Evolution of neurocontrollers for active control and behaviour modulation of tethered airfoils using genetic algorithms

729G11 Course Project by: Albin Sundin
Contents
Introduction .............................................................................................................................................. 2
The Microbial Genetic Algorithm ........................................................................................................... 3
The Neural Networks ................................................................................................................................. 3
Kite Physics Simulation .............................................................................................................................. 5
Results of Evolved Neurocontrollers for Flight Trajectory Control ...................................................... 6
Behaviour Modulation ............................................................................................................................... 6
Results of Evolved Neurocontrollers for Behaviour Modulation ......................................................... 7
Robustness of Evolved Neurocontrollers and Future Work ...................................................................... 8
Implementing the Microbial Genetic Algorithm using Python ............................................................... 8
References ................................................................................................................................................ 11
Introduction

With renewable energy being an important contemporary issue, an interesting possibility has somewhat unexpectedly presented itself from the area of artificial intelligence. Kite energy systems currently enjoy a window of opportunity due to kite development for other applications now potentially allowing the reliable production of large scale, low cost renewable energy. Previous studies have shown that “closed loop flight trajectories in the form of a figure eight are an optimal solution in terms of power generation”. This solution has the “additional advantage of not requiring a swivel mechanism at ground level to prevent line crossovers” (Furey & Harvey, 2007). But it does require active control of the kite to maintain the flight trajectory, and “if the lines are being reeled out from a spool coupled to a dynamo, the lines will need occasional retraction, at which point the lines force will need to be minimised without stalling the kite by reducing kite speed and placing the kite perpendicular to the wind” (ibid.)

Kite wind power is a relatively new and promising field in sustainable energy. Several research groups, commercial and academic, are currently working on different techniques for using tethered airfoils to produce electricity. One approach in particular makes innovative use of AI techniques to produce software controllers for kite wind power. The principle behind this technique is to use biologically inspired methods from artificial intelligence in order to produce controllers that modulate kite flight trajectory to generate the most amount of power while pulling out a spool powering an electricity generating dynamo.

Furey & Harvey (2007) have shown in simulations that the use evolutionary robotics techniques produce robust kite controllers that fly the kite in stable lemniscate trajectories, shown previously to be an optimal flight path for recovering energy from the wind. Evolved neurocontrollers maintain these flight trajectories in face of significant wind deviations suggesting that using evolutionary robotics for kite control is worth pursuing.

Furey & Harvey have produced three articles concerning maintaining flight trajectories during the power generation phase (Furey & Harvey, 2007), switching between behaviours (Furey & Harvey, 2008[1]) and flight trajectory robustness (Furey & Harvey, 2008[2]). In the first of these studies focusing on the power generating phase, a kite physics simulation is implemented. “Real world kite systems are subject to a high degree of environmental variability; wind speed, precipitation and icing will all affect the kite’s performance. The real world will also present such non-linearities that current simulations are unable to render in detail without prohibitive computational costs, primarily turbulence.”

Microbial genetic algorithms have been used successfully to produce neural network controllers that modulate the kites flight path (Furey & Harvey, 2007) as well as producing controllers modulate agent behaviour in a toy-problem i.e. switching between mutually exclusive behaviours of charging and discharging (Furey & Harvey, 2008[1]). For the kite problem this would involve switching between power generating behaviour and reeling in the kite. Additionally, Furey & Harvey consider the robustness of the neurocontrollers to large deviations in wind speed and direction, as well as addressing the problem of controlling the kite with different line lengths (Furey & Harvey, 2008[2]).
The objective of this work is to provide an overview of available literature by Furey & Harvey, and to make a simple implementation of the genetic algorithm used to evolve the neural networks controlling the kite. Though the implementation presented in this paper is applied to a toy-problem of evolving pi estimates in a population of integers.

The Microbial Genetic Algorithm

The microbial genetic algorithm used is a stripped down, minimalist version of traditional steady state- and generational genetic algorithms. It has been taught and used at University of Sussex for more than ten years, its simplicity makes it particularly suitable for teaching. The algorithm is based around bacterial conjugation rather than sexual reproduction. Selection is implemented through tournament trials, where individuals are selected at random and have their fitness evaluated through comparison. “The individual with least fitness has a percentage of its genetic material replaced by corresponding material from the fitter individual (possibly adding mutation)” basically infecting the other individual, “while keeping both individuals in the population. In bacterial conjugation it will typically be a rather low percentage that is replaced; to reproduce the typical effects of sexual reproduction this rate should be around 50%, but could in principle be any value between 0% and 100%” (Harvey, 2009). The core part of the Microbial GA is given as C-based pseudo-code in Harvey (2009). “This minimal GA routine incorporates Rank Selection, (variable rates of) Recombination and Mutation, (variable size) Demes, and Elitism, in just a few lines” (ibid.):

```c
void microbial_tournament(void) {
    int A, B, W, L, I;
    A= P*rnd();                // Choose A randomly
    B=(A+1+D*rnd())%P;        // B from Deme, %P[modulus]..
    if  (eval(A) > eval(B)) {W=A; L=B}  // ..for wrap-around
    else  {W=B; L=A}          // W=winner L=loser
    for  (i=0; i<N;i++) {     // walk down N genes
        if (rnd() < REC)     // RECombn rate
            gene[L][i]=gene[W][i]; // Copy from winner
        if (rnd()<MUT)       // MUTation rate
            gene[L][i]ˆ1;      // Flip a bit
    }
}
```

“We shall assume that there is a population size P of binary genotypes length N, stored in an array of the form gene[P][N], that have been suitably initialized at random for the start of the GA; we have a function eval(i), that returns some real value for the i:th member of the population. This code represents a single tournament, that is repeated as many times as is considered necessary. We assume a pseudo-random number function rnd() that returns a real number in the range [0.0,1.0]. REC is the recombination or ‘infection’ rate (suggested value 0.5) and MUT is the (per locus) mutation rate. D is the deme size [sub-population size].

The Neural Networks
Two classes of small, initially naïve time recurrent neural networks were evaluated by Furey & Harvey (2007), both consisting of 5 input neurons and 7 fully interconnected neurons, both inhibitory and excitatory connections were permissible. Only data measurable with line length and tension sensors at ground level were available to the network, as shown in Table 1 (Furey & Harvey, 2007), all sensory data is subject to low level Gaussian noise i.e. statistical white noise.

<table>
<thead>
<tr>
<th>Input No.</th>
<th>Input data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Total line tension</td>
</tr>
<tr>
<td>2</td>
<td>Tension difference between left and right line sets</td>
</tr>
<tr>
<td>3</td>
<td>Average line azimuth</td>
</tr>
<tr>
<td>4</td>
<td>Average line elevation</td>
</tr>
<tr>
<td>5</td>
<td>Difference in elevation between left and right steering lines</td>
</tr>
</tbody>
</table>

Table 1

The two classes of neurocontrollers that were evaluated by Furey & Harvey (2007) are a discrete time recurrent network and a continuous time recurrent network. The discrete time recurrent networks nodes’ activation value at a given timestep \( t (t\Delta=0.004) \) is given by Eq. 1 (Furey & Harvey, 2007). The activation \( a \) value for a specific neuron \( j \), at given timestep \( t \), is equal to the difference of the neurons sigmoid function value and its threshold value \( \theta \). The activation value and sigmoid function differs somewhat from a typical textbook example (Russel & Norvig, 1995) in that it uses timesteps, and the previous activation value from node \( I \) in the sigmoid function.

\[
a'_j = \sigma\left( \sum w_i a_i^{t-1} \right) - \theta_j
\]

Equation 1

The continuous time recurrent network is integrated using Euler integration at the same timestep as the physics simulation, effectively letting the simulations timesteping determine the timestep of the nodes in the network. A single neurons’ dynamics in the continuous time recurrent networks is given by Eq. 2 (Furey & Harvey, 2007). This class is a bit different, the neurons’ time constant and activation value is equal to the negative of the neurons activation value added to the sum of the weight of input neurons and the sigmoid function, which takes the difference of the activation value from the previous timestep and the threshold value.

\[
\tau_j a_j = -a_j + \sum w_i \sigma (a_i - \theta_i)
\]

Equation 2

“In both cases… …One of the neurons is chosen to be the output neuron, and 0.5 subtracted from its sigmoided output. The motor position is modified by 1% of the difference between the network output and its current position each millisecond. This process has three consequences; the motor output is shielded from the majority noise in the network, the motors are prevented from moving at speeds that are unrealistically fast and finally the extremes of
motor output that correspond to a 1 m difference in line lengths, usually destabilising for the kite, are only rarely achieved.” (Furey & Harvey, 2007)

To produce neurocontrollers for the kite simulation a simple version of the microbial genetic algorithm was used to produce “DNA strings composed of real values determining weights both between all neurons and sensory inputs, thresholds and in the [continuous time recurrent network’s] case, time constants for each neuron in the network.” ”The algorithm uses the simulation gust generator to generate a wind trace, two individuals are then selected at random from the population” and their fitness is evaluated. “The individual with least fitness has its DNA string copied over by the winner with a small mutation applied to every value in the string.” Two different fitness functions were used in the study, “either the average of the aerodynamic forces produced by the whole kite, or the component of the aerodynamic force in line with the lines, is determined using the identical wind trace.”

Kite Physics Simulation

In the simulation implementation the kite is treaded as a single entity forming a curved airfoil, which forms a semi-circular arc. “The kite is tethered by 4 lines and controlled from the ground by adjusting the relative lengths of the rear 2 lines. The kite is allowed to flex, without the fabric stretching and following the common Leading Edge Inflatable (LEI) kite configuration, in which the leading edge is an inflatable baton i.e. the leading edge of the kite that normally faces into the wind is more rigid than the trailing edge opposite. Line tension and angle data is fed to the neurocontroller, which feeds back length actuation to the kite model.”

**Figure 1 & 2**

**Figure 1:** A simplified schematic in which neurocontrollers are evolved. (Furey & Harvey, 2007)

**Figure 2:** The initial configuration of particles (dots) and constraints (lines). “The light grey constraints reinforce the arc shape of the kite and prevent ‘jellyfish’ type flapping motion, effectively performing the same role as the inflatable ribs that maintain the shape of LEI kites. The three lowest particles are the tether points. Zigzag lines indicate the positions at which the canopy is sliced for aerodynamic calculations.” (Furey & Harvey, 2007)

The constraints linking the particles were treated as infinitely stiff springs. The aerodynamic forces were calculated for each slice of the kite, as demarcated by the zigzag lines running from the leading to the trailing edge of the kite. The aerodynamic and gravitational forces upon each slice were distributed among its constituent particles, and acceleration calculated
via Newton’s second laws of motion. Continuous real time was approximated by small timesteps at which a discrete update was made. A simple arbitrary gust generation model was used that generated deviations around a base wind speed of 8 m/s. At each timestep there was a small probability of that a gust or lull initiating. No lateral wind deviations were implemented, although this and the use of real recorded wind data would be logical extensions to the simulation (Furey & Harvey, 2007).

**Results of Evolved Neurocontrollers for Flight Trajectory Control**

The key result from the initial study in Furey & Harvey (2007) is that a simple 7 interneuron network of evolved neurocontrollers is able to control a simulated kite to fly in figure eight trajectories in only 200 generations with a population size of 20. With a trial time of 42 seconds, this corresponds to less than 47 hours of real world flight time.”

The discrete time recurrent neural networks outperformed the continuous time recurrent neural networks, for reasons unknown. “The improvement of fitness across generations does not plateau after 200 generations suggesting that additional evolutionary time would result in further improvement in performance.” Whether the total force generated at the airfoil or the force that was acting along the lines was used for measure of fitness did not affect the flight trajectory, “although controllers evolved under the latter condition performed significantly more figure repetitions within the trial period.

**Behaviour Modulation**

One of the challenges in developing controllers for the active control of kite power generating systems is to modulate behaviour between a power generation phase and a reeling in phase (Fig 3).

![Figure 3: The kite should alternate between mutually exclusive behaviours of the kite pulling out the spool (A) and the kite being reeled in (B). (Furey & Harvey, 2007)](image)

Furey & Harvey (2008[1]) addresses this problem by designing a toy problem task capturing the key properties of the kite strategy modulation problem, “without the computational burden of modelling the aerodynamics of the kite”. The task was “based around a simple 2 wheeled,
2 sensor agent situated within an unbound 2d area. The analogy for the flying lines is a battery, charged by an orientation independent ‘solar panel’ on the agent. As with the lines there are upper and lower hard limits to the battery level. As success is judged by quantity of energy passed to the battery, the agent can score more highly by repetitively switching between charging and discharging behaviour.” In the task domain there is a single light source which constitutes the energy source, the area around the light consists of three different zones demarcated by concentric circles. The zone closest to the light (zone C) overpowers the battery; the next zone is where the battery charging rate plateaus (zone B), when the agent is in the zone furthest away from the light (zone A) the charging rate of the battery becomes negative. Two regimes were evaluated in Furey & Harvey (2008[1]), the first is the ‘threshold’ regime were a value determined by the environment independent of the agents activities sets the boundary for switching behaviour. The other regime is where the agent has some control of its own switching point. “This is implemented by implicitly by introducing a battery cost to movement, if the agent stays still it will charge until full at any point in the environment at a rate determined by the distance to the light, it is moving at a speed that which consumes more energy that is recouped from the environment that will drive the down the battery. The agent in this scenario is forced to make a more complex , but more realistic trade off in that moving far from the high-quality resource will speed up discharging, but mean that for portions of the journey, time is spent near energy neutral and at suboptimal charging speeds”, the authors label this ‘enactive switching’.

The same dynamics as the continuous time recurrent neural network from Furey & Harvey (2007) as specified in equation 2 was used for this task. The neural network that was used for the agents’ behaviour consisted of 4 input neurons, 3 hidden layer neurons and 2 output neurons, both inhibitory and excitatory connections were permissible. There was 4 inputs to the agent; right light sensor value, left light sensor value proportion of battery capacity remaining and battery charge rate. Each input neuron connected to every hidden and output layer neuron, but receives no other input. The hidden and output layer neurons were fully interconnected.

The microbial genetic algorithm was used also in this task. A population of genotypes (artificial DNA) was randomly initialised, specifying possible parameter values. Pairs of neurocontroller parameter sets taken from a population of 20 had their performance evaluated. The winner would overwrite the loser with a small additive random mutation added to a random 50% subset of its parameter values. This process would continue for several hundred iterations each consisting of 10 competitive trials. For all tasks, fitness was evaluated judging by the amount the battery charged (Furey & Harvey, 2008[1]).

Results of Evolved Neurocontrollers for Behaviour Modulation

Both the ‘threshold switching’ and the ‘enactive switching’ regimes enjoyed relative success. Although agents evolved under the ‘threshold switching’ regime were outperformed by the agents evolved under the ‘enactive switching’ regime, suggesting that a similar approach of letting the agent set its own threshold values for switching behavior would be appropriate for the kite behaviour modulation task.
Robustness of Evolved Neurocontrollers and Future Work

In their most recent published work, Furey & Harvey (2008[2]) present an overview of previous work and demonstrate that neurocontrollers evolved using microbial genetic algorithms are able to fly a kite in trajectories previously shown through mathematical analysis to be optimal for generating power. With a modified version of the simulation used in Furey & Harvey (2007) they also demonstrated that the evolved neurocontrollers were robust to small lateral wind deviations. Issues concerning limited temporal dynamics, along with active control of reel-out speed will likely be address in future research.

Implementing the Microbial Genetic Algorithm using Python

In order to gain further understanding of genetic algorithms in general, I have written a small program in Python using the microbial genetic algorithm to find the integer closest to pi out of a population of random integers. The Python modules random and math are used for the program.

The first function of the program creates population of a given size, consisting of a list of random integers, in this example the integers are set to be in the range of 0.0-10.0:

```python
def makePopulation(populationSize):
    population = []
    populationIndex = populationSize
    while populationIndex != 0:
        population.append(random.uniform(0,10))
        populationIndex = populationIndex - 1
    return population
```

The fitness function returns the difference between a given integer from the population and pi.

```python
def evaluateFitness(individual):
    if individual > math.pi:
        return (individual - math.pi)
    elif individual < math.pi:
        return (math.pi - individual)
```

The mutation function chooses randomly to either add or subtract a percentage of an integer. The mutation rate is set in this function.

```python
def mutateIndividual(individual):
    mutateMethod = random.choice(['+', '-'])
    mutationRate = 0.05
    if mutateMethod == '+':
        individual = individual + (individual*mutationRate)
    return individual
    elif mutateMethod == '-':
    ```
individual = individual - (individual*mutationRate)  
return individual

The microbial tournament function is closest to the C pseudo-code presented in Harvey (2009) and on the previous page. It chooses a primary individual (A) randomly from the population and defines a subpopulation (deme) of a given size, the subpopulation will consist of the elements following A in the population list. The list wraps around itself, so that if A is in the end of the list, the first elements in the list will be selected as the subpopulation. B is chosen randomly from the subpopulation and its fitness is compared to A. The loser is overwritten by the winner (100%) and mutates.

def microbialTournament(population, demeSize):
    A = random.choice(population)  
posA = 0
    for individual in population:
        if individual != A:
            posA = posA+1
        elif individual == A:
            posA = posA
    deme = population[(posA+1):((posA+1)+demeSize)]
    if len(deme) != demeSize:
        deme = deme+(population[0:(demeSize-len(deme))])
    B = random.choice(deme)
    if evaluateFitness(A) < evaluateFitness(B):
        B = mutateIndividual(A)
    elif evaluateFitness(B) < evaluateFitness(A):
        A = mutateIndividual(B)

The main loop is where population size and generation values are configured. It runs the microbial tournament function as many times as specified in the generations-variable. When it’s done the fitness function is called to find the integer closest to pi in the population and print it.

def main():
    population = makePopulation(2000)
    generations = 6000
    genInfo = generations
    results = []
    while generations != 0:
        microbialTournament(population, 5)
        generations = generations - 1
    best = population[-1]
    for individual in population:
        if evaluateFitness(individual) < evaluateFitness(best):
            best = individual
    best = str(best)
    print best
While the program is far from perfect it made for good practice in implementing genetic algorithms. The results vary somewhat, using integers between 0.0 and 50.0 a population size of 2000, mutation rate of 15% and 6000 generations, ten runs yielded the following results:

3.1405238593, 3.13536453231
3.14790611466, 3.14219293261
3.14027232168, 3.14239670673
3.1324121019, 3.1392566842
3.14321133012, 3.12695991632

The value of pi from the python math module is 3.14159265359, while the program occasionally gets at least two decimals correct, it is not very accurate, but it consistently returns a near-pi value. The average result from the trial above is 3.13904964998.
References


