Manipulator RRT

v1

Generated by Doxygen 1.8.2

Tue Sep 25 2012 01:24:38

Contents

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 cone_obs Struct Reference

Structure for conical obstacles.

Public Attributes

- double $*$ beta

  Vector of cone half-angles (deg)
- double $*$ h1

  Vector of cone heights from apex of truncated cone bottoms.
- double $*$ h2

  Vector of cone heights from apex of truncated cone tops.
- double $***$ Tinv

  Vector of transformation matrices from the world frame to the cone frame.

3.1.1 Detailed Description

Structure for conical obstacles.

Structure variable used to store conical obstacle data, including cone half-angles, β, cone lower boundaries, $z = h_1$, cone upper boundaries, $z = h_2$, and the inverse transformation matrices, T^{-1}, that resolve coordinates from the world frame into the frame of the cone (z-axis aligned, origin at cone apex). Currently unimplemented.

3.2 coords Struct Reference

Structure for coordinate storage.

Public Attributes

- double $*$ x

  Array of x-coordinates.
- double $*$ y

  Array of y-coordinates.
- double $*$ z

  Array of z-coordinates.

3.2.1 Detailed Description

Structure for coordinate storage.

Structure variable used to point to a set of 3D Cartesian coordinates of type double. Often used to save the world coordinates of a transformed set of manipulator geometry points, corresponding to a particular joint angle configuration vector.

3.3 cuboid_obs Struct Reference

Structure for cuboidal obstacles.

Public Attributes

- double $*$ a

  Vector of unit-normal vector x-component values, in groups of 6, one for each cuboid.
- double $*$ b

  Vector of unit-normal vector y-component values, in groups of 6, one for each cuboid.
- double $*$ c

  Vector of unit-normal vector z-component values, in groups of 6, one for each cuboid.
- double $*$ d

  Vector of distances from the origin, $d = -ax - by - cz$, in groups of 6, for any point (x,y,z) in the planes defining each cuboid.
- double $***$ T

  Vector of transformation matrices from the cuboid frame to the world frame (only used for plots; not required for collision-checking)

3.3.1 Detailed Description

Structure for cuboidal obstacles.

Structure variable used to store cuboidal obstacle data, including the vector normals, $\hat{n} = (a,b,c)$, and distances, d, for the set of 6 planes defining each cuboid. cuboid_obs is distinguished from planar_obs by the fact that sets of 6 planes at a time correspond to an obstacle, meaning $f = ax + by + cz + d < 0$ must hold for a group of all 6 planes at once for a collision whereas in planar_obs only one violation is required.

3.4 cylindrical_obs Struct Reference

Structure for cylindrical obstacles.

Public Attributes

- double $*$ r

  Vector of cylinder radii.
- double $*$ H

  Vector of cylinder heights.
- double $***$ Tinv

  Vector of transformation matrices from the world frame to the cylinder frame.

3.4.1 Detailed Description

Structure for cylindrical obstacles.

Structure variable used to store cylindrical obstacle data, including the cylinder radii, r, the heights, H, and the inverse transformation matrices, T^{-1}, that resolve coordinates from the world frame into the frame of the cylinder (z-axis aligned, origin at geometric center). Used during constraint checking to test whether the inequalities $f_1 = x^2+y^2-r^2 < 0$, $f_2 = z-\frac{H}{2} < 0$, and $f_3 = z+\frac{H}{2} > 0$ hold, indicating a collision.

3.5 DHparams Struct Reference

Structure of Denavit-Hartenberg parameters.

Public Attributes

- double $*$ d

  Vector of translations along the z_i (rotation) axes.
- double $*$ a

  Vector of translations along the x_{i-1} (perp) axes.
- double $*$ alpha

  Vector of rotation angles about the x_{i-1} (perp) axes.

3.5.1 Detailed Description

Structure of Denavit-Hartenberg parameters.

Structure variable containing the Denavit-Hartenberg (DH) parameters of the manipulator arm. DH parameters are a particular manipulator parametrization that models robotic arm joints as two "screw" operations - one rotation and translation about the original x-axis followed by a rotation and translation about the intermediate z-axis. This can be used to represent any 2-DOF joint; more complicated joints can be represented by several degenerate-case sets of DH parameters (refer to p.103 of "Planning Algorithms" by Steven LaValle). The "theta" parameters about z are not included as each corresponds to a node stored in the RRTs.

3.6 geom Struct Reference

Structure for manipulator geometry.

Public Attributes

- double $*$ L

  Array of link lengths.
- double $*$ W

  Array of link widths.
- double $*$ H

  Array of link heights.
- double $*$ rho_x

  x-coordinates in each link's body-fixed frame to the back bottom-left corner of their respective Oriented Bounding Boxes (OBB's)
- double $*$ rho_y

  y-coordinates in each link's body-fixed frame to the back bottom-left corner of their respective Oriented Bounding Boxes (OBB's)

- double * rho_z

  z-coordinates in each link's body-fixed frame to the back bottom-left corner of their respective Oriented Bounding Boxes (OBB's)
- int * N_coords

  Array of the numbers of coordinates used to represent each link (at least 8, one for each OBB corner, plus any face points)
- double *** Body_coords

  An array of the x-y-z body-fixed OBB coordinate arrays, one for each link.

3.6.1 Detailed Description

Structure for manipulator geometry.

Structure variable encapsulating manipulator geometry information, including link dimensions, Oriented-Bounding Box (OBB) positioning, and body-fixed OBB coordinates.

3.7 list_node Struct Reference

Structure for linked lists.

Public Attributes

- int data

  Integer data associated with the list node.
- struct list_node * next

  Pointer to the next linked list element (or NULL at end of list)

3.7.1 Detailed Description

Structure for linked lists.

Structure variable used to represent the element of a linked list of integer data. List nodes are employed primarily to store lists of subtree leaves for each node in an RRT data structure.

3.8 obstacles Struct Reference

Structure for all obstacle data.

Public Attributes

- struct cone_obs * cones

  All conical obstacle information.
- struct cylindrical_obs * cylinders

  All cylindrical obstacle information.
- struct cuboid_obs * cuboids

  All cuboidal obstacle information.
- struct planar_obs * planes

  All planar obstacle information.
- struct temp_obs * temp_zones

  All current temperature obstacle information.

- int n_cones

  Number of conical obstacles.
- int n_cylinders

  Number of cylindrical obstacles.
- int n_cuboids

  Number of cuboidal obstacles.
- int n_planes

  Number of planar obstacles.
- int n_temp_zones

  Number of temperature obstacles.

3.8.1 Detailed Description

Structure for all obstacle data.

Structure variable representing the compilation of all obstacle information, including conical obstacles, cylindrical obstacles, cuboidal obstacles, planar obstacles (all considered "static") and finally temperature obstacles (considered "dynamic", though static once generated). Only temperature obstacles can increase in number during motion plan execution; all else must be prescribed prior to the simulation. The final element of the cuboidal obstacle structure is taken to be the cuboidal obstacle, eliminated from consideration once grasped by the manipulator.

3.9 planar_obs Struct Reference

Structure for planar obstacles.

Public Attributes

- double $*$ a

  Vector of unit-normal vector x-component values, one for each plane.
- double $*$ b

  Vector of unit-normal vector y-component values, one for each plane.
- double $*$ c

  Vector of unit-normal vector z-component values, one for each plane.
- double $*$ d

  Vector of distances from the origin, $d = -ax - by - cz$, for any point (x, y, z) in each plane.

3.9.1 Detailed Description

Structure for planar obstacles.

Structure variable used to store planar obstacle data, including the vector normal, $\hat{n} = (a, b, c)$, and distance, d, to the plane from the origin. The plane equation, $f = ax + by + cz + d$, corresponds to that of the world-frame.

3.10 temp_obs Struct Reference

Structure for conical temperature obstacles.

Public Attributes

- double $*$ beta

  Vector of cone half-angles (deg)
- double $*$ h1

  Vector of cone heights from apex of truncated cone bottoms.
- double $*$ h2

  Vector of cone heights from apex of truncated cone tops.
- double $***$ Tinv

  Vector of transformation matrices that transform from the world frame to the temp obs frame.

3.10.1 Detailed Description

Structure for conical temperature obstacles.

Structure variable used to store temperature obstacles, which are modelled as truncated circular cones (frustra). Includes cone half-angles, β, cone lower boundaries, $z = h_1$, cone upper boundaries, $z = h_2$, and the inverse transformation matrices, T^{-1}, that resolve coordinates from the world frame into the frame of the cone (z-axis aligned, origin at cone apex). Used to generate temperature obstacles along sensor unit normals during motion plan execution, and during constraint checking to test whether the inequalities $f_1 = x^2 + y^2 - (z\beta)^2 < 0$, $f_2 = z - h_2 < 0$, and $f_3 = z - h_1 > 0$ hold, indicating a collision.

3.11 tree Struct Reference

Structure for Rapidly-exploring Random Trees (RRT's).

Public Attributes

- double $**$ nodes

  Array of pointers to the vectors q of joint angles, whose k-th element corresponds to node k.
- double $*$ costs

  Array of costs-to-come (for a forward tree) or costs-to-go (for a reverse tree), whose k-th element corresponds to node k.
- int $*$ parents

  An array whose k-th index corresponds to the parent node to node k (or a -1 for the root node)
- int $*$ connections

  An array whose k-th index corresponds to the index of the leaf node in the partner tree that is connected to node k (or a -1 if no connection was made)
- struct list_node $**$ leaf_lists

  An array of linked list pointers, the k-th element of which is used to find the "leaves" to which node k is connected.
- int $*$ safety

  Array of booleans used to identify if saved nodes are safe to use after tree construction (tracks if they violate new constraints)
- int $*$ indices

  An array whose k-th index is the index for node k (necessary for the KD-tree algorithm)
- struct kdtree $*$ kd_tree

  Pointer to the tree's associated KD-tree, used for NearestNeighbor selection.

3.11.1 Detailed Description

Structure for Rapidly-exploring Random Trees (RRT's).

Structure variable used to represent an RRT. The particular data structure used by Manipulator RRT is augmented to include indicators of node safety, linked lists for each tree node, and other elements in order to improve the efficiency of replanning and closed-loop control.

Chapter 4

File Documentation

4.1 ManipulatorRRT.cpp File Reference

Source file for the control code of the Stanford Structures and Composites Lab manipulator arm.

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <math.h>
#include <assert.h>
#include <string.h>
#include <float.h>
#include <windows.h>
#include <time.h>
#include <phidget21.h>
#include <engine.h>
#include <kdtree.h>
```

Classes

- struct list_node

  Structure for linked lists.
- struct tree

  Structure for Rapidly-exploring Random Trees (RRT's).
- struct coords

  Structure for coordinate storage.
- struct geom

  Structure for manipulator geometry.
- struct DHparams

  Structure of Denavit-Hartenberg parameters.
- struct planar_obs

  Structure for planar obstacles.
- struct cuboid_obs

  Structure for cuboidal obstacles.
- struct cylindrical_obs

  Structure for cylindrical obstacles.
- struct cone_obs

  Structure for conical obstacles.

- struct temp_obs

  Structure for conical temperature obstacles.
- struct obstacles

  Structure for all obstacle data.

Macros

- #define _CRT_SECURE_NO_DEPRECATE

  Suppress compiler warnings about deprecated functions, e.g. `strcpy` *(warning C4996)*
- #define PI 3.14159265358979323846

  Numerical approximation of π.
- #define SORT_SWITCH 100

  Array size above which an $O(n \log n)$ sorting algorithm should be used instead of InsertionSort.
- #define Numel(x) (sizeof(x) / sizeof(x[0]))

  Determines the number of elements in a non-dynamically allocated array.
- #define Round(number) (number >= 0) ? (int)(number + 0.5) : (int)(number - 0.5);

  Round a number to the nearest integer depending on its sign and fractional part (CURRENTLY UNUSED)
- #define Round2Res(value, resolution) (((double) Round(value/resolution)) * resolution)

  Rounds a double to a desired scale, e.g. resolution = 0.5 rounds to nearest half-integer (CURRENTLY UNUSED)
- #define Max(a, b) (((a) > (b)) ? (a) : (b))

  Returns the maximum of two values a and b.
- #define Min(a, b) (((a) < (b)) ? (a) : (b))

  Returns the minimum of two values a and b.
- #define Sqrt(x) (pow(x,0.5))

  Returns the square root of a value (no error-checking - value must be non-negative)
- #define va_copy(dest, src) ((dest) = (src))

  Copies a `va_list` *("variable argument" list) pointer to another* `va_list` *pointer (CURRENLTY UNUSED)*

Typedefs

- typedef enum datatypes Type

  Enumerated list of datatypes.

Enumerations

- enum datatypes {
  Long, Short, Char, Int,
  Float, Double }

  Enumerated list of datatypes.

Functions

- int CCONV AttachHandler (CPhidgetHandle HANDLE, void *userptr)

  Code to execute when a Phidget device has been successfully attached.
- int CCONV DetachHandler (CPhidgetHandle HANDLE, void *userptr)

  Code to execute when a Phidget device has been successfully attached.
- int CCONV ErrorHandler (CPhidgetHandle HANDLE, void *userptr, int ErrorCode, const char *Description)

  Code to execute when an error has occurred while interacting with a Phidget device.
- CPhidgetAdvancedServoHandle InitializeServos (int n, int *channels, int grip_channel, double *AccelThrottle, double *VelLimThrottle, double grip_AccelThrottle, double grip_VelLimThrottle)

Attempts to connect to robotic arm servomotors and initializes the Phidget advanced servo global variable with user-defined values.

- CPhidgetInterfaceKitHandle InitializeTempSensors (int n_tempsensors, int ∗sensor_channels, int rate_-tempsensors)

  Attempts to connect to the temperature sensor potentiometer board and initializes the Phidget interface kit global variable with user-defined values.
- void PrintListToFile (FILE ∗filename, char ∗format, struct list_node ∗ptr)

  Print the elements of a linked list to file.
- struct list_node ∗ AddListElement (struct list_node ∗root, int new_data)

  Insert an element to the front of a linked list.
- void SetDiffLeafListBfromA (struct tree ∗T, int A_index, int B_index)

  Remove the non-unique elements of the reverse-sorted leaf list B from the reverse-sorted leaf list A
- void MergeLeafListBwithA (struct tree ∗T, int A_index, int B_index)

  Merge the elements of reverse-sorted leaf list B into the reverse-sorted leaf list A
- void FreeList (struct list_node ∗root)

  Free the memory stored in a linked list given the pointer root to its root node.
- double ElapsedTime (clock_t start, clock_t stop)

  Return the elapsed time between two clock timers in milliseconds.
- double Deg2Command (double q_deg)

  Convert a joint angle in degrees to its appropriate command value.
- double Command2Deg (double q_cmd)

  Convert a joint angle command back to degrees.
- double ∗∗ Make2DDoubleArray (int nx, int ny)

  Creates a dynamic 2D `double` array of pointers of size nx rows by ny cols.
- int ∗∗ Make2DIntArray (int nx, int ny)

  Creates a dynamic 2D `int` array of pointers of size nx rows by ny cols.
- double ∗∗∗ Make3DDoubleArray (int nx, int ny, int nz)

  Creates a dynamic 3D `double` array of pointers of size nx x ny x nz
- void VectorDiff (double ∗v_left, double ∗v_right, double ∗v_out, int n)

  Compute the vector difference, v_left - v_right, for two `double` vectors of length n
- double Norm (double ∗v, int n, double p)

  Compute the p-norm of a `double` vector v of size n (Ex: p = 1 for Manhattan-norm, p = 2 for Euclidean-norm, or p = `DBL_MAX` for ∞-norm)
- double DistSq (double ∗v1, double ∗v2, int n, double ∗w)

  Compute the weighted Euclidean distance-squared, $d = \left(\sqrt{w \cdot (v_2 - v_1)}\right)^2$ between two vectors of dimension n
- void InsertionSort (double A[], int length, int I[])

  Insertion sort (stable) a `double` vector A (in-place) and return the re-ordered indices I
  *($O(n)$ **best-case** = already sorted, $O(n^2)$ **worst-case** = reverse-sorted, $O(n^2)$ **average time**)*
- void SiftDown (double A[], int root, int bottom, int I[])

  Auxiliary function for `HeapSort` used to float down elements of A into their appropriate place in a heap subtree.
- void HeapSort (double A[], int length, int I[])

  Heap sort (unstable) a `double` vector A (in-place) and return the re-ordered indices I
  *($\Omega(n)$, $O(n\log n)$ **best-case**, $O(n\log n)$ **worst-case**, $O(n\log n)$ **average time**)*
  Modified from the source code found here: `http://www.algorithmist.com/index.php/Heap_sort.c`.
- void Merge (double L[], double R[], int I_L[], int I_R[], int length_L, int length_R, double B[], int J[])

  Auxiliary function for `MergeSort` used to merge two sorted sublists into a combined sorted list.
- void MergeSort (double A[], int length, int I[])

  Merge sort (stable) a `double` vector A (using $O(2n)$ memory) and return the re-ordered indices I
  *($\Omega(n)$, $O(n\log n)$ **best-case**, $O(n\log n)$ **worst-case**, $O(n\log n)$ **average time**)*
  Modified from the source code found here: `http://www.algorithmist.com/index.php/Merge_sort.c`.

- void RearrangeIntVector (int A[], int length, int I[])

  Rearrange integer vector A according to the indices in `int` *vector I*
- void RearrangeIntPtrVector (int ∗A[], int length, int I[])

  Rearrange integer pointer vector A according to the indices in `int` *vector I*
- int SumInts (int ∗A, int n)

  Sum integers in `int` *array A of length n.*
- void Cross (double ∗u, double ∗v, double ∗w)

  Compute the vector cross product, $w = u \times v$, for 3-D `double` *vectors u and v*
- void MatrixMultiply (double ∗∗A, double ∗∗B, int m, int n, int p, double ∗∗C)

  Compute the matrix product of two 2-D `double` *arrays, $A * B$, where A is size $(m \times n)$ and B is size $(n \times p)$.*
- void Matrix3by3Inverse (double ∗∗M, double ∗∗M_inv)

  Compute the inverse of a 3×3 matrix M through calculation of its adjugate, adj $(M) = C^T$, where C is the matrix of co-factors of M.
- void ScalarMultiply (double ∗∗A, double ∗c, int m, int n)

  Compute the scalar multiplication of a 2-D `double` *array, A (size $m \times n$), with* `double` *c.*
- void IdentityMatrix (double ∗∗A, int n)

  Initialize the values of matrix A (size $n \times n$) as an identity matrix I_n.
- void FindRotMat (double ∗v, double ∗v_out, double ∗∗R)

  Find a rotation matrix R from its operand, v, and output, v_{out}, 3×1 column vectors.
- char ∗ Dec2Base (int num, int base, int ∗ptr_N)

  Convert a base-10 integer num to arbitrary base between 2 and 36, returning its string representation and its length N
- int ∗ Dec2BaseInts (int num, int base, int ∗ptr_N)

  Convert a base-10 integer num to arbitrary base between 2 and 36, returning its vector-of-integers representation and its length N
- void Halton (int ∗sequence, int length, int D, double ∗∗h)

  Generates the specified values of the D-dimensional Halton sequence.
- void TransformMatrix (double yaw, double pitch, double roll, double ∗trans, double ∗∗T)

  Compute the homogeneous transformation matrix given yaw-pitch-roll Euler angles.
- void InvTransformMatrix (double yaw, double pitch, double roll, double ∗trans, double ∗∗T)

  Computes the inverse of the transform matrix corresponding to the given yaw-pitch-roll Euler angles and translation vector, transforming the coordinates back to their original frame by reversing the translation and rotation sequence.
- void HomTransformMatrix (double a, double d, double q, double alpha, double ∗∗T)

  Compute the homogeneous transformation matrix for the i^{th} link: $(x,y,z,1)|_{i-1} = T_i\ (x,y,z,1)|_i$.
- void InvHomTransformMatrix (double a, double d, double q, double alpha, double ∗∗T)

  Compute the inverse homogeneous transformation matrix for the i^{th} link: $(x,y,z,1)|_i = T_i^{-1}\ (x,y,z,1)|_{i-1}$.
- void BodyFixedOBBcoords (struct geom ∗G, int ∗n_facepts, double ∗grip_pos, int n)

  Compute the body-fixed coordinates of manipulator Oriented Bounding Boxes (OBB's), including the 8 corner points for each box, the facepoints specified by n_facepts, and the end effector point given by grip_pos
- void WorldCoords (struct geom ∗G, struct DHparams ∗DH, double ∗q, int n, struct coords ∗C)

  Output the coordinates, C, w.r.t. the world frame of the corners and face points of each link's OBB.
- void ConstructTempObstacle (int I, double ∗pos, double radius, double H, double offset, double ∗n_hat, double beta, struct DHparams ∗DH, double ∗q, struct obstacles ∗obs)

  Generate new temperature obstacle.
- void PropagateTemperatures (double ∗∗∗T, double dx, double dy, double dz, int nx, int ny, int nz, double deltaT, double alpha)

  Propagate the temperature map forwards in time by deltaT according to the heat equation (CURRENTLY UNUSED)
- void GenerateInput (char ∗filename, char ∗soln, char ∗sampling, int ∗max_iter, int ∗max_neighbors, double ∗epsilon, int ∗n, int ∗n_waypoints, double ∗q_waypoints, double ∗q_min, double ∗q_max, int ∗grip_actions, double ∗grip_angles, int ∗n_facepts, double ∗L, double ∗W, double ∗H, double ∗rho_x, double ∗rho_y, double ∗rho_z, double ∗d, double ∗a, double ∗alpha, int ∗n_planes, double ∗nhat_planes, double ∗xyz_planes, int ∗n_cylinders, double ∗YPR_cylinders, double ∗xyz_cylinders, double ∗r_cylinders, double ∗H_cylinders, int ∗n_cuboids, double ∗YPR_cuboids, double ∗LWH_cuboids, double ∗xyz_cuboids, char load_input)

Print user input values to file or load values from previous run.

- void GenerateObstacles (struct obstacles *obs, int n_planes, double *nhat_planes, double *xyz_planes, int n_cylinders, double *r_cylinders, double *H_cylinders, double *xyz_cylinders, double *YPR_cylinders, int n_cuboids, double *xyz_cuboids, double *LWH_cuboids, double *YPR_cuboids, int i_grip_obs)

  Generate obstacle primitives from parameters determined by `GenerateInput`
- void GenerateSamples (double **Q, char *sampling, int n, int max_iter, double *q_max, double *q_min, char *filename)

  Generate array of samples, Q.
- void Sample (int feedback_mode, char *sampling, int n, double **Q, double *q_max, double *q_min, int *iter, double *q)

  Determines the next sample joint-angle vector to use for RRT construction.
- double Steer (double *q, double *q_near, int n, double epsilon, double *q_new, double *w)

  Navigation function from configuration q_near to configuration q.
- void SendDoublesToMATLAB (Engine *matlab, int arg_count,...)

  Sends arg_count number of `double` *variables to the MATLAB Engine, entered as a list of names followed by values, e.g. "v1", v1, "v2", v2 (CURRENTLY UNUSED)*
- void Send2DDoubleArraysToMATLAB (Engine *matlab, int arg_count,...)

  Sends arg_count number of real 2D `double` *array variables to the MATLAB Engine, entered as a list of names followed by values, row dimension, and column dimension,*
  e.g. "v1", m1, n1, v1, "v2", m2, n2, v2, etc...
  (NOTE: Assumes rows of v1, v2, ... are each contiguous blocks of memory.
- int SendArraysToMATLAB (int line, Engine *matlab, int arg_count,...)

  Sends real N-D numeric arrays up to $N = 3$ to the MATLAB Engine, saving the variables as formatted numeric matrices.
- void PlotNearestInMATLAB (struct tree *T, int n, double *q, double *q_near, Engine *matlab)

  Adds an illustration of sample node q and the selected node q_near in tree T to the RRT construction figure with handle RRTfig.
- void PlotEdgeInMATLAB (struct tree *T, int n, int node_index, Engine *matlab)

  Adds a straight line plot between a node (index node_index) and its parent to the RRT construction figure with handle RRTfig.
- void PlotRewiringInMATLAB (struct tree *T, int n, int neighbor_index, Engine *matlab)

  Adds a straight line plot between a neighbor node (index neighbor_index) and its new parent to the RRT construction figure with handle RRTfig, deleting the old edge.
- void PlotPathInMATLAB (int plan_index, int current_path_index, int pathlen_new, int n, double **path_new, Engine *matlab)

  Plot a new path segment in MATLAB to figure handle PLANfig.
- void PlotRobotConfigInMATLAB (struct coords *C, int n_points, double opacity, Engine *matlab)

  Plots the current manipulator configuration to figure handle TRAJfig.
- void PlotEndEffectorPathInMATLAB (int n, double epsilon, double *w, int n_cuboids_total, struct obstacles *obs, struct geom *G, struct DHparams *DH, int pathlen_new, double **path_new, Engine *matlab)

  Plots the end effector trajectory corresponding to a given path in figure handle TRAJfig.
- void Nearest (struct tree *T, int num_nodes, double *q, int n, double *w, int cost_type, int *nearest_index, double *q_near, char *NN_alg, Engine *matlab)

  Searches the tree T for the nearest node to q.
- void NearestNeighbors (struct tree *T, int num_nodes, double *q, int n, double *w, int max_neighbors, int cost_type, double eta_RRT, double gamma_RRT, int *neighbors, double *costs, int *n_neighbors, char *N-N_alg)

  Searches the tree T for up to max_neighbors neighbors near q.
- int ConstraintViolation (double *q_new, int n, struct obstacles *obs, struct geom *G, struct DHparams *DH, int obs_indicator)

  Tests a manipulator configuration q_new for violation of constraints.
- void TempObsViolation (struct tree **T, int *num_nodes, int n, struct obstacles *obs, struct geom *G, struct DHparams *DH)

  Identify and mark as unsafe any temperature obstacle violators.

- int Extend (double *q, double *q_near, double epsilon, int n, double *w, double *q_new, struct obstacles *obs, struct geom *G, struct DHparams *DH, double *cost_to_go, int indicator)

  Implements one iteration of `Steer` *and tests for constraint violation, attempting to extend from node q_near towards q.*

- int Connect (double *q, double *q_near, double epsilon, int n, double *w, double *q_new, struct obstacles *obs, struct geom *G, struct DHparams *DH, double *cost_to_go, int indicator)

  Repeatedly calls `Extend` *to incrementally test path safety and attempt to grow a branch from node q_near to q.*

- void InsertNode (struct tree *T, int n, int *num_nodes, double *q_new, int parent_index, double cost_to_go, int connection, list_node *leaf_list, int safety, char *NN_alg, Engine *matlab)

  Adds a new node to an RRT structure.

- void ReWire (int feedback_mode, struct tree *T, int n, double *w, int rewire_node_index, int *neighbors, double *costs_to_neighbors, int n_neighbors, double epsilon, struct obstacles *obs, struct geom *G, struct DHparams *DH, int indicator, Engine *matlab)

  Updates tree branches by re-wiring them into more efficient connections.

- void AddLeafToLists (struct tree *T, int leaf_index)

  Adds leaf_index to all ancestors' leaf lists up to and including the root node.

- int BuildRRTs (struct tree *Ta, struct tree *Tb, int *n_nodesA, int *n_nodesB, int n, double *w, int *iter, int max_iter, char *soln, char *sampling, double **Q, double *q_max, double *q_min, char *NN_alg, int max_neighbors, double eta_RRT, double gamma_RRT, double epsilon, int obs_indicator, struct obstacles *obs, struct geom *G, struct DHparams *DH, int *node_star_index, char load_trees, char *filename, int l, fpos_t *fpos, Engine *matlab)

  Build the Rapidly-Exploring Random Trees (RRT's).

- int * TracePath (struct tree *T, int node1, int node2, int *pathlen)

  Traces the indices along a connected node path.

- void StorePath (struct tree *Ta, struct tree *Tb, int n, int *pathA, int *pathB, int pathlenA, int pathlenB, char *filename, int l, double **path)

  Stores the total sequence of nodes along a connected pair of forward-tree and reverse-tree paths.

- void SplitPathAtNode (double **path, double *q, double *w, int *pathA, int *pathB, int pathlenA, int index, struct tree **T_ptrs, int *n_nodes, int n, char *NN_alg, Engine *matlab)

  Split a path edge at node q and insert it into the appropriate tree.

- void SaveBestPlans (int *num_replans, int max_replans, double cost_to_go, int tree_index, int newpath_start_node, int newpath_end_node, double *replan_costs, int **replan_indices)

  Saves an index representation of the re-plan path to a list of best re-plan paths.

- int RePlan (struct tree **T_ptrs, int *n_nodes, double *q, int n, double *w, int max_replans, int max_replan_neighbors, double eta_RRT, double gamma_RRT, int **replan_indices, char *NN_alg)

  Replan according to the shortest-distance paths that do not currently violate obstacle constraints.

- int FindSafePath (struct tree **T_ptrs, int **replan_indices, int num_replans, int *pathlenA, int *pathlenB, int **pathA, int **pathB, double ***path, double *q, double epsilon, int n, double *w, struct obstacles *obs, struct geom *G, struct DHparams *DH, char *filename)

  Reconstructs the best safe path produced by `RePlan` *and/or* `ExhaustiveRePlan`*.*

- int ExhaustiveRePlan (struct tree **T_ptrs, int *n_nodes, double eta_RRT, double gamma_RRT, char *NN_alg, int *pathlenA, int *pathlenB, int **pathA, int **pathB, double ***path, double *q, double epsilon, int n, double *w, struct obstacles *obs, struct geom *G, struct DHparams *DH, char *filename)

  Replan by searching over all safe nearest neighbors until a feasible solution is discovered (if one exists).

- void ResetSimulation (struct tree *trees, int n_trees, int n_plans, int *num_nodes, int *feasible, int **node_star_index, struct obstacles *obs, Engine *matlab)

  Reset the simulation for a new run.

- int main ()

  Main function for running the manipulator RRT code.

Variables

- CPhidgetAdvancedServoHandle servo = 0

  Declares a (global) Phidgets AdvancedServo handle.
- CPhidgetInterfaceKitHandle ifKit = 0

  Declares a (global) Phidgets InterfaceKit handle.

4.1.1 Detailed Description

Source file for the control code of the Stanford Structures and Composites Lab manipulator arm. Code for controlling the Stanford Structures and Composites Lab manipulator arm. Generates a closed-loop motion plan using a variant of the `RRT*-Connect` Algorithm that has been adapted for fast re-planning and efficient motion plan constraint evaluation.

Written by: Joseph A. Starek, Aero/Astro PhD Student, Autonomous Systems Lab (ASL)

Last updated: 09/25/2012

Original use: Windows 7 x64 OS, Microsoft Visual Studio 2010 Express, MATLAB 2011a Student Version

Assumptions:

- A manipulator arm is tasked with traversing a sequence of known joint angle ("configuration") waypoints
- A single cuboidal object is grasped and dropped during the course of the sequence, in between two of the specified waypoints
- A weighted Euclidean distance-squared metric in joint space (configuration space) is used to represent the cost of motion
- Arm servos are controlled by Phidget 21 control boards
- At most one temperature violation can occur per iteration of a feedback cycle
- Temperature obstacles do not move from their original position and are not removed once introduced.

Units:

- All angles are input in degrees and converted to radians where necessary
- Currently, lengths are given in units of "inches"
- The Phidget controllers rely on internal scales for units, dimensions unknown (standard ranges are given when possible)

Output:

- Data files, determined by the *filename* variable representing the *full path* + *filename*, specified at the top of the `int main()` function
  - <*filename*>input.dat - List of major variables specified by the user, including waypoints, C-space dimension, and run-time parameters
  - <*filename*>obstacles.dat - Static obstacle parameters specified by the user
  - <*filename*>samples.dat - Coordinate file of all sample points used during the generation of the pre-computed RRT's (prior to motion control)
  - <*filename*>trees.dat - Detailed list of contents stored in all pairs of pre-computed RRT's (prior to motion control)
  - <*filename*>output.dat - Reference joint angle commands determined during pre-computation (prior to motion control)

- Various MATLAB plots generated during runtime to provide visualization into the progress and accuracy of the code (can be repressed)
- Feedback motion control of the SACL manipulator arm through a joint angle command history sent to the Phidget 21 control boards

Portability Concerns:

- `SendArraysToMATLAB` uses structured error handling (__try/__except statements) to read data appropriately (catches access violations), which is Windows-specific
- The variables *filename* and *tempsensor_file* defined at the top of `main` must be redefined for any new computer running this code
- For the most part, C-style comments are used; however, some C++ commenting remains, which may be in conflict with some compilers
- Most but not all switch statements use a "default" statement
- Function names are must longer than 6 characters, which can conflict with some systems
- Includes "windows.h", which is compatible only with Windows systems

Instructions for Setup:

1. Install a copy of MATLAB with MATLAB Engine capability (should likely need to be 2010 or later)
   - Set Windows Path variable in Advanced System Settings → Environment Variables:
     C:\Program Files\MATLAB\R2011a Student\bin\win64 (or the path where "libeng.dll" is located)
2. Install the Phidgets21 Library
   - Go to `http://www.phidgets.com/` → Programming tab → C/C++ link → Quick Downloads → **64 bit Windows Driver and Library Files (Zipped)** (or 32 bit if your OS is 32 bit)
   - You should confirm that C: → Program Files → Phidgets contains "phidget21.h" and → x86 contains "phidget21.lib"
3. Install a copy of the latest KD-tree code (`http://code.google.com/p/kdtree/`)
   - Save the folder "kdtree-x.y.z", where x, y, and z are the version numbers, to any directory you like
   - Let's call the directory with this folder *kdtree-dir*
   - Create a "kdtree.obj" file, a static library of KD-tree functions recognizable by MS Visual Studio
     - Open a session of Microsoft Visual Studio 2010 or above.
     - Go to File → New → Project. Select "Win32 Console Application".
     - For "Name", enter "kdtree" and click OK.
     - On the Win32 Application Wizard, click Next and select "DLL" for the Application Type. Check "Empty Project" and click "Finish".
     - In the Solution Explorer pane on the left, right click Header Files → Add → New Item. Browse to *kdtree-dir*\kdtree-x.y.z\kdtree.h and add it to the project.
     - In the Solution Explorer pane on the left, right click Source Files → Add → New Item. Browse to *kdtree-dir*\kdtree-x.y.z\kdtree.c and add it to the project.
     - Under the Solution Configurations bar next to the green arrow, switch "Debug" to "Release". Press F5 or click the green arrow to build the library file.
     - When finished (even if there are some warnings), navigate to your kdtree solution folder for the MS Visual Studio project. Go to → kdtree_DLL → Release and check that the "kdtree.obj" was generated.
   - Move the "kdtree.obj" file to *kdtree-dir*\kdtree-x.y.z
4. Unzip and save the ManipulatorRRT folder to your Documents folder → Visual Studio 2010 → Projects (or wherever the default Projects folder is for your version of MS Visual Studio)

5. Open ManipulatorRRT.sln in Microsoft Visual Studio 2010 or above.
   - In the "Solution Explorer" pane on the left, right-click the ManipulatorRRT project and go to "Properties"
6. Go to Configuration Properties → C/C++ → General → Additional Include Directories
   - Click the drop-down button on the right and select <Edit...>
   - Enter the following in their own fields and select OK:
     - **C:\Program Files (x86)\MATLAB\R2011a Student\extern\include** (or wherever "engine.h" is located)
     - **C:\Program Files\Phidgets** (or wherever "phidget21.h" is located)
     - ***kdtree-dir*\kdtree-x.y.z** (or wherever "kdtree.h" is located)
7. Go to Configuration Properties → Linker → General → Additional Library Directories
   - Click the drop-down button on the right and select <Edit...>
   - Enter the following in their own fields and select OK:
     - **C:\Program Files (x86)\MATLAB\R2011a Student\extern\lib\win32\microsoft** (or wherever "libeng.lib" and "libmx.lib" are located)
     - **C:\Program Files\Phidgets\x86** (or wherever "phidget21.lib" is located)
     - ***kdtree-dir*\kdtree-x.y.z** (or wherever "kdtree.obj" is located)
8. Add the following libraries to Configuration Properties → Linker → Additional Dependencies:
   - Click the drop-down button on the right and select <Edit...>
   - Enter the following in their own fields and select OK:
     - **libeng.lib**
     - **libmx.lib**
     - **phidget21.lib**
     - **kdtree.obj**
9. The code should now be prepared. Plug in the power cord to the Phidget servo control board into an outlet, plug in the USB from the control board into your computer and allow drivers to install. You should see an icon in your taskbar that looks like a "Ph" entitled Phidget Control Panel. If double-clicked, you should see that the device is connected. You can manually adjust servo values here, if necessary, so long as the code is not currently engaging the control board (in which case all you will see is a blank screen with no access to the servo parameters).
10. Set all static user input values in the first section of the `main` function. Select the green button or press F5 to run. Or press CTRL + F5 to run and keep the command window open once finished.

```
You should see the arm developing a motion plan and completing its task!
```

Useful URL's:

- Stanford Autonomous Systems Lab
- C/C++ Code Reference
- kdtree Library
- MATLAB Engine Setup
- Phidget Advanced Servo Reference
- Phidget Interface Kit Reference

4.1.2 Typedef Documentation

4.1.2.1 typedef enum datatypes Type

Enumerated list of datatypes.

Enumerated list of allowable datatypes for the custom function `SendArraysToMATLAB` that transfers C/C++ variables to the MATLAB Engine environment. This is used to send data to MATLAB prior to plotting. Only the datatypes listed here have been coded for transfer. Note some of the datatypes in this list are automatically promoted to alternate datatypes as required by `va_arg`.

4.1.3 Enumeration Type Documentation

4.1.3.1 enum datatypes

Enumerated list of datatypes.

Enumerated list of allowable datatypes for the custom function `SendArraysToMATLAB` that transfers C/C++ variables to the MATLAB Engine environment. This is used to send data to MATLAB prior to plotting. Only the datatypes listed here have been coded for transfer. Note some of the datatypes in this list are automatically promoted to alternate datatypes as required by `va_arg`.

Enumerator:

Long Long integer (saved in MATLAB as uint64)

Short Short integer (saved in MATLAB as uint32, promoted to int)

Char Char (saved in MATLAB as uint32, promoted to int)

Int Signed integer (saved in MATLAB as uint32)

Float Float (saved in MATLAB as double, promoted to double)

Double Double (saved in MATLAB as double)

4.1.4 Function Documentation

4.1.4.1 struct list_node∗ AddListElement (struct list_node ∗ *root,* int *new_data*) `[read]`

Insert an element to the front of a linked list.

Parameters

`in`	*root*	Current root node of the linked list
`in`	*new_data*	Integer data to be inserted

Returns

new_root Pointer to newly-added root node

4.1.4.2 int CCONV AttachHandler (CPhidgetHandle *HANDLE,* void ∗ *userptr*)

Code to execute when a Phidget device has been successfully attached.

4.1.4.3 void BodyFixedOBBcoords (struct geom ∗ *G,* int ∗ *n_facepts,* double ∗ *grip_pos,* int *n*)

Compute the body-fixed coordinates of manipulator Oriented Bounding Boxes (OBB's), including the 8 corner points for each box, the facepoints specified by *n_facepts*, and the end effector point given by *grip_pos*

See Also

geom, WorldCoords, main

Parameters

in	*n_facepts*	Vector containing the numbers of points to use for each pair of OBB faces, given in groups of 3 for each link (faces parallel to xy-, yz-, and xz-planes, respectively). Computed using the Halton sequence, scaled to the link face dimensions.
in	*grip_pos*	1×3 vector in the body-fixed frame of link *n* of the representative end-effector position

4.1.4.4 int BuildRRTs (struct tree * *Ta*, struct tree * *Tb*, int * *n_nodesA*, int * *n_nodesB*, int *n*, double * *w*, int * *iter*, int *max_iter*, char * *soln*, char * *sampling*, double ** *Q*, double * *q_max*, double * *q_min*, char * *NN_alg*, int *max_neighbors*, double *eta_RRT*, double *gamma_RRT*, double *epsilon*, int *obs_indicator*, struct obstacles * *obs*, struct geom * *G*, struct DHparams * *DH*, int * *node_star_index*, char *load_trees*, char * *filename*, int *l*, fpos_t * *fpos*, Engine * *matlab*)

Build the Rapidly-Exploring Random Trees (RRT's).

Constructs RRTs for motion plan *l* using a modified form of the bi-directional RRT-Connect algorithm. Operates in two separate phases, depending on whether it has been called previously or not (as indicated by the value pointed to by *iter*). If unrun before, uses the sample array *Q* to generate samples and waits until the end of all iterations to update node leaf lists. If not, assumes the manipulator is executing closed-loop motion control and continuing to add samples. In this case, one node is added to each tree at a time, and leaf lists must be updated. If it is requested to load trees from a previous simulation, calls <*filename*>trees.dat to upload previous values instead. If the MATLAB Engine pointer *matlab* is not NULL, also generates a plot with handle RRTfig illustrating tree construction.

See Also

Sample, Nearest, NearestNeighbors, Extend, Connect, ReWire, InsertNode, ConstraintViolation, SendArrays-ToMATLAB

Parameters

in,out	*Ta,Tb*	Pointers to the forward and reverse trees. Each must be initialized with the initial and goal joint-angle vectors, respectively. Returned as fully-constructed trees.
in,out	*n_nodesA,n_-nodesB*	Pointers to the number of nodes in each tree. Returned with new values prior to function exit.
in	*n*	Dimensionality of tree nodes, i.e. the number of joint-angles
in,out	*iter*	Pointer to the current sampling iteration
in,out	*node_star_index*	Pointer to each tree's node leaf indices through which the most optimal path passes
in	*load_trees*	User-specification indicating whether to load old trees (Enter 'y' to read values from file, or 'n' to compute using current user settings)
in,out	*fpos*	Pointer to current file position of <*filename*>trees.dat (used to properly load all motion plan trees)

Returns

Boolean indicating whether a feasible path has been found

4.1.4.5 int Connect (double * *q*, double * *q_near*, double *epsilon*, int *n*, double * *w*, double * *q_new*, struct **obstacles** * *obs*, struct **geom** * *G*, struct **DHparams** * *DH*, double * *cost_to_go*, int *indicator*)

Repeatedly calls `Extend` to incrementally test path safety and attempt to grow a branch from node *q_near* to *q*.

See Also

BuildRRTs, Extend, DistSq, ConstraintViolation

Parameters

in	*q*	Target joint-angle vector, of length *n*
in	*q_near*	Nearest joint-angle vector to *q* in the current tree, of length *n*
in,out	*q_new*	Pre-allocated vector of length *n*, used to store new joint-angle configuration
in,out	*cost_to_go*	Returned as pointer to cost-to-go from *q_near* to *q_new*
in	*indicator*	Indicator of obstacle test type (see `ConstraintViolation`)

Returns

The status of `Connect` (must be either *Trapped* (0), or *Reached* (2))

4.1.4.6 int ConstraintViolation (double * *q_new*, int *n*, struct **obstacles** * *obs*, struct **geom** * *G*, struct **DHparams** * *DH*, int *obs_indicator*)

Tests a manipulator configuration *q_new* for violation of constraints.

Determines whether a node *q_new* is safe to add to a tree by testing the manipulator configuration for violation of obstacle constraints. First generates all world-frame coordinates corresponding the new manipulator configuration using `WorldCoords`. Then, depending on the value specified by *obs_indicator*, examines whether any point intersects one of the obstacles stored within *obs*. Conducts tests in order of increasing computational complexity, starting with planes and ending with truncated cones.

See Also

::obs, geom, DHparams, BuildRRTs, WorldCoords

Parameters

in	*q_new*	Query node, of length *n*
in	*obs_indicator*	Indicator of obstacle test type (Enter 0 to test static constraints, 1 to test dynamic constraints, or 2 to test both)

Returns

A boolean indicating violation (1) or not (0)

4.1.4.7 void ConstructTempObstacle (int *l*, double * *pos*, double *radius*, double *H*, double *offset*, double * *n_hat*, double *beta*, struct **DHparams** * *DH*, double * *q*, struct **obstacles** * *obs*)

Generate new temperature obstacle.

Computes new temperature obstacle primitives from the circular sensor region currently centered at pos = (x,y,z) [in] with radius *radius* [in] in the skin surface-normal direction $\hat{n} = (n_x, n_y, n_z)$. All quantities must be defined in the body-fixed frame of link *l* (currently oriented by joint-angle vector *q*). Adds the new obstacle to *obs* as a `temp_obs` struct.

See Also

DHparams, temp_obs, obstacles, TempObsViolation

Parameters

in	*l*	Link frame in which the obstacle, position vector, and normal vector are defined
in	*pos*	Position of maximum-temperature sensor from which the obstacle should emanate
in	*radius,H,beta*	Temperature obstacle cone parameters (radius of frustrum floor [in], total height H [in], and cone half-angle β [deg])
in	*offset*	Height offset above the sensor along its surface normal at which to start the truncated cone
in	*n_hat*	Direction $\hat{n} = (n_x, n_y, n_z)$ of the skin surface normal to be used for the cone axis

4.1.4.8 char∗ Dec2Base (int *num*, int *base*, int ∗ *ptr_N*)

Convert a base-10 integer *num* to arbitrary base between 2 and 36, returning its string representation and its length *N*

See Also

Dec2BaseInts

4.1.4.9 int∗ Dec2BaseInts (int *num*, int *base*, int ∗ *ptr_N*)

Convert a base-10 integer *num* to arbitrary base between 2 and 36, returning its vector-of-integers representation and its length *N*

See Also

Dec2Base

4.1.4.10 int CCONV DetachHandler (CPhidgetHandle *HANDLE*, void ∗ *userptr*)

Code to execute when a Phidget device has been successfully attached.

4.1.4.11 double DistSq (double ∗ *v1*, double ∗ *v2*, int *n*, double ∗ *w*)

Compute the weighted Euclidean distance-squared, $d = \left(\sqrt{w \cdot (v_2 - v_1)}\right)^2$ between two vectors of dimension *n*

Parameters

in	*v1,v2*	Vectors used in difference
in	*n*	Number of elements in each vector
in	*w*	Vector of weighting factors, one for each dimension

4.1.4.12 double ElapsedTime (clock_t *start*, clock_t *stop*)

Return the elapsed time between two clock timers in milliseconds.

Parameters

in	*start*	Beginning timer, generated using `clock()`
in	*stop*	Ending timer, generated using `clock()`

4.1.4.13 int CCONV ErrorHandler (CPhidgetHandle *HANDLE*, void ∗ *userptr*, int *ErrorCode*, const char ∗ *Description*)

Code to execute when an error has occurred while interacting with a Phidget device.

4.1.4.14 int ExhaustiveRePlan (struct tree ∗∗ *T_ptrs*, int ∗ *n_nodes*, double *eta_RRT*, double *gamma_RRT*, char ∗ *NN_alg*, int ∗ *pathlenA*, int ∗ *pathlenB*, int ∗∗ *pathA*, int ∗∗ *pathB*, double ∗∗∗ *path*, double ∗ *q*, double *epsilon*, int *n*, double ∗ *w*, struct obstacles ∗ *obs*, struct geom ∗ *G*, struct DHparams ∗ *DH*, char ∗ *filename*)

Replan by searching over all safe nearest neighbors until a feasible solution is discovered (if one exists).

Feasibility-Priority Search: In the case that the cost-priority search from `RePlan` was unsuccessful, conduct an exhaustive search with focus not on costs but on feasibility. First search for feasible nodes over each tree (listed in order of proximity), then accept the first feasible solution found. Calls `FindSafePath` one path at a time until successful.

See Also

SaveBestPlans, RePlan, FindSafePath

4.1.4.15 int Extend (double ∗ *q*, double ∗ *q_near*, double *epsilon*, int *n*, double ∗ *w*, double ∗ *q_new*, struct obstacles ∗ *obs*, struct geom ∗ *G*, struct DHparams ∗ *DH*, double ∗ *cost_to_go*, int *indicator*)

Implements one iteration of `Steer` and tests for constraint violation, attempting to extend from node *q_near* towards *q*.

See Also

BuildRRTs, Connect, Steer, DistSq, ConstraintViolation

Parameters

in	*q*	Target joint-angle vector, of length *n*
in	*q_near*	Nearest joint-angle vector to *q* in the current tree, of length *n*
in,out	*q_new*	Pre-allocated vector of length *n*, used to store new joint-angle configuration
in,out	*cost_to_go*	Returned as pointer to cost-to-go from *q_near* to *q_new*
in	*indicator*	Indicator of obstacle test type (see `ConstraintViolation`)

Returns

The status of `Extend` (either *Trapped* (0), *Advanced* (1), or *Reached* (2))

4.1.4.16 int FindSafePath (struct tree ∗∗ *T_ptrs*, int ∗∗ *replan_indices*, int *num_replans*, int ∗ *pathlenA*, int ∗ *pathlenB*, int ∗∗ *pathA*, int ∗∗ *pathB*, double ∗∗∗ *path*, double ∗ *q*, double *epsilon*, int *n*, double ∗ *w*, struct obstacles ∗ *obs*, struct geom ∗ *G*, struct DHparams ∗ *DH*, char ∗ *filename*)

Reconstructs the best safe path produced by `RePlan` and/or `ExhaustiveRePlan`.

Finds a safe path (that satisfies obstacle constraints) given the sorted locally-optimal motion plans in *replan_indices* (if one exists). Tests plans sequentially, returning the first feasible path found. Hierarchically determines path safety, first checking node safety, then checking safety of connection from node *q* to the first element of the path, and finally checking the safety of edges within the new path (only requires check against temperature obstacles as the edges already satisfy static obstacle constraints).

See Also

SaveBestPlans, RePlan, ExhaustiveRePlan, StorePath, ConstraintViolation

Parameters

`in`	*replan_indices*	List of indices of re-plan paths (see `SaveBestPlans` and `RePlan`)
`in,out`	*pathlenA,pathlenB*	Current forward and reverse tree pathlengths. Returned with new pathlengths if feasible path found.
`in,out`	*pathA,pathB*	Current forward and reverse tree index paths. Returned with new index paths if feasible path found.
`in,out`	*path*	Current array of nodes. Returned with new path if feasible path found. (see `StorePath`)

Returns

Boolean indicating whether a new safe path has been found

4.1.4.17 void GenerateInput (char ∗ *filename*, char ∗ *soln*, char ∗ *sampling*, int ∗ *max_iter*, int ∗ *max_neighbors*, double ∗ *epsilon*, int ∗ *n*, int ∗ *n_waypoints*, double ∗ *q_waypoints*, double ∗ *q_min*, double ∗ *q_max*, int ∗ *grip_actions*, double ∗ *grip_angles*, int ∗ *n_facepts*, double ∗ *L*, double ∗ *W*, double ∗ *H*, double ∗ *rho_x*, double ∗ *rho_y*, double ∗ *rho_z*, double ∗ *d*, double ∗ *a*, double ∗ *alpha*, int ∗ *n_planes*, double ∗ *nhat_planes*, double ∗ *xyz_planes*, int ∗ *n_cylinders*, double ∗ *YPR_cylinders*, double ∗ *xyz_cylinders*, double ∗ *r_cylinders*, double ∗ *H_cylinders*, int ∗ *n_cuboids*, double ∗ *YPR_cuboids*, double ∗ *LWH_cuboids*, double ∗ *xyz_cuboids*, char *load_input*)

Print user input values to file or load values from previous run.

Generates/calls <*filename*>input.dat and <*filename*>obstacles.dat depending on the setting *load_input* specified by the user. Requires that *filename* fully-specify the path and filename root.

See Also

main

Parameters

`in`	*filename*	String containing the full path + root of input filename
`in`	*load_input*	`char` indicating whether to load a previous file ('y') or save a new one ('n')

4.1.4.18 void GenerateObstacles (struct obstacles ∗ *obs*, int *n_planes*, double ∗ *nhat_planes*, double ∗ *xyz_planes*, int *n_cylinders*, double ∗ *r_cylinders*, double ∗ *H_cylinders*, double ∗ *xyz_cylinders*, double ∗ *YPR_cylinders*, int *n_cuboids*, double ∗ *xyz_cuboids*, double ∗ *LWH_cuboids*, double ∗ *YPR_cuboids*, int *i_grip_obs*)

Generate obstacle primitives from parameters determined by `GenerateInput`

See Also

obstacles, GenerateInput, main

4.1.4.19 void GenerateSamples (double ** *Q*, char * *sampling*, int *n*, int *max_iter*, double * *q_max*, double * *q_min*, char * *filename*)

Generate array of samples, *Q*.

Computes the sample array used during construction of pre-computed RRT's (prior to motion plan execution). Used so that samples can be sent in batch rather than generated individually up to *max_iter* times during the call to `BuildRRTs`. Generates <*filename*>samples.dat

See Also

Sample, Halton

Parameters

`in,out`	*Q*	Pre-allocated `double` array of size $(*\text{max\_iter}* \times n)$, used to store samples
`in`	*sampling*	User specification of sampling method ("pseudorandom" or "halton")
`in`	*q_min,q_max*	Joint angle bounds
`in`	*filename*	String containing the full path + root of input filename

4.1.4.20 void Halton (int * *sequence*, int *length*, int *D*, double ** *h*)

Generates the specified values of the D-dimensional Halton sequence.

Computes the members of the Halton sequence corresponding to the elements of *sequence*, returning a *length* $\times D$ array of `doubles` to array *h*. Each row *m* corresponds to the m^{th} element of *sequence*, while each column *n* corresponds to the dimension, $n = 1, \ldots, D$. See p.207 of "Planning Algorithms" by Steven LaValle.

See Also

Dec2BaseInts, Sample, GenerateSamples

Parameters

`in`	*sequence*	Array of the desired members in the Halton sequence
`in`	*length*	Number of elements in *sequence*
`in`	*D*	Dimension of hypercube used for sequencing (requires $D \leq 32$)
`in,out`	*h*	Pre-allocated array of size *length* $\times D$, populated and returned with Halton samples (by row)

4.1.4.21 void HeapSort (double *A[ ]*, int *length*, int *I[ ]*)

Heap sort (unstable) a `double` vector *A* (in-place) and return the re-ordered indices *I*

($\Omega(n)$, $O(n \log n)$ **best-case**, $O(n \log n)$ **worst-case**, $O(n \log n)$ **average time**)

Modified from the source code found here: http://www.algorithmist.com/index.php/Heap_sort.c.

See Also

SiftDown

Parameters

`in,out`	*A*	Array of elements to be sorted
`in`	*length*	Number of elements in A
`in,out`	*I*	Empty `int` array of same size as A. Returned with the rearranged indices of A.

4.1.4.22 void HomTransformMatrix (double *a*, double *d*, double *q*, double *alpha*, double $**$ *T*)

Compute the homogeneous transformation matrix for the i^{th} link: $(x,y,z,1)|_{i-1} = T_i\,(x,y,z,1)|_i$.

See Also

DHparams, TransformMatrix, InvTransformMatrix, InvHomTransformMatrix

Parameters

in	*a,d,alpha*	DH-parameters for the manipulator arm (constant)
in	*q*	Manipulator joint-angle configuration
in,out	*T*	Pre-allocated array of size 4×4, returned as the output matrix

4.1.4.23 CPhidgetAdvancedServoHandle InitializeServos (int *n*, int $*$ *channels*, int *grip_channel*, double $*$ *AccelThrottle*, double $*$ *VelLimThrottle*, double *grip_AccelThrottle*, double *grip_VelLimThrottle*)

Attempts to connect to robotic arm servomotors and initializes the Phidget advanced servo global variable with user-defined values.

See Also

servo

Parameters

in	*n*	Number of channels (dimension of C-space)
in	*channels*	Channel numbers for joint servos $i = 1,\ldots,n$ with angles q_i
in	*grip_channel*	Channel corresponding to the end effector
in	*AccelThrottle*	Acceleration settings $\in [0,1]$ for each joint servo as a fraction from AccelMin to AccelMax
in	*VelLimThrottle*	Velocity limit settings $\in [0,1]$ for each joint servo as a fraction from VelMin to VelMax
in	*grip_Accel-Throttle*	Acceleration setting $\in [0,1]$ for the end effector as a fraction from AccelMin to AccelMax
in	*grip_VelLim-Throttle*	Velocity limit setting $\in [0,1]$ for the end effector as a fraction from VelMin to VelMax

4.1.4.24 CPhidgetInterfaceKitHandle InitializeTempSensors (int *n_tempsensors*, int $*$ *sensor_channels*, int *rate_tempsensors*)

Attempts to connect to the temperature sensor potentiometer board and initializes the Phidget interface kit global variable with user-defined values.

See Also

ifKit

Parameters

in	*n_tempsensors*	Number of simulated temperature sensors (potentiometer channels)
in	*sensor_channels*	Channel numbers for simulated sensors
in	*rate_-tempsensors*	Data rate (period in milliseconds) to use for reading temperatures

4.1.4.25 void InsertionSort (double *A[]*, int *length*, int *I[]*)

Insertion sort (stable) a `double` vector *A* (in-place) and return the re-ordered indices *I*

($O(n)$ **best-case** = *already sorted*, $O(n^2)$ **worst-case** = reverse-sorted, $O(n^2)$ **average time**)

Parameters

`in,out`	*A*	Array of elements to be sorted
`in`	*length*	Number of elements in A
`in,out`	*I*	Empty `int` array of same size as A. Returned with the rearranged indices of A.

4.1.4.26 void InsertNode (struct tree * *T*, int *n*, int * *num_nodes*, double * *q_new*, int *parent_index*, double *cost_to_go*, int *connection*, list_node * *leaf_list*, int *safety*, char * *NN_alg*, Engine * *matlab*)

Adds a new node to an RRT structure.

Re-allocates memory and inserts new node *q_new* into the tree pointed to by *T*, increasing the number of nodes by 1 and setting the nodes properties. Depending on the NearestNeighbor algorithm indicated by *NN_alg*, also adds the new node to the corresponding KD-tree. If the MATLAB Engine pointer *matlab* is not NULL, adds the node and the edge from its parent to RRTfig.

See Also

tree, BuildRRTs, PlotPathInMATLAB

Parameters

`in,out`	*T*	Pointer to the tree in which the new node should be inserted. Returned with updated tree.
`in,out`	*num_nodes*	Pointer to the number of nodes in the tree. Returned as pointer to updated node count value.
`in`	*q_new*	New node to insert into the tree
`in`	*parent_-index,cost_to_-go,connection,leaf-_list,safety*	Properties of the new node (typically initially assumed to be unconnected (-1), with a NULL leaf list, and safe (1))

4.1.4.27 void InvHomTransformMatrix (double *a*, double *d*, double *q*, double *alpha*, double ** *T*)

Compute the inverse homogeneous transformation matrix for the i^{th} link: $(x,y,z,1)|_i = T_i^{-1}\ (x,y,z,1)|_{i-1}$.

See Also

DHparams, TransformMatrix, InvTransformMatrix, HomTransformMatrix

Parameters

`in`	*a,d,alpha*	DH-parameters for the manipulator arm (constant)
`in`	*q*	Manipulator joint-angle configuration
`in,out`	*T*	Pre-allocated array of size 4×4, returned as the output matrix

4.1.4.28 void InvTransformMatrix (double *yaw*, double *pitch*, double *roll*, double $*$ *trans*, double $**$ *T*)

Computes the inverse of the transform matrix corresponding to the given yaw-pitch-roll Euler angles and translation vector, transforming the coordinates back to their original frame by reversing the translation and rotation sequence.

See Also

TransformMatrix, HomTransformMatrix, InvHomTransformMatrix

Parameters

`in`	*yaw,pitch,roll*	Euler angles of rotation for the original transform about the z-, y-, and x-axes, respectively [deg]
`in`	*trans*	3×1 translation vector for the original transform (applied after rotations), done w.r.t. original axes directions
`in,out`	*T*	Pre-allocated array of size 4×4, returned as the output matrix

4.1.4.29 int main ( )

Main function for running the manipulator RRT code.

Code for controlling the Stanford SACL manipulator. Enter all static input parameters here, including general user settings, goal waypoint profile, manipulator geometry, obstacle parameters, manipulator hardware parameters, and closed-loop simulation parameters.

4.1.4.30 void Merge (double *L[ ]*, double *R[ ]*, int *I_L[ ]*, int *I_R[ ]*, int *length_L*, int *length_R*, double *B[ ]*, int *J[ ]*)

Auxiliary function for `MergeSort` used to merge two sorted sublists into a combined sorted list.

See Also

MergeSort

Parameters

`in`	*L,R*	Left and right subarrays to be merged
`in`	*I_L,I_R*	Indices of the left and right subarrays
`in`	*length_L,length_R*	Lengths of each subarray
`out`	*B*	Combined, sorted array of the elements of L and R (stable merge)
`out`	*J*	Indices of A corresponding to the rearrangement of elements in B

4.1.4.31 void MergeLeafListBwithA (struct tree $*$ *T*, int *A_index*, int *B_index*)

Merge the elements of reverse-sorted leaf list *B* into the reverse-sorted leaf list *A*

Parameters

`in`	*T*	Pointer to RRT
`in`	*A_index*	Index of list A in tree T
`in`	*B_index*	Index of list B in tree T

4.1.4.32 void MergeSort (double *A[]*, int *length*, int *I[]*)

Merge sort (stable) a `double` vector *A* (using $O(2n)$ memory) and return the re-ordered indices *I*

($\Omega(n)$, $O(n\log n)$ **best-case**, $O(n\log n)$ **worst-case**, $O(n\log n)$ **average time**)

Modified from the source code found here: http://www.algorithmist.com/index.php/Merge_sort.c.

See Also

Merge

Parameters

`in,out`	*A*	Array of elements to be sorted
`in`	*length*	Number of elements in A
`in,out`	*I*	Empty `int` array of same size as A. Returned with the rearranged indices of A.

4.1.4.33 void Nearest (struct tree ∗ *T*, int *num_nodes*, double ∗ *q*, int *n*, double ∗ *w*, int *cost_type*, int ∗ *nearest_index*, double ∗ *q_near*, char ∗ *NN_alg*, Engine ∗ *matlab*)

Searches the tree *T* for the nearest node to *q*.

Determines the single nearest-neighbor to joint-angle vector *q* among all *num_nodes* nodes within tree *T*. Proximity is determined by weighted squared-Euclidean-distance for brute-force search or unweighted distance for KD-tree search. Uses either a brute-force search or KD-tree search depending on the specification of the user. The selection is also illustrated by a call to the `PlotNearestInMATLAB` command if the MATLAB Engine pointer *matlab* is not NULL.

See Also

BuildRRTs, DistSq, PlotNearestInMATLAB

Parameters

`in`	*T*	Pointer to the tree to be searched
`in`	*q*	Query node, often a sample generated by `Sample`, of length *n*
`in`	*cost_type*	Indicator of cost-function type (enter 1 for global cost function (*brute-force only*) or 2 for local (greedy) cost function)
`in,out`	*nearest_index*	Returned as pointer to index of nearest-neighbor node
`in,out`	*q_near*	Returned as nearest-neighbor node
`in`	*NN_alg*	User-specification of NearestNeighbor method (either "brute_force" for search over all nodes or "kd_tree" for KD-tree nearest search)

4.1.4.34 void NearestNeighbors (struct tree ∗ *T*, int *num_nodes*, double ∗ *q*, int *n*, double ∗ *w*, int *max_neighbors*, int *cost_type*, double *eta_RRT*, double *gamma_RRT*, int ∗ *neighbors*, double ∗ *costs*, int ∗ *n_neighbors*, char ∗ *NN_alg*)

Searches the tree *T* for up to *max_neighbors* neighbors near *q*.

Determines at most *max_neighbors* nearest-neighbors to joint-angle vector *q* among all *num_nodes* nodes within tree *T*. Proximity is determined by weighted squared-Euclidean-distance for brute-force search or unweighted distance for KD-tree search. Uses either a brute-force search or KD-tree search depending on the specification of the user. KD-trees currently return all nodes within a variable radius, and are unconstrained by *max_neighbors*.

See Also

BuildRRTs, DistSq

Parameters

in	*T*	Pointer to the tree to be searched
in	*q*	Query node, often a sample generated by `Sample`, of length *n*
in	*cost_type*	Indicator of cost-function type (enter 1 for global cost function (*brute-force only*) or 2 for local (greedy) cost function)
in	*eta_RRT*	Maximum radius possible between any two samples produced by `Steer` and stored in the trees
in	*gamma_RRT*	Must choose $\gamma_{RRT} > 2*\left[(1+\frac{1}{n})*(\frac{\mu(C_{free})}{\mu(V_{B_n})})\right]^{(}1/n)$, where μ is the Lebesgue measure, C_{free} is the free configuration space, and V_{B_n} is the volume of a normed ball of dimension *n*
in,out	*neighbors*	Pre-allocated `int` array of length *max_neighbors*. Returned as pointer to the indices of nearest-neighbor nodes, of length *n_neighbors*
in,out	*costs*	Pre-allocated `double` array of length *max_neighbors*. Returned as array of costs to/from *q* from/to each neighbor, of length *n_neighbors*
in,out	*n_neighbors*	Returned as pointer to the number of discovered nearest-neighbors
in	*NN_alg*	User-specification of NearestNeighbor method (either "brute_force" for search over all nodes or "kd_tree" for KD-tree variable radius search)

4.1.4.35 void PlotEdgeInMATLAB (struct **tree** $*$ *T*, int *n*, int *node_index*, Engine $*$ *matlab*)

Adds a straight line plot between a node (index *node_index*) and its parent to the RRT construction figure with handle RRTfig.

(Use C convention, i.e. starting from 0, for *node_index*. Nothing is done if the MATLAB Engine pointer *matlab* is NULL.)

See Also

BuildRRTs, InsertNode

4.1.4.36 void PlotEndEffectorPathInMATLAB (int *n*, double *epsilon*, double $*$ *w*, int *n_cuboids_total*, struct **obstacles** $*$ *obs*, struct **geom** $*$ *G*, struct **DHparams** $*$ *DH*, int *pathlen_new*, double $**$ *path_new*, Engine $*$ *matlab*)

Plots the end effector trajectory corresponding to a given path in figure handle TRAJfig.

Generates the points traced out by the end-effector position along a given joint-angle path using `Steer`, adding a curve of the traced-path plus the planned and former planned paths to TRAJfig. Also overlays a plot of planar obstacles (updated with each plot in order to properly span the axis dimensions), the current temperature obstacles, and removes the grasped object (last cuboidal obstacle) if "plan_index" exceeds the grasp maneuver index. Must be called immediately after `PlotPathInMATLAB` (relies in the MATLAB Engine environment on many of the same variables).

See Also

main, PlotRobotConfigInMATLAB, PlotPathInMATLAB, Steer

Parameters

in	*n_cuboids_total*	Total number of cuboids, including the grasped object. Used to test whether the grasped object has already been eliminated from plots.
in	*pathlen_new*	Length of new path to be added to the plot
in	*path_new*	Nodes along the new path

4.1.4.37 void PlotNearestInMATLAB (struct tree ∗ *T*, int *n*, double ∗ *q*, double ∗ *q_near*, Engine ∗ *matlab*)

Adds an illustration of sample node *q* and the selected node *q_near* in tree *T* to the RRT construction figure with handle RRTfig.

(Nothing is done if the MATLAB Engine pointer *matlab* is NULL.)

See Also

BuildRRTs, Nearest

4.1.4.38 void PlotPathInMATLAB (int *plan_index*, int *current_path_index*, int *pathlen_new*, int *n*, double ∗∗ *path_new*, Engine ∗ *matlab*)

Plot a new path segment in MATLAB to figure handle PLANfig.

Sends *path_new* and *pathlen_new* to MATLAB, merging path[1:1:(pathlen - pathlen_old + current_path_index)] with the new path, and displays the result in figure with handle PLANfig (defined in `main`). "path_old" is the previously planned trajectory for indices beyond "current_path_index", which is instead replaced by path_new. (Use C convention for indices. Nothing is done if the MATLAB Engine pointer *matlab* is NULL.)

See Also

main

Parameters

in	*plan_index*	Index of the current motion plan. Enter as -1 to finalize the plot after motion is over.
in	*current_path_-index*	Entering 0 erases the old plan, while entering "pathlen_oldplan" appends the new one.
in	*pathlen_new*	Length of new path to be added to the plot
in	*path_new*	Nodes along the new path

4.1.4.39 void PlotRewiringInMATLAB (struct tree ∗ *T*, int *n*, int *neighbor_index*, Engine ∗ *matlab*)

Adds a straight line plot between a neighbor node (index *neighbor_index*) and its new parent to the RRT construction figure with handle RRTfig, deleting the old edge.

Assumes q_{rewire} has already been sent to MATLAB from `ReWire` and that "tree" is unchanged from its former definition from `InsertNode` and `PlotEdgeInMATLAB`. (Nothing is done if the MATLAB Engine pointer *matlab* is NULL.)

See Also

BuildRRTs, InsertNode, PlotEdgeInMATLAB

4.1.4.40 void PlotRobotConfigInMATLAB (struct coords ∗ *C*, int *n_points*, double *opacity*, Engine ∗ *matlab*)

Plots the current manipulator configuration to figure handle TRAJfig.

Plots the manipulator configuration corresponding to the coordinate structure C returned by `WorldCoords`, adding a visualization of the arm OBB's and point representation to TRAJfig (Requires previous definition of "coord_format", "coord_color", "link_colors", "link_alpha", and "N_coords". Nothing is done if the MATLAB Engine pointer *matlab* is NULL.)

See Also

main, PlotEndEffectorPathInMATLAB

Parameters

in	*C*	Coordinates in the world-frame of the manipulator arm
in	*n_points*	Total number of points in C
in	*opacity*	Face opacity. Enter 0 for transparent or 1 for full.

4.1.4.41 void PrintListToFile (FILE * *filename*, char * *format*, struct list_node * *ptr*)

Print the elements of a linked list to file.

Parameters

in	*filename*	Pointer to file, generated using `fopen()`
in	*format*	Format string compatible with the `fprintf()` command
in	*ptr*	Pointer to the first element of the linked list

4.1.4.42 int RePlan (struct tree ** *T_ptrs*, int * *n_nodes*, double * *q*, int *n*, double * *w*, int *max_replans*, int *max_replan_neighbors*, double *eta_RRT*, double *gamma_RRT*, int ** *replan_indices*, char * *NN_alg*)

Replan according to the shortest-distance paths that do not currently violate obstacle constraints.

Cost-Priority Search: Generates a sorted list of potential re-plan paths according to the shortest-paths among all possible paths through the set of *max_replan_neighbors* number of nearest-neighbors in each tree. Note this does not conduct a full search, nor does it compute the globally-shortest path (but merely the globally shortest path among the set of locally closest safe nodes).

See Also

SaveBestPlans, FindSafePath, ExhaustiveRePlan

Parameters

in	*max_replans*	Maximum number of cost-priority re-plan paths to search for
in	*max_replan_-neighbors*	Maximum number of replan neighbors to use during search, for each tree (forward and reverse)
in,out	*replan_indices*	Pre-allocated list of size *max_replans* $\times 3$, used to store indices of re-plan paths (see `SaveBestPlans`).

Returns

The number of re-plan paths found and stored in *replan_indices*

4.1.4.43 void ResetSimulation (struct tree * *trees*, int *n_trees*, int *n_plans*, int * *num_nodes*, int * *feasible*, int ** *node_star_index*, struct obstacles * *obs*, Engine * *matlab*)

Reset the simulation for a new run.

Resets node saftey properties to 1, removes any temperature obstacles from previous run, re-defines the new "best" plan by finding the new shortest paths that result from any node and edge additions made during the previous simulation, and closes old plots so that they may be regenerated during the next simulation.

4.1.4.44 void ReWire (int *feedback_mode*, struct tree * *T*, int *n*, double * *w*, int *rewire_node_index*, int * *neighbors*, double * *costs_to_neighbors*, int *n_neighbors*, double *epsilon*, struct obstacles * *obs*, struct geom * *G*, struct DHparams * *DH*, int *indicator*, Engine * *matlab*)

Updates tree branches by re-wiring them into more efficient connections.

As the sampling sequence is relatively arbitrary, new nodes introduced to the tree at later times may turn out to be more efficient parents than previous ones. This means that suboptimal edges must be removed and re-wired to the most recently-added nodes. It can be shown that, so long as searching within an appropriate radius, this re-wiring leads to convergence towards the optimal path. `ReWire` implements this routine. Given *rewire_node_index* as the most-recently added node index, its list of neighbors, and costs of connecting to each one, the edges of each are tested and replaced if the path through the rewire node is found to be more efficient.

See Also

BuildRRTs, SetDiffLeafListBfromA, MergeLeafListBwithA, PlotRewiringInMATLAB, ConstraintViolation

Parameters

`in,out`	*T*	Pointer to tree. Returned with re-wirings and cost updates.
`in`	*feedback_mode*	Boolean specifying whether or not tree-construction is taking place during motion plan execution. If not, leaf list updates need not be made, as those are reserved for after pre-computation. If so, leaf list merges and set differences are required.
`in`	*rewire_node_-index*	Index of the node through which re-wiring takes place. Tested as potential new parent.
`in`	*neighbors*	Indices of nearest-neighbor nodes, of length *n_neighbors*
`in`	*costs_to_-neighbors*	Costs to nearest-neighbor nodes from the rewire node, of length *n_neighbors*
`in`	*indicator*	Indicator of obstacle test type (see `ConstraintViolation`)

4.1.4.45 void Sample (int *feedback_mode*, char * *sampling*, int *n*, double ** *Q*, double * *q_max*, double * *q_min*, int * *iter*, double * *q*)

Determines the next sample joint-angle vector to use for RRT construction.

Uses sample array *Q* during tree pre-computation, or else the method specified by *sampling* during closed-loop motion plan execution.

See Also

GenerateSamples, Halton, Extend, Connect

Parameters

`in`	*feedback_mode*	Boolean specifying whether or not tree-construction is taking place during feedback
`in`	*sampling*	User specification of sampling method ("pseudorandom" or "Halton")
`in`	*Q*	Sample array output from `GenerateSamples`
`in`	*q_min,q_max*	Joint angle bounds
`in`	*iter*	Pointer to the current iteration number, used to prevent redundant samples
`in,out`	*q*	Pre-allocated `double` vector of length *n*, returned as the next sample

4.1.4.46 void SaveBestPlans (int ∗ *num_replans,* int *max_replans,* double *cost_to_go,* int *tree_index,* int *newpath_start_node,* int *newpath_end_node,* double ∗ *replan_costs,* int ∗∗ *replan_indices*)

Saves an index representation of the re-plan path to a list of best re-plan paths.

Given the *cost_to_go* and the tree, starting index, and ending index characterizing the re-plan path, saves its values to the list of best paths found so far. If fewer than *max_replans* number of re-plans have been found, saves the path by default. On the other hand, if *max_replans* number have indeed already been found, maintains the *replan_costs* vector in sorted order by optimality, along with its corresponding *replan_indices* array.

See Also

RePlan, FindSafePath, ExhaustiveRePlan

Parameters

`in,out`	*num_replans*	Current number of re-plans saved. Increments by 1 until at most *max_replans* number of paths have been found.
`in`	*cost_to_go*	Cost-to-go for the current re-plan path.
`in`	*tree_index*	Index representing the type of tree (0 = forward tree, 1 = reverse tree)
`in`	*newpath_start_-node*	Index for the replan path starting node (depends on type of tree)
`in`	*newpath_end_-node*	Index for the replan path ending node (=leaf for forward tree, =root for reverse tree)
`in,out`	*replan_costs*	Vector of length *num_replans* of the costs-to-go for each replan path.
`in,out`	*replan_indices*	List of size *num_replans* $\times 3$, each row corresponding to a different set of replan indices: tree_index, newpath_start_node, newpath_end_node.

4.1.4.47 void Send2DDoubleArraysToMATLAB (Engine ∗ *matlab,* int *arg_count,* ...)

Sends *arg_count* number of real 2D `double` array variables to the MATLAB Engine, entered as a list of names followed by values, row dimension, and column dimension,

e.g. "v1", m1, n1, v1, "v2", m2, n2, v2, etc...

(NOTE: Assumes rows of v1, v2, ... are each contiguous blocks of memory.

Can handle `double` vectors (`double*` or `double[]`) as well, provided they are entered with m = 1.) (CURRE-NTLY UNUSED)

4.1.4.48 int SendArraysToMATLAB (int *line,* Engine ∗ *matlab,* int *arg_count,* ...)

Sends real N-D numeric arrays up to $N = 3$ to the MATLAB Engine, saving the variables as formatted numeric matrices.

Returns an `int` on success/failure.

See Also

datatypes

- Assuming a successful read, all relevant pointers corresponding to the type (tensor, matrix, vector, etc.) have been defined for

4.1.4.49 void SetDiffLeafListBfromA (struct tree ∗ *T,* int *A_index,* int *B_index*)

Remove the non-unique elements of the reverse-sorted leaf list *B* from the reverse-sorted leaf list *A*

Parameters

in	*T*	Pointer to RRT
in	*A_index*	Index of list A in tree T
in	*B_index*	Index of list B in tree T

4.1.4.50 void SiftDown (double *A[ ]*, int *root*, int *bottom*, int *I[ ]*)

Auxiliary function for `HeapSort` used to float down elements of *A* into their appropriate place in a heap subtree.

See Also

HeapSort

Parameters

in,out	*A*	Array of heap elements
in	*root*	Index of subtree root whose element should be floated down
in	*bottom*	Index of the last element in the heap
in,out	*I*	Rearranged indices of A

4.1.4.51 void SplitPathAtNode (double ∗∗ *path*, double ∗ *q*, double ∗ *w*, int ∗ *pathA*, int ∗ *pathB*, int *pathlenA*, int *index*, struct tree ∗∗ *T_ptrs*, int ∗ *n_nodes*, int *n*, char ∗ *NN_alg*, Engine ∗ *matlab*)

Split a path edge at node *q* and insert it into the appropriate tree.

See Also

TracePath, StorePath, InsertNode, MergeLeafListBwithA

4.1.4.52 double Steer (double ∗ *q*, double ∗ *q_near*, int *n*, double *epsilon*, double ∗ *q_new*, double ∗ *w*)

Navigation function from configuration *q_near* to configuration *q*.

Yields the new state *q_new* within weighted distance *epsilon* from *q_near* in the direction of *q*. Currently chooses new configurations along the straight line segment connecting the near and goal states.

See Also

DistSq, Extend, Connect

Parameters

in	*q*	Target joint-angle vector, of length *n*
in	*q_near*	Nearest joint-angle vector to *q* in the current tree, of length *n*
in	*epsilon*	Maximum weighted incremental distance to travel from *q* to *q_near* (terminates if squared-distance is less than ε^2)
in,out	*q_new*	Pre-allocated vector of length *n*, used to store new joint-angle configuration
in	*w*	Vector of weighting factors as defined by the user

Returns

Cost-to-go from *q_near* to *q_new* as `double`

4.1.4.53 void StorePath (struct tree * *Ta*, struct tree * *Tb*, int *n*, int * *pathA*, int * *pathB*, int *pathlenA*, int *pathlenB*, char * *filename*, int *l*, double ** *path*)

Stores the total sequence of nodes along a connected pair of forward-tree and reverse-tree paths.

Traverses from the initial point in *pathA* to the final point in *pathB*, in correct order, saving the nodes encountered along the way. Saves the resulting sequence of nodes to <*filename*>output.dat.

See Also

TracePath

Parameters

in	*Ta,Tb*	Pointers to forward and reverse trees, respectively
in	*pathA,pathB*	Index path arrays from Ta and Tb to-be-merged (see `TracePath`)
in	*pathlen-A,pathlenB*	Lengths of each *pathA* and *pathB*
in	*l*	Index of current motion plan (used to label saved motion plans. Enter -1 to suppress file output).
in,out	*path*	Pre-allocated `double` array of size $((pathlenA{+}{*}pathlenB{*})\ \times n)$, returned with node path

4.1.4.54 void TempObsViolation (struct tree ** *T*, int * *num_nodes*, int *n*, struct obstacles * *obs*, struct geom * *G*, struct DHparams * *DH*)

Identify and mark as unsafe any temperature obstacle violators.

Modifies the "safety" property of tree nodes that are found to reside in or result in inevitable collision with temperature obstacles pointed to by *obs*. Scans and marks the nodes of trees T[0] (forward tree) and T[1] (reverse tree) one-by-one. If the tree is a reverse tree, all decendents leading to an unsafe node are also unsafe, as well as any leaves in the other tree directly connected to it. This is where the benefits of the augmented data "connections" and "leaf_lists" come into play. If the goal node, i.e. root of T[1], is found to be unsafe, then abort the program, as nothing can be done to recover the arm and find a new safe path (all paths lead to collision).

See Also

tree, ::obs, WorldCoords, ConstructTempObstacle

4.1.4.55 int* TracePath (struct tree * *T*, int *node1*, int *node2*, int * *pathlen*)

Traces the indices along a connected node path.

Traces the path from a node *node1* in the tree *T* up to its ancestor *node2* (or root, if it comes first) and returns a pointer to the index path array. Note this path is traced in the backwards direction from what is intended for a forward tree.

See Also

StorePath

Parameters

in	*node1*	Origin node at which to begin the traversal
in	*node2*	Ancestor node at which to stop
in,out	*pathlen*	Pointer to the number of nodes along the tree path

4.1.4.56 void TransformMatrix (double *yaw*, double *pitch*, double *roll*, double $*$ *trans*, double $**$ *T*)

Compute the homogeneous transformation matrix given yaw-pitch-roll Euler angles.

See Also

InvTransformMatrix, HomTransformMatrix, InvHomTransformMatrix

Parameters

in	*yaw,pitch,roll*	Euler angles of rotation about the original z-, y-, and x-axes, respectively [deg]
in	*trans*	3×1 translation vector (applied after rotations), done w.r.t. original axes directions
in,out	*T*	Pre-allocated array of size 4×4, returned as the output matrix

4.1.4.57 void WorldCoords (struct **geom** $*$ *G*, struct **DHparams** $*$ *DH*, double $*$ *q*, int *n*, struct **coords** $*$ *C*)

Output the coordinates, *C*, w.r.t. the world frame of the corners and face points of each link's OBB.

See Also

geom, DHparams, coords, BodyFixedOBBcoords, main

Parameters

in	*q*	Manipulator joint-angle configuration
in	*C*	Pointer to coordinates structure, pre-allocated to hold sum(G->N_coords) of (x,y,z) ordered-pairs

Index