Problem set #5

Release 0.1

Brian Granger, John Hunter and Fernando Pérez.

February 17, 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 **a**, **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

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

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 **a** and **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 [2]: p = np.poly1d([2, 3])

In [3]: a = np.array([3.5, 4.5])

In [4]: p + 3.5
Out[4]: poly1d([ 2. ,  6.5])

In [5]: p + a[0]
Out[5]: poly1d([ 2. ,  6.5])

In [6]: a[0] + p
Out[6]: 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 [7]: p + 3.5
Out[7]: poly1d([ 2. ,  6.5])

In [8]: p + float(a[0])
Out[8]: poly1d([ 2. ,  6.5])
```

```
In [9]: float(a[0]) + p
Out[9]: 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. You can use the following signature and docstring to get started. Note that when a function has *all* of its arguments take a None default value, it means that in order to actually use it users must supply at least some of them, but they are mutually exclusive. In this case, you must provide *either* `(na, nb, nk)` as numerical arrays, *or* `nterms`:

```
def sym_rpoly(na=None, nb=None, nk=None, nterms=None):
    """Compute the R polynomial as defined above.

    Internally the construction of R is done symbolically. If the numerical
    variables (na, nb, nk) were supplied, these values are substituted at the
    end and a sympy.Poly object with numerical coefficients is returned.  If
    (na, nb, nk) are not given, then nterms *must* be given, and a symbolic
    answer is returned.

    Parameters
    ----------
    na : ndarray, optional
      Numerical array of 'a' coefficients.

    nb : ndarray, optional
      Numerical array of 'b' coefficients.

    k : float, optional
      Numerical value of k.

    nterms : int, optional.
      Number of terms. This is only used if na, nb and nk are *not* given, in
      which case a symbolic answer is returned with nterms total.

    Returns
    -------
    poly : sympy.Poly instance
      A univariate polynomial in x.

    Examples
    --------
    With only nterms, a symbolic polynomial is returned:
    >>> sym_rpoly(nterms=1)
    Poly(x**2 - k*x - b_0*k/a_0, x)

    But if numerical values are supplied, the output polynomial has numerical
    values:
    >>> sym_rpoly([2], [5], 3)
    Poly(x**2 - 3*x - 15/2, x)
    >>> sym_rpoly([1, 2], [4, 6], 1)
    Poly(x**3 + 3*x**2 - 7/2*x - 6, x)"""
```

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)
```

1.1 Solution

```
"""Univariate polynomial manipulations with Numpy and Sympy.

This code shows how to use univariate polynomials with Numpy and Sympy, based
on a simple problem that appears often in least-squares fitting contexts.

Consider solving the equation

  x^2 sum_i(a_i/(a_i x + b_i)) = k

where a, b are arrays and k is a scalar.  This can be rewritten as

  x^2 P(x)/Q(x) = k

and outside of the roots of Q(x), the solutions are given by the roots of the
polynomial R(x):

  R(x) := x^2 P(x) - k Q(x)
```

```
In this example, we 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).

Notes:

- We always normalize R(x) so its leading coefficient is 1.

- 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.
"""

#-----------------------------------------------------------------------------
# Imports
#-----------------------------------------------------------------------------
from __future__ import print_function # for doctests

import numpy as np
import sympy as sym

#-----------------------------------------------------------------------------
# Function definitions
#-----------------------------------------------------------------------------

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


def prod(seq):
    """Multiply the elements of a sequence.

    Note that the product accumulator is initialized to 1, so prod([])==1.

    Parameters
    ----------

    seq : iterable
      Any iterable sequence whose elements can be multiplied.

    Returns
    -------
    prod : scalar
      The result of the multiplication of all the elements in the input.

    Examples
    --------
    >>> prod([1,2,3])
    6
    >>> prod(['a',2,3])
    'aaaaaa'
    >>> prod([])
    1
```

```
    """
    return reduce(lambda x,y: x*y, seq, 1)


def spoly2npoly1d(poly):
    """Convert a univariate Sympy polynomial to a numpy.poly1d.

    The coefficients of the

    Parameters
    ----------
    poly : sympy.Poly instance
      A univariate sympy polynomial object.

    Returns
    -------
    poly1d : a numpy.poly1d instance

    Examples
    --------
    >>> import sympy
    >>> x = sympy.Symbol('x')
    >>> p = sympy.Poly(x**2 + 3*x + 1, x)
    >>> p_num = spoly2npoly1d(p)
    >>> p_num
    poly1d([ 1.,  3.,  1.])
    """

    if poly.is_multivariate:
        e = 'multivariate polynomials can not be converted to poly1d'
        raise ValueError(e)
    return np.poly1d(np.array(map(float, poly.coeffs), dtype=float))


def sym_rpoly(na=None, nb=None, nk=None, nterms=None):
    """Compute the R polynomial as defined above.

    Internally the construction of R is done symbolically. If the numerical
    variables (na, nb, nk) were supplied, these values are substituted at the
    end and a sympy.Poly object with numerical coefficients is returned.  If
    (na, nb, nk) are not given, then nterms *must* be given, and a symbolic
    answer is returned.

    Parameters
    ----------
    na : ndarray, optional
      Numerical array of 'a' coefficients.

    nb : ndarray, optional
      Numerical array of 'b' coefficients.

    k : float, optional
      Numerical value of k.
```

```
    nterms : int, optional.
      Number of terms. This is only used if na, nb and nk are *not* given, in
      which case a symbolic answer is returned with nterms total.

    Returns
    -------
    poly : sympy.Poly instance
      A univariate polynomial in x.

    Examples
    --------
    With only nterms, a symbolic polynomial is returned:
    >>> sym_rpoly(nterms=1)
    Poly(x**2 - k*x - b_0*k/a_0, x)

    But if numerical values are supplied, the output polynomial has numerical
    values:
    >>> sym_rpoly([2], [5], 3)
    Poly(x**2 - 3*x - 15/2, x)
    >>> sym_rpoly([1, 2], [4, 6], 1)
    Poly(x**3 + 3*x**2 - 7/2*x - 6, x)
    """
    if na is None and nterms is None:
        raise ValueError('You must provide either arrays or nterms')

    if nterms is None:
        na = np.asarray(na)
        nb = np.asarray(nb)
        # We have input arrays, return numerical answer
        if na.ndim > 1:
            raise ValueError('Input arrays must be one-dimensional')
        nterms = na.shape
        mode = 'num'
    else:
        # We have only a length, return symbolic answer.
        mode = 'sym'

    # Create symbolic variables
    k, x = sym.symbols('k x')
    a = symarray(nterms, 'a')
    b = symarray(nterms, 'b')

    # Construct polynomial symbolically
    t = [ ai/(ai*x+bi) for (ai, bi) in zip(a,b)]
    P, Q = sym.fraction(sym.together(sum(t)))
    Rs = sym.Poly(x**2*P -k*Q, x)

    if mode == 'num':
        # Substitute coefficients for numerical values
        Rs = Rs.subs([(k, nk)] + zip(a, na) + zip(b, nb))

    # Normalize
```

```
    Rs /= Rs.lead_coeff
    return Rs


def num_rpoly(a, b, k):
    """Compute the R polynomial as defined above using numpy.

    Parameters
    ----------
    na : ndarray

    nb : ndarray

    k : float

    Returns
    -------
    poly : numpy.poly1d instance
      A univariate polynomial in x.

    Examples
    --------
    >>> num_rpoly([2.0], [5.0], 3.0)
    poly1d([ 1. , -3. , -7.5])

    Printing numpy's poly1d objects gives a more familiar readout:
    >>> print(num_rpoly([2.0], [5.0], 3.0))
       2
    1 x - 3 x - 7.5

    >>> num_rpoly([1.0, 2.0], [4.0, 6.0], 1.0)
    poly1d([ 1. ,  3. , -3.5, -6. ])
    >>> print(num_rpoly([1.0, 2.0], [4.0, 6.0], 1.0))
       3     2
    1 x + 3 x - 3.5 x - 6
    """

    a = np.asarray(a)
    b = np.asarray(b)
    x2 = np.poly1d([1, 0, 0])
    Qr = -b/a
    Q = np.poly1d(Qr, r=True)
    denoms = [ np.poly1d([1, -di]) for di in Qr ]
    P = sum( (prod(dj for (j, dj) in enumerate(denoms) if j != i ))
                   for i in range(len(a)) )
    R = x2*P - k*Q
    R /= R.coeffs[0]
    return R


def num_rpoly2(a, b, k):
    """Compute the R polynomial as defined above using numpy.
```

```
    This is just an alternate implementation which keeps the a factors in the
    numerators, it should be identical to num_rpoly and is kept here only to
    illustrate some aspects of numpy poly1d objects.


    Parameters
    ----------
    na : ndarray

    nb : ndarray

    k : float

    Returns
    -------
    poly : numpy.poly1d instance
      A univariate polynomial in x.

    Examples
    --------
    >>> num_rpoly([2.0], [5.0], 3.0)
    poly1d([ 1. , -3. , -7.5])

    Printing numpy's poly1d objects gives a more familiar readout:
    >>> print(num_rpoly([2.0], [5.0], 3.0))
       2
    1 x - 3 x - 7.5

    >>> num_rpoly([1.0, 2.0], [4.0, 6.0], 1.0)
    poly1d([ 1. ,  3. , -3.5, -6. ])
    >>> print(num_rpoly([1.0, 2.0], [4.0, 6.0], 1.0))
       3     2
    1 x + 3 x - 3.5 x - 6
    """
    a = np.asarray(a)
    b = np.asarray(b)
    x2 = np.poly1d([1, 0, 0])
    # When building the polynomial Q out of roots, by default numpy keeps the
    # leading coefficient to 1, so we need to multiply back by the product of
    # all the a's
    Q = np.poly1d(-b/a, r=True)
    Q *= a.prod()
    # We need the individual components of the denominator Q
    denoms = [ np.poly1d([ai, bi]) for (ai, bi) in zip(a, b) ]
    # With these, we can compute the numerator P.  Note that we must call
    # float() on each ai, because multiplying a poly1d by a numpy scalar
    # produces odd results.  I consider this a bug in numpy.
    P  = sum( float(ai)* prod(dj for (j, dj) in enumerate(denoms) if j != i)
              for (i, ai) in enumerate(a) )
    # Now the computation proceeds identically.
    R = x2*P - k*Q
    R /= R.coeffs[0]
    return R
```

```
def test_compare_with_sympy():
    """Use the symbolic solution as a reference for the numpy ones."""
    import numpy.testing as npt

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

    Rn = num_rpoly(a, b, k)
    Rn2 = num_rpoly2(a, b, k)
    Rs = sym_rpoly(a, b, k)
    Rns = spoly2npoly1d(Rs)

    # Check that both numerical implementations coincide with the symbolic one
    npt.assert_almost_equal(Rn.coeffs, Rns.coeffs, 15)
    npt.assert_almost_equal(Rn2.coeffs, Rns.coeffs, 15)


#-----------------------------------------------------------------------------
# Main script
#-----------------------------------------------------------------------------

if __name__ == '__main__':
    # Simple illustrative example
    a = np.array([3.4, 4.5, 3.2])
    b = np.array([2.1, 5.5, 4.5])
    k = 0.5

    Rn = num_rpoly(a, b, k)
    Rn2 = num_rpoly2(a, b, k)
    Rs = sym_rpoly(a, b, k)
    Rns = spoly2npoly1d(Rs)

    print('Numpy, R(x) =', Rn, sep='\n')
    print('Sympy, R(x) =', Rs, sep='\n')
    roots = Rn.roots
    roots.sort()
    print()
    print('Roots:', roots)
```

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

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

2.1 Solution

This problem can be simply solved if seen as a root finding question. Let

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

then we just need to find the root for $g(t)$, which is guaranteed to be unique given the conditions above. In this case the derivative can be trivially computed in exact form, and we can then pass them to the Newton-Raphson solver.

The following code implements this solution:

```
#!/usr/bin/env python
"""Root finding using SciPy's Newton's method routines.
"""

from math import sin

from scipy.integrate import quad
from scipy.optimize import newton

# test input function
def f(t):
    return t*(sin(t))**2

def g(t):
    "Exact form for g by integrating f(t)"
    u = 0.25
    return .25*(t**2-t*sin(2*t)+(sin(t))**2)-u

def gn(t):
    "g(t) obtained by numerical integration"
    u = 0.25
    return quad(f,0.0,t)[0] - u
```

```
# main
tguess = 10.0

print '"Exact" solution (knowing the analytical form of the integral)'
t0 = newton(g,tguess,f)
print "t0, g(t0) =",t0,g(t0)

print
print "Solution using the numerical integration technique"
t1 = newton(gn,tguess,f)
print "t1, g(t1) =",t1,g(t1)

print
print "To six digits, the answer in this case is t==1.06601."
```