Motion Planning Documentation

Release 1.0

José MENDES

May 18, 2015

Contents:

CHAPTER

ONE

PLANNING_SIM — MOTION PLANNING SIMULATION

The planning_sim module implements classes to simulate a navigation scenario consisting of one or more mobile robots that autonomously plan their motion from an initial state to a final state avoiding static obstacles and other robots, and respecting kinematic (including nonhonolonomic) constraints.

The motion planner is based on the experimental work developed by Michael Defoort that seeks a near-optimal solution minimizing the time spend by a robot to complete its mission.

1.1 Obstacles and Boundary

---

class planning_sim.**Obstacle**(*position*)

Bases: object

Base class for implementing simulated obstacles.

Input *position*: array-like type representing the obstacle's position (x,y).

detected_dist(*pt*)

Return the detected distance of this obstacle as seen by a robot at the position pt (as a float).

pt_2_obst(*pt*, *offset=0.0*)

Return the point-to-obstacle distance less the offset value (as a float).

centroid = None

Obstacle's centroid. Considered here the obstacle given position.

---

class planning_sim.**RoundObstacle**(*position*, *radius*)

Bases: planning_sim.Obstacle

Representation of an obstacle as a circle.

_plt_circle(*color='k'*, *linestyle='solid'*, *filled=False*, *alpha=1.0*, *offset=0.0*)

Return a Circle object representing the obstacle geometry.

detected_dist(*pt*)

Return the detected distance of this obstacle as seen by a robot at the position pt (as a float).

plot(*fig*, *offset=0.0*)

Given a figure this method gets its active axes and plots a grey circle representing the obstacle as well as a second, concentric, dashed circle having the original circle radius plus the offset value as its own radius.

pt_2_obst(*pt*, *offset=0.0*)

Return the point-to-obstacle distance less the offset value and less the obstacle radius (as a float).

---

radius = None
Obstacle radius.

x = None
x-coordinate of the obstacle centroid.

y = None
y-coordinate of the obstacle centroid.

---

class planning_sim.**PolygonObstacle**(*vertices*)
Bases: planning_sim.Obstacle

static _calculate_polygon_area(*polygon*, *signed=False*)
Calculate the signed area of non-self-intersecting polygon

Input

***polygon*: Numeric array of points (longitude, latitude). It is assumed** to be closed, i.e. first and last points are identical

***signed*: Optional flag deciding whether returned area retains its sign:** If points are ordered counter clockwise, the signed area will be positive. If points are ordered clockwise, it will be negative Default is False which means that the area is always positive.

Output Area of polygon (subject to the value of argument signed)

static _calculate_polygon_centroid(*polygon*)
Calculate the centroid of non-self-intersecting polygon

Input *polygon*: Numeric array of points (longitude, latitude). It is assumed to be closed, i.e. first and last points are identical

Output Numeric (1 x 2) array of points representing the centroid

_create_aug_vertices(*offset*)
Create augmented vertices from original vertices and offset value

Input *offset*: Real value used as offset from original vertices. Can also be seen as the radius of the circle that will be subtracted (in the Minkowski difference sense) from the obstacle geometry.

detected_dist(*pt*)
Return the detected distance of this obstacle as seen by a robot at the position pt (as a float).

plot(*fig*, *offset=0.0*)
Given a figure this method gets its active axes and plots a grey polygon representing the obstacle as well as some dashed lines associated with the Minkowski difference of the obstacle representation and a circle of radius equals to offset.

pt_2_obst(*pt*, *offset=0.0*)
Calculate distance from point to the obstacle

Input *pt*: cardinal coordinates of the point

offset: offset distance that will be subtracted from the actual point-to-obstacle distance

Output Real number representing the distance pt_2_obstacle less offset. Negative value means that the point is closer then the offset value from the obstacle, or even inside it.

Todo

Fix error handling

---

bounding_circle_radius = None
Compute the radius of a bounding circle for the obstacle centred on the centroid.

> **Warning:** This is not necessarily the smaller bounding circle. It would be better to get the the midpoint of the greatest line segment formed by two of the polygons vertices as the circle center and half of the that line segment as its radius.

centroid = None
Obstacle's centroid

x = None
x-coordinate of the obstacle centroid.

y = None
y-coordinate of the obstacle centroid.

---

class planning_sim.**Boundary**(*x*, *y*)
Bases: object

x_max = None
Max bound in the x direction.

x_min = None
Min bound in the x direction.

y_max = None
Max bound on y direction.

y_min = None
Min bound in the y direction.

1.2 Robot and Kinematic Model

---

class planning_sim.**UnicycleKineModel**(*q_init, q_final, u_init=[0.0, 0.0], u_final=[0.0, 0.0], u_max=[1.0, 5.0], a_max=[2.0, 10.0]*)
Bases: object

This class defines the kinematic model of an unicycle mobile robot.

Unicycle kinematic model:

$$\dot{q} = \mathrm{f}(q, u) \Rightarrow \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos(\theta) \\ v\sin(\theta) \\ w \end{bmatrix}$$

Changing variables ($z = [x, y]^T$) we can rewrite the system using the flat output z as:

$$\begin{bmatrix} x \\ y \\ \theta \\ v \\ w \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ \arctan(\dot{z}_2/\dot{z}_1) \\ \sqrt{\dot{z}_1^2 + \dot{z}_2^2} \\ \dfrac{\dot{z}_1\ddot{z}_2 - \dot{z}_2\ddot{z}_1}{\dot{z}_1^2 + \dot{z}_2^2} \end{bmatrix}$$

where the the state and input vector (q, v) are function of z and its derivatives.

static _unsigned_angle(*angle*)
Map signed angles ($\theta \in (-\pi, \pi]$) to unsigned ($\theta \in [0, 2\pi)$).

Note: Method not used.

---

phi_0(*q*)
Returns z given q

phi_1(*zl*)
Returns $[x\ y\ \theta]^T$ given $[z\ \dot{z}\ \dots\ z^{(l)}]$ (only z and $\dot{z}$ are used). θ is in the range $(-\pi, \pi]$.

$$\varphi_1(z(t_k), \dots, z^{(l)}(t_k)) = \begin{bmatrix} x \\ y \\ \omega \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ \arctan(\dot{z}_2/\dot{z}_1) \end{bmatrix}$$

phi_2(*zl*)
Returns $[v\ \omega]^T$ given $[z\ \dot{z}\ \dots\ z^{(l)}]$ (only $\dot{z}$ and $\ddot{z}$ are used).

$$\varphi_2(z(t_k), \dots, z^{(l)}(t_k)) = \begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \sqrt{\dot{z}_1^2 + \dot{z}_2^2} \\ \dfrac{\dot{z}_1\ddot{z}_2 - \dot{z}_2\ddot{z}_1}{\dot{z}_1^2 + \dot{z}_2^2} \end{bmatrix}$$

phi_3(*zl*)
Returns $[\dot{v}\ \dot{\omega}]^T$ given $[z\ \dot{z}\ \dots\ z^{(l+1)}]$. (only $\dot{z}$, $\ddot{z}$ and $z^{(3)}$ are used).

$$\varphi_3(z(t_k), \dots, z^{(l+1)}(t_k)) = \begin{bmatrix} \dot{v} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial t} v \\ \frac{\partial}{\partial t} \omega \end{bmatrix} = \begin{bmatrix} \frac{\dot{z}_1\ddot{z}_1 + \dot{z}_2\ddot{z}_2}{\|\dot{z}\|} \\ \frac{(\ddot{z}_1\ddot{z}_2 + z_2^{(3)}\dot{z}_1 - (\ddot{z}_2\ddot{z}_1 + z_1^{(3)}\dot{z}_2))\|\dot{z}\|^2 - 2(\dot{z}_1\ddot{z}_2 - \dot{z}_2\ddot{z}_1)\|\dot{z}\|\dot{v}}{\|\dot{z}\|^4} \end{bmatrix}$$

acc_max = None
Absolute maximum values for the first derivative of the input vector.

Note: Value not used.

---

l = None
Number of flat output derivatives that are needed to calculate the state and input vectors.

q_dim = None
State vector order.

q_final = None
Final state.

q_init = None
Initial state.

u_dim = None
Input vector order.

u_final = None
Final input.

u_init = None
Initial input.

u_max = None
Absolute max input.

z_dim = None
Flat output vector order.

z_final = None
Final flat output.

z_init = None
Initial flat output.

---

class planning_sim.**Robot**(*eyed*, *kine_model*, *obstacles*, *phy_boundary*, *neigh*, *N_s=20*, *n_knots=6*, *t_init=0.0*, *t_sup=10000000000.0*, *Tc=1.0*, *Tp=3.0*, *Td=3.0*, *rho=0.2*, *detec_rho=3.0*, *com_range=15.0*, *def_epsilon=0.5*, *safe_epsilon=0.1*, *ls_time_opt_scale=1.0*, *dist_opt_offset=100.0*, *ls_min_dist=0.5*)

Bases: object

Class for creating a robot in the simulation. It implements the Defoort's experimental motion planning algorithm

_co_criterion(*x*)
Cost function to be minimized used for optimizing the first and intermediaries sections of the plan when conflicts are detected.

Input *x*: optimization argument.

Return Cost value.

> **Warning:** Optimization solver can misbehave for costs too big ($> 10^6$).

Note: This method is just a call to the _sa_criterion() method.

_co_feqcons(*x*)
Calculate the **equations** constraints values for the first and intermadiaries section of the plan when there are conflicts.

Input *x*: optimization argument.

Return Array with the equations' values.

Note: This method is just a call to the _sa_feqcons() method.

_co_fieqcons(*x*)
Calculate the **inequations** constraints values for the first and intermadiaries section of the plan when there are conflicts.

The following expressions are evaluated:

$$
\begin{cases}
u_{max} - |\varphi_2(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec}))| \\
u_{max} - |\varphi_2(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec}))| \\
\ldots \\
u_{max} - |\varphi_2(z(t_{N_s-1,\,sec}), \ldots, z^{(l)}(t_{N_s-1,\,sec}))| \\
\text{pt2obstacle}(\varphi_1(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec})), O_0) \\
\text{pt2obstacle}(\varphi_1(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec})), O_0) \\
\ldots \\
\text{pt2obstacle}(\varphi_1(z(t_{N_s-1,\,sec}), \ldots, z^{(l)}(t_{N_s-1,\,sec})), O_0) \\
\text{pt2obstacle}(\varphi_1(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec})), O_1) \\
\text{pt2obstacle}(\varphi_1(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec})), O_1) \\
\ldots \\
\text{pt2obstacle}(\varphi_1(z(t_{N_s-1,\,sec}), \ldots, z^{(l)}(t_{N_s-1,\,sec})), O_1) \\
\ldots \\
\text{pt2obstacle}(\varphi_1(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec})), O_{M-1}) \\
\text{pt2obstacle}(\varphi_1(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec})), O_{M-1}) \\
\ldots \\
\text{pt2obstacle}(\varphi_1(z(t_{N_s-1,\,sec}), \ldots, z^{(l)}(t_{N_s-1,\,sec})), O_{M-1}) \\
\text{collision}() \\
\text{communication}() \\
\text{deviation}()
\end{cases}
$$

where:

sec	indicates the current plan section.
O_i	is the ith detected obstacle.
M	is the number of detected obstacles.
N_s	is the number of time samples for a plan section.

Todo

Add the expression for collision, communication and deviation constraints.

Input x: optimization argument.

Return Array with the inequations' values.

_comb_bsp(*t*, *ctrl_pts*, *deriv_order*)

Combine base b-splines into a Bezier curve given control points and derivate order.

Input *ctrl_pts*: numpy array with dimension $n \times m$ n being the number of the control points and m flat output dimension.

deriv_order: derivative order of the Bezier curve.

t: discrete time array.

Return $m \times p$ numpy array representing the resulting Bezier curve, m being the flat output dimension and p the discrete time array dimension.

_compute_conflicts()

Determine the list of conflictous robots among all other robots. This method updates the `_conflict_robots_idx` private attribute.

_detect_obst()

Determinate which obstacles are within the detection radius. This method updates the `_detected_obst_idxs` private attribute.

_gen_knots(*t_init*, *t_final*)

Generate b-spline knots given initial and final times.

_log (*logid*, *strg*)
Log writer (multiprocess safe).

Input

***logid*:**	'd'	for debug
	'i'	for information
	'w'	for warning
	'e'	for error
	'c'	for critical

strg: log string.

_ls_co_criterion (*x*)
Cost function to be minimized used for optimizing the last section of the plan when conflicts are detected. It calculates the square of the total plan time.

Input *x*: optimization argument.

Return Cost value.

Warning: Optimization solver can misbehave for costs too big ($> 10^6$).

Note: This method is just a call to the `_ls_sa_criterion()` method.

_ls_co_feqcons (*x*)
Calculate the **equations** constraints values for the last section of the plan when there are conflicts.

Input *x*: optimization argument.

Return Array with the equations' values.

Note: This method is just a call to the `_ls_co_criterion()` method.

_ls_co_fieqcons (*x*)
Calculate the **inequations** constraints values for the last section of the plan when there are conflicts.

The following expressions are evaluated:

$$
\begin{cases}
u_{max} - |\varphi_2(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec}))| \\
u_{max} - |\varphi_2(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec}))| \\
\ldots \\
u_{max} - |\varphi_2(z(t_{N_s-2,\,sec}), \ldots, z^{(l)}(t_{N_s-2,\,sec}))| \\
\text{pt2obstacle}(\varphi_1(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec})), O_0) \\
\text{pt2obstacle}(\varphi_1(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec})), O_0) \\
\ldots \\
\text{pt2obstacle}(\varphi_1(z(t_{N_s-2,\,sec}), \ldots, z^{(l)}(t_{N_s-2,\,sec})), O_0) \\
\text{pt2obstacle}(\varphi_1(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec})), O_1) \\
\text{pt2obstacle}(\varphi_1(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec})), O_1) \\
\ldots \\
\text{pt2obstacle}(\varphi_1(z(t_{N_s-2,\,sec}), \ldots, z^{(l)}(t_{N_s-2,\,sec})), O_1) \\
\ldots \\
\text{pt2obstacle}(\varphi_1(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec})), O_{M-1}) \\
\text{pt2obstacle}(\varphi_1(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec})), O_{M-1}) \\
\ldots \\
\text{pt2obstacle}(\varphi_1(z(t_{N_s-2,\,sec}), \ldots, z^{(l)}(t_{N_s-2,\,sec})), O_{M-1})
\end{cases}
$$

where:

sec	indicates the current plan section.
O_i	is the ith detected obstacle.
M	is the number of detected obstacles.
N_s	is the number of time samples for a plan section.

Input x: optimization argument.

Return Array with the inequations' values.

Todo

Take into account the coflict constraints! The way it is this method is identical to the `_ls_sa_fieqcons()` method, which is pretty bad.

`_ls_sa_criterion`(x)

Cost function to be minimized used for optimizing the last section of the plan when no conflicts are detected. It calculates the square of the total plan time.

Input x: optimization argument.

Return Cost value.

Warning: Optimization solver can misbehave for costs too big ($> 10^6$).

`_ls_sa_feqcons`(x)

Calculate the **equations** constraints values for the last section of the plan when there are no conflicts.

The following expressions are evaluated:

$$\begin{cases} \varphi_1(z(t_{0,\,sec}), \ldots, z^{(l)}(t_{0,\,sec})) & - \quad q_{N_s-1,\,sec-1} \\ \varphi_1(z(t_{N_s-1,\,sec}), \ldots, z^{(l)}(t_{N_s-1,\,sec})) & - \quad q_{final} \\ \varphi_2(z(t_{0,\,sec}), \ldots, z^{(l)}(t_{0,\,sec})) & - \quad u_{N_s-1,\,sec-1} \\ \varphi_2(z(t_{N_s-1,\,sec}), \ldots, z^{(l)}(t_{N_s-1,\,sec})) & - \quad u_{final} \end{cases}$$

where:

sec	indicates the current plan section.

Input x: optimization argument.

Return Array with the equations' values.

`_ls_sa_fieqcons`(x)

Calculate the **inequations** constraints values for the last section of the plan when there are no conflicts.

The following expressions are evaluated:

$$\begin{cases} u_{max} - |\varphi_2(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec}))| \\ u_{max} - |\varphi_2(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec}))| \\ \ldots \\ u_{max} - |\varphi_2(z(t_{N_s-2,\,sec}), \ldots, z^{(l)}(t_{N_s-2,\,sec}))| \\ \text{pt2obstacle}(\varphi_1(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec})), O_0) \\ \text{pt2obstacle}(\varphi_1(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec})), O_0) \\ \ldots \\ \text{pt2obstacle}(\varphi_1(z(t_{N_s-2,\,sec}), \ldots, z^{(l)}(t_{N_s-2,\,sec})), O_0) \\ \text{pt2obstacle}(\varphi_1(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec})), O_1) \\ \text{pt2obstacle}(\varphi_1(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec})), O_1) \\ \ldots \\ \text{pt2obstacle}(\varphi_1(z(t_{N_s-2,\,sec}), \ldots, z^{(l)}(t_{N_s-2,\,sec})), O_1) \\ \ldots \\ \text{pt2obstacle}(\varphi_1(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec})), O_{M-1}) \\ \text{pt2obstacle}(\varphi_1(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec})), O_{M-1}) \\ \ldots \\ \text{pt2obstacle}(\varphi_1(z(t_{N_s-2,\,sec}), \ldots, z^{(l)}(t_{N_s-2,\,sec})), O_{M-1}) \end{cases}$$

where:

sec	indicates the current plan section.
O_i	is the ith detected obstacle.
M	is the number of detected obstacles.
N_s	is the number of time samples for a plan section.

Input *x*: optimization argument.

Return Array with the inequations' values.

_plan()

Motion/path planner method. At the end of its execution `rtime`, `ctime` and `sol` attributes will be updated with the plan for completing the mission.

_plan_section()

This method takes care of planning a section of the final path over a $T_{d/p}$ time horizon.

It also performs syncronization and data exchange among the robots.

_sa_criterion(*x*)

Cost function to be minimized used for optimizing the first and intermediaries sections of the plan when no conflicts are detected.

It calculates the distance between the final position of the proposed plan and the *goal point*.

The *goal point* is calcualted as follows:

Todo

Add the expression for computing the goal point.

Input *x*: optimization argument.

Return Cost value.

> **Warning:** Optimization solver can misbehave for costs too big ($> 10^6$).

_sa_feqcons(*x*)

Calculate the **equations** constraints values for the first and intermediaries sections of the plan when there are no conflicts.

The following expressions are evaluated:

$$\begin{cases} \varphi_1(z(t_{0,\,sec}), \ldots, z^{(l)}(t_{0,\,sec})) & - & q_{N_s-1,\,sec-1} \\ \varphi_2(z(t_{0,\,sec}), \ldots, z^{(l)}(t_{0,\,sec})) & - & u_{N_s-1,\,sec-1} \end{cases}$$

where:

sec	indicates the current plan section.

Input *x*: optimization argument.

Return Array with the equations' values.

_sa_fieqcons(*x*)

Calculate the **inequations** constraints values for the first and intermediaries sections of the plan when there are no conflicts.

The following expressions are evaluated:

$$\begin{cases} u_{max} - |\varphi_2(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec}))| \\ u_{max} - |\varphi_2(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec}))| \\ \ldots \\ u_{max} - |\varphi_2(z(t_{N_s-1,\,sec}), \ldots, z^{(l)}(t_{N_s-1,\,sec}))| \\ \text{pt2obstacle}(\varphi_1(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec})), O_0) \\ \text{pt2obstacle}(\varphi_1(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec})), O_0) \\ \ldots \\ \text{pt2obstacle}(\varphi_1(z(t_{N_s-1,\,sec}), \ldots, z^{(l)}(t_{N_s-1,\,sec})), O_0) \\ \text{pt2obstacle}(\varphi_1(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec})), O_1) \\ \text{pt2obstacle}(\varphi_1(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec})), O_1) \\ \ldots \\ \text{pt2obstacle}(\varphi_1(z(t_{N_s-1,\,sec}), \ldots, z^{(l)}(t_{N_s-1,\,sec})), O_1) \\ \ldots \\ \text{pt2obstacle}(\varphi_1(z(t_{1,\,sec}), \ldots, z^{(l)}(t_{1,\,sec})), O_{M-1}) \\ \text{pt2obstacle}(\varphi_1(z(t_{2,\,sec}), \ldots, z^{(l)}(t_{2,\,sec})), O_{M-1}) \\ \ldots \\ \text{pt2obstacle}(\varphi_1(z(t_{N_s-1,\,sec}), \ldots, z^{(l)}(t_{N_s-1,\,sec})), O_{M-1}) \end{cases}$$

where:

sec	indicates the current plan section.
O_i	is the ith detected obstacle.
M	is the number of detected obstacles.
N_s	is the number of time samples for a plan section.

Input *x*: optimization argument.

Return Array with the inequations' values.

_solve_opt_pbl()

Call the optimization solver with the appropriate parameters and parse the information returned by it. This method updates the `_C`, `_dt_final` and `_t_final` attributes.

set_option(*name*, *value=None*)

Setter for some optimation parameters.

ctime = None

List of computation time spend for calculating each planned section.

eyed = None

Robot ID.

k_mod = None
Robot kinematic model.

planning_process = None
Planning process handler for where the planning routine is called.

rho = None
Robot's radius.

rtime = None
"Race" time. Discrete time vector of the planning process.

sol = None
Solution, i.e., finded path.

1.3 WorldSim

---

class planning_sim.**WorldSim**(*name_id*, *robots*, *obstacles*, *phy_boundary*)
Bases: object

This class is a container of all simulation elements and also the interface for running the simulation.

run()
Run simulation by first calling the multiprocessing.Process.start() method to initiate the motion planning of each robot. And secondly by parsing theirs solutions and prompting the option to plot/save it.

1.4 Communication message

---

class planning_sim.**RobotMsg**(*dp*, *ip_z1*, *ip_z2*, *lz*)
Bases: object

Robot message class for exchanging information about theirs intentios.

done_planning = None
Flag to indicate that the robot has finished its planning process.

intended_path_z1 = None
Intended path (z1 coordiante).

intended_path_z2 = None
Intended path (z2 coordiante).

latest_z = None
z value calculated on the previews planned section.

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

p

Symbols

A

B

C

D

E

I

K

L

O

P

Q

R

S

U

W

X

Y

Z