PHYS-512: Problem Set 1

Simon Briesenick

September 17, 2022

1. (a) An approximation for the first derivative of a function $f(x)$ whose Taylor expansion is evaluated at four points, $x \pm \delta$ and $x \pm 2\delta$, can be found by linear algebra:
The Taylor expansion at $f(x+\delta)$ reads

$$f(x+\delta) = \sum_{n=0}^{\infty} \frac{f^{(n)}}{n!}((x+\delta)-x)^n \tag{1}$$

$$= \sum_{n=0}^{\infty} \frac{f^{(n)}}{n!}\delta^n. \tag{2}$$

Up to fourth order in δ, this results in

$$f_{+1} = f(x+\delta) = f(x) + \delta f'(x) + \frac{\delta^2}{2}f''(x) + \alpha\delta^3 f'''(x) + O(\delta^4), \tag{3}$$

where $\alpha = \frac{1}{6}$.
Expressions at the other points are obtained by letting $\delta \mapsto -\delta$ and $\delta \mapsto \pm 2\delta$, respectively:

$$f_{-1} = f(x-\delta) = f(x) - \delta f'(x) + \frac{\delta^2}{2}f''(x) - \alpha\delta^3 f'''(x) + O(\delta^4), \tag{4}$$

$$f_{+2} = f(x+2\delta) = f(x) + 2\delta f'(x) + 2\delta^2 f''(x) + 8\alpha\delta^3 f'''(x) + O(\delta^4), \tag{5}$$

$$f_{-2} = f(x-2\delta) = f(x) - 2\delta f'(x) + 2\delta^2 f''(x) - 8\alpha\delta^3 f'''(x) + O(\delta^4). \tag{6}$$

Casting into a matrix equation, $f = Mf^*$

$$\begin{bmatrix} f_{+1} \\ f_{-1} \\ f_{+2} \\ f_{-2} \end{bmatrix} = \begin{bmatrix} 1 & \delta & \delta^2/2 & \alpha\delta^3 \\ 1 & -\delta & \delta^2/2 & -\alpha\delta^3 \\ 1 & 2\delta & 2\delta^2 & 8\alpha\delta^3 \\ 1 & -2\delta & 2\delta^2 & -8\alpha\delta^3 \end{bmatrix} \begin{bmatrix} f(x) \\ f'(x) \\ f''(x) \\ f'''(x) \end{bmatrix} \tag{7}$$

and finding the inverse to M, M^{-1}:

$$\begin{bmatrix} f(x) \\ f'(x) \\ f''(x) \\ f'''(x) \end{bmatrix} = \begin{bmatrix} 2/3 & 2/3 & -1/6 & -1/6 \\ 2/(3\delta) & -2/(3\delta) & -1/(12\delta) & 1/(12\delta) \\ -1/(3\delta^2) & -1/(3\delta^2) & 1/(3\delta^2) & 1/(3\delta^2) \\ -1/(6\alpha) & 1/(6\alpha) & 1/(12\alpha) & -1/(12\alpha) \end{bmatrix} \begin{bmatrix} f_{+1} \\ f_{-1} \\ f_{+2} \\ f_{-2} \end{bmatrix}. \tag{8}$$

Hence, one can write

$$f'(x) = \frac{2}{3\delta}f_{+1} - \frac{2}{3\delta}f_{-1} - \frac{1}{12\delta}f_{+2} + \frac{1}{12\delta}f_{-2}. \tag{9}$$

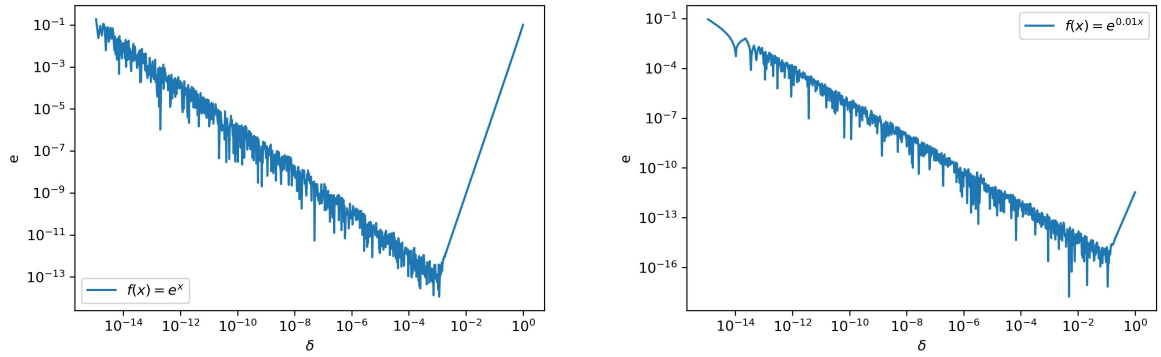

Figure 1: Error, $e(\delta)$ in the derivative approximation for the two exponential functions e^x (left) and $e^{0.01x}$ (right). Calculated for $x = 1$.

Simplifying and recovering the original notation leads to the final expression

$$f'(x) = \frac{1}{12\delta}\left[-f(x+2\delta) + 8f(x+\delta) - 8f(x-\delta) + f(x-2\delta)\right]. \tag{10}$$

By construction, the bracketed sum in eqn. 10 is independent of the quadratic, cubic and quartic powers of δ (Can also seen by inspection of the sign symmetry for the individual terms in eqns. 3-6). Consequently, the truncation error (in the brackets) is quintic in δ. Indeed,

$$\frac{1}{12\delta}\left[-\frac{2^5}{5!} + 8\frac{1}{5!} + 8\frac{1}{5!} - \frac{2^5}{5!}\right]\delta^5 \rightarrow O(\delta^4). \tag{11}$$

(b) Following the general framework for error estimation in numerical derivatives (Chapter 5.7 in Numerical Recipes), expressions for the two sources of error need to be found/estimated. The truncation error, e_t, due to higher order terms in the Taylor expansion scales as $|\delta^4 f^{(5)}(x)|$, while the round-off error is proportional to $\epsilon_f |f(x)/\delta|$. ϵ_f is the fractional uncertainty with which $f(x)$ for a given x is computed, i.e.

$$\epsilon_f = \frac{f_{\text{computed}}(x)}{f_{\text{analytic}}(x)} \tag{12}$$

and is taken to be on the same order of magnitude as the fractional uncertainty in the 64-bit floating point format (10^{-16}). With this assumption, the optimal δ is calculated as

$$\delta_{\text{opt}} \simeq \left(\frac{\epsilon_f f(x)}{f^{(5)}(x)}\right)^{1/5}. \tag{13}$$

Rough estimates for δ_{opt} are then 6×10^{-4} and 6×10^{-2} for $f(x) = e^x$ and $f(x) = e^{0.01x}$, respectively. The absolute difference between the derivative value from eqn. 10 and the analytical value is plotted in Fig. 1. It is seen that the calculated δ_{opt} is a fairly good approximation for the smallest overall error, e.

2. The numerical differentiator is based on the first-order, central difference:

$$f'(x) = \frac{f(x+dx) - f(x-dx)}{2dx}. \tag{14}$$

The truncation error scales with the third derivative - making the total error an equation of the form

$$e(dx) \simeq \epsilon_f \frac{f(x)}{dx} + f'''(x)dx^2 \rightarrow dx_{\text{opt}} \simeq \left(\frac{\epsilon_f f(x)}{f'''(x)}\right)^{1/3}. \tag{15}$$

For the algorithm, dx was initially guessed to be on the order of $\epsilon_f^{1/3}$. Subsequently, the third-order derivative was estimated using the central difference formula

$$f'''(x) = \frac{f(x+2dx) - 2f(x+dx) + 2f(x-dx) - f(x-2dx)}{2dx^3} \tag{16}$$

which then served to calculate an improved, rough estimate for dx_{opt} via eqn. 15.
Function Code:

```
import numpy as np

e_f = 1e-15 # machine precission

def ndiff(fun, x, full=False):
    # initial guess for dx
    dx = 1e-5
    # define derivative
    def deriv(fun, x, dx):
        return (fun(x + dx) - fun(x - dx))/(2*dx)
    # using the centered third order derivative for error
    def third_deriv(fun, x, dx):
        return (fun(x + 2*dx) - 2*fun(x + dx) +
            2*fun(x - dx) - fun(x - 2*dx))/(2*dx**3)
    # iterating once
    dx_opt = np.power(e_f*abs(fun(x)/third_deriv(fun, x, dx)), 1/3)
    if dx_opt == 0 or abs(dx_opt) == np.inf:
        dx_opt = dx
    error = e_f*abs(fun(x)/dx_opt) +
        abs(third_deriv(fun, x, dx_opt))*dx_opt**2
    if full:
        return (deriv(fun, x, dx_opt), dx_opt, error)
    else:
        return deriv(fun, x, dx_opt)
```

If `full` is set to `True`, the output is given as a tuple with the function value in the first place, dx_{opt} in the second and error estimate in the last. The third derivative can evaluate to zero in some cases such that `dx_opt` diverges. In such cases, the initial guess is taken. Similarly also for vanishing `dx_opt`.
Some test cases:

```
>>> ndiff(np.sin, np.pi/2, True)
(0.0, 1e-05, 9.999999999999999e-11)
>>> ndiff(np.sin, 0, True)
(0.9999999999833332, 1e-05, 1.000000321264849e-10)
>>> ndiff(np.exp, 0, False)
1.0000000000199893
>>> ndiff(np.exp, 1, True)
(2.718281828518058, 1.0363040380489199e-05, 5.408510906738229e-10)
```

This is sensible. For the first case, the third derivative evaluates to zero such that the initial guess for `dx_opt` is used instead. Similarly for the second case - Here, the function evaluates to zero and `dx_opt` vanishes. In the last example, the error is sufficiently large for the real value of e to lie within bounds.

3. To interpolate the temperature for an arbitrary input (array), I fit a B-spline to the existing data. Errors are estimated using a bootstrapping method, for which of the original 144 data points, 60 are chosen at random and re-sampled. For each sampling, a different B-spline model was constructed and compared to the original result. A thousand populations were drawn in this manner. The standard deviation of the differences thus calculated served as basis for the error estimation.

   A large portion of the function code is shown below. The output is then given as a tuple of numpy arrays, with the first entry being the estimated temperature values and the second the respective uncertainties.

```
from scipy.interpolate import splrep, splev
import random

dat = np.loadtxt('lakeshore.txt')

def lakeshore(V, data = dat):
    [SKIPPED]
    # create empty lists to store differences and final uncertainties
    differences, est_unc = [], []
    for i in range(1001):
        # strategy to draw 60 indices without replacement
        # such that those can be used to collect sub-population
        # from original population
        indices = random.sample(range(0, len(V_values)), 60)
        indices.sort() # need to be sorted for splrep
        V_picked = [V_values[j] for j in indices] # get data points
        T_picked = [T_values[k] for k in indices]
        # fit bootstrapped population
        fit_alt = splrep(V_picked, T_picked)
        # evaluate fit for input V
        estimates_alt = splev(V, fit_alt)
        difference = abs(estimates - estimates_alt)
        # store differences as array in list
        differences.append(difference)
    for index, voltage in enumerate(V):
```

```
        # for every voltage input, get all differences
        diffs = [differences[l][index] for l in range(len(differences))]
        # compute standard deviation of differences
        est_unc.append(np.std(diffs))
    est_unc = np.array(est_unc)
    return (estimates, est_unc)
```

As an example, the function is called on

```
lakeshore([1.1, 0.6, 1.54, 0.1])
```

with output

```
(array([ 33.38617731, 282.46404604,   5.32214554, 495.66893911]),
array([0.0134748 , 0.10520912, 0.00212785, 0.42657819]))
```

4. For this problem, three fit models are implemented to a simple cosine, $y(x) = \cos(x)$, as well as a Lorentzian $y(x) = 1/(1+x^2)$. The results in the former case are shown in Fig. 2. There are no apparent differences between the pure cosine, b-spline and rational models. Indeed, the respective errors (given as the standard deviation, summarized in Table 1) show that the polynomial fits the given data poorly in comparison, while the b-spline and rational models are comparable. However, this is hugely dependant on the number of points in the training set `x`. In the case presented, 5 points, evenly spaced between $-\pi/2$ and $\pi/2$, are taken. If the size of the training set is increased, the error in the rational function model quickly decreases to smaller values then those in the b-spline.

   Code Output for Cosine:

```
The errors (std) are
polynomial model:  0.02383061075092319
b-spline:  0.0034168785145555487
rational function:  0.001111531338043079
```

   Code Output for Lorentzian:

```
The errors (std) are
polynomial model:  0.03566274852070256
b-spline:  0.002465087739602392
rational function:  1.0590112691943469e-16
```

   For the Lorentzian, the rational model error is on the order of machine precision, which is to be expected, because the Lorentzian is a rational. Modifying the model parameters for the rational such that $(n, m) \mapsto (4, 5)$. The results are shown in Fig. 3. The rational model does not any longer accurately reflect the underlying data. Switching from `linalg.inv` to `linalg.pinv` restores the previous fit - however it is not entirely clear to me why.

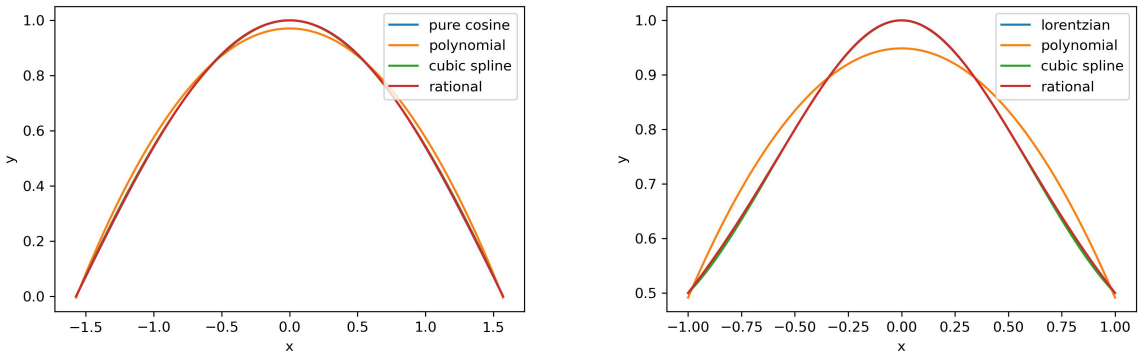

Figure 2: Comparison of fit models for the pure cosine (left) and the Lorentzian (right). In both cases, the polynomial is a quadratic. The rational function orders are chosen as $(n, m) = (2, 2)$.

Table 1: Fitting errors (σ).

Model	Cosine	Lorentzian
polynomial	0.02	0.04
b-spline	0.003	0.002
rational	0.001	1×10^{-16}

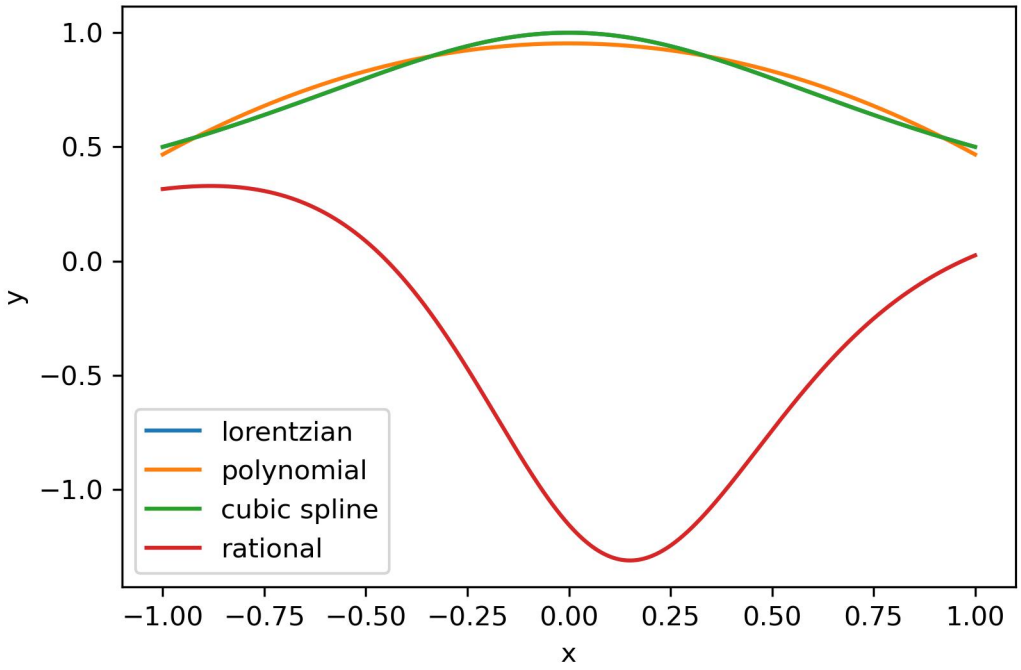

Figure 3: With the linalg.inv() function and $(n, m) = (4, 5)$, the fit does not converge like in Fig. 2.

References

[1] LakeShore Cryotronics, [Accessed: 11 September 2022]. https://www.lakeshore.com/products/categories/overview/temperature-products/cryogenic-temperature-sensors/dt-670-silicon-diodes