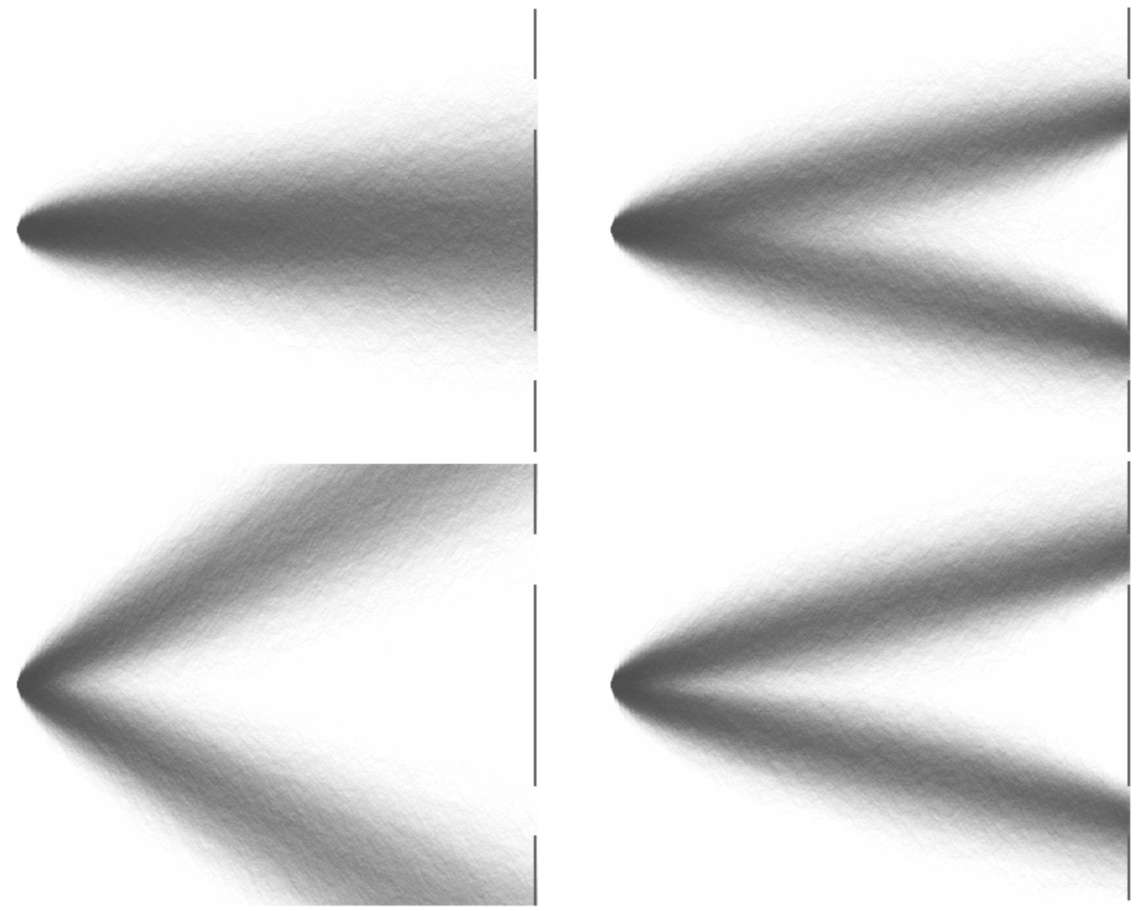
Path Integral Control with history dependent basis functions

Anton de Groot

March 27, 2019



aammdegroot@gmail.com

Contents

1. Introduction **3**

2. Theory **4**

2.1. Basis functions 4

2.2. Delay lines and history dependent basis functions 4

2.3. Taken's embedding theorem 4

2.4. The algorithm 5

2.5. The algorithm written in Matlab 7

2.5.1. the main algorithm 7

3. Discussion of the results **9**

Appendices **12**

A. About the cover page 12

1. Introduction

This research was conducted to explore whether it is possible to successfully apply time dependent basis functions to a specific problem which involves multi-task learning. In general it is not trivial to find the optimal control to a given scenario, certainly when it involves solving multiple tasks at once. To find the optimal control Path Integral Control theory (PIC) will be used. PIC formulates an optimal control in terms of a cost associated with a path taken and an end-cost, the successfulness of a taken path. In our daily lives we would like to avoid obstacles that are in our paths, but we also do not want to swerve excessively. PIC theory will provide an algorithmic description to accomplish this.

In this thesis we can imagine a drone trying to fly through a gap in a wall. The algorithm as defined in section 2.4 will compute an optimal control function, such that the drone will fly through one of two gaps with a high percentage of success. Then a constant drift term will be added, a constant breeze trying to push the drone away from the gaps. The drone then needs to learn 2 tasks: to fly through the gap, and to adjust for the wind and perhaps utilize it to its advantage. If the drift term is too large the drone will not be able to fight against the drift. If not a single path ends up through a gap, that is to say, not a single target is reached, then the algorithm in section 2.4 cannot be applied and is therefore not able to learn.[1] This is the only constraint on the magnitude of the drift term. The direction of the drift term is not known and is per path randomly chosen to be either positive or negative.

The model without basis functions has inputs that are only dependent on its current position to determine what the control function should be. This means that there is no input that informs us about the drift term. Time dependent basis functions, however, may give us an insight into what the drift term might be if we keep record of our past steps. The model with history dependent basis functions might then, in principle, be able to learn about the drift and use it to its advantage to lower the control cost and use it to drift towards its target.

In summary: the goal of this thesis is to see if history dependent basis functions are able to learn a so called multi-task problem.

[1]In addition all end-costs will be infinite and there will not even be a measure of 'better performance'.

2. Theory

2.1. Basis functions

In the case of linear regression, where we want to approximate a certain function given some inputs we can write it in the form: [Bishop(2006)]

$$u(\vec{x},\vec{\omega}) = \sum_{i=1}^{D} \omega_i \phi_i(x) \tag{1}$$

where ϕ_i is a so called basis function and $u(x)$ will be our control function. There is a plethora of basis functions; in fourier series a combination of sines and cosines can approximate certain functions. So in this case the basis functions could be sines, but other options also include polynomials such that $\sum_{0}^{3} \phi_i(x) = 1 + x + x^2 + x^3$. In this thesis I will mostly look at the hyperbolic tangent function of the form: $\phi_i(x) = \tanh(xb_i + c_i)$

This basis function is chosen because we would like to have a basis function that does not result in a control term that can grow infinitely large, as it is not physically possible to exert such a force. This can be the case for polynomial basis functions. There is only one fixed point per basis function for hyperbolic tangents, but infinitely many stable and unstable fixed points if we choose sinusoidal basis functions.

2.2. Delay lines and history dependent basis functions

Delay lines repeat what the past few positions were. For example the current position $x(t_i)$ for N different paths will be a vector, a generalized position X will then be a matrix consisting of n past positions

$$X(t_i) = \begin{pmatrix} \vec{x}(t_i) & \vec{x}(t_{i-1}) & \dots & \vec{x}(t_{i-n}) \end{pmatrix} \tag{2}$$

The position $X(t_i)$ is then used in basis functions of the form $\tanh(\vec{X}b_j + c_j)$.

2.3. Taken's embedding theorem

The goal is to learn based on $O(f(x_1, x_2, \dots, x_N))$ where O is the model which is determined by some unknown function f. Taken's embedding theorem states we can determine this f by conducting several measurements of its state at different times. However, it should be noted that Taken's theorem applies to deterministic cases to determine strange attractors. [Huke(2006)] In this case we ignore that it should be deterministic and apply it to a stochastic scenario. A model in this case could be of the form $\dot{x} = b + u + \zeta$. The delay lines are then the equivalent of taking multiple consecutive measurements. [Huke(2006)]

2.4. The algorithm

In order to compute the trajectory of a certain path in discrete time, we can express the displacement dX as the following equation:

$$dX = bdt + \sigma(udt + \rho dW), \tag{3}$$

where b is a drift term, which is either plus or minus a certain constant value. This means that the wind is blowing constantly in the same direction for an entire path, but that the wind direction for different paths might blow in opposite directions. U is the aforementioned control term. This can be any arbitrary Markov control [Thijssen(2016)], i.e. a stochastic control in discrete time. And dW is a stochastic variable, ρ and σ are positive real constants. Which can be rewritten to the form:

$$\dot{X} = b + u + \xi \tag{4}$$

where ρ and σ are set to 1 and where ξ equals:

$$\xi = \frac{dW}{dt} \tag{5}$$

To be able to use Path Integral Control theory we must use a certain path cost, defined as:

$$S^u(t) = \int_{t_i}^{t_f} V(\tau, X^u(\tau)) + \frac{1}{2}u(\tau, X^u(\tau))^2 d\tau + \int_{t_i}^{t_f} u(\tau, X^u(\tau))dW \tag{6}$$

Note that the cost S depends on future values of X and is therefore not adaptive. [Thijssen(2016)] It also includes a stochastic integral with respect to Brownian motion. This is somewhat unusual because the stochastic integral vanishes when taking the expected value, as is done in equations (10),(**??**). However when performing a change of measure with a drift u, such a term appears naturally. [Thijssen(2016)]

The function $V(\tau, X(\tau))$ is a cost-function associated with a certain position. This cost-function will be infinite when it coincides with the wall at time t_f and zero otherwise, for all other timesteps $t \neq t_f$ this will also be zero. This means that a path is only punished by this term by the result it yields: success or failure. This cost term is called the end-cost $\Phi(X(t_f))$.

$$S^u(t) = \Phi(X^u(t_f)) + \int_{t_i}^{t_f} \frac{1}{2}u(\tau, X^u(\tau))^2 d\tau + \int_{t_i}^{t_f} u(\tau, X^u(\tau))dW \tag{7}$$

The second term in the above equation: $\int u^2 d\tau$, means that any large control functions will be punished. The last term in the equation that reads: $\int udW$, indicates that the deviation from a certain path by the product of the control times a stochastic variable cannot be too large either.

It can be shown that if you assume the optimal controller $u^\star$ to be of the parametrized form: [Thijssen and Kappen(2015)]

$$u^\star(x,t) = A^\star(t)h(x,t) \tag{8}$$

that this leads to equation 9. Where h is the k-dimensional basis functions applied to x and the parameters constituting the parameters $A \in \mathbb{R}^{N\times k}$

Solving this equation will then lead to:

$$A(t)\langle hf'\rangle(t) = \langle uf'\rangle(t) + \lim_{r\to t}\left\langle \frac{\int_t^r f'(\tau)dW(\tau)}{r-t}\right\rangle \tag{9}$$

$$A = \left(\langle uh'\rangle + \langle h(dW+u)\rangle\right)\langle hh'\rangle^{-1} \tag{10}$$

$$u(x,t) = A(t)h(x,t) \tag{11}$$

We now have almost everything we need to start applying the algorithm. We also need a measure of performance for this we first need the weight of a path: [Thijssen(2016)]

$$\alpha^u = \frac{e^{-S^u_{t_0}}}{\mathbb{E}\left[e^{-S^u_{t_0}}\right]} \tag{12}$$

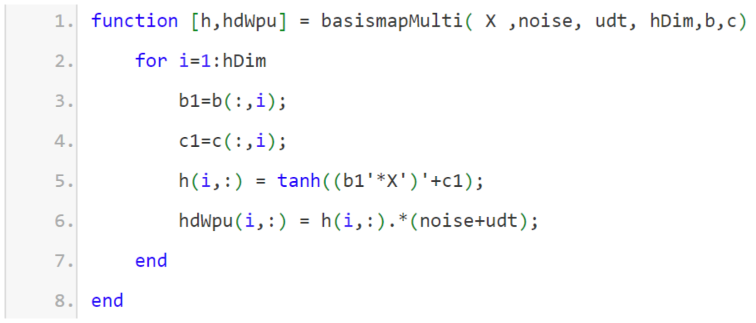
and the fraction of effective samples which is

$$\lambda^u = \frac{1}{\mathbb{E}\left[(\alpha^u)^2\right]} = \frac{1}{1+var\left((\alpha^u)^2\right)} \qquad 0 \le \lambda^u \le 1 \tag{13}$$

So if we can compute A we can also compute the control $u(x,t)$ after which we can compute a new A. Here the term $\int dW$ is a stochastic integral with dW brownian noise.

2.5. The algorithm written in Matlab

The function in figure 2.5 is used to calculate the terms $h(dW+u)$ and the term h in equation 10. The term $h(x)$ is a matrix with length *"the number of paths"*, and width *"the number of basisfunctions"*. Furthermore the terms b, c are zero mean random variables. The variables c_{ij} are the same for every path i but different per basisfunction j, whereas the variables b_{ij} are different for every path and basisfunction. The steps in line 3 and 4 are there to cut matrix of width 1 down to a vector. [2]



```
1. function [h,hdWpu] = basismapMulti( X ,noise, udt, hDim,b,c)
2.     for i=1:hDim
3.         b1=b(:,i);
4.         c1=c(:,i);
5.         h(i,:) = tanh((b1'*X')'+c1);
6.         hdWpu(i,:) = h(i,:).*(noise+udt);
7.     end
8. end
```

Figure 1: The function used in the algorithm

The algorithm described in figure 2.5.1 is only one iteration of the algorithm. The whole algorithm needs to be put in a loop to keep on finding new expressions for A in equation 10 which can be used to find a new and hopefully better control term in equation 11.

2.5.1. the main algorithm

The variable X in line 18 is a matrix consisting of the vector of the current positions x in the first column followed by the subsequent delaylines expressed in a matrix.

Only the successful paths ending in one of the gaps are eventually important. So in line 21 we only look if a path ends in a gap and give it the value 1, otherwise it is 0. In line 28 we multiply this with the pathcosts. Doing it this way we can avoid using infinities in the end-cost $\Phi(X(t_f))$ as we saw in equation 7

[2]and var1? or something else?

```matlab
for t = 1:timesteps
    h1 = h(:,:,t);
    udt(:,t) = h1'*NewA(:,t)*dt;
    [h(:,:,t+1),hdW(:,:,t+1)] = basismapMulti(X(:,:,t),sqrt(dt)*noise(:,t)',udt(:,t)',hDim,b,c);

    x(:,t+1) = x(:,t) + udt(:,t) + sqrt(dt)*noise(:,t)+Drift*dt;
    for d = 1:Ndelay+1
        if d==1
            delay(:,t+1,d) = x(:,t);
        else
            delay(:,t+1,d) = delay(:,t,d-1);
        end
    end
    DelayArray = delay(:,t+1,:);
    Delay1ine = [];
    Delay1ine = reshape(DelayArray,N,Ndelay);
    DelayArray = [];
    X(:,:,t+1) = [x(:,t+1) Delay1ine];
end
%% calculate pathcost
w = boolean((abs(x(:,timesteps))<2).*(abs(x(:,timesteps))>1));
logpathcost = -(udt.^2/dt + udt.*noise/sqrt(dt));
logpathcost = sum(logpathcost,2);
logpathcost = logpathcost - max(logpathcost);
pathcost = exp(logpathcost);
%% success /EffSS
successratio = mean(w);
w = w.*pathcost;
w = w/sum(w);
EffSS = 1/(w'*w);
%% determine New A
if EffSS > 0
    for t = 1:timesteps
        hdWweighted = hdW(:,:,t)*w;
        n = length(w);
        D = spdiags(w(:),0,n,n);
        hh = h(:,:,t)*D*h(:,:,t)';
        NewA(:,t) = (pinv(hh)*hdWweighted)'/dt;
    end
end
x = x_reset;
```

Figure 2: The main algorithm, written in Matlab.

3. Discussion of the results

During the experiment I looked at two scenarios. In the first scenario an extra drift term is multiplied to the control term which is represented by the symbol σ in equation 2.4, and in the second scenario an extra drift term is added which is represented by the symbol b in the same equation.

As it turns out the algorithm cannot improve when the drift is multiplied with the control. There is no difference between using 0 and 4 delay lines and it is unable to learn.[3] So I took a look at the situation where the drift was just the added term Drift*dt.

When I used 15 basisfunctions and looked at 0 and 4 delay lines the cases where I used delay lines worked a lot better. The next thing to consider is when you are using delay lines you are in principle using more basisfunctions, could this simply be a case of just using more basisfunctions? As it turns out a lot of the difference can be explained by using more basisfunctions, but not all of it. So what had to be checked is if using 50 basisfunctions without any delay lines works differently from using 50 basisfunctions spread out over the original input line and the 4 delay lines with 10 basisfunctions for each line.[4]

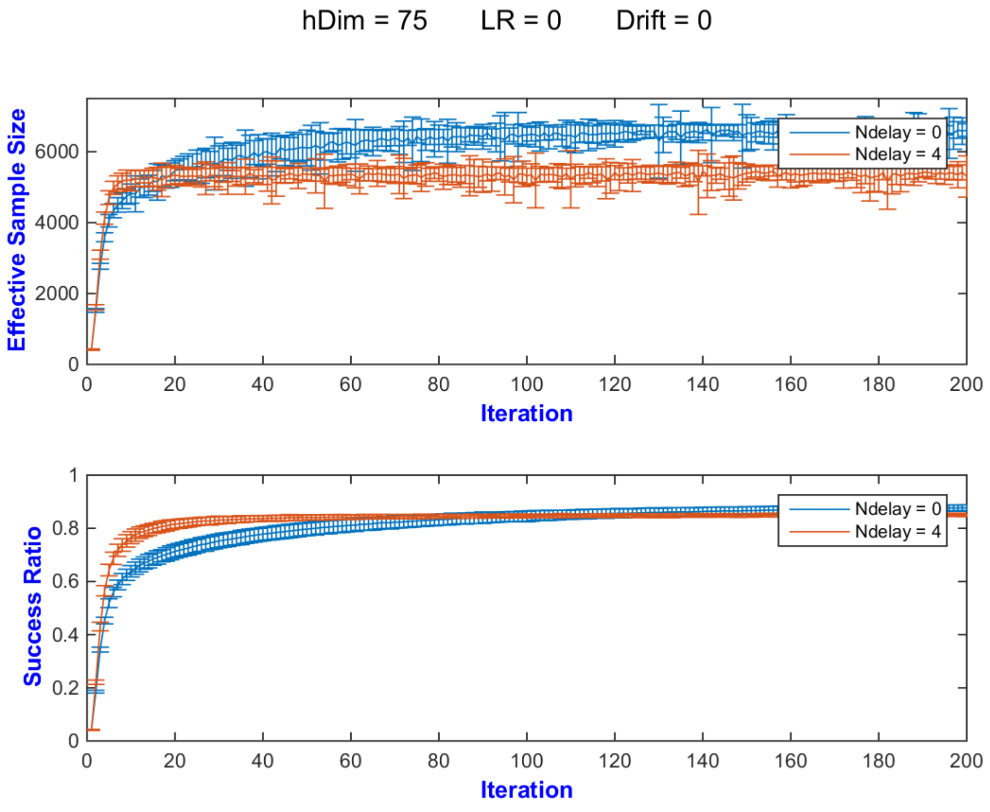


Figure 3: Comparing 4 delay lines with 0 delay lines when there is no drift term

[3]why is this? Expand on this
[4]explain what a delay line is

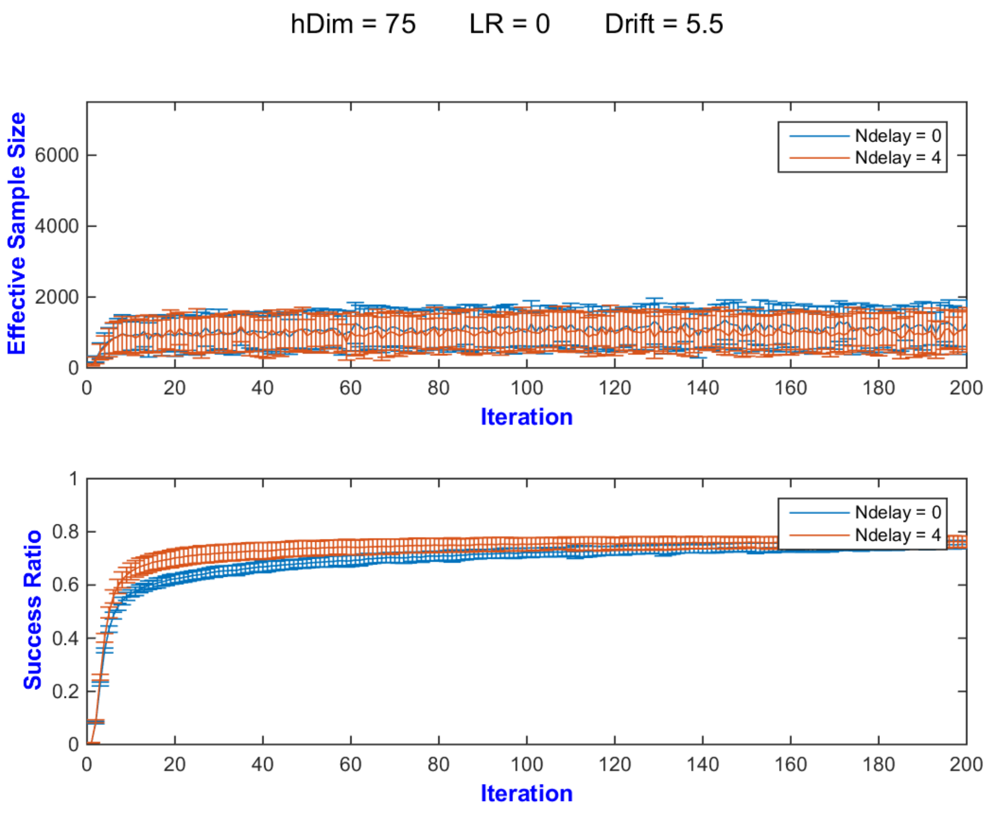


Figure 4: Comparing 4 delay lines with 0 delay lines when there is a strong drift term

The result of which is that for sufficient iterations of the algorithm[5] they will both converge to the same result in terms of effective sample size and the same success rate. But when looked at the first 50 iterations the delay lines go faster to a higher success rate than when no delay lines are used, the effective sample size however does not differ significantly from one another.

Because an earlier version of the algorithm there was a suspicion that there was a fundamental different behavior when we added a large drift term. In those instances we would see that the algorithm learned The difference between now and the previous situation where we saw that the effective sample size and success ratio went down to zero after a few iterations. Whereas after 1 or 2 iterations the success ratio was going up. In that scenario there was a significant difference between using delay lines and no delay lines, where for no delay lines the Effss went to zero in the case for delay lines the effss didn't go to zero but remained constant. The problem here however was that the Effss didn't go up after a random initialization and the success rate went to zero. This means that the algorithm was not learning. The problem was a bug in the update rules for the algorithm which caused overfitting if there was no drift term present (meaning the Effss went down even if the success rate was high). A higher drift meant that the Effss went faster to zero after fewer iterations of the algorithm.

This is no longer applicable, there is no more significant difference. The algorithm

[5]i.e. 80+ iterations

is completely dependent on rollouts going through the gaps. As long as even a single rollout goes through a gap the algorithm can learn to improve itself to increase the success rate. If there are no successfull rollouts then the algorithm will break down since it inverts a matrix consisting of components that are dependent on the amount of successful rollouts it would then try to invert a zero-matrix causing errors.

It can be concluded that the algorithm using delay lines works significantly faster than the algorithm without delay lines.

Appendices

A. About the cover page

The image on the cover page shows what the result is when Path Integral Control is applied. The upper row shows the scenario where there is no drift term and the lower row shows the scenario where there is a strong drift term. On the left side we see the initial unlearned scenario, and on the right side we see the result of applying the algorithm. [Thijssen and Kappen(2015)]

References

[Bishop(2006)] Christopher Bishop. *Pattern recognition and machine learning*, chapter 3, page 138. Springer, New York, 2006.

[Huke(2006)] J.P. Huke. Embedding nonlinear dynamical systems: A guide to takens' theorem. 2006. ISSN 1749-9097. URL `http://eprints.maths.manchester.ac.uk/175/1/embed.pdf`.

[Thijssen(2016)] Sep Thijssen. *Path Integral Control*. PhD thesis, Radboud University, 28 nov 2016. URL `http://www.snn.ru.nl/~bertk/thesis.pdf`.

[Thijssen and Kappen(2015)] Sep Thijssen and H. J. Kappen. Path integral control and state-dependent feedback. *Phys. Rev. E*, 91:032104, Mar 2015. doi: 10.1103/PhysRevE.91.032104. URL `https://link.aps.org/doi/10.1103/PhysRevE.91.032104`.