11.3 Classical gases with a microscopic thermometer

Question 1

In the microcanonical ensemble, each microstate is equally likely. This means that a higher probability of finding the thermometer with energy E_d relates to a greater number of microstates corresponding to it.

We now need to consider what these microstates that correspond to $P(E_d)$ are.

In the microcanonical ensemble the energy is fixed such that the exchange of energy between thermometer and gas is governed by $E_d = E_{tot} - E_g$. Hence, E_d depends only on E_g and thus the number of microstates corresponding to a particular E_d goes with the number of microstates of the gas with corresponding energy E_g.

Therefore, $P(E_d) \propto \Omega_g(E_g)$.

Question 2

We start with the definition of the temperature of the gas and rewrite this using the chain rule.

$$\frac{1}{T} = \frac{\partial S_g}{\partial E_g} \tag{1}$$

$$= \frac{\partial S_g}{\partial E_d}\frac{\partial E_d}{\partial E_g} \tag{2}$$

Energy is conserved such that $E_d = E_{tot} - E_g$, therefore $\frac{\partial E_d}{\partial E_g} = -1$.

$$\frac{1}{T} = -\frac{\partial S_g}{\partial E_d} \tag{3}$$

This can be integrated to give:

$$S_g(E_g) = -\frac{E_d}{T} + c \tag{4}$$

where c is a constant of integration. We now equate equation 4 with the definition of entropy for the gas: $S_g(E_g) = k_B \ln \Omega_g(E_g)$.

$$-\frac{E_d}{T} + c = k_B \ln \Omega_g(E_g) \tag{5}$$

$$\Omega_g(E_g) \propto \exp\left(-\frac{E_d}{k_B T}\right) \tag{6}$$

The proportionality is introduced as c and k_B are constants. From question 1, we established $P(E_d) \propto \Omega_g(E_g)$.

$$\therefore\ P(E_d) \propto \exp\left(-\frac{E_d}{k_B T}\right) \tag{7}$$

Question 3

The program *monte_carlo_acceptance.py* on page 20 performs Monte Carlo simulations of a 3-dimensional gas of N non-relativistic particles. The program evaluates perturbations to the momentum and energy of the particles. The acceptance/rejection algorithm ensures the system stays in the microcanonical ensemble (ME), where different microstates can be accessed while keeping the total energy conserved.

The program *3d_non_relativistic.py* on page 21 gives figure 1, figure 2 and figure 3 below which present the results of the program as histograms with 50 bins used each, all normalised to 1.

Results and Dependence on $N_{\mathbf{sweeps}}$

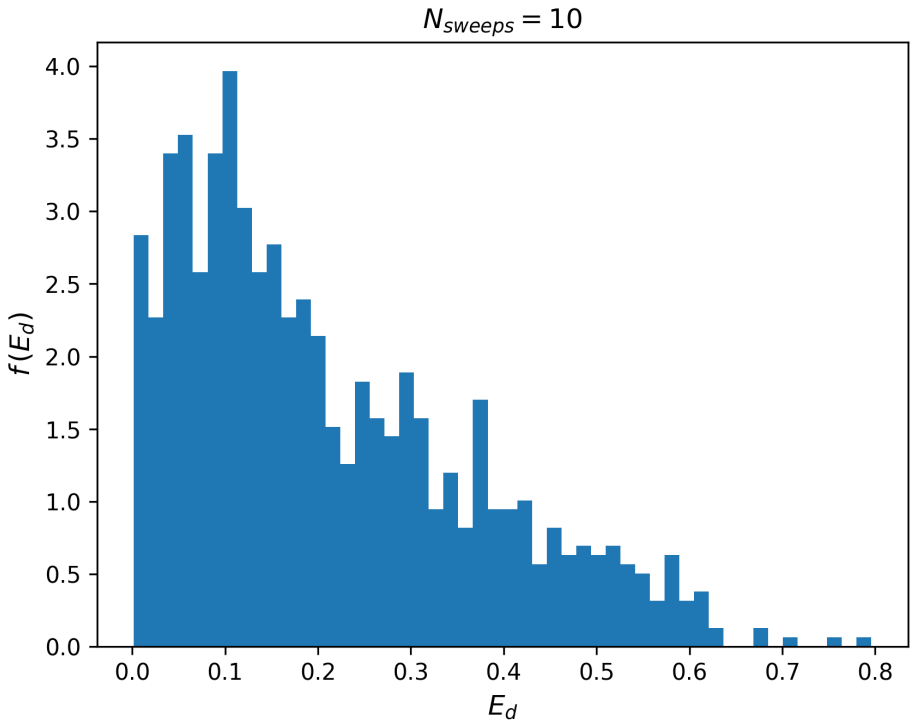


Figure 1: Monte Carlo distribution of thermometer energies for $N_{\text{sweeps}} = 10$.

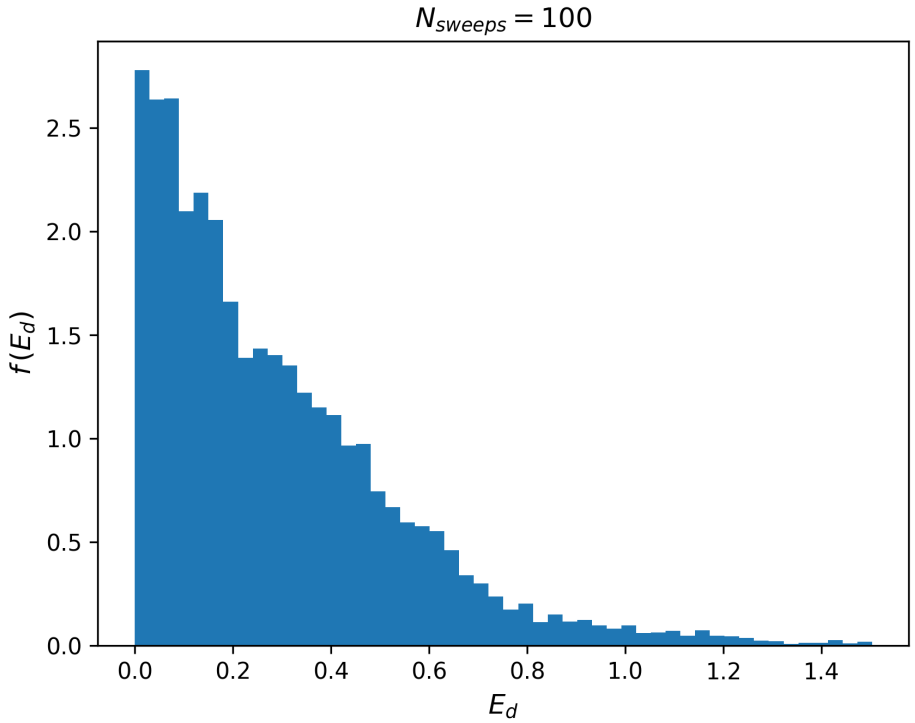


Figure 2: Monte Carlo distribution of thermometer energies for $N_{\text{sweeps}} = 100$.

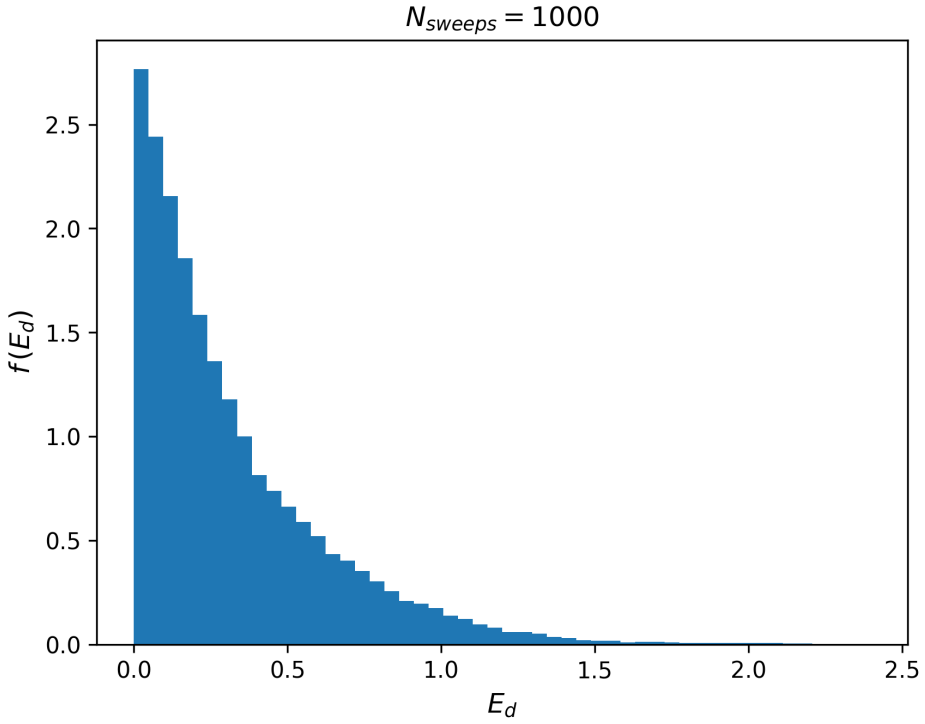


Figure 3: Monte Carlo distribution of thermometer energies for $N_{\text{sweeps}} = 1000$.

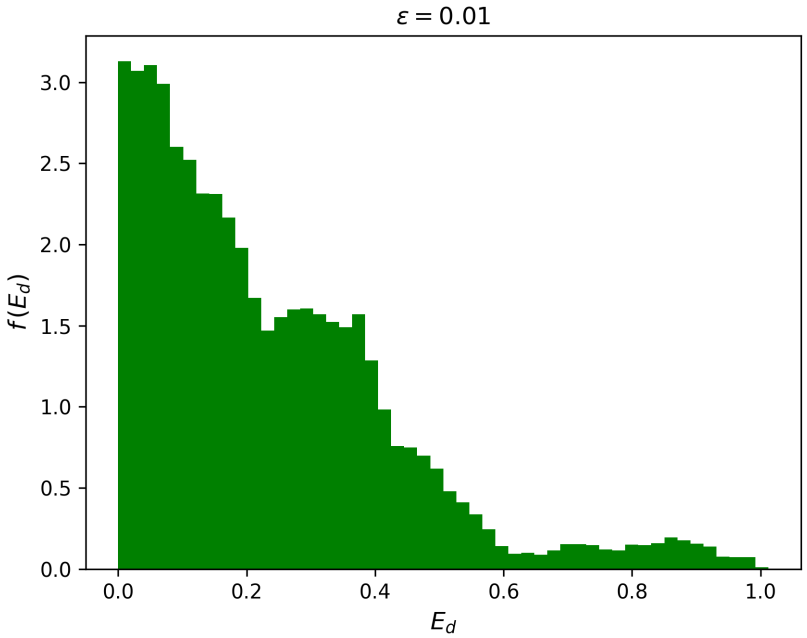
We see that as N_{sweeps} increases to 1000, there is an increase in the frequency of lower bins of E_d. We see that the highest occurrence across all 3 histograms is the zero E_d bin, which becomes more pronounced with increasing N_{sweeps}. With increasing N_{sweeps}, the histograms fluctuate less and there is a clearer overall shape to them. The histograms go from being relatively randomly distributed to becoming more stable with N_{sweeps}.

As the histograms are normalised, we can treat the histograms as probability density functions in this analysis. Therefore, the peak at zero E_d gives the most probable energy state - the one with the most number of equally likely microstates corresponding to it. From here, the distribution decreases more and more - indicating higher E_d states are less likely to be found.

This makes sense considering the theory of the ME in question 2, where the $P(E_d)$ decreases exponentially with E_d. As N_{sweeps} increases, each particle in the gas undergoes more updates allowing for a better representation of the frequencies of each bin. With an increasing number of acceptances/rejections in the program, the E_d distribution will tend towards a more stable state that more accurately represents the ME. So, the higher the value of N_{sweeps}, the more accurate the representation of the ME we achieve.

Dependence on ϵ

Above we just considered $\epsilon = 0.1$. To examine the effects of the parameter ϵ, which controls the size of the momentum perturbations, consider figure 4 below plotted for a fixed $N_{\text{sweeps}} = 1000$ (as this is the best representation of the ME out of the 3 previous histograms) for different values of ϵ.



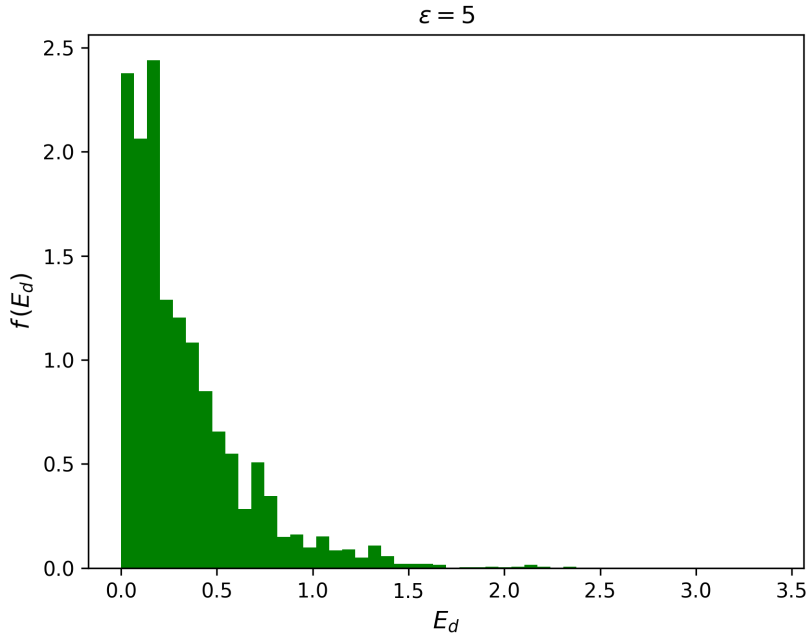


Figure 4: Monte Carlo distribution of thermometer energies for $\epsilon = 0.01$ and $\epsilon = 5$. N_{sweeps} is fixed at 1000.

There is a broader range of E_d with increasing ϵ. This is because the perturbations become larger, so the proposed energy E_{prop} has the potential to be greater than the current value E_{curr}. For example, we can have an acceptance in the algorithm where $\Delta E \leq E_d$ but ΔE is quite large such that the updated value of E_d is therefore significantly larger than before.

Too large of an ϵ causes more fluctuations in the histogram - not in equilibrium state despite having a large enough N_{sweeps}. This is shown in the $\epsilon = 5$ histogram above.

Also, too small of an ϵ results in a less stable distribution too as seen in the $\epsilon = 0.01$ histogram. A smaller variety of E_d states are explorde by the particles.

Therefore, to achieve a more accurate ME representation we need the parameter ϵ to be in between these two cases.

So the results indeed depend on ϵ.

Question 4

From this point, we set $k_B = 1$, working in natural units.

By setting $N_{\text{sweeps}} = 1000$, N_{sweeps} is large enough that it is in an equilibrium state, so running the simulation multiple times leads to approximately the same resulting histogram.

Comparison with $P(E_d)$

By plotting the distribution $P(E_d) \propto \exp\left(-\frac{E_d}{T}\right)$ on the histogram, we can see whether the distribution in the histogram is consistent the equation.

The histogram can be turned into a probability density function by normalising the area of all the bins to 1. This allows for comparison with $P(E_d)$, which needs to be normalised also.

Using some normalisation constant A and the fact that E_d is non-negative, we can write:

$$\int_0^\infty P(E_d)\, dE_d = 1 \tag{8}$$

$$A \int_0^\infty e^{-\frac{E_d}{T}}\, dE_d = 1 \tag{9}$$

$$A = \frac{1}{T} \tag{10}$$

$$\therefore\ P(E_d) = \frac{1}{T} \exp\left(-\frac{E_d}{T}\right) \tag{11}$$

The program *non_rel_temperature.py* runs the Monte Carlo algorithm and fits the curve in equation 11 to the histogram to find an estimate for the temperature. A value of T is found by using the histogram data with the *scipy* library which optimises the value of T with which the exponential curve is fitted to the histogram. The plot is shown in figure 5 below. This method gives an estimate of:

$$T = 0.319 \pm 0.001 \tag{12}$$

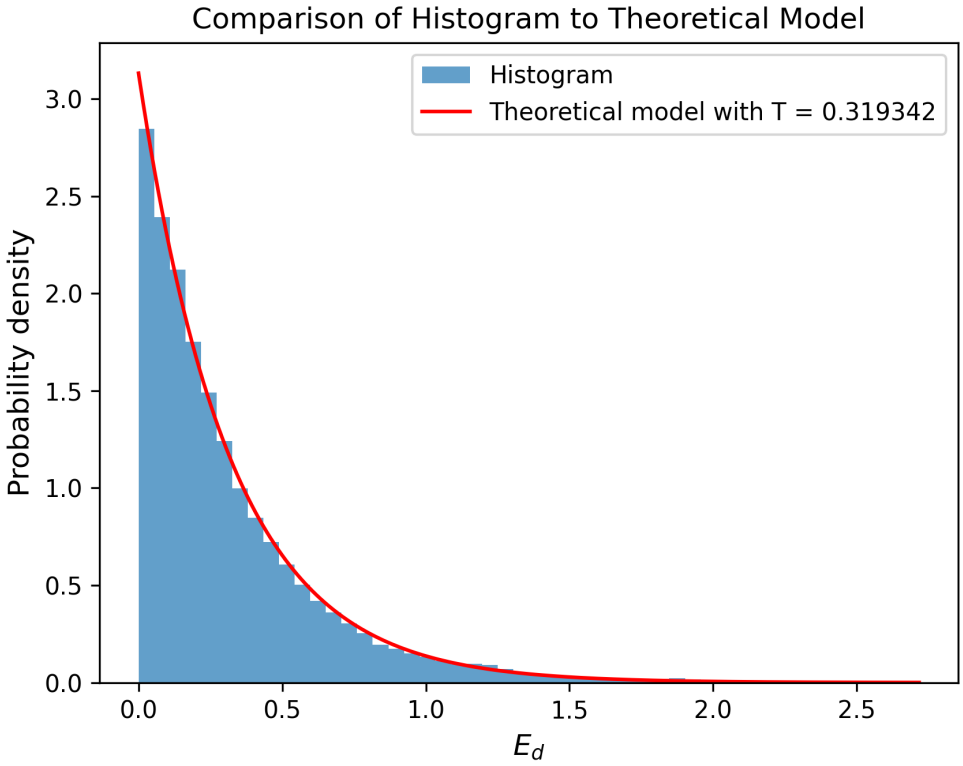


Figure 5: Monte Carlo probability distribution and theoretical probability distribution of thermometer energies for $N_{\text{sweeps}} = 1000$ equilibrium state, both normalised to 1.

Estimating T from $\langle E_d \rangle$

Figure 5 shows that the histogram of E_d is consistent with the theoretical distribution. Hence, we can alternatively estimate T directly from the average value of E_d.

$$\langle E_d \rangle = T \tag{13}$$

The program computes the average energy $\langle E_d \rangle$ and its error from its array. The error used for the average is $\sigma_{\langle E_d \rangle} = \sigma_{E_d} / \sqrt{N_{\text{updates}}}$.

$$T = 0.329 \pm 0.001 \tag{14}$$

We see that both estimates of the temperature are indeed consistent with each other as they produce similar values, of the same order of magnitude. This verifies the theoretical distribution.

We see that both estimates of the temperature differ from each other by approximately 3% relative to either one. As this difference is relatively small, we can deduce that the average energy method to compute T is appropriate.

Question 5

The program *non_rel_single_particle.py* on page 23 produces figure 6 below. It presents the ideal gas distribution and the single particle energy E_i corresponding to the i^{th} particle. If rejection occurs, the *monte_carlo.py* program sets the single particle energy as $E_i = E_{curr}$ (unperturbed energy). If accepted, then the program sets $E_i = E_{prop}$ (perturbed energy) for that particular change.

Consistency with ideal gas theory

To check whether the single particle energy histogram is consistent with the theory of ideal gases, we plot the Maxwell distribution for energy. This can be derived from the Maxwell distribution of speeds as below.

$$f(E) = f(v)\frac{dv}{dE} \tag{15}$$

$f(v), f(E)$ denote the distribution for speeds and energy respectively. The equality ensures the energy distribution is normalised to 1 as in $f(v)$.

For $k_B = m = 1$, we relate the kinetic energy via $E = \frac{1}{2}v^2$ and substitute in the Maxwell distribution.

$$f(E) = \frac{1}{(2\pi T)^{3/2}}\, 4\pi v^2\, e^{-\frac{v^2}{2T}}\, \frac{dv}{dE} \tag{16}$$

$$= \frac{1}{(2\pi T)^{3/2}}\, 8\pi E\, e^{-\frac{E}{T}}\, \frac{1}{\sqrt{2E}} \tag{17}$$

$$= 2\pi\, E^{1/2}\, \frac{1}{(\pi T)^{3/2}}\, e^{-\frac{E}{T}} \tag{18}$$

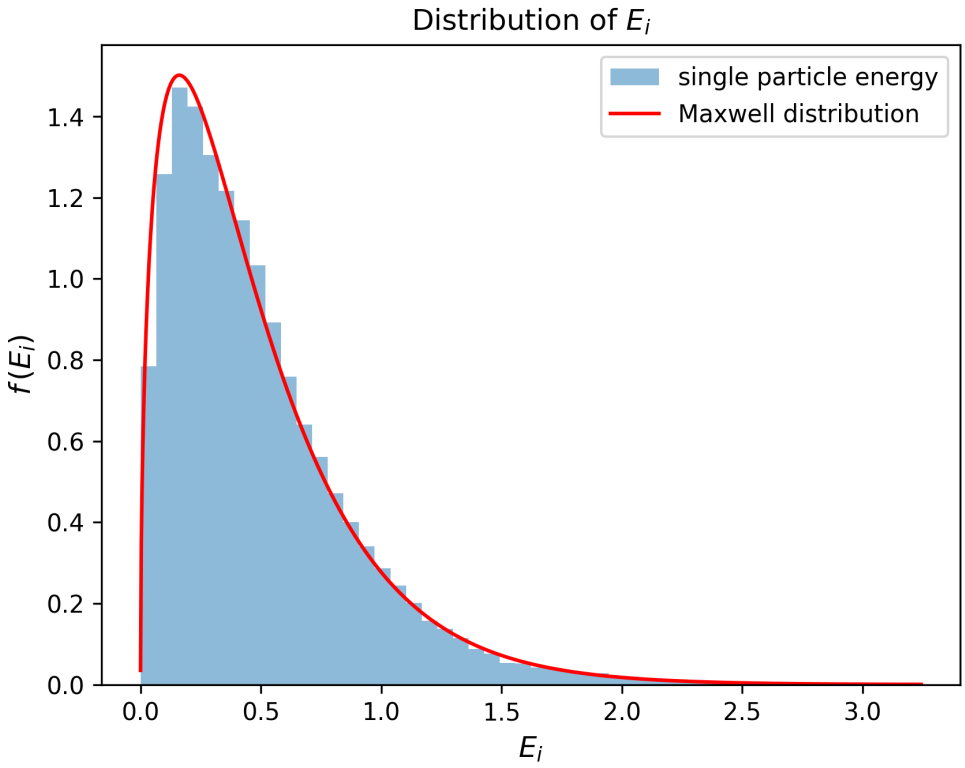


Figure 6: Monte Carlo and Maxwell distribution of single particle energy for the equilibrium state of $N_{\text{sweeps}} = 1000$. Both distributions have been normalised to 1.

From figure 6, we can see that histogram is consistent with the Maxwell distribution. Therefore, this verifies that the histogram represents an ideal gas.

Question 6

The program *random_momentum_ideal.py* on page 24 runs the Monte Carlo simulation for a 3D ideal gas, where all particles are initialised with a random momentum.

For each value of a used, the corresponding value of the perturbation parameter ϵ is determined from $\epsilon = \frac{a}{10}$. This ensures the perturbations are neither too small or large. This maintains the shape of the histograms so that they are in the equilibrium state.

Effects of $a = 1$

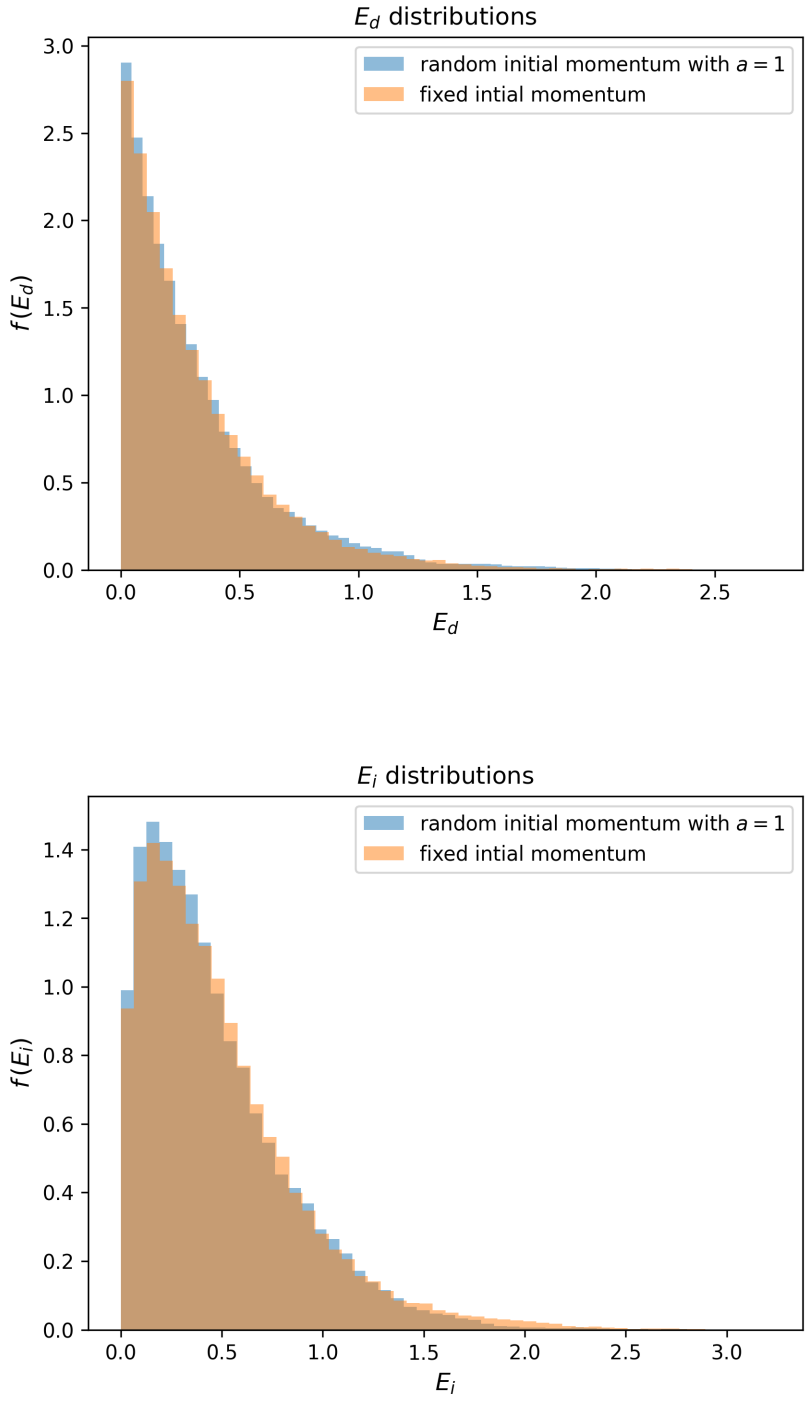


Figure 7: Fixed and randomised momentum initial conditions plotted together for direct comparison.

From figure 7, we see that this change in initial conditions does not affect the histograms of E_d and E_i. A very similar equilibrium state is reached. This is consistent with the theory of ideal gases from statistical physics, where over enough time, a system accesses all available microstates so that the chance of finding the equilibrium state in any one of these microstates is the same. Therefore, initial conditions do not have an effect.

Effects of varying a

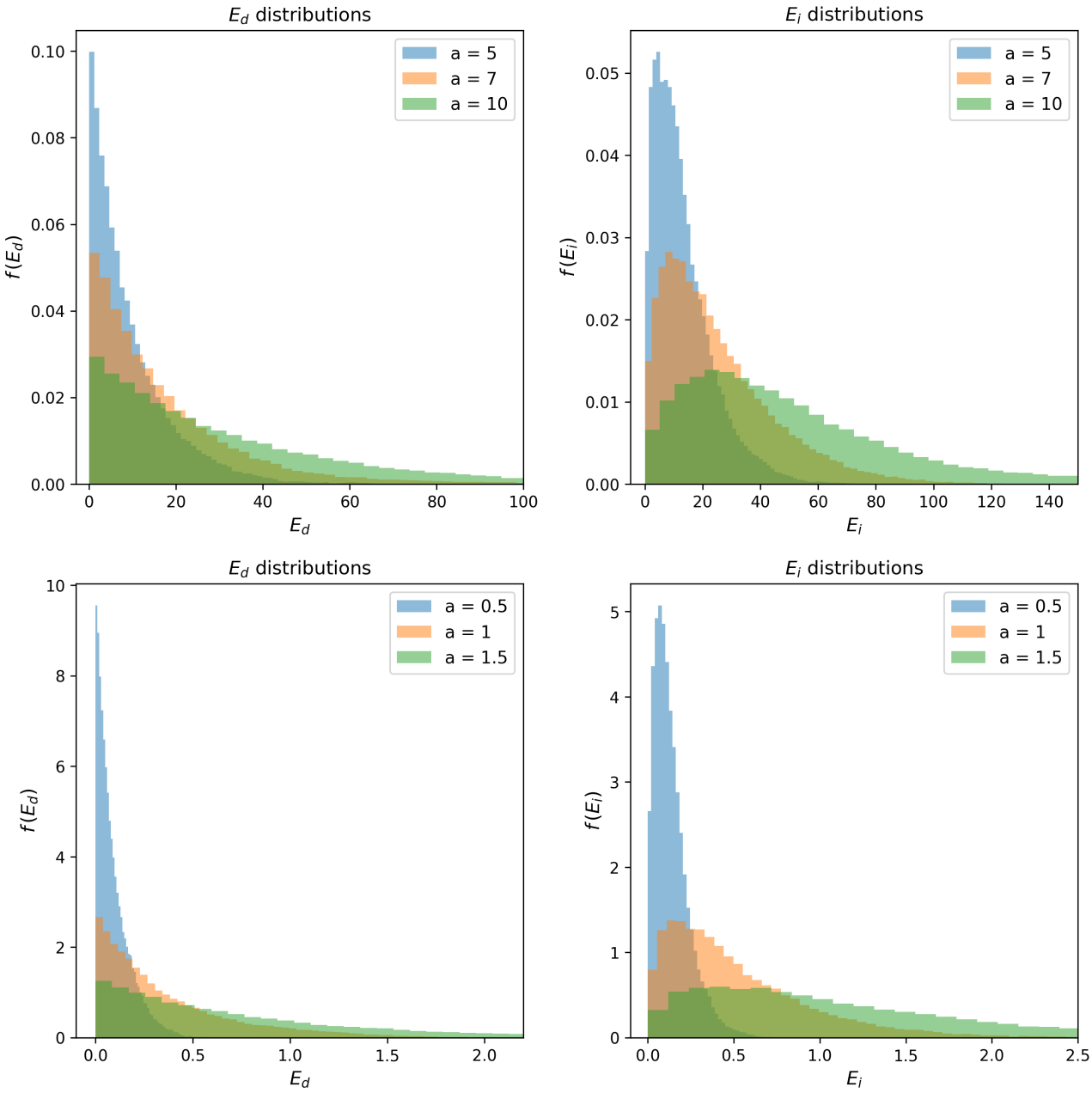


Figure 8: Normalised E_d and single particle energies for multiple values of a.

Looking particularly at the single particle energy histograms in figure 8, we see they are still consistent with the Maxwell distribution despite the different scales of energy and momentum. The theory of ideal gases includes the Maxwell distribution curves having a broader and smaller peak as temperature increases. The histograms show this relationship as a is related to momentum and therefore kinetic energy and temperature.

Temperature dependence on a

To examine the exact relationship between T and a, the program runs the simulation for multiple values of a in the range $0 < a < 10$. Using the resulting temperatures, a curve has been fitted to the data to determine relationship in figure 9 below.

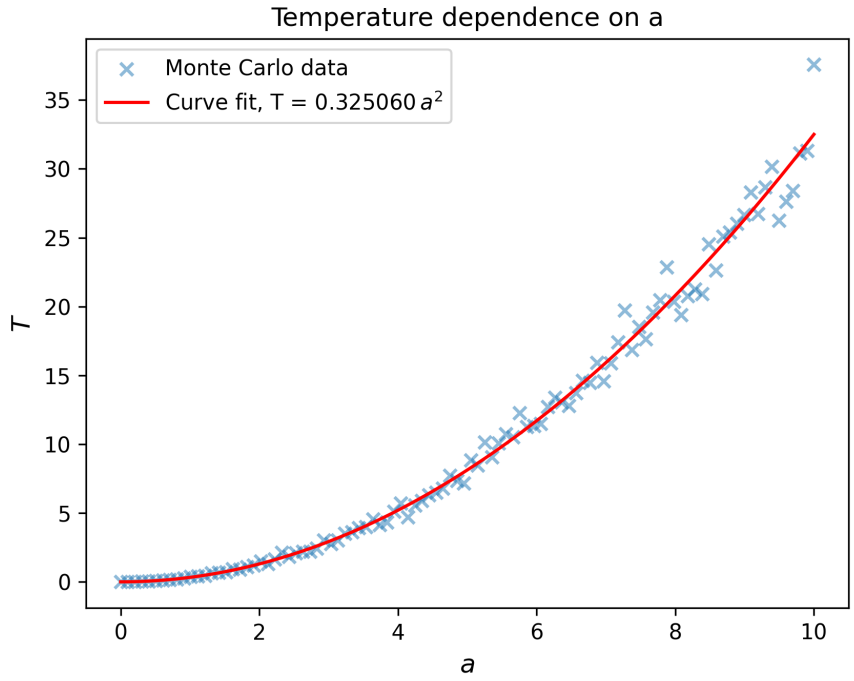


Figure 9: Fixed and randomised momentum initial conditions plotted together for direct comparison.

We see that the relationship is quadratic in a: $T = Ca^2$, where C is a constant from the curve fit determined to be $C = 0.325 \pm 0.002$.

This is consistent with ideal gas theory, where the equipartition of energy gives us $\langle E \rangle = \frac{3}{2} k_B T$ for a single particle of a 3D ideal gas. Increasing a, increases the momentum and therefore single particle energy. Due to this relation, the temperature will increase, therefore verifying the above result that T increases with a.

Question 7

The program *2d_relativistic.py* on page 28 continues with random initial conditions but for a 2D ultra-relativistic gas. Single particle energy is now calculated via $E(\boldsymbol{p}) = |\boldsymbol{p}|$.

Results for $a = 1$

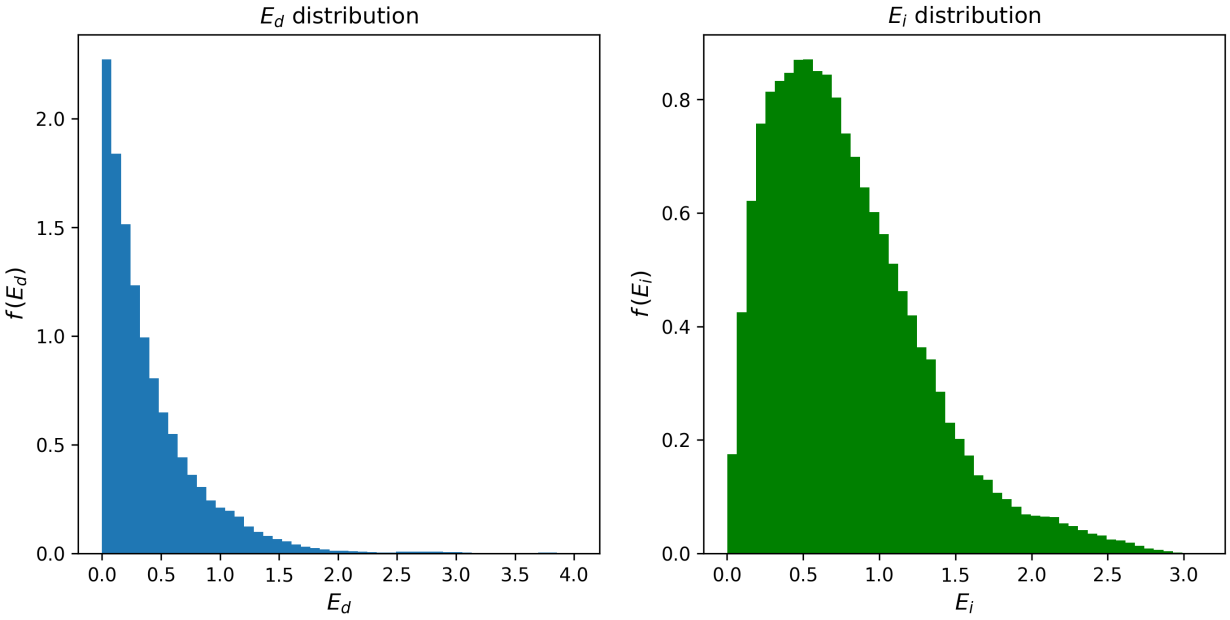


Figure 10: E_d and single particle energies plotted for $a = 1$ for a 2D relativistic gas.

Using $\langle E_d \rangle = T$, the program determines the temperature of the gas to be:

$$T = 0.401$$

We see that the single particle energy histogram peaks at a smaller value of E_i in figure 10 than in the corresponding histogram in question 5. Therefore, the gas temperature is greater in this ultra-relativistic case than in the non-relativistic case.

Varying a

The temperatures at a various values of a have been computed with their corresponding total energies E_{tot} in figure 11. This can be computed using the conservation of energy and that $E_d \ll E_g$. The energy of the gas is just the sum of the energies of the N particles after the simulation has ended.

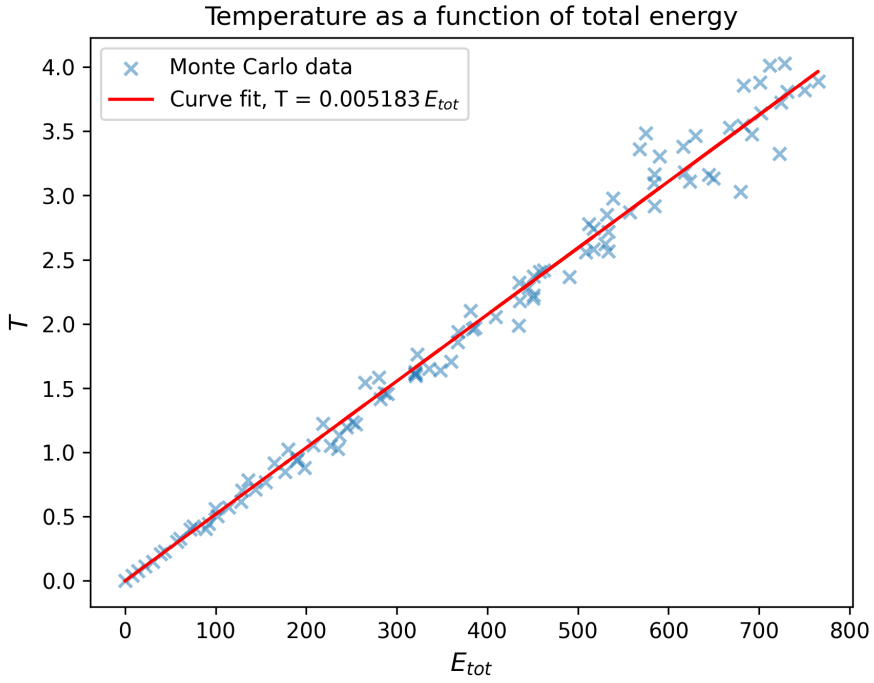


Figure 11: T and corresponding values of E_{tot} plotted for various values of a within the range $0 < a < 10$.

We see that this 2D ultra-relativistic gas demonstrates a linear relationship between T and E_{tot}. This differs from the non-relativistic gas in question 5 where we later deduced the quadratic relationship between T and a.

This makes sense from the theory where ultra-relativistic gases have energy scaling linearly with momentum $|\boldsymbol{p}|$, whereas for the ideal gas case energy scales quadratically with $|\boldsymbol{p}|$.

Question 8

The program *3d_relativistic.py* on page 30 continues with random initial conditions but for a 3D relativistic gas. Single particle energy is now calculated via $E(\boldsymbol{p}) = \sqrt{1 + |\boldsymbol{p}|^2} - 1$.

Temperature dependence on total energy

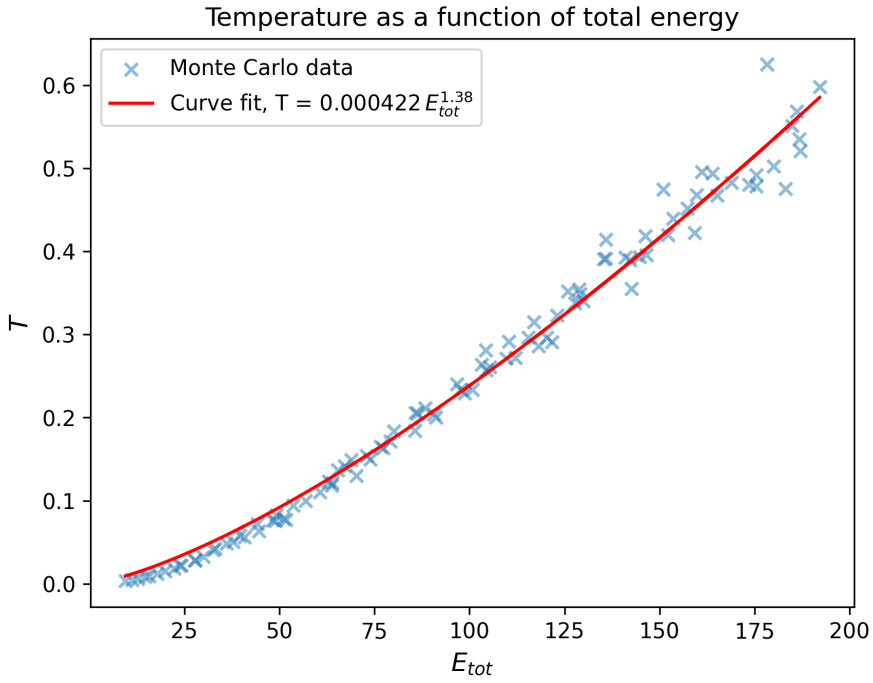


Figure 12: T and corresponding values of E_{tot} plotted for various values of a within the range $0.1 < a < 2$. A curve fit has been performed of the form $T = \alpha\, E_{\text{tot}}^{\beta}$, where the fit parameters are determined as $\alpha = 0.000387$ and $\beta = 1.40$.

For large values of $|\boldsymbol{p}|$, $|\boldsymbol{p}|^2 \gg 1 \;\therefore\; E(\boldsymbol{p}) \approx |\boldsymbol{p}|$, which is the same as in the 2D ultra-relativistic gas.

For small values of $|\boldsymbol{p}|$, we can binomially expand $E(\boldsymbol{p})$.

$$E(\boldsymbol{p}) = (1 + |\boldsymbol{p}|^2)^{\frac{1}{2}} - 1 \tag{19}$$

$$\approx 1 + \frac{1}{2}\,|\boldsymbol{p}|^2 - 1 \tag{20}$$

$$= \frac{|\boldsymbol{p}|^2}{2} \tag{21}$$

This is the same case as in the 3D ideal gas.

Smaller values of $\boldsymbol{p}$ arise from smaller values of a. This is captured in the lower end of figure 12. Here, T appears initially quadratic with E_{tot} which matches the relationship in question 6 for the 3D ideal gas.

As T increases with a, $\boldsymbol{p}$ will become larger. This is seen towards the higher end of the graph. Here, T becomes increasingly linear with E_{tot} instead, which is consistent with the 2D ultra-relativistic case in question 7.

Hence, we see that increasing E_{tot} results in a transition from an ideal, non-relativistic gas at low temperatures, into an ultra-relativistic gas at high temperatures.

Comparison of E_i for different cases

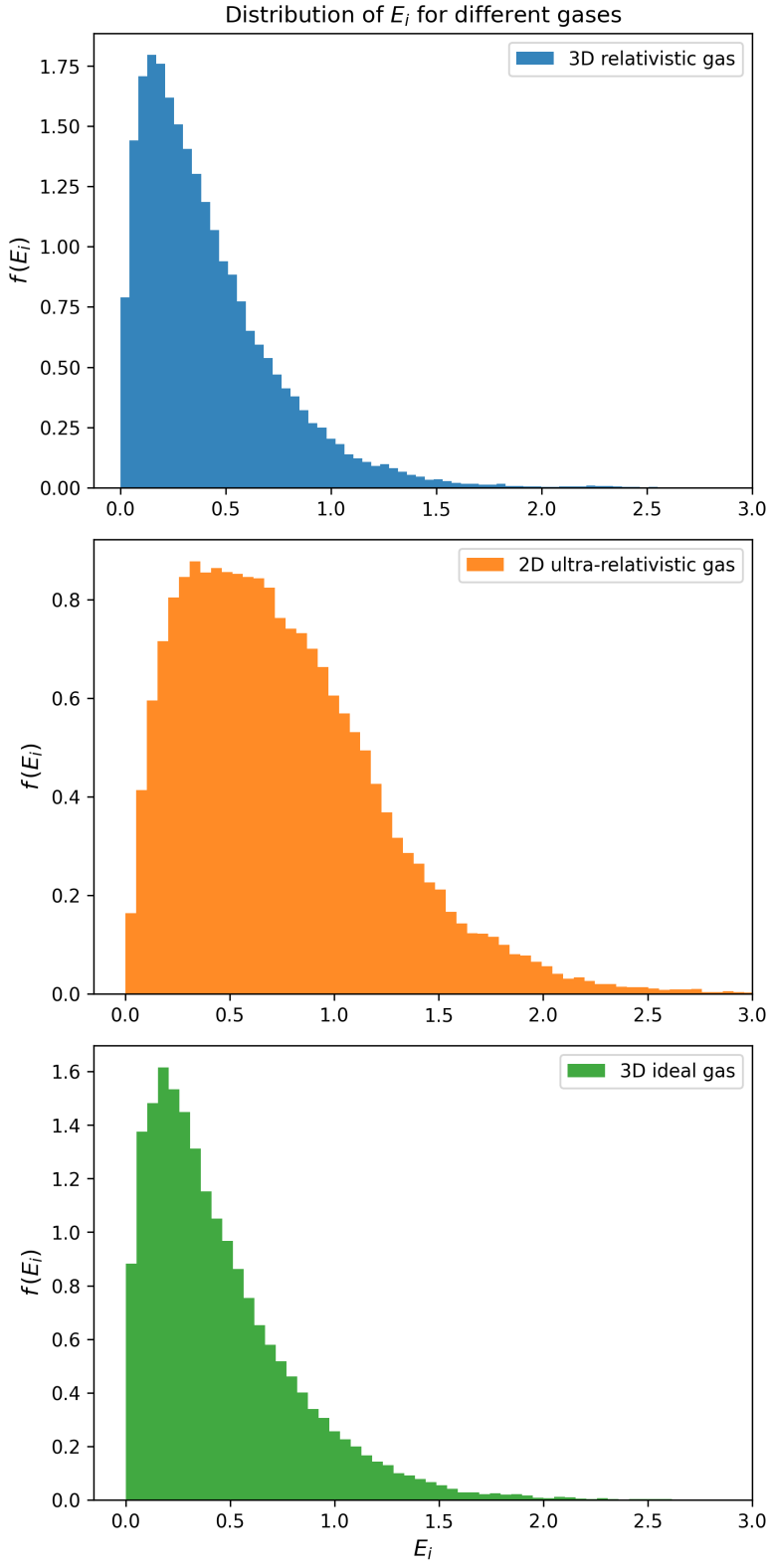


Figure 13: Single particle energy from a few representative cases: 3D ideal gas, 2D ultra-relativistic gas and a 3D relativistic gas.

The height of each peak represents the spread of energy states occupied. The breadth of the distribution represents the average energy and therefore gas temperature.

We note that the 3D relativistic gas is a generalisation of other 2 gases. We can recover the 2D ultra-relativistic histogram for large $|\boldsymbol{p}|$ and recover the 3D ideal gas histogram for small $|\boldsymbol{p}|$.

Therefore, we can deduce from figure 13 that as the momentum of the particles increases, the histograms become broader and demonstrate a higher T. Also, there is a wider spread of energy states - the most likely energy state is not as distinct as it is for gases with low momentum.

Programs

monte_carlo_acceptance.py - used in Questions 3,4,5

```
import numpy as np

def monte_carlo(N, N_sweeps, epsilon):
    N_updates = N * N_sweeps

    ### STEP 1: intialise momentum for all N particles as e_1 and
       initialise E_d
    p = np.array([1.,0.,0.])  # set momentum to e_1 for one particle
    p_i = []

    for i in range(0,N):
        p_i.append(p)

    p_i = np.array(p_i) # momentum set to e_1 for all particles
    E_d = 0

    single_particle_energy_values = []
    E_d_values = []

    ### STEP 2: randomising N_updates amount of times
    for n in range(int(N * N_sweeps)):
        index = np.random.randint(N)
        p_curr = p_i[index]
        E_curr = np.dot(p_curr, p_curr) / 2 # computing current energy
            of particle E_curr

        delta_p = np.random.uniform(-epsilon, epsilon, 3)  # random
           perturbation
        p_prop = p_curr + delta_p
        E_prop = np.dot(p_prop, p_prop) / 2  # new proposed energy
           after perturbation

        delta_E = E_prop - E_curr

        ### STEP 3: accept or reject
        if delta_E <= E_d:  # accept
            p_i[index] = p_prop
            E_d -= delta_E

        # reject so do nothing, if statement is ignored
        E_d_values.append(E_d)
        single_particle_energy_values.append(E_curr)

    return np.array(E_d_values), np.array(
       single_particle_energy_values)
```

3d_non_relativistic.py - used in Question 3

```
import numpy as np
import matplotlib.pyplot as plt
from monte_carlo_acceptance import monte_carlo


eps = 0.1
energy_therm_10, energy_single_particle_10 = monte_carlo(100,10,eps)
energy_therm_100, energy_single_particle_100 = monte_carlo(100,100,eps
    )
energy_therm_1000, energy_single_particle_1000 = monte_carlo(100,1000,
    eps)

thermom_energies = [energy_therm_10, energy_therm_100,
    energy_therm_1000]
number_of_sweeps = [10,100,1000]

for i in range(0,3):

    plt.figure(figsize=(7,5))
    plt.hist(thermom_energies[i], bins=50)
    plt.title(f'$N_{{sweeps}} = {number_of_sweeps[i]}$')
    plt.xlabel('$E_d$', fontsize = 12)
    plt.ylabel('$log(f\, (E_d))$', fontsize = 12)
    plt.yscale('log')
    # plt.savefig(f'E_d histograms {number_of_sweeps[i]} sweeps.png',
        dpi=300)
    plt.show()

a1, b1 = monte_carlo(100,1000,1)
a2, b2 = monte_carlo(100,1000,0.1)
a3, b3 = monte_carlo(100,1000,0.01)

thermom_energies = [a1, a2, a3]
epsilon_number = [1,0.1,0.01]

for i in range(0,3):

    plt.figure(figsize=(7,5))
    plt.hist(thermom_energies[i], bins=50)
    plt.title(f'$\epsilon = {epsilon_number[i]}$')
    plt.xlabel('$E_d$', fontsize = 12)
    plt.ylabel('$log(f\, (E_d))$', fontsize = 12)
    plt.yscale('log')
    # plt.savefig(f'Varying epsilon {epsilon_number[i]} sweeps.png',
        dpi=300)
    plt.show()
```

non_rel_temperature.py - used in Question 4

```
from monte_carlo_acceptance import monte_carlo
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

N = 100
N_sweeps = 1000
N_updates = N * N_sweeps
energy_therm, energy_single_particle = monte_carlo(N,N_sweeps,0.1)

### Extracting data from the histogram, using them to find the
    temperature using curve fit, then plot both
ydata, bins = np.histogram(energy_therm, bins=100, density=True)
xdata = (bins[:-1] + bins[1:]) / 2

def theoretical_model(E, T):   # the prob distribution in eqn (1)
    return (1 / T) * np.exp(-E / T)

popt, pcov = curve_fit(theoretical_model, xdata, ydata, 1)
T = popt[0]
T_std_error = np.sqrt(pcov[0, 0])

thermom_energy_values = np.linspace(min(energy_therm), max(
    energy_therm), 1000) # data to plot the curve now
prob_density_values = theoretical_model(thermom_energy_values, T)

plt.hist(energy_therm, bins = 50, density = True, alpha=0.7, label='
    Histogram')
plt.plot(thermom_energy_values, prob_density_values, color = 'r',
    label = f'Theoretical model with T = {T:f}')
plt.title('Comparison of Histogram to Theoretical Model')
plt.xlabel('$E_d$', fontsize = 12)
plt.ylabel('Probability density', fontsize = 12)
plt.legend()
# plt.savefig('Temperature estimates comparison.png', dpi=300)
plt.show()

print(f'Curve fitting method T = {T}')
print(f'Error = {T_std_error}')

average = np.mean(energy_therm)
error_average = np.std(energy_therm)

print('\n')
print(f'Average method T = {average}')
print(f'Error = {error_average / np.sqrt(N_updates)}') #sigma / root(
    N_updates)
```

non_rel_single_particle.py - used in Question 5

```
import numpy as np
import matplotlib.pyplot as plt
from monte_carlo_acceptance import monte_carlo

energy_therm, energy_single_particle = monte_carlo(100,1000,0.1)

### Plotting Maxwell-Boltzmann distribution from theory
T = np.mean(energy_therm)
print(T)

def maxwell_boltzmann(E):
    return 2 * np.pi * E**(0.5) * 1/(np.pi * T)**(1.5) * np.exp(-E / T
       )

E_values = np.linspace(min(energy_single_particle), max(
   energy_single_particle), 1000)
plt.hist(energy_single_particle, bins=50, alpha = 0.5, density=True,
   label='single particle energy')
plt.plot(E_values, maxwell_boltzmann(E_values), 'r', label = 'Maxwell
   distribution')
plt.title('Distribution of $E_i$')
plt.xlabel('$E_i$', fontsize = 12)
plt.ylabel('$f\, (E_i)$', fontsize = 12)
plt.legend()
# plt.savefig('Single_maxwell.png',dpi=300)
plt.show()
print(np.mean(energy_therm))
```

random_momentum_ideal.py - used in Question 6

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from monte_carlo_acceptance import monte_carlo

def monte_carlo_random(N, N_sweeps, epsilon,a):
    N_updates = N * N_sweeps

    ### STEP 1: intialise momentum for all N particles as e_1 and
       initialise E_d
    p_i = []

    for i in range(0,N):
        p = np.random.uniform(-a,a,3)  # set random momentum each
           particle
        p_i.append(p)

    p_i = np.array(p_i)
    E_d = 0

    single_particle_energy_values = []
    E_d_values = []

    ### STEP 2: randomising N_updates amount of times
    for n in range(int(N * N_sweeps)):
        index = np.random.randint(N)
        p_curr = p_i[index]
        E_curr = np.dot(p_curr, p_curr) / 2 # computing current energy
            of particle E_curr

        delta_p = np.random.uniform(-epsilon, epsilon, 3)  # random
           perturbation
        p_prop = p_curr + delta_p
        E_prop = np.dot(p_prop, p_prop) / 2  # new proposed energy
           after perturbation

        delta_E = E_prop - E_curr

        ### STEP 3: accept or reject
        if delta_E <= E_d:  # accept
            p_i[index] = p_prop
            E_d -= delta_E

        # reject so do nothing, if statement is ignored
        E_d_values.append(E_d)
        single_particle_energy_values.append(E_curr)

    return np.array(E_d_values), np.array(
       single_particle_energy_values)

energy_therm, energy_single_particle = monte_carlo_random(100, 1000,
```

```
   0.1, 1)
x,y = monte_carlo(100, 1000, 0.1)

print(np.mean(energy_single_particle))

A = 0.5
plt.hist(energy_therm, bins=50, density=True, alpha = A, label = '
   random initial momentum with $a = 1$')
plt.hist(x, bins=50, density=True, alpha = A, label = 'fixed intial
   momentum')
plt.title('$E_d$ distributions')
plt.xlabel('$E_d$', fontsize = 12)
plt.ylabel('$f\, (E_d)$', fontsize = 12)
plt.legend()
# plt.savefig('E_d random vs fixed momentum for a=1', dpi=300)
plt.show()

plt.hist(energy_single_particle, bins=50, density=True, alpha = A,
   label = 'random initial momentum with $a = 1$')
plt.hist(y, bins=50, density=True, alpha = A, label = 'fixed intial
   momentum')
plt.title('$E_i$ distributions')
plt.xlabel('$E_i$', fontsize = 12)
plt.ylabel('$f\, (E_i)$', fontsize = 12)
plt.legend()
# plt.savefig('E_i random vs fixed momentum for a=1', dpi=300)
plt.show()

energy_therm10, energy_single_particle10 = monte_carlo_random(100,
   1000, 0.7, 7)
energy_therm9, energy_single_particle9 = monte_carlo_random(100, 1000,
    0.5, 5)
energy_therm8, energy_single_particle8 = monte_carlo_random(100, 1000,
    1, 10)
energy_therm7, energy_single_particle7 = monte_carlo_random(100, 1000,
    0.05, 0.5)
energy_therm6, energy_single_particle6 = monte_carlo_random(100, 1000,
    0.1, 1)
energy_therm5, energy_single_particle5 = monte_carlo_random(100, 1000,
    0.15, 1.5)

fig, ax = plt.subplots(figsize=(10, 10), nrows=2, ncols=2)
A = 0.5
b=60

ax[0,0].hist(energy_therm9, bins=b, density=True, alpha=A, label='a =
   5')
ax[0,0].hist(energy_therm10, bins=b, density=True, alpha=A, label='a =
    7')
ax[0,0].hist(energy_therm8, bins=b, density=True, alpha=A, label='a =
   10')
ax[0,0].set_title('$E_d$ distributions', fontsize = 12)
ax[0,0].set_xlabel('$E_d$', fontsize = 12)
ax[0,0].set_xlim(-3, 100)
```

```
ax[0,0].set_ylabel('$f \, (E_d)$', fontsize = 12)
ax[0,0].legend(fontsize = 11)

ax[0,1].hist(energy_single_particle9, bins=b, density=True, alpha=A,
   label='a = 5')
ax[0,1].hist(energy_single_particle10, bins=b, density=True, alpha=A,
   label='a = 7')
ax[0,1].hist(energy_single_particle8, bins=b, density=True, alpha=A,
   label='a = 10')
ax[0,1].set_title('$E_i$ distributions', fontsize = 12)
ax[0,1].set_xlabel('$E_i$', fontsize = 12)
ax[0,1].set_xlim(-5, 150)
ax[0,1].set_ylabel('$f \, (E_i)$', fontsize = 12)
ax[0,1].legend(fontsize = 11)

ax[1,0].hist(energy_therm7, bins=b, density=True, alpha=A, label='a =
   0.5')
ax[1,0].hist(energy_therm6, bins=b, density=True, alpha=A, label='a =
   1')
ax[1,0].hist(energy_therm5, bins=b, density=True, alpha=A, label='a =
   1.5')
ax[1,0].set_title('$E_d$ distributions', fontsize = 12)
ax[1,0].set_xlabel('$E_d$', fontsize = 12)
ax[1,0].set_xlim(-0.1, 2.2)
ax[1,0].set_ylabel('$f \, (E_d)$', fontsize = 12)
ax[1,0].legend(fontsize = 11)

ax[1,1].hist(energy_single_particle7, bins=b, density=True, alpha=A,
   label='a = 0.5')
ax[1,1].hist(energy_single_particle6, bins=b, density=True, alpha=A,
   label='a = 1')
ax[1,1].hist(energy_single_particle5, bins=b, density=True, alpha=A,
   label='a = 1.5')
ax[1,1].set_title('$E_i$ distributions', fontsize = 12)
ax[1,1].set_xlabel('$E_i$', fontsize = 12)
ax[1,1].set_xlim(-0.1, 2.5)
ax[1,1].set_ylabel('$f \, (E_i)$', fontsize = 12)
ax[1,1].legend(fontsize = 11)

plt.tight_layout()
plt.savefig('E_i_random_vs_fixed_momentum_for_different_a.png', dpi
   =300, bbox_inches='tight')
plt.show()

N = 100
N_sweeps = 1000
a_values = np.linspace(0,10,100)
temperatures = np.zeros_like(a_values)

for i in range(0, len(a_values)):
    energy_therm2, energy_single_particle2 = monte_carlo_random(100,
       1000, a_values[i]/10, a_values[i])
    current_temp = np.mean(energy_therm2)
    temperatures[i] = current_temp
```

```
def theoretical_model(x,C):   # the prob distribution in eqn (1)
    return C * x**2

popt, pcov = curve_fit(theoretical_model, a_values, temperatures)
C_fit = popt[0]
C_std_error = np.sqrt(pcov[0, 0])

plt.scatter(a_values, temperatures, alpha=0.5, marker = 'x', label = '
   Monte Carlo data')
plt.plot(a_values, theoretical_model(a_values, C_fit),'r', label = f'
   Curve fit, T = ${C_fit:4f} \, a^2$')
plt.title('Temperature dependence on a')
plt.xlabel('$a$', fontsize = 12)
plt.ylabel('$T$', fontsize = 12)
plt.legend()
# plt.savefig('Temperature dependence on a', dpi=300)
plt.show()

print(C_fit)
print(C_std_error)
```

2d_relativistic.py - used in Question 7

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

def monte_carlo_random(N, N_sweeps, epsilon,a):
    N_updates = N * N_sweeps

    ### STEP 1: intialise momentum for all N particles as e_1 and
       initialise E_d
    p_i = []

    for i in range(0,N):
        p = np.random.uniform(-a,a,2)  # set random momentum each
           particle
        p_i.append(p)

    p_i = np.array(p_i)
    E_d = 0

    single_particle_energy_values = []
    E_d_values = []

    ### STEP 2: randomising N_updates amount of times
    for n in range(int(N * N_sweeps)):
        index = np.random.randint(N)
        p_curr = p_i[index]
        E_curr = np.sqrt(np.dot(p_curr, p_curr)) # computing current
           energy of particle E_curr

        delta_p = np.random.uniform(-epsilon, epsilon, 2)  # random
           perturbation
        p_prop = p_curr + delta_p
        E_prop = np.sqrt(np.dot(p_prop, p_prop))  # new proposed
           energy after perturbation

        delta_E = E_prop - E_curr

        ### STEP 3: accept or reject
        if delta_E <= E_d:  # accept
            p_i[index] = p_prop
            E_d -= delta_E

        # reject so do nothing, if statement is ignored
        E_d_values.append(E_d)
        single_particle_energy_values.append(E_curr)

    return np.array(E_d_values), np.array(
       single_particle_energy_values), p_i


energy_therm, energy_single_particle, mom = monte_carlo_random(100,
   1000, 0.1, 1) # a = 1
```

```
print(f'Temperature of gas = {np.mean(energy_therm)}')

fig, ax = plt.subplots(figsize = (11,5), nrows = 1, ncols = 2)

ax[0].hist(energy_therm, bins = 50, density = True)
ax[0].set_title('$E_d$ distribution', fontsize = 12)
ax[0].set_ylabel('$f \, (E_d)$', fontsize = 12)
ax[0].set_xlabel('$E_d$', fontsize = 12)

ax[1].hist(energy_single_particle, bins = 50, density = True, color =
   'green')
ax[1].set_title('$E_i$ distribution', fontsize = 12)
ax[1].set_ylabel('$f \, (E_d)$', fontsize = 12)
ax[1].set_xlabel('$E_d$', fontsize = 12)

plt.savefig('rel a=1', dpi = 300, bbox_inches='tight')
plt.show()
```

3d_relativistic.py - used in Question 8

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from random_momentum_ideal import monte_carlo_random
import importlib
relativistic_module = importlib.import_module("2d_relativistic")
monte_carlo_random_2d = relativistic_module.monte_carlo_random_2d

def monte_carlo_random_3d(N, N_sweeps, epsilon,a):
    N_updates = N * N_sweeps

    ### STEP 1: intialise momentum for all N particles as e_1 and
       initialise E_d
    p_i = []

    for i in range(0,N):
        p = np.random.uniform(-a,a,3)  # set random momentum each
           particle
        p_i.append(p)

    p_i = np.array(p_i)
    E_d = 0

    single_particle_energy_values = []
    E_d_values = []

    ### STEP 2: randomising N_updates amount of times
    for n in range(int(N * N_sweeps)):
        index = np.random.randint(N)
        p_curr = p_i[index]
        E_curr = np.sqrt(1 + np.dot(p_curr, p_curr)) - 1   # computing
            current energy of particle E_curr

        delta_p = np.random.uniform(-epsilon, epsilon, 3)  # random
           perturbation
        p_prop = p_curr + delta_p
        E_prop = np.sqrt(1 + np.dot(p_prop, p_prop)) - 1    # new
           proposed energy after perturbation

        delta_E = E_prop - E_curr

        ### STEP 3: accept or reject
        if delta_E <= E_d:  # accept
            p_i[index] = p_prop
            E_d -= delta_E

        # reject so do nothing, if statement is ignored
        E_d_values.append(E_d)
        single_particle_energy_values.append(E_curr)

    return np.array(E_d_values), np.array(
```

```
        single_particle_energy_values), p_i


energy_therm, energy_single_particle, mom = monte_carlo_random_3d(100,
    1000, 0.1, 1) # a = 1

print(f'Temperature of gas = {np.mean(energy_therm)}')

fig, ax = plt.subplots(figsize = (11,5), nrows = 1, ncols = 2)

ax[0].hist(energy_therm, bins = 50, density = True)
ax[0].set_title('$E_d$ distribution', fontsize = 12)
ax[0].set_ylabel('$f \, (E_d)$', fontsize = 12)
ax[0].set_xlabel('$E_d$', fontsize = 12)

ax[1].hist(energy_single_particle, bins = 50, density = True, color =
   'green')
ax[1].set_title('$E_i$ distribution', fontsize = 12)
ax[1].set_ylabel('$f \, (E_i)$', fontsize = 12)
ax[1].set_xlabel('$E_i$', fontsize = 12)

# plt.savefig('rel a=1', dpi = 300, bbox_inches='tight')
plt.show()

a_values = np.linspace(0.1,2,100)
temperatures = np.zeros_like(a_values)
total_energies = np.zeros_like(temperatures)

for i in range(0, len(a_values)):
    energy_therm2, energy_single_particle2, mom2 =
       monte_carlo_random_3d(100, 1000, a_values[i]/10, a_values[i])
    current_temp = np.mean(energy_therm2)
    temperatures[i] = current_temp

    ### Calc total energy by from E_tot = (N * E_i) + E_d, determine
       E_i from the final momentum of each particle
    particle_energies = np.zeros(100)
    for j in range(0, 100):
        current_energy = np.sqrt(abs(np.dot(mom2[j], mom2[j])))
        particle_energies[j] = current_energy

    total_energy = np.sum(particle_energies)
    total_energies[i] = total_energy


def theoretical_model(x,alpha,beta):   # the prob distribution in eqn
   (1)
    return alpha * x**beta

popt, pcov = curve_fit(theoretical_model, total_energies, temperatures
   )
alpha_fit = popt[0]
beta_fit = popt[1]
alpha_std_error = np.sqrt(pcov[0, 0])
```

```
curve_fit_data = theoretical_model(total_energies, alpha_fit, beta_fit
    )

plt.scatter(total_energies, temperatures, alpha = 0.5, marker = 'x',
    label = 'Monte Carlo data')
plt.plot(total_energies, curve_fit_data, color = 'r', label = f'Curve
    fit, T = ${alpha_fit:.6f} \, E_{{tot}}^{{{beta_fit:.2f}}}$')
plt.title('Temperature as a function of total energy')
plt.xlabel('$E_{{tot}}$', fontsize = 12)
plt.ylabel('$T$', fontsize = 12)
plt.legend()
# plt.savefig('3D Temperature dependence on E_tot', dpi=300,
    bbox_inches='tight')
plt.show()

print(alpha_fit)
print(beta_fit)
print(alpha_std_error)

energy_therm_ideal, energy_single_particle_ideal = monte_carlo_random
    (100, 1000, 0.1, 1)
energy_therm_2d, energy_single_particle_2d, mom_2d =
    monte_carlo_random_2d(100, 1000, 0.1, 1)
energy_therm_3d, energy_single_particle_3d, mom_3d =
    monte_carlo_random_3d(100, 1000, 0.1, 1)

A = 0.9
b = 60

fig, ax = plt.subplots(figsize=(6,12), nrows = 3)
ax[0].hist(energy_single_particle_3d, bins=b, alpha = A, density=True,
     label='3D relativistic gas')
ax[1].hist(energy_single_particle_2d, bins=b, alpha = A, density=True,
     label='2D ultra-relativistic gas', color='C1')
ax[2].hist(energy_single_particle_ideal, bins=b, alpha = A, density=
    True, label='3D ideal gas', color='C2')

ax[0].set_title('Distribution of $E_i$ for different gases')
ax[2].set_xlabel('$E_i$', fontsize = 12)
ax[0].set_xlim(right = 3)
ax[1].set_xlim(right = 3)
ax[2].set_xlim(right = 3)
ax[0].set_ylabel('$f\, (E_i)$', fontsize = 12)
ax[1].set_ylabel('$f\, (E_i)$', fontsize = 12)
ax[2].set_ylabel('$f\, (E_i)$', fontsize = 12)
ax[0].legend()
ax[1].legend()
ax[2].legend()
plt.tight_layout()
plt.savefig('final comparison',dpi=300, bbox_inches='tight')
plt.show()

print(np.mean(energy_therm_ideal), np.mean(energy_therm_2d), np.mean(
    energy_therm_3d))
```