CS 3430: S19: SciComp with Py
Assignment 2
Computing Local Extrema and Inflection Points

Vladimir Kulyukin
Department of Computer Science
Utah State University

January 19, 2019

Learning Objectives

1. Differentiation
2. Local Maxima and Minima
3. First- and Second-Derivative Tests
4. Inflection Points

Introduction

In this assignment, you will use your differentiation functions from Assignment 1 to write functions that detect local extrema and inflection points.

Problem 1: Polynomials (1 point)

Let us assume that we will work with polynomials of the 1st, 2nd, and 3rd degrees. Let us further assume that we will represent each polynomial member explicitly. For example, a $2x^2 - 1$ will be represented as $2x^2 + 0x - 1$, $3x^2 + 5x$ as $3x^2 + 5x + 0$, and $4x^3 + 10x$ as $4x^3 + 0x^2 + 10x + 0$. Representing everything explicitly allows us to avoid getting bogged down in many special algebraic cases we would otherwise have to code. Another simplification that we will adopt is that every polynomial will start with the member of the highest degree. In other words, $10 + 0.005x - 0.78x^2$ will be represented as $-0.78x^2 + 0.005x + 10$.

Before we can do the derivative tests, we need to implement a couple of functions that find the zeros of polynomials. Toward that end, implement a function `find_poly_1_zeros`(expr) that takes a functional representation of a 1st degree polynomial, finds its zero point, and returns it as a constant object (see `const.py` for the definition of the `const` class). Write your code in `poly12.py` included in the zip. Below is a test run to find the zero of $2x + 5$.

We represent $2x+5$ with a couple of functions from `maker.py` to save ourselves some typing.

```
>>> from maker import make_prod, make_const, make_pwr, make_plus
>>> f1 = make_prod(make_const(2.0),
                   make_pwr('x', 1.0))
>>> f2 = make_plus(f1, make_const(5.0))
>>> print f2
((2.0*(x^1.0))+5.0)
```

On to finding the zeros of this polynomial.

```
>>> z = find_poly_1_zeros(f2)
>>> isinstance(z, const)
True
>>> print z
-2.5
```

To test if this is a true zero, we convert the polynomial's representation to a function with `tof` and test if it returns 0 when applied to the value of the constant `z`, i.e, 2.5.

```
>>> from tof import tof
>>> f = tof(f2)
>>> assert f(z.get_val()) == 0.0
```

Here is another test case which finds the zero of $3x + 100$.

```
def test_01():
    f1 = make_prod(make_const(3.0),
                   make_pwr('x', 1.0))
    f2 = make_plus(f1, make_const(100.0))
    print f2
    print is_pwr_1(f1)
    z = find_poly_1_zeros(f2)
    f2f = tof(f2)
    assert f2f(z.get_val()) == 0.0
    print str(z)
```

Below is the output by `test_01()` in the Python Shell.

```
>>> test_01()
((3.0*(x^1.0))+100.0)
True
-33.3333333333
```

We need to handle 2nd degree polynomials as well. So, implement a function `find_poly_2_zeros`(expr) that takes a functional representation of a 2nd degree polynomial, finds its zero point, and returns it as a constant object. Write your code in `poly12.py` included in the zip. Below is a test to find the zeros of $0.5x^2 + 6x$.

```
def test_02():
    f0 = make_prod(make_const(0.5), make_pwr('x', 2.0))
    f1 = make_prod(make_const(6.0), make_pwr('x', 1.0))
    f2 = make_plus(f0, f1)
    poly = make_plus(f2, make_const(0.0))
    print poly
    zeros = find_poly_2_zeros(poly)
    for c in zeros:
        print c
    pf = tof(poly)
    for c in zeros:
        assert abs(pf(c.get_val()) - 0.0) <= 0.0001
```

Here is the output of running `test_02()` in the shell.

```
>>> test_02()
(((0.5*(x^2.0))+(6.0*(x^1.0)))+0.0)
-12.0
0.0
```

Problem 2: Derivative Tests (3 points)

Implement a function `loc_xtrm_1st_drv_test(fexpr)` that takes a representation of a 2nd- or 3rd-degree polynomial and uses the 1st-derivative test to find its local extrema. Write your code in `derivtest.py` included in the zip. You may assume that `loc_xtrm_1st_drv_test` will handle functions with 0, 1, or, at most, 2 local extrema.

The function `loc_xtrm_1st_drv_test(fexpr)` returns a list, possibly empty, of 2-tuples. The first element of each 2-tuple is the string `'min'` or `'max'` and the second element is a `point2d` object (see `point2d.py`) that represents an extreme point.

The class `point2d` is a simple model of 2D points. The two members of each `point2d` object (i.e., `__x__` and `__y__`) are the constants representing the `x`-

and y-coordinates of an extreme point. Here is a quick example of how you can construct `point2d` objects, check their types, and get their members.

```
>>> from maker import make_point2d
>>> from point2d import point2d
>>> p1 = make_point2d(1.0, 2.0)
>>> type(p1)
<class 'point2d.point2d'>
>>> x = p1.get_x()
>>> y = p1.get_y()
>>> type(x)
<class 'const.const'>
>>> type(y)
<class 'const.const'>
>>> print p1
(1.0, 2.0)
>>> isinstance(p1, point2d)
True
```

Here is a test that calls `loc_xtrm_1st_drv_test(fexpr)` to compute the local extrema of $\frac{1}{3}x^3 - 2x^2 + 3x + 1$.

```
def test_03():
    f1 = make_prod(make_const(1.0/3.0), make_pwr('x', 3.0))
    f2 = make_prod(make_const(-2.0), make_pwr('x', 2.0))
    f3 = make_prod(make_const(3.0), make_pwr('x', 1.0))
    f4 = make_plus(f1, f2)
    f5 = make_plus(f4, f3)
    poly = make_plus(f5, make_const(1.0))
    print 'f(x) = ', poly
    xtrma = loc_xtrm_1st_drv_test(poly)
    for i, j in xtrma:
        print i, str(j)

>>> test_03()
f(x) =  ((((0.333333333333*(x^3.0))+(-2.0*(x^2.0)))+(3.0*(x^1.0)))+1.0)
max (1.0, 2.33333333333)
min (3.0, 1.0)
```

Here is a test of a cubic function with no local extrema. The function is $27x^3 - 27x^2 + 9x - 1$.

```
def test_04():
    f1 = make_prod(make_const(27.0), make_pwr('x', 3.0))
    f2 = make_prod(make_const(-27.0), make_pwr('x', 2.0))
    f3 = make_prod(make_const(9.0), make_pwr('x', 1.0))
```

```
    f4 = make_plus(f1, f2)
    f5 = make_plus(f4, f3)
    f6 = make_plus(f5, make_const(-1.0))
    print 'f(x) = ', f6
    drv = deriv(f6)
    assert not drv is None
    print 'f\'(x) = ', drv
    xtrma = loc_xtrm_1st_drv_test(f6)
    assert xtrma is None
```

The output of `test_04()` is as follows.

```
>>> test_04()
f(x) =  ((((27.0*(x^3.0))+(-27.0*(x^2.0)))+(9.0*(x^1.0)))+-1.0)
f'(x) =  ((((27.0*(3.0*(x^2.0)))+(-27.0*(2.0*(x^1.0))))+(9.0*(1.0*(x^0.0))))+0.0)
```

On to the 2nd derivative test. Implement a function `loc_xtrm_2nd_drv_test(fexpr)` that takes a representation of a second- or third-degree polynomial and uses the second-derivative test to find its local extrema. Write your code in `derivtest.py` included in the zip. You may assume that this function will only handle functions with 0, 1, or 2 local extrema. Like `loc_xtrm_1st_drv_test(fexpr)`, this function returns a list, possibly empty, of 2-tuples. The first element of each 2-tuple is the string `'min'` or `'max'` and the second element is a `point2d` object that represents an extreme point. Here is a test that finds the local extrema of $\frac{1}{4}x^2 - x + 2.0$.

```
def test_05():
    f1 = prod(mult1=make_const(1.0/4.0),
              mult2=make_pwr('x', 2.0))
    f2 = prod(mult1=make_const(-1.0),
              mult2=make_pwr('x', 1.0))
    f3 = plus(elt1=f1, elt2=f2)
    f4 = plus(elt1=f3, elt2=make_const(2.0))
    print f4
    xtrma = loc_xtrm_2nd_drv_test(f4)
    for i, j in xtrma:
        print i, str(j)
    assert len(xtrma) == 1 and \
           xtrma[0][1].get_x().get_val() == 2.0 and \
           xtrma[0][1].get_y().get_val() == 1.0
```

Below is the output of `test_05()` in the Python Shell.

```
>>> test_05()
(((0.25*(x^2.0))+(-1.0*(x^1.0)))+2.0)
min (2.0, 1.0)
```

Problem 3: Inflection Points (1 point)

We are ready to look for inflection points. Implement a function `find_infl_pnts(expr)` that takes a representation of a function and returns a list, possibly empty, of its inflection points represented as `point2d` objects. Write your code in `infl.py` included in the zip. The function `test_06()` below returns one inflection point of $x^3 - 3x^2 + 5$. Note that in `test_06`, the polynomial is represented as $x^3 - 3x^2 + 0x + 5$.

```
def test_06():
    f1 = make_pwr('x', 3.0)
    f2 = make_prod(make_const(-3.0), make_pwr('x', 2.0))
    f3 = make_plus(f1, f2)
    f4 = make_plus(f3, make_prod(make_const(0.0), make_pwr('x', 1.0)))
    poly = make_plus(f4, make_const(5.0))
    print poly
    infls = find_infl_pnts(poly)
    for ip in infls:
        print str(ip)
```

Here is the output in the Python Shell.

```
f(x) =  ((((x^3.0)+(-3.0*(x^2.0)))+(0.0*(x^1.0)))+5.0)
(1.0, 3.0)
```

What to Submit

Submit the files `poly12.py`, `derivtest.py`, and `infl.py` with your code via Canvas.

Happy Hacking!