Root finding – Part 2

Introduction to Numerical Problem Solving, Spring 2017

Sakari Lukkarinen

Helsinki Metropolia University of Applied Sciences

Content

- Zeros of polynomials
  - numpy.polynomial
    - polyval()
    - polyroots()
    - polyfromroots()
    - Polynomial calculus and algebra
- Systems of nonlinear equations
  - scipy.optimize
    - root()
    - fsolve()

Polynomials (numpy.polynomial)

SciPy.org Sponsored By ENTHOUGHT

Scipy.org | Docs | NumPy v1.12 Manual | NumPy Reference | Routines | Polynomials | Polynomial Package

Polynomial Module (numpy.polynomial.polynomial)

New in version 1.4.0.

This module provides a number of objects (mostly functions) useful for dealing with Polynomial series, including a Polynomial class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its “parent” sub-package, numpy.polynomial).

Polynomial Class

Polynomial(coef[, domain, window])	A power series class.

Basics

polyval(x, c[, tensor])	Evaluate a polynomial at points x.
polyval2d(x, y, c)	Evaluate a 2-D polynomial at points (x, y).
polyval3d(x, y, z, c)	Evaluate a 3-D polynomial at points (x, y, z).
polygrid2d(x, y, c)	Evaluate a 2-D polynomial on the Cartesian product of x and y.
polygrid3d(x, y, z, c)	Evaluate a 3-D polynomial on the Cartesian product of x, y and z.
polyroots(c)	Compute the roots of a polynomial.
polyfromroots(roots)	Generate a monic polynomial with given roots.
polyvalfromroots(x, r[, tensor])	Evaluate a polynomial specified by its roots at points x.

Fitting

polyfit(x, y, deg[, rcond, full, w])	Least-squares fit of a polynomial to data.
polyvander(x, deg)	Vandermonde matrix of given degree.
polyvander2d(x, y, deg)	Pseudo-Vandermonde matrix of given degrees.
polyvander3d(x, y, z, deg)	Pseudo-Vandermonde matrix of given degrees.

Polyval

Evaluate a polynomial at points x.

If *c* is of length *n* + *1*, this function returns the value

$$p(x) = c_0 + c_1 * x + ... + c_n * x^n$$

Examples

```
>>> from numpy.polynomial.polynomial import polyval
>>> polyval(1, [1,2,3])
6.0
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> polyval(a, [1,2,3])
array([[  1.,   6.],
       [ 17.,  34.]])
```

How to evaluate ? $p(x) = 1 + 2x + 3x^3$

```
In [55]: x0 = 1.0
         c = [1, 2, 3]
         polyval(x0, c)
Out[55]: 6.0
```

```
In [56]: def f(x):
             return 1 + 2*x + 3*x**3
         f(x0)
Out[56]: 6.0
```

```
In [57]: def f(x, c):
             return c[0] + c[1]*x + c[2]*x**2
         f(x0, c)
Out[57]: 6.0
```

Finding the roots of polynomial

Find the roots of polynomial $f(x) = -6 + 11x - 6x^2 + x^3$

```
In [63]: polyroots([-6, 11., -6, 1.])
Out[63]: array([ 1.,  2.,  3.])
```

$$f(x) = (x - 1)(x - 2)(x - 3)$$

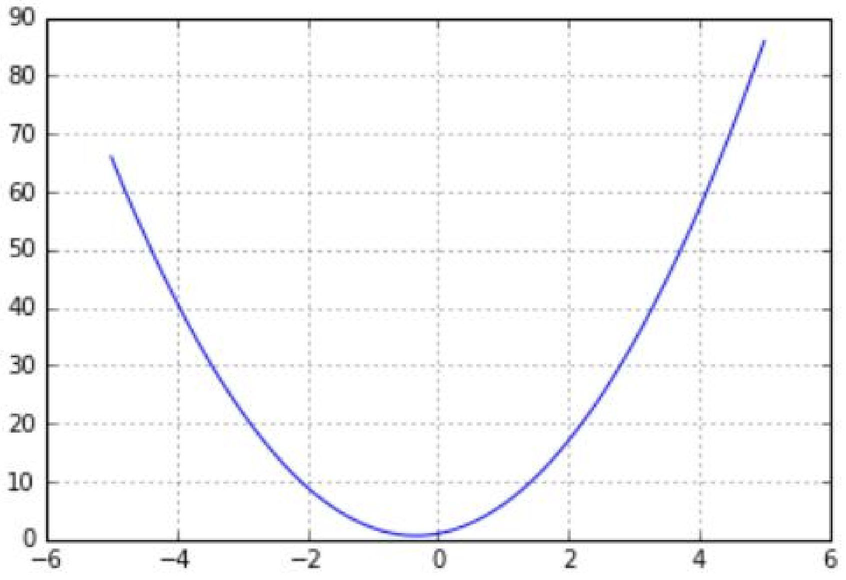
Convert back to polynomial coefficients

```
In [64]: polyfromroots([1., 2., 3.])
Out[64]: array([ -6.,  11.,  -6.,   1.])
```

What if the polynomial is always positive?

Draw a polynomial function $f(x) = 1 + 2x + 3x^2$ and check if it has any real roots.

In [65]:
```
x = linspace(-5, 5, 1000)
y = polyval(x, [1, 2, 3])
plot(x, y)
grid()
```

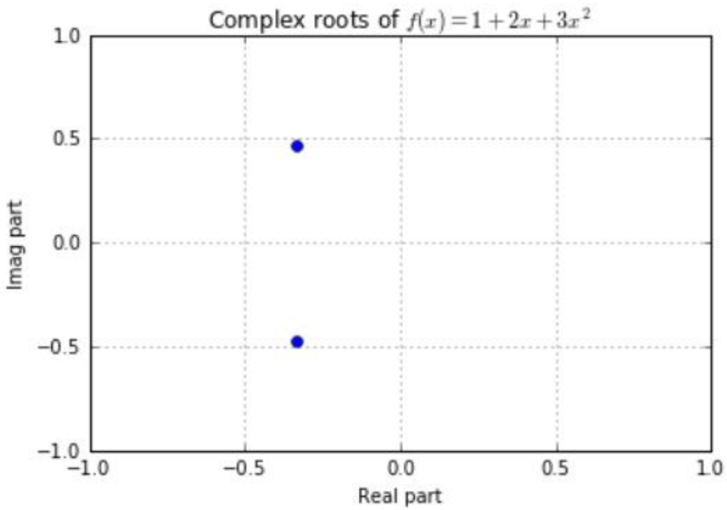


Complex valued roots

In [67]:
```
r = polyroots([1., 2., 3.])
r
```

Out[67]:
```
array([-0.33333333-0.47140452j, -0.33333333+0.47140452j])
```

In [77]:
```
plot(real(r), imag(r), 'o')
xlim((-1, 1))
ylim((-1, 1))
xlabel('Real part')
ylabel('Imag part')
title('Complex roots of $f(x) = 1 + 2x + 3x^2$');
grid()
```

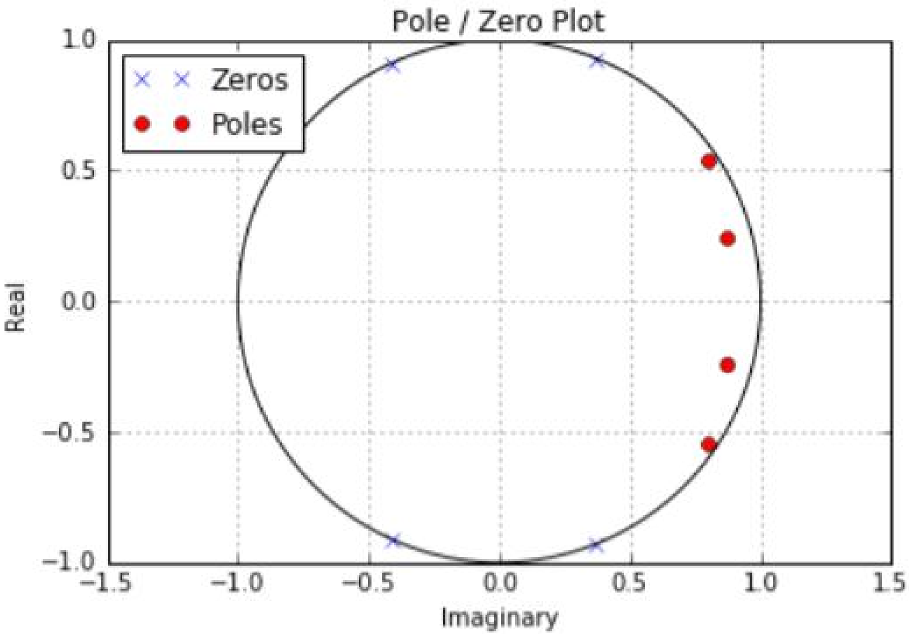


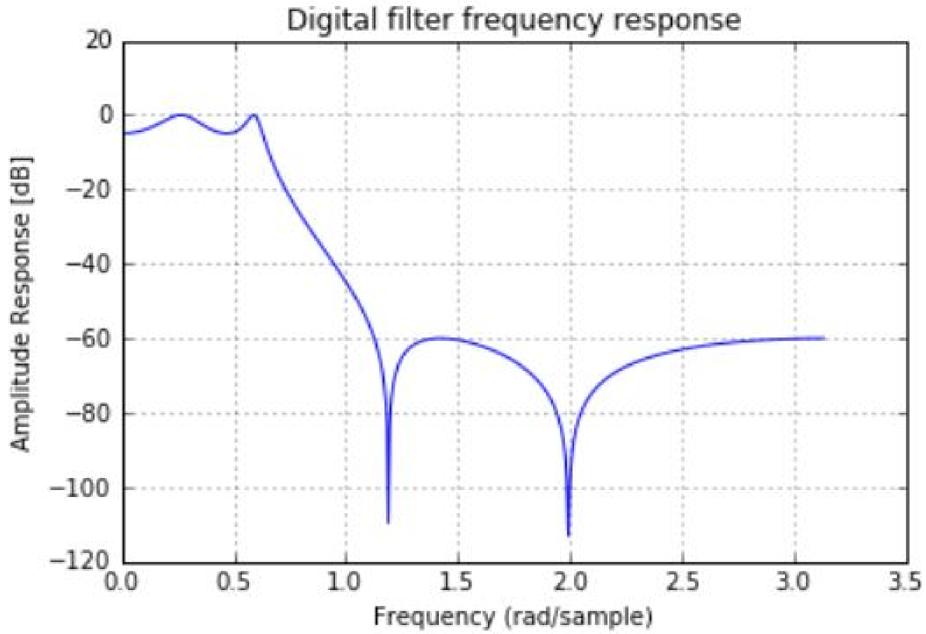
Application: pole-zero analysis

Continuous-time systems [edit]

In general, a rational transfer function for a continuous-time LTI system has the form:

$$H(s) = \frac{B(s)}{A(s)} = \frac{\displaystyle\sum_{m=0}^{M} b_m s^m}{s^N + \displaystyle\sum_{n=0}^{N-1} a_n s^n} = \frac{b_0 + b_1 s + b_2 s^2 + \cdots + b_M s^M}{a_0 + a_1 s + a_2 s^2 + \cdots + a_{(N-1)} s^{(N-1)} + s^N}$$





Polynomial calculus and algebra

Calculus

polyder(c[, m, scl, axis])	Differentiate a polynomial.
polyint(c[, m, k, lbnd, scl, axis])	Integrate a polynomial.

Algebra

polyadd(c1, c2)	Add one polynomial to another.
polysub(c1, c2)	Subtract one polynomial from another.
polymul(c1, c2)	Multiply one polynomial by another.
polymulx(c)	Multiply a polynomial by x.
polydiv(c1, c2)	Divide one polynomial by another.
polypow(c, pow[, maxpower])	Raise a polynomial to a power.

Roots of nonlinear equation systems

Root finding (scipy.optimize)

Scalar functions

brentq(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in a bracketing interval using Brent's method.
brenth(f, a, b[, args, xtol, rtol, maxiter, ...])	Find root of f in [a,b].
ridder(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in an interval.
bisect(f, a, b[, args, xtol, rtol, maxiter, ...])	Find root of a function within an interval.
newton(func, x0[, fprime, args, tol, ...])	Find a zero using the Newton-Raphson or secant method.

Fixed point finding:

fixed_point(func, x0[, args, xtol, maxiter, ...])	Find a fixed point of the function.

Multidimensional

General nonlinear solvers:

root(fun, x0[, args, method, jac, tol, ...])	Find a root of a vector function.
fsolve(func, x0[, args, fprime, ...])	Find the roots of a function.
broyden1(F, xin[, iter, alpha, ...])	Find a root of a function, using Broyden's first Jacobian approximation.
broyden2(F, xin[, iter, alpha, ...])	Find a root of a function, using Broyden's second Jacobian approximation.

scipy.optimize.root

scipy.optimize.root(*fun*, *x0*, *args=()*, *method='hybr'*, *jac=None*, *tol=None*, *callback=None*, *options=None*)

Find a root of a vector function.

Parameters:

fun : *callable*

A vector function to find a root of.

x0 : *ndarray*

Initial guess.

args : *tuple, optional*

Extra arguments passed to the objective function and its Jacobian.

method : *str, optional*

Type of solver. Should be one of

- 'hybr' *(see here)*
- 'lm' *(see here)*
- 'broyden1' *(see here)*
- 'broyden2' *(see here)*
- 'anderson' *(see here)*
- 'linearmixing' *(see here)*
- 'diagbroyden' *(see here)*
- 'excitingmixing' *(see here)*
- 'krylov' *(see here)*
- 'df-sane' *(see here)*

scipy.optimize.fsolve¶

scipy.optimize.fsolve(*func*, *x0*, *args=()*, *fprime=None*, *full_output=0*, *col_deriv=0*, *xtol=1.49012e-08*, *maxfev=0*, *band=None*, *epsfcn=None*, *factor=100*, *diag=None***)** [source]

Find the roots of a function.

Return the roots of the (non-linear) equations defined by `func(x) = 0` given a starting estimate.

Parameters:

func : *callable* `f(x, *args)`

A function that takes at least one (possibly vector) argument.

x0 : *ndarray*

The starting estimate for the roots of `func(x) = 0`.

args : *tuple, optional*

Any extra arguments to *func*.

fprime : *callable(x), optional*

A function to compute the Jacobian of *func* with derivatives across the rows. By default, the Jacobian will be estimated.

full_output : *bool, optional*

If True, return optional outputs.

Example

Problem

Determine the points of intersection between the circle $x^2 + y^2 = 3$ and the hyperbola $xy = 1$.

Solution

First we reformulate the equations so that we have zeros on the right sides:

$$\begin{aligned} x^2 + y^2 - 3 &= 0 \\ xy &= 1 \end{aligned}$$

Next we rename the variables $x_0 = x$ and $x_1 = y$, and write a vector function

In [79]:
```
def f(x):
    f1 = x[0]**2 + x[1]**2 - 3
    f2 = x[0]*x[1] - 1
    return [f1, f2]
```

Example (continued)

Now we can apply `root` function from `scipy.optimize` module and using initial guess $x = x[0] = 0.5$ and $y = x[1] = 1.5$.

In [82]:
```
from scipy.optimize import root
x0 = [0.5, 1.5]
r = root(f, x0)
r
```

Out[82]:
```
    fjac: array([[-0.58381252, -0.8118885 ],
       [ 0.8118885 , -0.58381252]])
     fun: array([  0.00000000e+00,  -2.22044605e-16])
 message: 'The solution converged.'
    nfev: 8
     qtf: array([ -9.20708768e-10,   4.83706707e-10])
       r: array([-1.9231426 , -2.50318187,  2.32543004])
  status: 1
 success: True
       x: array([ 0.61803399,  1.61803399])
```

In [83]:
```
# Get the solution
r.x
```

Out[83]:
```
array([ 0.61803399,  1.61803399])
```

Example: Graphical presentation

In [83]:
```
# Get the solution
r.x
```

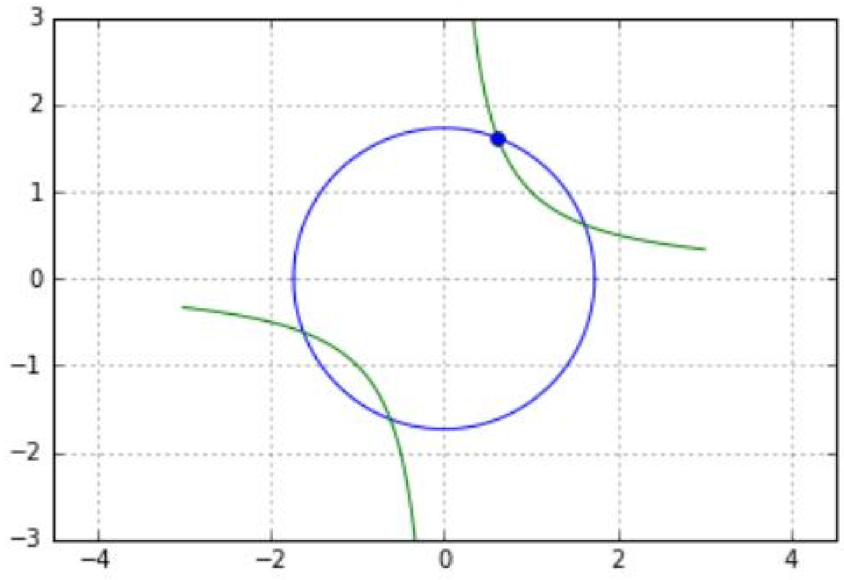
Out[83]: `array([ 0.61803399,  1.61803399])`

In [98]:
```
x = linspace(-sqrt(3), sqrt(3), 1000)
y1 = sqrt(3 - x**2)
y2 = -y1
plot(x, y1, 'b')
plot(x, y2, 'b')
x = linspace(-3, -0.1, 500)
y3 = 1/x
plot(x, y3, 'g')
x = linspace(0.1, 3, 500)
y3 = 1/x
plot(x, y3, 'g')

axis('equal')
ylim((-3, 3))
grid()

plot(r.x[0], r.x[1], 'o')
```

<matplotlib.lines.Line2D at 0x1c4a58d0>]



Which values converge and where?

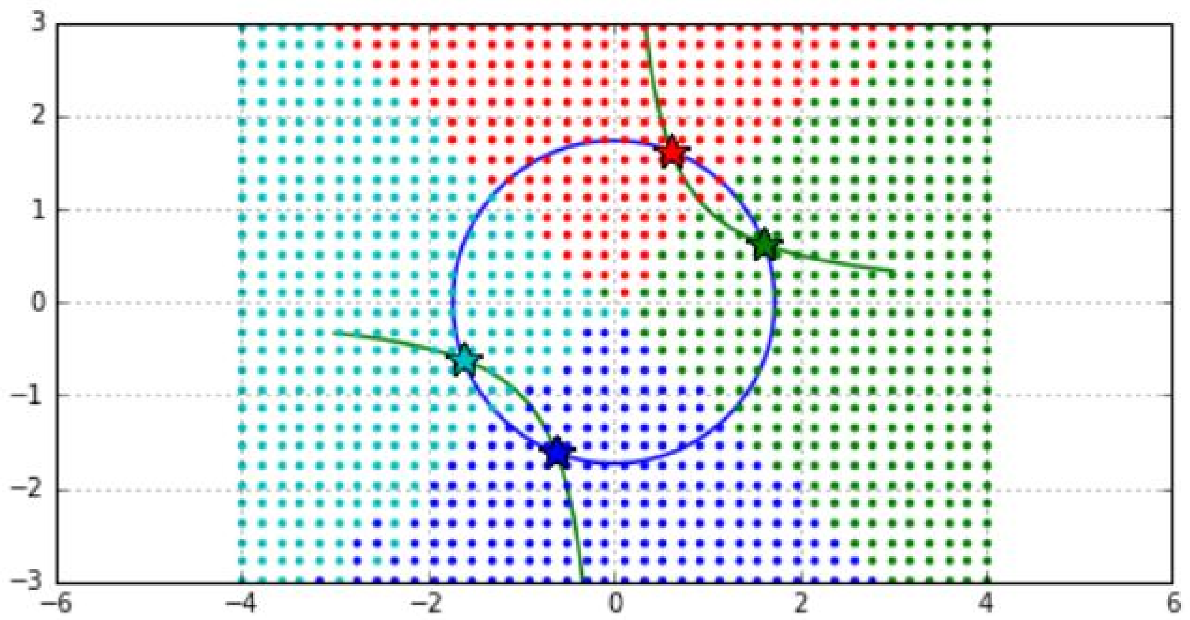
```
# Find all 4 solutions
sol1 = optimize.fsolve(f, [0, 2])
sol2 = optimize.fsolve(f, [2, 0])
sol3 = optimize.fsolve(f, [0, -2])
sol4 = optimize.fsolve(f, [-2, 0])
print("solution 1:", sol1)
print("solution 2:", sol2)
print("solution 3:", sol3)
print("solution 4:", sol4)

colors = ['r', 'g', 'b', 'c']
for m in linspace(-4, 4, 40):
    for n in linspace(-4, 4, 40):
        x_guess = [m, n]
        sol = optimize.fsolve(f, x_guess)

        for idx, s in enumerate([sol1, sol2, sol3, sol4]):
            if abs(s-sol).max() < 1e-8:
                ax.plot(sol[0], sol[1], colors[idx]+'*', markersize = 15)
                ax.plot(x_guess[0], x_guess[1], colors[idx]+'.')
```

Convergence graphics

```
solution 1: [ 0.61803399  1.61803399]
solution 2: [ 1.61803399  0.61803399]
solution 3: [-0.61803399 -1.61803399]
solution 4: [-1.61803399 -0.61803399]
```



References

Chapra & Canale. (2010). Numerical Methods for Engineers, 6th edition. Part two: Roots of equations.

Kiusalaas. (2013). Numerical Methods in Engineering with Python 3. Third Edition. Ch 4. Roots of Equations.

Johansson. (2015). Numerical Python: A Practical Techniques Approach for Industry. Ch. 5. Equation Solving