UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

ZDIM source code overview

Zacharias Liasi [†]

Supplementary document for “Electric response properties of silver clusters”

Date: January 6, 2020

Department of Chemistry, University of Copenhagen

[†]UCPH mail: flx527@alumni.ku.dk

Contents

Notice

Due to the majority of the code within the programme being rewritten and reformatted in late November 2019, certain non-essential parts of the programme have, at the time of this document's creation, not yet been updated for use within the version described here. Furthermore, as a consequence of heavy testing around the time of this document's creation, certain code snippets displayed here may vary from those used to achieve the results presented in "Electric response properties of silver clusters", including but not pertained to numerical values. All documentation strings have been removed from the function definitions shown in this document, due to the lengthiness of the complete code. All documentation strings are present in the uncut source code available through the link below, though it should be noted that no guarantee is given regarding the correctness of the doc. strings, as these are the last part of a function to be corrected when changes happen.

The most recent version of the source code is available at: https://github.com/zliasi/ZDIM

Date: January 6, 2020

Libraries

For full functionality the following modules are needed.

```
import argparse
import os
import sys
import numpy as np
import time
import psutil
```

As the programme is composed of a multitude of custom functions, rather than being a large continuous piece of declarative code, the source code has been split into five separate files–not counting the main script–that each contains a collections of functions. The five modules are all contained within a package named `zdimpy`, which needs to be imported into the main script.

```
from zdimpy import (
    fread as f,
    calc,
    shprint as sh,
    txtprint as txt,
    plot
)
```

Package/Module	Use	Documentation
argparse	Initiate parser arguments	Link
os	File handling and path checking	Link
sys	Reading argument input	Link
numpy	Array objects and operations, and linear algebra tools	Link
time	Process and function timing	Link
psutil	Monitoring memory and processor usage	Link
fread	File reading functions	N/A
calc	Computational functions	N/A
shprint	File output (shell)	N/A
txtprint	File output (.txt)	N/A
plot	Graphical plot output	N/A

argparse configurations

See the table at page 5 for summary of the arguments defined below.

Before the initiation of the parser and its arguments, a couple of custom settings are setup. First off, the default method the `argparse` parser uses to read the lines of a given text file containing arguments, is not compatible with arguments containing multiple values, e.g. vector input such as the external electrical field vector. This is due to the way the end of an argument is defined in the default method (Ref. argparse documentation). So in order to make the parser able to handle vector type arguments, i.e. arguments containing white space, a new definition of the used function is written as

```
def convert_arg_line_to_args(arg_line):
    for arg in arg_line.split():
        if not arg.strip():
            continue
        yield arg
```

The programme needs up to three files to run:

1. .xyz file containing the coordinates of the system
2. .out file containing the frequency values
3. .txt file containing the arguments for the parser

Therefore, a new argument type is defined, which first checks whether or not the given file path is valid. If it is valid the file path is returned as a string constant, otherwise an error is raised.

```
def file_path(path):
    if not os.path.exists(path):
        raise argparse.ArgumentTypeError("{} does not exist".format(path))

    return path
```

Since the default return type of a vector input is a list, a new argument action is setup. The following class takes the given input and returns it as an NumPy array. As the external electrical field is the only input argument which contains a set of values, rather than a single, the class also contains an if statement, which raises an error if the given list is not of the correct length.

```
class StoreAsArray(argparse._StoreAction):
    def __call__(self, parser, namespace, values, option_string=None):
        values = np.array(values)
        if len(values) != 3:
            raise argparse.ArgumentTypeError(
                "{0} is not a valid vector".format(values)
            )
        return super().__call__(parser, namespace, values, option_string)
```

The parser is then imitated as

```
parser = argparse.ArgumentParser(
    fromfile_prefix_chars='@',
    description="computations of response properties",
)
```

As the parser has been initiated, the default function used by the parser to read lines in a file containing arguments is overwritten as

```
parser.convert_arg_line_to_args = convert_arg_line_to_args
```

This is a very kludge way of replacing the default function, an alternative would be to subclass `ArgumentParser` and then add the before mentioned function (Ref. Stack Overflow).

Since the programme needs to be usable for a variety of systems, a couple of "modes" have been setup:

- mode 1 runs the calculations for a single nanoparticle
- mode 2 runs the calculations for a single nanoparticle with a neighbouring molecule
- mode 3 runs the calculations for a single nanoparticle with predefined polarizability values

The argument is added to the parser as

```
parser.add_argument(
    "-m", "--mode",
    metavar="",
    type=int,
    requried=True,
    help="choose mode"
)
```

The argument for the absolute file path of the .xyz file is added to the parser as

```
parser.add_argument(
    "-x", "--xyz",
    metavar="",
    dest="xyz_file_path",
    type=file_path,
    requried=True,
    help="absolute path of .xyz file"
)
```

The argument for the absolute file path of the .out file is added to the parser as

```
parser.add_argument(
    "-o", "--out",
    metavar="",
    dest="out_file_path",
    type=file_path,
    required=True,
    help="absolute path of .out file"
)
```

The argument for the external electric field vector is added to the parser as

```
parser.add_argument(
    "-e", "--external",
     metavar="",
     dest="E_external",
     type=int,
     nargs="+",
     action=StoreAsArray,
     default=np.array([5, 0, 0]),
     help="cartesian values for the external field vector"
)
```

The argument for the absolute path of the output file containing the calculated values is added to the parser as

```
parser.add_argument(
    "-p", "--path",
    metavar="",
    dest="output_path",
    type=file_path,
    default=os.path.dirname(os.path.abspath(sys.argv[0])),
    help="absolute path for the output files"
)
```

A mutually exclusive group is added to store the options of `verbose` and `quiet`, in order to give the user some control over how much data, as well as what type/format, should be returned from the programme at the end of each runtime, as

```
group =  parser.add_mutually_exclusive_group()
```

The verbose option returns the maximum amount of graphical plots as well as plain data.

```
group.add_argument(
    "-v", "--verbose",
    action="store_true",
    help="give more output"
)
```

The quiet option only returns a prints the induced dipole moment in the terminal.

```
group.add_argument(
    "-q", "--quiet",
    action="store_true",
    help="give less output"
)
```

If non of the above flags are used, a moderate amount of plain data will be returned as well as a graphical plot of the induced dipole moment as a function of the frequency.

The argument for the return of the programme version is added as

```python
parser.add_argument(
    "-V", "--version",
    action="version",
    version="%(prog)s v5.1",
    help="show version and exit"
)
```

As the last step regarding the arguments, the arguments are parsed as

```python
args = parser.parse_args()
```

Now to make sure that the `StoreAsArray` class returned the input argument as a proper Numpy array, the following assertion is set to execute before any of the arguments are used.

```python
assert isistance(args.E_external, np.ndarray), "invalid type for external"
```

In order to limit the programme to the defined "modes" the given value for the mode argument is asserted as well.

```python
assert args.mode == 1 or args.mode == 2 or args.mode == 3, "invalid mode"
```

Argument	Use	Options	Required
`-h`, `--help`	Prints help menu	True, False	No
`-m`, `--mode`	Sets calculation type	1, 2, 3	Yes
`-x`, `--xyz`	Absolute path for the .xyz file	Path	Yes
`-o`, `--out`	Absolute path for the .out file	Path	Yes
`-e`, `--external`	List of the x, y, and z values of the field vector	x y z	No
`-p`, `--path`	Absolute path of the .txt output file	Path	No
`-v`, `--verbose`	Maximum amount of output	True, False	No
`-q`, `--quiet`	Minimum amount of output	True, False	No
`-V`, `--version`	Prints programme version and exits	True, False	No
`@`	Reads arguments from .txt file	Path	No

zdimpy

fread

The purpose of the the `fread` module is to provide functions for reading specific data from text files. The first function in the file reads a given .xyz file, and returns the values as NumPy arrays. The function is defined as

```python
def xyz(path):
    coordinates = []
    x_coordinates = []
    y_coordinates = []
    z_coordinates = []

    with open(path) as fp:
        n_atoms = fp.readline()
        title = fp.readline()
        for line in fp:
            atom, x, y, z = line.split()
            coordinates.append([float(x), float(y), float(z)])
            x_coordinates.append([float(x)])
            y_coordinates.append([float(y)])
            z_coordinates.append([float(z)])

    return (
        np.asarray(coordinates),
        np.asarray(x_coordinates),
        np.asarray(y_coordinates),
        np.asarray(z_coordinates)
    )
```

The above function is used for all three modes. Though, when the .out file is read one of three functions is used, depending on which mode has been used for the current runtime, as different values are needed from the .out file depending on what type of system the calculations are run for.

In case of the mode being set to 3, the following function is executed. The function reads a given .out file, and returns the polarizabilities and frequencies as separate NumPy arrays. The polarizabilites are only read when the mode is set to 2 or 3, as for mode 1, the polarizabilites are computed by the programme itself.

```
def out_NP(path):
    freq = []
    aNP_xx = []
    aNP_yy = []
    aNP_zz = []

    with open(path) as fp:
        for line in fp:
            if "XDIPLEN   XDIPLEN" in line:
                XX = line.strip().split()
                aNP_xx.append(complex(float(XX[4]), float(XX[5])))
                freq.append(float(XX[3]))

            if "YDIPLEN   YDIPLEN" in line:
                YY = line.strip().split()
                aNP_yy.append(complex(float(YY[4]), float(YY[5])))

            if "ZDIPLEN   ZDIPLEN" in line:
                ZZ = line.strip().split()
                aNP_zz.append(complex(float(ZZ[4]), float(ZZ[5])))

    return (
        np.asarray(freq),
        np.asarray(aNP_xx),
        np.asarray(aNP_yy),
        np.asarray(aNP_zz)
    )
```

In case of the mode being set to 2, the following function is executed. The function is currently not up to date, as it has not been used since the programme has been rewritten.

```python
def out_MOL(path):
    freq = []
    aMOL_xx = []
    aMOL_yy = []
    aMOL_zz = []

    with open(path) as fp:
        for line in fp:
            if "XDIPLEN   XDIPLEN" in line:
                XX = line.strip().split()
                aMOL_xx.append(complex(float(XX[4]), float(XX[5])))
                freq.append(float(XX[3]))

            if "YDIPLEN   YDIPLEN" in line:
                YY = line.strip().split()
                aMOL_yy.append(complex(float(YY[4]), float(YY[5])))

            if "ZDIPLEN   ZDIPLEN" in line:
                ZZ = line.strip().split()
                aMOL_zz.append(complex(float(ZZ[4]), float(ZZ[5])))

    return (
        np.asarray(freq),
        np.asarray(aMOL_xx),
        np.asarray(aMOL_yy),
        np.asarray(aMOL_zz)
    )
```

In case of the mode being set to 1, the following function is executed. The function reads the frequencies from the .out file and stores them in a NumPy array.

```python
def freq(path):
    freq = []

    with open(path) as fp:
        for line in fp:
            if "XDIPLEN   XDIPLEN" in line:
                XX = line.strip().split()
                freq.append(float(XX[3]))

    return np.asarray(freq)
```

calc

The purpose of the functions contained in the `calc` module, is to do the main calculations. When the programme is set to compute the polarizabilities for the nanoparticle, the following function is executed. The function simply initiates a set of values, depending on the type of nanoparticle (copper, silver or gold), which are needed for the calculations. This function is therefore not needed if the polarizabilities are read from an .out file. Three sets are initiated in the beginning, these are used for the following if...elif...else statements, which check if any of the entries in one of the sets are present in the name of the given .xyz file. So depending on what element name is given in the file name, a different set of values are returned.

```
def pol_par(xyz_file_name):
    cu_set = {"cu", "copper", "kobber"}
    ag_set = {"ag", "silver", "sølv"}
    au_set = {"au", "gold", "guld"}

    if any(name in xyz_file_name.lower() for name in cu_set):
        pl_freq = 0.39799501649970476
        ex_freq = np.array(
            [
                [0.01069405],
                [0.10866771],
                [0.19477134],
                [0.41085727]
            ]
        )
        pl_str = np.array(
            [
                [0.00224171],
                [0.00382193],
                [0.02656975],
                [0.02344606]
            ]
        )
        ex_lftm = np.array(
            [
                [0.01389124],
                [0.03880727],
                [0.11807553],
                [0.15820578]
            ]
        )
        stat_pol = 33.7420
```

[...]

```python

    elif any(name in xyz_file_name.lower() for name in ag_set):
        pl_freq = 0.3311160928
        ex_freq = np.array(
            [
                [0.02998787255],
                [0.1646760501],
                [0.3007974716],
                [0.3337988314],
                [0.7456543310]
            ]
        )
        pl_str = np.array(
            [
                [0.002388739848],
                [0.004556980633],
                [0.0004042482819],
                [0.03086986880],
                [0.2074896182]
            ]
        )
        ex_lftm = np.array(
            [
                [0.1428098931],
                [0.01661092940],
                [0.002388739848],
                [0.03366285693],
                [0.08889787218]
            ]
        )
        stat_pol = 49.9843
```

[...]

```

    elif any(name in xyz_file_name.lower() for name in au_set):
        pl_freq = 0.33184626029476766
        ex_freq = np.array(
            [
                [0.01525096],
                [0.03050193],
                [0.1091087],
                [0.15816903],
                [0.4895008]
            ]
        )
        pl_str = np.array(
            [
                [0.00088198],
                [0.00036749],
                [0.0026092],
                [0.02208633],
                [0.16110897]
            ]
        )
        ex_lftm = np.array(
            [
                [0.00885658],
                [0.01267851],
                [0.0319719],
                [0.09165278],
                [0.08136297]
            ]
        )
        stat_pol = 31.0400

    else:
        raise NameError(
            "{0} is not a valid file name".format(xyz_file_name)
        )

    return (
        pl_freq,
        ex_freq,
        pl_str,
        ex_lftm,
        stat_pol
    )
```

Note: The above numerical values are not those used to yield the results seen in “Electric response properties of silver clusters”.

In order to calculate the spatial distance between each atom, as well as the spatial distance from the origin to each atom, the following function is executed.

```python
def spatial_dist(coordinates):
    return (
        np.linalg.norm(coordinates - coordinates[:, None], axis=-1),
        np.linalg.norm(np.array([0, 0, 0]) - coordinates[:, None], axis=-1)
    )
```

The difference between each x-coordinate, each y-coordinate, and each z-coordinate is computed as

```python
def point_diff(coordinates):
    return (
        (coordinates - coordinates[:, None])[..., 0],
        (coordinates - coordinates[:, None])[..., 1],
        (coordinates - coordinates[:, None])[..., 2]
    )
```

Note: `None` is `np.newaxis`.

The following function is used to compute the dipole-dipole interaction tensor components.

```python
def T(x_diff, y_diff, z_diff, p_dist):
    with np.errstate(divide="ignore", invalid="ignore"):
        T_xx = ((3 * x_diff**2) / p_dist**5) - (1 / p_dist**3)
        T_yy = ((3 * y_diff**2) / p_dist**5) - (1 / p_dist**3)
        T_zz = ((3 * z_diff**2) / p_dist**5) - (1 / p_dist**3)
        T_xy = ((3 * x_diff * y_diff) / p_dist**5)
        T_xz = ((3 * x_diff * z_diff) / p_dist**5)
        T_yz = ((3 * z_diff * y_diff) / p_dist**5)

    T_xx[np.isnan(T_xx)] = 0
    T_yy[np.isnan(T_yy)] = 0
    T_zz[np.isnan(T_zz)] = 0
    T_xy[np.isnan(T_xy)] = 0
    T_xz[np.isnan(T_xz)] = 0
    T_yz[np.isnan(T_yz)] = 0

    return (
        T_xx,
        T_yy,
        T_zz,
        T_xy,
        T_xz,
        T_yz
    )
```

As the first step in the assembly of the A matrix, the newly initiated T tensor components are stacked in accordance with the proper structure of the A matrix.

```python
def tensor_stack(arrays):
    arrays = np.asarray(arrays)
    n, p, q = arrays.shape
    s = int(round(np.sqrt(n)))
    arrays = arrays.reshape(s, -1, p, q)

    return arrays.transpose(2, 0, 3, 1).reshape(s * p, -1)
```

The T tensors needs to be negative and complex, so in order to make them that, the following function is executed.

```python
def tensor_gentrificator(tensor):
    tensor = np.negative(tensor)
    return tensor.astype(complex)
```

In case the polarizabilities are read from an .out file, the following function is used to insert the polarizabilities as the diagonal in the array containing all the T tensors as

```python
def nano_diag(aNP_xx, aNP_yy, aNP_zz, temp_A, i):
    temp_aNP = np.vstack((aNP_xx, aNP_yy, aNP_zz))
    diag_len = int(len(temp_A[1, :]) / 3)
    arr = temp_aNP[:, i]
    diag_list = np.hstack([arr for element in range(diag_len)])
    diag_array = np.asarray(diag_list)
    np.fill_diagonal(temp_A, diag_array)
    return np.linalg.inv(temp_A)
```

Note: the `i` variable is initiated in the for loop this function is within, using the `enumerate()` function, and is used to extract the correct frequency value from the frequency array.

The gist of the function is, that the arrays containing each Cartesian element of the polarizability are stacked, whereafter the polarizability values matching the frequency used for the current loop are extracted. The polarizabilities are then inserted into the diagonal of the A matrix using list comprehension. The reason for not using the `numpy.fill_diagonal()` function, or a similar one, is that the three components of the pre-calculated polarizability are different. This means that three different values have to be inserted as the diagonal in a dynamically scaled array, in the correct repeating order, matching the length of the diagonal, and this cannot be done with any of the NumPy functions.

If the polarizabilities are not read from an .out file, they are computed with the following function, which also inserts them as the diagonal in the combined T tensor array.

```python
def nano_pol(temp_A, pl_freq, ex_freq, pl_str, ex_lftm, stat_pol, freq, i):
    aNP = (np.sum(
        stat_pol * ((pl_str * pl_freq**2) /
                    ((ex_freq**2 - freq[i]**2) - ((freq[i] * ex_lftm) * 1j)))
    ) / 5)
    np.fill_diagonal(temp_A, aNP)

    return (
        np.linalg.inv(temp_A),
        aNP
    )
```

The permanent electric field is calculated as

```python
def E(o_dist, E_external, dipole_x, dipole_y, dipole_z, coordinates,
   x_coordinates, y_coordinates, z_coordinates):
    E_x = E_external[0] + (
        dipole_x * (1 / (o_dist**3) - 3 * ((x_coordinates**2) / (o_dist**5)))
        + dipole_y * ((- 3 * x_coordinates * y_coordinates) / o_dist**5)
        + dipole_z * (-(3 * x_coordinates * z_coordinates) / o_dist**5)
    )

    E_y = E_external[1] + (
        dipole_y * (1 / (o_dist**3) - 3 * ((y_coordinates**2) / (o_dist**5)))
        + dipole_x * ((- 3 * y_coordinates * x_coordinates) / o_dist**5)
        + dipole_z * (- (3 * y_coordinates * z_coordinates) / o_dist**5)
    )

    E_z = E_external[2] + (
        dipole_z * (1 / (o_dist**3) - 3 * ((z_coordinates**2) / (o_dist**5)))
        + dipole_y * ((- 3 * z_coordinates * y_coordinates) / o_dist**5)
        + dipole_x * (-(3 * y_coordinates * z_coordinates) / o_dist**5)
    )

    E = []
    for element in zip(E_x, E_y, E_z):
        E.extend(element)

    E = np.asarray(E)
    E = E.astype(complex)

    return E
```

The calculation of the induced dipole moment is initiated in the main script.

shprint

The main purpose of the functions within the `shprint` module, it to print the results as well as additional information and data in the terminal instance used to execute the programme. The following function is used in case of the polarizabilities being read from an .out file, in which case the frequency values are printed in a.u., eV, and cm^{-1} as well as the polarizabilities (a.u.).

```
def header(freq, xyz_file_name, out_file_name, coordinates, aNP_xx, aNP_yy,
    aNP_zz):
    freq_eV = np.array(freq * 27.211396)
    freq_cm = np.array(freq * 219474.6305)
    print("\n" + "="*80 + "\n")
    print("{:^80}".format("ZDIM") + "\n")
    print("{:^80}".format("Computations of response properties") + "\n")
    print("="*80 + "\n"*2)
    print("Input files:")
    print("    .xyz file: {}".format(xyz_file_name))
    print("    .out file: {}".format(out_file_name))
    print("\nTotal no. of atoms: {}".format(len(coordinates)) + "\n"*2)
    print("="*56)
    print("{:^56}".format("FREQUENCIES"))
    print("="*56 + "\n")
    print("-"*56)
    print("        " + " (a.u.) "+"\t    ", " (eV) "+"\t   ", "(cm-1) ")
    print("-"*56)
    print("        ", np.array_str(np.c_[freq, freq_eV, freq_cm], precision=3)
        .replace(" ", "     ").replace(
        '[', '   ').replace(']', '').strip())
    print("-"*56)
    print("Total number of frequencies:", len(freq))
    print("-"*56 + "\n")
    print("*Conversion: 1 a.u. = 27.211396 eV = 219474.6305 cm-1" + "\n"*2)
    print("="*69)
    print("{:^69}".format("POLARIZABILITIES (a.u.)"))
    print("="*69 + "\n")
    print("-"*69)
    print("   " + "XX "+"\t"*3, " YY "+"\t"*3, "ZZ")
    print("-"*69)
    print(
        "   " + np.array_str(np.c_[aNP_xx, aNP_yy, aNP_zz], precision=4).
            replace(
            '[', '').replace(']', '').replace('j', 'i').strip()
    )
    print("\n"*2 + "="*56)
    print("{:^56}".format("INDUCED DIPOLE MOMENTS"))
    print("="*56 + "\n"*2)
```

As the calculations are run through a for loop for each frequency, the following function is initiated in order to print the value of the frequency being used in the current calculation.

```
def iteration(freq, i):
    print("-" * 56)
    print("# {0} | Frequency = {1} a.u.".format(i + 1, freq[i]))
    print("-" * 56)
```

The function defined below prints the Cartesian components of the frequency-dependent induced dipole moment for the molecule and nanoparticle (this is only used in case mode == 2 is True), as well as the number of iterations the loop has run through in order to yield the result.

```python
def dipole(counter, dipole):
    print("Number of iterations:", counter)
    print("-" * 25)
    print(
        "\nMolecule dipole (x) = {0:.4e} {1} {2:.4e}i a.u.".format(
            dipole[0, 0].real, "+-"[dipole[0, 0].imag < 0],
            abs(dipole[0, 0].imag))
    )
    print(
        "Molecule dipole (y) = {0:.4e} {1} {2:.4e}i a.u.".format(
            dipole[1, 0].real, "+-"[dipole[1, 0].imag < 0],
            abs(dipole[1, 0].imag))
    )
    print(
        "Molecule dipole (z) = {0:.4e} {1} {2:.4e}i a.u.".format(
            dipole[2, 0].real, "+-"[dipole[2, 0].imag < 0],
            abs(dipole[2, 0].imag))
    )
    print(
        "\nNanoparticle dipole (x) = {0:.4e} {1} {2:.4e}i a.u.".format(
            np.sum(dipole[3:len(dipole):3]).real, "+-"
            [np.sum(dipole[3:len(dipole):3]).imag < 0],
            abs(np.sum(dipole[3:len(dipole):3]).imag))
    )
    print(
        "Nanoparticle dipole (y) = {0:.4e} {1} {2:.4e}i a.u.".format(
            np.sum(dipole[4:len(dipole):3]).real, "+-"
            [np.sum(dipole[4:len(dipole):3]).imag < 0],
            abs(np.sum(dipole[4:len(dipole):3]).imag))
    )
    print(
        "Nanoparticle dipole (z) = {0:.4e} {1} {2:.4e}i a.u.\n".format(
            np.sum(dipole[5:len(dipole):3]).real, "+-"
            [np.sum(dipole[5:len(dipole):3]).imag < 0],
            abs(np.sum(dipole[5:len(dipole):3]).imag))
    )
```

In case of the calculations being run for only a nanoparticle (mode == 1 or mode == 3 is True), the following function is initiated in order to print the induced dipole moment for the nanoparticle, as well as the number of iterations of the loop.

```
def NP_dipole(counter, dipole):
    print("Number of iterations:", counter)
    print("-" * 25)
    print(
        "\nNanoparticle dipole (x) = {0:.4e} {1} {2:.4e}i a.u.".format(
            np.sum(dipole[0:len(dipole):3]).real, "+-"
            [np.sum(dipole[0:len(dipole):3]).imag < 0],
            abs(np.sum(dipole[0:len(dipole):3]).imag))
    )
    print(
        "Nanoparticle dipole (y) = {0:.4e} {1} {2:.4e}i a.u.".format(
            np.sum(dipole[1:len(dipole):3]).real, "+-"
            [np.sum(dipole[1:len(dipole):3]).imag < 0],
            abs(np.sum(dipole[1:len(dipole):3]).imag))
    )
    print(
        "Nanoparticle dipole (z) = {0:.4e} {1} {2:.4e}i a.u.\n".format(
            np.sum(dipole[2:len(dipole):3]).real, "+-"
            [np.sum(dipole[2:len(dipole):3]).imag < 0],
            abs(np.sum(dipole[2:len(dipole):3]).imag))
    )
```

Since the transition is seen as a steep increase in the induced dipole and a decrease in the absorption, the following function is set to print the maximum value of the induced dipole moment and the minimum of the absorption, as well as the frequency at which the max. and min. are.

```
def min_max(freq, abs_x, abs_y, abs_z, dip_x, dip_y, dip_z):
    print("=" * 56)
    print()
    print(
        "Real maximum is at: Frequency = {0} a.u. (#{1})".format(
            freq[dip_x.index(np.max(dip_x))], dip_x.index(np.max(dip_x))+1)
    )
    print(
        "\nReal maximum = {:.4e} a.u.\n".format(np.max(dip_x))
    )
    print(
        "Imag. minimum is at: Frequency = {0} a.u. (#{1})".format(
            freq[abs_x.index(np.min(abs_x))], abs_x.index(np.min(abs_x))+1)
    )
    print(
        "\nImag. minimum = {:.4e}i a.u.\n".format(np.min(abs_x))
    )
    print("=" * 56, "\n")
```

The last function in the shprint module prints the time elapsed and the memory used by the programme.

```
def misc(start_time):
    process = psutil.Process(os.getpid())

    print()
    print("=" * 80)
    print(
        "Process time =", "{:.4f} seconds\n".format(
            (time.time() - start_time))
    )

    print(
        "Process memory usage =", "{:.4f} GB\n".format(
            process.memory_info().rss * (9.31*10**(-10)))
    )
    print(
        "CPU usage = {0}%".format(psutil.cpu_percent())
    )
    print("=" * 80, "\n")
    print()
    print("{:^80}".format("**** END ****"))
```

txtprint

The main purpose of the functions within the `txtprint` module is, as the name indicates, to print, what is printed in the terminal by the functions of `shprint`, in a .txt file. This is done by firstly initiating a new .txt file, with an automatically generated name, as

```python
def initialise_file(xyz_filename, out_filename, output_path):
    i = 0
    while os.path.exists("{0}{1}_{2}.zdim".format(
        output_path, xyz_filename.split('.', 1)[0], str(i)
    )):
        i += 1

    txt_output = "{0}{1}_{2}.zdim".format(
        output_path, xyz_filename.split('.', 1)[0], str(i)
    )

    with open(txt_output, "w") as txt_file:
        print("\n" + "=" * 56, file=txt_file)
        print("zdim_v4x", file=txt_file)
        print("Induced dipoles for {0} and {1}".format(
            out_filename, xyz_filename), file=txt_file
        )
        print("=" * 56, "\n", file=txt_file)

    return txt_output
```

This function first checks if a file with the automatically initiated name already exists. If the name is taken, the numerical value in the name is simply incremented by one. After the new file has been initiated, the rest of the functions are fundamentally the same as those in `shprint`, with the difference being the destination of the print statements. The `txtprint` module is at the time of writing not updated for use in the newest version of the programme, and is therefore not displayed in full here.

plot

The `plot` module contains functions for outputting graphical plots of the induced dipole moment and polarizability as a function of the frequency. Since the variables containing the results gets overwritten after each successful loop, the results has to be saved at the break of the main loop. In order to do so a set of empty lists are initiated as

```
def MOL_dipole_append(dipMOL_x, dipMOL_y, dipMOL_z, absMOL_x, absMOL_y,
    absMOL_z, dipole):
    return (
        dipMOL_x.append(np.real(dipole[0, 0])),
        dipMOL_y.append(np.real(dipole[1, 0])),
        dipMOL_z.append(np.real(dipole[2, 0])),
        absMOL_x.append(np.imag(dipole[0, 0])),
        absMOL_z.append(np.imag(dipole[2, 0])),
        absMOL_y.append(np.imag(dipole[1, 0]))
    )
```

The lists initiated by the above function are for a molecule, the appending function for a nanoparticle is given as

```
def NP_dipole_append(NP_dip_x, NP_dip_y, NP_dip_z, NP_abs_x, NP_abs_y,
    NP_abs_z, dipole):
    return (
        NP_dip_x.append(np.real(np.sum(dipole[0:len(dipole):3]))),
        NP_dip_y.append(np.real(np.sum(dipole[1:len(dipole):3]))),
        NP_dip_z.append(np.real(np.sum(dipole[2:len(dipole):3]))),
        NP_abs_x.append(np.imag(np.sum(dipole[0:len(dipole):3]))),
        NP_abs_y.append(np.imag(np.sum(dipole[1:len(dipole):3]))),
        NP_abs_z.append(np.imag(np.sum(dipole[2:len(dipole):3])))
    )
```

The following function is used for plotting the induced dipole moment of a molecule as a function of the frequency. Though it is not up to date as it has not been used since the rewriting of the programme.

```python
def dipole(freq, dipMOL_x, dipMOL_y, dipMOL_z, absMOL_x, absMOL_y, absMOL_z, xyz_filename,
           coordinates, output_path, out_filename):
    fig, (ax1, ax2, ax3) = plt.subplots(
        3, 1, sharex=True, sharey=False, figsize=(10, 6)
    )

    ax1.plot(W, dipMOL_x, 'b', label='Real')
    ax1.plot(W, absMOL_x, 'r', label='Imaginary')
    ax1.set_title("X component", fontsize=10)
    ax1.legend(loc=0, fontsize=10)

    ax2.plot(W, dipMOL_y, 'b')
    ax2.plot(W, absMOL_y, 'r')
    ax2.set_title("Y component", fontsize=10)
    ax2.set_ylabel(r"Induced dipole moment [a.u.]", fontsize=15)

    ax3.plot(W, dipMOL_z, 'b')
    ax3.plot(W, absMOL_z, 'r')
    ax3.set_title("Z component", fontsize=10)
    ax3.set_xlabel(r"External field frequency [a.u.]", fontsize=15)

    f = mticker.ScalarFormatter(useOffset=False, useMathText=True)

    def g(w, pos):
        return "${}$".format(f._formatSciNotation("%1.10e" % w))

    ax1.yaxis.set_major_formatter(mticker.FuncFormatter(g))
    ax2.yaxis.set_major_formatter(mticker.FuncFormatter(g))
    ax3.yaxis.set_major_formatter(mticker.FuncFormatter(g))

    plt.suptitle(
        "{0} \n {1} ({2} atoms)".format(out_filename.split('.', 1)[0],
                                        xyz_filename.split('.', 1)[0],
                                        len(coordinates)-1), fontsize=12
    )

    i = 0
    while os.path.exists(
        "{0}{1}_{2}.png".format(
            output_path, xyz_filename.split('.', 1)[0], i)
    ):
        i += 1

    plot_filename = "{0}{1}_{2}.png".format(
        output_path, xyz_filename.split('.', 1)[0], i)

    plt.savefig(plot_filename, bbox_inches='tight')
    plt.close("all")
```

The following function is up to date, as it used to plot the induced dipole moment of a nanoparticle as a function of the frequency.

```python
def NP_dipole(freq, NP_dip_x, NP_dip_y, NP_dip_z, NP_abs_x, NP_abs_y,
    NP_abs_z, xyz_file_name, coordinates, output_path, out_file_name):
    fig, (ax1, ax2, ax3) = plt.subplots(
        3, 1, sharex=True, sharey=False, figsize=(10, 6)
    )

    ax1.plot(freq, NP_dip_x, 'b', label='Real')
    ax1.plot(freq, NP_abs_x, 'r', label='Imaginary')
    ax1.set_title("X component", fontsize=10)
    ax1.legend(loc=0, fontsize=10)

    ax2.plot(freq, NP_dip_y, 'b')
    ax2.plot(freq, NP_abs_y, 'r')
    ax2.set_title("Y component", fontsize=10)
    ax2.set_ylabel(r"Induced dipole moment [a.u.]", fontsize=15)

    ax3.plot(freq, NP_dip_z, 'b')
    ax3.plot(freq, NP_abs_z, 'r')
    ax3.set_title("Z component", fontsize=10)
    ax3.set_xlabel(r"External field frequency [a.u.]", fontsize=15)

    f = mticker.ScalarFormatter(useOffset=False, useMathText=True)

    def g(w, pos):
        return "${}$".format(f._formatSciNotation("%1.10e" % w))

    ax1.yaxis.set_major_formatter(mticker.FuncFormatter(g))
    ax2.yaxis.set_major_formatter(mticker.FuncFormatter(g))
    ax3.yaxis.set_major_formatter(mticker.FuncFormatter(g))

    plt.suptitle(
        "{0} \n {1} ({2} atoms)".format(out_file_name.split('.', 1)[0],
                                        xyz_file_name.split('.', 1)[0],
                                        len(coordinates)), fontsize=12
    )

    i = 0
    while os.path.exists(
        "{0}{1}_NPdip_{2}.png".format(
            output_path, xyz_file_name.split('.', 1)[0], i)
    ):
        i += 1

    plot_file_name = "{0}{1}_NPdip_{2}.png".format(
        output_path, xyz_file_name.split('.', 1)[0], i)

    plt.savefig(plot_file_name, bbox_inches='tight')
    plt.close("all")
```

The function above returns a plot containing three separate graphs, containing the induced dipole moment in the x-direction, y-direction, and z-direction. Below is a function which outputs a single graphical plot of the average induced dipole moment as a function of the frequency.

```python
def avg_NP_dipole(freq, NP_dip_x, NP_dip_y, NP_dip_z, NP_abs_x, NP_abs_y,
    NP_abs_z, out_file_name, xyz_file_name, output_path, coordinates):
    avgr = (NP_dip_x + NP_dip_y + NP_dip_z)/3
    avgi = (NP_abs_x + NP_abs_y + NP_abs_z)/3

    fig, (ax1, ax2) = plt.subplots(
        2, 1, sharex=True, sharey=False, figsize=(10, 6)
    )

    ax1.plot(freq, avgr, 'b', label='Real')
    ax1.legend(loc=0, fontsize=10)

    ax2.plot(freq, avgi, 'r', label='Imaginary')
    ax2.legend(loc=0, fontsize=10)

    ax1.set_ylabel(r"Avg. induced dipole moment [a.u.]", fontsize=15)
    ax2.set_xlabel(r"External field frequency [a.u.]", fontsize=15)

    f = mticker.ScalarFormatter(useOffset=False, useMathText=True)

    def g(w, pos):
        return "${}$".format(f._formatSciNotation("%1.10e" % w))

    ax1.yaxis.set_major_formatter(mticker.FuncFormatter(g))
    ax2.yaxis.set_major_formatter(mticker.FuncFormatter(g))

    plt.suptitle(
        "{0} \n {1} ({2} atoms)".format(out_file_name.split('.', 1)[0],
                                        xyz_file_name.split('.', 1)[0],
                                        len(coordinates)), fontsize=12
    )

    i = 0
    while os.path.exists(
        "{0}{1}_avgNPdip_{2}.png".format(
            output_path, xyz_file_name.split('.', 1)[0], i)
    ):
        i += 1

    plot_filename = "{0}{1}_avgNPdip_{2}.png".format(
        output_path, xyz_file_name.split('.', 1)[0], i)

    plt.savefig(plot_filename, bbox_inches='tight')
    plt.close("all")
```

The following function, as well as the one after, is only used when the mode is set to 3, meaning the calculations are done with polarizabilities read from a .out file. The function below plots the polarizability of the nanoparticle as a function of the frequency.

```python
def NP_pol(freq, aNP_xx, aNP_yy, aNP_zz, out_file_name, xyz_file_name,
    output_path, coordinates):
    fig, (ax1, ax2, ax3) = plt.subplots(
        3, 1, sharex=True, sharey=False, figsize=(10, 6)
    )

    ax1.plot(freq, np.real(aNP_xx), 'b', label='Real')
    ax1.plot(freq, np.imag(aNP_xx), 'r', label='Imaginary')
    ax1.set_title("X component", fontsize=10)
    ax1.legend(loc=0, fontsize=10)

    ax2.plot(freq, np.real(aNP_yy), 'b')
    ax2.plot(freq, np.imag(aNP_yy), 'r')
    ax2.set_title("Y component", fontsize=10)
    ax2.set_ylabel(r"Polarizability [a.u.]", fontsize=15)

    ax3.plot(freq, np.real(aNP_zz), 'b')
    ax3.plot(freq, np.imag(aNP_zz), 'r')
    ax3.set_title("Z component", fontsize=10)
    ax3.set_xlabel(r"External field frequency [a.u.]", fontsize=15)

    f = mticker.ScalarFormatter(useOffset=False, useMathText=True)

    def g(w, pos):
        return "${}$".format(f._formatSciNotation("%1.10e" % w))

    ax1.yaxis.set_major_formatter(mticker.FuncFormatter(g))
    ax2.yaxis.set_major_formatter(mticker.FuncFormatter(g))
    ax3.yaxis.set_major_formatter(mticker.FuncFormatter(g))

    plt.suptitle(
        "{0} \n {1} ({2} atoms)".format(out_file_name.split('.', 1)[0],
                                        xyz_file_name.split('.', 1)[0],
                                        len(coordinates)), fontsize=12
    )

    i = 0
    while os.path.exists(
        "{0}{1}_pol_{2}.png".format(
            output_path, xyz_file_name.split('.', 1)[0], i)
    ):
        i += 1

    plot_file_name = "{0}{1}_pol_{2}.png".format(
        output_path, xyz_file_name.split('.', 1)[0], i)

    plt.savefig(plot_file_name, bbox_inches='tight')
    plt.close("all")
```

The function which plots the average polarizability as a function of the frequency, for the nanoparticle, is given as

```python
def avg_NP_pol(aNP_xx, aNP_yy, aNP_zz, freq, out_file_name, xyz_file_name,
               output_path, coordinates):
    avgr = (np.real(aNP_xx) + np.real(aNP_yy) + np.real(aNP_zz))/3
    avgi = (np.imag(aNP_xx) + np.imag(aNP_yy) + np.imag(aNP_zz))/3

    fig, (ax1) = plt.subplots(
        1, 1, sharex=True, sharey=False, figsize=(10, 6)
    )

    ax1.plot(freq, avgr, 'b', label='Real')
    ax1.plot(freq, avgi, 'r', label='Imaginary')
    ax1.set_title("X component", fontsize=10)
    ax1.legend(loc=0, fontsize=10)

    ax1.set_ylabel(r"Avg. polarizability [a.u.]", fontsize=15)
    ax1.set_xlabel(r"External field frequency [a.u.]", fontsize=15)

    f = mticker.ScalarFormatter(useOffset=False, useMathText=True)

    def g(w, pos):
        return "${}$".format(f._formatSciNotation("%1.10e" % w))

    ax1.yaxis.set_major_formatter(mticker.FuncFormatter(g))

    plt.suptitle(
        "{0} \n {1} ({2} atoms)".format(out_file_name.split('.', 1)[0],
                                        xyz_file_name.split('.', 1)[0],
                                        len(coordinates)), fontsize=12
    )

    i = 0
    while os.path.exists(
        "{0}{1}_avgpol_{2}.png".format(
            output_path, xyz_file_name.split('.', 1)[0], i)
    ):
        i += 1

    plot_file_name = "{0}{1}_avgpol_{2}.png".format(
        output_path, xyz_file_name.split('.', 1)[0], i)

    plt.savefig(plot_file_name, bbox_inches='tight')
    plt.close("all")
```

If the mode is set to 1, the following function is used to plot the polarizability of the nanoparticle as a function of the frequency. As of now, there is no function for plotting the average polarizability when running the programme in mode 1, meaning when the polarizabilities are calculated within the programme itself.

```python
def NP_polarizability(aNP_list, freq, out_file_name,
                      xyz_file_name, output_path, coordinates):
    fig, (ax1) = plt.subplots(
        1, 1, sharex=True, sharey=False, figsize=(10, 6)
    )

    ax1.plot(freq, np.real(aNP_list), 'b', label='Real')
    ax1.plot(freq, np.imag(aNP_list), 'r', label='Imaginary')
    ax1.set_title("X component", fontsize=10)
    ax1.legend(loc=0, fontsize=10)

    ax1.set_ylabel(r"Avg. polarizability [a.u.]", fontsize=15)
    ax1.set_xlabel(r"External field frequency [a.u.]", fontsize=15)

    f = mticker.ScalarFormatter(useOffset=False, useMathText=True)

    def g(w, pos):
        return "${}$".format(f._formatSciNotation("%1.10e" % w))

    ax1.yaxis.set_major_formatter(mticker.FuncFormatter(g))

    plt.suptitle(
        "{0} \n {1} ({2} atoms)".format(out_file_name.split('.', 1)[0],
                                        xyz_file_name.split('.', 1)[0],
                                        len(coordinates)), fontsize=12
    )

    i = 0
    while os.path.exists(
        "{0}{1}_Cpol_{2}.png".format(
            output_path, xyz_file_name.split('.', 1)[0], i)
    ):
        i += 1

    plot_file_name = "{0}{1}_Cpol_{2}.png".format(
        output_path, xyz_file_name.split('.', 1)[0], i)

    plt.savefig(plot_file_name, bbox_inches='tight')
    plt.close("all")
```

Main script

The main script is, as the name indicates, the main file in which all the different functions from `zdimpy` are executed in the right order, and tied together. In the initial part of the script all the needed libraries are imported, followed by the `argparse` setup. In order to time the programme, a start time is initiated at the beginning, right before any of the main code is executed.

```
start_time = time.time()
```

Before any calculations can be carried out, the coordinates of the system has to be retrieved from the .xyz file, as

```
coordinates, x_coordinates, y_coordinates, z_coordinates = f.xyz(
    args.xyz_file_path
)
```

followed by the import of the frequency values from the .out file.

```
if args.mode == 1:
    freq = f.freq(args.out_file_path)
```

In case of the polarrizabilities needs to be read from the .out file, the following lines of code are executed rather than those above.

```
if args.mode == 3:
    freq, aNP_xx, aNP_yy, aNP_zz = f.out_NP(args.out_file_path)
```

If the mode is set to 3, the frequencies and polarizabilities are printed as

```
if args.mode == 3:
    sh.header(
        freq,
        os.path.basename(args.xyz_file_path),
        os.path.basename(args.out_file_path),
        coordinates,
        aNP_xx,
        aNP_yy,
        aNP_zz
    )
```

If the polarizabilities are not read from a .out file, the parameters needed for the calculation of the polarizabilites are initiated.

```
if args.mode == 1:
    pl_freq, ex_freq, pl_str, ex_lftm, stat_pol = calc.pol_par(
        os.path.basename(args.xyz_file_path)
    )
```

The function which calculates the spatial distance between each atom, as well as the spatial distance from the origin to each atom, is executed

```
p_dist, o_dist = calc.spatial_dist(
    coordinates
)
```

The function which calculates the coordinate difference is executed

```
x_diff, y_diff, z_diff = calc.point_diff(
    coordinates
)
```

Using the coordinate differences which were calculated above, the dipole-dipole interaction tensor components are computed

```
T_xx, T_yy, T_zz, T_xy, T_xz, T_yz = calc.T(
    x_diff,
    y_diff,
    z_diff,
    p_dist
)
```

where after they are stacked, in order to form the basis of the A matrix.

```
temp_A = calc.tensor_stack(
    [
        T_xx,
        T_xy,
        T_xz,
        T_xy,
        T_yy,
        T_yz,
        T_xz,
        T_xy,
        T_zz
    ]
)
```

The dipole-dipole interaction tensors are converted to negative complex numbers.

```
temp_A = calc.tensor_gentrificator(temp_A)
```

Empty lists for storing the results are initiated as

```
if args.mode == 1 or args.mode == 3:
    NP_dip_x = []
    NP_dip_y = []
    NP_dip_z = []
    NP_abs_x = []
    NP_abs_y = []
    NP_abs_z = []

    if args.mode == 1:
        aNP_list = []
```

In order to run the calculations for each frequency, the calculations are run within a loop, which loops through the list of frequencies

```
for i, frequency in enumerate(freq):
```

The frequency value is printed to the terminal.

```
sh.iteration(freq, i)
```

If the polarizabilities are not imported from a .out file, the polarizabilities are calculated and placed as the diagonal in the stacked array of the dipole-dipole tensors.
quiet

```
if args.mode == 1:
        A, aNP = calc.nano_pol(
            temp_A,
            pl_freq,
            ex_freq,
            pl_str,
            ex_lftm,
            stat_pol,
            freq,
            i
        )
        aNP_list.append(aNP)
```

If the polarizabilities are imported from a .out file, they are placed as the diagonal of the diagonal in the stacked array of the dipole-dipole tensors.

```
if args.mode == 3:
        A = calc.nano_diag(
            aNP_xx,
            aNP_yy,
            aNP_zz,
            temp_A,
            i
        )
```

Both of the above functions returns an inverted version of the stacked array containing the polarizabilities as the diagonal and dipole-dipole interaction tensors. As the induced dipole moment is needed for the calculation of the permanent external field, a set of temporary dipole variables are create, to do the initial iteration, whereafter the these variables are overwritten with the newfound values.

```
dipole_x = np.full((len(coordinates), 1), 1 + 0.j)
dipole_y = np.full((len(coordinates), 1), 1 + 0.j)
dipole_z = np.full((len(coordinates), 1), 1 + 0.j)
```

A counter is initiated before a while loop within the for loop is initiated. The for loop loops over each index in the frequency list, whereas the while loop loops the calculation of the induced dipole moment for each iteration of the for loop. Meaning, that the for loop runs as long as there are more frequencies, while the while loop runs as long as the value of the dipole moments used for the calculations, are different than those returned as a results of the calculations.

```
counter = 0
```

The while loop is initiated as an infinite loop (The loop will always be ture), and is eventually broken by an if statement comparing the initial dipole moment values with those returned at the end.

```
while True:
```

The counter is incremented by 1 for each iteration.

```
counter += 1
```

The permanent electrical field is calculated as

```
E = calc.E(o_dist, args.E_external, dipole_x, dipole_y, dipole_z,
                    coordinates, x_coordinates, y_coordinates, z_coordinates)
```

Finally, the induced dipole moment is computed as

```
dipole = np.dot(A, E)
```

A set of temporary variables are initiated. These are for checking whether there is a difference between the values used for the calculations and those which were just returned.

```
check_x = dipole[0:len(dipole):3]
check_y = dipole[1:len(dipole):3]
check_z = dipole[2:len(dipole):3]
```

If the returned values for the induced dipole moment are equal to the values used for the calculations, the following if statement triggers.

```
if (np.array_equal(dipole_x, check_x) == True
    and np.array_equal(dipole_y, check_y) == True
    and np.array_equal(dipole_z, check_z) == True
        or counter > 999):
```

If a nanoparticle is present in the system, the real and imaginary part of the returned dipole moment values, i.e. the induced dipole moment and the corresponding absorption.

```
if args.mode == 1 or args.mode == 3:
                sh.NP_dipole(counter, dipole)

                plot.NP_dipole_append(
                    NP_dip_x,
                    NP_dip_y,
                    NP_dip_z,
                    NP_abs_x,
                    NP_abs_y,
                    NP_abs_z,
                    dipole
                )
```

The otherwise infinite while loop is then terminated.

```
break
```

In the case where the if statement has not been satisfied, the dipole moment variables used in the calculations are overwritten by those just returned, whereafter the while loop goes through another iteration with the new values.

```
dipole_x = dipole[0:len(dipole):3]
dipole_y = dipole[1:len(dipole):3]
dipole_z = dipole[2:len(dipole):3]
```

If a nanoparticle is present in the system, the maximum values of the induced dipole moments and the minimum of the absorption are printed. These are usable since the maximum of the real values an the minimum of the imaginary ones, are where a transition will happen.

```
if args.mode == 1 or args.mode == 3:
    sh.min_max(freq, NP_abs_x, NP_abs_y, NP_abs_z, NP_dip_x, NP_dip_y,
        NP_dip_z)
```

As the quiet option should only return something in the terminal, and not create any plots, the following code is only executed if the quiet argument is false.

```
if not args.quiet == True:
```

If the quiet argument has not been used, the induced dipole moment values for the nanoparticle are converted into NumPy arrays.

```
if args.mode == 1 or args.mode == 3:
        NP_dip_x = np.asarray(NP_dip_x)
        NP_dip_y = np.asarray(NP_dip_y)
        NP_dip_z = np.asarray(NP_dip_z)
        NP_abs_x = np.asarray(NP_abs_x)
        NP_abs_y = np.asarray(NP_abs_y)
        NP_abs_z = np.asarray(NP_abs_z)
```

The arrays are then used for plotting the nanoparticles induced dipole moment as

```
plot.NP_dipole(
            freq,
            NP_dip_x,
            NP_dip_y,
            NP_dip_z,
            NP_abs_x,
            NP_abs_y,
            NP_abs_z,
            os.path.basename(args.xyz_file_path),
            coordinates,
            args.output_path,
            os.path.basename(args.out_file_path)
        )
```

If the verbose argument has been used, the following code will be executed.

```
if args.verbose == True:
```

The average induced dipole moment for the nanoparticle will be plotted as a function of the frequency.

```
        plot.avg_NP_dipole(
            freq,
            NP_dip_x,
            NP_dip_y,
            NP_dip_z,
            NP_abs_x,
            NP_abs_y,
            NP_abs_z,
            os.path.basename(args.out_file_path),
            os.path.basename(args.xyz_file_path),
            args.output_path,
            coordinates
        )
```

As those polarizabilities which are calculated by the programme itself differ from those imported from an .out file, in terms of wether they are isotropic or not, the way they are plotted differ. The polarizabilities calculated by the programme itself are isotropic where as those imported from an .out file are not. This means that the lists containing the values differ, as those read from an .out file are separated into xx-, yy-, and zz-components.

```
if args.mode == 3:
```

In the case where the polarizabilities have been imported from an .out file, the following function is used to plot the polarizabilities as well as the average polarizabilities.

```
plot.NP_pol(
                freq,
                aNP_xx,
                aNP_yy,
                aNP_zz,
                os.path.basename(args.out_file_path),
                os.path.basename(args.xyz_file_path),
                args.output_path,
                coordinates
            )
            plot.avg_NP_pol(
                aNP_xx,
                aNP_yy,
                aNP_zz,
                freq,
                os.path.basename(args.out_file_path),
                os.path.basename(args.xyz_file_path),
                args.output_path,
                coordinates
            )
```

Otherwise, if the polarizabilities are computed by the programme, the following code is used to plot the polarizabilities for the nanoparticle.

```
elif args.mode == 1:
            aNP_list = np.asarray(aNP_list)

            plot.NP_polarizability(
                aNP_list,
                freq,
                os.path.basename(args.out_file_path),
                os.path.basename(args.xyz_file_path),
                args.output_path,
                coordinates
            )
```

As the very last thing to be executed, the elapsed time and used memory are printed to the terminal.

```
sh.misc(start_time)
```