# Describing Three-Dimensional Movements in an Audio Scene Authoring Format

Dissertation
zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.),
der Fakultät für Informatik und Elektrotechnik
der Universität Rostock
vorgelegt von
Matthias Geier, geb. am 12. Feb. 1979 in Salzburg.

Rostock, Juni 2023

https://doi.org/10.18453/rosdok_id00004604

**Gutachter:**
Prof. Dr.-Ing. Sascha Spors, Universität Rostock
Prof. Dr.-Ing. Thomas Sporer, Universität der Künste Berlin
Prof. Dr.-Ing. Alexander Raake, TU Ilmenau

**Jahr der Einreichung:** 2023
**Jahr der Verteidigung:** 2024

## Abstract

After a brief historic overview about *spatial audio reproduction*, the concept of *object-based* audio reproduction is explained and the need for *spatial audio scenes* is stated. Several existing formats for describing object-based audio scenes are reviewed, with special focus on the description of movement of scene objects over time. A new scene authoring format named Audio Scene Description Format (ASDF) is presented. Its description of movement of scene objects is based on several types of *splines*, which are thoroughly investigated, both for position and for rotation. Finally, an open-source ASDF library implementation and two integrations of this library are presented, which make it possible for everyone to try the ASDF right now.

## Zusammenfassung

Nach einem kurzen Abriss über die Geschichte der *räumlichen Audiowiedergabe* wird das Konzept der *objektbasierten* Audiowiedergabe erklärt und die Notwendigkeit von *räumlichen Audioszenen* wird festgestellt. Einige existierende Beschreibungsformate für objektbasierte Audioszenen werden betrachtet, mit Hauptaugenmerk auf die Beschreibung der Bewegung von Szenenobjekten im Zeitverlauf. Ein neues Format für Szenenautoren namens Audio Scene Description Format (ASDF) wird präsentiert. Seine Beschreibung der Szenenobjektbewegungen fußt auf mehreren Arten von *Splines*, die gründlich untersucht werden, sowohl für Position als auch für Rotation. Zu guter Letzt wird die Implementierung einer quelloffenen ASDF Softwarebibliothek sowie zwei Einbindungen dieser Bibliothek präsentiert, die es ab sofort jedem ermöglichen, das ASDF auszuprobieren.

# Contents

It has been found that by artificially causing the source of sound to move rapidly in space the result can be highly dramatic and desirable.

*(Garity and Hawkins 1941)*

# Introduction

The goal of *spatial audio reproduction* is to create auditory events for a listener or multiple listeners, which are perceived as arriving from specific spatial directions. This can be achieved either with headphones or with arrangements of multiple loudspeakers. Such technology can be used, for example, for creating spatial music performances, audio plays or spatial sound tracks for movies.

To set everything into perspective, chapter 1 will give an overview about the historic development of spatial audio reproduction and present some old and new techniques for creating spatial audio experiences for an audience. This is also where the term *channel-based* will be introduced to classify reproduction systems where each loudspeaker driving signal is stored and distributed in its own separate channel. This will be contrasted with the term *object-based*, which represents a whole new reproduction paradigm. Instead of delivering an *audio mix* which is tied to a fixed loudspeaker setup (or to two headphone channels), a so-called *audio scene* comprised of individual sound sources is created which holds all source signals as well as additional data like source positions and other scene parameters. Based on such an audio scene, the loudspeaker or headphone signals are generated in real time for any given reproduction setup.

Chapter 2 will describe some existing file formats to store object-based audio scenes. The main focus will be on the handling of three-dimensional movements of scene objects. Most of those formats were not explicitly designed for authoring, and even though many formats are text-based, their syntaxes make it unnecessarily hard to author scenes by hand. Most formats do not provide any high-level *declarative*[1] syntax for the description of smooth movements.

The Audio Scene Description Format (ASDF) – a new authoring format for object-based audio scenes – will be presented in chapter 3. Instead of using spatial relationships as the main structural element, it uses temporal relationships. Elements of a scene can happen in parallel or in sequence. By nesting so-called *time containers*, arbitrary time relationships can be defined. Both audio clips and the *transforms* that control their spatial positions and movements are part of the same timeline. Spatial relations are still very important, but they are not defined by the top-level file structure. Instead, spatial relationships have to be explicitly established by referencing other scene objects or transforms via their IDs.

A detailed description of all features of the ASDF is available in appendix A, which also includes a lot of small examples. The *declarative* definition of movements of objects in a scene – including the rotation of objects and groups of objects – is based on *splines*. An extensive review of all the relevant types of splines is provided in appendix B. The appendices are an integral part of this thesis and they should not be overlooked. The reason why those two appendices are not part of the main text is that they are actually self-contained projects, which are separately available online. Furthermore, they are meant to evolve beyond being used as part of this thesis. Appendix B not only illustrates the properties of all the types of splines used in the ASDF, but it also provides tools for investigating further types and maybe developing new types. It thoroughly derives the fundamentals of interpolatory polynomial splines in Euclidean space and it applies the same methods – as far as possible – to rotation splines. Three-dimensional rotation splines are notoriously hard to visualize and since this thesis is printed on paper, only sequences of snapshots

[1]For an explanation of the term *declarative* see section 2.1.1.

of rotations can be shown. However, in the online HTML version[2], a number of animations are available that can much better illustrate the behavior of subtly different rotation splines.

It is an important goal of this thesis to not only define a theoretical file format on paper, but also to enable its practical usage in order to be able to properly evaluate its capabilities and weaknesses. Therefore, an open-source software library has been implemented, which is described in chapter 4. This library has also been integrated in a stand-alone software for spatial audio reproduction, which means that the ASDF is ready to be tried out by anyone who is interested.

## Acknowledgements

I would like to thank Sebastian Möller, head of the Quality and Usability (QU) Lab, Technische Universität Berlin[3], for taking me in as a PhD student and for giving me all the support I could wish for during the time I worked there. I am indebted to Sascha Spors, at the time head of the spatial audio group at QU Lab and later professor at the Institute of Communications Engineering at Universität Rostock[4], for acting as my thesis advisor since the beginning and for his support over all those years. Thanks to Jens Ahrens for introducing me to the QU Lab in the first place and for his continuous collaboration on the *SoundScape Renderer* which now serves as a proving ground for the ASDF. Thanks to all my colleagues at QU Lab in Berlin and at the Institute of Communications Engineering in Rostock for many fruitful discussions and for contributing to a great working experience. I would like to thank Jürgen Herre for helping me find the motivation to finally finish this thesis. A big thank you goes to Tracy Harris for proofreading the whole text including the appendices. Thanks to Mikhail Korotiaev for several suggestions for improvements in appendix B. Finally, I would like to thank all the maintainers and contributors of the many open-source software projects that were used for creating this thesis and the associated software and documentation: LaTeX, Sphinx, Jupyter, Matplotlib, NumPy, SymPy, Python and Rust, just to name a few. Specifically, I would like to thank Jean-François B.[5] for his work on the LaTeX output of *nbsphinx*[6], without which the Jupyter code cells in the appendix would look much less pleasing.

[2]`https://splines.readthedocs.io/`
[3]`https://www.tu.berlin/qu/`
[4]`https://www.int.uni-rostock.de/`
[5]`https://github.com/jfbu`
[6]`https://nbsphinx.readthedocs.io/`

# Chapter 1

# Spatial Audio From the Beginning

There were sound waves in the early universe, as can be observed today via *baryon acoustic oscillations*[1], and since that era there have been – and still are – many places in the universe which are filled with an elastic medium that allows sound propagation. This chapter will focus on sound waves in the atmosphere of planet Earth. However, any sound travelling through any medium that inhabits any three-dimensional space can be considered *spatial sound*.

The terms *spatial sound* and *spatial audio* can be used interchangeably. Etymologically, the term *audio*, which is the first-person singular form of the verb *audire* – Latin for *hearing/listening* – implies a sentient being capable of auditory perception (and, strictly speaking, with Latin language skills) actually hearing the sound.

Even before life existed on Earth, its atmosphere (and its oceans as well) must have carried spatial sound, caused, for example, by storms, meteor strikes or volcanic eruptions (see figure 1).

no initiator
natural occurrence
no observer

**Figure 1:** *Spatial sound produced by inanimate processes in a lifeless medium*

Animals first evolved and diversified in the oceans, where many of them developed an auditory sense. When the first animals migrated to land, they adapted their spatial hearing capabilities to the new atmospheric medium or developed entirely new auditory mechanisms. The development of hearing – especially spatial hearing – certainly was (and still is) a very helpful tool for the survival of a species. It can help predators find prey, but it can also help prey evade predators (see figure 2).

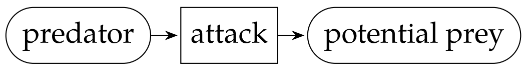


**Figure 2:** *Spatial sound unintentionally produced by a predator*

Apart from sound as an accidental by-product of locomotion and bodily functions, animals have developed a wide variety of ways to actively create sounds, serving a multitude of purposes. Spatial sound can help finding a mate, localizing one's own offspring in an overcrowded breeding colony and in general with intra- and inter-species communication. In addition to mammals, several classes of animals are known to be capable of spatial hearing, like fish (Popper and Fay 1993), birds (Konishi 2003; MacLeod et al. 2006) and even insects (Schmidt and Römer 2011; Yager 1999). A notable exception is the *praying mantis*, which has an acoustic sense but only a single ear, making it an *auditory cyclops* which can most likely not differentiate between different angles of sound incidence (Yager and Hoy 1986).

[1] `https://en.wikipedia.org/wiki/Baryon_acoustic_oscillations`

Whenever an individual produces sound and another individual (or even the same one) perceives it in a way that spatial information is conveyed, we can consider it a *spatial audio performance* (see figure 3).

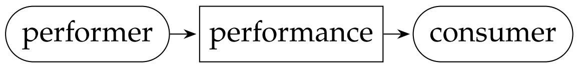


**Figure 3:** *Spatial audio performance*

Like all diagrams in this section, the diagram in figure 3 is simplified. There can be multiple performers working together, but there can also be multiple performances coalescing into a single experience for a consumer. There can be multiple consumers, often called an *audience* (which happens to have the same Latin root as the term *audio*). There can be feedback from consumers that influences the performance and consumers may themselves create spatial sound in form of applause or other audible reactions. The spatial audio performance can of course be part of a multi-sensory performance.

Not only the words *performer* and *consumer*, but also the term *performance* itself are used very loosely here. The performance could simply be the act of communicating between two animals, including humans. The auditory and spatial aspect thereof would be covered by the field of communication acoustics (Blauert 2005; Pulkki and Karjalainen 2015). Instead of (or in addition to) communication, the goal of the performance could also simply be entertainment. Whether a performance is artistic or accidental or anything in between, here we are interested in the spatial audio aspect of it.

Many animals are capable of spatial audio performances, but as far as we know, only humans can write down instructions for others to perform their ideas (see figure 4).

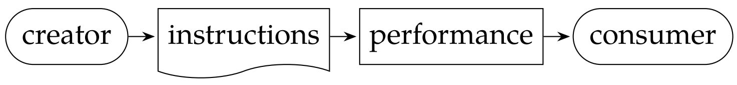


**Figure 4:** *Spatial audio performance based on written instructions*

Of course not every detail of the performance can be controlled by written instructions, and performers will have a lot of leeway in interpreting those instructions. For example, listening to the performance of an ancient Greek tragedy in an amphitheater was certainly very much a spatial audio experience, but the control of spatial audio aspects of the performance via written stage directions was limited.

Not only theater, but also traditional musical performances have an inherent spatial component. An early example for consciously using spatial properties in musical pieces is the polychoral practice from 16th century Venice, where multiple choirs are instructed to perform from different places within a venue, most famously in the *Basilica San Marco di Venezia*. In modern literature, this is often referred to as *cori spezzati*, but this term was probably not used at the time (Bryant 1981; Gembicki 2020).

Some classical and romantic operas and symphonies contain short parts for a separate group of off-stage musicians – often positioned outside the main hall – to achieve an effect of great spatial distance. Some compositions require spatially separated groups of musicians (like *Notturno in D, K. 286* by Wolfgang Amadeus Mozart and *Symphony No. 4* by Charles Ives) or even multiple full orchestras (like *Gruppen* by Karlheinz Stockhausen).

The performances described so far were originally intended to be performed for an audience located in the same room as the performers or – in case of open air performances – in the same outdoor area. With the invention of the *telephone* around 1860, it became possible to transmit sound over an electrical wire and listen to it at the far end. This way, it was possible to listen to an audio performance without being anywhere near the performers (see figure 5).

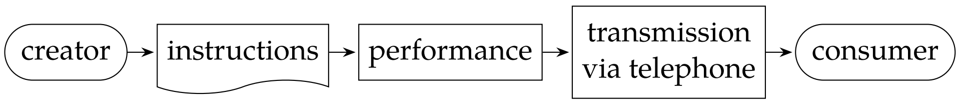


**Figure 5:** *Transmitted audio performance*

The *phonograph* – invented in 1877 – allowed to store the recording of an audio performance (originally in grooves of varying depth on a wax cylinder) and to reproduce it at a later time (see figure 6).

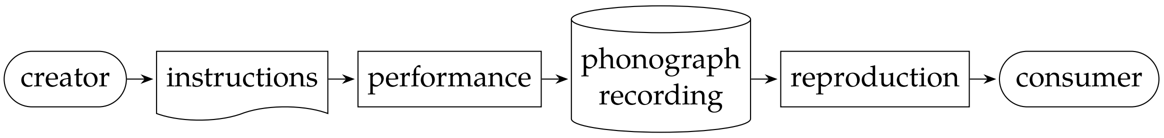


**Figure 6:** *Recorded audio performance*

Even though those inventions were certainly groundbreaking, the poor sound quality (compared to today's standards) and the use of only a single audio channel severely limited the perception of spatial aspects of the performance. In 1881, Clément Ader presented his *théâtrophone* at the *International Exposition of Electricity* in Paris (Hospitalier 1881). Instead of a single transmission channel, it used two telephone mouthpieces mounted on the left and right side of the stage of *Opéra Garnier*, connected with two separate telephone lines to two earpieces (located in a different building in Paris), which allowed listening to the performances happening on the stage, including some crude spatial perception of the performers' positions. The original installation was able to accommodate multiple listeners at once, each one using a separate pair of mouthpieces, telephone lines and earpieces. A patent[2] was granted in 1882. In 1890, the system was established as a commercial service which was in operation until 1932.

The théâtrophone had shown the advantages of using two channels instead of one, but since no amplifiers were available at the time, applications were limited. In the following decades, further development of microphones, amplifiers and loudspeakers opened up more possibilities. In 1924, Franklin M. Doolittle was granted a patent[3] suggesting the transmission of left and right audio signals over two separate AM radio channels. Doolittle also made experimental broadcasts from his radio station WPAJ in New Haven, Connecticut (Doolittle 1925). In his studio, two microphones with a center-to-center distance of about 18 cm were used to pick up sound. The two signals were broadcast over two separate radio frequencies and listeners had to use separate AM receivers for each frequency. The distance between microphones was based on the distance between human ears and broadcasts were mainly intended for headphone reproduction. In a patent[4] from the year 1927, Harvey Fletcher and Leon Sivian suggest the use of an artificial head containing two microphones near the ears to simulate the acoustic scattering of a real human head which strongly affects spatial perception. Fletcher (1933) describes a realization of this idea in the form of an acoustic manikin named *Oscar*.

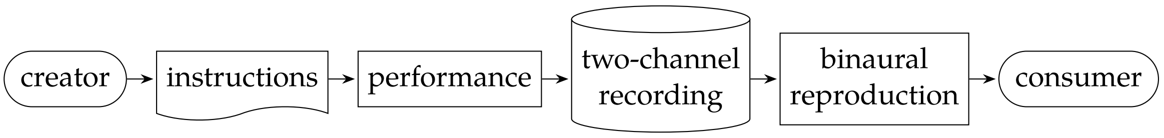


**Figure 7:** *Binaural recording of a spatial audio performance*

[2]`https://worldwide.espacenet.com/patent/search?q=US257453A`
[3]`https://worldwide.espacenet.com/patent/search?q=US1513973A`
[4]`https://worldwide.espacenet.com/patent/search?q=US1624486A`

Just as sound *transmission* was extended to more than one channel, sound *recording* followed suit (see figure 7). Another patent[5] by Doolittle – published in 1931 but filed already in 1921 – describes how two channels can be stored on a phonograph record with two grooves running side by side. The tonearm would have two needles next to each other to reproduce the left and right signals. A very similar approach is described in a 1924 patent[6] by Harry Wier (also filed in 1921). A 1932 patent[7] by W. Bartlett Jones (filed in 1927) mentions multiple variations of two-channel phonographs, including one using a record with a single groove with variations in both depth and lateral shift. This was later known as *V/L* for *vertical/lateral*. Alan Dower Blumlein's UK patent[8] from 1933 (filed in 1931) suggests rotating the single groove recording apparatus by 45 degrees, so that the variations of the two channels still happen at an angle of 90 degrees to each other, but at 45 degrees relative to the disk surface. This type of record is also known as *45/45*. A few years later, the same *45/45* approach was also proposed in a US patent[9] submitted by Arthur Keller and Irad Rafuse. The *45/45* method is still used in today's vinyl records, which are once again quite popular, despite the abundance of modern digital storage media.

Most of the reproduction systems mentioned so far were mainly targeted for headphone listening. The goal was to place two microphones at the same distance as between two ears to create appropriate phase differences. Ideally, some kind of dummy head was used to model the acoustic shadow of a real head. Since the two signals are meant to be reproduced very close to the two ears of a listener, respectively, this method is called *binaural* reproduction, after the Latin words *bis* and *auris*, which mean *two times* and *ear*. An overview of the history of binaural recording technology can be found in (Paul 2009).

The idea of using more than one channel was of course also applied to loudspeaker-based reproduction. This was especially relevant for movie theaters. Even when commercial movies were still silent (the first feature film with sound – *The Jazz Singer* – came out in the year 1927), inventors were already thinking about multi-channel sound for movies. A patent[10] by Edward Amet (filed 1911, granted 1915) describes a device that uses a mono phonograph recording which is switched (not panned!) between multiple telephone receivers. Loudspeaker technology was still in its infancy, and apparently telephone receivers were state of the art for sound reproduction. The telephone receivers would be placed at different positions close to the screen, and the idea was that switching between them allowed the sound to follow the corresponding actions shown on the screen, or as it is phrased in the patent text, "By the means shown a picture of a moving sound-making object may be accompanied in its travel across the screen by its appropriate reproduced sound." The switching of the signal was meant to be achieved by means of electrically conducting lines mounted on small insulating plates – individually handcrafted for each phonograph record used – touching an electric contact point which moves together with the tonearm of the phonograph.

In a patent[11] from 1926, Earl H. Foley suggests using a three-channel recording reproduced over loudspeakers at the left and the right and behind the center of the movie screen. The sound track is to be recorded with three microphones placed across the field of view when shooting the movie. The goal is again for the sound to follow the movements of the performers visible on screen.

[5]https://worldwide.espacenet.com/patent/search?q=US1817177A
[6]https://worldwide.espacenet.com/patent/search?q=US1508432A
[7]https://worldwide.espacenet.com/patent/search?q=US1855149A
[8]https://worldwide.espacenet.com/patent/search?q=GB394325A
[9]https://worldwide.espacenet.com/patent/search?q=US2114471A
[10]https://worldwide.espacenet.com/patent/search?q=US1124580A
[11]https://worldwide.espacenet.com/patent/search?q=US1589139A

Three channels were also used in 1933, when a performance of the Philadelphia Symphony Orchestra was picked up in the *Academy of Music* in Philadelphia with three microphones, transmitted via three telephone lines and reproduced in *Constitution Hall* in Washington, D. C., with three loudspeakers. The system and its capability for spatial audio reproduction has been presented in a *Symposium on Wire Transmission of Symphonic Music and its Reproduction in Auditory Perspective*. The microphone and loudspeaker setup that was used in the two halls is shown in (Bedell and Kerney 1934). Steinberg and Snow (1934) describe additional speech localization experiments that have been conducted with three loudspeakers in the auditorium at the *Bell Telephone Laboratories* connected to three microphones in a smaller pick-up room. Using only two channels instead of three still provided good angular localization from a center listening position, albeit with slightly less accurate depth perception. As the observer moved to one side, however, the virtual source shifted more rapidly toward the nearer loudspeaker than in the three-channel setup. They conclude that "2-channel reproduction of orchestral music gives good satisfaction, and the difference between it and 3-channel reproduction for music probably is less than for speech reproduction or the reproduction of sounds from moving sources."

A three-channel sound track was also used in the first commercial feature-length movie with stereophonic sound – Walt Disney's *Fantasia* – which premiered in the year 1940. The movie consists of a series of classical music pieces, accompanied by animation sequences in different styles. One of the segments features none other than Mickey Mouse as the titular character in Paul Dukas's *Sourcerer's Apprentice*. A narrator gives some explanations in-between the segments, and Mickey Mouse has a short conversation with Leopold Stokowski – the conductor of the sound track – but there is no spoken word during the music pieces and there are no sound effects. This means that any spatial audio positioning and movement was only applied to the music, which is not very common nowadays.

Most of the sound track was recorded in the *Academy of Music* in Philadelphia onto eight tracks of optical film. Six tracks were used for recording different sections of the orchestra, one track held a mono mix of those first six tracks and one track contained a distant pick-up of the whole orchestra. These tracks were afterwards mixed in a so-called *re-recording* process, resulting in three program audio channels destined for the left, center and right loudspeakers, respectively. To be able to dynamically distribute a single recorded track between the three loudspeakers, while at the same time keeping the total power constant, a device nicknamed *"The Panpot"* was used. This name is still used today and the whole process is nowadays called *panning*.

Optical sound tracks at the time did not have enough dynamic range to faithfully reproduce the sound of an orchestra. To overcome this, three variable-gain amplifiers were used, which were controlled by the amplitudes of three sine tones at constant frequencies of 250, 630 and 1600 Hz, respectively. These control tones were generated in the aforementioned re-recording process and were recorded onto a separate channel besides the three program channels. The four channels were then printed on 35-mm film. When *Fantasia* was shown in movie theaters, this film was played back in synchrony with a second film containing the moving images. The channel with the three control tones was fed into the Tone-Operated Gain-Adjusting Device (TOGAD), which in turn controlled the gain stages of the three amplifiers. Those amplifiers were of course using vacuum tubes, which was state of the art at the time. The whole technology was called *Fantasound* (Garity and Hawkins 1941).

Modern texts sometimes claim that the TOGAD has also been used for panning and even for moving sounds around the audience, and some texts mention a relatively large number of loudspeaker channels. With only three control tones and three input channels, the panning options would have been very limited, though. Therefore, we can infer that the control tones were mostly used for increasing the dynamic range. However, in addition to the three main loudspeakers behind the projection screen, there were indeed auxiliary loudspeakers placed in the auditorium. But those were manually switched on by the projectionist in the last segment of the movie – *Ave Maria* – and they were fed by two of the main three sound channels. In some of the installations, the switching was automated by a mechanical relay system operated by means of notches on the edge of the film.

The *Fantasound* system required a lot of equipment that was generally not available at cinemas. Therefore, the movie was presented as a road-show only at selected cinemas, where the custom equipment was installed temporarily. Partly because of the large cost of these installations, *Fantasia* was not a big commercial success. Another reason for discontinuing the project was the involvement of the US in the Second World War.

At the time, the terms *stereophonic* and *binaural* were often used interchangeably. However, Snow (1953) makes a clear distinction between the terms as they are still used today: *binaural* recordings are made with two microphones – preferably in an artificial head – and intended for headphone reproduction, while *stereophonic* recordings are made with two or more microphones and intended for reproduction with two or more loudspeakers "spaced in front of a listening area." It is important to note that even though two-channel recordings were – and still are – very common, stereophony is not limited to two channels at all. The Greek word *stereos* simply means *firm* or *solid*. The term *stereophonic* was most likely inspired by *stereoscopic* images and photographs[12], which were all the rage in the second half of the 19th century.

After the Second World War, magnetic tape was rapidly displacing phonograph records and optical film as a recording medium and in the 1950s, tape recorders were also conquering the concert halls. Electronic sounds and natural sounds were recorded on tape, edited and often electronically modified. Those tapes were then played back during a concert in a concert hall. For example, the composition *Williams Mix* by John Cage, composed between 1951 and 1953, was realized on eight heavily spliced mono tapes, which were played back by eight separate tape machines with eight equidistant loudspeakers around the audience.

Olivier Messiaen's composition *Timbres-durées* was first performed in 1952 from a four-track tape using four loudspeakers. Two tracks were assigned to the left and right loudspeakers in front of the audience. Another track was played back over a loudspeaker behind the audience and one mounted on the ceiling above the audience. Finally, one special track called *cinématique* was interactively distributed over all four loudspeakers using a spatialization device called *pupitre d'espace*, developed Jacques Poullin. This device consisted of four rings of 50 cm diameter placed around the conductor, representing the four loudspeaker positions left, right, above and towards the rear end of the room. The conductor was holding an electrical coil in his hand and moving this coil with big gestures between the rings made the sound move between the four loudspeakers (Battier 2015).

Another famous example is the piece *Kontakte* by Karlheinz Stockhausen (composed 1958–60), in which he used a rotation table on which a loudspeaker was mounted on and which was manually rotated. The resulting sound was recorded on tape via four microphones at fixed positions around the table. During a concert, the four-track tape was played back on four loudspeakers around the audience.

The concept of playing tapes over loudspeakers was taken to the next level at the Philips Pavilion on the 1958 Brussels World's Fair. Edgar Varèse composed a spatial music piece as part of an eight-minutes-long multimedia spectacle called *Poème électronique*, involving a wardrobe-sized 3-track magnetic tape machine, 20 amplifiers and 350 loudspeakers (Kalff

[12] `https://en.wikipedia.org/wiki/Stereoscopy`

et al. 1958). The routing of the audio tracks to groups of loudspeakers, as well as the control of illumination effects, was facilitated by another magnetic tape machine, running in synchrony with the first. The second tape contained 15 tracks with 12 fixed-frequency control signals per track. Some of the continuous signals controlled relays to activate certain loudspeaker groups, others were used to select which of the three audio tracks to use as input for those groups. A part of the loudspeakers formed five so-called *sound routes* along the inside walls of the pavilion, which created the illusion of moving sound sources. A train of pulses in one of the control signals was used to turn a rotary control to switch from one loudspeaker to the next. The rate of these pulses – up to 10 per second – determined the speed of the apparent sound source. In the following years, many more spatial audio compositions for tape and loudspeakers were created, for example *Bohor* (1962) by Iannis Xenakis and *HPSCHD* (1969) by John Cage.

Visitors of the 1970 World's Fair in Osaka, Japan, had the opportunity to experience large-scale spatial music performances in more than one custom-built pavilion on the same exhibition grounds. The German Pavilion was a spherical building housing a spherical auditorium. 50 groups of loudspeakers were mounted at different heights, surrounding a central listening platform from all directions, including above and below. Compositions by Stockhausen for live soloists and multichannel tape were performed with live spatialization. During the performances, the tracks could either be directly routed to fixed loudspeaker positions or via two custom-built *rotation mills* which could be used to move sounds along trajectories of 10 loudspeaker groups each by turning a hand crank (Bates 2015).

Another notable venue at the Osaka Expo in 1970 was the *Space Theatre* in the so-called *Steel Pavilion* of the *Japan Iron and Steel Federation*, where Iannis Xenakis' composition *Hibiki-Hana-Ma* was performed. A 12-track tape was spatialized along elaborate trajectories according to control commands that were recorded on film. The exact number of loudspeakers is unclear, but maybe 264 groups of loudspeakers were installed in the pavilion, with a total of 800 individual loudspeakers (Paland 2015).

While most of the compositions mentioned above were made without the help of computers (Cage's *HPSCHD* being a notable exception), the use of computers became much more common in the 1970s. The computer-generated signals, however, were still often recorded on analog tapes to be used for the performances. In 1972, John Chowning composed the piece *Turenas*, which was to be played on four loudspeakers placed in the corners of a square. Both the sound synthesis – using the recently discovered FM synthesis – and the spatialization – using elaborate trajectories based on Lissajous curves – were realized with the same computer program. It was created on a DEC PDP-10 mainframe using the *Music 10* programming environment, which itself was written in a mix of assembly language and Fortran (Chowning 2011). The method used for spatializing moving sound sources is described in (Chowning 1971).

The principle of playing back pre-recorded audio tracks and distributing them among loudspeakers during a concert is shown in figure 8. In some instances, composers produce a score which is used by other people to create appropriate tapes for performances. Often, however, the composers would record and edit the tapes themselves. Note that typically, the composer gives instructions for the recording and editing of the tracks beforehand as well as for their spatialization during the concert (which may be automated or done manually). This differs from classical orchestra recordings (see figure 9), where the composer only gives instructions for the performance itself.

After spatial audio being successfully used in concert halls and cinemas, it was just a matter of time that it would also find its way into private homes. Commercial two-channel stereo records and the corresponding reproducers were widely available to consumers since the late 1950s. Later, reproduction systems with four loudspeakers – to be placed

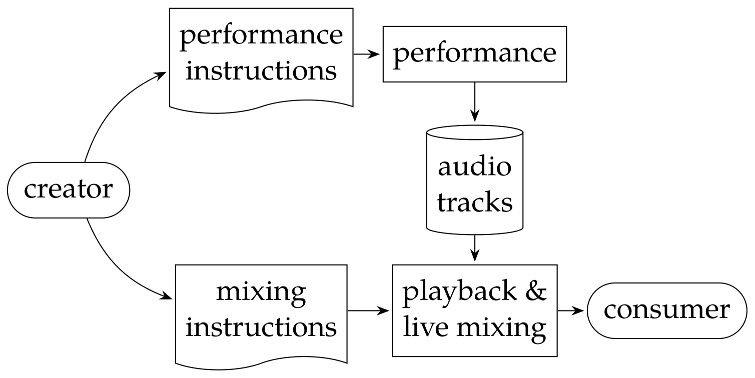


**Figure 8:** *Tape-based multi-channel audio performance*

in the corners of a square, surrounding the listener from all directions in the horizontal plane – were sold under the name *Quadraphony* (Woodward 1977). In 1967, *Pink Floyd* gave a live concert using a quadraphonic loudspeaker setup controlled by the *Azimuth Co-ordinator*[13] panning device. Their album *The Dark Side of the Moon* was released a few years later in quadraphonic sound. Even though a number of quadraphonic records were produced and sold in the 1970s, the technology never really took off on the mass market.

Cooper (1970) suggested to extend the horizontal quadraphonic system with a height dimension by using a tetrahedral microphone and loudspeaker setup. The suggested setup was quickly superseded by a more systematic approach by Gerzon (1970). This approach – which was based on *spherical harmonics* – was generalized to larger numbers of loudspeakers (Gerzon 1973) and later became known as *Ambisonics* (Fellgett 1975; Gerzon 1975).

The quadraphonic loudspeaker setup – with a loudspeaker in each corner of a square – was not much used beyond the 1970s. This was probably because the movie industry did not want to give up the center loudspeaker behind the projection screen, which was especially useful for a stable perception of the dialogue in the whole auditorium. Cinemas were using a different setup for four-channel sound tracks: they used the traditional left/center/right positions behind the screen, while the fourth channel was providing ambient sounds from the back wall. Later, a fifth channel was added in order to be able to position ambient sounds between the left and the right side behind the audience. An additional, bandwith-restricted channel was used to provide low frequency effects. Variations of this setup were used throughout the 1980s. In the early 1990s the loudspeaker layout was standardized by ITU-R as Recommendation BS.775[14] – also known as *5.1 surround* – which is still widely used today. In the following years, more setups with more and more loudspeakers – including loudspeakers at different heights – have been proposed, up to *22.2 surround* (Hamasaki, Hatano, et al. 2004; Hamasaki, Hiyama, et al. 2005).

The signals for the different loudspeaker channels are typically obtained by using a combination of microphone techniques and panning. As an example, figure 9 shows a typical modern procedure for recording an orchestral performance.

Pairwise panning has been used for many decades with a multitude of panning devices and panning curves. Modern panning techniques which allow for arbitrary three-dimensional loudspeaker layouts include Vector Base Amplitude Panning (VBAP) (Pulkki 1997), Distance-Based Amplitude Panning (DBAP) (Lossius et al. 2009) and All-Round Ambisonic Panning (AllRAP) (Zotter and Frank 2012).

[13]https://collections.vam.ac.uk/item/O76817/
[14]https://www.itu.int/rec/R-REC-BS.775/

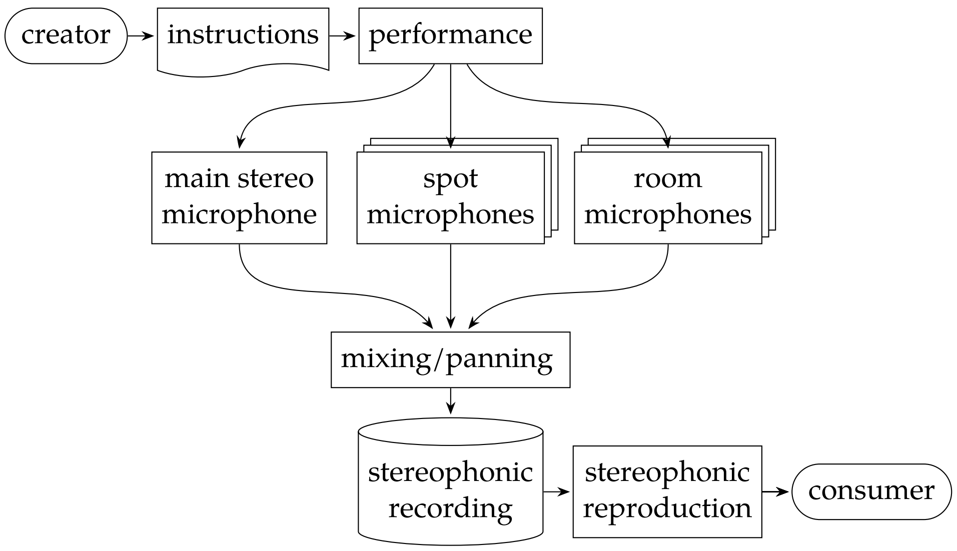


**Figure 9:** *Modern multi-microphone orchestra recording*

Our ability to localize sounds in stereophonic reproduction is based on a psychoacoustic effect called *summing localization* (Theile 1980). This is known to work well if the listener position is at equal distance to each loudspeaker – in a point that's called *sweet spot*. However, it does not work very well at all between the left back and left front loudspeaker nor between the right front and right back loudspeaker (Theile and Plenge 1977).

To overcome these limitations, an alternative approach – aiming at physically accurate reconstruction of a sound field and therefore termed *sound field synthesis* – has been brought forward. It is based on Huygens' principle[15], which has already been discovered in the 17th century. The idea was maybe first mentioned by Fletcher (1934) who describes a hall with an acoustically transparent curtain with microphones "scattered uniformly over it" mounted between the orchestra and the audience. The signals picked up by those (hypothetical) microphones could then be transmitted to a different hall with a similar curtain, but with loudspeakers instead of microphones affixed to it. In this thought experiment, the audience in the second hall "should obtain the same effect as those listening to the original music" if the two halls have the same size, shape and acoustical properties and – last but not least – the number of microphones and loudspeakers is infinite and they are infinitesimally small. This fictional setup is brought up again by Snow (1953), who, instead of using the word *curtain*, mentions a *screen* consisting of an extremely large number of extremely small microphones and a corresponding *screen* of extremely small loudspeakers. Talking about reality again, he makes sure to clarify that, when using a practical setup of two or three loudspeakers, a different hearing mechanism is used by the brain. Decades later, the principle was mathematically formalized using Rayleigh's first integral equation (Berkhout 1988; Berkhout et al. 1993). This finally led – together with advances in amplifier, loudspeaker and digital signal processing technology – to an actually realizable system named Wave Field Synthesis (WFS). Still in theory, the *screen* of loudspeakers would completely enclose a volume around the listener, but in order to be built in practice, a single horizontal line of small loudspeakers would be mounted at ear height around the listening area. Originally, only linear loudspeaker arrays were supported, but the method has been extended to allow for arbitrary convex loudspeaker layouts – but with all loudspeakers still located in a horizontal plane (Spors, Rabenstein, et al. 2008).

[15]`https://en.wikipedia.org/wiki/Huygens-Fresnel_principle`

Because of the large number of loudspeakers, it was not practical to record each loudspeaker signal on its own channel for later reproduction. This would be called *channel-based* reproduction, which is the typical way to store recordings for stereophonic systems like 5.1, for example. Not only do stereophonic systems have a smaller number of loudspeaker channels, but they also have standardized loudspeaker layouts, which makes it possible to reproduce the same *channel-based* recording on any compatible system. In contrast, WFS systems typically not only have many more loudspeakers than stereophonic systems, but they also have very different – and nearly arbitrary – loudspeaker layouts, which would mean that a recording for one system would not be able to be played back on a different WFS system. This led to a new paradigm for the storage of recordings called *object-based* reproduction (Geier, Ahrens, and Spors 2010; Geier, Spors, and Weinzierl 2010; Pereira and Ebrahimi 2002; Tsingos 2018).

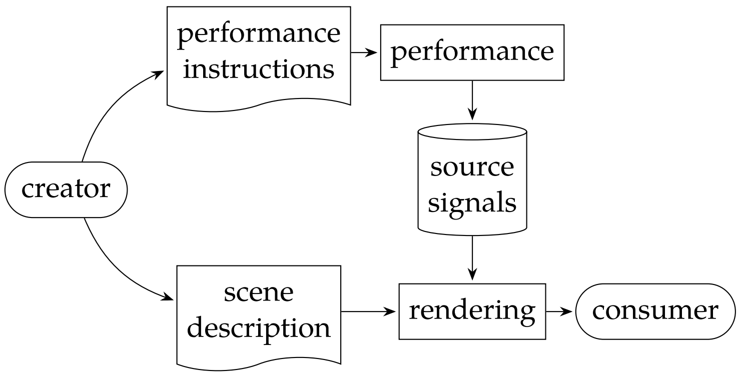


**Figure 10:** *Object-based audio reproduction*

The *object-based* production workflow is depicted schematically in figure 10. Instead of storing the signals of all loudspeakers, the source signals – or sub-mixes – are stored together with some data indicating when and from which spatial positions these signals should be heard by the audience. These source signals together with their associated spatio-temporal and other information are called *scene description*. From this scene description, the loudspeaker signals are then generated in real time during reproduction in a procedure called *rendering*. Since the scene description ideally doesn't contain any information about the reproduction system, it can be rendered on any system, regardless of number and layout of loudspeakers. Existing *channel-based* recordings can still be used in *object-based* systems by placing the individual channels at appropriate positions in the scene description. Theile, Wittek, et al. (2003) call this *virtual panning spots*. Note that figure 10 has the exact same structure as figure 8. In a way, *object-based* reproduction can be viewed as a modern reincarnation of the tape-based spatial music techniques from the 1950s and 1960s.

In the 1990s, another method for *sound field synthesis* was developed by extending the above-mentioned *Ambisonics* approach. Originally, spherical harmonics of zeroth and first order had been used to create driving signals for four loudspeakers mounted in a three-dimensional tetrahedral arrangement. This led to a rather low spatial resolution, but by using higher orders – and a correspondingly larger number of loudspeakers – the accuracy of the reproduced sound field could be improved. To distinguish it from the original approach, this was given the name Higher-Order Ambisonics (HOA). The original theory of Ambisonics assumed that the distances to the loudspeakers are very large, leading to incoming plane wave fronts within the designated listening area. However, the distances in real loudspeaker setups are much smaller, which causes curved wave fronts and therefore errors in the reproduced sound field. To overcome this, Near-Field-Corrected HOA (NFC-HOA) has been developed (Daniel 2001, 2003).

For a three-dimensional HOA system of order $N$, the number of spherical harmonics components – i. e. the number of storage channels – is $(N+1)^2$. For example, a third order Ambisonics system needs 16 storage channels. For two-dimensional horizontal systems, only $2N+1$ circular harmonics components – and therefore storage channels – are needed. These component signals – also called *B-format* – are independent of the targeted loudspeaker positions. Given a certain loudspeaker setup, the Ambisonics components have to be *decoded* at the reproduction site in order to generate the appropriate loudspeaker driving signals.

Since the loudspeaker signals are created on the fly and not used for storage or transmission, HOA is not considered a *channel-based* method. However, there seems to be no consensus on how to call it instead. Spors, Wierstorf, et al. (2013) use the term *transform domain-based*, Nicol (2018) and Robinson and Tsingos (2015) use the term *sound field-based* and others simply call it *HOA-based* for the lack of a better term. There are even authors who for no apparent reason call it *scene-based*, which is quite misleading since the word *scene* is already being heavily used in the context of *object-based* reproduction and the term *scene-based* has already been used earlier for something completely different by Rumsey (2002).

Storing and/or transmitting Ambisonics component channels is not the only way to use HOA, though. It can also be used on the reproduction side of an *object-based* system by means of real-time Ambisonics amplitude panning (Neukom and Schacher 2008; Zotter and Frank 2019). This way, high Ambisonics orders can be used without having the need for storing the Ambisonics component signals. However, this is not advantageous if the number of simultaneous source signals is larger than the number of Ambisonics components for the desired order.

For more information about loudspeaker-based reproduction see (Blauert and Rabenstein 2012; Spors, Wierstorf, et al. 2013) and for the mathematical fundamentals of sound field synthesis see (Ahrens 2012).

As mentioned earlier, the history of spatial audio transmission and recording started towards the end of the 19th century with *binaural* transmission and reproduction. The usage of an artificial head with microphones mounted in it has already been described in the late 1920s. Stereophonic loudspeaker reproduction came later, but it slowly took over nearly all of the market. Binaural reproduction had a very short-lived renaissance in the 1970s, where several dummy head recordings were produced and gained some popularity, but were later quickly forgotten again by the general public.

Another milestone in binaural technology was the measurement of Head-Related Transfer Functions (HRTFs), which compactly represent the acoustic effect of a listener's outer ears, head and torso on an incoming sound, depending on the direction of incidence. This enabled *binaural rendering* of object-based audio scenes. By using a *head tracker*, the rotation of the listener's head can be compensated for in real time (Wenzel et al. 1990). This process is known as *dynamic binaural rendering* or *dynamic binaural synthesis*.

Figure 11 shows a potential problem with *channel-based* techniques: since multiple different standardized loudspeaker setups are in general use, each production has to be manually mixed multiple times – once for each target setup. Automatic up- and down-mixing methods exist, but they often lead to reduced sound quality (Avendano and Jot 2004; Faller 2006; Vilkamo et al. 2014; Zielinski et al. 2003). As an alternative, the *object-based* approach can be used for *system-independent mixing*. A scene description is created only once, and arbitrary *channel-based* mixes can be rendered automatically. This approach also allows *binaural monitoring* of loudspeaker systems (Geier, Ahrens, and Spors 2009).

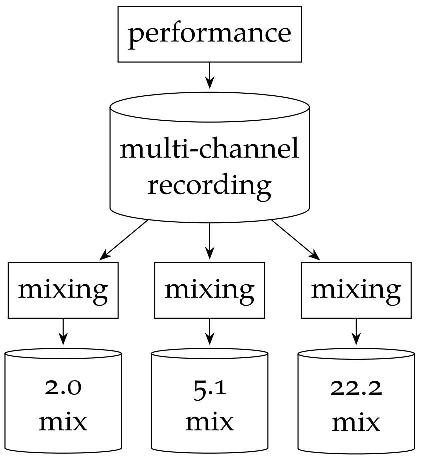


**Figure 11:** *Multiple mixes for different output formats*

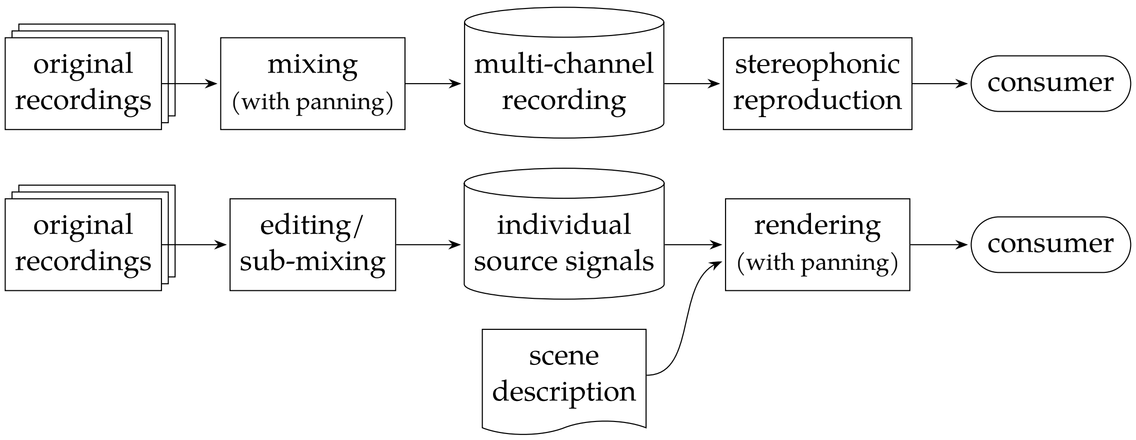


**Figure 12:** *Panning during production of* channel-based *content (above) compared to panning during consumption of* object-based *content (below)*

Switching from *channel-based* to *object-based* can be seen as moving part of the mixing process from the producer to the consumer (see figure 12). For example, panning can be used to create *channel-based* mixes – which has been done since the late 1930s and is still done today at a massive scale – but it can also be used for the consumer-side rendering of an *object-based* recording. These days – as it has been the case since more than a century – the movie industry is the biggest driver for commercial spatial audio technology. A hybrid of *channel-based* and *object-based* movie sound tracks (Robinson, Mehta, et al. 2012) have been widely used in cinemas in the last 10 years.

# Chapter 2

# Movement and Time in Existing Formats

The end of the previous chapter has chronicled the emergence of *object-based* audio reproduction systems and their need for scene descriptions. A *scene description*, in this context, consists of a detailed description of *sound objects*, including their source signals, their positions over time and other information that is needed to reproduce the desired auditory scene for one or more listeners with any compatible reproduction system. Over time, several scene description formats have been proposed. This chapter will present a few of those formats, with special focus on how they handle the description of movement (of sound sources and other scene objects) over time. The list of formats shown here is by no means exhaustive, and only formats are presented where public information is available, which excludes some proprietary commercial formats.

## 2.1 Recurring Concepts

The following subsections will describe a few common concepts that are relevant for multiple formats and that hopefully make it easier to discuss and categorize the individual formats which will be presented after this section.

### 2.1.1 Declarative vs. Procedural vs. Sampled Data

These terms are probably best explained by example. Consider the sentence "the sound of a bumblebee approaches from the far left, circles the head of the listener two times and then stops abruptly." This sentence can be seen as a *declarative* audio scene description. It is a very high-level description and also very vague and inexact. To make it actually usable in a real system, more information probably has to be provided. The trajectory of the bumblebee could be described more exactly by providing a few coordinates and the times when those should be reached by the bumblebee and maybe its velocity at those points. The intermediate positions would be interpolated by the renderer according to a pre-determined mathematical procedure. This would still be considered *declarative*. Describing the sound itself in a *declarative* way might be harder. The bumblebee sound could be created by a generic synthesizer module, and the scene description would contain standardized parameters for this synthesizer – most likely changing over time – like selection of sound generators and filter coefficients.

More realistically, the sound would be provided by a natural recording of a bumblebee, stored as a digital signal, which is just a stream of *sampled data*. The sound could then be spatialized according to a *declarative* trajectory as described above. However, the trajectory could just as well be stored as a stream of coordinates, regularly *sampled* at certain intervals. This is used in object-based cinema sound systems, where the position data is typically sampled at 30 Hz or more (Riedmiller et al. 2015). *Sampled data* typically refers to temporal sampling, but it could also be angular or other spatial sampling, for example, when storing measured source directivity patterns as part of a scene.

Some scene description formats allow the usage of a scripting language to generate arbitrary movements on the fly. This is called *procedural*, because the actual procedure is stored as part of the scene description. An important difference to the *declarative* approach is that the rendering engine has no knowledge about trajectories or sound synthesis parameters or any such high-level constructs. It would instead just render the source signal at whatever position the script generates at any given time. The signal itself could also be generated by a piece of software that is distributed as part of the scene description, for example as a software plugin.

Oftentimes the three concepts are mixed, for instance in the X3D format (see section 2.3): *declarative* trajectories can be defined by a small number of control points, either with linear or smooth interpolation (`PositionInterpolator`, `SplinePositionInterpolator`), *sampled data* is used for the source signals (stored separately as conventional mono audio files), and *procedural* animations can be realized by means of `Script` nodes.

### 2.1.2 Scene Graph

A *scene graph* is commonly used in 3D scene descriptions as the top-level structural element. It is a hierarchical, tree-like representation containing all scene components as so-called *nodes*. Those scene components can be part of other nodes which define their positions and orientations with respect to a local coordinate system. Those nodes can in turn be part of other nodes with their own local coordinate systems and so on. All nested containers contribute to the final position and orientation of the scene components. A change in the position, orientation or scaling of a node affects all its child nodes. For example, a car could be modelled as a node in the scene graph with a certain position and orientation in the virtual world. The car can then consist of multiple sub-nodes like tyres and seats and doors, which are all defined in the local coordinate system of the car. When the car node – and therefore its local coordinate system – is moved in the scene, all its child nodes move together with it. This is commonly used for visual scene descriptions, but a scene graph could also be used for audio scenes. In such a case, a car node could define a local coordinate system containing multiple audio objects for the noise of the tyres and another audio object with engine noise. This makes for a very clear and well-structured spatial hierarchy, which can be very useful to define a static 3D model consisting of many elements which are again comprised of many sub-elements.

On the flip side, the temporal structure of a scene if often obscured. For an example see the `ROUTE` node of VRML (section 2.2) which breaks the tree-like hierarchy and connects events, timers and interpolators with the scene properties that are supposed to be changed over time. Routes, interpolators and other timing nodes are stored within the scene graph, but they are not logically part of the hierarchy. Moving a parent element has no effect on those timing nodes, which means that they could basically be placed anywhere in the scene graph without a change in behavior.

An alternative to the *scene graph* concept is provided by the SMIL format (see section 2.11). Instead of using spatial relationships for the main structural organization, it focuses on temporal relationships with a primary structure called *time graph*.

### 2.1.3 Metadata

Many authors use the term *metadata* to describe spatial data – trajectories, for example – within a scene description, maybe to distinguish it from *audio data*. The word *metadata* means "data about data" and its use is inappropriate in this case. Actual *metadata* of an audio signal could be the date of recording, the used microphone type or other equipment, a description of the audible sound sources and even their positions at the time of recording. The intended positions and movements of sources in a scene description, however, are simply additional *data* and not *metadata*, just like the sound track is not *metadata* of a movie.

An audio file can be swapped with another one while the movement of the virtual sound source remains the same, which confirms that the movement data is not *metadata*. A scene description can (and often does) contain actual metadata, though. This could be – among many other things – the name of the author(s), date of creation and a license governing the usage and re-distribution of the content.

## 2.2 Virtual Reality Modeling Language (VRML)

The VRML is a storage format mainly for 3D computer graphics, developed in the early days of the *world wide web*. In addition to graphics, it can also describe spatial audio sources, including moving ones. VRML version 2.0 – also known as VRML97[1] – became an ISO standard[2] in 1997.

The main structure of a VRML scene is a scene graph, which is built from (possibly nested) `Transform` nodes which define nested local coordinate systems. All visible geometric elements (defined by polygons), as well as light sources, camera views and also audio objects are added to this scene graph. If the same data is needed repeatedly (e. g. vertex coordinates), it can be defined once with `DEF` and used multiple times with `USE`.

The positions and orientations of all local coordinate systems of the scene graph are specified statically. Positions are given as triples of Cartesian coordinates and orientations are given as three numbers representing a normalized rotation vector plus a fourth number representing the rotation angle in radians. The movement of scene elements can be achieved in multiple ways. `TimeSensor` nodes can be defined to change certain properties of certain scene elements at given times or at regular intervals. Interactive changes can be triggered based on the virtual position of the viewer by means of `ProximitySensor` nodes. Continuous position changes can be realized with `PositionInterpolator` nodes, which interpolate linearly between a given sequence of positions and their associated time values. For continuous orientation changes, the `OrientationInterpolator` node uses **s**pherical **l**inear int**erp**olation (Slerp)[3] between a sequence of orientations and their associated time values. The interpolation between two orientations always happens along the smallest angle, which means that a single rotation step cannot be larger than 180 degrees. If the angle is exactly 180 degrees, the result is explicitly undefined. Proximity sensors can trigger time sensors, which themselves can control the progression of interpolators. All these nodes have to be connected by `ROUTE` commands. The following (abridged) example defines a simple scene showcasing position and orientation interpolation as well as the `Sound` and `AudioClip` nodes, which can be used to define sound sources and their corresponding source signals:

```
#VRML V2.0 utf8
Viewpoint { position 23.01 16.46 8.282 }
DEF TimeSensor01 TimeSensor {
  cycleInterval 5 loop FALSE stopTime 1 }
DEF Transform01 Transform {
  translation 0.0759 0 -0.4247
```

(continues on next page)

[1]https://www.web3d.org/documents/specifications/14772/V2.0/
[2]https://www.iso.org/standard/25508.html
[3]For an explanation of *Slerp* see section B.3.2 in the appendix.

(continued from previous page)

```
  rotation -1 0.004363 -0.004363 -1.571
  children [
    DEF PosInterp01 PositionInterpolator {
      key [ 0, 0.02, ..., 1 ]
      keyValue [ 0.0759 0 -0.4247, 0.4856 0 -2.427,
        ..., 0.0759 0 -0.4247 ] },
    DEF OriInterp01 OrientationInterpolator {
      key [ 0, 0.02, ..., 1 ]
      keyValue [ -1 0.004363 -0.004363 -1.571,
        -0.9994 0.02502 -0.02502 -1.571,
        ..., -1 0.004363 -0.004363 -1.571 ] },
    Shape {
      appearance Appearance { material Material { ... } }
      geometry IndexedFaceSet {
        coord Coordinate { point [ 48.67 8.484 -14.88,
          48.67 8.484 -16.96, 48.67 7.446 -14.88, ...  ] }
        coordIndex [ 0, 1, 2, -1, 0, 2, 3, -1, ... ] } }
    Shape { ... }
    Transform {
      translation 1 2.5 -1
      children [
        Sound {
          source DEF Sound01 AudioClip {
            url "my-sound.wav" loop TRUE } } ] }
  ]
  ROUTE PosInterp01.value_changed TO Transform01.set_translation
  ROUTE TimeSensor01.fraction_changed TO PosInterp01.set_fraction
  ROUTE OriInterp01.value_changed TO Transform01.set_rotation
  ROUTE TimeSensor01.fraction_changed TO OriInterp01.set_fraction
}
DEF ProxSensor01 ProximitySensor {
  enabled TRUE center 47.01 15.92 7.43 size 2.5 2 1.8
}
ROUTE ProxSensor01.enterTime TO TimeSensor01.startTime
ROUTE ProxSensor01.enterTime TO Sound01.startTime
ROUTE ProxSensor01.exitTime TO Sound01.stopTime
```

In this example, the transform node called `Transform01` is moved (i. e. translated) by the interpolator named `PosInterp01`. This interpolator is controlled by the time sensor called `TimeSensor01` which is in turn triggered by the proximity sensor named `ProxSensor01`.

As an alternative to interpolators, `Script` nodes can be used to provide custom code – typically implemented in ECMAScript/JavaScript – which allows generating arbitrary parameter progressions, including the animation of position and orientation. The inputs and outputs of `Script` nodes are connected to other nodes by `ROUTE` commands. The VRML standard also describes the so-called External Authoring Interface (EAI), which allows manipulating the scene graph during runtime from external applications.

## 2.3 Extensible 3D (X3D)

The successor of the VRML is X3D, which is an ISO standard[4] since 2004. It is maintained by the Web3D Consortium[5]. The previous version[6] of the standard was released in 2013 and the latest version[7] is just being released in 2023.

X3D builds on the same concept of a scene graph, and most nodes from VRML are still available, including `Transform`, `TimeSensor`, `ProximitySensor`, `Script`, `PositionInterpolator` and `OrientationInterpolator`. There is also a `ROUTE` element to connect events and `DEF` and `USE` attributes to define and re-use elements. There are also several new nodes, providing additional features on top of VRML. The `SplinePositionInterpolator` node can be used to create trajectories along Hermite splines, allowing to specify incoming and outgoing velocity vectors at each control point (see section B.2.4 in the appendix). If no velocity vectors are specified, the tangents are automatically calculated to produce *Catmull–Rom splines*. The X3D standard doesn't get the equations quite right, for a correct derivation of the tangents for non-uniform Catmull–Rom splines see section B.2.8 in the appendix. The `SquadOrientationInterpolator` can be used to animate orientations with spherical **quad**rangle interpolation (Squad)[8].

X3D has three syntaxes: a new XML-based syntax, the old VRML syntax and a binary format for efficient storage and transmission. Similar to the EAI in VRML, X3D defines a Scene Access Interface (SAI) for real-time interaction with external programs. X3D has 4 baseline profiles: *Interchange*, *Interactive*, *Immersive* and *Full*. The `Sound` and `AudioClip` elements are part of the *Immersive* profile.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN">
<X3D profile="Immersive" version="3.0">
  <Scene>
    <Viewpoint position="0 0 4"/>
    <TimeSensor DEF="Time01" cycleInterval="5" loop="true"/>
    <PositionInterpolator DEF="PosInterp01"
      key="0 0.08 0.16 0.24 0.32 0.4 0.48 0.56 0.64 0.72 0.8 0.88 0.96"
      keyValue="0 0 0  -2 0 0  0 0 0  2 0 0
                0 0 0  0 -2 0  0 0 0  0 2 0
                0 0 0  0 0 -2  0 0 0  0 0 2  0 0 0"/>
    <Transform DEF="Transform01" rotation="0 1 0 1.57">
      <Sound>
        <AudioClip loop="true" url="chimes.wav" />
      </Sound>
    </Transform>
    <ROUTE fromNode="Time01" fromField="fraction_changed"
           toNode="PosInterp01" toField="set_fraction"/>
    <ROUTE fromNode="PosInterp01" fromField="value_changed"
           toNode="Transform01" toField="set_translation"/>
  </Scene>
</X3D>
```

Many more examples are available on the web. It is even possible to embed X3D scenes in HTML5 pages with *x3dom*[9].

[4]https://www.iso.org/standard/60760.html
[5]https://www.web3d.org/
[6]https://www.web3d.org/standards/version/V3.3
[7]https://www.web3d.org/x3d4
[8]For an explanation of *Squad* see section B.3.9 in the appendix.
[9]https://www.x3dom.org/

## 2.4 MPEG-4 AudioBIFS

MPEG-4 Audio is an ISO/IEC standard[10] since 1999. A separate part of the standard[11], known as BInary Format for Scenes (BIFS), which deals with the description of scenes, contains a superset of the functionality provided by VRML. It therefore inherits the *scene graph* approach from VRML including the `Sound`, `AudioClip`, `Viewpoint`, `PositionInterpolator` and `OrientationInterpolator` nodes. The set of additional nodes for spatial audio (for example `AudioSource`, `AudioBuffer`, `AudioMix`, `AudioFX`) is known under the name AudioBIFS (Scheirer et al. 1999). However, no new nodes have been added for describing the movement of sound sources. The additional nodes from X3D (`SplinePositionInterpolator` and `SquadOrientationInterpolator`) are not included. Movement of sound sources can be achieved by the same means as in VRML, using event routing with `ROUTE`. In addition, "BIFS Animation" commands can be included in the BIFS data stream, and nodes in the scene graph can be added and replaced with "BIFS Update" commands.

Scenes are stored and transmitted using a binary format, but there is also an alternative textual representation called eXtensible MPEG-4 Textual (XMT) (Kim, Wood, et al. 2000). It comes in two flavors: XMT-A has an XML-based syntax similar to X3D (see section 2.3), and XMT-Ω has a syntax inspired by SMIL (see section 2.11).

A second version of MPEG-4 was published as an amendment to the standard in the year 2000. The added audio nodes `AcousticScene`, `AcousticMaterial`, `DirectiveSound` and `PerceptualParameters` are also known as Advanced AudioBIFS (Väänänen and Huopaniemi 2004). These new nodes are used for physical and perceptual modeling of acoustic environments, but no features regarding sound source movement have been added.

A third version of AudioBIFS added the nodes `AudioChannelConfig`, `Transform3DAudio`, `WideSound`, `SurroundingSound` and `AdvancedAudioBuffer` and it changed the handling of `AudioFX` nodes (Schmidt and Schröder 2004).

## 2.5 Audio3D

Audio3D is an XML-based audio scene description format presented by Hoffmann et al. (2003). It uses a *scene graph* to define the spatial arrangement. The position of sound sources and of the listener can be animated with the `<Animation>` element:

```
<Source position="-5 0 1" startTime="0" minDistance="0.4" maxDistance="100
 ↪">
  <Animation startTime="0" loop="true" attribute="position">
    <VectorKey value="0 5 1" time="5000"/>
    <VectorKey value="5 0 1" time="10000"/>
    <VectorKey value="0 -5 1" time="15000"/>
    <VectorKey value="-5 0 1" time="20000"/>
  </Animation>
  <Sound input="FILE" loading="STATIC" location="loop1.wav" loop="true"/>
</Source>
```

The "direction" – presumably a direction vector – can be animated as well:

```
<Source position="0 0 0" minDistance="1" maxDistance="100"
        direction="-1 0 0" insideConeAngle="40" outsideConeAngle="120">
```

(continues on next page)

[10]https://www.iso.org/standard/76383.html
[11]https://www.iso.org/standard/63548.html

(continued from previous page)

```
  <Animation attribute="direction" loop="true">
    <VectorKey value="0 -1 0" time="5000"/>
    <VectorKey value="1 0 0" time="10000"/>
    <VectorKey value="0 1 0" time="15000"/>
    <VectorKey value="-1 0 0" time="20000"/>
  </Animation>
  <Sound input="FILE" loading="STATIC" location="testloop.wav" loop="true
↪"/>
</Source>
```

Not only spatial properties, but also gain values can be animated with the same approach:

```
<Source position="0 5 1.5" minDistance="1" maxDistance="100">
  <Sound input="FILE" loading="STATIC" location="loop1.wav" gain="0" loop=
↪"true">
    <Animation startTime="0" attribute="gain">
      <FloatKey value="100" time="4000"/>
      <FloatKey value="100" time="6000"/>
      <FloatKey value="0" time="10000"/>
    </Animation>
  </Sound>
</Source>
```

The values between key frames are interpolated linearly.

## 2.6 XML3DAUDIO

Like the name suggests, XML3DAUDIO is an XML-based description format for three-dimensional audio scenes. It has been proposed and defined in (Potard 2006; Potard and Burnett 2002, 2004; Potard and Ingham 2003).

Positions of sound sources are specified using left-handed Cartesian coordinates (in meters). Source orientations can be given with `<Rotate>`, `<Tilt>` and `<Tumble>` (in degrees). A listener position and orientation can also be specified. The listener's orientation can be given with `<Azimuth>`, `<Elevation>` and `<Roll>`. Multiple listeners can be defined and the point of view can be switched between them.

Dynamic movements of scene objects can be implemented with an "orchestra and score" approach inspired by the venerable computer music software Csound[12]. The *orchestra* part contains a list of objects like listeners and sound sources together with some static attributes. The *score* part contains *lines of score*, each one incorporating one of a given set of *opcodes*. There are again two parts: an *initialization score* and a *performance score*. The former can for example be used for creating groups of sources. The latter allows for changing scene parameters dynamically over time, for example translating and rotating sound sources and listener objects.

Potard (2006, p. 127) provides a single example scene in XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<AUDIO_SCENE>
  <ORCHESTRA>
    <Listener>
      <Id>Guillaume</Id>
      <Position>
```

(continues on next page)

[12] https://csound.com/

(continued from previous page)

```
        <X1>3</X1>
        <Y1>5</Y1>
        <Z1>0</Z1>
      </Position>
      <Orientation>
        <Azimuth>0</Azimuth>
        <Elevation>0</Elevation>
        <Roll>0</Roll>
      </Orientation>
    </Listener>
    <Source>
      <Id>beachfront</Id>
      <URL>beachfront.wav</URL>
      <Dimensions>
        <X>0</X>
        <Y>100</Y>
        <Z>0</Z>
      </Dimensions>
    </Source>
    <Source>
      <Id>seagull</Id>
      <URL>http:\\dummyserver.com\seagull-stream.mp3</URL>
      <Position>
        <X>10</X>
        <Y>5</Y>
        <Z>10</Z>
      </Position>
    </Source>
    <Recorded_Scene>
      <B-format>
        <Id>beach-crowd</Id>
        <URL>beach-crowd.wxyz</URL>
      </B-format>
    </Recorded_Scene>
  </ORCHESTRA>
  <SCORE>
    <Performance_Score>
      <Line_of_Score>
        <start_time>0</start_time>
        <Duration>100</Duration>
        <Command>play</Command>
        <Object>beachfront</Object>
        <Object>beach-crowd</Object>
        <Parameter>loop</Parameter>
      </Line_of_Score>
      <Line_of_Score>
        <start_time>20</start_time>
        <Duration>30</Duration>
        <Command>play</Command>
        <Object>seagul</Object>
      </Line_of_Score>
      <Line_of_Score>
        <start_time>20</start_time>
        <Duration>30</Duration>
        <Command>move</Command>
```

(continues on next page)

(continued from previous page)

```
        <Object>seagul</Object>
        <Parameter>20</Parameter>
        <Parameter>10</Parameter>
        <Parameter>8</Parameter>
      </Line_of_Score>
    </Performance_Score>
  </SCORE>
</AUDIO_SCENE>
```

The same example scene – with minor modifications – is also printed in (Potard and Burnett 2004, fig. 5, p. 4). Other than that, no further scenes seem to be publicly available.

The example shows how the “move” opcode can be used to animate the translation of a sound source by providing target coordinates. Presumably, linear interpolation is used. According to the aforementioned literature, it is also possible to dynamically control rotations by a 3D rotation vector. However, no example is provided. Other scene parameters can be dynamically controlled with the `ChangeParameter` opcode.

Potard and Ingham (2003) mention that “complex trajectories can be described by the `TRAJ` opcode and a long list of coordinates.” Sadly, no more information and no examples are provided. In the other publications, this opcode is not even mentioned at all.

## 2.7 Audio Definition Model (ADM)

The ADM is an audio scene description format defined by ITU-R in Recommendation BS.2076[13]. Its latest edition BS.2076-2 was published in 2019.

The ADM is an XML-based format that’s typically embedded in the `<axml>` chunk of Broadcast Wave Format (BWF) files, as specified by ITU-R in Recommendation BS.2088[14]. The `<chna>` chunk is used to associate the track ID used in the XML description with the appropriate audio channels from the BWF file.

Positions of sound sources are by default stored using a spherical coordinate system with `azimuth` and `elevation` angles in degrees and `distance` in relative units. Alternatively, Cartesian coordinates can be used with `X` (to the right), `Y` (forward) and `Z` (up) in relative units. Relative units are mapped to physical units with the `absoluteDistance` value (in meters). All coordinates are understood as relative to a fixed listener position. If the listener’s head rotation is tracked during playback, the `headLocked` flag can be used to interpret the given coordinates as relative to the head rotation. Other than that, there is no way to specify any rotations.

Positions (and other dynamic scene parameters) are assigned to sound sources using `<audioBlockFormat>` elements. Each of those elements can only contain one position value. To define moving sound sources, multiple such elements have to be used. Each block has to have a start time (`rtime`) relative to the start time of the parent element and a `duration` value in seconds.

Example (simplified excerpt) from annex 2, section 2.3 of the ADM specification:

```
<audioChannelFormat typeDefinition="Objects">
  <audioBlockFormat rtime="00:00:00.00000" duration="00:00:05.00000">
    <position coordinate="azimuth">-22.5</position>
    <position coordinate="elevation">5.0</position>
    <position coordinate="distance">1.0</position>
```

(continues on next page)

[13]https://www.itu.int/rec/R-REC-BS.2076/
[14]https://www.itu.int/rec/R-REC-BS.2088/

(continued from previous page)

```
  </audioBlockFormat>
  <audioBlockFormat rtime="00:00:05.00000" duration="00:00:10.00000">
    <position coordinate="azimuth">-24.5</position>
    <position coordinate="elevation">6.0</position>
    <position coordinate="distance">0.9</position>
  </audioBlockFormat>
  <audioBlockFormat rtime="00:00:15.00000" duration="00:00:20.00000">
    <position coordinate="azimuth">-26.5</position>
    <position coordinate="elevation">7.0</position>
    <position coordinate="distance">0.8</position>
  </audioBlockFormat>
</audioChannelFormat>
```

By default, the values are interpolated between blocks. This means that at the beginning of a block the previous block's value is still used and the value specified for the current block is only reached at the end of the block. This also means that the very first block has an undefined value. In this case, the ADM specification recommends setting the `jumpPosition` flag, which applies the given value as a constant to the whole block. Interpolations can also be limited to a shorter time than the block duration with the `interpolationLength` value (in seconds), but this is discouraged by the specification. The ADM specification doesn't mention any further details about the exact type of interpolation to be used, but the illustrated examples hint at linear interpolation. Experts at ITU-R are currently working on a revision that will hopefully clarify the situation.

Apart from the positions of sound sources, their physical size can be specified with `width`, `depth` and `height` (in relative units).

Source signals can be boosted or attenuated by means of the `gain` value, which is interpreted as a linear value by default. The `gainUnit` option can be used to switch to decibels (dB). Gain values are interpolated just like positions, but again, the exact shape of the interpolation is not specified.

## 2.8 Spatial Sound Description Interchange Format (SpatDIF)

SpatDIF was first proposed in (Peters, Ferguson, et al. 2007), where it is described as a stream of Open Sound Control (OSC) messages which can optionally be stored in Sound Description Interchange Format (SDIF) files. Sound source positions are encoded in a listener-relative normalized coordinate system. They can be specified either in cartesian (x, y, z) or spherical (azimuth, elevation, distance) coordinates:

```
/SpatDIF/source/3/xyz -0.5 0.5 0.0
/SpatDIF/source/3/aed -45.0 0.0 0.0
```

In (Peters 2008), the coordinate system(s) can be chosen nearly arbitrarily:

```
/SpatDIF/*/xyz :/def right front top
/SpatDIF/*/xyz :/units meter meter meter
/SpatDIF/source.1/xyz -0.5 0.5 0.0
/SpatDIF/*/aed :/def clockwise
/SpatDIF/*/aed :/units deg deg meter
/SpatDIF/source.2/aed -45.0 0.0 0.707
```

As addition to a stream of tightly sampled values without any high-level structure, (Peters, Lossius, et al. 2013) introduces a so-called *authoring layer* which theoretically allows defining complex movements more compactly. No concrete examples are given, but some ideas for a trajectory extension can be found on the project's (archived) Wiki page[15].

The paper stresses that "SpatDIF is a syntax rather than a programming interface or file format" and the examples for file storage are expanded to use XML and YAML (in addition to the aforementioned OSC and SDIF). An abridged version of one of the example files[16] is provided here in OSC format:

```
/spatdif/version 0.3
/spatdif/meta/media/1/type file
/spatdif/meta/media/1/location "../audio/hello.wav"
/spatdif/time 0.0
/spatdif/source/1/position 0.0 5.0 0.0
/spatdif/source/1/media/1
/spatdif/source/1/media/1/loop/type repeat
/spatdif/source/1/media/1/gain 1.0
/spatdif/time 0.0087944
/spatdif/source/1/position 0.174 4.996 0.
/spatdif/time 0.0176514
/spatdif/source/1/position 0.349 4.986 0.
/spatdif/time 0.0265303
/spatdif/source/1/position 0.522 4.97 0.
```

The individual OSC messages do not carry timestamps, but `time` messages can be used to provide timing information for the immediately following messages. The same example is also provided in XML format:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<spatdif version="0.3">
  <meta>
    <ordering>time</ordering>
    <media>
      <id>1</id>
      <type>file</type>
      <location>../audio/hello.wav</location>
    </media>
  </meta>
  <time>0.0</time>
  <source>
    <name>1</name>
    <position>0.0 5.0 0.0</position>
    <media id="1">
      <gain>1.0</gain>
      <loop>
        <type>repeat</type>
      </loop>
    </media>
  </source>
  <time>0.0087944</time>
  <source>
    <name>1</name>
    <position>0.174 4.996 0.0</position>
  </source>
  <time>0.0176514</time>
```

(continues on next page)

[15]https://web.archive.org/web/20220629192742/http://redmine.spatdif.org/projects/spatdif/wiki/Trajectory_Extension
[16]https://web.archive.org/web/20220819072708/http://spatdif.org/examples.html

(continued from previous page)

```
  <source>
    <name>1</name>
    <position>0.349 4.986 0.0</position>
  </source>
  <time>0.0265303</time>
  <source>
    <name>1</name>
    <position>0.522 4.97 0.0</position>
  </source>
</spatdif>
```

The initial SpatDIF tools were only able to run on proprietary software. Later, an fully open source library implementation[17] was presented in (Miyama et al. 2013).

Version 0.4 of SpatDIF is presented in (Schacher et al. 2016), which promises “the ability to define and store continuous trajectories on the authoring layer in a human-readable way.” Cubic Bézier curves (see section B.2.6 in the appendix) can be used to define trajectories. By default, movement along those trajectories happens with a constant speed, but *easing curves* can be specified – either selected from a number of easing functions or defined by yet another cubic Bézier curve. In the latter case, a nearly arbitrary mapping between time and position along the curve can be provided, which allows moving sound sources forwards and backwards along a trajectory. Due to backwards compatibility concerns, SpatDIF v0.4 doesn't allow transmitting or storing trajectories on their own. A sampled (discretized) version of each trajectory has to be stored as well. According to Schacher et al. (2016), “...authoring is expected to be done with software tools that provide graphical user interfaces; hence there should be little or no need to interact directly with cubic-bezier parameter values.” Unfortunately, no information about version 0.4 could be found on the (now defunct) SpatDIF website and no example files seem to be available.

## 2.9 Spat-SDIF

The Sound Description Interchange Format (SDIF) does not natively support spatialization, but as suggested in the previous section, it can be extended for storing source positions and similar data. One way of doing this is by using Spat-SDIF as described in (Bresson and Schumacher 2011). One or more control values can be stored in an SDIF frame, and each of these frames contains its own timestamp. The Spat-SDIF player application[18] can generate real-time SpatDIF-compatible OSC messages.

## 2.10 Toolbox for Acoustic Scene Creation And Rendering (TASCAR)

TASCAR is a software for creation and rendering of dynamic virtual acoustic environments, developed for application in hearing aid research and audiology (Grimm et al. 2019). It comes with its own XML-based file format. This is a snippet from the example file `example_vertices.tsc`, as shown in the user manual[19]:

[17]https://github.com/SpatDIF/SpatDIFLib
[18]https://github.com/j-bresson/Spat-SDIF-Player
[19]https://tascar.org/manual.pdf

```
<source name="piano" color="#101077">
  <position>
    0 -3.2 1.7 1.4
    10 3.2 2.7 1.4
  </position>
  <orientation>0 -24 0 0</orientation>
  <sound name="leftside" x="-0.7">
    <plugins>
      <sndfile name="sounds/jazzclub-piano1.wav" level="75"/>
    </plugins>
  </sound>
  <sound name="rightside" x="0.7">
    <plugins>
      <sndfile name="sounds/jazzclub-piano2.wav" level="75"/>
    </plugins>
  </sound>
</source>
```

Sound sources can have trajectories, which can be defined by specifying coordinate triples, prepended with the time (in seconds) of their occurrence. Position values between those times are linearly interpolated. An *interpolation mode* can be chosen between *cartesian* (which is the default) and *spherical*. Latter mode interpolates in arcs around the origin, which is probably only useful if the receiver is positioned at the origin, which, admittedly, is a common case.

Sound sources can also have an orientation, which – unlike in most computer graphics applications – is applied to the source object after the position. The interpolation of orientations is not explicitly mentioned in the user manual, but according to the current implementation, the three Euler angles seem to be interpolated separately, which works for rotations around one of the coordinate axes, but it hopelessly breaks in the general case, see section B.3.12 of the appendix.

## 2.11 Synchronized Multimedia Integration Language (SMIL)

The SMIL – which is supposed to be pronounced like “smile” – is a format for temporal control and synchronization of audio, video, images and text elements and their arrangement on a 2D screen. It is a recommendation[20] of the World Wide Web Consortium since 1998; the current version (SMIL 3.0) was released in 2008. In contrast to the formats mentioned in previous sections, SMIL was never intended for describing 3D scenes, but extending it to three dimensions has been suggested (Goose et al. 2002; Pihkala and Lokki 2003).

However, the interesting part of this format are not its spatial aspects nor its rather limited audio capabilities but rather its handling of the timing[21] of content elements. Most notably, it uses so-called *time containers* as a top-level structure. All elements in a `<par>` container are played back in *parallel* and all elements in a `<seq>` container are played back *sequentially*. These containers can be arbitrarily nested, building up a single hierarchical structure called *time graph*. Inside those time containers, media files are linked to the SMIL file with `<img>`, `<audio>`, `<text>` and similar elements. The following example shows how the time containers can be used to define a slide show. The background image and music are played/shown in parallel to slides, which are themselves shown in sequence.

[20]https://www.w3.org/TR/SMIL/
[21]https://www.w3.org/TR/SMIL/smil-timing.html

```xml
<?xml version="1.0"?>
<!DOCTYPE smil PUBLIC "-//W3C//DTD SMIL 2.0//EN"
                      "http://www.w3.org/2001/SMIL20/SMIL20.dtd">
<smil xmlns="http://www.w3.org/2001/SMIL20/Language">
  <head>
    <meta name="title" content="Example Slideshow"/>
    <layout>
      <root-layout width="240" height="270"/>
      <region id="background" left="0" width="240" top="0" height="270"/>
      <region id="images" left="18" width="220" top="6" height="240"/>
      <region id="captions" left="19" width="220" top="184" height="85"/>
    </layout>
  </head>
  <body>
    <par>
      <audio src="background-music.mp3" systemComponent="multiChannelAudio
↪"/>
      <seq>
        <par dur="8s">
          <img region="images" dur="8s" src="title.png"/>
          <audio begin="1.2s" src="title-sound.mp3"/>
        </par>
        <par>
          <img region="images" dur="5s" src="image01.png"/>
          <text region="captions" dur="5s" src="caption01.txt"/>
        </par>
        <par>
          <img region="images" dur="5s" src="image02.png"/>
          <audio begin="2s" src="sound-effect02.mp3"/>
        </par>
        <!-- some slides omitted for brevity -->
        <par>
          <img region="images" dur="5s" src="image74.png"/>
          <text region="captions" dur="5s" src="caption74.txt"/>
        </par>
      </seq>
      <img region="background" src="background.png" width="240" height=
↪"270"/>
    </par>
  </body>
</smil>
```

In this case, there is no need to specify any start times because the start times of the slides are determined by the duration of the preceding slides. If desired, however, elements can have a start and end time relative to their parent or sibling elements. Those times can also be specified relative to the start or end of non-adjacent elements in the time graph and they can be triggered by user actions, for example mouse clicks. Graphical elements can be animated along linear or smooth 2D-paths with the `animateMotion` element.

## 2.12 Bottom Line

None of the presented 3D audio scene description formats explicitly targets scene authoring. All of the text-based formats can of course be created in a plain text editor, but their syntaxes are not optimized for manual creation. The declarative definition of source movements is often limited to linear interpolation. One counter-example that allows smooth interpolation is X3D (see section 2.3). In X3D, time instances can be specified for each control point, which affects the velocity along the trajectory. However, the resulting speed between control points might be unexpected. It is not possible to control the shape and the speed of a trajectory separately and it is not straightforward to define a smooth trajectory with constant speed. Interpolation between orientations is in most cases not supported, except for Slerp in VRML (section 2.2) and Squad in X3D. In the latter case, the standard text is quite vague and it is doubtful whether different implementations agree in their behavior – if this feature is implemented at all. The next chapter will present a new format that tries to overcome these shortcomings.

# Chapter 3

# Development of a Scene Authoring Format: ASDF

As part of this thesis, the Audio Scene Description Format (ASDF) has been developed. The first ideas for this format have already been presented by Geier, Ahrens, and Spors (2008a) and Geier and Spors (2008). One of the goals was – and still is – to define a single common scene description format that could be used for any number of different reproduction methods to be able to conveniently compare them. For an overview of some of those reproduction methods and the emergence of *object-based* audio reproduction see the end of chapter 1. To ensure the smooth exchange of scenes between different reproduction systems, the ASDF is independent of the rendering algorithm and contains no implementation-specific or platform-specific data. Scenes can be played back on both loudspeaker-based and headphone-based systems. Another goal was to provide a reasonably simple means for everyone to create three-dimensional spatial audio scenes and experiment with them. This is facilitated by the ASDF library implementation described in chapter 4 and especially by the software integrations described in sections 4.5 and 4.6.

Geier, Spors, and Weinzierl (2010) state that "the ASDF aims at being both an authoring and a storage format at the same time. In absence of a dedicated editing application, scene authors should still be able to create and edit audio scenes with their favorite text editor." This is still true, but recently the focus has shifted even more towards authoring. It is easy to create a lower-level representation from a high-level description, but it is much harder – and often lossy – to go into the other direction. The main focus of the ASDF is to be able to define elaborate sound source trajectories by specifying only a few points in three-dimensional space. Not only does this need less storage space than densely sampled movement data, it is also much easier to edit existing trajectories.

The exact syntax of the examples in the aforementioned primordial papers is slightly different, but most of the basic ideas are already there. The full documentation – including many examples – for the current version of the ASDF is available in appendix A. The ASDF in its current form is focused on describing three-dimensional movements – including rotations – to allow experimenting with complex trajectories. Detailed background about the underlying research into various types of splines can be found in appendix B. All other features are deliberately minimalistic. For this thesis, the choice was made to concentrate mainly on one topic – movement over time – and thoroughly investigate position and rotation splines in considerable depth, instead of developing a fully-featured scene description format where each feature receives only a superficial treatment.

Section 3.4 shows an incomplete list of missing features that may or may not be implemented in future versions of the format. The goal was not to come up with as many features as possible, but to define a few basic features and make a working implementation to be able to thoroughly assess their utility. Geier, Spors, and Weinzierl (2010) promise that "a reference implementation will be provided in form of a software library." This library is now finally available, and it is described in chapter 4.

The analysis of existing formats in chapter 2 has shown that most of them use linear interpolation between the given object positions. A few formats allow specifying (cubic) splines, but they are complicated to define and hard to use without a separate editor application. Control over the speed of objects along trajectories is limited. Interpolation between different orientations of objects and groups of objects is rarely possible at all, and if it is, often only spherical linear interpolation (see section B.3.2 in the appendix) is supported. Only the X3D format (see 2.3) supports smooth interpolation of orientations, but there is still room for improvements.

The ASDF uses splines to provide trajectories that can be defined by a simple sequence of points in space. Optionally, the time of the given points can be specified, which in turn influences the speed along the trajectory. The same approach as for position splines is also used for orientation splines. Both position and orientation values can be combined in so-called *transforms* (see section 3.2).

## 3.1 Storage Format and Syntax

Since the ASDF is designed to be an authoring format, it is an obvious choice to use text-based files to store scene descriptions. This way, it is possible to create and edit scenes with any text editor, without the need for specialized software. This does not include audio data, though. Audio signals are stored in separate files using common binary audio formats and they are linked to the text-based scene description via their file name. The audio files still have to be recorded and edited with specialized software, but audio editing applications are widely available.

Originally, the ASDF was planned to become an extension to the SMIL format (Geier, Ahrens, and Spors 2010; Geier, Spors, and Weinzierl 2010), which was presented in section 2.11. This way, it would be possible to use an existing SMIL library to get the timing and media management for free and only implement the 3D audio aspects. Sadly, the SMIL format didn't stand the test of time and nowadays it is not really used anymore and no usable libraries are available. Instead, the ASDF was developed as a new format from scratch. Staying in the tradition of SMIL, XML[1] was chosen as a storage format, because it lends itself to representing the potentially deeply nested containers of the time graph. In recent years, XML-based formats have lost a lot of popularity, often being replaced by more modern text-based formats like JSON[2] or YAML[3]. However, XML is still uniquely suited to represent nested containers, each of which carrying additional information in form of attributes.

For a description of the entire syntax of the ASDF see appendix A.

## 3.2 Transforms

The focus of this thesis is the movement of sources (and the virtual listener) in three-dimensional audio scenes. In the ASDF, this movement can be described by means of transforms. Apart from movement – comprised of translation and rotation of objects – transforms can also be used to control the volume of the source signals. Transforms can be static, but – more interestingly – they can also change over time. When defining a transform with a sequence of positions, a *spline*[4] is created, which is used to smoothly animate the object position over time. Similarly, splines can be created for orientations

[1]`https://www.w3.org/TR/xml/`
[2]`https://www.json.org/`
[3]`https://yaml.org/spec/1.2/`
[4]For a definition of the term *spline* see section B.2.3 in the appendix.

and even for volume changes over time. The coordinate system conventions of ASDF are explained in section A.2 of the appendix, the syntax for defining transforms is shown in section A.3.7 and more details about the splines used in the ASDF can be found in section A.5.

All positions and orientations are generally three-dimensional. However, in simple cases where movements are limited to the horizontal plane, positions can be specified as two-dimensional coordinates and rotations can be defined by simple azimuth angles. The goal is to make it simple to create simple movements, while still making it possible to create arbitrarily complicated ones. Not only can transforms be applied to sound sources and to the listener position, but they can also be applied to other transforms. This way, complex movements can be achieved by combining multiple simpler movements.

A transform can be applied to multiple objects at the same time. Similarly, one object can be the target of multiple transforms. However, this is not allowed if there is more than one rotation involved, since the order of rotations would be unspecified, which would make the resulting orientation ambiguous. If multiple rotations are desired, their order has to be explicit. This can be done by applying the first transform to the target object, then applying the second transform to the first one (and the third to the second and so on, if desired).

Just like audio clips, transforms have a start and an end. They can be placed in the timeline relative to other transforms and audio clips, as explained in the following section.

## 3.3 Temporal Structure

Contrary to the majority of formats described in chapter 2, the spatial transforms don't make up the primary structure of ASDF files. Instead, the timeline is the central data structure, which is inspired by the SMIL format (see section 2.11). The timeline is defined by so-called *time containers* in the form of `<seq>` and `<par>` elements. These containers can be used to play audio clips one after another or at the same time, respectively. They can also contain transforms, which can be placed in time relative to audio clips and to other transforms. Time containers can also contain other time containers and they can be arbitrarily nested. See section A.3.5 in the appendix for some examples.

The timing of audio clips and of the transforms that might affect their spatial trajectories is independent. A transform might only affect a part of an audio clip or a whole playlist of clips. On the other hand, a single audio clip might be affected by a sequence of multiple transforms. As mentioned in the previous section, transforms can be applied to other transforms, leading to a combined overall transform. However, the involved transforms can start and end at different times. Each transform only has an effect while it is active. This means that when a sound source is moved along a trajectory, it does not stay at the final position when the trajectory is finished.

It is important that transforms can change over time, but sometimes only static transforms are needed. The ASDF provides a simplified syntax where transform attributes like `pos` or `rot` can be applied directly to audio clips, which means that those transforms are constant for the whole duration of the clip.

## 3.4 Out of Scope

This thesis is mostly about describing three-dimensional movements over time, and this is also the focus of the ASDF. There are many more features that a scene description format could have – and some of the formats from section 2 do have. The following (incomplete) list presents some features that could be interesting but were not included in the ASDF in order to limit the scope of the project. Some of these features could be considered for a future version of the format.

**Graphics** The ASDF is meant for pure audio scenes. There are many applications for audiovisual scenes, which are typically focused on the visual part. A range of 3D animation software and game engines are available, many of them already have support for spatial audio, and if not, it can be added via plugin interfaces. A declarative authoring format in this area would on the one hand be much more complicated than an audio-only format, and on the other hand it would probably not provide any advantage over what is already available in existing software solutions.

**Interactivity** Currently, the ASDF does not support any interactivity because deterministic scenes allow for a much simpler implementation. The whole scene is loaded into memory and cannot be changed during playback. It might be interesting to provide "hooks" for defining user interaction, as it is possible in the SMIL format (see section 2.11), for example.
However, if a lot of interactivity is desired, there might be more straightforward approaches like using existing game engines, which are centered about interactivity. A small amount of interactivity would probably still be useful for triggering different "parts" of an audio scene. For example, a spatial music composition might combine a dynamic audio scene with live musicians playing their instruments. In such a case it would be helpful to be able to synchronize the movements in the audio scene with the live performances.

**Perceptual Parameters** In addition to physical parameters, some formats like MPEG-4 (see section 2.4) also support perceptual parameters like *source presence*, *source brilliance*, *room presence* and *envelopment* (Väänänen 2003). Something like this might be added to a future ASDF version, but for now, it is limited to describing the physical positions of sound sources and the volume of their source signals.

**Room Acoustics/Reverberation** The ASDF does not allow the description of physical or perceptual room properties. The lack of built-in reverberation in a scene can be compensated by adding additional sound sources containing pre-recorded reverb signals, be it artificially created or from a real recording.

**Scene Scaling** It is likely that different reproduction venues have different room sizes. Sound source positions and trajectories created for a small room might not make sense in a large room. It would be useful if scenes could automatically be re-scaled based on the reproduction setup. The ASDF currently doesn't provide any tools for this, but some of the formats mentioned in section 2 use relative coordinates to tackle this problem, and so do object-based cinema systems (Robinson and Tsingos 2015). However, the psychoacoustic consequences of such a scaling are unclear.

**Ambisonics sources** Ambisonics microphones can be used to make spatial recordings of ambient sounds. Those recordings can be stored as *B-format* files. It would be useful to include such recordings in an object-based scene description, but the ASDF currently does not support that.
Channel-based recordings like 5.1 files can be used as audio clips, but their virtual loudspeaker positions have to be specified manually.

**Trimming Audio Clips** Audio clips in the ASDF can only be played to their full length. They cannot skip parts of the file in the beginning or in the end. Similarly, partial repetitions are not supported. It might be desirable to allow that, but it would make the implementation as well as the usage more complicated.

**Multiple Listeners** The ASDF allows the definition of a single *reference*, i. e. the position of a listener. This *reference* can be animated with *transforms* just like sound sources. However, there are other systems which support multiple listener positions at the same time. This would make it possible to "cut" between different listener positions like cutting between camera angles in a movie.

**Streaming** There are two aspects of streaming that can be considered. On the one hand, source signals could be provided via audio streams from a network. This is not possible in the current ASDF, but it could be implemented reasonably easily, if desired. On the other hand, the whole scene description could be streamed, as it is done, for example, in MPEG-4 (see section 2.4). This adds a lot of complexity, which arguably isn't worth the effort. ASDF scenes have to be parsed in their entirety before playback can start.

# Chapter 4

# Implementation and Integration of an ASDF Library

The development of the Audio Scene Description Format (ASDF) has been described in chapter 3 and its full specification and documentation can be found in appendix A. However, a file format is only useful if there is software available that can load files with that format. Therefore, a software library[1] has been implemented as well, using the Rust[2] programming language. The following sections describe the implementation of this library. A library alone is still not enough to listen to audio scenes, it also has to be integrated into a software that is able to reproduce those audio scenes. The ASDF library has been integrated in a standalone rendering software (see section 4.5) and in a plugin for a visual programming language for multimedia (see section 4.6). These integrations use the C language interface of the ASDF library which makes it easy to integrate with software written in other languages.

## 4.1 ASDF Parsing

The ASDF syntax is based on XML. An existing XML parser library[3] is used to load ASDF files. The XML library automatically checks whether a file is well-formed XML, but the conformance to ASDF syntax is checked manually. Manual validation makes it easier to provide helpful error messages in case an erroneous ASDF file is provided. It is important to provide informative and very specific error messages to help scene authors to create working scenes quickly. For example, the following small scene is well-formed and valid XML, but it still contains an error:

```xml
<asdf version="0.4">
  <clip file="non-existing.ogg">
    <channel pos="-1.5 3" />
    <channel pos="1.5 3" />
  </clip>
</asdf>
```

When trying to load this audio scene – assuming the referenced audio file indeed does not exist – the ASDF library would provide this error message:

```
Error: Error loading audio file
---
<asdf version="0.4">
```

(continues on next page)

[1] https://github.com/AudioSceneDescriptionFormat/asdf-rust
[2] https://www.rust-lang.org/
[3] https://crates.io/crates/xmlparser

(continued from previous page)

```
  <clip file="non-existing.ogg">
             ^^^^^^^^^^^^^^^^
---
error details: I/O error
error details: No such file or directory (os error 2)
```

If the audio file is changed to one that exists, but has only one channel, a different error will be raised:

```
Error: Too many <channel> elements: file has only 1 channel(s)
---
<asdf version="0.4">
  <clip file="mono-recording.ogg">
    <channel pos="-1.5 3" />
    <channel pos="1.5 3" />
  </clip>
    ^^^^
---
```

Showing the affected part of the scene within the error message should help localizing the cause of the error more quickly, especially in a non-trivial scene that is much larger than this minimalistic example.

## 4.2 Audio File Playback

The ASDF library allows playback of WAV, OGG (Vorbis), FLAC and MP3 sound files. It is not feasible to load all sound files of a scene in their entirety into memory. A scene might contain many very long files that are supposed to be played at the same time. It is sufficient to provide the audio data in small blocks. At any time during playback, though, the appropriate parts of the currently active sound files have to be provided to the rendering software within a very short time. Otherwise, the output signal could be interrupted, leading to audible artifacts. To avoid this, a certain amount of data from all relevant audio files is buffered, which makes sure that there are no excessive delays when reading and decoding the files. The buffering happens in a separate thread which runs in parallel to the audio processing, and typically with a lower priority. This parallel processing in multiple threads is traditionally very error-prone. This was the main reason why the programming language Rust was chosen to implement the library. Rust allows the implementation of very efficient programs that are still safe in the presence of multiple threads.

An additional requirement that complicates the implementation is that a user should be able to *seek* forward and backward in a scene. It should be possible to jump to any time within the timeline of a scene. When that happens, playback is stopped, all existing buffers are discarded, new buffers are filled with the audio data that is to be played back at the new time instance and finally playback is resumed at the desired point in time.

Different audio files in a scene can have different sampling rates, which could again be different from the output sampling rate of the reproduction software. In case of a mismatch, audio data is automatically re-sampled.

## 4.3 Transforms

As explained in chapter 3.2, so-called *transforms* are used to define spatial positions of objects and to change positions and orientations over time. When a sequence of positions and/or orientations and/or volume values is given in a transform, *splines* are created based on the given control values. A separate Rust library[4] has been implemented to provide all necessary types of splines. Within the main ASDF library, all transforms – including their splines – are created when loading an ASDF file and they are stored for quick retrieval by the IDs they are applying to.

Whenever the transform of a source (or of the listener) is requested for a given time, it is checked whether there are any transforms applying to its ID. Only for those transforms that are active at the given time, it is recursively checked for other transforms applying to them. Each transform represents a local coordinate system and all those nested transforms are combined into a final transform. If a transform contains both a translation and a rotation, the rotation is applied first. When multiple transforms apply to the same ID, only one of them is allowed to have a rotation, because the order of rotations would be ambiguous. This is checked when loading a scene and an error is raised if there are multiple rotations at the same level. A transform is not allowed to apply to itself, directly or indirectly. This is also checked when loading the scene.

## 4.4 API

The interface of the library, often called Application Programming Interface (API), allows to choose a scene file to load and to specify a few rendering options like the sampling rate and block size. After loading the scene, two main types of data are provided: audio signals of sound sources and their spatial transforms. The spatial transform of the listener's position (also called *reference*) is also available.

Audio data is provided in fixed-size blocks with the given block size and with as many channels as there are sources in the scene. If a source is not active at a given time, its signal consists of all zeros. Each audio block is provided immediately, thanks to the buffering described in section 4.2 above, which enables glitch-free playback.

It is also possible to *seek* in a scene, in other words, to jump to a different point in time. In this case, the next audio block is faded out and some empty audio blocks are delivered during buffering. Once the buffers are filled, the next audio block is faded in and playback continues at the new scene time.

While the audio data is provided in consecutive blocks, the spatial transform for any source (or for the reference) can be queried at any desired scene time. Typically, a rendering application will obtain the transforms once per audio block and interpolate or cross-fade from the previous values to the current values. However, it is possible to query the values at a higher or a lower rate, if desired. It is theoretically possible to query the transform values at every audio sample, but this will lead to a high computational load. The library is providing a transform value for each queried time instance, even if the value did not change compared to the previous query. If needed, the host application can filter out repeated values. This is done in the Pure Data external, see section 4.6.

Since the ASDF library is implemented in Rust, it can be easily used in any Rust program. However, having only a Rust API would severely limit the possible use cases. Therefore, C language bindings have been implemented using a widely used tool[5] from the Rust ecosystem. This extends the potential uses of the ASDF library to applications written in C (see section 4.6), C++ (see section 4.5) and a myriad of other languages.

[4]https://github.com/AudioSceneDescriptionFormat/asdfspline-rust
[5]https://github.com/lu-zero/cargo-c

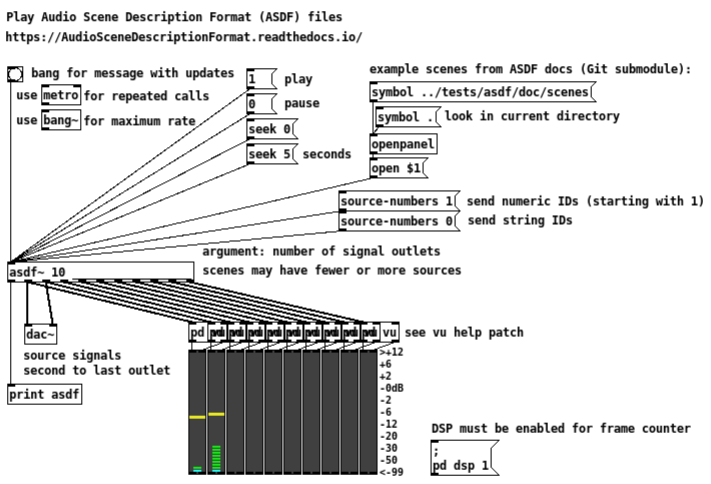

Play Audio Scene Description Format (ASDF) files
https://AudioSceneDescriptionFormat.readthedocs.io/
bang for message with updates
use metro for repeated calls
use bang~ for maximum rate
1 play
0 pause
seek 0
seek 5 seconds
example scenes from ASDF docs (Git submodule):
symbol ../tests/asdf/doc/scenes
symbol . look in current directory
openpanel
open $1
source-numbers 1 send numeric IDs (starting with 1)
source-numbers 0 send string IDs
argument: number of signal outlets
scenes may have fewer or more sources
asdf~ 10
dac~
source signals
second to last outlet
print asdf
pd vu see vu help patch
>+12
+6
+2
-0dB
-2
-6
-12
-20
-30
-50
<-99
DSP must be enabled for frame counter
;
pd dsp 1

**Figure 1:** *Help patch of the* `asdf~` *external for Pure Data*

## 4.5 Integration in a Standalone Rendering Application

The development of the ASDF happened in the context of a spatial audio rendering tool called SoundScape Renderer (SSR)[6] (Geier, Ahrens, Möhl, et al. 2007; Geier, Ahrens, and Spors 2008b; Geier and Spors 2012). The SSR was co-developed with Jens Ahrens and several collaborators[7]. The SSR can be used to render an audio scene with several reproduction methods, including dynamic binaural rendering, Wave Field Synthesis (WFS) and Higher-Order Ambisonics (HOA). It is based on an efficient multi-threaded rendering architecture (Geier, Hohn, et al. 2012) and it can be used both as a standalone application and as *externals* for Pure Data[8].

As part of the development of the ASDF, the ASDF library has been integrated into the SSR. At the time of writing, binaural rendering is the only three-dimensional reproduction method, the other rendering methods are limited to the horizontal plane. The default graphical user interface is also still limited to the horizontal plane, but an experimental three-dimensional user interface is also available, see section 4.7.

The SSR uses the JACK[9] audio server for the handling of audio signals. The audio file playback of the ASDF library has been integrated with the JACK *transport* mechanism using the *seek* functionality mentioned in section 4.2. This way it is possible to play ASDF scenes in synchrony with other JACK-enabled applications.

## 4.6 Integration as an External for Pure Data

The ASDF library implementation contains[10] an *external* for Pure Data[11] as a usage example for the C interface. Figure 1 shows a screenshot of the external's help patch. The number of signal outlets has to be provided when creating the external. Only that number of source signals are provided, even if more sources are defined in the scene. Playback of the scene can be started and stopped at any time and it is possible to *seek* to any point in time in the scene. The transforms of all sources and the listening position are not auto-

[6]http://spatialaudio.net/ssr/
[7]https://ssr.readthedocs.io/general.html#contributors
[8]https://puredata.info/
[9]https://jackaudio.org/
[10]https://github.com/AudioSceneDescriptionFormat/asdf-rust/tree/master/pure-data
[11]https://puredata.info/

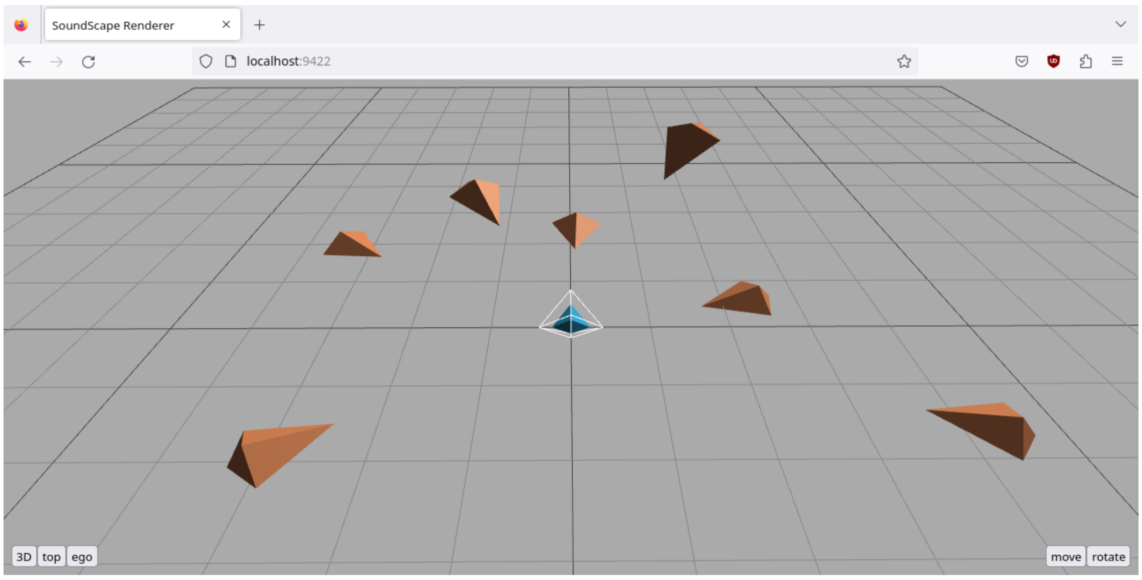


**Figure 2:** *Screenshot of the browser-based 3D GUI prototype*

matically provided at the message outlet, but they can be triggered at any time by sending a *bang* message to the external. However, all values that have not changed since the last request are filtered out.

Given the signals for each source and the corresponding messages with transform updates, any means of spatialization available in Pure Data can be used to auralize a scene. One possibility are the renderer externals provided by the SSR, see section 4.5. The messages sent by the ASDF external happen to be compatible with the SSR externals, the two just have to be connected. Both the ASDF and SSR externals can be installed from Pure Data's built-in package manager.

## 4.7 Visualization

The ASDF is – as its name suggests – an audio-only format with no visual aspects. Nevertheless, it is really helpful for scene authors to see a visual representation of the sound sources in the audio scene they are creating. The ASDF library itself has no capabilities for visualization, but to aid the development of the ASDF, a tool for three-dimensional visualization has been developed for the SoundScape Renderer (see section 4.5). The tool has been implemented using the WebGL-based JavaScript library three.js[12]. More specifically, it is based on the three.js editor[13]. This makes it easy to create a quick prototype for a three-dimensional user interface that can be displayed with any modern browser. The communication with the SSR takes place over the WebSocket protocol, which is supported by all modern browsers. A screenshot of web browser displaying an audio scene is shown in figure 2.

The visualization is quite basic and there are many desirable features that are missing, but it is already possible to get an idea about the three-dimensional movements of scene objects while the scene is playing.

[12]https://threejs.org/
[13]https://threejs.org/editor/

## 4.8 Example Scenes

Many minimalistic scenes are available in the ASDF documentation, see appendix A. Some larger example scenes are available for download from the ASDF development pages[14].

[14]https://github.com/AudioSceneDescriptionFormat/asdf-example-scenes

# Chapter 5
# Conclusion and Future Work

The previous chapters have shown the definition of the Audio Scene Description Format (ASDF), the implementation of a library for loading ASDF files and its integration into different software for spatial audio reproduction. All involved programs and libraries are available as open-source software and for free. Everyone can use the ASDF to create spatial audio scenes and the ASDF library and its integrations can be used to play them back. The software implementation can also be used as a basis for further experimentation and to prototype features that are currently not implemented.

There has been no systematic evaluation of the format yet, but everyone is encouraged to try it out and make their own judgement. As chapter 2 has shown, there are currently no dedicated authoring formats for spatial audio scenes available. Therefore, no direct comparison is possible. From the example scenes in appendix A it should be apparent that the ASDF syntax is more concise and easier to hand-write than any of the other text-based formats shown in chapter 2. The feature set of the ASDF, however, is very much limited compared to some of the other formats that were mentioned. The scope of the format is intentionally chosen to be very narrow. It is focused on the description of movements and rotation of scene objects over time. This description is based on different types of *splines*, which are covered in considerable depth in appendix B. Nearly everything else is out of scope, as section 3.4 describes.

The implementation of the ASDF library covers all features of the format as currently defined, but of course additional functionality could be implemented. One example would be the recording of movements using a tracking system, followed by a data reduction by means of some kind of *curve fitting*, which would allow to create a high-level declarative description from a low-level stream of sampled data.

The three-dimensional GUI prototype shown in section 4.7 could of course be improved in many ways. It would be a massive endeavour, but maybe an application could be implemented that allows creating and editing trajectories graphically, including the possibility to animate nested local coordinate systems. This would also need some kind of elaborate timeline editor to be able to define the relationships between objects in time. It is unlikely that such an extensive GUI project would be started (and more importantly, would lead to a usable program). More realistically, some features of the ASDF could probably be improved in order to simplify the scene authoring process via editing plain text files. The ASDF has its own issue tracker[1] where future features can be discussed.

Appendix B contains a thorough write-up about Euclidean splines and rotation splines, but it is of course by far not exhaustive, and more material – including more types of splines – can be added in the future. Some new findings might even lead to changes in future versions of the ASDF. For example, more and better end conditions could be brought forward, which could replace the end conditions that are currently used in the ASDF. There is still a lot of old and new literature that has not been incorporated. Many

[1]`https://github.com/AudioSceneDescriptionFormat/asdf/issues`

aspects that might be worth considering in the future are already mentioned in the issue tracker for the *splines* project[2].

The ASDF allows the creation of position trajectories and of orientation trajectories and it can even handle a combination of both. However, when a scene object or a group of objects moves along a position trajectory, it does not automatically change its orientation according to the curvature of the trajectory. This is a feature that might be desirable for scene authors.

The splines that are currently used in the ASDF guarantee a continuous change of velocity between spline segments, but the acceleration vector – i. e. the second derivative – is allowed to be discontinuous. It might be interesting to investigate whether choosing a different type of spline that guarantees continuity of acceleration will have any noticeable advantages.

Since all the specifications, software and documentation is publicly available, it should be easy for anyone who is interested to build upon this work.

[2]`https://github.com/AudioSceneDescriptionFormat/splines/issues`

# Appendices

# Appendix A

# The Audio Scene Description Format (ASDF)

This appendix contains the current version of the documentation for the ASDF, written by the author of this thesis. At the time of writing, a verbatim copy of this text is also available online at `https://AudioSceneDescriptionFormat.readthedocs.io/`. The online version might be updated in the future, though.

The example scenes shown in this chapter are available (including the referenced audio files) at `https://github.com/AudioSceneDescriptionFormat/asdf/` in the directory `doc/scenes/`. All scenes can be opened and played back with the software described in sections 4.5 and 4.6.

## A.1 Introduction

Let's start simple, with the file `minimal.asd`:

```
<asdf version="0.4">
  <clip file="audio/ukewave.ogg" pos="1 2" />
</asdf>
```

This plays the contents of the (mono) audio file `audio/ukewave.ogg`, coming from a spatial position of 2 meters in front and 1 meter to the right. For more details on the used coordinate system, see *Position and Orientation* (page 51).

If you want to play a file with more than one channel, you can provide positions for each of the channels, like shown in `minimal-multichannel.asd`:

```
<asdf version="0.4">
  <clip file="audio/marimba.ogg">
    <channel pos="-1 2" />
    <channel pos="1 2" />
  </clip>
</asdf>
```

This plays the contents of the (two-channel) audio file `audio/marimba.ogg`, each channel coming from its specified position. For further details, see *<clip> and <channel>* (page 56).

The examples above use a few shorthand notations to make frequently used scenarios a bit easier to type. Expanding most of the shortcuts used in the first example above would lead to the more complicated ASDF syntax shown in `minimal-expanded.asd`:

```
<?xml version="1.0"?>
<asdf version="0.4">
  <head>
    <source id="src1" />
  </head>
  <body>
    <seq>
      <clip file="audio/ukewave.ogg">
        <channel source="src1" pos="1 2 0" />
      </clip>
    </seq>
  </body>
</asdf>
```

Please note a few changes to the "minimal" version above:

- An XML declaration[1] has been added, which is optional in XML 1.0 (but not in XML 1.1).
- The *<head> and <body>* (page 52) elements are optional. The *<asdf>* (page 52) element (including version number) is always required.
- In the `<head>` section there is a separate *<source>* (page 52) element.
- The `<body>` element implicitly behaves like a `<seq>` element, see *<seq> and <par>* (page 55).
- Even though this is not necessary for a mono `<clip>`, a `<channel>` element has been provided explicitly. It has been associated with the *<source>* (page 52) that was defined in `<head>`.
- The z-component in `pos` is optional, see *<transform>* (page 57).

This still uses the shorthand of specifying the position directly in the `<channel>` element. As shown in `minimal-expanded-with-explicit-transform.asd`, it can be expanded even further:

```
<?xml version="1.0"?>
<asdf version="0.4">
  <head>
    <source id="src1" />
  </head>
  <body>
    <par>
      <clip file="audio/ukewave.ogg">
        <channel id="channel1" source="src1" />
      </clip>
      <transform apply-to="channel1" pos="1 2 0" />
    </par>
  </body>
</asdf>
```

[1] https://www.w3.org/TR/xml/#sec-prolog-dtd

- Because the `<clip>` and the `<transform>` happen at the same time, they are wrapped in a `<par>` element, see *<seq> and <par>* (page 55). Without this `<par>` element, the `<transform>` would only be active *after* the `<clip>` is finished (because the `<body>` element implicitly behaves like a `<seq>` element).
- If the clip has only one channel, it doesn't matter whether the `<transform>` is applied to the `<clip>` or to the `<channel>`. In this simple case it could be even directly applied to the `<source>`.
- The *<transform>* (page 57) element could be even further expanded to contain the `pos` information in a single `<o>` sub-element.

## A.2 Position and Orientation

The ASDF uses a right-handed cartesian coordinate system to specify positions in three-dimensional space. The x-, y- and z-axis can be thought of as pointing towards *east*, *north* and *up*, respectively, which is sometimes called an ENU system[2]. However, contrary to typical ENU systems, the default orientation in the ASDF is towards *north*, i.e. along the positive y-axis!

To understand the motivation for this choice of default orientation, imagine a treasure map lying on a table in front of you. The north direction typically points towards the top of the map and the east direction points to the right. On the other hand, if you had a piece of paper with a mathematical graph on it, the y-axis would point towards the top of the page and the x-axis would point to the right. Therefore it makes sense that the x-axis points towards east and the y-axis points northwards, right? Now imagine that you are sitting at the table with your treasure map in front of you. You will look straight ahead by default, and this happens to be northwards on the map. Therefore, the default orientation in the ASDF is towards north, which corresponds to the positive y-axis. To complete the triple of axes, the z-axis points up to the ceiling (or towards the zenith, if your table is in open air). Positive z-values are above the table, negative z-values are below the table. The resulting coordinate system is right-handed, which is convenient.

The coordinate values for positions are given in meters. The third coordinate is optional and defaults to zero.

As mentioned above, the default orientation (sometimes called *view* direction) is along the positive y-axis. To fully specify all three degrees of freedom, the default *up* direction is set to the positive z-axis (which should be an unsurprising choice). For specifying arbitrary rotations relative to this default orientation, up to three Tait–Bryan angles[3] can be specified. The first angle (*azimuth*) rotates around the z-axis, the second angle (*elevation*) around the (previously rotated) x-axis and the third angle (*roll*) around the (previously rotated) y-axis.

All angles are given in degrees. The *elevation* and *roll* angles are optional, with a default of zero. The sign of the rotation angles follows the right hand rule[4]. Rotations are specified in degrees because that is familiar to most people. However, for any further calculations in an ASDF library, the angles should be immediately converted to quaternions or rotation matrices, see *Implementation Notes* (page 70).

[2] `https://en.wikipedia.org/wiki/Axes_conventions`
[3] `https://en.wikipedia.org/wiki/Euler_angles#Tait-Bryan_angles`
[4] `https://en.wikipedia.org/wiki/Right-hand_rule#Rotations`

Multiple translations/rotations can be nested, which means that all coordinates are local with respect to the parent transform. For more details, see *Nested <transform>* (page 63).

## A.3 Elements

The following sections describe all XML elements that can be used in an ASDF file.

### A.3.1 `<asdf>`

An ASDF file must contain a single top-level `<asdf>` element with a required `version` attribute. Currently, only `version="0.4"` is supported.

The `<asdf>` element can optionally contain *<head> and <body>* (page 52) sub-elements.

If there is no `<body>` element, all sub-elements of `<asdf>` (except an optional `<head>` element) are treated as if they were contained in a `<body>` element, which in turn behaves like an implicit *<seq>* (page 55), see *<head> and <body>* (page 52). For example, the clips in `implicit-seq.asd` are played in sequence:

```xml
<asdf version="0.4">
  <clip file="audio/xmas.wav" pos="-2.5 0" />
  <clip file="audio/ukewave.ogg" pos="2.5 0" />
</asdf>
```

### A.3.2 `<head>` and `<body>`

Both `<head>` and `<body>` are optional. If there is a `<head>` element, it must be the first sub-element of *<asdf>* (page 52).

The `<head>` element can contain *<source>* (page 52) sub-elements and an optional *<reference>* (page 54). All elements within `<head>` exist for the whole duration of the scene. If they contain transform attributes like `pos` or `rot`, those values are static. Additional *<transform>* (page 57) elements can be used in the `<body>` to offset those values dynamically.

The `<body>` element can contain *<seq> and <par>* (page 55) elements, as well as *<clip>* (page 56) and *<transform>* (page 57) elements. If the `<body>` element contains multiple sub-elements, it acts like an implicit *<seq>* (page 55) element.

### A.3.3 `<source>`

`<source>` elements are defined within the `<head>` element and all sources exist for the entire duration of the scene.

#### A.3.3.1 File Inputs

*<clip> and <channel>* (page 56) elements can provide audio signals for `<source>` elements using the `source` attribute. If no `source` attribute is given, an unnamed `<source>` is implicitly created.

A `<source>` can be fed by multiple `<clip>` elements over time, but only if they don't overlap. If the `port` attribute (see below) is given, no `<clip>` elements can be assigned.

An implementation may re-use the same unnamed `<source>` for multiple non-overlapping `<clip>` elements, but this is not required.

#### A.3.3.2 Live Inputs

The `port` attribute can be used to provide live input signals, for example from microphones, external sound hardware or any software capable of producing audio signals (and connecting them with the software loading the ASDF scene).

The content of the `port` attribute isn't strictly specified and it is up to the reproduction software to interpret it.

For example, the SSR[5] provides an `--input-prefix` option to which the content of the `port` attribute is appended. By default, the prefix is `system:capture_` and appending numbers starting with `1` will select the corresponding hardware input channels.

The scene `live-sources.asd` shows an example of using the first 4 hardware inputs as sources:

```
<asdf version="0.4">
  <head>
    <source port="1" name="live input 1" pos="-1.5 2" />
    <source port="2" name="live input 2" pos="-0.5 2" />
    <source port="3" name="live input 3" pos="0.5 2" />
    <source port="4" name="live input 4" pos="1.5 2" />
  </head>
</asdf>
```

Live sources and sources driven by audio files can be mixed in one scene and *<transform>* (page 57) elements can apply to either. See e.g. `live-sources-and-file-sources.asd`:

```
<asdf version="0.4">
  <head>
    <source port="1" name="live input 1" pos="-1.5 2" />
    <source port="2" name="live input 2" pos="-0.5 2" />
    <source port="3" name="live input 3" id="three" />
    <source port="4" name="live input 4" pos="1.5 2" />
  </head>
  <body>
    <clip file="audio/xmas.wav" pos="0 2.5" />
    <!-- Source "three" is only active during this time -->
    <transform apply-to="three" pos="0.5 2" dur="1 min" />
    <clip file="audio/xmas.wav" pos="0 2.5" />
```

(continues on next page)

[5] `http://spatialaudio.net/ssr/`

(continued from previous page)

```
  </body>
</asdf>
```

### A.3.3.3 Transform Attributes

Any `<source>` element with an `id` attribute can be the target of a <*transform*> (page 57) (using the `apply-to` attribute). Like <*clip*> *and* <*channel*> (page 56), `<source>` can also use transform attributes like `pos`, `rot` etc. as a shortcut, see `source-transform.asd`:

```
<asdf version="0.4">
  <head>
    <source id="src-one" pos="-1 1" />
    <source id="src-two" pos="1 1" />
  </head>
  <clip file="audio/marimba.ogg">
    <channel source="src-one" />
    <channel source="src-two" />
  </clip>
  <clip file="audio/marimba.ogg">
    <channel source="src-two" />
    <channel source="src-one" />
  </clip>
</asdf>
```

## A.3.4 `<reference>`

The so-called *reference point* is a generalization of a *listener point*. In a headphone-based reproduction system it corresponds to the position (and orientation) of the listener's head in the virtual scene. In a loudspeaker-based system there might be multiple listeners, but the loudspeaker setup should still have a single *reference point*, which is typically somewhere in the center of the setup.

The `<reference>` can be specified explicitly within the `<head>` element and it can optionally have static transform attributes like `pos` and `rot`, as in the example scene `reference-transform.asd`:

```
<asdf version="0.4">
  <head>
    <reference pos="-1 1" rot="-45" />
  </head>
</asdf>
```

At most one `<reference>` element can be specified, and it implicitly has the reserved ID `"reference"`, which can be used as the target of a <*transform*> (page 57). If no `<reference>` element is given, the reference point can still be transformed using `apply-to="reference"`, as in `implicit-reference.asd`:

```
<asdf version="0.4">
  <par>
    <clip file="audio/ukewave.ogg" pos="0 0" />
    <transform apply-to="reference">
      <o pos="0 -1" />
      <o pos="-2 1" />
      <o pos="2 1" />
      <o pos="closed" />
    </transform>
  </par>
</asdf>
```

### A.3.5 <seq> and <par>

Both audio clips and `<transform>` elements are objects that have a certain duration. They can be placed in the timeline one after another by putting them into a `<seq>` (which means *sequential*) element. To delay an object or to create a pause between two objects, a *<wait>* (page 64) element can be inserted into the sequence.

To reproduce two or more clips and/or `<transform>` elements at the same time, you can put them into a `<par>` (which means *parallel*) element.

`<seq>` and `<par>` elements can be arbitrarily nested.

For a simple example, see `seq-par.asd`:

```
<asdf version="0.4">
  <par>
    <clip file="audio/ukewave.ogg" pos="0 2" />
    <seq>
      <clip file="audio/marimba.ogg">
        <channel pos="-1 2" />
        <channel pos="1 2" />
      </clip>
      <clip file="audio/xmas.wav" pos="-1.5 0" />
    </seq>
  </par>
</asdf>
```

If there is no `<body>` element, the main *<asdf>* (page 52) element implicitly behaves like a `<seq>` element, i.e. all contained elements are played in sequence, like in the example file `implicit-seq.asd`:

```
<asdf version="0.4">
  <clip file="audio/xmas.wav" pos="-2.5 0" />
  <clip file="audio/ukewave.ogg" pos="2.5 0" />
</asdf>
```

Within a `<par>` element, the first sub-element determines the duration of the whole `<par>` element. Any following sub-elements must not be longer than the first. A useful pattern is to use a `<clip>` as first sub-element (which defines the length of the `<par>`) and one or more `<transform>` elements afterwards, which will by default “inherit” the duration of the `<clip>`.

#### A.3.5.1 `repeat`

`<seq>` and `<par>` elements can be repeated, see *Repetition* (page 65).

### A.3.6 `<clip>` and `<channel>`

To load an audio file, a `<clip>` element can be inserted at the spot in the timeline where it should be played back. Each `<channel>` of a multi-channel file can have its own static transform attributes (`pos`, `rot`, etc.), as shown in the example scene `minimal-multichannel.asd`:

```
<asdf version="0.4">
  <clip file="audio/marimba.ogg">
    <channel pos="-1 2" />
    <channel pos="1 2" />
  </clip>
</asdf>
```

If the audio file only has a single channel, an explicit `<channel>` element is not necessary. If desired, transform attributes can be applied to the `<clip>` element itself, see `minimal.asd`:

```
<asdf version="0.4">
  <clip file="audio/ukewave.ogg" pos="1 2" />
</asdf>
```

Volume control is part of the *<transform>* (page 57) mechanism. A constant volume can be specified with the `vol` attribute of `<clip>` and/or `<channel>`, a dynamic volume envelope can be applied with a `<transform>` element that's running in parallel to the `<clip>` – see *<seq> and <par>* (page 55).

As `selecting-channels.asd` shows, not all channels of a `<clip>` have to be used:

```
<asdf version="0.4">
  <par repeat="3">
    <seq>
      <wait dur="1.18" />
      <clip file="audio/marimba.ogg">
        <channel pos="-2 2" />
        <!-- NB: second channel is unused -->
      </clip>
    </seq>
    <clip file="audio/marimba.ogg">
      <channel skip="1" />
      <channel pos="2 2" />
    </clip>
  </par>
</asdf>
```

Audio clips are always played in full length. Audio files should be trimmed to the desired length during scene authoring.

#### A.3.6.1 `repeat`

`<clip>` elements can be repeated, see *Repetition* (page 65).

#### A.3.6.2 `id`

Both `<clip>` and `<channel>` elements can be the target of a <*transform*> (page 57), as long as they have an `id` attribute. `<transform>` and `<clip>` can have differing begin and end times. A single `<transform>` can apply to multiple `<clip>` and/or `<channel>` elements. A `<clip>` can be transformed by multiple `<transform>` elements over time. The `<transform>` elements can overlap, but only one of them can contain a rotation in this case (because the order of applying those rotations would be ambiguous).

#### A.3.6.3 `source`

If no `source` attribute is given, a `<source>` is created implicitly for each channel. The order of implicit sources is unspecified. An implementation may re-use an implicit source for multiple clips (as long as the clips don't overlap in time), but this is not required.

Individual audio channels can also be explicitly assigned to existing <*source*> (page 52) elements, as demonstrated in `source-transform.asd`:

```
<asdf version="0.4">
  <head>
    <source id="src-one" pos="-1 1" />
    <source id="src-two" pos="1 1" />
  </head>
  <clip file="audio/marimba.ogg">
    <channel source="src-one" />
    <channel source="src-two" />
  </clip>
  <clip file="audio/marimba.ogg">
    <channel source="src-two" />
    <channel source="src-one" />
  </clip>
</asdf>
```

This illustrates that different `<channel>` elements can be assigned to the same `<source>`. However, this only works if the channels don't overlap in time.

### A.3.7 `<transform>`

A constant transform can be simply added to a `<clip>` element, like the `pos` attribute in `minimal.asd`:

```
<asdf version="0.4">
  <clip file="audio/ukewave.ogg" pos="1 2" />
</asdf>
```

Such attributes (`pos`, `rot` etc.) can be added to *<clip> and <channel>* (page 56), as well as *<source>* (page 52) and *<reference>* (page 54).

These attributes can be seen as shorthand notation to avoid using `<transform>` elements for such simple cases. Of course, explicit `<transform>` elements can also be used, as shown in `minimal-expanded-with-explicit-transform.asd`:

```xml
<?xml version="1.0"?>
<asdf version="0.4">
  <head>
    <source id="src1" />
  </head>
  <body>
    <par>
      <clip file="audio/ukewave.ogg">
        <channel id="channel1" source="src1" />
      </clip>
      <transform apply-to="channel1" pos="1 2 0" />
    </par>
  </body>
</asdf>
```

#### A.3.7.1 `apply-to`

The required attribute `apply-to` defines the target(s) for the transform. This is a space-separated list of IDs of any *<source>* (page 52), *<clip> and <channel>* (page 56) elements, as well as other `<transform>` elements. The special ID `"reference"` can be used to target the *<reference>* (page 54).

A `<transform>` element can apply to multiple objects. An object can be the target of multiple transforms, as long as at most one of them contains a rotation.

#### A.3.7.2 `pos`

This is named after *position*, but technically, the term *translation* would be more appropriate. The final *position* of a sound source – or the *<reference>* (page 54) – can be the result of multiple *translations* (and maybe *rotations* as well, see below) applied to the default *position* `(0, 0, 0)`.

The `pos` attribute contains a space-separated list of two or three coordinate values (in meters). If only two values are given, the third one is assumed to be zero. For coordinate system conventions, see *Position and Orientation* (page 51).

#### A.3.7.3 `rot`

Unlike `pos`, this is aptly named after *rotation*. The final *orientation* of a sound source – or the *<reference>* (page 54) – can be the result of multiple *rotations*, applied to the default *orientation* `(0, 0, 0)`.

The `rot` attribute contains a space-separated list of up to three angles (in degrees) called *azimuth*, *elevation* and *roll*. Only *azimuth* is required, the others default to zero if not specified. For angle conventions, see *Position and Orientation* (page 51).

The range of angle values is not limited, but the represented rotations are cyclically repeating and the number of turns is irrelevant. This means that the angles -90 and 270 both specify the same rotation. When using a sequence of rotations to define a rotation spline (see the `<o>` element below), the smallest possible angular difference between neighboring rotations is used. For example, an angle of 270 degrees followed by an angle of 0 degrees will lead to a rotation of 90 degrees. An angle of 180 degrees followed by -180 degrees will lead to no rotation at all.

The order of applying translations and rotations matters: within a `<transform>` element, `pos` is applied *after* `rot`. This means that the target of a `<transform>` is first rotated around the (local) origin and then translated to its final position.

#### A.3.7.4 `vol`

A (linear) volume change can be specified as a non-negative decimal value. Using `vol="0"` results in silence, `vol="0.5"` corresponds to an attenuation of about 6 decibels, `vol="1"` doesn't change the volume and `vol="2"` corresponds to a boost of about 6 decibels.

#### A.3.7.5 `<o>`

A `<transform>` element can contain zero, one or more `<o>` elements. Let's call them *transform nodes*. A `<transform>` with a single `<o>` element is able to describe a constant transform. If we specify two transform nodes, we can define a *linear movement* between two points. This is shown in `two-pos.asd`:

```
<asdf version="0.4">
  <par>
    <clip id="ukulele" file="audio/ukewave.ogg" />
    <transform apply-to="ukulele">
      <o pos="-2 2" />
      <o pos="2 2" />
    </transform>
  </par>
</asdf>
```

You can also specify two rotations, which leads to a (spherical) linear interpolation between them. See `two-rot.asd`:

```
<asdf version="0.4">
  <par>
    <clip id="marimba" file="audio/marimba.ogg">
      <channel pos="-1 2" />
      <channel pos="1 2" />
    </clip>
    <transform apply-to="marimba">
      <o rot="45" />
      <o rot="-45" />
    </transform>
  </par>
</asdf>
```

In fact, two nodes are not a special case. As soon as there is more than one node, a spline is constructed that passes through all the nodes. In the case of two nodes, this leads to a linear path, but with more than two nodes, curved trajectories can be created, as for example in `minimal-spline.asd`:

```
<asdf version="0.4">
  <par>
    <clip id="ukulele" file="audio/ukewave.ogg" />
    <transform apply-to="ukulele">
      <o pos="-2 -2" />
      <o pos="-2 2" />
      <o pos="2 2" />
      <o pos="2 -2" />
    </transform>
  </par>
</asdf>
```

In addition to `pos` and `rot`, the `vol` attribute can also be animated, see `transform-vol.asd`:

```
<asdf version="0.4">
  <par>
    <clip id="ukulele" file="audio/ukewave.ogg" pos="0 1.5" />
    <transform apply-to="ukulele">
      <o vol="0" />
      <o vol="1" />
      <o vol="0" />
      <o vol="1.5" />
      <o vol="0" />
    </transform>
  </par>
</asdf>
```

**Note:** This should only be used for relatively slow volume changes, because the renderer might only apply them on a block-by-block basis. If you need fast envelopes, those should be applied by modifying the audio file in a waveform editor.

#### A.3.7.5.1 `time`

By default, sources move with a constant speed along trajectories, but if desired, time values can be assigned to any node. The speed will be varied such that the source passes those nodes at the given times. The first node always implicitly has `time="0"`. See `spline-time.asd`:

```
<asdf version="0.4">
  <par>
    <clip id="ukulele" file="audio/ukewave.ogg" />
    <transform apply-to="ukulele">
      <o pos="-2 -2" />
      <o pos="-2 2" />
      <o pos="2 2" time="5" />
      <o pos="2 -2" />
    </transform>
```

(continues on next page)

(continued from previous page)

```
  </par>
</asdf>
```

If not specified otherwise, time values are interpreted as seconds. Hours and minutes can be spelled in `HH:MM:SS.sss` format (where hours and fractions of seconds are optional) or using the `h` and `min` suffixes. For an example, see `spline-time-hh-mm-ss.asd`:

```
<asdf version="0.4">
  <par>
    <clip id="ukulele" file="audio/ukewave.ogg" />
    <transform apply-to="ukulele">
      <o pos="-2 -2" />
      <o pos="-2 2" time="0:10" />
      <o pos="2 2" time="0.5 min" />
      <o pos="2 -2" />
    </transform>
  </par>
</asdf>
```

Time values can also be given in percent, where 100% is the total duration of (one repetition of) the `<transform>`. See `spline-time-percent.asd`:

```
<asdf version="0.4">
  <par>
    <clip id="ukulele" file="audio/ukewave.ogg" />
    <transform apply-to="ukulele">
      <o pos="-2 -2" />
      <o pos="-2 2" time="10%" />
      <o pos="2 2" time="50%" />
      <o pos="2 -2" />
    </transform>
  </par>
</asdf>
```

If the `<transform>` doesn't have a `dur` attribute (see below), the last node can have an explicit `time` value, but a percentage is not allowed. If unspecified, `time="100%"` is implied, i.e. the `<transform>` always ends with the last transform node.

If the `time` value of a node is not specified, it is deduced from the surrounding nodes.

#### A.3.7.5.2 `speed`

In addition to time values, concrete speed values can also be specified. However, not all speed values are allowed. In order to provide smooth movements, the possible speed values are limited to a certain range. The speed is given in meters per second.

For an example, see `spline-speed.asd`:

```
<asdf version="0.4">
  <par>
    <clip id="ukulele" file="audio/ukewave.ogg" />
    <transform apply-to="ukulele">
      <o pos="-2 -2" speed="0" />
      <o pos="-2 2" />
```

(continues on next page)

(continued from previous page)

```
      <o pos="2 2" time="15" speed="0.5" />
      <o pos="2 -2" />
    </transform>
  </par>
</asdf>
```

#### A.3.7.5.3 `tension/continuity/bias`

The ASDF uses *Kochanek–Bartels Splines* (page 191), which means that the so-called TCB attributes `tension`, `continuity` and `bias` (each ranging from `-1.0` to `1.0` with a default of `0.0`) can be used. These attributes can be applied to individual *transform nodes* or to the whole `<transform>`, as shown in `spline-tcb.asd`:

```
<asdf version="0.4">
  <par>
    <clip file="audio/marimba.ogg">
      <channel id="left" />
      <channel id="right" />
    </clip>
    <transform apply-to="left" tension="-0.5">
      <o pos="-2 -2" />
      <o pos="-2 2" time="33%" />
      <o pos="2 2" time="66%" />
      <o pos="2 -2" />
    </transform>
    <transform apply-to="right">
      <o pos="-2 -2" />
      <o pos="-2 2" bias="-1" time="33%" />
      <o pos="2 2" bias="1" time="66%" />
      <o pos="2 -2" />
    </transform>
  </par>
</asdf>
```

In most cases, specifying TCB values will not be necessary, but they can be useful for creating straight lines, sharp edges, circles and other *Special Shapes* (page 67).

TCB attributes can also be used for `rot` trajectories, leading to *Kochanek–Bartels-like Rotation Splines* (page 259).

#### A.3.7.5.4 Mixed Transform Attributes

We have seen that `pos`, `rot` and `vol` trajectories can be created. However, they can also be combined into a single trajectory.

None of the transform attributes are required, but if one of the attributes is used in any transform node, it also has to be specified in the first and last node. In other words, missing values are interpolated but not extrapolated.

The scene `mixed-transform-attributes.asd` illustrates this in an example trajectory:

```
<asdf version="0.4">
  <par>
    <clip id="marimba" file="audio/marimba.ogg">
      <channel pos="-1 0" />
      <channel pos="1 0" />
    </clip>
    <transform apply-to="marimba">
      <o pos="0 -2" rot="-20" vol="1" />
      <o pos="0 0" time="1s" />
      <o vol="1" />
      <o rot="0" time="2s" />
      <o vol="0" />
      <o vol="1" time="65%" />
      <o pos="0 2" rot="20" vol="1"/>
    </transform>
  </par>
</asdf>
```

#### A.3.7.6 `repeat`

`<transform>` elements can be repeated, see *Repetition* (page 65).

#### A.3.7.7 `dur`

If the last transform node has its `time` attribute set, this will determine the duration of the `<transform>`. Alternatively, the duration of a `<transform>` can be specified with the `dur` attribute, which allows the same syntax as the `time` attribute of transform nodes. If there are repetitions, the duration is that of a single repetition. A percentage can be given, which is relative to the duration of (one repetition of) the parent element.

If no duration is given, and the `<transform>` is part of a `<par>` container, the duration is taken from the `<par>` container (whose duration might be provided by its first sub-element). See *<seq> and <par>* (page 55).

#### A.3.7.8 Nested `<transform>`

Any `<transform>` that has an `id` attribute can be used as the target of another `<transform>`. The transforms can have different begin and end times. They only have an effect while they are active.

Multiple `<transform>` elements can target the same object, but at most one of them can specify a rotation.

An example of nested transforms can be seen in `nested-transforms.asd`:

```
<asdf version="0.4">
  <par>
    <clip id="ukulele" file="audio/ukewave.ogg" pos="-2 -2" />
    <transform id="horizontal-movement" apply-to="ukulele" repeat="10">
      <o pos="2 4" />
      <o pos="0 2" />
```

(continues on next page)

(continued from previous page)

```
      <o pos="2 0" />
      <o pos="4 2" />
      <o pos="closed" />
    </transform>
    <transform apply-to="horizontal-movement">
      <o rot="0" />
      <o rot="0 0 90" />
      <o rot="0 0 180" />
    </transform>
  </par>
</asdf>
```

The `<clip>` defines a static position, which is then dynamically translated in the horizontal plane according to the `<transform>` named `horizontal-movement`. This horizontal movement is then transformed again, this time with a dynamic rotation around the *roll* axis.

#### A.3.7.9 Creating Groups With `<transform>`

There is no dedicated "group" element, but a `<transform>` with multiple targets in the `apply-to` attribute is essentially defining a group. All transform attributes are optional, allowing us to create a group by using a non-transforming `<transform>`:

```
<transform id="my-group" apply-to="target1 target2 my-other-target" />
```

This group can then in turn be the target of further `<transform>` elements.

### A.3.8 `<wait>`

This can be used to wait for some time, see e.g. `wait.asd`:

```
<asdf version="0.4">
  <clip file="audio/xmas.wav" pos="-1.5 1" />
  <wait dur="5" />
  <clip file="audio/ukewave.ogg" pos="1.5 1" />
</asdf>
```

#### A.3.8.1 `dur`

The wait duration can be given either as a time duration or as a percentage of the parent duration. The same syntax as in the *time* (page 60) attribute of transform nodes is supported.

## A.4 Repetition

*<clip>* (page 56), *<transform>* (page 57), *<seq> and <par>* (page 55) elements can be repeated using the `repeat` attribute. Only full repetitions (i.e. integer values) are supported.

For an example of all elements that support `repeat`, see `repeat.asd`:

```
<asdf version="0.4">
  <par repeat="5">
    <clip id="ukulele" file="audio/ukewave.ogg" repeat="2" />
    <seq repeat="3">
      <transform apply-to="ukulele" dur="20%">
        <o pos="0 2" />
        <o pos="2 0" />
        <o pos="0 -2" />
        <o pos="-2 0" />
        <o pos="closed" />
      </transform>
      <transform apply-to="ukulele" repeat="4">
        <o pos="0 2" />
        <o pos="3 2" />
        <o pos="-3 2" />
        <o pos="closed" />
      </transform>
    </seq>
  </par>
</asdf>
```

It's not possible to repeat an element forever, but you might as well just use a huge number of repetitions, as shown in `repeat-nearly-indefinitely.asd`:

```
<asdf version="0.4">
  <par repeat="999999">
    <clip id="ukulele" file="audio/ukewave.ogg" />
    <transform apply-to="ukulele" repeat="4">
      <o pos="0 2" />
      <o pos="2 0" />
      <o pos="0 -2" />
      <o pos="-2 0" />
      <o pos="closed" />
    </transform>
  </par>
</asdf>
```

## A.5 ASDF Splines

Knowing the details about the splines used in the ASDF is not necessary to create scenes. However, it might still be interesting to know why the shape and behavior of trajectories is the way it is.

A reference implementation of ASDF splines is available at `https://github.com/AudioSceneDescriptionFormat/asdfspline-rust`. This library is implemented in Rust[6] and it provides language bindings for C[7] and Python[8].

We refer to a *general definition of splines and their properties* (page 101) and to detailed background information about all the different types of *Euclidean splines* (page 88) and *rotation splines* (page 232) mentioned here, including their mathematical derivation and their individual properties.

## A.5.1 Position Splines

The most obvious type of splines in the ASDF are *position splines*. The idea is that a scene author provides a sequence of positions in three-dimensional space and an ASDF library creates a smooth curve that goes through all of them. The scene author can also provide the times at which the positions should be reached, as well as – with certain limitations – the speed at those positions.

The ASDF uses (cubic) *Kochanek–Bartels Splines* (page 191), which provide three parameters per control point: *tension*, *continuity* and *bias*, which can be abbreviated to *TCB*. These TCB parameters allow changing the shape of the resulting curve without changing the original sequence of positions. The possible values range from -1 to 1, with 0 being the default. Kochanek–Bartels splines are a superset of the probably more familiar *Catmull–Rom Splines* (page 155). If all TCB values are zero, the two splines are identical.

To be guaranteed to avoid cusps and self-intersections (assuming default TCB values), *Centripetal Parameterization* (page 163) is used. This, however, means that the parameter values cannot be chosen freely anymore. Since we want to be able to specify the times when certain control points are reached (and to some degree the speed along the trajectory), we cannot directly interpret the parameter value as elapsed time. As a first step, we re-parameterize the spline to have constant speed, which is also known as *Arc-Length Parameterization* (page 229).

Having constant speed trajectories is useful, but only being able to use constant speed is also quite limiting. Therefore, on top of arc-length parameterization, ASDF splines are also *re-parameterized with a monotone spline* (page 231). This means that for each position in the spline, we can specify the time when this position should be reached. We can even specify the speed at these positions (as long as the monotonicity of the re-parameterization spline can be maintained). See the section about *<transform>* (page 57) for details.

It might have been tempting to use *Bézier Splines* (page 134) due to their widespread use in 2D drawing software. However, finding appropriate *drag points* in three-dimensional space is very hard compared to simply defining a sequence of 3D positions. Similarly, it would be quite cumbersome to explicitly define three-dimensional tangent vectors for use with *Hermite Splines* (page 105).

[6] `https://www.rust-lang.org/`
[7] `https://www.open-std.org/jtc1/sc22/wg14/`
[8] `https://www.python.org/`

### A.5.2 Rotation Splines

When a scene author provides a sequence of orientations for sound sources or groups of sound sources, the values between the given orientations will be smoothly interpolated.

The same kind of splines are used as for positions, just modified to work with rotations. Centripetal *Kochanek–Bartels-like Rotation Splines* (page 259) are used, which are a superset of *Catmull–Rom-Like Rotation Splines* (page 256). If specified, the same TCB values apply to both position and rotation splines. The rotation splines are arc-length parameterized by default, which means that they have a constant angular speed. Time instances can be specified for any of the given rotations, which in turn control the changing angular speeds along the spline. The angular speed cannot be specified explicitly, though. This would be technically possible, but it is currently not implemented because specifying an angular speed (for example in degrees per second) seems unintuitive. However, this might be added in a future ASDF version.

### A.5.3 Volume Splines

The volume of the *<reference>* (page 54), of *<source>* (page 52) elements and of groups of sources can be changed over time. Since volume can be applied just as translation and rotation, it is part of the *<transform>* (page 57) attributes, which can be applied to anything that has an `id` attribute.

Volume values should change smoothly, so they are controlled with splines as well. An important property of those splines is that they must not produce interpolated values that overshoot the given local maximum values, nor should they produce negative values. This can be ensured by using *Piecewise Monotone Interpolation* (page 214).

## A.6 Special Shapes

There are no pre-defined special shapes in the ASDF. All trajectories use the same underlying type of spline – see *ASDF Splines* (page 65).

### A.6.1 Square

Trajectories in the ASDF are smooth curves by default, and a little extra effort is required to create movements with sharp corners. There are two simple settings to get straight line segments: `tension="1"` or `continuity="-1"`. Both options are shown in `square.asd`:

```
<asdf version="0.4">
  <par>
    <clip file="audio/marimba.ogg">
      <channel id="one" />
      <channel id="two" />
    </clip>
    <transform apply-to="one" tension="1">
      <o pos="0 2" />
      <o pos="-2 0" />
```

(continues on next page)

(continued from previous page)

```
      <o pos="0 -2" />
      <o pos="2 0" />
      <o pos="closed" />
    </transform>
    <transform apply-to="two" continuity="-1">
      <o pos="0 2" />
      <o pos="2 0" />
      <o pos="0 -2" />
      <o pos="-2 0" />
      <o pos="closed" />
    </transform>
  </par>
</asdf>
```

### A.6.2 Circle

Non-rational cubic polynomial curves – which is the type of curve the ASDF uses for position trajectories – cannot exactly describe circles. But this is no problem, because circles can be approximated very closely. This can be done by providing the corner points of a square and using a `tension` value of about `-0.66`. However, there is actually a way to create exact circles: by applying a rotation spline to a translated object. The example scene `circle.asd` shows both approaches:

```
<asdf version="0.4">
  <par>
    <clip file="audio/marimba.ogg">
      <channel id="one" pos="0 2" />
      <channel id="two" />
    </clip>
    <!-- this is a perfect circle: -->
    <transform apply-to="one">
      <o rot="-10" />
      <o rot="-100" />
      <o rot="-190" />
      <o rot="-280" />
      <o rot="closed" />
    </transform>
    <!-- this is extremely close to a circle: -->
    <transform apply-to="two" tension="-0.66">
      <o pos="0 2" />
      <o pos="2 0" />
      <o pos="0 -2" />
      <o pos="-2 0" />
      <o pos="closed" />
    </transform>
  </par>
</asdf>
```

In this example, the center of rotation is the origin. If the center of rotation is supposed to be somewhere else, it can be moved by applying a new `<transform>` element with the desired `pos` attribute to the `<transform>` that does the rotation.

### A.6.3 Helix

A helical movement can be created by combining a (repeated) circular movement (using one of the methods shown above) with a linear movement perpendicular to the plane of the circle. This is shown in `helix.asd`:

```
<asdf version="0.4">
  <par>
    <clip id="ukulele" file="audio/ukewave.ogg" pos="-2 0" />
    <transform id="circular-motion" apply-to="ukulele" repeat="10">
      <o rot="0 0 0" />
      <o rot="0 0 90" />
      <o rot="0 0 180" />
      <o rot="0 0 -90" />
      <o rot="closed" />
    </transform>
    <transform id="forward-motion" apply-to="circular-motion">
      <o pos="0 -2" />
      <o pos="0 2" />
    </transform>
  </par>
</asdf>
```

In this example, the `<clip>` is offset to the left and a rotation spline rotates this offset multiple times around the *roll* axis. This circular motion is then translated along the default view direction. In this case, it doesn't matter if `forward-motion` is applied to `circular-motion` or directly to `ukulele`.

### A.6.4 Sinusoidal Oscillation

Sine waves are not directly supported by the ASDF, but they can be approximated to some degree. By setting `speed="0"` at the desired maxima and minima, something similar to sine and cosine oscillations can be created. This is illustrated in `sine-wave.asd`:

```
<asdf version="0.4">
  <par>
    <clip file="audio/marimba.ogg">
      <channel id="one" />
      <channel id="two" pos="0 2" />
    </clip>
    <transform id="left-right-motion" apply-to="one two" repeat="2">
      <o pos="0 0" />
      <o pos="2 0" speed="0" time="25%" />
      <o pos="-2 0" speed="0" time="75%" />
      <o pos="closed" />
    </transform>
    <transform id="forward-backward-motion" apply-to="one" repeat="2">
      <o pos="0 2" speed="0" />
      <o pos="0 -2" speed="0" time="50%" />
      <o pos="closed" />
    </transform>
  </par>
</asdf>
```

### A.6.5 Lissajous Figures

Once we have sinusoidal oscillations (or at least something similar), we can make Lissajous figures[9], as shown in `lissajous.asd`:

```
<asdf version="0.4">
  <par repeat="2">
    <clip id="ukulele" file="audio/ukewave.ogg" vol="0.3" />
    <par repeat="3">
      <transform id="left-right" apply-to="ukulele">
        <o pos="-2 0" speed="0" />
        <o pos="2 0" time="50%" speed="0" />
        <o pos="closed" />
      </transform>
      <seq repeat="3">
        <transform id="front-back" apply-to="ukulele">
          <o pos="0 0" />
          <o pos="0 2" time="25%" speed="0" />
          <o pos="0 -2" time="75%" speed="0" />
          <o pos="closed" />
        </transform>
      </seq>
    </par>
  </par>
</asdf>
```

## A.7 Implementation Notes

The information in this section is not needed in order to create audio scenes with the ASDF.

When implementing an ASDF library, it is recommended to convert all rotation angles as soon as possible into rotation matrices or quaternions, as the following sections show.

The following section was generated from `doc/rotation-matrices.ipynb` .....................................

### A.7.1 Converting ASDF Rotations to Rotation Matrices

To rotate objects in an ASDF scene, you can use *azimuth, elevation and roll angles* (page 58), for example like this:

```
<... rot="-30 12.5 5">
```

The used coordinate system conventions are shown in the *section about position and orientation* (page 51).

In this section we show how these angles can be converted to rotation matrices[10], in order to practically use those rotations in software.

There isn't just a single way to choose rotation angles in 3D space, in fact, there are very many ways to do this, many of them leading to different rotation matrices.

[9] https://en.wikipedia.org/wiki/Lissajous_curve
[10] https://en.wikipedia.org/wiki/Rotation_matrix

Here's a (hopefully somewhat complete) overview about the possible options and the choices taken by the ASDF:

- Right-handed vs. left-handed coordinate system[11]: The ASDF uses a right-handed one.
- Direction of the axes: The ASDF uses the ENU (east, north, up) convention.
- Euler angles vs. Tait–Bryan angles[12]: The ASDF uses a variation of Tait–Bryan.
- There are many possible conventions[13] for the order of angles and which axes they rotate around: The ASDF conventions are shown in detail below.
- "intrinsic"[14] vs. "extrinsic"[15] = "local" vs. "global" reference system: This sounds complicated, but it's really just about the order of transformations. See below for details.
- Rotating vectors (= "active" = "alibi") vs. rotating the coordinate system (= "passive" = "alias")[16]: In the following derivations we consider the *active* situation, but a similar derivation can be done for the *passive* case.

  In case you are wondering: the functions sympy.matrices.rot_axis1()[17] etc. do the latter, therefore we cannot use them here (at least not without some further manipulations).
- Rotation matrices can be derived for pre-multiplication with column vectors vs. post-multiplication with row vectors[18]: We are using column vectors here, but (different) matrices could be derived for use with row vectors.

Let's get started then, shall we?

First we import SymPy[19], which is great for doing this kind of symbolic derivations:

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

We have to define our three input angles. These are often called *azimuth/elevation/roll*, or *yaw/pitch/roll*, or *heading/elevation/bank*.

Here we just use the greek letters $\alpha$, $\beta$ and $\gamma$:

[2]:
```
alpha, beta, gamma = sp.symbols('alpha beta gamma')
```

The ASDF uses an ENU (east, north, up) coordinate system and the reference ("forward") direction is *north*, i.e. along the positive y-axis.

[3]:
```
alpha
```

[3]: $\alpha$

The *azimuth* angle $\alpha$ is:

[11] https://en.wikipedia.org/wiki/Coordinate_system
[12] https://en.wikipedia.org/wiki/Euler_angles
[13] https://en.wikipedia.org/wiki/Axes_conventions
[14] https://en.wikipedia.org/wiki/Euler_angles#Conventions_by_intrinsic_rotations
[15] https://en.wikipedia.org/wiki/Euler_angles#Conventions_by_extrinsic_rotations
[16] https://en.wikipedia.org/wiki/Active_and_passive_transformation
[17] https://docs.sympy.org/latest/modules/matrices/matrices.html#sympy.matrices.dense.rot_axis1
[18] https://en.wikipedia.org/wiki/Rotation_matrix#Ambiguities
[19] https://www.sympy.org/

- zero when pointing north (i.e. along the positive y-axis),
- rotating around the z-axis (which points up)
- positive when rotating towards west (right hand rule[20]).

```
[4]: beta
```

```
[4]: β
```

The *elevation* angle $\beta$ is:

- zero in the horizontal plane,
- rotating around the *local* x-axis
- positive when the nose goes up (right hand rule).

```
[5]: gamma
```

```
[5]: γ
```

The *roll* angle $\gamma$ is:

- zero when the *top* of the object points to the zenith (which is just the normal "upright" orientation),
- rotating around the local y-axis
- positive when the object is leaning towards starboard[21] (right hand rule).

The definitions above use the *intrinsic* way of describing the rotations (i.e. relative to *local* coordinate axes).

If you want to use the *extrinsic* way, you can use the same angles. You just have to choose the right order of *global* rotations: First *roll*, then *elevation*, then *azimuth*. We will be using the *extrinsic* style below.

Let's also define the cartesian components of a vector $a$:

```
[6]: a_x, a_y, a_z = sp.symbols('a_x:z')
```

We will need those only during the derivation, they will not appear in the final equations.

#### A.7.1.1 Azimuth: Rotation around the z-Axis

Writing the vector $a$ in cylindrical coordinates $r_z$ (radius), $\phi_z$ (angle) and $a_z$ (height):

```
[7]: r_z, phi_z = sp.symbols('r_z phi_z')
```

... we can get its cartesian coordinates like this:

[20] https://en.wikipedia.org/wiki/Right-hand_rule
[21] https://en.wikipedia.org/wiki/Port_and_starboard

[8]:
```
a = sp.Matrix([
    r_z * sp.cos(phi_z),
    r_z * sp.sin(phi_z),
    a_z,
])
a
```

[8]:
$$\begin{bmatrix} r_z \cos\left(\phi_z\right) \\ r_z \sin\left(\phi_z\right) \\ a_z \end{bmatrix}$$

We are using column vectors here, that means we are searching for a rotation matrix to left-multiply this vector in order to get the vector $b$.

To get a representation of the vector $b$, let's rotate $a$ by an azimuth angle $\alpha$:

[9]:
```
b = sp.Matrix([
    r_z * sp.cos(phi_z + alpha),
    r_z * sp.sin(phi_z + alpha),
    a_z,
])
b
```

[9]:
$$\begin{bmatrix} r_z \cos\left(\alpha + \phi_z\right) \\ r_z \sin\left(\alpha + \phi_z\right) \\ a_z \end{bmatrix}$$

Note that $a_z$ is not affected by the rotation.

We can use some trigonometric identities to expand this:

[10]:
```
b = b.expand(trig=True)
b
```

[10]:
$$\begin{bmatrix} -r_z \sin\left(\alpha\right) \sin\left(\phi_z\right) + r_z \cos\left(\alpha\right) \cos\left(\phi_z\right) \\ r_z \sin\left(\alpha\right) \cos\left(\phi_z\right) + r_z \sin\left(\phi_z\right) \cos\left(\alpha\right) \\ a_z \end{bmatrix}$$

... and re-write it using the (cartesian) coordinates of vector $a$: $a_x$, $a_y$ and $a_z$:

[11]:
```
b = b.subs(list(zip(a, [a_x, a_y, a_z])))
b
```

[11]:
$$\begin{bmatrix} a_x \cos\left(\alpha\right) - a_y \sin\left(\alpha\right) \\ a_x \sin\left(\alpha\right) + a_y \cos\left(\alpha\right) \\ a_z \end{bmatrix}$$

Remember, we are looking for a rotation matrix that, when $a$ is left-multiplied by it, yields $b$.

In other words (or rather symbols):

$$\begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = R_z(\alpha) \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}$$

Given the components of $b$ shown above, we can simply pick out the matrix elements.

Or we let SymPy do it:

[12]:
```
Rz = sp.Matrix([[line.coeff(var) for var in [a_x, a_y, a_z]]
                for line in b])
Rz
```

[12]:
$$\begin{bmatrix}\cos(\alpha) & -\sin(\alpha) & 0\\ \sin(\alpha) & \cos(\alpha) & 0\\ 0 & 0 & 1\end{bmatrix}$$

That's it!

Let's do a little sanity check, rotating the y unit vector (i.e. "looking straight ahead") by 90 degrees to the left:

[13]:
```
Rz.subs(alpha, sp.pi / 2) * sp.Matrix([0, 1, 0])
```

[13]:
$$\begin{bmatrix}-1\\ 0\\ 0\end{bmatrix}$$

This yields the negative x unit vector, which points westwards. That sounds about right!

#### A.7.1.2 Elevation: Rotation around the (local) x-Axis

Now the same thing, just using a different vector $a$.

[14]:
```
r_x, phi_x = sp.symbols('r_x phi_x')
a = sp.Matrix([
    a_x,
    r_x * sp.cos(phi_x),
    r_x * sp.sin(phi_x),
])
a
```

[14]:
$$\begin{bmatrix}a_x\\ r_x\cos(\phi_x)\\ r_x\sin(\phi_x)\end{bmatrix}$$

Let's rotate $a$ by the elevation angle $\beta$ to get a vector $b$:

[15]:
```
b = sp.Matrix([
    a_x,
    r_x * sp.cos(phi_x + beta),
    r_x * sp.sin(phi_x + beta),
])
b
```

[15]:
$$\begin{bmatrix}a_x\\ r_x\cos(\beta+\phi_x)\\ r_x\sin(\beta+\phi_x)\end{bmatrix}$$

Again, expand using trig identities and substitute $a$ back in:

```
[16]: b = b.expand(trig=True).subs(list(zip(a, [a_x, a_y, a_z])))
      b
```

[16]:
$$\begin{bmatrix} a_x \\ a_y \cos\left(\beta\right) - a_z \sin\left(\beta\right) \\ a_y \sin\left(\beta\right) + a_z \cos\left(\beta\right) \end{bmatrix}$$

... and obtain a matrix $R_x(\beta)$ that transforms $a$ into $b$:

```
[17]: Rx = sp.Matrix([[line.coeff(var) for var in [a_x, a_y, a_z]]
                      for line in b])
      Rx
```

[17]:
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\left(\beta\right) & -\sin\left(\beta\right) \\ 0 & \sin\left(\beta\right) & \cos\left(\beta\right) \end{bmatrix}$$

And again a sanity check, this time using an elevation of 90 degrees:

```
[18]: Rx.subs(beta, sp.pi / 2) * sp.Matrix([0, 1, 0])
```

[18]:
$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The result is a vector pointing up, which is what we expected, didn't we?

#### A.7.1.3 Roll: Rotation around the (local) y-Axis

Doing very similar steps as before:

```
[19]: r_y, phi_y = sp.symbols('r_y phi_y')
      a = sp.Matrix([
          r_y * sp.sin(phi_y),
          a_y,
          r_y * sp.cos(phi_y),
      ])
      a
```

[19]:
$$\begin{bmatrix} r_y \sin\left(\phi_y\right) \\ a_y \\ r_y \cos\left(\phi_y\right) \end{bmatrix}$$

```
[20]: b = sp.Matrix([
          r_y * sp.sin(phi_y + gamma),
          a_y,
          r_y * sp.cos(phi_y + gamma),
      ])
      b
```

[20]:
$$\begin{bmatrix} r_y \sin\left(\gamma + \phi_y\right) \\ a_y \\ r_y \cos\left(\gamma + \phi_y\right) \end{bmatrix}$$

[21]:
```
b = b.expand(trig=True).subs(list(zip(a, [a_x, a_y, a_z])))
b
```

[21]:
$$\left[\begin{matrix}a_{x} \cos{\left(\gamma \right)} + a_{z} \sin{\left(\gamma \right)}\\a_{y}\\- a_{x} \sin{\left(\gamma \right)} + a_{z} \cos{\left(\gamma \right)}\end{matrix}\right]$$

[22]:
```
Ry = sp.Matrix([[line.coeff(var) for var in [a_x, a_y, a_z]]
                for line in b])
Ry
```

[22]:
$$\left[\begin{matrix}\cos{\left(\gamma \right)} & 0 & \sin{\left(\gamma \right)}\\0 & 1 & 0\\- \sin{\left(\gamma \right)} & 0 & \cos{\left(\gamma \right)}\end{matrix}\right]$$

Sanity check: Applying a *roll* angle of 90 degrees to a vector pointing up ...

[23]:
```
Ry.subs(gamma, sp.pi / 2) * sp.Matrix([0, 0, 1])
```

[23]:
$$\left[\begin{matrix}1\\0\\0\end{matrix}\right]$$

... leads to a vector pointing east. This is what we wanted.

### A.7.1.4 Combining all Axes

As mentioned above, we have to choose the right sequence of (global) rotations: first *roll*, then *elevation*, then *azimuth*.

Note that we start with $R_y$ (roll) *on the right*, and then left-apply $R_x$ (elevation) and then left-apply $R_z$ (azimuth).

You should read this from right to left:

[24]:
```
R = Rz * Rx * Ry
R
```

[24]:
$$\left[\begin{matrix}- \sin{\left(\alpha \right)} \sin{\left(\beta \right)} \sin{\left(\gamma \right)} + \cos{\left(\alpha \right)} \cos{\left(\gamma \right)} & - \sin{\left(\alpha \right)} \cos{\left(\beta \right)} & \sin{\left(\alpha \right)} \sin{\left(\beta \right)} \cos{\left(\gamma \right)} + \sin{\left(\gamma \right)} \cos{\left(\alpha \right)}\\\sin{\left(\alpha \right)} \cos{\left(\gamma \right)} + \sin{\left(\beta \right)} \sin{\left(\gamma \right)} \cos{\left(\alpha \right)} & \cos{\left(\alpha \right)} \cos{\left(\beta \right)} & \sin{\left(\alpha \right)} \sin{\left(\gamma \right)} - \sin{\left(\beta \right)} \cos{\left(\alpha \right)} \cos{\left(\gamma \right)}\\- \sin{\left(\gamma \right)} \cos{\left(\beta \right)} & \sin{\left(\beta \right)} & \cos{\left(\beta \right)} \cos{\left(\gamma \right)}\end{matrix}\right]$$

That's it, that's our rotation matrix!

Copy this to use it with SymPy (you'll have to import `Matrix`, `sin` and `cos` and define `alpha`, `beta` and `gamma`):

[25]:
```
print(R)
```

```
Matrix([[-sin(alpha)*sin(beta)*sin(gamma) + cos(alpha)*cos(gamma), -
↪sin(alpha)*cos(beta), sin(alpha)*sin(beta)*cos(gamma) +␣
↪sin(gamma)*cos(alpha)], [sin(alpha)*cos(gamma) +␣
↪sin(beta)*sin(gamma)*cos(alpha), cos(alpha)*cos(beta),␣
```

(continues on next page)

(continued from previous page)

```
↪sin(alpha)*sin(gamma) - sin(beta)*cos(alpha)*cos(gamma)], [-
↪sin(gamma)*cos(beta), sin(beta), cos(beta)*cos(gamma)]])
```

If you want to use it with NumPy, you can copy this (you'll have to import `numpy` and define `alpha`, `beta` and `gamma`):

[26]:
```
from sympy.printing.numpy import NumPyPrinter
print(NumPyPrinter().doprint(R))
```

```
numpy.array([[-numpy.sin(alpha)*numpy.sin(beta)*numpy.sin(gamma) + numpy.
↪cos(alpha)*numpy.cos(gamma), -numpy.sin(alpha)*numpy.cos(beta), numpy.
↪sin(alpha)*numpy.sin(beta)*numpy.cos(gamma) + numpy.sin(gamma)*numpy.
↪cos(alpha)], [numpy.sin(alpha)*numpy.cos(gamma) + numpy.sin(beta)*numpy.
↪sin(gamma)*numpy.cos(alpha), numpy.cos(alpha)*numpy.cos(beta), numpy.
↪sin(alpha)*numpy.sin(gamma) - numpy.sin(beta)*numpy.cos(alpha)*numpy.
↪cos(gamma)], [-numpy.sin(gamma)*numpy.cos(beta), numpy.sin(beta), numpy.
↪cos(beta)*numpy.cos(gamma)]])
```

### A.7.1.5 Rotation Matrix to Angles

You may ask: how can we get back from the rotation matrix to our angles?

If you look at the matrix $R$ above, you see that one component only depends on one variable. Namely, the component in the last row, middle column:

[27]:
```
R[2, 1]
```

[27]: $\sin(\beta)$

Therefore, we can get the value of $\beta$ simply by taking the arc-sine of this matrix element. In a numeric calculation, this would probably look something like:

```
beta = asin(R[2, 1])
```

**Note:**

The argument of the `asin()` function has to be in the domain `[-1.0; 1.0]` (see `https://en.cppreference.com/w/c/numeric/math/asin`).

Due to rounding errors, the value might be slightly outside this range, which would lead to a return value of `NaN`.

Make sure to handle this case, e.g. by re-normalizing the rotation matrix.

The rest of the matrix components depend on more than one variable, but there are a few elements that depend only on two variables.

If we divide the top middle component (multiplied by $-1$) by the one below:

[28]:
```
-R[0, 1] / R[1, 1]
```

[28]:

$$\frac{\sin(\alpha)}{\cos(\alpha)}$$

... we get an expression that only depends on $\alpha$.

We can simplify this expression:

[29]:
```
_.simplify()
```

[29]: $\tan(\alpha)$

Therefore, to get the angle $\alpha$, we only have to calculate $\frac{-R_{0,1}}{R_{1,1}}$ and take the arc-tangent of the result.

To get the appropriate quadrant of the result, we will use the function atan2()[22] in numeric calculations:

```
alpha = atan2(-R[0, 1], R[1, 1])
```

We can do a similar thing to get $\gamma$:

[30]:
```
-R[2, 0] / R[2, 2]
```

[30]:

$$\frac{\sin(\gamma)}{\cos(\gamma)}$$

[31]:
```
_.simplify()
```

[31]: $\tan(\gamma)$

Similar to the above, we take the arc-tangent of $\frac{-R_{2,0}}{R_{2,2}}$ to get the angle $\gamma$.

```
gamma = atan2(-R[2, 0], R[2, 2])
```

#### A.7.1.5.1 Gimbal Lock

But wait a second, we might have a problem: the dreaded gimbal lock[23]!

Let's consider the case where $\beta = 90$ degrees:

[32]:
```
R1 = R.subs(beta, sp.pi/2)
R1
```

[32]:

$$\begin{bmatrix} -\sin(\alpha)\sin(\gamma) + \cos(\alpha)\cos(\gamma) & 0 & \sin(\alpha)\cos(\gamma) + \sin(\gamma)\cos(\alpha) \\ \sin(\alpha)\cos(\gamma) + \sin(\gamma)\cos(\alpha) & 0 & \sin(\alpha)\sin(\gamma) - \cos(\alpha)\cos(\gamma) \\ 0 & 1 & 0 \end{bmatrix}$$

If we try to calculate $\alpha$ and $\gamma$ like above, we end up calculating

```
atan2(0, 0)
```

Sadly, that is not defined:

[22] https://en.wikipedia.org/wiki/Atan2
[23] https://en.wikipedia.org/wiki/Gimbal_lock

[33]:
```
sp.atan2(0, 0)
```

[33]: NaN

**Note:**

If the implementation supports IEEE floating-point arithmetic (IEC 60559), no `NaN` is returned (except if one of the inputs is `NaN`), see `https://en.cppreference.com/w/c/numeric/math/atan2`.

In this case, `atan2()` will return $\pm 0$ or $\pm\pi$ (which is generally not correct).

Depending on your use case, however, this might be good enough. If not, keep reading below!

We can try to find alternative equations for $\alpha$ and $\gamma$ from the hitherto unused matrix elements (but let's simplify the matrix first):

[34]:
```
R1 = sp.trigsimp(R1)
R1
```

[34]:
$$\begin{bmatrix}\cos\left(\alpha+\gamma\right) & 0 & \sin\left(\alpha+\gamma\right)\\ \sin\left(\alpha+\gamma\right) & 0 & -\cos\left(\alpha+\gamma\right)\\ 0 & 1 & 0\end{bmatrix}$$

[35]:
```
sp.simplify(R1[1, 0] / R1[0, 0])
```

[35]: $\tan\left(\alpha+\gamma\right)$

[36]:
```
sp.simplify(R1[0, 2] / -R1[1, 2])
```

[36]: $\tan\left(\alpha+\gamma\right)$

There is no unique solution to these equations. You can freely choose either $\alpha$ or $\gamma$ and use that to calculate the other angle.

A very similar thing happens for $\beta = -90$ degrees:

[37]:
```
R2 = R.subs(beta, -sp.pi/2)
R2
```

[37]:
$$\begin{bmatrix}\sin(\alpha)\sin\left(\gamma\right)+\cos(\alpha)\cos\left(\gamma\right) & 0 & -\sin(\alpha)\cos\left(\gamma\right)+\sin\left(\gamma\right)\cos(\alpha)\\ \sin(\alpha)\cos\left(\gamma\right)-\sin\left(\gamma\right)\cos(\alpha) & 0 & \sin(\alpha)\sin\left(\gamma\right)+\cos(\alpha)\cos\left(\gamma\right)\\ 0 & -1 & 0\end{bmatrix}$$

[38]:
```
R2 = sp.trigsimp(R2)
R2
```

[38]:
$$\begin{bmatrix}\cos\left(\alpha-\gamma\right) & 0 & -\sin\left(\alpha-\gamma\right)\\ \sin\left(\alpha-\gamma\right) & 0 & \cos\left(\alpha-\gamma\right)\\ 0 & -1 & 0\end{bmatrix}$$

[39]:
```
sp.simplify(R2[1, 0] / R2[0, 0])
```

[39]: $\tan(\alpha - \gamma)$

[40]:
```
sp.simplify(-R2[0, 2] / R2[1, 2])
```

[40]: $\tan(\alpha - \gamma)$

Again, there is no unique solution. You can freely choose one of the angles and then calculate the other one.

The easiest way to avoid this whole *gimbal lock* problem, is simply to never convert rotation matrices to angles.

............................................................ `doc/rotation-matrices.ipynb` ends here.

The following section was generated from `doc/quaternions.ipynb` ............................................

### A.7.2 Converting ASDF Rotations to Quaternions

This notebook shows the same thing as the *notebook about rotation matrices* (page 70), just using quaternions instead of rotation matrices. For more detailed explanations, have a look over there.

You might be tempted to use the equations from Wikipedia[24], but those use different conventions for axes and angles! The resulting equations will have a similar structure but will not be quite identical.

With the code below, any convention can be calculated by adapting

- the pairing of angles with their corresponding axes
- the sign of angles (or direction of axes) according to handedness
- the order of combining the individual axis/angle quaternions

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

[2]:
```
from sympy.algebras import Quaternion
```

[3]:
```
alpha, beta, gamma = sp.symbols('alpha beta gamma')
```

#### A.7.2.1 Azimuth: Rotation around the z-Axis

[4]:
```
q_z = Quaternion.from_axis_angle((0, 0, 1), alpha)
q_z
```

[4]: $\cos\left(\frac{\alpha}{2}\right) + 0i + 0j + \sin\left(\frac{\alpha}{2}\right)k$

Example: Rotating the y unit vector (i.e. "looking north") by 90 degrees to the left:

[24] `https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles#Euler_angles_(in_3-2-1_sequence)_to_quaternion_conversion`

[5]:
```
Quaternion.rotate_point((0, 1, 0), q_z.subs(alpha, sp.pi / 2))
```
[5]:
```
(-1, 0, 0)
```

As expected, this yields the negative x unit vector, which points westwards.

#### A.7.2.2 Elevation: Rotation around the (local) x-Axis

[6]:
```
q_x = Quaternion.from_axis_angle((1, 0, 0), beta)
q_x
```
[6]:

$$\cos\left(\frac{\beta}{2}\right) + \sin\left(\frac{\beta}{2}\right)i + 0j + 0k$$

Example: Applying 90 degrees of elevation to the y unit vector:

[7]:
```
Quaternion.rotate_point((0, 1, 0), q_x.subs(beta, sp.pi / 2))
```
[7]:
```
(0, 0, 1)
```

As expected, this yields a vector pointing up.

#### A.7.2.3 Roll: Rotation around the (local) y-Axis

[8]:
```
q_y = Quaternion.from_axis_angle((0, 1, 0), gamma)
q_y
```
[8]:

$$\cos\left(\frac{\gamma}{2}\right) + 0i + \sin\left(\frac{\gamma}{2}\right)j + 0k$$

Example: Applying a roll angle of 90 degrees to a vector pointing up:

[9]:
```
Quaternion.rotate_point((0, 0, 1), q_y.subs(gamma, sp.pi / 2))
```
[9]:
```
(1, 0, 0)
```

As expected, this yields a vector pointing east.

#### A.7.2.4 Combining all Axes

This is easy, we only have to make sure to use the right order. As with rotation matrices, you should read this from right to left (first *roll*, then *elevation*, then *azimuth*):

[10]:
```
q = q_z * q_x * q_y
q
```
[10]:

$$\left(-\sin\left(\frac{\alpha}{2}\right)\sin\left(\frac{\beta}{2}\right)\sin\left(\frac{\gamma}{2}\right) + \cos\left(\frac{\alpha}{2}\right)\cos\left(\frac{\beta}{2}\right)\cos\left(\frac{\gamma}{2}\right)\right) +$$
$$\left(-\sin\left(\frac{\alpha}{2}\right)\sin\left(\frac{\gamma}{2}\right)\cos\left(\frac{\beta}{2}\right) + \sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\alpha}{2}\right)\cos\left(\frac{\gamma}{2}\right)\right)i +$$

(continues on next page)

(continued from previous page)

$$\left(\sin\left(\frac{\alpha}{2}\right)\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\gamma}{2}\right)+\sin\left(\frac{\gamma}{2}\right)\cos\left(\frac{\alpha}{2}\right)\cos\left(\frac{\beta}{2}\right)\right)j+$$
$$\left(\sin\left(\frac{\alpha}{2}\right)\cos\left(\frac{\beta}{2}\right)\cos\left(\frac{\gamma}{2}\right)+\sin\left(\frac{\beta}{2}\right)\sin\left(\frac{\gamma}{2}\right)\cos\left(\frac{\alpha}{2}\right)\right)k$$

If you want to copy-paste this:

[11]:
```
print(q)
```

```
(-sin(alpha/2)*sin(beta/2)*sin(gamma/2) + cos(alpha/2)*cos(beta/
↪2)*cos(gamma/2)) + (-sin(alpha/2)*sin(gamma/2)*cos(beta/2) + sin(beta/
↪2)*cos(alpha/2)*cos(gamma/2))*i + (sin(alpha/2)*sin(beta/2)*cos(gamma/
↪2) + sin(gamma/2)*cos(alpha/2)*cos(beta/2))*j + (sin(alpha/2)*cos(beta/
↪2)*cos(gamma/2) + sin(beta/2)*sin(gamma/2)*cos(alpha/2))*k
```

But you should probably pre-calculate the used terms in order to avoid repeated evaluation of the same functions. You could try something like this, for example:

[12]:
```
q.subs([
    (sp.sin(alpha/2), sp.symbols('s_alpha')),
    (sp.sin(beta/2), sp.symbols('s_beta')),
    (sp.sin(gamma/2), sp.symbols('s_gamma')),
    (sp.cos(alpha/2), sp.symbols('c_alpha')),
    (sp.cos(beta/2), sp.symbols('c_beta')),
    (sp.cos(gamma/2), sp.symbols('c_gamma')),
])
```

[12]: $\left(c_\alpha c_\beta c_\gamma - s_\alpha s_\beta s_\gamma\right) + \left(c_\alpha c_\gamma s_\beta - c_\beta s_\alpha s_\gamma\right)i + \left(c_\alpha c_\beta s_\gamma + c_\gamma s_\alpha s_\beta\right)j + \left(c_\alpha s_\beta s_\gamma + c_\beta c_\gamma s_\alpha\right)k$

[13]:
```
print(_)
```

```
(c_alpha*c_beta*c_gamma - s_alpha*s_beta*s_gamma) + (c_alpha*c_gamma*s_
↪beta - c_beta*s_alpha*s_gamma)*i + (c_alpha*c_beta*s_gamma + c_gamma*s_
↪alpha*s_beta)*j + (c_alpha*s_beta*s_gamma + c_beta*c_gamma*s_alpha)*k
```

### A.7.2.5 Quaternion to Rotation Matrix

Just to make sure the result is the same as in the *notebook about rotation matrices* (page 76), let's calculate the rotation matrix from our quaternion.

For some reason, SymPy seems to need two simplification steps for this ...

[14]:
```
R = sp.trigsimp(sp.trigsimp(q.to_rotation_matrix()))
R
```

[14]:
$$\begin{bmatrix} -\sin(\alpha)\sin(\beta)\sin(\gamma)+\cos(\alpha)\cos(\gamma) & -\sin(\alpha)\cos(\beta) & \sin(\alpha)\sin(\beta)\cos(\gamma)+\sin(\gamma)\cos(\alpha) \\ \sin(\alpha)\cos(\gamma)+\sin(\beta)\sin(\gamma)\cos(\alpha) & \cos(\alpha)\cos(\beta) & \sin(\alpha)\sin(\gamma)-\sin(\beta)\cos(\alpha)\cos(\gamma) \\ -\sin(\gamma)\cos(\beta) & \sin(\beta) & \cos(\beta)\cos(\gamma) \end{bmatrix}$$

#### A.7.2.6 Quaternion to ASDF rotations

Again, please note that the equations from Wikipedia[25] use different conventions for axes and angles.

We already know how to convert a rotation matrix to ASDF angles, and we know how to convert a quaternion to a rotation matrix, so let's try that:

[15]:
```
a, b, c, d = sp.symbols('a:d')
```

[16]:
```
sp.simplify(sp.Quaternion(a, b, c, d).to_rotation_matrix())
```

[16]:
$$\left[\begin{matrix}\frac{a^2+b^2-c^2-d^2}{a^2+b^2+c^2+d^2} & \frac{2(-ad+bc)}{a^2+b^2+c^2+d^2} & \frac{2(ac+bd)}{a^2+b^2+c^2+d^2}\\ \frac{2(ad+bc)}{a^2+b^2+c^2+d^2} & \frac{a^2-b^2+c^2-d^2}{a^2+b^2+c^2+d^2} & \frac{2(-ab+cd)}{a^2+b^2+c^2+d^2}\\ \frac{2(-ac+bd)}{a^2+b^2+c^2+d^2} & \frac{2(ab+cd)}{a^2+b^2+c^2+d^2} & \frac{a^2-b^2-c^2+d^2}{a^2+b^2+c^2+d^2}\end{matrix}\right]$$

Since we assume a unit quaternion, all the denominators are actually 1.

[17]:
```
Rq = sp.simplify(sp.Quaternion(a, b, c, d).to_rotation_matrix().subs(a**2␣
↪+ b**2 + c**2 + d**2, 1))
Rq
```

[17]:
$$\left[\begin{matrix}a^2+b^2-c^2-d^2 & -2ad+2bc & 2ac+2bd\\ 2ad+2bc & a^2-b^2+c^2-d^2 & -2ab+2cd\\ -2ac+2bd & 2ab+2cd & a^2-b^2-c^2+d^2\end{matrix}\right]$$

The *notebook about rotation matrices* (page 77) shows how to obtain $\alpha$, $\beta$ and $\gamma$ from this matrix.

We can get $\alpha$ from the top middle and the central element:

[18]:
```
sp.atan2(-Rq[0, 1], Rq[1, 1])
```

[18]:
$$\operatorname{atan}_2\left(2ad-2bc, a^2-b^2+c^2-d^2\right)$$

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

```
atan2(2*a*d - 2*b*c, a**2 - b**2 + c**2 - d**2)
```

The bottom middle element provides $\beta$:

[20]:
```
sp.asin(Rq[2, 1])
```

[20]:
$$\operatorname{asin}(2ab+2cd)$$

[21]:
```
print(_)
```

```
asin(2*a*b + 2*c*d)
```

[25] https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles#Quaternion_to_Euler_angles_(in_3-2-1_sequence)_conversion

**Note:**

As mentioned in the *notebook about rotation matrices* (page 77), the argument of the `asin()` function has to be in the domain `[-1.0; 1.0]`.

Make sure to handle this case, e.g. by re-normalizing the quaternion.

Finally, $\gamma$ can be obtained from the bottom left and right elements:

[22]:
```
sp.atan2(-Rq[2, 0], Rq[2, 2])
```

[22]: $\operatorname{atan}_2\left(2ac - 2bd, a^2 - b^2 - c^2 + d^2\right)$

[23]:
```
print(_)
```

```
atan2(2*a*c - 2*b*d, a**2 - b**2 - c**2 + d**2)
```

### A.7.2.6.1 Gimbal Lock

As shown in the *notebook about rotation matrices* (page 78), there is a problem when $\beta = \pm 90$ degrees.

For $\beta = 90$ degrees (which means $2ab + 2cd = 1$), we can obtain a value for $\alpha + \gamma$:

[24]:
```
sp.atan2(Rq[0, 2], -Rq[1, 2])
```

[24]: $\operatorname{atan}_2(2ac + 2bd, 2ab - 2cd)$

[25]:
```
print(_)
```

```
atan2(2*a*c + 2*b*d, 2*a*b - 2*c*d)
```

If we for example choose this value to be $\alpha$, this will result in $\gamma = 0$.

Alternatively, we can use this expression:

[26]:
```
sp.atan2(Rq[1, 0], Rq[0, 0])
```

[26]: $\operatorname{atan}_2\left(2ad + 2bc, a^2 + b^2 - c^2 - d^2\right)$

[27]:
```
print(_)
```

```
atan2(2*a*d + 2*b*c, a**2 + b**2 - c**2 - d**2)
```

For $\beta = -90$ degrees (which means $2ab + 2cd = -1$), we can use the following expression for $\alpha + \gamma$:

[28]:
```
sp.atan2(-Rq[0, 2], Rq[1, 2])
```

[28]: $\operatorname{atan}_2(-2ac - 2bd, -2ab + 2cd)$

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

```
atan2(-2*a*c - 2*b*d, -2*a*b + 2*c*d)
```

Again, if we for example choose this value to be $\alpha$, this will result in $\gamma = 0$.

Alternatively, we can use this expression:

```
[30]: sp.atan2(Rq[1, 0], Rq[0, 0])
```

[30]: $\operatorname{atan}_2\left(2ad + 2bc, a^2 + b^2 - c^2 - d^2\right)$

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

```
atan2(2*a*d + 2*b*c, a**2 + b**2 - c**2 - d**2)
```

........................................................................... doc/quaternions.ipynb ends here.

# Appendix B

# Splines

This appendix contains detailed information about many different types of *splines*, including the ones used in the Audio Scene Description Format (ASDF), as described in appendix A. The following content is the documentation for the publicly available Python module `splines`, written by the author of this thesis. At the time of writing, a verbatim copy of this text is also available online at `https://splines.readthedocs.io/`. The online version might be updated in the future, though.

Apart from being potentially improved over time, it is in fact recommended to read the online version instead of the printed version, because the online version contains a lot of animations which could not be included in this print version.

Nearly all of the content of this appendix has been written using Jupyter notebooks (see `https://jupyter.org/`), all of which can be downloaded from the sub-directory `doc/` at `https://github.com/AudioSceneDescriptionFormat/splines/` for further study and experimentation.

## B.1 Introduction

This is the documentation for the `splines`[1] module for Python. However, instead of a Python module with a bit of documentation, this project is mostly documentation, with a bit of Python module at the side. The goal is not so much to provide a turn-key software for using splines, but rather to provide the background and mathematical derivations for fully understanding the presented types of splines and their inter-relations. The Python module serves mostly for experimenting further with the presented ideas and methods. Therefore, the implementation is not focused on efficiency.

The documentation consists of two main parts. The *first part* (page 88) investigates some *polynomial splines* in their natural habitat, the Euclidean space. In the unlikely case you are reading this and don't know what "spline" means, the first part also contains *a definition of the term* (page 101) and a description of some of the common properties of splines. The *second part* (page 232) leaves the comfort zone of flat space and tries to apply some of the approaches from the first part to the curved space of rotations. The Python module is similarly split into two parts whose API documentation is available at *`splines`* (page 281) and *`splines.quaternion`* (page 287), respectively.

This project was originally inspired by Millington (2009), who concisely lists the *basis matrices* (a.k.a. *characteristic matrices*) of a few common types of splines and also provides

[1] `https://pypi.org/project/splines/`

matrices that can be used to convert *control points* between those different types. However, the derivation of those matrices is not shown. Furthermore, the paper only considers *uniform* curves, where all parameter intervals have a size of 1. One goal of this documentation is to show the derivation of all equations and matrices. The derivations often utilize SymPy[2] to make them more reproducible and to ease further experimentation. A special focus is put on *non-uniform* splines, which seem to have been neglected in some of the literature and especially in some online resources.

Another focus is the speed along curves. In many applications only the shape (a.k.a. the image[3]) of a curve matters. However, sometimes it is important how fast a point travels along a spline when changing the parameter (which can be interpreted as *time*). The "timing" of a spline is not visible in a static line plot (as is often used by default). That's why most of the plots in the following sections will instead use dots at regular parameter intervals, for example 15 dots per second. If a spline already has the desired image but the wrong timing, this can be fixed by *Re-Parameterization* (page 229).

A non-goal of this Python module and its documentation is to cover all possible types of splines. Maybe some additional types will be added in the future, but the list will always stay incomplete. One of the most glaring omissions for now are B-splines[4], which are mentioned a few times but not properly derived nor implemented. Another family of splines that is missing are *rational splines*, and therefore also their most popular member NURBS[5]. Spline surfaces are not covered either.

## B.2 Polynomial Curves in Euclidean Space

This section is mostly about different types of univariate non-rational polynomial splines in one-, two- and three-dimensional Euclidean space – for an application in a four-dimensional space, see *the section about 4D quaternion interpolation* (page 277).

But before diving into *splines* (page 101) – and before even defining what they are – we will discuss a few basics about polynomial curves and a spline-less interpolation method called *Lagrange interpolation* (page 92).

Many of the approaches shown in this section will later be adapted to the context of *rotation splines* (page 232).

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

### B.2.1 Parametric Polynomial Curves

The building blocks for *polynomial splines* are of course polynomials[6].

But first things first, let's import SymPy[7] and a few helper functions from `helper.py`:

[1]:
```python
import sympy as sp
sp.init_printing(order='grevlex')
from helper import plot_basis, plot_sympy, grid_lines, plot_spline_2d
```

[2] https://www.sympy.org/
[3] https://en.wikipedia.org/wiki/Image_(mathematics)
[4] https://en.wikipedia.org/wiki/B-spline
[5] https://en.wikipedia.org/wiki/Non-uniform_rational_B-spline
[6] https://en.wikipedia.org/wiki/Polynomial
[7] https://www.sympy.org/

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*, a.k.a. *power 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 creating polynomials of degree 3, which are also called *cubic* polynomials.

The ordering of the basis functions is purely a matter of convention, here we are sorting them in order of descending powers.

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 (and even higher-dimensional spaces).

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

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

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

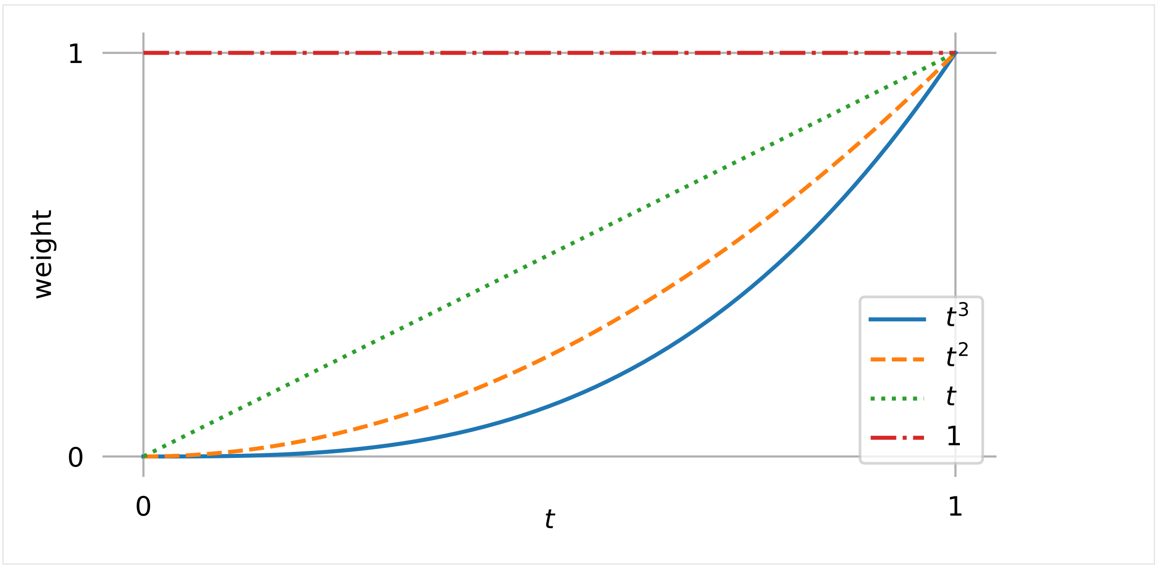
```
[5]: b_monomial.dot(coefficients)
```

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

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

Let's take a closer look at those basis functions:

```
[6]: plot_basis(*b_monomial)
```



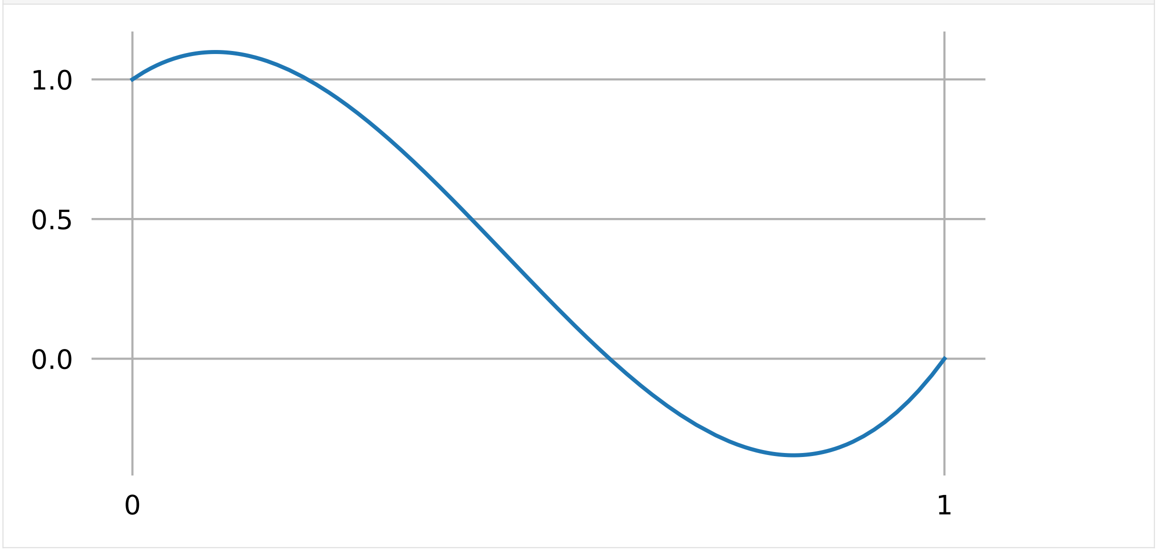
It doesn't look like much, but every conceivable cubic polynomial can be expressed as exactly one linear combination of those basis functions (i.e. using one specific list of coefficients).

An example polynomial that's not in canonical form …

[7]:
```
example_polynomial = (2 * t - 1)**3 + (t + 1)**2 - 6 * t + 1
example_polynomial
```

[7]: $(2t-1)^3 + (t+1)^2 - 6t + 1$

[8]:
```
plot_sympy(example_polynomial, (t, 0, 1))
grid_lines([0, 1], [0, 0.5, 1])
```



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

[9]:
```
example_polynomial.expand()
```

[9]: $8t^3 - 11t^2 + 2t + 1$

Any polynomial can be rewritten using any set of basis functions (as long as the degree of the basis function set matches the degree of the polynomial).

In later sections we will see more basis functions, for example those that are used for *Hermite* (page 109), *Bézier* (page 136) and *Catmull–Rom* (page 165) splines. In those sections we will also see how to convert between different bases by means of matrix multiplication.

In the previous example, we 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:

[10]:
```python
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
```

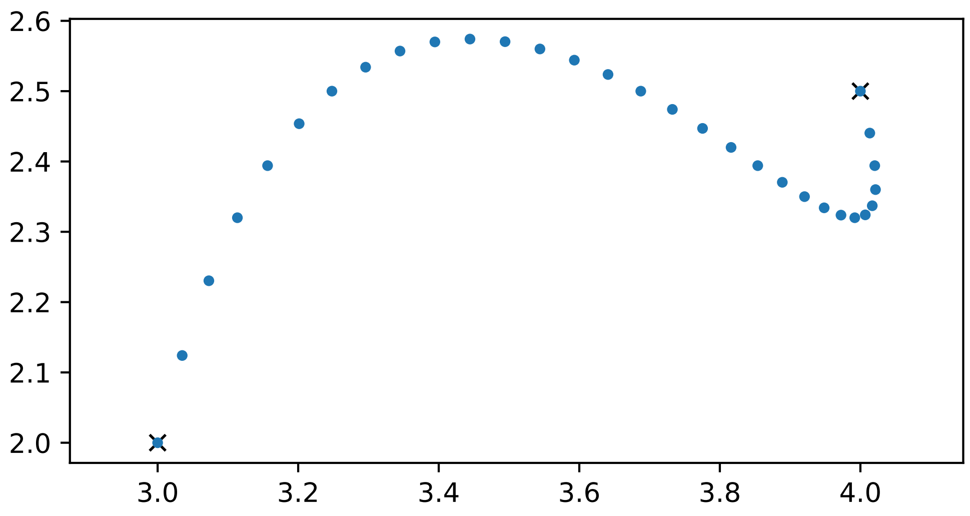
**Note**

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

[11]:
```python
poly_2d = CubicPolynomial([-1.5, 5], [1.5, -8.5], [1, 4], [3, 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 for plotting:

[12]:
```python
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 282) can be used to try this with arbitrary polynomial degree.

........................................................ `doc/euclidean/polynomials.ipynb` ends here.

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

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

### B.2.2 Lagrange Interpolation

Before diving into splines, let's have a look at an arguably simpler interpolation method using polynomials: Lagrange interpolation[9].

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
```

#### B.2.2.1 One-dimensional Example

Assume we have $N$ time instants $t_i$, with $0 \le i < N$ ...

[2]:
```
ts = -1.5, 0.5, 1.7, 3.5, 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 $\ell_i(t)$ for each of the $N$ given time instants, which will be weighted by the associated $x_i$. The final interpolation function is the weighted sum of these $N$ polynomials:

$$L(t) = \sum_{i=0}^{N-1} x_i \ell_i(t)$$

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. For example, let's look at the basis for $i = 3$:

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

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

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

```
[5]: array([ -0. ,    0. ,   -0. , -13.5,    0. ])
```

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

Now, for the first requirement, we can divide each term to yield 1 when evaluated at $t = t_3 = 3.5$ (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)) / (3.5 - (-1.5)) *
             (t - 0.5) / (3.5 - 0.5) *
             (t - 1.7) / (3.5 - 1.7) *
             (t - 4) / (3.5 - 4))
```

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

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

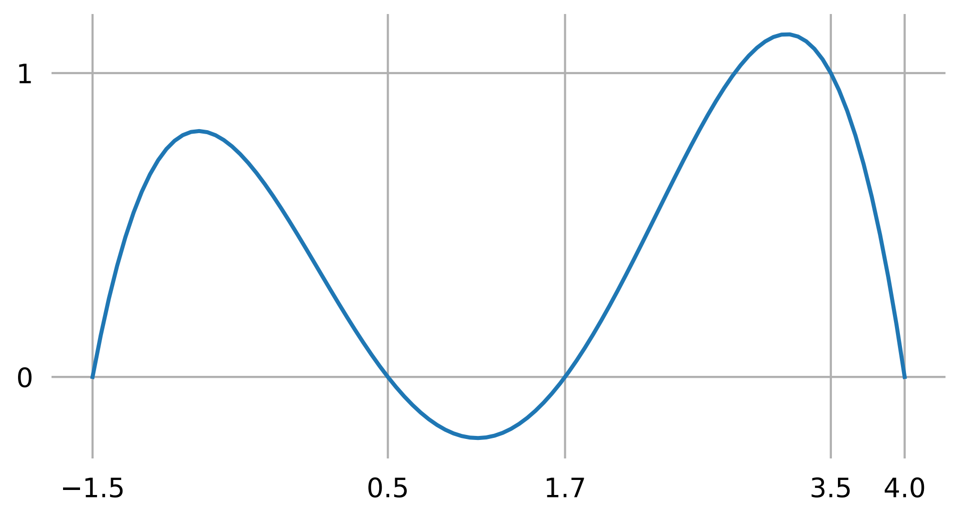
That's it!

To get a better idea what's going on between the given time instances $t_i$, 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 $t_3 = 3.5$, 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
          ])
```

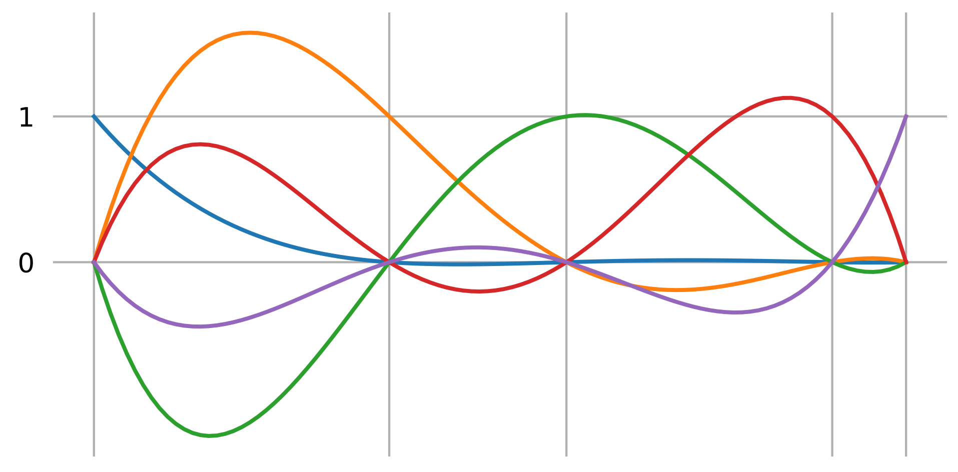
Putting this in mathematic notation, Lagrange basis polynomials can be written as

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

Now we can calculate and visualize all 5 basis 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])
```

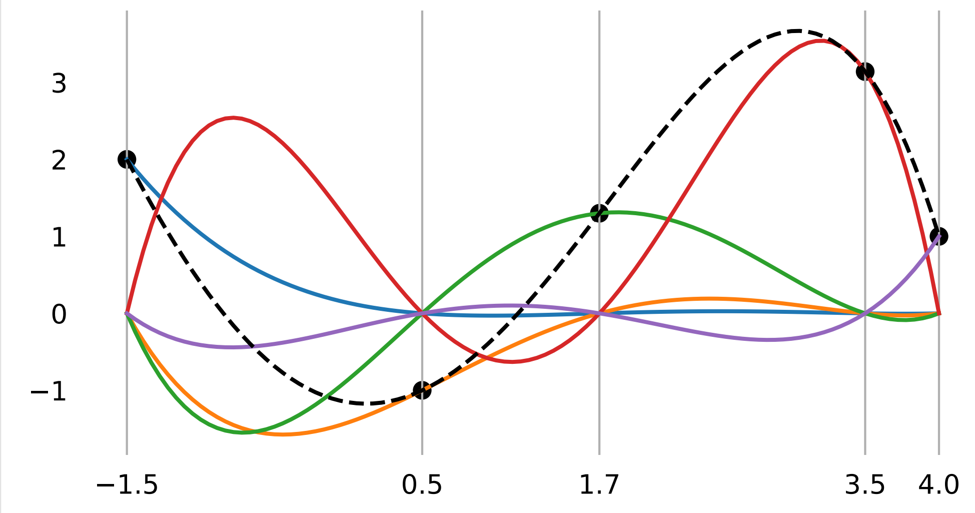


Finally, the interpolated values $L(t)$ can be obtained by applying the given $x_i$ values as weights to the polynomials $\ell_i(t)$ and summing everything up together:

```
[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)
```



### B.2.2.2 Neville's Algorithm

An alternative way to calculate interpolated values is Neville's algorithm[10]. We mention this algorithm mainly because it is referenced in the *derivation of non-uniform Catmull–Rom splines* (page 175) and the *description of the Barry–Goldman algorithm* (page 181).

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

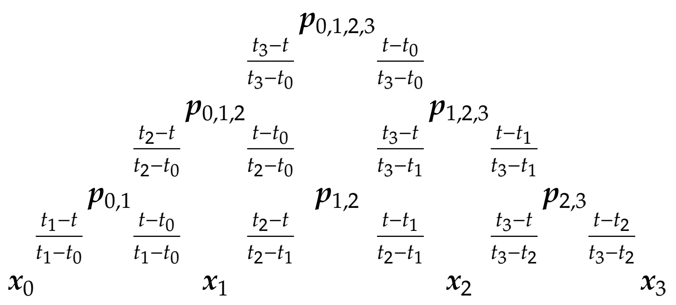
[17]:
```
def lerp(xs, ts, t):
    """Linear intERPolation.

    Returns the interpolated value(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 less 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.

[10] https://en.wikipedia.org/wiki/Neville's_algorithm

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 less 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.

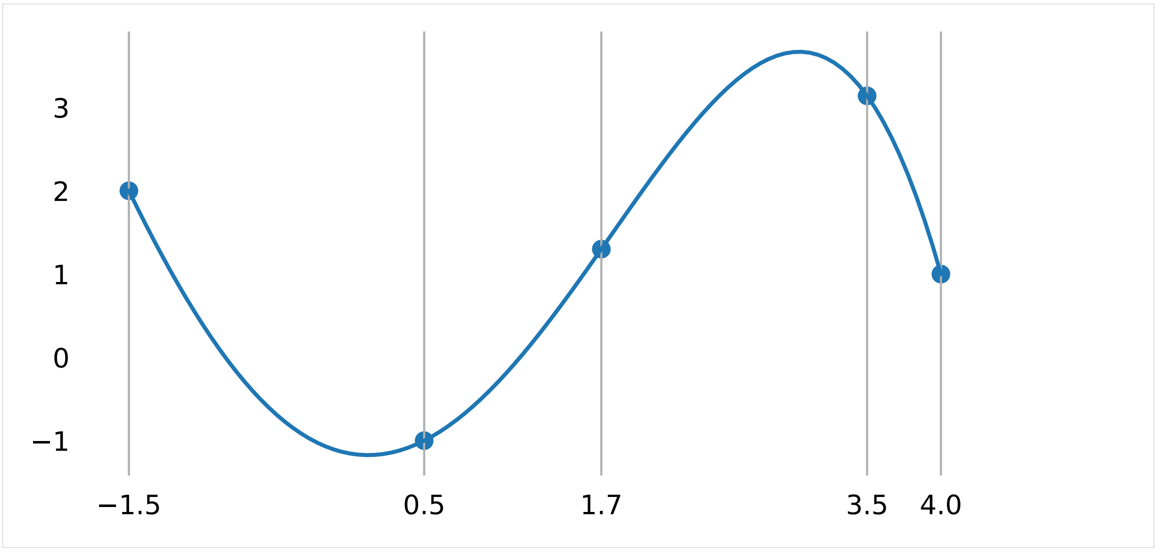
Barry and Goldman (1988) show (in figure 2) the cubic case, which looks something like this:



The polynomial $\boldsymbol{p}_{0,1,2,3}(t)$ at the apex can be evaluated for $t_0 \leq t \leq t_3$. For a detailed explanation of this triangular scheme, see the *notebook about the Barry–Goldman algorithm* (page 182). Neville's algorithm can be implemented for arbitrary degree:

```
[18]: 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*.

          """
          if len(xs) != len(ts):
              raise ValueError('xs and ts must have the same length')
          while len(xs) > 1:
              step = len(ts) - len(xs) + 1
              xs = [
                  lerp(*args, t)
                  for args in zip(zip(xs, xs[1:]), zip(ts, ts[step:]))]
          return xs[0]
```

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



### B.2.2.3 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)
```

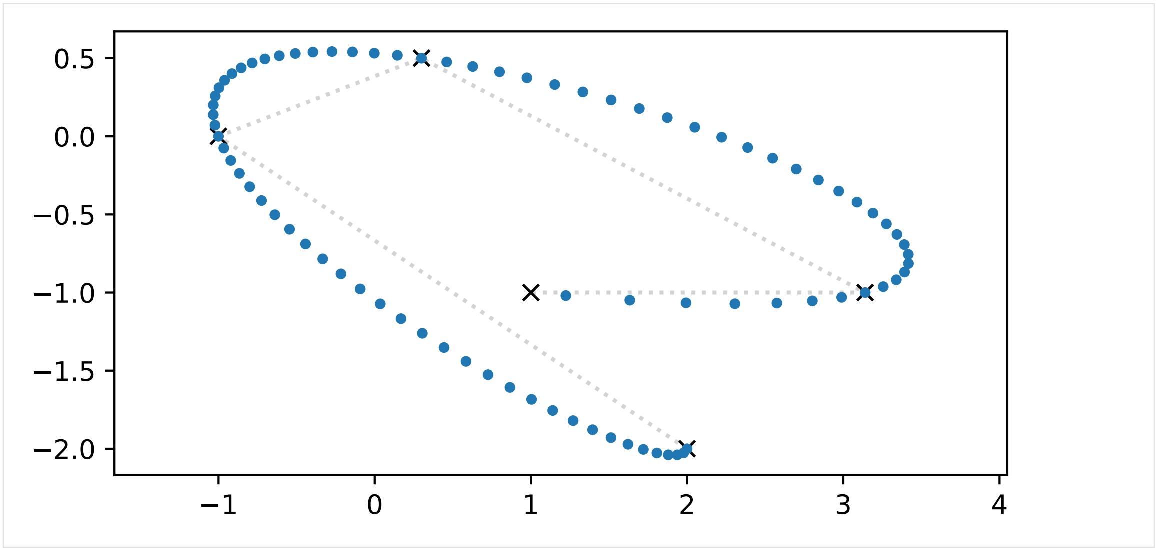
Since this class has the same interface as the splines that will be discussed in the following sections, we can use a spline helper function from `helper.py` for plotting:

[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), (0.3, 0.5), (3.14, -1), (1, -1)], ts)
```

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

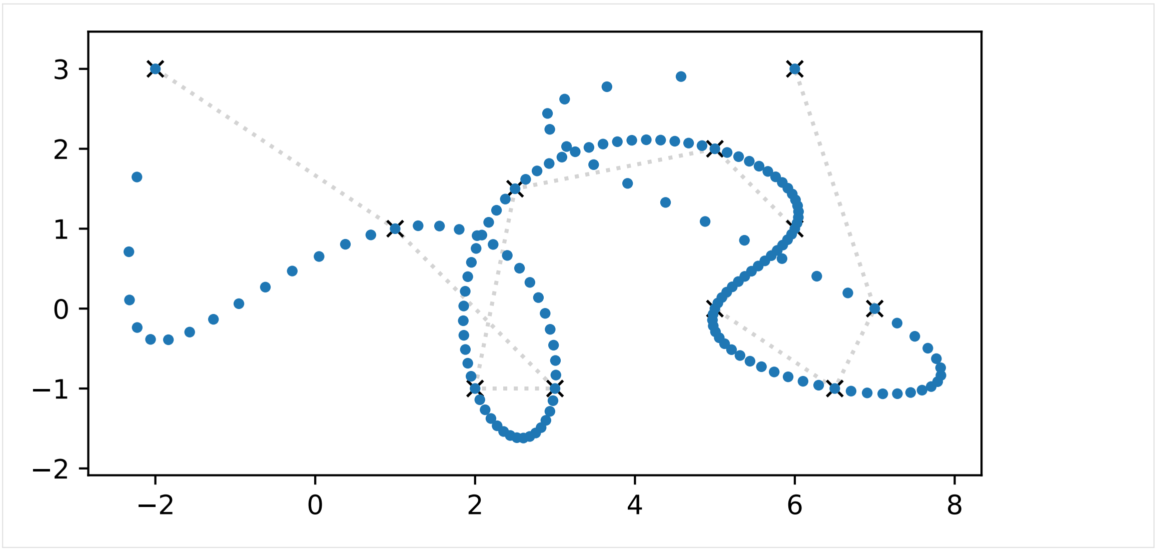


### B.2.2.4 Runge's Phenomenon

This seems to work to some degree, 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]:
```
vertices1 = [
    (-2, 3),
    (1, 1),
    (3, -1),
    (2, -1),
    (2.5, 1.5),
    (5, 2),
    (6, 1),
    (5, 0),
    (6.5, -1),
    (7, 0),
    (6, 3),
]
```

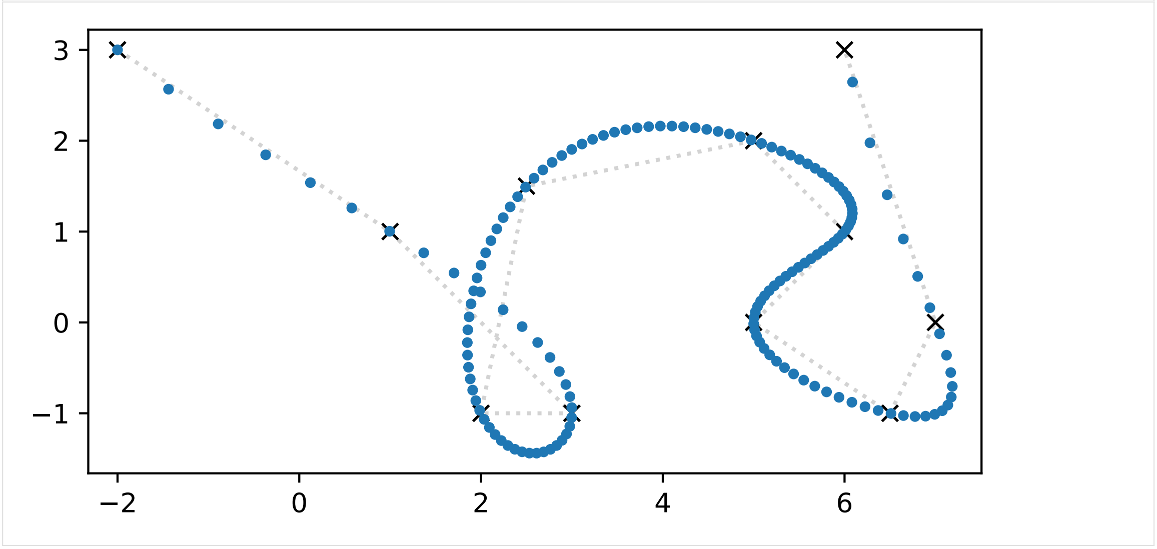
[25]:
```
l2 = Lagrange(vertices1, range(len(vertices1)))
plot_spline_2d(l2)
```



Here we see a severe overshooting effect, most pronounced at the beginning and the end of the curve. Moving some vertices can make this even worse. This effect is called Runge's phenomenon[11]. A possible mitigation for this overshooting is to use so-called Chebyshev nodes[12] as time instances:

[26]:
```
def chebyshev_nodes(a, b, n):
    k = np.arange(n) + 1
    nodes = np.cos(np.pi * (2 * k - 1) / (2 * n))
    return (a + b) / 2 - (b - a) * nodes / 2
```

[27]:
```
l3 = Lagrange(vertices1, chebyshev_nodes(0, len(vertices1) - 1,␣
↪len(vertices1)))
plot_spline_2d(l3)
```



This is definitely better. But it gets worse again when we move a few of the vertices.

[28]:
```
vertices2 = [
    (0, -1),
```

(continues on next page)

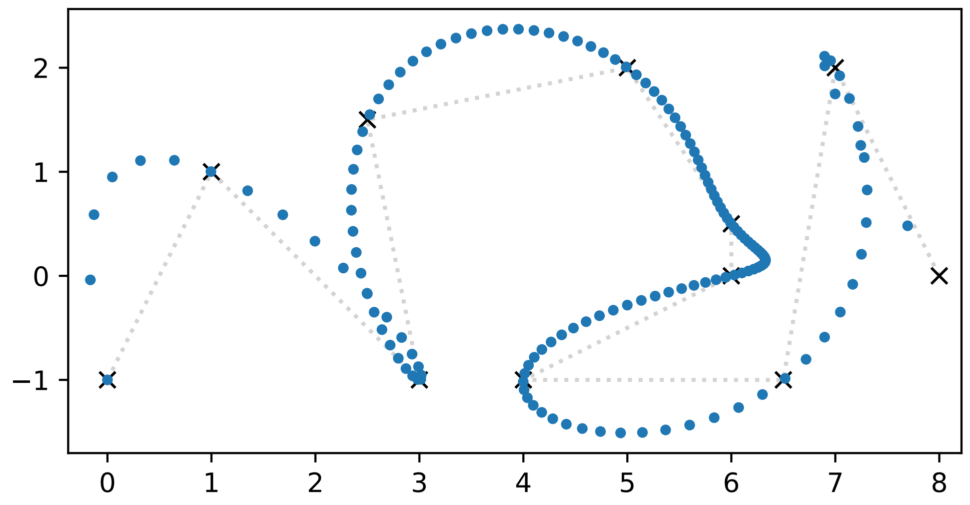
[11] https://en.wikipedia.org/wiki/Runge's_phenomenon

[12] https://en.wikipedia.org/wiki/Chebyshev_nodes

(continued from previous page)

```
    (1, 1),
    (3, -1),
    (2.5, 1.5),
    (5, 2),
    (6, 0.5),
    (6, 0),
    (4, -1),
    (6.5, -1),
    (7, 2),
    (8, 0),
]
```

```
[29]: l4 = Lagrange(vertices2, chebyshev_nodes(0, len(vertices2) - 1,␣
       ↪len(vertices2)))
      plot_spline_2d(l4)
```

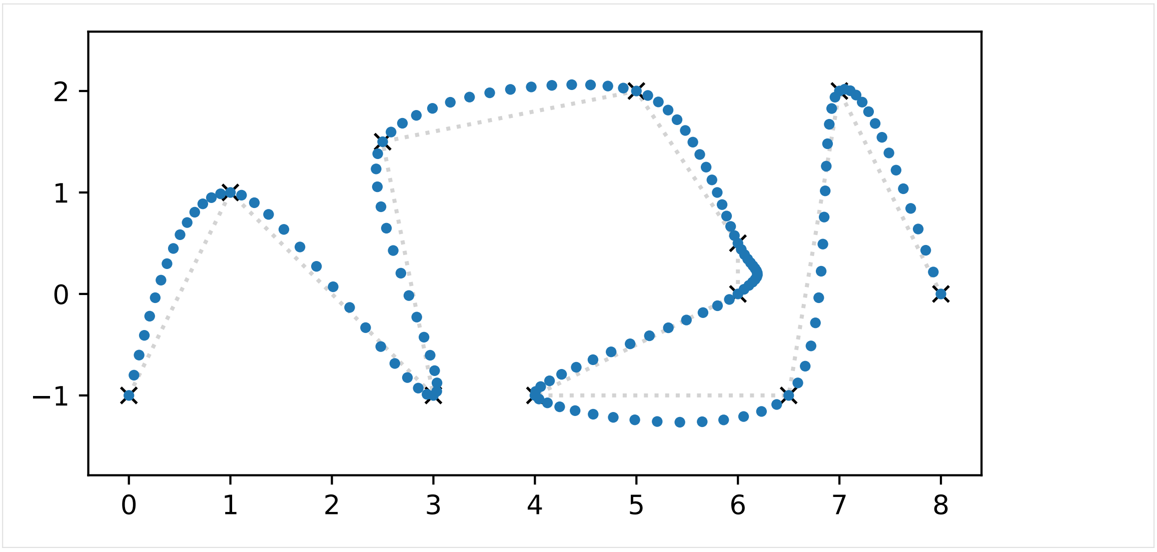


Long story short, Lagrange interpolation is typically not suitable for drawing curves. For comparison, and as a teaser for the following sections, let's use the same vertices to create a uniform *Catmull–Rom spline* (page 155):

```
[30]: import splines
```

```
[31]: cr_spline = splines.CatmullRom(vertices2)
```

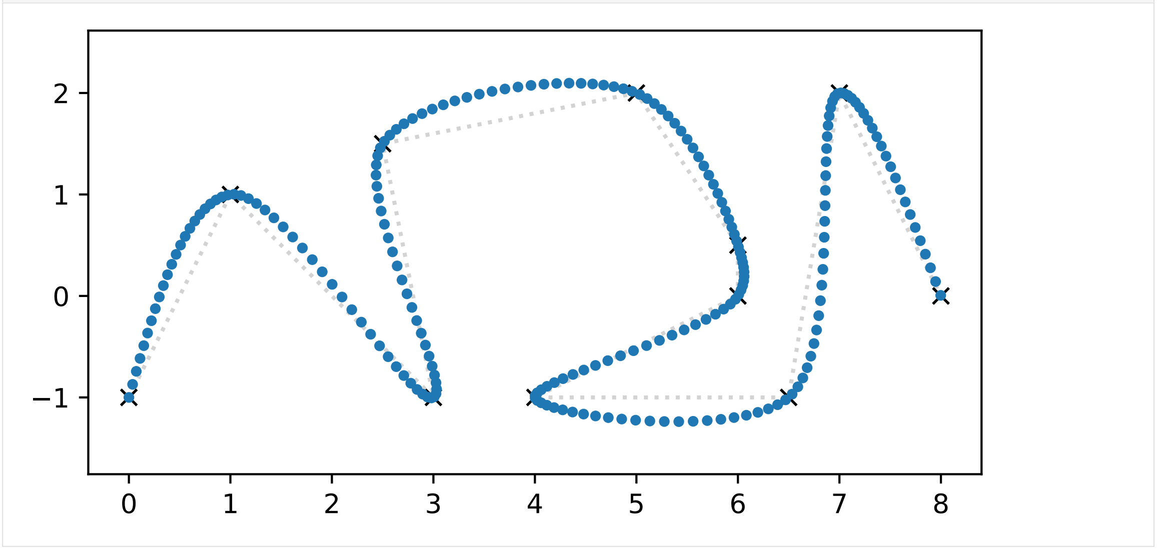
```
[32]: plot_spline_2d(cr_spline)
```



And to get an even better fit, we can try a *centripetal Catmull–Rom spline* (page 163):

[33]:
```
cr_centripetal_spline = splines.CatmullRom(vertices2, alpha=0.5)
```

[34]:
```
plot_spline_2d(cr_centripetal_spline)
```



**Note**

The class *splines.CatmullRom* (page 283) uses *"natural" end conditions* (page 207) by default.

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

### B.2.3 Splines

The term *spline* for the mathematical description of a smooth piecewise curve was introduced by Schoenberg (1946), with reference to a drawing tool called spline[13].

[13] https://en.wiktionary.org/wiki/spline

> A spline is a simple mechanical device for drawing smooth curves. It is a slender flexible bar made of wood or some other elastic material. The spline is place[d] on the sheet of graph paper and held in place at various points by means of certain heavy objects (called "dogs" or "rats") such as to take the shape of the curve we wish to draw.
>
> —Schoenberg (1946), page 67

The term is defined in the context of what is nowadays known as *natural splines* (page 124), especially cubic natural splines (i.e. of degree 3; i.e. of order 4), which have $C^2$ continuity.

> For $k = 4$ they represent approximately the curves drawn by means of a spline and for this reason we propose to call them *spline curves of order k*.
>
> —Schoenberg (1946), page 48

#### B.2.3.1 Definition

Different authors use different definitions for the term *spline*, here is ours: *splines* are composite parametric curves. Splines are typically used for defining curves in one-, two- or three-dimensional Euclidean space. Such splines will be described in the following sections. Later, we will also have a look at *rotation splines* (page 232).

Sometimes it is not obvious whether the term *spline* refers to the composite curve or to one of its segments, especially when talking about *Bézier splines* (page 134). In the rest of this text we are using the term *spline* to refer to the entire composite curve.

#### B.2.3.2 Properties

Different types of splines have different properties. In the following, we list the most important properties, focusing on the types of splines that will be described in more detail in later sections.

**piecewise**

Arguably the most important property of splines is that they are composed of somewhat independent pieces. This allows using simpler mathematical objects for the pieces, while still being able to construct a more or less arbitrarily complicated composite curve. For example, as shown in the previous section about *Lagrange interpolation* (page 92), using a curve with a high polynomial degree can lead to unintended behavior like *Runge's phenomenon* (page 98). This can be avoided by using multiple polynomial pieces of lower degrees.

**parametric**

Here we are only talking about *univariate* curves, i.e. curves with one parameter, i.e. a single real number, but similar approaches can be used to describe surfaces with two parameters. We are normally using the symbol $t$ for the free parameter. This parameter can often intuitively be interpreted as *time*, but it doesn't have to.

The individual segments of a spline are of course also parametric, and they may have their own parameter ranges (often, but not necessarily, the so-called *unit interval* from 0 to 1), which have to be calculated from the appropriate sub-range of the main spline parameter.

A spline can also be re-parameterized, see *Re-Parameterization* (page 229).

The sequence of parameter values at the start and end of segments is sometimes – e.g. by Gordon and Riesenfeld (1974) – called the *knot vector*. In the accompanying *Python Module* (page 281), however, it is called `grid`.

**non-uniform**

The parameter range of a spline can be uniquely separated into the parameter ranges of its segments. If those sub-ranges all have the same length, the spline is called *uniform*.

When a uniform spline has curve segments of very different lengths, the speed along the curve (assuming that the parameter $t$ is interpreted as time) varies strongly. By using *non-uniform* parameter intervals, this can be avoided.

**continuous**

Splines are not necessarily continuous. The segments of a spline might be defined by discontinuous functions, but for most practical applications it is more common to use continuous functions. Often, some derivatives of these functions are continuous as well. If the spline segments are polynomials, they are always continuous, and so are all their derivatives.

However, even if its segments are continuous, that doesn't automatically mean that the whole spline is continuous. The transitions between segments can still be discontinuous. But again, in most practical applications the transitions are continuous. If that's the case, the spline is said to have $C^0$ *continuity*. The spline is called $C^1$ continuous if the first derivatives of the two neighboring segments at each transition are equal and $C^2$ continuous if also the second derivatives match.

**control points**

Splines are fully defined by the mathematical functions of their segments and the corresponding parameter ranges. However, those functions (and their coefficients) have to be chosen somehow. And that's what differentiates different types of splines.

For some applications it is desired to specify a sequence of *control points* (sometimes also called *vertex/vertices*) where the curve is supposed to pass through. Based on those points, the appropriate functions for the spline segments are constructed. The *Catmull–Rom splines* (page 155) and *natural splines* (page 124) are examples where segments are derived from such a sequence of control points.

Some splines, most notably *Bézier splines* (page 134), only pass through some of their control points and the remaining control points affect the shape of the curve between those points.

The set of all control points, connected by straight lines, is sometimes called *control polygon*.

Some splines have a set of control points where they pass through and additional values that are not points at all. We call them *control values*. For example, *Hermite splines* (page 105) pass through a set of control points, but they need additional information about the tangent vectors (i.e. the first derivatives) at the transitions between segments. For higher-order splines they also need the second and higher derivatives.

**interpolating**

Splines are called *interpolating* if they pass through all of their aforementioned control points. If a spline is not interpolating, it is called *approximating*.

Here we are almost exclusively talking about interpolating splines. A notable special case are *Bézier splines* (page 134), which pass through a sequence of control points, but between each pair of those interpolated control points there are $d-1$ (where $d$ is the degree) additional control points that are only approximated by the curve (and they can be used to control the shape of the curve).

**local control**

For some types of splines, when changing a single control value, the shape of the whole curve changes. These splines are said to have *global control*. For many applications, however, it is preferable, when a control value is changed, that the shape of the curve only changes in the immediate vicinity of that control value. This is called *local control*.

**additional parameters**

Some types of splines have additional parameters, either separately for each vertex, or the same one(s) for all vertices. An example are *Kochanek–Bartels splines* (page 191) with their *tension*, *continuity* and *bias* parameters.

**polynomial**

The curve segments that make up a spline can have an arbitrary mathematical description. Very often, polynomial curve segments are used, and that's also what we will be mostly using here. The polynomials will be defined by their basis functions and corresponding coefficients, as described in the *notebook about polynomial parametric curves* (page 88).

The following properties are only relevant for polynomial splines.

**degree**

The degree of a polynomial spline is the highest degree among its segments. Splines of degree 3, a.k.a. *cubic* splines, are very common for drawing smooth curves. Old-school references by authors like Boor (1978) might use the term *order*, which is one more than the degree, which means that cubic splines are of order 4. We will mostly consider cubic splines, but some of the presented algorithms allow arbitrary degree, for example *De Casteljau's algorithm* (page 136).

**non-rational**

The splines discussed here are defined by one polynomial per segment. However, there are also splines whose segments are defined by ratios of polynomials instead. Those are called *rational* splines. Rational splines are invariant under perspective transformations (non-rational splines are only invariant under rotation/scale/translation), and they can precisely define conic sections (e.g. circles). They are also the foundation for NURBS[14].

### B.2.3.3 Types

There are an infinite number of types of splines, only very few of which will be presented in the following sections. Some of them can create the same curve from different control

[14] `https://en.wikipedia.org/wiki/Non-uniform_rational_B-spline`

values, like *Hermite splines* (page 105) and *Bézier splines* (page 134). Some create different curves from the same control values, like *Catmull–Rom splines* (page 155) and *natural splines* (page 124). Some have additional parameters to control the shape of the curve, like *Kochanek–Bartels splines* (page 191) with their TCB values.

Some spline types have certain constraints on the transitions between segments, for example, natural splines require $C^2$ continuity. Other splines have no such constraints, like for example Hermite splines, which allow specifying arbitrary derivatives at their segment transitions.

Cubic splines cannot be interpolating *and* have $C^2$ continuity *and* local control at the same time.

| type | local control | continuity | interpolating |
|---|---|---|---|
| *Catmull–Rom splines* (page 155) | yes | $C^1$ | yes |
| *natural splines* (page 124) | no | $C^2$ | yes |
| B-splines[15] | yes | $C^2$ | no |

Kochanek–Bartels splines with $C = 0$ are in the same category as Catmull–Rom splines (which are a subset of former).

From any polynomial segment of a certain degree the control values according to any polynomial spline type (of that same degree) can be computed and vice versa. This means that different types of polynomial splines can be unambiguously (if using the same parameter intervals) converted between each other as long as the target spline has the same or weaker constraints. For example, any natural spline can be converted into its corresponding Bézier spline. The reverse is not true. Catmull–Rom splines and natural splines can generally not be converted between each other because they have mutually incompatible constraints.

### B.2.4 Hermite Splines

Hermite splines[16] (named after Charles Hermite[17]) are the building blocks for many other types of interpolating polynomial splines, for example *natural splines* (page 124) and *Catmull–Rom splines* (page 155).

A Python implementation of (cubic) Hermite splines is available in the *splines.CubicHermite* (page 283) class.

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

#### B.2.4.1 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.

[15] `https://en.wikipedia.org/wiki/B-spline`
[16] `https://en.wikipedia.org/wiki/Cubic_Hermite_spline`
[17] `https://en.wikipedia.org/wiki/Charles_Hermite`

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

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

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

However, *cubic Hermite splines* are so overwhelmingly common that they are often simply referred to as *Hermite splines*. From this point forward, we will only be considering *cubic* Hermite splines.

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

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

We import a few helper functions from `helper.py`:

[3]:
```
from helper import plot_spline_1d, plot_slopes_1d, grid_lines
from helper import plot_spline_2d, plot_tangents_2d
```

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

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

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

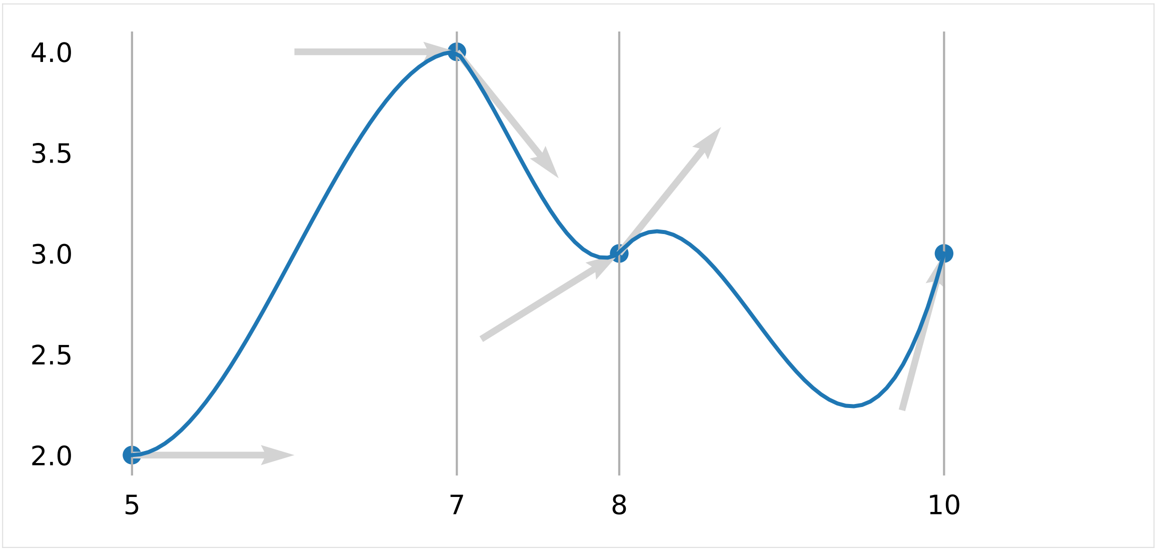
[5]:
```
slopes = 0, 0, -1, 0.5, 1, 3
```

We are using the *splines.CubicHermite* (page 283) class to create the spline:

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

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

[7]:
```
plot_spline_1d(s1)
plot_slopes_1d(slopes, values, grid)
grid_lines(grid)
```



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

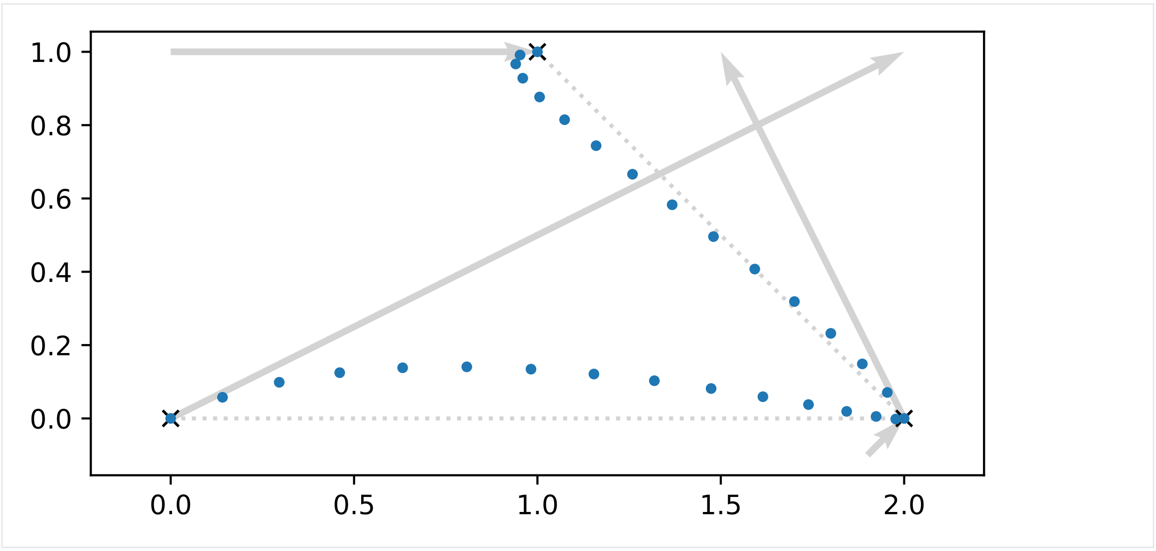
[8]:
```
vertices = [
    (0, 0),
    (2, 0),
    (1, 1),
]
```

The derivative of a curve is its tangent vector, so here is a list of associated tangent vectors (outgoing, incoming, outgoing, incoming, ...):

[9]:
```
tangents = [
    (2, 1),
    (0.1, 0.1),
    (-0.5, 1),
    (1, 0),
]
```

[10]:
```
s2 = splines.CubicHermite(vertices, tangents)
```

[11]:
```
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 283) 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:

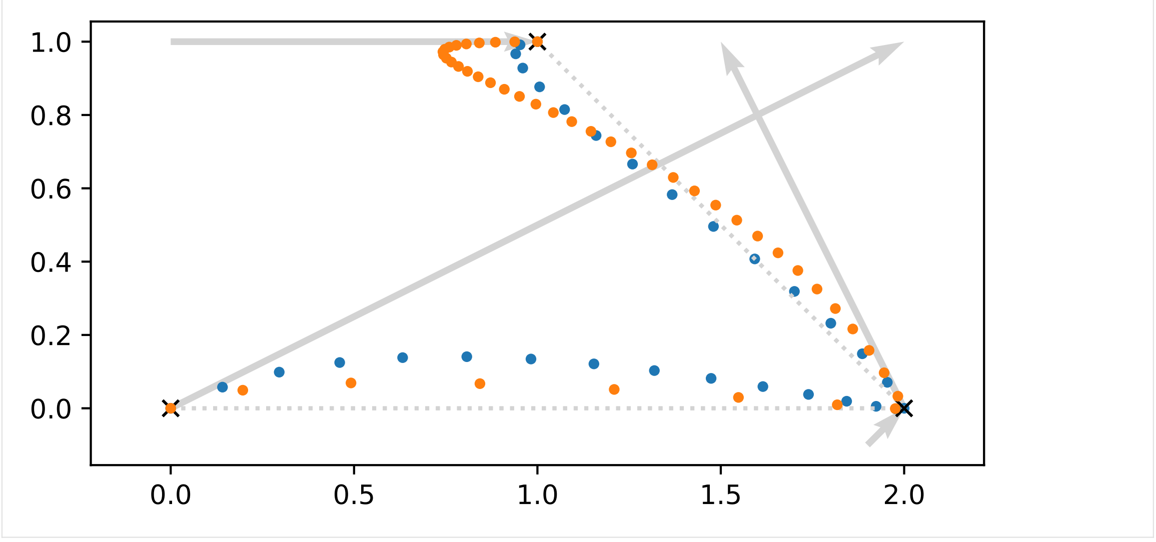
[12]:
```
grid = 0, 0.5, 3
```

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!):

[13]:
```
s3 = splines.CubicHermite(vertices, tangents, grid=grid)
```

[14]:
```
plot_spline_2d(s3, ax=ax)
fig
```

[14]:



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 155). *Kochanek–Bartels* *splines* (page 191) can also be $C^1$ continuous, but only if their "continuity" parameter $C$ is zero.

There is one unique choice of all of a cubic Hermite spline's tangents – given certain *end conditions* (page 207) – that leads to continuous second derivatives at all vertices, making the spline $C^2$ continuous. This is what *natural splines* (page 124) are all about.

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

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

### B.2.4.2 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 118).

[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, let's look at only one specific polynomial segment, let's say the fifth one. It goes from $\boldsymbol{x}_4$ to $\boldsymbol{x}_5$ and it is referred to as $\boldsymbol{p}_4(t)$, where $0 \leq t \leq 1$. The results will be easily generalizable to an arbitrary polynomial segment $\boldsymbol{p}_i(t)$ from $\boldsymbol{x}_i$ to $\boldsymbol{x}_{i+1}$, where $0 \leq t \leq 1$.

The polynomial has 4 coefficients, $\boldsymbol{a}_4$ to $\boldsymbol{d}_4$.

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

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

Combined with the *monomial basis* ...

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

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

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

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

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

For more information about polynomials, see *Polynomial Parametric Curves* (page 88).

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

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

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

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

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

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

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

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

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

$x_4 = a_4$

$x_5 = a_4 + b_4 + c_4 + d_4$

$\dot{x}_4 = b_4$

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

#### B.2.4.2.1 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]:
$$\begin{bmatrix} x_4 \\ x_5 \\ \dot{x}_4 \\ \dot{x}_5 \end{bmatrix}$$

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

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

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

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

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

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

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

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

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

... 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]:

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

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]:

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

Again, for copy & paste:

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

Now we have a new way to write the polynomial $\boldsymbol{p}_4(t)$, given our four control values. We take those control values, left-multiply them by the Hermite basis matrix $M_\text{H}$ (which gives us a column vector of coefficients), which we can then left-multiply by the monomial basis:

[20]:
```
sp.MatMul(b_monomial, M_H.expr, control_values_H.name)
```

[20]:

$$\begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_4 \\ \boldsymbol{x}_5 \\ \dot{\boldsymbol{x}}_4 \\ \dot{\boldsymbol{x}}_5 \end{bmatrix}$$

#### B.2.4.2.2 Basis Polynomials

However, instead of calculating from right to left, we can also start at the left and multiply the monomial basis with the Hermite basis matrix $M_\text{H}$, which yields (a row vector containing) the *Hermite basis polynomials*:

[21]:
```
b_H = NamedMatrix(r'{b_\text{H}}', b_monomial * M_H.expr)
b_H.factor().T
```

[21]:

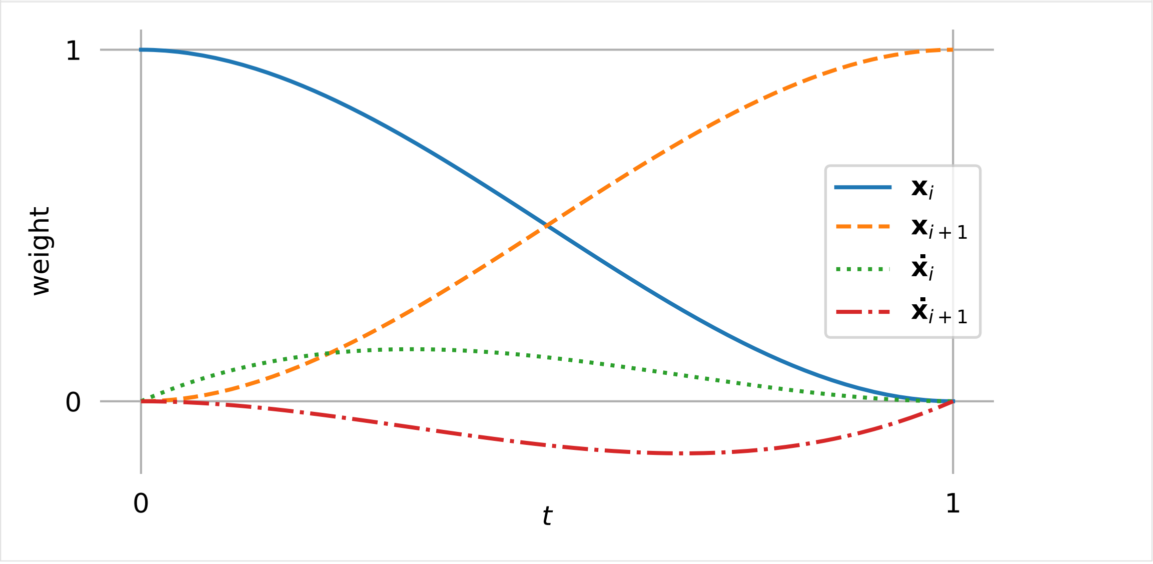
$$b_\text{H}^T = \begin{bmatrix} (t-1)^2 \cdot (2t+1) \\ -t^2 \cdot (2t-3) \\ t\,(t-1)^2 \\ t^2\,(t-1) \end{bmatrix}$$

The multiplication of this row vector with the column vector of control values again produces the polynomial $\boldsymbol{p}_4(t)$.

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

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

```
[23]: 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 $\boldsymbol{x}_i$ 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 $\boldsymbol{x}_i$ (which is exactly what we wanted to happen!).

Similarly, the basis function associated with $\dot{\boldsymbol{x}}_i$ 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 $\dot{\boldsymbol{x}}_i$ (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 $\boldsymbol{x}_{i+1}$ is the only one that has a non-zero value. More specifically, it has the value 1. Finally, the basis function associated with $\dot{\boldsymbol{x}}_{i+1}$ 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:

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

[24]:

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

#### B.2.4.2.3 Example Plot

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

[25]:
```
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`.

[26]:
```
from helper import plot_spline_2d, plot_tangents_2d
```

[27]:
```
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**

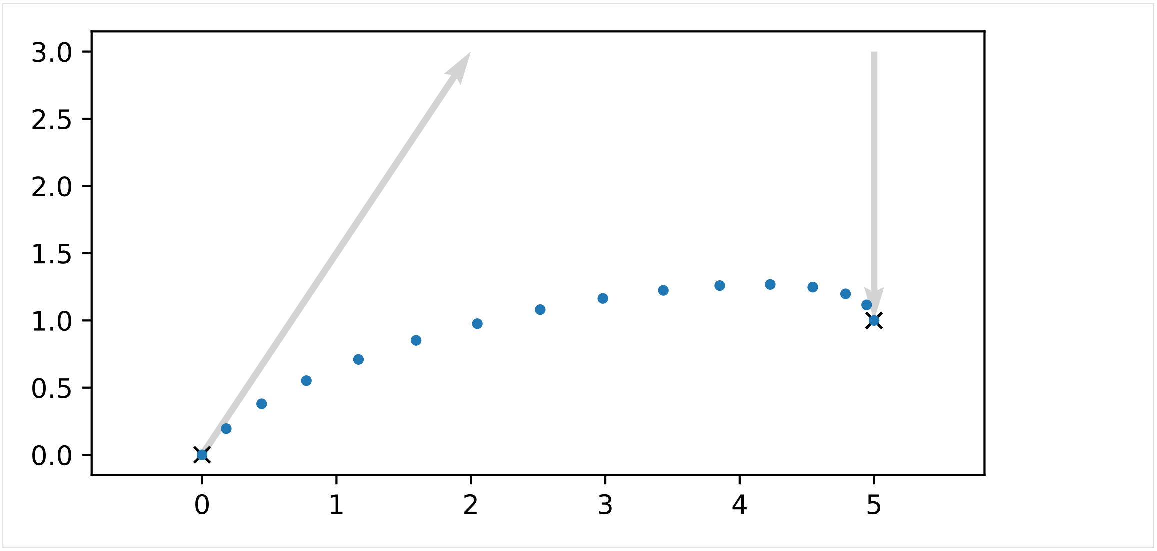
The `@` operator is used here to do NumPy's matrix multiplication[18].

[28]:
```
vertices = [0, 0], [5, 1]
tangents = [2, 3], [0, -2]
```

[29]:
```
s = UniformHermiteSegment([*vertices, *tangents])
```

[30]:
```
plot_spline_2d(s, chords=False)
plot_tangents_2d(tangents, vertices)
```

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



#### B.2.4.2.4 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:

```
[31]: control_values_H.name
```

[31]:

$$\begin{bmatrix} x_4 \\ x_5 \\ \dot{x}_4 \\ \dot{x}_5 \end{bmatrix}$$

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

Let's use the point $\tilde{x}_4$ as a "drag point" (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 drag points only at a third of the length of the tangents, shall we?

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

```
[32]: 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,
```

(continues on next page)

(continued from previous page)

```
    x5.name,
]))
control_values_B
```

[32]:

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

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

[33]:
```
M_HtoB = NamedMatrix(r'{M_\text{H$\to$B}}', 4, 4)
```

[34]:
```
NamedMatrix(control_values_B.name, M_HtoB.name * control_values_H.name)
```

[34]:

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

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

[35]:
```
M_HtoB.expr = sp.Matrix([
    [expr.coeff(cv) for cv in control_values_H.name]
    for expr in control_values_B.expr])
M_HtoB
```

[35]:

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

[36]:
```
print(_.expr)
```

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

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

[37]:
```
M_BtoH = NamedMatrix(r'{M_\text{B$\to$H}}', M_HtoB.I.expr)
M_BtoH
```

[37]:

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

[38]:
```
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.

[39]:
```
M_B = NamedMatrix(r'{M_\text{B}}', M_H.name * M_BtoH.name)
M_B
```

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

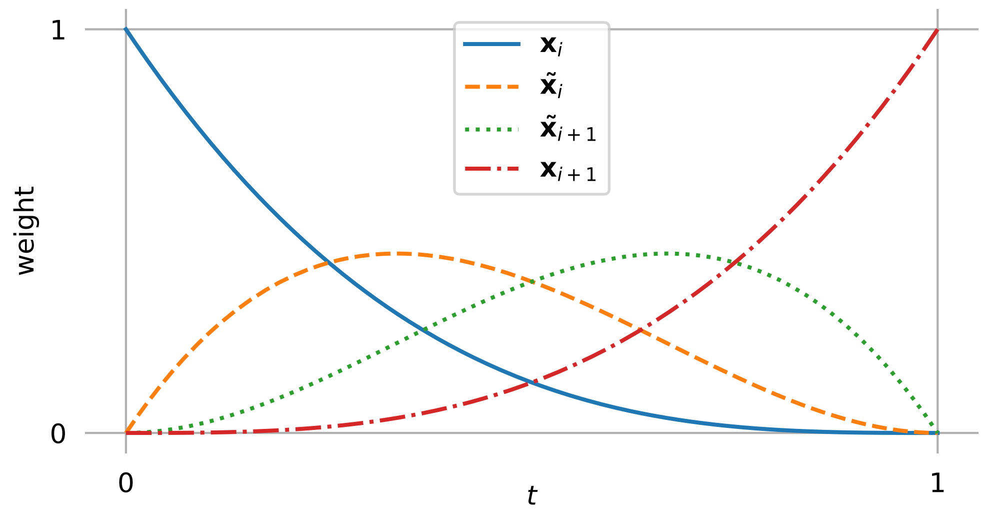
[40]:
```
M_B = M_B.subs_symbols(M_H, M_BtoH).doit()
M_B
```

[40]:
$$M_\text{B} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

[41]:
```
b_B = NamedMatrix(r'{b_\text{B}}', b_monomial * M_B.expr)
b_B.T
```

[41]:
$${b_\text{B}}^T = \begin{bmatrix} -t^3 + 3t^2 - 3t + 1 \\ 3t^3 - 6t^2 + 3t \\ -3t^3 + 3t^2 \\ t^3 \end{bmatrix}$$

[42]:
```
plot_basis(
    *b_B.expr,
    labels=sp.symbols('xbm_i xtildebm_i xtildebm_i+1 xbm_i+1'))
```



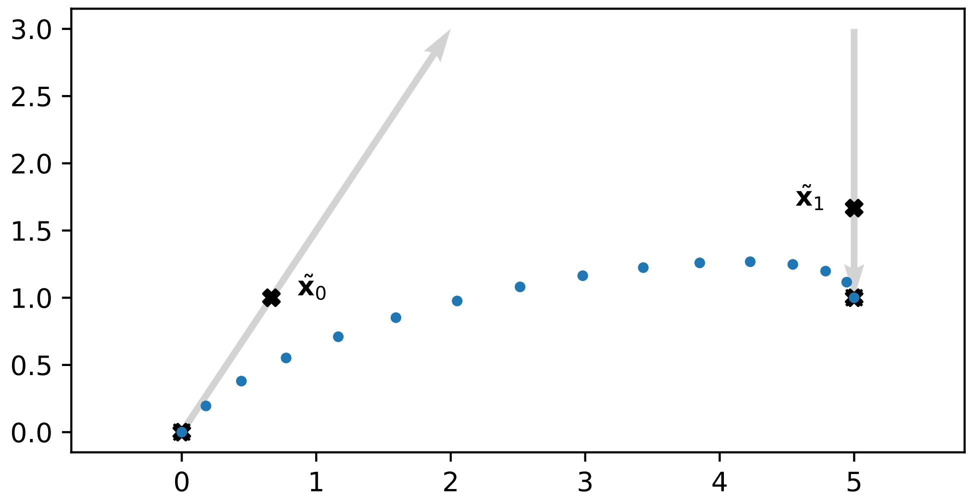
Those happen to be the cubic *Bernstein* polynomials and it turns out that we just invented *Bézier* curves! See *the section about Bézier splines* (page 134) for more information about them.

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

[43]:
```
points = sp.lambdify([], M_HtoB.expr)() @ [*vertices, *tangents]
```

[44]:
```
import matplotlib.pyplot as plt
```

[45]:
```
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` .......................

### B.2.4.3 Non-Uniform Cubic Hermite Splines

We have already derived *uniform cubic Hermite splines* (page 109), 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 $\Delta_i$).

The class *splines.CubicHermite* (page 283) is implemented using the second approach (because its parent class *splines.Monomial* (page 282) also uses the re-scaling approach).

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

[1]:
```
from pprint import pprint
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 again 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 $\boldsymbol{x}_{i+1}$, where $t_i \le t \le t_{i+1}$.

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

The next steps are very similar to what we did in the *uniform case* (page 109), 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))
```

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

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

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

$$\dot{\boldsymbol{x}}_5 = \frac{\boldsymbol{b}_4 + 2\boldsymbol{c}_4 + 3\boldsymbol{d}_4}{\Delta_4}$$

#### B.2.4.3.1 Basis Matrix

In contrast to the uniform case, where the same basis matrix could be used for all segments, here we need a different matrix for each segment.

[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]:

$$\begin{bmatrix}\boldsymbol{x}_4\\ \boldsymbol{x}_5\\ \dot{\boldsymbol{x}}_4\\ \dot{\boldsymbol{x}}_5\end{bmatrix} = {M_{\text{H},4}}^{-1}\begin{bmatrix}\boldsymbol{d}_4\\ \boldsymbol{c}_4\\ \boldsymbol{b}_4\\ \boldsymbol{a}_4\end{bmatrix}$$

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

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

[15]:

$$\begin{bmatrix}\boldsymbol{a}_4\\ \boldsymbol{a}_4 + \boldsymbol{b}_4 + \boldsymbol{c}_4 + \boldsymbol{d}_4\\ \frac{\boldsymbol{b}_4}{\Delta_4}\\ \frac{\boldsymbol{b}_4 + 2\boldsymbol{c}_4 + 3\boldsymbol{d}_4}{\Delta_4}\end{bmatrix} = {M_{\text{H},4}}^{-1}\begin{bmatrix}\boldsymbol{d}_4\\ \boldsymbol{c}_4\\ \boldsymbol{b}_4\\ \boldsymbol{a}_4\end{bmatrix}$$

[16]:
```
M_H.I = sp.Matrix([
    [expr.expand().coeff(c) for c in coefficients]
```

(continues on next page)

(continued from previous page)

```
    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]:
```
pprint(_.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]:
```
pprint(_.expr)
```

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

### B.2.4.3.2 Basis Polynomials

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

[20]:

$$
{b_{\text{H},4}}^{T} = \begin{bmatrix} (t-1)^2 \cdot (2t+1) \\ t^2 \cdot (-2t+3) \\ \Delta_4 t\,(t-1)^2 \\ \Delta_4 t^2\,(t-1) \end{bmatrix}
$$

Those are the *non-uniform* (cubic) Hermite basis functions. Not surprisingly, they are different for each segment, because generally the values $\Delta_i$ are different in the non-uniform case.

### B.2.4.3.3 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.

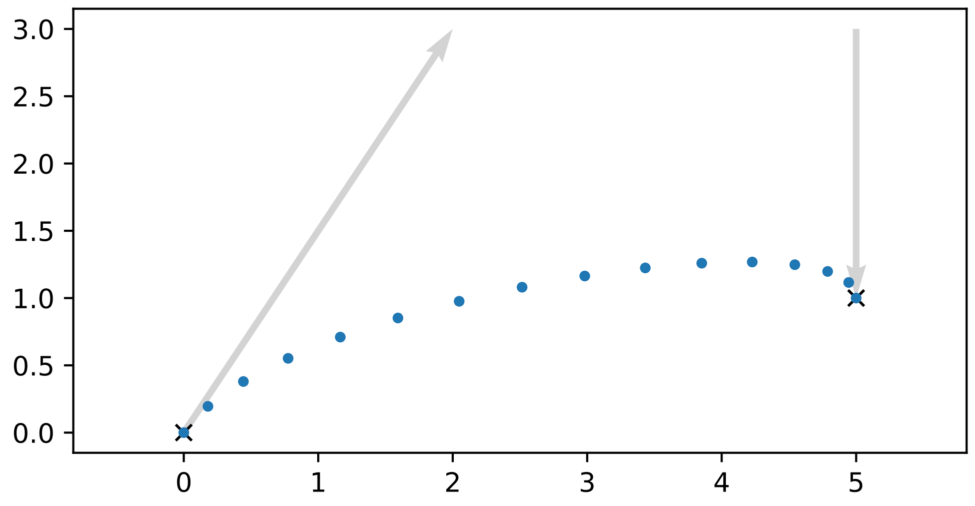
[23]:
```
class HermiteSegment:

    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)
```

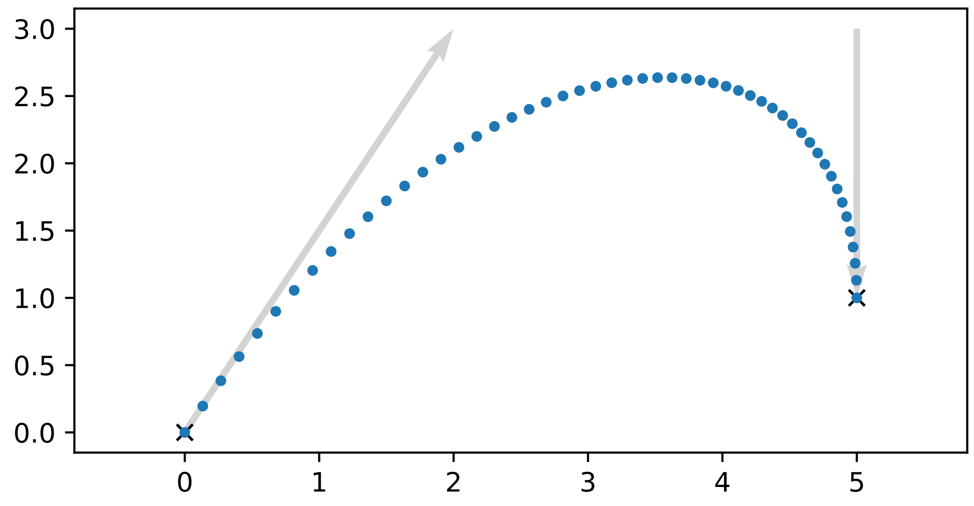
[26]:
```
plot_spline_2d(s1, chords=False)
plot_tangents_2d(tangents, vertices)
```



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)
```



### B.2.4.3.4 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 only applies 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 109), 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 $\Delta_4$. That means if we re-scale the parameter *and* multiply the given tangent vectors by $\Delta_i$, 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
```

To make a 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 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[19] (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 $\Delta_i$! If we want to maintain the original lengths of our tangent vectors, we can simply scale them by $\Delta_i$ beforehand.

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

### B.2.5 Natural Splines

Sometimes simply called (cubic) spline interpolation[20], a *natural* spline is modelled after a drawing tool called spline[21], which is made from a thin piece of elastic material like wood or metal.

[19] https://en.wikipedia.org/wiki/Chain_rule
[20] https://en.wikipedia.org/wiki/Spline_interpolation
[21] https://en.wiktionary.org/wiki/spline

A Python implementation is available in the class *splines.Natural* (page 285). Alternatively, the CubicSpline[22] class from SciPy can be used.

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

#### B.2.5.1 Properties of Natural Splines

The most important property of (cubic) natural splines is that they are $C^2$ continuous, which means that the second derivatives match at the transitions between segments. On top of that, they are *interpolating*, which means that the curve passes through the given control points.

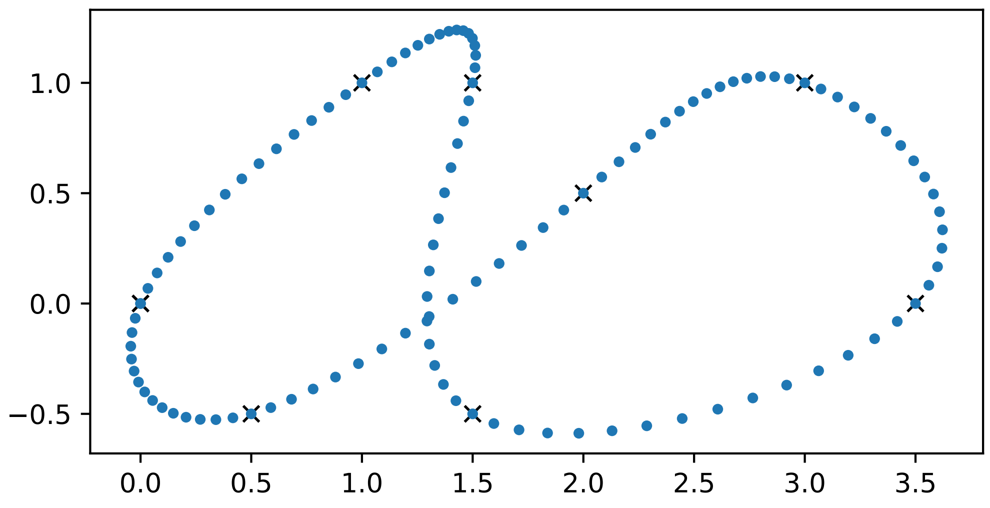
[1]:
```
import splines
import matplotlib.pyplot as plt
```

[2]:
```
vertices = [
    (0, 0),
    (1, 1),
    (1.5, 1),
    (1.5, -0.5),
    (3.5, 0),
    (3, 1),
    (2, 0.5),
    (0.5, -0.5),
]
```

To show an example, we use the class *splines.Natural* (page 285) and a plotting function from `helper.py`:

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

[4]:
```
plot_spline_2d(
    splines.Natural(vertices, endconditions='closed'),
    chords=False)
```

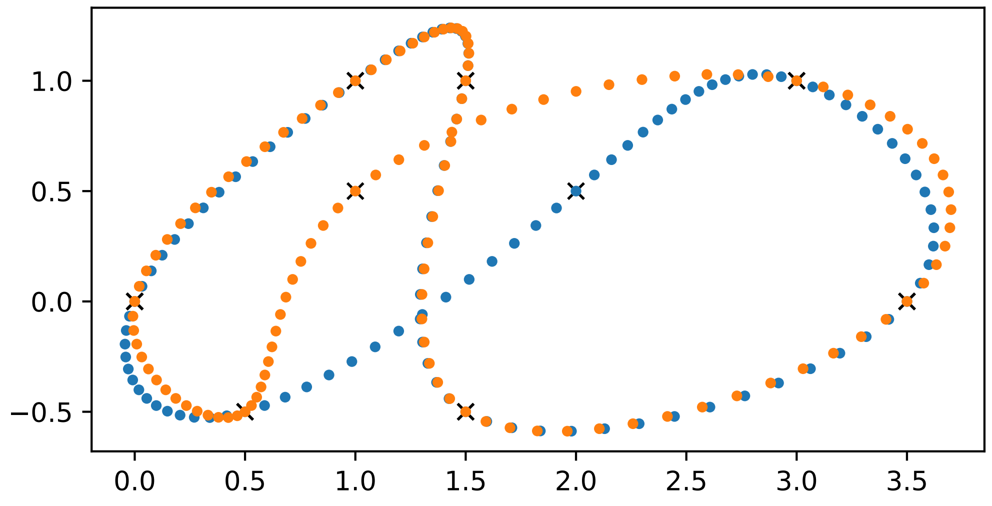


[22] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.CubicSpline.html

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

[5]:
```
modified_vertices = vertices.copy()
modified_vertices[6] = 1, 0.5
```

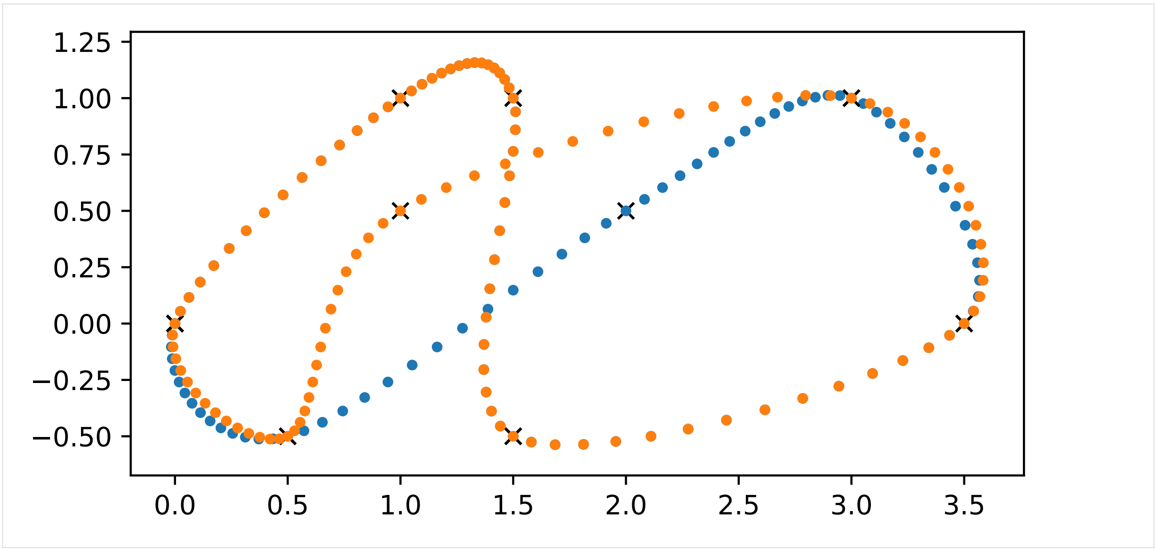
[6]:
```
plot_spline_2d(
    splines.Natural(vertices, endconditions='closed'),
    chords=False)
plot_spline_2d(
    splines.Natural(modified_vertices, endconditions='closed'),
    chords=False)
```



We can see that there are deviations in all segments, not only close to the modified vertex.

For comparison, we can use the same vertices to create a uniform cubic *Catmull–Rom spline* (page 155) using the *splines.CatmullRom* (page 283) class:

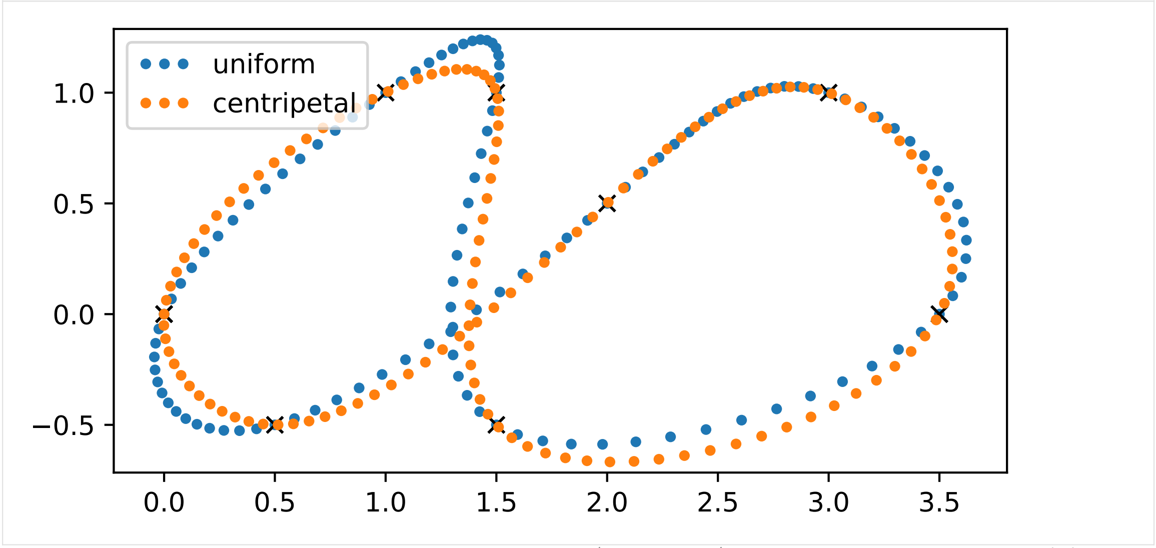
[7]:
```
plot_spline_2d(
    splines.CatmullRom(vertices, endconditions='closed'),
    chords=False)
plot_spline_2d(
    splines.CatmullRom(modified_vertices, endconditions='closed'),
    chords=False)
```



Here we can see that two segments before and two segments after the modified vertex are affected, but the rest of the segments remain unchanged.

Although this is typically only used with Catmull–Rom splines, we can also use *centripetal parameterization* (page 163) for a natural spline:

[8]:
```
plot_spline_2d(
    splines.Natural(vertices, endconditions='closed'),
    chords=False, label='uniform')
plot_spline_2d(
    splines.Natural(vertices, endconditions='closed', alpha=0.5),
    chords=False, label='centripetal')
plt.legend(numpoints=3);
```



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

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

### B.2.5.2 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 131).

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

We import some helpers from `utility.py`:

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

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

To get started, let's look at two neighboring segments: Let's say the fourth segment, from $\boldsymbol{x}_3$ to $\boldsymbol{x}_4$, defined by the polynomial $\boldsymbol{p}_3$, and the fifth segment, from $\boldsymbol{x}_4$ to $\boldsymbol{x}_5$, defined by the polynomial $\boldsymbol{p}_4$. In both cases, we use $0 \le t \le 1$.

[4]:
```
coefficients3 = sp.symbols('a:dbm3')[::-1]
coefficients4 = sp.symbols('a:dbm4')[::-1]
```

We apply these coefficients to the *monomial basis* (page 88) ...

[5]:
```
b_monomial = t**3, t**2, t, 1
```

... to define the two polynomials ...

[6]:
```
p3 = NamedExpression('pbm3', dotproduct(b_monomial, coefficients3))
p4 = NamedExpression('pbm4', dotproduct(b_monomial, coefficients4))
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'),
```

(continues on next page)

(continued from previous page)

```
]
display(*equations)
```

$x_3 = a_3$

$x_4 = a_3 + b_3 + c_3 + d_3$

$x_4 = a_4$

$x_5 = a_4 + b_4 + c_4 + d_4$

$\dot{x}_3 = b_3$

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

$\dot{x}_4 = b_4$

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

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

[9]:
```
coefficients = sp.solve(equations, coefficients3 + coefficients4)
for c, e in coefficients.items():
    display(NamedExpression(c, e))
```

$a_3 = x_3$

$a_4 = x_4$

$b_3 = \dot{x}_3$

$b_4 = \dot{x}_4$

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

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

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

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

So far, this is the same as we have done in *the notebook about uniform Hermite splines* (page 109). 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)
```

$\frac{d^2}{dt^2}p_3 = 6d_3t + 2c_3$

$\frac{d^2}{dt^2}p_4 = 6d_4t + 2c_4$

... 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:

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

### B.2.5.2.1 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 285) 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 207), 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 $D_{\text{begin}}$ and $D_{\text{end}}$ 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:

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

#### B.2.5.2.2 Solving the System of Equations

The matrices above are *tridiagonal* and can therefore be solved efficiently with a tridiagonal matrix algorithm[23]. The class *splines.Natural* (page 285), however, is not very concerned about efficiency and simply uses NumPy's linalg.solve()[24] 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 ........................

### B.2.5.3 Non-Uniform Natural Splines

The derivation is similar to *the uniform case* (page 127), but this time the parameter intervals can have arbitrary values.

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

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

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

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

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

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

[5]:
```
b_monomial = t**3, t**2, t, 1
```

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

[7]:
```
p3 = NamedExpression(
    'pbm3',
    dotproduct(b_monomial, coefficients3).subs(t, (t - t3)/(t4 - t3)))
p4 = NamedExpression(
    'pbm4',
    dotproduct(b_monomial, coefficients4).subs(t, (t - t4)/(t5 - t4)))
display(p3, p4)
```

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

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

[8]:
```
pd3 = p3.diff(t)
pd4 = p4.diff(t)
display(pd3, pd4)
```

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

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

[9]:
```
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 $t_i$:

[10]:
```
deltas = {
    t3: 0,
    t4: sp.Symbol('Delta3'),
    t5: sp.Symbol('Delta3') + sp.Symbol('Delta4'),
}
```

[11]:
```
for e in equations:
    display(e.subs(deltas))
```

$x_3 = a_3$

$x_4 = a_3 + b_3 + c_3 + d_3$

$x_4 = a_4$

$x_5 = a_4 + b_4 + c_4 + d_4$

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

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

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

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

[12]:
```
coefficients = sp.solve(equations, coefficients3 + coefficients4)
```

[13]:
```
for c, e in coefficients.items():
    display(NamedExpression(c, e.factor().subs(deltas).simplify()))
```

$a_3 = x_3$

$a_4 = x_4$

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

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

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

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

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

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

[14]:
```
pdd3 = pd3.diff(t)
pdd4 = pd4.diff(t)
display(pdd3, pdd4)
```

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

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

[15]:
```
sp.Eq(pdd3.expr.subs(t, t4), pdd4.expr.subs(t, t4))
```

[15]:
$$\frac{3d_3 (-2t_3 + 2t_4)}{(-t_3 + t_4)^3} + \frac{2c_3}{(-t_3 + t_4)^2} = \frac{2c_4}{(-t_4 + t_5)^2}$$

[16]:
```
_.subs(coefficients).subs(deltas).simplify()
```

[16]:
$$\frac{2\left(\Delta_3\dot{x}_3 + 2\Delta_3\dot{x}_4 + 3x_3 - 3x_4\right)}{\Delta_3^2} = \frac{2\left(-2\Delta_4\dot{x}_4 - \Delta_4\dot{x}_5 - 3x_4 + 3x_5\right)}{\Delta_4^2}$$

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 instructive than in the uniform case.

#### B.2.5.3.1 End Conditions

Like in the *uniform case* (page 130), we can come up with a few end conditions in order to define the missing matrix rows.

The Python class *splines.Natural* (page 285) uses "natural" end conditions by default.

"Natural" end conditions are derived in *a separate notebook* (page 207), 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 130).

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

### B.2.6 Bézier Splines

Named after Pierre Bézier[25], Bézier curves are defined by means of Bernstein polynomials[26] (Farouki 2012), which are named after Sergei Bernstein[27]. A popular method to evaluate Bézier curves at given parameter values is *De Casteljau's algorithm* (page 136). A very good online resource with many interactive examples is the website `https://pomax.github.io/bezierinfo/`.

Bézier *splines* are composed of Bézier *curve* segments.

A Python implementation is available in the class *splines.Bernstein* (page 282).

[25] `https://en.wikipedia.org/wiki/Pierre_Bézier`
[26] `https://en.wikipedia.org/wiki/Bernstein_polynomial`
[27] `https://en.wikipedia.org/wiki/Sergei_Bernstein`

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

### B.2.6.1 Properties of Bézier Splines

The terms *Bézier spline* and *Bézier curve* are sometimes used interchangeably for 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 lie 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.

We use the term *Bézier curve* for the former and *Bézier spline* for the latter. 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]:
```
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
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.

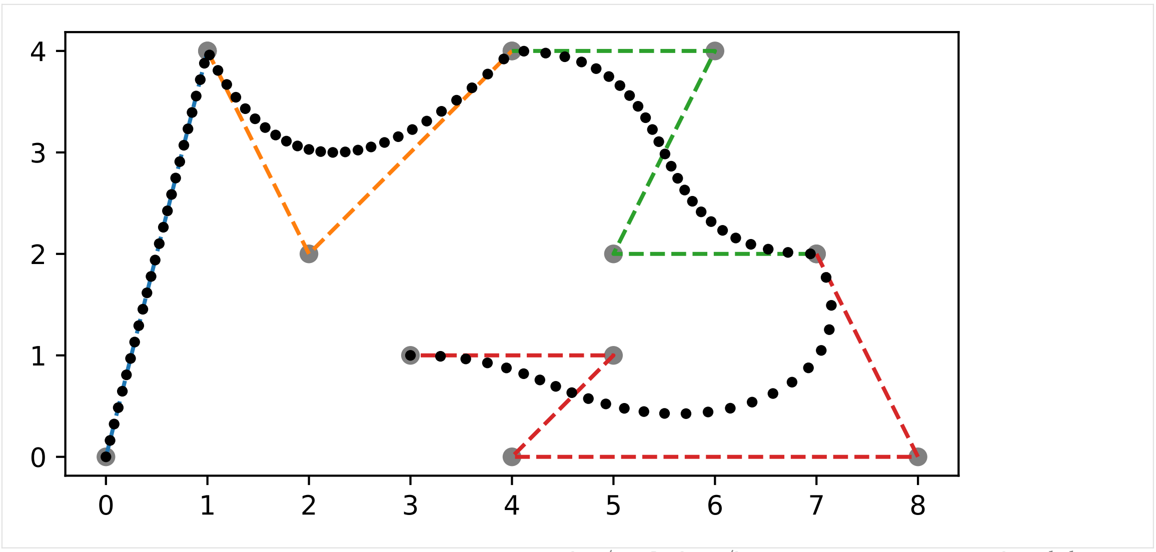
[3]:
```
control_points = [
    [(0, 0), (1, 4)],
    [(1, 4), (2, 2), (4, 4)],
    [(4, 4), (6, 4), (5, 2), (7, 2)],
    [(7, 2), (8, 0), (4, 0), (5, 1), (3, 1)],
]
```

We are using the class *splines.Bernstein* (page 282) to construct a Bézier spline 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 .......................

### B.2.6.2 De Casteljau's Algorithm

There are several ways that lead to Bézier curves, one (but only for cubic curves) was already shown in *the notebook about Hermite curves* (page 115). In this notebook, we will derive Bézier curves of arbitrary polynomial degree utilizing De Casteljau's algorithm[28].

#### B.2.6.2.1 Preparations

[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(
```

(continues on next page)

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

(continued from previous page)

```
        *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

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
```

#### B.2.6.2.2 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, $x_0$ and $x_1$, we can set up a linear equation:

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

Another way to write the same thing is like this:

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

where in both cases $0 \le t \le 1$. These linear interpolations are sometimes also called *affine combinations*. 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
```

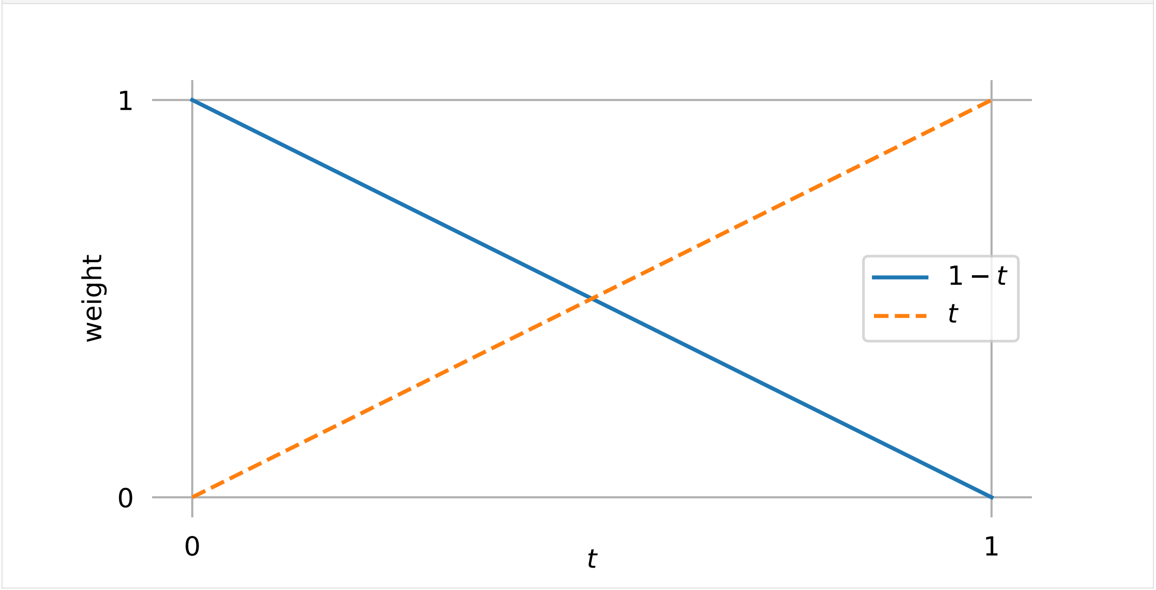
[7]: $\boldsymbol{p}_{0,1} = t\boldsymbol{x}_1 + \boldsymbol{x}_0 \cdot (1 - t)$

[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[29] for degree 1. It doesn't get much more interesting if we plot them:

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



If you want to convert this to coefficients for the *monomial basis* (page 88) $[t, 1]$ 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]: $\begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \end{bmatrix}$

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

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

[12]:
$$\begin{bmatrix} -x_0 + x_1 \\ x_0 \end{bmatrix}$$

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

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

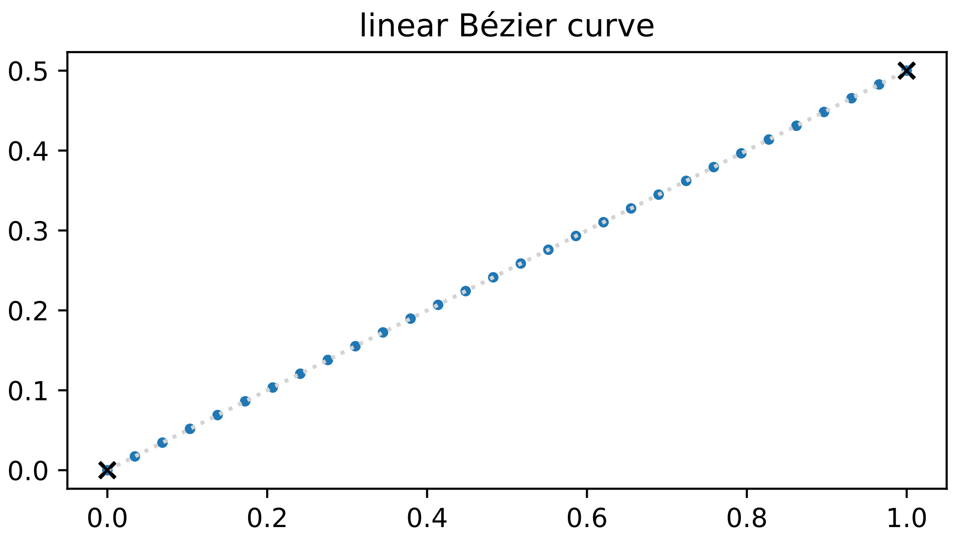
[13]:
$$\left(M_{\mathrm{B}}^{(1)}\right)^{-1} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

Anyhow, 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!

### B.2.6.2.3 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]: $p_{0,1} = tx_1 + x_0 \cdot (1 - t)$

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

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

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

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

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

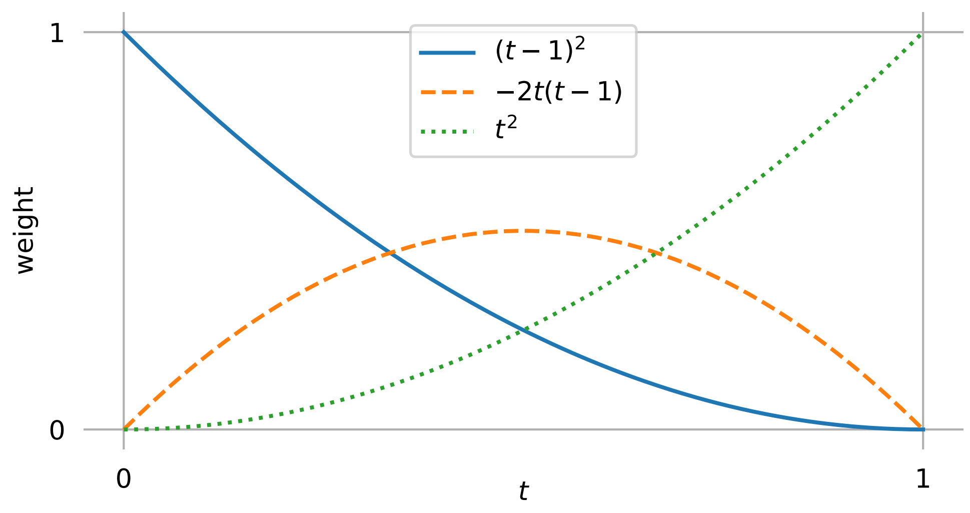
[20]: $p_{0,2} = t\,(tx_2 + x_1 \cdot (1 - t)) + (1 - t)\,(tx_1 + x_0 \cdot (1 - t))$

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

```
[21]: 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]: plot_basis(*b2)
```



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

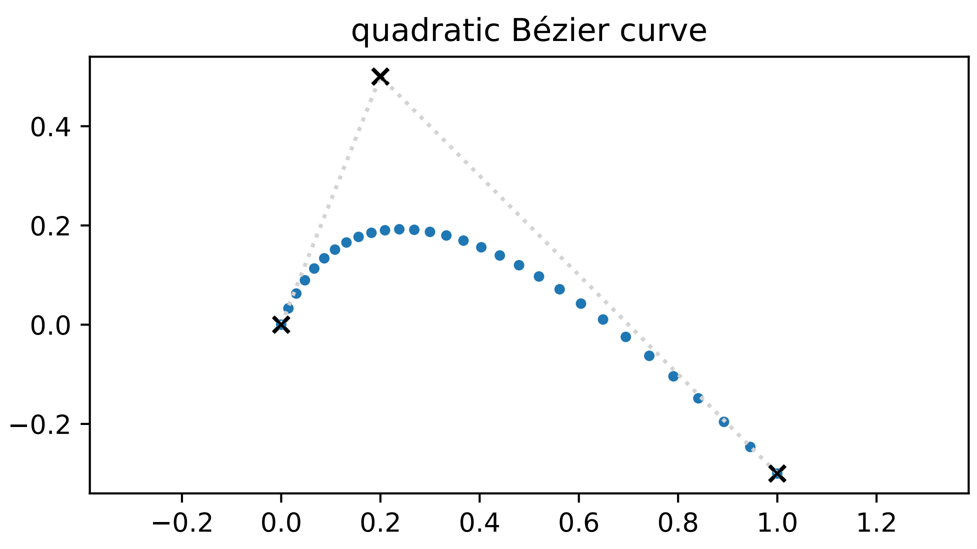
[24]:
```
M_B2.I
```

[24]:
$$\left(M_B^{(2)}\right)^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & \frac{1}{2} & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

[25]:
```
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]:
```
points = [
    (0, 0),
    (0.2, 0.5),
    (1, -0.3),
]
```

[27]:
```
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]:
$$\left.\frac{d}{dt}p_{0,2}\right|_{t=0} = -2x_0 + 2x_1$$

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

[31]:
$$\left.\frac{d}{dt}p_{0,2}\right|_{t=1} = -2x_1 + 2x_2$$

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 $x_1$ to $x_2$, respectively. The length of the tangent vectors is twice the length of those lines.

You might have already seen this coming, but it turns out that the last line in De Casteljau's algorithm ($p_{1,2}(t) - p_{0,1}(t)$ 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.

#### B.2.6.2.4 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 $x_3$, which we will derive in the following.

We still need the combination of $x_2$ and $x_3$ ...

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

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

... which we are using to calculate the combination of $x_1$, $x_2$ and $x_3$ ...

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

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

... which we need for the combination of $x_0$, $x_1$, $x_2$ and $x_3$:

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

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

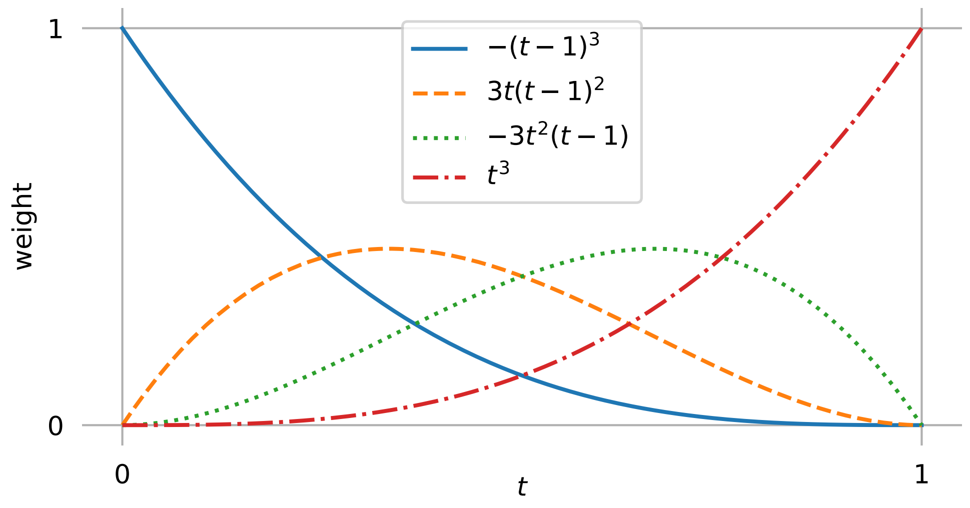
This leads to the cubic Bernstein bases:

[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 115).

```
[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
```

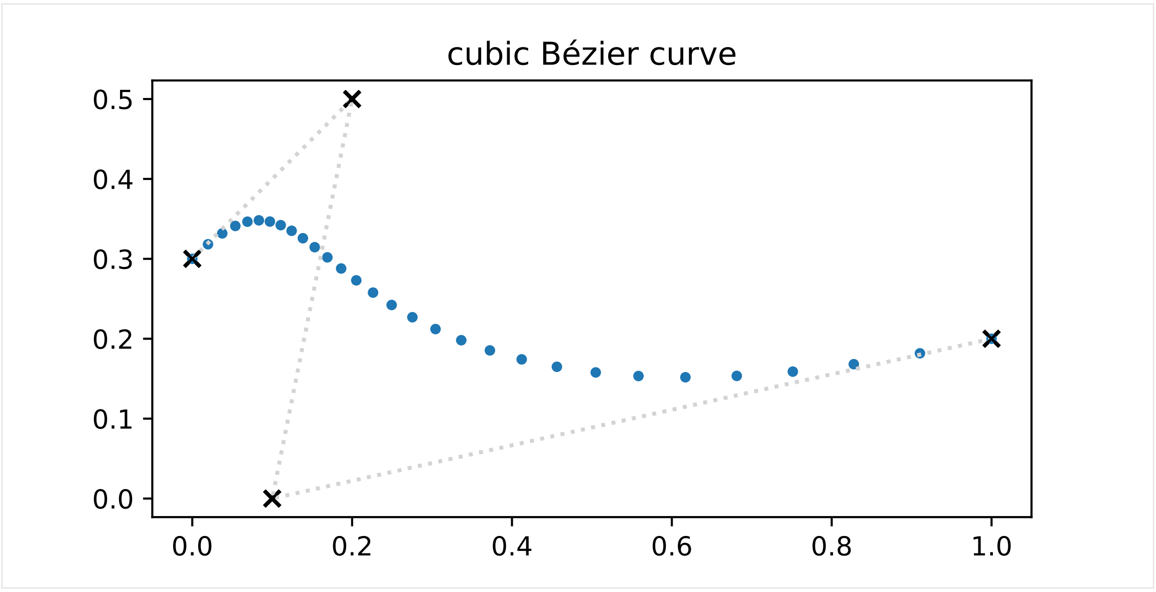
[39]:

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

```
[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 115) and to *construct uniform Catmull–Rom splines using Bézier segments* (page 174).

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

```
[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]: $\boldsymbol{p}_{0,3}\Big|_{t=0} = \boldsymbol{x}_0$

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

[49]: $\boldsymbol{p}_{0,3}\Big|_{t=1} = \boldsymbol{x}_3$

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 115).

#### B.2.6.2.5 Degree 4 (Quartic)

By now you know the drill, let's 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))
```

The resulting expression for $p_{0,4}(t)$ is quite long and unwieldy (and frankly, quite boring as well), so we are not showing it here.

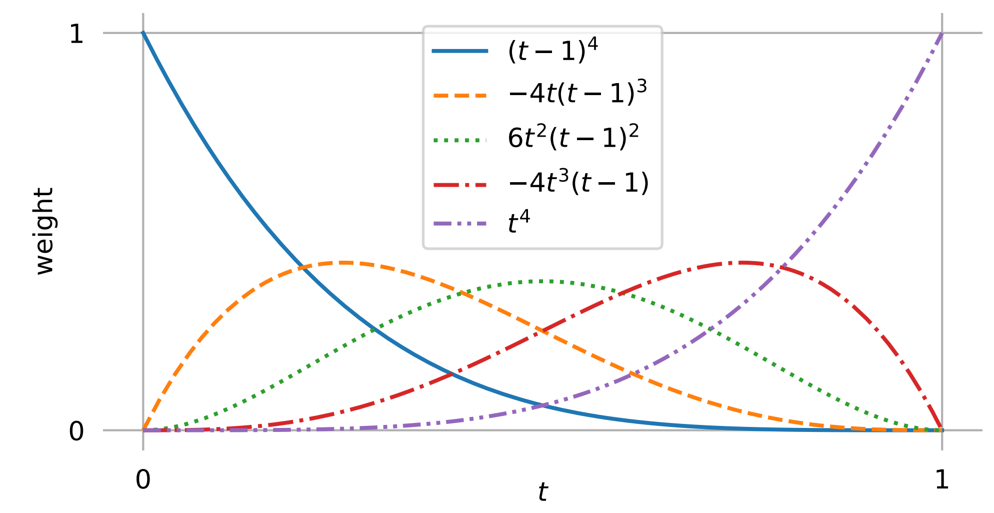
```
[53]: #p04
```

Instead, we are using it immediately to extract the Bernstein bases:

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

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

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



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

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

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

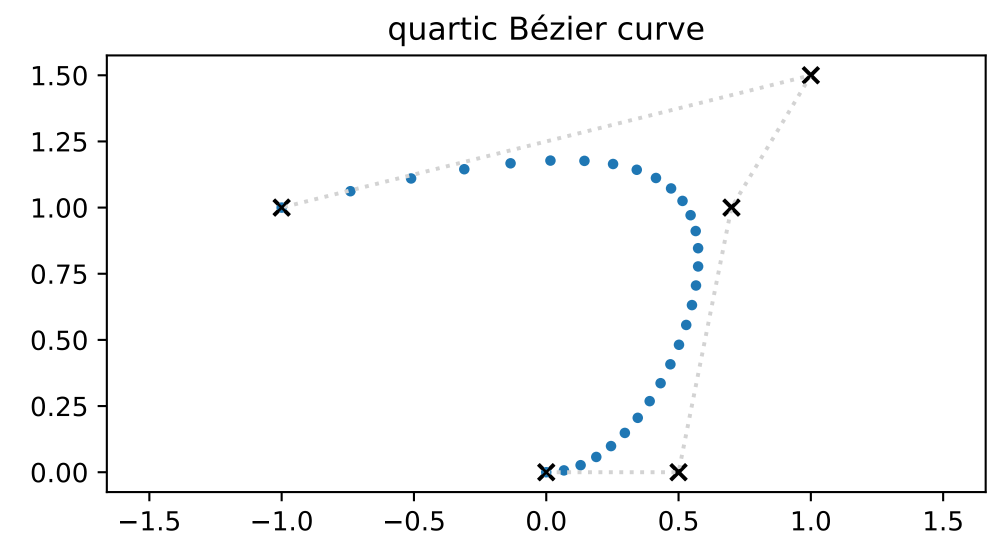
[57]:

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

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

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

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



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

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

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

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

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

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

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

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

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

By now it shouldn't be surprising that the tangent vector at the beginning and end of the curve is parallel to the line from $\boldsymbol{x}_0$ to $\boldsymbol{x}_1$ and from $\boldsymbol{x}_3$ to $\boldsymbol{x}_4$, respectively. The length of the tangent vectors is four times the length of those lines. The last line in De Casteljau's algorithm ($\boldsymbol{p}_{1,4}(t) - \boldsymbol{p}_{0,3}(t)$ in this case) is exactly a fourth of the tangent vector (at any given $t \in [0,1]$):

[65]:
```
assert (v04.expr - 4 * (p14.expr - p03.expr)).simplify() == 0
```

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

#### B.2.6.2.6 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[30] $\binom{n}{i} = \frac{n!}{i!(n-i)!}$):

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

This is used in the Python class *splines.Bernstein* (page 282).

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

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

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

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

### B.2.6.3 Non-Uniform (Cubic) 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 136).

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

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
```

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)))
```

#### B.2.6.3.1 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]: $$\left.\frac{d}{dt}p_4\right|_{t=t_4} = -\frac{3x_4}{-t_4+t_5} + \frac{3\tilde{x}_4^{(+)}}{-t_4+t_5}$$

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

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

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

Similarly, the incoming tangent vector at $\boldsymbol{x}_5$ ...

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

[10]:
$$\frac{d}{dt}\boldsymbol{p}_4\bigg|_{t=t_5} = \frac{3\boldsymbol{x}_5}{-t_4 + t_5} - \frac{3\tilde{\boldsymbol{x}}_5^{(-)}}{-t_4 + t_5}$$

... can be generalized to

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

This is similar to the *uniform case* (page 144), the tangent vectors are just divided by the parameter interval.

#### B.2.6.3.2 Control Points From Tangent Vectors

If the tangent vectors are given in the first place – i.e. when a non-uniform *Hermite spline* (page 105) is given, the cubic Bézier control points can be calculated like this:

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

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

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

### B.2.7 Quadrangle Interpolation

This doesn't seem to be a very popular type of spline. We are mainly mentioning it because it is the starting point for interpolating rotations with *Spherical Quadrangle Interpolation* (*Squad*) (page 268).

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

As usual, we import some helpers from `utility.py` and `helper.py`:

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

Let's start – as we have done before – by looking at the fifth segment of a spline, between $\boldsymbol{x}_4$ and $\boldsymbol{x}_5$. It will be referred to as $\boldsymbol{p}_4(t)$, where $0 \le t \le 1$.

```
[3]: x4, x5 = sp.symbols('xbm4:6')
```

Boehm (1982) mentions (on page 203) so-called *quadrangle points*:

```
[4]: x4bar = sp.symbols('xbarbm4^(+)')
     x5bar = sp.symbols('xbarbm5^(-)')
     x4bar, x5bar
```

[4]: $\left(\bar{x}_4^{(+)},\ \bar{x}_5^{(-)}\right)$

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

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

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

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

Boehm (1982) also mentions (on page 210) a peculiar algorithm to construct the spline segment. In a first step, a linear interpolation between the start and end point is done, as well as a linear interpolation between the two quadrangle points. The two resulting points are then interpolated again in a second step. However, the last interpolation does not happen along a straight line, but along a parabola defined by the expression $2t(1-t)$:

```
[7]: p4 = NamedExpression(
         'pbm4',
         lerp(lerp(x4, x5, t), lerp(x4bar, x5bar, t), 2 * t * (1 - t)))
```

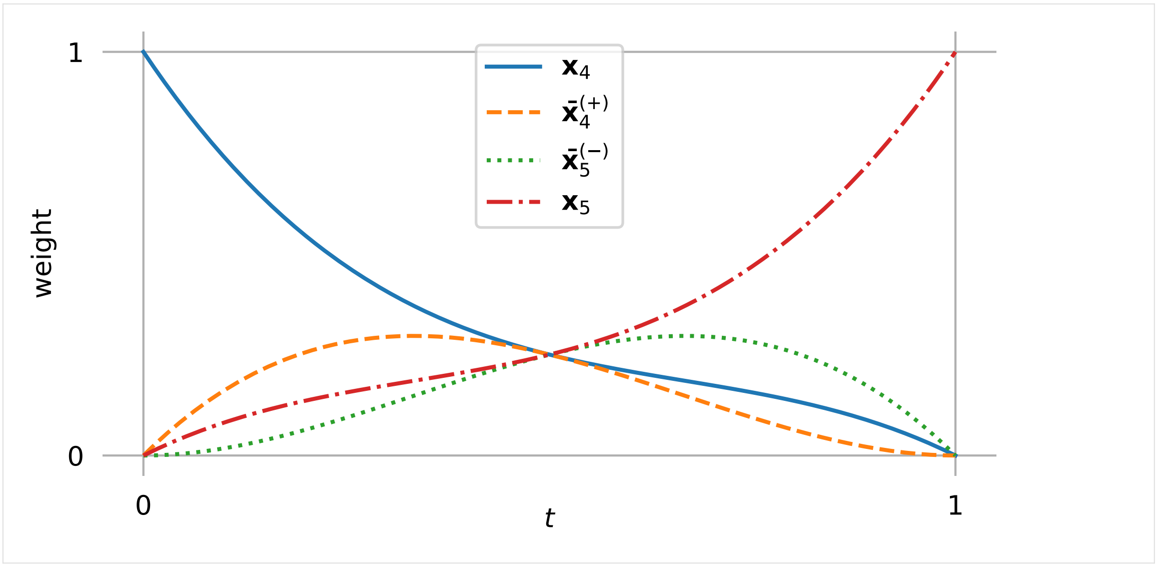
This leads to a cubic polynomial. The following steps are very similar to what we did for *cubic Bézier curves* (page 142).

#### B.2.7.1 Basis Polynomials

```
[8]: b = [p4.expr.expand().coeff(x) for x in (x4, x4bar, x5bar, x5)]
     b
```

[8]: $\left[-2t^3+4t^2-3t+1,\ 2t^3-4t^2+2t,\ -2t^3+2t^2,\ 2t^3-2t^2+t\right]$

```
[9]: plot_basis(*b, labels=(x4, x4bar, x5bar, x5))
```



### B.2.7.2 Basis Matrix

[10]:
```
M_Q = NamedMatrix(
    r'{M_\text{Q}}',
    sp.Matrix([[c.coeff(x) for x in (x4, x4bar, x5bar, x5)]
               for c in p4.as_poly(t).all_coeffs()]))
M_Q
```

[10]:

$$M_{\mathrm{Q}} = \begin{bmatrix} 2 & -2 & 2 & -2 \\ -4 & 4 & -2 & 2 \\ 3 & -2 & 0 & -1 \\ -1 & 0 & 0 & 0 \end{bmatrix}$$

[11]:
```
M_Q.I
```

[11]:

$$M_{\mathrm{Q}}{}^{-1} = \begin{bmatrix} 0 & 0 & 0 & -1 \\ \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ 0 & -\frac{1}{2} & -1 & -1 \\ -1 & -1 & -1 & -1 \end{bmatrix}$$

### B.2.7.3 Tangent Vectors

[12]:
```
pd4 = p4.diff(t)
```

[13]:
```
xd4 = pd4.evaluated_at(t, 0)
xd4
```

[13]:

$$\left.\frac{d}{dt}p_4\right|_{t=0} = 2\bar{x}_4^{(+)} - 3x_4 + x_5$$

[14]:
```
xd5 = pd4.evaluated_at(t, 1)
xd5
```

[14]:

$$\left.\frac{d}{dt}p_4\right|_{t=1} = -2\bar{x}_5^{(-)} - x_4 + 3x_5$$

This can be generalized to:

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

#### B.2.7.4 Quadrangle to Hermite Control Values

```
[15]: M_QtoH = NamedMatrix(
          r'{M_\text{Q$\to$H}}',
          sp.Matrix([[expr.coeff(cv) for cv in [x4, x4bar, x5bar, x5]]
                     for expr in [
                         x4,
                         x5,
                         xd4.expr,
                         xd5.expr]]))
      M_QtoH
```

[15]:

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

```
[16]: M_QtoH.I.pull_out(sp.S.One / 2)
```

[16]:

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

#### B.2.7.5 Quadrangle to Bézier Control Points

Since we already know the tangent vectors, it is easy to find the Bézier control points, as we have already shown in the notebook about *uniform Hermite splines* (page 115).

```
[17]: x4tilde = NamedExpression('xtildebm4^(+)', x4 + xd4.expr / 3)
      x4tilde
```

[17]:

$$\tilde{x}_4^{(+)} = \frac{2\bar{x}_4^{(+)}}{3} + \frac{x_5}{3}$$

```
[18]: x5tilde = NamedExpression('xtildebm5^(-)', x5 - xd5.expr / 3)
      x5tilde
```

[18]:

$$\tilde{x}_5^{(-)} = \frac{2\bar{x}_5^{(-)}}{3} + \frac{x_4}{3}$$

```
[19]: M_QtoB = NamedMatrix(
          r'{M_\text{Q$\to$B}}',
          sp.Matrix([[expr.coeff(cv) for cv in (x4, x4bar, x5bar, x5)]
                     for expr in [
                         x4,
                         x4tilde.expr,
                         x5tilde.expr,
                         x5]]))
      M_QtoB.pull_out(sp.S.One / 3)
```

[19]:

$$M_{\text{Q}\to\text{B}} = \frac{1}{3}\begin{bmatrix}3 & 0 & 0 & 0\\0 & 2 & 0 & 1\\1 & 0 & 2 & 0\\0 & 0 & 0 & 3\end{bmatrix}$$

```
[20]: M_QtoB.I.pull_out(sp.S.One / 2)
```

[20]:

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

The inverse matrix can be used for converting from Bézier control points to quadrangle points:

```
[21]: NamedMatrix(
          sp.Matrix([x4, x4bar, x5bar, x5]),
          M_QtoB.I.expr * sp.Matrix([x4, x4tilde.name, x5tilde.name, x5]))
```

[21]:

$$\begin{bmatrix}x_4\\\bar{x}_4^{(+)}\\\bar{x}_5^{(-)}\\x_5\end{bmatrix} = \begin{bmatrix}x_4\\-\frac{x_5}{2}+\frac{3\tilde{x}_4^{(+)}}{2}\\-\frac{x_4}{2}+\frac{3\tilde{x}_5^{(-)}}{2}\\x_5\end{bmatrix}$$

We can generalize the equations for the outgoing and incoming quadrangle points:

$$\bar{x}_i^{(+)} = \frac{3}{2}\tilde{x}_i^{(+)} - \frac{1}{2}x_{i+1}$$

$$\bar{x}_i^{(-)} = \frac{3}{2}\tilde{x}_i^{(-)} - \frac{1}{2}x_{i-1}$$

The two equations are also shown by Boehm (1982) on page 203.

### B.2.7.6 Non-Uniform Parameterization

Just like *cubic Bézier splines* (page 148), the shape of a segment (i.e. the image[31]) is fully defined by its four control points. Re-scaling the parameter does not change the shape, but it changes the speed and therefore the tangent vectors.

[31] https://en.wikipedia.org/wiki/Image_(mathematics)

[22]:
```
t4, t5 = sp.symbols('t4:6')
```

[23]:
```
p4nu = p4.subs(t, (t - t4) / (t5 - t4)).with_name(
    r'\boldsymbol{p}_\text{4,non-uniform}')
```

[24]:
```
pd4nu = p4nu.diff(t)
```

[25]:
```
pd4nu.evaluated_at(t, t4)
```

[25]:
$$\frac{d}{dt}\boldsymbol{p}_{\text{4,non-uniform}}\bigg|_{t=t_4} = \frac{2\bar{\boldsymbol{x}}_4^{(+)}}{-t_4+t_5} - \frac{3\boldsymbol{x}_4}{-t_4+t_5} + \frac{\boldsymbol{x}_5}{-t_4+t_5}$$

[26]:
```
pd4nu.evaluated_at(t, t5)
```

[26]:
$$\frac{d}{dt}\boldsymbol{p}_{\text{4,non-uniform}}\bigg|_{t=t_5} = -\frac{2\bar{\boldsymbol{x}}_5^{(-)}}{-t_4+t_5} - \frac{\boldsymbol{x}_4}{-t_4+t_5} + \frac{3\boldsymbol{x}_5}{-t_4+t_5}$$

This can be generalized to:

$$\begin{aligned}\dot{\boldsymbol{x}}^{(+)}_{i,\text{non-uniform}} &= \frac{2\bar{\boldsymbol{x}}_i^{(+)} - 3\boldsymbol{x}_i + \boldsymbol{x}_{i+1}}{\Delta_i}\\ \dot{\boldsymbol{x}}^{(-)}_{i,\text{non-uniform}} &= -\frac{2\bar{\boldsymbol{x}}_i^{(-)} - 3\boldsymbol{x}_i + \boldsymbol{x}_{i-1}}{\Delta_{i-1}}\end{aligned}$$

.................................................................. doc/euclidean/quadrangle.ipynb ends here.

### B.2.8 Catmull–Rom Splines

What is nowadays known as *Catmull–Rom spline* is a specific member of a whole family of splines introduced by Catmull and Rom (1974). 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 by Overhauser (1968) – are not the same!

A Python implementation of Catmull–Rom splines is available in the class *splines.CatmullRom* (page 283).

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

#### B.2.8.1 Properties of Catmull–Rom Splines

Catmull and Rom (1974) present 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 spline is very popular because they are very easy to use. Only a sequence of control points has to be specified, the corresponding tangents are calculated automatically from the given points. Using those tangents, the spline can be implemented using cubic

*Hermite splines* (page 105). Alternatively, spline values can be directly calculated with the *Barry–Goldman algorithm* (page 181).

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 283) to create both uniform and non-uniform splines. Only closed splines are shown, other *end conditions* (page 207) 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 281) module ...

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

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

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

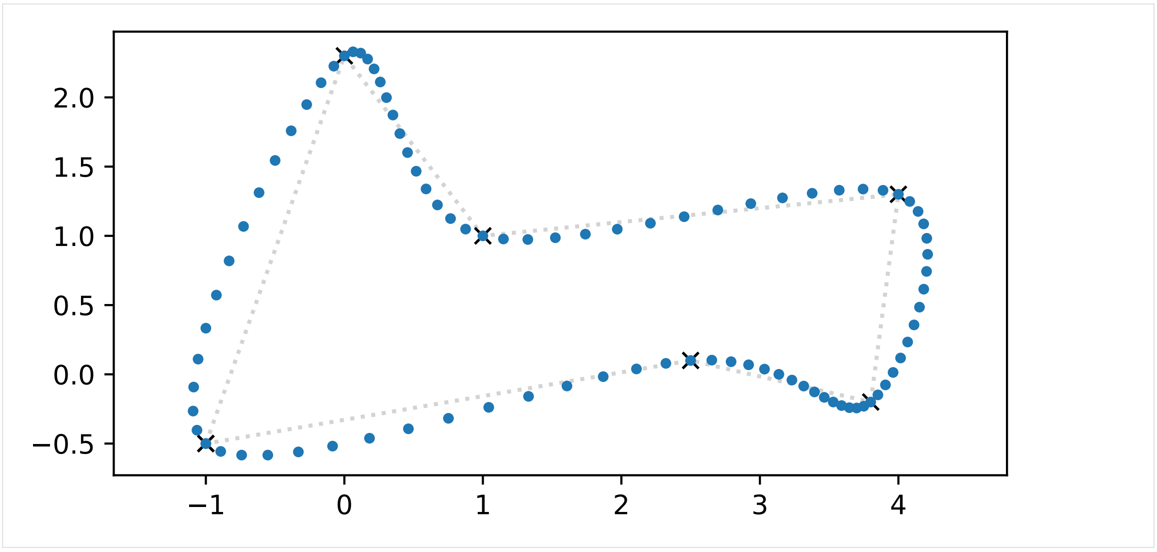
Let's choose a few points for an example:

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

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



### B.2.8.1.1 Tangent Vectors

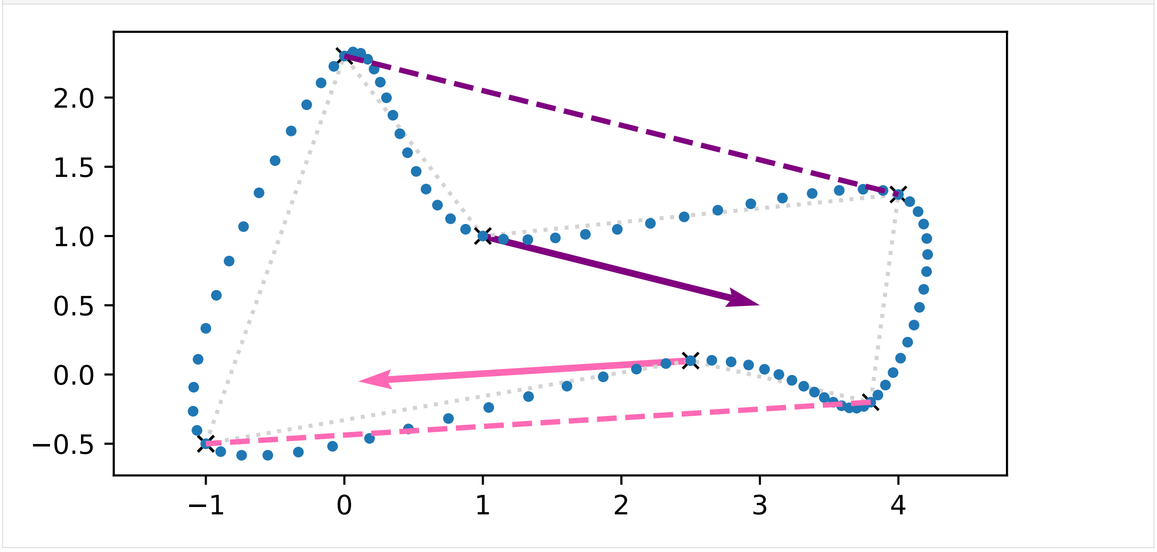
In the uniform case, the tangent vectors at any given control point are parallel to the line connecting the preceding point and the following point. The tangent vector has the same orientation as that line but only half its length. In other (more mathematical) words:

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

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

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

[7]:



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

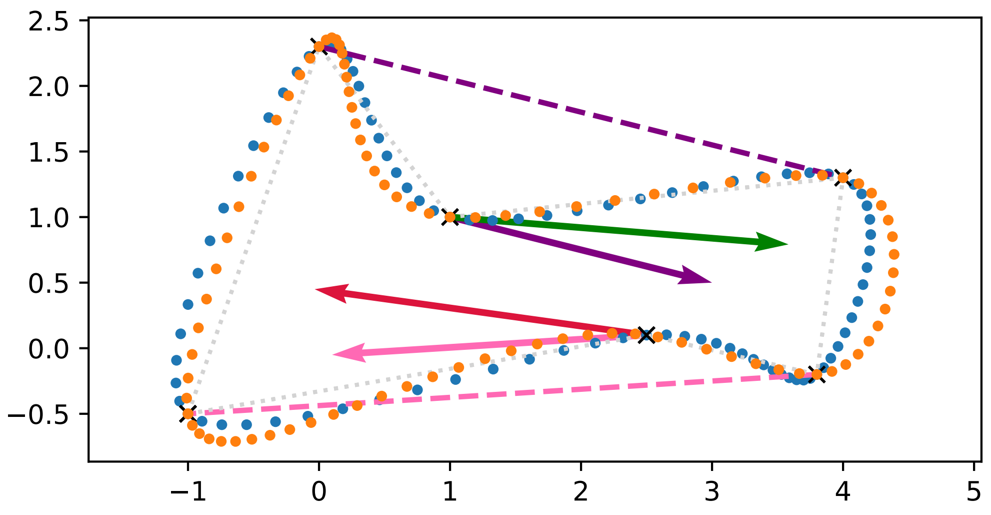
However, this will not be true anymore if we are using non-uniform time instances:

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

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

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

[10]:



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

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

The derivation of this equation is shown in *a separate notebook* (page 177).

Equivalently, this can be written as:

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

Also equivalently, with $v_i = \frac{x_{i+1} - x_i}{t_{i+1} - t_i}$, it can be written as:

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

#### B.2.8.1.2 Wrong Tangent Vectors

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

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

[11]:
```python
class MeanVelocity(splines.CatmullRom):

    @staticmethod
    def _calculate_tangent(points, times):
        x_1, x0, x1 = np.asarray(points)
        t_1, t0, t1 = times
        v_1 = (x0 - x_1) / (t0 - t_1)
        v0 = (x1 - x0) / (t1 - t0)
        return (v_1 + v0) / 2
```

Until April 2023, Wikipedia[32] showed yet a simpler equation. They mentioned that "this assumes uniform parameter spacing", but since $t_{i-1}$ and $t_{i+1}$ appeared in the equation, it might be tempting to use it for the non-uniform case as well. We'll see below how that turns out.

The authors of the page don't seem to have been quite sure about this equation, because it has changed over time. This was shown until mid-2021[33]:

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

[12]:
```python
class Wikipedia1(splines.CatmullRom):

    @staticmethod
    def _calculate_tangent(points, times):
        x_1, _, x1 = np.asarray(points)
        t_1, _, t1 = times
        return (x1 - x_1) / (t1 - t_1)
```

And this slight variation was shown since then[34] until April 2023:

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

[13]:
```python
class Wikipedia2(splines.CatmullRom):

    @staticmethod
    def _calculate_tangent(points, times):
        x_1, _, x1 = np.asarray(points)
        t_1, _, t1 = times
        return (1/2) * (x1 - x_1) / (t1 - t_1)
```

[32] https://en.wikipedia.org/wiki/Cubic_Hermite_spline#Catmull-Rom_spline
[33] https://web.archive.org/web/20210420082245/https://en.wikipedia.org/wiki/Cubic_Hermite_spline#Catmull-Rom_spline
[34] https://web.archive.org/web/20210727071020/https://en.wikipedia.org/wiki/Cubic_Hermite_spline#Catmull-Rom_spline

The first one is correct in the uniform case (which the Wikipedia page assumes), but not in the general non-uniform case, as we'll see in a moment.

The second one is obviously wrong in the case where all intervals are of length 1 (i.e. $t_{i+1} - t_i = t_i - t_{i-1} = 1$):

$$\frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_{i-1}}{4} \neq \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_{i-1}}{2} = \dot{\boldsymbol{x}}_i$$

Since April 2023, the page is showing the correct equation for the uniform case[35].

The X3D standard (version 3.3)[36] even suggests to use different incoming and outgoing tangents, which destroys $C^1$ continuity!

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

[14]:
```
class X3D(splines.KochanekBartels):
    # We derive from KochanekBartels because the
    # incoming and outgoing tangents are different:
    @staticmethod
    def _calculate_tangents(points, times, _ignored):
        x_1, _, x1 = np.asarray(points)
        t_1, t0, t1 = times
        incoming = (t1 - t0) * (x1 - x_1) / (t1 - t_1)
        outgoing = (t0 - t_1) * (x1 - x_1) / (t1 - t_1)
        return incoming, outgoing
```

To illustrate the different choices of tangent vectors, we use the vertex data from Lee (1989), figure 6:

[15]:
```
points3 = [
    (0, 0),
    (10, 25),
    (10, 24),
    (11, 24.5),
    (33, 25),
]
```

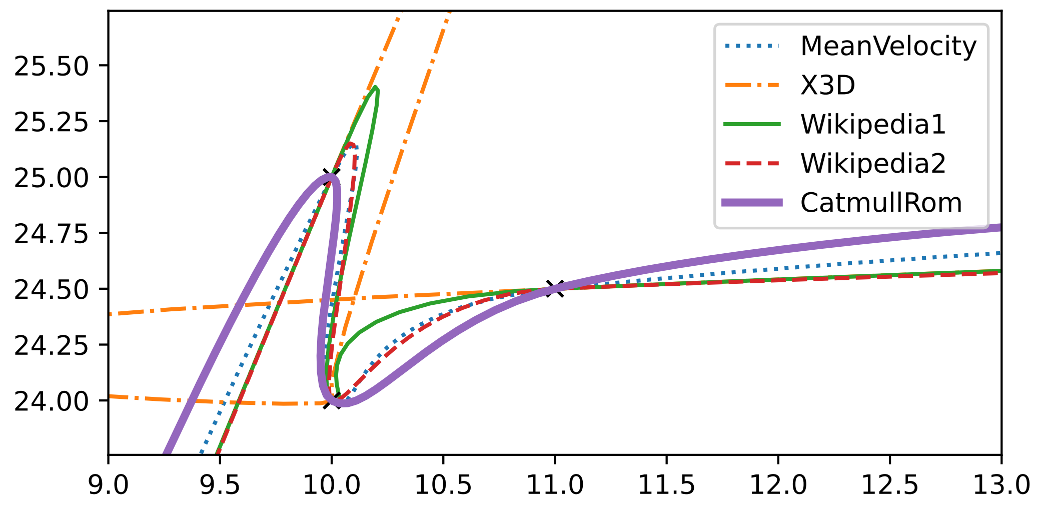
Deciding between "right" and "wrong" tangent vectors is surprisingly hard, because most of the options look somewhat reasonable in most cases. However, we can try to use quite extreme vertex positions and we can use *centripetal parameterization* (see below) and check if its guaranteed properties hold for different choices of tangent vectors.

35 https://web.archive.org/web/20230411124304/https://en.wikipedia.org/wiki/Cubic_Hermite_spline

36 https://www.web3d.org/documents/specifications/19775-1/V3.3/Part01/components/interp.html#HermiteSplineInterpolation

```
[16]: def plot_spline(cls, linestyle='-', **args):
          # alpha=0.5 => centripetal parameterization
          spline = cls(points3, alpha=0.5)
          plot_spline_2d(
              spline, label=cls.__name__, chords=False,
              marker=None, linestyle=linestyle, **args)
```

```
[17]: plot_spline(MeanVelocity, linestyle=':')
      plot_spline(X3D, linestyle='-.')
      plot_spline(Wikipedia1)
      plot_spline(Wikipedia2, linestyle='--')
      plot_spline(splines.CatmullRom, linewidth=3)
      plt.axis([9, 13, 23.9, 25.6])
      plt.legend();
```



As we can immediately see, the tangents from X3D are utterly wrong and the first one from Wikipedia is also quite obviously broken. The other two don't look too bad, but they slightly overshoot, and according to Yuksel et al. (2011) that is something that centripetal Catmull–Rom splines are guaranteed not to do.

Again, to be fair to the Wikipedia article's authors, they mentioned that uniform parameter spacing is assumed, so their equation is not supposed to be used in this non-uniform context. The equation has been changed in the meantime to avoid confusion.

#### B.2.8.1.3 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.

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

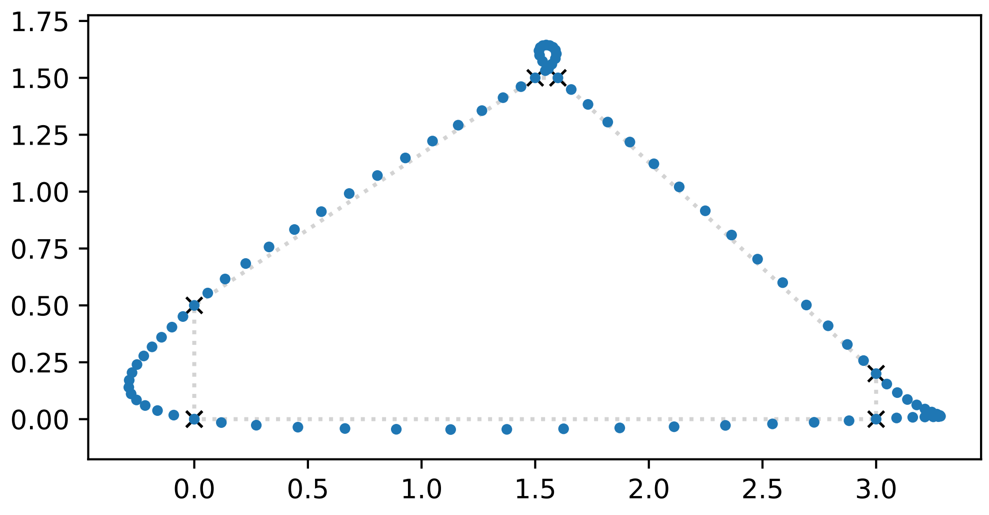
```
[19]: points4 = [
          (0, 0),
```

(continues on next page)

(continued from previous page)

```
    (0, 0.5),
    (1.5, 1.5),
    (1.6, 1.5),
    (3, 0.2),
    (3, 0),
]
```

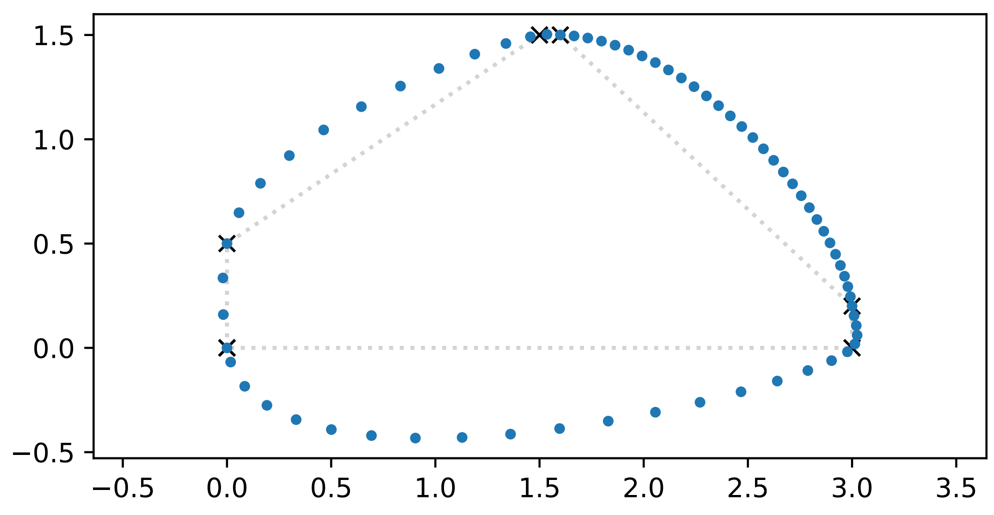
[20]:
```
plot_catmull_rom(points4)
```



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

[21]:
```
times4 = 0, 0.2, 0.9, 1, 3, 3.3, 4.5
```

[22]:
```
plot_catmull_rom(points4, times4)
```



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.

#### B.2.8.1.4 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:

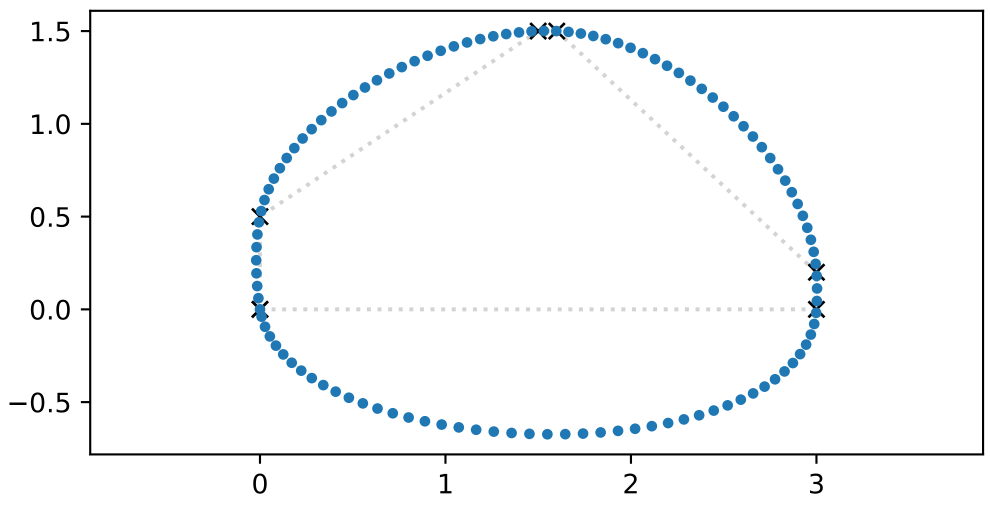
[23]:
```
distances = np.linalg.norm(np.diff(points4 + points4[:1], axis=0), axis=1)
distances
```

[23]:
```
array([0.5   , 1.8028, 0.1   , 1.9105, 0.2   , 3.    ])
```

[24]:
```
times5 = np.concatenate([[0], np.cumsum(distances)])
times5
```

[24]:
```
array([0.    , 0.5   , 2.3028, 2.4028, 4.3133, 4.5133, 7.5133])
```

[25]:
```
plot_catmull_rom(points4, times5)
```



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

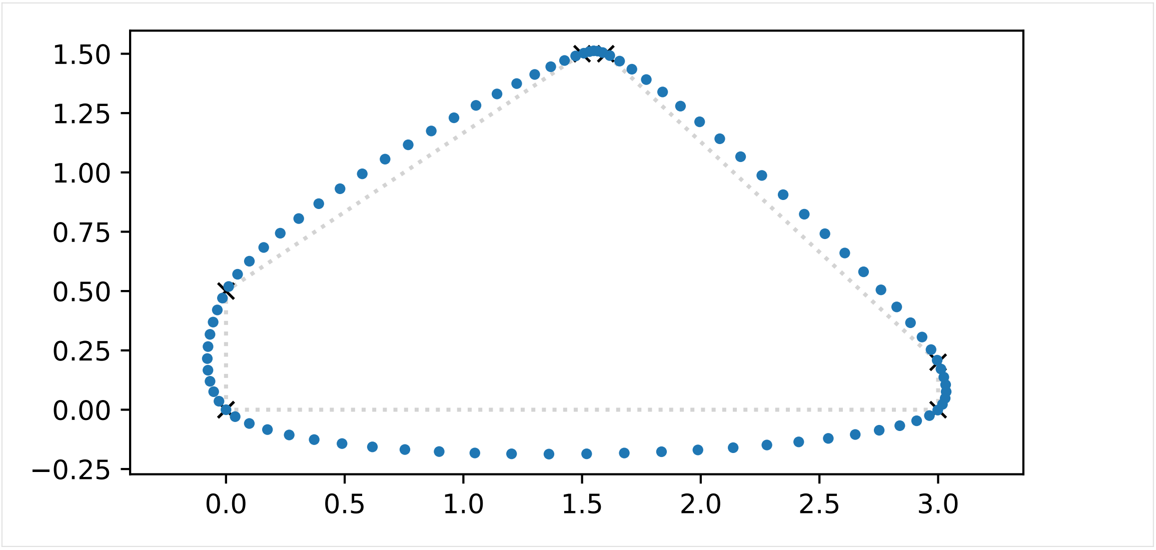
#### B.2.8.1.5 Centripetal Parameterization

As a variation of the previous method, the square roots of the chordal lengths can be used to define the time intervals (Lee 1989).

[26]:
```
times6 = np.concatenate([[0], np.cumsum(np.sqrt(distances))])
times6
```

[26]:
```
array([0.    , 0.7071, 2.0498, 2.366 , 3.7482, 4.1954, 5.9275])
```

[27]:
```
plot_catmull_rom(points4, times6)
```



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 Yuksel et al. (2011). 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.
>
> —Yuksel et al. (2011), Section 7.2: "Path Curves"

#### B.2.8.1.6 Parameterized Parameterization

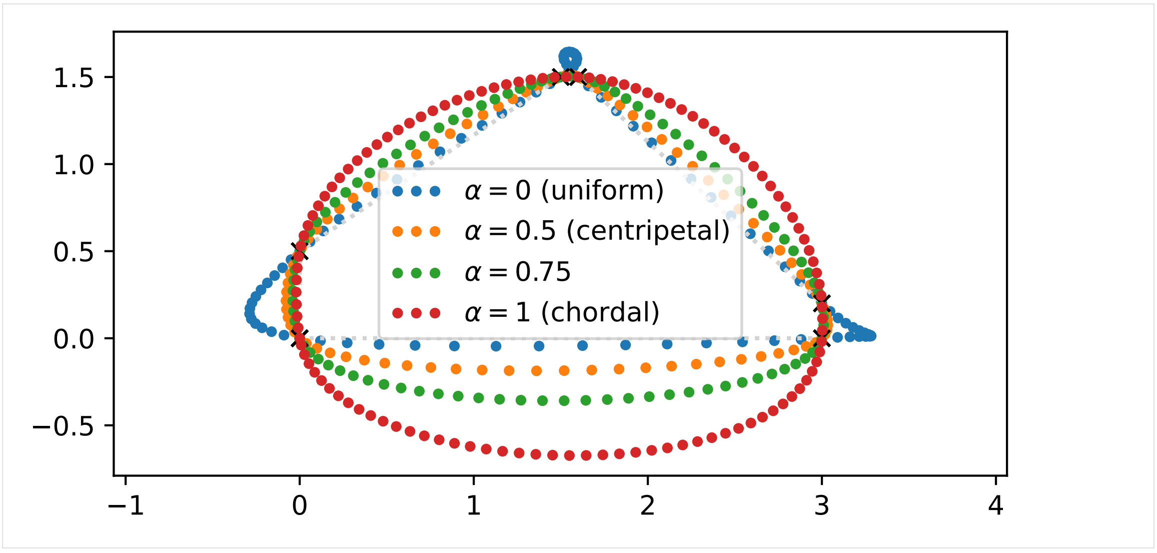
It turns out that the previous two parameterization schemes are just two special cases of a more general scheme for obtaining time intervals between control points:

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

In the Python class *splines.CatmullRom* (page 283), the parameter `alpha` can be specified.

[28]:
```
def plot_alpha(alpha, label):
    s = splines.CatmullRom(points4, alpha=alpha, endconditions='closed')
    plot_spline_2d(s, label=label)
```

[29]:
```
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 Yuksel et al. (2011) demonstrate 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 $\alpha$, 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.
>
> —Yuksel et al. (2011), 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.
>
> —Yuksel et al. (2011), 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.
>
> —Yuksel et al. (2011), 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` .......................

### B.2.8.2 Uniform Catmull–Rom Splines

Catmull and Rom (1974) presented a class of splines which can be described mathematically, in its most generic form, 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 $s$.
>
> —Catmull and Rom (1974), section 3, "Blending Functions"

The components of the equation are further constrained to produce an 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.
>
> —Catmull and Rom (1974), 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.
>
> —Catmull and Rom (1974), 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 $s_j = j$.
>
> —Catmull and Rom (1974), section 2: "The Model"

Whether or not generality is lost, this means that the rest of the paper doesn't give any hints on how to construct non-uniform splines. For those who are interested nevertheless, we show how to do that in the *notebook about non-uniform Catmull–Rom splines* (page 175) and once again in the *notebook about the Barry–Goldman algorithm* (page 181).

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 $F_i(s_j) = \delta_{ij}$.
>
> —Catmull and Rom (1974), 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 $w(s)$:
>
> $$C_{0,k}(s) = \sum_{i=0}^{k} \Bigg[ \prod_{\substack{j=i-k \\ j\neq 0}}^{i} \left(\frac{s}{j} + 1\right) \Bigg] w(s+i)$$
>
> 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 $\frac{k}{2}$.
>
> —Catmull and Rom (1974), 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]:
$$\sum_{i=0}^{k} w(i+t) \prod_{j=i-k}^{i} \begin{cases} 1 & \text{for } j = 0 \\ 1 + \frac{t}{j} & \text{otherwise} \end{cases}$$

### B.2.8.2.1 Blending Functions

Catmull and Rom (1974) leave the choice of blending function to the reader. They show two plots (figure 1 and figure 3) for a custom blending function stitched together from two Bézier curves, but they don'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 they refer to Gordon and Riesenfeld (1974). 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 (i.e. degree zero) ...

[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 (i.e. degree one) basis functions:

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

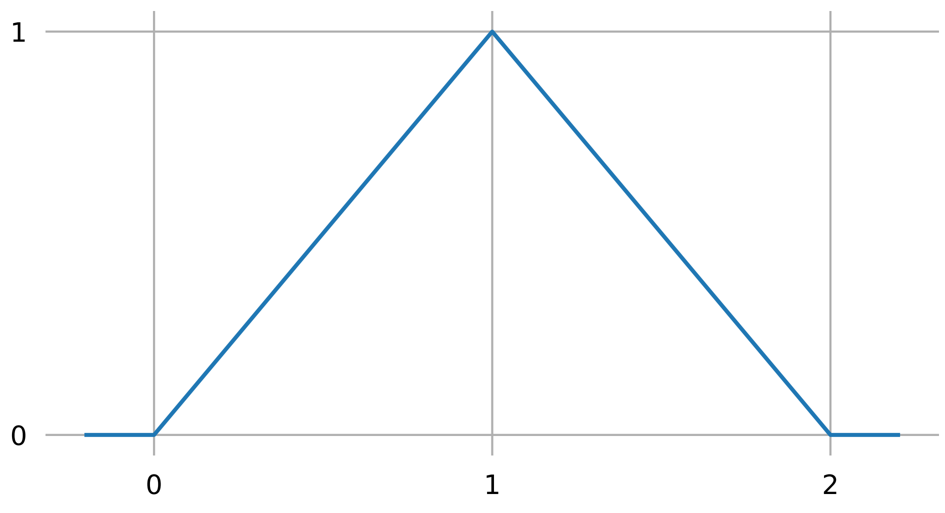
We can use one of them (where $i = 0$) as blending function:

[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])
```



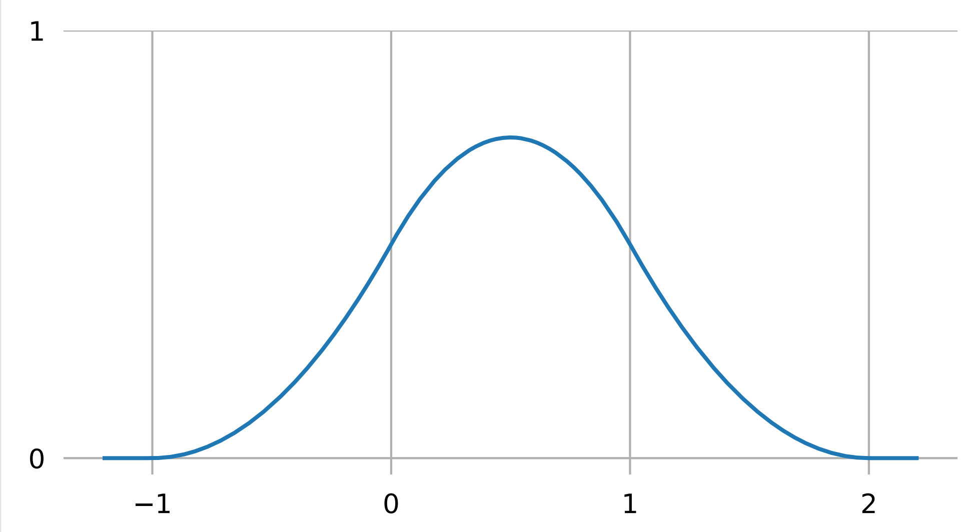
The quadratic (i.e. degree two) basis functions can be obtained like this:

[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 by Catmull and Rom (1974) in figure 2.

#### B.2.8.2.2 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.

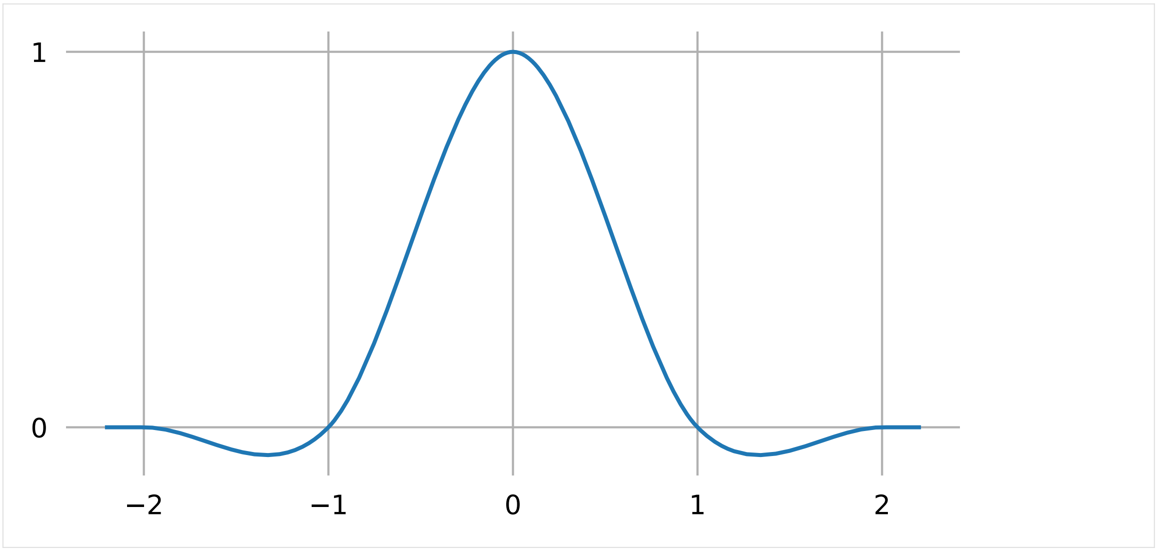
With the information so far, we can construct the cardinal function $C_{0,1}(t)$, using our *quadratic* B-spline blending function w2 (which is, as required, centered about $\frac{k}{2}$):

[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 by Catmull and Rom (1974) in figure 4.

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 181), 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 180).

#### B.2.8.2.3 Example Plot

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

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

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

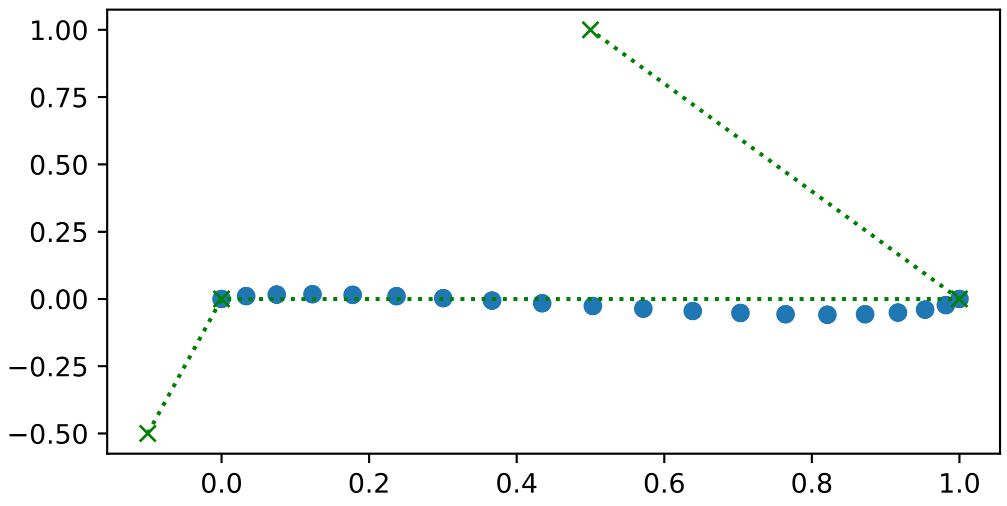
(continues on next page)

(continued from previous page)

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

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

```
[20]: plt.scatter(*np.array([
          sum([vertices[i] * CO1.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 283) can be used. For more plots, see *the notebook about properties of Catmull–Rom splines* (page 155).

#### B.2.8.2.4 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 already know how to deal with.

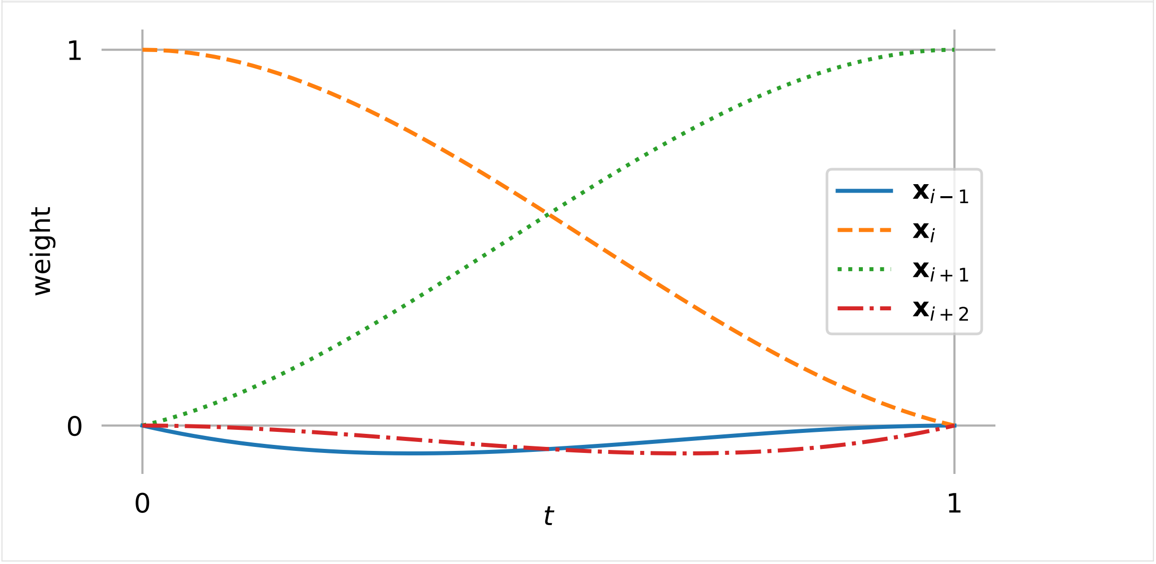
We are splitting the piecewise expression into four separate pieces, each one to be evaluated at $0 \leq t \leq 1$. We are also reversing the order of the pieces, to match our intended control point order:

```
[21]: b_CR = sp.Matrix([
          expr.subs(t, t + cond.args[1] - 1)
          for expr, cond in CO1.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
```

#### B.2.8.2.5 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 $\boldsymbol{p}_4(t)$ (from $\boldsymbol{x}_4$ to $\boldsymbol{x}_5$), where $0 \le t \le 1$. Later, we will be able to generalize this to an arbitrary polynomial segment $\boldsymbol{p}_i(t)$ (from $\boldsymbol{x}_i$ to $\boldsymbol{x}_{i+1}$), where $0 \le t \le 1$.

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

[26]:

$$\boldsymbol{p}_4 = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} M_{\text{CR}} \begin{bmatrix} \boldsymbol{x}_3 \\ \boldsymbol{x}_4 \\ \boldsymbol{x}_5 \\ \boldsymbol{x}_6 \end{bmatrix}$$

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

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

[27]:

$$\boldsymbol{p}_4 = t^3\left(-\frac{\boldsymbol{x}_3}{2} + \frac{3\boldsymbol{x}_4}{2} - \frac{3\boldsymbol{x}_5}{2} + \frac{\boldsymbol{x}_6}{2}\right) + t^2\left(\boldsymbol{x}_3 - \frac{5\boldsymbol{x}_4}{2} + 2\boldsymbol{x}_5 - \frac{\boldsymbol{x}_6}{2}\right) + t\left(-\frac{\boldsymbol{x}_3}{2} + \frac{\boldsymbol{x}_5}{2}\right) + \boldsymbol{x}_4$$

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

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

[28]:

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

Catmull and Rom (1974) show this matrix in section 6.

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

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

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

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

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

[30]:

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

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

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

#### B.2.8.2.6 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]:
```
start_tangent = pd4.evaluated_at(t, 0)
start_tangent
```

[33]:

$$\left.\frac{d}{dt}p_4\right|_{t=0} = -\frac{x_3}{2} + \frac{x_5}{2}$$

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

[34]:

$$\left.\frac{d}{dt}p_4\right|_{t=1} = -\frac{x_4}{2} + \frac{x_6}{2}$$

These two expressions can be generalized to – as already shown in *the notebook about Catmull–Rom properties* (page 157):

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

#### B.2.8.2.7 Using Bézier Segments

The above equation for the tangent vectors can be used to construct *Hermite splines* (page 105) or, after dividing them by 3, to obtain the control points for *cubic Bézier spline segments* (page 144):

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

```
[35]: x4, x5 = control_points[1:3]
```

```
[36]: x4tilde = x4 + start_tangent.expr / 3
      x4tilde
```

[36]: $-\frac{x_3}{6} + x_4 + \frac{x_5}{6}$

```
[37]: x5tilde = x5 - end_tangent.expr / 3
      x5tilde
```

[37]: $\frac{x_4}{6} + x_5 - \frac{x_6}{6}$

#### B.2.8.2.8 Using Quadrangle Interpolation

Remember the *notebook about quadrangle interpolation* (page 150)? It showed us how to calculate the quadrangle points given the Bézier control points:

$$\begin{aligned}\bar{x}_i^{(+)} &= \frac{3}{2}\tilde{x}_i^{(+)} - \frac{1}{2}x_{i+1}\\ \bar{x}_i^{(-)} &= \frac{3}{2}\tilde{x}_i^{(-)} - \frac{1}{2}x_{i-1}\end{aligned}$$

```
[38]: x4bar = 3 * x4tilde / 2 - x5 / 2
      x4bar
```

[38]: $-\frac{x_3}{4} + \frac{3x_4}{2} - \frac{x_5}{4}$

```
[39]: x5bar = 3 * x5tilde / 2 - x4 / 2
      x5bar
```

[39]:
$$-\frac{x_4}{4} + \frac{3x_5}{2} - \frac{x_6}{4}$$

Generalizing these expressions and juggling the terms around a bit, we get

$$\bar{x}_i^{(+)} = \bar{x}_i^{(-)} = x_i - \frac{(x_{i+1} - x_i) + (x_{i-1} - x_i)}{4}.$$

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

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

### B.2.8.3 Non-Uniform Catmull–Rom Splines

Catmull and Rom (1974) describe only the *uniform case* (page 165), but it is straightforward to extend their 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 169) and as we will again see in the *notebook about the Barry–Goldman algorithm* (page 184), the respective degrees can be swapped. This means that equivalently, two (overlapping) quadratic Lagrange interpolations can be used, followed by linearly blending the two resulting points.

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

We will derive the *tangent vectors* (page 177) at the segment boundaries, which will later serve as a starting point for deriving *non-uniform Kochanek–Bartels splines* (page 201). See the *notebook about the Barry–Goldman algorithm* (page 181) for an alternative (but closely related) derivation.

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

As usual, we look at the fifth polynomial segment $\boldsymbol{p}_4(t)$ from $\boldsymbol{x}_4$ to $\boldsymbol{x}_5$, where $t_4 \le t \le t_5$. Later, we will generalize this to an arbitrary polynomial segment $\boldsymbol{p}_i(t)$ from $\boldsymbol{x}_i$ to $\boldsymbol{x}_{i+1}$, where $t_i \le t \le t_{i+1}$.

[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 92), 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
    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*.

    """
    if len(xs) != len(ts):
        raise ValueError('xs and ts must have the same length')
    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()[37] 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 181) 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.

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

The following expressions can be simplified by introducing a few new symbols $\Delta_i$:

[8]:
```
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,
}
```

#### B.2.8.3.1 Tangent Vectors

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

[9]:
```
pd4 = p4.diff(t)
```

… and evaluate it at $t_4$ and $t_5$:

[10]:
```
start_tangent = pd4.evaluated_at(t, t4)
start_tangent.subs(deltas).simplify()
```

[10]:
$$\left.\frac{d}{dt}p_4\right|_{t=t_4} = \frac{-\Delta_3^2 x_4 + \Delta_3^2 x_5 - \Delta_4^2 x_3 + \Delta_4^2 x_4}{\Delta_3\Delta_4\left(\Delta_3 + \Delta_4\right)}$$

[11]:
```
end_tangent = pd4.evaluated_at(t, t5)
end_tangent.subs(deltas).simplify()
```

[11]:
$$\left.\frac{d}{dt}p_4\right|_{t=t_5} = \frac{\Delta_4^2\left(-x_5 + x_6\right) + \Delta_5^2\left(-x_4 + x_5\right)}{\Delta_4\Delta_5\left(\Delta_4 + \Delta_5\right)}$$

Both results lead to the same general expression (which is expected, since the incoming and outgoing tangents are supposed to be equal):

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

Equivalently, this can be written as:

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

An alternative (but very similar) way to derive these tangent vectors is shown in the *notebook about the Barry–Goldman algorithm* (page 189).

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.

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

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

[13]: 2

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

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

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

$$\dot{x}_i = \frac{\Delta_i v_{i-1} + \Delta_{i-1} v_i}{\Delta_{i-1} + \Delta_i},$$

with $\Delta_i = t_{i+1} - t_i$ and $v_i = \frac{x_{i+1} - x_i}{\Delta_i}$.

Boor (1978) 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 does *not* describe the tangents of *Overhauser splines* as presented by Overhauser (1968).

Long story short, it's the same as we had above:

```
[15]: 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.

```
[16]: second_quadratic = neville([x4, x5, x6], [t4, t5, t6], t)
      second_quadratic.diff(t).subs(t, t5)
```

[16]: $$\frac{\frac{(-t_4+t_5)(-x_5+x_6)}{-t_5+t_6} + \frac{(-t_5+t_6)(-x_4+x_5)}{-t_4+t_5}}{-t_4+t_6}$$

```
[17]: assert sp.simplify(_ - end_tangent.expr) == 0
```

You might encounter yet another way to write the equation for $\dot{x}_4$ (e.g. at `https://stackoverflow.com/a/23980479/`) ...

```
[18]: (x4 - x3) / (t4 - t3) - (x5 - x3) / (t5 - t3) + (x5 - x4) / (t5 - t4)
```

[18]:
$$\frac{-x_4 + x_5}{-t_4 + t_5} - \frac{-x_3 + x_5}{-t_3 + t_5} + \frac{-x_3 + x_4}{-t_3 + t_4}$$

... but again, this is equivalent to the equation shown above:

```
[19]: assert sp.simplify(_ - start_tangent.expr) == 0
```

### B.2.8.3.2 Using Non-Uniform Bézier Segments

Similar to *the uniform case* (page 174), the above equation for the tangent vectors can be used to construct non-uniform *Hermite splines* (page 105) 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 150):

$$\begin{aligned}
\tilde{x}_i^{(+)} &= x_i + \frac{\Delta_i \dot{x}_i}{3}\\
&= x_i + \frac{\Delta_i}{3} \frac{\Delta_i v_{i-1} + \Delta_{i-1} v_i}{\Delta_{i-1} + \Delta_i}\\
&= x_i + \frac{{\Delta_i}^2 (x_i - x_{i-1})}{3\Delta_{i-1}(\Delta_i + \Delta_{i-1})} + \frac{\Delta_{i-1}(x_{i+1} - x_i)}{3(\Delta_i + \Delta_{i-1})}\\
\tilde{x}_i^{(-)} &= x_i - \frac{\Delta_{i-1} \dot{x}_i}{3}\\
&= x_i - \frac{\Delta_{i-1}}{3} \frac{\Delta_i v_{i-1} + \Delta_{i-1} v_i}{\Delta_{i-1} + \Delta_i}\\
&= x_i - \frac{\Delta_i (x_i - x_{i-1})}{3(\Delta_i + \Delta_{i-1})} - \frac{{\Delta_{i-1}}^2 (x_{i+1} - x_i)}{3\Delta_i(\Delta_i + \Delta_{i-1})}
\end{aligned}$$

This is again using $\Delta_i = t_{i+1} - t_i$ and $v_i = \frac{x_{i+1} - x_i}{\Delta_i}$.

```
[20]: x4tilde = x4 + (t5 - t4) * start_tangent.expr / 3
```

```
[21]: x5tilde = x5 - (t5 - t4) * end_tangent.expr / 3
```

### B.2.8.3.3 Using Non-Uniform Quadrangle Interpolation

Just like in *the uniform case* (page 174), we calculate the quadrangle points from the Bézier control points, as shown in the *notebook about quadrangle interpolation* (page 150):

$$\begin{aligned}
\bar{x}_i^{(+)} &= \frac{3}{2}\tilde{x}_i^{(+)} - \frac{1}{2}x_{i+1}\\
\bar{x}_i^{(-)} &= \frac{3}{2}\tilde{x}_i^{(-)} - \frac{1}{2}x_{i-1}
\end{aligned}$$

[22]:
```
x4bar = 3 * x4tilde / 2 - x5 / 2
```

[23]:
```
terms4 = sp.collect(x4bar.expand(), [x3, x4, x5], evaluate=False)
```

Some manual rewriting leads to this expression:

[24]:
```
sp.factor(terms4[x4] + terms4[x5] + terms4[x3]) * x4 - (
    sp.factor(-terms4[x5]) * (x5 - x4) +
    sp.factor(-terms4[x3]) * (x3 - x4))
```

[24]:

$$x_4 - \frac{(t_4 - t_5)(-x_4 + x_5)}{2(t_3 - t_5)} - \frac{(t_4 - t_5)^2 (x_3 - x_4)}{2(t_3 - t_4)(t_3 - t_5)}$$

We should make sure that our re-written expression is actually the same as the one we started from:

[25]:
```
assert sp.simplify(_ - x4bar) == 0
```

Now the same for the incoming quadrangle point:

[26]:
```
x5bar = 3 * x5tilde / 2 - x4 / 2
```

[27]:
```
terms5 = sp.collect(x5bar.expand(), [x4, x5, x6], evaluate=False)
```

[28]:
```
sp.factor(terms5[x5] + terms5[x6] + terms5[x4]) * x5 - (
    sp.factor(-terms5[x6]) * (x6 - x5) +
    sp.factor(-terms5[x4]) * (x4 - x5))
```

[28]:

$$x_5 - \frac{(t_4 - t_5)^2 (-x_5 + x_6)}{2(t_4 - t_6)(t_5 - t_6)} - \frac{(t_4 - t_5)(x_4 - x_5)}{2(t_4 - t_6)}$$

[29]:
```
assert sp.simplify(_ - x5bar) == 0
```

The above expressions can be generalized to (as always with $\Delta_i = t_{i+1} - t_i$):

$$\begin{aligned}
\bar{x}_i^{(+)} &= x_i - \frac{\Delta_i}{2(\Delta_{i-1} + \Delta_i)} \left( (x_{i+1} - x_i) + \frac{\Delta_i}{\Delta_{i-1}} (x_{i-1} - x_i) \right) \\
\bar{x}_i^{(-)} &= x_i - \frac{\Delta_{i-1}}{2(\Delta_{i-1} + \Delta_i)} \left( \frac{\Delta_{i-1}}{\Delta_i} (x_{i+1} - x_i) + (x_{i-1} - x_i) \right)
\end{aligned}$$

#### B.2.8.3.4 Animation

To illustrate what two quadratic Lagrange interpolations followed by linear blending might look like, we can generate an animation by means of the file `catmull_rom.py`, with some help from `helper.py`:

[30]:
```
from catmull_rom import animation_2_1, animation_1_2
from helper import show_animation
```

[31]:
```
vertices = [
    (1, 0),
    (0.5, 1),
    (6, 2),
    (5, 0),
]
```

[32]:
```
times = [
    0,
    1,
    6,
    8,
]
```

[33]:
```
show_animation(animation_2_1(vertices, times))
```

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

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 what three linear interpolations (and *extra*polations) followed by quadratic B-spline blending would look like:

[34]:
```
show_animation(animation_1_2(vertices, times))
```

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

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 190).

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

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

#### B.2.8.4 Barry–Goldman Algorithm

The *Barry–Goldman algorithm* – named after Barry and Goldman (1988) – can be used to calculate values of *non-uniform Catmull–Rom splines* (page 175). We have also applied this algorithm to *rotation splines* (page 264).

Catmull and Rom (1974) describe "a class of local interpolating splines" and Barry and Goldman (1988) describe "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 92) 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.

—Barry and Goldman (1988), 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.
>
> —Barry and Goldman (1988), 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 189), which will show that the algorithm indeed generates *non-uniform Catmull–Rom splines* (page 177).

#### B.2.8.4.1 Triangular Schemes

Barry and Goldman (1988) illustrate the presented algorithms 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 $\boldsymbol{x}_4$ and $\boldsymbol{x}_5$ with corresponding parameter values $t_4$ and $t_5$, respectively):

$$\begin{array}{ccccc} & & \boldsymbol{p}_{4,5} & & \\ & \frac{t_5-t}{t_5-t_4} & & \frac{t-t_4}{t_5-t_4} & \\ \boldsymbol{x}_4 & & & & \boldsymbol{x}_5 \end{array}$$

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 $\boldsymbol{p}_{4,5}$ one has to add $\boldsymbol{x}_4$, weighted by the factor shown next to it ($\frac{t_5-t}{t_5-t_4}$), and $\boldsymbol{x}_5$, 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 $t_4 \le t \le t_5$. 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)
```

#### B.2.8.4.2 Neville's Algorithm

We have already seen this algorithm in our *notebook about Lagrange interpolation* (page 95), where we have shown the triangular scheme for the *cubic* case – which is also shown by Barry and Goldman (1988) in figure 2. In the *quadratic* case, it looks like this:

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

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

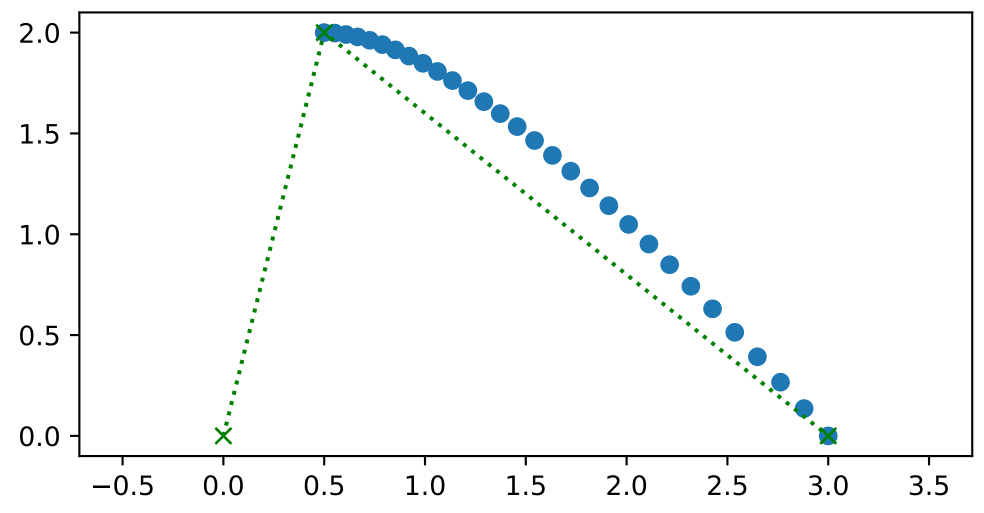
Let's try to plot this for three points:

```
[3]: points = np.array([
         (0, 0),
         (0.5, 2),
         (3, 0),
     ])
```

In the following example plots we show the *uniform* case (with $t_3 = 3$, $t_4 = 4$ and $t_5 = 5$), but don't worry, the algorithm works just as well for arbitrary non-uniform time values.

```
[4]: plot_times = np.linspace(4, 5, 30)
```

```
[5]: plt.scatter(*np.array([
         lerp(
             [lerp(points[:2], [3, 4], t), lerp(points[1:], [4, 5], t)],
             [3, 5], t)
         for t in plot_times]).T)
     plt.plot(*points.T, 'x:g')
     plt.axis('equal');
```



Note that the quadratic curve is defined by three points but we are only evaluating it between two of them (for $4 \leq t \leq 5$).

#### B.2.8.4.3 De Boor's Algorithm

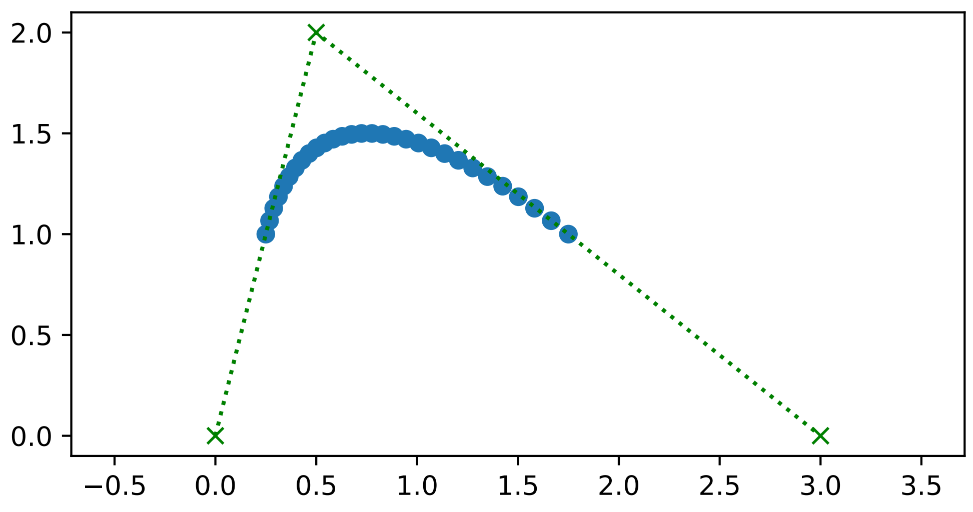
This algorithm (Boor 1972) can be used to calculate B-spline basis functions.

The quadratic case looks like this:

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

The *cubic* case is shown by Barry and Goldman (1988) in figure 1.

```
[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');
```



#### B.2.8.4.4 Combining Both Algorithms

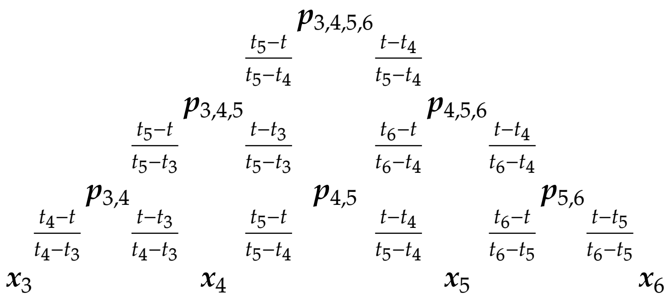
Catmull and Rom (1974) show (in figure 5) an example where linear interpolation is followed by quadratic B-spline blending to create a cubic curve.

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:



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 $t_4 \leq t \leq t_5$. 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 by Yuksel et al. (2011) in figure 3, except that here we shifted the indices by +3.

Another way to construct a cubic curve with this algorithm would be to swap the degrees of interpolation and blending, in other words:

- Instead of three linear interpolations (and extrapolations), apply two overlapping quadratic Lagrange interpolations using Neville's algorithm (as shown above) to $\boldsymbol{x}_3$, $\boldsymbol{x}_4$, $\boldsymbol{x}_5$ and $\boldsymbol{x}_4$, $\boldsymbol{x}_5$, $\boldsymbol{x}_6$, respectively. Note that the interpolation of $\boldsymbol{x}_4$ and $\boldsymbol{x}_5$ appears in both triangles but has to be calculated only once – see also figures 3 and 4 by Barry and Goldman (1988).
- 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 165) 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 by Barry and Goldman (1988) in figure 7, and they casually mention 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 $n = 2$, $m = 1$ [...]
>
> —Barry and Goldman (1988), section 3: "Examples"

**Not an Overhauser Spline**

Equally casually, they mention:

Finally, the particular case here is also an Overhauser spline (Overhauser 1968).

—Barry and Goldman (1988), section 3: "Examples"

This is not true. Overhauser splines – as described by Overhauser (1968) – 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 163). 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 by Barry and Goldman (1988) in figure 8.
- No interpolation at all, followed by cubic B-spline blending, which leads to an approximating spline (instead of an interpolating spline), as shown by Barry and Goldman (1988) in figure 5.

**Note**

Here we are using the time instances of the Lagrange interpolation also as B-spline knots. Barry and Goldman (1988) show a more generic formulation of the algorithm with separate parameters $s_i$ and $t_i$ in equation (9).

### B.2.8.4.5 Step by Step

The triangular figure above looks more complicated than it really is. It's just a bunch of linear *inter*polations and *extra*polations.

Let's go through the figure above, piece by piece.

[7]:
```
import sympy as sp
```

[8]:
```
t = sp.symbols('t')
```

[9]:
```
x3, x4, x5, x6 = sp.symbols('xbm3:7')
```

[10]:
```
t3, t4, t5, t6 = sp.symbols('t3:7')
```

We use some custom SymPy-based tools from `utility.py`:

[11]:
```
from utility import NamedExpression, NamedMatrix
```

**First Stage** In the center of the bottom row, there is a straightforward linear interpolation from $x_4$ to $x_5$ within the interval from $t_4$ to $t_5$.

```
[12]: p45 = NamedExpression('pbm_4,5', lerp([x4, x5], [t4, t5], t))
      p45
```

[12]:
$$p_{4,5} = \frac{x_4 \left(-t + t_5\right) + x_5 \left(t - t_4\right)}{-t_4 + t_5}$$

Obviously, this starts at:

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

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

... and ends at:

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

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

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]:
$$p_{3,4}\Big|_{t=t_4} = x_4$$

... and ending at some extrapolated point beyond $x_4$:

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

[17]:
$$p_{3,4}\Big|_{t=t_5} = \frac{x_3 \left(t_4 - t_5\right) + x_4 \left(-t_3 + t_5\right)}{-t_3 + t_4}$$

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]:
$$p_{5,6}\Big|_{t=t_4} = \frac{x_5(-t_4+t_6)+x_6(t_4-t_5)}{-t_5+t_6}$$

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

[20]:
$$p_{5,6}\Big|_{t=t_5} = x_5$$

**Second Stage** The second stage of the algorithm involves linear interpolations of the results of the previous stage.

[21]:
```
p345 = NamedExpression('pbm_3,4,5', lerp([p34.name, p45.name], [t3, t5],␣
↪t))
p345
```

[21]:
$$p_{3,4,5} = \frac{p_{3,4}(-t+t_5)+p_{4,5}(t-t_3)}{-t_3+t_5}$$

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

[22]:
$$p_{4,5,6} = \frac{p_{4,5}(-t+t_6)+p_{5,6}(t-t_4)}{-t_4+t_6}$$

Those interpolations are defined over a parameter range from $t_3$ to $t_5$ and from $t_4$ to $t_6$, respectively. In each case, we are only interested in a sub-range, namely from $t_4$ to $t_5$.

These are the start and end points at $t_4$ and $t_5$:

[23]:
```
p345.evaluated_at(t, t4, symbols=[p34, p45])
```

[23]:
$$p_{3,4,5}\Big|_{t=t_4} = \frac{p_{3,4}\Big|_{t=t_4}(-t_4+t_5)+p_{4,5}\Big|_{t=t_4}(-t_3+t_4)}{-t_3+t_5}$$

[24]:
```
p345.evaluated_at(t, t5, symbols=[p34, p45])
```

[24]:
$$p_{3,4,5}\Big|_{t=t_5} = p_{4,5}\Big|_{t=t_5}$$

[25]:
```
p456.evaluated_at(t, t4, symbols=[p45, p56])
```

[25]:
$$p_{4,5,6}\Big|_{t=t_4} = p_{4,5}\Big|_{t=t_4}$$

[26]:
```
p456.evaluated_at(t, t5, symbols=[p45, p56])
```

[26]:
$$p_{4,5,6}\Big|_{t=t_5} = \frac{p_{4,5}\Big|_{t=t_5}(-t_5+t_6)+p_{5,6}\Big|_{t=t_5}(-t_4+t_5)}{-t_4+t_6}$$

**Third Stage** The last step is quite simple:

[27]:
```
p3456 = NamedExpression(
    'pbm_3,4,5,6',
    lerp([p345.name, p456.name], [t4, t5], t))
p3456
```

[27]:

$$p_{3,4,5,6} = \frac{p_{3,4,5}\left(-t + t_5\right) + p_{4,5,6}\left(t - t_4\right)}{-t_4 + t_5}$$

This time, the interpolation interval is exactly the one we are interested in.

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()
```

This expression is quite unwieldy, so let's not even look at it.

[29]:
```
#p3456
```

Apart from checking whether it's really cubic ...

[30]:
```
sp.degree(p3456.expr, t)
```

[30]: 3

... and whether it's really interpolating ...

[31]:
```
p3456.evaluated_at(t, t4).simplify()
```

[31]:

$$\left.p_{3,4,5,6}\right|_{t=t_4} = x_4$$

[32]:
```
p3456.evaluated_at(t, t5).simplify()
```

[32]:

$$\left.p_{3,4,5,6}\right|_{t=t_5} = x_5$$

... the only thing left to do is to check its ...

#### B.2.8.4.6 Tangent Vectors

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

[33]:
```
pd3456 = p3456.diff(t)
```

... and evaluate it at $t_4$ and $t_5$:

[34]:
```
pd3456.evaluated_at(t, t4).simplify().simplify()
```

[34]:

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

[35]:
```
pd3456.evaluated_at(t, t5).simplify()
```

[35]:

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

If all went well, this should be identical to the result in *the notebook about non-uniform Catmull–Rom splines* (page 177). As we have mentioned there, it isn't even necessary to calculate the last interpolation to get the tangent vectors. At the beginning of the interval ($t = t_4$), only the first quadratic polynomial $\boldsymbol{p}_{3,4,5}(t)$ contributes to the final result, while the other one has a weight of zero. At the end of the interval ($t = t_5$), only $\boldsymbol{p}_{4,5,6}(t)$ is relevant. Therefore, we can simply take their tangent vectors at $t_4$ and $t_5$, respectively, and we get the same result:

```
[36]: p345.subs_symbols(p34, p45).diff(t).evaluated_at(t, t4).simplify()
```

[36]:
$$\left.\frac{d}{dt}\boldsymbol{p}_{3,4,5}\right|_{t=t_4} = \frac{(t_3 - t_4)^2 (\boldsymbol{x}_4 - \boldsymbol{x}_5) + (t_4 - t_5)^2 (\boldsymbol{x}_3 - \boldsymbol{x}_4)}{(t_3 - t_4)(t_3 - t_5)(t_4 - t_5)}$$

```
[37]: p456.subs_symbols(p45, p56).diff(t).evaluated_at(t, t5).simplify()
```

[37]:
$$\left.\frac{d}{dt}\boldsymbol{p}_{4,5,6}\right|_{t=t_5} = \frac{(t_4 - t_5)^2 (\boldsymbol{x}_5 - \boldsymbol{x}_6) + (t_5 - t_6)^2 (\boldsymbol{x}_4 - \boldsymbol{x}_5)}{(t_4 - t_5)(t_4 - t_6)(t_5 - t_6)}$$

#### B.2.8.4.7 Animation

The linear interpolations (and *extra*polations) of this algorithm can be shown graphically. By means of the file `barry_goldman.py` – and with the help of `helper.py` – we can show an animation of the algorithm:

```
[38]: from barry_goldman import animation
      from helper import show_animation
```

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

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

```
[41]: show_animation(animation(vertices, times))

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

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 180).

........................................... doc/euclidean/catmull-rom-barry-goldman.ipynb ends here.

### B.2.9 Kochanek–Bartels Splines

Kochanek–Bartels splines (a.k.a. TCB splines) are named after Kochanek and Bartels (1984).

A Python implementation is available in the class *splines.KochanekBartels* (page 284).

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

#### B.2.9.1 Properties of Kochanek–Bartels Splines

Kochanek–Bartels 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 155).

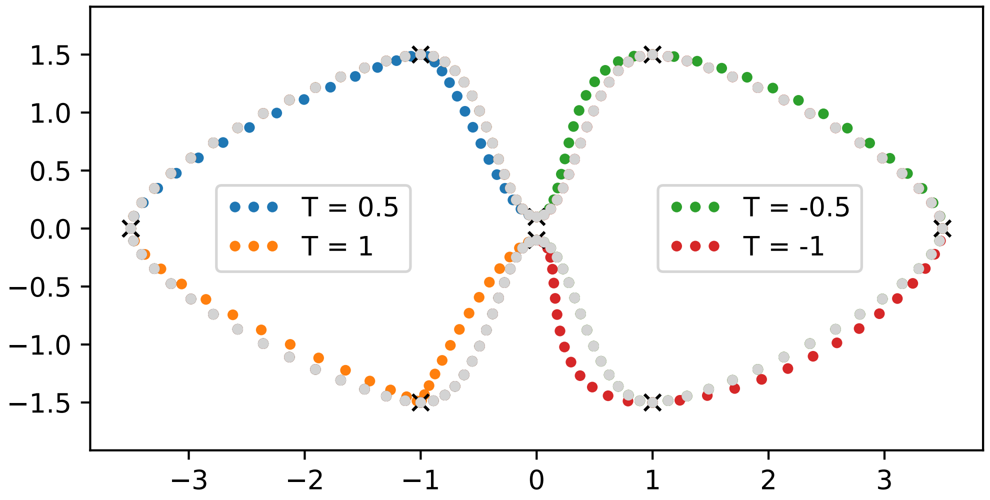
[1]:
```
import splines
from helper import plot_spline_2d
```

Let's use a bespoke plotting function from `kochanek_bartels.py` to illustrate the TCB parameters:

[2]:
```
from kochanek_bartels import plot_tcb
```

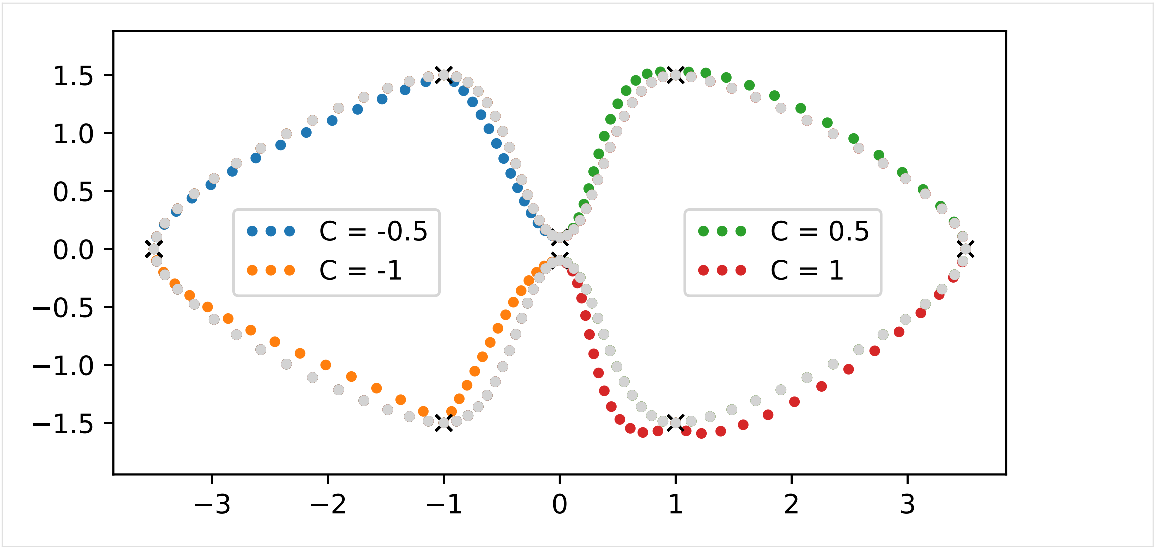
##### B.2.9.1.1 Tension

[3]:
```
plot_tcb((0.5, 0, 0), (1, 0, 0), (-0.5, 0, 0), (-1, 0, 0))
```



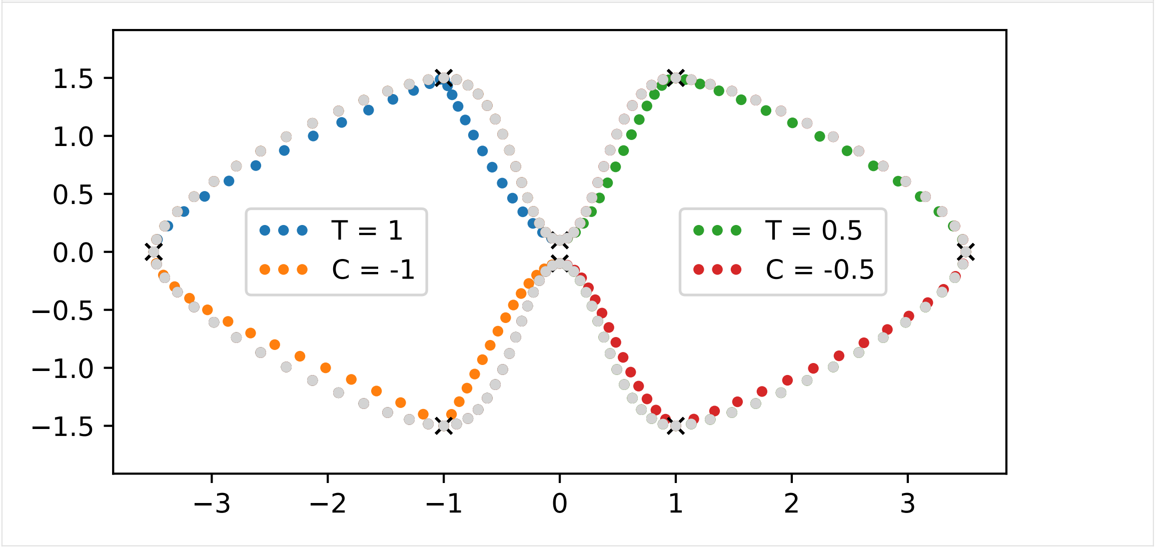
##### B.2.9.1.2 Continuity

[4]:
```
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[38]), but they have a different timing (and therefore different velocities):

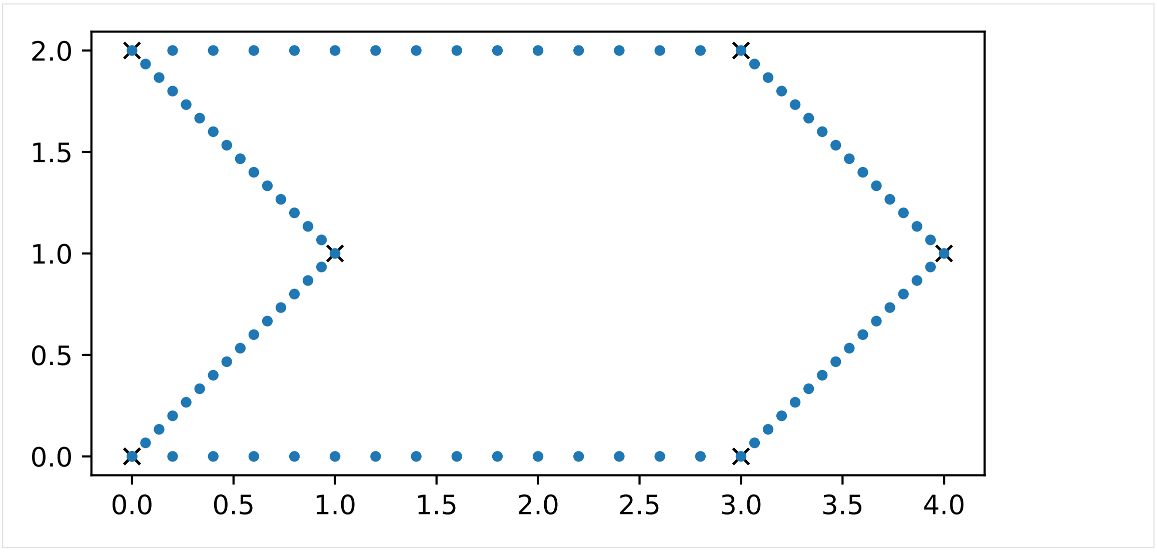
```
[5]: plot_tcb((1, 0, 0), (0, -1, 0), (0.5, 0, 0), (0, -0.5, 0))
```



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

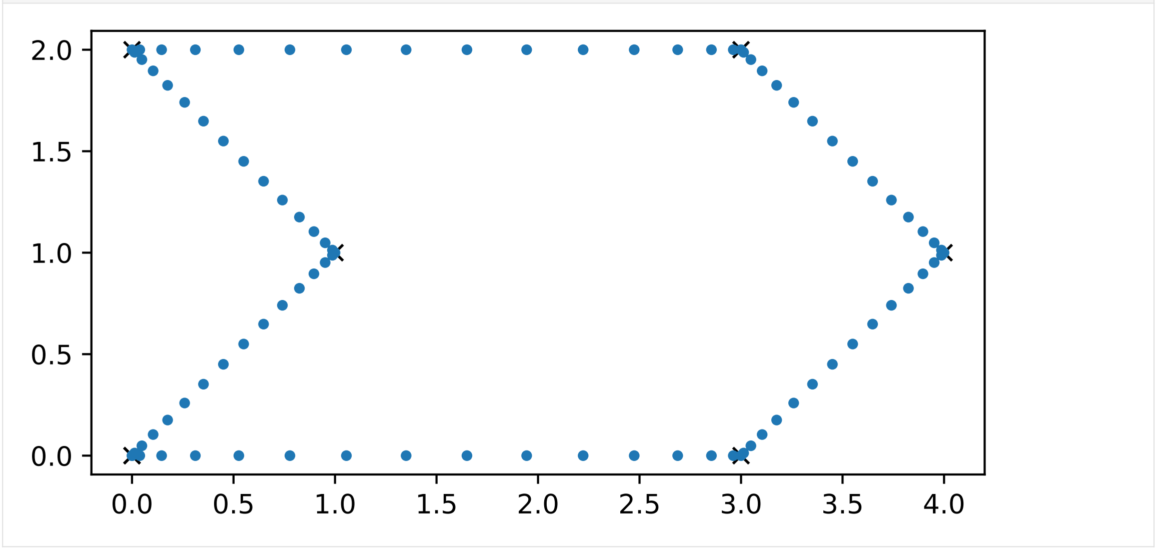
```
[6]: vertices1 = [(0, 0), (1, 1), (0, 2), (3, 2), (4, 1), (3, 0)]
s1a = splines.KochanekBartels(vertices1, tcb=(0, -1, 0), endconditions=
↪'closed')
plot_spline_2d(s1a, chords=False)
```

[38] https://en.wikipedia.org/wiki/Image_(mathematics)



A value of $T = 1$ will lead to linear segments as well, but the speed will fluctuate in each segment, coming to a complete halt at each control point:

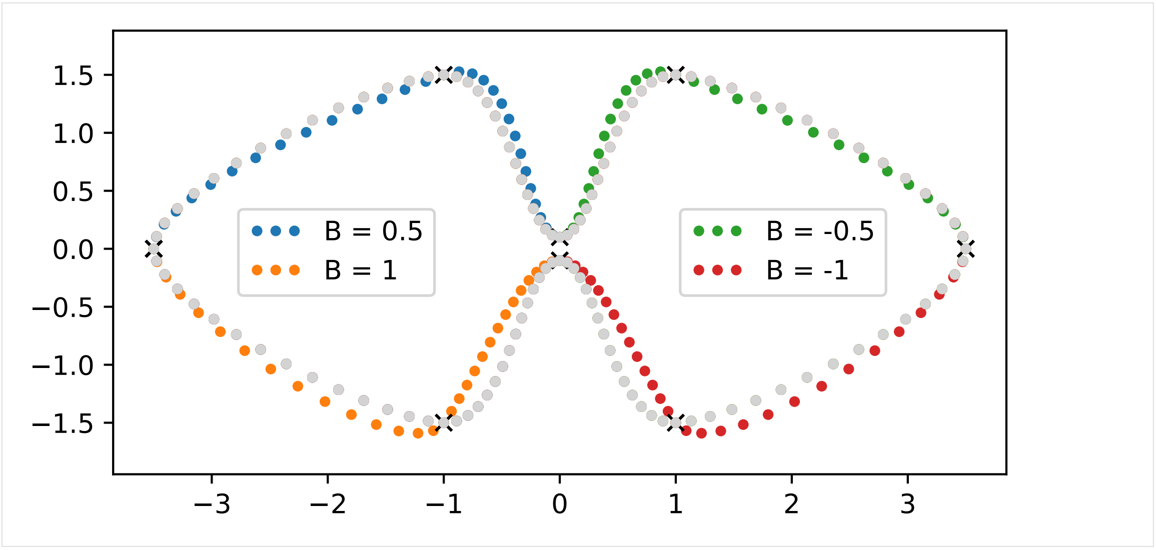
```
[7]: s1b = splines.KochanekBartels(vertices1, tcb=(1, 0, 0), endconditions=
      ↪'closed')
     plot_spline_2d(s1b, chords=False)
```



#### B.2.9.1.3 Bias

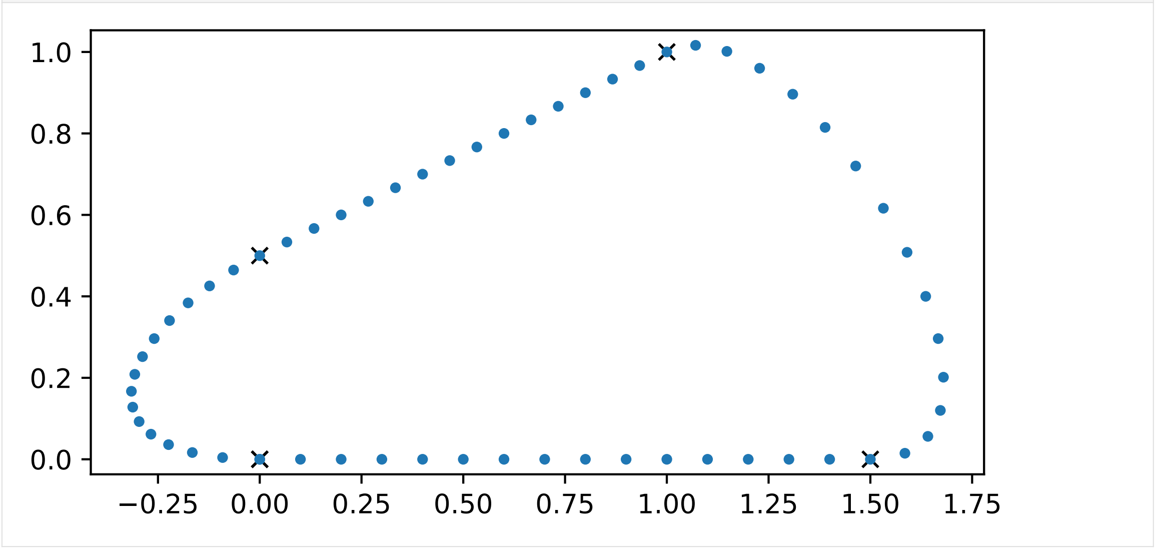
This could also be called *overshoot* (if $B > 0$) and *undershoot* (if $B < 0$):

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



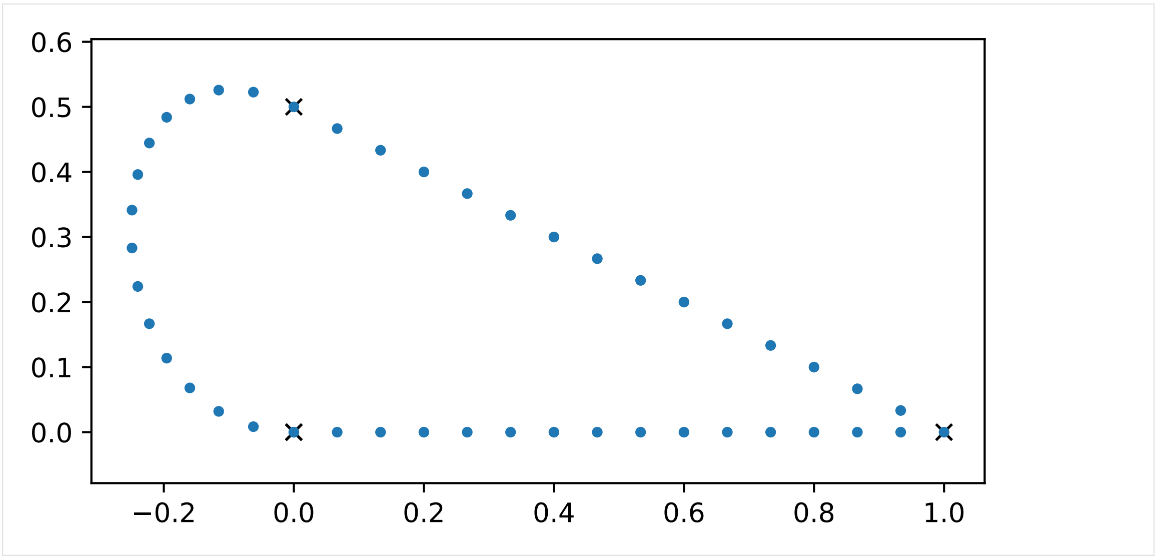
Bias −1 followed by +1 can be used to achieve linear segments between two control points:

```
[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:

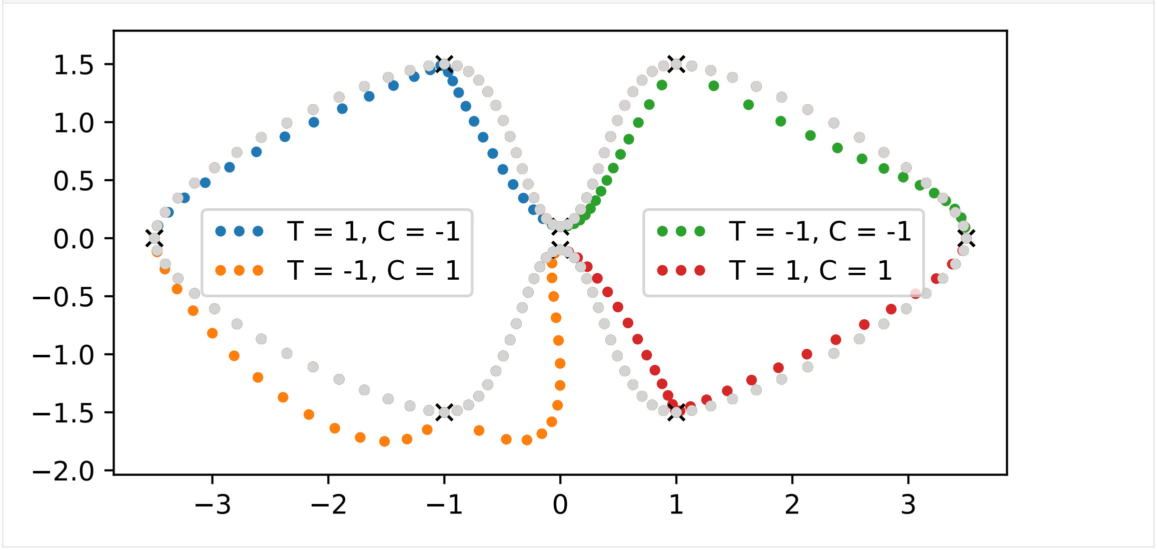
```
[10]: vertices3 = [(0, 0), (1, 0), (0, 0.5)]
      tcb3 = [(0, 0, -1), (0, -1, 0), (0, 0, 1)]
      s3 = splines.KochanekBartels(vertices3, tcb=tcb3, endconditions='closed')
      plot_spline_2d(s3, chords=False)
```



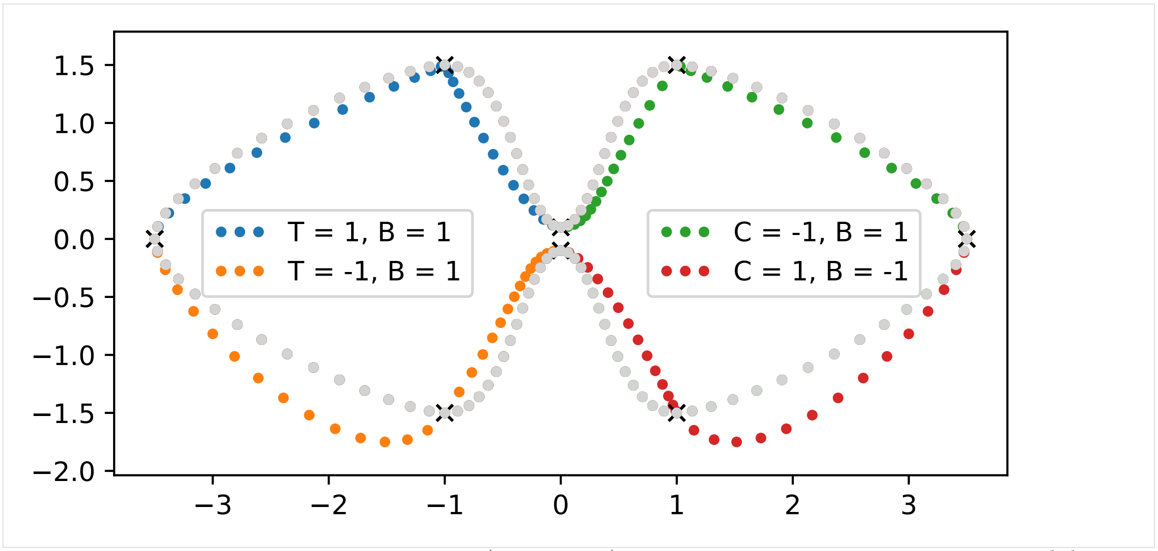
#### B.2.9.1.4 Combinations

Of course, multiple parameters can be combined:

[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` ................

### B.2.9.2 Uniform Kochanek–Bartels Splines

As a starting point, remember the *tangent vectors of uniform Catmull–Rom splines* (page 157) – see also equation 3 of the paper by Kochanek and Bartels (1984):

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

which can be re-written as

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

#### B.2.9.2.1 Parameters

Deriving *TCB splines* is all about inserting the parameters $T$, $C$ and $B$ into this equation.

**Tension** Kochanek and Bartels (1984) show the usage of $T$ in equation 4:

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

**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:

$$\dot{x}_i = \dot{x}_i^{(-)} = \dot{x}_i^{(+)}$$

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 the paper by Kochanek and Bartels (1984):

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

**Bias** Kochanek and Bartels (1984) show the usage of $B$ in equation 7:

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

**All Three Combined** To get the tangent vectors of a TCB spline, the three equations can be combined – see equations 8 and 9 in the paper by (Kochanek and Bartels 1984):

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

**Note**

There is an error in equation (6.11) from Millington (2009). 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 (Millington 2009):

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

which lead to the simplified equations

$$\begin{aligned}\dot{x}_i^{(+)} &= \frac{a_i(x_i - x_{i-1}) + b_i(x_{i+1} - x_i)}{2}\\ \dot{x}_i^{(-)} &= \frac{c_i(x_i - x_{i-i}) + d_i(x_{i+1} - x_i)}{2}\end{aligned}$$

#### B.2.9.2.2 Calculation

The above tangent vectors are sufficient to implement Kochanek–Bartels splines via *Hermite splines* (page 105). 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)
```

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

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

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

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

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

**Basis Matrix** Let's 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 110) …

[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 88) 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:

[17]:
```
b_KB = b_KB.subs_symbols(a4, b4, c5, d5).simplify()
```

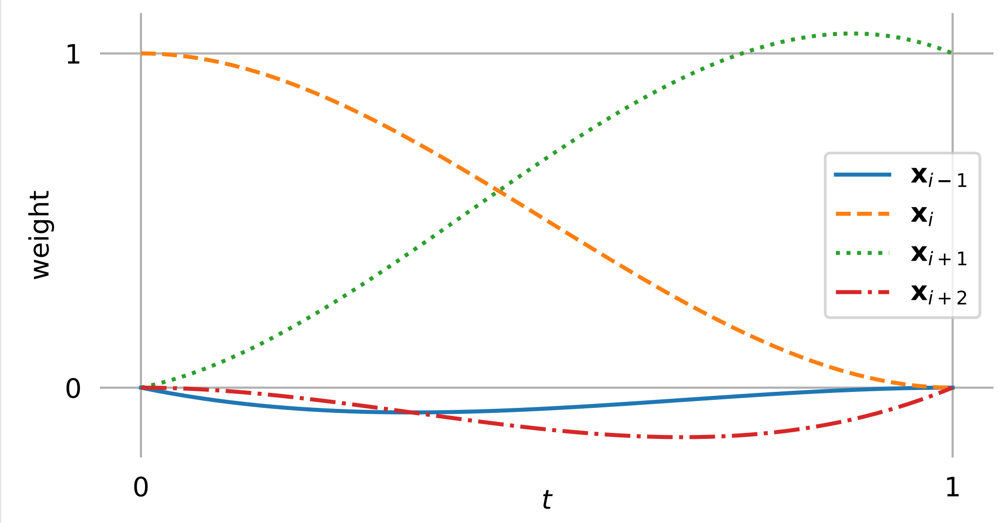
Let's use a helper function from `helper.py`:

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

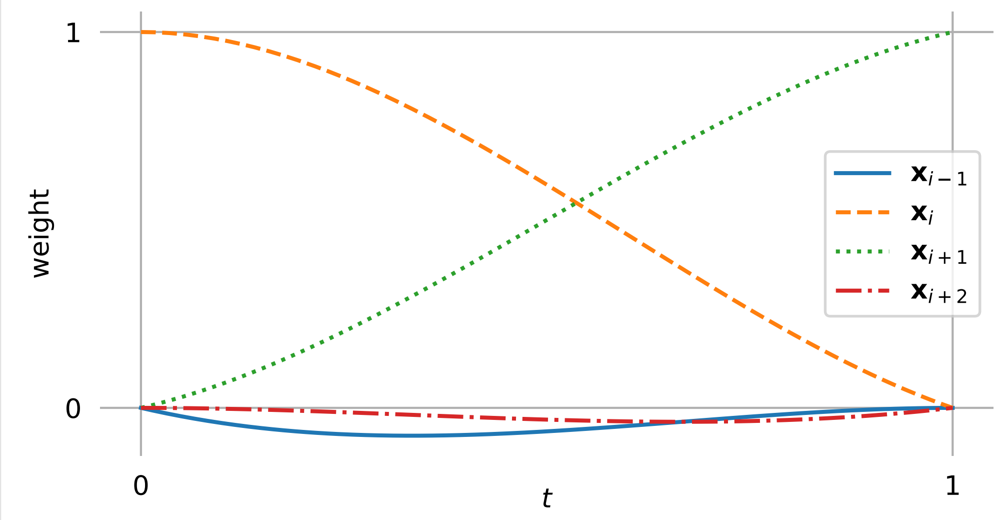
[19]:
```
labels = sp.symbols('xbm_i-1 xbm_i xbm_i+1 xbm_i+2')
```

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 171).

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

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

### B.2.9.3 Non-Uniform Kochanek–Bartels Splines

Kochanek and Bartels (1984) mainly talk about uniform splines. Only in section 4 – "Adjustments for Parameter Step Size" – do they 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:
>
> $$\text{adjusted } DD_i = DD_i \frac{2N_{i-1}}{N_{i-1} + N_i}$$
> $$\text{adjusted } DS_i = DS_i \frac{2N_i}{N_{i-1} + N_i}$$
>
> —Kochanek and Bartels (1984), section 4

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 *notebook about Catmull–Rom splines* (page 157).

[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 110) and which is also shown by Kochanek and Bartels (1984) in equation 2:

[3]:
```
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 281) module and have to re-implement everything from scratch:

[4]:
```
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)
    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**

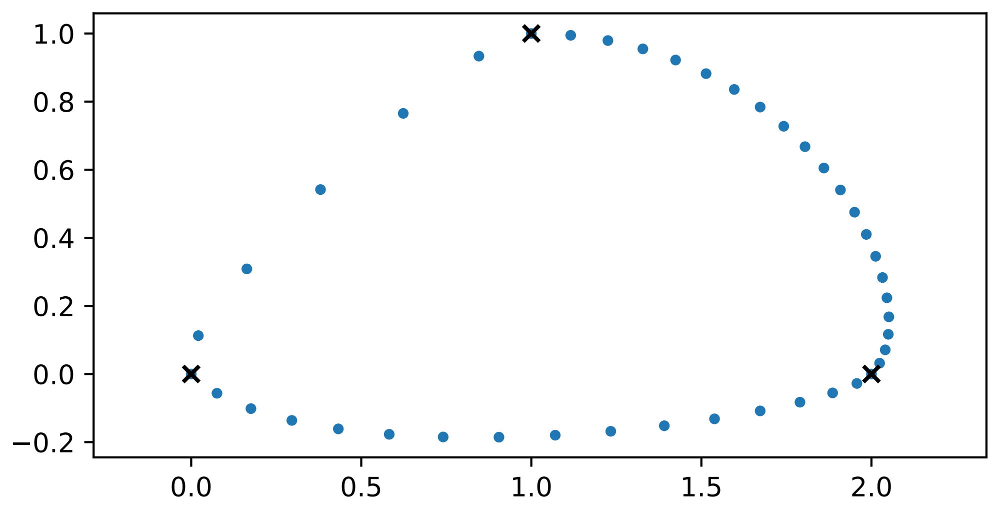
The @ operator is used here to do NumPy's matrix multiplication[39].

Let's plot an example:

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

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

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



This doesn't look too bad, let's plot the same thing with *splines.CatmullRom* (page 283) 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 ...

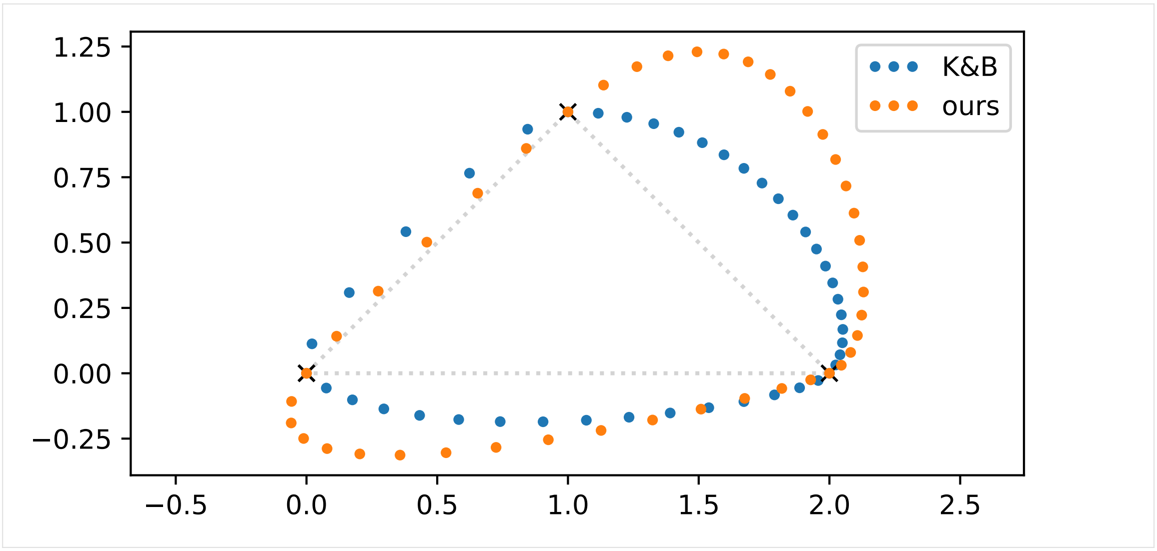
[10]:
```
cr_spline1 = CatmullRom(vertices1, times1, endconditions='closed')
```

... 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.

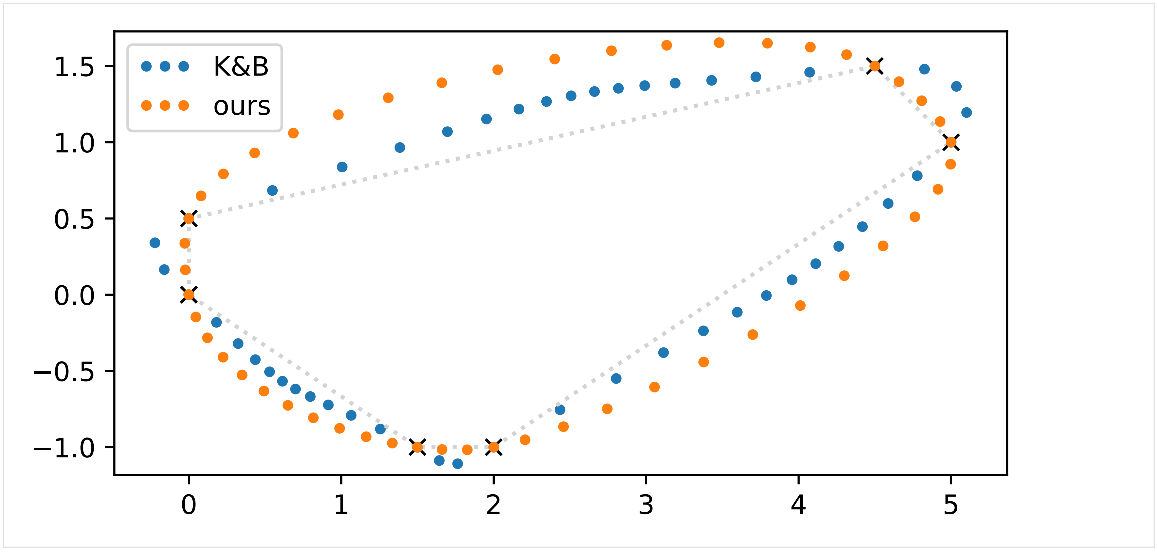
Let's look at a different example:

[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 177):

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

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 197), 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 284).

........................................... `doc/euclidean/kochanek-bartels-non-uniform.ipynb` ends here.

### B.2.10 End Conditions

Most spline types that are defined by a sequence of control points to be interpolated need some additional information to be able to draw their segments at the beginning and at the end. For example, cubic *Catmull–Rom splines* (page 155) need four consecutive control points to define the segment between the middle two. For the very first and last segment, the fourth control point is missing. Another example are *natural splines* (page 124), which would require to solve an underdetermined system of equations when only the control points are given.

There are many ways to provide this missing information, here we will mention only a few of them.

**clamped**
: This means providing a fixed tangent (i.e. first derivative) at the beginning and end of a cubic spline. For higher degree splines, additional derivatives have to be be specified.

**natural**
: For a cubic spline, this means setting the second derivative at the beginning and end of the spline to zero and calculating the first derivative from that constraint, see *Natural End Conditions* (page 207).

**closed**
: This problem can also be solved by simply not having a begin and an end. When reaching the last control point, the spline can just continue at the first control point. For non-uniform splines an additional parameter interval has to be specified for the segment that's inserted between the end and the beginning.

For most splines in the *splines module* (page 281), *clamped*, *natural* and *closed* end conditions are available via the `endconditions` argument. Except for *closed*, the end conditions can differ between the beginning and end of the spline.

Additional information is available for *end conditions of natural splines* (page 130) and *monotone end conditions* (page 224).

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

#### B.2.10.1 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 $\Delta_i = 1$.

Natural end conditions are naturally a good fit for *natural splines* (page 130). And 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')
```

#### B.2.10.1.1 Begin

We are starting with the first polynomial segment $p_0(t)$, 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 88) give us the uniform polynomial ...

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

[6]: $d_0 t^3 + c_0 t^2 + b_0 t + a_0$

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

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

[7]: $$p_0 = \frac{d_0 (t - t_0)^3}{(-t_0 + t_1)^3} + \frac{c_0 (t - t_0)^2}{(-t_0 + t_1)^2} + \frac{b_0 (t - t_0)}{-t_0 + t_1} + a_0$$

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

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

[8]: $$\frac{d}{dt} p_0 = \frac{3 d_0 (t - t_0)^2}{(-t_0 + t_1)^3} + \frac{c_0 \cdot (2t - 2t_0)}{(-t_0 + t_1)^2} + \frac{b_0}{-t_0 + t_1}$$

Similar to the *notebook about non-uniform Hermite splines* (page 118), 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]:
$$\frac{d^2}{dt^2} p_0 = \frac{3d_0 \cdot (2t - 2t_0)}{(-t_0 + t_1)^3} + \frac{2c_0}{(-t_0 + t_1)^2}$$

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

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

[15]:
$$\left.\frac{d^2}{dt^2} p_0\right|_{t=t_0} = \frac{2c_0}{(-t_0 + t_1)^2}$$

... is set to zero ...

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

[16]:
$$\frac{2 \cdot (2t_0\dot{x}_0 - 2t_1\dot{x}_0 + t_0\dot{x}_1 - t_1\dot{x}_1 - 3x_0 + 3x_1)}{(-t_0 + t_1)^2} = 0$$

... leading to an expression for the initial tangent vector:

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

[17]:
$$\dot{x}_0 = -\frac{\Delta_0 \dot{x}_1 + 3x_0 - 3x_1}{2\Delta_0}$$

This can also be written as

$$\dot{x}_0 = \frac{3(x_1 - x_0)}{2\Delta_0} - \frac{\dot{x}_1}{2}.$$

#### B.2.10.1.2 End

If a spline has $N$ vertices, it has $N-1$ polynomial segments and the last polynomial segment is $p_{N-2}(t)$, 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 $p_8$ 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]: $d_8 t^3 + c_8 t^2 + b_8 t + a_8$

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

[21]:
$$p_8 = \frac{d_8 (t - t_8)^3}{(-t_8 + t_9)^3} + \frac{c_8 (t - t_8)^2}{(-t_8 + t_9)^2} + \frac{b_8 (t - t_8)}{-t_8 + t_9} + a_8$$

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

[22]:
$$\frac{d}{dt} p_8 = \frac{3 d_8 (t - t_8)^2}{(-t_8 + t_9)^3} + \frac{c_8 \cdot (2t - 2t_8)}{(-t_8 + t_9)^2} + \frac{b_8}{-t_8 + t_9}$$

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

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

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

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

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

... is set to zero ...

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

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

... 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

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

#### B.2.10.1.3 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
```

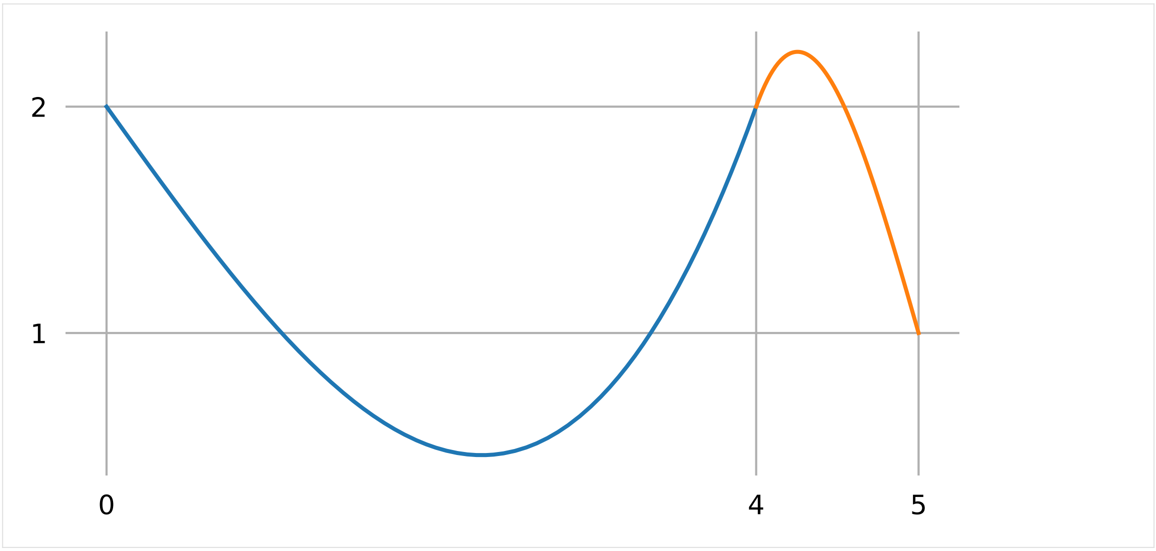
We are using a few helper functions from `helper.py` for plotting:

[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],
    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]:

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

#### B.2.10.1.4 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 105). What if we want to construct a *Bézier spline* (page 134) 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 149) 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),
```

(continues on next page)

(continued from previous page)

```
    xd9.name: 3 * (x9 - xtilde9) / (t9 - t8),
}), xtilde9)
```

[42]:

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

Note that all $\Delta_i$ 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:

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

$$\tilde{x}_{N-1}^{(-)} = \frac{x_{N-1} + \tilde{x}_{N-2}^{(+)}}{2}$$

.................................................. doc/euclidean/end-conditions-natural.ipynb ends here.

The following section was generated from doc/euclidean/piecewise-monotone.ipynb ........................

## B.2.11 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 222) below.

A Python implementation of one-dimensional piecewise monotone cubic splines is available in the class *splines.PiecewiseMonotoneCubic* (page 285).

The SciPy package provides a similar tool with the pchip_interpolate()[40] function and the PchipInterpolator[41] class (see below for more details).

The 3D animation software Blender[42] provides an Auto Clamped[43] property for creating piecewise monotone animation cuves.

### B.2.11.1 Examples

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

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

[40] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.pchip_interpolate.html
[41] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PchipInterpolator.html
[42] https://www.blender.org
[43] https://docs.blender.org/manual/en/dev/editors/graph_editor/fcurves/properties.html#editors-graph-fcurves-settings-handles

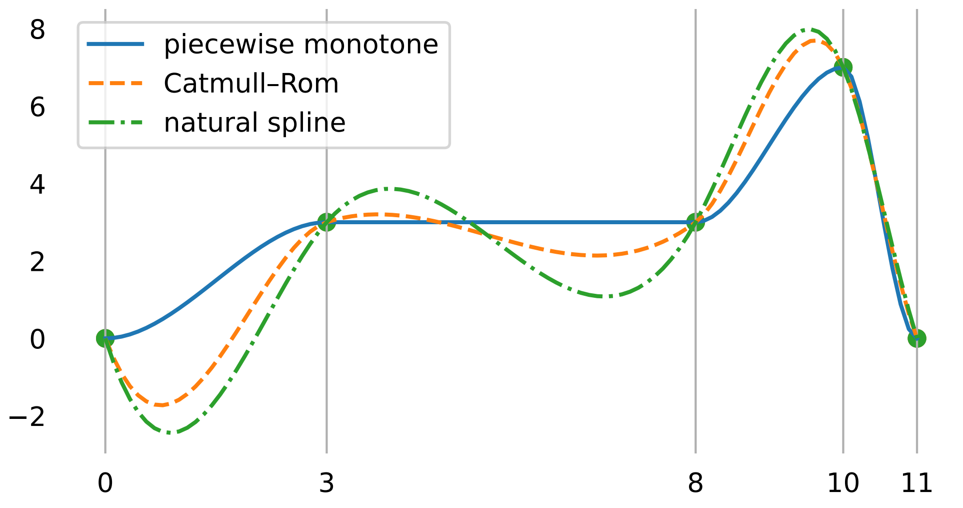
We use a few helper functions from `helper.py` for plotting:

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

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

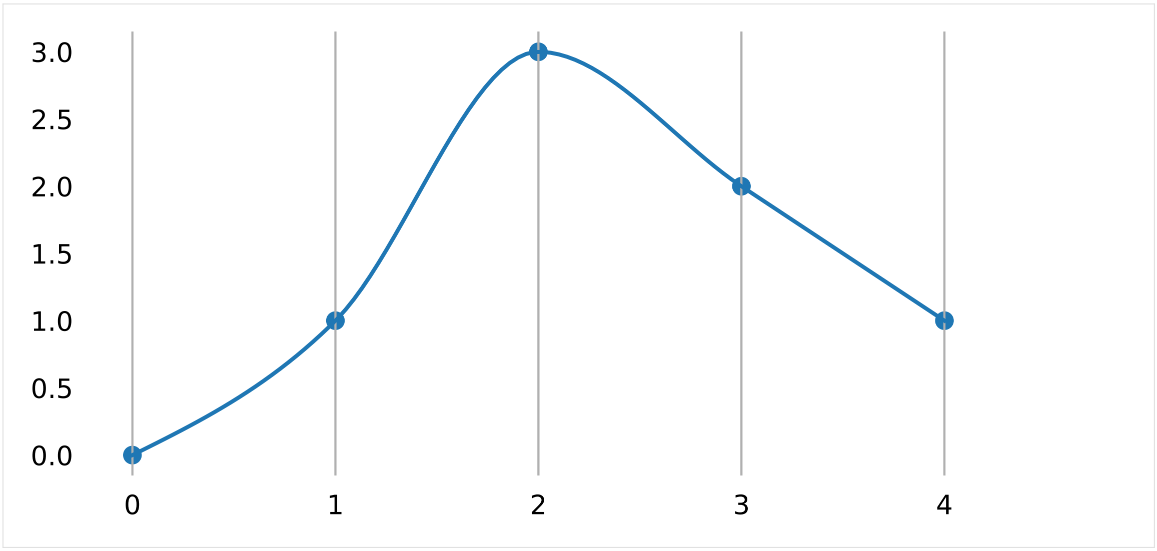
Let's compare a piecewise monotone spline with a *Catmull–Rom spline* (page 155) and a *natural spline* (page 124):

[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', linestyle='--')
plot_spline_1d(
    splines.Natural(values, times, endconditions='closed'),
    label='natural spline', linestyle='-.')
plt.legend()
grid_lines(times)
```



[6]:
```
def plot_piecewise_monotone(*args, **kwargs):
    s = splines.PiecewiseMonotoneCubic(*args, **kwargs)
    plot_spline_1d(s)
    grid_lines(s.grid)
```

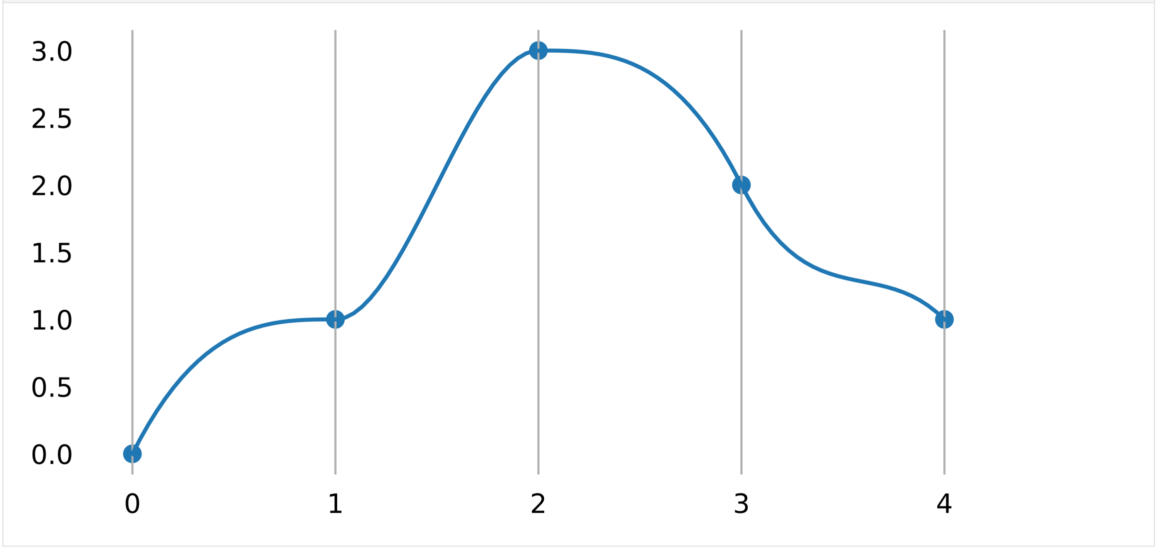
[7]:
```
plot_piecewise_monotone([0, 1, 3, 2, 1])
```



#### B.2.11.1.1 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
```

#### B.2.11.2 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.
>
> —Fritsch and Carlson (1980), section 1

Fritsch and Carlson (1980) derive 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

For the first step, they suggest using the *standard three-point difference*, which we have already seen in the *tangent vectors of non-uniform Catmull–Rom splines* (page 177) and which is implemented in the class *splines.CatmullRom* (page 283).

> To implement Step 1 we have found the standard three-point difference formula to be satisfactory for $d_2, d_3, \cdots, d_{n-1}$.
>
> —Fritsch and Carlson (1980), section 4

> This is what de Boor [(Boor 1978), p. 53] calls cubic Bessel interpolation, in which the interior derivatives are set using the standard three point difference formula.
>
> —Fritsch and Carlson (1980), section 5

In the 2001 edition of the book by Boor (1978), *piecewise cubic Bessel interpolation* is defined on page 42.

For the following equations, we define the slope of the secant lines as

$$S_i = \frac{x_{i+1} - x_i}{t_{i+1} - t_i}.$$

We use $x_i$ to represent the given data points and and $t_i$ to represent the corresponding parameter values. The slope at those values is represented by $\dot{x}_i$.

**Note**

In the literature, the parameter values are often represented by $x_i$, so try not to be confused!

Based on Fritsch and Carlson (1980), Dougherty et al. (1989) provide (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 algorithm is implemented in the class *splines.PiecewiseMonotoneCubic* (page 285).

#### B.2.11.3 PCHIP/PCHIM

A different approach for obtaining slopes that ensure monotonicity is described by Fritsch and Butland (1984), 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[44] 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[45] package described by Fritsch (1982), who 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.
>
> —Fritsch (1982)

[44] https://www.sympy.org/
[45] https://en.wikipedia.org/wiki/Fortran

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 online[46]. This is the code snippet responsible for calculating the slopes:

```
C
C       USE BRODLIE MODIFICATION OF BUTLAND FORMULA.
C
   45   CONTINUE
        HSUMT3 = HSUM+HSUM+HSUM
        W1 = (HSUM + H1)/HSUMT3
        W2 = (HSUM + H2)/HSUMT3
        DMAX = MAX( ABS(DEL1), ABS(DEL2) )
        DMIN = MIN( ABS(DEL1), ABS(DEL2) )
        DRAT1 = DEL1/DMAX
        DRAT2 = DEL2/DMAX
        D(1,I) = DMIN/(W1*DRAT1 + W2*DRAT2)
```

This looks different from the function $G$ defined above, but if we transform the Fortran code into math ...

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

```
[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 that they are in fact also equivalent to the aforementioned equation from Fritsch and Butland (1984):

```
[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 by Moler (2004), 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 $\delta_k = S_k$.

[46] `https://netlib.org/slatec/pchip/dpchim.f`

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()[47] function and the PchipInterpolator[48] class.

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

#### B.2.11.4 More Examples

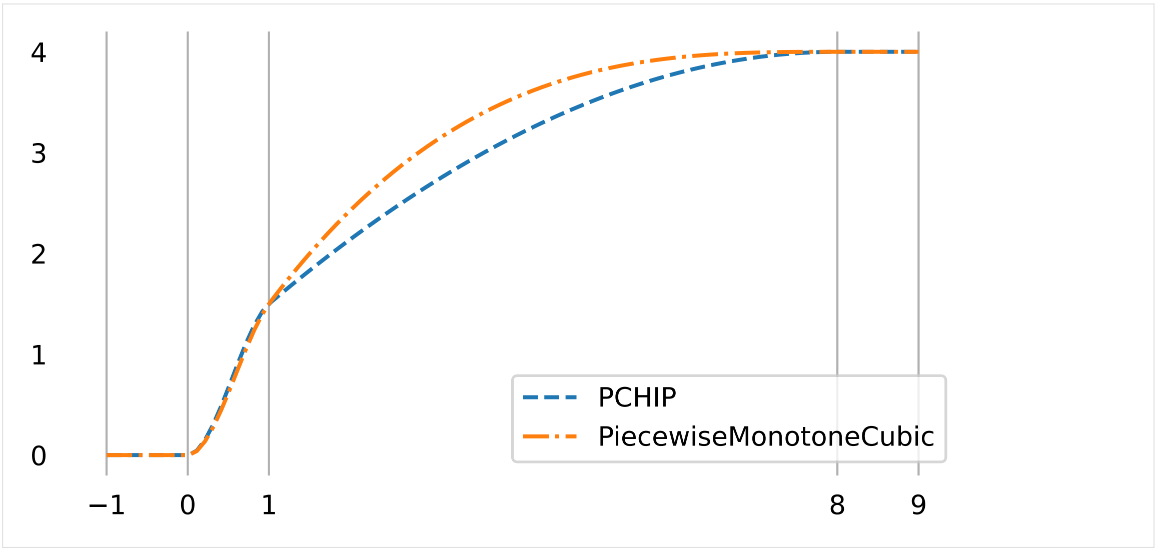
To illustrate the differences between the two approaches mentioned above, let's plot a few examples. Both methods are piecewise monotone, but their exact shape is slightly different. Decide for yourself which one is more "visually pleasing"!

```
[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', linestyle='--')
          plt.plot(
              plot_times,
              splines.PiecewiseMonotoneCubic(values, times).evaluate(plot_
      ↪times),
              label='PiecewiseMonotoneCubic', linestyle='-.')
          plt.legend()
          grid_lines(times)
```

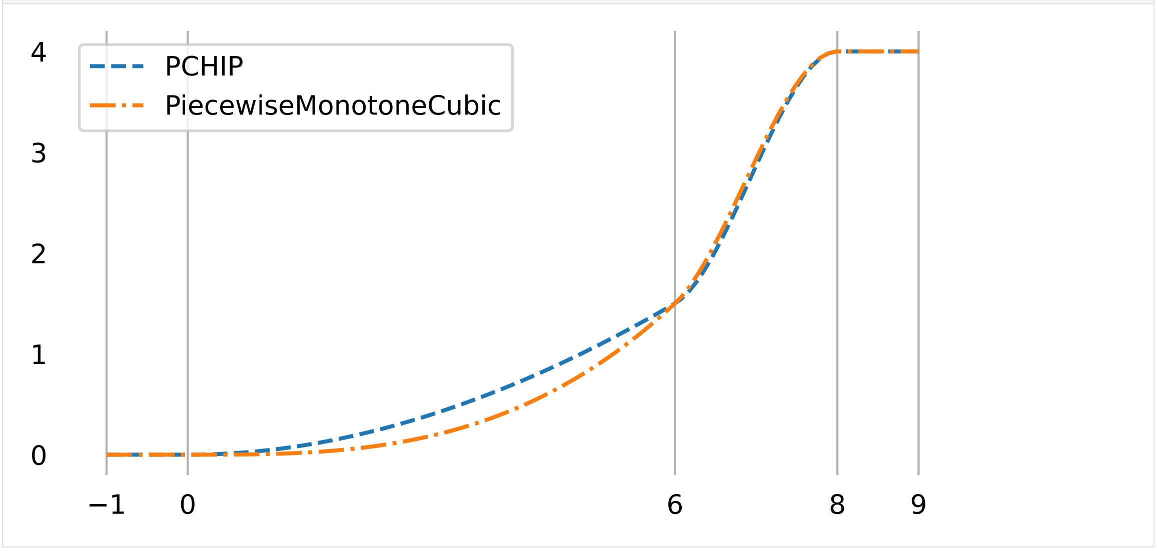
```
[23]: compare_pchip([0, 0, 1.5, 4, 4], [-1, 0, 1, 8, 9])
```

[47] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.pchip_interpolate.html

[48] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PchipInterpolator.html

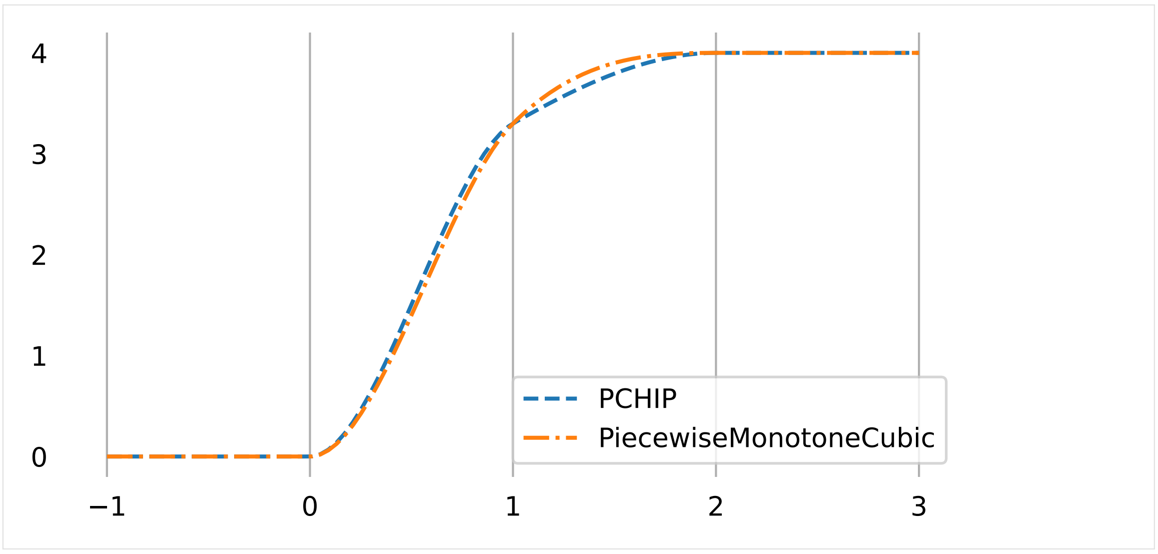


```
[24]: compare_pchip([0, 0, 1.5, 4, 4], [-1, 0, 6, 8, 9])
```

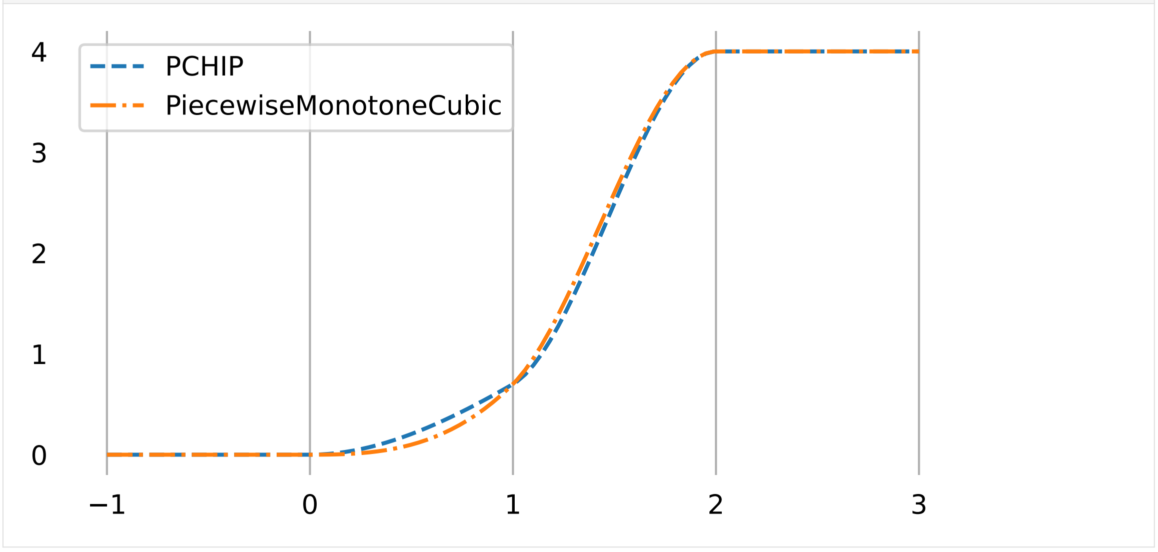


There is even a slight difference in the uniform case:

```
[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 224).

#### B.2.11.5 Monotone Interpolation

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

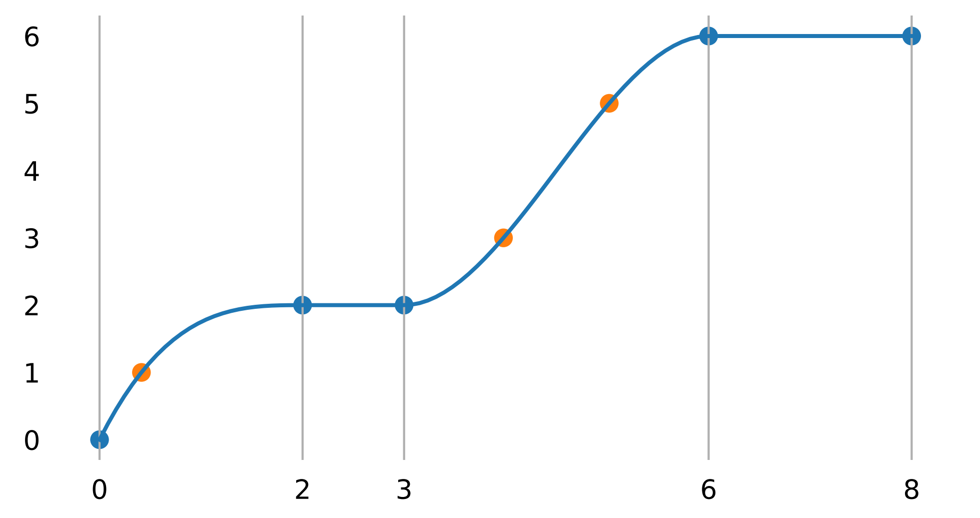
The class *splines.MonotoneCubic* (page 285) works very much the same as *splines.PiecewiseMonotoneCubic* (page 285), except that it only allows monotone data values.

Since the resulting interpolation function is monotone, it can be inverted. Given a function value, the method *.get_time()* (page 286) can be used to find the associated parameter value.

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

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

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

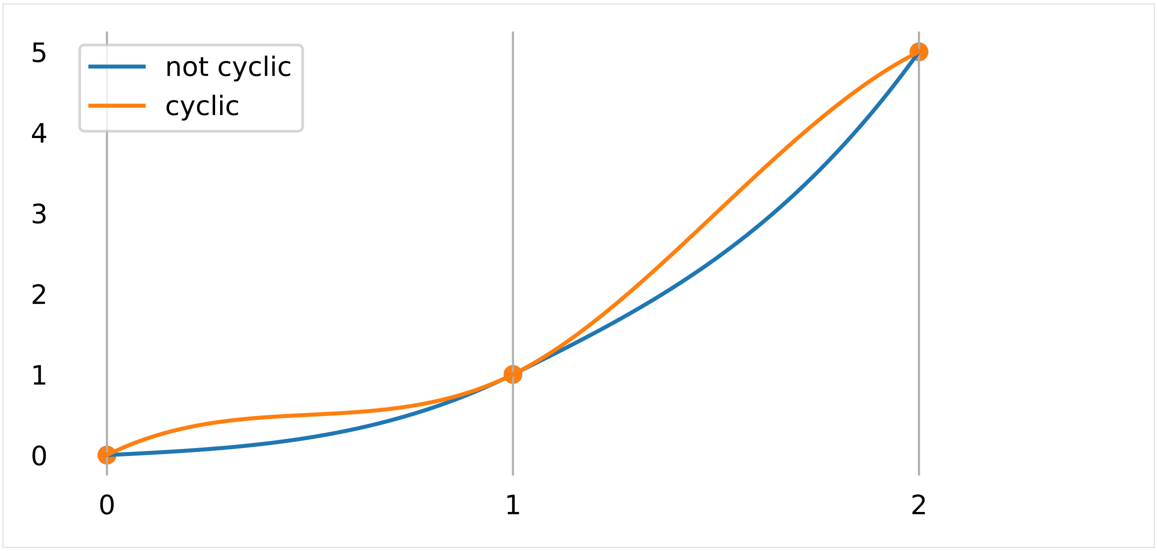
```
The "closed" argument is not allowed
```

However, in some situations it might be useful to automatically infer the same slope at the beginning and end of the spline. This can be achieved with the `cyclic` flag.

[32]:
```
s = splines.MonotoneCubic([0, 1, 5])
```

[33]:
```
s_cyclic = splines.MonotoneCubic([0, 1, 5], cyclic=True)
```

[34]:
```
plot_spline_1d(s, label='not cyclic')
plot_spline_1d(s_cyclic, label='cyclic')
grid_lines(s.grid)
plt.legend();
```



The `cyclic` flag is only allowed if the first and last slope is `None`:

[35]:
```
try:
    splines.MonotoneCubic([0, 1, 5], slopes=[1, None, None], cyclic=True)
except Exception as e:
    print(e)
    assert 'cyclic' in str(e)
else:
    assert False
```

```
If "cyclic", the first and last slope must be None
```

#### B.2.11.6 End Conditions

*The usual end conditions* (page 207) 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.
>
> —Fritsch and Carlson (1980), section 4

Fritsch and Carlson (1980) recommend using the *noncentered three point difference formula*, however, they fail to mention what that actually is. Luckily, we can have a look at the code[49]:

```
C
C  SET D(1) VIA NON-CENTERED THREE-POINT FORMULA, ADJUSTED TO BE
C     SHAPE-PRESERVING.
C
      HSUM = H1 + H2
      W1 = (H1 + HSUM)/HSUM
      W2 = -H1/HSUM
```

(continues on next page)

[49] https://netlib.org/slatec/pchip/dpchim.f

(continued from previous page)

```
      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`.
>
> —Moler (2004), section 3.4

In section 3.6, Moler (2004) 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[50] uses the same implementation (ported to Python)[51].

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.

[36]:
```
def plot_pchip(values, grid, **kwargs):
    pchip = PchipInterpolator(grid, values)
    times = np.linspace(grid[0], grid[-1], 100)
    plt.plot(times, pchip(times), **kwargs)
```

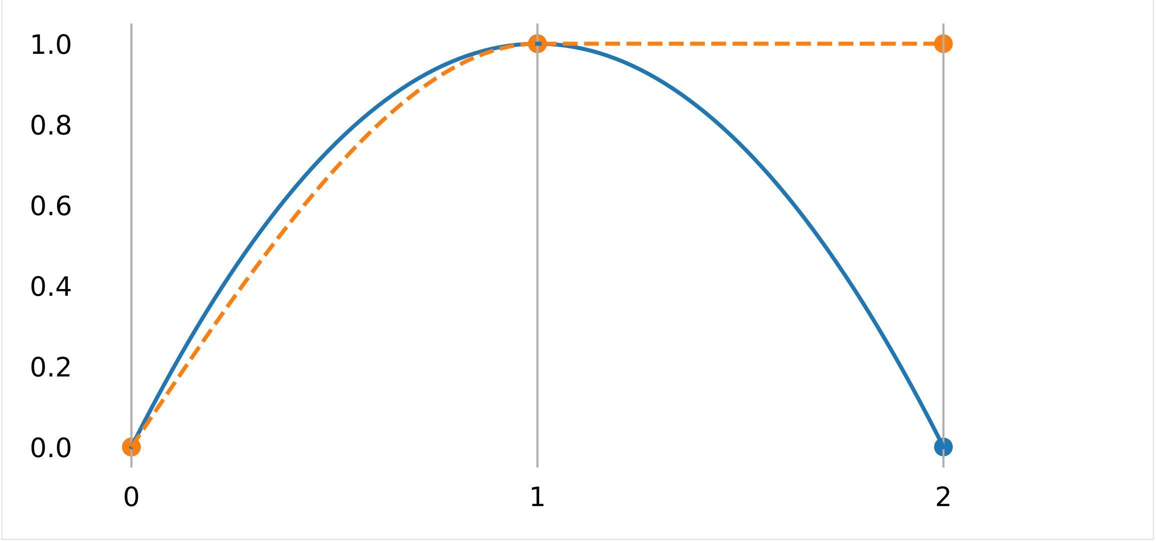
(continues on next page)

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

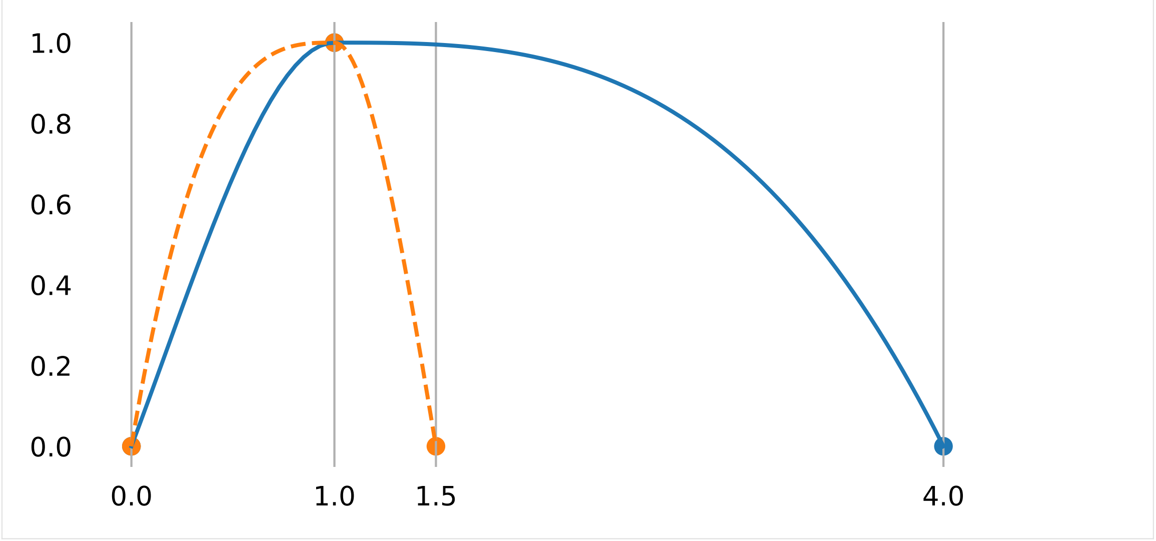
(continued from previous page)

```
    plt.scatter(grid, pchip(grid))
    grid_lines(grid)
```

[37]:
```
plot_pchip([0, 1, 0], [0, 1, 2])
plot_pchip([0, 1, 1], [0, 1, 2], linestyle='--')
grid_lines([0, 1, 2])
```



[38]:
```
plot_pchip([0, 1, 0], [0, 1, 4])
plot_pchip([0, 1, 0], [0, 1, 1.5], linestyle='--')
grid_lines([0, 1, 1.5, 4])
```

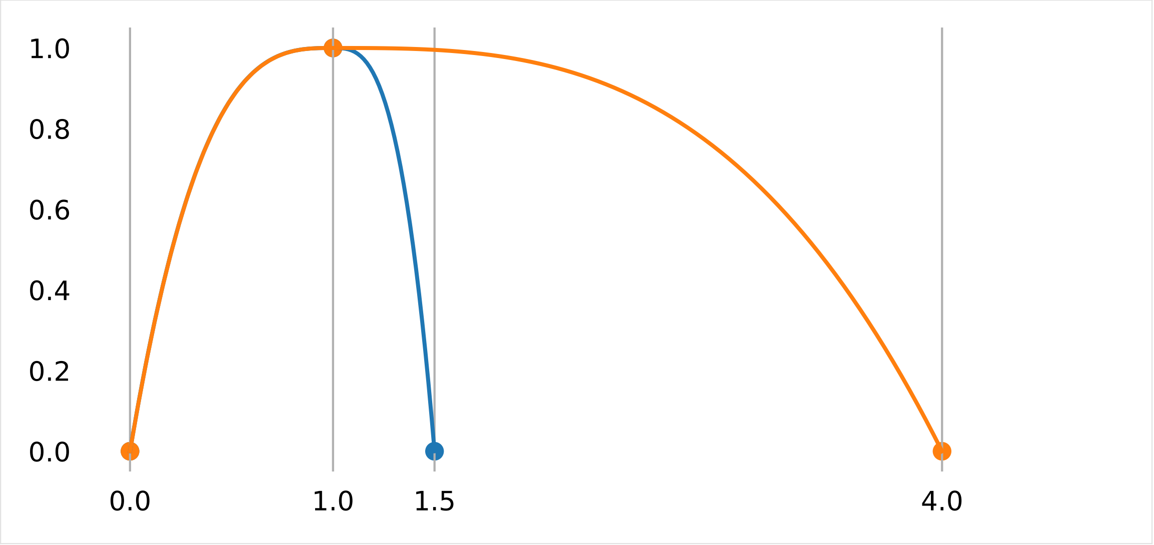


In both of the above examples, the very left slope depends on properties of the very right segment.

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 285) uses end conditions that depend on the slope at $t = 1$, but not explicitly on the shape of the curve at $t > 1$:

[39]:
```
plot_piecewise_monotone([0, 1, 0], grid=[0, 1, 1.5])
plot_piecewise_monotone([0, 1, 0], grid=[0, 1, 4])
grid_lines([0, 1, 1.5, 4])
```



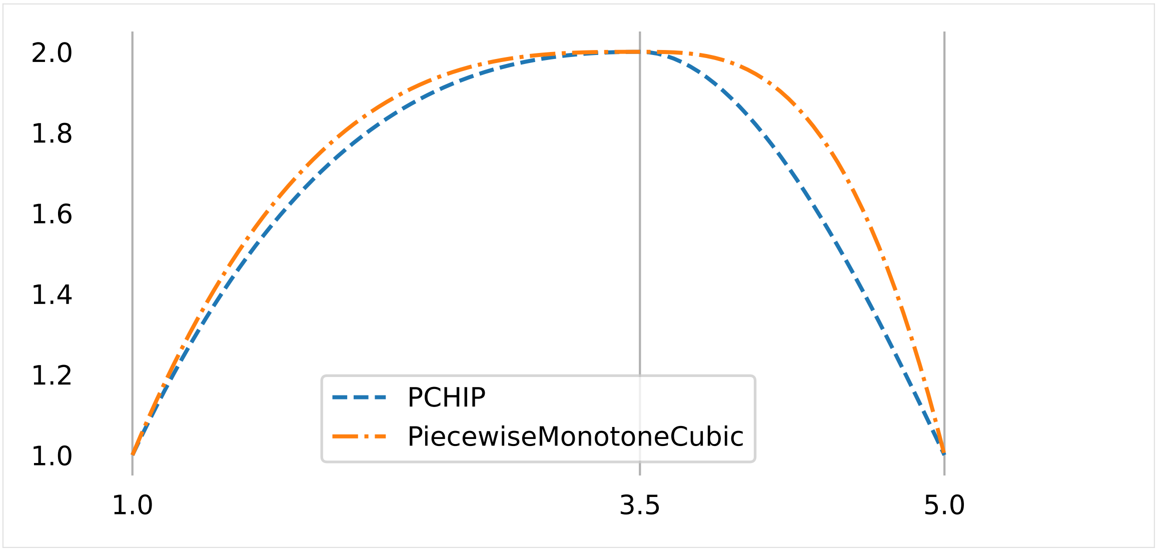
The initial and final slopes of *splines.PiecewiseMonotoneCubic* (page 285) are implemented like this:

[40]:
```
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
```

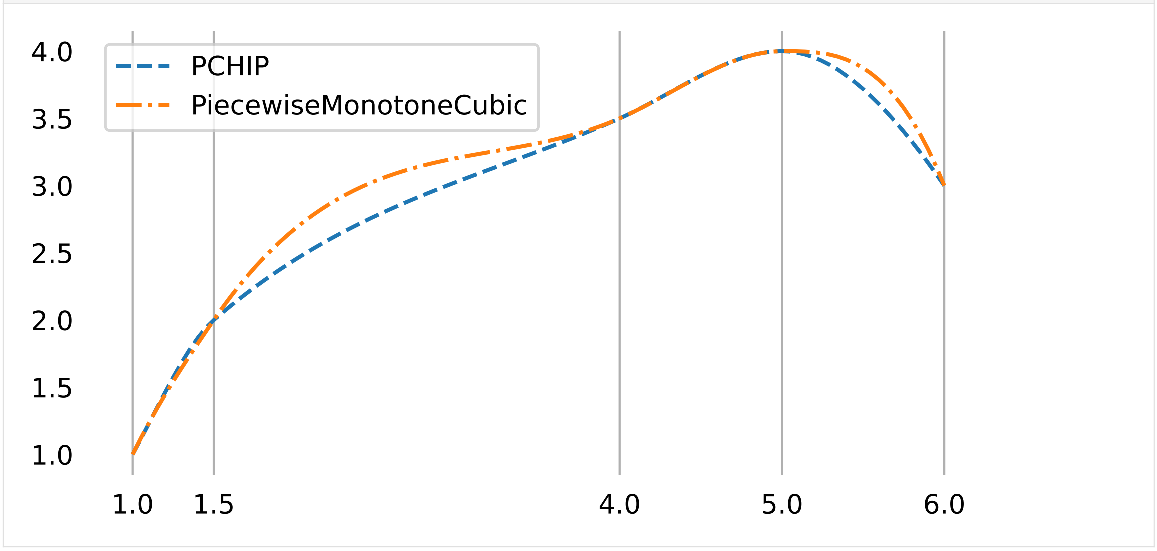
#### B.2.11.7 Even More Examples

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

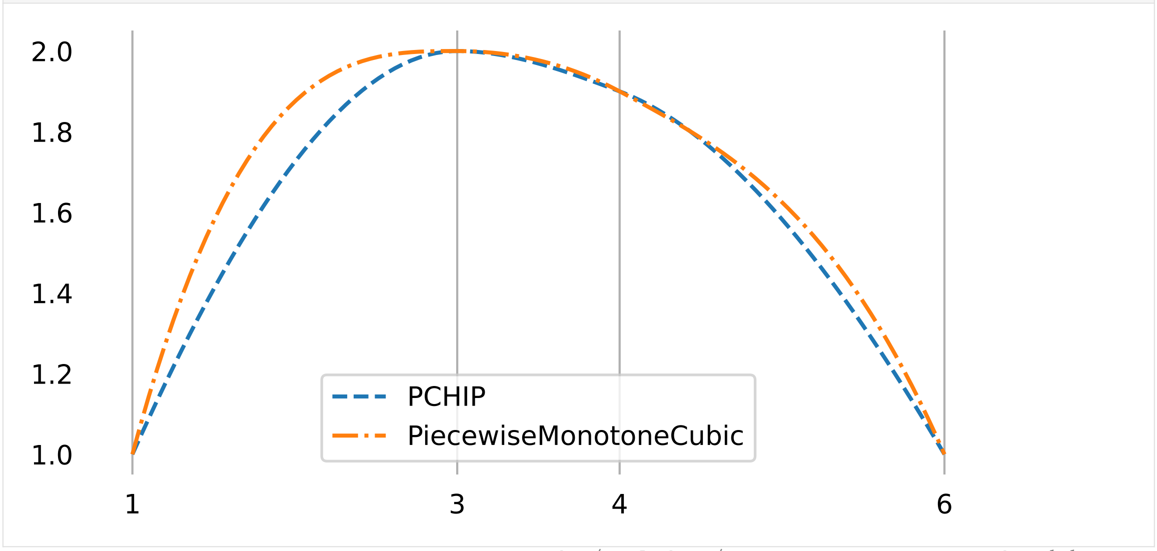
[41]:
```
compare_pchip([1, 2, 1], [1, 3.5, 5])
```



[42]:
```
compare_pchip([1, 2, 3.5, 4, 3], [1, 1.5, 4, 5, 6])
```



[43]:
```
compare_pchip([1, 2, 1.9, 1], [1, 3, 4, 6])
```



........................................................ doc/euclidean/piecewise-monotone.ipynb ends here.

The following section was generated from `doc/euclidean/re-parameterization.ipynb` ........................

## B.2.12 Re-Parameterization

As we have seen previously – for example with *Hermite splines* (page 121) and *Catmull–Rom splines* (page 164) – changing the relative amount of time (or more generally, the relative size of the parameter interval) per spline segment leads to different curve shapes. Given the same underlying polynomials, we cannot simply re-scale the parameter values without affecting the shape of the curve.

However, sometimes we want to keep the shape (or more accurately, the image[52]) of a curve intact and only change its timing.

This can be done by introducing a function that maps from a new set of parameter values to the parameter values of the original spline.

### B.2.12.1 Arc-Length Parameterization

Instead of using a curve $\boldsymbol{x}(t)$ with a free parameter $t$ (which we often interpret as time), it is sometimes useful to have a curve $\boldsymbol{x}_{\text{arc}}(s)$ with the same image but where the parameter $s$ represents the distance travelled since the beginning of the curve. The length of a piece of curve is called arc length[53] and therefore $\boldsymbol{x}_{\text{arc}}(s)$ is called *arc-length parameterized*. Sometimes, this is also called "natural" parameterization – not to be confused with *natural splines* (page 124) and *natural end conditions* (page 207).

An interesting (and slightly confusing) thing to do now, is to use $\boldsymbol{x}_{\text{arc}}(s)$ with time as a parameter. Note that the speed along a curve is calculated as distance per time interval ($v = \frac{ds}{dt}$), but if time and distance are the same ($s \equiv t$), we get a constant speed $v = \frac{ds}{ds} = 1$. In other words, the tangent vector of an arc-length parameterized curve always has unit length.

To turn an existing curve $\boldsymbol{x}(t)$ into its arc-length parameterized counterpart $\boldsymbol{x}_{\text{arc}}(s)$, we need the parameter $t$ as function of travelled distance $s$, i.e. $t(s)$:

$$\boldsymbol{x}_{\text{arc}}(s) = \boldsymbol{x}(t(s))$$

Sadly, we don't know $t(s)$, but we can find $s(t)$ and then try to find the inverse function.

Let's look at the tangent vector $\frac{d}{d\tau}\boldsymbol{x}(\tau)$ (i.e. the velocity) at every infinitesimally small time interval $d\tau$. The length travelled along the curve in that time interval is the length of the tangent vector $\left|\frac{d}{d\tau}\boldsymbol{x}(\tau)\right|$ (i.e. the speed) multiplied by the time interval $d\tau$. Adding all these small pieces from $t_0$ to $t$ results in the arc length

$$s(t) = \int_{t_0}^{t} \left|\frac{d}{d\tau}\boldsymbol{x}(\tau)\right| d\tau.$$

[52] `https://en.wikipedia.org/wiki/Image_(mathematics)`
[53] `https://en.wikipedia.org/wiki/Arc_length`

This looks straightforward enough, but it turns out that this integral cannot be solved analytically if $\boldsymbol{x}(t)$ is cubic (or of higher degree). The reason for that is the Abel–Ruffini theorem[54].

We'll have to use numerical integration[55] instead.

Finally, we need to invert this function. In other words, given an arc length $s$, we have to provide a way to obtain the corresponding $t$. This can be reduced to a root finding problem, which can be solved with different numerical methods, for example with the bisection method[56].

Arc-length re-parameterization is implemented in the Python class *splines.UnitSpeedAdapter* (page 286). This is using scipy.integrate.quad()[57] for numerical integration and scipy.optimize.bisect()[58] for root finding.

Let's show an example spline using the vertices from the *section about centripetal parameterization* (page 163):

```
[1]: points4 = [
         (0, 0),
         (0, 0.5),
         (1.5, 1.5),
         (1.6, 1.5),
         (3, 0.2),
         (3, 0),
     ]
```

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

First we create a centripetal Catmull–Rom spline ...

```
[3]: s1 = splines.CatmullRom(points4, alpha=0.5, endconditions='closed')
```

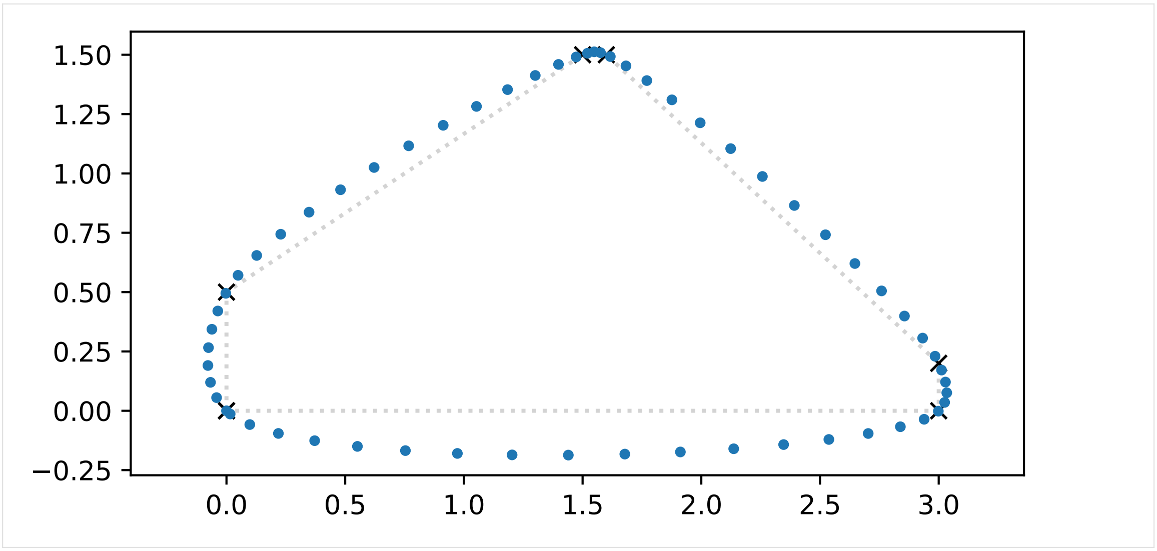
... which we then convert to an arc-length parameterized spline:

```
[4]: s2 = splines.UnitSpeedAdapter(s1)
```

```
[5]: %%time
     plot_spline_2d(s1, dots_per_second=10)
```

```
CPU times: user 19.4 ms, sys: 0 ns, total: 19.4 ms
Wall time: 19.2 ms
```

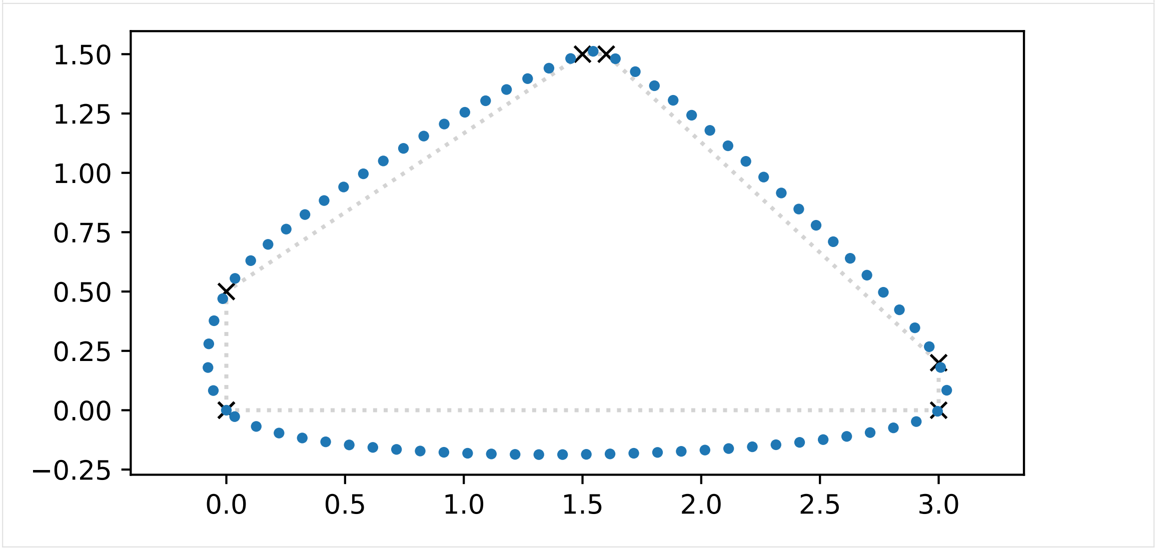
[54] https://en.wikipedia.org/wiki/Abel-Ruffini_theorem
[55] https://en.wikipedia.org/wiki/Numerical_integration
[56] https://en.wikipedia.org/wiki/Bisection_method
[57] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html
[58] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.bisect.html



Evaluating the arc-length parameterized spline takes quite a bit longer:

```
[6]: %%time
plot_spline_2d(s2, dots_per_second=10)
```

```
CPU times: user 1.89 s, sys: 32.2 ms, total: 1.92 s
Wall time: 1.88 s
```



We are plotting 10 dots per second, and we can count about 10 dots per unit of distance, which confirms that the spline has a speed of 1.

#### B.2.12.2 Spline-Based Re-Parameterization

We can choose any function to map a new parameter to old parameter values. Since we are already talking about splines, we might as well use a one-dimensional spline. To rule out backwards movement along the original spline, we should use use a *monotone spline* (page 222) as implemented, for example, in the class *splines.MonotoneCubic* (page 285).

A tool for re-parameterizing an existing spline is available in the class *splines.NewGridAdapter* (page 286).

This is especially useful when applied to an already arc-length parameterized spline, because then the slope of the parameter re-mapping function directly corresponds to the speed along the spline.

Not all new parameter values have to be explicitly given. If unspecified, they are interpolated from the surrounding values.

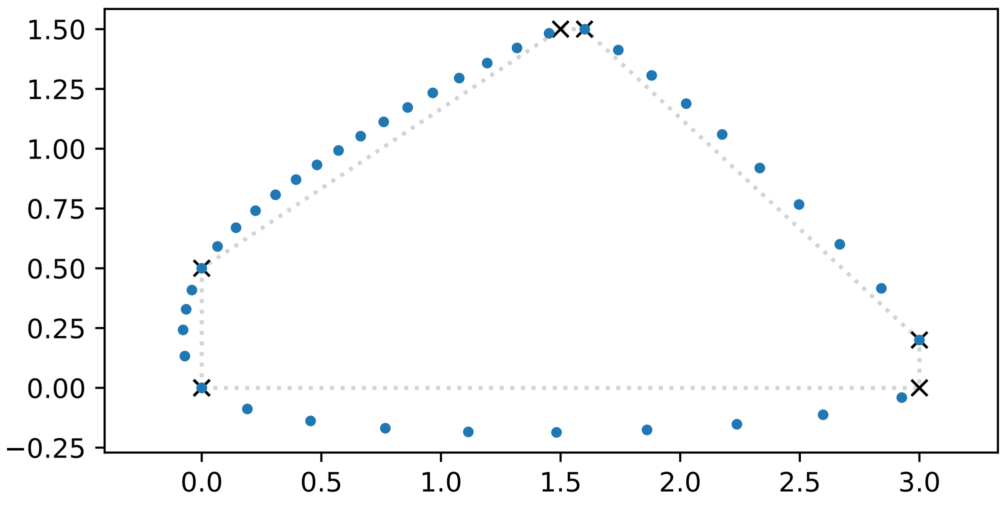
For closed curves it might be useful to have the same slope at the beginning and the end of the spline. This can be achieved by using `cyclic=True`.

[7]:
```
new_grid = [-1, -0.5, None, None, 2, None, 3]
s3 = splines.NewGridAdapter(s2, new_grid, cyclic=True)
s3.grid
```

[7]:
```
[-1, -0.5, 1.0334250405837568, 1.0992464899992576, 2, 2.0730953134961054,␣
↪3]
```

[8]:
```
%%time
plot_spline_2d(s3, dots_per_second=10)
```

```
CPU times: user 990 ms, sys: 92.9 ms, total: 1.08 s
Wall time: 962 ms
```



.................................................. doc/euclidean/re-parameterization.ipynb ends here.

## B.3 Rotation Splines

There are many ways to implement rotation splines. Here we use *unit quaternions* to represent rotations. First, we'll show what *quaternions* are, and how their subset of *unit quaternions* can be used to handle rotations. Based on a special form of linear interpolation called *Slerp* (page 240), we then use several algorithms that we have seen in *the section about Euclidean splines* (page 88) – which all utilize linear interpolations (and extrapolations) – to implement rotation splines. In the end of this section, we present a few methods which are *not* based on Slerp, but it will turn out that they all have severe limitations.

The following section was generated from `doc/rotation/quaternions.ipynb` ...................................

### B.3.1 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[59] 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 noncommutative[60]).

A Python implementation is available in the class *splines.quaternion.Quaternion* (page 287).

#### B.3.1.1 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 is important to note that the order in which the *basic quaternions* $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ are multiplied matters: $\mathbf{ij} = \mathbf{k}$, $\mathbf{ji} = -\mathbf{k}$ (i.e. their multiplication is anticommutative[61]). 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 287), these representations are available via the attributes *scalar* (page 287), *vector* (page 287), *wxyz* (page 288) and *xyzw* (page 288).

[59] `https://en.wikipedia.org/wiki/Quaternion`
[60] `https://en.wikipedia.org/wiki/Noncommutative`
[61] `https://en.wikipedia.org/wiki/Anticommutative_property`

There are even more ways to represent quaterions, for example as 2x2 complex matrices or as 4x4 real matrices (McDonald 2010).

### B.3.1.2 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 288).

### B.3.1.3 Unit Quaternions as Rotations

Given a (normalized) rotation axis $\vec{n}$ and a rotation angle $\alpha$ (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)[62]), which means that each rotation can be associated with two distinct quaternions. More specifically, the antipodal points $q$ and $-q$ represent the same rotation – see *Negation* (page 239) below.

More details can be found on Wikipedia[63].

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
```

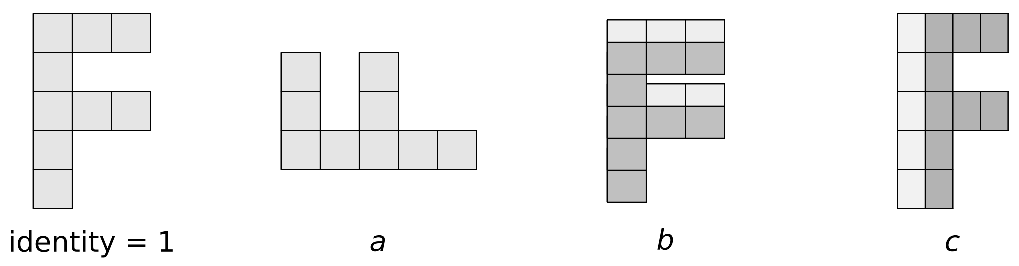
The quaternion 1 represents "no rotation at all".

```
[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)
```

[62] https://en.wikipedia.org/wiki/3D_rotation_group
[63] https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation

[4]:
```
plot_rotation({
    'identity = 1': identity,
    '$a$': a,
    '$b$': b,
    '$c$': c,
});
```



### B.3.1.4 Axes Conventions

When converting between rotation angles (see Euler/Tait–Bryan angles[64]) and unit quaternions, we can freely choose from a multitude of axes conventions[65]. 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 reader. The helper function `angles2quat()` takes three angles (in degrees) which are applied in this order:

- *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[66].

### B.3.1.5 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

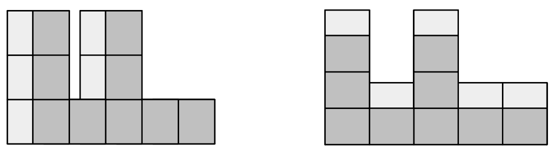
[64] `https://en.wikipedia.org/wiki/Euler_angles`
[65] `https://en.wikipedia.org/wiki/Axes_conventions`
[66] `https://en.wikipedia.org/wiki/Right-hand_rule#Rotations`

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});
```



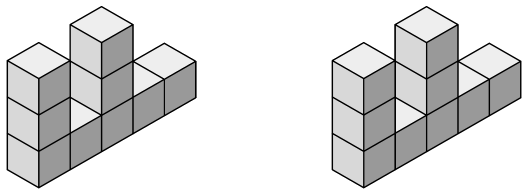
However, there is an exception when all rotation axes are the same, in which case the rotation angles can simply be added (in arbitrary order, of course).

The quaternion $1 = (1, (0, 0, 0))$ is the identity element with regards to quaternion multiplication. A multiplication with this (on either side) leads to an unchanged rotation.

Even though quaternion multiplication is *non-commutative*, it is still associative[67], which means that if there are multiple multiplications in a row, they can be grouped arbitrarily, leading to the same overall result:

$$(q_1 q_2) q_3 = q_1 (q_2 q_3)$$

```
[6]: plot_rotation({'$(bc)a$': (b * c) * a, '$b(ca)$': b * (c * a)});
```

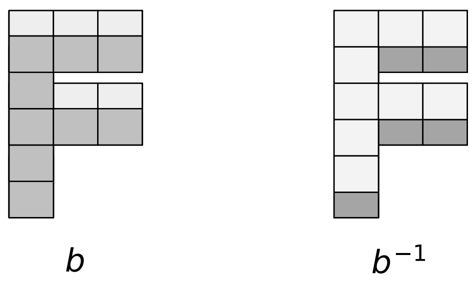


### B.3.1.6 Inverse

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).

[67] https://en.wikipedia.org/wiki/Associative_property

```
[7]: plot_rotation({'$b$': b, '$b^{-1}$': b.inverse()});
```



By multiplying a rotation with its inverse, the original rotation can be undone: $qq^{-1} = q^{-1}q = 1$. 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 $\bar{q}$. 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).

$$\bar{q} = \left(w, -\vec{v}\right) = \left(\cos\frac{\alpha}{2}, -\vec{n}\sin\frac{\alpha}{2}\right) = \left(\cos\frac{-\alpha}{2}, \vec{n}\sin\frac{-\alpha}{2}\right)$$

### B.3.1.7 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:

$$q_{0,1} = q_1{q_0}^{-1}$$

### B.3.1.8 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 ${q_0}^{-1}$:

$$q_0{}^{-1} q_0 q_{0,1} = q_0{}^{-1} q_1$$

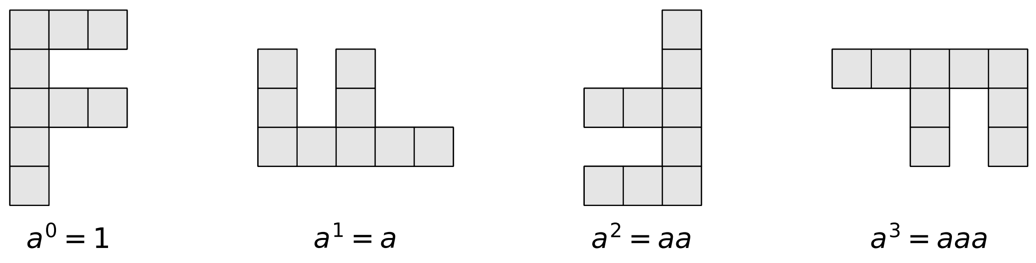
$q_0{}^{-1} q_0$ cancels out and we get:

$$q_{0,1} = q_0{}^{-1} q_1$$

#### B.3.1.9 Exponentiation

Raising a unit quaternion to an integer power simply means applying the same rotation multiple times:

[8]:
```
plot_rotation({
    '$a^0 = 1$': a**0,
    '$a^1 = a$': a**1,
    '$a^2 = aa$': a**2,
    '$a^3 = aaa$': a**3,
});
```

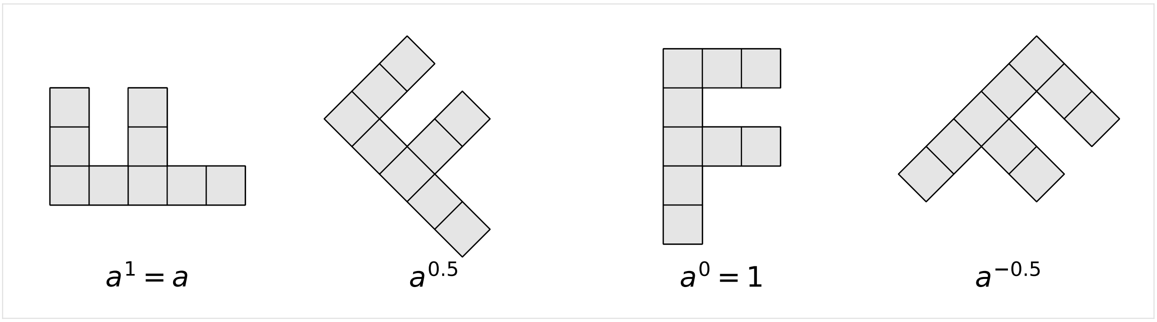


It shouldn't come as a surprise that $q^0 = 1$ and $q^1 = q$.

Using an exponent of $-1$ is equivalent to taking the inverse – see *above* (page 236). 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.

$$q^k = \left(\cos\frac{k\alpha}{2}, \vec{n}\sin\frac{k\alpha}{2}\right)$$

[9]:
```
plot_rotation({
    '$a^1 = a$': a**1,
    '$a^{0.5}$': a**0.5,
    '$a^0 = 1$': a**0,
    '$a^{-0.5}$': a**-0.5,
});
```

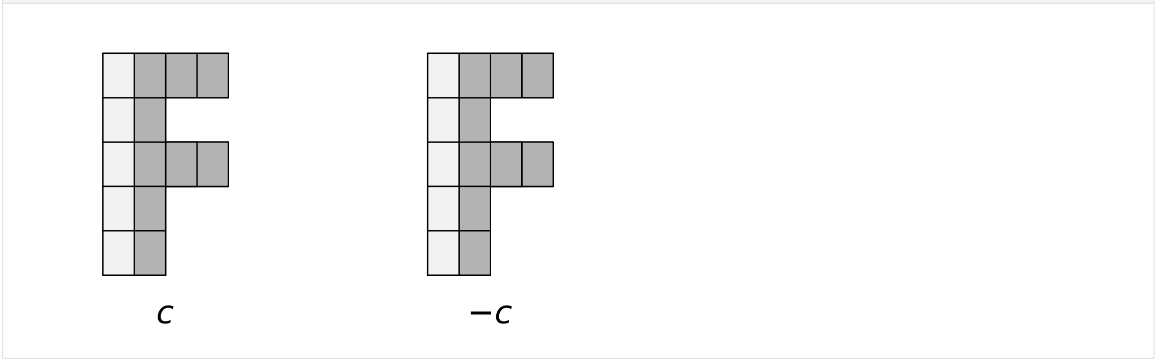


### B.3.1.10 Negation

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.

```
[10]: plot_rotation({'$c$': c, '$-c$': -c});
```



One way to negate the scalar part of a unit quaternion is to add $\pi$ 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 $2\pi$ to the rotation angle $\alpha$, which brings us back to the original rotation. Adding $2\pi$ 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.

$$-q = \left(-w, -\vec{v}\right) = \left(\cos\frac{\alpha + 2\pi}{2}, \vec{n}\sin\frac{\alpha + 2\pi}{2}\right)$$

### B.3.1.11 Canonicalization

When we are given multiple rotations and we want to represent them as quaternions, we have to take care of the ambiguity caused by the double cover property – see *Slerp Visualization* (page 241) for an example of this ambiguity.

One way to do that is to make sure that in a sequence of rotations (which we want to use as the control points of a spline, for example), the angle (in 4D space) between neighboring quaternions is at most 90 degrees (which corresponds to a 180 degree rotation in 3D space). For any pair of quaternions where this is not the case, one of the quaternions can simply be negated. The function *splines.quaternion.canonicalized*() (page 289) can be used to create an iterator of canonicalized quaternions from an iterable of arbitrary quaternions.

............................................................ `doc/rotation/quaternions.ipynb` ends here.

The following section was generated from `doc/rotation/slerp.ipynb` ........................................

## B.3.2 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 Shoemake (1985), 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[68] provides four equivalent ways to describe the same thing:

$$\begin{aligned}
\operatorname{Slerp}(q_0, q_1; t) &= q_0 \left({q_0}^{-1} q_1\right)^t \\
&= q_1 \left({q_1}^{-1} q_0\right)^{1-t} \\
&= \left(q_0 {q_1}^{-1}\right)^{1-t} q_1 \\
&= \left(q_1 {q_0}^{-1}\right)^t q_0
\end{aligned}$$

Shoemake (1985) 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 ...

[1]:
```python
def slerp(one, two, t):
    """Spherical Linear intERPolation."""
    return (two * one.inverse())**t * one
```

[68] `https://en.wikipedia.org/wiki/Slerp#Quaternion_Slerp`

... but for your convenience an implementation is also provided in *splines.quaternion.slerp*() (page 289).

### B.3.2.1 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 $q$.

$$\begin{aligned} 1 &= (1, (0, 0, 0)) \\ q &= \left(\cos\frac{\alpha}{2}, \vec{n}\sin\frac{\alpha}{2}\right) \end{aligned}$$

To move along the great arc from $1$ to $q$, we simply have to change the angle from $0$ to $\alpha$ while the rotation axis $\vec{n}$ stays unchanged.

$$\operatorname{Slerp}(1, q; t) = \left(\cos\frac{\alpha t}{2}, \vec{n}\sin\frac{\alpha t}{2}\right) = q^t, \text{ where } 0 \le t \le 1$$

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 237) $q_{0,1}$:

$$\operatorname{Slerp}(q_0, q_1; t) = \operatorname{Slerp}(1, q_{0,1}; t)\, q_0$$

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 237) $q_{0,1} = {q_0}^{-1} q_1$:

$$\operatorname{Slerp}(q_0, q_1; t) = q_0 \operatorname{Slerp}(1, q_{0,1}; t) = q_0\, \left({q_0}^{-1} q_1\right)^t$$

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.

### B.3.2.2 Visualization

First, let's import NumPy[69] ...

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

... and a few helper functions from `helper.py`:

[69] `https://numpy.org/`

[3]:
```
from helper import angles2quat, animate_rotations, display_animation
```

We can now define two example quaternions:

[4]:
```
q1 = angles2quat(45, -20, -60)
q2 = angles2quat(-45, 20, 30)
```

Just out of curiosity, let's use the method *rotation_to*() (page 289) 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 degrees, we know that we will get the smallest difference in rotation. If it is larger than 180 degrees, we can negate the second quaternion to get a smaller rotation – see *canonicalization* (page 239).

[6]:
```
ani_times = np.linspace(0, 1, 50)
```

We show both the original target quaternion and its antipodal point in this animation:

[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')
```

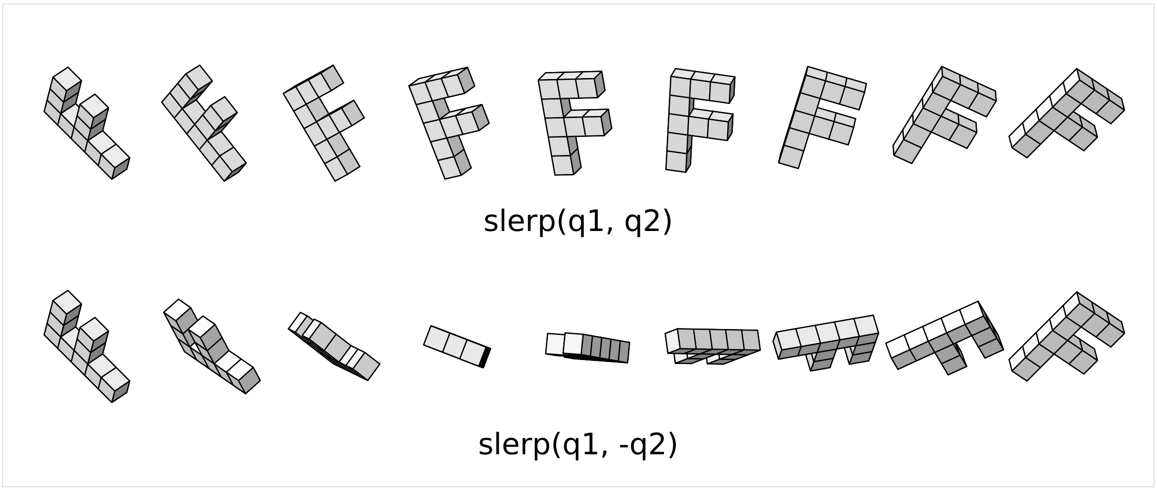
```
Animations can only be shown in HTML output, sorry!
```

Let's create some still images as well:

[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))
```



`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).

### B.3.2.3 Piecewise Slerp

The class *PiecewiseSlerp* (page 290) 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.

### B.3.2.4 Slerp vs. Nlerp

While *Slerp* interpolates along a great arc between two quaternions, it is also possible to interpolate along a straight line (in four-dimensional quaternion space) between those two quaternions. The resulting interpolant is *not* part of the unit hypersphere, i.e. the interpolated values are *not* unit quaternions. However, they can be normalized to become unit quaternions. This is called "**n**ormalized **l**inear int**erp**olation", in short *Nlerp*. The resulting interpolant travels through the same quaternions as Slerp does, but it doesn't do it with constant angular velocity.

[16]:
```
from splines.quaternion import Quaternion
```

[17]:
```
def lerp(one, two, t):
    """Linear intERPolation."""
    one = np.asarray(one)
    two = np.asarray(two)
    return (1 - t) * one + t * two
```

[18]:
```
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 degrees ...

[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)
```

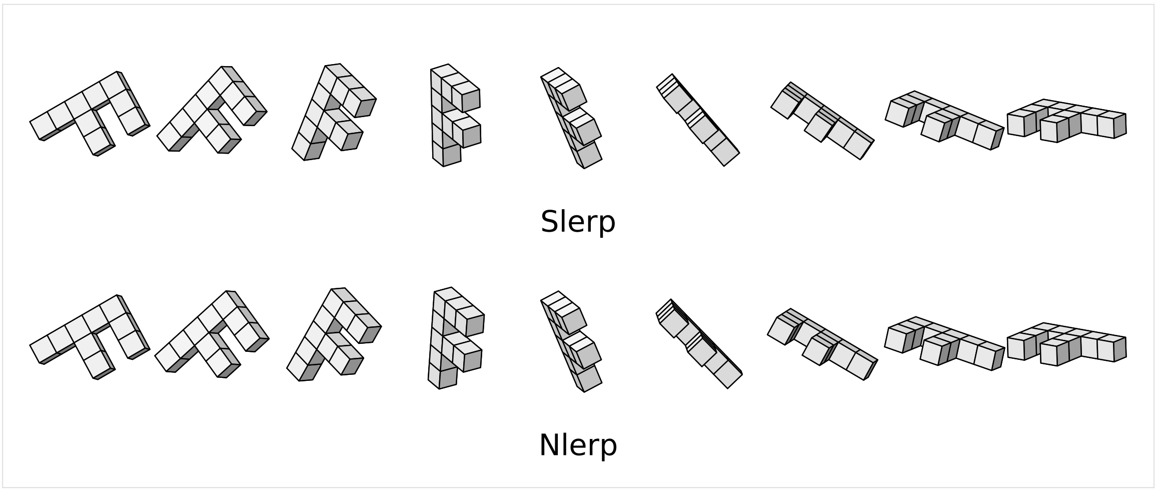
[23]:
```
ani = animate_rotations({
    'Slerp': slerp(q1, q2, ani_times),
    'Nlerp': nlerp(q1, q2, ani_times),
})
```

[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!

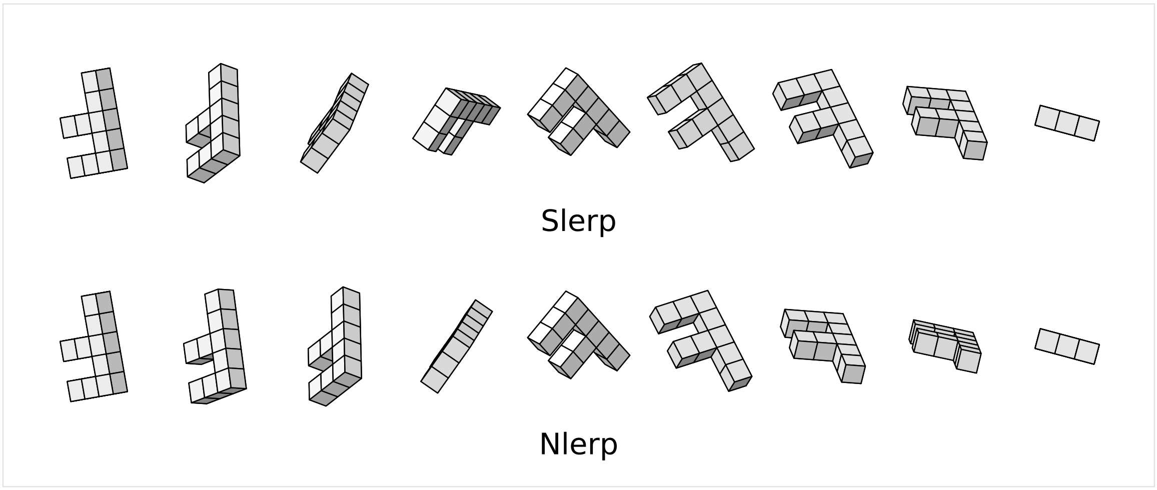
```
[28]: assert q3.dot(q4) < 0
```

```
[29]: ani = animate_rotations({
          'Slerp': slerp(q3, q4, ani_times),
          'Nlerp': nlerp(q3, q4, ani_times),
      })
```

```
[30]: display_animation(ani, default_mode='reflect')
```

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

```
[31]: plot_rotations({
          'Slerp': slerp(q3, q4, plot_times),
          'Nlerp': nlerp(q3, q4, 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 degrees anyway, so this might not be relevant.

.......... `doc/rotation/slerp.ipynb` ends here.

The following section was generated from `doc/rotation/de-casteljau.ipynb` ..........

## B.3.3 De Casteljau's Algorithm With Slerp

Shoemake (1985), who famously introduced quaternions to the field of computer graphics, suggests to apply a variant of *De Casteljau's Algorithm* (page 136) to a unit quaternion control polygon, using *Slerp* (page 240) instead of linear interpolations.

[1]:
```
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]:
```
import numpy as np
from helper import angles2quat, plot_rotation, plot_rotations
from helper import animate_rotations, display_animation
```

### B.3.3.1 "Cubic"

Shoemake (1985) 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)
```

(continues on next page)

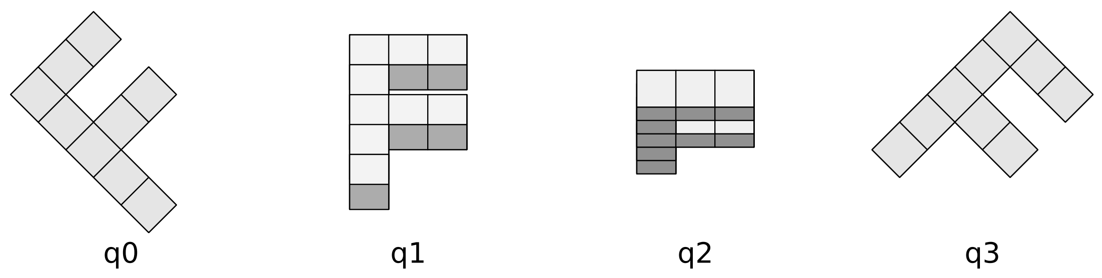
(continued from previous page)

```
    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,
    )
```

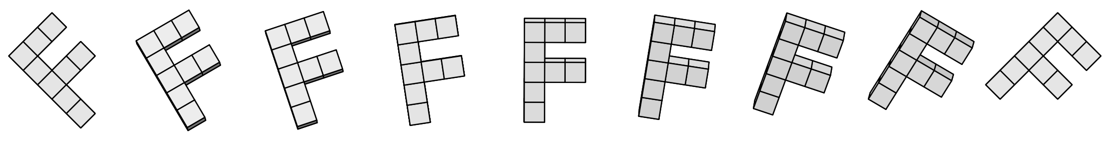
To illustrate this, let's define 4 unit quaternions that we can use as control points:

```
[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!
```

### B.3.3.2 Arbitrary "Degree"

The class *splines.quaternion.DeCasteljau* (page 290) 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!
```

### B.3.3.3 Constant Angular Speed

Is there a way to construct a curve parameterized by arc length? This would be very useful.

—Shoemake (1985), section 6: "Questions"

Remember *arc-length parameterization of Euclidean splines* (page 229)? We used the class *splines.UnitSpeedAdapter* (page 286) which happens to be implemented in a way that it is also usable for rotation splines, how convenient! The only requirement is that the second derivative of the wrapped spline yields an angular velocity vector, which is nothing else than the instantaneous rotation axis scaled by the angular speed.

[13]:
```
from splines import UnitSpeedAdapter
```

[14]:
```
s1 = DeCasteljau([[
    angles2quat(90, 0, 0),
    angles2quat(0, -45, 90),
    angles2quat(0, 0, 0),
    angles2quat(180, 0, 180),
]])
```

[15]:
```
s2 = UnitSpeedAdapter(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!
```

### B.3.3.4 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 249).

.................................................................... doc/rotation/de-casteljau.ipynb ends here.

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

## B.3.4 Uniform Catmull–Rom-Like Quaternion Splines

We have seen how to use *De Casteljau's algorithm with Slerp* (page 246) 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 174) 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 (non-zero length) 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 237) $q_{i,\text{offset}}$:

$$\tilde{q}_i^{(+)} = {q_{i,\text{offset}}}^{\frac{1}{3}} q_i$$
$$\tilde{q}_i^{(-)} = {q_{i,\text{offset}}}^{-\frac{1}{3}} q_i$$

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 ...

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

... 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)
```

### B.3.4.1 Relative Rotations

As a first attempt, we can try to "translate" the equation ...

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

... 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 237) associated with the the incoming and the outgoing chord:

$$q_\text{in} = q_i {q_{i-1}}^{-1}$$
$$q_\text{out} = q_{i+1} {q_i}^{-1}$$

```
[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 …

$$q_{i,\text{offset}} \overset{?}{=} \left(q_\text{in} q_\text{out}\right)^{\frac{1}{2}}$$

```
[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 …

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

… 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 $q_\text{in}$ and $q_\text{out}$ 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:

$$q_{i,\text{offset}} \overset{?}{=} \left(\left({q_\text{out}}^{\frac{1}{10}} {q_\text{in}}^{-\frac{1}{10}}\right)^{\frac{1}{2}} {q_\text{in}}^{\frac{1}{10}}\right)^{10} = \left(\left({q_\text{in}}^{\frac{1}{10}} {q_\text{out}}^{-\frac{1}{10}}\right)^{\frac{1}{2}} {q_\text{out}}^{\frac{1}{10}}\right)^{10}$$

```
[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. 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.

#### B.3.4.2 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 flat, 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 ...

$$q_{i,\text{offset}} \overset{?}{=} \exp\left(\frac{\ln(q_\text{in}) + \ln(q_\text{out})}{2}\right)$$

```
[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 291) class.

Let's compare this to the variants from the previous section:

```
[23]: max(map(abs, (offset_l - offset_f).xyzw))
[23]: 0.017423237526556398
```

```
[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.

### B.3.4.3 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 290) class to create a Bézier-like curve from the given control points, using *canonicalized*() (page 289) 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, so let's 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!
```

#### B.3.4.4 Shoemake's Approach

In section 4.2, Shoemake (1985) provides two function definitions:

$$\begin{aligned}\operatorname{Double}(p,q) &= 2(p\cdot q)q - p\\ \operatorname{Bisect}(p,q) &= \frac{p+q}{\|p+q\|}\end{aligned}$$

[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.
>
> —Shoemake (1985), 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 287) result into a *UnitQuaternion* (page 288).

[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 $q_{n+1}$.

—Shoemake (1985), 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 ...................

### B.3.5 Non-Uniform Catmull–Rom-Like Rotation Splines

What is the best way to allow varying intervals between sequence points in parameter space?

—Shoemake (1985), section 6: "Questions"

In the *uniform case* (page 249) we have used *De Casteljau's algorithm with Slerp* (page 246) 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 148). This is implemented in the class *splines.quaternion.DeCasteljau* (page 290).

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 179) 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 237) $\delta_i = q_{i+1} {q_i}^{-1}$ we can try to "translate" this to quaternions (using some vector operations in the tangent space):

$$
\begin{aligned}
\vec{\rho}_i &= \frac{\ln(\delta_i)}{t_{i+1} - t_i}\\
\vec{\omega}_i &= \frac{(t_{i+1} - t_i)\,\vec{\rho}_{i-1} + (t_i - t_{i-1})\,\vec{\rho}_i}{t_{i+1} - t_{i-1}}\\
\tilde{q}_i^{(+)} &\overset{?}{=} \exp\left(\frac{t_{i+1} - t_i}{3}\,\vec{\omega}_i\right) q_i\\
\tilde{q}_i^{(-)} &\overset{?}{=} \exp\left(\frac{t_i - t_{i-1}}{3}\,\vec{\omega}_i\right)^{-1} q_i,
\end{aligned}
$$

where $\vec{\rho}_i$ 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 Bézier-like 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 class *splines.quaternion.CatmullRom* (page 291).

To illustrate this, let's load NumPy, a few helpers from `helper.py` and *splines.quaternion.canonicalized*() (page 289).

[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]
```

(continues on next page)

(continued from previous page)

```
    extended_grid = [grid[0] - last_interval, *grid, grid[-1] + first_
↪interval]
    control_points = []
    for qs, ts in zip(
            zip(rotations, rotations[1:], rotations[2:]),
            zip(extended_grid, extended_grid[1:], extended_grid[2:])):
        q_before, q_after = 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 = 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!
```

### B.3.5.1 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 249), 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 163), 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 163), 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),
    'centripetal': evaluate(centripetal),
    'chordal': evaluate(chordal),
})
```

[19]:
```
display_animation(ani, default_mode='loop')
```

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

The class *splines.quaternion.CatmullRom* (page 291) provides a parameter `alpha` that allows arbitrary parameterization between *uniform* and *chordal* – see also *parameterized parameterization of Euclidean splines* (page 164).

.................................................. `doc/rotation/catmull-rom-non-uniform.ipynb` ends here.

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

### B.3.6 Kochanek–Bartels-like Rotation Splines

Remember *Kochanek–Bartels splines in Euclidean space* (page 191)? We can try to "translate" those to quaternions by using *De Casteljau's algorithm with Slerp* (page 246). 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 256).

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 201) 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 179):

$$\begin{aligned}
\tilde{\boldsymbol{x}}_i^{(+)} &= \boldsymbol{x}_i + \frac{(t_{i+1} - t_i)}{3}\dot{\boldsymbol{x}}_i^{(+)}\\
\tilde{\boldsymbol{x}}_i^{(-)} &= \boldsymbol{x}_i - \frac{(t_i - t_{i-1})}{3}\dot{\boldsymbol{x}}_i^{(-)}
\end{aligned}$$

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 237) 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 class *splines.quaternion.KochanekBartels* (page 290).

[1]:
```python
from splines.quaternion import KochanekBartels
```

### B.3.6.1 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 191).

As so often, we import NumPy and a few helpers from `helper.py`:

[2]:
```python
import numpy as np
from helper import angles2quat, animate_rotations, display_animation
```

Let's define a few example rotations …

[3]:
```python
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]:
```python
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 240) – segments.

[10]:
```
show_tcb({
    'Catmull-Rom': [0, 0, 0],
    '2 linear segments': [
        (0, 0, 1),
        (0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
        (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 ....................

## B.3.7 "Natural" End Conditions

In the *notebook about "natural" end conditions for Euclidean splines* (page 213) we have derived the following equations for calculating the second and penultimate control points of cubic Bézier splines:

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

These equations can be "translated" to quaternions like this:

$$\tilde{q}_0^{(+)} = \left(\tilde{q}_1^{(-)} {q_0}^{-1}\right)^{\frac{1}{2}} q_0$$
$$\tilde{q}_{N-1}^{(-)} = \left(\tilde{q}_{N-2}^{(+)} {q_{N-1}}^{-1}\right)^{\frac{1}{2}} q_{N-1}$$

When considering that the control polygon starts with the quaternions $\left(q_0, \tilde{q}_0^{(+)}, \tilde{q}_1^{(-)}, q_1, \tilde{q}_1^{(+)}, \dots\right)$ and ends with $\left(\dots, q_{N-2}, \tilde{q}_{N-2}^{(+)}, \tilde{q}_{N-1}^{(-)}, q_{N-1}\right)$, 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
```

### B.3.7.1 Examples

Let's first import NumPy, a few helpers from helper.py and the class *splines.quaternion.DeCasteljau* (page 290):

[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 ................................

### B.3.8 Barry–Goldman Algorithm With Slerp

We can try to use the *Barry–Goldman algorithm for non-uniform Euclidean Catmull–Rom splines* (page 181) using *Slerp* (page 240) instead of linear interpolations, just as we have done with *De Casteljau's algorithm* (page 246).

[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.
```

(continues on next page)

(continued from previous page)

```
    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)),
            slerp(q2, q3, (t - t2) / (t3 - t2)),
            (t - t1) / (t3 - t1)),
        (t - t1) / (t2 - t1))
```

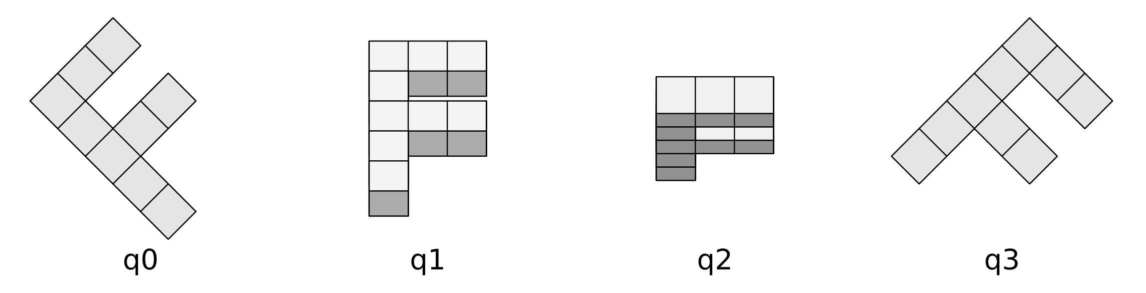
To illustrate this, let's import NumPy and a few helpers from `helper.py`:

```
[3]: import numpy as np
     from helper import angles2quat, plot_rotation, plot_rotations
     from helper import animate_rotations, display_animation
```

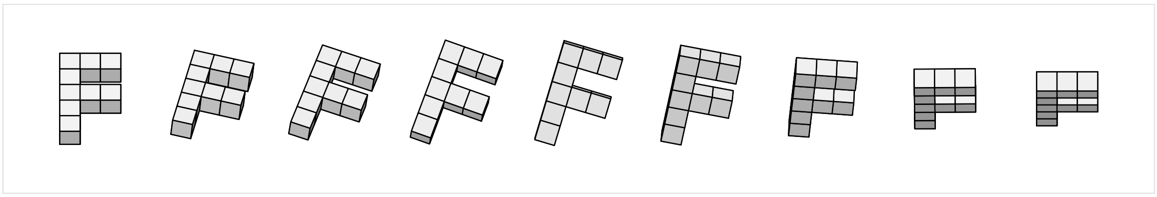
```
[4]: q0 = angles2quat(45, 0, 0)
     q1 = angles2quat(0, -40, 0)
     q2 = angles2quat(0, 70, 0)
     q3 = angles2quat(-45, 0, 0)
```

```
[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 class *splines.quaternion.BarryGoldman* (page 291):

```
[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 256) using the class *splines.quaternion.CatmullRom* (page 291):

```
[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]:
```
0.00266944746615122
```

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!
```

### B.3.8.1 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 corresponding tangent vector. When using the Barry–Goldman algorithm, the tangent vector has to be calculated separately, which makes 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 UnitSpeedAdapter
```

[23]:
```
bg3 = UnitSpeedAdapter(BarryGoldmanWithDerivative(rotations, alpha=0.5))
```

**Warning**

Evaluating this spline takes a long time!

[24]:
```
%%time
bg3_evaluated = evaluate(bg3)
```

```
CPU times: user 58.5 s, sys: 4.27 ms, total: 58.5 s
Wall time: 58.5 s
```

[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/squad.ipynb ........................................

### B.3.9 Spherical Quadrangle Interpolation (Squad)

The *Squad* method was introduced by Shoemake (1987). For a long time, his paper was not available online, but thanks to the nice folks at the Computer History Museum[70] (who only suggested a completely voluntary donation[71]), it is now available as PDF file[72] on their website[73].

The main argument for using *Squad* over *De Casteljaus's algorithm with Slerp* (page 246) is computational efficiency:

> Boehm (1982), in comparing different geometric controls for cubic polynomial segments, describes an evaluation method using "quadrangle points" which requires only 3 Lerps, half the number needed for the Bézier method adapted in Shoemake (1985).
>
> —Shoemake (1987)

Given the start and end points $p$ and $q$ of a curve segment and the so-called *quadrangle points* $a$ and $b$, Shoemake provides an equation for *Squad*:

> The interpretation of this algorithm is simple: $p$ and $q$ form one side of a quadrilateral, $a$ and $b$ the opposite side; the sides may be non-parallel and non-coplanar. The two inner Lerps find points on those sides, then the outer Lerp finds a point in between. Essentially, a simple parabola drawn on a square is subjected to an arbitrary bi-linear warp, which converts it to a cubic. Transliterated into Slerps, Boehm's algorithm gives a spherical curve,

$$\operatorname{Squad}(p, a, b, q; \alpha) = \operatorname{Slerp}(\operatorname{Slerp}(p, q; \alpha), \operatorname{slerp}(a, b; \alpha); 2(1-\alpha)\alpha)$$

[70] https://computerhistory.org/
[71] https://chm.secure.nonprofitsoapbox.com/donate
[72] https://archive.computerhistory.org/resources/access/text/2023/06/102724883-05-10-acc.pdf
[73] https://www.computerhistory.org/collections/catalog/102724883

—Shoemake (1987)

Shoemake also derives equations for the quadrangle points, which involves differentiation of Squad and assuming tangent vectors similar to *uniform Euclidean Catmull–Rom splines* (page 157).

> Given a series of quaternions $q_n$, use of Squad requires filling in values $a_n$ and $b_n$ on both sides of the interpolation points, so that each "cubic" segment is traced out by $\operatorname{Squad}(q_n, a_n, b_{n+1}, q_{n+1}; \alpha)$ [...] the values for $a_n$ and $b_n$ are given by
>
> $$a_n = b_n = q_n \exp\left(-\frac{\ln\left(q_n^{-1}q_{n+1}\right) + \ln\left(q_n^{-1}q_{n-1}\right)}{4}\right)$$
>
> —Shoemake (1987)

**Note**

Allegedly, the proof of continuity of tangents by Shoemake (1987) is flawed. Kim, Kim, et al. (1996) and Dam et al. (1998) provide new proofs, in case somebody wants to look that up.

The equation for the inner quadrangle points uses *relative rotations in the local frame of reference* (page 237) defined by $q_i$. Since we have mainly used rotations in the global frame of reference so far, we can also rewrite this equation to the equivalent form (changing the index $n$ to $i$ while we are at it)

$$a_i = b_i = \exp\left(-\frac{\ln\left(q_{i+1}q_i^{-1}\right) + \ln\left(q_{i-1}q_i^{-1}\right)}{4}\right) q_i.$$

We can try to get some intuition by looking at the Euclidean case. Euclidean quadrangle interpolation is shown in *a separate notebook* (page 150) and we know how to calculate outgoing and incoming quadrangle points for *uniform Euclidean Catmull–Rom splines* (page 174):

$$\tilde{x}_i^{(+)} = \tilde{x}_i^{(-)} = x_i - \frac{(x_{i+1} - x_i) + (x_{i-1} - x_i)}{4}.$$

With a bit of squinting, we can see that this is analogous to the quaternion equation shown above.

To show an example, we import *splines.quaternion.Squad* (page 291) and a few helper functions from `helper.py` ...

[1]:
```python
from splines.quaternion import Squad
from helper import angles2quat, animate_rotations, display_animation
```

... we define a sequence of rotations ...

```
[2]: rotations = [
         angles2quat(0, 0, 0),
         angles2quat(90, 0, -45),
         angles2quat(-45, 45, -90),
         angles2quat(135, -35, 90),
         angles2quat(90, 0, 0),
     ]
```

... and create a Squad object:

```
[3]: sq = Squad(rotations)
```

For comparison, we use *splines.quaternion.CatmullRom* (page 291) with the same sequence of rotations:

```
[4]: from splines.quaternion import CatmullRom
```

```
[5]: cr = CatmullRom(rotations, endconditions='closed')
```

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

```
[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({
         'Squad': evaluate(sq),
         'Catmull-Rom-like': evaluate(cr),
     })
     display_animation(ani, default_mode='loop')

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

As you can see, the two splines are nearly identical, but not quite:

```
[9]: max(max(map(abs, q.xyzw)) for q in (evaluate(sq) - evaluate(cr)))

[9]: 0.04640377605179979
```

#### B.3.9.1 Non-Uniform Parameterization

Shoemake (1987) uses uniform parameter intervals and doesn't talk about the non-uniform case at all. But we can try! In the *notebook about non-uniform Euclidean Catmull–Rom splines* (page 179) we have seen the equations for the Euclidean quadrangle points (with $\Delta_i = t_{i+1} - t_i$):

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

This can be "translated" to unit quaternions:

$$\bar{q}_i^{(+)} = \exp\left(-\frac{\Delta_i}{2(\Delta_{i-1}+\Delta_i)}\left(\ln\left(q_{i+1}q_i^{-1}\right) + \frac{\Delta_i}{\Delta_{i-1}}\ln\left(q_{i-1}q_i^{-1}\right)\right)\right) q_i$$

$$\bar{q}_i^{(-)} = \exp\left(-\frac{\Delta_{i-1}}{2(\Delta_{i-1}+\Delta_i)}\left(\frac{\Delta_{i-1}}{\Delta_i}\ln\left(q_{i+1}q_i^{-1}\right) + \ln\left(q_{i-1}q_i^{-1}\right)\right)\right) q_i$$

These two equations are implemented in *splines.quaternion.Squad* (page 291).

Being able to use non-uniform time values means that we can create a centripetal Squad spline:

```
[10]: sq2 = Squad(rotations, alpha=0.5)
```

```
[11]: cr2 = CatmullRom(rotations, alpha=0.5, endconditions='closed')
```

```
[12]: ani = animate_rotations({
          'Squad': evaluate(sq2),
          'Catmull-Rom-like': evaluate(cr2),
      })
      display_animation(ani, default_mode='loop')

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

The two movements are still very close.

```
[13]: max(max(map(abs, q.xyzw)) for q in (evaluate(sq2) - evaluate(cr2)))
[13]: 0.05019343803811403
```

Let's try some random non-uniform parameter values:

```
[14]: times = 0, 0.75, 1.6, 2, 3.5, 4
```

```
[15]: sq3 = Squad(rotations, times)
```

```
[16]: cr3 = CatmullRom(rotations, times, endconditions='closed')
```

```
[17]: ani = animate_rotations({
          'Squad': evaluate(sq3),
          'Catmull-Rom-like': evaluate(cr3),
      })
      display_animation(ani, default_mode='loop')

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

Now the two movements have some obvious differences.

```
[18]: max(max(map(abs, q.xyzw)) for q in (evaluate(sq3) - evaluate(cr3)))
[18]: 0.42509139916677563
```

With more uneven time values, the behavior of the Squad curve becomes more and more erratic. The reason for this might be the fact that the quadrangle control points are in general much further away from the curve than the Bézier control points. To check this, let's show the angle between adjacent control points in each segment, starting with the Bézier control points of our Catmull–Rom-like spline:

[19]:
```
%precision 1
[[np.degrees(q1.rotation_to(q2).angle) for q1, q2 in zip(s, s[1:])]
 for s in cr3.segments]
```

[19]:
```
[[17.0, 106.3, 19.9],
 [22.5, 107.3, 91.0],
 [42.8, 83.2, 44.9],
 [168.4, 209.5, 68.6],
 [22.9, 57.2, 11.3]]
```

An angle of 180 degree would mean a quarter of a great circle around the unit hypersphere.

Let's now compare that to the quadrangle control points:

[20]:
```
[[np.degrees(q1.rotation_to(q2).angle) for q1, q2 in zip(s, s[1:])]
 for s in sq3.segments]
```

[20]:
```
[[67.6, 206.7, 48.6],
 [62.4, 205.0, 108.4],
 [24.0, 209.4, 17.9],
 [251.1, 259.1, 103.9],
 [11.5, 130.6, 30.1]]
```

The angles are clearly much larger here.

With even more extreme time values, the control quaternions might even "wrap around" the unit hypersphere, leading to completely wrong movement between the given sequence of rotations. This will at some point also happen with the `CatmullRom` class, but with `Squad` it will happen much earlier.

........................................................................ doc/rotation/squad.ipynb ends here.

The following section was generated from doc/rotation/cumulative-form.ipynb ............................

### B.3.10 Cumulative Form

The basic idea, as proposed by Kim, Kim, et al. (1995) 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 134) – they suggest to use the relative difference vectors $\Delta p_i$ 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.

#### B.3.10.1 Piecewise Slerp

As an example, they define a piecewise linear curve

$$p(t) = p_0 + \sum_{i=1}^{n} \alpha_i(t) \Delta p_i,$$

where

$$\Delta p_i = p_i - p_{i-1}$$
$$\alpha_i(t) = \begin{cases} 0 & t < i-1 \\ t-i+1 & i-1 \leq t < i \\ 1 & t \geq i. \end{cases}$$

[1]:
```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 $\alpha_i(t)$:

$$\alpha_i(t) = \begin{cases} 0 & t < i \\ t-i & i \leq t < i+1 \\ 1 & t \geq i+1. \end{cases}$$

This assumes that $i$ starts with 0, but it actually starts with 1.

This "cumulative form" can be "translated" to a rotation spline by replacing addition with multiplication and the relative difference vectors by relative (i.e. local) rotations (represented by unit quaternions):

$$q(t) = q_0 \prod_{i=1}^{n} \exp(\omega_i \alpha_i(t)),$$

where

$$\omega_i = \log\left(q_{i-1}^{-1} q_i\right).$$

The paper uses above notation, but this could equivalently be written as

$$q(t) = q_0 \prod_{i=1}^{n} \left(q_{i-1}^{-1} q_i\right)^{\alpha_i(t)}.$$

[2]:
```
import numpy as np
```

Let's import a few helper functions from `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!
```

### B.3.10.2 Cumulative Bézier/Bernstein Curve

After the piecewise Slerp, Kim, Kim and Shin (1995) 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 148).

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 283) ...

```
[11]: from splines import Bernstein
```

... and create a simple helper function to sum them up:

```
[12]: from itertools import accumulate
```

```
[13]: def cumulative_bases(degree, t):
          return list(accumulate(Bernstein.basis(degree, t)[::-1]))[::-1]
```

Finally, they "translate" this into a rotation spline using quaternions, like before:

$$q(t) = q_0 \prod_{i=1}^{n} \exp\left(\omega_i \tilde{\beta}_{i,n}(t)\right),$$

where

$$\omega_i = \log(q_{i-1}^{-1} q_i).$$

Again, they use above notation in the paper, but this could equivalently be written as

$$q(t) = q_0 \prod_{i=1}^{n} \left(q_{i-1}^{-1} q_i\right)^{\tilde{\beta}_{i,n}(t)}.$$

```
[14]: def cumulative_bezier(qs, t):
          degree = len(qs) - 1
          bases = cumulative_bases(degree, t)
          assert np.isclose(bases[0], 1)
          return qs[0] * product([
              (qs[i - 1].inverse() * qs[i])**bases[i]
              for i in range(1, len(qs))
          ])
```

```
[15]: times = np.linspace(0, 1, 100)
```

```
[16]: rotations = [cumulative_bezier(qs, t) for t in times]
```

```
[17]: ani = animate_rotations(rotations)
```

```
[18]: display_animation(ani, default_mode='reflect')

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

#### B.3.10.3 Comparison with De Casteljau's Algorithm

> This Bézier quaternion curve has a different shape from the Bézier quaternion curve of Shoemake (1985).
>
> —Kim, Kim, et al. (1995), section 5.1

The method described by Shoemake (1985) is shown in *a separate notebook* (page 246). An implementation is available in the class *splines.quaternion.DeCasteljau* (page 290):

[19]:
```python
from splines.quaternion import DeCasteljau
```

[20]:
```python
times = np.linspace(0, 1, 100)
```

[21]:
```python
control_polygon = [
    angles2quat(90, 0, 0),
    angles2quat(0, -45, 90),
    angles2quat(0, 0, 0),
    angles2quat(180, 0, 180),
]
```

[22]:
```python
cumulative_rotations = [
    cumulative_bezier(control_polygon, t)
    for t in times
]
```

[23]:
```python
cumulative_rotations_reversed = [
    cumulative_bezier(control_polygon[::-1], t)
    for t in times
][::-1]
```

[24]:
```python
casteljau_rotations = DeCasteljau([control_polygon]).evaluate(times)
```

[25]:
```python
ani = animate_rotations({
    'De Casteljau': casteljau_rotations,
    'Cumulative': cumulative_rotations,
    'Cumulative reversed': cumulative_rotations_reversed,
})
```

[26]:
```python
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]:
```python
casteljau_rotations_reversed = DeCasteljau([control_polygon[::-1]]).
↪evaluate(times)[::-1]
```

[28]:
```python
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])
```

(continues on next page)

(continued from previous page)

```
    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 ....................

### B.3.11 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 88) 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.
>
> —Shoemake (1985), 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.
>
> —Shoemake (1985), 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 283) 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 291) 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 163). 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,
})
display_animation(ani, default_mode='loop')
```

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

Let's also try *arc-length parameterization* with the *UnitSpeedAdapter* (page 286):

[18]:
```
s3 = splines.UnitSpeedAdapter(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.UnitSpeedAdapter(
    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 ........

### B.3.12 Naive Interpolation of Euler Angles

This method for interpolating 3D rotations is not recommended at all!

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 283) to interpolate the Euler angles independently and *splines.quaternion.CatmullRom* (page 291) 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.

## B.4 Python Module

| | |
|---|---|
| *splines* (page 281) | Piecewise polynomial curves (in Euclidean space). |
| *splines.quaternion* (page 287) | Quaternions and unit-quaternion splines. |

### B.4.1 splines

Piecewise polynomial curves (in Euclidean space).

**Submodules**

| | |
|---|---|
| *quaternion* (page 287) | Quaternions and unit-quaternion splines. |

**Classes**

| | |
|---|---|
| *Bernstein* (page 282) | Piecewise Bézier curve using Bernstein basis. |
| *CatmullRom* (page 283) | Catmull--Rom spline. |
| *CubicHermite* (page 283) | Cubic Hermite curve. |
| *KochanekBartels* (page 284) | Kochanek--Bartels spline. |
| *Monomial* (page 282) | Piecewise polynomial curve using monomial basis. |
| *MonotoneCubic* (page 285) | Monotone cubic curve. |
| *Natural* (page 285) | Natural spline. |
| *NewGridAdapter* (page 286) | Re-parameterize a spline with new grid values. |
| *PiecewiseMonotoneCubic* (page 285) | Piecewise monotone cubic curve. |
| *UnitSpeedAdapter* (page 286) | Re-parameterize a spline to have a constant speed of 1. |

**class** `splines.`**`Monomial`**(*segments*, *grid=None*)

Bases: `object`

Piecewise polynomial curve using monomial basis.

See *Parametric Polynomial Curves* (page 88).

Coefficients can have an arbitrary number of dimensions. An arbitrary polynomial degree $d$ can be used by specifying $d + 1$ coefficients per segment. The $i$-th segment is evaluated using

$$\boldsymbol{p}_i(t) = \sum_{k=0}^{d} \boldsymbol{a}_{i,k} \left(\frac{t - t_i}{t_{i+1} - t_i}\right)^k \text{ for } t_i \leq t < t_{i+1}.$$

This is similar to `scipy.interpolate.PPoly`[74], 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 degrees.
- **`grid`** (*`optional`*) – Sequence of parameter values $t_i$ corresponding to segment boundaries. Must be strictly increasing. If not specified, a uniform grid is used (0, 1, 2, 3, ...).

**`evaluate`**(*t*, *n=0*)

Get value (or $n$-th derivative) at given parameter value(s) $t$.

[74] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PPoly.html#scipy.interpolate.PPoly`

**class** splines.**Bernstein**(*segments*, *grid=None*)

Bases: `object`

Piecewise Bézier curve using Bernstein basis.

See *Bézier Splines* (page 134).

**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

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i},$$

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

**evaluate**(*t*, *n=0*)

Get value at the given parameter value(s) *t*.

Only n=0 is currently supported.

**class** splines.**CubicHermite**(*vertices*, *tangents*, *grid=None*)

Bases: *Monomial* (page 282)

Cubic Hermite curve.

See *Hermite Splines* (page 105).

**Parameters**

- **vertices** – Sequence of vertices.
- **tangents** – Sequence of tangent vectors (two per segment: outgoing and incoming).
- **grid** (*optional*) – Sequence of parameter values. Must be strictly increasing. If not specified, a uniform grid is used (0, 1, 2, 3, ...).

```
matrix = array([[ 2, -2, 1, 1], [-3, 3, -2, -1], [ 0, 0, 1, 0],
[ 1, 0, 0, 0]])
```

**class** splines.**CatmullRom**(*vertices*, *grid=None*, ***, *alpha=None*, *endconditions='natural'*)

Bases: *CubicHermite* (page 283)

Catmull–Rom spline.

This class implements one specific member of the family of splines described by Catmull and Rom (1974), 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 283) spline.

See *Catmull--Rom Splines* (page 155).

**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*) – See *Parameterized Parameterization* (page 164).
- **endconditions** (*optional*) – Start/end conditions. Can be `'closed'`, `'natural'` or a pair of tangent vectors (a.k.a. "clamped"). If `'closed'`, the first vertex is re-used as last vertex and an additional *grid* value has to be specified.

**class** splines.**KochanekBartels**(*vertices*, *grid=None*, ***, *tcb=(0, 0, 0)*, *alpha=None*, *endconditions='natural'*)

Bases: *CubicHermite* (page 283)

Kochanek–Bartels spline.

See *Kochanek--Bartels Splines* (page 191).

**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*) – See *Parameterized Parameterization* (page 164).

- **`endconditions`** (*`optional`*) – Start/end conditions. Can be `'closed'`, `'natural'` or a pair of tangent vectors (a.k.a. "clamped"). If `'closed'`, the first vertex is re-used as last vertex and an additional *grid* value has to be specified.

**`class`** `splines.`**`Natural`**(*vertices*, *grid=None*, ***, *alpha=None*, *endconditions='natural'*)

Bases: *`CubicHermite`* (page 283)

Natural spline.

See *Natural Splines* (page 124).

**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`*) – See *Parameterized Parameterization* (page 164).
- **`endconditions`** (*`optional`*) – Start/end conditions. Can be `'closed'`, `'natural'` or a pair of tangent vectors (a.k.a. "clamped"). If `'closed'`, the first vertex is re-used as last vertex and an additional *grid* value has to be specified.

**`class`** `splines.`**`PiecewiseMonotoneCubic`**(*values*, *grid=None*, *slopes=None*, ***, *alpha=None*, *closed=False*)

Bases: *`CatmullRom`* (page 283)

Piecewise monotone cubic curve.

See *Piecewise Monotone Interpolation* (page 214).

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*, *grid=None*, *slopes=None*, ***, *alpha=None*, *cyclic=False*, ***kwargs*)

Bases: *PiecewiseMonotoneCubic* (page 285)

Monotone cubic curve.

This takes the same arguments as *PiecewiseMonotoneCubic* (page 285) (except `closed` is replaced by `cyclic`), but it raises an error if the given values are not montone.

See *Monotone Interpolation* (page 222).

**get_time**(*value*)

Get the time instance for the given *value*.

If the solution is not unique (i.e. if there is a plateau), `None` is returned.

**class** splines.**UnitSpeedAdapter**(*curve*)

Bases: `object`

Re-parameterize a spline to have a constant speed of 1.

For splines in Euclidean space this amounts to *Arc-Length Parameterization* (page 229).

However, this class is implemented in a way that also allows using rotation splines, which will be re-parameterized to have a *Constant Angular Speed* (page 248) of 1. For this to work, the second derivative of *curve* must yield an angular velocity vector. See *splines.quaternion.DeCasteljau* (page 290) for an example of a compatible rotation spline.

The parameter *s* represents the cumulative arc-length or the cumulative rotation angle, respectively.

**evaluate**(*s*)

Get value at the given parameter value(s) *s*.

**class** splines.**NewGridAdapter**(*curve*, *new_grid=1*, *cyclic=False*)

Bases: `object`

Re-parameterize a spline with new grid values.

This can be used for both Euclidean splines and rotation splines.

**Parameters**

- **curve** – A spline.
- **new_grid** (*optional*) – If a single number is given, the new parameter will range from 0 to that number. Otherwise, a sequence of numbers has to be given, one for each grid value. Instead of a value, `None` can be specified to choose a value automatically. The first and last value cannot be `None`.
- **cyclic** (*optional*) – If `True`, the slope of the re-parameterization function (but not necessarily the speed of the final spline!) will be the same at the beginning and end of the spline.

**evaluate**(*u*)

Get value at the given parameter value(s) *u*.

## B.4.2 splines.quaternion

Quaternions and unit-quaternion splines.

**Functions**

| | |
|---|---|
| *canonicalized* (page 289) | Iterator adapter to ensure minimal angles between *quaternions*. |
| *slerp* (page 289) | Spherical Linear intERPolation. |

**Classes**

| | |
|---|---|
| *BarryGoldman* (page 291) | Rotation spline using the Barry--Goldman algorithm with *slerp()* (page 289). |
| *CatmullRom* (page 291) | Catmull--Rom-like rotation spline. |
| *DeCasteljau* (page 290) | Rotation spline using De Casteljau's algorithm with *slerp()* (page 289). |
| *KochanekBartels* (page 290) | Kochanek--Bartels-like rotation spline. |
| *PiecewiseSlerp* (page 290) | Piecewise Slerp. |
| *Quaternion* (page 287) | A very simple quaternion class. |
| *Squad* (page 291) | Spherical Quadrangle Interpolation. |
| *UnitQuaternion* (page 288) | Unit quaternion. |

**class** splines.quaternion.**Quaternion**(*scalar, vector*)

Bases: `object`

A very simple quaternion class.

This is the base class for the more relevant class *UnitQuaternion* (page 288).

See the *notebook about quaternions* (page 233).

**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 287) part, negated *vector* (page 287) part.

**`normalized`**`()`

Return quaternion with same 4D direction but unit *`norm`* (page 288).

**`dot`**`(`*other*`)`

Dot product of two quaternions.

This is the four-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 287) last.

**`property wxyz`**

Components of the quaternion, *`scalar`* (page 287) first.

**`class`** `splines.quaternion.`**`UnitQuaternion`**

Bases: *`Quaternion`* (page 287)

Unit quaternion.

See the *section about unit quaternions* (page 234).

**`classmethod from_axis_angle`**`(`*axis*, *angle*`)`

Create a unit quaternion from a rotation *`axis`* (page 289) and *`angle`* (page 289).

**Parameters**

- **`axis`** – Three-component rotation axis. This will be normalized.
- **`angle`** – Rotation angle in radians.

**`classmethod from_unit_xyzw`**`(`*xyzw*`)`

Create a unit quaternion from another unit quaternion.

**Parameters**

**`xyzw`** – Components of a unit quaternion (`scalar` last). This will *not* be normalized, it must already have unit length.

**`inverse`**`()`

Multiplicative inverse.

For unit quaternions, this is the same as *`conjugate()`* (page 287).

**`classmethod exp_map`**`(`*value*`)`

Exponential map from $R^3$ to unit quaternions.

The *exponential map* operation transforms a three-dimensional vector that's a member of the tangent space at the identity quaternion into a unit quaternion.

This is the inverse operation to *`log_map()`* (page 289).

**Parameters**
`value` (*3-tuple*) – Element of the tangent space at the quaternion identity.

`log_map()`

Logarithmic map from unit quaternions to $R^3$.

The *logarithmic map* operation transforms a unit quaternion into a three-dimensional vector that's a member of the tangent space at the identity quaternion.

This is the inverse operation to *`exp_map()`* (page 288).

**Returns**
Corresponding three-element vector in the tangent space at the quaternion identity.

`property axis`

The (normalized) rotation axis.

`property angle`

The rotation angle in radians.

`rotation_to`(*other*)

Rotation required to rotate *self* into *other*.

See *Relative Rotation (Global Frame of Reference)* (page 237).

**Parameters**
`other` (`UnitQuaternion`) – Target rotation.

**Returns**
Relative rotation – as `UnitQuaternion`.

`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 240).

**Parameters**

- `one` (`UnitQuaternion`) – Start rotation.
- `two` (`UnitQuaternion`) – End rotation.
- `t` – Parameter value(s) between 0 and 1.

`splines.quaternion.`**`canonicalized`**(*quaternions*)

Iterator adapter to ensure minimal angles between *quaternions*.

See *Canonicalization* (page 239).

**`class`** `splines.quaternion.`**`PiecewiseSlerp`**(*quaternions, *, grid=None, closed=False*)

Bases: `object`

Piecewise Slerp.

See *Piecewise Slerp* (page 243).

**Parameters**

- **`quaternions`** – Sequence of rotations to be interpolated. The quaternions will be *`canonicalized()`* (page 289).
- **`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*)

Get value at the given parameter value(s) *t*.

Only `n=0` is currently supported.

**`class`** `splines.quaternion.`**`DeCasteljau`**(*segments, grid=None*)

Bases: `object`

Rotation spline using De Casteljau's algorithm with *`slerp()`* (page 289).

See *the corresponding notebook* (page 246) for details.

**Parameters**

- **`segments`** – Sequence of segments, each one consisting of multiple control quaternions. Different segments can have different numbers of control points.
- **`grid`** (*`optional`*) – Sequence of parameter values corresponding to segment boundaries. Must be strictly increasing. If not specified, a uniform grid is used (0, 1, 2, 3, ...).

**`evaluate`**(*t, n=0*)

Get value or angular velocity at given parameter value(s).

**Parameters**

- **`t`** – Parameter value(s).
- **`n`** (*`{0, 1}, optional`*) – Use `0` for calculating the value (a quaternion), `1` for the angular velocity (a three-element vector).

**class** `splines.quaternion.`**`KochanekBartels`**(*quaternions, grid=None, *, tcb=(0, 0, 0), alpha=None, endconditions='natural'*)

Bases: *`DeCasteljau`* (page 290)

Kochanek–Bartels-like rotation spline.

See *the corresponding notebook* (page 259) for details.

**Parameters**

- **`quaternions`** – Sequence of rotations to be interpolated. The quaternions will be *`canonicalized()`* (page 289).
- **`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`*) – See *Parameterized Parameterization* (page 164).
- **`endconditions`** (*`optional`*) – Start/end conditions. Can be `'closed'` or `'natural'`. If `'closed'`, the first rotation is re-used as last rotation and an additional *grid* value has to be specified.

**class** `splines.quaternion.`**`CatmullRom`**(*quaternions, grid=None, *, alpha=None, endconditions='natural'*)

Bases: *`KochanekBartels`* (page 290)

Catmull–Rom-like rotation spline.

This is just *`KochanekBartels`* (page 290) without TCB values.

See *Uniform Catmull--Rom-Like Quaternion Splines* (page 249) and *Non-Uniform Catmull--Rom-Like Rotation Splines* (page 256).

**class** `splines.quaternion.`**`BarryGoldman`**(*quaternions, grid=None, *, alpha=None*)

Bases: `object`

Rotation spline using the Barry–Goldman algorithm with *`slerp()`* (page 289).

Always closed (for now).

See *Barry--Goldman Algorithm With Slerp* (page 264).

**`evaluate`**(*t*)

Get value at the given parameter value(s) *t*.

**class** `splines.quaternion.`**`Squad`**(*quaternions*, *grid=None*, ***, *alpha=None*)

Bases: `object`

Spherical Quadrangle Interpolation.

Always closed (for now).

See *Spherical Quadrangle Interpolation* (*Squad*) (page 268).

**`evaluate`**(*t*)

Get value at the given parameter value(s) *t*.

# References

Ahrens, J. (2012). *Analytic Methods of Sound Field Synthesis*. Springer. DOI: 10.1007/978-3-642-25743-8 (cit. on p. 13).

Avendano, C. and J.-M. Jot (2004). "A Frequency-Domain Approach to Multichannel Upmix." In: *Journal of the Audio Engineering Society* 52.7, pp. 740–749 (cit. on p. 13).

Barry, P. J. and R. N. Goldman (1988). "A Recursive Evaluation Algorithm for a Class of Catmull–Rom Splines." In: *15th Annual Conference on Computer Graphics and Interactive Techniques*. ACM SIGGRAPH, pp. 199–204. DOI: 10.1145/54852.378511 (cit. on pp. 96, 181–186).

Bates, E. (2015). "Before and After Kontakte: Developments and Changes in Stockhausen's Approach to Spatial Music in the 1960s and 1970s." In: *Compositions for Audible Space*. Ed. by M. Brech and R. Paland. transcript Verlag, pp. 177–192. DOI: 10.1515/9783839430767-011 (cit. on p. 9).

Battier, M. (2015). "Recent Discoveries in the Spatial Thought of Early Musique concrète." In: *Compositions for Audible Space*. Ed. by M. Brech and R. Paland. transcript Verlag, pp. 123–136. DOI: 10.1515/9783839430767-007 (cit. on p. 8).

Bedell, E. H. and I. Kerney (1934). "Auditory perspective—System adaptation." In: *Electrical Engineering* 53.1, pp. 216–219. DOI: 10.1109/EE.1934.6540389 (cit. on p. 7).

Berkhout, A. J. (1988). "A Holographic Approach to Acoustic Control." In: *Journal of the Audio Engineering Society* 36.12, pp. 977–995 (cit. on p. 11).

Berkhout, A. J., D. De Vries, and P. Vogel (1993). "Acoustic Control by Wave Field Synthesis." In: *Journal of the Acoustical Society of America* 93.5, pp. 2764–2778 (cit. on p. 11).

Blauert, J., ed. (2005). *Communication Acoustics*. Springer. DOI: 10.1007/b139075 (cit. on p. 4).

Blauert, J. and R. Rabenstein (2012). "Providing Surround Sound with Loudspeakers: A Synopsis of Current Methods." In: *Archives of Acoustics* 37.1, pp. 5–18 (cit. on p. 13).

Boehm, W. (1982). "On cubics: A survey." In: *Computer Graphics and Image Processing* 19.3, pp. 201–226. DOI: 10.1016/0146-664X(82)90009-0 (cit. on pp. 151, 154, 268).

Boor, C. de (1972). "On calculating with B-splines." In: *Journal of Approximation Theory* 6.1, pp. 50–62. DOI: 10.1016/0021-9045(72)90080-9 (cit. on p. 184).

Boor, C. de (1978). *A Practical Guide to Splines*. Springer. ISBN: 978-0-387-95366-3 (cit. on pp. 104, 178, 217).

Bresson, J. and M. Schumacher (2011). "Representation and Interchange of Sound Spatialization Data for Compositional Applications." In: *International Computer Music Conference* (cit. on p. 28).

Bryant, D. (1981). "The 'cori spezzati' of St Mark's: myth and reality." In: *Early Music History* 1, pp. 165–186. DOI: 10.1017/S0261127900000280 (cit. on p. 4).

Catmull, E. and R. Rom (1974). "A Class of Local Interpolating Splines." In: *Computer Aided Geometric Design*. Ed. by R. E. Barnhill and R. F. Riesenfeld. Academic Press, pp. 317–326. DOI: 10.1016/B978-0-12-079050-0.50020-5 (cit. on pp. 155, 165–167, 169, 170, 173, 175, 181, 184, 284).

Chowning, J. M. (1971). "The Simulation of Moving Sound Sources." In: *Journal of the Audio Engineering Society* 19.1, pp. 2–6 (cit. on p. 9).

Chowning, J. M. (2011). "Turenas: the realization of a dream." In: *17es Journées d'Informatique Musicale, Saint-Etienne, France* (cit. on p. 9).

Cooper, G. (1970). "Tetrahedral Ambiophony – Part One." In: *Studio Sound* 12.6, pp. 233–234 (cit. on p. 10).

Dam, E. B., M. Koch, and M. Lillholm (1998). *Quaternions, Interpolation and Animation*. Technical Report DIKU-TR-98/5. Department of Computer Science, University of Copenhagen (cit. on p. 269).

Daniel, J. (2001). "Représentation de champs acoustiques, application à la transmission et à la reproduction de scènes sonores complexes dans un contexte multimédia." PhD thesis. Université Pierre et Marie Curie (Paris VI) (cit. on p. 13).

Daniel, J. (2003). "Spatial sound encoding including near field effect: Introducing distance coding filters and a viable, new Ambisonic format." In: *23rd International Conference of the Audio Engineering Society* (cit. on p. 13).

Doolittle, F. M. (1925). "Binaural Broadcasting." In: *Electrical World* 85.17, pp. 867–870 (cit. on p. 5).

Dougherty, R. L., A. S. Edelman, and J. M. Hyman (1989). "Nonnegativity-, monotonicity-, or convexity-preserving cubic and quintic Hermite interpolation." In: *Mathematics of Computation* 52.186, pp. 471–494. DOI: `10.1090/S0025-5718-1989-0962209-1` (cit. on p. 218).

Faller, C. (2006). "Multiple-Loudspeaker Playback of Stereo Signals." In: *Journal of the Audio Engineering Society* 54.11, pp. 1051–1064 (cit. on p. 13).

Farouki, R. T. (2012). "The Bernstein polynomial basis: A centennial retrospective." In: *Computer Aided Geometric Design* 29.6, pp. 379–419. DOI: `10.1016/j.cagd.2012.03.001` (cit. on p. 134).

Fellgett, P. (1975). "Ambisonics. Part one: General System Description." In: *Studio Sound* 17.8, pp. 20–22, 40 (cit. on p. 10).

Fletcher, H. (1933). "An Acoustic Illusion Telephonically Achieved." In: *Bell Laboratories Record* 11.10, pp. 286–289 (cit. on p. 5).

Fletcher, H. (1934). "Auditory Perspective—Basic Requirements." In: *Electrical Engineering* 53.1, pp. 9–11. DOI: `10.1109/EE.1934.6540356` (cit. on p. 11).

Fritsch, F. N. (1982). *Piecewise Cubic Hermite Interpolation Package (Final Specifications)*. Technical Report UCID-30194. USA: Lawrence Livermore National Laboratory. DOI: `10.2172/6838406` (cit. on p. 218).

Fritsch, F. N. and J. Butland (1984). "A Method for Constructing Local Monotone Piecewise Cubic Interpolants." In: *SIAM Journal on Scientific and Statistical Computing* 5.2, pp. 300–304. DOI: `10.1137/0905021` (cit. on pp. 218, 219).

Fritsch, F. N. and R. E. Carlson (1980). "Monotone Piecewise Cubic Interpolation." In: *SIAM Journal on Numerical Analysis* 17.2, pp. 238–246. DOI: `10.1137/0717021` (cit. on pp. 217, 218, 224).

Garity, W. E. and J. N. A. Hawkins (1941). "Fantasound." In: *Journal of the Society of Motion Picture Engineers* 37.8, pp. 127–146. DOI: `10.5594/J12890` (cit. on pp. 1, 7).

Geier, M., J. Ahrens, A. Möhl, S. Spors, J. Loh, and K. Bredies (2007). "The SoundScape Renderer: A Versatile Software Framework for Spatial Audio Reproduction." In: *WFS Symposium Ilmenau*. Deutsche Gesellschaft für Akustik (DEGA) (cit. on p. 42).

Geier, M., J. Ahrens, and S. Spors (2008a). "ASDF: Ein XML Format zur Beschreibung von virtuellen 3D Audioszenen." In: *34. Jahrestagung der Deutschen Gesellschaft für Akustik* (cit. on p. 33).

Geier, M., J. Ahrens, and S. Spors (2008b). "The SoundScape Renderer: A Unified Spatial Audio Reproduction Framework for Arbitrary Rendering Methods." In: *124th Convention of the Audio Engineering Society* (cit. on p. 42).

Geier, M., J. Ahrens, and S. Spors (2009). "Binaural Monitoring of Massive Multichannel Sound Reproduction Systems using Model-Based Rendering." In: *NAG/DAGA International Conference on Acoustics* (cit. on p. 13).

Geier, M., J. Ahrens, and S. Spors (2010). "Object-based Audio Reproduction and the Audio Scene Description Format." In: *Organised Sound* 15.3, pp. 219–227 (cit. on pp. 12, 34).

Geier, M., T. Hohn, and S. Spors (2012). "An Open-Source C++ Framework for Multithreaded Realtime Multichannel Audio Applications." In: *Linux Audio Conference* (cit. on p. 42).

Geier, M. and S. Spors (2008). "ASDF: Audio Scene Description Format." In: *International Computer Music Conference* (cit. on p. 33).

Geier, M. and S. Spors (2012). "Spatial Audio Reproduction with the SoundScape Renderer." In: *27th Tonmeistertagung – VDT International Convention* (cit. on p. 42).

Geier, M., S. Spors, and S. Weinzierl (2010). "The Future of Audio Reproduction: Technology – Formats – Applications." In: *Adaptive Multimedia Retrieval. Identifying, Summarizing, and Recommending Image and Music*. Ed. by M. Detyniecki, U. Leiner, and A. Nürnberger. Vol. 5811. LNCS. Springer, pp. 1–17. DOI: `10.1007/978-3-642-14758-6_1` (cit. on pp. 12, 33, 34).

Gembicki, B. (2020). "The Memory of Meaning: Polychorality in Venice and the Cori Spezzati Meme." In: *Sounding the Past*. Brepols Publishers, pp. 257–271. DOI: `10.1484/M.EMEB.5.122016` (cit. on p. 4).

Gerzon, M. A. (1970). "The Principles of Quadraphonic Recording – Part Two." In: *Studio Sound* 12.9, pp. 380–384 (cit. on p. 10).

Gerzon, M. A. (1973). "Periphony: With-height Sound Reproduction." In: *Journal of the Audio Engineering Society* 21.1, pp. 2–10 (cit. on p. 10).

Gerzon, M. A. (1975). "Ambisonics. Part two: Studio techniques." In: *Studio Sound* 17.8, pp. 24–26, 28, 30 (cit. on p. 10).

Goose, S., S. Kodlahalli, W. Pechter, and R. Hjelsvold (2002). "Streaming Speech$^3$: A Framework for Generating and Streaming 3D Text-To-Speech and Audio Presentations to Wireless PDAs as Specified Using Extensions to SMIL." In: *International Conference on World Wide Web*. Association for Computing Machinery, pp. 37–44. DOI: `10.1145/511446.511452` (cit. on p. 29).

Gordon, W. J. and R. F. Riesenfeld (1974). "B-spline Curves and Surfaces." In: *Computer Aided Geometric Design*. Academic Press, pp. 95–126. DOI: `10.1016/B978-0-12-079050-0.50011-4` (cit. on pp. 103, 168).

Grimm, G., J. Luberadzka, and V. Hohmann (2019). "A Toolbox for Rendering Virtual Acoustic Environments in the Context of Audiology." In: *Acta Acustica united with Acustica* 105.3, pp. 566–578. DOI: `10.3813/AAA.919337` (cit. on p. 28).

Hamasaki, K., W. Hatano, K. Hiyama, S. Komiyama, and H. Okubo (2004). "5.1 and 22.2 Multichannel Sound Productions Using an Integrated Surround Sound Panning System." In: *117th Convention of the Audio Engineering Society* (cit. on p. 10).

Hamasaki, K., K. Hiyama, and R. Okumura (2005). "The 22.2 Multichannel Sound System and Its Application." In: *118th Convention of the Audio Engineering Society* (cit. on p. 10).

Hoffmann, H., R. Dachselt, and K. Meissner (2003). "An Independent Declarative 3D Audio Format on the Basis of XML." In: *International Conference on Auditory Display* (cit. on p. 22).

Hospitalier, É. (1881). "Les auditions téléphoniques théatrales – Système Clément Ader." In: *L'Électricien* 1.12, pp. 572–579 (cit. on p. 5).

Kalff, L. C., W. Tak, and S. L. de Bruin (1958). "The 'Electronic Poem' Performed in the Philips Pavilion at the 1958 Brussels World Fair." In: *Philips Technical Review* 20.2, pp. 37–84 (cit. on p. 8).

Kim, M., S. Wood, and L.-T. Cheok (2000). "Extensible MPEG-4 textual format (XMT)." In: *ACM Workshops on Multimedia*, pp. 71–74. DOI: `10.1145/357744.357763` (cit. on p. 22).

Kim, M.-J., M.-S. Kim, and S. Y. Shin (1995). “A General Construction Scheme for Unit Quaternion Curves with Simple High Order Derivatives.” In: *SIGGRAPH: Computer graphics and interactive techniques*, pp. 369–376. DOI: 10.1145/218380.218486 (cit. on pp. 272, 276).

Kim, M.-J., M.-S. Kim, and S. Y. Shin (1996). “A Compact Differential Formula for the First Derivative of a Unit Quaternion Curve.” In: *The Journal of Visualization and Computer Animation* 7.1, pp. 43–57. DOI: 10.1002/(SICI)1099-1778(199601)7:1<43::AID-VIS136>3.0.CO;2-T (cit. on p. 269).

Kochanek, D. H. U. and R. H. Bartels (1984). “Interpolating Splines with Local Tension, Continuity, and Bias Control.” In: *11th Annual Conference on Computer Graphics and Interactive Techniques*. ACM SIGGRAPH, pp. 33–41. DOI: 10.1145/800031.808575 (cit. on pp. 191, 196, 197, 201, 202).

Konishi, M. (2003). “Coding of Auditory Space.” In: *Annual Review of Neuroscience* 26.1, pp. 31–55. DOI: 10.1146/annurev.neuro.26.041002.131123 (cit. on p. 3).

Lee, E. T. Y. (1989). “Choosing nodes in parametric curve interpolation.” In: *Computer-Aided Design* 21.6, pp. 363–370. DOI: 10.1016/0010-4485(89)90003-1 (cit. on pp. 160, 163).

Lossius, T., P. Baltazar, and T. de la Hogue (2009). “DBAP – Distance-Based Amplitude Panning.” In: *International Computer Music Conference* (cit. on p. 10).

MacLeod, K. M., D. Soares, and C. E. Carr (2006). “Interaural Timing Difference Circuits in the Auditory Brainstem of the Emu (Dromaius novaehollandiae).” In: *Journal of Comparative Neurology* 495.2, pp. 185–201. DOI: 10.1002/cne.20862 (cit. on p. 3).

McDonald, J. (2010). “Teaching Quaternions is not Complex.” In: *Computer Graphics Forum* 29.8, pp. 2447–2455. DOI: 10.1111/j.1467-8659.2010.01756.x (cit. on p. 234).

Millington, I. (2009). *Matrices and Conversions for Uniform Parametric Curves*. URL: https://web.archive.org/web/20160305083440/http://therndguy.com (cit. on pp. 87, 197).

Miyama, C., J. C. Schacher, and N. Peters (2013). “SpatDIF Library – Implementing the Spatial Sound Descriptor Interchange Format.” In: *Journal of the Japanese Society for Sonic Arts* 5.3, pp. 1–5 (cit. on p. 28).

Moler, C. B. (2004). *Numerical Computing with MATLAB*. Society for Industrial and Applied Mathematics. ISBN: 978-0-89871-660-3 (cit. on pp. 219, 225).

Neukom, M. and J. C. Schacher (2008). “Ambisonics equivalent panning.” In: *International Computer Music Conference* (cit. on p. 13).

Nicol, R. (2018). “Sound Field.” In: *Immersive Sound: The Art and Science of Binaural and Multi-Channel Audio*. Ed. by A. Roginska and P. Geluso. Taylor & Francis, pp. 276–310 (cit. on p. 13).

Overhauser, A. W. (1968). *Analytic Definition of Curves and Surfaces by Parabolic Blending*. Technical Report SL 68-40. Dearborn, Michigan: Scientific Laboratory, Ford Motor Company (cit. on pp. 155, 178, 186).

Paland, R. (2015). “‘... every movement is possible’: Spatial Composition in Iannis Xenakis’s Hibiki-Hana-Ma.” In: *Compositions for Audible Space*. Ed. by M. Brech and R. Paland. transcript Verlag, pp. 305–321. DOI: 10.1515/9783839430767-019 (cit. on p. 9).

Paul, S. (2009). “Binaural Recording Technology: A Historical Review and Possible Future Developments.” In: *Acta Acustica united with Acustica* 95.5, pp. 767–788. DOI: 10.3813/AAA.918208 (cit. on p. 6).

Pereira, F. C. and T. Ebrahimi (2002). *The MPEG-4 Book*. Prentice Hall PTR. 550 pp. ISBN: 978-0-13-061621-0 (cit. on p. 12).

Peters, N. (2008). “Proposing SpatDIF – The Spatial Sound Description Interchange Format.” In: *International Computer Music Conference* (cit. on p. 26).

Peters, N., S. Ferguson, and S. McAdams (2007). "Towards a Spatial Sound Description Interchange Format (SpatDIF)." In: *Canadian Acoustics* 35.3, pp. 64–65 (cit. on p. 26).

Peters, N., T. Lossius, and J. C. Schacher (2013). "The Spatial Sound Description Interchange Format: Principles, Specification, and Examples." In: *Computer Music Journal* 37.1, pp. 11–22. DOI: 10.1162/COMJ_a_00167 (cit. on p. 26).

Pihkala, K. and T. Lokki (2003). "Extending SMIL with 3D Audio." In: *International Conference on Auditory Display* (cit. on p. 29).

Popper, A. N. and R. R. Fay (1993). "Sound Detection and Processing by Fish: Critical Review and Major Research Questions." In: *Brain, Behavior and Evolution* 41.1, pp. 14–38. DOI: 10.1159/000113821 (cit. on p. 3).

Potard, G. (2006). "3D-Audio Object Oriented Coding." Dissertation. University of Wollongong (cit. on p. 23).

Potard, G. and I. Burnett (2002). "Using XML Schemas to Create and Encode Interactive 3-D Audio Scenes for Multimedia and Virtual Reality Applications." In: *Distributed Communities on the Web Workshop* (cit. on p. 23).

Potard, G. and I. Burnett (2004). "An XML-based 3D audio scene metadata scheme." In: *25th International Conference of the Audio Engineering Society* (cit. on pp. 23, 25).

Potard, G. and S. Ingham (2003). "Encoding 3D sound scenes and music in XML." In: *International Computer Music Conference* (cit. on pp. 23, 25).

Pulkki, V. (1997). "Virtual Sound Source Positioning using Vector Base Amplitude Panning." In: *Journal of the Audio Engineering Society* 45.6, pp. 456–466 (cit. on p. 10).

Pulkki, V. and M. Karjalainen (2015). *Communication Acoustics: An Introduction to Speech, Audio and Psychoacoustics*. John Wiley & Sons. ISBN: 978-1-118-86654-2 (cit. on p. 4).

Riedmiller, J., S. Mehta, N. Tsingos, and P. Boon (2015). "Immersive and Personalized Audio: A Practical System for Enabling Interchange, Distribution, and Delivery of Next-Generation Audio Experiences." In: *SMPTE Motion Imaging Journal* 124.5, pp. 1–23. DOI: 10.5594/j18578 (cit. on p. 17).

Robinson, C. Q., S. Mehta, and N. Tsingos (2012). "Scalable Format and Tools to Extend the Possibilities of Cinema Audio." In: *SMPTE Motion Imaging Journal* 121.8, pp. 63–69. DOI: 10.5594/j18248XY (cit. on p. 15).

Robinson, C. Q. and N. Tsingos (2015). "Cinematic Sound Scene Description and Rendering Control." In: *SMPTE Motion Imaging Journal* 124.8, pp. 47–53. DOI: 10.5594/j18640 (cit. on pp. 13, 36).

Rumsey, F. (2002). "Spatial Quality Evaluation for Reproduced Sound: Terminology, Meaning, and a Scene-based Paradigm." In: *Journal of the Audio Engineering Society* 50.9, pp. 651–666 (cit. on p. 13).

Schacher, J. C., N. Peters, T. Lossius, and C. Miyama (2016). "Authoring Spatial Music with SpatDIF Version 0.4." In: *Sound & Music Computing Conference* (cit. on p. 28).

Scheirer, E. D., R. Väänänen, and J. Huopaniemi (1999). "AudioBIFS: Describing Audio Scenes with the MPEG-4 Multimedia Standard." In: *IEEE Transactions on Multimedia* 1.3, pp. 237–250 (cit. on p. 22).

Schmidt, A. K. D. and H. Römer (2011). "Solutions to the Cocktail Party Problem in Insects: Selective Filters, Spatial Release from Masking and Gain Control in Tropical Crickets." In: *PLoS ONE* 6.12. DOI: 10.1371/journal.pone.0028593 (cit. on p. 3).

Schmidt, J. and E. F. Schröder (2004). "New and Advanced Features for Audio Presentation in the MPEG-4 Standard." In: *116th Convention of the Audio Engineering Society* (cit. on p. 22).

Schoenberg, I. J. (1946). "Contributions to the problem of approximation of equidistant data by analytic functions. Part A.–On the problem of smoothing or graduation. A first class of analytic approximation formulae." In: *Quarterly of Applied Mathematics* 4.1, pp. 45–99. DOI: 10.1090/qam/15914 (cit. on pp. 101, 102).

Schütz, R. (2010). "Numerical Modelling of Shotcrete for Tunnelling." Ph.D. Thesis. Imperial College London (cit. on p. 48).

Shoemake, K. (1985). "Animating Rotation with Quaternion Curves." In: *SIGGRAPH Computer Graphics* 19.3, pp. 245–254. DOI: 10.1145/325165.325242 (cit. on pp. 240, 246, 248, 254–256, 268, 276, 277).

Shoemake, K. (1987). "Quaternion Calculus and Fast Animation." In: *Computer Animation: 3D Motion Specification and Control*. ACM SIGGRAPH course notes 10, pp. 101–121 (cit. on pp. 268–270).

Snow, W. B. (1953). "Basic Principles of Stereophonic Sound." In: *Journal of the Society of Motion Picture and Television Engineers* 61.5, pp. 567–589. DOI: 10.5594/J00963 (cit. on pp. 8, 11).

Spors, S., R. Rabenstein, and J. Ahrens (2008). "The Theory of Wave Field Synthesis revisited." In: *124th Convention of the Audio Engineering Society* (cit. on p. 11).

Spors, S., H. Wierstorf, A. Raake, F. Melchior, M. Frank, and F. Zotter (2013). "Spatial Sound With Loudspeakers and Its Perception: A Review of the Current State." In: *Proceedings of the IEEE* 101.9, pp. 1920–1938. DOI: 10.1109/JPROC.2013.2264784 (cit. on p. 13).

Steinberg, J. C. and W. B. Snow (1934). "Auditory perspective—Physical factors." In: *Electrical Engineering* 53.1, pp. 12–17. DOI: 10.1109/EE.1934.6540357 (cit. on p. 7).

Theile, G. (1980). "Über die Lokalisation im überlagerten Schallfeld." Dissertation. Technische Universität Berlin (cit. on p. 11).

Theile, G. and G. Plenge (1977). "Localization of Lateral Phantom Sources." In: *Journal of the Audio Engineering Society* 25.4, pp. 196–200 (cit. on p. 11).

Theile, G., H. Wittek, and M. Reisinger (2003). "Potential Wavefield Synthesis Applications in the Multichannel Stereophonic World." In: *24th International Conference of the Audio Engineering Society*, pp. 43–57 (cit. on p. 12).

Tsingos, N. (2018). "Object-Based Audio." In: *Immersive Sound: The Art and Science of Binaural and Multi-Channel Audio*. Ed. by A. Roginska and P. Geluso. Taylor & Francis, pp. 244–275 (cit. on p. 12).

Väänänen, R. (2003). "Parametrization, Auralization, and Authoring of Room Acoustics for Virtual Reality Applications." Dissertation. Helsinki University of Technology (cit. on p. 36).

Väänänen, R. and J. Huopaniemi (2004). "Advanced AudioBIFS: Virtual Acoustics Modeling in MPEG-4 Scene Description." In: *IEEE Transactions on Multimedia* 6.5, pp. 661–675 (cit. on p. 22).

Vilkamo, J., A. Kuntz, and S. Füg (2014). "Reduction of Spectral Artifacts in Multichannel Downmixing with Adaptive Phase Alignment." In: *Journal of the Audio Engineering Society* 62.7, pp. 516–526 (cit. on p. 13).

Wenzel, E. M., S. S. Fisher, P. K. Stone, and S. H. Foster (1990). "A System for Three-dimensional Acoustic "Visualization" in a Virtual Environment Workstation." In: *1st IEEE Conference on Visualization*, pp. 329–337 (cit. on p. 13).

Woodward, J. G. (1977). "Quadraphony–A Review." In: *Journal of the Audio Engineering Society* 25.10, pp. 843–854 (cit. on p. 10).

Yager, D. D. (1999). "Structure, Development, and Evolution of Insect Auditory Systems." In: *Microscopy Research and Technique* 47.6, pp. 380–400. DOI: 10.1002/(SICI)1097-0029(19991215)47:6<380::AID-JEMT3>3.0.CO;2-P (cit. on p. 3).

Yager, D. D. and R. R. Hoy (1986). "The Cyclopean Ear: A New Sense for the Praying Mantis." In: *Science* 231.4739, pp. 727–729. DOI: 10.1126/science.3945806 (cit. on p. 3).

Yuksel, C., S. Schaefer, and J. Keyser (2011). “Parameterization and applications of Catmull–Rom curves.” In: *Computer-Aided Design* 43.7, pp. 747–755. DOI: 10.1016/j.cad.2010.08.008 (cit. on pp. 161, 164, 165, 185).

Zielinski, S. K., F. Rumsey, and S. Bech (2003). “Effects of Down-Mix Algorithms on Quality of Surround Sound.” In: *Journal of the Audio Engineering Society* 51.9, pp. 780–798 (cit. on p. 13).

Zotter, F. and M. Frank (2012). “All-Round Ambisonic Panning and Decoding.” In: *Journal of the Audio Engineering Society* 60.10, pp. 807–820 (cit. on p. 10).

Zotter, F. and M. Frank (2019). *Ambisonics: A Practical 3D Audio Theory for Recording, Studio Production, Sound Reinforcement, and Virtual Reality*. Vol. 19. Cham: Springer International Publishing. DOI: 10.1007/978-3-030-17207-7 (cit. on p. 13).