1. a) We begin by write out the expressions

$$\begin{aligned}
&f(x+\delta)-f(x-\delta)\\
=&\left[f(x)+f'(x)\delta+\frac{1}{2}f''(x)\delta^2+\frac{1}{6}f'''(x)\delta^3+\frac{1}{24}f^{(4)}(x)\delta^4+\frac{1}{120}f^{(5)}(x)\delta^5\right]\\
&-\left[f(x)-f'(x)\delta+\frac{1}{2}f''(x)\delta^2-\frac{1}{6}f'''(x)\delta^3+\frac{1}{24}f^{(4)}(x)\delta^4-\frac{1}{120}f^{(5)}(x)\delta^5\right]+\mathcal{O}(\delta^6)\\
=&2f'(x)\delta+\frac{1}{3}f'''(x)\delta^3+\frac{1}{60}f^{(5)}(x)\delta^5+\mathcal{O}(\delta^6)
\end{aligned}$$

and

$$\begin{aligned}
&f(x+2\delta)-f(x-2\delta)\\
=&\left[f(x)+2f'(x)\delta+2f''(x)\delta^2+\frac{4}{3}f'''(x)\delta^3+\frac{2}{3}f^{(4)}(x)\delta^4+\frac{4}{15}f^{(5)}(x)\delta^5\right]\\
&-\left[f(x)-2f'(x)\delta+2f''(x)\delta^2-\frac{4}{3}f'''(x)\delta^3+\frac{2}{3}f^{(4)}(x)\delta^4-\frac{4}{15}f^{(5)}(x)\delta^5\right]+\mathcal{O}(\delta^6)\\
=&4f'(x)\delta+\frac{8}{3}f'''(x)\delta^3+\frac{8}{15}f^{(5)}(x)\delta^5+\mathcal{O}(\delta^6).
\end{aligned}$$

Using both $f(x\pm\delta)$ and $f(x\pm 2\delta)$, we can cancel out the term of $\mathcal{O}(\delta^3)$ by

$$8\left[f(x+\delta)-f(x-\delta)\right]-\left[f(x+2\delta)-f(x-2\delta)\right]=12f'(x)\delta-\frac{2}{5}f^{(5)}(x)\delta^5+\mathcal{O}(\delta^6)$$

thus the estimate is

$$f'(x)=\frac{8\left[f(x+\delta)-f(x-\delta)\right]-\left[f(x+2\delta)-f(x-2\delta)\right]}{12\delta}$$

with error

$$e_1=\frac{-\frac{2}{5}f^{(5)}(x)\delta^5+\mathcal{O}(\delta^6)}{12\delta}=-\frac{1}{30}f^{(5)}(x)\delta^4+\mathcal{O}(\delta^6).$$

b) We have two sources of possible error, e_1 and rounding error e_2 of floating point numbers. Suppose the rounding error for a single float number is ϵ. We have total error

$$e=|e_1|+|e_2|=\frac{1}{30}f^{(5)}(x)\delta^4+\frac{8\left[\epsilon+\epsilon\right]+\left[\epsilon+\epsilon\right]}{12\delta}=\frac{1}{30}f^{(5)}(x)\delta^4+\frac{5}{6\delta}\epsilon.$$

Minimize e with respect to δ using

$$0=\left.\frac{\mathrm{d}e}{\mathrm{d}\delta}\right|_{\delta=\delta_{\min}}=\frac{2}{15}f^{(5)}(x)\delta_{\min}^3-\frac{5}{6\delta_{\min}^2}\epsilon$$

which gives

$$\delta_{\min}=\left(\frac{25}{4}\frac{\epsilon}{f^{(5)}(x)}\right)^{1/5}.$$

We know for Python's float, $\epsilon\approx 1\times 10^{-16}$. Let's consider the derivative near $x=1$. If $f(x)=\exp(x)$, we have $f^{(5)}(1)\approx 1$, plugging in gives

$$\delta_{\min}\approx 9\times 10^{-4}$$

and if $f(x)=\exp(0.01x)$, we have $f^{(5)}(1)\approx 0.01^4$, plugging in gives

$$\delta_{\min}\approx 0.04.$$

This can be tested with Python using `1.py` which gives the output of

```
Estimating f(x)=exp(x) at x=1:
Error at delta=0.01: 9.061000838528344e-10
Error at delta=0.001: 5.133671265866724e-13
Error at delta=0.0009: 9.015010959956271e-14
Error at delta=0.0001: 1.1546319456101628e-13
Error at delta=1e-05: 1.787858749935367e-11
Estimating f(x)=exp(0.01x) at x=1:
Error at delta=1: 3.3668987736712452e-12
Error at delta=0.1: 1.5178830414797062e-15
Error at delta=0.04: 1.491862189340054e-16
Error at delta=0.01: 5.401928904191777e-15
Error at delta=0.001: 8.681944052568724e-14
```

2. We let $\mathrm{d}x = \delta$ in this question. Using $[f(x+\delta) - f(x-\delta)]/2\delta$ to estimate $f'(x)$, we have total error from the expressions for $f(x+\delta) - f(x-\delta)$ in 1 a) as

$$e = \frac{1}{6} f'''(x)\delta^2 + \frac{\epsilon}{\delta}$$

thus

$$\delta_{\min} = \left(\frac{3\epsilon f(x)}{f'''(x)} \right)^{1/3}.$$

We need to find a way to estimate $f'''(x)$. From the expressions for $f(x+\delta) - f(x-\delta)$ and $f(x+2\delta) - f(x-2\delta)$ in 1 a), we have

$$f'''(x) = \frac{[f(x+2\delta) - f(x-2\delta)] - 2\,[f(x+\delta) - f(x-\delta)]}{2\delta^3}.$$

The implementation is as follows,

```python
import numpy as np


def ndiff(fun, x, full=False):
    """
    Numeric 1st order differentiation
    :param fun: function
    :param x: x value of list of x values
    :param full: if True, also print delta and estimated error
    :return: estimate and possibly additional info, type depends on parameters
    """
    # Determine if x is iterable
    try:
        iter(x)
    # If not iterable, i.e. a number
    except TypeError:
        # Float rounding error
        epsilon = 1e-16

        # Guess a sensible delta to compute f'''(x)
        delta = 1e-4
        # Compute f'''(x)
        fppp = ((fun(x + 2 * delta) - fun(x - 2 * delta))
                - 2 * (fun(x + delta) - fun(x - delta))) / (2 * delta ** 3)

        # Estimate the optimal delta
```

```
            delta = np.abs(3 * epsilon * fun(x) / fppp) ** (1 / 3)

            # Compute the estimate of 1st derivative
            estimate = (fun(x + delta) - fun(x - delta)) / (2 * delta)
            # Around the upper limit of the estimation's error
            error = fppp * delta ** 3 / 6 + epsilon / delta

            if full:
                return estimate, delta, error
            else:
                return estimate
        # If iterable, we iterate over all x and call ndiff for each of them
        else:
            result = []
            for y in x:
                result.append(ndiff(fun, y, full=full))

            # Return a list
            return result


    if __name__ == '__main__':
        # Let's test it with a simple example of f(x)=1/x
        func = lambda x: 1 / x
        dfunc = lambda x: - 1 / x ** 2
        x0 = [-1, 0.01, 0.1, 1, 10, 1000000]

        print(ndiff(func, x0, full=True))
        print(dfunc(np.array(x0)) - np.array(ndiff(func, x0)))
```

3. The Python code is as follows,

```
    import numpy as np
    import pandas as pd
    from scipy.interpolate import interp1d
    import matplotlib.pyplot as plt


    def lakeshore(v, data):
        """
        Perform interpolation on lakeshore data
        :param v: voltage or list of voltages
        :param data: data loaded by numpy.loadtxt()
        :return: tuple of interpolated value and estimated uncertainty or list of tuples
        """
        # Store data in a DataFrame
        df = pd.DataFrame(data, columns=['t', 'v', 'dt'])

        # Method of interpolation
        method = 'cubic'
        # Used for estimating uncertainty of the interpolation
        # The ratio of data that are sampled
        sample_ratio = 0.5
        # The number of samples
        sample_num = 100
```

```
    # Determine if v is iterable
    try:
        iter(v)
    # If not iterable, i.e. a number
    except TypeError:
        # Perform interpolation
        interp = interp1d(df['v'], df['t'], kind=method)

        # Array to store interpolation of samples
        sampled = np.empty(sample_num)
        for i in range(sample_num):
            # Sample the data set
            df_sample = df.sample(frac=sample_ratio)
            # Perform interpolation on the sampled data set
            # Allow extrapolate here bcause v may lie outside the range of the sample
            sampled[i] = interp1d(df_sample['v'], df_sample['t'], kind=method,
                                  fill_value="extrapolate")(v)
        # Compute the standard deviation of the samples
        error = sampled.std()

        return float(interp(v)), error
    # If iterable, we iterate over all v and call lakeshore for each of them
    else:
        result = []
        for w in v:
            result.append(lakeshore(w, data))

        # Return a list
        return result


if __name__ == '__main__':
    # Some testing
    d = np.loadtxt('lakeshore.txt')
    print(lakeshore([0.1, 0.12, 0.19], d))

    plt.figure()
    plt.scatter(d[:, 1], d[:, 0])
    xs = np.linspace(d[:, 1].min(), d[:, 1].max(), num=100)
    ys = np.array([i[0] for i in lakeshore(xs, d)])
    dys = np.array([i[1] for i in lakeshore(xs, d)])
    plt.fill_between(xs, ys - dys, ys + dys, color='tab:orange', alpha=0.2)
    plt.plot(xs, ys, color='tab:orange')
    plt.show()
```

4. We randomly selected 8 points in each interval and computed the interpolation. We used $n = 4, m = 5$ for the rational function fit. To determine the error for the fit, we select 100 evenly spaced points in the interval and computed the difference to the real function. The result is as follows,

```
Considering for f(x)=cos(x):
Polynomial error: 1.0987348692290908e-06
Spline error: 0.000196242721440658
Rational function error: 3.6398026675842803e-06
```

```
p = [ 1.00000112 -0.15271247 -0.40762691  0.06337944]
q = [-0.15272245  0.09241769 -0.01290997  0.004251  ]
Rational function (pinv) error: 3.639803427799762e-06
p = [ 1.00000112 -0.15271247 -0.40762691  0.06337944]
q = [-0.15272245  0.09241769 -0.01290997  0.004251  ]
Considering for f(x)=1/(x^2+1):
Polynomial error: 0.0003815104672788008
Spline error: 0.0011014462211567553
Rational function error: 40.247478726932
p = [12.473295    8.          0.          0.78755793]
q = [ 4. -2.  4. -2.]
Rational function (pinv) error: 4.738766901130254e-16
p = [ 1.00000000e+00 -7.10542736e-15 -3.33333333e-01  1.24344979e-14]
q = [-7.10542736e-15  6.66666667e-01  1.42108547e-14 -3.33333333e-01]
```

For $f(x) = \cos(x)$, all three interpolation gives reasonably good fit to the real function. The polynomial interpolation has the smallest error. Rational function interpolation using `numpy.linalg.inv` and `numpy.linalg.pinv` gives the same result.

For the Lorentzian, rational function interpolation using `numpy.linalg.pinv` gives the smallest error while polynomial interpolation and spline interpolation gives error of around the same order of magnitude. The rational function interpolation using `numpy.linalg.inv` has an abnormally high error. We expect the error to be zero when using the rational function, as the Lorentzian function is already in the form of a rational function. Looking at p, q, we can see that the terms that we expect to be zero have values much closer to zero when using `numpy.linalg.pinv`. Since there are terms that should be zero, the matrix we need to inverse is singular. The implementation of `numpy.linalg.pinv` can handle such matrices much better than `numpy.linalg.inv` by setting the eigenvalues that are close to zero to identically zero.