

# Agent Based Modeling in Perl

Guinevere Nell

Agent-based programming, or multi-agent programming, can be used to model anything from Conway's Game of Life to molecular or ecological representations or even social interactions such as marketplaces, rumor mills and politics. But its great versatility does not mean that it's difficult or complicated to learn.

Agent based modeling probably originated with the Von Neumann machine. The theoretical machine would follow detailed instructions to create a copy of itself; after the addition of Stanislaw Ulam's idea that it be represented as collection of cells on a grid, this machine became the first cellular automata. John Conway created the well-known Game of Life, which existed in a virtual world in the form of a 2-dimensional checkerboard.

The birth of agent based model as a model for social systems began with computer scientist Craig Reynold. He attempted to model the reality of biological agents which became known as "artificial life", a term coined by Christopher Langton.

Perl is an excellent language for agent based programming because it is so easy to churn out code – it has the right tools and the fluid syntax and the get-on-with-it attitude. So, if

you are new to agent based programming, there is no better way to get started than to use Perl.

At it's most basic, an agent based program includes separate units – objects, forked processes, threads – that each act as an agent with distinct features or actions over time. These agents can then interact with each other or update a common set of variables, or simply each run in unison or in sequence, doing their own thing. What really distinguishes agent based programming from other kinds, is that it always produces distinct agents for some purpose. But the most exciting applications are those that ultimately allow the agents to interact in some form and even learn and grow and change.

You could fork a process for each agent, but that would be CPU intensive. So, how can one create such a program without forking or threading? Simple. Loop through and create objects. Can they interact if they are run sequentially? Sure. They can use a common pool of goods for exchange; global variables (or a database) for environmental factors; delayed messaging, etc. So why do you want to use agents?

In economics, a recurring problem is that models can over-simplify and it is sometimes hard to know whether a model is too simple. Assumptions of a static state in economics can lead to models which miss major consequences of policy changes. Recently, a new kind of modeling is emerging for economics – agent based modeling. Industrial organization, labor markets, cooperation and game theory and many other concentrations have all been areas of focus for agent based modeling. But economics is only one field

where agent based models are used – biology (molecular, evolutionary and ecological), information theory, political science and many other fields are all being investigated with agent models. In each of these fields the agent model allows for emergence of phenomena from the lower (micro) level of the social system where individuals act to the higher level (macro), which is society.

So, how does agent based modeling help to solve this problem of a dynamic economy where secondary consequences can unexpectedly occur? Well, the agents can make choices based on initial (micro) assumptions, such a response to higher prices; then when policy changes are enacted (such as a price control) we can see what the model predicts not only immediately, but also further down the road. If lower prices create more demand, will they also reduce supply or will some firms find it profitable to expand, cut costs and monopolize the market? If prices are low for this good, how will this change consumption and employment? Will this be an overall boon to the economy or will it actually hurt the consumer?

One way that agent based models are superior to other dynamic models based on systems of equations (other than with agent based modeling you can avoid systems of equations) is that the agents can learn and evolve. Some learning is theoretically possible when using other kinds of modeling, but with agents that individually learn and then interact, the society is truly an emergent property of micro level behavioral assumptions.

Now lets look at an example.

## Listing 1: Procedural shell

---

```
#!/usr/bin/perl
use strict;
use Firm;
use Customer;
use Shop;

## have 25 customers and 5 firms, 1 product

## product costs $1 to make, there is also $5 fixed costs

my(@customers);
my(@firms);

## create agents once
for(1 .. 25){
    my $agent = Customer->new;
    push @customers, $agent;
}

my $Count = 1;
my $shop = Shop->new;
for(1 .. 5){
    my $agent = Firm->new;
    my $name = "Firm" . $Count;
    $shop->addFirm($name);
    $agent->{Name} = $name;
    $Count++;
    push @firms, $agent;
}

my $tick = 0;

### loop
while(1){

# randomize order
    (@customers) = sort { rand(100) <=> rand(100) } @customers;
    (@firms) = sort { rand(100) <=> rand(100) } @firms

    $tick++;

# customers shop, (first round all cost $5)
    for my $cust(@customers){
        $cust->tick;
    }

# firms check sales, get profit, set price
    for my $firm(@firms){
        $firm->tick;
    }

# lets find out how price has changed
```

```

my($prices) = $shop->getFirms;
my $average_price;
for my $firm(keys %$prices){
    $average_price += $prices->{$firm};
}
$average_price /= scalar(keys %$prices);

print "\n\nTick: $tick Ave Price: $average_price\n\n\n";

# last rounds shopping is wiped clean
$shop->clearTick;
}

# thats it

```

---

We create agents and in a loop we randomize the order (so that no firm or customer gets any preferential treatment by accident), and then call tick() which acts as the wrapper method for all agent functions; then we do clean up, calculate the average price in the market and print it out. We are aware of the order in which the agents are run. First the customers go shopping, then the firms modify their prices, then we do clean up from last round's shopping.

Now, all the action happens within those tick methods:

---

**Listing 2: Customer class**

---

```

package Customer;
use strict;

sub new{
    my $self = {};
    bless $self, "Customer";
    return $self;
}

sub tick{
    my($self) = @_;
    # all we really do is go shopping
    $self->go_shopping;
    return 1;
}

sub go_shopping{
    my($self) = @_;
    my $shop = Shop->new;
    my($prices) = $shop->getPrices;
    my(@cheapest) = sort { $prices->{$a} <=> $prices->{$b} } keys
%$prices;
    $shop->shop($cheapest[0]);
    return 1;
}

```

```
}
```

```
1;
```

---

The customer class is very simple. The customer gets the prices available to him – in this model information is limited, so only two of the five firm’s prices are returned by the call to `getPrices()` – and he picks the cheapest firm and buys from it.

In a more expanded model, you might also use stochastic preferences and variations (yes, even using Perl’s `rand()` function, but you can also use other kinds of distributions if they are more realistic such as the normal distribution in `Math::Random::OO::Normal`); for example preferences for what products to consume and attributes such as education, age and productivity which differentiate the agents, who could then both work and consume. The agents, as the model progresses, can then build upon these attributes. This is something I have done in my models of the marketplace. Now lets look at the Firm.

### **Listing 3: Firm class**

---

```
package Firm;
use strict;

sub new{
    my $self = {};
    bless $self, "Firm";
    return $self;
}

sub tick{
    my($self) = @_;

    # we saved our price from last time, first ticks its 5
    my($price) = $self->{last_price} || 5;
```

```

### first, firms must check how much profit they made
my($profit) = $self->get_profit($price);
my($last_profit) = $self->{last_profit} || 0; # from last tick

### next, firms set their price

my($price) = $self->set_price($profit,$last_profit,$price);

# save for next tick
$self->{last_profit} = $profit;
$self->{last_price} = $price;
return 1;
}

sub get_profit{
my($self,$price) = @_;

#lets get our sales
my $name = $self->{Name};
my $shop = Shop->new;
my($sales) = $shop->getShopping($name);

print "Sales: $sales\n";

# calculate profit
my $cost = 5; # fixed costs
my($profit) = ($sales * $price) - $cost;

#return it
return $profit;
}

sub set_price{
my($self,$profit,$last_profit,$price) = @_;

my $last_strategy = $self->{last_strategy} || 0;
my $profit_dir;
print "$self->{Name} Profit: $profit ; last profit: $last_profit\n"
and sleep 1;

if($profit > $last_profit){
    $profit_dir = "up";
}else{
    $profit_dir = "down";
}

# get our probabilities
my($prob_matrix) = $self->{prob_matrix};
unless($prob_matrix){
    $prob_matrix = [0.4,0.3,0.3]; # defaults
}
print "My probabilities: ", @$prob_matrix, "\n";

```



```

        }
    }else{
        unless($prob_matrix->[$c] > 0.950001){
            $prob_matrix->[$c] += 0.05;
        }
    }
}

} # end if strategy

$self->{prob_matrix} = $prob_matrix; # save for next time

my(@strategies) = $self->GetStrat(@{$prob_matrix});

my $c=0;
for my $strat(@strategies){
    if ($strat){
        $last_strategy = $c; # change
        if($c==0){
            # lower price
            $price -= ($change_price * $price);
            print "$self->{Name} Lowering price, NEW price =
$price\n\n";
        }elseif($c==1){
            $price += ($change_price * $price);
            print "$self->{Name} Raising price, NEW price=
$price\n\n";
        }else{ # else stay same
            print "$self->{Name} keeping price the same - $price\n\n";
        }
    }
    $c++;
}

$self->{last_strategy} = $last_strategy; # save for next time

# set it with the Shop
my $shop = Shop->new;
my $name = $self->{Name};
$shop->setPrice($name, $price);

#and return it
return $price;
}

sub GetStrat{
    my($self,@prob) = @_ ;

# probs will be eg 0.9,0.05,0.05
# 90% give 1,0,0 5% each give 0,1,0 or 0,0,1

    my $index;
    my $rand = rand(1);
    if($rand <= $prob[0]){ # if less than 0.9 or < 0.05

```

```

        #print "Rand < $prob[0] = $rand\n";
        $index = 0;
    }elseif($rand > (1 - $prob[1])){ # else great than .95 (eg between
0.95-1) or > 0.05
        #print "Rand > (1 - $prob[1]) = $rand\n";
        $index = 1;
    }else{ # else must be between 0.95 - 1
        #print "ELSE $rand\n";
        $index = 2;
    }
    my(@ret) = (0,0,0);
    $ret[$index] = 1;
    #print "Set ret[$index] = 1 so @ret\n\n";
    return(@ret);
}

```

1;

---

So, here is where all the major action is. Firms calculate their profit and then potentially change their pricing strategy in the method `set_price()`. The learning algorithm is simple and commonly used. It is based on a genetic algorithm learning classifier system (GALCS), which assigns probability vectors to various states; but this simplification only uses one probability vector. The probability vector contains three choices: lower the price (L), raise the price (R), and keep the price the same (C). The probabilities default to 0.4, 0.3, 0.3, which means that there is a 40% that firms will try lowering their price first, 30% that they will raise it and 30% that they will leave it alone.

After they make this price adjustment, they track the change in profits. Because of the limited information in the market (customers seeing only two prices and choosing from among the two), it may be a good strategy to raise your price. Profits are calculated in the next cycle and if profits went up, the probability of choosing that strategy is incremented; if profits decreased, the probability for choosing that strategy is decremented.

We increment or decrement the probability of using that strategy first, then adjust the other probabilities accordingly. This is important because if we cannot increment or decrement the strategy anymore because it has reached its limit, then we must be able to exit the loop and leave everything exactly as it is.

Finally, we send our probabilities in to `GetStrat()` where a strategy is chosen based on the probability distribution, and return to evoke the strategy – raising, lowering or leaving our price alone.

This simple algorithm actually provides a realistic model of learning. Agents do not immediately raise their price simply because it worked once; but they have a higher probability of trying that strategy again. The simple algorithm is fast, clean and realistic enough for many purposes. If your agent simulation requires intensive learning trees, neural nets, genetic algorithms and so forth, you can always write the most intensive code in C and Inline it, using Inline C to prevent bottleneck. But this small agent program is incredibly fast – and would be even with many more agents – I include calls to sleep() so that the output is readable.

Finally, lets look at the Shop class.

**Listing 4: Shop class**

---

```
package Shop;
use strict;

my(%Firms);
my(%Shopping);

sub new{
    my $self = {};
    bless $self, "Shop";
    return $self;
}

sub getPrices{
    my($self) = @_ ;
    # get all firms and then send back just three random
    my($firms) = $self->getFirms;

    # sort randomly
    my(@two_firms) = sort { rand(100) <=> rand(100) } keys %$firms;

    # now pur three in a hash with their price and return it
    my(%return_two);
    for(1..2){
        my $firm = shift @two_firms;
        $return_two{$firm} = $firms->{$firm};
    }
    return(\%return_two);
}

sub getShopping{
    my($self,$firm) = @_ ;

    return $Shopping{$firm};
}

sub shop{
    my($self,$firm) = @_ ;
    $Shopping{$firm}++;
}
```

```

        return 1;
    }

    sub clearTick{
        my($self) = @_;
        %Shopping = ();
        return 1;
    }

    sub getFirms{
        my($self) = @_;
        return \%Firms;
    }

    sub setPrice{
        my($self,$firm,$price) = @_;
        $Firms{$firm} = $price;
        return 1;
    }

    sub addFirm{
        my($self,$firm) = @_;
        $Firms{$firm} = 5; # added, with price of 5
        return 1;
    }

1;

```

---

The Shop class is simply a class to store the global variables about the marketplace, such as the names and prices of the firms. As mentioned earlier, the getPrices method only returns two firms chosen at random, so that information is limited in the market for this model. And that's it. 360 lines of code, one learning algorithm and an interesting model of the market is created.

The results look like this:

### Listing 3: Results

---

Sales: 3

Firm1 Profit: 2.60232448577881 ; last profit: 3.4470272064209

My probabilities: 0.0500000000000010.850.0999999999999999

Firm1 keeping price the same - 2.53410816192627

Sales: 4

Firm2 Profit: -1.1988377571106 ; last profit: 0.701743364334106

My probabilities: 0.20.550.25

Firm2 Raising price, NEW price= 1.42543584108353

Sales: 15

Firm3 Profit: 20.3410816192627 ; last profit: 14.1465950012207

My probabilities: 0.1500000000000010.80.0499999999999999

Firm3 Raising price, NEW price= 2.53410816192627

---

User input can determine how limited information is, and the model can be run to compare prices over time in the different situations. For example, are prices lower if 80% of prices in the market are known versus only 20% (as in this example)? This model could then be expanded to include hiring and firing at the firms based on profit; multiple products and many other interesting market phenomena. Then policies could be added such as price caps or limitations on mergers. How does limited information affect competition among firms and does anti-trust regulation reverse the phenomenon of conglomeration? Does it have unexpected consequences in terms of employment, prices, wages or standard of living?

Once the model has reached some early equilibrium, instead of printing the results, you'll likely want to write them to a database or take an average for the cycle and save the averages and write them to an excel file at the end. In my model of the market economy, I wait 25 cycles for firms to do their initial hiring, workers to find the right job, firms to try different prices for their products and consumers and firms to settle on an early equilibrium price; then I save the wages, prices and employment levels for each firm and individual until the end of the loop and save the data until the cycles are complete. Before exiting the program, I use Spreadsheet::SimpleExcel to write the data to an excel file.

For more on agent based modeling, the inquiring mind should visit this website:

<http://www.econ.iastate.edu/tesfatsi/ace.htm>