Splines in Euclidean Space and Beyond

Release 0.2.0

Matthias Geier

2022-03-04

Contents

. . . 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

The building blocks for *polynomial splines* (page 2) are of course polynomials[1].

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

We are mostly interested in *univariate* splines, i.e. curves with one free parameter, which are built using polynomials with a single parameter.

Here we are calling this parameter t. You can think about it as time (e.g. in seconds), but it doesn't have to represent time.

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

Polynomials typically consist of multiple *terms*. Each term contains a *basis function*, which itself contains one or more integer powers of t. The highest power of all terms is called the *degree* of the polynomial.

The arguably simplest set of basis functions is the *monomial basis*, which simply consists of all powers of t up to the given degree:

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

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

In this example we are using polynomials of degree 3, which are also called *cubic* polynomials.

These basis functions are multiplied by (constant) *coefficients*. We are writing the coefficients with bold symbols, because apart from simple scalars (for one-dimensional functions), these symbols can also represent vectors in two- or three-dimensional space.

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

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

[4]:
$$\left[\begin{matrix}d\\c\\b\\a\end{matrix}\right]$$

We can create a polynomial by multiplying the basis functions with the coefficients and then adding all terms:

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

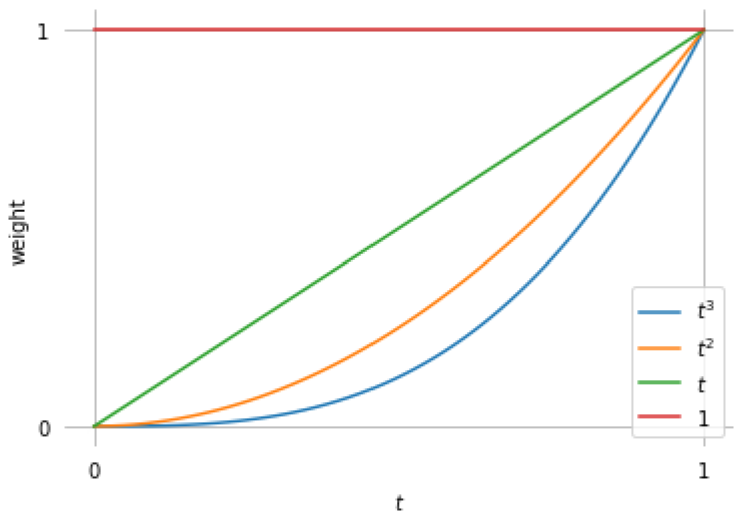
[5]: $dt^3 + ct^2 + bt + a$

This is a cubic polynomial in its *canonical form* (because it uses monomial basis functions).

Let's take a closer look at those basis functions (with some help from helper.py):

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

[7]:
```
plot_basis(*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 (i.e. using one specific list of coefficients).

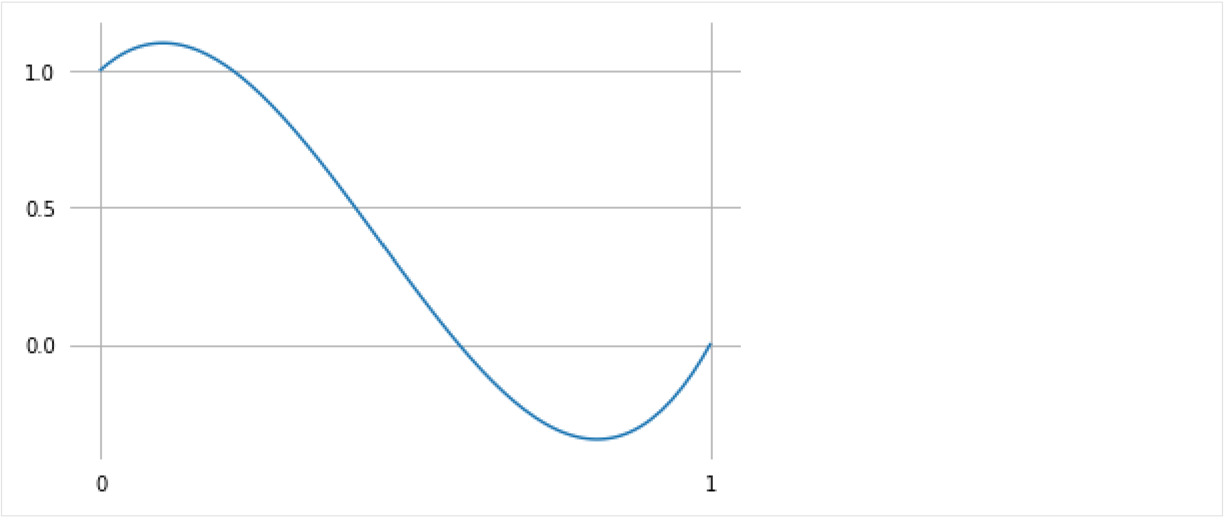
An example polynomial that's not in canonical form …

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

[10]:
```
plot_sympy(example_polynomial, (t, 0, 1))
grid_lines([0, 1], [0, 0.5, 1])
```



... can simply be re-written with monomial basis functions:

[11]:
```
example_polynomial.expand()
```

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

Any polynomial can be rewritten using any set of basis functions (of appropriate degree).

In later sections we will see more basis functions, for example those that are used for *Hermite* (page 16), *Bézier* (page 40) and *Catmull–Rom* (page 62) splines.

In the previous example, we have used scalar coefficients to create a one-dimensional polynomial. We can use two-dimensional coefficients to create two-dimensional polynomial curves. Let's create a little class to try this:

[12]:
```
import numpy as np

class CubicPolynomial:

    grid = 0, 1

    def __init__(self, d, c, b, a):
        self.coeffs = d, c, b, a

    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[2].

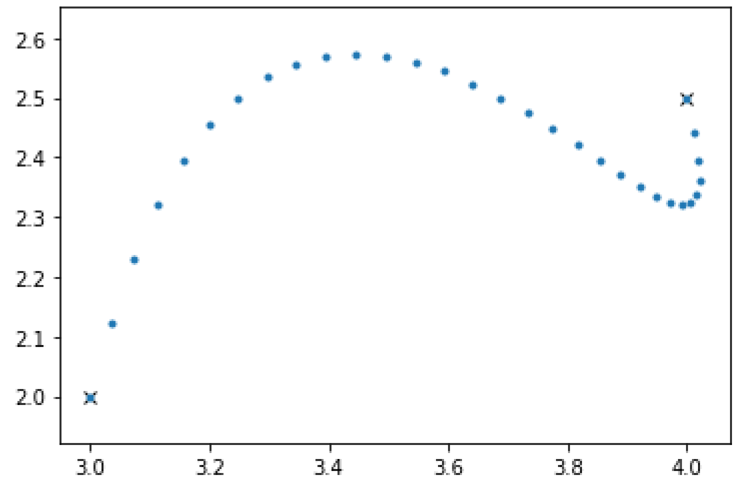
Since this class has the same interface as the splines that will be discussed in later sections, we can use a spline helper function from helper.py for plotting:

[13]:
```
from helper import plot_spline_2d
```

[14]:
```
poly_2d = CubicPolynomial([-1.5, 5], [1.5, -8.5], [1, 4], [3, 2])
```

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

[15]:
```
plot_spline_2d(poly_2d, dots_per_second=30, chords=False)
```



This class can also be used with three and more dimensions. The class *splines.Monomial* (page 159) can be used to try this with arbitrary polynomial degree.

`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[3].

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
```

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 $\ell_j(t)$ for each of the N given time instants, which will be weighted by the associated x_j. The final interpolation function is the weighted sum of these N polynomials:

$$L(t) = \sum_{j=0}^{N-1} x_j \ell_j(t)$$

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

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). Mathematically, this could be written as

$$\ell_j(t) \overset{?}{=} \prod_{\substack{i=0\\i\neq j}}^{N-1} t - t_i.$$

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!

Generally, Lagrange basis polynomials can be written as

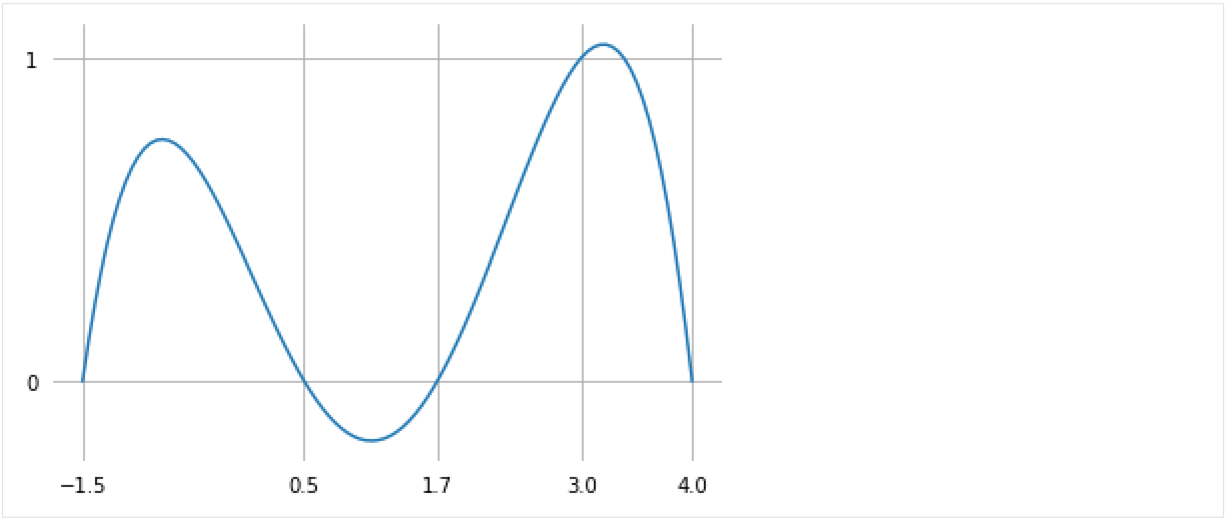
$$\ell_j(t) = \prod_{\substack{i=0\\i\neq j}}^{N-1} \frac{t - t_i}{t_j - t_i}.$$

To get a better idea what's going on between the given time instances t_i, 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.

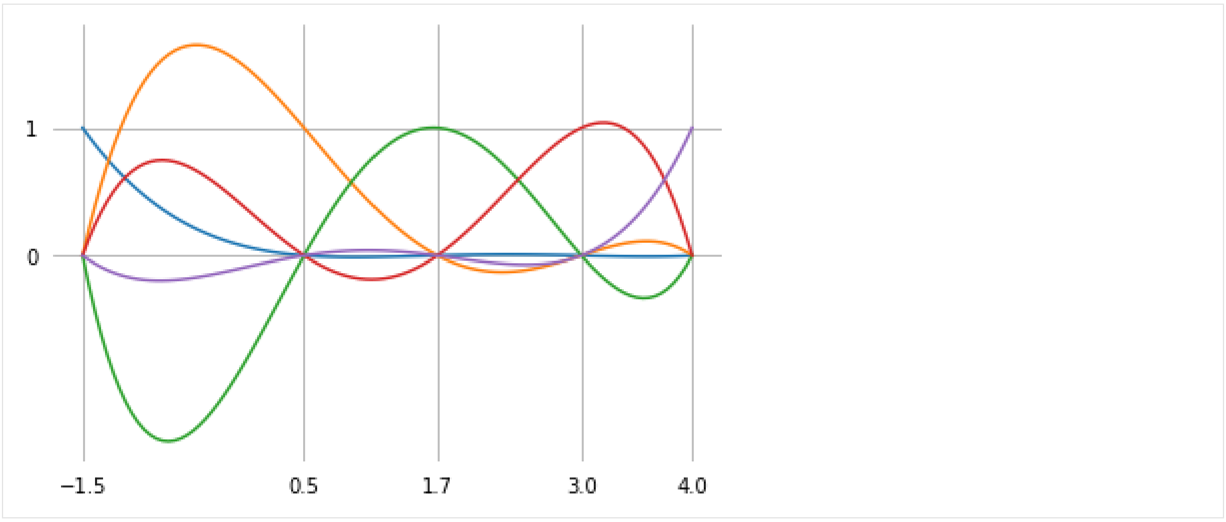
The above calculation can be easily generalized to be able to get any one of the set of polynomials defined by an arbitrary list of time instants:

```
[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])
```

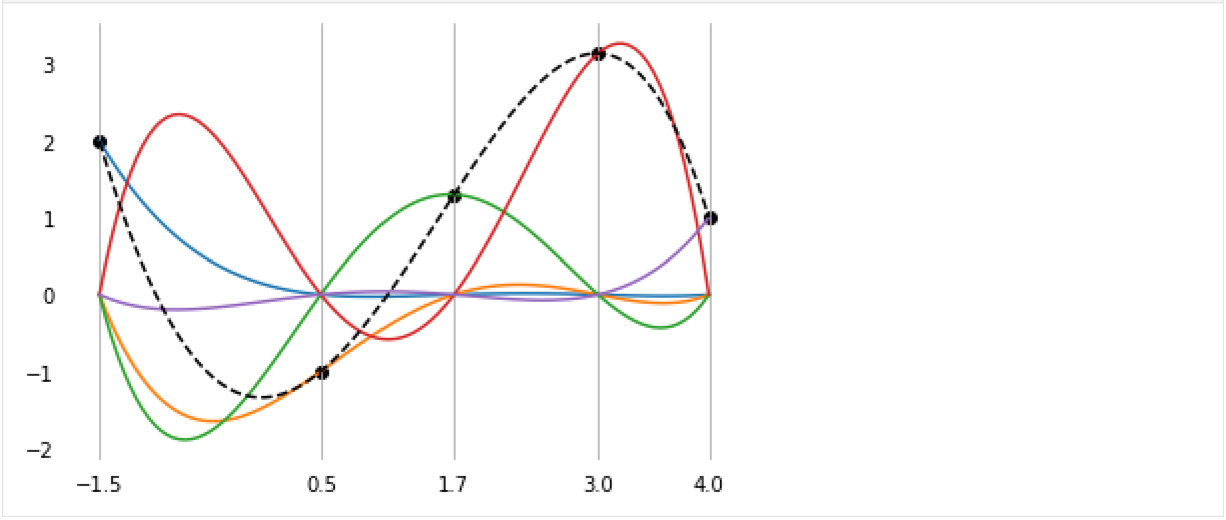


Finally, the interpolated values can be obtained by applying the given x_i values as weights to the polynomials and summing everything together:

```
[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[4] (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:

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

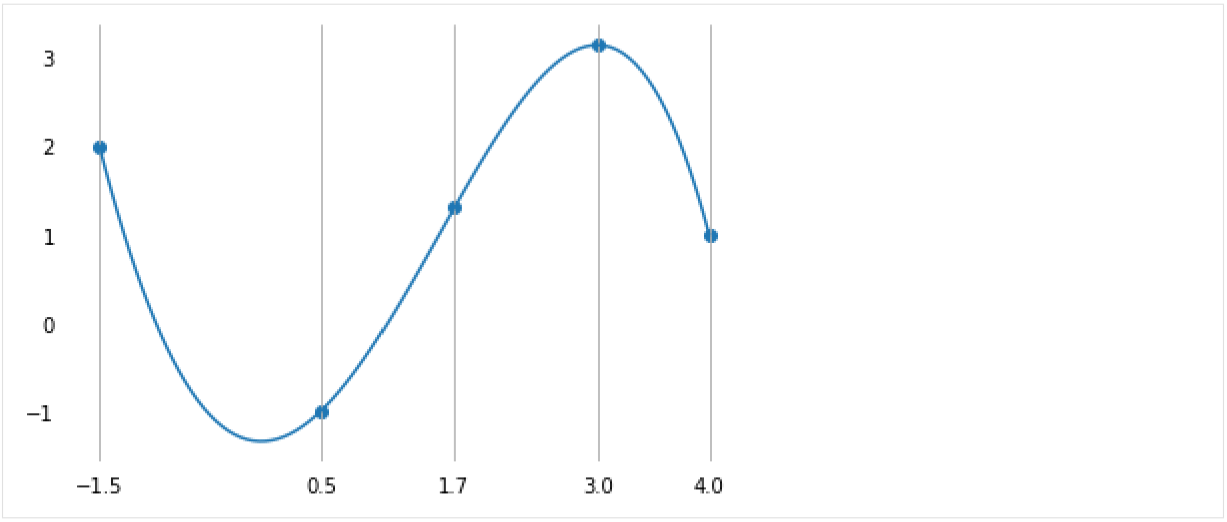
[17]:
```python
def lerp(xs, ts, t):
    """Linear intERPolation.

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

    The two x-values can be scalars or vectors,
    or even higher-dimensional arrays
    (as long as the shape of *t* is compatible).

    """
    x_begin, x_end = map(np.asarray, xs)
    t_begin, t_end = ts
    if not np.isscalar(t):
        # This allows using an array of *t* values:
        t = np.expand_dims(t, axis=-1)
    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]:
```python
def neville(xs, ts, t):
    """Lagrange interpolation using Neville's algorithm.

    Returns the interpolated value(s) at time(s) *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)
            for args in zip(zip(xs, xs[1:]), zip(ts, ts[step:]))]
    return xs[0]
```

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



Two-dimensional Example

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

[20]:
```
class Lagrange:

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

    def evaluate(self, t):
        return neville(self.vertices, self.grid, t)
```

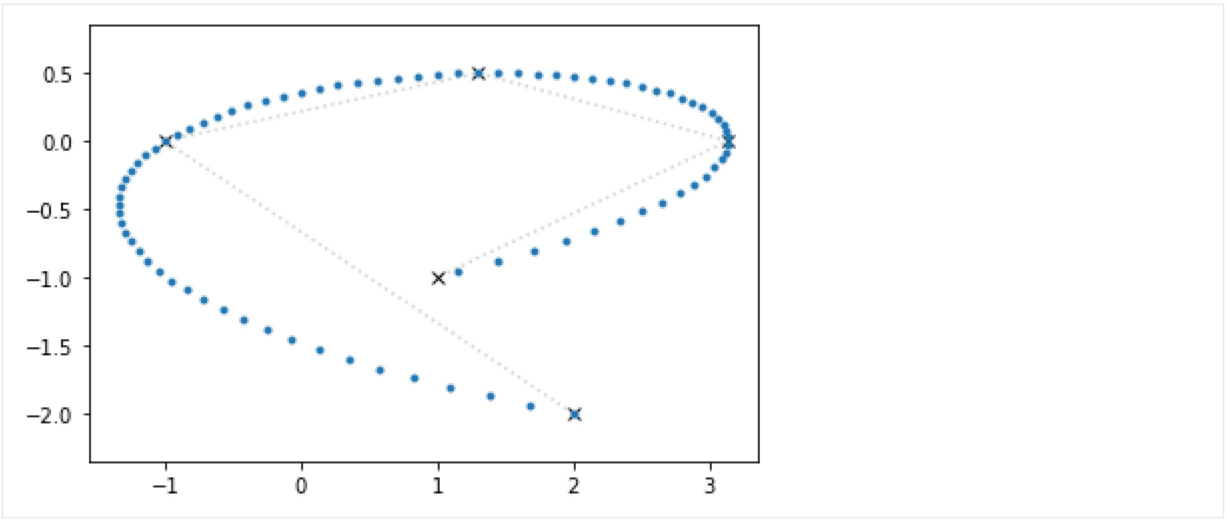
Since this class has the same interface as the splines that will be discussed in the following sections, we can use a spline helper function from helper.py for plotting:

[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, -2), (-1, 0), (1.3, 0.5), (3.14, 0), (1, -1)], ts)
```

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

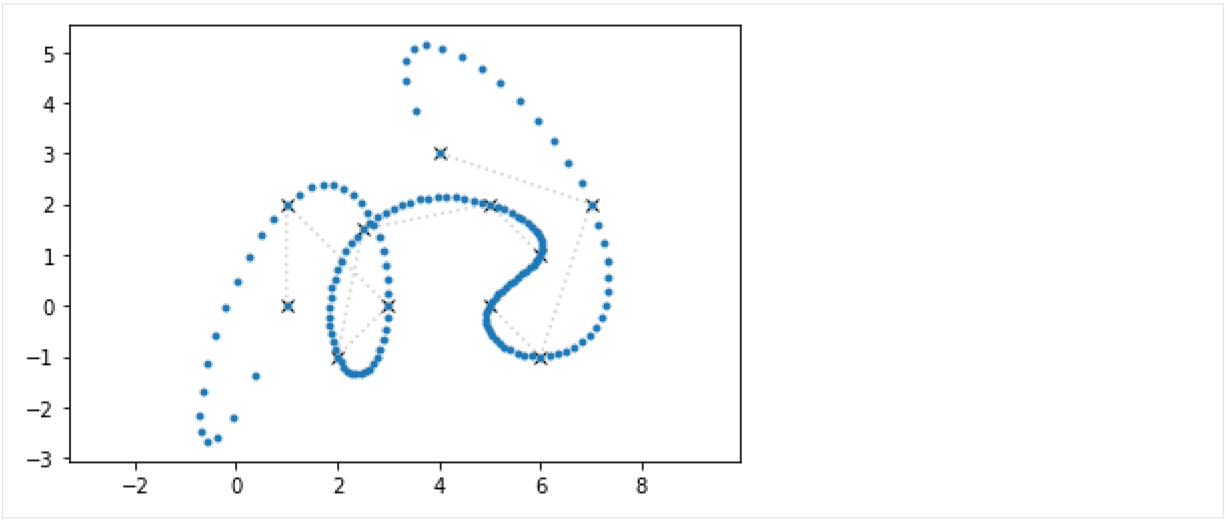


Runge's Phenomenon

This seems to work reasonably 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, -1),
    (7, 2),
    (4, 3),
]
times = range(len(vertices))
```

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



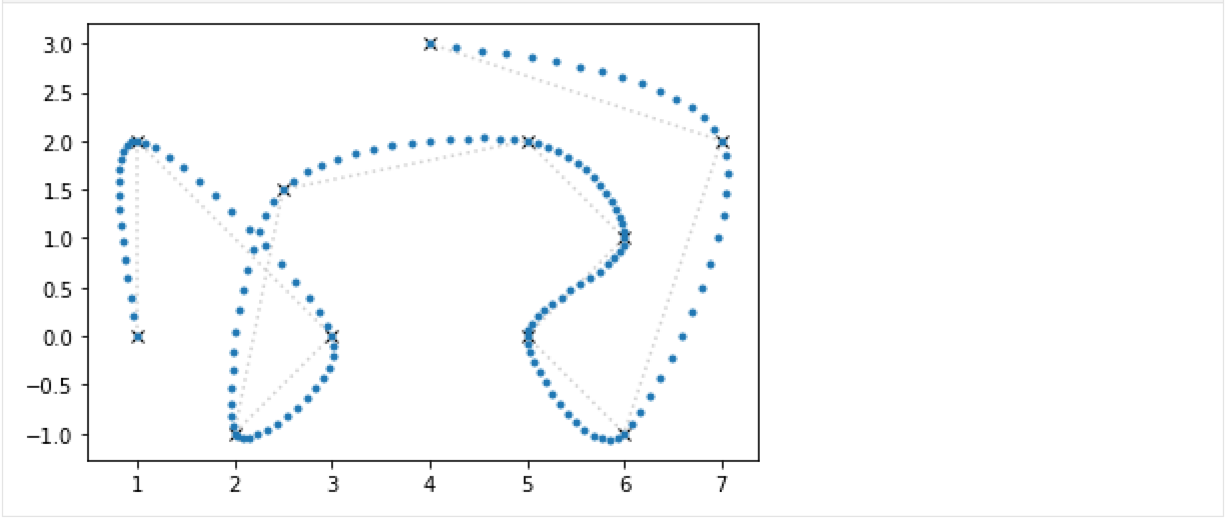
Here we see a severe overshooting effect, most pronounced at the beginning and the end of the curve. This effect is called Runge's phenomenon[5].

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 55):

[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.

Note

The *splines.CatmullRom* (page 161) class uses *"natural" end conditions* (page 101) by default.

.... doc/euclidean/lagrange.ipynb ends here.

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

1.3 Hermite Splines

Hermite splines[6] (named after Charles Hermite[7]) are the building blocks for many other types of interpolating polynomial splines, for example *natural splines* (page 29) and *Catmull–Rom splines* (page 55).

A Python implementation of (cubic) Hermite splines is available in the *splines.CubicHermite* (page 160) 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]:
```python
import splines
```

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

We import a few helper functions from helper.py:

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

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]:
```python
slopes = 0, 0, -1, 0.5, 1, 3
```

We are using the *splines.CubicHermite* (page 160) class to create the spline:

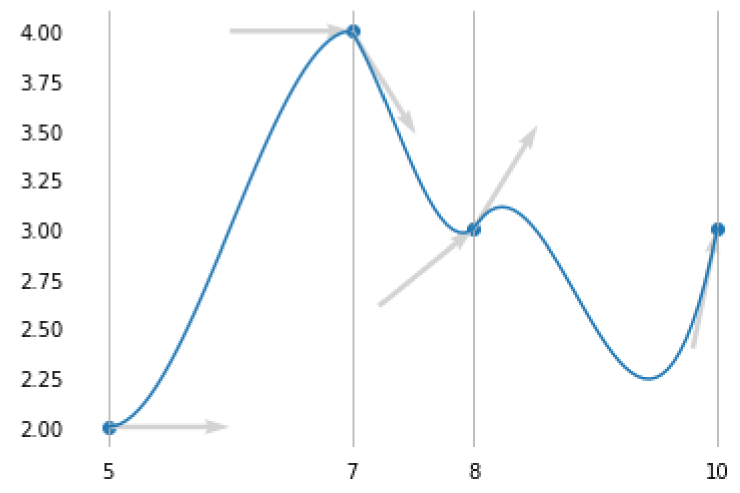
[6] https://en.wikipedia.org/wiki/Cubic_Hermite_spline
[7] https://en.wikipedia.org/wiki/Charles_Hermite

```
[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).

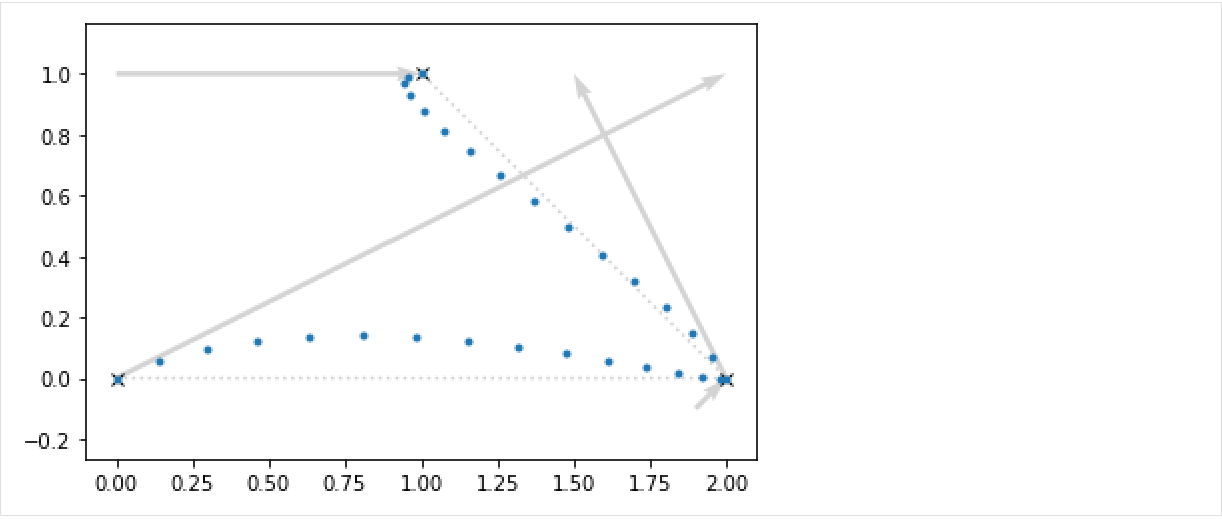
```
[9]: vertices = [
         (0, 0),
         (2, 0),
         (1, 1),
     ]
```

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]: fig, ax = plt.subplots()
      plot_spline_2d(s2, ax=ax)
      plot_tangents_2d(tangents, vertices, ax=ax)
```



If no parameter values are given (by means of the `grid` argument), the *splines.CubicHermite* (page 160) 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:

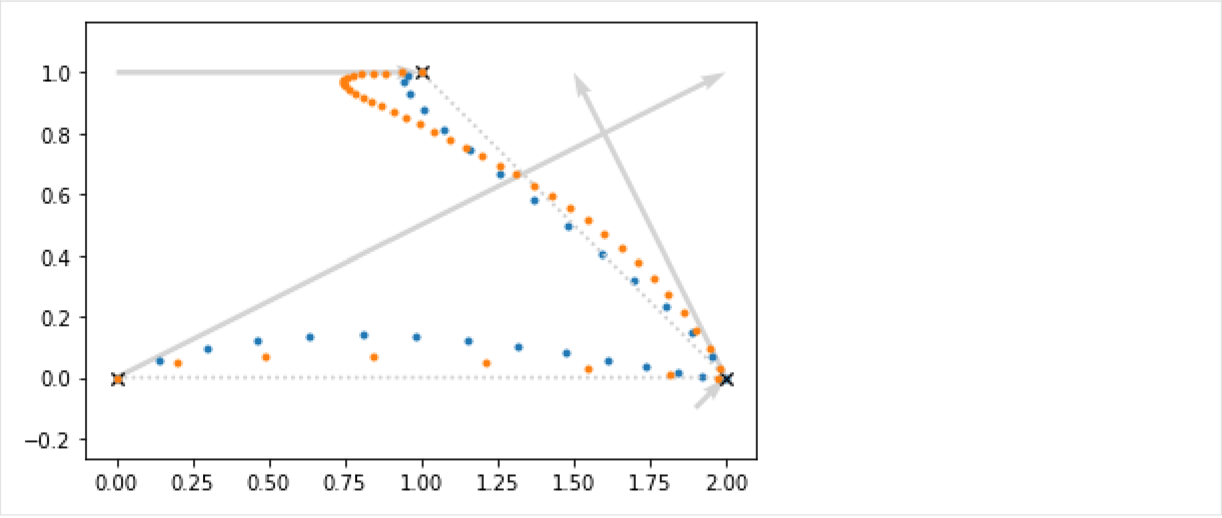
```
[13]: grid = 0, 0.5, 3
```

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)
```

```
[15]: plot_spline_2d(s3, ax=ax)
      fig
```

[15]:



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 55). *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 101)) that leads to continuous second derivatives at all vertices, making the spline C^2 continuous. This is what *natural splines* (page 29) 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 derivation 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='grevlex')
```

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, $\boldsymbol{a_4}$ to $\boldsymbol{d_4}$.

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

[4]:
$$\left[\begin{matrix}\boldsymbol{d}_4\\ \boldsymbol{c}_4\\ \boldsymbol{b}_4\\ \boldsymbol{a}_4\end{matrix}\right]$$

Combined with the *monomial basis* . . .

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

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

. . . the coefficients form an expression for our polynomial segment $\boldsymbol{p}_4(t)$:

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

[6]:
$$\boldsymbol{p}_4 = \boldsymbol{d}_4 t^3 + \boldsymbol{c}_4 t^2 + \boldsymbol{b}_4 t + \boldsymbol{a}_4$$

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

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.

$$\begin{aligned}
\boldsymbol{x}_4 &= \boldsymbol{p}_4\big|_{t=0}\\
\boldsymbol{x}_5 &= \boldsymbol{p}_4\big|_{t=1}\\
\dot{\boldsymbol{x}}_4 &= \frac{d}{dt}\boldsymbol{p}_4\bigg|_{t=0}\\
\dot{\boldsymbol{x}}_5 &= \frac{d}{dt}\boldsymbol{p}_4\bigg|_{t=1}
\end{aligned}$$

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)
```

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

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

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

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

Basis Matrix

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]:

$$\begin{bmatrix}\boldsymbol{x}_4\\ \boldsymbol{x}_5\\ \dot{\boldsymbol{x}}_4\\ \dot{\boldsymbol{x}}_5\end{bmatrix}$$

... 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]:

$$\begin{bmatrix}\boldsymbol{d}_4\\ \boldsymbol{c}_4\\ \boldsymbol{b}_4\\ \boldsymbol{a}_4\end{bmatrix} = M_\text{H}\begin{bmatrix}\boldsymbol{x}_4\\ \boldsymbol{x}_5\\ \dot{\boldsymbol{x}}_4\\ \dot{\boldsymbol{x}}_5\end{bmatrix}$$

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

... 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]])
```

Basis Polynomials

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
```

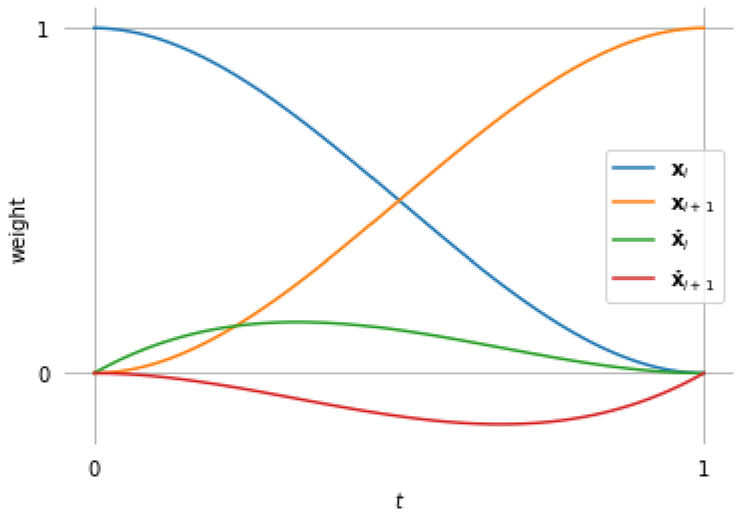
[20]:

$$b_{\mathrm{H}}{}^{T} = \begin{bmatrix} (t-1)^2\,(2t+1) \\ t^2\,(-2t+3) \\ t\,(t-1)^2 \\ t^2\,(t-1) \end{bmatrix}$$

Let's plot the basis polynomials with some help from helper.py:

[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),
```

(continues on next page)

(continued from previous page)

```
    b.diff(t).subs(t, 1),
] for b in b_H.expr])
```

[23]:

$$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}$$

Example Plot

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

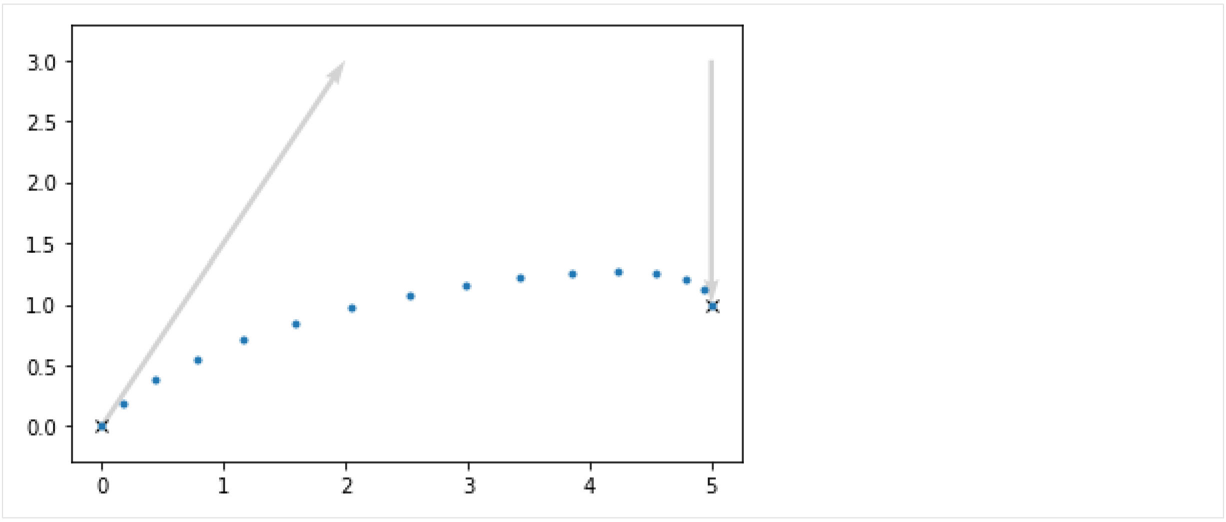
The @ operator is used here to do NumPy's matrix multiplication[8].

[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)
```

[8] 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]:

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

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]:

$$\begin{bmatrix}x_4\\\tilde{x}_4\\\tilde{x}_5\\x_5\end{bmatrix} = M_{\text{H}\to\text{B}}\begin{bmatrix}x_4\\x_5\\\dot{x}_4\\\dot{x}_5\end{bmatrix}$$

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]:

$$M_{\text{H}\to\text{B}} = \frac{1}{3}\begin{bmatrix}3 & 0 & 0 & 0\\3 & 0 & 1 & 0\\0 & 3 & 0 & -1\\0 & 3 & 0 & 0\end{bmatrix}$$

```
[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]:

$$M_{\text{B}\to\text{H}} = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 0 & 0 & 1\\-3 & 3 & 0 & 0\\0 & 0 & -3 & 3\end{bmatrix}$$

```
[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]: $M_{\text{B}} = M_{\text{H}} M_{\text{B}\to\text{H}}$

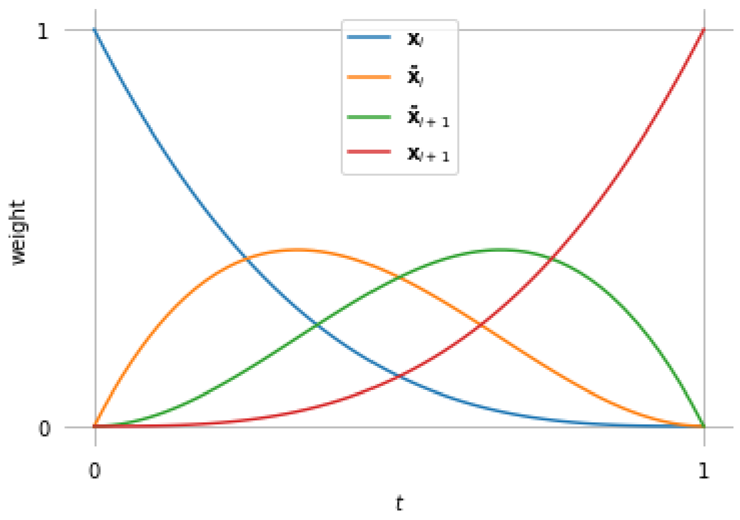
[39]:
```
M_B = M_B.subs_symbols(M_H, M_BtoH).doit()
M_B
```

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

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

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

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



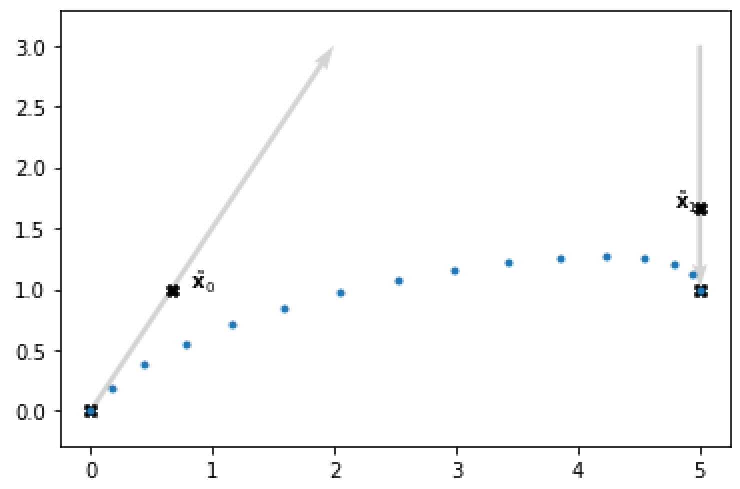
Those happen to be the cubic *Bernstein* polynomials and it turns out that we just invented *Bézier* curves! See *the section about Bézier splines* (page 39) for more information about them.

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 $M_{\text{H}\to\text{B}}$:

[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');
```



. . . . `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 16), 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).

The class *splines.CubicHermite* (page 160) is implemented using the second approach (because its parent class *splines.Monomial* (page 159) also uses the re-scaling approach).

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='grevlex')
```

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

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

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

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

The next steps are very similar to what we did in the *uniform case* (page 16), 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))
```

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

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

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

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

Basis Matrix

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

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

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

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

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

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

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

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

Basis Polynomials

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

[20]:

$$
{b_{\text{H},4}}^{T} = \left[\begin{matrix}(t-1)^{2}(2t+1)\\ t^{2}(-2t+3)\\ \Delta_{4} t (t-1)^{2}\\ \Delta_{4} t^{2}(t-1)\end{matrix}\right]
$$

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.

Example Plot

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.

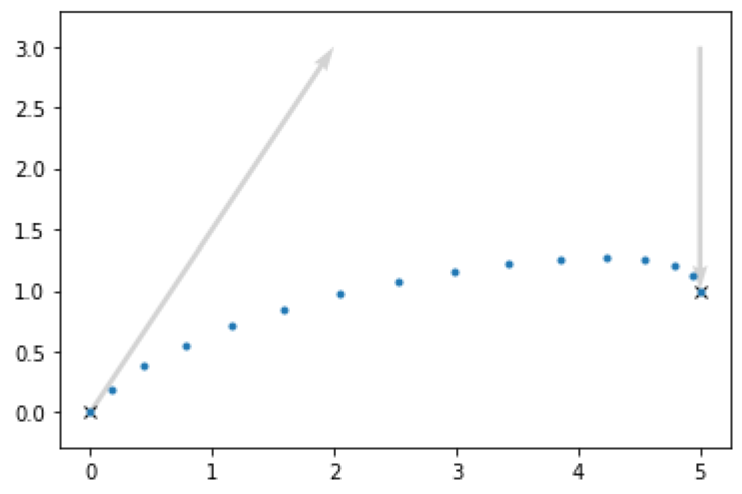
```
[23]: class HermiteSegment:

          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)
```

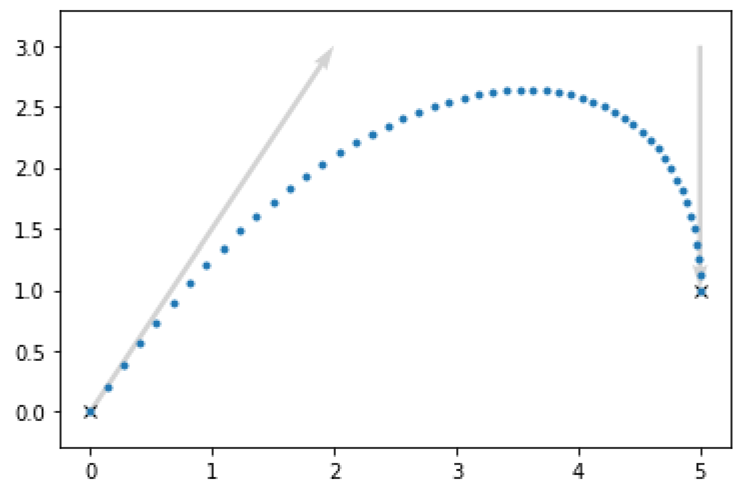
```
[26]: plot_spline_2d(s1, chords=False)
      plot_tangents_2d(tangents, vertices)
```



But other ranges should work as well:

```
[27]: s2 = HermiteSegment([*vertices, *tangents], 2.1, 5.5)
```

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



Utilizing the Uniform Basis Matrix

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 16), 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.Eq(
    sp.simplify(M_H.expr * control_values_H.name),
    sp.simplify(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[9] (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.

1.4 Natural Splines

Sometimes simply called (cubic) spline interpolation[10], a *natural* spline is modelled after a drawing tool called spline[11], which is made from a thin piece of elastic material like wood or metal.

A Python implementation is available in the class *splines.Natural* (page 161). Alternatively, the CubicSpline[12] class from SciPy can be used.

[9] https://en.wikipedia.org/wiki/Chain_rule
[10] https://en.wikipedia.org/wiki/Spline_interpolation
[11] https://en.wiktionary.org/wiki/spline
[12] 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

The most important property of natural splines is that they are C^2 continuous, which means that the second derivatives match at section borders.

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

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

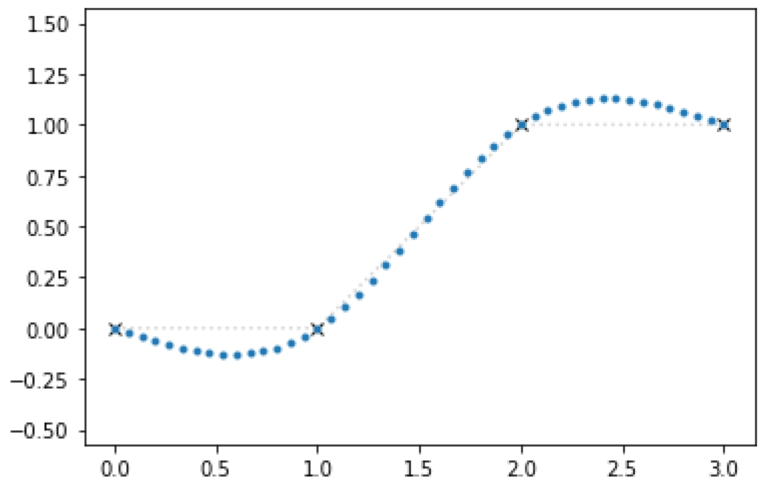
We use the class *splines.Natural* (page 161) . . .

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

. . . and a plotting function from 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)
```

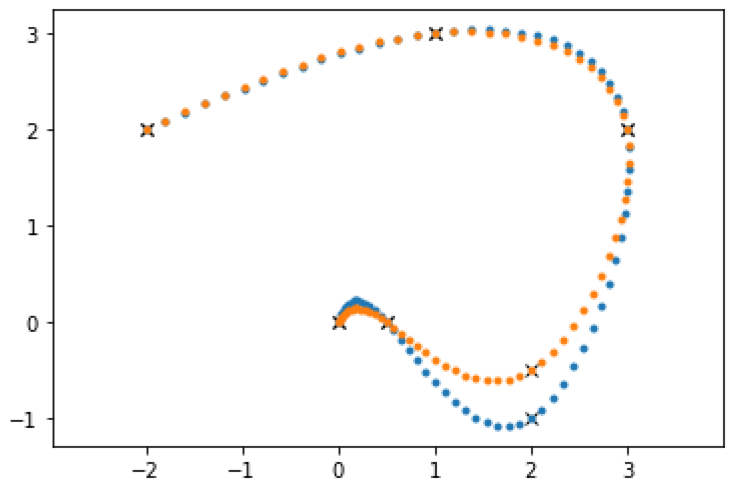
A downside of natural splines is that they don't provide *local control*. Changing only a single control point potentially influences the whole curve.

```
[7]: plot_natural([
         (0, 0),
         (0.5, 0),
         (2, -1),
         (3, 2),
         (1, 3),
         (-2, 2),
     ])
```

(continues on next page)

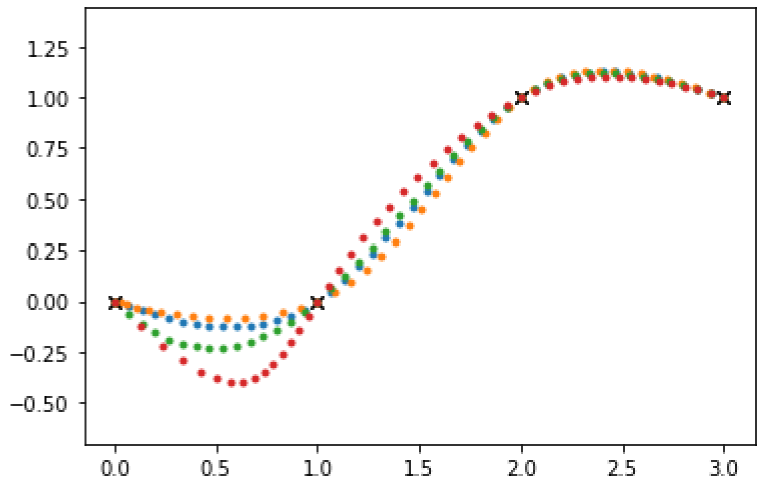
(continued from previous page)

```
plot_natural([
    (0, 0),
    (0.5, 0),
    (2, -0.5),
    (3, 2),
    (1, 3),
    (-2, 2),
])
```

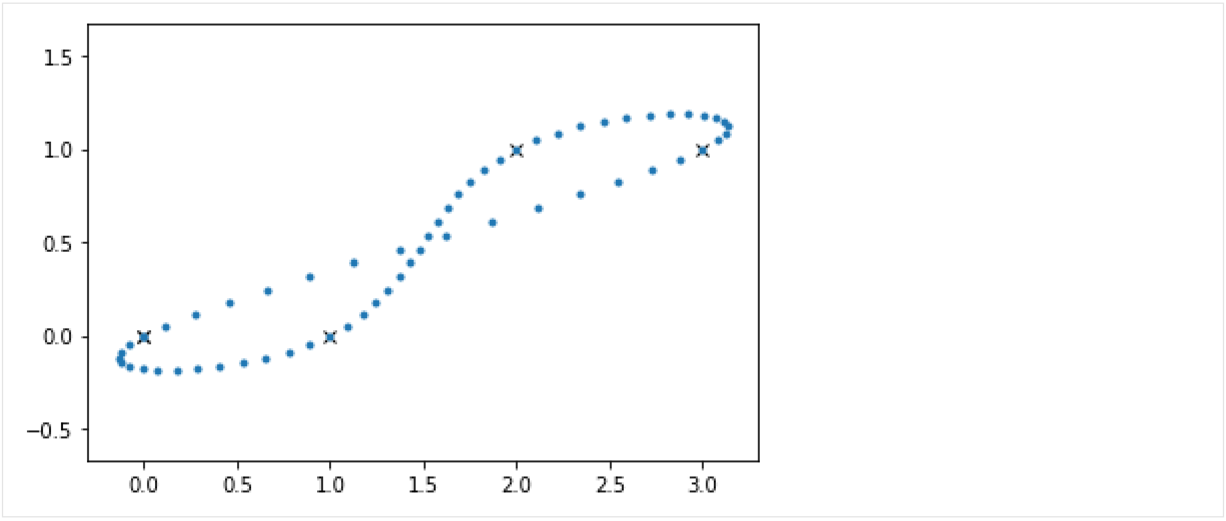


By default, *natural end conditions* (page 35) are used, but alternatively, the end tangents can be *clamped* to given values.

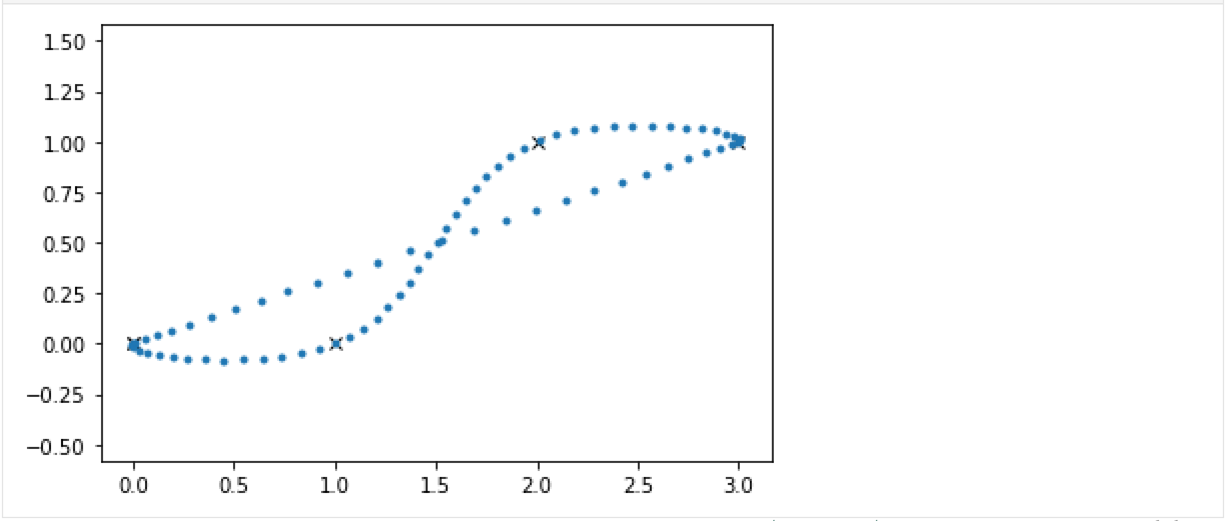
[8]:
```
plot_natural(vertices, endconditions='natural')
plot_natural(vertices, endconditions=[[0, 0], 'natural'])
plot_natural(vertices, endconditions=[[1, -1], 'natural'])
plot_natural(vertices, endconditions=[[2, -2], 'natural'])
```



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



```
[10]: 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

For deriving natural splines, we first look at the *uniform* case, which means that the parameter interval in each segment is chosen to be 1.

The more general case with arbitrary parameter intervals is derived in a separate *notebook about non-uniform natural splines* (page 36).

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

We import a helper class for named SymPy expressions from utility.py:

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

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

In the beginning, we are looking at two neighboring segments: The fourth segment, from x_3 to x_4, defined by the polynomial p_3, and the fifth segment, from x_4 to x_5, defined by the polynomial p_4. In both cases, we use $0 \leq t \leq 1$.

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

We are using the *monomial basis* (page 2) to define the two polynomials . . .

```
[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)
```

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

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

. . . and we calculate their first derivatives:

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

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

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

From this, we obtain 8 equations containing the 8 yet unknown coefficients.

```
[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)
```

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

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

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

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

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

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

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

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

We can solve the system of equations to get an expression for each coefficient:

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

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

$b_3 = \dot{x}_3$

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

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

$a_4 = x_4$

$b_4 = \dot{x}_4$

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

$d_4 = 2x_4 - 2x_5 + \dot{x}_4 + \dot{x}_5$

So far, this is the same as we have done in *the notebook about uniform Hermite splines* (page 16). In fact, the above constants are the same as in M_H!

An additional constraint for natural splines is that the second derivatives are continuous, so let's calculate those derivatives . . .

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

$\frac{d^2}{dt^2} p_3 = 6d_3 t + 2c_3$

$\frac{d^2}{dt^2} p_4 = 6d_4 t + 2c_4$

. . . and set them to be equal at the segment border:

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

[11]: $2c_3 + 6d_3 = 2c_4$

Inserting the equations from above leads to this equation:

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

[12]: $3x_3 = 3x_5 - \dot{x}_3 - 4\dot{x}_4 - \dot{x}_5$

We can generalize this expression by renaming index 4 to i:

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

This can be used for each segment – except for the very first and last one – yielding a matrix with N columns and $N-2$ rows:

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

End Conditions

We need a first and last row for this matrix to be able to fully define a natural spline. The following subsections show a selection of a few end conditions which can be used to obtain the missing rows of the matrix. End conditions (except "closed") can be mixed, e.g. "clamped" at the beginning and "natural" at the end. The Python class *splines.Natural* (page 161) uses "natural" end conditions by default.

Natural

Natural end conditions are commonly used for natural splines, which is probably why they are named that way.

There is a *separate notebook about "natural" end conditions* (page 101), from which we can get the uniform case by setting $\Delta_i = 1$:

$$\begin{aligned} 2\dot{x}_0 + \dot{x}_1 &= 3(x_1 - x_0) \\ \dot{x}_{N-2} + 2\dot{x}_{N-1} &= 3(x_{N-1} - x_{N-2}) \end{aligned}$$

Adding this to the matrix from above leads to a full $N \times N$ matrix:

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

Clamped

We can simply provide arbitrarily chosen values for the end tangents. This is called *clamped* end conditions.

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

This leads to a very simple first and last line:

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

Closed

We can close the spline by connecting x_{N-1} with x_0. This can be realized by cyclically extending the matrix in both directions:

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

Solving the System of Equations

The matrices above are *tridiagonal* and can therefore be solved efficiently with a tridiagonal matrix algorithm[13]. The class *splines.Natural* (page 161), however, is not very concerned about efficiency and simply uses NumPy's linalg.solve()[14] function to solve the system of equations.

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

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

Non-Uniform Natural Splines

The derivation is similar to *the uniform case* (page 32), but this time the parameter intervals can have arbitrary values.

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

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

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

Just like in the uniform case, we are considering two adjacent spline segments, but this time we must allow arbitrary parameter values:

[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]: $d_3t^3 + c_3t^2 + b_3t + a_3$

[13] https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm
[14] https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html

```
[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)
```

$$p_3 = \frac{d_3 (t - t_3)^3}{(-t_3 + t_4)^3} + \frac{c_3 (t - t_3)^2}{(-t_3 + t_4)^2} + \frac{b_3 (t - t_3)}{-t_3 + t_4} + a_3$$

$$p_4 = \frac{d_4 (t - t_4)^3}{(-t_4 + t_5)^3} + \frac{c_4 (t - t_4)^2}{(-t_4 + t_5)^2} + \frac{b_4 (t - t_4)}{-t_4 + t_5} + a_4$$

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

$$\frac{d}{dt} p_3 = \frac{3 d_3 (t - t_3)^2}{(-t_3 + t_4)^3} + \frac{c_3 (2t - 2t_3)}{(-t_3 + t_4)^2} + \frac{b_3}{-t_3 + t_4}$$

$$\frac{d}{dt} p_4 = \frac{3 d_4 (t - t_4)^2}{(-t_4 + t_5)^3} + \frac{c_4 (2t - 2t_4)}{(-t_4 + t_5)^2} + \frac{b_4}{-t_4 + t_5}$$

```
[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'),
      ]
```

We introduce a few new symbols 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))
```

$$x_3 = a_3$$

$$x_4 = a_3 + b_3 + c_3 + d_3$$

$$x_4 = a_4$$

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

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

$$\dot{x}_4 = \frac{b_3}{\Delta_3} + \frac{2c_3}{\Delta_3} + \frac{3d_3}{\Delta_3}$$

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

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

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

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

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

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

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

$$a_3 = x_3$$

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

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

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

$$a_4 = x_4$$

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

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

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

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

[16]:

$$\frac{3d_3\left(-2t_3 + 2t_4\right)}{\left(-t_3 + t_4\right)^3} + \frac{2c_3}{\left(-t_3 + t_4\right)^2} = \frac{2c_4}{\left(-t_4 + t_5\right)^2}$$

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

[17]:

$$\frac{2\left(\Delta_3\dot{x}_3 + 2\Delta_3\dot{x}_4 + 3x_3 - 3x_4\right)}{\Delta_3^2} = \frac{2\left(-2\Delta_4\dot{x}_4 - \Delta_4\dot{x}_5 - 3x_4 + 3x_5\right)}{\Delta_4^2}$$

Like in the uniform case, we can generalize by renaming index 4 to i:

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

We are not showing the full matrix here, because it would be quite a bit more complicated and less helpful than in the uniform case.

End Conditions

Like in the *uniform case* (page 35), we can come up with a few end conditions in order to define the missing matrix rows.

The Python class *splines.Natural* (page 161) uses "natural" end conditions by default.

"Natural" end conditions are derived in *a separate notebook* (page 101), yielding these expressions:

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

Other end conditions can be derived as shown in *the notebook about uniform "natural" splines* (page 35).

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

1.5 Bézier Splines

Named after Pierre Bézier[15], Bézier splines are defined by means of Bernstein polynomials[16] (which are named after Sergei Bernstein[17]). A popular method to evaluate Bézier splines at given parameter values is *de Casteljau's algorithm* (page 40). A very good online resource with many interactive examples is *A Primer on Bézier Curves*: https://pomax.github.io/bezierinfo/.

A Python implementation is available in the class *splines.Bernstein* (page 160).

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

Properties of Bézier Splines

The terms *Bézier spline* and *Bézier curve* can mean two slightly different things:

1. A curve constructed from a single Bernstein polynomial of degree d, given a *control polygon* consisting of a sequence of $d + 1$ vertices. The first and last vertex lies on the curve (at its start and end, respectively), while the other vertices in general don't (the curve *approximates* them).
2. A piecewise polynomial curve consisting of multiple segments, each of them constructed from a separate Bernstein polynomial. The start and end points of neighboring control polygons typically coincide, leading to C^0 continuity. However, the overall control polygon can be chosen in a way to achieve G^1 or C^1 (or even higher) continuity.

Bézier splines in the latter sense are well known from their common use in 2D vector graphics software, where cubic (i.e. degree 3) curve segments are typically used. Each segment has four control points: The start and end point of the segment (shared with the end and start of the previous and next segment, respectively) as well as two additional points that control the shape of the curve segment.

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

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

As an example, we create control points for a Bézier spline consisting of four segments, having polynomial degrees of 1, 2, 3 and 4.

[15] https://en.wikipedia.org/wiki/Pierre_B%C3%A9zier
[16] https://en.wikipedia.org/wiki/Bernstein_polynomial
[17] https://en.wikipedia.org/wiki/Sergei_Bernstein

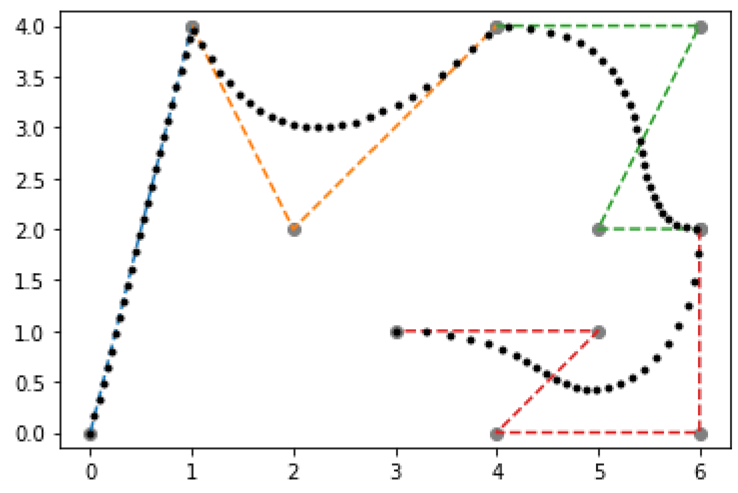
[3]:
```
control_points = [
    [(0, 0), (1, 4)],
    [(1, 4), (2, 2), (4, 4)],
    [(4, 4), (6, 4), (5, 2), (6, 2)],
    [(6, 2), (6, 0), (4, 0), (5, 1), (3, 1)],
]
```

We are using the *splines.Bernstein* (page 160) class to construct a Bézier splines from these control points.

[4]:
```
s = splines.Bernstein(control_points)
```

[5]:
```
times = np.linspace(s.grid[0], s.grid[-1], 100)
```

[6]:
```
fig, ax = plt.subplots()
for segment in control_points:
    xy = np.transpose(segment)
    ax.plot(*xy, '--')
    ax.scatter(*xy, color='grey')
ax.plot(*s.evaluate(times).T, 'k.')
ax.axis('equal');
```



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

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

De Casteljau's Algorithm

There are several ways that lead to Bézier curves, one was already shown in *the notebook about Hermite curves* (page 21) (but only for cubic curves). In this notebook, we will derive Bézier curves of arbitrary polynomial degree utilizing De Casteljau's algorithm[18].

[18] https://en.wikipedia.org/wiki/De_Casteljau's_algorithm

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 a few utilities and helpers from the files utility.py and helper.py.

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

Let's prepare a few symbols for later use . . .

[3]:
```
t, x0, x1, x2, x3, x4 = sp.symbols('t, xbm:5')
```

. . . and a helper function for plotting:

[4]:
```
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.plot(
        *np.asarray(points).T,
        color='lightgrey',
        linestyle=':',
        marker='x',
        markeredgecolor='black',
    )
    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:

[5]:
```
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 (Linear)

After all those preparations, 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.

Assuming that we have two control points, $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$, we can set up a linear equation:

$$\boldsymbol{p}_{0,1}(t) = \boldsymbol{x}_0 + t(\boldsymbol{x}_1 - \boldsymbol{x}_0).$$

Another way to write the same thing it like this:

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

where in both cases $t \in [0, 1]$. Since we will be needing quite a few of those linear interpolations, let's create a helper function:

[6]:
```
def lerp(one, two):
    """Linear interpolation.

    The parameter *t* is expected to be between 0 and 1.

    """
    return (1 - t) * one + t * two
```

Now we can define the equation in SymPy:

[7]:
```
p01 = NamedExpression('pbm_0,1', lerp(x0, x1))
p01
```

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

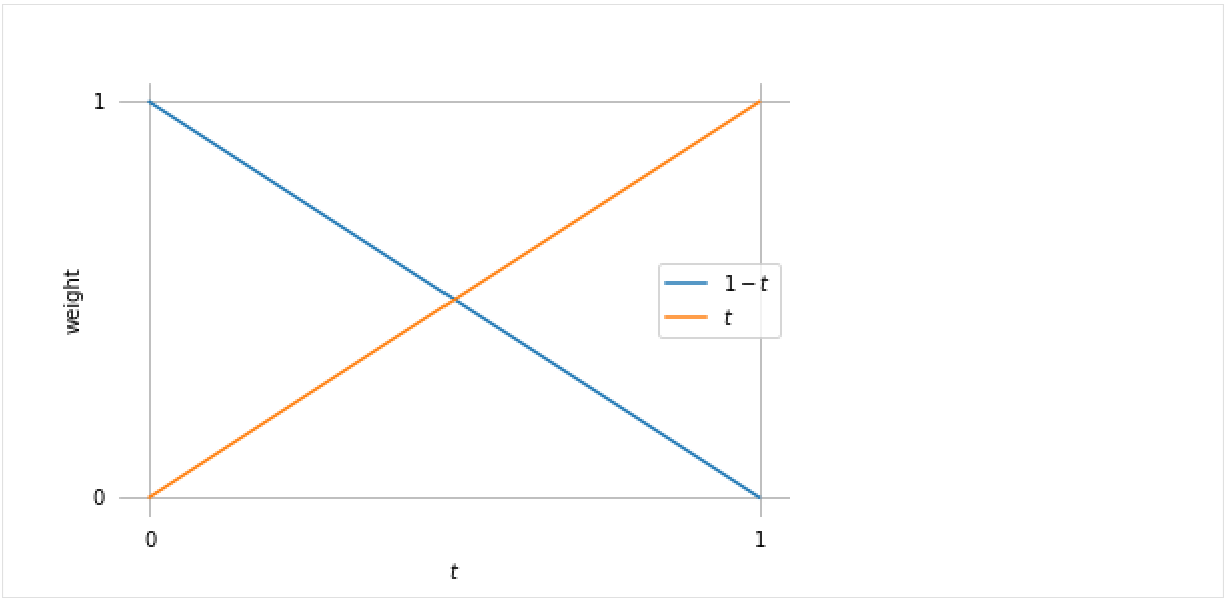
[8]:
```
b1 = [p01.expr.expand().coeff(x.name).factor() for x in (x0, x1)]
b1
```

[8]: $[1 - t,\ t]$

Doesn't look like much, but those are the Bernstein bases[19] for degree 1. It doesn't get much more interesting if we plot them:

[9]:
```
plot_basis(*b1)
```

[19] https://en.wikipedia.org/wiki/Bernstein_polynomial



If you want to convert this to coefficients for the *monomial basis* (page 2) $[t, 1]$ instead of the Bernstein basis functions, you can use this matrix:

```
[10]: 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
```

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

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

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

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

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

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

In case you ever need that, here's the inverse:

```
[13]: M_B1.I
```

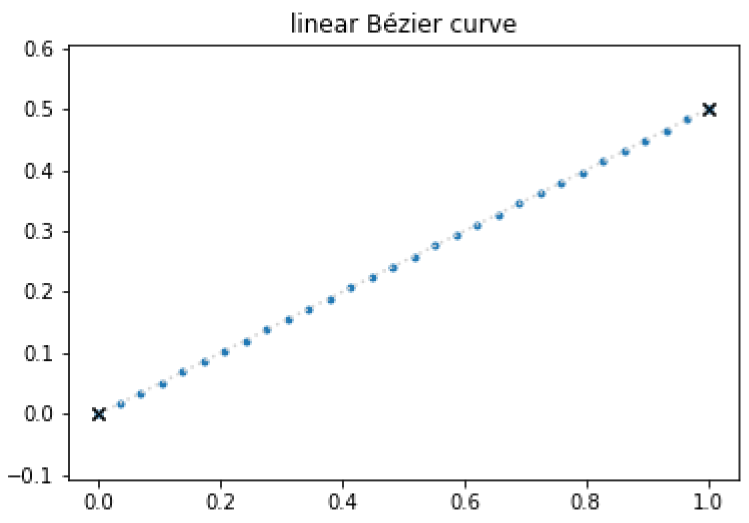
[13]: $M_{\text{B}}^{(1)\,-1} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$

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

```
[14]: 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
```

[15]:
```
points = [
    (0, 0),
    (1, 0.5),
]
```

[16]:
```
plot_curve(linear, points)
```



[17]:
```
show_casteljau_animation(points)
```

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

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

Degree 2 (Quadratic)

Now we consider three control points, x_0, x_1 and x_2. We use the linear interpolation of the first two points from above . . .

[18]:
```
p01
```

[18]: $p_{0,1} = tx_1 + x_0(1-t)$

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

[19]:
```
p12 = NamedExpression('pbm_1,2', lerp(x1, x2))
p12
```

[19]: $p_{1,2} = tx_2 + x_1(1-t)$

Finally, we make another linear interpolation between those two results:

[20]:
```
p02 = NamedExpression('pbm_0,2', lerp(p01.expr, p12.expr))
p02
```

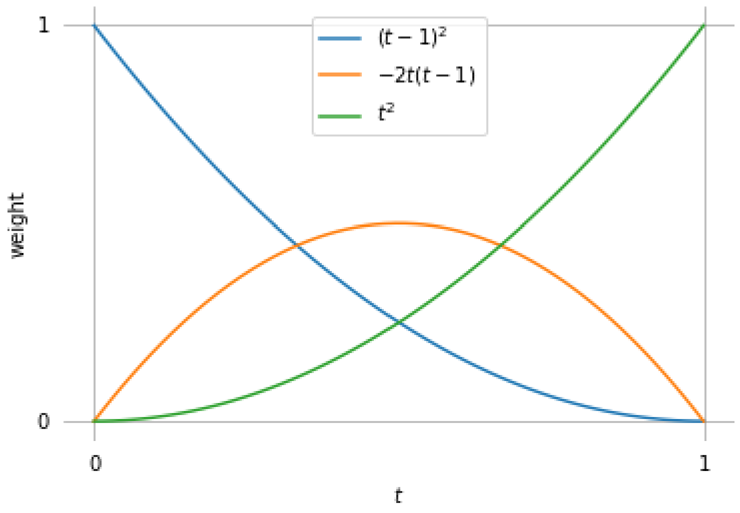
[20]: $p_{0,2} = t(tx_2 + x_1(1-t)) + (1-t)(tx_1 + x_0(1-t))$

From this, we can get the Bernstein basis functions of degree 2:

[21]:
```python
b2 = [p02.expr.expand().coeff(x.name).factor() for x in (x0, x1, x2)]
b2
```

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

[22]:
```python
plot_basis(*b2)
```



[23]:
```python
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
```

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

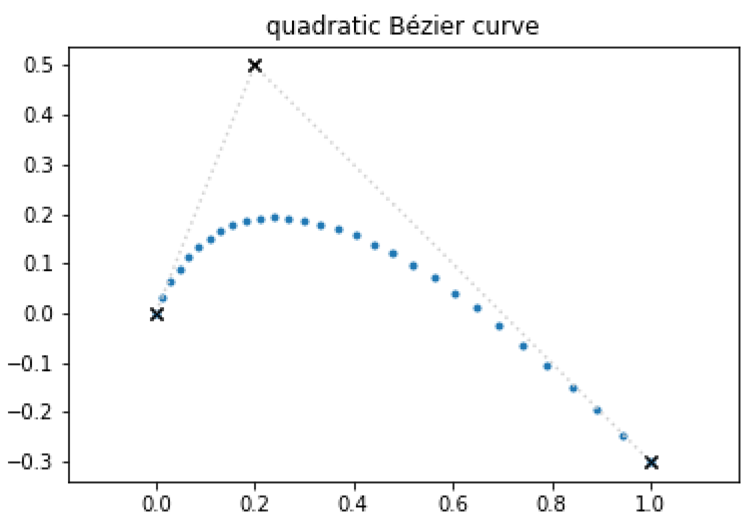
[24]:
```python
M_B2.I
```

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

[25]:
```python
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
```

[26]:
```python
points = [
    (0, 0),
    (0.2, 0.5),
    (1, -0.3),
]
```

[27]:
```python
plot_curve(quadratic, points)
```



[28]:
```
show_casteljau_animation(points)
```
```
Animations can only be shown in HTML output, sorry!
```

Quadratic Tangent Vectors

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

[29]:
```
v02 = p02.diff(t)
```

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

[30]:
```
v02.evaluated_at(t, 0)
```

[30]:
$$\left.\frac{d}{dt}\boldsymbol{p}_{0,2}\right|_{t=0} = -2\boldsymbol{x}_0 + 2\boldsymbol{x}_1$$

[31]:
```
v02.evaluated_at(t, 1)
```

[31]:
$$\left.\frac{d}{dt}\boldsymbol{p}_{0,2}\right|_{t=1} = -2\boldsymbol{x}_1 + 2\boldsymbol{x}_2$$

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

[32]:
```
assert (v02.expr - 2 * (p12.expr - p01.expr)).simplify() == 0
```

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

Degree 3 (Cubic)

Let's now 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 linearly interpolate it with the result for the three points x_1, x_2 and x_3.

Combination of x_2 and x_3:

```
[33]: p23 = NamedExpression('pbm_2,3', lerp(x2, x3))
      p23
```

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

Combination of x_1, x_2 and x_3:

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

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

Combination of x_0, x_1, x_2 and x_3:

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

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

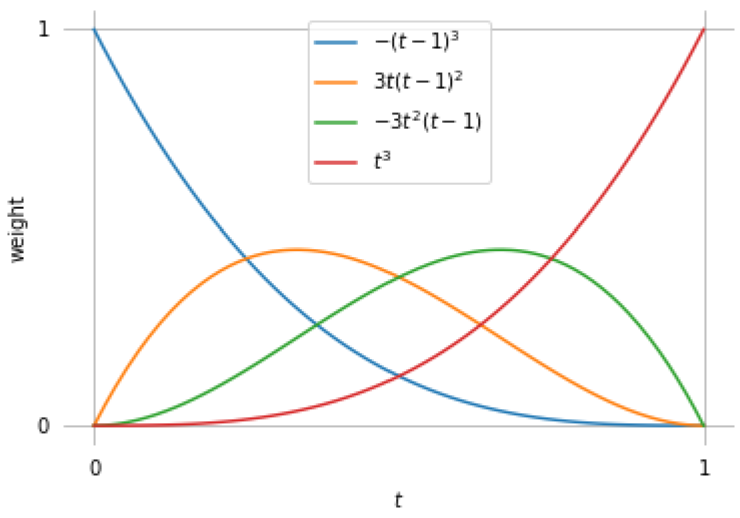
This leads to the cubic Bernstein bases:

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

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

Those are of course the same Bernstein bases as we found in *the notebook about Hermite splines* (page 21).

```
[37]: plot_basis(*b3)
```



[38]:
```
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
```

[38]:

$$M_\text{B}^{(3)} = \begin{bmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 3 & 0\\ -3 & 3 & 0 & 0\\ 1 & 0 & 0 & 0\end{bmatrix}$$

[39]:
```
M_B3.I
```

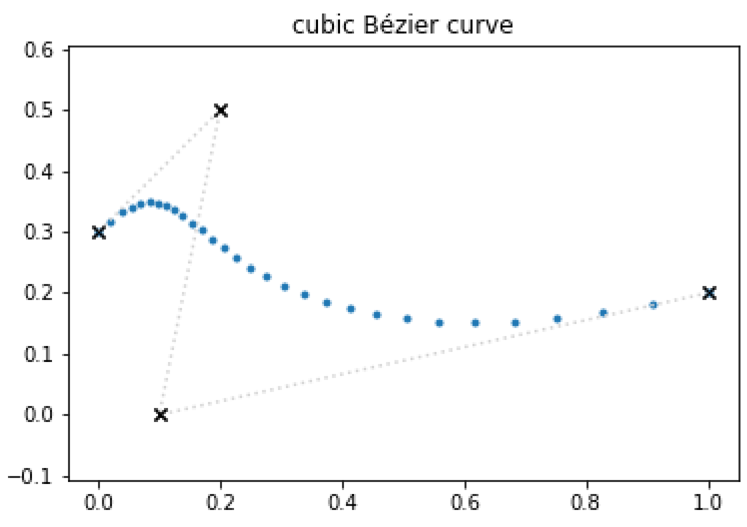
[39]:

$${M_\text{B}^{(3)}}^{-1} = \begin{bmatrix}0 & 0 & 0 & 1\\ 0 & 0 & \frac{1}{3} & 1\\ 0 & \frac{1}{3} & \frac{2}{3} & 1\\ 1 & 1 & 1 & 1\end{bmatrix}$$

[40]:
```
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
```

[41]:
```
points = [
    (0, 0.3),
    (0.2, 0.5),
    (0.1, 0),
    (1, 0.2),
]
```

[42]:
```
plot_curve(cubic, points)
```



[43]:
```
show_casteljau_animation(points)
```

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

Cubic Tangent Vectors

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

[44]:
```
v03 = p03.diff(t)
```

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

[45]:
```
v03.evaluated_at(t, 0)
```

[45]: $\left.\frac{d}{dt}p_{0,3}\right|_{t=0} = -3x_0 + 3x_1$

[46]:
```
v03.evaluated_at(t, 1)
```

[46]: $\left.\frac{d}{dt}p_{0,3}\right|_{t=1} = -3x_2 + 3x_3$

This shows that the tangent vector at the beginning and end of the curve is parallel to the line from x_0 to x_1 and from x_2 to x_3, respectively. The length of the tangent vectors is three times the length of those lines. This also means that if the begin and end positions x_0 and x_3 as well as the corresponding tangent vectors $\dot{x}_0$ and $\dot{x}_3$ are given, it's easy to calculate the two missing control points:

$$x_1 = x_0 + \frac{\dot{x}_0}{3}$$
$$x_2 = x_3 - \frac{\dot{x}_3}{3}$$

This can be used to *turn uniform Hermite splines into Bézier splines* (page 21) and to *construct uniform Catmull–Rom splines using Bézier segments* (page 70).

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

[47]:
```
assert (v03.expr - 3 * (p13.expr - p02.expr)).simplify() == 0
```

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

Cubic Bézier to Hermite Segments

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:

[48]:
```
p03.evaluated_at(t, 0)
```

[48]: $\left.p_{0,3}\right|_{t=0} = x_0$

[49]:
```
p03.evaluated_at(t, 1)
```

[49]: $\left.p_{0,3}\right|_{t=1} = x_3$

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 a Hermite spline segment:

[50]:
```
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
```

[50]:
$$M_{\text{B}\to\text{H}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix}$$

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

[51]:
```
M_BtoH.I.pull_out(sp.S.One / 3)
```

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

Of course, those are the same matrices as shown in the *notebook about uniform cubic Hermite splines* (page 21).

Degree 4 (Quartic)

By now you know the drill, we consider five control points, x_0, x_1, x_2, x_3 and x_4, which lead to more linear interpolations:

[52]:
```
p34 = NamedExpression('pbm_3,4', lerp(x3, x4))
p24 = NamedExpression('pbm_2,4', lerp(p23.expr, p34.expr))
p14 = NamedExpression('pbm_1,4', lerp(p13.expr, p24.expr))
p04 = NamedExpression('pbm_0,4', lerp(p03.expr, p14.expr))
p04
```

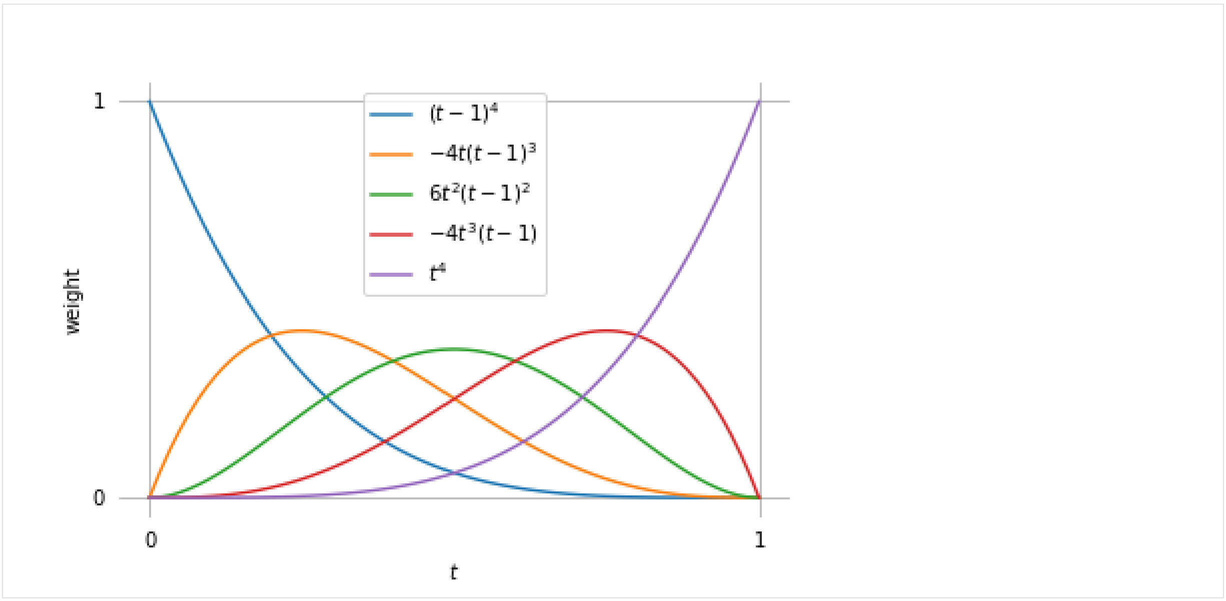
[52]:
$$p_{0,4} = t\left(t\left(t\left(tx_4 + x_3\left(1-t\right)\right) + \left(1-t\right)\left(tx_3 + x_2\left(1-t\right)\right)\right) + \left(1-t\right)\left(t\left(tx_3 + x_2\left(1-t\right)\right) + \left(1-t\right)\left(tx_2 + x_1\left(1-t\right)\right)\right)\right) + \left(1-t\right)\left(t\left(t\left(tx_3 + x_2\left(1-t\right)\right) + \left(1-t\right)\left(tx_2 + x_1\left(1-t\right)\right)\right) + \left(1-t\right)\left(t\left(tx_2 + x_1\left(1-t\right)\right) + \left(1-t\right)\left(tx_1 + x_0\left(1-t\right)\right)\right)\right)$$

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

[53]:
```
b4 = [p04.expr.expand().coeff(x.name).factor() for x in (x0, x1, x2, x3, x4)]
b4
```

[53]:
$$\left[\left(t-1\right)^4,\ -4t\left(t-1\right)^3,\ 6t^2\left(t-1\right)^2,\ -4t^3\left(t-1\right),\ t^4\right]$$

[54]:
```
plot_basis(*b4)
```



```
[55]: 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
```

[55]:

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

```
[56]: M_B4.I
```

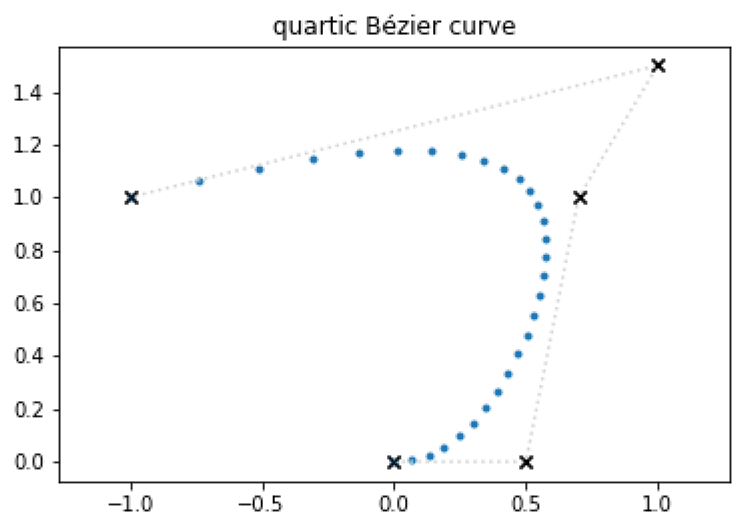
[56]:

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

```
[57]: 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
```

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

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



[60]:
```
show_casteljau_animation(points)
```
```
Animations can only be shown in HTML output, sorry!
```

Quartic Tangent Vectors

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

[61]:
```
v04 = p04.diff(t)
```

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

[62]:
```
v04.evaluated_at(t, 0)
```

[62]:
$$\left.\frac{d}{dt}p_{0,4}\right|_{t=0} = -4x_0 + 4x_1$$

[63]:
```
v04.evaluated_at(t, 1)
```

[63]:
$$\left.\frac{d}{dt}p_{0,4}\right|_{t=1} = -4x_3 + 4x_4$$

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 x_0 to x_1 and from x_3 to 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 ($p_{1,4}(t) - p_{0,3}(t)$ in this case) is exactly a fourth of the tangent vector (at any given $t \in [0, 1]$):

[64]:
```
assert (v04.expr - 4 * (p14.expr - p03.expr)).simplify() == 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 (using the binomial coefficient[20] $\binom{n}{i} = \frac{n!}{i!(n-i)!}$):

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

This is used in the Python class *splines.Bernstein* (page 160).

```
[65]: 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.

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

Non-Uniform Bézier Splines

Very commonly, Bézier splines are used with a parameter range of $0 \le t \le 1$, which has also been used to derive the basis polynomials and basis matrices in *the notebook about De Casteljau's algorithm* (page 40).

The parameter range can be re-scaled to any desired parameter range, but since the shape of a Bézier curve is fully defined by its control polygon, this will not change the shape of the curve, but only its speed, and therefore its tangent vectors.

To derive equations for non-uniform tangent vectors, let us quickly re-implement De Casteljau's algorithm:

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

```
[2]: def de_casteljau(points, t):
         while len(points) > 1:
             points = [lerp(a, b, t) for a, b in zip(points, points[1:])]
         return points[0]
```

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

We'll also use our trusty SymPy tools from utility.py:

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

[20] https://en.wikipedia.org/wiki/Binomial_coefficient

In this notebook we are only looking at cubic Bézier splines. More specifically, we are looking at the fifth spline segment, from x_4 to x_5 within a parameter range from t_4 to t_5, but later we can easily generalize this.

```
[5]: control_points = sp.symbols('xbm4 xtildebm4^(+) xtildebm5^(-) xbm5')
     control_points
```

[5]: $\left(x_4,\ \tilde{x}_4^{(+)},\ \tilde{x}_5^{(-)},\ x_5\right)$

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

As before, we are using De Casteljau's algorithm, but this time we are re-scaling the parameter range using the transformation $t \to \frac{t-t_i}{t_{i+1}-t_i}$:

```
[7]: p4 = NamedExpression(
         'pbm4',
         de_casteljau(control_points, (t - t4) / (t5 - t4)))
```

Tangent Vectors

As always, the tangent vectors can be obtained by means of the first derivative:

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

```
[9]: pd4.evaluated_at(t, t4)
```

[9]: $\frac{d}{dt}p_4\Big|_{t=t_4} = -\frac{3x_4}{-t_4+t_5} + \frac{3\tilde{x}_4^{(+)}}{-t_4+t_5}$

This expression for the outgoing tangent vector at x_4 can be generalized to

$$\dot{x}_i^{(+)} = \frac{3\left(\tilde{x}^{(+)}{}_i - x_i\right)}{\Delta_i},$$

where $\Delta_i = t_{i+1} - t_i$.

Similarly, the incoming tangent vector at x_5 ...

```
[10]: pd4.evaluated_at(t, t5)
```

[10]: $\frac{d}{dt}p_4\Big|_{t=t_5} = \frac{3x_5}{-t_4+t_5} - \frac{3\tilde{x}_5^{(-)}}{-t_4+t_5}$

... can be generalized to

$$\dot{x}_i^{(-)} = \frac{3\left(x_i - \tilde{x}^{(-)}{}_i\right)}{\Delta_{i-1}}.$$

This is similar to the *uniform case* (page 49), the tangent vectors are just divided by the parameter interval.

Control Points From Tangent Vectors

If the tangent vectors are given in the first place (i.e. when a non-uniform *Hermite spline* (page 13) is given), the cubic Bézier control points can be calculated like this:

$$\tilde{x}^{(+)}{}_i = x_i + \frac{\Delta_i \dot{x}_i^{(+)}}{3}$$
$$\tilde{x}^{(-)}{}_i = x_i - \frac{\Delta_{i-1} \dot{x}_i^{(-)}}{3}$$

.... `doc/euclidean/bezier-non-uniform.ipynb` ends here.

1.6 Catmull–Rom Splines

What is nowadays known as *Catmull–Rom spline* (named after Edwin Catmull[21] and Raphael Rom[22]) 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 161) 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*.

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 13). 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 161) to create both uniform and non-uniform splines. Only closed splines are shown, other *end conditions* (page 101) can also be used, but they are not specific to this type of spline.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
np.set_printoptions(precision=4)
```

Apart from the *splines* (page 159) module ...

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

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

[21] https://en.wikipedia.org/wiki/Edwin_Catmull
[22] https://en.wikipedia.org/wiki/Raphael_Rom

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

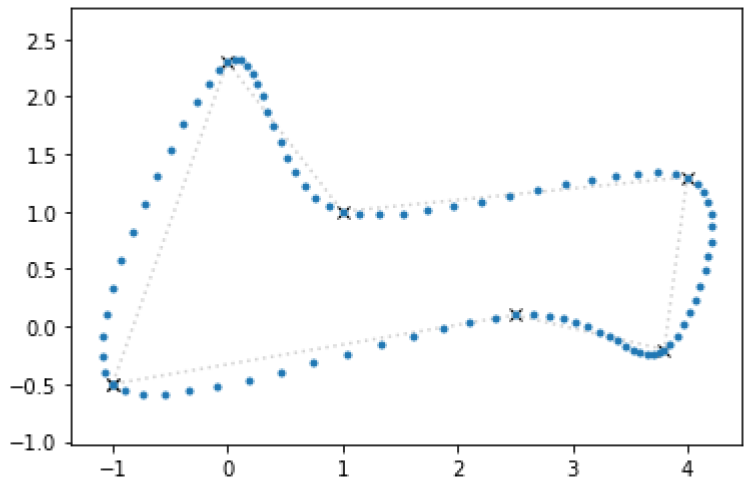
Let's choose a few points for an example:

[4]:
```python
points1 = [
    (-1, -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]:
```python
s1 = splines.CatmullRom(points1, endconditions='closed')
```

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



Tangent Vectors

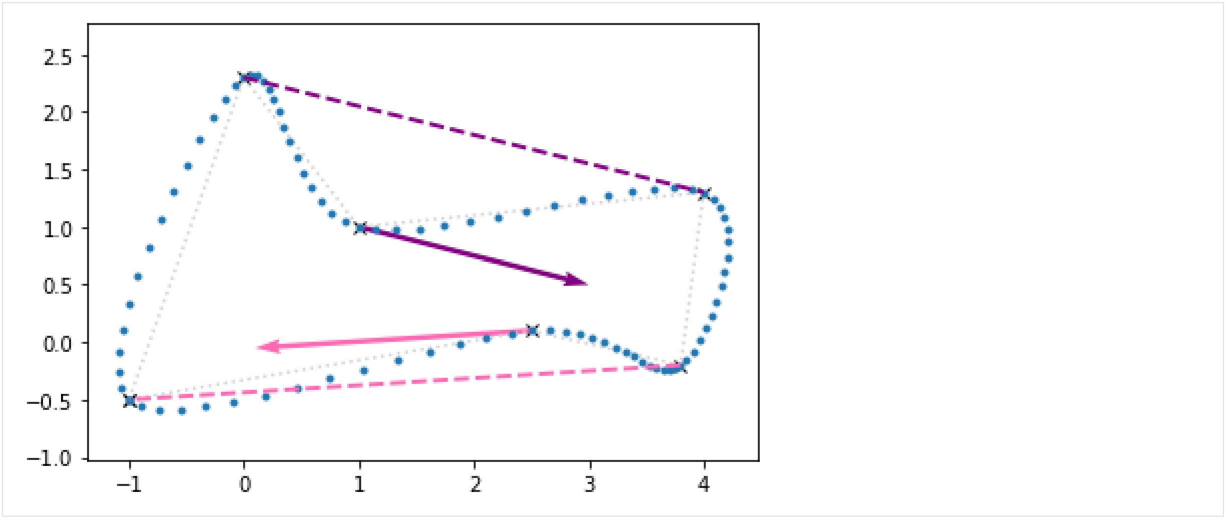
In the uniform case, the tangent vectors at any given control point are parallel to the line connecting the preceding point and the following point. The tangent vector has the same orientation as that line but only half its length. In other (more mathematical) words:

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

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

[7]:
```python
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.

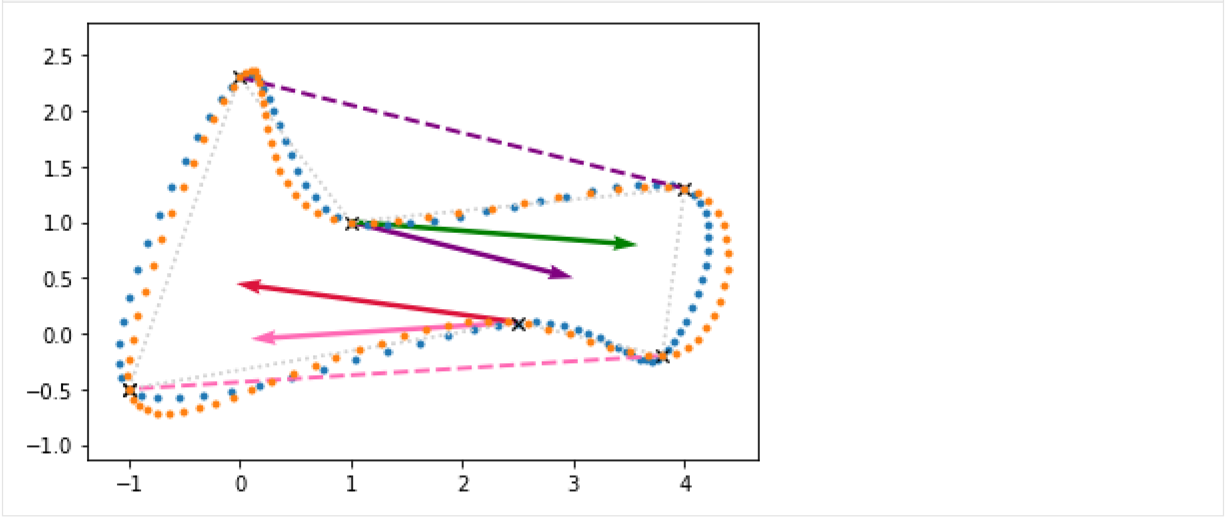
However, this will not be true anymore if we are using non-uniform time instances:

[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:

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

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

Equivalently, this can be written as:

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

Also equivalently, with $\boldsymbol{v}_i = \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_i}{t_{i+1} - t_i}$, it can be written as:

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

Some sources provide a simpler equation which is different from the tangent vector of a Catmull–Rom spline (except in the uniform case):

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

There are even sources (e.g. Wikipedia[23]) which show yet a simpler equation, which has even less to do with Catmull–Rom splines (except in the uniform case):

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

Cusps and Self-Intersections

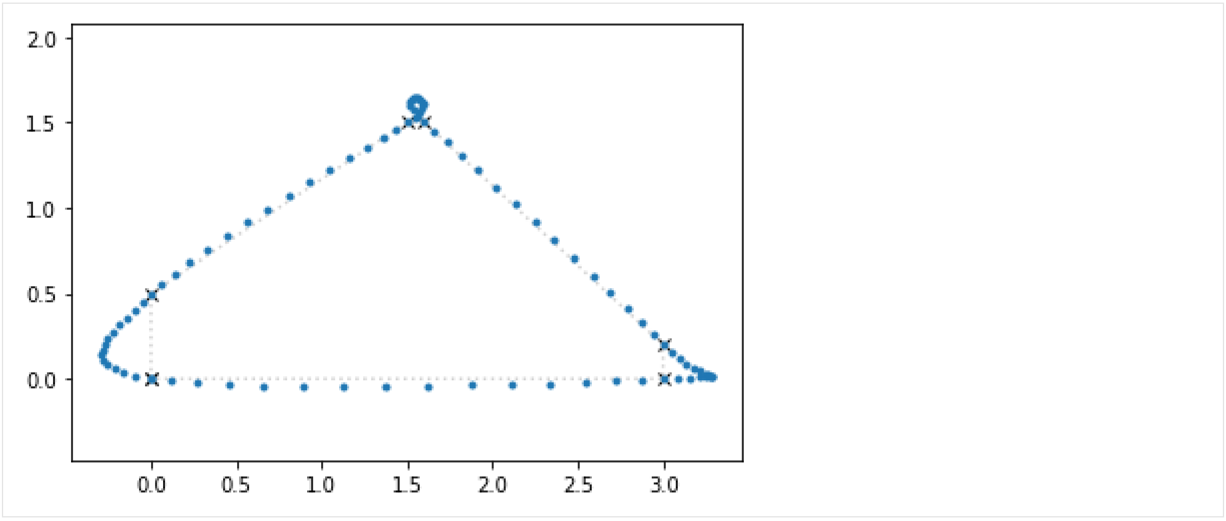
Uniform parametrization typically works very well if the (Euclidean) distances between consecutive vertices are all similar. However, if the distances are very different, the shape of the spline often turns out to be unexpected. Most notably, in extreme cases there might be even cusps or self-intersections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

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

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

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

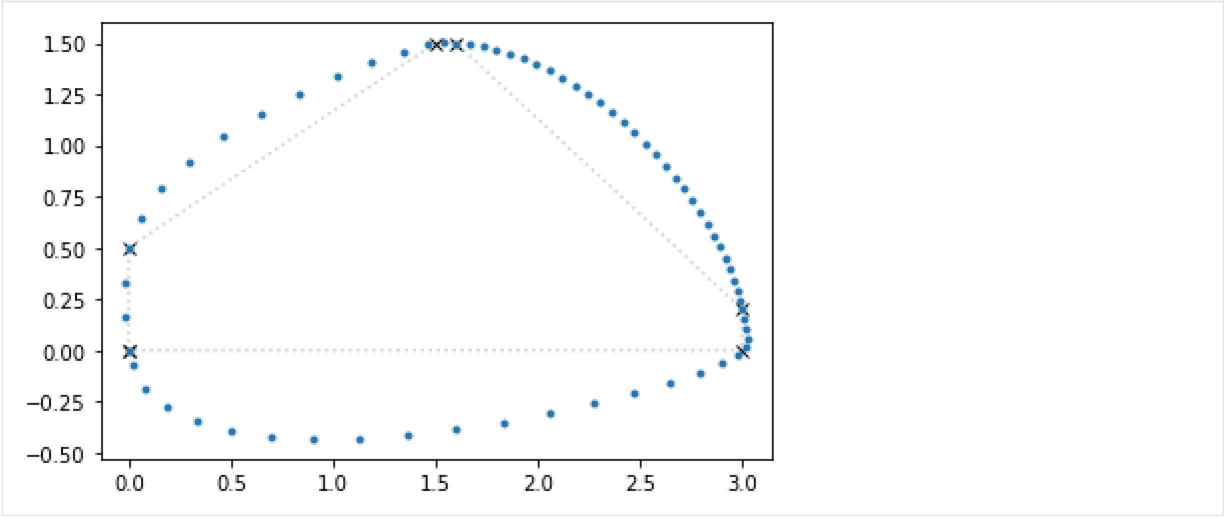
[23] https://en.wikipedia.org/wiki/Cubic_Hermite_spline#Catmull%E2%80%93Rom_spline



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:

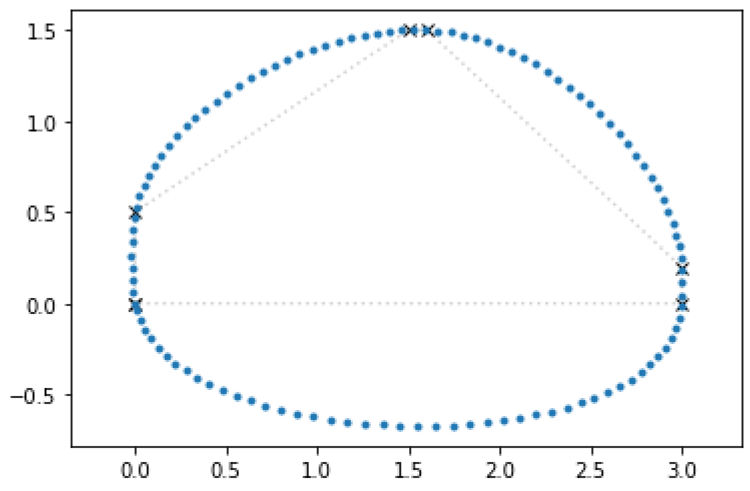
```
[16]: distances = np.linalg.norm(np.diff(points3 + points3[:1], axis=0), axis=1)
      distances
```

```
[16]: array([0.5   , 1.8028, 0.1   , 1.9105, 0.2   , 3.    ])
```

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

```
[17]: array([0.    , 0.5   , 2.3028, 2.4028, 4.3133, 4.5133, 7.5133])
```

[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.

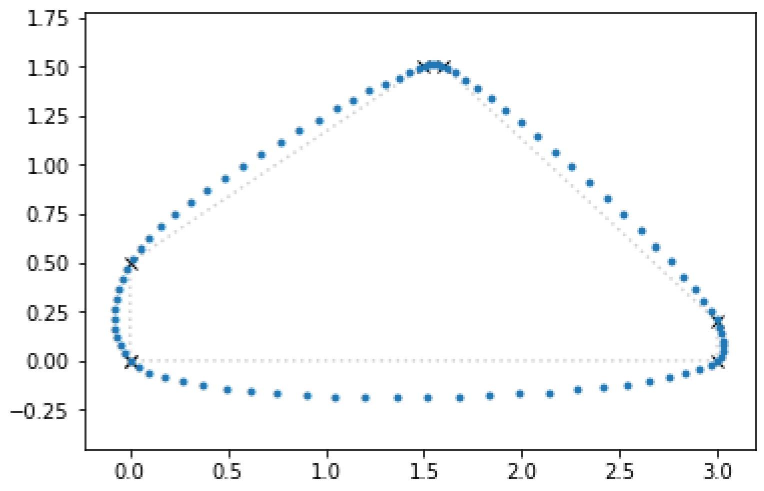
Centripetal Parameterization

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.    , 0.7071, 2.0498, 2.366 , 3.7482, 4.1954, 5.9275])
```

[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

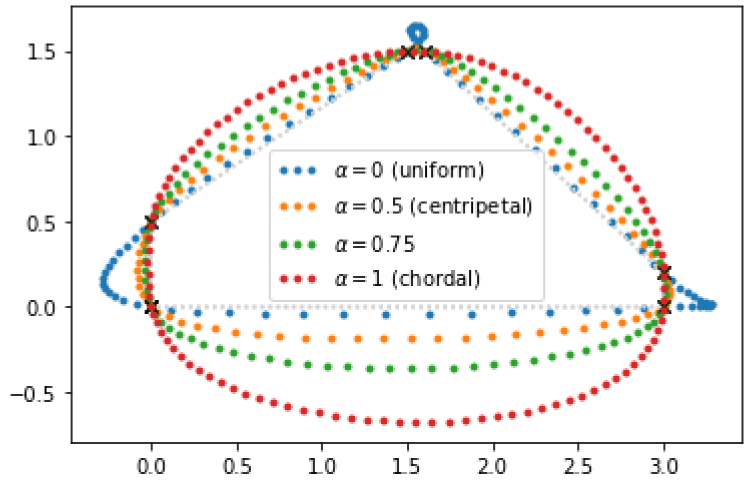
It turns out that the previous two parameterization schemes are just two special cases of a more general scheme for obtaining time intervals between control points:

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

In the Python class *splines.CatmullRom* (page 161), the parameter `alpha` can be specified.

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

[22]:
```python
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);
```



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 $w(s)$:
>
> $$C_{0,k}(s) = \sum_{i=0}^{k} \Big[\prod_{\substack{j=i-k \\ j \neq 0}}^{i} \left(\frac{s}{j} + 1 \right) \Big] w(s+i)$$
>
> 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', integer=True)
```

```
[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)
```

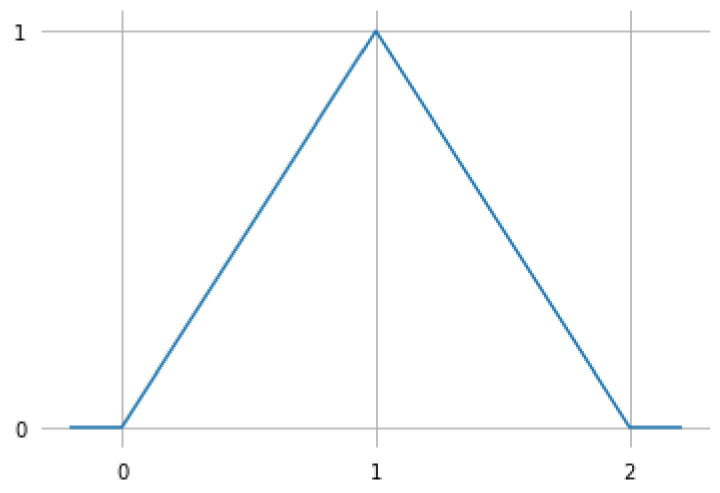
We can use one of them (where $i = 0$) as blending function:

[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])
```



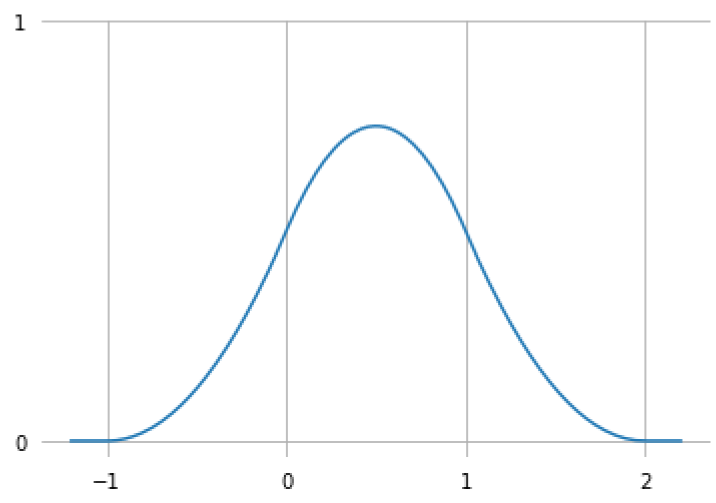
The quadratic (degree 2) basis functions can be obtained like this:

[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.

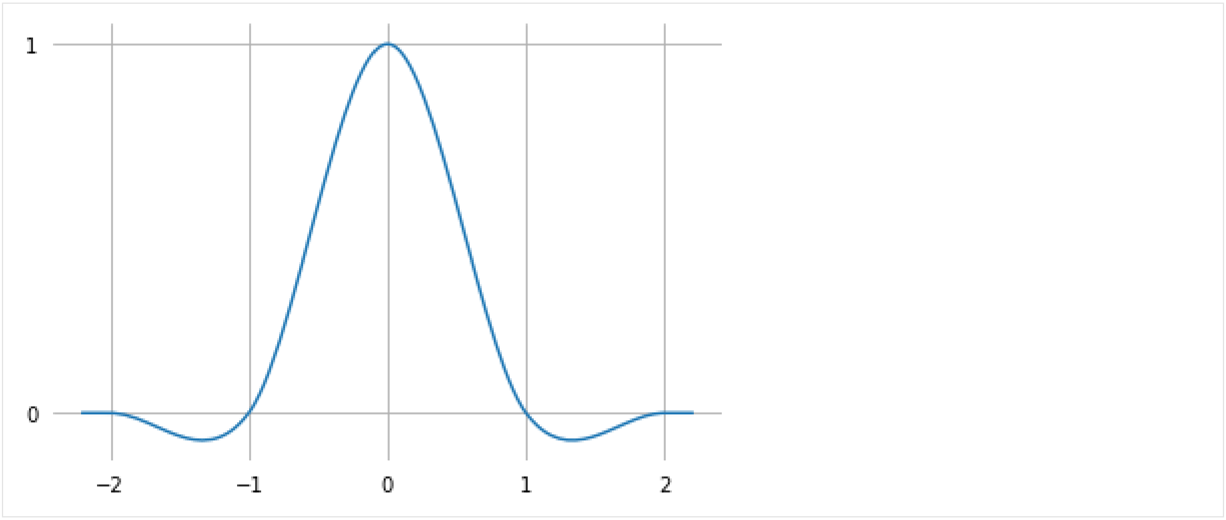
With the information so far, we can construct the cardinal function $C_{0,1}(t)$, using our *quadratic* B-spline blending function w2 (which is, as required, centered about $\frac{k}{2}$):

```
[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. The two equivalent approaches are illustrated by means of animations in the *notebook about non-uniform Catmull–Rom splines* (page 74).

Example Plot

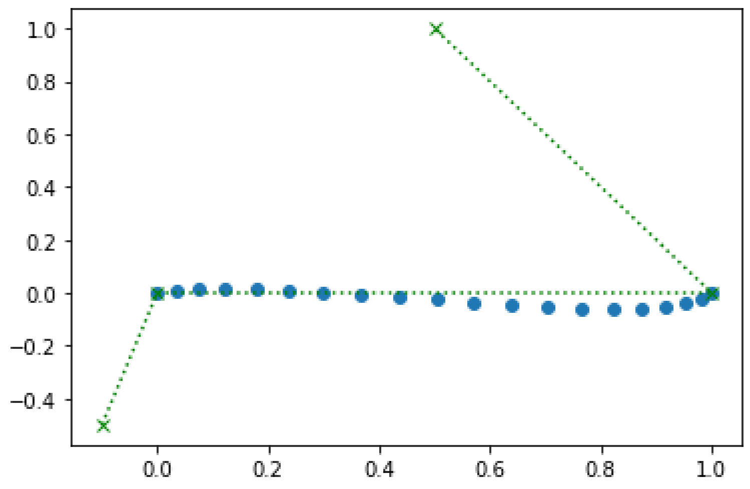
```
[18]: import matplotlib.pyplot as plt
      import numpy as np
```

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),
          (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');
```



For calculating more than one segment, and also for creating non-uniform Catmull–Rom splines, the class *splines.CatmullRom* (page 161) can be used. For more plots, see *the notebook about properties of Catmull–Rom splines* (page 55).

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.

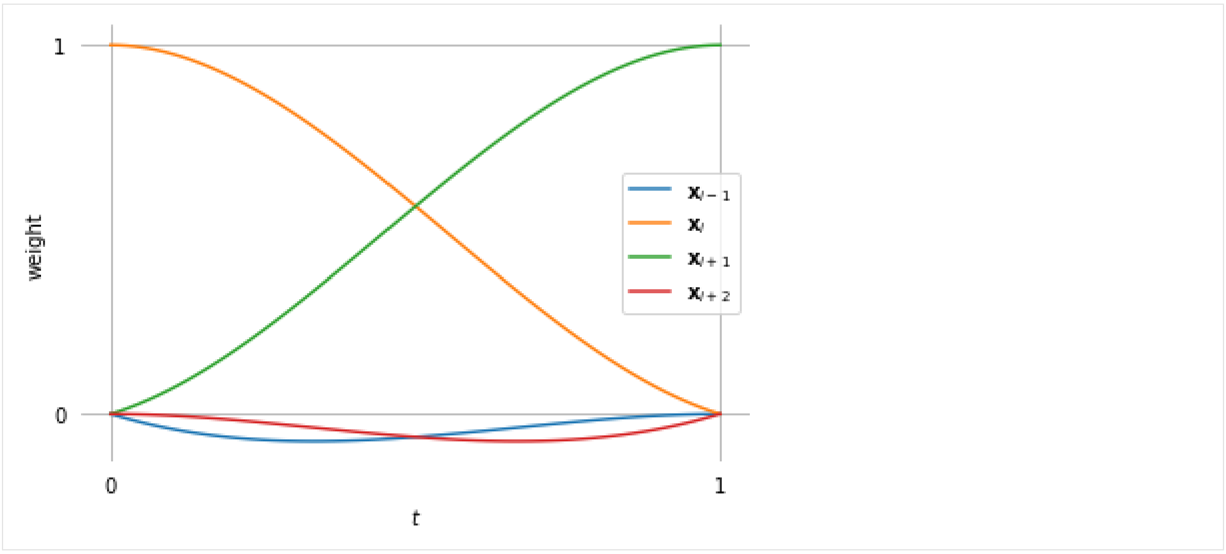
We are splitting the piecewise expression into four separate pieces, each one to be evaluated at $0 \leq t \leq 1$. We are also reversing the order of the pieces, to match our intended control point order:

```
[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]:
$$\left[\begin{matrix}- \frac{t \left(t - 1\right)^{2}}{2}\\\frac{3 t^{3}}{2} - \frac{5 t^{2}}{2} + 1\\- \frac{3 \left(t - 1\right)^{3}}{2} - \frac{5 \left(t - 1\right)^{2}}{2} + 1\\\frac{t^{2} \left(t - 1\right)}{2}\end{matrix}\right]$$

```
[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]:

$$p_4 = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} M_{\text{CR}} \begin{bmatrix} x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

From the basis polynomials and the control points, we can already calculate $p_4(t)$ …

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

[27]:

$$p_4 = t^3 \left(-\frac{x_3}{2} + \frac{3x_4}{2} - \frac{3x_5}{2} + \frac{x_6}{2}\right) + t^2 \left(x_3 - \frac{5x_4}{2} + 2x_5 - \frac{x_6}{2}\right) + t \left(-\frac{x_3}{2} + \frac{x_5}{2}\right) + x_4$$

… 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]:

$$M_{\text{CR}} = \frac{1}{2}\begin{bmatrix}-1 & 3 & -3 & 1\\ 2 & -5 & 4 & -1\\ -1 & 0 & 1 & 0\\ 0 & 2 & 0 & 0\end{bmatrix}$$

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]:

$$M_{\text{CR}}{}^{-1} = \begin{bmatrix}1 & 1 & -1 & 1\\ 0 & 0 & 0 & 1\\ 1 & 1 & 1 & 1\\ 6 & 4 & 2 & 1\end{bmatrix}$$

[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]:

$$\left.\frac{d}{dt}\boldsymbol{p}_4\right|_{t=0} = -\frac{\boldsymbol{x}_3}{2} + \frac{\boldsymbol{x}_5}{2}$$

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

$$\left.\frac{d}{dt}\boldsymbol{p}_4\right|_{t=1} = -\frac{\boldsymbol{x}_4}{2} + \frac{\boldsymbol{x}_6}{2}$$

These two expressions can be generalized to (as already shown in *the notebook about Catmull–Rom properties* (page 56)):

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

Using Bézier Segments

The above equation for the tangent vectors can be used to construct *Hermite splines* (page 13) or, after dividing them by 3, to obtain the control points for *cubic Bézier spline segments* (page 49):

$$\begin{aligned}
\tilde{x}_i^{(+)} &= x_i + \frac{\dot{x}_i}{3} = x_i + \frac{x_{i+1} - x_{i-1}}{6} \\
\tilde{x}_i^{(-)} &= x_i - \frac{\dot{x}_i}{3} = x_i - \frac{x_{i+1} - x_{i-1}}{6}
\end{aligned}$$

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

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

Non-Uniform Catmull–Rom Splines

[CR74] describes only the *uniform case* (page 62), but it is straightforward to extend the method to non-uniform splines.

The method creates three linear interpolations (and *extra*polations) between neighboring pairs of the four relevant control points and then blends 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 **??**). 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 5), it can be implemented 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
```

(continues on next page)

(continued from previous page)

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

Alternatively, sympy.interpolate()[24] 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,
```

(continues on next page)

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

(continued from previous page)

```
    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,
}
```

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 (which is expected, since the incoming and outgoing tangents are supposed to be equal):

$$\begin{aligned}\dot{\boldsymbol{x}}_i &= \frac{(t_{i+1}-t_i)^2(\boldsymbol{x}_i-\boldsymbol{x}_{i-1})+(t_i-t_{i-1})^2(\boldsymbol{x}_{i+1}-\boldsymbol{x}_i)}{(t_{i+1}-t_i)(t_i-t_{i-1})(t_{i+1}-t_{i-1})}\\ &= \frac{{\Delta_i}^2(\boldsymbol{x}_i-\boldsymbol{x}_{i-1})+{\Delta_{i-1}}^2(\boldsymbol{x}_{i+1}-\boldsymbol{x}_i)}{\Delta_i\Delta_{i-1}(\Delta_i+\Delta_{i-1})}\end{aligned}$$

Equivalently, this can be written as:

$$\begin{aligned}\dot{\boldsymbol{x}}_i &= \frac{(t_{i+1}-t_i)(\boldsymbol{x}_i-\boldsymbol{x}_{i-1})}{(t_i-t_{i-1})(t_{i+1}-t_{i-1})}+\frac{(t_i-t_{i-1})(\boldsymbol{x}_{i+1}-\boldsymbol{x}_i)}{(t_{i+1}-t_i)(t_{i+1}-t_{i-1})}\\ &= \frac{\Delta_i(\boldsymbol{x}_i-\boldsymbol{x}_{i-1})}{\Delta_{i-1}(\Delta_i+\Delta_{i-1})}+\frac{\Delta_{i-1}(\boldsymbol{x}_{i+1}-\boldsymbol{x}_i)}{\Delta_i(\Delta_i+\Delta_{i-1})}\end{aligned}$$

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]: $$\frac{\frac{(-t_3+t_4)(-x_4+x_5)}{-t_4+t_5} + \frac{(-t_4+t_5)(-x_3+x_4)}{-t_3+t_4}}{-t_3+t_5}$$

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 $\dot{\boldsymbol{x}}_4$ (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
```

Using Non-Uniform Bézier Segments

Similar to *the uniform case* (page 70), the above equation for the tangent vectors can be used to construct non-uniform *Hermite splines* (page 13) or, after multiplying them with the appropriate parameter interval and dividing them by 3, to obtain the two additional control points for *non-uniform cubic Bézier spline segments* (page 55):

$$\begin{aligned}
\tilde{\boldsymbol{x}}_i^{(+)} &= \boldsymbol{x}_i + \frac{\Delta_i \dot{\boldsymbol{x}}_i}{3} \\
&= \boldsymbol{x}_i + \frac{\Delta_i}{3} \frac{\Delta_i \boldsymbol{v}_{i-1} + \Delta_{i-1} \boldsymbol{v}_i}{\Delta_{i-1} + \Delta_i} \\
&= \boldsymbol{x}_i + \frac{{\Delta_i}^2 (\boldsymbol{x}_i - \boldsymbol{x}_{i-1})}{3\Delta_{i-1}(\Delta_i + \Delta_{i-1})} + \frac{\Delta_{i-1}(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{3(\Delta_i + \Delta_{i-1})} \\
\tilde{\boldsymbol{x}}_i^{(-)} &= \boldsymbol{x}_i - \frac{\Delta_{i-1} \dot{\boldsymbol{x}}_i}{3} \\
&= \boldsymbol{x}_i - \frac{\Delta_{i-1}}{3} \frac{\Delta_i \boldsymbol{v}_{i-1} + \Delta_{i-1} \boldsymbol{v}_i}{\Delta_{i-1} + \Delta_i} \\
&= \boldsymbol{x}_i - \frac{\Delta_i (\boldsymbol{x}_i - \boldsymbol{x}_{i-1})}{3(\Delta_i + \Delta_{i-1})} - \frac{{\Delta_{i-1}}^2 (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{3\Delta_i(\Delta_i + \Delta_{i-1})}
\end{aligned}$$

This is again using $\Delta_i = t_{i+1} - t_i$ and $\boldsymbol{v}_i = \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_i}{\Delta_i}$.

Animation

To illustrate how two quadratic Lagrange interpolations followed by linear blending might look like, we can generate an animation by means of the file catmull_rom.py:

```
[21]: from catmull_rom import animation_2_1, animation_1_2
      from IPython.display import HTML
```

```
[22]: vertices = [
          (1, 0),
          (0.5, 1),
          (6, 2),
          (5, 0),
      ]
```

```
[23]: times = [
          0,
          1,
          6,
          8,
      ]
```

```
[24]: ani_2_1 = animation_2_1(vertices, times)
```

```
[25]: HTML(ani_2_1.to_jshtml(default_mode='reflect'))
```

```
[25]: <IPython.core.display.HTML object>
```

In the beginning of this notebook we claimed that two quadratic interpolations followed by linear blending are easier to understand. To prove this, let's have a look at how three linear interpolations (and *extra*polations) followed by quadratic B-spline blending would look like:

```
[26]: ani_1_2 = animation_1_2(vertices, times)
```

```
[27]: HTML(ani_1_2.to_jshtml(default_mode='reflect'))
```

```
[27]: <IPython.core.display.HTML object>
```

Would you agree that this is less straightforward?

If you would rather replace the quadratic B-spline basis function with a bunch of linear interpolations (using De Boor's algorithm), take a look at *the notebook about the Barry–Goldman algorithm* (page 83).

.... 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 147).

[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 5) 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):

$$\begin{array}{ccccc}
 & & p_{4,5} & & \\
 & \frac{t_5-t}{t_5-t_4} & & \frac{t-t_4}{t_5-t_4} & \\
x_4 & & & & x_5
\end{array}$$

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]:
```python
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 5).

In the *quadratic* case, it looks like this:

$$\begin{array}{ccccccccc}
 & & & & p_{3,4,5} & & & & \\
 & & & \frac{t_5-t}{t_5-t_3} & & \frac{t-t_3}{t_5-t_3} & & & \\
 & & p_{3,4} & & & & p_{4,5} & & \\
 & \frac{t_4-t}{t_4-t_3} & & \frac{t-t_3}{t_4-t_3} & & \frac{t_5-t}{t_5-t_4} & & \frac{t-t_4}{t_5-t_4} & \\
x_3 & & & & x_4 & & & & x_5
\end{array}$$

The *cubic* case is shown in figure 2 of [BG88].

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

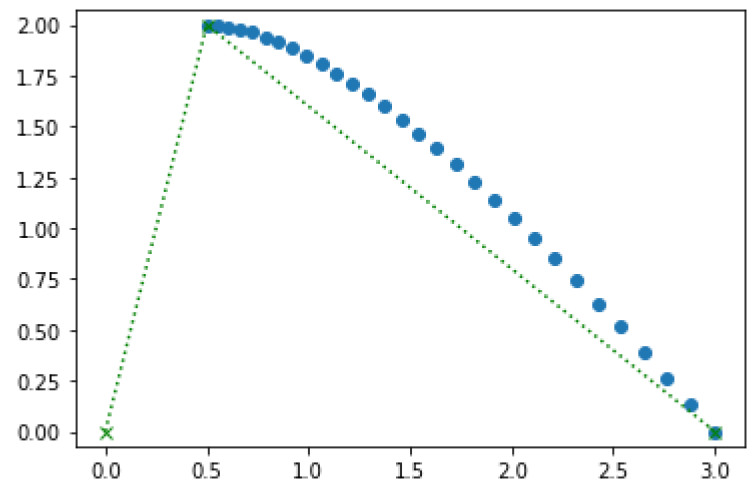
Let's try to plot this for three points:

[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');
```



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[25], see [dB72]) can be used to calculate B-spline basis functions.

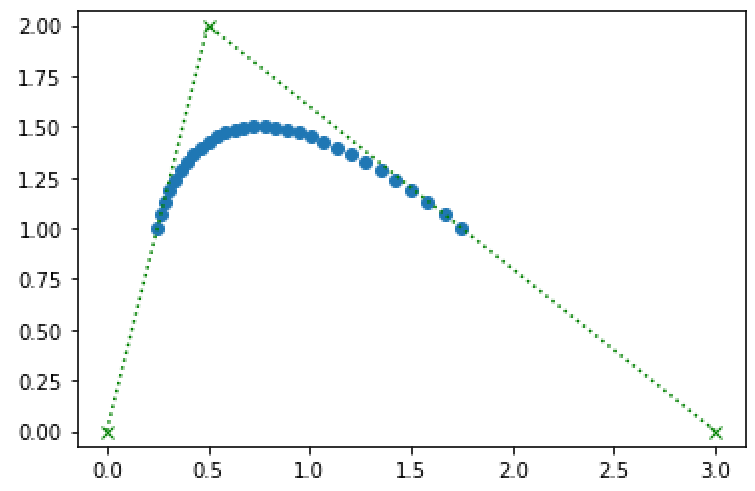
The quadratic case looks like this:

$$\begin{array}{ccccccccc} & & & & p_{3,4,5} & & & & \\ & & & \frac{t_5-t}{t_5-t_4} & & \frac{t-t_4}{t_5-t_4} & & & \\ & & p_{3,4} & & & & p_{4,5} & & \\ & \frac{t_5-t}{t_5-t_3} & & \frac{t-t_3}{t_5-t_3} & & \frac{t_6-t}{t_6-t_4} & & \frac{t-t_4}{t_6-t_4} & \\ x_3 & & & & x_4 & & & & x_5 \end{array}$$

The *cubic* case shown in figure 1 of [BG88].

[25] https://en.wikipedia.org/wiki/Carl_R._de_Boor

```
[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');
```



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:

$$
\begin{array}{ccccccccccccc}
 & & & & & & \boldsymbol{p}_{3,4,5,6} & & & & & & \\
 & & & & & \frac{t_5-t}{t_5-t_4} & & \frac{t-t_4}{t_5-t_4} & & & & & \\
 & & & & \boldsymbol{p}_{3,4,5} & & & & \boldsymbol{p}_{4,5,6} & & & & \\
 & & & \frac{t_5-t}{t_5-t_3} & & \frac{t-t_3}{t_5-t_3} & & \frac{t_6-t}{t_6-t_4} & & \frac{t-t_4}{t_6-t_4} & & & \\
 & & \boldsymbol{p}_{3,4} & & & & \boldsymbol{p}_{4,5} & & & & \boldsymbol{p}_{5,6} & & \\
 & \frac{t_4-t}{t_4-t_3} & & \frac{t-t_3}{t_4-t_3} & & \frac{t_5-t}{t_5-t_4} & & \frac{t-t_4}{t_5-t_4} & & \frac{t_6-t}{t_6-t_5} & & \frac{t-t_5}{t_6-t_5} & \\
\boldsymbol{x}_3 & & & & \boldsymbol{x}_4 & & & & \boldsymbol{x}_5 & & & & \boldsymbol{x}_6
\end{array}
$$

Here we are considering the fifth spline segment $\boldsymbol{p}_{3,4,5,6}(t)$ (represented at the apex of the triangle) from $\boldsymbol{x}_4$ to $\boldsymbol{x}_5$ (to be found at the base of the triangle) which corresponds to the parameter range $t_4 \le t \le t_5$. To calculate the values in this segment, we also need to know the preceding control point $\boldsymbol{x}_3$ (at the bottom left) and the following control point $\boldsymbol{x}_6$ (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 x_3, x_4, x_5 and x_4, x_5, x_6, respectively. Note that the interpolation of x_4 and 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 62) 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 59). 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 *interpolations* and *extrapolations*.

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]:
$$p_{4,5} = \frac{x_4\left(-t + t_5\right) + x_5\left(t - t_4\right)}{-t_4 + t_5}$$

Obviously, this starts at:

[13]:
```
p45.evaluated_at(t, t4)
```

[13]:
$$\left.p_{4,5}\right|_{t=t_4} = x_4$$

. . . and ends at:

[14]:
```
p45.evaluated_at(t, t5)
```

[14]:
$$\left.p_{4,5}\right|_{t=t_5} = x_5$$

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]:
$$p_{3,4} = \frac{x_3\left(-t + t_4\right) + x_4\left(t - t_3\right)}{-t_3 + t_4}$$

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]:

$$\left.p_{3,4}\right|_{t=t_4} = x_4$$

… and ending at some extrapolated point beyond x_4:

[17]:
```
p34.evaluated_at(t, t5)
```
[17]:

$$\left.p_{3,4}\right|_{t=t_5} = \frac{x_3 \left(t_4 - t_5\right) + x_4 \left(-t_3 + t_5\right)}{-t_3 + t_4}$$

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]:

$$p_{5,6} = \frac{x_5 \left(-t + t_6\right) + x_6 \left(t - t_5\right)}{-t_5 + t_6}$$

[19]:
```
p56.evaluated_at(t, t4)
```
[19]:

$$\left.p_{5,6}\right|_{t=t_4} = \frac{x_5 \left(-t_4 + t_6\right) + x_6 \left(t_4 - t_5\right)}{-t_5 + t_6}$$

[20]:
```
p56.evaluated_at(t, t5)
```
[20]:

$$\left.p_{5,6}\right|_{t=t_5} = x_5$$

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]:

$$p_{3,4,5} = \frac{p_{3,4} \left(-t + t_5\right) + p_{4,5} \left(t - t_3\right)}{-t_3 + t_5}$$

[22]:
```
p456 = NamedExpression('pbm_4,5,6', lerp([p45.name, p56.name], [t4, t6], t))
p456
```
[22]:

$$p_{4,5,6} = \frac{p_{4,5} \left(-t + t_6\right) + p_{5,6} \left(t - t_4\right)}{-t_4 + t_6}$$

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]:

$$\left.p_{3,4,5}\right|_{t=t_4} = \frac{\left.p_{3,4}\right|_{t=t_4} \left(-t_4 + t_5\right) + \left.p_{4,5}\right|_{t=t_4} \left(-t_3 + t_4\right)}{-t_3 + t_5}$$

[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]:
$$p_{3,4,5,6} = \frac{(t-t_4)(t_3-t_4)(t_3-t_5)\left(-(t-t_4)(t_4-t_5)(-x_5(t-t_6)+x_6(t-t_5)) + (t-t_6)(t_5-t_6)(-x_4(t-t_5)+x_5(t-t_4))\right) -}{(t_3-t_4)(t_3-t_5)(t_4 -}$$

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]:

$$p_{3,4,5,6} =$$
$$-\Delta_3 \left(-\Delta_3 - \Delta_4\right) \left(t - t_4\right) \left(\Delta_4 \left(t - t_4\right) \left(-x_5 \left(t - t_6\right) + x_6 \left(t - t_5\right)\right) - \Delta_5 \left(t - t_6\right) \left(-x_4 \left(t - t_5\right) + x_5 \left(t - t_4\right)\right)\right) + \Delta_5 (-\Delta_4 - \Delta$$
$$\Delta_3 \Delta_4^2 \Delta_5 \left(-\Delta_3 - \Delta_4\right) (-\Delta_4 - \Delta_5$$

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]: $$\frac{d}{dt} p_{3,4,5,6}\Big|_{t=t_4} = \frac{\left(t_3 - t_4\right)^2 \left(x_4 - x_5\right) + \left(t_4 - t_5\right)^2 \left(x_3 - x_4\right)}{\left(t_3 - t_4\right) \left(t_3 - t_5\right) \left(t_4 - t_5\right)}$$

[36]:
```
pd3456.evaluated_at(t, t5).simplify()
```

[36]: $$\frac{d}{dt} p_{3,4,5,6}\Big|_{t=t_5} = \frac{\left(t_4 - t_5\right)^2 \left(x_5 - x_6\right) + \left(t_5 - t_6\right)^2 \left(x_4 - x_5\right)}{\left(t_4 - t_5\right) \left(t_4 - t_6\right) \left(t_5 - t_6\right)}$$

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 an animation of the algorithm:

[37]:
```
from barry_goldman import animation
```

[38]:
```
from IPython.display import HTML
```

[39]:
```
vertices = [
    (1, 0),
    (0.5, 1),
    (6, 2),
    (5, 0),
]
```

[40]:
```
times = [
    0,
    1,
    6,
    8,
]
```

[41]:
```
ani = animation(vertices, times)
```

[42]:
```
HTML(ani.to_jshtml(default_mode='reflect'))
```

[42]:
```
<IPython.core.display.HTML object>
```

If this doesn't look very intuitive to you, you are not alone. For a different (and probably more straightforward) point of view, have a look at the *notebook about non-uniform Catmull–Rom splines* (page 74).

.... 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 is available in the *splines.KochanekBartels* (page 161) 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 55).

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

Let's import a plotting function from helper.py ...

[2]:
```
from helper import plot_spline_2d
```

... and use it to implement a bespoke plotting function to illustrate the TCB parameters:

[3]:
```
import matplotlib.pyplot as plt
import numpy as np

def plot_tcb(*tcb, ax=None):
```

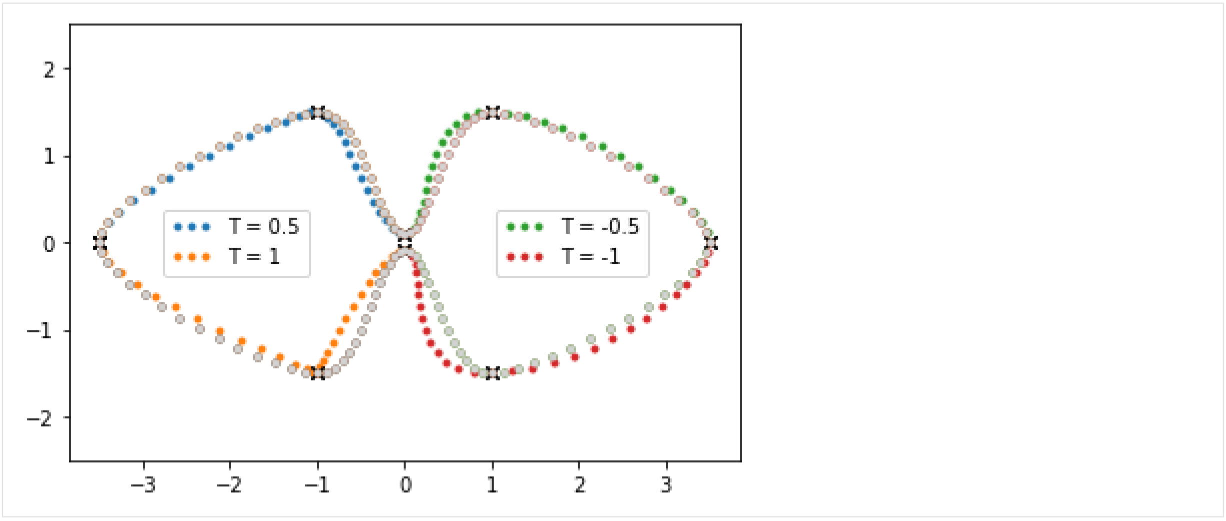
(continues on next page)

(continued from previous page)

```
    """Plot four TCB examples."""
    if ax is None:
        ax = plt.gca()
    vertices = [
        (-3.5, 0),
        (-1, 1.5),
        (0, 0.1),
        (1, 1.5),
        (3.5, 0),
        (1, -1.5),
        (0, -0.1),
        (-1, -1.5),
    ]
    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)
```

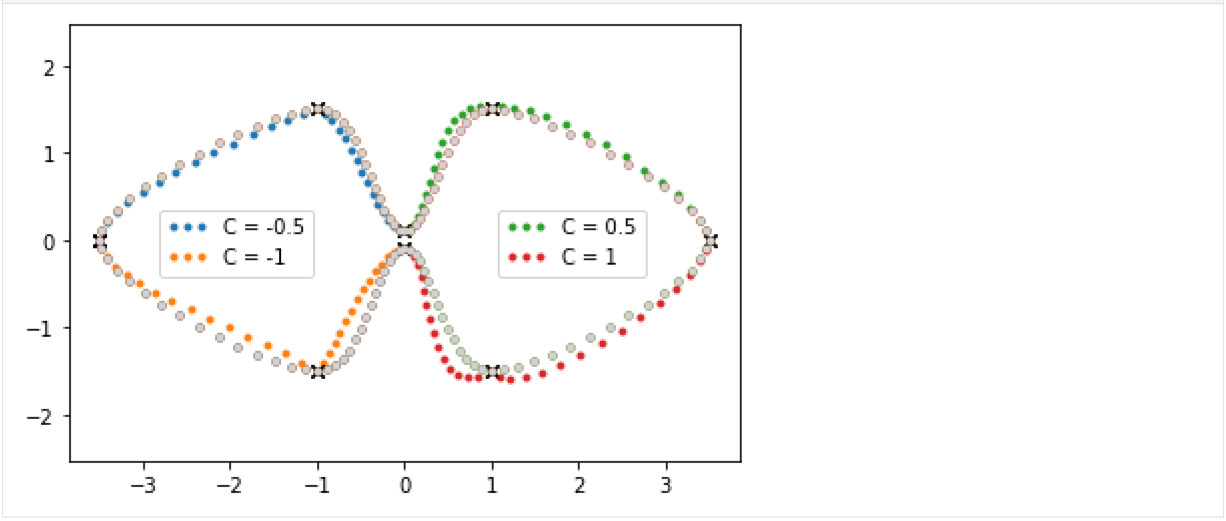
Tension

[4]:
```
plot_tcb((0.5, 0, 0), (1, 0, 0), (-0.5, 0, 0), (-1, 0, 0))
```



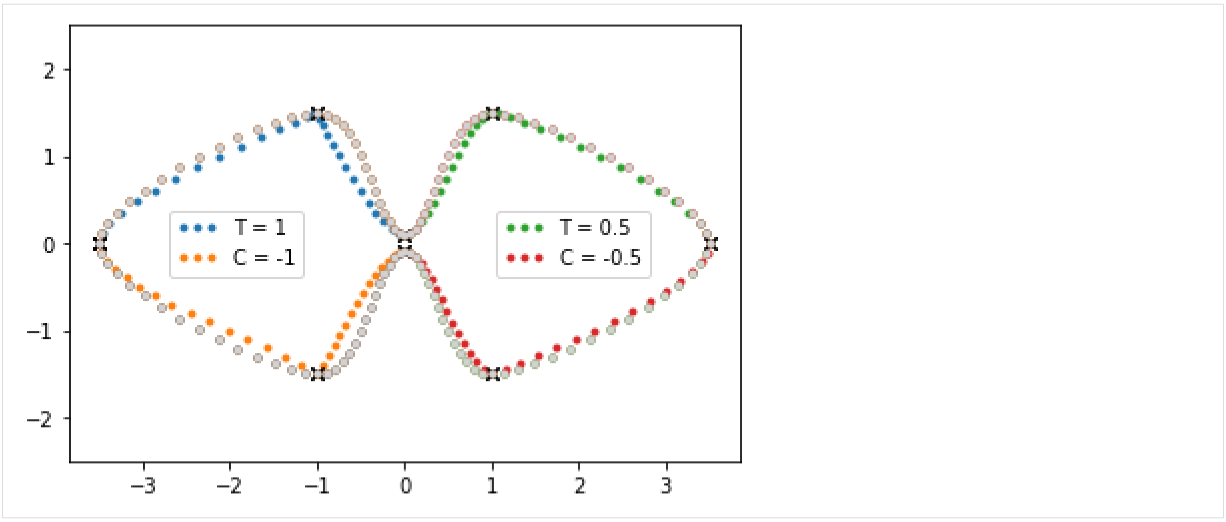
Continuity

```
[5]: plot_tcb((0, -0.5, 0), (0, -1, 0), (0, 0.5, 0), (0, 1, 0))
```



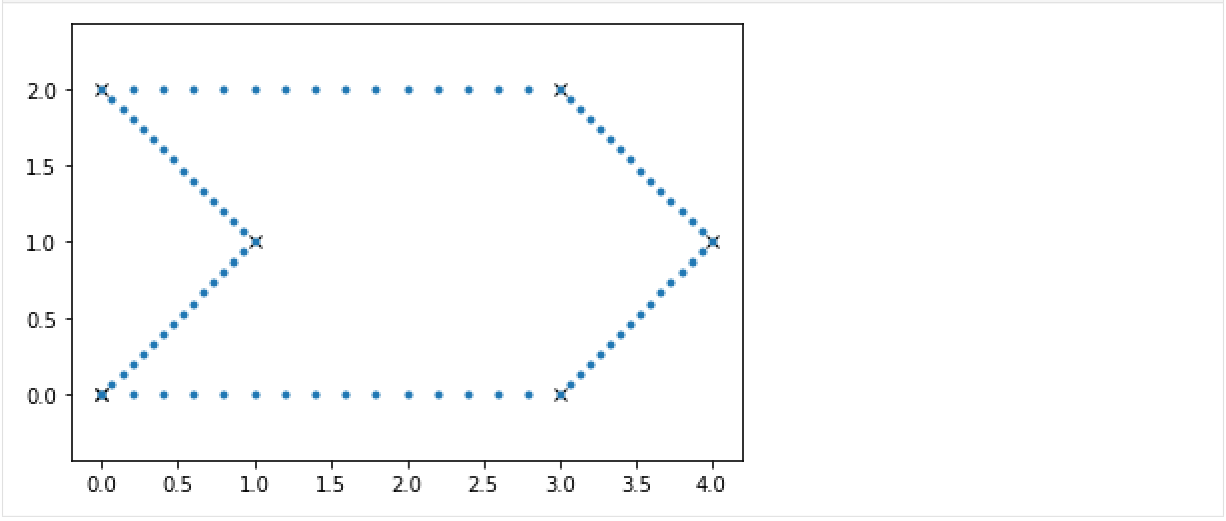
Note that the cases $T = 1$ and $C = -1$ have a very similar shape (a.k.a. "image"), but they have a different timing (and therefore different velocities):

```
[6]: plot_tcb((1, 0, 0), (0, -1, 0), (0.5, 0, 0), (0, -0.5, 0))
```



A value of $C = -1$ on adjacent vertices leads to linear segments:

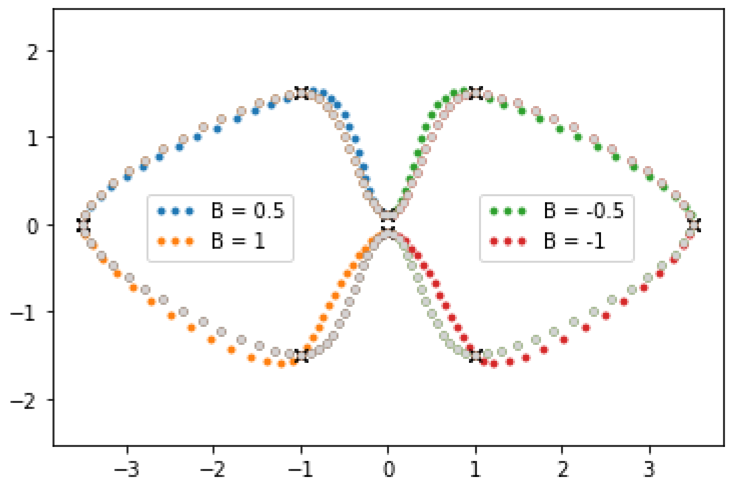
```
[7]: vertices1 = [(0, 0), (1, 1), (0, 2), (3, 2), (4, 1), (3, 0)]
     s1 = splines.KochanekBartels(vertices1, tcb=(0, -1, 0), endconditions='closed')
     plot_spline_2d(s1, chords=False)
```



Bias

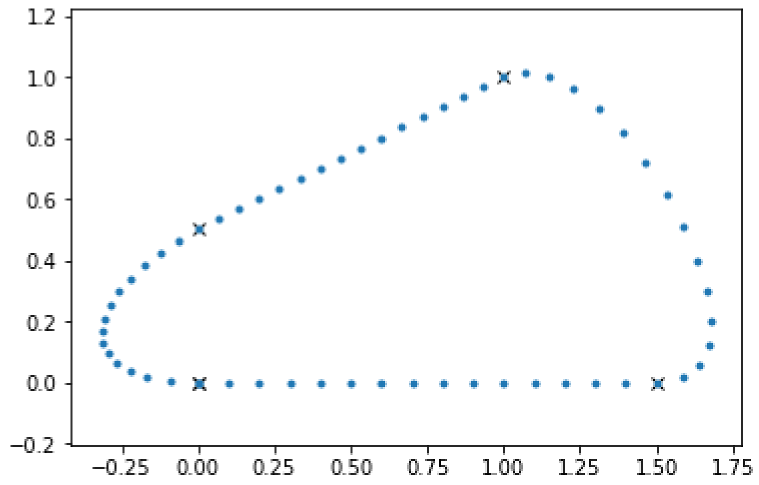
This could also be called "overshoot" (if $B > 0$) and "undershoot" (if $B < 0$):

```
[8]: plot_tcb((0, 0, 0.5), (0, 0, 1), (0, 0, -0.5), (0, 0, -1))
```



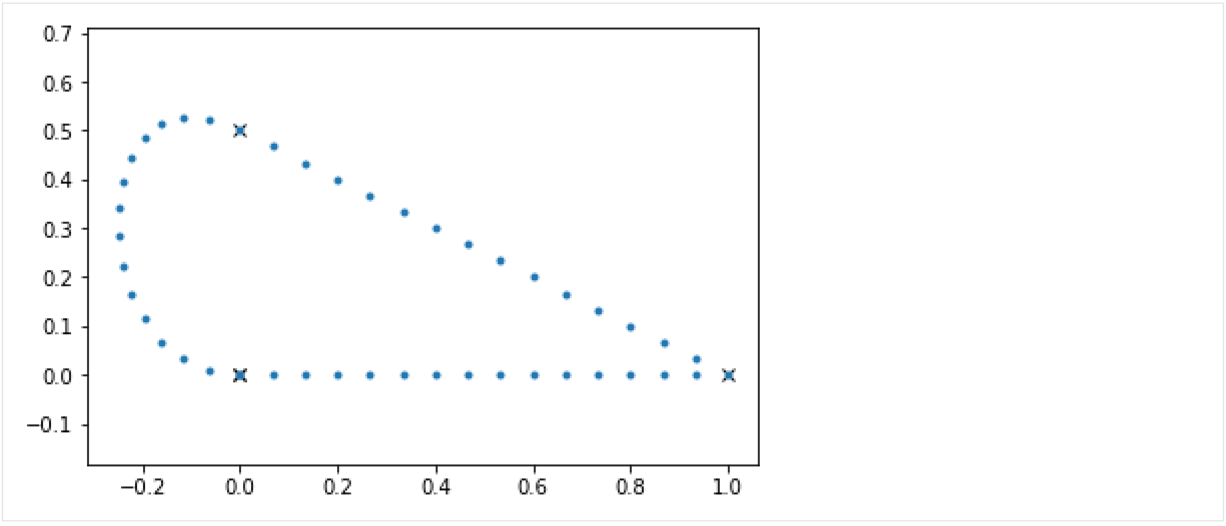
Bias -1 followed by $+1$ can be used to achieve linear segments between two control points:

```
[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)
```



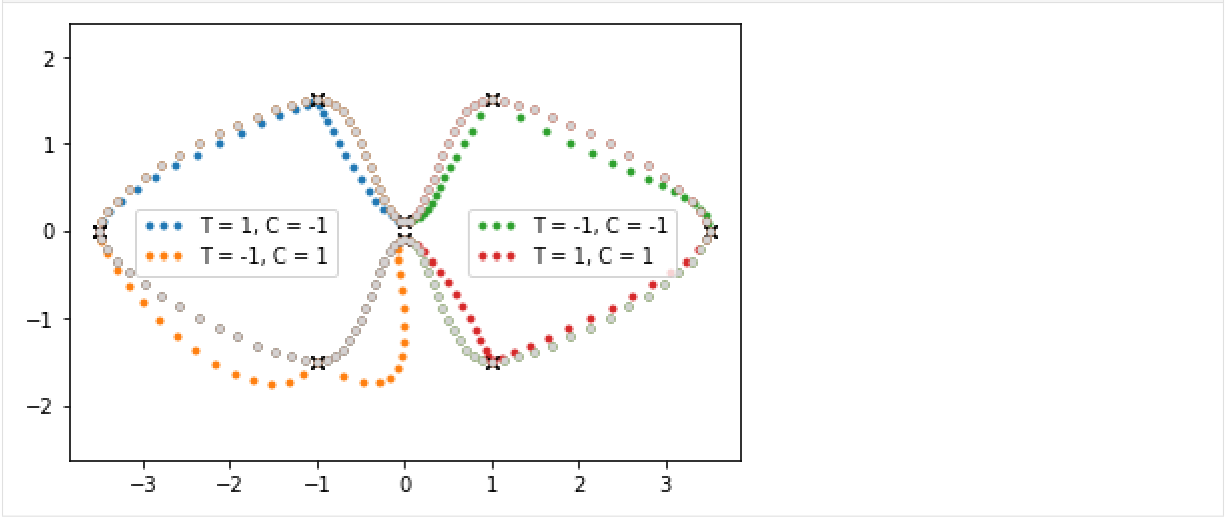
A sequence of $B = -1$, $C = -1$ and $B = +1$ can be used to get two adjacent linear segments:

```
[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)
```

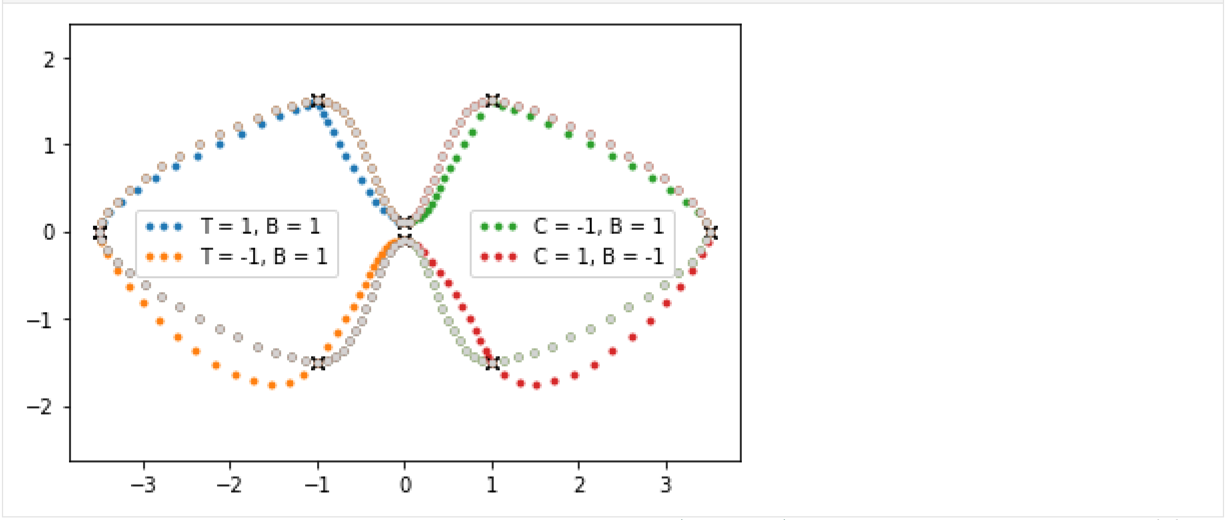


Combinations

[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))
```



doc/euclidean/kochanek-bartels-properties.ipynb ends here.

The following section was generated from `doc/euclidean/kochanek-bartels-uniform.ipynb` ....

Uniform Kochanek–Bartels Splines

As a starting point, remember the *tangent vectors of uniform Catmull–Rom splines* (page 56) (see also equation 3 of [KB84]):

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

which can be re-written as

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

Parameters

TCB splines are all about inserting the parameters T, C and B into this equation.

Tension

The usage of T is shown in equation 4 of [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 C 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]):

$$\begin{aligned}
\dot{\boldsymbol{x}}_i^{(-)} &= \frac{(1 - C_i)(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (1 + C_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2}\\
\dot{\boldsymbol{x}}_i^{(+)} &= \frac{(1 + C_i)(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (1 - C_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2}
\end{aligned}$$

Bias

The usage of B is shown in equation 7 of [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

To get the tangent vectors of a TCB spline, the three equations can be combined (see equations 8 and 9 in [KB84]):

$$
\begin{aligned}
\dot{\boldsymbol{x}}_i^{(+)} &= \frac{(1-T_i)(1+C_i)(1+B_i)(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (1-T_i)(1-C_i)(1-B_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2} \\
\dot{\boldsymbol{x}}_i^{(-)} &= \frac{(1-T_i)(1-C_i)(1+B_i)(\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) + (1-T_i)(1+C_i)(1-B_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{2}
\end{aligned}
$$

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 results we will get later, we introduce the following shorthands (as suggested in [Mil]):

$$
\begin{aligned}
a_i &= (1-T_i)(1+C_i)(1+B_i), \\
b_i &= (1-T_i)(1-C_i)(1-B_i), \\
c_i &= (1-T_i)(1-C_i)(1+B_i), \\
d_i &= (1-T_i)(1+C_i)(1-B_i),
\end{aligned}
$$

which lead 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

The tangent vectors are sufficient to implement Kochanek–Bartels splines (via *Hermite splines* (page 13)). In the rest of this notebook we are deriving the basis matrix and the basis polynomials for comparison with other spline types.

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

As in previous notebooks, we are using some SymPy helper classes from utility.py:

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

And again, we are looking at the fifth spline segment from x_4 to x_5 (which can easily be generalized to arbitrary segments).

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

```
[4]: control_values_KB = sp.Matrix([x3, x4, x5, x6])
     control_values_KB
```

[4]:

$$\begin{bmatrix} x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

We need three additional parameters per vertex: T, C and B. In our calculation, however, only the parameters belonging to x_4 and x_5 are relevant:

```
[5]: T4, T5 = sp.symbols('T4 T5')
     C4, C5 = sp.symbols('C4 C5')
     B4, B5 = sp.symbols('B4 B5')
```

Using the shorthands mentioned above . . .

```
[6]: 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)
```

$a_4 = (1 - T_4)(B_4 + 1)(C_4 + 1)$

$b_4 = (1 - B_4)(1 - C_4)(1 - T_4)$

$c_5 = (1 - C_5)(1 - T_5)(B_5 + 1)$

$d_5 = (1 - B_5)(1 - T_5)(C_5 + 1)$

. . . we can define the tangent vectors:

```
[7]: 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)
```

$$\dot{x}_4^{(+)} = \frac{a_4(-x_3 + x_4)}{2} + \frac{b_4(-x_4 + x_5)}{2}$$

$$\dot{x}_5^{(-)} = \frac{c_5(-x_4 + x_5)}{2} + \frac{d_5(-x_5 + x_6)}{2}$$

```
[8]: display(xd4.subs_symbols(a4, b4))
     display(xd5.subs_symbols(c5, d5))
```

$$\dot{x}_4^{(+)} = \frac{(1 - B_4)(1 - C_4)(1 - T_4)(-x_4 + x_5)}{2} + \frac{(1 - T_4)(B_4 + 1)(C_4 + 1)(-x_3 + x_4)}{2}$$

$$\dot{x}_5^{(-)} = \frac{(1 - B_5)(1 - T_5)(C_5 + 1)(-x_5 + x_6)}{2} + \frac{(1 - C_5)(1 - T_5)(B_5 + 1)(-x_4 + x_5)}{2}$$

Basis Matrix

We try to find a transformation from the control values defined above to *Hermite control values*:

[9]:
```
control_values_H = sp.Matrix([x4, x5, xd4.name, xd5.name])
M_KBtoH = NamedMatrix(r'{M_{\text{KB$,4\to$H}}}', 4, 4)
NamedMatrix(control_values_H, M_KBtoH.name * control_values_KB)
```

[9]:

$$\begin{bmatrix}x_4\\x_5\\\dot{x}_4^{(+)}\\\dot{x}_5^{(-)}\end{bmatrix} = M_{\text{KB},4\to\text{H}}\begin{bmatrix}x_3\\x_4\\x_5\\x_6\end{bmatrix}$$

If we substitute the above definitions of $\dot{x}_4$ and $\dot{x}_5$, we can obtain the matrix elements:

[10]:
```
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)
```

[10]:

$$M_{\text{KB},4\to\text{H}} = \frac{1}{2}\begin{bmatrix}0 & 2 & 0 & 0\\0 & 0 & 2 & 0\\-a_4 & a_4-b_4 & b_4 & 0\\0 & -c_5 & c_5-d_5 & d_5\end{bmatrix}$$

Once we have a way to get Hermite control values, we can use the Hermite basis matrix from the *notebook about uniform cubic Hermite splines* (page 17) . . .

[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]:

$$M_\text{H} = \begin{bmatrix}2 & -2 & 1 & 1\\-3 & 3 & -2 & -1\\0 & 0 & 1 & 0\\1 & 0 & 0 & 0\end{bmatrix}$$

. . . to calculate the basis matrix for Kochanek–Bartels splines:

[12]:
```
M_KB = NamedMatrix(r'{M_{\text{KB},4}}', M_H.name * M_KBtoH.name)
M_KB
```

[12]: $M_{\text{KB},4} = M_\text{H} M_{\text{KB},4\to\text{H}}$

[13]:
```
M_KB = M_KB.subs_symbols(M_H, M_KBtoH).doit()
M_KB.pull_out(sp.S.Half)
```

[13]:

$$M_{\text{KB},4} = \frac{1}{2}\begin{bmatrix}-a_4 & a_4-b_4-c_5+4 & b_4+c_5-d_5-4 & d_5\\2a_4 & -2a_4+2b_4+c_5-6 & -2b_4-c_5+d_5+6 & -d_5\\-a_4 & a_4-b_4 & b_4 & 0\\0 & 2 & 0 & 0\end{bmatrix}$$

And for completeness' sake, its inverse looks like this:

[14]:
```
M_KB.I
```

[14]:

$$M_{\text{KB},4}{}^{-1} = \begin{bmatrix} \frac{b_4}{a_4} & \frac{b_4}{a_4} & \frac{b_4-2}{a_4} & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ \frac{-c_5+d_5+6}{d_5} & \frac{-c_5+d_5+4}{d_5} & \frac{-c_5+d_5+2}{d_5} & 1 \end{bmatrix}$$

Basis Polynomials

[15]:
```
t = sp.symbols('t')
```

Multiplication with the *monomial basis* (page 2) leads to the basis functions:

[16]:
```
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)
```

[16]:

$$b_{\text{KB},4}{}^{T} = \frac{1}{2}\begin{bmatrix} a_4 t\left(-t^2+2t-1\right) \\ t^3\left(a_4-b_4-c_5+4\right)+t^2\left(-2a_4+2b_4+c_5-6\right)+t\left(a_4-b_4\right)+2 \\ t\left(b_4+t^2\left(b_4+c_5-d_5-4\right)+t\left(-2b_4-c_5+d_5+6\right)\right) \\ d_5 t^2\left(t-1\right) \end{bmatrix}$$

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):

[17]:
```
b_KB = b_KB.subs_symbols(a4, b4, c5, d5).simplify()
b_KB.T.pull_out(sp.S.Half)
```

[17]: $b_{\text{KB},4}{}^{T} =$

$$\frac{1}{2}\left[\begin{matrix} t\left(B_4+1\right)\left(C_4 \\ t^3\left(\left(B_4-1\right)\left(C_4-1\right)\left(T_4-1\right)-\left(B_4+1\right)\left(C_4+1\right)\left(T_4-1\right)-\left(B_5+1\right)\left(C_5-1\right)\left(T_5-1\right)+4\right)+t^2\left(-2\left(B_4-1\right)\left(C_4-1\right.\right. \\ -t\left(t^2\left(\left(B_4-1\right)\left(C_4-1\right)\left(T_4-1\right)+\left(B_5-1\right)\left(C_5+1\right)\left(T_5-1\right)-\left(B_5+1\right)\left(C_5-1\right)\left(T_5-1\right)+4\right)-t\left(2\left(B\right.\right.\right. \\ t^2\left(B_5-1\right)\left(\right. \end{matrix}\right.$$

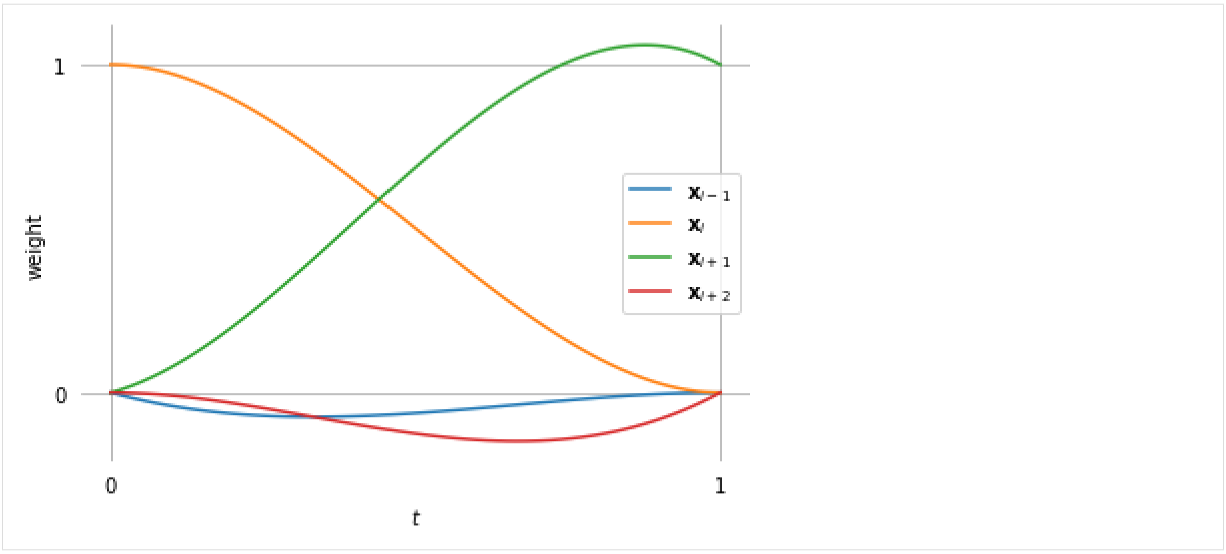
Let's use a helper function from helper.py:

[18]:
```
from helper import plot_basis
```

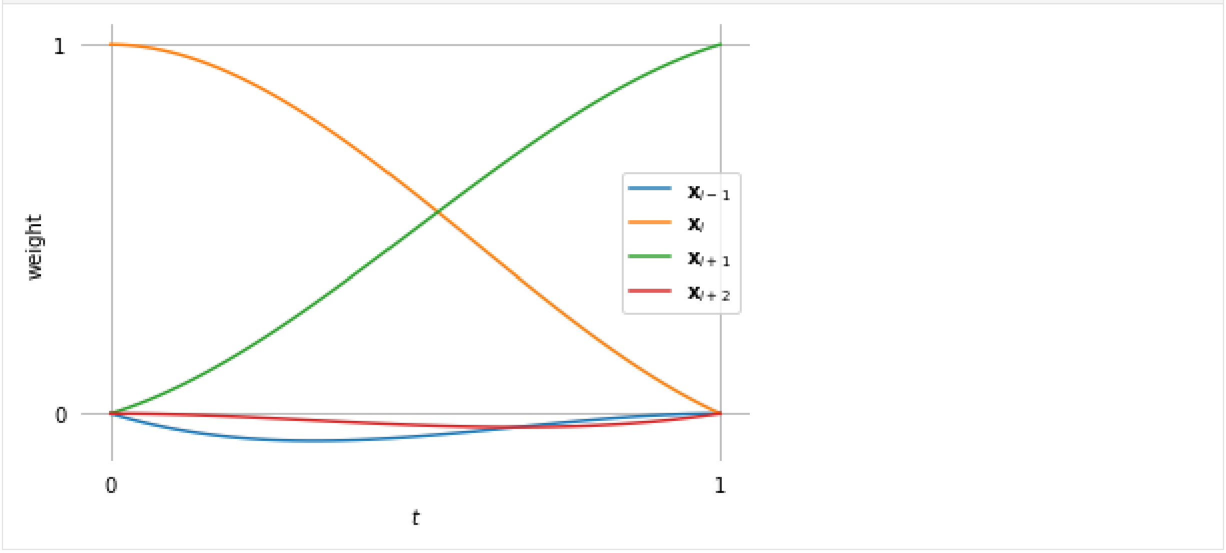
[19]:
```
labels = sp.symbols('xbm_i-1 xbm_i xbm_i+1 xbm_i+2')
```

To be able to plot the basis functions, we have to choose some concrete TCB values.

[20]:
```
plot_basis(
    *b_KB.expr.subs({T4: 0, T5: 0, C4: 0, C5: 1, B4: 0, B5: 0}),
    labels=labels)
```



[21]:
```
plot_basis(
    *b_KB.expr.subs({T4: 0, T5: 0, C4: 0, C5: -0.5, B4: 0, B5: 0}),
    labels=labels)
```



Setting all TCB values to zero leads to the *basis polynomials of uniform Catmull–Rom splines* (page 67).

.... 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" – the authors briefly mention the non-uniform case and provide equations for "adjusted tangent vectors":

> The formulas [...] assume an equal time spacing of key frames, implying an equal number of inbetweens within each key interval. A problem can exist if the animator requests a different number of inbetweens for adjacent intervals. [...] If the same parametric derivative is used for both splines at P_i, these different step sizes will cause a discontinuity in the speed of motion. What is required, if this discontinuity is not intentional, is a means of making a local adjustment to the interval separating successive frames before and after the key frame so that the speed of entry matches the speed of exit. This can be accomplished by adjusting the specification of the tangent vector at the key frame based on the number

of inbetweens in the adjacent intervals. [. . .] Once the tangent vectors have been found for an equal number of inbetweens in the adjacent intervals, the adjustment required for different numbers of inbetweens (N_{i-1} frames between P_{i-1} and P_i followed by N_i frames between P_i and P_{i+1}) can be made by weighting the tangent vectors appropriately:

$$\text{adjusted } DD_i = DD_i \frac{2N_{i-1}}{N_{i-1} + N_i}$$
$$\text{adjusted } DS_i = DS_i \frac{2N_i}{N_{i-1} + N_i}$$

—[KB84], section 4, "Adjustments for Parameter Step Size"

In their notation, DS_i is the *source derivative* (i.e. the *incoming* tangent vector) at point P_i, and DD_i is the *destination derivative* (i.e. the *outgoing* tangent vector). The point P_i corresponds to $\boldsymbol{x}_i$ in our notation.

To be able to play around with that, let's implement it in a function. It turns out that for the way we will be using this function, we have to use the reciprocal value of the adjustment mentioned in the paper:

[1]:
```python
def kochanek_bartels_tangents(xs, ns):
    """Adjusted tangent vectors according to Kochanek & Bartels."""
    x_1, _, x1 = xs
    N_1, N0 = ns
    uniform = (x1 - x_1) / 2
    # NB: the K&B paper uses reciprocal weighting factors:
    incoming = uniform * (N_1 + N0) / (2 * N0)
    outgoing = uniform * (N_1 + N0) / (2 * N_1)
    return incoming, outgoing
```

We can see that the uniform tangents are re-scaled but their direction is unchanged.

This is a hint that – although the paper claims to be using Catmull–Rom splines – we'll get different results than in the *notebooks about Catmull–Rom splines* (page 56).

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

We'll need the Hermite basis matrix that we derived in the *notebook about uniform Hermite splines* (page 17) and which is also shown in equation 2 of [KB84]:

[3]:
```python
hermite_matrix = np.array([
    [ 2, -2,  1,  1],
    [-3,  3, -2, -1],
    [ 0,  0,  1,  0],
    [ 1,  0,  0,  0]])
```

Since the paper uses a different (implicit) re-scaling of parameter values (based on the numbers of *inbetweens*), we cannot use the classes from the *splines* (page 159) module and have to re-implement everything from scratch:

[4]:
```python
def pseudo_catmull_rom(xs, ns):
    """Closed Catmull-Rom spline according to Kochanek & Bartels."""
    xs = np.asarray(xs)
    L = len(xs)
    assert L >= 2
    assert L == len(ns)
```

(continues on next page)

(continued from previous page)

```
    tangents = [
        tangent
        for i in range(L)
        for tangent in kochanek_bartels_tangents(
            [xs[i], xs[(i + 1) % L], xs[(i + 2) % L]],
            [ns[i], ns[(i + 1) % L]])
    ]
    # Move last (outgoing) tangent to the beginning:
    tangents = tangents[-1:] + tangents[:-1]
    ts = [
        np.linspace(0, 1, n + 1, endpoint=False).reshape(-1, 1)
        for n in ns]
    return np.concatenate([
        t**[3, 2, 1, 0] @ hermite_matrix @ [xs[i], xs[(i + 1) % L], v0, v1]
        for i, (t, v0, v1)
        in enumerate(zip(ts, tangents[::2], tangents[1::2]))])
```

Note

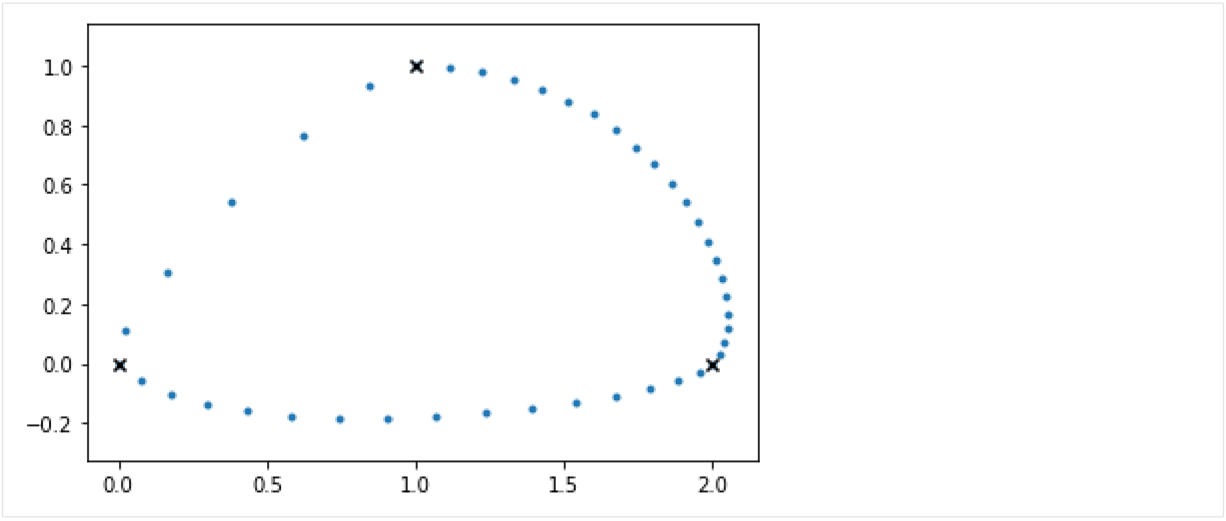
The `@` operator is used here to do NumPy's matrix multiplication[26].

Let's plot an example:

[5]:
```
vertices1 = [
    (0, 0),
    (1, 1),
    (2, 0),
]
inbetweens1 = [
    5,
    20,
    15,
]
```

[6]:
```
plt.scatter(*pseudo_catmull_rom(vertices1, inbetweens1).T, marker='.')
plt.scatter(*np.array(vertices1).T, marker='x', color='k')
plt.axis('equal');
```

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



This doesn't look too bad, let's plot the same thing with the *splines.CatmullRom* (page 161) class for comparison.

```
[7]: from splines import CatmullRom
```

In oder to be able to compare the results, we have to convert the discrete numbers of *inbetweens* into re-scaled parameter values:

```
[8]: def inbetweens2times(inbetweens):
         return np.cumsum([0, *(n + 1 for n in inbetweens)])
```

```
[9]: times1 = inbetweens2times(inbetweens1)
```

Now we have everything to create a non-uniform Catmull–Rom spline . . .

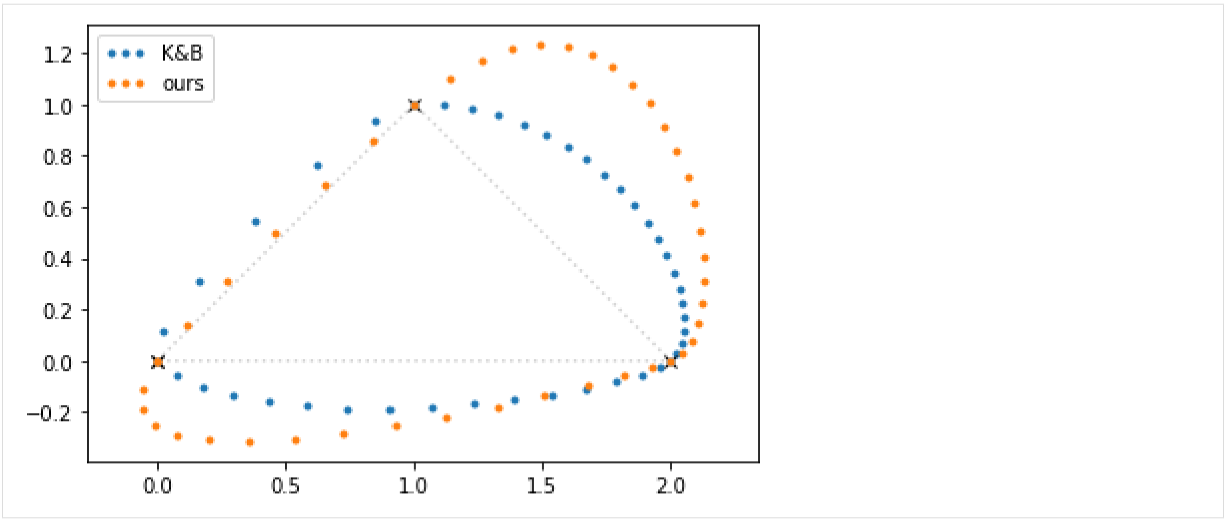
```
[10]: cr_spline1 = CatmullRom(vertices1, times1, endconditions='closed')
```

. . . and with a helper function from helper.py . . .

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

. . . we can plot it for direct comparison with the one suggested by Kochanek and Bartels:

```
[12]: plt.plot(
          *pseudo_catmull_rom(vertices1, inbetweens1).T,
          marker='.', linestyle='', label='K&B')
      plot_spline_2d(cr_spline1, dots_per_second=1, label='ours')
      plt.legend(numpoints=3);
```



Here we can clearly see that not only the lengths of the tangent vectors but also their directions have been adjusted according to the neighboring parameter intervals.

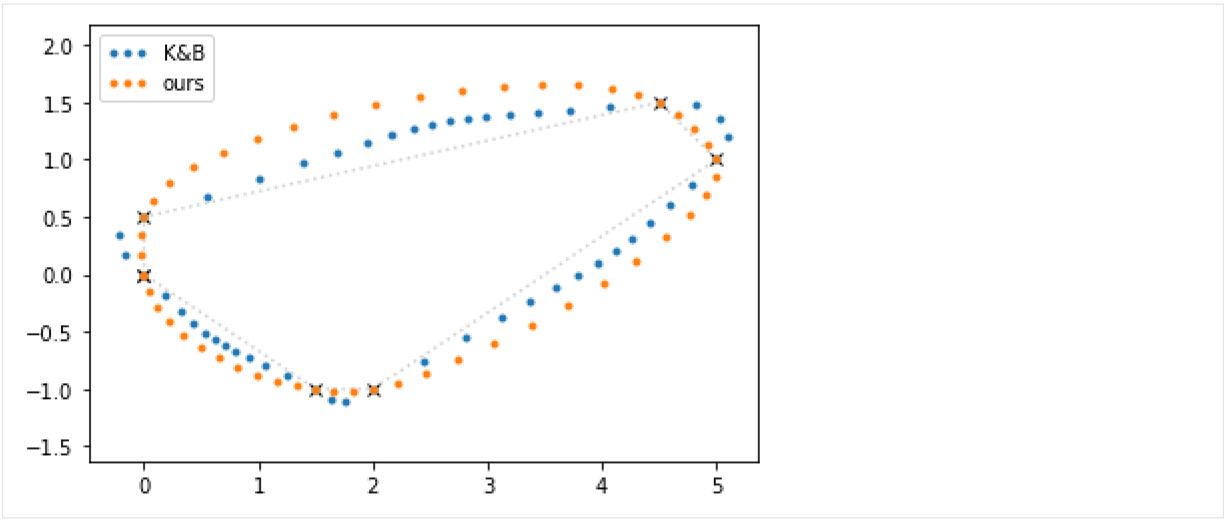
Let's look at a different example:

[13]:
```
vertices2 = [
    (0, 0),
    (0, 0.5),
    (4.5, 1.5),
    (5, 1),
    (2, -1),
    (1.5, -1),
]
inbetweens2 = [
    2,
    15,
    3,
    12,
    2,
    10,
]
```

[14]:
```
times2 = inbetweens2times(inbetweens2)
```

[15]:
```
cr_spline2 = CatmullRom(vertices2, times2, endconditions='closed')
```

[16]:
```
plt.plot(
    *pseudo_catmull_rom(vertices2, inbetweens2).T,
    marker='.', linestyle='', label='K&B')
plot_spline_2d(cr_spline2, dots_per_second=1, label='ours')
plt.legend(numpoints=3);
```



This should illustrate the shortcomings of the tangent vectors suggested by Kochanek and Bartels.

Instead of sticking with their suggestion, we use the correct expression for *tangent vectors of non-uniform Catmull–Rom splines* (page 72):

$$\dot{\boldsymbol{x}}_{i,\text{Catmull–Rom}} = \frac{(t_{i+1} - t_i)\,\boldsymbol{v}_{i-1} + (t_i - t_{i-1})\,\boldsymbol{v}_i}{t_{i+1} - t_{i-1}},$$

where $\boldsymbol{v}_i = \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_i}{t_{i+1} - t_i}$.

To this equation, we can simply add the TCB parameters like we did in the *notebook about uniform Kochanek–Bartels splines* (page 91), leading to the following equations for the incoming tangent $\dot{\boldsymbol{x}}_i^{(-)}$ and the outgoing tangent $\dot{\boldsymbol{x}}_i^{(+)}$ at vertex $\boldsymbol{x}_i$:

$$\begin{aligned}
a_i &= (1 - T_i)(1 + C_i)(1 + B_i)\\
b_i &= (1 - T_i)(1 - C_i)(1 - B_i)\\
c_i &= (1 - T_i)(1 - C_i)(1 + B_i)\\
d_i &= (1 - T_i)(1 + C_i)(1 - B_i)
\end{aligned}$$

$$\begin{aligned}
\dot{\boldsymbol{x}}_i^{(+)} &= \frac{a_i(t_{i+1} - t_i)\,\boldsymbol{v}_{i-1} + b_i(t_i - t_{i-1})\,\boldsymbol{v}_i}{t_{i+1} - t_{i-1}}\\
\dot{\boldsymbol{x}}_i^{(-)} &= \frac{c_i(t_{i+1} - t_i)\,\boldsymbol{v}_{i-1} + d_i(t_i - t_{i-1})\,\boldsymbol{v}_i}{t_{i+1} - t_{i-1}}
\end{aligned}$$

These equations are used in the implementation of the class *splines.KochanekBartels* (page 161).

`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. To find the outer tangent according to *natural* end conditions, the second derivative is set to 0 at the beginning and end of the curve.

We are looking only at the non-uniform case here, it's easy to get to the uniform case by setting $\Delta_i = 1$.

In case you were wondering, natural end conditions are sometimes also called "relaxed" end conditions.

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

As usual, we are getting some help from utility.py:

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

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

Begin

We are starting with the first polynomial segment $\boldsymbol{p}_0(t)$, with $t_0 \leq t \leq t_1$.

[4]:
```
t0, t1 = sp.symbols('t:2')
```

The coefficients ...

[5]:
```
a0, b0, c0, d0 = sp.symbols('a:dbm0')
```

... multiplied with the *monomial basis* (page 2) give us the uniform polynomial ...

[6]:
```
d0 * t**3 + c0 * t**2 + b0 * t + a0
```

[6]: $\boldsymbol{d}_0 t^3 + \boldsymbol{c}_0 t^2 + \boldsymbol{b}_0 t + \boldsymbol{a}_0$

... which we re-scale to the desired parameter range:

[7]:
```
p0 = NamedExpression('pbm0', _.subs(t, (t - t0) / (t1 - t0)))
p0
```

[7]:
$$\boldsymbol{p}_0 = \frac{\boldsymbol{d}_0 (t - t_0)^3}{(-t_0 + t_1)^3} + \frac{\boldsymbol{c}_0 (t - t_0)^2}{(-t_0 + t_1)^2} + \frac{\boldsymbol{b}_0 (t - t_0)}{-t_0 + t_1} + \boldsymbol{a}_0$$

We need the first derivative (a.k.a velocity, a.k.a. tangent vector):

[8]:
```
pd0 = p0.diff(t)
pd0
```

[8]:
$$\frac{d}{dt}\boldsymbol{p}_0 = \frac{3\boldsymbol{d}_0 (t - t_0)^2}{(-t_0 + t_1)^3} + \frac{\boldsymbol{c}_0 (2t - 2t_0)}{(-t_0 + t_1)^2} + \frac{\boldsymbol{b}_0}{-t_0 + t_1}$$

Similar to the *notebook about non-uniform Hermite splines* (page 24), we are interested in the function values and first derivatives at the control points:

$$\begin{aligned} x_0 &= p_0(t_0) \\ x_1 &= p_0(t_1) \\ \dot{x}_0 &= p_0'(t_0) \\ \dot{x}_1 &= p_0'(t_1) \end{aligned}$$

[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('xdotbm0'),
    pd0.evaluated_at(t, t1).with_name('xdotbm1'),
]
```

To get simpler equations, we are substituting $\Delta_0 = t_1 - t_0$. Note that this is 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))
```

$x_0 = a_0$

$x_1 = a_0 + b_0 + c_0 + d_0$

$\dot{x}_0 = \frac{b_0}{\Delta_0}$

$\dot{x}_1 = \frac{b_0}{\Delta_0} + \frac{2c_0}{\Delta_0} + \frac{3d_0}{\Delta_0}$

[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)))
```

$a_0 = x_0$

$b_0 = \Delta_0 \dot{x}_0$

$c_0 = -2\Delta_0 \dot{x}_0 - \Delta_0 \dot{x}_1 - 3x_0 + 3x_1$

$d_0 = \Delta_0 \dot{x}_0 + \Delta_0 \dot{x}_1 + 2x_0 - 2x_1$

The second derivative (a.k.a. acceleration) . . .

[14]:
```
pdd0 = pd0.diff(t)
pdd0
```

[14]: $\frac{d^2}{dt^2} p_0 = \frac{3d_0 (2t - 2t_0)}{(-t_0 + t_1)^3} + \frac{2c_0}{(-t_0 + t_1)^2}$

. . . at the beginning of the curve ($t = t_0$) . . .

```
[15]: pdd0.evaluated_at(t, t0)
```

[15]: $$\left.\frac{d^2}{dt^2}\boldsymbol{p}_0\right|_{t=t_0} = \frac{2\boldsymbol{c}_0}{(-t_0+t_1)^2}$$

… is set to zero …

```
[16]: sp.Eq(_.expr, 0).subs(coefficients_begin)
```

[16]: $$\frac{2\left(2t_0\dot{\boldsymbol{x}}_0 - 2t_1\dot{\boldsymbol{x}}_0 + t_0\dot{\boldsymbol{x}}_1 - t_1\dot{\boldsymbol{x}}_1 - 3\boldsymbol{x}_0 + 3\boldsymbol{x}_1\right)}{(-t_0+t_1)^2} = 0$$

… leading to an expression for the initial tangent vector:

```
[17]: xd0 = NamedExpression.solve(_, 'xdotbm0')
      xd0.subs(delta_begin)
```

[17]: $$\dot{\boldsymbol{x}}_0 = -\frac{\Delta_0\dot{\boldsymbol{x}}_1 + 3\boldsymbol{x}_0 - 3\boldsymbol{x}_1}{2\Delta_0}$$

This can also be written as

$$\dot{\boldsymbol{x}}_0 = \frac{3\left(\boldsymbol{x}_1 - \boldsymbol{x}_0\right)}{2\Delta_0} - \frac{\dot{\boldsymbol{x}}_1}{2}.$$

End

If a spline has N vertices, it has $N-1$ polynomial segments and the last polynomial segment is $\boldsymbol{p}_{N-2}(t)$, with $t_{N-2} \le t \le t_{N-1}$. To simplify the notation a bit, let's assume we have $N = 10$ vertices, which makes $\boldsymbol{p}_8$ the last polynomial segment. The following steps are very similar to the above derivation of the start conditions.

```
[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]: $$\boldsymbol{d}_8 t^3 + \boldsymbol{c}_8 t^2 + \boldsymbol{b}_8 t + \boldsymbol{a}_8$$

```
[21]: p8 = NamedExpression('pbm8', _.subs(t, (t - t8) / (t9 - t8)))
      p8
```

[21]: $$\boldsymbol{p}_8 = \frac{\boldsymbol{d}_8\left(t-t_8\right)^3}{(-t_8+t_9)^3} + \frac{\boldsymbol{c}_8\left(t-t_8\right)^2}{(-t_8+t_9)^2} + \frac{\boldsymbol{b}_8\left(t-t_8\right)}{-t_8+t_9} + \boldsymbol{a}_8$$

```
[22]: pd8 = p8.diff(t)
      pd8
```

[22]: $$\frac{d}{dt}\boldsymbol{p}_8 = \frac{3\boldsymbol{d}_8\left(t-t_8\right)^2}{(-t_8+t_9)^3} + \frac{\boldsymbol{c}_8\left(2t-2t_8\right)}{(-t_8+t_9)^2} + \frac{\boldsymbol{b}_8}{-t_8+t_9}$$

$$\begin{aligned} \boldsymbol{x}_{N-2} &= \boldsymbol{p}_{N-2}(t_{N-2}) \\ \boldsymbol{x}_{N-1} &= \boldsymbol{p}_{N-2}(t_{N-1}) \\ \dot{\boldsymbol{x}}_{N-2} &= \boldsymbol{p}'_{N-2}(t_{N-2}) \\ \dot{\boldsymbol{x}}_{N-1} &= \boldsymbol{p}'_{N-2}(t_{N-1}) \end{aligned}$$

```
[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('xdotbm8'),
          pd8.evaluated_at(t, t9).with_name('xdotbm9'),
      ]
```

We define $\Delta_8 = t_9 - t_8$:

```
[24]: delta_end = [
          (t8, 0),
          (t9, sp.Symbol('Delta8')),
      ]
```

```
[25]: for e in equations_end:
          display(e.subs(delta_end))
```

$\boldsymbol{x}_8 = \boldsymbol{a}_8$

$\boldsymbol{x}_9 = \boldsymbol{a}_8 + \boldsymbol{b}_8 + \boldsymbol{c}_8 + \boldsymbol{d}_8$

$\dot{\boldsymbol{x}}_8 = \frac{\boldsymbol{b}_8}{\Delta_8}$

$\dot{\boldsymbol{x}}_9 = \frac{\boldsymbol{b}_8}{\Delta_8} + \frac{2\boldsymbol{c}_8}{\Delta_8} + \frac{3\boldsymbol{d}_8}{\Delta_8}$

```
[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)))
```

$\boldsymbol{a}_8 = \boldsymbol{x}_8$

$\boldsymbol{b}_8 = \Delta_8 \dot{\boldsymbol{x}}_8$

$\boldsymbol{c}_8 = -2\Delta_8 \dot{\boldsymbol{x}}_8 - \Delta_8 \dot{\boldsymbol{x}}_9 - 3\boldsymbol{x}_8 + 3\boldsymbol{x}_9$

$\boldsymbol{d}_8 = \Delta_8 \dot{\boldsymbol{x}}_8 + \Delta_8 \dot{\boldsymbol{x}}_9 + 2\boldsymbol{x}_8 - 2\boldsymbol{x}_9$

This time, the second derivative . . .

```
[28]: pdd8 = pd8.diff(t)
      pdd8
```

[28]: $\frac{d^2}{dt^2}\boldsymbol{p}_8 = \frac{3\boldsymbol{d}_8\,(2t - 2t_8)}{(-t_8 + t_9)^3} + \frac{2\boldsymbol{c}_8}{(-t_8 + t_9)^2}$

. . . *at the end* of the last segment ($t = t_9$) . . .

```
[29]: pdd8.evaluated_at(t, t9)
```

[29]: $\left.\frac{d^2}{dt^2}\boldsymbol{p}_8\right|_{t=t_9} = \frac{3\boldsymbol{d}_8\,(-2t_8 + 2t_9)}{(-t_8 + t_9)^3} + \frac{2\boldsymbol{c}_8}{(-t_8 + t_9)^2}$

… is set to zero …

[30]:
```
sp.Eq(_.expr, 0).subs(coefficients_end)
```

[30]:
$$\frac{3\left(-2t_8+2t_9\right)\left(-t_8\dot{x}_8+t_9\dot{x}_8-t_8\dot{x}_9+t_9\dot{x}_9+2x_8-2x_9\right)}{\left(-t_8+t_9\right)^3}+\frac{2\left(2t_8\dot{x}_8-2t_9\dot{x}_8+t_8\dot{x}_9-t_9\dot{x}_9-3x_8+3x_9\right)}{\left(-t_8+t_9\right)^2}=0$$

… leading to an expression for the final tangent vector:

[31]:
```
xd9 = NamedExpression.solve(_, 'xdotbm9')
xd9.subs(delta_end)
```

[31]:
$$\dot{x}_9=-\frac{\Delta_8\dot{x}_8+3x_8-3x_9}{2\Delta_8}$$

Luckily, that's symmetric to the result we got above.

The equation can be generalized to

$$\dot{x}_{N-1}=\frac{3\left(x_{N-1}-x_{N-2}\right)}{2\Delta_{N-2}}-\frac{\dot{x}_{N-2}}{2}.$$

Example

We are showing a one-dimensional example where 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, 1
times = 0, 4, 5
slope = 2
```

We are using a few helper functions from helper.py for plotting:

[33]:
```
from helper import plot_sympy, grid_lines
```

[34]:
```
x0, x1 = sp.symbols('xbm0:2')
x8, x9 = sp.symbols('xbm8:10')
xd1 = sp.symbols('xdotbm1')
xd8 = sp.symbols('xdotbm8')
```

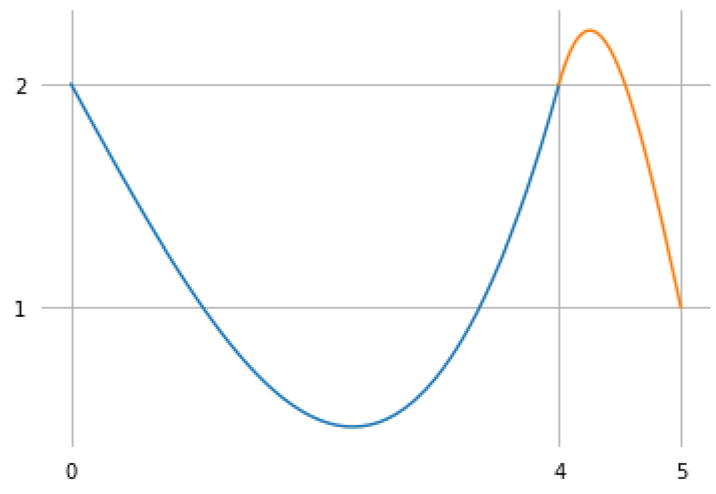
[35]:
```
begin = p0.subs(coefficients_begin).subs_symbols(xd0).subs({
    t0: times[0],
    t1: times[1],
    x0: values[0],
    x1: values[1],
    xd1: slope,
}).with_name(r'p_\text{begin}')
end = p8.subs(coefficients_end).subs_symbols(xd9).subs({
    t8: times[1],
    t9: times[2],
    x8: values[1],
    x9: values[2],
```

(continues on next page)

(continued from previous page)

```
    xd8: slope,
}).with_name(r'p_\text{end}')
```

[36]:
```
plot_sympy(
    (begin.expr, (t, times[0], times[1])),
    (end.expr, (t, times[1], times[2])))
grid_lines(times, [1, 2])
```



[37]:
```
begin.diff(t).evaluated_at(t, times[0])
```

[37]: $\left.\frac{d}{dt} p_{\text{begin}}\right|_{t=0} = -1$

[38]:
```
end.diff(t).evaluated_at(t, times[-1])
```

[38]: $\left.\frac{d}{dt} p_{\text{end}}\right|_{t=5} = -\frac{5}{2}$

Bézier Control Points

Up to now we have assumed that we know one of the tangent vectors and want to find the other tangent vector in order to construct a *Hermite spline* (page 13). What if we want to construct a *Bézier spline* (page 39) instead?

If the inner Bézier control points $\tilde{x}_1^{(-)}$ and $\tilde{x}_{N-2}^{(+)}$ are given, we can insert the equations for the tangent vectors from the *notebook about non-uniform Bézier splines* (page 54) into our tangent vector equations from above and solve them for the outer control points $\tilde{x}_0^{(+)}$ and $\tilde{x}_{N-1}^{(-)}$, respectively.

[39]:
```
xtilde0, xtilde1 = sp.symbols('xtildebm0^(+) xtildebm1^(-)')
```

[40]:
```
NamedExpression.solve(xd0.subs({
    xd0.name: 3 * (xtilde0 - x0) / (t1 - t0),
    xd1: 3 * (x1 - xtilde1) / (t1 - t0),
}), xtilde0)
```

[40]: $\tilde{x}_0^{(+)} = \frac{x_0}{2} + \frac{\tilde{x}_1^{(-)}}{2}$

[41]:
```
xtilde8, xtilde9 = sp.symbols('xtildebm8^(+) xtildebm9^(-)')
```

[42]:
```
NamedExpression.solve(xd9.subs({
    xd8: 3 * (xtilde8 - x8) / (t9 - t8),
    xd9.name: 3 * (x9 - xtilde9) / (t9 - t8),
}), xtilde9)
```

[42]:
$$\tilde{x}_9^{(-)} = \frac{x_9}{2} + \frac{\tilde{x}_8^{(+)}}{2}$$

Note that all Δ_i cancel each other out (as well as the inner vertices x_1 and x_{N-2}) and we get very simple equations for the "natural" end conditions:

$$\begin{aligned} \tilde{x}_0^{(+)} &= \frac{x_0 + \tilde{x}_1^{(-)}}{2} \\ \tilde{x}_{N-1}^{(-)} &= \frac{x_{N-1} + \tilde{x}_{N-2}^{(+)}}{2} \end{aligned}$$

.... 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 114) below).

A Python implementation of piecewise monotone one-dimensional cubic splines is available in the *splines.PiecewiseMonotoneCubic* (page 162) class.

The SciPy package provides a similar tool with the pchip_interpolate()[27] function and the PchipInterpolator[28] class (see below for more details).

The 3D animation software Blender[29] provides an Auto Clamped[30] property for creating piecewise monotone animation cuves.

Examples

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

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

We use a few helper functions from helper.py for plotting:

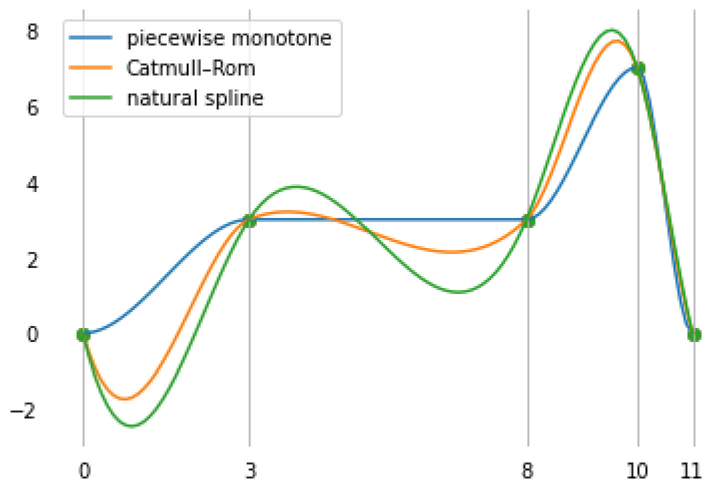
[3]:
```
from helper import plot_spline_1d, grid_lines
```

[27] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.pchip_interpolate.html
[28] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PchipInterpolator.html
[29] https://www.blender.org
[30] https://docs.blender.org/manual/en/dev/editors/graph_editor/fcurves/properties.html#editors-graph-fcurves-settings-handles

[4]:
```
values = 0, 3, 3, 7
times = 0, 3, 8, 10, 11
```

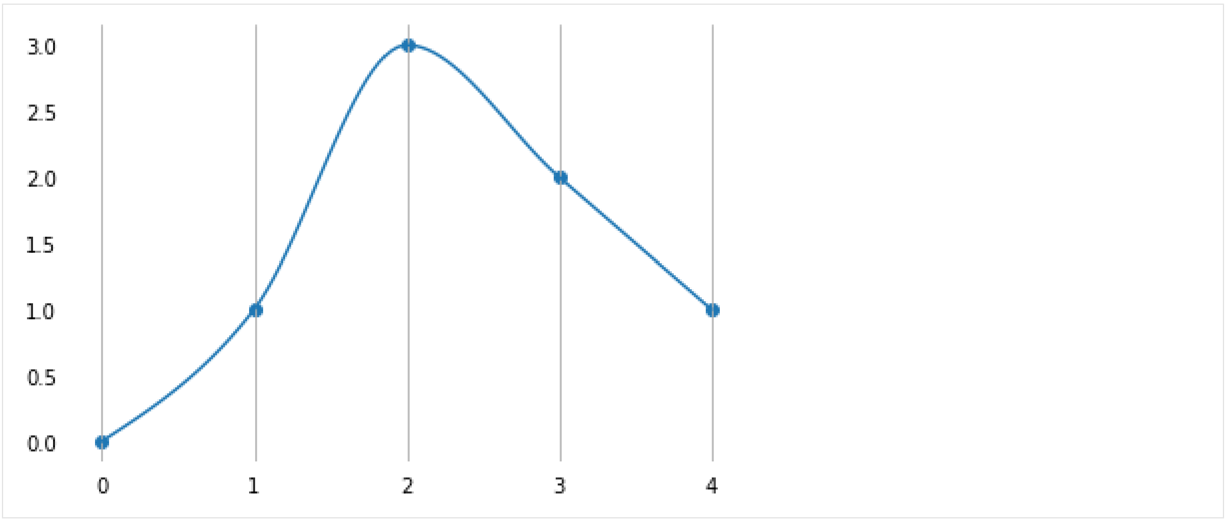
Let's compare a piecewise monotone spline with a *Catmull–Rom spline* (page 55) and a *natural spline* (page 29):

[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)
```

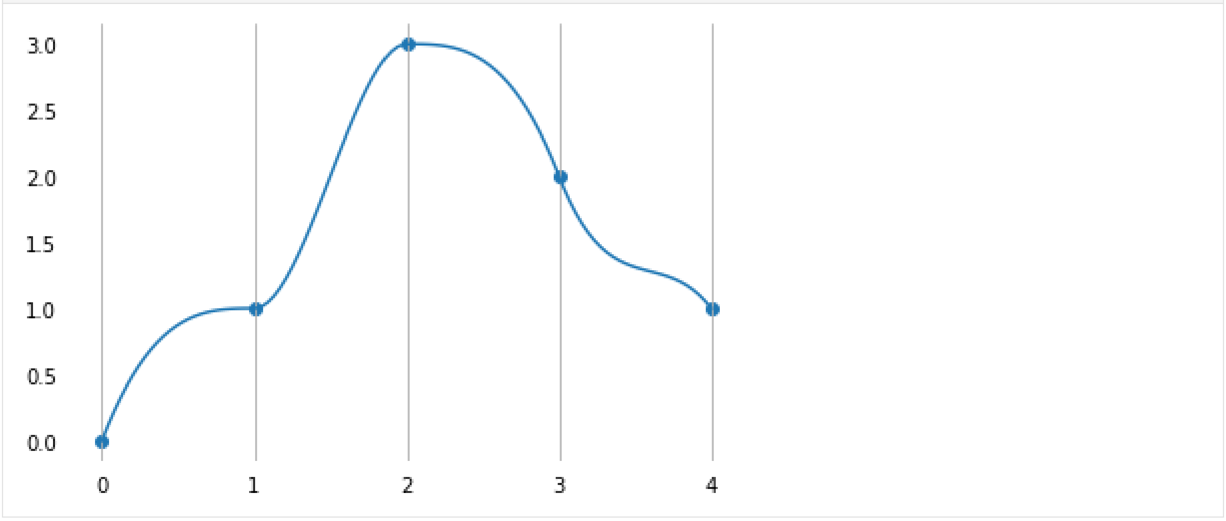
[7]:
```
plot_piecewise_monotone([0, 1, 3, 2, 1])
```



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:
    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. We have already seen the *standard three-point difference* in the *tangent vectors of non-uniform Catmull–Rom splines* (page 72).

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 162) 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[31] 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[32] 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 ...

[31] https://www.sympy.org/
[32] https://en.wikipedia.org/wiki/Fortran

[13]:
```
HSUM = h1 + h2
```

[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 [Mol04], 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()[33] function and the PchipInterpolator[34] class.

[21]:
```
from scipy.interpolate import PchipInterpolator
```

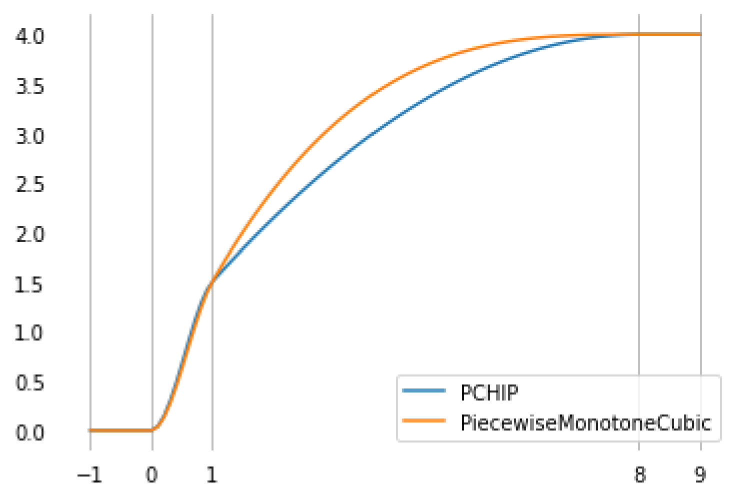
[33] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.pchip_interpolate.html
[34] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PchipInterpolator.html

More Examples

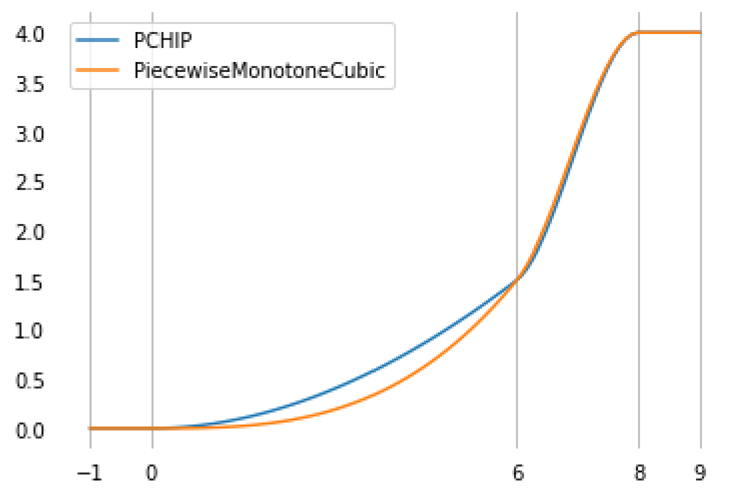
To illustrate the differences between the two approaches mentioned above, let's plot a few examples. Both methods are piecewise monotone, but their exact shape is slightly different. Decide for yourself which one is more "visually pleasing"!

[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)
```

[23]:
```
compare_pchip([0, 0, 1.5, 4, 4], [-1, 0, 1, 8, 9])
```

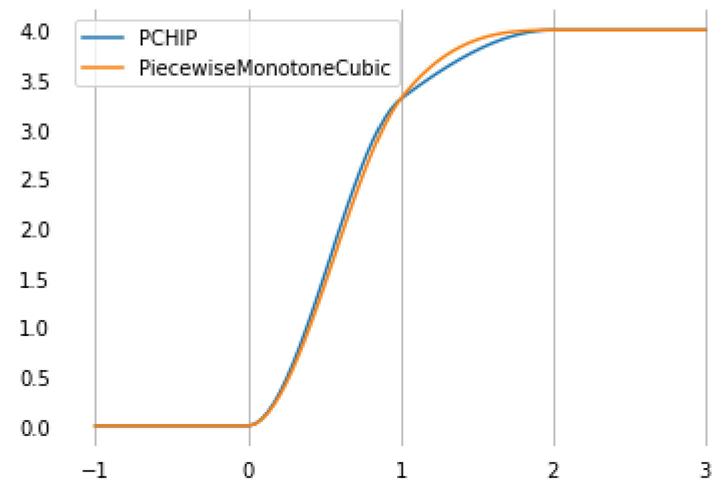


[24]:
```
compare_pchip([0, 0, 1.5, 4, 4], [-1, 0, 6, 8, 9])
```

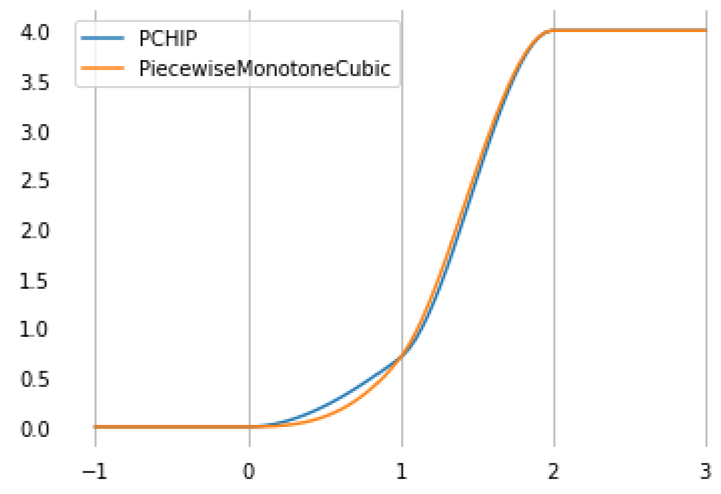


There is even a slight difference in the uniform case:

```
[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 115).

Monotone Interpolation

When using the aforementioned piecewise monotone algorithms with monotone data, the whole interpolant will be monotone.

The class *splines.MonotoneCubic* (page 162) works very much the same as *splines.PiecewiseMonotoneCubic* (page 162), 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 162) can be used to find the associated parameter value.

```
[27]: s = splines.MonotoneCubic([0, 2, 2, 6, 6], grid=[0, 2, 3, 6, 8])
```

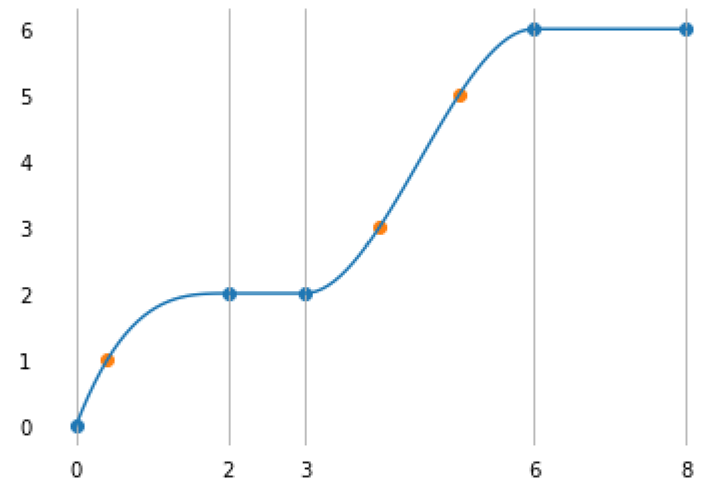
```
[28]: probes = 1, 3, 5
```

```
[29]: fig, ax = plt.subplots()
      plot_spline_1d(s)
```

(continues on next page)

(continued from previous page)

```
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 101) 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
```

(continues on next page)

(continued from previous page)

```
      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.
>
> —[Mol04], section 3.4

Section 3.6 of [Mol04] 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[35] uses the same implementation (ported to Python)[36].

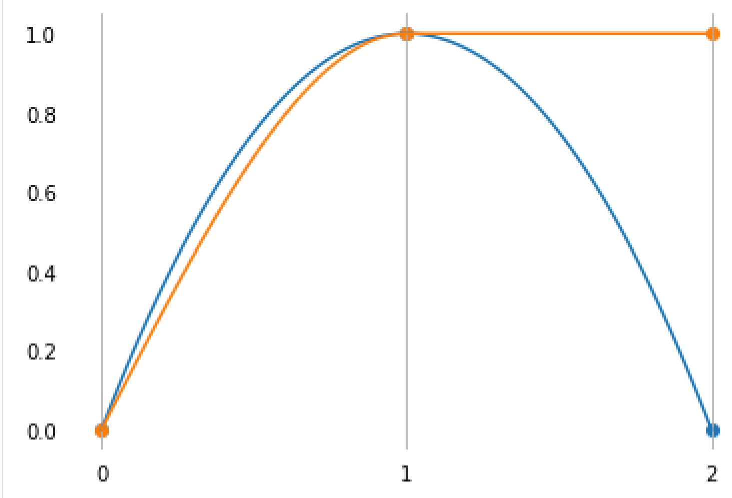
This implementation ensures monotonicity, but it might seem a bit strange that for calculating the first slope, the second slope is not directly taken into account.

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.

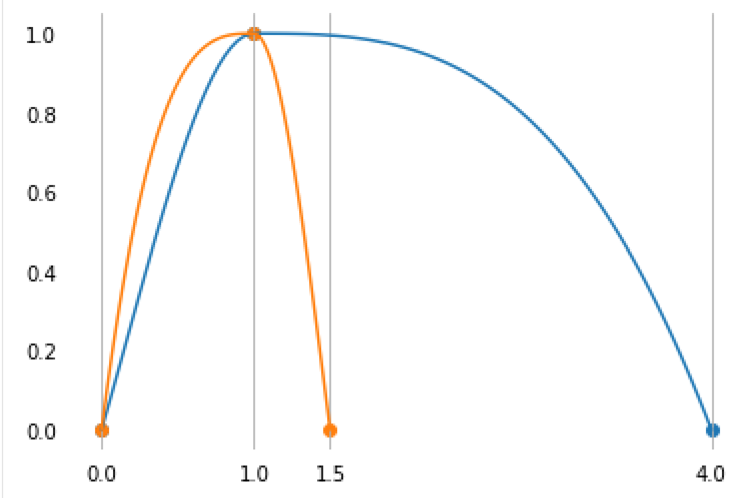
[32]:
```
def plot_pchip(values, grid, **kwargs):
    pchip = PchipInterpolator(grid, values)
    times = np.linspace(grid[0], grid[-1], 100)
    plt.plot(times, pchip(times), **kwargs)
    plt.scatter(grid, pchip(grid))
    grid_lines(x=grid)
```

[35] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PchipInterpolator.html
[36] https://github.com/scipy/scipy/blob/v1.6.1/scipy/interpolate/_cubic.py#L237-L250

[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])
```

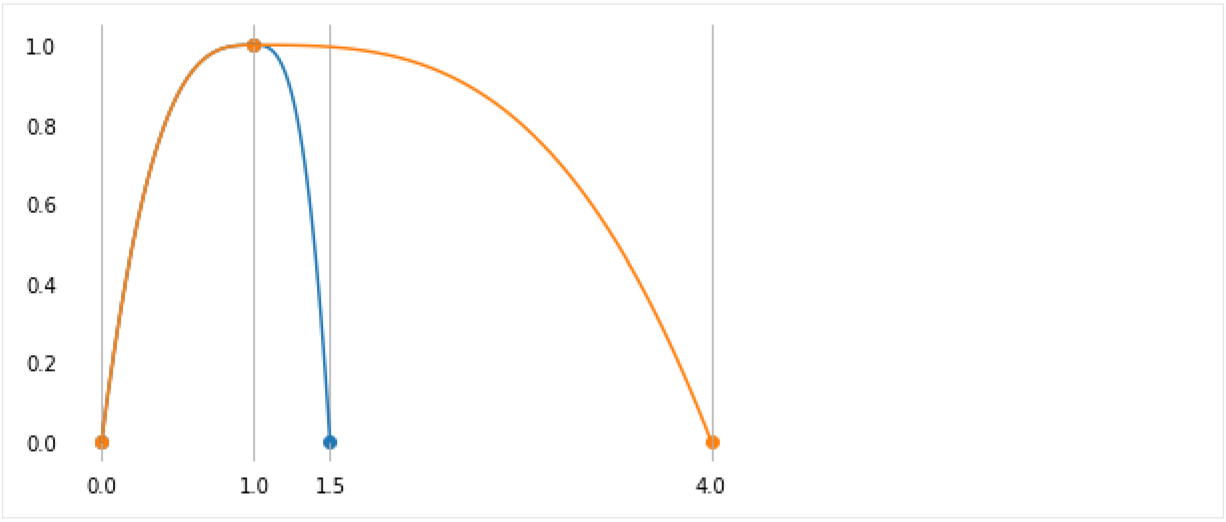


In both of the above examples, the very left slope depends on properties of the very right segment.

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 162) 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])
```



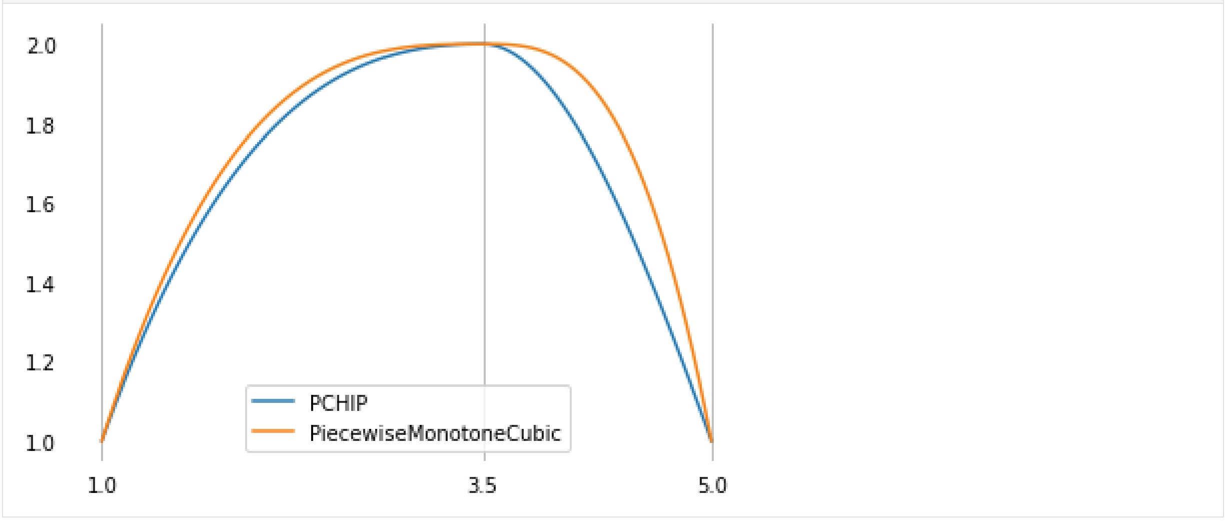
The initial and final slopes of *splines.PiecewiseMonotoneCubic* (page 162) are implemented like this:

```
[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
```

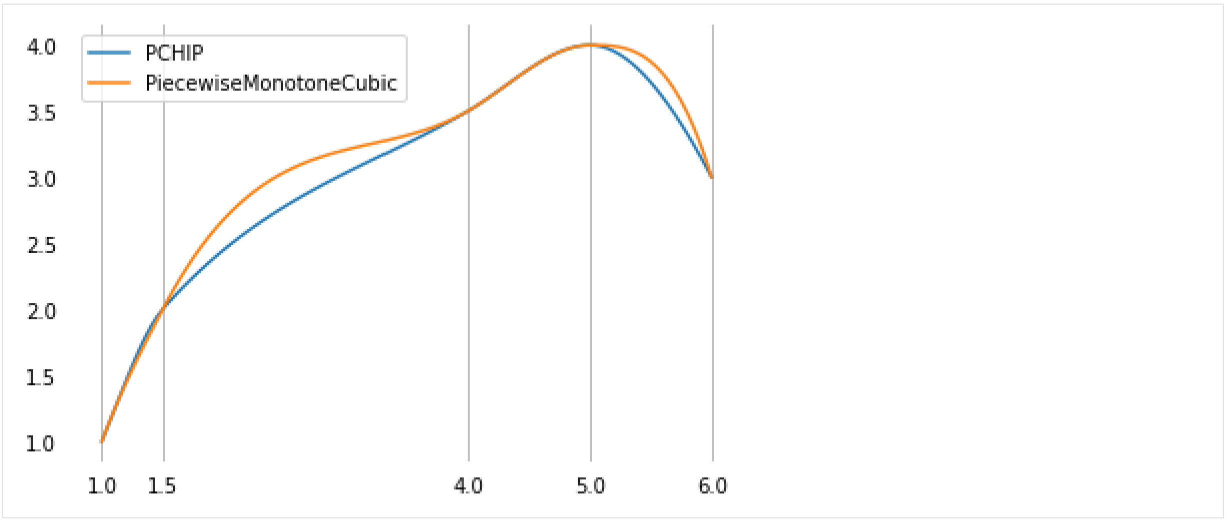
Even More Examples

The following example plots show different slopes at the beginning and end due to different end conditions.

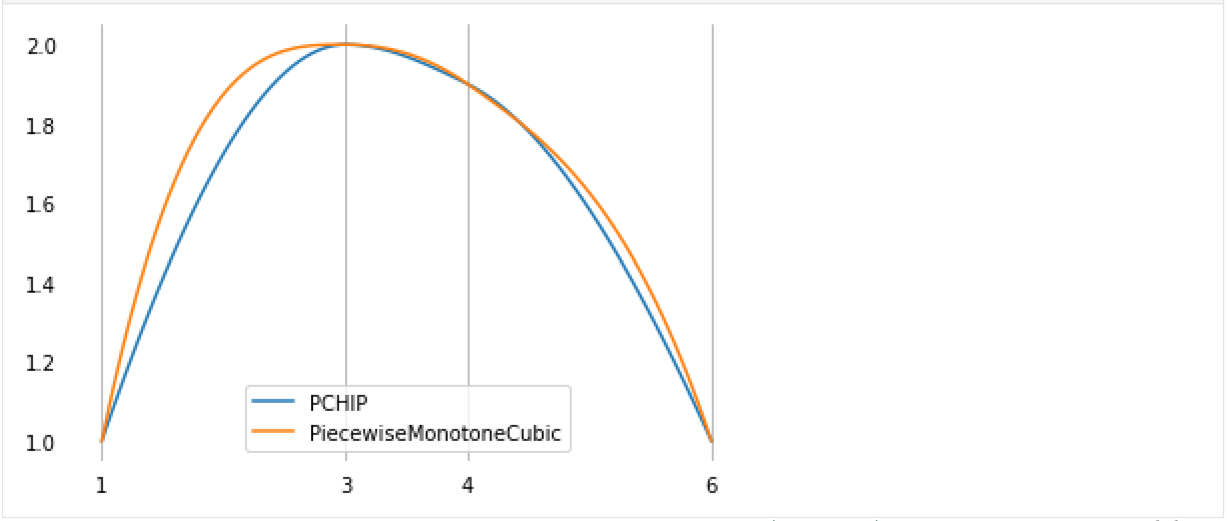
```
[37]: compare_pchip([1, 2, 1], [1, 3.5, 5])
```



```
[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

We are interested in *unit quaternions* (see below), because they are a very useful representation of rotations. But before we go into that, we should probably mention what a quaternion[37] is. We don't need all the details, we just need to know a few facts (without burdening ourselves too much with mathematical rigor):

- Quaternions live in the four-dimensional Euclidean space $\mathbb{R}^4$. Each quaternion has exactly one corresponding element of $\mathbb{R}^4$ and vice versa.
- Unlike elements of $\mathbb{R}^4$, quaternions support a special kind of *quaternion multiplication*.
- Quaternion multiplication is weird. The order of operands matters (i.e. multiplication is non-commutative[38]).

[37] https://en.wikipedia.org/wiki/Quaternion
[38] https://en.wikipedia.org/wiki/Noncommutative

A Python implementation is available in the class *splines.quaternion.Quaternion* (page 163).

Quaternion Representations

There are multiple equivalent ways to represent quaternions. Their original algebraic representation is

$$q = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k},$$

where $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$. It it important to note that the order in which the *basic quaternions* **i**, **j** and **k** are multiplied matters: $\mathbf{ij} = \mathbf{k}$, $\mathbf{ji} = -\mathbf{k}$ (i.e. their multiplication is anticommutative[39]). The information given so far should be sufficient to derive quaternion multiplication, but let's not do that right now. Quaternions can also be represented as pairs containing a scalar and a 3D vector:

$$q = (w, \vec{v}) = (w, (x, y, z))$$

Sometimes, the scalar and vector parts are also called "real" and "imaginary" parts, respectively. The four components can also be displayed as simple 4-tuples, which can be interpreted as coordinates of the four-dimensional Euclidean space $\mathbb{R}^4$:

$$q = (w, x, y, z) \quad \text{or} \quad q = (x, y, z, w)$$

The order of components can be chosen arbitrarily. In mathematical textbooks, the order (w, x, y, z) is often preferred (and sometimes written as (a, b, c, d)). In numerical software implementations, however, the order (x, y, z, w) is more common (probably because it is memory-compatible with 3D vectors (x, y, z)). In the Python class *splines.quaternion.Quaternion* (page 163), these representations are available via the attributes *scalar* (page 163), *vector* (page 163), *wxyz* (page 164) and *xyzw* (page 164).

There are even more ways to represent quaterions, for example as 2x2 complex matrices or as 4x4 real matrices, see for example [McD10].

Unit Quaternions

Quite simply, unit quaternions are the set of all quaternions whose distance to the origin $(0, (0, 0, 0))$ equals 1. In $\mathbb{R}^3$, all elements with unit distance from the origin form the *unit sphere* (a.k.a. S^2), which is a two-dimensional curved space. Since quaternions inhabit $\mathbb{R}^4$, the unit quaternions form the *unit hypersphere* (a.k.a. S^3), which is a three-dimensional curved space.

One important unit quaternion is $(1, (0, 0, 0))$, sometimes written as **1**, which corresponds to the real number 1.

A Python implementation of unit quaternions is available in the class *splines.quaternion.UnitQuaternion* (page 164).

[39] https://en.wikipedia.org/wiki/Anticommutative_property

Unit Quaternions as Rotations

Given a (normalized) rotation axis $\vec{n}$ and a rotation angle α (in radians), we can create a corresponding quaternion (which will have unit length):

$$q = \left(\cos\frac{\alpha}{2}, \vec{n}\sin\frac{\alpha}{2}\right)$$

Unit quaternions are a *double cover* over the rotation group (a.k.a. SO(3)[40]), 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 details can be found on Wikipedia[41].

To get a bit of intuition, let's plot a few quaternion rotations (with the help of helper.py).

[1]:
```
from helper import angles2quat, plot_rotation
```

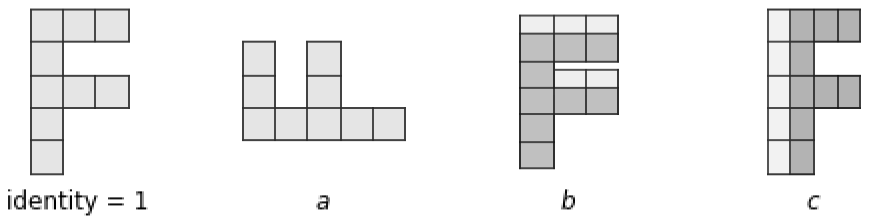
The quaternion **1** represents "no rotation at all".

[2]:
```
identity = angles2quat(0, 0, 0)
identity
```

[2]:
```
UnitQuaternion(scalar=1.0, vector=(0.0, 0.0, 0.0))
```

[3]:
```
a = angles2quat(90, 0, 0)
b = angles2quat(0, 35, 0)
c = angles2quat(0, 0, 45)
```

[4]:
```
plot_rotation({
    'identity = 1': identity,
    '$a$': a,
    '$b$': b,
    '$c$': c,
});
```



Axes Conventions

When converting between rotation angles (see Euler/Tait–Bryan angles[42]) and unit quaternions, we can freely choose from a multitude of axes conventions[43]. Here we choose a (global) coordinate system where the x-axis points towards the right margin of the page and the y-axis points towards the top of the page. We are using a right-handed coordinate system, which leaves the z-axis pointing out of the page, towards the viewer. The helper function `angles2quat()` takes three angles (in degrees) which are applied in this order:

[40] https://en.wikipedia.org/wiki/3D_rotation_group
[41] https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation
[42] https://en.wikipedia.org/wiki/Euler_angles
[43] https://en.wikipedia.org/wiki/Axes_conventions

- *azimuth*: rotation around the (global) z-axis
- *elevation*: rotation around the (previously rotated local) x-axis
- *roll*: rotation around the (previously rotated local) y-axis

This is equivalent to applying the angles in the opposite order, but using a global frame of reference for each rotation.

The sign of the rotation angles always follows the right-hand rule[44].

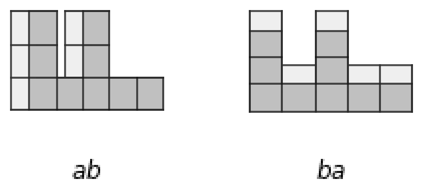
Quaternion Multiplication

As mentioned above, quaternion multiplication (sometimes called *Hamilton product*) is noncommutative, i.e. the order of operands matters. When using unit quaternions to represent rotations, quaternion multiplication can be used to apply rotations to other rotations. Given a rotation q_0, we can apply another rotation q_1 by left-multiplication: $q_1 q_0$. In other words, applying a rotation of q_0 followed by a rotation of q_1 is equivalent to applying a single rotation $q_1 q_0$. Note that q_1 represents a rotation in the global frame of reference.

When dealing with local frames of reference, the order of multiplications has to be reversed. Given a rotation q_2, which describes a new local coordinate system, we can apply a *local* rotation q_3 (relative to this new coordinate system) by right-multiplication: $q_2 q_3$. In other words, applying a rotation of q_2 followed by a rotation of q_3 (relative to the local coordinate system defined by q_2) is equivalent to applying a single rotation $q_2 q_3$.

In general, changing the order of rotations changes the resulting rotation ($q_m q_n \neq q_n q_m$):

```
[5]: plot_rotation({'$ab$': a * b, '$ba$': b * a});
```



However, there is an exception when all rotation axes are the same, in which case the rotation angles can simply be added (in arbitrary order, of course).

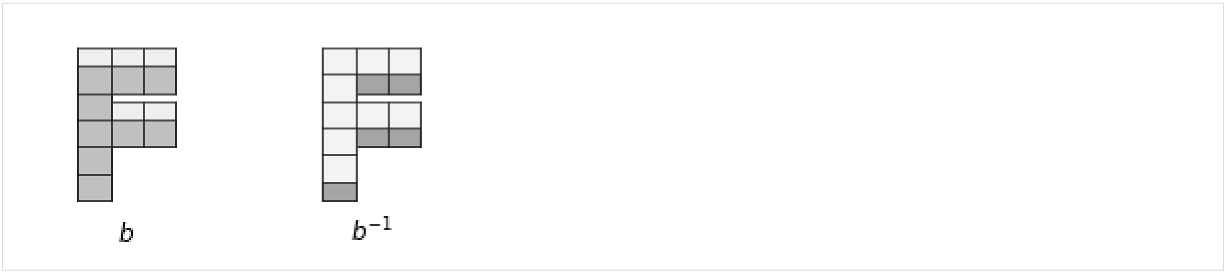
The quaternion $\mathbf{1} = (1, (0, 0, 0))$ is the identity element with regards to quaternion multiplication. A multiplication with this (on either side) leads to an unchanged rotation.

Inverse

The multiplicative inverse of a quaternion is written as q^{-1}. When talking about rotations, this operation leads to a new rotation with the same rotation axis but with negated angle (or equivalently, the same angle with a flipped rotation axis).

```
[6]: plot_rotation({'$b$': b, '$b^{-1}$': b.inverse()});
```

[44] https://en.wikipedia.org/wiki/Right-hand_rule#Rotations



By multiplying a rotation with its inverse, the original rotation can be undone: $qq^{-1} = q^{-1}q = \mathbf{1}$. Since both operands have the same rotation axis, the order doesn't matter in this case.

For unit quaternions, the inverse q^{-1} equals the conjugate $\overline{q}$. The conjugate of a quaternion is constructed by negating its vector part (and keeping its scalar part unchanged). This can be achieved by negating the rotation axis $\vec{n}$. Alternatively, we can negate the rotation angle, since $\sin(-\phi) = -\sin(\phi)$ (antisymmetric) and $\cos(-\phi) = \cos(\phi)$ (symmetric).

$$\overline{q} = (w, -\vec{v}) = \left(\cos\frac{\alpha}{2}, -\vec{n}\sin\frac{\alpha}{2}\right) = \left(\cos\frac{-\alpha}{2}, \vec{n}\sin\frac{-\alpha}{2}\right)$$

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:

$$q_0q_{0,1} = q_1$$

Now we can left-multiply both sides with ${q_0}^{-1}$:

$${q_0}^{-1}q_0q_{0,1} = {q_0}^{-1}q_1$$

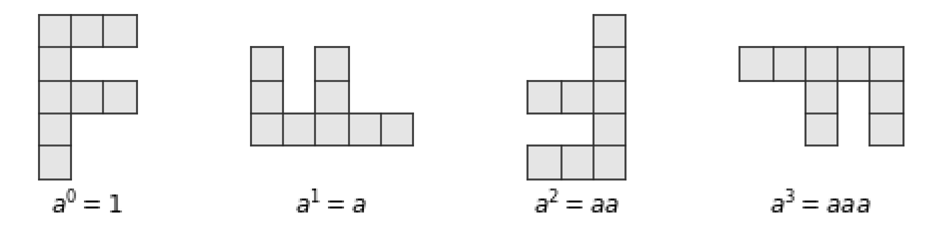
${q_0}^{-1}q_0$ cancels out and we get:

$$q_{0,1} = {q_0}^{-1}q_1$$

Exponentiation

Raising a unit quaternion to an integer exponent simply means applying the same rotation multiple times:

```
[7]: plot_rotation({
         '$a^0 = 1$': a**0,
         '$a^1 = a$': a**1,
         '$a^2 = aa$': a**2,
         '$a^3 = aaa$': a**3,
     });
```

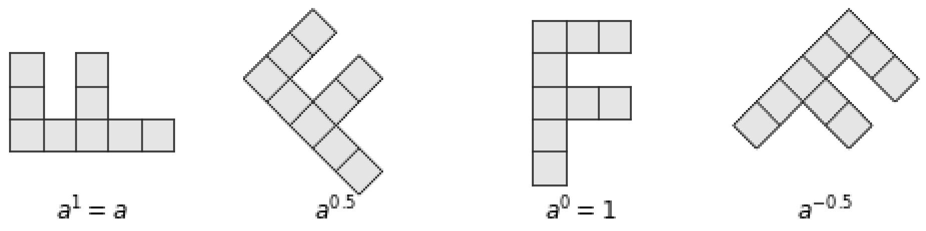


It shouldn't come as a surprise that $q^0 = \mathbf{1}$ and $q^1 = q$.

Using an exponent of -1 is equivalent to taking the inverse (see *above* (page 122)), negative integer exponents apply the inverse rotation multiple times. Non-integer exponents lead to partial rotations, with the exponent k being proportional to the rotation angle. The rotation axis $\vec{n}$ is unchanged by exponentiation.

$$q^k = \left(\cos\frac{k\alpha}{2}, \vec{n}\sin\frac{k\alpha}{2}\right)$$

```
[8]: plot_rotation({
         '$a^1 = a$': a**1,
         '$a^{0.5}$': a**0.5,
         '$a^0 = 1$': a**0,
         '$a^{-0.5}$': a**-0.5,
     });
```

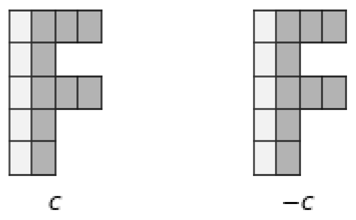


Negation

A quaternion can be negated by negating all 4 of its components. This corresponds to flipping its orientation in 4D space (but keeping its direction and length). For unit quaternions, this means selecting the diametrically opposite (antipodal) point on the unit hypersphere.

Due to the *double cover* property mentioned above, negating a unit quaternion doesn't change the rotation it is representing.

```
[9]: plot_rotation({'$c$': c, '$-c$': -c});
```



One way to negate the scalar part of a unit quaternion is to add π to the argument of the cosine function, since $\cos(\phi + \pi) = -\cos(\phi)$. Because only half of the rotation appears in the argument of the cosine, we have to add 2π to the rotation angle α, which brings us back to the original rotation. Adding 2π to the rotation angle also negates the vector part of the unit quaternion (since $\sin(\phi + \pi) = -\sin(\phi)$), assuming the rotation axis $\vec{n}$ stays unchanged.

$$-q = (-w, -\vec{v}) = \left(\cos\frac{\alpha + 2\pi}{2}, \vec{n}\sin\frac{\alpha + 2\pi}{2}\right)$$

.... 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 describes an interpolation (with constant angular velocity) along the shortest path (a.k.a. geodesic) on the unit hypersphere between two quaternions q_1 and q_2. It is defined as:

$$\operatorname{Slerp}(q_1, q_2; u) = q_1 \left({q_1}^{-1} q_2\right)^u$$

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[45] provides four equivalent ways to describe the same thing:

$$\begin{aligned}\operatorname{Slerp}(q_0, q_1; t) &= q_0 \left({q_0}^{-1} q_1\right)^t \\ &= q_1 \left({q_1}^{-1} q_0\right)^{1-t} \\ &= \left(q_0 {q_1}^{-1}\right)^{1-t} q_1 \\ &= \left(q_1 {q_0}^{-1}\right)^t q_0\end{aligned}$$

[Sho85] also provides an alternative formulation (attributed to Glenn Davis):

$$\operatorname{Slerp}(q_1, q_2; u) = \frac{\sin(1-u)\theta}{\sin\theta} q_1 + \frac{\sin u\theta}{\sin\theta} q_2,$$

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 for quaternions is quite easy to implement ...

[45] https://en.wikipedia.org/wiki/Slerp#Quaternion_Slerp

[1]:
```python
def slerp(one, two, t):
    """Spherical Linear intERPolation."""
    return (two * one.inverse())**t * one
```

... but for your convenience an implementation is also provided in *splines.quaternion.slerp()* (page 165).

Derivation

Before looking at the general case $\operatorname{Slerp}(q_0, q_1; t)$, which interpolates from q_0 to q_1, let's look at the much simpler case of interpolating from the identity **1** to some unit quaternion q.

$$\begin{aligned}\mathbf{1} &= (1, (0, 0, 0))\\ q &= \left(\cos\frac{\alpha}{2}, \vec{n}\sin\frac{\alpha}{2}\right)\end{aligned}$$

To move along the great arc from **1** to q, we simply have to change the angle from 0 to α while the rotation axis $\vec{n}$ stays unchanged.

$$\operatorname{Slerp}(\mathbf{1}, q; t) = \left(\cos\frac{\alpha t}{2}, \vec{n}\sin\frac{\alpha t}{2}\right) = q^t\text{, where } 0 \le t \le 1$$

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 123) $q_{0,1}$:

$$\operatorname{Slerp}(q_0, q_1; t) = \operatorname{Slerp}(\mathbf{1}, q_{0,1}; t)\, q_0$$

Inserting $q_{0,1} = q_1 {q_0}^{-1}$, we get:

$$\operatorname{Slerp}(q_0, q_1; t) = \left(q_1 {q_0}^{-1}\right)^t q_0$$

Alternatively, we can start with q_0 and right-multiply an appropriate Slerp using the *relative rotation (local frame)* (page 123) $q_{0,1} = {q_0}^{-1} q_1$:

$$\operatorname{Slerp}(q_0, q_1; t) = q_0 \operatorname{Slerp}(\mathbf{1}, q_{0,1}; t) = q_0 \left({q_0}^{-1} q_1\right)^t$$

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[46] ...

[2]:
```python
import numpy as np
```

... and a few helper functions from helper.py:

[3]:
```python
from helper import angles2quat, animate_rotations, display_animation
```

We can now define two example quaternions:

[46] https://numpy.org/

```
[4]: q1 = angles2quat(45, -20, -60)
     q2 = angles2quat(-45, 20, 30)
```

Just out of curiosity, let's use the method *rotation_to()* (page 164) to calculate the angle between the two quaternions:

```
[5]: np.degrees(q1.rotation_to(q2).angle)
```

```
[5]: 123.9513586527906
```

If this angle is smaller than 180°, we know that we will get the smallest difference in rotation. If it is larger than 180°, we can negate the second quaternion to get a smaller rotation.

```
[6]: ani_times = np.linspace(0, 1, 50)
```

We show both the original target quaternion and its antipodal point here:

```
[7]: ani = animate_rotations({
         'slerp(q1, q2)': slerp(q1, q2, ani_times),
         'slerp(q1, -q2)': slerp(q1, -q2, ani_times),
     })
```

```
[8]: display_animation(ani, default_mode='reflect')
```

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

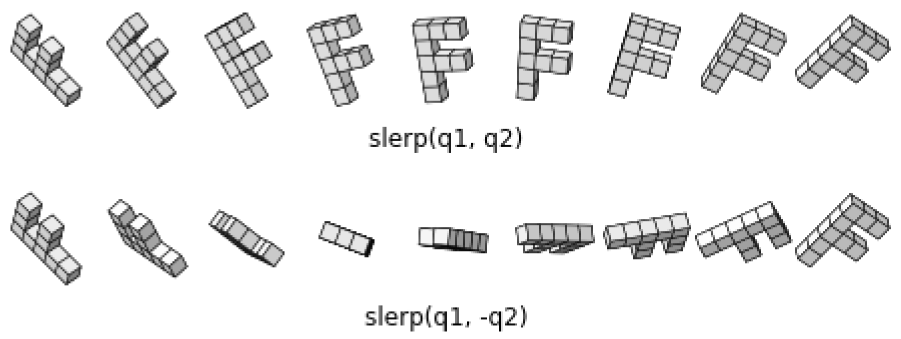
`slerp(q1, q2)` and `slerp(q1, -q2)` move along the same great circle, albeit in different directions. In total, they cover half the circumference of that great circle, which means a rotation angle of 360 degrees. Note that `q2` and `-q2` represent the same rotation (because of the *double cover* property).

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 165) provides a rotation spline that consists of Slerp sections between the given quaternions.

[12]:
```python
from splines.quaternion import PiecewiseSlerp
```

[13]:
```python
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]:
```python
ani = animate_rotations({
    'piecewise Slerp': s.evaluate(np.linspace(s.grid[0], s.grid[-1], 100)),
})
```

[15]:
```python
display_animation(ani, default_mode='loop')
```

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

Each section has its own 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]:
```python
from splines.quaternion import Quaternion
```

[17]:
```python
def lerp(one, two, t):
    """Linear intERPolation."""
    one = np.asarray(one)
    two = np.asarray(two)
    return (1 - t) * one + t * two
```

[18]:
```python
def nlerp(one, two, t):
    """Normalized Linear intERPolation.

    Linear interpolation in 4D quaternion space,
    normalizing the result.

    """
    if not np.isscalar(t):
        # If t is a list, return a list of unit quaternions
        return [nlerp(one, two, t) for t in t]
    *vector, scalar = lerp(one.xyzw, two.xyzw, t)
    return Quaternion(scalar, vector).normalized()
```

As a first example, we try an angle below 180° . . .

[19]:
```
q1 = angles2quat(-60, 10, -10)
q2 = angles2quat(80, -35, -110)
```

[20]:
```
np.degrees(q1.rotation_to(q2).angle)
```

[20]:
```
174.5768498146622
```

. . . which we can also quickly check by means of the dot product:

[21]:
```
assert q1.dot(q2) > 0
```

[22]:
```
ani_times = np.linspace(0, 1, 50)
```

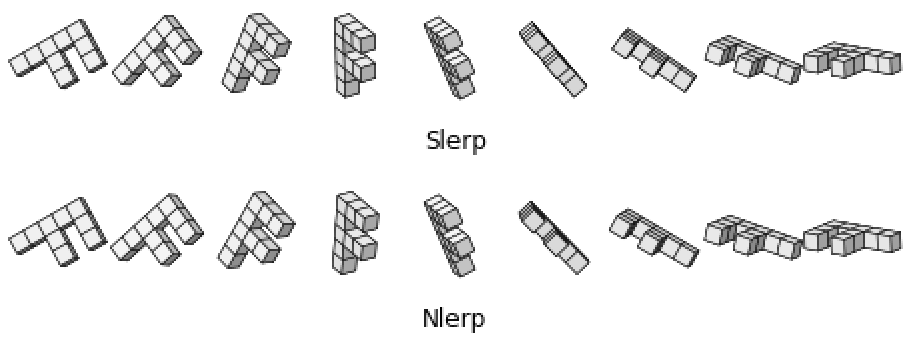
[23]:
```
ani = animate_rotations({
    'Slerp': slerp(q1, q2, ani_times),
    'Nlerp': nlerp(q1, q2, ani_times),
})
```

[24]:
```
display_animation(ani, default_mode='reflect')
```

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

Again, we plot some still images:

[25]:
```
plot_rotations({
    'Slerp': slerp(q1, q2, plot_times),
    'Nlerp': nlerp(q1, q2, plot_times),
}, figsize=(8, 3))
```



The start and end values are (by definition) the same, the middle one 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]:
```
np.degrees(q3.rotation_to(q4).angle)
```

[27]:
```
268.27205892764954
```

Please note that this is a rotation by an angle of far more than 180 degrees!

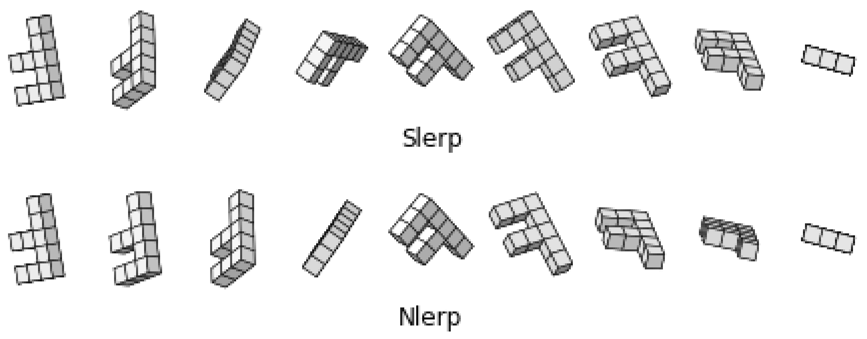
[28]:
```
assert q3.dot(q4) < 0
```

[29]:
```python
ani = animate_rotations({
    'Slerp': slerp(q3, q4, ani_times),
    'Nlerp': nlerp(q3, q4, ani_times),
})
```

[30]:
```python
display_animation(ani, default_mode='reflect')
```

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

[31]:
```python
plot_rotations({
    'Slerp': slerp(q3, q4, plot_times),
    'Nlerp': nlerp(q3, q4, plot_times),
}, figsize=(8, 3))
```



Now the difference is clearly visible, but depending on the application you might want to limit your rotations to ±180 anyway, so this might not be relevant.

.... doc/rotation/slerp.ipynb ends here.

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

2.3 De Casteljau's Algorithm With Slerp

In [Sho85], which famously introduces quaternions to the field of computer graphics, Shoemake suggests to apply a variant of *de Casteljau's Algorithm* (page 40) to a unit quaternion control polygon, using *Slerp* (page 125) instead of linear interpolations.

[1]:
```python
def slerp(one, two, t):
    """Spherical Linear intERPolation."""
    return (two * one.inverse())**t * one
```

We'll also need NumPy and a few helpers from helper.py:

[2]:
```python
import numpy as np
from helper import angles2quat, plot_rotation, plot_rotations
from helper import animate_rotations, display_animation
```

“Cubic”

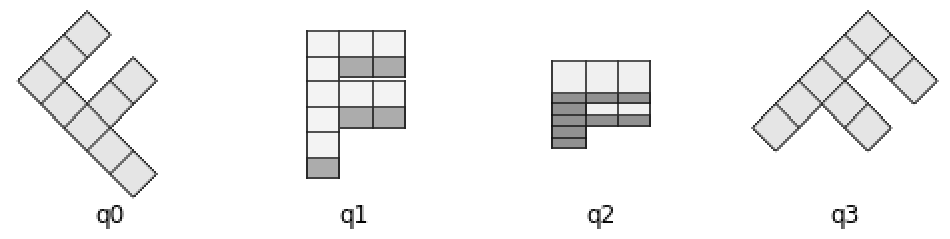
[Sho85] only talks about the “cubic” case, consisting of three nested applications of Slerp. Since this is done in a curved space, the resulting curve is of course not simply a polynomial of degree 3, but something quite a bit more involved. Therefore, we use the term “cubic” in quotes. Shoemake doesn’t talk about the “degree” of the curves at all, they are only called “spherical Bézier curves”.

```
[3]: def cubic_de_casteljau(q0, q1, q2, q3, t):
         """De Casteljau's algorithm of "degree" 3 using Slerp."""
         if not np.isscalar(t):
             # If t is a list, return a list of unit quaternions
             return [cubic_de_casteljau(q0, q1, q2, q3, t) for t in 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,
         )
```

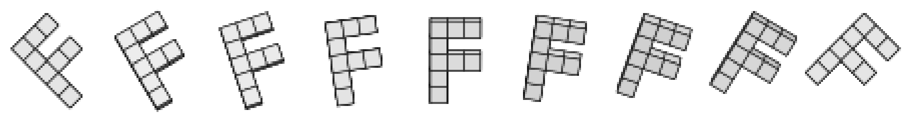
To illustrate this, let’s define 4 unit quaternions that we can use as control points:

```
[4]: q0 = angles2quat(45, 0, 0)
     q1 = angles2quat(0, -40, 0)
     q2 = angles2quat(0, 70, 0)
     q3 = angles2quat(-45, 0, 0)
```

```
[5]: plot_rotation({'q0': q0, 'q1': q1, 'q2': q2, 'q3': q3});
```



```
[6]: plot_rotations(
         cubic_de_casteljau(q0, q1, q2, q3, np.linspace(0, 1, 9)),
         figsize=(8, 1))
```



We can see that the curve starts with the first rotation and ends with the last one. The two middle control quaternions q1 and q2 influence the shape of the rotation curve but they are not part of the interpolant themselves.

```
[7]: ani = animate_rotations(
         cubic_de_casteljau(q0, q1, q2, q3, np.linspace(0, 1, 100)))
```

```
[8]: display_animation(ani, default_mode='reflect')
```

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

Arbitrary "Degree"

The class *splines.quaternion.DeCasteljau* (page 165) allows arbitrary numbers of unit quaternions per segment and therefore arbitrary "degrees":

[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]:
```
ani = animate_rotations(s.evaluate(np.linspace(s.grid[0], s.grid[-1], 100)))
```

[12]:
```
display_animation(ani, default_mode='reflect')
```

```
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"

[13]:
```
from splines import ConstantSpeedAdapter
```

[14]:
```
s1 = DeCasteljau([[
    angles2quat(90, 0, 0),
    angles2quat(0, -45, 90),
    angles2quat(0, 0, 0),
    angles2quat(180, 0, 180),
]])
```

[15]:
```
s2 = ConstantSpeedAdapter(s1)
```

[16]:
```
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)),
})
```

[17]:
```
display_animation(ani, default_mode='reflect')
```

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

Joining Curves

Until now, we have assumed that four control quaternions are given for each "cubic" segment.

If a list of quaternions is given, which is supposed to be interpolated, the intermediate control quaternions can be computed from neighboring quaternions as shown in *the notebook about uniform Catmull–Rom-like quaternion splines* (page 133).

`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

We have seen how to use *de Casteljau's algorithm with Slerp* (page 130) to create "cubic" Bézier-like quaternion curve segments. However, if we only have a sequence of rotations to be interpolated and no additional Bézier control quaternions are provided, it would be great if we could compute the missing control quaternions automatically from neighboring quaternions.

In the *notebook about (uniform) Euclidean Catmull–Rom splines* (page 70) we have already seen how this can be done for splines in Euclidean space:

$$
\begin{aligned}
\tilde{\boldsymbol{x}}_i^{(+)} &= \boldsymbol{x}_i + \frac{\dot{\boldsymbol{x}}_i}{3} \\
\tilde{\boldsymbol{x}}_i^{(-)} &= \boldsymbol{x}_i - \frac{\dot{\boldsymbol{x}}_i}{3}
\end{aligned}
$$

Note that the velocity vectors $\dot{\boldsymbol{x}}_i$ live in the same Euclidean space as the position vectors $\boldsymbol{x}_i$. We can simply add a fraction of a velocity to a position and we get a new position in return.

Applying this to rotations is unfortunately not very straightforward. When unit quaternions are moving along the the unit hypersphere, their velocity vectors are tangential to that hypersphere, which means that the velocity vectors are generally not unit quaternions themselves. Furthermore, adding a tangent vector to a unit quaternion never leads to a unit quaternion as a result.

Instead of using tangent vectors, we can introduce a (yet unknown) *relative quaternion (in the global frame of reference)* (page 123) $q_{i,\text{offset}}$:

$$
\begin{aligned}
\tilde{q}_i^{(+)} &= {q_{i,\text{offset}}}^{\frac{1}{3}} \, q_i \\
\tilde{q}_i^{(-)} &= {q_{i,\text{offset}}}^{-\frac{1}{3}} \, q_i
\end{aligned}
$$

When trying to obtain $q_{i,\text{offset}}$, the problem is that there are many equivalent ways to write the equation for tangent vectors in Euclidean space …

$$\dot{x}_i = \frac{x_{i+1} - x_{i-1}}{2} = \frac{(x_i - x_{i-1}) + (x_{i+1} - x_i)}{2} = \frac{x_i - x_{i-1}}{2} + \frac{x_{i+1} - x_i}{2}$$

... but "translating" them to quaternions will lead to different results!

For the following experiments, let's define three quaternions using the `angles2quat()` function from helper.py:

```
[1]: from helper import angles2quat
```

```
[2]: q3 = angles2quat(0, 0, 0)
     q4 = angles2quat(0, 45, -10)
     q5 = angles2quat(90, 0, -90)
```

Relative Rotations

As a first attempt, we can try to "translate" the equation ...

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

... to unit quaternions like this:

$$q_{i,\text{offset}} \overset{?}{=} \left(q_{i+1} {q_{i-1}}^{-1}\right)^{\frac{1}{2}}$$

```
[3]: offset_a = q3.rotation_to(q5)**(1/2)
```

We'll see later whether that's reasonable or not.

For the next few examples, we define the *relative rotations* (page 123) associated with the the incoming and the outgoing chord:

$$\begin{aligned} q_\text{in} &= q_i {q_{i-1}}^{-1} \\ q_\text{out} &= q_{i+1} {q_i}^{-1} \end{aligned}$$

```
[4]: q_in = q3.rotation_to(q4)
     q_out = q4.rotation_to(q5)
```

The next equation ...

$$\dot{x}_i = \frac{(x_i - x_{i-1}) + (x_{i+1} - x_i)}{2}$$

... can be "translated" to unit quaternions like this:

$$q_{i,\text{offset}} \overset{?}{=} \left(q_\text{out} q_\text{in}\right)^{\frac{1}{2}}$$

```
[5]: offset_b = (q_out * q_in)**(1/2)
```

We can see that this is actually equivalent to the previous one:

```
[6]: max(map(abs, (offset_b - offset_a).xyzw))
[6]: 1.1102230246251565e-16
```

In the Euclidean case, the order doesn't matter, but in the quaternion case . . .

$$q_{i,\text{offset}} \overset{?}{=} (q_\text{in} q_\text{out})^{\frac{1}{2}}$$

```
[7]: offset_c = (q_in * q_out)**(1/2)
```

. . . there is a (quite large!) difference:

```
[8]: max(map(abs, (offset_b - offset_c).xyzw))
[8]: 0.2563304531880035
```

Based on the equation . . .

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

. . . we can try another pair of equations . . .

$$q_{i,\text{offset}} \overset{?}{=} \left({q_\text{out}}^{\frac{1}{2}} {q_\text{in}}^{\frac{1}{2}}\right)$$

```
[9]: offset_d = (q_out**(1/2) * q_in**(1/2))
```

$$q_{i,\text{offset}} \overset{?}{=} \left({q_\text{in}}^{\frac{1}{2}} {q_\text{out}}^{\frac{1}{2}}\right)$$

```
[10]: offset_e = (q_in**(1/6) * q_out**(1/6))
```

. . . but they are also non-symmetric:

```
[11]: max(map(abs, (offset_e - offset_d).xyzw))
[11]: 0.20225984693486293
```

Let's try a slightly more involved variant, where the order of q_in and q_out can actually be reversed:

$$q_{i,\text{offset}} \overset{?}{=} \left(q_\text{out} {q_\text{in}}^{-1}\right)^{\frac{1}{2}} q_\text{in} = \left(q_\text{in} {q_\text{out}}^{-1}\right)^{\frac{1}{2}} q_\text{out}$$

```
[12]: offset_f = (q_out * q_in**-1)**(1/2) * q_in
```

```
[13]: offset_g = (q_in * q_out**-1)**(1/2) * q_out
```

```
[14]: max(map(abs, (offset_g - offset_f).xyzw))
[14]: 1.1102230246251565e-16
```

It is nice to have symmetric behavior, but the curvature of the unit hypersphere still causes an error. We can check that by scaling down the components before the calculation (leading to a smaller curvature) and scaling up the result:

$$q_{i,\text{offset}} \stackrel{?}{=} \left(\left({q_\text{out}}^{\frac{1}{10}} {q_\text{in}}^{-\frac{1}{10}}\right)^{\frac{1}{2}} {q_\text{in}}^{\frac{1}{10}}\right)^{10} = \left(\left({q_\text{in}}^{\frac{1}{10}} {q_\text{out}}^{-\frac{1}{10}}\right)^{\frac{1}{2}} {q_\text{out}}^{\frac{1}{10}}\right)^{10}$$

```
[15]: offset_h = ((q_out**(1/10) * q_in**(-1/10))**(1/2) * q_in**(1/10))**10
```

```
[16]: offset_i = ((q_in**(1/10) * q_out**(-1/10))**(1/2) * q_out**(1/10))**10
```

```
[17]: max(map(abs, (offset_h - offset_i).xyzw))
[17]: 2.1094237467877974e-15
```

```
[18]: offset_j = ((q_out**(1/100) * q_in**(-1/100))**(1/2) * q_in**(1/100))**100
```

```
[19]: offset_k = ((q_in**(1/100) * q_out**(-1/100))**(1/2) * q_out**(1/100))**100
```

```
[20]: max(map(abs, (offset_j - offset_k).xyzw))
[20]: 1.4277468096679513e-13
```

If we choose a larger scaling factor, the the error caused by curvature becomes smaller (as we will see in the next section). However, the numerical error gets bigger (as we can already see in the increasing differences). We cannot scale down the components arbitrarily, but there is a different mathematical tool that we can use, which boils down to the same thing, as we'll see in the next section.

Tangent Space

The *logarithmic map* operation transforms a unit quaternion into a vector that's a member of the tangent space at the identity quaternion (a.k.a. **1**). In this tangent space – which is a three-dimensional Euclidean space – we can add and scale components without worrying about curvature. Using the *exponential map* operation, the result can be projected back onto the unit hypersphere. This way, we can take the equation for the tangent vector in Euclidean space . . .

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

. . . and "translate" it into unit quaternions . . .

$$q_{i,\text{offset}} \stackrel{?}{=} \exp\left(\frac{\ln(q_\text{in}) + \ln(q_\text{out})}{2}\right)$$

```
[21]: from splines.quaternion import UnitQuaternion
```

```
[22]: offset_l = UnitQuaternion.exp_map((q_in.log_map() + q_out.log_map()) / 2)
```

This approach is implemented in the *splines.quaternion.CatmullRom* (page 166) class.

Let's compare this to the variants from the previous section:

```
[23]: max(map(abs, (offset_l - offset_f).xyzw))
[23]: 0.017423237526556339
```

```
[24]: max(map(abs, (offset_l - offset_h).xyzw))
```

```
[24]: 0.00016775844275412
```

```
[25]: max(map(abs, (offset_l - offset_j).xyzw))
```

```
[25]: 1.6769343111344703e-06
```

Increasing the scaling factor from the previous section will get us closer and closer, but only until the numerical errors eventually take over.

Example

After all those more or less successful experiments, let's show an example with actual rotations.

```
[26]: def offset(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)
```

We'll use the *DeCasteljau* (page 165) class to create a Bézier-like curve from the given control points, using *canonicalized()* (page 165) to avoid angles greater than 180 degrees.

```
[27]: from splines.quaternion import DeCasteljau, canonicalized
```

Also, some helper functions from helper.py will come in handy.

```
[28]: from helper import animate_rotations, display_animation
```

We don't want to worry about end conditions here, therefore we create a closed curve.

```
[29]: def create_closed_curve(rotations):
          rotations = list(canonicalized(rotations + rotations[:2]))
          control_points = []
          for q_1, q0, q1 in zip(rotations, rotations[1:], rotations[2:]):
              q_offset = offset(q_1, q0, q1)
              control_points.extend([
                  q_offset**(-1/3) * q0,
                  q0,
                  q0,
                  q_offset**(1/3) * q0])
          control_points = control_points[-2:] + control_points[:-2]
          segments = list(zip(*[iter(control_points)] * 4))
          return DeCasteljau(segments)
```

```
[30]: 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),
      ]
```

```
[31]: s = create_closed_curve(rotations)
```

[32]:
```
import numpy as np
```

[33]:
```
times = np.linspace(0, len(rotations), 200, endpoint=False)
```

[34]:
```
ani = animate_rotations(s.evaluate(times))
```

[35]:
```
display_animation(ani, default_mode='loop')
```

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

Shoemake's Approach

In section 4.2, [Sho85] provides two function definitions:

$$\begin{aligned}\text{Double}(p, q) &= 2(p \cdot q)q - p \\ \text{Bisect}(p, q) &= \frac{p + q}{\|p + q\|}\end{aligned}$$

[36]:
```
def double(p, q):
    return 2 * p.dot(q) * q - p
```

[37]:
```
def bisect(p, q):
    return (p + q).normalized()
```

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):

$$\begin{aligned}a_n &= \text{Bisect}(\text{Double}(q_{n-1}, q_n), q_{n+1}) \\ b_n &= \text{Double}(a_n, q_n)\end{aligned}$$

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

[38]:
```
def shoemake_control_quaternions(q_1, q0, q1):
    """Shoemake's control quaternions.

    Given three key quaternions, return the control quaternions
    preceding and following the middle one.

    Actually, the great arc distance of the returned quaternions to q0
    still has to be reduced to 1/3 of the distance
    to get the proper control quaternions (see the note below).

    """
    a = bisect(double(q_1, q0), q1)
    b = double(a, q0).normalized()
    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 `normalized()` to turn the *Quaternion* (page 163) result into a *UnitQuaternion* (page 164).

```
[39]: b, a = shoemake_control_quaternions(q3, q4, q5)
```

The results are close (but by far not identical) to the tangent space approach from above:

```
[40]: max(map(abs, (a - offset_l * q4).xyzw))
```

```
[40]: 0.013831724198409168
```

```
[41]: max(map(abs, (b - offset_l.inverse() * q4).xyzw))
```

```
[41]: 0.018852903209093046
```

Note

Shoemake's result has to be scaled by $\frac{1}{3}$, just as we did with $q_{i,\text{offset}}$ above:

> 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"

.... `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

> What is the best way to allow varying intervals between sequence points in parameter space?
>
> —[Sho85], section 6: "Questions"

In the *uniform case* (page 133) we have used *de Casteljau's algorithm with Slerp* (page 130) to create a "cubic" rotation spline. To extend this to the non-uniform case, we can transform the parameter $t \to \frac{t-t_i}{t_{i+1}-t_i}$ for each spline segment (as shown in *the notebook about non-uniform Euclidean Bézier splines* (page 53)). This is implemented in the *splines.quaternion.DeCasteljau* (page 165) class.

Assuming the control points at the start and the end of each segment are given (from a sequence of quaternions to be interpolated), we'll also need a way to calculate the missing two control points. For inspiration, we can have a look at the *notebook about non-uniform (Euclidean) Catmull–Rom splines* (page 74) which provides these equations:

$$
\begin{aligned}
\boldsymbol{v}_i &= \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_i}{t_{i+1} - t_i} \\
\dot{\boldsymbol{x}}_i &= \frac{(t_{i+1} - t_i)\,\boldsymbol{v}_{i-1} + (t_i - t_{i-1})\,\boldsymbol{v}_i}{t_{i+1} - t_{i-1}} \\
\tilde{\boldsymbol{x}}_i^{(+)} &= \boldsymbol{x}_i + \frac{(t_{i+1} - t_i)\,\dot{\boldsymbol{x}}_i}{3} \\
\tilde{\boldsymbol{x}}_i^{(-)} &= \boldsymbol{x}_i - \frac{(t_i - t_{i-1})\,\dot{\boldsymbol{x}}_i}{3}
\end{aligned}
$$

With the *relative rotation* (page 123) $\delta_i = q_{i+1} {q_i}^{-1}$ we can try to "translate" this to quaternions (using some vector operations in the tangent space):

$$
\begin{aligned}
\vec{\rho}_i &= \frac{\ln(\delta_i)}{t_{i+1} - t_i}\\
\vec{\omega}_i &= \frac{(t_{i+1} - t_i)\,\vec{\rho}_{i-1} + (t_i - t_{i-1})\,\vec{\rho}_i}{t_{i+1} - t_{i-1}}\\
\tilde{q}_i^{(+)} &\stackrel{?}{=} \exp\left(\frac{t_{i+1} - t_i}{3}\,\vec{\omega}_i\right) q_i\\
\tilde{q}_i^{(-)} &\stackrel{?}{=} \exp\left(\frac{t_i - t_{i-1}}{3}\,\vec{\omega}_i\right)^{-1} q_i,
\end{aligned}
$$

where $\vec{\rho}_i$ is the angular velocity along the great arc from q_i to q_{i+1} within the parameter interval from t_i to t_{i+1} and $\vec{\omega}_i$ is the angular velocity of the Catmull–Rom-like quaternion curve at the control point q_i (which is reached at parameter value t_i). Finally, $\tilde{q}_i^{(-)}$ and $\tilde{q}_i^{(+)}$ are the control quaternions before and after q_i, respectively.

[1]:
```python
from splines.quaternion import UnitQuaternion

def cr_control_quaternions(qs, ts):
    q_1, q0, q1 = qs
    t_1, t0, t1 = ts
    rho_in = q_1.rotation_to(q0).log_map() / (t0 - t_1)
    rho_out = q0.rotation_to(q1).log_map() / (t1 - t0)
    w0 = ((t1 - t0) * rho_in + (t0 - t_1) * rho_out) / (t1 - t_1)
    return [
        UnitQuaternion.exp_map(-w0 * (t0 - t_1) / 3) * q0,
        UnitQuaternion.exp_map(w0 * (t1 - t0) / 3) * q0,
    ]
```

This approach is also implemented in the *splines.quaternion.CatmullRom* (page 166) class.

To illustrate this, let's load NumPy, a few helpers from helper.py and *splines.quaternion.canonicalized()* (page 165).

[2]:
```python
import numpy as np
np.set_printoptions(precision=4)
from helper import angles2quat, animate_rotations, display_animation
from splines.quaternion import canonicalized
```

The following function can create a closed spline using the above method to calculate control quaternions.

[3]:
```python
from splines.quaternion import DeCasteljau

def catmull_rom_curve(rotations, grid):
    """Create a closed Catmull-Rom-like quaternion curve."""
    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, *grid, grid[-1] + first_interval]
```

(continues on next page)

(continued from previous page)

```
    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 = cr_control_quaternions(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)
```

To try this out, we need a few example quaternions and time instances:

```
[4]: rotations1 = [
         angles2quat(0, 0, 180),
         angles2quat(0, 45, 90),
         angles2quat(90, 45, 0),
         angles2quat(90, 90, -90),
         angles2quat(180, 0, -180),
         angles2quat(-90, -45, 180),
     ]
```

```
[5]: grid1 = np.array([0, 0.5, 2, 5, 6, 7, 9])
```

```
[6]: cr = catmull_rom_curve(rotations1, grid1)
```

```
[7]: def evaluate(spline, frames=200):
         times = np.linspace(
             spline.grid[0], spline.grid[-1], frames, endpoint=False)
         return spline.evaluate(times)
```

```
[8]: ani = animate_rotations(evaluate(cr))
```

```
[9]: display_animation(ani, default_mode='loop')
```

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

Parameterization

Instead of choosing arbitrary time intervals between control quaternions (via the `grid` argument), we can calculate time intervals based on the control quaternions themselves.

```
[10]: rotations2 = [
          angles2quat(90, 0, -45),
          angles2quat(179, 0, 0),
          angles2quat(181, 0, 0),
          angles2quat(270, 0, -45),
          angles2quat(0, 90, 90),
      ]
```

We have seen uniform parameterization already in the *previous notebook* (page 133), where each parameter interval is set to 1:

```
[11]: uniform = catmull_rom_curve(rotations2, grid=range(len(rotations2) + 1))
```

For *chordal parameterization of Euclidean splines* (page 59) we used the Euclidean distance as basis for calculating the time intervals. For rotation splines, it makes more sense to use rotation angles, which are proportional to the lengths of the great arcs between control quaternions:

```
[12]: angles = np.array([
          a.rotation_to(b).angle
          for a, b in zip(rotations2, rotations2[1:] + rotations2[:1])])
      angles
[12]: array([1.7027, 0.0349, 1.7027, 2.5936, 1.7178])
```

The values are probably easier to understand when we show them in degrees:

```
[13]: np.degrees(angles)
[13]: array([ 97.5592,   2.    ,  97.5592, 148.6003,  98.4211])

[14]: chordal_grid = np.concatenate([[0], np.cumsum(angles)])

[15]: chordal = catmull_rom_curve(rotations2, grid=chordal_grid)
```

For *centripetal parameterization of Euclidean splines* (page 60) we used the square root of the Euclidean distances, here we use the square root of the rotation angles:

```
[16]: centripetal_grid = np.concatenate([[0], np.cumsum(np.sqrt(angles))])

[17]: centripetal = catmull_rom_curve(rotations2, grid=centripetal_grid)

[18]: ani = animate_rotations({
          'uniform': evaluate(uniform),
          'chordal': evaluate(chordal),
          'centripetal': evaluate(centripetal),
      })

[19]: display_animation(ani, default_mode='loop')
      Animations can only be shown in HTML output, sorry!
```

The *splines.quaternion.CatmullRom* (page 166) class provides a parameter `alpha` that allows arbitrary parameterization between *uniform* and *chordal* (see also *parameterized parameterization of Euclidean splines* (page 61)).

`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

Remember *Kochanek–Bartels splines in Euclidean space* (page 84)? We can try to "translate" those to quaternions by using *de Casteljau's algorithm with Slerp* (page 130). We only need a way to create the appropriate incoming and outgoing control quaternions, similarly to what we did to create *Catmull–Rom-like rotation splines* (page 139).

We are only considering the more general *non-uniform* case here. The *uniform* case can be obtained by simply using time instances t_i with a step size of 1.

In the *notebook about non-uniform Euclidean Kochanek–Bartels splines* (page 95) we showed the following equations for the incoming tangent vector $\dot{\boldsymbol{x}}_i^{(-)}$ and the outgoing tangent vector $\dot{\boldsymbol{x}}_i^{(+)}$ at vertex $\boldsymbol{x}_i$ (which corresponds to the parameter value t_i):

$$\begin{aligned}
a_i &= (1 - T_i)(1 + C_i)(1 + B_i)\\
b_i &= (1 - T_i)(1 - C_i)(1 - B_i)\\
c_i &= (1 - T_i)(1 - C_i)(1 + B_i)\\
d_i &= (1 - T_i)(1 + C_i)(1 - B_i)
\end{aligned}$$

$$\begin{aligned}
\dot{\boldsymbol{x}}_i^{(+)} &= \frac{a_i(t_{i+1} - t_i)\,\boldsymbol{v}_{i-1} + b_i(t_i - t_{i-1})\,\boldsymbol{v}_i}{t_{i+1} - t_{i-1}}\\
\dot{\boldsymbol{x}}_i^{(-)} &= \frac{c_i(t_{i+1} - t_i)\,\boldsymbol{v}_{i-1} + d_i(t_i - t_{i-1})\,\boldsymbol{v}_i}{t_{i+1} - t_{i-1}},
\end{aligned}$$

where $\boldsymbol{v}_i = \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_i}{t_{i+1} - t_i}$.

Given those tangent vectors, we know the equations for the incoming control value $\tilde{\boldsymbol{x}}_i^{(-)}$ and the outgoing control value $\tilde{\boldsymbol{x}}_i^{(+)}$ from the *notebook about non-uniform Euclidean Catmull–Rom splines* (page 74):

$$\begin{aligned}
\tilde{\boldsymbol{x}}_i^{(+)} &= \boldsymbol{x}_i + \frac{(t_{i+1} - t_i)}{3}\dot{\boldsymbol{x}}_i^{(+)}\\
\tilde{\boldsymbol{x}}_i^{(-)} &= \boldsymbol{x}_i - \frac{(t_i - t_{i-1})}{3}\dot{\boldsymbol{x}}_i^{(-)}
\end{aligned}$$

We can try to “translate” those equations to quaternions (using some vector operations in the tangent space):

$$\begin{aligned}
\vec{\rho}_i &= \frac{\ln(\delta_i)}{t_{i+1} - t_i}\\
\vec{\omega}_i^{(+)} &= \frac{a_i(t_{i+1} - t_i)\,\vec{\rho}_{i-1} + b_i(t_i - t_{i-1})\,\vec{\rho}_i}{t_{i+1} - t_{i-1}}\\
\vec{\omega}_i^{(-)} &= \frac{c_i(t_{i+1} - t_i)\,\vec{\rho}_{i-1} + d_i(t_i - t_{i-1})\,\vec{\rho}_i}{t_{i+1} - t_{i-1}}\\
\tilde{q}_i^{(+)} &\overset{?}{=} \exp\left(\frac{t_{i+1} - t_i}{3}\,\vec{\omega}_i^{(+)}\right) q_i\\
\tilde{q}_i^{(-)} &\overset{?}{=} \exp\left(\frac{t_i - t_{i-1}}{3}\,\vec{\omega}_i^{(-)}\right)^{-1} q_i,
\end{aligned}$$

where $\delta_i = q_{i+1} q_i^{-1}$ is the *relative rotation* (page 123) from q_i to q_{i+1}, $\vec{\rho}_i$ is the angular velocity along the great arc from q_i to q_{i+1} within the parameter interval from t_i to t_{i+1}, $\vec{\omega}_i^{(-)}$ is the incoming angular velocity of the Kochanek–Bartels-like quaternion curve at the control point q_i (which is reached at parameter value t_i) and $\vec{\omega}_i^{(+)}$ is the outgoing angular velocity. Finally, $\tilde{q}_i^{(-)}$ and $\tilde{q}_i^{(+)}$ are the control quaternions before and after q_i, respectively.

A Python implementation of these equations is available in the *splines.quaternion.KochanekBartels* (page 166) class.

[1]:
```
from splines.quaternion import KochanekBartels
```

Examples

This is all a bit abstract, so let's try a few of those TCB values to see their influence on the rotation spline.

For comparison, you can have a look at the *examples for Euclidean Kochanek–Bartels splines* (page 84).

As so often, we import NumPy and a few helpers from helper.py:

[2]:
```
import numpy as np
from helper import angles2quat, animate_rotations, display_animation
```

We define a few example rotations . . .

[3]:
```
rotations = [
    angles2quat(0, 0, 0),
    angles2quat(90, 0, -45),
    angles2quat(-45, 45, -90),
    angles2quat(135, -35, 90),
    angles2quat(90, 0, 0),
]
```

. . . and a helper function that allows us to try out different TCB values:

[4]:
```
def show_tcb(tcb):
    """Show an animation of rotations with the given TCB values."""
    if not isinstance(tcb, dict):
        tcb = {'': tcb}
    result = {}
    for name, tcb in tcb.items():
        s = KochanekBartels(
            rotations,
            alpha=0.5,
            endconditions='closed',
            tcb=tcb,
        )
        times = np.linspace(s.grid[0], s.grid[-1], 100, endpoint=False)
        result[name] = s.evaluate(times)
    display_animation(animate_rotations(result))
```

When using the default TCB values, a Catmull–Rom-like spline is generated:

[5]:
```
show_tcb([0, 0, 0])
```

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

We can vary *tension* (T) . . .

[6]:
```
show_tcb({
    'T = 1': [1, 0, 0],
    'T = 0.5': [0.5, 0, 0],
    'T = -0.5': [-0.5, 0, 0],
    'T = -1': [-1, 0, 0],
})
```

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

… *continuity* (C) …

```
[7]: show_tcb({
         'C = -1': [0, -1, 0],
         'C = -0.5': [0, -0.5, 0],
         'C = 0.5': [0, 0.5, 0],
         'C = 1': [0, 1, 0],
     })
```

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

… and *bias* (B):

```
[8]: show_tcb({
         'B = 1': [0, 0, 1],
         'B = 0.5': [0, 0, 0.5],
         'B = -0.5': [0, 0, -0.5],
         'B = -1': [0, 0, -1],
     })
```

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

Using the largest *tension* value ($T = 1$) produces the same rotations as using the smallest *continuity* value ($C = -1$). However, the timing is different. With large tension values, rotation slows down close to the control points. With small continuity, angular velocity varies less.

```
[9]: show_tcb({
         'T = 1': [1, 0, 0],
         'C = -1': [0, -1, 0],
     })
```

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

Just like in the Euclidean case, $B = -1$ followed by $B = 1$ can be used to create linear (i.e. *Slerp* (page 125)) segments.

```
[10]: show_tcb({
          'Catmull-Rom': [0, 0, 0],
          '2 linear segments': [
              (0, 0, 1),
              (0, 0, 0),
              (0, 0, -1),
              (0, 0, 1),
              (0, 0, -1),
          ],
          'C = -1': [0, -1, 0],
      })
```

```
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

In the *notebook about "natural" end conditions for Euclidean splines* (page 106) we have derived the following equations for calculating the second and penultimate control points of cubic Bézier splines:

$$\begin{aligned}\tilde{\boldsymbol{x}}_0^{(+)} &= \frac{\boldsymbol{x}_0 + \tilde{\boldsymbol{x}}_1^{(-)}}{2}\\ \tilde{\boldsymbol{x}}_{N-1}^{(-)} &= \frac{\boldsymbol{x}_{N-1} + \tilde{\boldsymbol{x}}_{N-2}^{(+)}}{2}\end{aligned}$$

These equations can be "translated" to quaternions like this:

$$\begin{aligned}\tilde{q}_0^{(+)} &= \left(\tilde{q}_1^{(-)} {q_0}^{-1}\right)^{\frac{1}{2}} q_0\\ \tilde{q}_{N-1}^{(-)} &= \left(\tilde{q}_{N-2}^{(+)} {q_{N-1}}^{-1}\right)^{\frac{1}{2}} q_{N-1}\end{aligned}$$

When considering that the control polygon starts with the quaternions $\left(q_0, \tilde{q}_0^{(+)}, \tilde{q}_1^{(-)}, q_1, \tilde{q}_1^{(+)}, \ldots\right)$ and ends with $\left(\ldots, q_{N-2}, \tilde{q}_{N-2}^{(+)}, \tilde{q}_{N-1}^{(-)}, q_{N-1}\right)$, we can see that the equations are symmetrical. The resulting control quaternion is calculated as the rotation half-way between the first and third control quaternion, counting either from the beginning (q_0) or the end (q_{N-1}) of the spline.

```
[1]: def natural_end_condition(first, third):
         """Return second control quaternion given the first and third.

         This also works when counting from the end of the spline.

         """
         return first.rotation_to(third)**(1 / 2) * first
```

Examples

Let's first import NumPy, a few helpers from helper.py and the *splines.quaternion.DeCasteljau* (page 165) class:

```
[2]: import numpy as np
     from helper import angles2quat, animate_rotations, display_animation
     from splines.quaternion import DeCasteljau
```

Furthermore, let's define a helper function for evaluating a single spline segment:

```
[3]: def calculate_rotations(control_quaternions):
         times = np.linspace(0, 1, 50)
         return DeCasteljau(
             segments=[control_quaternions],
         ).evaluate(times)
```

```
[4]: q0 = angles2quat(45, 0, 0)
     q1 = angles2quat(-45, 0, 0)
```

[5]:
```
q1_control = angles2quat(-45, 0, -90)
```

[6]:
```
ani = animate_rotations({
    'natural begin': calculate_rotations(
        [q0, natural_end_condition(q0, q1_control), q1_control, q1]),
})
```

[7]:
```
display_animation(ani, default_mode='reflect')
```

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

[8]:
```
q0_control = angles2quat(45, 0, 90)
```

[9]:
```
ani = animate_rotations({
    'natural end': calculate_rotations(
        [q0, q0_control, natural_end_condition(q1, q0_control), q1]),
})
```

[10]:
```
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 With Slerp

We can try to use the *Barry–Goldman algorithm for non-uniform Euclidean Catmull–Rom splines* (page 75) using *Slerp* (page 125) instead of linear interpolations, just as we have done with *De Casteljau's algorithm* (page 130).

[1]:
```
def slerp(one, two, t):
    """Spherical Linear intERPolation."""
    return (two * one.inverse())**t * one
```

[2]:
```
def barry_goldman(rotations, times, t):
    """Calculate a spline segment with the Barry-Goldman algorithm.

    Four quaternions and the corresponding four time values
    have to be specified.  The resulting spline segment is located
    between the second and third quaternion.  The given time *t*
    must be between the second and third time value.

    """
    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)),
```

(continues on next page)

(continued from previous page)

```
            slerp(q2, q3, (t - t2) / (t3 - t2)),
            (t - t1) / (t3 - t1)),
        (t - t1) / (t2 - t1))
```

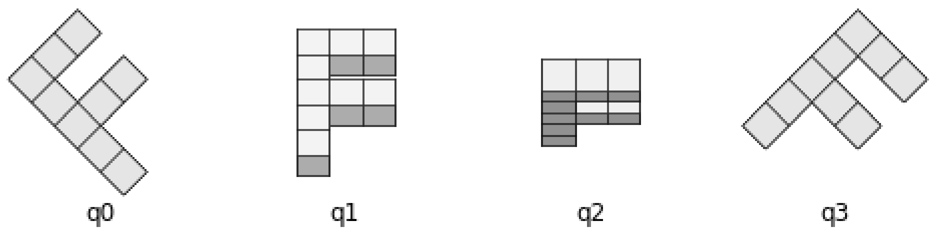
To illustrate this, let's import NumPy and a few helpers from helper.py:

[3]:
```
import numpy as np
from helper import angles2quat, plot_rotation, plot_rotations
from helper import animate_rotations, display_animation
```

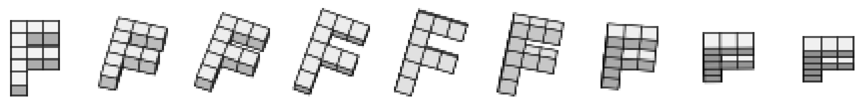
[4]:
```
q0 = angles2quat(45, 0, 0)
q1 = angles2quat(0, -40, 0)
q2 = angles2quat(0, 70, 0)
q3 = angles2quat(-45, 0, 0)
```

[5]:
```
t0 = 0
t1 = 1
t2 = 5
t3 = 8
```

[6]:
```
plot_rotation({'q0': q0, 'q1': q1, 'q2': q2, 'q3': q3});
```



[7]:
```
plot_rotations([
    barry_goldman([q0, q1, q2, q3], [t0, t1, t2, t3], t)
    for t in np.linspace(t1, t2, 9)
], figsize=(8, 1))
```



[8]:
```
ani = animate_rotations([
    barry_goldman([q0, q1, q2, q3], [t0, t1, t2, t3], t)
    for t in np.linspace(t1, t2, 50)
])
```

[9]:
```
display_animation(ani, default_mode='reflect')
```

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

For the next example, we use the *splines.quaternion.BarryGoldman* (page 166) class:

[10]:
```
from splines.quaternion import BarryGoldman
```

[11]:
```
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),
]
```

[12]:
```
bg1 = BarryGoldman(rotations, alpha=0.5)
```

For comparison, we also create a *Catmull–Rom-like quaternion spline* (page 139) using the class *splines.quaternion.CatmullRom* (page 166):

[13]:
```
from splines.quaternion import CatmullRom
```

[14]:
```
cr1 = CatmullRom(rotations, alpha=0.5, endconditions='closed')
```

[15]:
```
def evaluate(spline, frames=200):
    times = np.linspace(
        spline.grid[0], spline.grid[-1], frames, endpoint=False)
    return spline.evaluate(times)
```

[16]:
```
ani = animate_rotations({
    'Barry-Goldman': evaluate(bg1),
    'Catmull-Rom-like': evaluate(cr1),
})
display_animation(ani, default_mode='loop')
```

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

Don't worry if you don't see any difference, the two are indeed extremely similar:

[17]:
```
max(max(map(abs, q.xyzw)) for q in (evaluate(bg1) - evaluate(cr1)))
```

[17]:
```
0.0026694474661510537
```

However, when different time values are chosen, the difference between the two can become significantly bigger.

[18]:
```
grid = 0, 0.5, 1, 5, 6, 7, 10
```

[19]:
```
bg2 = BarryGoldman(rotations, grid)
cr2 = CatmullRom(rotations, grid, endconditions='closed')
```

[20]:
```
ani = animate_rotations({
    'Barry-Goldman': evaluate(bg2),
    'Catmull-Rom-like': evaluate(cr2),
})
display_animation(ani, default_mode='loop')
```

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

Constant Angular Speed

A big advantage of de Casteljau's algorithm is that when evaluating a spline at a given parameter value, it directly provides the appropriate tangent vector. When using the Barry–Goldman algorithm, the tangent vector has to be calculated separately, which make re-parameterization for constant angular speed very inefficient.

[21]:
```
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))
```

[22]:
```
from splines import ConstantSpeedAdapter
```

[23]:
```
bg3 = ConstantSpeedAdapter(BarryGoldmanWithDerivative(rotations, alpha=0.5))
```

Warning

Evaluating this spline takes a long time!

[24]:
```
%%time
bg3_evaluated = evaluate(bg3)
```

```
CPU times: user 1min 33s, sys: 17.7 ms, total: 1min 33s
Wall time: 1min 33s
```

[25]:
```
ani = animate_rotations({
    'non-constant speed': evaluate(bg1),
    'constant speed': bg3_evaluated,
})
```

[26]:
```
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 39)), 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

$$\begin{aligned}\Delta p_i &= p_i - p_{i-1}\\ \alpha_i(t) &= \begin{cases} 0 & t < i - 1\\ t - i + 1 & i - 1 \le t < i\\ 1 & t \ge i.\end{cases}\end{aligned}$$

[1]:
```python
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 $\alpha_i(t)$:

$$\alpha_i(t) = \begin{cases} 0 & t < i\\ t - i & i \le t < i + 1\\ 1 & t \ge i + 1.\end{cases}$$

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])
```

```
[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 53).

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 160):

[11]:
```python
from splines import Bernstein
```

... and create a simple helper function to sum them up:

[12]:
```python
from itertools import accumulate
```

[13]:
```python
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]:
```python
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]:
```python
times = np.linspace(0, 1, 100)
```

[16]:
```python
rotations = [cumulative_bezier(qs, t) for t in times]
```

[17]:
```python
ani = animate_rotations(rotations)
```

[18]:
```python
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 130). An implementation is available in the *splines.quaternion.DeCasteljau* (page 165) 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,
})
```

[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.

The following section was generated from doc/rotation/naive-4d-interpolation.ipynb

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 161) 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)).normalized()
          for x, y, z, w in interpolated_xyzw]
```

For comparison, we also create a *splines.quaternion.CatmullRom* (page 166) instance:

```
[11]: spherical_cr = splines.quaternion.CatmullRom(rotations, endconditions='closed')
```

```
[12]: ani = animate_rotations({
          'normalized 4D interp.': normalized,
          'spherical interp.': spherical_cr.evaluate(times),
      })
      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,
      })
      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 60).

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)).normalized()
          for x, y, z, w in s2.evaluate(times2)]
```

```
[17]: ani = animate_rotations({
          'uniform': normalized,
          'centripetal': normalized2,
```

(continues on next page)

(continued from previous page)

```
})
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 162):

[18]:
```
s3 = splines.ConstantSpeedAdapter(s2)
times3 = np.linspace(s3.grid[0], s3.grid[-1], len(times))
```

[19]:
```
normalized3 = [
    Quaternion(w, (x, y, z)).normalized()
    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({
    'const. 4D speed': normalized3,
    'const. angular speed': s4.evaluate(times4),
})
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 161) to interpolate the Euler angles independently and *splines.quaternion.CatmullRom* (page 166) 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),
    })
    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 159)	Piecewise polynomial curves (in Euclidean space).
`splines.quaternion` (page 163)	Quaternions and unit-quaternion splines.

3.1 splines

Piecewise polynomial curves (in Euclidean space).

Submodules

`quaternion` (page 163)	Quaternions and unit-quaternion splines.

Classes

`Bernstein` (page 160)(segments[, grid])	Piecewise Bézier curve using Bernstein basis.
`CatmullRom` (page 161)(vertices[, grid, alpha, ...])	Catmull--Rom spline.
`ConstantSpeedAdapter` (page 162)(curve)	Re-parameterize a spline to have constant speed.
`CubicHermite` (page 160)(vertices, tangents[, grid])	Cubic Hermite curve.
`KochanekBartels` (page 161)(vertices[, grid, tcb, ...])	Kochanek--Bartels spline.
`Monomial` (page 159)(segments, grid)	Piecewise polynomial curve using monomial basis.
`MonotoneCubic` (page 162)(values, *args, **kwargs)	Monotone cubic curve.
`Natural` (page 161)(vertices[, grid, alpha, endconditions])	Natural spline.
`NewGridAdapter` (page 162)(curve[, new_grid])	Re-parameterize a spline with new grid values.
`PiecewiseMonotoneCubic` (page 162)(values[, grid, ...])	Piecewise monotone cubic curve.

class splines.**Monomial**(*segments, grid*)

Bases: `object`[47]

Piecewise polynomial curve using monomial basis.

See *Polynomial Parametric Curves* (page 2).

Coefficients can have arbitrary dimension. An arbitrary polynomial degree d can be used by specifying $d + 1$ coefficients per segment. The i-th segment is evaluated using this equation:

$$\boldsymbol{p}_i(t) = \sum_{k=0}^{d} \boldsymbol{a}_{i,k} \left(\frac{t - t_i}{t_{i+1} - t_i} \right)^k \text{ for } t_i \leq t < t_{i+1}$$

This is similar to `scipy.interpolate.PPoly`[48], 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."

This shouldn't be a problem since most commonly splines of degree 3 (i.e. cubic splines) are used.

Parameters

- **`segments`** – Sequence of polynomial segments. Each segment $\boldsymbol{a}_i$ contains coefficients for the monomial basis (in order of decreasing degree). Different segments can have different polynomial degree.
- **`grid`** – Sequence of parameter values t_i corresponding to segment boundaries. Must be strictly increasing.

`evaluate`(*t*, *n=0*)
Get value (or n-th derivative) at given parameter value(s) t.

`class splines.`**`Bernstein`**(*segments*, *grid=None*)
Bases: `object`[49]

Piecewise Bézier curve using Bernstein basis.

See *Bézier Splines* (page 39).

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 159)

Cubic Hermite curve.

See *Hermite Splines* (page 13).

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]])`

[47] https://docs.python.org/3/library/functions.html#object
[48] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PPoly.html#scipy.interpolate.PPoly
[49] https://docs.python.org/3/library/functions.html#object

class splines.**CatmullRom**(*vertices*, *grid=None*, ***, *alpha=None*, *endconditions='natural'*)

Bases: `splines.CubicHermite` (page 160)

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 160) spline.

See *Catmull–Rom Splines* (page 55).

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.**KochanekBartels**(*vertices*, *grid=None*, ***, *tcb=(0, 0, 0)*, *alpha=None*, *endconditions='natural'*)

Bases: `splines.CubicHermite` (page 160)

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 160)

Natural spline.

See *Natural Splines* (page 29).

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 161)

Piecewise monotone cubic curve.

See *Piecewise Monotone Interpolation* (page 107).

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 162)

Monotone cubic curve.

This takes the same arguments as `PiecewiseMonotoneCubic` (page 162) (except `closed`), but it raises an error if the given values are not montone.

See *Monotone Interpolation* (page 114).

`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`[50]

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`[51]

Re-parameterize a spline with new grid values.

Parameters

- **curve** – A spline.

[50] https://docs.python.org/3/library/functions.html#object

- **`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*)

3.2 splines.quaternion

Quaternions and unit-quaternion splines.

Functions

`canonicalized` (page 165)(quaternions)	Iterator adapter to ensure minimal angles between *quaternions*.
`slerp` (page 165)(one, two, t)	Spherical Linear intERPolation.

Classes

`BarryGoldman` (page 166)(quaternions[, grid, alpha])	Rotation spline using Barry--Goldman algorithm.
`CatmullRom` (page 166)(quaternions[, grid, alpha, ...])	Catmull--Rom-like rotation spline.
`DeCasteljau` (page 165)(segments[, grid])	Spline using De Casteljau's algorithm with *`slerp()`* (page 165).
`KochanekBartels` (page 166)(quaternions[, grid, tcb, ...])	Kochanek--Bartels-like rotation spline.
`PiecewiseSlerp` (page 165)(quaternions, *[, grid, closed])	Piecewise Slerp.
`Quaternion` (page 163)(scalar, vector)	A very simple quaternion class.
`UnitQuaternion` (page 164)()	Unit quaternion.

`class` `splines.quaternion.`**`Quaternion`**(*scalar*, *vector*)

Bases: `object`[52]

A very simple quaternion class.

This is the base class for the more relevant *`UnitQuaternion`* (page 164) class.

See the *notebook about quaternions* (page 119).

`property scalar`

The scalar part (a.k.a. real part) of the quaternion.

`property vector`

The vector part (a.k.a. imaginary part) of the quaternion.

`conjugate`()

Return quaternion with same *`scalar`* (page 163) part, negated *`vector`* (page 163) part.

`normalized`()

Return quaternion with same 4D direction but unit *`norm`* (page 164).

[51] https://docs.python.org/3/library/functions.html#object

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 noncommutative).

property norm
Length of the quaternion in 4D space.

property xyzw
Components of the quaternion, `scalar` (page 163) last.

property wxyz
Components of the quaternion, `scalar` (page 163) first.

class splines.quaternion.**UnitQuaternion**
Bases: `splines.quaternion.Quaternion` (page 163)

Unit quaternion.

See the *section about unit quaternions* (page 120).

classmethod from_axis_angle(*axis*, *angle*)
Create a unit quaternion from a rotation `axis` (page 164) and `angle` (page 164).

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` (page 163) 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 163).

classmethod exp_map(*value*)
Exponential map from R^3 to unit quaternions.

This is the inverse operation to `log_map()` (page 164).

Parameters **value** (*3-tuple*) – Element of the tangent space at the quaternion identity.

log_map()
Logarithmic map from unit quaternions to R^3.

Returns Corresponding vector in the tangent space at the quaternion identity.

property axis
The (normalized) rotation axis.

property angle
The rotation angle in radians.

[52] https://docs.python.org/3/library/functions.html#object

`rotation_to`(*other*)

Rotation required to rotate *self* into *other*.

See *Relative Rotation (Global Frame of Reference)* (page 123).

Parameters **`other`** (`UnitQuaternion` (page 164)) – Target rotation.

Returns Relative rotation (as *`UnitQuaternion`* (page 164)).

`rotate_vector`(*v*)

Apply rotation to a 3D vector.

Parameters **`v`** (*`3-tuple`*) – A vector in R^3.

Returns The rotated vector.

`splines.quaternion.`**`slerp`**(*one*, *two*, *t*)

Spherical Linear intERPolation.

See *Spherical Linear Interpolation (Slerp)* (page 125).

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`[53]

Piecewise Slerp.

See *Piecewise Slerp* (page 128).

Parameters

- **`quaternions`** – Sequence of rotations to be interpolated. The quaternions will be *`canonicalized()`* (page 165).
- **`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`[54]

Spline using De Casteljau's algorithm with *`slerp()`* (page 165).

See *the corresponding notebook* (page 130) 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, . . .).

[53] https://docs.python.org/3/library/functions.html#object

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 165)

Kochanek–Bartels-like rotation spline.

See *the corresponding notebook* (page 142) for details.

Parameters

- **quaternions** – Sequence of rotations to be interpolated. The quaternions will be *canonicalized()* (page 165).
- **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

  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 166)

Catmull–Rom-like rotation spline.

This is just *KochanekBartels* (page 166) without TCB values.

See *Uniform Catmull–Rom-Like Quaternion Splines* (page 133) and *Non-Uniform Catmull–Rom-Like Rotation Splines* (page 139).

class splines.quaternion.**BarryGoldman**(*quaternions*, *grid=None*, ***, *alpha=None*)

Bases: `object`[55]

Rotation spline using Barry–Goldman algorithm.

Always closed (for now).

evaluate(*t*)

[54] https://docs.python.org/3/library/functions.html#object
[55] https://docs.python.org/3/library/functions.html#object

4 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[56].

[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[57].

[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[58].

[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[59].

[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[60].

[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[61].

[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[62].

[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[63].

[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[64].

[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[65].

[McD10] John McDonald. Teaching quaternions is not complex. *Computer Graphics Forum*, 29(8):2447–2455, 2010. doi:10.1111/j.1467-8659.2010.01756.x[66].

[56] https://doi.org/10.1145/54852.378511
[57] https://doi.org/10.1016/B978-0-12-079050-0.50020-5
[58] https://doi.org/10.1016/0021-9045(72)90080-9
[59] https://doi.org/10.1090/S0025-5718-1989-0962209-1
[60] https://doi.org/10.2172/6838406
[61] https://doi.org/10.1137/0905021
[62] https://doi.org/10.1137/0717021
[63] https://doi.org/10.1016/B978-0-12-079050-0.50011-4
[64] https://doi.org/10.1145/218380.218486
[65] https://doi.org/10.1145/800031.808575
[66] https://doi.org/10.1111/j.1467-8659.2010.01756.x

[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[67].

[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[68].

[67] https://doi.org/10.1145/325165.325242
[68] https://doi.org/10.1016/j.cad.2010.08.008