Problem set #5

Release 0.1

Brian Granger, John Hunter and Fernando Pérez.

February 15, 2010

Contents

1 Univariate polynomials

This continues our previous exercise with univariate polynomials, but in a slightly more generalized formulation. Now, consider solving the equation:

$$x^2 \sum_i \frac{a_i}{a_i x + b_i} = k$$

where $\mathbf{a}$, $\mathbf{b}$ are arrays and k is a scalar. Again, this can be rewritten in terms of finding the roots of $R(x)$

$$R(x) \equiv x^2 P(x) - kQ(x)$$

outside of the roots of Q(x), where

$$P(x) = \sum_i a_i \prod_{j \neq i} \frac{1}{a_j x + b_j}$$

and

$$Q(x) = \prod_i a_i x + b_i$$

In this problem, you must compute this polynomial R(x) using both numpy and sympy, and then find its roots via numpy (sympy has limited support for numerical root finding).

First, use `np.poly1d` as before to construct $R(x)$, but now from a pair of arrays $\mathbf{a}$ and $\mathbf{b}$. Normalize $R(x)$ so its leading coefficient is 1 to avoid ambiguity.

As a validation reference, if the inputs are:

```
a = np.array([3.4, 4.5, 3.2])
b = np.array([2.1, 5.5, 4.5])
k = 0.5
```

then you should obtain the following $R(x)$ and roots:

```
Numpy, R(x) =
   4         3           2
1 x + 1.997 x + 0.5731 x - 0.557 x - 0.1769

Roots: [-1.32141936 -0.86720646 -0.30879126  0.50000423]
```

You must pay attention to the (unfortunate) fact that numpy's `poly1d` objects, when combined in binary operations like addition or multiplication with scalars obtained from numpy arrays, behave in rather counterintuitive ways. Observe:

```
In [88]: p = np.poly1d([2, 3])

In [89]: a = np.array([3.5, 4.5])

In [90]: p + 3.5
Out[90]: poly1d([ 2. ,  6.5])

In [91]: p + a[0]
Out[91]: poly1d([ 2. ,  6.5])

In [92]: a[0] + p
Out[92]: array([ 5.5,  6.5])
```

As you can see, addition is not commutative! There's an obscure reason for this behavior and an ongoing discussion on the numpy list on how to best resolve it. It should be noted that in newer versions of numpy (after February 2010), a new Polynomial class will be available that provides more sophisticated behavior than the basic poly1d shown here.

In the meantime, the simple solution is to call `float()` on all scalars before combining them with `poly1d` objects:

```
In [93]: p + 3.5
Out[93]: poly1d([ 2. ,  6.5])

In [94]: p + float(a[0])
Out[94]: poly1d([ 2. ,  6.5])

In [95]: float(a[0]) + p
Out[95]: poly1d([ 2. ,  6.5])
```

Next, use Sympy to construct a generic form of $R(x)$, that works both if the `(a, b, k)` are given, and if instead the user specifies only the number of desired terms in the construction. In this case, the returned value should be a symbolic polynomial. The signature of your function should be something like:

```
def sym_rpoly(na=None, nb=None, nk=None, nterms=None):
```

For example, for 1 and 2 terms you should obtain:

```
In [35]: sym_rpoly (nterms=1)
Out[35]: Poly(x**2 - k*x - b_0*k/a_0, x)

In [36]: sym_rpoly (nterms=2)
Out[36]: Poly(x**3 + (a_0*b_1 + a_1*b_0 - a_0*a_1*k)/(2*a_0*a_1)*x**2 +
(-a_0*b_1*k - a_1*b_0*k)/(2*a_0*a_1)*x - b_0*b_1*k/(2*a_0*a_1), x)
```

For this symbolic construction, you will find it useful to create arrays of symbolic elements. For this, you can use the following little utility in your code (this has been included in Sympy itself as of Feb 15 2010):

```
def symarray(shape, prefix=''):
    """Create a numpy ndarray of symbols (as an object array).

    The created symbols are named prefix_i1_i2_...  You should thus provide a
    non-empty prefix if you want your symbols to be unique for different output
    arrays, as Sympy symbols with identical names are the same object.

    Parameters
    ----------

    shape : int or tuple
      Shape of the created array.  If an int, the array is one-dimensional; for
      more than one dimension the shape must be a tuple.

    prefix : string
      A prefix prepended to the name of every symbol.

    Examples
    --------

    >>> symarray(3)
    array([_0, _1, _2], dtype=object)

    If you want multiple symarrays to contain distinct symbols, you *must*
    provide unique prefixes:
    >>> a = symarray(3)
    >>> b = symarray(3)
    >>> a[0] is b[0]
    True
    >>> a = symarray(3, 'a')
    >>> b = symarray(3, 'b')
    >>> a[0] is b[0]
```

```
    False

    Creating symarrays with a prefix:
    >>> symarray(3, 'a')
    array([a_0, a_1, a_2], dtype=object)

    For more than one dimension, the shape must be given as a tuple:
    >>> symarray((2,3), 'a')
    array([[a_0_0, a_0_1, a_0_2],
           [a_1_0, a_1_1, a_1_2]], dtype=object)
    >>> symarray((2,3,2), 'a')
    array([[[a_0_0_0, a_0_0_1],
            [a_0_1_0, a_0_1_1],
            [a_0_2_0, a_0_2_1]],
    <BLANKLINE>
           [[a_1_0_0, a_1_0_1],
            [a_1_1_0, a_1_1_1],
            [a_1_2_0, a_1_2_1]]], dtype=object)
    """
    arr = np.empty(shape, dtype=object)
    for index in np.ndindex(shape):
        arr[index] = sym.Symbol('%s_%s' % (prefix, '_'.join(map(str, index))))
    return arr
```

Finally, convert this symbolic object to a numerical `np.poly1d` object so you can validate your numpy solution against the symbolic one, and also compute its roots (sympy only has limited support for polynomial root finding).

Hint

In sympy, you will find useful the following functions and methods: `together()`, `fraction()` and `subs()`.

For the same inputs as above, sympy produces this polynomial object:

```
In [41]: sym_rpoly(a, b, k)
Out[41]: Poly(x**4 + 1.9974128540305*x**3 + 0.573052832244009*x**2 -
0.55703635620915*x - 0.176930147058824, x)
```

2 Newton's method

Illustrates: functions as first class objects, use of the scipy libraries.

Consider the problem of solving for t in

$$\int_o^t f(s)ds = u$$

where $f(s)$ is a monotonically increasing function of s and $u > 0$.

Think about how to cast this problem as a root-finding question and use Newton's method to solve it. The Scipy library includes an optimization package that contains a Newton-Raphson solver called `scipy.optimize.newton`. This solver can optionally take a known derivative for the function whose roots are being sought.

For this exercise, implement the solution for the test function

$$f(t) = t\sin^2(t),$$

using

$$u = \frac{1}{4}.$$