

# CS545: Assignment 8

## Learning to Control Octopus Arm with Reinforcement Learning

December 16, 2009

---

### Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Reinforcement Learning</b>	<b>1</b>
<b>3</b>	<b>Octopus Arm</b>	<b>2</b>
3.1	Dynamics . . . . .	2
3.2	Simplified Actions . . . . .	3
3.3	Method . . . . .	4
3.4	The Simulation Test . . . . .	11
<b>4</b>	<b>Conclusion</b>	<b>12</b>
	<b>References</b>	<b>12</b>

---

## 1 Introduction

In this assignment, I built octopus arm model for reinforcement learning. Octopus arm is one of the most fascinating appendages in nature. Composed with muscles and tissues without a rigid skeleton, it is exceptionally flexible, and it makes controlling the octopus arm difficult but interesting. Reinforcement learning (RL) was one of the most interesting algorithms that I learned in class. After applying RL to solve positioning marble problem and mountain car problem, I wanted to play more with RL agent. For this reason, I chose the octopus arm, complex system enough to study and experience RL further.

In this paper, I will introduce the octopus arm modeling. In Section 2, I introduced RL algorithms, Least Square Time Difference (LSTD) and Gaussian Process Time Difference (GPTD) briefly along with Time Difference (TD) algorithm. Section 3 contains the linear modeling of octopus arm, implementation, and test simulation. Section 4 presents conclusion and future works including RL experiments that I have not covered because of the error in the model.

## 2 Reinforcement Learning

In the previous assignment, I used the neural networks for TD. Below shows the feed-forward and backpropagation rules.

$$\mathbf{Z} = h(\tilde{\mathbf{X}}\mathbf{V})$$
$$\mathbf{Y} = \tilde{\mathbf{Z}}\mathbf{W}$$

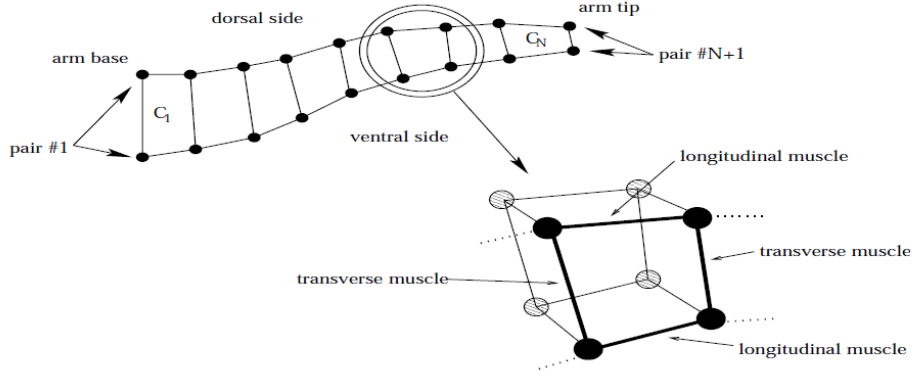


Figure 1: The two dimensional octopus arm model. Muscular hydrostats are separated into several compartments. [5]

$$\mathbf{V} \leftarrow \mathbf{V} + \rho_h \left( \frac{1}{N} \tilde{\mathbf{X}}^T (\mathbf{R} + \mathbf{Q} - \mathbf{Y}) \hat{\mathbf{W}}^T \cdot (1 - \mathbf{Z}^2) + \lambda \mathbf{V} \right)$$

$$\mathbf{W} \leftarrow \mathbf{W} + \rho_o \left( \frac{1}{N} \tilde{\mathbf{Z}}^T (\mathbf{R} + \mathbf{Q} - \mathbf{Y}) + \lambda \mathbf{W} \right)$$

To improve efficient use of data and eliminate step-size parameters that need to configure manually, Bradtke and Barto [3] proposed Least Square Time Difference (LSTD). LSTD converges to the same the coefficient, but unlike TD performing gradient descent, LSTD( $\lambda$ ) builds explicit estimates of the  $\mathbf{C}$  matrix and  $\mathbf{d}$  vector, and then solves  $\mathbf{d} + \mathbf{C}\beta_\lambda = 0$  directly [2] when  $\mathbf{d}$  is the estimated sum of reinforcements, and  $\mathbf{C}$  is the estimated sum of the term  $(Q_{t+1} - Q_t)$ .

Gaussian Process Temporal Difference (GPTD) learning offers a Bayesian solution to the policy evaluation problem of reinforcement learning. GPTD modeling assumptions are that the prior and the observation-noise distribution are Gaussian. GPTD also use Bayesian rule to compute the posterior distribution over value functions [4].

### 3 Octopus Arm

In this section, I implemented the two dimensional octopus arm modeled by Yoram Yekutieli [6]. The basis of the model is that muscular hydrostats maintain a constant volume [1]. This constraints allow forces to be transferred to other parts. Modeling in two dimension reduces the complexity, however, there were still wide variety of motions. In this model, there are four types of forces: internal forces generated by muscle contraction, forces influenced by gravity and buoyancy, drag forces moving through water, and compartmental pressure that maintains the constant volume. Figure 3 shows the two dimensional model of octopus arm. Each compartment is composed with longitudinal muscles (ventral and dorsal) and transverse muscles. Following sections describes the physics of each force, implementation of simplified actions, implementation of forces, and simulation test.

#### 3.1 Dynamics

According to Newton's second law, the motion can be

$$\mathbf{M}\ddot{\mathbf{q}} = \mathbf{f}^m + \mathbf{f}^g + \mathbf{f}^w + \mathbf{f}^c + \mathbf{f}^{base}$$

where  $\mathbf{M}$  is a diagonal mass matrix and  $\mathbf{q}$  is the position vector.  $\mathbf{f}^m$ ,  $\mathbf{f}^g$ ,  $\mathbf{f}^w$ , and  $\mathbf{f}^c$  are muscle forces, vertical forces by gravity and buoyancy, drag forces, and constraint forces in order.  $\mathbf{f}^{base}$  is the force applied only on base and rotates it.

The arm's weight in water can be

$$f^g = (\rho_{arm} - \rho_{water}) \mathbf{V}_{arm} \vec{g}$$

where  $\rho$  is density and  $\mathbf{V}_{arm}$  is a matrix of segment volumes.  $\vec{g}$  is the gravitational acceleration.

The constraint force should counteract all changes in shape that lead to changes in the area of a compartment. Thus,

$$f^c = \mathbf{C}p$$

where  $\mathbf{C}$  is the matrix that we can get using Lagrange multiplier corresponding to the compartmental pressures. The pressure vector is

$$p = (\mathbf{GM}^{-1}\mathbf{C})^{-1} \left[ \gamma - \mathbf{GM}^{-1}(f^m + f^g + f^w + f^{base}) \right].$$

The definition and derivation of the matrix  $\mathbf{G}$  and the vector  $\gamma$  are on Appendix B in Yekutieli's paper [6].

The drag force is dependent not only the velocity, but also Reynolds number indirectly. Reynold number is the dimensionless ration of inertial to viscous forces, and it is given by velocity multiplied by length, and divided by kinematic viscosity of the water. When we decompose the movement of a segment into x and y axis, the drag components in stead flow are

$$\|\vec{d}_{per}\| = \frac{1}{2} \rho_{water} P_a c_{per} \|\vec{v}_{[per]}\|^2$$

$$\|\vec{d}_{tan}\| = \frac{1}{2} \rho_{water} S_a c_{tan} \|\vec{v}_{[tan]}\|^2$$

where  $P_a$  is the projected area of the segment and  $S_a$  is the surface area.

Finally, the muscle force can be modeled linearly like below.

$$f^m = (k_0 + k_{max}a(t))(l(t) - l_{rest}) + c \frac{dl(t)}{dt}$$

$l(t)$  means the length at time  $t$ , and  $l_{rest}$  is the rest length when forces are not applied.  $a(t)$  is an activation function that controls the stiffness  $k_0$  and  $k_{max}$ .

## 3.2 Simplified Actions

For the experiment, I adapted the simplified fixed set of actions defined by Yaakov Engel [5]. Controlling predefined groups of muscles, Engel simplified complex and high-dimensional muscular actions to six actions (Figure 2). Listing 1 shows the code defining six actions.

Listing 1: Defining simplified actions

```

1 #simplified actions
2 actions <- seq(1,6) # action model by Yaakov Engel
3 #activations
4 nAction <- length(actions)
5 base_activation <- rep(0,nAction)
6 activation <- matrix(0,nAction,NC1*3)
7 # ventral, dorsal, transversal
8 activation[1,] <- rep(c(0.0,0.7,0.1),NC1)
9 activation[2,] <- rep(c(0.7,0.0,0.1),NC1)
10 activation[3,] <- c(rep(c(0.5,0.5,0.5),round(NC1/2)),rep(c(0.3,0.3,0.3),NC1-round(
    NC1/2)))
11 activation[4,] <- c(rep(c(0.0,0.7,0.1),round(NC1/3)),rep(c(0.7,0.0,0.1),NC1-round(
    NC1/3)))
12 activation[5,] <- c(rep(c(0.7,0.0,0.1),round(NC1/3)),rep(c(0.0,0.7,0.1),NC1-round(
    NC1/3)))

```

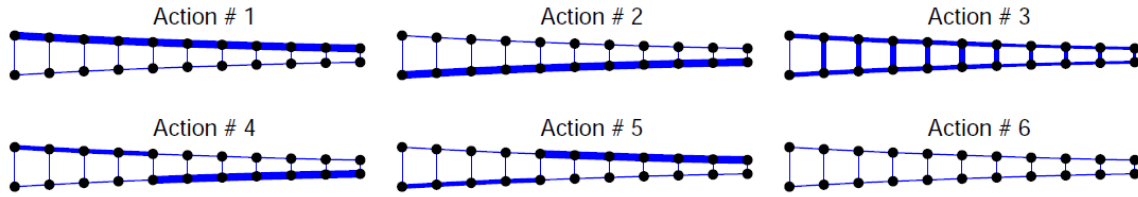


Figure 2: The simplified actions [5]

### 3.3 Method

This section shows the implementation all forces. First, I defined state vector with x-y positions and velocities on dorsal and ventral muscles. In case of  $n$  compartments, a state had  $8n$  variables. Listing ?? showed the utility function to retrieve the necessary variables.

Listing 2: Getting position and velocity

```

1 getPosVel <- function(s)
2 {
3     xvent <- s[XvMIN:XvMAX]
4     yvent <- s[YvMIN:YvMAX]
5     xdors <- s[XdMIN:XdMAX]
6     ydors <- s[YdMIN:YdMAX]
7     dxvent <- s[dXvMIN:dXvMAX]
8     dyvent <- s[dYvMIN:dYvMAX]
9     dxdors <- s[dXdMIN:dXdMAX]
10    dydors <- s[dYdMIN:dYdMAX]
11
12    list(xdors=xdors, ydors=ydors, xvent=xvent, yvent=yvent,
13         dxdors=dxdors, dydors=dydors, dxvent=dxvent, dyvent=dyvent)
14 }

```

The position of the next state is determined by the calculated forces. Listing 3 shows the code for function `nextState()`.

Listing 3: Next State

```

1 nextState <- function(s,a) {
2
3     ## Euler integration time step
4     deltaT <- 0.001
5
6     fbase <- calcFbaseForces(s, masses, base_activation[a])
7     fw <- calcWaterForces(s, trans_len)
8     fm <- calcMuscleForces(s, trans_len, a)
9     fc <- calcConstraintsForces(s, trans_len, masses, fw, fm, fg, fbase)
10
11    fracMasses <- NULL
12    for(i in 1:NC1)
13        fracMasses <- c(fracMasses, rep(masses[i],4))
14
15    # calculate acceleration
16    accel <- (fm + fg + fw + fc + fbase) / fracMasses
17    lacc <- length(accel)
18
19    # calculate velocity
20    s[dXvMIN:dXvMAX] <- s[dXvMIN:dXvMAX] + deltaT * accel[seq(1,lacc,by=4)] #
    dxvent

```

```

21     s[dYvMIN:dYvMAX] <- s[dYvMIN:dYvMAX] + deltaT * accel[seq(2,lacc,by=4)] #
      dyvent
22     s[dXdMIN:dXdMAX] <- s[dXdMIN:dXdMAX] + deltaT * accel[seq(3,lacc,by=4)] #
      dxdors
23     s[dYdMIN:dYdMAX] <- s[dYdMIN:dYdMAX] + deltaT * accel[seq(4,lacc,by=4)] #
      dydors
24     # calculate position
25     s[XvMIN:XvMAX] <- s[XvMIN:XvMAX] + deltaT * s[dXvMIN:dXvMAX] # xvent
26     s[YvMIN:YvMAX] <- s[YvMIN:YvMAX] + deltaT * s[dYvMIN:dYvMAX] # yvent
27     s[XdMIN:XdMAX] <- s[XdMIN:XdMAX] + deltaT * s[dXdMIN:dXdMAX] # xdors
28     s[YdMIN:YdMAX] <- s[YdMIN:YdMAX] + deltaT * s[dYdMIN:dYdMAX] # ydors

30     ## Return new state
31     s
32 }

```

calGravityForces() is the function that is called once at first, and the results such as mass,  $f^g$  and area do not change during the test.

#### Listing 4: Gravity

```

2 calcGravityForces <- function(tlen, state)
3 {
4     pv <- getPosVel(state)

6     # calc (constant) area
7     area <- 0.5*((pv$xvent[2:NC1]-pv$xdors[1:NC1-1])*(pv$ydors[2:NC1]-pv$yvent
      [1:NC1-1]) +
8         (pv$xdors[2:NC1]-pv$xvent[1:NC1-1])*(pv$ydors[1:NC1-1]-pv$yvent
      [2:NC1]))
9     tarea <- c(area,0) # void the second term in masses

11    # calc mass - a function of the current area and the previous component's
      area
12    masses <- c(FIRST_MASS,
13                0.5*0.5*arm_sw*
14                ( tarea[1:(length(tarea)-1)]*0.5*(tlen[1:
      NC1-1] + tlen[2:NC1]) +
15                tarea[2:length(tarea)]*0.5*(tlen[2:NC1]
      + c(tlen[3:NC1],0)) ) # 2nd term is
      void at last

16    )

18    # calc weight
19    tw <- c(0, g*masses[2:NC1]*(arm_sw-water_sw)/arm_sw)
20    w <- diff(tw,1)
21    sum_w <- sum(w)
22    # The first component's weight cancel the gravitational forces
23    # of all other components, in order to save the arm connected to
24    # the body
25    w <- c(w[length(w)]-sum_w,w)

27    fg <- NULL
28    for(i in 1:NC1) {
29        # assign weight in gravity forces matrix
30        fg <- c(fg, 0, w[i], 0, w[i])
31    }

33    list(area=area, fg=fg, masses=masses)

```

34 }

`calcMuscleForces()` implemented the linear muscle force model. Three separate muscle forces are calculated separately and composed into x-y components of ventral and dorsal points.

Listing 5: Muscle force

```
1 calcMuscleForces <- function(state, tlen, act)
2 {
3     pv <- getPosVel(state)
4
5     # ventral, dorsal length, & transversal length
6     ventral_length<-sqrt(diff(pv$xvent,1)^2 + diff(pv$yvent,1)^2)
7     dorsal_length <-sqrt(diff(pv$xdors,1)^2 + diff(pv$ydors,1)^2)
8     transversal_length<-sqrt((pv$xdors-pv$xvent)^2 + (pv$ydors-pv$yvent)^2)
9     # dl(t)/dt - normalized
10    ventral_velocity <- (diff(pv$xvent,1)*diff(pv$dxvent,1) + diff(pv$yvent,1)
11        *diff(pv$dyvent,1))/ventral_length
12    dorsal_velocity <- (diff(pv$xdors,1)*diff(pv$dxdors,1) + diff(pv$ydors,1)*
13        diff(pv$dydors,1))/dorsal_length
14    transversal_velocity <- ((pv$xdors-pv$xvent)*(pv$dxdors-pv$dxvent)+
15        (pv$ydors-pv$yvent)*(pv$dydors-pv$
16        dyvent)) / transversal_length
17
18    ventral_length <- c(ventral_length,0)
19    dorsal_length <- c(dorsal_length,0)
20    ventral_velocity <- c(ventral_velocity,0)
21    dorsal_velocity <- c(dorsal_velocity,0)
22
23    # k - the spring constant, considering the activation (excluding k0)
24    k_ventral<-k_max_horizontal * activation[act, seq(1,3*NC1,by=3)]
25    k_dorsal<-k_max_horizontal * activation[act, seq(2,3*NC1,by=3)]
26    k_transversal<-k_max_transversal * activation[act, seq(3,3*NC1,by=3)]
27
28    # fsraw - temporary variable
29    fsraw_ventral<-(k_ventral + k0_horizontal)*(ventral_length-length0[seq(1,3
30        *NC1,by=3)]) + alpha_horizontal * ventral_velocity
31    fsraw_dorsal<-(k_dorsal + k0_horizontal)*(dorsal_length-length0[seq(2,3*
32        NC1,by=3)]) + alpha_horizontal * dorsal_velocity
33    fsraw_transversal<-(k_transversal + k0_transversal)*(transversal_length-
34        length0[seq(3,3*NC1,by=3)]) + alpha_transversal * transversal_velocity
35
36    # Check compartments contracts
37    ventminIdx <- which(ventral_length/length0[seq(1,3*NC1,by=3)] < 1.0 )
38    dorsminIdx <- which(dorsal_length/length0[seq(2,3*NC1,by=3)] < 1.0)
39    transminIdx <- which(transversal_length/length0[seq(3,3*NC1,by=3)] < 1.0)
40    fsraw_ventral[ventminIdx]<-0
41    fsraw_dorsal[dorsminIdx]<-0
42    fsraw_transversal[transminIdx]<-0
43
44    # scaling?
45    fs_xvent<-fsraw_ventral[1:(NC1-1)]*diff(pv$xvent, 1)/ventral_length[1:(NC1
46        -1)]
47    fs_yvent<-fsraw_ventral[1:(NC1-1)]*diff(pv$yvent, 1)/ventral_length[1:(NC1
48        -1)]
49    fs_xdors<-fsraw_dorsal[1:(NC1-1)]*diff(pv$xdors, 1)/dorsal_length[1:(NC1
50        -1)]
51    fs_ydors<-fsraw_dorsal[1:(NC1-1)]*diff(pv$ydors, 1)/dorsal_length[1:(NC1
52        -1)]
53    fs_xvent <- c(fs_xvent,0)
54    fs_yvent <- c(fs_yvent,0)
```

```

44     fs_xdors <- c(fs_xdors,0)
45     fs_ydors <- c(fs_ydors,0)

47     fs_xtrans<-fsraw_transversal*(pv$xdors-pv$xvent)/transversal_length
48     fs_ytrans<-fsraw_transversal*(pv$ydors-pv$yvent)/transversal_length

50     fm <- c((fs_xtrans[1]+fs_xvent[1]),
51             (fs_ytrans[1]+fs_yvent[1]),
52             (-fs_xtrans[1]+fs_xdors[1]),
53             (-fs_ytrans[1]+fs_ydors[1]))
54 }

```

calcFbaseForces() computes the force to rotate the base in certain degree.

Listing 6: Base rotation

```

1 calcFbaseForces <- function(state, masses, force)
2 {
3     pv <- getPosVel(state)
4     slow_base<-0.4

6     x1 <- pv$xvent[1] - pv$xdors[1]
7     y1 <- pv$yvent[1] - pv$ydors[1]
8     dx1 <- pv$dxvent[1]
9     dy1 <- pv$dyvent[1]
10    fbase <- rep(0, 4*NC1)

12    r <- sqrt(x1^2 + y1^2)
13    if(r==0) r<- 0.0000001
14    v <- sqrt(dx1^2+dy1^2)

16    x1 <- x1 / r
17    y1 <- y1 / r

19    if(v!=0){
20        dx1 <- dx1 / v
21        dy1 <- dy1 / v
22    } else {
23        dx1 <- 0
24        dy1 <- 0
25    }

27    fcen_x <- -1*masses[1]*v*v/r*2*x1
28    fcen_y <- -1*masses[1]*v*v/r*2*y1

30    fbase[1:4] <- c(force*-y1 - slow_base*dx1 + fcen_x,
31                  force * x1 - slow_base*dy1 + fcen_y,
32                  force * y1 + slow_base*dx1 - fcen_x,
33                  force * -x1 + slow_base*dy1 - fcen_y)
34    fbase
35 }

```

calcWaterForces computes the drag forces generated when the arm moves in water.

Listing 7: Water force

```

1 calcWaterForces <- function(state, tlen)
2 {
3     pv <- getPosVel(state)

5     NI <- 1e-6 #kinetic velocity

```

```

6     x <- (pv$xvent + pv$xdors)/2
7     y <- (pv$yvent + pv$ydors)/2
8     dx <- (pv$dxvent + pv$dxdors)/2
9     dy <- (pv$dyvent + pv$dydors)/2
10    fw <- c(0,0,0,0) # no water force on the base

12    xv <- (pv$xvent[2:NC1]+pv$xvent[1:(NC1-1)])/2
13    yv <- (pv$yvent[2:NC1]+pv$yvent[1:(NC1-1)])/2
14    xd <- (pv$xdors[2:NC1]+pv$xdors[1:(NC1-1)])/2
15    yd <- (pv$ydors[2:NC1]+pv$ydors[1:(NC1-1)])/2

17    # vertical length between mid points in each component
18    # example:
19    # * * *
20    #   |
21    # * * *
22    wa <- sqrt((xd-xv)^2+(yd-yv)^2)

24    # horizontal length between mid points in each component
25    # example:
26    # * * *
27    # *-----*
28    # * * *
29    m <- sqrt((x[2:NC1]-x[1:(NC1-1)])^2 + (y[2:NC1]-y[1:(NC1-1)])^2)
30    sum_m <- sum(m)

32    cosmid <- (x[2:NC1]-x[1:(NC1-1)])/m
33    sinmid <- (y[2:NC1]-y[1:(NC1-1)])/m
34    Vmidx <- (dx[2:NC1]-dx[1:(NC1-1)])/2
35    Vmidy <- (dy[2:NC1]-dy[1:(NC1-1)])/2

37    # Velocities - perpendicular and tangent
38    Vper <- Vmidy*cosmid-Vmidx*sinmid
39    Vtan <- Vmidy*sinmid+Vmidx*cosmid

41    # average transversal length
42    d_seg <- 0.5*tlen[1:(NC1-1)] + tlen[2:NC1]
43    # Area
44    A <- d_seg * m
45    S <- 2*m*(d_seg+wa)

47    # Calc Cper and Ctan - constants depend on the component's shape and
      velocity.
48    # make sure we don't divide by zero
49    zeroIdx <- which(Vper==0)
50    nZeros <- length(zeroIdx)
51    Cper <- rep(0,length(Vper))
52    if(nZeros<length(Vper))
53        Cper[-zeroIdx] <- 1+10*((d_seg[-zeroIdx]*abs(Vper[-zeroIdx])/NI)
      ^(-2/3))

55    zeroIdx <- which(Vtan==0)
56    nZeros <- length(zeroIdx)
57    Ctan <- rep(0,length(Vtan))
58    if(nZeros<length(Vtan))
59        Ctan[-zeroIdx] <- 0.64*((sum_m*abs(Vtan[-zeroIdx])/NI)^(-0.5))

61    # Calc Dper and Dtan - drag forces
62    Dper <- 0.5 * water_sw * A * Cper * Vper^2

```

```

63     Dtan <- 0.5 * water_sw * S * Ctan * Vtan^2
64
65     # Calc Dx and Dy - drag in the x/y direction
66     Dx <- -sign(Vtan) * Dtan * cosmid + sign(Vper) * Dper * sinmid
67     Dy <- -sign(Vtan) * Dtan * sinmid - sign(Vper) * Dper * cosmid
68
69     # Assign the water force
70     lenD <- length(Dx)
71     fw1 <- 0.25*(Dx[2:lenD] + Dx[1:lenD-1])
72     fw1 <- c(fw1, 0.25*Dx[lenD])
73     fw2 <- 0.25*(Dy[2:lenD] + Dy[1:lenD-1])
74     fw2 <- c(fw2, 0.25*Dy[lenD])
75     lenfwn <- length(fw1)
76
77     for(i in 1:lenfwn)
78         fw <- c(fw, fw1[i],fw2[i], fw1[i],fw2[i])
79     fw
80 }

```

calcConstraintsForces() calculates the forces that maintain the constant volume of the octopus arm.

Listing 8: Constraints Forces

```

1 calcConstraintsForces <- function(state, tlen, masses, fw, fm, fg, fbase)
2 {
3     pv <- getPosVel(state)
4
5     # cal GM
6     GM <- matrix(0, NC1-1, 4*NC1)
7     A <- matrix(0, NC1-1, NC1-1)
8     C <- matrix(0, 4*NC1, NC1-1)
9
10    gm<-matrix(0,8,NC1-1)
11    lc<-matrix(0,8,NC1-1)
12
13    tmp<-matrix
14    gm[1,] <- (pv$yvent[2:NC1]-pv$ydors[1:(NC1-1)])/masses[1:(NC1-1)]
15    gm[2,] <- (pv$xdors[1:(NC1-1)]-pv$xvent[2:NC1])/masses[1:(NC1-1)]
16    gm[3,] <- (pv$yvent[1:(NC1-1)]-pv$ydors[2:NC1])/masses[1:(NC1-1)]
17    gm[4,] <- (pv$xdors[2:NC1]-pv$xvent[1:(NC1-1)])/masses[1:(NC1-1)]
18    gm[5,] <- (pv$ydors[2:NC1]-pv$yvent[1:(NC1-1)])/masses[2:NC1]
19    gm[6,] <- (pv$xvent[1:(NC1-1)]-pv$xdors[2:NC1])/masses[2:NC1]
20    gm[7,] <- (pv$ydors[1:(NC1-1)]-pv$yvent[2:NC1])/masses[2:NC1]
21    gm[8,] <- (pv$xvent[2:NC1]-pv$xdors[1:(NC1-1)])/masses[2:NC1]
22
23    lc[1,] <- -0.5*(-pv$ydors[1:(NC1-1)]+pv$yvent[2:NC1])
24    lc[2,] <- -0.5*(pv$xdors[1:(NC1-1)]-pv$xvent[2:NC1])
25    lc[3,] <- -0.5*(pv$yvent[1:(NC1-1)]-pv$ydors[2:NC1])
26    lc[4,] <- -0.5*(-pv$xvent[1:(NC1-1)]+pv$xdors[2:NC1])
27    lc[5,] <- -0.5*(pv$ydors[2:NC1]-pv$yvent[1:(NC1-1)])
28    lc[6,] <- -0.5*(-pv$xdors[2:NC1]+pv$xvent[1:(NC1-1)])
29    lc[7,] <- -0.5*(-pv$yvent[2:NC1]+pv$ydors[1:(NC1-1)])
30    lc[8,] <- -0.5*(pv$xvent[2:NC1]-pv$xdors[1:(NC1-1)])
31
32    a_diag <- colSums(gm*lc)
33    a_upright <- colSums(gm[5:8,1:(NC1-2)] * lc[1:4,2:(NC1-1)])
34    a_downleft <- colSums(gm[1:4,2:(NC1-1)] * lc[5:8,1:(NC1-2)])
35
36    for(i in 1:(NC1-1)) {
37        p <- 4*(i-1)+1

```

```

39         GM[i,seq(p,len=8)] <- gm[,i]
40         C[seq(p,len=8),i] <- lc[,i]

42         A[i,i] <- a_diag[i]
43         if (i>1)
44         {
45             A[i-1,i] <- a_upright[i-1]
46             A[i,i-1] <- a_downleft[i-1]
47         }
48     }

50     # cal pie
51     pie <- matrix(0, NC1-1, 1)

53     pie[,1] <- 2*(pv$dxvent[1:(NC1-1)]*pv$dydors[2:NC1]+
54                 pv$dxdors[1:(NC1-1)]*pv$dydors[2:NC1]-
55                 pv$dyvent[1:(NC1-1)]*pv$dxdors[2:NC1]-
56                 pv$dydors[1:(NC1-1)]*pv$dxdors[2:NC1]+
57                 pv$dxdors[2:NC1]*pv$dyvent[2:NC1]+
58                 pv$dxvent[2:NC1]*pv$dyvent[1:(NC1-1)]-
59                 pv$dydors[2:NC1]*pv$dxvent[2:NC1]-
60                 pv$dyvent[2:NC1]*pv$dxvent[1:(NC1-1)])

62     tmp1 <- matrix(fg + fw + fm + fbase, 4*NC1,1)
63     fc <- C %*% (solve(A) %*% (pie - GM %*% tmp1))
64     fc <- as.vector(fc)
65 }

```

If the arm touches the goal position with any part of the arm, the steps are over. To support this episodic task, I modified the function `reinforcement()` and `getSamples()` to stop collecting samples when it reached the goal. To avoid matrix size error when the agent reached the goal, I stored the number of steps and used it when it called `updateNN`. Listing 9 showed the changed code.

Listing 9: Changes in reinforcement learning codes

```

1 reinforcement <- function(s,s1,goal) {
2     pv <- getPosVel(s)

4     if(check_arms_tip==TRUE) start <- NC1 -1
5     else start <- 1

7     r_vent <- (pv$xvent - goal[1])^2 + (pv$yvent - goal[2])^2
8     r_dors <- (pv$xdors - goal[1])^2 + (pv$ydors - goal[2])^2
9     i_minv <- which.min(r_vent)
10    i_mind <- which.min(r_dors)

12    min_r <- min(r_vent[i_minv],r_dors[i_mind])
13    if(min_r==r_vent[i_minv]) {
14        min_v <- pv$dxvent[i_minv]^2 + pv$dyvent[i_minv]^2
15    } else {
16        min_v <- pv$dxdors[i_mind]^2 + pv$dydors[i_mind]^2
17    }

19    if ((min_r <= goal[3]^2) && (min_v <= max_velocity_at_goal^2)) {
20        switch(reward_type,
21            reward <- Big_Reward,
22            reward <- Big_Reward,
23            reward <- 1/goal[3],
24            reward <- Big_Reward,

```

```

25         reward <- Big_Reward
26     )
27 } else {
28     switch(reward_type,
29         reward <- Bad_Reward,
30         reward <- Bad_Reward,
31         reward <- 1/min_r,
32         reward <- Big_Reward * exp(-(sqrt(r_sqr)-goal[3]) / exp_
            reward_sigma_sqr),
33         reward <- Bad_Reward + abs(Bad_Reward) * exp(-(sqrt(r_sqr)
            - goal[3]) / exp_reward_sigma_sqr)
34     )
35 }
36
37 reward
38 }
39
40 getSamples <- function(net,numSamples,epsilon) {
41     ...
42     for (step in 1:numSamples) {
43         ...
44         r1 <- reinforcement(s,s1,goal)
45         ...
46         if(r1==10) break
47     }
48     list(X=X, R=R, Q=Q, Y=Y, AI=AI,Steps=step-1)
49 }
50
51 // Main Loop
52 ...
53 net <- updateNN(samples$X[1:samples$Steps,], samples$R[1:samples$Steps],
54                 samples$Q[1:samples$Steps], samples$Y[1:samples$Steps],
55                 samples$AI[1:samples$Steps],
56                 net, gamma=gamma,lambda=lambda, fPrec=1e-8,nIter=
                    maxIterations)
57 ...

```

### 3.4 The Simulation Test

Before running reinforcement learning, first I tested the octopus arm model simulation. To observe the movement, I draw the arm and goal with function `drawOctArm`.

Listing 10: Drawing octopus arm

```

1 drawOctArm <- function(state, goal=c(-1,-1,0.1))
2 {
3     #if(bdraw==FALSE) return(invisible(NULL))
4
5     xvent <- state[XvMIN:XvMAX]
6     yvent <- state[YvMIN:YvMAX]
7     xdors <- state[XdMIN:XdMAX]
8     ydors <- state[YdMIN:YdMAX]
9
10    plot(xdors,ydors,type="l",xlim=xrange, ylim=yrange)
11    lines(xvent,yvent,type="l")
12    lines(c(xdors[NC1],xvent[NC1]), c(ydors[NC1],yvent[NC1]),type="l")
13
14    scalex <- xrange[1]-xrange[2]
15    if(goal[1]!=-1)

```

```
16         points(goal[1],goal[2], pch=21, bg="pink", cex=2)
17     }
```

Figure 3 showed the current simulation screen. The some flaws in implementation of physics caused breaking the arm from the tip. The breaking propagated to the base as the simulation progressed. I tried hard to fix this problem, but I could not figure out what caused this. I have a few guesses. Miscalculated masses could make the light tip move faster than it has to. A flaw in muscular equation could cause dorsal and ventral points to move in opposite direction. Finally, The calculation in the volume constraints function was also responsible for maintaining the shape. I should investigate these parts further.

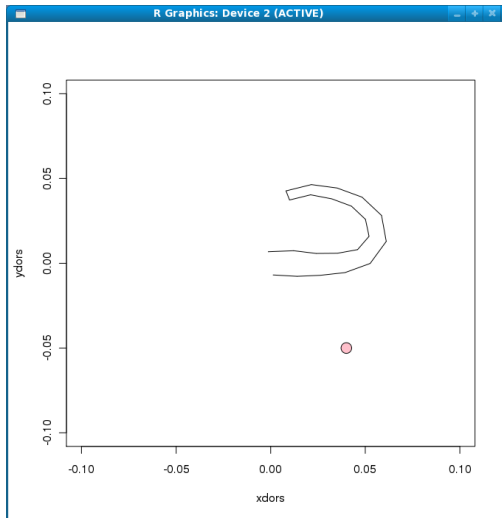
## 4 Conclusion

Experiencing not only the modeling itself but testing the complex octopus arm, I learned a lot about abstraction and accuracy. Even the basic forces applied to the small and simplified octopus arm have many parameters to consider. Some mistakes or wrong parameters can cause the target not to work properly. I learned how important simplifying the problem is as well as the importance of accurate model. Building a simple model and not losing the expected behavior is extremely important and would affect the performance of learning process.

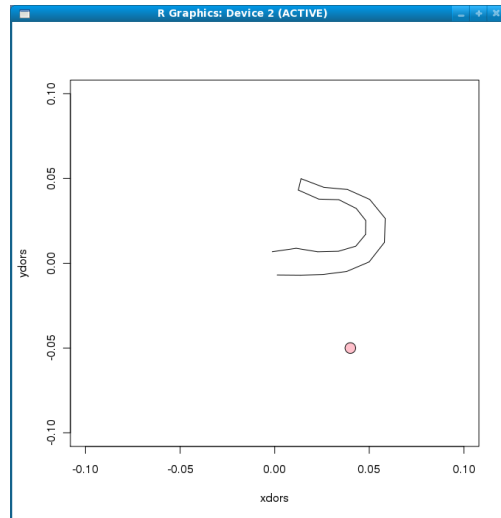
Since the problem in the octopus arm model, I could not proceed the reinforcement learning experiments. To complete the test I planned, giving up current time-stacking approach, test and find, I should recreate a descent octopus arm model from the scratch. After that, I will be able to test reinforcement learning algorithms such as TD( $\lambda$ ), LSTD( $\lambda$ ), and GPTD. I would like to extend this learning agent to intelligent one by adding planning methods.

## References

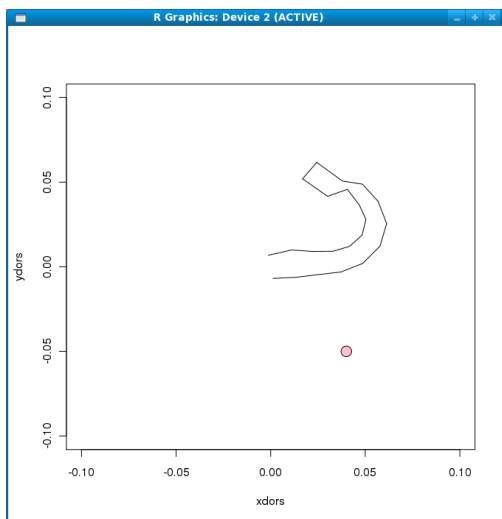
- [1] Tongues, tentacles and trunks: the biomechanics of movement in muscular-hydrostats. *Zoological Journal of the Linnean Society*, 83, 1985.
- [2] Justin Boyan. Least-squares temporal difference learning. In *In Proceedings of the Sixteenth International Conference on Machine Learning*, pages 49–56. Morgan Kaufmann, 1999.
- [3] Steven J. Bradtke, Andrew G. Barto, and Pack Kaelbling. Linear least-squares algorithms for temporal difference learning. In *Machine Learning*, pages 22–33, 1996.
- [4] Yaakov Engel, Shie Mannor, and Ron Meir. Reinforcement learning with gaussian processes. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 201–208, New York, NY, USA, 2005. ACM.
- [5] Yaakov. Learning to control an octopus arm with gaussian process temporal difference methods. 2006.
- [6] Yoram Yekutieli, Roni Sagiv-zohar, Ranit Aharonov, Yaakov Engel, Binyamin Hochner, and Tamar Flash. A dynamic model of the octopus arm. i. biomechanics of the octopus reaching movement. *journal of neurophysiology* (in press. *J Neurophysiol*, 5:291–323, 2005.



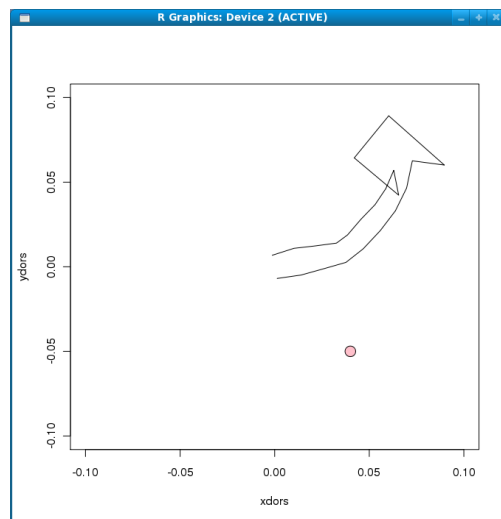
(a)



(b)



(c)



(d)

Figure 3: Simulating octopus arm movement