1.6

1.6.1, PLT

```
import numpy as np
import pylab as plt
import os
from sklearn.metrics import mean_absolute_error

S = [2, 5, 10, 16, 24, 27, 35, 50]
data = list(transformer(z) for z in S)

#How do I make it faster so I don't have multiple for loops
xList = [x[0] for x in data]
yList = [y[1] for y in data]

plt.xlabel('Guess')
plt.ylabel('Iterations')
plt.plot(xList, yList, '-o', color='black')

plt.savefig("Collatz_exercise.png", dpi = 300)

with open('Collatz.out', 'w') as file:
    for guess, iterations in data:
        file.write(f'Initial Guess: {guess}, Iterations: {iterations}\n')
```

1.6.2, String

```
def print_even(string_in):
    if (len(string_in) == 0):
        raise Exception("Empty String")

    evenChar = ""
    for x in range(1, len(string_in), 2):
        evenChar += string_in[x] + " "
    print(evenChar)

'''
def print_even(string_in):
    if (len(string_in) == 0):
        raise Exception("Empty String")
    evenChar = string_in[1::2]
    print(evenChar)
'''

print_even("Python")
#print_even("")

S = ["Gojackets", "Call me Ishmael", "ILoveChBE"]
for x in S:
    print_even(x)
```

1.6.3, Numbers

```
def get_tax(income):
    finalTax = 0
    if (income > 10000):
        finalTax = (income - 10000) * 0.1
    if (income > 25000):
        finalTax += (income - 25000) * 0.13
    return finalTax

get_tax(58500)
```

1.6.4, Reverse

```
def print_reverse(number):
    number_reverse = str(number)[::-1]
    print(number_reverse)

print_reverse(6572)

S = [1, 15, 658, 2940, 44112]
for number in S:
    print_reverse(number)
```

1.6.5, MAE

```
x = [-7,1,5,2,9,-2,0,1]
y = [-6,4,4.5,2,11,-2.1,1,3]

mean_absolute_error(x,y)
```

2.5

2.5.1, CaLTA

```python
atom = read('Data/ACAJIZ.cif')
atom.get_global_number_of_atoms()
cif.get_atomic_numbers()
print(cif.symbols.species())
print(cif.get_volume())

from ase.calculators.emt import EMT
from ase.units import Bohr,Hartree,mol,kcal,kJ,eV
from ase.optimize import BFGSLineSearch

print(f"Energy before relaxation: ", atom.get_total_energy())

atom.set_calculator(EMT())
optimizer = BFGSLineSearch(atom, trajectory='opt.traj', logfile='opt.log')
optimizer.run(fmax=0.3)

print(f"Energy after relaxation: ", atom.get_total_energy())
```

3.3

3.3.1, Test.py

- ssh ayoon37@login-ice.pace.gatech.edu → Type in password
- cd scratch
- mkdir Training
- nano test.py → print('Hello World') → ctrl+x, y, ¡enter¿

3.3.2, Bash Script

```
#!/bin/bash
#SBATCH --job-name=test_job #SBATCH --output=test.out
#SBATCH --error=test.err
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --time=5:00
cd /home/hice1/ayoon37/scratch/Training python Test.py
```

sbatch submit.sbatch to submit

3.3.3, Storage

- mkdir Storage:
- mv example.txt /Storage
- cd Storage
- nano Example.py

3.3.4, Delete

```
cd /home/hice1/ayoon37/scratch/Storage
rm *.py
```

5.4

SSH and Environment Setup

```
ssh ayoon37@login-ice.pace.gatech.edu
module load anaconda3
conda create -name vip5
conda activate vip5
conda install -c conda-forge ase
conda install -c conda-forge sparc-x
cd scratch
```

Slurm Submission Bash Code

```
#!/bin/bash
#SBATCH --job-name=test_job
#SBATCH --output=test.out
#SBATCH --error=test.err
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --time=5:00

cd /home/hice1/ayoon37/scratch
python code.py
```

5.4.1, PBE

```
from sparc import SPARC
from ase.build import molecule
from ase.units import Bohr,Hartree,mol,kcal,kJ,eV

# make the atoms
atoms = molecule('XXX')
atoms.cell = [[8,0,0],[0,8,0],[0,0,8]]
atoms.center()

# setup calculator
parameters = dict(
                EXCHANGE_CORRELATION = 'GGA_PBE',
                D3_FLAG=1,   #Grimme D3 dispersion correction
                SPIN_TYP=0,   #non spin-polarized calculation
                KPOINT_GRID=[1,1,1],  #molecule needs single kpt !
                ECUT=500/Hartree,   #set ECUT (Hartree) or h (Angstrom)
                #h = 0.15,
                TOL_SCF=1e-5,
                RELAX_FLAG=1, #Do structural relaxation (only atomic positions)
                PRINT_FORCES=1,
                PRINT_RELAXOUT=1)

calc = SPARC(atoms = atoms, **parameters)

# set the calculator on the atoms and run
atoms.set_calculator(calc)
print(atoms.get_potential_energy())
```

Replace the “XXX” with H2O, CO2, and NH3

	CO2	H2O	NH3
Free Energy per atom	-1.3026422117E+01 (Ha/atom)	-5.9091464488E+00 (Ha/atom)	-3.0423820785E+00 (Ha/atom)
Total free energy	-3.9079266350E+01 (Ha)	-1.7727439346E+01 (Ha)	-1.2169528314E+01 (Ha)
Exchange correlation energy	-1.0346416024E+01 (Ha)	-4.8723761321E+00 (Ha)	-4.1181041434E+00 (Ha)
Self and correction energy	-5.8356705267E+01 (Ha)	-2.6885066152E+01 (Ha)	-2.0889501184E+01 (Ha)

CO2	-4.1280012214
H2O	-2.7126289694
NH3	-0.8158049333000003

	CO2	H2O	NH3
Total Time	68.228s	69.386s	141.823s
Calculation Time	0.050s	0.052s	0.034s

Comparison of my results vs experimentation:

- The variance between my results and experimentation seem to be quite large (when considering percentile off), however since the scale is so small, these results/approximations could possibly be considered to be fairly accurate.

Comparison between DFT Methods (B3LYP vs. PBE) I could not resolve this functional. I couldn't find the function in the documentation (https://github.com/SPARC-X/SPARC-X-API/blob/master/sparc/calculator.py) and when asked on slack, the answer proved inconclusive. The following comparison/results are from what I found online/independent research.

- Accuracy: B3LYP is a hybrid functional and generally offers better accuracy for various properties compared to PBE, which is a generalized gradient approximation (GGA) functional. B3LYP tends to provide more accurate energies but at a higher computational cost.
- Computational Time: B3LYP calculations typically take longer than PBE due to the increased complexity of the functional. B3LYP involves a mix of exact Hartree-Fock exchange with DFT, leading to higher computational demand.

5.4.2, CaLTA

W/O SPIN & D3 DISPERSION CORRECTION:

```
from sparc import SPARC
from ase.build import molecule
from ase.units import Bohr,Hartree,mol,kcal,kJ,eV
from ase.io import read, write

# make the atoms
atoms = molecule('CaLTA.vasp')
atoms.cell = [[8,0,0],[0,8,0],[0,0,8]]
atoms.center()

# setup calculator
parameters = dict(
                EXCHANGE_CORRELATION = 'GGA_PBE',
                D3_FLAG=0,   #Grimme D3 dispersion correction
                SPIN_TYP=0,   #non spin-polarized calculation
                KPOINT_GRID=[1,1,1],  #molecule needs single kpt !
                ECUT=500/Hartree,   #set ECUT (Hartree) or h (Angstrom)
                #h = 0.15,
                TOL_SCF=1e-5,
                RELAX_FLAG=1, #Do structural relaxation (only atomic positions)
                PRINT_FORCES=1,
                PRINT_RELAXOUT=1)

calc = SPARC(atoms = atoms, **parameters)

# set the calculator on the atoms and run
atoms.set_calculator(calc)
print(atoms.get_potential_energy() / eV)
```

W/ SPIN & D3 DISPERSION CORRECTION:

- D3_FLAG= 1
- SPIN_TYP= 1

Results

	W/O	W/
Free energy per atom	7.7324776074E+00 (Ha/atom)	-7.7324776074E+00 (Ha/atom)
Total free energy	-5.7220334294E+02 (Ha)	-5.7220334294E+02 (Ha)
Exchange correlation energy	-2.9469300704E+02 (Ha)	-2.9469300704E+02 (Ha)
Self and correction energy	-1.1191877875E+03 (Ha)	-1.1191877875E+03 (Ha)
Time for force calculation	1.910s	1.910s
Relax step time	298.654s	298.654s

Comparison of the results
The results with spin/D3 dispersion are equal to the results without.

Spin polarization accounts for the difference in energy between spin-up and spin-down electrons in a system. D3 dispersion correction, on the other hand, improves the accuracy of energy calculations by including dispersion forces that are not captured by traditional exchange-correlation functionals. When both spin polarization and D3 dispersion correction are applied to a calculation, their combined effect might influence the total energy, yet in some scenarios, their impact might cancel each other out or result in minimal alteration. In cases where the system's properties are not strongly affected by spin polarization or dispersion interactions, the results with or without these corrections might converge, leading to the same calculated energy.

5.4.3, LiH5C5N2O5

SPARC

```
from sparc import SPARC
from ase.build import molecule
from ase.units import Bohr,Hartree,mol,kcal,kJ,eV
from ase.io import read, write

# make the atoms
atoms = read('LiH5C5N2O5.cif')
atoms.pbc = True
atoms.cell = [[8,0,0],[0,8,0],[0,0,8]]
atoms.center()

# setup calculator
parameters = dict(
                EXCHANGE_CORRELATION = 'GGA_PBE',
                D3_FLAG=0,   #Grimme D3 dispersion correction
                SPIN_TYP=0,   #non spin-polarized calculation
                KPOINT_GRID=[1,1,1],  #molecule needs single kpt !
                ECUT=500/Hartree,   #set ECUT (Hartree) or h (Angstrom)
                #h = 0.15,
                TOL_SCF=1e-5,
                RELAX_FLAG=0, #Do structural relaxation (only atomic positions)
                PRINT_FORCES=1,
                PRINT_RELAXOUT=1)

calc = SPARC(atoms = atoms, **parameters)

# set the calculator on the atoms and run
atoms.set_calculator(calc)
print(atoms.get_potential_energy())
```

ASE

```
from ase import Atoms
from ase.calculators.emt import EMT  # Example EMT potential, replace with your
    desired calculator

atoms = Atoms('LiH5C5N2O5')
atoms.calc = EMT()

potential_energy = atoms.get_potential_energy()
print("Potential Energy per atom (eV/atom):", potential_energy / len(atoms))
```

PLOT

```
import numpy as np
import matplotlib.pyplot as plt
from ase.build import molecule
from ase.calculators.emt import EMT

atoms = molecule('LiH5C5N2O5')

calc = EMT()
atoms.set_calculator(calc)

original_volume = atoms.get_volume()
original_cell = atoms.get_cell()

percentage_change = np.linspace(-0.20, 0.20, 5)  # -20% to 20% in 5 steps
```

```

volumes = []
energies = []

for change in percentage_change:
    atoms_copy = atoms.copy()

    new_a = original_cell[0, 0] * (1 + change)
    scaled_positions = atoms_copy.get_scaled_positions()
    scaled_positions[:, 0] *= (1 + change)

    atoms_copy.set_cell([new_a, original_cell[1, 1], original_cell[2, 2]],
    scale_atoms=True)
    atoms_copy.set_scaled_positions(scaled_positions)

    energy = atoms_copy.get_potential_energy()

    volumes.append(atoms_copy.get_volume())
    energies.append(energy)

plt.figure(figsize=(8, 6))
plt.plot(volumes, energies, marker='o', linestyle='-')
plt.xlabel('Cell Volume')
plt.ylabel('Potential Energy (eV)')
plt.title('Energy vs. Cell Volume')
plt.grid(True)
plt.show()
```

CUBIC EQUATION

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

volumes = np.array(volumes)
energies = np.array(energies)

coefficients = np.polyfit(volumes, energies, 3)  # Fit a polynomial of order 3 (cubic
    )

def eos_function(volume):
    return np.polyval(coefficients, volume)

result = minimize(eos_function, volumes.mean(), method='Nelder-Mead')

optimal_volume = result.x[0]
minimized_energy = result.fun

plt.figure(figsize=(8, 6))
plt.plot(volumes, energies, marker='o', linestyle='-', label='Energy vs. Volume')
plt.plot(optimal_volume, minimized_energy, marker='o', markersize=8, color='red',
    label='Minimum Energy')
plt.xlabel('Cell Volume')
plt.ylabel('Potential Energy (eV)')
plt.title('Energy vs. Cell Volume with Fitted EOS')
plt.legend()
plt.grid(True)
plt.show()
```

6.4

6.4.1, Structural Relaxation

BUILD

```
from ase import Atoms
from ase.build import molecule

CO2 = molecule('CO2')
CO2.set_distance(0, 1, distance=1.8)
print(CO2.get_positions())
```

SPARC

```
from sparc import SPARC
from ase.build import molecule
from ase.units import Bohr,Hartree,mol,kcal,kJ,eV

# make the atoms
atoms = molecule('CO2')
atoms.set_distance(0, 1, distance=1.8)
atoms.cell = [[8,0,0],[0,8,0],[0,0,8]]
atoms.center()

# setup calculator
parameters = dict(
                EXCHANGE_CORRELATION = 'GGA_PBE',
                D3_FLAG=0,   #Grimme D3 dispersion correction
                SPIN_TYP=0,   #non spin-polarized calculation
                KPOINT_GRID=[1,1,1],  #molecule needs single kpt !
                ECUT=500/Hartree,   #set ECUT (Hartree) or h (Angstrom)
                #h = 0.15,
                TOL_SCF=1e-5,
                RELAX_FLAG=1, #Do structural relaxation (only atomic positions)
                PRINT_FORCES=1,
                PRINT_RELAXOUT=1)

calc = SPARC(atoms = atoms, **parameters)

# set the calculator on the atoms and run
atoms.set_calculator(calc)
print(atoms.get_potential_energy())
```

QUANTUM ESPRESSO

```
image = molecule('CO2',vacuum=10.)
image.set_cell([10, 10, 10])
image.set_pbc([1,1,1])
image.center()
image.rattle(0.0001)
calc = Espresso(atoms=image,
                pw=500.0,
                xc='PBE',
                kpts="gamma")

dyn = BFGS(image,logfile='opt_pbe.log', trajectory='h2o_pbe_optimization.traj')
dyn.run(fmax=0.005)
image.calc.close()

print(image.get_potential_energy())
```

6.4.3, CO_2 Adsorption

RELAX PRISTINE MOF

```
import os
from ase import Atoms , io
from ase.io import read , write
from ase.build import bulk , molecule , surface , add_adsorbate
from ase.units import Bohr ,Hartree ,mol ,kcal ,kJ ,eV
from ase.constraints import FixAtoms
from sparc import SPARC

parameters = dict(
                EXCHANGE_CORRELATION = 'GGA_PBE',
                D3_FLAG=1,   #Grimme D3 dispersion correction
                SPIN_TYP=0,   #spin-polarized calculation
                KPOINT_GRID=[1,1,1],
                ECUT=600/Hartree ,   #set ECUT (Hartree) or h (Angstrom)
                #h = 0.15,
                TOL_SCF=1e-4,
                RELAX_FLAG=1,
                TOL_RELAX = 2.00E-03,  #convergence criteria (maximum force) (Ha/Bohr
    )
                PRINT_FORCES=1,
                PRINT_RELAXOUT=1)

cwd = os.getcwd()
parameters['directory'] = cwd + '/Exercise_3/pristine'

atoms = read('CaLTA.vasp')
c = FixAtoms (indices = [atom.index for atom in atoms])
atoms.set_constraint(c)

calc = SPARC(atoms = atoms , **parameters)
atoms.set_calculator(calc)

eng_pristine = atoms.get_potential_energy()
atoms.write('Exercise_3/CONTCAR_3_pristine')
eng_pristine
```

SINGLE SITE

```
atoms = read('CaLTA.vasp')
c = FixAtoms (indices = [atom.index for atom in atoms])
atoms.set_constraint(c)

CO2 = CO2_original.copy()
CO2.cell = atoms.cell
CO2.rotate(-45, 'y', 'COM')
d_trans = atoms[73].position - CO2[0].position + np.array([1,1,2])
CO2.translate(d_trans)
atoms.extend(CO2)

atoms.write('Exercise_3/POSCAR_3_SS')?
```

DUAL SITE

```
atoms = read('CaLTA.vasp')
c = FixAtoms (indices = [atom.index for atom in atoms])
atoms.set_constraint(c)

CO2 = CO2_original.copy()
```

```
CO2.cell = atoms.cell
d_trans = atoms[73].position - CO2[1].position + np.array([3.55,1,0])
CO2.translate(d_trans)
atoms.extend(CO2)

atoms.write('Exercise_3/POSCAR_3_DS')
```

RESULTS
Pristine: -0.203
Single Site: -0.767
Dual Site: -0.470

CONCLUSION:
Seeing as the greater adsorption energy equates to a more stable active site - it should also be noted that we are taking the absolute value of the results, for the negative simply signifies attraction/adsorption. Furthermore there is a clear ”peak” in the results at the single site. As coverage increases, the most stable sites will saturate first, forcing CO2 to adsorb at secondary sites, resulting in increasingly stable active sites.

7.3

7.3.1, Dimensions

ENERGY CALCULATION

```
from ase.build import bulk, molecule, surface, add_adsorbate
from ase.constraints import FixAtoms
from ase.io import read
from ase.units import Bohr,Hartree,mol,kcal,kJ,eV
from sparc import SPARC
import numpy as np
import json, os, re

atoms = molecule('C')
atoms.cell = [[10,0,0],[0,10,0],[0,0,10]]
atoms.center()

parameters = dict(
                EXCHANGE_CORRELATION = 'GGA_PBE',
                D3_FLAG=1,   #Grimme D3 dispersion correction
                SPIN_TYP=0,   #non spin-polarized calculation
                KPOINT_GRID=[1,1,1],  #slab needs only 1 kpt in z-direction
                ECUT=800/Hartree,   #set ECUT (Hartree) or h (Angstrom)
                #h = 0.15,
                TOL_SCF=1e-5,
                RELAX_FLAG=0,
                TOL_RELAX = 2.00E-03,  #convergence criteria (maximum force) (Ha/Bohr
    )
                PRINT_FORCES=1,
                PRINT_RELAXOUT=1)


cwd = os.getcwd()
eng_list, t_list = [], []

for i in np.arange(2, 15, 1):
    i = int(i)
    dir_i = f"{cwd}/C_gas_lat/calc_{i}"

    atoms.cell = [[i,0,0],[0,i,0],[0,0,i]]
    parameters['directory'] = dir_i
    calc = SPARC(atoms = atoms, **parameters)
    atoms.set_calculator(calc)
    eng = atoms.get_potential_energy()
    eng_list.append(eng)

    sparc_out = os.path.join(dir_i, 'SPARC.out')
    with open(sparc_out, 'r') as f: data = f.read()
    t_SPARC = float(re.search(r'Total walltime\s+:\s+([0-9.]+)', data).group(1))
    t_SPARC = f"{t_SPARC:.3f}"
    t_list.append(t_SPARC)


json.dump(eng_list, open(f"{cwd}/C_gas_lat/eng_list.json", 'w'))
json.dump(t_list, open(f"{cwd}/C_gas_lat/t_list.json", 'w'))
```

PLOT AGAINST L

```
import numpy as np
import pylab as plt
```

```python

L = np.arange(2,15)
E = [] # FILL THIS IN
E = json.load(open(f"{cwd}/C_gas_lat/eng_list.json", 'r'))

plt.plot(L[3:], E[3:], 'o-');
plt.title('Box dimension convergence testing for C');
plt.xlabel('Lattice parameter [$\AA$]');
plt.ylabel('Energy [eV]');

print("Energy converges around the lattice parameter of 8 Ang.")
```

CONCLUSION:
The minimal dimension I would consider using is 3, due to it being the absolute minimum of the graph. Energy converges around the lattice parameter of 8 Ang. The lattice converges at this point for that is the absolute minimum energy of the given lattice parameter. Furthermore, as stated in section 6.4.3, the greater (absolute value) energy value equates to the more stable site, which is why 8 Ang is where the energy converges.

7.3.2, Energy Per Atom

CONVERGENCE TEST WITH MESH SPACING

```python
image = bulk('Cu')

parameters = dict(
                EXCHANGE_CORRELATION = 'GGA_PBE',
                D3_FLAG=1,   #Grimme D3 dispersion correction
                SPIN_TYP=0,   #non spin-polarized calculation
                KPOINT_GRID=[6,6,1],  #slab needs only 1 kpt in z-direction
                #ECUT=500/Hartree,   #set ECUT (Hartree) or h (Angstrom)
                #h = 0.15,
                TOL_SCF=1e-5,
                RELAX_FLAG=0,
                TOL_RELAX = 2.00E-03,  #convergence criteria (maximum force) (Ha/Bohr
    )
                PRINT_FORCES=1,
                PRINT_RELAXOUT=1)


cwd = os.getcwd()
eng_list, t_list = [], []

for i in np.arange(0.10, 0.21, 0.01):
    i = round(i, 2)
    dir_i = f"{cwd}/Cu_h/calc_{i}"

    parameters['h'] = i
    parameters['directory'] = dir_i
    calc = SPARC(atoms = image, **parameters)
    image.set_calculator(calc)
    eng = image.get_potential_energy()
    eng_per_atom = eng / len(image)
    eng_list.append(eng_per_atom)

    sparc_out = os.path.join(dir_i, 'SPARC.out')
    with open(sparc_out, 'r') as f: data = f.read()
    t_SPARC = float(re.search(r'Total walltime\s+:\s+([0-9.]+)', data).group(1))
    t_SPARC = f"{t_SPARC:.3f}"
    t_list.append(t_SPARC)


json.dump(eng_list, open(f"{cwd}/Cu_h/eng_list.json", 'w'))
json.dump(t_list, open(f"{cwd}/Cu_h/t_list.json", 'w'))
```

PLOT

```
k = np.arange(0.10, 0.21, 0.01)
E = [] # FILL THIS IN
CPU_time = [] # FILL THIS IN
E = json.load(open(f"{cwd}/Cu_h/eng_list.json", 'r'))
CPU_time = json.load(open(f"{cwd}/Cu_h/t_list.json", 'r'))

fig, axs = plt.subplots(1,2, figsize=(15,5))
ax2 = axs[0].twinx();
axs[0].plot(k, E, 'bo-');
ax2.plot(k, CPU_time, 'ro-');

plt.title('Mesh spacing convergence testing for FCC Cu');
axs[0].set_xlabel('Mesh spacing [$\AA$]');
axs[0].set_ylabel('Energy [eV]', color = 'b');
ax2.set_ylabel('runtime [second]', color = 'r');

axs[1].plot(k[:-5], E[:-5], 'bo-');
axs[1].set_xlabel('Mesh spacing [$\AA$]');
axs[1].set_ylabel('Energy [eV]', color = 'b');
plt.subplots_adjust(wspace=0.4)

print(np.array(E[0:-1]) - E[1:])
```

Energy converges around the mesh spacing of 0.1 Ang, which is not practical (< 0.01 eV per atom).

CONVERGENCE TEST WITH ENERGY-CUTOFF

```
image = bulk('Cu')

parameters = dict(
                EXCHANGE_CORRELATION = 'GGA_PBE',
                D3_FLAG=1,   #Grimme D3 dispersion correction
                SPIN_TYP=0,   #non spin-polarized calculation
                KPOINT_GRID=[6,6,1],  #slab needs only 1 kpt in z-direction
                #ECUT=500/Hartree,   #set ECUT (Hartree) or h (Angstrom)
                #h = 0.15,
                TOL_SCF=1e-5,
                RELAX_FLAG=0,
                TOL_RELAX = 2.00E-03,  #convergence criteria (maximum force) (Ha/Bohr
    )
                PRINT_FORCES=1,
                PRINT_RELAXOUT=1)


cwd = os.getcwd()
eng_list, t_list = [], []

for i in np.arange(500, 1200, 100):
    i = round(i, 2)
    dir_i = f"{cwd}/Cu_ECUT/calc_{i}"

    parameters['ECUT'] = i/Hartree
    parameters['directory'] = dir_i
    calc = SPARC(atoms = image, **parameters)
    image.set_calculator(calc)
    eng = image.get_potential_energy()
    eng_per_atom = eng / len(image)
    eng_list.append(eng_per_atom)

    sparc_out = os.path.join(dir_i, 'SPARC.out')
    with open(sparc_out, 'r') as f: data = f.read()
    t_SPARC = float(re.search(r'Total walltime\s+:\s+([0-9.]+)', data).group(1))
    t_SPARC = f"{t_SPARC:.3f}"
    t_list.append(t_SPARC)


json.dump(eng_list, open(f"{cwd}/Cu_ECUT/eng_list.json", 'w'))
json.dump(t_list, open(f"{cwd}/Cu_ECUT/t_list.json", 'w'))
\end{LS}
```

PLOT

```
k = np.arange(500, 1200, 100)
E = [] # FILL THIS IN
CPU_time = [] # FILL THIS IN
E = json.load(open(f"{cwd}/Cu_ECUT/eng_list.json", 'r'))
CPU_time = json.load(open(f"{cwd}/Cu_ECUT/t_list.json", 'r'))

fig, axs = plt.subplots(1,2, figsize=(15,5))
ax2 = axs[0].twinx();
axs[0].plot(k, E, 'bo-');
ax2.plot(k, CPU_time, 'ro-');

plt.title('Mesh spacing convergence testing for FCC Cu');
axs[0].set_xlabel('Mesh spacing [$\AA$]');
axs[0].set_ylabel('Energy [eV]', color = 'b');
ax2.set_ylabel('runtime [second]', color = 'r');

axs[1].plot(k[3:], E[3:], 'bo-');
axs[1].set_xlabel('Mesh spacing [$\AA$]');
axs[1].set_ylabel('Energy [eV]', color = 'b');
plt.subplots_adjust(wspace=0.4)
```

Energy converges around the mesh spacing of 0.1 Ang, which is not practical (< 0.01 eV per atom)."

SPARC CONVERGENCE TEST WITH MESH SPACING

```
from sparc import SPARC
from ase.build import molecule
from ase.units import Bohr,Hartree,mol,kcal,kJ,eV
from ase.io import read, write

# make the atoms
atoms = molecule('Cu)
atoms.cell = [[8,0,0],[0,8,0],[0,0,8]]
atoms.center()

# setup calculator
parameters = dict(
                EXCHANGE_CORRELATION = 'GGA_PBE',
                D3_FLAG=0,   #Grimme D3 dispersion correction
                SPIN_TYP=0,   #non spin-polarized calculation
                KPOINT_GRID=[1,1,1],  #molecule needs single kpt !
                #ECUT=500/Hartree,   #set ECUT (Hartree) or h (Angstrom)
                h = 0.15,
                TOL_SCF=1e-5,
                RELAX_FLAG=1, #Do structural relaxation (only atomic positions)
                PRINT_FORCES=1,
                PRINT_RELAXOUT=1)

calc = SPARC(atoms = atoms, **parameters)

# set the calculator on the atoms and run
atoms.set_calculator(calc)
print(atoms.get_potential_energy() / eV)
```

SPARC CONVERGENCE TEST WITH ENERGY-CUTOFF

```
from sparc import SPARC
from ase.build import molecule
from ase.units import Bohr,Hartree,mol,kcal,kJ,eV
from ase.io import read, write

# make the atoms
atoms = molecule('Cu)
atoms.cell = [[8,0,0],[0,8,0],[0,0,8]]
atoms.center()

# setup calculator
parameters = dict(
                EXCHANGE_CORRELATION = 'GGA_PBE',
                D3_FLAG=0,   #Grimme D3 dispersion correction
                SPIN_TYP=0,   #non spin-polarized calculation
                KPOINT_GRID=[1,1,1],  #molecule needs single kpt !
                ECUT=500/Hartree,   #set ECUT (Hartree) or h (Angstrom)
                #h = 0.15,
                TOL_SCF=1e-5,
                RELAX_FLAG=1, #Do structural relaxation (only atomic positions)
                PRINT_FORCES=1,
                PRINT_RELAXOUT=1)

calc = SPARC(atoms = atoms, **parameters)

# set the calculator on the atoms and run
atoms.set_calculator(calc)
print(atoms.get_potential_energy() / eV)
```

7.3.3, Diamond Unit Cell

CONVERGENCE WITH K-POINT MESH

```
from ase.spacegroup import crystal
image = bulk('Pt', 'fcc')

parameters = dict(
                EXCHANGE_CORRELATION = 'GGA_PBE',
                D3_FLAG=1,   #Grimme D3 dispersion correction
                SPIN_TYP=0,   #non spin-polarized calculation
                KPOINT_GRID=[4,4,4],  #bulk needs kpoints in all directions
                ECUT=800/Hartree,   #set ECUT (Hartree) or h (Angstrom)
                #h = 0.15,
                TOL_SCF=1e-5,
                RELAX_FLAG=0,
                TOL_RELAX = 2.00E-03,  #convergence criteria (maximum force) (Ha/Bohr
    )
                PRINT_FORCES=1,
                PRINT_RELAXOUT=1)


cwd = os.getcwd()
eng_list, t_list = [], []

for i in np.arange(1, 10, 1):
    i = round(i, 2)
    dir_i = f"{cwd}/bulk_kpoint/calc_{i}"

    #if os.path.isdir(dir_i):
    #    print(f"{dir_i} already exists.")
    #    continue

    parameters['KPOINT_GRID'] = [i,i,i]
    parameters['directory'] = dir_i
    calc = SPARC(atoms = image, **parameters)
    image.set_calculator(calc)
    eng = image.get_potential_energy()
    eng_per_atom = eng / len(image)
    eng_list.append(eng_per_atom)

    sparc_out = os.path.join(dir_i, 'SPARC.out')
    with open(sparc_out, 'r') as f: data = f.read()
    t_SPARC = float(re.search(r'Total walltime\s+:\s+([0-9.]+)', data).group(1))
    t_SPARC = f"{t_SPARC:.3f}"
    t_list.append(t_SPARC)


json.dump(eng_list, open(f"{cwd}/bulk_kpoint/eng_list.json", 'w'))
json.dump(t_list, open(f"{cwd}/bulk_kpoint/t_list.json", 'w'))
```

PLOT

```
import numpy as np
import pylab as plt

k = np.arange(1, 10, 1)
E = [] # FILL THIS IN
CPU_time = [] # FILL THIS IN
E = json.load(open(f"{cwd}/bulk_kpoint/eng_list.json", 'r'))
CPU_time = json.load(open(f"{cwd}/bulk_kpoint/t_list.json", 'r'))

fig, axs = plt.subplots(1,2, figsize=(15,5));
ax2 = axs[0].twinx();
axs[0].plot(k, E, 'bo-');
ax2.plot(k, CPU_time, 'ro-');

plt.title('K-point convergence testing for bulk');
axs[0].set_xlabel('K-point mesh');
axs[0].set_ylabel('Energy [eV]', color = 'b');
ax2.set_ylabel('runtime [second]', color = 'r');

axs[1].plot(k[5:], E[5:], 'bo-');

plt.subplots_adjust(wspace=0.4)
axs[1].set_xlabel('K-point mesh');
axs[1].set_ylabel('Energy [eV]', color = 'b');
```

```

print(np.array(E[0:-1]) - E[1:])
print("Energy converges around the k-mesh of [9,9,9] (~ 0.01 eV per atom).")
```

CONVERGENCE WITH LATTICE CONSTRAINTS

```
#image = bulk('Pt', 'fcc')

parameters = dict(
                EXCHANGE_CORRELATION = 'GGA_PBE',
                D3_FLAG=1,   #Grimme D3 dispersion correction
                SPIN_TYP=0,   #non spin-polarized calculation
                KPOINT_GRID=[7,7,7],  #bulk needs kpoints in all directions
                ECUT=800/Hartree,   #set ECUT (Hartree) or h (Angstrom)
                #h = 0.15,
                TOL_SCF=1e-5,
                RELAX_FLAG=0,
                TOL_RELAX = 2.00E-03,  #convergence criteria (maximum force) (Ha/Bohr
    )
                PRINT_FORCES=1,
                PRINT_RELAXOUT=1)


cwd = os.getcwd()
eng_list, t_list = [], []

for i in np.arange(0.95, 1.05, 0.01):
    i = round(i, 2)
    dir_i = f"{cwd}/bulk_lattice/calc_{i}"

    image = bulk('Pt', 'fcc')
    image.set_cell(image.get_cell() * i, scale_atoms=True)

    parameters['directory'] = dir_i
    calc = SPARC(atoms = image, **parameters)
    image.set_calculator(calc)
    eng = image.get_potential_energy()
    eng_per_atom = eng / len(image)
    eng_list.append(eng_per_atom)

    sparc_out = os.path.join(dir_i, 'SPARC.out')
    with open(sparc_out, 'r') as f: data = f.read()
    t_SPARC = float(re.search(r'Total walltime\s+:\s+([0-9.]+)', data).group(1))
    t_SPARC = f"{t_SPARC:.3f}"
    t_list.append(t_SPARC)


json.dump(eng_list, open(f"{cwd}/bulk_lattice/eng_list.json", 'w'))
json.dump(t_list, open(f"{cwd}/bulk_lattice/t_list.json", 'w'))
```

PLOT

```
import numpy as np
import pylab as plt

k = np.arange(0.95, 1.05, 0.01)
E = [] # FILL THIS IN
CPU_time = [] # FILL THIS IN
E = json.load(open(f"{cwd}/bulk_lattice/eng_list.json", 'r'))
CPU_time = json.load(open(f"{cwd}/bulk_lattice/t_list.json", 'r'))

fig, axs = plt.subplots(1,2, figsize=(15,5));
```

```
#ax2 = axs[0].twinx();
axs[0].plot(k, E, 'bo-');
#ax2.plot(k, CPU_time, 'ro-');

plt.title('Lattice constant convergence testing for bulk');
axs[0].set_xlabel('Lattice constant multiplier');
axs[0].set_ylabel('Energy [eV]', color = 'b');
#ax2.set_ylabel('runtime [second]', color = 'r');

axs[1].plot(k[3:-2], E[3:-2], 'bo-');

plt.subplots_adjust(wspace=0.4)
axs[1].set_xlabel('Lattice constant multiplier');
axs[1].set_ylabel('Energy [eV]', color = 'b');

print(np.array(E[0:-1]) - E[1:])
print("Energy converges at the lattice constant multiplier of 1.00 (the lowest energy
    ).")
```

8.6

8.6.1, Kernel Regression

DATAFRAME

```
import pandas as pd
file_path = 'ethanol_spectrum.csv'
ethanol_data = pd.read_csv(file_path)
```

KERNEL REGRESSION

```
import pandas as pd
import numpy as np
from scipy.interpolate import Rbf
import matplotlib.pyplot as plt

ethanol_data.sort_values(by='Wavenumber', inplace=True)
X = ethanol_data['Wavenumber'].values.reshape(-1, 1)  # Reshaping for Rbf
y = ethanol_data['Intensity'].values

sigma = 100
rbf_model = Rbf(X, y, function='gaussian', epsilon=sigma)
y_pred = rbf_model(X)

mae = np.mean(np.abs(y - y_pred))
print(f"Mean Absolute Error (MAE): {mae}")

plt.figure(figsize=(8, 6))
plt.scatter(X, y, label='Original Data', color='blue')
plt.plot(X, y_pred, label='Fitted Model', color='red')
plt.scatter(X, y, label='Model Training Points', color='green')

plt.xlabel('Wavenumber')
plt.ylabel('Intensity')
plt.title('Radial Basis Function Kernel Regression')
plt.legend()
plt.grid(True)
plt.show()
```

EVERY 3RD

```
X_train = X[::3]
y_train = y[::3]
```

When training on every third point, the fit might not capture the intricacies of the data as closely as when using all data points. Consequently, the MAE increases compared to training on the entire data set.

SIGMA = 1

```
sigma = 1
```

With a smaller sigma value (1 in this case), the fitted model now has over fitting. This will lead to the inclusion of outliers and higher MAE for new data points.

8.6.3, PCA

IMPORT DIABETES

```python
from sklearn.datasets import load_diabetes
diabetes = load_diabetes()

X = diabetes.data  # Features
y = diabetes.target  # Target
```

PCA

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

pca = PCA(n_components=10)
pca.fit(X)
principal_components = pca.components_

cov_matrix = np.cov(principal_components)

plt.figure(figsize=(8, 6))
plt.imshow(cov_matrix, cmap='viridis', interpolation='nearest')
plt.colorbar(label='Covariance')
plt.title('Covariance Matrix of Principal Components')
plt.xlabel('Principal Components')
plt.ylabel('Principal Components')
plt.show()
```

Higher Variance Components: The diagonal elements with larger values correspond to principal components that retain more information from the original features. These components capture more variation in the data.

Orthogonality: Due to the nature of PCA, the principal components are orthogonal to each other, which means they are uncorrelated. Therefore, off-diagonal elements should ideally be close to zero, indicating little to no covariance between different principal components.

Decay of Variance Explained: It's common to observe a decay in the variance explained by each subsequent principal component. The first few principal components usually retain most of the variance, while subsequent components carry less information about the data.

MINIMUM FEATURES

```python
from sklearn.decomposition import PCA
pca = PCA()
pca.fit(X)

cumulative_variance_ratio = np.cumsum(pca.explained_variance_ratio_)
n_components_90 = np.argmax(cumulative_variance_ratio >= 0.90) + 1  # +1 to start
    from 1, not 0

print(f"Number of components to capture 90% variance: {n_components_90}")
```

PLOT

```python
def myplot(score,coeff,labels=None):
    xs = score[:,0]
    ys = score[:,1]
    n = coeff.shape[0]
    scalex = 1.0/(xs.max() - xs.min())
    scaley = 1.0/(ys.max() - ys.min())
    #plt.scatter(xs * scalex,ys * scaley, c = Y1)
    for i in range(n):
        plt.arrow(0, 0, coeff[i,0], coeff[i,1],color = 'r',alpha = 0.5)
        if labels is None:
            plt.text(coeff[i,0]* 1.15, coeff[i,1] * 1.15, "Var"+str(i+1), color = 'g', ha = 'center', va = 'center')
        else:
            plt.text(coeff[i,0]* 1.15, coeff[i,1] * 1.15, labels[i], color = 'g', ha = 'center', va = 'center')
plt.xlim(-1,1)
plt.ylim(-1,1)
plt.xlabel("PC{}".format(1))
plt.ylabel("PC{}".format(2))
plt.grid()
```

Self Assessment

Needed Improvements
Misjudged Difficulty: Initially, I underestimated the trajectory of the training, assuming a sustained level of complexity aligned with my background. This led to an incorrect assumption about the forthcoming exercises' difficulty after the introductory phase, affecting my preparedness.

Approach to Documentation: Instead of investing more time in methodically delving into the documentation, I adopted a less efficient approach of trial-and-error, hoping for solutions by random attempts. This ineffective method hindered both progress and substantial learning.

Research Focus: Redirecting my research readings towards SPARC articles and a comprehensive understanding of library/package functionality rather than delving into unrelated research papers on DFT and computational chemistry might have been more beneficial.

Continued Learning: Acknowledging the need for further exploration and comprehension of packages and functions within Quantum Espresso, SPARC, and ASE remains a priority for my ongoing development.

Accomplishments

Despite identified areas for improvement, I've made substantial progress in several key areas:

Documentation Navigation: Strengthened my ability to navigate through documentation effectively and implement new functions accurately, particularly within the realm of open-source code. This proficiency has become more intuitive and seamless.

Reinforcement of Machine Learning Skills: The exercises served as valuable reinforcement opportunities, allowing me to practically apply concepts learned in my Machine Learning class (CS 4641), thereby enhancing my skills in this domain.

Data Visualization Proficiency: Acquired practical experience in data visualization techniques, particularly utilizing matplotlib, which has bolstered my proficiency in this essential aspect of analysis.

Technological Aptitude: Significantly improved familiarity and comfort with Linux, PACE, and Bash. From initially having no knowledge or familiarity, I now adeptly navigate and execute commands without continual dependence on external references or documentation.