Random Walk in Electron Diffusion PDE: All you need to know for parallel computing

Handbook by

Peike Sun

Supervisor:

Prof Samjid Mannan

King's College London

DRAFT 2023-07-31 20:39

Acknowledgments

I would like to give my acknowledgments to Prof. Samjid Mannan, who supervised this KURF project, also to Andrew Whittington, a PhD student in the Nanoscience group at King's.

I would also like to acknowledge the funding from King's Experience Research Award, also hardware supporting from King's College London. (2022). King's Computational Research, Engineering and Technology Environment (CREATE). Retrieved July 28, 2023, from https://doi.org/10.18742/rnvf-m076

CHAPTER 1

The problem and two approaches

Now we are working to solve the PDEs with nice boundary conditions. To be more specific, we aim to solve this diffusion problem following these steps:

- Calculate the flux of electrons
- Calculate the potential from Laplace equation
- Calculate the electric field E
- Calculate the flux using $J = \sigma E$
- Calculate the atomic flex
- calculate the Joule heat
- From heat calculate temperature
- From temperature calculate the velocity of diffusion

This handbook focus on the very first step, and serves as a detailed instruction on how to calculate the flux of electrons numerically using Python. You may need some python basis to read this handbook, however, I don't assume in advance that you know how to manipulate your python models in a Unix-like operating system, so a detailed instruction will be given.

1.1 Method No.1: Electrons diffuse from one side to another

Imaging there is a rectangular domain, and electrons are released from the left side of this domain and absorbed at the right side of this domain. We established a random walk model to study this scenario shown below.

These codes are easy to read. Logically, certain number of particles are released equally spaced, at the left boundary of the domain, then after each time step, each particle is allowed to take a step with certain step length s but random direction. There are also certain boundary conditions to restrict the movements: electrons cannot be absorbed on the left, up or down boundary of the domain, so if a particle reaches such boundary, it will be bounced back. However, electrons could be absorbed at the right boundary of the domain, so if a particle reaches the right boundary, it will be absorbed and there will be no further random walk process. At the same time, a counting variable C

will be used to keep a record of particles reached the right boundary. The maximum step parameter is considered to be sufficient if the majority of the electrons are absorbed.

```
    """
x1, x2, y1, y2: lower and upper boundary of the domain
s: length of each step
PT : maximum number of steps a particle can take,
PQ: total number of particles released.
"""


def simulate_walk(x1, x2, y1, y2, s, PT, PQ):
    PW = np.zeros((x2, y2))   #record where the particles landed
    steplength = (y2-y1)/PQ
    C = 0

    for y in np.arange(y1, y2, steplength):   #for every y position
        xx = x1   # walk from x1
        yy = y

        for i in range(PT):    # walk, maximum PT steps
            r = np.random.rand() * 2 * math.pi
            dx = s * math.cos(r)    # The step length is fixed = s
            dy = s * math.sin(r)

            xx += dx
            yy += dy

            if xx > x2:     #if particle reaches the right hand side it is absorbed
                C += 1     #C is used to keep track of particles absorbed
                break

            if xx < x1:    #left if not in the domain, reflect s.t. goes back to domain
                xx = 2 * x1 - xx
                dx = -dx

            if yy < y1:
                yy = 2 * y1 - yy
                dy = -dy
```

```
            if yy > y2:
                yy = 2 * y2 - yy
                dy = -dy

            a = math.ceil(xx)-1
            b = math.ceil(yy)-1

            if a < x2:
                PW[a, b] += 1    #landed particles

    return PW, C   #the position of the particles and number of absorbed particles
```

This scenario can be enhanced to a more realistic case. In real electron diffusion problems, voids will be formed in the domain. Voids are certain subdomain where electrons cannot move into. We also developed corresponding Python scripts to model this scenario, allowing electrons to ”move a step” back if they are encountered with voids. So below codes generates a single void located at (a1, a2), (b1, b2) and simulates such case.

```
"""
x1, x2, y1, y2: lower and upper boundary of the domain
s: length of each step
PT : maximum number of steps a particle can take,
PQ: total number of particles released.
"""


def simulate_walk_void(x1, x2, y1, y2, s, PT, PQ, a1, a2, b1, b2):
    PW = np.zeros((x2, y2))   #record where the particles landed
    steplength = (y2-y1)/PQ
    C = 0

    for y in np.arange(y1, y2, steplength):   #for every y position
        xx = x1   # walk from x1
        yy = y

        for i in range(PT):    # walk, maximum PT steps
            r = np.random.rand() * 2 * math.pi
            dx = s * math.cos(r)    # The step length is fixed = s
            dy = s * math.sin(r)
```

```
        xx += dx
        yy += dy

        if xx > x2:     #if particle reaches the right hand side it is absorbed
            C += 1     #C is used to keep track of particles absorbed
            break

        if xx < x1:           #left if not in the domain, reflect
            xx = 2 * x1 - xx
            dx = -dx

        if yy < y1:
            yy = 2 * y1 - yy
            dy = -dy

        if yy > y2:
            yy = 2 * y2 - yy
            dy = -dy

        if xx > a1 and xx < a2 and yy > b1 and yy < b2:
        # if a particle moves into a void
            xx -= dx      # moves back
            yy -= dy

        a = math.ceil(xx)-1
        b = math.ceil(yy)-1

        if a < x2:
            PW[a, b] += 1    #landed particles

    return PW, C   #the position of the particles and number of absorbed particles
```

Also, it is natural to put this problem into the 3-d case —- after all, we are living in a 3-d world (before you start to think of the sounding ”cool” string theory and argue that we are actually living in a even higher-dimension world, the 3-d model in the scale of electron diffusion problem is well sufficient). Moving to the 3-d case we just need to add the front and back boundaries of the domain and apply similar reflecting boundary conditions to the up and down boundaries.

There are also other interesting changes here to pay attention to. First, you must be very careful when choosing the method to parameterized the direction on the step. At a very first glance, it

may not be very straightforward that the parameterization of x, y and z using spherical coordinates, which we are quite familiar with and have used thousands of times, is actually not a wise choice to indicate the walk direction of electrons because it won't give a even distribution along all directions, that is to say, electrons parameterized using the traditional spherical coordinates will not take the equal probability to walk in three directions. I myself fell into this trap and was advised by Andrew Whittington for a nicer parameterization method.

Another thing to pay attention to is in the 3-d case, the number of particles input in the model may be slightly different from the particles in the model. This is because unlike the 2-d case where we can simply divide the length of the left-side of the domain and establish the intervals, in the 3-d case we need to build grids to release the electrons. So the derivation of number of particles comes from the rounding problem. Imaging we asked the model to start with 101 particles and the left side of the domain is a 30×30 square, then on average each particle is taken up the place of 8.91, then we set the length of grid to be the square root of 8.91, which is 2.985, and place the initial electrons as so, then there will 100 rather than 101 particles in the grids. Python codes shown below can be used to simulate such case.

```
    """
x1, x2, y1, y2, z1, z2: lower and higher boundary of the domain
s: length of each step
PT : maximum number of steps a particle can take,
PQ: total number of particles released.
WARNING: the number of particles in the output is  slightly different to
the initial setting numbers (usually larger than input), because of rounding error,
when the y-z domain is not strictly square but rectangular. It's not very severe.
"""


def simulate_3d_walk_void(x1, x2, y1, y2, z1, z2, s, PT, PQ,
                          a1, a2, b1, b2, c1, c2):

    PW = np.zeros((x2, y2, z2))   #record where the particles landed
    area_interval = (y2*z2)/PQ
    interval = np.sqrt(area_interval)  #the interval in between starting points
    C = 0

    for y in np.arange(y1, y2, interval):   #for every y position

        for z in np.arange(z1, z2, interval):
            xx = x1   # walk from x1
            yy = y
```

```
zz = z

for i in range(PT):    # walk, maximum PT steps
    #CHECK THE RIGHT FUNCTION!!!
    theta = np.random.rand() * 2 * np.pi
    phi = np.arccos(2 * np.random.rand() - 1)
    dx = s * np.sin(phi) * np.cos(theta)  # dx
    dy = s * np.sin(phi) * np.sin(theta)  # dy
    dz = s * np.cos(phi)  # dz

    xx += dx
    yy += dy
    zz += dy

    if xx > a1 and xx < a2 and yy > b1 and yy < b2 and zz > c1 and zz < c2:
        xx -= dx      # moves back
        yy -= dy
        zz -= dz

    if xx > x2:     #if particle reaches the right hand side it is absorbed
        C += 1     #C is used to keep track of particles absorbed

    if xx < x1:           #left
        xx = 2 * x1 - xx
        dx = -dx

    if yy < y1:             #front
        yy = 2 * y1 - yy
        dy = -dy

    if yy > y2:               #back
        yy = 2 * y2 - yy
        dy = -dy

    if zz < z1:             #low
        zz = 2 * z1 - zz
        dz = -dz
```

```
            if zz > z2:                #high
                zz = 2 * z2 - zz
                dz = -dz

            a = math.ceil(xx)-1
            b = math.ceil(yy)-1
            c = math.ceil(zz)-1

            if a < x2:
                PW[a, b, c] += 1    #landed particles

    return PW, C   #the position of the particles and number of absorbed particles
```

However in real case, there could be more than one voids, so you may use the following codes to indicate certain number of voids (the location of the voids are randomly decided by the program because I assume you don't want to input the location information one by one by one).

```
import numpy as np
import math
import random


def walk_3d_voids(params):
    # Unpack parameters
    y, z, params_dict = params
    x1, x2, y1, y2, z1, z2, s, PT, num_void, void_length = params_dict.values()

    # Initialize the PW array
    PW = np.zeros((x2, y2, z2))

    # Generate voids
    random_points = [(random.uniform(x1, x2), random.uniform(y1, y2),\
    random.uniform(z1, z2)) for _ in range(num_void)]
    sub_intervals = [[(x, x + void_length), (y, y + void_length), (z, z + void_length)] \
    for x, y, z in random_points]

    # Initialize count
    C = 0
```

```
# Random walk
xx, yy, zz = x1, y, z
for _ in range(PT):
    theta = np.random.rand() * 2 * np.pi
    phi = np.arccos(2 * np.random.rand() - 1)
    dx = s * np.sin(phi) * np.cos(theta)  # dx
    dy = s * np.sin(phi) * np.sin(theta)  # dy
    dz = s * np.cos(phi)  # dz

    xx += dx
    yy += dy
    zz += dz

    # Check if a particle is in the void
    for sub_interval in sub_intervals:
        if sub_interval[0][0] <= xx <= sub_interval[0][1] and \
            sub_interval[1][0] <= yy <= sub_interval[1][1] and \
            sub_interval[2][0] <= zz <= sub_interval[2][1]:
            xx -= dx  # Moves back
            yy -= dy
            zz -= dz

    # Check if a particle reaches the boundary
    if xx > x2:
        C += 1
        break
    elif xx < x1:
        xx = 2 * x1 - xx
        dx = -dx
    if yy > y2:
        yy = 2 * y2 - yy
        dy = -dy
    elif yy < y1:
        yy = 2 * y1 - yy
        dy = -dy
    if zz > z2:
        zz = 2 * z2 - zz
```

```
            dz = -dz
        elif zz < z1:
            zz = 2 * z1 - zz
            dz = -dz

    # After maximum of steps, record the position
    a, b, c = int(np.ceil(xx))-1, int(np.ceil(yy))-1, int(np.ceil(zz))-1
    if a < x2:
        PW[a, b, c] += 1

    return PW, C

def save_results(x1, x2, y1, y2, z1, z2, s, PT, PQ, num_void, void_length):
    # The parameters to be used in walk_3d_voids_parallel
    params_dict = {'x1': x1, 'x2': x2, 'y1': y1, 'y2': y2, 'z1': z1, 'z2': z2, 's': s, \
    'PT': PT, 'num_void': num_void, 'void_length': void_length}

    # Calculate the interval
    area_interval = (y2 - y1) * (z2 - z1) / PQ
    interval = np.sqrt(area_interval)

    # Generate the y, z pairs
    yz_pairs = [(y, z, params_dict) for y in np.arange(y1, y2, interval) \
    for z in np.arange(z1, z2, interval)]

    # The main process
    PW_results = np.zeros((x2, y2, z2))
    C = 0
    for params in yz_pairs:
        pw, c = walk_3d_voids(params)
        PW_results += pw
        C += c

    # Save PW to an .npy file
    np.save('PW_results.npy', PW)

    # Save C to a .txt file
    with open('C_results.txt', 'w') as file:
```

```
        file.write(str(C))

    print("Data saved successfully.")

if __name__ == '__main__':
    save_results(x1 = 0, x2 = 20, y1 = 0, y2 = 30, z1 = 0, z2 = 15,
              s = 0.4, PT = 50000, PQ = 12000, num_void = 10, void_length = 1)
```

It is always interesting to visualize the results in a graph, which is more intuitive presentation of the results. To visualize a 3-d case, a very natural thought is to cut the 3-d cude into several 2-d slices and investigate the performance of these slices. Below python program implements this function.

```
import numpy as np
import matplotlib.pyplot as plt

def plot_solution(PW, z_range):
    plt.figure(figsize=(6,6))
    plt.imshow(PW, cmap='Blues', interpolation='nearest')
    plt.title(f'Z = {z_range[0]} ')
    plt.colorbar(label='number')
    plt.xlabel('y')
    plt.ylabel('x')
    plt.show()

def load_and_plot():
    PW = np.load('PW_results.npy')
    num_slices = 5

    # Determine z range from the data itself
    z1 = 0
    z2 = PW.shape[2]

    slice_indices = np.linspace(z1, z2, num_slices+1, endpoint=True).astype(int)

    for i in range(len(slice_indices)-1):
        PW_slice = np.sum(PW[:, :, slice_indices[i]:slice_indices[i+1]], axis=2)
        plot_solution(PW_slice, z_range=(slice_indices[i], slice_indices[i+1]))

if __name__ == "__main__":
```

```
load_and_plot()
```

To use this plotting function, you will first need to run the 3-d model above, which gives you the PW_results. You will need to save the results in a .npy file named PW_results.npy in the same directory as the plotting function, then run the plotting function. Some example pictures generated by this plotting function is shown below.

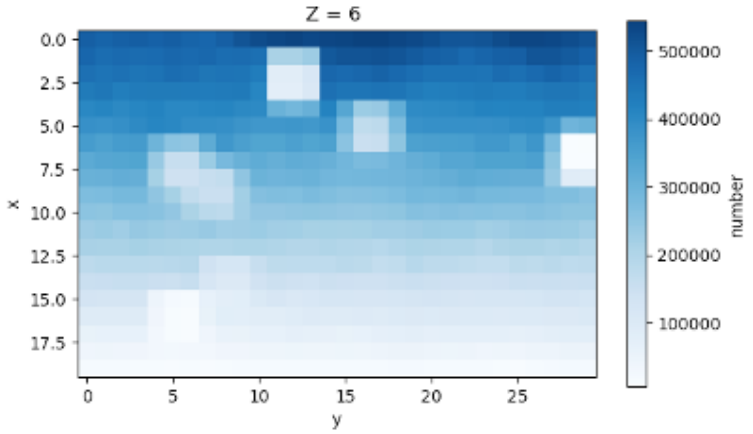

FIGURE 1.1 Insert the full caption here for this floating figure.

1.2 Method No.2: Electrons diffuse one point

not records The ”Chati” method, on the other hand, approaches this problem from another prospective. Rather than started from a boundary, it starts from a single point. Since the procedure is quite similar to the ”walk from 3d” method, we can easilyand directly jump to to the 3-d with voids case.

```
import numpy as np
import random

def prect_rand_3d(x, y, z, dt, n, a, b, c, num_voids, void_size):
    psi = 0

    random_points = []
    for _ in range(num_voids):
        random_x = random.uniform(0, a)
        random_y = random.uniform(0, b)
        random_z = random.uniform(0, c)
        random_points.append((random_x, random_y, random_z))
```

```
    sub_intervals = []
    for point in random_points:
        sub_interval = [(point[0], point[0] + void_size),
                        (point[1], point[1] + void_size),
                        (point[2], point[2] + void_size)]
        sub_intervals.append(sub_interval)

    for k in range (1, n+1):
        xe = x
        ye = y
        ze = z

        while xe > 0 and xe < a and ye > 0 and ye < b and ze > 0 and ze < c:
            dx = np.sqrt(dt) * np.random.randn()
            dy = np.sqrt(dt) * np.random.randn()
            dz = np.sqrt(dt) * np.random.randn()

            xe_new = xe + dx
            ye_new = ye + dy
            ze_new = ze + dz

            for sub_interval in sub_intervals:
                if sub_interval[0][0] <= xe_new <= sub_interval[0][1] and \
                   sub_interval[1][0] <= ye_new <= sub_interval[1][1] and \
                   sub_interval[2][0] <= ze_new <= sub_interval[2][1]:
                    xe_new -= dx  # moves back
                    ye_new -= dy
                    ze_new -= dz

            xe, ye, ze = xe_new, ye_new, ze_new



        psi += xe + ye + ze

    psi /= n
    return psi
```

Once you have this function, you can play around with it by indicating the input variables, like:

```
prect_rand_3d(x = 3, y = 4, z = 5, dt = 0.1, n = 2000, a = 10, b = 10, c = 10,
```

```
num_voids = 15, void_size = 1)
```

CHAPTER 2

Hands-on parallel computation tutorial

2.1 Method No.1: Electrons diffuse from one side to another

Now we already have our two basic models to solve the problem. If you have some basic knowledge with Python editor such as Jupyter Notebook, Colab, VSCode or whatever you like, you can just simply copy and paste the models into your .ipynb with and play with the inputs.

We want to do something even more interesting here: we want to make the computation parallel. To begin with, I will give a very easy example illustrating why are we seeking for parallel methods. You can imagine there is a swimming pool full of water in a backyard, and it stores 100 cubic meter of water, and your task is to drain the pool. The drainage capacity of each water pipe is 1 cubic meter per day. In order to complete your task and have more leisure time, you may want to add as many as possible pipes to this swimming pool, which may be constrained by your budget, because buying pipes needs money.

In our electron diffusion example, this is pretty much like the same case. We are releasing a large amount number of particles, and we assume each step of a certain particle depends on its last step (as this step's starting point), particles are independent of each other. So we can using more than one cores (which act as the pipes) and ask them to compute in parallel.

We will implement this feature in a .py file, such that it can call up more than one cores.

2.2 Method No.1: Electrons diffuse from one side to another

We will need a complete .py file, rather than blocks in .ipynb file. Once you have followed the instructions on KCL e-research website page and succeed in ssh your local computer to the HPC, you can type

```
mkdir codes
```

and press enter to establish a new folder named "codes", use

```
cd codes
```

to enter this folder, then use

```
touch 3d_voids.py
```

to establish a blank file. To open and edit this file, you can use

```
vim 3d_voids.py
```

Once you are in this file, press ”i” to enter the INSERT mode, and you can confidently copy and paste (it always works to right click the mouse to do so by the way) the below python codes in to your ”3d_voids” file:

```
import numpy as np
import math
import random
import multiprocessing as mp
import matplotlib.pyplot as plt
import time  # Import time module

def walk_3d_voids_parallel(params):
    # Unpack parameters
    y, z, params_dict, sub_intervals = params
    x1, x2, y1, y2, z1, z2, s, PT, num_void, void_length = params_dict.values()
    PW = np.zeros((x2, y2, z2))

    C = 0
    xx, yy, zz = x1, y, z
    for _ in range(PT):
        theta = np.random.rand() * 2 * np.pi
        phi = np.arccos(2 * np.random.rand() - 1)
        dx = s * np.sin(phi) * np.cos(theta)  # dx
        dy = s * np.sin(phi) * np.sin(theta)  # dy
        dz = s * np.cos(phi)  # dz

        xx += dx
        yy += dy
        zz += dz

        for sub_interval in sub_intervals:
            if sub_interval[0][0] <= xx <= sub_interval[0][1] and \
                sub_interval[1][0] <= yy <= sub_interval[1][1] and \
```

```
                sub_interval[2][0] <= zz <= sub_interval[2][1]:
                xx -= dx  # Moves back
                yy -= dy
                zz -= dz

        if xx > x2:
            C += 1
            break

        elif xx < x1:
            xx = 2 * x1 - xx

        if yy > y2:
            yy = 2 * y2 - yy

        elif yy < y1:
            yy = 2 * y1 - yy

        if zz > z2:
            zz = 2 * z2 - zz

        elif zz < z1:
            zz = 2 * z1 - zz

        a, b, c = int(np.ceil(xx))-1, int(np.ceil(yy))-1, int(np.ceil(zz))-1

        if a < x2:
            PW[a, b, c] += 1
    return PW, sub_interval[0]

def save_results(x1, x2, y1, y2, z1, z2, s, PT, PQ, num_void, void_length):
    start_time = time.time()  # Start time

    # The parameters to be used in walk_3d_voids_parallel
    params_dict = {'x1': x1, 'x2': x2, 'y1': y1, 'y2': y2, 'z1': z1, 'z2': z2, 's': s, \
    'PT': PT, 'num_void': num_void, 'void_length': void_length}

    random_points = [(random.uniform(x1, x2), random.uniform(y1, y2), \
```

```
    random.uniform(z1, z2)) for _ in range(num_void)]
    sub_intervals = [[(x, x + void_length), (y, y + void_length),\
    (z, z + void_length)] for x, y, z in random_points]

    area_interval = (y2 - y1) * (z2 - z1) / PQ
    interval = np.sqrt(area_interval)

    yz_pairs = [(y, z, params_dict, sub_intervals) for y in np.arange(y1, y2, interval) \
    for z in np.arange(z1, z2, interval)]

    with mp.Pool(4) as pool:
        results = pool.map(walk_3d_voids_parallel, yz_pairs)
    PW_results, C_results = zip(*results)

    PW = np.sum(PW_results, axis=0)
    C = np.sum(C_results)

    np.save('PW_results.npy', PW)

    with open('C_results.txt', 'w') as file:
        file.write(str(C))

    print("Data saved successfully.")

    end_time = time.time()  # End time
    print(f"{end_time - start_time} seconds.")  # Print time

if __name__ == '__main__':
    save_results(x1 = 0, x2 = 20, y1 = 0, y2 = 30, z1 = 0, z2 = 15, s = 0.4, PT = 50000, \
    PQ = 12000, num_void = 10, void_length =3)
```

You may investigate some difference here. First, a time function is inserted to monitor the running time of this program, which is quite straight forward. Second, the function is split into several functions. This is because only certain sub-steps in the whole function can be made parallel, for instance, All of the particles share the same voids, so the step of establishing random voids cannot be parallelized. However, particles walk seperately, so the walking procedure can be parallelized. The "with mp.Pool(4) as pool" line states how many cores we want to use.

Great! now press the ESC key on your keyboard to exit the INSERT mode and press "ZZ" to save and close the "3d_voids.py".

You may notice that you can use:

```
python 3d_voids.py
```

to try to run the python file. This should be totally fine on your local computer, ie. on your own MacBook. However, if you want to try the power of HPC, there is another step that you should do. In your command lines, use

```
touch 3d_voids.sh
vim 3d_voids.sh
```

command to create and open a .sh file, then press ”i” to edit it. Now cope and paste the following lines into this .sh file

```
#!/bin/bash
#SBATCH --job-name=python_parallel    # Job name
#SBATCH --partition=interruptible_cpu # Specify the interruptible_cpu partition
#SBATCH --ntasks=1                  # Run on a single node
#SBATCH --cpus-per-task=4             # Number of CPU cores per task
#SBATCH --mem=2gb                     # Total memory limit
#SBATCH --time=01:00:00               # Time limit hrs:min:sec
#SBATCH --output=job_%j.log           # Standard output and error log
python3 17jul3d.py
```

Then again press the ESC key on your keyboard to exit the INSERT mode, and press ”ZZ” to save and close this file. This .sh file tells HPC how many resource you request, for example, ”SBATCH –cpus-per-task=4” tells HPC that you want to try on 4 cores at each time.

Nearly done! The very last step is to submit this .sh file to HPC, and you should be able to do so using below codes:

```
sbatch 17jul3d.sh
```

Then you will see your jobID, You can always use

```
squeue -u <your user name>
```

to check the status. Under the ”ST” column, you may see ”PD” indicating pending, ”R” indicating running, and if there is no rows, it means you have no tasks running on HPC —- so the calculation is done! Use:

```
sacct -j <jobID> --format=JobID,JobName,Elapsed
```

to check the running details, in particular, you will find the running time.

2.3 Method No.2: Chati Method

The Chati method is rather trivial. Actually, you won't "that" need the parallel process for the Chati method since it is this method itself is not heavy in computing. However, for the completeness, I still wrote a tutorial for the parallel computing in Chati method. Similar to previous steps, you may use:

```
touch chatipara.py
```

to establish a blank file. To open and edit this file, you can use

```
vim chatipara.py
```

Then copy and paste below python codes:

```
import numpy as np
import random
from multiprocessing import Pool
import time

def prect_rand_3d(x, y, z, dt, n, a, b, c, num_voids, void_size):
    psi = 0
    random_points = []
    for _ in range(num_voids):
        random_x = random.uniform(0, a)
        random_y = random.uniform(0, b)
        random_z = random.uniform(0, c)
        random_points.append((random_x, random_y, random_z))

    sub_intervals = []
    for point in random_points:
        sub_interval = [(point[0], point[0] + void_size),
                        (point[1], point[1] + void_size),
                        (point[2], point[2] + void_size)]
        sub_intervals.append(sub_interval)

    for k in range (1, n+1):
        xe = x
        ye = y
        ze = z
```

```
        while xe > 0 and xe < a and ye > 0 and ye < b and ze > 0 and ze < c:
            dx = np.sqrt(dt) * np.random.randn()
            dy = np.sqrt(dt) * np.random.randn()
            dz = np.sqrt(dt) * np.random.randn()

            xe_new = xe + dx
            ye_new = ye + dy
            ze_new = ze + dz

            for sub_interval in sub_intervals:
                if sub_interval[0][0] <= xe_new <= sub_interval[0][1] and \
                   sub_interval[1][0] <= ye_new <= sub_interval[1][1] and \
                   sub_interval[2][0] <= ze_new <= sub_interval[2][1]:
                    xe_new -= dx  # moves back
                    ye_new -= dy
                    ze_new -= dz

            xe, ye, ze = xe_new, ye_new, ze_new

        psi += xe + ye + ze

    psi /= n
    return psi

def main(x, y, z, dt, n_total, a, b, c, num_voids, void_size, cores):
    n_per_core = n_total // cores
    with Pool(cores) as p:
        results = p.starmap(prect_rand_3d, [(x, y, z, dt, n_per_core, a, \
        b, c, num_voids, void_size)] * cores)

    result = sum(results) / cores
    return result

if __name__ == '__main__':
    start_time = time.time()
    result = main(x = 3, y = 4, z = 5, dt = 0.1, n_total = 2000, a = 10, \
    b = 10, c = 10, num_voids = 15, void_size = 1, cores = 2)
    end_time = time.time()
```

```
    print(f"The result is: {result}")
    print(f"Execution time: {end_time - start_time} seconds")
```

Use ESC to exit the INSERT mode and press ZZ in your keyboard to exit this file. Create the .sh file:

```
touch chatipara.sh
vim chatipara.sh
```

then press "i" to edit it. Now cope and paste the following lines into this .sh file

```
#!/bin/bash
#SBATCH --job-name=python_parallel    # Job name
#SBATCH --partition=interruptible_cpu # Specify the interruptible_cpu partition
#SBATCH --ntasks=1                   # Run on a single node
#SBATCH --cpus-per-task=2             # Number of CPU cores per task
#SBATCH --mem=2gb                     # Total memory limit
#SBATCH --time=01:00:00               # Time limit hrs:min:sec
#SBATCH --output=job_%j.log           # Standard output and error log
python3 chatipara.py
```

There you go! Submit this to HPC using:

```
sbatch chatipara.sh
```

Then you will see your jobID, You can always use

```
squeue -u <your user name>
```

to check the status. Under the "ST" column, you may see "PD" indicating pending, "R" indicating running, and if there is no rows, it means you have no tasks running on HPC —- so the calculation is done! Use:

```
sacct -j <jobID> --format=JobID,JobName,Elapsed
```

to check the running details, in particular, you will find the running time.