

Lecture

July 15, 2026

1 Scientific Programming: Analytics

Christian Elsasser (che@physik.uzh.ch), Jonas Eschle (jonas.eschle@gmail.com)

The content of the lecture might be reused, also in parts, under the CC-licence by-sa 4.0

```
[1]: import scipy
print("%s -- Version: %s"%(scipy.__name__,scipy.__version__))
import numpy as np
print("%s -- Version: %s"%(np.__name__,np.__version__))
```

```
scipy -- Version: 1.18.0
numpy -- Version: 2.5.1
```

```
[2]: import matplotlib.pyplot as plt
```

```
[3]: from matplotlib import rcParams
rcParams['font.size'] = 14
rcParams['font.sans-serif'] = ['Arial']
```

1.1 Use case 1 - Root-finding in non-linear functions

```
[4]: from scipy import optimize
```

Let's define the non-linear function $f(x) = x^3 + 8$ having the root at -2 . And define also its first and second derivative

```
[5]: def f_1_0(x):
    return x**3+8

def f_1_0_p(x):
    return 3*x**2

def f_1_0_pp(x):
    return 6*x
```

1.1.1 First approach: Bisection method

```
[6]: optimize.bisect(f_1_0,a=-10,b=10) # a,b are the interval boundaries for the  
     ↪ bisection method
```

```
[6]: -1.9999999999993179
```

We want now to get more information about the result

```
[7]: optimize.bisect(f_1_0,a=-10,b=10,full_output=True) # In newer Scipy version (1.  
     ↪ 2.1+) full_output is the default output
```

```
[7]: (-1.9999999999993179,  
      converged: True  
      flag: converged  
      function_calls: 46  
      iterations: 44  
      root: -1.9999999999993179  
      method: bisect)
```

The first item is the result as float, the second item a RootResult object

1.1.2 Second approach: Newton method

```
[8]: optimize.newton(f_1_0,x0=1, full_output=True)
```

```
[8]: (np.float64(-2.0000000000000004),  
      converged: True  
      flag: converged  
      function_calls: 6  
      iterations: 5  
      root: -2.0000000000000004  
      method: secant)
```

In case of no specification of fprime argument the secant method is invoked, with fprime (first derivative) the Newton-Raphson method is used, and with fprime2 (second derivative) in addition the Halley method is used.

```
[9]: print(optimize.newton(f_1_0,x0=100,fprime=None, fprime2=None, ))  
     print(optimize.newton(f_1_0,x0=100,fprime=f_1_0_p,fprime2=None, ))  
     print(optimize.newton(f_1_0,x0=100,fprime=f_1_0_p,fprime2=f_1_0_pp))
```

```
-1.9999999999999085  
-2.0  
-2.0
```

```
[10]: %timeit optimize.newton(f_1_0,x0=100,fprime=None, fprime2=None, )  
      %timeit optimize.newton(f_1_0,x0=100,fprime=f_1_0_p,fprime2=None, )  
      %timeit optimize.newton(f_1_0,x0=100,fprime=f_1_0_p,fprime2=f_1_0_pp)
```

```

707 s ± 33.2 s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
303 s ± 12.6 s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
252 s ± 9.46 s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

```

But this result can vary depending on the initial guess, here starting closer to the solution would lead to a better performance of the secant case (1st case)

Let's be naughty to the algorithm ...

```
[11]: optimize.newton(f_1_0,x0=0,full_output=True)
```

```

[11]: (np.float64(9.99999999997499e-05),
      converged: True
      flag: converged
      function_calls: 4
      iterations: 3
      root: 9.99999999997499e-05
      method: secant)

```

... which is obviously a wrong result despite the convergence (In Scipy 1.2.1+ the `newton` method has also a `RootResult` object when full output is specified.) What happens when specifying the first derivative?

```
[12]: optimize.newton(f_1_0,x0=0,fprime=f_1_0_p)
```

```

-----
RuntimeError                                Traceback (most recent call last)
Cell In[12], line 1
----> 1 optimize.newton(f_1_0,x0=0,fprime=f_1_0_p)

File ~/Desktop/pyschool_2026/.venv/lib/python3.12/site-packages/scipy/optimize/
↳ _zeros_py.py:336, in newton(func, x0, fprime, args, tol, maxiter, fprime2, x1,
↳ rtol, full_output, disp)
    331 if disp:
    332     msg += (
    333         f" Failed to converge after {itr + 1} iterations,"
    334         f" value is {p0}."
    335     )
--> 336     raise RuntimeError(msg)
    337 warnings.warn(msg, RuntimeWarning, stacklevel=2)
    338 return _results_select(
    339     full_output, (p0, funcalls, itr + 1, _ECONVERR), method)

RuntimeError: Derivative was zero. Failed to converge after 1 iterations, value
↳ is 0.0.

```

1.1.3 Third method - Brent's method

```
[13]: optimize.brentq(f_1_0,-100,100,full_output=True) # brentq also exists as an  
      ↪ alternative implementation
```

```
[13]: (-1.9999999999996092,  
      converged: True  
      flag: converged  
      function_calls: 21  
      iterations: 20  
      root: -1.9999999999996092  
      method: brentq)
```

We can also specify parameters of the function in the root-finding

```
[14]: def f_1_1(x,a):  
      return x**3+a
```

```
[15]: optimize.brentq(f_1_1,-100,100,full_output=True,args=(27,))
```

```
[15]: (-2.9999999999995026,  
      converged: True  
      flag: converged  
      function_calls: 20  
      iterations: 19  
      root: -2.9999999999995026  
      method: brentq)
```

1.2 Use case 2 - Maximum-Likelihood estimation (Optimisation and Distributions)

```
[16]: from scipy import stats
```

```
[17]: stats.distributions
```

```
[17]: <module 'scipy.stats.distributions' from  
      '/Users/che/Desktop/pyschool_2026/.venv/lib/python3.12/site-  
      packages/scipy/stats/distributions.py'>
```

Let's load some data (imagine a sample of measurements and you want to estimate the underlying distribution)

```
[18]: data = np.loadtxt("data_ml.csv")  
      data
```

```
[18]: array([-4.13213207e+00, -2.08913041e+00,  1.11461807e+00,  1.59224122e-01,  
      -1.76221163e+00,  2.04277819e+00, -6.65511125e-01, -2.04995514e+00,  
       2.13371573e+00, -2.74897103e+00,  4.21481079e+00, -8.50988747e-01,  
      -9.71802478e-01,  3.25676655e-01, -1.43108364e+00,  1.08889251e+00,
```

-6.38033426e-02, 2.12857629e+00, 1.01102696e+00, -3.61478343e-01,
 4.82079393e+00, -2.91616743e-01, 1.17974363e+00, 1.90949855e-01,
 2.49867012e+00, 6.11863034e+00, -1.70223622e+00, 3.00195470e-01,
 -3.48167371e+00, 1.42249755e+00, -5.56201449e+00, -2.54442132e+00,
 2.78043815e+00, 1.79828387e-01, -3.41758553e-01, -1.16098923e+00,
 2.78321643e+00, 9.90712604e-01, -7.40908124e-01, 2.90223321e+00,
 -2.38957523e+00, -2.64295515e+00, -2.08422264e+00, -2.10920762e+00,
 1.26366130e+00, -2.74692097e+00, -3.48783000e-01, 5.10459078e-01,
 1.73718178e+00, -3.99600803e-01, -3.01240374e+00, -1.56806200e+00,
 5.40788769e-01, -2.24102496e-01, -1.62726898e+00, 2.93926249e+00,
 2.03476960e+00, -2.83039575e+00, 2.09118597e+00, -2.21087920e-01,
 2.76815588e+00, -7.28330743e-01, 2.79188370e-01, -8.48817442e-02,
 5.35144208e-01, -1.57825758e+00, 2.12887888e+00, -1.31421199e+00,
 3.78384503e-01, 1.43337494e+00, -9.52741142e-01, -1.93663546e+00,
 2.18146132e+00, 9.09884431e-01, -1.00018301e+00, 1.60699366e+00,
 1.14545493e+00, 2.68873648e+00, -8.78081644e-01, -7.94530997e-02,
 -5.75026488e-01, -7.28829869e-01, -6.44569436e-01, -1.85066160e+00,
 -1.92352130e+00, -9.58198157e-01, -2.92188681e+00, 1.27493904e+00,
 -4.69658733e+00, -1.88619259e+00, -2.33967395e+00, 2.12947728e+00,
 4.77954787e-01, -2.70828742e+00, -1.41644500e+00, 2.17250251e-02,
 -6.48481323e-01, 6.40702430e-01, -3.58702521e+00, -6.82820855e-01,
 -2.76623924e+00, 4.48175144e+00, 8.11316385e-01, 2.43318803e+00,
 2.80220355e+00, 1.39375344e+00, -2.12539597e+00, 8.90845696e-01,
 2.87295162e-01, 4.70009520e-02, 1.94702603e+00, 3.74183240e+00,
 -6.78777802e-01, -1.42666140e-01, -1.67189160e+00, -7.35632853e-01,
 6.97800430e-01, 1.87548457e+00, -1.02990882e-01, -1.09683045e+00,
 -1.07383671e+00, -1.73351686e+00, 1.15943379e+00, -1.92038624e+00,
 3.41217611e-01, -1.00893787e+00, 1.80222820e+00, -1.25086680e+00,
 5.23223450e-01, 1.37242908e+00, -2.05956698e+00, 2.38768669e+00,
 -7.83045360e-01, -1.32895668e+00, -4.33940894e+00, 4.16900866e-01,
 3.12719500e+00, 1.39886871e+00, -4.06999763e-01, -7.31304350e-01,
 -5.87289057e+00, -2.55334816e+00, -4.13094907e+00, -4.20225346e+00,
 -4.74007789e-01, -2.33053412e+00, 1.91486081e+00, -1.05498895e+00,
 2.19912353e+00, 1.84627193e+00, 1.07388363e+00, -4.29910659e+00,
 2.33132727e+00, -3.27689417e+00, -1.20181225e+00, 1.40841742e+00,
 -2.63494870e+00, -1.05022448e+00, 4.07906902e-01, -2.37717837e+00,
 -1.06228106e+00, 6.29656649e-01, -2.01168831e+00, -7.54924571e-01,
 3.27299235e+00, 1.48421843e+00, -2.77594294e+00, -2.18493551e+00,
 1.75328742e+00, 2.25761250e-01, -1.01772878e+00, -3.80857961e+00,
 -1.40709682e+00, 4.98356930e-01, -1.32071695e+00, 4.54211566e-01,
 1.44803127e+00, -2.84412638e+00, 3.31619726e-01, -1.08294781e+00,
 7.22970931e-02, 9.40823919e-02, -6.74537478e-01, -1.65548442e+00,
 1.91342272e+00, -9.46280011e-01, -1.07293012e+00, 5.39147818e-01,
 1.12664808e+00, -4.71499395e-01, -3.07695519e+00, 6.66250231e-01,
 -2.16099078e-01, 1.95031527e+00, -1.22675007e+00, -2.49510494e+00,
 -1.42948279e-01, -8.88347028e-01, 1.11364617e+00, 4.05204994e+00,
 1.36271242e+00, -5.60869579e-02, -9.32466666e-01, 2.48439062e-01,

1.10327048e+00, -8.77179059e-01, -2.13296751e+00, 1.35611332e-01,
 -6.06988840e-01, 3.15518557e+00, -1.98234240e+00, -1.63586758e+00,
 -9.75744751e-01, 2.30739818e-01, -2.26972382e+00, 6.54074033e-01,
 -3.89926863e+00, 5.80677219e-01, -2.18791235e+00, -2.57464752e+00,
 2.65658627e-01, 1.08315228e+00, -7.02527359e-01, -2.67474069e+00,
 -8.86844119e-01, -1.73841279e+00, -4.25096377e-01, -3.26147459e-02,
 -2.93255063e-02, 3.26684649e-02, 2.46961636e+00, -2.39117685e+00,
 2.65629868e+00, -6.78341919e-01, 5.05632614e-01, -1.88797396e+00,
 -3.95361810e+00, 9.57619257e-01, 4.52791099e-01, 2.09003814e-01,
 -2.32644867e-01, -7.14224815e-01, -8.77621087e-01, 5.18874511e+00,
 -3.63204217e-01, 1.26686661e+00, 3.35401168e-01, 6.48896394e-01,
 -2.46127330e+00, -4.39484550e+00, -3.53306325e+00, 1.82729642e+00,
 -2.01126771e-01, 1.65548934e-01, 6.24343264e-01, -1.93409810e+00,
 -1.59662879e+00, -5.79268419e-01, 3.95841208e-01, -2.31433929e+00,
 -1.22688169e+00, -2.16571140e-02, 2.72107465e+00, -1.12254242e+00,
 1.56436399e-01, -1.05999951e+00, -3.67531089e+00, 1.14420653e+00,
 1.23775615e+00, 9.43256317e-01, 2.19621410e+00, -1.77192715e+00,
 2.41842329e-01, -2.73143690e+00, 1.07013700e+00, 1.34068706e+00,
 -1.58360611e-01, 3.14122054e+00, 5.30812495e-01, -7.25669395e-01,
 -8.66908143e-01, -1.29888936e+00, -2.17701300e+00, 3.18421623e+00,
 -5.75057103e-01, -3.93315452e+00, 6.18928946e-01, 4.83708179e-01,
 -4.69287231e-01, -7.14957970e-01, -3.48487500e+00, -1.04254101e+00,
 8.68067208e-01, 1.61304792e+00, -2.18052257e-01, -3.86025829e+00,
 -1.16702123e+00, -1.19572073e+00, 2.87042773e+00, 2.33652136e+00,
 2.14976941e+00, -5.47069983e-02, -3.17225276e+00, -1.62259153e+00,
 -7.25875127e-01, -2.55553780e+00, 3.65929840e+00, 3.15057456e-01,
 -2.31655839e+00, 4.52286054e-01, -1.72805335e+00, -1.18642118e+00,
 -6.84914030e-01, -4.57601610e-01, 1.64374790e+00, -7.79602847e-01,
 -4.49057905e-01, -1.50779079e-01, 1.24569471e+00, 7.06540033e-01,
 -1.58204842e+00, -1.70004116e+00, -2.39133896e+00, -1.06164307e+00,
 3.30659346e+00, -3.05392263e+00, 2.52321204e-01, -2.50037840e+00,
 -3.03095577e-01, -3.27400525e+00, -2.17590802e+00, -1.18576485e-01,
 5.41634237e+00, -9.51379036e-01, -4.06572426e+00, -1.84219077e-01,
 -1.74273358e+00, -9.66977407e-01, -1.36834322e+00, -3.67642960e+00,
 2.52672207e+00, -2.02033687e+00, 2.17077326e-01, 7.47682043e-01,
 2.35420159e+00, 2.67762848e+00, -1.42080358e+00, 2.75138528e+00,
 -1.60043683e+00, 8.78479485e-01, -1.99325550e+00, -1.78250427e+00,
 -1.26815423e+00, -4.88216593e-01, 2.94104275e+00, 2.79372481e+00,
 1.43266194e+00, 5.52150386e-01, -5.52913274e-01, -2.44771576e+00,
 3.54276542e+00, -2.79018686e+00, 2.20445790e+00, 1.70425549e+00,
 9.34544872e-01, 1.03300686e+00, -7.39362791e-01, -2.12576677e+00,
 -2.39306174e+00, 3.51916362e+00, 9.15439302e-02, 2.51170320e+00,
 7.77807922e-01, -4.47402895e-01, -1.73095122e+00, -1.59262228e+00,
 -1.73036397e+00, 2.46961226e+00, 2.41387555e+00, -8.93164763e-01,
 1.40276571e+00, 6.76442625e-01, -9.38254529e-01, -7.63660516e-01,
 4.89431449e-02, -1.69330206e-01, -6.14778320e-01, -2.12171402e-01,
 1.73640928e+00, 7.77206927e-01, -2.53870458e-01, 7.81905794e-01,

4.63430489e-01, 1.20681064e-01, 1.22174910e+00, -1.96836788e+00,
 -2.03335308e+00, 1.38983935e+00, -5.55564366e+00, 1.59824888e+00,
 -2.11376643e+00, 5.40922073e-01, 2.28524600e-01, -2.27406863e+00,
 1.26773917e+00, -7.97888840e-01, -2.77442712e+00, -9.91415301e-01,
 1.62164236e-01, -3.92846203e+00, 1.62824364e+00, -2.60483577e+00,
 2.15585610e+00, -1.01349791e-01, -9.13252502e-01, -2.65369244e-01,
 -5.05568353e+00, 3.75984688e-01, -1.26804400e+00, -1.76116343e+00,
 1.92354415e+00, 7.04701040e-02, 3.08221425e+00, -4.82861018e+00,
 -1.94433245e+00, 1.28683334e+00, 1.03615145e+00, -1.29060416e+00,
 -8.89703204e-01, -2.49052877e+00, -4.90142188e-01, 2.15967969e+00,
 2.57010989e-01, 1.27005210e+00, 5.17455712e-01, -1.05577127e+00,
 -1.48798507e+00, 4.44023964e-01, 1.14437173e+00, -5.25823450e-01,
 -4.99641728e-01, -1.80756570e+00, -1.64689149e-01, 1.63242999e+00,
 1.32718451e+00, 6.00757493e-01, -2.45933430e+00, -3.09142906e+00,
 1.02117032e+00, -2.37325729e+00, -5.30385196e-01, -6.81206141e-01,
 -1.39431731e-01, 1.05794370e+00, -2.34712607e+00, 1.33806111e+00,
 -7.45630387e-01, -1.32109697e-02, 2.21113823e+00, -4.31850946e-01,
 -7.13042121e-01, -9.28084393e-01, -4.26154340e+00, -1.99235485e-01,
 -2.21997421e+00, 2.48096492e+00, 1.76108031e+00, 2.41710833e+00,
 8.58313163e-01, 1.25073184e-01, 1.55557926e+00, -3.82878244e+00,
 2.06392321e+00, 4.86256345e-01, -7.44241859e-01, 4.32264371e+00,
 1.76517975e+00, -4.88650984e-01, 1.42418660e+00, -1.66956370e+00,
 -7.02120987e-01, -1.87972845e+00, -7.77205810e-01, -4.01690766e+00,
 2.26043860e+00, -4.50391997e-01, -4.02216523e+00, -1.51658196e+00,
 4.90770756e-01, 1.78383941e+00, -2.18347159e+00, 2.23733721e+00,
 9.51071380e-01, 1.30774479e+00, 1.02637695e+00, -4.42366354e-01,
 -2.42566259e-02, 1.17120600e+00, -1.28894053e+00, -1.24041530e+00,
 -1.98588809e+00, 1.54078474e+00, -2.91637880e+01, -5.61537586e+00,
 -1.95959440e+01, 7.92083288e+00, 1.33913715e+00, 1.60793535e+01,
 -1.12211211e+01, 2.79473207e+00, -3.28738025e+00, 6.27426471e-01,
 -7.34566387e-02, -8.20950900e+00, -2.26566689e+01, 1.67904483e+00,
 -1.11515361e+00, 7.29163333e+00, 1.49404404e+01, -6.53294020e+00,
 -1.16916228e+01, -6.93870646e-01, 2.74967451e+01, -2.24124809e+00,
 9.19692881e+00, 2.70172706e+01, 3.53444929e+00, -1.03096919e+00,
 1.62867226e+01, -1.75666381e+00, -4.01876663e+00, -1.48126758e+00,
 -2.61478672e+01, 6.93166661e+00, -4.05316946e-01, 8.44564251e+00,
 -3.20288266e+00, 6.65925671e+00, -7.50701907e-01, 5.09010629e+00,
 -1.01731666e+01, 1.38355710e+01, 2.87436080e+00, -4.52408092e-01,
 1.49782520e+01, 9.58039994e-01, -1.16290262e+01, 5.66818703e+00,
 -1.74371699e+01, 7.08932077e+00, -6.97272909e-02, -1.49242251e+01,
 -5.50865781e-01, -9.84775521e+00, 2.48849878e+00, -5.80476879e-01,
 2.80555620e+00, -8.32935606e+00, 3.17395959e+00, 7.20585505e+00,
 -6.74830845e+00, 1.38496313e+01, -1.76312088e+01, 6.01508591e+00,
 -2.38603206e+00, -2.23038867e+01, 6.02748783e+00, 3.98033353e-01,
 1.39733020e+01, 1.61723649e+00, 4.68194906e-01, 2.38724908e+01,
 -1.81784942e+01, 1.40139603e+01, -1.44941239e+01, 6.76269898e+00,
 -6.24627321e+00, 7.88161325e+00, -1.51759911e+01, 1.16381111e+01,

```

2.86319338e+01, -7.27001950e-01, 2.29605000e+01, -5.10816319e+00,
8.26439939e+00, 3.19342105e+00, -2.39657533e+01, -7.01316056e+00,
1.35798437e+01, 4.24537197e+00, 1.77945185e+01, -2.04825195e+01,
-4.35263152e+00, -3.83496197e+00, 1.90676918e+01, 9.97890868e+00,
-1.70791536e+01, 7.40232114e+00, 1.67596473e+01, -1.91453643e+00,
1.00291762e+01, 5.71500683e+00, -3.07926987e+00, -1.57683273e+01,
-6.91918252e+00, 8.07654101e+00, -1.22707170e+01, 2.91237123e-01,
-1.14733692e-01, -1.43950095e+01, -5.37499265e+00, -1.35492114e+01,
-4.60273397e+00, -1.27092383e+00, -4.78901210e+00, -2.52192459e+00,
-1.15764787e+01, 3.72334346e+00, 3.05644296e+00, 4.47621922e+00,
-1.88376849e+01, 5.94700980e+00, -4.37102655e+00, 1.49204637e+01,
-5.75278381e+00, 4.25168421e+00, -1.25426305e+01, -3.61568648e+00,
1.24695840e+01, -2.51536588e+00, -1.52278855e+01, 8.33917318e+00,
-1.75316559e+00, 1.36840679e+01, -4.13532819e+00, 1.48423396e+01,
2.06671655e+01, 5.39663416e-01, -4.49805506e+00, 2.11967936e+01,
-2.57516331e+01, 1.37464299e+00, 6.17562793e+00, -9.78255959e+00,
1.94477092e+00, -2.04413636e+01, 1.08805415e+01, 9.14142179e+00,
-3.49281723e+00, -1.21211503e+00, -9.62205477e+00, 5.15731938e+00,
-1.03519030e+01, -2.47768132e+00, -2.31752353e+00, 1.87601666e+01,
7.47181398e-01, -1.36432734e+01, 7.97864198e+00, 8.44267017e+00,
-1.03672922e+01, -3.28013355e+00, 1.52444558e+01, -1.23851940e+01,
-1.08952095e+01, -1.89442486e+01, -7.00305480e-02, 7.82369729e+00,
6.56159825e+00, 1.96170712e+01, -9.60115780e+00, -1.97783873e+01,
4.79934103e+00, -7.87219401e+00, 9.11986826e-01, 7.01840375e+00,
-1.67486632e-01, 1.67137316e+00, 8.13301991e+00, 1.25257128e+01,
-1.21846874e+01, 2.10608599e+01, -3.39222552e+00, 1.20489163e+01,
-1.30076977e+01, -1.43686165e+01, 8.93679342e+00, 1.39232254e+01,
-2.25427581e+01, 6.39129061e+00, 1.33965387e+01, -1.52334101e+01,
1.38267494e+00, 1.04738787e+01, -5.97205720e+00, -4.08133801e+00,
-4.57270609e+00, 4.30711216e+00, 1.61869636e+01, 1.04568435e+01])

```

Let's quickly inspect them as a histogram data

```

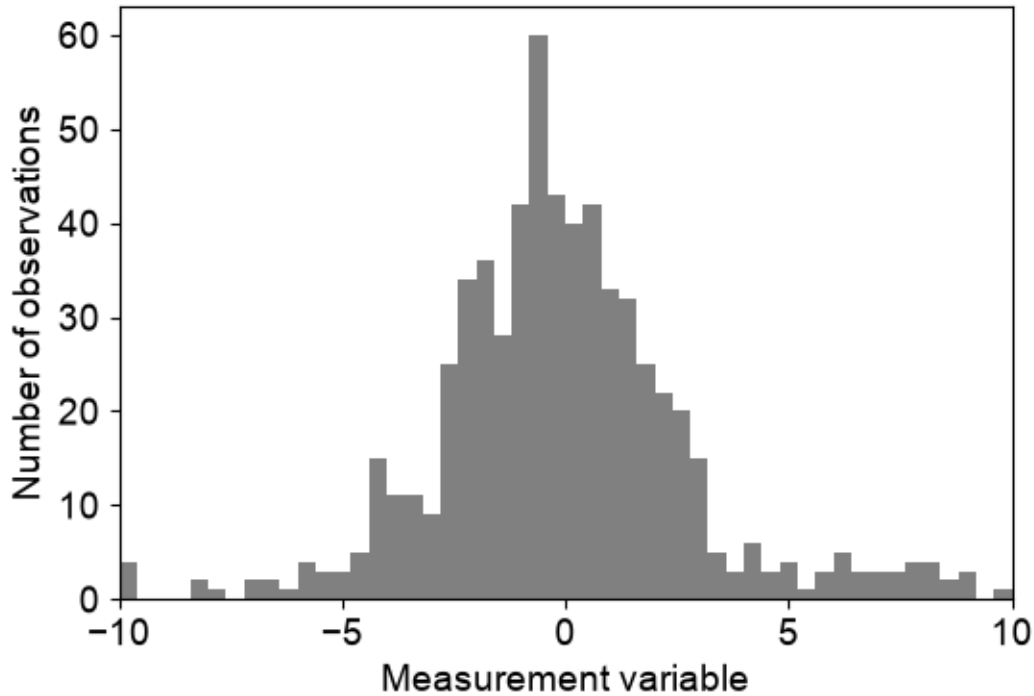
[19]: bins = np.linspace(-10,10,51)
      h,bins = np.histogram(data,bins)

```

```

[20]: fig = plt.figure("data",(6,4))
      ax = fig.subplots()
      ax.bar((bins[:-1]+bins[1:])*0.5,h,width=bins[1:]-bins[:-1],color="grey")
      ax.set_xlabel("Measurement variable")
      ax.set_ylabel("Number of observations")
      ax.set_xlim([-10,10])
      plt.show()

```



1.2.1 Distribution object

We can create the distribution object (Normal/Gaussian distribution with $\mu = 4$ and $\sigma = 2.5$)

```
[21]: f = stats.norm(4,2.5)
```

... and then use it in different ways

```
[22]: print(f.pdf(1.0))
print(f.cdf(1.0))
print(f.ppf(0.75))
print(f.std())
print(f.support())
```

```
0.0776744219932852
0.11506967022170822
5.686224375490204
2.5
(np.float64(-inf), np.float64(inf))
```

First approach: Gaussian distribution - we define the function describing the probability density with $p[0]$ as μ and $p[1]$ as σ .

```
[23]: f_gauss = lambda x,p: stats.norm(p[0],p[1]).pdf(x)
# Non-lambda definition:
# def f_gauss(x,p)
```

```
# return stats.norm(p[0],p[1]).pdf(x)
```

```
[24]: f_gauss(1.0,[4,2.5]) # The same as f.pdf(1.0) before
```

```
[24]: np.float64(0.0776744219932852)
```

!!! The order of the parameters is in scipy not always obvious and still not well documented !!! It follows for many distribution a location-scale approach (1st parameter: location, 2nd: scale), even for distributions like the exponential distribution that do not belong to this type of distribution

1.2.2 Simple minimisation

Let's try to minimise the function $f(x) = x^2$, which has a minimum at $x = 0$

```
[26]: def f_m_1(x):  
      return x**2
```

```
[27]: result = optimize.minimize(f_m_1,[4],method="CG")  
result
```

```
[27]: message: Optimization terminated successfully.  
      success: True  
      status: 0  
      fun: 1.3928233748494762e-13  
      x: [-3.732e-07]  
      nit: 1  
      jac: [-7.315e-07]  
      nfev: 6  
      njev: 3
```

```
[28]: result.x
```

```
[28]: array([-3.7320549e-07])
```

And we can also minimise multi-dimensional functions, so let's try $f(x_0, x_1) = x_0^2 + x_1^2$ with a minimum at $(x_0, x_1) = (0, 0)$

```
[30]: def f_m_2(x):  
      return x[0]**2+x[1]**2
```

```
[31]: optimize.minimize(f_m_2,[4,6],method="CG")
```

```
[31]: message: Optimization terminated successfully.  
      success: True  
      status: 0  
      fun: 2.5845142726365957e-17  
      x: [ 2.820e-09  4.230e-09]  
      nit: 2  
      jac: [ 2.054e-08  2.336e-08]
```

```
nfev: 15
njev: 5
```

And we can also have further parameters in the function, which we can define at the point of minimisation via the `args` variable

```
[32]: def f_m_3(x,a):
      return x[0]**2+x[1]**2+a
```

```
[33]: optimize.minimize(f_m_3,[4,6],method="CG",args=(5,))
```

```
[33]: message: Optimization terminated successfully.
      success: True
      status: 0
      fun: 5.0000000000000001
      x: [ 2.689e-08  1.053e-08]
      nit: 2
      jac: [ 5.960e-08  5.960e-08]
      nfev: 15
      njev: 5
```

Let's define the negative log-likelihood function

$$-\log \mathcal{L} = -\sum_i \log(f(d_i|p))$$

that we want to minimize

```
[34]: def log_likelihood_f_gauss_data(p):
      return -(np.log(f_gauss(data,p))).sum()
```

Let's minimise the log-likelihood function with the data and starting parameters $\mu = 0, \sigma = 2.5$. We use as first minimisation method the Conjugate Gradient method

```
[35]: res_g_hard = optimize.minimize(log_likelihood_f_gauss_data,[0.0,2.
      ↪5],method="CG")
      print(res_g_hard)
```

```
message: Optimization terminated successfully.
success: True
status: 0
fun: 2281.287183701291
x: [-1.433e-01  6.297e+00]
nit: 9
jac: [ 0.000e+00  0.000e+00]
nfev: 57
njev: 19
```

But this is cumbersome if we have different data sets and different distributions to test. For every combination we need to define a new function that should be minimised. So we can define a function to calculate the negative log-likelihood for a generic function `f` and generic data points `d`

```
[36]: def log_likelihood(p,d,f):
        return -(np.log(f(d,p))).sum()
        # here the data & distribution acts as parameter
```

```
[37]: res_g = optimize.minimize(log_likelihood,[0.0,2.
        ↪5],args=(data,f_gauss),method="CG")
        # we need to specify the data and the distribution fucntion as parameters
        print(res_g)
```

```
message: Optimization terminated successfully.
success: True
status: 0
      fun: 2281.287183701291
         x: [-1.433e-01  6.297e+00]
        nit: 9
         jac: [ 0.000e+00  0.000e+00]
        nfev: 57
        njev: 19
```

For completeness let's check the analytical results which are the mean and the standard deviation of the sample

```
[38]: print(f"mu      = {np.mean(data):10.7f}")
        print(f"sigma = {np.std(data):10.7f}")
```

```
mu      = -0.1433018
sigma =  6.2968100
```

```
[39]: res_g = optimize.minimize(log_likelihood,[0.0,2.5],
        ↪args=(data,f_gauss),method="TNC")
        print(res_g)
```

```
message: Converged (|f_n-f_(n-1)| ~= 0)
success: True
status: 1
      fun: 2281.287183701672
         x: [-1.433e-01  6.297e+00]
        nit: 9
         jac: [ 0.000e+00 -1.364e-04]
        nfev: 192
```

```
[40]: res_g["x"] # or res_g.x
```

```
[40]: array([-0.14330178,  6.29680535])
```

We create the fitted distribution

```
[41]: f_gauss_fit = stats.norm(*(res_g.x)) # by unpacking the fitted parameters when
        ↪creating the distribution
```

```
[ ]: optimize.minimize?
```

Let's try a t-distribution to accomodate for the longer tails. $p[0]$ as number of degrees of freedom, $p[1]$ as centre, $p[2]$ as width parameter.

```
[42]: f_t = lambda x,p: stats.t(p[0],p[1],p[2]).pdf(x)
```

```
[43]: res_t = optimize.minimize(log_likelihood,[11,0.0,2.6],  
    ↪args=(data,f_t),method="Nelder-Mead")  
print(res_t)
```

```
message: Optimization terminated successfully.  
success: True  
status: 0  
  fun: 2036.9785670880706  
   x: [ 1.314e+00 -2.535e-01  1.920e+00]  
  nit: 167  
 nfev: 302  
final_simplex: (array([[ 1.314e+00, -2.535e-01,  1.920e+00],  
                        [ 1.314e+00, -2.536e-01,  1.920e+00],  
                        [ 1.314e+00, -2.535e-01,  1.920e+00],  
                        [ 1.314e+00, -2.536e-01,  1.920e+00]]), array([  
2.037e+03,  2.037e+03,  2.037e+03,  2.037e+03]))
```

Let's do some testing for model selection with the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC):

$$AIC = 2k - 2\log \mathcal{L}$$

$$BIC = \log(n)k - 2\log \mathcal{L}$$

where L is the maximised likelihood, k is the number of estimated parameters and n the number of data points. A large difference between two models correspond to a significant difference in performance in favour of the model with the lower value.

```
[44]: AIC_g = 2*len(res_g.x)+2*res_g.fun  
AIC_t = 2*len(res_t.x)+2*res_t.fun  
BIC_g = np.log(len(data))*len(res_g.x)+2*res_g.fun  
BIC_t = np.log(len(data))*len(res_t.x)+2*res_t.fun
```

```
[45]: print(f"AIC Gauss:  {AIC_g:8.1f}")  
print(f"AIC t-dist:  {AIC_t:8.1f}")  
print(f"BIC Gauss:  {BIC_g:8.1f}")  
print(f"BIC t-dist:  {BIC_t:8.1f}")
```

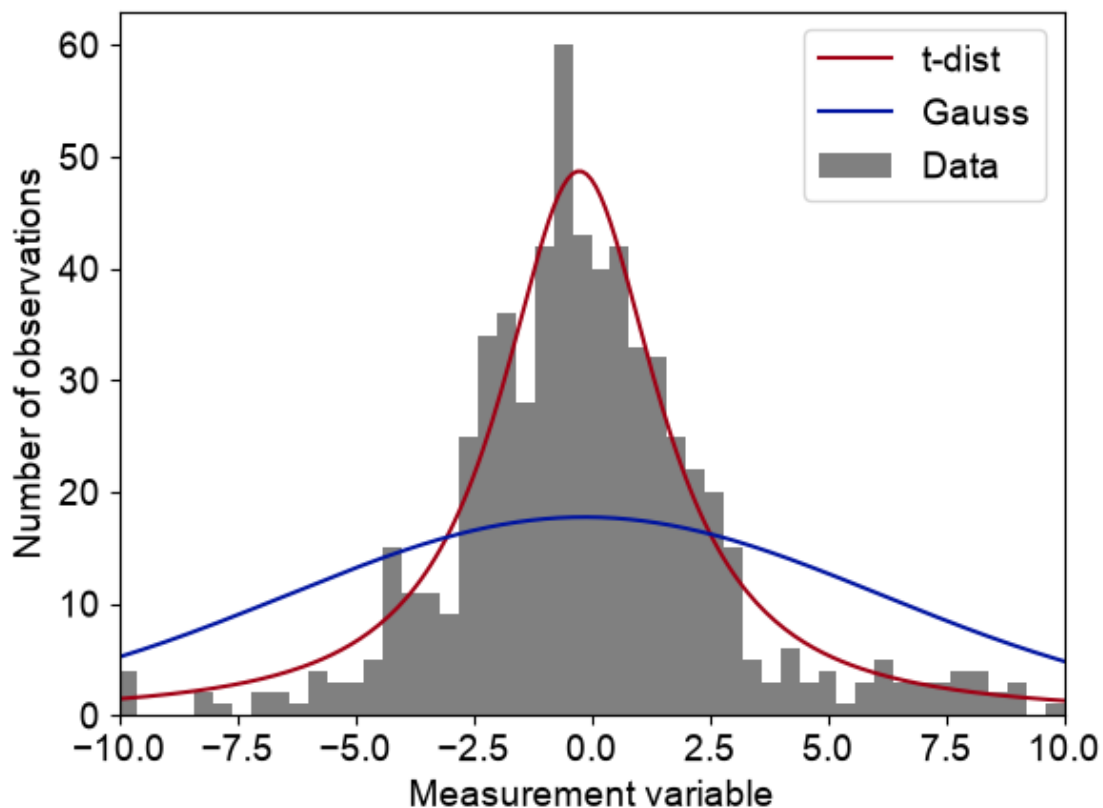
```
AIC Gauss:    4566.6  
AIC t-dist:    4080.0  
BIC Gauss:    4575.7  
BIC t-dist:    4093.6
```

Let's create the fitted t-distribution

```
[46]: f_t_fit = stats.t(*(res_t.x))
```

```
[47]: x_vec = np.linspace(-10,10,201)
fig = plt.figure("data", (6,4))
fig, ax = plt.subplots()
ax.bar((bins[:-1]+bins[1:])*0.5, h, width=bins[1:]-bins[:-1], color="grey")
ax.plot(x_vec, f_t_fit.pdf(x_vec)*data.size*(bins[1:]-bins[:-1]).
        ↳mean(), color="#a00014")
ax.plot(x_vec, f_gauss_fit.pdf(x_vec)*data.size*(bins[1:]-bins[:-1]).
        ↳mean(), color="#0014a0")
ax.set_xlabel("Measurement variable")
ax.set_ylabel("Number of observations")
ax.legend(["t-dist", "Gauss", "Data"])
ax.set_xlim([-10,10])
plt.show()
```

<Figure size 600x400 with 0 Axes>



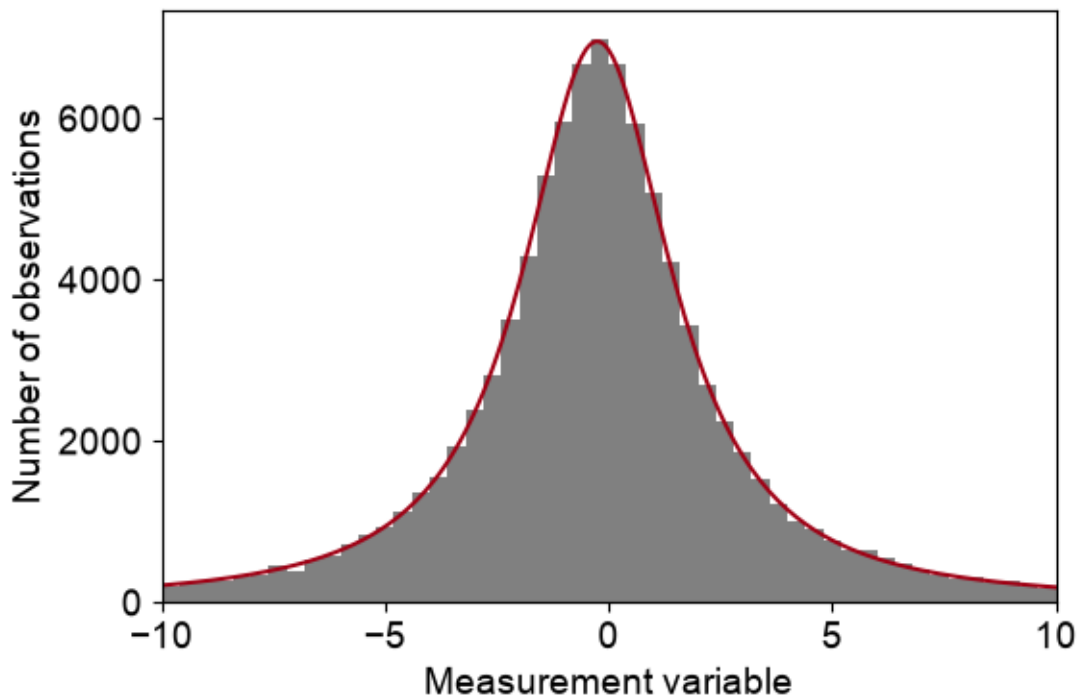
Sampling 100'000 = (500 by 200) random measurements according to the estimated t-distribution

```
[48]: mc_data = f_t_fit.rvs((500,200))
```

Histogram the measurements and plot them

```
[49]: bins = np.linspace(-10,10,51)
      h,bins = np.histogram(mc_data,bins)
```

```
[50]: fig = plt.figure("mc_data",(6,4))
      ax = fig.subplots()
      ax.bar((bins[:-1]+bins[1:])*0.5,h,width=bins[1:]-bins[:-1],color="grey")
      ax.plot(x_vec,f_t_fit.pdf(x_vec)*mc_data.size*(bins[1:]-bins[:-1]).
        ↪mean(),color="#a00014")
      ax.set_xlabel("Measurement variable")
      ax.set_ylabel("Number of observations")
      ax.set_xlim([-10,10])
      plt.show()
```



Get the 5% and the 95% quantile

```
[51]: print(f" 5% quantile: {f_t_fit.ppf(0.05):6.3f}")
      print(f"95% quantile: {f_t_fit.ppf(0.95):6.3f}")
```

```
5% quantile: -8.470
95% quantile:  7.963
```

Certain minimisation methods require you to indicate the first derivative (jacobian) or even second derivative (hessian) as analytical functions. Let's define a simple 2D-function: $f(x_0, x_1) = (x_0 - 5)^2 + (x_1 - 6)^4$

```
[53]: def f_2_0(x):
        return (x[0]-5)**2+(x[1]-6)**4

def f_2_0_p(x):
    return np.array([
        2*(x[0]-5),
        4*(x[1]-6)**3,
    ])

def f_2_0_pp(x):
    return np.array([
        [
            2,
            0
        ],
        [
            0,
            4*3*(x[1]-6)**2
        ]
    ])
```

Setting the initial guess

```
[54]: p_init = np.array([5,2])
# Please note that this is a special case where one of the coordinates is
↪ already at the optimal value
```

The usual Conjugate Gradient method will fail ...

```
[55]: optimize.minimize(f_2_0,p_init,method="CG")
```

```
[55]: message: Desired error not necessarily achieved due to precision loss.
success: False
status: 2
      fun: 15.369536160000013
       x: [ 5.000e+00  4.020e+00]
      nit: 1
      jac: [ 0.000e+00 -3.105e+01]
     nfev: 51
     njev: 16
```

But switching to the Newton

```
[56]: optimize.minimize(f_2_0,p_init,jac=f_2_0_p,method="Newton-CG")
```

```
[56]: message: Optimization terminated successfully.
success: True
```

```

status: 0
  fun: 1.3715480756696346e-09
    x: [ 5.000e+00  5.994e+00]
  nit: 17
  jac: [ 0.000e+00 -9.015e-07]
 nfev: 18
 njev: 35
nhev: 0

```

```
[57]: optimize.minimize(f_2_0,p_init,jac=f_2_0_p,hess=f_2_0_pp,method="Newton-CG")
```

```

[57]: message: Optimization terminated successfully.
      success: True
      status: 0
        fun: 1.3753061522324479e-09
          x: [ 5.000e+00  5.994e+00]
        nit: 17
        jac: [ 0.000e+00 -9.034e-07]
       nfev: 17
       njev: 17
      nhev: 17

```

Let's see what adding the hessian can do in terms of performance

```

[58]: %timeit optimize.minimize(f_2_0,np.array([100,100]),method="CG")
      %timeit optimize.minimize(f_2_0,np.
      ↪array([100,100]),jac=f_2_0_p,method="Newton-CG")
      %timeit optimize.minimize(f_2_0,np.
      ↪array([100,100]),jac=f_2_0_p,hess=f_2_0_pp,method="Newton-CG")

```

```

7.77 ms ± 530 s per loop (mean ± std. dev. of 7 runs, 100 loops each)
3.1 ms ± 292 s per loop (mean ± std. dev. of 7 runs, 100 loops each)
2.21 ms ± 86.8 s per loop (mean ± std. dev. of 7 runs, 100 loops each)

```

Depending on the situation (number of dimensions, starting position) the speed can be double!

1.3 Use case 3 - Linear equation solving (Matrices & Linear Algebra)

Previously Numpy provided the `matrix` class which is now depreciated in favour of the `ndarray` class, i.e. the standard numpy array. Matrix multiplication works via the `@` operator, matrix inversion and calculation of the Hermitian goes via the corresponding functions.

```

[59]: # Just here to ensure that the depreciation warning appears
      import warnings
      warnings.filterwarnings("default")

```

```

[60]: # Create a diagonal matrix
      d = np.matrix(np.diag([1,2,3,4]))

```

```

/var/folders/6f/7x1sl9x57g99x_3ss9j8wl4c0000gn/T/ipykernel_1143/2535179552.py:2:
PendingDeprecationWarning: the matrix subclass is not the recommended way to
represent matrices or deal with linear algebra (see
https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html). Please
adjust your code to use regular ndarray.
d = np.matrix(np.diag([1,2,3,4]))

```

```

[61]: d = np.diag([1,2,3,4])
      print(d)

```

```

[[1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]
 [0 0 0 4]]

```

```

[62]: # Create a 4x4 matrix with random entries and its inverse
s = np.array(np.random.rand(4,4))
s_inv = np.linalg.inv(s)

```

```

[63]: # De-diagonalise it
m = s_inv@d*s # matrix multiplication
      print(m)

```

```

[[ 6.87866263  1.37300226  4.77734544  2.98592802]
 [11.58819484  6.72159377  7.05027108  6.26443681]
 [ 7.76843172  2.25486727  5.82324049  3.37784761]
 [-22.57164309 -7.24713723 -14.42578787 -9.42349689]]

```

```

[64]: s_inv*d*s

```

```

[64]: array([[ -0.08874018,  -0.          ,  0.          ,  0.          ],
             [  0.          , -7.42180885,  0.          ,  0.          ],
             [  0.          , -0.          ,  0.51877056,  0.          ],
             [-0.          ,  0.          , -0.          , -25.69130403]])

```

```

[65]: # Get back the eigenvalues
      # ... and the eigenvectors
      values,transf = np.linalg.eig(m)
      print(values)
      print(transf)

```

```

[4.+0.j 1.+0.j 2.+0.j 3.+0.j]
[[ 0.17260084+0.j -0.05048208+0.j  0.08069801+0.j  0.04756868+0.j]
 [ 0.41993142+0.j  0.43640347+0.j  0.444642  +0.j  0.63756751+0.j]
 [ 0.29973766+0.j  0.42975531+0.j  0.31211743+0.j  0.23561671+0.j]
 [-0.83906132+0.j -0.78886878+0.j -0.83568178+0.j -0.73193555+0.j]]

```

For solving linear equations there are also dedicated algorithms available in the Scipy package:

$$5 = 6x_0 + 4x_1 - 2x_2 \quad (1)$$

$$3 = 3x_0 + 8x_1 + 2x_2 \quad (2)$$

$$7 = 8x_0 + 3x_1 + x_2 \quad (3)$$

i.e. $x = A^{-1}b$ with $A = \begin{pmatrix} 6 & 4 & -2 \\ 3 & 8 & 2 \\ 8 & 3 & 1 \end{pmatrix}$ and $b = \begin{pmatrix} 5 \\ 3 \\ 7 \end{pmatrix}$

```
[66]: A = np.array([[6,4,-2],[3,8,2],[8,3,1]])  
      b = np.array([5,3,7])
```

```
[67]: scipy.linalg.inv(A).dot(b)
```

```
[67]: array([0.85057471, 0.02873563, 0.1091954 ])
```

```
[68]: scipy.linalg.solve(A,b)
```

```
[68]: array([0.85057471, 0.02873563, 0.1091954 ])
```

For small equation systems the difference in the method is not that significant

```
[69]: %timeit scipy.linalg.inv(A).dot(b)
```

9.95 s ± 326 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

```
[70]: %timeit scipy.linalg.solve(A,b)
```

35.1 s ± 478 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

but for larger systems it starts to pay off

```
[71]: A = np.random.rand(1000,1000)  
      b = np.random.rand(1000)
```

```
[72]: %timeit scipy.linalg.inv(A).dot(b)
```

47.2 ms ± 2.92 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

```
[73]: %timeit scipy.linalg.solve(A,b)
```

13.6 ms ± 81.8 s per loop (mean ± std. dev. of 7 runs, 100 loops each)

1.4 Use case 4 - Signal processing (Fast-Fourier, Integration & Differentiation)

1.4.1 Differentiation

```
[74]: from scipy differentiate import derivative
```

```
[75]: def f_4_1(x):  
      return np.sin(x)
```

```
[76]: derivative(f_4_1,0,order=15)
      # `Order' is a confusing name since it is not the order of the derivative
```

```
[76]:      success: True
      status: 0
      df: 0.999999997712917
      error: 4.551914400963142e-15
      nit: 2
      nfev: 19
      x: 0.0
```

```
[77]: import numdifftools as nd
```

```
[78]: df_4_1 = nd.Derivative(f_4_1)
```

```
[79]: df_4_1(0)
```

```
[79]: array(1.)
```

```
[82]: df_4_1_2nd = nd.Derivative(f_4_1,n = 3)
      # `Order' is a confusing name since it is not the order of the derivative
```

```
[83]: df_4_1_2nd(0)
```

```
[83]: array(-1.)
```

1.4.2 Integration

```
[84]: from scipy import integrate
```

Let's integrate $f(x) = 1/x^2$ between 10^{-2} to 1 (analytical value: 99)

```
[85]: def f_4_2(x):
      return 1/x**2
```

Scipy has methods to calculate based on a sample of x_i and corresponding $y_i = f(x_i)$ the integral based on specific rules (Newton-Cotes methods)

```
[86]: x = np.linspace(0.01,1.00,100)
      y = f_4_2(x)
```

Let's first go for the Trapezoidal rule

```
[87]: integrate.trapezoid(y,x=x)
```

```
[87]: np.float64(113.49339001848928)
```

The integral shows a significant error due to the diverging behaviour towards 0. Let's try the Simpson's rule.

```
[88]: integrate.simpson(y,x=x)
```

```
[88]: np.float64(102.7445055588373)
```

Better due to the higher order, but still not good. So let's go for an adaptive algorithm

```
[89]: integrate.quad(f_4_2,a=0.01,b=1) # First return value is the integral, second
    ↪ the estimated error
```

```
[89]: (99.0, 3.008094132594605e-07)
```

```
[90]: %timeit integrate.trapezoid(y,x=x)
    %timeit integrate.simpson(y,x=x)
    %timeit integrate.quad(f_4_2,a=0.01,b=1)
```

9.24 s ± 147 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

272 s ± 3.54 s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

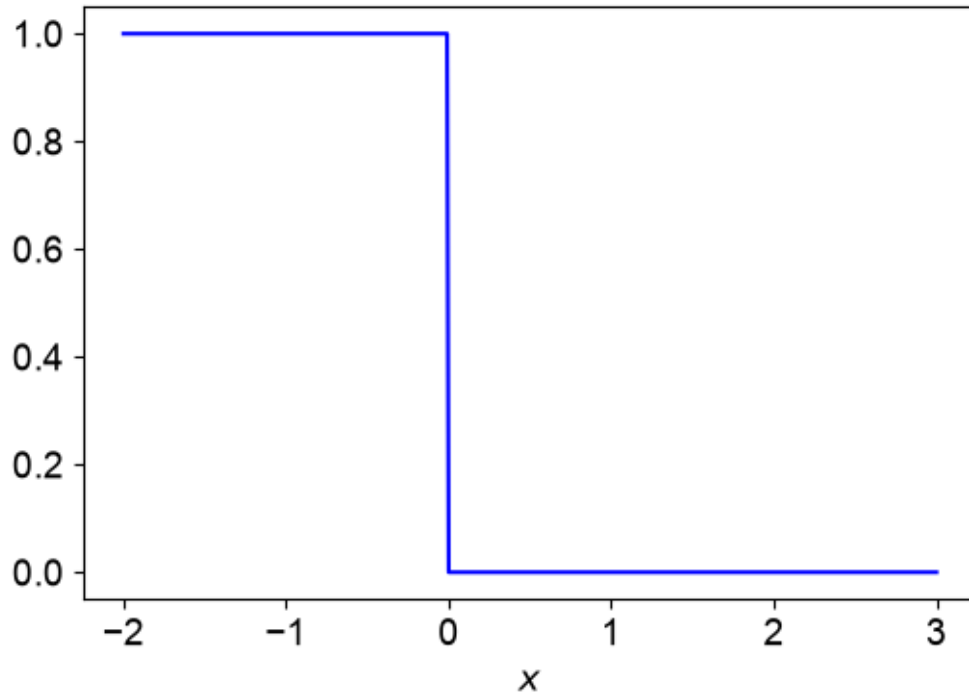
40.4 s ± 424 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Let's see what happens when we have a function with a discontinuity with $f(x) = \{x < 0\}$

```
[91]: def f_4_3(x):
    return (x<0)*1.0
```

```
[92]: fig = plt.figure("disc", (6,4))
    ax = fig.subplots()
    x = np.linspace(-2,3,501)
    ax.plot(x,f_4_3(x),color='b')
    ax.set_xlabel(r"$x$")
```

```
[92]: Text(0.5, 0, '$x$')
```



We integrate from -1 to $x > 0$ so the integral should be always one!

```
[93]: print(integrate.quad(f_4_3,-1,1))
      print(integrate.quad(f_4_3,-1,10))
      print(integrate.quad(f_4_3,-1,1000))
```

```
(1.0, 1.1102230246251565e-14)
(0.9999999999999989, 2.9976021664879227e-15)
(0.0, 0.0)
```

`points` allows to specify which point are particular points that need to be taken into account in the spacing, and can be even specified as a function.

```
[94]: print(integrate.quad(f_4_3,-1,1000,points=[0]))
```

```
(1.0, 1.1102230246251565e-14)
```

Scipy also provides `nquad` for multidimensional integration

```
[95]: def f_4_4(x0,x1,x2):
      return 1.0/(x0**2*x1**2*x2**2)
```

```
[96]: integrate.nquad(f_4_4,ranges=((0.1,1),(0.1,1),(0.1,1))) # It should be 9^3 = 729
```

```
[96]: (728.9999999999995, 2.0810636134419025e-06)
```

```
[97]: %timeit integrate.nquad(f_4_4,ranges=((0.1,1),(0.1,1),(0.1,1)))
```

399 ms ± 10.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

1.4.3 Application of integration: Fourier Transform

Scipy also includes a sub-moduel for Fast-Fourier Transforms (`scipy.fft`) which will be covered in a walk-through tutorial.

```
[98]: from scipy.integrate import quad
      from scipy import stats
```

Generating two signals

$$f(t) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(t-t_0)^2/2\sigma^2} \text{ (Gauss curve } \rightarrow \text{ bang)}$$

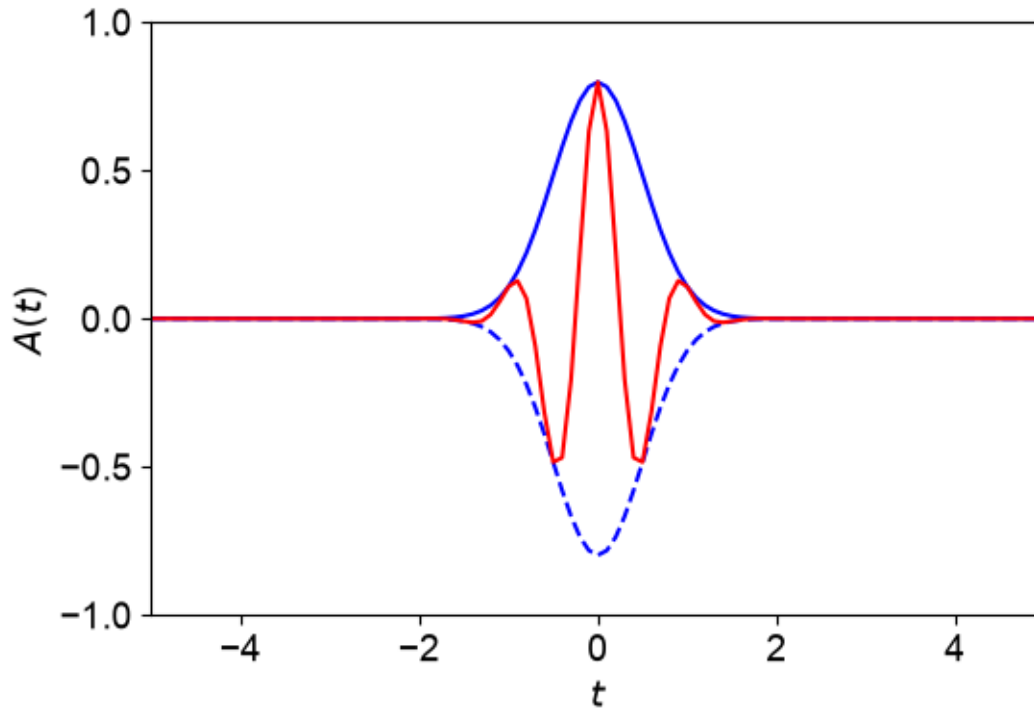
$$f(t) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(t-t_0)^2/2\sigma^2} \sin((t-t_0)\omega) \text{ (Gauss with sin wave)}$$

Here with $t_0 = 0$, $\sigma = 0.5$ and $\omega = 2\pi$

```
[99]: # Generating two signals
f_gauss = lambda x: stats.norm(0,0.5).pdf(x) # Gaussian signal
f_g_osc = lambda x: f_gauss(x)*np.cos(2*np.pi*x) # ... with oscillation
```

```
[100]: # time range considered
t = np.linspace(-5,5,101)
```

```
[101]: fig = plt.figure("signals",[6,4])
ax = fig.subplots()
ax.plot(t, f_gauss(t),'b')
ax.plot(t, -f_gauss(t),'b--')
ax.plot(t, f_g_osc(t),'r')
ax.set_xlim([t.min(),t.max()])
ax.set_ylim([-1.0,1.0])
ax.set_xlabel(r"$t$")
ax.set_ylabel(r"$A(t)$")
plt.show()
```



```
[104]: # Function that returns the fourier integrand as function, only considering the
      ↪ real part
def get_fourier_integrand(f):
    return lambda t,w: np.cos(w*t)*f(t)
```

```
[105]: # Define Fourier integrands for a given omega
f_i_gauss = lambda w : quad(get_fourier_integrand(f_gauss),-10,10,args=(w,))
f_i_g_osc = lambda w : quad(get_fourier_integrand(f_g_osc),-10,10,args=(w,))
```

```
[106]: f_i_gauss(1.0) # first entry: integration result, second entry: upper-bound on
      ↪ the error
```

```
[106]: (0.8824969025845957, 6.463572515755245e-11)
```

But we cannot apply a vector of ω to this function

```
[107]: f_i_gauss(np.array([0,0.5,1.0]))
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[107], line 1
----> 1 f_i_gauss(np.array([0,0.5,1.0]))

Cell In[105], line 2, in <lambda>(w)
```

```

----> 2 f_i_gauss = lambda w :
    ↪quad(get_fourier_integrand(f_gauss),-10,10,args=(w,))

File ~/Desktop/pyschool_2026/.venv/lib/python3.12/site-packages/scipy/integrate
    ↪_quadpack_py.py:479, in quad(func, a, b, args, full_output, epsabs, epsrel,
    ↪limit, points, weight, wvar, wopts, maxp1, limlst, complex_func)
    476     return retval
    478 if weight is None:
--> 479     retval = _quad(func, a, b, args, full_output, epsabs, epsrel, limit
    480                  points)
    481 else:
    482     if points is not None:

File ~/Desktop/pyschool_2026/.venv/lib/python3.12/site-packages/scipy/integrate
    ↪_quadpack_py.py:626, in _quad(func, a, b, args, full_output, epsabs, epsrel,
    ↪limit, points)
    624 if points is None:
    625     if infbounds == 0:
--> 626     return
    ↪_quadpack._qagse(func,a,b,args,full_output,epsabs,epsrel,limit)
    627 else:
    628     return _quadpack._qagie(func, bound, infbounds, args,
    ↪full_output,
    629                          epsabs, epsrel, limit)

TypeError: only 0-dimensional arrays can be converted to Python scalars

```

```

[108]: # Vectorise the integrands (allows to calculate the integration for different
    ↪omega in parallel)
int_f_gauss = np.vectorize(f_i_gauss)
int_f_g_osc = np.vectorize(f_i_g_osc)

```

```

[109]: int_f_gauss(np.array([0,0.5,1.0]))

```

```

[109]: (array([1.          , 0.96923323, 0.8824969 ]),
      array([8.67103044e-10, 8.40885132e-10, 6.46357252e-11]))

```

```

[110]: # Spectrum we are considering
w = np.linspace(0,10,51)

```

```

[111]: # Actual performance of the integration
yw_gauss = int_f_gauss(w)
yw_g_osc = int_f_g_osc(w)

```

```

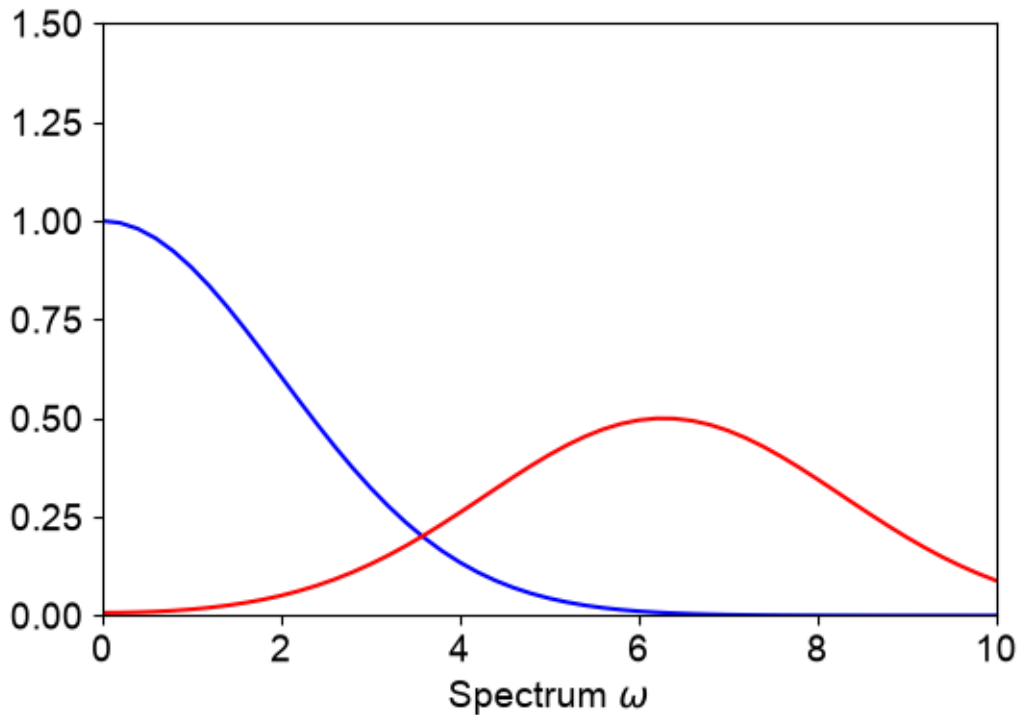
[112]: fig = plt.figure("Signal_spectra",[6,4])
ax = fig.subplots()
ax.plot(w, yw_gauss[0], 'b')

```

```

ax.plot(w, yw_g_osc[0], 'r')
ax.set_xlim([w.min(), w.max()])
ax.set_ylim([0, 1.5])
ax.set_xlabel(r"Spectrum  $\omega$ ")
plt.show()

```



[113]: *# But actually quad allows already to basically perform a Fourier transform via ↪ the weighting option*

```

f_i_gauss_alt = lambda w : quad(f_gauss, -10, 10, weight="cos", wvar=w)
int_f_gauss_alt = np.vectorize(f_i_gauss_alt)
yw_gauss_alt = int_f_gauss_alt(w)

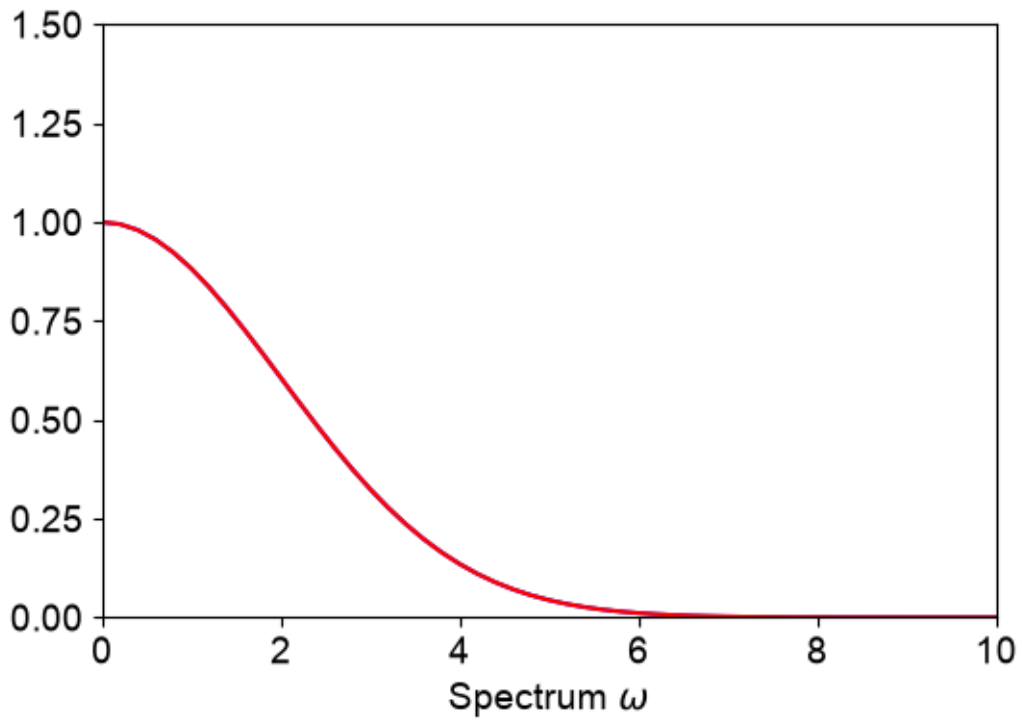
```

[114]: *# Comparing the original calculation with the weighting option calculation*

```

fig = plt.figure("Signal_spectrum_comparison", [6, 4])
ax = fig.subplots()
ax.plot(w, yw_gauss[0], 'b')
ax.plot(w, yw_gauss_alt[0], 'r')
ax.set_xlim([w.min(), w.max()])
ax.set_ylim([0, 1.5])
ax.set_xlabel(r"Spectrum  $\omega$ ")
plt.show()

```



[]:

[]: