Assignment 3 (Data Estimation)

This notebook describes a method to estimate the data given in the assignment.

Author: Kaushik G Iyer (EE23B135)

Instructions to run:

Just like run the cells one by one. All of the cells have a description of what is being done just above it :)

Imports

This notebook uses numpy and scipy to do the desired calculations. It uses matplotlib to plot the data.

Run `pip install numpy scipy matplotlib` to install the required libraries.

In [8]:
```
from scipy.optimize import curve_fit  # type: ignore
import matplotlib.pyplot as plt
import numpy as np
```

Planck's Law

In physics, Planck's law describes the spectral density of electromagnetic radiation emitted by a black body in thermal equilibrium at a given temperature T, when there is no net flow of matter or energy between the body and its environment.

It states that:

$$I(\lambda, T) = \frac{2hc^2}{\lambda^5} \frac{1}{\exp\left(\frac{hc}{k_B T \lambda}\right) - 1}$$

Where

- $I(\lambda, T)$ is the spectral radiance
- λ is the wavelength
- T is the absolute temperature of the body
- k_B is the Boltzmann constant with a value of $1.38 * 10^{-23}\ JK^{-1}$
- h is the Planck constant with a value of $6.626 * 10^{-34}\ J.s$
- c is the speed of light in the medium. In air it has a value of $2.99 * 10^8\ m.s^{-1}$

In [9]:
```
def spectral_radiance(
    lam: "float | np.ndarray[np.float64, np.dtype[np.float64]]",
    T: float,
    kB: float,
    h: float,
    c: float,
):
    """
    Function that calculates the spectral radiance as per the formula described above
    """
    return 2 * h * c**2 / (lam**5 * (np.exp(h * c / (kB * T * lam)) - 1))
```

Loading datasets

Loads data from every file mentioned in `DATA_FILES`

NOTE: Every data file is expected to have lines with 2 **comma seperated** floats

If the file doesn't exist or if the file couldn't be parsed, an exception will be raised.

In [10]:
```
DATA_FILES = ["d1.txt", "d2.txt", "d3.txt", "d4.txt"]

Coordinates = tuple[
    np.ndarray[np.float64, np.dtype[np.float64]],
    np.ndarray[np.float64, np.dtype[np.float64]],
]
data: list[Coordinates] = []  # Stores the points for every file

for file in DATA_FILES:
    xords: list[float] = []
    yords: list[float] = []
    try:
        with open(file) as f:
            for line in f.readlines():
                a, _, b = line.partition(",")
                xords.append(float(a.strip()))
                yords.append(float(b.strip()))

    except Exception:
        raise RuntimeError(f"ERROR! Could not handle file `{file}`")

    data.append((np.array(xords), np.array(yords)))
```

Plotting results

The cell below contains a function that allows us to visualize our estimates

In [11]:
```
def plot(estimates: list[list[float]]):
    for i in range(len(data)):
        # Plot the data in the file
        plt.plot(*data[i], label=DATA_FILES[i], color="pink")  # type: ignore

        estimated_function = spectral_radiance(data[i][0], *estimates[i])

        print("Estimates for", DATA_FILES[i])
        print(
            f"T: {estimates[i][0]}, \
              kB: {estimates[i][1]}, h: {estimates[i][2]}, c: {estimates[i][3]}"
        )

        # Plot the estimated function
        plt.plot(data[i][0], estimated_function, label="Fitted Radiance", color="red")  # type: ignore

        # Add labels and a legend
        plt.legend()  # type: ignore
        plt.title("Estimates for " + DATA_FILES[i])  # type: ignore
        plt.xlabel("Wavelength")  # type: ignore
        plt.ylabel("Spectral Radiance")  # type: ignore

        plt.show()  # type: ignore
```

Estimating parameters using `scipy.optimize.curve_fit`

Without using partial application

Over here I ask curve_fit to provide all the values at once.

Note that this has some inherent problems:

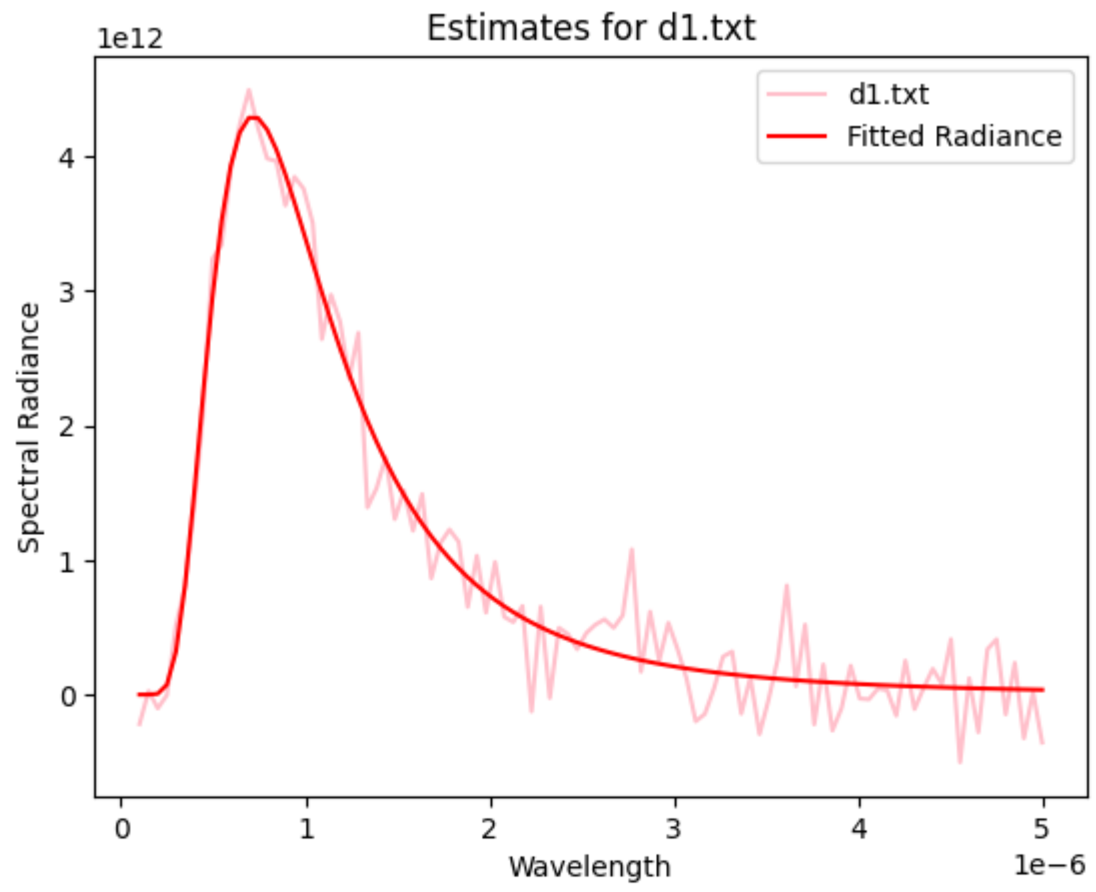
- The formula is symmetric over T and kB therefore it is simply impossible for scipy to provide an value of either
- The initial guess has to be good otherwise we either get overflow errors or scipy won't be able to estimate the parameters

Notice that even though the function estimated seems to look like a good fit, the parameter values aren't what we expect them to be.

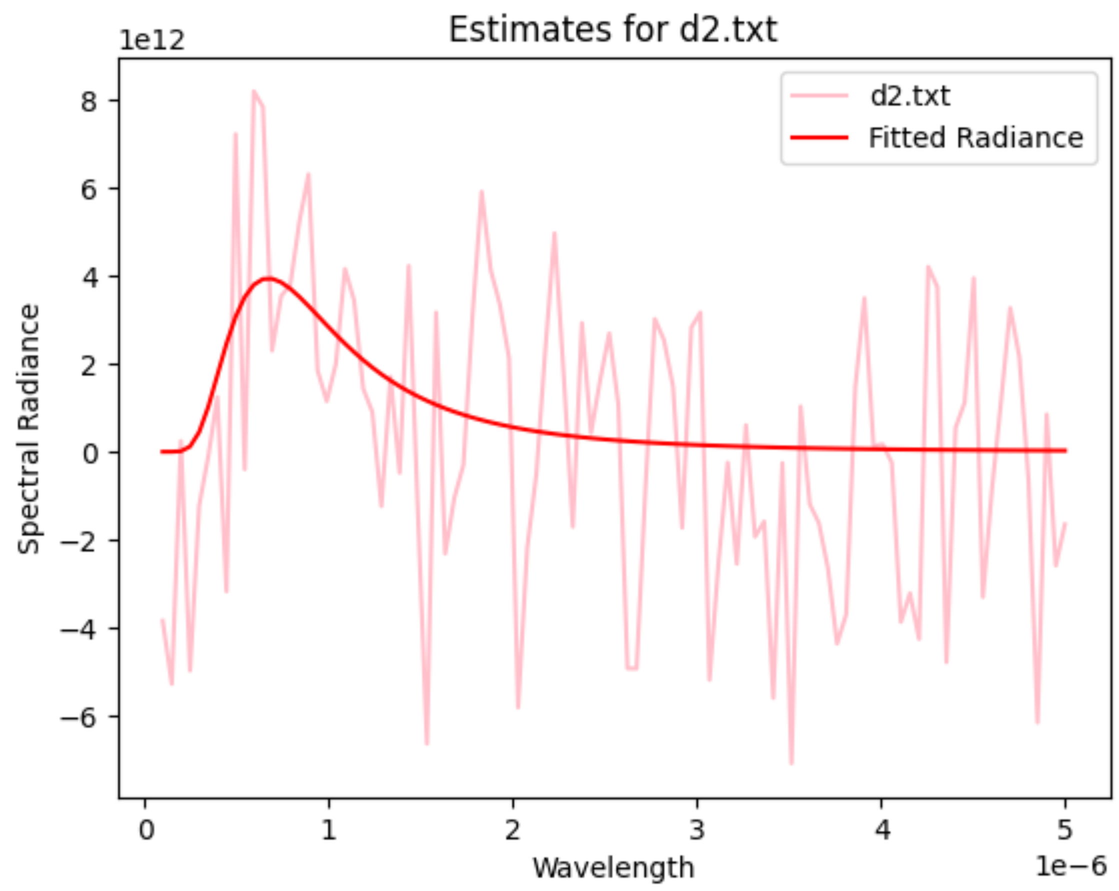
Also note that even though the initial guess for c (true for all parameters) is very accurate the estimate for it is actually worse. I suspect that this is simply because the data provided is very noisy

In [12]:
```
# Some random guesses for T, kB,  h, c (for each dataset)
# I've made all the initial estimates be the same for all datasets
# (Just because I'm lazy)
# But feel free to provide custom initial estimates for each of them :)
rough_guess = [[4e3, 2e-23, 6e-34, 3e8] for _ in range(len(data))]

# First we try to get the rough estimates without using partial application
rough_estimates: list[list[float]] = [
    curve_fit(spectral_radiance, *data[i], rough_guess[i])[0] for i in range(len(data))
]

plot(rough_estimates)
```

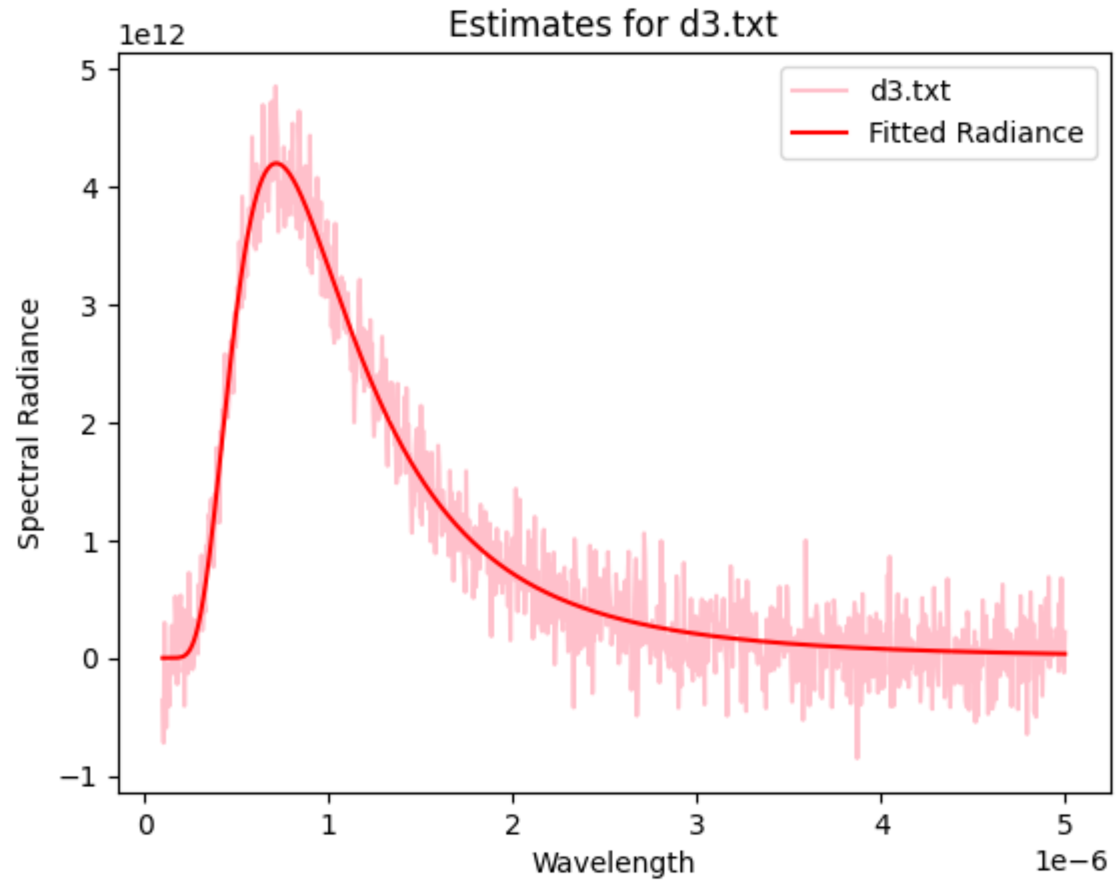
```
Estimates for d1.txt
T: 5023.72291652658,               kB: 2.0862278624343867e-23, h: 2.4015981492279797e-33, c: 155345214.41019705
```



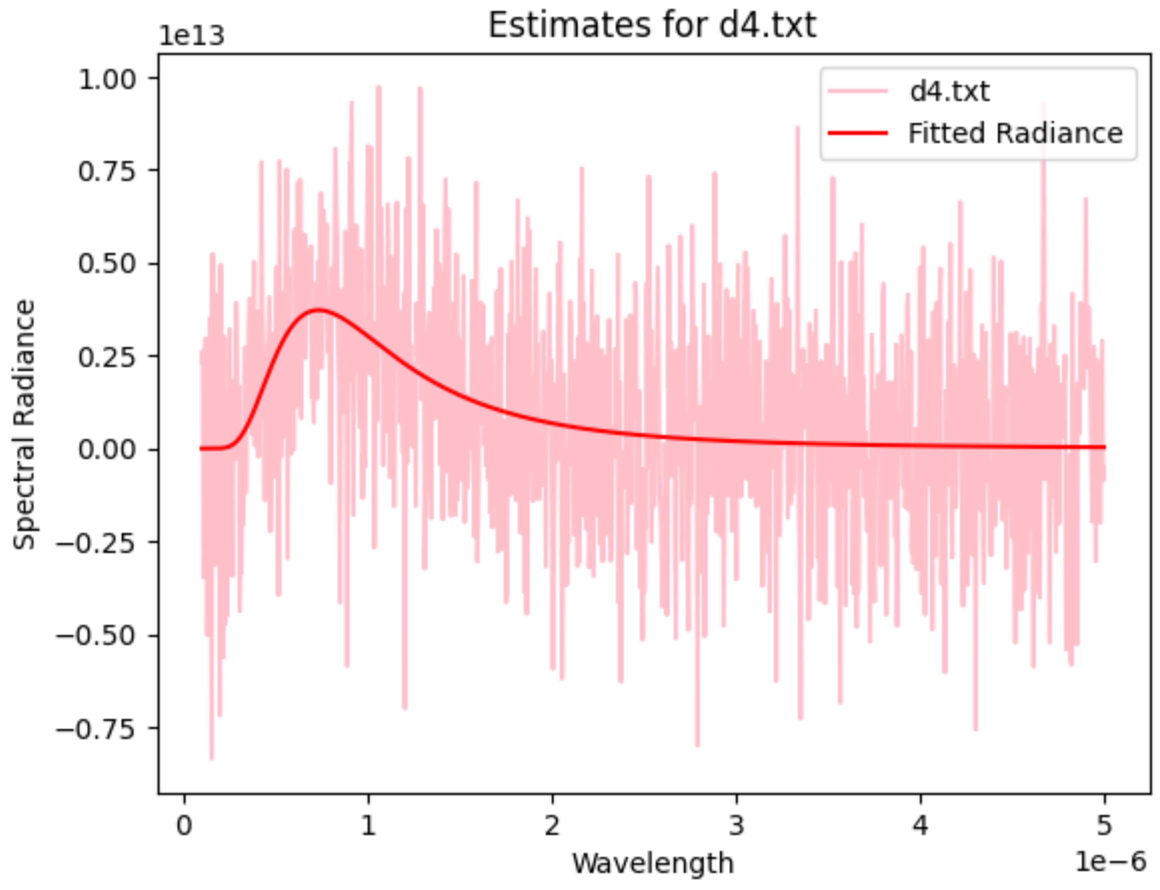
```
Estimates for d2.txt
T: 2396.460417456778,             kB: 1.284164307811824e-23, h: 2.7394480064891512e-34, c: 374999613.2471093
```



```
Estimates for d3.txt
T: 5053.557043508265,             kB: 2.0514325415489850e-23, h: 2.4083226000312106e-33, c: 153190013.22597837
```



```
Estimates for d4.txt
T: 3819.016241427982,             kB: 2.276394723643868e-23, h: 1.7808978730534827e-33, c: 177859994.9626566
```



Partial application

As seen in the plots above, even though the graphs look fine the estimates of the constants are off by a bit.

One way to get around this issue is using the partial application method.

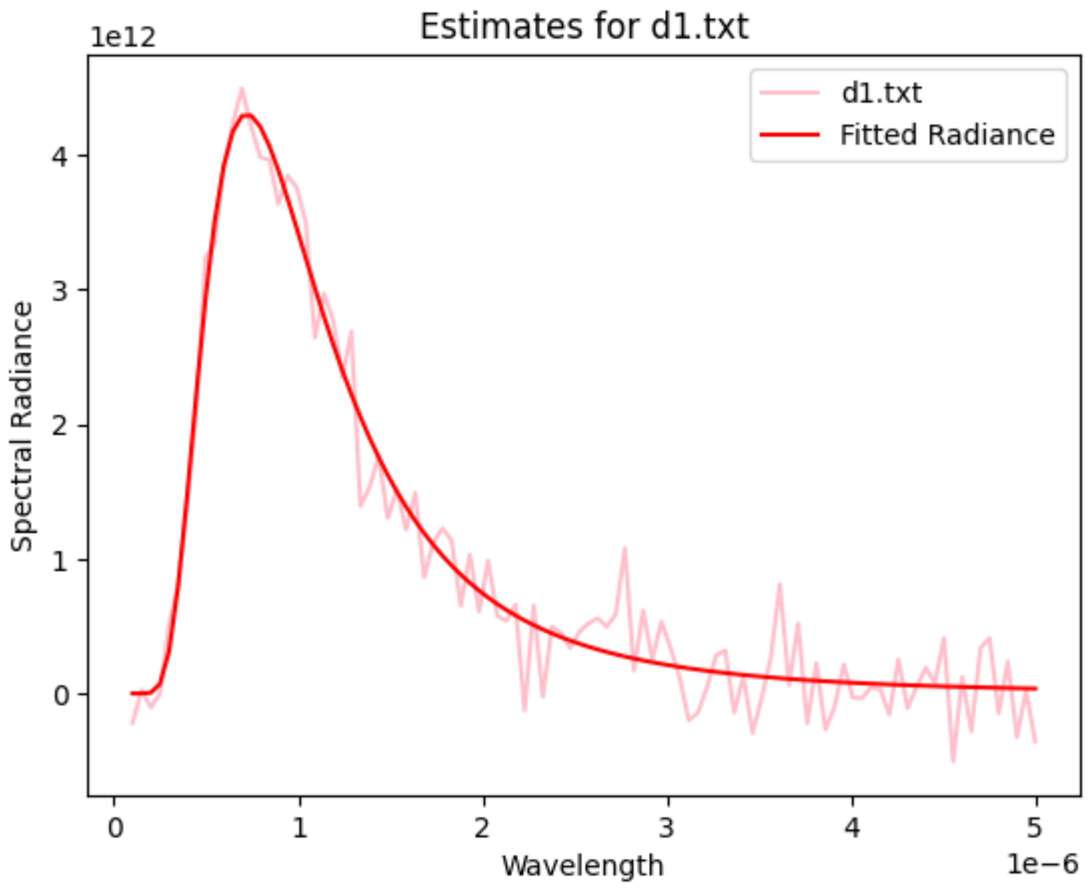
The idea behind this method is as follows:

- The main reason the previous method didn't work is that we had too many unknowns
- In this method we fix some of the variables (Perhaps they were found experimentally by other means ~~i.e. cheating~~) and estimate the values of parameters one at a time.
- We may then use these parameters to find our final estimate

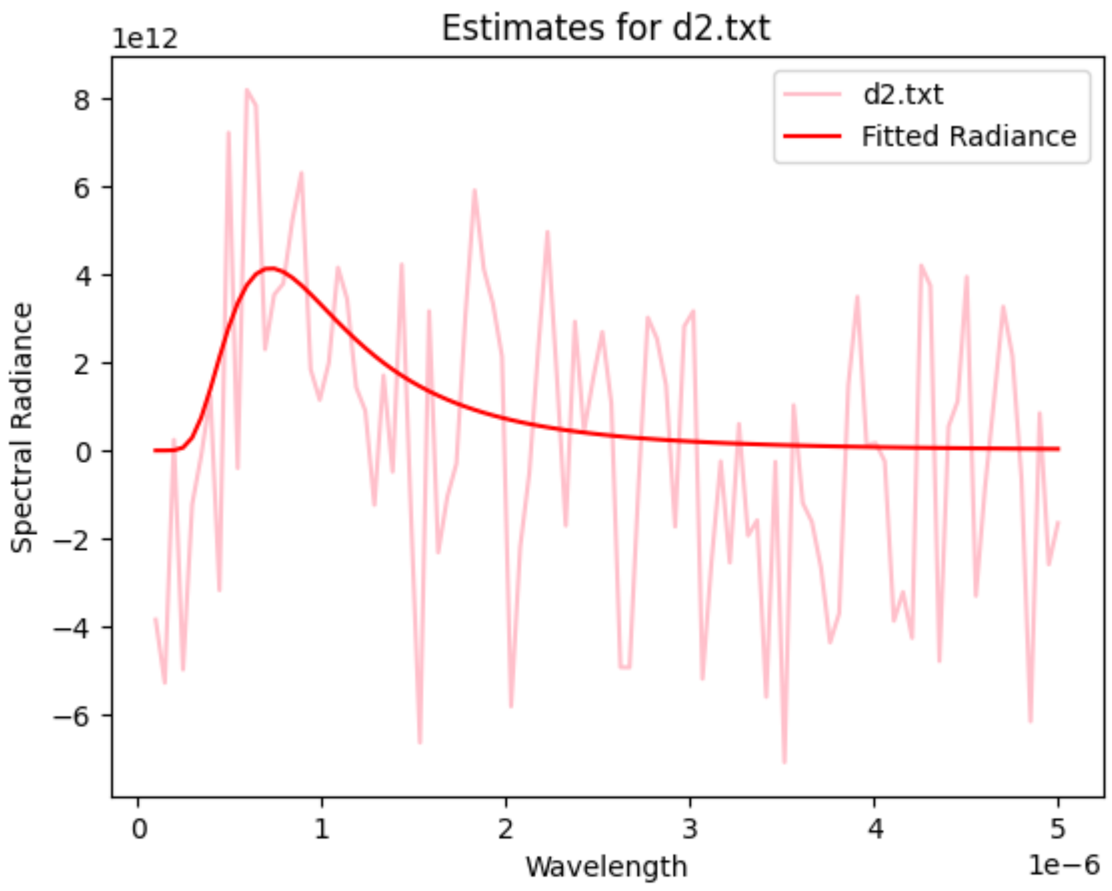
As you can see below the estimates are a lot better :)

In [13]:
```python
# Some random guesses for T, kB,  h, c (for each dataset)
# I've made all the initial estimates be the same for all datasets (Just because I'm lazy)
# But feel free to provide custom initial estimates for each of them :)
rough_guess = [[4e3, 2e-23, 6e-34, 3e8] for _ in range(len(data))]

# Fix kB, h and c inorder to find T
T_estimates: list[float] = [
    curve_fit(
        lambda lam, T: spectral_radiance(lam=lam, T=T, kB=1.68e-23, h=6.626e-34, c=2.99e8),  # type: ignore
        *data[i],
        [rough_guess[i][0]]
    )[0]
    for i in range(len(data))
]

# Fix T, h and c inorder to find kB
# NOTE: We use the previously estimated value of T here
kB_estimates: list[float] = [
    curve_fit(
        lambda lam, kB: spectral_radiance(lam=lam, T=T_estimates[i], kB=kB, h=6.626e-34, c=2.99e8),  # type: ignore
        *data[i],
        [rough_guess[i][1]]
    )[0]
    for i in range(len(data))
]

# Fix T, kB and c inorder to find h
# NOTE: We use the previously estimated value of T and kB here
h_estimates: list[float] = [
    curve_fit(
        lambda lam, h: spectral_radiance(lam=lam, T=T_estimates[i], kB=kB_estimates[i], h=h, c=2.99e8),  # type: ignore
        *data[i],
        [rough_guess[i][2]]
    )[0]
    for i in range(len(data))
]

# Fix T, kB and h inorder to find c
# NOTE: We use the previously estimated value of T, kB and h here
c_estimates: list[float] = [
    curve_fit(
        lambda lam, c: spectral_radiance(lam=lam, T=T_estimates[i], kB=kB_estimates[i], h=h_estimates[i], c=c),  # type: ignor
        *data[i],
        [rough_guess[i][3]]
    )[0]
    for i in range(len(data))
]

# The final estimates
estimates = [
    [T_estimates[i], kB_estimates[i], h_estimates[i], c_estimates[i]]
    for i in range(len(data))
]

plot(estimates)
```

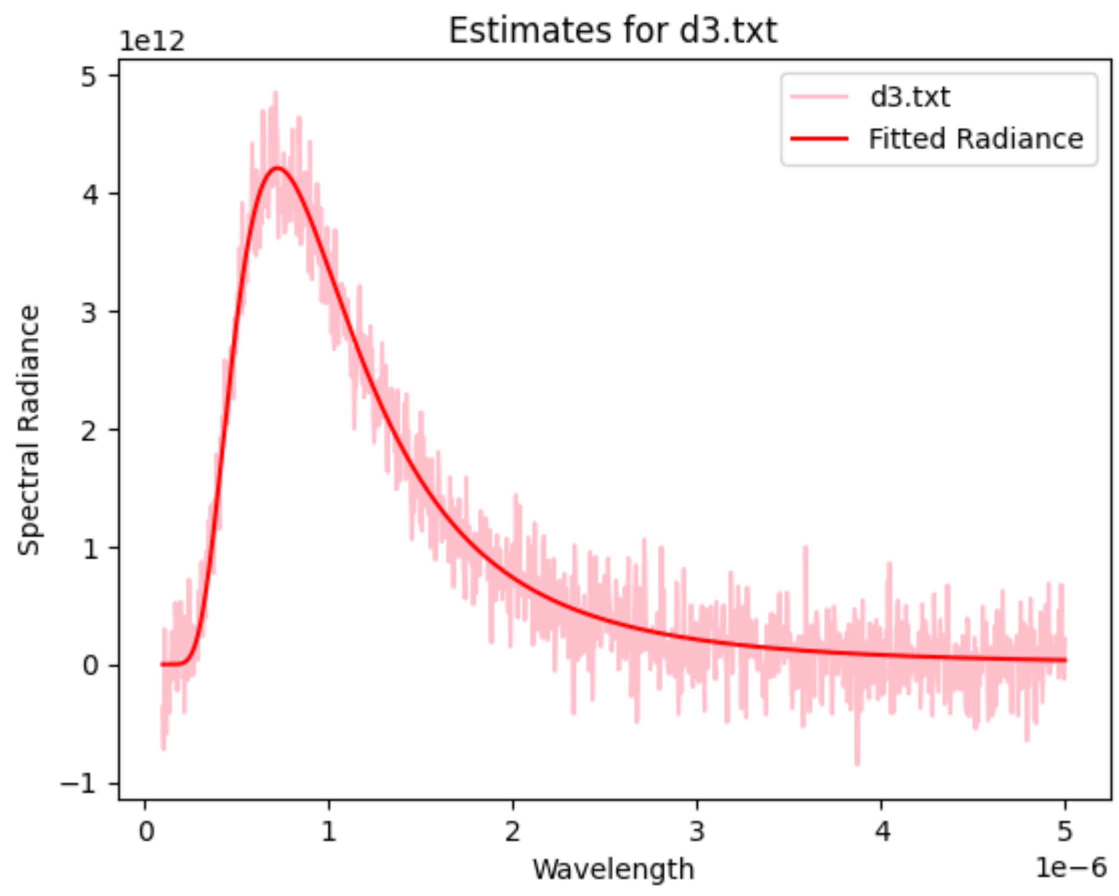
```
Estimates for d1.txt
T: [3298.28745506],              kB: [1.68e-23], h: [6.62504909e-34], c: [2.98908297e+08]
```



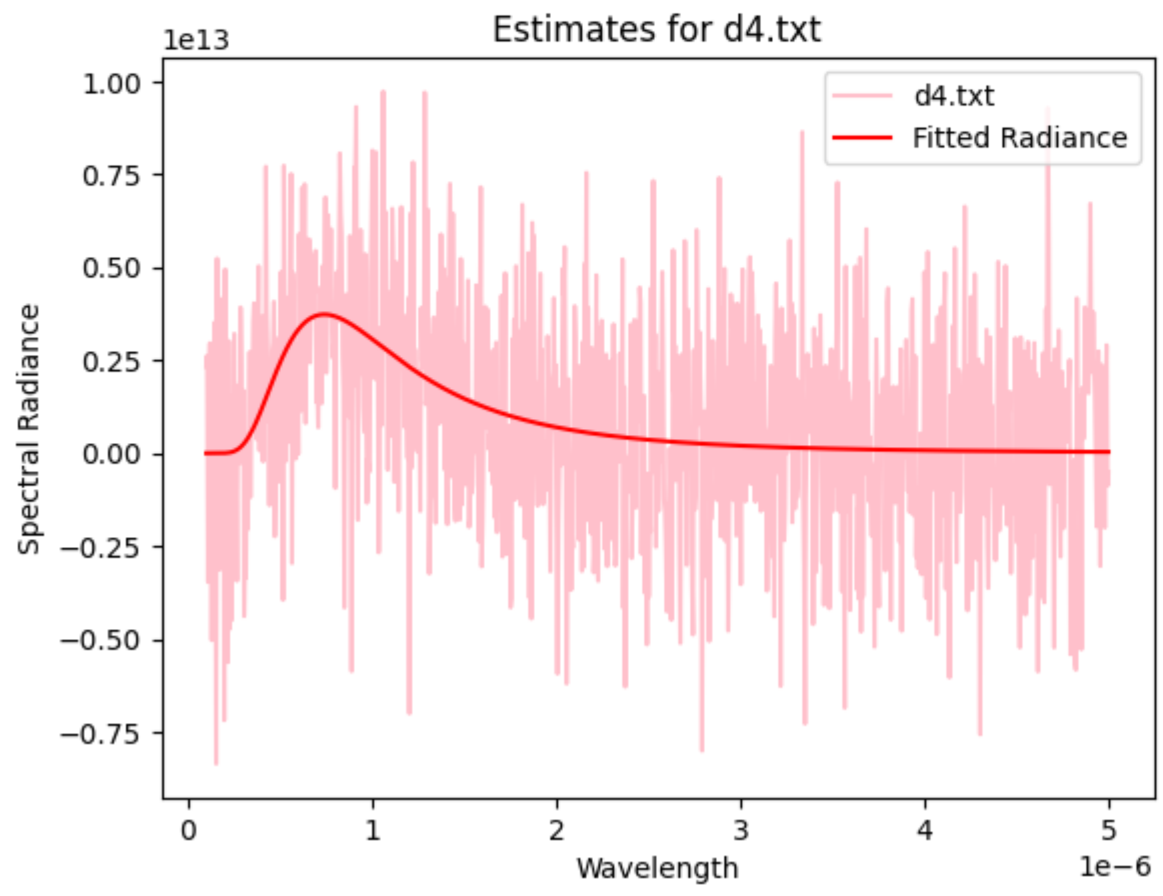
```
Estimates for d2.txt
T: [3246.53591477],              kB: [1.68000072e-23], h: [6.59850216e-34], c: [2.96534654e+08]
```



```
Estimates for d3.txt
T: [3282.46957445],              kB: [1.68e-23], h: [6.62396302e-34], c: [2.98803377e+08]
```



```
Estimates for d4.txt
T: [3203.82360241],             kB: [1.67999996e-23], h: [6.62392491e-34], c: [2.9879881e+08]
```



Comments ~~and Experimentation?~~

The data provided in dataset 2 (`d2.txt`) is very bad, it is evident in the fit created also (Notice how initally it doesn't really seem to fit well)

Perhaps one could try and do this partial application step iteratively (i.e After finding T, h, c, kB as shown in the above code, one could try and then use these values (instead of the true values) to generate another set of estimated values).

An iterative approach:

Logically the values can't get better after every iteration (as the constants stray further from their true value). But I suppose if I didn't know the true values (of h, c, and kB) and provided a very bad initial set of values, the values might get more accurate with more iterations. (I suspect this is what `curve_fit` does anyways when there are multiple unknowns)

Below I've given an example of how it might look. (You may play around with `ITERATION_COUNT` and the initial values of `estimates_iterative` to see what happens)

Although the estimates are worse than the estimates calculated in the cell above, the implementation below doesn't *cheat*. (It doesn't feel nice saying that you are estimating values after using their true values calculated by somebody else)

The cell below simply takes in some rough set of initial paramaters (similar to the implementation without partial application) but its results are much better than what the code without partial application returned. Although now that I'm looking at the results again, the approach below just gave estimates close to the inital estimates in most cases (Somehow h got more accurate in `d2.txt` ?!)

Partial partial application?

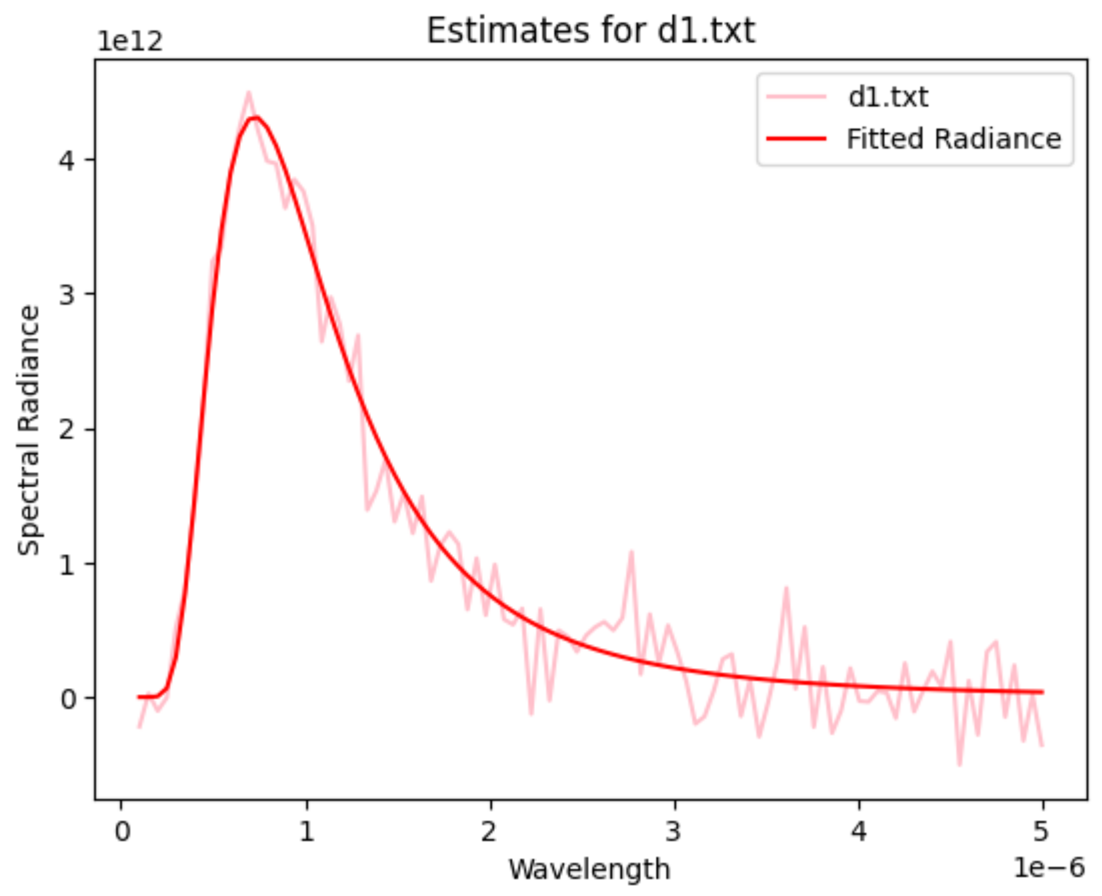
Another possible approach could be to fix kB and h to curve fit and find T and c. Or really any 2 of the parameters (Even 3!),to find the others. (Note that fixing h and c wouldn't work as the equation is symmetric wrt T and kB)

This sort of makes more sense (than just fixing everything) as now we can say that we actually estimated some value.

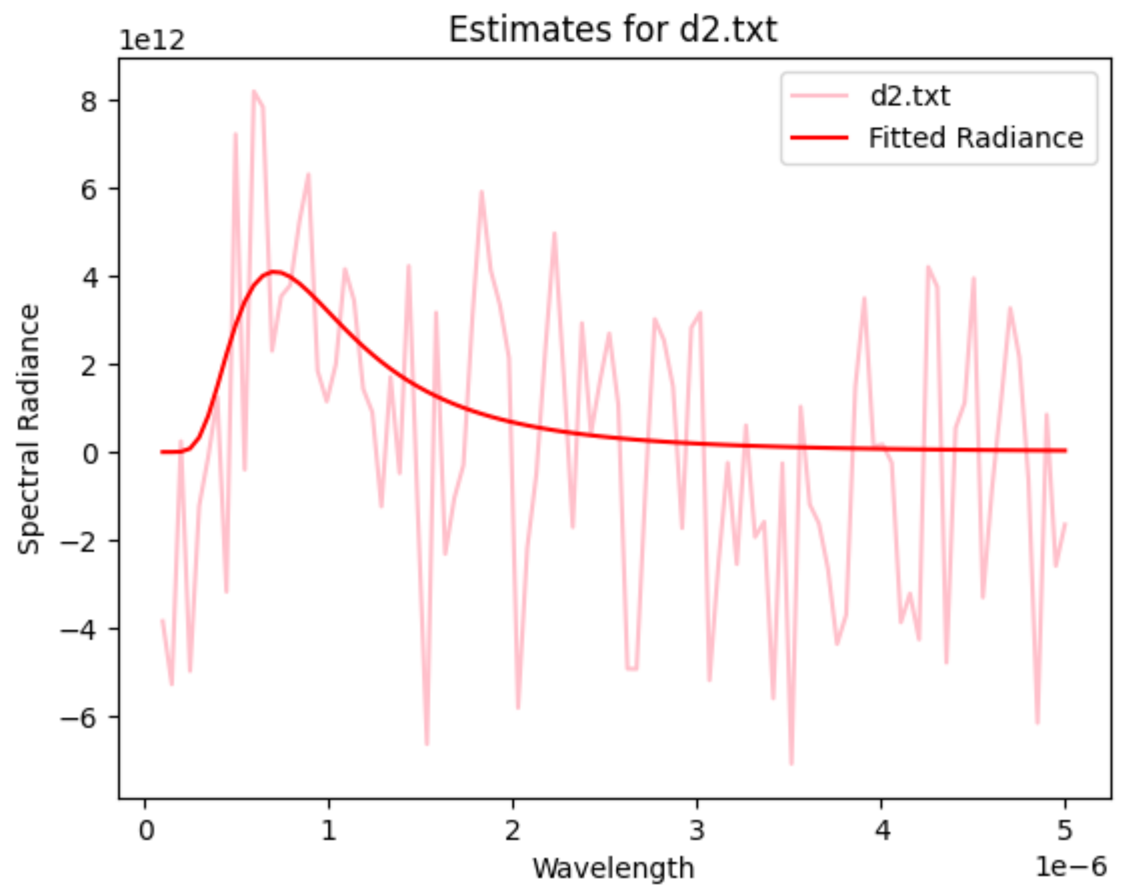
But since the iterative approach mentioned above seemed more interesting I've only played around with that. I shall leave this as an exercise to the reader :)

In [14]:
```
# Some random guesses for T, kB,  h, c (for each dataset)
# I've made all the initial estimates be the same for all datasets (Just because I'm lazy)
# But feel free to provide custom initial estimates for each of them :)
estimates_iterative = [[4e3, 2e-23, 7e-34, 3e8] for _ in range(len(data))]

# The number of times we do this partial application step
# (I've notice after like the 10th iteration it doesn't change as much)
ITERATION_COUNT = 10

for _ in range(ITERATION_COUNT):
    # Fix kB, h and c inorder to find T
    T_estimates_iterative: list[float] = [
        curve_fit(
            lambda lam, T: spectral_radiance(lam=lam, T=T, kB=estimates_iterative[i][1], h=estimates_iterative[i][2], c=estima
            *data[i],
            [estimates_iterative[i][0]]
        )[0]
        for i in range(len(data))
    ]

    # Fix T, h and c inorder to find kB
    # NOTE: We use the previously estimated value of T here
    kB_estimates_iterative: list[float] = [
        curve_fit(
            lambda lam, kB: spectral_radiance(lam=lam, T=T_estimates_iterative[i], kB=kB, h=estimates_iterative[i][2], c=estim
            *data[i],
            [estimates_iterative[i][1]]
        )[0]
        for i in range(len(data))
    ]

    # Fix T, kB and c inorder to find h
    # NOTE: We use the previously estimated value of T and kB here
    h_estimates_iterative: list[float] = [
        curve_fit(
            lambda lam, h: spectral_radiance(lam=lam, T=T_estimates_iterative[i], kB=kB_estimates_iterative[i], h=h, c=estimat
            *data[i],
            [estimates_iterative[i][2]]
        )[0]
        for i in range(len(data))
    ]

    # Fix T, kB and h inorder to find c
    # NOTE: We use the previously estimated value of T, kB and h here
    c_estimates_iterative: list[float] = [
        curve_fit(
            lambda lam, c: spectral_radiance(lam=lam, T=T_estimates_iterative[i], kB=kB_estimates_iterative[i], h=h_estimates_
            *data[i],
            [estimates_iterative[i][3]]
        )[0]
        for i in range(len(data))
    ]

    # The final estimates
    estimates_iterative = [
        [
            T_estimates_iterative[i],
            kB_estimates_iterative[i],
            h_estimates_iterative[i],
            c_estimates_iterative[i],
        ]
        for i in range(len(data))
    ]

plot(estimates_iterative)
```

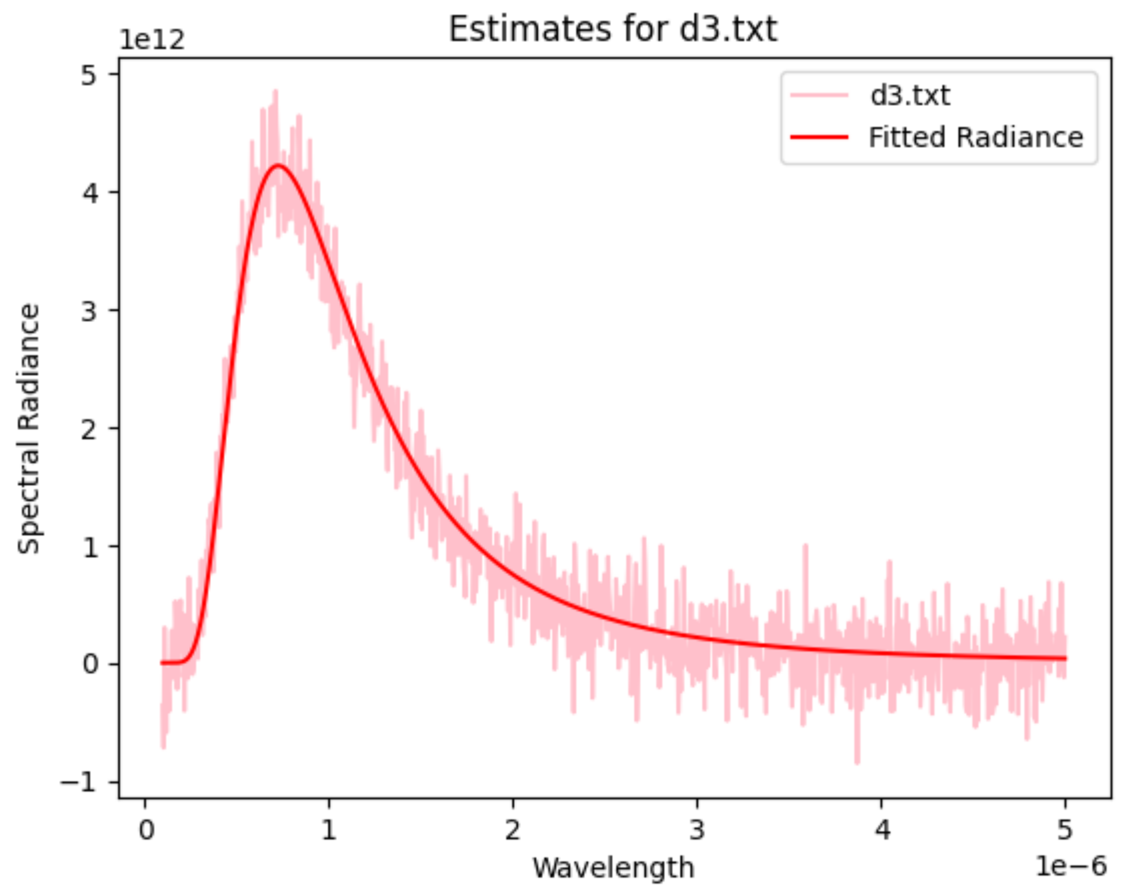
```
Estimates for d1.txt
T: [2874.62381308],               kB: [1.99999999e-23], h: [6.96711275e-34], c: [2.96994249e+08]
```



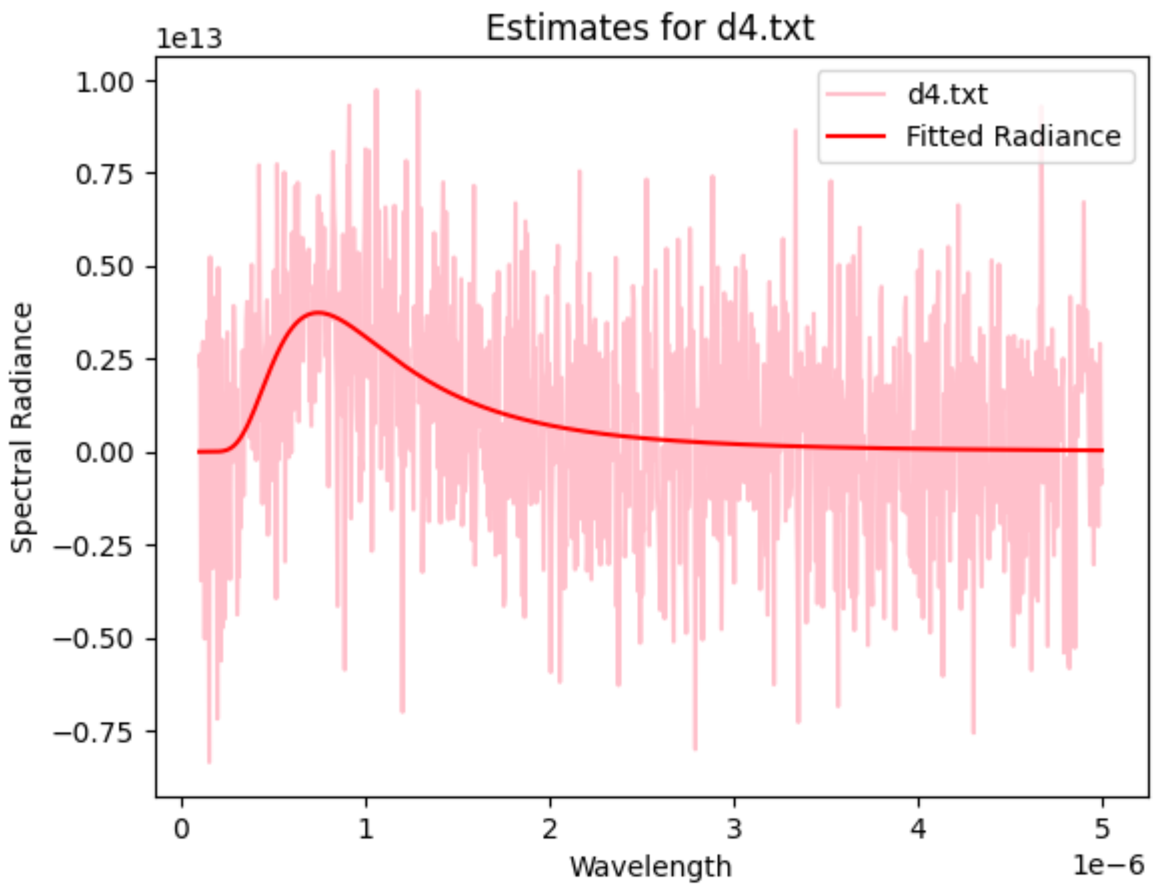
```
Estimates for d2.txt
T: [2661.35945464],              kB: [2.00000055e-23], h: [6.73660138e-34], c: [2.77954481e+08]
```



```
Estimates for d3.txt
T: [2853.18789852],              kB: [1.99999998e-23], h: [6.95736779e-34], c: [2.96103051e+08]
```



```
Estimates for d4.txt
T: [2785.0243282],              kB: [1.99999447e-23], h: [6.95793126e-34], c: [2.96123907e+08]
```


Estimates for d4.txt
1e13
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
Spectral Radiance
d4.txt
Fitted Radiance
0
1
2
3
4
5
1e−6
Wavelength