FN6815_Lect_05_SolveN

March 1, 2024

FN6815 Numerical Methods for Financial Instrument Pricing

Lecture 5: Solve Multiple Equations

- Dr. Yang Ye
- Email: yy@runchee.com

- 2023/2024 Mini Term 5

1. Recap

In the previous lecture, we discussed several methods for solving non-linear equation:

1. **Newton-Raphson method**: This method is used to find roots of $f(x) = 0$ when $f'(x)$ is known.
2. **Secant method**: This method is also used to find roots of $f(x) = 0$, but it doesn't require knowledge of $f'(x)$.
3. **Bisection method**: This method is used to find roots of $f(x) - y = 0$ within a known interval $[a, b]$ where $f(a)f(b) < 0$.

Discussion:

When considering the generalization of these methods to solve a system of equations rather than a single equation, which method is viable?

Bisection Method: In n-dimensions, for the bisection method to work, we need to find a range such that $f(\vec{x_a}) \cdot f(\vec{x_b}) < 0$. This can be challenging in higher dimensions.

Newton's Method: Newton's method, on the other hand, doesn't require a range but only a starting point. It can be generalized to n-dimensions using Jacobian matrix, making it more suitable for solving systems of equations.

2. Solve for Multiple Equations

2.1 Solving Linear Equations

Gaussian Elimination is a prevalent and robust method for solving systems of linear equations. It systematically reduces the system to a form from which the solution can be directly read.

Python's `numpy` package provides a module `linalg` that includes a function `solve()`. It solves a system of linear equations in the form $Ax = b$. with `x = numpy.linalg.solve(A, b)`.

Question: Which equation below is linear?

1. $3x_1 + 4x_2 - 3 = -5x_3$
2. $\frac{-x_1+x_2}{x_3} = 2$
3. $x_1 x_2 + x_3 = 5$

Let's Solve for below linear equations $\begin{cases} 3x_1 + 4x_2 - 3 = 0 \\ x_1 + x_2 = 5 \end{cases}$

[1]:
```
import numpy as np

A = np.array([[3, 4], [1, 1]])
b = np.array([3, 5])
x = np.linalg.solve(A, b)
print(x)

assert np.allclose(np.dot(A, x), b)
```

```
[ 17. -12.]
```

Alternatively, use inv method, $x = A^{-1}b$.

[2]:
```
import numpy as np

A_inv = np.linalg.inv(A)
x = np.dot(A_inv, b)
print(x)

assert np.allclose(np.dot(A, x), b)
```

```
[ 17. -12.]
```

2.2 Solve Non-linear Equations

2.2.1 Taylor expansions for multi-variable functions The concept behind Newton's method for solving a single equation can be extended to multiple variables. The idea is to iteratively approximate the nonlinear function f at a point $\mathbf{x}_i$ by a linear function and find its root $\mathbf{x}_{i+1}$. This process is repeated until the root approximation improves insignificantly.

In the case of n variables, we need to approximate a vector function $\mathbf{F}(\vec{\mathbf{x}})$ by some linear function $\tilde{\mathbf{F}} = \mathbf{J}\vec{\mathbf{x}} + \mathbf{c}$. Here, $\mathbf{J}$ is an $n \times n$ matrix (the Jacobian matrix of $\mathbf{F}$), and $\mathbf{c}$ is a vector of length n.

How to obtain the linear form of $\mathbf{F}$ at $\vec{\mathbf{x}}_i$? The Taylor series expansion. We use the first two terms of the expansion to form the linear approximation:

$$\mathbf{F}(\vec{\mathbf{x}}) \approx \mathbf{F}(\vec{\mathbf{x}_i}) + \mathbf{J}(\vec{\mathbf{x}_i})(\vec{\mathbf{x}} - \vec{\mathbf{x}_i})$$

This equation represents the linear approximation of $\mathbf{F}$ around the point $\mathbf{x}_i$.

The next terms in the expansions are omitted here and of ths scale $||\vec{\mathbf{x}} - \vec{\mathbf{x}_i}||^2$, which are assumed to be small and negligible compared with the two terms above.

2.2.2 Jacobian ∇ or J *Jacobian* **J** of vector function **F** is the matrix of all the partial derivatives of **F**. Component (i, j) in $\nabla\mathbf{F}$ is

$$\frac{\partial F_i}{\partial x_j}.$$

For 2-d vector function **F**, We can then write

$$\mathbf{J} = \begin{pmatrix} \frac{\partial F_0}{\partial x_0} & \frac{\partial F_0}{\partial x_1} \\ \frac{\partial F_1}{\partial x_0} & \frac{\partial F_1}{\partial x_1} \end{pmatrix}$$

2.2.3 Newton's method for n-dimension The Newton-Raphson method for one dimension is given by:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, 2, ...$$

We can rewrite this as:

$$f(x_n) + f'(x_n)(x_{n+1} - x_n) = 0$$

For multiple dimensions, we use the Jacobian matrix to obtain the linear equation at $\mathbf{x}_i$,

$$\mathbf{F}(\vec{\mathbf{x}}) \approx \mathbf{F}(\vec{\mathbf{x}_i}) + \mathbf{J}(\vec{\mathbf{x}_i})(\vec{\mathbf{x}} - \vec{\mathbf{x}_i})$$

And then solve for $\mathbf{x}_{i+1}$, by setting $\mathbf{F}(\mathbf{x}_{i+1}) = 0$.

$$\mathbf{F}(\vec{\mathbf{x}_i}) + \mathbf{J}(\vec{x_i})(\vec{x_{i+1}} - \vec{x_i}) = 0,$$

We use symbol for the the change in $\mathbf{x}$ to the next iteration, $\mathbf{x}_{i+1} = \mathbf{x}_i +$.

$$\mathbf{J}(\mathbf{x}_i) \ = -\mathbf{F}(\mathbf{x}_i),$$

We then go through the iterative process of following steps:

1. Solve the linear system $\mathbf{J}(\mathbf{x}_i) \ = -\mathbf{F}(\mathbf{x}_i)$ to find .
2. Update the estimate: $\mathbf{x}_{i+1} = \mathbf{x}_i +$.
3. Check the exit condition: if the norm of $\mathbf{F}(\mathbf{x}_i)$ is sufficiently small, exit the loop. Because $\mathbf{F}(\mathbf{x}_i)$ is a vector, we use `linalg.norm` to check all its values are close to zero.

Here's a simple Python example using the Newton's method for a system of equations:

[3]:
```
from typing import Callable

import numpy as np
import numpy.typing as npt
```

```
print(npt.NDArray)


def Newton_system(F: Callable, J: Callable, x: npt.NDArray, eps: float):
    """
    Solve nonlinear system F=0 by Newton's method.

    J is the Jacobian of F.
    Both F and J must be functions of x.
    x holds the start value.
    The iteration continues until ||F|| < eps.
    """
    F_value = F(x)
    F_norm = np.linalg.norm(F_value, ord=2)  # l2 norm of vector
    iteration_counter = 0
    while abs(F_norm) > eps and iteration_counter < 100:
        # Inverse is more expensive calculation than solving
        # Use inverse for simple verification only
        # Inverse: delta = np.matmul(np.linalg.inv(J(x)), -F_value)
        if F_value.shape == (1,):
            delta = 1 / J(x) * -F_value
        else:
            delta = np.linalg.solve(J(x), -F_value)
        x = x + delta
        F_value = F(x)
        F_norm = np.linalg.norm(F_value, ord=2)
        iteration_counter += 1

    # Here, either a solution is found, or too many iterations
    if abs(F_norm) > eps:
        return None, iteration_counter
    return x, iteration_counter


def test_Newton_system1(F, J, expected, start_x, tol=1e-4):
    x, n = Newton_system(F, J, x=start_x, eps=tol)
    print(f"x={x}, n={n}")
    error_norm = np.linalg.norm(expected - x, ord=2)
    assert error_norm < tol, "norm of error =%g" % error_norm
    print("norm of error =%g" % error_norm)
```

```
numpy.ndarray[typing.Any, numpy.dtype[+_ScalarType_co]]
```

2.3 Try it Out

2.3.1 1-D linear system

$$y = 3x + 3$$
$$\frac{\mathrm{d}y}{\mathrm{d}x} = 3$$

[4]:
```
# We can test the function Newton_system with the 2×2 system (172)-(173):
F = lambda x: np.array(
    [
        3 * x + 3,
    ]
)
J = lambda _x: np.array([3])

test_Newton_system1(F, J, expected=np.array(-1), start_x=np.array(100))
# Linear function's n == 1
```

```
x=[-1.], n=1
norm of error =0
```

2.3.2 2-D linear system

[5]:
```
A = np.array([[3, 4], [1, 1]])
b = np.array([3, 5])
x = np.linalg.solve(A, b)
print(x)
```

```
[ 17. -12.]
```

[6]:
```
# Use Newton's method
F = lambda x: np.array([3 * x[0] + 4 * x[1] - 3, 1 * x[0] + 1 * x[1] - 5])
J = lambda _x: np.array([[3, 4], [1, 1]])

test_Newton_system1(F, J, expected=np.array([17, -12]), start_x=np.array([1,␣
↪2]))
# For linear function's n == 1
```

```
x=[ 17. -12.], n=1
norm of error =3.97205e-15
```

2.3.3 Non-linear Equations

Let's solve for below non-linear equations

$$\begin{cases} y = 3x + sin(x)^2 \\ y^2 = cos(x) \end{cases}$$

[7]:
```
def F(x):
    x0, x1 = x[0], x[1]
    return np.array([x1 - 3 * x0 - np.sin(x0) ** 2, x1**2 - np.cos(x0)])
```

```
def J(x):
    x0, x1 = x[0], x[1]
    return np.array([[-2 * np.sin(x0) * np.cos(x0) - 3, 1], [np.sin(x0), 2 *␣
 ↪x1]])


expected = np.array(
    [-0.36456854, -0.96658041],
)
test_Newton_system1(F, J, expected, start_x=np.array([2.0, -1.0]), tol=1e-8)
F([-0.36456854, -0.96658041])
```

```
x=[-0.36456854 -0.96658041], n=6
norm of error =2.83469e-09
```

[7]: `array([-1.05223608e-11,  5.44045708e-09])`

2.4 Summary

To solve for n-dimensional non-linear equation, Newton's method has the advantage of speed and convergence. The method requires the Jacobian matrix of the function, and iteratively updates the estimate of the root until the function value is close to zero.

The Newton's method for solving systems of equations in vector form is given by:

$$\vec{x}_{n+1} = \vec{x}_n - [J(\vec{x}_n)]^{-1} \cdot \vec{f}(\vec{x}_n)$$

In this equation, $\vec{x}_n$ is the current estimate of the solution, $J(\vec{x}_n)$ is the Jacobian matrix evaluated at $\vec{x}_n$, and $\vec{f}(\vec{x}_n)$ is the vector function evaluated at $\vec{x}_n$. The term $[J(\vec{x}_n)]^{-1} \cdot \vec{f}(\vec{x}_n)$ represents the step taken in the direction of the solution.

3. Assignment

Q1. Gradient Descent

Gradient Descent, a first-order iterative optimization algorithm, is used to find a local minimum of a differentiable function. It's built on Newton's method and was first suggested by Augustin-Louis Cauchy in 1847. Jacques Hadamard independently proposed a similar method in 1907.

The algorithm uses the Jacobian matrix at position $\bar{x}_n$ to iteratively update $\bar{x}_{n+1}$, aiming to reach a local or global minimum. The learning rate, denoted as η, influences the step size in each iteration:

$$\bar{x}n + 1 = \bar{x}n - J * \eta$$

The learning rate must be carefully chosen. If it's too large, the algorithm may oscillate around the minimum. If it's too small, the convergence may be slow.

Consider the unimodal function $z^2 = (x - x_c)^2 + (y - y_c)^2$ with $x_c = 2$ and $y_c = -1$. We want to find the minimum of this function using gradient descent.

Here's a prototype for the Python function solve() that implements the gradient descent algorithm:

```
def solve(xs_init, _f_func, _j_func, ...)
```

In this function, `xs_init` is the initial guess, `_f_func` is the function to minimize, `_j_func` is the Jacobian of `_f_func`, `learning_rate` is the learning rate, and `precision` is the desired precision of the result be reached.

You can experiment with different learning rates and initial guesses to observe the behavior of the algorithm. while

Find two set of parameters such that

1. Not able to find the answer with whatever number of steps.
2. A suitable learning rate and initial guess can help the algorithm find the minimum $(2, -1)$ within 0.01 precision.

Appendix: timestamp

[8]:
```
from datetime import datetime

print(f"Generated on {datetime.now()}")
```

```
Generated on 2024-03-01 22:12:24.262064
```