Notes for simulation of traffic flow on an arbitrary network of one-way single-lane roads with traffic lights at intersections.

Charles C. Peskin
April 17, 2017
c = car index, nc = # of cars

i = intersect index, ni = # of intersects

b = block index, nb = # of blocks

i1(b), i2(b) = indices of intersects connected by block b, ordered by the direction of traffic flow. (All blocks are one-way.)

nb[i] = # of blocks entering intersect i

bin[i,j] = index of jth block entering intersect i
          j = 1 ... nb[i]

nout(i) = # of blocks leaving intersect i

bout(i,j) = index of jth block leaving intersect i
            j = 1 ... nout(i)
Note that nbin, bin can be derived from i2, and that nout, bout can be derived from i1, as follows:

\[
\text{for } i = 1:ni \\
\quad \text{nbin}(i) = \text{sum}(i2 == i) \\
\quad \text{nbout}(i) = \text{sum}(i1 == i)
\]
\[
\text{end}
\]
\[
\text{nbinmax} = \text{max}(\text{nbin}) \\
\text{nboutmax} = \text{max}(\text{nbout})
\]
\[
\bin = \text{zeros}(ni, \text{nbinmax}) \\
\bout = \text{zeros}(ni, \text{nboutmax})
\]
\[
\text{for } i = 1:ni \\
\quad \text{bm}(i, 1:nbin(i)) = \text{find}(i2 == i) \\
\quad \text{bout}(i, 1:nbout(i)) = \text{find}(i1 == i)
\]
\[
\text{end}
\]

As a check, it should be the case that

\[
\text{sum(nbin)} = \text{sum(nbout)} = nb
\]
Traffic lights

At any given time, the traffic light at intersection \( i \) is green for exactly one of the blocks that enter that intersection and red for all of the others entering that intersection.

Let \( j_{\text{green}}(i) \) be an integer designating which block has the green light, where

\[
1 \leq j_{\text{green}}(i) \leq n_{\text{bin}}(i)
\]

Let \( s(b) \) be the state of the light at the end of block \( b \), where \( s = 0 \) denotes red and \( s = 1 \) denotes green.

Given the array \( j_{\text{green}} \), \( s \) can be set as follows:

\[
s = \text{zeros}(1, nb) \\
\text{for } i = 1 : ni \\
\quad b = \text{bin}(i, j_{\text{green}}(i)) \\
\quad s(b) = 1 \\
\text{end}
\]
Geometric information about the network of roads

\( x_i(i), y_i(i) \) = coordinates of intersection \( i \)

\( L(b) \) = length of block \( b \)

\((ux(b), uy(b))\) = unit vector along block \( b \) in direction of traffic flow

Given \( x_i, y_i \), we can find \( L, ux, uy \) as follows

\[ ux = x_i(i2) - x_i(i1) \]
\[ uy = y_i(i2) - y_i(i1) \]

\[ L = \sqrt{ux \cdot ux + uy \cdot uy} \]

\[ ux = ux \div L \]
\[ uy = uy \div L \]
Cars on blocks

Let \( p(c) \) be the position of car \( c \) on whatever block it happens to be on, measured as distance from the start of the block. If car \( c \) is on block \( b \), then

\[
0 \leq p(c) < L(b)
\]

and the coordinates of car \( c \) are given by

\[
\begin{align*}
x(c) &= x_i(iL(b)) + p(c) \star ux(b) \\
y(c) &= y_i(iL(b)) + p(c) \star uy(b)
\end{align*}
\]

To access all of the cars on a block in order of decreasing \( p \), we use the following linked-list data structure

\[
\begin{align*}
\text{firstcan}(b) &= \text{index of first car on block } b \\
\text{nextcan}(c) &= \text{index of car immediately behind car } c \text{ on the same block} \\
\text{lastcan}(b) &= \text{index of last car on block } b
\end{align*}
\]

In all cases, an entry of 0 means that there is no such car. Thus \( \text{nextcan} (\text{lastcan}(b)) = 0 \), and if block \( b \) is empty then \( \text{firstcan}(b) = \text{lastcan}(b) = 0 \).
Entry of cars and choice of their destinations

Cars enter the roadway (from parking garages or parking spaces) at random times and locations. Let $R$ be the rate at which this occurs. Then $R$ has units of $1/(\text{time} \cdot \text{length})$. Choose the time step $dt$ small enough that $R \cdot L_{\text{max}} \cdot dt << 1$, where $L_{\text{max}}$ is the largest length of any block. Then we can make the approximation that at most one car enters the roadway per block per time step. To decide whether this happens and to choose the location $p$ on the block if it does, we can do the following for each block $b$:

```plaintext
if (rand < dt \cdot R \cdot L(b))
  nc = nc + 1
  p(nc) = rand \cdot L(b)
end
```

When a car enters the roadway, it is assigned a destination. This can also be done randomly. Let $bd(c)$ be the block on which the destination lies and let $pd(c)$ be the position on that block, expressed as distance from the
start of the block. A simple way to make
this choice is

\[
bd(c) = 1 + \text{floor}(\text{rand} \times nb)
\]
\[
pd(c) = \text{rand} \times L(bd(c))
\]

but note that this choice gives equal weight
to any block regardless of its length. To
make the probability of choosing a block be
proportional to its length, we can use
the method of rejection:

\[
bd(c) = 1 + \text{floor}(\text{rand} \times nb)
\]
\[
pd(c) = \text{rand} \times L_{\text{max}}
\]
while \((pd(c) >= L(bd(c)))\)
\[
bd(c) = 1 + \text{floor}(\text{rand} \times nb)
\]
\[
pd(c) = \text{rand} \times L_{\text{max}}
\]
end

In this version we keep trying until we find
a position that fits on the block, and this makes
the block that is ultimately chosen be more
likely to be a longer one. In fact, the probability
of choosing a block is exactly proportional to its
length, and pd is uniformly distributed over
that length.
Steering a car to its destination (despite one-way streets!)

For this we need the Cartesian coordinates of the destination, which are given by

\[ x_d(c) = x_i(\text{let}(bd(c))) + pd(c) \cdot ux(bd(c)) \]
\[ y_d(c) = y_i(\text{let}(bd(c))) + pd(c) \cdot uy(bd(c)) \]

When a car comes to an intersection, it can choose to enter any of the blocks leaving that intersection. The natural choice is the one that most nearly points towards the destination. To determine this, evaluate the vector from the intersection to the destination, and then the dot product of that vector with all of the unit vectors of the blocks leaving the intersection. The block that should be chosen is the one that maximizes this dot product (in the algebraic sense, i.e., choose the most positive or least negative result, not the one largest in magnitude).
According to the above prescription, if car c is at intersection \( i \), it should choose the next block \( b \) to enter in the following way:

\[
\begin{align*}
xdvec &= xd(c) - xi(i) \\
ydvec &= yd(c) - yi(i)
\end{align*}
\]

\[
dp = u_x(bout(i,1:nbout(i)))*xdvec \\
+ u_y(bout(i,1:nbout(i)))*ydvec
\]

\[
[dpmx, jb] = \max(dp)
\]

\( b = bout(i,jb) \)

In the above use of \( \max \), there are two outputs. The second one, \( jb \), is the index of the element of \( dp \) that has the maximum value.

The above steering algorithm works well for reasonable road networks, including cases in which it is necessary to go around the block to reach the destination because of one-way streets, but it is not guaranteed to work. For some roadway layouts and some destinations, a car can get trapped and go through a cycle of
blocks repeatedly by following the above algorithm without ever reaching its destination. One way to avoid this is to have the car to remember the intersections it has been to and the choices it has made there, and never make the same choice twice at any given intersection. Another way that is easier to program is for the car to decide randomly at each intersection whether to follow the above algorithm or to choose a random block. This can be programmed as follows:

```plaintext
if (rand < prechoice)
    jb = 1 + floor (rand * n bout (i))
    b = bout (i, jb)
else
    choose b by the method of maximizing the dot product as described above
end
```

Here, prechoice is the probability that a random choice will be made.
% main program: traffic.m

initialize
for clock = 1:clockmax
    t = clock * dt
    setlights
createcars
movecars
plotcars
end

% setlights.m
if t > tlc
    for i = 1:ni
        jgreen(i) = jgreen(i) + 1
        if jgreen(i) > nbins(i)
            jgreen(i) = 1
        end
    end
end
tlc = tlc + tlesstep
end
s = zeros(1,nb)
for i = 1:ni
    b = bin(i, jgreen(i))
    s(b) = 1
end
% initialization for setlights

green = ones(1, ni)
tlastep = % time interval between light changes
tlc = tlastep

% createcars.m
for b = 1: nb
    if (rand < dt*R*L(b))
        nc = nc + 1
        p(nc) = rand*L(b)
        x(nc) = xi(i1(b)) + p(nc)*ux(b)
        y(nc) = yi(i1(b)) + p(nc)*uy(b)
        onroad(nc) = 1
        insertnewcar
        choose destination
        nextb(nc) = b
        tenter(nc) = t
        benter(nc) = b
        penter(nc) = p(nc)
    end
end
insertnewcan.m

C = firstcan(b)
if (c == 0 || p(nc) > p(c))
  nextcan(nc) = c
  firstcan(b) = nc
  if (c == 0)
    lastcan(b) = nc
  end
else if p(nc) <= p(lastcan(b))
  nextcan(lastcan(b)) = nc
  lastcan(b) = nc
else
  ca = c
  c = nextcan(c)
  while (p(nc) <= p(c))
    ca = c
    c = nextcan(c)
  end
nextcan(ca) = nc
nextcan(nc) = c
end
% choose destination.m
% use method of rejection to choose a
% block with probability proportional to
% its length, and with p uniformly
% distributed in that block.

\[ bd(nc) = 1 + \text{floor} \ (\text{rand} \ * \ nb) \]
\[ pd(nc) = \text{rand} \ * \ L_{\text{max}} \]
while ( pd(nc) >= L ( bd(nc) ) )
  \[ bd(nc) = 1 + \text{floor} \ (\text{rand} \ * \ nb) \]
  \[ pd(nc) = \text{rand} \ * \ L_{\text{max}} \]
end
\[ xd(nc) = x_i ( i \ L(bd(nc))) + pd(nc) \ * \ ux(bd(nc)) \]
\[ yd(nc) = y_i ( i \ L(bd(nc))) + pd(nc) \ * \ uy(bd(nc)) \]

% \( L_{\text{max}} = \max (L) \)
\% movecars.m

for b = 1 : nb
    c = firstcan (b)
    while (c > 0)
        if (c == firstcan (b))
            if (bd(c) == b) & (pd(c) > p(c))
                d = dmax
            elseif (S(b) == 0)
                d = L(b) - p(c)
            else
                decide next block
                if (lastcan (nextb(c)) > 0)
                    d = (L(b) - p(c)) + p(lastcan (nextb(c)))
                else
                    d = dmax
                end
            end
        end
    end
else
    d = p(ca) - p(c)
end

p2 = p(c); nextc = nextcan (c)
p(c) = p(c) + dt * v(d)
if (b == bd(c)) \&\& (pz < pd(c)) \&\& (pd(c) <= p(c))
    remove car
else if (L(b) <= p(c))
    p(c) = p(c) - L(b)
    if (next b(c) == bd(c)) \&\& (pd(c) <= p(c))
        remove car
    else
        carto next block
    end
else
    x(c) = x i(i1(b)) + p(c) * ux(b)
    y(c) = y i(i1(b)) + p(c) * uy(b)
    ca = c
end

C = next c \% saved value of nextcar(c)
end \% while loop over cars on block
end \% for loop over blocks
\% decide next block
\% only do this if decision is not already made
if nextb(c) == b
    i = i2(b)
    if rand < prchoice
        jnext = floor(rand * nbout(i))
        nextb(c) = bout(i, jnext)
    else
        xvec = xd(c) - xi(i)
        yvec = yd(c) - yi(i)
        dp = ux(bout(i, 1:nbout(i))) * xvec
            + uy(bout(i, 1:nbout(i))) * yvec

        [dpmax, jnext] = max(dp)
        nextb(c) = bout(i, jnext)
    end
end
% remove car m
onroad(c) = 0; texit(c) = t
if (c == firstcan(b))
    firstcan(b) = nextcan(c)
    if (c == lastcan(b))
        lastcan(b) = 0
end
else
    nextcan(ca) = nextcan(c)
    if (c == lastcan(b))
        lastcan(b) = ca
end
end

% not really needed, but...
x(c) = xd(c)
y(c) = yd(c)
nextcan(c) = 0
% recall that we previously set nextc = nextcan(c)
% carto next block
firstcan(b) = nextcan(c)
if (c == lastcan(b))
    lastcan(b) = 0
end
if (lastcan(nextb(c)) == 0)
    firstcan(nextb(c)) = c
else
    nextcan(lastcan(nextb(c))) = c
end
lastcan(nextb(c)) = c
nextcan(c) = 0
% this is why we previously set next(c) = nextcan(c)

\[ p(c) = p(c - \hat{F}(b)) \]

\[ x(c) = x_i (i \in (nextb(c))) + p(c) \times u_X(nextb(c)) \]
\[ y(c) = x_i (i \in (nextb(c))) + p(c) \times u_Y(nextb(c)) \]
% plotcars.m

if (nc > 0 && sum(onroad) > 0)
    set(hcars, 'xdata', x(find(onroad)), 'ydata', y(find(onroad)))
end