Offline Training: Vertical Take-off

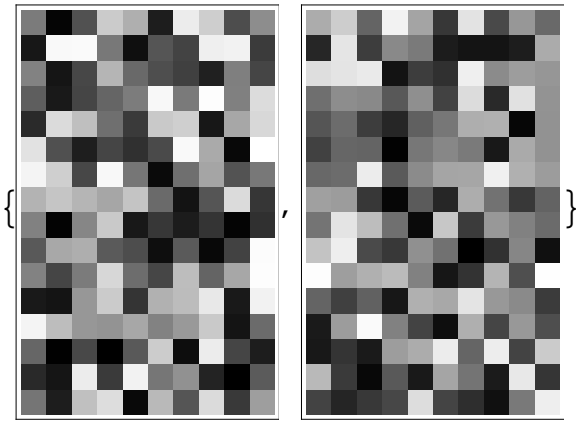
Experiment

Eight flight paths of a quad copter are recorded i.e. obtained from a simulator simulating the differential equations of motions of the copter. The said paths are then input into an Artificial Neural Network adaptive learning program.

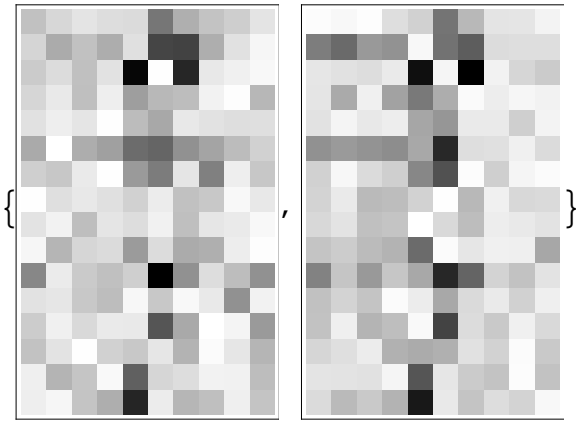
Two matrices made up of numbers between -1 and +1 are obtained as a result of the learning. These matrices originally were random numbers between -1 and +1, but through the algorithm of learning e.g. Backpropagation Adaptive Learning, the said random matrices form some discernible pattern of numbers which store the learnt knowledge of how the copter flies.

Remark: *No gust or wind cases have been included in the training data. Training was obtained from 3 seconds of flight, the actual flight duration was set to 20 seconds.*

Unlearnt Random Matrices



Learnt Patterened Matrices

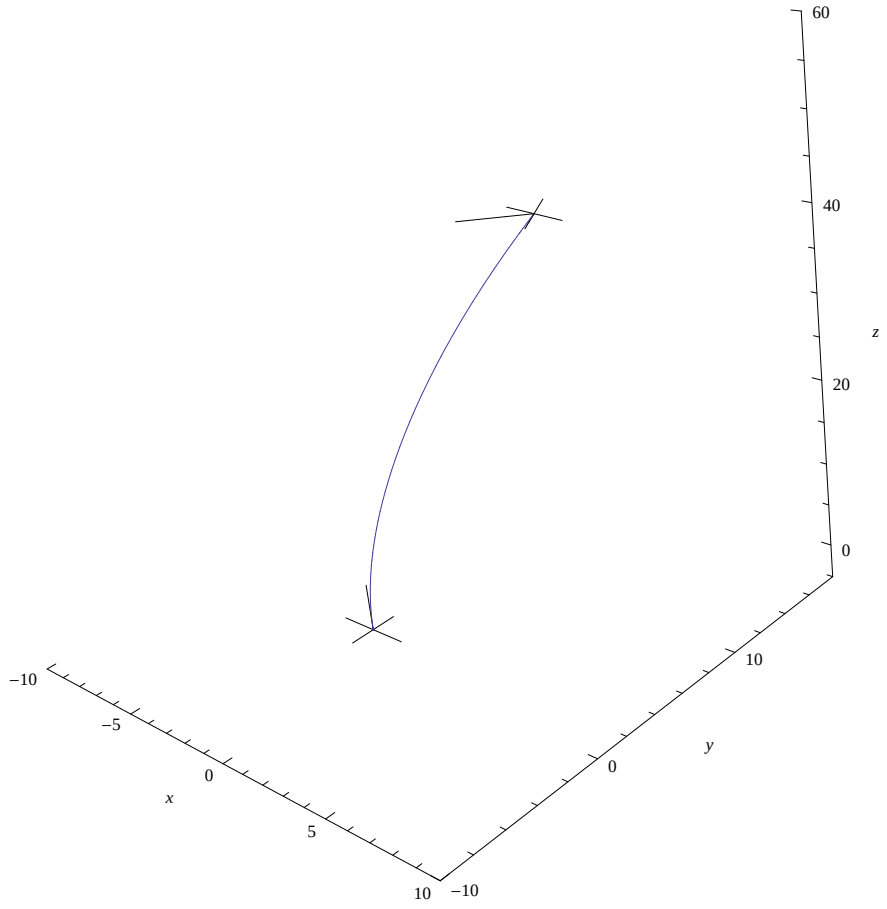


Remark: *Numbers close to 1 appear more and more as black, the numbers closer to 0 appear more and more as white. The more black areas correspond to the stronger synaptic pulse in a biological neuron.*

Sample Training Case

The sample cases of 9 flights where produced by constant speed motors. However the motors's speed were different therefore variety of take off scenarios were obtained.

Remark: *'+' is the frame of the quad-copter and the vertical line in the middle is the upward arrow so the viewer can visualize the orientation. Due to perspective and other scalings, the copter frame is rendered skewed.*



The Backpropagation Adaptive Learning

Because we have more weights in our network than in perceptrons, we firstly need to introduce the notation: wij to

specify the weight between unit i and unit j. As with perceptrons, we will calculate a value Δij to add on to each weight in the network after an example has been tried. To calculate the weight changes for a particular example, E, we first start with the information about how the network should perform for E. That is, we write down the target values ti(E) that each output unit Oi should produce for E. Note that, for categorisation problems, ti(E) will be zero for all the output units except one, which is the unit associated with the correct categorisation for E. For that unit, ti(E) will be 1.

Next, example E is propagated through the network so that we can record all the observed values oi(E) for the output nodes Oi. At the same time, we record all the observed values hi(E) for the hidden nodes. Then, for each output unit Ok, we calculate its error term as follows:

$$\delta_{O_k} = o_k(E)(1 - o_k(E))(t_k(E) - o_k(E))$$

The error terms from the output units are used to calculate error terms for the hidden units. In fact, this method gets its name because we propagate this information backwards through the network. For each hidden unit Hk, we calculate the error term as follows:

$$\delta_{H_k} = h_k(E)(1 - h_k(E)) \sum_{i \in outputs} w_{ki}\delta_{O_i}$$

In English, this means that we take the error term for every output unit and multiply it by the weight from hidden unit Hk to the output unit. We then add all these together and multiply the sum by hk(E)*(1 - hk(E)).

Having calculated all the error values associated with each unit (hidden and output), we can now transfer this information into the weight changes Δij between units i and j. The calculation is as follows: for weights wij between input unit Ii and hidden unit Hj, we add on:

$$\Delta_{ij} = \eta\delta_{H_j} x_i$$

[Remembering that xi is the input to the i-th input node for example E; that η is a small value known as the learning rate and that δHj is the error value we calculated for hidden node Hj using the formula above].

For weights wij between hidden unit Hi and output unit Oj, we add on:

$$\Delta_{ij} = \eta\delta_{O_j} h_i(E)$$

[Remembering that hi(E) is the output from hidden node Hi when example E is propagated through the network, and that δOj is the error value we calculated for output node Oj using the formula above].

Each alteration Δ is added to the weights and this concludes the calculation for example E. The next example is then used to tweak the weights further. As with perceptrons, the learning rate is used to ensure that the weights are only moved a short distance for each example, so that the training for previous examples is not lost. Note that the mathematical derivation for the above calculations is based on derivative of σ that we saw above. For a full description of this, see chapter 4 of Tom Mitchell's book "Machine Learning".

Given the array of input calculate the trained matrices as follows with input obtained from the list of chosen samples from above:

/* . is matrix multiplication */
H1 = (Sigmoid(w1.input));
output = Sigmoid(H1.w2);

```
/* outputdesired is from the list of chosen mapped tuples above */
/* * means element by element multiplication (1, 2) * (2, 3) = (1*2, 2*3) */
deltaO = (output * (1- output)) * (outputdesired - output);

deltaH = (H1 * (1 - H1)) * (w2.deltaO);

/* learning_rate usually set to 0.1 and serves as a fine tuning parameter */
delta1[i][j] = learing_rate * deltaH[i] * input[j];
detla2[i][j] = learning_rate * H1[i] * deltaO[j];

w1trained = w1 + delta1;
w2trained = w2 + delta2;
```

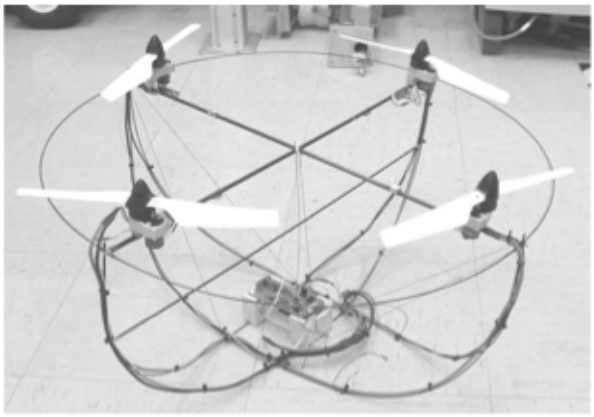


Fig. 1. Quadrotor

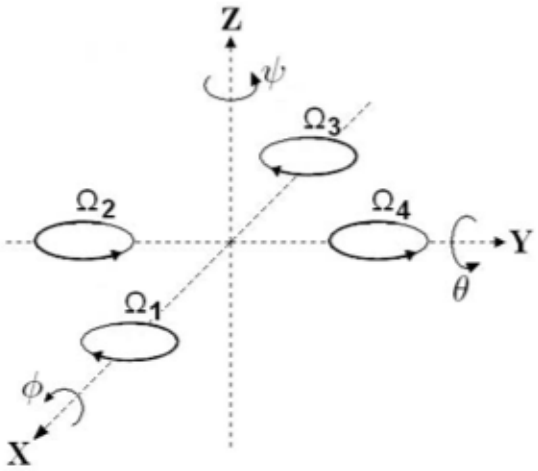


Fig. 2. Coordinate systems

2.2. Equations of Motion

Rather than use rotor speeds as control inputs, it is simpler to define control signals:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -1 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} \Omega_1^2 \\ \Omega_2^2 \\ \Omega_3^2 \\ \Omega_4^2 \end{bmatrix} \tag{3}$$

The above matrix is invertible so commanded speeds follow directly from control signals. Using the dynamics developed in (1) and (2) along with input definitions (3), we arrive at the angular and Cartesian equations of motion

$$\ddot{\phi} = \dot{\theta}\dot{\psi}(I_Y - I_Z)/I_X - (J_r/I_X)\dot{\theta}\Omega_r + (lb/I_X)u_2 \tag{4}$$

$$\ddot{\theta} = \dot{\phi}\dot{\psi}(I_Z - I_X)/I_Y + (J_r/I_Y)\dot{\phi}\Omega_r + (lb/I_Y)u_3 \tag{5}$$

$$\ddot{\psi} = \dot{\phi}\dot{\theta}(I_X - I_Y)/I_Z + (d/I_Z)u_4 \tag{6}$$

$$\ddot{X} = -(\sin\theta\cos\phi)(b/m)u_1 \tag{7}$$

$$\ddot{Y} = (\sin\phi)(b/m)u_1 \tag{8}$$

$$\ddot{Z} = -g + (\cos\theta\cos\phi)(b/m)u_1 \tag{9}$$

where l is the length of an arm, b is a thrust factor and d is a drag factor. For details on deriving (4)-(6) see, for example, [4] or [8]. The Cartesian equations (7)-(9) use the rotation matrix $\mathcal{R}_{ZXY}$ rather than the more common $\mathcal{R}_{ZYX}$. This makes providing the desired roll and pitch (and their velocities) to achieve desired Cartesian coordinates possible without linearization. With this rotation matrix, the commanded yaw should always be zero and roll should be small to ensure Euler angles are close to roll, pitch and yaw provided by gyroscopes. Thus, the X axis faces into the flight direction, or ideally into the wind when hovering. The constants used for system development and simulations were obtained from [9] (Table 1).

Table 1. Constants

Parameter	Definition	Value
l	lever length	0.232 m
m	mass of quadrotor	0.52 Kg
d	drag coef.	7.5e-7 N m s^2
b	thrust coef.	3.13e-5 N s^2
I_X	X Inertia (I_{xx})	6.228e-3 Kg m^2
I_Y	Y Inertia (I_{yy})	6.225e-3 Kg m^2
I_Z	Z Inertia (I_{zz})	1.121e-2 Kg m^2
J_r	rotor inertia	6e-5 Kg m^2

Single Layer Neural Network, 16 Neurons

Input:

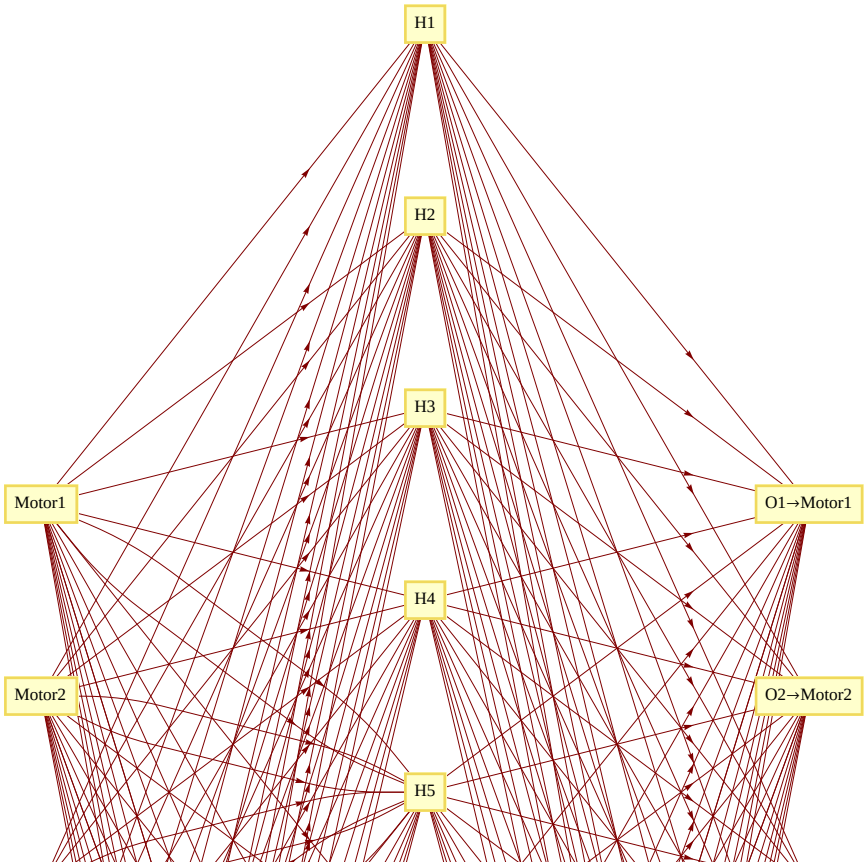
[motor1, motor2, motor3, motor4, Roll, Pitch, Yaw, x, y, z]

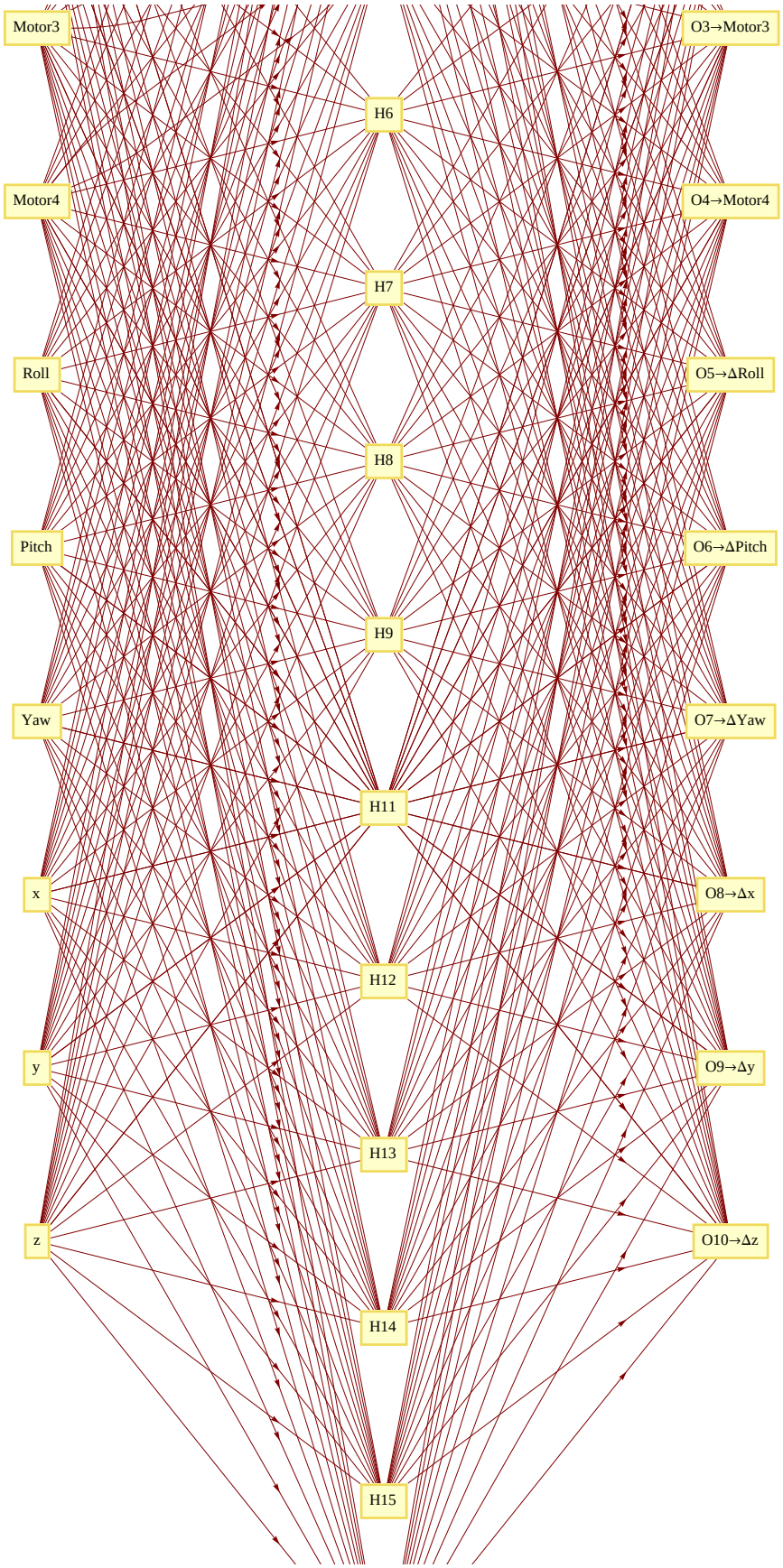
Output:

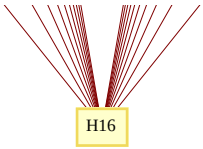
[motor1, motor2, motor3, motor4, ΔRoll, ΔPitch, ΔYaw, Δx, Δy, Δz]

Input is phase-space vector at time t + ticks while the Output is the corresponding phase-space vector at time t, for a fixed amount ticks.

Δ is the amount of change to the parameter at time t to get the corresponding parameter at time t + ticks.




Motor3
Motor4
Roll
Pitch
Yaw
x
y
z
H6
H7
H8
H9
H11
H12
H13
H14
H15
O3→Motor3
O4→Motor4
O5→ΔRoll
O6→ΔPitch
O7→ΔYaw
O8→Δx
O9→Δy
O10→Δz

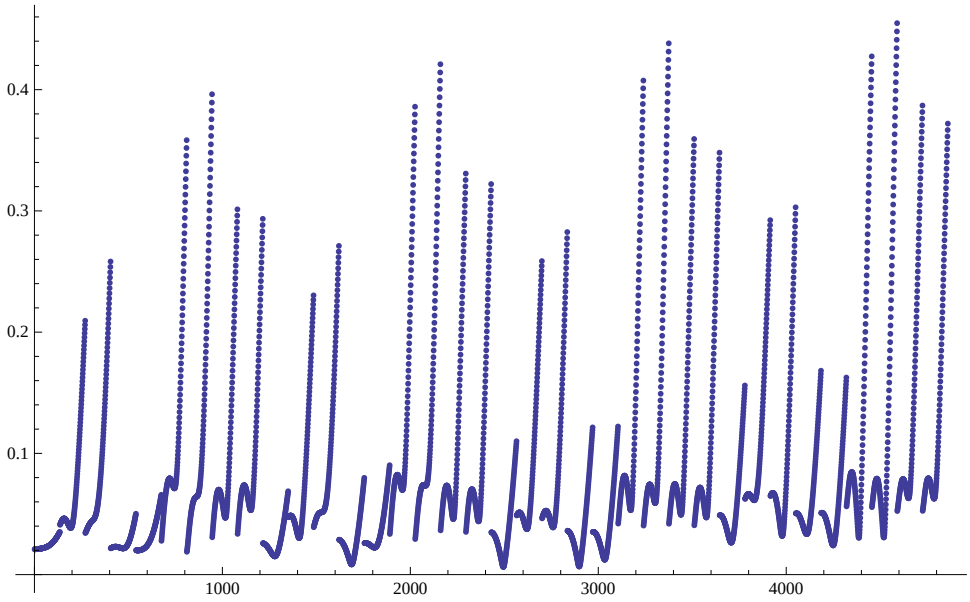


Error Statitics

μ = 0.0908736
σ = 0.0830461

x-axis is the index for the input ouput pair, y-axis is the error divided by the norm of the output phase-space vector.

4860 input output vectors provided, repeated 10 times, fully random-shuffled at each repetition of the learning, total of 48,600 learning pairs comprised the learning.



NN Controlled Take-off

takeoff()

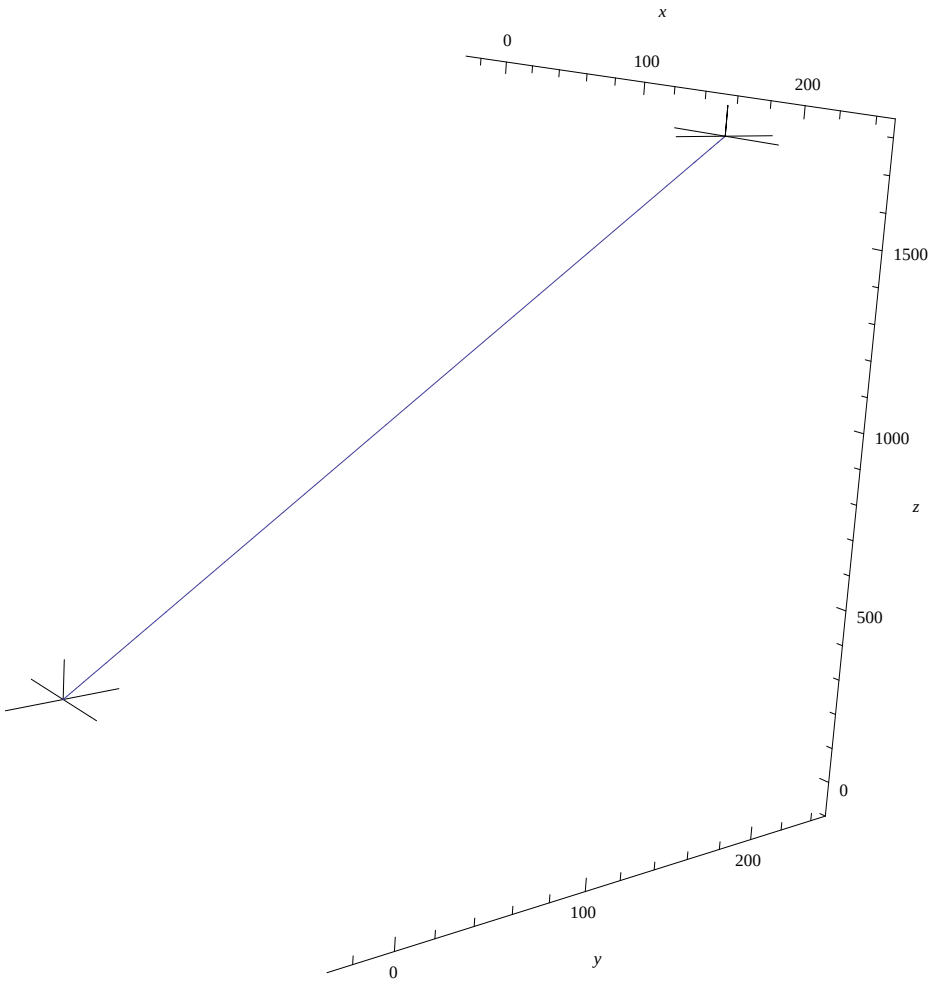
```
{

/* motors are between 0 and 1 in order for NN learning to work */
/* thrust_initial causes constant equal speed for all motors to be the take off speed */
motors = make_motors (thrust_initial);
/* set all deltas to 0 */
prev = init_diffs (motors);

/* these are wx, wy and so on see below */
get_coeffs();
/* mass and drag and so on */
config_copter();

/* get the learnt matrices from offline training, amounts for ticks and other related params */
/* w1 w2 are learnt matrices from offline training */
boot_NN(w1, w2);

/* ignition runs the motors for about 3/1000s to start take off*/
current = ignition(motors);

/* Control Loop, nornally there is two of them, but we are only doing vertical take off so one suffices */
while (current->z < hover_height)
{
        dx = current->x - prev->x;
        dy = current->y - prev->y;
        /* no z since x and y should be 0 but z must increase */

        dRoll = current->Roll - prev->Roll;
        dPitch = current->Pitch - prev->Pitch;
        dYaw = current->Yaw - prev->Yaw;

        /* must calculate the delta including the angles */
        delta = dx*dx + dy*dy +dRoll*dRoll + dPitch*dPitch + dYaw*dYaw;

        /* NN control should only be called when deviating off the desired path */
        if (delta < threshold)

                /* multiply by coefficients to account for the learning errors */
                 /* The differentials for ticks duration, multiply by -1 i.e. that much to adjust to get 0 coordinate in
ticks time */
                input->x = -current->x*wx;
                input->y = -current->y*wy;

                /* use abs to force positive z changes */
                input->z = abs(current->x)*wz;

                input->Roll = -current->Roll*rpy;
                input->Pitch = -current->Pitch*rpy;
                input->Yaw = -current->Yaw*rpy;

                /* copy the current motors into the input for NN */
                set_input_motors(input, motors);
```

```
                /* Approximator gives the new motors that will cause x, y, roll, pitch, yaw go back to 0 in ticks
duration */
                motors_new = NN_universal_approx (input, w1, w2);

                motors = motors_new;
                prev = current;
        }

        /* fly either blocks the loop for count seconds (fraction of ticks) or is interrupt driven */
        fly (motors, rpm, count);

        current = get_current();

        /* Reset motors to the constant take off configuration, or else the accumulation
           of errors might cause faulty control */
        motors = make_motors (thrust_initial);

} /* while */

}

Motors *
NN_universal_approx (input, w1, w2)
{

float H1[16];

/* . is matrix multiplication */
H1 = (Sigmoid(w1.input));
output = Sigmoid(H1.w2);

return (get_motors_output(output));
}
```

Gust

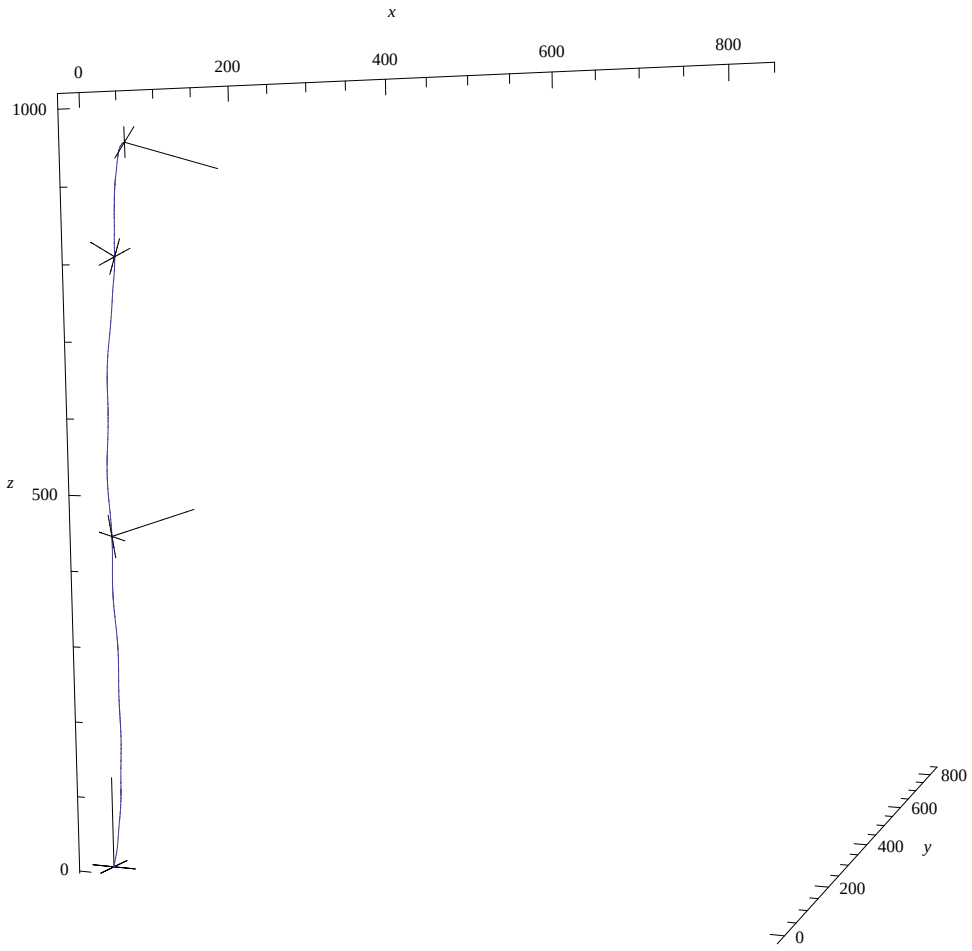
During the take-off, a strong gust i.e. acceleration of [1, 1, 0] applied along both x and y axis to the CG of the quad-copter (no torque generated). Consequently the quad-copter is blown away about 282.786 meters, as computed by the simulator.

Remark: *Acceleration of [0, 0, -6.5] does not allow the copter to take-off.*
Remark: *‘+‘ is the frame of the quad-copter and the vertical line in the middle is the upward arrow so the viewer can visualize the orientation.*

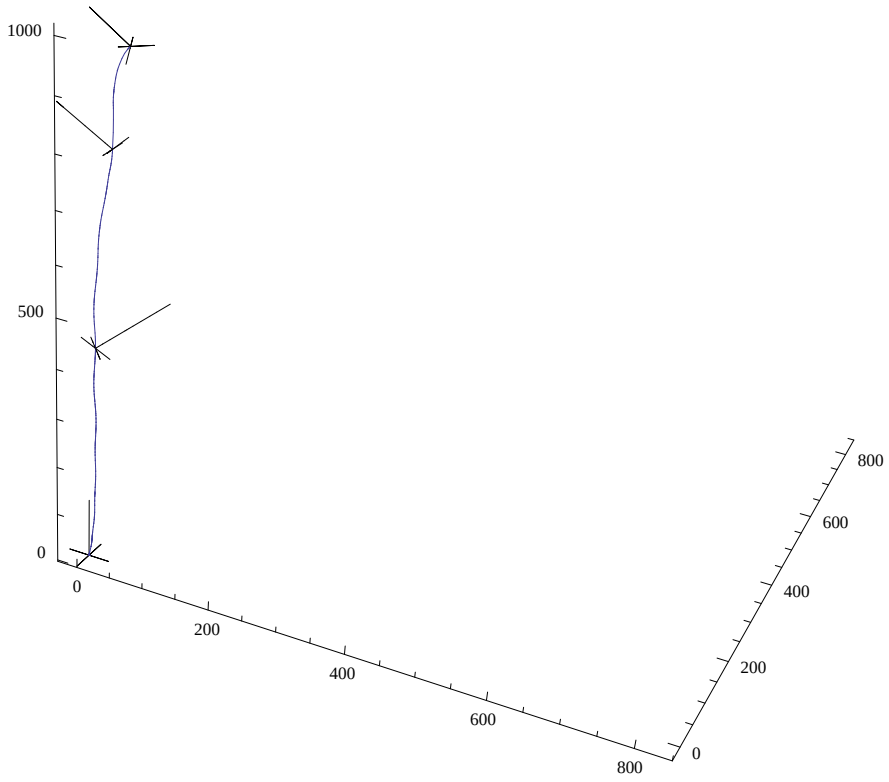


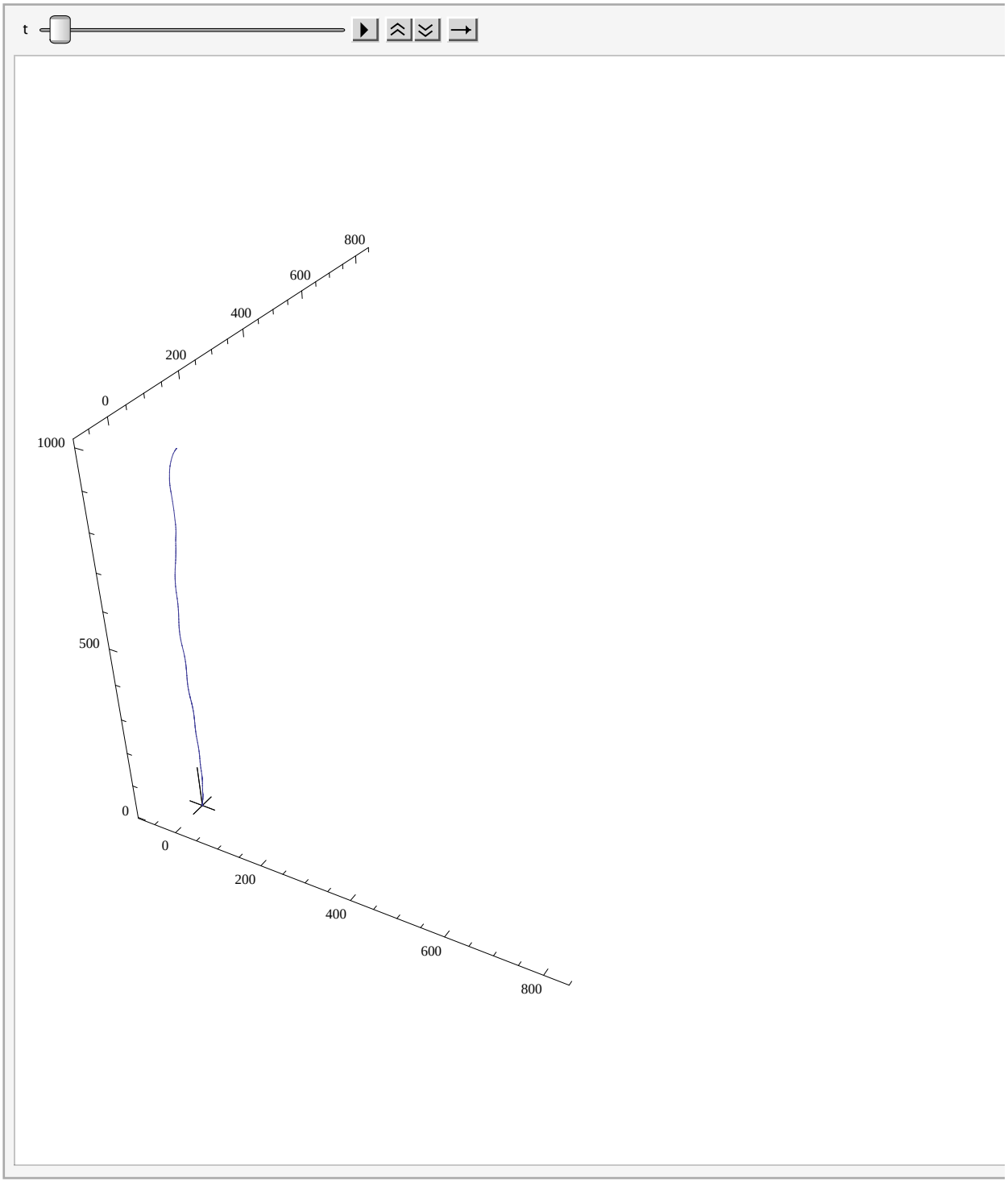
The NN control is turned on under the same gust conditions and surprisingly only 54.197 drift is computed by the simulator i.e. the NN control has fought against the gust to make sure the quad-copter flies vertically.

Remark: *There were no gusts present in the training cases and no equations entered or any calculations made to compensate for the speed of the wind or sense the speed of the wind. However the NN controller fights against the gust, learned from the test cases a general equation for flight, and therefore the drift drastically decreased.*



Acceleration of a gust at [1, 0, 0] drifts by 199.96 along x-axis, but turning on the NN control only a drift of 65.85 is computed by the simulator:


1000
500
0
0
200
400
600
800
0
200
400
600
800


t
800
600
400
200
0
1000
500
0
0
200
400
600
800