Polynomials

Ernesto P. Adorio

1 Basic Operations on Polynomials

A polynomial function f of degree n in the indeterminate x is written as

$$f(x) = \sum_{i=0}^{n} c_{n-i} x^i$$

where the leading coefficient associated with x^n is c_0. If $c_0 = 1$, we call f a monic polynomial. If the degree of f is respectively 1,2,3,4 we call f correspondingly a linear, quadratic, cubic or a quartic polynomial.

The file `poly.py` is a Python module which performs processing with polynomials. Internally, the polynomial is represented by an attribute degree and an array for storing the coefficients. The length of the array is at least $degree + 1$.

A polynomial object is created by the `poly.poly` function which expects a list of coefficients. The '+', '-', '*' signs are overloaded to indicate the sum, difference and product of polynomials. As an example of usage of the poly module, let $A = x^2 + 5x + 6$ and $B = x^3 + 3x^2 + 3x + 1$ be two polynomials. Now start a Python session and follow the commands as shown here.

```
>>> import poly
>>> A = poly.poly([1,5,6])
>>> A
<poly.poly instance at 0x402e848c>
>>> print A
poly[1, 5, 6]
>>> B = poly.poly([1,3,3,1])
>>> print B
poly[1, 3, 3, 1]
>>> print A - B
poly[-1, -2, 2, 5]
>>> print A + B
poly[1, 4, 8, 7]
>>> print A * B
poly[1, 8, 24, 34, 23, 6]
>>>
```

If you cannot import the `poly.py` module, then the directory where the module is may not be in the python search path. You can add the current directory to the searh path by the following:

```
>>> import sys
>>> sys.path.append('.')    # Current directory is in python search path.
>>> import poly             # You should succeed this time.
```

The value of the polynomial at a point x is given by the Eval() function. The derivative of a polynomial can be obtained by the `Deriv` method and the integral of a polynomial likewise can be obtained by the `Integ` method. Let us determine the dertivative of the polynomial A created in the above session and the integral with a constant of integration 1.

```
>>> A.Eval(1.0)
12.0
>>> D = A.Deriv(); print D
poly[2, 5]
>>> I = A.Integ(1); print I
poly[0.33333333333333331, 2.5, 6.0, 1]
```

The value of the polynomial A at the point 1.0 is 12. Note that the output derivative polynomial, $2x^+5$ has a degree 1 less and the integral polynomial $\frac{1}{3}x^3 + 2\frac{1}{2}x^2 + 6x + 1$ has a degree 1 more than the given polynomial.

2 Zeroes or Roots of Polynomials

To compute for the roots of a polynomial, we call the `Zero` or `Roots` routine. This routine uses Muller's method which can also be applied for finding zeroes or roots of an analytic function. Our Python code is based on Conte and deBoor's Fortran 77 program.

```
>>> print A.Zeroes()
[(-2+0j), (-2.9999999999999964+0j)]
>>> print B.Zeroes()
[(-0.99988389120391696+0.00036873616407736762j),
(-0.99973848366928009-0.00028458368319977077j),
(-1.000377625418265-8.4152704043285412e-05j)]
>>>
```

From the above results, we see that the routine Zeroes obtains accurate approximation to the exact roots -2 and -3 of A. Given a list of roots of a polynomial, we can build up the polynomial with the given roots. The method of calculation is attributed to Vieta. Let's try to see if the coefficients of B can be recovered from its roots.

```
>>> C = B.Build(B.Zeroes())
>>> print C
poly[1, (3.000000000291462+2.2316568856544858e-10j),
(3.0000000003643787+2.7789984456151866e-10j),
(1.0000000001186751+9.005693796147171e-11j)]
>>>
```

The zermuller routine has been called with a fixed root tolerance ZTOL and function tolerance FTOL value of 1.0e-10 for maximum of 50 iterations. We will rewrite in the immediate future the code to accept varying specified tolerance parameters.

3 Normalizing a polynomial

It is helpful to provide a routine to fix leading coefficient values to zeroes and to set imaginary or real components of other coefficients to zero based on an input tolerance. The `Normalize()` function does this. Here is an illustration for the polynomial C.

```
>>> C.Normalize(1.0e-6)
>>> print C
poly[1, 3.000000000291462, 3.0000000003643787, 1.0000000001186751]
>>>
```

Note that we have not recovered exactly the original coefficients of B. Another routine which may be handy is the routine `Real()` which will drop any imaginary components (even if they are significant) of the coefficients. The corresponding procedural oriented routine is `makereal()` to avoid name clash with the cmath real routine. These routines require that the input argument must be mutable.

4 Summary and source codes

Here is a summary of the functions available in the `poly.py` module.

Build(roots)	Build up polynomial from roots
Degree()	Returns the degree of the polynomial (integer)
Deriv()	Returns the derivative of the polynomial
Eval(x)	Evaluates the polynomial at point x
Integ()	Returns the integral of the polynomial
Real()	Drops any imaginary components of the coefficients
Roots()	Returns the zeros of the polynomial

The poly.py module is written with both procedural oriented and class oriented routines. The procedural routines can be used for polynomials expressed

as ordinary arrays. The routines are basic. A full polynomial library is planned in the future. It should include polynomial interpolation and extrapolation including least squares and min absolute deviation sum and rational polynomials.

```
"""
poly.py --  a basic object based polynomial library

author/programmer
   Ernesto P. Adorio

versions
   V0.1.0   01.21.2004  First Release

remarks
   We first create procedural oriented methods and create
   an object-based wrapper around them.
"""
import copy
from math import *
from cmath import *

# ##########################################################################
# Polynomial primitives
# ##########################################################################
def quadeval(a, b, c, x):
   """
   Returns the value of the quadratic at x.
   """
   return (a * x + b) * x + c

def quadroots(a, b, c):
   """
   Solves for the roots of a x^2 + b x + c = 0.
   """
   r = sqrt(b**2 - 4 * a * c)
   d = 0.5 / a
   return [(-b + r) * d, (-b -r) * d]

def quadbuild(r1, r2):
   """
   Builds up the coefficients of the quadratic polynomial.
   """
   return [1.0, -(r1 + r2), r1 * r2]

def eval(coefs, x):
   """
```

```
   Evaluates a polynomial with coefs and degree n.
   """
   sum = 0
   for i in range(len(coefs)):
      sum = sum * x + coefs[i]
   return sum


def evaldeg(coefs, deg, x):
   """
   Evaluates a polynomial with coefs and degree n.
   This variant is needed when the length of
   coefs may not equal the degree + 1 of the polynomial
   """
   sum = 0.0
   for i in range(deg+1):
      sum = sum * x + coefs[i]
   return sum

def add(p1, p2):
   """
   Returns the sum of two polynomials, p1, p2.
   """
   l1 = len(p1)
   l2 = len(p2)
   d  = max(l1, l2)
   p  = [0] * d
   for x in range(d-1, -1, -1):
      if l1 > 0:
         l1 = l1 -1
         p[x] += p1[l1]
      if l2 > 0:
         l2 = l2 - 1
         p[x] += p2[l2]
   return p


def sub(p1, p2):
   """
   Returns the difference of two polynomials p1-p2,
   """
   l1 = len(p1)
   l2 = len(p2)
   d  = max(l1, l2)
   p  = [0] * d
   for x in range(d-1, -1, -1):
```

```
      if l1 > 0:
         l1 = l1 -1
         p[x] += p1[l1]
      if l2 > 0:
         l2 = l2 - 1
         p[x] -= p2[l2]
   return p

def mul(p1, p2):
   """
   Returns the product  (p1 p2).
   """
   l1 = len(p1)
   l2 = len(p2)
   d  = l1 + l2 - 1
   if l1 < 1 or l2 < 1:
      return []
   p = [0] * d
   for i in range(l1):
      for j in range(l2):
         p[i+j] += p1[i] * p2[j]
   return p


def mulx(p, x):
   """
   Multiplies all coefficients of p by x.
   """
   pp = p * 1
   for i in range(len(p)):
      pp[i] *= x
   return pp


def shift(p, k):
   """
   Changes the degree of the polynomial by k.
   """
   pp = p*1
   if k > 0:
      while (k > 0):
         pp.append(0)
         k = k - 1
   elif k < 0:
      k = -k
      while k > 0 and pp != []:
```

```
         del pp[-1]
         k = k - 1
   return pp

def makereal(p, k)
   """
   Drops any imaginary component of the coefficients.
   """
   for i in range(len(p)):
      if type(p[i])  == type(1.0 + 1j):
         p[i] = p[i].real


def idiv(p, q):
   """
   Divides p by q to return the (quotient, remainder).
   WARNING: no checking for integer coefficients.
   """
   pdeg = len(p) -1
   qdeg = len(q) -1
   if (qdeg > pdeg):
      return [], p

   d  = []
   pp = p * 1
   while len(pp) >= len(q):
      f = 1.0 * pp[0] / q[0]
      d.append(f)
      pp =  sub(pp, shift(mulx(q, f), len(pp) - len(q)))
      print "f=",f, "d=",d, "pp=", pp
      del pp[0]
   return d, pp

def normalize(coeffs, ztol):
   """
   Returns a polynomial where the leading coefficient is not zero.
   """
   N = copy.deepcopy(coeffs)
   for i in range(len(N)):
      if abs(N[i]) > ztol:
         # Now perform tests on complex components
         for j in range(i, len(N)):
            if type(N[j]) == type(1.0 + 0.0j):
               if abs(N[j].real) < ztol:
                  N[j] = complex(0, N[j].imag)
               if abs(N[j].imag) < ztol:
```

```
                N[j] = N[j].real
        return N[i:]
    return N



def build(roots):
    """
    Builds up the coefficients of polynomials using Viete's formula.
    """
    degree = len(roots) + 1
    coeffs = [0]*(degree)
    coeffs[0] = 1.0
    for i in range(1, degree):
        r = roots[i-1]
        for j in range(i-1, 0, -1):
            coeffs[j] = coeffs[j-1] - r * coeffs[j]
        coeffs[0] = -coeffs[0] * r
        coeffs[i] = 1
    coeffs.reverse()
    return coeffs

def deriv(z):
    """
    Computes for the derivative of the polynomial.
    """
    deg = len(z) - 1
    zp  = [0]*(deg)
    for i in range(deg):
        zp[i] = (deg - i) * z[i]
    return zp



def integ(z, konst):
    """
    Computes for the integral of a polynomial.
    """
    deg = len(z)
    zp  = [0]*(deg+1)
    for i in range(deg):
        zp[i] = 1.0*z[i] / (deg - i)
    zp[deg] = konst
    return zp
```

```
def roots(coeffs, nroots, initval = 0.5, ztol = 1.0e-10, ftol = 1.0e-10,
          maxiter = 100, wantreal = False):
   """
   Roots of polynomials using Muller's method.
   """
   degree = len(coeffs) - 1
   if nroots <= 0 or degree < 1:
      return []

   if nroots > degree:
      nroots = degree

   if degree == 1:
      # print 'Length of array = ', len(coeffs)
      return [coeffs[1]/coeffs[0]]

   # print 'degree = ', degree, ' nroots = ', nroots
   def f(z):
      return evaldeg(coeffs, degree, z)

   return zermuller(f, nroots, initval, ztol, ftol, maxiter, wantreal)


def deflate(f, z, kroots, roots):
   """
   f                 Input: complex<double> function whose root is desired
   z                 Input: test root
   kroots            Input: number of roots found so far
   roots             Input/Output: saved array of roots
   undeflate         Output: undeflated function value
   """

   undeflate = t = f(z)
   for i in range(kroots):
      denom = z - roots[i]
      # avoid  small divisors
      while (abs(denom) < 1e-12):
         denom += 1.0e-8
      t = t / denom
   return undeflate, t


def zermuller(f, nroots, xinit, ztol, ftol, maxiter,  wantreal):
"""
        Zero finding of analytic functions by Muller's method
Reference: Conte and de Boor[1980], pp. 120-122
```

```
"""
nmaxiter = 0
retflag  = 0
roots = [0]*(nroots)
for j in range(nroots):
x1  = xinit
x0  = x1 - 0.5
x2  = x1 + 0.5
undeflate , f0  = deflate(f, x0, j, roots)
undeflate , f1  = deflate(f, x1, j, roots)
undeflate , f2  = deflate(f, x2, j, roots)
h21 = x2 - x1
h10 = x1 - x0
f21 = (f2 - f1) / h21
f10 = (f1 - f0) / h10
for i in range(maxiter):
f210 = (f21 - f10) / (h21+h10)
b    = f21 + h21 * f210
t    = (b*b- 4.0 * f2 * f210)
if (wantreal):
if type(t) == type(1j):
    if (t.real < 0.0):
    t = 0.0
    else:
    t = t.real
elif t < 0.0:
    t = 0.0
Q = sqrt(t)
D = b + Q
E = b - Q

if (abs(D) < abs(E)):
D = E
if (abs(D) <= ztol):
xm = 2 * x2 - x1
hm = xm - x2
else:
hm = -2.0 * f2 / D
             xm = x2 + hm
undeflate, fm = deflate(f, xm, j, roots)
absfm = abs(fm)           #  Divergence control
absf2 = 100. * abs(f2)
if (absf2 > ztol and absfm >= absf2):
hm    = hm * 0.5
xm    = x2 + hm
fm    = f(xm)
```

```
absfm = abs(fm)
if (abs(undeflate) <= ftol or abs(hm) <= ztol):
nmaxiter = i
retflag  = 0
break
x0  = x1
x1  = x2
x2  = xm
f0  = f1
f1  = f2
f2  = fm
h10 = h21
h21 = hm
f10 = f21
f21 = (f2 - f1) / h21
if (i >= maxiter) :
nmaxiter = i
retflag  = 2
break
roots[j] = xinit = xm
xinit    = xinit + 0.85
maxiter  = nmaxiter
return roots

# ##############################################################################
# Polynomial class
# ##############################################################################
class poly:
    """Module for computing with polynomials."""
    def __init__(self, c):
       """Initialize the coefficient array and the degree."""
       self.c   = copy.deepcopy(c)
       self.deg = max(0, len(c) - 1)

    def __str__(self):
       """String representation of polynomial"""
       return self.c.__str__();

    def __add__(self, other):
       """Adds two polynomials"""
       return poly(add(self.c, other.c))

    def __sub__(self, other):
       """Difference of two polynomials"""
       return poly(sub(self.c, other.c))
```

```
def __mul__(self, other):
   """Product of two polynomials"""
   return poly(mul(self.c, other.c))

def __div__(self, other):
   """
   Quotient of two polynomials in (quotient,remainder) form.
   WARNING: No verification for integer valued coefficients.
   """
   quotient, remainder = idiv(self.c, other.c)
   return poly(quotient), poly(remainder)

def Degree(self):
   return self.deg

def Normalize(self, ztol=1.0e-10):
   """
   Adjusts the degree and coefficients of the polynomial for
   leading zeroes and near zero real and imag components of
   coefficients.
   """
   c = normalize(self.c, ztol)
   self.c   = c
   self.deg = max(0, len(c)-1)

def Eval(self, x):
   """Evaluates the polynomial at x"""
   return eval(self.c, x)

def Roots(self, nroots, initval = 0.5, ztol = 1.0e-10, ftol = 1.0e-10,
          maxiter = 100, wantreal = False):
   """Returns the zeroes of polynomial"""
   print 'DEBUG: about to call roots()'
   return roots(self.c, nroots, initval, ztol, ftol, maxiter,  wantreal)

def Zeroes(self):
   """Same as Roots()"""
   return roots(self.c, nroots, initval, ztol, ftol, maxiter,  wantreal)

def Build(self, roots):
   """Builds up polynomial coefficients"""
   return poly(build(roots))

def Deriv(self):
   """Returns the derivative polynomial"""
   return poly(deriv(self.c))
```

```
def Integ(self, konst):
   """Returns the integral polynomial"""
   return poly(integ(self.c, konst))

def Real(self):
   """Drops any imaginary component of the coefficients of the polynomial"""
   for i in range(len(self.c)):
      if type(self.c[i])  == type(1.0 + 1j):
         self.c[i] = self.c[i].real
```