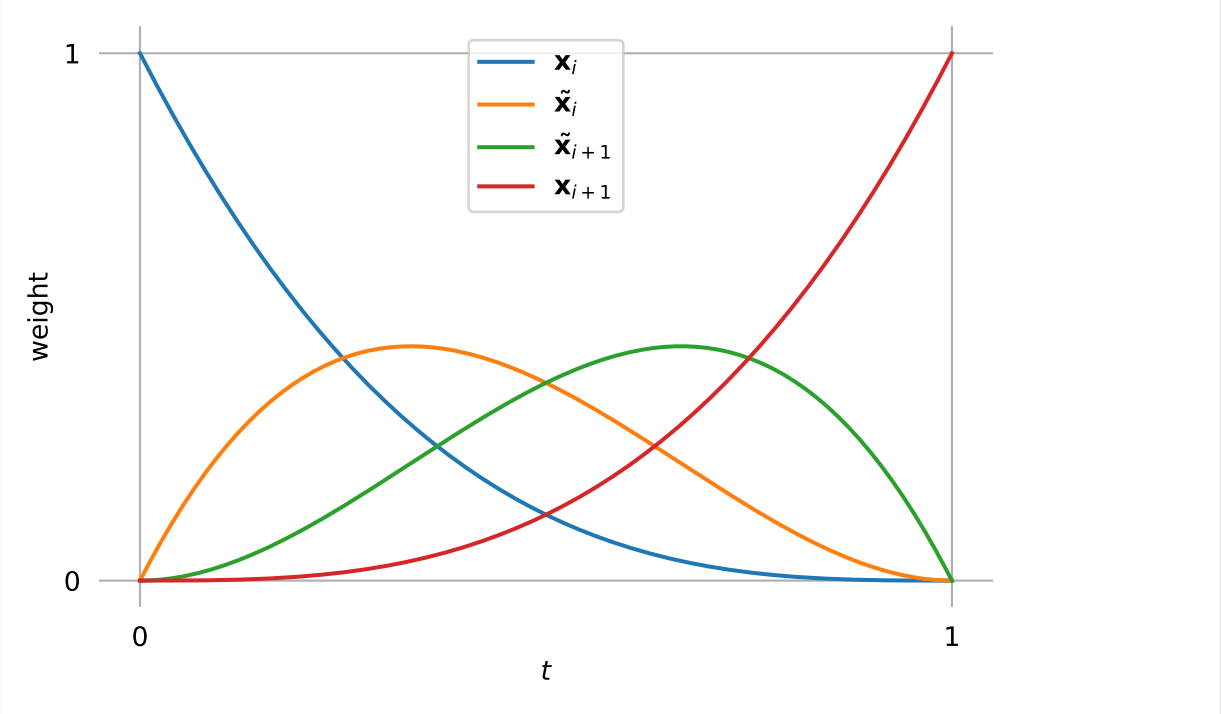
[39]:
```
M_B = M_B.subs_symbols(M_H, M_BtoH).doit()
M_B
```

[39]:
$$M_{\mathrm{B}} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

[40]:
```
b_B = NamedMatrix(r'{b_\text{B}}', b_monomial * M_B.expr)
b_B.T
```

[40]:
$${b_{\mathrm{B}}}^T = \begin{bmatrix} 1 - 3t + 3t^2 - t^3 \\ 3t - 6t^2 + 3t^3 \\ 3t^2 - 3t^3 \\ t^3 \end{bmatrix}$$

[41]:
```
plot_basis(
    *b_B.expr,
    labels=sp.symbols('xbm_i xtildebm_i xtildebm_i+1 xbm_i+1'))
```



Those happen to be the cubic *Bernstein* polynomials and it turns out that we just invented *Bézier* curves! See *the section about Bézier/Bernstein splines* (page 38) for more information about them.

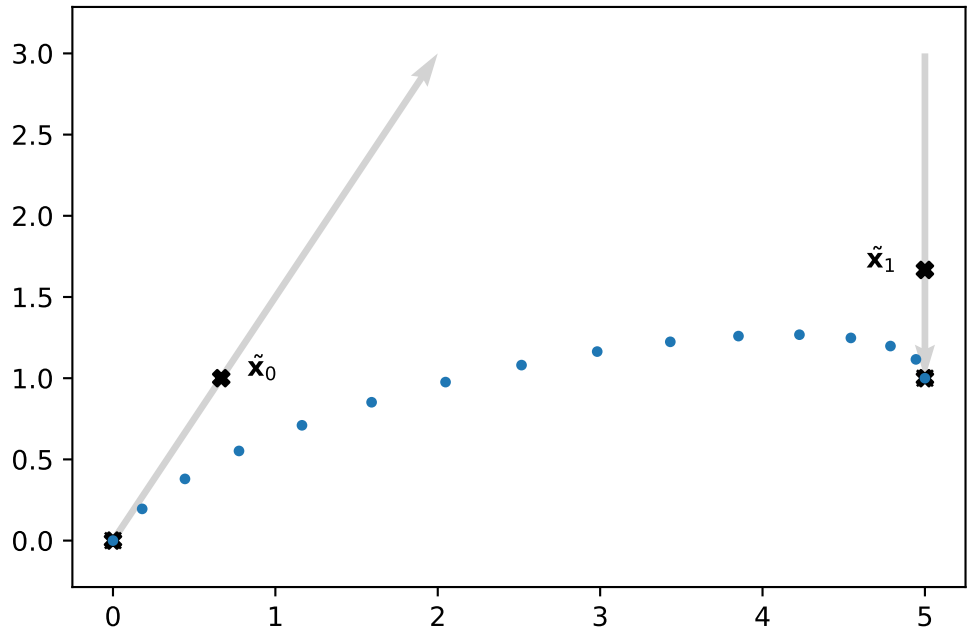
We chose the additional control points to be located at $\frac{1}{3}$ of the tangent vector. Let's quickly visualize this using the example from above and $M_{\mathrm{H}\to\mathrm{B}}$:

[42]:
```
points = sp.lambdify([], M_HtoB.expr)() @ [*vertices, *tangents]
```

[43]:
```
import matplotlib.pyplot as plt
```

[44]:
```
plot_spline_2d(s, chords=False)
plot_tangents_2d(tangents, vertices)
plt.scatter(*points.T, marker='X', color='black')
plt.annotate(r'$\quad\tilde{\bf{x}}_0$', points[1])
plt.annotate(r'$\tilde{\bf{x}}_1\quad$', points[2], ha='right');
```

[44]:
```
Text(5.0, 1.6666666666666665, '$\\tilde{\\bf{x}}_1\\quad$')
```



doc/euclidean/hermite-uniform.ipynb ends here.

The following section was generated from doc/euclidean/hermite-non-uniform.ipynb

Non-Uniform Cubic Hermite Splines

We have already derived *uniform cubic Hermite splines* (page 15), where the parameter t ranges from 0 to 1.

When we want to use *non-uniform* cubic Hermite splines, and therefore arbitrary ranges from t_i to t_{i+1}, we have (at least) two possibilities:

- Do the same derivations as in the *uniform* case, except when we previously evaluated an expression at the parameter value $t = 0$, we now evaluate it at the value $t = t_i$. Of course we do the same with $t = 1 \to t = t_{i+1}$.
- Re-scale the *non-uniform* parameter using $t \to \frac{t - t_i}{t_{i+1} - t_i}$ (which makes the new parameter go from 0 to 1) and then simply use the results from the *uniform* case.

The first approach leads to more complicated expressions in the basis matrix and the basis polynomials, but it has the advantage that the parameter value doesn't have to be re-scaled each time when evaluating the spline for a given parameter (which *might* be slightly more efficient).

The second approach has the problem that it doesn't actually work correctly, but we will see that we can make a slight adjustment to fix that problem (spoiler alert: we will have to multiply the tangent vectors by Δ_i).

We show the second approach here, but the first approach can be done very similarly, with only very few changed steps. The appropriate changes are mentioned below.

---

[1]:
```
import sympy as sp
sp.init_printing(order='rev-lex')
```

[2]:
```
from utility import NamedExpression, NamedMatrix
```

To simplify the indices in the following derivation, we are looking at the fifth polynomial segment $\boldsymbol{p}_4(t)$ from $\boldsymbol{x}_4$ to $\boldsymbol{x}_5$, where $t_4 \leq t \leq t_5$. The results will be easily generalizable to an arbitrary polynomial segment $\boldsymbol{p}_i(t)$ from $\boldsymbol{x}_i$ to $\boldsymbol{x}_{i+1}$.

[3]:
```
t, t4, t5 = sp.symbols('t t4:6')
```

[4]:
```
coefficients = sp.Matrix(sp.symbols('a:dbm4')[::-1])
b_monomial = sp.Matrix([t**3, t**2, t, 1]).T
b_monomial.dot(coefficients)
```

[4]:
$$\boldsymbol{d}_4 t^3 + \boldsymbol{c}_4 t^2 + \boldsymbol{b}_4 t + \boldsymbol{a}_4$$

We use the humble cubic polynomial (with monomial basis) to represent our curve segment $\boldsymbol{p}_4(t)$, but we re-scale the parameter to map $t_4 \to 0$ and $t_5 \to 1$:

[5]:
```
p4 = NamedExpression('pbm4', _.subs(t, (t - t4) / (t5 - t4)))
```

If you don't want to do the re-scaling, simply un-comment the next line!

---

[6]:
```
#p4 = NamedExpression('pbm4', b_monomial.dot(coefficients))
```

Either way, this is our polynomial segment ...

[7]:
```
p4
```

[7]:
$$\boldsymbol{p}_4 = \frac{\boldsymbol{d}_4 \left(-t_4 + t\right)^3}{\left(t_5 - t_4\right)^3} + \frac{\boldsymbol{c}_4 \left(-t_4 + t\right)^2}{\left(t_5 - t_4\right)^2} + \frac{\boldsymbol{b}_4 \left(-t_4 + t\right)}{t_5 - t_4} + \boldsymbol{a}_4$$

... and it's derivative/velocity/tangent vectors:

[8]:
```
pd4 = p4.diff(t)
pd4
```

[8]:
$$\frac{d}{dt}\boldsymbol{p}_4 = \frac{3\boldsymbol{d}_4 \left(-t_4 + t\right)^2}{\left(t_5 - t_4\right)^3} + \frac{\boldsymbol{c}_4 \left(-2t_4 + 2t\right)}{\left(t_5 - t_4\right)^2} + \frac{\boldsymbol{b}_4}{t_5 - t_4}$$

The next steps are very similar to what we did in the *uniform case* (page 15), except that we use t_4 and t_5 instead of 0 and 1, respectively.

[9]:
```
x4 = p4.evaluated_at(t, t4).with_name('xbm4')
x5 = p4.evaluated_at(t, t5).with_name('xbm5')
xd4 = pd4.evaluated_at(t, t4).with_name('xdotbm4')
xd5 = pd4.evaluated_at(t, t5).factor().with_name('xdotbm5')
```

To simplify things, we define a new symbol $\Delta_4 = t_5 - t_4$, representing the duration of the current segment. However, we only use this for simplifying the display, further calculations are still carried out with t_i.

[10]:
```
delta = {
    t5 - t4: sp.Symbol('Delta4'),
}
```

[11]:
```
display(x4, x5, xd4.subs(delta), xd5.subs(delta))
```

$$\boldsymbol{x}_4 = \boldsymbol{a}_4$$

$$\boldsymbol{x}_5 = \boldsymbol{d}_4 + \boldsymbol{c}_4 + \boldsymbol{b}_4 + \boldsymbol{a}_4$$

$$\dot{x}_4 = \frac{b_4}{\Delta_4}$$

$$\dot{x}_5 = \frac{3d_4 + 2c_4 + b_4}{\Delta_4}$$

[12]:
```
M_H = NamedMatrix(r'{M_{\text{H},4}}', 4, 4)
```

[13]:
```
control_values_H = NamedMatrix(
    sp.Matrix([x4.name, x5.name, xd4.name, xd5.name]),
    M_H.name.I * coefficients)
control_values_H
```

[13]:
$$\begin{bmatrix} x_4 \\ x_5 \\ \dot{x}_4 \\ \dot{x}_5 \end{bmatrix} = {M_{\mathrm{H},4}}^{-1} \begin{bmatrix} d_4 \\ c_4 \\ b_4 \\ a_4 \end{bmatrix}$$

[14]:
```
substitutions = x4, x5, xd4, xd5
```

[15]:
```
control_values_H.subs_symbols(*substitutions).subs(delta)
```

[15]:
$$\begin{bmatrix} a_4 \\ d_4 + c_4 + b_4 + a_4 \\ \frac{b_4}{\Delta_4} \\ \frac{3d_4 + 2c_4 + b_4}{\Delta_4} \end{bmatrix} = {M_{\mathrm{H},4}}^{-1} \begin{bmatrix} d_4 \\ c_4 \\ b_4 \\ a_4 \end{bmatrix}$$

[16]:
```
M_H.I = sp.Matrix([
    [expr.expand().coeff(c) for c in coefficients]
    for expr in control_values_H.subs_symbols(*substitutions).name])
M_H.I.subs(delta)
```

[16]:
$${M_{\mathrm{H},4}}^{-1} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & \frac{1}{\Delta_4} & 0 \\ \frac{3}{\Delta_4} & \frac{2}{\Delta_4} & \frac{1}{\Delta_4} & 0 \end{bmatrix}$$

[17]:
```
print(_.expr)
```

```
Matrix([[0, 0, 0, 1], [1, 1, 1, 1], [0, 0, 1/Delta4, 0], [3/Delta4, 2/Delta4, 1/Delta4,␣
↪0]])
```

[18]:
```
M_H.factor().subs(delta)
```

[18]:
$$M_{\mathrm{H},4} = \begin{bmatrix} 2 & -2 & \Delta_4 & \Delta_4 \\ -3 & 3 & -2\Delta_4 & -\Delta_4 \\ 0 & 0 & \Delta_4 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

[19]:
```
print(_.expr)
```

```
Matrix([[2, -2, Delta4, Delta4], [-3, 3, -2*Delta4, -Delta4], [0, 0, Delta4, 0], [1, 0, 0,␣
↪0]])
```

[20]:
```
b_H = NamedMatrix(r'{b_{\text{H},4}}', b_monomial * M_H.expr)
b_H.factor().subs(delta).simplify().T
```

[20]:
$$b_{\mathrm{H},4}{}^{T} = \begin{bmatrix} (-1+t)^2(1+2t) \\ t^2(3-2t) \\ \Delta_4 t(-1+t)^2 \\ \Delta_4 t^2(-1+t) \end{bmatrix}$$

Those are the *non-uniform* Hermite basis functions. Not surprisingly, they are different for each segment, because generally the values Δ_i are different in the non-uniform case.

To quickly check whether the matrix $M_{H,4}$ does what we expect, let's plot an example segment:

[21]:
```
import numpy as np
```

If we use the same API as for the other splines, we can reuse the helper functions for plotting from helper.py:

[22]:
```
from helper import plot_spline_2d, plot_tangents_2d
```

The following code re-scales the parameter with `t = (t - begin) / (end - begin)`. If you did *not* re-scale t in the derivation above, you'll have to remove this line.

---

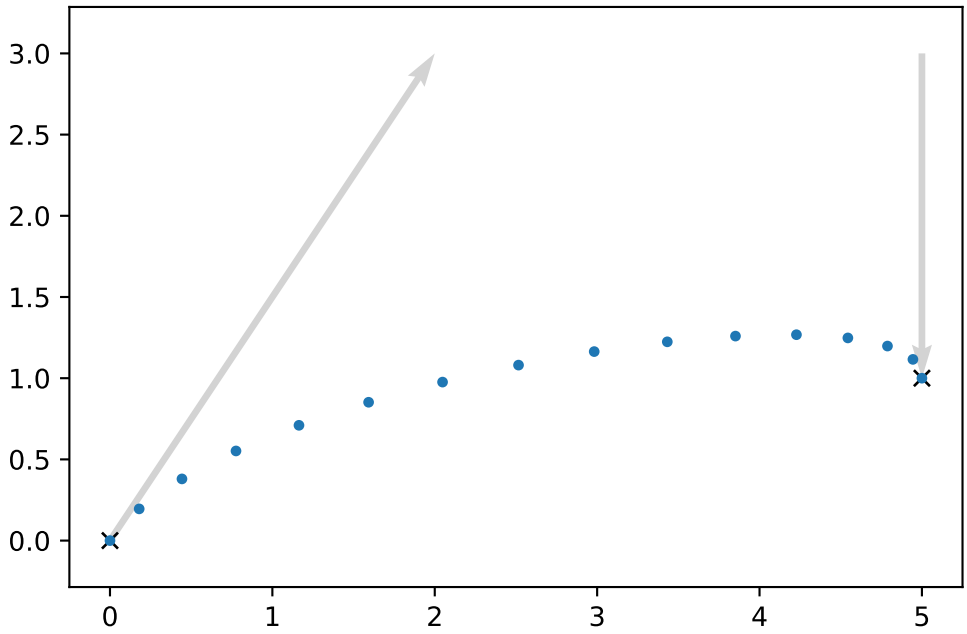
[23]:
```
class HermiteSegment:

    def __init__(self, control_values, begin, end):
        array = sp.lambdify([t4, t5], M_H.expr)(begin, end)
        self.coeffs = array @ control_values
        self.grid = begin, end

    def evaluate(self, t):
        t = np.expand_dims(t, -1)
        begin, end = self.grid
        # If you derived M_H without re-scaling t, remove the following line:
        t = (t - begin) / (end - begin)
        return t**[3, 2, 1, 0] @ self.coeffs
```

[24]:
```
vertices = [0, 0], [5, 1]
tangents = [2, 3], [0, -2]
```

We can simulate the *uniform* case by specifying a parameter range from 0 to 1:

[25]:
```
s1 = HermiteSegment([*vertices, *tangents], 0, 1)
```

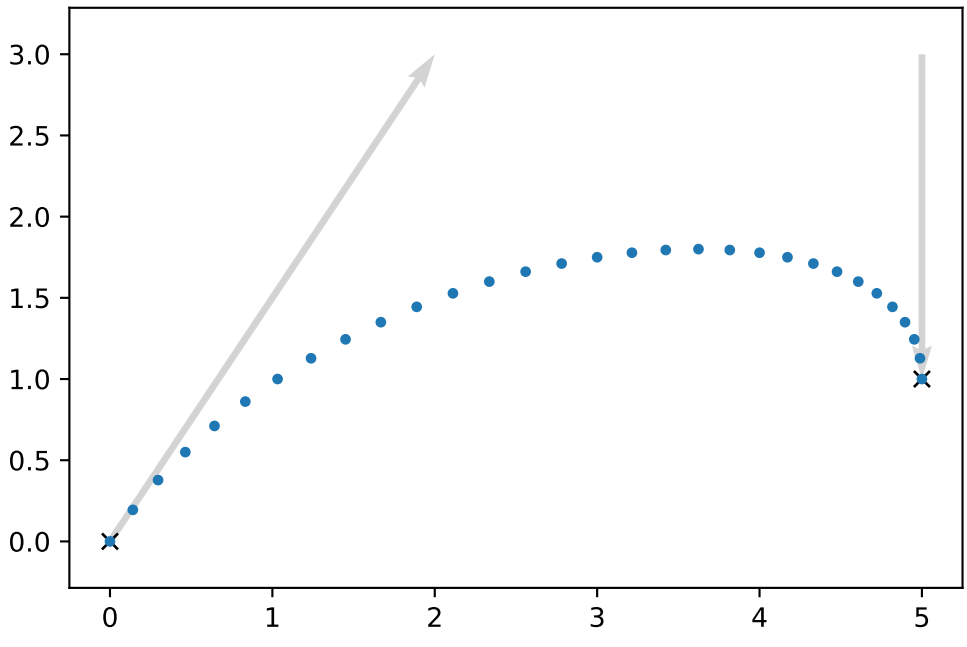
[26]:
```
plot_spline_2d(s1, chords=False)
plot_tangents_2d(tangents, vertices)
```



But other ranges should work as well:

[27]:
```
s2 = HermiteSegment([*vertices, *tangents], 3, 5)
```

[28]:
```
plot_spline_2d(s2, chords=False)
plot_tangents_2d(tangents, vertices)
```



If you did *not* re-scale t in the beginning of the derivation, you can use the matrix $M_{H,i}$ to calculate the monomial coefficients of each segment (as shown in the example code above) and be done with it. The following simplification does only apply if you *did* re-scale t.

If you *did* re-scale t, the basis matrix and the basis polynomials will look very similar to the *uniform case* (page 15), but they are not quite the same. This means that simply re-scaling the parameter is not

enough to correctly use the *uniform* results for implementing *non-uniform* Hermite splines.

However, we can see that the only difference is that the components associated with $\dot{x}_4$ and $\dot{x}_5$ are simply multiplied by Δ_4. That means if we re-scale the parameter *and* multiply the given tangent vectors by Δ_i, we can indeed use the *uniform* workflow.

Just to make sure we are actually telling the truth, let's check that the control values with scaled tangent vectors ...

[29]:
```
control_values_H_scaled = sp.Matrix([x4.name,
                                     x5.name,
                                     (t5 - t4) * xd4.name,
                                     (t5 - t4) * xd5.name])
control_values_H_scaled.subs(delta)
```

[29]:
$$\begin{bmatrix} x_4 \\ x_5 \\ \Delta_4 \dot{x}_4 \\ \Delta_4 \dot{x}_5 \end{bmatrix}$$

... really lead to the same result as when using the *uniform* basis matrix:

[30]:
```
sp.simplify(sp.Eq(
    M_H.expr * control_values_H.name,
    sp.Matrix([[2, -2, 1, 1],
               [-3, 3, -2, -1],
               [0, 0, 1, 0],
               [1, 0, 0, 0]]) * control_values_H_scaled))
```

[30]: True

The following line will fail if you did *not* rescale t:

---

[31]:
```
assert _ == True
```

Long story short, to implement a *non-uniform* cubic Hermite spline segment, we can simply re-scale the parameter to a range from 0 to 1 (by substituting $t \to \frac{t-t_i}{t_{i+1}-t_i}$), multiply both given tangent vectors by $\Delta_i = t_{i+1} - t_i$ and then simply use the implementation of the *uniform* cubic Hermite spline segment.

Another way of looking at this is to consider the *uniform* polynomial segment $\boldsymbol{u}_i(t)$ and its tangent vector (i.e. first derivative) $\boldsymbol{u}_i'(t)$. If we want to know the tangent vector after substituting $t \to \frac{t-t_i}{\Delta_i}$, we have to use the chain rule[7] (with the inner derivative being $\frac{1}{\Delta_i}$):

$$\frac{d}{dt}\boldsymbol{u}_i\left(\frac{t-t_i}{\Delta_i}\right) = \frac{1}{\Delta_i}\boldsymbol{u}_i'\left(\frac{t-t_i}{\Delta_i}\right).$$

This means the tangent vectors have been shrunk by Δ_i! If we want to maintain the original lengths of our tangent vectors, we can simply scale them by Δ_i beforehand.

doc/euclidean/hermite-non-uniform.ipynb ends here.

[7] https://en.wikipedia.org/wiki/Chain_rule

1.4 Natural Splines

TODO

Python implementation: *splines.Natural* (page 151)

https://en.wikipedia.org/wiki/Spline_interpolation

https://en.wikiversity.org/wiki/Cubic_Spline_Interpolation

https://en.wiktionary.org/wiki/spline

https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.CubicSpline.html

The following section was generated from `doc/euclidean/natural-properties.ipynb`

Properties of Natural Splines

[1]:
```
import splines
```

[2]:
```
vertices = [
    (0, 0),
    (1, 0),
    (2, 1),
    (3, 1),
]
```

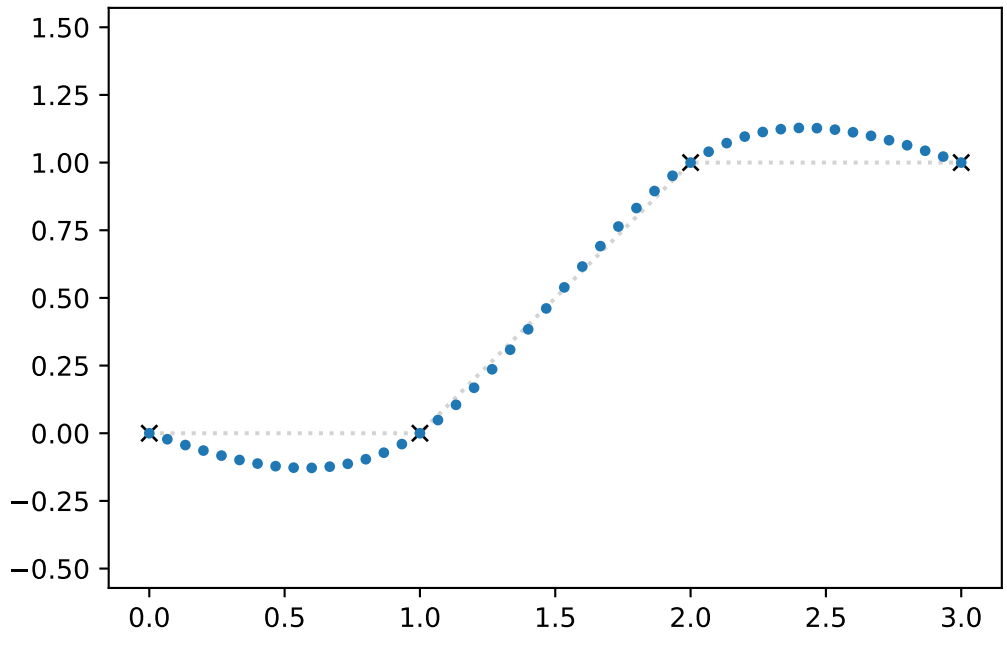
splines.Natural (page 151)

[3]:
```
s = splines.Natural(vertices)
```

helper.py

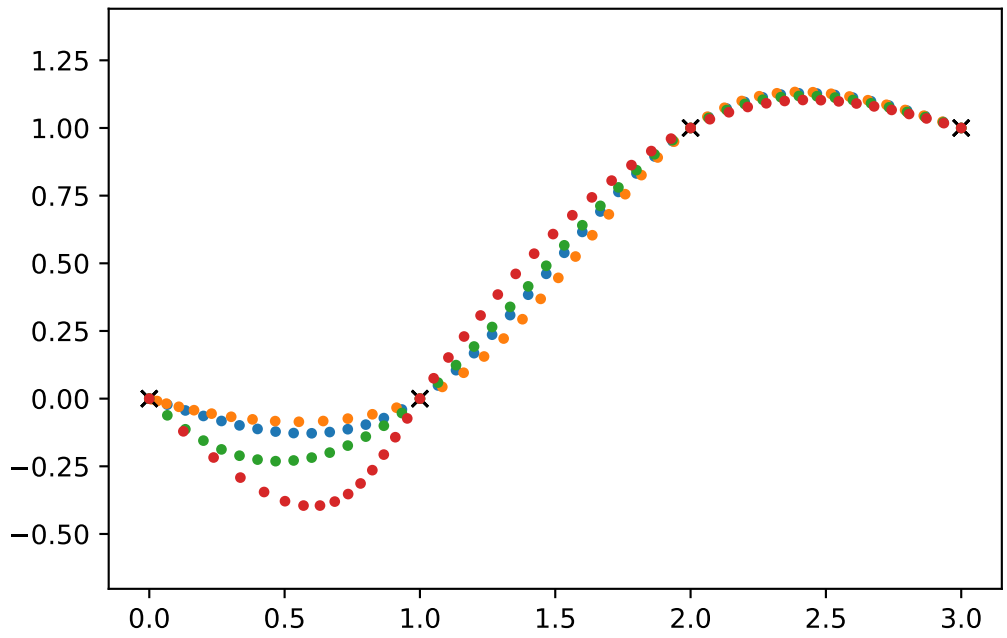
[4]:
```
from helper import plot_spline_2d
```

[5]:
```
plot_spline_2d(s)
```

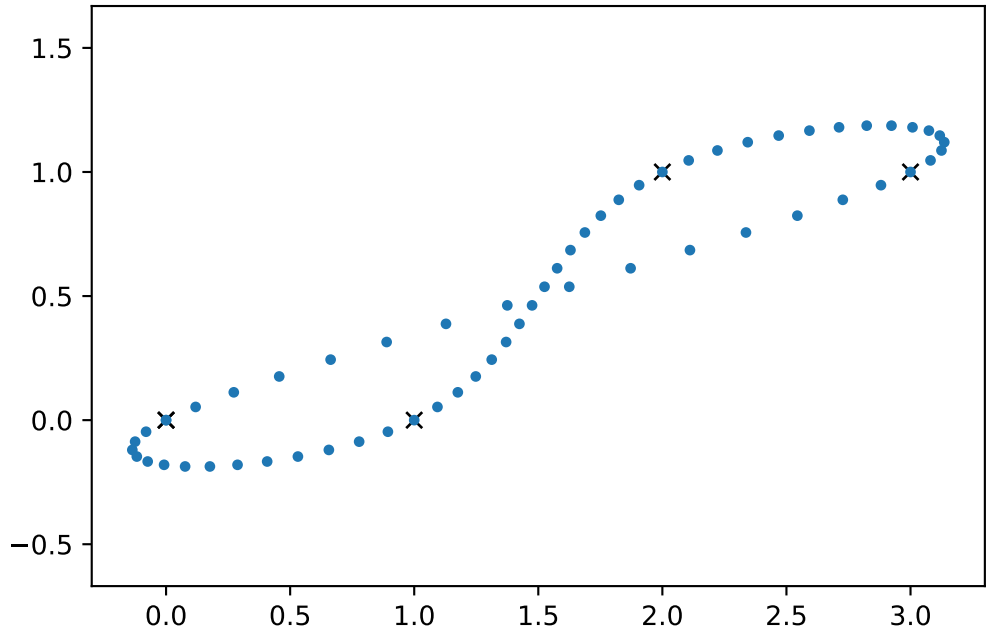


[6]:
```
def plot_natural(*args, **kwargs):
    plot_spline_2d(splines.Natural(*args, **kwargs), chords=False)
```

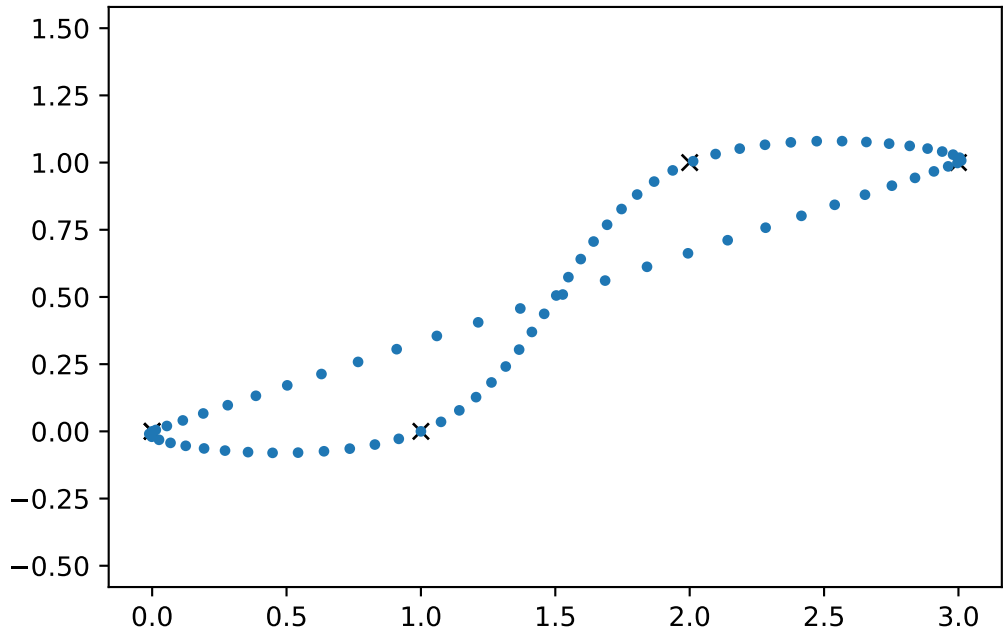
[7]:
```
plot_natural(vertices, endconditions='natural')
plot_natural(vertices, endconditions=[[0, 0], 'natural'])
plot_natural(vertices, endconditions=[[1, -1], 'natural'])
plot_natural(vertices, endconditions=[[2, -2], 'natural'])
```



[8]:
```
plot_natural(vertices, endconditions='closed')
```



[9]:
```
plot_natural(vertices, endconditions='closed', alpha=0.5)
```



doc/euclidean/natural-properties.ipynb ends here.

The following section was generated from doc/euclidean/natural-uniform.ipynb

Uniform Natural Splines

non-uniform (page 35)

[1]:
```
import sympy as sp
sp.init_printing(order='rev-lex')
```

utility.py

[2]:
```
from utility import NamedExpression
```

[3]:
```
t = sp.symbols('t')
```

[4]:
```
a3, a4, b3, b4, c3, c4, d3, d4 = sp.symbols('a:dbm3:5')
```

[5]:
```
b_monomial = sp.Matrix([t**3, t**2, t, 1]).T
```

[6]:
```
p3 = NamedExpression('pbm3', d3 * t**3 + c3 * t**2 + b3 * t + a3)
p4 = NamedExpression('pbm4', d4 * t**3 + c4 * t**2 + b4 * t + a4)
display(p3, p4)
```

$$\boldsymbol{p}_3 = \boldsymbol{d}_3 t^3 + \boldsymbol{c}_3 t^2 + \boldsymbol{b}_3 t + \boldsymbol{a}_3$$

$$\boldsymbol{p}_4 = \boldsymbol{d}_4 t^3 + \boldsymbol{c}_4 t^2 + \boldsymbol{b}_4 t + \boldsymbol{a}_4$$

[7]:
```
pd3 = p3.diff(t)
pd4 = p4.diff(t)
display(pd3, pd4)
```

$$\frac{d}{dt}\boldsymbol{p}_3 = 3\boldsymbol{d}_3 t^2 + 2\boldsymbol{c}_3 t + \boldsymbol{b}_3$$

$$\frac{d}{dt}\boldsymbol{p}_4 = 3\boldsymbol{d}_4 t^2 + 2\boldsymbol{c}_4 t + \boldsymbol{b}_4$$

[8]:
```
equations = [
    p3.evaluated_at(t, 0).with_name('xbm3'),
    p3.evaluated_at(t, 1).with_name('xbm4'),
    p4.evaluated_at(t, 0).with_name('xbm4'),
    p4.evaluated_at(t, 1).with_name('xbm5'),
    pd3.evaluated_at(t, 0).with_name('xbmdot3'),
    pd3.evaluated_at(t, 1).with_name('xbmdot4'),
    pd4.evaluated_at(t, 0).with_name('xbmdot4'),
    pd4.evaluated_at(t, 1).with_name('xbmdot5'),
]
display(*equations)
```

$$\boldsymbol{x}_3 = \boldsymbol{a}_3$$
$$\boldsymbol{x}_4 = \boldsymbol{d}_3 + \boldsymbol{c}_3 + \boldsymbol{b}_3 + \boldsymbol{a}_3$$
$$\boldsymbol{x}_4 = \boldsymbol{a}_4$$
$$\boldsymbol{x}_5 = \boldsymbol{d}_4 + \boldsymbol{c}_4 + \boldsymbol{b}_4 + \boldsymbol{a}_4$$
$$\dot{\boldsymbol{x}}_3 = \boldsymbol{b}_3$$
$$\dot{\boldsymbol{x}}_4 = 3\boldsymbol{d}_3 + 2\boldsymbol{c}_3 + \boldsymbol{b}_3$$
$$\dot{\boldsymbol{x}}_4 = \boldsymbol{b}_4$$
$$\dot{\boldsymbol{x}}_5 = 3\boldsymbol{d}_4 + 2\boldsymbol{c}_4 + \boldsymbol{b}_4$$

[9]:
```
coefficients = sp.solve(equations, [a3, a4, b3, b4, c3, c4, d3, d4])
for c, e in coefficients.items():
    display(NamedExpression(c, e))
```

$$\boldsymbol{a}_3 = \boldsymbol{x}_3$$
$$\boldsymbol{b}_3 = \dot{\boldsymbol{x}}_3$$
$$\boldsymbol{c}_3 = -\dot{\boldsymbol{x}}_4 - 2\dot{\boldsymbol{x}}_3 + 3\boldsymbol{x}_4 - 3\boldsymbol{x}_3$$
$$\boldsymbol{d}_3 = \dot{\boldsymbol{x}}_4 + \dot{\boldsymbol{x}}_3 - 2\boldsymbol{x}_4 + 2\boldsymbol{x}_3$$
$$\boldsymbol{a}_4 = \boldsymbol{x}_4$$
$$\boldsymbol{b}_4 = \dot{\boldsymbol{x}}_4$$
$$\boldsymbol{c}_4 = -\dot{\boldsymbol{x}}_5 - 2\dot{\boldsymbol{x}}_4 + 3\boldsymbol{x}_5 - 3\boldsymbol{x}_4$$
$$\boldsymbol{d}_4 = \dot{\boldsymbol{x}}_5 + \dot{\boldsymbol{x}}_4 - 2\boldsymbol{x}_5 + 2\boldsymbol{x}_4$$

NB: these are the same constants as in M_H (see *Uniform Hermite Splines* (page 15))!

[10]:
```
pdd3 = pd3.diff(t)
pdd4 = pd4.diff(t)
display(pdd3, pdd4)
```

$$\frac{d^2}{dt^2}\boldsymbol{p}_3 = 6\boldsymbol{d}_3 t + 2\boldsymbol{c}_3$$
$$\frac{d^2}{dt^2}\boldsymbol{p}_4 = 6\boldsymbol{d}_4 t + 2\boldsymbol{c}_4$$

[11]:
```
sp.Eq(pdd3.expr.subs(t, 1), pdd4.expr.subs(t, 0))
```

[11]: $$6\boldsymbol{d}_3 + 2\boldsymbol{c}_3 = 2\boldsymbol{c}_4$$

[12]:
```
_.subs(coefficients).simplify()
```

[12]: $$3\boldsymbol{x}_3 = -\dot{\boldsymbol{x}}_5 - 4\dot{\boldsymbol{x}}_4 - \dot{\boldsymbol{x}}_3 + 3\boldsymbol{x}_5$$

generalize by setting index $4 \to i$

$$\dot{\boldsymbol{x}}_{i-1} + 4\dot{\boldsymbol{x}}_i + \dot{\boldsymbol{x}}_{i+1} = 3(\boldsymbol{x}_{i+1} - \boldsymbol{x}_{i-1})$$

$$\begin{bmatrix} 1 & 4 & 1 & & \cdots & 0 \\ & 1 & 4 & 1 & & \vdots \\ & & \ddots & \ddots & & \\ \vdots & & 1 & 4 & 1 & \\ 0 & \cdots & & 1 & 4 & 1 \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{x}}_0 \\ \dot{\boldsymbol{x}}_1 \\ \vdots \\ \dot{\boldsymbol{x}}_{N-2} \\ \dot{\boldsymbol{x}}_{N-1} \end{bmatrix} = \begin{bmatrix} 3(\boldsymbol{x}_2 - \boldsymbol{x}_0) \\ 3(\boldsymbol{x}_3 - \boldsymbol{x}_1) \\ \vdots \\ 3(\boldsymbol{x}_{N-2} - \boldsymbol{x}_{N-4}) \\ 3(\boldsymbol{x}_{N-1} - \boldsymbol{x}_{N-3}) \end{bmatrix}$$

N columns, $N - 2$ rows

End Conditions

add first and last row

end conditions can be mixed, e.g. “clamped” at the beginning and “natural” at the end.

The Python class *splines.Natural* (page 151) uses “natural” end conditions by default.

Natural

notebook about “natural” end conditions (page 98)

Get the uniform case by setting $\Delta_i = 1$.

$$2\dot{\boldsymbol{x}}_0 + \dot{\boldsymbol{x}}_1 = 3(\boldsymbol{x}_1 - \boldsymbol{x}_0)$$
$$\dot{\boldsymbol{x}}_{N-2} + 2\dot{\boldsymbol{x}}_{N-1} = 3(\boldsymbol{x}_{N-1} - \boldsymbol{x}_{N-2})$$

$$\begin{bmatrix} 2 & 1 & & & \cdots & 0 \\ 1 & 4 & 1 & & & \vdots \\ & 1 & 4 & 1 & & \\ & & \ddots & \ddots & & \\ & & 1 & 4 & 1 & \\ \vdots & & & 1 & 4 & 1 \\ 0 & \cdots & & & 1 & 2 \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{x}}_0 \\ \dot{\boldsymbol{x}}_1 \\ \vdots \\ \dot{\boldsymbol{x}}_{N-2} \\ \dot{\boldsymbol{x}}_{N-1} \end{bmatrix} = \begin{bmatrix} 3(\boldsymbol{x}_1 - \boldsymbol{x}_0) \\ 3(\boldsymbol{x}_2 - \boldsymbol{x}_0) \\ 3(\boldsymbol{x}_3 - \boldsymbol{x}_1) \\ \vdots \\ 3(\boldsymbol{x}_{N-2} - \boldsymbol{x}_{N-4}) \\ 3(\boldsymbol{x}_{N-1} - \boldsymbol{x}_{N-3}) \\ 3(\boldsymbol{x}_{N-1} - \boldsymbol{x}_{N-2}) \end{bmatrix}$$

Clamped

clamped (end tangents are given)

$$\dot{\boldsymbol{x}}_0 = D_{\text{begin}}$$
$$\dot{\boldsymbol{x}}_{N-1} = D_{\text{end}}$$

$$\begin{bmatrix} 1 & & & & \cdots & 0 \\ 1 & 4 & 1 & & & \vdots \\ & 1 & 4 & 1 & & \\ & & \ddots & \ddots & & \\ & & 1 & 4 & 1 & \\ \vdots & & & 1 & 4 & 1 \\ 0 & \cdots & & & & 1 \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{x}}_0 \\ \dot{\boldsymbol{x}}_1 \\ \vdots \\ \dot{\boldsymbol{x}}_{N-2} \\ \dot{\boldsymbol{x}}_{N-1} \end{bmatrix} = \begin{bmatrix} D_{\text{begin}} \\ 3(\boldsymbol{x}_2 - \boldsymbol{x}_0) \\ 3(\boldsymbol{x}_3 - \boldsymbol{x}_1) \\ \vdots \\ 3(\boldsymbol{x}_{N-2} - \boldsymbol{x}_{N-4}) \\ 3(\boldsymbol{x}_{N-1} - \boldsymbol{x}_{N-3}) \\ D_{\text{begin}} \end{bmatrix}$$

Closed

$$\begin{bmatrix} 4 & 1 & & \cdots & 0 & 1 \\ 1 & 4 & 1 & & 0 & 0 \\ & 1 & 4 & 1 & & \vdots \\ & & \ddots & \ddots & & \\ \vdots & & 1 & 4 & 1 & \\ 0 & 0 & & 1 & 4 & 1 \\ 1 & 0 & \cdots & & 1 & 4 \end{bmatrix} \begin{bmatrix} \dot{x}_0 \\ \dot{x}_1 \\ \vdots \\ \dot{x}_{N-2} \\ \dot{x}_{N-1} \end{bmatrix} = \begin{bmatrix} 3(x_1 - x_{N-1}) \\ 3(x_2 - x_0) \\ 3(x_3 - x_1) \\ \vdots \\ 3(x_{N-2} - x_{N-4}) \\ 3(x_{N-1} - x_{N-3}) \\ 3(x_0 - x_{N-2}) \end{bmatrix}$$

Solving the System of Equations

tridiagonal matrix algorithm

https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm

https://www.cfd-online.com/Wiki/Tridiagonal_matrix_algorithm[8]

https://gist.github.com/cbellei/8ab3ab8551b8dfc8b081c518ccd9ada9

https://gist.github.com/TheoChristiaanse/d168b7e57dd30342a81aa1dc4eb3e469

doc/euclidean/natural-uniform.ipynb ends here.

The following section was generated from doc/euclidean/natural-non-uniform.ipynb

Non-Uniform Natural Splines

uniform (page 32)

[1]:
```
import sympy as sp
sp.init_printing(order='rev-lex')
```

utility.py

[2]:
```
from utility import NamedExpression
```

[3]:
```
t = sp.symbols('t')
```

[4]:
```
t3, t4, t5 = sp.symbols('t3:6')
```

[5]:
```
b_monomial = sp.Matrix([t**3, t**2, t, 1]).T
b_monomial
```

[5]: $\begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}$

[6]:
```
coefficients3 = sp.symbols('a:dbm3')[::-1]
coefficients4 = sp.symbols('a:dbm4')[::-1]
```

[7]:
```
b_monomial.dot(coefficients3)
```

[7]: $d_3 t^3 + c_3 t^2 + b_3 t + a_3$

[8] https://www.cfd-online.com/Wiki/Tridiagonal_matrix_algorithm_-_TDMA_(Thomas_algorithm)

[8]:
```
p3 = NamedExpression(
    'pbm3',
    b_monomial.dot(coefficients3).subs(t, (t - t3)/(t4 - t3)))
p4 = NamedExpression(
    'pbm4',
    b_monomial.dot(coefficients4).subs(t, (t - t4)/(t5 - t4)))
display(p3, p4)
```

$$\boldsymbol{p}_3 = \frac{\boldsymbol{d}_3 \left(-t_3 + t\right)^3}{\left(t_4 - t_3\right)^3} + \frac{\boldsymbol{c}_3 \left(-t_3 + t\right)^2}{\left(t_4 - t_3\right)^2} + \frac{\boldsymbol{b}_3 \left(-t_3 + t\right)}{t_4 - t_3} + \boldsymbol{a}_3$$

$$\boldsymbol{p}_4 = \frac{\boldsymbol{d}_4 \left(-t_4 + t\right)^3}{\left(t_5 - t_4\right)^3} + \frac{\boldsymbol{c}_4 \left(-t_4 + t\right)^2}{\left(t_5 - t_4\right)^2} + \frac{\boldsymbol{b}_4 \left(-t_4 + t\right)}{t_5 - t_4} + \boldsymbol{a}_4$$

[9]:
```
pd3 = p3.diff(t)
pd4 = p4.diff(t)
display(pd3, pd4)
```

$$\frac{d}{dt}\boldsymbol{p}_3 = \frac{3\boldsymbol{d}_3 \left(-t_3 + t\right)^2}{\left(t_4 - t_3\right)^3} + \frac{\boldsymbol{c}_3 \left(-2t_3 + 2t\right)}{\left(t_4 - t_3\right)^2} + \frac{\boldsymbol{b}_3}{t_4 - t_3}$$

$$\frac{d}{dt}\boldsymbol{p}_4 = \frac{3\boldsymbol{d}_4 \left(-t_4 + t\right)^2}{\left(t_5 - t_4\right)^3} + \frac{\boldsymbol{c}_4 \left(-2t_4 + 2t\right)}{\left(t_5 - t_4\right)^2} + \frac{\boldsymbol{b}_4}{t_5 - t_4}$$

[10]:
```
equations = [
    p3.evaluated_at(t, t3).with_name('xbm3'),
    p3.evaluated_at(t, t4).with_name('xbm4'),
    p4.evaluated_at(t, t4).with_name('xbm4'),
    p4.evaluated_at(t, t5).with_name('xbm5'),
    pd3.evaluated_at(t, t3).with_name('xbmdot3'),
    pd3.evaluated_at(t, t4).with_name('xbmdot4'),
    pd4.evaluated_at(t, t4).with_name('xbmdot4'),
    pd4.evaluated_at(t, t5).with_name('xbmdot5'),
]
```

to simplify the display, but we keep calculating with t_i

[11]:
```
deltas = {
    t3: 0,
    t4: sp.Symbol('Delta3'),
    t5: sp.Symbol('Delta3') + sp.Symbol('Delta4'),
}
```

[12]:
```
for e in equations:
    display(e.subs(deltas))
```

$$\boldsymbol{x}_3 = \boldsymbol{a}_3$$

$$\boldsymbol{x}_4 = \boldsymbol{d}_3 + \boldsymbol{c}_3 + \boldsymbol{b}_3 + \boldsymbol{a}_3$$

$$\boldsymbol{x}_4 = \boldsymbol{a}_4$$

$$\boldsymbol{x}_5 = \boldsymbol{d}_4 + \boldsymbol{c}_4 + \boldsymbol{b}_4 + \boldsymbol{a}_4$$

$$\dot{\boldsymbol{x}}_3 = \frac{\boldsymbol{b}_3}{\Delta_3}$$

$$\dot{\boldsymbol{x}}_4 = \frac{3\boldsymbol{d}_3}{\Delta_3} + \frac{2\boldsymbol{c}_3}{\Delta_3} + \frac{\boldsymbol{b}_3}{\Delta_3}$$

$$\dot{\boldsymbol{x}}_4 = \frac{\boldsymbol{b}_4}{\Delta_4}$$

$$\dot{x}_5 = \frac{3d_4}{\Delta_4} + \frac{2c_4}{\Delta_4} + \frac{b_4}{\Delta_4}$$

[13]:
```
coefficients = sp.solve(equations, coefficients3 + coefficients4)
```

[14]:
```
for c, e in coefficients.items():
    display(NamedExpression(c, e.subs(deltas)))
```

$$a_3 = x_3$$

$$b_3 = \Delta_3 \dot{x}_3$$

$$c_3 = 3x_4 - 3x_3 - \Delta_3 \dot{x}_4 - 2\Delta_3 \dot{x}_3$$

$$d_3 = -2x_4 + 2x_3 + \Delta_3 \dot{x}_4 + \Delta_3 \dot{x}_3$$

$$a_4 = x_4$$

$$b_4 = \dot{x}_4 (\Delta_4 + \Delta_3) - \Delta_3 \dot{x}_4$$

$$c_4 = -\dot{x}_5 (\Delta_4 + \Delta_3) - 2\dot{x}_4 (\Delta_4 + \Delta_3) + 3x_5 - 3x_4 + \Delta_3 \dot{x}_5 + 2\Delta_3 \dot{x}_4$$

$$d_4 = \dot{x}_5 (\Delta_4 + \Delta_3) + \dot{x}_4 (\Delta_4 + \Delta_3) - 2x_5 + 2x_4 - \Delta_3 \dot{x}_5 - \Delta_3 \dot{x}_4$$

[15]:
```
pdd3 = pd3.diff(t)
pdd4 = pd4.diff(t)
display(pdd3, pdd4)
```

$$\frac{d^2}{dt^2} p_3 = \frac{3d_3 (-2t_3 + 2t)}{(t_4 - t_3)^3} + \frac{2c_3}{(t_4 - t_3)^2}$$

$$\frac{d^2}{dt^2} p_4 = \frac{3d_4 (-2t_4 + 2t)}{(t_5 - t_4)^3} + \frac{2c_4}{(t_5 - t_4)^2}$$

[16]:
```
sp.Eq(pdd3.expr.subs(t, t4), pdd4.expr.subs(t, t4))
```

[16]:
$$\frac{3d_3 (2t_4 - 2t_3)}{(t_4 - t_3)^3} + \frac{2c_3}{(t_4 - t_3)^2} = \frac{2c_4}{(t_5 - t_4)^2}$$

[17]:
```
_.subs(coefficients).subs(deltas).simplify()
```

[17]:
$$\frac{2 (-3x_4 + 3x_3 + 2\Delta_3 \dot{x}_4 + \Delta_3 \dot{x}_3)}{\Delta_3^2} = \frac{2 (3x_5 - 3x_4 - \Delta_4 \dot{x}_5 - 2\Delta_4 \dot{x}_4)}{\Delta_4^2}$$

generalize by substituting index $4 \to i$

$$\frac{1}{\Delta_{i-1}} \dot{x}_{i-1} + \left(\frac{2}{\Delta_{i-1}} + \frac{2}{\Delta_i} \right) \dot{x}_i + \frac{1}{\Delta_i} \dot{x}_{i+1} = \frac{3(x_i - x_{i-1})}{{\Delta_{i-1}}^2} + \frac{3(x_{i+1} - x_i)}{{\Delta_i}^2}$$

End Conditions

The Python class *splines.Natural* (page 151) uses "natural" end conditions by default.

Natural

notebook about "natural" end conditions (page 98)

$$2\Delta_0\dot{x}_0 + \Delta_0\dot{x}_1 = 3(x_1 - x_0)$$
$$\Delta_{N-2}\dot{x}_{N-2} + 2\Delta_{N-2}\dot{x}_{N-1} = 3(x_{N-1} - x_{N-2})$$

Other End Conditions

See *notebook about uniform "natural" splines* (page 34)

doc/euclidean/natural-non-uniform.ipynb ends here.

1.5 Bézier Splines

The following section was generated from doc/euclidean/bezier-properties.ipynb

Properties of Bézier Splines

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
import splines
```

[3]:
```
# TODO: example plot of cubic Bézier with "handles"
```

[4]:
```
control_points = [
    [(0, 0), (1, 4)],
    [(1, 4), (2, 2), (4, 4)],
    [(4, 4), (6, 4), (5, 2), (6, 2)],
    [(6, 2), (6, 0), (4, 0), (5, 1), (3, 1)],
]
```

[5]:
```
s = splines.Bernstein(control_points)
```

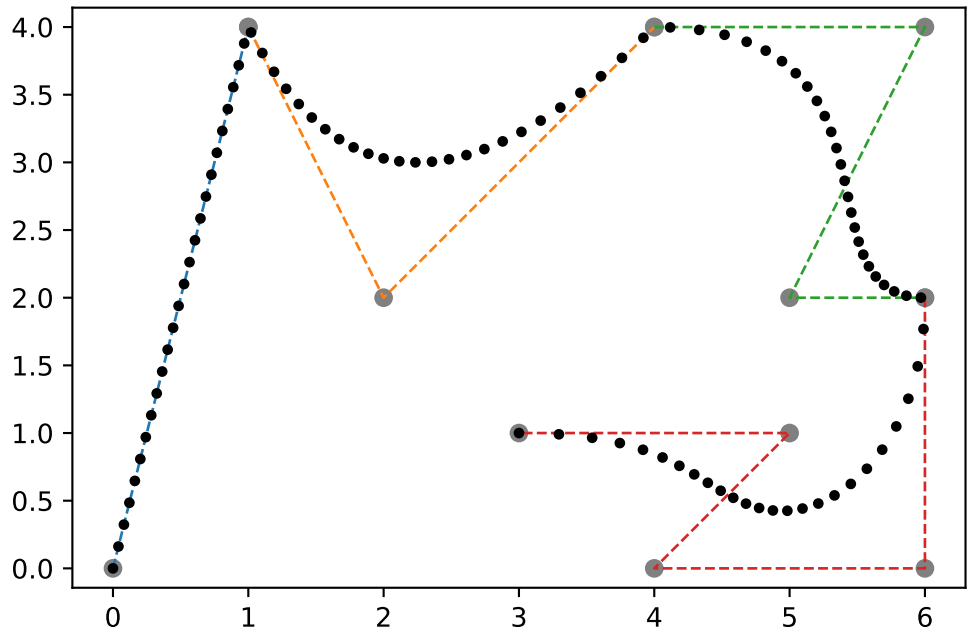
[6]:
```
s.grid
```

[6]:
```
[0, 1, 2, 3, 4]
```

[7]:
```
times = np.linspace(s.grid[0], s.grid[-1], 100)
```

[8]:
```
fig, ax = plt.subplots()
for segment in control_points:
    xy = np.transpose(segment)
    ax.plot(*xy, '--', linewidth=1)
    ax.scatter(*xy, color='grey')
ax.plot(*s.evaluate(times).T, 'k.')
ax.axis('equal');
```

[8]:
```
(-0.30000000000000004, 6.3, -0.2, 4.2)
```



[9]:
```
# TODO: example with non-uniform time
```

doc/euclidean/bezier-properties.ipynb ends here.

The following section was generated from doc/euclidean/bezier-de-casteljau.ipynb

De Casteljau's Algorithm

See also https://pomax.github.io/bezierinfo/.

There are several ways to get to Bézier curves, one was already shown in *the notebook about Hermite curves* (page 21) (but only for cubic curves).

TODO: first explain control polylines and then link to Hermite splines?

Another one is the so-called De Casteljau's algorithm. (TODO: link to De Casteljau)

One nice aspect of this is that the algorithm can be used for arbitrary polynomial degrees.

A Bézier spline is defined by a so-called *control polyline* (or *control polygon*), which comprises a sequence of *control points*. Some of those control points are part of the final spline curve, others lie outside of it. The degree of a spline segment determines how many "off-curve" control points are between two "on-curve" control points.

For example, in a cubic (degree = 3) Bézier spline there are two (= degree - 1) "off-curve" control points.

Two equally valid viewpoints for what a Bézier spline is:

- A sequence of curve segments, each defined by degree + 1 control points. The first control point of a segment is the same as the last control point of the previous one.
- A sequence of control points that can be used to shape the resulting curve. Every degree'th control point lies on the curve and the others define the shape of the curve segments.

TODO: most well-known: cubic Bézier splines (show screenshot from drawing program, e.g. Inkscape). The two "off-curve" control points are shown as "handles".

TODO: typical set of constraints on continuity in drawing programs: C0, C1, G1

Preparations

Before we continue, here are are few preparations for the following calculations:

[1]:
```
%config InlineBackend.print_figure_kwargs = {'bbox_inches': None}
import matplotlib.pyplot as plt
import numpy as np
import sympy as sp
sp.init_printing()
```

We import stuff from the file utility.py:

[2]:
```
from utility import NamedExpression, NamedMatrix
```

helper.py

[3]:
```
from helper import plot_basis
```

Let's prepare a few symbols for later use:

[4]:
```
t, x0, x1, x2, x3, x4 = sp.symbols('t, xbm:5')
```

... and a helper function for plotting:

[5]:
```
def plot_curve(func, points, dots=30, ax=None):
    if ax is None:
        ax = plt.gca()
    times = np.linspace(0, 1, dots)
    ax.plot(*func(points, times).T, '.')
    ax.scatter(*np.asarray(points).T, marker='x', c='black')
    ax.set_title(func.__name__ + ' Bézier curve')
    ax.axis('equal')
```

We also need to prepare for the animations we will see below. This is using code from the file casteljau.py:

[6]:
```
from casteljau import create_animation
from IPython.display import display, HTML

def show_casteljau_animation(points, frames=30, interval=200):
    ani = create_animation(points, frames=frames, interval=interval)
    display({
        'text/html': ani.to_jshtml(default_mode='reflect'),
        'text/plain': 'Animations can only be shown in HTML output, sorry!',
    }, raw=True)
    plt.close()  # avoid spurious figure display
```

Degree 1, a.k.a. linear

But let's start with the trivial case: A Bézier spline of degree 1 is just a piecewise linear curve connecting all the control points. There are no "off-curve" control points that could bend the curve segments.

Assume that we have two control points, x_0 and x_1 ...

... linear equation ...:

$$p_{0,1}(t) = x_0 + t(x_1 - x_0) \tag{1}$$

... in other words ... this is called *affine combination*, but we don't really have to worry about it ...

$$p_{0,1}(t) = (1 - t)x_0 + tx_1 \tag{2}$$

... with $t \in [0,1]$ (which is called *uniform*)

TODO: show change of variables for *non-uniform* curve?

Since we will be needing quite a bunch of those affine combinations, let's create a helper function:

[7]:
```
def affine_combination(one, two):
    return (1 - t) * one + t * two
```

Now we can define the equation in SymPy:

[8]:
```
p01 = NamedExpression('pbm_0,1', affine_combination(x0, x1))
p01
```

[8]: $\boldsymbol{p}_{0,1} = t\boldsymbol{x}_1 + \boldsymbol{x}_0\,(1-t)$

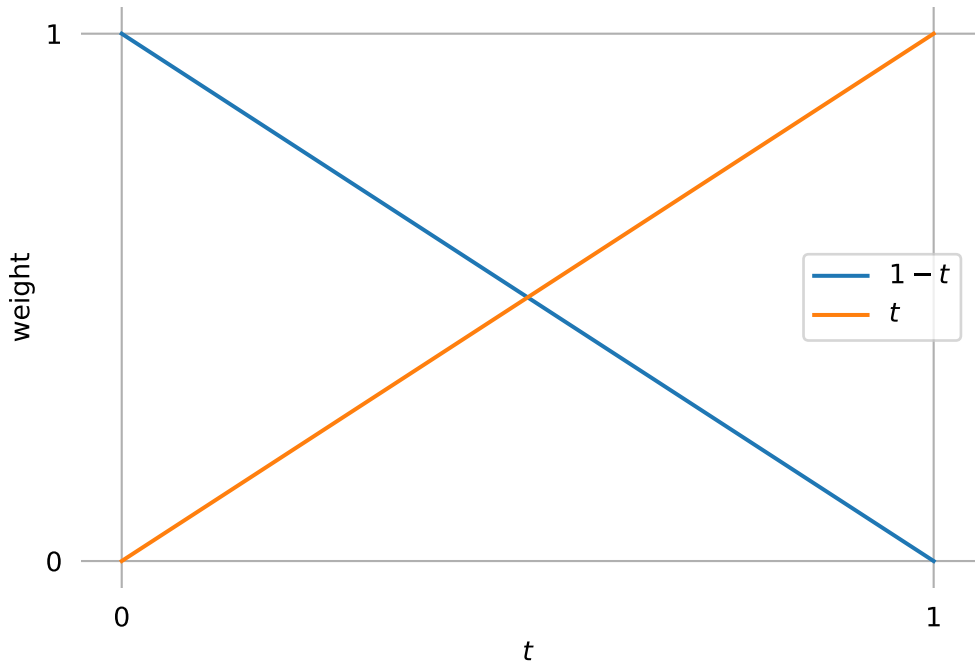
[9]:
```
b1 = [p01.expr.expand().coeff(x.name).factor() for x in (x0, x1)]
b1
```

[9]: $[1-t,\ t]$

Doesn't look like much, but those are the Bernstein bases for degree 1 (https://en.wikipedia.org/wiki/Bernstein_polynomial).

It doesn't get much more interesting if we plot them:

[10]:
```
plot_basis(*b1, labels=b1)
```



If you want to convert this to coefficients for the monomial basis $[t,1]$ instead of the Bernstein basis functions, you can use this matrix:

[11]:
```
M_B1 = NamedMatrix(
    r'{M_\text{B}^{(1)}}',
    sp.Matrix([[c.coeff(x) for x in (x0, x1)]
               for c in p01.expr.as_poly(t).all_coeffs()]))
M_B1
```

[11]:
$$M_{\mathrm{B}}^{(1)} = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}$$

Applying this matrix leads to the coefficients of the linear equation mentioned in the beginning of this section ($\boldsymbol{p}_{0,1}(t) = t(\boldsymbol{x}_1 - \boldsymbol{x}_0) + \boldsymbol{x}_0$):

[12]:
```
sp.MatMul(M_B1.expr, sp.Matrix([x0, x1]))
```

[12]:
$$\begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \end{bmatrix}$$

[13]:
```
_.doit()
```

[13]:
$$\begin{bmatrix} -\boldsymbol{x}_0 + \boldsymbol{x}_1 \\ \boldsymbol{x}_0 \end{bmatrix}$$

If you ever need that, here's the inverse:

[14]:
```
M_B1.I
```

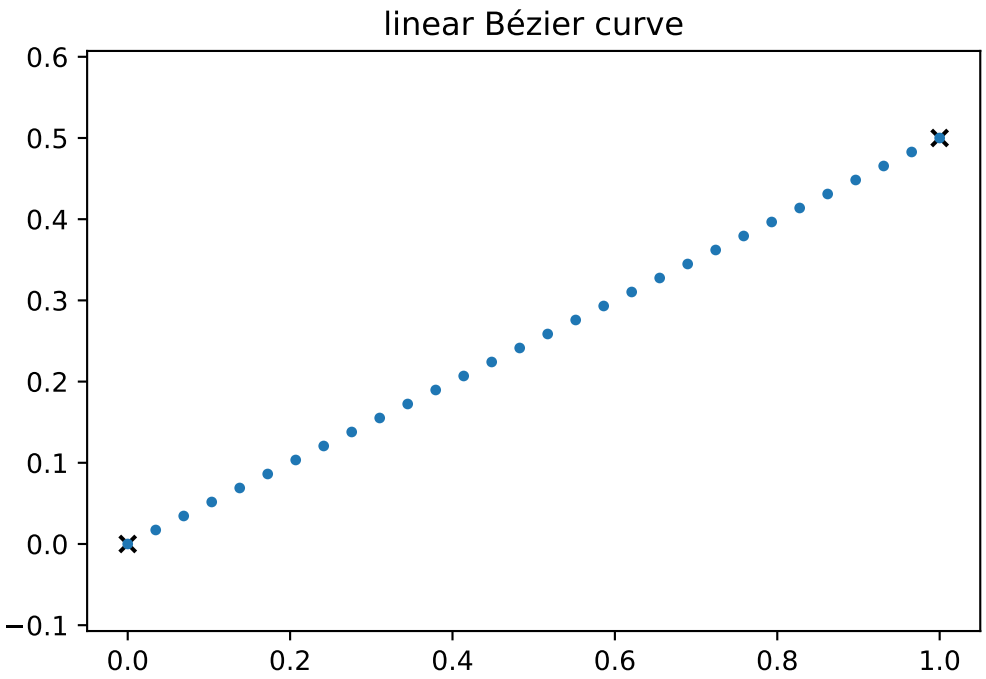
[14]:
$$M_{\mathrm{B}}^{(1)^{-1}} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

Anywho, let's calculate points on the curve by using the Bernstein basis functions:

[15]:
```
def linear(points, times):
    """Evaluate linear Bézier curve (given by two points) at given times."""
    return np.column_stack(sp.lambdify(t, b1)(times)) @ points
```

[16]:
```
points = [
    (0, 0),
    (1, 0.5),
]
```

[17]:
```
plot_curve(linear, points)
```



[18]:
```
show_casteljau_animation(points)
```
```
Animations can only be shown in HTML output, sorry!
```

I know, not very exciting. But it gets better!

Degree 2, a.k.a. quadratic

Consider three control points, x_0, x_1 and x_2 ...

We use the affine combinations of the first two points from above ...

[19]:
```
p01
```
[19]: $p_{0,1} = tx_1 + x_0\,(1-t)$

... and we do the same thing for the second and third point:

[20]:
```
p12 = NamedExpression('pbm_1,2', affine_combination(x1, x2))
p12
```
[20]: $p_{1,2} = tx_2 + x_1\,(1-t)$

Finally, we make another affine combination of those two results:

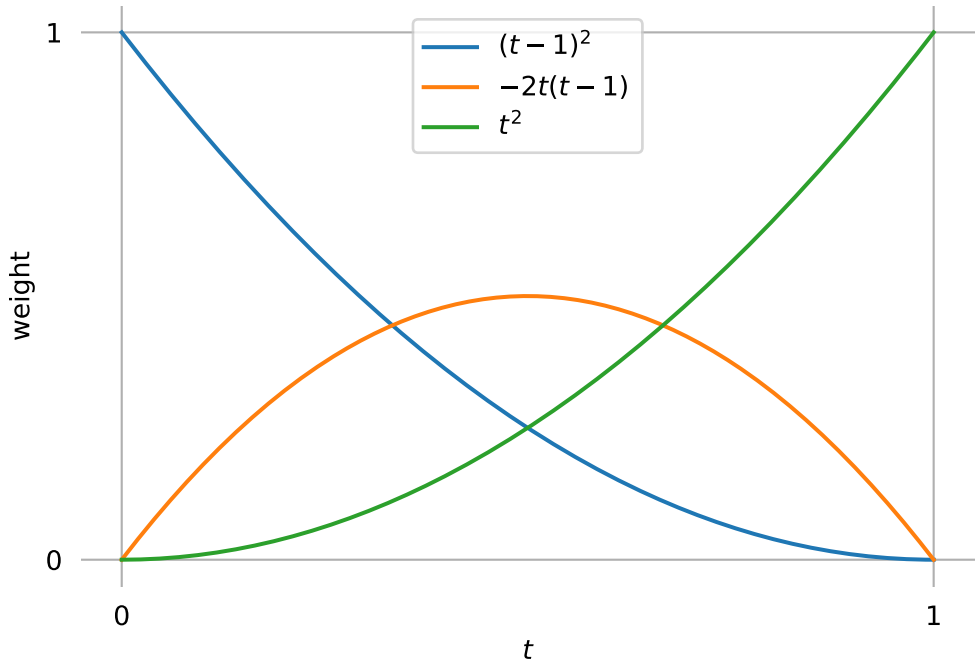
[21]:
```
p02 = NamedExpression('pbm_0,2', affine_combination(p01.expr, p12.expr))
p02
```
[21]: $p_{0,2} = t\,(tx_2 + x_1\,(1-t)) + (1-t)\,(tx_1 + x_0\,(1-t))$

Bernstein basis functions:

[22]:
```
b2 = [p02.expr.expand().coeff(x.name).factor() for x in (x0, x1, x2)]
b2
```

[22]:
$$\left[(t-1)^2,\ -2t\,(t-1),\ t^2\right]$$

[23]:
```
plot_basis(*b2, labels=b2)
```



[24]:
```
M_B2 = NamedMatrix(
    r'{M_\text{B}^{(2)}}',
    sp.Matrix([[c.coeff(x) for x in (x0, x1, x2)]
               for c in p02.expr.as_poly(t).all_coeffs()]))
M_B2
```

[24]:
$$M_{\mathrm{B}}^{(2)} = \begin{bmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

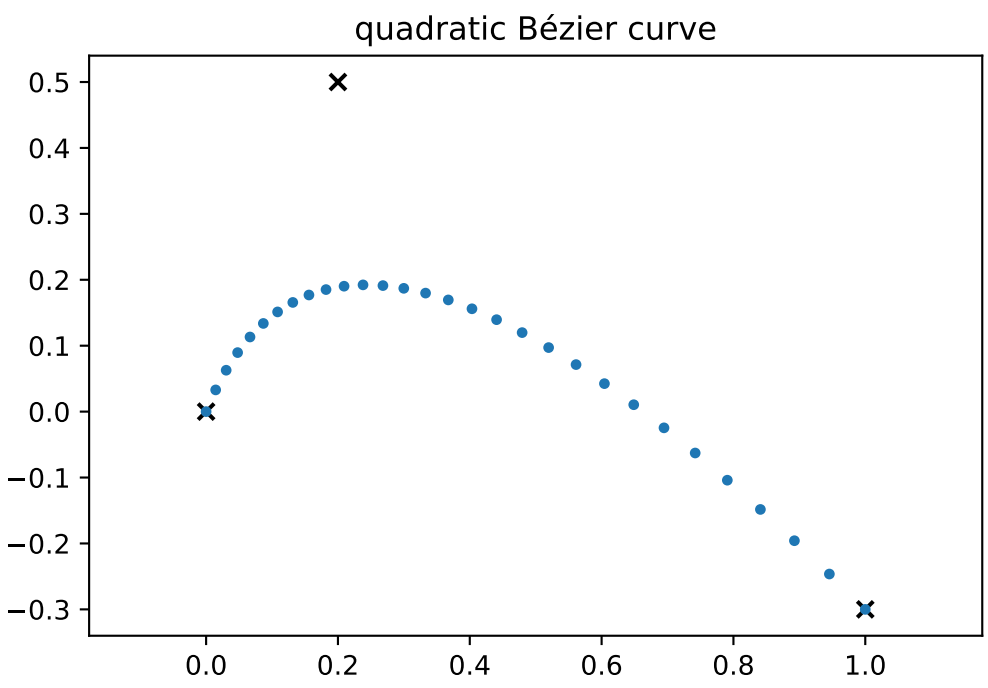
[25]:
```
M_B2.I
```

[25]:
$$M_{\mathrm{B}}^{(2)^{-1}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & \frac{1}{2} & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

[26]:
```
def quadratic(points, times):
    """Evaluate quadratic Bézier curve (given by three points) at given times."""
    return np.column_stack(sp.lambdify(t, b2)(times)) @ points
```

[27]:
```
points = [
    (0, 0),
    (0.2, 0.5),
    (1, -0.3),
]
```

[28]:
```
plot_curve(quadratic, points)
```



[29]:
```
show_casteljau_animation(points)
```

```
Animations can only be shown in HTML output, sorry!
```

For some more insight, let's look at the first derivative of the curve (i.e. the tangent vector):

[30]:
```
v02 = p02.diff(t)
```

... at the beginning and the end of the curve:

[31]:
```
v02.evaluated_at(t, 0)
```

[31]: $\left.\frac{d}{dt}\boldsymbol{p}_{0,2}\right|_{t=0} = -2\boldsymbol{x}_0 + 2\boldsymbol{x}_1$

[32]:
```
v02.evaluated_at(t, 1)
```

[32]: $\left.\frac{d}{dt}\boldsymbol{p}_{0,2}\right|_{t=1} = -2\boldsymbol{x}_1 + 2\boldsymbol{x}_2$

This shows that the tangent vector at the beginning and end of the curve is parallel to the line from $\boldsymbol{x}_0$ to $\boldsymbol{x}_1$ and from $\boldsymbol{x}_1$ to $\boldsymbol{x}_2$, respectively. The length of the tangent vectors is twice the length of those lines.

You might have already seen that coming, but it turns out that the last line in de Casteljau's algorithm ($\boldsymbol{p}_{1,2}(t) - \boldsymbol{p}_{0,1}(t)$ in our case) is exactly half of the tangent vector (at any given $t \in [0, 1]$).

[33]:
```
(v02.expr - 2 * (p12.expr - p01.expr)).simplify()
```

[33]: 0

In case you are wondering, the factor 2 comes from the degree 2 of our quadratic curve.

Degree 3, a.k.a. cubic

Consider four control points, x_0, x_1, x_2 and x_3 ...

By now, the pattern should be clear: We take the result from the first three points from above and affine-combine it with the result for the three points x_1, x_2 and x_3.

Combination of x_2 and x_3:

[34]:
```
p23 = NamedExpression('pbm_2,3', affine_combination(x2, x3))
p23
```

[34]: $p_{2,3} = tx_3 + x_2(1-t)$

Combination of x_1, x_2 and x_3:

[35]:
```
p13 = NamedExpression('pbm_1,3', affine_combination(p12.expr, p23.expr))
p13
```

[35]: $p_{1,3} = t(tx_3 + x_2(1-t)) + (1-t)(tx_2 + x_1(1-t))$

Combination of x_0, x_1, x_2 and x_3:

[36]:
```
p03 = NamedExpression('pbm_0,3', affine_combination(p02.expr, p13.expr))
p03
```

[36]: $p_{0,3} = t(t(tx_3 + x_2(1-t)) + (1-t)(tx_2 + x_1(1-t))) +$
$(1-t)(t(tx_2 + x_1(1-t)) + (1-t)(tx_1 + x_0(1-t)))$

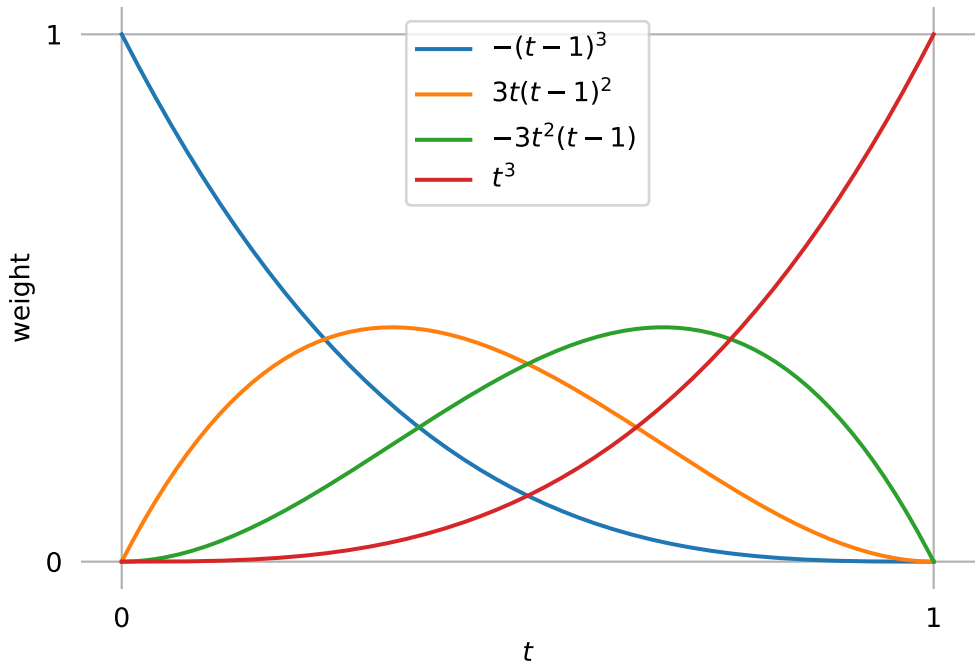
Bernstein bases:

[37]:
```
b3 = [p03.expr.expand().coeff(x.name).factor() for x in (x0, x1, x2, x3)]
b3
```

[37]: $\left[-(t-1)^3,\ 3t(t-1)^2,\ -3t^2(t-1),\ t^3\right]$

TODO: show that those are the same Bernstein bases as in the notebook about Hermite splines

[38]:
```
plot_basis(*b3, labels=b3)
```



[39]:
```
M_B3 = NamedMatrix(
    r'{M_\text{B}^{(3)}}',
    sp.Matrix([[c.coeff(x) for x in (x0, x1, x2, x3)]
               for c in p03.expr.as_poly(t).all_coeffs()]))
M_B3
```

[39]:
$$M_{\mathrm{B}}^{(3)} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

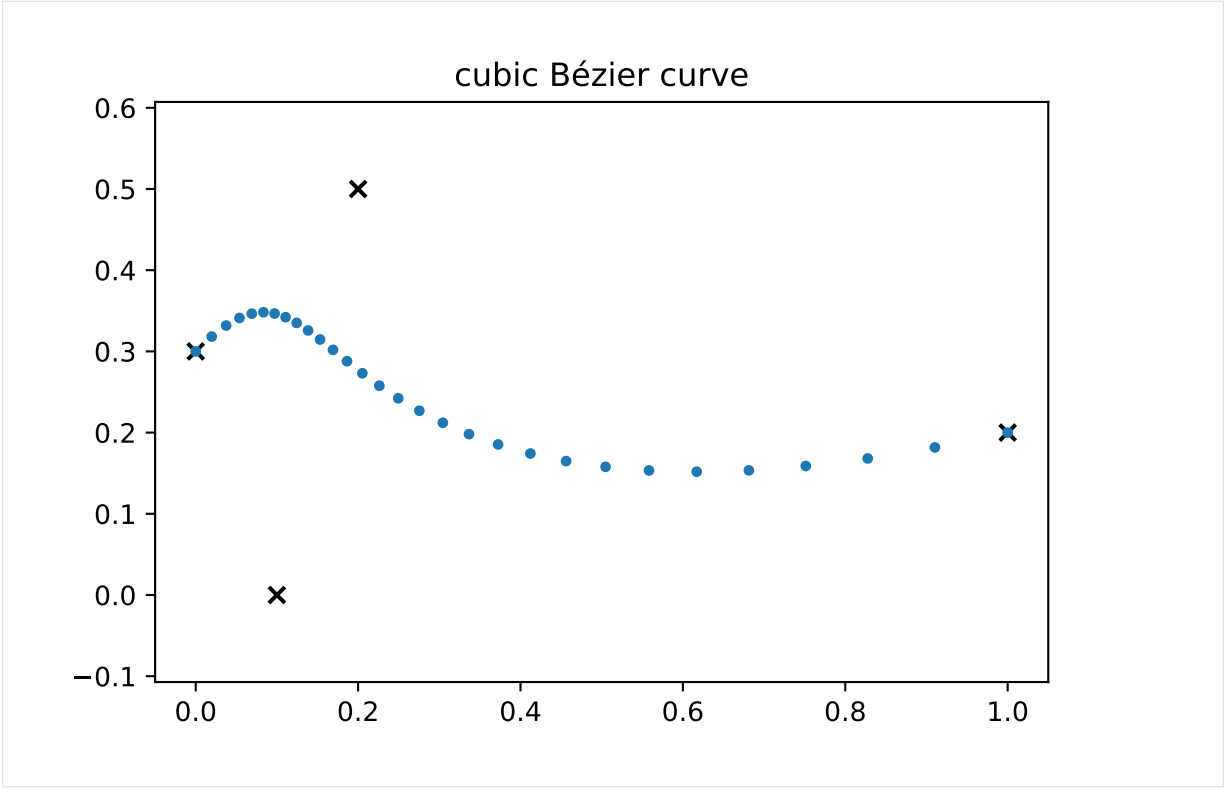
[40]:
```
M_B3.I
```

[40]:
$$M_{\mathrm{B}}^{(3)^{-1}} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{1}{3} & 1 \\ 0 & \frac{1}{3} & \frac{2}{3} & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

[41]:
```
def cubic(points, times):
    """Evaluate cubic Bézier curve (given by four points) at given times."""
    return np.column_stack(sp.lambdify(t, b3)(times)) @ points
```

[42]:
```
points = [
    (0, 0.3),
    (0.2, 0.5),
    (0.1, 0),
    (1, 0.2),
]
```

[43]:
```
plot_curve(cubic, points)
```



[44]:
```
show_casteljau_animation(points)
```
```
Animations can only be shown in HTML output, sorry!
```

As before, let's look at the derivative (i.e. the tangent vector) of the curve:

[45]:
```
v03 = p03.diff(t)
```

... at the beginning and the end of the curve:

[46]:
```
v03.evaluated_at(t, 0)
```
[46]: $$\frac{d}{dt}\boldsymbol{p}_{0,3}\Big|_{t=0} = -3\boldsymbol{x}_0 + 3\boldsymbol{x}_1$$

[47]:
```
v03.evaluated_at(t, 1)
```
[47]: $$\frac{d}{dt}\boldsymbol{p}_{0,3}\Big|_{t=1} = -3\boldsymbol{x}_2 + 3\boldsymbol{x}_3$$

This shows that the tangent vector at the beginning and end of the curve is parallel to the line from $\boldsymbol{x}_0$ to $\boldsymbol{x}_1$ and from $\boldsymbol{x}_2$ to $\boldsymbol{x}_3$, respectively. The length of the tangent vectors is three times the length of those lines.

We can now see that the last line in de Casteljau's algorithm ($\boldsymbol{p}_{1,3}(t) - \boldsymbol{p}_{0,2}(t)$ in this case) is exactly a third of the tangent vector (at any given $t \in [0,1]$):

[48]:
```
(v03.expr - 3 * (p13.expr - p02.expr)).simplify()
```
[48]: 0

Again, the factor 3 comes from the degree 3 of our curve.

We now know the tangent vectors at the beginning and the end of the curve, and obviously we know the values of the curve at the beginning and the end:

[49]:
```
p03.evaluated_at(t, 0)
```

[49]:
$$p_{0,3}\Big|_{t=0} = x_0$$

[50]:
```
p03.evaluated_at(t, 1)
```

[50]:
$$p_{0,3}\Big|_{t=1} = x_3$$

With these four pieces of information, we can find a transformation from the four Bézier control points to the two control points and two tangent vectors of Hermite splines:

[51]:
```
M_BtoH = NamedMatrix(
    r'{M_\text{B$\to$H}}',
    sp.Matrix([[expr.coeff(cv) for cv in [x0, x1, x2, x3]]
               for expr in [
                   x0,
                   x3,
                   v03.evaluated_at(t, 0).expr,
                   v03.evaluated_at(t, 1).expr]]))
M_BtoH
```

[51]:
$$M_{\mathrm{B}\to\mathrm{H}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix}$$

And we can simply invert this if we want to go in the other direction, from Hermite to Bézier:

[52]:
```
M_BtoH.I.pull_out(sp.S.One / 3)
```

[52]:
$$M_{\mathrm{B}\to\mathrm{H}}{}^{-1} = \frac{1}{3}\begin{bmatrix} 3 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 0 & 3 & 0 & -1 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

Of course, those are the same matrices as shown in the *notebook about uniform cubic Hermite splines* (page 15).

TODO: show tangent vectors for non-uniform case

Degree 4, a.k.a. quartic

Consider five control points, x_0, x_1, x_2, x_3 and x_4 ...

More combinations!

[53]:
```
p34 = NamedExpression('pbm_3,4', affine_combination(x3, x4))
p24 = NamedExpression('pbm_2,4', affine_combination(p23.expr, p34.expr))
p14 = NamedExpression('pbm_1,4', affine_combination(p13.expr, p24.expr))
p04 = NamedExpression('pbm_0,4', affine_combination(p03.expr, p14.expr))
p04
```

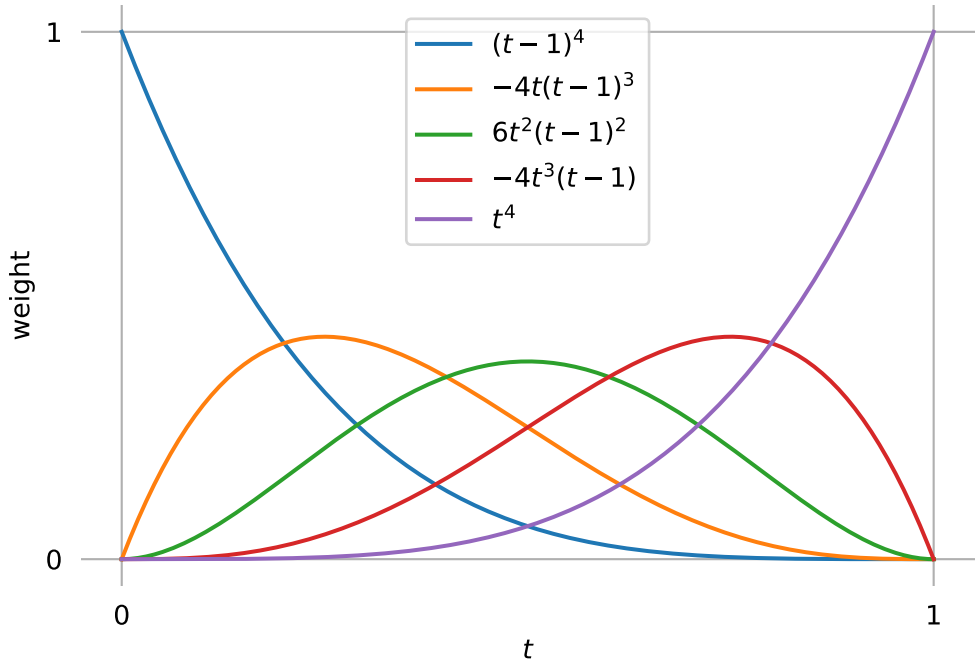
[53]:
$$p_{0,4} = t\left(t\left(t\left(tx_4 + x_3\left(1-t\right)\right) + \left(1-t\right)\left(tx_3 + x_2\left(1-t\right)\right)\right) + \left(1-t\right)\left(t\left(tx_3 + x_2\left(1-t\right)\right) + \left(1-t\right)\left(tx_2 + x_1\left(1-t\right)\right)\right)\right) + \left(1-t\right)\left(t\left(t\left(tx_3 + x_2\left(1-t\right)\right) + \left(1-t\right)\left(tx_2 + x_1\left(1-t\right)\right)\right) + \left(1-t\right)\left(t\left(tx_2 + x_1\left(1-t\right)\right) + \left(1-t\right)\left(tx_1 + x_0\left(1-t\right)\right)\right)\right)$$

Kinda long, but anyway, let's try to extract the Bernstein bases:

[54]:
```
b4 = [p04.expr.expand().coeff(x.name).factor() for x in (x0, x1, x2, x3, x4)]
b4
```

[54]:
$$\left[(t-1)^4,\ -4t\,(t-1)^3,\ 6t^2\,(t-1)^2,\ -4t^3\,(t-1),\ t^4\right]$$

[55]:
```
plot_basis(*b4, labels=b4)
```



[56]:
```
M_B4 = NamedMatrix(
    '{M_B^{(4)}}',
    sp.Matrix([[c.coeff(x) for x in (x0, x1, x2, x3, x4)]
               for c in p04.expr.as_poly(t).all_coeffs()]))
M_B4
```

[56]:
$$M_B^{(4)} = \begin{bmatrix} 1 & -4 & 6 & -4 & 1 \\ -4 & 12 & -12 & 4 & 0 \\ 6 & -12 & 6 & 0 & 0 \\ -4 & 4 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

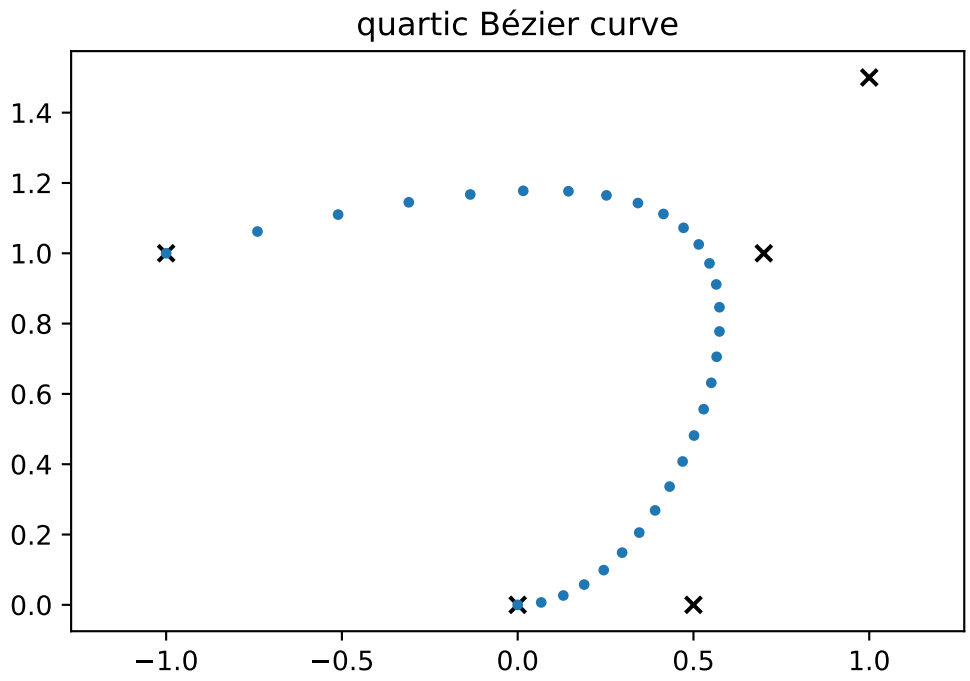
[57]:
```
M_B4.I
```

[57]:
$$M_B^{(4)^{-1}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & \frac{1}{4} & 1 \\ 0 & 0 & \frac{1}{6} & \frac{1}{2} & 1 \\ 0 & \frac{1}{4} & \frac{1}{2} & \frac{3}{4} & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

[58]:
```
def quartic(points, times):
    """Evaluate quartic Bézier curve (given by five points) at given times."""
    return np.column_stack(sp.lambdify(t, b4)(times)) @ points
```

[59]:
```
points = [
    (0, 0),
    (0.5, 0),
    (0.7, 1),
    (1, 1.5),
    (-1, 1),
]
```

[60]:
```
plot_curve(quartic, points)
```



[61]:
```
show_casteljau_animation(points)
```

```
Animations can only be shown in HTML output, sorry!
```

For completeness' sake, let's look at the derivative (i.e. the tangent vector) of the curve:

[62]:
```
v04 = p04.diff(t)
```

... at the beginning and the end of the curve:

[63]:
```
v04.evaluated_at(t, 0)
```

[63]:
$$\frac{d}{dt}\boldsymbol{p}_{0,4}\Big|_{t=0} = -4\boldsymbol{x}_0 + 4\boldsymbol{x}_1$$

[64]:
```
v04.evaluated_at(t, 1)
```

[64]:
$$\frac{d}{dt}\boldsymbol{p}_{0,4}\Big|_{t=1} = -4\boldsymbol{x}_3 + 4\boldsymbol{x}_4$$

By now it shouldn't be surprising that the tangent vector at the beginning and end of the curve is parallel to the line from $\boldsymbol{x}_0$ to $\boldsymbol{x}_1$ and from $\boldsymbol{x}_3$ to $\boldsymbol{x}_4$, respectively. The length of the tangent vectors is four times the length of those lines.

The last line in de Casteljau's algorithm ($\boldsymbol{p}_{1,4}(t) - \boldsymbol{p}_{0,3}(t)$ in this case) is exactly a fourth of the tangent vector (at any given $t \in [0,1]$):

[65]:
```
(v04.expr - 4 * (p14.expr - p03.expr)).simplify()
```

[65]:
```
0
```

Again, the factor 4 comes from the degree 4 of our curve.

Arbitrary Degree

We could go on doing this for higher and higher degrees, but this would get more and more annoying.

Luckily, there is a closed formula available to calculate Bernstein polynomials for an arbitrary degree n!

$$b_{i,n}(x) = \binom{n}{i} x^i (1-x)^{n-i}, \quad i = 0, \dots, n. \tag{3}$$

with the *binomial coefficient* $\binom{n}{i} = \frac{n!}{i!(n-i)!}$.

TODO: link to proof?

TODO: show Bernstein polynomials for "quintic" etc.?

[66]:
```
show_casteljau_animation([
    (0, 0),
    (-1, 1),
    (-0.5, 2),
    (1, 2.5),
    (2, 2),
    (2, 1.5),
    (0.5, 0.5),
    (1, -0.5),
])
```

```
Animations can only be shown in HTML output, sorry!
```

doc/euclidean/bezier-de-casteljau.ipynb ends here.

1.6 Catmull–Rom Splines

What is nowadays known as *Catmull–Rom spline* (named after Edwin Catmull[9] and Raphael Rom[10]) is a specific member of a whole family of splines introduced in [CR74]. That paper only describes *uniform* splines, but their definition can be straightforwardly extended to the *non-uniform* case.

Contrary to popular belief, *Overhauser splines* (as presented in [Ove68]) are not the same!

A Python implementation of Catmull–Rom splines is available in the *splines.CatmullRom* (page 150) class.

The following section was generated from doc/euclidean/catmull-rom-properties.ipynb

Properties of Catmull–Rom Splines

[CR74] presents a whole class of splines with a whole range of properties. Here we only consider one member of this class which is a cubic polynomial interpolating spline with C^1 continuity and local support. Nowadays, this specific case is typically simply referred to as *Catmull–Rom spline*.

[9] https://en.wikipedia.org/wiki/Edwin_Catmull
[10] https://en.wikipedia.org/wiki/Raphael_Rom

This type of splines is very popular because they are very easy to use. Only a sequence of control points has to be specified, the tangents are calculated automatically from the given points. Using those tangents, the spline can be implemented using cubic *Hermite splines* (page 12). Alternatively, spline values can be directly calculated with the *Barry–Goldman algorithm* (page 75).

To calculate the spline values between two control points, the preceding and the following control points are needed as well. The tangent vector at any given control point can be calculated from this control point, its predecessor and its successor. Since Catmull–Rom splines are C^1 continuous, incoming and outgoing tangent vectors are equal.

The following examples use the Python class *splines.CatmullRom* (page 150) to create both uniform and non-uniform splines. Only closed splines are shown, other *end conditions* (page 98) can also be used, but they are not specific to this type of spline.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

Apart from the *splines* (page 149) module ...

[2]:
```
import splines
```

... we also import a few helper functions from helper.py:

[3]:
```
from helper import plot_spline_2d, plot_tangent_2d
```

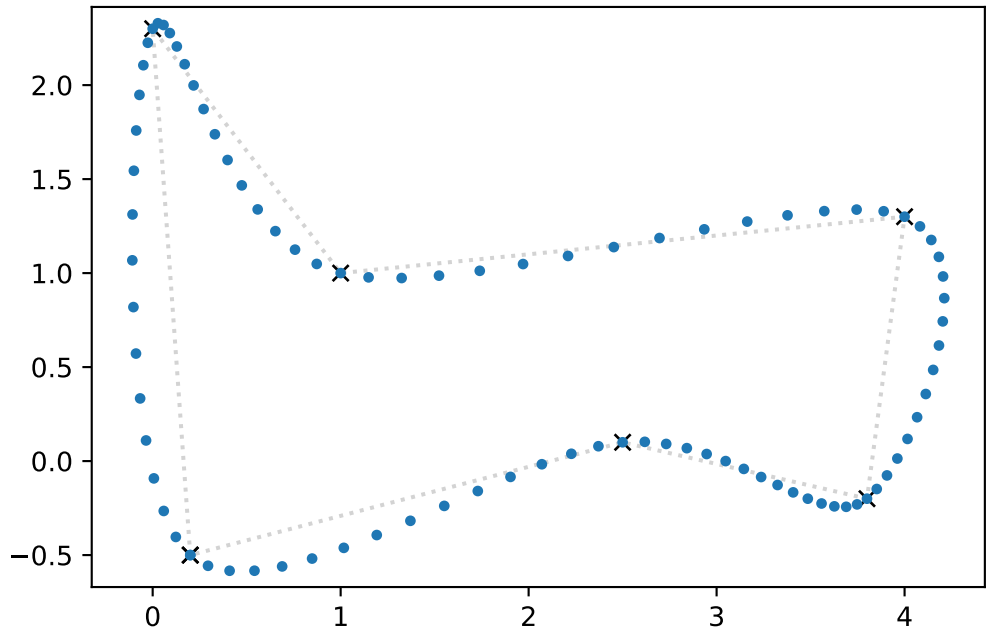
Let's choose a few points for an example:

[4]:
```
points1 = [
    (0.2, -0.5),
    (0, 2.3),
    (1, 1),
    (4, 1.3),
    (3.8, -0.2),
    (2.5, 0.1),
]
```

Without specifying any time values, we get a uniform spline:

[5]:
```
s1 = splines.CatmullRom(points1, endconditions='closed')
```

[6]:
```
fig, ax = plt.subplots()
plot_spline_2d(s1, ax=ax)
```



Tangent Vectors

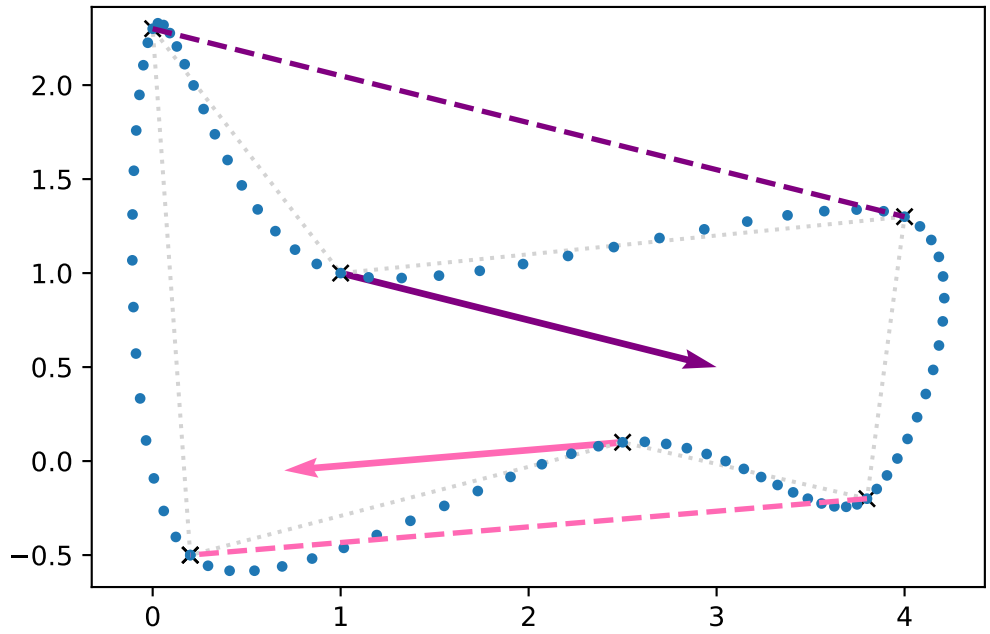
In the uniform case, the tangent vectors at any given control point are parallel to the line connecting the preceding point and the following point. The tangent vector has the same orientation as that line but only half its length. In other (more mathematical) words:

$$\dot{\boldsymbol{x}}_i = \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_{i-1}}{2}$$

This is illustrated for two control points in the following plot:

[7]:
```
for idx, color in zip([2, 5], ['purple', 'hotpink']):
    plot_tangent_2d(
        s1.evaluate(s1.grid[idx], 1),
        s1.evaluate(s1.grid[idx]), color=color, ax=ax)
    ax.plot(
        *s1.evaluate([s1.grid[idx - 1], s1.grid[idx + 1]]).T,
        '--', color=color, linewidth=2)
fig
```

[7]:



We can see here that each tangent vector is parallel to and has half the length of the line connecting the preceding and the following vertex, just as promised.

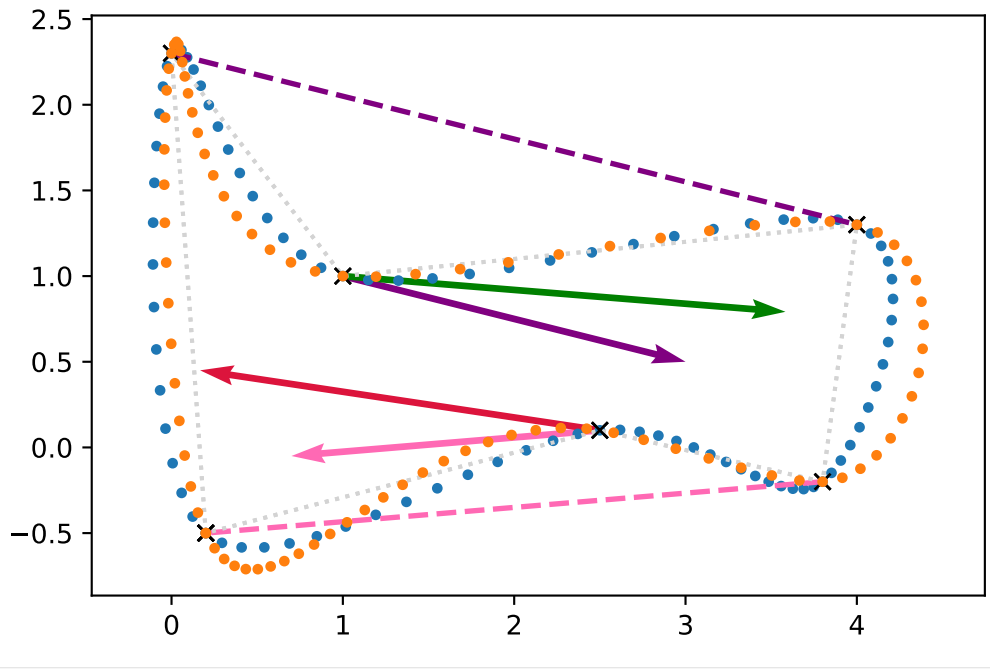
However, this will not be true anymore if we are using non-uniform time instances:

[8]:
```
times2 = 0, 1, 2.2, 3, 4, 4.5, 6
```

[9]:
```
s2 = splines.CatmullRom(points1, grid=times2, endconditions='closed')
```

[10]:
```
plot_spline_2d(s2, ax=ax)
for idx, color in zip([2, 5], ['green', 'crimson']):
    plot_tangent_2d(
        s2.evaluate(s2.grid[idx], 1),
        s2.evaluate(s2.grid[idx]), color=color, ax=ax)
fig
```

[10]:



In the non-uniform case, the equation for the tangent vector gets quite a bit more complicated:

$$\dot{\boldsymbol{x}}_i = \frac{(t_{i+1} - t_i)^2(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (t_i - t_{i-1})^2(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{(t_{i+1} - t_i)(t_i - t_{i-1})(t_{i+1} - t_{i-1})}$$

Equivalently, this can be written as:

$$\dot{\boldsymbol{x}}_i = \frac{(t_{i+1} - t_i)(\boldsymbol{x}_i - \boldsymbol{x}_{i-1})}{(t_i - t_{i-1})(t_{i+1} - t_{i-1})} + \frac{(t_i - t_{i-1})(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{(t_{i+1} - t_i)(t_{i+1} - t_{i-1})}$$

The derivation of this equation is shown in *a separate notebook* (page 72).

Some sources use a simpler equation which is (arguably) not correct (except in the uniform case):

$$\dot{\boldsymbol{x}}_i = \frac{1}{2}\left(\frac{\boldsymbol{x}_i - \boldsymbol{x}_{i-1}}{t_i - t_{i-1}} + \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_i}{t_{i+1} - t_i}\right)$$

There are even sources (e.g. Wikipedia[11]) which show yet a simpler (but even less correct, except in the uniform case) equation:

$$\dot{\boldsymbol{x}}_i = \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_{i-1}}{t_{i+1} - t_{i-1}}$$

Cusps and Self-Intersections

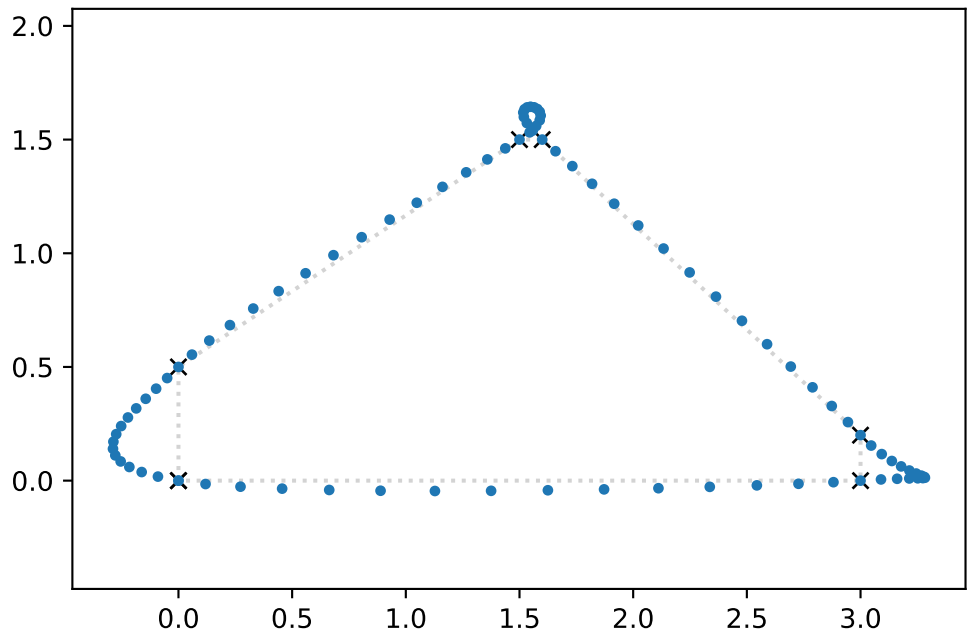
Uniform parametrization typically works very well if the (Euclidean) distances between consecutive vertices are all similar. However, if the distances are very different, the shape of the spline often turns out to be unexpected. Most notably, in extreme cases there might be even cusps or self-intersections within a spline segment.

[11]:
```
def plot_catmull_rom(*args, **kwargs):
    plot_spline_2d(splines.CatmullRom(*args, endconditions='closed', **kwargs))
```

[11] https://en.wikipedia.org/wiki/Cubic_Hermite_spline#Catmull%E2%80%93Rom_spline

[12]:
```
points3 = [
    (0, 0),
    (0, 0.5),
    (1.5, 1.5),
    (1.6, 1.5),
    (3, 0.2),
    (3, 0),
]
```

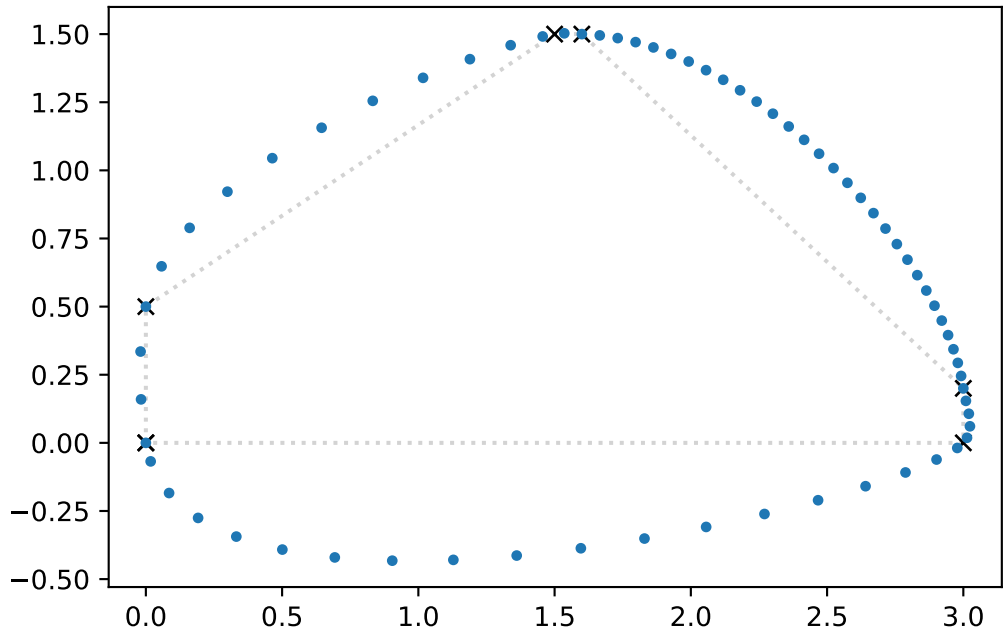
[13]:
```
plot_catmull_rom(points3)
```



We can try to compensate this by manually selecting some non-uniform time instances:

[14]:
```
times3 = 0, 0.2, 0.9, 1, 3, 3.3, 4.5
```

[15]:
```
plot_catmull_rom(points3, times3)
```



Time values can be chosen by trial and error, but there are also ways to choose the time values automatically, as shown in the following sections.

Chordal Parameterization

One way to go about this is to measure the (Euclidean) distances between consecutive vertices (i.e. the "chordal lengths") and simply use those distances as time intervals:

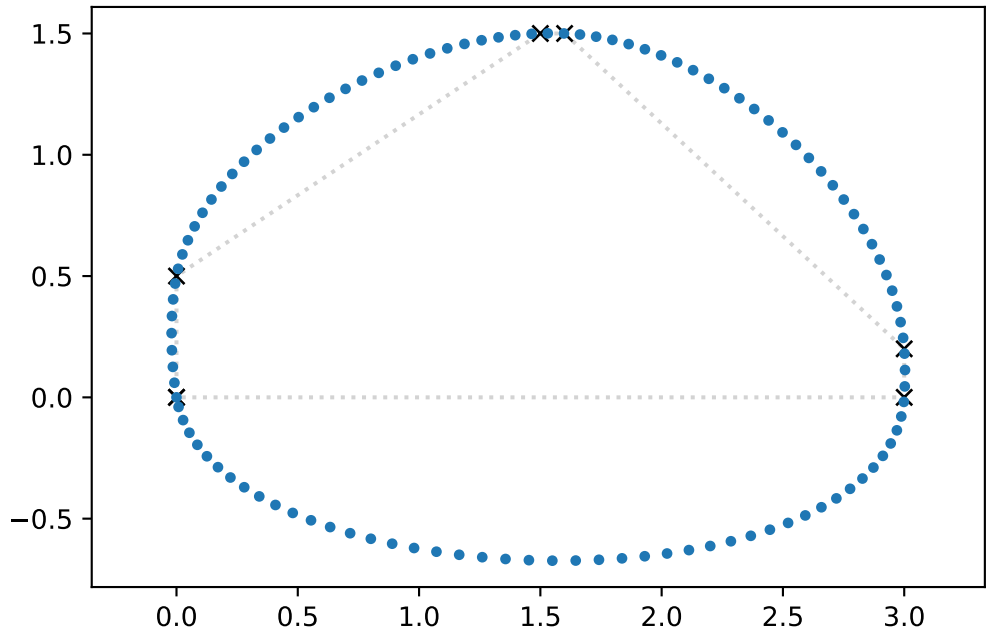
[16]:
```
distances = np.linalg.norm(np.diff(points3 + points3[:1], axis=0), axis=1)
distances
```

[16]:
```
array([0.5       , 1.80277564, 0.1       , 1.91049732, 0.2       ,
       3.        ])
```

[17]:
```
times4 = np.concatenate([[0], np.cumsum(distances)])
times4
```

[17]:
```
array([0.        , 0.5       , 2.30277564, 2.40277564, 4.31327296,
       4.51327296, 7.51327296])
```

[18]:
```
plot_catmull_rom(points3, times4)
```



This makes the speed along the spline nearly constant, but the distance between the curve and its longer chords can become quite huge.

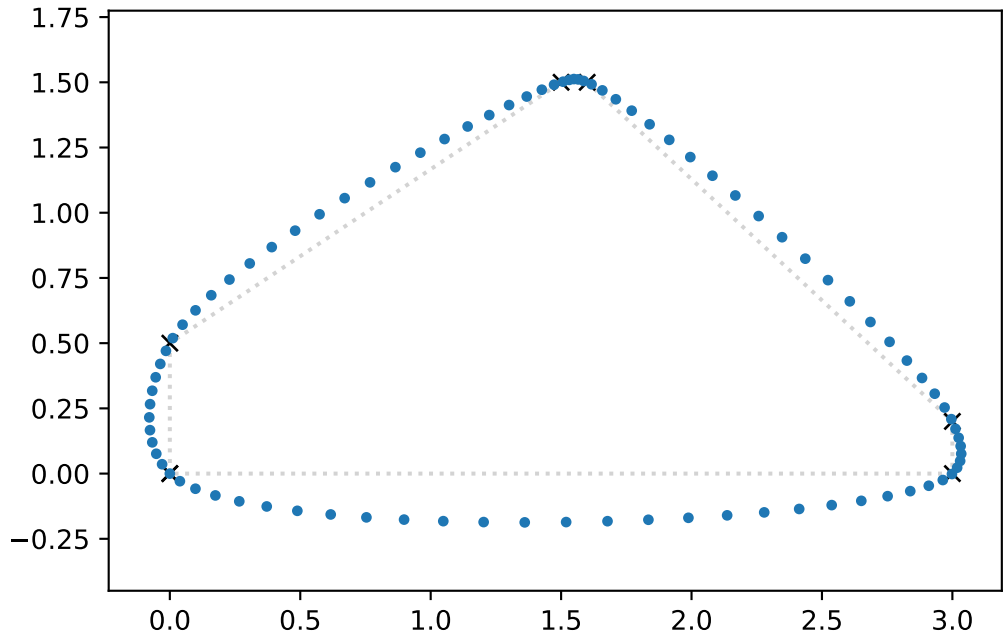
Centripetal Parameterization

As a variation of the previous method, the square roots of the chordal lengths can be used to define the time intervals.

[19]:
```
times5 = np.concatenate([[0], np.cumsum(np.sqrt(distances))])
times5
```

[19]:
```
array([0.        , 0.70710678, 2.04978159, 2.36600935, 3.74821676,
       4.19543036, 5.92748116])
```

[20]:
```
plot_catmull_rom(points3, times5)
```



The curve takes its course much closer to the chords, but its speed is obviously far from constant.

Centripetal parameterization has the very nice property that it guarantees no cusps and no self-intersections, as shown by [YSK11]. The curve is also guaranteed to never "move away" from the successive vertex:

> When centripetal parameterization is used with Catmull–Rom splines to define a path curve, the direction of motion for the object following this path will always be towards the next key-frame position.
>
> —[YSK11], Section 7.2: "Path Curves"

Parameterized Parameterization

It turns out that the previous two parameterization schemes are just two special cases of a more general scheme for obtaining time intervals between control points:

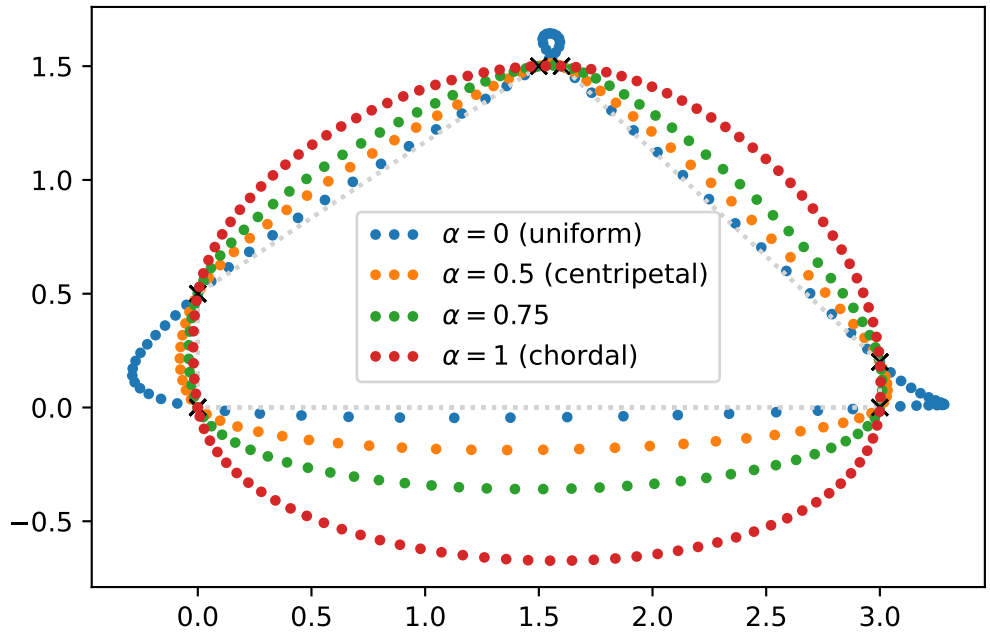
$$t_{i+1} = t_i + |\boldsymbol{x}_{i+1} - \boldsymbol{x}_i|^\alpha, \text{ with } 0 \leq \alpha \leq 1.$$

In the Python class *splines.CatmullRom* (page 150), the parameter `alpha` can be specified.

[21]:
```
def plot_alpha(alpha, label):
    s = splines.CatmullRom(points3, alpha=alpha, endconditions='closed')
    plot_spline_2d(s, label=label)
```

[22]:
```
plot_alpha(0, r'$\alpha = 0$ (uniform)')
plot_alpha(0.5, r'$\alpha = 0.5$ (centripetal)')
plot_alpha(0.75, r'$\alpha = 0.75$')
plot_alpha(1, r'$\alpha = 1$ (chordal)')
plt.legend(loc='center', numpoints=3);
```

[22]:
```
<matplotlib.legend.Legend at 0x7fabe0ac4950>
```



As can be seen here (and as [YSK11] shows to be generally true), the uniform curve is farthest away from short chords and closest to long chords. The chordal curve behaves contrarily: closest to short chords and awkwardly far from long chords. The centripetal curve is closer to the uniform curve for long chords and closer to the chordal curve for short chords, providing a very good compromise.

Any value between 0 and 1 can be chosen for α, but $\alpha = \frac{1}{2}$ (i.e. centripetal parameterization) stands out because it is the only one of them that guarantees no cusps and self-intersections:

> In this paper we prove that, for cubic Catmull–Rom curves, centripetal parameterization is the only parameterization in this family that guarantees that the curves do not form cusps or self-intersections within curve segments.
>
> —[YSK11], abstract

> [...] we mathematically prove that centripetal parameterization of Catmull–Rom curves guarantees that the curve segments cannot form cusps or local self-intersections, while such undesired features can be formed with all other possible parameterizations within this class.
>
> —[YSK11], Section 1: "Introduction"

> Cusps and self-intersections are very common with Catmull–Rom curves for most parameterization choices. In fact, as we will show here, the only parameterization choice that guarantees no cusps and self-intersections within curve segments is centripetal parameterization.
>
> —[YSK11], Section 3: "Cusps and Self-Intersections"

doc/euclidean/catmull-rom-properties.ipynb ends here.

The following section was generated from doc/euclidean/catmull-rom-uniform.ipynb

Uniform Catmull-Rom Splines

In [CR74], a class of splines is presented which can be, in its most generic form, described mathematically with what is referred to as equation (1):

$$F(s) = \frac{\sum x_i(s) w_i(s)}{\sum w_i(s)},$$

where the part $w_i(s) / \sum w_i(s)$ is called *blending functions*.

> Since the blending functions presented above are, as of now, completely arbitrary we impose some constraints in order to make them easier to use. We shall deal only with blending functions that are zero outside of some given interval. Also we require that $\sum w_i(s)$ does not vanish for any s. We shall normalize $w_i(s)$ so that $\sum w_i(s) = 1$ for all s.
>
> —[CR74], section 3, "Blending Functions"

The components of the equation are further constrained to produce a interpolating function:

> Consider the following case: Let $x_i(s)$ be any function interpolating the points p_i through p_{i+k} and let $w_i(s)$ be zero outside (s_{i-1}, s_{i+k+1}). The function $F(s)$ defined in equation (1) will thus be an interpolating function. Intuitively, this says that if all of the functions that have an effect at a point, pass through the point, then the average of the functions will pass through the point.
>
> —[CR74], section 2: "The Model"

> Typo Alert
>
> The typo "p_i through s_{i+k}" has been fixed in the quote above.

> A polynomial of degree k that pass[e]s through $k+1$ points will be used as $x(s)$. In general it will not pass through the other points. If the width of the interval in which $w_i(s)$ is non zero is less than or equal to $k+2$ then $x_i(s)$ will not affect $F(s)$ outside the interpolation interval. This means that $F(s)$ will be an interpolating function. On the other hand if the width of $w_i(s)$ is greater than $k+2$ then $x_i(s)$ will have an effect on the curve outside the interpolation interval. $F(s)$ will then be an approximating function.
>
> —[CR74], section 2: "The Model"

After limiting the scope of the paper to *interpolating* splines, it is further reduced to *uniform* splines:

> [...] in the parametric space we can, without loss of generality, place $s_j = j$.
>
> —[CR74], section 2: "The Model"

Whether or not generality is lost, this means that the rest of the paper doesn't give any hints how to construct non-uniform splines. For those who are interested anyway, we show how to do that in *the notebook about non-uniform Catmull–Rom splines* (page 70) and once again in *the notebook about the Barry–Goldman algorithm* (page 75).

After the aforementioned constraints and the definition of the term *cardinal function* ...

> Cardinal function: a function that is 1 at some knot, 0 at all other knots and can be anything in between the other knots. It satisfies $F_i(s_j) = \delta_{ij}$.
>
> —[CR74], section 1: "Introduction"

... the gratuitously generic equation (1) is made a bit more concrete:

> If in equation (1) we assume $x_i(s)$ to be polynomials of degree k then this equation can be reduced to a much simpler form:
>
> $$F(s) = \sum_j p_j C_{jk}(s)$$
>
> where the $C_{jk}(s)$ are cardinal blending functions and j is the knot to which the cardinal function and the point belong and each $C_{jk}(s)$ is a shifted version of $C_{0,k}(s)$. $C_{0,k}(s)$ is a function of both the degree k of the polynomials and the blending functions $w(s)$:
>
> $$C_{0,k}(s) = \sum_{i=0}^{k} \Big[\prod_{j=i-kj\neq 0}^{i} \left(\frac{s}{j} + 1 \right) \Big] w(s+i)$$

> In essence we see that for a polynomial case our cardinal functions are a blend of Lagrange polynomials. When calculating $C_{0,k}(s)$, $w(s)$ should be centered about $\frac{k}{2}$.
>
> —[CR74], section 4: "Calculating Cardinal Functions"

This looks like something we can work with, even though the blending function $w(s)$ is still not defined.

[1]:
```
import sympy as sp
```

We use t instead of s:

[2]:
```
t = sp.symbols('t')
```

[3]:
```
i, j, k = sp.symbols('i j k')
```

[4]:
```
w = sp.Function('w')
```

[5]:
```
C0k = sp.Sum(
        sp.Product(
            sp.Piecewise((1, sp.Eq(j, 0)), ((t / j) + 1, True)),
            (j, i - k, i)) * w(t + i),
        (i, 0, k))
C0k
```

[5]:
$$\sum_{i=0}^{k} w(i+t) \prod_{j=i-k}^{i} \begin{cases} 1 & \text{for } j = 0 \\ 1 + \frac{t}{j} & \text{otherwise} \end{cases}$$

Blending Functions

[CR74] leaves the choice of blending function to the reader. It shows two plots (figure 1 and figure 3) for a custom blending function stitched together from two Bézier curves, but it doesn't show the cardinal function nor an actual spline created from it.

The only other concrete suggestion is to use B-spline basis functions as blending functions. A quadratic B-spline basis function is shown in figure 2 and both cardinal functions and example curves are shown that utilize both quadratic and cubic B-spline basis functions (figures 4 through 7). No mathematical description of B-spline basis functions is given, instead the paper refers to [GR74]. That paper provides a pair of equations (3.1 and 3.2) that can be used to recursively construct B-spline basis functions. Simplified to the *uniform* case, this leads to the base case (degree 0) ...

[6]:
```
B0 = sp.Piecewise((0, t < i), (1, t < i + 1), (0, True))
B0
```

[6]:
$$\begin{cases} 0 & \text{for } i > t \\ 1 & \text{for } t < i + 1 \\ 0 & \text{otherwise} \end{cases}$$

... which can be used to obtain the linear (degree 1) basis functions:

[7]:
```
B1 = (t - i) * B0 + (i + 2 - t) * B0.subs(i, i + 1)
```

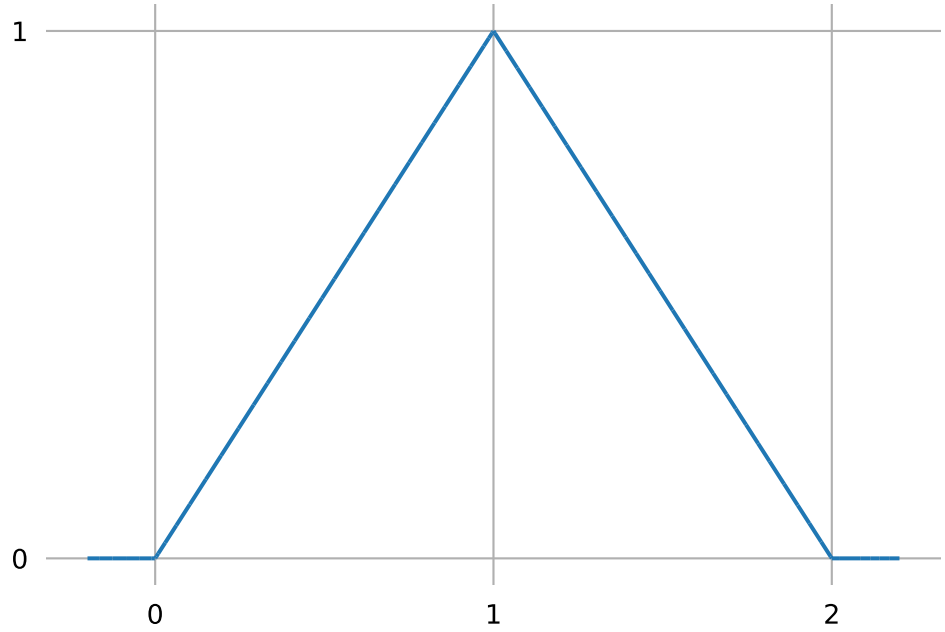
We can use one of them (where $i = 0$) as blending function:

[8]:
```
w1 = B1.subs(i, 0)
```

With some helper functions from helper.py we can plot this.

[9]:
```
from helper import plot_sympy, grid_lines
```

[10]:
```
plot_sympy(w1, (t, -0.2, 2.2))
grid_lines([0, 1, 2], [0, 1])
```



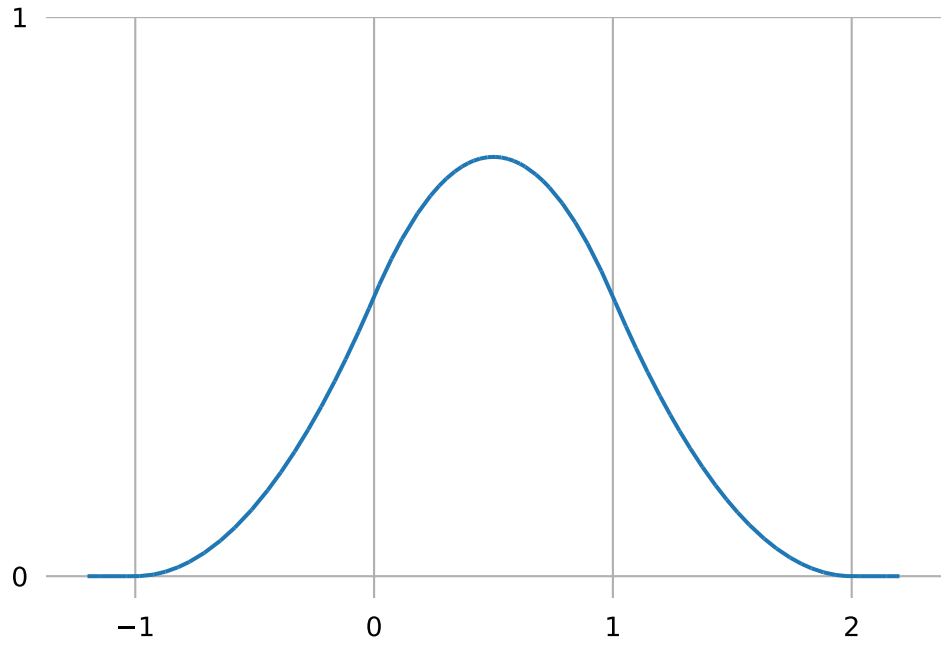
The quadratic (degree 2) basis functions can be obtained like this:

[11]:
```
B2 = (t - i) / 2 * B1 + (i + 3 - t) / 2 * B1.subs(i, i + 1)
```

For our further calculations, we use the function with $i = -1$ as blending function:

[12]:
```
w2 = B2.subs(i, -1)
```

[13]:
```
plot_sympy(w2, (t, -1.2, 2.2))
grid_lines([-1, 0, 1, 2], [0, 1])
```



This should be the same function as shown in figure 2 of [CR74].

Cardinal Functions

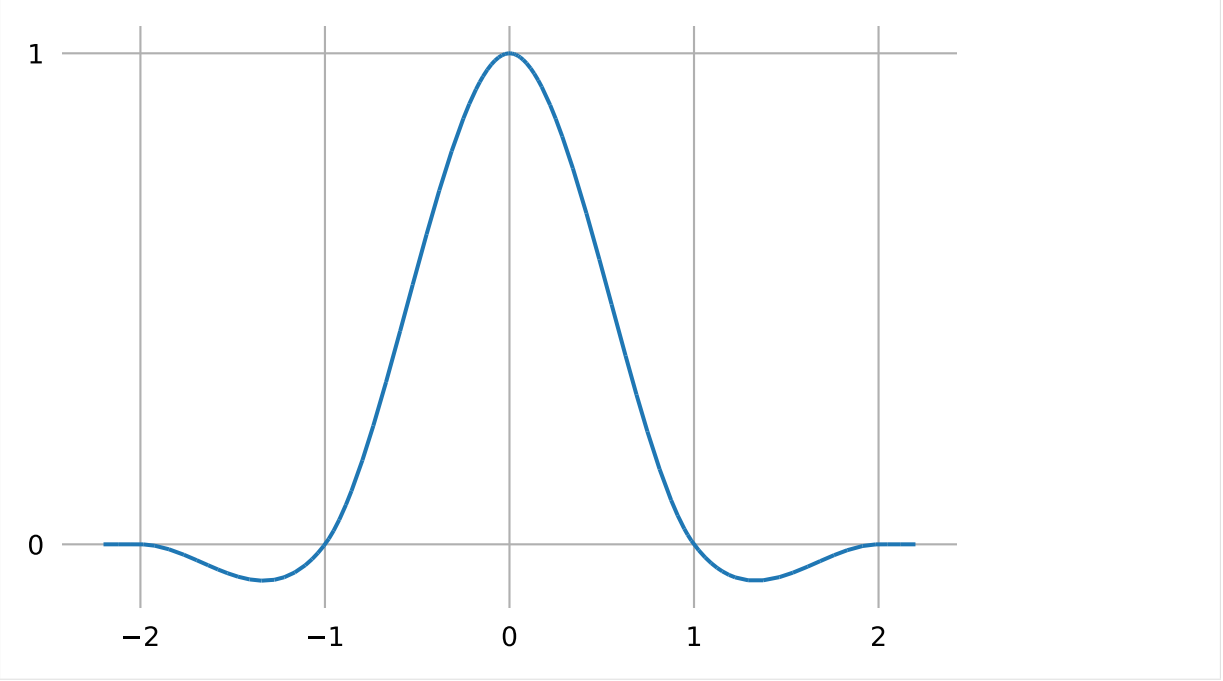
The first example curve in the paper (figure 5) is a cubic curve, constructed using a cardinal function with $k = 1$ (i.e. using linear Lagrange interpolation) and a quadratic B-spline basis function (as shown above) as blending function.

With the information so far, we can construct the cardinal function $C_{0,1}(t)$, using our *quadratic* B-spline blending function `w2` (which is, as required, centered about $\frac{k}{2}$):

[14]:
```
C01 = C0k.subs(k, 1).replace(w, lambda x: w2.subs(t, x)).doit().simplify()
C01
```

[14]:
$$\begin{cases} 0 & \text{for } t < -2 \\ \frac{(t+1)(t+2)^2}{2} & \text{for } t < -1 \\ -\frac{3t^3}{2} - \frac{5t^2}{2} + 1 & \text{for } t < 0 \\ \frac{3t^3}{2} - \frac{5t^2}{2} + 1 & \text{for } t < 1 \\ \frac{(1-t)(t-2)^2}{2} & \text{for } t < 2 \\ 0 & \text{otherwise} \end{cases}$$

[15]:
```
plot_sympy(C01, (t, -2.2, 2.2))
grid_lines(range(-2, 3), [0, 1])
```



This should be the same function as shown in figure 4 of [CR74].

The paper does not show that, but we can also try to flip the respective degrees of Lagrange interpolation and B-spline blending. In other words, we can set $k = 2$ to construct the cardinal function $C_{0,2}(t)$, this time using the *linear* B-spline blending function `w1` (which is also centered about $\frac{k}{2}$) leading to a total degree of 3:

[16]:
```
C02 = C0k.subs(k, 2).replace(w, lambda x: w1.subs(t, x)).doit().simplify()
```

And as it turns out, this is exactly the same thing!

[17]:
```
assert C01 == C02
```

By the way, we come to the same conclusion in our *notebook about the Barry–Goldman algorithm* (page 75), which means that this is also true in the *non-uniform* case.

Many authors nowadays, when using the term *Catmull–Rom spline*, mean the cubic spline created using exactly this cardinal function.

As we have seen, this can be equivalently understood either as three linear interpolations (more exactly: one interpolation and two extrapolations) followed by quadratic B-spline blending or as two overlapping quadratic Lagrange interpolations followed by linear blending.

Example Plot

[18]:
```
import matplotlib.pyplot as plt
import numpy as np
```

To quickly check how a spline segment would look like when using the cardinal function we just derived, let's define a few points …

[19]:
```
vertices = np.array([
    (-0.1, -0.5),
    (0, 0),
    (1, 0),
```

(continues on next page)

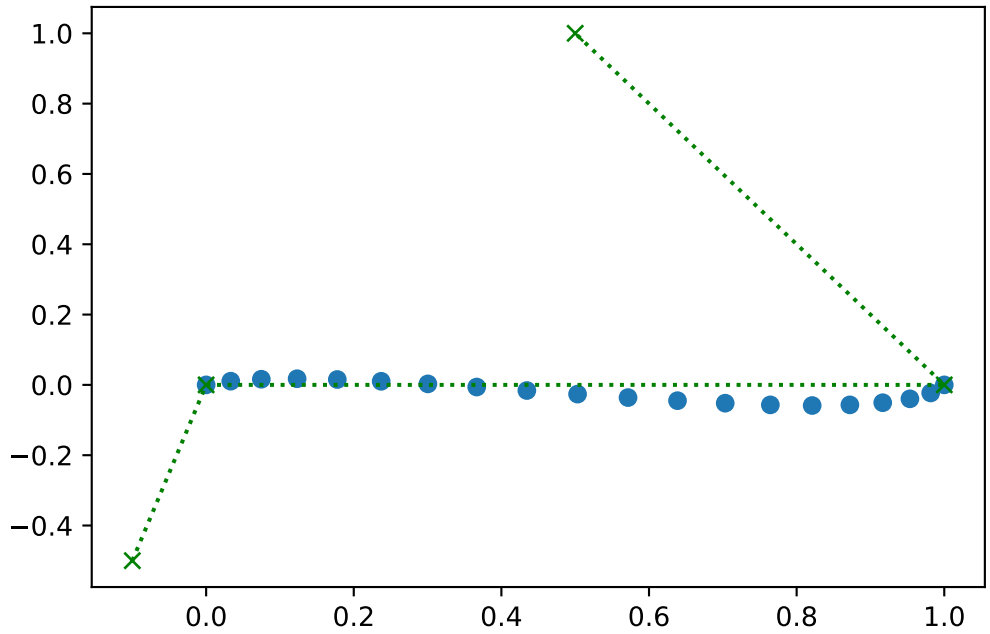
(continued from previous page)

```
    (0.5, 1),
])
```

… and plot $F(t)$ (or $F(s)$, as it has been called originally):

[20]:
```
plt.scatter(*np.array([
    sum([vertices[i] * C01.subs(t, s - i + 1) for i in range(4)])
    for s in np.linspace(0, 1, 20)]).T)
plt.plot(*vertices.T, 'x:g');
```

[20]:
```
[<matplotlib.lines.Line2D at 0x7f7d4b5d0b90>]
```



For calculating more than one segment, and also for creating non-uniform Catmull–Rom splines, the class *splines.CatmullRom* (page 150) can be used. For more plots, see *the notebook about properties of Catmull–Rom splines* (page 52).

Basis Polynomials

The piecewise expression for the cardinal function is a bit unwieldy to work with, so let's bring it into a form we know how to deal with.

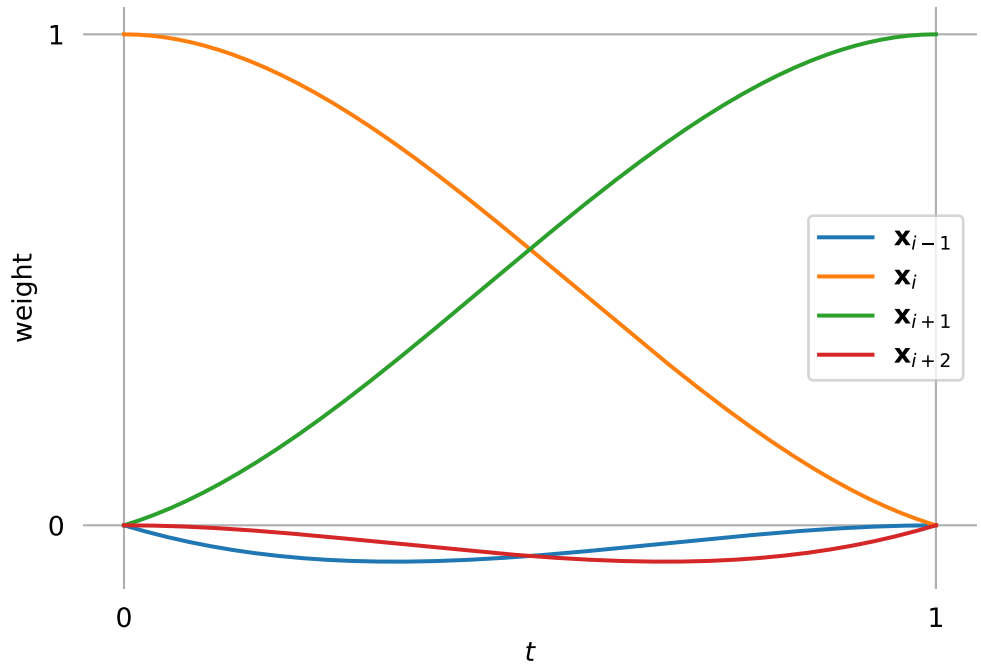
We are splitting the piecewise expression into four separate pieces, each one to be evaluated at $0 \leq t \leq 1$. We are also reversing the order of the pieces, to match our intended control point order:

[21]:
```
b_CR = sp.Matrix([
    expr.subs(t, t + cond.args[1] - 1)
    for expr, cond in C01.args[1:-1][::-1]]).T
b_CR.T
```

[21]:
$$\begin{bmatrix} -\frac{t(t-1)^2}{2} \\ \frac{3t^3}{2} - \frac{5t^2}{2} + 1 \\ -\frac{3(t-1)^3}{2} - \frac{5(t-1)^2}{2} + 1 \\ \frac{t^2(t-1)}{2} \end{bmatrix}$$

[22]:
```
from helper import plot_basis
```

[23]:
```
plot_basis(*b_CR, labels=sp.symbols('xbm_i-1 xbm_i xbm_i+1 xbm_i+2'))
```



For the following sections, we are using a few tools from utility.py:

[24]:
```
from utility import NamedExpression, NamedMatrix
```

Basis Matrix

[25]:
```
b_monomial = sp.Matrix([t**3, t**2, t, 1]).T
M_CR = NamedMatrix(r'{M_\text{CR}}', 4, 4)
control_points = sp.Matrix(sp.symbols('xbm3:7'))
```

As usual, we look at the fifth polynomial segment (from x_4 to x_5):

[26]:
```
p4 = NamedExpression('pbm4', b_monomial * M_CR.name * control_points)
p4
```

[26]:
$$\boldsymbol{p}_4 = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} M_{\text{CR}} \begin{bmatrix} x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

From the basis polynomials and the control points, we can already calculate $\boldsymbol{p}_4(t)$ ...

[27]:
```
p4.expr = b_CR.dot(control_points).expand().collect(t)
p4
```

[27]:
$$\boldsymbol{p}_4 = t^3 \left(-\frac{x_3}{2} + \frac{3x_4}{2} - \frac{3x_5}{2} + \frac{x_6}{2} \right) + t^2 \left(x_3 - \frac{5x_4}{2} + 2x_5 - \frac{x_6}{2} \right) + t \left(-\frac{x_3}{2} + \frac{x_5}{2} \right) + x_4$$

... and with a little bit of squinting, we can directly read off the coefficients of the basis matrix:

[28]:
```
M_CR.expr = sp.Matrix([
    [b.get(m, 0) for b in [
        p4.expr.expand().coeff(cv).collect(t, evaluate=False)
        for cv in control_points]]
    for m in b_monomial])
M_CR.pull_out(sp.S.Half)
```

[28]:
$$M_{\mathrm{CR}} = \frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

This matrix also appears in section 6 of [CR74].

In case you want to copy&paste it, here's a plain text version:

[29]:
```
print(_.expr)
```

```
(1/2)*Matrix([
[-1,  3, -3,  1],
[ 2, -5,  4, -1],
[-1,  0,  1,  0],
[ 0,  2,  0,  0]])
```

And, in case somebody needs it, its inverse looks like this:

[30]:
```
M_CR.I
```

[30]:
$${M_{\mathrm{CR}}}^{-1} = \begin{bmatrix} 1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 6 & 4 & 2 & 1 \end{bmatrix}$$

[31]:
```
print(_.expr)
```

```
Matrix([[1, 1, -1, 1], [0, 0, 0, 1], [1, 1, 1, 1], [6, 4, 2, 1]])
```

Tangent Vectors

To get the tangent vectors, we simply have to take the first derivative …

[32]:
```
pd4 = p4.diff(t)
```

… and evaluate it at the beginning and the end of the segment:

[33]:
```
pd4.evaluated_at(t, 0)
```

[33]:
$$\frac{d}{dt}\boldsymbol{p}_4\Big|_{t=0} = -\frac{\boldsymbol{x}_3}{2} + \frac{\boldsymbol{x}_5}{2}$$

[34]:
```
pd4.evaluated_at(t, 1)
```

[34]:
$$\frac{d}{dt}\boldsymbol{p}_4\Big|_{t=1} = -\frac{\boldsymbol{x}_4}{2} + \frac{\boldsymbol{x}_6}{2}$$

These two expressions can be generalized to (as already shown in *the notebook about Catmull–Rom properties* (page 54)):

$$\dot{\boldsymbol{x}}_i = \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_{i-1}}{2}$$

doc/euclidean/catmull-rom-uniform.ipynb ends here.

Non-Uniform Catmull–Rom Splines

[CR74] describes only the *uniform case* (page 61), but it is straightforward to extend the method to non-uniform splines.

The method comprises using three linear interpolations (and *extra*polations) between neighboring pairs of the four relevant control points and then blending the three resulting points with a quadratic B-spline basis function.

As we have seen in the *notebook about uniform Catmull–Rom splines* (page 65) and as we will again see in the *notebook about the Barry–Goldman algorithm* (page 78), the respective degrees can be reversed. This means that equivalently, two (overlapping) quadratic Lagrange interpolations can be used, followed by linearly blending the two resulting points.

Since latter is both easier to implement and easier to wrap one's head around, we use it in the following derivations.

We will derive the *tangent vectors* (page 72) at the segment boundaries (which will serve as basis for deriving *non-uniform Kochanek–Bartels splines* (page 95) later) and the *basis matrix* (page 73). See the *notebook about the Barry–Goldman algorithm* (page 75) for an alternative (but closely related) derivation.

[1]:
```
import sympy as sp
sp.init_printing()
```

[2]:
```
x3, x4, x5, x6 = sp.symbols('xbm3:7')
```

[3]:
```
t, t3, t4, t5, t6 = sp.symbols('t t3:7')
```

We use some tools from utility.py:

[4]:
```
from utility import NamedExpression, NamedMatrix
```

As shown in the *notebook about Lagrange interpolation* (page 4), it can be interpolated using *Neville's algorithm*:

[5]:
```
def lerp(xs, ts, t):
    """Linear interpolation.

    Returns the interpolated value at time *t*,
    given the two values *xs* at times *ts*.

    """
    x_begin, x_end = xs
    t_begin, t_end = ts
    return (x_begin * (t_end - t) + x_end * (t - t_begin)) / (t_end - t_begin)
```

[6]:
```
def neville(xs, ts, t):
    """Lagrange interpolation using Neville's algorithm.

    Returns the interpolated value at time *t*,
    given the values *xs* at times *ts*.

    """
    assert len(xs) == len(ts)
    while len(xs) > 1:
        step = len(ts) - len(xs) + 1
        xs = [
            lerp(*args, t)
```

(continues on next page)

(continued from previous page)

```
            for args in zip(zip(xs, xs[1:]), zip(ts, ts[step:]))]
    return xs[0]
```

Alternatively, sympy.interpolate()[12] could be used.

We use two overlapping quadratic Lagrange interpolations followed by linear blending:

[7]:
```
p4 = NamedExpression(
    'pbm4',
    lerp([
        neville([x3, x4, x5], [t3, t4, t5], t),
        neville([x4, x5, x6], [t4, t5, t6], t),
    ], [t4, t5], t))
```

Note

Since the two invocations of Neville's algorithm overlap, some values that are used by both are unnecessarily computed by both. It would be more efficient to calculate each of these values only once.

The *Barry–Goldman algorithm* (page 75) avoids this repeated computation.

But here, since we are using symbolic expressions, this doesn't really matter because the redundant expressions should be simplified away by SymPy.

[8]:
```
p4.simplify()
```

[8]:
$$p_4 = \frac{(t-t_4)\,(t_3-t_4)\,(t_3-t_5)\,(-(t-t_4)\,(t_4-t_5)\,(-x_5\,(t-t_6)+x_6\,(t-t_5))+(t-t_6)\,(t_5-t_6)\,(-x_4\,(t-t_5)+x_5\,(t-t_4)))}{(t_3-t_4)\,(t_3-t_5)\,(t_4-}$$

The following expressions can be simplified by introducing a few new symbols Δ_i:

[9]:
```
delta3, delta4, delta5 = sp.symbols('Delta3:6')
deltas = {
    t4 - t3: delta3,
    t5 - t4: delta4,
    t6 - t5: delta5,
    t5 - t3: delta3 + delta4,
    t6 - t4: delta4 + delta5,
    t6 - t3: delta3 + delta4 + delta5,
    # A few special cases that SymPy has a hard time resolving:
    t4 + t4 - t3: t4 + delta3,
    t6 + t6 - t3: t6 + delta3 + delta4 + delta5,
}
```

[12] https://docs.sympy.org/latest/modules/polys/reference.html#sympy.polys.polyfuncs.interpolate

Tangent Vectors

To get the tangent vectors at the control points, we just have to take the first derivative …

[10]:
```
pd4 = p4.diff(t)
```

… and evaluate it at t_4 and t_5:

[11]:
```
start_tangent = pd4.evaluated_at(t, t4)
start_tangent.subs(deltas).simplify()
```

[11]:
$$\left.\frac{d}{dt}\boldsymbol{p}_4\right|_{t=t_4} = \frac{\Delta_3^2\left(-\boldsymbol{x}_4+\boldsymbol{x}_5\right)+\Delta_4^2\left(-\boldsymbol{x}_3+\boldsymbol{x}_4\right)}{\Delta_3\Delta_4\left(\Delta_3+\Delta_4\right)}$$

[12]:
```
end_tangent = pd4.evaluated_at(t, t5)
end_tangent.subs(deltas).simplify()
```

[12]:
$$\left.\frac{d}{dt}\boldsymbol{p}_4\right|_{t=t_5} = \frac{-\Delta_4^2\boldsymbol{x}_5+\Delta_4^2\boldsymbol{x}_6-\Delta_5^2\boldsymbol{x}_4+\Delta_5^2\boldsymbol{x}_5}{\Delta_4\Delta_5\left(\Delta_4+\Delta_5\right)}$$

Both results lead to the same general expression:

$$\dot{\boldsymbol{x}}_i = \frac{(t_{i+1}-t_i)^2(\boldsymbol{x}_i-\boldsymbol{x}_{i-1})+(t_i-t_{i-1})^2(\boldsymbol{x}_{i+1}-\boldsymbol{x}_i)}{(t_{i+1}-t_i)(t_i-t_{i-1})(t_{i+1}-t_{i-1})}$$

An alternative (but very similar) way to derive these tangent vectors is shown in the *notebook about the Barry–Goldman algorithm* (page 83).

And there is yet another way to calculate the tangents, without even needing to obtain a *cubic* polynomial and its derivative: Since we are using a linear blend of two *quadratic* polynomials, we know that at the beginning ($t = t_4$) only the first quadratic polynomial has an influence and at the end ($t = t_5$) only the second quadratic polynomial is relevant. Therefore, to determine the tangent vector at the beginning of the segment, it is sufficient to get the derivative of the first quadratic polynomial.

[13]:
```
first_quadratic = neville([x3, x4, x5], [t3, t4, t5], t)
```

[14]:
```
sp.degree(first_quadratic, t)
```

[14]:
2

[15]:
```
first_quadratic.diff(t).subs(t, t4)
```

[15]:
$$\frac{\frac{(-t_3+t_4)(-\boldsymbol{x}_4+\boldsymbol{x}_5)}{-t_4+t_5}+\frac{(-t_4+t_5)(-\boldsymbol{x}_3+\boldsymbol{x}_4)}{-t_3+t_4}}{-t_3+t_5}$$

This can be written as (which is sometimes called the *standard three-point difference formula*):

$$\dot{\boldsymbol{x}}_i = \frac{\Delta_i\boldsymbol{v}_{i-1}+\Delta_{i-1}\boldsymbol{v}_i}{\Delta_{i-1}+\Delta_i},$$

with $\Delta_i = t_{i+1}-t_i$ and $\boldsymbol{v}_i = \frac{\boldsymbol{x}_{i+1}-\boldsymbol{x}_i}{\Delta_i}$.

[dB78] calls this *piecewise cubic Bessel interpolation*, and it has also been called *Bessel tangent method*, *Overhauser method* and *Bessel–Overhauser splines*.

Note

Even though this formula is commonly associated with the name *Overhauser*, it is *not* describing the tangents of *Overhauser splines* (as presented in [Ove68]).

Long story short, it's the same as we had above:

[16]:
```
assert sp.simplify(_ - start_tangent.expr) == 0
```

The first derivative of the second quadratic polynomial can be used to get the tangent vector at the end of the segment.

[17]:
```
second_quadratic = neville([x4, x5, x6], [t4, t5, t6], t)
second_quadratic.diff(t).subs(t, t5)
```

[17]:
$$\frac{\frac{(-t_4+t_5)(-x_5+x_6)}{-t_5+t_6} + \frac{(-t_5+t_6)(-x_4+x_5)}{-t_4+t_5}}{-t_4+t_6}$$

[18]:
```
assert sp.simplify(_ - end_tangent.expr) == 0
```

You might encounter another way to write the equation for $\dot{x}_4$ (e.g. at https://stackoverflow.com/a/23980479/) ...

[19]:
```
(x4 - x3) / (t4 - t3) - (x5 - x3) / (t5 - t3) + (x5 - x4) / (t5 - t4)
```

[19]:
$$\frac{-x_4+x_5}{-t_4+t_5} - \frac{-x_3+x_5}{-t_3+t_5} + \frac{-x_3+x_4}{-t_3+t_4}$$

... but again, this is equivalent to the equation shown above:

[20]:
```
assert sp.simplify(_ - start_tangent.expr) == 0
```

Basis Matrix

We already have the correct result, but if we want to derive our *basis matrix*, we have to re-scale this a bit. The parameter is supposed to go from 0 to 1 instead of from t_4 to t_5:

[21]:
```
p4_normalized = p4.expr.subs(t, t * (t5 - t4) + t4)
```

[22]:
```
M_CR = NamedMatrix(
    r'{M_{\text{CR},4}}',
    sp.Matrix([[c.expand().coeff(x).factor() for x in (x3, x4, x5, x6)]
               for c in p4_normalized.as_poly(t).all_coeffs()]))
M_CR.subs(deltas).simplify()
```

[22]:
$$M_{\text{CR},4} = \begin{bmatrix} -\frac{\Delta_4^2}{\Delta_3(\Delta_3+\Delta_4)} & \frac{\Delta_4(\Delta_3+\Delta_4+\Delta_5)}{\Delta_3(\Delta_4+\Delta_5)} & -\frac{\Delta_4(\Delta_3+\Delta_4+\Delta_5)}{\Delta_5(\Delta_3+\Delta_4)} & \frac{\Delta_4^2}{\Delta_5(\Delta_4+\Delta_5)} \\ \frac{2\Delta_4^2}{\Delta_3(\Delta_3+\Delta_4)} & -\frac{\Delta_4(\Delta_3+2\Delta_4+2\Delta_5)}{\Delta_3(\Delta_4+\Delta_5)} & \frac{\Delta_4(\Delta_3+\Delta_4+2\Delta_5)}{\Delta_5(\Delta_3+\Delta_4)} & -\frac{\Delta_4^2}{\Delta_5(\Delta_4+\Delta_5)} \\ -\frac{\Delta_4^2}{\Delta_3(\Delta_3+\Delta_4)} & \frac{-\Delta_3+\Delta_4}{\Delta_3} & \frac{\Delta_3}{\Delta_3+\Delta_4} & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

We can try to manually simplify this a bit more:

[23]:
```
M_CR.subs(deltas).simplify().subs([[e.factor(), e] for e in [
    delta4 / (delta4 + delta5) + delta4 / delta3,
    -delta4 / (delta3 + delta4) - delta4 / delta5,
    -delta4 / (delta4 + delta5) - 2 * delta4 / delta3,
    2 * delta4 / (delta3 + delta4) + delta4 / delta5,
]])
```

[23]:

$$M_{\mathrm{CR},4} = \begin{bmatrix} -\frac{\Delta_4^2}{\Delta_3(\Delta_3+\Delta_4)} & \frac{\Delta_4}{\Delta_4+\Delta_5} + \frac{\Delta_4}{\Delta_3} & -\frac{\Delta_4}{\Delta_3+\Delta_4} - \frac{\Delta_4}{\Delta_5} & \frac{\Delta_4^2}{\Delta_5(\Delta_4+\Delta_5)} \\ \frac{2\Delta_4^2}{\Delta_3(\Delta_3+\Delta_4)} & -\frac{\Delta_4}{\Delta_4+\Delta_5} - \frac{2\Delta_4}{\Delta_3} & \frac{2\Delta_4}{\Delta_3+\Delta_4} + \frac{\Delta_4}{\Delta_5} & -\frac{\Delta_4^2}{\Delta_5(\Delta_4+\Delta_5)} \\ -\frac{\Delta_4^2}{\Delta_3(\Delta_3+\Delta_4)} & \frac{-\Delta_3+\Delta_4}{\Delta_3} & \frac{\Delta_3}{\Delta_3+\Delta_4} & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

We can even introduce two new symbols in order to simplify it yet a bit more:

[24]:
```
phi, psi = sp.symbols('Phi4 Psi4')
phi_psi_subs = {
    phi: delta4 / (delta3 + delta4),
    psi: delta4 / (delta4 + delta5),
}
phi_psi_subs
```

[24]:

$$\left\{\Phi_4 : \frac{\Delta_4}{\Delta_3+\Delta_4}, \Psi_4 : \frac{\Delta_4}{\Delta_4+\Delta_5}\right\}$$

[25]:
```
sp.Matrix([
    [
        -phi * delta4 / delta3,
        psi + delta4 / delta3,
        -phi - (delta4 / delta5),
        psi * delta4 / delta5,
    ], [
        phi * 2 * delta4 / delta3,
        -psi - 2 * delta4 / delta3,
        2 * phi + delta4 / delta5,
        -psi * delta4 / delta5,
    ], [
        -phi * delta4 / delta3,
        (delta4 - delta3) / delta3,
        delta3 / (delta3 + delta4),
        0
    ], [0, 1, 0, 0]
])
```

[25]:

$$\begin{bmatrix} -\frac{\Delta_4\Phi_4}{\Delta_3} & \Psi_4 + \frac{\Delta_4}{\Delta_3} & -\frac{\Delta_4}{\Delta_5} - \Phi_4 & \frac{\Delta_4\Psi_4}{\Delta_5} \\ \frac{2\Delta_4\Phi_4}{\Delta_3} & -\Psi_4 - \frac{2\Delta_4}{\Delta_3} & \frac{\Delta_4}{\Delta_5} + 2\Phi_4 & -\frac{\Delta_4\Psi_4}{\Delta_5} \\ -\frac{\Delta_4\Phi_4}{\Delta_3} & \frac{-\Delta_3+\Delta_4}{\Delta_3} & \frac{\Delta_3}{\Delta_3+\Delta_4} & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

[26]:
```
assert sp.simplify(
    _.subs(phi_psi_subs) - M_CR.expr.subs(deltas)) == sp.Matrix.zeros(4, 4)
```

Just to make sure that $M_{\mathrm{CR},i}$ is consistent with the result from *uniform Catmull–Rom splines* (page 61), let's set all Δ_i to 1:

[27]:
```
uniform = {
    t3: 3,
    t4: 4,
    t5: 5,
    t6: 6,
    M_CR.name: sp.Symbol(r'{M_\text{CR,uniform}}'),
}
```

[28]:
```
M_CR.subs(uniform).pull_out(sp.S.Half)
```

[28]:

$$M_{\text{CR,uniform}} = \frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

doc/euclidean/catmull-rom-non-uniform.ipynb ends here.

The following section was generated from doc/euclidean/catmull-rom-barry-goldman.ipynb

Barry–Goldman Algorithm

The *Barry–Goldman algorithm* (named after *Phillip Barry* and *Ronald Goldman*) can be used to calculate values of *non-uniform Catmull–Rom splines* (page 70). We have also applied this algorithm to *rotation splines* (page 138).

[CR74] describes "a class of local interpolating splines" and [BG88] describes "a recursive evaluation algorithm for a class of Catmull–Rom splines", by which they mean a sub-class of the original class, which only contains splines generated from a combination of *Lagrange interpolation* (page 4) and B-spline blending:

> In particular, they observed that certain choices led to interpolatory curves. Although Catmull and Rom discussed a more general case, we will restrict our attention to an important class of Catmull–Rom splines obtained by combining B-spline basis functions and Lagrange interpolating polynomials. [...] They are piecewise polynomial, have local support, are invariant under affine transformations, and have certain differentiability and interpolatory properties.
>
> —[BG88], section 1: "Introduction"

The algorithm can be set up to construct curves of arbitrary degree (given enough vertices and their parameter values), but here we only take a look at the cubic case (using four vertices), which seems to be what most people mean by the term *Catmull–Rom splines*.

The algorithm is a combination of two sub-algorithms:

> The Catmull–Rom evaluation algorithm is constructed by combining the de Boor algorithm for evaluating B-spline curves with Neville's algorithm for evaluating Lagrange polynomials.
>
> —[BG88], abstract

Combining the two will lead to a multi-stage algorithm, where each stage consists of only linear interpolations (and *extra*polations).

We will use the algorithm here to derive an expression for the *tangent vectors* (page 83), which will show that the algorithm indeed generates *non-uniform Catmull–Rom splines* (page 72).

Triangular Schemes

In [BG88], the presented algorithms are illustrated using triangular evaluation patterns, which we will use here in a very similar form.

As an example, let's look at the most basic building block: linear interpolation between two given points (in this case x_4 and x_5 with corresponding parameter values t_4 and t_5, respectively):

$$\begin{array}{ccc} & p_{4,5} & \\ \frac{t_5-t}{t_5-t_4} & & \frac{t-t_4}{t_5-t_4} \\ x_4 & & x_5 \end{array}$$

The values at the base of the triangle are known, and the triangular scheme shows how the value at the apex can be calculated from them.

In this example, to obtain the *linear* polynomial $p_{4,5}$ one has to add x_4, weighted by the factor shown next to it ($\frac{t_5-t}{t_5-t_4}$), and x_5, weighted by the factor next to it ($\frac{t-t_4}{t_5-t_4}$).

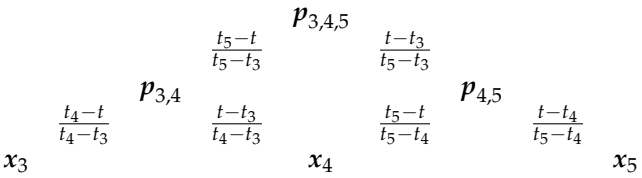
The parameter t can be chosen arbitrarily, but in this example we are mostly interested in the range $t_4 \leq t \leq t_5$. If the parameter value is outside this range, the process is more appropriately called *extra*polation instead of *inter*polation. Since we will need linear interpolation (and extrapolation) quite a few times, let's define a helper function:

[1]:
```
def lerp(xs, ts, t):
    """Linear interpolation.

    Returns the interpolated value at time *t*,
    given the two values *xs* at times *ts*.

    """
    x_begin, x_end = xs
    t_begin, t_end = ts
    return (x_begin * (t_end - t) + x_end * (t - t_begin)) / (t_end - t_begin)
```

Neville's Algorithm

We have already seen this algorithm in our *notebook about Lagrange interpolation* (page 4).

In the *quadratic* case, it looks like this:



The *cubic* case shown in figure 2 of [BG88].

[2]:
```
import matplotlib.pyplot as plt
import numpy as np
```

Let's try to plot this for three points:

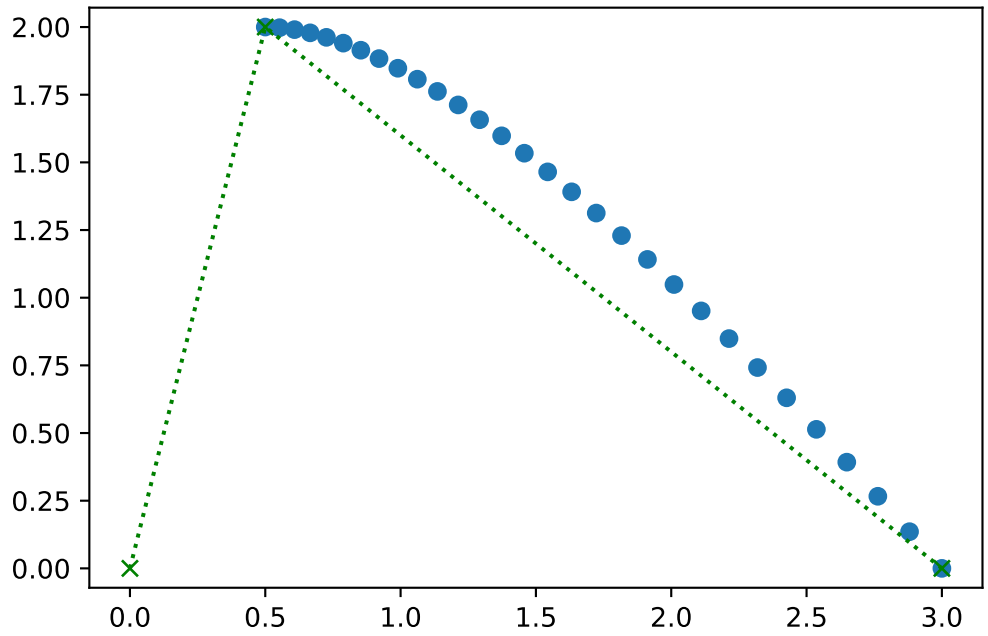
[3]:
```
points = np.array([
    (0, 0),
    (0.5, 2),
    (3, 0),
])
```

In the following example plots we show the *uniform* case (with $t_3 = 3$, $t_4 = 4$ and $t_5 = 5$), but don't worry, the algorithm works just as well for arbitrary non-uniform time values.

[4]:
```
plot_times = np.linspace(4, 5, 30)
```

[5]:
```
plt.scatter(*np.array([
    lerp(
        [lerp(points[:2], [3, 4], t), lerp(points[1:], [4, 5], t)],
        [3, 5], t)
    for t in plot_times]).T)
plt.plot(*points.T, 'x:g')
plt.axis('equal');
```

[5]:
```
(-0.15000000000000002, 3.15, -0.1, 2.1)
```



Note that the quadratic curve is defined by three points but we are only evaluating it between two of them (for $4 \leq t \leq 5$).

De Boor's Algorithm

This algorithm (named after Carl de Boor[13], see [dB72]) can be used to calculate B-spline basis functions.

The quadratic case looks like this:

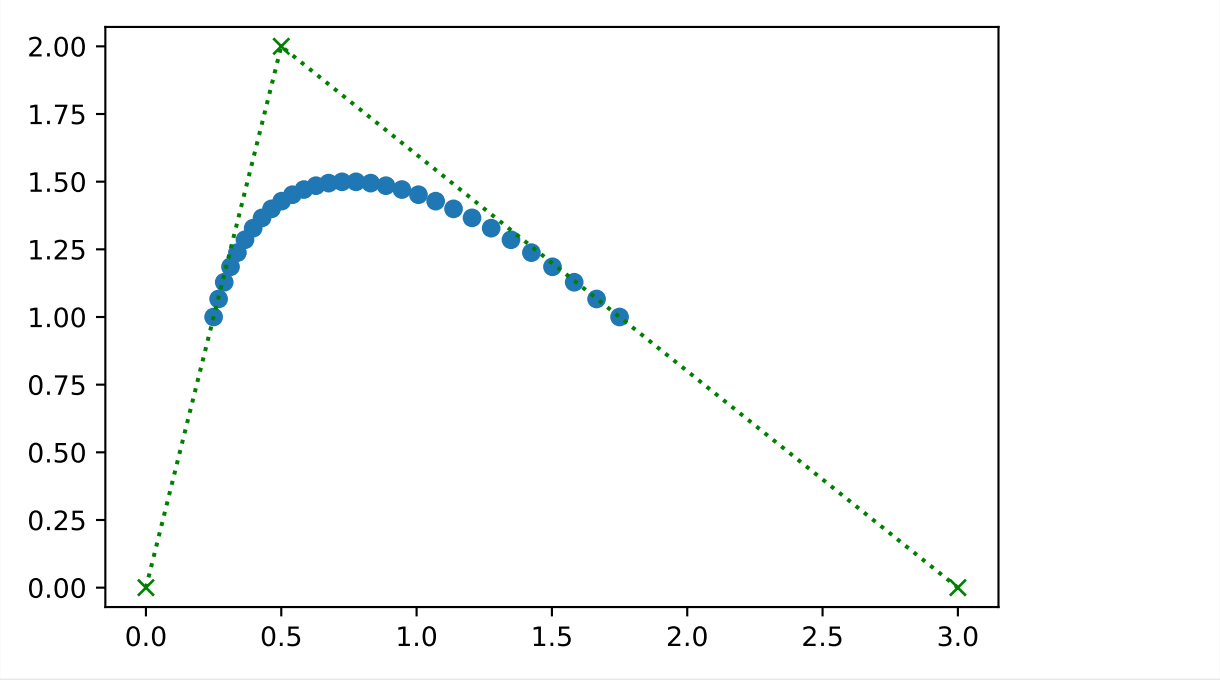
$$
\begin{array}{ccccccccc}
 & & & & \boldsymbol{p}_{3,4,5} & & & & \\
 & & & \frac{t_5-t}{t_5-t_4} & & \frac{t-t_4}{t_5-t_4} & & & \\
 & & \boldsymbol{p}_{3,4} & & & & \boldsymbol{p}_{4,5} & & \\
 & \frac{t_5-t}{t_5-t_3} & & \frac{t-t_3}{t_5-t_3} & & \frac{t_6-t}{t_6-t_4} & & \frac{t-t_4}{t_6-t_4} & \\
\boldsymbol{x}_3 & & & & \boldsymbol{x}_4 & & & & \boldsymbol{x}_5
\end{array}
$$

The *cubic* case shown in figure 1 of [BG88].

[6]:
```
plt.scatter(*np.array([
    lerp(
        [lerp(points[:2], [3, 5], t), lerp(points[1:], [4, 6], t)],
        [4, 5], t)
    for t in plot_times]).T)
plt.plot(*points.T, 'x:g')
plt.axis('equal');
```

[6]:
```
(-0.15000000000000002, 3.15, -0.1, 2.1)
```

[13] https://en.wikipedia.org/wiki/Carl_R._de_Boor



Combining Both Algorithms

Figure 5 of [CR74] shows an example where linear interpolation is followed by quadratic B-spline blending to create a cubic curve.

We can re-create this example with the building blocks from above:

- At the base of the triangle, we put four known vertices.
- Consecutive pairs of these vertices form three linear interpolations (and *extra*polations), resulting in three interpolated (and *extra*polated) values.
- On top of these three values, we arrange a quadratic instance of de Boor's algorithm (as shown above).

This culminates in the final value of the spline (given an appropriate parameter value t) at the apex of the triangle, which looks like this:

$$
\begin{array}{ccccccc}
 & & & p_{3,4,5,6} & & & \\
 & & \frac{t_5-t}{t_5-t_4} & & \frac{t-t_4}{t_5-t_4} & & \\
 & p_{3,4,5} & & & & p_{4,5,6} & \\
 & \frac{t_5-t}{t_5-t_3} & & \frac{t-t_3}{t_5-t_3} & & \frac{t_6-t}{t_6-t_4} & & \frac{t-t_4}{t_6-t_4} \\
p_{3,4} & & p_{4,5} & & & & p_{5,6} & \\
\frac{t_4-t}{t_4-t_3} & \frac{t-t_3}{t_4-t_3} & & \frac{t_5-t}{t_5-t_4} & & \frac{t-t_4}{t_5-t_4} & & \frac{t_6-t}{t_6-t_5} & \frac{t-t_5}{t_6-t_5} \\
x_3 & & x_4 & & & x_5 & & & x_6
\end{array}
$$

Here we are considering the fifth spline segment $p_{3,4,5,6}(t)$ (represented at the apex of the triangle) from x_4 to x_5 (to be found at the base of the triangle) which corresponds to the parameter range $t_4 \leq t \leq t_5$. To calculate the values in this segment, we also need to know the preceding control point x_3 (at the bottom left) and the following control point x_6 (at the bottom right). But not only their positions are relevant, we also need the corresponding parameter values t_3 and t_6, respectively.

This same triangular scheme is also shown in figure 3 of [YSK11], except that here we shifted the indices by $+3$.

Another way to construct a cubic curve with this algorithm would be to flip the degrees of interpolation and blending, in other words:

- Instead of three linear interpolations (and extrapolations), apply two overlapping quadratic Lagrange interpolations using Neville's algorithm (as shown above) to $\boldsymbol{x}_3$, $\boldsymbol{x}_4$, $\boldsymbol{x}_5$ and $\boldsymbol{x}_4$, $\boldsymbol{x}_5$, $\boldsymbol{x}_6$, respectively. Note that the interpolation of $\boldsymbol{x}_4$ and $\boldsymbol{x}_5$ appears in both triangles but has to be calculated only once (see also figures 3 and 4 in [BG88]).
- This will occupy the lower two stages of the triangle, yielding two interpolated values.
- Those two values are then linearly blended in the final stage.

Readers of the *notebook about uniform Catmull–Rom splines* (page 61) may already suspect that, for others it might be a revelation: both ways lead to exactly the same triangular scheme and therefore they are equivalent!

The same scheme, but only for the *uniform* case, is also shown in figure 7 of [BG88], which casually mentions the equivalent cases (with m being the degree of Lagrange interpolation and n being the degree of the B-spline basis functions):

> Note too from Figure 7 that the case $n = 1$, $m = 2$ [...] is identical to the case $n = 2$, $m = 1$ [...]
>
> —[BG88], section 3: "Examples"

Not an Overhauser Spline

Equally casually, they mention:

> Finally, the particular case here is also an Overhauser spline [Ove68].
>
> —[BG88], section 3: "Examples"

This is not true. Overhauser splines – as described in [Ove68] – don't provide a choice of parameter values. The parameter values are determined by the Euclidean distances between control points, similar, but not quite identical to *chordal parameterization* (page 58). Calculating a value of a Catmull–Rom spline doesn't involve calculating any distances.

For completeness' sake, there are two more combinations that lead to cubic splines, but they have their limitations:

- Cubic Lagrange interpolation, followed by no blending at all, which leads to a cubic spline that's not C^1 continuous (only C^0), as shown in figure 8 of [BG88].
- No interpolation at all, followed by cubic B-spline blending, which leads to an approximating spline (instead of an interpolating spline), as shown in figure 5 of [BG88].

Note

Here we are using the time instances of the Lagrange interpolation also as B-spline knots. Equation (9) of [BG88] shows a more generic formulation of the algorithm with separate parameters s_i and t_i.

Step by Step

The triangular figure above looks more complicated than it really is. It's just a bunch of linear *inter*polations and *extra*polations.

Let's go through the figure above, piece by piece.

[7]:
```
import sympy as sp
```

[8]:
```
t = sp.symbols('t')
```

[9]:
```
x3, x4, x5, x6 = sp.symbols('xbm3:7')
```

[10]:
```
t3, t4, t5, t6 = sp.symbols('t3:7')
```

We use some custom SymPy-based tools from utility.py:

[11]:
```
from utility import NamedExpression, NamedMatrix
```

First Stage

In the center of the bottom row, there is a straightforward linear interpolation from x_4 to x_5 within the interval from t_4 to t_5.

[12]:
```
p45 = NamedExpression('pbm_4,5', lerp([x4, x5], [t4, t5], t))
p45
```

[12]:
$$\boldsymbol{p}_{4,5} = \frac{\boldsymbol{x}_4 \left(-t + t_5\right) + \boldsymbol{x}_5 \left(t - t_4\right)}{-t_4 + t_5}$$

Obviously, this starts at:

[13]:
```
p45.evaluated_at(t, t4)
```

[13]:
$$\boldsymbol{p}_{4,5}\Big|_{t=t_4} = \boldsymbol{x}_4$$

... and ends at:

[14]:
```
p45.evaluated_at(t, t5)
```

[14]:
$$\boldsymbol{p}_{4,5}\Big|_{t=t_5} = \boldsymbol{x}_5$$

The bottom left of the triangle looks very similar, with a linear interpolation from x_3 to x_4 within the interval from t_3 to t_4.

[15]:
```
p34 = NamedExpression('pbm_3,4', lerp([x3, x4], [t3, t4], t))
p34
```

[15]:
$$\boldsymbol{p}_{3,4} = \frac{\boldsymbol{x}_3 \left(-t + t_4\right) + \boldsymbol{x}_4 \left(t - t_3\right)}{-t_3 + t_4}$$

However, that's not the parameter range we are interested in! We are interested in the range from t_4 to t_5. Therefore, this is not actually an *inter*polation between x_3 and x_4, but rather a linear *extra*polation starting at x_4 ...

[16]:
```
p34.evaluated_at(t, t4)
```

[16]:
$$p_{3,4}\Big|_{t=t_4} = x_4$$

... and ending at some extrapolated point beyond x_4:

[17]:
```
p34.evaluated_at(t, t5)
```

[17]:
$$p_{3,4}\Big|_{t=t_5} = \frac{x_3\,(t_4 - t_5) + x_4\,(-t_3 + t_5)}{-t_3 + t_4}$$

Similarly, at the bottom right of the triangle there isn't a linear *inter*polation from x_5 to x_6, but rather a linear *extra*polation that just reaches x_5 at the end of the parameter interval (i.e. at $t = t_5$).

[18]:
```
p56 = NamedExpression('pbm_5,6', lerp([x5, x6], [t5, t6], t))
p56
```

[18]:
$$p_{5,6} = \frac{x_5\,(-t + t_6) + x_6\,(t - t_5)}{-t_5 + t_6}$$

[19]:
```
p56.evaluated_at(t, t4)
```

[19]:
$$p_{5,6}\Big|_{t=t_4} = \frac{x_5\,(-t_4 + t_6) + x_6\,(t_4 - t_5)}{-t_5 + t_6}$$

[20]:
```
p56.evaluated_at(t, t5)
```

[20]:
$$p_{5,6}\Big|_{t=t_5} = x_5$$

Second Stage

The second stage of the algorithm involves linear interpolations of the results of the previous stage.

[21]:
```
p345 = NamedExpression('pbm_3,4,5', lerp([p34.name, p45.name], [t3, t5], t))
p345
```

[21]:
$$p_{3,4,5} = \frac{p_{3,4}\,(-t + t_5) + p_{4,5}\,(t - t_3)}{-t_3 + t_5}$$

[22]:
```
p456 = NamedExpression('pbm_4,5,6', lerp([p45.name, p56.name], [t4, t6], t))
p456
```

[22]:
$$p_{4,5,6} = \frac{p_{4,5}\,(-t + t_6) + p_{5,6}\,(t - t_4)}{-t_4 + t_6}$$

Those interpolations are defined over a parameter range from t_3 to t_5 and from t_4 to t_6, respectively. In each case, we are only interested in a sub-range, namely from t_4 to t_5.

These are the start and end points at t_4 and t_5:

[23]:
```
p345.evaluated_at(t, t4, symbols=[p34, p45])
```

[23]:
$$p_{3,4,5}\Big|_{t=t_4} = \frac{p_{3,4}\Big|_{t=t_4}\,(-t_4 + t_5) + p_{4,5}\Big|_{t=t_4}\,(-t_3 + t_4)}{-t_3 + t_5}$$

[24]:
```
p345.evaluated_at(t, t5, symbols=[p34, p45])
```

[24]:
$$p_{3,4,5}\Big|_{t=t_5} = p_{4,5}\Big|_{t=t_5}$$

[25]:
```
p456.evaluated_at(t, t4, symbols=[p45, p56])
```

[25]:
$$p_{4,5,6}\Big|_{t=t_4} = p_{4,5}\Big|_{t=t_4}$$

[26]:
```
p456.evaluated_at(t, t5, symbols=[p45, p56])
```

[26]:
$$p_{4,5,6}\Big|_{t=t_5} = \frac{p_{4,5}\Big|_{t=t_5}(-t_5+t_6) + p_{5,6}\Big|_{t=t_5}(-t_4+t_5)}{-t_4+t_6}$$

Third Stage

The last step is quite simple:

[27]:
```
p3456 = NamedExpression(
    'pbm_3,4,5,6',
    lerp([p345.name, p456.name], [t4, t5], t))
p3456
```

[27]:
$$p_{3,4,5,6} = \frac{p_{3,4,5}(-t+t_5) + p_{4,5,6}(t-t_4)}{-t_4+t_5}$$

This time, the interpolation interval is exactly the one we care about.

To get the final result, we just have to combine all the above expressions:

[28]:
```
p3456 = p3456.subs_symbols(p345, p456, p34, p45, p56).simplify()
p3456
```

[28]:
$$p_{3,4,5,6} = \frac{(t-t_4)(t_3-t_4)(t_3-t_5)(-(t-t_4)(t_4-t_5)(-x_5(t-t_6)+x_6(t-t_5)) + (t-t_6)(t_5-t_6)(-x_4(t-t_5)+x_5(t-t_4))) -}{(t_3-t_4)(t_3-t_5)(t_4 -}$$

We can make this marginally shorter if we rewrite the segment durations as $\Delta_i = t_{i+1} - t_i$:

[29]:
```
delta3, delta4, delta5 = sp.symbols('Delta3:6')
deltas = {
    t4 - t3: delta3,
    t5 - t4: delta4,
    t6 - t5: delta5,
    t5 - t3: delta3 + delta4,
    t6 - t4: delta4 + delta5,
    t6 - t3: delta3 + delta4 + delta5,
    # A few special cases that SymPy has a hard time resolving:
    t4 + t4 - t3: t4 + delta3,
    t6 + t6 - t3: t6 + delta3 + delta4 + delta5,
}
```

[30]:
```
p3456.subs(deltas)
```

[30]:
$$p_{3,4,5,6} = \frac{-\Delta_3(-\Delta_3-\Delta_4)(t-t_4)(\Delta_4(t-t_4)(-x_5(t-t_6)+x_6(t-t_5)) - \Delta_5(t-t_6)(-x_4(t-t_5)+x_5(t-t_4))) + \Delta_5(-\Delta_4-\Delta}{\Delta_3\Delta_4^2\Delta_5(-\Delta_3-\Delta_4)(-\Delta_4-\Delta_5}$$

Apart from checking if it's really cubic ...

[31]:
```
sp.degree(p3456.expr, t)
```

[31]:
```
3
```

… and if it's really interpolating …

[32]:
```
p3456.evaluated_at(t, t4).simplify()
```

[32]:
$$p_{3,4,5,6}\Big|_{t=t_4} = x_4$$

[33]:
```
p3456.evaluated_at(t, t5).simplify()
```

[33]:
$$p_{3,4,5,6}\Big|_{t=t_5} = x_5$$

… the only thing left to do is to check its …

Tangent Vectors

To get the tangent vectors at the control points, we just have to take the first derivative …

[34]:
```
pd3456 = p3456.diff(t)
```

… and evaluate it at t_4 and t_5:

[35]:
```
pd3456.evaluated_at(t, t4).simplify().simplify()
```

[35]:
$$\frac{d}{dt}p_{3,4,5,6}\Big|_{t=t_4} = \frac{(t_3-t_4)^2(x_4-x_5)+(t_4-t_5)^2(x_3-x_4)}{(t_3-t_4)(t_3-t_5)(t_4-t_5)}$$

[36]:
```
pd3456.evaluated_at(t, t5).simplify()
```

[36]:
$$\frac{d}{dt}p_{3,4,5,6}\Big|_{t=t_5} = \frac{(t_4-t_5)^2(x_5-x_6)+(t_5-t_6)^2(x_4-x_5)}{(t_4-t_5)(t_4-t_6)(t_5-t_6)}$$

If all went well, this should be identical to the result in *the notebook about non-uniform Catmull–Rom splines* (page 72).

Animation

The linear interpolations (and *extra*polations) of this algorithm can be shown graphically.

By means of the file barry_goldman.py, we can generate animations of the algorithm:

[37]:
```
from barry_goldman import animation
```

[38]:
```
from IPython.display import HTML
```

[39]:
```
vertices = [
    (0, 0),
    (0.5, 1),
    (6, 1),
    (6, 2),
]
```

[40]:
```
times = [
    0,
    1,
    5,
    7,
]
```

[41]:
```
ani = animation(vertices, times)
```

[42]:
```
HTML(ani.to_jshtml(default_mode='reflect'))
```

[42]:
```
<IPython.core.display.HTML object>
```

doc/euclidean/catmull-rom-barry-goldman.ipynb ends here.

1.7 Kochanek–Bartels Splines

Kochanek–Bartels splines (a.k.a. TCB splines) are named after *Doris Kochanek* and *Richard Bartels* (more specifically, after their paper [KB84]).

A Python implementation if available in the *splines.KochanekBartels* (page 151) class.

The following section was generated from doc/euclidean/kochanek-bartels-properties.ipynb

Properties of Kochanek–Bartels Splines

Kochanek–Bartels splines (a.k.a. TCB splines) are interpolating cubic polynomial splines, with three user-defined parameters per vertex (of course they can also be chosen to be the same three values for the whole spline), which can be used to change the shape and velocity of the spline.

These three parameters are called T for "tension", C for "continuity" and B for "bias". With the default values of $C = 0$ and $B = 0$, a Kochanek–Bartels spline is identical to a *cardinal spline*. If the "tension" parameter also has its default value $T = 0$, it is also identical to a *Catmull–Rom spline* (page 61).

TODO: comparison of T with "tension" parameter of cardinal splines

[1]:
```
import splines
```

helper.py

[2]:
```
from helper import plot_spline_2d
```

[3]:
```
import matplotlib.pyplot as plt
import numpy as np

def plot_tcb(*tcb, ax=None):
    """Plot four TCB examples."""
    if ax is None:
        ax = plt.gca()
    vertices = [
        (-2.5, 0),
        (-1, 1.5),
        (0, 0.1),
        (1, 1.5),
        (2.5, 0),
        (1, -1.5),
        (0, -0.1),
        (-1, -1.5),
```

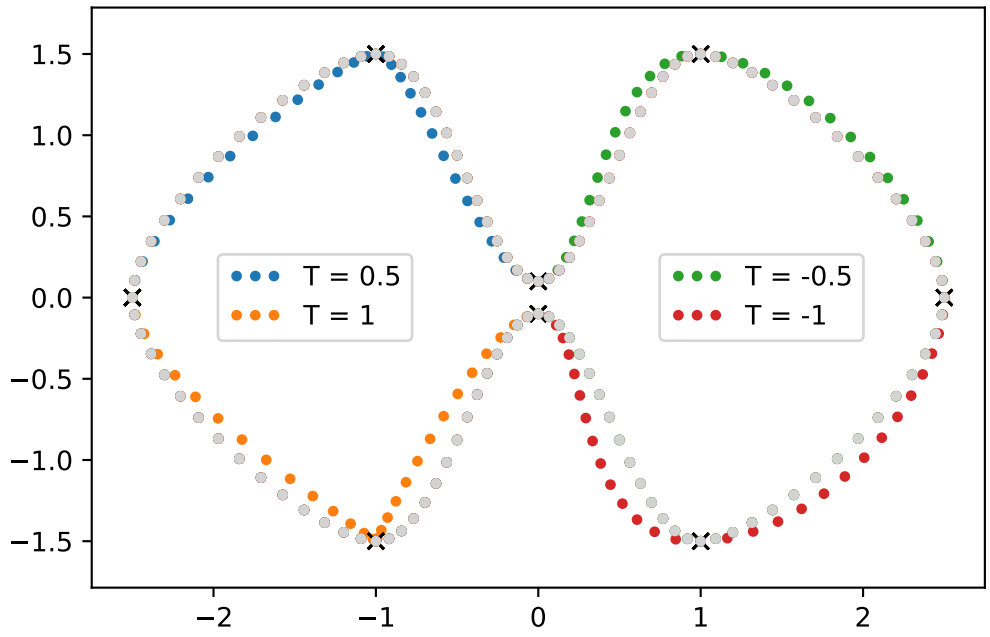
(continues on next page)

(continued from previous page)

```
    ]
    for idx, tcb in zip([1, 7, 3, 5], tcb):
        all_tcb = np.zeros((len(vertices), 3))
        all_tcb[idx] = tcb
        s = splines.KochanekBartels(
            vertices, tcb=all_tcb, endconditions='closed')
        label = ', '.join(
            f'{name} = {value}'
            for name, value in zip('TCB', tcb)
            if value)
        plot_spline_2d(s, chords=False, label=label, ax=ax)
    plot_spline_2d(
        splines.KochanekBartels(vertices, endconditions='closed'),
        color='lightgrey', chords=False, ax=ax)
    lines = [l for l in ax.get_lines() if not l.get_label().startswith('_')]
    # https://matplotlib.org/tutorials/intermediate/legend_guide.html#multiple-legends-on-
↪the-same-axes
    ax.add_artist(ax.legend(
        handles=lines[:2], bbox_to_anchor=(0, 0., 0.5, 1),
        loc='center', numpoints=3))
    ax.legend(
        handles=lines[2:], bbox_to_anchor=(0.5, 0., 0.5, 1),
        loc='center', numpoints=3)
```

Tension

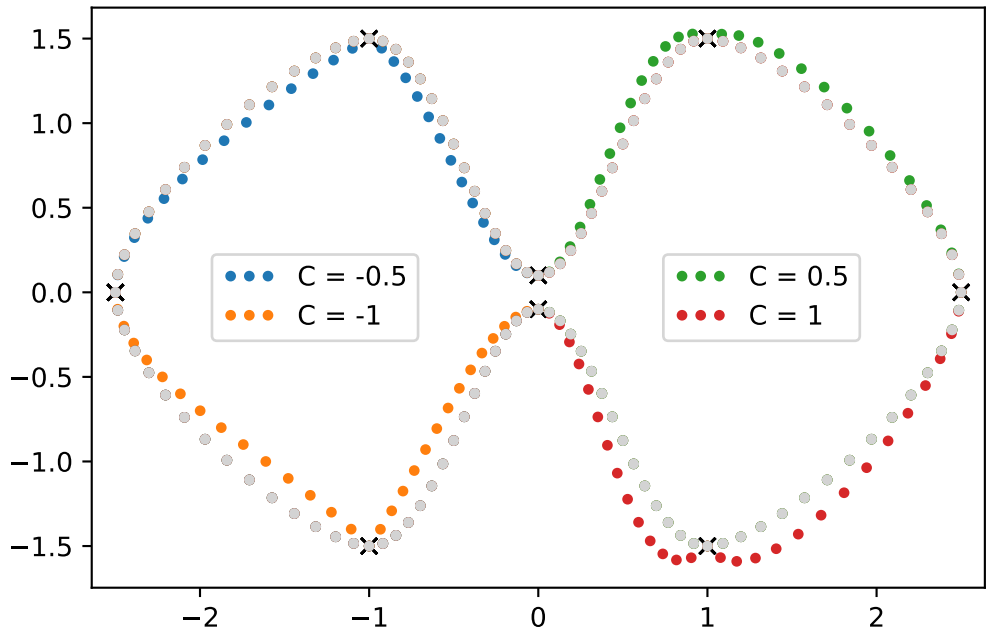
[4]:
```
plot_tcb((0.5, 0, 0), (1, 0, 0), (-0.5, 0, 0), (-1, 0, 0))
```



Continuity

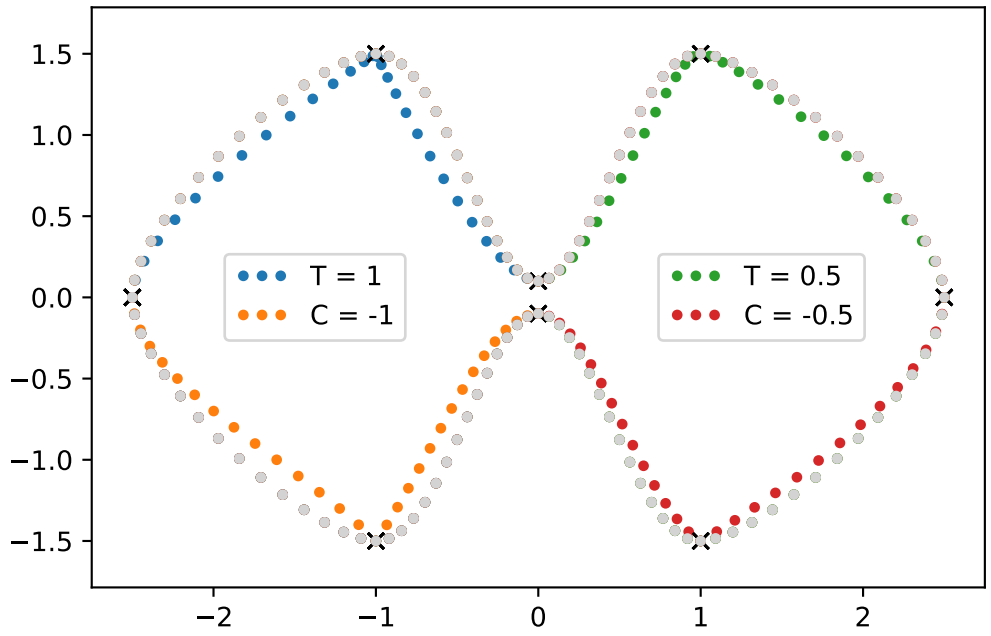
TODO: When $C_i = 0$, we are back at a Catmull–Rom spline. When $C_i = -1$, we get a tangent like in a piecewise linear curve. When $C_i = 1$, we get some weird "inverse corners".

[5]:
```
plot_tcb((0, -0.5, 0), (0, -1, 0), (0, 0.5, 0), (0, 1, 0))
```



$T = 1$ and $C = -1$: similar shape (a.k.a. "image"), different timing:

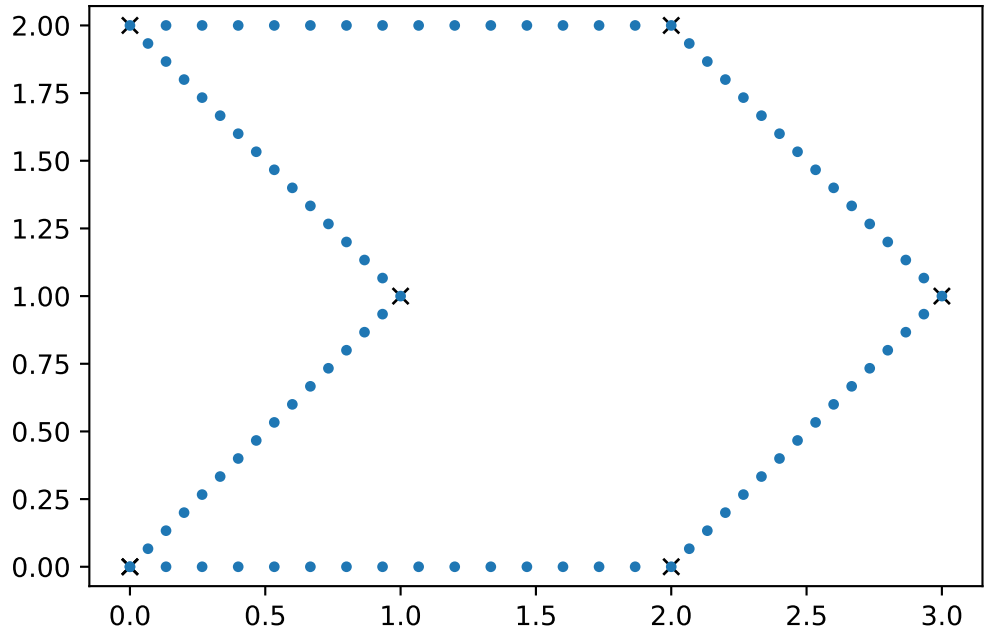
[6]:
```
plot_tcb((1, 0, 0), (0, -1, 0), (0.5, 0, 0), (0, -0.5, 0))
```



shape in "corners" is similar, but speed is different! with re-parameterization (TODO: link), it doesn't make too much of a difference.

A value of $C = -1$ on adjacent vertices leads to linear segments:

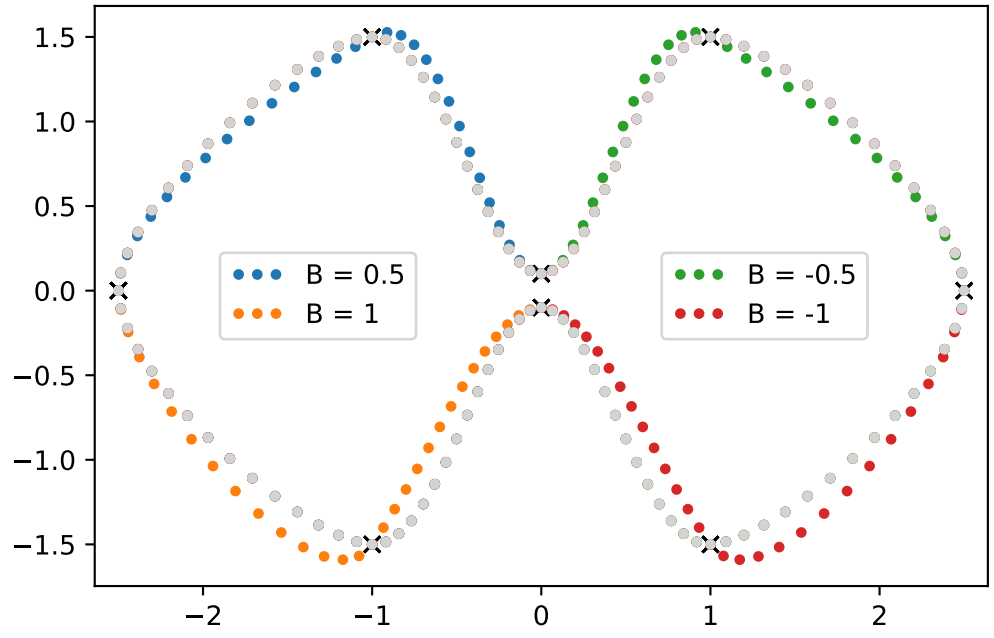
[7]:
```
vertices1 = [(0, 0), (1, 1), (0, 2), (2, 2), (3, 1), (2, 0)]
s1 = splines.KochanekBartels(vertices1, tcb=(0, -1, 0), endconditions='closed')
plot_spline_2d(s1, chords=False)
```



Bias

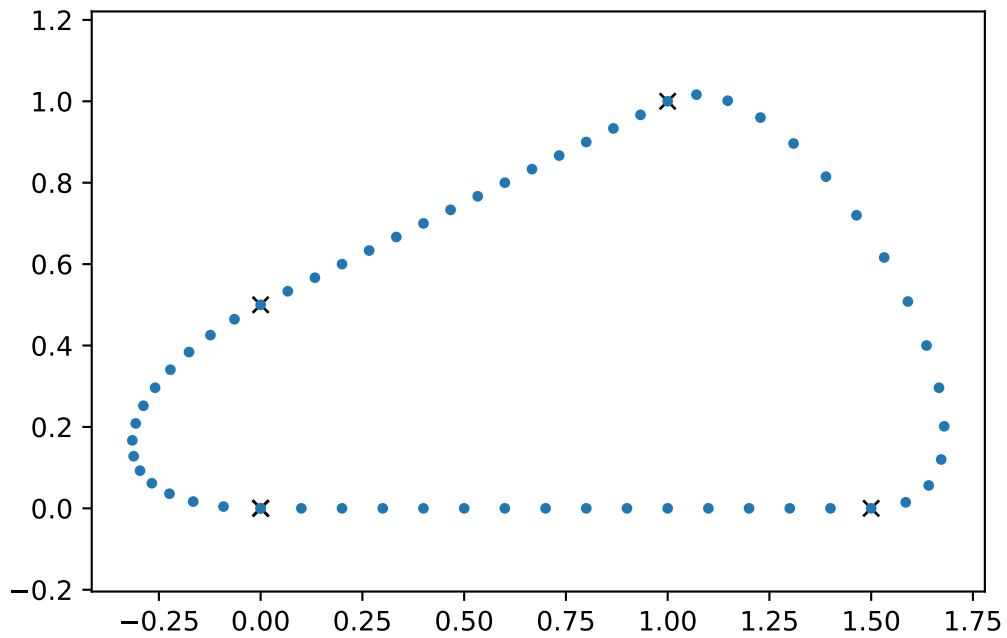
"overshoot": -1 is full "undershoot"

[8]:
```
plot_tcb((0, 0, 0.5), (0, 0, 1), (0, 0, -0.5), (0, 0, -1))
```



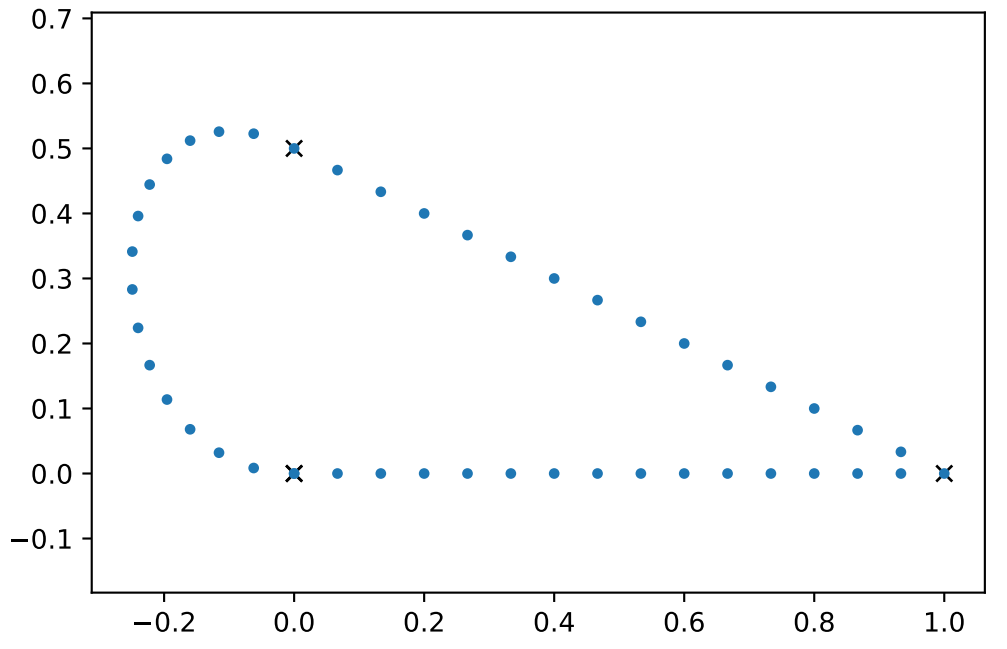
Bias -1 followed by $+1$ can be used to achieve linear segments between two control points:

[9]:
```
vertices2 = [(0, 0), (1.5, 0), (1, 1), (0, 0.5)]
tcb2 = [(0, 0, -1), (0, 0, 1), (0, 0, -1), (0, 0, 1)]
s2 = splines.KochanekBartels(vertices2, tcb=tcb2, endconditions='closed')
plot_spline_2d(s2, chords=False)
```



A sequence of $B = -1$, $C = -1$ and $B = +1$ can be used to get two adjacent linear segments:

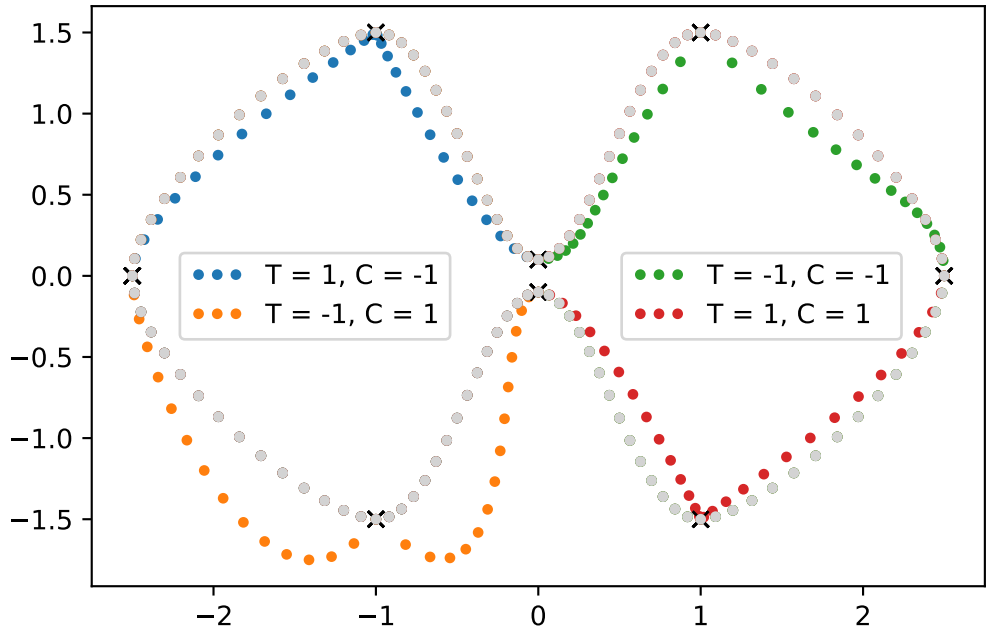
[10]:
```
vertices3 = [(0, 0), (1, 0), (0, 0.5)]
tcb3 = [(0, 0, -1), (0, -1, 0), (0, 0, 1)]
s3 = splines.KochanekBartels(vertices3, tcb=tcb3, endconditions='closed')
plot_spline_2d(s3, chords=False)
```



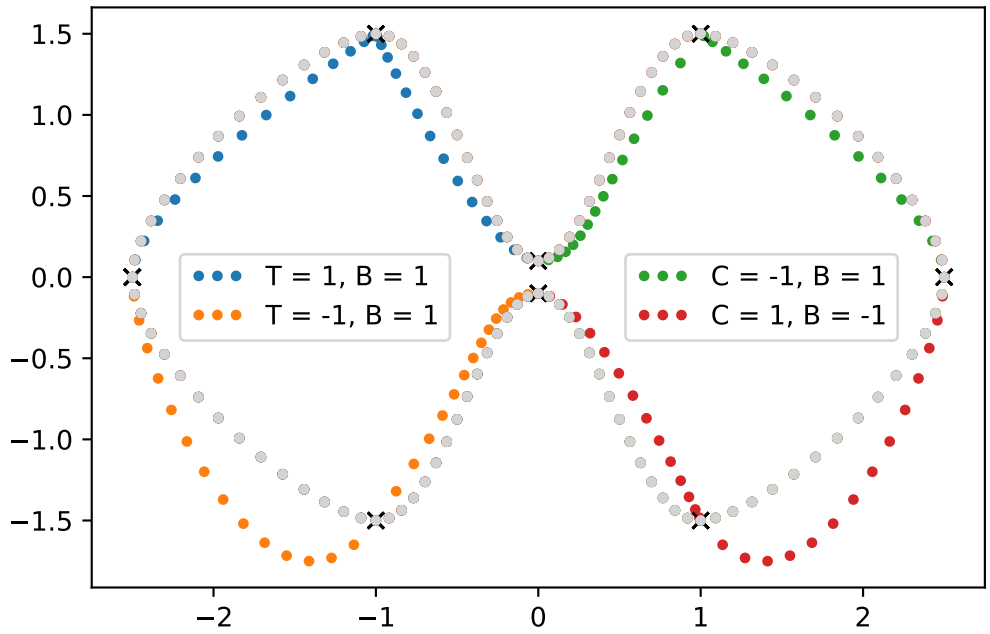
Combinations

accumulated tension and continuity vs. opposite T and C:

[11]:
```
plot_tcb((1, -1, 0), (-1, 1, 0), (-1, -1, 0), (1, 1, 0))
```



[12]:
```
plot_tcb((1, 0, 1), (-1, 0, 1), (0, -1, 1), (0, 1, -1))
```



TODO: expain non-intuitive cases

doc/euclidean/kochanek-bartels-properties.ipynb ends here.

Uniform Kochanek–Bartels Splines

As a starting point, remember the tangent vectors from *Catmull-Rom splines* (page 52):

$$\dot{\boldsymbol{x}}_i = \frac{(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2}$$

Parameters

TCB splines are all about inserting the parameters T, C and B into this equation.

Tension

see equation 4 in [KB84]

$$\dot{\boldsymbol{x}}_i = (1 - T_i)\frac{(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2}$$

Continuity

Up to now, the goal was to have a continuous first derivative at the control points, i.e. the incoming and outgoing tangent vectors were identical:

$$\dot{\boldsymbol{x}}_i = \dot{\boldsymbol{x}}_i^{(-)} = \dot{\boldsymbol{x}}_i^{(+)}$$

This also happens to be the requirement for a spline to be C^1 continuous.

The "continuity" parameter allows us to break this continuity if we so desire, leading to different incoming and outgoing tangent vectors (see equations 5 and 6 in [KB84]):

$$\dot{\boldsymbol{x}}_i^{(-)} = \frac{(1 - C_i)(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (1 + C_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2}$$
$$\dot{\boldsymbol{x}}_i^{(+)} = \frac{(1 + C_i)(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (1 - C_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2}$$

Bias

see equation 7 in [KB84]

$$\dot{\boldsymbol{x}}_i = \frac{(1 + B_i)(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (1 - B_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2}$$

All Three Combined

see equations 8 and 9 in [KB84]

$$\dot{\boldsymbol{x}}_i^{(+)} = \frac{(1 - T_i)(1 + C_i)(1 + B_i)(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (1 - T_i)(1 - C_i)(1 - B_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2}$$
$$\dot{\boldsymbol{x}}_i^{(-)} = \frac{(1 - T_i)(1 - C_i)(1 + B_i)(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (1 - T_i)(1 + C_i)(1 - B_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2}$$

Note: There is an error in equation (6.11) of [Mil] (all subscripts of x are wrong, most likely copy-pasted from the preceding equation).

To simplify the result we will get later, we introduce the following shorthands (as suggested in [Mil]):

$$
\begin{aligned}
a_i &= (1 - T_i)(1 + C_i)(1 + B_i) \\
b_i &= (1 - T_i)(1 - C_i)(1 - B_i) \\
c_i &= (1 - T_i)(1 - C_i)(1 + B_i) \\
d_i &= (1 - T_i)(1 + C_i)(1 - B_i)
\end{aligned}
$$

This leads to the simplified equations

$$
\begin{aligned}
\dot{\boldsymbol{x}}_i^{(+)} &= \frac{a_i(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + b_i(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2} \\
\dot{\boldsymbol{x}}_i^{(-)} &= \frac{c_i(\boldsymbol{x}_i - \boldsymbol{x}_{i-i}) + d_i(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2}
\end{aligned}
$$

Calculation

[1]:
```
import sympy as sp
sp.init_printing()
```

[2]:
```
from utility import NamedExpression, NamedMatrix
```

helper.py

[3]:
```
from helper import plot_basis
```

Same control values as Catmull-Rom ...

[4]:
```
x3, x4, x5, x6 = sp.symbols('xbm3:7')
```

[5]:
```
control_values_KB = sp.Matrix([x3, x4, x5, x6])
control_values_KB
```

[5]:
$$\begin{bmatrix} \boldsymbol{x}_3 \\ \boldsymbol{x}_4 \\ \boldsymbol{x}_5 \\ \boldsymbol{x}_6 \end{bmatrix}$$

... but three additional parameters per vertex. In our calculation, the parameters belonging to $\boldsymbol{x}_4$ and $\boldsymbol{x}_5$ are relevant:

[6]:
```
T4, T5 = sp.symbols('T4 T5')
C4, C5 = sp.symbols('C4 C5')
B4, B5 = sp.symbols('B4 B5')
```

[7]:
```
a4 = NamedExpression('a4', (1 - T4) * (1 + C4) * (1 + B4))
b4 = NamedExpression('b4', (1 - T4) * (1 - C4) * (1 - B4))
c5 = NamedExpression('c5', (1 - T5) * (1 - C5) * (1 + B5))
d5 = NamedExpression('d5', (1 - T5) * (1 + C5) * (1 - B5))
display(a4, b4, c5, d5)
```

$a_4 = (1 - T_4)\,(B_4 + 1)\,(C_4 + 1)$

$b_4 = (1 - B_4)\,(1 - C_4)\,(1 - T_4)$

$c_5 = (1 - C_5)\,(1 - T_5)\,(B_5 + 1)$

$d_5 = (1 - B_5)\,(1 - T_5)\,(C_5 + 1)$

[8]:
```
xd4 = NamedExpression(
    'xdotbm4^(+)',
    sp.S.Half * (a4.name * (x4 - x3) + b4.name * (x5 - x4)))
xd5 = NamedExpression(
    'xdotbm5^(-)',
    sp.S.Half * (c5.name * (x5 - x4)  + d5.name * (x6 - x5)))
display(xd4, xd5)
```

$$\dot{x}_4^{(+)} = \frac{a_4 \left(-x_3 + x_4\right)}{2} + \frac{b_4 \left(-x_4 + x_5\right)}{2}$$

$$\dot{x}_5^{(-)} = \frac{c_5 \left(-x_4 + x_5\right)}{2} + \frac{d_5 \left(-x_5 + x_6\right)}{2}$$

[9]:
```
display(xd4.subs_symbols(a4, b4))
display(xd5.subs_symbols(c5, d5))
```

$$\dot{x}_4^{(+)} = \frac{\left(1 - B_4\right) \left(1 - C_4\right) \left(1 - T_4\right) \left(-x_4 + x_5\right)}{2} + \frac{\left(1 - T_4\right) \left(B_4 + 1\right) \left(C_4 + 1\right) \left(-x_3 + x_4\right)}{2}$$

$$\dot{x}_5^{(-)} = \frac{\left(1 - B_5\right) \left(1 - T_5\right) \left(C_5 + 1\right) \left(-x_5 + x_6\right)}{2} + \frac{\left(1 - C_5\right) \left(1 - T_5\right) \left(B_5 + 1\right) \left(-x_4 + x_5\right)}{2}$$

Same as with Catmull-Rom, try to find a transformation from cardinal control values to Hermite control values. This can be used to get the full basis matrix.

[10]:
```
control_values_H = sp.Matrix([x4, x5, xd4.name, xd5.name])
control_values_H
```

[10]:
$$\begin{bmatrix} x_4 \\ x_5 \\ \dot{x}_4^{(+)} \\ \dot{x}_5^{(-)} \end{bmatrix}$$

From the *notebook about uniform Hermite splines* (page 15):

[11]:
```
M_H = NamedMatrix(
    r'{M_\text{H}}',
    sp.Matrix([[2, -2, 1, 1],
               [-3, 3, -2, -1],
               [0, 0, 1, 0],
               [1, 0, 0, 0]]))
M_H
```

[11]:
$$M_{\mathrm{H}} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

[12]:
```
M_KBtoH = NamedMatrix(r'{M_{\text{KB$,4\to$H}}}', 4, 4)
M_KB = NamedMatrix(r'{M_{\text{KB},4}}', M_H.name * M_KBtoH.name)
M_KB
```

[12]:
$$M_{\mathrm{KB},4} = M_{\mathrm{H}} M_{\mathrm{KB},4 \to \mathrm{H}}$$

[13]:
```
NamedMatrix(control_values_H, M_KBtoH.name * control_values_KB)
```

[13]:
$$\begin{bmatrix} x_4 \\ x_5 \\ \dot{x}_4^{(+)} \\ \dot{x}_5^{(-)} \end{bmatrix} = M_{\mathrm{KB},4\to\mathrm{H}} \begin{bmatrix} x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

If we substitute the above definitions of $\dot{x}_4$ and $\dot{x}_5$, we can directly read off the matrix elements:

[14]:
```
M_KBtoH.expr = sp.Matrix([
    [expr.coeff(cv) for cv in control_values_KB]
    for expr in control_values_H.subs([xd4.args, xd5.args]).expand()])
M_KBtoH.pull_out(sp.S.Half)
```

[14]:
$$M_{\mathrm{KB},4\to\mathrm{H}} = \frac{1}{2}\begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ -a_4 & a_4 - b_4 & b_4 & 0 \\ 0 & -c_5 & c_5 - d_5 & d_5 \end{bmatrix}$$

[15]:
```
M_KB = M_KB.subs_symbols(M_H, M_KBtoH).doit()
M_KB.pull_out(sp.S.Half)
```

[15]:
$$M_{\mathrm{KB},4} = \frac{1}{2}\begin{bmatrix} -a_4 & a_4 - b_4 - c_5 + 4 & b_4 + c_5 - d_5 - 4 & d_5 \\ 2a_4 & -2a_4 + 2b_4 + c_5 - 6 & -2b_4 - c_5 + d_5 + 6 & -d_5 \\ -a_4 & a_4 - b_4 & b_4 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

And for completeness' sake, its inverse:

[16]:
```
M_KB.I
```

[16]:
$${M_{\mathrm{KB},4}}^{-1} = \begin{bmatrix} \frac{b_4}{a_4} & \frac{b_4}{a_4} & \frac{b_4-2}{a_4} & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ \frac{-c_5+d_5+6}{d_5} & \frac{-c_5+d_5+4}{d_5} & \frac{-c_5+d_5+2}{d_5} & 1 \end{bmatrix}$$

[17]:
```
t = sp.symbols('t')
```

[18]:
```
b_KB = NamedMatrix(r'{b_{\text{KB},4}}', sp.Matrix([t**3, t**2, t, 1]).T * M_KB.expr)
b_KB.T.pull_out(sp.S.Half)
```

[18]:
$${b_{\mathrm{KB},4}}^T = \frac{1}{2}\begin{bmatrix} a_4 t\left(-t^2 + 2t - 1\right) \\ t^3\left(a_4 - b_4 - c_5 + 4\right) + t^2\left(-2a_4 + 2b_4 + c_5 - 6\right) + t\left(a_4 - b_4\right) + 2 \\ t\left(b_4 + t^2\left(b_4 + c_5 - d_5 - 4\right) + t\left(-2b_4 - c_5 + d_5 + 6\right)\right) \\ d_5 t^2\left(t - 1\right) \end{bmatrix}$$

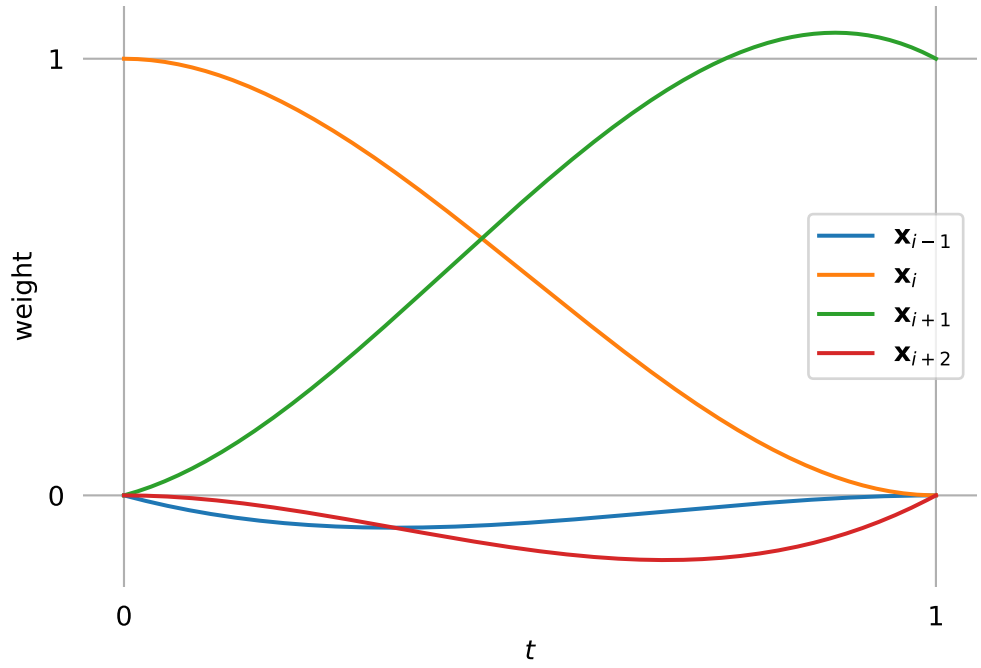
To be able to plot the basis functions, let's substitute a_4, b_4, c_5 and d_5 back in (which isn't pretty):

[19]:
```
b_KB = b_KB.subs_symbols(a4, b4, c5, d5).simplify()
b_KB.T.pull_out(sp.S.Half)
```

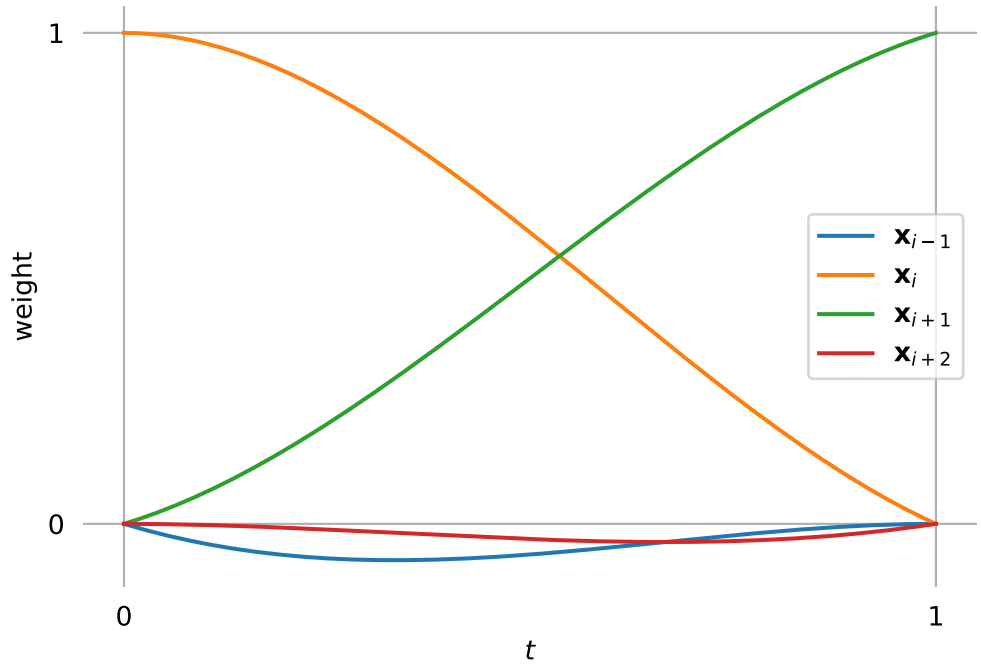
[19]:
$${b_{\mathrm{KB},4}}^T =$$
$$\frac{1}{2}\begin{bmatrix} t^3\left(\left(B_4 - 1\right)\left(C_4 - 1\right)\left(T_4 - 1\right) - \left(B_4 + 1\right)\left(C_4 + 1\right)\left(T_4 - 1\right) - \left(B_5 + 1\right)\left(C_5 - 1\right)\left(T_5 - 1\right) + 4\right) + t^2\left(-2\left(B_4 - 1\right)\left(C_4 - 1\right) \\ -t\left(t^2\left(\left(B_4 - 1\right)\left(C_4 - 1\right)\left(T_4 - 1\right) + \left(B_5 - 1\right)\left(C_5 + 1\right)\left(T_5 - 1\right) - \left(B_5 + 1\right)\left(C_5 - 1\right)\left(T_5 - 1\right) + 4\right) - t\left(2\left(B \\ t^2\left(B_5 - 1\right)\left(\end{bmatrix}$$

[20]:
```
labels = sp.symbols('xbm_i-1 xbm_i xbm_i+1 xbm_i+2')
```

[21]:
```
plot_basis(
    *b_KB.expr.subs({T4: 0, T5: 0, C4: 0, C5: 1, B4: 0, B5: 0}),
    labels=labels)
```



[22]:
```
plot_basis(
    *b_KB.expr.subs({T4: 0, T5: 0, C4: 0, C5: -0.5, B4: 0, B5: 0}),
    labels=labels)
```



TODO: plot some example curves

doc/euclidean/kochanek-bartels-uniform.ipynb ends here.

The following section was generated from doc/euclidean/kochanek-bartels-non-uniform.ipynb

Non-Uniform Kochanek–Bartels Splines

[KB84] mainly talks about uniform splines. Only in section 4, "Adjustments for Parameter Step Size", they briefly mention the non-uniform case.

TODO: show equations for adjusted tangents

Unfortunately, this is wrong.

TODO: show why it is wrong.

Instead, we should start from the correct tangent vector for non-uniform Catmull–Rom splines:

$$\dot{\boldsymbol{x}}_i = \frac{(t_{i+1} - t_i)^2(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (t_i - t_{i-1})^2(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{(t_{i+1} - t_i)(t_i - t_{i-1})(t_{i+1} - t_{i-1})}$$

Parameters

In general incoming tangent $\dot{\boldsymbol{x}}_i^{(-)}$ and outgoing tangent $\dot{\boldsymbol{x}}_i^{(+)}$ at vertex $\boldsymbol{x}_i$:

$$\begin{aligned}
a_i &= (1 - T_i)(1 + C_i)(1 + B_i) \\
b_i &= (1 - T_i)(1 - C_i)(1 - B_i) \\
c_i &= (1 - T_i)(1 - C_i)(1 + B_i) \\
d_i &= (1 - T_i)(1 + C_i)(1 - B_i)
\end{aligned}$$

$$\begin{aligned}
\dot{\boldsymbol{x}}_i^{(+)} &= \frac{a_i(t_{i+1} - t_i)^2(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + b_i(t_i - t_{i-1})^2(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{(t_{i+1} - t_i)(t_i - t_{i-1})(t_{i+1} - t_{i-1})} \\
\dot{\boldsymbol{x}}_i^{(-)} &= \frac{c_i(t_{i+1} - t_i)^2(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + d_i(t_i - t_{i-1})^2(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{(t_{i+1} - t_i)(t_i - t_{i-1})(t_{i+1} - t_{i-1})}
\end{aligned}$$

In the calculation below, we consider the outgoing tangent at $\boldsymbol{x}_4$ and the incoming tangent at $\boldsymbol{x}_5$.

$$\begin{aligned}
a_4 &= (1 - T_4)(1 + C_4)(1 + B_4) \\
b_4 &= (1 - T_4)(1 - C_4)(1 - B_4) \\
c_5 &= (1 - T_5)(1 - C_5)(1 + B_5) \\
d_5 &= (1 - T_5)(1 + C_5)(1 - B_5)
\end{aligned}$$

$$\begin{aligned}
\dot{\boldsymbol{x}}_4^{(+)} &= \frac{a_4(t_5 - t_4)^2(\boldsymbol{x}_4 - \boldsymbol{x}_3) + b_4(t_4 - t_3)^2(\boldsymbol{x}_5 - \boldsymbol{x}_4)}{(t_5 - t_4)(t_4 - t_3)(t_5 - t_3)} \\
\dot{\boldsymbol{x}}_5^{(-)} &= \frac{c_5(t_6 - t_5)^2(\boldsymbol{x}_5 - \boldsymbol{x}_4) + d_5(t_5 - t_4)^2(\boldsymbol{x}_6 - \boldsymbol{x}_5)}{(t_6 - t_5)(t_5 - t_4)(t_6 - t_4)}
\end{aligned}$$

Calculation

[1]:
```
import sympy as sp
sp.init_printing()
```

[2]:
```
from utility import NamedExpression, NamedMatrix
```

[3]:
```
x3, x4, x5, x6 = sp.symbols('xbm3:7')
```

[4]:
```
t, t3, t4, t5, t6 = sp.symbols('t t3:7')
```

Same control values as Catmull-Rom ...

[5]:
```
control_values_KB = sp.Matrix([x3, x4, x5, x6])
control_values_KB
```

[5]:
$$\begin{bmatrix} x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

... but three additional parameters per vertex. In our calculation, the parameters belonging to x_4 and x_5 are relevant:

[6]:
```
T4, T5 = sp.symbols('T4 T5')
C4, C5 = sp.symbols('C4 C5')
B4, B5 = sp.symbols('B4 B5')
```

[7]:
```
a4 = NamedExpression('a4', (1 - T4) * (1 + C4) * (1 + B4))
b4 = NamedExpression('b4', (1 - T4) * (1 - C4) * (1 - B4))
c5 = NamedExpression('c5', (1 - T5) * (1 - C5) * (1 + B5))
d5 = NamedExpression('d5', (1 - T5) * (1 + C5) * (1 - B5))
display(a4, b4, c5, d5)
```

$$a_4 = (1 - T_4)\,(B_4 + 1)\,(C_4 + 1)$$

$$b_4 = (1 - B_4)\,(1 - C_4)\,(1 - T_4)$$

$$c_5 = (1 - C_5)\,(1 - T_5)\,(B_5 + 1)$$

$$d_5 = (1 - B_5)\,(1 - T_5)\,(C_5 + 1)$$

[8]:
```
xd4 = NamedExpression(
    'xdotbm4^(+)',
    (a4.name * (t5 - t4)**2 * (x4 - x3) + b4.name * (t4 - t3)**2 * (x5 - x4)) /
    ((t5 - t4) * (t4 - t3) * (t5 - t3)))
xd5 = NamedExpression(
    'xdotbm5^(-)',
    (c5.name * (t6 - t5)**2 * (x5 - x4) + d5.name * (t5 - t4)**2 * (x6 - x5)) /
    ((t6 - t5) * (t5 - t4) * (t6 - t4)))
display(xd4, xd5)
```

$$\dot{x}_4^{(+)} = \frac{a_4\,(-t_4 + t_5)^2\,(-x_3 + x_4) + b_4\,(-t_3 + t_4)^2\,(-x_4 + x_5)}{(-t_3 + t_4)\,(-t_3 + t_5)\,(-t_4 + t_5)}$$

$$\dot{x}_5^{(-)} = \frac{c_5\,(-t_5 + t_6)^2\,(-x_4 + x_5) + d_5\,(-t_4 + t_5)^2\,(-x_5 + x_6)}{(-t_4 + t_5)\,(-t_4 + t_6)\,(-t_5 + t_6)}$$

[9]:
```
display(xd4.subs_symbols(a4, b4))
display(xd5.subs_symbols(c5, d5))
```

$$\dot{x}_4^{(+)} = \frac{(1 - B_4)\,(1 - C_4)\,(1 - T_4)\,(-t_3 + t_4)^2\,(-x_4 + x_5) + (1 - T_4)\,(B_4 + 1)\,(C_4 + 1)\,(-t_4 + t_5)^2\,(-x_3 + x_4)}{(-t_3 + t_4)\,(-t_3 + t_5)\,(-t_4 + t_5)}$$

$$\dot{x}_5^{(-)} = \frac{(1 - B_5)\,(1 - T_5)\,(C_5 + 1)\,(-t_4 + t_5)^2\,(-x_5 + x_6) + (1 - C_5)\,(1 - T_5)\,(B_5 + 1)\,(-t_5 + t_6)^2\,(-x_4 + x_5)}{(-t_4 + t_5)\,(-t_4 + t_6)\,(-t_5 + t_6)}$$

Same as with Catmull-Rom, try to find a transformation from cardinal control values to Hermite control values. This can be used to get the full basis matrix.

[10]:
```
control_values_H = sp.Matrix([x4, x5, xd4.name, xd5.name])
control_values_H
```

[10]:
$$\begin{bmatrix} x_4 \\ x_5 \\ \dot{x}_4^{(+)} \\ \dot{x}_5^{(-)} \end{bmatrix}$$

From the *notebook about non-uniform Hermite splines* (page 24):

[11]:
```
M_H = NamedMatrix(
    r'{M_{\text{H},4}}',
    sp.Matrix([[2, -2, 1, 1],
               [-3, 3, -2, -1],
               [0, 0, 1, 0],
               [1, 0, 0, 0]]) * sp.diag(1, 1, t5 - t4, t5 - t4))
M_H
```

[11]:
$$M_{\mathrm{H},4} = \begin{bmatrix} 2 & -2 & -t_4+t_5 & -t_4+t_5 \\ -3 & 3 & 2t_4-2t_5 & t_4-t_5 \\ 0 & 0 & -t_4+t_5 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

[12]:
```
M_KBtoH = NamedMatrix(r'{M_{\text{KB$,4\to$H}}}', 4, 4)
M_KB = NamedMatrix(r'{M_{\text{KB},4}}', M_H.name * M_KBtoH.name)
M_KB
```

[12]:
$$M_{\mathrm{KB},4} = M_{\mathrm{H},4} M_{\mathrm{KB},4 \to \mathrm{H},4}$$

[13]:
```
NamedMatrix(control_values_H, M_KBtoH.name * control_values_KB)
```

[13]:
$$\begin{bmatrix} x_4 \\ x_5 \\ \dot{x}_4^{(+)} \\ \dot{x}_5^{(-)} \end{bmatrix} = M_{\mathrm{KB},4 \to \mathrm{H},4} \begin{bmatrix} x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

If we substitute the above definitions of $\dot{x}_4^{(+)}$ and $\dot{x}_5^{(-)}$, we can directly read off the matrix elements:

[14]:
```
M_KBtoH.expr = sp.Matrix([
    [expr.coeff(cv).simplify() for cv in control_values_KB]
    for expr in control_values_H.subs([xd4.args, xd5.args]).expand()])
M_KBtoH
```

[14]:
$$M_{\mathrm{KB},4 \to \mathrm{H},4} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{a_4(t_4-t_5)}{t_3^2-t_3t_4-t_3t_5+t_4t_5} & \frac{-a_4t_4^2+2a_4t_4t_5-a_4t_5^2+b_4t_3^2-2b_4t_3t_4+b_4t_4^2}{t_3^2t_4-t_3^2t_5-t_3t_4^2+t_3t_5^2+t_4^2t_5-t_4t_5^2} & \frac{b_4(-t_3+t_4)}{t_3t_4-t_3t_5-t_4t_5+t_5^2} & 0 \\ 0 & \frac{c_5(t_5-t_6)}{t_4^2-t_4t_5-t_4t_6+t_5t_6} & \frac{-c_5t_5^2+2c_5t_5t_6-c_5t_6^2+d_5t_4^2-2d_5t_4t_5+d_5t_5^2}{t_4^2t_5-t_4^2t_6-t_4t_5^2+t_4t_6^2+t_5^2t_6-t_5t_6^2} & \frac{d_5(-t_4+t_5)}{t_4t_5-t_4t_6-t_5t_6+t_6^2} \end{bmatrix}$$

[15]:
```
delta3, delta4, delta5 = sp.symbols('Delta3:6')
deltas = {
    t3: 0,
    t4: delta3,
    t5: delta3 + delta4,
```

(continues on next page)

(continued from previous page)

```
    t6: delta3 + delta4 + delta5,
}
```

[16]:
```
M_KBtoH.subs(deltas).simplify()
```

[16]:
$$M_{\mathrm{KB},4 \to \mathrm{H},4} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\frac{\Delta_4 a_4}{\Delta_3(\Delta_3+\Delta_4)} & \frac{-\Delta_3^2 b_4+\Delta_4^2 a_4}{\Delta_3\Delta_4(\Delta_3+\Delta_4)} & \frac{\Delta_3 b_4}{\Delta_4(\Delta_3+\Delta_4)} & 0 \\ 0 & -\frac{\Delta_5 c_5}{\Delta_4(\Delta_4+\Delta_5)} & \frac{-\Delta_4^2 d_5+\Delta_5^2 c_5}{\Delta_4\Delta_5(\Delta_4+\Delta_5)} & \frac{\Delta_4 d_5}{\Delta_5(\Delta_4+\Delta_5)} \end{bmatrix}$$

[17]:
```
M_KB = M_KB.subs_symbols(M_H, M_KBtoH).doit()
M_KB.subs(deltas).expand()
```

[17]:
$$M_{\mathrm{KB},4} = \begin{bmatrix} -\frac{\Delta_4^2 a_4}{\Delta_3^2+\Delta_3\Delta_4} & \frac{\Delta_3^2\Delta_4 b_4}{-\Delta_3^2\Delta_4-\Delta_3\Delta_4^2} - \frac{\Delta_4^3 a_4}{-\Delta_3^2\Delta_4-\Delta_3\Delta_4^2} - \frac{\Delta_4\Delta_5 c_5}{\Delta_4^2+\Delta_4\Delta_5} + 2 & \frac{\Delta_3\Delta_4 b_4}{\Delta_3\Delta_4+\Delta_4^2} + \frac{\Delta_4^3 d_5}{-\Delta_4^2\Delta_5-\Delta_4\Delta_5^2} - \frac{\Delta_4\Delta_5^2 c_5}{-\Delta_4^2\Delta_5-\Delta_4\Delta_5^2} - 2 & \frac{\Delta_4^2 d_5}{\Delta_4\Delta_5+\Delta_5^2} \\ \frac{2\Delta_4^2 a_4}{\Delta_3^2+\Delta_3\Delta_4} & -\frac{2\Delta_3^2\Delta_4 b_4}{-\Delta_3^2\Delta_4-\Delta_3\Delta_4^2} + \frac{2\Delta_4^3 a_4}{-\Delta_3^2\Delta_4-\Delta_3\Delta_4^2} + \frac{\Delta_4\Delta_5 c_5}{\Delta_4^2+\Delta_4\Delta_5} - 3 & -\frac{2\Delta_3\Delta_4 b_4}{\Delta_3\Delta_4+\Delta_4^2} - \frac{\Delta_4^3 d_5}{-\Delta_4^2\Delta_5-\Delta_4\Delta_5^2} + \frac{\Delta_4\Delta_5^2 c_5}{-\Delta_4^2\Delta_5-\Delta_4\Delta_5^2} + 3 & -\frac{\Delta_4^2 d_5}{\Delta_4\Delta_5+\Delta_5^2} \\ -\frac{\Delta_4^2 a_4}{\Delta_3^2+\Delta_3\Delta_4} & \frac{\Delta_3^2\Delta_4 b_4}{-\Delta_3^2\Delta_4-\Delta_3\Delta_4^2} - \frac{\Delta_4^3 a_4}{-\Delta_3^2\Delta_4-\Delta_3\Delta_4^2} & \frac{\Delta_3\Delta_4 b_4}{\Delta_3\Delta_4+\Delta_4^2} & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

And for completeness' sake, its inverse:

[18]:
```
M_KB.subs(deltas).expand().I
```

[18]:
$$M_{\mathrm{KB},4}{}^{-1} = \begin{bmatrix} \frac{\Delta_3^2 b_4}{\Delta_4^2 a_4} & \frac{\Delta_3^2 b_4}{\Delta_4^2 a_4} & \frac{\Delta_3(\Delta_3 b_4-\Delta_3-\Delta_4)}{\Delta_4^2 a_4} & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ \frac{\Delta_4^2 d_5+3\Delta_4\Delta_5-\Delta_5^2 c_5+3\Delta_5^2}{\Delta_4^2 d_5} & \frac{\Delta_4^2 d_5+2\Delta_4\Delta_5-\Delta_5^2 c_5+2\Delta_5^2}{\Delta_4^2 d_5} & \frac{\Delta_4^2 d_5+\Delta_4\Delta_5-\Delta_5^2 c_5+\Delta_5^2}{\Delta_4^2 d_5} & 1 \end{bmatrix}$$

TODO: plot some example curves

doc/euclidean/kochanek-bartels-non-uniform.ipynb ends here.

1.8 End Conditions

The following section was generated from doc/euclidean/end-conditions-natural.ipynb

Natural End Conditions

For the first and last segment, we assume that the inner tangent is known. We try to find the outer tangent by setting the second derivative to 0.

We are looking only at the non-uniform case here, it's easy to get to the uniform case by setting $\Delta_i = 1$.

natural (a.k.a. "relaxed"?)

[1]:
```
import sympy as sp
sp.init_printing(order='rev-lex')
```

utility.py

[2]:
```
from utility import NamedExpression
```

[3]:
```
t = sp.symbols('t')
```

Begin

first polynomial segment: $\boldsymbol{p}_0(t)$, $t \in [t_0, t_1]$

[4]:
```
t0, t1 = sp.symbols('t:2')
```

[5]:
```
a0, b0, c0, d0 = sp.symbols('a:dbm0')
```

[6]:
```
d0 * t**3 + c0 * t**2 + b0 * t + a0
```

[6]:
$$\boldsymbol{d}_0 t^3 + \boldsymbol{c}_0 t^2 + \boldsymbol{b}_0 t + \boldsymbol{a}_0$$

[7]:
```
p0 = NamedExpression('pbm0', _.subs(t, (t - t0) / (t1 - t0)))
p0
```

[7]:
$$\boldsymbol{p}_0 = \frac{\boldsymbol{d}_0 \left(-t_0 + t\right)^3}{\left(t_1 - t_0\right)^3} + \frac{\boldsymbol{c}_0 \left(-t_0 + t\right)^2}{\left(t_1 - t_0\right)^2} + \frac{\boldsymbol{b}_0 \left(-t_0 + t\right)}{t_1 - t_0} + \boldsymbol{a}_0$$

Velocity = Tangent Vector = Derivative:

[8]:
```
pd0 = p0.diff(t)
pd0
```

[8]:
$$\frac{d}{dt}\boldsymbol{p}_0 = \frac{3\boldsymbol{d}_0 \left(-t_0 + t\right)^2}{\left(t_1 - t_0\right)^3} + \frac{\boldsymbol{c}_0 \left(-2t_0 + 2t\right)}{\left(t_1 - t_0\right)^2} + \frac{\boldsymbol{b}_0}{t_1 - t_0}$$

similar to *notebook about non-uniform Hermite splines* (page 24)

$$\boldsymbol{x}_0 = \boldsymbol{p}_0(t_0) \tag{4}$$
$$\boldsymbol{x}_1 = \boldsymbol{p}_0(t_1) \tag{5}$$
$$\dot{\boldsymbol{x}}_0 = \boldsymbol{p}_0'(t_0) \tag{6}$$
$$\dot{\boldsymbol{x}}_1 = \boldsymbol{p}_0'(t_1) \tag{7}$$

[9]:
```
equations_begin = [
    p0.evaluated_at(t, t0).with_name('xbm0'),
    p0.evaluated_at(t, t1).with_name('xbm1'),
    pd0.evaluated_at(t, t0).with_name('xbmdot0'),
    pd0.evaluated_at(t, t1).with_name('xbmdot1'),
]
```

only for display purposes, the calculations are still done with t_i

[10]:
```
delta_begin = [
    (t0, 0),
    (t1, sp.Symbol('Delta0')),
]
```

[11]:
```
for e in equations_begin:
    display(e.subs(delta_begin))
```

$\boldsymbol{x}_0 = \boldsymbol{a}_0$

$$x_1 = d_0 + c_0 + b_0 + a_0$$

$$\dot{x}_0 = \frac{b_0}{\Delta_0}$$

$$\dot{x}_1 = \frac{3d_0}{\Delta_0} + \frac{2c_0}{\Delta_0} + \frac{b_0}{\Delta_0}$$

[12]:
```
coefficients_begin = sp.solve(equations_begin, [a0, b0, c0, d0])
```

[13]:
```
for c, e in coefficients_begin.items():
    display(NamedExpression(c, e.subs(delta_begin)))
```

$$a_0 = x_0$$

$$b_0 = \Delta_0 \dot{x}_0$$

$$c_0 = 3x_1 - 3x_0 - \Delta_0 \dot{x}_1 - 2\Delta_0 \dot{x}_0$$

$$d_0 = -2x_1 + 2x_0 + \Delta_0 \dot{x}_1 + \Delta_0 \dot{x}_0$$

Acceleration = Second Derivative

[14]:
```
pdd0 = pd0.diff(t)
pdd0
```

[14]:
$$\frac{d^2}{dt^2} p_0 = \frac{3d_0 \left(-2t_0 + 2t\right)}{\left(t_1 - t_0\right)^3} + \frac{2c_0}{\left(t_1 - t_0\right)^2}$$

[15]:
```
pdd0.evaluated_at(t, t0)
```

[15]:
$$\left.\frac{d^2}{dt^2} p_0\right|_{t=t_0} = \frac{2c_0}{\left(t_1 - t_0\right)^2}$$

[16]:
```
sp.Eq(_.expr, 0).subs(coefficients_begin)
```

[16]:
$$\frac{2\left(3x_1 - 3x_0 - t_1\dot{x}_1 - 2t_1\dot{x}_0 + t_0\dot{x}_1 + 2t_0\dot{x}_0\right)}{\left(t_1 - t_0\right)^2} = 0$$

[17]:
```
xd0 = NamedExpression.solve(_, 'xbmdot0')
xd0.subs(delta_begin)
```

[17]:
$$\dot{x}_0 = -\frac{-3x_1 + 3x_0 + \Delta_0 \dot{x}_1}{2\Delta_0}$$

End

N vertices, $N-1$ polynomial segments

last polynomial: $p_{N-2}(t)$, $t \in [t_{N-2}, t_{N-1}]$

To simplify the notation a bit, let's assume we have $N = 10$ vertices, which makes p_8 the last polynomial segment.

[18]:
```
a8, b8, c8, d8 = sp.symbols('a:dbm8')
```

[19]:
```
t8, t9 = sp.symbols('t8:10')
```

[20]:
```
d8 * t**3 + c8 * t**2 + b8 * t + a8
```

[20]:
$$d_8 t^3 + c_8 t^2 + b_8 t + a_8$$

[21]:
```
p8 = NamedExpression('pbm8', _.subs(t, (t - t8) / (t9 - t8)))
p8
```

[21]:
$$p_8 = \frac{d_8 \left(-t_8 + t\right)^3}{\left(t_9 - t_8\right)^3} + \frac{c_8 \left(-t_8 + t\right)^2}{\left(t_9 - t_8\right)^2} + \frac{b_8 \left(-t_8 + t\right)}{t_9 - t_8} + a_8$$

[22]:
```
pd8 = p8.diff(t)
pd8
```

[22]:
$$\frac{d}{dt} p_8 = \frac{3 d_8 \left(-t_8 + t\right)^2}{\left(t_9 - t_8\right)^3} + \frac{c_8 \left(-2 t_8 + 2 t\right)}{\left(t_9 - t_8\right)^2} + \frac{b_8}{t_9 - t_8}$$

$$x_{N-2} = p_{N-2}(t_{N-2}) \tag{8}$$
$$x_{N-1} = p_{N-2}(t_{N-1}) \tag{9}$$
$$\dot{x}_{N-2} = p'_{N-2}(t_{N-2}) \tag{10}$$
$$\dot{x}_{N-1} = p'_{N-2}(t_{N-1}) \tag{11}$$

[23]:
```
equations_end = [
    p8.evaluated_at(t, t8).with_name('xbm8'),
    p8.evaluated_at(t, t9).with_name('xbm9'),
    pd8.evaluated_at(t, t8).with_name('xbmdot8'),
    pd8.evaluated_at(t, t9).with_name('xbmdot9'),
]
```

[24]:
```
delta_end = [
    (t8, 0),
    (t9, sp.Symbol('Delta8')),
]
```

[25]:
```
for e in equations_end:
    display(e.subs(delta_end))
```

$$x_8 = a_8$$

$$x_9 = d_8 + c_8 + b_8 + a_8$$

$$\dot{x}_8 = \frac{b_8}{\Delta_8}$$

$$\dot{x}_9 = \frac{3 d_8}{\Delta_8} + \frac{2 c_8}{\Delta_8} + \frac{b_8}{\Delta_8}$$

[26]:
```
coefficients_end = sp.solve(equations_end, [a8, b8, c8, d8])
```

[27]:
```
for c, e in coefficients_end.items():
    display(NamedExpression(c, e.subs(delta_end)))
```

$$a_8 = x_8$$

$$b_8 = \Delta_8 \dot{x}_8$$

$$c_8 = 3 x_9 - 3 x_8 - \Delta_8 \dot{x}_9 - 2 \Delta_8 \dot{x}_8$$

$$d_8 = -2 x_9 + 2 x_8 + \Delta_8 \dot{x}_9 + \Delta_8 \dot{x}_8$$

[28]:
```
pdd8 = pd8.diff(t)
pdd8
```

[28]:
$$\frac{d^2}{dt^2}\boldsymbol{p}_8 = \frac{3\boldsymbol{d}_8\left(-2t_8+2t\right)}{\left(t_9-t_8\right)^3}+\frac{2\boldsymbol{c}_8}{\left(t_9-t_8\right)^2}$$

second derivative *at the end* of the last segment:

[29]:
```
pdd8.evaluated_at(t, t9)
```

[29]:
$$\left.\frac{d^2}{dt^2}\boldsymbol{p}_8\right|_{t=t_9} = \frac{3\boldsymbol{d}_8\left(2t_9-2t_8\right)}{\left(t_9-t_8\right)^3}+\frac{2\boldsymbol{c}_8}{\left(t_9-t_8\right)^2}$$

[30]:
```
sp.Eq(_.expr, 0).subs(coefficients_end)
```

[30]:
$$\frac{2\left(3\boldsymbol{x}_9-3\boldsymbol{x}_8-t_9\dot{\boldsymbol{x}}_9-2t_9\dot{\boldsymbol{x}}_8+t_8\dot{\boldsymbol{x}}_9+2t_8\dot{\boldsymbol{x}}_8\right)}{\left(t_9-t_8\right)^2}+\frac{3\left(2t_9-2t_8\right)\left(-2\boldsymbol{x}_9+2\boldsymbol{x}_8+t_9\dot{\boldsymbol{x}}_9+t_9\dot{\boldsymbol{x}}_8-t_8\dot{\boldsymbol{x}}_9-t_8\dot{\boldsymbol{x}}_8\right)}{\left(t_9-t_8\right)^3} = 0$$

[31]:
```
xd9 = NamedExpression.solve(_, 'xbmdot9')
xd9.subs(delta_end)
```

[31]:
$$\dot{\boldsymbol{x}}_9 = -\frac{-3\boldsymbol{x}_9+3\boldsymbol{x}_8+\Delta_8\dot{\boldsymbol{x}}_8}{2\Delta_8}$$

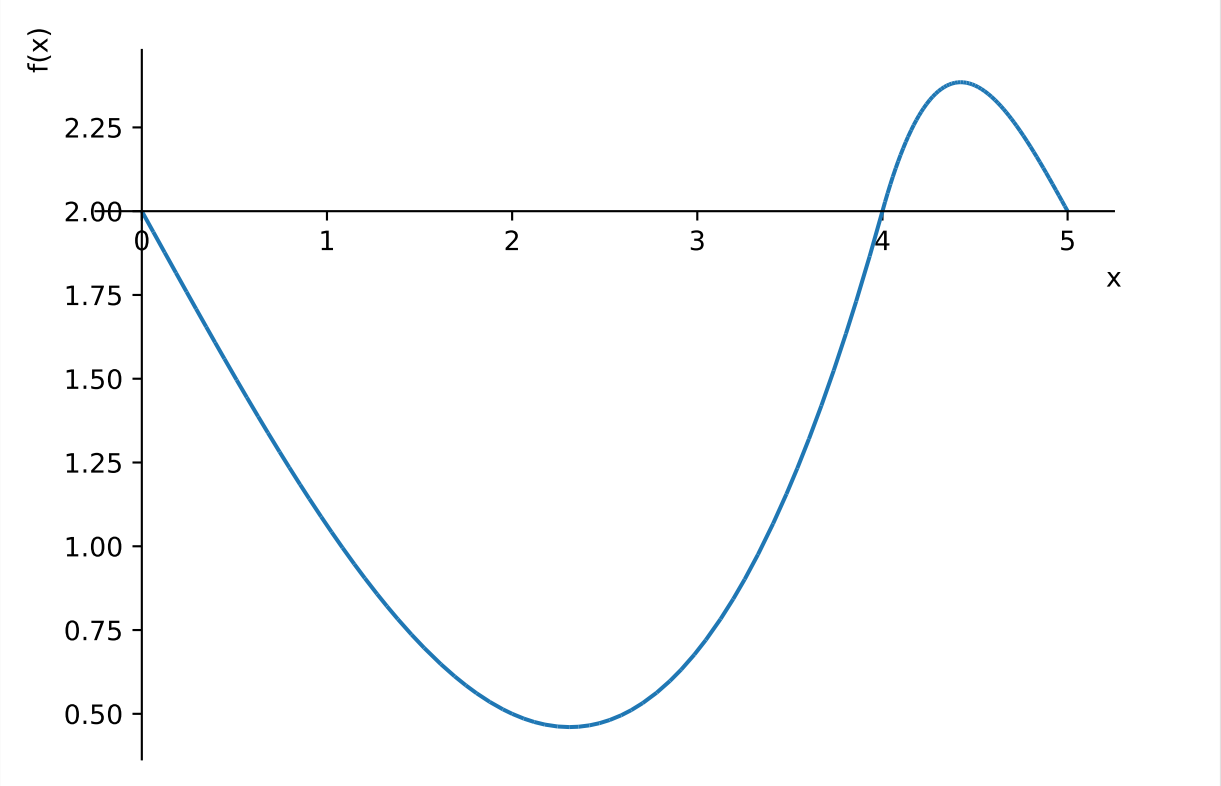
Luckily, that's symmetric to the result we got above.

Example

one-dimensional; 3 time/value pairs are given. The slope for the middle value is given, the begin and end slopes are calculated using the "natural" end conditions as calculated above.

[32]:
```
values = 2, 2, 2
times = 0, 4, 5
slope = 2
```

[33]:
```
sp.plot((
    p0.subs(coefficients_begin).subs_symbols(xd0).expr.subs({
        t0: times[0],
        t1: times[1],
        sp.Symbol('xbm0'): values[0],
        sp.Symbol('xbm1'): values[1],
        sp.Symbol('xbmdot1'): slope,
    }),
    (t, times[0], times[1])
), (
    p8.subs(coefficients_end).subs_symbols(xd9).expr.subs({
        t8: times[1],
        t9: times[2],
        sp.Symbol('xbm8'): values[1],
        sp.Symbol('xbm9'): values[2],
        sp.Symbol('xbmdot8'): slope,
    }),
    (t, times[1], times[2])
), axis_center=(0, values[1]));
```



[33]: `<sympy.plotting.plot.Plot at 0x7f635dabdbd0>`

doc/euclidean/end-conditions-natural.ipynb ends here.

The following section was generated from doc/euclidean/piecewise-monotone.ipynb

1.9 Piecewise Monotone Interpolation

When interpolating a sequence of one-dimensional data points, it is sometimes desirable to limit the interpolant between any two adjacent data points to a monotone function. This makes sure that there are no overshoots beyond the given data points. In other words, if the data points are within certain bounds, all interpolated data will also be within those same bounds. It follows that if all data points are non-negative, interpolated data will be non-negative as well. Furthermore, this makes sure that monotone data leads to a monotone interpolant (see also *Monotone Interpolation* (page 111) below).

A Python implementation of piecewise monotone one-dimensional cubic splines is available in the *splines.PiecewiseMonotoneCubic* (page 151) class.

The SciPy package provides a similar tool with the pchip_interpolate()[14] function and the PchipInterpolator[15] class (see below for more details).

The 3D animation software Blender[16] provides an Auto Clamped[17] property for creating piecewise monotone animation cuves.

[14] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.pchip_interpolate.html
[15] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PchipInterpolator.html
[16] https://www.blender.org
[17] https://docs.blender.org/manual/en/dev/editors/graph_editor/fcurves/introduction.html#handle-type

Examples

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

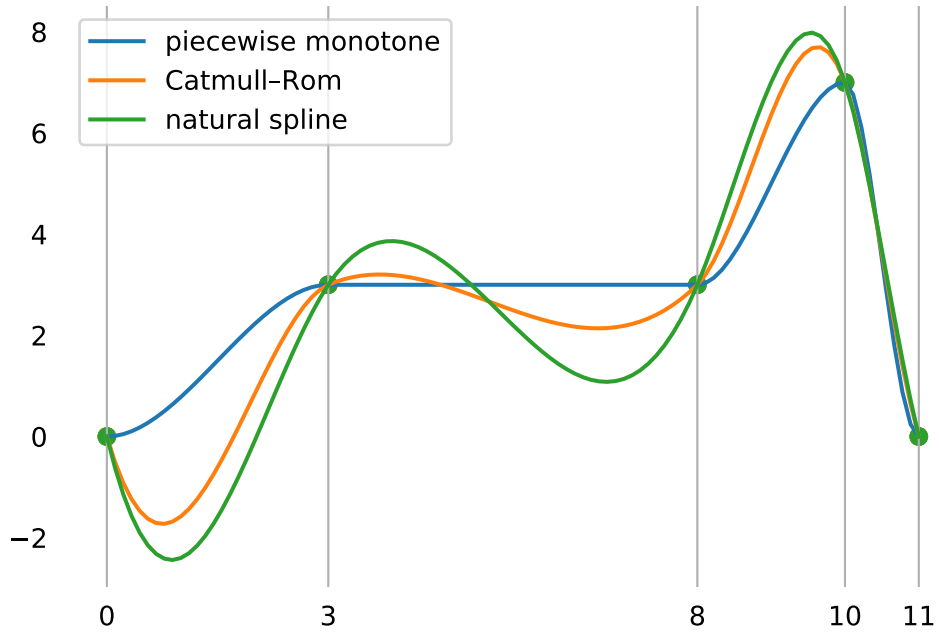
[2]:
```
import splines
```

helper.py

[3]:
```
from helper import plot_spline_1d, grid_lines
```

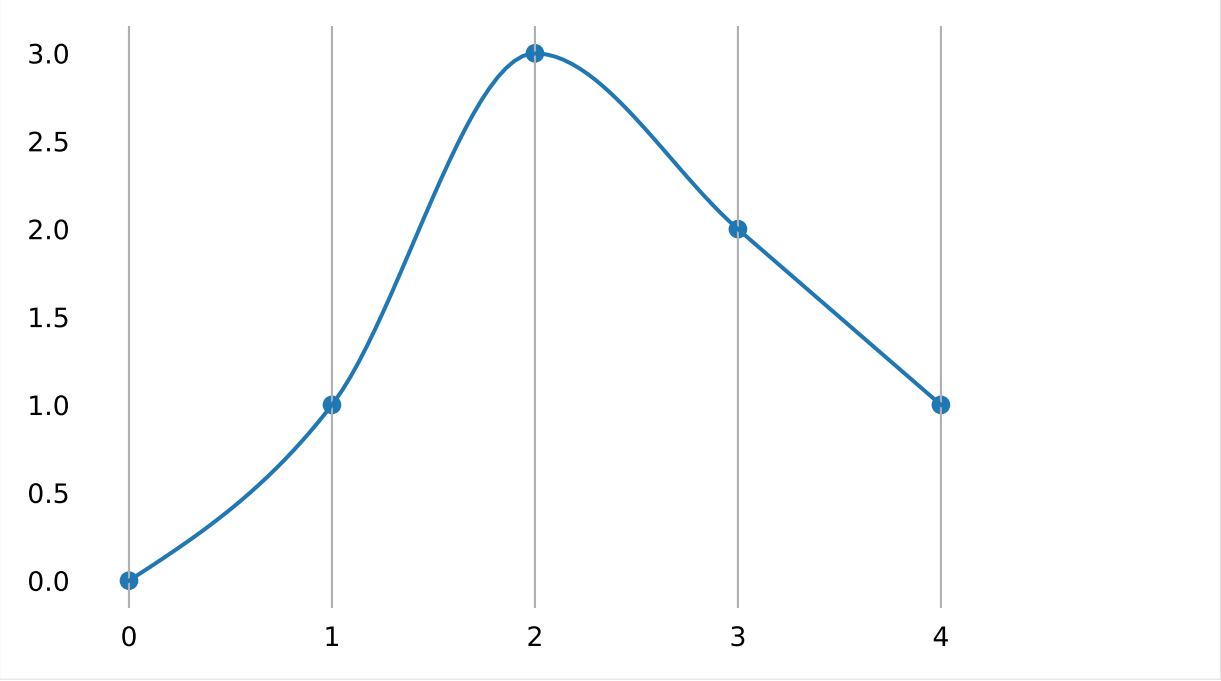
[4]:
```
values = 0, 3, 3, 7
times = 0, 3, 8, 10, 11
```

[5]:
```
plot_spline_1d(
    splines.PiecewiseMonotoneCubic(values, times, closed=True),
    label='piecewise monotone')
plot_spline_1d(
    splines.CatmullRom(values, times, endconditions='closed'),
    label='Catmull-Rom')
plot_spline_1d(
    splines.Natural(values, times, endconditions='closed'),
    label='natural spline')
plt.legend()
grid_lines(times)
```



[6]:
```
def plot_piecewise_monotone(*args, **kwargs):
    s = splines.PiecewiseMonotoneCubic(*args, **kwargs)
    plot_spline_1d(s)
    grid_lines(x=s.grid)
```

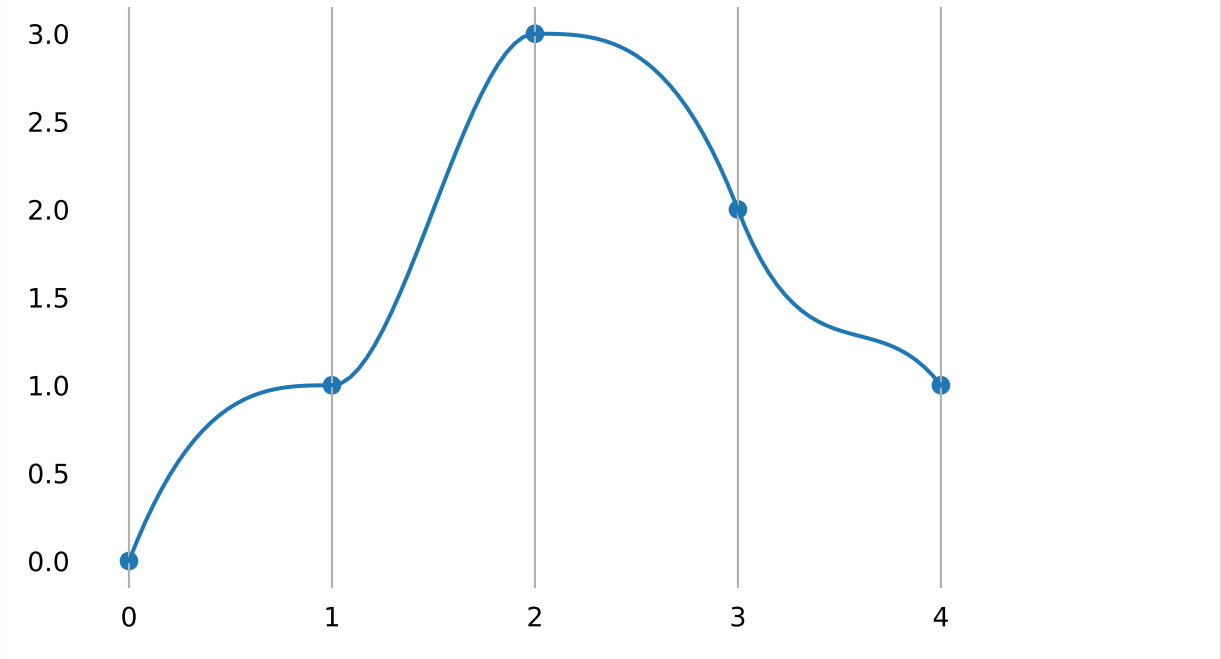
[7]:
```
plot_piecewise_monotone([0, 1, 3, 2, 1])
```



Providing Slopes

By default, appropriate slopes are calculated automatically. However, those slopes can be overridden if desired. Specifying `None` falls back to the auto-generated default.

[8]:
```
plot_piecewise_monotone([0, 1, 3, 2, 1], slopes=[None, 0, None, -3, -1.5])
```



Slopes that would lead to non-monotone segments are prohibited:

[9]:
```
try:
    plot_piecewise_monotone([0, 1, 3, 2, 1], slopes=[None, 4, None, None, None])
except Exception as e:
```

(continues on next page)

(continued from previous page)

```
    print(e)
    assert 'too steep' in str(e)
else:
    assert False
```

```
Slope too steep: 4
```

Generating and Modifying the Slopes at Segment Boundaries

> In this paper we derive necessary and sufficient conditions for a cubic to be monotone in an interval. These conditions are then used to develop an algorithm which constructs a $\mathscr{C}^1$ monotone piecewise cubic interpolant to monotone data. The curve produced contains no extraneous "bumps" or "wiggles", which makes it more readily acceptable to scientists and engineers.
>
> —[FC80], section 1: "Introduction"

[FC80] derives necessary and sufficient conditions for a cubic curve segment to be monotone, based on the slopes of the secant lines (i.e. the piecewise linear interpolant) and their endpoint derivatives. Furthermore, they provide a two-step algorithm to generate piecewise monotone cubics:

1. calculate initial tangents (with whatever method)
2. tweak the ones that don't fulfill the monotonicity conditions

> To implement Step 1 we have found the standard three-point difference formula to be satisfactory for $d_2, d_3, \cdots, d_{n-1}$.
>
> —[FC80], section 4: "Monotone piecewise cubic interpolation algorithm"
>
> This is what de Boor [[dB78], p. 53] calls cubic Bessel interpolation, in which the interior derivatives are set using the standard three point difference formula.
>
> —[FC80], section 5: "Numerical examples"

In the 2001 edition of [dB78], *piecewise cubic Bessel interpolation* is defined on page 42.

For the following equations, we define the slope of the secant lines as

$$S_i = \frac{x_{i+1} - x_i}{t_{i+1} - t_i}.$$

We use x_i to represent the given data points and and t_i to represent the corresponding parameter values. The slope at those values is represented by $\dot{x}_i$.

Note

In the literature, the parameter values are often represented by x_i, so try not to be confused!

Based on [FC80], [DEH89] provides (in equation 4.2) an algorithm for modifying the initial slopes to ensure monotonicity. Adapted to our notation, it looks like this:

$$\dot{x}_i \leftarrow \begin{cases} \min(\max(0, \dot{x}_i), 3\min(|S_{i-1}|, |S_i|)), & \sigma_i > 0, \\ \max(\min(0, \dot{x}_i), -3\min(|S_{i-1}|, |S_i|)), & \sigma_i < 0, \\ 0, & \sigma_i = 0, \end{cases}$$

where $\sigma_i = \operatorname{sgn}(S_i)$ if $S_i S_{i-1} > 0$ and $\sigma_i = 0$ otherwise.

This algoritm is implemented in the *splines.PiecewiseMonotoneCubic* (page 151) class.

PCHIP/PCHIM

A different approach for obtaining slopes that ensure monotonicity is described in [FB84], equation (5):

$$G(S_1, S_2, h_1, h_2) = \begin{cases} \frac{S_1 S_2}{\alpha S_2 + (1-\alpha) S_1} & \text{if } S_1 S_2 > 0, \\ 0 & \text{otherwise,} \end{cases}$$

where

$$\alpha = \frac{1}{3}\left(1 + \frac{h_2}{h_1 + h_2}\right) = \frac{h_1 + 2h_2}{3(h_1 + h_2)}.$$

The function G can be used to calculate the slopes at segment boundaries, given the slopes S_i of the neighboring secant lines and the neighboring parameter intervals $h_i = t_{i+1} - t_i$.

Let's define this using SymPy[18] for later reference:

[10]:
```
import sympy as sp
```

[11]:
```
h1, h2 = sp.symbols('h1:3')
S1, S2 = sp.symbols('S1:3')
```

[12]:
```
alpha = (h1 + 2 * h2) / (3 * (h1 + h2))
G1 = (S1 * S2) / (alpha * S2 + (1 - alpha) * S1)
```

This has been implemented in a Fortran[19] package described in [Fri82], which has coined the acronym PCHIP, originally meaning *Piecewise Cubic Hermite Interpolation Package*.

> It features software to produce a monotone and "visually pleasing" interpolant to monotone data.
>
> —[Fri82]

The package contains many Fortran subroutines, but the one that's relevant here is `PCHIM`, which is short for *Piecewise Cubic Hermite Interpolation to Monotone data*.

The source code (including some later modifications) is available at https://people.sc.fsu.edu/~jburkardt/f77_src/pchip/pchip.html. This is the code snippet responsible for calculating the slopes:

```
C
C        USE BRODLIE MODIFICATION OF BUTLAND FORMULA.
C
   45    CONTINUE
         HSUMT3 = HSUM+HSUM+HSUM
         W1 = (HSUM + H1)/HSUMT3
         W2 = (HSUM + H2)/HSUMT3
         DMAX = MAX( ABS(DEL1), ABS(DEL2) )
         DMIN = MIN( ABS(DEL1), ABS(DEL2) )
         DRAT1 = DEL1/DMAX
         DRAT2 = DEL2/DMAX
         D(1,I) = DMIN/(W1*DRAT1 + W2*DRAT2)
```

This looks different from the function G defined above, but if we transform the Fortran code into math ...

[13]:
```
HSUM = h1 + h2
```

[18] https://www.sympy.org/
[19] https://en.wikipedia.org/wiki/Fortran

[14]:
```
W1 = (HSUM + h1) / (3 * HSUM)
W2 = (HSUM + h2) / (3 * HSUM)
```

... and use separate expressions depending on which of the neighboring secant slopes is larger ...

[15]:
```
G2 = S1 / (W1 * S1 / S2 + W2 * S2 / S2)
G3 = S2 / (W1 * S1 / S1 + W2 * S2 / S1)
```

... we see that the two cases are mathematically equivalent ...

[16]:
```
assert sp.simplify(G2 - G3) == 0
```

... and they are in fact also equivalent to the aforementioned equation from [FB84]:

[17]:
```
assert sp.simplify(G1 - G2) == 0
```

Presumably, the Fortran code uses the larger one of the pair of secant slopes in the denominator in order to reduce numerical errors if one of the slopes is very close to zero.

Yet another variation of this theme is shown in [Molo4], section 3.4, which defines the slope d_k as a weighted harmonic mean of the two neighboring secant slopes:

$$\frac{w_1 + w_2}{d_k} = \frac{w_1}{\delta_{k-1}} + \frac{w_2}{\delta_k},$$

with $w_1 = 2h_k + h_{k-1}$ and $w_2 = h_k + 2h_{k-1}$. Using the notation from above, $d_k = \dot{x}_k$ and $\delta_k = S_k$.

Again, when defining this using SymPy ...

[18]:
```
w1 = 2 * h2 + h1
w2 = h2 + 2 * h1
```

[19]:
```
G4 = (w1 + w2) / (w1 / S1 + w2 / S2)
```

... we can see that it is actually equivalent to the previous equations:

[20]:
```
assert sp.simplify(G4 - G1) == 0
```

The PCHIM algorithm, which is nowadays known by the less self-explanatory name PCHIP, is available in the *SciPy* package in form of the pchip_interpolate()[20] function and the PchipInterpolator[21] class.

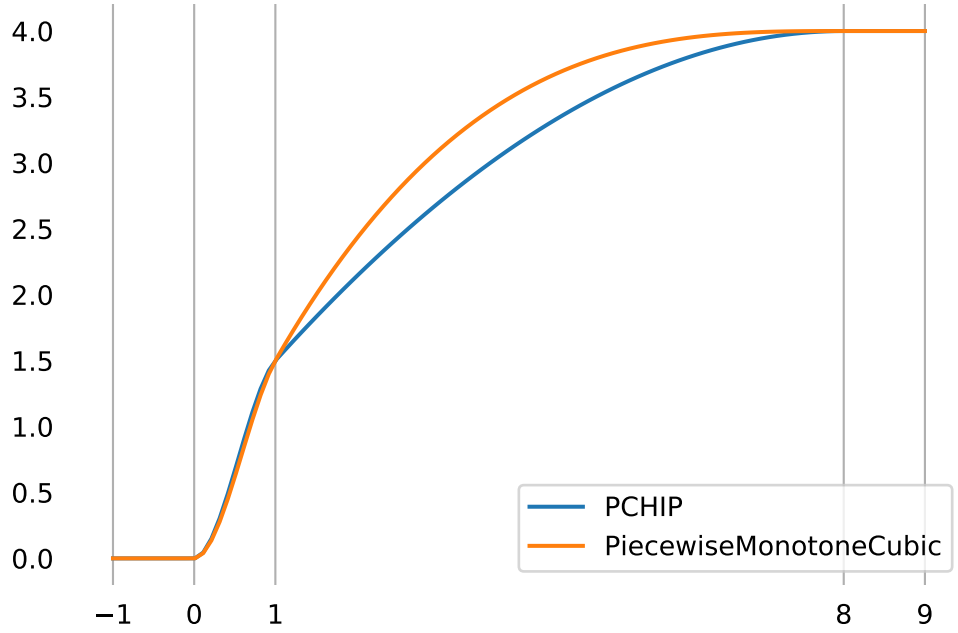
[21]:
```
from scipy.interpolate import PchipInterpolator
```

More Examples

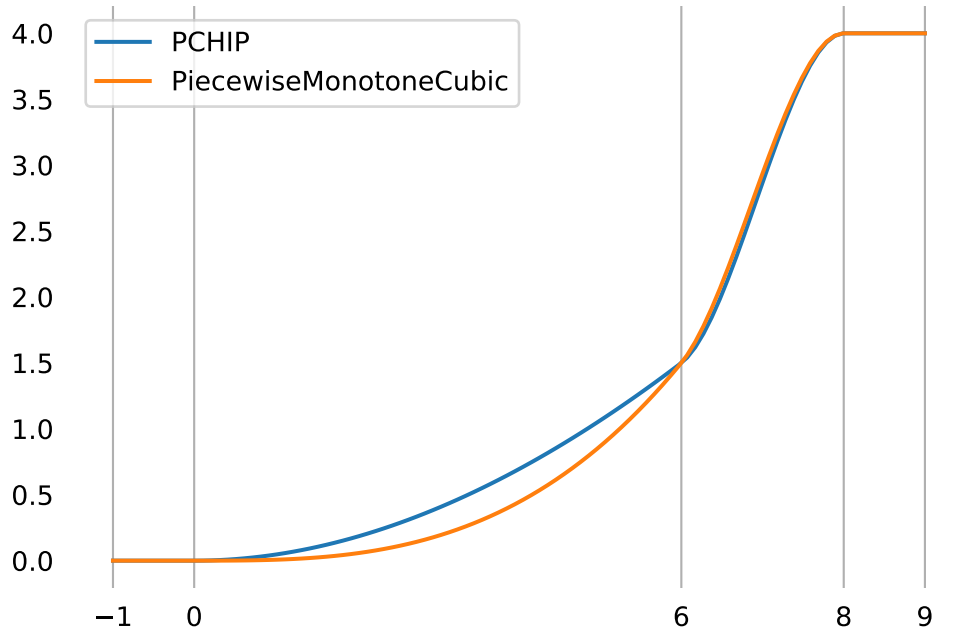
[22]:
```
def compare_pchip(values, times):
    plot_times = np.linspace(times[0], times[-1], 100)
    plt.plot(
        plot_times,
        PchipInterpolator(times, values)(plot_times),
        label='PCHIP')
    plt.plot(
        plot_times,
        splines.PiecewiseMonotoneCubic(values, times).evaluate(plot_times),
        label='PiecewiseMonotoneCubic')
    plt.legend()
    grid_lines(x=times)
```

[20] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.pchip_interpolate.html
[21] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PchipInterpolator.html

[23]:
```
compare_pchip([0, 0, 1.5, 4, 4], [-1, 0, 1, 8, 9])
```

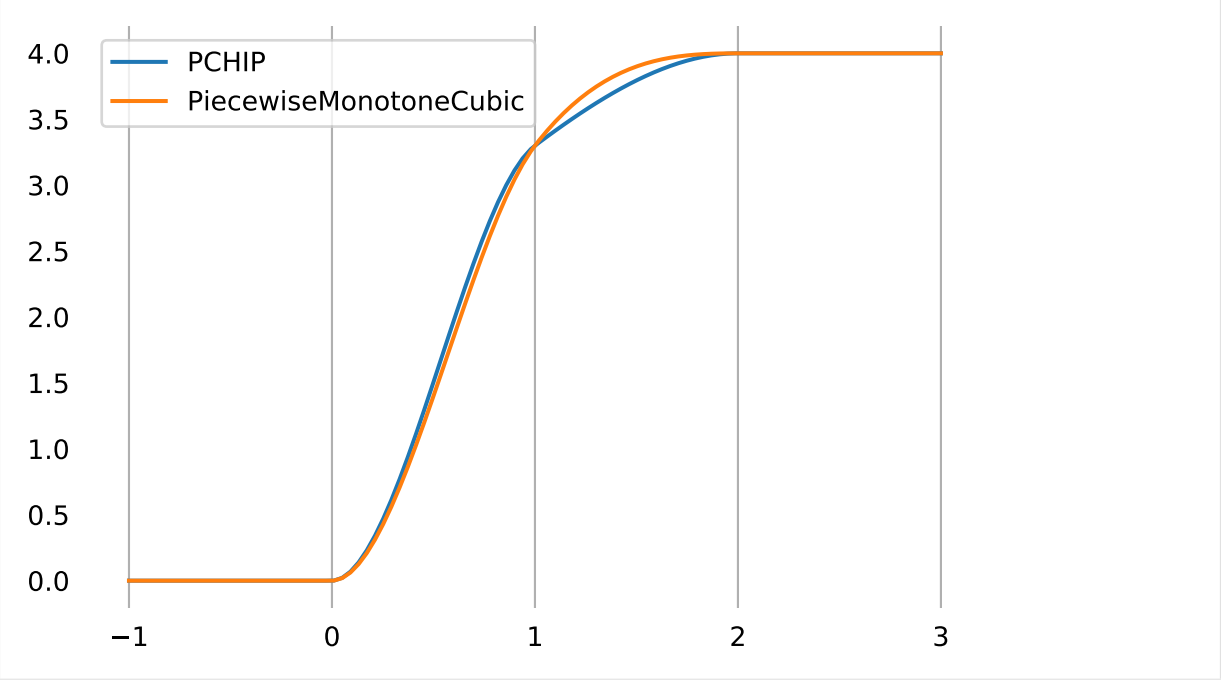


[24]:
```
compare_pchip([0, 0, 1.5, 4, 4], [-1, 0, 6, 8, 9])
```

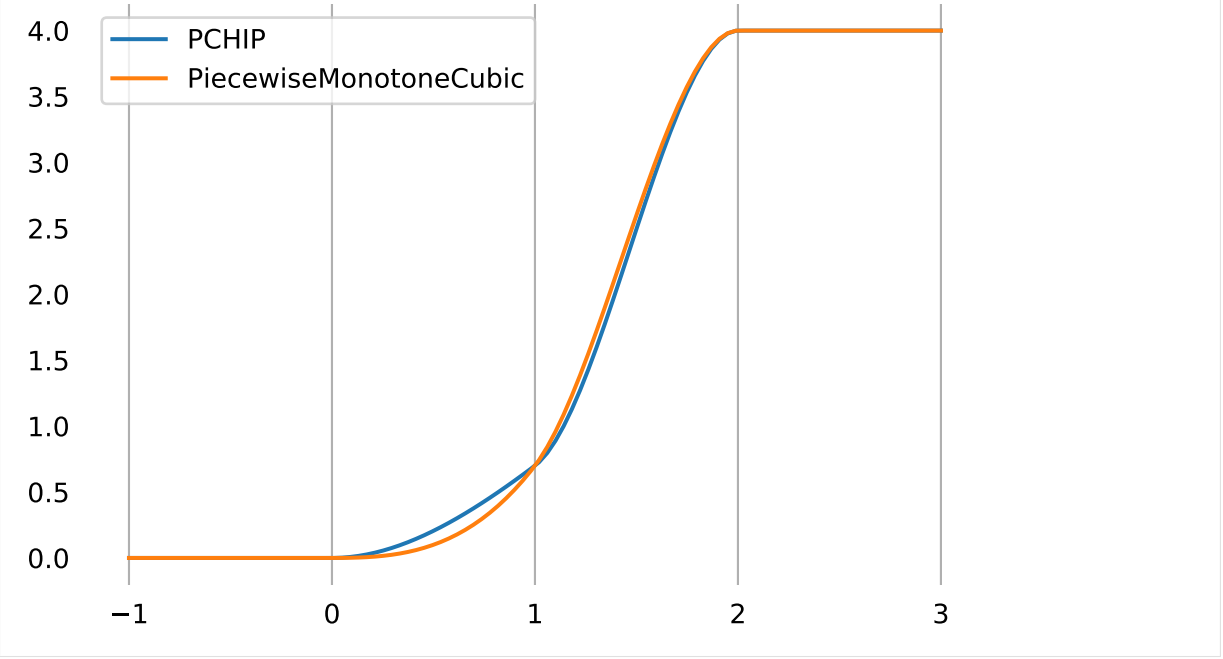


There is even a slight difference in the uniform case:

[25]:
```
compare_pchip([0, 0, 3.3, 4, 4], [-1, 0, 1, 2, 3])
```



[26]:
```
compare_pchip([0, 0, 0.7, 4, 4], [-1, 0, 1, 2, 3])
```



For differences at the beginning and the end of the curve, see the *section about end conditions* (page 112).

Monotone Interpolation

When using the aforementioned piecewise monotone algorithms with monotone data, the whole interpolant will be monotone.

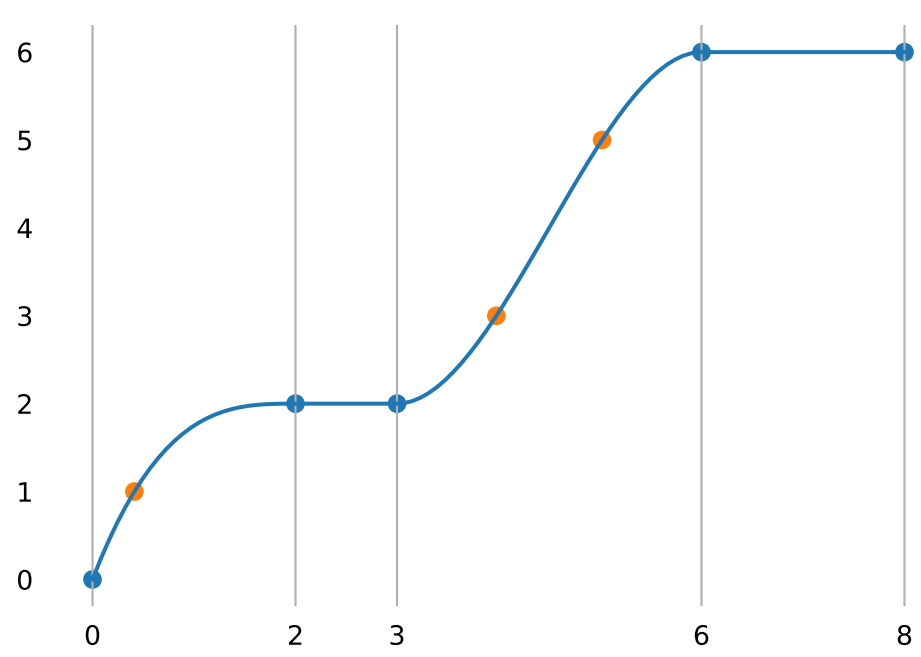
The class *splines.MonotoneCubic* (page 152) works very much the same as *splines.PiecewiseMonotoneCubic* (page 151), except that it only allows monotone data values.

Since the resulting interpolation function is monotone, it can be inverted. Given a function value, the method *.get_time()* (page 152) can be used to find the associated parameter value.

[27]:
```
s = splines.MonotoneCubic([0, 2, 2, 6, 6], grid=[0, 2, 3, 6, 8])
```

[28]:
```
probes = 1, 3, 5
```

[29]:
```
fig, ax = plt.subplots()
plot_spline_1d(s)
ax.scatter(s.get_time(probes), probes)
grid_lines(x=s.grid)
```



If the solution is not unique (i.e. on plateaus), the return value is `None`:

[30]:
```
assert s.get_time(2) is None
```

Closed curves are obviously not possible:

[31]:
```
try:
    splines.MonotoneCubic([0, 2, 2, 6, 6], closed=True)
except Exception as e:
    print(e)
    assert 'closed' in str(e)
else:
    assert False
```

```
The "closed" argument is not allowed
```

End Conditions

The usual end conditions (page 98) don't necessarily lead to a monotone interpolant, therefore we need to come up with custom end conditions that preserve monotonicity.

> For the end derivatives, the noncentered three point difference formula may be used, although it is sometimes necessary to modify d_1 and/or d_n if the signs are not appropriate. In these cases we have obtained better results setting d_1 or d_n equal to zero, rather than equal to the slope of the secant line.
>
> —[FC80], section 4: "Monotone piecewise cubic interpolation algorithm"

[FC80] recommends using the *noncentered three point difference formula*, however, it fails to mention what that actually is. Luckily, we can have a look at the code at https://people.sc.fsu.edu/~jburkardt/f77_src/pchip/pchip.html:

```
C
C  SET D(1) VIA NON-CENTERED THREE-POINT FORMULA, ADJUSTED TO BE
C     SHAPE-PRESERVING.
C
      HSUM = H1 + H2
      W1 = (H1 + HSUM)/HSUM
      W2 = -H1/HSUM
      D(1,1) = W1*DEL1 + W2*DEL2
      IF ( PCHST(D(1,1),DEL1) .LE. ZERO)  THEN
         D(1,1) = ZERO
      ELSE IF ( PCHST(DEL1,DEL2) .LT. ZERO)  THEN
C        NEED DO THIS CHECK ONLY IF MONOTONICITY SWITCHES.
         DMAX = THREE*DEL1
         IF (ABS(D(1,1)) .GT. ABS(DMAX))  D(1,1) = DMAX
      ENDIF
```

The function `PCHST` is a simple sign test:

```
PCHST = SIGN(ONE,ARG1) * SIGN(ONE,ARG2)
IF ((ARG1.EQ.ZERO) .OR. (ARG2.EQ.ZERO))  PCHST = ZERO
```

This implementation seems to be used by "modern" PCHIP/PCHIM implementations as well.

> This defines the `pchip` slopes at interior breakpoints, but the slopes d_1 and d_n at either end of the data interval are determined by a slightly different, one-sided analysis. The details are in `pchiptx.m`.
>
> —[Molo4], section 3.4

Section 3.6 of [Molo4] shows the implementation of `pchiptx.m`:

```
function d = pchipend(h1,h2,del1,del2)
%  Noncentered, shape-preserving, three-point formula.
    d = ((2*h1+h2)*del1 - h1*del2)/(h1+h2);
    if sign(d) ~= sign(del1)
        d = 0;
    elseif (sign(del1)~=sign(del2))&(abs(d)>abs(3*del1))
        d = 3*del1;
    end
```

Apparently, this is the same as the above Fortran implementation.

The class scipy.interpolate.PchipInterpolator[22] uses the same implementation (ported to Python)[23].

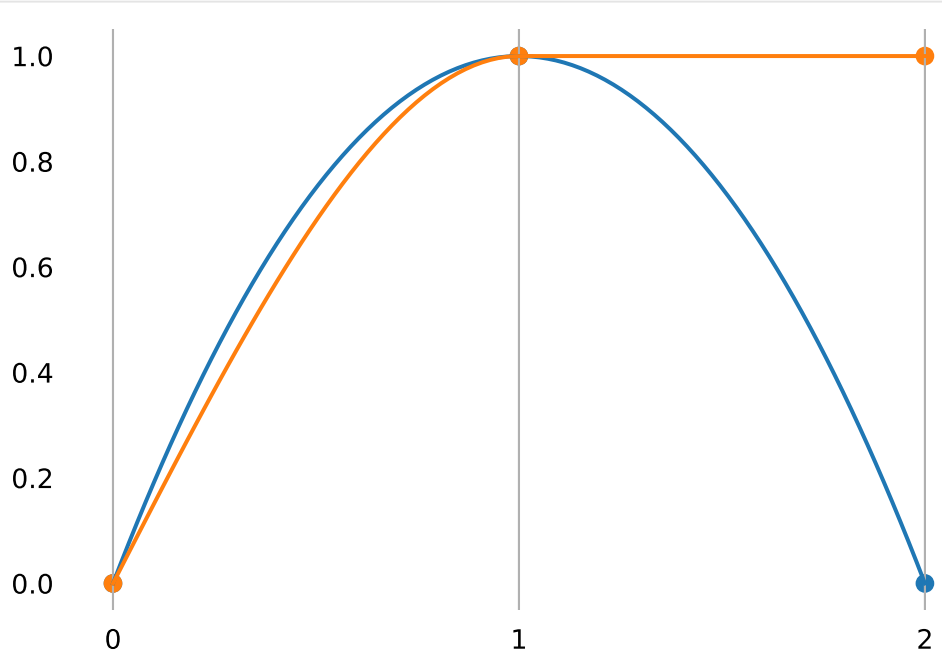
[22] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PchipInterpolator.html
[23] https://github.com/scipy/scipy/blob/v1.6.1/scipy/interpolate/_cubic.py#L237-L250

This implementation ensures monotonicity, but it might seem a bit strange that for calculating the first slope, the second slope is not directly taken into account.

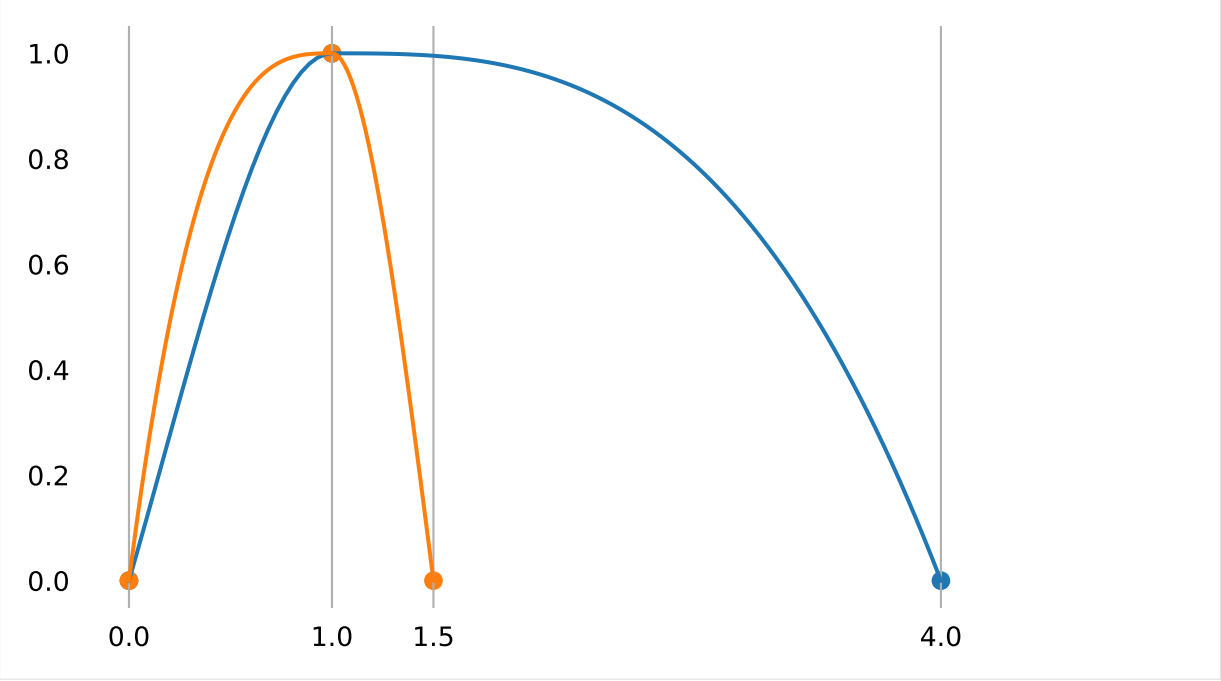
Another awkward property is that for calculating the inner slopes, only the immediately neighboring secant slopes and time intervals are considered, while for calculating the initial and final slopes, both the neighboring segment and the one next to it are considered. This makes the curve less locally controlled at the ends compared to the middle.

[32]:
```
def plot_pchip(values, grid, **kwargs):
    pchip = PchipInterpolator(grid, values)
    times = np.linspace(grid[0], grid[-1], 100)
    plt.plot(times, pchip(times), **kwargs)
    plt.scatter(grid, pchip(grid))
    grid_lines(x=grid)
```

[33]:
```
plot_pchip([0, 1, 0], [0, 1, 2])
plot_pchip([0, 1, 1], [0, 1, 2])
grid_lines([0, 1, 2])
```



[34]:
```
plot_pchip([0, 1, 0], [0, 1, 4])
plot_pchip([0, 1, 0], [0, 1, 1.5])
grid_lines([0, 1, 1.5, 4])
```

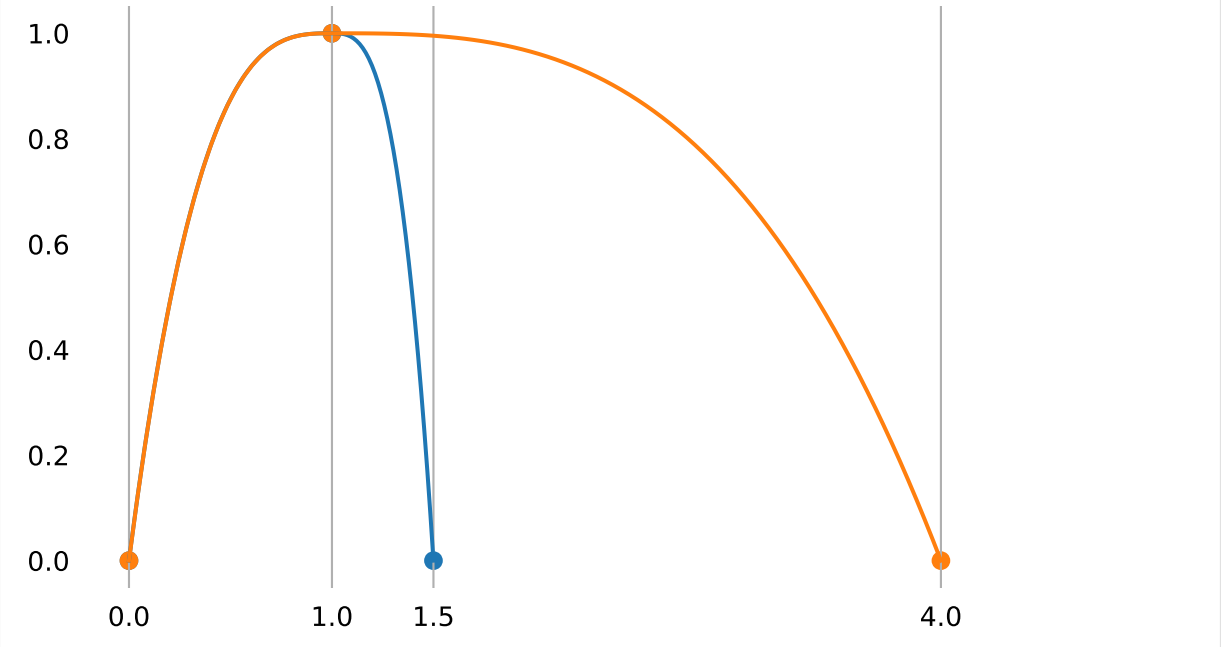


In both of the above examples, the very left slope depends on properties of the very right segment.

The slope at $t = 1$ is clearly zero in both cases and apart from that fact, the shape of the curve at $t > 1$ should, arguably, not have any influence on the slope at $t = 0$.

To provide an alternative to this behavior, the class *splines.PiecewiseMonotoneCubic* (page 151) uses end conditions that depend on the slope at $t = 1$, but not explicitly on the shape of the curve at $t > 1$:

[35]:
```
plot_piecewise_monotone([0, 1, 0], grid=[0, 1, 1.5])
plot_piecewise_monotone([0, 1, 0], grid=[0, 1, 4])
grid_lines(x=[0, 1, 1.5, 4])
```

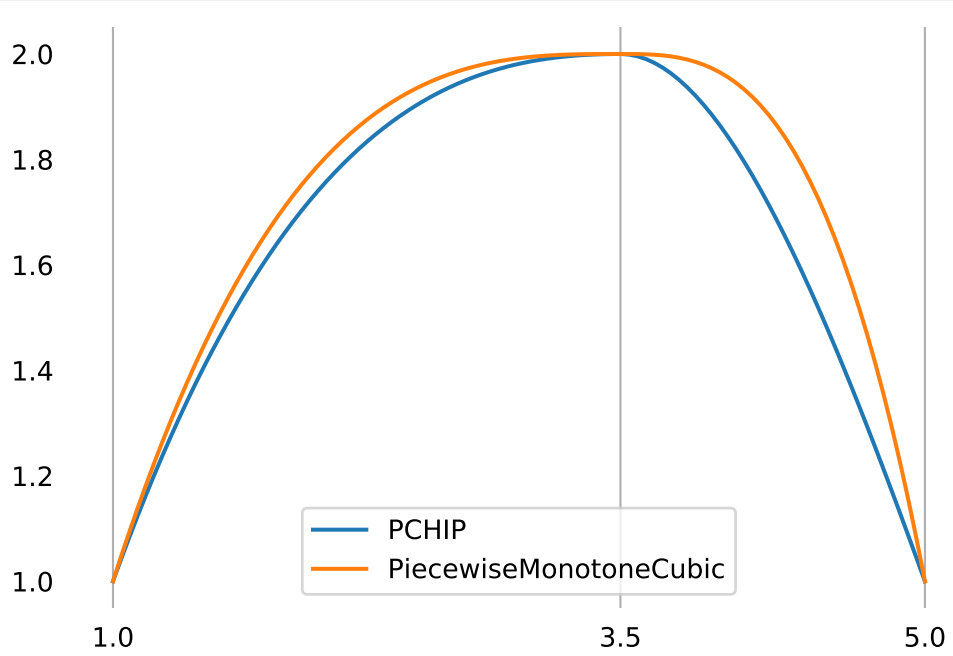


The initial and final slopes of *splines.PiecewiseMonotoneCubic* (page 151) are implemented like this:

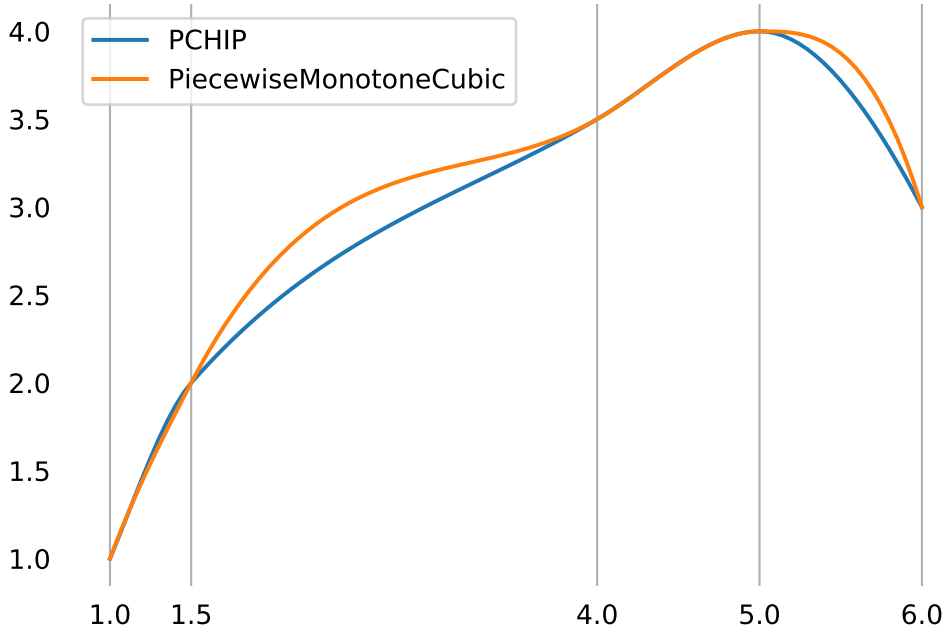
[36]:
```
def monotone_end_condition(inner_slope, secant_slope):
    if secant_slope < 0:
        return -monotone_end_condition(-inner_slope, -secant_slope)
    assert 0 <= inner_slope <= 3 * secant_slope
    if inner_slope <= secant_slope:
        return 3 * secant_slope - 2 * inner_slope
    else:
        return (3 * secant_slope - inner_slope) / 2
```

Even More Examples

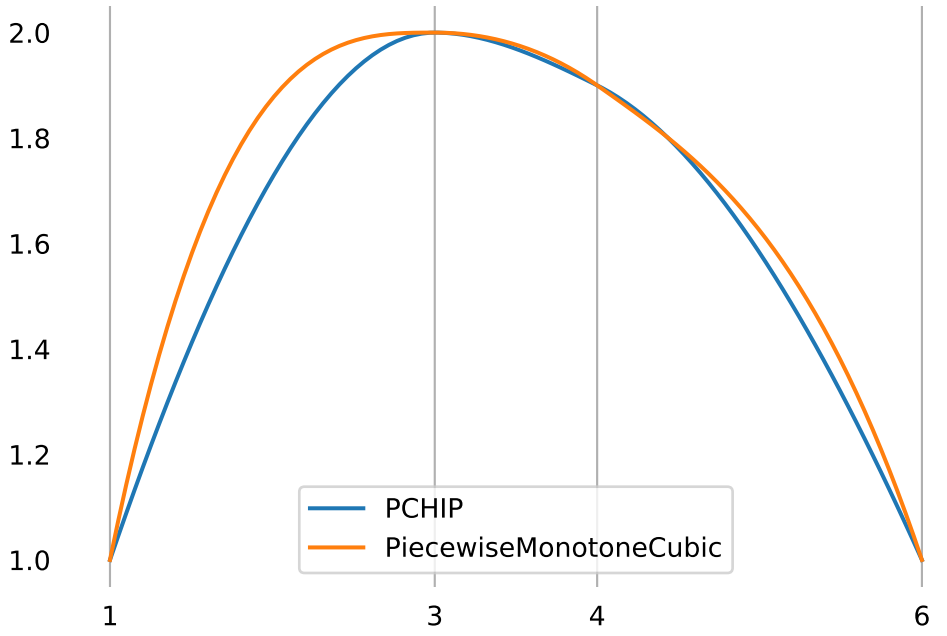
[37]:
```
compare_pchip([1, 2, 1], [1, 3.5, 5])
```



[38]:
```
compare_pchip([1, 2, 3.5, 4, 3], [1, 1.5, 4, 5, 6])
```



[39]:
```
compare_pchip([1, 2, 1.9, 1], [1, 3, 4, 6])
```



doc/euclidean/piecewise-monotone.ipynb ends here.

2 Rotation Splines

The following section was generated from `doc/rotation/quaternions.ipynb`

2.1 Quaternions

splines.quaternion.Quaternion (page 153)

splines.quaternion.UnitQuaternion (page 153)

helper.py

[1]:
```
from helper import angles2quat, plot_rotation
```

Quaternion Representations

Algebraic

$$q = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

where $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$.

The order matters: $\mathbf{ij} = \mathbf{k}$, $\mathbf{ji} = -\mathbf{k}$.

Scalar and Vector

$$q = (w, \vec{v}) = (w, (x, y, z))$$

Sometimes, the scalar and vector parts are also called "real" and "imaginary" parts, respectively.

4D Space

Quaternions can also be imagined as four dimensional vector space with some additional operations.

$$q = (w, x, y, z) \quad \text{or} \quad q = (x, y, z, w)$$

More Representations

There are even more ways to represent quaterions, for example as 4x4 real matrices or as 2x2 complex matrices.

Quaternion Operations

TODO

Unit Quaternions as Rotations

Given a (normalized) rotation axis $\vec{n}_i$ and a rotation angle α_i (in radians), we can create a corresponding quaternion (which will have unit length):

$$q_i = (\cos\frac{\alpha_i}{2}, \vec{n}_i \sin\frac{\alpha_i}{2})$$

Quaternions are a *double cover* over the rotation group (a.k.a. SO(3)[24]), which means that each rotation can be associated with two distinct quaternions. More concretely, the antipodal points q and $-q$ represent the same rotation.

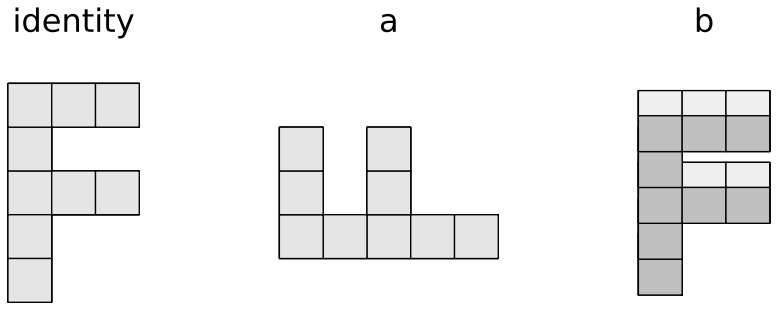
More information can be found on Wikipedia[25].

[2]:
```
identity = angles2quat(0, 0, 0)
```

[3]:
```
a = angles2quat(90, 0, 0)
b = angles2quat(0, 35, 0)
```

[4]:
```
plot_rotation({
    'identity': identity,
    'a': a,
    'b': b,
}, figsize=(6, 2));
```

[4]:
```
[<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f28a2f4af50>,
 <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f28a5068550>,
 <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f28a2f2c310>]
```



[5]:
```
identity
```

[5]:
```
UnitQuaternion(scalar=1.0, vector=(0.0, 0.0, 0.0))
```

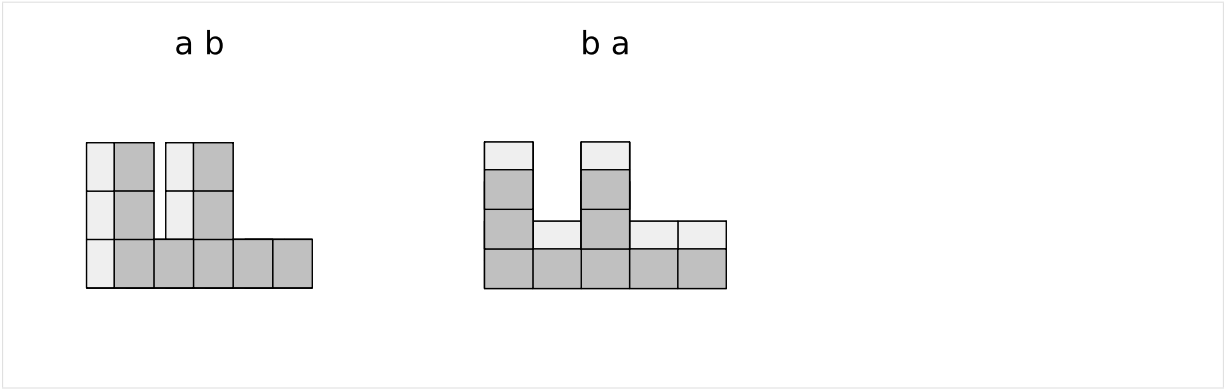
Unit Quaternion Operations

- quaternion multiplication: $q_1 q_0$
- rotation q_0 followed by rotation q_1 (read from right to left)
- $q_0 q_1 \neq q_1 q_0$ (except if the rotation axis is the same)

[6]:
```
plot_rotation({'a b': a * b, 'b a': b * a}, figsize=(5, 2));
```

[6]:
```
[<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f28a2b7fe90>,
 <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f28a2b57450>]
```

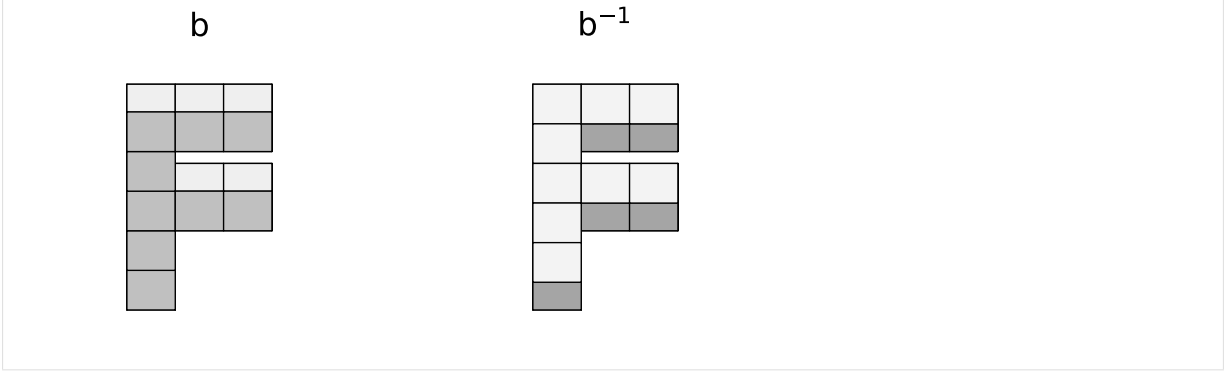
[24] https://en.wikipedia.org/wiki/3D_rotation_group
[25] https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation



- inverse: q^{-1}
- same rotation axis, negated angle
- $qq^{-1} = q^{-1}q = (1, (0, 0, 0))$

[7]:
```
plot_rotation({'b': b, 'b$^{-1}$': b.inverse()}, figsize=(5, 2));
```

[7]:
```
[<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f28a2a61350>,
 <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f28a2e9c950>]
```



Relative Rotation (Global Frame of Reference)

Given two rotations q_0 and q_1, we can try to find a third rotation $q_{0,1}$ that rotates q_0 into q_1. Since we are considering the global frame of reference, $q_{0,1}$ must be left-multiplied with q_0:

$$q_{0,1}q_0 = q_1$$

Now we can right-multiply both sides with ${q_0}^{-1}$:

$$q_{0,1}q_0{q_0}^{-1} = q_1{q_0}^{-1}$$

$q_0{q_0}^{-1}$ cancels out and we get:

$$q_{0,1} = q_1{q_0}^{-1}$$

Relative Rotation (Local Frame of Reference)

If $q_{0,1}$ is supposed to be a rotation in the local frame of q_0, we have to change the order of multiplication:

$$q_0 q_{0,1} = q_1$$

Now we can left-multiply both sides with ${q_0}^{-1}$:

$${q_0}^{-1} q_0 q_{0,1} = {q_0}^{-1} q_1$$

${q_0}^{-1} q_0$ cancels out and we get:

$$q_{0,1} = {q_0}^{-1} q_1$$

doc/rotation/quaternions.ipynb ends here.

The following section was generated from doc/rotation/slerp.ipynb

2.2 Spherical Linear Interpolation (Slerp)

The term "Slerp" for "**s**pherical **l**inear int**erp**olation" (a.k.a. "great arc in-betweening") has been coined by [Sho85] (section 3.3). It is defined as:

$$\text{Slerp}(q_1, q_2; u) = q_1 \left({q_1}^{-1} q_2\right)^u$$

The parameter u moves from 0 (where the expression simplifies to q_1) to 1 (where the expression simplifies to q_2).

The Wikipedia article for Slerp[26] provides four equivalent ways to describe the same thing:

$$\begin{aligned}
\text{Slerp}(q_0, q_1; t) &= q_0 \left({q_0}^{-1} q_1\right)^t \\
&= q_1 \left({q_1}^{-1} q_0\right)^{1-t} \\
&= \left(q_0 {q_1}^{-1}\right)^{1-t} q_1 \\
&= \left(q_1 {q_0}^{-1}\right)^t q_0
\end{aligned}$$

[Sho85] also provides an alternative formulation (attributed to Glenn Davis):

$$\text{Slerp}(q_1, q_2; u) = \frac{\sin(1-u)\theta}{\sin\theta} q_1 + \frac{\sin u\theta}{\sin\theta} q_2,$$

where the dot product $q_1 \cdot q_2 = \cos\theta$.

Latter equation works for unit-length elements of any arbitrary-dimensional inner product space (i.e. a vector space that also has an inner product), while the preceding equations only work for quaternions.

The Slerp function is quite easy to implement ...

[1]:
```
def slerp(one, two, t):
    return (two * one.inverse())**t * one
```

... but for your convenience an implementation is also provided in *splines.quaternion.slerp()* (page 154).

[26] https://en.wikipedia.org/wiki/Slerp#Quaternion_Slerp

Derivation

Before looking at the general case $\text{Slerp}(q_0, q_1; t)$, which interpolates from q_0 to q_1, let's look at the much simpler case of interpolating from the identity $\mathbf{1}$ to some unit quaternion q.

$$\mathbf{1} = (1, (0, 0, 0))$$
$$q = \left(\cos\frac{\alpha}{2}, \vec{n}\sin\frac{\alpha}{2}\right)$$

To move along the great arc from $\mathbf{1}$ to q, we simply have to change the angle from 0 to α while the rotation axis $\vec{n}$ stays unchanged.

$$\text{Slerp}(\mathbf{1}, q; t) = \left(\cos\frac{\alpha t}{2}, \vec{n}\sin\frac{\alpha t}{2}\right) = q^t, \text{ where } 0 \leq t \leq 1$$

To generalize this to the great arc from q_0 to q_1, we can start with q_0 and left-multiply an appropriate Slerp using the *relative rotation (global frame)* (page 119) $q_{0,1}$:

$$\text{Slerp}(q_0, q_1; t) = \text{Slerp}(\mathbf{1}, q_{0,1}; t)\, q_0$$

Inserting $q_{0,1} = q_1 {q_0}^{-1}$, we get:

$$\text{Slerp}(q_0, q_1; t) = \left(q_1 {q_0}^{-1}\right)^t q_0$$

Alternatively, we can start with q_0 and right-multiply an appropriate Slerp using the *relative rotation (local frame)* (page 120) $q_{0,1} = {q_0}^{-1} q_1$:

$$\text{Slerp}(q_0, q_1; t) = q_0 \left({q_0}^{-1} q_1\right)^t$$

We can also start with q_1, swap q_0 and q_1 in the relative rotation and invert the parameter by using $1 - t$, leading to the two further alternatives mentioned above.

Visualization

First, let's import NumPy[27] ...

[2]:
```
import numpy as np
```

... and a few helper functions from helper.py:

[3]:
```
from helper import angles2quat, animate_rotations, display_animation
```

[4]:
```
q1 = angles2quat(45, -20, -60)
q2 = angles2quat(-45, 20, 30)
```

The angle between the two quaternions:

[5]:
```
np.degrees(np.arccos(q1.dot(q2)) * 2)
```

[5]:
```
123.9513586527906
```

[6]:
```
ani_times = np.linspace(0, 1, 50)
```

[7]:
```
ani = animate_rotations({
    'slerp(q1, q2)': slerp(q1, q2, ani_times),
    'slerp(q1, -q2)': slerp(q1, -q2, ani_times),
}, figsize=(6, 3))
```

[27] https://numpy.org/

[8]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

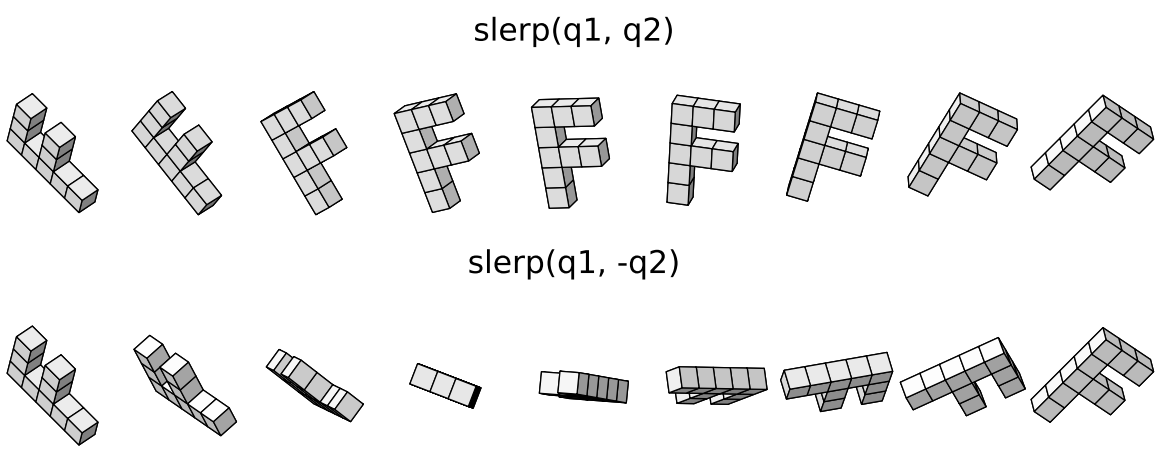
`slerp(q1, q2)` and `slerp(q1, -q2)` move along the same great circle, albeit in different directions. In total, they cover half the circumference of that great circle, which means a rotation angle of 360 degrees.

Let's create some still images:

[9]:
```
from helper import plot_rotations
```

[10]:
```
plot_times = np.linspace(0, 1, 9)
```

[11]:
```
plot_rotations({
    'slerp(q1, q2)': slerp(q1, q2, plot_times),
    'slerp(q1, -q2)': slerp(q1, -q2, plot_times),
}, figsize=(8, 3))
```



Piecewise Slerp

The class *PiecewiseSlerp* (page 154) provides a rotation spline that consists of Slerp sections between the given quaternions.

[12]:
```
from splines.quaternion import PiecewiseSlerp
```

[13]:
```
s = PiecewiseSlerp([
    angles2quat(0, 0, 0),
    angles2quat(90, 0, 0),
    angles2quat(90, 90, 0),
    angles2quat(90, 90, 90),
], grid=[0, 1, 2, 3, 6], closed=True)
```

[14]:
```
ani = animate_rotations({
    'piecewise Slerp': s.evaluate(np.linspace(s.grid[0], s.grid[-1], 100)),
}, figsize=(3, 3))
```

[15]:
```
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

Each section has a constant rotation angle and a constant angular velocity.

Slerp vs. Nlerp

While *Slerp* interpolates along a great arc between two quaternions, it is also possible to interpolate along a straight line (in four-dimensional quaternion space) between those two quaternions.

The resulting interpolant is *not* part of the unit hypersphere, i.e. the interpolated values are not unit quaternions. However, they can be normalized to become unit quaternions.

This is called "**n**ormalized **l**inear int**erp**olation", in short *Nlerp*.

The resulting interpolant travels through the same quaternions as Slerp does, but it doesn't do it with constant angular velocity.

[16]:
```
from splines.quaternion import Quaternion
```

[17]:
```
def lerp(one, two, t):
    """Linear interpolation.

    t can go from 0 to 1.

    """
    return (1 - t) * one + t * two
```

[18]:
```
def nlerp(one, two, t):
    """Normalized linear interpolation.

    Linear interpolation in 4D quaternion space,
    normalizing the result.

    t can go from 0 to 1.

    """
    one = np.array(one.xyzw)
    two = np.array(two.xyzw)
    *vector, scalar = lerp(one, two, t)
    return Quaternion(scalar, vector).normalize()
```

[19]:
```
q1 = angles2quat(-60, 10, -10)
q2 = angles2quat(80, -35, -110)
```

[20]:
```
assert q1.dot(q2) > 0
```

[21]:
```
np.degrees(np.arccos(q1.dot(q2)) * 2)
```

[21]:
```
174.5768498146622
```

[22]:
```
ani_times = np.linspace(0, 1, 50)
```

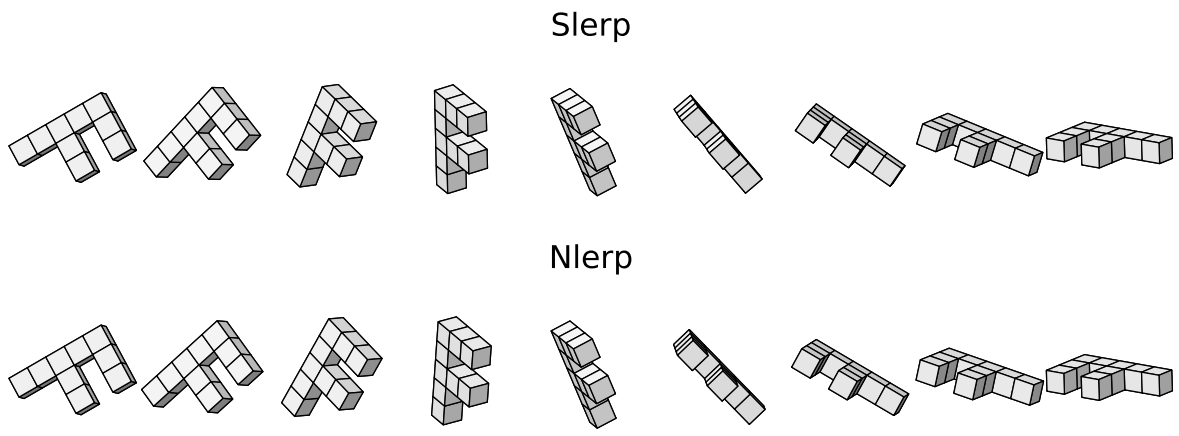
[23]:
```
ani = animate_rotations({
    'Slerp': slerp(q1, q2, ani_times),
    'Nlerp': [nlerp(q1, q2, t) for t in ani_times],
}, figsize=(6, 3))
```

[24]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

Some still images:

[25]:
```
plot_rotations({
    'Slerp': slerp(q1, q2, plot_times),
    'Nlerp': [nlerp(q1, q2, t) for t in plot_times],
}, figsize=(8, 3))
```



Start and end are (by definition) the same, the middle is also the same (due to symmetry). And in between, there are very slight differences.

Since the differences are barely visible, we can try a more extreme example:

[26]:
```
q3 = angles2quat(-170, 0, 45)
q4 = angles2quat(120, -90, -45)
```

[27]:
```
assert q3.dot(q4) < 0
```

[28]:
```
np.degrees(np.arccos(q3.dot(q4)) * 2)
```

[28]:
```
268.2720589276495
```

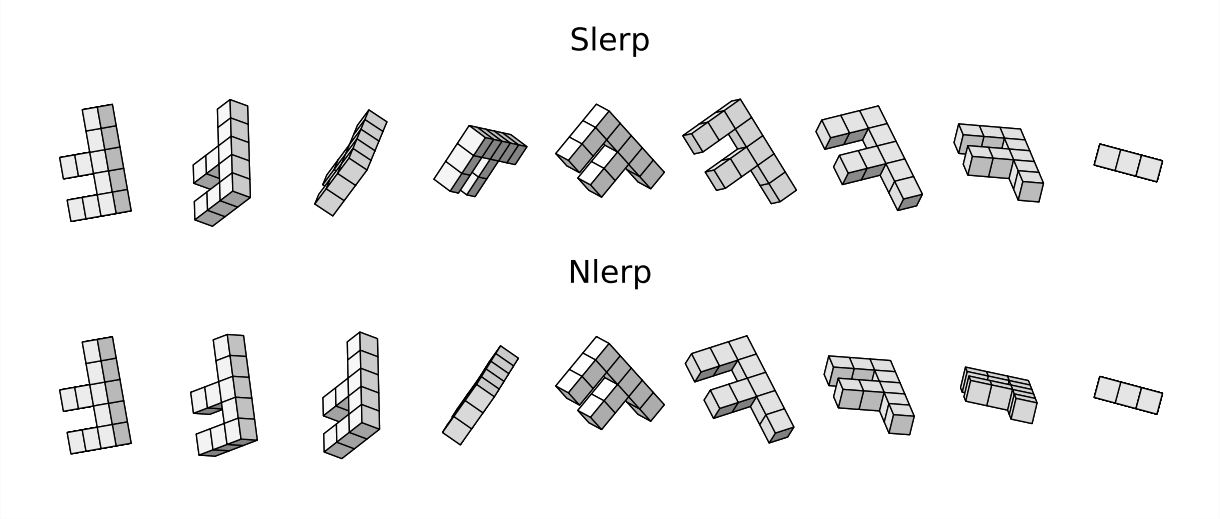
Please note that this is a rotation by an angle of far more than 180 degrees!

[29]:
```
ani = animate_rotations({
    'Slerp': slerp(q3, q4, ani_times),
    'Nlerp': [nlerp(q3, q4, t) for t in ani_times],
}, figsize=(6, 3))
```

[30]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

[31]:
```
plot_rotations({
    'Slerp': slerp(q3, q4, plot_times),
    'Nlerp': [nlerp(q3, q4, t) for t in plot_times],
}, figsize=(8, 3))
```



doc/rotation/slerp.ipynb ends here.

The following section was generated from doc/rotation/de-casteljau.ipynb

2.3 De Casteljau's Algorithm

In [Sho85], which famously introduces quaternions to the field of computer graphics, Shoemake suggests to apply a variant of the *De Casteljau's Algorithm* (page 39) to a quaternion control polygon, using *Slerp* (page 120) instead of linear interpolations.

[1]:
```
def slerp(one, two, t):
    return (two * one.inverse())**t * one
```

[2]:
```
import numpy as np
```

helper.py

[3]:
```
from helper import angles2quat, animate_rotations, display_animation
```

"Cubic"

[Sho85] only talks about the "cubic" case, consisting of three nested applications of Slerp.

The resulting curve is of course not simply a polynomial of degree 3, but something quite a bit more involved. Therefore, we use the term "cubic" in quotes.

Shoemake doesn't talk about the "degree" of the curves at all, they are only called "spherical Bézier curves".

[4]:
```
def de_casteljau(q0, q1, q2, q3, t):
    slerp_0_1 = slerp(q0, q1, t)
    slerp_1_2 = slerp(q1, q2, t)
    slerp_2_3 = slerp(q2, q3, t)
    return slerp(
        slerp(slerp_0_1, slerp_1_2, t),
        slerp(slerp_1_2, slerp_2_3, t),
        t,
    )
```

[5]:
```
q0 = angles2quat(45, 0, 0)
q1 = angles2quat(0, 0, 0)
q2 = angles2quat(-90, 90, -90)
q3 = angles2quat(-90, 0, 0)
```

[6]:
```
times = np.linspace(0, 1, 100)
```

[7]:
```
ani = animate_rotations(
    [de_casteljau(q0, q1, q2, q3, t) for t in times],
    figsize=(3, 2),
)
```

[8]:
```
display_animation(ani, default_mode='once')
```

```
Animations can only be shown in HTML output, sorry!
```

Arbitrary "Degree"

splines.quaternion.DeCasteljau (page 155) class

[9]:
```
from splines.quaternion import DeCasteljau
```

[10]:
```
s = DeCasteljau([
    [
        angles2quat(0, 0, 0),
        angles2quat(90, 0, 0),
    ],
    [
        angles2quat(90, 0, 0),
        angles2quat(0, 0, 0),
        angles2quat(0, 90, 0),
    ],
    [
        angles2quat(0, 90, 0),
        angles2quat(0, 0, 0),
        angles2quat(-90, 0, 0),
        angles2quat(-90, 90, 0),
    ],
], grid=[0, 1, 3, 6])
```

[11]:
```
times = np.linspace(s.grid[0], s.grid[-1], 100)
```

[12]:
```
ani = animate_rotations(s.evaluate(times), figsize=(3, 2))
```

[13]:
```
display_animation(ani, default_mode='once')
```

```
Animations can only be shown in HTML output, sorry!
```

Constant Angular Speed

> Is there a way to construct a curve parameterized by arc length? This would be very useful.
>
> —[Sho85], section 6: "Questions"

[14]:
```
from splines import ConstantSpeedAdapter
```

[15]:
```
s1 = DeCasteljau([[
    angles2quat(90, 0, 0),
    angles2quat(0, -45, 90),
    angles2quat(0, 0, 0),
    angles2quat(180, 0, 180),
]])
```

[16]:
```
s2 = ConstantSpeedAdapter(s1)
```

[17]:
```
ani = animate_rotations({
    'non-constant speed': s1.evaluate(np.linspace(s1.grid[0], s1.grid[-1], 100)),
    'constant speed': s2.evaluate(np.linspace(s2.grid[0], s2.grid[-1], 100)),
}, figsize=(5, 2))
```

[18]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

Joining Curves

In section 4.2, [Sho85] provides two function definitions:

$$\text{Double}(p, q) = 2(p \cdot q)q - p$$
$$\text{Bisect}(p, q) = \frac{p + q}{\|p + q\|}$$

[19]:
```
def double(p, q):
    return 2 * p.dot(q) * q - p
```

[20]:
```
def bisect(p, q):
    return (p + q).normalize()
```

Given three successive key quaternions q_{n-1}, q_n and q_{n+1}, these functions are used to compute control quaternions b_n (controlling the incoming tangent of q_n) and a_n (controlling the outgoing tangent of q_n):

$$a_n = \text{Bisect}(\text{Double}(q_{n-1}, q_n), q_{n+1})$$
$$b_n = \text{Double}(a_n, q_n)$$

It is unclear where these equations come from, we only get a little hint:

> For the numerically knowledgeable, this construction approximates the derivative at points of a sampled function by averaging the central differences of the sample sequence.
>
> —[Sho85], footnote on page 249

[21]:
```
def shoemake_control_quaternions(q_1, q0, q1):
    """Shoemake's control quaternions (part 1).

    Given three key quaternions, return the control quaternions
    preceding and following the middle one.

    """
    a = bisect(double(q_1, q0), q1)
    b = double(a, q0).normalize()
    return b, a
```

Normalization of b_n is not explicitly mentioned in the paper, but even though the results have a length very close to `1.0`, we still have to call `normalize()` to turn the *Quaternion* (page 153) result into a *UnitQuaternion* (page 153).

But wait, we are not finished yet!

In a later section, the last missing piece of the puzzle is unveiled:

> A simple check proves the curve touches q_n and q_{n+1} at its ends. A rather challenging differentiation shows it is tangent there to the segments determined by a_n and b_{n+1}. However, as with Bézier's original curve, the magnitude of the tangent is three times that of the segment itself. That is, we are spinning three times faster than spherical interpolation along the arc. Fortunately we can correct the speed by merely truncating the end segments to one third their original length, so that a_n is closer to q_n and b_{n+1} closer to q_{n+1}.
>
> —[Sho85], section 4.4: "Tangents revisited"

No further hints are given about how to actually implement this, but it should work something like this:

[22]:
```
def shoemake_corrected_control_quaternions(q_1, q0, q1):
    """Shoemake_s control quaternions, corrected."""
    b, a = shoemake_control_quaternions(q_1, q0, q1)
    return q0.rotation_to(b)**(1/3) * q0, q0.rotation_to(a)**(1/3) * q0
```

We create a helper function for building a spline, because we will re-use the same thing further below:

[23]:
```
def create_closed_shoemake_curve(rotations):
    rotations = rotations + rotations[:2]
    control_points = []
    for q_1, q0, q1 in zip(rotations, rotations[1:], rotations[2:]):
        b, a = shoemake_corrected_control_quaternions(q_1, q0, q1)
        control_points.extend([b, q0, q0, a])
    control_points = control_points[-2:] + control_points[:-2]
    segments = list(zip(*[iter(control_points)] * 4))
    return DeCasteljau(segments)
```

We don't want to worry about end conditions here, therefore we create a closed curve.

Let's come up with some example rotations and see how Shoemake's curve-joining method works.

[24]:
```
rotations = [
    angles2quat(0, 0, 180),
    angles2quat(0, 45, 90),
    angles2quat(90, 45, 0),
    angles2quat(90, 90, -90),
    angles2quat(180, 0, -180),
    angles2quat(-90, -45, 180),
]
```

[25]:
```
s = create_closed_shoemake_curve(rotations)
```

[26]:
```
times = np.linspace(s.grid[0], s.grid[-1], 200)
```

[27]:
```
ani = animate_rotations(s.evaluate(times), figsize=(3, 2))
```

[28]:
```
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

Joining Curves, Another Approach

uniform Catmull–Rom-like quaternion splines (page 129)

splines.quaternion.CatmullRom (page 156)

[29]:
```
from splines.quaternion import CatmullRom
```

[30]:
```
cr = CatmullRom(rotations, endconditions='closed')
```

[31]:
```
ani = animate_rotations({
    "Shoemake's method": s.evaluate(times),
    'Other approach': cr.evaluate(times),
}, figsize=(5, 2))
```

[32]:
```
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

There is no visible difference between the two approaches. They are not quite identical, though:

[33]:
```
max(max(map(abs, q.xyzw)) for q in (s.evaluate(times) - cr.evaluate(times)))
```

[33]:
```
0.0082592514906695
```

doc/rotation/de-casteljau.ipynb ends here.

The following section was generated from doc/rotation/catmull-rom-uniform.ipynb

2.4 Uniform Catmull–Rom-Like Quaternion Splines

see *notebook about De Casteljau's algorithm (with Slerp)* (page 125)

notebook about Euclidean Catmull–Rom splines (page 54)

$$\dot{x}_i = \frac{x_{i+1} - x_{i-1}}{2} = \frac{(x_i - x_{i-1}) + (x_{i+1} - x_i)}{2} = \frac{x_i - x_{i-1}}{2} + \frac{x_{i+1} - x_i}{2}$$

section about cubic Euclidean Bézier splines (page 46): division by 3

$$q_{i,\text{tangent}} \overset{?}{=} \begin{cases} \left(q_{i+1} {q_{i-1}}^{-1}\right)^{\frac{1}{2}\frac{1}{3}} \\ \left(q_\text{out} q_\text{in}\right)^{\frac{1}{2}\frac{1}{3}} \\ \left({q_\text{out}}^{\frac{1}{2}} {q_\text{in}}^{\frac{1}{2}}\right)^{\frac{1}{3}} \\ {q_\text{out}}^{\frac{1}{2}\frac{1}{3}} {q_\text{in}}^{\frac{1}{2}\frac{1}{3}} \\ \left(\left({q_\text{out}}^{\frac{1}{2}\frac{1}{3}} {q_\text{in}}^{-1\frac{1}{2}\frac{1}{3}}\right)^{\frac{1}{2}} {q_\text{in}}^{\frac{1}{2}\frac{1}{3}}\right)^2 = \left(\left({q_\text{in}}^{\frac{1}{2}\frac{1}{3}} {q_\text{out}}^{-1\frac{1}{2}\frac{1}{3}}\right)^{\frac{1}{2}} {q_\text{out}}^{\frac{1}{2}\frac{1}{3}}\right)^2 \\ \exp\left(\frac{1}{2}\frac{1}{3}(\ln(q_\text{in}) + \ln(q_\text{out}))\right) \end{cases}$$

where $q_{\text{in}} = q_i {q_{i-1}}^{-1}$ and $q_{\text{out}} = q_{i+1} {q_i}^{-1}$.

The first four options are quite certainly wrong (although not by much), so we are only looking at the last two here:

[1]:
```
def tangent1(q_1, q0, q1):
    q_in = (q0 * q_1.inverse())**(1 / (2 * 3))
    q_out = (q1 * q0.inverse())**(1 / (2 * 3))
    return ((q_out * q_in.inverse())**(1 / 2) * q_in)**2
```

[2]:
```
def tangent2(q_1, q0, q1):
    q_in = q0 * q_1.inverse()
    q_out = q1 * q0.inverse()
    return UnitQuaternion.exp_map((q_in.log_map() + q_out.log_map()) / (2 * 3))
```

[3]:
```
import numpy as np
```

[4]:
```
from splines.quaternion import DeCasteljau, UnitQuaternion, canonicalized
```

helper.py

[5]:
```
from helper import angles2quat, animate_rotations, display_animation
```

[6]:
```
def create_closed_curve(rotations, tangent_func):
    rotations = list(canonicalized(rotations + rotations[:2]))
    control_points = []
    for q_1, q0, q1 in zip(rotations, rotations[1:], rotations[2:]):
        q_tangent = tangent_func(q_1, q0, q1)
        control_points.extend([q_tangent.inverse() * q0, q0, q0, q_tangent * q0])
    control_points = control_points[-2:] + control_points[:-2]
    segments = list(zip(*[iter(control_points)] * 4))
    return DeCasteljau(segments)
```

[7]:
```
rotations = [
    angles2quat(0, 0, 180),
    angles2quat(0, 45, 90),
    angles2quat(90, 45, 0),
    angles2quat(90, 90, -90),
    angles2quat(180, 0, -180),
    angles2quat(-90, -45, 180),
]
```

[8]:
```
s1 = create_closed_curve(rotations, tangent1)
s2 = create_closed_curve(rotations, tangent2)
```

[9]:
```
times = np.linspace(0, len(rotations), 200, endpoint=False)
```

[10]:
```
ani = animate_rotations({
    '1': s1.evaluate(times),
    '2': s2.evaluate(times),
}, figsize=(4, 2))
```

[11]:
```
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

The results are very similar, but not quite identical:

[12]:
```
max(max(map(abs, q.xyzw)) for q in (s1.evaluate(times) - s2.evaluate(times)))
```

[12]:
```
0.0002228419691587824
```

doc/rotation/catmull-rom-uniform.ipynb ends here.

The following section was generated from doc/rotation/catmull-rom-non-uniform.ipynb

2.5 Non-Uniform Catmull–Rom-Like Rotation Splines

uniform case (page 129)

> What is the best way to allow varying intervals between sequence points in parameter space?
>
> —[Sho85], section 6: "Questions"

notebook about non-uniform (Euclidean) Catmull–Rom splines (page 72)

$$\vec{v}_i = \frac{\vec{v}_{i,\text{in}}(t_{i+1} - t_i) + \vec{v}_{i,\text{out}}(t_i - t_{i-1})}{t_{i+1} - t_{i-1}}$$

$$\begin{aligned}\vec{v}_{i,\text{in}} &= \frac{\vec{x}_i - \vec{x}_{i-1}}{t_i - t_{i-1}} \\ \vec{v}_{i,\text{out}} &= \frac{\vec{x}_{i+1} - \vec{x}_i}{t_{i+1} - t_i}\end{aligned}$$

"translated" to quaternions …

$$\begin{aligned}\tilde{q}_i^{(-)} &= \exp\left(\vec{\omega}_i \frac{t_i - t_{i-1}}{3}\right)^{-1} q_i \\ \tilde{q}_i^{(+)} &= \exp\left(\vec{\omega}_i \frac{t_{i+1} - t_i}{3}\right) q_i\end{aligned}$$

$$\begin{aligned}\vec{\omega}_i &= \frac{\vec{\omega}_{i,\text{in}}(t_{i+1} - t_i) + \vec{\omega}_{i,\text{out}}(t_i - t_{i-1})}{t_{i+1} - t_{i-1}} \\ \vec{\omega}_{i,\text{in}} &= \frac{\ln(q_{i,\text{in}})}{t_i - t_{i-1}} \\ \vec{\omega}_{i,\text{out}} &= \frac{\ln(q_{i,\text{out}})}{t_{i+1} - t_i} \\ q_{i,\text{in}} &= q_i q_{i-1}^{-1} \\ q_{i,\text{out}} &= q_{i+1} q_i^{-1}\end{aligned}$$

factor of $\frac{1}{3}$ because we are dealing with cubic splines

[1]:
```
from splines.quaternion import UnitQuaternion
```

[2]:
```
def control_quaternions1(qs, ts):
    q_1, q0, q1 = qs
    t_1, t0, t1 = ts
    q_in = q0 * q_1.inverse()
    q_out = q1 * q0.inverse()
    w_in = q_in.log_map() / (t0 - t_1)
    w_out = q_out.log_map() / (t1 - t0)
    w0 = ((t1 - t0) * w_in + (t0 - t_1) * w_out) / (t1 - t_1)
    return [
```

(continues on next page)

(continued from previous page)

```
        UnitQuaternion.exp_map(-w0 * (t0 - t_1) / 3) * q0,
        UnitQuaternion.exp_map(w0 * (t1 - t0) / 3) * q0,
    ]
```

similar, but not quite identical:

[3]:
```
def control_quaternions2(qs, ts):
    q_1, q0, q1 = qs
    t_1, t0, t1 = ts
    q_in = (q0 * q_1.inverse())**((t1 - t0) / (3 * (t0 - t_1)))
    q_out = (q1 * q0.inverse())**((t0 - t_1) / (3 * (t1 - t0)))
    q_tangent = ((q_out * q_in.inverse())**(1 / 2) * q_in)**2
    return [
        (q_tangent**((t0 - t_1) / (t1 - t_1))).inverse() * q0,
        q_tangent**((t1 - t0) / (t1 - t_1)) * q0,
    ]
```

[4]:
```
import numpy as np
```

helper.py

[5]:
```
from helper import angles2quat, animate_rotations, display_animation
```

splines.quaternion.DeCasteljau (page 155) class

[6]:
```
from splines.quaternion import DeCasteljau, canonicalized
```

[7]:
```
def create_closed_curve(rotations, grid, control_quaternion_func):
    assert len(rotations) + 1 == len(grid)
    rotations = rotations[-1:] + rotations + rotations[:2]
    # Avoid angles of more than 180 degrees (including the added rotations):
    rotations = list(canonicalized(rotations))
    first_interval = grid[1] - grid[0]
    last_interval = grid[-1] - grid[-2]
    extended_grid = [grid[0] - last_interval] + list(grid) + [grid[-1] + first_interval]
    control_points = []
    for qs, ts in zip(
            zip(rotations, rotations[1:], rotations[2:]),
            zip(extended_grid, extended_grid[1:], extended_grid[2:])):
        q_before, q_after = control_quaternion_func(qs, ts)
        control_points.extend([q_before, qs[1], qs[1], q_after])
    control_points = control_points[2:-2]
    segments = list(zip(*[iter(control_points)] * 4))
    return DeCasteljau(segments, grid)
```

[8]:
```
rotations = [
    angles2quat(0, 0, 180),
    angles2quat(0, 45, 90),
    angles2quat(90, 45, 0),
    angles2quat(90, 90, -90),
    angles2quat(180, 0, -180),
    angles2quat(-90, -45, 180),
]
```

[9]:
```
grid = np.array([0, 0.5, 2, 5, 6, 7, 10])
```

[10]:
```
s1 = create_closed_curve(rotations, grid, control_quaternions1)
s2 = create_closed_curve(rotations, grid, control_quaternions2)
```

[11]:
```
def evaluate(spline, frames=200):
    times = np.linspace(spline.grid[0], spline.grid[-1], frames, endpoint=False)
    return spline.evaluate(times)
```

for comparison, *Barry–Goldman* (page 138)

[12]:
```
from splines.quaternion import BarryGoldman
```

[13]:
```
bg = BarryGoldman(rotations, grid)
```

[14]:
```
ani = animate_rotations({
    '1': evaluate(s1),
    '2': evaluate(s2),
    'Barry-Goldman': evaluate(bg),
}, figsize=(6, 2))
```

[15]:
```
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

[16]:
```
max(max(map(abs, q.xyzw)) for q in (evaluate(s1) - evaluate(s2)))
```

[16]:
```
0.01051294090598398
```

Parameterization

[17]:
```
rotations = [
    angles2quat(90, 0, -45),
    angles2quat(179, 0, 0),
    angles2quat(181, 0, 0),
    angles2quat(270, 0, -45),
    angles2quat(0, 90, 90),
]
```

[18]:
```
uniform2 = create_closed_curve(rotations, range(len(rotations) + 1), control_quaternions1)
```

chordal parameterization (page 58)

[19]:
```
angles = np.array([
    np.arccos(a.dot(b))
    #np.arccos(a.dot(b)) * 2
    #a.dot(b)
    #(b * a.inverse()).angle
    #(b - a).norm
    for a, b in zip(rotations, rotations[1:] + rotations[:1])])
angles
```

[19]:
```
array([0.85136439, 0.01745329, 0.85136439, 1.29678212, 0.85888576])
```

[20]:
```
chordal_grid = np.concatenate([[0], np.cumsum(angles)])
```

[21]:
```
chordal2 = create_closed_curve(rotations, chordal_grid, control_quaternions1)
```

centripetal parameterization (page 59)

[22]:
```
centripetal_grid = np.concatenate([[0], np.cumsum(np.sqrt(angles))])
```

[23]:
```
centripetal2 = create_closed_curve(rotations, centripetal_grid, control_quaternions1)
```

[24]:
```
ani = animate_rotations({
    'uniform': evaluate(uniform2),
    'chordal': evaluate(chordal2),
    'centripetal': evaluate(centripetal2),
}, figsize=(6, 2))
```

[25]:
```
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

The other method is very similar:

[26]:
```
uniform1 = create_closed_curve(rotations, range(len(rotations) + 1), control_quaternions2)
chordal1 = create_closed_curve(rotations, chordal_grid, control_quaternions2)
centripetal1 = create_closed_curve(rotations, centripetal_grid, control_quaternions2)
```

[27]:
```
max(max(map(abs, q.xyzw)) for q in (evaluate(uniform1) - evaluate(uniform2)))
```

[27]:
```
0.0026683373798292997
```

[28]:
```
max(max(map(abs, q.xyzw)) for q in (evaluate(chordal1) - evaluate(chordal2)))
```

[28]:
```
0.002259641902391918
```

[29]:
```
max(max(map(abs, q.xyzw)) for q in (evaluate(centripetal1) - evaluate(centripetal2)))
```

[29]:
```
0.002486960572741559
```

doc/rotation/catmull-rom-non-uniform.ipynb ends here.

The following section was generated from doc/rotation/kochanek-bartels.ipynb

2.6 Kochanek–Bartels-like Rotation Splines

[1]:
```
import numpy as np
```

helper.py

[2]:
```
from helper import angles2quat, animate_rotations, display_animation
```

splines.quaternion.KochanekBartels (page 155)

[3]:
```
from splines.quaternion import KochanekBartels
```

[4]:
```
rotations = [
    angles2quat(0, 0, 0),
    angles2quat(90, 0, 0),
    angles2quat(90, 90, 0),
    angles2quat(90, 90, 90),
]
```

Uniform Catmull–Rom

[5]:
```
s = KochanekBartels(rotations)
```

[6]:
```
times = np.linspace(s.grid[0], s.grid[-1], 100)
```

[7]:
```
ani = animate_rotations(s.evaluate(times), figsize=(4, 3))
```

[8]:
```
display_animation(ani, default_mode='reflect')
```
```
Animations can only be shown in HTML output, sorry!
```

Non-Uniform Catmull–Rom

[9]:
```
grid = 0, 0.2, 0.9, 1.2
```

[10]:
```
s = KochanekBartels(rotations, grid)
```

[11]:
```
times = np.linspace(s.grid[0], s.grid[-1], 100)
```

[12]:
```
ani = animate_rotations(s.evaluate(times), figsize=(4, 3))
```

[13]:
```
display_animation(ani, default_mode='reflect')
```
```
Animations can only be shown in HTML output, sorry!
```

TCB

[14]:
```
s = KochanekBartels(rotations, tcb=[0, 1, 1])
```

[15]:
```
times = np.linspace(s.grid[0], s.grid[-1], 100)
```

[16]:
```
ani = animate_rotations(s.evaluate(times), figsize=(4, 3))
```

[17]:
```
display_animation(ani, default_mode='reflect')
```
```
Animations can only be shown in HTML output, sorry!
```

Edge Cases

- 0 or 1 quaternions: not allowed
- 2 quaternions: Slerp
- 180° rotations (dot product = 0)?
- ...

2 quaternions:

[18]:
```
rotations = [
    angles2quat(0, 0, 0),
    angles2quat(90, 90, 90),
]
```

[19]:
```
s = KochanekBartels(rotations)
```

[20]:
```
times = np.linspace(s.grid[0], s.grid[-1], 100)
```

[21]:
```
ani = animate_rotations(s.evaluate(times), figsize=(4, 3))
```

[22]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

doc/rotation/kochanek-bartels.ipynb ends here.

The following section was generated from doc/rotation/end-conditions-natural.ipynb

2.7 "Natural" End Conditions

notebook about "natural" end conditions for Euclidean splines (page 98)

[1]:
```
import numpy as np
```

helper.py

[2]:
```
from helper import angles2quat, animate_rotations, display_animation
```

splines.quaternion.DeCasteljau (page 155)

[3]:
```
from splines.quaternion import DeCasteljau
```

[4]:
```
def calculate_rotations(control_quaternions):
    times = np.linspace(0, 1, 50)
    return DeCasteljau(
        segments=[control_quaternions],
    ).evaluate(times)
```

[5]:
```
q0 = angles2quat(45, 0, 0)
q1 = angles2quat(-45, 0, 0)
```

Begin

[6]:
```
def natural_begin(begin, end_control, end):
    """Return second control quaternion given the other three."""
    return (
        (end_control * end.inverse()) *
        (end * begin.inverse())
    )**(1 / 2) * begin
```

[7]:
```
q1_control = angles2quat(-45, 0, -90)
```

[8]:
```
ani = animate_rotations({
    'natural begin': calculate_rotations(
        [q0, natural_begin(q0, q1_control, q1), q1_control, q1]),
}, figsize=(4, 3))
```

[9]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

End

[10]:
```
def natural_end(begin, begin_control, end):
    """Return third control quaternion given the other three."""
    return (
        end.inverse() *
        (
            (end * begin.inverse()) *
            (begin * begin_control.inverse())
        )**(1 / 2)).inverse()
```

[11]:
```
q0_control = angles2quat(45, 0, 90)
```

[12]:
```
ani = animate_rotations({
    'natural end': calculate_rotations(
        [q0, q0_control, natural_end(q0, q0_control, q1), q1]),
}, figsize=(4, 3))
```

[13]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

(Non-)Symmetries

Instead of using the function for the begin condition, we could of course also reverse the control quaternions, use the function for the end condition and time-reverse the result. And vice-versa.

Let's make sure that works:

[14]:
```
ani = animate_rotations({
    'natural end': calculate_rotations(
        [q0, q0_control, natural_end(q0, q0_control, q1), q1]),
    'natural begin, time-reversed': calculate_rotations(
        [q1, natural_begin(q1, q0_control, q0), q0_control, q0])[::-1],
}, figsize=(6, 3))
```

[15]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

[16]:
```
begin = natural_begin(q0, q1_control, q1)
```

[17]:
```
ani = animate_rotations({
    'natural begin': calculate_rotations(
        [q0, begin, q1_control, q1]),
    'natural end from natural begin': calculate_rotations(
        [q0, begin, natural_end(q0, begin, q1), q1]),
}, figsize=(6, 3))
```

[18]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

[19]:
```
end = natural_end(q0, q0_control, q1)
```

[20]:
```
ani = animate_rotations({
    'natural end': calculate_rotations(
        [q0, q0_control, end, q1]),
    'natural begin from natural end': calculate_rotations(
        [q0, natural_begin(q0, end, q1), end, q1]),
}, figsize=(6, 3))
```

[21]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

[ ]:
```

```

doc/rotation/end-conditions-natural.ipynb ends here.

The following section was generated from doc/rotation/barry-goldman.ipynb

2.8 Barry–Goldman Algorithm

We can try to use the *Barry–Goldman algorithm for non-uniform Euclidean Catmull–Rom splines* (page 75) using *Slerp* (page 120) instead of linear interpolations, just as we have done with *De Casteljau's algorithm* (page 125).

[1]:
```
def slerp(one, two, t):
    return (two * one.inverse())**t * one
```

[2]:
```
def barry_goldman(rotations, times, t):
    q0, q1, q2, q3 = rotations
    t0, t1, t2, t3 = times
    return slerp(
        slerp(
            slerp(q0, q1, (t - t0) / (t1 - t0)),
            slerp(q1, q2, (t - t1) / (t2 - t1)),
            (t - t0) / (t2 - t0)),
        slerp(
            slerp(q1, q2, (t - t1) / (t2 - t1)),
            slerp(q2, q3, (t - t2) / (t3 - t2)),
            (t - t1) / (t3 - t1)),
        (t - t1) / (t2 - t1))
```

Example:

[3]:
```
import numpy as np
```

helper.py

[4]:
```
from helper import angles2quat, animate_rotations, display_animation
```

[5]:
```
q0 = angles2quat(0, 0, 0)
q1 = angles2quat(90, 0, 0)
q2 = angles2quat(90, 90, 0)
q3 = angles2quat(90, 90, 90)
```

[6]:
```
t0 = 0
t1 = 1
```

(continues on next page)

(continued from previous page)

```
t2 = 3
t3 = 3.5
```

[7]:
```
frames = 50
```

[8]:
```
ani = animate_rotations({
    'Barry-Goldman (q0, q1, q2, q3)': [
        barry_goldman([q0, q1, q2, q3], [t0, t1, t2, t3], t)
        for t in np.linspace(t1, t2, frames)
    ],
    'Slerp (q1, q2)': slerp(q1, q2, np.linspace(0, 1, frames)),
}, figsize=(6, 2))
```

[9]:
```
display_animation(ani, default_mode='once')
```

```
Animations can only be shown in HTML output, sorry!
```

splines.quaternion.BarryGoldman (page 156) class

[10]:
```
from splines.quaternion import BarryGoldman
```

[11]:
```
import numpy as np
```

helper.py

[12]:
```
from helper import angles2quat, animate_rotations, display_animation
```

[13]:
```
rotations = [
    angles2quat(0, 0, 180),
    angles2quat(0, 45, 90),
    angles2quat(90, 45, 0),
    angles2quat(90, 90, -90),
    angles2quat(180, 0, -180),
    angles2quat(-90, -45, 180),
]
```

[14]:
```
grid = np.array([0, 0.5, 2, 5, 6, 7, 9])
```

[15]:
```
bg = BarryGoldman(rotations, grid)
```

For comparison . . . *Catmull–Rom-like quaternion spline* (page 131)

splines.quaternion.CatmullRom (page 156) class

[16]:
```
from splines.quaternion import CatmullRom
```

[17]:
```
cr = CatmullRom(rotations, grid, endconditions='closed')
```

[18]:
```
def evaluate(spline, samples=200):
    times = np.linspace(spline.grid[0], spline.grid[-1], samples, endpoint=False)
    return spline.evaluate(times)
```

[19]:
```
ani = animate_rotations({
    'Barry-Goldman': evaluate(bg),
    'Catmull-Rom-like': evaluate(cr),
}, figsize=(4, 2))
```

[20]:
```
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

[21]:
```
rotations = [
    angles2quat(90, 0, -45),
    angles2quat(179, 0, 0),
    angles2quat(181, 0, 0),
    angles2quat(270, 0, -45),
    angles2quat(0, 90, 90),
]
```

[22]:
```
s_uniform = BarryGoldman(rotations)
s_chordal = BarryGoldman(rotations, alpha=1)
s_centripetal = BarryGoldman(rotations, alpha=0.5)
```

[23]:
```
ani = animate_rotations({
    'uniform': evaluate(s_uniform, samples=300),
    'chordal': evaluate(s_chordal, samples=300),
    'centripetal': evaluate(s_centripetal, samples=300),
}, figsize=(6, 2))
```

[24]:
```
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

Constant Angular Speed

Not very efficient, De Casteljau's algorithm is faster because it directly provides the tangent.

[25]:
```
from splines import ConstantSpeedAdapter
```

[26]:
```
class BarryGoldmanWithDerivative(BarryGoldman):

    delta_t = 0.000001

    def evaluate(self, t, n=0):
        """Evaluate quaternion or angular velocity."""
        if not np.isscalar(t):
            return np.array([self.evaluate(t, n) for t in t])
        if n == 0:
            return super().evaluate(t)
        elif n == 1:
            # NB: We move the interval around because
            #     we cannot access times before and after
            #     the first and last time, respectively.
            fraction = (t - self.grid[0]) / (self.grid[-1] - self.grid[0])
            before = super().evaluate(t - fraction * self.delta_t)
            after = super().evaluate(t + (1 - fraction) * self.delta_t)
            # NB: Double angle
            return (after * before.inverse()).log_map() * 2 / self.delta_t
        else:
            raise ValueError('Unsupported n: {!r}'.format(n))
```

[27]:
```
s = ConstantSpeedAdapter(BarryGoldmanWithDerivative(rotations, alpha=0.5))
```

Takes a long time!

[28]:
```
ani = animate_rotations({
    'non-constant speed': evaluate(s_centripetal),
    'constant speed': evaluate(s),
}, figsize=(4, 2))
```

[29]:
```
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

doc/rotation/barry-goldman.ipynb ends here.

The following section was generated from doc/rotation/cumulative-form.ipynb

2.9 Cumulative Form

The basic idea, as proposed by [KKS95] (section 4) is the following:

Instead of representing a curve as a sum of basis functions weighted by its control point's position vectors p_i (as it's for example done with *Bézier splines* (page 38)), they suggest to use the relative difference vectors Δp_i between successive control points.

These relative difference vectors can then be "translated" to *local* rotations (replacing additions with multiplications), leading to a form of rotation splines.

Piecewise Slerp

As an example, they define a piecewise linear curve

$$p(t) = p_0 + \sum_{i=1}^{n} \alpha_i(t) \Delta p_i,$$

where

$$\Delta p_i = p_i - p_{i-1}$$

$$\alpha_i(t) = \begin{cases} 0 & t < i-1 \\ t-i+1 & i-1 \leq t < i \\ 1 & t \geq i. \end{cases}$$

[1]:
```
def alpha(i, t):
    if t < i - 1:
        return 0
    elif t >= i:
        return 1
    else:
        return t - i + 1
```

Note

There is an off-by-one error in the paper's definition of $\alpha_i(t)$:

$$\alpha_i(t) = \begin{cases} 0 & t < i \\ t-i & i \leq t < i+1 \\ 1 & t \geq i+1. \end{cases}$$

This assumes that i starts with 0, but it actually starts with 1.

This "cumulative form" can be "translated" to a rotation spline by replacing addition with multiplication and the relative difference vectors by relative (i.e. local) rotations (represented by unit quaternions):

$$q(t) = q_0 \prod_{i=1}^{n} \exp(\omega_i \alpha_i(t)),$$

where

$$\omega_i = \log\left(q_{i-1}^{-1} q_i\right).$$

The paper uses above notation, but this could equivalently be written as

$$q(t) = q_0 \prod_{i=1}^{n} \left(q_{i-1}^{-1} q_i\right)^{\alpha_i(t)}.$$

[2]:
```
import numpy as np
```

helper.py

[3]:
```
from helper import angles2quat, animate_rotations, display_animation
```

[4]:
```
from splines.quaternion import UnitQuaternion
```

[5]:
```
# NB: math.prod() since Python 3.8
product = np.multiply.reduce
```

[6]:
```
def piecewise_slerp(qs, t):
    return qs[0] * product([
        (qs[i - 1].inverse() * qs[i])**alpha(i, t)
        for i in range(1, len(qs))])
```

[7]:
```
qs = [
    angles2quat(0, 0, 0),
    angles2quat(90, 0, 0),
    angles2quat(90, 90, 0),
    angles2quat(90, 90, 90),
]
```

[8]:
```
times = np.linspace(0, len(qs) - 1, 100)
```

[9]:
```
ani = animate_rotations([piecewise_slerp(qs, t) for t in times], figsize=(4, 3))
```

[10]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

Cumulative Bézier/Bernstein Curve

After the piecewise Slerp, [KKS95] show (in section 5.1) how to create a *cumulative form* inspired by Bézier splines, i.e. using Bernstein polynomials.

They start with the well-known equation for Bézier splines:

$$p(t) = \sum_{i=0}^{n} p_i \beta_{i,n}(t),$$

where $\beta_{i,n}(t)$ are Bernstein basis functions as shown in *the notebook about Bézier splines* (page 52).

They re-formulate this into a *cumulative form*:

$$p(t) = p_0\tilde{\beta}_{0,n}(t) + \sum_{i=1}^{n} \Delta p_i \tilde{\beta}_{i,n}(t),$$

where the cumulative Bernstein basis functions are given by

$$\tilde{\beta}_{i,n}(t) = \sum_{j=i}^{n} \beta_{j,n}(t).$$

We can get the Bernstein basis polynomials via the function *splines.Bernstein.basis()* (page 150):

[11]:
```
from splines import Bernstein
```

... and create a simple helper function to sum them up:

[12]:
```
from itertools import accumulate
```

[13]:
```
def cumulative_bases(degree, t):
    return list(accumulate(Bernstein.basis(degree, t)[::-1]))[::-1]
```

Finally, they "translate" this into a rotation spline using quaternions, like before:

$$q(t) = q_0 \prod_{i=1}^{n} \exp\left(\omega_i \tilde{\beta}_{i,n}(t)\right),$$

where

$$\omega_i = \log(q_{i-1}^{-1} q_i).$$

Again, they use above notation in the paper, but this could equivalently be written as

$$q(t) = q_0 \prod_{i=1}^{n} \left(q_{i-1}^{-1} q_i\right)^{\tilde{\beta}_{i,n}(t)}.$$

[14]:
```
def cumulative_bezier(qs, t):
    degree = len(qs) - 1
    bases = cumulative_bases(degree, t)
    assert np.isclose(bases[0], 1)
    return qs[0] * product([
        (qs[i - 1].inverse() * qs[i])**bases[i]
        for i in range(1, len(qs))
    ])
```

[15]:
```
times = np.linspace(0, 1, 100)
```

[16]:
```
rotations = [cumulative_bezier(qs, t) for t in times]
```

[17]:
```
ani = animate_rotations(rotations, figsize=(4, 3))
```

[18]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

Comparison with De Casteljau's Algorithm

> This Bézier quaternion curve has a different shape from the Bézier quaternion curve of [Sho85].
>
> —[KKS95], section 5.1

The method described by [Sho85] is shown in *a separate notebook* (page 125). An implementation is available in the *splines.quaternion.DeCasteljau* (page 155) class:

[19]:
```
from splines.quaternion import DeCasteljau
```

[20]:
```
times = np.linspace(0, 1, 100)
```

[21]:
```
control_polygon = [
    angles2quat(90, 0, 0),
    angles2quat(0, -45, 90),
    angles2quat(0, 0, 0),
    angles2quat(180, 0, 180),
]
```

[22]:
```
cumulative_rotations = [
    cumulative_bezier(control_polygon, t)
    for t in times
]
```

[23]:
```
cumulative_rotations_reversed = [
    cumulative_bezier(control_polygon[::-1], t)
    for t in times
][::-1]
```

[24]:
```
casteljau_rotations = DeCasteljau([control_polygon]).evaluate(times)
```

[25]:
```
ani = animate_rotations({
    'De Casteljau': casteljau_rotations,
    'Cumulative': cumulative_rotations,
    'Cumulative reversed': cumulative_rotations_reversed,
}, figsize=(9, 3))
```

[26]:
```
display_animation(ani, default_mode='reflect')
```

```
Animations can only be shown in HTML output, sorry!
```

Applying the same method on the reversed list of control points and then time-reversing the resulting sequence of rotations leads to an equal (except for rounding errors) sequence of rotations when using De Casteljau's algorithm:

[27]:
```
casteljau_rotations_reversed = DeCasteljau([control_polygon[::-1]]).evaluate(times)[::-1]
```

[28]:
```
for one, two in zip(casteljau_rotations, casteljau_rotations_reversed):
    assert np.isclose(one.scalar, two.scalar)
    assert np.isclose(one.vector[0], two.vector[0])
    assert np.isclose(one.vector[1], two.vector[1])
    assert np.isclose(one.vector[2], two.vector[2])
```

However, doing the same thing with the "cumulative form" can lead to a significantly different sequence, as can be seen in the above animation.

doc/rotation/cumulative-form.ipynb ends here.

2.10 Naive 4D Quaternion Interpolation

This method for interpolating rotations is normally not recommended. But it might still be interesting to try it out ...

Since quaternions form a vector space (albeit a four-dimensional one), all methods for *Euclidean splines* (page 2) can be applied. However, even though rotations can be represented by *unit* quaternions, which are a subset of all quaternions, this subset is *not* a Euclidean space. All *unit* quaternions form the unit hypersphere S^3 (which is a curved space), and each point on this hypersphere uniquely corresponds to a rotation.

When we convert our desired rotation "control points" to quaternions and naively interpolate in 4D quaternion space, the interpolated quaternions are in general *not* unit quaternions, i.e. they are not part of the unit hypersphere and they don't correspond to a rotation. In order to force them onto the unit hypersphere, we can normalize them, though, which projects them onto the unit hypersphere.

Note that this is a very crude form of interpolation and it might result in unexpected curve shapes. Especially the temporal behavior might be undesired.

> If, for some application, more speed is essential, non-spherical quaternion splines will undoubtedly be faster than angle interpolation, while still free of axis bias and gimbal lock.
>
> —[Sho85], section 5.4

> Abandoning the unit sphere, one could work with the four-dimensional Euclidean space of arbitrary quaternions. How do standard interpolation methods applied there behave when mapped back to matrices? Note that we now have little guidance in picking the inverse image for a matrix, and that cusp-free $\mathbf{R}^4$ paths do not always project to cusp-free S^3 paths.
>
> —[Sho85], section 6

[1]:
```
import numpy as np
```

[2]:
```
import splines
```

[3]:
```
from splines.quaternion import Quaternion
```

As always, we use a few helper functions from helper.py:

[4]:
```
from helper import angles2quat, animate_rotations, display_animation
```

[5]:
```
rotations = [
    angles2quat(0, 0, 0),
    angles2quat(0, 0, 45),
    angles2quat(90, 90, 0),
    angles2quat(180, 0, 90),
]
```

We use `xyzw` coordinate order here (because it is more common), but since the 4D coordinates are independent, we could as well use `wxyz` order (or any order, for that matter) with identical results (apart from rounding errors).

However, for illustrating the non-normalized case, we rely on the implicit conversion from `xyzw` coordinates in the function `animate_rotations()`.

[6]:
```
rotations_xyzw = [q.xyzw for q in rotations]
```

As an example we use *splines.CatmullRom* (page 150) here, but any Euclidean spline could be used.

[7]:
```
s = splines.CatmullRom(rotations_xyzw, endconditions='closed')
```

[8]:
```
times = np.linspace(s.grid[0], s.grid[-1], 100)
```

[9]:
```
interpolated_xyzw = s.evaluate(times)
```

[10]:
```
normalized = [
    Quaternion(w, (x, y, z)).normalize()
    for x, y, z, w in interpolated_xyzw]
```

For comparison, we also create a *splines.quaternion.CatmullRom* (page 156) instance:

[11]:
```
spherical_cr = splines.quaternion.CatmullRom(rotations, endconditions='closed')
```

[12]:
```
ani = animate_rotations({
    'normalized 4D interpolation': normalized,
    'spherical interpolation': spherical_cr.evaluate(times),
}, figsize=(6, 3))
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

In case you are wondering what would happen if you forget to normalize the results, let's also show the non-normalized data:

[13]:
```
ani = animate_rotations({
    'normalized': normalized,
    'not normalized': interpolated_xyzw,
}, figsize=(6, 3))
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

Obviously, the non-normalized values are not pure rotations.

To get a different temporal behavior, let's try using *centripetal parameterization* (page 59).

Note that this guarantees the absence of cusps and self-intersections in the 4D curve, but this guarantee doesn't extend to the projection onto the unit hypersphere.

[14]:
```
s2 = splines.CatmullRom(rotations_xyzw, alpha=0.5, endconditions='closed')
```

[15]:
```
times2 = np.linspace(s2.grid[0], s2.grid[-1], len(times))
```

[16]:
```
normalized2 = [
    Quaternion(w, (x, y, z)).normalize()
    for x, y, z, w in s2.evaluate(times2)]
```

[17]:
```
ani = animate_rotations({
    'uniform': normalized,
    'centripetal': normalized2,
}, figsize=(6, 3))
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

Let's also try *arc-length parameterization* with the *ConstantSpeedAdapter* (page 152):

[18]:
```
s3 = splines.ConstantSpeedAdapter(s2)
times3 = np.linspace(s3.grid[0], s3.grid[-1], len(times))
```

[19]:
```
normalized3 = [
    Quaternion(w, (x, y, z)).normalize()
    for x, y, z, w in s3.evaluate(times3)]
```

The arc-length parameterized spline has a constant speed in 4D quaternion space, but that doesn't mean it has a constant angular speed!

For comparison, we also create a rotation spline with constant angular speed:

[20]:
```
s4 = splines.ConstantSpeedAdapter(
    splines.quaternion.CatmullRom(
        rotations, alpha=0.5, endconditions='closed'))
times4 = np.linspace(s4.grid[0], s4.grid[-1], len(times))
```

[21]:
```
ani = animate_rotations({
    'constant 4D speed': normalized3,
    'constant angular speed': s4.evaluate(times4),
}, figsize=(6, 3))
display_animation(ani, default_mode='loop')
```

```
Animations can only be shown in HTML output, sorry!
```

The difference is subtle, but it is definitely visible. More extreme examples can certainly be found.

doc/rotation/naive-4d-interpolation.ipynb ends here.

The following section was generated from doc/rotation/naive-euler-angles-interpolation.ipynb

2.11 Naive Interpolation of Euler Angles

This method for interpolating 3D rotations is very much not recommended!

Since 3D rotations can be represented by a list of three angles, it might be tempting to simply interpolate those angles independently.

Let's try it and see what happens, shall we?

[1]:
```
import numpy as np
```

[2]:
```
import splines
```

As always, we use a few helper functions from helper.py:

[3]:
```
from helper import angles2quat, animate_rotations, display_animation
```

We are using *splines.CatmullRom* (page 150) to interpolate the Euler angles independently and *splines.quaternion.CatmullRom* (page 156) to interpolate the associated quaternions for comparison:

[4]:
```
def plot_interpolated_angles(angles):
    s1 = splines.CatmullRom(angles, endconditions='closed')
    times = np.linspace(s1.grid[0], s1.grid[-1], 100)
    s2 = splines.quaternion.CatmullRom(
        [angles2quat(azi, ele, roll) for azi, ele, roll in angles],
        endconditions='closed')
    ani = animate_rotations({
        'Euler angles': [angles2quat(*abc) for abc in s1.evaluate(times)],
        'quaternions': s2.evaluate(times),
```

(continues on next page)

(continued from previous page)

```
    }, figsize=(6, 3))
    display_animation(ani, default_mode='loop')
```

[5]:
```
plot_interpolated_angles([
    (0, 0, 0),
    (45, 0, 0),
    (90, 45, 0),
    (90, 90, 0),
    (180, 0, 90),
])
```

```
Animations can only be shown in HTML output, sorry!
```

There is clearly a difference between the two, but the Euler angles don't look that bad.

Let's try another example:

[6]:
```
plot_interpolated_angles([
    (-175, 0, 0),
    (175, 0, 0),
])
```

```
Animations can only be shown in HTML output, sorry!
```

Here we see that the naive interpolation isn't aware that the azimuth angle is supposed to wrap around at 180 degrees.

This could be fixed with a less naive implementation, but there are also unfixable problems, as this example shows:

[7]:
```
plot_interpolated_angles([
    (45, 45, 0),
    (45, 90, 0),
    (-135, 45, 180),
])
```

```
Animations can only be shown in HTML output, sorry!
```

Even though all involved rotations are supposed to happen around a single rotation axis, The Euler angles interpolation is all over the place.

doc/rotation/naive-euler-angles-interpolation.ipynb ends here.

3 Python Module

splines (page 149)	Piecewise polynomial curves (in Euclidean space).
splines.quaternion (page 153)	Quaternions and unit-quaternion splines.

3.1 splines

Piecewise polynomial curves (in Euclidean space).

Submodules

quaternion (page 153)	Quaternions and unit-quaternion splines.

Classes

Bernstein (page 150)(segments[, grid])	Piecewise Bézier curve using Bernstein basis.
CatmullRom (page 150)(vertices[, grid, alpha, ...])	Catmull–Rom spline.
ConstantSpeedAdapter (page 152)(curve)	Re-parameterize a spline to have constant speed.
CubicHermite (page 150)(vertices, tangents[, grid])	Cubic Hermite curve.
KochanekBartels (page 151)(vertices[, grid, tcb, ...])	Kochanek–Bartels spline.
Monomial (page 149)(segments, grid)	Piecewise polynomial curve using monomial basis.
MonotoneCubic (page 152)(values, *args, **kwargs)	Monotone cubic curve.
Natural (page 151)(vertices[, grid, alpha, end-conditions])	Natural spline.
NewGridAdapter (page 152)(curve[, new_grid])	Re-parameterize a spline with new grid values.
PiecewiseMonotoneCubic (page 151)(values[, grid, ...])	Piecewise monotone cubic curve.

class splines.**Monomial**(*segments*, *grid*)

Bases: object[28]

Piecewise polynomial curve using monomial basis.

Arbitrary degree, arbitrary dimension.

$$\boldsymbol{p}_i(t) = \sum_{k=0}^{n} \boldsymbol{a}_k \left(\frac{t - t_i}{t_{i+1} - t_i} \right)^k \quad \text{for } t_i \le t < t_{i+1}$$

Similar to https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PPoly.html, which states:

> "High-order polynomials in the power basis can be numerically unstable. Precision problems can start to appear for orders larger than 20-30."

Parameters

- **segments** – Sequence of polynomial segments. Each segment contains coefficients for the monomial basis (in order of decreasing degree). Different segments can have different polynomial degree.
- **grid** – Sequence of parameter values corresponding to segment boundaries. Must be strictly increasing.

evaluate(*t*, *n=0*)

Get value (or *n*-th derivative) at given parameter value(s).

class splines.**Bernstein**(*segments*, *grid=None*)

Bases: object[29]

Piecewise Bézier curve using Bernstein basis.

Parameters

- **segments** – Sequence of segments, each one consisting of multiple Bézier control points. Different segments can have different numbers of control points (and therefore different polynomial degrees).
- **grid** (*optional*) – Sequence of parameter values corresponding to segment boundaries. Must be strictly increasing. If not specified, a uniform grid is used (0, 1, 2, 3, ...).

static basis(*degree*, *t*)

Bernstein basis polynomials of given *degree*, evaluated at *t*.

Returns a list of values corresponding to $i = 0, \ldots, n$, given the degree n, using the formula

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i},$$

with the *binomial coefficient* $\binom{n}{i} = \frac{n!}{i!(n-i)!}$.

evaluate(*t*, *n=0*)

Get value at the given parameter value(s).

class splines.**CubicHermite**(*vertices*, *tangents*, *grid=None*)

Bases: *splines.Monomial* (page 149)

Cubic Hermite curve.

See *Hermite Splines* (page 12).

Parameters

- **vertices** – Sequence of vertices.
- **tangents** – Sequence of tangent vectors (two per segment, outgoing and incoming).
- **grid** (*optional*) – Sequence of parameter values. Must be strictly increasing. If not specified, a uniform grid is used (0, 1, 2, 3, ...).

```
matrix = array([[ 2, -2, 1, 1], [-3, 3, -2, -1], [ 0, 0, 1, 0], [ 1, 0, 0, 0]])
```

class splines.**CatmullRom**(*vertices*, *grid=None*, ***, *alpha=None*, *endconditions='natural'*)

Bases: *splines.CubicHermite* (page 150)

Catmull–Rom spline.

This class implements one specific member of the family of splines described in [CR74], which is commonly known as *Catmull–Rom spline*: The cubic spline that can be constructed by linear Lagrange interpolation (and extrapolation) followed by quadratic B-spline blending, or equivalently, quadratic Lagrange interpolation followed by linear B-spline blending.

The implementation used in this class, however, does nothing of that sort. It simply calculates the appropriate tangent vectors at the control points and instantiates a *CubicHermite* (page 150) spline.

See *Catmull--Rom Splines* (page 52).

Parameters

[28] https://docs.python.org/3/library/functions.html#object
[29] https://docs.python.org/3/library/functions.html#object

- **vertices** – Sequence of vertices.
- **grid** (*optional*) – Sequence of parameter values. Must be strictly increasing. If not specified, a uniform grid is used (0, 1, 2, 3, ...).
- **alpha** (*optional*) – TODO
- **endconditions** (*optional*) – Start/end conditions. Can be `'closed'`, `'natural'` or pair of tangent vectors (a.k.a. "clamped"). If `'closed'`, the first vertex is re-used as last vertex and an additional *grid* time has to be specified.

class splines.**KochanekBartels**(*vertices*, *grid=None*, ***, *tcb=(0, 0, 0)*, *alpha=None*, *endconditions='natural'*)

Bases: *splines.CubicHermite* (page 150)

Kochanek–Bartels spline.

See *Kochanek--Bartels Splines* (page 84).

Parameters

- **vertices** – Sequence of vertices.
- **grid** (*optional*) – Sequence of parameter values. Must be strictly increasing. If not specified, a uniform grid is used (0, 1, 2, 3, ...).
- **tcb** (*optional*) – Sequence of *tension*, *continuity* and *bias* triples. TCB values can only be given for the interior vertices.
- **alpha** (*optional*) – TODO
- **endconditions** (*optional*) – Start/end conditions. Can be `'closed'`, `'natural'` or pair of tangent vectors (a.k.a. "clamped"). If `'closed'`, the first vertex is re-used as last vertex and an additional *grid* time has to be specified.

class splines.**Natural**(*vertices*, *grid=None*, ***, *alpha=None*, *endconditions='natural'*)

Bases: *splines.CubicHermite* (page 150)

Natural spline.

See *Natural Splines* (page 30).

Parameters

- **vertices** – Sequence of vertices.
- **grid** (*optional*) – Sequence of parameter values. Must be strictly increasing. If not specified, a uniform grid is used (0, 1, 2, 3, ...).
- **alpha** (*optional*) – TODO
- **endconditions** (*optional*) – Start/end conditions. Can be `'closed'`, `'natural'` or pair of tangent vectors (a.k.a. "clamped"). If `'closed'`, the first vertex is re-used as last vertex and an additional *grid* time has to be specified.

class splines.**PiecewiseMonotoneCubic**(*values*, *grid=None*, *slopes=None*, ***, *alpha=None*, *closed=False*)

Bases: *splines.CatmullRom* (page 150)

Piecewise monotone cubic curve.

See *Piecewise Monotone Interpolation* (page 103).

This only works for one-dimensional values.

For undefined slopes, `_calculate_tangent()` is called on the base class.

Parameters

- **values** – Sequence of values to be interpolated.

- **grid** (*optional*) – Sequence of parameter values. Must be strictly increasing. If not specified, a uniform grid is used (0, 1, 2, 3, ...).
- **slopes** (*optional*) – Sequence of slopes or `None` if slope should be computed from neighboring values. An error is raised if a segment would become non-monotone with a given slope.

class splines.**MonotoneCubic**(*values*, **args*, ***kwargs*)

Bases: *splines.PiecewiseMonotoneCubic* (page 151)

Monotone cubic curve.

This takes the same arguments as *PiecewiseMonotoneCubic* (page 151) (except `closed`), but it raises an error if the given values are not montone.

See *Monotone Interpolation* (page 111).

get_time(*value*)

Get the time instance for the given value.

If the solution is not unique (i.e. there is a plateau), `None` is returned.

class splines.**ConstantSpeedAdapter**(*curve*)

Bases: `object`[30]

Re-parameterize a spline to have constant speed.

For splines in Euclidean space this amounts to arc-length parameterization.

However, this class is implemented in a way that also allows using rotation splines which will be re-parameterized to have constant angular speed.

The parameter s represents the cumulative arc-length or the cumulative rotation angle, respectively.

evaluate(*s*)

class splines.**NewGridAdapter**(*curve*, *new_grid=1*)

Bases: `object`[31]

Re-parameterize a spline with new grid values.

Parameters

- **curve** – A spline.
- **new_grid** (*optional*) – If a single number is given, the new parameter will range from 0 to that number. Otherwise, a sequence of numbers has to be given, one for each grid value. Instead of a value, `None` can be specified to choose a value automatically. The first and last value cannot be `None`.

evaluate(*u*)

[30] https://docs.python.org/3/library/functions.html#object
[31] https://docs.python.org/3/library/functions.html#object

3.2 splines.quaternion

Quaternions and unit-quaternion splines.

Functions

canonicalized (page 154)(quaternions)	Iterator adapter to ensure minimal angles between quaternions.
slerp (page 154)(one, two, t)	Spherical linear interpolation.

Classes

BarryGoldman (page 156)(quaternions[, grid, alpha])	Rotation spline using Barry–Goldman algorithm.
CatmullRom (page 156)(quaternions[, grid, alpha, ...])	Catmull–Rom-like rotation spline.
DeCasteljau (page 155)(segments[, grid])	Spline using De Casteljau's algorithm with *slerp()* (page 154).
KochanekBartels (page 155)(quaternions[, grid, tcb, ...])	Kochanek–Bartels-like rotation spline.
PiecewiseSlerp (page 154)(quaternions, *[, grid, closed])	Piecewise Slerp.
Quaternion (page 153)(scalar, vector)	A very simple quaternion class.
UnitQuaternion (page 153)()	Unit quaternion.

class splines.quaternion.**Quaternion**(*scalar*, *vector*)

Bases: object[32]

A very simple quaternion class.

This is the base class for the more relevant *UnitQuaternion* (page 153) class.

property scalar

property vector

conjugate()

normalize()

dot(*other*)

Dot product of two quaternions.

This is the 4-dimensional dot product, yielding a scalar result. This operation is commutative.

Note that this is different from the quaternion multiplication (`q1 * q2`), which produces another quaternion (and is non-commutative).

property norm

property xyzw

property wxyz

class splines.quaternion.**UnitQuaternion**

Bases: *splines.quaternion.Quaternion* (page 153)

[32] https://docs.python.org/3/library/functions.html#object

Unit quaternion.

classmethod from_axis_angle(*axis*, *angle*)
Create a unit quaternion from a rotation axis and angle.

Parameters

- **axis** – Three-component rotation axis. This will be normalized.
- **angle** – Rotation angle in radians.

classmethod from_unit_xyzw(*xyzw*)
Create a unit quaternion from another unit quaternion.

Parameters **xyzw** – Components of a unit quaternion (scalar last). This will *not* be normalized, it must already have unit length.

inverse()
Multiplicative inverse.

For unit quaternions, this is the same as *conjugate()* (page 153).

classmethod exp_map(*value*)
Exponential map from R^3 to quaternions.

This is the inverse operation to *log_map()* (page 154).

Parameters

- **value** – Element of the tangent space at the quaternion identity.
- **type** – 3-tuple

Returns Corresponding unit quaternion.

log_map()
Logarithmic map from quaternions to R^3.

Returns Corresponding vector in the tangent space at the quaternion identity.

property axis

property angle

rotation_to(*other*)
Rotation required to rotate *self* into *other*.

See *Relative Rotation (Global Frame of Reference)* (page 119).

rotate_vector(*v*)

splines.quaternion.**slerp**(*one*, *two*, *t*)
Spherical linear interpolation.

See *Spherical Linear Interpolation (Slerp)* (page 120).

Parameters

- **one** – Start quaternion.
- **two** – End quaternion.
- **t** – Parameter value(s) between 0 and 1.

splines.quaternion.**canonicalized**(*quaternions*)
Iterator adapter to ensure minimal angles between quaternions.

class splines.quaternion.**PiecewiseSlerp**(*quaternions*, *, *grid=None*, *closed=False*)
Bases: object[33]

Piecewise Slerp.

See *Piecewise Slerp* (page 122).

Parameters

- **quaternions** – Sequence of rotations to be interpolated. The quaternions will be `canonicalized()` (page 154).
- **grid** (*optional*) – Sequence of parameter values. Must be strictly increasing. Must have the same length as *quaternions*, except when *closed* is `True`, where it must be one element longer. If not specified, a uniform grid is used (0, 1, 2, 3, ...).
- **closed** (*optional*) – If `True`, the first quaternion is repeated at the end.

evaluate(*t*, *n=0*)

class splines.quaternion.**DeCasteljau**(*segments*, *grid=None*)

Bases: `object`[34]

Spline using De Casteljau's algorithm with `slerp()` (page 154).

See *the corresponding notebook* (page 125) for details.

Parameters

- **segments** – Sequence of segments, each one consisting of multiple control quaternions. Different segments can have different numbers of control points.
- **grid** (*optional*) – Sequence of parameter values corresponding to segment boundaries. Must be strictly increasing. If not specified, a uniform grid is used (0, 1, 2, 3, ...).

evaluate(*t*, *n=0*)

Get value or angular velocity at given parameter value(s).

Parameters

- **t** – Parameter value(s).
- **n** (*{0, 1}, optional*) – Use `0` for calculating the value (a quaternion), `1` for the angular velocity (a 3-element vector).

class splines.quaternion.**KochanekBartels**(*quaternions*, *grid=None*, ***, *tcb=(0, 0, 0)*, *alpha=None*, *endconditions='natural'*)

Bases: `splines.quaternion.DeCasteljau` (page 155)

Kochanek–Bartels-like rotation spline.

Parameters

- **quaternions** – Sequence of rotations to be interpolated. The quaternions will be `canonicalized()` (page 154).
- **grid** (*optional*) – Sequence of parameter values. Must be strictly increasing. If not specified, a uniform grid is used (0, 1, 2, 3, ...).
- **tcb** (*optional*) – Sequence of *tension*, *continuity* and *bias* triples. TCB values can only be given for the interior quaternions. If only two quaternions are given, TCB values are ignored.
- **alpha** (*optional*) – TODO
- **endconditions** (*optional*) – Start/end conditions. Can be `'closed'`, `'natural'` or pair of tangent vectors (a.k.a. "clamped").

  TODO: clamped

[33] https://docs.python.org/3/library/functions.html#object
[34] https://docs.python.org/3/library/functions.html#object

If `'closed'`, the first rotation is re-used as last rotation and an additional *grid* time has to be specified.

class splines.quaternion.**CatmullRom**(*quaternions*, *grid=None*, ***, *alpha=None*, *endconditions='natural'*)

Bases: *splines.quaternion.KochanekBartels* (page 155)

Catmull–Rom-like rotation spline.

This is just *KochanekBartels* (page 155) without TCB values.

class splines.quaternion.**BarryGoldman**(*quaternions*, *grid=None*, ***, *alpha=None*)

Bases: `object`[35]

Rotation spline using Barry–Goldman algorithm.

Always closed (for now).

evaluate(*t*)

4 External Resources

A Primer on Bézier Curves: https://pomax.github.io/bezierinfo/

5 References

References

[BG88] Phillip J. Barry and Ronald N. Goldman. A recursive evaluation algorithm for a class of Catmull–Rom splines. In *15th Annual Conference on Computer Graphics and Interactive Techniques*, 199–204. 1988. doi:10.1145/54852.378511[36].

[CR74] Edwin Catmull and Raphael Rom. A class of local interpolating splines. In Robert E. Barnhill and Richard F. Riesenfeld, editors, *Computer Aided Geometric Design*, pages 317–326. Academic Press, 1974. doi:10.1016/B978-0-12-079050-0.50020-5[37].

[dB72] Carl de Boor. On calculating with B-splines. *Journal of Approximation Theory*, 6(1):50–62, 1972. doi:10.1016/0021-9045(72)90080-9[38].

[dB78] Carl de Boor. *A Practical Guide to Splines*. Springer-Verlag, 1978. ISBN 978-0-387-95366-3.

[DEH89] Randall L. Dougherty, Alan S. Edelman, and James M. Hyman. Nonnegativity-, monotonicity-, or convexity-preserving cubic and quintic Hermite interpolation. *Mathematics of Computation*, 52(186):471–494, 1989. doi:10.1090/S0025-5718-1989-0962209-1[39].

[Fri82] Frederick N. Fritsch. Piecewise cubic Hermite interpolation package (final specifications). Technical Report UCID-30194, Lawrence Livermore National Laboratory, CA (USA), 1982. doi:10.2172/6838406[40].

[35] https://docs.python.org/3/library/functions.html#object
[36] https://doi.org/10.1145/54852.378511
[37] https://doi.org/10.1016/B978-0-12-079050-0.50020-5
[38] https://doi.org/10.1016/0021-9045(72)90080-9
[39] https://doi.org/10.1090/S0025-5718-1989-0962209-1
[40] https://doi.org/10.2172/6838406

[FB84] Frederick N. Fritsch and Judy Butland. A method for constructing local monotone piecewise cubic interpolants. *SIAM Journal on Scientific and Statistical Computing*, 5(2):300–304, 1984. doi:10.1137/0905021[41].

[FC80] Frederick N. Fritsch and Ralph E. Carlson. Monotone piecewise cubic interpolation. *SIAM Journal on Numerical Analysis*, 17(2):238–246, 1980. doi:10.1137/0717021[42].

[GR74] William J. Gordon and Richard F. Riesenfeld. B-spline curves and surfaces. In *Computer Aided Geometric Design*, pages 95–126. Academic Press, 1974. doi:10.1016/B978-0-12-079050-0.50011-4[43].

[KKS95] Myoung-Jun Kim, Myung-Soo Kim, and Sung Yong Shin. A general construction scheme for unit quaternion curves with simple high order derivatives. In *SIGGRAPH: Computer graphics and interactive techniques*, 369–376. 1995. doi:10.1145/218380.218486[44].

[KB84] Doris H. U. Kochanek and Richard H. Bartels. Interpolating splines with local tension, continuity, and bias control. In *11th Annual Conference on Computer Graphics and Interactive Techniques*, 33–41. 1984. doi:10.1145/800031.808575[45].

[Mil] Ian Millington. Matrices and conversions for uniform parametric curves. URL: https://web.archive.org/web/20160305083440/http://therndguy.com.

[Mol04] Cleve B. Moler. *Numerical Computing with MATLAB*. Society for Industrial and Applied Mathematics, 2004. ISBN 978-0-89871-660-3. URL: https://www.mathworks.com/moler/index_ncm.html.

[Ove68] Albert W. Overhauser. Analytic definition of curves and surfaces by parabolic blending. Technical Report SL 68-40, Scientific Laboratory, Ford Motor Company, Dearborn, Michigan, 1968.

[Sho85] Ken Shoemake. Animating rotation with quaternion curves. *SIGGRAPH Computer Graphics*, 19(3):245–254, 1985. doi:10.1145/325165.325242[46].

[YSK11] Cem Yuksel, Scott Schaefer, and John Keyser. Parameterization and applications of Catmull–Rom curves. *Computer-Aided Design*, 43(7):747–755, 2011. doi:10.1016/j.cad.2010.08.008[47].

[41] https://doi.org/10.1137/0905021
[42] https://doi.org/10.1137/0717021
[43] https://doi.org/10.1016/B978-0-12-079050-0.50011-4
[44] https://doi.org/10.1145/218380.218486
[45] https://doi.org/10.1145/800031.808575
[46] https://doi.org/10.1145/325165.325242
[47] https://doi.org/10.1016/j.cad.2010.08.008