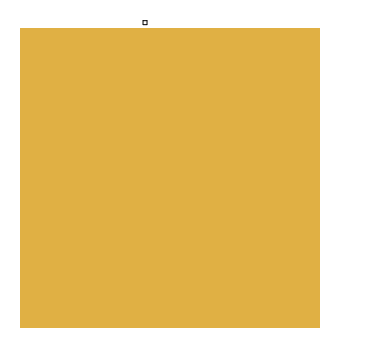
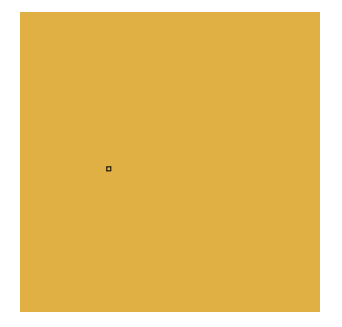
Previously, I demonstrated how to use a genetic algorithm to program your way out of a paper bag. This had the main drawback that an initial model was required. In this specific case, we use the well-known equation for the ballistic trajectory of a projectile

In this article, I will introduce a method for solving a problem, 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. Though the genetic algorithm approach worked well, there are often circumstances in which one does not have a model, and even if armed with a model it often requires calibration, to find suitable constants. In the projectile equations, we just require a numerical value for *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. [??Refs??]

## 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 it 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.



Starting with the particle, 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.

//TODO \_ show drawing code, mention the canvas.

//TODO – actually do this and check it works

function move(canvas, bag\_width, x, y) {

var new\_x\_move = bag\_width \* 0.2 \* (-0.5 + Math.random());

var new\_y\_move = bag\_width \* 0.2 \* (-0.5 + Math.random());

x += new\_x\_move;

x %= canvas.width;

y += new\_y\_move;

y %= canvas.height;

}

This is somewhat boring to be honest. [//TODO : How long on average?] It could possibly be sped up, or optimised in a sense, by allowing several particles to set off on their journey simultaneously.

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:

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 a button click forming a new particle, though there are other options.

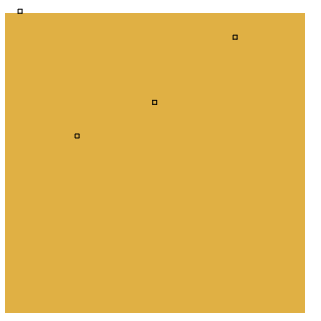
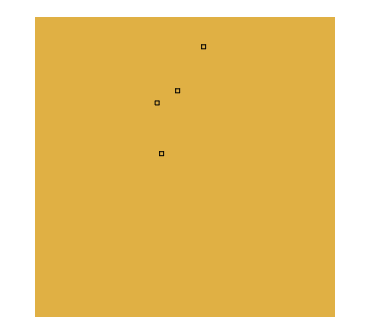
id = setInterval(function() { move(index); }, 100);

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

ids.push(beast);

//TODO \_ tidy this code up, it’s mental… can I avoid the globals ids?

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. It might be ideal to end with all the particles escaping.



Previously we used a genetic algorithm to allow several attempts at problem solving to “share knowledge”. Currently 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 1 – 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. A more fruitful approach might be a variant of k nearest neighbours [KNN]. Allowing each particle to take a step independently, but also pulling it towards its nearest neighbours will allow the particles to actually move but tend to swarm together.

The nearest neighbour code must decide how many neighbours to consider, which may (or may not) depend on the number of particles involved. To find the nearest neighbours of a given particle in our array of particles with index *index* we find the distance from the particle under consideration, order by distance and just use the top n as follows:

function knn(items, index, n) {

var results =[];

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

if (i !==index) {

var neighbour = items[i];

var distance = Math.sqrt(neighbour.x\*neighbour.x + neighbour.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 or simulation runs. I settled for the minimum of 5 and the number of particles. 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. 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:

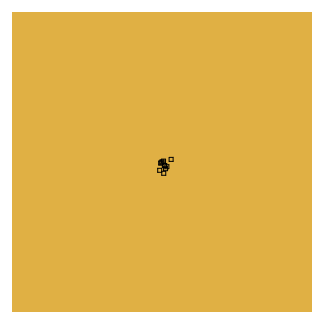
x\_move += (x\_nudge - beast.x) \* neighbour\_weight \* (-0.5 + Math.random());

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

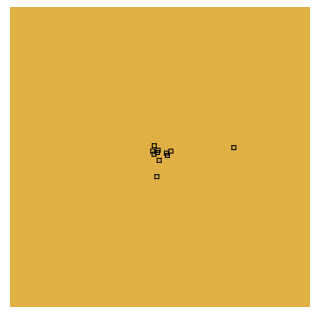
Unfortunately, this means the particles do tend to swarm together but if some of them are not doing very well, they can greatly increase the time taken for all of the particles to escape

//TODO \_ more timings

We tend to find the particles clump together initially



Sometimes one starts to escape



But 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.

## 

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. This was not our intention, though it has paved the way for a more successful approach.

## Attempt 2 – 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. In fact, this is the essence of a particle swarm optimisation [PSO]. Each particle has a position and "velocity". On each iteration in the PSO main processing loop, 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 math terms the two update equations are:

v(t+1) = (w \* v(t)) + (c1 \* r1 \* (p(t) – x(t)) + (c2 \* r2 \* (g(t) – x(t))

x(t+1) = x(t) + v(t+1)

The best positions can either be found synchronously or asynchronously – I have *//TODO \_ wtf have I done?*

"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”. In other words, no difficult maths is required to work out optimal ways to find the minimum or maximum of some function. *//TODO more details.*

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.

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*

var r1 = getRandomInt(0, 5);//need to think about this

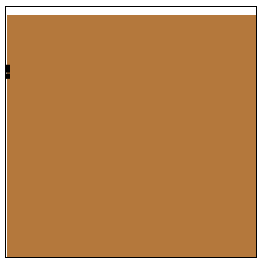
var r2 = getRandomInt(0, 5);//need to think about this

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));

//TODO add the code…

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 “things” *//TODO \_ better word* will 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, among many others, ant colony optimisations [TODO ref my conference talk] or bee foraging algorithms [TODO ref]. 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.

//TODO – consider mentioning other variants given in <http://www.mii.lt/antanas/uploads/Heuristic%20Algorithms/Lectures/Lect4/PSO2.pdf> and name some other types of swarm optimisations

*//TODO mention code on github*

## References

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

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

/////////////notes to self//////

<http://chern.ie.nthu.edu.tw/gen/9.pdf>

<http://www.tech.plym.ac.uk/spmc/links/pso/pso_papers.html>

<http://www.mii.lt/antanas/uploads/Heuristic%20Algorithms/Lectures/Lect4/PSO2.pdf>

chrome console.timestamp

<https://developer.chrome.com/devtools/docs/console-api#consoletimestamplabel>

var milliseconds = new Date().getTime();

//nb milliseconds since epoch