## CONTENTS

1 Commands 3
   1.1 create-config ........................................ 3
   1.2 create-input-py ..................................... 3
   1.3 new .................................................. 3
   1.4 input-py ............................................. 3
   1.5 solve ................................................ 4
   1.6 report ................................................ 4

2 Examples .............................................. 5

3 Config File Format .................................. 7

4 Python Input Format ................................ 9

5 The concept of grooves in PyRolL 11
   5.1 The Generalized Groove .............................. 11
   5.2 Box-like Grooves .................................... 12
   5.3 Diamond-like grooves ............................... 13
   5.4 Round-like Grooves ................................ 14
   5.5 Oval-like Grooves ................................... 15
   5.6 Reference of Groove Classes ...................... 17

6 The Concept of Profiles 23
   6.1 Hooks ............................................... 25
   6.2 Derived classes ..................................... 26

7 Pass Sequence Units in PyRolL 27
   7.1 Roll Passes .......................................... 27
   7.2 Transports .......................................... 30

8 HTML Report Generation 31
   8.1 Class Documentation ................................ 31
   8.2 Hooks ............................................... 31

9 Data Export 33
   9.1 Specifying data to include ......................... 33
   9.2 Adding new file formats ............................ 33
   9.3 Class Documentation ................................ 34
   9.4 Hooks ............................................... 34

10 The Plugin System 35
PyRoL is an OpenSource rolling framework, aimed to provide a fast and extensible base for rolling simulation. The current focus lies on groove rolling in elongation grooves. The core packages comes with a basic set of models to allow a first estimation of forces and torques occurring in a pass sequence. There is a flexible plugin system, able to modify and extend the model set available to describe the process.

If the pyroll package is installed via pip, a command line tool name pyroll is installed alongside in the system.

    If the tool is not available, please check the content of your PATH environment variable.

The tool has the following syntax:

```
pyroll [OPTIONS] COMMAND1 [ARGS] ... [COMMAND2 [ARGS] ...]
```

As one can see, subcommands can be chained in one command line. For this reason subcommands take never arguments, but only options. Commands that rely on the data generated by another command must be chained, since no data persists between different runs of the tool.

At OPTIONS several gloabl options can be set, namely:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c, --configfile PATH</td>
<td>Give a path to a config YAML file. Per default the file ./config.yaml is used if it exists. See here for details.</td>
</tr>
<tr>
<td>-p, --plugin NAME</td>
<td>Give a plugin to load additionally to the ones specified in the config. Can be used multiple times.</td>
</tr>
</tbody>
</table>

The tool provides a set of subcommands explained in the following section.
1.1 create-config

Creates a standard config file with basic logging configuration and empty plugins list. The file is created in the path specified by the -f/--file option, which defaults to ./config.yaml.

1.2 create-input-py

Creates a sample input script for use with the input-py command. The file is created in the path specified by the -f/--file option, which defaults to ./input.py. This file can be used as is and modified as desired. It represents the conditions of the 3-high experimental rolling stand at the Institute of Metals Forming.

1.3 new

Creates a new PyRolL simulation project in the directory specified by -d/--dir. The directory will be created if not already existing. Creates a config.yaml and an input.py in the specified directory. This command is basically a shortcut for

```
pyroll -c <dir>/config.yaml create-config -p -f <dir>/config.yaml create-input-py -k min␣˓→-f <dir>/input.py
```

in a fresh or existing directory.

1.4 input-py

Reads input data from the file specified by the -f/--file option, which defaults to ./input.py. Use the create-input-py command to create a sample which can be modified. See here for more information on the format of this file.
1.5 solve

Runs the solution procedure for the pass sequence loaded by one of the input commands.

1.6 report

Generates a HTML report page from the simulation results. The contents of the page can be extended by plugins, please see [here](#) for additional information. The file to write to can be specified with the `-f/--file` option, which defaults to `report.html`. 
CHAPTER TWO

EXAMPLES

To read from a python script, solve and generate a report just use:

```
pyroll input-py solve report
```

To specify the files explicitly use:

```
pyroll input-py -f input.py solve report -f report.html
```

To use a different config file:

```
pyroll -c config2.yaml input-py solve report
```

To load additional plugins:

```
pyroll -p pyroll_plugin1 -p pyroll_plugin2 input-py solve report
```
CHAPTER THREE

CONFIG FILE FORMAT

The configuration has to be specified in YAML format. The content of the default config file is as follows:

```yaml
plugins: [ ] # list full qualified names of plugins to load (as they were importable in real python code)

logging: # configuration for the logging standard library package
  version: 1
  formatters:
    console:
      format: '[%(levelname)s] %(name)s: %(message)s'
    file:
      format: '%(asctime)s [%(levelname)s] %(name)s: %(message)s'

  handlers:
    console:
      class: logging.StreamHandler
      level: INFO
      formatter: console
      stream: ext://sys.stdout
    file:
      class: logging.FileHandler
      level: INFO
      formatter: file
      filename: pyroll.log

  root:
    level: INFO
    handlers: [ console, file ]

loggers:
  matplotlib:
    level: ERROR
  ...

In the plugins node several plugins can be specified to load additionally to the core functionalities. List the full qualified package name you want to load, as you would import in Python. For example to load the Wusatowski Spreading and the Integral Thermal plugins:

```yaml
plugins:
  - pyroll_wusatowski_spreading
  - pyroll_integral_thermal
```
The logging node configures logging using the Python standard `logging` package, see here for further information on that. The default config specifies logging on the console and to the `pyroll.log` file on information level. If you want more detailed logging, replace the `INFO` specifiers with `DEBUG`. To avoid log pollution by the `matplotlib` package, their level is set to `ERROR`. 
CHAPTER FOUR

PYTHON INPUT FORMAT

The most flexible way of defining input for PyRoIL is the direct use of a python script. A script loadable by the `input-py` command must define at least two variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_profile</td>
<td>A <code>Profile</code> object defining the properties of the incoming workpiece.</td>
</tr>
<tr>
<td>sequence</td>
<td>A list of <code>Unit</code> objects (either <code>RollPass</code> or <code>Transport</code>) defining the pass sequence.</td>
</tr>
</tbody>
</table>

A minimal input script is shown below:

```python
from pyroll.core.grooves import SquareGroove, DiamondGroove
from pyroll.core import Profile
from pyroll.core import RollPass
from numpy import pi

# initial profile
in_profile = Profile(
    width=68e-3,
    height=68e-3,
    groove=SquareGroove(r1=0, r2=3e-3, tip_angle=pi / 2, tip_depth=34e-3),
    temperature=1200 + 273.15,
    strain=0,
    material="C45",
    flow_stress=50e6
)

# pass sequence
sequence = [
    RollPass(
        label="Diamond I",
        groove=DiamondGroove(
            usable_width=76.55e-3,
            tip_depth=22.1e-3,
            r1=12e-3,
            r2=8e-3
        ),
        roll_radius=160e-3,
        velocity=1.4,
        gap=3e-3,
    ),
    RollPass(
```

(continues on next page)
The attributes to give as keyword arguments to the constructors depend on the plugins loaded. Most plugins need additional data about the pass sequence or the incoming profile. For information on the basic data needed see the docs of `Profile`, `RollPass` and `Transport`.
5.1 The Generalized Groove

All elongation grooves can be traced back to a generalized elongation groove consisting of two straights and four radii. The geometry of this is shown below.

All geometric key values like cross-sections and perimeters can be calculated on this generalized groove. The generalized groove is implemented in the GrooveBase class, all special groove types are derived from that.

In the following the measures of the groove are listed, their names are used in source code and throughout the documentation. The radii and angles are numbered from outside to inside.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>Radius</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Angle corresponding to a radius</td>
</tr>
<tr>
<td>$\beta$, $\gamma$</td>
<td>Angles useful for coordinate calculation</td>
</tr>
<tr>
<td>$b_d$</td>
<td>Ground width</td>
</tr>
<tr>
<td>$b_{\text{even}}$</td>
<td>Even ground width</td>
</tr>
<tr>
<td>$b_{\text{ks}}$</td>
<td>Tip width</td>
</tr>
<tr>
<td>$b_{\text{kn}}$</td>
<td>Usable width</td>
</tr>
<tr>
<td>$d$</td>
<td>Depth</td>
</tr>
<tr>
<td>$i$</td>
<td>Indent</td>
</tr>
<tr>
<td>$s$</td>
<td>Roll gap</td>
</tr>
</tbody>
</table>

The coordinates of the points 1 to 12 shown in the figure can be calculated as follows, where the angles $\beta = \alpha_4 - \alpha_3/2$ and $\gamma = \frac{\pi}{2} - \alpha_2 - \alpha_3 + \alpha_4$. 
However, in the current implementation the term “groove” is more narrow. From now on, the term should represent only the shape machined into the roll surface. Therefore, the roll gap $s$ is no measure of the groove itself but of the RollPass. Also, the tip width $b_{kn}$ is not inherent to the groove, since it depends on the roll gap.

5.2 Box-like Grooves

5.2.1 The BoxGroove class

The BoxGroove class represents a rectangular shaped groove as shown in the figure. For wear reasons, the flanks a typically inclined by a small angle.

Mandatory measures of the box groove are the two radii $r_1$ and $r_2$, as well as the depth $d$. To constrain geometry fully, any two of the following must be given:

- usable width $b_{kn}$
- ground width $b_d$
- flank angle $\alpha_1$

So the constructor has the following signature:

```
BoxGroove(r1, r2, depth, usable_width, ground_width, flank_angle)
```

The radii are typically small, the depth is $d$ typically $\leq b_{kn}$. $r_3$ and $r_4$ are considered to be zero.

$b_d$ was chosen in favor of the even ground width $b'_d$, because it does not change when the radii are modified. So the overall geometry remains the same if one modifies only the radii.
5.2.2 The ConstrictedBoxGroove class

The ConstrictedBoxGroove class represents a BoxGroove but with an indent in the ground as shown in the figure.

Mandatory measures of the box groove are the two radii \( r_1 \) and \( r_2 \), as well as the depth \( d \) and the indent \( i \). To constrain geometry fully, any two of the following must be given:

- usable width \( b_{kn} \)
- ground width \( b_d \)
- flank angle \( \alpha_1 \)

So the constructor has the following signature:

\[
\text{ConstrictedBoxGroove}(r_1, r_2, \text{depth}, \text{indent}, \text{usable_width}, \text{ground_width}, \text{flank_angle})
\]

The radii are typically small, the depth is \( d \) typically \( \leq \frac{b_{kn}}{2} \).

\( r_3 \) and \( r_4 \) are considered to be zero.

5.3 Diamond-like grooves

5.3.1 The DiamondGroove class

The DiamondGroove class represents a rhombus shaped groove as shown in the figure.

Mandatory measures of this groove are the two radii \( r_1 \) and \( r_2 \). To constrain geometry fully, any two of the following must be given:

- usable width \( b_{kn} \)
- tip depth \( d_t \)
- tip angle \( \delta \)

So the constructor has the following signature:

\[
\text{DiamondGroove}(r_1, r_2, \text{usable_width}, \text{tip_depth}, \text{tip_angle})
\]

The radii are typically small, the depth is \( d_t \) typically \( < \frac{b_{kn}}{2} \) so that the tip angle \( \delta \) is larger than \( 90^\circ \).

\( r_3 \) and \( r_4 \) are considered to be zero, as well as \( b_d \) and \( b'_d \).

The tip depth \( d_t \) was chosen in favor of the real depth \( d \), because it does not change, when the radii are modified. So the overall geometry remains the same if one modifies only the radii. The tip depth can be considered as the diagonal of the rhombus with sharp corners.
5.3.2 The SquareGroove class

The SquareGroove class represents a square shaped groove as shown in the figure.

Mandatory measures of this groove are the two radii $r_1$ and $r_2$. To constrain geometry fully, any two of the following must be given:

- usable width $b_{kn}$
- tip depth $d_t$
- tip angle $\delta$

So the constructor has the following signature:

```
SquareGroove(r1, r2, usable_width, tip_depth, tip_angle)
```

The radii are typically small, the depth is $d_t$ typically $\approx \frac{b_{kn}}{4}$. The tip angle $\delta$ is typically a one or two degree larger than 90° for wear reasons.

$r_3$ and $r_4$ are considered to be zero, as well as $b_d$ and $b'_d$.

The tip depth $d_t$ was chosen in favor of the real depth $d$, because it does not change, when the radii are modified. So the overall geometry remains the same if one modifies only the radii. The tip depth can be considered as the diagonal of the square with sharp corners.

The constructor will raise a warning, if the tip angle significantly deviates from 90°, consider to use a DiamondGroove instead.

5.4 Round-like Grooves

5.4.1 The RoundGroove class

The RoundGroove class represents a groove with a circular cross-section as shown in the figure.

It is defined by two radii $r_1$ and $r_2$ and the depth $d$, so the constructor has the following signature:

```
RoundGroove(r1, r2, depth)
```

The geometric constraints are $r_1 < r_2$ and $d < r_2$.

$r_3$ and $r_4$ are considered to be zero, as well as $b_d$ and $b'_d$.

The angles can be calculated as following:

$$\alpha_1 = \alpha_2 = \arccos \left(1 - \frac{d}{r_1 + r_2}\right)$$

The usable width is then:

$$b_{kn} = 2 \left( r_1 \sin \alpha_1 + r_2 \sin \alpha_2 - r_1 \tan \frac{\alpha_1}{2} \right)$$
5.4.2 The FalseRoundGroove class

The FalseRoundGroove class represents a groove with a roughly circular cross-section, which shows a small straight flank, as shown in the figure.

It is defined by two radii $r_1$ and $r_2$, the depth $d$ and the flank angle $\alpha_1$, so the constructor has the following signature:

```
FalseRoundGroove(r1, r2, depth, flank_angle)
```

The geometric constraints are $r_1 << r_2$, $d < r_2$ and $\alpha_1 < 90$. $r_3$ and $r_4$ are considered to be zero, as well as $b_d$ and $b'_d$.

The usable width can be calculated as:

$$b_{kn} = 2 \frac{d + \frac{r_2}{\cos \alpha_1} - r_2}{\tan \alpha_1}$$

5.5 Oval-like Grooves

5.5.1 The CircularOvalGroove class

The CircularOvalGroove class represents an oval shaped groove consisting of two radii as shown in the figure.

It is defined by two radii $r_1$ and $r_2$ and the depth $d$, so the constructor has the following signature:

```
CircularOvalGroove(r1, r2, depth)
```

The geometric constraints are $r_1 << r_2$ and $d << r_2$.

$r_3$ and $r_4$ are considered to be zero, as well as $b_d$ and $b'_d$.

The topology of this groove is similar to the RoundGroove, with the main difference, that the center of $r_2$ is not placed in the center of the groove. For this reason $d$ is typically much smaller than $r_2$.

5.5.2 The FlatOvalGroove class

The FlatOvalGroove class represents an oval shaped groove consisting of two radii and an even ground as shown in the figure.

Mandatory measures of this groove are the two radii $r_1$ and $r_2$, as well as the depth $d$ and the usable width $b_{kn}$.

So the constructor has the following signature:

```
FlatOvalGroove(r1, r2, depth, usable_width)
```

The depth is $d$ typically $\leq \frac{b_{kn}}{2}$.

$r_3$ and $r_4$ are considered to be zero.
5.5.3 The SwedishOvalGroove class

The SwedishOvalGroove class represents a hexagonal shaped groove as shown in the figure. The term “hexagonal” is also used for this type of groove, but can be confused with regular hexagon shaped grooves. The current type of groove is used as an oval and therefore the term swedish oval should be used, which is derived from its origin in swedish steel plants.

Mandatory measures of this groove are the two radii \( r_1 \) and \( r_2 \), as well as the depth \( d \). To constrain geometry fully, any two of the following must be given:

- usable width \( b_{kn} \)
- ground width \( b_d \)
- flank angle \( \alpha_1 \)

So the constructor has the following signature:

SwedishOvalGroove(r1, r2, depth, usable_width, ground_width, flank_angle)

The radii are typically small, the depth is \( d \) typically \( \ll \frac{b_{kn}}{2} \).

\( r_3 \) and \( r_4 \) are considered to be zero.

\( b_d \) was chosen in favor of the even ground width \( b'_d \), because it does not change when the radii are modified. So the overall geometry remains the same if one modifies only the radii.

The topology of this groove is similar to the BoxGroove, but typically the flank angles are smaller and the groove is less deep.

5.5.4 The ConstrictedSwedishOvalGroove class

The ConstrictedSwedishOvalGroove class represents a SwedishOvalGroove but with an indent in the ground as shown in the figure.

Mandatory measures of this groove are the two radii \( r_1 \) and \( r_2 \), as well as the depth \( d \) and the indent \( 'i' \). To constrain geometry fully, any two of the following must be given:

- usable width \( b_{kn} \)
- ground width \( b_d \)
- flank angle \( \alpha_1 \)

So the constructor has the following signature:

ConstrictedSwedishOvalGroove(r1, r2, depth, indent, usable_width, ground_width, flank_angle)

The radii are typically small, the depth is \( d \) typically \( \ll \frac{b_{kn}}{2} \).

\( r_3 \) and \( r_4 \) are considered to be zero.
5.5.5 The Oval3RadiiGroove class

The Oval3RadiiGroove class represents an oval shaped groove consisting of three radii as shown in the figure.

Mandatory measures of this groove are the three radii \( r_1, r_2 \) and \( r_3 \), as well as the depth \( d \) and the usable width \( b_{kn} \).

So the constructor has the following signature:

```python
Oval3RadiiGroove(r1, r2, r3, depth, usable_width)
```

The depth \( d \) typically \( \leq b_{kn} \).

\( r_4 \) and \( b_d' \) are considered to be zero.

5.5.6 The Oval3RadiiFlankedGroove class

The Oval3RadiiFlankedGroove class represents an oval shaped groove consisting of three radii and a small straight flank as shown in the figure.

Mandatory measures of this groove are the three radii \( r_1, r_2 \) and \( r_3 \), as well as the depth \( d \), the usable width \( b_{kn} \) and the flank angle \( \alpha_1 \).

So the constructor has the following signature:

```python
Oval3RadiiFlankedGroove(r1, r2, r3, depth, usable_width, flank_angle)
```

The depth \( d \) typically \( \leq b_{kn} \).

\( r_4 \) and \( b_d' \) are considered to be zero.

5.6 Reference of Groove Classes

```python
class BoxGroove(r1: float, r2: float, depth: float, ground_width: Optional[float] = None, usable_width: Optional[float] = None, flank_angle: Optional[float] = None)
```

Represents a box shaped groove.

Exactly two of `ground_width`, `usable_width` and `flank_angle` must be given.

**Parameters**

- `r1` (`float`) – radius of the first edge
- `r2` (`float`) – radius of the second edge
- `depth` (`float`) – depth of the groove
- `ground_width` (`float`) – width of the groove from intersection between two flanks and ground width
- `usable_width` (`float`) – ground width excluding influence of radii
- `flank_angle` (`float`) – angle of the flanks

**Raises**

`ValueError` – if not exactly two of `ground_width`, `usable_width` and `flank_angle` are given
property types: 'box'
A tuple of keywords to specify the types of this groove.

class CircularOvalGroove(r1: float, r2: float, depth: float)
Represents an oval shaped groove with one main radius.

Parameters
• r1 (float) – radius of the first edge
• r2 (float) – radius of the second edge
• depth (float) – depth of the groove

property types: 'oval', 'circular_oval'
A tuple of keywords to specify the types of this groove.

class ConstrictedBoxGroove(r1: float, r2: float, r4: float, depth: float, indent: float, ground_width: Optional[float] = None, usable_width: Optional[float] = None, flank_angle: Optional[float] = None)
Represents a box shaped groove with an indented ground.

Exactly two of ground_width, usable_width and flank_angle must be given.

Parameters

property types: 'box', 'constricted_box'
A tuple of keywords to specify the types of this groove.

class ConstrictedSwedishOvalGroove(r1: float, r2: float, r4: float, depth: float, indent: float, ground_width: Optional[float] = None, usable_width: Optional[float] = None, flank_angle: Optional[float] = None)
Represents a hexagonal shaped groove with an indented ground that is used like an oval groove (swedish oval).

Exactly two of ground_width, usable_width and flank_angle must be given.

Parameters
• r1 (float) – radius of the first edge
• r2 (float) – radius of the second edge
• depth (float) – depth of the groove
• indent (float) – indentation of the depth of the groove towards the grooves center
• ground_width (float) – width of the groove from intersection between two flanks and ground width
• usable_width (float) – ground width excluding influence of radii
• flank_angle (float) – angle of the flanks

Raises
ValueError – if not exactly two of ground_width, usable_width and flank_angle are given
property types: 'oval', 'swedish_oval', 'constricted_swedish_oval'
A tuple of keywords to specify the types of this groove.

class DiamondGroove(r1: float, r2: float, usable_width: Optional[float] = None, tip_depth: Optional[float] = None, tip_angle: Optional[float] = None)
Represent a diamond shaped groove.
Exactly two of usable_width, tip_depth and tip_angle must be given.

Parameters
• r1 (float) – radius of the first edge
• r2 (float) – radius of the second edge
• usable_width (float) – ground width excluding influence of radii
• tip_depth (float) – depth of the tip of the groove
• tip_angle (float) – angle at which the tip is formed

Raises
ValueError – if not exactly two of usable_width, tip_depth and tip_angle are given

property types: 'diamond'
A tuple of keywords to specify the types of this groove.

class FalseRoundGroove(r1: float, r2: float, depth: float, flank_angle: float)
Represents a round shaped groove with a dedicated flank (false round).

Parameters
• r1 (float) – radius of the first edge
• r2 (float) – radius of the second edge
• depth (float) – depth of the groove
• flank_angle (float) – angle of the flanks

property types: 'round', 'false_round'
A tuple of keywords to specify the types of this groove.

class FlatGroove(width: float)
Represents a box shaped groove.

Parameters
width (float) – width of the forming zone

property types: 'flat'
A tuple of keywords to specify the types of this groove.

class FlatOvalGroove(r1: float, r2: float, depth: float, usable_width: float)
Represent an oval shaped groove with a flat ground.

Parameters
• r1 (float) – radius of the first edge
• r2 (float) – radius of the second edge
• depth (float) – depth of the groove
• usable_width (float) – ground width excluding influence of radii
property types: 'oval', 'flat_oval'
A tuple of keywords to specify the types of this groove.

class GenericElongationGroove(usable_width: float = 0, depth: float = 0, r1: float = 0, r2: float = 0, r3: float = 0, r4: float = 0, even_ground_width: float = 0, indent: float = 0, alpha1: float = 0, alpha2: float = 0, alpha3: float = 0, alpha4: float = 0, types: Tuple[str, ...] = ())

Represents a groove defined by the generic elongation groove geometry.

property contour_line: LineString
A line representing the geometry of the groove contour.

property cross_section: Polygon
A polygon representing the cross-section of this groove limited by the contour line and y=0.

property depth: float
The maximum depth of the groove.

local_depth(z) → Union[float, ndarray]
Function of the local groove depth in dependence on the z-coordinate.

property types: Tuple[str, ...]
A tuple of keywords to specify the types of this groove.

property usable_width: float
The usable width of the groove, meaning the width of ideal filling.

class GrooveBase
Abstract base class for all grooves.

abstract property contour_line: LineString
A line representing the geometry of the groove contour.

abstract property cross_section: Polygon
A polygon representing the cross-section of this groove limited by the contour line and y=0.

abstract property depth: float
The maximum depth of the groove.

abstract local_depth(z: Union[float, ndarray]) → Union[float, ndarray]
Function of the local groove depth in dependence on the z-coordinate.

abstract property types: Tuple[str, ...]
A tuple of keywords to specify the types of this groove.

abstract property usable_width: float
The usable width of the groove, meaning the width of ideal filling.

class Oval3RadiiFlankedGroove(r1: float, r2: float, r3: float, depth: float, usable_width: float, flank_angle: float)

Represents an oval shaped groove with 3 main radii and a dedicated flank.

Exactly two of ground_width, usable_width and flank_angle must be given.

Parameters
- r1 (float) – radius of the first edge
- r2 (float) – radius of the second edge
• **depth** (*float*) – depth of the groove
• **usable_width** (*float*) – ground width excluding influence of radii
• **flank_angle** (*float*) – angle of the flanks

**Raises**

```
ValueError
```

– if not exactly two of ground_width, usable_width and flank_angle are given

**property types:** 'oval', 'oval_3_radii', 'oval_3_radii_flanked'

A tuple of keywords to specify the types of this groove.

### class Oval3RadiiGroove

```
(r1: float, r2: float, r3: float, depth: float, usable_width: float)
```

Represents an oval shaped groove with 3 main radii.

**Parameters**

• **r1** (*float*) – radius of the first edge
• **r2** (*float*) – radius of the second edge
• **r3** – radius of the third edge
• **depth** (*float*) – depth of the groove
• **usable_width** (*float*) – ground width excluding influence of radii

**property types:** 'oval', 'oval_3_radii'

A tuple of keywords to specify the types of this groove.

### class RoundGroove

```
(r1: float, r2: float, depth: float)
```

Represents a round shaped groove.

**Parameters**

• **r1** (*float*) – radius of the first edge
• **r2** (*float*) – radius of the second edge
• **depth** (*float*) – depth of the groove

**property types:** 'round'

A tuple of keywords to specify the types of this groove.

### class SplineGroove

```
(contour_points: Union[Iterable[Tuple[float, float]], ndarray], types: Iterable[str],
 usable_width: Optional[float] = None)
```

Represents a groove defined by a linear spline contour.

**Parameters**

• **contour_points** – an iterable of contour points to be used for the spline
• **types** – an iterable of string keys used as type classifiers
• **usable_width** – the usable width to assume for this instance, if None, the maximum width will be used

**property contour_line:** *LineString*

A line representing the geometry of the groove contour.

**property cross_section:** *Polygon*

A polygon representing the cross-section of this groove limited by the contour line and y=0.

---

5.6. Reference of Groove Classes
property depth: float
    The maximum depth of the groove.

local_depth(z) \rightarrow \text{Union[float, ndarray]}
    Function of the local groove depth in dependence on the z-coordinate.

property types: Tuple[str, ...]
    A tuple of keywords to specify the types of this groove.

property usable_width: float
    The usable width of the groove, meaning the width of ideal filling.

class SquareGroove(r1: float, r2: float, usable_width: Optional[float] = None, tip_depth: Optional[float] = None, tip_angle: Optional[float] = None)
    Represents a square shaped groove (diamond with tip angle near 90°).
    Exactly two of usable_width, tip_depth and tip_angle must be given.
    
    Parameters
    • r1 (float) – radius of the first edge
    • r2 (float) – radius of the second edge
    • usable_width (float) – ground width excluding influence of radii
    • tip_depth (float) – depth of the tip of the groove
    • tip_angle (float) – angle at which the tip is formed
    
    Raises
    • ValueError – if not exactly two of usable_width, tip_depth and tip_angle are given
    • ValueError – if tip angle is <85° or >95° (no matter if given or calculated internally)

    property types: 'diamond', 'square'
    A tuple of keywords to specify the types of this groove.

class SwedishOvalGroove(r1: float, r2: float, depth: float, ground_width: Optional[float] = None, usable_width: Optional[float] = None, flank_angle: Optional[float] = None)
    Represents a hexagonal shaped groove that is used like an oval groove (swedish oval).
    Exactly two of ground_width, usable_width and flank_angle must be given.
    
    Parameters
    • r1 (float) – radius of the first edge
    • r2 (float) – radius of the second edge
    • depth (float) – depth of the groove
    • ground_width (float) – width of the groove from intersection between two flanks and ground width
    • usable_width (float) – ground width excluding influence of radii
    • flank_angle (float) – angle of the flanks
    
    Raises
    ValueError – if not exactly two of ground_width, usable_width and flank_angle are given

    property types: 'oval', 'swedish_oval'
    A tuple of keywords to specify the types of this groove.
CHAPTER
SIX

THE CONCEPT OF PROFILES

Think of a Profile object as of a state of the workpiece anywhere in the pass sequence. Every sequence unit has an incoming and an outgoing profile. Also, you must provide a profile as definition for the initial workpiece being processed in the pass sequence.

Profiles Each profile has a certain shape, defined by the upper_contour_line and lower_contour_line and its main dimensions height and width.

For creating an initial profile, several class methods exist in the Profile class. One can either derive the profile shape from an existing groove object by use of the Profile.from_groove() method, or created some standard shapes use of the other class methods of Profile, like Profile.round(). More values can be given as keyword arguments and are saved automatically as attributes in the instance. Which you may or must provide depends on the loaded plugins.

class Profile(**kwargs)

Parameters
    hook_args – keyword arguments to pass to hook calls

classmethod box(height: float, width: float, corner_radius: float = 0, **kwargs) → Profile

Creates a box shaped profile (a real rectangular shape with rounded corners, without imperfections of box grooves). A box is oriented to stand on its side, use square() to create a corner standing square.

Parameters
    • height – the height of the box profile, must be > 0
    • width – the width of the box profile, must be > 0
    • corner_radius – the radius of the square’s corners, must be >= 0, <= height / 2 and <= width / 2
    • kwargs – additional keyword arguments to be passed to the Profile constructor

Raises
    ValueError – if arguments are out of range

classmethod diamond(height: float, width: float, corner_radius: float = 0, **kwargs) → Profile

Creates a diamond shaped profile (a real diamond shape with rounded corners, without imperfections of diamond grooves). A diamond is oriented to stand on its corner.

Parameters
    • height – the height of the diamond profile, must be > 0
    • width – the width of the diamond profile, must be > 0
    • corner_radius – the radius of the diamond’s corners, must be >= 0, <= height / 2 and <= width / 2
    • kwargs – additional keyword arguments to be passed to the Profile constructor
Raises

ValueError – if arguments are out of range

classmethod from_groove(groove: GrooveBase, width: Optional[float] = None, filling: Optional[float] = None, height: Optional[float] = None, gap: Optional[float] = None, **kwargs) → Profile

Create a profile instance based on a given groove. The dimensioning of the profile is determined by the parameters width, filling, height and gap. Give exactly one of width and filling. Give exactly one of height and gap.

Parameters

- groove – the groove the profile should be created from
- width – the width of the resulting profile, must be > 0
- filling – the filling ratio of the groove, must be > 0
- height – the height of the profile, must be > 0
- gap – the gap between the groove contours (roll gap), must be >= 0
- kwargs – additional keyword arguments to be passed to the Profile constructor

Raises

- TypeError – on invalid argument combinations
- ValueError – if arguments are out of range

hook_result_attributes: Set[str]

Set remembering all hooks that were called on this class, used by delete_hook_result_attributes().

classmethod round(radius: Optional[float] = None, diameter: Optional[float] = None, **kwargs) → Profile

Creates a round shaped profile (a real circle round, without imperfections of round grooves). Give exactly one of radius and diameter.

Parameters

- radius – the radius of the round profile, must be > 0
- diameter – the diameter of the round profile, must be > 0
- kwargs – additional keyword arguments to be passed to the Profile constructor

Raises

- TypeError – on invalid argument combinations
- ValueError – if arguments are out of range

classmethod square(side: Optional[float] = None, diagonal: Optional[float] = None, corner_radius: float = 0, **kwargs) → Profile

Creates a square shaped profile (a real square with rounded corners, without imperfections of square grooves). A square is oriented to stand on its corner, use box() to create a side standing one. Give exactly one of side and diagonal.

Parameters

- side – the side length of the square profile, must be > 0
- diagonal – the diagonal’s length of the square profile, must be > 0. Note, that the diagonal is measured at the tips, as if the corner radii were not present for consistency with box().
• corner_radius – the radius of the square’s corners, must be >= 0 and <= side / 2
• kwargs – additional keyword arguments to be passed to the Profile constructor

Raises
• TypeError – on invalid argument combinations
• ValueError – if arguments are out of range

6.1 Hooks

To read about the basics of hooks and plugins, see here.
The following hooks are defined on plain profiles per default:
cross_section(profile: Profile) → Polygon
Cross-section polygon of the profile.
equivalent_rectangle(profile: Profile) → Polygon
Get the dimensions of the equivalent rectangle of the profile.
flow_stress(profile: Profile) → str
Flow stress of workpiece material.
height(profile: Profile) → float
Height of the profile.
lower_contour_line(profile: Profile) → LineString
Lower bounding contour line of the profile.
material(profile: Profile) → Union[str, Iterable[str]]
Material identifier string for use in several other hooks to get material properties.
strain(profile: Profile) → float
Equivalent strain of the profile.
temperature(profile: Profile) → float
Temperature of the profile.
types(profile: Profile) → float
A tuple of keywords to specify the shape types of the profile.
upper_contour_line(profile: Profile) → LineString
Upper bounding contour line of the profile.
width(profile: Profile) → float
Width of the profile.
6.2 Derived classes

For the units types RollPass and Transport, specialized versions of the Profile class are defined as nested classes within the respective unit class. They all maintain their own hooks, so it is possible to specify hooks on profiles only for those places, were they are applicable.

All hooks on those classes receive additionally to the profile instance also the instance of the roll pass or transport they are belonging to.
This part of the documentation is currently work in progress.

Think of a rolling process as of a sequence of roll passes and intermediate times, called transports. Both are subsumed under the term unit. The Unit class is the base class representing this concept. A unit can most abstractly be considered as a black box transforming the state of a profile, thus taking an incoming profile instance, simulation its evolution within the unit and yielding an outgoing profile instance.

It defines three attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>A label string for human identification, used in log messages and output.</td>
</tr>
<tr>
<td>in_profile</td>
<td>The profile that represents the incoming workpiece state of the unit.</td>
</tr>
<tr>
<td>out_profile</td>
<td>The profile that represents the outgoing workpiece state of the unit.</td>
</tr>
</tbody>
</table>

Currently, two derived classes exist in the core library: RollPass and Transport.

The unit class defines an abstract method `solve(in_profile: Profile)`, which triggers the solution procedure and accepts a profile object that has to be treated as the incoming profile. This object is copied and modified and made available in the in_profile attribute by the implementations of this method.

Also, the Unit class maintains hooks that should be applicable to all types of units.

To read about the basics of hooks and plugins, see here.

### 7.1 Roll Passes

The roll pass is the most important unit, since forming of the workpiece is happening here. It is represented by the RollPass class. The RollPass constructor takes a Roll object, which is defining the properties of the working rolls including the groove.

#### 7.1.1 Rolls

Roll objects represent a working roll implemented in a rolling stand. The main properties are about the geometry and rotational movement of the roll. Rolls define the basic hooks specified below. With appropriate plugins, elastic deformation of the rolls during the process can be modelled.
7.1.2 Hooks

To read about the basics of hooks and plugins, see here.

On roll passes, several basic hooks are specified and implemented. You can provide your own implementations of them and also specify new ones.

The figure below shows an overview over the respective classes and their hook function signature.

The following are defined by default.

RollPass

\texttt{gap}(roll\_pass: \texttt{RollPass}) \rightarrow \text{float}
Gap between the rolls.

\texttt{height}(roll\_pass: \texttt{RollPass}) \rightarrow \text{float}
Maximum height of the pass contour.

\texttt{in\_profile\_rotation}(roll\_pass: \texttt{RollPass}) \rightarrow \text{float}
Rotation of the in profile for the specified roll pass.

\texttt{mean\_flow\_stress}(roll\_pass: \texttt{RollPass}) \rightarrow \text{float}
Mean flow stress of the material for the respected roll pass.

\texttt{roll}(roll\_pass: \texttt{RollPass}) \rightarrow \text{float}
Object representing the working rolls of the roll pass.

\texttt{roll\_force}(roll\_pass: \texttt{RollPass}) \rightarrow \text{float}
Roll force of the pass.

\texttt{spread}(roll\_pass: \texttt{RollPass}) \rightarrow \text{float}
Spread in the pass as ratio b1/b0.

\texttt{strain\_change}(roll\_pass: \texttt{RollPass}) \rightarrow \text{float}
Applied strain in the pass.

\texttt{strain\_rate}(roll\_pass: \texttt{RollPass}) \rightarrow \text{float}
Mean strain rate in the pass.

\texttt{tip\_width}(roll\_pass: \texttt{RollPass}) \rightarrow \text{float}
Tip width of the pass contour.

\texttt{velocity}(roll\_pass: \texttt{RollPass}) \rightarrow \text{float}
Mean rolling velocity.

\texttt{volume}(roll\_pass: \texttt{RollPass}) \rightarrow \text{float}
Volume of the workpiece within the roll gap.
Roll

**contour_line**(*roll*: Roll) → LineString
Contour line of the roll’s surface.

**groove**(*roll*: Roll) → GrooveBase
Object representing the groove shape carved into the roll.

**max_radius**(*roll*: Roll) → float
Maximal (outer) radius of the roll.

**min_radius**(*roll*: Roll) → float
Minimal (inner) radius of the roll.

**nominal_radius**(*roll*: Roll) → float
Nominal radius of the roll (equal to the grooves y=0 axis).

**rotational_frequency**(*roll*: Roll) → float
The rotational frequency of the roll.

**working_radius**(*roll*: Roll) → float
Working radius of the roll (some kind of equivalent radius to flat rolling).

RollPass.Roll

**contact_area**(*roll_pass*: RollPass, *roll*: Roll) → float
Area of contact between workpiece and one roll.

**contact_length**(*roll_pass*: RollPass, *roll*: Roll) → float
Contact length in rolling direction between rolls and workpiece.

**roll_torque**(*roll_pass*: RollPass, *roll*: Roll) → float
Roll torque of the pass.

RollPass.Profile

**flow_stress**(*roll_pass*: RollPass, *profile*: Profile) → float
Flow stress of workpiece material.

RollPass.OutProfile

**filling_ratio**(*roll_pass*: RollPass, *profile*: OutProfile) → float
Filling ratio of profile width to usable groove width.

**strain**(*roll_pass*: RollPass, *profile*: OutProfile) → float
Strain of the out profile.

**width**(*roll_pass*: RollPass, *profile*: OutProfile) → float
Width of the out profile.

Below you will find detailed descriptions of selected hooks as example of using them.
in_profile_rotation

The angle in degree by which the incoming profile is rotated at feeding into the roll pass. Currently only integers are valid values. Per default common rotations are implemented for the available groove types. Typically you will use the applies_to_in_grooves and applies_to_in_grooves decorators from pyroll.utils to provide new implementations. The code block below shows an example implementation of this hook, the explicit specname is used to avoid naming conflicts when providing more than one implementation in one file.

```python
@RollPass.hookimpl(specname="in_profile_rotation")
@applies_to_in_grooves("diamond")
@applies_to_out_grooves("diamond")
def diamonds(roll_pass):
    return 90
```

7.2 Transports

7.2.1 Hooks

To read about the basics of hooks and plugins, see here.

On transports, several basic hooks are specified and implemented. You can provide your own implementations of them and also specify new ones.

The figure below shows an overview over the respective classes and their hook function signature.

The following hooks are defined by default.

Transport

```python
duration(transport: Transport) → float
```

Duration of the transport.

Transport.OutProfile

```python
strain(transport: Transport, profile: OutProfile) → float
```

The equivalent strain of the outgoing profile of the transport unit.
PyRoLL includes a class capable of generating an HTML page presenting the simulation results, which can be archived and printed. The report can be generated by use of the CLI through the `report` command.

The report includes per default key properties of all units, plots of incoming and outgoing profiles in each roll pass, as well as some plots of key values along the whole sequence. The contents of the report can be modified using hooks.

The report includes tables listing properties of units or the whole sequence. One can customize the properties shown there by providing hook implementations. For the table listing unit properties use the `unit_properties()` hook. For the table listing properties of the whole sequence, use the `sequence_properties()` hook. The hook implementations must return in both cases a mapping from string keys to values. The keys are printed in the first column of the table. The values may be of any type, but they should have meaningful `__str__` methods to lead to feasible results.

The report includes several plots. Plots visualizing the whole sequence of units or parts of it can be added by providing an implementation of the `sequence_plot()` hook. Plots visualizing a single unit can be added by providing an implementation of the `unit_plot()` hook. The hook implementations must return in both cases an instance of matplotlib’s `Figure` class.

### 8.1 Class Documentation

**class Reporter**

Class able to generate an HTML report sheet out of simulation results.

```python
render(units: List[Unit]) → str
```

Render an HTML report from the specified units list.

- **Parameters**
  - `units` – list of units to take the data from

- **Returns**
  - generated HTML code as string

### 8.2 Hooks

**sequence_plot(units: List[Unit]) → Figure**

Generate a matplotlib figure visualizing the whole pass sequence, f.e. plot the distribution of roll forces. All loaded hook implementations are listed in the report.

**sequence_properties(units: List[Unit]) → Mapping[str, Any]**

Extract some data from the unit sequence to be listed in the report. Return a mapping of label names to values. All hook implems will be joined in order of definition.
unit_plot(unit: Unit) → Figure

Generate a matplotlib figure visualizing a unit. All loaded hook implementations are listed in the report.

unit_properties(unit: Unit) → Mapping[str, Any]

Extract some data from a unit to be listed in the report. Return a mapping of label names to values. All hookimpls will be joined in order of definition.
PyRolL includes a class capable of converting the simulation results to a pandas DataFrame and save this to different file formats. The feature can be accessed by use of the CLI through the export command.

The data included in the frame can be modified by hooks. The available file formats can be extended by the use of hooks.

To read about the basics of hooks and plugins, see [here](#).

### 9.1 Specifying data to include

There is a hook `columns(unit : Unit)` that can be used to specify the columns included in the data frame. One can use the `pyroll.utils.hookutils.applies_to_unit_types(types)` decorator to specify the unit types the hook implementation should apply to (currently only `Unit`, `RollPass`, `Transport`).

Each implementation must return a mapping of column names (string) to values (any type that can be data in a DataFrame). The list of hook results will be combined to the final set of columns. Later registered implementations will override earlier ones.

Define new implementations of this hook to include more data in the export. Commonly you would return a dict mapping from `str` to a numeric type or string.

### 9.2 Adding new file formats

For exporting to a file a hook is defined to handle the formatting:

```python
export(data: pandas.DataFrame, export_format: str)
```

It takes the generated DataFrame and a string specifying the format as arguments. Depending on the value of `export_format` an implementation can decide whether it is able to handle the format or not. If it can, it should return the binary data that will be saved to file. If it can not, it should return `None`. The first implementation not returning `None` will be used for the file content (`firstresult`).

Current basic implementations support CSV and XML formats by use of the methods provided by DataFrame.
9.3 Class Documentation

class Exporter
   Class able to export simulation results to several data formats.

   `export(units: List[Unit], export_format: str) → bytes`
   Call `get_dataframe` and export its results to a specified format.

   Parameters
   • `units` – list of units to take the data from
   • `export_format` – a string key identifying the export format, valid values depend on the
     loaded implementations of the `export` hook

   Returns
   the exported data as binary stream

get_dataframe(units: List[Unit]) → DataFrame
   Generate a pandas DataFrame by use of the `unit_columns`, `roll_pass_columns` and `transport_columns`
   hooks.

   Parameters
   `units` – list of units to take the data from

   Returns
   a pandas data frame filled with the exported data

9.4 Hooks

columns(unit) → Mapping[str, Any]
   Take a unit object and extract some data to be listed in the CSV output. Return a mapping of column names to
   values. All hookimpls will be joined in order of definition.

export(data: DataFrame, export_format: str) → bytes
   Export the data to a specified format. Return binary data that can be saved to a file. First hookimpl that does not
   return None is taken. Return None to signal, that the impl does not support the `export_format`
PyRoL is mainly built on the plugin system `pluggy`, which is also used in well known projects like `pytest` and `pytask`. Many core functionalities are also implemented as plugins. The PyRoL Core project only implements a minimal set of model approaches, look into the various official and unofficial plugins available for more.

Unlike the other mentioned projects, PyRoL has not only one plugin system, but several. Many main classes of PyRoL hold class attributes used to maintain plugins on that class, these are in detail:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plugin_manager</td>
<td>A <code>pluggy.PluginManager</code> instance used to maintain the plugins on this class.</td>
</tr>
<tr>
<td>hookspec</td>
<td>A wrapper around a <code>pluggy.HookspecMarker</code> instance for defining new hook specifications. Supports only a subset of the original arguments.</td>
</tr>
<tr>
<td>hookimpl</td>
<td>A wrapper around a <code>pluggy.HookimplMarker</code> instance for defining new hook implementations.</td>
</tr>
</tbody>
</table>

This is implemented using the `pyroll.plugin_host.PluginHost` class and the `pyroll.plugin_host.PluginHostMeta` metaclass.

```python
class PluginHostMeta(name, bases, dct):
    Metaclass that provides plugin functionality to a class.
    Not for direct uses but through `PluginHost` base class.

    hookimpl: HookimplMarker
        A wrapper around a `pluggy.HookimplMarker` instance for defining new hook implementations.

    hookspec: HookspecMarker
        A wrapper around a `pluggy.HookspecMarker` instance for defining new hook specifications. Supports only a subset of the original arguments.

    plugin_manager: PluginManager
        A `pluggy.PluginManager` instance used to maintain the plugins on this class.

    root_hooks: Set[str]
        Set of hooks to call in every solution iteration.

class PluginHost(hook_args: Dict[str, Any])
    A base class providing plugin functionality using the `PluginHostMeta` metaclass.

    The `get_from_hook()` method is also callable through the attribute syntax (period notation), where the key equals the attributes name.

    Parameters
        hook_args -- keyword arguments to pass to hook calls
```
```python
__getattr__(key: str)
Call a hook through attribute syntax if there is no explicit attribute with that name by use of get_from_hook().

delete_hook_result_attributes()
Deletes the attributes created by get_from_hook() calls, except those present in root_hooks.

get_from_hook(key: str)
Explicitly tries to get a value from a hook specified on this class. Returns and caches the result of the hook call as attribute. Use clear_hook_results() to clear the cache. Hook calls done by this function are not cleared, only those by attribute syntax.

If the plugin manager does not know a hook of name key, the function dispatches to eventual base classes.

Parameters

key (str) – the hook name to call

Raises

• AttributeError – if the hook call resulted in None
• AttributeError – if the hook name is not known to this class, nor to base classes
• ValueError – if the hook call resulted in an infinite value

get_root_hook_results()
Call necessary root hooks of this instance and return an array of their results.

hook_args
Keyword arguments to pass to hook calls.

hook_result_attributes: Set[str]
Set remembering all hooks that were called on this class, used by delete_hook_result_attributes().
```

The hookspec markers of all classes derived from Unit (RollPass and Transport) and Profile are preconfigured as firstresult. That means, that the first hook implementation, that returns not None is used as only result of the hook call. This offers the possibility of implementing many specialized versions of a hook and fall back to general ones if no special one applies.

Almost every attribute on the mentioned classes can be represented by a hook. This is achieved by overriding __getattr__, so that if no attribute with a desired name is present on an object, the framework searches for a hook of equal name. If there is no such hook, or the hook call results in None, an error is raised. Therefore, it is easy to specify new hooks, just use the hookspec marker on a dummy function and add it to the plugin_manager by use of plugin_manager.add_hookspecs(). It is common in writing plugins for PyRoLL to specify hooks for all intermediate and result values on profiles and units you want to calculate, and then to provide at least one general implementation of them. Afterwards you can proceed providing more specialized implementations in the same plugin package, or maybe also in another one if you need more flexibility in loading different implementations.

The classes Reporter and Exporter are also maintaining a plugin system, to allow plugins to contribute their own results to the output. But those hooks are not firstresult per default and specifying new hooks is not as easy as with units and profiles.

Details affecting only the distinct classes are described in their documentation.

For examples on specifying and implementing hooks, please read the pluggy documentation and look into the source code of PyRoLL.
The PyRoLL Core package is installable via PyPI

```
pip install pyroll
```

A collection of plugin packages can be installed the same way, the packages names usually start with pyroll-. Use the PyPI search or look at the projects GitHub page for discovering plugins.
The package provides a simple CLI tool that can be used to load input data, run the solution procedure and export the solution data. The CLI provides several commands that can and must be chained in one call. No state is preserved between different program runs.

The simplest use case is to read from a python script, solve and render the results to an HTML report page. The default input file is `input.py`, the default report file `report.html`.

```
pyroll input-py solve report
```

One may specify the files explicitly with the `-f/--file` option:

```
pyroll input-py -f other_input.py solve report -f other_report.html
```

A most basic input file may look like:

```python
from pyroll.core import Profile, RollPass, Transport, Roll, DiamondGroove, SquareGroove

in_profile = Profile.square(
    side=45e-3, corner_radius=3e-3,
    temperature=1200 + 273.15, flow_stress=100e6, strain=0,
)

sequence = [
    RollPass(
        label="Diamond I", velocity=1, gap=3e-3,
        roll=Roll(
            groove=DiamondGroove(
                usable_width=76.5e-3, tip_depth=22e-3, r1=12e-3, r2=8e-3
            ),
            nominal_radius=160e-3
        ),
    ),
    Transport(duration=2),
    RollPass(
        label="Square II", velocity=1, gap=3e-3,
        roll=Roll(
            groove=SquareGroove(
                usable_width=52.7e-3, tip_depth=26e-3, r1=8e-3, r2=6e-3
            ),
            nominal_radius=160e-3
        )
    )
]
```

(continues on next page)
The file must define the variables `in_profile` and `sequence` defining the state of the initial workpiece and the sequence of roll passes and transport ranges. For a more advanced example, representing a pass sequence at the 3-high mill at the Institute of Metals Forming, run:

```
pyroll create-input-py -k trio -f input.py
```

The PyRoIL command line interface resides additionally on a YAML configuration file `config.yaml`. The default file can be created using the following command:

```
pyroll create-config
```

The core section of this file is the `plugins` section. Here one can specify a list of plugins that will be loaded in each simulation run. Another way of loading plugins is to directly import them in the input Python script.

It is recommended to create a fresh directory for each simulation project to avoid the need to specify the filenames explicitly. A basic input and config file can be created in the current directory using

```
pyroll new
```
p
pyroll.core.grooves, 17
pyroll.core.profile.hookspe,cs, 25
pyroll.core.roll.hookspe,cs, 29
pyroll.core.roll_pass.hookspe,cs.out_profile, 29
pyroll.core.roll_pass.hookspe,cs.profile, 29
pyroll.core.roll_pass.hookspe,cs.roll, 29
pyroll.core.roll_pass.hookspe,cs.roll_pass, 28
pyroll.core.transport.hookspe,cs.out_profile, 30
pyroll.core.transport.hookspe,cs.transport, 30
pyroll.ui.exporter.hookspe,cs, 34
pyroll.ui.reporter.hookspe,cs, 31
Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__getattr__()</code></td>
<td>(PluginHost method)</td>
<td>35</td>
</tr>
<tr>
<td><code>box()</code></td>
<td>(Profile class method)</td>
<td>23</td>
</tr>
<tr>
<td><code>BoxGroove</code></td>
<td>(class in pyroll.core.grooves)</td>
<td>17</td>
</tr>
<tr>
<td><code>CircularOvalGroove</code></td>
<td>(class in pyroll.core.grooves)</td>
<td>18</td>
</tr>
<tr>
<td><code>columns()</code></td>
<td>(in module pyroll.ui.exporter.hookspecs)</td>
<td>34</td>
</tr>
<tr>
<td><code>ConstrictedBoxGroove</code></td>
<td>(class in pyroll.core.grooves)</td>
<td>18</td>
</tr>
<tr>
<td><code>ConstrictedSwedishOvalGroove</code></td>
<td>(class in pyroll.core.grooves)</td>
<td></td>
</tr>
<tr>
<td><code>contact_area()</code></td>
<td>(in module pyroll.core.roll_pass.hookspecs.roll)</td>
<td>29</td>
</tr>
<tr>
<td><code>contact_length()</code></td>
<td>(in module pyroll.core.roll_pass.hookspecs.roll)</td>
<td>29</td>
</tr>
<tr>
<td><code>contour_line</code></td>
<td>(GenericElongationGroove property)</td>
<td>20</td>
</tr>
<tr>
<td><code>contour_line</code></td>
<td>(GrooveBase property)</td>
<td>20</td>
</tr>
<tr>
<td><code>contour_line</code></td>
<td>(SplineGroove property)</td>
<td>21</td>
</tr>
<tr>
<td><code>contour_line()</code></td>
<td>(in module pyroll.core.hookspecs)</td>
<td>29</td>
</tr>
<tr>
<td><code>cross_section</code></td>
<td>(GenericElongationGroove property)</td>
<td>20</td>
</tr>
<tr>
<td><code>cross_section</code></td>
<td>(GrooveBase property)</td>
<td>20</td>
</tr>
<tr>
<td><code>cross_section</code></td>
<td>(SplineGroove property)</td>
<td>21</td>
</tr>
<tr>
<td><code>cross_section()</code></td>
<td>(in module pyroll.core.profile.hookspecs)</td>
<td>25</td>
</tr>
</tbody>
</table>

D

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>delete_hook_result_attributes()</code></td>
<td>(PluginHost method)</td>
<td>36</td>
</tr>
<tr>
<td><code>depth</code></td>
<td>(GenericElongationGroove property)</td>
<td>20</td>
</tr>
<tr>
<td><code>depth</code></td>
<td>(GrooveBase property)</td>
<td>20</td>
</tr>
<tr>
<td><code>depth</code></td>
<td>(SplineGroove property)</td>
<td>21</td>
</tr>
<tr>
<td><code>diamond()</code></td>
<td>(Profile class method)</td>
<td>23</td>
</tr>
<tr>
<td><code>DiamondGroove</code></td>
<td>(class in pyroll.core.grooves)</td>
<td>19</td>
</tr>
<tr>
<td><code>duration()</code></td>
<td>(in module pyroll.core.transport.hookspecs.transport)</td>
<td>30</td>
</tr>
</tbody>
</table>

E

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>equivalent_rectangle()</code></td>
<td>(in module pyroll.core.profile.hookspecs)</td>
<td>25</td>
</tr>
<tr>
<td><code>export()</code></td>
<td>(Exporter method)</td>
<td>34</td>
</tr>
<tr>
<td><code>export()</code></td>
<td>(in module pyroll.ui.exporter.hookspecs)</td>
<td>34</td>
</tr>
<tr>
<td><code>Exporter</code></td>
<td>(class in pyroll.ui.exporter)</td>
<td>34</td>
</tr>
</tbody>
</table>

F

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FalseRoundGroove</code></td>
<td>(class in pyroll.core.grooves)</td>
<td>19</td>
</tr>
<tr>
<td><code>filling_ratio()</code></td>
<td>(in module pyroll.core.roll_pass.hookspecs.out_profile)</td>
<td>29</td>
</tr>
<tr>
<td><code>FlatGroove</code></td>
<td>(class in pyroll.core.grooves)</td>
<td>19</td>
</tr>
<tr>
<td><code>FlatOvalGroove</code></td>
<td>(class in pyroll.core.grooves)</td>
<td>19</td>
</tr>
<tr>
<td><code>flow_stress()</code></td>
<td>(in module pyroll.core.profile.hookspecs)</td>
<td>25</td>
</tr>
<tr>
<td><code>flow_stress()</code></td>
<td>(in module pyroll.core.roll_pass.hookspecs.profile)</td>
<td>29</td>
</tr>
<tr>
<td><code>from_groove()</code></td>
<td>(Profile class method)</td>
<td>24</td>
</tr>
</tbody>
</table>

G

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gap()</code></td>
<td>(in module pyroll.core.roll_pass)</td>
<td>28</td>
</tr>
<tr>
<td><code>GenericElongationGroove</code></td>
<td>(class in pyroll.core.grooves)</td>
<td></td>
</tr>
<tr>
<td><code>get_dataframe()</code></td>
<td>(Exporter method)</td>
<td>34</td>
</tr>
<tr>
<td><code>get_from_hook()</code></td>
<td>(PluginHost method)</td>
<td>36</td>
</tr>
<tr>
<td><code>get_root_hook_results()</code></td>
<td>(PluginHost method)</td>
<td>36</td>
</tr>
<tr>
<td><code>groove()</code></td>
<td>(in module pyroll.core.roll.hookspecs)</td>
<td>29</td>
</tr>
<tr>
<td><code>GrooveBase</code></td>
<td>(class in pyroll.core.grooves)</td>
<td>20</td>
</tr>
</tbody>
</table>

H

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>height()</code></td>
<td>(in module pyroll.core.profile.hookspecs)</td>
<td>25</td>
</tr>
<tr>
<td><code>height()</code></td>
<td>(in module pyroll.core.roll_pass.hookspecs.roll)</td>
<td>28</td>
</tr>
<tr>
<td><code>hook_args</code></td>
<td>(PluginHost attribute)</td>
<td>36</td>
</tr>
<tr>
<td><code>hook_result_attributes</code></td>
<td>(PluginHost attribute)</td>
<td>36</td>
</tr>
<tr>
<td><code>hook_result_attributes</code></td>
<td>(Profile attribute)</td>
<td>24</td>
</tr>
<tr>
<td><code>hookimpl</code></td>
<td>(PluginHostMeta attribute)</td>
<td>35</td>
</tr>
<tr>
<td><code>hookspec</code></td>
<td>(PluginHostMeta attribute)</td>
<td>35</td>
</tr>
</tbody>
</table>
PyRoL, Release 1.0.0

<table>
<thead>
<tr>
<th>Module/Class/Function</th>
<th>Module/Class/Function</th>
<th>Module/Class/Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_profile_rotation()</td>
<td>pyroll.core.roll_pass.hookspecs.roll_pass</td>
<td></td>
</tr>
<tr>
<td>local_depth()</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>local_depth()</td>
<td>pyroll.core.roll.hookspecs</td>
<td></td>
</tr>
<tr>
<td>local_depth()</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>lower_contour_line()</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>material()</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>max_radius()</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>mean_flow_stress()</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>min_radius()</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>module</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>pyroll.core.grooves</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>pyroll.core.roll.hookspecs</td>
<td>pyroll.core.roll_pass.hookspecs</td>
<td></td>
</tr>
<tr>
<td>pyroll.core.roll_pass.hookspecs</td>
<td>pyroll.core.roll_pass.hookspecs</td>
<td></td>
</tr>
<tr>
<td>pyroll.core.roll_pass.hookspecs</td>
<td>pyroll.core.roll_pass.hookspecs</td>
<td></td>
</tr>
<tr>
<td>pyroll.core.roll_pass.hookspecs</td>
<td>pyroll.core.roll_pass.hookspecs</td>
<td></td>
</tr>
<tr>
<td>pyroll.core.roll_pass.hookspecs</td>
<td>pyroll.core.roll_pass.hookspecs</td>
<td></td>
</tr>
<tr>
<td>pyroll.core.roll_pass.hookspecs</td>
<td>pyroll.core.roll_pass.hookspecs</td>
<td></td>
</tr>
<tr>
<td>pyroll.core.transport.hookspecs</td>
<td>pyroll.core.transport.hookspecs</td>
<td></td>
</tr>
<tr>
<td>pyroll.core.transport.hookspecs</td>
<td>pyroll.core.transport.hookspecs</td>
<td></td>
</tr>
<tr>
<td>pyroll.core.transport.hookspecs</td>
<td>pyroll.core.transport.hookspecs</td>
<td></td>
</tr>
<tr>
<td>pyroll.core.transport.hookspecs</td>
<td>pyroll.core.transport.hookspecs</td>
<td></td>
</tr>
<tr>
<td>pyroll.core.transport.hookspecs</td>
<td>pyroll.core.transport.hookspecs</td>
<td></td>
</tr>
<tr>
<td>pyroll.ui.exporter.hookspecs</td>
<td>pyroll.ui.exporter.hookspecs</td>
<td></td>
</tr>
<tr>
<td>pyroll.ui.reporter.hookspecs</td>
<td>pyroll.ui.reporter.hookspecs</td>
<td></td>
</tr>
<tr>
<td>nominal_radius()</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>Oval3RadiiFlankedGroove</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>Oval3RadiiGroove</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>plugin_manager</td>
<td>pyroll.core.plugin_host</td>
<td></td>
</tr>
<tr>
<td>PluginHost</td>
<td>pyroll.core.plugin_host</td>
<td></td>
</tr>
<tr>
<td>PluginHostMeta</td>
<td>pyroll.core.plugin_host</td>
<td></td>
</tr>
<tr>
<td>profile</td>
<td>pyroll.core.profile.hookspecs</td>
<td></td>
</tr>
<tr>
<td>render()</td>
<td>Reporter</td>
<td></td>
</tr>
<tr>
<td>roll()</td>
<td>pyroll.core.roll_pass.hookspecs.roll_pass</td>
<td></td>
</tr>
<tr>
<td>roll_force()</td>
<td>pyroll.core.roll_pass.hookspecs.roll_pass</td>
<td></td>
</tr>
<tr>
<td>roll_torque()</td>
<td>pyroll.core.roll_pass.hookspecs.roll_pass</td>
<td></td>
</tr>
<tr>
<td>root_hooks</td>
<td>PluginHostMeta</td>
<td></td>
</tr>
<tr>
<td>rotational_frequency()</td>
<td>pyroll.core.roll.hookspecs</td>
<td></td>
</tr>
<tr>
<td>round()</td>
<td>Profile</td>
<td></td>
</tr>
<tr>
<td>RoundGroove</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>sequence_plot()</td>
<td>pyroll.ui.reporter.hookspecs</td>
<td></td>
</tr>
<tr>
<td>sequence_properties()</td>
<td>pyroll.ui.reporter.hookspecs</td>
<td></td>
</tr>
<tr>
<td>SplineGroove</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>spread()</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>square()</td>
<td>Profile</td>
<td></td>
</tr>
<tr>
<td>SquareGroove</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
<tr>
<td>strain()</td>
<td>pyroll.core.grooves</td>
<td></td>
</tr>
</tbody>
</table>
strain() (in module pyroll.core.roll_pass.hookspecs.out_profile), 29
strain() (in module pyroll.core.transport.hookspecs.out_profile), 30
strain_change() (in module pyroll.core.roll_pass.hookspecs.roll_pass), 28
strain_rate() (in module pyroll.core.roll_pass.hookspecs.roll_pass), 28
SwedishOvalGroove (class in pyroll.core.grooves), 22

T
temperature() (in module pyroll.core.profile.hookspecs), 25
tip_width() (in module pyroll.core.roll_pass.hookspecs.roll_pass), 28
types (BoxGroove property), 17
types (CircularOvalGroove property), 18
types (ConstrictedBoxGroove property), 18
types (ConstrictedSwedishOvalGroove property), 18
types (DiamondGroove property), 19
types (FalseRoundGroove property), 19
types (FlatGroove property), 19
types (FlatOvalGroove property), 19
types (GenericElongationGroove property), 20
types (GrooveBase property), 20
types (Oval3RadiiFlankedGroove property), 21
types (Oval3RadiiGroove property), 21
types (RoundGroove property), 21
types (SplineGroove property), 22
types (SquareGroove property), 22
types (SwedishOvalGroove property), 22
types() (in module pyroll.core.profile.hookspecs), 25

U
unit_plot() (in module pyroll.ui.reporter.hookspecs), 31
unit_properties() (in module pyroll.ui.reporter.hookspecs), 32
upper_contour_line() (in module pyroll.core.profile.hookspecs), 25
usable_width (GenericElongationGroove property), 20
usable_width (GrooveBase property), 20
usable_width (SplineGroove property), 22

V
velocity() (in module pyroll.core.roll_pass.hookspecs.roll_pass), 28

W
width() (in module pyroll.core.profile.hookspecs), 25
width() (in module pyroll.core.roll_pass.hookspecs.out_profile), 29
working_radius() (in module pyroll.core.roll.hookspecs), 29

Index 45