A motor control model

Thomas Beucher

June 8, 2015

Chapter 1

Introduction

This report is about my internship at ISIR covering a period of five months from February to July 2015.

The Institute for Intelligent Systems and Robotics (ISIR) is a multidisciplinary research laboratory that brings together researchers and academics from different disciplines of Engineering Sciences and Information and the Life Sciences. It is based in Paris France.

1.1 State of the art

There has been a recent progress in motor control research on understanding how the time of a reaching movement is chosen. In particular, two recent models from Shadmehr et al. [**?**] and Rigoux&Guigon [**?**] proposed an optimization criterion that involves a trade-off between the muscular effort and the subjective value of getting the reward, hence a cost-benefit trade-off (CBT). On one hand, reaching a target faster requires a larger muscular effort (refs?). On the other hand, the subjective value of reaching a target decreases as the time needed to reach the target is increased (refs?). As a result, the net expected return consisting of the subjective value minus the muscular effort is optimal for a certain time, as illustrated in Fig. 1.1(A).

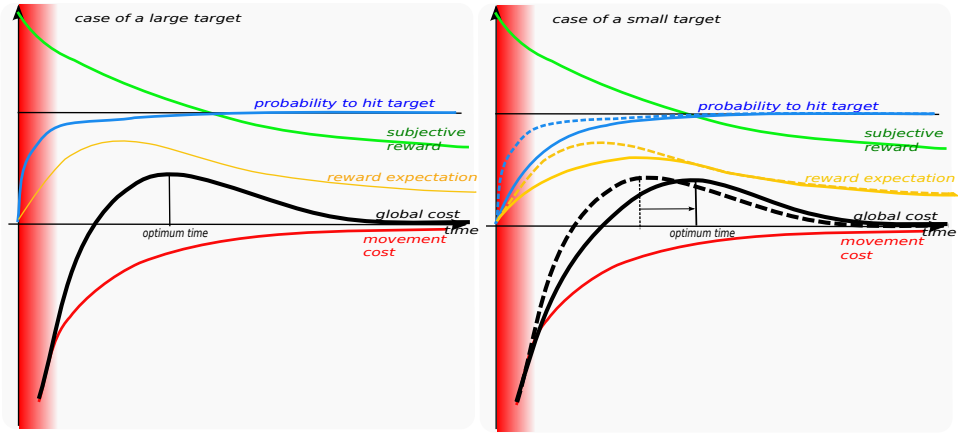


Figure 1.1: Influence of movement time on cost related quantities. Green: subjective utility of hitting the target; red: muscular energy cost; black: global cost versus reward trade-off. The red area denotes infeasible short times; blue: probability to hit the target; orange:reward expectation (subjective reward times probability). A: Sketch of the models in [**?**] and [**?**]. The subjective utility of hitting the reward decreases over time as one is less interested in gains that will occur in a distant future than at the present time. Hitting is less and less costly in terms of efforts as the movement is performed more slowly. The expected gain, resulting from the sum of the subjective reward and the (negative) cost reaches a maximum for a certain time. When the gain is negative (outside the useful interval), one should not move. B: Sketch of the presented model. In the case of a larger target, the hitting probability is higher for faster movements (solid lines) than for a smaller target (dashed lines). As a result, the maximum of the reward expectation is shifted towards longer time for smaller targets, and the optimum movement time is also longer for smaller targets.

However, these models do not account directly for basic facts about the

relation between movement difficulty and movement duration as captured more than fifty years ago by Fitts' law [**?**]. According to this law, the smaller a target, the slower the reaching movement. This is well explained by the so-called *speed-accuracy trade-off* (SAT) stating that, the faster a movement, the less accurate it is, hence the higher the probability to miss the target. So a subject reaching too fast may not get the subjective value associated to reaching and should slow down.

In contrast with the models of [**?**] and [**?**], the model of Dean [**?**] takes the SAT into account. The key difference with respect to [**?**] and [**?**] is that, instead of maximizing a reward, this model maximizes a *reward expectation*, i.e. the reward times the probability to get it.

However, the model proposed in [**?**] is an abstract model of movement time selection that looks for an optimal trade-off between an externally decayed reward and a SAT that relates the probability of missing to movement time. As such, it does not account for movement execution, neither for the choice of a motor trajectory and its impact on the cost of movement. The model does not explain Fitts' law, it rather incorporates its consequences into an abstract model of the SAT that is fitted to experimental data. The mathematical design of the model is based on several simplifying assumptions and it predicts optimal movement times that are systematically shorter than those observed with subjects. The authors of [**?**] discuss that this may result from the fact that the model does not take the cost of movement into account.

In the paper of Olivier Sigaud and Kevin Monfray, they show that the models of Shadmehr [**?**] and Rigoux [**?**] as well as the model of Dean [**?**] can be unified into a model that solves the difficulties faced by these previous models.

This unification is simply implemented by including sensory and motor noise into the optimal control model proposed in [**?**], shifting from a deterministic account of the movement to a stochastic one, in line with the models of [**?**, **?**, **?**, **?**, 3].

As a matter of fact, in the models of [**?**] and [**?**], the target is given as a single point and the movement is considered as always reaching it, irrespective of the size of the target. In order to fully account for Fitts' law, one must consider the intrinsic dispersion of reaching movements towards a target and the effect of sensory and muscular noise on this dispersion (e.g. [**?**], see [**?**] for a review), which is not the case of the models of [**?**] and [**?**].

As a result, the reward and muscular activation terms in the optimization criterion proposed in [**?**] are replaced by reward and cost expectation terms. Considering expectation is a way to account for the fact that, in case of a miss, one would not get the reward, so the global outcome of the movement would only consist of its incurred cost.

1.2 Previous work

I used a code, written by Kevin Monfray (a trainee at ISIR), which implement a model based on optimal control called NOPS but it is very time consuming. To reduce the computation time, Olivier Sigaud decided to try to learn the NOPS controller using a regression algorithm. So I generated trajectories using the NOPS controller then I used the regression algorithm called RBFN(Radial Basis Function Networks) to learn the new controller from these trajectories. The new controller could then be optimized with a stochastic optimization method called CMAES(Covariance Matrix Adaptation Evolution Strategy).

1.3 My job

First, I have to implement the model of arm and the regression algorithm to learn the new controller from the trajectories generated with the NOPS controller (implemented by Didier Marin in C++ and in Java by Kevin Monfray). Secondly, I must use the CMAES algorithm to optimize RBFN controller for different sizes of target. Finally, generate various relevant curves and compare them with the curves obtained by Slovenian colleagues, Jan babic and Luka Peternel, who experiment on humans around the same issue.

Chapter 2

Material and methods

2.1 Arm model

The plant is a two degrees-of-freedom (dofs) planar arm controlled by 6 muscles, illustrated in Fig. 2.1. There are several such models in the literature. The model described in [1] lies in the vertical plane so it takes the gravity force into account. Most other models are defined in the saggital plane and ignore gravity effects. They all combine a simple two dofs planar rigid-body dynamics model with a muscular actuation model. The differences between models mostly lie in the latter component.

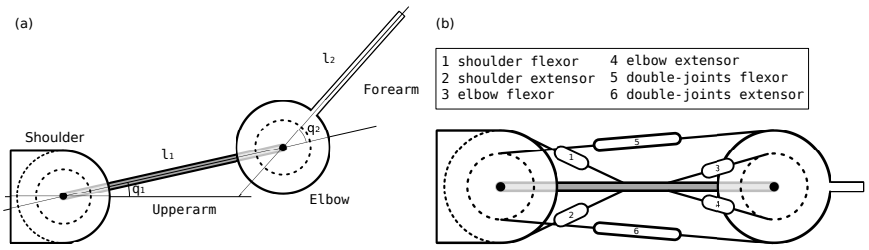


Figure 2.1: Arm model. (a) Schematic view of the arm mechanics. (b) Schematic view of the muscular actuation of the arm, where each number represents a muscle whose name is in the box.

Table 4.1 in Appendix 4.1 reminds the nomenclature of all the parameters and variables of the arm model.

2.1.1 Arm parameters

All parameters of the arm are defined in the file *setupArmParameters* and implemented in the class *ArmParameters*. This class defines the following functions:

readSetupFile : Reads the setup file.

massMatrix : Defines the inertia matrix parameters.

BMatrix : Defines the damping matrix $\mathbf{B}$, with $\mathbf{B} = \begin{bmatrix} .05 & .025 \\ .025 & .05 \end{bmatrix} \dot{q}$.

AMatrix : Defines the moment arm matrix A.

$$\begin{aligned} \mathbf{A}^\top &= \begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 & a_{10} & a_{11} & a_{12} \end{bmatrix} \\ &= \begin{bmatrix} .04 & -.04 & 0 & 0 & .028 & -.035 \\ 0 & 0 & .025 & -.025 & .028 & -0.35 \end{bmatrix} \end{aligned}$$

All the arm parameters values are summarized in Table 4.2 in Appendix 4.1.

2.1.2 Muscles parameters

All muscles parameters are defined in the file *setupMusclesParameters* and implemented in the class *MusclesParameters*. This class defines the following functions:

fmaxMatrix : Defines the matrix of the maximum force exerted by each muscle.

$$\begin{aligned} \mathbf{f_{max}} &= \begin{pmatrix} f_{\max 1} & 0 & 0 & 0 & 0 & 0 \\ 0 & f_{\max 2} & 0 & 0 & 0 & 0 \\ 0 & 0 & f_{\max 3} & 0 & 0 & \\ 0 & 0 & 0 & f_{\max 4} & 0 & 0 \\ 0 & 0 & 0 & 0 & f_{\max 5} & 0 \\ 0 & 0 & 0 & 0 & 0 & f_{\max 6} \end{pmatrix} \\ &= \begin{pmatrix} 700 & 0 & 0 & 0 & 0 & 0 \\ 0 & 382 & 0 & 0 & 0 & 0 \\ 0 & 0 & 572 & 0 & 0 & 0 \\ 0 & 0 & 0 & 445 & 0 & 0 \\ 0 & 0 & 0 & 0 & 159 & 0 \\ 0 & 0 & 0 & 0 & 0 & 318 \end{pmatrix} \end{aligned}$$

activationVectorInit : Initializes the muscular activation vector. (Create the vector and initializes to zero)

activationVectorUse : Builds the muscular activation vector given its 6 components.

All the muscles parameters values are summarized in Table 3 in Appendix 4.1.

2.1.3 Rigid-body dynamics

The rigid-body dynamics equation of a mechanical system is:

$$\ddot{\boldsymbol{q}} = \boldsymbol{M}(\boldsymbol{q})^{-1}(\boldsymbol{\tau} - \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) - \boldsymbol{g}(\boldsymbol{q}) - \boldsymbol{B}\dot{\boldsymbol{q}}) \tag{2.1}$$

where $\mathbf{q}$ is the current articular position, $\dot{\boldsymbol{q}}$ the current articular speed, $\ddot{\boldsymbol{q}}$ the current articular acceleration, $\mathbf{M}$ the inertia matrix, $\mathbf{C}$ the Coriolis force vector, $\boldsymbol{\tau}$ the segments torque, $\mathbf{g}$ the gravity force vector and $\mathbf{B}$ a damping term that contains all unmodelled effects. Here, $\mathbf{g}$ is ignored since the arm is working in the sagittal plane. All angles are expressed in radians. We can compute the inertia matrix as: $\mathbf{M} = \begin{bmatrix} k_1 + 2k_2\cos(q_2) & k_3 + k_2\cos(q_2) \\ k_3 + k_2\cos(q_2) & k_3 \end{bmatrix}$, with $k_1 = d_1 + d_2 + m_2 l_1^2,\ k_2 = m_2 l_1 s_2,\ k_3 = d_2$ where d_i, m_i, l_i and s_i are parameters of the arm previously defined in Section 2.1.1.

The Coriolis force vector is given by

$$\mathbf{C} = \begin{bmatrix} -\dot{q_2}(2\dot{q_1} + \dot{q_2})k_2\sin(q_2) \\ \dot{q_1}^2 k_2 \sin(q_2) \end{bmatrix}.$$

The computation of the torque τ exerted on the system given an input muscular actuation $\mathbf{u}$ is explained in the section 2.1.4.

Equation 2.1 is implemented in the class *ArmDynamics* line 57:

```
57      ddotq = np.dot(Minv,(Gamma - C - np.dot(armP.B,
            dotq)))
```

where *np* refers to the numpy library in python. We also find in this class all elements of Equation 2.1:

```
44      #Inertia matrix
45      M = np.array([[armP.k1+2*armP.k2*math.cos(q[1,0]),
            armP.k3+armP.k2*math.cos(q[1,0])],[armP.k3+armP
            .k2*math.cos(q[1,0]),armP.k3]])
46      #Coriolis force vector
47      C = np.array([[-dotq[1,0]*(2*dotq[0,0]+dotq[1,0])*
            armP.k2*math.sin(q[1,0])],[(dotq[0,0]**2)*armP.
            k2*math.sin(q[1,0])]])
48      #inversion of M
49      Minv = np.linalg.inv(M)
50      #torque term
51      Q = np.diag([q[0,0], q[0,0], q[1,0], q[1,0], q
            [0,0], q[0,0]])
52      #the commented version uses a non null stiffness
            for the muscles
53      #Gamma = np.dot((np.dot(armP.At, musclesP.fmax)-np
            .dot(musclesP.Kraid, Q)), U)
```

```
54        Gamma = np.dot((np.dot(armP.At, musclesP.fmax)-np.
              dot(musclesP.Knulle, Q)), U)
```

2.1.4 Muscular actuation

Our muscular actuation model is taken from [2] (pp. 356-357) through [4]. It is a simplified version of the one described in [3] in the sense that it uses a constant moment arm matrix $\mathbf{A}$ whereas [3] is computing this matrix as a function of the state of the arm.

Finally, given an action $\mathbf{u}$ corresponding to a raw muscular activation as output of the controller, the muscular activation is augmented with Gaussian noise using $\tilde{\mathbf{u}} = \log(\exp(\kappa \times \mathbf{u}_t \times (1 + \mathcal{N}(0, \mathbf{I}\sigma_u^2))) + 1)/\kappa$, where $\times$ refers to the element-wise multiplication, $\mathbf{I}$ is a 6×6 identity matrix. and $\kappa = 25$ is the Heaviside filter parameter, and the input torque is computed as $\boldsymbol{\tau} = \mathbf{A}^\top(\mathbf{f_{max}} \times \tilde{\mathbf{u}})$.

2.2 Experimental set-up

The state-space consists of the current articular position $\mathbf{q}$ of the arm and its current articular speed $\dot{\boldsymbol{q}}$. The state $\boldsymbol{s} = (\dot{\boldsymbol{q}}, \boldsymbol{q})$ has a total of 4 dimensions. The initial state is defined by null speed and a variable initial position. The positions are bounded to represent the reachable space of a standard human arm, with $q_1 \in$ [-0.6, 2.6] and $q_2 \in$ [-0.2, 3.0], as shown in Figure 2.2. The action-space consists of an activation signal for each muscle, which also makes a total of 6 dimensions.

When using CMAES, reaching the goal point is replaced by hitting the target on the screen. The target is defined as an interval of varying length around (x = 0, y = 0.6175). The movement is stopped once the line y = 0.6175 has been crossed, and the intersect between the trajectory and this line is computed to determine whether the target was hit. The reward for immediately hitting the target without taking incurred costs into account is set to 300.

In order to train RBFN, we generate trajectories with the NOPS from the initial positions shown in Figure 2.2 to the goal point. The starting points have been organized into four groups of different distances with respect to the goal point (d = 18cm, 20cm, 22cm and 24cm respectively). There are 12 initial positions per distance, thus a total of 48 initial positions.

We measure the dispersion over 100 movements towards this target, as well as the average movement time and average movement cost.

We run 60 iterations of CMAES with a population of 100 and a sigma equal to 1.10^{-6} on the RBFN controller for each target.

For all the obtained controllers, we measure again the dispersion over 100 movements, the average movement time and average movement cost.

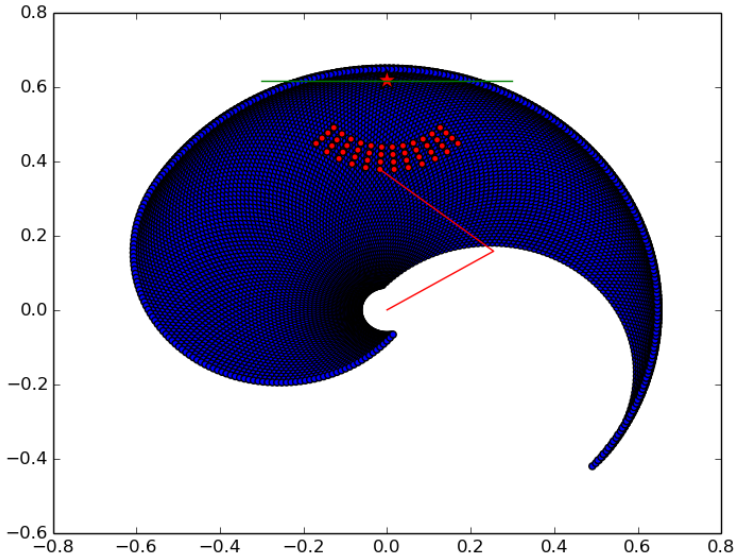


Figure 2.2: The arm workspace. The reachable space is delimited by a spiral-shaped envelope. The two segments of the arm are represented by two red lines. Initial movement positions are represented with red dots organized into four sets of different distances to the target. The screen is represented as a green line positioned at y = 0.6175. The origin of the arm is at x = 0.0, y = 0.0.

Finally, for all these targets, we record the velocity profile.

2.3 Mathematical formulation of the model

The cost function $J(\vec{u})$ proposed for a control $\vec{u}$ in the model of [**?**] is

$$J(\vec{u}) = \int_0^\infty e^{-t/\gamma}[\rho R(\vec{s}_t) - \nu L(\vec{u}_t)]dt \tag{2.2}$$

where $R(\vec{s}_t)$ is the immediate reward function that equals 1 at the goal point (also called rewarded state) and is null everywhere else. The function $L(\vec{u}_t)$ is the movement cost. The authors of [**?**] take $L(\vec{u}_t) = \|\vec{u}_t\|^2$, as in many motor control models. The continuous-time discount factor γ accounts for the "greediness" of the controller, i.e. the smaller γ, the more the agent is focused on short term rewards. Finally, ρ is the weight of the reward term and ν the weight of the effort term. In all experiments presented here, based, on the previous work from [**?**], we took $\gamma = 0.998$, $\rho = 300$ and $\nu = 1$.

A near optimal deterministic policy to solve this problem is obtained through a computationally expensive variation calculus method (see [**?**] for details). Given that the policy does not take the presence of noise in the model of the plant into account, the actions must be computed again at each time step depending on the new state reached by the plant which further contributes to the cost of the method. The controller resulting from this model is called the NOPS (for Near-Optimal Planning System).

Now let us consider the integration of accuracy constraints. Instead of a deterministic controller, the new model is based on a stochastic controller where the rewarded state is reached or not. As a result, the outcome of a large set of movements performed with noise is computed as the value of the reward mutiplied by the probability to obtain it over the different movements. Mathematically, the value multiplied by the probability is called the expectation.

Taking the probability to reach the target into account as described above, the new optimization criterion is written

$$J(\vec{u}) = \int_0^\infty e^{-t/\gamma}\mathbb{E}[\rho R(\vec{s}_t) - \nu L(\vec{u}_t)]dt \tag{2.3}$$

where $\mathbb{E}[]$ stands for the expectation of the cumulated reward, and $R(\vec{s}_t)$ equals 1 if the end effector hits the target.

2.4 Learning method, RBFN controller

2.4.1 Theory

Regression is the process of learning relationships between inputs and continuous outputs from example data, which enables predictions for novel inputs. The history of regression is closely related to the history of artificial neural networks since the seminal work of Rosenblatt (1958). There are many regression algorithms, in our case we use the *Regression with Radial Basis Function Networks* (RBFNs).

We want to approximate the function $f(s) = U$ where s is the state and U the vector of muscular activations. We use a parametric function $\hat{f}_\theta : \mathbb{R} \to \mathbb{R}$ where $\theta \in M_{k,n}(\mathbb{R})$ is a real matrix of parameters. We make the assumption that f is non-linear and can be approximated using:

$$\hat{f}(s) = \Phi(s)^T \boldsymbol{\theta} \tag{2.4}$$

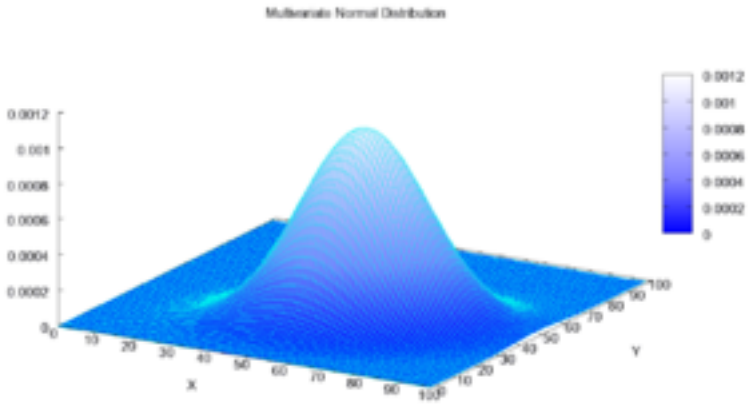
where $\phi(s) = [\phi_1(s), \phi_2(s), \cdots, \phi_k(s)]^T$ is a vector function commonly referred as features.

We use Multivariate Gaussian basis functions throughout the input space. We are in the non-degenerate case i.e. when the symmetric covariance matrix Σ is positive definite. The multivariate normal distribution in our case

could be write as:

$$f_x(s_1, \cdots, s_k) = \frac{1}{\sqrt{(2\pi)^k |\Sigma|}} exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right) \tag{2.5}$$

where x is a real k-dimensional column vector and $|\Sigma|$ is the determinant of the covariance matrix Σ. The descriptive statistic $(x-\mu)^T\Sigma^{-1}(x-\mu)$ in the non-degenerate multivariate normal distribution equation is known as the square of the Mahalanobis distance, which represents the distance of the test point x from the mean μ.



The goal is to determine the parameters θ which permit $\hat{f}$ to best approximate f. Let ∇_θ denote the gradient according to θ, the gradient of the error function is:

$$\begin{aligned}
\nabla_\theta \hat{\epsilon}(\theta) = \nabla_\theta \mathbb{E}\left[\frac{1}{2}(y-\hat{f}(s))^2\right] &\approx \nabla_\theta \frac{1}{Ns}\sum_{i=1}^{Ns}\left[\frac{1}{2}(y-\hat{f}(s))^2\right] \\
&= \frac{1}{Ns}\sum_{i=1}^{Ns}\left[(\nabla_\theta(y-\hat{f}(s)))(y-\hat{f}(s))\right] \\
&= \frac{1}{Ns}\sum_{i=1}^{Ns}\left[(\nabla_\theta \hat{f}(s))(\hat{f}(s)-y)\right] \qquad (2.6)
\end{aligned}$$

For our linear function approximator with Gaussian basis functions, we have

$$\nabla_\theta \hat{f}(s) = \phi(s) \tag{2.7}$$

This gradient cancels out for any local optimum of the error function,

therefore we can find one of these by formulating θ as follows:

$$\begin{aligned}
& \nabla_\theta \hat{\epsilon}(\theta) = 0 \\
\Leftrightarrow \quad & \tfrac{1}{Ns} \sum_{i=1}^{Ns} \phi(s_i)(\phi(s_i)^T\theta - y_i) \approx 0 \\
\Leftrightarrow \quad & \sum_{i=1}^{Ns} (\phi(s_i)\phi(s_i)^T)\theta \approx \sum_{i=1}^{Ns} (\phi(s_i)y_i) \\
\Leftrightarrow \quad & A\theta \approx b \\
\Leftrightarrow \quad & \theta \approx A^{\#}b
\end{aligned}$$

with,

$$A = \sum_{i=1}^{Ns} (\phi(s_i)\phi(s_i)^T) \tag{2.8}$$

$$b = \sum_{i=1}^{Ns} (\phi(s_i)y_i) \tag{2.9}$$

where $A^{\#}$ is the pseudo-inverse of matrix A.

2.4.2 Implementation

We have a set of Ns samples (the data of trajectories generated by the Monfray's controller given the state and the muscles activations at each time for each trajectories). We feed the algorithm with this data through the function *setTrainingData* which check the input dimension, the output dimension and if there is the same number of samples for input and output.

Then we set the centers and widths for the gaussian used by calling the function *setCentersAndWidths*.

Finally, we run the function *train_rbfn* to create the controller. We find is this function:

- the computation of A line 46:

```
46          At = np.dot(fop, fop.T)
```

  where $fop = \phi(s)$.

- the computation of b line 52:

```
52          bt = np.dot(fop, self.outputData.T)
```

- the computation of θ line 80:

```
80          self.theta = np.dot(np.linalg.pinv(A), b)
```

  where np.linalg.pinv(A) = $A^{\#}$.

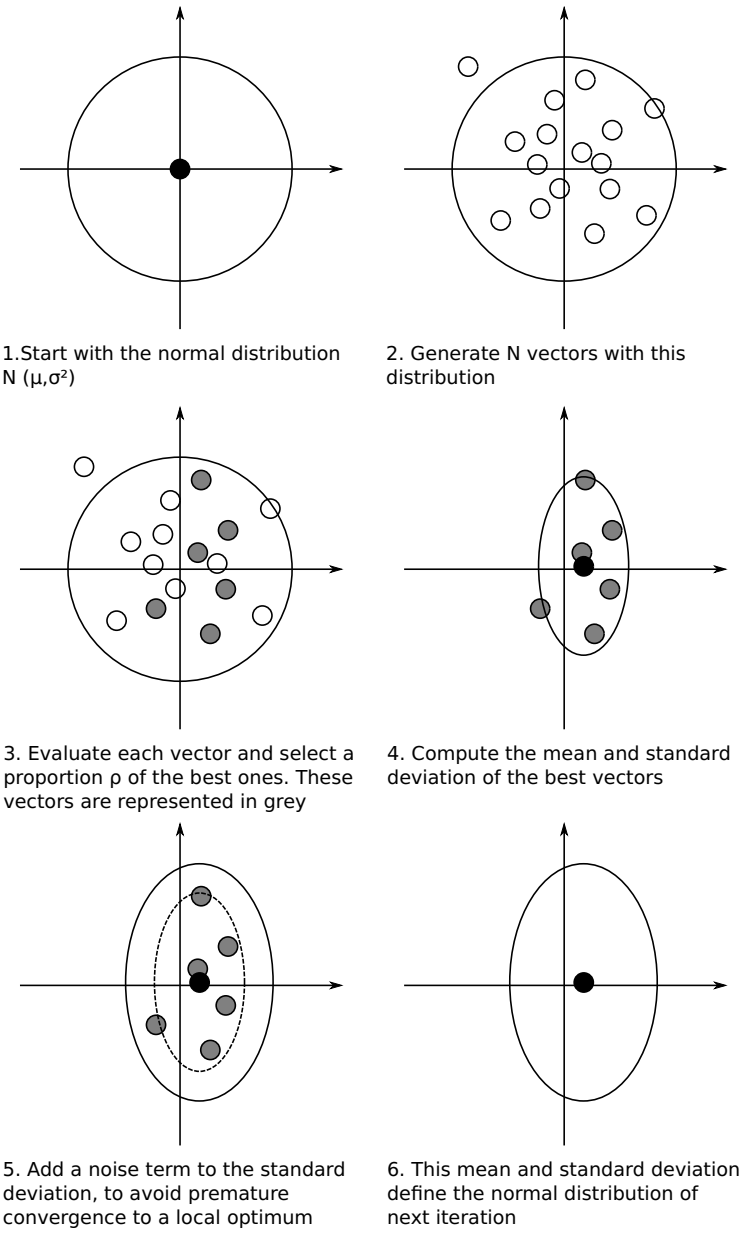
You can find the code of the all class fa_rbfn in appendix 4.2.

2.5 Optimisation, CMAES controller

After obtaining the RBFN controller, the purpose is to create optimized controllers for different sizes of target.

We use in this part the CMA-ES(Covariance Matrix Adaptation Evolution Strategy) algorithm implemented in the python module cma.

We can roughly explain how this algorithm work by the following scheme:



To launch the CMA-ES algorithm, we use the function *LaunchTrajectoriesCMAES* in *LaunchTrajectories.py*. This function run x times (here x = 5)

each trajectories, evaluate them and return the average cost (here the cost equal to the cost in term of muscle activations plus the reward if the target is reach). The algorithm feed the function with the matrix $\boldsymbol{\theta}$ and plays on each weight in order to maximize the cost obtained (i.e. to find the best compromize between the speed to reach the target and the accuracy).

Chapter 3

Study

3.1 Experiments

All algorithms are implemented in Python. The parameters used for the experiments are given in Bidule.

3.1.1 Model tests

We now verify that our model behaves as expected. So we perform two experiments.

1. First, we create a set Γ of initial positions. Γ contains 48 initial positions organized into four sets of different distances to the target. Using the trajectories generated by the NOPS controller, we create the RBFN controller and evaluate his performance on each trajectories of Γ.

2. Second, for each target sizes, we perform x times the reaching task for each initial positions of Γ. We measure different parameters during the generation of the trajectories:

    - coordinate of the elbow and the end-effector
    - articular speed of the shoulder and the elbow
    - the muscle activations
    - the cost of the trajectories (with and without the reward term)
    - the time taken to reach the goal

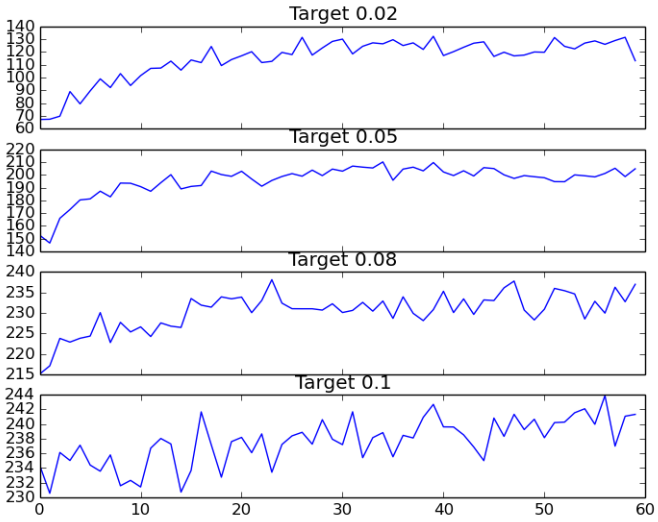
3.1.2 Create a RBFN controller

In this section we study the performance and trajectories obtained with the RBFN controller.

First of all, we generate 1650 trajectories with the NOPS controller in order to train our new controller.(Ici la figure montrant tous les points initiaux des trajectoires generees par NOPS, utilises pour entrainer RBFN) Then, we evaluate the new controller obtained by looking at the cost map.

3.1.3 Optimize the RBFN controller

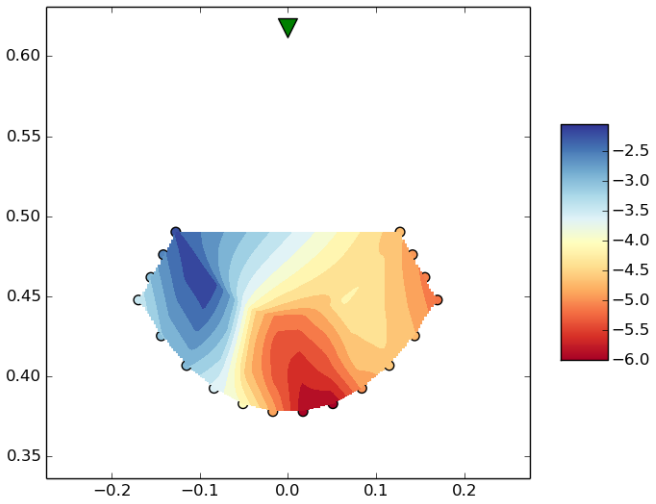
Starting back from the best RBFN controller (best set of θ), we optimize, using the CMA-ES algorithm, for four differents target's size. We specify the number of iterations required and the population size for CMA-ES. We follow the evolution of the performance during the optimization. We now have four new controller optimized for each target's size and we can evaluate them to see if the results are consistent with the model.



3.2 Results

3.2.1 Performance of the RBFN controller

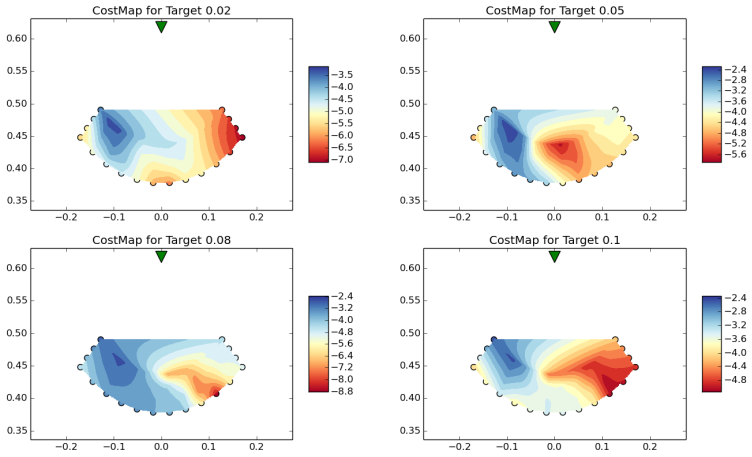
(Here the costMap of NOPS)



As expected, we can observe that starting from the left side of the goal point results in a lower cost than starting from the right side. This is explained by the fact that the optimal muscular strategy for performing these movements differs depending on the direction and the initial position.

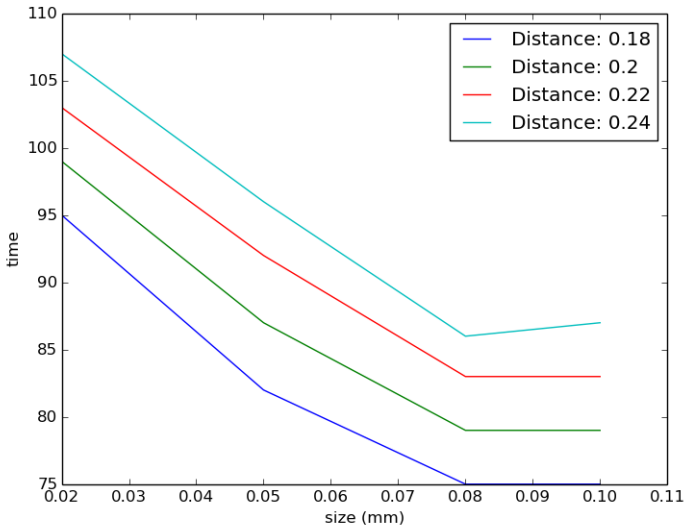
3.2.2 Performance of the CMA-ES controllers

First, we can evaluate the results of the optimization by looking the cost map and if the strategy is still the same. Id est, see that starting point from the left side of the goal point results in a lower cost than starting from the right side. So we plot the cost map obtained after evaluate the different controllers create by the optimization:

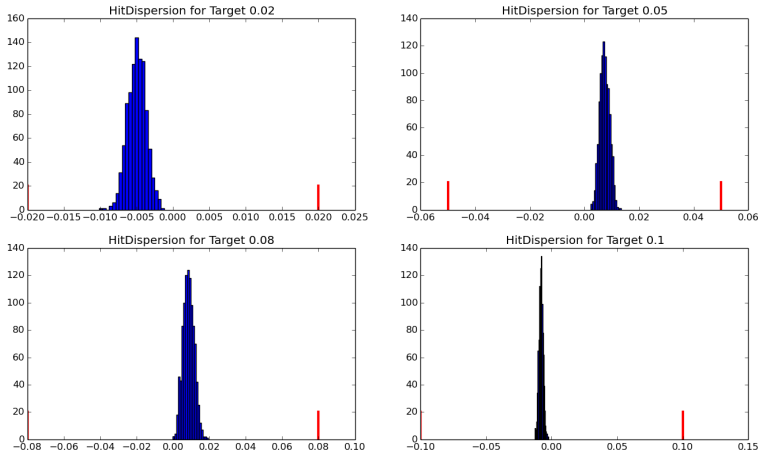


We barely see the previous strategy but the area where the trajectories are the most costly is not as we expected in the righ side. Maybe this is due to the strategy for muscle activation in this area because all the muscles have to be activate to ensure a good trajectory.

Concerning the strategy about the trade-off between the speed to reach the target and the precision. We expect that to ensure to reach the target, when the size is reduced the speed must decrease at the end in order to be more accurate. In order to check this point, we plot the time to reach the target over the size of the target for different distance between the initial points and the target:



We can logically see that the time to reach the target increases as more as the distance to the target increases. One can also observe that the more the target is small, the time taken to reach the target increases.



This figure show the dispertion of the hit points. We see that the dispersion increase as much as the size of the target increases but not as much as we can expect.

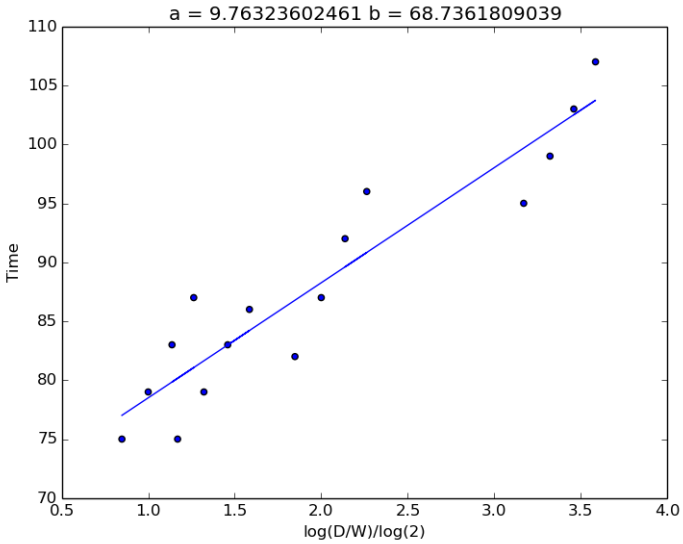
Reproduction of Fitt's law

Fitts' law states that movement time (MT) is linear in its difficulty index (DI), this index being bigger for longer movements and smaller targets. Fitts' law is written:

$$\mathrm{MT} = a + b.\underbrace{\log_2\left(\frac{D}{W}\right)}_{\mathrm{DI}} \tag{3.1}$$

where D is the distance of the movement, W is the width of the target and a and b are linear coefficients. This law was initially studied for one dimensional movements, and then extended for many other contexts [**?**, **?**, **?**, **?**, **?**, **?**].

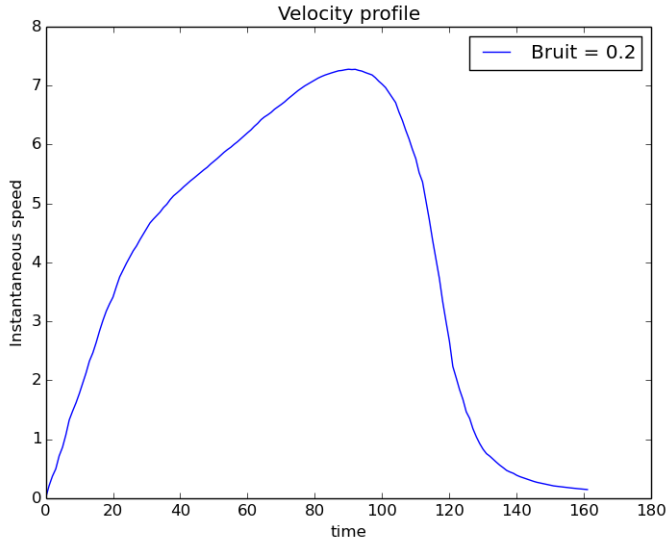
We compute DI values for different distances D and target widths W. the figure below shows the resulting movement time MT over DI.



One can see that we get a clear linear relationships, thus the data is consistent with Fitt's law.

The obtained values of a and b cannot be compared to empirical data from the human motor control literature given the wide variability of these values accross subjects [**?**, **?**].

Velocity profile



This velocity profile is obtained with the smallest target, we can see that at the end the speed decrease a lot. It can be explained by the fact that

to reach a small target, the controller must decrease the speed in order to increase the precision.

The final dispersion in reaching trajectories is generated by motor noise. Following the minimum intervention principle from [**?**], motor noise being proportional to muscular activation, the only way to decrease motor noise is to decrease muscular activation.

Thus, in order to hit a small trajectory, muscular activations should be small by the end of the movement, which can result in first instance in less co-contraction and then in less velocity. Furthermore, a slower movement provides a better opportunity for state estimation to compensate for delayed feedback about the position of the end effector. Taken together, those two phenomena contribute to the fact that an optimal controller should generate less velocity by the end of the movement for a smaller target. So one way to make sure to hit a small target would be to perform a slow reaching movement.

However, as explained above, a slower movement results in a discounted reward, thus the movement should nevertheless be as fast as possible.

As a consequence, the best option for optimizing reaching accuracy under temporal constraints consists in being very fast in the beginning of the movement and much slower in the end. Thus the velocity profile should be asymmetric. The main drive for this asymmetry being motor noise, the more motor noise, the more asymmetric the movement should be. Incidentally, one can observe on the ascending parts of the profiles that we do not get strictly the shape of a bell curve.

At this point of my internship, I have not yet implemented the delayed feedback about the position of the end effector so I can not show this part of the experimentation.

Chapter 4

Appendix

4.1 Nomenclature of arm parameters

Table 4.1: Parameters of the arm model.

m_i	mass of segment i (kg)
l_i	length of segment i (m)
s_i	inertia of segment i $(kg.m^2)$
d_i	distance from the center of segment i to its center of mass (m)
κ	Heaviside filter parameter
$\mathbf{A}$	moment arm matrix $(\in \mathbb{R}^{6\times 2})$
$\mathbf{f_{max}}$	maximum muscular tension $(\in \mathbb{R}^6)$
$\mathbf{M}$	inertia matrix $(\in \mathbb{R}^{2\times 2})$
$\mathbf{C}$	Coriolis force $(N.m \in \mathbb{R}^2)$
τ	segments torque $(N.m \in \mathbb{R}^2)$
$\mathbf{B}$	damping term $(N.m \in \mathbb{R}^2)$
$\mathbf{u}$	raw muscular activation (action) $(\in [0,1]^6)$
σ_u^2	multiplicative muscular noise $(\in [0,1]^6)$
$\tilde{u}$	filtered noisy muscular activation $(\in [0,1]^6)$
$\mathbf{q}^*$	target articular position $(rad \in [0, 2\pi[^2)$
$\mathbf{q}$	current articular position $(rad \in [0, 2\pi[^2)$
$\dot{q}$	current articular speed $(rad.s^{-1})$
$\ddot{q}$	current articular acceleration $(rad.s^{-2})$

Table 4.2: Parameters of the arm.

$\mathbf{l_1}$	arm length (m)	0.3
$\mathbf{l_2}$	forearm length (m)	0.35
$\mathbf{m_1}$	arm mass (kg)	1.4
$\mathbf{m_2}$	forearm mass (kg)	1.1
$\mathbf{s_1}$	arm inertia $(kg.m^2)$	0.11
$\mathbf{s_2}$	forearm inertia $(kg.m^2)$	0.16
$\mathbf{d_1}$	distance from the center of segment 1 to its center of mass (m)	0.025
$\mathbf{d_2}$	distance from the center of segment 2 to its center of mass (m)	0.045
$\mathbf{k_6}$	damping term	0.05
$\mathbf{k_7}$	damping term	0.025
$\mathbf{k_8}$	damping term	0.025
$\mathbf{k_9}$	damping term	0.05
$\mathbf{a_1}$	moment arm matrix	0.04
$\mathbf{a_2}$	moment arm matrix	-0.04
$\mathbf{a_3}$	moment arm matrix	0.0
$\mathbf{a_4}$	moment arm matrix	0.0
$\mathbf{a_5}$	moment arm matrix	0.028
$\mathbf{a_6}$	moment arm matrix	-0.035
$\mathbf{a_7}$	moment arm matrix	0.0
$\mathbf{a_8}$	moment arm matrix	0.0
$\mathbf{a_9}$	moment arm matrix	0.025
$\mathbf{a_{10}}$	moment arm matrix	-0.025
$\mathbf{a_{11}}$	moment arm matrix	0.028
$\mathbf{a_{12}}$	moment arm matrix	-0.035

Table 4.3: Parameters of the muscles.

$\mathbf{f_{max1}}$	Maximum force exerted by the shoulder flexor	700
$\mathbf{f_{max2}}$	Maximum force exerted by the shoulder extensor	382
$\mathbf{f_{max3}}$	Maximum force exerted by the elbow flexor	572
$\mathbf{f_{max4}}$	Maximum force exerted by the elbow extensor	445
$\mathbf{f_{max5}}$	Maximum force exerted by the double-joints flexor	159
$\mathbf{f_{max6}}$	Maximum force exerted by the double-joints extensor	318

4.2 RBFN implementation

```
15 class fa_rbfn():
16
17     def __init__(self, nbFeature):
18         '''
19         Initializes class parameters
20
21         Input:      -nbFeature: int, number of feature
               in order to perform the regression
22
23         '''
24         self.nbFeat = nbFeature
25         self.title = "rbfn"
26
27     def setTrainingData(self, inputData, outputData):
28         '''
29         Verifies the validity of the given input and
               output data
30         Data should be organize by columns
31
32         Input:       -inputdata, numpy N-D array
33                      -outputData, numpy N-D array
34         '''
35         self.inputData = inputData
36         self.outputData = outputData
37         #Getting input and output dimensions and
               number of samples
38         self.inputDimension, numberOfInputSamples = np
               .shape(inputData)
39         self.outputDimension, numberOfOutputSamples =
               np.shape(outputData)
40         #check if there are the same number of samples
```

```
            for input and output data
41          assert(numberOfInputSamples ==
               numberOfOutputSamples), "Number of samples
               not equal for output and input"
42          self.numberOfSamples = numberOfInputSamples
43          self.theta = np.zeros((self.nbFeat, self.
               outputDimension))
44
45      def computeA(self, A, fop):
46          At = np.dot(fop, fop.T)
47          for i in range(At.shape[0]):
48              for j in range(At.shape[1]):
49                  A[i,j] = At[i,j]
50
51      def computeb(self, b, fop):
52          bt = np.dot(fop, self.outputData.T)
53          for i in range(bt.shape[0]):
54              for j in range(bt.shape[1]):
55                  b[i,j] = bt[i,j]
56
57      def train_rbfn(self):
58          '''
59          Defines the training function to find solution
               of the approximation
60
61          '''
62          fop = self.computeFeatureWeight(self.inputData
               .T)
63          n = self.nbFeat**self.inputDimension
64          #creation of the shared objects between
               process
65          AshareObj = Array(ct.c_double, n*n)
66          bshareObj = Array(ct.c_double, n*self.
               outputDimension)
67          #link shared object to a numpy object
68          AnumpyShare = np.frombuffer(AshareObj.get_obj
               ())
69          bnumpyShare = np.frombuffer(bshareObj.get_obj
               ())
70          #Reshaping of the numpy object to obtain an
               array with the dimension desired
71          A = AnumpyShare.reshape((n, n))
72          b = bnumpyShare.reshape((n, self.
               outputDimension))
```

```
73          #creation of the different processes
74          p1 = Process(target=self.computeA, args=(A,
                fop))
75          p2 = Process(target=self.computeb, args=(b,
                fop))
76          p1.start()
77          p2.start()
78          p1.join()
79          p2.join()
80          self.theta = np.dot(np.linalg.pinv(A), b)
81
82      def setCentersAndWidths(self):
83          '''
84          Sets the centers and widths of Gaussian
                features.
85          Uses linspace to evenly distribute the
                features.
86
87          '''
88          #get max and min of the input data
89          minInputData = np.min(self.inputData, axis =
                1)
90          maxInputData = np.max(self.inputData, axis =
                1)
91          rangeForEachDim = maxInputData - minInputData
92          #set the sigmas
93          widthConstant = rangeForEachDim / self.nbFeat
94          #create the diagonal matrix of sigmas to
                compute the gaussian
95          self.widths = np.diag(widthConstant)
96          #coef for gaussian
97          self.norma = 1/np.sqrt(((2*np.pi)**self.
                inputDimension)*np.linalg.det(self.widths))
98          linspaceForEachDim = []
99          #set the number of gaussian used and allocate
                them in each dimensions
100         for i in range(self.inputDimension):
101             linspaceForEachDim.append(np.linspace(
                    minInputData[i], maxInputData[i], self.
                    nbFeat))
102         #get matrix with all the possible combinations
                to find each centers
103         self.centersInEachDimensions = cartesian(
                linspaceForEachDim)
```

```
104
105     def computeFeatureWeight(self, inputData):
106         '''
107         Computates Gaussian parameters
108
109         Input:      -inputData: numpy N-D array
110
111         Output:     -phi: numpy N-D array
112         '''
113         #if only one sample
114         if inputData.shape[1] == 1:
115             x_u = inputData - self.
                    centersInEachDimensions.T
116             x_u_s = np.dot(x_u.T, np.linalg.pinv(self.
                    widths))
117             x = x_u_s * (x_u.T)
118             xf = np.sum(x, axis = 1)
119             phi = self.norma*np.exp(-0.5*xf)
120         #if more than one sample
121         else:
122             for i in range(inputData.shape[0]):
123                 x_u = np.array([inputData[i]]).T -
                        self.centersInEachDimensions.T
124                 x_u_s = np.dot(x_u.T, np.linalg.pinv(
                        self.widths))
125                 x = x_u_s * (x_u.T)
126                 xf = np.sum(x, axis = 1)
127                 xfe = self.norma*np.exp(-0.5*xf)
128                 if i == 0:
129                     phi = np.array([xfe]).T
130                 else:
131                     phi = np.hstack((phi, np.array([
                            xfe]).T))
132         return phi
133
134
135     def computesOutput(self, inputData, theta):
136         '''
137         Returns the output depending on the given
                input and theta
138
139         Input:      -inputData: numpy N-D array
140                     -theta: numpy N-D array
141
```

```
142         Output:     -fa_out: numpy N-D array, output
                approximated
143         '''
144         if inputData.shape[1] == 1:
145             phi = self.computeFeatureWeight(inputData)
146             fa_out = np.dot(phi.T, theta)
147         else:
148             phi = self.computeFeatureWeight(inputData)
149             fa_out = np.dot(phi.T, theta)
150         return fa_out
```

Bibliography

[1] H. Kambara, K. Kim, D. Shin, M. Sato, and Y. Koike. Learning and generation of goal-directed arm reaching from scratch. *Neural networks*, 22(4):348–61, 2009.

[2] M. Katayama and M. Kawato. Virtual trajectory and stiffness ellipse during multijoint arm movement predicted by neural inverse models. *Biological Cybernetics*, 69(5):353–362, 1993.

[3] W. Li. *Optimal control for biological movement systems*. PhD thesis, University of California, San Diego, 2006.

[4] D. Mitrovic, S. Klanke, and S. Vijayakumar. Adaptive optimal control for redundantly actuated arms. In *Proceedings of the Tenth International Conference on Simulation of Adaptive Behavior*, 2008.