Solving High-Dimensional Nonlinear Equations with Infinite Solutions by the SVM Visualization Method

Yu-Yuan Lin and Jeng-Nan Tzeng(✉)

National Chengchi University, No. 64, Sec. 2, ZhiNan Road, Wenshan District, Taipei City 16302, Taiwan R.O.C.

Abstract. There are many standard mathematical methods for solving nonlinear equations. But when it comes to equations with infinite solutions in high dimension, the results from current methods are quite limited. Usually these methods apply to differentiable functions only and have to satisfy some conditions to converge during the iteration. Even if they converge, only one single root is found at a time. However, using the features of SVM, we present a simple fast method which could tell the distribution of these infinite solutions and is capable of finding approximation of the roots with accuracy up to at least 10^{-12}. In the same time, we could also have a visual understanding about these solutions.

Keywords: High-dimensional nonlinear equations · Infinite solutions · SVM

1 Introduction

In physics, chemistry and engineering, many problems appear in the form of nonlinear equations. Using numerical methods, the solutions can be computed to a desired degree of accuracy.

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 [3,4]. The most famous is Newton's method which can be seen as a special case of fixed-point iteration. Newton's method has many extensions and variations. Secant method [5] and Steffensen's method [6,7] both replace the derivative in Newton's method by the slope of secant line. And Halley's method [8,9] uses second order of Taylor series instead of first order linear approximation in Newton's method.

For nonlinear equations involve several variables, we don't have much reference. In [10], Yuri Levin and Adi BenIsrael introduce directional Newton method for differentiable $f : \mathbb{R}^n \to \mathbb{R}$ with conditions for direction vectors, gradient of f, and second derivative (Hessian matrix) of f to reach quadratic convergence.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022

In [11], 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 [12], HengBin An and ZhongZhi Bai give Broyden method for nonlinear equation in several variables, a modification of the classical Broyden method: 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. $k := k + 1$

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 simplifications of the recursive process, [12] 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. Under suitable assumptions, Broyden method is locally super-linearly convergent and semi-locally convergent. Compared to directional Newton method and directional secant method, numerical experiments show that Broyden method is more efficient and the difference is significant especially in high dimension or the initial point is not ideal. This shows the superiority of Broyden method.

Later in Sect. 2, we will give an overview of Monte Carlo method and Support Vector Machine (SVM), after that illustrate our new method. And then in Sect. 3, we will apply the new method to some examples to see its performance and limitations. Finally in Sect. 4, we will summarize the results in Sect. 3 and draw a conclusion about the pros and cons of the new method.

2 Methodology

The above methods for solving nonlinear equations usually 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 point and direction vector. For example, consider this quite simple equation: $f(x, y) = x^2 + y^2 - 1 = 0$. First we randomly choose an initial value $x^0 = [1, 1]$ and initial matrix $A_0^T = [2, 2]$. Then apply the Broyden method with tolerance 10^{-3} as showed in Table 1:

Table 1. The broyden method.

$x^0 = [1, 1]$ and $A_0^T = [2, 2]$	
x^k	Function value
$x^1 = [0.750.75]$	0.125
$x^2 = [0.785714290.78571429]$	0.23469387755102034
$x^3 = [0.862126250.86212625]$	0.48652332755709105
...	
$x^{20} = [2106.503718872106.50371887]$	8874714.835218174

We can see that this combination fails, so does $x^0 = [1, 1]$ and $A_0^T = [-2, -2]$. A successful try is $x^0 = [1, 1]$ and $A_0^T = [1, 0]$. But even we succeed, every time we apply those methods to our target, only one single root is found. It is not efficient to deal with infinite solutions. And 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.

The basic concept of Monte Carlo method 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.

Support vector machine(SVM) is a supervised learning model for classification and regression analysis in machine learning. Basically, it is a binary linear classifier. First we give a set of training data to the model. These data are already marked as belonging to one category 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 categories. 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 dimension space to apply a linear classification. We can also apply SVM to multiclass problems by distinguish between one category and the rest(one-versus-all) or between every pair of categories(one-versus-one).

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 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 this hyper cuboid. So we can choose several candidates to continue. Or we could choose the 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 of root until it satisfies demanded accuracy.

3 Examples

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

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

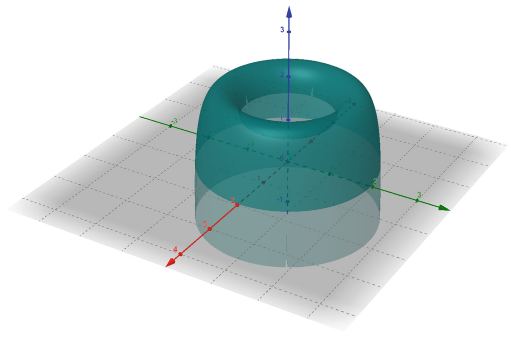


Fig. 1. Graph of Example 1.

The graph of f is presented in Fig. 1. The roots of f on xy plane is similar to a circle with radius a little less than 2. Now we use Python code 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}. Test results are presented in the order of approximation of a root, function value of the approximation, number of iterations, and time of operations.

Table 2. Test results of Example 1 with python code.

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 2 shows that out of 10 test results, we have found 9 roots (approximations). Each root takes less than 1 s. 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 2 on xy plane, we may obtain a very rough contour of the roots in $[1, 3] \times [-1, 1]$ as showed in Fig. 2.

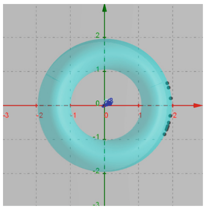
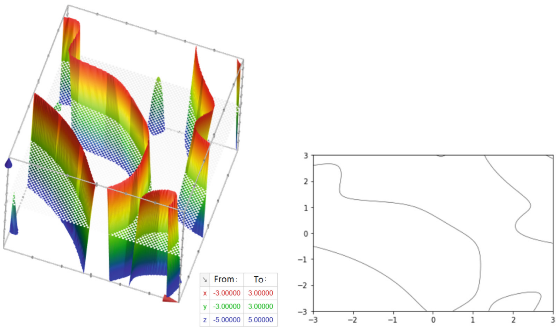


Fig. 2. Roots founded by python code.

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

The graph and roots of f around the origin are presented in Fig. 3(a) and Fig. 3(b) respectively. Similarly, in example 2 we also use Python code 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 in Table 3.



	From:	To:
x	-3.00000	3.00000
y	-3.00000	3.00000
z	-5.00000	5.00000

(a) Graph of the Function (b) Roots on xy Plane

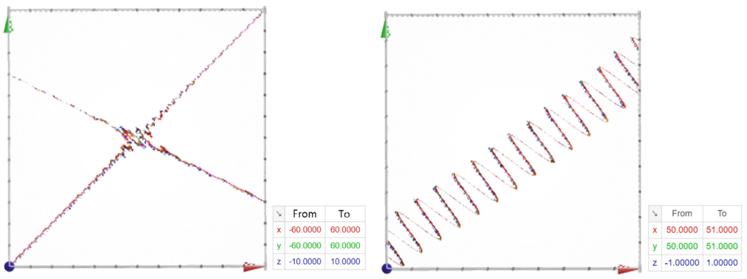
Fig. 3. Graph of Example 2.

Table 3. Test results of Example 2 with python code in $[-1,1] \times [-1,1]$.

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 in Table 3 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.

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 Fig. 4 (a). If we try to find roots in regions that contain no root, we will get "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 Fig. 4 (b). Table 4 is the test results.



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

Fig. 4. Roots of Example 2 in different scales.

Compare Table 4 with Table 3, the accuracy of function values apparently is worse in Table 4 than in Table 3. If we raise the standard of tolerance, Python code even fails in $[50,51] \times [50,51]$, see Table 5 where the tolerance of diagonal of the rectangle is 10^{-15} and ii in the table represents the maximal value allowed in number of iterations.

Table 4. Test results of Example 2 with python code in $[50, 51] \times [50, 51]$.

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

Table 5. 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 5, we can see that in region $[50, 51] \times [50, 51]$, even if we allow number of iterations up to 5000 times, Python code 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 Python code to g in $[0, 1] \times [0, 1]$ with tolerance of diagonal of the rectangle 10^{-15}, see Table 6. After that we move our approximations of roots back to $[50, 51] \times [50, 51]$ and compute their function values, see Table 7.

Table 6. Test results of g with python code in $[0, 1] \times [0, 1]$.

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

Table 7. 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. $f(x, y) = \frac{\cos y + 1}{y - \sin x}$

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 Fig. 5(b). Therefore, when we apply Python code to this function, it fails and we'll get "ValueError: The number of classes has to be greater than one; got 1 class". Moreover, for this function, on regions that contain discontinuous points, Python code may still return an answer, but it's not a root. For example, one test result of Python code is the approximation of root (0.5584914658730595, 0.5299074807530488), but it's function value is 17034614928732.754, obviously not a root of $f(x) = 0$.

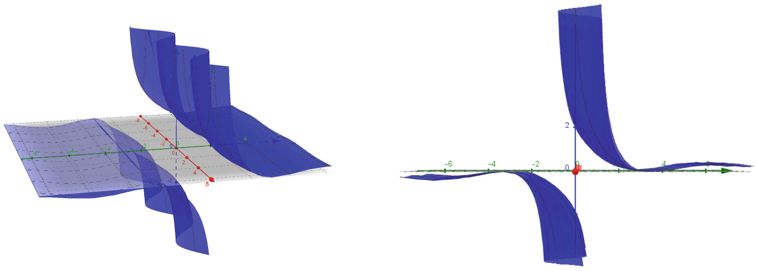

(a) Graph of the Function (b) Projection on yz Plane

Fig. 5. Graph of example 3.

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 Python code 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 s.

In Table 8, we only list the last three items of outputs since all approximations of roots suggest the origin.

Table 8. Test results of example 4 with python code.

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 Python code 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 9 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 9. 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 9, time of operations increases to over 50 s 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 results and compute the average and standard deviation of operation time, see Table 10. Also we can find a quadratic function to fit these data points as in Fig. 6.

Table 10. 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

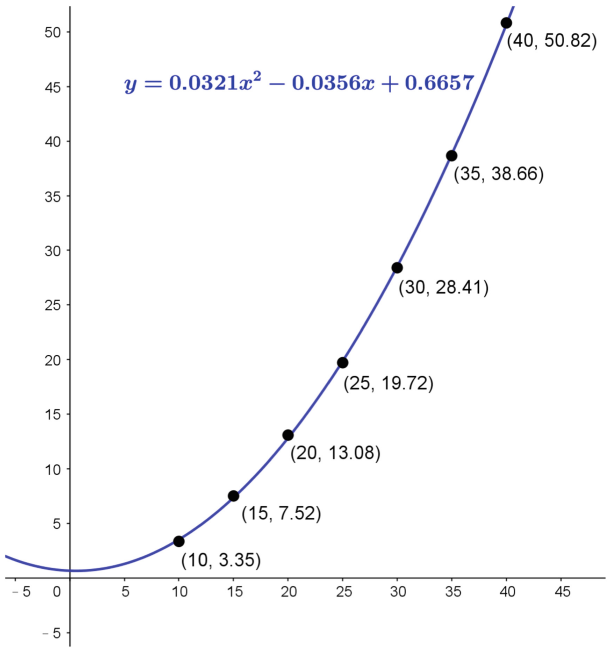


Fig. 6. Dimension and operation time.

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.
- 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.

There are still some aspects that can be improved. For example, when we select the next smaller region to continue the iteration, we may change the weigh of each factor to raise the efficiency, or maybe there are better filters to decide the next smaller region. Moreover, the Python code we use now in dimension m (greater than 2) is designed to find roots in hyper cubic $\overbrace{[a,b]\times[a,b]\times\cdots\times[a,b]}^{m}$. The following step could be applying the method to more high-dimensional examples in arbitrary region to see its performance.

References

1. Radi, B., El Hami, A.: Advanced Numerical Methods with Matlab 2 Resolution of Nonlinear. Differential and Partial Differential Equations. John Wiley & Sons, Incorporated (2018)
2. Wall, D.D.: The order of an iteration formula. Math. Comput. **10**(55), 167–168 (1956). https://doi.org/10.1090/s0025-5718-1956-0080981-0
3. Wegstein, J.H.: Accelerating convergence of iterative processes. Comm. ACM **1**(6), 9–13 (1958). https://doi.org/10.1145/368861.368871
4. Gutzler, C.H.: An iterative method of Wegstein for solving simultaneous nonlinear equations. (1959)
5. Ezquerro, J.A., Grau, A., Grau-Sánchez, M., Hernández, M.A., Noguera, M.: Analysing the efficiency of some modifications of the secant method. Comput. Math. Appl. **64**(6), 2066–2073 (2012). https://doi.org/10.1016/j.camwa.2012.03.105
6. Liu, G., Nie, C., Lei, J.: A novel iterative method for nonlinear equations. IAENG Int. J. Appl. Math. **48**, 44–448 (2018)
7. Kumar, M., Singh, A.K., Srivastava, A.: Various newtontype iterative methods for solving nonlinear equations. J. Egypt. Math. Soc. **21**(3), 334–339 (2013). https://doi.org/10.1016/j.joems.2013.03.001

8. Alefeld, G.: On the convergence of Halley's method. Am. Math. Mon. **88**(7), 530 (1981). https://doi.org/10.2307/2321760
9. George, H.: Brown: on Halley's variation of newton's method. Am. Math. Mon. **84**(9), 726 (1977). https://doi.org/10.2307/2321256
10. Levin, Y., Ben-Israel, A.: Directional Newton methods in n variables. Math. Comput. **71**(237), 251–263 (2001). https://doi.org/10.1090/s0025-5718-01-01332-1
11. An, H.-B., Bai, Z.-Z.: Directional secant method for nonlinear equations. J. Comput. Appl. Math. **175**(2), 291–304 (2005). https://doi.org/10.1016/j.cam.2004.05.013
12. An, H.-B., Bai, Z.-Z.: Math. Num. Sin. **26**(4), 385–400 (2004)