UNIVERSITAT POLITÈCNICA
DE CATALUNYA

Sampling *C*-obstacles border using a filtered deterministic sequence

Jan Rosell, Alexander Pérez.

IOC- Divisió de Robòtica

IOC-DT-P-2008-10
Setembre 2008

Institut d'Organització i Control
de Sistemes Industrials

Sampling $\mathcal{C}$-obstacles border using a filtered deterministic sequence

Jan Rosell and Alexander Pérez

Abstract

This paper is focused on the sampling process for path planners based on probabilistic roadmaps. The paper first analyzes three sampling sources: the random sequence and two deterministic sequences, Halton and $s_d(k)$, and compares them in terms of dispersion, computational efficiency (including the finding of nearest neighbors), and sampling probabilities. Then, based on this analysis and on the recognized success of the Gaussian sampling strategy, the paper proposes a new efficient sampling strategy based on deterministic sampling that also samples more densely near the $\mathcal{C}$-obstacles. The proposal is evaluated and compared with the original Gaussian strategy in both 2D and 3D configuration spaces, giving promising results.

I. INTRODUCTION

The sampling-based approach to path planning consists in the generation of collision-free samples of configuration space ($\mathcal{C}$-space) and in their interconnection with free paths, forming either roadmaps (PRM [1]) or trees (RRT [2]). PRM planners are conceived as multi-query planners, while RRT planners are developed to rapidly solve a single-query problem. These methods are giving very good results, being its success mainly due to its sampling-based nature, i.e. they do not require the explicit characterization of the obstacles of $\mathcal{C}$-space and its efficiency relies on the sample set. Therefore, the generation of samples is one of the crucial factors in the performance of sampling-based planers.

Sampling-based methods based on probabilistic sampling are demonstrated to be probabilistic complete, e.g. for the basic PRM method the number of samples necessary to achieve a probability of failure below a given threshold has been determined [3]. For difficult path-planning problems, however, like those involving narrow passages, this number might be quite large and, therefore, importance sampling methods have been introduced. Those strategies increase the density of sampling in some areas of $\mathcal{C}$-space, thus facilitating the finding of a solution using a reasonable amount of samples. A good classification of importance sampling methods can be found in [4], that divide them by the type of strategy followed: those that bias samples using workspace information (e.g. [5], [6]); those that over-sample the $\mathcal{C}$-space but quickly filter any not-promising configuration (e.g. [7], [8]); those that bias the sampling using the information gathered during the construction of the roadmap or tree (e.g. [9], [10]); and those that deform (dilate) the free $\mathcal{C}$-space to make it more expansive to easily capture its connectivity (e.g. [11], [12]).

Sampling-based methods usually rely on the use of a random number generation source, although the use of deterministic sampling sequences has also been proposed [13]. Deterministic sampling has proven slightly better results than random sampling in roadmap planners for few degrees of freedom tasks [4], [14]. For motion planing purposes, the desired feature of deterministic sampling sequences is to provide a good incremental and uniform coverage of $\mathcal{C}$-space. The Halton sequence [15] is a low-discrepancy sequence that satisfies this requirement. Nevertheless, another useful property is to have a lattice structure to easily allow the determination the neighborhood relations. This can be fulfilled by methods based on multisresolution grids [16], [17]. When this lattice structure is not provided, efficient nearest neighbor searching must be adopted to decrease the computational cost of this much required operation [18].

This work was partially supported by the CICYT projects DPI2005-00112 and DPI2007-63665.

The authors are with the Institute of Industrial and Control Engineering - Technical University of Catalonia, Barcelona, Spain jan.rosell@upc.edu

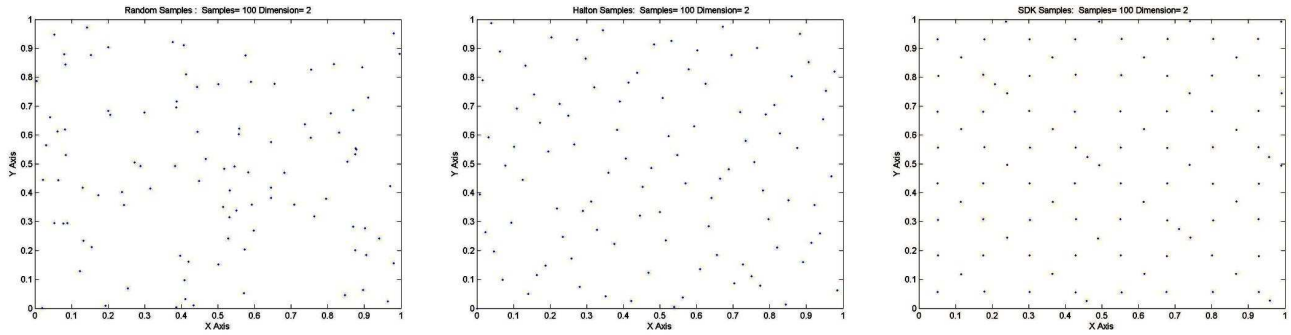


Fig. 1. 100 samples generated using random (left), Halton (middle) and $s_d(k)$ (right) sequences.

II. MOTIVATION, OBJECTIVES AND OVERVIEW

Deterministic sampling has given relative good results in comparison with random sampling when basic sampling-based planners are used (i.e. using uniform distributions), but this advantage diminishes when non-uniform sampling is considered, like the Gaussian sampling strategy, being the selection of a non-uniform sampling the relevant issue rather than the sampling source [4]. However, no sampling strategy has yet taken full profit of the use of a deterministic sampling source. Therefore, the aim of this paper is the proposal of an efficient non-uniform sampling strategy based on deterministic sampling that, like the Gaussian sampling strategy, sample more densely near the $\mathcal{C}$-obstacles.

The paper is structured as follows. Section III makes a comparative study of sampling sources based on dispersion, computational efficiency (including the finding of nearest neighbors), and sampling probabilities. Section IV then proposes the new sampling strategy inspired by the Gaussian sampling strategy and based on the previous study, that is evaluated in Section V. Finally Section VI discusses the results and the contributions of the paper.

III. ANALYSIS OF SAMPLING SOURCES

A. The sampling sources under study

The following three sampling sources are considered to sample a d-dimensional unitary hypercube representing the $\mathcal{C}$-space: the random sequence and the deterministic sequences of Halton [15] and $s_d(k)$ [17].

The random sequence: This sequence is usually obtained using a pseudo-random number generator with long period and good statistical acceptance; for the present study the one provided by [19] is used. A sample of the random sequence is:

$$(x_1, x_2, \ldots, x_d) \text{ with } x_j = \text{rand}\{[0,1)\} \text{ and } j = 1..d \tag{1}$$

The Halton sequence: This deterministic sequence is a d-dimensional generalization of the van der Corput sequence. It is constructed as follows. Let $p_1, p_2, \ldots, p_d$ be d distinct primes, k be the index of the sequence and $a_0 + pa_1 + p^2a_2 + \ldots$ with $a_j \in \{0, 1, \ldots, p\}$ be the base p representation of k. Then, the kth sample of the Halton sequence is[1]:

$$(r_{p_1}, r_{p_2}, \ldots, r_{p_d}) \text{ with } r_{p_i}(k) = \frac{a_0}{p_i} + \frac{a_1}{p_i^2} + \frac{a_2}{p_i^3} + \ldots \tag{2}$$

The $s_d(k)$ sequence: This sequence introduced in [17] is a deterministic sampling sequence based on a multi-grid cell decomposition with an efficient cell coding and on the use of the digital construction method, first proposed in [16].

[1]Implementation available from http://people.scs.fsu.edu/∼ burkardt.

Let M be the maximum partition level (i.e. M-cells have sides of size 2^{-M}). The $s_d(k)$ sequence is a sequence of M-cells; the samples are random configurations within those M-cells. Then, the code of the kth sample is obtained as follows[2]:

$$s_d(k) = (T_d V_k^M) \cdot W'^M \tag{3}$$

where T_d is a $d \times d$ binary matrix, e.g. $T_2 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, V_k^M is the binary $d \times M$ index matrix corresponding to an M-cell with code k, e.g. $V_6^3 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ - see Fig.4 (left)-, and W'^M is a $d \times M$ matrix of weights, with $w'_{ij} = 2^{(j-1)d+i-1}$ for $i \in 1 \dots d$ and $j \in 1 \dots M$, e.g. $W'^3 = \begin{pmatrix} 1 & 4 & 16 \\ 2 & 8 & 32 \end{pmatrix}$.

Let $(w_1^M, \dots, w_d^M)$ be the indices of the cell with code $s_d(k)$. Then the coordinates of the kth sample are randomly chosen within the cell:

$$x_j = \text{rand}\{[w_j^M 2^{-M}, (w_j^M + 1)2^{-M})\} \qquad \forall j \in 1 \dots d \tag{4}$$

When k is greater than 2^{dM} the sequence repeats the cell codes generated, but the samples are different since the configuration coordinates are randomly chosen.

In the implementation done for the present study, a random initial value has been introduced:

$$s'_d(k) = s_d((k + r)\%2^{dM}) \text{ with } r = \text{rand}\{[0, 2^{-dM} - 1)\} \tag{5}$$

$\%$ being the modulo operator.

To illustrate the three sequences considered, Fig. 1 shows the first 100 samples obtained for each of them.

B. Study of dispersion

This section compares the uniformity obtained by the different sampling sources under study. Dispersion is the uniformity measure that best suits path planning purposes since it is a metrics-based criterion that measures the radius of the largest empty hypersphere that can be inscribed in the space being sampled [14]. Let $X = [0, 1]^d \subset \mathbb{R}^d$ be such space and P be the set of samples taken from X. Then, dispersion for P is defined as:

$$\delta(P, \rho) = \sup_{q \in X} \min_{p \in P} \rho(q, p) \tag{6}$$

being ρ any metrics on X.

An analytical expression of the dispersion can be obtained as the function that best fits the normalized and approximate values of dispersion. It is done as follows. First, an approximate value of dispersion is computed (after the generation of each sample in the sequence) by considering a set Q of control points of a fine regular grid:

$$\delta(P, \rho) = \sup_{q \in Q} \min_{p \in P} \rho(q, p) \tag{7}$$

ρ being the Euclidean distance.

Then, in order to make equivalent the values of dispersion for different dimensional spaces, the obtained dispersion is normalized by a factor $\sqrt{d}$. Also, in order to take into account that the higher d the more samples are needed to obtain the same dispersion, a fixed set of N *normalized* samples are considered, i.e. those with index: (int)$(i^{(d/2)})$, $i = 1..N$, (e.g. the 4th normalized sample corresponds to the 4th and to the 8th samples of the sequences $s_2(k)$ and $s_3(k)$, respectively).

[2]The operation $A \cdot B$ represents the scalar product of matrices A and B, and AB is the standard binary matrix multiplication between matrices A and B.

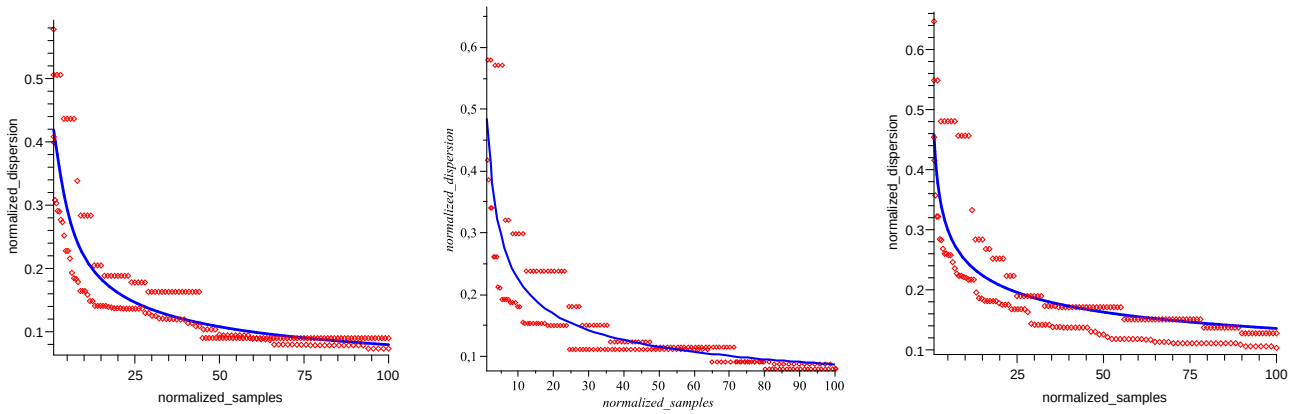


Fig. 2. Analytical curves that fit the dispersion approximated by Eq. (7): $s_d(k)$ (left), Halton (middle) and random (right).

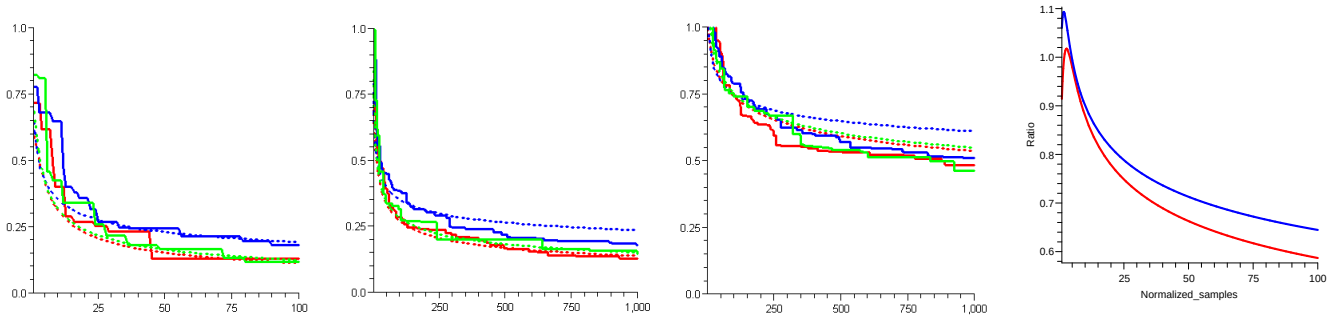


Fig. 3. Dispersion for the random (blue), Halton (green) and $s_d(k)$ (red) sequences for $d = 2$ (left), $d = 3$ (middle-left), $d = 6$ (middle-right). Right: Normalized dispersion ratios.

Results of the computed normalized dispersion for $d = 2$ and $d = 3$ are simultaneously shown in Fig. 2 for the $s_d(k)$, Halton and random sequences. Setting $N = 100$, the number of samples required are 100 and 1000 for $d = 2$ and $d = 3$, respectively. These values are used to fit the following functions (shown in blue in the figures):

$$f_{sdk}(t) = \frac{1}{t^{0.4400}}(0.6036 - 0.4052e^{(-0.7862t)}) \quad (8)$$

$$f_{ran}(t) = \frac{1}{t^{0.2644}}(0.4582 - 0.3210e^{(-11.0076t)}) \quad (9)$$

$$f_{hal}(t) = \frac{1}{t^{0.4095}}(0.5762 - 0.3474e^{(-1.3474t)}) \quad (10)$$

For its uses in a given d-dimensional space, these expressions are denormalized, $f'_{type} = \sqrt{d}\ f_{type}(t^{2/d})$ with $type = \{sdk, ran, hal\}$. As an example, Fig. 3 shows, for different dimensional $\mathcal{C}$-spaces, the dispersion measured and the denormalized analytical approximations of the three sequences.

The ratios f_{sdk}/f_{ran} and f_{hal}/f_{ran} show that both Halton and $s_d(k)$ are better than the random sequence, being the $s_d(k)$ the best one, obtaining a reduction of up to 40% with respect to the random sequence (Fig. 3b).

C. Computational efficiency

This section compares the computational efficiency of the random and the $s_d(k)$ sequences when generating a set of samples and connecting each of them with its K nearest neighbors. The Halton sequence is omitted since the cost of generating samples is higher than in the random case and the same nearest neighbor computation procedure would be used.

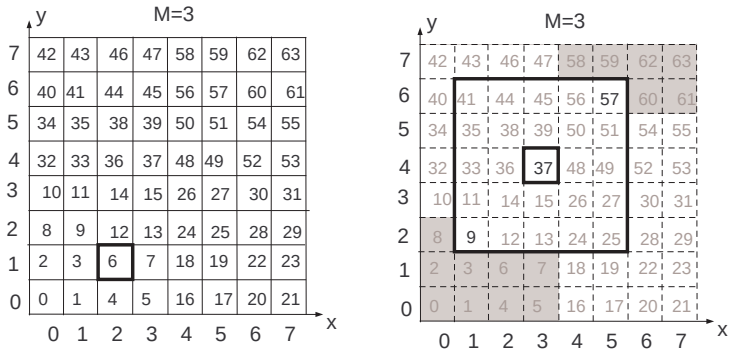


Fig. 4. Cell coding (left); Neighbor box (right).

For the random sequence, the procedure used to find the nearest neighbors is the one available from [18], which is an extension of the algorithm ANN, that is based on an efficient classification of samples within a *kd*-tree. The K-neighbors can be searched within a maximum user-specified distance D. For the $s_d(k)$ sequence, on the other hand, a method is implemented that gives the K neighbors within a given hyperbox, of side s_{box}. Then, for comparison purposes, the distance D has been set to the value of the radius r that makes the volume of the hypershere equal to the hyperbox volume ($V = \frac{2r^d\pi^{(1/2d)}}{d\Gamma(1/2d)} = s_{box}^d$):

$$D = r = \frac{s_{box}}{\sqrt{\pi}} \sqrt[d]{\frac{d}{2}\Gamma(d/2)} \tag{11}$$

The computation of neighbors for the $s_d(k)$ sequence is done as follows. Consider the finding, for a sample contained in an M-cell, of all the neighbors within a hyperbox of side $s_{box} = L\ 2^{-M}$, like cell 37 in Fig. 4 (right) with $L = 5$. Let TR and BL be, respectively, the codes of the top-right and bottom-left corners of the hyperbox (cells 57 and 9 in Fig. 4). Then, the procedure is performed from the list of cell codes in two steps:

1) Prune all those samples that lies within cells with codes greater than TR or lower than BL (gray areas in Fig. 4). Depending on the position of the cell, this step makes a more or less efficient pruning (Fig. 5).
2) For the non-pruned cells in the list determine if they lie within the hyperbox by verifying its indices (x-index between 1 and 5 and y-index between 2 and 6 in the example of Fig. 4)

Tables I and II show the execution times required for the generation of 10,000 samples and the neighbor query for 1,000 of them (using $K = 50$) for $\mathcal{C}$-spaces of 2, 3 and 6 dimensions, run on a computer with Intel Core 2 Duo processor at 1.8Hz. For the random sequence, the whole set of samples is first generated for each experiment and then neighbor queries are answered. Therefore, the construction of the *kd*-tree required by the ANN-based method is only done once and the construction time is not reported. The neighbor computation for the $s_d(k)$ sequence is, on the other hand, computed incrementally, i.e. after the generation of each sample. It can be seen that total time is better for the random sequence (although the difference decreases with dimension), being, on the other hand, the incremental construction a merit of $s_d(k)$.

D. Probabilities

When considering deterministic sequences, like $s_d(k)$, one may think that the probability of sampling a configuration of a given region in n trials can be, for some regions and some value of n, zero. This is in fact not true if the initial sample of the sequence is selected randomly.

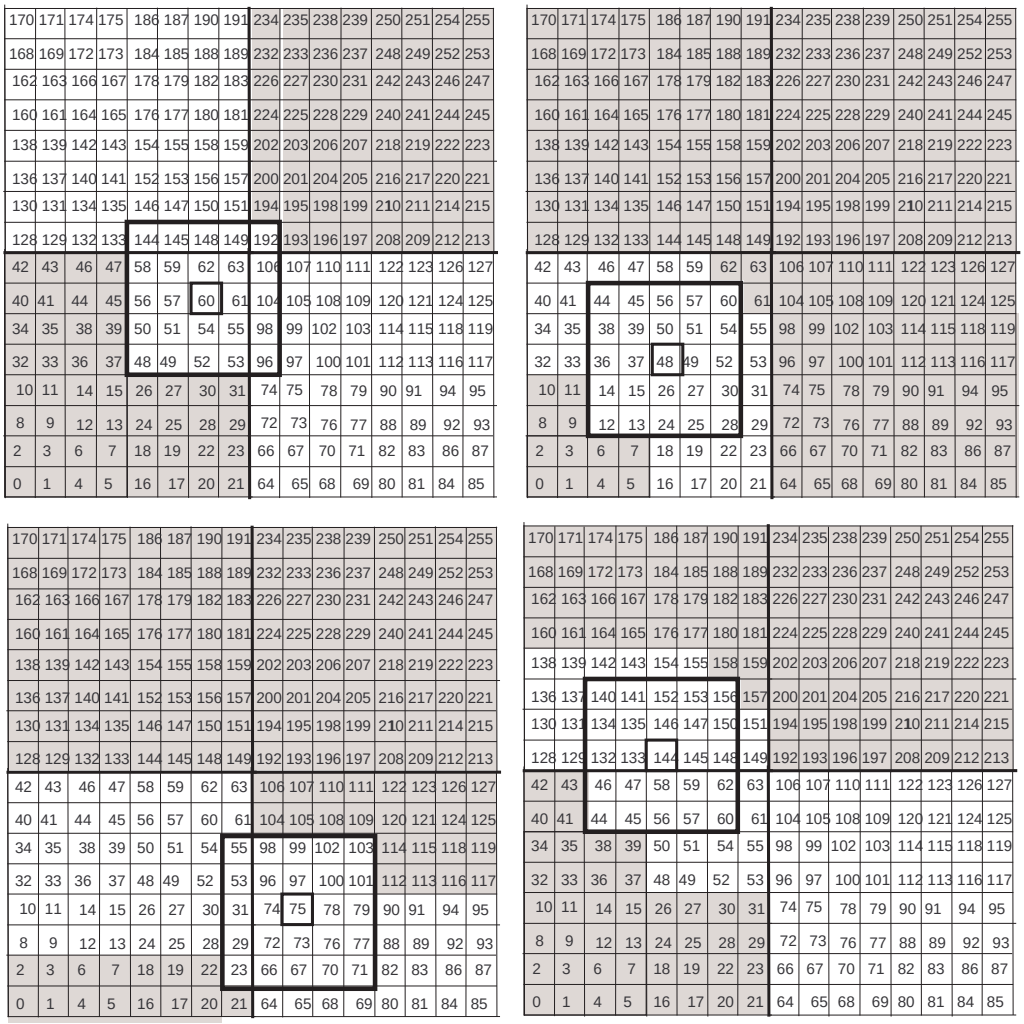


Fig. 5. Pruning in neighbor searching.

TABLE I

COMPUTATION TIMES FOR RANDOM+ANN (ms).

d	2	3	6
Generation	0.7	1.0	1.8
Neigh. search	77.6	115.9	346.0
Total	78.3	116.9	347.8

In order to ease the computation of sampling probabilities, let consider the samples as cells of a very fine regular grid. Let these cells be of side 2^{-S}. Then, for the random sequence, the probability to select a cell in a single throw is:

$$P_{ran_1} = \frac{1}{2^{dS}} \tag{12}$$

and the probability to select a cell in the ith throw is:

$$P_{ran_i} = (1 - P_{ran_1})^{(i-1)} P_{ran_1} \tag{13}$$

TABLE II

COMPUTATION TIMES FOR $s_d(k)$ (ms).

d	2	2	2	3	3	6	6
M	5	6	7	6	7	4	5
Generation	15	15	15	31	31	31	31
Neigh. search	125	125	141	187	188	359	359
Total	140	140	156	218	219	390	390

Finally, the probability to select an S-cell in n throws or less is:

$$P_{ran_{1...n}} = \sum_{i=1}^{n} P_{ran_i} = 1 - (1 - 2^{(-dS)})^n \tag{14}$$

Consider now the $s_d(k)$ sequence, with $M < S$. When an M-cell is selected by the sequence, then an S-cell within the M-cell is randomly chosen. First, the probability of selecting an M-cell in a single throw (i.e. the probability that the cell be the first in the sequence) is:

$$P_M = \frac{1}{2^{dM}} \tag{15}$$

This is also the probability that the M-cell be the jth cell in the sequence (i.e. the cell located jth positions before in the sequence was the one selected as the first in the sequence, with probability P_M).

Second, the probability to select a given S-cell within the M-cell given by the sequence is:

$$P_S = \frac{1}{2^{d(S-M)}} \tag{16}$$

If an S-cell is not selected, then it has to wait until the deterministic sequence covers the whole set of M-cells to have a second chance. Therefore, the probability that an S-cell be chosen in the ith throw is:

$$P_{sdk_i} = P_M(1 - P_S)^{iquo(i,2^{dM})}\ P_S \tag{17}$$

where the function $iquo(a, b)$ returns the integer part of the quocient between a and b. Finally, the probability to select a cell in n throws or less is:

$$P_{sdk_{1...n}} = \sum_{i=1}^{n} P_{sdk_i} \tag{18}$$

Fig. 6 shows the curves of $P_{ran_{1...n}}$ and $P_{sdk_{1...n}}$ obtained with Maple considering $d = 2$, $M = 4$ and $S = 5$. When M is decreased, the probabilities tend to coincide.

IV. THE SDK SAMPLING STRATEGY

As stated in Section II, the aim of this paper is to propose an efficient non-uniform sampling strategy that, like the Gaussian sampling strategy, samples more densely near the $\mathcal{C}$-obstacles. The proposal is called SDK and is based on the deterministic sampling sequence $s_d(k)$. The following three subsections present, respectively, some facts that have guided the proposal, its basic points and the procedure.

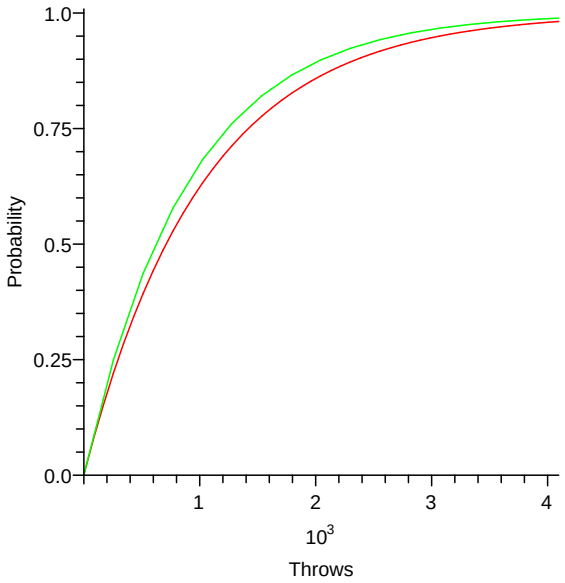


Fig. 6. Sampling probabilities $P_{ran_{1\ldots n}}$ (red) and $P_{sdk_{1\ldots n}}$ (green).

A. *Guides*

The non-uniform sampling provided by the Gaussian strategy is very useful since configurations with poor visibility often lie close to the boundary of the $\mathcal{C}$-obstacles. The Gaussian sampling strategy randomly selects a configuration q and then, within a neighborhood of q determined by a Gaussian measure, randomly selects another configuration q'. If these configurations are both free or both obstacle then they are discarded. Otherwise, the free one is added to the roadmap. In this approach, all sampled configurations are collision-checked, and no information is kept when samples are discarded. Also, the standard deviation σ used is constant and usually determined experimentally with few initial samples. It would be desirable to obtain the same sampling bias with less calls to the collision-detection algorithm.

Model-based approaches, like those based on locally weighted regression methods [20] or on probabilistic cell decompositions [21], [22], use the information of the model to decide weather a sampled configuration needs to be collision-checked or not, i.e. as a lazy evaluation test. This allows to reduce the calls to the collision-detection algorithm, but requires the maintenance of a model of the whole $\mathcal{C}$-space, which can be computationally costly or not possible for many degrees of freedom. It would be desirable to have a similar lazy evaluation method but without the need to maintain the $\mathcal{C}$-space model.

The previous section has demonstrated that the deterministic sequence $s_d(k)$ provides samples with the best dispersion, which is an interesting feature for the exploration of $\mathcal{C}$-space. The computational efficiency has shown not favorable, but the difference is affordable if, as expected, reduction of computational time is obtained by the proper exploitation of the incremental construction of neighbors.

B. *Proposal*

The non-uniform sampling strategy proposed is based on the following points:

- The $s_d(k)$ deterministic sequence is used to provide samples over the $\mathcal{C}$-space.
- K neighbors of each generated sample are computed within the minimum hyperbox (centered at the M-cell containing the sample) such that it guarantees the finding of neighbors. This hyperbox, called *neighbor box* B_i, is of side $s_{box} = L\ 2^{-M}$ with:

$$L = 2^{M-(\text{int})(\frac{1}{d}log_2(k))+1} \tag{19}$$

The decrease of the size of the neighbor box is like that of the dispersion (Fig. 7).

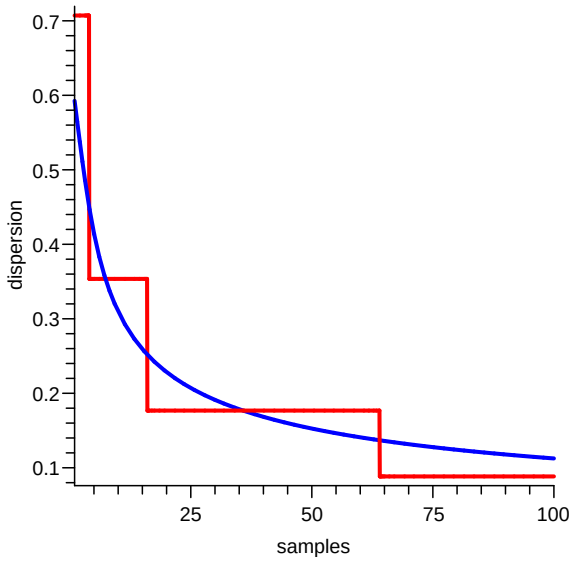


Fig. 7. Neighbor box diagonal (in red) vs. dispersion (denormalized $f_{sdk}(t)$ for $d = 2$ in blue).

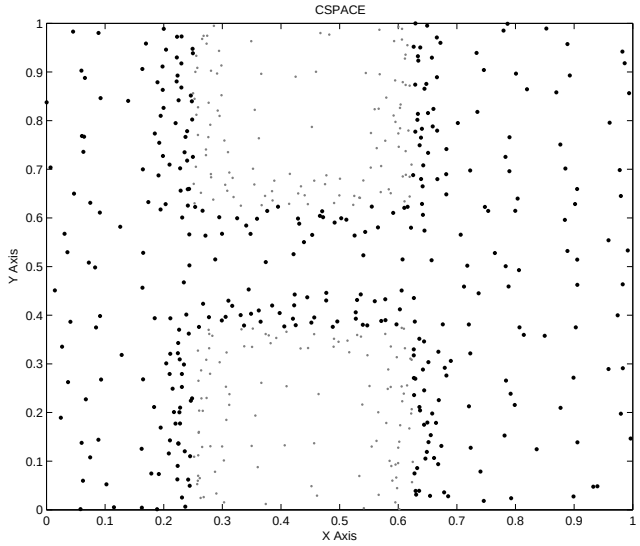


Fig. 8. A 2D $\mathcal{C}$-space sampling using 2,000 generated samples, 617 collision-checks, 376 of them resulting in free configurations (black dots).

- A collision detection algorithm determines weather a configuration is free or not. A parameter, called *color* C_i, is associated to each sample s_i storing the information related to its free or obstacle nature:

$$C_i = \begin{cases} +1 & \text{if } s_i \text{ is a free configuration} \\ 0 & \text{if } s_i \text{ is not collision-checked} \\ -1 & \text{if } s_i \text{ is an obstacle configuration} \end{cases} \tag{20}$$

- The call to the collision detection algorithm depends on the color of the neighbor samples. A parameter, called *transparency* T_i, is defined as the mean value of the color of the neighbors:

$$T_i = \frac{\sum_{j=1}^{j=K} C_j}{K} \tag{21}$$

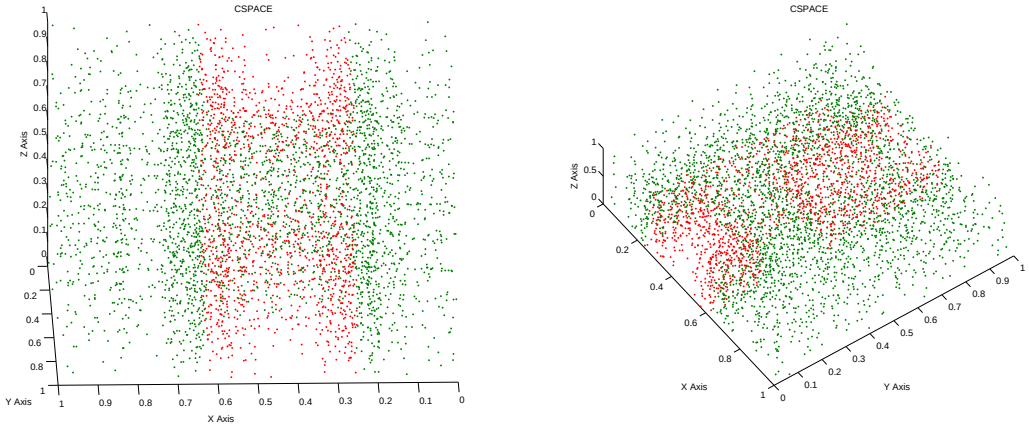


Fig. 9. Two views of a 3D example.

The transparency satisfies $-1 \leq T_j \leq 1$. It is close to zero if there are roughly the same number of free and obstacle samples, and close to one of the extremes if they are mainly either free or obstacle samples. The collision-check will be performed if the transparency lies within an interval, called *uncertainty interval* U, around zero. The size of this interval will be set at two different values depending on weather there are collision-checked neighbor samples with different color or not ($U[1]$ and $U[0]$, respectively). Setting $U[1]$ with a greater size makes greater the chance to collision-check the sample.

C. Procedure

With the basis set on the previous subsection, the procedure of the proposed sampling strategy is as follows:

1) Generate some initial samples until the dispersion is below a given threshold.
2) Loop until a total given number of samples is reached:
   a) Generate a sample s_i
   b) Compute the neighbor box B_i
   c) Find within B_i a maximum number of K evaluated samples and store them as neighbor on a set N_i
   d) Add s_i to the neighbor lists of the samples in N_i, i.e. $\forall s_j \in N_i,\ N_j = N_j \cup s_i$
   e) Compute the transparency T_i
   f) If there are evaluated samples in N_i with different color, then set $f = 1$, otherwise set $f = 0$
   g) If $T_i \in U[f]$ then $C_i = \text{collisioncheck}(s_i)$
   h) Otherwise $C_i = 0$
   i) If $C_i \neq 0$ update the transparency of the samples in N_i and collision-check them if necessary

D. Results

As an example, Fig. 8 shows a 2D $\mathcal{C}$-space generated using $K = 4$, $U[0] = [-0.1, 0.1]$, $U[1] = [-1, 1]$ and $S = 2000$, being 617 of those samples collision-checked (with 376 found free). It can be seen how the proposed method samples more densely near the $\mathcal{C}$-obstacles. Similarly, Fig. 9 shows the sampling of a 3D $\mathcal{C}$-space.

V. COMPARATIVE ANALYSIS

This section compares the proposed strategy with respect to the original Gaussian sampling strategy. For the comparison, the Gaussian method has also been applied to the 2D example, generating as many samples as required until a similar set of free samples is obtained (since the Gaussian strategy does not

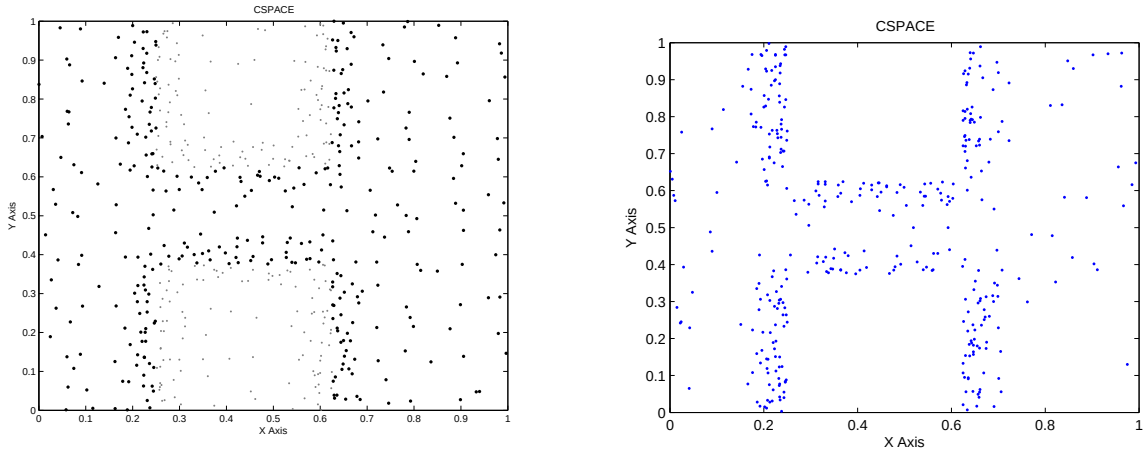


Fig. 10. (Left) The 2D example of Fig. 8. (Right) The 2D example using the Gaussian sampling strategy, using 11,000 samples (generated and collision-checked). The number of output free samples is 362.

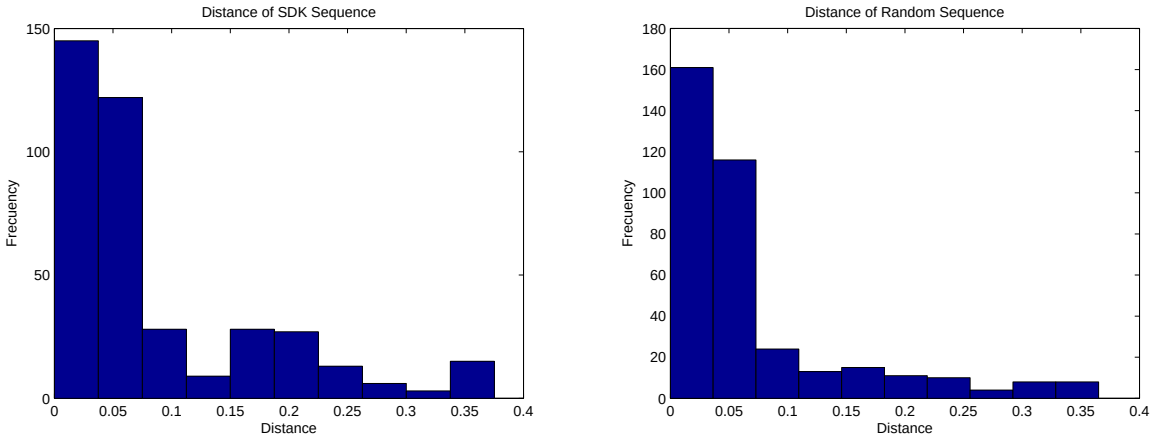


Fig. 11. Histograms of the distances to $\mathcal{C}$-obstacles for the proposed strategy (left) and for the Gaussian one (right).

place samples far away from obstacles, few samples - 1.4% - have been added with uniform sampling in order to also have some configurations on free-space regions). Fig. 10 shows the proposed method on the left and the Gaussian on the right. Similarity of the sampling obtained is verified by the histograms of the distances from the samples to the $\mathcal{C}$-obstacles, as shown in Fig. 11.

By comparing figures 8 and 10 it can be see that the proposed method distributes samples near the $\mathcal{C}$-obstacles more evenly than the Gaussian; the same occurs far from the obstacles where samples are also more evenly distributed than those generated from the uniform sampling.

No collision detection algorithm has been used, but a discretized $\mathcal{C}$-space has been precomputed, giving a negligible computational cost for the collision-check queries. The following table summarizes the results of the comparison, showing for each strategy the number of samples generated, the number of free samples returned, the number of collision-checks performed, and the total time required.

It can be seen that, with negligible collision-check times, the proposed method is slower. Nevertheless, compared to the Gaussian strategy, it only requires a 6% of the calls to the collision-checker. Then, if collisions are to be computed on the workspace using a collision detection algorithm, the proposed method outperforms the Gaussian, as shown in Fig. 12 for collision-check times of up to 1ms and the collision-check requirements of Table III.

TABLE III

COMPARATIVE ANALYSIS.

	# generated samples	# free samples	# collision checks	Time (ms)
SDK	2,000	376	617	46
Gaussian	10,140	370	10,140	16

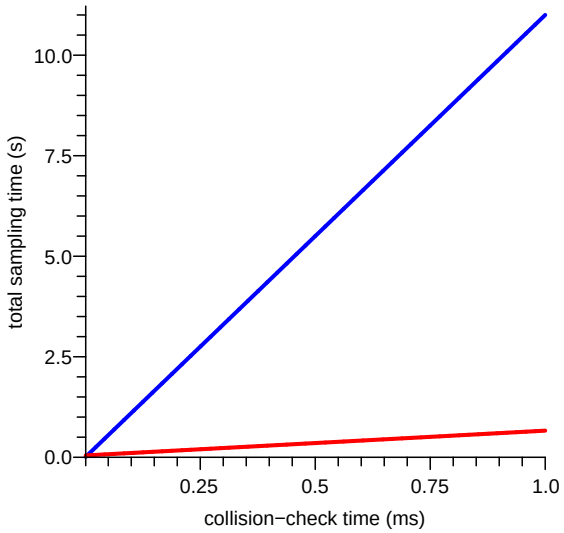


Fig. 12. Total sampling times (in seconds) for the sampling of the 2D example represented with respect to the time to perform a collision check (in milliseconds): the Gaussian sampling strategy in blue, the SDK sampling strategy in red.

VI. CONCLUSIONS

Deterministic sequences have previously been used mainly with uniform sampling, giving just slightly better results than random sampling in PRM path planners. Its use with non-uniform sampling has given no significative improvements. No sampling strategy has yet, however, been designed to take full profit of deterministic sequences. This paper was focused in this direction: it has presented a new and efficient non-uniform sampling strategy, SDK, based on the deterministic sequence $s_d(k)$. Like the Gaussian sampling strategy, this new proposal samples more densely near the $\mathcal{C}$-obstacles, but using much less calls to the collision detection algorithms, resulting in less computational time. Also, samples near the $\mathcal{C}$-obstacles are more evenly distributed and no extra uniform sampling is required since the proposed strategy also puts (few) samples over open-free regions.

REFERENCES

[1] L. E. Kavraki and J.-C. Latombe, "Randomized preprocessing of configuration for fast path planning," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, vol. 3, 1994, pp. 2138–2145.

[2] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2000, pp. 995–1001.

[3] L. E. Kavraki, M. N. Kolountzakis, and J.-C. Latombe, "Analysis of probabilistic roadmaps for path planning," *IEEE Trans. on Robotics and Automation*, vol. 14, no. 1, pp. 166 –171, Feb. 1998.

[4] D. Hsu, J.-C. Latombe, and H. Kurniawati, "On the probabilistic foundations of probabilistic roadmap planning," *Int. Journal of Robotics Research*, vol. 25, no. 7, pp. 627 – 643, 2006.

[5] J. P. van der Berg and M. H. Overmars, "Using workspace information as a guide to non-uniform sampling in probabilistic roadmap planners," *Int. J. of Robotics Res.*, vol. 24 (12), pp. 1055–1071, 2005.

[6] H. Kurniawati and D. Hsu, "Workspace-based connectivity oracle: An adaptive sampling strategy for PRM planning," in *Algorithmic Foundations of Robotics VII*, S. Akella and et.al., Eds. Springer–Verlag, 2006.

[7] V. Boor, M. H. Overmars, and A. F. van der Stappen, "The Gaussian sampling strategy for probabilistic roadmap planners," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 1999, pp. 1018–1023.
[8] D. Hsu, T. Jiang, J. Reif, and Z. Sun, "The bridge test for sampling narrow passages with probabilistic roadmap planners," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2003, pp. 4420–4426.
[9] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. K. Overmars, "Probabilistic roadmaps for path planning in high - dimensional configuration spaces," *IEEE Trans. on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, August 1996.
[10] D. Hsu, G. Sanchez-Ante, and Z. Sun, "Hybrid PRM sampling with a cost-sensitive adaptive strategy," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2005, pp. 3874 – 3880.
[11] M. Saha, J. C. Latombe, Y. C. Chang, and F. Prinz, "Finding narrow passages with probabilistic roadmaps: The small-step retraction method," *Autonomous robots*, vol. 19(3), pp. 301–319, 2005.
[12] H. L. Cheng, D. Hsu, J. C. Latombe, and G. Sanchez-Ante, "Multi-level free space dilation for sampling narrow passages in prm planning," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2006, pp. 1255– 1260.
[13] M. S. Branicky, S. M. LaValle, K. Olson, and L. Yang, "Quasi-randomized path planning," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2001, pp. 1481–1487.
[14] S. M. LaValle, M. S. Branicky, and S. R. Lindemann, "On the relationship between classical grid search and probabilistic roadmaps," *Int. Journal of Robotics Research*, vol. 23, no. 7-8, pp. 673–692, 2004.
[15] J. Halton, "On the effciency of certain quasi-random sequences of points in evaluating multi-dimensional integrals," *Numer. Math.*, vol. 2, pp. 84–90, 1960.
[16] S. R. Lindemann, A. Yershova, and S. M. LaValle, "Incremental grid sampling strategies in robotics," in *Proc. of the Sixth Int. Workshop on the Algorithmic Foundations of Robotics*, 2004, pp. 297 – 312.
[17] J. Rosell, M. Roa, A. Pérez, and F. García, "A general deterministic sequence for sampling d-dimensional configuration spaces," *J. of Intelligent and Robotic Systems*, vol. 50, no. 4, pp. 361–374, 2007.
[18] A. Yershova and S. LaValle, "Improving motion planning algorithms by efficient nearest-neighbor searching," *IEEE Trans. on Robotics*, vol. 23(1), pp. 151 – 157, 2006.
[19] E. S. Rabin, *AI Game Programming Wisdom 2*. Charles River Medis, 2004.
[20] B. Burns and O. Brock, "Sampling-based motion planning using predictive models," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2005, pp. 3131–3136.
[21] J. Rosell and P. Iñiguez, "Path planning using harmonic functions and probabilistic cell decomposition," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2005, pp. 1815–1820.
[22] L. Zhang, Y. J. Kim, and D. Manocha, "A hybrid approach for complete motion planning," in *Accepted to IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2007.

SDK Reference Manual

Generated by Doxygen 1.5.3

Jun 2008

Technical University of Catalonia

Institute of Industrial and Control Engineering

Contents

Chapter 1

SDK: Deterministic Sequence

This is a brief explanation about the S.D.K. deterministic sequence. The S.D.K. is used as a sampler inside a particular Space in the Kautham Planner. This planner was developed in the Institute of Industrial and Control Engineering (IOC, acronym in spanish, www.ioc.upc.edu) from Technical University of Catalonia in Barcelona, Spain (www.upc.edu) in the PhD Programme in Automatic Control, Robotics and Computer Vision.

This sample code is provided "as is". This piece of code is used to show the facilities and potentials of this S.D.K. over other sampling strategies like random or halton sequences.

More information about it, can be found in:

Jan Rosell, Maximo Roa, Alexander Perez, and Fernando Garcia. A general deterministic sequence for sampling d-dimensional configuration spaces. Journal of Intelligent and Robotic Systems, 50(4):361 - 373, December 2007. (omitted accents)

Chapter 2

SDK Namespace Index

2.1 SDK Namespace List

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

Chapter 3

SDK Class Index

3.1 SDK Class List

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

Chapter 4

SDK Namespace Documentation

4.1 SDK Namespace Reference

This namespace contains the deterministic sequence and other useful tools. This namespace is defined to make a compact and reusable piece of code containing the deterministic sequence and many other tool that describes a samplig space (a configuration space for motion planning purpose).

Classes

- class **CSpace**

  This class represents the sampling space. It is a set that contains lists of samples and other useful tools. This class can generates deterministic and random samples, it can defines testing obstacles inside sampling space and it can be asked about sample's neighbours.

- class **Sample**

  This class is the abstraction of a sample entity. This class contains the code, the indexes and the coordinates of a sample and it provides some methods to extract and to use its information in the exploration process.

- class **Sequence**

  This class provides the simply and fast way to use the deterministic sequence. This class is the minimal implementation of the deterministic sequence algorithm. More information about it, will be found in: Jan Rosell, Maximo Roa, Alexander Perez, and Fernando Garcia. A general deterministic sequence for sampling d-dimensional configuration spaces. Journal of Intelligent and Robotic Systems, 50(4):361 - 373, December 2007. (omitted accents).

- class **TMat**

  This class is the abstraction of the dxd binary matrix T_d . This class implements the square (dxd) matrix T_d that is used to find the sequence of 2^d samples of a d-dimensional space that satisfy that the mutual distance is maximized, i.e. the minimum distance to all the previous samples of the sequence is maximized.

- class **WMat**

  This class is the abstraction of the dxM matrix of weights, W. This class is the matrix that it contains the values of weigths.

Chapter 5

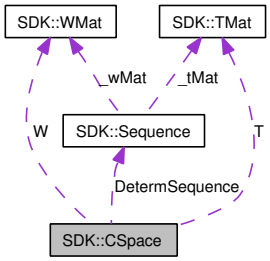
SDK Class Documentation

5.1 SDK::CSpace Class Reference

This class represents the sampling space. It is a set that contains lists of samples and other useful tools. This class can generates deterministic and random samples, it can defines testing obstacles inside sampling space and it can be asked about sample's neighbours.

`#include <cspace.h>`

Collaboration diagram for SDK::CSpace:



Public Member Functions

- **CSpace** (char dimParti, char maxPartion, **WMat** &w, **TMat** &t)

  It is the constructor of space representation. The dimensions, the maximum sampling level, the W and T matrices are the parameters.

- long int **findOrder** (unsigned long code)

  Returns the index of sample with code "code" in the general sampling sequence.

- bool **existSample** (unsigned long code)

  Returns true if there exists a sample with code "code" in the general sample list.

- **TMat** * **getTMat** ()

  Returns a pointer to T matrix.

- **WMat** * **getWMat** ()

Returns a pointer to W matrix.

- void **searchAllNeighs** (int threshold)

  This method searchs all neighs for all existing samples within a threshold distance.

- void **searchNeighs** (**Sample** &smp, int threshold)

  This method searchs all neighs for a particular sample within a threshold distance.

- **Sample** * **nextSample** (bool random=true, bool sorted=false)

  This method makes a new sample. Its parameters are two, the Random parameter that switch between center or any other random coordinates asigned to sample, and the Sorted parameter point out if a new generated sample is putted into a list under time or code order.

- **Sample** * **addSample** (**Sample** &smp, bool sorted=false)

  This method add a sample "smp" into the general sample list. If the Sorted parameter is false, the sample is pushed to the end of the list, otherwise is pushed in code order.

- void **printSamples** (string filename)

  This method writes a file with filename in the run directory with all deterministic samples.

- void **printRandSamples** (string filename)

  This method writes a file with filename in the run directory with all random samples.

- void **printNeighs** (string filename)

  This method writes a file with filename in the run directory with all deterministic samples and their neighs.

- void **printCspace** (string filename)

  This method writes a file with filename in the run directory with all free determinstic samples.

- void **printCspaceRandom** (string filename)

  This method writes a file with filename in the run directory with all free random samples.

- void **printExcel** (string filename)

  This is a convenient method used to write text to be pushed into an Excel spreadsheet.

- int **loadObstacles** (string filename)

  This method loads the sample codes of the c-obstacles of an artificial c-space.

- double * **getTimes** ()

  This method returns the time spent in the generation and the neighbours seach processes.

- int **collisionCheckRand** (long int code)

  This method return the collision status of a code. It is useful to know if an obstacle is in this cell.

- long **exploreRandom** (long numSamples, double radio)

  *This method makes the random exploration of the **CSpace** (p. 9).*

- void **explore** (long numSamples, bool Neighs=true, int threshold=2, bool random=true, bool sorted=false)

*This method makes the deterministic exploration of the **CSpace** (p. 9).*

- void **findingTimes** (long numSamples, long numNeighs, int threshold=2, bool random=true, bool sorted=false)

  This method generate a numSamples of samples and looking for a numNeighs of neighbours, and it measures the time elapsed.

Public Attributes

- **Sequence** * **DetermSequence**

  Pointer to the deterministic sequence object.

Private Attributes

- char **dimProblem**

  *Dimension of a problem. Be sure that Dim*M is less than 32.*

- vector< **Sample** * > **samples**

  List of deterministic samples.

- vector< double * > **randsamples**

  List of random samples.

- **WMat** * **W**

  Pointer to W matrix.

- **TMat** * **T**

  Pointer to T matrix.

- vector< int > **obs3**

  List of c-obstacle cell codes.

- double **_times** [2]

  Array of two times: the generation time and the neighbours search time.

- long **numCollCheck**

  Number of collision checks done in the exploration procedure.

Static Private Attributes

- static LCPRNG * **gen1** = new LCPRNG()

  Pointer to object of LCPRNG class that generates random numbers.

The documentation for this class was generated from the following files:

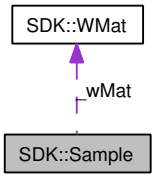
- cspace.h
- cspace.cpp

5.2 SDK::Sample Class Reference

This class is the abstraction of a sample entity. This class contains the code, the indexes and the coordinates of a sample and it provides some methods to extract and to use its information in the exploration process.

`#include <sample.h>`

Collaboration diagram for SDK::Sample:



Public Member Functions

- **Sample** (unsigned long int **code**, char *indexes, bool random=true)

  Unique constructor for a class. Indexes parameter is used for neighbours search.

- unsigned long int **getCode** ()

  Returns the sample code.

- void **searchNeighs** (std::vector< **Sample** * > *candidates, int threshold)

  This method searchs the neighbours of the sample that belong to the vector of candidate samples provided that the partition level of the sample is over a given threshold.

- vector< **Sample** * >::iterator **getNeighs** (**Sample** &smp)

  Returns an iterator to point to neighbours vector of the sample smp.

- string **print** (bool extend=true)

  Returns a string containing the coordinates information and if extend parameter is true, it contains the code and the coordinates information in a explicit form.

- string **printNeighs** ()

  Returns a string that contais the sample code and the codes of neighbour samples.

- char **getFlagT** (void) const

  Returns the flag to be used as index in the threshold transparency vector.

- long **getNumNeighs** ()

  Returns the neighbours number of the sample.

- double **getTransparency** (void) const

  Returns the transparency value of the sample.

- int **getColor** () const

  Returns the color of sample. This color is the collision status. If sample is free the color is 1 otherwise the color is -1. If the sample is not evaluated then the color is 0 .

- void **setColor** (int c)

  This method set the sample color.

- void **updateTNeighs** ()

  This method updates the transparency.

- double **computeTransparency** ()

  This method calculates and returns the value of transparency. This value is calculated based on the neighbourset collision status.

- int **collisionCheck** ()

  This method evaluates the collision status of the sample.

Static Public Attributes

- static float **sizeContainer** = 1.0

  This is the size of an M-Cell.

- static char **DIM** = 1

  This is the dimension of the sampling space.

- static char **M** = 1

  This is the grid partition level.

- static double **thresholdT** [2] = {0.0,0.0}

  This array contains the two threshold transparency levels. This thresholds are used to indicate when the tranparency must be recalculated or not.

- static int **kNeighs** = 4

  This is the number of neighbours considered to calculate the transparency.

- static **WMat** * **_wMat** = NULL

  This is a static pointer to the W matrix. This is a unique object used for any sample in sampleset.

- static vector< int > * **Obst3** = NULL

  This is the static pointer to the obstacles code list. The obstacles are represented for a list with the samples code to be occuped by it.

Private Attributes

- char * **index**

  Pointer to the grid indexes.

- vector< **Sample** * > **neighset**

  Standar vector of pointer to neighbour samples.

- unsigned long int **code**

This is the sample code.

- double * **coord**

  Pointer to the samples coordinates array.

- double **transparency**

  This is the sample transparency value. This is calculated as the mean of the color of its neighs.

- char **flagT**

  This flag is used to indicate that sample has neighbours with different color.

- int **color**

  This is the sample color.

Static Private Attributes

- static LCPRNG * **gen1** = new LCPRNG()

  Pointer to object that generates a random number sequence.

- static int * **topIndex** = NULL

  This is the pointers used to show the neighbour sample with the highest code.

- static int * **lowIndex** = NULL

  This is the pointers used to show the neighbour sample with the lower code.

5.2.1 Member Function Documentation

5.2.1.1 void SDK::Sample::updateTNeighs ()

update the transparency of the neighs and call collisionchsk if needed

5.2.1.2 double SDK::Sample::computeTransparency ()

computeTransparency

The documentation for this class was generated from the following files:

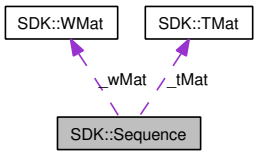
- sample.h
- cspace.cpp
- sample.cpp

5.3 SDK::Sequence Class Reference

This class provides the simply and fast way to use the deterministic sequence. This class is the minimal implementation of the deterministic sequence algorithm. More information about it, will be found in: Jan Rosell, Maximo Roa, Alexander Perez, and Fernando Garcia. A general deterministic sequence for sampling d-dimensional configuration spaces. Journal of Intelligent and Robotic Systems, 50(4):361 - 373, December 2007. (omitted accents).

`#include <sequence.h>`

Collaboration diagram for SDK::Sequence:



Public Member Functions

- **Sequence** (int dim, int M, bool randOffset=true)

  *Simply constructor. This constructor is a simply way to obtain the **SDK** (p. 7) generator. Be careful with dim and M because dim∗M will be less of 32. If randoOffset is true, the first code of sequence is random, otherwise it is zero.*

- ∼**Sequence** (void)

  Simply destructor.

- unsigned long **getCode** (char ∗indexes)

  This method returns the code of the cell with grid coordinates "indexes".

- char ∗ **getIndexes** (unsigned long int code)

  This method returns an array with the indexes of a cell with code "code".

- char ∗ **getIndexes** (void)

  Returns the indexes corresponding to the last generated cell code of the sequence.

- unsigned long **getSequenceCode** ()

  Returns the new code in the sequence.

- unsigned long **getSequenceCode** (unsigned long K)

  Returns the code corresponding to the K^{th} sample of the sequence.

- char ∗∗ **getVMatrix** (unsigned long int code)

  Returns the V matrix for a cell with code "code". V is the matrix of index in binary representation.

- char ∗∗ **getVMatrix** (void)

  Returns the V matrix corresponding to the last sample generated of the sequence.

- void **setW** (**WMat** &w)

This method sets the W matrix.

- **WMat** * **getW** ()

  This method returns a pointer to the W matrix.

- void **setT** (**TMat** &t)

  This method sets the T matrix.

- **TMat** * **getT** ()

  This method returns a pointer to the T matrix.

Private Attributes

- unsigned long int **_index**

  This is the index of the sequence.

- unsigned long int **_lastCode**

  This is the last code generated by the sequence.

- unsigned long int **_offset**

  This is the initial random offset for the sequence.

- unsigned long **_maxNumCells**

  This is the maximum number of cells. It is 2^{Dim*M}.

- char **_maxSamplingLevel**

  This is the M value. This is the maximum sampling level.

- char **_dim**

  This is the dimension of the space to be sampled.

- **TMat** * **_tMat**

  Pointer to T matrix.

- **WMat** * **_wMat**

  Pointer to W matrix.

- char ** **_V**

  This is the pointer to the unique V matrix used to calculate the binary values for indexes of a cell.

- char * **_indexes**

  Pointer to the unique indexes matrix that contains the indexes of a cell.

The documentation for this class was generated from the following files:

- sequence.h
- sequence.cpp

5.4 SDK::TMat Class Reference

This class is the abstraction of the *dxd* binary matrix T_d . This class implements the square (dxd) matrix T_d that is used to find the sequence of 2^d samples of a d-dimensional space that satisfy that the mutual distance is maximized, i.e. the minimum distance to all the previous samples of the sequence is maximized.

`#include <tmat.h>`

Public Member Functions

- **TMat** (int **d**=0)

  This is a unique constructor.

- string **printMatrix** ()

  This method returns a string that contains the text representation of the matrix.

- int **multiply** (const int k)

  This method multiply the matrix for a constant k.

- void **multiply** (const char *const k, char *const l)
- void **multiply** (const char *const k, char *const l, const int m)
- void **multiply** (const char *const w, char *const res) const
- char ** **multiply** (const char *const *const v, const int m) const

  This method multiply the matrix for other matrix V that it corresponds to m level and return the pointer to the result.

Private Member Functions

- void **prime_factorization** (long int x, int *fact, int *numfactors)
- void **compose** (int *primefactors, int numfactors, char **vC, int dimC, int trunc=0)

  This method creates the base matrices of the prime numbers involved.

- void **insert** (char **vA, char **vB, int dimA, int dimB)
- void **createTd** ()

  This method create the matrix properly.

Private Attributes

- int **d**

  This is the dimension of the matrix.

- char ** **_tMat**

  This is the matrix values.

- char ** **matRes**

  Pointer to the matrix that contains the results of any multiplication operation.

5.4.1 Member Function Documentation

5.4.1.1 void SDK::TMat::multiply (const char *const *w*, char *const *res*) const

Multiplies the matrix by a binary vector "w". Returns the result at the parameter "res" A standard matrix-vector operation is performed and then a mod2 operation is done. Therefore the resulting vector is a binary vector.

5.4.1.2 void SDK::TMat::prime_factorization (long int *x*, int * *fact*, int * *numfactors*) [private]

Does the prime factorization of number 'x'. Puts the result in parameter 'fact' and the number of prime factors in parameter 'numfactors'. This function is addapted from Steven S. Skiena (www.programming-challenges.com).

The documentation for this class was generated from the following files:

- tmat.h
- tmat.cpp

5.5 SDK::WMat Class Reference

This class is the abstraction of the dxM matrix of weights, W. This class is the matrix that it contains the values of weigths.

`#include <wmat.h>`

Public Member Functions

- **WMat** (int dim, int level)

  This is the unique constructor provided.

- std::string **printMatrix** ()

  This member method returns a string that contains a text representationm of the matrix.

- void **setRow** (const int index1, const int index2, const long value)

  This method sets the value of the row (index1) and the column (index2) specified.

- long **getRow** (const int index1, const int index2)

  This method returns the value of the row (index1) and the column (index2) specified.

Protected Member Functions

- **WMat** ()

  This is a protected constructor used to restrict the construction way without a correct parameters.

Private Attributes

- int **d**

  This is the dimension of matrix.

- int **m**

  This is the maximum level of samplig.

- long int ** **w**

  This is the matrix values.

The documentation for this class was generated from the following files:

- wmat.h
- wmat.cpp

Index