Karlsruhe Institute of Technology

Carnegie Mellon University

Figuring out How Automats Use AI to Understand and Solve Physical Puzzles

Master's Thesis
of

Xizhe Lian

KIT Department of Informatics
Institute for Anthropomatics and Robotics (IAR)
High Perfomance Humanoid Technologies Lab (H^2T)

Referees:	Prof. Dr. Rüdiger Dillmann
	Prof. Dr. Christopher G. Atkeson
Advisors:	Prof. Dr. Christopher G. Atkeson
	Prof. Dr. Rüdiger Dillmann

Duration: January 1st, 2018 – July 1st, 2018

Statement of Authorship

Aknowledgements:

Here, I would like to thank my advisor Prof. Dr. Christopher G. Atkeson and Prof. Dr. Rüdiger Dillmann for their precious guidance and generous support on my master thesis. It is also highly appreciated that the CLICS programm [1] gave me this valuable opportunity to write my thesis at the Carnegie Mellon Univeristy in Pittsburgh, USA. Particularily, for the helps from Mrs Margit Rödder during the application phase.

Thousand thanks to my colleagues and freinds at the CMU, especially on my labmates at CMU. Akshara Rai, Amarjyoti Smruti, Leonid Keselman, Samuel Clarke and Seungmoon Song, I would not have been able to finish my thesis without your precious help and advices. I would also like to thank my friends, Bettina Weller, MengMeng Yan and Rebecca Seelos, I would not have been able to get this opportunity and finish the thesis without your support.

Last but not least, I would like to thank my parents in particular. Thank you for standing at my back for all my life and respecting every decision I made.

[1] `https://www.clics-network.org/index.php`

Abstract:

Within the scope of this work, we research how to use artificial intelligence (AI) to solve physical puzzles, for instance, how AlphaGO plays the Go game, and how children play with educational physical toys. Physical puzzles are puzzles with essential physical properties, such as gravity or friction. We used magnetic *marble run* as reference in reality. To achieve this, we used computer vision techniques to track the marble ball. Then, we simulated the game in a simulator with Open Dynamics Engine (ODE) and built a dynamic model using this simulator. We ran the optimization algorithms (both gradient-based and non-gradient-based) to find successful solutions for the placement of the movable objects in the simulation. Hence the robot can play such games with the space knowledge of the obstacles.

Kurzzusammenfassung:

Im Rahmen dieser Arbeit erforschen wir, wie künstliche Intelligenz physikalische Rätsel verstehen und lösen kann. Beispiele hierfür wären, wie AlphaGO Go spielt oder wie Kinder mit physikalischen Spielzeugen spielen. Mit phsikalischen Rätseln meinen wir Rätsel, die grundlegende physikalischen Eigenschaften aufweisen, z.B. Gravitation und Reibung. Dabei werden Computer-Vision-Techniken zur Datensammlung verwendet. Ein Video mit einem echten Marbel-Ball-Spielzeug wurde aufgenommen und die Bewegungsbahn der Kugel wurde verfolgt. Mit dieser Bahn haben wir das Spiel in einer Simulation nachgebildet und ein dynamisches Modell daraus erstellt. Mit dem dynamischen Modell und mit Optimierungsalgorithmen wird die Platzierung der beweglichen Spielzeugteile generiert, damit der Marble Ball sein gegebenes Ziel erreichen kann. Ein Roboter kann mit den Informationen über den Raum das Spiel lösen.

Contents

1 Introduction **2**
1.1 Motivation 2
1.2 Problem Statement 3
1.3 Overview 3
1.4 Approaches 3
1.5 Contribution 4

2 Background **5**
2.1 Reinforcement Learning 5
2.2 Optimization Algorithms 5
2.2.1 Gradient-based 5
2.2.2 Black-box Optimization 9
2.3 Separating Axis Theorem 14
2.4 Data analysis 15
2.4.1 Principle Component Analysis (PCA) 15
2.4.2 Linear Discriminant Analysis (LDA) 17

3 Related Work **19**
3.1 Open Dynamics Engine (ODE), dynamic model 19
3.2 Object Detection 19
3.3 Optimization Algorithms 21
3.4 Learning from scratch 21

4 The Approach **23**
4.1 Preparation 23
4.2 Implementation 23
4.2.1 The clean-up step 26
4.2.2 Cost Function 28

5 Experiments **32**
5.1 Computing one valid solution 32
5.1.1 Conjugate Gradient (CG) 32
5.1.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES) 33
5.1.3 Differential Evolution (DE) 33
5.1.4 Random Search (RS) 35
5.1.5 SLSQP 35
5.2 For a robust result : computing ten valid solutions 35
5.2.1 Conjugate Gradient (CG) 37
5.2.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES) 37
5.2.3 Differential Evolution (DE) 40
5.2.4 Random Search (RS) 40
5.2.5 Sequential Least SQuares Programming (SLSQP) 40
5.3 The longer the computational time the better the results ? Computing twenty valid solutions 42
5.3.1 Conjugate Gradient (CG) 42
5.3.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES) 42
5.3.3 Differential Evolution (DE) 45

5.3.4 Random Search (RS) 45
5.3.5 Sequential Least SQuares Programming (SLSQP) 45

6 Evaluation **48**
6.1 Run time and robustness 51
6.1.1 Conjugate Gradient (CG) 51
6.1.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES) 51
6.1.3 Differential Evolution (DE) 52
6.1.4 The random method 52
6.1.5 Sequential Least SQuares Programming (SLSQP) 52
6.2 The path to valid solutions 53
6.2.1 Conjugate Gradient (CG) 53
6.2.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES) 54
6.2.3 Differential Evolution (DE) 54
6.2.4 Random Search (RS) 54
6.2.5 Sequential Least SQuares Programming (SLSQP) 54

7 Summary and future work **76**
7.1 Summary 76
7.2 Future work 76
7.2.1 3D Object Detection 76
7.2.2 Deep neuronal networks and optimization algorithms 77

1 Introduction

In 2015, a human professional Go player was defeated by the computer program *AlphaGo* for the first time that was developed by Google DeepMind. Shortly after that, in March 2016, AlphaGo defeated 18-times world champion Lee Sedol, who is considered to be the greatest player of the past decade. For a long time, the game of GO has been considered as the most challenging games for artificial intelligence (AI) due to its complexity in huge search space and evaluation of board positions and moves, see David Silver1 et al. (2016). The success of AlphaGo has brought back the term *artificial intelligence (AI)* to researchers' attention.

Numerous AI reseachers investigate their effort in AI gaming, for instance, teaching a computer program to play GO or video games such as Atari 2600 games - a collection of arcade games developed by the company Atari, see Mnih et al. (2013), Guo et al. (2014). The popularity does not stand for no reason. Some researcher use games as training ground for the real world. Let's take autonomous mobil for example, the programm has to recongnize signs under all circumstances (rainy, foggy, sunny, etc) and under all kinds of lightings. To gather all these training data in the real world is a tedious task, it is hard to obtain them online as well. Some researchers obtain the wishing data from car driven video games, where realistic signs are featured in games, see Artur Filipowicz and Kornhauser (2017). Further, to play games it needs various cognitive skills. The procedure to solve games can help researchers to understand how the whole task involving intelligence to be resolved into smaller, more manageable subtasks, hence they can resemble AI.

Nowadays, AI has surpassed human-level performance in Atari games. The company OpenAI has transferred the games into Arcade Learning Environments which can be directly used for neural network research[1]. If we train an agent with neural networks for a single Atari game, the agent can beat human performance Mnih et al. (2013). However when we look into the field of AI with common sense, or at artificial general intelligence (AGI), we will find out that there is still a long way to go. Common sense means things or facts that we, as humans, take for granted, learn, and predict from scientifc laws - covering everything from universal physical laws such as ("if the ball drops, it will fall to the ground") to complex social laws. So far, there is no deployed AI system that can answer a broad range of simple common sense questions such as "if I put my socks in the drawer, will they still be there tomorrow ?" Metz (2018).

Hence we want to explore how AI can understand physical phenomena such as gravity or friction, and can solve general physical puzzles by itself . **Marble run**, a.k.a rolling ball sculpture, or ball run, is a perfect tool for the intellectual development of children. When humans design the trajectory of the marble ball, they use their knowledge and understanding of gravity, friction and velocity. It is interesting to see how AI interacts with the real world and plays physical puzzles.

1.1 Motivation

With AI being extremely successful with Go and Atari 2600 games, the question of whether AI can achieve more in games with physical principles has arisen. In the year of 2014, a fourteen-year-old school boy, Robert Nay, released a physical puzzle game application, *the bubble ball*, that has beaten Angry Birds, and hit number one of Apple's App charts that year, as reported by BBC[2]. Meanwhile, in

[1]`http://gym.openai.com/envs/#atari`
[2]`http://www.bbc.co.uk/news/technology-12241564`

the market of marble run as educational toys there are adequate variations. Inspired by the application of Bubble Ball, we chose *marble run* as the starting point for our research on how AI solves physical puzzles. Our ultimate goal is to see how AI understands such puzzles.

1.2 Problem Statement

Our main focus is on investigating how a robot plays magnetic marble run automatically. Magnetic marble run can be clipped on a whiteboard thus the objects can be viewed as two-dimensional regular geometries, which does not have huge requirement on object recognition. Precisely, the robot is given some marble balls, some obstacles and a goal object. The start position is marked and fixed, as well as the goal object. The robot should be able to place the obstacles and free the marble ball.

1.3 Overview

In this thesis, first the background knowledge on current artificial technology in field robotics (Reinforcement Learning (RL)) will be introduced. Then an alternative to classic Reinforcement Learning (RL), optimization algorithms, will be loosely introduced. At the same time, brief introductions on each algorithm deployed in our experiement are presented, which should be helpful for fruther discussion on various algortihms in later chapters. A quick overview on the library of dynamic simulation, the Open Dynamics Engine (ODE), is given. Following, the library on computer vision, the OpenCV, is introduced. OpenCV is an open source library, which covers a large range on topics of computer vision. The part for object detection is presented with details. Subsequently, a pseudo-code of random optimization is shown, that other algorithms have existing implementations. The following chapter describe our approach comprehensively. Pseudo codes of the optimizer and the collision detection are explained, and how to get the best output is depicted as well. Then results in simulation of experiments with diverse optimation algorithms are displayed. An evaluation of run time and output quality of various algorithms is carried out. Meanwhile, data analysis techniques are deployed on the sampling data from each algortihm, which reveals an insight into the optimization landscape of our problem. Last but not least, a summary on this thesis and future work are presented.

1.4 Approaches

In our approach to implement the project, the first step would be that the robot needs to "see" or "recognize" the parts of the game, e.g., object detection or object recognition. The next step would be to build a simulation environment, where the agent or computer program can play the physical puzzles. For an accurate simulation environment, a real marble run is built and the ball is run. The ball trajectory is recorded and is compared with the simulated ball run. The parameters of the simulation environment are tuned so that the two ball trajectories resemble each other.

We use the word "layout" to refer to the placement of the parts (and later we will call them obstacles) in the game. Letting the robot play the game is equivalent to letting it generate a layout. To compute a layout, the *Optimization Algorithms* are applied, which are inspired by the lecture of Atkeson (2018), and the implementation is based on the assignments from the course [3]. Last but not least, after completion of the calibration process to align the simulation environment and the real playing environment, the robot is ready to play the physical puzzle game. I.e, the robot will locate the initial position of the ball and the position of the target object. It will place the available obstacles in the real world according to the output of the optimizer, and then free the ball.

[3]`https://github.com/leonidk/Box2D/`

1.5 Contribution

In this thesis, we figured out a framework for a robot to play basic marble run. Figure 1.1 illustrates the flow of the framework. Roughly speaking, the optimizer is designed for finding a valid solution, where the solution can be extended to be directly deployed in the real world. The robot just needs to move the obstacles to the desired positions according to the output of the optimizer.

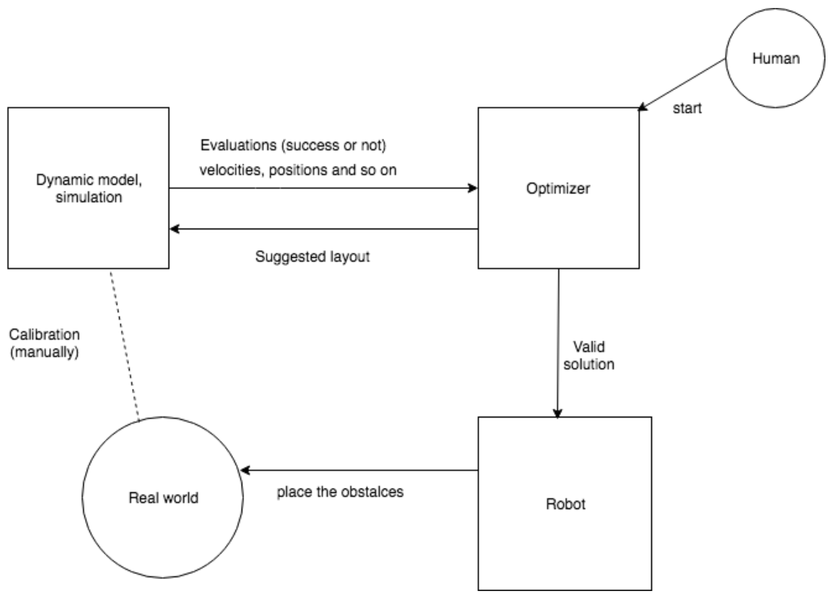


Figure 1.1: A sketch about the framwork for a robot to play basic marble run

2 Background

This chapter gives a basic overview of *Reinforcement Learning (RL)* in the gaming area. Followed by some brief introductions on *optimization algorithms*, and a theorem in geometry - the separating axis theorem- that was used for implementation is explained. At the end a precise explaination on data analysis tools - the Principle Component Analysis (PCA) and Linear Discriminant Analysis (LDA) - is presented.

2.1 Reinforcement Learning

The classic approach of RL algorithms is based on the *Markov Decision Process (MDP)* formalism and the concept of value functions and reward systems. The agent, which is an autonomous or semi-autonomous AI-driven system, is trained to select the best action, where the expectation value of taking a particular action is represented by a Q-network. A Q-network is nothing else than a lookup table for rewards of every state-action pair. Successful representations of this approach include systems that learn to play Atari from pixels Mnih et al. (2013), or the impressive AlphaGo, David Silver1 et al. (2016)

The use of optimization algorithms, on the other hand, has been considered an alternative to solve RL problems. More than a decade ago, this approach was already known as the direct policy search, Schmidhuber and Zhao (1998). If the neuronal network is trained by evolutionary algorithms, a subset of black-box optimization, then this approach is called neuro-evolution by Risi and Togelius (2015). These optimization algorithms have come back to researchers' attention as they have attractive advantages - for example they are highly parallelizable and data efficient - as compared to deep RL networks.

2.2 Optimization Algorithms

Optimization algorithms solve problems with the formula:

$$\min_{x \in S} f(x)$$

i.e, with a given a set and an objective function or cost function, to find an element x_0 out of it so that the function value of x_0 is smaller or equal than all other elements' function values. When we say optimization problems we refer to find the minimum, because the maximization problem can be calculated based on the minimization problem. A mathematical expression would be:

$$find\, x_0 \in S\ s.t. \quad \forall\, x \in S\ :\ f(x_0)\ \leq\ f(x)$$

Patryk Chrabaszcz (2018) found out that with a simple variation of the Evolution Strategies (ES) the agent will be able to play Atari games as well. Some researches find out that Genetic Algorithm (GA) has been doing well on finding weights for large Deep artificial Neuronal Networks (DNNs), Such et al. (2018).

2.2.1 Gradient-based

Gradient-based method follows the gradient of the given objective function at the current point to find the optimum of the function. In this section two gradient-based methods, the Conjugate Gradient (CG) and the Sequential Least SQuares Programming (SLSQP), are briefly introduced.

Conjugate Gradient (CG)

A solid acknowledgment of *linear algera* is a precondition to understand the CG algorithm. More details can be found in literature that provides some good explanations, for example Shewchuk (1994). The method is originally proposed by Hestenes and Stiefel (1952) to solve linear systems using the formula of n linear equations:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.1}$$

An important mathematical term that needs to be clarified is *conjugate*. Two vectors d_i and d_j are conjugate to some semi-definite matrix A , if

$$\mathbf{d}_i^T \mathbf{A} \mathbf{d}_j = 0 \tag{2.2}$$

This is a.k.a. A-orthogonal, as shown in subfigure (a) of figure 2.1, and the pairs of vectors are **A**-orthogonal, because they are orthogonal in subfigure (b). The space in the subgraph 2.1 (b) is spanned by elementary vectors $\{\mathbf{e}_1, \mathbf{e}_2, \dots \mathbf{e}_n\}$, denoted as $\mathbf{E}_n$. The matrix **A** transforms the standard circle form of subgraph 2.1 (b) into an ellipse in subgraph 2.1 (a), the space of (a) can be written as $\mathbf{AE}_n$, which is basically A. So, if the two vectors $\mathbf{d}_i$ and $\mathbf{d}_j$ are orthogonal in elementary space, it will be :

$$\mathbf{d}_i^T \mathbf{d}_j = 0 \tag{2.3}$$

Analogously, in the transformed space from A, we will have the equation (2.2) for those orthogonal pairs in the standard space that is spanned by elementary vectors.

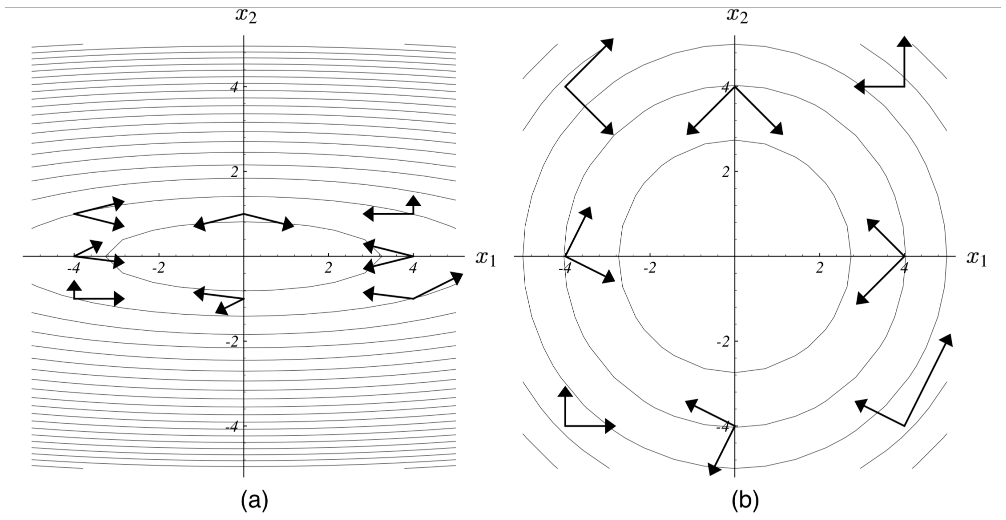


Figure 2.1: A-orthogonal and orthogonal, figure credicts to Shewchuk (1994)

Like every other gradient-based algorithm, at each step, CG will take a direction $\mathbf{d}_i$ with a step size β_i to update the current value of $\mathbf{x}$:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \beta_i \mathbf{d}_i, \quad i = 0, \dots, n-1 \tag{2.4}$$

The equation 2.4 also implies that the algorithm will terminate in n steps, where n is the dimension of the problem, i.e., the dimension of the matrix **A** and the target vector **b**. The reader who wants to know the details about the termination and the correctness of the algorithm in more detail is referred to the literature Shewchuk (1994).

Moreover, what differs CG from other gradient-based algorithms is that the directions $\{\mathbf{d}_1, ..., \mathbf{d}_n\}$ are well selected so that they are pairwise orthogonal. They are constructed by the conjugation of the current residual, where the residual is the distance from the current point (which is in vector form) to the vector $\mathbf{b}$:

$$\mathbf{r}_i = \mathbf{b} - \mathbf{A}\,\mathbf{x}_i \tag{2.5}$$

Here we introduce the *error term* $\mathbf{e}$ to denote the difference between the current estimate of $\mathbf{x}$ to the minimum point $\mathbf{x}_*$, where with point $\mathbf{x}_*$ the function derivative is 0 :

$$\mathbf{e}_i = \mathbf{x}_i - \mathbf{x}_* \tag{2.6}$$

The correlation between the error term $\mathbf{e}_i$ and the residual $\mathbf{r}_i$ is simple but important :

$$\mathbf{r}_i = -\mathbf{A}\mathbf{e}_i \tag{2.7}$$

This can be roughly inferred from the meanings of these two terms. $\mathbf{e}_i$ indicates how far we are from the solution, where $\mathbf{r}_i$ presents how far we are from the correct value of $\mathbf{b}$.

A brief explaination for using the conjugation of the current residual is that the current risidual is orthogonal to the previous search directions. To see this, first recall that in n steps, we will have the result $\mathbf{x}_*$. If we do the substitution of equation 2.4 backwards, we will get :

$$\mathbf{x}_* = \mathbf{x}_0 + \sum_{i=0}^{n-1} \beta_i\,\mathbf{d}_i \tag{2.8}$$

And

$$\mathbf{e}_0 = -\sum_{i=0}^{n-1} \beta_i\,\mathbf{d}_i \quad for\, a\, k \leq n \tag{2.9}$$

Similarly, $\mathbf{e}_k$ is the $\mathbf{e}_0$ plus some corrections (walked through some search directions):

$$\mathbf{e}_k = \mathbf{e}_0 + \sum_{i=0}^{k-1} \beta_i\,\mathbf{d}_i \tag{2.10}$$

Replace the $\mathbf{e}_0$ with equation 2.9, then :

$$\begin{aligned} \mathbf{e}_k &= -\sum_{i=0}^{n-1} \beta_i\,\mathbf{d}_i + \sum_{i=0}^{k-1} \beta_i\,\mathbf{d}_i \\ &= -\sum_{i=k}^{n-1} \beta_i\,\mathbf{d}_i \qquad for\, k = 0, ...n-1 \end{aligned} \tag{2.11}$$

Here we need to apply a little trick that replace -β_i with α_i. Then recall the correlation between error $\mathbf{e}$ and residual $\mathbf{r}$ and the conjugation property, we multiple equation 2.11 with -$\mathbf{d}_j^T\,\mathbf{A}$ from the left on both sides to get :

$$\begin{aligned} -\mathbf{d}_j^T\,\mathbf{A}\,\mathbf{e}_k = &\quad -\sum_{i=k}^{n-1} \alpha_i\,\mathbf{d}_j^T\,\mathbf{A}\,\mathbf{d}_i \\ \mathbf{d}_j\,\mathbf{r}_k = &\quad 0 \qquad\qquad for \quad j < k \end{aligned} \tag{2.12}$$

This shows the beautiful property of the residual that it is orthogonal to the previous search directions, which provides the guarantee of the existence of a new, linearly independent search direction unless the residual is zero. Till then the problem is already solved, that a zero residual shows the function value of the current point is already the target function value. Above all is how the CG works in linear systems, furthermore, it is extended to solve non-linear problems.

Sequential Least SQuares Programming (SLSQP)

Sequential Least SQuares Programming (SLSQP) follows the schema of solving nonlinear programming problem that it iteratively fits the search directionKraft (1988). Which means at k+1-th step, the estimated x^{k+1} will be obtained from x^k :

$$x^{k+1} = x^k + \alpha^k d^k \tag{2.13}$$

where d^k is the k-th search direction and α^k is the step size.

Special in SLSQP is that the search direction is assumed by a qudatric funtion, which is approximated by the second-order Taylor series approximation. Note that minimizing over the difference between the new **x** and the current **x** is equivalent to minimizing over the current **x**, that the current **x** is fixed. The original problem of nonlinear programming problem (NLP) is discribed as followed:

$$(\mathbf{NLP}) : \min_{\mathbf{x}\in\mathbf{R}^n} f(\mathbf{x}) \tag{2.14}$$

subject to:

$$g_j(\mathbf{x}) = 0, \quad j = 1, \dots, m_e, \tag{2.15}$$

$$g_j(\mathbf{x}) \geq 0, \quad j = m_e + 1, \dots, m, \tag{2.16}$$

$$\mathbf{x}_{lowerbound} \leq \mathbf{x} \leq \mathbf{x}_{upperbound} \tag{2.17}$$

Where the value function f and the constrain function g are assumed to be continously differentiable and have no specific structure.

So the problem is substituted by the standard quadratic programming minimizing over the difference of the **x**. Denoting the difference of **x** as **d**, the problem now can be expressed as below :

$$(\mathbf{QP}) : \min_{\mathbf{d}\in\mathbf{R}^n} \frac{1}{2}\mathbf{d}^T\mathbf{B}^k\mathbf{d} + \bigtriangledown f(\mathbf{x}^k)\,\mathbf{d} \tag{2.18}$$

subject to

$$\bigtriangledown g_j(\mathbf{x}^k)\mathbf{d} + g_j(\mathbf{x}^k) = 0, \qquad j = 1, ..., m_e, \tag{2.19}$$

$$\bigtriangledown g_j(\mathbf{x}^k)\mathbf{d} + g_j(\mathbf{x}^k) \geq 0, \qquad j = m_e + 1, ..., m. \tag{2.20}$$

Moreover, the original problem can be approximated by a quadratic form of the LAGRANGE function of nonlinear programming, where the approximation of the constrains g are linear:

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) - \sum_{j=1}^{m} \lambda_j g_j(\mathbf{x}) \tag{2.21}$$

Now putting these together, the choice of the search direction is :

$$B := \bigtriangledown^2_{xx} L(\mathbf{x}, \lambda) \tag{2.22}$$

Which has already been proposed by Robinson (1963) in 1963. As other gradient-based algortihms, the SLSQP suffers from early exit during our experiments as well.

There are further fine techniques on how to control the step size and how to update the **B**-matrix computational efficiently, readers are referred to Kraft (1988).

Step size epsilon

This thesis uses a version of Scipy [1] for optimization algortihms. In the implementation of gradient-based methods, the gradients are approximated by the *forward finite difference formula*:

$$f'(i) = \frac{f(\mathbf{x}[i] + epsilon) - f(\mathbf{x}[i])}{epsilon} \tag{2.23}$$

where :

$$f \; : \; the\ function\ f$$
$$\mathbf{x} \; : \; the\ current\ point$$

Generally speaking, tuning the parameter step size epsilon may lead to different results. In our experiments, 0.01 and 0.1 are applied on CG, and 1e-8 and 0.1 are chosen for SLSQP. For CG we have to use large step sizes otherwise the algorithm will suffer from early exit. At the beginning of our experiments with the naive cost function from chapter 6.1, neither of those step sizes helped to prevent an early exit; this is discussed in more detail in the Chapter *Evaluation* 6.

2.2.2 Black-box Optimization

Unlike gradient-based methods, "black-box" optimization does neither require a clearly defined objective function nor a cost function. Instead of computing gradients of the objective function, the algorithm samples in the parameter space and the search process are optimized by fitness evaluation. Hence, it can be a compensate for gradient-based optimization. Additionally, an important type of black-box optimization is gradient-free population-based *Genetic Algorithm (GA)* a.k.a *Evolution Strategies (ES)*. The idea of Genetic Algorithm (GA) draws on biological evolution, that repeatedly modifies a population of individual solutions. At each evolution step, the genetic algorithm selects individuals with some criterion (for example top two individuals) from the current population to be parents and produces the offsprings for the next generation (mutation). Over several successive generations, the population "evolves" toward an optimal solution. Such method is also called Evolution Strategies (ES).

Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

As the name indicates, CMA-ES adapts a covariance matrix of a multivariate normal search distribution, as a derivation from self-adaptation in ES. A multivariate normal distribution is a generalization of the one-dimensional (univariate) normal distribution to higher dimensions, whereas a normal distribution is a uni-modal (which means one highest number) continuous probability distribution. Any normal distribution, $N(\mathbf{m}, \mathbf{C})$ can be uniquely represented by its mean $\mathbf{m} \in \mathbb{R}^n$ and its symmetric, positive definite covariance matrix $\mathbf{C} \in \mathbb{R}^{n\times n}$.

The covariance matrix has an appealing geometric interpretation that it can be uniquely identified by the (hyper-)ellipsoid $\{\mathbf{x} \in \mathbb{R}^n \,|\, \mathbf{x}^T\mathbf{C}\mathbf{x} = 1\}$, as shown in figure 2.2. As mentioned in the section of CG 2.2.1, the covariance matrix can be uniquely written by an orthogonal matrix $\mathbf{B}$ and a diagonal matrix $\mathbf{D}$, which is formed by the eigenvalues (eigenvalue decomposition) :

$$\mathbf{C} = \mathbf{B}\mathbf{D}^2\mathbf{B}^T \tag{2.24}$$

where:

[1] `https://www.scipy.org/`

$\mathbf{B}$ *is an orthogonal matrix*, $\mathbf{B}^T\mathbf{B} = \mathbf{B}\mathbf{B}^T = \mathbf{I}$. *Columns of* $\mathbf{B}$ *form an orthonormal basis of eigenvectors.*

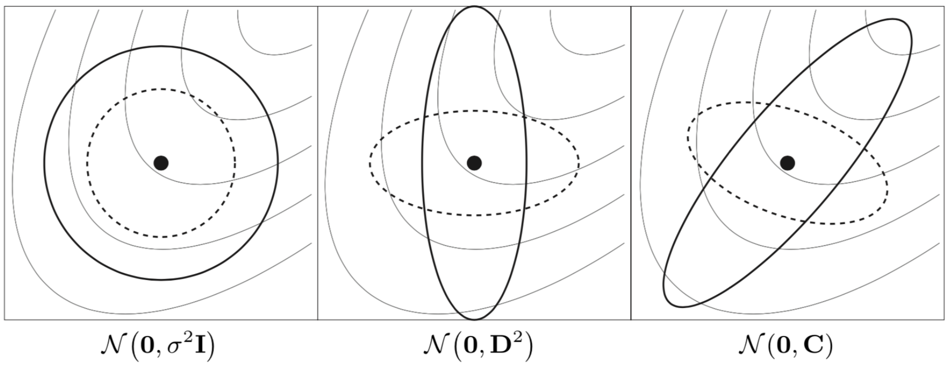
$\mathbf{D}^2 = \mathbf{D}\mathbf{D} = diag(d_{12}, ...d_n{}^2)$ *is a diagonal matrix with eigenvalues of* $\mathbf{C}$ *as diagonal elements.*

I.e., $e_i = d_i{}^2$, *for* $i \in \{1, ...n\}$

$\mathbf{D} = diag(d_1, ..., d_n)$ *is a diagonal matrix with square roots of eigenvalues of* $\mathbf{C}$ *as diagonal elements.*

Naturally, derived from equation 2.24, we can define the square root of $\mathbf{C}$ as :

$$\mathbf{C}^{\frac{1}{2}} = \mathbf{BDB}^T \tag{2.25}$$



$\mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$
$\mathcal{N}(\mathbf{0}, \mathbf{D}^2)$
$\mathcal{N}(\mathbf{0}, \mathbf{C})$

Figure 2.2: Three different normal distributions and their convariance matrix, where the solid line ellipsoids are the geometric interpretation of the covariance matrix, figure credicts to Hansen (2007)

Now, assuming that we have a normal distribution $N(\mathbf{m}, \mathbf{C})$, with the matrix decomposition from equation 2.24, the normal distribution can be written as :

$$\begin{aligned} N(\mathbf{m}, \mathbf{C}) &\sim \mathbf{m} + N(\mathbf{0}, \mathbf{C}) \\ &\sim \mathbf{m} + \mathbf{C}^{\frac{1}{2}} N(\mathbf{0}, \mathbf{I}) \\ &\sim \mathbf{m} + \mathbf{BD}\underbrace{\mathbf{B}^T N(\mathbf{0}, \mathbf{I})}_{\sim N(\mathbf{0}, \mathbf{I})} \\ &\sim \mathbf{m} + \mathbf{B}\underbrace{\mathbf{D} N(\mathbf{0}, \mathbf{I})}_{\sim N(\mathbf{0}, \mathbf{D}^2)} \end{aligned} \tag{2.26}$$

Where "$\sim$" denotes equality in distributions, and $\mathbf{C}^{\frac{1}{2}} = \mathbf{BDB}^T$ Hansen (2016) .

Together with figure 2.2, the equation 2.26 reveals how the the orthogonal matrix $\mathbf{B}$ and the diagonal matrix $\mathbf{D}$ influence the ellipsoid.

$N(\mathbf{0}, \mathbf{I})$ *produces a spherical distribution as illustrated in the first subgraph of figure* 2.2

$\mathbf{D}$ *scales the spherical distribution within the coordinate axes as in the middle subgragh of figure* 2.2.

$\mathbf{B}$ *rotates the ellipsoid, and after the rotation the ellipsoid looks like the one on the rightside of figure*2.2.

It is not hard to see that matrix $\mathbf{D}$ scales the ellipsoid, that multiplying with a diagonal matrix is multiple the diagonal entries to its corresponding columns. I.e., each column i will be scaled by the i-th entry of the diagonal matrix. Morover, recall that $\mathbf{D}$ represents the eigenvalues of $\mathbf{C}$, where eigenvalues indicates the variances of corresponding eigenvectors (also called principle components), that is the elementary normal distribution(the spherical one) is reformed by the variances of the covariance matrix **C**. More will be discussed in section Principle Component Analysis (PCA) 2.4.1.

Analogously, the matrix **B** rotates the formed spheral in figure 2.2 and **B** represents the eigenvectors of **C**. Vectors indicate directions, so, together, **B** is a rotation of the spheral.

Now with this background knowledge of the covariance matrix, it is not difficult to understand the fundamental idea of CMA-ES. In CMA-ES, the update step, where new generation is developed, is sampling a multivariate normal distribution. The basic equation for sampling the search points, for generation number g = 0, 1, 2, . . . , maximum iterations is :

$$x_k^{(g+1)} \sim \mathbf{m}^{(g)} + \sigma^{(g)} N(\mathbf{0}, \mathbf{C}^{(g)}) \qquad for\, k = 1, ..., \lambda \tag{2.27}$$

where :

$\sim$: *denotes the equivalent of the distributions on the both sides.*

$N(\mathbf{0}, \mathbf{C}^{(g)})$: *a multivariate normal distribution with zero mean and convaraince matrix* $\mathbf{C}^{(g)}$,
as mentioned previously, it is equivalent to $N(\mathbf{m}^{(g)}, \mathbf{C}^{(g)})$

$x_k^{(g+1)}$: *k − th individual of generation* $g+1$, $x_k^{(g+1)} \in \mathbb{R}^n$

$\mathbf{m}^{(g)}$: *mean value of the search distribution at generation g*, $\mathbf{m}^{(g)} \in \mathbb{R}^n$.

$\sigma^{(g)}$: *"overall" standard deviation, step − size, at generation g*, $\sigma^{(g)} > 0$.

$\mathbf{C}^{(g)}$: *covariance matrix at generation g. Up to scalar factor* ${\sigma^{(g)}}^2$,
$\mathbf{C}^{(g)}$ *is the covariance matrix of the search direction.*

λ : *population size*, $\lambda \geq 2$.

So now with the update equation 2.27, what remains to be done for a complete iteration is to define how to calculate $\mathbf{m}^{(g+1)}$, $\sigma^{(g+1)}$ and $\mathbf{C}^{(g+1)}$ for the next generation g+1 Hansen (2007).

Choosing the new mean

The new mean $\mathbf{m}^{(g+1)}$ consists of a *weighted average* of μ best individuals of the new generation g+1 :

$$\mathbf{m}^{(g+1)} = \sum_{i=1}^{\mu} w_i x_{i:\lambda}^{(g+1)} \tag{2.28}$$

where :

μ : $\mu \leq \lambda$ *(the population size), is the parent population size.*

w_i : *with the sum* $\sum_{i=1}^{\mu} w_i = 1,\ w_i > 0,\ for\, i \in \{1, , , , \mu\}$
Moreover, $w_1 \geq w_2 \geq ..., w_\mu \geq 0$. *Therefore with the sum in equation* 2.28
calculates the average of the individuals.

$x_{i:\lambda}^{(g+1)}$: *i − th best individuals of genertation* $g+1$, *evaluated by the objective function*
that needs to be minimized.

When the new mean is chosen using this method of selection and recombination, the estimated population mean remains in the area of the μ-best individuals of the next generation.

Estimating the covariance matrix

Similar to calculating the new mean, the covariance matrix is updated in the manner of *weighted selection* as well. First the covariance of the best-μ individuals is calculated :

$$\mathbf{C}_{\mu}^{(g+1)} = \sum_{i=1}^{\mu} w_i \,(x_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)})\,(x_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)})^T \tag{2.29}$$

Sampling within the best-μ individuals tends to reproduce "selected", i.e. successful generation. To ensure that $\mathbf{m}_{\mu}^{(g+1)}$ is a reliable estimator, the μ must be sufficiently large. For a fast search, the population size λ has to be small, hence, the μ must be small as well. Thus, the idea of **rank-μ-update** is deployed :

$$\mathbf{C}^{(g+1)} = (1 - c_{cov})\mathbf{C}^{(g)} + c_{cov}\,\frac{1}{\sigma^{(g)^2}}\mathbf{C}_{\mu}^{(g+1)} \tag{2.30}$$

where:

c_{cov} : $c_{cov} \leq 1,$ *learning rate for updating the covariance matrix*

$\sigma^{(g)}$: *the step size.*

With the factor $\frac{1}{\sigma^{(g)^2}}$ in equation 2.30, the recent generation is assigned to a higher weight, see Hansen (2007). One more concept needs to be introduced, the *evolution path*. The evolution path reveals the mean movement irrespective of the step-size in the search space. The evolution path of generation g+1 is defined as :

$$\mathbf{P}_c^{(g+1)} = (1 - c_c)\mathbf{P}_c^{(g)} + k\,\frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \tag{2.31}$$

where :

$\mathbf{P}_c^{(g)}$: *evolution path at generation g.*

c_c : $c_c \leq 1,$ *similar to the learning rate in equation* 2.29.

k : *a constance constructed by c_c and μ for normalization.*

The final update of the covariance matrix of CMA-ES combines the *rank-μ-update*, see equation 2.30, and the *evolution path*, see equation 2.31, which has a complicated form that is not necessary to be explained here, for detailed information refer to Hansen (2007).

Step size control

The step-size σ is modified through a method called *cumulative path length control*, *cumulative step size control*, or *cumulative step size adaptation*, see Alexandre Chotard and Hansen (2012). The step size is adapted according to the measurement of the length of a so-called cumulative path. The cumulative path is a weighted sum of the previous steps realized by the algorithm, where the weights of each step decreases with time. Different strategies are exploited regarding the length of the cumulative path.

1. If the evolution path is long, it means that the steps are similar, hence correlated. That means the same distance can be summed up to longer steps in the same direction, consequently there are fewer steps. The step-size should be increased.

2. If the evolution path is short, it means that the samples are moving back and forth. Roughly speaking, the steps are anti-correlated that they cancel each other out. In this case, the step size should be decreased.

3. In the ideal case, the steps are approximately perpendicular and therefore uncorrelated.

To approximate the ideal case, we want to construct the distribution mean of consecutive steps so that they are approximately $\mathbf{C}^{(g)^{-1}}$-conjugate. Recall that $\mathbf{C}^{(g)^{-1}}$-conjugate is also called $\mathbf{C}^{(g)^{-1}}$-orthogonal, which leads to the equation that :

$$(\mathbf{m}^{(g)} - \mathbf{m}^{(g-1)})^T\,\mathbf{C}^{(g)^{-1}}\,(\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}) \approx 0 \tag{2.32}$$

The update of the step-size is derived from equation 2.32.

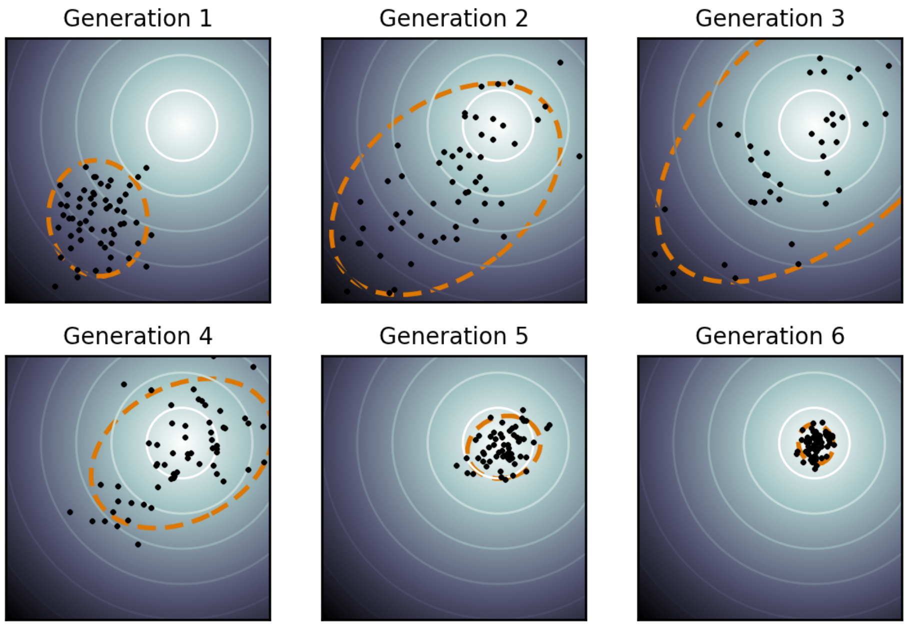


Figure 2.3: Illustration of an actual optimization run with CMA-ES on a simple two-dimensional problem. The spherical optimization landscape is depicted with solid lines of equal values. On this simple problem, the population (the dots) concentrates over the global optimum within a few generations. Figure credicts to Wikipedia (2018)

Illustration 2.3 shows the path of the population to the global optimum, where the orange dotted line is the contour of the covariance matrix of that generation.

Differential Evolution

The Differential Evolution (DE) is a member of the population-based Evolution Strategies (ES), which belongs to the category of random search algorithms. As the name suggests, randomness or probability is involved in the definition of this kind of algorithm. In literature, it is also be called Monte Carlo method or stochastic algorithm, see Zabinsky (2009).

The idea of DE is simple, yet it works robustly and efficiently, see Storn and Price (1997). Here the vectors that represent the parameters to be optimized are denoted as *parameter vector* or *population vector*. For every generation, it mutates every population vector from the current set. The term *mutation* it means that the algorithm generates a new vector by adding some weighted differences to some chosen vectors as described in equation 2.33.

$$v_{i,\mathbf{G}+1} = x_{r_1,\mathbf{G}} + F \times (x_{r_2,\mathbf{G}} - x_{r_3,\mathbf{G}}) \qquad (2.33)$$

Where :

$r_1,\ r_2,\ r_3\ \in\ \{1,\ 2,\ \dots\ ,\ population\ size\},\ arbitrarily\ chosen,\ mutually\ different$

$\mathbf{G}+1\ denotes\ the\ next\ generation\ of\ \mathbf{G}$

$F\ :\ the\ scale\ factor,\ > 0$

Then, the parameter vectors might be mixed up, this process is also referred to as *crossover* in the ES community for the purpose of increasing the diversity of the parameter set. This will happen if the generated random number is smaller than the crossover threshold (CR), which is determined by the user. Figure 2.2.2 illustrates the crossover process and the selected vector to be mixed is denoted as *target vector*. The vector after mixing is called *trial vector*.

The newly generated vector, a.k.a. the trial vector, will replace the target vector, which is the original vector before crossed over, that is being added to the new generation when it has a better cost function value. This process is referred to as a *selection*.

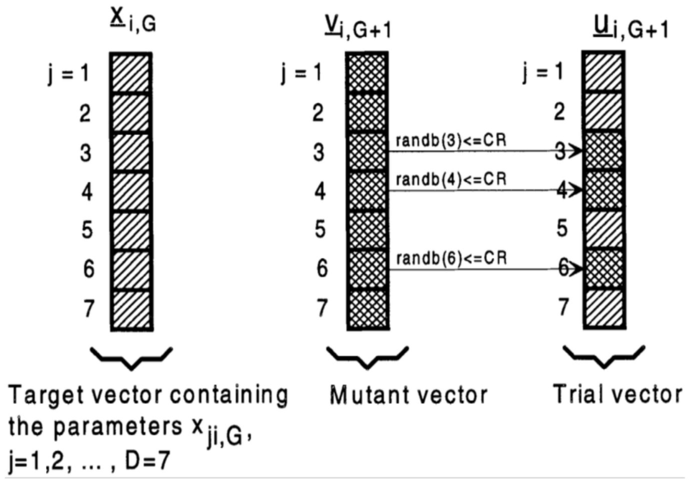


Figure 2.4: Illustration of the crossover process for 7 parameters, figure credits to Storn and Price (1997)

Note that in this selection and crossover process, DE does not involve any probability distribution which makes this scheme completely self-organizing (self-adaption), and hence gradient-free. The crucial idea behind DE is the perturbation process (mutation and crossover) that ensures the exploration of the search space.

2.3 Separating Axis Theorem

The **separating axis theorem** (SAT) says that:
Two convex objects do not overlap if there exists a line (called axis) onto which the two objects' projections do not overlap.
This is used in our object collision detection to avoid overlappings in the simulation. It is a diversion of the **hyperplane separation theorem** in convex geometry.

The *hyperplane separation theorem* states that:
Suppose C and D are non-empty disjoint convex sets, i.e., $C \cap D = \emptyset$, $C \neq \emptyset$, $D \neq \emptyset$. Then there exist a $\neq 0$ and b such that $a^T x \leqslant b$ for all $x \in C$ and $a^T x \geqslant b$ for all $x \in D$.

Figure 2.3 suggests that the affine function $a^T x$ - b is less or equal than 0 on C and greater or equal than 0 on D.

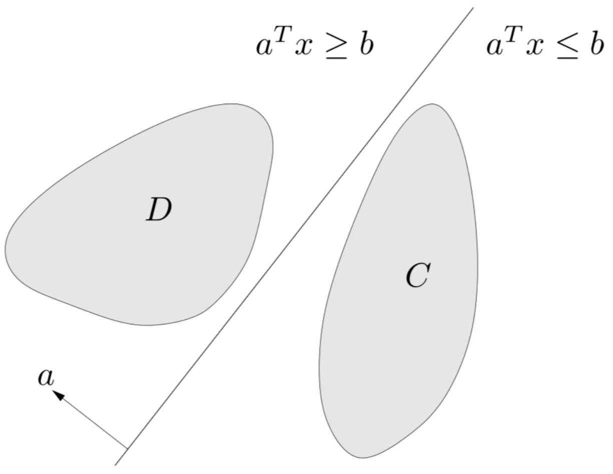


Figure 2.5: Illustration of two disjoint convex sets C and D, where the hyperplane {x | a^Tx = b} seperates them. Illustration credits to Boyd and Vandenberghe (2004)

2.4 Data analysis

In this section two basic data analysis techniques will be introduced: Principle Component Analysis (PCA) and Linear Discriminant Analysis (LDA). Both of them are aimed for dimension reduction by finding a meaningful re-expression of the data.

2.4.1 Principle Component Analysis (PCA)

PCA is a mainstay of data analysis, revealing the underlying data with a lower dimension expression through a simple, non-parametric manner. The PCA can be considered as a rotation on axes, from the original coordinate system to orthogonal axes, which are called *principle axes*. Rotating the axes is analogous to seeing the data from a different angles, which is realised through the changing of the basis in *linear algebra*, which implies the assumption of *linearity* that PCA makes. The new axes coincide with maximally varied directions to the original observations, see Campbell and Atchley (1982). Imaging that feature points are distributed in the ellipse from figure 2.4.1, the center of the ellipse ($\bar{x}_1$, $\bar{x}_2$) is the average of the points within the ellipse. The cosine of the angle θ between Y_1 and X_1 gives the first component $\mathbf{u}_{11}$ of the eigenvector corresponding to Y_1. This will be explained in this section below.

In practice, PCA iteratively finds the eigenvalues e_i and eigenvectors $\mathbf{u}_i$ of the covariance matrix or the correlation matrix of the original data, because the covariance matrix or the correlation matrix are symmetric and the eigenvalues are orthogonal(which is breifly mentioned in 2.2.1). A *covariance* matrix is a matrix whose element in the position (i,j) is the covariance between the i-th and j-th elements of a random vector. According to the definition, the positions (i,j) and (j,i) should have the same value, hence the symmetricity. The correlation matrix is the standardrized version of its corresponding covariance matrix, that is divided by the standard deviation. To see that the eigenvalues are orthogonal, we first need to recall the knowledge of the *scalar product* or *dot product*. The dot product of two vectors **a** and **b** is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i \tag{2.34}$$

A dot product in cartean coordinates (as is this case here) is a scalar product $<\cdot, \cdot>$. For any matrix A

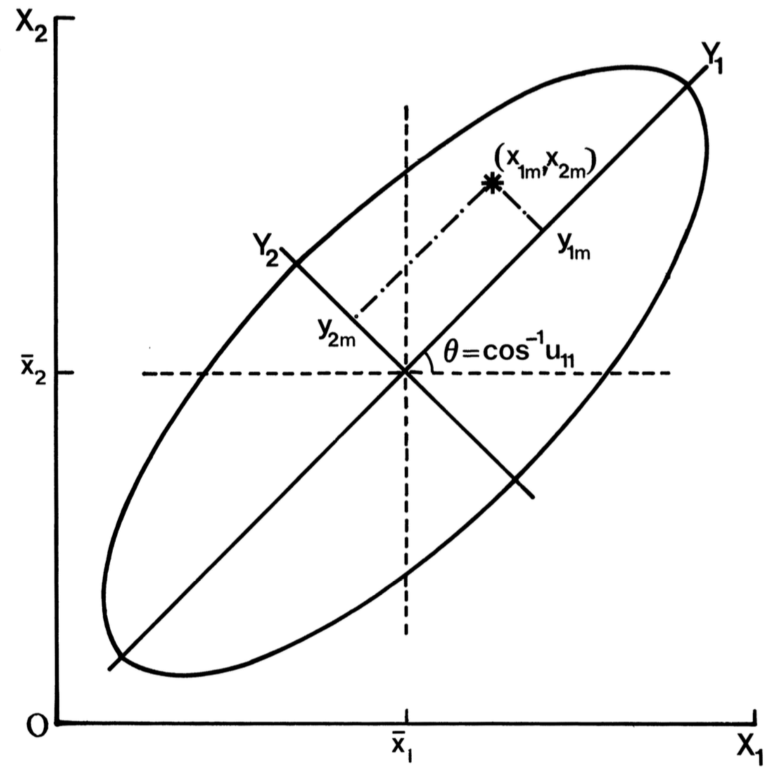


Figure 2.6: The long axis Y_1 of the ellipse is the 1. principle axe. Then the perpendicular one, Y_2, is the 2. principle axe. Illustration credits to Campbell and Atchley (1982).

who has its transpose, the following applies :

$$< A\mathbf{x}, \mathbf{y} > = < \mathbf{x}, A^T\mathbf{y} > \tag{2.35}$$

And now we have a symmetric matrix A and **x**, **y** are the eigenvectors of A coresponding to distinct eigenvalues λ and μ (that is $\lambda \neq \mu$). We have :

$$\lambda < \mathbf{x}, \mathbf{y} > = < \lambda\mathbf{x}, \mathbf{y} > = < A\mathbf{x}, \mathbf{y} > = < \mathbf{x}, A^T\mathbf{y} > = < \mathbf{x}, A\mathbf{y} > = < \mathbf{x}, \mu\mathbf{y} > = \mu < \mathbf{x}, \mathbf{y} > \tag{2.36}$$

Then, if we put the head and tail of equation 2.36 together, we will get :

$$(\lambda - \mu) < \mathbf{x}, \mathbf{y} > = 0 \tag{2.37}$$

Since λ , μ are distinct, that means $\lambda - \mu \neq 0$; thus, $<\mathbf{x},\mathbf{y}> = 0$, i.e., **x** is orthogonal to **y**. Furthermore, eigenvectors form an orthogonal basis for the matrix A space. A further property of a symmetric matrix is its diagonalizability. Let's put the eigenvalues $\mathbf{e}_i$ of A on the diagonal to form a diagonal matrix E, i.e., E = diag($\mathbf{e}_1$, ..., $\mathbf{e}_n$) and put the eigenvectors together to get a matrix U, where U = ($\mathbf{u}_1$, ..., $\mathbf{u}_n$). The covariance matrix of the original data is denoted as V. Then the eigen-equation becomes :

$$\begin{aligned} V &= UEU^T \\ &= \sum_{i=1}^{n} \mathbf{e}_i\mathbf{u}_i\mathbf{u}_i{}^T \end{aligned} \tag{2.38}$$

The equation 2.38 reveals an important property, i.e. that the sum of the variances of the original variables is equal to the sum of the eigenvalues. This also shows the correctness of the implementation of PCA.

The PCA ranks the eigenvalues in a descending order, that is, $e_1 > e_2 > ... > e_n$. Recall that an eigenvector corresponds to a direction. The corresponding eigenvalue of it reveals the variance of the data along that eigenvector (or principal component), see Hamilton (2014). Now, we have a better understanding of figure 2.4.1, i.e. that the cosine of the angle θ is a direction and it corresponds to the first eigenvector $\mathbf{u}_1$. And the perpendicular axe to Y_1, denoted as Y_2 in the figure, is the second principal component, that the variance of Y_2 is smaller than that of Y_1.

2.4.2 Linear Discriminant Analysis (LDA)

The crucial idea behind LDA is to find a lower dimensional space, where the *between-class variance* is maximized and *within-class variance* is minimized. The *between-class variance* (S_B) is the distance between the means of different classes, a.k.a. the separability between different classes. The *within-class variance* (S_W) is the distance between the mean and the samples of each class.

After calculating the between-class variance (S_B) and the within-class variance (S_W), the transformation matrix (W) of the LDA technique can be calculated according to Fisher's criterion, as described in equation 2.39, which is basically to find a matrix W that maximize the difference between S_B and S_W after transformation.

$$\underset{W}{\operatorname{argmax}} \frac{W^T S_B W}{W^T S_W W} \tag{2.39}$$

The above equation 2.39 is equivalent to the one below (equation 2.40):

$$S_W W = \lambda S_B W \tag{2.40}$$

where :
λ : the eigenvalues of the transformation matrix W.

If the matrix S_W has its own inverse S_W^{-1}, i.e. S_W is non-singular, then the matrix W can be obtained by the following equation 2.41 :

$$W = S_W^{-1} S_B \tag{2.41}$$

Recall that eigenvalues are scalar values and eigenvectors are directions(non-zero vectors), which provide the information of the LDA space. The eigenvalues of W will be calculated, as well as their corresponding eigenvectors. Similar to PCA, each eigenvector represents an axis in LDA space. The eigenvalues reflect the discriminant between different classes, i.e. a high eigenvalue increases the between-class variance, and decreases the within-class variance of each class. Thus, the k eigenvectors with their k highest eigenvalues form a lower dimensional space V_k, where the rest are neglected.

In the visualization 2.7 we can see that the first eigenvector v_1 shows a better separability of the classes. The distances between-classes are greater than those projected on the second eigenvector v_2. Moreover the within-class variances of the projections on v_1 is smaller than those on v_2. So in this case, v_1 is selected to construct a lower dimension representation.

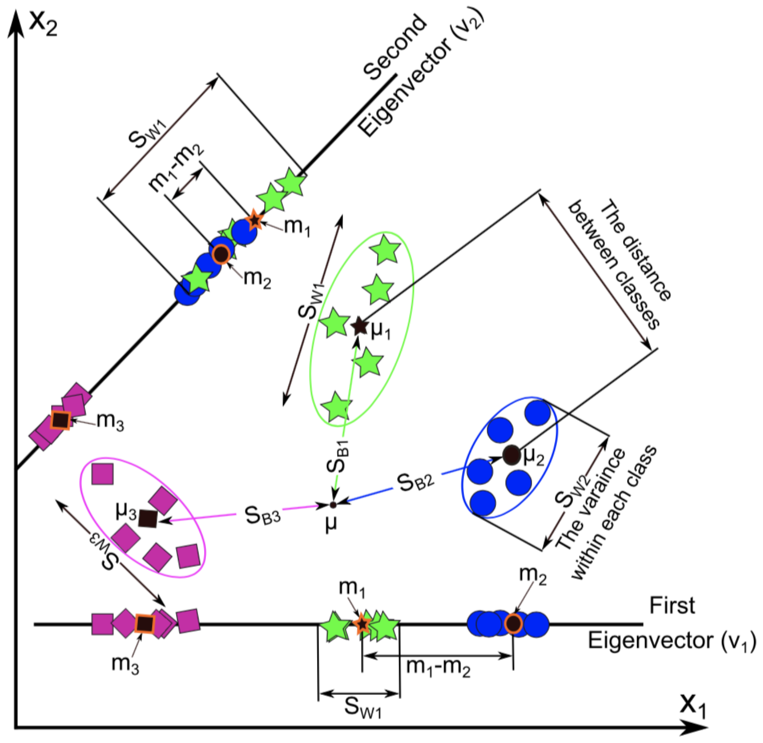


Figure 2.7: A visualization of LDA for three classes in two lower dimensions. The illustration credits to Alaa Tharwat and Hassanien (2017).

3 Related Work

3.1 Open Dynamics Engine (ODE), dynamic model

Our simulation is built with ODE, an open source library for simulating rigid body dynamics [1]. The implementation is written in C++, extended from the script of the course, see Atkeson (2018). In ODE, a rigid body is effected by the physical environment, that is it will fall to the ground due to the gravity. ODE provides a possibility to create an immovable body (such as the obstacles in our case) that yet interacts with other bodies, that it has a body type referred to *geometry* (body), which is designed for collision, see Smith (2006) .

In our real world experiment, a small initial force is added to the ball so that the ball can start to run. Otherwise, we need to remove the supporting magnet under the ball so that its fall is initiated by gravity, which is more complicated and will affect the result of color detection. Thereby, an initial velocity in the horizontal direction is applied on the ball.

Our intention is that with the output of the optimizer, which provides for a successful run in the simulation, it can be applied to the real world straight by the robot. To ensure that the result can be transformed seamlessly into the real world, the measurements of obstacles and the gird are proportionately correspond to the real world. Some paramters for the physical properties, for example bounciness factor or friction factor, are tuned as well. We tuned the parameters so that the plot of the simulated trajectory is approximating the real world one, as illustrated in figure 4.3.

3.2 Object Detection

To detect the given goal position and obstacles, the robot has to resolve the perception problem. In our approach, this is solved by color detection. Our implementation is based on OpenCV, an open source computer vision library[2]. OpenCV is developed in C++ and has a python wrapper. Our code is written in python, inspired by online tutorials[3].

To distinguish the function of the objects, each object has its own color in our real world experiments. The ball is red, movable obstacles are blue, fixed obstacles (for instance, the goal object) are lilac. Each color has its own range in the HSV map. The **HSV (hue, saturation, value)** is an alternative color representation of the RBG (red, blue, green) color model. A common technique of color detection is that the image is converted into a HSV representation first, then the color mask is applied on the image, which is principally thresholding on the HSV values. Figure 3.1 depicts the layout used in our real world experiment, the two magnets on the top of the grid are for defining the grid area in color detection. The two figures from 3.2 reveal that color detection is sensitive to lighting. Though the value range for red in HSV remains the same all the time, and with different lighting, the color detector allows other objects to pass the mask; for instance, are some parts of the hand recognized as red in the graph 3.2(b). To overcome the disturbance from the environment, the recognition area is restricted, that is the grid between the two magnets in figure 3.1. The contour area of detected objects also serve as a constraint. If the pixel number in the contour area is too small or too large, the detector will regard the contour as noise.

[1]`http://www.ode.org/`
[2]`http://opencv.org/`
[3]`https://www.pyimagesearch.com/2015/09/14/ball-tracking-with-opencv/`

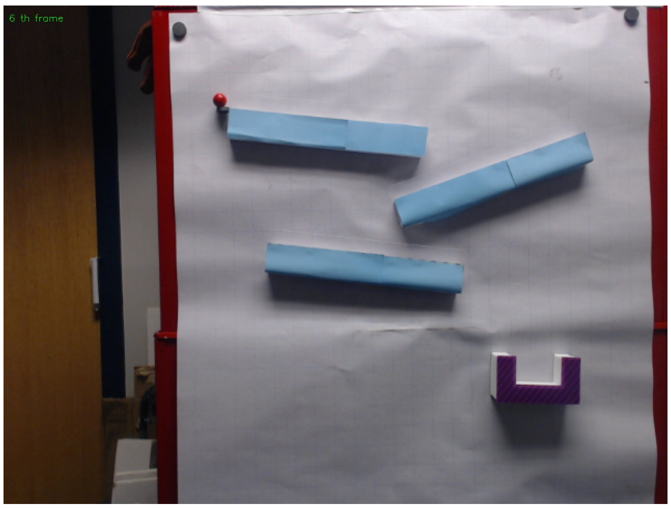
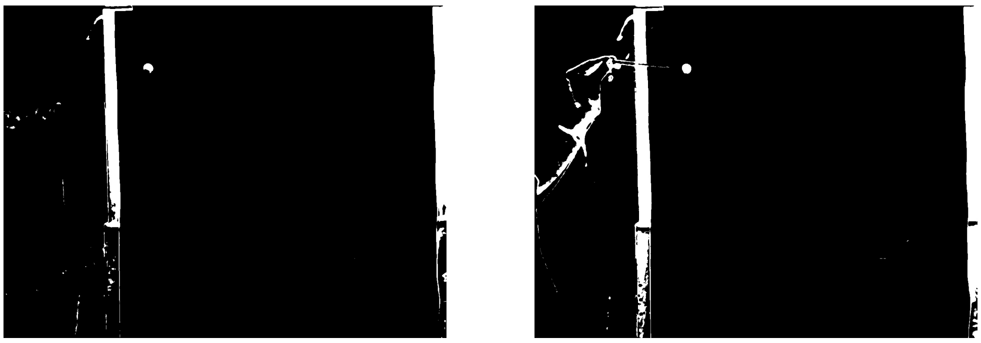


Figure 3.1: The layout for color detection



(a) The layout applied with a red mask before a hand at the ball

(b) The layout applied with a red mask when a hand is at the ball

Figure 3.2: The layout applied with a red mask

3.3 Optimization Algorithms

Most of the algorithms deployed in this thesis have an implementation in Scipy, a scientific computing tool for python[4]. An exception is the Random Search (RS), which we implemented by ourselves. The implementation of RS is straight forward, as shown in algorithm 3.3. It does nothing else than random sampling and comparing the cost function values.

```
function RANDOM METHOD
    best = 1e20
    best_x = None

    for i in range(20000) do:
        $x_0$ = GET_RANDOM_X0

        ▷ the function f is the function to be optimized
        cost = f(x0)

        if cost < best then
            best = cost
            best_x = x0
        end if
    end for

    return best, best_x
end function

function GET_RANDOM_X0
    ▷ bounds is a two dimensional array, with each elements lower bound and upper bound as tuple
    randn = numpy.random.random_sample(bounds.size)
    return numpy.array([x[0] + a $\times$ (x[1]-x[0]) for x, a in zip(bounds, randn)])
end function
```

3.4 Learning from scratch

This section is rather related to future work, that one of the long-term goal of AI research is to bypass human knowledge, which is sometimes too expensive, too unreliable or simply unavailable. I.e., AI researchers have the ambition to create algorithms that have superhuman performance without human input. As mentioned before, in chapter 2, RL uses a lookup table called Q-network to decide which action should be picked according to the current state. There are multiple ways to gain such Q-networks, one possiblity is to train a supervised network for move probabilities. For example, the early version of AlphaGo, the one which beated the human world champion Lee Sedol, used two deep neural networks, a policy network and a value network, see David Silver and Hassabis (2017). The policy network was first trained to approximate human experts moves, and then refined by policy-gradient reinforcement learning. The value network is designed to predict the winner of games, where games played by the policy network against itself. In 2007, the new version of AlphaGo, the AlphaGo Zero is released, which is trained completely by self-play reinforcement learning, without any supervision or use of human data. It surpassed the performance of previous versions of AlphaGo. The training of AlphaGo Zero started with random play, and the input is merely the black and white stones from the board. We can say that the program AlphaGo Zero figured out how to play the game of Go by itself.

[4]https://www.scipy.org/about.html

Nonetheless, Martin Riedmiller and Springenberg (2018) trained the robot arms to pick up objects with reinforcement learning without human input. We can see that AI researchers are investigating to find a way to achieve autonomous systems.

4 The Approach

In the field of robotics, simulation allows multiple runs at low cost and with the flexibility to change parameters. It is common to run experiments in simulations and then transfer the results that can be directly applied on the robots. As a real world reference, a video was recorded with the magnet marble run. Afterwards, we reproduced it with the Open Dynamics Engine (ODE) [1] simulator. In our case, there are three movable obstacles in cuboid shape, and a goal object in a "cup" shape that consists of three fixed obstacles. To enhance the variability, another fixed cuboid obstacle may be added in the simulation. Therefore the problem is reduced to compute the positions and angles of the three movable obstacles, so that with such layout, the marble ball, whose initial position is in the top-left corner of the wall, will reach the goal object and stay in the goal object, we defined such a layout as a successful layout. The goal object is located at the bottom of the wall, the x-axis position of the goal object may vary within the right half of the wall. To solve this problem the optimization algorithm is applied to compute the positions and rotation angles of the movable obstacles.

4.1 Preparation

As described in detail in the previous chapter 3.2, the ball, in a real world video (available online in normal speed [2] and in slow-motion as well [3]) is tracked with OpenCV [4]. First, the objects are detected based on the colors and then, with the background subtraction function from OpenCV, the ball trajectory is subtracted. The video is recorded with a Logitech webcam with a frame-rate of 30 frames per second (fps). In order to avoid losing frames, this analyzing process is run offline. Figure 4.1 shows the trajectory of the magnet ball run in the real world.

We built a dynamic model with *Open Dynamics Engine (ODE)*, which simulates the physical environment of the game. Figure 4.2 shows the layout as a simulation.

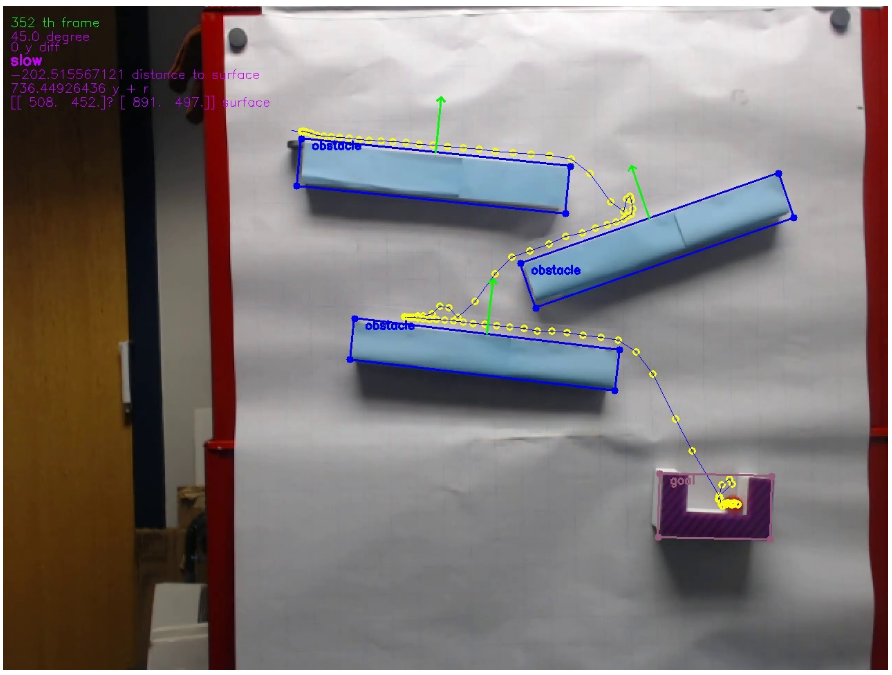
To minimize the gap between simulation and reality, first the dimensions of the obstacles were measured and transformed from the pixel space of the video into the simulation environment space. Then we manually adjusted the parameters of the simulation environment from ODE which are described in Chapter 3, so that the simulated trajectory approximates the real trajectory, which is shown in figure 4.3.

4.2 Implementation

Our implementation to compute a successful layout consists of basically three parts:

- The optimizer
- Clean up the solution
- Select the best from a set of successful layouts

[1] `http://www.ode.org/`
[2] `https://youtu.be/_zpa1WufInU`
[3] `https://youtu.be/wfrcggF0Yus`
[4] `https://opencv.org/`



```
352 th frame
45.0 degree
0 y diff
slow
-202.515567121 distance to surface
736.44926436 y + r
[[ 508.  452.]? [ 891.  497.]] surface
```

Figure 4.1: Ball trajectory

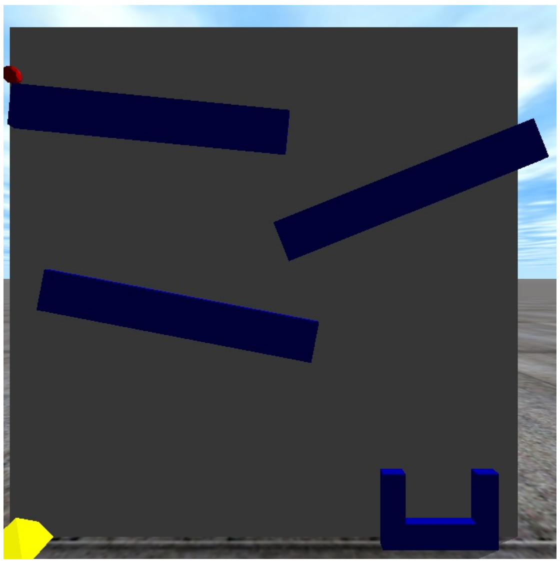


Figure 4.2: The reproduction in our simulator

```
procedure OPTIMIZING(x)
Input: x is a 9-dimensional float array, with space information about three movable obstacles
global: hit, best_obst, best_score

    ▷ f is the function to be optimized
    function F(x)
        obstacles ← get_params(x)                        ▷ convert position information into obstacles

        ▷ Result is a list of different features counted by the dynamic model,
        ▷ such as velocities, positions, bounce count etc...
        result ← run_prog_process(obstacles)               ▷ pass the obstacles to the dynamic model

        cost ← COST FUNCTION(result, obstacles)

        return cost
    end function

    function COST FUNCTION(result, obstacles)
      j Input:     result  :   a list of velocities when the ball enters the goal
                               plane, positions, bounce count of the ball during
                               the current run
                obstacles  :   current obstacles with space information of the
                               current run
        overlap_count ← get_overlap_count(obstacles)

        penalty ← 0 if run succeeded else 1

        cost ← cost equation 4.1

        if run succeeded $\wedge$ overlap_count == 0 then
            score ← score equation 4.2
            if score $\leq$ best_score then
                best_score ← score
                best_obst = obstacles
            end if

            if len(hit) $\geq$ 10 then
                write_result_to_file(best_obst)
                sys.exit(0)
            end if
        end if
    end function

    function RUN
        for v in range(10) do:
            if $\neg$ overtime $\wedge$ hit oversize then
                DIFFERENTIAL EVOLUTION
            end if
        end for
    end function

    function DIFFERENTIAL EVOLUTION
        result = differential_evolution(F, bounds, popsize = 5, maxiter= 5, disp= False, atol =1)
        return result.fun, result.x
    end function

end procedure
```

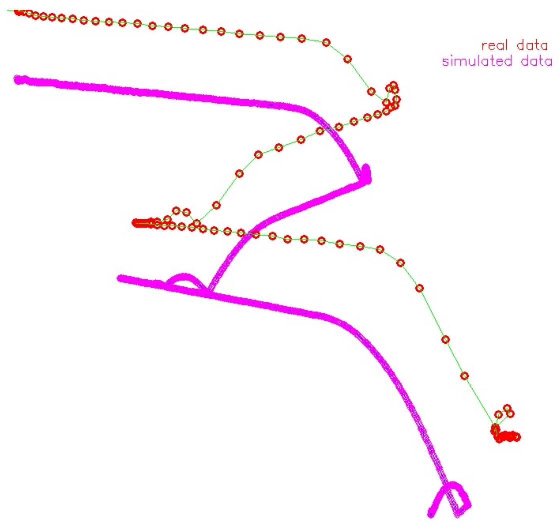


Figure 4.3: The plots of real and simulated trajectories

To find a robust and successful solution of the placement of the movable obstacles in a reasonable run time, experiments are carried out on different kinds of optimization algorithms. Both in gradient-based and non-gradient-based manner. The **Covariance Matrix Adaptation Evolution Strategy** with fixed initialization stood out in our case; this is discussed in more detail in the chapter *Evaluation* 6.

We used the center of mass position in both x and y-axes and the rotation angle of each movable obstacle as input for the optimizer. Since our optimization space is a nine-dimensional continuous space (we have three movable obstacles and each has three parameters), the numerical result from the optimizer may lead to intersections of obstacles in reality. Figure 4.4 illustrates one possible solution returned by the optimizer without clean-up, where the obstacles overlap each other. It is a successful but not valid output, because in the reality the obstacles should not overlap. Therefore, there is an additional clean up step before the parameters are being evaluated by the cost function from the optimizer.

4.2.1 The clean-up step

In the clean-up step, we first identified which movable obstacles were idle, to find out with which obstacles collide with the ball and which not. Only the obstacles colliding with the ball during the run make an impact on the ball's trajectory. Therefore, we moved the idle obstacles near the goal object to increase the robustness of the output layout. Since a gap between simulation and reality always exists, the ball may bounce out of the goal object in the reality even if it would not do so in the simulation. Adding idle obstacles near the goal object can prevent the ball from bouncing out of the goal object. Since our setup is that the ball drops from the upper left corner of the "wall" and the goal object is located at the bottom right corner, the first idle obstacle will be moved to the right side of the goal object and placed in an inclined position facing the goal object. If there are more idle obstacles available, the second idle obstacle will be moved to the left side of the goal object and be placed in an inclined position facing the goal object. If there are still more available idle obstacles, they will be moved out of the "wall", which is simplily setting the posistion to some large corrdinates that are not visible from the current point of view.

We can see from figure 4.5 that the layout as shown in 4.5(a) can help the ball slide back towards the goal object if the ball bounces out of the goal. When using the layout as shown in 4.5(b) is used, the ball

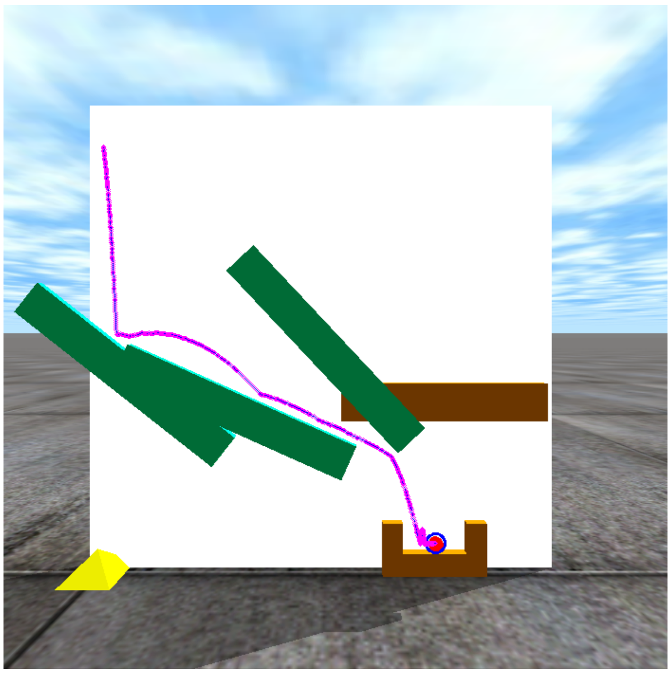
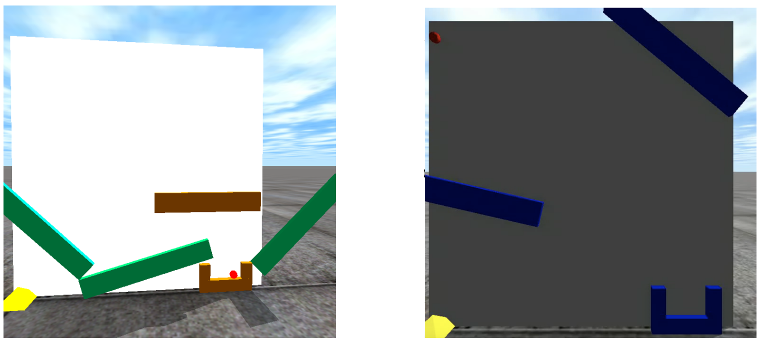


Figure 4.4: A possible successful layout with clean-up

will bounce out of the goal and the run will fail.



(a) A robust layout example (b) A not robust layout example

Figure 4.5: Examples of robust and non-robust layouts

After moving the idle obstacles, on the setting is inspected for intersections of obstacles. An algorithm for checking whether two convex polygons are overlapping is used, that was inspired by the *Separating Axis Theorem*. The pseudo-code is shown in Algorithm 4.2.1 - implemented based on the post of Cozic (2006). We compare the movable obstacles pairwise and then each movable obstacle with all fixed obstacles. We separate the movable obstacles into two sets, one set contains those obstacles colliding with the ball and the other one is the idle set. If there is an intersection between two non-idle obstacles, we move one of the obstacle along its longitudinal direction; by doing so, the ball trajectory stays locally remained. That is, if the ball bounces to the right side of its falling direction, after this move of obstacles, the ball will still bounce to the right side of its falling direction. If a non-idle obstacle overlaps with an idle obstacle, the idle obstacle will be moved along its longitudinal direction. Similarily, if a movable obstacle intersects with a fixed obstacle, the movable obstacle will be moved along its longitudinal direction. Our obstacles are cuboid in reality and thus can be seen as two dimensional rectangles in the view of camera. The longitudinal direction is the direction along its longer edge of the rectangle.

Note it is rather naive to clean up in this way and it does not always return a successful layout or even a cleaned up layout. Recall that the obstacles are compared pairwise, where one possible scenario is that obstacle A first intersects with obstacle B and both obstacles have an influence on the ball trajectory. After fixing this pair of obstacles, one of the obstacles, say obstacle A, is overlapping with another obstacle C and so on. After fixing all of them, the ball trajectory will have changed and an unsuccessful layout is computed. However, it is not trivial to move obstacles and yet maintain the ball trajectory without seeing the obstacles as in our case. In the early phase of our optimization process, the clean-up step is placed after a valid solution is found. Due to the imperfectness, the clean-up may return faulty results. A quick fix would be to pass the imperfectly cleaned-up layout to the evaluation step and to constrain from there. Thus, the evaluation of cost function is executed after the clean-up step.

4.2.2 Cost Function

Since the imperfect clean up step takes place during the optimization run, the range of possible successful solutions downsized. This may result in a longer time for the optimizer to find a successful solution but once it is found, the solution is successful and non-overlapping as desired. Our results show that by deploying the right optimization algorithms, the optimizer is still able to find desirable solutions within a reasonable time - with the use of some tricks. We keep a list of desirable solutions and a best score record with the best solution, which is immediately written into an output file, where the best solution is updated and the former best solution is overwritten. Once the list has more than the desired amount of elements, the program will exit.

Regarding the robustness, a small number of bouncing up and down in the goal object is preferred. In our dynamic model, we count how often the ball bounces into the goal object and out, as well as the velocity when the ball enters the goal entry plane for the first time. Jumping into the goal entry plane with a smaller velocity means that the energy that results from the ball bounce is smaller, hence it is more likely that the ball will not bounce out.

Our dynamic model records the velocities of x-and y-axes when the ball is slightly over the goal entry plane for the first time and when the ball is slightly under the goal entry plane for the first time. As in the direction of the y-axis the ball will be accelerated by gravity anyway, the velocities in the direction of x-axis should be minimized (slightly over and under the plane). Our naive version of the optimizer only minimized the velocities when the ball entered the goal entry plane. Then however, in many of these solutions, the ball hits the edge of the goal object first and then slides into the entry plane. That is to say the successfulness depends on the edge of the goal object, which may lead to an unsuccessful run in reality. Note that in reality the obstacles are magnetic and stick on a wall, and a hit on the edge may change the goal object's layout. This is not the case in the simulation, where a fixed obstacle never changes its position. A clean and safe solution would be to let the ball fall into the entry plane with a low velocity in the middle of the goal object. Hence the dynamic model records the x position of the ball when it is entering the entry plane and the x-position after the ball entered the entry plane and is slightly

Algorithm 1 2D Polygon Collision Detection

```
function OVERLAP($ver_1, ver_2$)
    Input :   $ver_1$ : $list\ of\ vertices\ of\ first\ rectangle,\ in\ tuple\ form$
              $ver_2$ : $list\ of\ vertices\ of\ second\ rectangle,\ in\ tuple\ form$
    Output :  $True$   : if two rectangles overlap each other
              $False$  : if they do not  intersect
    rects = [$ver_1$, $ver_2$]                                    ▷ rects: a list of 2 rectangles

    for r in rects do

        for i in range( length of r ) do
            $min_1$, $min_2$, $max_1$, $max_2$ = None

            $ind_2$ = (i + 1) % len(r)
            $p_1$ = r[i]
            $p_2$ = r[$ind_2$]

            ▷ normal of this edge ($p_1$,$p_2$)
            normal = ($p_2$.y - $p_1$.y , $p_1$.x - $p_2$.x)

            for  v in $ver_1$ do

                ▷ the projected point of v along vector $\overrightarrow{p_1 p_2}$
                projected = normal.x $\times$ v.x + normal.y $\times$ v.y

                if $\neg$ $min_1$ || projected < $min_1$ then
                    $min_1 \leftarrow projected$
                end if
                if $\neg$ $max_1$ || $projected$> $max_1$ then
                    $max_1 \leftarrow projected$
                end if

            end for

            for  v in $ver_2$ do

                ▷ the projected point of v along vector $\overrightarrow{p_1 p_2}$
                projected = normal.x $\times$ v.x + normal.y $\times$ v.y

                if $\neg$ $min_2$ || projected < $min_2$ then
                    $min_2 \leftarrow projected$
                end if
                if $\neg$ $max_2$ || $projected$ > $max_2$ then
                    $max_2 \leftarrow projected$
                end if

                if $max_1 < min_2$ || $max_2 < min_1$ then
                    return False
                end if
            end for

        end for
    end for

    return True

end function
```

under the plane. This is also encoded in the cost function:

$$\begin{aligned} cost = \quad & penalty\_factor \times ((x - goal\_pos\_x + penalty)^2) + (vel\_x\_up)^2 + (vel\_x\_down)^2 \\ & + penalty\_factor \times ((x\_pos\_down - goal\_pos\_x)^2) \\ & + penalty\_factor \times ((x\_pos\_up - goal\_pos\_x)^2) \\ & + (y - goal\_pos\_y - (BALL\_DIAMETER/2) - goal.bottom\_height/2)^2 \\ & + overlap\_count \times hard\_constrain\_factor \end{aligned} \tag{4.1}$$

where:
x : *the x position of the ball at the end of the simulation run*
y : *the y position of the ball at the end of the simulation run*
$goal\_pos\_x$: *the geometry center x position of the goal object*
$goal\_pos\_y$: *the geometry center y position of the goal object*
vel_x_up : *the velocity in x direction when the ball falls into the plane for the first time and the ball is slightly over the plane*
vel_x_down : *the velocity in x direction when the ball falls into the plane for the first time and the ball is slightly under the plane*
x_pos_down : *the x position of the ball when it is through the plane for the first time and it is slightly under the plane*
x_pos_up : *the x position of the ball when it is through the plane for the first time and it is slightly over the plane*
$goal.bottom\_height$: *the thickness of the bottom of the goal object*
$overlap\_count$: *the number of the overlappings in the current layout*
$penalty$: *set to* 1 *if this run fails otherwise* 0

Since in our simulation the x-and y-positions of the obstacles are in the intervals from 0 to 0.5, the quadratic term of the position difference is smaller than 1; a penalty and the penalty factor are added accordingly. If the run fails, the penalty will be set to 1 and otherwise 0; therefore, the quadratic term of position difference will be greater than 1. This would make the minimum cost unambiguously related to the successful solutions. Without the penalty, some gradient-based algorithms may exit before computing a successful solution due to local minima. Considering that the intersection of obstacles is unrealistic, the hard constraint factor is set to 50, which forces the optimizer to sample in the non-overlapping combinations.

As mentioned above, we also keep track of the best-scored combination of obstacles out of the successful and feasible solutions. A solution is scored once it creates a successful run and there are no intersections between all the obstacles. The score is an additive result of the velocity of the ball in x and y direction, when the ball enters the plane for the first time and it's slightly above the plane, and the number of times that the ball bounces in and out of the plane in the simulation.

$$\begin{aligned} score = & penalty\_factor\_1 \times v\_x\_up^2 + penalty\_factor\_1 \times v\_y\_up^2 + bounce\_count \\ & penalty\_factor\_2 \times ((x\_pos\_up - goal\_pos\_x + 1)^2) \end{aligned} \tag{4.2}$$

The best solution is also the minimum regarding the score function. Because low velocity in both directions and low bounce count represent a robust solution in our case. Through our observations of the experiments, the term measuring the difference of the position of the ball through the goal plane for the first time and the goal object geometry center is appended. This term helps to avoid that these solutions are the best solution, where the ball first hits the edge of the goal object then slides slowly into the "cup".

In this case the velocities of the ball when it enters the goal plane are small, but hitting the goal edge may result in a faulty run in the real world.

5 Experiments

In this chapter, the outputs of the optimizer using different optimization algorithms are depicted. The trajectories of the ball are illustrated with the approach described in chapter Related Work 3.

5.1 Computing one valid solution

In this section, the optimizer is designed to find one valid solution, and in this connection the run time and the score of the output is recorded. The score is calculated by the score function 4.1.

5.1.1 Conjugate Gradient (CG)

In CG we have tuned the parameter step size, figure 5.1 illustrates when the trajectory of the ball. The ball has a complex trejectory, and it hits the edge of the obstacle at the right side of the goal object first. Then it changes its direction and hits the goal object, and then it slides down into the goal object. After these two hits the velocities are reduced, therefore the score is moderate (25.89). It took the optimizer 121.81s to find this solution.

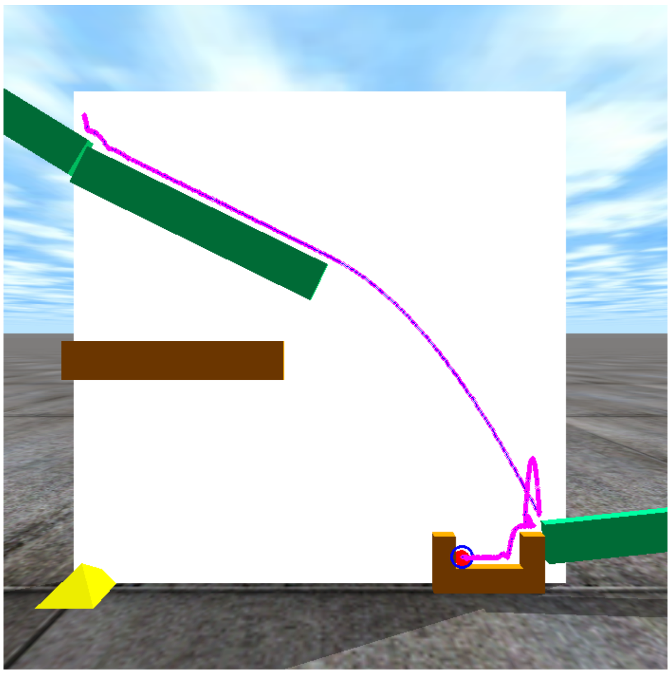


Figure 5.1: The output of using CG with step size 0.1 in simulation

Figure 5.2 shows the result of using CG with step size 0.01. After hitting the edge of the fixed given obstacle, the ball falls straight into the goal object. The ball is accelerated in the air by gravity, hence it falls through the goal object opening with relative high velocity. This results in a bounce and a high score (34.19). It took the optimizer less than a second to find this solution (0.26s).

From both CG run times, we can see that the variance of the run time is large (one is 121s, the other is 0.26s). This happens regularly with the algorithm CG, as the algorithm is start position sensitive. That is, if the minimum lies near the start position, then the algorithm will be terminate soon. Sometimes it takes CG a long time to even find one valid solution.

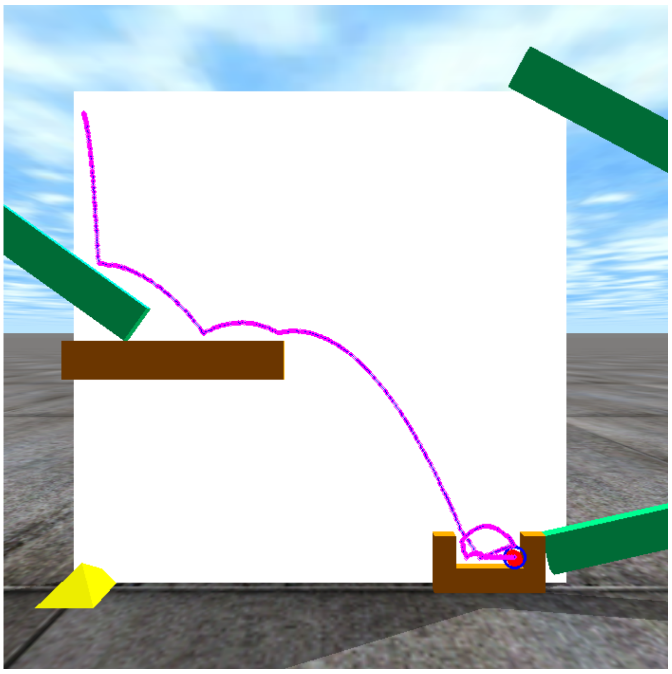


Figure 5.2: The output of using CG with step size 0.01 in simulation

5.1.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

Two experiments are conducted on CMA-ES , one with a fixed start point for all iterations, the other with random start points for each iteration. Figure 5.3 depicts the output of deploying CMA-ES with a fixed start point. In the simulation, the obstacles that interact with the ball are almost horizontal, where the friction cancels out the horizontal velocity of the ball. However, in the vertical direction it is still accelerated, hence the score is moderately good - 30.95. It took CMA-ES 13.46s to find this solution.

Figure 5.4 shows the result of CMA-ES with random start points, where the ball features high velocities when it hits the edge of the goal object. This may lead to a faulty run in reality,as it seems that the success of the run depends on the hit, therefore, it has a high score of 41.49. The run time is 10.07s.

5.1.3 Differential Evolution (DE)

There are not many parameters left for tuning by the user with the DE algorithm . So, we only conducted one experiment on DE to find one valid solution. Figure 5.5 shows the result. It is similar to the one that CG found with step size 0.01, where the ball also bounces back once above the goal object entry plane. Thus, this solution also has a similar score as the CG one - 35.82. The run time is 7.15s.

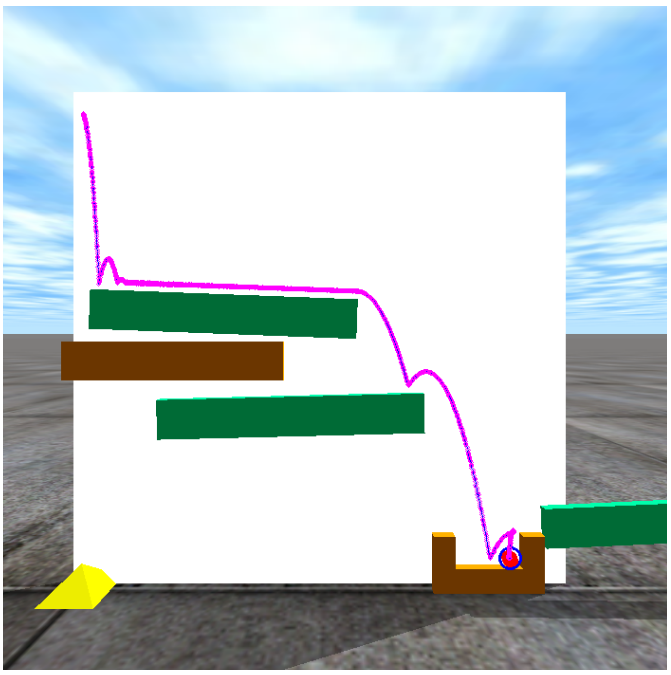


Figure 5.3: The output of using CMA-ES with fixed initial point in simulation

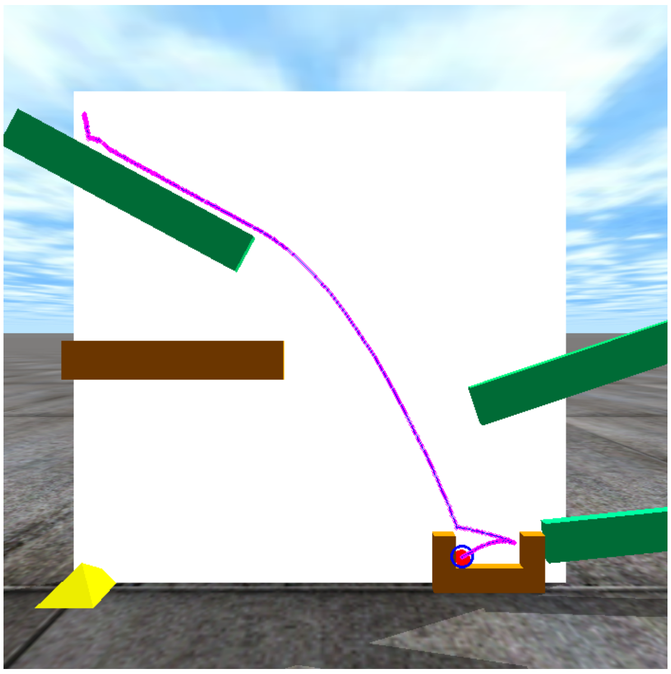


Figure 5.4: The output of using CMA-ES with random initial points in simulation

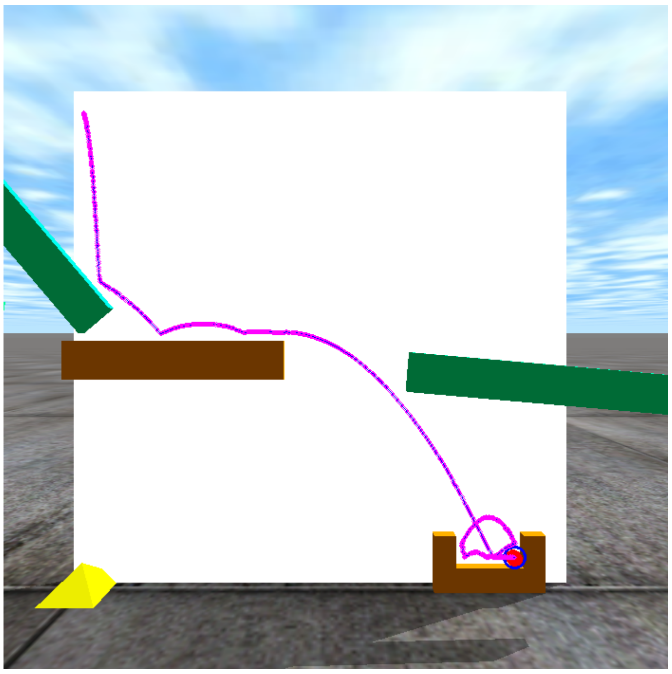


Figure 5.5: The output of using DE in simulation

5.1.4 Random Search (RS)

As our benchmark, RS performed moderately well in finding one valid solution, with a run time of 0.40s resulting in a scored 34.83 solution. Figure 5.6 depicts the result, which is similar to the one found by DE and CG with step size 0.01.

5.1.5 SLSQP

There were two experiments conducted on SLSQP: one with step size 0.01, one with step size 1e-8. Figure 5.7 exhibits the outcome with step size 0.01. With this solution, the ball hits the edges of the goal obstacle twice, which in the reality may lead to a faulty run. This output featured a score of 40.31 and run time of 6.88s.

Figure 5.8 depicts the output with step size 1e-8. After sliding on the fixed horizontal obstacle, which was added for increasing the difficulty of the game, the ball then hits the edge of the goal object and bounces into the goal object. The long trace in the air leads to high velocities, this resulted in a high score of 33.45. It took the optimizer 14.85s to find this solution.

From the examples of the experiment on finding one valid solution, we can see that the solutions found are similar and they are not necessarily robust in fulfilling our criterion.

5.2 For a robust result : computing ten valid solutions

From the trajectories of the ball we can see that, the first solution found is not necessarily a good one. In this section, the results of the best solution out of ten valid solutions of each algorithms will be presented. That is, the algorithm will return the best out of the ten valid solutions it found during a run. The best valid solution is the one with the lowest score according to the evaluation score function.

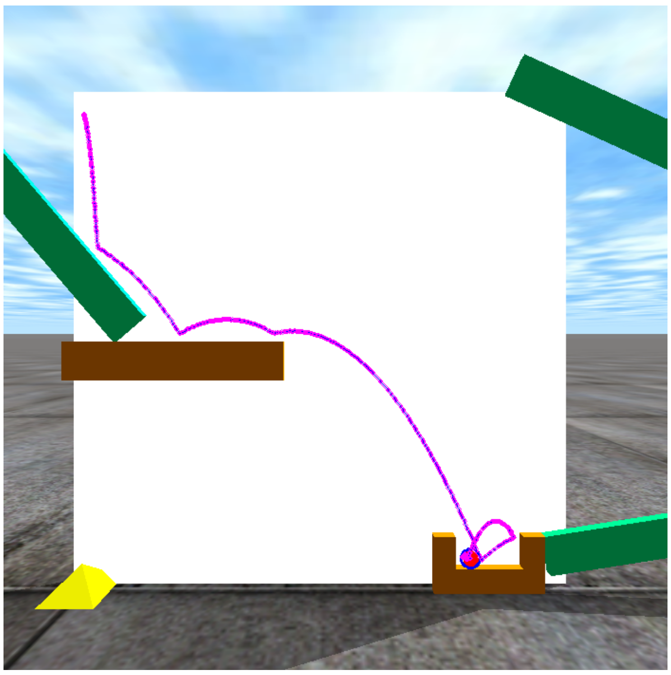


Figure 5.6: The output of using RS in simulation

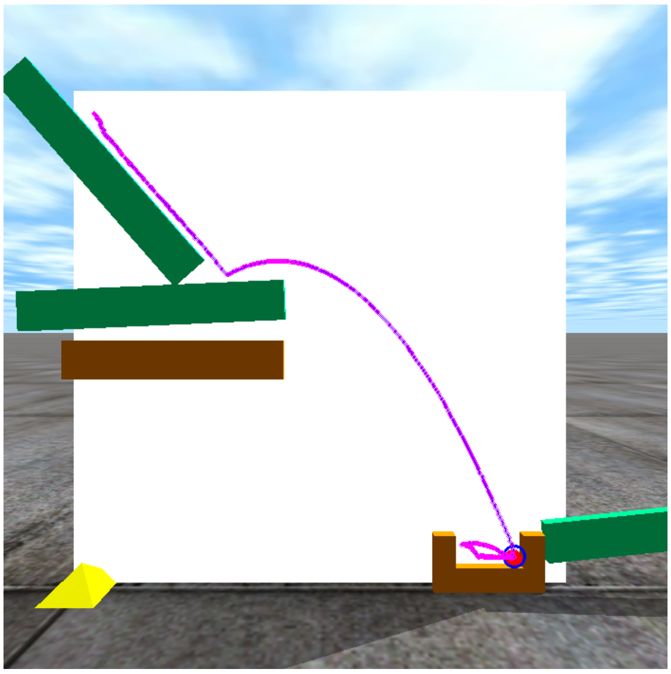


Figure 5.7: The output of using SLSQP with step size 0.01 in simulation

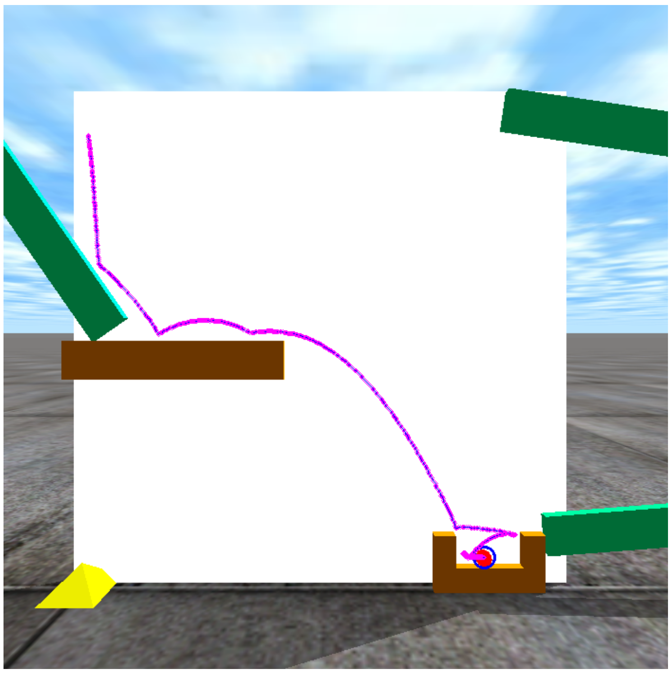


Figure 5.8: The output of using SLSQP with step size 1e-8 in simulation

5.2.1 Conjugate Gradient (CG)

Like the experiments on computing one valid solution 5.1, two experiments were conducted with CG and different step sizes. Figure 5.9 shows the selected output - out of ten valid solutions using CG with step size 0.1. The ball hits obstacles three times and then, with the third hit, it bounces into the goal object. The three hits result in smaller velocities, since the energy is lost in the hitting of the obstacles. Thus, this solution achieved a good score of 24.37. It took the algorithm 87.32s to return the best output.

Figure 5.10 depicts the solution returned by CG with step size 0.01. The layout is similar to our layout in reality, see figure 4.1. The ball hits the edge of the goal object and thus slightly affects the score - 29.09. It took the optimizer 11.02s to finish the task.

5.2.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

CMA-ES was also run twice, once with random start positions for different iterations, and the second time with a fixed start position for all iterations. Figure 5.11 illustrates the output of CMA-ES with fixed start position. The layout is robust: with two obstacles forming a slope, the ball is not accelerated as much as in the air. It hits the edge of the goal obstacle as well, but this does not produce enough energy to result in a bounce; therefore, generally speaking, it deserves its good score of 21.14. It took the optimizer 68.58s to terminate.

Figure 5.12 shows the output of CMA-ES with random start positions for each iteration. The layout is similar to those found by computing one valid solution as shown in figure 5.1. It starts with a new position every time, so the tentatively adapted covariance matrix does not necessarily work for the new position, and the algorithm needs to adapt the matrix from scratch again. Hence, in this manner the matrixes are not accumulatively adapted. This solution features a score of 34.20, and a run time of 28s.

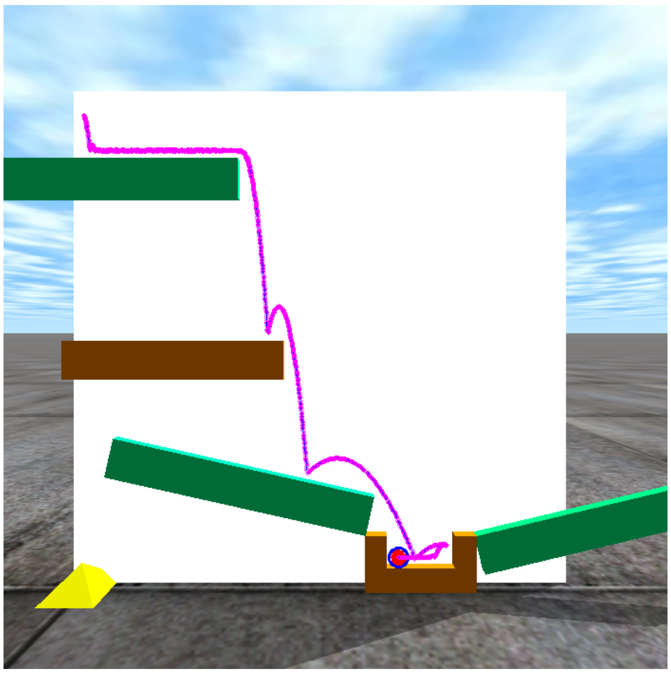


Figure 5.9: The output of using CG with step size 0.1 in simulation

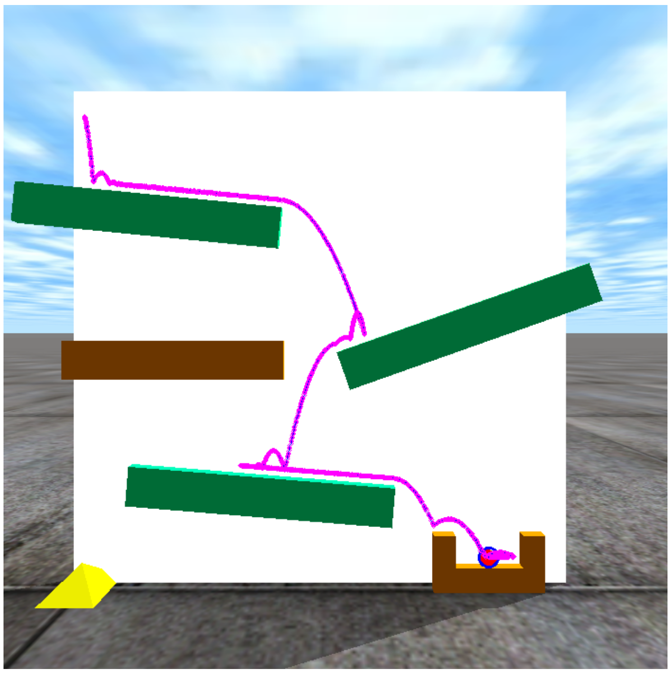


Figure 5.10: The output of using CG with step size 0.01 in simulation

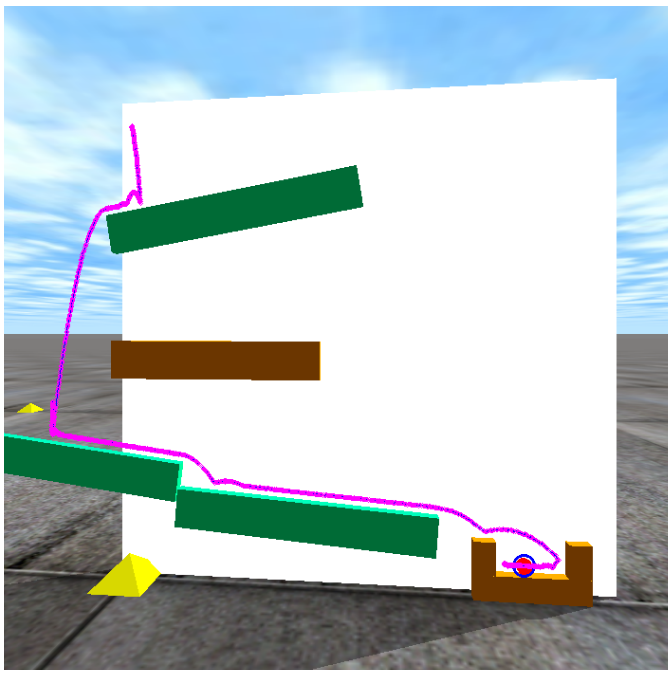


Figure 5.11: The output of using CMA-ES with fixed start postion

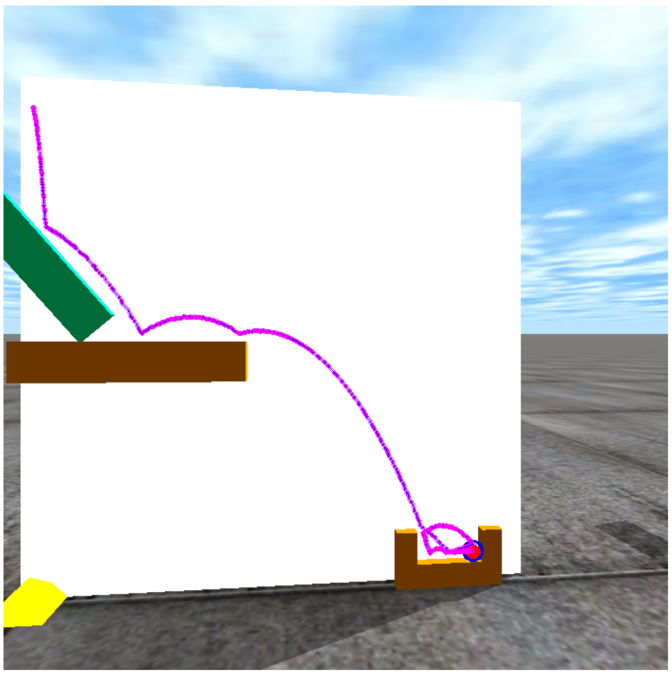


Figure 5.12: The output of using CMA-ES with random start postions

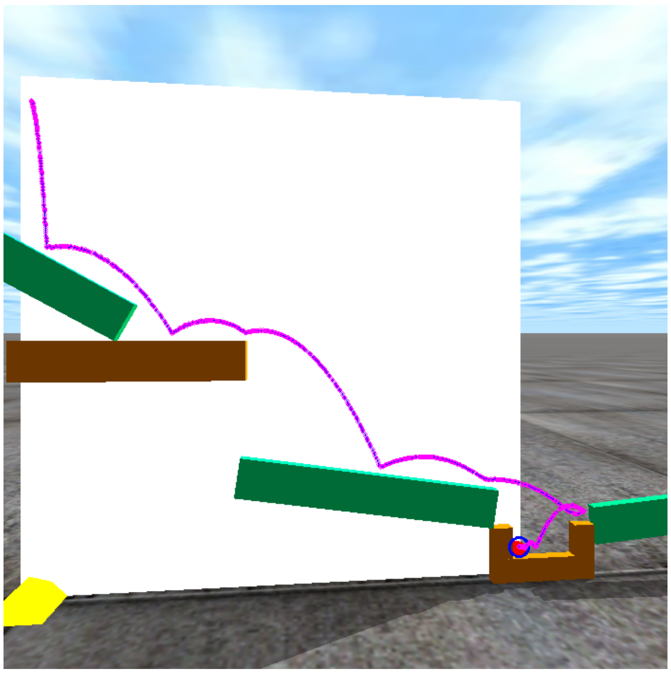


Figure 5.13: The output of using DE

5.2.3 Differential Evolution (DE)

The result from DE is an improved version of the valid solution found first, see figure 5.5. The layout in figure 5.13 achieved a better score of 27.45 too.However, In trade-off of the score, on the downside, it took 115.49s to compute this solution.

5.2.4 Random Search (RS)

In this run, the RS showed a moderate performance; within 82.53s, it found a solution that scored 23.31. The solution is quite robust based on our criteria. It lets the ball slide down near the goal object, then lets it hit the obstacle on the right side of the goal object. With that hit, the ball slides down into the goal object with small velocities.

5.2.5 Sequential Least SQuares Programming (SLSQP)

The layout found through SLSQP, with step size 0.01, achieved a good score (22.20). The trajectory in figure 5.15 shows that the ball hits the edge of the obstacle at the right side of the goal object, which moves the ball moderately near the center of the goal object when it falls through. The algorithm took 83.90s to return this solution.

The output of SLSQP with step size 1e-8 displays a robust valid solution, illustrated in figure 5.16. In the simulation the ball hits the edge of the obstacle on the left side of the goal object; it then changes its direction and lands on the obstacle on the right side. It slides back to the goal object with low velocities. The solution has a relative low score of 21.37 and was found relatively quickly in 51.75s.

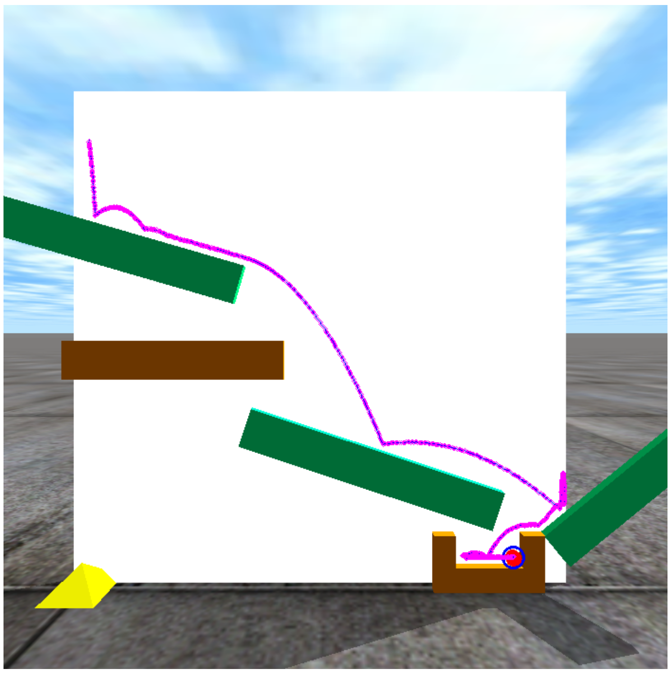


Figure 5.14: The output of using RS

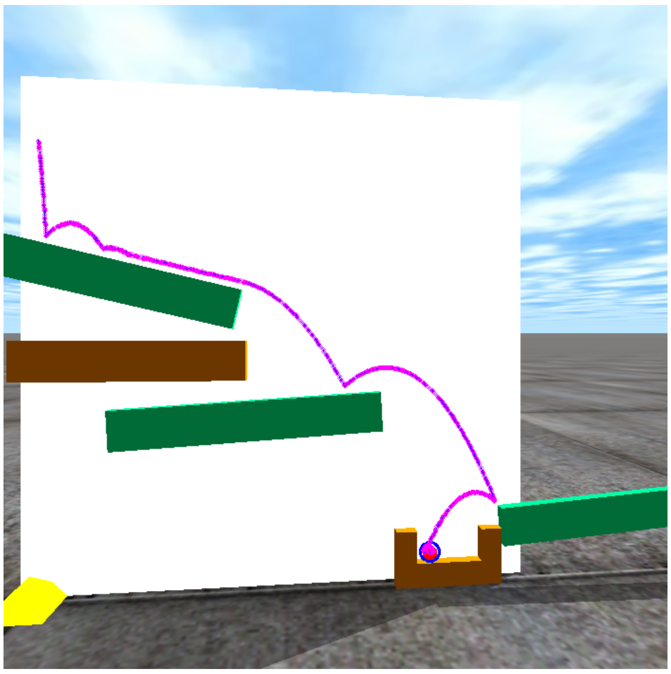


Figure 5.15: The output of using SLSQP with step size 0.01

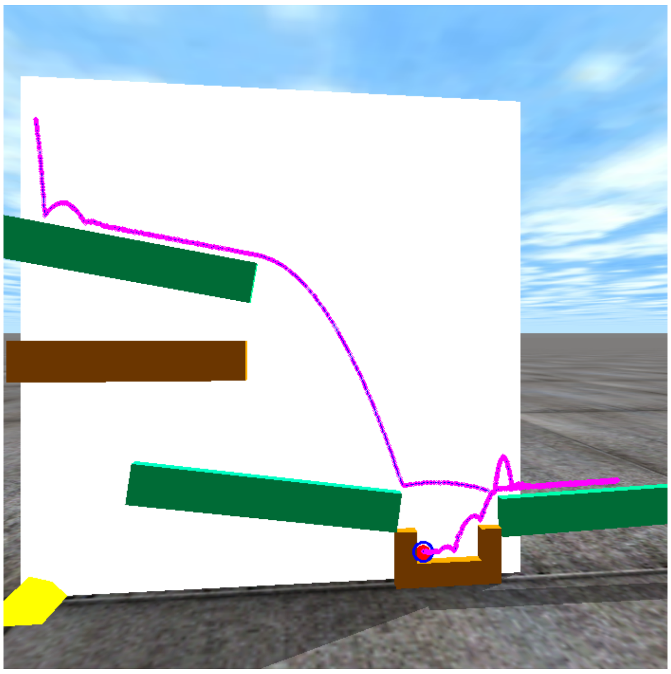


Figure 5.16: The output of using SLSQP with step size 1e-8

5.3 The longer the computational time the better the results ? Computing twenty valid solutions

The outputs of computing ten valid solutions have improved. Thus, we wanted to explore whether the output will be improved by computing twenty valid solutions.

5.3.1 Conjugate Gradient (CG)

The output of CG with step size 0.1 (see figure 5.17) was quite robust; the hits on the obstacles next to the goal object reduced the velocities which resulted in a good score (21.84). However, the run time was unsatisfying as it took 224.60s to finish the task.

Furthermore, the output from CG with step size 0.01 was robust too. See the trajectory of the ball in figure 5.18 where the ball first bounces on the obstacle on the right side of the goal object and then slides down into the goal object. In this manner, the ball is not accelerated by gravity as compared to the air. This solution features a score of 22.80 and the run time is 160.96s.

5.3.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

The CMA-ES with fixed initial position for all iterations in a run returns a satisfying result (scored 21.39). Though, it is slightly worse than the one found by CMA-ES with a fixed start position for computing of ten valid solutions, which scored 21.14. The solution illustrated in figure 5.19 is rather the same as found by CG. CMA-ES terminated within a minute in this run (52.55s).

The layout from CMA-ES with random start positions for all iterations as shown in figure 5.20 is less robust than the one found with a fixed start position(scored 23.45). It took CMA-ES with random

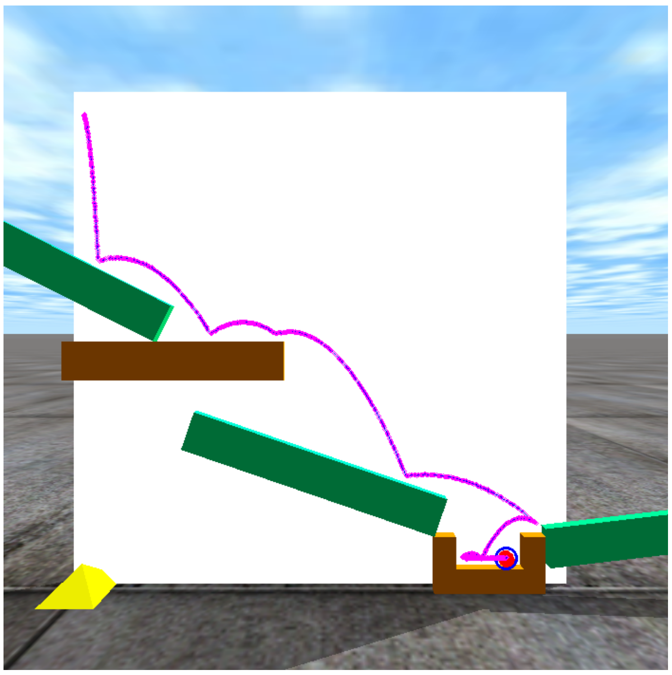


Figure 5.17: The output of using CG with step size 0.1

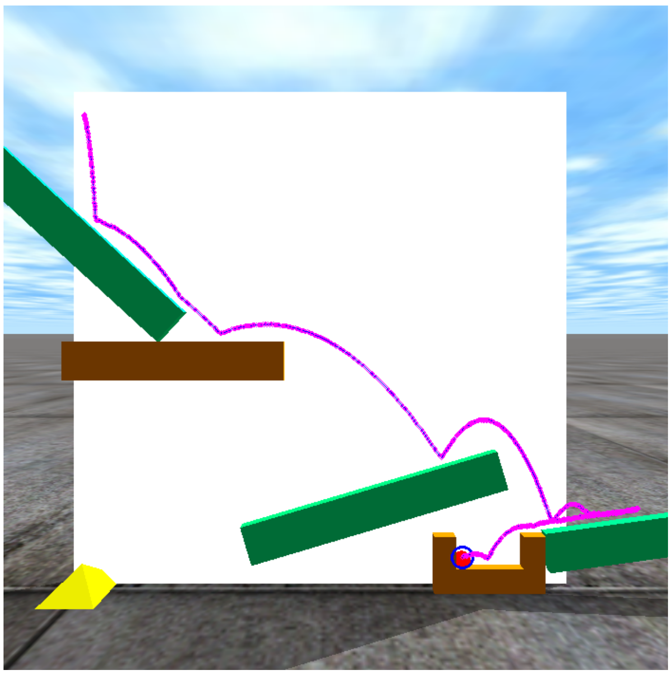


Figure 5.18: The output of using CG with step size 0.01

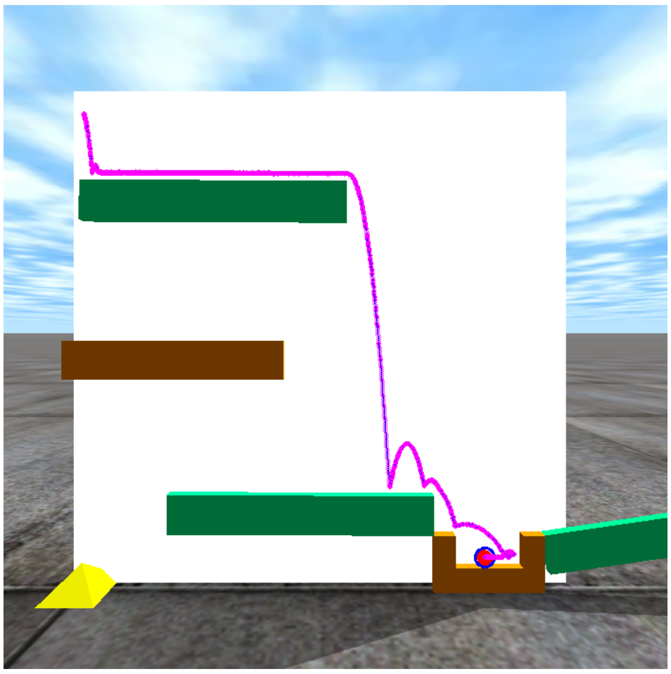


Figure 5.19: The output of using CMA-ES with fixed initial position

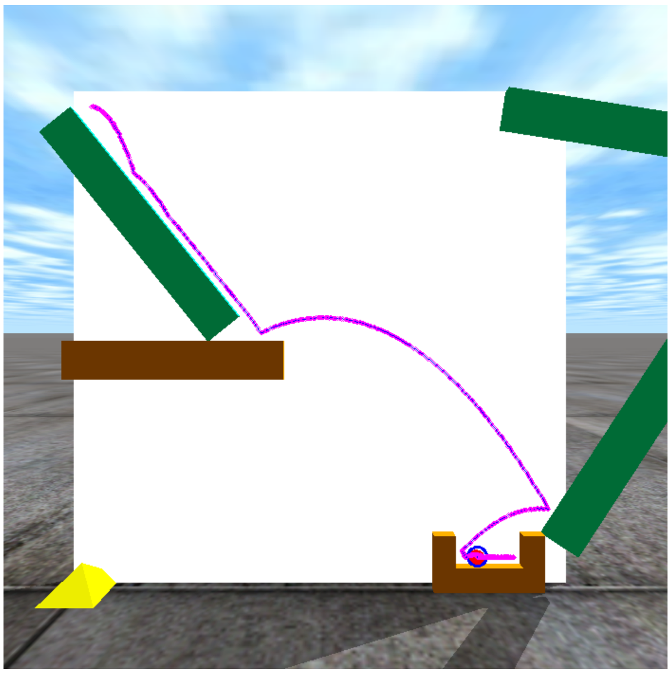


Figure 5.20: The output of using CMA-ES with random initial positions

start positions, 79.82s to finish and find twenty valid solutions. During our experiments, CMA-ES, with random start positions often exceeded the maximum iteration number and therefore failed to find twenty valid solutions. More will be discussed in chapter Evaluation 6.

5.3.3 Differential Evolution (DE)

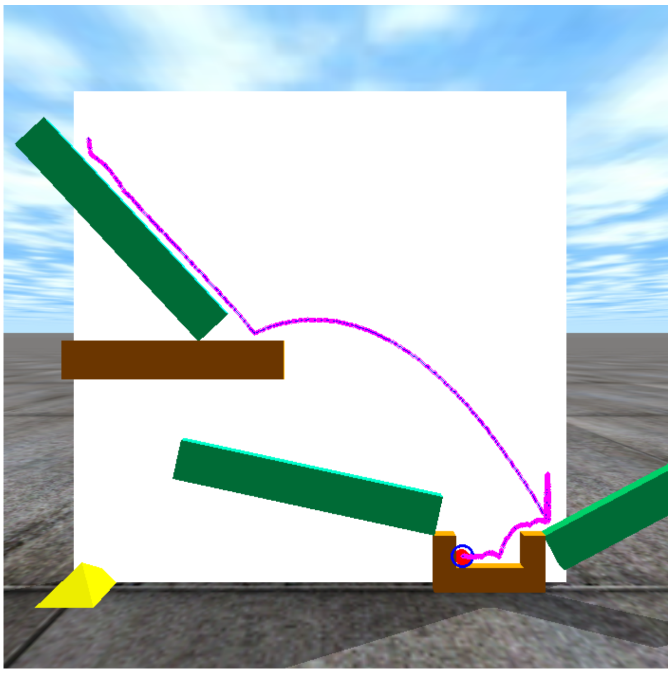


Figure 5.21: The output of using DE

The output from DE as depicted in figure 5.21 is similar to the one found by CG with step size 0.1. It features a score that is typical for a robust solution, that is 22.46. The run time is moderate for computing twenty valid solutions: 108.21s.

5.3.4 Random Search (RS)

The layout from RS is also a regular robust layout and a score of 22.77. Though the trajectory in figure 5.22 shows that the ball hits on the edge of the obstacle on the right side of the goal object directly after hitting the edge of the fixed given obstacle. After accelerating in the air, the last hit on the obstacle is a strike, which needs to be tested in the real environment. The algorithm took 274.65s to find twenty valid solutions. The longest run time indicates that when computing for more than one valid solutions, random sampling loses its advantages in run time.

5.3.5 Sequential Least SQuares Programming (SLSQP)

Figure 5.23 depicts the solution returned by SLSQP with step size 0.01, which is similar to the one found by DE and CG with step size 0.1. It also features a moderate score of 22.85, offering a regular robust solution. The algotihms took 110.19s to terminate.

The layout displayed in 5.24, as returned by SLSQP with step size 1e-8, shows another variety of those frequently seen. It "forms" a bridge, which lets the ball fall onto the obstacle on the right side of the goal

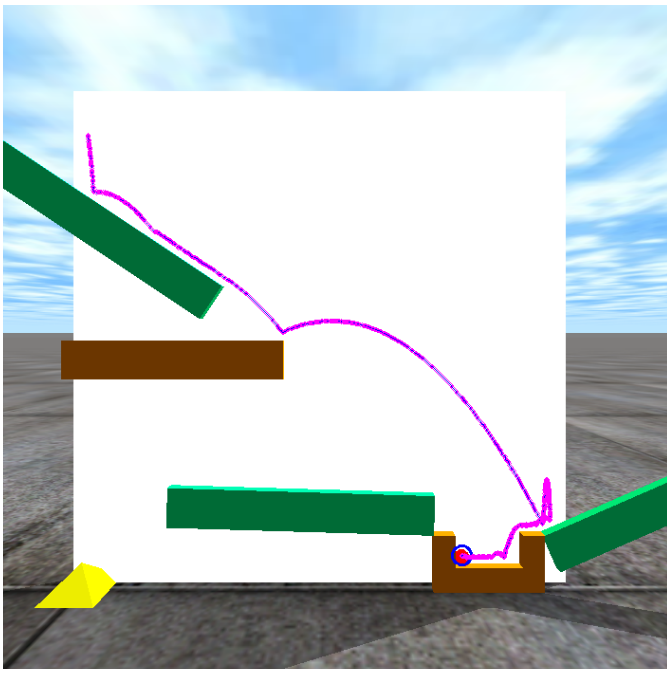


Figure 5.22: The output of using RS

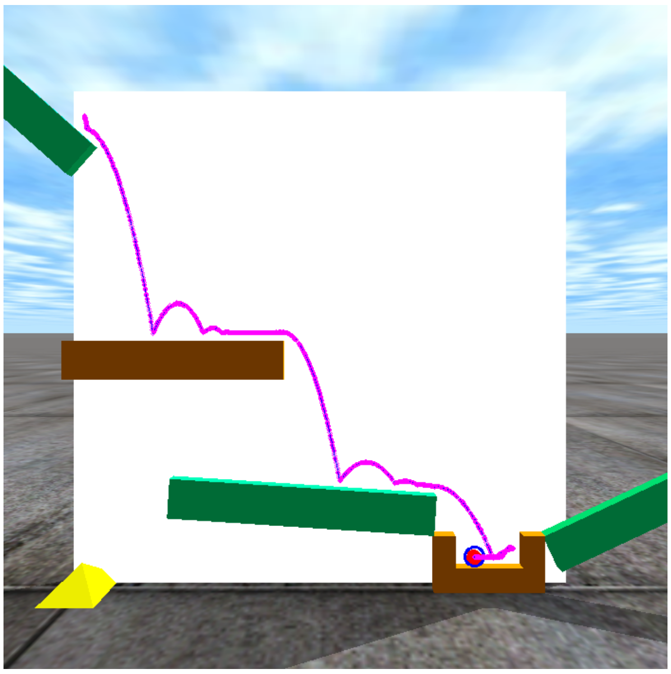


Figure 5.23: The output of using SLSQP with step size 0.01

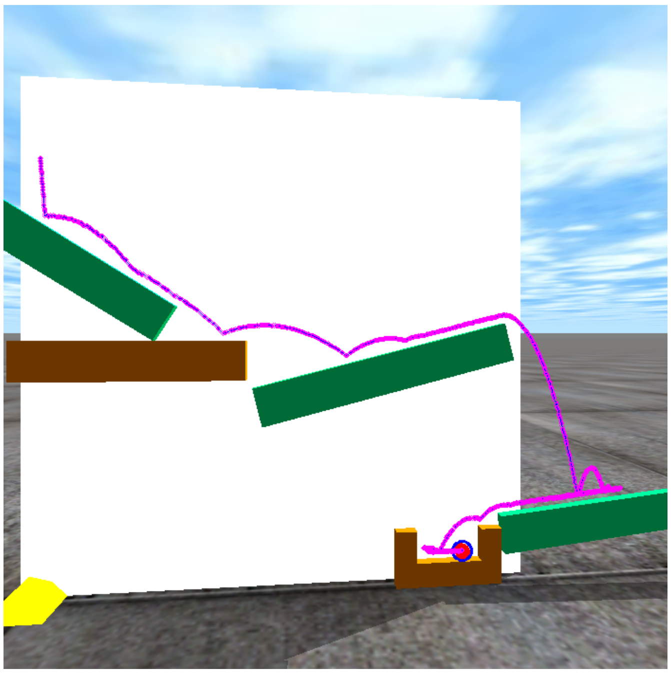


Figure 5.24: The output of using SLSQP with step size 1e-8

object, then lets the ball slide down into the goal object with small velocities. The score (22.88) indicates that it is a robust solution. The run time of this run is 134.62s.

6 Evaluation

For better performance, various optimization algorithms with different parameters are applied, CG and SLSQP for gradient-based, and further, Differential Evolution (DE), Covariance Matrix Adaptation Evolution Strategy (CMA-ES) with random initial vectors for each iteration, or fixed initial vectors and random search for non-gradient-based. Roughly speaking, gradient-based methods converge fast as expected, but they may fail to compute the desired numbers of valid solutions before they exit the program, which is called early exit henceforth. I.e., gradient-based methods will break the program as soon as they encounter a zero-gradient. In the following, we use *valid* to denote that a solution is successful in the simulation and shows no overlapping in the layout. In our case, this would happen when the ball does not intersect with any of the obstacles and falls directly to the ground for two successive runs. Where the zero gradient occurs, gradient-based methods would fail, which happened frequently at the beginning of our experiments with a naive cost function 6.1:

$$\begin{aligned} cost = \; & (x - goal\_pos\_x + penalty)^2 \\ & + (y - goal\_pos\_y - (BALL\_DIAMETER/2) - goal.bottom\_height/2)^2 \end{aligned} \tag{6.1}$$

It is likely that at the beginning of sampling, the ball does not hit any of the obstacles and the gradient-based methods would exit the program.

Meanwhile, non-gradient-based algorithms were able to compute valid solutions from the very beginning. This corresponds to the advantage of black-box optimization algorithms, which have the ability to handle non-differential, non-linear and multimodal cost functions. Note that even though our cost function consists of quadratic terms only, it is neither continuous nor differential to the input vector (positions and angles of the movable obstacles). The input vector is passed to the dynamic model, and the model produces velocities, positions, and bounce counts needed for the cost function. This explains why the *Random Search method* performs fairly well, as it always return a valid solution within a reasonable run time.

After several adjustments of the cost function, which is further discussed in chapter 5, all optimization algorithms are capable of computing at least one valid solution. Once we find one valid solution, it will be evaluated by the score function under 4.2, which represents the robustness in our case. The reason for using a slightly different function than cost function 4.1 is that the cost function serves mainly to help the algorithm to sample in feasible parameter regions and to avoid an early exit of the gradient-based methods. The score function under 4.2 measures the core features of a valid solution. It selects the output from a range of valid solutions.

Generally, letting optimization algorithms run multiple iterations creates better results. It is basically a tradeoff between run time and output quality. To find a good balance, several experiments on run time and output scores were conducted, which is further discussed in section 6.1

Two kinds of gradient-based algorithms, CG and SLSQP as detailed in section 2.2.1 and each with two different step sizes, are run. Three types of non-gradient-based methods are used. These are CMA-ES, DE and random search as explained in section 2.2.2, where the random method serves as a benchmark for the algorithms. In order to find the balance between run time and output quality, each algorithm is run ten times to compute one valid solution, ten valid solutions, and twenty valid solutions.

Figure 6.1 reveals a first glance at the optimization landscape of our problem, showing the average run time and the output scores, evaluated by the score function under 4.2 - for the algorithms to find one valid solution. DE, SLSQP with small step size, and random show all small deviations in time. For DE, CMA-ES with fixed initialization and random methods it took less than 10 seconds to find one valid solution. An attentive reader may realize that the left bound of the CG with step size 0.01 is smaller than 0, which is physically impossible. It is due to the excessive variance in the run time, the shortest recorded of run time was 1.38s and the largest is 198.26s. The large variance in scores for all algorithms is reasonable, because the first valid solution found is not necessarily a robust one.

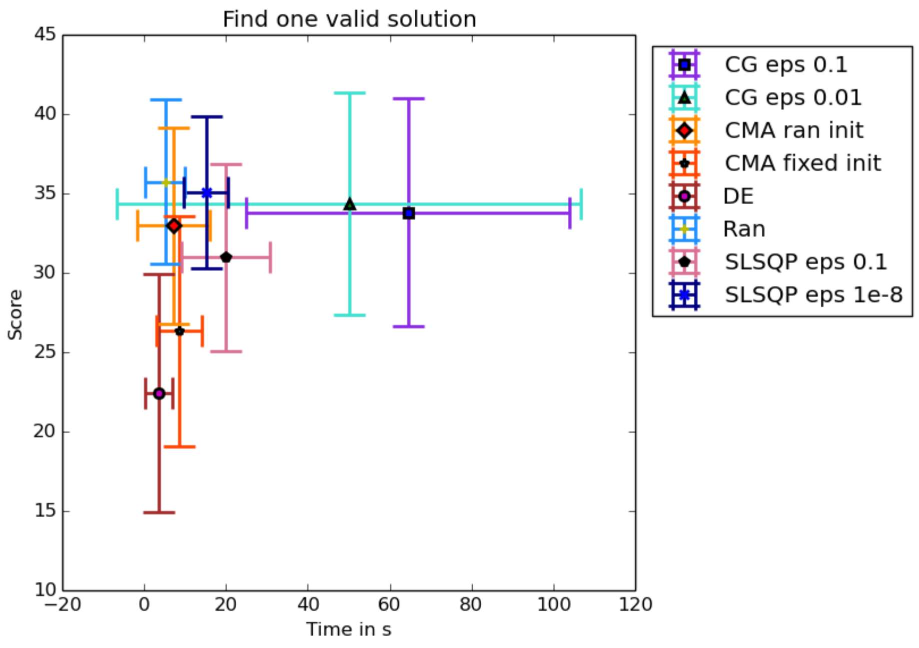


Figure 6.1: Average run time and scores for each algorithm to find one valid solution

From figure 6.2, showing the average run time and scores for ten valid solutions, we can see that all methods tend to shift to the right, meaning that it took more time to finish the tasks. This is because the average run time is calculated only from successful runs, which generated at least ten valid solutions. In this case, both of the CG methods failed to compute ten valid solutions in one of the ten trials. The high variance in run times indicates that CG strongly depends on the start positions. Furthermore, the random method slides down from the top two to the third-to-last place in run time. This suggests that other algortihms, except for CG, have advantages in computing multiple valid solutions. Compared to the last graph of one valid solution as shown in figure 6.1, all the scores go down, arguing for better output qualities, which indicates it is worthy to let the algorithms run a bit longer.

If we let the algorithms continue to run till twenty valid solutions are found, we get the average performance shown in figure 6.3. All the algorithms took longer to finish the tasks, and the random methode features the worst run time performance. CMA-ES with fixed start positions works smoothly and well, for twenty valid solutions it only needs around 20s more than computing ten valid solutions. The CMA-ES with fixed start positions has also the least variances both in run time and scores, which reveals the stability of CMA-ES with fixed start positions. Meanwhile CMA-ES with random initialization has a slightly longer run time than DE, which still makes for good performance. However, in fact, CMA-ES with random initialization failed 3 times to finish computing twenty valid solutions during ten trials.

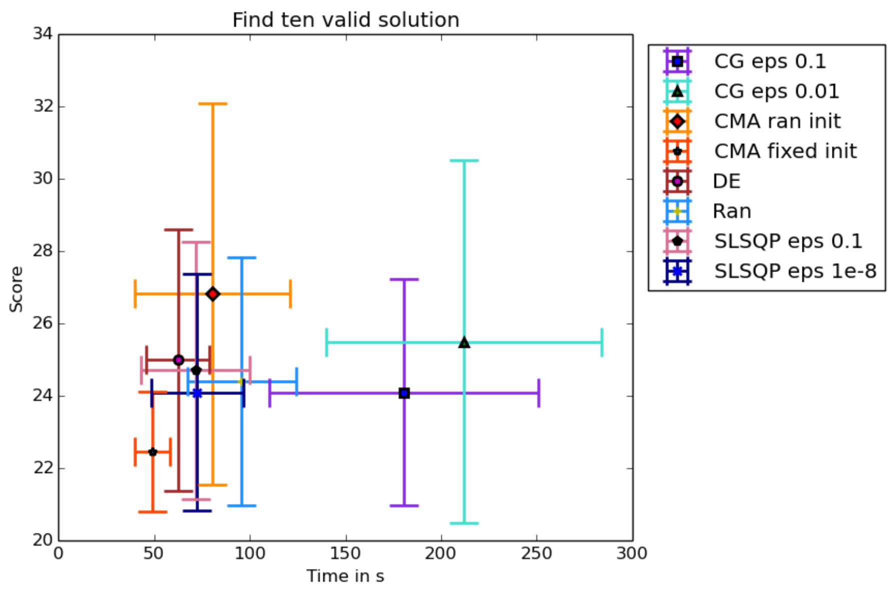


Figure 6.2: Average run time and scores for each algorithm to find ten valid solutions

Table 6, which compares the algorithms' performances with different amounts of valid solutions, gives a better overview with details on the performance. We can see that DE, CMA-ES with fixed start positions and Random are the only threes algorithms which always manage to compute the desired numbers of valid solutions. Nevertheless, all of the gradient-based methods encountered the early exit problem, while CMA-ES with random initializations exceeded the maximum iterations to find ten or twenty valid solutions .

Generally speaking, the low deviations of DE and CMA-ES with fixed initialization indicate stable performances, both in time and score. They either feature the best performance or slightly differ from the best. For computing larger numbers of valid solutions, CMA-ES with fixed initialization outperforms DE. Non-gradient based methods performed more satisfactorily in our case. In the meantime, SLSQP eps 1e-8 performs best under gradient-based algorithms. It did can handle any degrees of non-linearity as claimed by Kraft (1988), though in our case it suffered from early exit as did the other gradient-based methods. In contrast to CG, a small step size worked better for SLSQP in our case. It could be that with SLSQP the step size is for approximating second-order derivates, which, with a small change, would lead to a sharper turn than first-order derivates.

The Random method fell from the third best to the worst, which is encouraging as a benchmark, that it means other algortihms with fine techniques work better than random sampling. Remember that it is able to compute any desired number of valid solutions and that this is greatly due to the clean-up step described under section 4.2.1 before the solutions are passed to the dynamic model. The clean-up step may also be the reason for irritating some algorithms when they try to find the correlation between input vectors and the cost function. Random sampling did help to explore the search space in our case.

CMA-ES with fixed start positions has almost the same performance as DE, and meanwhile CMA-ES with random start positions performs much worse than with fixed initialization. This is because, with a random start position for each iteration, it needs to adapt the covariance matrix from scratch each time,

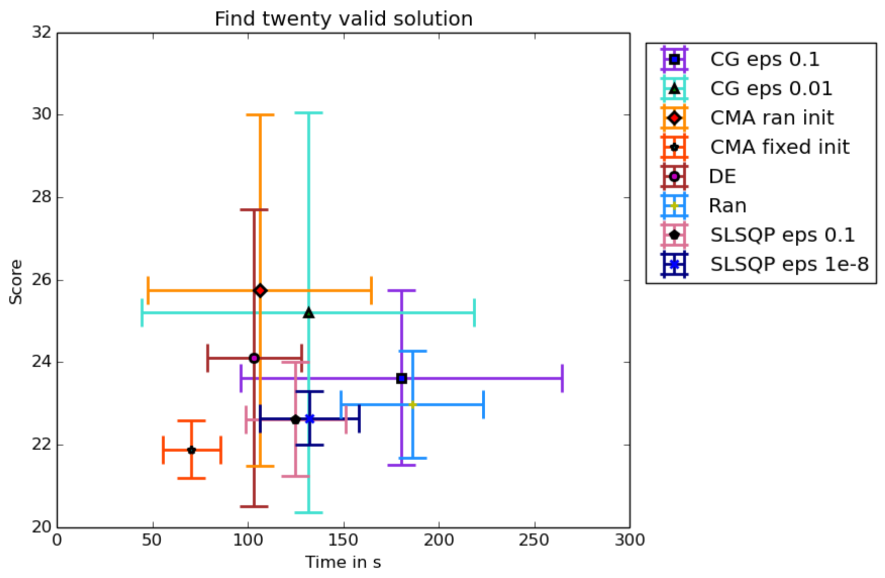


Figure 6.3: Average run time and scores for each algorithm to find twenty valid solutions

which does not take advantage of the adaptation principle of the algorithm. CMA-ES with random initialization did return moderate outputs if it did not exceed the number of maximum iterations. CG is meant for sparse linear systems, which may not be the case here.

6.1 Run time and robustness

Here we take a closer look to the performance of each algorithm. Each algorithm was run to compute one, ten and twenty valid solutions, where the successful runs are recorded. A successful run is a trial that the optimizer manages to compute for the desired number of valid solutions.

6.1.1 Conjugate Gradient (CG)

Two step sizes, 0.1 and 0.01, were tested using the CG. Figure 6.4(b) shows an interesting phenomenon, that the average run time for twenty valid solutions is slightly better than the run time for computing ten valid solutions. Tests for one or twenty valid solutions have at least one instance for which the run time is less than 10s. CG did converge fast if it was not distracted by faulty local minima. Both of the graphs from figure 6.4 show an improvement of scores over time, where between one to ten valid solutions the improvement is significant, whereas between ten to twenty it is tiny. For CG with small step size, the average run time for twenty valid solutions is even smaller than the one for ten valid solutions. It implies that the algorithm CG depends strongly on the start position, if the start position is near a group of valid solutions, the algorithm terminates fast indeed.

6.1.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

For CMA-ES, the initialization vector should remain constant for all iterations in one trial. This is because CMA-ES adapts a covariance matrix iteratively. If the initial vectors are different every iteration, the tentative covariance matrix is not built based on the current vector, hence it needs to be adapted

	CG eps 0.01	CG eps 0.1	CMA-ES random	CMA-ES fixed	DE	Random	SLSQP eps 1e-8	SLSQP eps 0.1
avg. time	50.08	64.47	7.18	5.15	3.55	5.15	15.10	20.03
dev. time	56.65	39.41	8.75	4.91	3.29	4.91	5.37	10.84
avg. score	34.34	33.80	32.97	35.71	35.71	35.05	30.96	32.31
dev. score	7.03	7.17	8.04	6.19	5.20	5.20	4.77	5.88
early exit / max iters	-	-	-	-	-	-	-	-

Table 6.1: Table of average performance on finding one valid solution

	CG eps 0.01	CG eps 0.1	CMA-ES random	CMA-ES fixed	DE	Random	SLSQP eps 1e-8	SLSQP eps 0.1
avg. time	211.91	180.38	80.61	95.69	62.38	95.67	72.47	71.66
dev. time	71.93	70.39	40.62	28.36	16.56	28.36	24.01	28.45
avg. score	25.49	24.09	26.82	24.39	24.98	24.39	24.09	24.70
dev. score	5.01	3.13	5.27	3.43	3.62	3.43	3.28	3.55
early exit / max iters	1	1	1*	-	-	-	-	1

Table 6.2: Table of average performance on finding ten valid solutions

	CG eps 0.01	CG eps 0.1	CMA-ES random	CMA-ES fixed	DE	Random	SLSQP eps 1e-8	SLSQP eps 0.1	
avg. time	131.42	180.00	105.96	70.38	103.11	58.20	158.03	74.62	72.99
dev. time	87.16	84.13	58.32	15.16	37.13	24.59	65.58	22.18	15.92
avg. score	25.21	23.63	25.74	21.89	22.99	24.11	12.65	12.32	12.43
dev. score	4.86	2.11	4.26	0.69	1.30	3.60	1.13	22.18	0.58
early exit / max iters	1	1	3*	-	-	-	-	1	

Table 6.3: Table of average performance on finding twenty valid solutions

from scratch, which means discarding the provisional results. This explains why CMA-ES with random initialization needs so long for computing more than one valid solutions. Moreover, figure 6.5(b) shows much better performance both on run time and output quality.

6.1.3 Differential Evolution (DE)

Figure 6.6 has an " L " shape, where the score represents the output quality in terms of robustness. The high variance of scores on finding one valid solution indicates that the first valid solution is not necessarily a good one. The small variance of time and score for ten valid solutions indicates that valid solutions may be grouped together. Whereas the high variance of run time at finding twenty valid solutions indicates that the clusters of valid solutions are rather small, that each cluster may have around ten valid solutions. The optimizer needs to travers the optimization landscape for an while to gather twenty valid solutions. Meanwhile, the small variance for scores of twenty valid solutions implies that it is not worth to continue finding valid solutions.

6.1.4 The random method

Figure 6.7 depicts a similar distribution as the one from DE, as shown in figure 6.6. The score does not improve much after ten valid solutions neither, i.e., the differences of scores between ten and twenty valid solutions are small. This is an indicator to stop running the optimitzer. Hence we do not let the optimizer continue to compute thirty valid solutions.

6.1.5 Sequential Least SQuares Programming (SLSQP)

Figure 6.8 shows the run times and scores of the output from the optimizer while using SLSQP with two different step sizes (it is denoted as eps, epsilon in the SciPy implementation). Figure 6.8 shows that with a larger step size the output is better when the program has to find more than ten valid solutions. During the ten trials to find ten or twenty valid solutions, both of them were failed once to find all the ten or twenty valid solutions before the maximum number of iterations was reached or the program exited too early. This can be seen from the marker number, for ten runs of ten or twenty valid solutions, the markers of the same kind are less than 10. With a smaller step size, the time for computing more valid solutions differs. Within less than 20 seconds the optimizer would find one valid solution in around 50s and within 100s it would find ten valid solutions if it is does not get stuck in some faulty local minima, and within an average of 75 seconds it will find twenty valid solutions. Meanwhile, with a larger step size, the time differences bewteen computing ten or twenty valid solutions are not so significant as with a smaller step size. With a small step size the optimizer is more likely to stay in one local minimum region and gather ten valid solutions. But for twenty it probably needs to jump to other local minima and thus the time increases. With a large step size, the optimizer may jump back and forth in the function landscape which leads to no significant differences in run time and a slightly better on the outputs. Roughly speaking, in our case, there are multiple feasible valid solution regions, which form the local minima area. The optimizer needs to have the ability to jump out of them and keep on searching to gather the desired amount of valid solutions.

Another noticeable point is that the suitable step size varies for different algorithms. For CG the step size needs to be relatively large - we tested 0.1 and 0.01. Meanwhile, tests were conducted for SLSQP 1e-8 and 0.01. It may need to be conducted for efficient step sizes, otherwise the algorithm may fail to return a result.

6.2 The path to valid solutions

In this section we will take a look into how the algorithms for finding valid solutions are intuitive. In the following we will present plots of the examined obstacles and of successful obstacle combinations, which are identified with in larger dark red markers. The order of the obstacles is determinded by their center x-direction posistion. The leftmost obstacle is henceforth the first obstacle and so on.

6.2.1 Conjugate Gradient (CG)

From figure 6.2.1 we can see the sampling distribution of the algorithm CG with step size 0.1. The first subgraph, which depicts the distribution of finding one valid solution, shows that the sampling path is clustered, which indicates that for the x-positions of the leftmost and rightmost obstacles have their own ranges. For the other subgraphs, the x-positions of the leftmost obstacle varias by a large range, whereas the rightmost still looks constrained. This is due to the clean-up step, where the idle obstacles are moved in a certain manner.

Figure 6.2.1 using CG step size 0.01 shows similar plots as shown in figure 6.2.1. That the plots in the example for computing twenty solutions to be disordered - for a more descriptive overview, we applied Principle Component Analysis (PCA) and Linear Discriminant Analysis (LDA) to reduce the dimensions of the space distributions of the obstacles. For a sufficient data amount, the data for PCA and LDA are all the sampled obstacles space information(x,y-positions and the angle) of ten trials for each valid solution amounts. The reduced dimensions plots from figure 6.2.1 reveal a clearly defined gap between the rightmost obstacle and the other two obstacles. Note that the PCAs of the three subgraphs have quite the same form, where the line of the third obstacles (also the rightmost one), always remain parallel to the other two, with a definite gap in-between. Whereas in the plots of the LDA, the clusters grow thicker as more the valid solutions need to be found. Moreover the gaps between the clusters of the rightmost obstacle and of the rest are getting smaller. This is because the LDA tries to project the data into the most separable lower dimension space. A smaller gap implies that the sampling positions of the obstacles have smaller distinguishing range as the sampling amount grows.

The reduced plots of CG with step size 0.01 shows a bit of a difference from the one with a larger step size. Both PCA and LDA have similar plots for one or twenty valid solution batches. However for the ten valid solutions batch, both PCA and LDA show that the data is rather inseparable. It could be that the ten trials for computing ten valid solutions have well mixed up space distribution of the obstacles.

6.2.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

Figure 6.2.2 shows the sampling distributions from CMA-ES with fixed initialization for all iterations. The points are sampled in a specific area in the space, in a wat that the points form a "band" in the middle of the space.

Subgraphs using the same reduction techniques have the almost the same results of reduced illustration, which indicates that the algorithm CMA-ES with fixed initialization for all iterations does not involve much randomness.

The plots of the points sampled by CMA-ES with random initialization for each iteration do have more points than the one with fixed initialization, although for twenty valid solutions the points are similarly clustered as in the subgraph for finding one valid solution. More points means longer run time. According to our experiments, CMA-ES does not need random initialization in our case.

The reduced dimension representations by PCA are similarly to those using the CG algorithm, see figures 6.11(a) to 6.11(e). CG also involves a covariance matrix in the algorithm, but in a gradient-based manner. For our case, non-gradient-based algorithms showed better performance since they don't exit the program before the task is finished. Moreover, PCA deploys eigenvector decomposition on the covariance matrix of sampling data and takes the first two greatest eigenvalues and their eigenvector to form the subspace. Therefore the samplings of CG and CMA-ES with fixed initialization for all iterations come out with similar distributions.

The reduced dimension representations by LDA for CMA-ES with random initialization for each iteration result in different representations on finding one, ten and twenty valid solutions. The first reduced distribution on finding one valid solution by LDA is again similar to those in figures 6.2.1 and 6.2.1. The three subgraphs in the LDA column with different shapes imply that the randomness of initialization has an influence on the sampling distribution. Unlike the subgraphs in fixed initialization, see figure 6.2.2, where the subgraphs of the same column have the same shape.

6.2.3 Differential Evolution (DE)

The plots from PCA for DE have common shape in line with other representations. Whereas the plots from LDA have distinct shapes between themselves. Interestingly, the reduced distributions with regard to finding ten or twenty valid solutions are more concise than the one for one valid solution. And the randomness of Differential Evolution (DE), which is realized through mutation and crossover between individuals as discussed before under section 2.2.2 , does not affect the shape of reduced distribution.

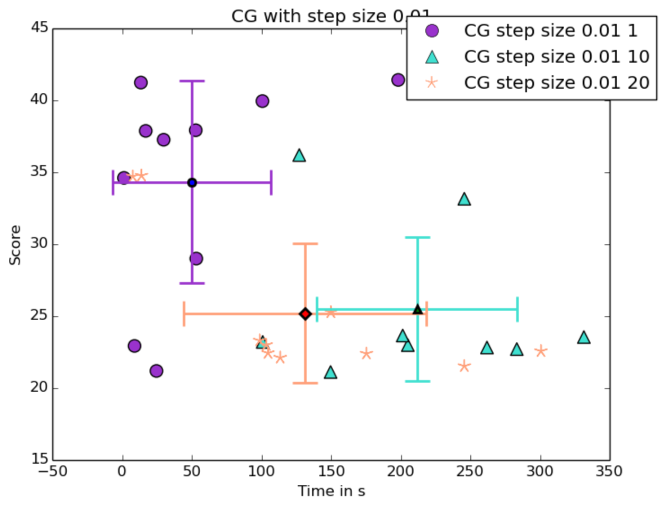
6.2.4 Random Search (RS)

The plots from Random Search (RS) shown in figure 6.2.4 does not show any regulations, especially the first obstacle, the leftmost one, which is in most cases sampled arbitrarily. We can see from figure 6.2.4 that the third obstacle, the rightmost one is the most distinguishable from others due to the clean-up step.

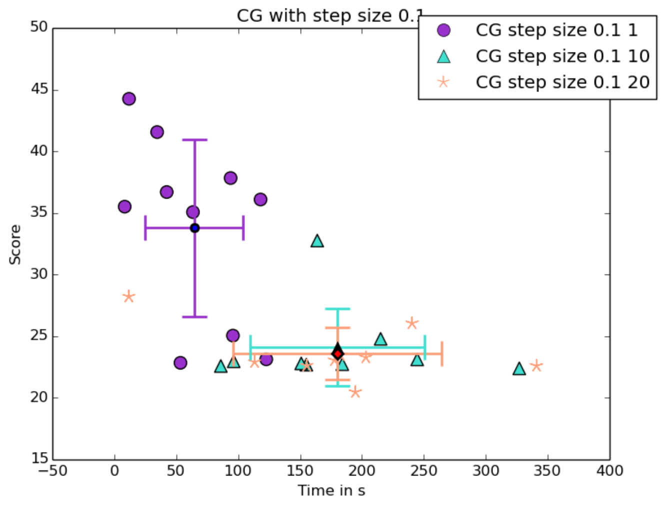
6.2.5 Sequential Least SQuares Programming (SLSQP)

The points in the plots in figures 6.2.5 and 6.2.5 that were sampled by SLSQP have certain structures, in the way that some points form lines, which suggests local minima and that the algorithm has the intention to sample in this region.

The reduced dimension shape of SLSQP is entirely different from the others. This means that the SLSQP algorithm uses another strategy to sample the data. Basically the representation of reduced dimension reveals that the leftmost and the rightmost obstacles are separated by the second obstacle, which corresponds to the space order when the obstacles are labelled. Moreover, in most of the plots of the reduced dimension expressions, the line formed by the second object is almost vertical to lthe ine or cluster formed by the third object. This implies that the second object usually has an angle that is opposite to the thrid obstacle in a layout. So that the third obstacle is likely to be placed at the right hand side of the goal object and the second one is probably on the left hand side of the goal object. They should have opposite angles so that they are facing each other and form a “V" shape so that if the ball bounces out of the goal object it can still bounce back after colliding with these two obstacles.

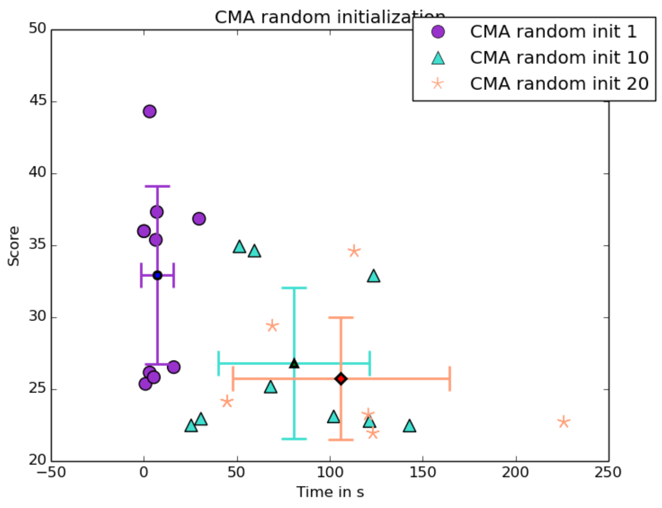


(a) CG eps 0.01

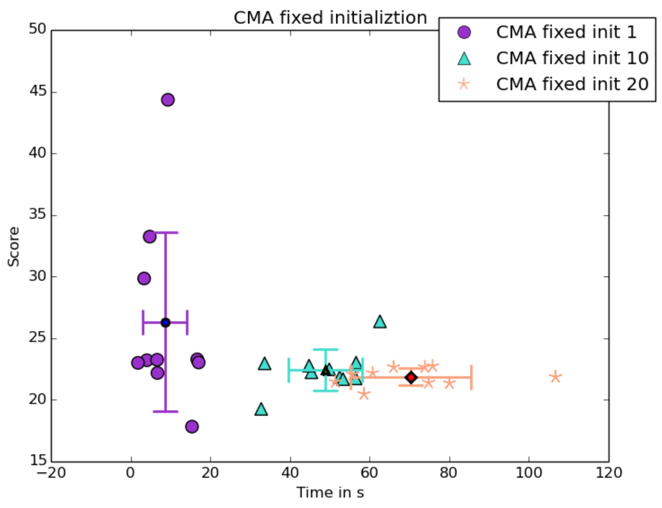


(b) CG eps 0.1

Figure 6.4: CG run time and score of best output



(a) CMA random initialization



(b) CMA fixed initialization

Figure 6.5: CMA run time and score of best output

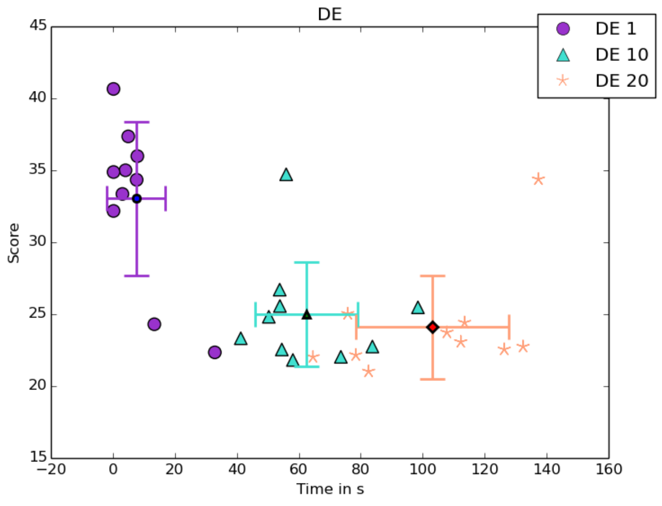


Figure 6.6: DE run time and score

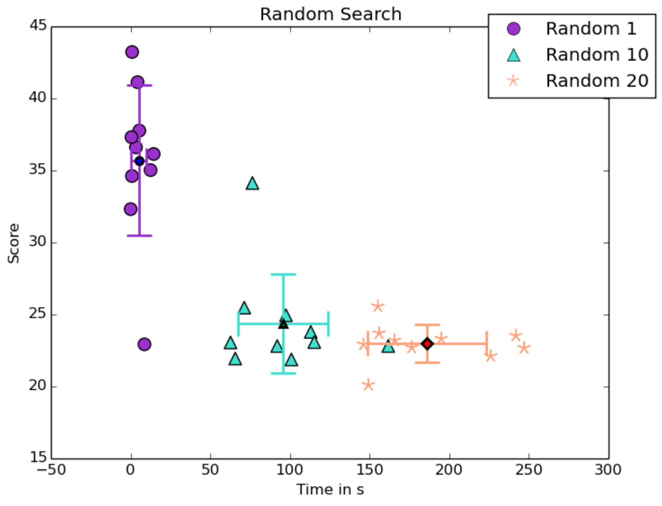
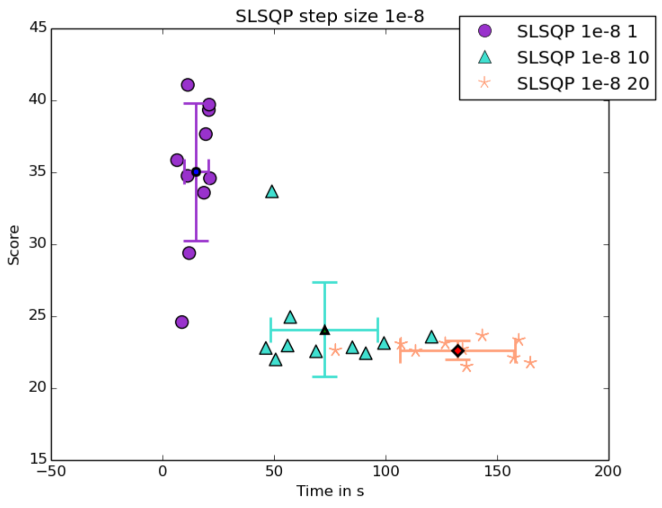
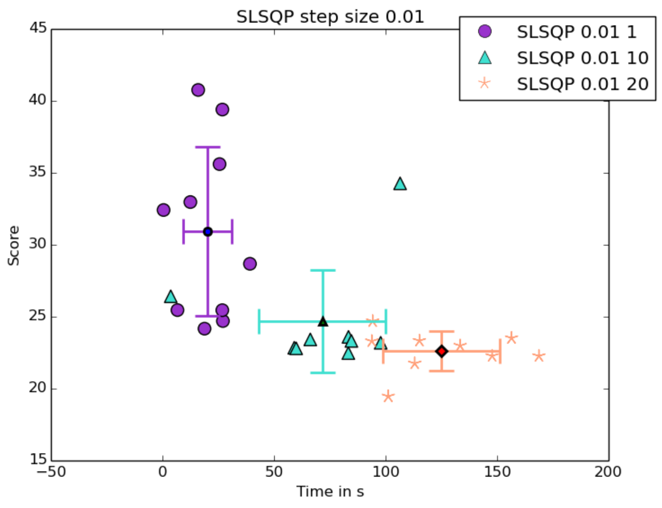


Figure 6.7: The random methods' run time and score

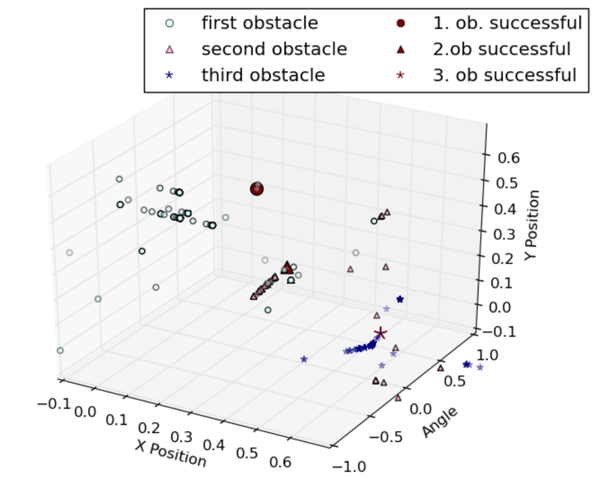


(a) SLSQP eps 1e-8

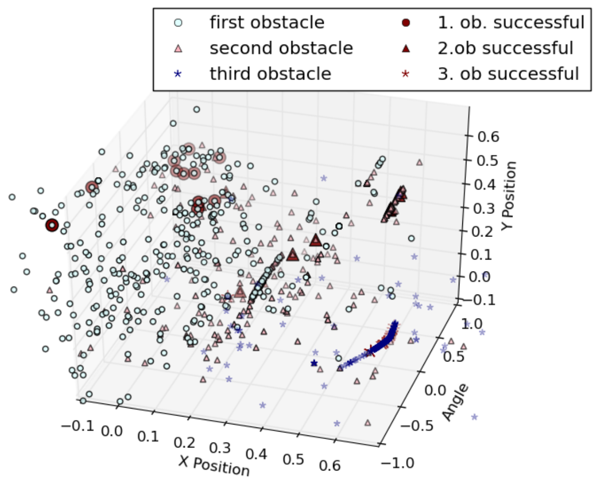


(b) SlSQP eps 0.1

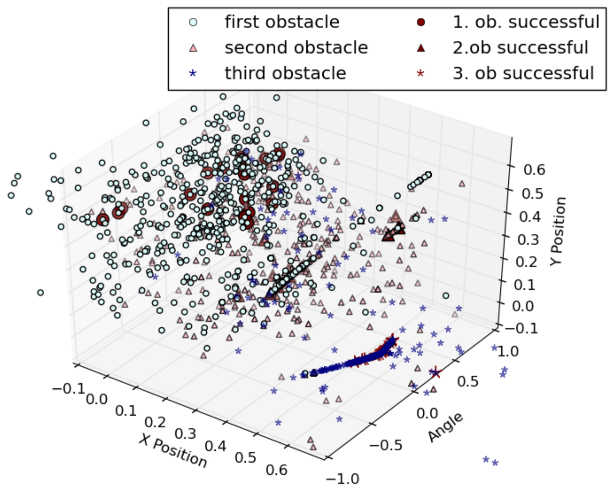
Figure 6.8: SLSQP run time and score of best output



(a) Sampling example for finding one valid solution

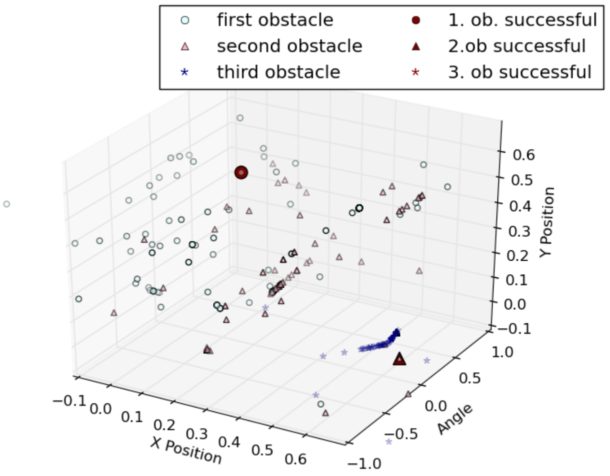


(b) Sampling example for finding ten valid solutions

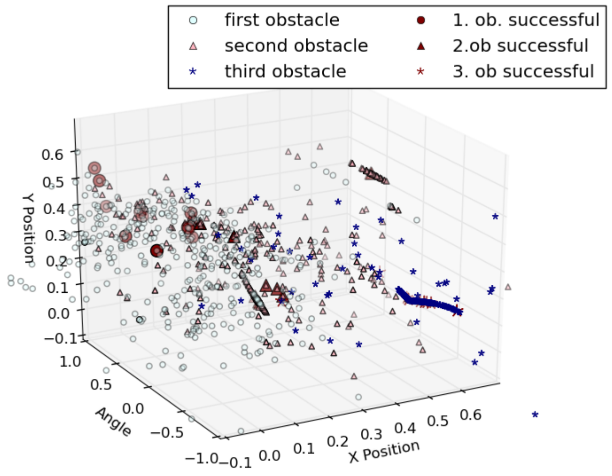


(c) Sampling example for finding twenty valid solutions

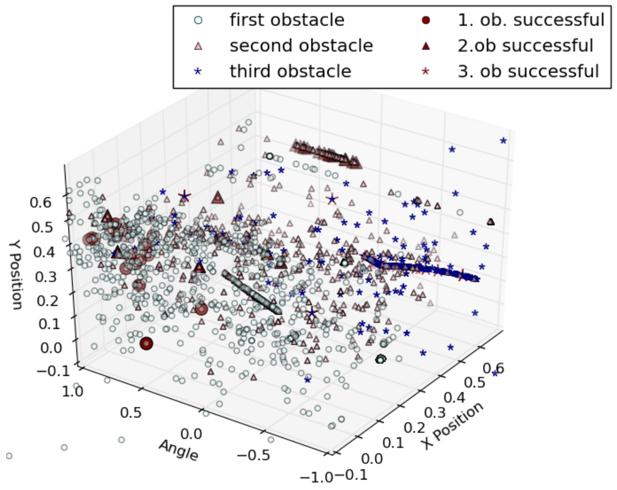
Figure 6.9: Sampling examples of CG with step size 0.1



(a) Sampling example for finding one valid solution

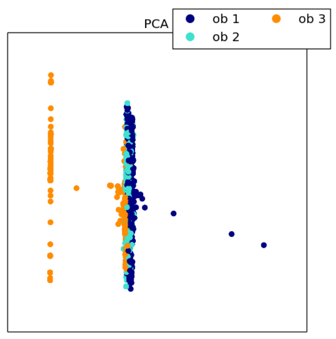


(b) Sampling example for finding ten valid solutions

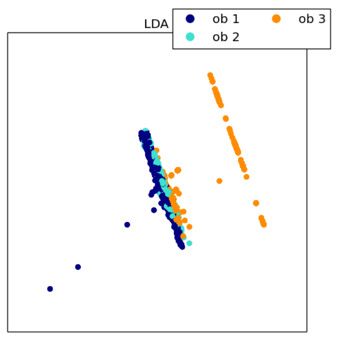


(c) Sampling example for finding twenty valid solutions

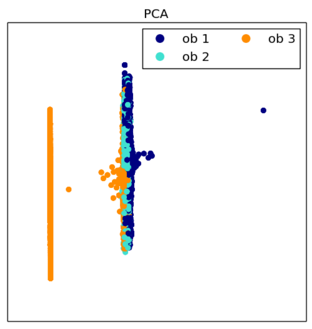
Figure 6.10: Sampling examples of CG with step size 0.01



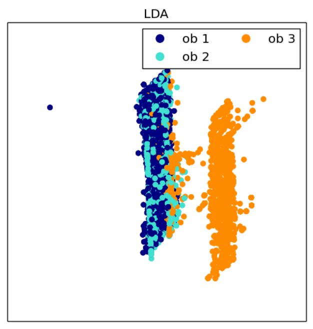
(a) PCA on space information of obstacles for finding one valid solution



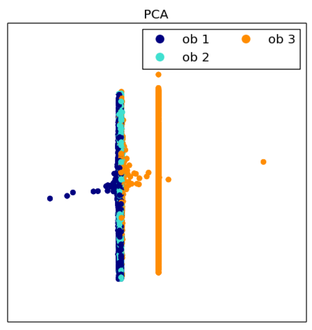
(b) LDA on space information of obstacles for finding one valid solution



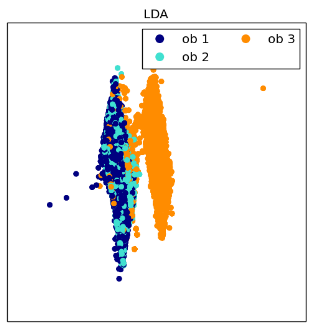
(c) PCA on space information of obstacles for finding ten valid solutions



(d) LDA on space information of obstacles for finding ten valid solutions

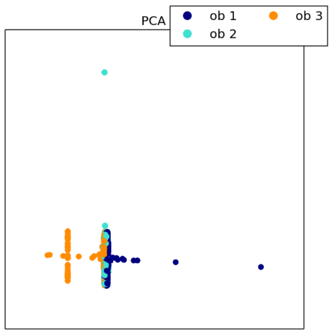


(e) PCA on space information of obstacles for finding twenty valid solutions

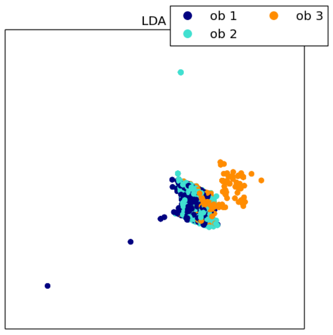


(f) LDA on space information of obstacles for finding twenty valid solutions

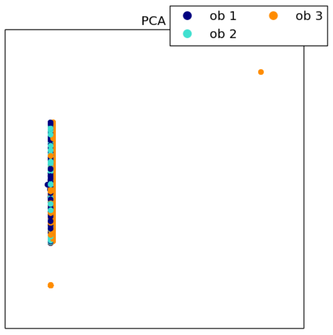
Figure 6.11: Reduced plots of obstacle space information using CG with step size 0.1



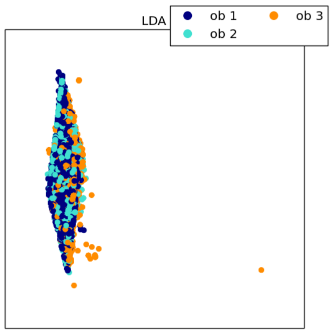
(a) PCA on space information of obstacles for finding one valid solution



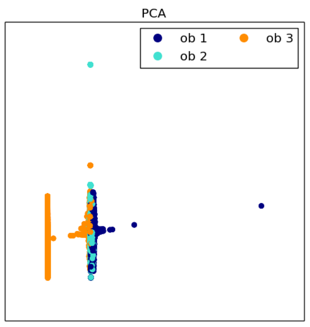
(b) LDA on space information of obstacles for finding one valid solution



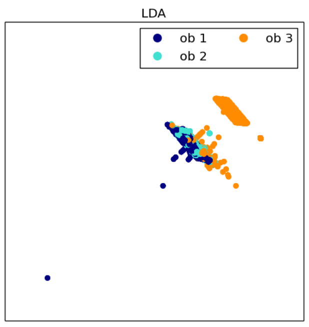
(c) PCA on space information of obstacles for finding ten valid solutions



(d) LDA on space information of obstacles for finding ten valid solutions

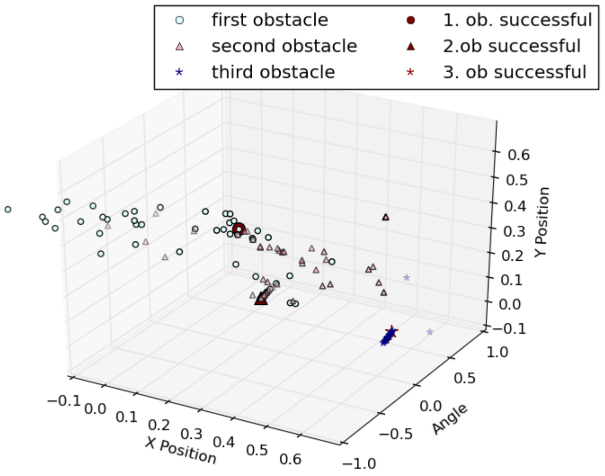


(e) PCA on space information of obstacles for finding twenty valid solutions

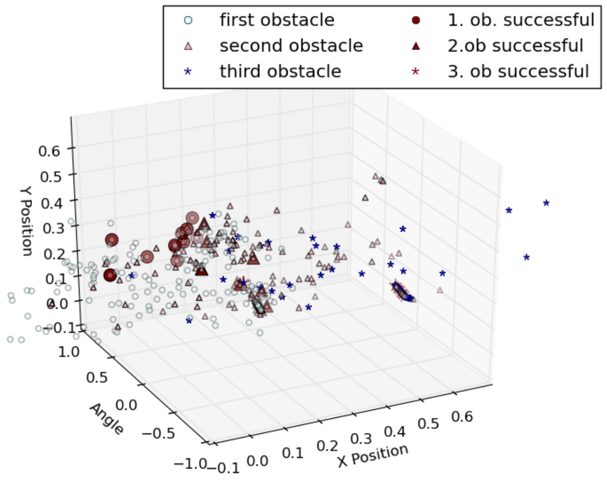


(f) LDA on space information of obstacles for finding twenty valid solutions

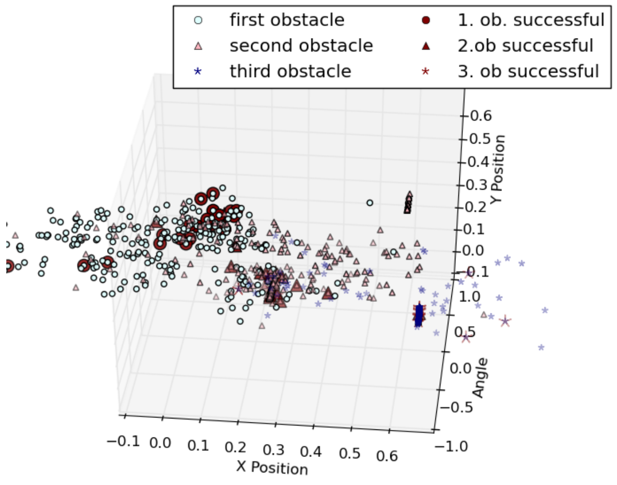
Figure 6.12: Reduced plots of obstacle space information using CG with step size 0.01



(a) Sampling example for finding one valid solution

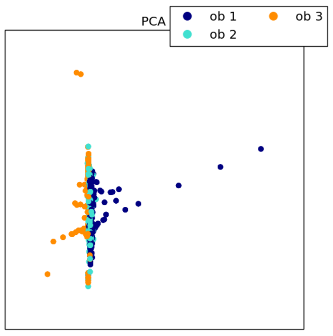


(b) Sampling example for finding ten valid solutions

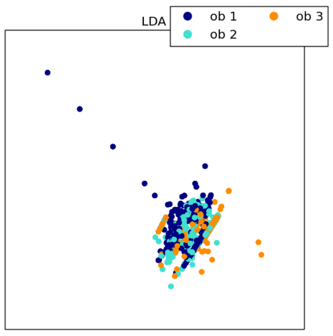


(c) Sampling example for finding twenty valid solutions

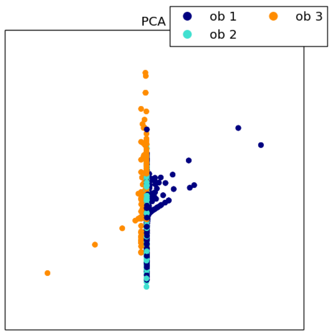
Figure 6.13: Sampling examples of CMA-ES with fixed initialization for all iterations



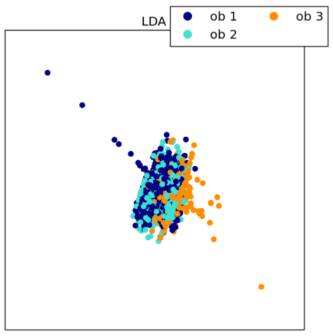
(a) PCA on space information of obstacles for finding one valid solution



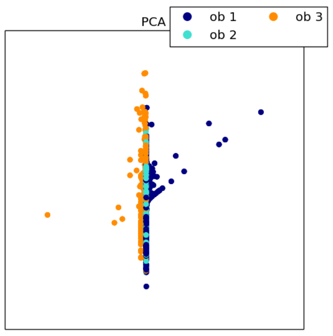
(b) LDA on space information of obstacles for finding one valid solution



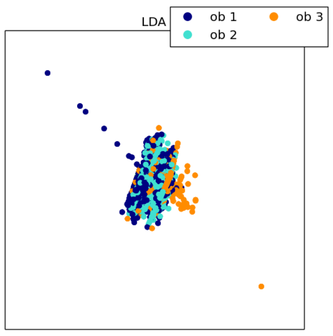
(c) PCA on space information of obstacles for finding ten valid solutions



(d) LDA on space information of obstacles for finding ten valid solutions

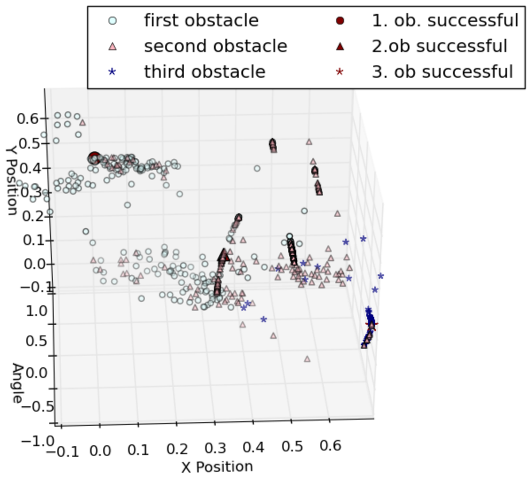


(e) PCA on space information of obstacles for finding twenty valid solutions

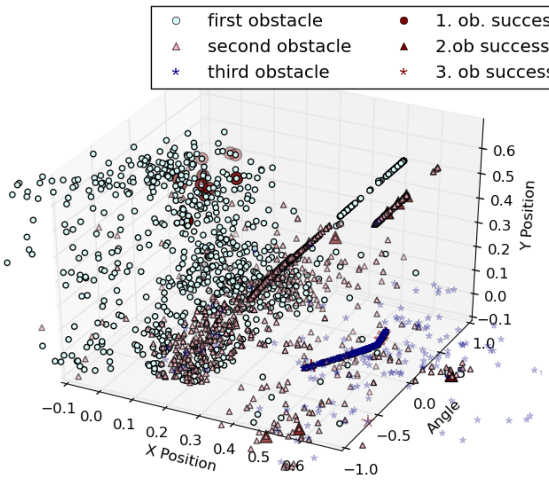


(f) LDA on space information of obstacles for finding twenty valid solutions

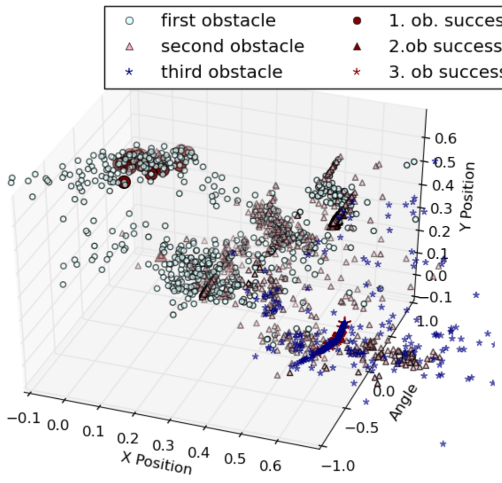
Figure 6.14: Reduced plots of obstacle space information using CMA-ES with fixed initialization for all iterations



(a) Sampling example for finding one valid solution

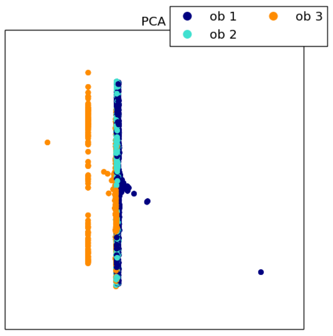


(b) Sampling example for finding ten valid solutions

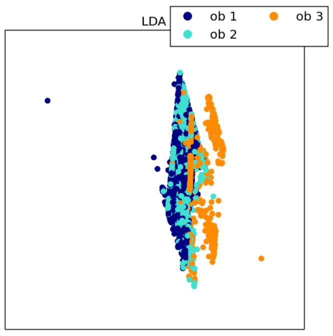


(c) Sampling example for finding twenty valid solutions

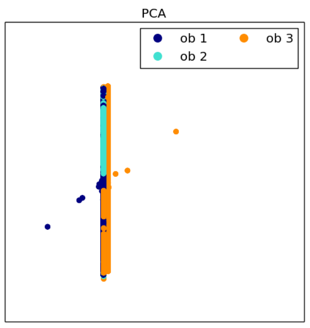
Figure 6.15: Sampling examples of CMA-ES with random initialization for each iteration



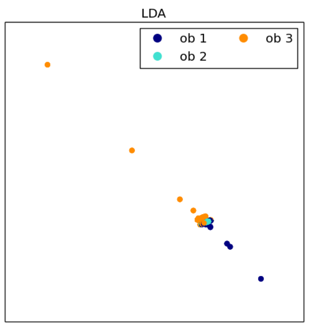
(a) PCA on space information of obstacles for finding one valid solution



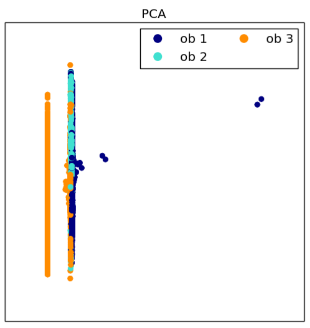
(b) LDA on space information of obstacles for finding one valid solution



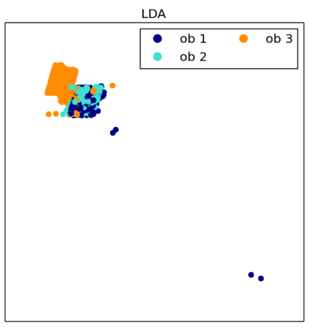
(c) PCA on space information of obstacles for finding ten valid solutions



(d) LDA on space information of obstacles for finding ten valid solutions

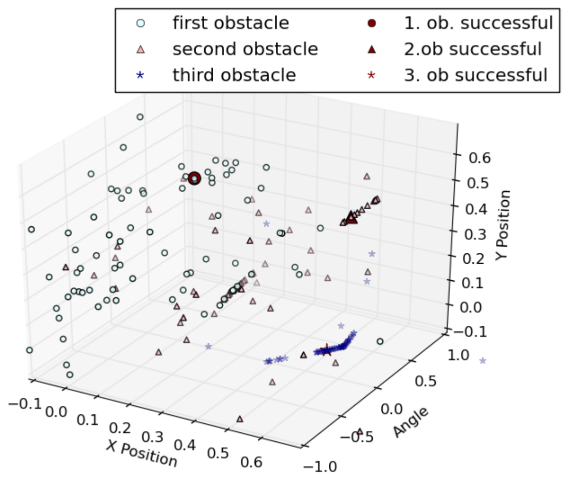


(e) PCA on space information of obstacles for finding twenty valid solutions

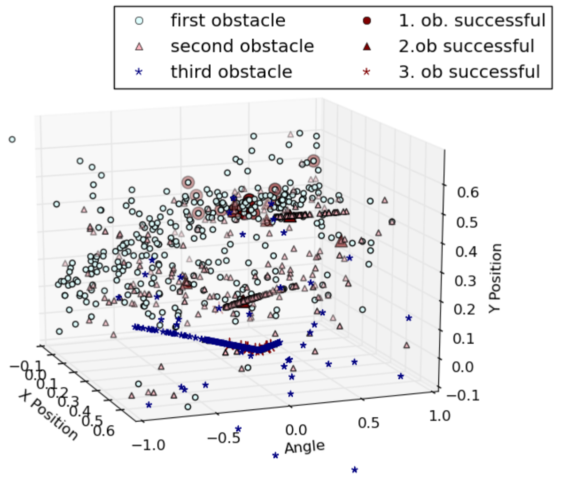


(f) LDA on space information of obstacles for finding twenty valid solutions

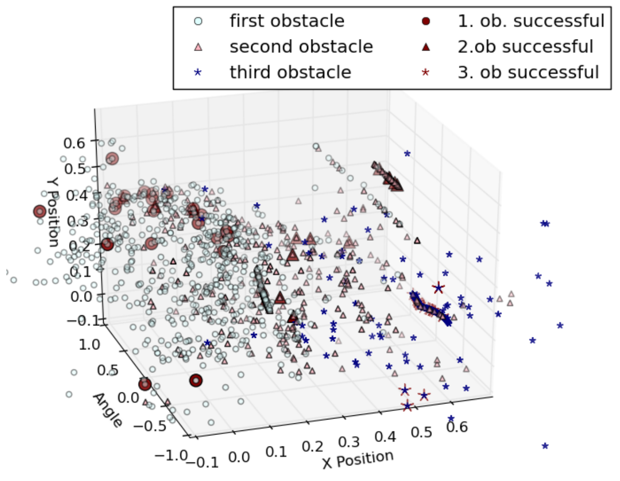
Figure 6.16: Reduced plots of obstacle space information using CMA-ES with random initialization for each iteration



(a) Sampling example for finding one valid solution

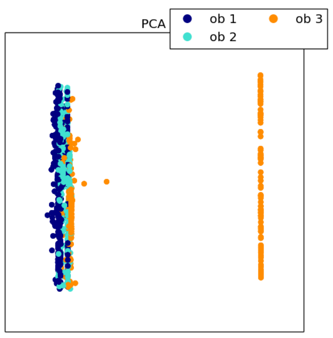


(b) Sampling example for finding ten valid solutions

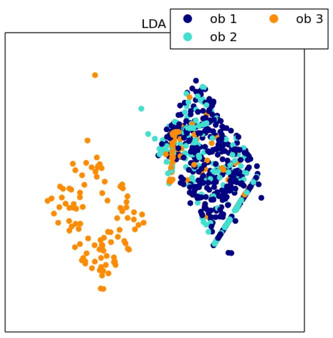


(c) Sampling example for finding twenty valid solutions

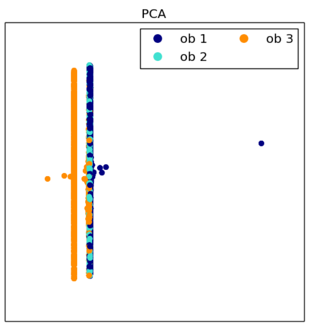
Figure 6.17: Sampling examples of DE



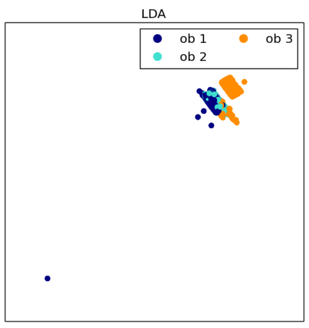
(a) PCA on space information of obstacles for finding one valid solution



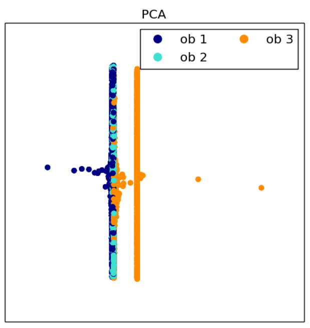
(b) LDA on space information of obstacles for finding one valid solution



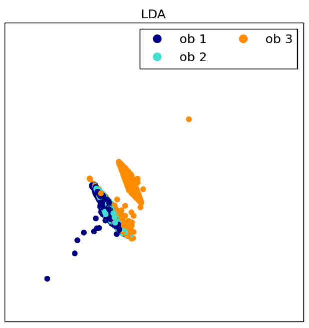
(c) PCA on space information of obstacles for finding ten valid solutions



(d) LDA on space information of obstacles for finding ten valid solutions

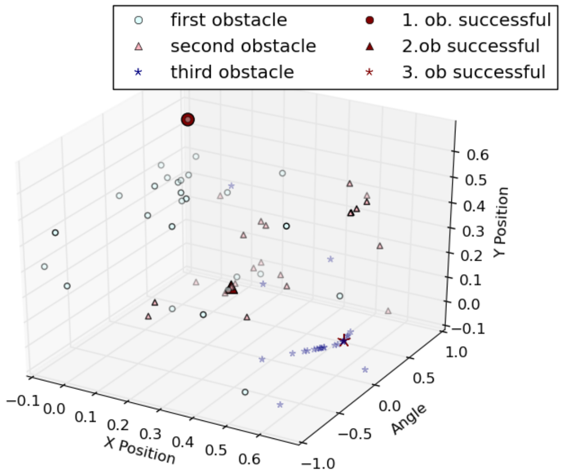


(e) PCA on space information of obstacles for finding twenty valid solutions

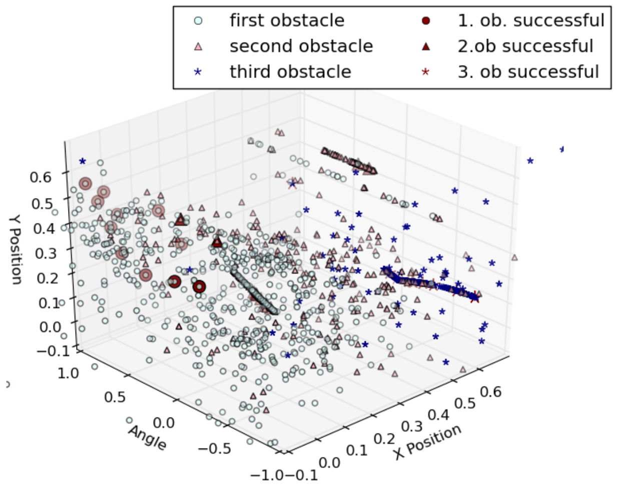


(f) LDA on space information of obstacles for finding twenty valid solutions

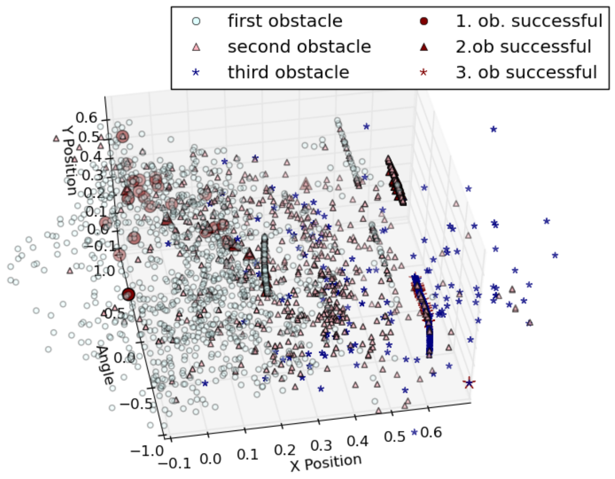
Figure 6.18: Reduced plots of obstacle space information using DE



(a) Sampling example for finding one valid solution

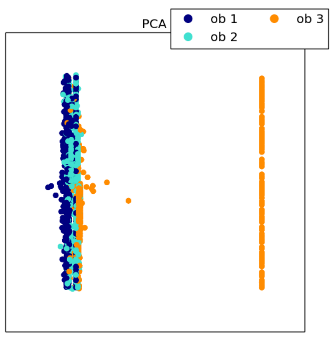


(b) Sampling example for finding ten valid solutions

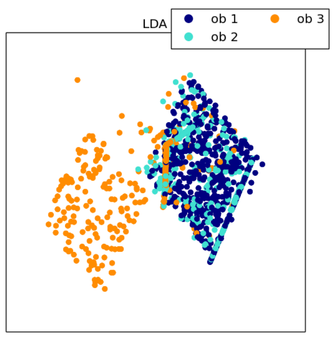


(c) Sampling example for finding twenty valid solutions

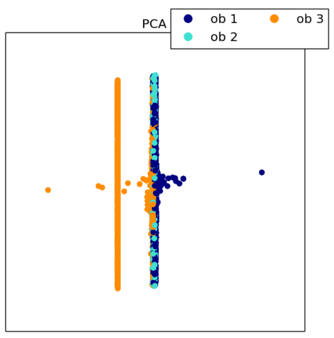
Figure 6.19: Sampling examples of RS



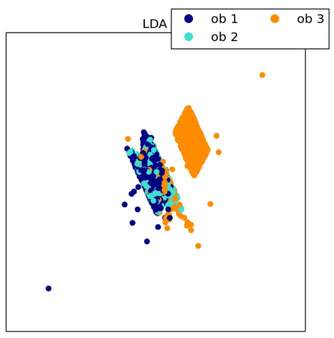
(a) PCA on space information of obstacles for finding one valid solution



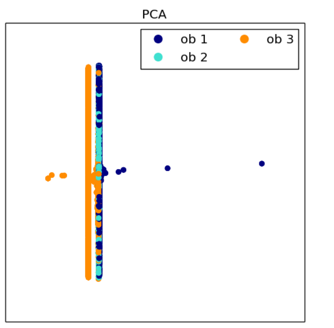
(b) LDA on space information of obstacles for finding one valid solution



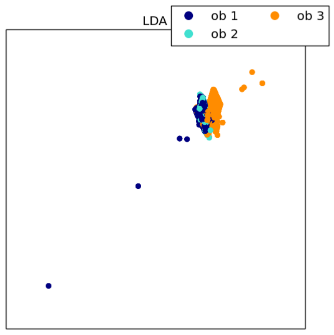
(c) PCA on space information of obstacles for finding ten valid solutions



(d) LDA on space information of obstacles for finding ten valid solutions

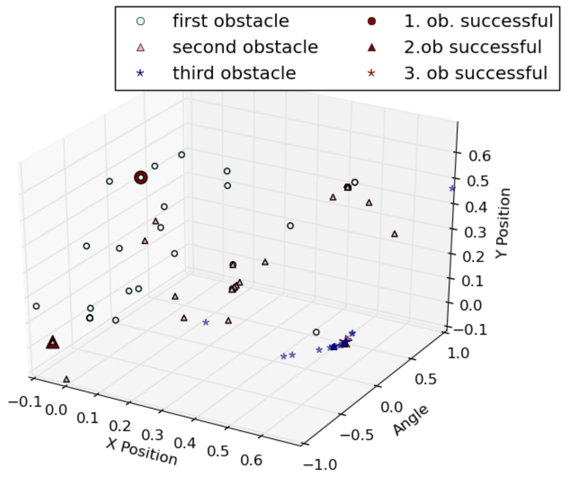


(e) PCA on space information of obstacles for finding twenty valid solutions

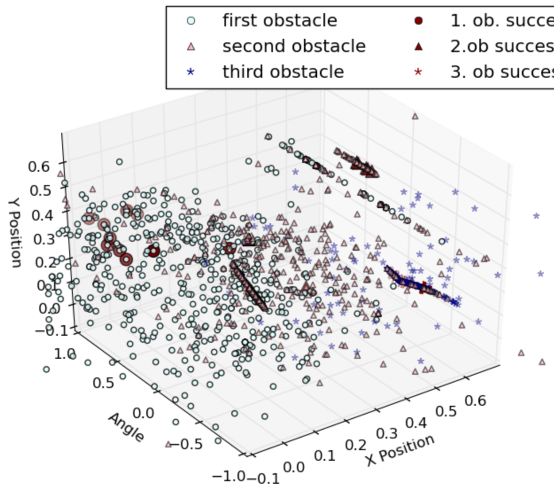


(f) LDA on space information of obstacles for finding twenty valid solutions

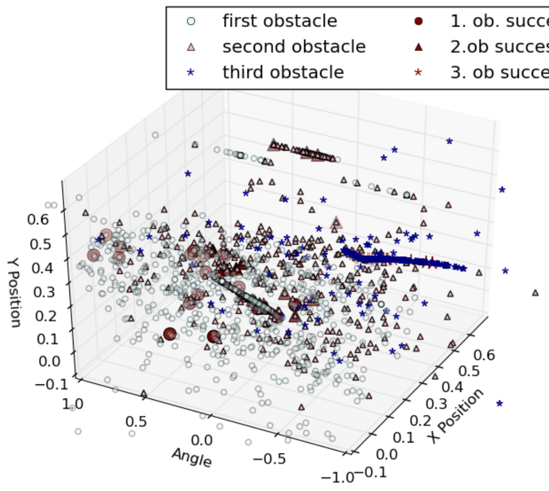
Figure 6.20: Reduced plots of obstacle space information using RS



(a) Sampling example for finding one valid solution

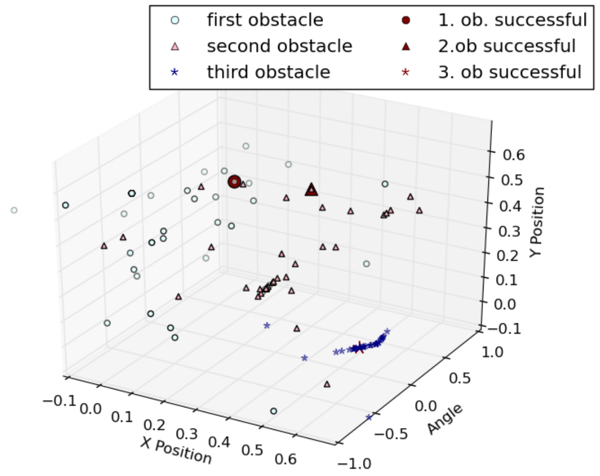


(b) Sampling example for finding ten valid solutions

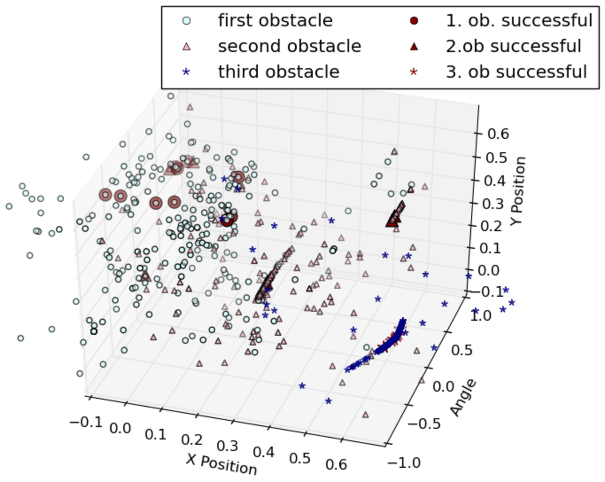


(c) Sampling example for finding twenty valid solutions

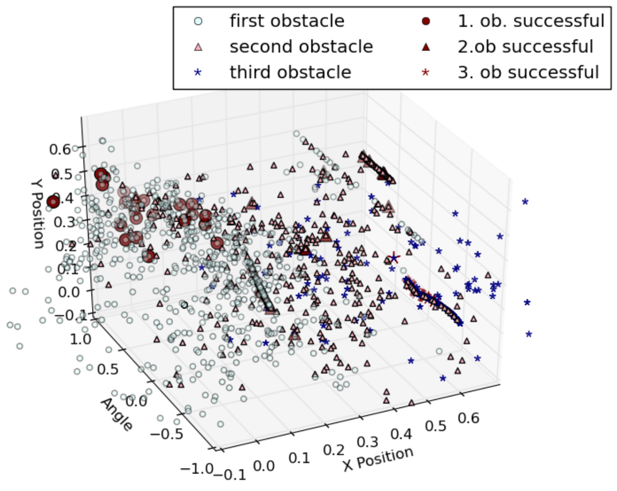
Figure 6.21: Sampling examples of SLSQP with step size 0.01



(a) Sampling example for finding one valid solution

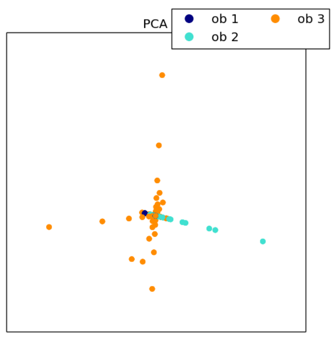


(b) Sampling example for finding ten valid solutions

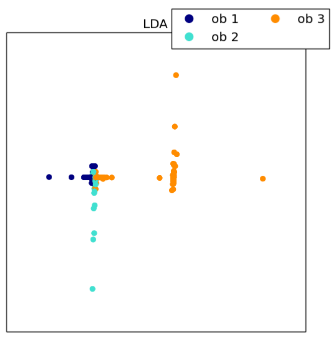


(c) Sampling example for finding twenty valid solutions

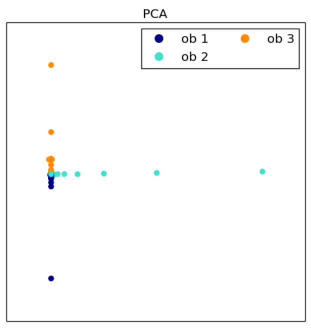
Figure 6.22: Sampling examples of SLSQP with step size 1e-8



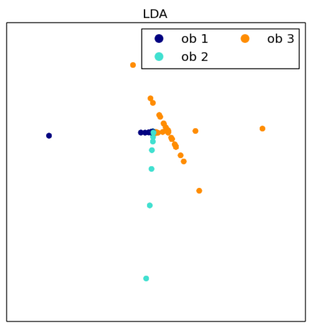
(a) PCA on space information of obstacles for finding one valid solution



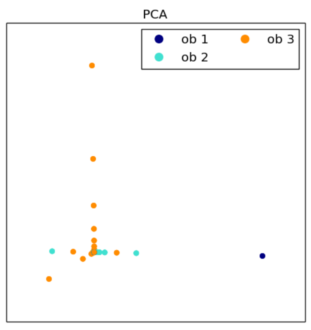
(b) LDA on space information of obstacles for finding one valid solution



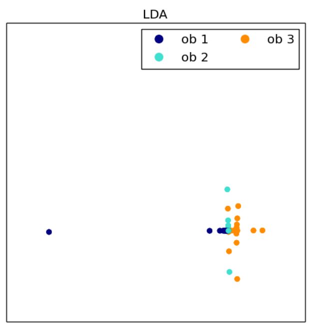
(c) PCA on space information of obstacles for finding ten valid solutions



(d) LDA on space information of obstacles for finding ten valid solutions

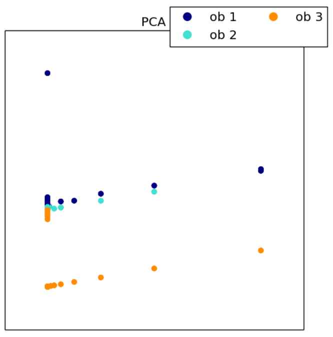


(e) PCA on space information of obstacles for finding twenty valid solutions

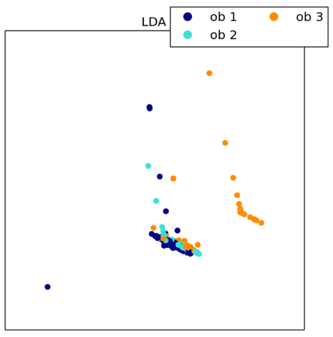


(f) LDA on space information of obstacles for finding twenty valid solutions

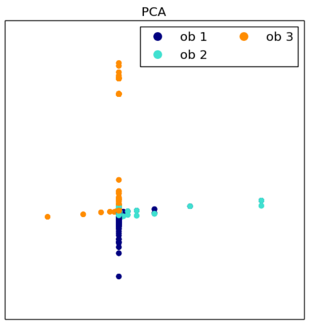
Figure 6.23: Reduced plots of obstacle space information using SLSQP with step size 0.01



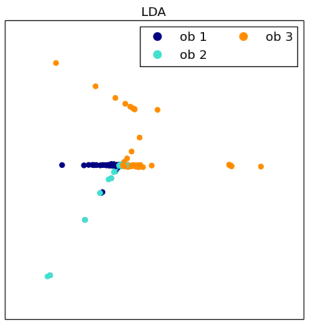
(a) PCA on space information of obstacles for finding one valid solution



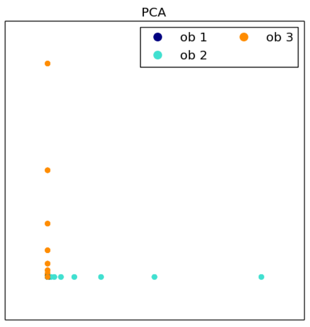
(b) LDA on space information of obstacles for finding one valid solution



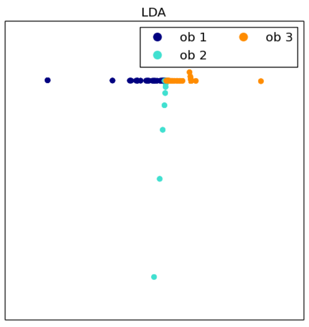
(c) PCA on space information of obstacles of finding ten valid solutions



(d) LDA on space information of obstacles for finding ten valid solutions



(e) PCA on space information of obstacles for finding twenty valid solutions



(f) LDA on space information of obstacles for finding twenty valid solutions

Figure 6.24: Reduced plots of obstacle space information using SLSQP with step size 1e-8

7 Summary and future work

7.1 Summary

We have developed a framework for a robot to play basic marble run. After the robot precepted the positions of the goal object and given fixed obstacles, this space information will be transferred to the simulator and also the optimizer. The optimizer is started, it will pass some solutions to the simulator and use the information provided by the simulator (velocities, end positions, etc.) to find a valid solution. The optimizer will return a layout of a valid solution, which contains positions and angles of the movable obstacles. The robot can place the obstacles directly using the layout.

7.2 Future work

In our approach, a dynamic model of the physical environment is built and passed to the robot. It would be more intelligent if the robot can understand the physical environment by itself, i.e, it can learn the dynamic modell by itself. In other words, the robot would know there exists earth gravity in the real world, and therefore the objects will fall to the ground without applying any outer seen forces on them. Another example would be the robot would notice that if the ball hits an obstacle, its trajectory will change. And eventually the robot would be aware of the effect of the friction, that sliding on different materials may lead to different velocities hence trajectories due to various friction factors.

7.2.1 3D Object Detection

So far our robot can only play with the 2D version of marble run. For a 3-dimensional marble run, the challenging problem of 3D object detection needs to be solved first, as pointed out by Pang and Neumann (2016) and illustrated in Fig. 7.1. In our early phase of this thesis, we used to use the depth camera Kinect V1 to recognize 3D marble run parts.

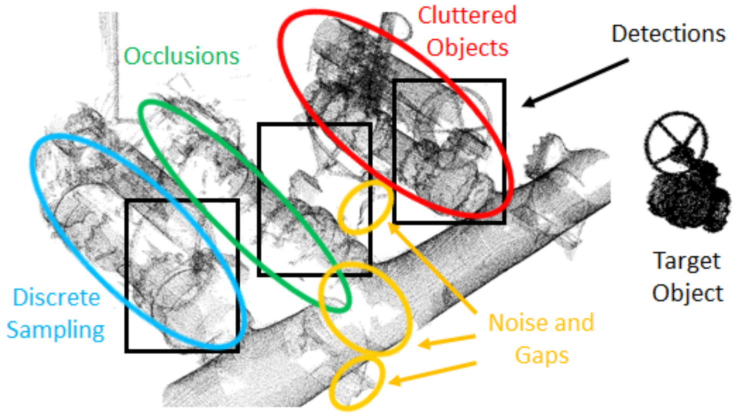


Figure 7.1: Discrete sampling, noisy scans, occlusions and cluttered scenes, the barriers of successful 3D object detection. The figure is from Pang and Neumann (2016)

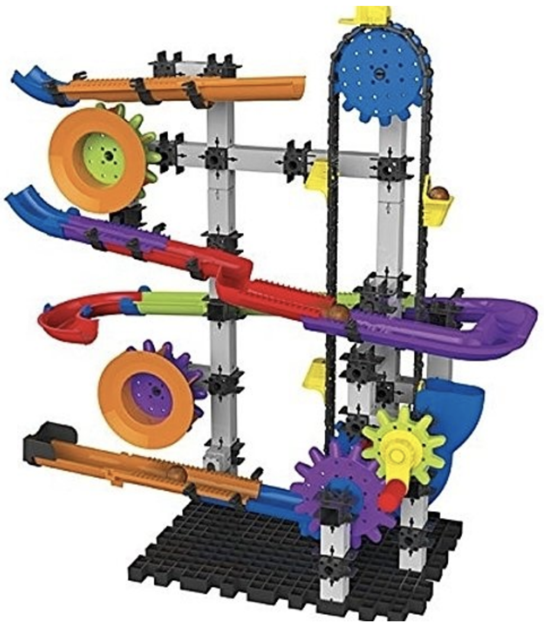


Figure 7.2: An example of a 3D marble run from the brand Marble Mania

Unfortunately, the parts are too small for the existing 3D object-recognition packages, such as Tabletop by Muja and Ciocarlie, or in-hand scanner for small objects by Point Cloud Library (PCL) [1]. We also attempted to use 2D real-time object detection neural network YOLO, developed by Redmon and Farhadi (2018), to detect the objects, but the success rate and the run time is rather unsatisfying on a computer without GPU. On our computer, a four-core Dell i5 with no GPU acceleration, YOLO took more than one second to recognize a single object with a success rate lower than 80%. As a consequence, we came to color detection for object recognition, which works astounishingly perfect in single environment, that is the lighting circumstance has to remain the same.
A further possibility for 3D object-recognition would be projecting the 3D point cloud to 2D images, and then register them into a 3D model, as supposed by Pang and Neumann (2016).

7.2.2 Deep neuronal networks and optimization algorithms

As mentioned before, our model can be extended to learn the dynamic model by itself. Recently, University Stanford released the news that they have built a programm (AI) that can recreates the chemistry's periodic table of elements, see Than (2018). The paper of this research will be first published in July,2018, on Proceedings of the National Academy of Sciences, which exceeds the due of this thesis. According to the announcement, the programm is inspired by the Word2Vec, a language neural network created by Google to parse natural language. The language neural network converts words into numerical vectors, and hence vector operations can be applied. For instance, the word "king" is often followed by the word "queen", and "man" often with "woman". So the word "king" might be roughly seen in a vector manner that "king" is "a queen minus a woman plus a man". Similar idea is applied in the Stanford AI, all the known chemical compounds are passed to the network, such as NaCl, KCl, H20, and so on. Then from

[1] `http://pointclouds.org/documentation/tutorials/in_hand_scanner.php`

these compounds the network figured out that some elements are similar to each other. For example, it has discovered the element potassium (K) and sodium (Na) must have similar properties because both elements can bind with chlorine (Cl).

Back to our problem, this might be an inspiration that physical phenomena can be fed to a neural network, such as "the ball will fall to the ground if there is no obstacle inbetween" or "the ball will stop sliding at some point if the surface is rough". The Stanford AI is trained unsupervised, hence we can start to train our network in an unsupervised manner.

However, knowing the physical phenomena "semantically" is not sufficient for a robot to play the puzzles. The simulated dynamicall model has the exact calculations of gravity and collision. Hence we can acquire velocities and so on from the dynamical model and pass them to the optimizer. Without an exact model, the robot has to estimate the model roughly by itself. One possibility for that is to let the robot play in the real world to gain expierence, as stated in the DeepMind paper by Martin Riedmiller and Springenberg (2018), where a robot arm learned complex behaviors from scratch by directly trying to finish the given task, such as moving an object into a box. In our scenario, the robot needs not only to know that "hitting on an obstacle the ball will change its trajectory" but also "If the obstacle is inclined facing the goal object, the ball will thus fly towards the goal objects". Moreover, with different angles of the inclined obstacles the ball will have different trajectories, though the placement positions remain the same.

Humans play such physical games also by adjusting the angles or placements of the obstacles through vision feedback. To archieve this, the robot also needs the ability of online object recognition, which is not yet completely solved in our system. Additionally, training the robot to play the game is more complex than the merely moving objects, the rewards could be even more sparse, which increases the difficulty for applying reinforement learning. Considering the success of AlphaZero, which is trained without any human input as mentioned in the chapter Related Work, it is possible to train a network from scratch to play physical puzzles. Overall, this particular area which employs artificial intelligence for solving physical puzzles, allows for more thorough research in future works.

Bibliography

Abdelhameed Ibrahim Alaa Tharwat, Tarek Gaber and Aboul Ella Hassanien. Linear discriminant analysis: A detailed tutorial. *Ai Communications*, 2017. 18

Anne Auger Alexandre Chotard and Nikolaus Hansen. Cumulative step-size adaptation on linear functions: Technical report. *arXiv*, 2012. 12

Jeremiah Liu Artur Filipowicz and Alain Kornhauser. Learning to recognize distance to stop signs using the virtual world of grand theft auto 5. *Transportation Research Record*, 2017. 2

Chris Atkeson. Dynamic optimization. University Lecture, 2018. 3, 19

BBC. Fourteen-year-old creates top apple game app. URL `http://www.bbc.co.uk/news/technology-12241564`. 2

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. 2004. 15

N. A. Campbell and William R. Atchley. The geometry of canonical variate analysis. *Systematic Zoology*, 1982. 15, 16

Laurent Cozic. 2d polygon collision detection, 2006. URL `https://www.codeproject.com/Articles/15573/2D-Polygon-Collision-Detection`. 27

Karen Simonyan Ioannis Antonoglou Aja Huang Arthur Guez Thomas Hubert Lucas baker Matthew Lai Adrian bolton Yutian Chen Timothy Lillicrap Fan Hui Laurent Sifre George van den Driessche Thore Graepel David Silver, Julian Schrittwieser and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 2017. 21

Chris J. Maddison1 Arthur Guez1 Laurent Sifre1 George van den Driessche1 Julian Schrittwieser David Silver1, Aja Huang1, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Dominik Grewe Sander Dieleman, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016. 2, 5

Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3338–3346. Curran Associates, Inc., 2014. URL `http://papers.nips.cc/paper/5421-deep-learning-for-real-time-atari-game-play-using-offline-monte-carlo-tree-se pdf`. 2

Laura Diane Hamilton. Introduction to principal component analysis (pca), 2014. URL `http://www.lauradhamilton.com/introduction-to-principal-component-analysis-pca`. 17

Nikolaus Hansen. *The CMA Evolution Strategy: A Comparing Review*. Springer, 2007. 10, 11, 12

Nikolaus Hansen. The cma evolution strategy: A tutorial. *arXiv*, 2016. 10

Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 1952. 6

Dieter Kraft. A software package for sequential quadratic programming, 1988. 8, 50

Thomas Lampe Michael Neunert Jonas Degrave Tom Van de Wiele Volodymyr Mnih Nicolas Heess Martin Riedmiller, Roland Hafner and Tobias Springenberg. Learning by playing – solving sparse reward tasks from scratch. *arXiv*, 2018. 22, 78

Cade Metz. Paul allen wants to teach machines common sense, 2018. 2

Volodymyr Mnih, Koray Kavukcuogl, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, and Daan Wierstra. Playing atari with deep reinforcement learning. *arXiv*, 2013. 2, 5

Marius Muja and Matei Ciocarlie. Tabletop object detector. URL `http://wiki.ros.org/tabletop_object_detector`. 77

G. Pang and U. Neumann. 3d point cloud object detection with multi-view convolutional neural network. pages 585–590, 2016. doi: 0.1109/ICPR.2016.7899697. 76, 77

Frank Hutter Patryk Chrabaszcz, Ilya Loshchilov. Back to basics: Benchmarking canonical evolution strategies for playing atari, February 2018. 5

Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018. 77

Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *arXiv*, 2015. 5

Wilson Robinson. *A simplicial algorithm for concave programming*. PhD thesis, Graduate School of Business Adminstration, Havard University, 1963. 8

Jürgen Schmidhuber and Jieyu Zhao. Direct policy search and uncertain policy evaluation. *AAAI Spring Symposium on Search under Uncertain and Incomplete Information Stanford Univ*, page 119–124, 1998. 5

Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994. 6

Russell Smith. Open dynamics engine, v0.5 user guide, 2006. URL `http://ode.org/ode-latest-userguide.html`. 19

Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 1997. 13, 14

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O.Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv*, 2018. 5

Ker Than. Stanford ai recreates chemistry's periodic table of elements, 2018. URL `https://news.stanford.edu/2018/06/25/ai-recreates-chemistrys-periodic-table-elements/`. 77

Wikipedia. Cma-es, 2018. URL `https://en.wikipedia.org/wiki/CMA-ES`. 13

Zelda B. Zabinsky. Random search algorithms. University Lecture, 2009. 13