scipy-equations-solving

March 17, 2024

[ ]:
```
'''There are two approaches of solving  equations in python:
Symbolic Approach (sympy) which gives analytical solutions and
Numerical Approach (Numpy + Scipy) which gives numerical solutions.
The advantage of using SymPy is of course that we may obtain exact results and
we can also include symbolic variables in the matrices. However, not all␣
↪problems
are solvable symbolically, or it may give exceedingly lengthy results. The␣
↪advantage
of using a numerical approach with NumPy/SciPy, on the other hand, is that we␣
↪are
guaranteed to obtain a result, although it will be an approximate solution due␣
↪to
floating-point errors.'''
```

[ ]:
```
'''Note that univariate linear eq. can be simply solved symbolically by sympy.
↪solve method.
However, multivariate system have various mehtods depending on nature of the␣
↪problem.'''
#Solving Sytem of Linear Eqs.
```

[2]:
```
#Symbolic solution of Square system of Linear eqs:
from sympy import *
A=Matrix([[2,6,2],[-3,0,5],[5,4,-7]])
b = Matrix([4, 3,7])
x=A.solve(b)
x
```

[2]:
$$\begin{bmatrix} -7 \\ \frac{21}{5} \\ -\frac{18}{5} \end{bmatrix}$$

[37]:
```
p = symbols("p", positive=True)
A = Matrix([[1, sympy.sqrt(p)], [1, 1/sympy.sqrt(p)]])
b = Matrix([1, 2])
x = A.solve(b)
x
```

[37]:

$$\begin{bmatrix}\frac{2p-1}{p-1}\\ \frac{1}{-\sqrt{p}+\frac{1}{\sqrt{p}}}\end{bmatrix}$$

[38]:
```
#Numerical solution of Square system of Linear eqs:
from scipy import linalg as la
from numpy import *
A=array([[2,6,2],[-3,0,5],[5,4,-7]])
b = array([4, 3,7])
x=la.solve(A,b)
print(x)
```

```
[-7.   4.2 -3.6]
```

[58]:
```
def A(p):
    return array([[1, sqrt(p)], [1, 1/sqrt(p)]])
b = array([1, 2])
def x(p):
    return la.solve(A(p),b)
print(x(2))
```

```
[ 3.         -1.41421356]
```

[6]:
```
#Symbolic solution of Rectangular System (underdetermined) of Linear Eqs.
'''Rectangular systems, with m   n, can be either underdetermined or␣
 ↪overdetermined.
Underdetermined systems have more variables than equations, so the solution␣
 ↪cannot
be fully determined. Therefore, for such a system, the solution must be given␣
 ↪in terms
of the remaining free variables. This makes it difficult to treat this type of␣
 ↪problem
numerically, but a symbolic approach can often be used instead i.e'''
x_vars = symbols("x1, x2, x3")
A = Matrix([[1, 2, 3], [4, 5, 6]])
x = Matrix(x_vars)
b = Matrix([7, 8])
solve(A*x - b, x_vars)
```

[6]: $$\left\{x_1 : x_3 - \frac{19}{3},\ x_2 : \frac{20}{3} - 2x_3\right\}$$

[14]:
```
#Numerical solution of Rectangular System (overdetemined) of Linear Eqs.
'''On the other hand, if the system is overdetermined and has more equations␣
 ↪than
unknown variables, m > n, then we have more constraints than degrees of␣
 ↪freedom, and
```

```
in general there is no exact solution to such a system. However, it is often␣
 ↪interesting to
find an approximate solution to an overdetermined system by least square method␣
 ↪in Mathematics.
Note learn least square method for betterr understanding of underlying␣
 ↪concepts'''
from scipy import linalg as la
# Define the coefficients matrix A and the constants vector b
A = np.array([[1, 2], [3, 4], [5, 6]])
b = np.array([5, 6, 7])
# Solve the least squares problem to find the solution x
x, residuals, rank, singular_values =la.lstsq(A, b)
# Print the solution x and residuals
print("Solution x:", x)
print("Residuals:", residuals)
print('Rank:',rank)
'''The rank of a matrix is the maximum number of linearly independent rows or␣
 ↪columns in the matrix.
It gives insight into the dimensionality and properties of the linear system.'''
print('Singular_values:',singular_values)
```

```
Solution x: [-4.   4.5]
Residuals: 8.735582265323586e-31
Rank: 2
Singular_values: [9.52551809 0.51430058]
```

[ ]:
```
#Finding EigenValues and EigenVectors of a Square Matrix:
```

[21]:
```
#By Sympy:
from sympy import *
init_printing()
'''In SymPy, we can use the eigenvals and eigenvects methods of the Matrix␣
 ↪class, which are able to compute
the eigenvalues and eigenvectors of some matrices with elements that are␣
 ↪symbolic expressions.'''
eps, delta = symbols("epsilon, Delta")
H = Matrix([[eps, delta], [delta, -eps]])
H
```

[21]:
$$\begin{bmatrix} \epsilon & \Delta \\ \Delta & -\epsilon \end{bmatrix}$$

[23]:
```
H.eigenvals()
```

[23]:
$$\left\{-\sqrt{\Delta^2+\epsilon^2} : 1,\ \sqrt{\Delta^2+\epsilon^2} : 1\right\}$$

[24]:
```
H.eigenvects()
```

[24]:

$$\left[\left(-\sqrt{\Delta^2 + \epsilon^2},\ 1,\ \left[\begin{bmatrix} \frac{\epsilon}{\Delta} - \frac{\sqrt{\Delta^2+\epsilon^2}}{\Delta} \\ 1 \end{bmatrix} \right] \right),\ \left(\sqrt{\Delta^2 + \epsilon^2},\ 1,\ \left[\begin{bmatrix} \frac{\epsilon}{\Delta} + \frac{\sqrt{\Delta^2+\epsilon^2}}{\Delta} \\ 1 \end{bmatrix} \right] \right) \right]$$

[38]:

```
'''Obtaining analytical expressions for eigenvalues and eigenvectors using␣
↪these methods is often very desirable indeed,
but unfortunately it only works for small matrices. For anything larger than a␣
↪3 × 3, the analytical expression
typically becomes extremely lengthy and cumbersome to work with even using a␣
↪computer algebra system such as SymPy.
Therefore, for larger systems we must resort to a fully numerical approach'''
#By Scipy/Numpy:
from numpy import *
from scipy import linalg as la
A = array([[1, 3, 5], [3, 5, 3], [5, 3, 9]])
evals, evecs = la.eig(A)
print(evals,'\n',evals.real)
```

```
[13.35310908+0.j -1.75902942+0.j  3.40592034+0.j]
 [13.35310908 -1.75902942  3.40592034]
```

[27]:

```
evecs
```

[27]:

```
array([[ 0.42663918,  0.90353276, -0.04009445],
       [ 0.43751227, -0.24498225, -0.8651975 ],
       [ 0.79155671, -0.35158534,  0.49982569]])
```

[ ]:

```
#Solving Sytem of Non-Linear Eqs.
```

[39]:

```
#Univariate systems:
'''Some univariate non-linear eqs. can be solved analyticaly by sympy, however␣
↪most require numerical solutions
such as newton method or bisection method'''
x, a, b, c = symbols("x, a, b, c")
solve(a + b*x + c*x**2, x)
```

[39]:

$$\left[\frac{-b - \sqrt{-4ac + b^2}}{2c},\ \frac{-b + \sqrt{-4ac + b^2}}{2c} \right]$$

[48]:

```
'''The SciPy optimize module provides multiple functions for numerical root␣
↪finding.
The optimize.bisect and optimize.newton functions implement variants of␣
↪bisection
and Newton methods. The optimize.bisect takes three arguments: first a Python
function (e.g., a lambda function) that represents the mathematical function␣
↪for the
```

```
equation for which a root is to be calculated and the second and third␣
↪arguments are the
lower and upper values of the interval for which to perform the bisection␣
↪method. Note
that the sign of the function has to be different at the points a and b for the␣
↪bisection
method to work, as discussed earlier. '''
from scipy import optimize
optimize.bisect(lambda x:x-cos(x),-2,2)
```

[48]: 0.739085133214758

[52]:
```
'''In contrast, the function optimize.newton for Newton's method
takes a function as the first argument and an initial guess for the root of the␣
↪function as
the second argument. Optionally, it also takes an argument for specifying the␣
↪derivative
of the function, using the fprime keyword argument. If fprime is given, Newton's
method is used; otherwise the secant method is used instead.'''
x_root_guess = 2
f = lambda x:exp(x)-2
fprime = lambda x:exp(x)
optimize.newton(f, x_root_guess) #Secant Method
```

[52]: 0.693147180559946

[53]:
```
optimize.newton(f, x_root_guess,fprime=fprime) #Newton Method
```

[53]: 0.693147180559945

[62]:
```
#Multivariate system:
'''No symbolic solution possible, only numerical solution. fsolve is a␣
↪powerful function for solving systems of
nonlinear equations numerically. It requires an initial guess for the solution␣
↪and the definition of the system
of equations as a function. The function iteratively refines the solution until␣
↪it converges to a root or reaches
the specified tolerance level.'''
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve

# Define the system of nonlinear equations as a function
def equations(x):
    # Define the equations
    eq1 = x[0]**2 + x[1]**2 - 4  # x^2 + y^2 = 4
    eq2 = x[0]*x[1] - 1  # xy = 1
```

```python
    # Return the equations as a tuple
    return (eq1, eq2)

# Initial guess for the solution
initial_guess = [1, 1]

# Solve the system of equations
solution = fsolve(equations, initial_guess)

# Print the solution
print("Solution:", solution)
```

```
Solution: [1.93185165 0.51763809]
```