Code Documentation

Index:

Ringbuffer

struct ringbuffer aka ringbuffer_t

> simple ring/circular buffer structure

Inputs

- uint8_t tail: index of where the element will be inserted next;
- uint8_t head: index of where the oldest element is in the buffer;
- uint8_t length: number of elements in the buffer;
- rbelement buffer[RBUF_SZ]: array that holds the elements of the buffer;

Ringbuffer

rbpush

> pushes data into the ring buffer, implemented as a circular FIFO buffer;

Inputs

- ringbuffer_t *buffer: buffer where the data will be pushed;
- rbelement_t data: value that will be pushed in the ring buffer;

Outputs

- rberror_t: whether the push operation was completed. By the nature of the circular buffer: if the buffer is full then the oldest value will be overwritten.

rbpop

> pops the oldest value in the buffer

Inputs

- ringbuffer_t *buffer: buffer from which the element will be popped;
- rbelement_t *data: pointer to the variable that will hold the popped value;

Outputs

- rberror_t: whether the popping procedure was concluded successfully.

rbpeek

> it returns the oldest value in the buffer without removing it.

Inputs

- ringbuffer *buffer: buffer from which the element will be taken;
- rbelement_t *data: pointer to the variable that will hold the value;

Outputs

- rberror_t: whether the operation was concluded successfully or not;

rblast

> returns the most recent (last) element of the buffer without removing it;

Inputs

- ringbuffer_t *buffer: buffer from which the value will be taken;
- rbelement_t *data: pointer to the variable that will be taken from the buffer;

Outputs

- rbelement_t: whether the operation was concluded successfully.

rbclear

> method that clears the buffer (can be used also for initialization)

Inputs

- ringbuffer_t *buffer: buffer to clear;

Outputs

- void;

Pid_controller

struct pid_controller_t

> struct that holds all the data of a PID controller.

Inputs

- int type: whether the controller will be PI or PID;
- float Kp: proportional gain of the controller;
- float Ti: integral time constant of the controller;
- float Td: derivative time constant of the controller;
- float N: discrete derivative factor;
- float prev_error: previous value of the error;
- float prev_meas: previous value of the output of the system;
- float integrator: integrator value;
- float derivative: derivative value;
- float lim_out_min: minimum limit value for the output of the controller;
- float lim_out_max: maximum limit value for the output of the controller;
- float lim_integ_min: minimum limit value for the integrator value of the controller;
- float lim_integ_max: maximum limit value for the integrator value of the controller;
- float out: output value of the controller;

Pid_controller

PID_init

> initialize the values of the PID controller;

Inputs

- pid_controller *pid: struct data where all the PID data is be stored;
- float KP: proportional constant;
- float TI: integration time constant;
- float TD: derivation time constant;
- float N: discrete derivation factor;
- int controller_type: whether the controller will be a P, PI or PID controller;

Outputs

- void;

set_limit

> initialize the values of the maximum and the minimum allowed for the output and for the integrator value;

Inputs

- pid_controller *pid: struct data where all the data will be stored ;
- float lim_out_min: minimum limit value for the output;
- float lim_out_max: maximum limit value for the output;
- float lim_integ_min: minimum limit value for the integrator value;
- float lim_integ_max: maximum limit value for the integrator value;

Outputs

- void;

PID_update

> update the value of the controller calculated in relation to the error value.

the output value of the PID controller is memorized in the `out` member of the PID struct.

Inputs

- pid_controller *pid: pointer to the struct where all the PID data is be stored;
- float set_point: value of the setpoint used to compute the error;
- float measure: measured value of the output of the controlled system;
- float T_C: sampling time;

Outputs

- void;

Custom

struct manipulator aka manipulator_t

> struct representation of a 2Dof planar manipulator

it holds RBUF_SZ values of the reference and actual position, speed and acceleration of each motor via multiple ring/circular buffer;

Inputs

- ringbuffer_t q0: holds up to RBUF_SZ values of position of the first motor;

- ringbuffer_t q1: holds up to RBUF_SZ values of position of the second motor;
- ringbuffer_t dq0: holds up to RBUF_SZ values of speed of the first motor;
- ringbuffer_t dq1: holds up to RBUF_SZ values of speed of the second motor;
- ringbuffer_t ddq0: holds up to RBUF_SZ values of acceleration of the first motor;
- ringbuffer_t ddq1: holds up to RBUF_SZ values of acceleration of the second motor;
- ringbuffer_t penup: holds up to RBUF_SZ values of position of the end-effector motor;
- ringbuffer_t q0_actual: holds up to RBUF_SZ values of actual position (encoder readings) of the first motor;
- ringbuffer_t q1_actual: holds up to RBUF_SZ values of actual position (encoder readings) of the second motor;
- ringbuffer_t dq0_actual: holds up to RBUF_SZ values of actual speed (estimation) of the first motor;
- ringbuffer_t dq1_actual: holds up to RBUF_SZ values of actual speed (estimation) of the second motor;
- ringbuffer_t ddq0_actual: holds up to RBUF_SZ values of actual acceleration (estimation) of the first motor;
- ringbuffer_t ddq1_actual: holds up to RBUF_SZ values of actual acceleration (estimation) of the second motor;
- ringbuffer_t penup_actual: holds up to RBUF_SZ values of actual position of the end-effector motor;
- float B[4]: array that represents a linearized (as in making a 2x2 become a 1x4) inertia matrix;
- float C[4]: array that represents a linearized (as in making a 2x2 become a 1x4) coriolis matrix;
- TIM_HandleTypeDef *htim_encoder1: pointer to the timer used to read the first encoder;
- TIM_HandleTypeDef *htim_encoder2: pointer to the timer used to read the second encoder;
- TIM_HandleTypeDef *htim_motor1: pointer to the timer used to apply the PWM signal to the first motor;
- TIM_HandleTypeDef *htim_motor2: pointer to the timer used to apply the PWM signal to the second motor;

Custom

init_man

> initializes the members of the man_t struct

Inputs

- man_t *manip: man_t obj. to initialize;
- TIM_HandleTypeDef *htim1: pointer to the timer used to decode the output of the first encoder;
- TIM_HandleTypeDef *htim2: pointer to the timer used to decode the output of the second encode;

Outputs

- void;

dot

> computes the dot product between two matrices, A and B, represented in vector form;

Inputs

- float *A: pointer to a vector of floats of size nA*mA, which represents the first nAxmA matrix;
- uint8_t nA: number of rows of matrix A;
- uint8_t mA: number of columns of matrix A;
- float *B: pointer to a vector of floats of size nB*mB, which represents the second nBxmB matrix;
- uint8_t nB: number of rows of matrix B;
- uint8_t mB: number of columns of matrix B;
- float *C: pointer to a vector of floats of size nA*mB, which represents the resulting nAxmB matrix -> if the operation cannot be done, it will be NULL;

Outputs

- uint8_t: `0` or 1 that shows whether the operation completed successfully or not.

inv2x2

> Inverts (if possible) a 2x2 matrix

Inputs

- float *M: pointer to the matrix to invert;
- float *invM: pointer to the inverted matrix (NULL if inversion is not possible);

Outputs

- uint8_t: shows whether the inversion was completed or not

sum

> sums two matrices (with the same size)

Inputs

- float *A: pointer to the first matrix to sum;
- float *B: pointer to the second matrix to sum;
- uint8_t n: number of elements in the matrix;
- float *C: pointer to the resulting matrix;

Outputs

- void;

diff

> subtracts two matrices (with the same size)

Inputs

- float *A: pointer to the first matrix to subtract;
- float *B: pointer to the second matrix to subtract;
- uint8_t n: number of elements in the matrix;
- float *C: pointer to the resulting matrix;

Outputs

- void;

det

> computes the determinant of a n x n matrix

this particular implementation of the determinant algorithm does not use recursion: is uses the property of elementary row operations that do not change the determinant of the matrix to find a upper triangular matrix with the same determinant of the original matrix whose determinant is to be computed. The determinant of a triangular matrix is the multiplication of the elements on its main diagonal: by finding a triangular matrix B with the same determinant of a given square matrix A, the determinant of matrix A can be found by computing the determinant of matrix B.

Inputs

- float *M: pointer to the determinant whose determinant is to be computed. IMPORTANT: the values in this matrix will be changed: ensure to keep the original data safe by passing a copy of the original matrix;
- uint8_t n: order of the matrix;
- float *d: pointer to the variable that will hold the resulting determinant;

Outputs

- void;

inv

> computes the inverse of a square matrix M;

the method requires pointer to temporary variables that will hold data used for the computation;

Inputs

- float *M: pointer to the matrix whose inverse should be computed;
- float *adjM: pointer to the temporary variable that will hold the adjugate matrix >> should be a nxn array just like M;
- float *subM: pointer to the temporary variable that will hold the submatrices used for the computation of the adjugate matrix >> should be a (n-1)x(n-1) array;
- float *trM: pointer to the temporary variable that will hold the transposed adjugate matrix >> should be a nxn array just like M;
- uint8_t n: order of the matrices;
- float *invM: pointer to the temporary variable that will hold the inverse matrix of M;

Outputs

- uint8_t: it is a boolean value that shows whether the inversion is completed successfully or not.

adj

> computes the adjugate matrix of M

the method requires temporary variables to hold useful data for the computation;

Inputs

- float *M: pointer to the **square** matrix of which the adjugate should be computed;
- float *subM: pointer to the temporary variable that will hold the submatrices used for the computation >> should be a (n-1)x(n-1) array;
- uint8_t n: order of the matrix;
- float *adjM: pointer to the variable that will hold the resulting adjugate matrix;

Outputs

- void;

tr

> transposes a matrix

Inputs

- float *M: pointer to the matrix to transpose;
- uint8_t n: number of rows;
- uint8_t m: number of columns;
- float *trM: pointer to the variable that will hold the transposed matrix;

Outputs

- void;

pseudo_inv

> computes the pseudo inverse of matrix M: (M$^{T*M)}$(-1)*M^T

the method requires temporary variables to hold useful data for the computation;

Inputs

- float *M: pointer to the matrix to pseudo-invert;
- float *trM: pointer to the variable that will hold the transposed matrix used in the pseudo-inversion;
- float *tempM: pointer to the variable that will hold the temporary transposition during the inversion;

- float *adjM: pointer to the variable that will hold the adjugate matrix used during the inversion;
- float *subM: pointer to the variable that will hold the submatrix used during the computation of the adjugate;
- float *invM: pointer to the variable that will hold the inverted matrix (A$^{T*A)}$(-1);
- float *dotM: pointer to the variable that will hold the dot product between A^T and A;
- uint8_t n: order of the matrix to invert;
- float *psinvM: pointer to the variable that will hold the pseudo-inverse;

Outputs

- void;

B_calc

> computes the inertia matrix of the manipulator

the inertia matrix B of the manipulator depends on its current configuration: it is computed analytically with the dynamic model found via the Matlab Peter Corke toolbox

Inputs

- man_t *manip: pointer to the manipulator struct that olds the reference and actual values of the position, speed and acceleration of the motors;

Outputs

- void;

C_calc

> computes the coriolis matrix of the manipulator

the coriolis matrix C of the manipulator depends on its current configuration and joint speed: it is computed analytically with the dynamic model found via the Matlab Peter Corke toolbox

Inputs

- man_t *manip: pointer to the manipulator struct that olds the reference and actual values of the position, speed and acceleration of the motors;

Outputs

- void;

controller

> implements the control law

it implements the "Inverse Dynamics" control in the joint space.

Inputs

- man_t *manip: pointer to the manipulator struct that holds all the current motors' position, speed and acceleration and their reference values;
- float *u: float[2] vector pointer that holds the control input to apply to motors (speed control);

Outputs

- void;

rad2stepdir

> converts velocity (rad/s) to step and direction (step dir);

Inputs

- float dq: velocity (rad/s);
- float resolution: resolution of the motor (how many radians is a single step?) -> expressed in radians;
- float frequency: reciprocal of delta_t -> time period in which the change of position happens;
- uint8_t *steps: pointer to the variable that will hold the number of steps;
- int8_t *dir: pointer to the variable that will hold the direction (+1 means counterclockwise, -1 means clockwise);

Outputs

- void;

speed_estimation

> computes the speed and acceleration estimations from a fixed number of previous motor positions

Inputs

- ringbuffer_t *q_actual: pointer to the ringbuffer struct that holds all the data relative to the motor position;
- float *v_est: pointer to the variable that will hold the speed estimation;
- float *a_est: pointer to the variable that will hold the acceleration estimation;

Outputs

- void;

init_rate

> initializes the rate struct

Inputs

- rate_t *rate: pointer to the rate struct to initialize;
- uint32_t ms: number of millisecond that define the rate;

Outputs

- void;

rate_sleep

> stops the process to maintain a fixed framerate (useful in while loops to implement fixed time control loops)

Inputs

- rate_t *rate: pointer to the rate struct;

Outputs

- void;

read_encoders

> reads data from both the encoders of the 2Dofs planar manipulator

the method uses two timers to decode the signals coming from both the encoders and memorizes the measured positions of the motors, taking into account the reduction ratio of each motor by means of the ARR registers of the timers (ARR=CPR*REDUCTION). The method also estimates the speed and accelerations by using the timestamp method.

Inputs

- man_t *manip: pointer to the manipulator struct that holds both the desired and actual motor positions, speeds and accelerations;

Outputs

outputs

update_speeds

> update the estimated speed data from both the encoders of the 2Dofs planar manipulator

Inputs

- man_t *manip: pointer to the manipulator struct that holds both the desired and actual motor positions, speeds and accelerations;

Outputs

outputs

apply_velocity_input

> applies the velocity inputs to the motors

it uses a timer to send a pulse width modulation signal whose frequency regulates the rotation speed of the motors.

The formulae used to set the ARR and CCR registers:

$$ARR = \frac{Res \cdot freq_{clock}}{v \cdot reduction \cdot microstepping \cdot prescaler}$$

$$CCR = ARR/2$$

with Res being the motor step resolution (in radians), v is the velocity, reduction is the reduction ratio and prescaler is the prescaler value (which is fixed).

Inputs

- TIM_HandleTypeDef *htim1: pointer to the handler of the timer that sends the PWM signal to the first motor;
- TIM_HandleTypeDef *htim2: pointer to the handler of the timer that sends the PWM signal to the second motor;
- float *u: array of two floats containing the velocity inputs of the motors;

Outputs

- void;

apply_position_input

> applies the position inputs to the motors

it uses a timer to send a pulse width modulation signal whose frequency regulates the rotation speed of the motors.

The position input is converted into a velocity input (using the relationship between max acceleration and time duration of a cycloidal trajectory):

$$\Delta t = \sqrt{2\pi \cdot |u - pos| / factor}$$

$$v = \frac{(u - pos)}{\Delta t}$$

where the factor regulates how much the distance to move influences the time duration of the movement. The formulae used to set the ARR and CCR registers:

$$ARR = \frac{Res \cdot freq_{clock}}{v \cdot reduction \cdot microstepping \cdot prescaler}$$

$$CCR = ARR/2$$

with Res being the motor step resolution (in radians), v is the velocity, reduction is the reduction ratio and prescaler is the prescaler value (which is fixed).

Inputs

- TIM_HandleTypeDef *htim1: pointer to the handler of the timer that generates the PWM signal for the first motor;
- TIM_HandleTypeDef *htim2: pointer to the handler of the timer that generates the PWM signal for the second motor;
- float *u: pointer to a float array of size two containing the position input;
- float *pos: pointer to a float array of size two containing the actual position of the manipulator;

Outputs

- void;

start_timers

> starts all the timers needed for the control

Inputs

- TIM_HandleTypeDef *htim1: pointer to the first timer handler;
- TIM_HandleTypeDef *htim2: pointer to the second timer handler;
- TIM_HandleTypeDef *htim3: pointer to the third timer handler;
- TIM_HandleTypeDef *htim4: pointer to the fourth timer handler;

Outputs

- void;

stop_timers

> stops all the timers needed for the control

Inputs

- TIM_HandleTypeDef *htim1: pointer to the first timer handler;
- TIM_HandleTypeDef *htim2: pointer to the second timer handler;
- TIM_HandleTypeDef *htim3: pointer to the third timer handler;
- TIM_HandleTypeDef *htim4: pointer to the fourth timer handler;

Outputs

- void;

log_data

> Send all the manipulator data via serial connection with a message

Inputs

- UART_HandleTypeDef *huart: pointer to the uart handler;
- man_t *manip: pointer to the man_t structure that holds all the manipulator data;

Outputs

- void;

setup_encoders

> performs the necessary operations for the setup of the timer that will manage the econder sampling

Inputs

- TIM_HandleTypeDef *htim: pointer to the sampling timer handler;

Outputs

- void;

PID_controller_velocity

> Calculate the velocity control output for the motors.

The motor driver (DRV8825) take a STEP/DIR input, that represent a velocity command. To accomplish the translation from position input to velocity input, first of all the current position is read, after that the distance between the setpoint position and the current position is calculated, then the needed time for motion is calculated using the cycloidal trajectory. The velocity output is given by the ratio between distance and time. To avoid vibration a threshold is set.

Inputs

- man_t *manip: pointer to the manipulator struct;
- pid_controller_t *pid1 : pointer to the PI velocity controller of the first joint;
- pid_controller_t *pid2 : pointer to the PI velocity controller of the second joint;
- float *u: pointer to the velocity output;

Outputs

- void;

PID_controller_velocity

> Calculate the velocity control output for the links.

Inputs

- man_t *manip: pointer to the manipulator struct;
- pid_controller_t *pid1 : pointer to the PI velocity controller of the first joint
- pid_controller_t *pid2 : pointer to the PI velocity controller of the second joint
- float u: velocity output

Outputs

- void;

homing

> Start the homing sequence when the HOM command is recived from the serial

In the serial callback we don't call directly the homing method because the if this happen we are unable to sense other the interrupt when the homing code is executed, to get around this drawback, the flag variable homing_triggered is set high and than in the main loop the homing sequence starts. The flag variables is_home1 and is_home2 are used to avoid changing the offset errors measured during the homing procedure.

Inputs

- man_t *manip: pointer to the manipulator struct;
- TIM_HandleTypeDef *htim1: First joint PWM out timer;
- TIM_HandleTypeDef *htim2: Second joint PWM out timer;
- pid_controller_t *pid_v1 : pointer to the PI velocity controller of the first joint;
- pid_controller_t *pid_v2 : pointer to the PI velocity controller of the second joint;
- pid_controller_t *pid_p1 : pointer to the PID position controller of the first joint;
- pid_controller_t *pid_p2 : pointer to the PID position controller of the second joint;

Outputs

- void;

Manipulator

class Manipulator

> class that represents a 2Dof manipulator and its trajectory;

Inputs

- list q: generalized coordinate values;

Manipulator.draw_pose

> draws the two links of the manipulator

Inputs

- canvas context 2D ctx;

async Manipulator.draw_traces

> draws the "traces" of the manipulator (the trajectory that the links should have followed)

the method was made async so that it may not increase too much the page reaction time;

Inputs

- canvas context 2D ctx;
- optional list[string] colors: list of 2 colors to use for the links traces;

Manipulator.dk

> computes the direct kinematics of the manipulator

Inputs

- list[float] q: configuration of the robot;

Outputs

- list[float]: list containing the positions of the ending points (list of coordinates) of the links

class Point

> class modeling a point in the operational space

Inputs

- float x: x coordinate of the point;
- float y: y coordinate of the point;

- dict settings: dictionary containing data for coordinate conversion (from canvas space to operational space)

Point.add

> adds to point vectors together

Inputs

- Point other: Point to add;

Outputs

- Point: result

Point.sub

> subtracts two point vectors together

Inputs

- Point other: point to subtract;

Outputs

- Point: result

Point.mag

> computes the length of the point vector

Outputs

- float: length of the point vector

Point.scale

> scales the point vector length with the specified scalar

Inputs

- float scalar: value used to scale the length;

Outputs

- Point: scaled point vector

Point.rot

> rotates (of the specified angle) the point vector around its origin

Inputs

float delta: angle (in radians) used for the rotation (a positive rotation is counterclockwise);

Outputs

- Point: rotated point vector

Point.set

> sets the length of the point vector to the specified value

Inputs

- float scalar: length of the new vector;

Outputs

- Point: scaled point vector;

Point.angle

> computes the angle of the point vector (in radians)

Outputs

- float: angle of the point vector

class Trajectory

> class that models a trajectory made fo multiple patches

the trajectory modelled by this class can be made up of two different kind of patches:

- line: a linear movement from point A to point B;
- circle: a curvilinear movement between two points (A and P) by following a circumference with given center and radius;

Trajectory.add_line

> adds a line patch to the trajectory

Inputs

- Point p0: starting point of the patch;
- Point p1: ending point of the patch;
- bool raised: boolean that states wether the pen on the end effector of the manipulator should be raised or not;

Trajectory.add_circle

> adds a circular patch to the trajectory

Inputs

- Point c: center of the circumference;
- float r: radius of the circumference;
- float theta_0: angle of the starting point;
- float theta_1: angle of the ending point;
- bool raised: states wether the pen on the end effector of the manipulator should be up or not;
- Point a: starting point of the circumference;
- Point p: ending point of the circumference;

Trajectory.reset

> resets the trajectory data

Trajectory.draw

> draws the trajectory

each patch of the trajectory will be drawn on the canvas specified by the user

Inputs

- 2D canvas context ctx: context of the canvas that will be used to draw on;

Main

(eel) js_get_data

> gives trajectory data to the python backend

Outputs

- list: list of json objs containing trajectory data

serial_online

> shows a red or green circle depending on the serial com. status

Inputs

- bool is_online: boolean stating if the serial communication is online

draw_background

> draws the background on the canvas

the background shows two circumferences with radii equal to the lengths of the two links of the manipulator, with red areas showing where the end effector should avoid going to avoid problems with cable intertwining, etc...

Inputs

- optional string color: background color;
- optional string line: line color;
- optional string limit: limited zones color;

draw_point

> draws a point on the canvas

Inputs

- float x: x coordinate of the point;
- float y: y coordinate of the point;

find_circ

> find the circumference arc given 3 points

the circumference arc can be defined with 3 points:

- the first point determines where the arc should start;
- the second point determines the diameter of the circumference;
- the third point determines the angle of the arc, which is the angle between the vector from the center of the circumference to the first point and the vector from the center to the third point;

Outputs

- Point center: center of the circumference;
- Point a: starting point of the arc;
- Point p: ending point of the arc;
- float r: radius of the circumference;
- theta_0: angle of the vector a-c;
- theta_1: angle of the vector p-c;

handle_input

> handles the click event on the canvas

Inputs

- mouse event e;

line_tool

> method that shows the line tool gui

circle_tool

> method that shows the circle tool gui

Trajpy

time_row

> computes and returns as a list a row of the vandermont matrix

the returned row is actually the one linked to the equation a0t^0 + a1t^1 + a2t^2 + ... + ant^n = 0 =>

$$\sum_{i}^{n} a_i t^i = 0$$

with n specified by the user. The method can also return the *derivative* of this equation, up to the 2nd derivative

Inputs

- float t: the value of t used to compute the list;
- int deg: degree of the equation (n in the example above);
- int der: order of the derivative (from 0 up to 2);

Outputs

- list[float] : row of the vandermont matrix: [1, t, t**2, ..., t**n] (or its derivatives)

spline3

> computes the coefficients of a cubic spline

the cubic spline has the following structure:
q = a0+a1t+a2t^2 + a3t^3
dq = a1+2a2t+3a3t^2
ddq = 2a2+6a3t
where q is the position spline, dq is the velocity spline and ddq is the acceleration spline

Inputs

- list[point_time] q: it is a list of tuples of a value (float) and a time instant (float). These values and time instants will be used to write a polynomial (with variable "t") that will cross the specified values at t equal to the specified times;
- list[point_time] dq: it is a list of a value (float) and a time instant (float). These values and time instants will be used to write a polynomial (with variable "t") which derivative will cross the specified values at t equal to the specified times;

Outputs

- list[function] : list of functions of the variable t that represent the trajectories q(t), dq(t) and ddq(t)

spline5

> computes the coefficients of a 5th order spline

the 5th order spline has the following structure:
q = a0+a1t+a2t^2 + a3t^3 + a4t^4 + a5t^5
dq = a1+2a2t+3a3t^2 + 4a4t^3 + 5a5t^4
ddq = 2a2+6a3t+12a4t^2 + 20a5t^3

Inputs

- list[point_time] q: it is a list of tuples of a value (float) and a time instant (float). These values and time instants will be used to write a polynomial (with variable "t") that will cross the specified values at t equal to the specified times;
- list[point_time] dq: it is a list of a value (float) and a time instant (float). These values and time instants will be used to write a polynomial (with variable "t") which derivative will cross the specified values at t equal to the specified times;

Outputs

- ndarray : numpy array of the coefficients of the 5th order polynomial that crosses the specified points.

rangef

> returns a list containing all the values between the initial and final values with the specified step size (that can be float)

Inputs

- float start: the starting value of the range;
- float step : the step size used to find the values withing the specified range;
- float end: the final value of the range;
- bool consider_limit: boolean that indicates whether the end value should be inserted in the returned list or not;

Outputs

- list[float] : list of all the values found in the specified range with the specified step size.

compose_spline3

> returns the trajectory that results from the composition of the cubic splines obtained for each couple of points in the specified path.

Inputs

- list[float] q: list of points that compose the path;
- float ddqm: maximum acceleration;
- list[float] dts: duration of each splines;

Outputs

- list[tuple[list[function], float]] : list of function/spline-duration tuples.

cubic_speeds

> computes the speeds of the intermediate points of a cubic spline

Inputs

- list[float] q: list of the points of the path;
- list[float] dts: list of the duration of each section of the path;

Outputs

- list[float]: list of intermediate speeds.

preprocess

> subdivides the ranges passed as a list into smaller ranges of size equal to the specified limit.

Inputs

- list[float] q: list of values;
- float limit: limit value used to subdivide the specified ranges (q);

Outputs

- list[float]: new list of values whose ranges are smaller or equal to the specified limit;

trapezoidal

> computes the trapezoidal speed profile trajectory for the specified points;

the trapezoidal trajectory is subdivided into 3 sections, 2 parabolic ones of equal duration (initial and final ones) and a linear section with constant velocity.

Inputs

- list[float] q: list that contains the initial and final values of the trajectory;
- float ddqm: maximum acceleration;
- float tf: duration of the trajectory;

Outputs

- list[tuple[ndarray, float]]: list containing the coefficients of each section of the trajectory and their durations.

compose_trapezoidal

> returns the trajectory that results from the composition of the trapezoidal speed profile trajectories obtained for each couple of points of the specified path.

Inputs

- list[float] q: list of points that compose the path;
- float ddqm: maximum acceleration;

Outputs

- list[tuple[ndarray, float]] : list of coefficients/trapezoidal-duration tuples.

cycloidal

> computes a cycloidal trajectory

the cycloidal trajectory is not polynomial, so it cannot be represented as a list of coefficients: for this reason a function handle is created for the position q, the speed dq and the acceleration ddq that can be used to compute the trajectory given t.

Inputs

- list[float] q: initial and final values of the trajectory;
- float ddqm: maximum acceleration;
- float tf: duration of the trajectory;

Outputs

- tuple[list[function], float] : function-handle/cycloidal-duration tuple.

compose_cycloidal

> returns the trajectory resulting from the composition of the cycloidal trajectories obtained from each couple of values in the specified path.

given that the cycloidal trajectory cannot be represented with just a list of coefficients, the returned trajectory will be a list of function handles.

Inputs

- list[float] q: list of points in the path (the timing law will be autogenerated);
- float ddqm: maximum acceleration;

Outputs

- list[tuple[list[function], float]]: list of trajectory/cycloidal-duration tuples.

ik

> inverse kinematics of a 2Dofs planar manipulator

it can compute the joint variables values even if the orientation of the end effector is not specified.

Inputs

- float x: x coordinate of the end effector;
- float y: y coordinate of the end effector;
- float theta: orientation of the end effector (angle of rotation relative to the z axis with theta=0 when the x axis of the end effector is aligned with the x axis of the base frame of reference);
- dict[float] sizes: sizes of the two links that make up the manipulator, accessed via 'l1' and 'l2';

Outputs

- ndarray: column numpy array containing the values of the joint coordinates.

dk

> direct kinematics of a 2Dofs planar manipulator

it can compute the x, y coordinates of the end effector and its orientation theta (angle of rotation relative to the z axis with theta=0 when the x axis of the end effector is aligned with the x axis of the base frame of reference)

Inputs

- ndarray q: colum numpy array containing the values of the joint coordinates;
- dict[float] sizes: sizes of the two links that make up the manipulator, accessed via 'l1' and 'l2';

Outputs

- ndarray: column numpy array containing the values of the coordinates of the end effector (x, y and the rotation angle theta).

Point (class)

> Point class used to represent points in the operational space with a cartesian frame of reference

Inputs

- float x: x coordinate of the point
- float y: y coordinate of the point

Point.mag

> computes the length of the vector <x, y>

Outputs

- float: length of the vector

Point.angle

> computes the angle of the vector <x, y>

Outputs

- float: angle of the vector

Point.rotate

> rotates the vector around its origin

Inputs

- float phi: angle (in radians) of rotation;

Outputs

- Point: rotated vector

Point.ew

> computes the element-wise multiplication (scalar product)

given two points/vectors a and b, the method returns the value x_*ax_b+y_a*y_b (equivalent to a*b^T)

Inputs

- Point other: the other point with which the element wise multiplication is done;

Outputs

- float: scalar product

Point.angle_between

> computes the angle between two vectors

Inputs

- Point other: vector with which the computation will be done;

Outputs

- float: the angle between the two vectors

slice_trj

> slices the trajectory patch

depending on the type of notes (line or circle) this function slices the trajectory patch in segments depending on a timing law s(t) specified by the user

Inputs

- dict patch: trajectory patch with the following structure:

```
{
'type': 'line' or 'circle',
'points': [[x0, y0], [x1, y1]], # start and end points
'data': {'center':c, 'penup':penup, ...}
}
```

- **kargs:
  - 'max_acc': maximum acceleration;
  - 'line': timing law s(t) for a linear trajectory patch;
  - 'circle': timing law s(t) for a circular trajectory patch;
  - 'sizes': sizes dict containing the sizes of the two links of the manipulator ({'l1': l1, 'l2':l2});
  - 'Tc': time step used for the timing law;

Outputs

- list q0s: list of values for the generalized coordinate q of the first motor;
- list q1s: list of values for the generalized coordinate q of the second motor;
- list penups: list of values that show wether the pen should be up or down;
- list ts: list of time instants;

find_velocities

> computes the velocity of the trajectory in each time instant

Inputs

- list[float] q: list of motor positions;
- list[float] ts: list of time instants;

Outputs

- list[float]: list of velocities

find_accelerations

> computes the acceleration of the trajectory in each time instant

Inputs

- list[float] dq: list of the motor velocities;
- list[float] ts: list of time instants;

Outputs

- list[float]: list of accelerations

Main

send_data

> sends data to the micro controller

Inputs

- str msg_type: type of message to send : "trj" is used for trajectory data;
- any list **data: any number of lists (containing the position setpoints to send in case of trj data);

Outputs

- None;

trace_trajectory

> draws the trajectories on the GUI

Inputs

- tuple[list, list] q: a tuple containing the list of positions for each motor;

Outputs

- None;

eel.expose py_log

> simply prints a message on the python console

Inputs

- str msg: message to be print;

Outputs

- None;

eel.expose py_get_data

> gets the trajectory data from the web GUI and converts it into a list of setpoints to be sent to the micro controller

Inputs

- None;

Outputs

- None;

eel.expose py_homing_cmd

> sends the homing command to the micro controller

Inputs

- None;

Outputs

- None;

eel.expose py_serial_online

> return whether the serial is online or not

Inputs

- None;

Outputs

- bool: bool value that shows if the serial is online or not;

eel.expose py_serial_sartup

> initializes the serial communication

Inputs

- None;

Outputs

- None;