

# week01-02–optimization-and-derivatives

February 1, 2025

## 1 [George McNinch](#) Math 87 - Spring 2025

## 2 § Week 2

## 3 Optimization & derivatives of functions

## 4 Using code to calculate derivatives

In our discussion of the `oil spill` problem, you may have been disappointed to have to do calculations with paper-and-pencil.

There are two possible ways around this, which I'd like to discuss briefly (with examples).

- We can use software for symbolic calculation of derivatives.
- Alternatively, we can *numerically approximate* derivatives.

This `notebook` will discuss these possibilities. For each method, we first treat some simple examples, and then we apply the method to the `oil spill` problem.

## 5 Symbolic calculations

First, let's investigate how `python` can make symbolic calculations using the `sympy` package.

For more details about symbolic calculations in `python` consult the [symbolic mathematics package](#).

### 5.1 A simple example

Let's find and classify the critical points for the cubic polynomial

$$G(t) = t^3 - 4t^2 - 5t - 2.$$

Let's import the `sympy` package, and declare `tt` to be a *symbol*:

```
[35]: import sympy as sp
      sp.init_printing()

      tt = sp.Symbol('t')
```

We now define the function  $G$ , and we create a corresponding *symbolic* version of  $G$  by evaluating the function  $G$  at the symbol `tt`.

```
[36]: def G(t): return t**3 - 4*t**2 - 5*t - 2

Gs = G(tt)
Gs
```

```
[36]: t3 - 4t2 - 5t - 2
```

Now we symbolically find the first and second derivative of  $G$ , using the function `diff` from the `sympy` package:

```
[37]: DGs = sp.diff(Gs,tt)          # first derivative
      DDGs = sp.diff(DGs,tt)        # second derivative
```

For example, we can see the first derivative:

```
[38]: DGs
```

```
[38]: 3t2 - 8t - 5
```

Now we use the `sympy` solver to find the critical points of  $G$  - i.e the solutions of the equation  $DGs == 0$

```
[39]: crits = sp.solve(DGs,tt)
      crits
```

```
[39]: [4/3 - sqrt(31)/3, 4/3 + sqrt(31)/3]
```

```
[40]: list(map(lambda c: c.evalf(),crits))
```

```
[40]: [-0.522588120943341, 3.18925478761001]
```

Using the function `lambdify`, we make an actual function `DDG` out of the symbolic expression `DDGs` and apply this function to each critical point:

```
[41]: DDG = sp.lambdify(tt,DDGs)
      list(map(DDG,crits))
```

```
[41]: [-2*sqrt(31), 2*sqrt(31)]
```

Since the value of `DDG` is *negative* at the first critical point, we see that  $G$  has a local max at  $t = \frac{4}{3} - \frac{\sqrt{31}}{3}$ .

Similarly,  $G$  has a local min at  $t = \frac{4}{3} + \frac{\sqrt{31}}{3}$ .

We confirm this with a sketch of the graph of  $G$ :

```
[42]: import matplotlib.pyplot as plt
      import numpy as np
```

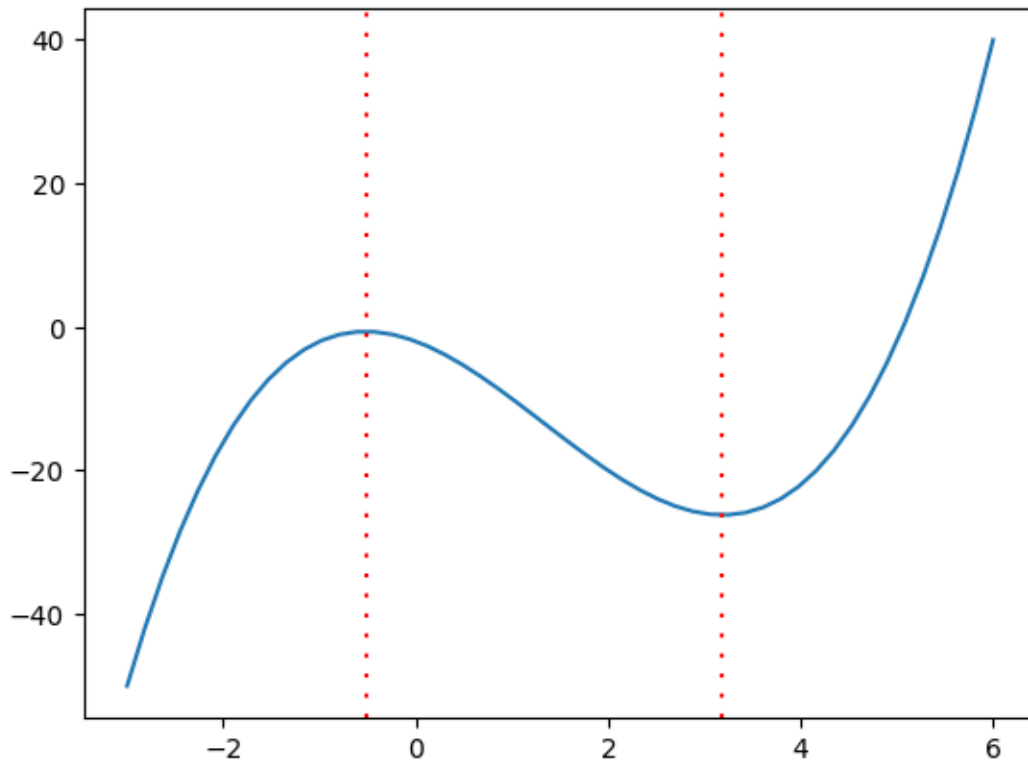
```

t = np.linspace(-3,6)

fig, ax = plt.subplots()
ax.plot(t,G(t),label="G")

for t in crits:
    ax.axvline(x=t, color="red", dashes=[1,4])

```



## 5.2 A trig example

Let  $H1(t) = \sin(5t)$  and  $H2(t) = \sin(5t + 3\pi/8)$ . Let's classify the critical points of  $H1(t)$  and  $H2(t)$  on the interval  $[-\pi, \pi]$ .

This time, we use the `sin` function from the `sympy` library.

```

[43]: import sympy as sp
      sp.init_printing()

      tt = sp.Symbol('t')

      H1s = sp.sin(5*tt)

```

```
H2s = sp.sin(5*tt + 3*sp.S.Pi/8)
```

```
[44]: DH1s = sp.diff(H1s)
      DH1s
```

```
[44]:  $5 \cos(5t)$ 
```

```
[45]: DH2s = sp.diff(H2s)
      DH2s
```

```
[45]:  $5 \cos\left(5t + \frac{3\pi}{8}\right)$ 
```

```
[46]: DDH1s = sp.diff(DH1s)
      DDH1s
```

```
[46]:  $-25 \sin(5t)$ 
```

```
[47]: DDH2s = sp.diff(DH2s)
      DDH2s
```

```
[47]:  $-25 \sin\left(5t + \frac{3\pi}{8}\right)$ 
```

Now, we want to find the critical points in the interval  $[-\pi, \pi]$ . For this, we first define this `interval` and use the `solveset` function to find the solutions to `DHs==0` on this interval:

```
[48]: int = sp.sets.sets.Interval(-np.pi,np.pi)

      crits1 = sp.solveset(DH1s,tt,domain=int)
      list(crits1)
```

```
[48]:  $\left[-\frac{9\pi}{10}, -\frac{7\pi}{10}, -\frac{\pi}{2}, -\frac{3\pi}{10}, -\frac{\pi}{10}, \frac{\pi}{10}, \frac{3\pi}{10}, \frac{\pi}{2}, \frac{7\pi}{10}, \frac{9\pi}{10}\right]$ 
```

```
[49]: crits2 = sp.solveset(DH2s,tt,domain=int)
      crits2
```

```
[49]:  $\left\{-\frac{39\pi}{40}, -\frac{31\pi}{40}, -\frac{23\pi}{40}, -\frac{3\pi}{8}, -\frac{7\pi}{40}, \frac{\pi}{40}, \frac{9\pi}{40}, \frac{17\pi}{40}, \frac{5\pi}{8}, \frac{33\pi}{40}\right\}$ 
```

We now use the second derivative test to classify the critical points as a (local) `min` or `max`

```
[50]: def classify(DD,cp):
      if DD.subs(tt,cp)>0:
          return "min"
      elif DD.subs(tt,cp)<0:
          return "max"
      else: return "inconclusive"

      list(map(lambda x: (x,classify(DDH1s,x)),crits1.evalf()))
```

```
[50]: [(-2.82743338823081, 'min'),
      (-2.19911485751286, 'max'),
      (-1.57079632679490, 'min'),
      (-0.942477796076938, 'max'),
      (-0.314159265358979, 'min'),
      (0.314159265358979, 'max'),
      (0.942477796076938, 'min'),
      (1.57079632679490, 'max'),
      (2.19911485751286, 'min'),
      (2.82743338823081, 'max')]
```

```
[51]: results = list(map(lambda x: (x, classify(DDH2s, x)), crits2.evalf()))
      results
```

```
[51]: [(-3.06305283725005, 'min'),
      (-2.43473430653209, 'max'),
      (-1.80641577581413, 'min'),
      (-1.17809724509617, 'max'),
      (-0.549778714378214, 'min'),
      (0.0785398163397448, 'max'),
      (0.706858347057703, 'min'),
      (1.33517687777566, 'max'),
      (1.96349540849362, 'min'),
      (2.59181393921158, 'max')]
```

Let's confirm our classification using graphs:

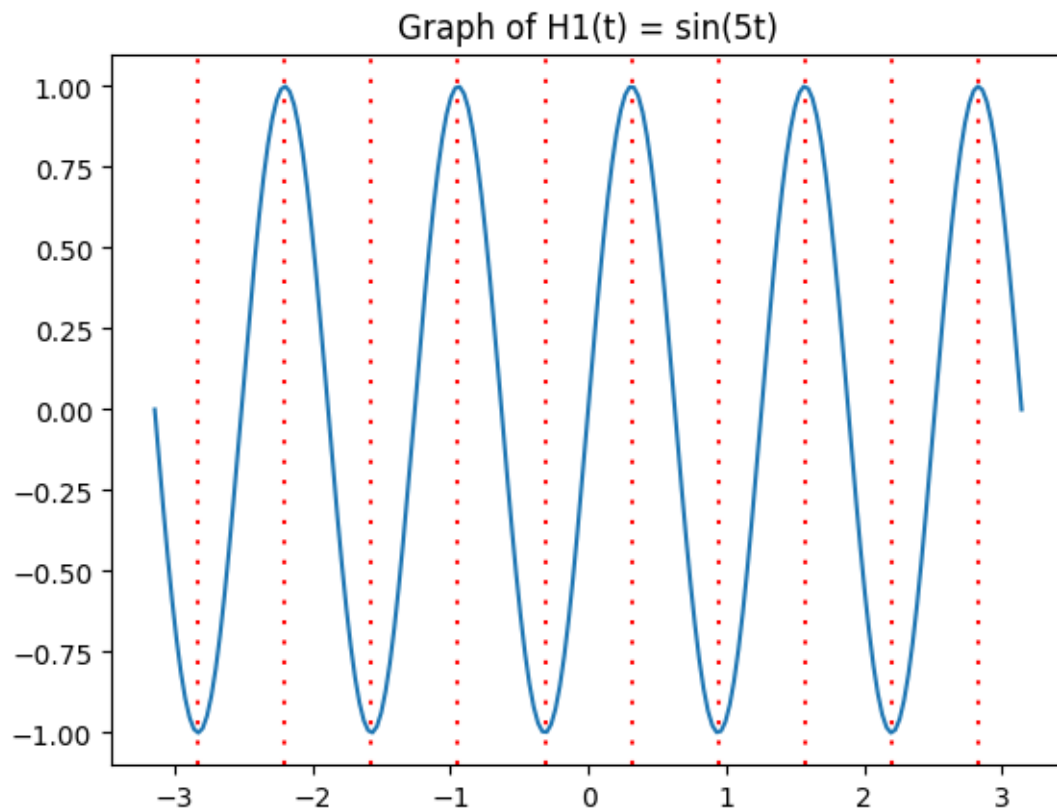
```
[52]: import matplotlib.pyplot as plt
      import numpy as np

      t1 = np.linspace(-np.pi, np.pi, 200)

      def H1(t): return np.sin(5*t)
      def H2(t): return np.sin(5*t + 3*np.pi/8)

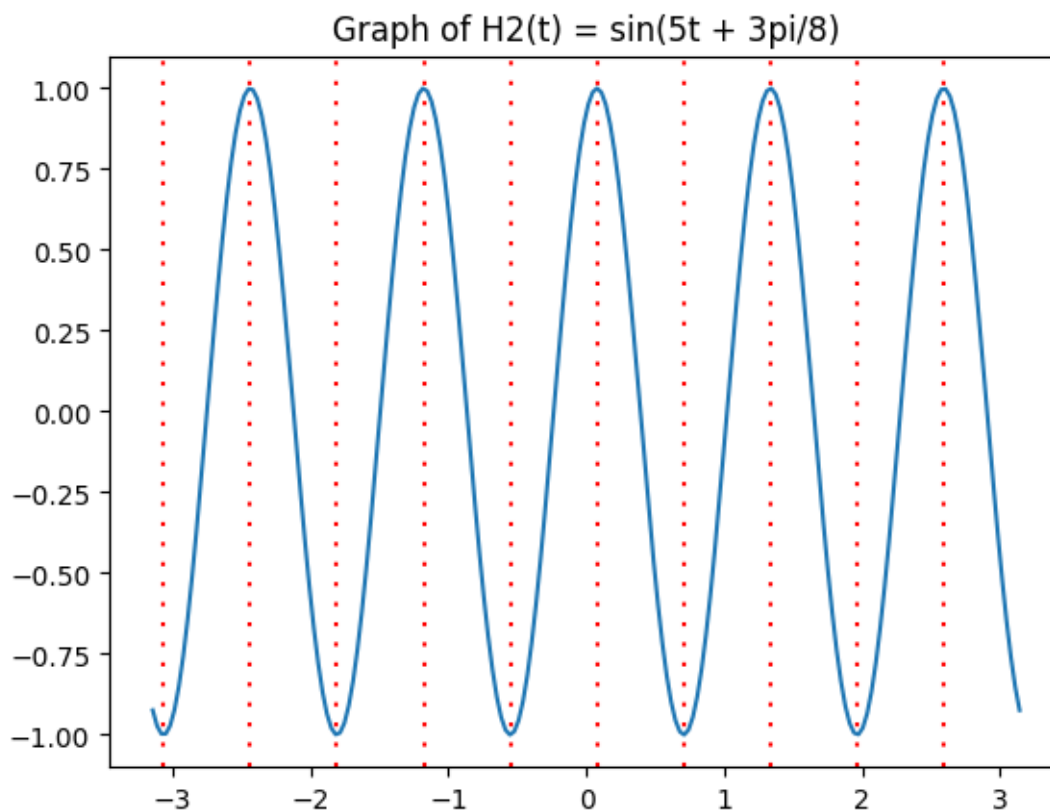
      fig, ax = plt.subplots()
      ax.set_title("Graph of H1(t) = sin(5t)")
      ax.plot(t1, H1(t1), label="H1")

      for t in crits1:
          ax.axvline(x=t, color="red", dashes=[1,4])
```



```
[53]: fig, ax = plt.subplots()
      ax.plot(t1,H2(t1),label="H2")
      ax.set_title("Graph of H2(t) = sin(5t + 3pi/8)")

      for t in crits2:
          ax.axvline(x=t, color="red", dashes=[1,4])
```



### 5.3 Return to the “oil spill” problem

Recall the `python` expressions for the main function of interest:

- $\$C_{\text{tot}}(n)$  `c.cost(n)`

We will make a “symbolic variable” we’ll call `y`.

We would like to make a symbolic version the `python` function `c.cost(n)` by valuation at `n=y`.

Unfortunately, our definition of `c.cost(n)` involved a test of inequality (to decide whether the fine calculation applied). But it is not “legal” to test inequalities with the symbol `y`. (More precisely, such tests can’t be sensibly interpreted).

For small enough `n`, `c.cost(n)` is equal to `c.crew_costs(n) + c.fine_per_day * (c.time(n)-14)`. And this latter expression *can* be evaluated at the symbolic variable `y`.

And *sympy* permits us to symbolically differentiate the resulting expression:

In the next cell, we load the *definitions* from the `oil spill` notebook.

```
[54]: %%capture
```

```
%run week01-01--optimization.ipynb import *
```

```
[55]: import sympy as sp
      sp.init_printing()

      c = OilSpillCleanup()

      y = sp.Symbol('y')    # symbolic variable

      def lcost(n):
          return c.crew_costs(n) + c.fine_per_day * (c.time(n) - 14)

      lcost_symb = lcost(y)
      D_lcost_symb = sp.diff(lcost_symb,y) # first derivative, for n<19
      DD_lcost_symb = sp.diff(D_lcost_symb,y) # second derivative, for n<19

      lcost_symb
```

```
[55]:
```

$$18000y + \frac{160000y}{0.714285714285714y + 0.714285714285714} - 140000 + \frac{2100000}{0.714285714285714y + 0.714285714285714}$$

```
[56]: D_lcost_symb
```

```
[56]:
```

$$-\frac{224000.0y}{(y+1)^2} + 18000 - \frac{2940000.0}{(y+1)^2} + \frac{160000}{0.714285714285714y + 0.714285714285714}$$

```
[57]: DD_lcost_symb
```

```
[57]:
```

$$\frac{448000.0y}{(y+1)^3} - \frac{448000.0}{(y+1)^2} + \frac{5880000.0}{(y+1)^3}$$

Now e.g. `sympy` solvers are able to find the critical point for the symbolic derivative `D_lcost_symb`, as follows:

```
[58]: crits = sp.solve(D_lcost_symb,y)
      print(crits)
```

```
[-13.2836838484589, 11.2836838484589]
```

Notice that the value of the second derivative at the positive critical point 11.28 is positive:

```
[59]: DD_lcost = sp.lambdify(y,DD_lcost_symb)

      DD_lcost(crits[1])>0
```

```
[59]: True
```

This the second derivative test shows that our positive critical point of 11.28 determines a *local minimum* for the cost function; this is the conclusion we came to previously.



Note that this symbolic method doesn't completely solve the problem: we still require analysis about the interval  $19 < n$  (where the cost function isn't modeled by our symbolic function `lcost_symb`).

## 6 Numerical calculations

In another direction, rather than relying on symbolic calculations, we can use numerical methods to approximate derivatives.

Let's see what this might look like. We import the `numpy` package, and define some functions to extract critical points. These functions depend on the `numpy` function `gradient` which - in the case of a function of a single variable - approximates the derivative.

```
[60]: import numpy as np

def crit_pts(ff,xx,tol=1E-5):
    gg = np.gradient(ff,xx)
    res = [ x for (x,g) in zip(xx,gg)
            if np.abs(g)<tol ]
    return res

def crit_pts_fun(f,a,b,n,tol=1E-5):
    xx=np.arange(a,b,1/n)
    ff=f(xx)
    return crit_pts(ff,xx,tol)
```

Let use these functions on our cubic polynomial  $G(t)$  from above. Remember that the `sympy` solve found the critical points to be  $\frac{4}{3} \pm \frac{\sqrt{31}}{3}$ .

```
[61]: def G(t): return t**3 - 4*t**2 - 5*t - 2

crit_pts_fun(G,-2,6,5E3,tol=1E-3)
```

```
[61]: [-0.5226000000000163, 3.189199999999943]
```

Compare with:

```
[62]: [4/3 - np.sqrt(31)/3, 4/3 + np.sqrt(31)/3]
```

```
[62]: [-0.522588120943341, 3.18925478761001]
```

**But:** if we change the tolerances in the argument to `crit_pts_fun`, we get redundant critical points, or we miss critical points.

```
[63]: crit_pts_fun(G,-2,6,5E3,tol=5E-3)
```

```
[63]: [-0.5230000000000163, -0.5228000000000163, -0.5226000000000163, -0.5224000000000163, -0.5222000000000163,
```

```
[64]: crit_pts_fun(G,-2,6,5E3,tol=1E-4)
```

```
[64]: []
```

Let's return to our oil spill problem.

```
[65]: %%capture

      %run week01-01--optimization.ipynb import *
```

If we make good choices of tolerances, we can get a pretty good estimate for the critical point of the cost function:

```
[66]: c = OilSpillCleanup()

      f = np.vectorize(c.cost)

      res=crit_pts_fun(f,0,19,1E4,1E-1)
      res
```

```
[66]: [11.2837]
```

But in some sense, this required us to already know the answer!

with the wrong tolerances, it is easy to miss the critical point:

```
[67]: res=crit_pts_fun(f,0,19,1E4,1E-2)
      res
```

```
[67]: []
```

And it is easy to get redundant reported critical points:

```
[68]: res=crit_pts_fun(f,0,19,1E4,4E-1)
      res
```

```
[68]: [11.2836, 11.2837, 11.2838]
```