TP3 : Spin Boson Hamiltonian (SBH) - PES Modelling Laboratory

The adiabatic SBH model represents two-state systems (such as spin), which is coupled to a surrounding environment. The environment represents that of a harmonic oscillator, which are called bosons, hence the name of the model.

The SBH model is used to calculate a potential energy surface for a given particle, which is given by a series of analytical functions, hence it is used to calculate many properties such as coupling, energies, and energy gradients. Since the functions in the model are analytical, the Schrodinger equation does not need to be solved in order to obtain these properties.

The SBH model is given as the equation below for a two state system coupled to N harmonic oscillators. This gives two adiabatic PES's which represents the ground state potential energy (i=1) and the excited state potential energy (i=2). R are the nuclear co-ordinates.

$$E_i = \frac{1}{2}\sum_{j=1}^{N} M_j\omega_j^2 R_j^2 + (-1)^i[\eta^2 + v_0^2]^{\frac{1}{2}} \qquad (i = 1, 2) \qquad eq. [1]$$

where

$$\eta = (\sum_{j=1}^{N} g_j R_j + \varepsilon_0)$$

Computing the derivative provides the potential energy gradient of the two adiabatic states, which is given as the following:

$$\frac{\partial E_i}{\partial R_k} = M_j\omega_j^2 R_j^2 + (-1)^i g_k \frac{\eta}{[\eta^2 + v_0^2]^{\frac{1}{2}}} \qquad (i = 1, 2;\ k = 1...N) \qquad eq. [2]$$

In this coding lab, the SBH model is reduced down to one dimension and to its lowest energy state:

$$(i = 1; N = 1)$$

The energy and energy gradient will be computed for a given input of the nuclear geometry. A graph of each energy against nuclear co-ordinates R will be produced, which will then be used to locate the minimun of the two parabolic shapes in the potential energy surface.

Task 1:

1.1: Compute the adaibatic SBH model's energy for the lowest state (i=1) and in one dimension (N=1).

From the equation [1], we can obtain the SBH model's energy in 1D using the given restrictions:

$$E_i = \frac{1}{2}M_1\omega_1^2R_1^2 + (-1)^i[\eta^2 + v_0^2]^{\frac{1}{2}} \qquad (i = 1) \qquad eq.[3]$$

where

$$\eta = (g_1R_1 + \varepsilon_0)$$

In [1]:
```python
import numpy as np
# Varaibles from table 1
m = 1836
omega_1 = 0.01136364
g_1 = 0.22
e_0 = 0.03674933
v_0 = 0.03674933


# Function to calculate energy E_i
def sbh_energy(R_1):
    # First term: sum over (1/2 * M_j * omega_j^2 * R_j^2)
    energy_term_1 = 0.5 * (m * omega_1**2 * R_1**2)

    # Second term: (-1)^i * [eta^2 + v_0^2]^(1/2)
    eta = g_1 * R_1 + e_0
    energy_term_2 = (-1)**1 * np.sqrt(eta**2 + v_0**2) #i = 1 for the lowes
t state

    # Total energy E_i
    E_i = energy_term_1 + energy_term_2
    return E_i

# Input
R_1 = 1 # Example geometry

# Calculate energy E_i for a given geometry
E_i = sbh_energy(R_1)
print(f'Energy E_i for the given geometry: {E_i}')
```

```
Energy E_i for the given geometry: -0.14082255450774128
```

Task 1:

1.2: Compute the adaibatic SBH model's energy gradient for the lowest state (i=1) and in one dimension (N=1).

From the equation [1] , we can obtain the SBH model's energy in 1D using the given restrictions and computing the derivative:

$$\frac{\partial E_i}{\partial R_k} = M_j \omega_j^2 R_j^2 + (-1)^i g_k \frac{\eta}{[\eta^2 + v_0^2]^{\frac{1}{2}}} \qquad (i = 1;\ k = 1) \qquad eq. [4]$$

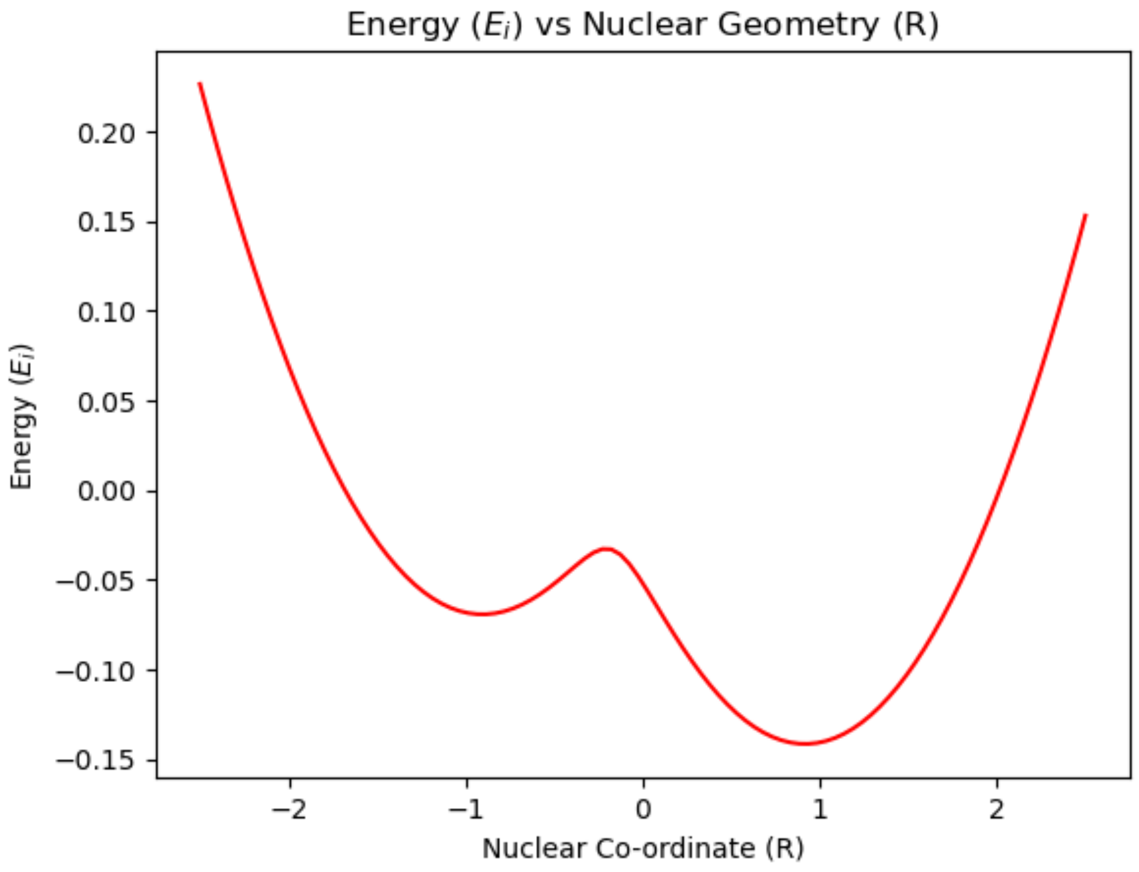
```
In [2]: # Function to calculate energy E_i
def sbh_dE_dR(R_1):
    # First term: (1/2 * M_j * omega_j^2 * R_j^2)
    energy_term_1 = (m * omega_1**2 * R_1)

    # Second term: (-1)^i * [eta^2 + v_0^2]^(1/2)
    eta = g_1 * R_1 + e_0
    energy_term_2 = ((-1)**1) * (g_1 * (eta)/(np.sqrt(eta**2 + v_0**2))) #i
= 1 for the lowest state

    # Total energy E_i
    dE_i = energy_term_1 + energy_term_2
    return dE_i

# Input
R_1 = 1 # Example geometry

# Calculate energy E_i for a given geometry
dE_i = sbh_dE_dR(R_1)
print(f"Energy E_i for the given geometry: {dE_i}")
```

```
Energy E_i for the given geometry: 0.019306461978089468
```

Task 2:

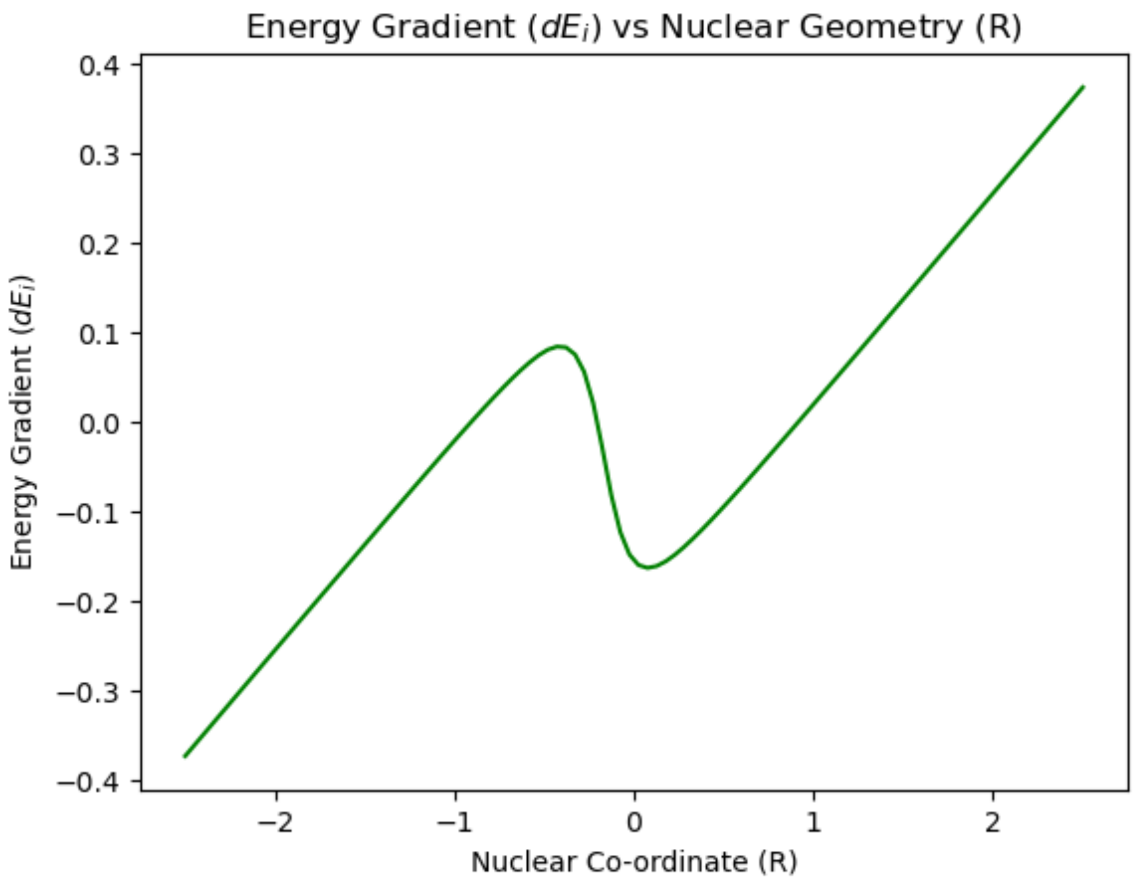
2.1: Using the function defined in *1.1*, a plot of Energy E_i vs Geometry R is produced.

In [3]:
```python
import matplotlib.pyplot as plt

# Creating the x and y values for the plot
R_range = np.linspace(-2.5,2.5,100) # Setting the range for the nuclear co-ordinates R
E_values = sbh_energy(R_range) # Creating energy values for R

plt.plot(R_range, E_values, color='red') # Plots Nuclear co-ords vs Energy with cosmetic customisation
plt.xlabel('Nuclear Co-ordinate (R)') # x axis label
plt.ylabel('Energy $(E_i)$') # y axis label
plt.title('Energy $(E_i)$ vs Nuclear Geometry (R)') # Graph title
plt.show()
```



2.2: Using the function defined in *1.2* a plot of Energy Gradient dE_i vs Geometry R is produced.

In [4]:
```
# Creating the x and y values for the plot
R_range = np.linspace(-2.5,2.5,100) # Setting the range for the nuclear co-
ordinates R
E_gradient = sbh_dE_dR(R_range) # Creating energy gradient values for R

plt.plot(R_range, E_gradient, color = 'green') # Plots Nuclear co-ords vs E
nergy gradient with cosmetic customisation
plt.xlabel('Nuclear Co-ordinate (R)') # x axis label - single co-ord due to
1D
plt.ylabel('Energy Gradient $(dE_i)$') # y axis label
plt.title('Energy Gradient $(dE_i)$ vs Nuclear Geometry (R)') # Graph title
plt.show()
```



2.3: Playing with the parameters.

Increasing the following $\boldsymbol{M_1}$, $\boldsymbol{\varepsilon_0}$, $\boldsymbol{v_0}$, $\boldsymbol{\omega_1}$, $\boldsymbol{g_1}$: causes the energy barries inbetween the two potential wells to decrease, essentially the wells will join for a high enough value of M_1. This decrease in the energy barrier causes the dE_i vs R plot to tend to linearity.

ε_0 is a function which contains and is proportional to M_1, v_0, ω_1, g_1. Each parameter exhibits a similar trend when being increased or decreased due to the energy differece between the two quantum states being equal to $2\varepsilon_0$. Here, the two quantum states displays the spins of the particle, where a change in the value of any of these parameters will cause one quantum spin state to be favoured relative to the other. As a result, this alters the probablilty distrubution and energy differences between these states.

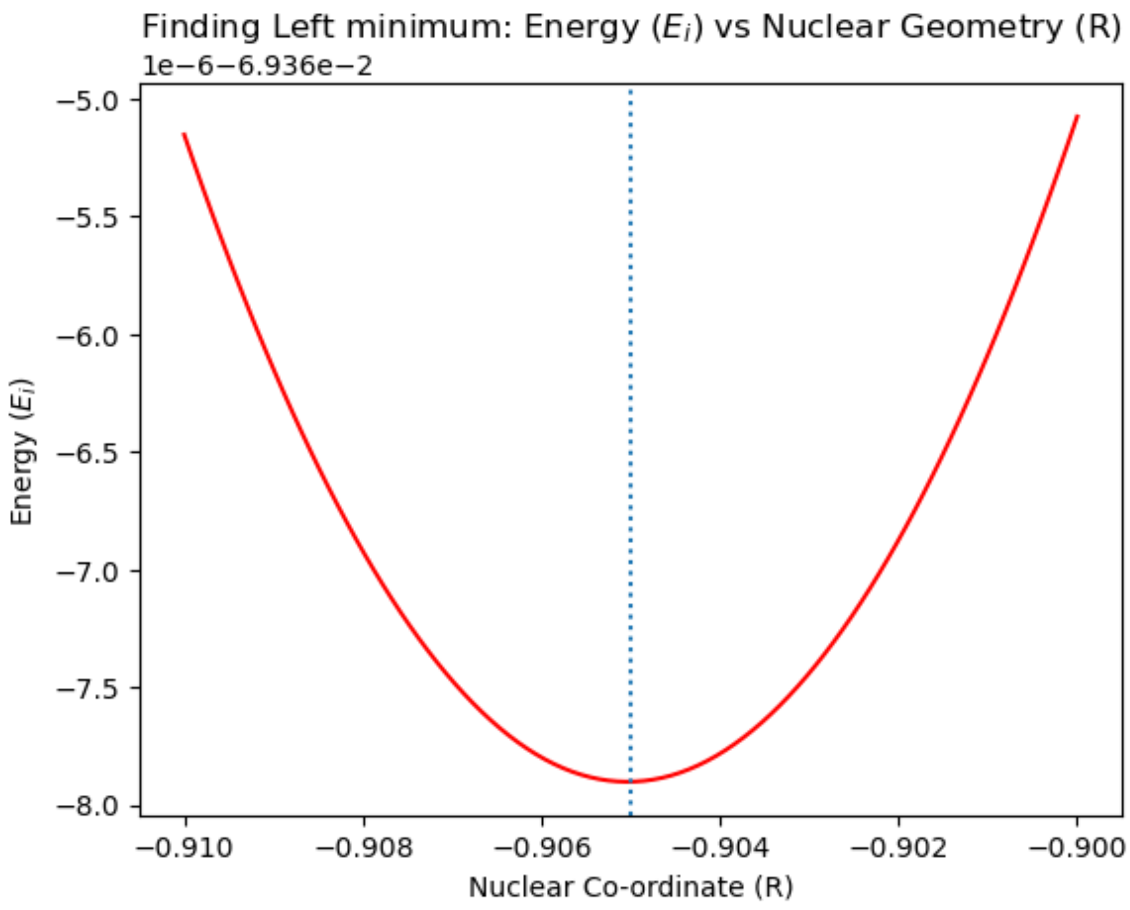
Task 3:

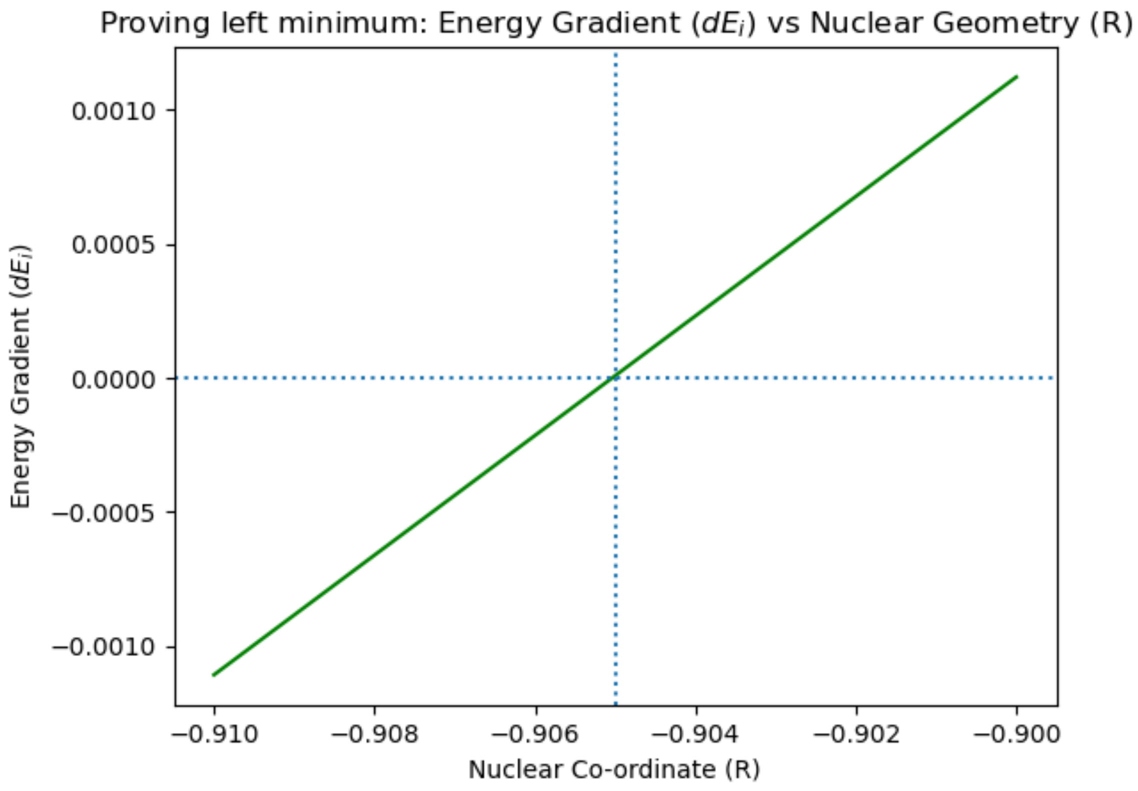
Determine approximately (R_1, R_2) for the two minima of the plot in 2.1.

Here the linspace is visually reduced for each minima and then approximated.

3.1 Approximating the left minimum

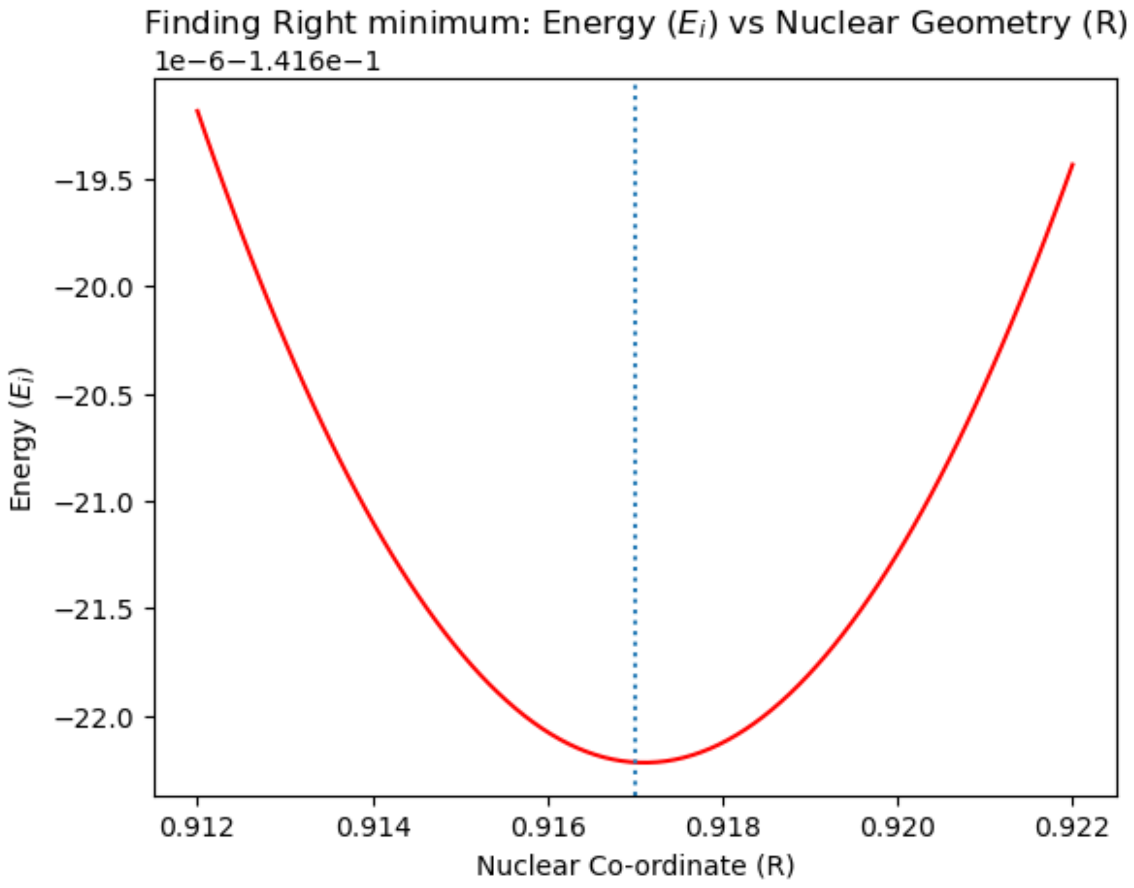
In [5]:
```
# Creating the x and y values for the plot
R_range = np.linspace(-0.9,-0.91,100) # Reducing Linspace to find the left
minimum
E_values = sbh_energy(R_range) # Creating energy values for R
E_gradient = sbh_dE_dR(R_range) # Creating energy gradient values for R

plt.plot(R_range, E_values, color='red') # Plots Nuclear co-ords vs Energy
with cosmetic customisation
plt.xlabel('Nuclear Co-ordinate (R)') # x axis label
plt.ylabel('Energy $(E_i)$') # y axis label
plt.title('Finding Left minimum: Energy $(E_i)$ vs Nuclear Geometry (R)') #
Graph title
plt.axvline(x=-0.905, linestyle = 'dotted') # Used to clearly approximately
display minimum
plt.show()

plt.plot(R_range, E_gradient, color = 'green') # Plots Nuclear co-ords vs E
nergy gradient with cosmetic customisation
plt.xlabel('Nuclear Co-ordinate (R)') # x axis label - single co-ord due to
1D
plt.ylabel('Energy Gradient $(dE_i)$') # y axis label
plt.title('Proving left minimum: Energy Gradient $(dE_i)$ vs Nuclear Geomet
ry (R)') # Graph title
plt.axhline(y=0, linestyle="dotted", label="R_1 left min")
plt.axvline(x=-0.905, linestyle = 'dotted') # Used to clearly approximately
display minimum
plt.show()
```

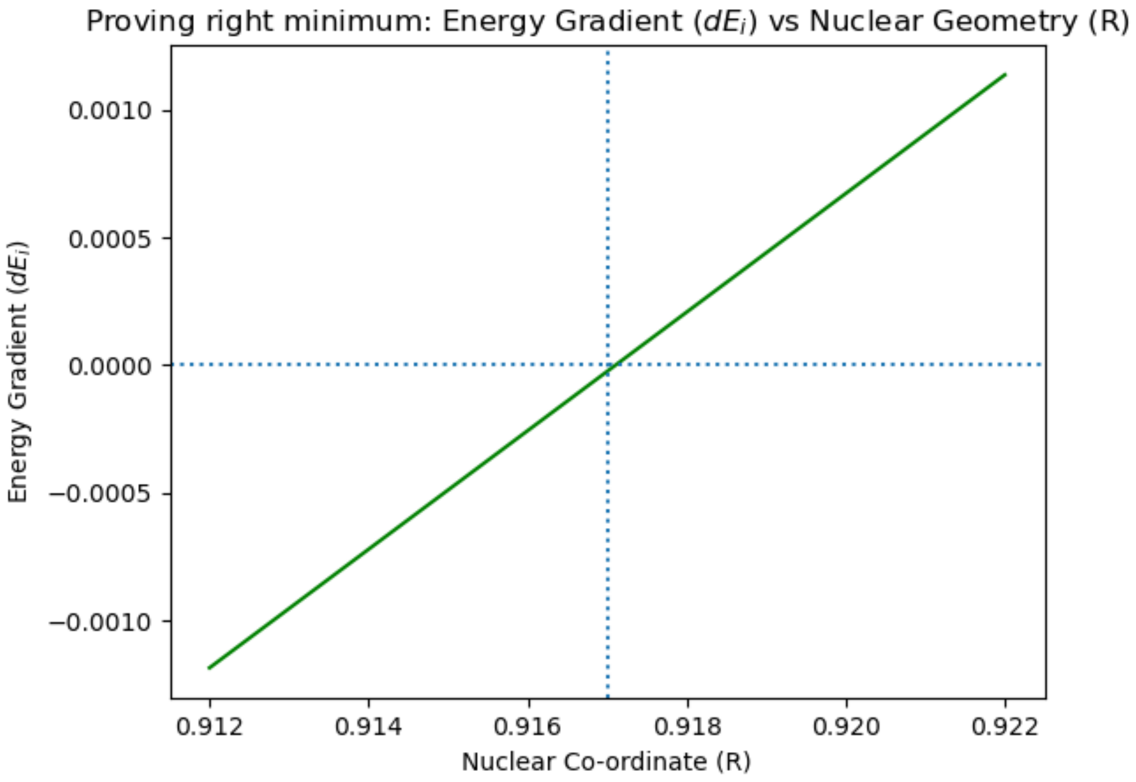

Finding Left minimum: Energy (E_i) vs Nuclear Geometry (R)
1e−6−6.936e−2
−5.0
−5.5
−6.0
−6.5
−7.0
−7.5
−8.0
Energy (E_i)
−0.910
−0.908
−0.906
−0.904
−0.902
−0.900
Nuclear Co-ordinate (R)


Proving left minimum: Energy Gradient (dE_i) vs Nuclear Geometry (R)
0.0010
0.0005
0.0000
−0.0005
−0.0010
Energy Gradient (dE_i)
−0.910
−0.908
−0.906
−0.904
−0.902
−0.900
Nuclear Co-ordinate (R)

3.2 Approximating the right minimum

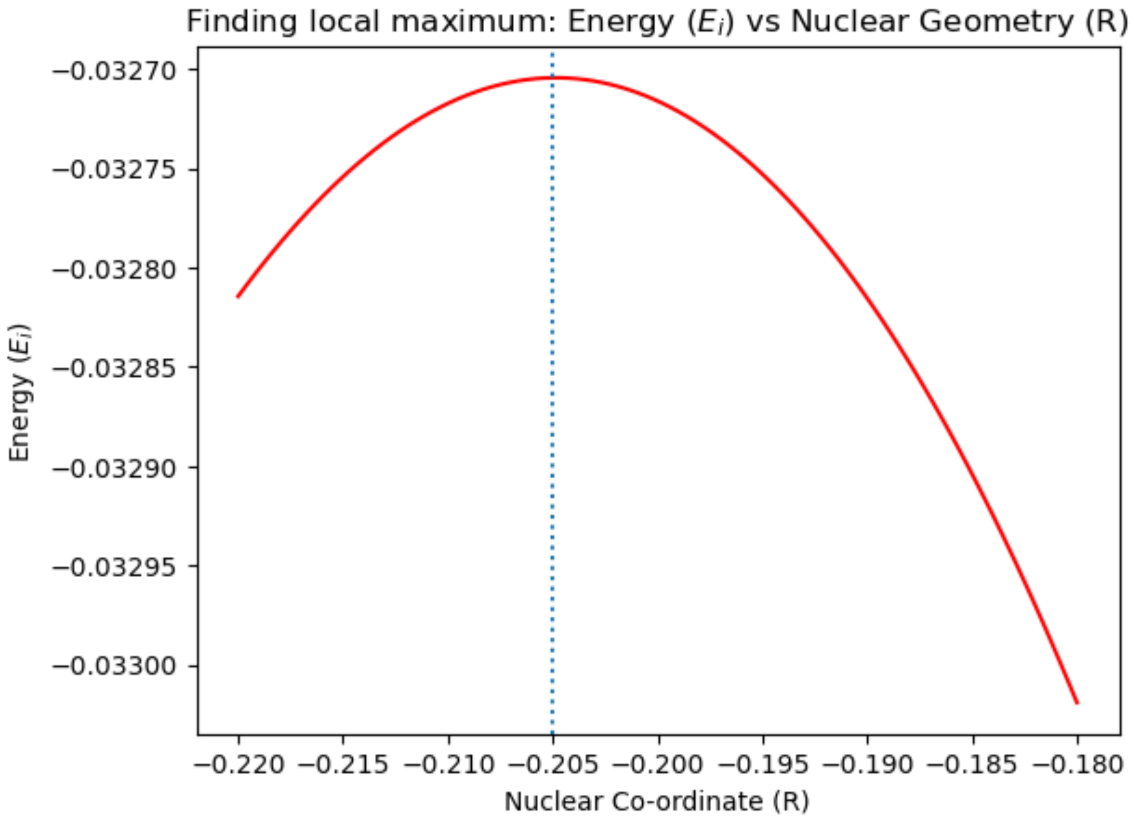
In [6]:
```
# Creating the x and y values for the plot
R_range = np.linspace(0.912,0.922,100) # Reducing Linspace to find the righ
t minimum
E_values = sbh_energy(R_range) # Creating energy values for R
E_gradient = sbh_dE_dR(R_range) # Creating energy gradient values for R

plt.plot(R_range, E_values, color='red') # Plots Nuclear co-ords vs Energy
with cosmetic customisation
plt.xlabel('Nuclear Co-ordinate (R)') # x axis label
plt.ylabel('Energy $(E_i)$') # y axis label
plt.title('Finding Right minimum: Energy $(E_i)$ vs Nuclear Geometry (R)')
# Graph title
plt.axvline(x=0.917, linestyle = 'dotted') # Used to approximately display
right minimum
plt.show()

plt.plot(R_range, E_gradient, color = 'green') # Plots Nuclear co-ords vs E
nergy gradient with cosmetic customisation
plt.xlabel('Nuclear Co-ordinate (R)') # x axis label - single co-ord due to
1D
plt.ylabel('Energy Gradient $(dE_i)$') # y axis label
plt.title('Proving right minimum: Energy Gradient $(dE_i)$ vs Nuclear Geome
try (R)') # Graph title
plt.axhline(y=0, linestyle="dotted")
plt.axvline(x=0.917, linestyle = 'dotted') # Used to clearly approximately
display minimum
plt.show()
```

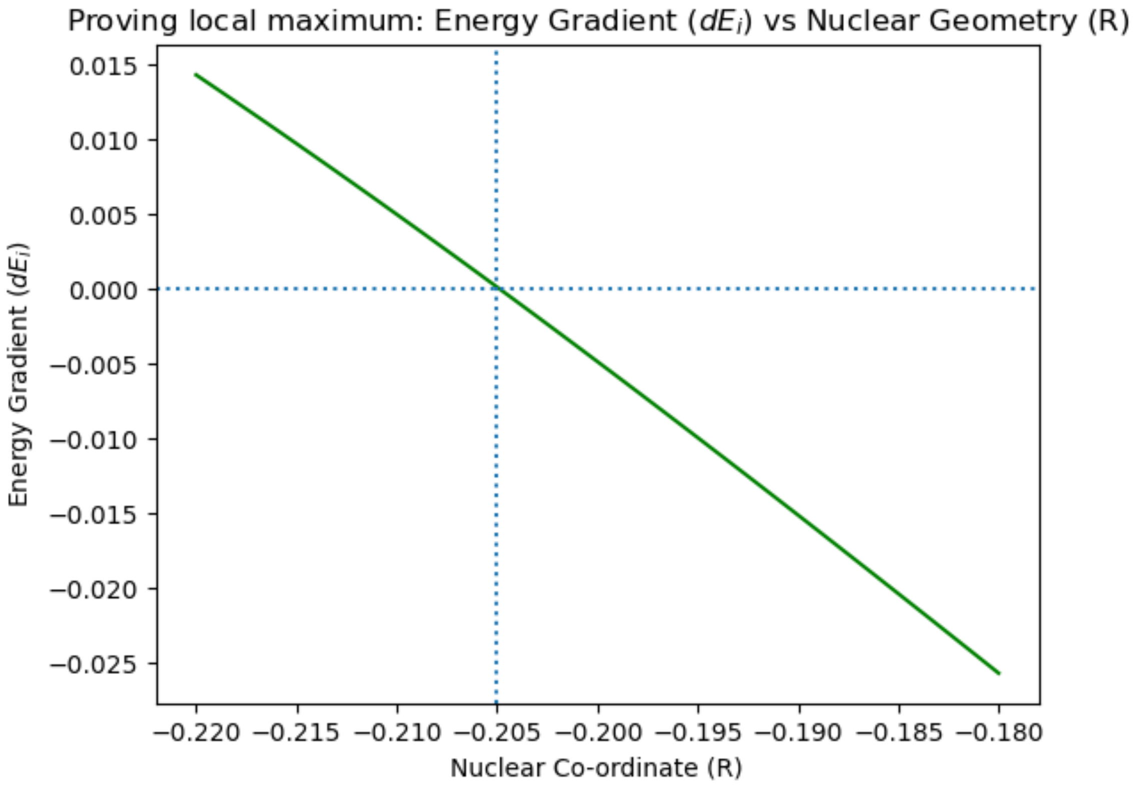

Finding Right minimum: Energy (E_i) vs Nuclear Geometry (R)
1e−6−1.416e−1
−19.5
−20.0
−20.5
−21.0
−21.5
−22.0
Energy (E_i)
0.912
0.914
0.916
0.918
0.920
0.922
Nuclear Co-ordinate (R)


Proving right minimum: Energy Gradient (dE_i) vs Nuclear Geometry (R)
0.0010
0.0005
0.0000
−0.0005
−0.0010
Energy Gradient (dE_i)
0.912
0.914
0.916
0.918
0.920
0.922
Nuclear Co-ordinate (R)

3.3 Approximating transition energy barrier - local maximum

In [7]:
```
# Creating the x and y values for the plot
R_range = np.linspace(-0.18,-0.22,100) # Reducing Linspace to find the loca
l maximum
E_values = sbh_energy(R_range) # Creating energy values for R
E_gradient = sbh_dE_dR(R_range) # Creating energy gradient values for R

plt.plot(R_range, E_values, color='red') # Plots Nuclear co-ords vs Energy
with cosmetic customisation
plt.xlabel('Nuclear Co-ordinate (R)') # x axis label
plt.ylabel('Energy $(E_i)$') # y axis label
plt.title('Finding local maximum: Energy $(E_i)$ vs Nuclear Geometry (R)')
# Graph title
plt.axvline(x=-0.205, linestyle = 'dotted') # Used to approximately display
local maximum
plt.show()

plt.plot(R_range, E_gradient, color = 'green') # Plots Nuclear co-ords vs E
nergy gradient with cosmetic customisation
plt.xlabel('Nuclear Co-ordinate (R)') # x axis label - single co-ord due to
1D
plt.ylabel('Energy Gradient $(dE_i)$') # y axis label
plt.title('Proving local maximum: Energy Gradient $(dE_i)$ vs Nuclear Geome
try (R)') # Graph title
plt.axhline(y=0, linestyle="dotted")
plt.axvline(x=-0.205, linestyle = 'dotted') # Used to clearly approximately
display maximum
plt.show()
```


Finding local maximum: Energy (E_i) vs Nuclear Geometry (R)
Energy (E_i)
−0.03270
−0.03275
−0.03280
−0.03285
−0.03290
−0.03295
−0.03300
−0.220 −0.215 −0.210 −0.205 −0.200 −0.195 −0.190 −0.185 −0.180
Nuclear Co-ordinate (R)


Proving local maximum: Energy Gradient (dE_i) vs Nuclear Geometry (R)
Energy Gradient (dE_i)
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
−0.020
−0.025
−0.220 −0.215 −0.210 −0.205 −0.200 −0.195 −0.190 −0.185 −0.180
Nuclear Co-ordinate (R)

From the approximatations above we can estimate the value of each minimum and their respective SBH energies (R_1, R_2).

$R_1: \ -0.905 \ a.u$ $\qquad$ $E_1: \ -0.0694 \ Hartree$

$R_2: \ 0.917 \ a.u$ $\qquad$ $E_2: \ -0.142 \ Hartree$

The local maximum can be found to be at R_3, with a respective energy of E_3,

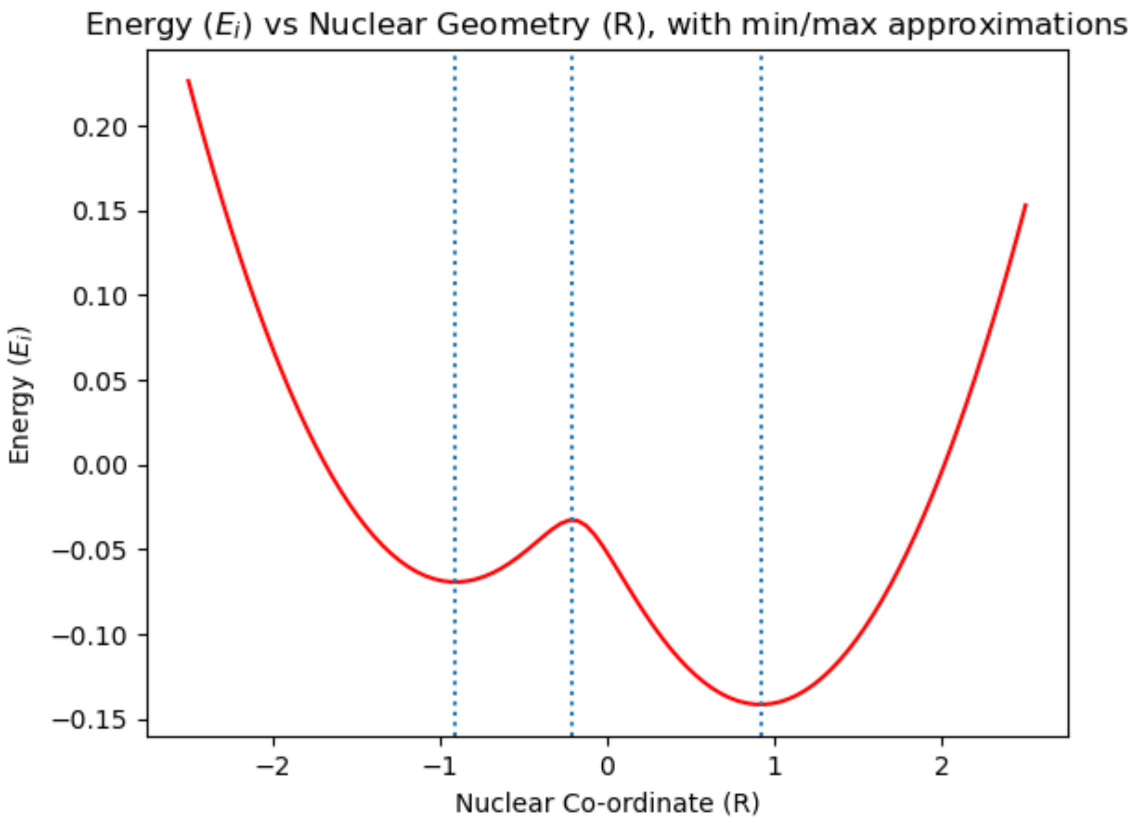
$R_3: \ -0.205 \ a.u$ $\qquad$ $E_3: \ -0.0327 \ Hartree$

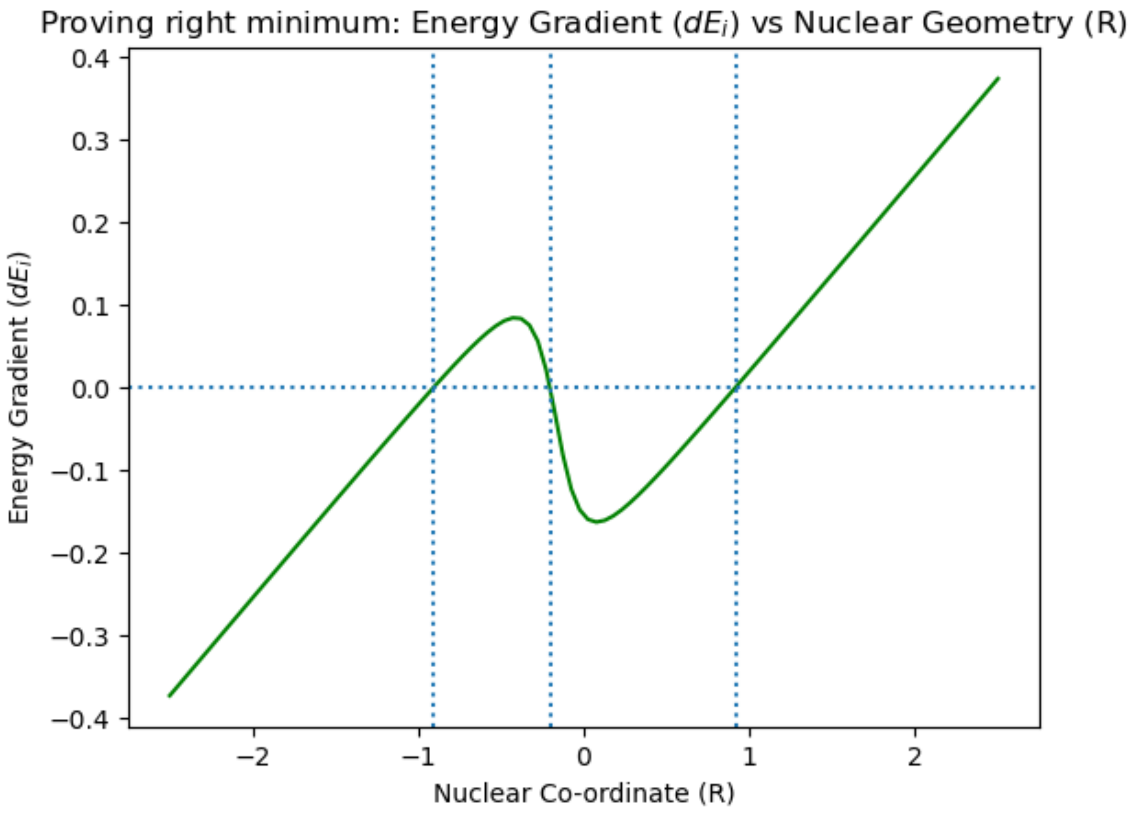
Using these values we can approximate the minimum energy needed to hop between minimums,

$1)$ Hop from R_1 to R_2: $- - \ > (E_3 - E_1) = 0.0367 \ Hartree$

$2)$ Hop from R_2 to R_1: $- - \ > (E_3 - E_2) = 0.1093 \ Hartree$

In [8]:
```
# Creating the x and y values for the plot
R_range = np.linspace(-2.5,2.5,100) # Reducing Linspace to find the right m
inimum
E_values = sbh_energy(R_range) # Creating energy values for R
E_gradient = sbh_dE_dR(R_range) # Creating energy gradient values for R

plt.plot(R_range, E_values, color='red') # Plots Nuclear co-ords vs Energy
with cosmetic customisation
plt.xlabel('Nuclear Co-ordinate (R)') # x axis label
plt.ylabel('Energy $(E_i)$') # y axis label
plt.title('Energy $(E_i)$ vs Nuclear Geometry (R), with min/max approximati
ons') # Graph title
plt.axvline(x=0.917, linestyle = 'dotted') # Used to approximately display
right minimum
plt.axvline(x=-0.905, linestyle = 'dotted') # Used to approximately display
left minimum
plt.axvline(x=-0.205, linestyle = 'dotted') # Used to clearly approximately
display maximum
plt.show()

plt.plot(R_range, E_gradient, color = 'green') # Plots Nuclear co-ords vs E
nergy gradient with cosmetic customisation
plt.xlabel('Nuclear Co-ordinate (R)') # x axis label - single co-ord due to
1D
plt.ylabel('Energy Gradient $(dE_i)$') # y axis label
plt.title('Proving right minimum: Energy Gradient $(dE_i)$ vs Nuclear Geome
try (R)') # Graph title
plt.axvline(x=0.917, linestyle = 'dotted') # Used to clearly approximately
display right minimum
plt.axvline(x=-0.905, linestyle = 'dotted') # Used to approximately display
left minimum
plt.axvline(x=-0.205, linestyle = 'dotted') # Used to clearly approximately
display maximum
plt.axhline(y=0, linestyle="dotted")
plt.show()
```


Energy (E_i) vs Nuclear Geometry (R), with min/max approximations
Energy (E_i)
Nuclear Co-ordinate (R)
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−2
−1
0
1
2


Proving right minimum: Energy Gradient (dE_i) vs Nuclear Geometry (R)
Energy Gradient (dE_i)
Nuclear Co-ordinate (R)
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
−2
−1
0
1
2

TP4 : Velocity Verlet algorithm - Molecular Dymanics Laboratory

The velocity Verlet algorithm is a numerical integration method commonly used in molecular dynamics simulations. It provides an efficient and accurate way to update the positions and velocities of particles under the influence of forces derived from potential energy (SBH energy). The algorithm ensures good energy conservation over time. Here a 1D velcoty Verlet algorithm is implemented for a particle in a PES represented by the Spin-Boson Hamiltonian as produced above in TP3. Velocity Verlet gives us the following equations:

Formulas

1. **Position update**:
   Position is updated using the previous position , velocity , and acceleration:

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2 \qquad eq. [5]$$

2. **Velocity update**:
   Velocity is updated using the previous, velocity , acceleration , and current acceleration:

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \frac{1}{2}(\mathbf{a}(t) + \mathbf{a}(t+\Delta t)\Delta t \qquad eq. [6]$$

3. **Intial and updated acceleration**:
   Evaluation of acceleration with mass and energy gradient from the SBH model of the previous lab (TP3):

$$\mathbf{a} = \frac{-1}{\mathbf{M}}\frac{\mathbf{dE}(\mathbf{x}(t))}{\mathbf{dx}} \qquad eq. [7]$$

   Where the updated acceleration is the following:

$$\mathbf{a}(t+\Delta t) = \frac{-1}{\mathbf{M}}\frac{\mathbf{dE}(\mathbf{x}(t+\Delta t))}{\mathbf{dx}} \qquad eq. [8]$$

   where $\mathbf{E}(t)$ is the energy derived from the SBH energy
4. **Total Energy**:
   The total energy at time t:

$$\mathbf{E}_t(t) = \mathbf{E}_k(t) + \mathbf{E}(t) \qquad eq. [9]$$

   Where the kinetic energy is the following:

$$\mathbf{E}_k(t) = \frac{1}{2}\mathbf{M}\mathbf{v}(t)^2 \qquad eq. [10]$$

Task 4:

Implement the above 1D velocity verlet algorithm for a single co-ordinate with any external forces (no force term in acceleration), where the total energy is conversed if it follows this condition:

$$\left|E_T(t) - E_T(t - \Delta t)\right| \leq \delta\varepsilon \qquad eq. [11]$$

Where $\delta\varepsilon = 0.004$

```
In [9]: def verlet_vel(x, v, dt, n, eth):
            # Initialise arrays, better than lists as it is more memory efficient.
            xs, vs, ac, t, ks, ets, us = (np.zeros(n) for _ in range(7))

            # Set initial conditions
            xs[0], vs[0], t[0] = x, v, 0 # Initial position, velocity and time
            ac[0] = - (sbh_dE_dR(xs[0])) * (1/m) # Using SBH gradient from TP3 as E
        _i to calculate acceleration
            ks[0] = 0.5 * m * vs[0]**2 # Initial kinetic Energy
            us[0] = sbh_energy(xs[0]) # Using SBH energy from TP3 for potential ene
        rgy
            ets[0] = ks[0] + us[0] # Initial total Energy

            for i in range(1, n):
                # Update position, acceleration, and velocity
                xs[i] = xs[i-1] + vs[i-1] * dt + 0.5 * ac[i-1] * dt**2
                ac[i] = - (sbh_dE_dR(xs[i])) * (1/m)
                vs[i] = vs[i-1] + 0.5 * (ac[i-1] + ac[i]) * dt

                # Update time and energies
                t[i] = t[i-1] + dt
                ks[i] = 0.5 * m * vs[i]**2
                us[i] = sbh_energy(xs[i])
                ets[i] = ks[i] + us[i]

                # Check energy threshold, where eth = 0.004 for energy to be conser
        ved
                if abs(ets[i] - ets[i-1]) > eth:
                    break

            return xs, vs, t, ets, ks, ac, us
```

Task 5:

Run a 1-Picosecond $(41341.37\ a.u.)$ simulation based on the parameters given in Table 1 (TP3), starting with R_1. The dyanmics is run with a $dt = 20$ starting at the position of the left well, $R_1 = -0.905$. If the system is at a minimum at R_1 and starting $\mathbf{E}_t(t)$ of 0.1 Hartree, the initial velocity becomes the following:

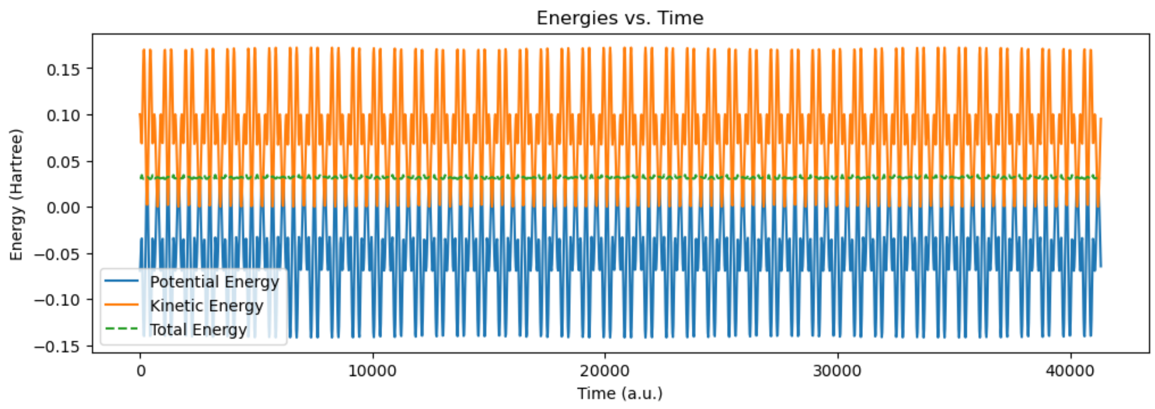
$$\mathbf{E}_t(t) = \mathbf{E}_k(t) + 0$$

$$\mathbf{E}_t(t) = \frac{1}{2}\mathbf{Mv0}(t)^2$$

$$\mathbf{v_0} = \sqrt{\frac{2\mathbf{E}_t(t)}{M}} \qquad eq. [12]$$

From the plot of the all energies vs time, we can see that the total energy is conserved, it is a relatively straight line.

In [10]:
```
# Initial conditions
x0 = -0.905 # R_1, left minimum
Etot = 0.1 # Total energy in Hartree, given
v0 = (2*(Etot)/m)**0.5 # As all kinetic energy equals E_total at R_1
dt = 20 # Timestep
n = int(41341.374575751/dt) # Steps for one picosecond in a.u.
eth = 0.004  # Energy threshold for conservation

# Running the dynamics
xs, vs, t, ets, ks, ac, us = verlet_vel(x0, v0, dt, n, eth)

# Plotting results
# Plotting energies as a function of time
plt.figure(figsize=(12, 8))
plt.subplot(2, 1, 2)
plt.plot(t, us, label="Potential Energy")
plt.plot(t, ks, label="Kinetic Energy")
plt.plot(t, ets, label="Total Energy", linestyle="--")
plt.legend()
plt.xlabel("Time (a.u.)")
plt.ylabel("Energy (Hartree)")
plt.title("Energies vs. Time")


plt.show()
```



Task 6:

Run the simulation with different total energy values where $\mathbf{E}_t(t) = \mathbf{E}_k(t) + 0$ in the R_2 starting position, the minimum energy is calculated from the plot of position vs time. Here we can use the minimum energy required to jump from R_2 to R_1 apporximated in TP3, which can also be estimated from the Verlet Velocity algorithm as shown below:

$$R_2 \ to \ R_1 : - - > (E_3 - E_2) = 0.1093 \ Hartree$$

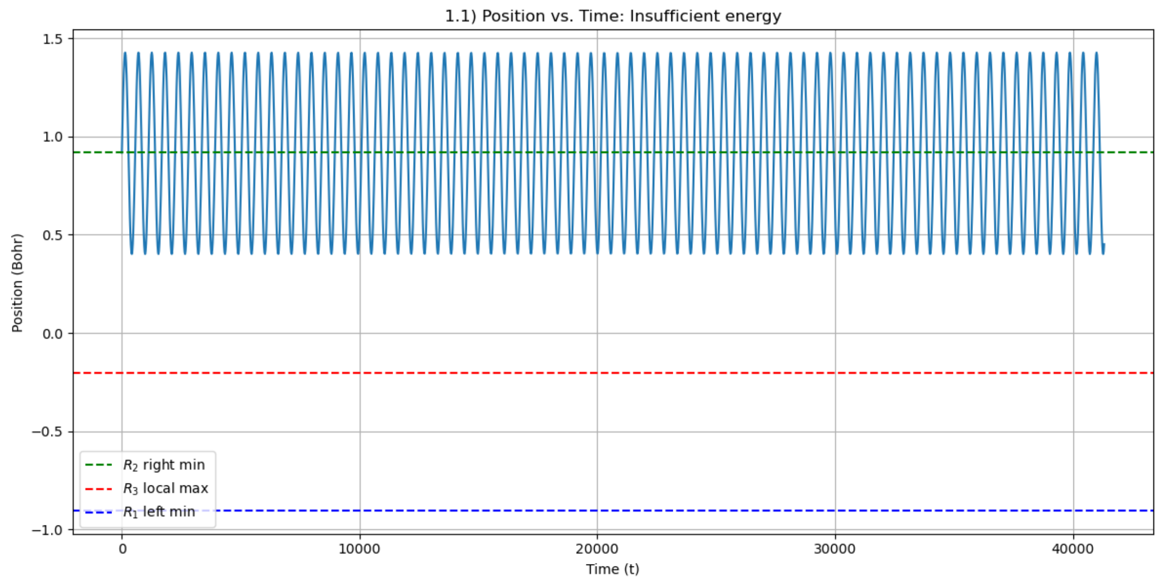
Below you can see results for the estimated energy requried to hop from R_2 to R_1, where three plots are displayed:

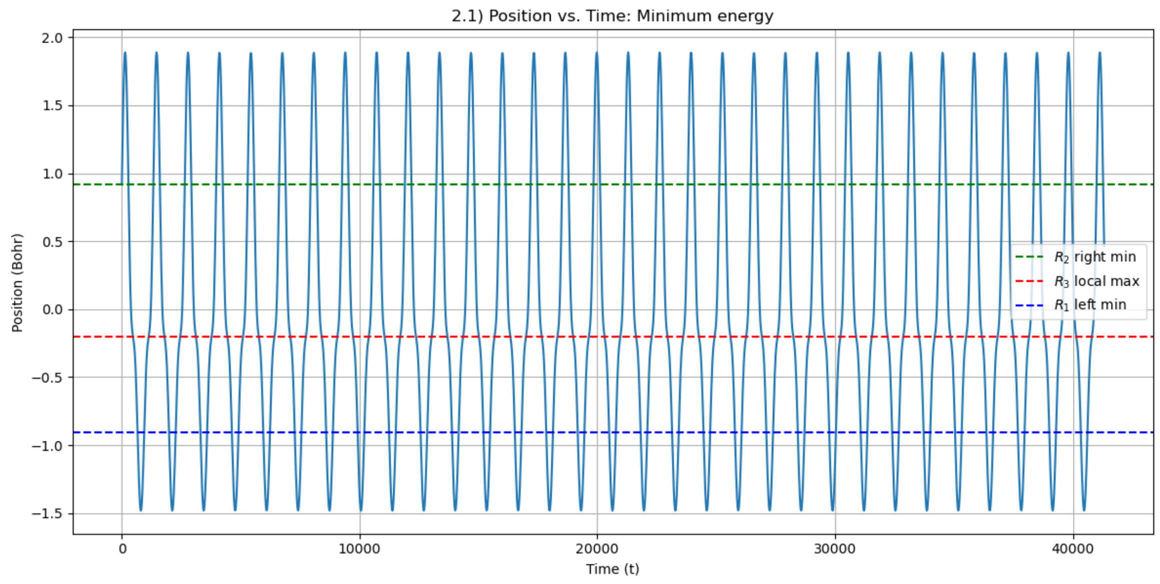
$1.1)$ Not sufficient energy for the particle to move from R_2 to R_1, as shown the particle is stuck in R_2

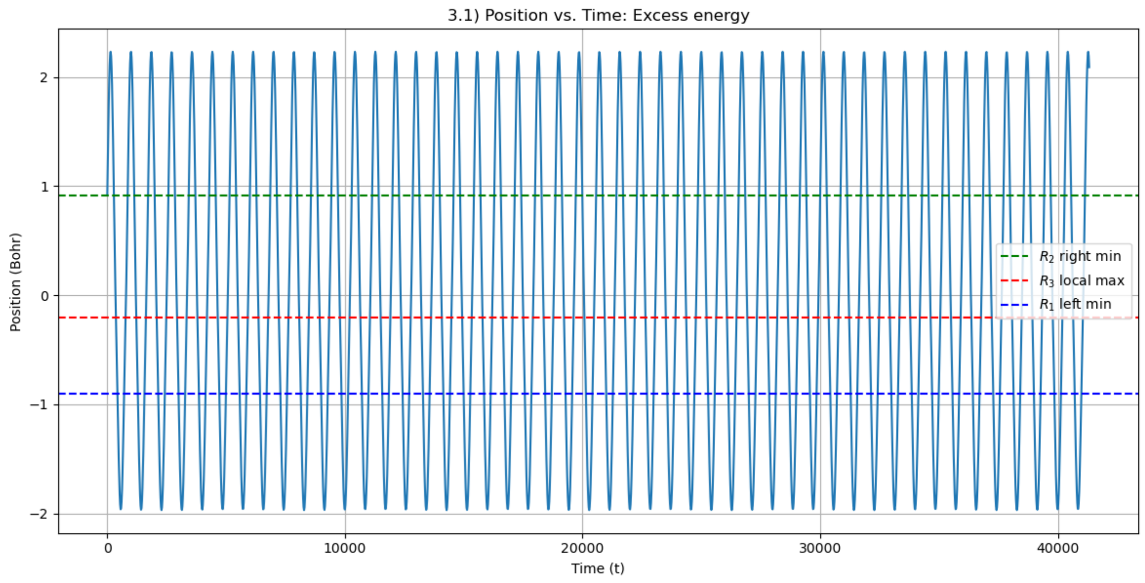
$2.1)$ Estimated minimum energy required for the particle to move from R_2 to R_1, as shown there is a decrease in velocity at R_3

$3.1)$ Excess energy for the particle to move from R_2 to R_1, as shown to be a smooth curve (transition between wells) throughout

In [11]:
```
# Results from TP3 for positions of minimums and maximum
R_1 = -0.905 # Left min
R_2 = 0.917 # Right min
R_3 = -0.205 # Local max

# Varying energy to find energy required to hop from right to left well

# 1) Not sufficient energy for the particle to move from R_2 to R_1
Etot_1 = 0.03 # Not suffcient total energy to move
v0_1 = (2*(Etot_1)/m)**0.5 # As all kinetic energy equals E_total at R_1

# 2) Estimated minimum energy required for the particle to move from R_2 to
R_1
Etot_2 = 0.1093 # Estimated min energy from TP3
v0_2 = (2*(Etot_2)/m)**0.5

# 3) Excess energy for the particle to move from R_2 to R_1
Etot_3 = 0.2 # Excess energy to move
v0_3 = (2*(Etot_3)/m)**0.5


# Running the dyanmics for the different values of total energies, where al
l kinetic enregy is in R_2
v0_values = [v0_1, v0_2, v0_3] # Loop through each v0 and call the function
results = [] # Store results
titles = ["1.1) Position vs. Time: Insufficient energy", "2.1) Position vs.
Time: Minimum energy", "3.1) Position vs. Time: Excess energy"] # Create an
array of titles to be used for each plot

# Run the simulation for each v0
for v0 in v0_values:
    xs, vs, t, ets, ks, ac, us = verlet_vel(R_2, v0, dt, n, eth)
    results.append((xs, t))

# Plot results
plt.figure(figsize=(12, 18))

for i, (xs, t) in enumerate(results): # for loop for seperate plots for eac
h v0
    plt.subplot(len(v0_values), 1, i + 1)  # Create a subplot for each v0
    plt.plot(t, xs)
    plt.xlabel('Time (t)')
    plt.ylabel('Position (Bohr)')
    plt.axhline(y=R_2, color="g", linestyle="--", label="$ R_2 $ right mi
n")
    plt.axhline(y=R_3, color="r", linestyle="--", label="$ R_3 $ local ma
x")
    plt.axhline(y=R_1, color="b", linestyle="--", label="$ R_1 $ left min")
    plt.legend()
    plt.title(titles[i])
    plt.grid()

plt.tight_layout() # Adjusting spacing, makes the layout more visually appe
aling
plt.show()
```


1.1) Position vs. Time: Insufficient energy
Position (Bohr)
Time (t)
R_2 right min
R_3 local max
R_1 left min
1.5
1.0
0.5
0.0
−0.5
−1.0
0
10000
20000
30000
40000


2.1) Position vs. Time: Minimum energy
Position (Bohr)
Time (t)
R_2 right min
R_3 local max
R_1 left min
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
10000
20000
30000
40000


3.1) Position vs. Time: Excess energy
Position (Bohr)
Time (t)
R_2 right min
R_3 local max
R_1 left min
2
1
0
−1
−2
0
10000
20000
30000
40000

Now the opposite, starting at R_1, the minimum energy is calculated from the plot of position vs time. Here we can use the minimum energy required to jump from R_1 to R_2 apporximated in TP3, which can also be estimated from the Verlet Velocity algorithm as shown below:

$$R_1 \ to \ R_2 : --> (E_3 - E_1) = 0.0367 \ Hartree$$

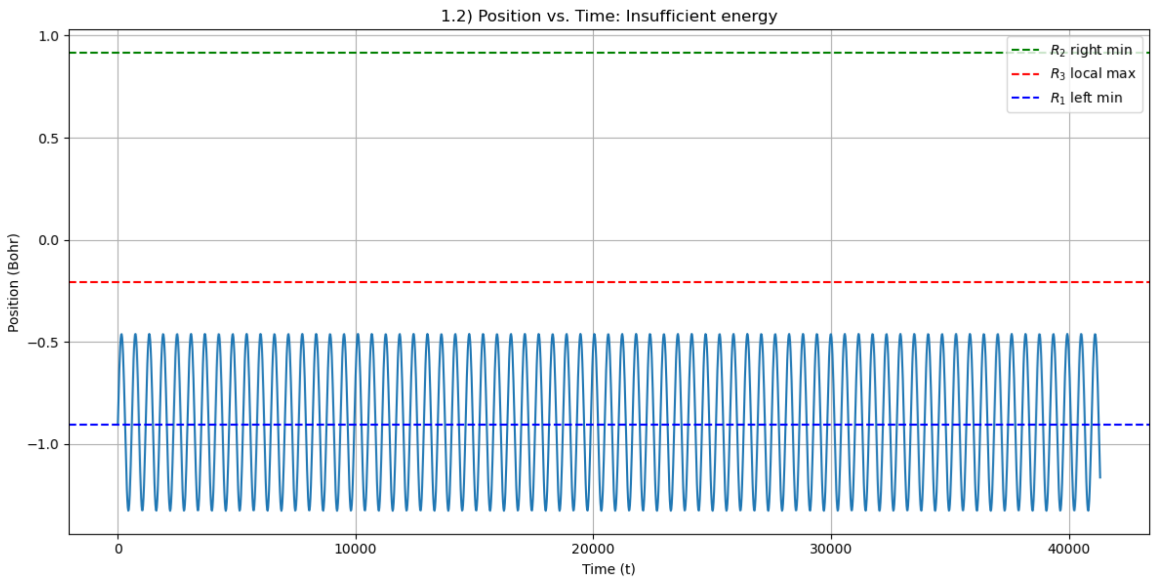
You can see the results for the estimated energy requried to hop from now R_1 to R_2, where three plots are displayed:

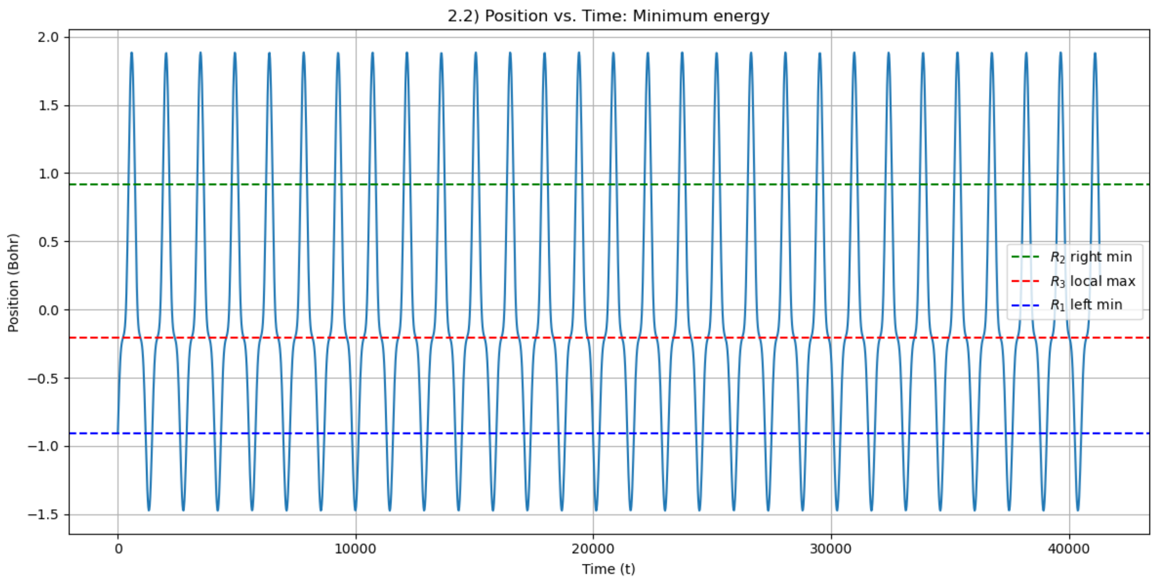
$1.2)$ Not sufficient energy for the particle to move from R_1 to R_2, as shown the particle is stuck in R_1

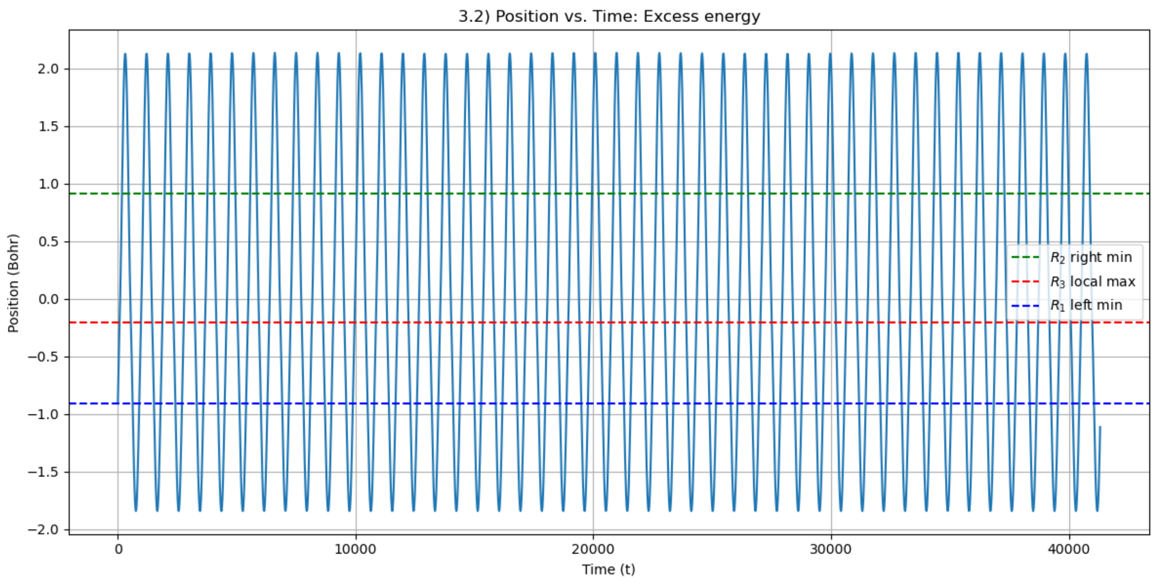
$2.2)$ Estimated minimum energy required for the particle to move from R_1 to R_1, as shown there is a decrease in velocity at R_3

$3.2)$ Excess energy for the particle to move from R_1 to R_2, as shown to be a smooth curve (transition between minima) throughout

In [12]:
```
# Varying energy to find energy required to hop from left to right well

# 1) Not sufficient energy for the particle to move from R_1 to R_2
Etot_1 = 0.02 # Not suffcient total energy to move
v0_1 = (2*(Etot_1)/m)**0.5 # As all kinetic energy equals E_total at R_1

# 2) Estimated minimum energy required for the particle to move from R_1 to
R_2
Etot_2 = 0.0367 # Estimated min energy from TP3
v0_2 = (2*(Etot_2)/m)**0.5

# 3) Excess energy for the particle to move from R_1 to R_2
Etot_3 = 0.1 # Excess energy to move
v0_3 = (2*(Etot_3)/m)**0.5


# Running the dyanmics for the different values of total energies, where al
l kinetic enregy is in R_1
v0_values = [v0_1, v0_2, v0_3] # Loop through each v0 and call the function
results = [] # Store results
titles = ["1.2) Position vs. Time: Insufficient energy", "2.2) Position vs.
Time: Minimum energy", "3.2) Position vs. Time: Excess energy"] # Create an
array of titles to be used for each plot

# Run the simulation for each v0
for v0 in v0_values:
    xs, vs, t, ets, ks, ac, us = verlet_vel(R_1, v0, dt, n, eth)
    results.append((xs, t))

# Plot results
plt.figure(figsize=(12, 18))

for i, (xs, t) in enumerate(results): # for loop for seperate plots for eac
h v0
    plt.subplot(len(v0_values), 1, i + 1)  # Create a subplot for each v0
    plt.plot(t, xs)
    plt.xlabel('Time (t)')
    plt.ylabel('Position (Bohr)')
    plt.axhline(y=R_2, color="g", linestyle="--", label="$ R_2 $ right mi
n")
    plt.axhline(y=R_3, color="r", linestyle="--", label="$ R_3 $ local ma
x")
    plt.axhline(y=R_1, color="b", linestyle="--", label="$ R_1 $ left min")
    plt.legend()
    plt.title(titles[i])
    plt.grid()

plt.tight_layout() # Adjusting spacing, makes the layout more visually appe
aling
plt.show()
```


1.2) Position vs. Time: Insufficient energy
R_2 right min
R_3 local max
R_1 left min
Position (Bohr)
Time (t)
1.0
0.5
0.0
−0.5
−1.0
0
10000
20000
30000
40000


2.2) Position vs. Time: Minimum energy
R_2 right min
R_3 local max
R_1 left min
Position (Bohr)
Time (t)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
10000
20000
30000
40000


3.2) Position vs. Time: Excess energy
R_2 right min
R_3 local max
R_1 left min
Position (Bohr)
Time (t)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0
10000
20000
30000
40000

TP5 : Thermostat - Molecular Dymanics Laboratory

Maintaining a constant temperature throughout a simulation is essential for modeling the canonical ensemble. This requires specific algorithms known as thermostats. These algorithms are necessary for ensuring molecular dynamics (MD) simulations are both accurate and reliable by carefully regulating the system's temperature.

The Andersen thermostat takes a stochastic approach, meaning it introduces randomness into the system's dynamics. This approach mimics the random collisions of particles in a thermal environment, effectively adjusting the system's temperature. By simulating these random interactions, the Andersen thermostat provides a robust way to maintain thermodynamic conditions while accounting for natural fluctuations. Here's how the Andersen thermostat algorithm works:

1. Obtain a definition for the collision frequency, Γ
2. Use the Velocity Verlet algorithm from TP4 to integrate dynamics in a single step Δt, removing the energy conservation term
3. For each step, obtain a random number from a Gaussian distribution of unit variance r_0, where:

If $r_0 > \Gamma\Delta t$, velocity does not change
If $r_0 \leq \Gamma\Delta t$, velocity becomes the following:

$$\mathbf{v}(t + \Delta t) = r_x\sqrt{\frac{k_B T}{M}} \qquad eq. [13]$$

where r_x is another random number from the same distribution

Task 7:

Implementing the Andersen thermostat. By using the velocity verlet function defined in TP4 the above thermostat algorithm can be implmented as shown below. Due to the model now being one of a canonical ensemble, energy is no longer conserved hence, the energy condition must be removed.

In [13]:
```
# Must reproduce the velocity verlet function defined in TP4 removing the e
nergy conservation condition and introudcing the Andersen Thermostat

def verlet_velTP5(x, v, dt, n, gamma, T_target):
    xs, vs, ac, t, ks, ets, us, r = (np.zeros(n) for _ in range(8))

    # Set initial conditions
    xs[0], vs[0], t[0], r[0] = x, v, 0, np.random.normal(0, 1) # Initial po
sition, velocity, time and r_0
    ac[0] = - (sbh_dE_dR(xs[0])) * (1/m) # Using SBH gradient from TP3 as E
_i to calculate acceleration
    ks[0] = 0.5 * m * vs[0]**2  # Initial kinetic Energy
    us[0] = sbh_energy(xs[0]) # Using SBH energy from TP3 for potential ene
rgy
    ets[0] = ks[0] + us[0] # Initial total Energy

    for i in range(1, n):
        # Update position, acceleration, and velocity
        xs[i] = xs[i-1] + vs[i-1] * dt + 0.5 * ac[i-1] * dt**2
        ac[i] = - (sbh_dE_dR(xs[i])) * (1/m)
        vs[i] = vs[i-1] + 0.5 * (ac[i-1] + ac[i]) * dt

        # Update time and random number r_0
        t[i] = t[i-1] + dt
        r[i] = np.random.normal(0, 1)

        # Implementing Andersen Thermostat
        if r[i] <= gamma * dt:
            vs[i] = np.random.normal(0, 1) * np.sqrt(k_B * T_target / m)

        # Update energies
        ks[i] = 0.5 * m * vs[i]**2
        us[i] = sbh_energy(xs[i])
        ets[i] = ks[i] + us[i]

    return xs, vs, t, ets, ks, ac, us
```

Task 8:

The dyanmics is run at 300K with the 1D SBH model preformed in TP3 and Verlet Velocity in TP4, using the parameters in table 1. The simulation starts at R_1 with low kinetic energy ($v0 \approx 0.001$).

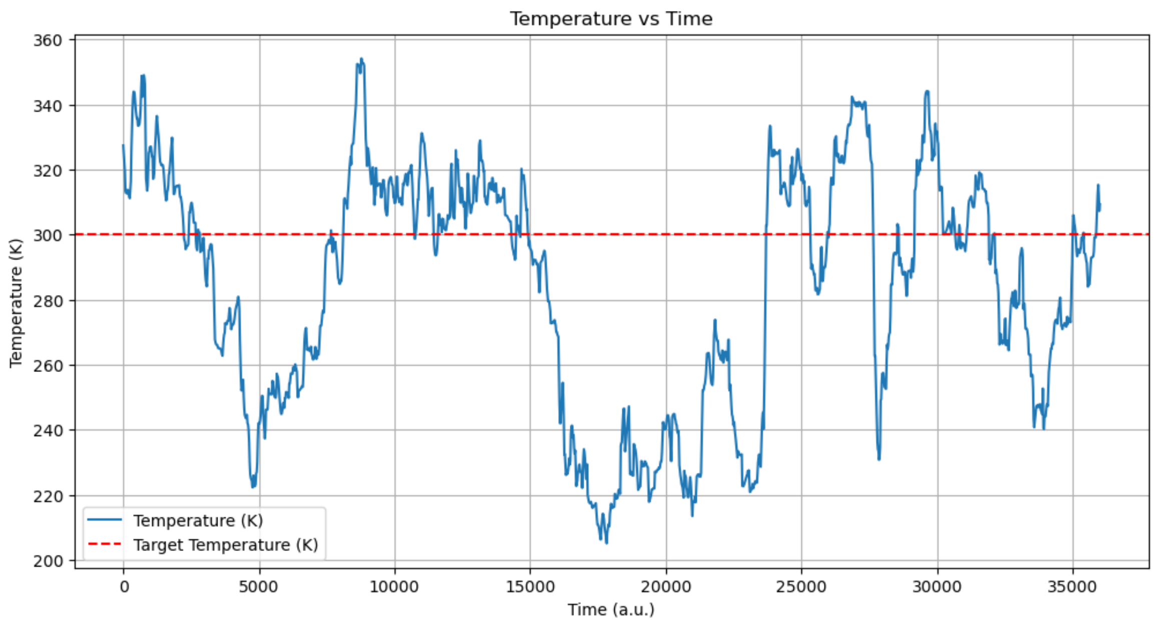
Here temperature can be approximated as the following:

$$T \approx \frac{2\langle E_k \rangle}{N k_B} \qquad eq. [14]$$

where $\langle E_k \rangle$ is the average kinetic energy and N is the number of dimensions, hence here: $N = 1$

Where the mean kinetic energy must be computed as a moving average over a number of previous steps. As shown below, we do not see the any jumps from the left to the right well as 300 K is too low for hopping between wells to be likely. From a plot of temperature vs time we can see how long it takes for the trajectory to reach its target temperature, in this case a target temperature of 300 K is used.

In [14]:
```
# Dyanmics Parameters
k_B = 3.1668114e-6  # Boltzmann constant in a.u.
T_target = 300 # Target temperature in Kelvin
gamma = 0.002  # Collision frequency
x0 = R_1  # Initial position, left well
v0 = 0.001  # Initial velocity
dt = 20  # Time step (a.u.)
n = 2000  # Number of steps for one picosecond in a.u.
window_size = 200 # Moving average window size for temperature

# Run simulation
xs, vs, t, ets, ks, ac, us = verlet_velTP5(x0, v0, dt, n, gamma, T_target)

# Compute moving average of kinetic energy and temperature
kinetic_avg = np.convolve(ks, np.ones(window_size) / window_size, mode="val
id")
temp = (2 * kinetic_avg) / (k_B)
new_time  = np.zeros(np.size(temp))
for i in range(1, np.size(temp)): # Makes both time and temperature arrays
the same length
    new_time[i] = new_time[i-1]+dt

# Plot of Temperature vs Time
plt.figure(figsize=(12, 6))
plt.plot(new_time, temp, label="Temperature (K)")
plt.axhline(y=T_target, color="r", linestyle="--", label="Target Temperatur
e (K)")
plt.xlabel("Time (a.u.)")
plt.ylabel("Temperature (K)")
plt.title('Temperature vs Time')
plt.legend()
plt.grid()
plt.show()
```



Task 9:

Estimate the temperature which corresponds to the minimum total energy for hopping from R_1 to R_2 obtained from task 6. Recall from task 6 that the minimum energy for hopping was found to be the following:

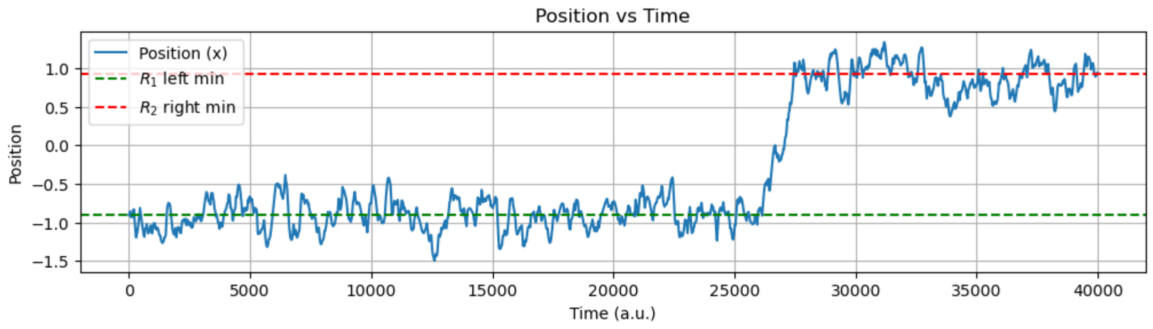
$$R_1 \ to \ R_2: --> (E_3 - E_1) = 0.0367 \ Hartree$$

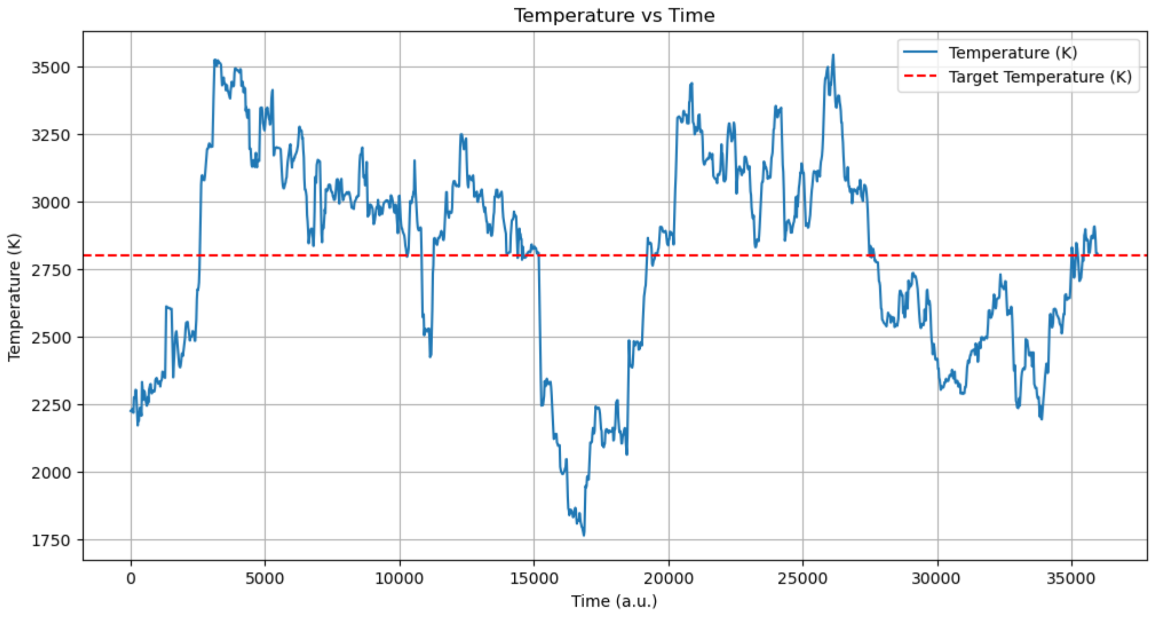
Finding the corresponding temperature can be found by manualy running the dyanmics, starting with a higher value of temperature which guarentees a hop from R_1 to R_2 ($\approx 10000 \ K$) and steadily decreasing it until no hopping can be seen when plotting position as a funciton of time.

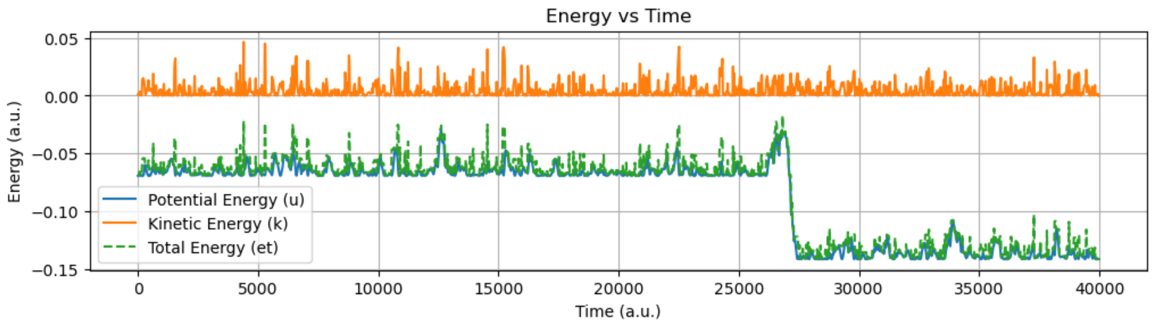
By manually running the dyanmics and as shown from the position vs time plot the minimum temperature required to hop from R_1 to R_2 is $\approx 2800 \ K$. A velocity vs position plot is also produced to further portray the hop between wells. However, this method is inaccurate and can provide different results everytime the dynamics runs.

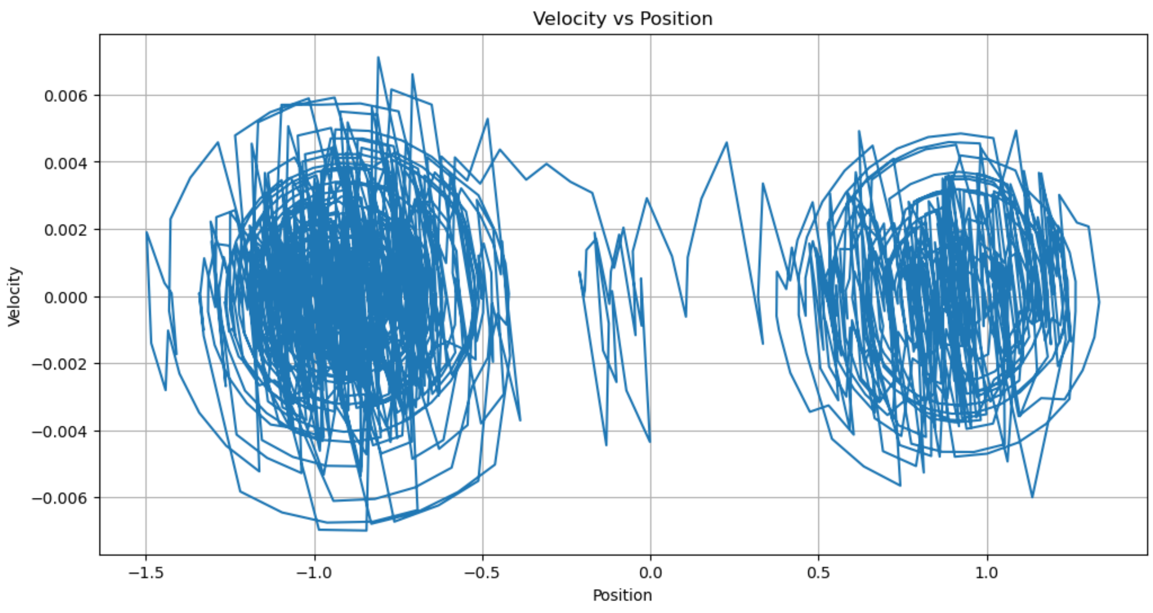
A more accurate method to determine the minimum temperature can be found below.

In [17]:
```python
min_T = 2800
xs, vs, t, ets, ks, ac, us = verlet_velTP5(x0, v0, dt, n, gamma, min_T)

kinetic_avg = np.convolve(ks, np.ones(window_size) / window_size, mode="val
id")
temp = (2 * kinetic_avg) / (k_B)
new_time  = np.zeros(np.size(temp))
for i in range(1, np.size(temp)): # Makes both time and temperature arrays
the same length
    new_time[i] = new_time[i-1]+dt

# Plotting results

# Plot of Position vs Time
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(t, xs, label="Position (x)")
plt.xlabel("Time (a.u.)")
plt.ylabel("Position")
plt.axhline(y=R_1, color="g", linestyle="--", label="$ R_1 $ left min")
plt.axhline(y=R_2, color="r", linestyle="--", label="$ R_2 $ right min")
plt.title('Position vs Time')
plt.legend()
plt.grid()
plt.show()

# Plot of Temperature vs Time
plt.figure(figsize=(12, 6))
plt.plot(new_time, temp, label="Temperature (K)")
plt.axhline(y=min_T, color="r", linestyle="--", label="Target Temperature
(K)")
plt.xlabel("Time (a.u.)")
plt.ylabel("Temperature (K)")
plt.title('Temperature vs Time')
plt.legend()
plt.grid()
plt.show()

# Plot of energies vs Time
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 2)
plt.plot(t, us, label="Potential Energy (u)")
plt.plot(t, ks, label="Kinetic Energy (k)")
plt.plot(t, ets, label="Total Energy (et)", linestyle="--")
plt.xlabel("Time (a.u.)")
plt.ylabel("Energy (a.u.)")
plt.title('Energy vs Time')
plt.legend()
plt.grid()
plt.show()

# Plot of Velocity vs Position
plt.figure(figsize=(12, 6))
plt.plot(xs, vs)
plt.xlabel("Position")
plt.ylabel("Velocity")
plt.title('Velocity vs Position')
plt.grid()
plt.show()
```


Position vs Time
Position (x)
R_1 left min
R_2 right min
Position
Time (a.u.)


Temperature vs Time
Temperature (K)
Target Temperature (K)
Temperature (K)
Time (a.u.)


Energy vs Time
Potential Energy (u)
Kinetic Energy (k)
Total Energy (et)
Energy (a.u.)
Time (a.u.)


Velocity vs Position
Velocity
Position
0.006
0.004
0.002
0.000
−0.002
−0.004
−0.006
−1.5
−1.0
−0.5
0.0
0.5
1.0

Accurate estimate of minimum temperature:

Although the previous method seemed to provide a close estimate of what has been achieved below, it is extremely improbable that the same results will be reproduced due to the system's randomness. Hence, it is more accurate and useful to present the probability of hopping from R_1 to R_2 as a function of temperature.

This is done by implementing a For loop, in which the dynamics can be run for a constant number of iterations, for each temperature step in an appropriate range. The loop measures the particle's position after each iteration, noting whether it has transitioned or not. If it has, it is added to a probability bucket. The probability of transitioning is measured as the number of successful transitions over the total number of iterations for a given temperature.

If $(x(t-1) - R_1) > (R_3 - R_1)$ then than a transition occurs for that iteration, hence transition probability becomes:

$$P_{\text{trans}} = \frac{\text{Successful Iterations}}{\text{Iterations per Temperature}} \qquad \text{(Eq. 15)}$$

For a balance between computational time and accuracy, the temperature range is set from $100\ K$ to $20000\ K$ in steps of $100\ K$ where the number of iterations is 50 per step, giving a total of 9950 cycles.

These parameters resulted in a computational time of 472 seconds, it has been reduced for your sake below while displaying a similar trend.

Using this we can accurately estimate the minimum temperature required for transitioning from R_1 to R_2, which was found to be $1700\ K$. This value is much more accurate than the previously obtained value $(2800\ K)$, being a $\approx 58\%$ difference between the two methods. This value for the minimum temperature correlates to the minimum energy value for transition received in TP3 and TP4. The same calculation can be carried out for estimating the minimum transition temperature from R_2 to R_1, where this temperature was found to be $5000\ K$.

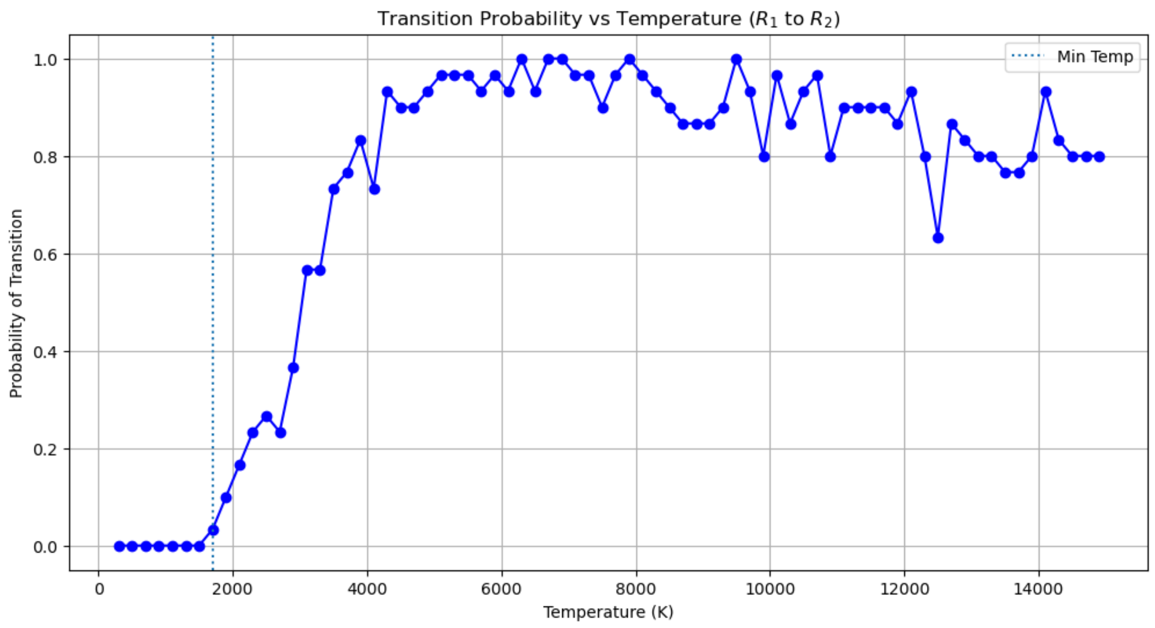
$$R_1\ to\ R_2 : --> (E_3 - E_1) = 0.0367\ Hartree \quad \propto 1700\ K$$

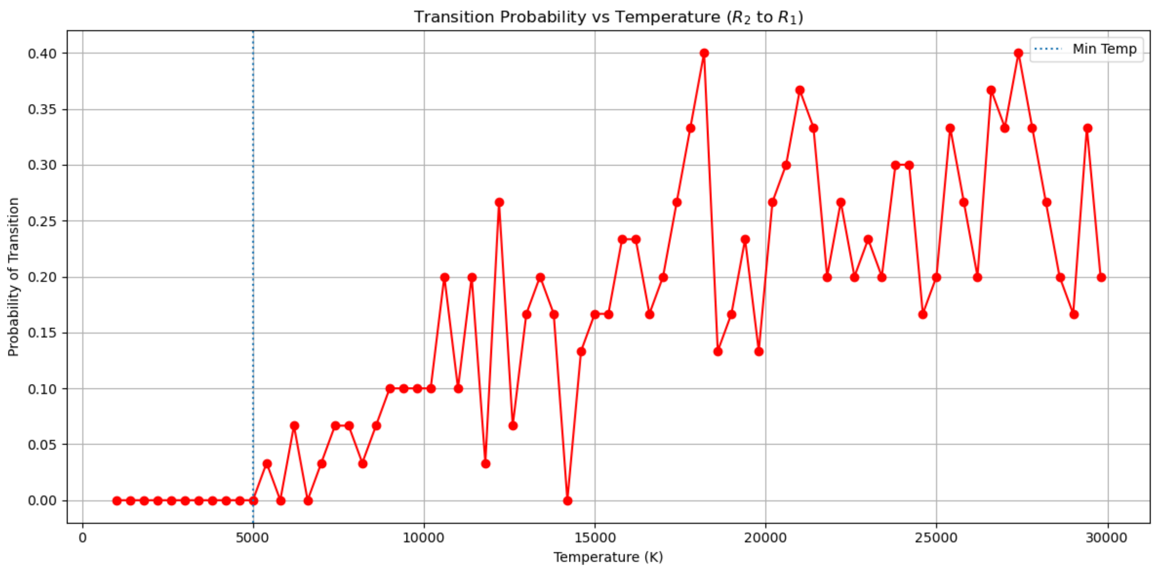
$$R_2\ to\ R_1 : --> (E_3 - E_2) = 0.1093\ Hartree \quad \propto 5000\ K$$

The minimum temperatures obtained are as expected, both having a factor 3 difference to their corresponding minimum energy. A noticeable trend in the R_1 to R_2 plot is at higher temperatures, the probability of transition can be seen to slowly decrease, this may be due to statistical instabilities and unpredicable behaviours which occur at increasing temperature. It may be also due to the Andersen thermostat's frequent velocity adjustments, which can disrupt the transitions even though the system has sufficient thermal energy to overcome the energy barrier.

Note the calculation below takes $\approx$ 136 *seconds*. *Reduce the iterations if you are time limited*

In [16]:
```
# Number of iterations for every temperature
iterations = 30

# Temperature range for R_1
tempsR_1 = np.arange(300, 15000, 200)
# Temeperature range for R_2
tempsR_2 = np.arange(1000, 30000, 400)

# Calculate probabilities
probR_1 = []
probR_2 = []

for T_target in tempsR_1:
    trans = 0
    for xs in range(iterations):  # Iterations per temperature
        xs, vs, t, ets, ks, ac, us = verlet_velTP5(R_1, v0, dt, n, gamma, T
_target)

        # Check if position changes from initial well (around R_1)
        if np.abs(xs[-1] - R_1) > np.abs(R_3 - R_1):  # Threshold for signi
ficant position change
            trans += 1

    probR_1.append(trans / iterations)

# Identical procedure for R_2
for T_target in tempsR_2:
    trans = 0
    for xs in range(iterations):
        xs, vs, t, ets, ks, ac, us = verlet_velTP5(R_2, v0, dt, n, gamma, T
_target)

        if np.abs(xs[-1] - R_2) > np.abs(R_3 - R_2):
            trans += 1

    probR_2.append(trans / iterations)

# Plotting Probability vs Temperautue for R_1 to R_2
plt.figure(figsize=(12, 6))
plt.plot(tempsR_1, probR_1, marker='o', color="b")
plt.axvline(x=1700, linestyle = 'dotted', label = "Min Temp")
plt.legend()
plt.xlabel('Temperature (K)')
plt.ylabel('Probability of Transition')
plt.title('Transition Probability vs Temperature $ (R_1 $ to $ R_2) $')
plt.grid()

# Plotting Probability vs Temperautue for R_2 to R_2
plt.figure(figsize=(12, 6))
plt.plot(tempsR_2, probR_2, marker='o', color="r")
plt.axvline(x=5000, linestyle = 'dotted', label = "Min Temp")
plt.legend()
plt.xlabel('Temperature (K)')
plt.ylabel('Probability of Transition')
plt.title('Transition Probability vs Temperature $ (R_2 $ to $ R_1) $')
plt.grid()


plt.tight_layout()
plt.show()
```


Transition Probability vs Temperature (R_1 to R_2)
Min Temp
Probability of Transition
Temperature (K)


Transition Probability vs Temperature (R_2 to R_1)
Min Temp
Probability of Transition
Temperature (K)