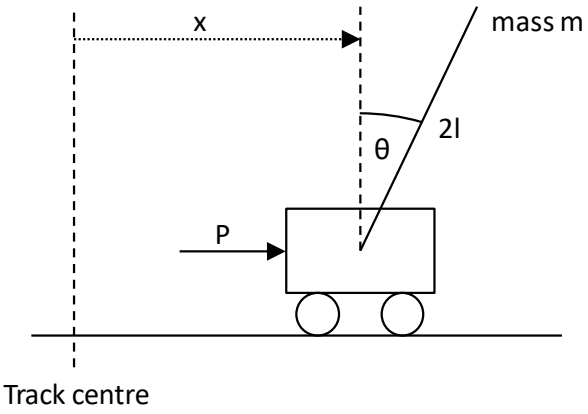
Applying an n-Step Sarsa learner to the cart-pole balance problem

Introduction

This paper documents my development of an n-Step Sarsa learner for the cart and pole balance problem, where the challenge is to keep a pole attached to a movable cart upright for as long as possible.

In summary, as illustrated below, a cart of mass M is free to move one-dimensionally along a track in either direction. A pole of mass m and length 2l is attached via a hinge to the cart at its centre of mass. A learning application may apply a horizontal force of P, -P or 0 to the cart through its centre of mass at given time intervals. Initially, at time t=0, the cart and pole are both stationary with the pole at some angle θ to the vertical and the cart located at the centre of the track. The tendency is for the pole to fall under gravity, and the challenge for the learning application is to apply the appropriate force to the cart at the appropriate times for the pole to remain upright. The track has a given length, and the learner is deemed to have failed each time the cart reaches either end of the track or the pole angle θ exceeds a given angle.



This paper includes:

- An explanation of the approach to the machine learning aspects of the problem.
- A description of the initial development and refinement of the solution, together with results for the initial version of the problem.
- Additional investigations and results generated.
- The derivation of the equations used to model the cart-pole dynamics.

Learning approach

The n-step Sarsa algorithm

In my earlier investigations into reinforcement learning I had used a Monte Carlo approach, where learning only takes place at the end of each episode. In this exercise I decided to explore a temporal difference approach, where the value of a state-action pair is updated using earlier estimates

without waiting for the episode to end.

As explained in (Sutton & Barto, 2018), a basic every-visit Monte Carlo algorithm (where all updates occur at the end of each episode) is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_t - Q(S_t, A_t)]$$

where:

$Q(S_t, A_t)$ is the value of a the state-action pair at timestep t

α is the learning rate, a value larger than 0 and less than or equal to 1

G_t is the return for the episode at timestep t given by $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots$ where R_i is the reward earned from timestep i and γ is a discount rate (a value between 0 and 1).

The simplest equivalent temporal difference algorithm is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

In essence, the episode return value G_t, of the Monte Carlo method has been replaced by a return *estimate* given by $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$, which means that the update of $Q(S_t, A_t)$ can occur at the very next timestep instead of at the end of the episode.

Because this algorithm relies on the five values of $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$, it is known as the Sarsa algorithm.

The n-step Sarsa bridges the gap between the Monte Carlo approach and the pure Sarsa algorithm. Instead of updating at every timestep, it updates every n timesteps, the goal being to get the best of both worlds – the accuracy of the Monte Carlo method and the speed of the pure Sarsa approach.

The n-step Sarsa algorithm is written as:

$$Q_{t+n}(S_t, A_t) \leftarrow Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)]$$

where:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n})$$

and $G_{t:t+n} = G_T$ if $t + n \geq T$ where T is the final timestep of the episode.

Function approximation

Where the state space of a problem is made up of a finite number of states (because in each dimension a state can only take a finite number of discrete values), a tabular method of state-action value assessment is possible. However, in this instance (as is often the case), the state space is made up of continuous variables – being cart position, cart speed, pole angle and pole angular velocity. Speed and position can take an infinite number of values, even though they may fall within a defined finite range, as they do in this case.

As a result, function approximation is required to evaluate state-action value instead of a tabular method. As explained in (Sutton & Barto, 2018), linear function approximation using semi-gradient descent is relatively simple but effective.

The general state-action valuation function is given by:

$$Q(S, A) = \sum_{i=1}^{d} w_i x_i(S, A) = \boldsymbol{w}^T \boldsymbol{x}(S, A)$$

where $\boldsymbol{w}$ is a weight vector and $\boldsymbol{x}$ is known as a feature vector. The translation of a state-action to a feature is required in linear function approximation so that complex interactions between state variables can be taken into account which the linear model cannot otherwise accomplish. (My

approach to feature construction is described in the following section.)

The goal of a function approximation learner is to learn the best possible values for the weight vector $\boldsymbol{w}$. A perfect answer is impossible, as there are infinite possible state-action pairs and only a finite number of weights – hence the function will always "generalize" for groups of state-action values rather than giving the best answer for each individual state-action pair.

The semi-gradient method works by adjusting the weight vector in the direction that would reduce the square of the error for the most recent timestep, where the square of the error is given by:

$E = \left(U_t - Q(S_t, A_t, \boldsymbol{w}_t)\right)^2$ where U_t is some "target value"

Thus the general weight update function is written as:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - {}^{\alpha}/_{2}\nabla([U_t - Q(S_t, A_t, \boldsymbol{w}_t)]^2)$$
$$= \boldsymbol{w}_t + \alpha[U_t - Q(S_t, A_t, \boldsymbol{w}_t)]\nabla Q(S_t, A_t, \boldsymbol{w}_t)$$

where $\nabla f(\boldsymbol{w})$ is the vector of partial derivates with respect to the coordinates of the weight vector, such that:

$$\nabla f(\boldsymbol{w}) = \left(\frac{\partial f(\boldsymbol{w})}{\partial w_1}, \frac{\partial f(\boldsymbol{w})}{\partial w_2}, \ldots, \frac{\partial f(\boldsymbol{w})}{\partial w_d}\right)^T$$

For linear function approximation, the weight update algorithm then becomes very simple:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[U_t - Q(S_t, A_t, \boldsymbol{w}_t)]\boldsymbol{x}(S_t, A_t,)$$

Finally, the weight update algorithm can be made specific to the n-step Sarsa case by supplying the appropriate target value as U_t:

$$\boldsymbol{w}_{t+n} = \boldsymbol{w}_{t+n-1} + \alpha[G_{t:t+n} - Q(S_t, A_t, \boldsymbol{w}_{t+n-1})]\boldsymbol{x}(S_t, A_t)$$

where, as before:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^{n} Q_{t+n-1}(S_{t+n}, A_{t+n})$$

and $G_{t:t+n} = G_T$ if $t + n \geq T$ where T is the final timestep of the episode.

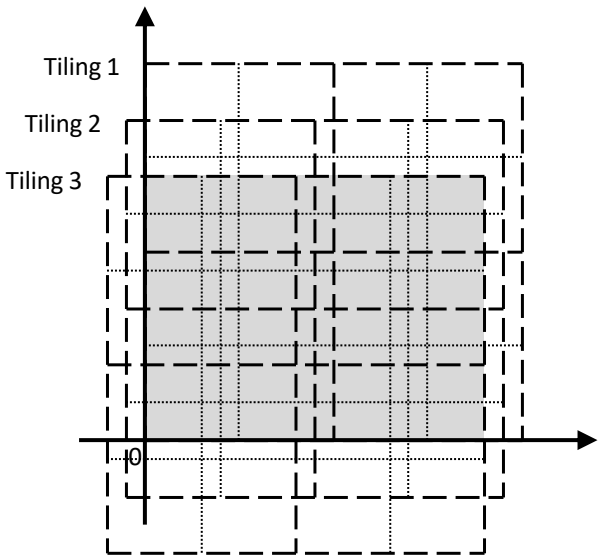
Feature construction

I chose to use two approaches described in (Sutton & Barto, 2018) – tile coding and a Fourier basis feature constructor. Note that in both cases all weight values were initially set to zero.

Tile coding

Tile coding is an approach where the state-action space is mapped with overlapping "tilings" made up of "tiles" each covering a region of the state-action space. A point in state-action space is then represented by those tiles in which that point falls. The feature vector representing a point in state-action space is a one-dimensional vector with a single value of 0 or 1 for each of the tiles. When the point in state-action space falls within a tile, the corresponding value in the feature vector is 1. Otherwise, it is 0.

The diagram below illustrates how a tile coder might be constructed for a problem with 2 dimensional space (i.e., a state of 1 dimension and an action).



The tile coder shown consists of three tilings, each containing 16 tiles in a 4 x 4 grid. The tilings are offset from each other in this case using a vector of $(1, 3)^T$. It is this offsetting that creates the overlapping effect and the granularity or resolution of the tile coder. As is apparent from the diagram, the offsetting means that the area covered by all tilings (the grey area) is smaller than the area covered by the aggregation of all tiles. For a consistent resolution therefore, the limits of the state space must be known and must correspond to those of the grey area.

The effective resolution of the tile coder is quantified by the dimension of a fundamental unit, given by

$u = w/n_t$ where u is the fundamental unit, w a tile width, and n_t the number of tilings

(Sutton & Barto, 2018) recommends offsetting the tilings asymmetrically so as to avoid a non-uniform generalisation effect. Specifically, they recommend using a displacement vector between

tilings of $(u, 3u, 5u, \dots, (2k-1)u)$ where u is the tile coder's fundamental unit as defined above.

A point in state-action space will only be present in one tile of each tiling, so the resulting feature vector for any given state-action will contain only zeros and ones, with the number of ones equalling the number of tilings. This is advantageous as only this number of weights is updated in each iteration. The Fourier basis approach is far less efficient, updating all weights in each iteration.

The approach I used to generate the tilings and then to calculate the resulting features from a given state-action is given in Appendix A.

Fourier basis

Fourier series are used to approximate some periodic function by combining sinusoidal functions with frequencies that are multiples of some base frequency. Clearly, the state-action valuation function is usually not periodic, but if a maximum and minimum is known or can be set for each state dimension (and the action values) then this is not a problem. The largest period of the Fourier

series is simply set to match the range between the maximum and minimum. And if the period is set to twice that range, so that the range matches the base half-period, then only the cosine elements of the Fourier series need be used.

A fuller explanation is available in (Sutton & Barto, 2018), but in summary, if each state (and action) value is normalised so that it is in the range 0 to 1, then the i[th] feature in a v-order Fourier cosine basis can be written as:

$x_i(\boldsymbol{s}) = \cos(\pi \boldsymbol{s}^T)\, \boldsymbol{c}_i$

where:

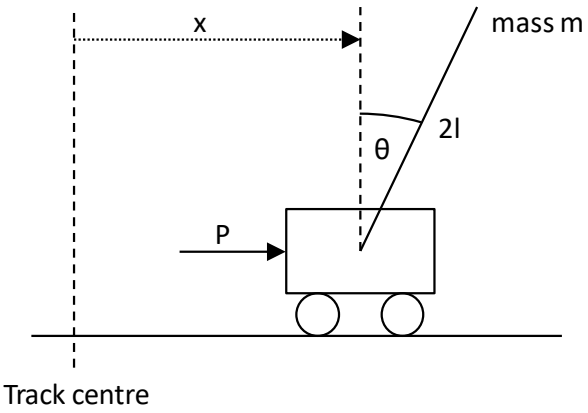
- $\boldsymbol{s}$ is the state-action represented by $\boldsymbol{s} = (s_1,\ s_2, \ldots\ s_k)^T$, and one of those s_j values represents the action value.
- $\boldsymbol{c_i} = \left({c_1}^i,\ {c_2}^i, \ldots\ {c_k}^i\right)^T$ with ${c_j}^i \in \{0, 1, \ldots, v\}$ and $i = 0, 1, \ldots, (v+1)^k$.

(The usual convention is to refer to a Fourier series of order *n*, but to avoid confusion with the *n* of the n-step Sarsa, I am referring instead to a v-order Fourier series.)

There are $(v+1)^k$ vectors in the form $\boldsymbol{c_i}$ (and the same number of weights) because there are $(v+1)^k$ possible permutations of the integers for use in those vectors. As reported in (Sutton & Barto, 2018) it is recommended that the learning rate α be adjusted for each x_i such that $\alpha_i = \alpha / \sqrt{({c_1}^i)^2 + ({c_2}^i)^2 + \ldots + ({c_k}^i)^2}$ except where this would result in division by zero, in which case $\alpha_i = \alpha$.

Initial design

Cart-pole dynamics



The physical characteristics of the cart-pole system I set simply by imagining a real-world example of a scale that might be seen in a laboratory, and which gave the learner a realistic prospect of reaching its goal.

Parameter	Value	Parameter	Value
P	5N	m	0.1kg
M	0.2kg	θ_0	Random -5° to +5°
L	6m	l	0.25m
$F_{c_{max}}$	0.01N	$M_{p_{max}}$	0.0001Nm
$\dot{x}_{F95}$	0.01ms^{-1}	$\dot{\theta}_{F95}$	1° s^{-1}

Although the logical fail conditions are that $|x| > L$ and $|\theta| > 90^o$, I changed the angle fail condition to $|\theta| > 60^o$ and added two other conditions of $\left|\dot{\theta}\right| > 360^o s^{-1}$ and $|\dot{x}| > 25ms^{-1}$. This seemed reasonable as if any of those conditions were met it was highly likely that the system had gone past a "point of no return" and the learner would soon fail. Either the pole was doomed to continue falling or the cart was moving too quickly to move the pole into an upright position before running out of track. The benefit of imposing such fail conditions was that an episode could be ended earlier, thus speeding up learning time.

Simulation algorithm fundamentals

Governing equations and initial conditions

As shown in Appendix B, the following equations can be derived:

$$\ddot{x} = \frac{P + ml\left(\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta\right) - F_p \cos\theta - F_c}{m + M} \quad (1)$$

$$\ddot{\theta} = \frac{g\sin\theta - \frac{F_p}{m} - \frac{\cos\theta\left[P + ml\dot{\theta}^2\sin\theta - F_p\cos\theta - F_c\right]}{m+M}}{l\left(1 - \frac{m\cos^2\theta}{m+M}\right)} \quad (2)$$

The initial conditions for each episode are:

$x_0 = \dot{x}_0 = \dot{\theta}_0 = 0$

$|\theta_0| \leq \frac{\pi\phi_0}{180}$ and $\theta_0 \neq 0$ where ϕ_0 is in degrees (e.g. 5°)

Simulation algorithm

At each time t, P is supplied as an input to the model, and the following steps are followed to calculate the new state of the model:

1. Calculate $\ddot{\theta}_t$ using (2).
2. Calculate $\ddot{x}_t$ using (1) and the result from step 1.
3. Calculate (using Euler integration):
   a) $\dot{\theta}_{t+1}$ using $\dot{\theta}_{t+1} \approx \dot{\theta}_t + \ddot{\theta}_t \delta t$
   b) θ_{t+1} using $\theta_{t+1} \approx \theta_t + \dot{\theta}_t \delta t$
   c) $\dot{x}_{t+1}$ using $\dot{x}_{t+1} \approx \dot{x}_t + \ddot{x}_t \delta t$
   d) x_{t+1} using $x_{t+1} \approx x_t + \dot{x}_t \delta t$

Initial design – Tile coder

I chose to normalize the state-action values to the range 0 to 1, so as to simplify the maths involved in the tile coder. The minimum and maximum (0 and 1) of each dimension were set to match the physical characteristics and failure modes of the system.

Note that the size of the feature vector $\boldsymbol{x}(S_t, A_t)$ is given by $n_t p^k$ where n_t is the number of tilings, p is the number of tiles per dimension in each tiling, and k is the number of dimensions in state-

action space. This means that increasing the tile coder resolution (by decreasing the magnitude of the fundamental unit, either by decreasing tile width or increasing the number of tilings) increases the number of features, and therefore the number of weights.

The key parameters for the tile coder were initially set as follows:

Parameter	**Value**
Tile width (w)	0.256
Number of tilings (n_t)	32
Fundamental unit (w/n_t)	0.1

Given that each dimension in state-action space ranges from 0 to 1, the chosen fundamental unit of 0.1 results in a "resolution" of 10 possible values in each dimension. This seemed likely to give the learner a sufficiently granular "picture" of the real world. Although a smaller fundamental unit might give the learner "finer" control, this would lead to a greater number of weights and therefore slower processing times.

Initial design – Fourier basis feature constructor

As shown earlier, a Fourier basis feature constructor of order *v* requires $(v+1)^k$ weights. (In this case *k* = 5 given the 5 dimensions of $x, \dot{x}, \theta, \dot{\theta}$, and the action.) This exponential relationship means that it is important to keep the order *v* as low as possible, or the updating of weights becomes impractical for an everyday laptop such as mine. Initially I set *v* to 2, and as shown later this proved to be sufficient.

Initial design – n-step Sarsa algorithm

Through experiment and choosing "typical values" as seen in (Sutton & Barto, 2018) I settled on the following initial parameter values of the n-step Sarsa algorithm.

Parameter	**Value**
n	2
α	When using a tile coder, as recommended by (Sutton & Barto, 2018) I set α to $1/10n_t$ where n_t is the number of tilings When using Fourier basis, set to 0.003125, only because this was the initial value applied using the tile coder (with 32 tilings).
Є	0.1
γ	1
Timestep δt	0.01s
Reward R_t	The length of the timestep in seconds (i.e., the time between each action taken by the learner). Thus the overall return from a single episode was the overall time of that episode.
Success stopping condition	Episode lasts 15 seconds. This was based on the assumption that if the pole could be balanced for 15 seconds, it more than likely meant the learner had "cracked" the problem and could keep it balance if not indefinitely, then for a considerable time.

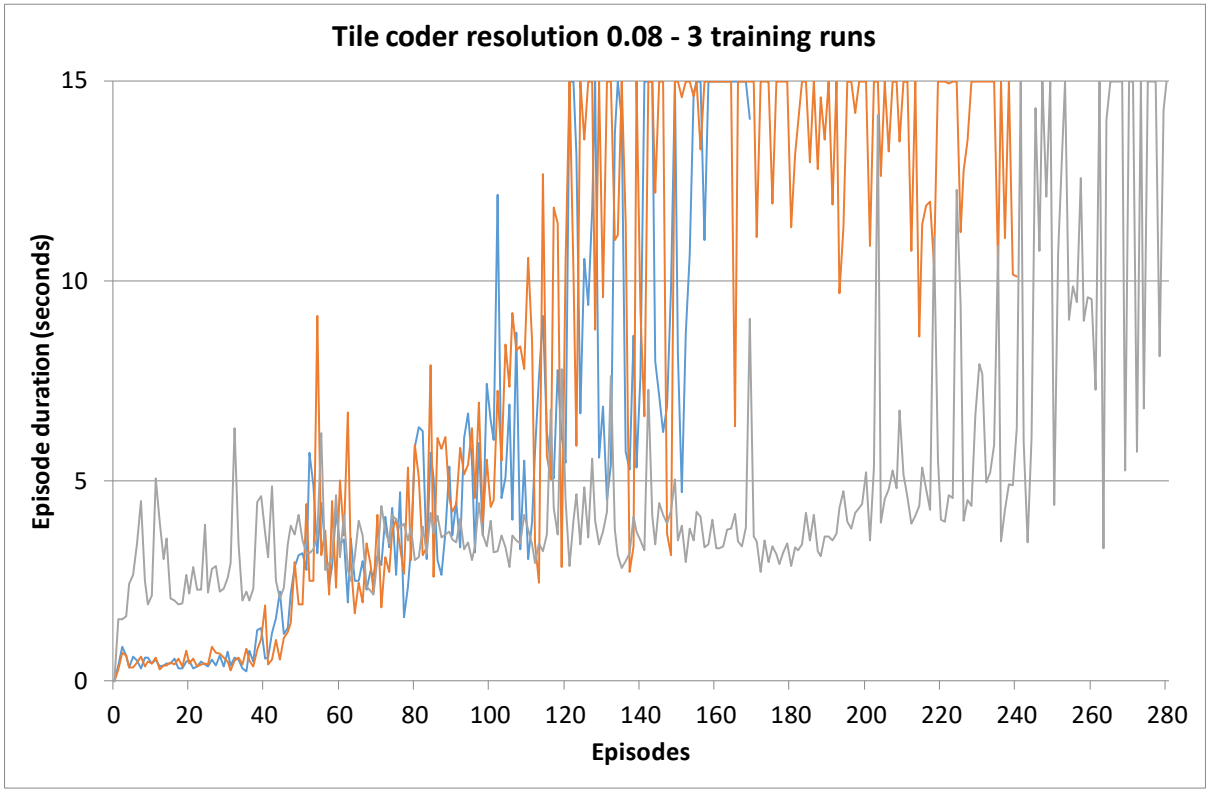
Implementation

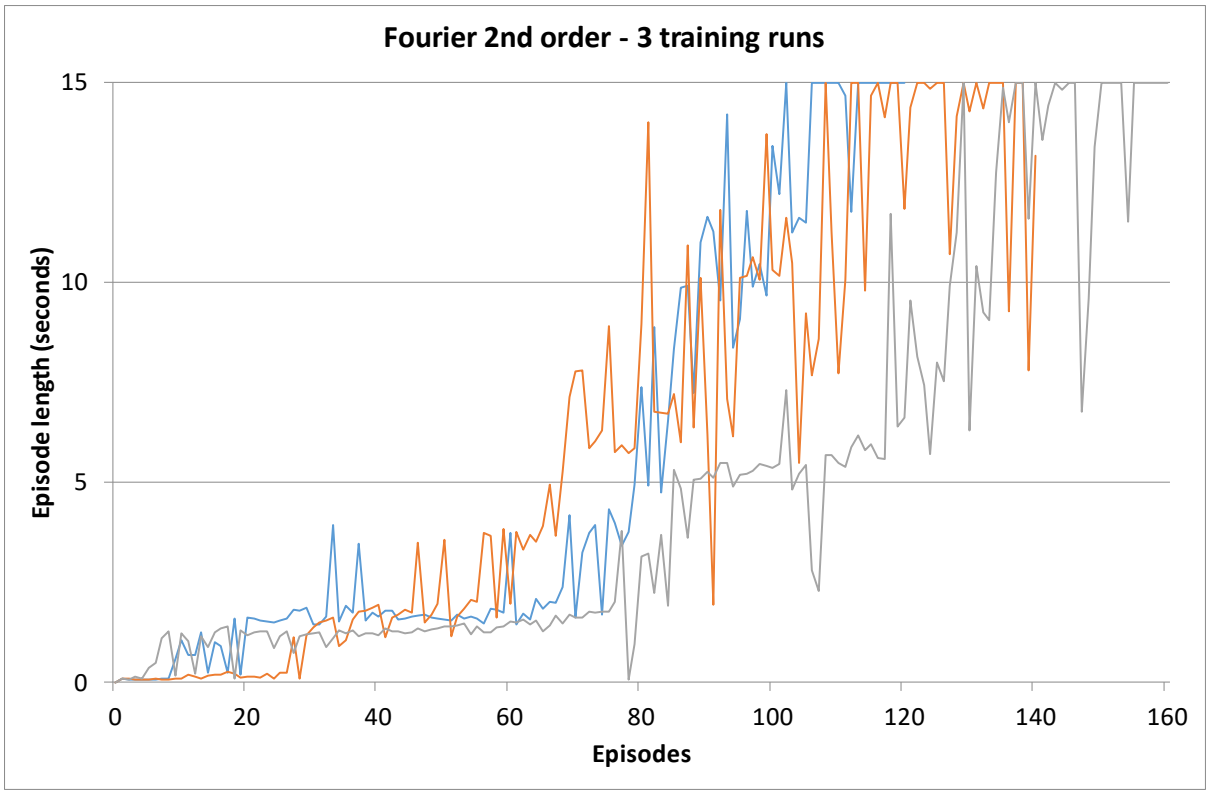
The learner, tile coder, Fourier basis feature constructor and model simulation were all created as classes in Excel VBA.

The learner could be configured to generate the complete history of an episode (i.e., the values of $x, \dot{x}, \theta, \dot{\theta}$ at each time interval) every Y episodes (e.g., every 20 episodes). I then used a different Excel file and macros which took such an output and displayed an animation of the simulation. This meant I could visualize how the learner was behaving at different points in its learning.

Initial results

Results are shown below for 6 different attempts to learn from zero knowledge (i.e., all weight values initially set to zero). The three attempts using a tile coder are shown in the first chart, and the three attempts using a Fourier basis feature constructor are shown in the second. All six attempts are "successful" in that the learner is able to keep the pole upright for 15 seconds reasonably consistently after a period of experimentation and learning. If Є had been reduced with increasing numbers of episodes, then the minor inconsistencies would eventually disappear.





Additional investigations

Once satisfied that the learner was performing as expected, I investigated the effect of varying:

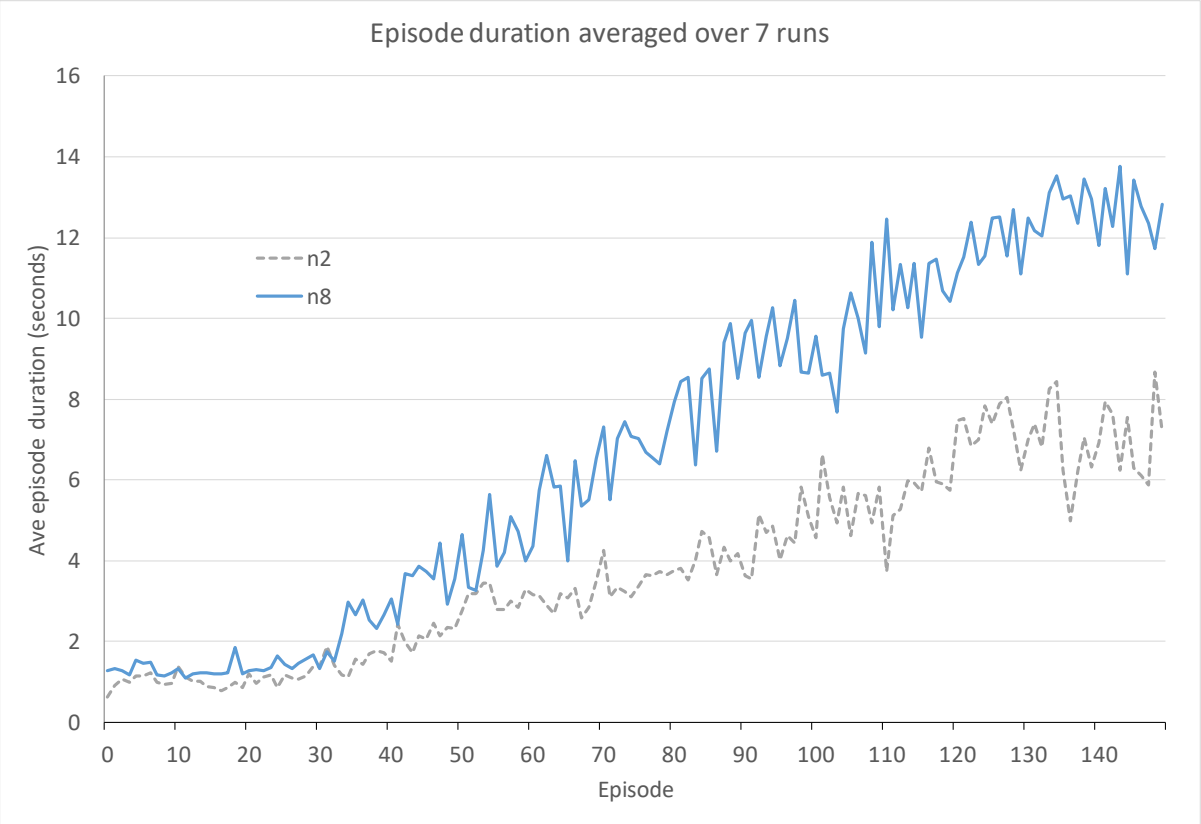
- The *n* value of the n-step Sarsa algorithm.
- Tile coder resolution.
- The order v of the Fourier basis feature constructor.

Each is described in turn.

Varying the *n* value of the n-step Sarsa Algorithm

(Sutton & Barto, 2018) state that increasing *n* will initially improve performance, but continued increases will then reduce performance. Looking at the values they used, it seemed like a value of *n*=8 should give noticeably better results than my initial choice of *n*=2.

The comparative results are shown in the chart below, and as suspected the new value out-performed the old.



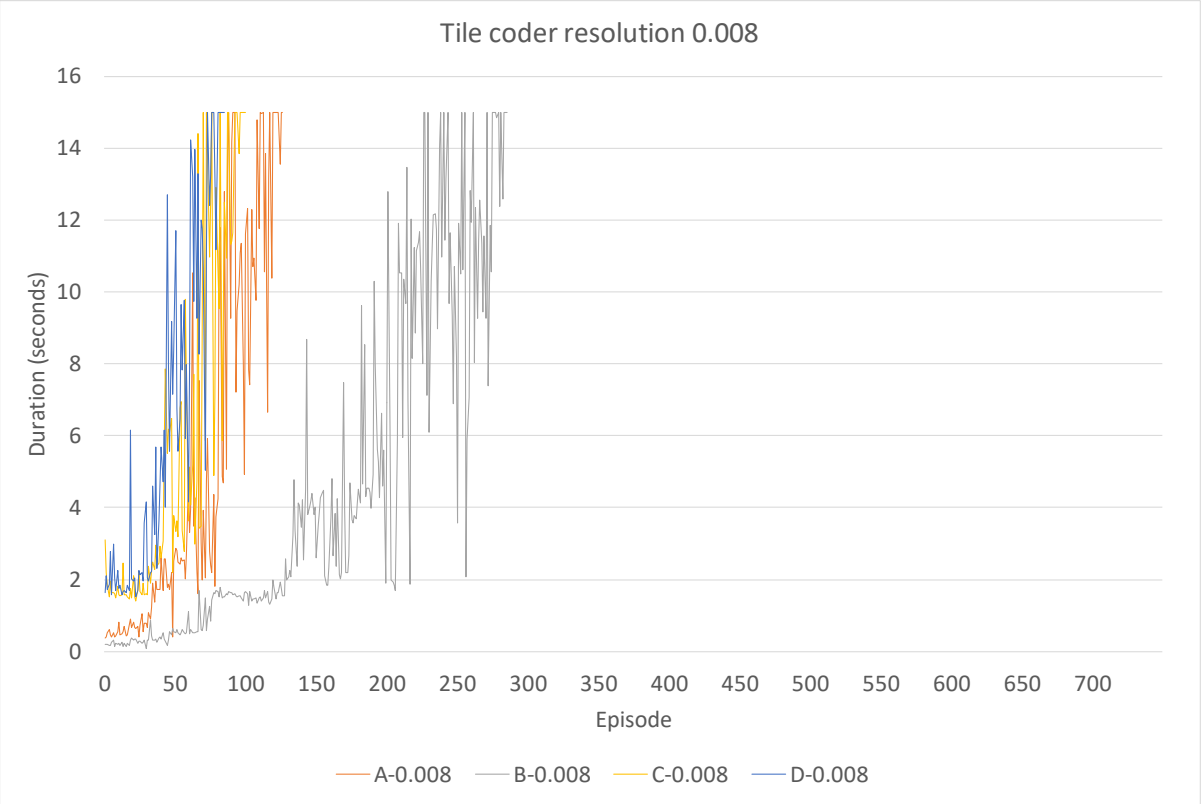
For this evaluation I used the tile coder with the same settings as my initial investigation. The only value varied was the *n* value.

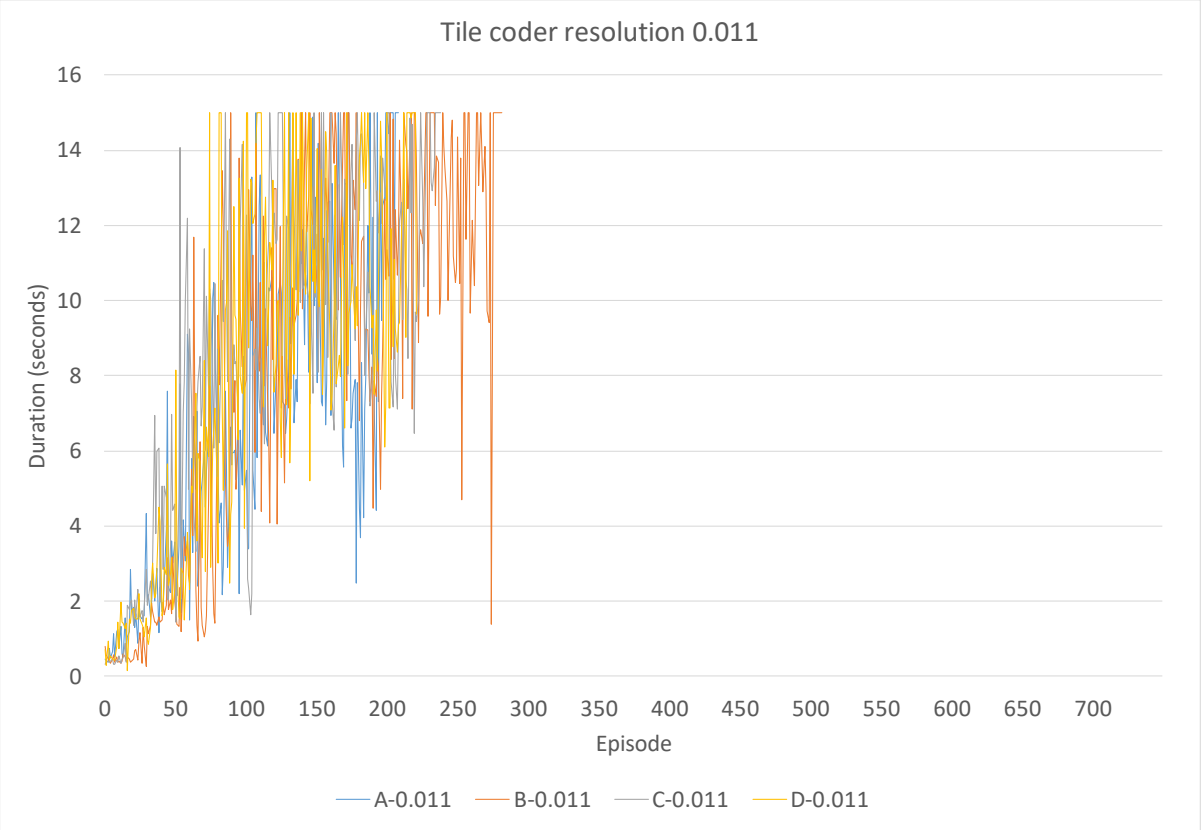
Varying the tile coder resolution

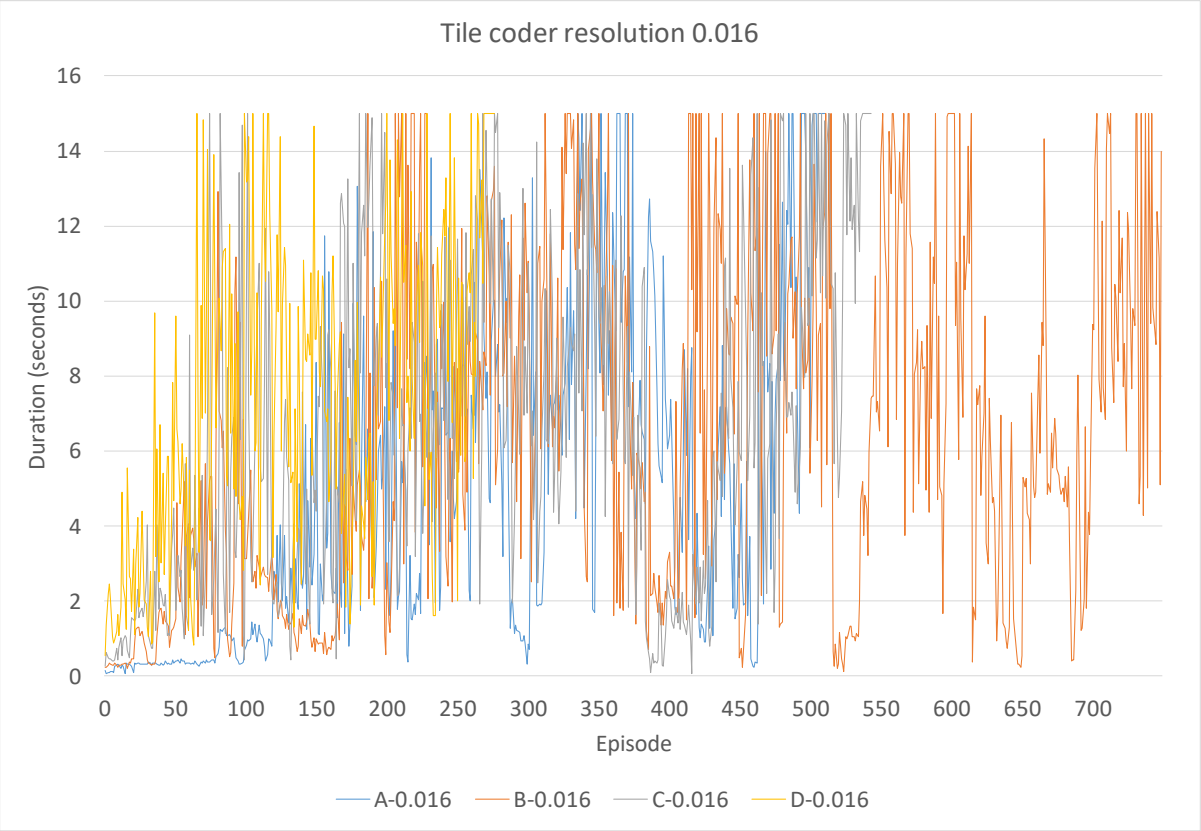
Since a tile coder resolution corresponding to a fundamental unit of 0.008 had already proven successful in the initial trials, I decided to decrease the resolution (i.e., increase the fundamental unit *u*) until the learner could no longer cope with the lack of granularity.

The results are shown below for *u*=0.008, 0.011 and 0.016. In each case 4 separate attempts were made, labelled A, B, C and D. Although the learner performs adequately at *u*=0.011 (albeit not as well as at *u*=0.008) it is unable to converge on a learnt solution at *u*=0.016.

As before, no values were changed from the initial trials apart from *u*.


Tile coder resolution 0.008
Duration (seconds)
Episode
A-0.008
B-0.008
C-0.008
D-0.008


Tile coder resolution 0.011
Duration (seconds)
Episode
A-0.011
B-0.011
C-0.011
D-0.011



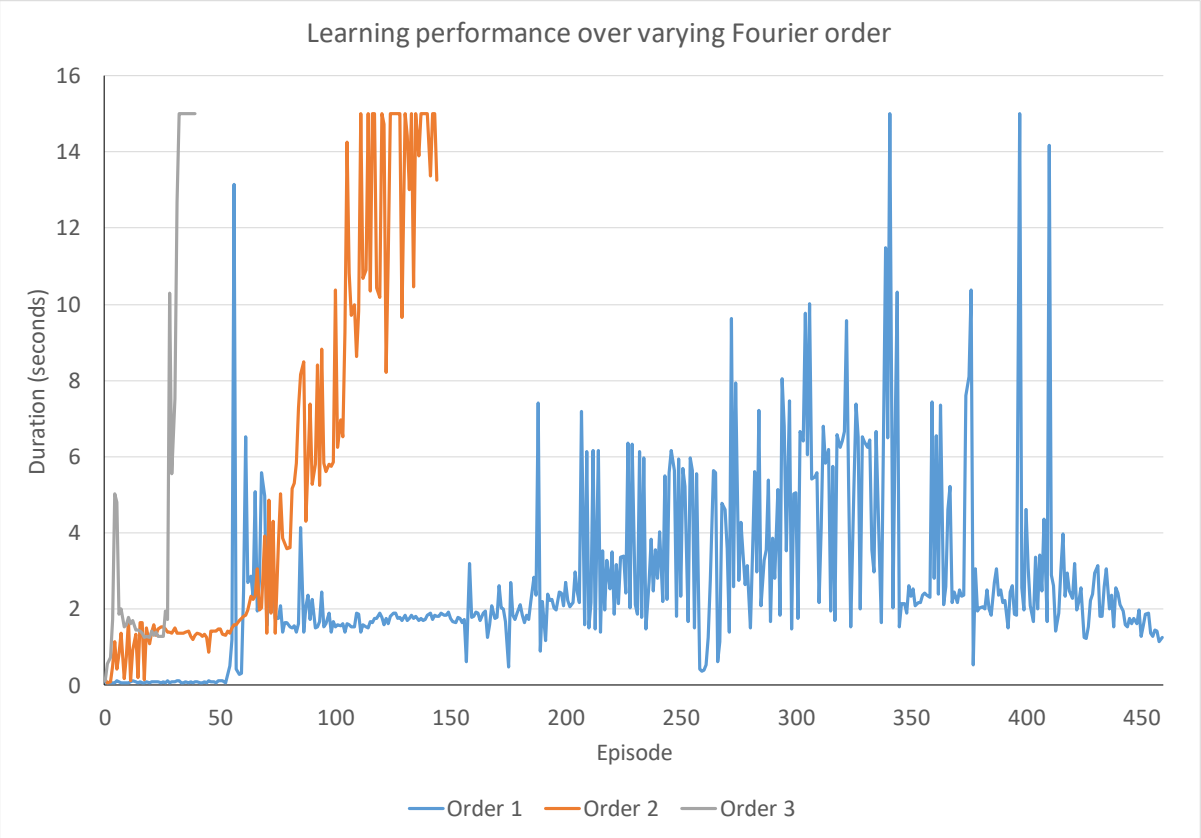
Fourier order analysis

The initial trials had been successful with a 2nd order Fourier basis feature constructor.

Increasing the order would likely lead to better performance in the sense that fewer episodes would be required to learn a solution, but the exponential increase in the number of weights would also greatly slow down processing speeds.

Decreasing the order would increase processing speeds, but might not give the learner sufficient granularity to be able to find a solution.

The results of 1st order, 2nd order and 3rd order are compared below. These results are nowhere near rigorous as they are only a single run in each case. The 3rd order run took so long to run on my standard laptop that it was impractical to do several runs. However, the results do show the expected pattern.



Bibliography

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). The MIT Press.

Appendix A – Approach to generating tile coder and calculating features

Algorithms

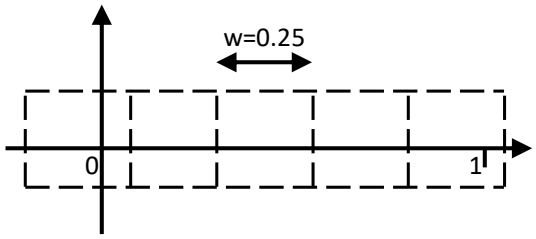
Tile coder layout

The algorithm to generate the layout is essentially an algorithm to calculate the "origins" of each tiling relative to the origin of the overall state-action space:

1. Create the *baseDisplacementVector* $\boldsymbol{h} = (h_0, h_1, \dots, h_{k-1})^T$ where $h_i = -(2i+1)u$, (where u is the fundamental resolution unit). This is the vector recommended by (Sutton & Barto, 2018) to avoid non-uniform generalisation.
2. Initialise the list *tilingOrigins* to an empty list.
3. Create the tiling origin for the first tiling as $(0, 0, \dots, 0)^T$ with k elements, and append it to *tilingOrigins*.
4. For each of the remaining tilings in turn:
   a) Initialise an empty list *origin*.
   b) For each state-action space dimension i:
      i. Set *coord* to the sum of the corresponding coordinate in the previous tiling and the corresponding coordinate in *baseDisplacementVector*.
      ii. If the calculated value of *coord* means that it would be outside the desired first tile location (i.e., *-coord* ≥ *w*, the tile width), then set *coord* to *coord + w*.
      iii. Append origin to *tilingOrigins*.

Calculating features

This algorithm must work out in which tile a given state-action appears. First, a tiling tile number is calculated. This assumes that for the tiling in question, the tiles are numbered starting with 0 for the tile at the tiling origin, running up to $p^k - 1$ where *p* is the number of tiles per dimension. *p* is given by *p = roundUp(1/w, 0) + 1* where *roundup(a, b)* rounds *a* up to the nearest number with *b* decimal places. This is illustrated below for one dimension in a 2-dimensional state-action space, where the state-action space is bounded by the values 0 and 1, and the tile width is 0.25. Since the tilings can be offset, there are 5 tiles required to cover the state-action dimension.

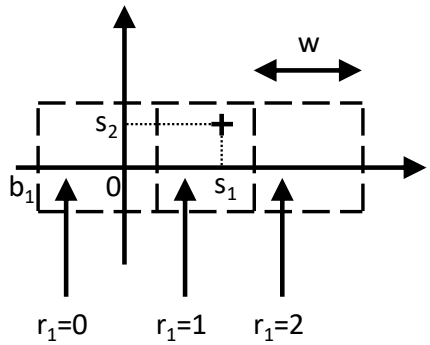


To make the calculations easier, my approach was to normalise all state-action value to the range 0 to 1 in each dimension.

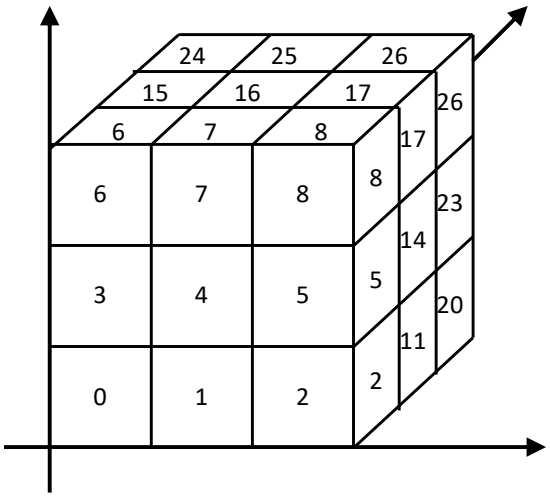
Now, given a state-action point of $(s_0, s_1, \dots, s_{k-1})^T$, the corresponding tile "coordinate" in a given dimension i is given by:

$r_i = int[(s_i - b_i)/w]$ where b_i is the coordinate in the tiling origin vector

This is illustrated in the diagram below.



The process to convert a tile's "coordinate" into a tile number for that tiling is based on constructing the tiling tile numbers as illustrated below for a 3-dimensional state-action space.



The tiling number for a given tiling *m* is given by

$T_m = r_0 + pr_1 + p^2 r_2 + \ldots + p^{k-1} r_{k-1}$

where *p* is the number of tiles per dimension, *i* is the dimension (starting at 0), and r_i is the tile coordinate in the i^{th} dimension.

The overall tile number is then given by:

$T = mp^k + T_m$ where the first tiling is *m*=0

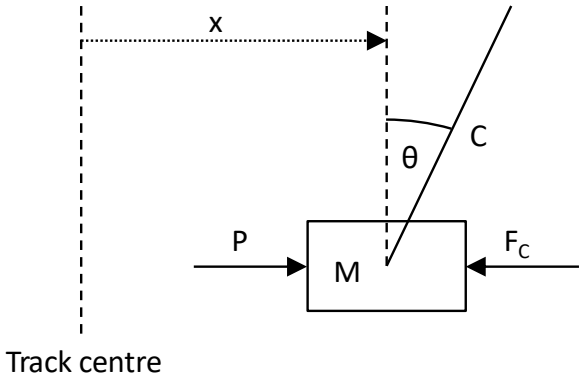
Appendix B – Derivation of cart-pole dynamics equations

The dynamics of the cart and pole are modelled as two separate but related systems.

Cart dynamics

As illustrated below, the cart is modelled as a point mass M subject to three forces:

- P – the force applied by the learner.
- F_c – the frictional force generated by the cart's motion along the track.
- C – the compressive force in the pole.



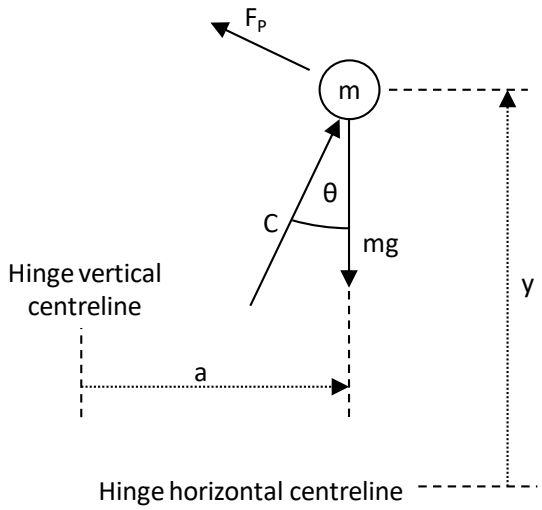
The governing equation is therefore:

$$P - C\sin\theta - F_c = M\ddot{x} \qquad (1)$$

Pole dynamics

The pole is modelled as a point mass m at the end of a mass-less pole of length I. As shown below, the point mass is also subject to three forces:

- mg – the gravitational force where g=9.818 ms^{-2}.
- F – the frictional force due to the torque of the rotational friction at the pole hinge.
- C – the compressive force in the pole.



The governing equations are therefore:

$C \sin\theta - F_p \cos\theta = m\ddot{a}$ (2)

$C \cos\theta + F_p \sin\theta - mg = m\ddot{y}$ (3)

Frictional forces

The frictional forces are modelled as being proportional to velocity / rotational velocity:

$F_c = F_{c_{max}} \tanh k_c\dot{x}$ where k_c is a constant

$F_p = \frac{M_{p_{max}}}{l} \tanh(k_p\dot{\theta})$ where k_p is a constant

The constants k_c and k_p can be calculated by stating that the frictional force must be a given percentage of its maximum at a certain velocity. For example, if it is desired that F_c be 95% of its maximum at a given velocity $\dot{x}_{95F}$, then:

$0.95 = \tanh(k_c\dot{x}_{95F})$

$\Rightarrow k_c = (\tanh^{-1}(0.95))/\dot{x}_{95F}$

Overall system dynamics

Using the governing equations above, it is possible to derive the equations necessary to model the system using Euler integration, as follows.

Since $a = x + l\sin\theta$, $\dot{a} = \dot{x} + l\,\dot{\theta}\sin\theta$:

$\ddot{a} = \ddot{x} + \left(-l\dot{\theta}^2 \sin\theta + l\ddot{\theta}\cos\theta\right)$

$= \ddot{x} - l\left(\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta\right)$

Substitute in (2):

$C \sin\theta = m\ddot{x} - ml\left(\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta\right) + F_p \cos\theta$ (4)

Substitute in (1):

$$P + ml\left(\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta\right) - F_p\cos\theta - F_c - m\ddot{x} = M\ddot{x}$$

$$\Rightarrow \ddot{x} = \frac{P + ml\left(\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta\right) - F_p\cos\theta - F_c}{m + M} \quad (5)$$

Since $y = l\cos\theta$, $\dot{y} = -l\dot{\theta}\cos\theta$:

$$\ddot{y} = -l\dot{\theta}^2\cos\theta - l\ddot{\theta}\sin\theta$$

$$= -l\left(\dot{\theta}^2\cos\theta + \ddot{\theta}\sin\theta\right)$$

Substitute in (3):

$$C\cos\theta + F_p\sin\theta - mg = -ml\left(\dot{\theta}^2\cos\theta + \ddot{\theta}\sin\theta\right)$$

$$\Rightarrow C\sin\theta\cos\theta + F_p\sin^2\theta - mg\sin\theta = -ml\sin\theta\left(\dot{\theta}^2\cos\theta + \ddot{\theta}\sin\theta\right)$$

Substitute for $C\sin\theta$ using (4):

$$m\ddot{x}\cos\theta - ml\cos\theta\left(\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta\right) + F_p\cos^2\theta + F_p\sin^2\theta - mg\sin\theta$$

$$= -ml\sin\theta\left(\dot{\theta}^2\cos\theta + \ddot{\theta}\sin\theta\right)$$

$$\Rightarrow m\ddot{x}\cos\theta - ml\dot{\theta}^2\sin\theta\cos\theta + ml\ddot{\theta}\cos^2\theta + F_p - mg\sin\theta = -ml\dot{\theta}^2\sin\theta\cos\theta - ml\ddot{\theta}\sin^2\theta$$

$$\Rightarrow m\ddot{x}\cos\theta + ml\ddot{\theta} + F_p - mg\sin\theta = 0$$

$$\Rightarrow \ddot{\theta} = \frac{mg\sin\theta - F_p - m\ddot{x}\cos\theta}{ml} \quad (6)$$

Substitute (5) in (6):

$$l\ddot{\theta} = g\sin\theta - \ddot{x}\cos\theta - {}^{F_p}/_{m}$$

$$\Rightarrow l\ddot{\theta} = g\sin\theta - \frac{F_p}{m} - \frac{\cos\theta\left[P + ml\left(\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta\right) - F_p\cos\theta - F_c\right]}{m + M}$$

$$\Rightarrow l\ddot{\theta} = g\sin\theta - \frac{F_p}{m} - \frac{\cos\theta\left[P + ml\dot{\theta}^2\sin\theta - F_p\cos\theta - F_c\right]}{m + M} + \frac{ml\ddot{\theta}\cos^2\theta}{m + M}$$

$$\Rightarrow l\ddot{\theta}\left(1 - \frac{m\cos^2\theta}{m + M}\right) = g\sin\theta - \frac{F_p}{m} - \frac{\cos\theta\left[P + ml\dot{\theta}^2\sin\theta - F_p\cos\theta - F_c\right]}{m + M}$$

$$\Rightarrow \ddot{\theta} = \frac{g\sin\theta - \frac{F_p}{m} - \frac{\cos\theta\left[P + ml\dot{\theta}^2\sin\theta - F_p\cos\theta - F_c\right]}{m + M}}{l\left(1 - \frac{m\cos^2\theta}{m + M}\right)} \quad (7)$$