# How to Program Your Way Out of a Paper Bag Using Particle Swarm Optimisation

Previously [Overload118], I demonstrated how to use a genetic algorithm to program your way out of a paper bag. This had the main drawback that an initial model was required. In this specific case, we used the well-known equations for the ballistic trajectory of a projectile

In this article, I will introduce a method for solving problems, given the right circumstances called optimisation, which does not require a model up front. This is one of a vast class of optimisation techniques which seems to be wells suited to paper bag escapology, though does have other somewhat more practical uses, for example finding training weights for neural networks*.* Though the genetic algorithm approach worked well, there are often circumstances in which one does not have a model, and even if armed with an upfront model it often requires calibration, to find suitable constants. In the projectile equations, we just require a numerical value for the single constant *g*. In the general case more “constants” may need finding, and frequently models need to be recalibrated since the constants they use are in fact not constant [AccuLondon].Though, for example, measuring instruments will tend to just need calibrating once, in other areas, such as finance, models are recalibrated weekly or even more frequently, perhaps suggesting that the models are not that close to reality. Specifically, various sources ascribe the quote "When you have to keep recalibrating a model, something is wrong with it," to Paul Wilmott [e.g. ScientificAmerican]. In contrast to approaches requiring a model, particle swarm optimisations still explore the solution space, partially randomly as with genetic algorithms, but exploit the idea of a social sharing of information, via the fitness function, thereby supposedly mimicking swarm or foraging behaviour.

## Initial attempt

Starting from first principles, it is not difficult to write code to make one particle move around in space and stop when (and if) it finally escapes from a paper bag. If you will indulge me, I will use JavaScript this time, and draw on the HTML canvas. If you are unfamiliar with this technology there are many online tutorials to get you up to speed [e.g. <https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Canvas_tutorial>]. Hopefully the code is intuitive enough that it needs no explanation.

Given a canvas declared in html, with a message for browsers that do not support this

<canvas id="myCanvas" width="800" height="400">

Your browser does not support the canvas element.

</canvas>

First, we draw a 2 dimensional bag and place a small rectangle, representing the particle somewhere in it.

//Draw the bag

var c=document.getElementById("myCanvas");

var ctx=c.getContext("2d");

ctx.clearRect(0,0,c.width,c.height);

ctx.fillStyle="#E0B044";

var bag\_width = 300;

var left = 75;

var right = left + bag\_width;

var up = 25;

var down = up + bag\_width;

ctx.fillRect(left,up,bag\_width,bag\_width);

//Draw the particle

ctx.beginPath();

ctx.rect(150, 150, width,width);

ctx.strokeStyle="black";

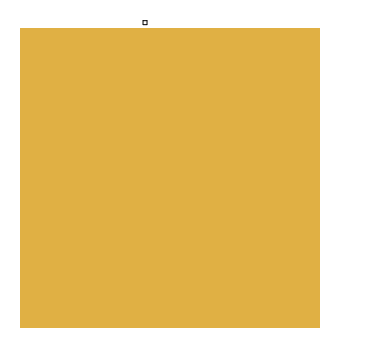
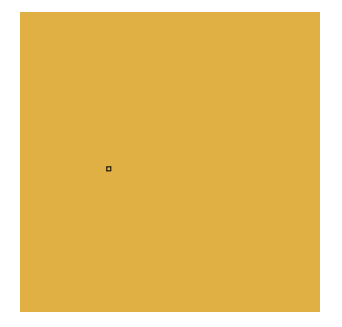
ctx.stroke();

Then, we set up a callback function, allowing the gap between invocations to redraw the canvas, which draws the present position and then moves the particle. If it escapes the bag we stop, otherwise we re-call the callback. Without loss of generality, assuming a square paper bag, we can move as follows. Starting with the particle at position (x, y), indicated by the small black rectangle at some point, say in the middle of the bag, it is allowed to move a little in either direction – up or down and left or right. The random movements are allowed to continue until the particle finally escapes the paper bag.

x += bag\_width \* 0.2 \* (-0.5 + Math.random());

y += bag\_width \* 0.2 \* (-0.5 + Math.random());

The particle starts as requested and wends its way round the canvas like an angry ant until it finds its way out of the bag.



When it escapes we cancel the callback using the id we remembered:

clearInterval(id);

In theory it may never escape, but always has done so far. It takes about 3 seconds, though varies greatly, sometimes taking just one second and other times more than 10.

What have we learnt from this? Aside from how to use the canvas, very little apart from simply trying some random moves can work. It could possibly be sped up, or optimised in a sense, by allowing several particles to set off on their journey simultaneously and seeing who wins.

## Attempt 1 – Every man for himself

Changing the original code to have an array of “Beasties” or particles, rather than just tracking the x and y coordinates of a single item is relatively straightforward. First we need a Beastie, perhaps given a starting x and y position:

function Beasty(x, y, id, index)

{

this.x = x;

this.y = y;

this.id = id;

this.index = index;

}

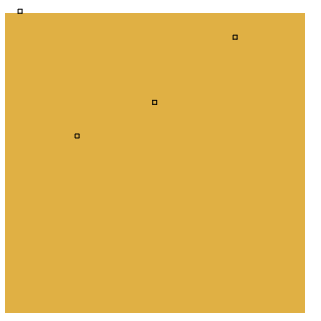
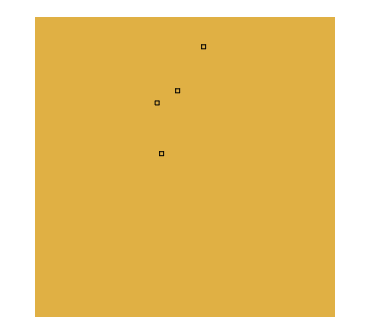
Then we need to track these, having decided how to start them off. I took the approach of clicking a button to form a new particle, though there are other options.

id = setInterval(function() { move(index); }, 100);

var beast = new Beasty(x, y, id, index);

ids.push(beast);

We store the id of the interval in order to cancel all the particles when we’re done. As previously, the move function moves the given particle by a small random amount. If the particle ends up outside the bag it then stops and freezes the others in their tracks, using their ids. The algorithm could be altered to wait for all of them to escape. This is left as an exercise for the reader.



Previously we used a genetic algorithm to allow several attempts at problem solving to “share knowledge” by combining the angle and velocity from randomly selected better particles of the previous generation to form a new better younger generation. In this approach, the particles each follow their own random walk and do not communicate with each other. If they influence one another we could end with all the particles escaping the paper bag.

## Attempt 2 – the blind following the blind

Making the particles follow each other is relatively easy though will prove to be a foolish thing to do. There are several options, but obviously we can’t have every particle following every other particle otherwise they are likely to freeze. If they all move towards each other they are likely to swarm, or indeed clump, together and move no further. A more fruitful approach might be a variant of the k nearest neighbours [KNN] algorithm. Allowing each particle to take a step independently, but also pulling it towards some of its nearest neighbours will allow the particles to actually move but still tend to swarm together.

To find the nearest neighbours of a given particle in our array of particles with index *index* we find its distance from the particle under consideration, order them by distance and just return the top n as follows:

function knn(items, index, n) {

var results =[];

var item = items[index];

for (var i=0; i<items.length; i++) {

if (i !==index) {

var neighbour = items[i];

var distance = Math.sqrt(item.x\*neighbour.x +

item.y\*neighbour.y);

results.push( new distance\_index(distance, i) );

}

}

results.sort( function(a,b) {

return a.distance - b.distance;

} );

var top\_n = Math.min(n, results.length);

return results.slice(0,top\_n);

}

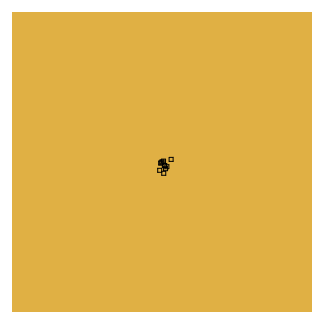
The number of neighbours can either be specified in advance or changed as the simulation runs. I settled for the minimum of 5 and the number of particles, though as you can see the function is flexible. In general, the distance function must be chosen carefully so it is suitable for the domain. In our case, the straightforward Euclidean distance should be suitable since this is inherently a spatial problem. For the mathematically challenged, think Pythagoras. Finding the average x and y displacement or nudge of these nearest neighbours from each particle can be used in conjunction with a small stochastic (or random) step:

x\_move += (x\_nudge - beast.x) \* neighbour\_weight \* (-0.5 + Math.random());

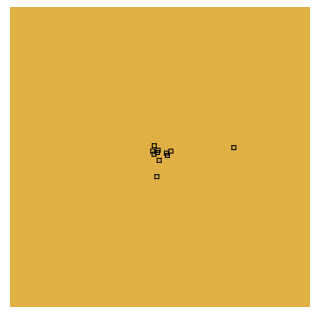
y\_move += (y\_nudge - beast.y) \* neighbour\_weight \* (-0.5 + Math.random());

Unfortunately, this means the particles do tend to swarm together but if some of them are not doing very well, they can greatly increase the time taken for all of the particles to escape. The particles do all tend to escape eventually but can take an hour or so to finish.

We tend to find the particles clump together initially



Sometimes one starts to escape



However, it can tend to be pulled back with the others. Clearly, its nearest neighbours are in the main clump or swarm of particles, so this is unsurprising. Usually, they do manage to move away from the main swarm.

## 

The simulation stops once all the particles have escaped the paper bag. Notice that even though we intended to encourage them to swarm together they have tended to each escape from a different spot. They did swarm together but it seems that only the individual randomness allow individuals to escape from the mindless herd and then escape from the paper bag. This was not our intention, though it has paved the way for a more successful approach.

## Attempt 3 – A swarm with memory

If each particle still tends to move randomly, but also moves towards the best of the rest rather than the nearest few of the rest and also is pulled towards its best position so far, it seems likely things may improve. This will allow the swarm to use what it discovers as it moves, both individually and from the swarm memory. In fact, this is the essence of a particle swarm optimisation [PSO]. The pseudo code is as follows:

Choose n

Initialise n particles to a random starting point in the bag

While some particles are still in the bag

Update best global position

Draw particles current positions

Move particles – updating each particle’s current best position

In order to move the particles, each has a position and "velocity". In the move function, each particle’s current velocity is first updated based on its current velocity, the particle’s local information and global swarm information. Then, each particle’s position is updated using the particle’s new velocity. In mathematical terms the two update equations are:

Here w, c1 and c2 are weighting constants though variants of the algorithm allow them to change over time. r1 and r2are random variables. p is personal best, and g is the global best. So, this combines the current velocity, a step in the direction or the personal best for each particle and a step towards the current global or swarm best. Choosing the weightings takes care.

The best positions can either be found synchronously or asynchronously, where best will be defined shortly. This paper presents results for synchronous updates, updating the global best after everything has moved.

"The algorithm above implements synchronous updates of particle positions and best positions, where the best position found is updated only after all particle positions and personal best positions have been updated. In asynchronous update mode, the best position found is updated immediately after each particle's position update. Asynchronous updates lead to a faster propagation of the best solutions through the swarm." [scholarpedia]

Unlike several other optimisation methods, this is “gradientless”. For example, neural networks traditionally find their weights by using the differences between error functions for a change in their value and stepping the weights in the direction of the best value. The steps are always based on the gradient (difference per step size) [Wolfram].In contrast, PSO does not require any calculus to find gradients and use them to infer a step size and direction. In other words, no difficult maths is required to work out optimal ways to find the minimum or maximum of some function or best solution to a problem. We simply try a few things and remember the best so far.

Finally, we need a definition of “best”. In our initial attempts, the particles were allowed to burst through the sides of the bag. In the case of PSO if we simply find the distance to the edge of the bag it is possible to have two equidistant particles with the current particle to be updated exactly in between them. In this case, it will not move towards either of them if both are allowed to exert influence. There are various ways to tackle this problem. I have decided to concentrate on the definition of “best”, providing a fitness function which, along with the approach taken in my genetic algorithm escapology, simply measures the distance to the top of the bag. This way all the particles will be given an imperative to move up.

function best(first, second) {

if (first.y > second.y) {

return first;

}

return second;

}

function updateBest(item, bestGlobal) {

var i;

for (i = 0; i < item.length; ++i) {

bestGlobal = best(item[i], bestGlobal);

item[i].best = best(item[i].best, item[i]);

}

return bestGlobal;

}

Note that the canvas has 0 at the top, but I have flipped things to have 0 at the bottom to git with my mathematical bent. Bigger y coordinates are better, though the drawing code in the case mist remember to flip the u coordinate back again:

var particle = item[i];

ctx.fillRect (particle.x, canvas.height - particle.y - 2, 4, 4);

Care must be taken with the weights, otherwise positions can zoom off to infinity very easily - need "sensible" weights. Specifically, since the velocity will tend to make the particles move up due to the chosen fitness function, we can end up with exponential upwards motion.

*//TODO Do some maths on this? Or explain what happen to me the first time*

*//TODO Do they all move to the far left or far right… do some maths*

function move(item, w, c1, c2, height, width, bestGlobal) {

var i;

for (i = 0; i < item.length; ++i) {

var current = item[i];

var r1 = getRandomInt(0, 5);

var r2 = getRandomInt(0, 5);

var vy = (w \* current.v.y) +

(c1 \* r1 \* (current.best.y - current.y)) +

(c2 \* r2 \* (bestGlobal.y - current.y));

var vx = (w \* current.v.x) +

(c1 \* r1 \* (current.best.x - current.x)) +

(c2 \* r2 \* (bestGlobal.x - current.x));

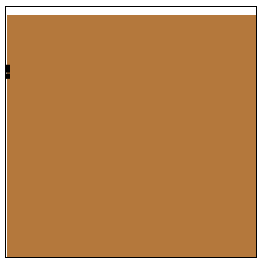
move\_in\_range(vy, height, item[i], "y");

move\_in\_range(vx, width, item[i], "x");

}

The move\_in\_range function simply clamps the particle to the edge of the bag or stops it when it peaks above the top. We could adapt the algorithm and allow it just to consider the best of its nearest neighbours rather than the global best, which could give us more of a flock than a swarm. We could also allow the particles to escape from the sides of the bag. There are several variants, but we shall just report the one approach outlined so far.

Unlike our first attempt, we can see all the particles tend to move together and escape in approximately the same place.



//TODO add some timings

## Conclusion

This article has considered how to program one’s way out of a paper bag without needing an up-front model. This has some obvious advantages over simulations which require a believable model of how a situation might evolve over time. Particle swarm optimisations are part of the more general swarm intelligence algorithms, which allow a collection or swarm of potential solutions to a problem to collaborate, gradually nudging towards a better solution. Other examples include ant colony optimisations [Ants] or bee foraging algorithms [Bees]. In general, the “points” explored will be values to solve another problem rather than spatial points, but hopefully this demonstration has served as a simple introduction for anyone who wishes to take this further. It would be nice to extend this to other swarming and flocking algorithms, perhaps having flights of birds or similar moving out of the bag. I will leave that as an exercise for the reader.

### Notes

The code is on github at <https://github.com/doctorlove/paperbag>

Many thanks to my reviewers.

## References

AccuLondon - <http://accu.org/content/pdf/presentations/accu2011nov/models.pdf>

Ants - <http://accu.org/content/conf2013/Frances_Buontempo_paperbag.pdf>

Bees - Honey bee social foraging algorithms for resource allocation: Theory and application Nicanor Quijanoa, , , Kevin M. Passinob, *Engineering Applications of Artificial Intelligence* Volume 23, Issue 6, September 2010, Pages 845–861 (See <http://www.sciencedirect.com/science/article/pii/S0952197610001090>)

KNN - <http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm>

Overload 118 – “How to Program Your Way Out of a Paper Bag Using Genetic Algorithms” Frances Buontempo, Dec 2013

PSO – J. Kennedy and R. Eberhart. Particle swarm optimization. In Proceedings of IEEE International Conference on Neural Networks, pages 1942-1948, IEEE Press, Piscataway, NJ, 1995. (See http://www.cs.tufts.edu/comp/150GA/homeworks/hw3/\_reading6%201995%20particle%20swarming.pdf)

scholarpedia - <http://www.scholarpedia.org/article/Particle_swarm_optimization>

ScientificAmerican - <http://www.scientificamerican.com/article/finance-why-economic-models-are-always-wrong/>

Wolfram - http://mathworld.wolfram.com/MethodofSteepestDescent.html