Ultra Motion Capture

Release v0.0

Dr. Kit-lun Yick's Research Team

Nov 24, 2022

TABLE OF CONTENTS

Attention: This project is under active development.

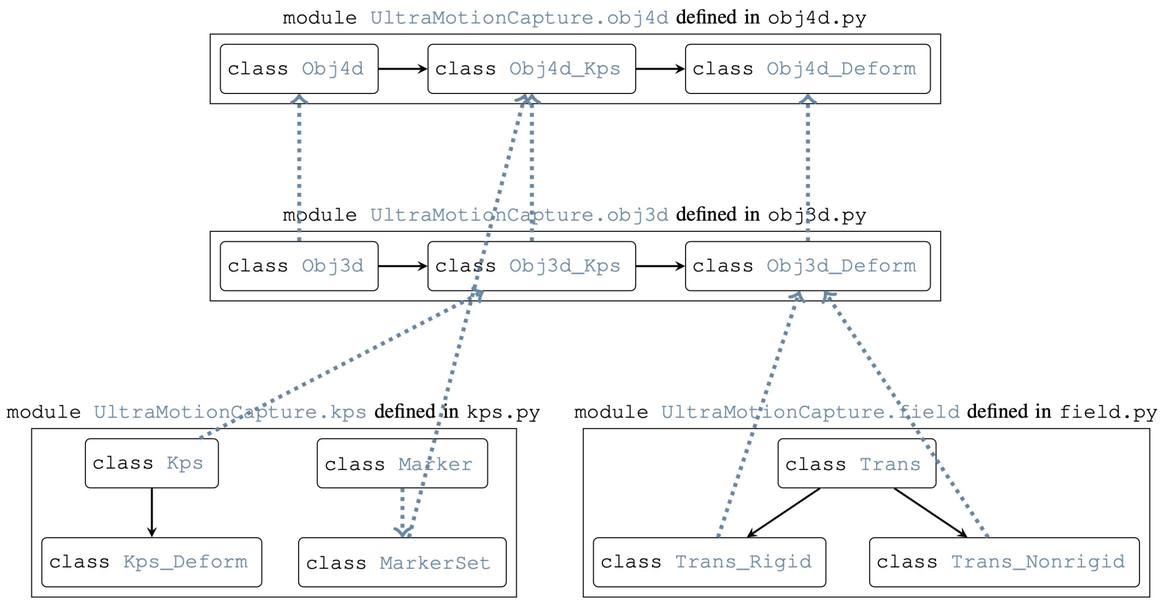

Fig. 1: Fig. Overall structure of the core modules.

This package (repository link) is developed for the data processing of the 3dMD 4D scanning system. Comparing with traditional motion capture system, such as Vicon:

- Vicon motion capture system can provided robust & accurate key points tracking based on physical marker points attached to human body. But it suffer from the lack of continuous surface deformation information.
- 3dMD 4D scanning system can record continuous surface deformation information. But it doesn't provide key point tracking feature and it's challenging to track the key points via Computer Vision approach, even with the state of the art methods in academia[1].

To facilitate human factor research, we deem it an important task to construct a hybrid system that can integrate the advantages and potentials of both systems. The motivation and the core value of this project can be described as: *adding continuous spatial dynamic information to Vicon* or *adding discrete key points information to 3dMD*, leading to advancing platform for human factor research in the domain of dynamic human activity.

Tip: Before jump into the *API References*, please read the *Development Notes* and *Design Principles* to get an overall understanding of the technical settings and program structure.

[1] Min, Z., Liu, J., Liu, L., & Meng, M. Q.-H. (2021). Generalized coherent point drift with multi-variate gaussian distribution and watson distribution. IEEE Robotics and Automation Letters, 6(4), 6749–6756. https://doi.org/10.1109/lra.2021.3093011

CHAPTER

ONE

DEVELOPMENT NOTES

1.1 Documentation

The documentation web pages can be found in `docs/build/html/`. Please open `index.html` to enter the documentation which provides comprehensive descriptions and working examples for each class and function we provided.

The documentation is generated with Sphinx. If you are not familiar with it, I would recommend two tutorials for quick-start:

- A "How to" Guide for Sphinx + ReadTheDocs - sglvladi provides an easy-to-follow learning curve but omitted some details.
- Getting Started - Sphinx is harder to understand, but provides some essential information to understand the background, which is vital even for very basic configuration.

1.2 Project Code

The project code is stored in `UltraMotionCapture/` folder. Under it is a `data/` folder, a default output folder `output/`, a `config/` folder storing configuration variables.

Except for these folders, you must have noticed that there are also some `.py` files, including `utils.py`, `field.py`, `kps.py`, `obj3d.py`, and `obj4d.py`. These are **core modules** for this package. They provide a skeleton for building any downstream analysis task and shall not be modified unless there are very strong reasons to do so.

Other than these, there is an `analysis/` folder haven't been discussed. It's the `UltraMotionCapture.analysis` sub-package storing all downstream analysis modules. At current stage, it's not completed and is still under active development.

1.3 Version Control

We use `git` as the version control tool, which is very popular among developers and works seamlessly with GitHub. If you are not familiar with it, I would recommend this tutorial for quick-start: Git Tutorial - Xuefeng Liao

Following is a series of notes that summarise major commands:

- 001-Repository Initialisation
- 002-Local Repository Operation
- 003-Remote Repository Operation

1.4 Dependencies

This project is built upon various open-source packages. All dependencies are listed in `requirements.txt`, please install them properly under a Python 3 environment.

CHAPTER

TWO

DESIGN PRINCIPLES

As discussed in *Project Code*, there are 5 `.py` files directly under the `UltraMotionCapture/` folder. These are 4 **core modules** (*`UltraMotionCapture.obj4d`*, *`UltraMotionCapture.obj3d`*, *`UltraMotionCapture.kps`*, *`UltraMotionCapture.field`*) and 1 **auxiliary modules** (*`UltraMotionCapture.utils`*) of this project.

They serve as the skeleton to support any downstream analysis task. Therefore, it's necessary to understand their inner relationship so that you would know how to utilise it, tweak it, and advance it to fit your customised need.

2.1 What makes an analysable 4D scene?

As suggested by the name, 4D scanning is an imaging system that can record 3D + T data, i.e. 3 dimensions of space and 1 dimension of time. To be more specific, the 3dMD 4D scanning system records 3D image series in very high time- and space- resolution.

Therefore, it can provide very rich information on the dynamic movement and deformation of the human body during active activities. However, there is a crucial lack of inner-relationship information between different frames of 3D images:

> **Attention:** For example, with the 5th and 6th frames of 3D images, we know that the former one transforms into the next one. However, for any specific point in the 5th frame, we don't know which point in the 6th frame it transfers to.
>
> *Such lack of information blocks the way of any sophisticated and thematic analysis of the 4D data*, such as tracing the movement of the nipple points and tracking the variation of the upper arm area during some kind of sports activity.

Actually, the whole *`UltraMotionCapture`* project is motivated and centred around this bottleneck problem. We aims at revealing the so-called inner-relationship information between different frames. **An analysable 4D scene must consists such information**. At the most meticulous level, the inner-relationship information can be represented as a *displacement field*. That is, specifically, in which direction and to what distance a point in a 3D frame is moving.

2.2 Construction of the analysable 4D scene

The development of the core modules and 1 auxiliary modules (`UltraMotionCapture.utils`) is centred around constructing an *analysable 4D scene*. It follows such a pattern:

- The 4D object defined in `UltraMotionCapture.obj4d` contains of a series of 3D objects.
- The 3D object defined in `UltraMotionCapture.obj3d` contains the loaded mesh, point cloud, key point coordinates (provided by Vicon), and the estimated displacement field.
- Tools for handling key point coordinates and estimating displacement field are provided in `UltraMotionCapture.kps` and `UltraMotionCapture.field`, respectively.

At this stage, the structure is still quite clear, right? It's even more simple in actual usage:

```
import UltraMotionCapture as umc

# load Vicon motion capture data
vicon = umc.kps.MarkerSet()
vicon.load_from_vicon('data/6kmh_softbra_8markers_1.csv')
vicon.interp_field()

# load 3dMD scanning data
o3_ls = umc.obj3d.load_obj_series(
 folder='data/6kmh_softbra_8markers_1/',
 start=0,
 end=1,
 sample_num=1000,
 obj_type=umc.obj3d.Obj3d_Deform
)

# initialise 4D scene object
o4 = umc.obj4d.Obj4d_Deform(
 markerset=vicon,
 fps=120,
 enable_rigid=True,
 enable_nonrigid=True,
)

# load 3D data to 3d scene
# the displacement field estimation will be implemented automatically
o4.add_obj(*o3_ls)
```

In this code example, a 4D scene object for further analysis is prepared in 4 steps.

Tip: For specific meaning and usage of these functions and their inputs, please refer to the *API References*.

2.3 Object-orientated development

Now we have an analysable 4D object, but inside it, there aren't many functions for analysis. What's the deal?

It's related to our *design idea*: object-orientated development. If you're not familiar with this idea, let me explain some bit of it to you. Object-orientated development, aka object-orientated programming, is a kind of programming paradigm that abstracts the programming problem into objects. In the real world, problems are always related to *objects*. For example:

Example

The traffic light scheduling problem is consist of 3 types of objects, aka 3 classes: `car`, `human`, and `road`. Each of the objects contains a series of actions (represented as a function in programming), variables (such as speed and size, represented as attributes in programming), and interactions with other objects.

In our project, the object relationship is:

- 4D object contains a series of 3D objects.
- 3D object contains mesh object, point cloud object, key points object, and displacement field object.

It makes it much easier to manage complex elements with numerous parameters that need to be taken care of since it groups elements into independent objects in a logical way. However, that's only half of the advantage of object-orientated programming. **Inheritance** is even more powerful. Let's go back to the traffic light scheduling example:

Example

There are various kinds of `car`, such as `bus`, `truck`, and `van`. They are all `car`s so they should share some common attributes and functions with `car` objects, while they all have some special attributes and functions.

In object-orientated programming, a `bus` class can be defined as derived from the `car` object, inheriting attributes and functions of `car` class, and adding/revising its supplement attributes and functions. And so do the `truck` and `van` classes.

With inheritance, the development of classes can lay out in an incremental fashion. If you jump into *`UltraMotionCapture.obj3d`* and *`UltraMotionCapture.obj4d`*, you will find that they all contain 3 classes, 1 without suffix and 2 with suffixe `_Kps` or `_Deform`:

- The classes without suffix (*`Obj3d`* and *`Obj4d`*) are the basic classes for 3D and 4D objects. Only basic features like loading from 3dMD scanning data and sampling the point cloud are realised.
- The classes with suffix `_Kps` (*`Obj3d_Kps`* and *`Obj4d_Kps`*) are derived from *`Obj3d`* and *`Obj4d`*, respectively. The major development is attaching key points (*`Kps`*) to it.
- The classes with suffix `_Deform` (*`Obj3d_Deform`* and *`Obj4d_Deform`*) are derived from *`Obj3d_Kps`* and *`Obj4d_Kps`*, respectively. The major development is attaching the displacement field (*`Trans_Rigid`* and *`Trans_Nonrigid`*) to it.

Tip: At the page of each module, such as *`UltraMotionCapture.obj4d`*, an inheritance relationship graph is shown under the table of classes.

Now let's go back to the original question: **the 4D object class provided by the core modules doesn't have many functions for analysis. Such functions will be realised in the derived classes.** Specifically, when we'd like to extend the ability of any of the classes that we discussed upwards, we derive a new class and insert/revise attributes or functions in it to fit the need.

In this way, with one set of unified core modules, this package can be fine-tuned for any future analysis demands. **With all the extended classes serving for advanced analysis, we form a future-proof, evolvable ecosystem for human factor research**.

Danger: The extended classes should be placed in *UltraMotionCapture.analysis* sub-package. Since the core modules are providing a skeleton for all downstream classes, it shall not be modified unless there are very strong reasons to do so, otherwise unpredictable issues may emerge.

Attention: Considering the intensive use of object-orientated development, developers involved in this project are expected to be proficient in Python object-orientated programming.

2.4 Overall structure of the core modules

The overall structure of the core modules is illustrated below. Noted that the solid arrow pointing from `class A` to `class B` indicates that `class B` is derived from `class A`, while the dotted arrow indicates that a `class A` object contains a `class B` object as an attribute:

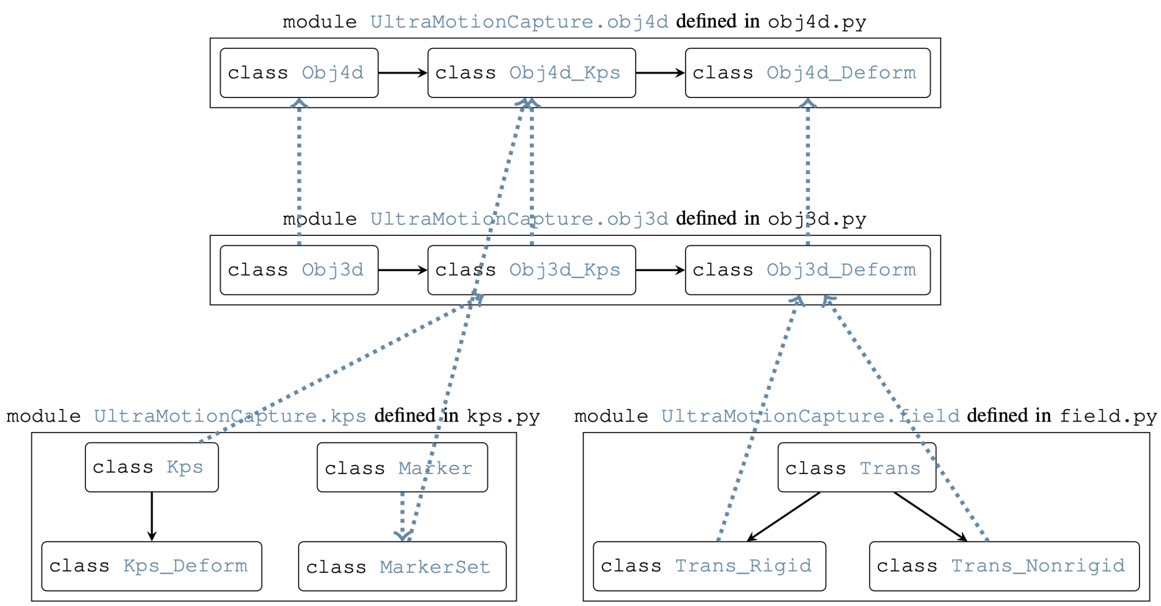

Fig. 1: Fig. Overall structure of the core modules.

CHAPTER

THREE

API REFERENCES

UltraMotionCapture

3.1 UltraMotionCapture

Modules

UltraMotionCapture.analysis	
UltraMotionCapture.config	
UltraMotionCapture.field	The 3dMD 4D scanning device can record 3D images series in very high time- and space-resolution, which provides very rich information of the dynamic movement and deformation of human body during active activities.
UltraMotionCapture.kps	The *UltraMotionCapture.kps* module stands for *key points*.
UltraMotionCapture.obj3d	The 4D object is consist of a series of 3D objects.
UltraMotionCapture.obj4d	The 4D object contains a series of 3D objects (*UltraMotionCapture.obj3d*).
UltraMotionCapture.utils	

3.1.1 UltraMotionCapture.analysis

3.1.2 UltraMotionCapture.config

3.1.3 UltraMotionCapture.field

The 3dMD 4D scanning device can record 3D images series in very high time- and space-resolution, which provides very rich information of the dynamic movement and deformation of human body during active activities. However, there is a crucial lack of inner-relationship information between different frames of 3D image:

Important: For example, with the 5-th and 6-th frames of 3D images, we know that the former one transforms to the next one. However, for any specific point on the 5-th frame, we don't know which point in the 6-th frame it transfers to.

Such lack of information blocks the way of any sophisticated and thematic analysis of the 4D data, such as tracing the movement of the nipple points and tracking the variation of the upper arm area during some kind of sports activity.

The `UltraMotionCapture.field` aims at revealing the so-called inner-relationship information between different frames. In the context of mathematical, the most meticulous level of such information can be represented as *displacement field* and other kinds of transformation. Actually, the whole `UltraMotionCapture` project is motivated and centred around this bottleneck problem.

Classes

`Trans`(source_obj, target_obj, **kwargs)	The base class of transformation.
`Trans_Nonrigid`(source_obj, target_obj, **kwargs)	The non-rigid transformation, under which points in different locations may be transformed in different directions and distances.
`Trans_Rigid`(source_obj, target_obj, **kwargs)	The rigid transformation, which can be expressed in the form of $\mathcal{T}$:

3.1.3.1 UltraMotionCapture.field.Trans

class UltraMotionCapture.field.**Trans**(*source_obj: Type[obj3d.Obj3d]*, *target_obj: Type[obj3d.Obj3d]*, ***kwargs*)

Bases: `object`

The base class of transformation. Different types of transformation, such as rigid and non-rigid transformation, are further defined in the children classes like `Trans_Rigid` and `Trans_Nonrigid`.

Parameters

- **source_obj** – The source 3D object of the transformation. Any object of the class derived from `UltraMotionCapture.obj3d.Obj3d` is valid.
- **target_obj** – The target 3D object of the transformation. Any object of the class derived from `UltraMotionCapture.obj3d.Obj3d` is valid.

Note:

self.source The source point cloud (`open3d.geometry.PointCloud`) of the transformation.

self.target The target point cloud (`open3d.geometry.PointCloud`) of the transformation.

__init__(*source_obj: Type[obj3d.Obj3d]*, *target_obj: Type[obj3d.Obj3d]*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(source_obj, target_obj, **kwargs)	Initialize self.

3.1.3.2 UltraMotionCapture.field.Trans_Nonrigid

class UltraMotionCapture.field.**Trans_Nonrigid**(*source_obj: Type[obj3d.Obj3d]*, *target_obj: Type[obj3d.Obj3d]*, ***kwargs*)

Bases: `UltraMotionCapture.field.Trans`

The non-rigid transformation, under which points in different locations may be transformed in different directions and distances. Such an idea can be expressed in the form of $\mathcal{T}$:

$$\mathcal{T}(\boldsymbol{S}) = \boldsymbol{S} + \boldsymbol{T}$$

where $\boldsymbol{S} \in \mathbb{R}^{N \times 3}$ and $\boldsymbol{T} \in \mathbb{R}^{N \times 3}$ stand for the original point cloud and the translation matrix, all stored in the form of $N \times 3$ matrix.

Note:

self.source_points The source points $\boldsymbol{S} \in \mathbb{R}^{N \times 3}$ stored in (N, 3) `numpy.array`.

self.deform_points The deformed points $\boldsymbol{S} + \boldsymbol{T} \in \mathbb{R}^{N \times 3}$ stored in (N, 3) `numpy.array`.

self.disp The displacement matrix $\boldsymbol{T} \in \mathbb{R}^{N \times 3}$ stored in (N, 3) `numpy.array`.

Attention: After initialisation, the registration method `regist()` must be called to estimate the non-rigid transformation between the source and target point cloud.

Example

After loading and registration, the rigid transformation parameters can then be accessed, including the scaling rate, the rotation matrix, and the translation vector:

```
import UltraMotionCapture as umc

o3_1 = umc.obj3d.Obj3d('data/6kmh_softbra_8markers_1/speed_6km_soft_bra.000001.obj')
o3_2 = umc.obj3d.Obj3d('data/6kmh_softbra_8markers_1/speed_6km_soft_bra.000002.obj')

trans = umc.field.Trans_Nonrigid(o3_1, o3_2)
trans.regist()
print(trans.deform_points, trans.disp)
```

__init__(*source_obj: Type[obj3d.Obj3d]*, *target_obj: Type[obj3d.Obj3d]*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(source_obj, target_obj, **kwargs)	Initialize self.
regist([method])	The registration method.
shift_points(points)	Implement the transformation to set of points.

regist(*method=<function registration_cpd>*, ***kwargs*)

The registration method.

Parameters

- **method** – At current stage, only methods from `probreg` package are supported. Default as `probreg.cpd.registration_cpd()`.
- ****kwargs** – Configurations parameters of the registration.

See also:

probreg.cpd.registration_cpd

__parse(*tf_param*)

Parse the registration result to provide `self.source_points`, `self.deform_points`, and `self.disp`. Called by *regist()*.

Parameters **tf_param** –

> **Attention:** At current stage, the default registration method is Coherent Point Drift (CPD) method realised by `probreg` package. Therefore the accepted transformation object to be parse is derived from `cpd.CoherentPointDrift`. Transformation object provided by other registration method shall be tested in future development.

__fix()

Fix the registration result. Called by *regist()*.

> **Attention:** At current stage, the fixing logic aligns the deformed points to their closest points in the target point cloud, to avoid distortion effect after long-chain registration procedure. This logic may be discarded or replaced by better scheme in future development.

shift_points(*points: numpy.array*) → numpy.array

Implement the transformation to set of points.

To apply proper transformation to an arbitrary point $\boldsymbol{x}$:

- Find the closest point $\boldsymbol{s}_{\boldsymbol{x}}$ and its displacement $\boldsymbol{t}_{\boldsymbol{x}}$.
- Use $\boldsymbol{t}_{\boldsymbol{x}}$ as $\boldsymbol{x}$'s displacement: $\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{t}_{\boldsymbol{x}}$

> **Warning:** This logic may be replaced by better scheme in future development.

Parameters **points** – N points in 3D space that we want to implement the transformation on. Stored in a (N, 3) `numpy.array`.

Returns (N, 3) `numpy.array` stores the points after transformation.

Return type `np.array`

3.1.3.3 UltraMotionCapture.field.Trans_Rigid

class UltraMotionCapture.field.**Trans_Rigid**(*source_obj: Type[obj3d.Obj3d]*, *target_obj: Type[obj3d.Obj3d]*, ***kwargs*)

Bases: *UltraMotionCapture.field.Trans*

The rigid transformation, which can be expressed in the form of $\mathcal{T}$:

$$\mathcal{T}(\boldsymbol{x}) = s\boldsymbol{R}\boldsymbol{x} + \boldsymbol{t}$$

where $s \in \mathbb{R}$, $\boldsymbol{R} \in \mathbb{R}^{3\times3}$, $\boldsymbol{t} \in \mathbb{R}^3$, $\boldsymbol{x} \in \mathbb{R}^3$ stand for the scaling rate, the rotation matrix, the translation vector, and an arbitrary point under transformation, respectively.

Note:

self.scale the scaling rate s.

self.rot the rotation matrix $\boldsymbol{R}$.

self.t the translation vector $\boldsymbol{t}$.

Attention: After initialisation, the registration method *regist()* must be called to estimate the rigid transformation between the source and target point cloud.

Example

After loading and registration, the rigid transformation parameters can then be accessed, including the scaling rate, the rotation matrix, and the translation vector:

```
import UltraMotionCapture as umc

o3_1 = umc.obj3d.Obj3d('data/6kmh_softbra_8markers_1/speed_6km_soft_bra.000001.obj')
o3_2 = umc.obj3d.Obj3d('data/6kmh_softbra_8markers_1/speed_6km_soft_bra.000002.obj')

trans = umc.field.Trans_Rigid(o3_1, o3_2)
trans.regist()
print(trans.scale, trans.rot, trans.t)
```

__init__(*source_obj: Type[obj3d.Obj3d]*, *target_obj: Type[obj3d.Obj3d]*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(source_obj, target_obj, **kwargs)	Initialize self.
regist([method])	The registration method.
shift_points(points)	Implement the transformation to set of points.
show()	Illustrate the estimated transformation.

regist(*method=<function registration_cpd>*, ***kwargs*)

The registration method.

Parameters

- **method** – At current stage, only methods from `probreg` package are supported. Default as `probreg.cpd.registration_cpd()`.
- ****kwargs** – Configurations parameters of the registration.

  See also:

  probreg.cpd.registration_cpd

__parse(*tf_param: Type[probreg.cpd.CoherentPointDrift]*)

Parse the registration result to provide `self.s`, `self.rot`, and `self.t`. Called by *regist()*.

Parameters **tf_param** –

> **Attention:** At current stage, the default registration method is Coherent Point Drift (CPD) method realised by `probreg` package. Therefore the accepted transformation object to be parse is derived from `cpd.CoherentPointDrift`. Transformation object provided by other registration method shall be tested in future development.

__fix()

Fix the registration result. Called by *regist()*.

> **Attention:** At current stage, the fixing logic only checks the scaling rate and raises a warning in terminal. The underline assumption is that since *UltraMotionCapture* focuses on human body which doesn't scale a lot, the scaling rate shall be closed to 1.

shift_points(*points: numpy.array*) → numpy.array

Implement the transformation to set of points.

Parameters **points** – N points in 3D space that we want to implement the transformation on. Stored in a (N, 3) `numpy.array`.

Returns (N, 3) `numpy.array` stores the points after transformation.

Return type `np.array`

> **Warning:** This method will be realised in future development.

show()

Illustrate the estimated transformation.

> **Warning:** This method will be realised in future development.

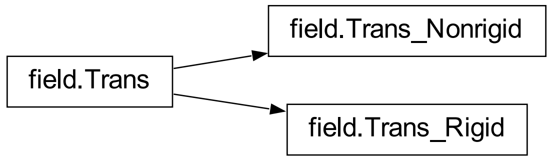

Fig. 1: Inheritance Relationship

Functions

transform_rst2sm(R, s, t)	Transform rigid transformation representation from
transform_sm2rst(s, M)	Transform rigid transformation representation from

3.1.3.4 UltraMotionCapture.field.transform_rst2sm

`UltraMotionCapture.field.`**`transform_rst2sm`**(*R: np.array*, *s: float*, *t: np.array*) → tuple[float, np.array]

Transform rigid transformation representation from

rotation matrix $\boldsymbol{R} \in \mathbb{R}^{3\times 3}$, scaling rate $s \in \mathbb{R}$, and translation vector $\boldsymbol{t} \in \mathbb{R}^3$

to

homogeneous transformation matrix $\boldsymbol{M} \in \mathbb{R}^{4\times 4}$ and scaling rate $s \in \mathbb{R}$.

$$\mathcal{T}(\boldsymbol{x}) = s\boldsymbol{R}\boldsymbol{x} + \boldsymbol{t} = s\boldsymbol{M}\boldsymbol{x}$$

See also:

Homogeneous transformation matrix is a very popular representation of rigid transformation, adopted by `OpenGL` and other computer vision packages. It applies rotation and translation in one 4×4 matrix.

More information: Spatial Transformation Matrices - Rainer Goebel

Parameters

- **`R`** – rotation matrix $\boldsymbol{R} \in \mathbb{R}^{3\times 3}$ stored in (3, 3) `numpy.array`.
- **`s`** – scaling rate $s \in \mathbb{R}$ stored in a `float` variable.
- **`t`** – translation vector $\boldsymbol{t} \in \mathbb{R}^3$ stored in (3,) `numpy.array`.

Returns

- `float` – scaling rate $s \in \mathbb{R}$ stored in a `float` variable.
- `numpy.array` – homogeneous transformation matrix $\boldsymbol{M} \in \mathbb{R}^{4\times 4}$ stored in (4, 4) `numpy.array`.

3.1.3.5 UltraMotionCapture.field.transform_sm2rst

`UltraMotionCapture.field.`**`transform_sm2rst`**(*s: float*, *M: np.array*) → tuple[np.array, float, np.array]

Transform rigid transformation representation from

homogeneous transformation matrix $\boldsymbol{M} \in \mathbb{R}^{4\times 4}$ and scaling rate $s \in \mathbb{R}$.

to

rotation matrix $\boldsymbol{R} \in \mathbb{R}^{3\times 3}$, scaling rate $s \in \mathbb{R}$, and translation vector $\boldsymbol{t} \in \mathbb{R}^3$

Parameters

- `s` – scaling rate $s \in \mathbb{R}$ stored in a `float` variable.
- `M` – homogeneous transformation matrix $\boldsymbol{M} \in \mathbb{R}^{4\times4}$ stored in (4, 4) `numpy.array`.

Returns

- `numpy.python` – rotation matrix $\boldsymbol{R} \in \mathbb{R}^{3\times3}$ stored in (3, 3) `numpy.array`.
- `float` – scaling rate $s \in \mathbb{R}$ stored in a `float` variable.
- `numpy.python` – translation vector $\boldsymbol{t} \in \mathbb{R}^3$ stored in (3,) `numpy.array`.

3.1.4 UltraMotionCapture.kps

The `UltraMotionCapture.kps` module stands for *key points*. In `UltraMotionCapture` package, key points are essential elements to facilitate the processing of 4D images.

There are two different perspectives to arrange key points data: *time-wise* and *point-wise*. Reflecting these two ways of arrangement:

- The `Kps` and *Kps_Deform* contain all key points' data at a specific moment;
- While the `Marker` contains a specific key point's data within a time period. To aggregate all key points' data, `MarkerSet` is provided.

Classes

`Kps`()	A collection of the key points that can be attached to a 3D object, i.e. a frame of the 4D object.
`Kps_Deform`()	Adding deformation feature to the `Kps` class.
`Marker`(name, start_time, fps)	Storing single key point's coordinates data within a time period.
`MarkerSet`()	A collection of `Marker` s.

3.1.4.1 UltraMotionCapture.kps.Kps

class UltraMotionCapture.kps.**Kps**

Bases: `object`

A collection of the key points that can be attached to a 3D object, i.e. a frame of the 4D object.

Note:

self.kps_source_points N key points in 3D space stored in a (N, 3) `numpy.array`.

Example

After initialisation, the *Kps* object is empty. There are two ways to load key points into it:

Manually selecting key points with *select_kps_points()*.

```
import UltraMotionCapture as umc

points = umc.kps.Kps()
points.select_kps_points()  # this will trigger a point selection window
```

Load key points from Vicon motion capture data stored in a *MarkerSet* object with *load_from_markerset_frame()* or *load_from_markerset_time()*.

```
import UltraMotionCapture as umc

vicon = umc.kps.MarkerSet()
vicon.load_from_vicon('data/6kmh_softbra_8markers_1.csv')
vicon.interp_field()

points = umc.kps.Kps()
points.load_from_markerset_frame(vicon)
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
get_kps_source_points()	Get the key points coordinates.
load_from_markerset_frame(markerset[, frame_id])	Load key points to the *Kps* object providing the *MarkerSet* and frame index.
load_from_markerset_time(markerset[, time])	Load key points to the *Kps* object providing the *MarkerSet* and time stamp.
select_kps_points(source)	Interactive manual points selection.
set_kps_source_points(points)	Other than manually selecting points or loading points from Vicon motion capture data, the `kps_source_points` can also be directly overridden with a (N, 3) `numpy.array`, representing N key points in 3D space.

select_kps_points(*source: open3d.cpu.pybind.geometry.PointCloud*)
Interactive manual points selection.

Parameters **source** – an `open3d.geometry.PointCloud` object for points selection.

> **Warning:** At current stage, the interactive manual points selection is realised with `open3d` package. It will be transferred to `pyvista` package in future development.

load_from_markerset_frame(*markerset: UltraMotionCapture.kps.MarkerSet*, *frame_id: int = 0*)
Load key points to the *Kps* object providing the *MarkerSet* and frame index.

Parameters

- **markerset** – a *MarkerSet* object carrying Vicon motion capture data, which contains various frames.
- **frame_id** – the frame index of the Vicon motion capture data to be loaded.

load_from_markerset_time(*markerset: UltraMotionCapture.kps.MarkerSet*, *time: float = 0.0*)
Load key points to the *Kps* object providing the *MarkerSet* and time stamp.

Parameters

- **markerset** – a *MarkerSet* object carrying Vicon motion capture data, which contains various frames.

> **Warning:** Before passing into *load_from_markerset_time()*, call *MarkerSet.interp_field()* first so that coordinates data at any specific time is accessible.

- **time** – the time stamp of Vicon motion data to be loaded.

set_kps_source_points(*points: numpy.array*)
Other than manually selecting points or loading points from Vicon motion capture data, the `kps_source_points` can also be directly overridden with a (N, 3) `numpy.array`, representing N key points in 3D space.

Parameters **points** – (N, 3) `numpy.array`.

get_kps_source_points() → numpy.array
Get the key points coordinates.

3.1.4.2 UltraMotionCapture.kps.Kps_Deform

class UltraMotionCapture.kps.**Kps_Deform**
Bases: *UltraMotionCapture.kps.Kps*

Adding deformation feature to the *Kps* class.

Note:

self.trans An *UltraMotionCapture.field.Trans_Nonrigid* object that stores the deformation information.

self.kps_deform_points (N, 3) `numpy.array`.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
get_kps_deform_points()	Get the key points coordinates after transformation.
get_kps_source_points()	Get the key points coordinates.
load_from_markerset_frame(markerset[, frame_id])	Load key points to the *Kps* object providing the *MarkerSet* and frame index.

continues on next page

Table 10 – continued from previous page

`load_from_markerset_time`(markerset[, time])	Load key points to the *Kps* object providing the *MarkerSet* and time stamp.
`select_kps_points`(source)	Interactive manual points selection.
`set_kps_source_points`(points)	Other than manually selecting points or loading points from Vicon motion capture data, the `kps_source_points` can also be directly overridden with a (N, 3) `numpy.array`, representing N key points in 3D space.
set_trans(trans)	Setting the transformation of the deformable key points object.

set_trans(*trans: field.Trans_Nonrigid*)

Setting the transformation of the deformable key points object.

Parameters **trans** – an *UltraMotionCapture.field.Trans_Nonrigid()* object that represents the transformation.

get_kps_deform_points() → numpy.array

Get the key points coordinates after transformation.

3.1.4.3 UltraMotionCapture.kps.Marker

class UltraMotionCapture.kps.**Marker**(*name: str*, *start_time: float = 0.0*, *fps: int = 100*)

Bases: `object`

Storing single key point's coordinates data within a time period. Usually loaded from the Vicon motion capture data. In this case, a key point is also referred as a marker.

Parameters

- **name** – the name of the marker.
- **start_time** – the start time of the coordinates data.
- **fps** – the number of frames per second (fps).

Note:

self.name The name of the marker.

self.start_time The start time of the coordinates data.

self.fps The number of frames per second (fps).

self.coord $3 \times N$ `numpy.array` storing the coordinates data, with x, y, z as rows and frame ids as the columns.

self.speed $3 \times N$ `numpy.array` storing the speed data, with x, y, z as rows and frame ids as the columns.

self.accel $3 \times N$ `numpy.array` storing the acceleration data, with x, y, z as rows and frame ids as the columns.

self.frame_num The number of total frames.

self.x_field An `scipy.interpolate.interp1d` object that storing the interpolated function of the x coordinates of all frames. Used for estimated the x coordinate of any intermediate time between frames.

self.y_field An `scipy.interpolate.interp1d` object that storing the interpolated function of the y coordinates of all frames. Used for estimated the y coordinate of any intermediate time between frames.

self.z_field An `scipy.interpolate.interp1d` object that storing the interpolated function of the z coordinates of all frames. Used for estimated the z coordinate of any intermediate time between frames.

Tip: When loading Vicon motion capture data, the whole process is arranged by a `MakerSet` object, which creates *Marker* objects for each key point and loads data into it accordingly. Therefore, usually the end user doesn't need to touch the *Marker* class.

`__init__`(*name: str*, *start_time: float = 0.0*, *fps: int = 100*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(name[, start_time, fps])	Initialize self.
`fill_data`(data_input)	Filling coordinates, speed, and acceleration data, one by one, into the *Marker* object.
`get_frame_coord`(frame_id)	Get coordinates data according to frame id.
`get_time_coord`(time)	Get coordinates data according to time stamp.
`interp_field`()	Interpolating the x, y, z coordinates data to estimate its continues change.
`plot_add_dot`(ax[, dot_start_frame, ...])	Adding motion dots in different frames to the `matplotlib.pyplot.subplot` object created in *`plot_track()`*.
`plot_add_line`(ax[, line_start_frame, ...])	Adding motion track lines to the `matplotlib.pyplot.subplot` object created in *`plot_track()`*.
`plot_track`([line_start_frame, ...])	Plotting the marker motion track.

`fill_data`(*data_input*)
Filling coordinates, speed, and acceleration data, one by one, into the *Marker* object.

Parameters **`data_input`** – $3 \times N$ `numpy.array`.

Attention: Called by the *MarkerSet* object when parsing the Vicon motion capture data (*`MarkerSet.load_from_vicon()`*). Usually the end user don't need to call this method manually.

`interp_field`()
Interpolating the x, y, z coordinates data to estimate its continues change. After that, the coordinates at the intermediate time between frames is accessible.

Warning: Before interpolation, the coordinates data, i.e. `self.coord`, must be properly loaded.

`get_frame_coord`(*frame_id: int*) → numpy.array
Get coordinates data according to frame id.

Parameters **`frame_id`** – index of the frame to get coordinates data.

Returns The structure of the returned array is `array[0-2 as x-z][frame_id]`

Return type `numpy.array`

`get_time_coord`(*time: float*) → numpy.array
Get coordinates data according to time stamp.

Parameters **time** – time stamp to get coordinates data.

Returns The structure of the returned array is `array[0-2 as x-z][time]`

Return type `numpy.array`

> **Warning:** The interpolation must be properly done before accessing coordinates data according to time stamp, which means the *interp_field()* must be called first.

plot_track(*line_start_frame: int = 0*, *line_end_frame: Optional[int] = None*, *dot_start_frame: int = 0*, *dot_end_frame: Optional[int] = None*, *line_alpha: float = 0.5*, *line_width: float = 1.0*, *dot_s: float = 10*, *dot_alpha: float = 0.5*, *dpi: int = 300*, *is_show: bool = True*, *is_save: bool = False*)

Plotting the marker motion track.

Parameters

- **line_start_frame** – start frame of line plotting.
- **line_end_frame** – end frame of line plotting, default as `None`, which means plot till the end.
- **dot_start_frame** – start frame of dot plotting.
- **dot_end_frame** – end frame of dot plotting, default as `None`, which means plot till the end.
- **is_show** – weather show the generated graph or not.
- **is_save** – weather save the generated graph or not.
- **Others** – parameters passed to *plot_add_line()* and *plot_add_dot()* to controlling the appearance.

plot_add_line(*ax: matplotlib.pyplot.subplot*, *line_start_frame: int = 0*, *line_end_frame: Optional[int] = None*, *line_alpha: float = 0.5*, *line_width: float = 1*, ***kwargs*)

Adding motion track lines to the `matplotlib.pyplot.subplot` object created in *plot_track()*.

> **Tip:** About the appearance controlling parameters, please refer to Pyplot tutorial - matplotlib.
>
> Additional appearance controlling parameters can be passed into `**kwargs`, please refer to *args and **kwargs - Python Tips.

plot_add_dot(*ax: matplotlib.pyplot.subplot*, *dot_start_frame: int = 0*, *dot_end_frame: Optional[int] = None*, *dot_s: int = 10*, *dot_alpha: float = 0.5*, ***kwargs*)

Adding motion dots in different frames to the `matplotlib.pyplot.subplot` object created in *plot_track()*.

> **Tip:** About the appearance controlling parameters, please refer to Pyplot tutorial - matplotlib.
>
> Additional appearance controlling parameters can be passed into `**kwargs`, please refer to *args and **kwargs - Python Tips.

3.1.4.4 UltraMotionCapture.kps.MarkerSet

class UltraMotionCapture.kps.**MarkerSet**

Bases: `object`

A collection of *Marker* s. At current stage, it's usually loaded from the Vicon motion capture data.

Note:

self.fps The number of frames per second (fps).

self.points A `Dictonary` of *Marker* s, with the corresponding marker names as their keywords.

Example

The Vicon motion capture data shall be exported as a `.csv` file. After initialising the *MarkerSet* data, we can load it providing the `.csv` file's directory:

```
import UltraMotionCapture as umc

vicon = umc.kps.MarkerSet()
vicon.load_from_vicon('data/6kmh_softbra_8markers_1.csv')
vicon.interp_field()
```

Usually we implement the interpolation after loading the data, as shown in the last line of code. Then we can access the coordinates, speed, and acceleration data of any marker at any specific time:

```
print(vicon.get_frame_coord(10))
print(vicon.get_time_coord(1.0012)
```

We can also access the specific marker with the marker name:

```
print(vicon.points.keys())
print(vicon.points['Bra_Miss Sun:CLAV'].get_frame_coord(10))
print(vicon.points['Bra_Miss Sun:CLAV'].get_time_coord(1.0012))
```

We can also plot and save the motion track as a `.gif` file for illustration:

```
vicon.plot_track(step=3, end_frame=100)
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
get_frame_coord(frame_id)	Get coordinates data according to frame id.
get_time_coord(time)	Get coordinates data according to time stamp.
interp_field()	After loading Vicon motion capture data, the *MarkerSet* object only carries the key points' coordinates in discrete frames.

continues on next page

Table 12 – continued from previous page

load_from_vicon(filedir)	Load and parse data from `.csv` file exported from the Vicon motion capture system.
plot_frame(frame_id[, dpi, is_add_line, ...])	Plot a specific frame.
plot_track([start_frame, end_frame, step, ...])	Plotting the marker motion track.

load_from_vicon(*filedir: str*)
Load and parse data from `.csv` file exported from the Vicon motion capture system.

Parameters **filedir** – the directory of the `.csv` file.

interp_field()
After loading Vicon motion capture data, the *MarkerSet* object only carries the key points' coordinates in discrete frames. To access the coordinates at any specific time, it's necessary to call *interp_field()*.

get_frame_coord(*frame_id: int*) → numpy.array
Get coordinates data according to frame id.

Parameters **frame_id** – index of the frame to get coordinates data.

Returns The structure of the returned array is `array[marker_id][0-2 as x-z][frame_id]`

Return type `numpy.array`

> **Warning:** The returned value will be transferred to *Kps* in future development.

get_time_coord(*time: float*) → numpy.array
Get coordinates data according to time stamp.

Parameters **time** – time stamp to get coordinates data.

Returns The structure of the returned array is `array[marker_id][0-2 as x-z][time]`

Return type `numpy.array`

> **Warning:** The interpolation must be properly done before accessing coordinates data according to time stamp, which means the *interp_field()* must be called first.

> **Warning:** The returned value will be transferred to *Kps* in future development.

plot_track(*start_frame: int = 0*, *end_frame: Optional[int] = None*, *step: int = 1*, *remove: bool = True*, **args*, ***kwargs*)
Plotting the marker motion track.

Parameters

- **start_frame** – start frame of plotting.
- **end_frame** – end frame of plotting, default as `None`, which means plot till the end.
- **step** – Plot 1 frame for every `step` frame. The purpose is reducing graph generating time.
- **remove** – after generating the `.gif` file, remove the frames' images or not.

Tip: Additional appearance controlling parameters can be passed into `**kwargs`, please refer to *args and **kwargs - Python Tips and Pyplot tutorial - matplotlib

plot_frame(*frame_id: int*, *dpi: int = 300*, *is_add_line: bool = True*, *is_show: bool = True*, *is_save: bool = False*, ***kwargs*)

Plot a specific frame.

Parameters

- **frame_id** – index of the frame to be plotted.
- **dpi** – the dots per inch (dpi) of the generated graph, controlling the graph quality.
- **is_add_line** – weather add track links or not
- **is_show** – weather show the generated graph or not.
- **is_save** – weather save the generated graph or not.

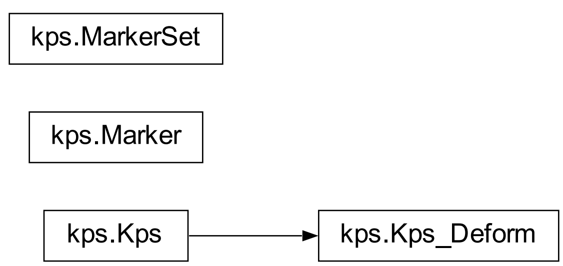

Fig. 2: Inheritance Relationship

3.1.5 UltraMotionCapture.obj3d

The 4D object is consist of a series of 3D objects. In `UltraMotionCapture.obj3d`, 3D object classes with different features and capabilities are developed, serving for different analysis needs and scenarios. At current stage, there are 3 types of 3D object:

- Static 3D object `Obj3d`

  It loads `.obj` 3D mesh image and sampled it as the point cloud.

- Static 3D object `Obj3d_Kps` with key points

  It's derived from `Obj3d` and attach the key points (`UltraMotionCapture.kps.Kps`) to it.

- Dynamic/Deformable 3D object `Obj3d_Deform`

  It's derived from `Obj3d_Kps` and attach the rigid transformation (`UltraMotionCapture.field.Trans_Rigid`) and non-rigid deformation (`UltraMotionCapture.field.Trans_Nonrigid`) to it.

Moreover, a wide range of utils functions are provided, serving for 3D images loading, processing, format transformation, ect.

Classes

Obj3d(filedir, scale_rate, scale_center, ...)	The basic 3D object class.
Obj3d_Deform(**kwargs)	The dynamic/deformable 3D object with key points and transformations attached to it.
Obj3d_Kps(**kwargs)	The 3D object with key points attached to it.

3.1.5.1 UltraMotionCapture.obj3d.Obj3d

class `UltraMotionCapture.obj3d.`**`Obj3d`**(*filedir: str*, *scale_rate: float = 0.01*, *scale_center: list = (0, 0, 0)*, *sample_num: int = 1000*)

Bases: `object`

The basic 3D object class. Loads `.obj` 3D mesh image and sampled it as the point cloud.

Parameters

- **`filedir`** – the direction of the 3D object.
- **`scale_rate`** – the scaling rate of the 3D object.

  See also:

  Reason for providing `scale_rate` parameter is explained in *Obj3d_Deform*.
- **`scale_center`** – the center of the scaling represented in (3,) `List`.
- **`sample_num`** – the number of the points sampled from the mesh to construct the point cloud.

Note:

self.mesh_o3d 3D mesh (`open3d.geometry.TriangleMesh`) loaded with `open3d` .

self.pcd 3D point cloud (`open3d.geometry.PointCloud`) sampled from `self.mesh_o3d`.

Attention: In future development, mesh may also be loaded with `pyvista` as `self.mesh_pv`, for its advantages in visualisation and some specific geometric analysis features.

Example

```
import UltraMotionCapture as umc
o3 = umc.obj3d.Obj3d(
    filedir = 'data/6kmh_softbra_8markers_1/speed_6km_soft_bra.000001.obj',
)
o3.show()
```

`__init__`(*filedir: str*, *scale_rate: float = 0.01*, *scale_center: list = (0, 0, 0)*, *sample_num: int = 1000*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(filedir[, scale_rate, ...])	Initialize self.
show()	Show the loaded mesh and the sampled point cloud.

show()
Show the loaded mesh and the sampled point cloud.

> **Attention:** Currently the visualisation is realised with `open3d`. It will be transferred to `pyvista` in future development for richer illustration features.

3.1.5.2 UltraMotionCapture.obj3d.Obj3d_Deform

class UltraMotionCapture.obj3d.**Obj3d_Deform**(***kwargs*)
Bases: *UltraMotionCapture.obj3d.Obj3d_Kps*

The dynamic/deformable 3D object with key points and transformations attached to it. Derived from *Obj3d_Kps* and attach the rigid transformation (*UltraMotionCapture.field.Trans_Rigid*) and non-rigid deformation (*UltraMotionCapture.field.Trans_Nonrigid*) to it.

Parameters ****kwargs** – parameters can be passed in via keywords arguments. Please refer to *Obj3d* and *Obj3d_Kps* for accepted parameters.

> **Attention:** The transformations (*UltraMotionCapture.field*) are estimated via registration. For effective registration iteration, as an empirical suggestion, the absolute value of coordinates shall falls into or near $(-1, 1)$. That's why we provide a `scale_rate` parameter defaulted as 10^{-2} in the initialisation method of the base class (*Obj3d*).

Note:

self.trans_rigid the rigid transformation (*UltraMotionCapture.field.Trans_Rigid*) of the 3D object.

self.trans_nonrigid the non-rigid transformation (*UltraMotionCapture.field.Trans_Nonrigid*) of the 3D object.

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(**kwargs)	Initialize self.
offset_rotate()	Offset the rotation according to the estimated rigid transformation.
set_trans_nonrigid(trans_nonrigid)	Set non-rigid transformation.
set_trans_rigid(trans_rigid)	Set rigid transformation.
show()	Show the loaded mesh and the sampled point cloud.

set_trans_rigid(*trans_rigid: field.Trans_Rigid*)
Set rigid transformation.

Parameters **trans_rigid** – the rigid transformation (*UltraMotionCapture.field.Trans_Rigid*).

set_trans_nonrigid(*trans_nonrigid: field.Trans_Nonrigid*)
Set non-rigid transformation.

Parameters **trans_nonrigid** – the non-rigid transformation (*UltraMotionCapture.field.Trans_Nonrigid*).

offset_rotate()
Offset the rotation according to the estimated rigid transformation.

Tip: This method usually serves for reorientate all 3D objects to a referencing direction, since that the rigid transformation (*UltraMotionCapture.field.Trans_Rigid*) is usually estimated according to the difference of two different 3D object.

3.1.5.3 UltraMotionCapture.obj3d.Obj3d_Kps

class UltraMotionCapture.obj3d.**Obj3d_Kps**(***kwargs*)
Bases: *UltraMotionCapture.obj3d.Obj3d*

The 3D object with key points attached to it. Derived from *Obj3d* and attach the key points (*UltraMotionCapture.kps.Kps*) to it.

Parameters ****kwargs** – parameters can be passed in via keywords arguments. Please refer to *Obj3d* for accepted parameters.

Note:

self.kps key points (*UltraMotionCapture.kps.Kps*) attached to the 3D object.

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(**kwargs)	Initialize self.
show()	Show the loaded mesh and the sampled point cloud.

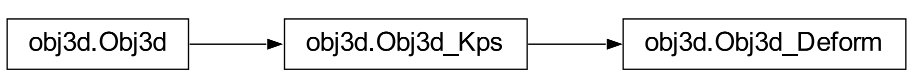

Fig. 3: Inheritance Relationship

Functions

load_obj_series(folder[, start, end, ...])	Load a series of point cloud obj files from a folder.
mesh2pcd(mesh, sample_num)	Sampled a `open3d` mesh (`open3d.geometry.TriangleMesh`) to a `open3d` point cloud (`open3d.geometry.PointCloud`).
mesh_crop(mesh[, min_bound, max_bound])	Crop the mesh (`open3d.geometry.TriangleMesh`) according the maximum and minimum boundaries.
np2pcd(points)	Transform the points coordinates stored in a `numpy.array` to a a `open3d` point cloud (`open3d.geometry.PointCloud`).
np2pvpcd(points, **kwargs)	Transform the points coordinates stored in a `numpy.array` to a a `pyvista` point cloud (`pyvista.PolyData`).
pcd2np(pcd)	Extracted the points coordinates data from a `open3d` point cloud (`open3d.geometry.PointCloud`).
pcd_crop(pcd[, min_bound, max_bound])	Crop the point cloud (`open3d.geometry.PointCloud`) according the maximum and minimum boundaries.
pcd_crop_front(pcd[, ratio])	Crop the front side of a point cloud (`open3d.geometry.PointCloud`) with a adjustable ratio.
pcd_get_center(pcd)	Get the center point of a point cloud.
pcd_get_max_bound(pcd)	Get the maximum boundary of a point cloud.
pcd_get_min_bound(pcd)	Get the minimum boundary of a point cloud.
search_nearest_point(point, target_points)	Search the nearest point from a collection of target points.
search_nearest_point_idx(point, target_points)	Search the index of the nearest point from a collection of target points.

3.1.5.4 UltraMotionCapture.obj3d.load_obj_series

`UltraMotionCapture.obj3d.`**`load_obj_series`**(*folder: str, start: int = 0, end: int = 1, stride: int = 1, obj_type: Type[UltraMotionCapture.obj3d.Obj3d] = <class 'UltraMotionCapture.obj3d.Obj3d'>, **kwargs*) → Iterable[Type[*UltraMotionCapture.obj3d.Obj3d*]]

Load a series of point cloud obj files from a folder.

Parameters

- **`folder`** – the directory of the folder storing the 3D images.
- **`start`** – begin loading from the `start`-th image.

> **Attention:** Index begins from 0. The `start`-th image is included in the loaded images. Index begins from 0.

- **`end`** – end loading at the `end`-th image.

> **Attention:** Index begins from 0. The `end`-th image is included in the loaded images.

- **stride** – the stride of loading. For example, setting `stride=5` means load one from every five 3D images.
- **obj_type** – The 3D object class. Any class derived from *Obj3d* is accepted.
- ****kwargs** – Configuration parameters for initialisation of the 3D object can be passed in via `**kwargs`.

Returns A list of 3D object.

Return type Iterable[Type[*Obj3d*]]

Example

The *load_obj_series()* is usually used for getting a list of 3D object and then loading to the 4D object:

```
import UltraMotionCapture as umc

o3_ls = umc.obj3d.load_obj_series(
    folder='data/6kmh_softbra_8markers_1/',
    start=0,
    end=1,
    sample_num=1000,
)

o4 = umc.obj4d.Obj4d()
o4.add_obj(*o3_ls)
```

3.1.5.5 UltraMotionCapture.obj3d.mesh2pcd

`UltraMotionCapture.obj3d.`**`mesh2pcd`**(*mesh: open3d.cpu.pybind.geometry.TriangleMesh*, *sample_num: int*) → open3d.cpu.pybind.geometry.PointCloud

Sampled a `open3d` mesh (`open3d.geometry.TriangleMesh`) to a `open3d` point cloud (`open3d.geometry.PointCloud`).

See also:

The sampling method is `open3d.geometry.sample_points_poisson_disk()` (link)[1].

Parameters

- **mesh** – the mesh (`open3d.geometry.TriangleMesh`) being sampled.
- **sample_num** – the number of sampling points.

Returns The sampled point cloud.

Return type `o3d.geometry.PointCloud`

[1] Cem Yuksel. “Sample elimination for generating poisson disk sample sets”. Computer Graphics Forum. 2015, 34(2): 25–32.

3.1.5.6 UltraMotionCapture.obj3d.mesh_crop

`UltraMotionCapture.obj3d.`**`mesh_crop`**(*mesh: open3d.cpu.pybind.geometry.TriangleMesh*, *min_bound: list = [- 1000, - 1000, - 1000]*, *max_bound: list = [1000, 1000, 1000]*) → open3d.cpu.pybind.geometry.TriangleMesh

Crop the mesh (`open3d.geometry.TriangleMesh`) according the maximum and minimum boundaries.

Parameters

- **`mesh`** – the mesh (`open3d.geometry.TriangleMesh`) being cropped.
- **`max_bound`** – a list containing the maximum value of x, y, z-coordinates: `[max_x, max_y, max_z]`.
- **`min_bound`** – a list containing the minimum value of x, y, z-coordinates: `[min_x, min_y, min_z]`.

Returns The cropped mesh.

Return type `o3d.geometry.TriangleMesh`

3.1.5.7 UltraMotionCapture.obj3d.np2pcd

`UltraMotionCapture.obj3d.`**`np2pcd`**(*points*)

Transform the points coordinates stored in a `numpy.array` to a a `open3d` point cloud (`open3d.geometry.PointCloud`).

Parameters **`points`** – the points coordinates data stored in a (N, 3) `numpy.array`.

Returns The point cloud (`open3d.geometry.PointCloud`).

Return type `open3d.geometry.PointCloud`

3.1.5.8 UltraMotionCapture.obj3d.np2pvpcd

`UltraMotionCapture.obj3d.`**`np2pvpcd`**(*points*, ***kwargs*)

Transform the points coordinates stored in a `numpy.array` to a a `pyvista` point cloud (`pyvista.PolyData`).

Parameters **`points`** – the points coordinates data stored in a (N, 3) `numpy.array`.

Returns the point cloud (`pyvista.PolyData`).

Return type `pyvista.PolyData`

Attention: Acutally, `pyvista` package doesn't have a specific class to represent point cloud. The returned `pyvista.PolyData` object is a point collection mainly for illustration purpose.

Tip: More configuration parameters can be passed in via `**kwargs`. Please refer to pyvista.PolyData for the accepted parameters.

3.1.5.9 UltraMotionCapture.obj3d.pcd2np

UltraMotionCapture.obj3d.**pcd2np**(*pcd: open3d.cpu.pybind.geometry.PointCloud*) → numpy.array

Extracted the points coordinates data from a `open3d` point cloud (`open3d.geometry.PointCloud`).

Parameters **pcd** – the point cloud (`open3d.geometry.PointCloud`).

Returns the points coordinates data stored in a (N, 3) `numpy.array`.

Return type `numpy.array`

3.1.5.10 UltraMotionCapture.obj3d.pcd_crop

UltraMotionCapture.obj3d.**pcd_crop**(*pcd: open3d.cpu.pybind.geometry.PointCloud*, *min_bound: list = [- 1000, - 1000, - 1000]*, *max_bound: list = [1000, 1000, 1000]*) → open3d.cpu.pybind.geometry.PointCloud

Crop the point cloud (`open3d.geometry.PointCloud`) according the maximum and minimum boundaries.

Parameters

- **pcd** – the point cloud (`open3d.geometry.PointCloud`) being cropped.
- **max_bound** – a list containing the maximum value of x, y, z-coordinates: `[max_x, max_y, max_z]`.
- **min_bound** – a list containing the minimum value of x, y, z-coordinates: `[min_x, min_y, min_z]`.

Returns The cropped point cloud.

Return type `o3d.geometry.PointCloud`

3.1.5.11 UltraMotionCapture.obj3d.pcd_crop_front

UltraMotionCapture.obj3d.**pcd_crop_front**(*pcd: open3d.cpu.pybind.geometry.PointCloud*, *ratio: float = 0.5*) → open3d.cpu.pybind.geometry.PointCloud

Crop the front side of a point cloud (`open3d.geometry.PointCloud`) with a adjustable ratio.

Parameters

- **pcd** – the point cloud (`open3d.geometry.TriangleMesh`) being cropped.
- **ratio** – the ratio of the cropped front part.

Returns the cropped point cloud.

Return type `o3d.geometry.PointCloud`

3.1.5.12 UltraMotionCapture.obj3d.pcd_get_center

UltraMotionCapture.obj3d.**pcd_get_center**(*pcd: open3d.cpu.pybind.geometry.PointCloud*) → numpy.array

Get the center point of a point cloud. The center point is defined as the geometric average point of all points:

$$\boldsymbol{c} = \frac{\sum_i \boldsymbol{p}_i}{N}$$

where N denotes the total number of points.

Parameters **pcd** – the point cloud (`open3d.geometry.TriangleMesh`).

Returns The center point coordinates represented in a (3,) `numpy.array`.

Return type `numpy.array`

3.1.5.13 UltraMotionCapture.obj3d.pcd_get_max_bound

`UltraMotionCapture.obj3d.`**`pcd_get_max_bound`**(*pcd: open3d.cpu.pybind.geometry.PointCloud*) → numpy.array

Get the maximum boundary of a point cloud.

Parameters **`pcd`** – the point cloud (`open3d.geometry.TriangleMesh`).

Returns a list containing the maximum value of x, y, z-coordinates: `[max_x, max_y, max_z]`.

Return type `numpy.array`

3.1.5.14 UltraMotionCapture.obj3d.pcd_get_min_bound

`UltraMotionCapture.obj3d.`**`pcd_get_min_bound`**(*pcd: open3d.cpu.pybind.geometry.PointCloud*) → open3d.cpu.pybind.geometry.PointCloud

Get the minimum boundary of a point cloud.

Parameters **`pcd`** – the point cloud (`open3d.geometry.TriangleMesh`).

Returns a list containing the minimum value of x, y, z-coordinates: `[min_x, min_y, min_z]`.

Return type `numpy.array`

3.1.5.15 UltraMotionCapture.obj3d.search_nearest_point

`UltraMotionCapture.obj3d.`**`search_nearest_point`**(*point: numpy.array*, *target_points: numpy.array*) → numpy.array

Search the nearest point from a collection of target points.

Parameters

- **`point`** – the source point coordinates stores in a (3,) `numpy.array`.
- **`target_points`** – the target points collection stores in a (N, 3) `numpy.array`.

Returns the nearest point coordinates stores in a (3,) `numpy.array`.

Return type `numpy.array`

3.1.5.16 UltraMotionCapture.obj3d.search_nearest_point_idx

`UltraMotionCapture.obj3d.`**`search_nearest_point_idx`**(*point: numpy.array*, *target_points: numpy.array*) → int

Search the index of the nearest point from a collection of target points.

Parameters

- **`point`** – the source point coordinates stores in a (3,) `numpy.array`.
- **`target_points`** – the target points collection stores in a (N, 3) `numpy.array`.

Returns the index of the nearest point.

Return type `int`

3.1.6 UltraMotionCapture.obj4d

The 4D object contains a series of 3D objects (`UltraMotionCapture.obj3d`). In `UltraMotionCapture.obj4d`, 4D object classes with different features and capabilities are developed, serving for different analysis needs and scenarios. At current stage, there are 3 types of 4D object:

- Static 4D object `Obj4d`

  It contains a list of 3D objects.

- Static 4D object `Obj4d_Kps` with key points

  It's derived from `Obj4d` and attach key points (`UltraMotionCapture.kps.Kps`) to each of the 3D object via Vicon motion capture data (`UltraMotionCapture.kps.MarkerSet`).

- Dynamic/Deformable 4D object `Obj4d_Deform`

  It's derived from `Obj4d_Kps` and attach the rigid transformation (`UltraMotionCapture.field.Trans_Rigid`) and non-rigid deformation (`UltraMotionCapture.field.Trans_Nonrigid`) to each of the 3D object by registration.

Classes

`Obj4d`(start_time, fps)	Static 4D object.
`Obj4d_Deform`(enable_rigid, enable_nonrigid, . . .)	Dynamic/Deformable 4D object `Obj4d_Deform`.
`Obj4d_Kps`(markerset, **kwargs)	Static 4D object `Obj4d_Kps` with key points.

3.1.6.1 UltraMotionCapture.obj4d.Obj4d

class UltraMotionCapture.obj4d.**Obj4d**(*start_time: float = 0.0*, *fps: int = 120*)

Bases: `object`

Static 4D object. Contains a list of 3D objects.

Parameters

- **start_time** – the start time of the coordinates data.
- **fps** – the number of frames per second (fps).

Note:

self.start_time the start time of the coordinates data.

self.fps the number of frames per second (fps).

self.obj_ls a `list` of 3D objects.

Example

Use `load_obj_series()` to load a list of 3D objects and then add them to the 4D object:

```
import UltraMotionCapture as umc

o3_ls = umc.obj3d.load_obj_series(
    folder='data/6kmh_softbra_8markers_1/',
    start=0,
    end=1,
    sample_num=1000,
)

o4 = umc.obj4d.Obj4d()
o4.add_obj(*o3_ls)
```

`__init__`(*start_time: float = 0.0*, *fps: int = 120*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([start_time, fps])	Initialize self.
`add_obj`(*objs, **kwargs)	Add object(s).

`add_obj`(**objs: Iterable(Type[obj3d.Obj3d])*, ***kwargs*)
Add object(s).

Parameters **`*objs`** – unspecified number of 3D objects.

See also:

About the * symbol and its effect, please refer to *args and **kwargs - Python Tips

Example

Let's say we have two 3D objects `o3_a`, `o3_b` and 4D object `o4`. 3D objects can be passed into the *`add_obj()`* method one by one:

```
o4.add_obj(o3_a, o3_b)
```

3D objects can be passed as a list:

```
o3_ls = [o3_a, o3_b]
o4.add_obj(*o3_ls)
```

3.1.6.2 UltraMotionCapture.obj4d.Obj4d_Deform

class UltraMotionCapture.obj4d.**Obj4d_Deform**(*enable_rigid: bool = False*, *enable_nonrigid: bool = False*, ***kwargs*)

Bases: *UltraMotionCapture.obj4d.Obj4d_Kps*

Dynamic/Deformable 4D object *Obj4d_Deform*. Derived from *Obj4d_Kps* and attach the rigid transformation (*UltraMotionCapture.field.Trans_Rigid*) and non-rigid deformation (*UltraMotionCapture.field.Trans_Nonrigid*) to each of the 3D object by registration.

Parameters

- **enable_rigid** – whether enable rigid transformation registration ot not.
- **enable_nonrigid** – whether enable non-rigid transformation registration ot not.
- ****kwargs** – configuration parameters of the base classes (`Obj3d` and `Obj3d_Kps`) can be passed in via `**kwargs`.

Note:

self.enable_rigid whether enable rigid transformation registration ot not. Default as `False`.

self.enable_nonrigid whether enable non-rigid transformation registration ot not. Default as `False`.

Example

Load Vicon motion capture data (*UltraMotionCapture.kps.MarkerSet*) when initialising the 4D object. Use `load_obj_series()` to load a list of 3D objects. And then add them to the 4D object. In the procedure of adding, the program will implement the activated transformation estimation automatically:

```
import UltraMotionCapture as umc

o3_ls = umc.obj3d.load_obj_series(
    folder='data/6kmh_softbra_8markers_1/',
    start=0,
    end=1,
    sample_num=1000,
    obj_type=umc.obj3d.Obj3d_Deform
)

vicon = umc.kps.MarkerSet()
vicon.load_from_vicon('data/6kmh_softbra_8markers_1.csv')
vicon.interp_field()

o4 = umc.obj4d.Obj4d_Deform(
    markerset=vicon,
    fps=120,
    enable_rigid=True,
    enable_nonrigid=True,
)

o4.add_obj(*o3_ls)
```

`__init__`(*enable_rigid: bool = False*, *enable_nonrigid: bool = False*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([enable_rigid, enable_nonrigid])	Initialize self.
`add_obj`(*objs, **kwargs)	Add object(s) and attach key points (`UltraMotionCapture.kps.Kps`) to each of the 3D object via Vicon motion capture data (`markerset`).
`offset_rotate`()	Offset the rotation according to the estimated rigid transformation.

`add_obj`(**objs: Iterable[Type[obj3d.Obj3d_Deform]]*, ***kwargs*)
Add object(s) and attach key points (`UltraMotionCapture.kps.Kps`) to each of the 3D object via Vicon motion capture data (`markerset`). And then implement the activated transformation estimation.

Parameters

- **`*objs`** – unspecified number of 3D objects.

  > **Warning:** The 3D objects' class must derived from `UltraMotionCapture.obj3d.Obj3d_Deform`.

  See also:

  About the * symbol and its effect, please refer to *args and **kwargs - Python Tips

- **`**kwargs`** – configuration parameters for the registration and the configuration parameters of the base classes (`Obj3d` and `Obj3d_Kps`)'s `add_obj()` method can be passed in via `**kwargs`.

  See also:

  Technically, the configuration parameters for the registration are passed to `UltraMotionCapture.field.Trans_Rigid.regist()` for rigid transformation and `UltraMotionCapture.field.Trans_Nonrigid.regist()`, and they then call `probreg`'s registration method.

  For accepted parameters, please refer to probreg.cpd.registration_cpd.

Example

Let's say we have two 3D objects `o3_a`, `o3_b` and 4D object `o4`. 3D objects can be passed into the `add_obj()` method one by one:

```
o4.add_obj(o3_a, o3_b)
```

3D objects can be passed as a list:

```
o3_ls = [o3_a, o3_b]
o4.add_obj(*o3_ls)
```

__process_first_obj()

Process the first added 3D object.

> **Attention:** Called by *add_obj()*.

__process_rigid_dynamic(***kwargs*)

Estimate the rigid transformation of the added 3D object. The lastly added 3D object is used as source object and the newly added 3D object as the target object.

> **Attention:** The estimated transformation is load to the source object, via its *UltraMotionCapture.obj3d.Obj3d_Deform.set_trans_rigid()* method.

> **Attention:** Called by *add_obj()*.

__process_nonrigid_dynamic(***kwargs*)

Estimate the non-rigid transformation of the added 3D object. The lastly added 3D object is used as source object and the newly added 3D object as the target object.

> **Attention:** The estimated transformation is load to the source object, via its *UltraMotionCapture.obj3d.Obj3d_Deform.set_trans_nonrigid()* method.

> **Attention:** Called by *add_obj()*.

offset_rotate()

Offset the rotation according to the estimated rigid transformation.

> **Tip:** This method usually serves for reorientate all 3D objects to a referencing direction, since that the rigid transformation (*UltraMotionCapture.field.Trans_Rigid*) is estimated one follow one in the 3D objects list.

Example

Let's say we have an properly loaded 4D object `o4`, we'd like to view it before and after reorientated:

```
import copy

o4_offset = copy.deepcopy(o4)
o4_offset.offset_rotate()

o4.show()
o4_offset.show()
```

3.1.6.3 UltraMotionCapture.obj4d.Obj4d_Kps

class UltraMotionCapture.obj4d.**Obj4d_Kps**(*markerset: Optional[kps.MarkerSet] = None*, ***kwargs*)

Bases: *UltraMotionCapture.obj4d.Obj4d*

Static 4D object *Obj4d_Kps* with key points. Derived from *Obj4d* and attach key points (*UltraMotionCapture.kps.Kps*) to each of the 3D object via Vicon motion capture data (*UltraMotionCapture.kps.MarkerSet*).

Parameters

- **markerset** – Vicon motion capture data (*UltraMotionCapture.kps.MarkerSet*).
- ****kwargs** – configuration parameters of the base classes (Obj3d) can be passed in via **kwargs.

Note:

self.markerset Vicon motion capture data (*UltraMotionCapture.kps.MarkerSet*).

Example

Load Vicon motion capture data (*UltraMotionCapture.kps.MarkerSet*) when initialising the 4D object. And use load_obj_series() to load a list of 3D objects and add them to the 4D object:

```
import UltraMotionCapture as umc

o3_ls = umc.obj3d.load_obj_series(
    folder='data/6kmh_softbra_8markers_1/',
    start=0,
    end=1,
    sample_num=1000,
    obj_type=umc.obj3d.Obj3d_Kps
)

vicon = umc.kps.MarkerSet()
vicon.load_from_vicon('data/6kmh_softbra_8markers_1.csv')
vicon.interp_field()

o4 = umc.obj4d.Obj4d_Kps(
    markerset=vicon,
    fps=120,
)

o4.add_obj(*o3_ls)
```

__init__(*markerset: Optional[kps.MarkerSet] = None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([markerset])	Initialize self.
`add_obj`(*objs, **kwargs)	Add object(s) and attach key points (`UltraMotionCapture.kps.Kps`) to each of the 3D object via Vicon motion capture data (`markerset`).

add_obj(**objs: Iterable[Type[obj3d.Obj3d_Kps]], **kwargs*)

Add object(s) and attach key points (`UltraMotionCapture.kps.Kps`) to each of the 3D object via Vicon motion capture data (`markerset`).

Parameters

- **`*objs`** – unspecified number of 3D objects.

  > **Warning:** The 3D objects' class must derived from `UltraMotionCapture.obj3d.Obj3d_Kps`.

  See also:

  About the * symbol and its effect, please refer to *args and **kwargs - Python Tips

- **`**kwargs`** – configuration parameters of the base classes (`Obj3d`)'s `add_obj()` method can be passed in via `**kwargs`.

Example

Let's say we have two 3D objects `o3_a`, `o3_b` and 4D object `o4`. 3D objects can be passed into the `add_obj()` method one by one:

```
o4.add_obj(o3_a, o3_b)
```

3D objects can be passed as a list:

```
o3_ls = [o3_a, o3_b]
o4.add_obj(*o3_ls)
```

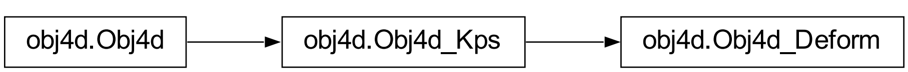

Fig. 4: Inheritance Relationship

3.1.7 UltraMotionCapture.utils

Functions

images_to_gif([path, remove])	Convert images in a folder into a gif.
obj_pick_points(filedir[, has_texture, ...])	Manually pick points from 3D mesh loaded from a `.obj` file.

3.1.7.1 UltraMotionCapture.utils.images_to_gif

`UltraMotionCapture.utils.`**`images_to_gif`**(*path: Optional[str] = None*, *remove: bool = False*)

Convert images in a folder into a gif.

Parameters

- **`path`** – the directory of the folder storing the images.
- **`remove`** – after generating the `.gif` image, whether remove the original static images or not.

Example

```
import UltraMotionCapture as umc
umc.utils.images_to_gif(path="output/", remove=True)
```

3.1.7.2 UltraMotionCapture.utils.obj_pick_points

`UltraMotionCapture.utils.`**`obj_pick_points`**(*filedir: str*, *has_texture: bool = False*, *save_folder: str = 'output/'*, *save_name: str = 'points'*)

Manually pick points from 3D mesh loaded from a `.obj` file. The picked points are stored in a (N, 3) `numpy.array` and saved as `.npy` `numpy` binary file.

Parameters

- **`filedir`** – The directory of the `.obj` file.
- **`has_texture`** – Whether the `.obj` file has texture file or not. If set as `True`, the texture will be loaded and rendered.
- **`save_folder`** – The folder for saving `.npy` binary file.

> **Attention:** The folder directory shall be ended with `/`, e.g. `output/`.

- **`save_name`** – The name of the saved `.npy` binary file.

Example

One application of this function is preparing data for **calibration between 3dMD scanning system and the Vicon motion capture system**: Firstly, we acquire the markers' coordinates from the 3dMD scanning image. Then it can be compared with the Vicon data, leading to the reveal of the transformation parameters between two system's coordinates.

```
import UltraMotionCapture as umc
umc.utils.obj_pick_points(
    filedir='dataset/6kmh_softbra_8markers_1/speed_6km_soft_bra.000001.obj'
    has_texture=True,
    save_folder='conf/calibrate/',
    save_name='points_3dmd',
)
```

Dragging the scene to adjust perspective and clicking the marker points in the scene. Press `q` to quite the interactive window and then the picked point's coordinates will be stored in a (N, 3) `numpy.array` and saved as `conf/calibrate/points_3dmd.npy`. Terminal will also print the saved `numpy.array` for your reference.

> The remaining procedure to completed the calibration is realised in the following Jupyter notebook script:
>
> `config/calibrate/calibrate_vicon_3dmd.ipynb`

See also:

About the `.npy` `numpy` binary file: numpy.save numpy.load

About point picking feature provided by the `pyvista` package: Picking a Point on the Surface of a Mesh - PyVista

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

u

INDEX

Symbols

A

F

G

I

K

L