高阶常微分方程初值问题的神经网络解法

A Neural Network Solution to Initial Value Problem for Higher Order Ordinary Differential Equations

聂蕴哲、夏睿杰、朱哲浩

Yunzhe Nie, Ruijie Xia, Zhehao Zhu

学校：世界联合学院（中国·常熟）

United World College Changshu China

所属省份：江苏

Province: Jiangsu

指导老师：沈佳麒

Supervisor: Jiaqi Shen

Abstract

This research developed an algorithm for solving the initial value problem of ordinary differential equation (ODE), based on the traditional Picard iteration method. This algorithm explores machine learning in artificial intelligence, constructing two correlated radial basis function networks (RBFNs) to ensure efficiency and precision within the integral process. The research thereby provides a fast and accurate method to integral calculation as a byproduct. Because RBFN requires solutions to system of linear equations of multiple variables, the ill-conditioned matrix problem inevitably deterred calculation process. The currently adapted truncated singular value decomposition (TSVD) method is an effective solution for ill-posed matrices. Theoretical and experimental analysis is performed on error estimation/calculation, with both giving favorable results in terms of precision and efficiency.

Key words: Initial value problem, ordinary differential equation system, Picard-Lindelöf Theorem, neural network, RBF network, singular value decomposition

1 Introduction

This paper develops an algorithm to solve initial value problem for higher order ODEs, which have extended application in math, physics, etc. The classic Picard–Lindelöf theorem which provides the basis for initial value problems to first-order ODEs with given initial conditions is proved to be applicable to systems of ODEs, and thus their equivalent higher order ODEs.

Yet the greatest problem with Picard iteration is that in order to obtain an accurate solution, the method requires many repeated integral calculation, thus posing an extreme demand for the speed and accuracy of the integral process. Hence, in this work, a neural network-based integral calculator is developed to solve this problem.

Neural network, as a branch of machine learning, is useful in clustering, pattern recognition, and function approximation. Organized in layers, neural network can process and transform the signals they acquire. By adjusting the synaptic weights of each connection, neural network has proven to be robust in extracting features form sample data, and thus efficient for our purpose.

The architecture of neural network this work chosen is radial basis function networks (RBFNs). RBFNs use radial basis functions as activation functions and are shown to be robust in approximating functions. Detailed network constitution is introduced in the following sections.

Picard-iteration is later integrated into the RBFNs and form the final algorithm. It is then put to test with various integrals and later higher order ODEs and systems of ODEs. Error estimation and other problems are also discussed in length.

2 Method

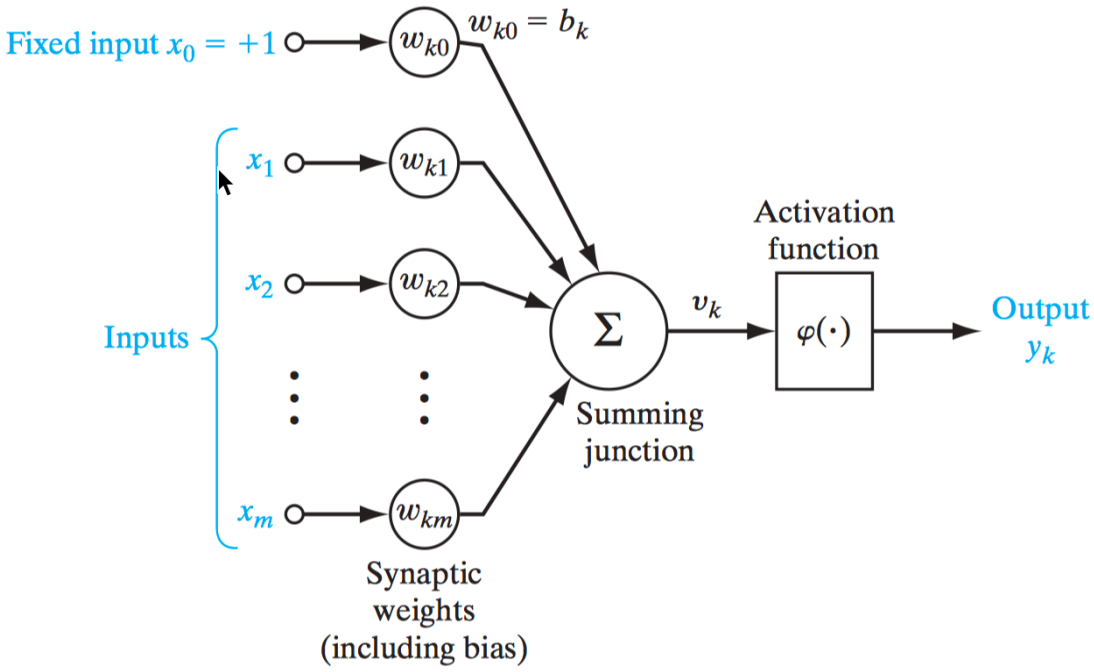
2.1 Platform

Hardware (processor): 2.7 GHz Intel Core i7

Software: MATLAB R2016b (9.1.0.441655)

2.2 RBFN introduction

The Neural Network, as precursor and representative of Artificial Intelligence, has wide-spread appliance in multiple academic fields (Kalogirou 2001, Dougherty 1995, Maier et al. 2000). Normally, an artificial neuron can be identified as a structure, through which one or multiple input variables can be processed to give off one or multiple output results. The figure below displays the structural bases of a single neuron.



Haykin, Simon S. 2009. Neural Networks And Learning Machines. Upper Saddle River: Pearson Education.

Fig. 1 $x_0, x_1, x_2, \ldots, x_m$ represent inputs (x_0 represents the bias), $\omega_{k0}, \omega_{k1}, \omega_{k2}, \ldots, \omega_{km}$ represent synaptic weights, $\varphi(\cdot)$ represents the activation function, and y_k represents the output. Hence, $y_k = \varphi(v_k) = \varphi(b_k + \sum_{i=1}^{m} \omega_{ki} x_i)$.

Inter-connecting multiple neurons is defined as a neural network. The output of the network would be determined by inputs and weights. Hence, to a certain extent, the trained weights of a network can be recognized as knowledge (Haykin 2009).

The following **Fig.2** demonstrates the structure of our training RBFN.

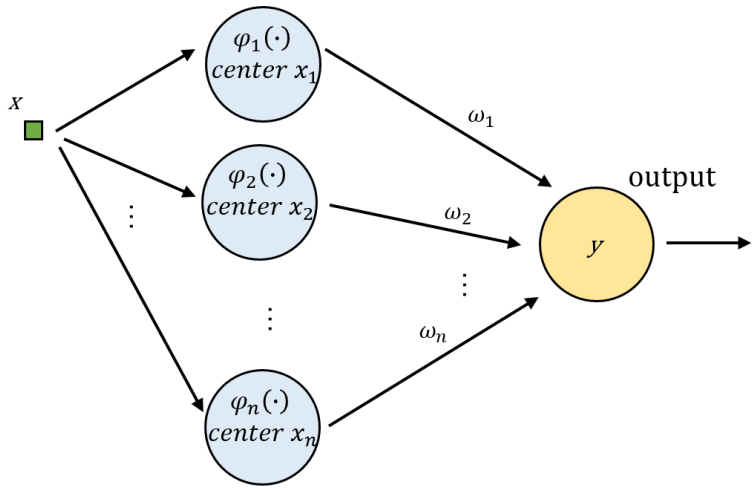


Fig. 2 This is a figure for the training RBFN. Green cube X represents the input. In our research, there is only one input, for the integral process solely relies on single variable. Also, no bias is needed due to the requirement of RBFN. Blue circles represent processing unit, function of which, $\varphi_1(\cdot), \varphi_2(\cdot), \dots, \varphi_n(\cdot)$, will be discussed further in the following paragraphs. $\omega_1, \omega_2, \dots, \omega_n$ represents the synaptic weights. Yellow circle y represents the output results.

Functions φ_i $(i = 1, 2, \dots)$ is the radial basis function as well as the active function of the corresponding neuron. Hence, $y = \sum_{i=1}^{n} \omega_i \varphi_i(x)$, where $\varphi_i(x) = \varphi(|x - x_i|)$. This function φ normally is constructed in one of the three following forms:

a) **Multi-quadratic function** $\varphi(x) = (x^2 + c^2)^{\frac{1}{2}}, c > 0$ is a constant, $x \in \mathbb{R}$

b) **Inverse multi-quadratic function** $\varphi(x) = (x^2 + c^2)^{-\frac{1}{2}}, \mathrm{c} > 0$ is a constant, $x \in \mathbb{R}$

c) **Gaussian Function** $\varphi(x) = e^{\left(-\frac{x^2}{2\sigma^2}\right)}, \sigma > 0$ is a constant, $x \in \mathbb{R}$

Powell (1988) discussed that applying multi-quadratic function in creation of interpolation matrix can approach a smoother mapping between the input and output at a higher precision. Furthermore, the original function of multi-quadratic function is less complex. Therefore, the neural network in this research applies multi-quadratic function as activation function and we

chose the constant $c = 1$. If $y = g(x) = \sum_{i=1}^{n} \omega_i \varphi_i\,(x)$ has the same value upon all interpolation points $x_1, x_2, \dots, x_n$ as the derivative function $f(x)$, then its primitive function

$$G(x) = \int \sum_{i=1}^{n} \omega_i \varphi_i(x)\,dx = \sum_{i=1}^{n} \omega_i \int \varphi_i\,(x)\,dx \dots (1)$$

can be considered as the approximate function of $F(x)$, the primitive function of $f(x)$. In order to obtain the interpolation effect above, certain weight ω_i $(i = 1, 2, \dots)$ must be selected. Supposing $d_i = f(x_i)$, the training of the network would be equivalent to looking for the solution to the displayed system of equation

$$\begin{bmatrix} \varphi_{11} & \varphi_{12} & \cdots & \varphi_{1N} \\ \varphi_{21} & \varphi_{22} & & \varphi_{2N} \\ \vdots & & \ddots & \vdots \\ \varphi_{N1} & \varphi_{N2} & \cdots & \varphi_{NN} \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_N \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \end{bmatrix}$$

in which $N = n$ represents the number of neurons as well as the number of interpolations. $\varphi_{kN} = \varphi(x_k - x_N)$ in which $= 1, 2, \dots, N$. Micchelli (1986) proved the following theorem:

Theorem 1 *If $\{x_i\}_{i=1}^{N}$ are N distinct set of points upon $\mathbb{R}^n$, then former $N \times N$ interpolation matrix as shown above is invertible.*

The **Theorem 1** ensured the accessibility of algorithm, as well as the uniqueness of the weights.

There after we can construct the other RBFN backwards calculating the integral, since the weights of the two functions are identical. In order to obtain the approximate primitive function $F(x)$ of $f(x)$, we construct the second similar RBFN presented below.

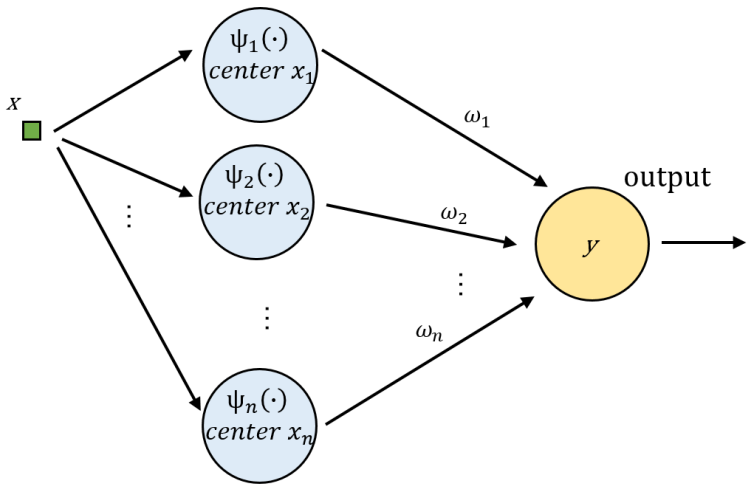


Fig. 3 This is a figure for the output neural network. The functions of processing units are $\psi_1(\cdot), \psi_2(\cdot), \dots, \psi_n(\cdot)$. Other parts of the RBFN are exactly the same as the training RBFN.

For $i = 1, 2, \dots, n$, we define

$$\psi_i(x) = \int \varphi_i(x)dx = \int \sqrt{1 + (x - x_i)^2}\, dx$$

$$= \frac{x - x_i}{2}\sqrt{1 + (x - x_i)^2} + \frac{1}{2} ln\, (x - x_i + \sqrt{1 + (x - x_i)^2})$$

which is used as the activation function for this neural network. Therefore, the output

$$y = \sum_{i=1}^{n} \omega_i \psi_i(x) = \sum_{i=1}^{n} \omega_i \int \varphi_i(x)dx$$

which according to (1), could be converted into

$$= \int \sum_{i=1}^{n} \omega_i \varphi_i(x)\, dx = G(x)$$

Therefore, the output y of this output neural network can be used as the approximate primitive function of $f(x)$.

The link between the training RBFN and the output neural network is that they share common weights $(\omega_1, \omega_2, \dots, \omega_n)$. Therefore, the output neural network can work independently of the training RBFN, thus increasing its efficiency.

3 Result

We obtained two main results from our works: an RBFN-based integral calculator (3.1) and an algorithm of solving initial value problem for a class of ordinary differential equations (3.2).

3.1 Integral calculator

With the two RBFNs stated above in the method part, we thus acquired an integral calculator with universality, speediness, and accuracy. It can be used for Riemann integrable function $f(x), x \in [a, b]$ with a corresponding primitive function.

3.1.1 Error estimation

Assume that $g(x)$ is the output we get from the learning network, from which we need to estimate the error of the output network, i.e., the error of the integral calculator. We denote the actual error of the learning network as $e(x) = |f(x) - g(x)|$, and the actual error of the output network as $E(x) = |F(x) - G(x)|$, where $F(x)$ and $G(x)$ are corresponding primitive functions. With scaling and the first mean value theorem for Integrals, we could get

$$E(x) = \left| \int_a^b \big(f(x) - g(x)\big) dx \right|$$

$$\leq \int_a^b |f(x) - g(x)|\, dx$$

$$= \int_a^b e(x)\, dx \ldots (2)$$

Therefore, we could obtain the estimated error for the desired integrated function from our calculation of the leaning network. By first mean value theorem for definite integrals (2) can be further converted to $\mu_e(b - a)$, where μ_e denotes the mean value of $e(x)$ on $[a, b]$.

Although we cannot directly obtain μ_e, but according to central limit theorem (Feller 1968), the variable

$$E = \frac{1}{N}\sum_{i=1}^{N} e_i$$

is normally distributed with mean $\mu_E = \mu_e$ and variance $\sigma_E = \sigma_e/\sqrt{N}$. When the sample size N is extremely large, σ_E is so small that we can ignore its influence and use one sample of E to estimate μ_e. As shown in **Table 1**, the true error falls into the range of estimated error, thus proving the calculator's ability to perform error estimation.

3.1.2 Testing

We select a total of 12 typical definite integrals to test the reliability and accuracy of the integral calculator. The value of parameters and integral interval are randomized integers in the range of $[1,10]$. Because our integral calculator is only built for the situation of upper limit greater than the lower limit, we will switch these two values if the lower limit is greater than the upper limit. The selected function, integrating range, computed value, true value, estimated error, true error, calculating time, and number of points in interpolation are listed in the table below (**Table 1**).

	Selected function	Computed value	True value	True error	True error within the range of Estimated error	Fractional error	Calculation time (s)	Interpolation points
1	$\int_1^2 \frac{dx}{x(2x+5)^2}$	0.011324103455991	0.011324103741955	$-2.859639990082652\times10^{-10}$	Yes	-2.53×10^{-8}	0.008749	100
2	$\int_4^7 \frac{\sqrt{9x+1}}{x^2}dx$	0.740059514530003	0.740059514877045	$-3.470419507323186\times10^{-10}$	Yes	-4.69×10^{-10}	0.003593	100
3	$\int_2^3 \frac{dx}{x^2-1}$	0.202732529491186	0.202732554054082	$-2.456289585617277\times10^{-8}$	Yes	-1.21×10^{-7}	0.003461	100
4	$\int_5^6 \frac{dx}{(9x^2+8)^2}$	$1.306672204748338\times10^{-5}$	$1.306672212884901\times10^{-5}$	$-8.136562915941024\times10^{-14}$	Yes	-6.23×10^{-9}	0.003643	100
5	$\int_1^4 \frac{dx}{7x^2+x+10}$	0.066335794283077	0.066335793973844	$3.092331252174674\times10^{-10}$	Yes	-4.66×10^{-9}	0.003722	100
6	$\int_8^{10} \frac{\sqrt{x^2+100}}{x^2}dx$	0.335272825323045	0.335272827959249	$-2.636203755290722\times10^{-9}$	Yes	-7.86×10^{-9}	0.003586	100
7	$\int_5^7 x\sqrt{x^2-16}\,dx$	54.190189242362976	54.190189111918315	$1.304446612948595\times10^{-7}$	Yes	2.41×10^{-9}	0.003825	100
8	$\int_3^5 x^2\sqrt{81-x^2}\,dx$	$2.598458279371262\times10^{2}$	$2.598458278114559\times10^{2}$	$1.256702830687573\times10^{-7}$	Yes	4.84×10^{-10}	0.055682	100
9	$\int_6^9 \sqrt{5x^2+10x+3}\,dx$	56.859989166259766	56.859990098027190	$-9.317674241060558\times10^{-7}$	Yes	-1.64×10^{-8}	0.012548	100
10	$\int_5^9 \sqrt{(x-4)(10-x)}\,dx$	11.039684817194939	11.039684861042277	$-4.384733820472775\times10^{-8}$	Yes	-3.97×10^{-9}	0.009749	100
11	$\int_3^4 e^{2x}\cos^3 5x\,dx$	$3.697187500000000\times10^{2}$	$3.695991815613422\times10^{2}$	0.119568438657780	Yes	3.24×10^{-4}	0.005073	100
12	$\int_4^9 x^{10}(\ln x)^7\,dx$	$5.428115537920000\times10^{11}$	$5.428118794232649\times10^{11}$	$-3.256312648925781\times10^{5}$	Yes	6.00×10^{-7}	0.003251	100

Table 1 Testing results and error analysis of the integral calculator

The integral calculator turns out to be very accurate from cases 1 to 10. The absolute value for true error in each case is smaller than 10^{-6} with an average of 1.26×10^{-7}, the fractional error around or smaller than 10^{-7} with an average of 1.89×10^{-8}. The average calculation time is 1.09×10^{-2}, which is also very short, indicating the speediness of the calculator. The variety of the selected definite integral demonstrated the universality of our integral method. However, for case 11 and 12, their errors are both too large (case 11 has a fractional error of 4.80×10^{-3} and case 12 -6.00×10^{-7}). To solve the problem, we did two groups of experiment, and the result was presented below. (**Table 2 and 3**)

	Selected function	Computed value	True value	True error	Fractional error	Calculation time	Interpolation points
11.1	$\int_3^4 e^{2x}\cos^3 5x\,dx$	3.697187500000000 $\times10^2$	3.695991815613422 $\times10^2$	0.119568438657780	3.24×10^{-4}	0.005073	100
11.2	$\int_3^4 e^{2x}\cos^3 5x\,dx$	3.696562500000000 $\times10^2$	3.695991815613422 $\times10^2$	0.057068438657780	1.54×10^{-4}	0.049433	300
11.3	$\int_3^4 e^{2x}\cos^3 5x\,dx$	3.695625000000000 $\times10^2$	3.695991815613422 $\times10^2$	-0.036681561342220	9.93×10^{-5}	0.170855	500
11.4	$\int_3^4 e^{2x}\cos^3 5x\,dx$	3.690625000000000 $\times10^2$	3.695991815613422 $\times10^2$	-0.536681561342220	1.45×10^{-3}	2.319108	1000

Table 2 Error analysis of case 11 by increasing interpolation points

	Selected function	Computed value	True value	True error	Fractional error	Calculating time	Interpolation points
12.1	$\int_4^9 x^{10}(\ln x)^7\,dx$	5.428115537920000 $\times10^{11}$	5.428118794232649 $\times10^{11}$	-3.256312648925781 $\times10^5$	-6.00 $\times10^{-7}$	0.003251	100
12.2	$\int_4^9 x^{10}(\ln x)^7\,dx$	5.428121108480000 $\times10^{11}$	5.428118794232649 $\times10^{11}$	$2.314247351074219\times10^5$	-4.26 $\times10^{-7}$	0.160807	500
12.3	$\int_4^9 x^{10}(\ln x)^7\,dx$	5.428118814720000 $\times10^{11}$	5.428118794232649 $\times10^{11}$	$2.048735107421870\times10^3$	-3.77 $\times10^{-9}$	2.306977	1000

Table 3 Error analysis of case 12 by increasing interpolation points

In both cases, we try to increase the interpolation points to increase accuracy and to reduce error. However, we arrived at two distinct results from case 11, 12.

In case 11, the fractional error only reduced by 2×10^{-4} when the interpolation points increased to 300, 6×10^{-5} when increased to 500. However, when it is increased to 1000, the

fractional error even increased by two orders of magnitude. This is a strange phenomenon, because normally increasing interpolation points will decrease the error. We will discuss this finding further in the discussion part about ill-conditioned matrix.

In case 12, the reason for the true error being so high (approximately at the magnitude of 10^5) is that the true value is also very large (at the magnitude of 10^{11}). If we look at the fractional error, we will find the value (-6.00×10^{-7}) just exceeding the normal range for a little bit. Therefore, the error is just slightly higher than normal ones. By increasing interpolation points to 500, the improvement isn't that obvious. But when it is increased to 1000, the fractional error decreased by two orders of magnitude (3.77×10^{-9}), proving the effectivity of increasing interpolation point in this case.

3.2 Picard Iteration at Higher Dimension

The traditional Picard Iteration method is validated and applied in first-degree ODEs. Numerical Methods for higher dimensional ODEs have been discussed as a readily attainable side product of the theorem. This algorithm would hereby be explained and validated, along with the error estimation for stand-alone iteration process.

3.2.1 Theorem and corollary

Theorem 2 *Let* $\boldsymbol{y} = (y_1, y_2, \dots, y_1) \in \mathbb{R}^n, x \in \mathbb{R}.\ \boldsymbol{f}(x, \boldsymbol{y}) = \big(f_1(x, \boldsymbol{y}), f_2(x, \boldsymbol{y}), f_3(x, \boldsymbol{y}), \dots, f_n(x, \boldsymbol{y})\big)$*, that*

$$\boldsymbol{f}: \mathbb{R}^{n+1} \to \mathbb{R}^n$$
$$(x, \boldsymbol{y}) \longmapsto \boldsymbol{f}(x, \boldsymbol{y})$$

The norm of vector is defined as following:

$$|\boldsymbol{y}| = max(|y_1|, |y_2|, \dots, |y_n|) \dots (3)$$

For the following ODE initial value problem:

$$\begin{cases} \dfrac{d\boldsymbol{y}}{dx} = f(x, \boldsymbol{y}) \\ \boldsymbol{y}(x_0) = \boldsymbol{y}_0 \end{cases} \dots (4)$$

Let D *be the following rectangular enclosed domain:*

$$|x - x_0| \le a$$
$$|y_i - y_{0i}| \le b$$

where a, b *are positive constants and* $i = 1, 2, \dots, n$*.*

Suppose that $f(x, \boldsymbol{y})$ *satisfies Lipschitz condition for* y*,* $(x, y) \in D$*, such that*

$$|f(x, \boldsymbol{y}) - f(x, \boldsymbol{z})| \le L|\boldsymbol{y} - \boldsymbol{z}|$$

for constant $L > 0$ and $f(x, \boldsymbol{y})$ continuous on D, a solution for equation (4) *exists and is unique on interval $I = [x_0 - h, x_0 + h]$; upon which*

$$h = min\left\{a, \frac{b}{M}\right\}, M > max\{|\boldsymbol{f}(x, \boldsymbol{y})|, (x, \boldsymbol{y}) \in D\}$$

The proof for this theory is given by (Ding and Li 1991), but it is only for Picard iteration of one-dimensional ODEs. Method of the proof is generally applicable to equation (4).

However, there are certain notations redefined due to the use of vectors:

a) the absolute value is replaced with vector norm, as defined in (3);

b) the rectangular domain D is replaced with cuboid domain D.

Particularly, the iterative formula for Picard sequence is

$$\boldsymbol{y_n} = \boldsymbol{y_0} + \int_{x_0}^{x} \boldsymbol{f}\big(x, \boldsymbol{y_{n-1}}(t)\big)dt$$

where $\boldsymbol{y_0}$ denotes the initial value.

Corollary *Suppose that for the Picard sequence $\boldsymbol{y}_n(x) \to \boldsymbol{y}(x)$, i.e. $\{\boldsymbol{y}_k\}$ converges uniformly on $\boldsymbol{y}$, where $\boldsymbol{y}$ is the accurate solution for (3). $|\boldsymbol{y}_n - \boldsymbol{y}| \leq C\left(\frac{|Lh|^{n+1}}{(n+1)!}\right)$, where $C = \left(\frac{M}{L}\right)e^{\theta Lh}, 0 < \theta < 1$. The definition of h is the same as shown above.*

Proof From one process of the former proof, there is an inequality that

$$|\boldsymbol{y}_{n+1} - \boldsymbol{y}_n| \leq \frac{M}{L}\left(\frac{(Lh)^{n+1}}{(n+1)!}\right)$$

as $\boldsymbol{y}_n(x) \to \boldsymbol{y}(x)$, i.e. $\{\boldsymbol{y}_k\}$ converges uniformly on $\boldsymbol{y}$.

$$\because \boldsymbol{y}_n(x) = \boldsymbol{y_0}(x) + \sum_{k=1}^{n}\big(\boldsymbol{y}_k(x) - \boldsymbol{y}_{k-1}(x)\big)$$

$$\because \boldsymbol{y}(x) = \boldsymbol{y_0}(x) + \sum_{k=1}^{\infty}\big(\boldsymbol{y}_k(x) - \boldsymbol{y}_{k-1}(x)\big)$$

$$\therefore |\boldsymbol{y}(x) - \boldsymbol{y}_n(x)| = \left| \sum_{k=n+1}^{\infty} \left(\boldsymbol{y}_k(x) - \boldsymbol{y}_{k-1}(x) \right) \right|$$

$$\leq \sum_{k=n+1}^{\infty} |\boldsymbol{y}_k(x) - \boldsymbol{y}_{k-1}(x)|$$

$$\leq \frac{M}{L} \sum_{k=n+1}^{\infty} \frac{|Lh|^k}{k!}$$

Consider the remainder of the expansion of e^x, there is $\sum_{k=n+1}^{\infty} \frac{|Lh|^k}{k!} = \frac{|Lh|^k}{n+1} e^{\theta Lh}$, in which $0 < \theta < 1$. Therefore $|\boldsymbol{y}_n - \boldsymbol{y}| \leq C \left(\frac{|Lh|^{n+1}}{(n+1)!} \right)$, where constant $C = \left(\frac{M}{L} \right) e^{\theta Lh}$.

In our code, we set the number of iteration $n = 1000$. Taking $Lh < 1$, the error

$$|\boldsymbol{y}_n - \boldsymbol{y}| \leq C \left(\frac{1}{1001!} \right) \ll 10^{-8} C$$

Therefore, the error is so small that we deem it as negligible in our research.

Also, because $\frac{d^n y}{dx^n} = f(x, y)$ is equivalent to the following system of equations:

$$\begin{cases} \dfrac{dy_1}{dx} = y_2 \\ \vdots \\ \dfrac{dy_k}{dx} = y_{k+1} \\ \vdots \\ \dfrac{dy_{n-1}}{dx} = y_n \\ \dfrac{dy_n}{dx} = f(x, y_1, y_2, \dots, y_{n-1}) \end{cases}$$

Thus, our method can be used to solve ODE of any order.

3.2.2 Testing

When at a small interval, i.e., the h chosen in **Theorem 2** is small, in single differential equation, the mean absolute error (MAE) between the result and actual solution is generally lower than 10^{-6}. The device generally yields high accuracy and efficiency, with the actual

ability to solve (system of) ODEs of finite order. In the form below, certain examples of equations along with its initial value is given at random. And we adopted the single value decomposition technology already in solving the ill-conditioned matrix which will be discussed in the next part.

	(system of) ODEs	Order	Domain of the solution	MAE	Step
1	$\begin{cases}\frac{dy}{dx}=-xy\\ y(0)=1\end{cases}$	First-order	[0,1]	4.0006×10^{-7}	0.1
2	$\begin{cases}\frac{dy}{dx}=-xy\\ y(0)=1\end{cases}$	First-order	[0,5]	$\underline{3.0950\times10^{37}}$	0.1
3	$\begin{cases}\frac{d^2y}{dx^2}=-y\\ y(0)=0\\ y'(0)=1\end{cases}$	Second-order	$[0,2\pi]$	2.9552×10^{-7}	0.1
4	$\begin{cases}\frac{d^2y}{dx^2}=-y+\sin 2x\\ y(0)=0\\ y'(0)=\frac{1}{3}\end{cases}$	Second-order	$[0,2\pi]$	3.2606×10^{-7}	0.1
5	$\begin{cases}\frac{dx}{dy}=2\\ \frac{dy}{dt}=-10t\\ x(0)=0\\ y(0)=10\end{cases}$	First-order	[0,1]	1.2817×10^{-7}	0.1

Table 4 Testing result and error analysis of solutions to (system of) ODEs

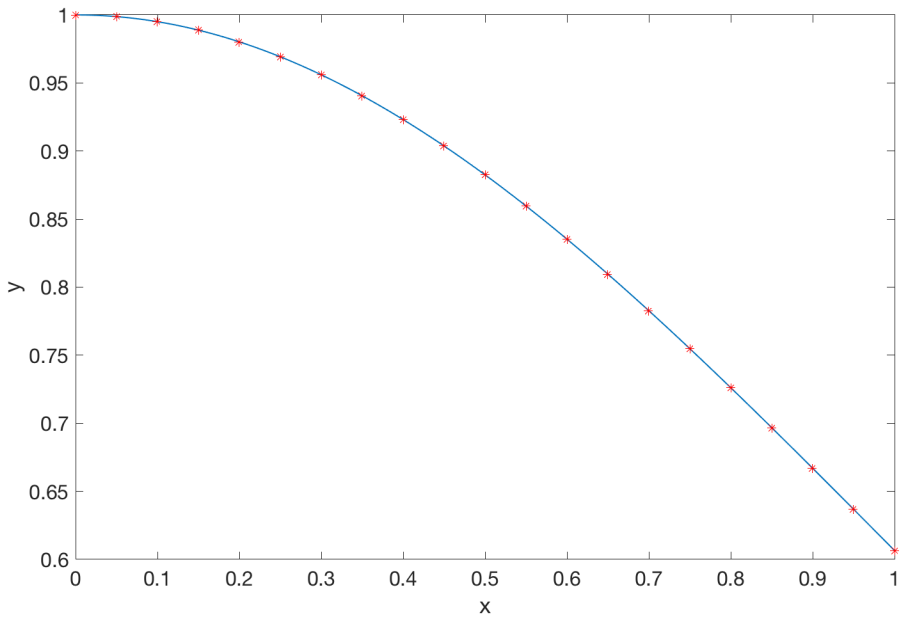


Fig. 4 for case 1

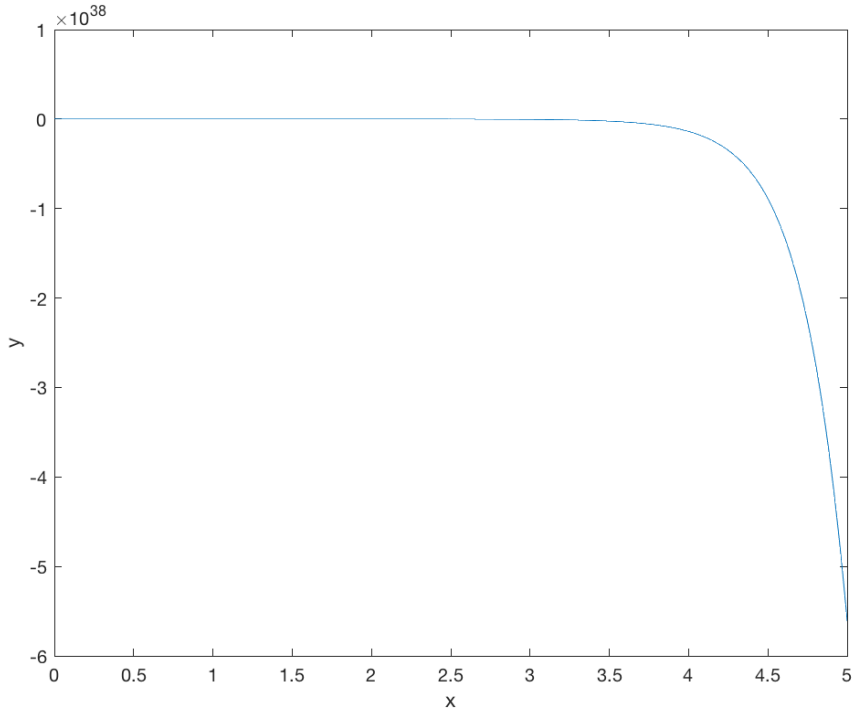


Fig. 5 for case 2

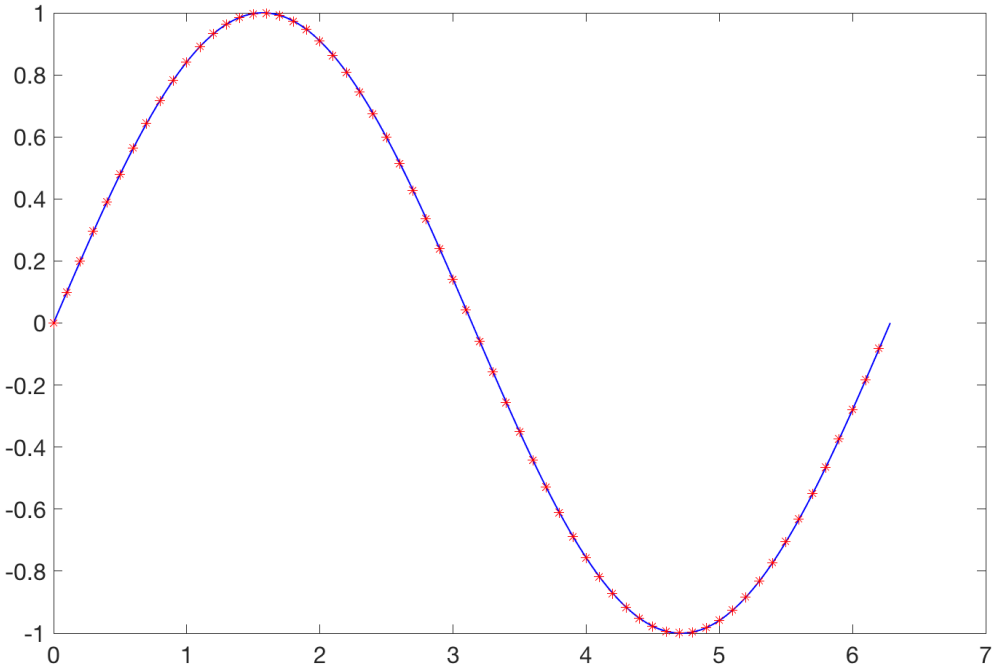


Fig. 6 for case 3

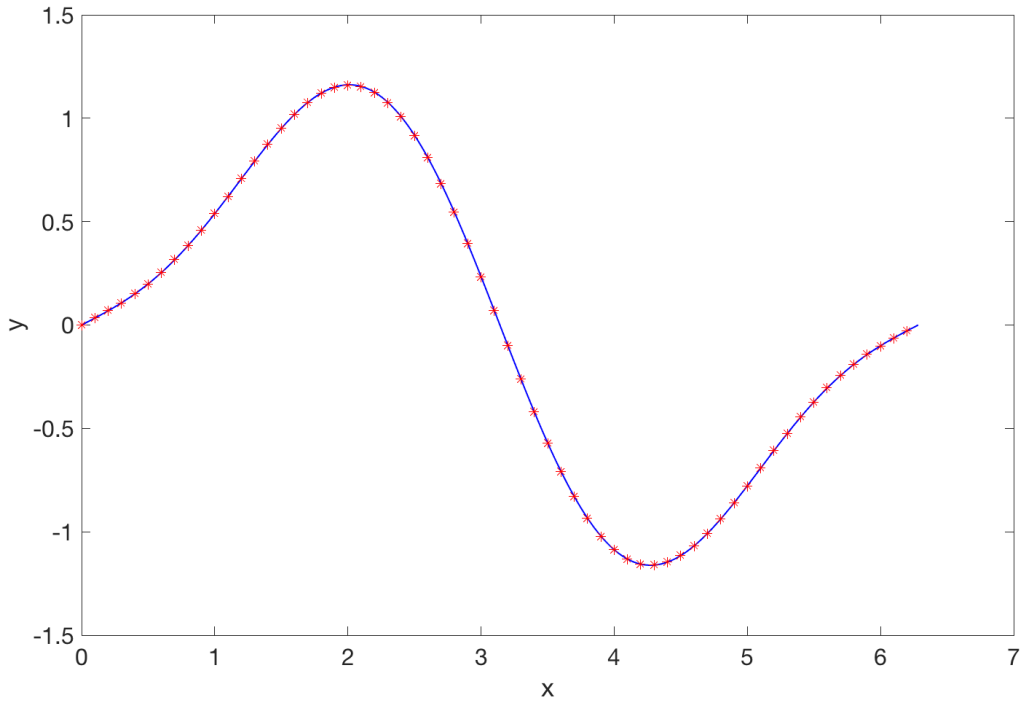


Fig. 7 for case 4

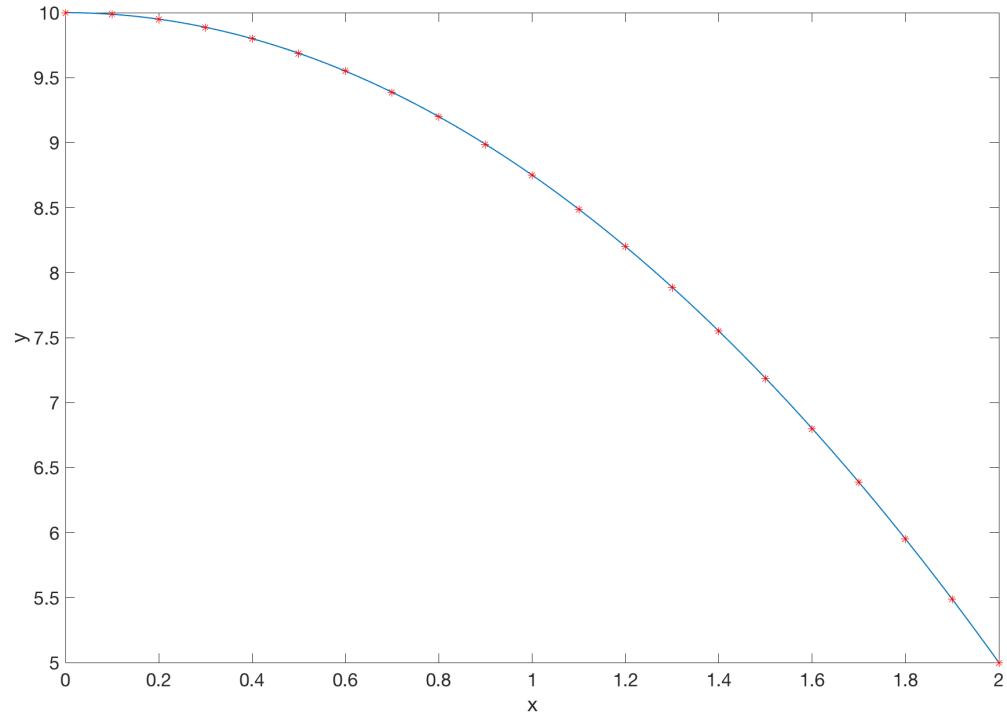


Fig. 8 for case 5

These scatter-point plots are matches of the original equations and the simulated functions. As displayed, the blue curve is the simulated equation, taking 1000 points between the

corresponding intervals. The red points are accurate points fetched from the exact solution of the equation.

For **Fig. 4**,**6**,**7**,**8** the result is generally accurate, with the blue curve yielding no visible deviance from the red scatter-points. For **Fig. 5** the blue curve is far from the accurate results, as to the red scatter-points are displayed out of the plotting area. This phenomenon would be investigated and solved over the following researches.

4 Discussion

4.1 Ill-conditioned matrix

Ill-condition matrix is defined as a matrix with high conditional number. Conditional number (C) is the ratio of the largest singular value to the smallest one in the singular value decomposition of the matrix. A conditional number is said to be too large if $\log(C) \gtrsim$ the precision of matrix entries. If we calculate the conditional number for the interpolation matrix in case 11, we will find out the results in following table:

	Conditional number	Log(C)	Precision of matrix entries
11.1	$8.311776549926087\times10^{18}$	43.564205044499488	10^{-15}
11.2	$1.411988192612196\times10^{19}$	44.094115543750291	10^{-15}
11.3	$8.567733304259643\times10^{20}$	48.199705065588084	10^{-15}
11.4	$2.224135253346766\times10^{21}$	49.153655142767882	10^{-15}

Table 5 Conditional numbers for case 11

As we can clearly see from the table, $\log(C)$ is much larger than the precision of matrix entries. So, all of these interpolation matrices are extremely ill-conditioned, which possibly accounts for the abnormally high error in these four cases. We have used the Gauss Jordan elimination through partial pivoting to avoid the problem of ill-conditioned matrix in solving the linear equation, but it turns out to be non-effective, as the conditional number presented in table 4 is still very high. Therefore, we employed another method to solve this problem.

4.1.1 Single Value Decomposition

Single value decomposition (SVD) is a method that can be used to solve the problem of ill-conditioned matrix.

According to Theorem 2.6.3 by Roger A. Horn (1985), for $A \in \mathbb{R}_{n,n}$, supposing that $\text{rank}(A) = r$, there exists unitary matrix V, W and a square diagonal matrix

$$\Sigma = \begin{bmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_n \end{bmatrix}$$

such that singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0 = \sigma_{r+1} = \cdots = \sigma_n$ and $A = V\Sigma W^*$.

Roger (1985) also offered a method to solve a linear system $Ax = b$, supposing the singular value decomposition $A = V\Sigma W^*$,

$$V\Sigma W^* x = b$$

Then, one of the solutions to the linear system is

$$x = \sum_{i=1}^{r} \frac{{v_i}^* b w_i}{\sigma_i}$$

where v_i, w_i are the ith column of the unitary matrices V, W respectively. Because for all $j > r, Aw_j = V\left(\Sigma W^* w_j\right) = 0$, the arbitrary linear combinations of the last $n - r$ right singular vectors (if they exist) all belong to the null space of A. For arbitrary $\mathrm{c}_{r+1}, \cdots, c_n \in \mathbb{C}$, the vectors

$$x = \sum_{i=1}^{r} \frac{{v_i}^* b}{\sigma_i} w_i + \sum_{i=r+1}^{n} c_i w_i$$

are all solutions to $Ax = b$. Obviously, if $n = r$, the latter sum term will disappear. Because the vector group $\{w_i\}$ is an orthonormal system of vectors, when all $\mathrm{c}_i = 0$, it is a solution of minimum l_2 norm.

Neumaier et al. (1998) concluded that ill-conditioned matrices are characterized by the existence of extremely small singular values of σ_i. Also, the fact that σ_i is in the place of denominator magnifies every tiny change in ${v_i}^* b w_i$, thus rendering the minimum norm least squares solution useless. In order to solve that question, we need to employ the method called

truncated SVD (TSVD) (Hansen 1987), and to replace $(n-m)$ singular values smaller than k, the critical value for truncation, with 0, so that

$$A_m = V\Sigma_m W^*, \Sigma_m = diag(\sigma_1, \cdots, \sigma_m, 0, \cdots, 0) \in \mathbb{R}^{n\times n}$$

where Σ_m equals Σ with the smallest $(n-m)$ singular values replaced by zeros. If k is properly chosen, the conditional number σ_1/σ_m of the resulting A_m will be small. In the case of interpolation matrix, the selection of k is based on the accuracy requirement and ability to solve ill-conditioned matrix problem. If k is larger, the ability to solve ill-conditioned matrix will increase, whereas the accuracy will decrease. *Vice Versa.* When used only as an integral calculator, the conditional number for the interpolation matrix may not be extremely high ($< 10^{20}$), but the accuracy requirement is indeed high. Therefore, we choose a smaller k in most of the cases. When used in solving initial value problem of ODEs, the conditional number is extremely high ($> 10^{40}$), thus requiring a bigger k. Therefore, we chose $k = 10^{-6}$, a relatively large number for the algorithm in our experiments.

4.1.2 Improved results

Based on the above-mentioned theories, we thereby apply it to solve the problem of ill-conditioned interpolation matrix. The results are shown in the table below.

	Selected function	Computed value	True value	True error	Fractional error	Calculating time	Interpolation points	Method
11.4	$\int_3^4 e^{2x}\cos^3 5x\,dx$	3.690625000000000 $\times 10^2$	3.695991815613422 $\times 10^2$	-0.536681561342220	1.45×10^{-3}	2.319108	1000	Gauss Jordan elimination through partial pivoting
11.5	$\int_3^4 e^{2x}\cos^3 5x\,dx$	3.695987377166748 $\times 10^2$	3.695991815613422 $\times 10^2$	$-4.438446674157603 \times 10^{-4}$	1.20×10^{-6}	0.229321	1000	TSVD $k = 10^{-10}$

Table 6 Comparison between previous results and improved results in case 11.4

In case 11.4.2, using TVSD and choosing k as 10^{-10}, we successfully reduced the fractional error to 1.20×10^{-6}, 10^{-3} smaller than the original one, which proves the validity of the TSVD method.

4.2 Joint Method

There are certain cases in which the MAE would be extremely large (as shown in **Table 4** case 2). We analyzed many similar cases and found that this phenomenon is caused by two main factors:

1) The local existence of the solution is not satisfied by a wide given domain i.e. the chosen of h as mentioned in **Theorem 2** is too large.
2) The step chosen is too large such that the integral calculator does not reach a satisfactory quality.

A solution to both problems is called *joint method*, by which interpolation points will be adequate for a certain value of h. The mechanism for this method is that

1) we choose a small h and solve the (system of) ODEs on the interval $[x_0, x_0 + h]$;
2) then, we solve the same (system of) ODEs on the interval $[x_0 + h, x_0 + 2h]$ with the initial value $\boldsymbol{y}(x_0 + h)$ obtained by the final output of the previous solution;
3) by repeating this process, a wide range solution with very small error can be thus obtained.

The following table shows that the problem can be solved perfectly by employing the *joint method*.

	(system of) ODEs	Order	Domain of the solution	MAE	Joint Method
2.1	$\begin{cases} \frac{dy}{dx} = -xy \\ y(0) = 1 \end{cases}$	First-order	[0,5]	2.7567×10^{-7}	Yes h=0.5 step=0.05
2.2	$\begin{cases} \frac{dy}{dx} = -xy \\ y(0) = 1 \end{cases}$	First-order	[0,5]	3.0950×10^{37}	No h=5 step=0.1

Table 7 Comparison between previous results and improved result

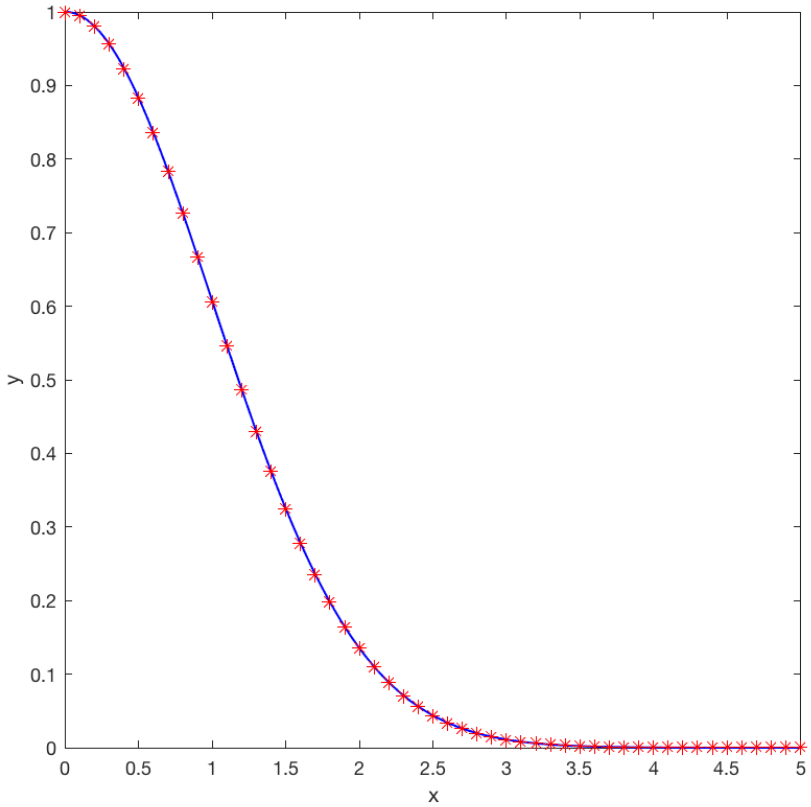


Fig. 9 Improved result by join method of **table 7** case 2.1

As shown in the figure above, the solution is very accurate after the treatment of *joint method*. Hence, *joint method* can be considered as a global solution algorithm. In most of our experiments we choose $h = 0.5$ and $step = 0.05$. Yet as the input function varies to custom

(system of) ODEs, users might need to manually adjust the h and $step$ in `solglob.m` in Appendix.2 so as to obtain the optimal solution.

There are also certain equations, such as the function

$$\begin{cases} \frac{dy}{dx} = \sin xy \\ y(0) = 1 \end{cases} \dots (5)$$

In this case, extra adjustments would be required in shortening the h and $step$ taken as to give an accurate result. We do two experiments on (5) to present the problem.

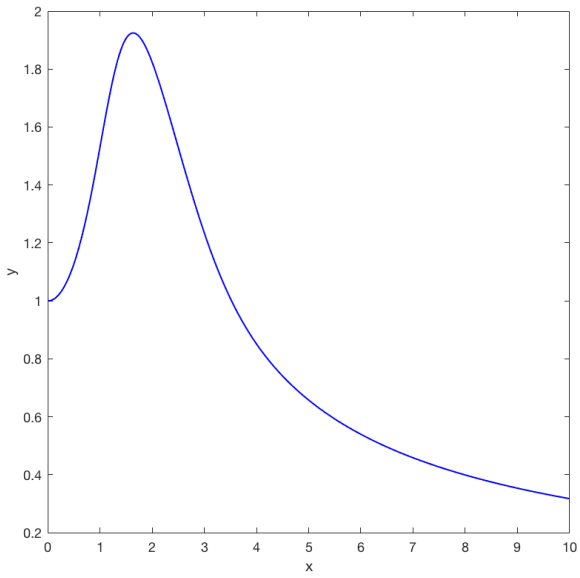


Fig. 10 Figure for ODE (5) with domain $[0,10]$ and normal step

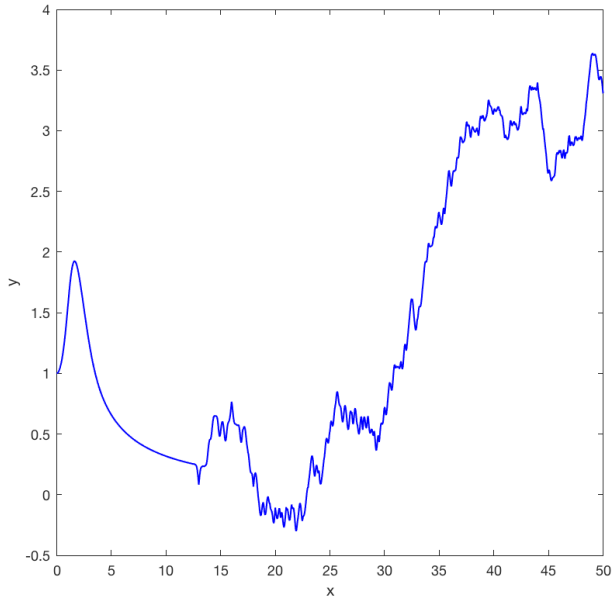


Fig. 11 Figure for ODE (5) with domain [0,50] and normal step

As shown in **Fig. 10** and **11**, this solution becomes inaccurate when $x > 10$. We must choose shorter steps in order to overcome the problem. Now, with $h = 0.1$ and $step = 0.01$, we perform another solution.

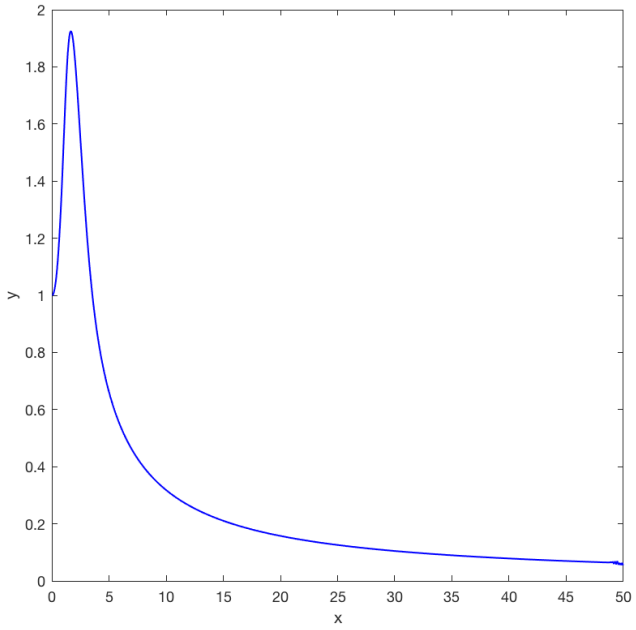


Fig. 12 Figure for ODE (5) with domain [0,50] and small step

As shown in **Fig. 12**, we obtained a much better solution. This phenomenon is caused by the frequent change of sign of $\boldsymbol{f}(x, \boldsymbol{y})$ in **Theorem 2** when x is large. Hence, more interpolation points are needed for a given interval, i.e., smaller steps are needed.

4.3 Performance in simulation

The device above is put into simulation, used in situation of single pendulum. Engaged in two different situations, the device gave stable results over repeats of the experiment.

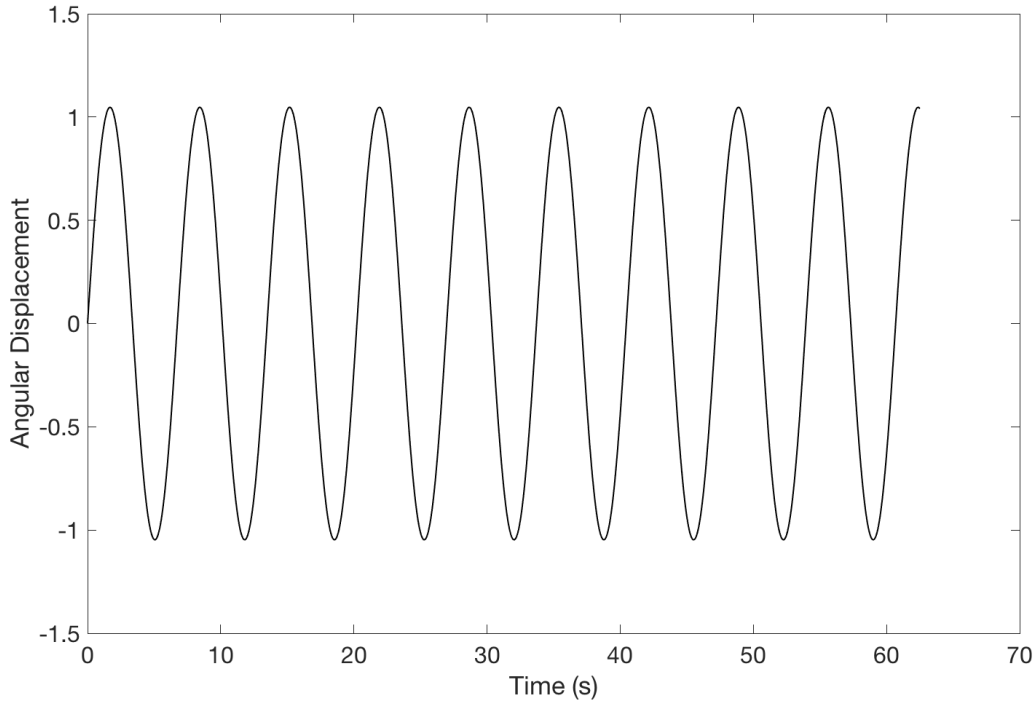


Fig. 13 the $\theta - t$ graph for single pendulum at large angle without damping

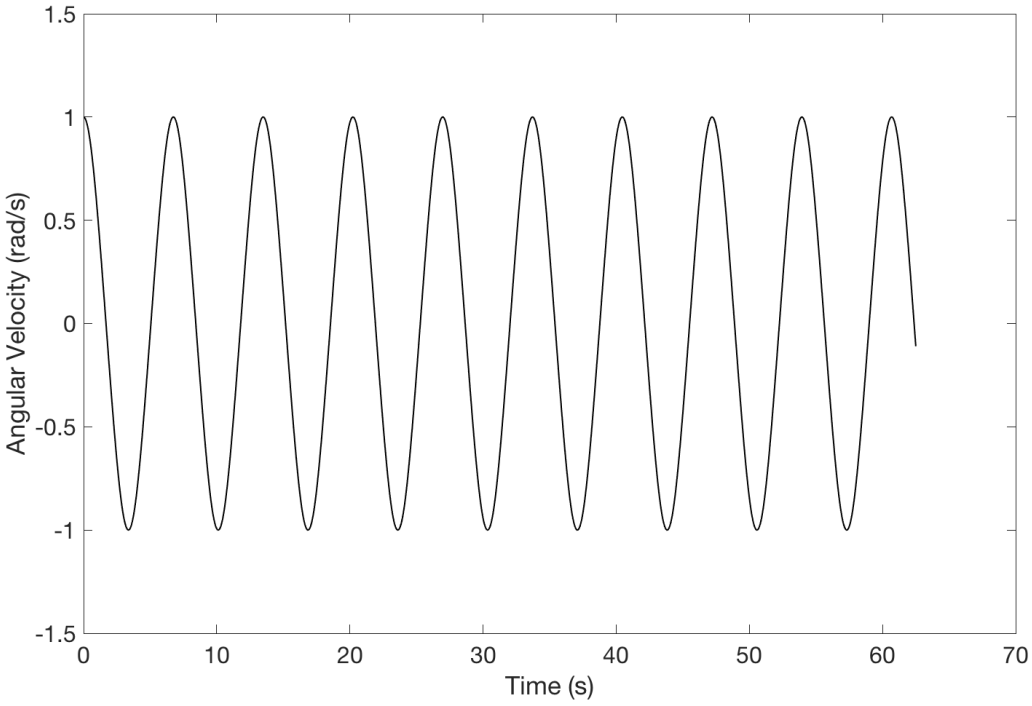


Fig. 14 the $\omega - t$ graph for single pendulum at large angle without damping

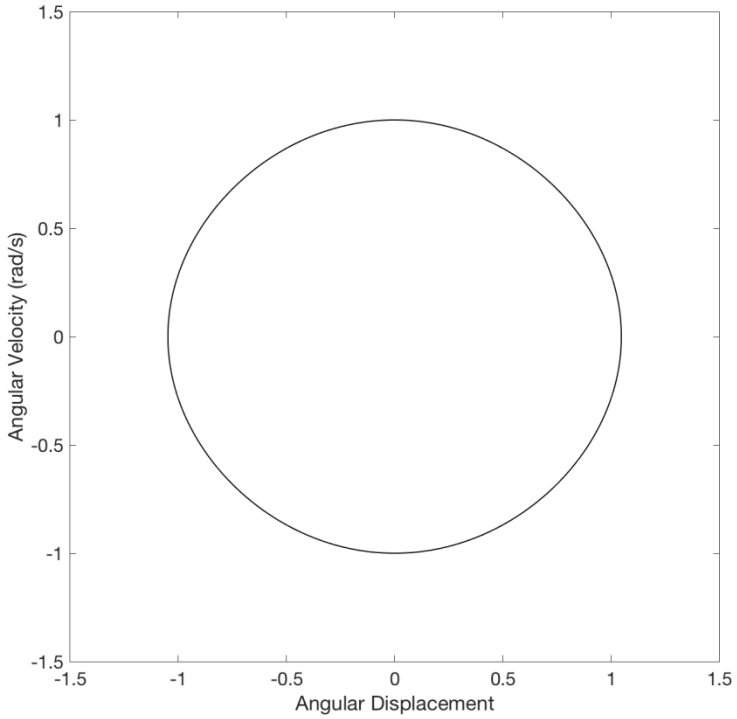


Fig. 15 the $\omega - \theta$ graph for single pendulum at large angle without damping

Within the simulations (**Fig. 13**,**14**,**15**) of equation for single pendulum at large angle without damping ($\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin\theta = 0$), for distinct periods there is no visible divergence from one another. The coherence of the test result is further demonstrated in the phase diagram (**Fig. 15**). Over 20 complete periods of oscillations, the graph forms an ellipse that yields no visible deviation. This phenomenon strictly conforms with the Conservation of Energy, which further indicates accuracy of the device.

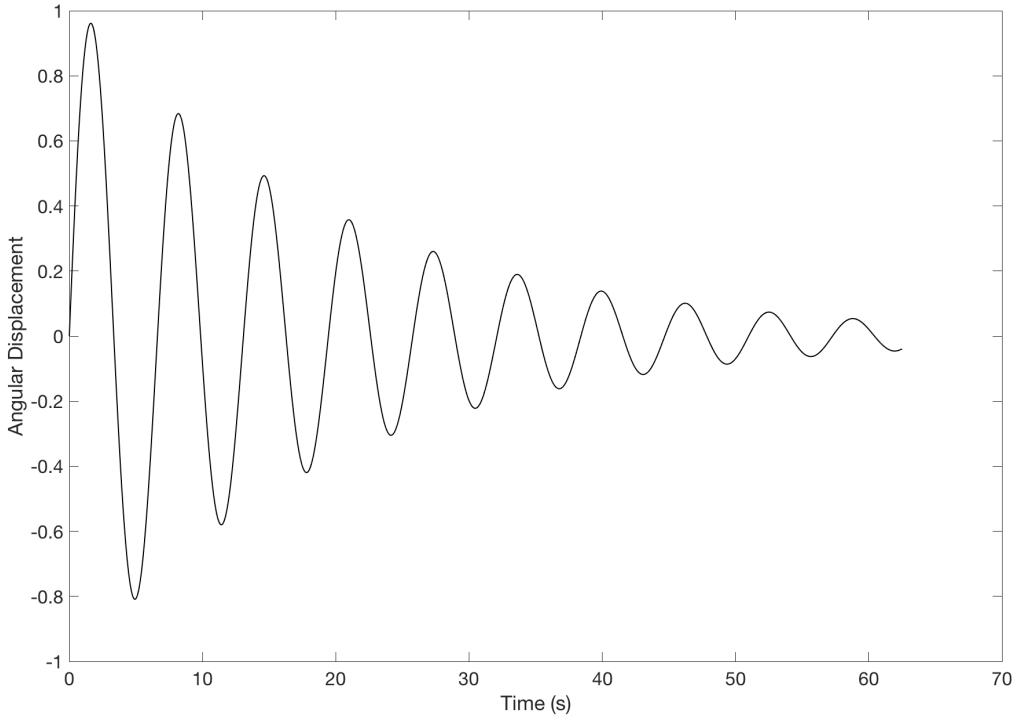


Fig. 16 the $\theta - t$ graph for single pendulum at large angle with damping

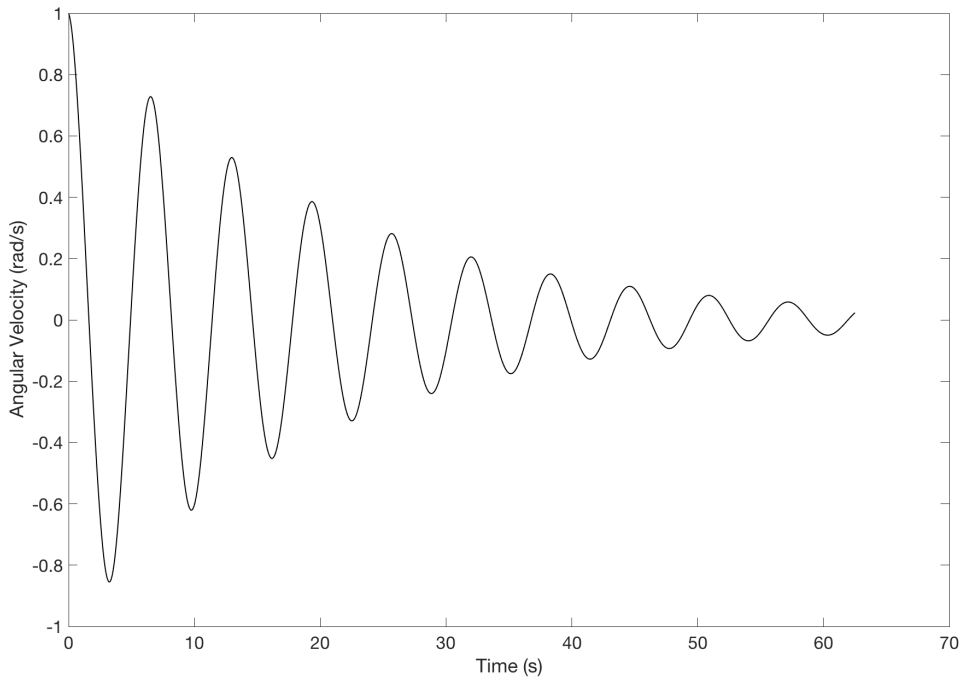


Fig. 17 the $\omega - t$ graph for single pendulum at large angle with damping

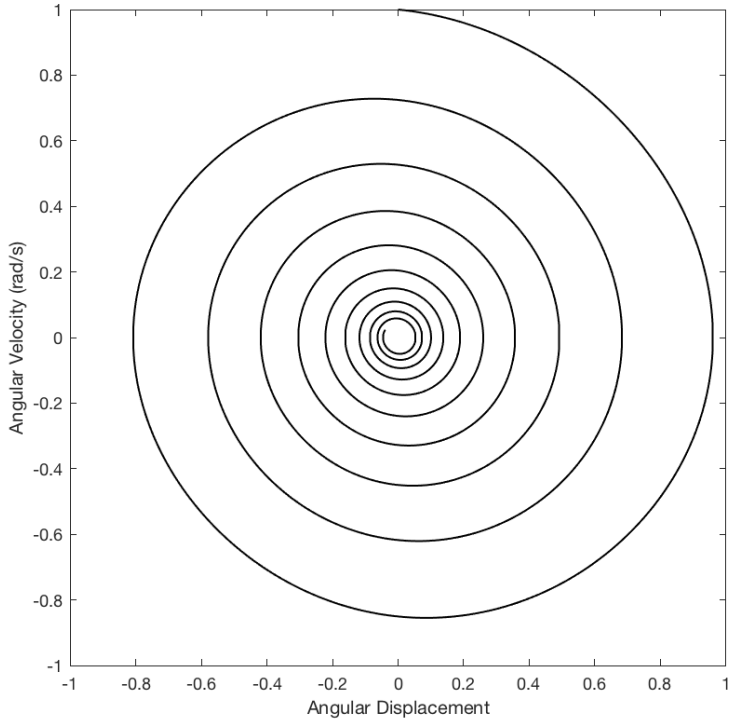


Fig. 18 the $\omega - \theta$ graph for single pendulum at large angle with damping

Within the simulations (**Fig. 16**,**17**,**18**) of equation for single pendulum at large angle with damping ($\frac{d^2\theta}{dt^2} + 2\beta\frac{d\theta}{dt} + \frac{g}{l}\sin\theta = 0$), along with the increase in time, both maximum displacement and velocity in distinct periods gradually converges to 0. The phrase diagram displays a spiral that collects towards the center. This indicates the loss of energy within the processes of vibration.

5 Conclusion

The algorithm acquired to solve (system of) ODEs is proved to be robust in most cases. It can also be put into the approximate solutions for various experiments in mechanics, three body problem, etc. In some cases, we still need to adjust step to acquire a more accurate result. However, how to adjust to an optimal step is still a question needed to be researched in the future.

As the byproduct, the RBFN-based integral calculator is proved to be both accurate and fast in all of the cases tested. Because the limitation to the input function is very easy to be satisfied, it can even be used to calculate integral of functions that cannot be obtained from all existing methods.

Thus, the two results of this work have both shown the potential to be put into real application, helping solve various problems in math, physics, and other science fields.

Bibliography

Comenetz, Michael. Calculus: the elements. World Scientific Publishing Co Inc, 2002.

Dougherty, Mark. "A review of neural networks applied to transport." Transportation Research Part C: Emerging Technologies 3, no. 4 (1995): 247-260.

Feller, William. An introduction to probability theory and its applications: volume I. Vol. 3. New York: John Wiley & Sons, 1968.

Hansen, Per Christian. "The truncatedsvd as a method for regularization." BIT Numerical Mathematics 27, no. 4 (1987): 534-553.

Haykin, Simon S. 2009. Neural Networks And Learning Machines. Upper Saddle River: Pearson Education.

Kalogirou, Soteris A. "Artificial neural networks in renewable energy systems applications: a review." Renewable and sustainable energy reviews 5, no. 4 (2001): 373-401.

Maier, Holger R., and Graeme C. Dandy. "Neural networks for the prediction and forecasting of water resources variables: a review of modelling issues and applications." Environmental modelling & software 15, no. 1 (2000): 101-124.

Neumaier, Arnold. "Solving ill-conditioned and singular linear systems: A tutorial on regularization." SIAM review 40, no. 3 (1998): 636-666.

Powell, Michael JD. "The theory of radial basis function approximation." Advances in numerical analysis 2 (1992): 105-210.

丁, 同仁, and 承治 李. 1991. *常微分方程教程*. 2nd ed. 北京: 高等教育出版社.

(Ding, Tongren and Li, Chengzhi. 1991. A Course of Ordinary Differential Equation. 2nd ed. Beijing: Higher Education Press.)

Appendix.1 MATLAB code for integral calculator

f.m
```
function y=f(x)
y=exp(2.*x).*cos(5*x).^3;
```

fp.m
```
function y=fp(x0,x,w,c)
%x0 is the centres in the form of column
%x is the input variable
%w is the trained weights in the form of column
%c is the parameter for the integral of multiquadrics
%y is the estimated output value for the primitive function
x1=x-x0;
y1=intmulqua(x1,c);
y=w'*y1;
```

fp1.m
```
function y=fp1(x0,x,w,c)
%x0 is the centres in the form of column
%x is the input variable
%w is the trained weights in the form of column
%c is the parameter for the multiquadrics
%y is the output estimated value for the 1st derivative
x1=x-x0;
y1=mulqua(x1,c);
y=w'*y1;
```

int1.m
```
function y=int1(w,x,a,b)
%a in the lower bound
%b is the upper bound
%w is the trained weights in the form of column
y=fp(x,b,w,1)-fp(x,a,w,1);
```

intmulqua.m
```
function y=intmulqua(x,c)
y=0.5.*x.*(x.^2+c^2).^0.5+0.5*c^2.*log(x+(x.^2+c^2).^0.5);
```

mulqua.m
```
function y=mulqua(x,c)
y=(x.^2+c^2).^0.5;
```

pca.m
```
function a=pca(a,i)
%a input is the matrix of n by n+1
%a output is the matrix after the application of pca for the ith column
[m,~]=size(a);
b=a(i:m,i:m+1);
v=find(abs(b(:,1))==max(abs(b(:,1))),1);
c=b(v,:);
b(v,:)=b(1,:);
b(1,:)=c./c(1,1);
for j=2:(m-i+1)
    b(j,:)=b(j,:)-b(j,1).*b(1,:);
end
a(i:m,i:m+1)=b;
```

pcagauss.m
```
function c=pcagauss(a,b)
%a is a n by n matrix
%b is a n by 1 matrix
%c is the output n by 1 matrix which is inv(a)*b
a=[a b];
[m,~]=size(a);
for i=1:m-1
    a=pca(a,i);
end
a(m,m:m+1)=a(m,m:m+1)./a(m,m);
b=a(:,m+1);
for i=m:-1:2
    for j=1:i-1
        b(j)=b(j)-a(j,i)*b(i);
    end
end
c=b;
```

rbf.m
```
function [w,ker]=rbf(x,d,c)
%x is the samples for training in the form of column
```

```
%d is the expected outputs int the for of column
%c is the parameter for the multiquadrics
%w is the weights in the form of row
n=length(x);
ker=zeros(n,n);
for i=1:n
    x1=x(i)-x;
    ker(:,i)=mulqua(x1,c);
end
%w=pcagauss(ker,d);
%w=ker\d;
w=solill(ker,d);
```

rbfint.m

```
function [w,x,ker]=rbfint(a,b)
%a in the lower bound
%b is the upper bound
%w is the trained weights in the form of column
%c is the parameter for the multiquadrics
c=1;
h=(b-a)/999;
x=(a:h:b)';
d=f(x);
[w,ker]=rbf(x,d,c);
```

solill.m

```
function x=solill(a,b)
[u,s,v]=svd(a);
s=diag(s);
k=find(s<1e-10,1);
x=0;
if isempty(k)
   k=length(s)+1;
end
if k==1
   k=2;
end
for i=1:k-1
    x=x+u(:,i)'*b*v(:,i)./s(i);
end
```

Appendix.2 MATLAB code for differential equations

f.m
```
function z=f(x,y)
z(:,1)=-x.*y;
```

fp.m
```
%x0 is the centres in the form of column
%x is the input variable
%w is the trained weights in the form of column
%y is the output estimated value for the primative function
function y=fp(x0,x,w)
x1=x-x0;
y1=intmulqua(x1);
y=w'*y1;
```

fp1.m
```
function y=fp1(x0,x,w)
%x0 is the centres in the form of column
%x is the input variable
%w is the trained weights in the form of column
%y is the output estimated value for the 1st derivative
x1=x-x0;
y1=mulqua(x1);
y=w'*y1;
```

int.m
```
function y=int1(w,x,a,b)
%a in the lower bound
%b is the upper bound
%w is the trained weights in the form of column
y=fp(x,b,w)-fp(x,a,w);
```

intmulqua.m
```
function y=intmulqua(x)
v=(x.^2+1).^0.5;
y=0.5.*(x.*v+log(x+v));
```

mulqua.m
```
function y=mulqua(x)
y=(x.^2+1).^0.5;
```

rbf.m
```
function w=rbf(x,d)
%x is the samples for training in the form of column
%d is the expected outputs int the for of column
%w is the weights in the form of row
n=length(x);
ker=zeros(n,n);
for i=1:n
    x1=x(i)-x;
    ker(:,i)=mulqua(x1);
end
w=solill(ker,d);
```

rebfint.m
```
function [w,x]=rbfint(x,y)
%a in the lower bound
%b is the upper bound
%w is the trained weights in the form of column
z=f(x,y);
[m,n]=size(z);
w=zeros(m,n);
for i=1:n
    w(:,i)=rbf(x,z(:,i));
end
```

sol.m
```
function [y1,x1,w,x,ma,yf]=sol(a,b,d0,step)
x=(a:step:b)';
n=length(x);
m=length(d0);
y1=zeros(n,m);
for i=1:m
    y1(:,i)=d0(i);
end
```

```
for j=1:1000
      [w,x]=rbfint(x,y1);
      for i=1:n
          for k=1:m
              y1(i,k)=int1(w(:,k),x,a,x(i));
          end
      end
      for k=1:m
          y1(:,k)=y1(:,k)+d0(k);
      end
end
ma=max(abs(w(:)));
h=(b-a)/1000;
x1=(a:h:b)';
l=length(x1);
y1=zeros(l,m);
for i=1:l
    for k=1:m
        y1(i,k)=int1(w(:,k),x,a,x1(i));
    end
end
for k=1:m
    y1(:,k)=y1(:,k)+d0(k);
end
yf=y1(l,:);
```

solglob.m

```
function [y,x,w,x2,yf]=solglob(a,b,d0)
t=a:0.5:b;
n=length(t);
yf=d0;
y=cell(1,n-1);
x=y;
x2=y;
w=y;
step=0.05;
for i=1:n-1
    [y{i},x{i},w{i},x2{i},~,yf]=sol(t(i),t(i+1),yf,step);
end
```

solill.m

```
function x=solill(a,b)
[u,s,v]=svd(a);
s=diag(s);
k=find(s<1e-6,1);
x=0;
if isempty(k)
   k=length(s)+1;
end
if k==1
   k=2;
end
for i=1:k-1
    x=x+u(:,i)'*b*v(:,i)./s(i);
end
```