flexlib

Release 1.1.1

Jonas Walkling (MinkTec)

Aug 01, 2025

CONTENTS:

1 Coordinate Systems and Orientation **2**

2 Measurement **5**

3 SensorAngles **9**

4 Coordinates **11**

5 CaseOrientation **12**

6 FullFlexTailReconstructionPy **13**

7 Common Metrics and Analysis **14**

8 Distance Parameter Usage Examples **15**

9 flexlib **17**
9.1 flexlib package 17

10 Demo: Plotting **33**

11 Dishwasher Demo **39**
11.1 Schmitt Trigger for Bending Detection 41
11.2 Summary 43

flexlib is a Python library for processing and analyzing sensor data from FlexTail wearable sensors. The library provides tools for loading, processing, and evaluating biomechanical measurements.

Core Functionality:

- **Data Processing**: Load and process sensor measurements from various file formats (RSF v1/v2, CSV)
- **Biomechanical Analysis**: Calculate spine angles, posture metrics, and movement parameters
- **Signal Processing**: Apply filters for motion and bending detection.
- **Data Evaluation**: Extract metrics from measurement data for analysis

Key Components:

- `Measurement` objects representing individual sensor readings with timestamp, flex sensor data, accelerometer, and gyroscope values
- `AnnotatedRecording` for time-series data with event annotations
- Evaluation metrics for extracting biomechanical features (lumbar angles, twist, lateral/sagittal flexion)
- Signal processing utilities including Schmitt triggers for robust motion detection
- File I/O support for reading and writing sensor data in multiple formats

CHAPTER

ONE

COORDINATE SYSTEMS AND ORIENTATION

For anatomical reference, the following orientation planes are used:

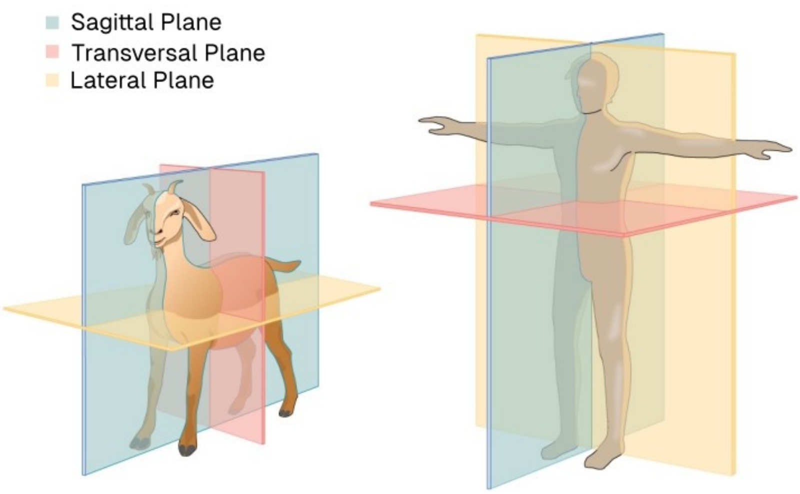


Figure 1.1: **Anatomical Planes:** Reference planes for understanding sensor orientation relative to human anatomy.

All references to sagittal or lateral are in reference to these planes.

The sensor comes in two variants: 36 cm and 45 cm. The amount of sensor pairs is the same which means that the distance between the measuring points is 20mm for the 36 cm variant and 25mm for the 45 cm variant. To handle different sensor variants, specify the appropriate `dist` parameter when creating a `Measurement` object:

- For 36cm sensors: `flexlib.Measurement(data, time, dist=20.0)` (default)
- For 45cm sensors: `flexlib.Measurement(data, time, dist=25.0)`

The distance parameter can also be overridden for individual calculations using *`coordinates_with_dist()`* or *`full_reconstruction_with_dist()`* methods, allowing you to compare results with different sensor spacings without recreating the measurement object.

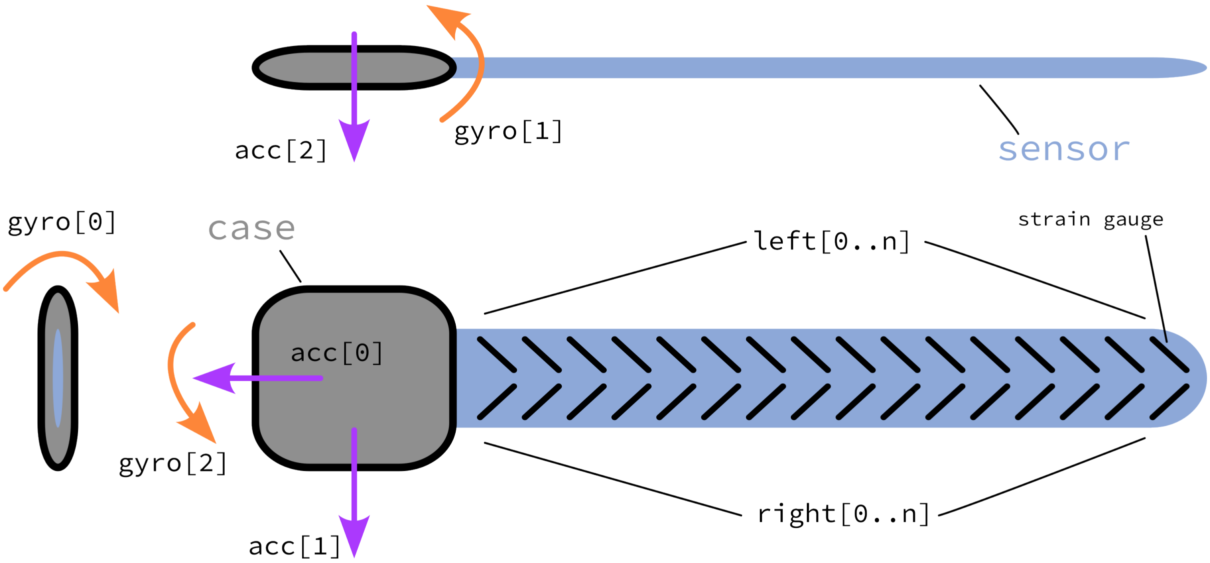


Figure 1.2: **Sensor Overview:** A drawing of the sensor strip and the case including directions of the accelerometer and gyroscope. The `acc`, `gyro`, `left` and `right` are lists of values that are used in the measurement class.

The main sensor data fields are:

- `left`: List of integer values from the left side of the sensor strip. Positive values indicate bending forward, negative values indicate bending backward.
- `right`: List of integer values from the right side of the sensor strip. Positive values indicate bending forward, negative values indicate bending backward.
- `gyro`: Gyroscope readings. This is a list of three integer values representing angular velocity (rotation rate) around the X, Y, and Z axes. Used to analyze rotational motion.
- `acc`: Accelerometer readings. This is a list of three integer values representing acceleration along the X, Y, and Z axes. A value of 2000 corresponds to 1g (gravitational acceleration). Used to determine orientation and movement.

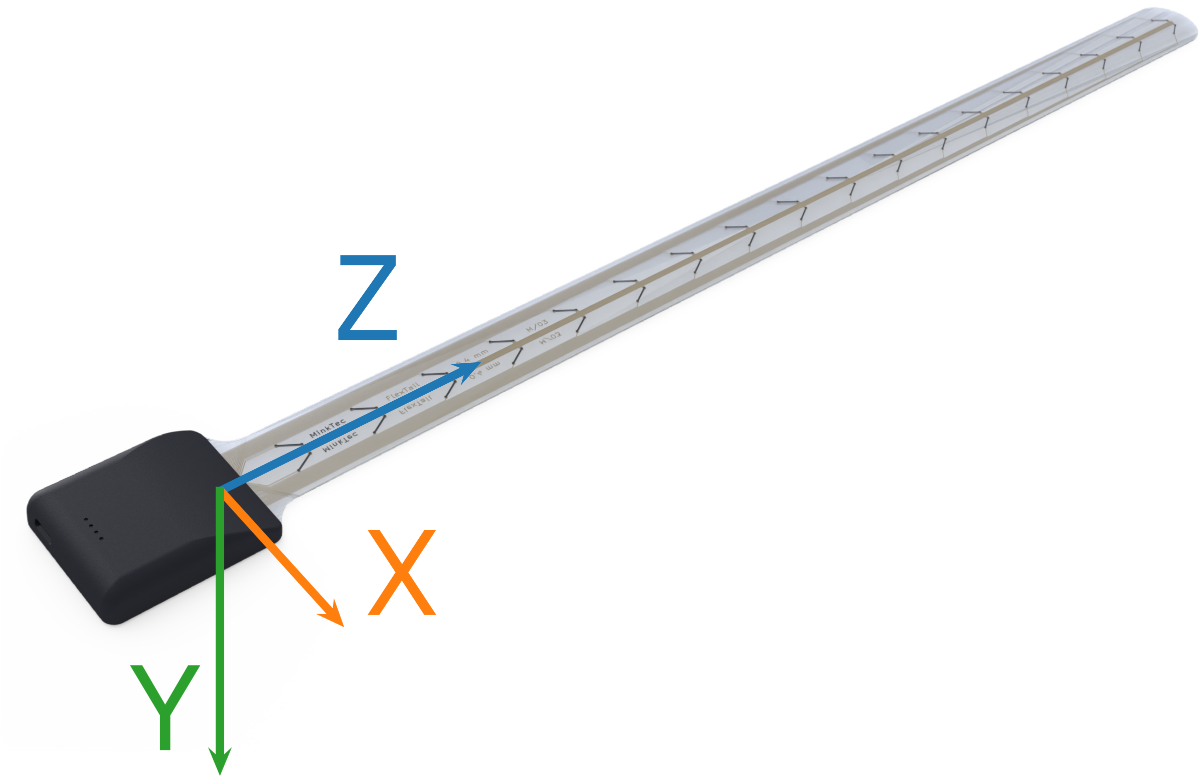


Figure 1.3: **Coordinate System:** All coordinates that are calculated by the sensor follow the coordinate system shown here and are in mm. The X axis is the right/left axis in the dorsal plane, the Y axis is the forward/backward axis in the midsagittal plane, and the Z axis is the vertical axis.

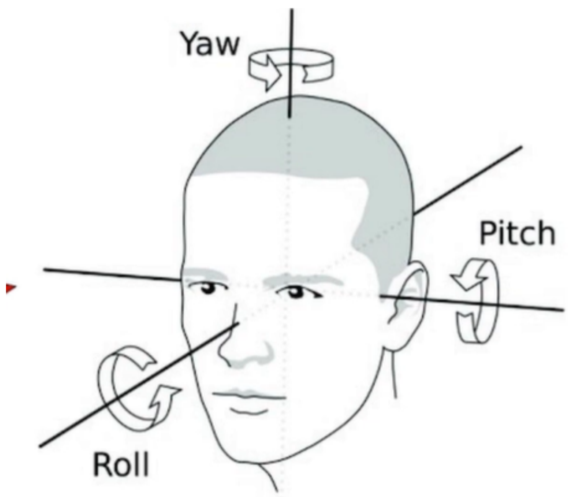


Figure 1.4: **Case Orientation:** We measure the case orientation by two angles: pitch and roll. Those angles are calculated from the accelerometer data and represent the tilt of the sensor case in space. Pitch is positive to the front, roll is positive to the right.

CHAPTER TWO

MEASUREMENT

class `flexlib.Measurement`(*data*, *time=None*, *dist=20.0*)

The Measurement class is the core representation of a FlexTail measurement. It encapsulates the raw sensor data (sensor strip, accelerometer and gyroscope), the number of sensors, and the timestamp of the measurement.

Parameters

- **data** (*list[int]*) – Raw sensor, accelerometer, and gyroscope data
- **time** (*int or None*) – Timestamp in ms since epoch
- **dist** (*float*) – Distance between sensor elements in mm (default: 20.0 for 36cm sensors, use 25.0 for 45cm sensors)

`values`

Raw sensor data values.

Type
list[int]

`timestamp`

Timestamp in milliseconds since epoch.

Type
int or None

`dist`

Distance between sensor elements in mm.

Type
float

`datetime`

Python datetime object from timestamp.

Type
datetime.datetime or None

`iso_time`

ISO formatted time string.

Type
str

`left`

The measured values from the left sensor side. Positive values indicate bending forward, negative values indicate bending backward.

Type
list[int]

`right`

The measured values from the right sensor side. Positive values indicate bending forward, negative values indicate bending backward.

Type
list[int]

`acc`

Accelerometer readings. Value of 2000 corresponds to 1g (gravitational acceleration).

Type
list[int]

`gyro`

Gyroscope readings (angular velocity around each axis).

Type
list[int]

`voltage`

Voltage reading.

Type
int

`angles`

Angles between adjacent measurement points along the sensor strip.

Type
SensorAngles

`coordinates`

3D coordinates of each measurement point along the sensor strip, calculated from the bending angles.

Type
Coordinates

`case_orientation`

Orientation of the sensor case, derived from accelerometer data.

Type
CaseOrientation

`number_of_sensors`

Number of sensors in the strip.

Type
int

`lateral_flexion`

Lateral flexion angle. Measures lateral (side-to-side) flexion. Useful for analyzing sideways bending.

Type
float

`sagittal_flexion`

Sagittal flexion angle. Measures sagittal (forward/backward) flexion. This is the angle between the case and the tip of the sensor. If person would lean to the right the angle would be positive. Useful for analyzing bending in the sagittal plane.

Type
float

coordinates_with_dist(*dist*)

3D coordinates of each measurement point along the sensor strip with custom distance parameter.

Parameters

dist (*float or None*) – Distance between sensor elements in mm. If None, uses the measurement's dist value.

Returns

3D positions in mm

Return type

Coordinates

full_reconstruction()

Get full 3D reconstruction of the sensor strip.

Returns

Full reconstruction data

Return type

FullFlexTailReconstructionPy

full_reconstruction_with_dist(*dist*)

Get full 3D reconstruction of the sensor strip with custom distance parameter.

Parameters

dist (*float or None*) – Distance between sensor elements in mm. If None, uses the measurement's dist value.

Returns

Full reconstruction data

Return type

FullFlexTailReconstructionPy

to_csv_row()

Convert measurement to CSV row format.

Returns

CSV formatted string

Return type

str

movement()

Calculate movement metric from gyroscope data.

Returns

Movement value

Return type

float

calc_sagittal_approx()

Calculates the sagittal approximation based on the angles. The approximation assumes that the sensor does not deform along the x axis.

Returns

Sagittal approximation

Return type

float

calc_lateral_approx()

Calculate lateral approximation.

Returns
Lateral approximation

Return type
float

CHAPTER

THREE

SENSORANGLES

class flexlib.**SensorAngles**(*left*, *right*)

Bending and rotational angles between measurement points.

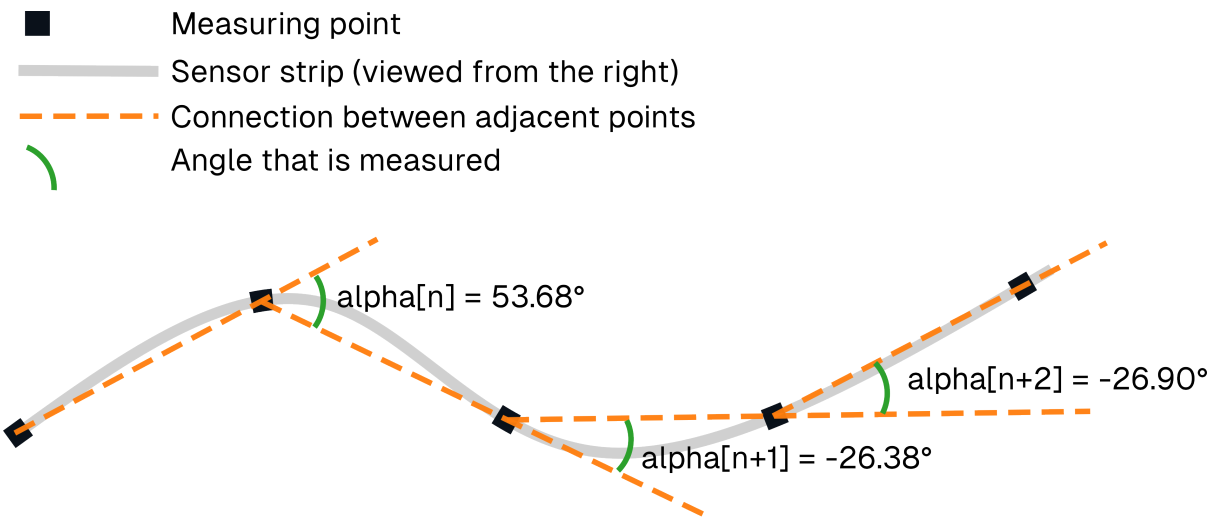


Figure 3.1: **Sensor Angles:** Visualization of alpha (bending) angles between adjacent points along the sensor strip. The black squares represent measurement points on the sensor. The numbers are the readouts produced by each specific measuring point. Each point measures the angle between the straight line connecting it (n) to the next point (n+1) and the connection between the next (n+1) and the point after that (n+2). The angles depicted here are the `alpha` values. The `beta` values represent the rotation or twist between adjacent measurement points.

Parameters

- **left** (*list[int]*) – Left sensor readings
- **right** (*list[int]*) – Right sensor readings

alpha

Bending angles (radians) between adjacent points (sagittal plane).

Type

list[float]

beta

Rotational angles (radians) between adjacent points (twisting).

Type

list[float]

`bend`

The overall bending (flexion/extension) of the lumbar region, calculated as the sum of the bottom 9 alpha values. Useful for posture and spinal movement analysis.

Type
float

`twist`

The total rotational movement (twisting) of the sensor strip, calculated as the sum of all beta angles. Useful for detecting torsional motions of the back.

Type
float

`static calc_angles`(*left*, *right*)

Calculate angles from left and right sensor data using default distance (20.0mm).

Parameters

- **left** (*list[int]*) – Left sensor readings
- **right** (*list[int]*) – Right sensor readings

Returns
Tuple of (alpha, beta) angles

Return type
tuple[list[float], list[float]]

`static calc_angles_with_dist`(*left*, *right*, *dist*)

Calculate angles from left and right sensor data with custom distance parameter.

Parameters

- **left** (*list[int]*) – Left sensor readings
- **right** (*list[int]*) – Right sensor readings
- **dist** (*float*) – Distance between sensor elements in mm

Returns
Tuple of (alpha, beta) angles

Return type
tuple[list[float], list[float]]

`static new_with_dist`(*left*, *right*, *dist*)

Create SensorAngles with custom distance between sensor elements.

Parameters

- **left** (*list[int]*) – Left sensor readings
- **right** (*list[int]*) – Right sensor readings
- **dist** (*float*) – Distance between sensor elements in mm

Returns
SensorAngles object with calculated angles

Return type
SensorAngles

CHAPTER

FOUR

COORDINATES

class flexlib.**Coordinates**(*x*, *y*, *z*)

3D positions of each measurement point along the sensor strip. The coordinate system is shown in Figure 1.3.

Parameters

- **x** (*list[float]*) – X-coordinates (right/left in dorsal plane, mm)
- **y** (*list[float]*) – Y-coordinates (forward/backward in midsagittal plane, mm)
- **z** (*list[float]*) – Z-coordinates (vertical, mm)

x

X-coordinates (right/left in dorsal plane, mm).

Type

list[float]

y

Y-coordinates (forward/backward in midsagittal plane, mm).

Type

list[float]

z

Z-coordinates (vertical, mm).

Type

list[float]

rotated(*pitch*, *roll*)

Returns a new Coordinates with all points rotated by the given pitch (x-axis) and roll (y-axis) angles (in radians).

Parameters

- **pitch** (*float*) – Rotation around x-axis (radians)
- **roll** (*float*) – Rotation around y-axis (radians)

Returns

Rotated coordinates

Return type

Coordinates

CHAPTER

FIVE

CASEORIENTATION

class flexlib.**CaseOrientation**(*pitch*, *roll*)

Orientation of the sensor case in space.

Parameters

- **pitch** (*float*) – Angle in sagittal plane (forward/backward tilt, radians)
- **roll** (*float*) – Angle in dorsal plane (side-to-side tilt, radians)

pitch

Angle in sagittal plane (forward/backward tilt, radians).

Type

float

roll

Angle in dorsal plane (side-to-side tilt, radians).

Type

float

static from_acc(*acc*)

Calculate case orientation from accelerometer data.

Parameters

acc (*list[int]*) – Accelerometer readings

Returns

Case orientation

Return type

CaseOrientation

CHAPTER

SIX

FULLFLEXTAILRECONSTRUCTIONPY

class `flexlib.FullFlexTailReconstructionPy`

3D reconstruction of the sensor strip, including all calculated coordinates and angles. The `left` and `right` coordinates represent the left / right outline of the sensor strip, when viwed from the top. The `center` coordinates represent the center of the sensor strip, and are equivalent to the `coords` attribute.

alpha

Bending angles along the midsagittal plane (forward/backward flexion).

Type
list[float]

beta

Rotational angles (twisting) along the sensor strip.

Type
list[float]

coords

3D coordinates of the sensor strip in space.

Type
Coordinates

case_position

Orientation of the sensor case (pitch, roll).

Type
CaseOrientation

left

3D coordinates for left points of the sensor tip.

Type
Coordinates

center

3D coordinates for center points of the sensor tip.

Type
Coordinates

right

3D coordinates for right points of the sensor tip.

Type
Coordinates

CHAPTER

SEVEN

COMMON METRICS AND ANALYSIS

The FlexTail sensor system provides several key metrics for biomechanical analysis:

Primary Measurements:

- **LUMBAR_ANGLE (bend)**: Measures the overall bending (flexion/extension) of the lumbar region. Calculated by summing the bottom 9 alpha values. Useful for posture and spinal movement analysis.
- **TWIST**: Measures rotational movement (twisting) of the sensor strip. Calculated by summing all beta angles. Useful for detecting torsional motions of the back.
- **LATERAL**: Measures lateral (side-to-side) flexion. Useful for analyzing sideways bending.
- **SAGITTAL**: Measures sagittal (forward/backward) flexion. Useful for analyzing bending in the sagittal plane.

Approximation Methods:

- **LATERAL_APPROX**: Provides an approximation of lateral flexion using a simplified calculation. Useful for quick or less precise assessments.
- **SAGITTAL_APPROX**: Provides an approximation of sagittal flexion using a simplified calculation. Useful for quick or less precise assessments.

Specialized Metrics:

- **THORACIC_ANGLE**: Focuses on the bending angle in the thoracic (upper back) region, typically using the second half of the sensor's angles. Useful for upper back posture analysis.
- **ACCELERATION**: Extracts the acceleration along a specific axis (usually the main axis of the sensor). Useful for detecting movement or orientation changes.
- **GYRO**: Measures angular velocity (rotation rate) from the gyroscope. Useful for analyzing dynamic movements and rotations.

CHAPTER

EIGHT

DISTANCE PARAMETER USAGE EXAMPLES

The FlexTail library supports different sensor variants with varying element spacing. Here are practical examples of using the distance parameter functionality:

Basic Usage with Different Sensor Types:

```
import flexlib

# Load measurement data
data = [100, 150, 200, ...]  # Your sensor data

# For 36cm sensors (20mm spacing) - default
measurement_36cm = flexlib.Measurement(data, dist=20.0)

# For 45cm sensors (25mm spacing)
measurement_45cm = flexlib.Measurement(data, dist=25.0)
```

Method-Level Distance Overrides:

```
# Create measurement with default 20mm spacing
measurement = flexlib.Measurement(data)

# Get coordinates with different spacing values
coords_default = measurement.coordinates      # Uses 20.0mm
coords_15mm = measurement.coordinates_with_dist(15.0)  # Override to 15mm
coords_25mm = measurement.coordinates_with_dist(25.0)  # Override to 25mm

# Get full reconstruction with custom spacing
recon_default = measurement.full_reconstruction           # Uses 20.0mm
recon_30mm = measurement.full_reconstruction_with_dist(30.0)  # Override to 30mm
```

Custom Angle Calculations:

```
# Calculate angles with specific sensor spacing
left_data = [100, 110, 120, ...]
right_data = [95, 105, 115, ...]

# Default 20mm spacing
angles_default = flexlib.SensorAngles(left_data, right_data)

# Custom 25mm spacing
angles_25mm = flexlib.SensorAngles.new_with_dist(left_data, right_data, 25.0)

# Raw angle calculation (returns tuple)
alpha, beta = flexlib.SensorAngles.calc_angles_with_dist(left_data, right_data, 30.0)
```

Comparing Different Sensor Configurations:

```python
# Compare coordinate scaling effects
measurement = flexlib.Measurement(data)

distances = [15.0, 20.0, 25.0, 30.0]
for dist in distances:
    coords = measurement.coordinates_with_dist(dist)
    last_point = coords.x[-1], coords.y[-1], coords.z[-1]
    print(f"Distance {dist}mm - Last point: {last_point}")
```

This approach ensures accurate physics-based calculations rather than post-processing scaling, providing more reliable biomechanical analysis results.

CHAPTER

NINE

FLEXLIB

9.1 flexlib package

class AbstractSchmittTrigger

Bases: `Generic[T]`

Base class for Schmitt triggers with hysteresis-based state detection. The Schmitt trigger classes implementations for robust signal state detection based on hysteresis. Schmitt triggers are fundamental components in signal processing and digital electronics, used to convert noisy or analog input signals into clean, discrete digital outputs. They achieve this by introducing hysteresis, which means the trigger has two different threshold levels: one for transitioning from LOW to HIGH (the upper threshold), and another for transitioning from HIGH to LOW (the lower threshold). This prevents rapid toggling or "chattering" of the output when the input signal is noisy or hovers near the threshold. This module provides several flexible and extensible Schmitt trigger implementations:

_state

The current state of the trigger (HIGH or LOW).

Type

SchmittState

add(*x*)

Add new input value and return new state if changed.

Parameters

x (*T*) – Input value to evaluate.

Returns

New state if changed, None if no state change.

Return type

SchmittState or None

property state: SchmittState

Current trigger state.

Returns

The current state (HIGH or LOW).

Return type

SchmittState

class AnnotatedRecording(*measurements*, *annotations=[]*)

Bases: `object`

A collection of sensor measurements with time-based annotations.

Contains a time-series of sensor measurements and associated annotations that mark specific events or states during the recording period.

property end_time

Get the timestamp of the last measurement.

static from_json_entry(*json*, *dist=20.0*)

Create an AnnotatedRecording from a JSON representation.

Parameters

- **json** – Dictionary with 'annotations' and 'content' keys
- **dist** (*float*) – Distance between sensor elements in mm (default: 20.0)

Returns

AnnotatedRecording instance

Return type

AnnotatedRecording

split_by_annotations()

Split the recording into segments based on annotations.

Returns

List of tuples containing (annotation_label, measurements_in_segment)

Return type

List[*Tuple*[str, *List*[*Measurement*]]]

property start_time

Get the timestamp of the first measurement.

to_json_entry()

Convert the annotated recording to a JSON-serializable dictionary.

Returns

Dictionary with 'annotations' and 'content' keys

Return type

dict

class CSVReader

Bases: `object`

Parse sensor measurement data from CSV files or strings.

static parse(*path*, *dist=20.0*)

Parse CSV content from a file.

Parameters

- **path** (*str*) – Path to the CSV file to parse.
- **dist** (*float*) – Distance between sensor elements in mm (default: 20.0)

Returns

Parsed measurements in an annotated recording.

Return type

AnnotatedRecording

static parse_string(*content*, *dist=20.0*)

Parse CSV content from a string.

Parameters

- **content** (*str*) – The CSV content as a string.
- **dist** (*float*) – Distance between sensor elements in mm (default: 20.0)

Returns
Parsed measurements in an annotated recording.

Return type
AnnotatedRecording

`class CSVWriter`

Bases: `object`

A writer for exporting measurement data to CSV format.

Writes sensor values followed by timestamp in comma-separated format.

`static write`(*file_path*, *recording*)

Write an annotated recording to a CSV file.

Parameters

- **`file_path`** (`str`) – Path where the CSV file should be written.
- **`recording`** (`AnnotatedRecording`) – The recording data to export.

`class CountingPredicateSchmittTrigger`(*high_predicate*, *low_predicate*, *low_count=0*, *high_count=0*)

Bases: `PredicateSchmittTrigger[T]`

`add`(*x*)

Add new input value and return new state if changed.

Parameters
x (*T*) – Input value to evaluate.

Returns
New state if changed, None if no state change.

Return type
SchmittState or None

`copy_with`(*high_predicate=None*, *low_predicate=None*, *low_count=None*, *high_count=None*)

`class CountingSchmittTrigger`(*high_bound*, *low_bound*, *invert=False*, *low_count=0*, *high_count=0*)

Bases: `SchmittTrigger[T]`

`add`(*x*)

Add new input value and return new state if changed.

Parameters
x (*T*) – Input value to evaluate.

Returns
New state if changed, None if no state change.

Return type
SchmittState or None

`copy_with`(*high_bound=None*, *low_bound=None*, *low_count=None*, *high_count=None*, *invert=None*)

`class DelayedPredicateSchmittTrigger`(*high_predicate*, *low_predicate*, *low_duration=datetime.timedelta(0)*, *high_duration=datetime.timedelta(0)*)

Bases: `PredicateSchmittTrigger[T]`

`add(x)`

Add new input value and return new state if changed.

Parameters
x (*T*) – Input value to evaluate.

Returns
New state if changed, None if no state change.

Return type
SchmittState or None

`copy_with(high_predicate=None, low_predicate=None, low_duration=None, high_duration=None)`

`class DelayedSchmittTrigger(high_bound, low_bound, low_duration=datetime.timedelta(0), high_duration=datetime.timedelta(0), invert=False)`

Bases: `SchmittTrigger[TimedComparable]`

`add(x)`

Add new input value and return new state if changed.

Parameters
x (*T*) – Input value to evaluate.

Returns
New state if changed, None if no state change.

Return type
SchmittState or None

`class FlexReader`

Bases: `object`

Universal reader that auto-detects and parses different sensor file formats.

`static parse(path, dist=20.0)`

Parse a sensor data file, auto-detecting the format.

Supports RSF v1, RSF v2, and CSV formats. The format is detected by examining the file header magic bytes.

Parameters

- **path** (`str`) – Path to the sensor data file
- **dist** (`float`) – Distance between sensor elements in mm (default: 20.0)

Returns
AnnotatedRecording containing the parsed measurements

Return type
AnnotatedRecording

`class MeasurementEvaluationMetric(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)`

Bases: `Enum`

Enum providing standardized metrics for analyzing sensor measurements.

Each metric extracts specific features from Measurement objects, such as posture angles, movement patterns, or sensor readings.

Metrics:

- LUMBAR_ANGLE: Measures the overall bending (flexion/extension) of the lumbar region. Calculated by summing the bottom 9 alpha values. Useful for posture and spinal movement analysis.
- TWIST: Measures rotational movement (twisting) of the sensor strip. Calculated by summing all beta angles. Useful for detecting torsional motions of the back.
- LATERAL: Measures lateral (side-to-side) flexion. Useful for analyzing sideways bending.
- SAGITTAL: Measures sagittal (forward/backward) flexion. Useful for analyzing bending in the sagittal plane.
- LATERAL_APPROX: Provides an approximation of lateral flexion using a simplified calculation. Useful for quick or less precise assessments.
- SAGITTAL_APPROX: Provides an approximation of sagittal flexion using a simplified calculation. Useful for quick or less precise assessments.
- THORACIC_ANGLE: Focuses on the bending angle in the thoracic (upper back) region, typically using the second half of the sensor's angles. Useful for upper back posture analysis.
- ACCELERATION: Extracts the acceleration along a specific axis (usually the main axis of the sensor). Useful for detecting movement or orientation changes.
- GYRO: Measures angular velocity (rotation rate) from the gyroscope. Useful for analyzing dynamic movements and rotations.

ACCELERATION = 'acceleration'

Acceleration along main axis

GYRO = 'gyro'

Angular velocity (gyroscope)

LATERAL = 'lateral'

Lateral (side-to-side) flexion

LATERAL_APPROX = 'lateralApprox'

Approximate lateral flexion

LUMBAR_ANGLE = 'lumbarAngle'

Overall lumbar flexion/extension (sum of bottom 9 alpha values)

SAGITTAL = 'sagittal'

Sagittal (forward/backward) flexion

SAGITTAL_APPROX = 'sagittalApprox'

Approximate sagittal flexion

THORACIC_ANGLE = 'thoracicAngle'

Thoracic (upper back) flexion (sum of second half of alpha)

TWIST = 'twist'

Rotational movement (sum of all beta angles)

exec(*measurements*)

Apply this metric to a collection of measurements.

Parameters

measurements (*list[Measurement] | AnnotatedRecording*) – List of measurements or annotated recording

Returns

List of metric values, one per measurement `func(m)`

Extract the metric value from a single measurement.

Parameters

m (`Measurement`) – The measurement to evaluate

Returns

The computed metric value (angle, acceleration, etc.)

`property is_angle: bool`

`class MeasurementMetric(first, second)`

Bases: `object`

Calculates comparison metrics between two sensor measurements.

Provides various metrics to quantify differences in posture, movement, and orientation between two measurement instances.

`property alpha_metric`

Difference in alpha angles (bending forward and backward) between measurements.

`property backward_metric`

Difference in beeing bend backward

`property beta_metric`

Difference in beta angles (rotation of the spine) between measurements.

`property forward_metric`

Difference in beeing bend forward

`property lateral_metric`

Difference between the angle of the tip compared to the case

`property metric`

Overall difference betweent two measurements

`property orientation_metric`

Difference in case orientation

`property sagittal_metric`

Difference of the angle of the sensor tip in relation to the case in the sagittal plane.

`class PredicateSchmittTrigger(high_predicate, low_predicate)`

Bases: `AbstractSchmittTrigger[T]`

Schmitt trigger using predicates for state transitions.

Parameters

- **high_predicate** (*Callable[[T], bool]*) – Predicate to trigger transition from LOW to HIGH.
- **low_predicate** (*Callable[[T], bool]*) – Predicate to trigger transition from HIGH to LOW.

`class RSFV1Reader`

Bases: `object`

Parse RSF v1 binary sensor data files into AnnotatedRecording objects.

`static parse(file_path, dist=20.0)`

class RSFV1Writer

Bases: `object`

static write(*file_path*, *recording*)

class RSFV2Reader

Bases: `object`

static parse(*file_path*, *dist=20.0*)

class RSFV2Writer

Bases: `object`

static write(*file_path*, *recording*)

class SchmittState(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `Enum`

States for Schmitt trigger: HIGH or LOW. High indicates that filter condition is currently true.

HIGH = 'high'

LOW = 'low'

negate()

Return the opposite state.

Returns

The opposite state (HIGH <-> LOW).

Return type

SchmittState

class SchmittTrigger(*high_bound*, *low_bound*, *invert=False*)

Bases: `AbstractSchmittTrigger[T]`

copy_with(*high_bound=None*, *low_bound=None*, *invert=None*)

class TimedAnnotation(*time*, *label*)

Bases: `object`

A time-stamped annotation label for marking events in sensor recordings.

static from_json_entry(*json*)

Create a TimedAnnotation from a JSON dictionary.

Parameters

json – Dictionary with ‘time’ and ‘value’ keys

Returns

TimedAnnotation instance

Return type

TimedAnnotation

to_json_entry()

Convert the annotation to a JSON-serializable dictionary.

Returns
Dictionary with 'time' and 'value' keys

Return type
dict

class TimedComparable(*value*, *time*)

Bases: `Generic[T]`

create_dataframe(*data*)

Create a pandas DataFrame from a list of Measurement objects or an AnnotatedRecording.

Parameters
data (*list[Measurement] or AnnotatedRecording*) – The sensor data to convert. If an AnnotatedRecording, annotations will be included.

Returns
DataFrame with time, measurement metrics, and optional annotation column.

Return type
pd.DataFrame

9.1.1 Subpackages

flexlib.models package

Subpackages

flexlib.models.readers package

Submodules

flexlib.models.readers.csv_parser module

class CSVReader

Bases: `object`

Parse sensor measurement data from CSV files or strings.

static parse(*path*, *dist=20.0*)

Parse CSV content from a file.

Parameters

- **path** (*str*) – Path to the CSV file to parse.
- **dist** (*float*) – Distance between sensor elements in mm (default: 20.0)

Returns
Parsed measurements in an annotated recording.

Return type
AnnotatedRecording

static parse_string(*content*, *dist=20.0*)

Parse CSV content from a string.

Parameters

- **content** (*str*) – The CSV content as a string.
- **dist** (*float*) – Distance between sensor elements in mm (default: 20.0)

Returns
Parsed measurements in an annotated recording.

Return type
AnnotatedRecording

flexlib.models.readers.rsf_v1 module

class RSFV1Reader
Bases: `object`

Parse RSF v1 binary sensor data files into AnnotatedRecording objects.

static parse(*file_path*, *dist=20.0*)

flexlib.models.readers.rsf_v2 module

class RSFV2Reader
Bases: `object`

static parse(*file_path*, *dist=20.0*)

flexlib.models.writers package

Submodules

flexlib.models.writers.csv_writer module

class CSVWriter
Bases: `object`

A writer for exporting measurement data to CSV format.

Writes sensor values followed by timestamp in comma-separated format.

static write(*file_path*, *recording*)
Write an annotated recording to a CSV file.

Parameters

- **file_path** (*str*) – Path where the CSV file should be written.
- **recording** (`AnnotatedRecording`) – The recording data to export.

flexlib.models.writers.rsf_v1_writer module

class RSFV1Writer
Bases: `object`

static write(*file_path*, *recording*)

flexlib.models.writers.rsf_v2_writer module

class RSFV2Writer
Bases: `object`

static write(*file_path*, *recording*)

Submodules

flexlib.models.annotated_recording module

class AnnotatedRecording(*measurements*, *annotations=[]*)

Bases: `object`

A collection of sensor measurements with time-based annotations.

Contains a time-series of sensor measurements and associated annotations that mark specific events or states during the recording period.

property end_time

Get the timestamp of the last measurement.

static from_json_entry(*json*, *dist=20.0*)

Create an AnnotatedRecording from a JSON representation.

Parameters

- **json** – Dictionary with 'annotations' and 'content' keys
- **dist** (*float*) – Distance between sensor elements in mm (default: 20.0)

Returns

AnnotatedRecording instance

Return type

AnnotatedRecording

split_by_annotations()

Split the recording into segments based on annotations.

Returns

List of tuples containing (annotation_label, measurements_in_segment)

Return type

List[*Tuple*[str, *List*[*Measurement*]]]

property start_time

Get the timestamp of the first measurement.

to_json_entry()

Convert the annotated recording to a JSON-serializable dictionary.

Returns

Dictionary with 'annotations' and 'content' keys

Return type

dict

class TimedAnnotation(*time*, *label*)

Bases: `object`

A time-stamped annotation label for marking events in sensor recordings.

static from_json_entry(*json*)

Create a TimedAnnotation from a JSON dictionary.

Parameters

json – Dictionary with 'time' and 'value' keys

Returns

TimedAnnotation instance

Return type
TimedAnnotation

`to_json_entry()`

Convert the annotation to a JSON-serializable dictionary.

Returns
Dictionary with 'time' and 'value' keys

Return type
dict

flexlib.models.flex_reader module

flex_reader: Universal file reader for various sensor data formats.

Supports auto-detection and parsing of RSF v1, RSF v2, and CSV sensor files.

`class FlexReader`

Bases: `object`

Universal reader that auto-detects and parses different sensor file formats.

`static parse`(*path*, *dist=20.0*)

Parse a sensor data file, auto-detecting the format.

Supports RSF v1, RSF v2, and CSV formats. The format is detected by examining the file header magic bytes.

Parameters

- **path** (*str*) – Path to the sensor data file
- **dist** (*float*) – Distance between sensor elements in mm (default: 20.0)

Returns
AnnotatedRecording containing the parsed measurements

Return type
AnnotatedRecording

flexlib.models.measurement_evaluation_metric module

measurement_evaluation_metric: Metrics for extracting features from sensor data.

Defines the MeasurementEvaluationMetric enum for standardized feature extraction from sensor measurements.

`class MeasurementEvaluationMetric`(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `Enum`

Enum providing standardized metrics for analyzing sensor measurements.

Each metric extracts specific features from Measurement objects, such as posture angles, movement patterns, or sensor readings.

Metrics:

- LUMBAR_ANGLE: Measures the overall bending (flexion/extension) of the lumbar region. Calculated by summing the bottom 9 alpha values. Useful for posture and spinal movement analysis.
- TWIST: Measures rotational movement (twisting) of the sensor strip. Calculated by summing all beta angles. Useful for detecting torsional motions of the back.
- LATERAL: Measures lateral (side-to-side) flexion. Useful for analyzing sideways bending.
- SAGITTAL: Measures sagittal (forward/backward) flexion. Useful for analyzing bending in the sagittal plane.

- LATERAL_APPROX: Provides an approximation of lateral flexion using a simplified calculation. Useful for quick or less precise assessments.
- SAGITTAL_APPROX: Provides an approximation of sagittal flexion using a simplified calculation. Useful for quick or less precise assessments.
- THORACIC_ANGLE: Focuses on the bending angle in the thoracic (upper back) region, typically using the second half of the sensor's angles. Useful for upper back posture analysis.
- ACCELERATION: Extracts the acceleration along a specific axis (usually the main axis of the sensor). Useful for detecting movement or orientation changes.
- GYRO: Measures angular velocity (rotation rate) from the gyroscope. Useful for analyzing dynamic movements and rotations.

`ACCELERATION = 'acceleration'`

Acceleration along main axis

`GYRO = 'gyro'`

Angular velocity (gyroscope)

`LATERAL = 'lateral'`

Lateral (side-to-side) flexion

`LATERAL_APPROX = 'lateralApprox'`

Approximate lateral flexion

`LUMBAR_ANGLE = 'lumbarAngle'`

Overall lumbar flexion/extension (sum of bottom 9 alpha values)

`SAGITTAL = 'sagittal'`

Sagittal (forward/backward) flexion

`SAGITTAL_APPROX = 'sagittalApprox'`

Approximate sagittal flexion

`THORACIC_ANGLE = 'thoracicAngle'`

Thoracic (upper back) flexion (sum of second half of alpha)

`TWIST = 'twist'`

Rotational movement (sum of all beta angles)

`exec`(*measurements*)

Apply this metric to a collection of measurements.

Parameters
measurements (*list[Measurement] | AnnotatedRecording*) – List of measurements or annotated recording

Returns
List of metric values, one per measurement

`func`(*m*)

Extract the metric value from a single measurement.

Parameters
m (*Measurement*) – The measurement to evaluate

Returns
The computed metric value (angle, acceleration, etc.)

`property is_angle: bool`

flexlib.models.measurement_metric module

measurement_metric: Metrics for comparing two sensor measurements.

Provides the MeasurementMetric class for quantifying differences in posture, movement, and orientation. This returns a metric in the mathematical sense and thus can be used for clustering.

class MeasurementMetric(*first*, *second*)

Bases: `object`

Calculates comparison metrics between two sensor measurements.

Provides various metrics to quantify differences in posture, movement, and orientation between two measurement instances.

property alpha_metric

Difference in alpha angles (bending forward and backward) between measurements.

property backward_metric

Difference in beeing bend backward

property beta_metric

Difference in beta angles (rotation of the spine) between measurements.

property forward_metric

Difference in beeing bend forward

property lateral_metric

Difference between the angle of the tip compared to the case

property metric

Overall difference betweent two measurements

property orientation_metric

Difference in case orientation

property sagittal_metric

Difference of the angle of the sensor tip in relation to the case in the sagittal plane.

flexlib.models.schmitt_trigger module

schmitt_trigger: Classes for robust edge detection, debouncing, and filtering using Schmitt triggers.

Includes: - AbstractSchmittTrigger: Base class for Schmitt triggers - PredicateSchmittTrigger: Predicate-based triggers - CountingPredicateSchmittTrigger: Adds count-based noise immunity - DelayedPredicateSchmittTrigger: Adds time-based filtering - SchmittTrigger: Classic numeric Schmitt trigger - CountingSchmittTrigger: Numeric + count-based filter - DelayedSchmittTrigger: Numeric + time-based filter

These classes are useful for signal processing, user input handling, and control systems.

class AbstractSchmittTrigger

Bases: `Generic[T]`

Base class for Schmitt triggers with hysteresis-based state detection. The Schmitt trigger classes implementations for robust signal state detection based on hysteresis. Schmitt triggers are fundamental components in signal processing and digital electronics, used to convert noisy or analog input signals into clean, discrete digital outputs. They achieve this by introducing hysteresis, which means the trigger has two different threshold levels: one for transitioning from LOW to HIGH (the upper threshold), and another for transitioning from HIGH to LOW (the lower threshold). This prevents rapid toggling or “chattering” of the output when the input signal is noisy or hovers near the threshold. This module provides several flexible and extensible Schmitt trigger implementations:

`_state`

The current state of the trigger (HIGH or LOW).

Type
SchmittState

`add`(*x*)

Add new input value and return new state if changed.

Parameters
x (*T*) – Input value to evaluate.

Returns
New state if changed, None if no state change.

Return type
SchmittState or None

`property state: SchmittState`

Current trigger state.

Returns
The current state (HIGH or LOW).

Return type
SchmittState

`class CountingPredicateSchmittTrigger`(*high_predicate*, *low_predicate*, *low_count=0*, *high_count=0*)

Bases: `PredicateSchmittTrigger[T]`

`add`(*x*)

Add new input value and return new state if changed.

Parameters
x (*T*) – Input value to evaluate.

Returns
New state if changed, None if no state change.

Return type
SchmittState or None

`copy_with`(*high_predicate=None*, *low_predicate=None*, *low_count=None*, *high_count=None*)

`class CountingSchmittTrigger`(*high_bound*, *low_bound*, *invert=False*, *low_count=0*, *high_count=0*)

Bases: `SchmittTrigger[T]`

`add`(*x*)

Add new input value and return new state if changed.

Parameters
x (*T*) – Input value to evaluate.

Returns
New state if changed, None if no state change.

Return type
SchmittState or None

`copy_with`(*high_bound=None*, *low_bound=None*, *low_count=None*, *high_count=None*, *invert=None*)

class DelayedPredicateSchmittTrigger(*high_predicate*, *low_predicate*, *low_duration=datetime.timedelta(0)*, *high_duration=datetime.timedelta(0)*)

Bases: `PredicateSchmittTrigger[T]`

add(*x*)

Add new input value and return new state if changed.

Parameters
x (*T*) – Input value to evaluate.

Returns
New state if changed, None if no state change.

Return type
SchmittState or None

copy_with(*high_predicate=None*, *low_predicate=None*, *low_duration=None*, *high_duration=None*)

class DelayedSchmittTrigger(*high_bound*, *low_bound*, *low_duration=datetime.timedelta(0)*, *high_duration=datetime.timedelta(0)*, *invert=False*)

Bases: `SchmittTrigger[TimedComparable]`

add(*x*)

Add new input value and return new state if changed.

Parameters
x (*T*) – Input value to evaluate.

Returns
New state if changed, None if no state change.

Return type
SchmittState or None

class HasDateTime(**args*, ***kwargs*)

Bases: `Protocol`

datetime: datetime

class PredicateSchmittTrigger(*high_predicate*, *low_predicate*)

Bases: `AbstractSchmittTrigger[T]`

Schmitt trigger using predicates for state transitions.

Parameters

- **high_predicate** (*Callable[[T], bool]*) – Predicate to trigger transition from LOW to HIGH.
- **low_predicate** (*Callable[[T], bool]*) – Predicate to trigger transition from HIGH to LOW.

class SchmittState(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `Enum`

States for Schmitt trigger: HIGH or LOW. High indicates that filter condition is currently true.

HIGH = 'high'

`LOW = 'low'`

`negate()`

Return the opposite state.

Returns
The opposite state (HIGH <-> LOW).

Return type
SchmittState

`class SchmittTrigger(`*high_bound*, *low_bound*, *invert=False*`)`

Bases: `AbstractSchmittTrigger[T]`

`copy_with(`*high_bound=None*, *low_bound=None*, *invert=None*`)`

`class TimedComparable(`*value*, *time*`)`

Bases: `Generic[T]`

`get_schmitt_episodes(`*trigger*`)`

9.1.2 Submodules

9.1.3 flexlib.dataframe_builder module

dataframe_builder: Utilities for converting sensor data to pandas DataFrames for analysis.

`create_dataframe(`*data*`)`

Create a pandas DataFrame from a list of Measurement objects or an AnnotatedRecording.

Parameters
data (*list[Measurement] or AnnotatedRecording*) – The sensor data to convert. If an AnnotatedRecording, annotations will be included.

Returns
DataFrame with time, measurement metrics, and optional annotation column.

Return type
pd.DataFrame

CHAPTER

TEN

DEMO: PLOTTING

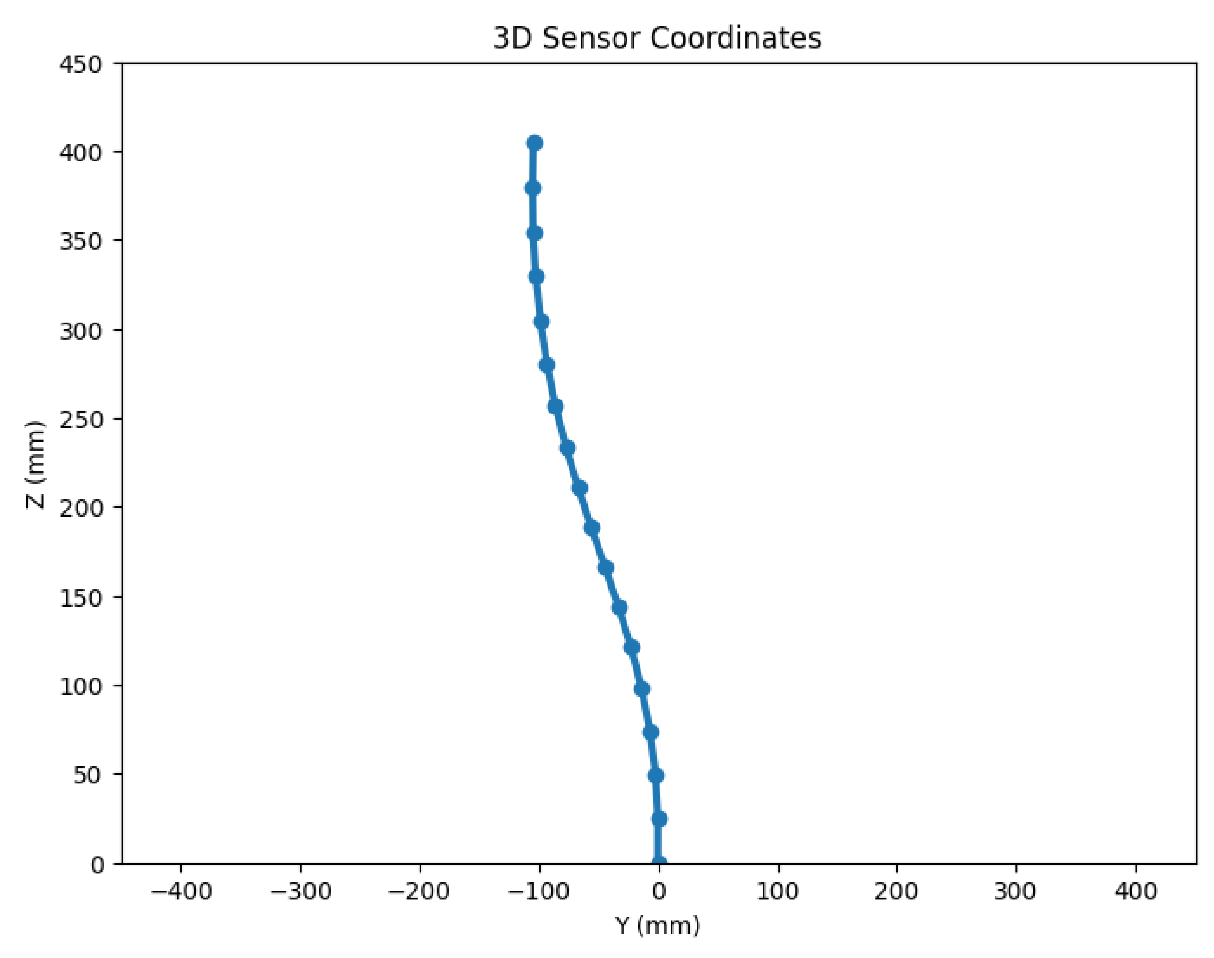
This notebook contains a few simple plots that can be created with flexlib

[2]:
```
from flexlib import Measurement, FlexReader
dist = 25
data = FlexReader().parse("../docs/test_data/tidy_up.rsf", dist = dist)
m = data[0]
```

This create a simple 2D of the sensor viewed from the right.

[3]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

max_val = 18 * dist

# Extract coordinates
# Plot 3D coordinates
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111)

ax.plot(m.coordinates.y, m.coordinates.z, marker='o')

for line in ax.get_lines():
    line.set_linewidth(3)
ax.set_xlabel('Y (mm)')
ax.set_ylabel('Z (mm)')
ax.set_title('3D Sensor Coordinates')
ax.set_xlim(-max_val, max_val)
ax.set_ylim(0, max_val)
```

[3]:
```
(0.0, 450.0)
```



[4]:
```
def equally_spaced_sample(lst, n):
    if n >= len(lst):
        return lst
    idxs = [round(i * (len(lst) - 1) / (n - 1)) for i in range(n)]
    return [lst[i] for i in idxs]
```

This as a kind of 2D-Heatmap with 500 sensor positions overlayed on each other.

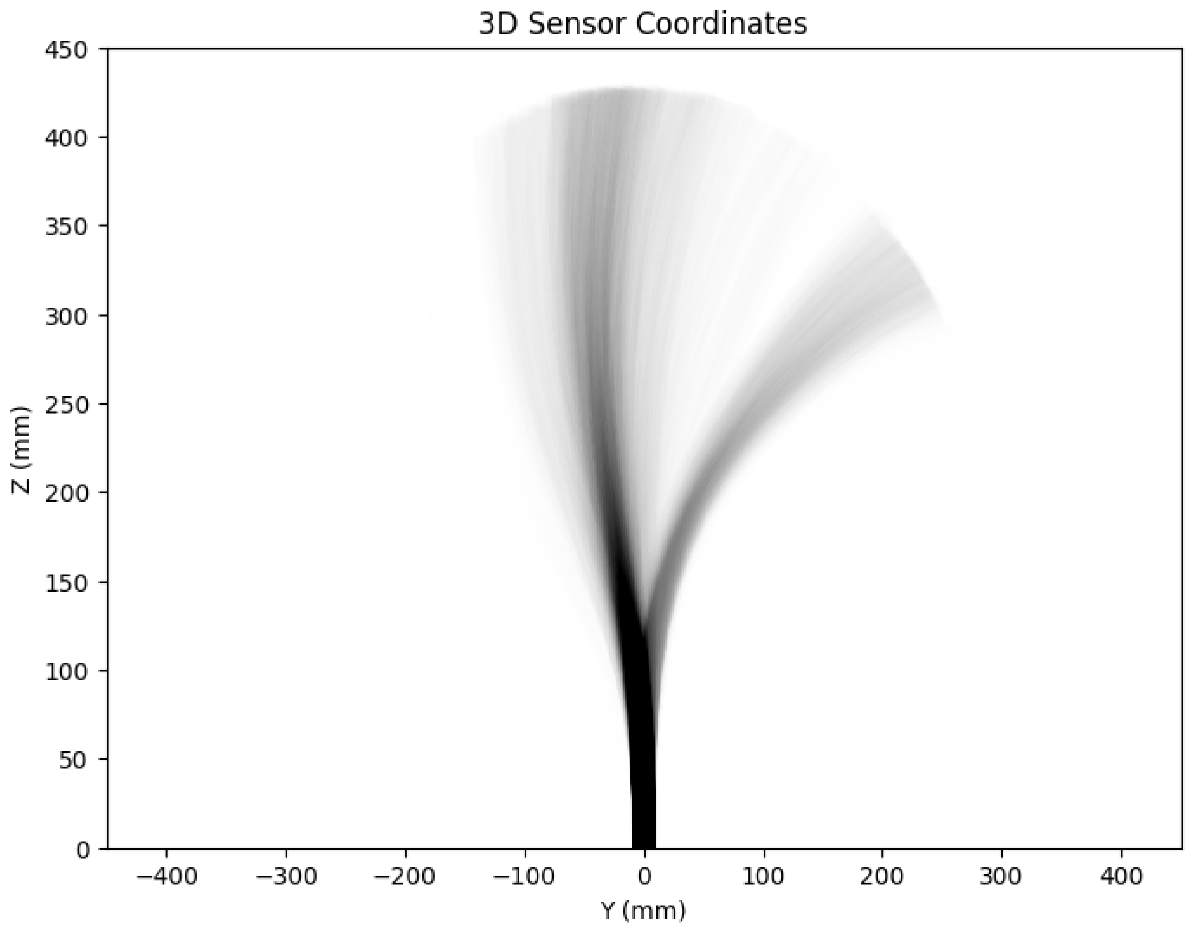
[5]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

max_val = 18 * dist

# Extract coordinates
# Plot 3D coordinates
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111)

for m in equally_spaced_sample(data.measurements, 500):
    ax.plot(m.coordinates.y, m.coordinates.z, color="#00000001")

for line in ax.get_lines():
    line.set_linewidth(10)
ax.set_xlabel('Y (mm)')
ax.set_ylabel('Z (mm)')
ax.set_title('3D Sensor Coordinates')
```

(continues on next page)

(continued from previous page)

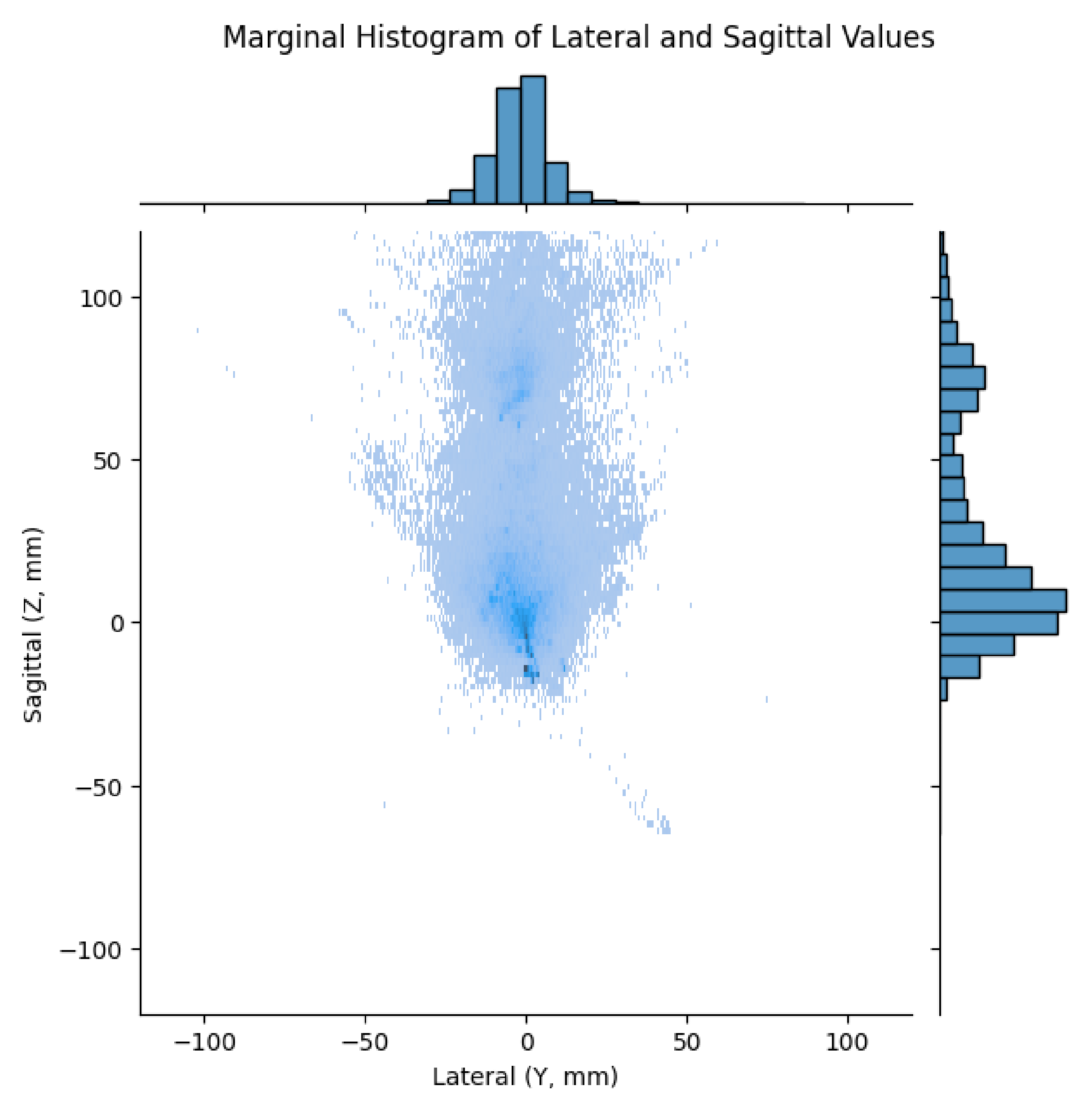
```
ax.set_xlim(-max_val, max_val)
ax.set_ylim(0, max_val)
```

[5]:
```
(0.0, 450.0)
```



This is a heatmap of the person viwed from above. In this case the person was primaliry bend foward and head more movement in the sagittal plane than in the lateral.

[6]:
```
import numpy as np
import seaborn as sns

from math import pi

f = 180 / pi

# Concatenate all y and z coordinates from all measurements
all_y = [f * m.lateral_flexion for m in data.measurements]
all_z = [f * m.sagittal_flexion for m in data.measurements]
sns.jointplot(x=all_y, y=all_z, kind="hist", marginal_kws=dict(bins=30, fill=True))
plt.xlabel('Lateral (Y, mm)')
plt.ylabel('Sagittal (Z, mm)')
plt.suptitle('Marginal Histogram of Lateral and Sagittal Values', y=1.02)
plt.xlim(-120, 120)
plt.ylim(-120, 120)
plt.show()
```



[7]:
```
def measurement_to_3d_coords(m : Measurement):
    reconstruction = m.full_reconstruction

    # Stack left and right edge coordinates into arrays
    left = np.vstack([reconstruction.left.x, reconstruction.left.y, reconstruction.
↪left.z]).T
    right = np.vstack([reconstruction.right.x, reconstruction.right.y, reconstruction.
↪right.z]).T

    return left, right, np.vstack([left[:, 0], right[:, 0]]), np.vstack([left[:, 1],␣
↪right[:, 1]]), np.vstack([left[:, 2], right[:, 2]])
```

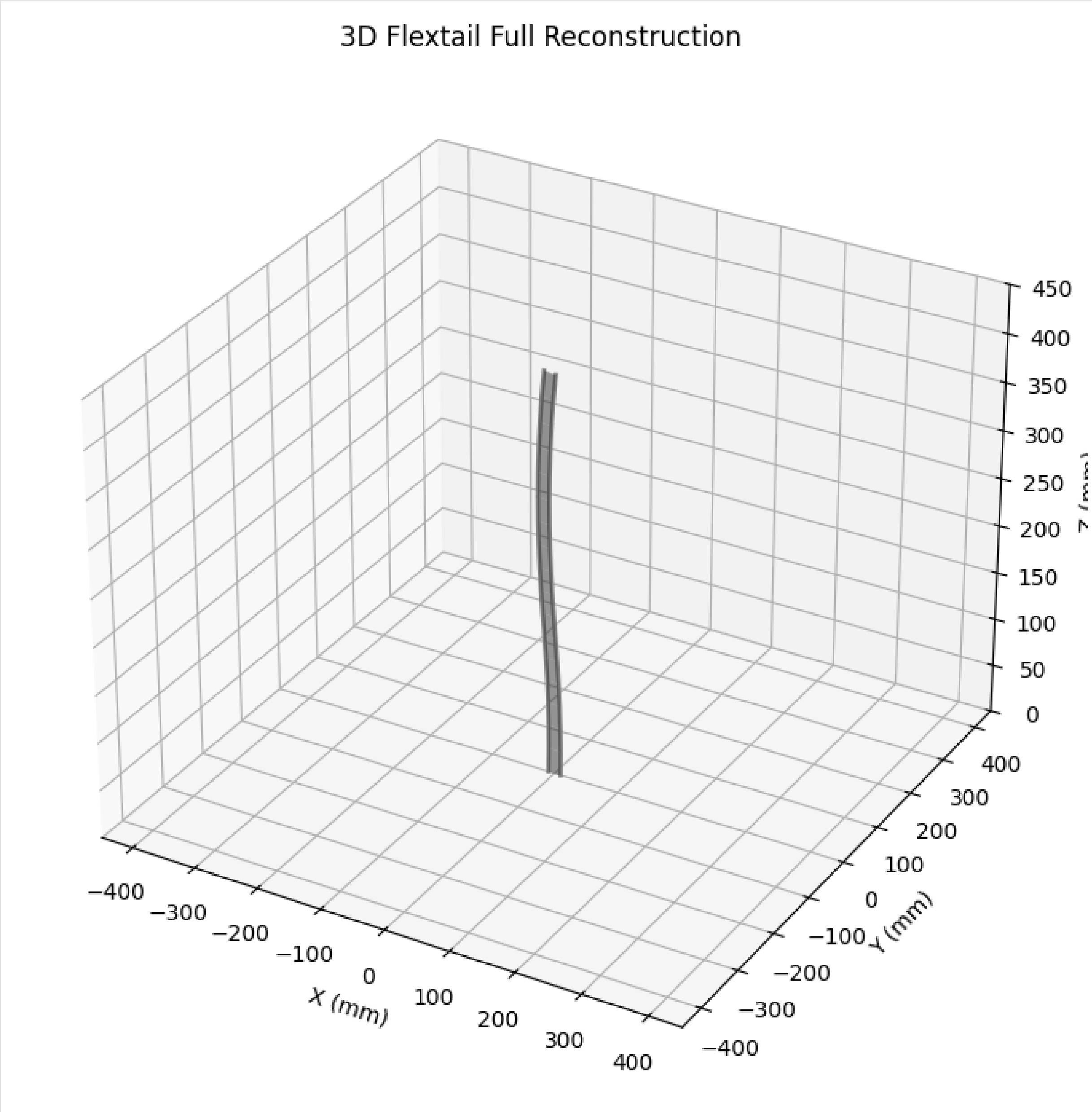
This code creates a 3D-Plot of the sensor

[8]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
```

(continues on next page)

(continued from previous page)

```
# Use the full reconstruction object

fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')


left, right, X, Y, Z = measurement_to_3d_coords(m)
# Plot the 3D foil as a surface
ax.plot_surface(X, Y, Z, color='gray', alpha=0.5, rstride=1, cstride=1, linewidth=0)

# Optionally plot the left and right edges
ax.plot(left[:, 0], left[:, 1], left[:, 2], color='gray', alpha=1, linewidth=2)
ax.plot(right[:, 0], right[:, 1], right[:, 2], color='gray', alpha=1, linewidth=2)

ax.set_xlabel('X (mm)')
ax.set_ylabel('Y (mm)')
ax.set_zlabel('Z (mm)')

lim = dist * 18
ax.set_xlim(-lim, lim)
ax.set_ylim(-lim, lim)
ax.set_zlim(0, lim)
ax.set_title('3D Flextail Full Reconstruction')
plt.tight_layout()
plt.show()
```


3D Flextail Full Reconstruction
X (mm)
Y (mm)
Z (mm)
−400
−300
−200
−100
0
100
200
300
400
0
50
100
150
200
250
300
350
400
450

CHAPTER

ELEVEN

DISHWASHER DEMO

This notebook provides a straightforward overview of ergonomic analysis for unloading a dishwasher, using sensor data to detect bending periods and evaluate posture quality. The data features here is a person that is unloading a dishwasher for 1 minute measured with a frequency of 25 Hz.

The data analysis includes:

- Loading and processing time series data for body angles and movement.
- Detecting "bending forward" actions
- Scoring posture quality during each bending period based on lumbar angle and torso twist.

[1]:
```
# flexlib is a python library written by Minktec to facilitate easy flextail data␣
↪processing
import flexlib
from flexlib import MeasurementEvaluationMetric as mem
from flexlib import CountingPredicateSchmittTrigger, SchmittState

import seaborn as sns # plotting
import matplotlib.pyplot as plt # plotting
import pandas as pd # dataframes / tables
import numpy as np # math
```

After importing the libraries, we can load the data recorded with the FlexTail-Sensor. The `create_dataframe` method takes the raw values and computes metrics derived from the sensor data. This makes it easy to work with the sensor data. If more complicated evaluations are necessary, the sensor angles or coordinates can also be computed.

[2]:
```
measurements : flexlib.AnnotatedRecording = flexlib.FlexReader().parse("../docs/test_
↪data/dishwasher.rsf")
df : pd.DataFrame = flexlib.create_dataframe(measurements)
df
```

```
e:\flexlib\.venv\Lib\site-packages\flexlib\dataframe_builder.py:47: FutureWarning:␣
↪Setting an item of incompatible dtype is deprecated and will raise an error in a␣
↪future version of pandas. Value 'loading dishwasher' has dtype incompatible with␣
↪float64, please explicitly cast to a compatible dtype first.
  df.loc[closest_idx, 'annotation'] = annotation.label
```

[2]:
```
                        Time          annotation  lumbarAngle     twist  \
0     2024-12-10 08:30:19.051  loading dishwasher    -0.302237 -0.032632
1     2024-12-10 08:30:19.118                 NaN    -0.306199 -0.017651
2     2024-12-10 08:30:19.169                 NaN    -0.305224 -0.021672
3     2024-12-10 08:30:19.200                 NaN    -0.279675  0.032262
4     2024-12-10 08:30:19.258                 NaN    -0.318241 -0.013603
...                       ...                 ...          ...       ...
1480  2024-12-10 08:31:19.709                 NaN     0.119518  0.103478
1481  2024-12-10 08:31:19.743                 NaN     0.123758  0.083975
1482  2024-12-10 08:31:19.829                 NaN     0.125396  0.081779
```

(continues on next page)

(continued from previous page)

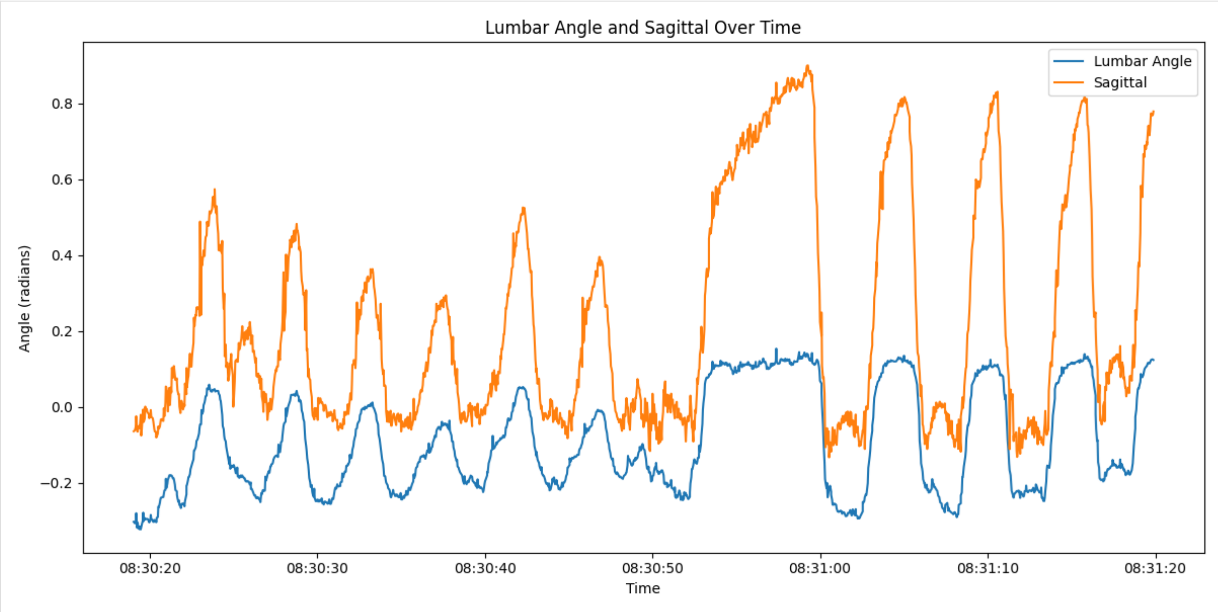
```
1483 2024-12-10 08:31:19.860                    NaN     0.124641  0.091700
1484 2024-12-10 08:31:19.888                    NaN     0.123758  0.086738

       lateral  sagittal  lateralApprox  sagittalApprox  thoracicAngle  \
0     0.071393 -0.064038       0.046761       -0.027765       0.310219
1     0.062159 -0.061405       0.034435       -0.025203       0.306267
2     0.070013 -0.053050       0.041685       -0.016517       0.308264
3     0.101480 -0.024342       0.069655        0.007804       0.311819
4     0.077045 -0.064217       0.044471       -0.031815       0.311717
...        ...       ...            ...             ...            ...
1480  0.370382  0.748794       0.340798        0.833886       0.474724
1481  0.373323  0.773125       0.341342        0.858399       0.474082
1482  0.402006  0.767567       0.369646        0.852625       0.469927
1483  0.416074  0.769438       0.381614        0.854627       0.468836
1484  0.407534  0.777981       0.374646        0.863071       0.471117

      acceleration  gyro
0           0.9830  1.66
1           1.0015  1.36
2           0.9830  1.16
3           0.9675  0.80
4           1.0000  0.26
...            ...   ...
1480        0.8685  3.14
1481        0.8670  3.46
1482        0.8530  2.94
1483        0.8560  2.84
1484        0.8405  2.06

[1485 rows x 11 columns]
```

Lumbar Angle:
The lumbar angle is computed from sensor data placed on the lower back, representing the flexion or extension of the lumbar. It quantifies how much the lower back bends forward or backward during movement.

Sagittal
The sagittal angle is the value of the torso's forward/backward tilt in the sagittal plane, derived from the sensor shape and the case position.

Both values are reported in radians where positive means bending forward.

[3]:
```
plt.figure(figsize=(12, 6))
plt.plot(df['Time'], df['lumbarAngle'], label='Lumbar Angle')
plt.plot(df['Time'], df['sagittal'], label='Sagittal')
plt.xlabel('Time')
plt.ylabel('Angle (radians)')
plt.title('Lumbar Angle and Sagittal Over Time')
plt.legend()
plt.tight_layout()
plt.show()
```



11.1 Schmitt Trigger for Bending Detection

Next, we want to know when the person is bending forward. For this, we use a Schmitt trigger.
This allows us to efficiently filter out noise or jitter caused by the person.
If we were to count everything as bending where the sagittal angle is > 30 degrees, it could lead to very short or undefined edges in the detected timestamps. The Schmitt trigger allows us to set a trigger threshold (high), in this example 30 degrees, and a release threshold (low), for example 25 degrees.

In this case, we use a `CountingSchmittTrigger`, which only changes the state when the set thresholds are met for a certain number of measurements.

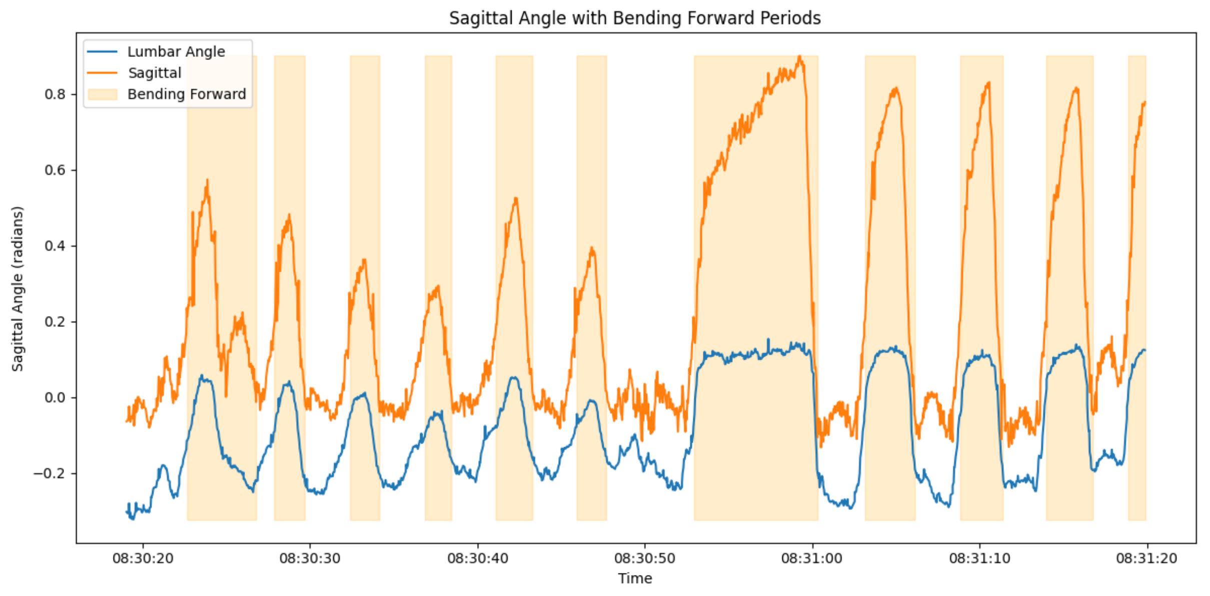
[7]:

```
# Define thresholds and counts for the Schmitt trigger
sagittal_high = 0.15
sagittal_low = 0.05
high_count = 3  # Number of consecutive points above high to trigger
low_count = 3   # Number of consecutive points below low to reset

# Define predicates for the sagittal signal
high_predicate = lambda x: x > sagittal_high
low_predicate = lambda x: x < sagittal_low

# Create the CountingPredicateSchmittTrigger
trigger = CountingPredicateSchmittTrigger(
    high_predicate=high_predicate,
    low_predicate=low_predicate,
    high_count=high_count,
    low_count=low_count
)

# Track the trigger state for each time point
bending_mask = []
for val in df['sagittal']:
    trigger.add(val)
```

(continues on next page)

(continued from previous page)

```
    bending_mask.append(trigger.state == SchmittState.HIGH)

df['bending'] = bending_mask

plt.figure(figsize=(12, 6))
plt.plot(df['Time'], df['lumbarAngle'], label='Lumbar Angle')
plt.plot(df['Time'], df['sagittal'], label='Sagittal')
plt.fill_between(df['Time'], df['lumbarAngle'].min(), df['sagittal'].max(), where=df[
↪'bending'], color='orange', alpha=0.2, label='Bending Forward')
plt.xlabel('Time')
plt.ylabel('Sagittal Angle (radians)')
plt.title('Sagittal Angle with Bending Forward Periods')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```



Great. We can clearly see that the periods marked as being bend forward match our expecations and are clearly separated. In this case one "Bending Forward" block represents taking a mug or plate out of the dishwasher.

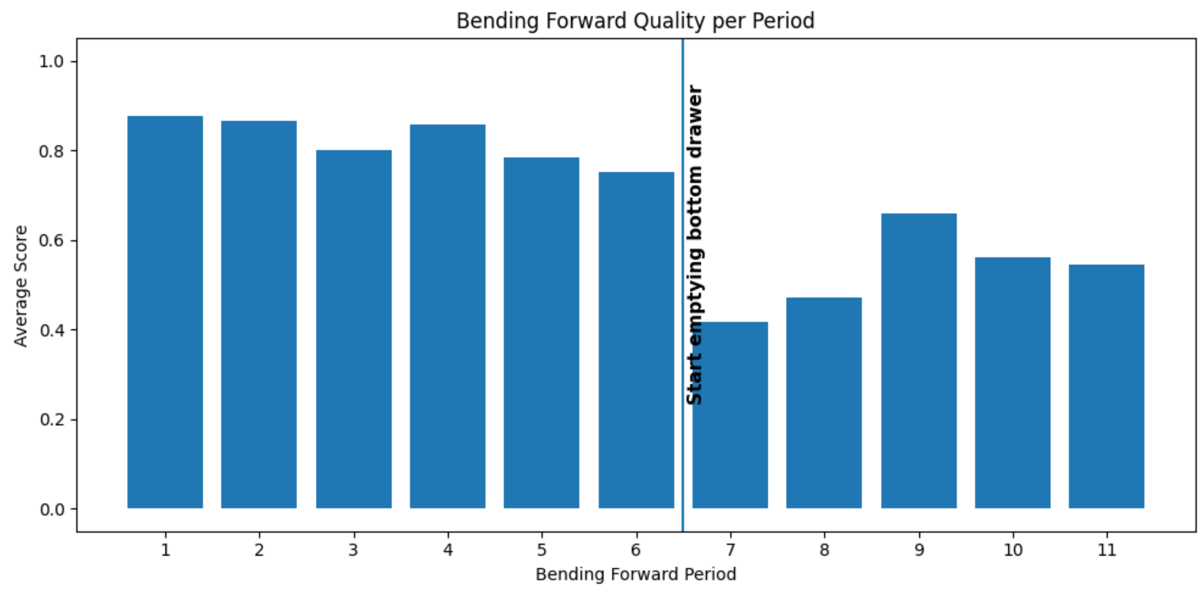
Next we want to do a simple evaluation of the ergonomic quality of the movements. For that we use a combination of the bending of the lumbar spine and the rotation of the thorso `twist`. We are going to rate the movements from 0 (bad) to 1 (good).

[9]:
```
lumbar = df['lumbarAngle']
twist = df['twist']

# Score: 1 is perfect, 0 is bad; penalize more when both are bad
# This score is kind of arbitrary and can be adjusted based on the specific␣
↪requirements
lumbar_norm = (lumbar / 0.5).clip(0, 1)
twist_norm = (twist.abs() / 0.15).clip(0, 1)
df['score'] = 1 - (lumbar_norm + twist_norm) / 3 - (lumbar_norm * twist_norm)

# Find contiguous bending forward periods using pandas magic
bending_periods = [group.index.tolist() for _, group in df[df['bending']].groupby((~
↪df['bending']).cumsum())]
```

(continues on next page)

(continued from previous page)

```
# Compute average score for each bending period (simple version)
period_scores = []
for i, period in enumerate(bending_periods):
    avg = df['score'].loc[period].mean()
    period_scores.append({'period': str(i+1), 'score': avg})

period_scores_df = pd.DataFrame(period_scores)

plt.figure(figsize=(10, 5))
plt.bar(period_scores_df['period'], period_scores_df['score'])
plt.xlabel('Bending Forward Period')
plt.ylabel('Average Score')
plt.title('Bending Forward Quality per Period')
plt.ylim(-0.05, 1.05)

x_pos = 5.5
plt.axvline(x=x_pos)
plt.text(x_pos+0.05, 0.95, 'Start emptying bottom drawer', rotation=90, va='top',␣
↪fontsize=11, fontweight='bold')

plt.tight_layout()
plt.show()
```



Based on our simple score we can clearly see that once the person starts unloading the bottom tray of the dishwasher, the ergonomic quality starts to degrade.

11.2 Summary

We have:

1. Loaded sensor data
2. Created a plot
3. Detected bending forward periods
4. Evaluated the ergonomic impact