# Manual

Structural Inspection Path Planner

Version 1.0

27.02.2015

by

Andreas Bircher

(andreas.bircher@mavt.ethz.ch)

ETH Zürich

Autonomous Systems Lab

# General

These packages implement the inspection path planning algorithm presented in [1]. It iteratively samples viewpoints for every triangle of a mesh representing the structure and in a second step connects them by finding the best tour. In every iteration, the viewpoints are chosen such that the connections to the neighbouring viewpoints on the tour are minimized. The viewpoint sampling is formulated as a quadratic programming problem as described in [1] and solved with a fast solver [2]. The traveling salesman problem (finding the best tour) is solved using the Lin-Kernighan heuristic. Specifically, use is made of the implementation described in [3]. Local planning is achieved by means of the RRT* planner using the original implementation described in [4].
Two types of vehicles are supported: Rotorcraft and fixed-wing systems. For both, simplified models are employed. Conservative parameter choices allow the real system to track the computed path.

In order to find high quality paths, specific properties are required for meshes used to represent the structures to be inspected. Size and shape of the triangles must be such, that it can be inspected from a single pose, without too strict constraints on the choice. Namely, small size and similar lengths of their sides favour flexibility of choice for the viewpoints.

To run the demo problem setup, download the following three packages to the source space of your catkin workspace:

- *koptplanner* (containing the inspection planner, the LKH implementation [2] and the RRT* implementation [3])
- *optec* (solver for quadratic programming problems)
- *request* (auxiliary package to load inspection path planning problem and call the planner)

After compilation, type the two commands in seperate shells:

```
roslauch koptplanner kopt.launch
```

This starts the planner service.

```
rosrun request request
```

This loads the path planning problem and calls the planner service.

The output should look similar to this:

```
[ INFO] [1424861236.821882299]: Calculating
[ INFO] [1424861239.691402858]: Request received
[ INFO] [1424861239.695396523]: STARTING ITERATION 0
[ INFO] [1424861241.011815322]: New tour cost = 56880.76
[ INFO] [1424861241.470583087]: STARTING ITERATION 1
[ INFO] [1424861242.992391763]: New tour cost = 39136.78
[ INFO] [1424861245.584589555]: STARTING ITERATION 2
[ INFO] [1424861246.970115608]: New tour cost = 38676.38
[ INFO] [1424861247.090536158]: New tour cost = 38646.93
[ INFO] [1424861248.045477915]: STARTING ITERATION 3
[ INFO] [1424861249.312673064]: New tour cost = 38142.36
[ INFO] [1424861249.540115327]: STARTING ITERATION 4
[ INFO] [1424861250.813313267]: New tour cost = 36167.05
[ INFO] [1424861251.162431963]: STARTING ITERATION 5
[ INFO] [1424861252.339910815]: New tour cost = 33695.19
[ INFO] [1424861252.498651324]: STARTING ITERATION 6
[ INFO] [1424861254.117184799]: STARTING ITERATION 7
[ INFO] [1424861255.551476755]: STARTING ITERATION 8
[ INFO] [1424861256.666804830]: New tour cost = 32291.78
[ INFO] [1424861256.850452346]: STARTING ITERATION 9
[ INFO] [1424861257.910331468]: New tour cost = 30811.15
[ INFO] [1424861258.088930575]: STARTING ITERATION 10
[ INFO] [1424861259.105583950]: New tour cost = 26947.03
[ INFO] [1424861259.197022955]: STARTING ITERATION 11
[ INFO] [1424861260.212032405]: New tour cost = 25538.89
[ INFO] [1424861260.303917900]: STARTING ITERATION 12
[ INFO] [1424861261.343619738]: New tour cost = 25085.24
[ INFO] [1424861261.436994463]: STARTING ITERATION 13
[ INFO] [1424861262.564734167]: STARTING ITERATION 14
[ INFO] [1424861263.714934205]: STARTING ITERATION 15
[ INFO] [1424861264.737435738]: New tour cost = 24694.55
[ INFO] [1424861264.828687286]: STARTING ITERATION 16
[ INFO] [1424861265.839331517]: New tour cost = 24478.94
[ INFO] [1424861265.930559529]: STARTING ITERATION 17
[ INFO] [1424861266.955946628]: New tour cost = 24455.67
[ INFO] [1424861267.047811380]: STARTING ITERATION 18
[ INFO] [1424861268.151226558]: STARTING ITERATION 19
[ INFO] [1424861269.166251372]: New tour cost = 24283.03
[ INFO] [1424861269.255795094]: Calculation time was:                29564 ms
[ INFO] [1424861269.256008179]: LKH time consumption:                8326 ms
[ INFO] [1424861269.256114059]: Initial RRT* time consumption:       7 ms
[ INFO] [1424861269.256196192]: Distance evaluation time:             6528 ms
[ INFO] [1424861269.256296485]: Viewpoint samplin time consumption:  12972 ms
[ INFO] [1424861269.256395659]: sending back response
```

To visualize the planning process in rviz, refer to the Visalization section of this manual.

The code is distributed for research use. All rights are reserved by the authors.
The code is available at : https://github.com/ethz-asl/StructuralInspectionPlanner

# Interface

The package is configured as a ROS service. To call the service use the class *koptplanner::inspection* defined in “**koptplanner/inspection.h**”. The object takes the following elements:

- Header
    - std_msgs/Header header
- Operation space: Center and size as arrays (only cuboidical spaces)
    - float64[] spaceCenter
    - float64[] spaceSize
- Required poses: An array of poses specifies the start pose, any number of other required poses and optionally a final pose. A start pose is required and if more than one pose is specified, the last will be the final pose.
    - geometry_msgs/Pose[] requiredPoses

- Mesh: The Mesh is stored as an array of polygons, one for each triangle. The polygons contain three points, specifying the three corners of the triangle ordered in a positive sense around the normal.
    - geometry_msgs/Polygon[]
- Obstacles: are stored in multiple arrays. One for the pose, one for the shape (only coordinate system-aligned cuboids are supported) and one to specify whether the obstacle is transparent (1) or not (0).
    - geometry_msgs/Pose[] obstaclesPoses
    - shape_msgs/SolidPrimitive[] obstacles
    - int32[] obstacleIntransparancy
- Inspection parameters
    - float64 incidenceAngle
    - float64 minDist
    - float64 maxDist
- Number of planner iterations
    - int16 numIterations

The service returns the computed path:

- Inspection path and its cost
    - nav_msgs/Path inspectionPath
    - float64 cost
- Error code: Corresponding to the errors described in “**koptplanner/plan.hpp**”
    - int8 errorCode

# Parameters

To chose the used model (rotorcraft or fixed-wing) make the corresponding definition in the file “**plan.hpp**” that can be found in the package *koptplanner* in the folder **include/koptplanner/**

For the rotorcraft model:

```
#define USE_ROTORCRAFT_MODEL
```

And for the fixed-wing model

#define USE_FIXEDWING_MODEL

Below, available parameters are listed. For both systems fixed-wing and rotorcraft, default values are given if applicable. The parameters have to be loaded before the call to the service. A sample parameter file is given as “**koptplanner/koptParam.yaml**”.

## Sensor Parameters

The sensor is modeled as a camera with a rectangular field of view. The parameters of that model are the angular opening of the field of view in horizontal and vertical direction, as well as the pitch angle of the centre of the field of view.

| Parameter | Rotorcraft | Fixed-wing | Optional | Description |
|---|---|---|---|---|
| camera/horizontal | - | - | No | Horizontal angle of field of view |
| camera/vertical | - | - | No | Vertical angle of field of view |
| camera/pitch | - | - | No | Sensor front down pitch |

## Rotorcraft Parameters

Dynamical constraints on the motion of the rotorcraft are imposed as a maximum translational speed, as well as a turnrate limit.

| Parameter | Rotorcraft | Fixed-wing | Optional | Description |
|---|---|---|---|---|
| rotorcraft/maxSpeed | 0.25 | NA | No | Translational speed in *m/s* |
| rotorcraft/maxAngularSpeed | 0.5 | NA | No | Rotational speed in *rad/s* |

### Fixed-wing Parameters

The fixed-wing model moves with a constant flight speed, either straight or in a left or right turn with a constant radius. It climbes or sinks with up to a maximum rate, adding loitering circles to the end of the path to overcome steeper connections.

| Parameter | Rotorcraft | Fixed-wing | Optional | Description |
|---|---|---|---|---|
| fixedwing/speed | NA | 9.0 | No | Flightspeed of plane |
| fixedwing/Rmin | NA | 60.0 | No | Minimal turn radius |
| fixedwing/maxClimbSinkRate | NA | 0.1 | No | Maximal climb and sink rate |

### Algorithm Parameters

This part contains parameters for the algorithm. Namely, *vp_tol* gives a tolerance on the Euclidean distance deviation of the viewpoint to the previous to recalculate the RRT* tree. If the tolerance is not exceeded, the old viewpoint is used along with its tree. The size of the bounding box of said RRT* trees is defined as *rrt_scope* in all directions, centred at the root. Every time an attempted connection fails, *rrt_it* iterations are performed, before another attempt is made. *rrt_it_init* gives the number of apriori iterations. The space around obstacles is searched individually on all sides for good viewpoints. This recursive search allows a maximum of *max_obstacles_depth* obstacles encountered. If *lazy_obstacle_check*, also lazy connection (whose distance is bigger than *rrt_scope*) are checked for obstacles. Collision checks of connections are performed using a discretization with step size *discretization_step*. Additional parameters allow tuning of the viewpoint sampling.

| Parameter | Rotorcraft | Fixed-wing | Optional | Description |
|---|---|---|---|---|
| algorithm/vp_tol | 0.005 | 0.05 | Yes | Tolerance to reuse RRT* tree (default value problem dependent) |
| algorithm/rrt_scope | 25.0 | 500.0 | Yes | Size of RRT*-space (default |

| | | | | |
|---|---|---|---|---|
| | | | | value problem dependent) |
| algorithm/rrt_it | 50 | 100 | Yes | Iterations of RRT* planner if no connection could be established |
| algorithm/rrt_it_init | 0 | 0 | Yes | Initial RRT* planner iterations |
| algorithm/max_obstacle_depth | 3 | 3 | Yes | Depth of recursive obstacle avoidance search (large depths slow the algorithm down) |
| algorithm/discretization_step | 0.1 | 10.0 | Yes | Collision check intervall in *m* (default value problem dependent) |
| algorithm/angular_discretization_step | 0.2 | 0.2 | Yes | Discretization for exhaustive grid search of orientation in *rad* |
| algorithm/const_A | - | 1000.0 | Yes | Additional weight factor for height differences. Scaling with A' = A/maxClimbSinkRate |
| algorithm/const_B | - | 3.0 | Yes | Weight factor for the squared distance to neighbour viewpoints |
| algorithm/const_C | - | 1e12 | Yes | Weight factor for the slack variable constraining the minimal distance to neighbours |
| algorithm/const_D | 1.0 | 3000 | Yes | Weight factor for the squared distance to the viewpoint in the previous iteration |
| algorithm/lazy_obstacle_check | false | false | Yes | Check lazy connections for collision |

### Scenario Parameters

To avoid collision, a minimal distance to obstacles is enforced when chosing viewpoints. To account for the dynamics of fixed-wing systems, this distance should at least be chosen equal to the minimal turn radius.

| Parameter | Rotorcraft | Fixed-wing | Optional | Description |
|---|---|---|---|---|
| scenario/security_distance | 2.0 | Rmin | Yes | Minimal distance for viewpoints to an obstacle |

# Error description

In the folder **koptplanner/data/** a file "**report.log**" will be generated at runtime. The file lists a summary of the scenario and any error that occurs during execution. An explanation of the errors that can occur is given below:

**Too many obstacles in the sampling area** (*OBSTACLE_INFEASIBILITY*): The problem could be related with the clustering of obstacles in a certain place. Consider increasing *max_obstacle_depth*. The corner coordinates of the specific triangle, where the error occurred are given. Remark: It is normal, that obstacles dramatically reduce the efficiency of path computation.

**No feasible position to inspect triangle** (*VIEWPOINT_INFEASIBILITY*): The given triangle cannot be inspected with the current set of parameters. This might mean that either the triangle is too big or the camera constraints are too constraining. Remedy: Increase the minimum distance, use a wider field of view or adjust the mesh resolution.

**No feasible heading to inspect triangle** (*VIEWPOINT_INFEASIBILITY*): The given triangle cannot be inspected with the current set of parameters. This usually means, that the minimum distance is too small to allow inspection of the whole triangle. Remedy: Increase the minimum distance or adjust the mesh resolution.

**No feasible connection has been found** (*CONNECTION_INFEASIBILITY*): No valid states can be sampled by the RRT* planner.

**A necessary parameter is not specified** (*MISSING_PARAMETER*): One or more required parameters are not loaded by the roslaunch file.

**No start position has been specified** (*NO_START_POSITION*): At least one required pose has to be specified, corresponding to the the start position for the inspection.

**Too few inspection areas have been specified** (*TOO_FEW_INSPECTION_AREAS*): The minimum triangles in the mesh are 4.

**The shape of an obstacle is invalid** (*INVALID_OBSTACLE_SHAPE*): Only cuboidal obstacles are accepted (shape_msgs::SolidPrimitive::BOX).

# Visualization

## Displaying the planning process in rviz

During planning the current best path and viewpoints are outputted to rviz (run: `rosrun rviz rviz`). The necessary displays are:

- ‘Path’ on topic ‘visualization_marker’
- ‘Marker’ on topic ‘viewpoint_marker’
- ‘Path’ on topic ‘stl_mesh’
- ‘Marker’ on topic ‘scenario’

To display the progress, chose ‘/kopt_frame’ as fixed frame or publish a suitable transform.

## Displaying the results in MATLAB

Use the supplied MATLAB script “**inspectionPathVisualization.m**” together with the generated file “**inspectionScenario.m**” to visualize the planning results. Both files can be found in the folder **visualization/** in the *request* package. The folder **data/** in the package *koptplanner* contains the files “**latestPath.m**” with the current best path during execution, as well as “**tourlength.m**” containing the tourlength history together with the time each path was found.

## References

[1] A. Bicher, K. Alexis, M. Burri, P. Oettershagen, S. Omari, T. Mantel, R. Siegwart, “Structural Inspection Path Planning via Iterative Viewpoint Resampling with Application to Aerial Robotics, in *Robotics and Automation (ICRA), 2015 IEEE International Conference on,* May 2015, (accepted).
[2] K.Helsgaun, “An effective implementation of the lin-kernighan traveling salesman heuristic”, *European Journal of Operational Research*, vol. 126, no. 1, pp. 106-130, 2000.
[3] H.J. Ferreau and A. Potschka and C. Kirches, “qpOASES”, [online] http://www.qpOASES.org/.
[4] S. Karaman and E. Frazzoli, “Sampling-based algorithms for optimal motion planning”, *International Journal of Robotics Research,* vol. 30, no. 7, pp. 846-894, 2011.