Agiprobot Measurement Documentation

Release 1.0

Sven Becker

Apr 27, 2021

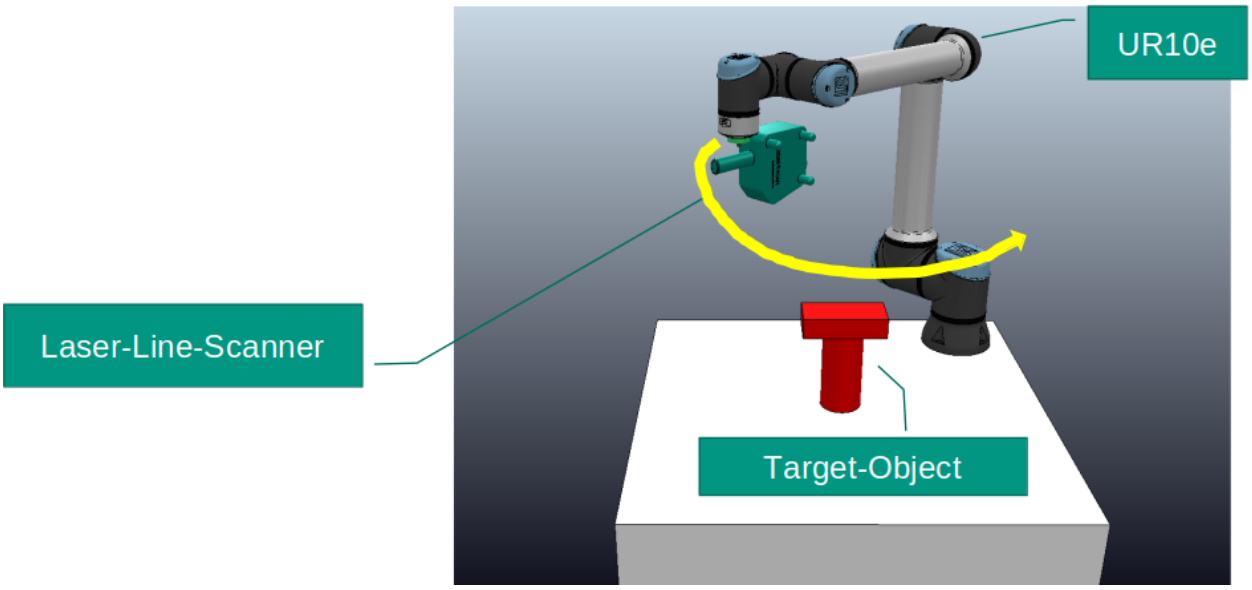
EXPLANATION

CHAPTER

ONE

OVERVIEW

This package provides a pipeline to plan, execute and handle a measurement-trajectory for a given CAD-object.



The important module is *Trajectory Manager*. It performs the motion planning pipeline with assistance by using the other modules *ViewPoint* and *Sensor Model*.

During planning, *Trajectory Manager* samples the surface of the target-mesh into discrete surface points and generates straight measurement-trajectories based on those points. Starting at each surface point, the program places the *anchor* of a possible trajectory above this sample point and then processes the corresponding straight trajectory metrologically and mechanically. To do so, the *Sensor Model* is used to determine which other sample-points would be visible during this trajectory and how their uncertainties are. MoveIt is used to review, if and how the trajectory-candidate is actually executable in the modeled scene (collision-awareness, reachablility, ...). Both information - the sensor-evaluation and the kinematics - are then stored into a *ViewPoint*-object. An algorithm is used to determine from the set of all viewpoints one subset, that covers all sampled surface points with an adjustable objective (like "minimize amount of viewpoints"), i.e. solves the Set Covering Problem. Afterwards, the chosen *ViewPoint*-objects are connected in a time-optimal way and the combined trajectory by connecting all viewpoint-trajectories and their in-between-segments can be executed or stored for later execution.

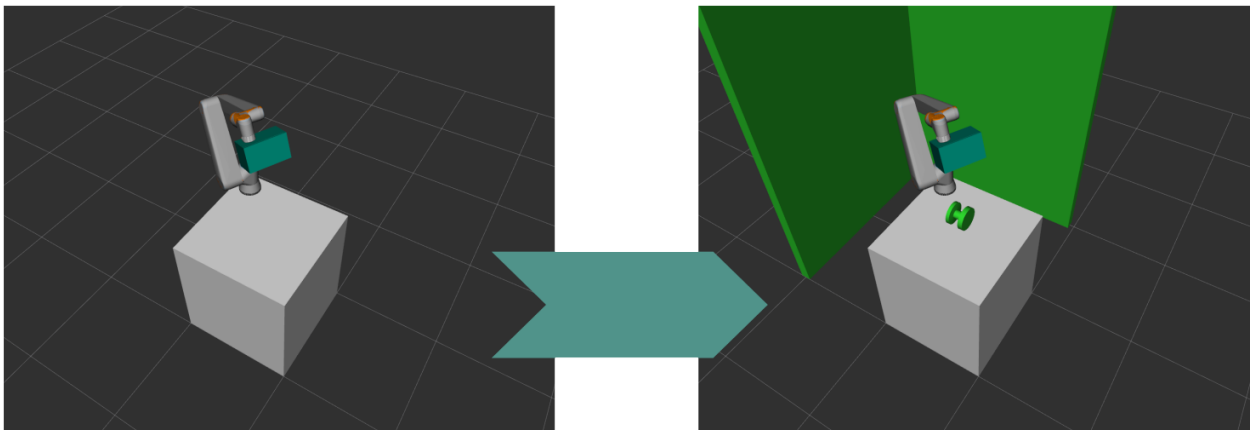
CHAPTER

TWO

PATH PLANNING DETAILED

2.1 0. Load the Task

Read in the given CAD-file of the target-object as well as its pose. The mesh is made available to the local trimesh-handler and loaded into the motion-planning-framework MoveIt so that it will be considered for collision-checks.



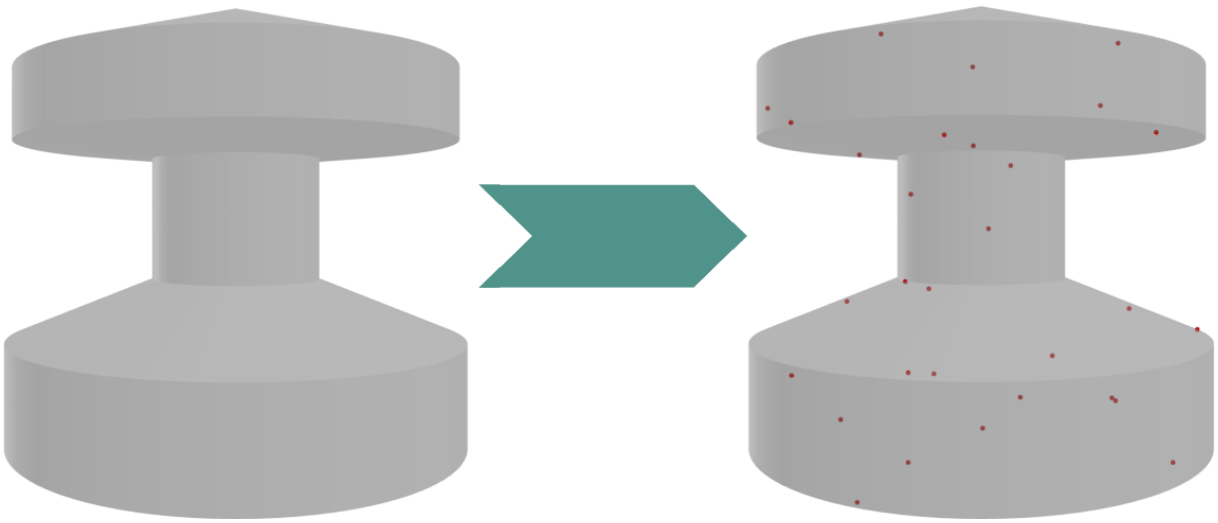
2.2 1. Preprocessing and Sampling

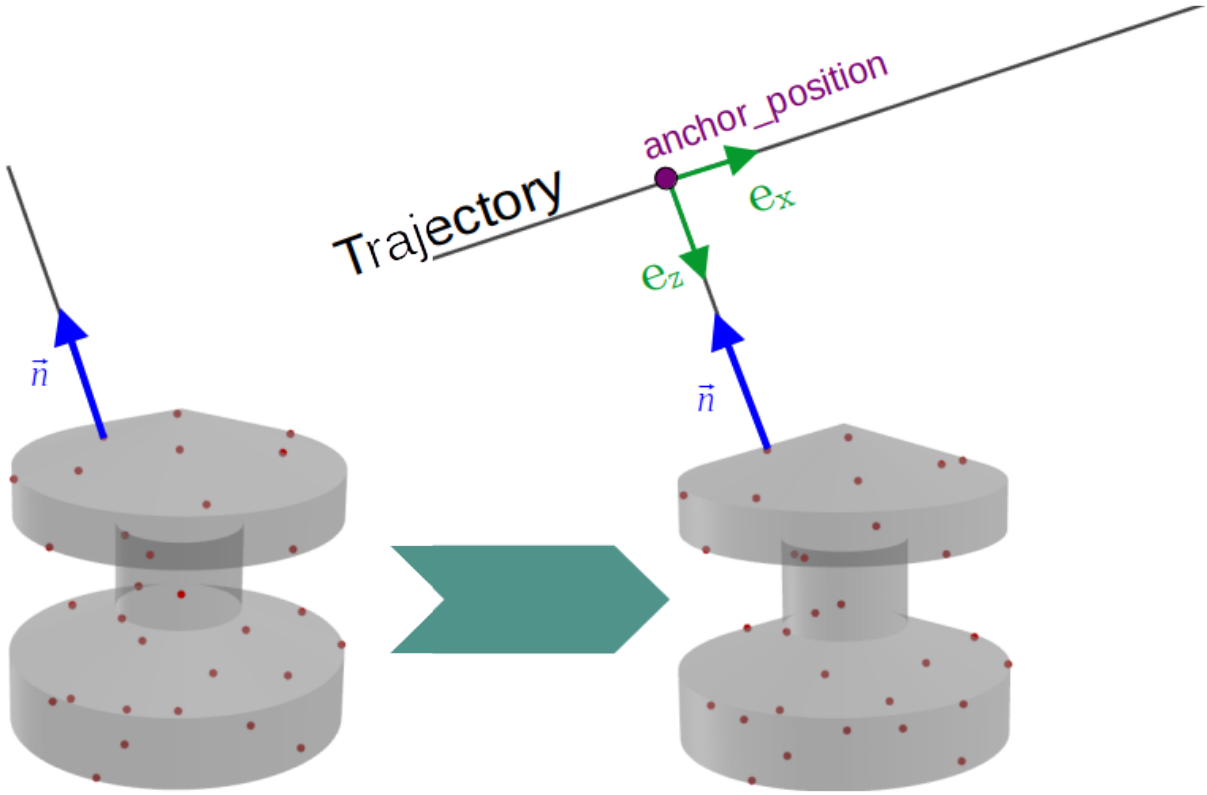
Removes all downward facing surface faces since they are not considered measurable. This is based on the common assumption that the fixture holds the target object from below. Use trimesh to sample the mesh's surface into discrete surface-points depending on a specified sampling-density. The samples, corresponding face-normals and the target-mesh are set as context in a *Sensor Model*-instance.

2.3 2. ViewPoint-Generation and -Evaluation

For each-sample, generate *ViewPoint*-objects by moving into the points face-normal and then applying rotations to get a variety of candidates per sampled surface point. The resulting point is the anchor of the viewpoint. Each viewpoint is assigned a straight trajectory-line.

After that, 2 main evaluations are executed:




Trajectory
anchor_position
e_x
e_z
$\vec{n}$
$\vec{n}$

2.3.1 Metrological Evaluation

- Use the *Sensor Model*-instance to analyze, which other sample-points are visible from this trajectory, and at what uncertainty
- Result = Visibilities and uncertainties (will be stored in the *ViewPoint*-object)

2.3.2 Mechanical Evaluation

- Utilize MoveIt to examine, if this trajectory is actually executable in the scene (collision, reacability, ...).
- **Critical**: If the mechanical evaluation fails, the viewpoint will not be considered any further
- Can also consider trajectory-parts: If e.g. 80% of the trajectory are be executable, this *ViewPoint*-object might not be rejected
- Result = List of actual joint-values (will be stored in the *ViewPoint*-object)

The sequence of these evaluations is implemented in a way to minimize compute-time.

2.4 3. ViewPoint-Selection (Set Covering Problem)

When a set of valid *ViewPoint* s has been found, not all *ViewPoint* -elements must be executed for a valid result: A subset must be found that covers the same surface-points as all the found *ViewPoint* s (= *Set Covering Problem*). This subset is in general much smaller than the original set and can be built using different algorithms:

As the first option, 'Greedy' implementation (both used in the papers) selects in each iteration that new *ViewPoint* from the original set, that can contribute the most not-yet-visible surface-sample-points to the subset. If 2 or more *ViewPoint* -objects can contribute the same amount, the one with the lowest uncertainty will be selected.

Another option is Integer Programming (IP):

Minimize $\sum_{\forall i} c_j v_j$ subject to $\sum_{\forall j} m_{i,j} v_j \geq 1 \quad \forall i$

where $v_j \in \{0, 1\}$ indicates if viewpoint j is element of the subset and $m_{i,j} \in \{0, 1\}$, if the sampled surface point i is measurable by the trajectory of viewpoint j. c_j is a cost-term. If it is constant, the IP-problems is identically to the Greedy approach. However, by assigning c_i the time of the measurement-trajectory of viewpoint i or its uncertainty, a bigger variety of solutions becomes possible than in the first option.

2.5 4. Connecting the ViewPoints

Lastly, the determined set of *ViewPoint*-objects must be ordered for optimal execution. To do so, the current robot-pose is enqueued in an 'execution-list'. Paths from the endpoint of the last element of execution-list are calculated to every non-enqueued viewpoint's trajectory-start- and -endpoint. The *ViewPoint* with the shortest path-time becomes enqueued. Also, the inter-viewpoint path will be stored in that *ViewPoint* so that it will perform the exact same path during execution (if the path would get planned again dynamically, it might be completely different due to the stochastic nature of path-planning). This step relies again on MoveIt.

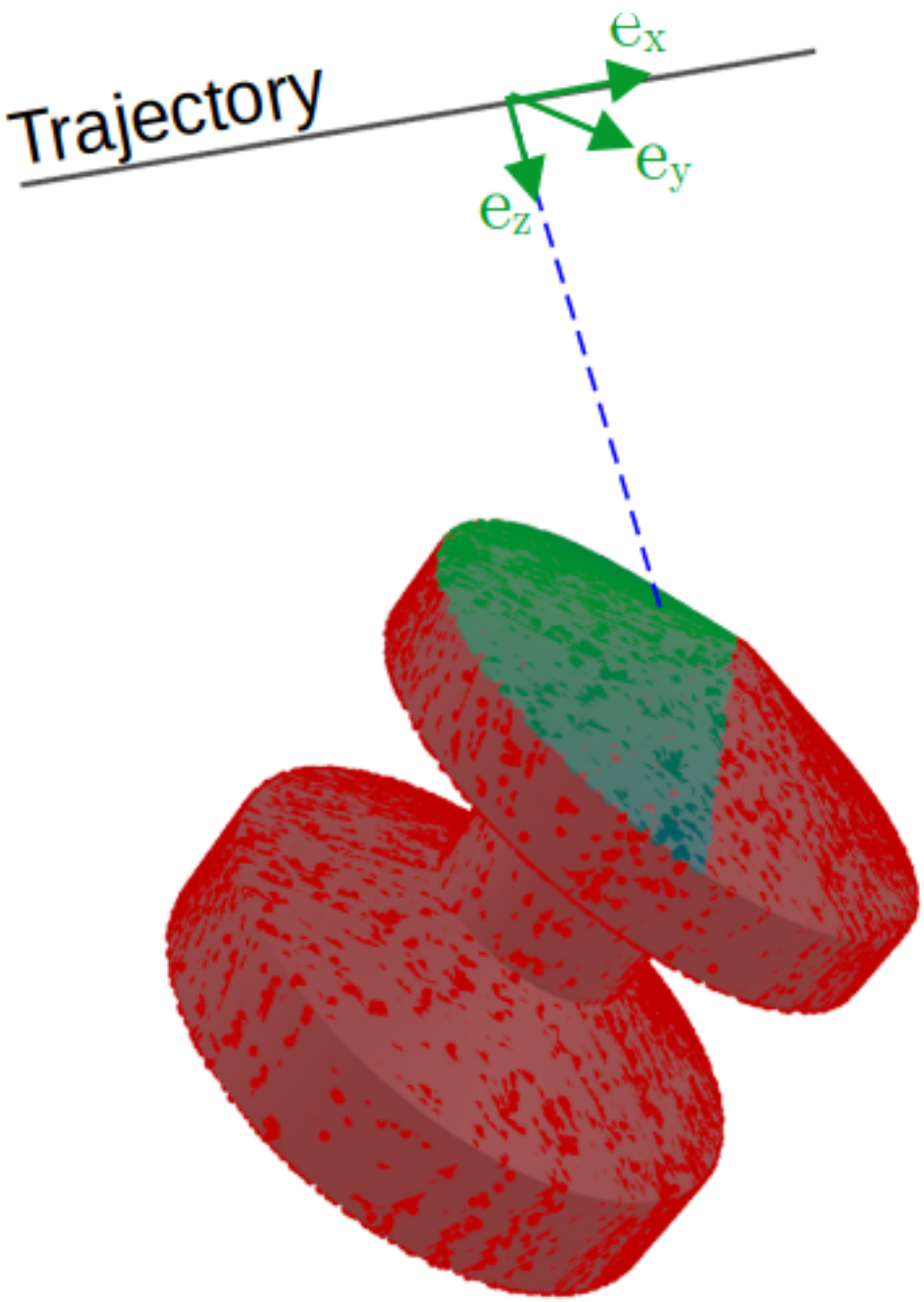
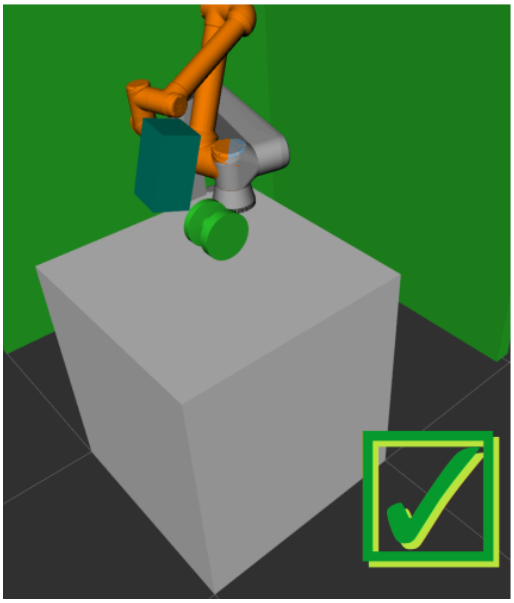
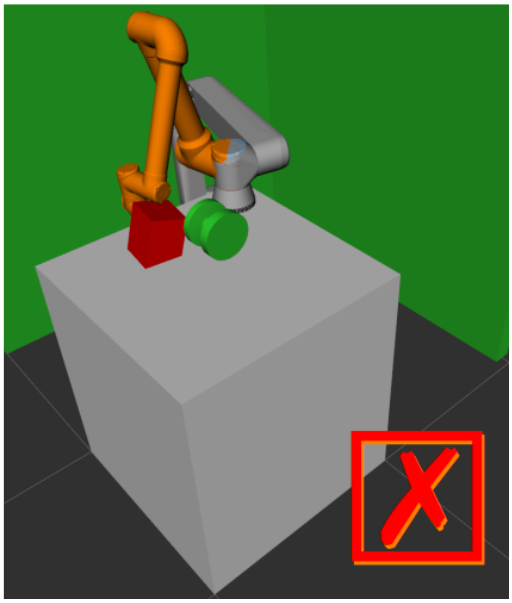


Fig. 2.1: Higher sampling density than previously for vividness. Red points are immeasurable; Green points are measurable at low uncertainty, blue ones at high uncertainty.





CHAPTER

THREE

TRAJECTORY MANAGER

3.1 Introduction and Basic Ideas

This module provides an configurable system to plan, execute, store and load measurement-trajectories for scanning objects with a laser-line-scanner mounted on a robotic arm.

The planning is heavily inspired by "View and sensor planning for multi-sensor surface inspection" (Gronle et al., 2016) and "Model-based view planning" (Scott, 2009), however this module was adapted to fit this use-case and general improvements were incorporated.

For a given task, the user can specifiy the mesh-to-measure, properties of the trajectory-segments (like lenght) and other parameters. Then, the planning-pipeline discussed in section *Path Planning Detailed*, will be performed. Afterwards, the generated total trajectory can be executed via MoveIt on the real robot or in a simulation (see CoppeliaSim-Interface's documentation in that case). The trajectory can also be written to disk for later or repetitive execution. As 'good' trajectories currently require long calculations (>30min), this module enables users to load and execute previously computed trajectories.

3.2 API-Specification

class agiprobot_measurement.trajectory_manager.**TrajectoryManager**

Completely integrated system for high-coverage and uncertainty-minimal scanning of given objects with an optical scanner. An instance can perform the entire planning pipeline - from the CAD-file and pose to a list of consecutively executable trajectories.

connect_viewpoints(*unordered_viewpoints*, *min_planning_time=0.2*)

Connect a set of unordered viewpoints with the current state and in between greedily so that they can be executed as fast as possible. Until all viewpoints are enqueued, do: From the end-point of the last enqueued trajetory, motion plans are calculated to the start-/end-poses of all unenqueued viewpoint's measurement-trajectories and the shortest (in time domain) will be selected.

Parameters

- **unordered_viewpoints** (*set[viewpoint]*) – Set of viewpoints to be connected where each has a stored measurement-trajectory
- **min_planning_time** (*float, optional*) – Planning time that is used for connection-path-planning (will be increased automatically if no plan was found at first), defaults to 0.2

Returns List of ordered and execution-ready viewpoints

Return type list[*ViewPoint*]

convert_viewpointlist_to_execution_plan(*viewpointlist*)

Convert a list of ViewPoint-objects into a list of moveit_msgs/RobotTrajectory-entries so that the viewpoints can be executed consecutively.

Parameters **viewpointlist** – List of ViewPoint-objects with set trajectories for steering-to-viewpoint and measurement

Returns List of moveit_msgs/RobotTrajectory with 2 entries per provided ViewPoint-object (first = steering-to-viewpoint-trajectory, second = measurement-trajectory)

Return type list

execute(*execution_list*, *surface_points=None*)

Execute a list of RobotTrajectories or ViewPoints via MoveIt. When problems occur during execution, the robot will be stopped and an exception will be raised. When executing from a list of viewpoints, the currently measured surface_points are published in "/currently_measured_points" during execution.

Parameters

- **execution_list** (*list[moveit_msgs/RobotTrajectory] or list[ViewPoint]*) – List of RobotTrajectories or viewpoints that can be executed consecutively (the next segment's start point is the last segment's end point)
- **surface_points** (*list[numpy.array], optional*) – List of the actual sampled surface points, defaults to None

generate_samples_and_viewpoints(*sampling_density*, *uncertainty_threshold*, *orientations_around_boresight*, *viewpoint_tilt_mode='full'*, *plan_path_to_check_reachability=False*, *minimum_trajectory_length=25*, *trajectory_sample_step=2*)

Samples the mesh's surface into discrete points, generates viewpoints for each sample and processes these viewpoints metrologically and mechanically. Only viewpoints that do meet the contraints specified in the method's parameters and in MoveIt (collision, reachability, ...) will be returned. This method allows to generate multiple ViewPoint-objects for a single surface-point by varying the anchor-pose (the boresight is always focused on the corresponding surface_point).

Parameters

- **sampling_density** (*float*) – Density of sampling to generate the points for measurement-calculations and viewpoint-generation in points per mm^2
- **orientations_around_boresight** (*int*) – Number of orientations around the sensor's boresight to be considered per sampled_surface_point
- **viewpoint_tilt_mode** (*str, optional*) – For each orientation, deviate the psi- and theta-values of the viewpoint-anchor-pose slightly from the optimum according to (one deviation-step per angle and orientation available right now):
  - "none": Do not perform any tilting.
  - "limited": When the optimal angle-configuration did not work, try deviations and stop after finding the first valid solution.
  - "full": Calculation for every possible angle-calculation. Every boresight orientation has 9 sub-viewpoints.

  , defaults to "full"
- **plan_path_to_check_reachability** (*bool, optional*) – Do not use one inverse-kinematics-request to check viewpoint-reachability but try to create complete plan from the current-state to the viewpoint's anchor-pose, defaults to False

- **minimum_trajectory_length** (*float, optional*) – Minimum length a generated trajectory has to have so that it is accepted in mm, defaults to 50
- **trajectory_sample_step** (*float, optional*) – Euclidian distance between 2 trajectory-points in cartesian sapce in mm, defaults to 2

Returns 2 lists with the first containing all sampled_surface_points and the second containing all viewpoints that were processed successfully

Return type list[numpy.array], list[*ViewPoint*]

load_execution_plan (*file_path*, *adapt_to_current_start_pose=True*)

Extracts a list of moveit_msgs/RobotTrajectory specified in the yaml-file at the provided path.

The trajectories can be executed consecutively, however if the execution should start from the current state (which is most likely the case), the parameter 'adapt_to_current_start_pose' should be set true to add movement from the current point (which in general deviates from the 'current point' during planning) to the trajectory start. The yaml-file must correspond to a list with the first entry being a dictionary for metadata and the following entries being dictionaries of 'expanded' RobotTrajectories with following structure: Trajecories connecting measurements (C) and measurement trajectories (M) themselves in an alternating fashion (C-M-C-M-C-...).

Parameters

- **file_path** (*str*) – Where the yaml-file is located
- **adapt_to_current_start_pose** (*bool*) – If true (default), the plan will be adapted so that it can start from the current state

Returns List of moveit_msgs/RobotTrajectory-ies that can be consecutively executed

Return type list

load_target_mesh (*file_name, transform=array([[1., 0., 0., 0.], [0., 1., 0., 0.], [0., 0., 1., 0.], [0., 0., 0., 1.]]), add_wbk_mirrors=True, remove_downside=True, remove_threshold_angle_deg=20*)

Loads the mesh-to-measure from file into the instance's target_mesh-member (to generate and evaluate viewpoint-trajectories) and into MoveIt (for collision-prevention). To move the mesh into a feasible measurement-pose, a transform may be applied to set the mesh's reference frame specified in CAD with respect to the 'world' frame

Parameters

- **file_name** (*str*) – Where the target_mesh's CAD file is located (should be given starting at root-level: '/X/Y/...')
- **transform** (*numpy.array, optional*) – Homogeneous 4x4-matrix to move the mesh into the desired pose with translation in mm, defaults to identity-matrix
- **add_wbk_mirrors** (*bool, optional*) – Adds hard-coded collision objects to MoveIt as the robot station at the institute is placed besides 2 fragile mirrors, defaults to True

Returns Whether all operations (esp. loading from file into trimesh and MoveIt) were successful

Return type bool

postprocess_trajectory (*trajectory*)

Check if trajectory meets the limits specified in 'joint_limits.yaml' and if the time between two points is increasing. MoveIt does not apply the values from 'joint_limits.yaml' when computing a cartesian path, which is dangerous for trajectory-execution. For prevention, trajectories with too heavy violations will be rejected. However, specifying the joint-limits from the yaml in the xacro-urdf as well seems to have eliminated this problem (but this code is still active for safety purposes).

Parameters **trajectory** (*moveit_msgs/RobotTrajectory*) – RobotTrajectory to check and improve

Returns Boolean value indicating if the trajectory could be post-processed reasonably or if it has to be rejected

Return type bool

process_viewpoint(*viewpoint*, *ik_service*, *uncertainty_threshold*, *minimum_required_overmeasure=5*, *trajectory_sample_step=2*, *joint_jump_threshold=1.5*, *minimum_trajectory_length=50*)

Processes a single viewpoint entirely. Starting on a simple reachability-analysis of the anchor-pose, a metrological evaluation is performed using the sensor_model to reviewpoint the measurement-gain this viewpoint can contribute in theory. Then, the actual meausrement-trajectories are calculated and examined regarding the provided contraints. In the end, the processing-result (trajectory and metrological values) are stored in the viewpoint object

Parameters

- **viewpoint** (ViewPoint) – ViewPoint to be processed (if processing was successful, members of this viewpoint-object will be changed)
- **ik_service** (*rospy.ServiceProxy*) – ROS-Serviceproxy that can resolve inverse-kinematics via moveit_msgs/GetPositionIK
- **minimum_required_overmeasure** (*float, optional*) – How much to add to the trajectory in trajectory-direction in mm after the last measurable sample_point is measured (as samples usually do not occur exactly at edges), defaults to 5
- **trajectory_sample_step** (*float, optional*) – Euclidian distance between 2 trajectory-points in cartesian sapce in mm, defaults to 2
- **joint_jump_threshold** (*float, optional*) – Maximum change in joint values allowed between 2 trajectory-points in rad (since all joints are revolute), defaults to 1.5
- **minimum_trajectory_length** (*float, optional*) – Minimum length a generated trajectory has to have so that it is accepted in mm, defaults to 50

Returns Boolean value that is true, if the viewpoint could be processed and all contraints were met, and false otherwise

Return type bool

solve_scp(*provided_viewpoints*, *solver_type='greedy'*)

Solve Set Covering Problem to cover all measurable surface_points with a fraction of the set of provided viewpoints. Possible solver_types are:

- "greedy": Fills return set at each step with the trajectory that delivers the most additional coverage compared to the points already in the set. If this additional coverage is identical in size for several new optimal viewpointpoint-candidates, the one with the lowest maximum uncertainty will be added similarly to the improvement introduced in chapter 4.4 of "ViewPoint and sensor planning for multi-sensor surface inspection" (Gronle et al., 2016) (default)
- "IP_basic": Solves the SCP-Problem with integer programming (IP) using the formulation in "Model-based viewpoint planning" (Scott, 2009), i.e. the objective function is the number of all selected viewpointpoints whereas the constraint is that every surface point must be covered at least by one viewpointpoint
- "IP_uncertainty": Solves the SCP using IP with respect to the uncertainty. The formulas are similar to "IP_basic" but in the objective, costs corresponding to the worst uncertainty are assigned to every viewpointpoint-trajectory.

- "IP_time": Solves the SCP using IP with respect to the time of trajectory-execution. The formulas are similar to "IP_basic" but in the objective, costs corresponding to the duration of the trajectory are assigned to every viewpointpoint.

  Parameters

  - **provided_viewpoints** (*set[ViewPoint] or list[ViewPoint]*) – All processed viewpoints where each has a valid measurement-trajectory and information about the measurable surface_points
  - **solver_type** (*str, optional*) – See function description, defaults to "greedy"

  Returns Set of viewpoints that can measure the union of measurable surface points of all provided viewpoints

  Return type set[*ViewPoint*]

store_execution_plan (*file_path*, *execution_plan*, *metadata={}*)

Stores the provided execution_plan segments (= moveit_msgs/RobotTrajectory) in a yaml-file at the provided file_path formatted so that it can be read in by the class's 'load_execution_plan'-method.

Parameters

- **file_path** (*str*) – Path specifying where to safe the generated file at
- **execution_plan** (*list of moveit_msgs/RobotTrajectory*) – List of path-segments that can be executed consecutively (i.e. joint-values at the last point of any entry = joint-values at the beginning of the next entry).
- **metadata** (*dict*) – Dictionary of information about this plan in key-value-fashion (e.g. {'sampling_density': 0.01, 'timestamp': '12:00, 01.01.2021', 'planning_duration_in_s':42}")#

Returns True (always)

Return type bool

CHAPTER

FOUR

SENSOR MODEL

4.1 Introduction and Basic Ideas

The Sensor Model is a utility-module to perform metrological evaluations on a given input.

The mathematical foundation of the model is derived in the corresponding Bachelor's Thesis, however the theoretical understanding is not needed to work with this module. One task is to make qualified predictions about the uncertainty of a given point within the scanners visibility, when certain spatial information (see below) is given. Within the *Trajectory Manager*'s pipeline, this module can also be used to determine which points can be measured and how 'uncertain' this measure for a *ViewPoint*-object's trajectory given the occluding target-mesh, possible sampled surface points, ... ('context').

4.1.1 Commonly Used Geometric Terms

See the image below for additional context.

- laser_emitter_frame: See *ViewPoint*
- psi/ψ: Angle between the laser_emitter_frame's z-y-plane and the surface-normal of the sample
- theta/ϑ: Angle between the in normal-vector and the laser-emitter-ray-direction towards the sample, projected on the y-z-plane
- z: Distance of the sample-point projected onto the z-axis of the laser_emitter_frame = Distance from the x-y-plane of the laser_emitter_frame
- α, K_u, u_0: Symbols of the uncertainty-formular (not important to use the sensor-model, detailed description in accompanying Bachelor's Thesis)

4.1.2 Visualization of Concepts

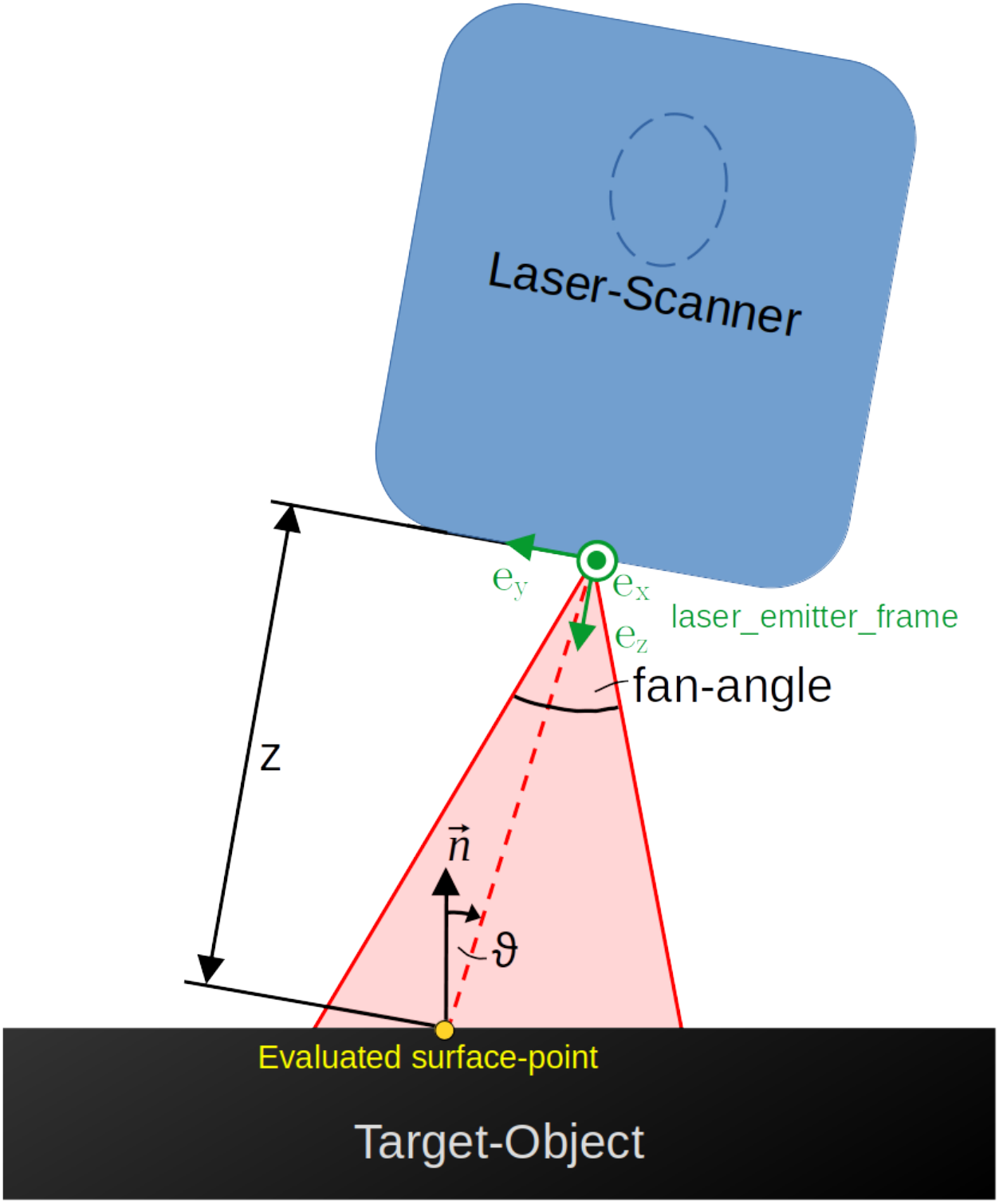
Mind the orientation of the green laser_emitter_frame!

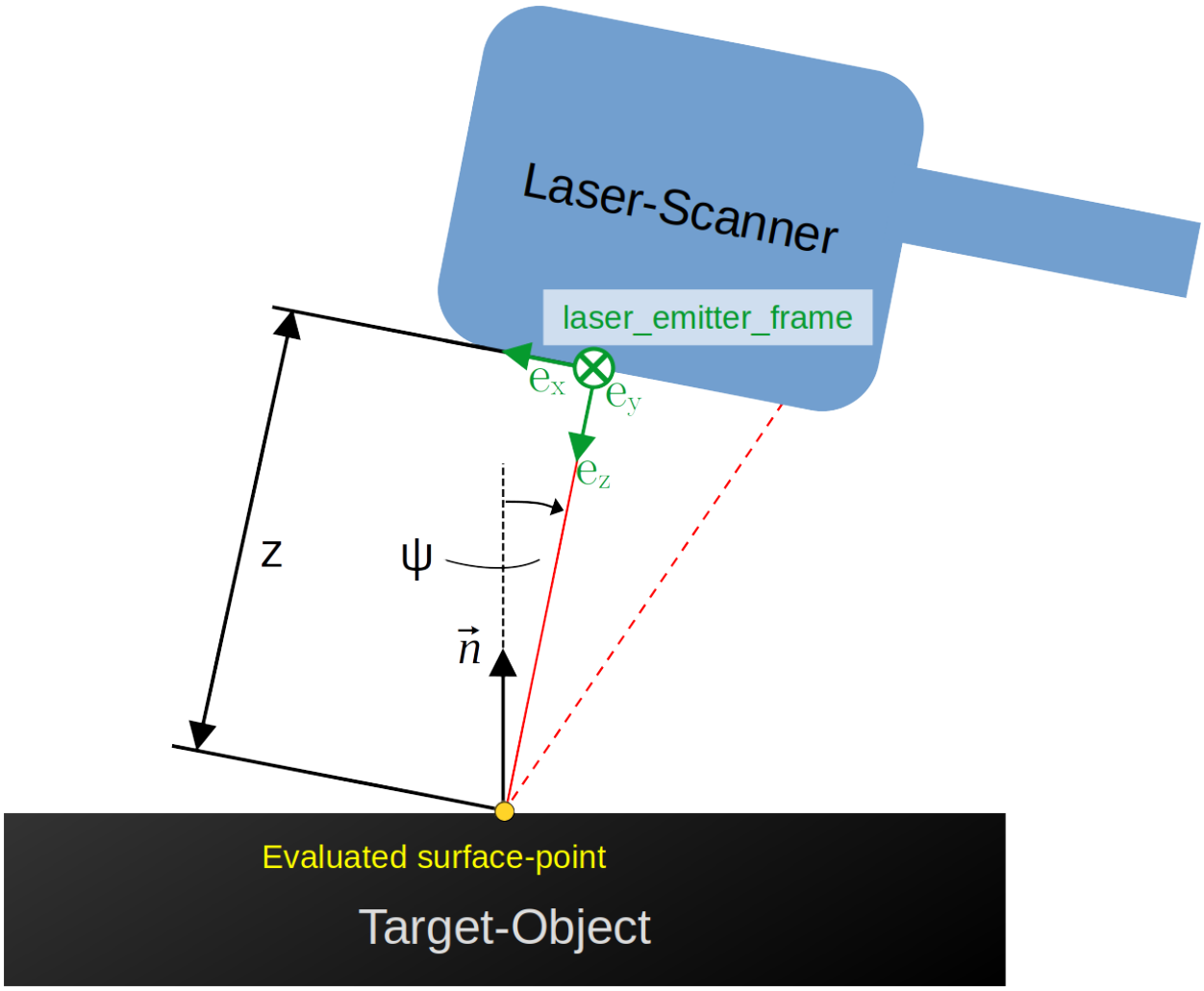
Perpendicular To Laser-Fan

Parallel To Laser-Fan

4.2 API-Specification

class agiprobot_measurement.sensor_model.**SensorModel**(*parameter_map*)
Computional representation of an laser triangulation sensor under influence of uncertainty.


Laser-Scanner
e_y
e_x
e_z
laser_emitter_frame
fan-angle
z
$\vec{n}$
ϑ
Evaluated surface-point
Target-Object


Laser-Scanner
laser_emitter_frame
e_x
e_y
e_z
z
ψ
$\vec{n}$
Evaluated surface-point
Target-Object

evaluate_score(*z*, *psi*)
Calculates a score of uncertainty $\in [0, 1]$ for the given z and psi (higher score means lower uncertainty). The score is linear affine in the uncertainty based on z and psi - It is 1 if the provided values match the best possible case (lowest possible uncertainty of the sensor model) and 0 when both values are right at the rejection limit (i.e. are equal to **max_deviation_**...): $\frac{u_{max} - u(z,\psi)}{u_{max} - u_{min}}$

Parameters

- **z** (*float*) – z-coordinate of the point to evaluate in the laser_emitter_frame in mm
- **psi** (*float*) – Angle of the laser_emitter_frame's z-y-plane with the surface-triangle of the point to evaluate in rad

Returns Score $\in [0, 1]$ based on the parameters and the max_deviations of them

Return type float

evaluate_uncertainty(*z*, *psi*)
Computes the uncertainty of a surface point with given z and psi using the specified geometric sensor parameters.

Parameters

- **z** (*float*) – z-coordinate of the point to evaluate in the laser_emitter_frame in mm
- **psi** (*float*) – Angle of the laser_emitter_frame's z-y-plane with the surface-triangle of the point to evaluate in rad

Returns Uncertainty based on the z, psi, and the geometric sensor parameters, or 'NaN' if z and psi are invalid

Return type float

get_max_uncertainty()
Computes maximum possible uncertainty of a measured point that is not rejected using geometric sensor parameters. The uncertainty-formula is evaluated at z- and psi-values that are at rejection-limits for a surface point, so that this value corresponds to the worst uncertainty assignable to a measurable point.

Returns Maximum possible uncertainty of a measurable point (= is within the **max_deviation_**...-range) in mm

Return type float

get_median_deviation_angle()
Get the smallest angle allowed for tilting (smallest of maximum deviations of theta and psi devided by 2).

Returns Smallest angle allowed for tilting

Return type float

get_min_uncertainty()
Computes the minimum possible uncertainty using the geometric sensor parameters. It returns the uncertainty value assignable to a surface point which has been measured with optimal z- and psi-values, i.e. the best-case.

Returns Minimum possible uncertainty in mm

Return type float

get_optimal_standoff()
Gets the optimal standoff, i.e. required the . The returned optimal standoff is not equal to the standoff providing the lowest uncertainty, but in the center between the z-rejection-limits to allow maximum flexibility in surface height deviation when moving the sensor.

Returns z-coordinate in the laser_emitter_frame of an surface point for measurement in mm

Return type float

get_scanning_frustum(*half_length*)

Generates a trimesh-mesh object representing the 'frustum of measurability' in the laser_emitter_frame. Points within this frustum have the potential to be measurable, but **maximum_deviation_**... may still reject points within this frustum. The frustum is constructed based on the laser-fan-angle and the allowed z-range and the idea that the trajectory is a straight line (it can be imagined as a cut-off 'Toblerone'). So points lying wthin this frustum can be touched by the laser line AND can be mapped to the optical sensor.

Parameters **half_length** (*float*) – Length of the frustum perpendicular to the fan-plane in each direction of the laser_emitter_frame in mm

Returns Scan frustum represting the volume of potentially measurable points

Return type trimesh.Trimesh

process_viewpoint_metrologically(*viewpoint*, *uncertainty_threshold*, *maximum_deflection=500.0*)

Computes all measurable surface-points by a viewpoint-object as well as uncertainties and where they are measurable on the viewpoint-measurement-trajectory. Requires context to be set via set_processing_context(...). Checks for every sampled surface point of the given context whether it is visible and calculates the uncertainty for it eventually. Also, the deflection of the laser_emitter_frame along the trajectory-line from the viewpoint-anchor pose is evaluated.

Parameters

- **viewpoint** (ViewPoint) – ViewPoint with set viewpoint-anchor-pose
- **uncertainty_threshold** (*float*) – Maximum permissible uncertainty of a measured surface point, in mm
- **maximum_deflection** (*float, optional*) – Maximum deflection of the trajectory to be considered for processing in mm, defaults to 5e2

Returns

3 unpacked arrays of the same length in order:

- Indices of the measurable surface points in samples_surface_points_list
- Corresponding uncertainty-scores
- Metric distance in mm in trajectory-direction from the viewpoint-anchor where the corresponding surface point is measurable

Return type array[int], array[float], array[float]

set_processing_context(*mesh*, *sampled_surface_points*, *sampled_face_indices*)

Loads a context for metrological processing to the sensor model. This includes loading the target mesh into a ray-tracer as well as gaining awareness over the sampling results because they are directly used to assess a ViewPoint during process_viewpoint_metrologically(). This method must be called before any metrological processing can be performed. A previously set context becomes overwritten completely by calling this method again.

Parameters

- **mesh** (*trimesh.Trimesh*) – Mesh-object to load into the RayMeshIntersector
- **sampled_surface_points** (*list[numpy.array]*) – List of all sampled surface points that should be considered for the metrological processing
- **sampled_face_ids** (*list[int]*) – List of face-indices, where each entry corresponds to the face of the sampled surface point at the same position

CHAPTER

FIVE

VIEWPOINT

5.1 Introduction and Basic Ideas

A ViewPoint is a container of one measurement-trajectory with extra-functionality.

It can store besides the actual measurement-trajectory another trajectory towards its measurement trajectory as well as quantitative metrological predictions about the measurement. A View can construct itself in a bare geometrical sense: Given a surface point, a anchor-position as well as an orientation of the laser_emitter_frame are created so that the z-axis of this frame points towards this surface_point, when in anchor-position.

5.1.1 Commonly Used Terms

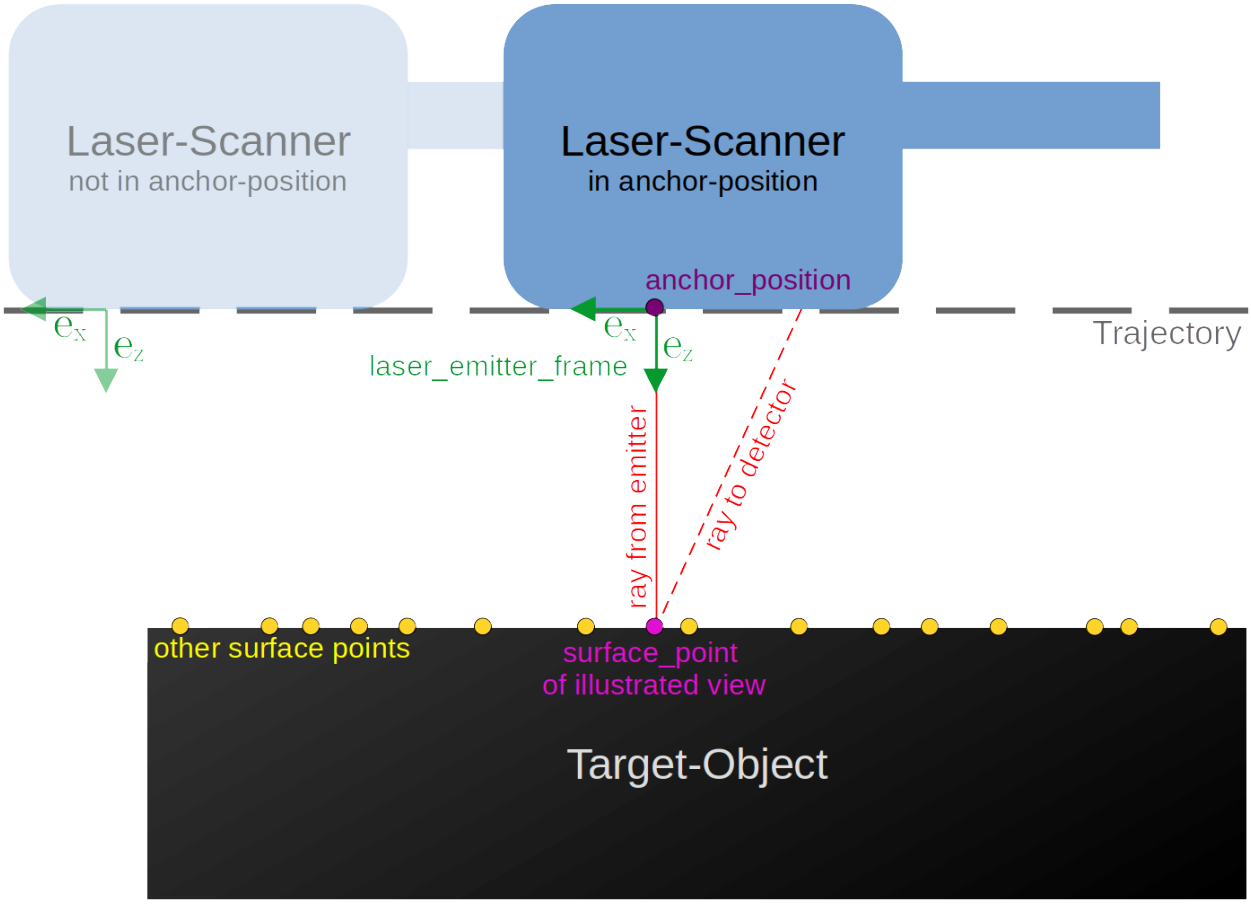
- surface-point: The sampled surface point on the target-mesh this viewpoint is based on
- anchor-position: The position this viewpoint's trajectory goes through and where its laser_emitter_frame's z-axis points towards the corresponding surface-point
- **laser_emitter_frame: Frame positioned at the scanners's laser-emitter ("where the laser leaves the scanner") that is move**
  - x-axis points in the trajectory-direction
  - z-axis points towards the corresponding surface-point when it becomes measured
  - y-axis results from the right-handedness of the coordinate-system
  - During trajectory-execution, the orientation of this frame is constant in the 'world'-frame but the position changes
- angle-around-boresight: Angle for rotation of this viewpoints straight-trajectory around the laser_emitter_frame's z-axis in anchor-position

5.1.2 Visualization of Concepts

5.2 API-Specification

class agiprobot_measurement.viewpoint.**ViewPoint**(*surface_point*, *surface_normal*, *angle_around_boresight*, *stand-off_distance*, *tilt_theta=0*, *tilt_psi=0*)

Class for keeping track about one viewpoint. A viewpoint consists on the one hand of geometrical specifications (about location and length of the viewpointand execution informations), on the other hand of metrological specifications (e.g. what can be measured by this viewpoint).


Laser-Scanner
not in anchor-position
Laser-Scanner
in anchor-position
anchor_position
e_x
e_z
e_x
e_z
laser_emitter_frame
Trajectory
ray from emitter
ray to detector
other surface points
surface_point
of illustrated view
Target-Object

get_anchor_position(*as_matrix=False*)
Gets the laser_emitter_frame's position that was generated during construction. This is the point where the z-axis of the laser_emitter_frame hits the surface_point of the target mesh that was used to design this viewpoint. Keep in mind that 'anchor' does not necessarely mean 'anchor' of the trajectory.

Parameters **as_matrix** (*bool, optional*) – Whether to give the translation as homogeneous 4x4-matrix or vector, defaults to False

Returns Homogeneous 4x4-matrix or vector of the view's anchor position

Return type np.array (dimensions depend on parameter as_matrix)

get_measurable_surface_point_indices()
Gets the surface_point-indices measurable by this view's measurement-trajectory. The indices are abstract and are only useful when inserted into an externally maintained list of the actual surface points.

Returns Surface_point-indices measurable by this view's measurement-trajectory of an external list

Return type list[int]

get_measurable_surface_point_uncertainties()
Gets the uncertainties of each measurable surface point. The values in the returned list must be matched with the actual surface points in an external viewpoint list using this views measurable_surface_point_indices: The uncertainty at index i is meant for the surface_point in the external list evaluated at index measurable_surface_point_indices[i].

Returns uncertainties of each measurable surface point

Return type list[float]

get_orientation_matrix()
Gets the orientation of the laser_emitter_frame of this viewpoint. The orientation (not the position) remains the same for every point on the assigned measurement-trajectory

Returns Orientation of the laser_emitter_frame

Return type numpy.array

get_surface_point(*as_matrix=False*)
Gets the position of the surface point this viewpoint is based on. This is not the same as the view_anchor-point.

Parameters **as_matrix** (*bool, optional*) – Whether to give the translation as homogeneous 4x4-matrix or vector, defaults to False

Returns Homogeneous 4x4-matrix or vector of the surface point

Return type numpy.array (dimensions depend on parameter as_matrix)

get_trajectory_for_measurement()
Returns trajectory that is performed to execute the measurement corresponding to this viewpoint. This measurement-trajectory is a straight line in cartesian space.

Returns Trajectory describing how to move the robot for this views measurement

Return type moveit_msgs/RobotTrajectory

get_trajectory_to_view()
Returns the stored trajectory from a former state (might be robot's current_state or the end point of a previously executed trajectory) to the start state of this views's measurement trajectory. This is useful to connect multiple views into an execution plan.

Returns Trajectory from certain start state to this views's measurement trajectory

Return type moveit_msgs/RobotTrajectory

reverse_trajectory_for_measurement()
Flips the measurement-trajectory of this viewpoint. The start-pose becomes the end-pose and vice versa. The execution time stays the same.

Returns None

Return type NoneType

set_measurable_surface_point_indices_and_uncertainties(*measurable_point_indices*, *uncertainties*)
Sets the indices of surface points measurable by this view's measurement-trajectory as well as their model-predicted uncertainties. The indices are abstract and only meaningful when connected to a concrete surface-point-list which is maintained externally of the View-scope. Both lists must have the same length.

Parameters

- **measurable_point_indices** (*list[int]*) – List of the indices of points in an external surface_point-list that can be measured by this view
- **uncertainties** (*list[float]*) – List of uncertainty values for the measured surface_point masked through the index at the same position

set_trajectory_for_measurement(*trajectory_for_measurement*)
Trajectory that is performed to execute the measurement corresponding to this viewpoint. This measurement-trajectory is a straight line in cartesian space.

Parameters **trajectory_for_measurement** (*moveit_msgs/RobotTrajectory*) – Trajectory describing how to move the robot for this views measurement

set_trajectory_to_view(*trajectory_to_view*)
Sets a trajectory that moves the robot to the start of the view's measurement trajectory. The trajectory can be arbitrary but must end at this start point. This is useful to connect multiple views into an execution plan.

Parameters **trajectory_to_view** (*moveit_msgs/RobotTrajectory*) – Trajectory from certain start state to this views's measurement trajectory

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

T

V