week01-02–optimization-and-derivatives

January 21, 2024

1 George McNinch Math 87 - Spring 2024

2 § Week 1.3

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 investige 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*:

[1]:
```
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`.

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

Gs = G(tt)
Gs
```

[2]: $t^3 - 4t^2 - 5t - 2$

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

[3]:
```
DGs = sp.diff(Gs,tt)            # first derivative
DDGs = sp.diff(DGs,tt)          # second derivative
```

For example, we can see the first derivative:

[4]:
```
DGs
```

[4]: $3t^2 - 8t - 5$

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

[5]:
```
crits = sp.solve(DGs,tt)
crits
```

[5]: $\left[\frac{4}{3} - \frac{\sqrt{31}}{3}, \ \frac{4}{3} + \frac{\sqrt{31}}{3} \right]$

[7]:
```
list(map(lambda c: c.evalf(),crits))
```

[7]: $[-0.522588120943341, \ 3.18925478761001]$

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

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

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:

[ ]:
```
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.

[ ]:
```
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)
```

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

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

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

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

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:

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

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

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

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

[ ]:
```python
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()))
```

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

Let's confirm our classification using graphs:

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

tl = 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(tl,H1(tl),label="H1")

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

[ ]:
```python
fig, ax = plt.subplots()
ax.plot(tl,H2(tl),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_{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.

[ ]:
```
%%capture

%run week01-02--optimization.ipynb import *
```

[ ]:
```
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
```

[ ]:
```
D_lcost_symb
```

[ ]:
```
DD_lcost_symb
```

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

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

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

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

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

This the second derivative test shows that our postive 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.

[9]:
```python
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 $\frac{4}{3} \pm \frac{\sqrt{31}}{3}$.

[10]:
```python
def G(t): return t**3 - 4*t**2 - 5*t - 2

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

[10]: $[-0.522600000000163,\ 3.18919999999943]$

Compare with:

[12]:
```python
[4/3 - np.sqrt(31)/3,4/3 + np.sqrt(31)/3]
```

[12]: $[-0.522588120943341,\ 3.18925478761001]$

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

[22]:
```python
crit_pts_fun(G,-2,6,5E3,tol=5E-3)
```

[22]: $[-0.523000000000163,\ -0.522800000000163,\ -0.522600000000163,\ -0.522400000000163,\ -0.522200000000163,$

[23]:
```python
crit_pts_fun(G,-2,6,5E3,tol=1E-4)
```

[23]: []

Let's return to our `oil spill` problem.

[19]:
```
%%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:

[32]:
```
c = OilSpillCleanup()

f = np.vectorize(c.cost)

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

[32]: [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:

[37]:
```
res=crit_pts_fun(f,0,19,1E4,1E-2)
res
```

[37]: []

And it is easy to get redundant reported critical points:

[36]:
```
res=crit_pts_fun(f,0,19,1E4,4E-1)
res
```

[36]: [11.2836, 11.2837, 11.2838]