Simulation: Optimal path

labay11

Mètodes Numèrics – UAB
June 17, 2017

1 Introduction

In this simulation we will try to find the best path that joins to points on a bi-dimensional space in the presence of a external potential $U(\boldsymbol{r})$ which only depends on the position. The definition of *optimal path* is ambiguous as we are considering only conservative forces which implies that the work done only depends on the start and final position ($\Delta U = U_f - U_i$) and in consequence all path are in this sense optimal. Taking this into account we will have to find a different criteria for which we can decide if one way is more optimal than another.

For simplicity, we will consider a particle (or walker) that at each step can only move in four directions: front, back, left and right; with fixed length steps h. This way, the general idea of the algorithm is to start walking, in some sense blindly, but trying at each step to get as closer as it can to the final position.

2 Algorithm

Consider that we are at first at position $\boldsymbol{r}_i = (x_i, y_i)$ and we want to get to $\boldsymbol{r}_f = (x_f, y_f)$. Both of this points have a potential associated with its position $U_i = U(\boldsymbol{r}_i)$ and $U_f = U(\boldsymbol{r}_f)$ respectively that we can somehow consider as a virtual height, this way we can extend our 2D space into a 3D space being the third component given by the potential energy, so:

$$\tilde{\boldsymbol{r}} = (x, y, U(x, y)) = (\boldsymbol{r}, U(\boldsymbol{r})) \tag{1}$$

Now our concept of distance will be slightly different but we can take as reference the distance between two points in a 3D euclidean space to get

$$\tilde{d}^2(\boldsymbol{r}_1, \boldsymbol{r}_2) = (x_2 - x_1)^2 + (y_2 - y_1)^2 + [U(\boldsymbol{r}_2) - U(\boldsymbol{r}_1)]^2 = d^2(\boldsymbol{r}_1, \boldsymbol{r}_2) + (U_2 - U_1)^2 \tag{2}$$

being $d(\boldsymbol{r}_1, \boldsymbol{r}_2)$ the distance in our original 2D space.

As it was said, for simplicity, we will only consider four possible movements from a point $\boldsymbol{r}$ that will be

1. Left: $\boldsymbol{r}_1 = \boldsymbol{r} + (-h, 0)$
2. Right: $\boldsymbol{r}_2 = \boldsymbol{r} + (h, 0)$
3. Front: $\boldsymbol{r}_3 = \boldsymbol{r} + (0, h)$
4. Back: $\boldsymbol{r}_4 = \boldsymbol{r} + (0, -h)$

where h is the length of the step, the smaller the step the more continuous the path will be. It is also important to mention that h is also the uncertainty of our walker as we won't be able to get as close as the final point as $\boldsymbol{r}_f \pm h$.

The choose of the points in the path is given by this two conditions:

(a) We can only move in the directions that verify that the distance to the final point is smaller than our current distance, mathematically this condition is given by

$$\tilde{d}(\boldsymbol{r}_i, \boldsymbol{r}_f) \leq \tilde{d}(\boldsymbol{r}, \boldsymbol{r}_f) \tag{3}$$

(b) Between the directions that verify the previous condition, we will move in the direction that makes the weight function $w(\boldsymbol{r}_j)$ minimum where this weight can be defined as

$$w_1(\boldsymbol{r}_j) = |U(\boldsymbol{r}_j)| \qquad \text{or} \tag{4}$$

$$w_2(\boldsymbol{r}_j) = U(\boldsymbol{r}_j) \tag{5}$$

The first condition enables us to verify that the algorithm is correct, this means that if there is any possible path then the algorithm will find it and if it doesn't exist then no path will be found, because we have for sure that somehow we will approach the final point at each step and when we get there the program will stop because there will be no possible directions that verify (3) as in any of them we will move away from $\boldsymbol{r}_f$ (remember that $0 \leq \tilde{d}(\boldsymbol{r}_i, \boldsymbol{r}_f) \leq \tilde{d}(\boldsymbol{r}_f, \boldsymbol{r}_f) \leq 0 \Rightarrow \nexists \boldsymbol{r}_i$). On the other hand, we can have the possibility that the potential energy is discontinuous or has an infinite divergence, in this cases although we approach physically to the point in the xy plane the change in potential energy will be so large that at some point the first condition won't be verified for any neighbour and the program will stop without finding any path because it doesn't exist.

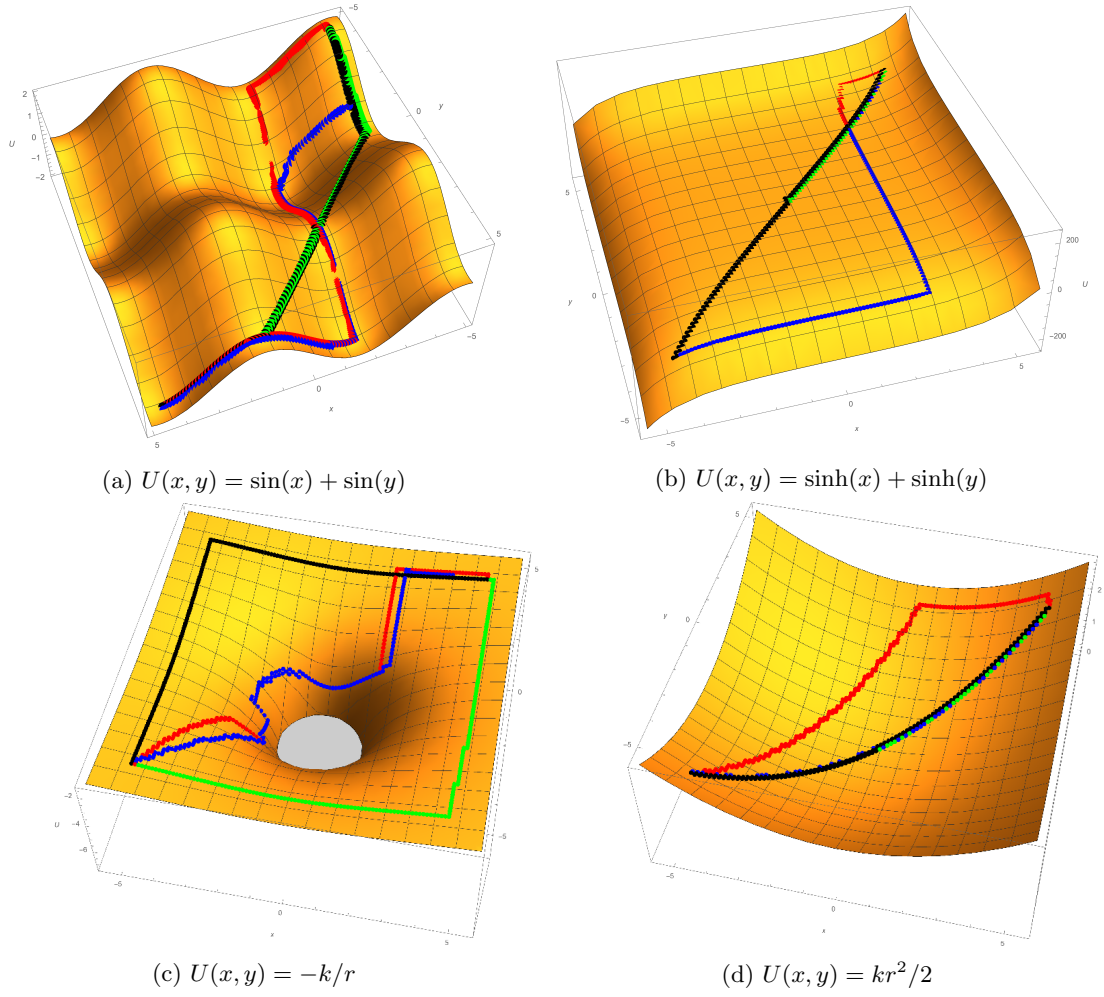
The second conditions give us the ability to choose the following point in our path. The two definitions of this weight function give us two different cases: with the absolute value paths will follow points with potential energy $U \to 0$ and without the absolute value the path will always follow minimums of potential because at each step it will try to minimise its value.

The *pseudo-code* of the implementation of this two methods (one using (4) and the other using (5)) can be found in appendix A with its explanation. For the discussion there will also be considered two of the most well known *shortest path algorithms*: Dijkstra and Bellmand-Ford (BF). To use these algorithms, the space must be divided into vertex of area h^2, then each vertex is connected with an edge to its neighbours that verify condition (3) with a weight equal to the value of the weight function, used in each method, evaluated in that vertex. Necessarily, Dijkstra must use (4) because it only works for positive weights and Bellmand-Ford, as it can handle negative weighted edges, will be tested using condition (5).

3 Discussion

As there is no *optimal path* (or all the path are optimal), this discussion will be focused on how the different algorithms behave under different potentials to really understand how they work, verify that the assumptions made are valid and the physical interpretation of the path found.

In figure 1 there are considered 4 potentials and for each of those 4 path are represented corresponding to the four methods mentioned before with starting and ending points $\boldsymbol{r}_0 = (-4.5, -4.5)$ and $\boldsymbol{r}_f = (4.5, 4.5)$ respectively and a step size of $h = 0.1$.



(a) $U(x, y) = \sin(x) + \sin(y)$

(b) $U(x, y) = \sinh(x) + \sinh(y)$

(c) $U(x, y) = -k/r$

(d) $U(x, y) = kr^2/2$

Figure 1: Paths found using the three methods: in green method 1, in red method 2, in blue BF and in black Dijkstra.

In figures 1a and 1b it can be seen the real difference between using (4) and (5), the first one causes the path to go following a straight line following the equipotential line $U = 0$ while the other methods they first follow the line with minimum energy and then approach to the final point by means of reducing U at each step. Physically this will translate into moving at constant velocity for the first weight because $\boldsymbol{F} = -\boldsymbol{\nabla} U \simeq -(\Delta U/\Delta x, \Delta U/\Delta y) = 0$ when moving through equipotential lines whereas, in the other cases, there is a force on the particle and in consequence an acceleration as $\Delta U \neq 0$.

Focusing on method 2 and Bellman-Ford because, even though both path are similar at the beginning, they differ in the end. This is caused mostly because in method 2 the walker is walking blindly without knowing nothing but the potential of his neighbours, Bellman-Ford's algorithm instead has a complete knowledge of all the graph so it can decide better its movements.

The other two examples of potentials are real physical potentials given by: Hooke's law and the inverse square force (gravity, electromagnetism...). It can be seen in figure 1d that all the

methods output the same path except of method 2 for the same reason explained earlier, the path corresponding to the other three is the geodesic that joins $\boldsymbol{r}_0$ with $\boldsymbol{r}_f$ which is the expected result for this case as this is the shortest path on a (curved) surface.

The third case (fig. 1c) is of most interest due to the large number of physical phenomena that follow this *inverse square law*. In can be seen clearly that there are two types of paths:

1. Weight 1 (eq. (4)): these two paths, got with method 1 and Dijkstra, follow the points with potential energy near 0 as they minimise its absolute value causing the path to go in two straight lines, something not very natural physically. This path could be corrected if condition (3) was replaced by a less restrictive one to allow more movements and in consequence the choice of the next point would be more accurate, consequently the path found would be more similar to a geodesic in this surface.

2. Weight 2 (eq. (5)): these two, in order to go always to the minimum energy, they approach to the singularity where the difference of potential is so huge between neighbours that the path starts to oscillate trying not to fall into the singularity but at the same time finding the minimum energy. These two path could be corrected by considering, as before, a less restrictive condition.

A way of changing condition (3) to a less restrictive one is by changing the definition of distance in equation (2). The simplest way of proceeding is to add a proportional constant $\alpha > 0$ as

$$d^*(\boldsymbol{r}_1, \boldsymbol{r}_2) = d(\boldsymbol{r}_1, \boldsymbol{r}_2) + \alpha(U_2 - U_1)^2 \tag{6}$$

which will cause the distance to be more sensitive to changes of potential energy ($\alpha > 1$) or less sensitive ($0 < \alpha < 1$). Testing BF for different values of α we obtain the paths shown in figure 2.

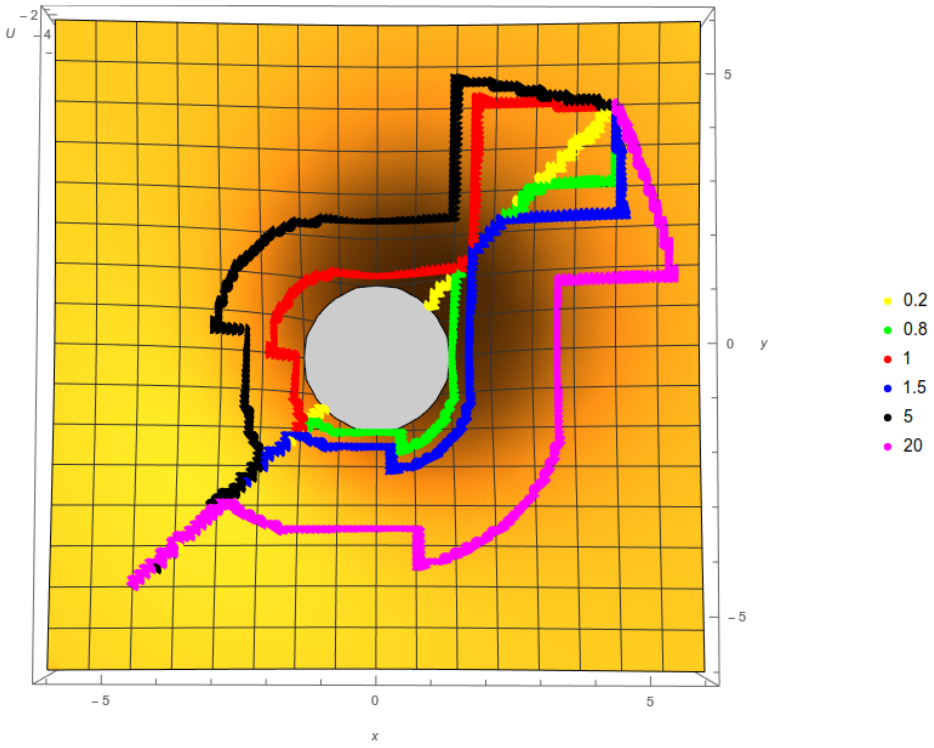


Figure 2: Path found using Bellman-Ford algorithm for different values of α.

We see that as α gets bigger the influence of the potential in the distance increases and as a result the optimal path found becomes more similar to the real one which in this case would be described, for example, by Newton's gravitational force or by an electric force. Of course, the path is not perfect, for better results we could try to increase the value of α or use a smaller value for h

to get a more continuous trajectory but, numerically, this will increase computational time as well as memory allocations which will cause the program to crash.

If we test the other algorithms under the three other potentials shown in figure 1 for all the different values of α the results obtained are the same as for $\alpha = 1$, this is because of the regularity of the function. So for regular functions with no discontinuities ("well-behabed"), there is no special difference in using distance as defined in equation (2) or (6). It is also interesting to mention that none of the path found using condition (4) modifies with α, this fact is a direct consequence of the definition of the weight applied because as in these methods $U \to 0$ then only for very large values of α we would get $d(\boldsymbol{r}_1, \boldsymbol{r}_2) \ll \alpha(U_2 - U_1)^2$ so the difference in potential rules over the difference on physical distance.

To conclude this discussion, we have seen that method 1 and Dijkstra output the same or similar paths and the same happens with method 2 and BF, so only looking at the result the two methods implemented are equivalent to its corresponding *shortest path algorithm*. In contrast, numerically, there is a huge difference between using our methods than the other two algorithms, while method 1 & 2 complete in about $< 0.5\,\mathrm{s}$ for a grid of size $[-5, 5]^2$ and $h = 0.1$, Dijkstra takes $\approx 1\,\mathrm{s}$ and BF $\approx 1\,\mathrm{min}$. This is because the last two algorithms they first need to build the graph, then find the shortest path for all the connected points and finally build the desired trajectory. This causes the algorithms to run very slowly for a large number of vertices (large grids or small h) as the complexity of these algorithms is $O(V \ln(V))$ and $O(V^2)$ for Dijkstra and Bf respectively[1] whereas our implementations, although they are based on these two algorithms, we build the graph and find the path together to reduce the number of iterations and complete the algorithm in $O(V)$ for both methods[2].

[1] Here we've used that $E \sim V$ where V is the number of vertex and E the number of edges (four for each vertex).
[2] A complete explanation of this fact can be found in appendix A

4 Conclusion

- The movements in a system constituted only of conservative forces imply that all path are optimal in terms of energy as the total work done is $W = U(\boldsymbol{r}_f) - U(\boldsymbol{r}_i)$ for any path so conditions (3) to (5) have been imposed to calculate an actual path.
- No condition is better than the other but for cases where is preferable to move at constant velocity (no forces) then condition (4) should be applied and for cases where is preferable to have the particle accelerated then condition (5) should be used even though the distance walked increases.
- Methods using the same condition for the weight output the same or an equivalent path.
- Reducing the step size h produces more continuous results but causes the program to crash or not complete.
- Bellman-Ford path found with a discontinuous potential can be improved by redefining the concept of distance as shown in equation (6) getting better results with $\alpha > 1$.

A Pseudo-code

A.1 Method 1

The first approximation of the algorithm is to look at each of the neighbours given a point $\boldsymbol{r}$, moving to the one that reduces the weight function (4) and continue this way until we reach the end, in *pseudo-code* this algorithm can expressed as:

Input: $\boldsymbol{r}_0$: initial point
Input: $\boldsymbol{r}_f$: final point
Ensure: $\boldsymbol{r}_0 \neq \boldsymbol{r}_f$
  $\boldsymbol{r} \leftarrow \boldsymbol{r}_0$
  while $\boldsymbol{r} \neq \boldsymbol{r}_f$ **do**
    $\boldsymbol{r}_j \leftarrow \min\{|U(\boldsymbol{r}_i)| \mid \tilde{d}(\boldsymbol{r}_i, \boldsymbol{r}_f) \leq \tilde{d}(\boldsymbol{r}, \boldsymbol{r}_f)\ i = 1, \ldots, 4\}$
    $\boldsymbol{r} \leftarrow \boldsymbol{r}_j$
  end while

First, we check that we are not already in the end of out journey (maybe trivial but important) and then we start moving through the space starting at $\boldsymbol{r}$, moving to one of its neighbours, choosing this neighbour as the next $\boldsymbol{r}$ and repeating till $\boldsymbol{r}$ equals $\boldsymbol{r}_f$.

A simple analysis of this algorithm is to consider that in the worst case we will go through all the points in a interval $[a, b] \times [c, d]$ so if we divide this into a grid of square of length h the total number of points (vertices) will be

$$V = \frac{b-a}{h}\frac{d-c}{h} \tag{7}$$

We have guaranteed that we will only pass 1 time through each point because if we go from $\boldsymbol{r}_1$ to $\boldsymbol{r}_2$ then for condition (3) $\tilde{d}(\boldsymbol{r}_2, \boldsymbol{r}_f) \leq \tilde{d}(\boldsymbol{r}_1, \boldsymbol{r}_f)$ so we are not able to back because it will cause the distance to increase in any sense (physical or energetically). So, the order of growth of this algorithms is $O(V) \propto 1/h^2$, we see clearly that the cost of the algorithm increases quadratically when h is reduced.

A.2 Method 2: Shortest path solution

This method can be conceived as an approximation to Bellman-Ford algorithm reducing the number of times the relaxation function is executed for each pair points, here we only relax the vertices that verify condition (3). This relaxation consists on replacing the parent vertex of a point (the previous point) if the total weight is smaller than the current one where the weight of a point $\boldsymbol{b}$ with parent $\boldsymbol{a}$ is given by $W[\boldsymbol{b}] = W[\boldsymbol{a}] + w_2(\boldsymbol{b})$ with w_2 defined by equation (5).

The implementation of this method is somehow more complicated than the previous one but the *pseudo-code* looks like this:

Input: $\boldsymbol{r}_0$: initial point
Input: $\boldsymbol{r}_f$: final point
Ensure: $\boldsymbol{r}_0 \neq \boldsymbol{r}_f$
  $G \leftarrow \{\}$ ▷ Empty dictionary of points
  $W \leftarrow \{\}$ ▷ Empty dictionary of weights
  $W[\boldsymbol{r}_0] \leftarrow 0$
  function RELAX($\boldsymbol{a}, \boldsymbol{b}, G, W$)
    if $W[\boldsymbol{b}] > W[\boldsymbol{a}] + U(\boldsymbol{b})$ **then**
      $W[\boldsymbol{b}] \leftarrow W[\boldsymbol{a}] + U(\boldsymbol{b})$
      $G[\boldsymbol{b}] \leftarrow \boldsymbol{a}$
    end if
  end function

  function LOOP($\boldsymbol{r}$)
    for $i \leftarrow 1, 4$ **do**
      if $\tilde{d}(\boldsymbol{r}_i, \boldsymbol{r}_f) \leq \tilde{d}(\boldsymbol{r}, \boldsymbol{r}_f)$ **then**
        Relax($\boldsymbol{r}, \boldsymbol{r}_i, G, W$)

```
            Loop(r_i)
        end if
    end for
end function
```

The complexity is the same as for method 1 because, in the worst case, the algorithms will only visit a vertex one time so the order of growth is $O(V)$.

One of the problems that this algorithm has is the use of recursion, this causes when then number of vertices is very big ($h \to 0$) a *max-depth recursion limit exceeded* exception. A solution would be, of course, stop using recursion but that would increase the complexity O because we would have to relax all the points with each of its neighbours and do this again for the neighbours which is exactly what BF does turning the complexity of this algorithm into $O(V^2)$.