Technical Milestone Report

Candidate Number: 5590E

January 17, 2019

Summary

Technical Summary

1 Introduction

AlphaGo Zero was a revolutionary AI that used reinforcement learning from self play to achieve superhuman performance in the game of go, winning 100–0 against the previously published, champion-defeating AlphaGo. AlphaZero is a more general algorithm that can achieve superhuman performance in many games, however it uses many of the same algorithms as AlphaGo Zero.

One of the major benefits of AlphaZero is that it only requires to be able to simulate the environment. It does not need to be told how to win, or to be given an exact model of the system dynamics, $p(s_{t+1}, return_{t+1}|s_t, a_t)$, as this can be learnt through self-play. Amazingly,this self play often leads to novel solutions to problems, as shown by 'move 37' in a game of Go against the reigning world champion, Lee Sedol.

Its aim is to learn a policy, $p(\boldsymbol{a}|\boldsymbol{s})$ (a $p.m.f.$ over actions given a state, where the probabilities are proportional to the expected outcome for each action). It achieves this by playing against itself - picking the moves that it thinks will do best, whilst inconveniencing the opponent the most (which is itself). Randomness in the moves picked gives the variability within games and allows it to actually win. Two key algorithms have been pioneered to achieve this: A convolutional neural network (cnn) and 'Monte-Carlo Tree Search' (MCTS).

AlphaGo Zero and AlphaZero use two different methods of policy improvement. For both, the weights and biases are randomly initialized (so there should be a 50% chance of winning). Then a batch of games/episodes are played via self-play. With AlphaGo Zero, the self-play games are generated by the best player from all previous iterations and then the new player was compared with the best player; whereas AlphaZero just maintains a single neural network that is updated continually: self-play games are generated by using the latest parameters of the neural network. [?]

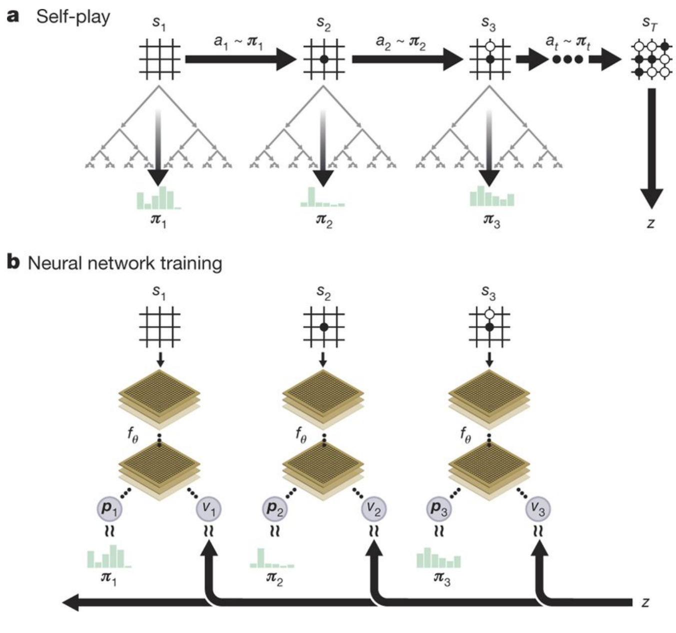


Figure 1: A schematic showing how self-play and policy training are performed. Taken from [?].

Figure ??a shows how self-play is performed in AlphaGo Zero. An episode/game, $\{s_1, ..., s_T\}$ is played against itself. For each state, s_t, a Monte-Carlo Tree Search is performed, guided by the current policy f_θ. The MCTS outputs an improved action-probability, $\boldsymbol{\pi}_t = [\hat{p}(a_1|s_t), \hat{p}(a_2|s_t), ..., \hat{p}(a_N|s_t)]$. The next move is then selected by sampling from $\boldsymbol{\pi}_t$. The final player of the game is then given a score of 1 or -1 if they win or lose respectively. The game score, z, is then appended to each state-action pair depending on which who the current player was on that move $[s_t, \boldsymbol{\pi}_t, z]$.

Figure ??b depicts how the policy is trained. The state, s_t, is taken as input and is passed through a neural network with parameters θ. The network outputs an action-probability, $\boldsymbol{p}_t$, and a state value - the estimated probability of the current player winning the game. The neural network is trained to more closely match the MCTS-informed action-probabilities π_t, and the predicted winner (state-value), v_t to the actual winner, z.

The loss for the neural network is given by:

$$\mathcal{L} = (z - v)^2 - \boldsymbol{\pi} \cdot log(\boldsymbol{p}) + c||\theta||^2 \tag{1}$$

Figure **??**a: A MCTS is performed at each step of an episode. The state at which the tree search starts then becomes the root state, s_{root}. From the root state, the tree search can move to an edge (s, a) by selecting an action, a. Each edge stores a prior action-probability output by the policy, p(s, a); a visit count, N(s, a); and an action-value, Q(s, a), which is the value of the state that action a will result in. Actions are selected by maximising an the action-value plus an upper confidence bound:

$$a_{selected} = \underset{\forall a}{argmax}\Big\{Q(s,a) + c \cdot \frac{p(s,a)}{1 + N(s,a)}\Big\} \tag{2}$$

Where c is a constant of magnitude ~ 1.

Figure **??**b: Once a leaf node (N(s, a) = 0) is reached, the neural network is evaluated at that state: $f_\theta(s) = (p(s, \cdot), V(s))$. The action-probability and state-value are stored for the leaf state.

Figure **??**c: The action-values, Q, are calculated as the mean of state-values in the subtree below that action. The state-value are then calculated as the mean of all the action-values branching from that state, and so on.

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{s_{t+1}|s_t,a_t} v(s_{t+1}) \tag{3}$$

Figure **??**d: After a set number searches, the MCTS-improved action-probabilities $\boldsymbol{\pi} = p(\boldsymbol{a}|s_{root})$ are returned, $\propto N^{1/\tau}$, where N is the visit count of each move from the root state and τ controls the sparseness of the probability mass function ($\{\tau = 0\} \rightarrow argmax\{N(s,a)\}$).

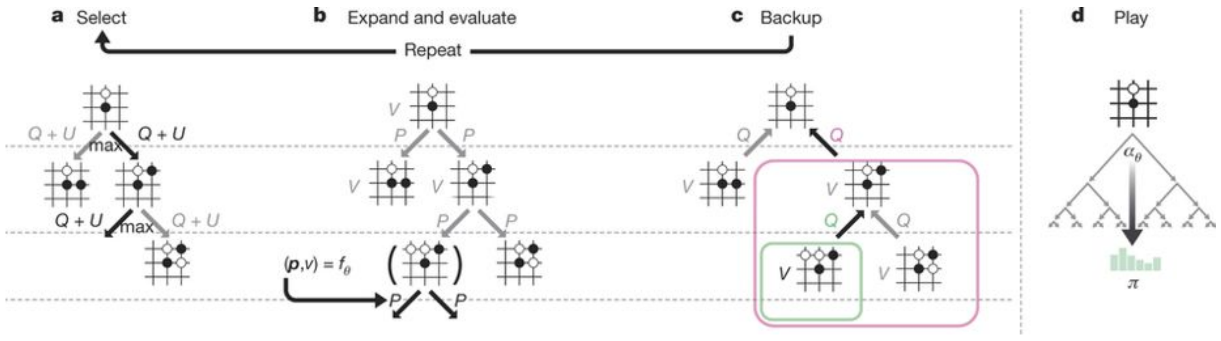


Figure 2: A schematic outlining the steps involved in a monte-carlo tree search. The figure is taken from [?].

Control theory is the study of how to get optimal corrective behaviour for dynamical systems. Reinforcement learning has recently found success in control problems such as atari-games and robotics [**?**]. A common theme with control problems is that they are easy to simulate, but hard to find the dynamics for. AlphaZero is an incredibly powerful RL algorithm that requires simulation but does not know the dynamics: therefore, it is a good candidate for a general purpose controller.

There are a few problems in adapting AlphaZero to a control problem, namely:

- How do we represent a continuous state as a 2D state?
- How do we change the value function to reflect a non-stationary and continuous problem? We cannot use whether we win or lose as the value for the end of the game.
- How do we implement self-play for problems that already have non-intelligent adversaries such as gravity?

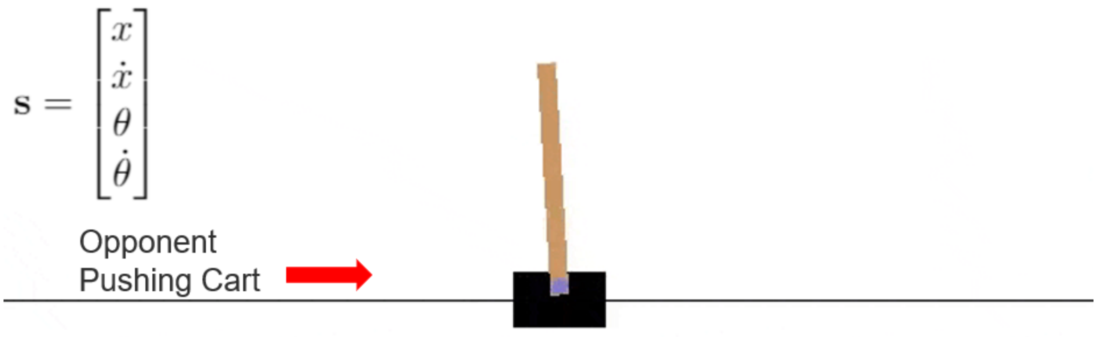
The purpose of this project is to try to find an answer to these three questions and therefore show that AlphaZero can be used as a powerful general-purpose control algorithm.

2 Method

Currently, a 2D state representation and a value function have been chosen and implemented, an adversary other than gravity has not been implemented yet.

2.1 OpenAI's Gym Environment

OpenAI's gym is a toolkit for developing reinforcement learning algorithms. They provide 'environments' with common interfaces to allow the writing of general algorithms. Initially, the CartPole environment is being used as it is the simplest classical control problem, see figure ??.



$\mathbf{s} = \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}$
Opponent
Pushing Cart

Figure 3: The OpenAI gym CartPole environment. The classical state representation is shown in the top left. Actions by the player and the adversary are taken as an impulse to the left or right.

The cart is defined as having fallen over, or 'done', if the angle from upright exceeds $12^o(\theta_{max})$ or the cart wanders $> 2.4(x_{max})$ units from the origin. We also define steps_beyonds_done as the number of steps that we continue to calculate states for after it has fallen over. For each step/impulse, the 2D state is calculated and a state_loss is calculated as:

$$\mathcal{L} = -\frac{1}{2}\left[\left(\frac{x_t}{x_{max}}\right)^2 + \left(\frac{\theta_t}{\theta_{max}}\right)^2\right] \tag{4}$$

Thus ensuring that $0 \geq \mathcal{L} \geq -1$.

2.2 2D State Representation

The state is a histogrammed and discounted function of previous state positions and angles, i.e. $NewState = Binned\ Current\ Position + \gamma * PreviousState$, where γ is a discounting factor, currently set at 0.7. The numpy library in python has a function histogram2d which allows the binning of two-dimensional arrays.

A 2D representation like this allows us to use a convolutional neural network,which has the benefit of various transformation invariances - these are particularly useful for cartPole since it is highly symmetric.

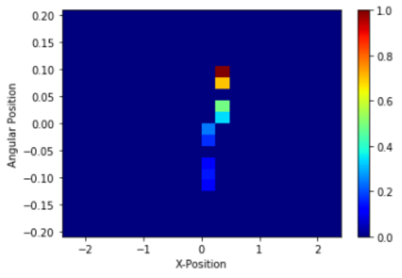


Figure 4: An example of a 2D state representation where there are 20 bins and 17 random actions have been taken.

2.3 The Value Function

A value function is computed after an episode has completed as the discounted future losses at each state with the constraint that $\gamma^{(n-1)} < \frac{1}{20}$, where $\frac{1}{20}$ was chosen as it is a standard factor for insignificance. Since steps_beyonds_done (n) must be defined in the CartPoleWrapper class, this is a constant, and therefore γ is calculated as $\gamma < \frac{1}{20}^{\frac{1}{(n-1)}}$. Algorithm ?? shows the procedure for calculating this. The discounted value can be written as:

$$v_t = \frac{\sum_{t'=t,\tau=0}^{t+k,k} \gamma^{\tau} \mathcal{L}_{t'}}{\sum_{\tau}^{k} \gamma_{\tau}}, \qquad \text{where } \gamma^k < \frac{1}{20} \tag{5}$$

Algorithm 1 Calculate State Values

```
1: procedure VALUE_FUNCTION(state_losses)
2:     Constants: $\gamma$, k                                  ▷ discount factor, steps_beyonds_done
3:     Initialise: values ← [ ]
4:     for step_idx := 0 to (length(state_losses) - k) do
5:         state_value ← 0
6:         discount ← 1
7:         for idx := step_idx+1 to (step_idx+k) do
8:             value ← value + discount×state_losses[idx]
9:             discount ← $\gamma$×discount
10:        values.append($\frac{\text{value}}{\sum\gamma}$)
11:    return values                            ▷ values is k elements shorter than state_losses
```

2.4 Program Structure

An outline of the program structure is shown in figure **??**. The green boxes are implemented as described in [**?**] in the introduction with the exceptions that the neural network predicts the value as defined above, not the expected outcome, and episodes are currently compared simply as $mean$(challenger losses) $>$ $0.95 * mean$(currrent losses).

3 Results

- Graphs of statistics for each policy iteration, eg, errors over time for each nnet
- Performance against random and a greedy algorithm (make the moves that move x pos closer to 0)
- Graphs of steps taken vs expected steps left
- Graphs of return vs time
- If I have time, ill compare different adversaries: vs greedy, vs itself with 1/2 power etc

4 Discussion

- Explain the graphs above
- Talk about problems?
- Which adversary was the best
- Which method learnt the quickest
- Talk about which representation of 2D state is the best? I havent done any of the others...

5 Future Plan

The first task to complete moving forwards is to implement an intelligent adversary. For the CartPole problem, it can't be an equal adversary since we are already fighting against gravity. Some possible adversaries are:

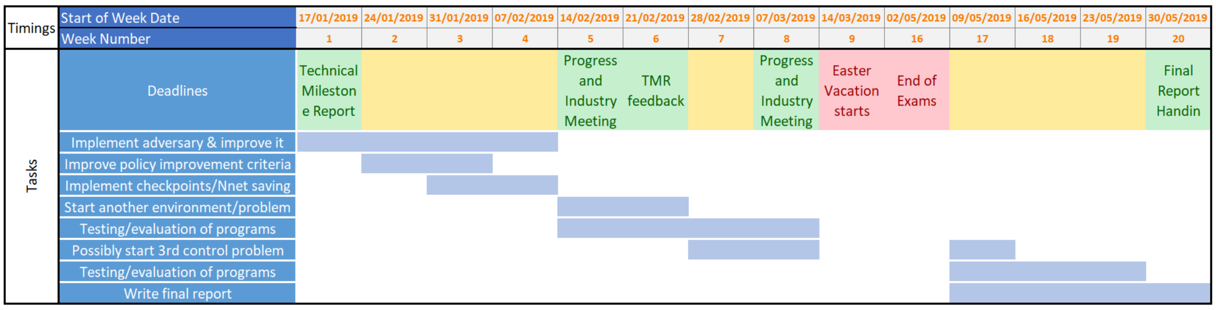
- Alternate steps and only give the adversary 0.5F for each push.
- The adversary only plays every 3 steps.

- A more 'environmental' adversary such as changing gravity or damping.

As a starting point, the adversary will act in the same plane as the force and act ever n steps (not necessarily 3 since this may be too strong).

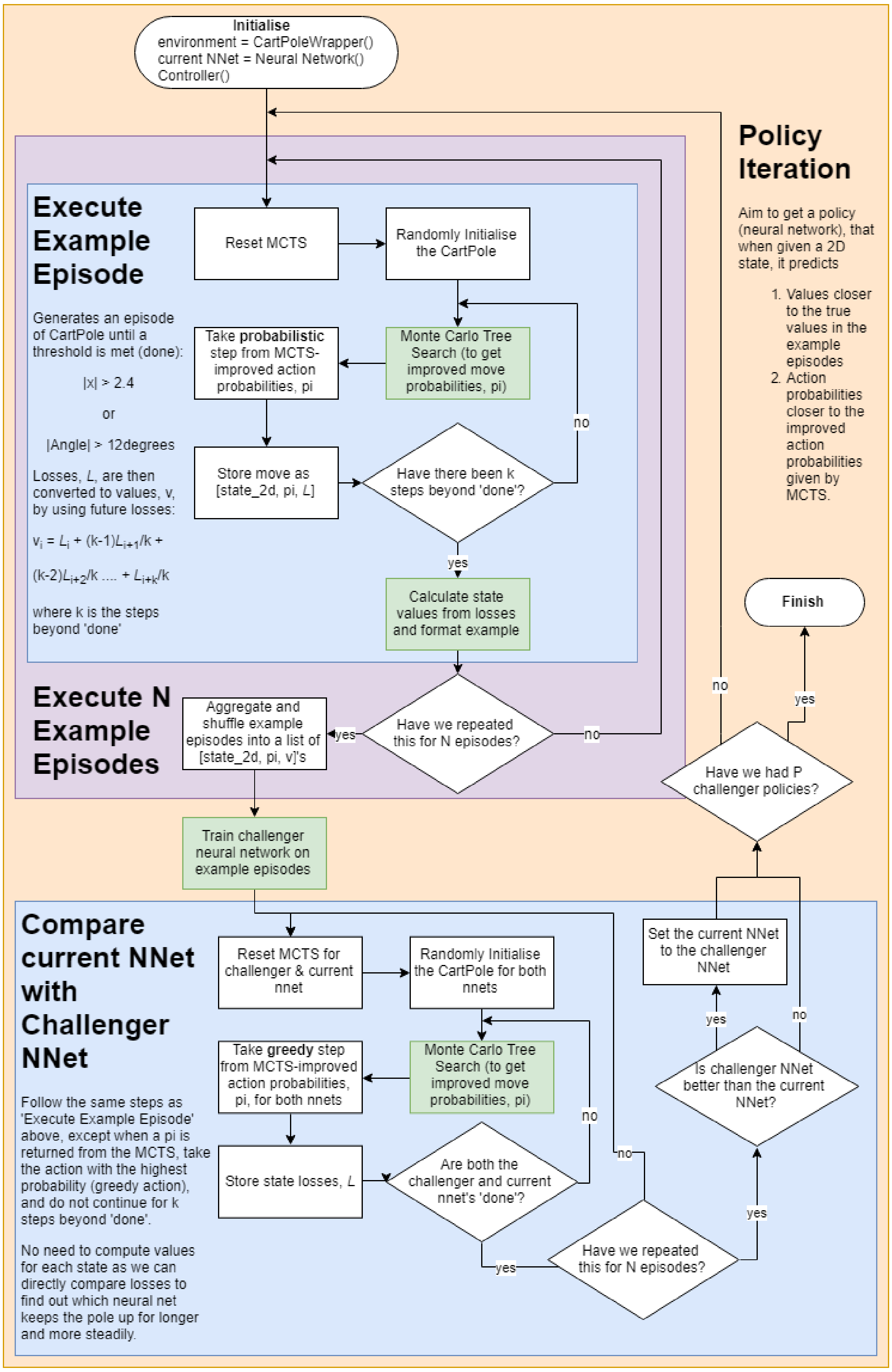
One of the difficulties of self-play with a control problem is determining how 'good' the policy is compared to other policies. Currently, the AlphaGo Zero style of policy iteration is being used (i.e. comparing a challenger policy with the current best), however, by introducing a

- Expansion to other problems
- Timeline moving forward
- Improvement of Nnet? Improvement of binning?



Timings	Start of Week Date	17/01/2019	24/01/2019	31/01/2019	07/02/2019	14/02/2019	21/02/2019	28/02/2019	07/03/2019	14/03/2019	02/05/2019	09/05/2019	16/05/2019	23/05/2019	30/05/2019
	Week Number	1	2	3	4	5	6	7	8	9	16	17	18	19	20
Tasks	Deadlines	Technical Milestone Report				Progress and Industry Meeting	TMR feedback		Progress and Industry Meeting	Easter Vacation starts	End of Exams				Final Report Handin
	Implement adversary & improve it	■	■	■	■										
	Improve policy improvement criteria		■	■											
	Implement checkpoints/Nnet saving			■	■										
	Start another environment/problem					■	■								
	Testing/evaluation of programs					■	■	■	■						
	Possibly start 3rd control problem							■	■			■			
	Testing/evaluation of programs											■	■	■	
	Write final report											■	■	■	■

Figure 5: A Gantt Chart of the proposed timeline for lent and easter term.


Initialise
environment = CartPoleWrapper()
current NNet = Neural Network()
Controller()
Policy Iteration
Execute Example Episode
Reset MCTS
Randomly Initialise the CartPole
Take **probabilistic** step from MCTS-improved action probabilities, pi
Monte Carlo Tree Search (to get improved move probabilities, pi)
Store move as [state_2d, pi, *L*]
Have there been k steps beyond 'done'?
no
yes
Calculate state values from losses and format example
Execute N Example Episodes
Aggregate and shuffle example episodes into a list of [state_2d, pi, v]'s
Have we repeated this for N episodes?
yes
no
Train challenger neural network on example episodes
Finish
yes
no
Have we had P challenger policies?
Compare current NNet with Challenger NNet
Reset MCTS for challenger & current nnet
Randomly Initialise the CartPole for both nnets
Take **greedy** step from MCTS-improved action probabilities, pi, for both nnets
Monte Carlo Tree Search (to get improved move probabilities, pi)
Store state losses, *L*
Are both the challenger and current nnet's 'done'?
no
yes
Have we repeated this for N episodes?
no
yes
Set the current NNet to the challenger NNet
yes
no
Is challenger NNet better than the current NNet?

Aim to get a policy (neural network), that when given a 2D state, it predicts

1. Values closer to the true values in the example episodes
2. Action probabilities closer to the improved action probabilities given by MCTS.

Generates an episode of CartPole until a threshold is met (done):

$|x| > 2.4$

or

$|\text{Angle}| > 12\text{degrees}$

Losses, L, are then converted to values, v, by using future losses:

$v_i = L_i + (k-1)L_{i+1}/k + (k-2)L_{i+2}/k \,....\, + L_{i+k}/k$

where k is the steps beyond 'done'

Follow the same steps as 'Execute Example Episode' above, except when a pi is returned from the MCTS, take the action with the highest probability (greedy action), and do not continue for k steps beyond 'done'.

No need to compute values for each state as we can directly compare losses to find out which neural net keeps the pole up for longer and more steadily.

Figure 6: A flowchart showing the control flow of the current program. Green boxes depict more complex processes with further explanation in the main text.