ASDF

Release 25ffa03

Matthias Geier

2023-09-27

Contents

The ASDF is an XML-based file format for authoring 3D audio scenes.

Online documentation/specification
https://AudioSceneDescriptionFormat.readthedocs.io/

Source code repository (and issue tracker)
https://github.com/AudioSceneDescriptionFormat/asdf

Reference implementation (implemented in Rust, with a C API)
https://github.com/AudioSceneDescriptionFormat/asdf-rust

License
Dedicated to the public domain using CC0 – see the file `LICENSE` for details.

1 Introduction

Let's start simple, with the file `minimal.asd`[1]:

```
<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`[2], 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 3).

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`[3]:

```
<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`[4], each channel coming from its specified position. For further details, see *<clip> and <channel>* (page 8).

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`[5]:

```
<?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[6] has been added, which is optional in XML 1.0 (but not in XML 1.1).

[1] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/minimal.asd
[2] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/audio/ukewave.ogg
[3] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/minimal-multichannel.asd
[4] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/audio/marimba.ogg
[5] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/minimal-expanded.asd
[6] https://www.w3.org/TR/xml/#sec-prolog-dtd

- The *<head> and <body>* (page 5) elements are optional. The *<asdf>* (page 4) element (including version number) is always required.
- In the <head> section there is a separate *<source>* (page 5) element.
- The <body> element implicitly behaves like a <seq> element, see *<seq> and <par>* (page 7).
- 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 5) that was defined in <head>.
- The z-component in pos is optional, see *<transform>* (page 10).

This still uses the shorthand of specifying the position directly in the <channel> element. As shown in minimal-expanded-with-explicit-transform.asd[7], 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>
```

- Because the <clip> and the <transform> happen at the same time, they are wrapped in a <par> element, see *<seq> and <par>* (page 7). 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 10) element could be even further expanded to contain the pos information in a single <o> sub-element.

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[8]. 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

[7] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/minimal-expanded-with-explicit-transform.asd

[8] https://en.wikipedia.org/wiki/Axes_conventions

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[9] 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[10]. 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 22).

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 16).

3 Elements

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

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 5) 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 7), see *<head> and <body>* (page 5). For example, the clips in `implicit-seq.asd`[11] are played in sequence:

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

[9] https://en.wikipedia.org/wiki/Euler_angles#Tait\T1\textendash{}Bryan_angles
[10] https://en.wikipedia.org/wiki/Right-hand_rule#Rotations
[11] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/implicit-seq.asd

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 4).

The <head> element can contain *<source>* (page 5) sub-elements and an optional *<reference>* (page 6). 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 10) elements can be used in the <body> to offset those values dynamically.

The <body> element can contain *<seq> and <par>* (page 7) elements, as well as *<clip>* (page 8) and *<transform>* (page 10) elements. If the <body> element contains multiple sub-elements, it acts like an implicit *<seq>* (page 7) element.

3.3 <source>

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

File Inputs

<clip> and <channel> (page 8) 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.

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[12] 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[13] 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>
```

[12] http://spatialaudio.net/ssr/
[13] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/live-sources.asd

Live sources and sources driven by audio files can be mixed in one scene and *<transform>* (page 10) elements can apply to either. See e.g. `live-sources-and-file-sources.asd`[14]:

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

Transform Attributes

Any `<source>` element with an `id` attribute can be the target of a *<transform>* (page 10) (using the `apply-to` attribute). Like *<clip> and <channel>* (page 8), `<source>` can also use transform attributes like `pos`, `rot` etc. as a shortcut, see `source-transform.asd`[15]:

```
<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>
```

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`[16]:

[14] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/live-sources-and-file-sources.asd
[15] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/source-transform.asd
[16] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/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 10). If no <reference> element is given, the reference point can still be transformed using apply-to="reference", as in implicit-reference.asd[17]:

```
<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>
```

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 16) 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[18]:

```
<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 4) element implicitly behaves like a <seq> element, i.e. all contained elements are played in sequence, like in the example file implicit-seq.asd[19]:

[17] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/implicit-reference.asd
[18] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/seq-par.asd
[19] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/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>.

repeat

<seq> and <par> elements can be repeated, see *Repetition* (page 17).

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[20]:

```
<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[21]:

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

Volume control is part of the *<transform>* (page 10) 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 7).

As selecting-channels.asd[22] 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" />
```

(continues on next page)

[20] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/minimal-multichannel.asd
[21] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/minimal.asd
[22] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/selecting-channels.asd

(continued from previous page)

```
      <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.

repeat

`<clip>` elements can be repeated, see *Repetition* (page 17).

id

Both `<clip>` and `<channel>` elements can be the target of a *<transform>* (page 10), 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).

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 5) elements, as demonstrated in `source-transform.asd`[23]:

```
<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.

[23] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/source-transform.asd

3.7 <transform>

A constant transform can be simply added to a <clip> element, like the pos attribute in minimal.asd[24]:

```
<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 8), as well as *<source>* (page 5) and *<reference>* (page 6).

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[25]:

```
<?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>
```

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 5), *<clip> and <channel>* (page 8) elements, as well as other <transform> elements. The special ID "reference" can be used to target the *<reference>* (page 6).

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.

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 6) – 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 3).

[24] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/minimal.asd
[25] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/minimal-expanded-with-explicit-transform.asd

rot

Unlike `pos`, this is aptly named after *rotation*. The final *orientation* of a sound source – or the *<reference>* (page 6) – 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 3).

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.

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.

<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`[26]:

```
<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`[27]:

```
<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" />
```

(continues on next page)

[26] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/two-pos.asd
[27] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/two-rot.asd

(continued from previous page)

```
      <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`[28]:

```
<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`[29]:

```
<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.

[28] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/minimal-spline.asd
[29] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/transform-vol.asd

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`[30]:

```
<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>
  </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`[31]:

```
<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`[32]:

```
<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.

[30] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/spline-time.asd
[31] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/spline-time-hh-mm-ss.asd
[32] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/spline-time-percent.asd

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`[33]:

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

tension/continuity/bias

The ASDF uses Kochanek–Bartels Splines[34], 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`[35]:

```
<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 19).

[33] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/spline-speed.asd
[34] https://splines.readthedocs.io/en/latest/euclidean/kochanek-bartels.html
[35] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/spline-tcb.asd

TCB attributes can also be used for `rot` trajectories, leading to Kochanek–Bartels-like Rotation Splines[36].

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`[37] 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>
```

repeat

`<transform>` elements can be repeated, see *Repetition* (page 17).

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 7).

[36] https://splines.readthedocs.io/en/latest/rotation/kochanek-bartels.html
[37] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/mixed-transform-attributes.asd

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[38]:

```
<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" />
      <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.

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.

3.8 <wait>

This can be used to wait for some time, see e.g. wait.asd[39]:

```
<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>
```

[38] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/nested-transforms.asd
[39] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/wait.asd

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 13) attribute of transform nodes is supported.

4 Repetition

<clip> (page 8), *<transform>* (page 10), *<seq> and <par>* (page 7) 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`[40]:

```
<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`[41]:

```
<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>
```

[40] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/repeat.asd
[41] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/repeat-nearly-indefinitely.asd

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[42] and it provides language bindings for C[43] and Python[44].

We refer to a general definition of splines and their properties[45] and to detailed background information about all the different types of Euclidean splines[46] and rotation splines[47] mentioned here, including their mathematical derivation and their individual properties.

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[48], 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[49]. 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[50] 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[51].

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[52]. 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 10) for details.

It might have been tempting to use Bézier Splines[53] 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[54].

[42] https://www.rust-lang.org/
[43] https://www.open-std.org/jtc1/sc22/wg14/
[44] https://www.python.org/
[45] https://splines.readthedocs.io/en/latest/euclidean/splines.html
[46] https://splines.readthedocs.io/en/latest/euclidean/index.html
[47] https://splines.readthedocs.io/en/latest/rotation/index.html
[48] https://splines.readthedocs.io/en/latest/euclidean/kochanek-bartels.html
[49] https://splines.readthedocs.io/en/latest/euclidean/catmull-rom.html
[50] https://splines.readthedocs.io/en/latest/euclidean/catmull-rom-properties.html#Centripetal-Parameterization
[51] https://splines.readthedocs.io/en/latest/euclidean/re-parameterization.html#Arc-Length-Parameterization
[52] https://splines.readthedocs.io/en/latest/euclidean/re-parameterization.html#Spline-Based-Re-Parameterization
[53] https://splines.readthedocs.io/en/latest/euclidean/bezier.html
[54] https://splines.readthedocs.io/en/latest/euclidean/hermite.html

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[55] are used, which are a superset of Catmull–Rom-Like Rotation Splines[56]. 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.

5.3 Volume Splines

The volume of the *<reference>* (page 6), of *<source>* (page 5) 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 10) 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[57].

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 18).

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`[58]:

```
<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" />
      <o pos="0 -2" />
      <o pos="2 0" />
      <o pos="closed" />
```

(continues on next page)

[55] https://splines.readthedocs.io/en/latest/rotation/kochanek-bartels.html
[56] https://splines.readthedocs.io/en/latest/rotation/catmull-rom-non-uniform.html
[57] https://splines.readthedocs.io/en/latest/euclidean/piecewise-monotone.html
[58] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/square.asd

(continued from previous page)

```
    </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>
```

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`[59] 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.

[59] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/circle.asd

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`[60]:

```
<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`.

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`[61]:

```
<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>
```

[60] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/helix.asd
[61] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/sine-wave.asd

6.5 Lissajous Figures

Once we have sinusoidal oscillations (or at least something similar), we can make Lissajous figures[62], as shown in `lissajous.asd`[63]:

```
<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>
```

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`

7.1 Converting ASDF Rotations to Rotation Matrices

To rotate objects in an ASDF scene, you can use *azimuth, elevation and roll angles* (page 11), for example like this:

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

The used coordinate system conventions are shown in the *section about position and orientation* (page 3).

In this section we show how these angles can be converted to rotation matrices[64], 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.

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

[62] https://en.wikipedia.org/wiki/Lissajous_curve
[63] https://github.com/AudioSceneDescriptionFormat/asdf/blob/25ffa03/doc/scenes/lissajous.asd
[64] https://en.wikipedia.org/wiki/Rotation_matrix

- Right-handed vs. left-handed coordinate system[65]: 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[66]: The ASDF uses a variation of Tait–Bryan.
- There are many possible conventions[67] for the order of angles and which axes they rotate around: The ASDF conventions are shown in detail below.
- "intrinsic"[68] vs. "extrinsic"[69] = "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")[70]: 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()[71] 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[72]: 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[73], 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 α, β and γ:

[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]: α

The *azimuth* angle α is:

- 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[74]).

[4]:
```
beta
```

[4]: β

The *elevation* angle β is:

[65] https://en.wikipedia.org/wiki/Coordinate_system
[66] https://en.wikipedia.org/wiki/Euler_angles
[67] https://en.wikipedia.org/wiki/Axes_conventions
[68] https://en.wikipedia.org/wiki/Euler_angles#Conventions_by_intrinsic_rotations
[69] https://en.wikipedia.org/wiki/Euler_angles#Conventions_by_extrinsic_rotations
[70] https://en.wikipedia.org/wiki/Active_and_passive_transformation
[71] https://docs.sympy.org/latest/modules/matrices/matrices.html#sympy.matrices.dense.rot_axis1
[72] https://en.wikipedia.org/wiki/Rotation_matrix#Ambiguities
[73] https://www.sympy.org/
[74] https://en.wikipedia.org/wiki/Right-hand_rule

- 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 γ 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[75] (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.

Azimuth: Rotation around the z-Axis

Writing the vector a in cylindrical coordinates r_z (radius), ϕ_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:

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

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

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 α:

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

(continues on next page)

[75] https://en.wikipedia.org/wiki/Port_and_starboard

(continued from previous page)

```
])
b
```

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

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

... 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]:
$$\begin{bmatrix} a_x \cos(\alpha) - a_y \sin(\alpha) \\ a_x \sin(\alpha) + a_y \cos(\alpha) \\ a_z \end{bmatrix}$$

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!

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 β 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]:
$$\begin{bmatrix} a_x \\ a_y \cos(\beta) - a_z \sin(\beta) \\ a_y \sin(\beta) + a_z \cos(\beta) \end{bmatrix}$$

... 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]:
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\beta) & -\sin(\beta) \\ 0 & \sin(\beta) & \cos(\beta) \end{bmatrix}$$

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?

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

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

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

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

[21]:
$$\begin{bmatrix} a_x \cos(\gamma) + a_z \sin(\gamma) \\ a_y \\ -a_x \sin(\gamma) + a_z \cos(\gamma) \end{bmatrix}$$

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

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

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

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

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

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),␣
↪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)]])
```

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 β 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 α.

We can simplify this expression:

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

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

Therefore, to get the angle α, 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()[76] in numeric calculations:

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

We can do a similar thing to get γ:

[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 γ.

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

[76] https://en.wikipedia.org/wiki/Atan2

Gimbal Lock

But wait a second, we might have a problem: the dreaded gimbal lock[77]!

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 α and γ like above, we end up calculating

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

Sadly, that is not defined:

[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 ± 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 α and γ from the hitherto unused matrix elements (but let's simplify the matrix first):

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

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

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

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

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

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

There is no unique solution to these equations. You can freely choose either α or γ 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
```

[77] https://en.wikipedia.org/wiki/Gimbal_lock

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

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

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

[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

7.2 Converting ASDF Rotations to Quaternions

This notebook shows the same thing as the *notebook about rotation matrices* (page 22), 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[78], 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')
```

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

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:

[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.

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.

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.

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

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
```

Quaternion to Rotation Matrix

Just to make sure the result is the same as in the *notebook about rotation matrices* (page 28), 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]:
$$\begin{bmatrix} -\sin(\alpha)\sin(\beta)\sin(\gamma)+\cos(\alpha)\cos(\gamma) & -\sin(\alpha)\cos(\beta) & \sin(\alpha)\sin(\beta)\cos(\gamma)+\sin(\gamma)\cos(\alpha) \\ \sin(\alpha)\cos(\gamma)+\sin(\beta)\sin(\gamma)\cos(\alpha) & \cos(\alpha)\cos(\beta) & \sin(\alpha)\sin(\gamma)-\sin(\beta)\cos(\alpha)\cos(\gamma) \\ -\sin(\gamma)\cos(\beta) & \sin(\beta) & \cos(\beta)\cos(\gamma) \end{bmatrix}$$

Quaternion to ASDF rotations

Again, please note that the equations from Wikipedia[79] 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]:
$$\begin{bmatrix} \frac{a^2+b^2-c^2-d^2}{a^2+b^2+c^2+d^2} & \frac{2(-ad+bc)}{a^2+b^2+c^2+d^2} & \frac{2(ac+bd)}{a^2+b^2+c^2+d^2} \\ \frac{2(ad+bc)}{a^2+b^2+c^2+d^2} & \frac{a^2-b^2+c^2-d^2}{a^2+b^2+c^2+d^2} & \frac{2(-ab+cd)}{a^2+b^2+c^2+d^2} \\ \frac{2(-ac+bd)}{a^2+b^2+c^2+d^2} & \frac{2(ab+cd)}{a^2+b^2+c^2+d^2} & \frac{a^2-b^2-c^2+d^2}{a^2+b^2+c^2+d^2} \end{bmatrix}$$

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]:
$$\begin{bmatrix} a^2+b^2-c^2-d^2 & -2ad+2bc & 2ac+2bd \\ 2ad+2bc & a^2-b^2+c^2-d^2 & -2ab+2cd \\ -2ac+2bd & 2ab+2cd & a^2-b^2-c^2+d^2 \end{bmatrix}$$

The *notebook about rotation matrices* (page 28) shows how to obtain α, β and γ from this matrix.

We can get α 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 β:

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

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

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

[21]:
```
print(_)
```
```
asin(2*a*b + 2*c*d)
```

Note:

As mentioned in the *notebook about rotation matrices* (page 28), 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, γ 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)
```

Gimbal Lock

As shown in the *notebook about rotation matrices* (page 30), 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]: $\operatorname{atan_2}\left(2ac + 2bd, 2ab - 2cd\right)$

[25]:
```
print(_)
```
```
atan2(2*a*c + 2*b*d, 2*a*b - 2*c*d)
```

If we for example choose this value to be α, 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]: $\operatorname{atan_2}\left(-2ac - 2bd, -2ab + 2cd\right)$

[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 α, 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.