_classical

Appendix A. Pseudocode for the classical algorithm

Here we present the pseudocode for the classical algorithm for the 2D and 3D periodic Lorentz gas. The 2D version is made of the functions $crossing(\vec{x},\vec{v},n,m)$ and $collisions\_classical(\vec{x},\vec{v},r,t_{max})$.

The function $crossing(\vec{x},\vec{v},n,m)$ outputs the new position in square $[-0.5,0.5)^2$ and updates the position of the new square in case there is no collision in the current cell. First we initialize an empty array of times to each boundary, then take the minimum positive time value. The variable *number* determines the exiting side of the square positioned in the xy-coordinates with the x-axis being horizontal and y-axis vertical (1 - left, 2 - right, 3 - bottom, 4 - top). Then, using this time value, we calculate the new position and the displacement. Then, depending on the exiting side, we increment or decrement the corresponding integer coordinate of the cell and return the new coordinates, displacement, and the integer coordinates (n,m) of the new cell.

The function $collisions\_classical(\vec{x},\vec{v},r,t_{max})$ calculates the trajectory of a particle in a periodic 2D Lorentz gas. We initialize arrays for the coordinates of each collision (*places*), integer coordinates of each circle that experiences a collision (*circles*) and velocities at each step (*velocities*). We push the initial position, initial velocity and initial time (0) into the corresponding arrays. Then we calculate the initial unit square where the initial position is and decrement the initial position by the position of the square so that the position would be counted with respect to the centre of the unit square. We also make sure that the initial position cannot be inside of a circle. While the time is less than the maximum allocated time t_{max} for the calculation of the trajectory, we check the condition (3) for the collision within the unit cell (in the two-dimensional case, the vectors in the cross product are extended with a zero third coordinate).

If the collision takes place, using the formulas from Section 2, we determine the coordinates of the collision, time to the collision and the new velocity after the collision. Then we push those values into the corresponding arrays (notice that we need to push the value of the coordinates with respect to the origin, not to the centre of the circle, therefore, we increment it back). After the collision, we determine the exiting parameters using the previously defined function $crossing(\vec{x}_1,\vec{v}_1,n,m)$. The velocity will take on the new value $\vec{v}_1$ and the time will be incremented by the displacement divided by the speed.

If the collision does not take place (the condition (3) is not satisfied), the particle continues moving with the same velocity and eventually exits the unit square. Using the function $crossing(\vec{x}_1,\vec{v}_1,n,m)$, we determine the exiting parameters and the new unit cell, and increment the time.

Appendix A.1. Modifications for the 3D version of the classical algorithm

In the 3D version of the function *crossing* ($crossing3d(\vec{x},\vec{v},n,m,l)$), there will be 3D vectors for $\vec{x}$ and $\vec{v}$ and also three integer values of the centre of the obstacle (n,m,l). The variable i would run from 1 to 3 as the number of dimensions increases to 3. Also, there would be two more possible outcomes as in 3D the exit may be also made through the two sides of the unit cube: $z = 1/2$ and $z = -1/2$. In the right-hand xyz-coordinates, the variable *number* will have the value "5" if the exit is made through the "back" side of the cube ($z = -1/2$) and the value "6" if the exit is made through the "front" side ($z = 1/2$). The lines of code corresponding to these two additional outcomes are presented in Algorithm 3.

Algorithm 1 Function ”crossing”

```
function $crossing(\vec{x},\vec{v},n,m)$
    Initialize empty scalar array: $array\_times$
    for $i = 1:2$ do
        for $j = 1:2$ do
            push the value of $\frac{\frac{(-1)^j}{2} - x_i}{v_i}$ into $array\_times$
        end for
    end for
    $minpos = \infty$
    $number = 0$
    for $i = 1$:length(array_times) do
        if $array\_times_i < minpos$ and $array\_times_i > 0$ then
            $minpos = array\_times_i$
            $number = i$
        end if
    end for
    $t = array\_times_{number}$
    $\vec{x}_1 = \vec{x} + \vec{v}t$
    $d = |\vec{v}t|$
    if $number = 2$ then
        $n+ = 1$
        return $\binom{-\frac{1}{2}}{(\vec{x}_1)_2}, d, n, m$
    else if $number = 4$ then
        $m+ = 1$
        return $\binom{(\vec{x}_1)_1}{-\frac{1}{2}}, d, n, m$
    else if $number = 1$ then
        $n- = 1$
        return $\binom{\frac{1}{2}}{(\vec{x}_1)_2}, d, n, m$
    else if $number = 3$ then
        $m- = 1$
        return $\binom{(\vec{x}_1)_1}{\frac{1}{2}}, d, n, m$
    else
        error(”Don’t know which wall was hit; this should not occur.”)
    end if
end function
```

The only difference between the 3D version of the function calculating the trajectory ($collisions3d\_classical(\vec{x},\vec{v},r,t_{max})$) is the third added dimension: the coordinates and velocities are now 3D and there is the variable $l = \lfloor x_3 + \frac{1}{2} \rfloor$ which will show up in all the instances of the integer coordinates of the obstacle (n,m,l), including the output of the function $crossing3d(\vec{x},\vec{v},n,m,l)$.

Algorithm 2 Function "collisions_classical"

```
function $collisions\_classical(\vec{x}, \vec{v}, r, t_{max})$
    Initialize empty vector arrays: $places, circles, velocities$
    Initialize empty scalar array: $times$
    push $\vec{x}$ into $places$, push $\vec{v}$ into $velocities$, push 0 into $times$
    $n = \lfloor x_1 + \frac{1}{2} \rfloor$
    $m = \lfloor x_2 + \frac{1}{2} \rfloor$
    $x- = \binom{n}{m}$
    if $|\vec{x}| < r$ then
        error("The initial position cannot be inside an obstacle")
    end if
    $t = 0$
    while $t \leq t_{max}$ do
        $C = |\vec{v} \times \vec{x}|$
        $B = |\vec{v}|r$
        $D = B^2 - C^2$
        if $C < B$ and $-\vec{v} \cdot \vec{x} - \sqrt{D} > 0$ then
            $t_1 = \frac{-\vec{v} \cdot \vec{x} - \sqrt{D}}{|\vec{v}|^2}$
            $\vec{N}_0 = \vec{x} + \vec{v} t_1$
            $\vec{N} = \frac{\vec{N}_0}{|\vec{N}_0|}$
            $\vec{v}_1 = \vec{v} - 2(\vec{v} \cdot \vec{N})\vec{N}$
            $\vec{x}_1 = \vec{N}_0$
            $t+ = t_1$
            push $\vec{x}_1 + \binom{n}{m}$ to $places$
            push $\binom{n}{m}$ to $circles$
            push $\vec{v}_1$ to $velocities$
            push $t$ to $times$
            $\vec{x}, d, n, m = crossing(\vec{x}_1, \vec{v}_1, n, m)$
            $\vec{v} = \vec{v}_1$
            $t+ = \frac{d}{|\vec{v}|}$
        else
            $\vec{v}_1 = \vec{v}$
            $\vec{x}_1 = \vec{x}$
            $\vec{x}, d, n, m = crossing(\vec{x}_1, \vec{v}_1, n, m)$
            $\vec{v} = \vec{v}_1$
            $t+ = \frac{d}{|\vec{v}|}$
        end if
    end while
    return $places, circles, velocities, times$
end function
```

Algorithm 3 Modifications to ”crossing” for the 3D case

```
function $crossing3d(\vec{x}, \vec{v}, n, m, l)$
    ...
    if $number = 2$ then
        $n+ = 1$
        return $(-\frac{1}{2}, (\vec{x_1})_2, (\vec{x_1})_3)^T, d, n, m, l$
    else if $number = 4$ then
        $m+ = 1$
        return $((\vec{x_1})_1, -\frac{1}{2}, (\vec{x_1})_3)^T, d, n, m, l$
    else if $number = 1$ then
        $n- = 1$
        return $(\frac{1}{2}, (\vec{x_1})_2, (\vec{x_1})_3)^T, d, n, m, l$
    else if $number = 3$ then
        $m- = 1$
        return $((\vec{x_1})_1, \frac{1}{2}, (\vec{x_1})_3)^T, d, n, m, l$
    else if $number = 5$ then
        $l- = 1$
        return $((\vec{x_1})_1, (\vec{x_1})_2, \frac{1}{2})^T, d, n, m, l$
    else if $number = 6$ then
        $l+ = 1$
        return $((\vec{x_1})_1, (\vec{x_1})_2, -\frac{1}{2})^T, d, n, m, l$
    end if
    ...
end function
```