Non-Symbolic AI: Computing Exercise

# Cart pole control

Thomas Pitschel

May 30, 2003

# Contents

# 1 Introduction

This coursework deals with the control of a cart-pole system. In our case this is a pole fixed at a cart through a joint. The cart may move only on a line perpendicular to the joint axis. The pole is supposed to be kept upright for as long as possible by moving the cart in the appropriate direction. To this end, the controller decides the sign of the force to be applied at the cart based on the current state of the cart-pole system (cart position and velocity, pole angle and angle velocity). The cart position is restricted to a prespecified interval on the line of movement.

Different controller implementations are possible, among which are:

- a neural network that has its weights updated by a GA
- a Q-learned agent
- a look-up table whose entries

In this coursework I have implemented the agent and the look-up table, since both are computationally similiar in that they work on a discrete version of the cart state space.

Some results...

# 2 Methods used

For both methods the cart state space was discretized by partitioning into intervals as recommended in the coursework sheet, i.e. $x$: $+-0.8, +-2.4m$, $\dot{x}$: $+-0.5ms^{-1}$, $\theta$: $0, +-1, +-6, +-12°$ and $\dot{\theta}: 50° s^{-1}$.

The simulation was started with the cart placed at $x = 1.0$ and pole angle and angle velocity and cart velocity set to zero. It was stopped when the pole fell down ($|\theta| > 50°$) or the cart moved out of its allowed interval ($|x| > 2.4$) or the number of steps reached 20000. For the GA method the number of steps run was recorded as fitness.

To better explore the state space the decision made by the controller was made noisy by interverting it with probability 0.0001.

## 2.1 Q-learned agent

## 2.2 Introduction to Q-learning

One way of controlling the cart is using an agent that learns appropriate behaviour by trying various actions and receiving a reinforcement signal to adapt its decisions. The area concerned with the construction of those agents is called reinforcement learning (RL). In the RL framework the environment in which the agents behaves is modelled as an automat that changes its state (non-deterministically) under influence of the agent's actions. Every time step $t$ the agent receives a reward $r_t$ that depends (non-deterministically) on the current state of the environment and the action taken. The problem faced by the agent is how to choose its actions at every time step such that it maximizes the *accumulated* rewards.

Our controlling task is solved in this framework as follows: reward every action that makes the simulation stop with $-1.0$ and every other action with 0.0. Then, choosing an infinite-horizon discounted model for accumulating the rewards, i.e.

$$maximize E(\sum_t r_t \cdot \gamma^t)$$

with discounting factor $\gamma$, the agent will adapt to run the cart-pole system for as long as possible. The RL framework solves the adaption task by introducing a function $V^*(s)$ that contains the expected accumulated reward when starting in state $s$ and choosing the optimal actions in future time steps. If this function is known the agent can choose always the action that leads to the state with the highest $V^*$ value. According to [1] the following equation holds for every state $s$:

$$V^*(s) = max_a \left( R(s,a) + \gamma \sum_{s'} T(s,a,s') V^*(s') \right)$$

where $T(s,a,s')$ is the probability of the environment changing state to $s'$ when the agents chooses action $a$ in state $s$, and $R(s,a)$ the expected reward. This can be expressed equivalently by

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s'} T(s,a,s') max_{a'} Q^*(s',a')$$

for all $s$ and $a$. To approximate the $Q^*$ function one uses an array $Q$ which is updated according to

$$Q(s,a) := Q(s,a) \cdot (1-\alpha) + (r(s,a) + \gamma \cdot max_{a'} Q(s',a')) \cdot \alpha$$

while the agent experiences the state change $s - s'$ using action $a$ and receiving $r(s,a)$. That is the Q-learning algorithm.

### 2.2.1 My strategies

For exploring the state space I used a Boltzmann similar strategy. Action 0 was chosen when a $[0,1]$-uniformly distributed random variable was smaller than $(Q(s,0) - Q(s,1)) \cdot k$, Action 1 otherwise. ($k$ is a parameter which was initially set to 25 to increase the exploration of the state space and after 2000 steps increased enough to make the decision deterministic.)

### 2.3 GA on look-up table

As a second method I used the same Q array as before but evolved the entries by a GA instead of Q-learning them. The decision rule stayed the same. (Choose action with higher Q value.)

I used a Steady State GA with Tournament Selection for implementation simplicity. (Two pairs of individuals are picked, the fitter one in each pair becomes a parent.)

The recombination was just the average of the parent Q arrays. This makes sense since it makes the decision of the child on a state less evolutionary stable when the decisions of the two parents are conflicting on that state.

Mutation was applied to the child by adding a real value uniformly distributed in $[-0.01, 0.01]$.

## 3 Results

Various values for $\alpha$ and $\gamma$ were tried. Best values in the end were $\alpha = 0.1$ and $\gamma = 0.99$. The Q-learned agent adapted to run the cart-pole in most of the cases for longer than 20000 steps.

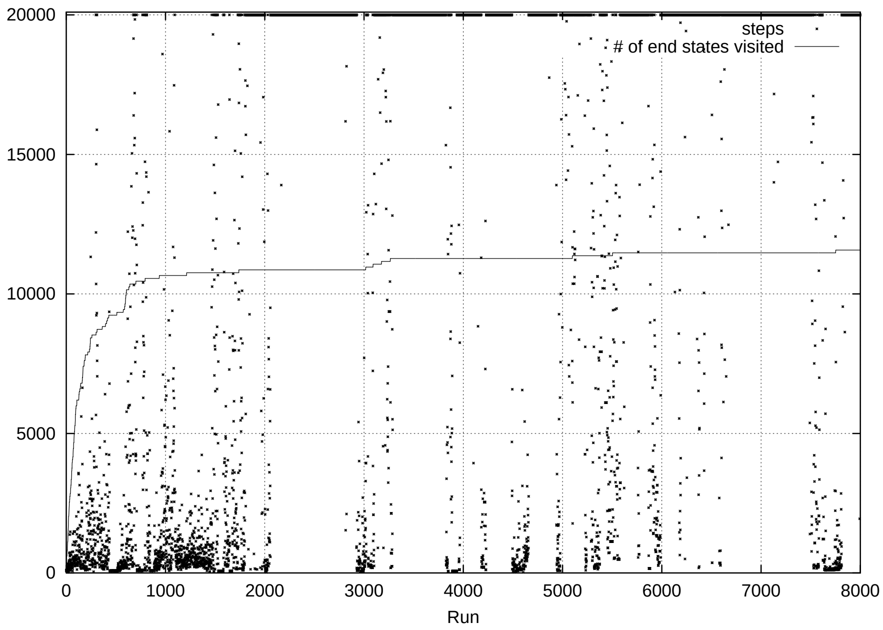


Figure 1: The duration of each run and the accumulated number of end states explored for Q-learning

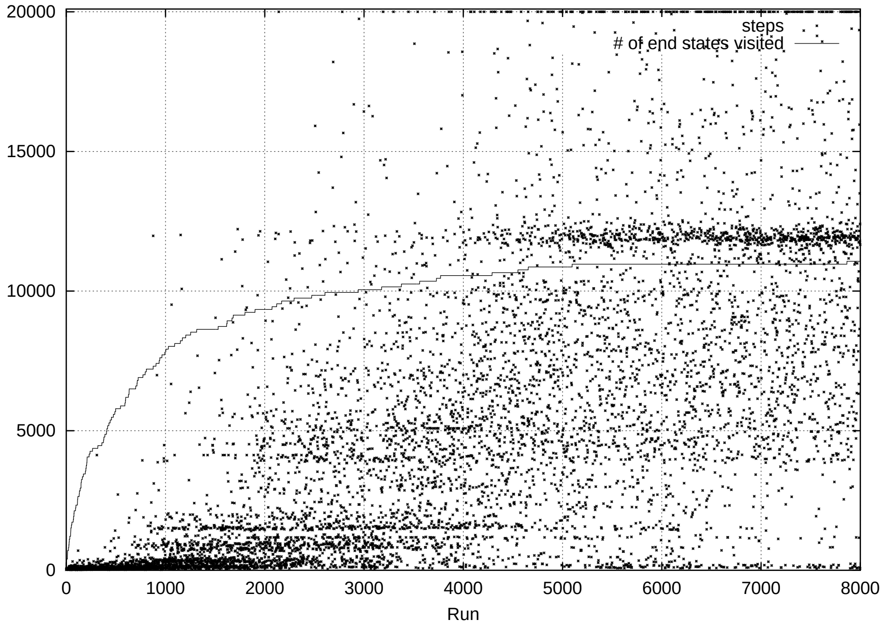


Figure 2: The duration of each run and the accumulated number of end states explored for the GA omitting decision noise

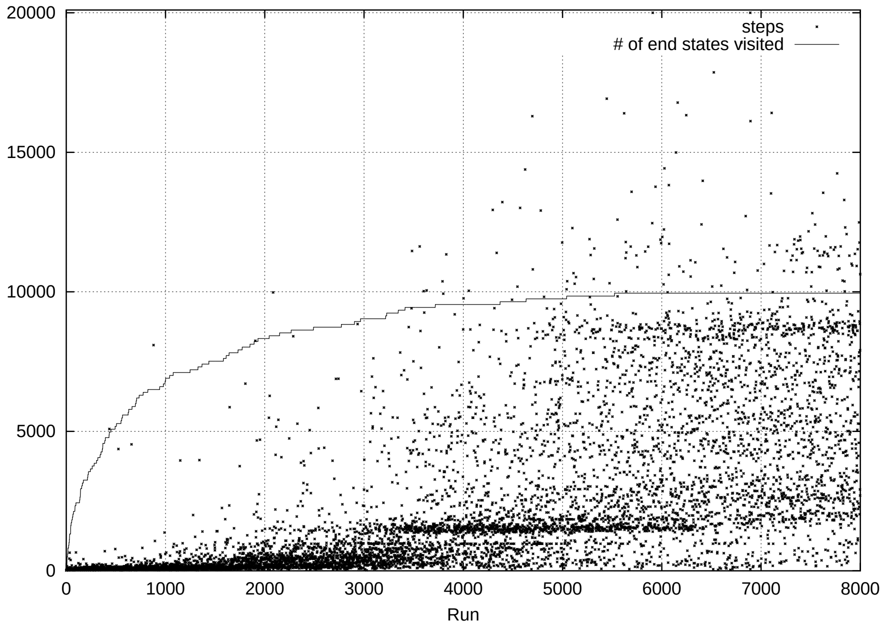


Figure 3: The duration of each run and the accumulated number of end states explored for the GA

Interesting is that the state space is better covered by the Q-learned agent. I recorded the index of the state when the simulation was stopped due to step bigger than 20000:

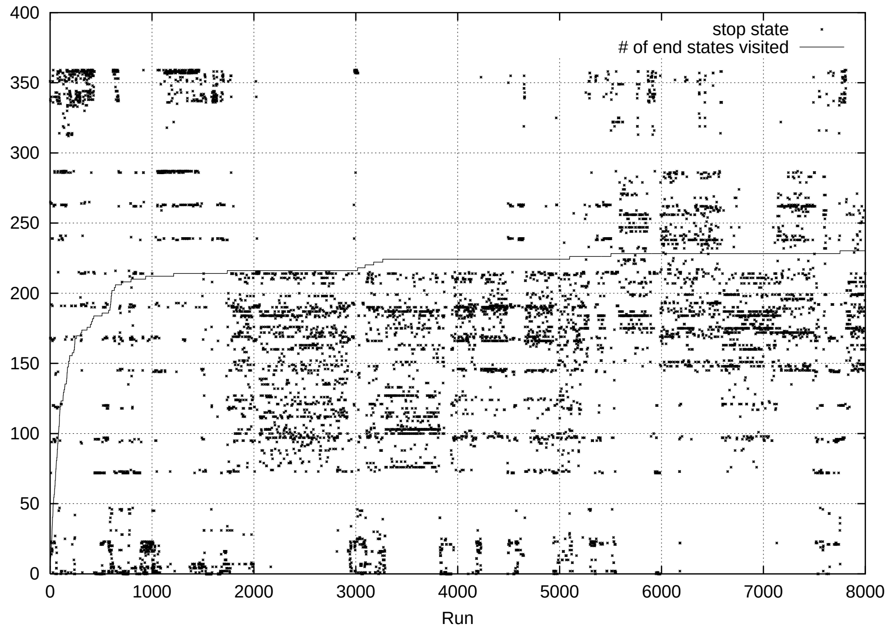


Figure 4: The index of the stopState for every run for Q-learning

This shows that the performance results of GAs may rely on only a small number of stop states. One definitely has to introduce some randomness in the initial conditions of the cart-pole system.

# 4 Implementation details

The **boxes array (= Q array)** was implemented as one linear array, with the following index mapping

$$Q(CartState, Action) = \texttt{Q[Action + 2*getState(CartState)]}, \quad (1)$$

where **Action** is 0 (force -1) or 1 (force 1). **getState()** implements the partition of the cart state space into boxes and returns a integer unique to each box. The following formula is used:

$$stateNo = index(x) \cdot 3 * 8 * 3 + index(\dot{x}) \cdot 8 * 3 + index(\theta) \cdot 3 + index(\dot{\theta})$$

where $index(x)$ is $0, 1, 2, 3, 4$; $index(\dot{x})$ is $0, 1, 2$; $index(\theta)$ is $0, 1, 2, 3, 4, 5, 6, 7$; $index(\dot{\theta})$ is $0, 1, 2$ according to which interval the state variables occupy.

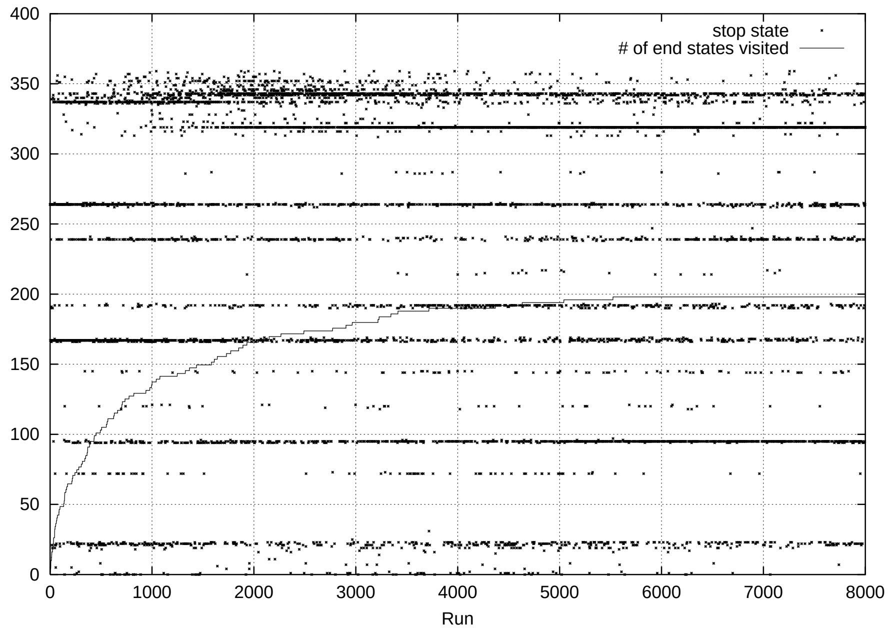


Figure 5: The index of the stopState for every run for the GA

Both the Q-learning and the genetic algorithm method were integrated into one piece of code. By setting the **opt.method** variable in main() accordingly (0=Q-lng, 1=GA), different parts of the code are switched into the execution line.

The essential part of the code specific to the **GA** can be found in main(). The essential part of the **Q-learning** can be found in runSim(). Here is a reduced version:

```
int runSim(Options opt, double *Q, int *endStateHistory, int *endStateCntr)
{
  SimulatorInit();
  for (step = 0; !outOfRange(CART); step++) {
    oldState = getState(CART);
    action = getAction(opt, Q, oldState);
    CART.force_sign = action*2.0 - 1.0;
    SimulatorStep();
    if (outOfRange(CART)) reward = -1.0;
    else                  reward =  0.0;
    updateQarray(opt, Q, oldState, action, reward, getState(CART));
  }
}
```

**getAction()** chooses the currently optimal action according to the Q array, with **updateQarray()** the Q array gets updated using the oldState, the action just taken and the new state (=getState(CART)).

The opt variable carries all parameter values into the functions.

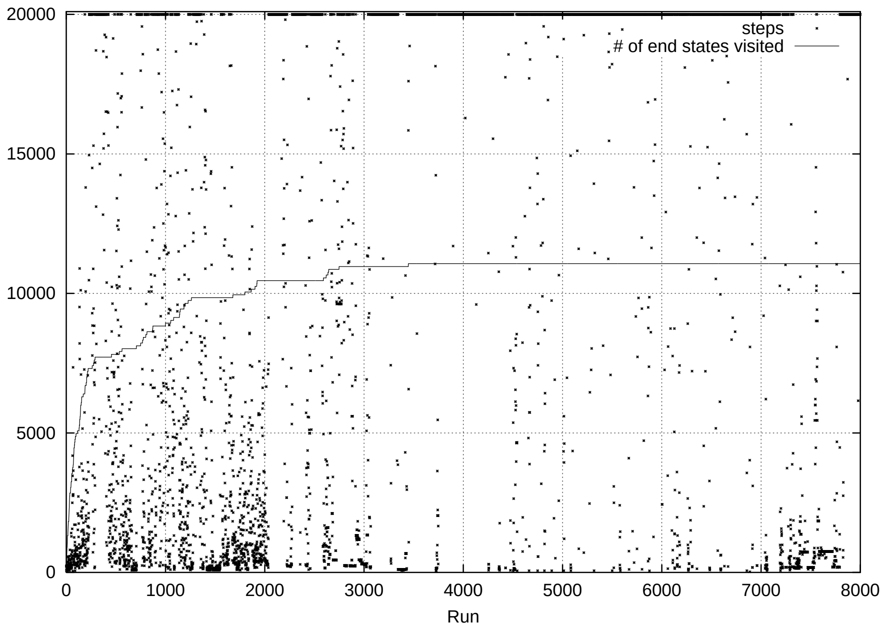


Figure 6: sdf

# 5 References

## References

[1] "Reinforcement Learning: A Survey", Leslie Pack Kaelbing, Michael L. Littman, Journal of Artificial Intelligence Research 4 (1996) 237-285

# A Printouts

## A.1 zufall.c

```
maximum number of endstates was 120

==============================================================
genetic algo on boxes, uniform mutation with +-0.01, recombination:
average, deterministic action choice according to higher Q value

  endStateCounter: 109
  (8.139319e-01,-2.161562e+00,8.747072e-01,2.831451e+00)
  (0000001011010101010101000000101011110000000000000)
run 4793 yields 8355.00 steps
  endStateCounter: 109
  (2.400212e+00,1.967726e-02,-2.820280e-02,-1.435892e-02)
  (0000111111000000111111000000111111000000111111000)
run 4794 yields 2466.00 steps
  endStateCounter: 109
  (-2.444576e+00,-2.615103e+00,-2.279840e-01,1.230613e+00)
  (1110010000011101010100001111010111110000000011010)
run 4795 yields 1724.00 steps
  endStateCounter: 109
  (2.410446e+00,4.105576e-01,-6.566863e-02,-6.254499e-01)
  (1111110000001111110000001111110000001011111101100)
run 4796 yields 5864.00 steps
  endStateCounter: 109
  (1.321441e+00,6.397054e-01,8.876525e-01,7.877892e-01)
  (1111111111000000000101010101001101010101010101010)
run 4797 yields 7082.00 steps
  endStateCounter: 109
  (2.402195e+00,4.681344e-01,-1.850296e-02,-6.263562e-01)
  (1011111101000000101111110000001111110000001111110)
run 4798 yields 11629.00 steps
  endStateCounter: 109
  (2.407808e+00,2.886144e-01,-1.651710e-02,-4.100874e-01)
  (1000000111111010000001011111101000000011111110100)
run 4799 yields 10280.00 steps

==============================================================
with the same GA:
a Q array was evolved that presumably runs the cartpole forever
in the deterministic case:

  endStateCounter: 99
  (7.874134e-01,1.804515e-02,5.012005e-02,-9.786761e-02)
  (1011111101000000101111110100000011111100000010111)
run 4772 yields 20000001.00 steps

==============================================================
//state record code (wrap around buffer)
  stateHistory[pointer] = oldState; pointer = (pointer + 1) % 80;

//update code
```

```
  if (opt.method == 2) {
    //updateQarray along the state history
    factor = -opt.gamma;
    for (i=pointer+80-1; i>=pointer; i--) {
      printf("  %d (%.2f)", stateHistory[i%80], Q[stateHistory[i%80]]);
      if (stateHistory[i%80] == -1) break;
      Q[stateHistory[i%80]] *= (1-opt.alpha);
      Q[stateHistory[i%80]] += opt.alpha * factor;
      if (stateHistory[i%80] != stateHistory[(i-1)%80])
        factor *= opt.gamma;
    }
    printf("\n");
  }
```

```
==============================================================
Q-learning, alpha 0.2, gamma 0.95, permanent chance 0.0001

0.000/0.000  -0.012/-0.112  -0.004/-0.045  -0.001/-0.027  0.000/0.000
-0.024/-0.003  -0.004/-0.045  -0.001/-0.003
  endStateCounter: 114
  (-1.165029e+00,-6.363285e-01,8.910501e-01,1.368765e+00)
  (010101011101111011011101000100011000001010100101O)
run 4793 yields 5721.00 steps
0.000/0.000  -0.012/-0.112  -0.004/-0.045  -0.002/-0.027  0.000/0.000
-0.024/-0.003  -0.004/-0.045  -0.001/-0.002
  endStateCounter: 114
  (-9.252440e-01,-4.216000e-01,8.795991e-01,1.728508e+00)
  (1100000000000000111111100100001010010101010010101)
run 4794 yields 744.00 steps
0.000/0.000  -0.012/-0.112  -0.003/-0.045  -0.001/-0.027  0.000/0.000
-0.024/-0.003  -0.003/-0.045  -0.001/-0.002
  endStateCounter: 114
  (-1.923540e+00,-6.163899e-01,8.761684e-01,1.877854e+00)
  (1101010101011111101101011001010001100000101001010)
run 4795 yields 1635.00 steps
0.000/0.000  -0.012/-0.112  -0.003/-0.045  -0.001/-0.027  0.000/0.000
-0.024/-0.003  -0.003/-0.045  -0.001/-0.002
  endStateCounter: 114
  (-1.686039e+00,-4.123638e-01,8.928660e-01,8.427104e-01)
  (1010101110111101101011001010001100000101001010101)
run 4796 yields 2704.00 steps
0.000/0.000  -0.011/-0.112  -0.004/-0.045  -0.001/-0.027  0.000/0.000
-0.024/-0.005  -0.004/-0.045  -0.001/-0.004
  endStateCounter: 114
  (-3.636303e-01,1.115760e+00,-8.966037e-01,-1.917699e+00)
  (1000001110000010101010111111111111110111100000000)
run 4797 yields 3374.00 steps
0.000/0.000  -0.010/-0.112  -0.004/-0.045  -0.001/-0.027  0.000/0.000
-0.024/-0.003  -0.004/-0.045  -0.001/-0.004
  endStateCounter: 114
  (-3.914335e-01,3.678531e-01,-8.813149e-01,-2.889163e-01)
  (0000011100001010101011111111111110111100000000000)
run 4798 yields 424.00 steps
```

```
0.000/0.000  -0.010/-0.112  -0.003/-0.045  -0.001/-0.027  0.000/0.000
-0.024/-0.003  -0.003/-0.045  -0.001/-0.003
  endStateCounter: 114
  (8.885771e-01,1.234802e+00,8.788767e-01,8.502718e-01)
  (011111010011111111000000000000000000001111111111)
run 4799 yields 1473.00 steps
```

# B Code

## B.1 zufall.c

```c
#include <stdlib.h>
#include <math.h>

int rndInt(int m)
{
  return rand()%m;
}

double rndReal01()
{
  return (rand()+0.0)/RAND_MAX;
}

double rndReal(double min, double width)
{
  return (rand()+0.0)/RAND_MAX*width+min;
}

char dice(double p)
{
  if (rndReal01() < p)
    return (1==1);
  else
    return (1==0);
}
```

## B.2 cart_pole.h

```c
typedef struct STATES {
   float theta;
   float theta_dot;
   float x;
   float x_dot;
   int force_sign;  /* +/- 1  */
} STATES;


static STATES CART;
```

### B.3 cart_pole.c

```
/* needs to be linked using -lm */

#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#include "cart_pole.h"
#include "zufall.c"

#define GRAV             -9.8   /* g */
#define MASS_C            1.0    /* mass cart */
#define MASS_P            0.1    /* mass pole */
#define LENGTH            0.5    /* half length of pole */
#define SAMPLE_INTERVAL   0.02    /* delta t */
#define PI                3.1415927
#define MU_C              0.0005   /* coeff fric, cart on track */
#define MU_P              0.000002  /* coeff frict, pole on cart */
#define FORCE            10      /* magnitude of force applied at every
                                    time step (either plus or minus) */

void calc_new_state(STATES* );
int sgn(float );

/*******************************************************************/

/* cart pole simulation, updates 4 states of system according to
equations of motion of system, approximately integrated using
simple Euler integration   */


void calc_new_state(STATES *cs)
{
  float th, th_dot, top, bottom,th_dotdot;
  float x_dot, x_dotdot,f;

  th=cs->theta;
  th_dot=cs->theta_dot;

  f=FORCE*cs->force_sign;

  top= GRAV*sin(th) + (cos(th)*(-f - (MASS_P*LENGTH*th_dot*th_dot*sin(th)) +
  MU_C*sgn(th_dot))/(MASS_C + MASS_P)) - ((MU_P * th_dot)/(MASS_P*LENGTH));

  bottom= LENGTH*((4.0/3.0) - (MASS_P*cos(th)*cos(th))/(MASS_C + MASS_P));

  th_dotdot = top/bottom;


  x_dotdot = (f + MASS_P*LENGTH*(th_dot*th_dot*sin(th)-th_dotdot*cos(th)) -
  MU_C*sgn(x_dot))/(MASS_C +MASS_P);

  cs->theta = cs->theta + SAMPLE_INTERVAL*cs->theta_dot;
```

```
  cs->theta_dot = cs->theta_dot + SAMPLE_INTERVAL*th_dotdot;
  cs->x = cs->x + SAMPLE_INTERVAL*cs->x_dot;
  cs->x_dot = cs->x_dot + SAMPLE_INTERVAL*x_dotdot;
}


/**********************************/

int sgn(float x)
{
    if(x < 0) return(-1);
    else return 1;
}

/*************************************/
void SimulatorInit()
{
  CART.x = 1;
  CART.x_dot = 0.0;
  CART.theta = 0;
  CART.theta_dot = 0;
  CART.force_sign = 1;
}

void SimulatorStep()
{
  calc_new_state(&CART);
}

//======================================================================
//======================================================================
//==== here is my code

#define ToRad(x) (x/180.0*PI)
#define nmbBoxes        360      /* adjust this if you use other box
                                    intervals */
#define nmbActions        2
#define Qsize            (nmbActions*nmbBoxes)
// action is 0 (force_sign -1) or 1 (force_sign 1)
#define MAX(a,b) ((a)>(b) ? a : b)
#define MIN(a,b) ((a)<(b) ? a : b)

#include <strings.h>

struct optionsStr {
  double method;
  double alpha;
  double gamma;
  int popSize;
  double k;
};

typedef struct optionsStr Options;
```

```
int getState(STATES cart)
  // maps cart state to an integer value that corresponds
  // to the according box
  // (by this the states get countable, use this then for the Q learning)
{
  int res=0;
  //number of boxes is 5*3*8*3 = 360

  if (cart.x < -2.4)
    res += 0;
  else
  if (cart.x < -0.8)
    res += 1;
  else
  if (cart.x <  0.8)
    res += 2;
  else
  if (cart.x <  2.4)
    res += 3;
  else
    res += 4;

  res *= 3;
  if (cart.x_dot < -0.5)
    res += 0;
  else
  if (cart.x_dot <  0.5)
    res += 1;
  else
    res += 2;

  res *= 8;
  if (cart.theta < ToRad(-12.0))
    res += 0;
  else
  if (cart.theta < ToRad(- 6.0))
    res += 1;
  else
  if (cart.theta < ToRad(- 1.0))
    res += 2;
  else
  if (cart.theta < ToRad(  0.0))
    res += 3;
  else
  if (cart.theta < ToRad(  1.0))
    res += 4;
  else
  if (cart.theta < ToRad(  6.0))
    res += 5;
  else
  if (cart.theta < ToRad( 12.0))
    res += 6;
  else
    res += 7;
```

```
  res *= 3;
  if (cart.theta_dot < ToRad(-50.0))
    res += 0;
  else
  if (cart.theta_dot < ToRad( 50.0))
    res += 1;
  else
    res += 2;

  return res;
}

char outOfRange(STATES cart)
  // returns true if pole has fallen too far or cart has run away
{
  if ((cart.x < -2.4) || (cart.x > 2.4))
    return (0==0);
  if ((cart.theta < ToRad(-50.0)) || (cart.theta > ToRad(50.0)))
    return (0==0);
  return (1==0);
}

int getAction(Options opt, double *Q, int oldState)
{
  int action;

  if (opt.method == 0) {
    // dice(x) is true with probability x
    action = (dice((Q[0 + nmbActions*oldState]
                  - Q[1 + nmbActions*oldState])*opt.k + 0.5)
              ? 0 : 1);
    if ((0==0)
   //&& (MIN(Q[0 + nmbActions*oldState], Q[1 + nmbActions*oldState]) > -0.3)
        && dice(0.0001))     // <- permanent action choice randomness
      action = 1-action;
  } else
  if (opt.method == 1) {
    action = (Q[0 + nmbActions*oldState] > Q[1 + nmbActions*oldState] ?
                0                         :   1);
    if (dice(0.0001))     // <- permanent action choice randomness
      action = 1-action;
  }
  ;

  return action;
}

void updateQarray(Options opt, double *Q, int oldState, int action,
     double r, int newState)
  // updates Q after experiencing the tuple (os, a, r, ns)
  // where os = oldState, ns = newState, a = action, r = reward
{
  if (opt.method == 0) { // Q-learning
```

```
    double Q_os_a, Q_ns_a;

    Q_os_a = Q[action + nmbActions*oldState];
    Q_ns_a = MAX(Q[0 + nmbActions*newState], Q[1 + nmbActions*newState]);
    Q[action + nmbActions*oldState] *= 1-opt.alpha;
    Q[action + nmbActions*oldState] += opt.alpha * (r + opt.gamma * Q_ns_a);
  } else
  if (opt.method == 1) { // genetic algorithm on boxes
  } else
  ;
}

void printQ(double *Q)
  // prints Q values along special axises
{
  double x;
  STATES cart;
  cart.x = 0.0;
  cart.x_dot = 0.0;
  cart.theta = 0.0;
  cart.theta_dot = 0.0;
  for (x=-3.2; x < 4.0; x+=1.6) {
    cart.x = x;
    printf("%.3f/%.3f  ", Q[0 + 2*getState(cart)], Q[1 + 2*getState(cart)]);
  }
  printf("\n");
  cart.x = 0.0;
  for (x=-1.0; x < 1.5; x+=1.0) {
    cart.x_dot = x;
    printf("%.3f/%.3f  ", Q[0 + 2*getState(cart)], Q[1 + 2*getState(cart)]);
  }
  printf("\n");
}

void recordAction(char *buff, int action)
  // records the history of recent actions
{
  char buff2[50];
  strncpy(buff2, &(buff[1]), 49);
  strcpy(buff, buff2);
  if (action == 0)
    strcat(buff, "0");
  else
    strcat(buff, "1");
}

void addEndState(int endState, int *endStateHistory, int *endStateCounter)
{
  int i;
  for (i=0; i<(*endStateCounter); i++)
    if (endState == endStateHistory[i]) return;
  endStateHistory[(*endStateCounter)] = endState;
  (*endStateCounter)++;
}
```

```
int runSim(Options opt, double *Q, int *endStateHistory, int *endStateCntr,
           int *stopState)
{
  int step;
  int action, reward;
  int oldState;
  char actionBuff[50];

  SimulatorInit();
  printQ(Q);
  strcpy(actionBuff, "**************************************************");
  for (step = 0; !outOfRange(CART); step++) {
    oldState = getState(CART);
    action = getAction(opt, Q, oldState);
    recordAction(actionBuff, action);
    CART.force_sign = action*2.0 - 1.0;
    SimulatorStep();
    if (outOfRange(CART)) reward = -1.0;
    else                  reward =  0.0;
    updateQarray(opt, Q, oldState, action, reward, getState(CART));
    if (step > 20000) {
      break;
    }
  }
  if (outOfRange(CART))
    addEndState(getState(CART), endStateHistory, endStateCntr);
  *stopState = getState(CART);
  // here would have been the code for the Q-learning variant that
  // updates the array after the run has finished
  printf("  (%e,%e,%e,%e)\n",CART.x,CART.x_dot,CART.theta,CART.theta_dot);
  printf("  (%s)\n", actionBuff);
  return step;
}

int main(void)
{
  int i,j;
  int run;
  double **Q;
  double *fitness;
  int stopState;
  Options opt;
  int pickA, pickB, mum, dad;

  // there are 5*3*8*3 = 360 boxes, of which
  //           2*3*8*3
  //         + 5*3*2*3
  //         - 2*3*2*3 = 144 + 90 - 36 = 198 are end states
  //         (at least this is an upper bound, value maybe
  //          smaller since some states may not reachable by physics)
  int endStateHistory[198];
  int endStateCntr=0;
```

```
opt.alpha = 0.1;
opt.gamma = 0.99;
opt.method = 0;
opt.popSize = 100;
opt.k = 25;

if (opt.method == 0) // Q-learning
  opt.popSize = 1;
else
if (opt.method == 1) // genetic algo on boxes
  opt.popSize = 100;
else
;

Q = (double**)malloc(opt.popSize*sizeof(double*));
fitness = (double*)malloc(opt.popSize*sizeof(double));
for (i=0; i<opt.popSize; i++) {
  Q[i] = (double*)malloc(Qsize*sizeof(double));
}

//initialize Q array,
for (i=0; i<opt.popSize; i++)
  for (j=0; j<Qsize; j++)
    Q[i][j] = 0.0;

pickA = 0;
endStateCntr = 0;
for (run=0; run<8000; run++) {
  fitness[pickA] = runSim(opt, Q[pickA], endStateHistory, &endStateCntr,
                          &stopState);
  printf("sim1key run %d steps %.0f endstatecntr %d stopState %d\n", run,
         fitness[pickA], endStateCntr, stopState);
  if (run == 2000) opt.k = 100000;
  if (opt.method == 1) { // genetic algo on boxes
    // selection, breeding and mutation (using Steady State and
    //   Tournament selection)
    pickA = rndInt(opt.popSize);
    pickB = rndInt(opt.popSize);
    mum = (fitness[pickA] > fitness[pickB] ? pickA : pickB);
    pickA = rndInt(opt.popSize);
    pickB = rndInt(opt.popSize);
    dad = (fitness[pickA] > fitness[pickB] ? pickA : pickB);
    pickA = rndInt(opt.popSize); // the one that gets overwritten
                                 // and is evaluated next time
    for (j=0; j<Qsize; j++)
      Q[pickA][j] = (Q[mum][j] + Q[dad][j])/2.0 + rndReal(-0.01, 0.02);
    for (j=0; j<nmbBoxes; j++) {
      double min = MIN(Q[pickA][0 + 2*j], Q[pickA][1 + 2*j]);
      Q[pickA][0 + 2*j] -=  min;
      Q[pickA][1 + 2*j] -=  min;
    }
  }
}
```

```
  for (i=0; i<opt.popSize; i++) {
    free(Q[i]);
  }
  free(Q);
  return 0;
}
```