國立政治大學應用數學系

碩士學位論文

SVM 在解非線性方程式的應用

The Application of SVM in Solving Nonlinear Equations

指導教授：曾正男　博士
吳柏林　博士

研究生：林雨鵬　撰

中 華 民 國 109 年 12 月

DOI:10.6814/NCCU202001843

致謝

衷心地感謝指導教授曾正男老師及吳柏林老師，在論文寫作期間，正男老師對於論文的方向制定及內容修改皆給予細心的指導，柏林老師提供寶貴的建議及暖心的支持。同時他們的教學精神及生活態度也令我敬佩，正男老師授課認真嚴謹，課餘對偏鄉教育及數位學習的推動不遺餘力，柏林老師著作等身，有著幽默及平易近人的人格魅力，這些都成為我學習的榜樣及目標。此外，還要感謝晉維、以洵等同學在 LaTeX、Python 方面的熱情協助，有了老師及同學們的指導和幫助，這篇論文才能順利完成。

工作多年後又再次回到校園，如今這兩年多的驚奇旅程即將告一段落，覺得自己好像學到了一點，卻又發現其實自己不懂的更多，感謝可愛的政大師生，也感謝親愛家人的支持，讓我擁有再當一次學生的幸福時光。

DOI:10.6814/NCCU202001843

中文摘要

解非線性方程式雖然有許多數學標準方法，但是在高維度的求解以及有無窮多解的問題上，現有的方法可以計算出來的結果仍然非常有限，我們希望可以提出一個簡單快速的方法，可以了解無窮多解的分布狀況，並且在局部區域也能找出精確解，同時希望對這些解有可視化的了解。我們利用 SVM 的特性開發了一個新的方法，可以同時達到以上目標。

DOI:10.6814/NCCU202001843

Abstract

There are many standard mathematical methods for solving nonlinear equations. But when it comes to equations in high dimension with infinite solutions, the results from current methods are quite limited. We present a simple fast way which could tell the distribution of these infinite solutions and is capable of finding accurate approximations. In the same time, we also want to have a visual understanding about the roots. Using the features of SVM, we have developed a new method that achieves the above goals.

Keywords: nonlinear equations, SVM

DOI:10.6814/NCCU202001843

Contents

DOI:10.6814/NCCU202001843

DOI:10.6814/NCCU202001843

List of Tables

DOI:10.6814/NCCU202001843

List of Figures

DOI:10.6814/NCCU202001843

Chapter 1

Introduction

In physics, chemistry and engineering, many problems appear in the form of nonlinear equations. But even for relatively simple example like polynomial equations, there is no formula for radical solutions to degree 5 or higher. Not to mention those equations of transcendental functions like $f(x) = e^x - 2x - 1$ or $f(x) = \sin^{-1} x + x^2 - 1$. However, using numerical methods, the solutions can be computed to a desired degree of accuracy. Therefore in this chapter, we will introduce some numerical methods for solving nonlinear equations $f(x) = 0$ where $f : \mathbb{R} \to \mathbb{R}$ or $f : \mathbb{R}^n \to \mathbb{R}$.

1.1 Nonlinear equations of one variable

For equations with one variable, we already have many tools to obtain the approximation of roots. For example, two-point bracketing method like Bisection method [1] and false position method(or Regula falsi) [2]. Also we have Fixed-point iteration [1] and its improved version Wegstein' s method [3] [5]. Besides, we cannot fail to mention the most famous Newton's method which can be seen as a special case of fixed-point iteration. Newton's method has many extensions and variations. For example, secant method [6] and Steffensen's method [7] [8] both replace the derivative in Newton's method by the slope of secant line but with different step size. And Halley's method [9] [10] uses second order of Taylor series instead of first order linear approximation in Newton's method. Now, to begin this section, suppose that $f : [a, b] \to \mathbb{R}$ is continuous on $[a, b] \subseteq \mathbb{R}$ and that x^* is the only root of $f(x) = 0$ in $[a, b]$. Then we may try these following methods to obtain a numerical approximation of the root.

DOI:10.6814/NCCU202001843

1.1.1 Bisection method

In bisection method, first we need an interval which contains the root x^*. Suppose that $[a_0, b_0]$ is such an interval. That is, $a_0 < b_0$ in $[a, b]$ and $f(a_0)f(b_0) < 0$. Let $x_0 = \frac{a_0+b_0}{2}$ be the initial approximation of x^*. Then we decide the next interval and approximation by the following procedures:

1. If $f(a_0)f(x_0) < 0$, then set $a_1 = a_0$, $b_1 = x_0$ and $x_1 = \frac{a_1+b_1}{2}$.
2. If $f(x_0)f(b_0) < 0$, then set $a_1 = x_0$, $b_1 = b_0$ and $x_1 = \frac{a_1+b_1}{2}$.
3. If none of the above happens, that means $f(x_0) = 0$ and we have found the root.

Continue the above iteration and we will get $\{x_n\}$, a sequence of approximations of the root x^*. When the absolute value of $f(x_n)$ is sufficiently small, return x_n and stop iterating. See Figure 1.1.

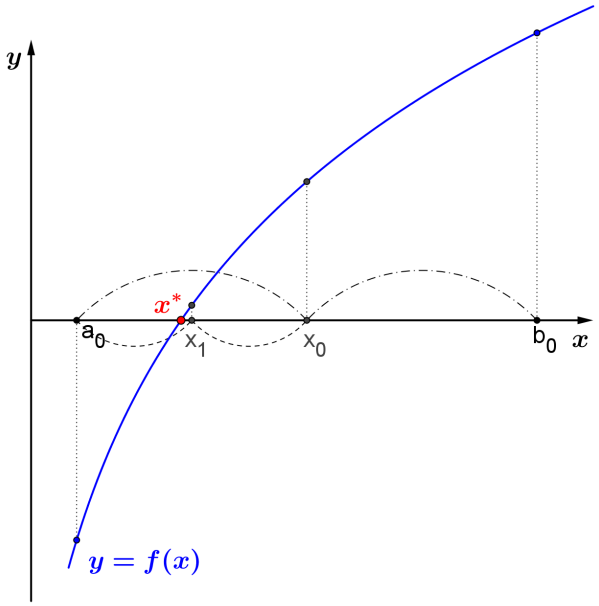


Figure 1.1: Bisection method.

And from [1, p. 5], bisection method has the following pros and cons:

- It is simple and straightforward.
- It can be applied to non-analytic functions.

DOI:10.6814/NCCU202001843

- It needs first an interval that contains the root.
- It cannot be applied to double roots.
- It cannot be applied to multiple equations.
- The absolute error is halved at each step so the method converges linearly, which is comparatively slow.

1.1.2 False position method (or Regula falsi)

In bisection method, the successive approximation x_n is the midpoint of a_n and b_n. Now in false position method, the middle point is replaced by the intersection of x-axis and the secant line constructed by $(a_n, f(a_n))$ and $(b_n, f(b_n))$. See Figure 1.2.

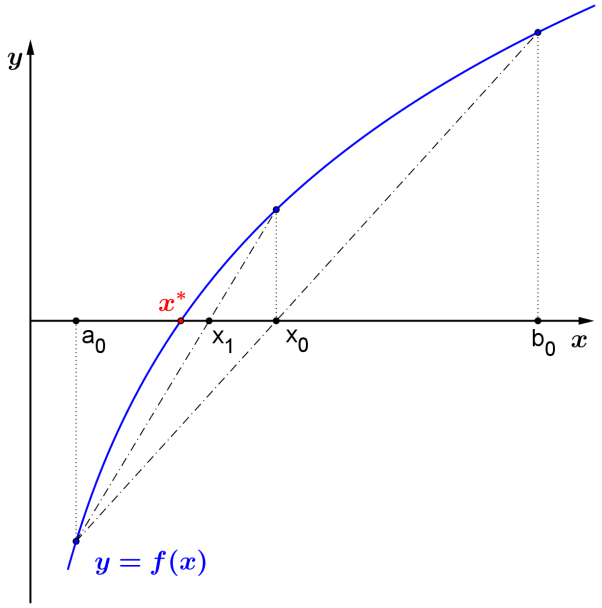


Figure 1.2: False position method.

So the procure is modified as follows:

1. Find $a_0 < b_0$ in $[a, b]$ such that $f(a_0)f(b_0) < 0$.
2. Replace y by 0 in the equation

$$\frac{y - f(a_0)}{x - a_0} = \frac{f(b_0) - f(a_0)}{b_0 - a_0}$$

DOI:10.6814/NCCU202001843

and we have

$$x = \frac{a_0 f(b_0) - b_0 f(a_0)}{f(b_0) - f(a_0)} = x_1$$

3. Hence the iterative formula is

$$x_{n+1} = \frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}, n \geq 0$$

And by [2], the rate of convergence of false position method is about $\frac{1+\sqrt{5}}{2} \approx 1.6$, higher than bisection method, but it could be inefficient in some extreme cases. See Figure 1.3.

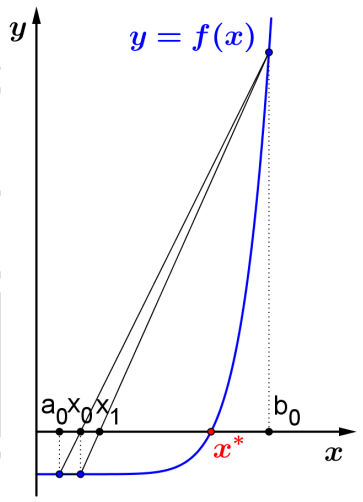


Figure 1.3: An extreme case of false position method.

1.1.3 Fixed-point iteration

This method comes from the fixed-point theorem [1, p. 6]:

> *Suppose that* $g : [a, b] \to \mathbb{R}$ *is continuous on* $[a, b] \subseteq \mathbb{R}$. *If* $g([a, b]) \subseteq [a, b]$, *then* g *has at least one fixed point in* $[a, b]$.

Moreover, in https://en.wikipedia.org/wiki/Banach_fixed-point_theorem we have:

> *If* g *is differentiable on* (a, b) *and* $|g'(x)| \leq K$ *for some positive constant* $K < 1$, *then* g *has exactly one fixed point in* $[a, b]$.

So first, we write $f(x) = 0$ as the form $x = g(x)$. For example:

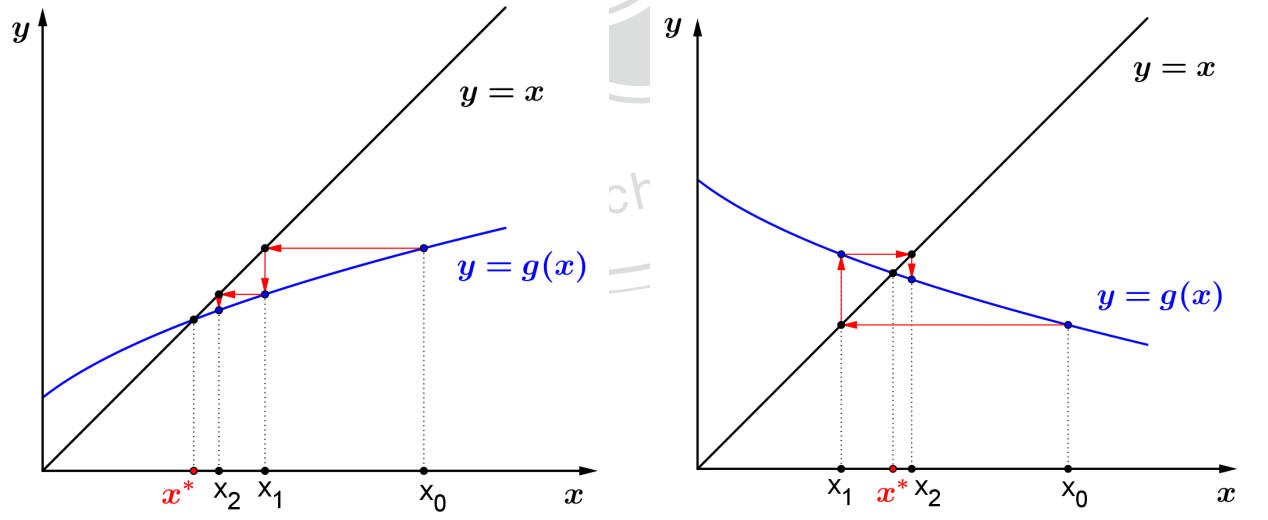
DOI:10.6814/NCCU202001843

1. $f(x) = e^x - 2x - 2$ and from $f(x) = 0$ we can obtain $g(x)$ in several ways:
   - $x = \frac{1}{2}(e^x - 2) = g(x)$
   - $e^x = 2x + 2 \Rightarrow x = \ln(2x + 2) = g(x)$
2. $f(x) = x^3 - 2x - 5 = 0$
   - $x = \frac{1}{2}(x^3 - 5) = g(x)$
   - $x = \sqrt[3]{2x + 5} = g(x)$
   - $x^3 = 2x + 2 \Rightarrow x = \frac{2x+2}{x^2} = g(x)$

Or we could always obtain g by setting $x = x + \lambda f(x) = g(x)$ where $\lambda \neq 0$. Geometrically, it is to find the intersection of $y = x$ and $y = g(x)$. Next, we try to find the root x^* through the iterative process

$$x_n = g(x_{n-1}), n \geq 1 \qquad \text{where } x_0 \text{ is arbitrary in } [a, b]$$

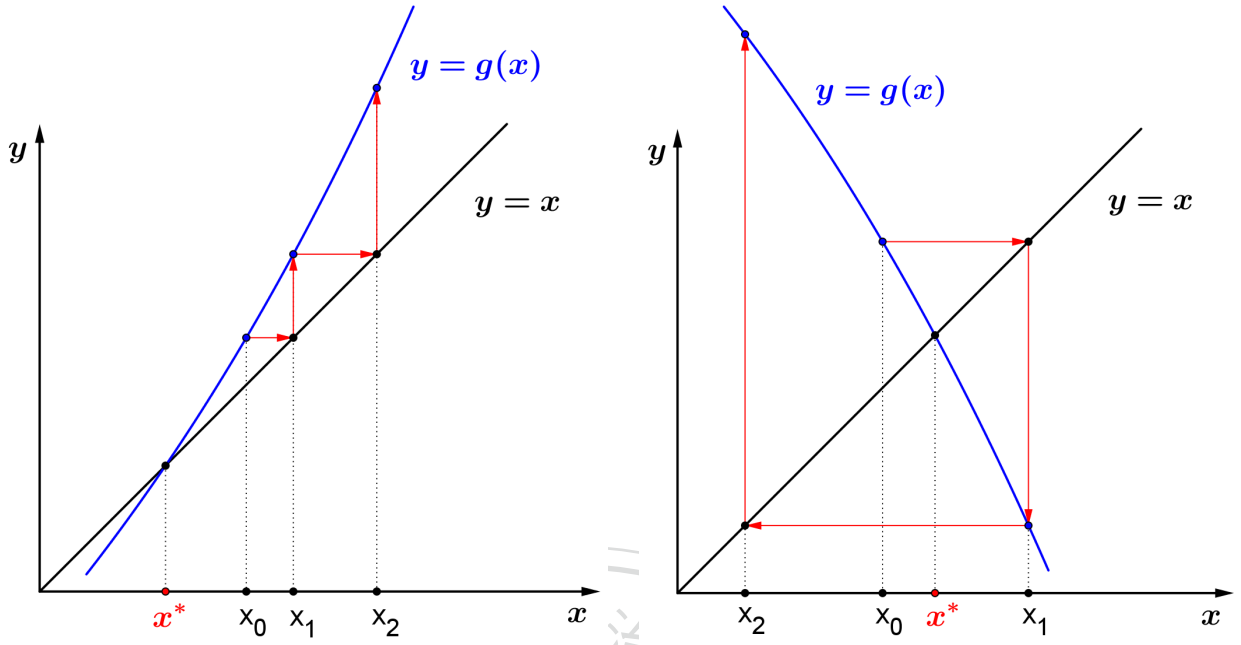
see Figure 1.4. Notice that $\{x_n\}$ doesn't always converge, see Figure 1.5. So "the challenge lies in choosing a proper g such that $\{x_n\}$ converges and does it as quickly as possible" [1, p. 7].



(a) zigzag towards the root when $0 < g'(x) < 1$ (b) spiral towards the root when $-1 < g'(x) < 0$

Figure 1.4: Convergent cases of fixed-point iteration.

DOI:10.6814/NCCU202001843



(a) zigzag away from the root when $g'(x) > 1$ (b) spiral away from the root when $g'(x) < -1$

Figure 1.5: Divergent cases of fixed-point iteration.

Furthermore, fixed-point iteration is at least of linear convergence as explained below. Let $e_n = x_n - x^*$. From $x_{i+1} = g(x_i)$ and $x^* = g(x^*)$, we have $e_{i+1} = x_{i+1} - x^* = g(x_i) - g(x^*)$. By mean value theorem, there exists ξ between x_i and x^* such that $g'(\xi) = \frac{g(x_i)-g(x^*)}{x_i-x^*}$, so $e_{i+1} = g'(\xi)(x_i - x^*) = g'(\xi)e_i$, which indicates the linear convergence. By choosing g wisely, the rate of convergence can be improved as we will see in section 1.1.5.

1.1.4 Wegstein's method

"This method is a modification for accelerating the rate of convergence in fixed-point iteration and capable of producing solutions even in those cases where the fixed-point iteration diverges" [3, p. 9]. In fixed-point iteration, the iterative procedure is

$$x_n = g(x_{n-1}), n \geq 1 \qquad \text{where } x_0 \text{ is arbitrary in } [a, b]$$

Now we change the successive values of x_n in the following way [4, p. 22,23]:

1. Choose arbitrary x_0 in $[a, b]$ and let $x_1 = g(x_0)$.

DOI:10.6814/NCCU202001843

2. Find the intersection of the line $y = x$ and the line constructed with the points $(x_0, g(x_0))$ and $(x_1, g(x_1))$. And let x_2 be the x coordinate of the intersection. That is, denote $g(x_0)$ by g_0 and $g(x_1)$ by g_1, and replace y with x in the equation

$$\frac{y - g_0}{x - x_0} = \frac{g_1 - g_0}{x_1 - x_0}$$

which yields

$$x = \frac{x_1 g_0 - g_1 x_0}{(x_1 - x_0) - (g_1 - g_0)} = x_2$$

3. So we have the iterative formula

$$x_{n+1} = \frac{x_n g_{n-1} - g_n x_{n-1}}{(x_n - x_{n-1}) - (g_n - g_{n-1})}, n \geq 1$$

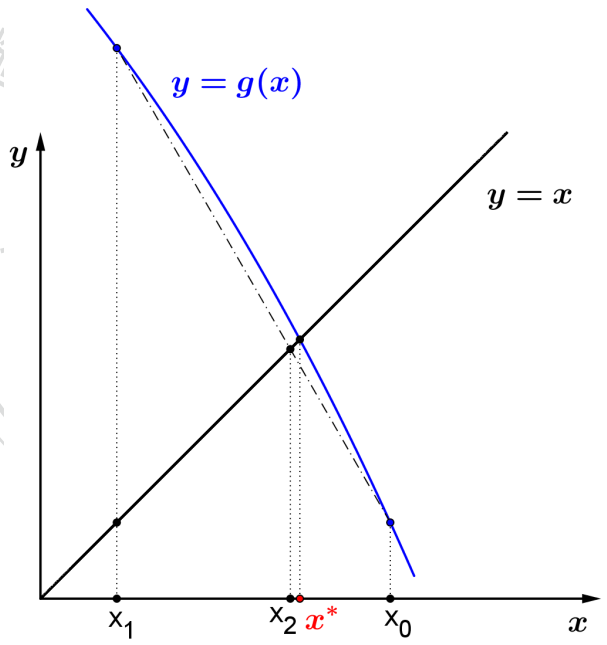


Figure 1.6: Wegstein's method.

Figure 1.6 explains how Wegstein's method finds the next approximation. And by [5, p. 4], the advantages of this method are:

- A derivative is not used.
- The rate of convergence is nearly quadratic. (Greater than $\frac{1+\sqrt{5}}{2}$ [5, p. 9])

DOI:10.6814/NCCU202001843

- The condition that $|g'(x)| < 1$ near the root is not required.
- A less accurate first guess can be tolerated.
- In many cases, it will make otherwise divergent cases convergent.

1.1.5 Newton's method

This is the most discussed method in finding the approximation of a root by an iterative way and it has many variations. In short, Newton's method uses the intersection of tangent line at $(x_n, f(x_n))$ and x-axis to find the successive value. See Figure 1.7. Now suppose that f is differentiable on (a, b). Newton's method works as follows:

1. Choose x_0 in (a, b). The tangent line at $(x_0, f(x_0))$ is $y = f(x_0) + f'(x_0)(x - x_0)$.
2. Find the intersection of the tangent line and the x-axis. And the successive approximation is the x-coordinate of the intersection: $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$.
3. Hence the iterative formula is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, n \geq 1$$

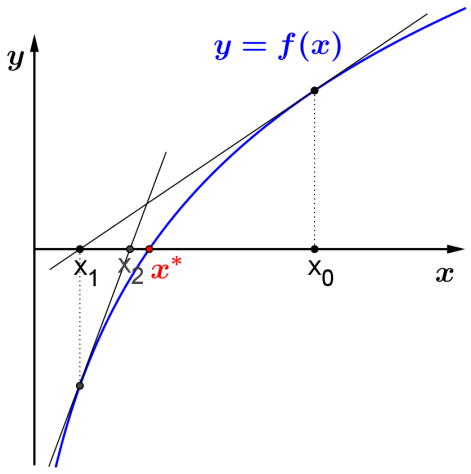
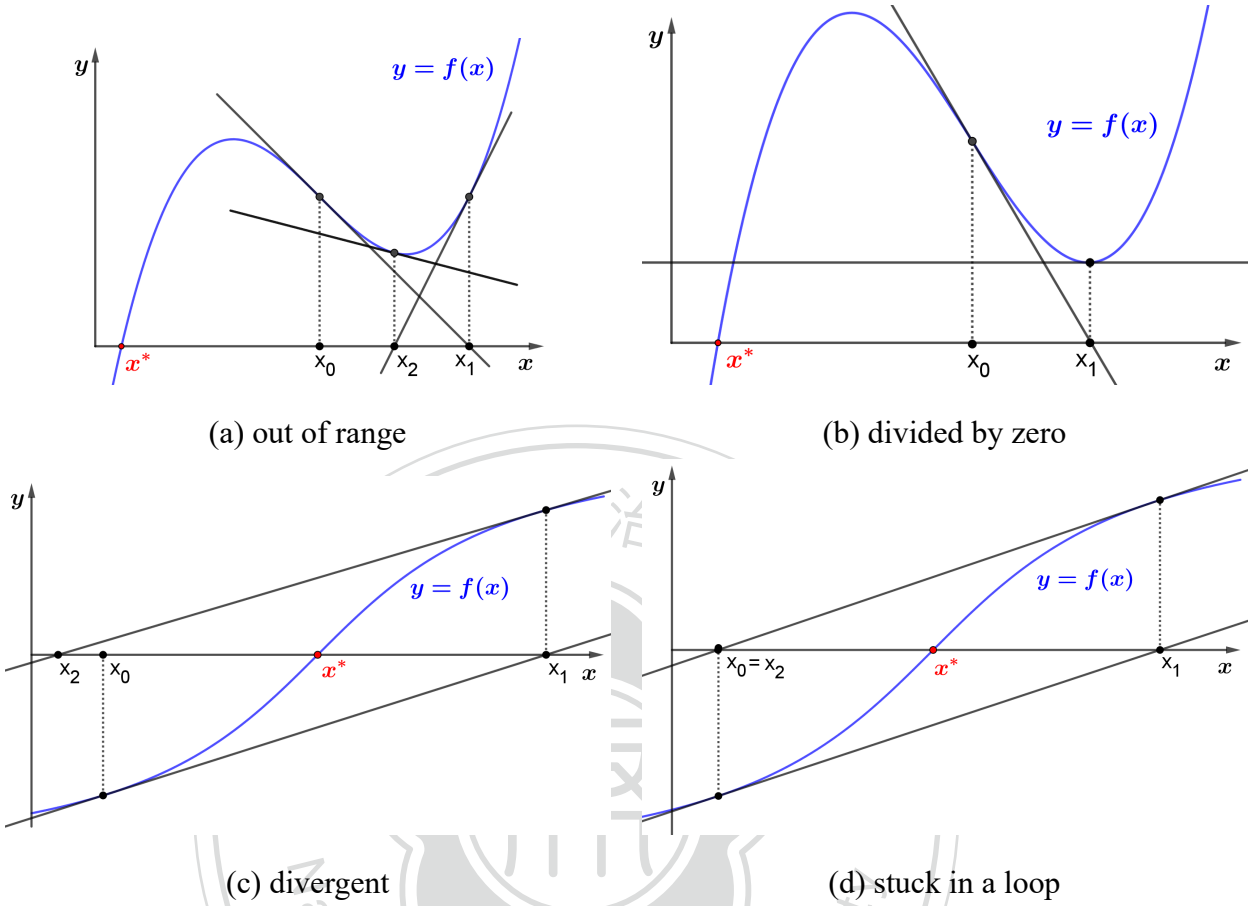


Figure 1.7: Newton's method.

Newton's method also has some weaknesses, one is that we need to compute the derivative of f which could be complicated or take a lot of time. Another is that not all the choice of initial

DOI:10.6814/NCCU202001843

value x_0 will lead to the root x^*. See Figure 1.8. So the challenge is to find a proper x_0 or we simply try another x_0 when it doesn't converge.



(a) out of range (b) divided by zero

(c) divergent (d) stuck in a loop

Figure 1.8: Failed cases of Newton's method.

From the iterative formula of Newton's method, we can see that it is a special case of fixed-point iteration where $g(x) = x - \frac{f(x)}{f'(x)}$. Moreover, it achieves quadratic convergence as explained below:

Since Taylor series of $f(x)$ at x_n is

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{f''(\xi)}{2!}(x - x_n)^2 \qquad \text{for some } \xi \text{ between } x \text{ and } x_n$$

Let $x = x^*$ and we have

$$0 = f(x_n) + f'(x_n)(x^* - x_n) + \frac{f''(\xi)}{2!}(x^* - x_n)^2 \tag{1.1.1}$$

DOI:10.6814/NCCU202001843

On the other hand,

$$0 = f(x_n) + f'(x_n)(x_{n+1} - x_n) \tag{1.1.2}$$

Denote $x^* - x_i$ by e_i for all positive integer i and subtract (1.1.2) from (1.1.1):

$$0 = f'(x_n)e_{n+1} + \frac{f''(\xi)}{2!}(e_n)^2$$

So

$$\frac{|e_{n+1}|}{|e_n|^2} = |\frac{f''(\xi)}{2f'(x_n)}|$$

Which implies quadratic convergence. Notice that if x^* is a multiple root, then we only have linear convergence.

1.1.6 Secant method

The idea of secant method is to replace the tangent line of f at $(x_n, f(x_n))$ in Newton's method with the secant line that passes $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$ when in some cases we can't compute $f'(x)$ explicitly. That is, we use the slope of the secant line as an approximation of the slope of the tangent line. See Figure 1.9. So first we need two initial values, and the iterative formula becomes

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}, x_0, x_1 \in [a, b], n \geq 1$$

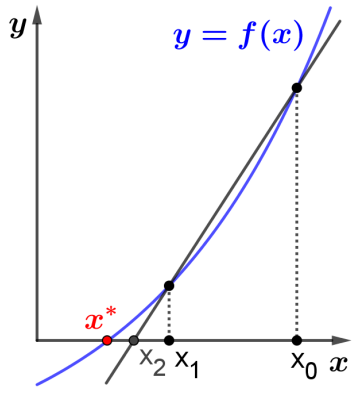
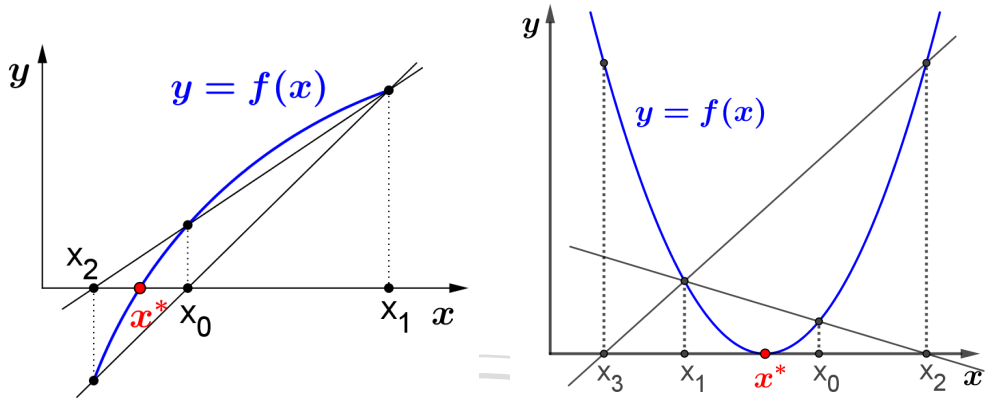


Figure 1.9: Secant method.

DOI:10.6814/NCCU202001843

We can see that the secant method is very much like the false position method, and actually "they have the same order of convergence $\frac{1+\sqrt{5}}{2}$" [6]. And like Newton's method, we may encounter some divergent cases. See Figure 1.10.



(a) stuck in a loop (b) divergent

Figure 1.10: Divergent cases of secant method.

1.1.7 Steffensen's method

"Steffensen's method is similar to Newton's method. It also achieves quadratic convergence, but without using derivatives" [7]. To apply this method, as in the fixed-point iteration, first we write $f(x) = 0$ in the form of $x = g(x)$. Let $x_{n+1} = g(x_n)$ where $n \geq 1$ and x_0 is given. In https://en.wikipedia.org/wiki/Steffensen%27s_method explains how Steffensen's method works:

Assume that three consecutive values of the sequence $\{x_n\}$ are known, say x_n, x_{n+1} and x_{n+2}. Then we can use Aitken's delta-squared process to accelerate convergence of the sequence $\{x_n\}$. Since fixed-point iteration is linearly convergent, for n large enough, we have

$$\frac{x_{n+1} - x^*}{x_n - x^*} \approx \frac{x_{n+2} - x^*}{x_{n+1} - x^*}$$

then

$$x^* \approx \frac{x_n x_{n+2} - x_{n+1}^2}{x_n - 2x_{n+1} + x_{n+2}}$$

or

$$x^* \approx x_n - \frac{(x_{n+1} - x_n)^2}{x_n - 2x_{n+1} + x_{n+2}} \tag{1.1.3}$$

DOI:10.6814/NCCU202001843

And the root x^* given by (1.1.3) is the successive approximation in Steffensen's method.

Another form of Steffensen's method is introduced in [8]:

$$x_{n+1} = x_n - \frac{(f(x_n))^2}{f(x_n + f(x_n)) - f(x_n)} \qquad n \geq 0 \tag{1.1.4}$$

If we write (1.1.4) as

$$x_{n+1} = x_n - \frac{f(x_n)}{\frac{f(x_n+h)-f(x_n)}{h}} \qquad \text{where } h = f(x_n)$$

then clearly (1.1.4) is just replacing the derivative $f'(x_n)$ in Newton's method by the slope of secant line near $(x_n, f(x_n))$ with step size $f(x_n)$.

To explain (1.1.3) and (1.1.4), observe the second term in the right hand side, for the numerator, notice that for large n,

$$f(x_n) \approx g(x_n) - x_n = x_{n+1} - x_n$$

And for the denominator,

$$\begin{aligned} x_n - 2x_{n+1} + x_{n+2} &= (x_{n+2} - x_{n+1}) - (x_{n+1} - x_n) \\ &\approx f(x_{n+1}) - f(x_n) \\ &= f(g(x_n)) - f(x_n) \\ &\approx f(x_n + f(x_n)) - f(x_n) \end{aligned}$$

Hence (1.1.3) and (1.1.4) do have the same meaning.

1.1.8 Halley's method

"Halley's method applies for functions with a continuous second derivative and the rate of convergence is cubic" [9]. Like Newton's method, Halley's method starts with an initial guess x_0, and then produces a sequence of approximations by the iterative formula:

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2(f'(x_n))^2 - f(x_n)f''(x_n)} \qquad n \geq 0 \tag{1.1.5}$$

DOI:10.6814/NCCU202001843

One way to obtain (1.1.5) is using Taylor series. In Newton's method, the tangent line of $y = f(x)$ at $x = x_n$ is

$$y = f(x_n) + f'(x_n)(x - x_n)$$

which can be seen as a linear approximation of $y = f(x)$ at $x = x_n$. That is, Newton's method uses first order of Taylor series:

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + ...$$

If we use second order of Taylor series:

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{f''(x_n)}{2}(x - x_n)^2 + ...$$

we have

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

write as

$$0 = f(x_n) + (x_{n+1} - x_n)[f'(x_n) + \frac{f''(x_n)}{2}(x_{n+1} - x_n)]$$

so

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

then use the result of Newton's method:

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

to replace $x_{n+1} - x_n$ gives (1.1.5).

"Halley's method can also be derived by applying Newton's method to $g(x) = \frac{f(x)}{\sqrt{|f'(x)|}}$" [10]. Notice that a root of $g(x)$ is a root of $f(x)$. So with initial value x_0 and use iterative formula

$$x_{n+1} = x_n - \frac{g(x)}{g'(x)}$$

with

$$g'(x) = \frac{2(f'(x))^2 - f(x)f''(x)}{2f'(x)\sqrt{|f'(x)|}}$$

gives (1.1.5) as well.

DOI:10.6814/NCCU202001843

1.2 Nonlinear equations of several variables

When we consider higher dimension, the bisection method and the false position method no longer work since the intermediate value theorem fails. For example, the spiral :

$$\begin{cases} x = 0.5tcos(4t) \\ y = 0.5tsin(4t) \quad 0 \le t \le 2\pi \\ z = 0.25t^2 - 0.8 \end{cases}$$

Although $A(0.79, 0, -0.18)$ and $B(-1.18, 0, 0.59)$ are points of the graph in different sides of

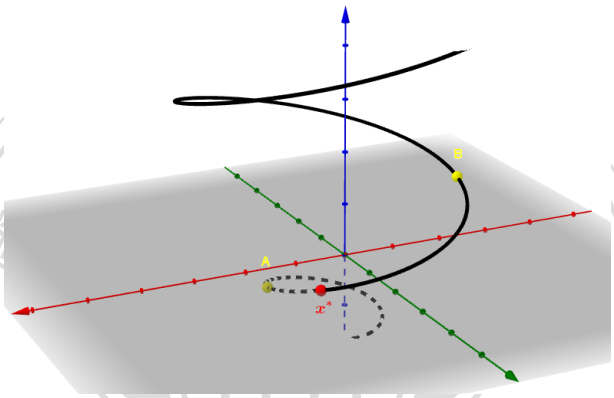


Figure 1.11: A spiral.

xy plane, there is no root between segment AB.

Hence for nonlinear equations of several variables, we don't have much reference. The methods we use is mainly modifications of Newton's method. In [11], Yuri Levin and Adi BenIsrael introduce directional Newton method for differentiable $f : \mathbb{R}^n \to \mathbb{R}$ with limitations for direction vectors, gradient of f, and second derivative (Hessian matrix) of f to reach quadratic convergence. In [12], HengBin An and ZhongZhi Bai present directional secant method, "a variant of directional Newton method, which also reaches quadratic convergence under suitable assumptions and has better numerical performance". In [13], HengBin An and ZhongZhi Bai give Broyden method. "It is more efficient, especially in high dimensions. Under suitable assumptions, Broyden method is locally superlinearly convergent and hence is a powerful alternative to directional Newton method and directional secant method."

DOI:10.6814/NCCU202001843

1.2.1 Directional Newton Method

In [11], Yuri Levin and Adi Ben-Israel introduce the directional Newton method as follows: Suppose that $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable. Given a point x^0 and a direction vector d with $\| d \| = 1$, the successive iteration is

$$x^1 := x^0 - \frac{f(x^0)}{\nabla f(x^0) \cdot d} d$$

where $\nabla f(x^0) \cdot d$ is the directional derivative of f at x^0 along d. Continuing this iteration, the formula is

$$x^{k+1} := x^k - \frac{f(x^k)}{\nabla f(x^k) \cdot d^k} d^k, k = 0, 1, ...$$

For d^k sufficiently close to the gradient $\nabla f(x^k)$ and f satisfies the assumptions:

- The gradient of f is not 'too small'.
- The second derivative (Hessian matrix) of f is not 'too large'.

directional Newton method reaches quadratic convergence.

Another choice of d^k is the unit vector $e^{m(k)}$ where $m(k)$ is the index of the component of $\nabla f(x^k)$ which has maximal absolute value. In this case, the formula becomes

$$x^{k+1} := x^k - \frac{f(x^k)}{\nabla f(x^k) \cdot e^{m(k)}} e^{m(k)}, k = 0, 1, ...$$

1.2.2 Directional Secant Method

In some situations, the function $f(x)$ may not be differentiable or the gradient $\nabla f(x)$ is uncomputable. So in [12], Heng-Bin An and Zhong-Zhi Bai present the directional secant method, "a variant of directional Newton method, which also reaches quadratic convergence under suitable assumptions and has better numerical performance".

In directional secant method, $\nabla f(x^k) \cdot d^k$ in directional Newton method is replaced by its approximation $\frac{f(x^k + t_k d^k) - f(x^k)}{t_k}$ where t_k is a prescribed step size. Hence the iterative formula becomes

$$x^{k+1} := x^k - \frac{t_k f(x^k)}{f(x^k + t_k d^k) - f(x^k)} d^k, k = 0, 1, ...$$

In [12], the authors also demonstrate some examples to compare directional Newton

DOI:10.6814/NCCU202001843

method and directional secant method. In these examples, $d^k = \frac{p^k}{\|p^k\|}$ where

$$p^k = (\frac{f(x^k + f(x^k)e^1) - f(x^k)}{f(x^k)}, ..., \frac{f(x^k + f(x^k)e^n) - f(x^k)}{f(x^k)})$$

is near to $\nabla f(x^k)$ when $f(x^k) \neq 0$ and $f(x^k) \approx 0$. The results show that:

- The choice of initial point is very important.
- Directional secant method needs a little more iteration numbers.
- Directional Newton method costs much more computing time especially when n becomes larger.

All these show the feasibility and efficiency of directional secant method.

1.2.3 Broyden Method

In directional Newton method and directional secant method, we need to compute gradient or difference quotient at each iteration in order to obtain direction vector d^k, which is time consuming and inconvenient. Hence in [13], Heng-Bin An and Zhong-Zhi Bai present Broyden method. "It is more efficient, especially in high dimensions. Under suitable assumptions, Broyden method is locally superlinearly convergent and hence is a powerful alternative to directional Newton method and directional secant method". Broyden Method works as follows:

For $f(x) = 0$ where $f : \mathbb{R}^n \to \mathbb{R}^n$ is differentiable, the classical Broyden method can be described as follows: given initial value $x^0 \in \mathbb{R}^n$, initial matrix $A_0 \in \mathbb{R}^{n \times n}$, and tolerance ϵ. Set $k = 0$. If $\| f(x^k) \| > \epsilon$, then

1. $s^k = -A_k^{-1} f(x^k)$
2. $x^{k+1} = x^k + s^k$
3. $y^k = f(x^{k+1}) - f(x^k)$
4. $A_{k+1} = A_k + \frac{(y^k - A_k s^k)(s^k)^T}{(s^k)^T s^k} = A_k + \frac{f(x^{k+1})(s^k)^T}{(s^k)^T s^k}$
5. $k := k + 1$

DOI:10.6814/NCCU202001843

where A_k is an approximation of the Jacobi matrix of f at x^k. In one dimension, it is simply secant method since $A_{k+1}s^k = y^k$.

Now apply the above to $f(x) = 0$ where $f : \mathbb{R}^n \to \mathbb{R}$. Given initial value $x^0 \in \mathbb{R}^n$, initial matrix $A_0^T \in \mathbb{R}^n$, and tolerance ϵ. Set $k = 0$. If $|f(x^k)| > \epsilon$, then we have

1. $s^k = -A_k^\dagger f(x^k)$
2. $x^{k+1} = x^k + s^k$
3. $y^k = f(x^{k+1}) - f(x^k)$
4. $A_{k+1} = A_k + \frac{(y^k - A_k s^k)(s^k)^T}{(s^k)^T s^k} = A_k + \frac{f(x^{k+1})(s^k)^T}{(s^k)^T s^k}$
5. $k := k + 1$

where $A_k^\dagger$ is the Moore-Penrose inverse of A_k.

By $A_k^\dagger = \frac{A_k^T}{\|A_k\|^2}$ and $A_k A_k^\dagger = 1$, through some computations, [13] rewrites the above in a more concise way: given initial value $x^0 \in \mathbb{R}^n$, initial matrix $A_0^T \in \mathbb{R}^n$, and tolerance ϵ. Set $\Delta_0 = -f(x^0)$ and $k = 0$. If $|f(x^k)| > \epsilon$, then

1. $x^{k+1} = x^k + \Delta_k A_0^\dagger$
2. $y^k = f(x^{k+1}) - f(x^k)$
3. $\Delta_{k+1} = \frac{f(x^{k+1})}{y^k}\Delta_k$
4. $k := k + 1$

which is the final form of Broyden method.

Compared to directional Newton method and directional secant method, numerical examples show that "although Broyden method needs more iteration numbers to reach the desired degree of accuracy, it takes much less computing time. The difference is significant when n gets larger or the initial point is not ideal. This shows the superiority of Broyden method in high dimensions".

Later in chapter 2, we will give an overview of Monte Carlo method and Support Vector Machine (SVM), then introduce a new method attached with two Python programming codes. And then in chapter 3, we will apply the new method to some examples to see its performance and limitations. Finally in chapter 4, we will summarize the results in chapter 3 and draw a conclusion about the pros and cons of the new method.

DOI:10.6814/NCCU202001843

Chapter 2

Methodology

The methods for solving nonlinear equations of several variables introduced in Chapter 1 need the assumption that f is differentiable. Hence they can't be applied to functions which don't satisfy the condition. Moreover, during the iterative process, when we need to compute gradient or difference quotient, it could be complicated, time consuming or even unable to calculate. Sometimes it is also tricky to find a suitable initial value and direction vector. Furthermore, every time we apply those methods to our target, only one single root is found. We don't have a whole picture of our solutions. To avoid the above disadvantages, we try a new method which combines Monte Carlo method and Support Vector Machine (SVM) to find roots.

First we give a basic introduction of Monte Carlo method. Its concept is using repeated random sampling and statistical analysis to obtain numerical results. It is often used in the fields of physics and mathematics. One main usage is simulating systems with randomness or modeling phenomena with significant uncertainty. Another usage is transforming the solution of unsolved problem to a parameter (such as expectation) of some kind of random distribution. In mathematics, the most common application of Monte Carlo method is in integration and optimization. Especially when the number of function evaluations grows exponentially in high dimensions. It is useful and powerful for obtaining numerical solutions to problems too complicated to solve analytically. Next in 2.1, we will introduce SVM, a classical algorithm in machine learning.

DOI:10.6814/NCCU202001843

2.1 Support Vector Machine (SVM)

SVM is a well-known algorithm in machine learning. It is easy to understand intuitively and also has a solid theoretical basis. SVM has a wide range of application. It can be used in categorization of text and hypertext, classification of images, recognition of hand-written characters and classification of proteins in medicine. So what is SVM? It is a supervised learning model and basically it is a binary linear classifier. To apply SVM, first we have to provide a set of training data to the model. These data are already marked as belonging to one class or the other according to some valued features. Then SVM model ”learns” how to classify from these training data by creating a hyper plane in the feature space. With the hyper plane, the rules of classification it learned, SVM model can assign new data to one of the two classes. For example, say we want to distinguish between apples and oranges. We may choose two features: color and weight. And the hyperplane in two-dimensional feature space is simply a straight line as showed in Figure 2.1.

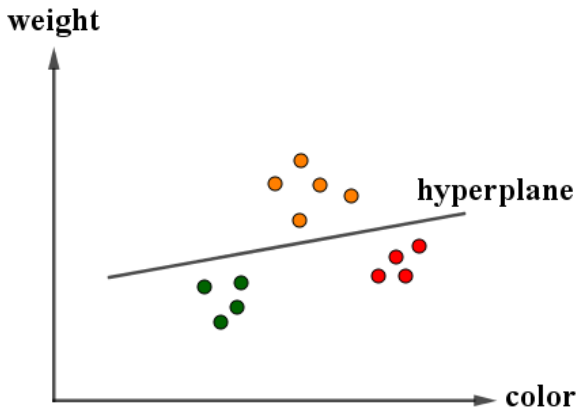


Figure 2.1: Distinguish between apples(red and green dots) and oranges(orange dots).

The key to this method is to find an optimal hyper plane. To determine the hyper plane, our goal is to maximize the margins between two classes. And the data on the boundary of the margins are called support vectors. Two-dimensional case is showed in Figure 2.2.

DOI:10.6814/NCCU202001843

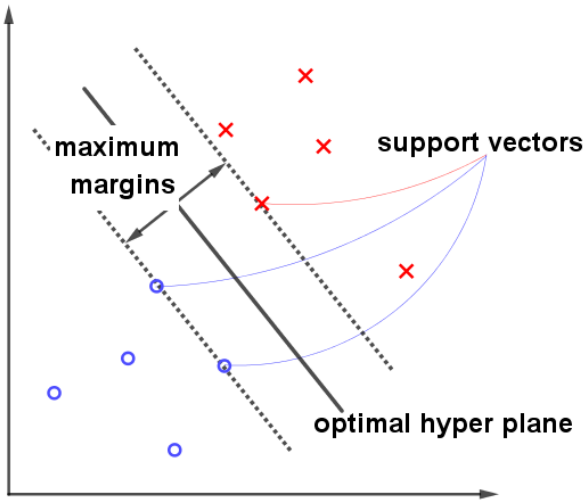


Figure 2.2: Determine the hyper plane.

But in real world, the data are unlikely being classified perfectly. So we may tolerate some deviations within certain extent. These deviations are called slack variables and we try to find the hyper plane with minimum deviations. This case is called soft margin. Also we can put a penalty parameter to adjust the influence of these deviations. That is, to control the weights between ”maximum margins” and ”minimum deviations”.

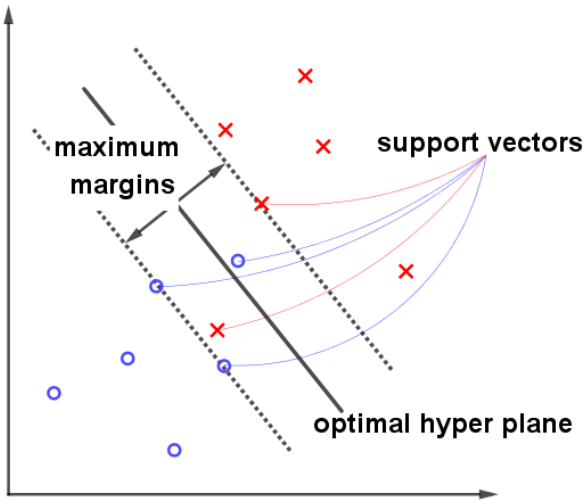


Figure 2.3: Soft margin.

In addition to linear classification, using the skill called ”kernel trick”, SVM is capable of performing nonlinear classification. The way it works is mapping featured data into a higher

DOI:10.6814/NCCU202001843

dimension space to apply a linear classification. For example, two-dimensional nonlinear data could be linearly classified in three-dimensional space with a plane as showed in Figure 2.4.

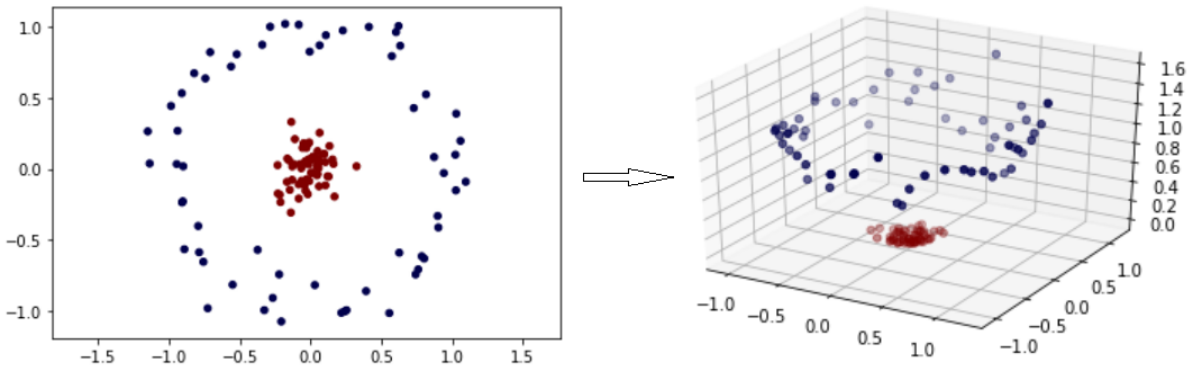


Figure 2.4: Kernel trick.

Finally, we can also apply SVM to multi-class problems by distinguish between one class and the rest (One-Versus-Rest) or between every pair of classes (One-Versus-One). The former approach OVR uses fewer classifications, but the numbers of data may be imbalanced and hence fails to classify them. The latter approach OVO is less sensitive to imbalance, but uses more classifications.

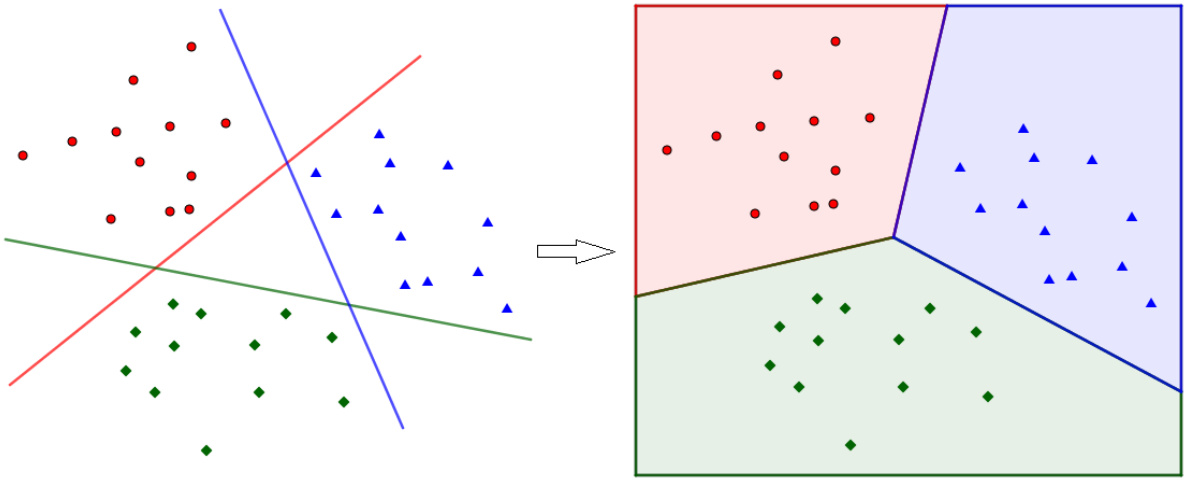


Figure 2.5: Multi-class: One-versus-Rest.

DOI:10.6814/NCCU202001843

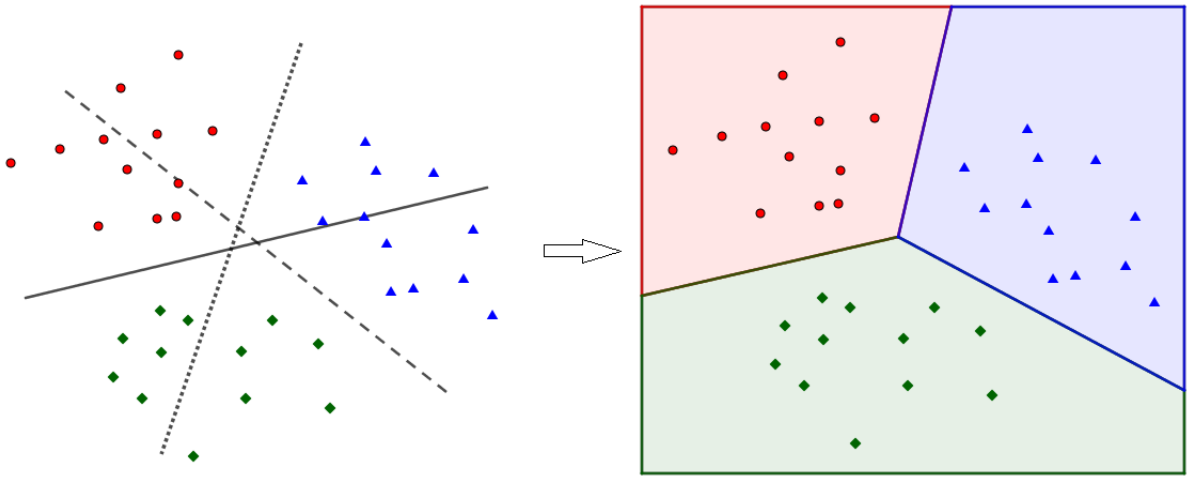


Figure 2.6: Multi-class: One-versus-One.

2.2 A New Method

Now we introduce another method for solving nonlinear equations of several variables. Suppose $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ is continuous. First we choose a desired region in the domain and randomly select sample points in the region. As to region we refer to a rectangle in two dimensional space, a cuboid in three dimensional space, etc. Compute the function values of these sample points. If we get only positive or negative values, then chances are high that there is no root in this region unless the number of sample points is too small. Now consider the case that we've got both positive and negative function values. Here these sample points are like training data with n features, and SVM model classifies them into two classes according to the sign of their function values. Now with the trained SVM model, the hyper plane could be seen as a rough approximation of the roots. Also we have a number of support vectors in each class. These support vectors are supposed to be "close" to the roots in the region. Then we randomly select the same number of sample support vectors in each class and make them into pairs. From every pair we can obtain a "hyper cuboid" by letting each coordinate of the 2^n vertices to be the minimum or maximum in each dimension of the paired support vectors. Take $n = 2$ for example, suppose (a, b) and (c, d) are paired support vectors from different classes, then we can obtain a hyper cuboid (here it is only a rectangle) with the following four vertices: $(\min\{a, c\}, \min\{b, d\})$, $(\max\{a, c\}, \min\{b, d\})$, $(\max\{a, c\}, \max\{b, d\})$ and $(\min\{a, c\}, \max\{b, d\})$. Now, to decide which hyper cuboid leads to a better opportunity for finding roots, we consider the following

DOI:10.6814/NCCU202001843

factors:

- Let r_1 be the volumetric ratio of the hyper cuboid to the region. We expect small r_1.
- Let n_1 and n_2 be the numbers of support vectors of the two classes in the hyper cuboid respectively. Let $r_2 = \frac{\max\{n_1,n_2\}}{\min\{n_1,n_2\}}$. We expect small r_2.
- Let $d = \frac{n_1+n_2}{\text{volumn of the hyper cuboid}}$ be the density of support vectors in the hyper cuboid. We expect large d.

Consider the value $r_1 + r_2 + \frac{1}{d}$ of each hyper cuboid. The smaller it is, the higher probability that we could find roots in that hyper cuboid. So we can choose several candidates to continue. Or as in our Python programming codes, we choose hyper cuboid with the smallest value to be the new region in the next iteration. And it's center point is an approximation of a root in this iteration. Continue the process, we can shrink the hyper cuboid and improve our approximation until it satisfies demanded accuracy. Now we demonstrate the method by the following Python programming codes.

Code 1 is for $f : \mathbb{R}^2 \to \mathbb{R}$ in region $[a_1, b_1] \times [a_2, b_2]$. The outputs are: approximation of a root in the region, function value of the approximation, number of iterations, and time of operations. If number of iterations is too large(>100), the iterative procedure will stop.

```
>>> import numpy as np
>>> import numpy.random as rn
>>> import random
>>> from sklearn import svm
>>> import time
...
...     # n: number of sample points, tol: tolerance of diagonal of the rectangle
>>> def root(f,a1,b1,a2,b2,n,tol):
...         start = time.process_time( )
...         ii = 0    # number of iterations
...         while np.sqrt((b1-a1)**2+(b2-a2)**2) > tol and ii <= 100:
...
...             X = [ ]
...             for i in range(n):
...                 x1 = rn.rand( )*(b1-a1) + a1
...                 x2 = rn.rand( )*(b2-a2) + a2
...                 X.append([x1,x2])
...             X = np.array(X)
...
...             Y = [ ]
```


DOI:10.6814/NCCU202001843

```
. . .       for i in range(n):
. . .           if f(X[i][0],X[i][1]) > 0:
. . .               Y.append(1)
. . .           else:
. . .               Y.append(-1)
. . .       Y = np.array(Y)
. . .
. . .       clf = svm.SVC(C=500,gamma='scale',kernel='rbf')
. . .       clf.fit(X,Y)
. . .
. . .       sv = clf.support_vectors_    # support vectors
. . .       sv_n = clf.n_support_    # numbers of support vectors in each class
. . .
. . .       c1 = sv[:sv_n[0]]    # support vectors in class 1
. . .       c2 = sv[sv_n[0]:]    # support vectors in class 2
. . .
. . .       # take k(<=3) sample support vectors in each class
. . .       k = min(int(sv_n[0] / 2),3)
. . .       c1_spl = random.sample(list(c1),k)
. . .       c2_spl = random.sample(list(c2),k)
. . .       c1_spl = np.array(c1_spl)    # sample support vectors in class 1
. . .       c2_spl = np.array(c2_spl)    # sample support vectors in class 2
. . .
. . .       den_rto = [ ]    # density and ratio of k small rectangles
. . .       for i in range(k):
. . .           aa1 = min(c1_spl[i][0],c2_spl[i][0])
. . .           bb1 = max(c1_spl[i][0],c2_spl[i][0])
. . .           aa2 = min(c1_spl[i][1],c2_spl[i][1])
. . .           bb2 = max(c1_spl[i][1],c2_spl[i][1])
. . .
. . .           n1 = 0    # number of support vectors of class 1 in the i-th small rectangle
. . .           for t in range(len(c1)):
. . .               if aa1 <= c1[t][0] <= bb1 and aa2 <= c1[t][1] <= bb2:
. . .                   n1+=1
. . .
. . .           n2 = 0    # number of support vectors of class 2 in the i-th small rectangle
. . .           for t in range(len(c2)):
. . .               if aa1 <= c2[t][0] <= bb1 and aa2 <= c2[t][1] <= bb2:
. . .                   n2+=1
. . .
. . .           # density of support vectors in the i-th small rectangle
. . .           den = (n1+n2) / ((bb1-aa1)*(bb2-aa2))
. . .           # volumetric ratio of the i-th small rectangle to the region
. . .           rto1 = ((bb1-aa1)*(bb2-aa2)) / ((b1-a2)*(b2-a2))
. . .           # the ratio of numbers of support vectors in different classes
. . .           rto2 = max(n1,n2) / min(n1,n2)
. . .           den_rto.append(rto1 + rto2 + 1/den)    # the smaller the better
```


DOI:10.6814/NCCU202001843

```
. . .
. . .                 d = np.argsort(den_rto)[0]    # index of the smallest value in den_rto
. . .                 # 4 vertices of the most recommended small rectangle
. . .                 a1 = min(c1_spl[d][0],c2_spl[d][0])
. . .                 b1 = max(c1_spl[d][0],c2_spl[d][0])
. . .                 a2 = min(c1_spl[d][1],c2_spl[d][1])
. . .                 b2 = max(c1_spl[d][1],c2_spl[d][1])
. . .
. . .                 ii+=1
. . .           end = time.process_time( )
. . .           return [(a1+b1)/2,(a2+b2)/2], f((a1+b1)/2,(a2+b2)/2), ii, end - start
```

Code 2 is for $f : \mathbb{R}^m \to \mathbb{R}$ in region $\overbrace{[a,b] \times [a,b] \times \cdots \times [a,b]}^{m}$. The outputs are: approximation of a root in the region, function value of the approximation, number of iterations, and time of operations. Moreover, in order to reduce operation time, we minus number of sample points along with the increase in number of operations.

```
>>> import numpy as np
>>> import numpy.random as rn
>>> import random
>>> from sklearn import svm
>>> import time
. . .
. . .  # m: dimension, n: number of sample points, tol: tolerance of diagonal of hyper cuboid
>>> root(f,a,b,m,n,tol):
. . .
. . .           x = [ ]
. . .           for j in range(m):
. . .                 x.append((a+b)/2)
. . .
. . .           ii = 0    # number of iterations
. . .           start = time.process_time( )
. . .
. . .           # every p iterations minus q sample points (output ii < p*(r-1) and n > q*(r-1))
. . .           p = 15
. . .           q = 1
. . .           r = 200
. . .
. . .           while np.sqrt(m*(b-a)**2) > tol:
. . .
. . .                 for i in range(1,r):
. . .                       if ii == p*i:
. . .                             n-=q
. . .                             break
. . .                       else:
. . .                             pass
```


DOI:10.6814/NCCU202001843

```
. . .
. . .                X = [ ]
. . .                for i in range(n):
. . .                    x = [ ]
. . .                    for j in range(m):
. . .                        x.append(rn.rand()*(b-a)+a)
. . .                    X.append(x)
. . .                X = np.array(X)
. . .
. . .                Y = [ ]
. . .                for i in range(n):
. . .                    if f(X[i]) > 0:
. . .                        Y.append(1)
. . .                    else:
. . .                        Y.append(-1)
. . .                Y = np.array(Y)
. . .
. . .                clf = svm.SVC(C=500,gamma='scale',kernel='rbf')
. . .                clf.fit(X,Y)
. . .
. . .                sv = clf.support_vectors_    # support vectors
. . .                sv_n = clf.n_support_    # numbers of support vectors in each class
. . .
. . .                c1 = sv[:sv_n[0]]    # support vectors in class 1
. . .                c2 = sv[sv_n[0]:]    # support vectors in class 2
. . .
. . .                k = 1    # take k=1 sample support vectors in each class
. . .                c1_spl = random.sample(list(c1),k)
. . .                c2_spl = random.sample(list(c2),k)
. . .                c1_spl = np.array(c1_spl)    # k sample support vectors in class 1
. . .                c2_spl = np.array(c2_spl)    # k sample support vectors in class 2
. . .
. . .                den_rto = [ ]    # density and ratio of k hyper cuboid
. . .                for i in range(k):
. . .
. . .                    # number of support vectors of class 1 in the i-th hyper cuboid
. . .                    n1 = 0
. . .                    for t in range(len(c1)):
. . .                        s = 0
. . .                        for j in range(m):
. . .                            if min(c1_spl[i][j],c2_spl[i][j]) <= c1[t][j] <= max(c1_spl[i]
[j],c2_spl[i][j]):
. . .                                s+=1
. . .                        if s == m:
. . .                            n1+=1
. . .
. . .                    # number of support vectors of class 2 in the i-th hyper cuboid
```


DOI:10.6814/NCCU202001843

```
. . .                    n2 = 0
. . .                    for t in range(len(c2)):
. . .                        s = 0
. . .                        for j in range(m):
. . .                            if min(c1_spl[i][j],c2_spl[i][j]) <= c2[t][j] <= max(c1_spl[i]
[j],c2_spl[i][j]):
. . .                                s+=1
. . .                        if s == m:
. . .                            n2+=1
. . .
. . .                    V = 1
. . .                    for j in range(m):
. . .                        # volume of the i-th hyper cuboid
. . .                        V = V * (max(c1_spl[i][j],c2_spl[i][j]) - min(c1_spl[i][j],c2_spl[i]
[j]))
. . .
. . .                    # density of support vectors in the i-th hyper cuboid
. . .                    den = (n1 + n2) / V
. . .                    # volumetric ratio of the i-th hyper cuboid to the region
. . .                    rto1 = V / (b-a)**m
. . .                    # ratio of numbers of support vectors in different classes
. . .                    rto2 = max(n1,n2) / min(n1,n2)
. . .                    den_rto.append(rto1 + rto2 + 1/ den)    # the smaller the better
. . .
. . .                d = np.argsort(den_rto)[0]    # index of the smallest value in den_rto
. . .
. . .                x = [ ]
. . .                for j in range(m):
. . .                    # the center point of the most recommended hyper cuboid
. . .                    x.append( (min(c1_spl[d][j],c2_spl[d][j]) + max(c1_spl[d][j],c2_spl[d]
[j])) / 2 )
. . .
. . .                aa = [ ]
. . .                for j in range(m):
. . .                    aa.append(min(c1_spl[d][j],c2_spl[d][j]))
. . .                bb = [ ]
. . .                for j in range(m):
. . .                    bb.append(max(c1_spl[d][j],c2_spl[d][j]))
. . .                a = min(aa)
. . .                b = max(bb)
. . .
. . .                ii+=1
. . .            end = time.process_time( )
. . .            t = end - start
. . .            return x, f(x), ii, t
```


DOI:10.6814/NCCU202001843

Chapter 3

Results

Now we apply the method in chapter 2 to some examples and see how it works. The equipment we use is a laptop with Intel(R) Core(TM) i5-8265U CPU 1.60GHz up to 1.80GHz and RAM 8.00 GB.

Example 1. $f(x,y) = \sqrt{x^2+y^2-1} + \ln(4-x^2-y^2)$

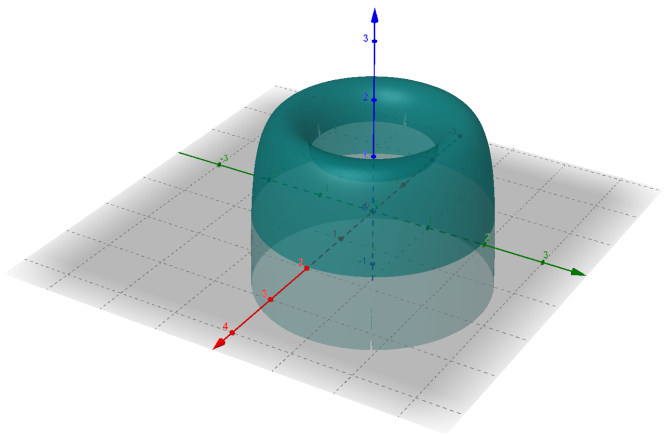


Figure 3.1: Graph of example 1.

The roots of f on xy plane is similar to a circle with radius a little less than 2. Now we use Code 1: root(f,1,3,-1,1,1000,tol=10**(-12)) to find roots in $[1,3] \times [-1,1]$ with 1000 sample points in each iteration and with tolerance of diagonal of the rectangle 10^{-12}. Table 3.1 is the

DOI:10.6814/NCCU202001843

results of 10 consecutive tests and is presented in the order of approximation of a root, function value of the approximation, number of iterations, and time of operations.

Table 3.1: Test results of example 1 with Code 1: root(f,1,3,-1,1,1000,tol=10**(-12)).

approximation	function value	number	time
(1.8393485316192288, 0.6556725842999632)	-5.114131340633321e-12	27	0.484375
(1.8404225031688544, -0.6526519504314735)	-4.672262576832509e-12	33	0.59375
(1.941254269782079, 0.21128515949820706)	2.9465319073551655e-13	41	0.71875
(1.7597515054678885, -0.8463948236929669)	3.2591707110896095e-12	32	0.453125
(1.8555758543841576, -0.6082333492989722)	-7.824629832953178e-12	32	0.515625
(1.9291048578468786, -0.3027606414355924)	2.4069635173873394e-12	39	0.640625
(1.8859593483090873, -0.5062281057965792)	-7.256639733554948e-12	37	0.625
(1.8806978984805065, 0.5254378871203835)	4.381828233590568e-12	37	0.703125
ValueError: The number of classes has to be greater than one; got 1 class			
(1.8191277005208264, -0.7098478483235422)	-1.8112178423734804e-12	38	0.546875

Table 3.1 shows that out of 10 test results, we have found 9 roots(approximations). Each root takes less than 1 seconds. And we have one "ValueError: The number of classes has to be greater than one; got 1 class". That means during the process of iteration, function values of sample points in the rectangle have the same sign. If we draw these roots in Table 3.1 on xy plane, we may obtain a very rough contour of the roots in $[1,3]\times[-1,1]$.

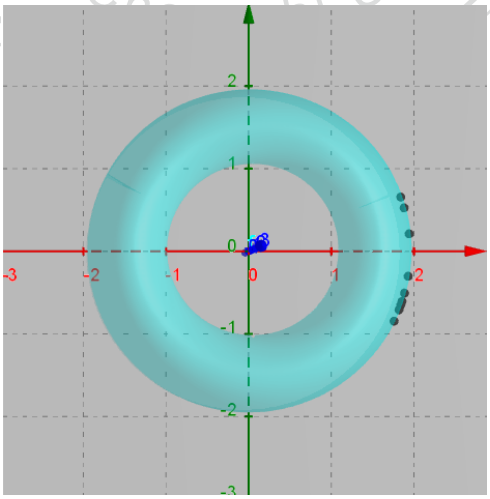
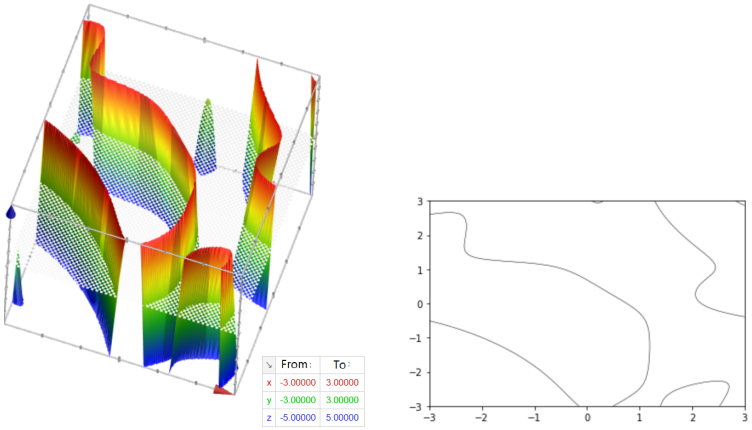


Figure 3.2: Roots founded by Code 1.

DOI:10.6814/NCCU202001843

Example 2. $f(x,y) = x^2 - 2y^2 + xy - 30\sin(x+y) + 20\cos xy$



(a) Graph of the function (b) Roots on xy plane

Figure 3.3: Graph of example 2.

Similarly, we use Code 1: root(f,-1,1,-1,1,1000,tol=10**(-12)) to find roots in $[-1,1] \times [-1,1]$ with 1000 sample points in each iteration and with tolerance of diagonal of the rectangle 10^{-12}. Test results are as follows.

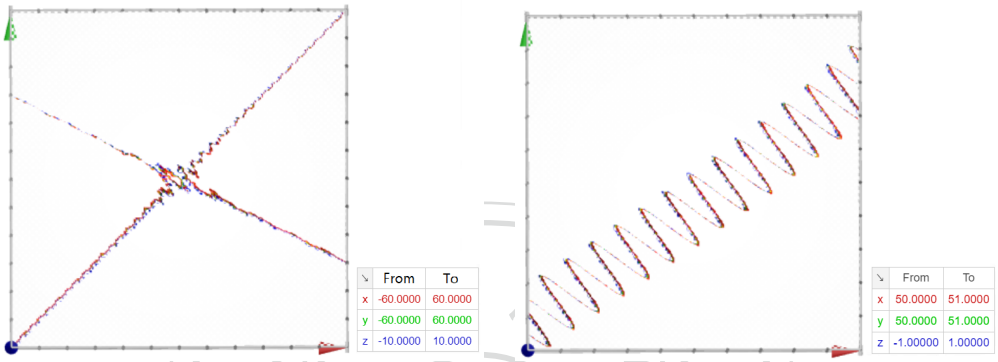
Table 3.2: Test results of example 2 with Code 1: root(f,-1,1,-1,1,1000,tol=10**(-12)).

approximation	function value	number	time
(0.46993790636813626, 0.2622453677771487)	-1.7763568394002505e-14	33	0.65625
(0.7684688297613869, -0.012505477809996928)	3.5136338283336954e-12	40	0.484375
(0.22176752901561408, 0.48857630079124126)	-9.78772618509538e-12	30	0.46875
IndexError: index 0 is out of bounds for axis 0 with size 0			
(-0.005547432373856567, 0.6929394537660047)	-2.9096725029376103e-12	46	0.8125
(0.5621218361256275, 0.17890958192506992)	3.7196912217041245e-12	37	0.703125
(-0.25695433093787506, 0.8894629520497923)	7.059242079776595e-12	32	0.5
(0.8050933184240345, -0.04871131491812492)	-7.226219622680219e-12	35	0.578125
(0.42833405733942814, 0.29992808858241676)	1.0043521569969016e-11	25	0.359375
(0.46601152390386, 0.265797865621633)	4.764189043271472e-12	32	0.65625

One of the test results is ”IndexError: index 0 is out of bounds for axis 0 with size 0”, that means the code can’t find the most recommended rectangle during the iterations.

DOI:10.6814/NCCU202001843

When we zoom out to see function $f(x,y) = x^2 - 2y^2 + xy - 30\sin(x+y) + 20\cos(xy)$, its graph on xy plane behaves like a hyperbola, see Figure 3.4 (a). If we try to find roots in regions that contain no root, Code 1 will return "ValueError: The number of classes has to be greater than one; got 1 class". Now we try to find roots in regions away from the origin, say $[50,51] \times [50,51]$, see Figure 3.4 (b). Table 3.3 is the test results.



(a) Roots in $[-60,60] \times [-60,60]$ (b) Roots in $[50,51] \times [50,51]$

Figure 3.4: Roots of example 2 in different scales.

Table 3.3: Test results of example 2 with Code 1: root(f,50,51,50,51,1000,tol=10**(-12)).

approximation	function value	number	time
(50.37108241597812, 50.30605405694342)	7.59312612785834e-11	36	0.734375
IndexError: index 0 is out of bounds for axis 0 with size 0			
(50.216759062176685, 50.209210588582806)	4.789209029354424e-10	25	0.484375
(50.91835623563183, 50.67395719627402)	3.6215297427588666e-10	33	0.484375
(50.08210621719444, 50.15376290895588)	3.0322144795036365e-10	32	0.53125
(50.493655231184896, 50.352278685604674)	9.49995637711254e-11	32	0.5
(50.76882719293579, 50.64478487427914)	2.929940734475167e-10	29	0.5625
(50.060893746994225, 50.18334780684751)	1.6474643871333683e-10	32	0.53125
(50.56335822951033, 50.48002788886255)	-9.930056776852325e-11	41	0.65625
(50.04464454039062, 50.12500465230691)	4.3338976851714506e-10	40	0.546875

Compare Table 3.3 with Table 3.2, the accuracy of function values apparently is worse in Table 3.3 than in Table 3.2. If we raise the standard of tolerance, Code 1 even fails in

DOI:10.6814/NCCU202001843

$[50, 51] \times [50, 51]$, see Table 3.4 where tolerance of diagonal of the rectangle is 10^{-15} and ii represents the maximal value allowed in number of iterations.

Table 3.4: Raise the standard of tolerance.

approximation	function value	number	time
Code 1: root(f,-1,1,-1,1,1000,tol=10**(-15)),ii=100			
(0.7461596882606241, 0.009033344195159302)	3.552713678800501e-15	41	0.578125
(0.2376360543484806, 0.4740324090606552)	0.0	34	0.546875
(0.39138819644970957, 0.3334828756138792)	3.552713678800501e-15	42	0.671875
(0.8130843644528254, -0.056777966137313576)	7.105427357601002e-15	52	0.78125
ValueError: The number of classes has to be greater than one; got 1 class			
Code 1: root(f,50,51,50,51,1000,tol=10**(-15)),ii=100			
(50.6987226809441, 50.52614710988952)	5.950795411990839e-14	101	2.078125
(50.3123398028162, 50.34944789599142)	7.105427357601002e-13	101	2.234375
(50.29289226163097, 50.258444639339814)	-8.219203095904959e-12	101	1.953125
(50.25733872551366, 50.21940854261966)	2.327027459614328e-12	101	1.9375
(50.204504343031914, 50.29298780543669)	-7.545963853772264e-12	101	2.03125
Code 1: root(f,50,51,50,51,1000,tol=10**(-15)),ii=1000			
(50.31353419225536, 50.295244027029945)	8.739675649849232e-13	1001	21.921875
ValueError: The number of classes has to be greater than one; got 1 class			
(50.842699814232994, 50.628859347893254)	6.235012506294879e-13	1001	21.640625
(50.71581863271457, 50.680914087131555)	-8.562039965909207e-13	1001	22.21875
(50.89625216670158, 50.76244021179818)	-6.986411449361185e-12	1001	21.90625
Code 1: root(f,50,51,50,51,1000,tol=10**(-15)),ii=5000			
(50.735470991990084, 50.62593175023604)	3.984368390774762e-12	5001	126.84375
IndexError: index 0 is out of bounds for axis 0 with size 0			
(50.297804446513624, 50.254821899553995)	4.4364512064021255e-12	5001	127.078125
(50.30503027546648, 50.30686591842504)	8.107736704232593e-12	5001	128.3125
(50.762567213039844, 50.589032786249454)	5.153211191100127e-12	5001	129.90625

From Table 3.4, we can see that in region $[50, 51] \times [50, 51]$, even if we allow number of

DOI:10.6814/NCCU202001843

iterations up to 5000 times, Code 1 still failed to obtain an approximation of root that satisfies the required tolerance and the accuracy of function value doesn't improve along with the increase in number of iterations. The reason is that sine and cosine function in Python are approximated by series of polynomials and rounding errors will accumulate when independent variables are away from the origin. To avoid this, we can translate the graph so that the independent variables are near the origin. For example, let $g(x, y) = f(x + 50, y + 50)$ and use the periodic property of sine and cosine function, we have $g(x, y) = x^2 + xy - 2y^2 + 150x - 150y - 30\sin(x + y + 100 - 32\pi) + 20\cos((x + 50)(y + 50) - 796\pi)$. Then apply Code 1 to g in $[0, 1] \times [0, 1]$ with tolerance of diagonal of the rectangle 10^{-15}, see Table 3.5.

Table 3.5: Test results of g with Code 1: root(g,0,1,0,1,1000,tol=10**(-15)).

approximation	function value	number	time
(0.46818599428272567, 0.38802043952474363)	3.952393967665557e-14	45	1.0625
(0.4462804306335044, 0.41798056459405625)	-2.9309887850104133e-13	37	0.8125
(0.4691852199445329, 0.38665487925191777)	2.7533531010703882e-14	41	0.859375
(0.5605807588099267, 0.48208915471610153)	-5.380584866543359e-12	55	1.046875
(0.2017860554505237, 0.2971099319591042)	-5.5209170568559784e-12	42	0.6875
(0.37514876948450493, 0.3632664037200196)	-4.413358567489922e-12	45	1.078125
(0.6542098954593933, 0.5872723444048912)	2.76578759894619e-12	44	0.8125
(0.12206118061828386, 0.09784852619758204)	5.53157519789238e-12	44	0.71875
ValueError: The number of classes has to be greater than one; got 1 class			
(0.7855399066982386, 0.7079424206421457)	-5.346834086594754e-12	47	0.875

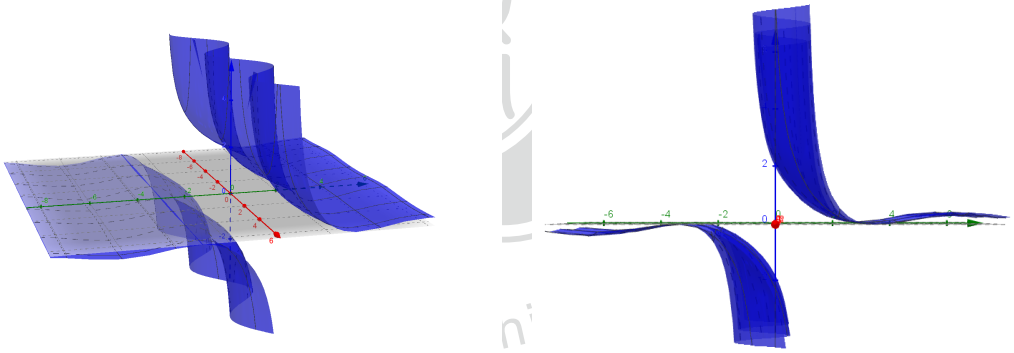
Now move our approximations of roots back to $[50, 51]\times[50, 51]$ and compute their function values, see Table 3.6.

DOI:10.6814/NCCU202001843

Table 3.6: Approximations of roots of f in $[50, 51] \times [50, 51]$.

approximation of root	function value
(50.46818599428272567, 50.38802043952474363)	1.354028000832841e-12
(50.4462804306335044, 50.41798056459405625)	1.91491267287347e-12
(50.4691852199445329, 50.38665487925191777)	1.0031975250512914e-11
(50.5605807588099267, 50.48208915471610153)	2.831068712794149e-12
(50.2017860554505237, 50.2971099319591042)	1.950439809661475e-12
(50.37514876948450493, 50.3632664037200196)	-2.568611989772762e-12
(50.6542098954593933, 50.5872723444048912)	3.382183422218077e-12
(50.12206118061828386, 50.09784852619758204)	-1.6697754290362354e-13
ValueError: The number of classes has to be greater than one; got 1 class	
(50.7855399066982386, 50.7079424206421457)	2.8919089345436078e-12

Example 3. $f(x, y) = \frac{\cos y+1}{y-\sin x}$



(a) Graph of the function (b) Projection on yz plane

Figure 3.5: Graph of example 3.

By function formula, we know the roots are points on xy plane with y-coordinate equals to $(2k + 1)\pi$ where $k \in \mathbb{Z}$. However, points around the roots belong to the same side of xy plane, see Figure 3.5 (b). Therefore, it fails when we apply Code 1 to this function. For example, Code 1: root(f,-1,1,3,4,1000,10**(-12)) returns "ValueError: The number of classes has to be greater than one; got 1 class". Moreover, for this function, on regions that contain discontinuous points, the code may still return an answer, but it's

DOI:10.6814/NCCU202001843

not a root. For example, one test result of Code 1: root(f,-1,1,-1,1,1000,tol=10**(-12)) is ([0.5584914658730595, 0.5299074807530488], 17034614928732.754, 37, 0.765625). The approximation of root is point (0.5584914658730595, 0.5299074807530488), but it's function value is 17034614928732.754, obviously not a root of $f(x) = 0$.

Example 4. $f : \mathbb{R}^m \to \mathbb{R}$ defined by $f(x) = \sum_{i=1}^{m} x_i \exp(1 - x_i^2)$ where $x = (x_1, x_2, ..., x_m)$

Apparently, the origin $\overbrace{(0, 0, ..., 0)}^{m}$ is a root of $f(x) = 0$. First, we apply Code 2: root(f,-0.9,1,10,250,10**(-12)) to find roots in $\overbrace{[-0.9, 1] \times [-0.9, 1] \times \cdots \times [0.9, 1]}^{10}$ with 250 samples points in each iteration and with tolerance of diagonal of the hyper cuboid 10^{-12}. Here is the first test result:

- The approximation of root :
  (1.2844263924027142e-15, 7.711984581422935e-14, 9.976145297340012e-14, -1.0961014831765263e-14, -1.120713405053965e-14, 3.436442229894976e-14, -8.308598458235813e-14, -9.737252214208976e-14, -1.9765518862372757e-14, -3.7386907986968744e-14)
- Function value of the approximation: -1.2843592136232755e-13
- Number of iterations: 310
- Time of operations: 3.328125 seconds.

In Table 3.7, we only list the last three items of outputs since all approximations of roots suggest the origin.

DOI:10.6814/NCCU202001843

Table 3.7: Test results of example 4 with Code 2: root(f,-0.9,1,10,250,10**(-12)).

function value	number	time
1.3779698843748291e-14	292	3.15625
-1.332515734999162e-14	317	3.40625
1.1987897774143587e-13	322	3.4375
-8.386255861714453e-14	310	3.34375
-1.0989147744968376e-13	332	3.515625
-2.0372613787811113e-13	318	3.40625
4.910130259939433e-14	305	3.359375
2.8895734888874934e-13	305	3.265625
-1.2903165704502644e-13	300	3.265625

Next, we apply Code 2: root(f,-0.9,1,m,250,10**(-12)) in $\overbrace{[-0.9,1]\times[-0.9,1]\times\cdots\times[0.9,1]}^{m}$ with dimension $m = 15, 20, 25, 30, 35, 40$. Table 3.8 list one test result of each dimension. Since all test results indicate the same root: the origin, we still only present the last three items of the outputs.

Table 3.8: One test result in different dimensions.

Dimension	function value	number	time
15	-1.1705003069935705e-13	471	7.65625
20	-2.1807932674276025e-13	618	13.109375
25	-8.214164474230035e-14	767	19.71875
30	1.7075573329547067e-13	899	28.546875
35	-4.4167018237416875e-14	1063	38.828125
40	-1.7446726440345464e-14	1184	51.265625

In Table 3.8, time of operations increases to over 50 seconds when dimension comes to 40. If we want to decrease operation time, we may shrink the region and adjust number of sample points. Now for each dimension $m = 10, 15, 20, 25, 30, 35, 40$, we take 10 consecutive test

DOI:10.6814/NCCU202001843

results and compute the average and standard deviation of operation time, see Table 3.9. Also we can find a quadratic function to fit these data points as in Figure 3.6.

Table 3.9: The statistics of operation time of 10 consecutive test results in different dimensions.

dimension	average of operation time	SD of operation time
10	3.35	0.10
15	7.52	0.22
20	13.08	0.41
25	19.72	0.38
30	28.41	0.61
35	38.66	1.58
40	50.82	1.71

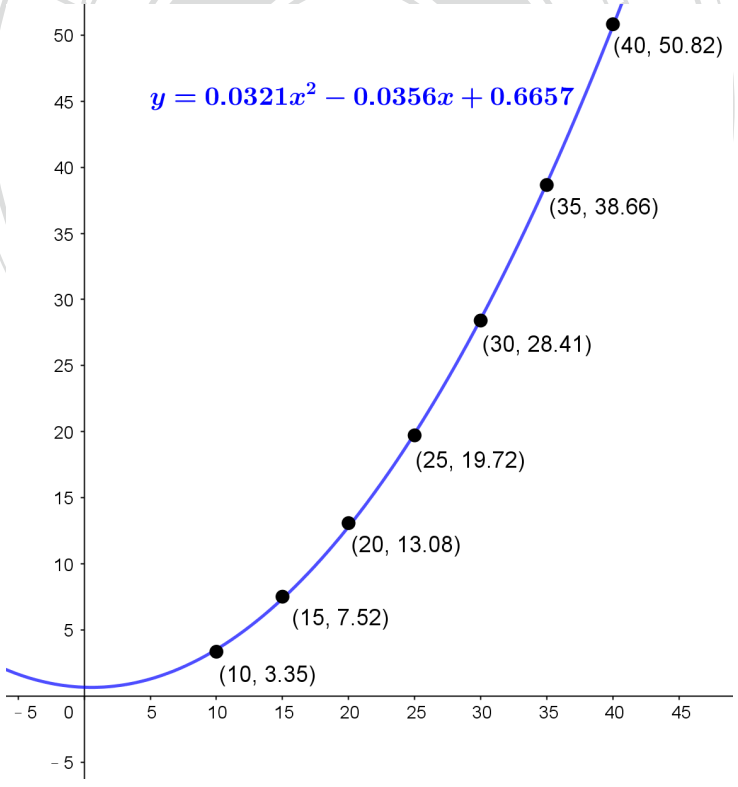


Figure 3.6: Dimension and operation time.

DOI:10.6814/NCCU202001843

Chapter 4

Conclusion

We propose a new method which combines Monte Carlo method and Support Vector Machine(SVM) to solve nonlinear equations of several variables. The new method has the following advantages:

- It only requires function to be continuous, not necessarily differentiable.
- It needs neither to compute gradient or difference quotient nor to find a proper initial value and direction vector.
- For designated region in the domain, it can tell if there are roots in this region as long as we throw enough sample points. And it can give several recommended smaller regions that contain roots and continue to improve the accuracy of roots.
- We can cut the region into partition and work in parallel. This could raise efficiency and save time.
- Even if it doesn't achieve the desired accuracy in regions away from the origin because it takes too many iterations or too much operation time, we still have pretty good approximations of roots.
- Operation time is not exponentially increasing along with the increase of dimension.

However, the method also has some disadvantages:

- It can't deal with multiple roots of even multiplicity.

DOI:10.6814/NCCU202001843

- For regions contain infinitely many roots, it can’t find all roots since it randomly takes sample points in the region.
- In high dimensions, it’ll take time to determine suitable number of sample points to have better performance in operation time.

DOI:10.6814/NCCU202001843

Bibliography

[1] Bouchaib Radi and Abdelkhalak El Hami. *Advanced Numerical Methods with Matlab 2 Resolution of Nonlinear, Differential and Partial Differential Equations*. John Wiley & Sons, Incorporated, 2018.

[2] D. D. Wall. The order of an iteration formula. *Mathematics of Computation*, 10(55):167–168, Jan 1956.

[3] J. H. Wegstein. Accelerating convergence of iterative processes. *Communications of the ACM*, 1(6):9–13, Jan 1958.

[4] 張榮興. *VISUAL BASIC 數值解析與工程應用*. 高立圖書, 2002.

[5] Charles Houston. Gutzler. An iterative method of wegstein for solving simultaneous nonlinear equations, 1959.

[6] J.a. Ezquerro, A. Grau, M. Grau-Sánchez, M.a. Hernández, and M. Noguera. Analysing the efficiency of some modifications of the secant method. *Computers & Mathematics with Applications*, 64(6):2066–2073, 2012.

[7] G. Liu, C. Nie, and J. Lei. A novel iterative method for nonlinear equations. *IAENG International Journal of Applied Mathematics*, 48:444–448, 01 2018.

[8] Manoj Kumar, Akhilesh Kumar Singh, and Akanksha Srivastava. Various newton-type iterative methods for solving nonlinear equations. *Journal of the Egyptian Mathematical Society*, 21(3):334–339, October 2013.

[9] G. Alefeld. On the convergence of halley's method. *The American Mathematical Monthly*, 88(7):530, August 1981.

DOI:10.6814/NCCU202001843

[10] George H. Brown. On halley's variation of newton's method. *The American Mathematical Monthly*, 84(9):726, November 1977.

[11] Yuri Levin and Adi Ben-Israel. Directional newton methods in n variables. *Mathematics of Computation*, 71(237):251–263, May 2001.

[12] Heng-Bin An and Zhong-Zhi Bai. Directional secant method for nonlinear equations. *Journal of Computational and Applied Mathematics*, 175(2):291–304, March 2005.

[13] Heng-Bin An and Zhong-Zhi Bai. 關於多元非線性方程的 broyden 方法. *Mathematica Numerica Sinica*, 26(4):385–400, November 2004.

DOI:10.6814/NCCU202001843