

NUMERICALLY SOLVING ORDINARY DIFFERENTIAL EQUATIONS



Stuart Brorson
Northeastern University

Numerically Solving Ordinary Differential Equations (Brorson)

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (<https://LibreTexts.org>) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by [NICE CXone Expert](#) and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptations contact info@LibreTexts.org or visit our main website at <https://LibreTexts.org>.

This text was compiled on 03/26/2026

TABLE OF CONTENTS

Licensing

1: Introduction

- 1.1: Solving ordinary differential equations
- 1.2: Sampled functions
- 1.3: Differentiation on the computer
- 1.4: Chapter summary

2: Forward Euler method

- 2.1: Forward Euler algorithm
- 2.2: Example - the exponential growth ODE
- 2.3: Local truncation error vs. global solution error
- 2.4: Example - a driven system
- 2.5: Example - the logistic equation
- 2.6: Extending Forward Euler to higher order
- 2.7: Example - the simple harmonic oscillator
- 2.8: Propagator matrix for SHO
- 2.9: Stability
- 2.10: Chapter summary

3: Backward Euler method

- 3.1: Backward Euler algorithm
- 3.2: Example - exponential growth ODE
- 3.3: Example - the logistic equation
- 3.4: Example - the simple harmonic oscillator
- 3.5: Chapter summary

4: Predictor-corrector methods and Runge-Kutta

- 4.1: Heun's method
- 4.2: Example - the exponential growth ODE and error scaling
- 4.3: Example - the simple harmonic oscillator
- 4.4: Midpoint method
- 4.5: Example - the exponential growth ODE
- 4.6: Runge-Kutta methods
- 4.7: Example - the exponential growth ODE
- 4.8: Example - the simple harmonic oscillator
- 4.9: Example - the hanging pendulum
- 4.10: Example - the Van der Pol oscillator
- 4.11: Example - Duffing's equation
- 4.12: Butcher tableaux
- 4.13: Chapter summary

5: Adaptive stepping

- 5.1: Choosing h
- 5.2: Example - the logistic equation
- 5.3: Stiff ODEs
- 5.4: Chapter summary

6: Linear multistep methods

- 6.1: Trapezoidal method
- 6.2: Example - the exponential growth ODE
- 6.3: Adams-Bashforth method – 2nd order
- 6.4: Adams-Bashforth method – 3rd order
- 6.5: Adams-Moulton method – 3rd order
- 6.6: Example - Adams methods and the Van der Pol oscillator
- 6.7: Generalizations
- 6.8: Chapter summary

7: Symplectic integrators

- 7.1: Energy conserving ODEs
- 7.2: Symplectic Euler
- 7.3: Example- the simple harmonic oscillator
- 7.4: Conservative systems, phase space, and Liouville's theorem
- 7.5: Euler methods and conservation of phase space volumes
- 7.6: Chapter summary

8: Review and summary

- 8.1: Review and summary
- 8.2: Guide to Matlab codes

[Index](#)

[Glossary](#)

[Detailed Licensing](#)

Licensing

A detailed breakdown of this resource's licensing can be found in [Back Matter/Detailed Licensing](#).

CHAPTER OVERVIEW

1: Introduction

This chapter covers solving first-order ordinary differential equations (ODEs), focusing on initial value problems (IVP) where unique solutions depend on initial conditions. It explains sampled functions for representing continuous data as discrete points, which is essential for numerical methods.

[1.1: Solving ordinary differential equations](#)

[1.2: Sampled functions](#)

[1.3: Differentiation on the computer](#)

[1.4: Chapter summary](#)

Solving ordinary differential equations

Solving ordinary differential equations (ODEs) is a major focus of numerical computing. We start by considering first-order ODEs of the form.

$$y' = f(t, y) \quad (1.1)$$

Where $y(t)$ is the unknown (i.e. the thing we want to find), and t is the independent variable. When dealing with such an ODE, we usually think of t as time, and y as something depending upon time, like the population of an animal species or the voltage of a node in an electronic circuit. Equation 1.1 says that the way y changes in time depends upon some function f . In this booklet, we assume the function f is smooth and well-behaved, and that it is easily calculable in a computer program.

One may write down solutions to 1.1, but to get a *unique* solution an additional piece of information is required: An initial condition. That is, we need to know something about $y(t)$ at one point in time so we can select one individual solution from all possible solutions which satisfy 1.1. One traditionally uses a value for $y(t)$ at time $t = 0$ as the initial condition, but other choices are also possible (but uncommon).

Here's a very basic example. Consider the ODE

$$y' = ay \quad (1.2)$$

You likely encountered this ODE early in your mathematical career, and you know the solution by heart:

$$y(t) = Ce^{at} \quad (1.3)$$

where C is some constant. The solution 1.3 obviously satisfies equation 1.2, but it is not useful in practice as the solution to the equation. Why? The reason is that there are an infinite number of possible solutions depending upon the value of C chosen. That is, 1.3 defines a family of solutions, but in a real, practical problem we only want one of those solutions. For example, how would you make a plot of 1.3? The answer is, you can't, until you chose a specific value of C .

In a practical problem, you usually know the value of your function at a start-point in time, almost always at $t = 0$. Call that value y_0 . In this case, the specific solution which satisfies 1.2 and also satisfies $y(t = 0) = y_0$ is

$$y(t) = y_0 e^{at} \quad (1.4)$$

The point is that numerically solving an ODE requires two pieces of information: 1. The ODE itself, and 2. An initial condition. The initial value picks one solution out of the entire family of solutions which satisfy 1.2. The concept is shown in 1.1

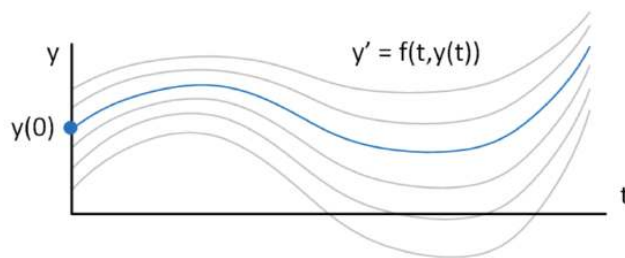


Figure 1.1: The ODE $y' = f(t, y)$ is satisfied by a family of curves. An initial condition $y(0)$ is required to pick out one of the solutions for numerical simulation.

Such a problem is called an "initial value problem" (IVP) to distinguish it from other types of differential equations (which we will encounter later in the class). "Initial value problem" simply means we know the beginning (initial) value of y , and want to find its behavior as we move forward in time. Once the equation 1.1 and an initial condition for y are specified, then we have all the information required to find a unique solution $y(t)$ numerically.

Methods to find numerical solutions IVPs are the focus of this booklet. As a matter of nomenclature, we generally say "solve the ODE", or "solve the IVP". This means we compute the function $y(t)$ starting from time $t = 0$ to some point in the future, t_{end} . Another way to say the same thing is we "integrate the ODE", or "integrate the IVP". When speaking of ODEs, "integrate" means the same thing as "solve".

Sampled functions

In mathematics we become familiar with the notion of a function. Consider the simplest case of a function $y(t)$ which takes a real scalar t and returns a real scalar value $y(t)$. This is frequently written formally as $y : \mathbb{R} \rightarrow \mathbb{R}$. This means, take any arbitrary real number t , plug it into y , and out will come a new real number $y(t)$. The important idea to glean from this is that the function is like a machine into which one can input any t and expect to get an output. That is, the function is defined on every real number t .

Unfortunately, when working on real data using a computer this nice abstraction doesn't always hold. Rather, it is very common for us to have only certain, discrete points t_n where the function is defined. Digital audio is a simple example: the sounds we hear with our ear are related to the sound pressure waves moving our eardrums. The sound pressure has a value for every point in time. But when processing audio using a computer the sound pressure is sampled at regular, discrete times and turned into numbers representing the sound pressure at each sample time. This process is often referred to as "discretization", meaning that the continuously-defined $y(t)$ is replaced by a set of discrete samples y_n . This is depicted in 1.2, which shows the relationship of a continuous-time function and its discrete-time (discretized) replacement.

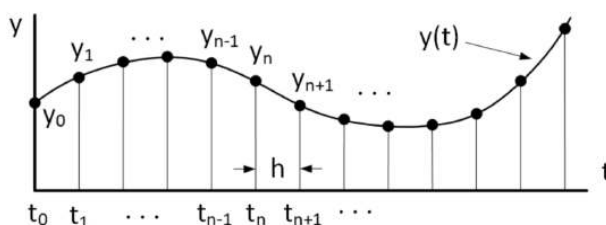


Figure 1.2: A sampled function. The continuous-time version of the function is $y(t)$. Sampling replaces $y(t)$ with samples taken at discrete times, t_n . The samples y_n may be handled by the computer as a vector or list of values. In the case shown here the function is sampled at evenly-spaced intervals separated by h . Evenly-spaced samples are the most frequent case – for example, digital audio commonly samples an input sound signal with a period h of about 22.7usec (44.1kHz sampling rate).

It turns out that the ODE solvers we will study all work with sampled functions. That is, the solvers compute the solution $y(t)$ at a sequence of discrete time values t_n similar to the situation shown in 1.2. The output of the solver will be a vector of discrete values y_n representing samples of the actual, underlying continuous function $y(t)$.

Here is an implementation hint when you write programs processing sampled functions: The sampled function itself is generally represented by a vector, y_n . This is the object your algorithm will use during its work. On top of y_n I recommend also carrying around a vector representing the sample times, t_n , in your program. Having your sample times readily accessible can help decrease

confusion when you want to plot y_n vs. time, for example. For moderately sized vectors, the extra memory required to hold t_n is a small price to pay to keep confusion at a minimum while writing your code.

Differentiation on the computer

Next before diving into solving 1.1 we need to review how differentiation is done on the computer. To start, recall the definition of the derivative of $y(t)$ which you probably learned in high school:

$$\frac{dy}{dt} = \lim_{\Delta t \rightarrow 0} \frac{y(t + \Delta t) - y(t)}{\Delta t} \quad (1.5)$$

In doing numerical computation, we can't represent the limiting transition $\lim_{\Delta t \rightarrow 0}$ – our data is usually discrete (e.g. a sampled function). More importantly, floating point numbers don't smoothly transition to zero – there is no "infinitesimally small but nonzero" number representable on a computer. Instead, we use the definition 1.5 to form an approximation to the derivative as

$$\frac{dy}{dt} \approx \frac{y(t + h) - y(t)}{h} \quad (1.6)$$

where h is a small number. This is called the "forward difference" approximation of the derivative.

First derivatives from Taylor's series

It's easy to write down the approximation 1.6 by remembering the high school definition of a derivative. However, we gain some appreciation for the accuracy of approximating derivatives by deriving them using Taylor series expansions. Suppose we know the value of a function y at the time t and want to know its value at a later time $t + h$. How to find $y(t + h)$? The Taylor's series expansion says we can find the value at the later time by

$$y(t + h) = y(t) + h \left. \frac{dy}{dt} \right|_t + \frac{h^2}{2} \left. \frac{d^2y}{dt^2} \right|_t + \frac{h^3}{6} \left. \frac{d^3y}{dt^3} \right|_t + O(h^4) \quad (1.7)$$

The notation $O(h^4)$ means that there are terms of power h^4 and above in the sum, but we will consider them details which we can ignore.

We can rearrange 1.7 to put the first derivative on the LHS, yielding

$$\left. \frac{dy}{dt} \right|_t = \frac{y(t + h) - y(t)}{h} - \frac{h}{2} \left. \frac{d^2y}{dt^2} \right|_t - \frac{h^2}{6} \left. \frac{d^3y}{dt^3} \right|_t - O(h^4)$$

Now we make the assumption that h is very small, so we can throw away terms of order h and above from this expression. With that assumption we get

$$y(t + h) \approx y(t) + h \left. \frac{dy}{dt} \right|_t$$

which can be rearranged to give

$$\frac{dy}{dt} \approx \frac{y(t + h) - y(t)}{h} \quad (1.8)$$

i.e. the same forward difference expression as found in 1.6 earlier. However, note that to get this approximation we threw away all terms of order h and above. This implies that when we use this approximation we incur an error! That error will be on the order of h . In the limit $h \rightarrow 0$ the error disappears but since computers can only use finite h , there will always be an error in our computations, and the error is proportional to h . We express this concept by saying the error scales as $O(h)$.

Using Taylor's series expansions we can discover other results. Consider sitting at the time point t and asking for the value of y at time $t - h$ in the past. In this case, we can write an expression for the past value of y as

$$y(t - h) = y(t) - h \left. \frac{dy}{dt} \right|_t + \frac{h^2}{2} \left. \frac{d^2y}{dt^2} \right|_t - \frac{h^3}{6} \left. \frac{d^3y}{dt^3} \right|_t + O(h^4) \quad (1.9)$$

Note the negative signs on the odd-order terms. Similar to the forward difference expression above, it's easy to rearrange this expression and then throw away terms of $O(h)$ and above to get a different approximation for the derivative,

$$\frac{dy}{dt} \approx \frac{y(t) - y(t-h)}{h} \quad (1.10)$$

This is called the "backward difference" approximation to the derivative. Since we threw away terms of $O(h)$ and above, this expression for the derivative also carries an error penalty of $O(h)$.

Now consider forming difference between Taylor series 1.7 and 1.9. When we subtract one from the other we get

$$y(t+h) - y(t-h) = 2h \left. \frac{dy}{dt} \right|_t + 2 \frac{h^3}{6} \left. \frac{d^3y}{dt^3} \right|_t + O(h^5)$$

This expression may be rearranged to become

$$\left. \frac{dy}{dt} \right|_t = \frac{y(t+h) - y(t-h)}{2h} - \frac{h^2}{6} \left. \frac{d^3y}{dt^3} \right|_t + O(h^4)$$

Now if we throw away terms of $O(h^2)$ and above, we get the so-called "central difference" approximation for the derivative,

$$\left. \frac{dy}{dt} \right|_t \approx \frac{y(t+h) - y(t-h)}{2h} \quad (1.11)$$

Note that to get this expression we discarded $O(h^2)$ terms. This implies that the error incurred upon using this expression for a derivative is $O(h^2)$. Since we usually have $h \ll 1$, the central difference error $O(h^2)$ is smaller than the $O(h)$ incurred when using either the forward or backward difference formulas. That is, the central difference expression 1.11 provides a more accurate approximation to the derivative than either the forward or the backward difference formulas, and should be your preferred derivative approximation when you can use it.

In each of these derivations we threw away terms of some order p (and above). Discarding terms of order p and above means we no longer have a complete Taylor series. This operation is called "truncation", meaning we have truncated (chopped off) the remaining terms of the full Taylor series. The error incurred by this operation is therefore called "truncation error".

Second derivatives from Taylor's series

We can also get an approximation for the second derivative from expressions 1.7 and 1.9. This time, form the sum of the two expressions to get

$$y(t+h) + y(t-h) = 2y(t) + 2 \frac{h^2}{2} \left. \frac{d^2y}{dt^2} \right|_t + O(h^4)$$

This may be rearranged to get an expression for the second derivative,

$$\left. \frac{d^2y}{dt^2} \right|_t = \frac{y(t+h) - 2y(t) + y(t-h)}{h^2} - O(h^2)$$

If we drop the $O(h^2)$ term, we get the approximation to the second derivative,

$$\left. \frac{d^2y}{dt^2} \right|_t \approx \frac{y(t+h) - 2y(t) + y(t-h)}{h^2} \quad (1.12)$$

And since we dropped the $O(h^2)$ term to make this approximation, it means that use of 1.12 will incur an error of $O(h^2)$ when using this approximation.

Chapter summary

Here are the important points made in this chapter:

- Solving an ODE numerically needs two pieces of information: The ODE and an initial condition. Such problems are called "initial value problems", or IVPs.
- As mathematicians, we are used to functions $y(t)$ which accept any continuous value of t and emit a value. However, when dealing with computers, we must frequently use "sampled functions" in which the continuous $y(t)$ is replaced with samples y_n taken at regular, discrete points in time, t_n . The ODE solvers presented in this booklet fall into this category – they provide samples of the solution at discrete times, similar to snapshots of the continuous function they try to approximate.

- summarizes the four different approximations for derivatives we have encountered so far. There are many more derivative approximations of higher order, and using more terms. The higher order approximations give better accuracy at the cost of additional complexity. Higher order approximations are used less frequently than the four shown here.

Commonly used finite difference derivatives.

Derivative	Method name	Expression	Error order
$\frac{dy}{dt}$	Forward difference	$\frac{y_{n+1}-y_n}{h}$	h
$\frac{dy}{dt}$	Backward difference	$\frac{y_n-y_{n-1}}{h}$	h
$\frac{dy}{dt}$	Central difference	$\frac{y_{n+1}-y_{n-1}}{2h}$	h^2
$\frac{d^2y}{dt^2}$	Second derivative	$\frac{y_{n+1}-2y_n+y_{n-1}}{h^2}$	h^2

This page titled [1: Introduction](#) is shared under a [CC BY-SA 3.0](#) license and was authored, remixed, and/or curated by [Stuart Brorson](#).

1.1: Solving ordinary differential equations

Solving ordinary differential equations (ODEs) is a major focus of numerical computing. We start by considering first-order ODEs of the form.

$$y' = f(t, y) \quad (1.1.1)$$

Where $y(t)$ is the unknown (i.e. the thing we want to find), and t is the independent variable. When dealing with such an ODE, we usually think of t as time, and y as something depending upon time, like the population of an animal species or the voltage of a node in an electronic circuit. Equation 1.1.1 says that the way y changes in time depends upon some function f . In this booklet, we assume the function f is smooth and well-behaved, and that it is easily calculable in a computer program.

One may write down solutions to 1.1.1, but to get a *unique* solution an additional piece of information is required: An initial condition. That is, we need to know something about $y(t)$ at one point in time so we can select one individual solution from all possible solutions which satisfy 1.1.1. One traditionally uses a value for $y(t)$ at time $t = 0$ as the initial condition, but other choices are also possible (but uncommon).

Here's a very basic example. Consider the ODE

$$y' = ay \quad (1.1.2)$$

You likely encountered this ODE early in your mathematical career, and you know the solution by heart:

$$y(t) = Ce^{at} \quad (1.1.3)$$

where C is some constant. The solution 1.1.3 obviously satisfies equation 1.1.2, but it is not useful in practice as the solution to the equation. Why? The reason is that there are an infinite number of possible solutions depending upon the value of C chosen. That is, 1.1.3 defines a family of solutions, but in a real, practical problem we only want one of those solutions. For example, how would you make a plot of 1.1.3? The answer is, you can't, until you chose a specific value of C .

In a practical problem, you usually know the value of your function at a start-point in time, almost always at $t = 0$. Call that value y_0 . In this case, the specific solution which satisfies 1.1.2 and also satisfies $y(t = 0) = y_0$ is

$$y(t) = y_0 e^{at} \quad (1.1.4)$$

The point is that numerically solving an ODE requires two pieces of information: 1. The ODE itself, and 2. An initial condition. The initial value picks one solution out of the entire family of solutions which satisfy 1.1.2. The concept is shown in 1.1

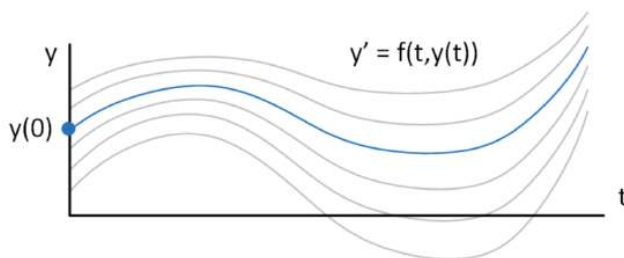


Figure 1.1: The ODE $y' = f(t, y)$ is satisfied by a family of curves. An initial condition $y(0)$ is required to pick out one of the solutions for numerical simulation.

Such a problem is called an "initial value problem" (IVP) to distinguish it from other types of differential equations (which we will encounter later in the class). "Initial value problem" simply means we know the beginning (initial) value of y , and want to find its behavior as we move forward in time. Once the equation 1.1.1 and an initial condition for y are specified, then we have all the information required to find a unique solution $y(t)$ numerically.

Methods to find numerical solutions IVPs are the focus of this booklet. As a matter of nomenclature, we generally say "solve the ODE", or "solve the IVP". This means we compute the function $y(t)$ starting from time $t = 0$ to some point in the future, t_{end} . Another way to say the same thing is we "integrate the ODE", or "integrate the IVP". When speaking of ODEs, "integrate" means the same thing as "solve".

1.1: Solving ordinary differential equations is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

- **1: Introduction** by [Stuart Brorson](#) is licensed [CC BY-SA 3.0](#).

1.2: Sampled functions

In mathematics we become familiar with the notion of a function. Consider the simplest case of a function $y(t)$ which takes a real scalar t and returns a real scalar value $y(t)$. This is frequently written formally as $y : \mathbb{R} \rightarrow \mathbb{R}$. This means, take any arbitrary real number t , plug it into y , and out will come a new real number $y(t)$. The important idea to glean from this is that the function is like a machine into which one can input any t and expect to get an output. That is, the function is defined on every real number t .

Unfortunately, when working on real data using a computer this nice abstraction doesn't always hold. Rather, it is very common for us to have only certain, discrete points t_n where the function is defined. Digital audio is a simple example: the sounds we hear with our ear are related to the sound pressure waves moving our eardrums. The sound pressure has a value for every point in time. But when processing audio using a computer the sound pressure is sampled at regular, discrete times and turned into numbers representing the sound pressure at each sample time. This process is often referred to as "discretization", meaning that the continuously-defined $y(t)$ is replaced by a set of discrete samples y_n . This is depicted in 1.2, which shows the relationship of a continuous-time function and its discrete-time (discretized) replacement.

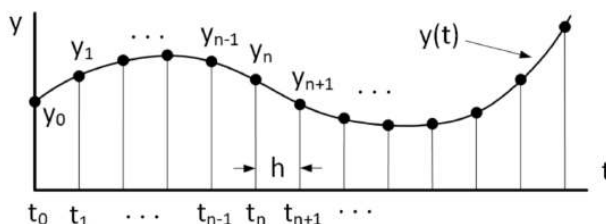


Figure 1.2: A sampled function. The continuous-time version of the function is $y(t)$. Sampling replaces $y(t)$ with samples taken at discrete times, t_n . The samples y_n may be handled by the computer as a vector or list of values. In the case shown here the function is sampled at evenly-spaced intervals separated by h . Evenly-spaced samples are the most frequent case – for example, digital audio commonly samples an input sound signal with a period h of about 22.7usec (44.1kHz sampling rate).

It turns out that the ODE solvers we will study all work with sampled functions. That is, the solvers compute the solution $y(t)$ at a sequence of discrete time values t_n similar to the situation shown in 1.2. The output of the solver will be a vector of discrete values y_n representing samples of the actual, underlying continuous function $y(t)$.

Here is an implementation hint when you write programs processing sampled functions: The sampled function itself is generally represented by a vector, y_n . This is the object your algorithm will use during its work. On top of y_n I recommend also carrying around a vector representing the sample times, t_n , in your program. Having your sample times readily accessible can help decrease confusion when you want to plot y_n vs. time, for example. For moderately sized vectors, the extra memory required to hold t_n is a small price to pay to keep confusion at a minimum while writing your code.

1.2: Sampled functions is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

- **1: Introduction** by [Stuart Brorson](#) is licensed [CC BY-SA 3.0](#).

1.3: Differentiation on the computer

Next before diving into solving (eq:1.1) we need to review how differentiation is done on the computer. To start, recall the definition of the derivative of $y(t)$ which you probably learned in high school:

$$\frac{dy}{dt} = \lim_{\Delta t \rightarrow 0} \frac{y(t + \Delta t) - y(t)}{\Delta t} \quad (1.3.1)$$

In doing numerical computation, we can't represent the limiting transition $\lim_{\Delta t \rightarrow 0}$ – our data is usually discrete (e.g. a sampled function). More importantly, floating point numbers don't smoothly transition to zero – there is no "infinitesimally small but nonzero" number representable on a computer. Instead, we use the definition 1.3.1 to form an approximation to the derivative as

$$\frac{dy}{dt} \approx \frac{y(t + h) - y(t)}{h} \quad (1.3.2)$$

where h is a small number. This is called the "forward difference" approximation of the derivative.

First derivatives from Taylor's series

It's easy to write down the approximation 1.3.2 by remembering the high school definition of a derivative. However, we gain some appreciation for the accuracy of approximating derivatives by deriving them using Taylor series expansions. Suppose we know the value of a function y at the time t and want to know its value at a later time $t + h$. How to find $y(t + h)$? The Taylor's series expansion says we can find the value at the later time by

$$y(t + h) = y(t) + h \left. \frac{dy}{dt} \right|_t + \frac{h^2}{2} \left. \frac{d^2y}{dt^2} \right|_t + \frac{h^3}{6} \left. \frac{d^3y}{dt^3} \right|_t + O(h^4) \quad (1.3.3)$$

The notation $O(h^4)$ means that there are terms of power h^4 and above in the sum, but we will consider them details which we can ignore.

We can rearrange 1.3.3 to put the first derivative on the LHS, yielding

$$\left. \frac{dy}{dt} \right|_t = \frac{y(t + h) - y(t)}{h} - \frac{h}{2} \left. \frac{d^2y}{dt^2} \right|_t + \frac{h^2}{6} \left. \frac{d^3y}{dt^3} \right|_t - O(h^4)$$

Now we make the assumption that h is very small, so we can throw away terms of order h and above from this expression. With that assumption we get

$$y(t + h) \approx y(t) + h \left. \frac{dy}{dt} \right|_t$$

which can be rearranged to give

$$\frac{dy}{dt} \approx \frac{y(t + h) - y(t)}{h} \quad (1.3.4)$$

i.e. the same forward difference expression as found in 1.3.2 earlier. However, note that to get this approximation we threw away all terms of order h and above. This implies that when we use this approximation we incur an error! That error will be on the order of h . In the limit $h \rightarrow 0$ the error disappears but since computers can only use finite h , there will always be an error in our computations, and the error is proportional to h . We express this concept by saying the error scales as $O(h)$.

Using Taylor's series expansions we can discover other results. Consider sitting at the time point t and asking for the value of y at time $t - h$ in the past. In this case, we can write an expression for the past value of y as

$$y(t - h) = y(t) - h \left. \frac{dy}{dt} \right|_t + \frac{h^2}{2} \left. \frac{d^2y}{dt^2} \right|_t - \frac{h^3}{6} \left. \frac{d^3y}{dt^3} \right|_t + O(h^4) \quad (1.3.5)$$

Note the negative signs on the odd-order terms. Similar to the forward difference expression above, it's easy to rearrange this expression and then throw away terms of $O(h)$ and above to get a different approximation for the derivative,

$$\frac{dy}{dt} \approx \frac{y(t) - y(t - h)}{h} \quad (1.3.6)$$

This is called the "backward difference" approximation to the derivative. Since we threw away terms of $O(h)$ and above, this expression for the derivative also carries an error penalty of $O(h)$.

Now consider forming difference between Taylor series 1.3.3 and 1.3.5. When we subtract one from the other we get

$$y(t+h) - y(t-h) = 2h \left. \frac{dy}{dt} \right|_t + 2 \frac{h^3}{6} \left. \frac{d^3y}{dt^3} \right|_t + O(h^5)$$

This expression may be rearranged to become

$$\left. \frac{dy}{dt} \right|_t = \frac{y(t+h) - y(t-h)}{2h} - \frac{h^2}{6} \left. \frac{d^3y}{dt^3} \right|_t + O(h^4)$$

Now if we throw away terms of $O(h^2)$ and above, we get the so-called "central difference" approximation for the derivative,

$$\left. \frac{dy}{dt} \right|_t \approx \frac{y(t+h) - y(t-h)}{2h} \quad (1.3.7)$$

Note that to get this expression we discarded $O(h^2)$ terms. This implies that the error incurred upon using this expression for a derivative is $O(h^2)$. Since we usually have $h \ll 1$, the central difference error $O(h^2)$ is smaller than the $O(h)$ incurred when using either the forward or backward difference formulas. That is, the central difference expression 1.3.7 provides a more accurate approximation to the derivative than either the forward or the backward difference formulas, and should be your preferred derivative approximation when you can use it.

In each of these derivations we threw away terms of some order p (and above). Discarding terms of order p and above means we no longer have a complete Taylor series. This operation is called "truncation", meaning we have truncated (chopped off) the remaining terms of the full Taylor series. The error incurred by this operation is therefore called "truncation error".

Second derivatives from Taylor's series

We can also get an approximation for the second derivative from expressions 1.3.3 and 1.3.5. This time, form the sum of the two expressions to get

$$y(t+h) + y(t-h) = 2y(t) + 2 \frac{h^2}{2} \left. \frac{d^2y}{dt^2} \right|_t + O(h^4)$$

This may be rearranged to get an expression for the second derivative,

$$\left. \frac{d^2y}{dt^2} \right|_t = \frac{y(t+h) - 2y(t) + y(t-h)}{h^2} - O(h^2)$$

If we drop the $O(h^2)$ term, we get the approximation to the second derivative,

$$\left. \frac{d^2y}{dt^2} \right|_t \approx \frac{y(t+h) - 2y(t) + y(t-h)}{h^2} \quad (1.3.8)$$

And since we dropped the $O(h^2)$ term to make this approximation, it means that use of 1.3.8 will incur an error of $O(h^2)$ when using this approximation.

1.3: Differentiation on the computer is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

- 1: Introduction by Stuart Brorson is licensed [CC BY-SA 3.0](#).

CHAPTER OVERVIEW

2: Forward Euler method

This page covers the Forward Euler method for solving first-order ordinary differential equations (ODEs), focusing on its implementation, error estimation (local truncation and global error), and limitations in accuracy and stability, particularly for exponential growth and oscillatory problems like the simple harmonic oscillator. The stability conditions are discussed, emphasizing the influence of step size on numerical solutions.

[2.1: Forward Euler algorithm](#)

[2.2: Example - the exponential growth ODE](#)

[2.3: Local truncation error vs. global solution error](#)

[2.4: Example - a driven system](#)

[2.5: Example - the logistic equation](#)

[2.6: Extending Forward Euler to higher order](#)

[2.7: Example - the simple harmonic oscillator](#)

[2.8: Propagator matrix for SHO](#)

[2.9: Stability](#)

[2.10: Chapter summary](#)

Forward Euler algorithm

Now we examine our first ODE solver: the Forward Euler method. Here is the problem and the goal: Given a scalar, first-order ODE,

$$\frac{dy}{dt} = f(t, y) \quad (2.1)$$

and an initial condition $y(t=0) = y_0$, find how the function $y(t)$ evolves for all times $t > 0$. In particular, write down an algorithm which may be executed by a computer to find the evolution of $y(t)$ for all times.

To derive the algorithm, first replace the exact equation with an approximation based on the forward difference derivative to get

$$\frac{y(t+h) - y(t)}{h} \approx f(t, y) \quad (2.2)$$

Now discretize the equation. That means we replace our function $y(t)$ defined on continuous t with a sampled function y_n defined on discrete times t_n . That is, $y_n = y(t_n)$. We also imagine the time step between samples is small, $h = t_{n+1} - t_n$. In this case, [2.2](#) becomes

$$\frac{y_{n+1} - y_n}{h} \approx f(t_n, y_n) \quad (2.3)$$

Moving forward, we will replace the $\approx$ sign with $=$ for convenience, and keep in mind that behind our expression lurks an approximation error which grows with increasing h . Next, imagine we are sitting at the present time point t_n , and rewrite [2.3](#) to move the future to the LHS and the present to the RHS. We get

$$y_{n+1} = y_n + hf(t_n, y_n) \quad (2.4)$$

Now observe: This expression says that if we know the values of t_n and y_n at the present, then to get the future value y_{n+1} we just perform the computation on the RHS. That is, equation [2.4](#) describes a way to step forward in time. We start at time $t = 0$ with initial condition $y = y_0$ and use equation [2.4](#) to step to y_1 , then use that result to step to y_2 , and then y_3 , and so on. By iterating [2.4](#) we generate a vector of y_n values which constitute the sampled version of the function $y(t)$.

Algorithm 1 shows pseudocode implementing the forward Euler method.

Algorithm 1: Forward Euler Method

Inputs: initial condition y_0 , end time t_{end} .

Initialize: $t_0 = 0, y_0$.

for $n = 0 : t_{end}/h$ **do**

$t_n = nh$

$y_{n+1} \leftarrow y_n + hf(t_n, y_n)$

end for

return y vector.

A graphical depiction of the algorithm is shown in 2.1.

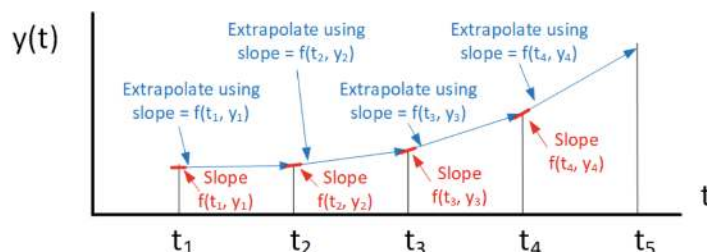


Figure 2.1: The forward Euler algorithm. At each time step the algorithm computes the slope of $f(t, y)$, then moves forward by step h to find the next value of y . The slope to use is the slope of the curve at the beginning of the interval. This figure shows a problem with this algorithm: the actual solution may curve away from the computed solution since forward Euler uses the old slope at t_n to compute the step, and that slope may not point the method to the correct y_{n+1} at time t_{n+1} .

Here are some things to notice about forward Euler:

- Referring to 2.1, you can see we compute the solution relies on computing the slope of y using $f(t, y)$ (i.e. the derivative) at each point t_n , then using the slope to extrapolate y forward one step. In the figure the thick blue line denotes the slope at each t_n . Note that the slope used is the slope at the beginning of the step interval – hence the name "forward" Euler. You can see how each forward step is determined by the slope of the blue line.
- Forward Euler is an instance of a so-called "explicit method". Explicit means all calculations required to take the step are present on the RHS of 2.4.
- shows clearly a problem with forward Euler: The actual solution can curve away from the forward Euler solution, causing an error to build up in the forward Euler solution. The error is clearly dependent upon the size of h : Larger h will lead to larger error, while smaller h leads to smaller error. The difficulty with smaller h is that smaller step sizes will require more steps – and longer computation time – to reach the same end time, t_{end} .

Besides understanding the math behind the method, there are important implementation features to notice. The code implementing forward Euler is broken into three parts:

- A top level main program called "test forward euler". This is the program run by the user. It sets the model parameters used and invokes the solver itself. It then makes plots of the result.
- The solver implementation called "forward euler". This function is written in a generic way, so it does not set any details of either the parameters of the particular ODE under consideration, nor algorithm features like the step size. The goal is to have a generic solver which may be used on any ODE without modification.
- An implementation of the ODE equations contained in a file called "f". This corresponds to the function $f(t_n, y_n)$ in 2.4. Model parameters are passed directly from the top level to "f" as global variables.

A block diagram showing this architecture is presented in 2.2.

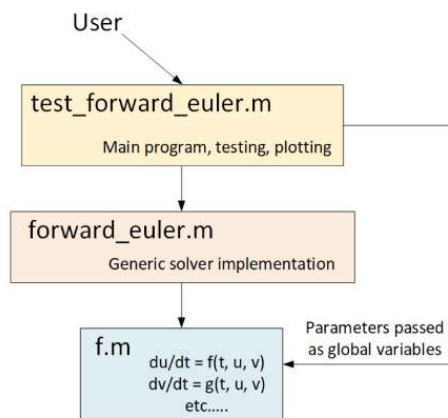


Figure 2.2: The three-layer architecture used in the example Matlab programs implementing forward Euler. The user runs the main program called "test forward euler". This in turn calls the solver "forward euler", which in turn calls the function "f" which contains the actual ODE. A similar design pattern is used for almost all solvers presented in this booklet.

The architecture in 2.2 is a simple "software pattern", meaning that it is a good way to partition the code into different pieces, each of which perform a specific function. Almost all ODE solver examples I present in this booklet are architected using this three-layer pattern. Real-world ODE solvers like Sundials or DifferentialEquations.jl are also frequently partitioned into different pieces, although their architectures are generally more complicated and have more pieces. The additional pieces might handle e.g initialization of the solver, management of memory, error handling, sensitivity analysis, and so on. The point is that it is helpful to partition your program into different pieces which handle logically distinct functions. Indeed, this is one of the most important things you learn to do when becoming a software engineer.

Example: the exponential growth ODE

As a first example of our simple solver, we'll apply forward Euler to integrating the exponential growth ODE,

$$\frac{dy}{dt} = \alpha y \quad (2.5)$$

We already know the solution to this equation – it is $y(t) = y(0)e^{\alpha t}$. For $\alpha < 0$ the solution decays to zero for $t \rightarrow \infty$, while for $\alpha > 0$ the solution escapes to infinity for $t \rightarrow \infty$.

Now use the prescription to discretize the equation for forward Euler and get

$$y_{n+1} = y_n + h\alpha y_n \quad (2.6)$$

The result computed using forward Euler is shown in 2.3 for varying values of α .

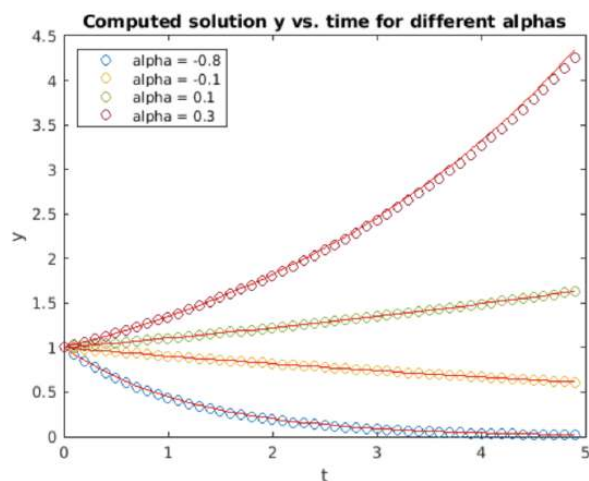


Figure 2.3: The solution to 2.5 computed using the forward Euler algorithm for different values of α . The computed function y_n is plotted using open circles and the analytic solution is plotted using red lines. The stepsize used was $h = 0.1$. Note that the computed and analytic solutions don't exactly match, particularly the case where $\alpha = 0.3$.

The figure shows the computed solution doesn't exactly track the analytic solution. What is wrong? The answer has to do with the errors incurred by using the forward difference formula to approximate the derivative. Recall that the forward difference expression (eq:1.8) is only true in the limit where the stepsize goes to zero, $h \rightarrow 0$. For non-zero h , the forward difference approximation is inaccurate, and per the derivation presented in 1.3.1 the magnitude of the error scales as $O(h)$. We can quantify this by examining the RMS error of the computed solution. The RMS error is computed by comparing the root-mean-square difference between the computed and the analytic solution as follows:

$$e = \sqrt{\frac{1}{N} \sum_{i=1}^N (y_t(i) - y_c(i))^2} \quad (2.7)$$

Here, y_t is the "mathematically true" (analytic) solution, y_c is the solution computed using forward Euler, N is the number of sample points in the solution, and e is the RMS error. This is also called the "global error" in some textbooks. We expect that the RMS error will increase with increasing h – larger stepsizes accumulate larger errors. The question is, how does the error behave with h ? A plot of RMS error vs. h is shown in fig. 2.4.

On the log-log plot the RMS error increases with h by following a straight line. A useful feature of log-log plots is that it clearly reveals power-law relationships. Specifically, if a function obeys a power-law $y \sim x^p$ then the slope of the y vs. x line on a log-log plot is the power p . This can be understood by taking the log of both sides of the power-law (which is what plotting on a log-log scale does) and noting that p becomes the slope of the line. The slope of the line in fig. 2.4 is exactly

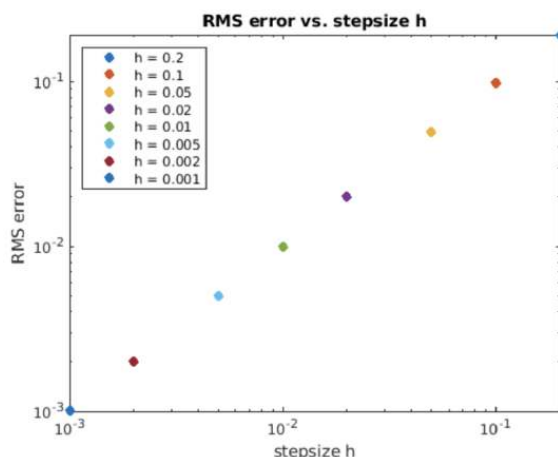


Figure 2.4: The error incurred by the forward Euler method vs. stepsize h plotted on a log-log scale. Note that the slope of the error points is one. This implies the error $e \sim O(h)$.

1 – This means the RMS error incurred by the forward Euler method is proportional to the stepsize

h – that is, error $e \sim O(h)$. This scaling relationship was derived in 1.3.1 when the forward difference expression was derived from the Taylor series – terms of $O(h)$ and above were discarded to get the forward difference expression, which means that using the approximation carries an $O(h)$ error penalty.

Detour: Local truncation error vs. global solution error

Back in 1.3.1 I showed how the approximations to the derivative were derived from the Taylor series. I also mentioned that by dropping higher-order terms, we were implicitly accepting that our computed solutions could only be approximations to the "mathematically true" solution. That is, the computed solution carried an error, and the major cause of error was due to truncating the Taylor series. This error is sometimes called the "local truncation error" or "LTE" in the ODE literature. We generally characterize the LTE in terms of the stepsize scaling $O(h^p)$. Generally, the error order p is the same as the first truncated term in the Taylor series. That is, if the first truncated term is h then the LTE scales as $O(h)$.

Now in the previous section we introduced the RMS error 2.7. Since the RMS error is a sum over the entire solution, it is a global error – it adds up contributions from all errors present in the entire solution. (It is normalized by N , but nonetheless is sensitive to all errors.) Therefore, this error is sometimes called the "global error" or GE in the ODE literature. Again, the GE is characterized by how it scales with h . All error analyses shown in this booklet are global errors computed using the formula 2.7 and are expressed as scaling plots similar to that shown in fig. 2.4.

The LTE and the GE are conceptually distinct quantities. Therefore, you might ask "what is the relationship between the two?" That is, do they measure the same thing? The answer is "yes". Although I won't do it here, it can be proven that the LTE and the GE obey the same scaling relationship for any given method. That is, if the LTE scales as $O(h^p)$, then the GE also scales as $O(h^p)$. Accordingly, although I speak loosely about "the error" without quantifying exactly what I mean, the equivalence of the LTE and the GE are sufficient to guarantee the term "the error" refers to something which may be quantified, and is a meaningful metric with which to measure the different ODE solvers presented in this booklet.

Example: a driven system

Now back to solving ODEs. A slightly more complicated ODE is this linear ODE with a driving term.

$$\frac{dy}{dt} = A \sin(\omega t) - \alpha y \quad (2.8)$$

Since the equation is linear, the general solution is the sum of two pieces: a homogeneous term and an inhomogeneous term (sometimes called the "particular solution"). We'll consider the case of positive α . That is, for $t \rightarrow \infty$ the homogeneous solution will decay away as $e^{-\alpha t}$ so the long-term behavior of the system is determined by the forcing term $A \sin(\omega t)$. The analytic solution to the equation is may be found after some algebra,

$$y(t) = Ce^{-\alpha t} + B \cos(\omega t + \phi) \quad (2.9)$$

We adjust the constants B , C , and ϕ to match the initial conditions as well as ensure the particular solution obeys 2.8 for $t \rightarrow \infty$. To make the plot shown below in fig. 2.5 we choose $y(0) = 1$. With this initial condition, an exercise in easy algebra will show the fitting constants are

$$\begin{aligned} \phi &= \arctan\left(\frac{\alpha}{\omega}\right) \\ B &= \frac{-A}{\omega \cos(\phi) + \alpha \sin(\phi)} \\ C &= 1 - B \cos(\phi) \end{aligned} \quad (2.10)$$

Discretized for forward Euler, the iteration is

$$y_{n+1} = y_n + h(A \sin(\omega t_n) - \alpha y_n) \quad (2.11)$$

Figure 2.5 shows the computed solution to 2.8 for 15 seconds assuming $A = 1.5$, $\omega = 5$ and $\alpha = 0.5$. The solution clearly shows an initial transient which decays away, leaving a solution which behaves like a sine wave as $t \rightarrow \infty$.

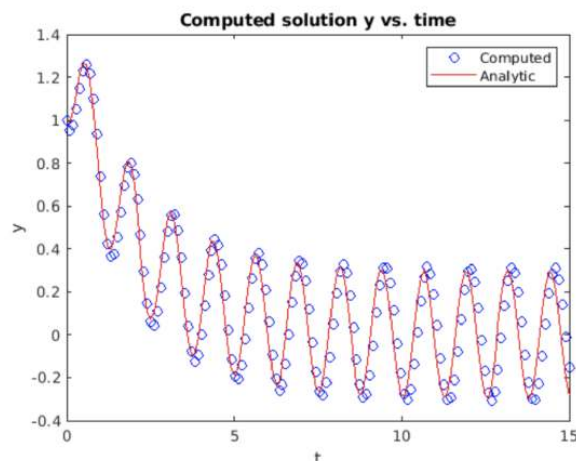


Figure 2.5: The solution to (eq:ForwardEulerDriven) computed using the forward Euler algorithm. The computed function y_n is plotted by the blue circles and the analytic solution is plotted using red lines. Although the two solutions are close, they don't exactly match, and the forward Euler solution seems to diverge from the analytic solution with increasing t .

Example: the logistic equation

Here is an example of a nonlinear ODE: the logistic equation. This ODE can be regarded as a very simple model of population dynamics. Here's the ODE:

$$\frac{dy}{dt} = \left(1 - \frac{y}{Y_m}\right) y \quad (2.12)$$

If you think of y as the population of some animal species, when $y \ll Y_m$, then the population can grow exponentially. But as the population $y \rightarrow Y_m$, the growth rate of the population saturates. Think of rabbits living in a carrot field – when there are very few rabbits (and no predators) then the rabbits have lots of carrots to eat and they may reproduce freely, leading to exponential growth. However, if too many rabbits live in the field, then they eat most of the carrots and can't reproduce freely anymore since food is scarce. Consequently, the rabbit population tends towards a fixed limit given by $y = Y_m$.

This equation has the following analytic solution:

$$y(t) = \frac{y_0 e^t}{\left(1 + \frac{y_0}{Y_m} e^t\right)} \quad (2.13)$$

where y_0 is used to match the initial condition. Inspection shows $y_0 = y(t = 0)$. Also note that when $y = Y_m$ the growth rate (derivative) goes to zero, consistent with the intuition that the rabbit population should saturate when the rabbits eat most of the carrots. Just enough carrots remain to carry the rabbit population, but the population can't grow any more.

Discretized for forward Euler, the iteration to implement for the logistic equation is

$$y_{n+1} = y_n + h \left(1 - \frac{y_n}{Y_m}\right) y_n \quad (2.14)$$

This iteration was implemented in Matlab and then run for three different values of Y_m . The results are shown in fig. 2.6. Forward Euler reproduces the saturation behavior of the logistic equation quite well – after around $t = 10$ the forward Euler solution matches the analytic solution. However, forward Euler does a worse job reproducing the period of exponential growth around $t = 5$ – forward Euler lags the analytic solution. This behavior is similar to that observed in the exponential growth ODE shown in fig. 2.3.

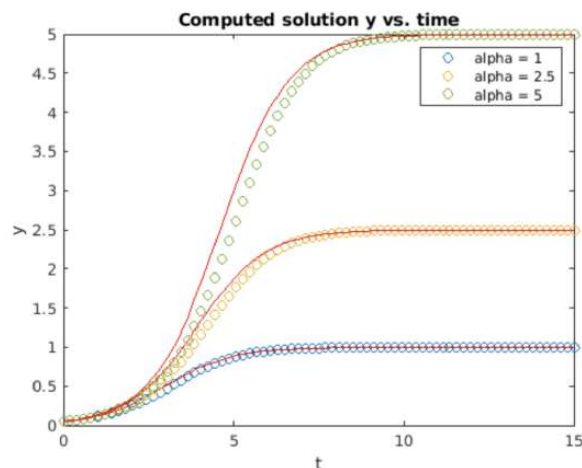


Figure 2.6: The solution to (eq:ForwardEulerLogistic) computed using the forward Euler algorithm for three different Y_m values. The computed function y_n is plotted using open circles and the analytic solution is plotted using red lines. Note that the computed solution lags (is behind) the analytic solution – forward Euler’s solution carries an error. One can imagine the rabbit population starting from a low base and growing until it hits a saturation point. The saturation point is determined by the value of Y_m .

Extending Forward Euler to higher order

So far, we have dealt with scalar, first-order ODEs like that shown in Equation 2.1. Many ODEs encountered in practice are higher order. For example, consider the most famous second order ODE of all: Newton’s second law: $F = ma$. It says that imposing a force on a body will cause it to accelerate in the direction of the applied force. Since acceleration is the second derivative of the body’s position, $a = d^2y/dt^2$, this implies we can write Newton’s second law as an ODE:

$$\frac{d^2y}{dt^2} = F(t, y', y) \quad (2.15)$$

Given the importance of Newton’s law, we want to be able to solve ODEs like 2.15 numerically. How to extend the forward Euler method to second (and higher) order ODEs?

The way to solve 2.15 numerically is to rearrange it to become two first order ODEs as follows:

1. Define two variables, $y_1 = dy/dt$ and $y_2 = y$.
2. Now note that 2.15 may be written as

$$\frac{dy_1}{dt} = \frac{d^2y}{dt^2} = F(t, y', y)$$

3. Next, by definition we have

$$\frac{dy_2}{dt} = \frac{dy}{dt} = y_1$$

4. Therefore, we have broken 2.15 up into two coupled, first-order ODEs,

$$\begin{aligned} \frac{dy_1}{dt} &= F(t, y_1, y_2) \\ \frac{dy_2}{dt} &= y_1 \end{aligned} \quad (2.16)$$

5. This system of two first order ODEs may be solved using forward Euler using the same methods as in 2.1.

This technique of breaking a second order ODE into two first order ODEs may be easily generalized. As a rule, if you start with an n th order ODE you may break it down into n first order ODEs. Then the system of first order ODEs may be solved using forward Euler or any of the more advanced methods we will encounter later.

It’s not enough to simply specify the equations to solve. We also need to specify initial conditions (ICs) for this system – one IC for each equation. In general, an n th order ODE will be accompanied by n initial conditions, usually one for the function $y(t = 0)$, one for the first derivative $y'(t = 0)$, one for the second derivative $y''(t = 0)$, and so on. Therefore, a second order ODE such as 2.15 will

require two ICs. Using the definitions of y_1 and y_2 above, it's easy to map the ICs for the second order equation onto the ICs for the system as

$$\begin{aligned} y_1(0) &= y'(0) \\ y_2(0) &= y(0) \end{aligned} \quad (2.17)$$

Example: the simple harmonic oscillator

To make these ideas concrete we will examine the so-called simple harmonic oscillator (SHO). This is a simple model of oscillatory motion beloved by physicists, and encountered in many real-world applications. One situation modeled by the SHO is a mass sliding on a (frictionless) floor, and tethered to a wall by a spring. See fig. 2.7. The force impressed by the spring on the mass is modeled using Hooke's law, which says the spring tries to restore the mass to an equilibrium position with a force varying linearly with position. Taking y as the position of the mass, Hooke's law says $F = -k(y - y_r)$, where k is a constant measuring the stiffness of the spring, and y_r is the rest position of the mass. From now on out we will set $y_r = 0$ since it is a constant and doesn't affect the math we want to explore.

Combining Hooke's law and Newton's second law, we can write down a second order ODE describing the motion of the mass under the influence of the spring,

$$\frac{d^2 y}{dt^2} = -\frac{k}{m} y \quad (2.18)$$

You probably already know the solution(s) to this equation. They are sines and cosines,

$$y(t) = A \cos(\omega t) + B \sin(\omega t) \quad (2.19)$$

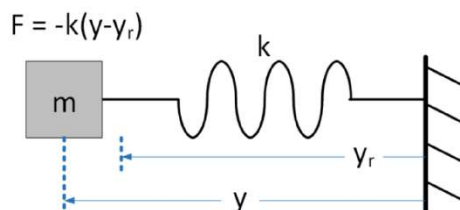


Figure 2.7: The simple harmonic oscillator (SHO). The mass m slides frictionlessly on the floor. It is attached to a wall through a spring. The spring constant (measure of spring stiffness) is k . When the mass is displaced from its rest position y_r the spring tends to push or pull the mass back to the rest position. However, the mass overshoots the rest position due to Newton's second law. Therefore, the spring will execute a sinusoidal oscillation around y_r , and if the system is lossless the oscillation will never stop. The oscillatory motion is governed by (eq:SHO), which is known as the simple harmonic oscillator equation.

where we have defined $\omega = \sqrt{k/m}$ for convenience. ω is generally referred to as the oscillation frequency. The coefficients A and B are determined by the initial conditions.

How to solve 2.18 numerically? We'll use our prescription from section 2.6 and break 2.18 into two first order ODEs. We define $u = dy/dt$ and $v = y$. (I use u and v instead of y_1 and y_2 for notational convenience in what comes next.) Now we can decompose 2.18 into

$$\begin{aligned} \frac{du}{dt} &= -\frac{k}{m} v \\ \frac{dv}{dt} &= u \end{aligned} \quad (2.20)$$

Now discretize using the forward difference approximation to the first derivative:

$$\begin{aligned} \frac{u_{n+1} - u_n}{h} &= -\frac{k}{m} v_n \\ \frac{v_{n+1} - v_n}{h} &= u_n \end{aligned} \quad (2.21)$$

Next rearrange these equations to place the future on the LHS and the present on the RHS:

$$\begin{aligned} u_{n+1} &= u_n - h \frac{k}{m} v_n \\ v_{n+1} &= v_n + h u_n \end{aligned} \quad (2.22)$$

Now we have a pair of equations which we can step forward in time! We start with known initial conditions u_0 and v_0 and use the system to compute the next values in time u_1 and v_1 , then use those to get the next values, and so on.

This algorithm is easy to implement using e.g. Matlab. The results of running a Matlab program implementing the iteration 2.22 are shown in fig. 2.8. Results for two different step sizes ($h = 0.01$ and $h = .001$) are shown for comparison. The initial conditions $u_0 = 1$ and $v_0 = 0$ were used in both cases. These ICs correspond to an oscillating mass whose starting position is 1 and whose starting velocity is 0 – similar to pulling the mass by hand to position 1, holding it there for a moment, then releasing it.

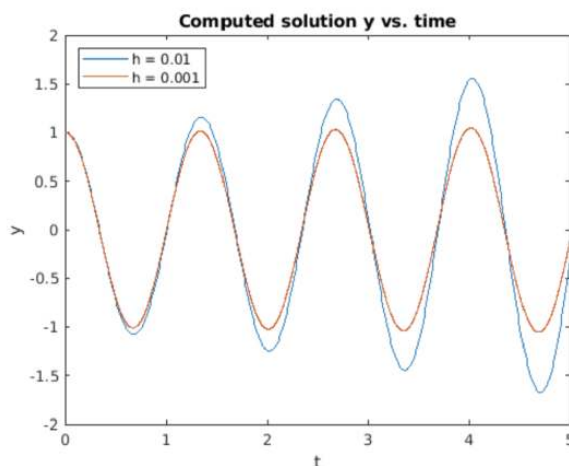


Figure 2.8: The simple harmonic oscillator (SHO) solved using forward Euler. Note that the sinusoidal amplitude increases exponentially with time. This is contrary to the known solution involving pure sines and cosines. Why does this happen? This effect has to do with the concept of stability discussed later in 2.9.

Things to note in the figure:

- Both plots show the mass oscillates back and forth in time. This is to be expected, since the solutions to 2.18 are sines and cosines.
- However, we see that both solutions grow in time. The $h = 0.01$ solution grows faster than the $h = .001$ one, but close inspection of the plot shows they both grow. This is unexpected, and is incorrect – the ODE 2.18 has sin and cos solutions which don't grow. What is wrong? The answer to this question involves studying the stability of the forward Euler method.

Propagator matrix for SHO

Before we talk about stability, it is convenient to recast the iteration 2.22 into a different format. Because 2.18 is linear, we can take the forward Euler system one step further. Note that the following doesn't work with all ODEs – only with linear ODEs! With a little rearranging we can write 2.22 as a matrix-vector system,

$$\begin{pmatrix} u_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -hk/m \\ h & 1 \end{pmatrix} \begin{pmatrix} u_n \\ v_n \end{pmatrix} \quad (2.23)$$

or, written in matrix-vector form,

$$y_{n+1} = Ay_n$$

where

$$y_n = \begin{pmatrix} u_n \\ v_n \end{pmatrix}$$

and A is the matrix shown in 2.23. A is commonly called the "propagator matrix" since repeated multiplication by A propagates (steps) the solution y_n forward in time.

Pseudocode implementing the forward Euler algorithm to solve the simple harmonic oscillator is shown in [alg:2].

Algorithm 2: Forward Euler method for SHO

Inputs: initial condition $y_0 = [u_0, v_0]^T$, end time t_{end} .

Initialize: $t_0 = 0, y_0$.

for $n = 0 : t_{end}/h$ **do**

$y_{n+1} \leftarrow Ay_n$

end for

return y_n .

The results of a run using this propagator method are shown in 2.9. The results are the same as those obtained in section 2.7

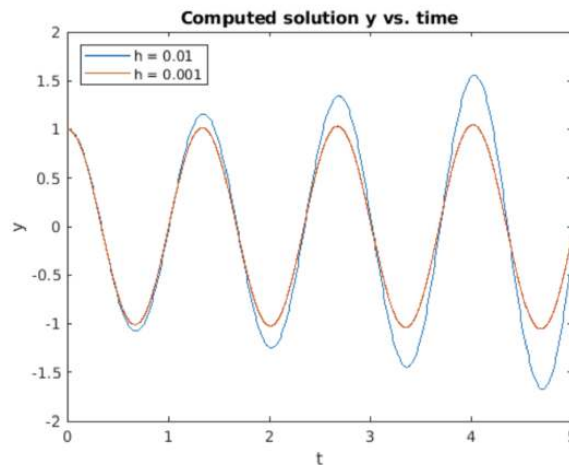


Figure 2.9: The simple harmonic oscillator (SHO) integrated using the propagator iteration. Use of the propagator matrix only works when the ODE under consideration is linear and each step may be turned into a matrix-vector multiplication.

Stability

In deriving the expressions for finite difference approximations we found that the process of discretization produced a difference (error) between the exact (analytical) derivative and the finite difference approximation. Now we find a second pitfall in numerically solving ODEs: Growing numerical solutions when we know the exact (analytic) solution should not grow in time. What is happening?

Stability of forward Euler for the exponential growth ODE

To understand the observed growth consider the exponential growth ODE:

$$\frac{dy}{dt} = ay \quad (2.24)$$

Note that this ODE is linear. We know this equation has an exact, mathematically-true solution $y_t(t) = y_0 e^{at}$, where we denote the exact, mathematically-true solution as y_t . Now ask the question: What happens if the exact solution is perturbed by some error e ? That is, take $y = y_t(t) + e(t)$. In this case linearity says we can write

$$\frac{dy_t}{dt} + \frac{de}{dt} = a(y_t + e)$$

This equation separates into two pieces,

$$\frac{dy_t}{dt} = ay_t$$

which is satisfied identically because y_t is the exact solution by definition, and

$$\frac{de}{dt} = ae$$

Which governs the behavior of the error term e . Now discretize this term using the forward Euler method to get

$$e_{n+1} = e_n + hae_n = (1 + ah)e_n \quad (2.25)$$

where e_n is the error at step n . Assuming we start with a small perturbation e_0 at time $t = 0$, and just watch the evolution of this one perturbation in time, we find 2.25 behaves as

$$e_n = (1 + ah)^n e_0 \quad (2.26)$$

Clearly, the perturbation grows exponentially as $n \rightarrow \infty$ if $|1 + ah| > 1$ but decays to zero as long as

$$|1 + ah| < 1 \quad (2.27)$$

Equation (2.23) is the forward Euler stability condition for the exponential growth ODE. A solver applied to an ODE is stable if an initial error or perturbation either maintains the same magnitude or dies out in time. But if an initial error grows exponentially, then the solver is unstable. Note that the property of stability depends upon both the solver and the ODE itself – a solver which is stable for one ODE might be unstable for a different one, and different solvers may be stable or unstable for the same ODE.

So now the question becomes, when is forward Euler stable for the simple ODE 2.24? For forward Euler to make sense, the step h must be a small, positive number. However, we have placed no constraints on a – it can be positive, negative, or even complex. Therefore, to understand when 2.24 is stable it makes sense to plot the region of stability in the complex plane of ah . This is shown in fig. 2.10. The region of stability is the grey circle to the left of the origin. Values inside the circle represent ah values which do not blow up – they are stable because they satisfy the stability condition $|1 + ah| < 1$.

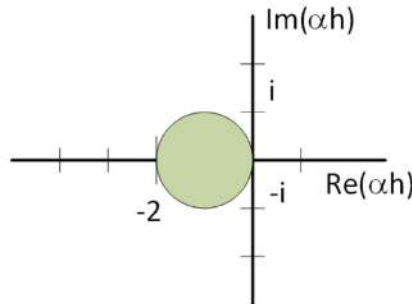


Figure 2.10: Forward Euler's stability region for the simple linear ODE 2.24. The stability region for this system is a circle of radius 1 in the complex plane centered at $ah = -1$.

Stability of forward Euler for the simple harmonic oscillator

Now we consider the stability of forward Euler applied to the simple harmonic oscillator. Recall the forward Euler iteration for this equation,

$$\begin{pmatrix} u_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -hk/m \\ h & 1 \end{pmatrix} \begin{pmatrix} u_n \\ v_n \end{pmatrix} \quad (2.28)$$

We know the "mathematically-true" solution to the SHO is composed of sines and cosines, or equivalently of complex exponentials, $e^{i\omega t}$. We'll choose one solution,

$$\begin{pmatrix} u_n \\ v_n \end{pmatrix} = \begin{pmatrix} e^{i\omega t} \\ i\omega e^{i\omega t} \end{pmatrix} \quad (2.29)$$

and examine its time evolution. Instead of considering an additive perturbation, we instead consider tracking errors using a multiplicative "error factor" e_n . That means we assume the solution

$$\begin{pmatrix} u_n \\ v_n \end{pmatrix} = e_n \begin{pmatrix} e^{i\omega t} \\ i\omega e^{i\omega t} \end{pmatrix} \quad (2.30)$$

and insert it into 2.28 and see how e_n behaves in time. Clearly, if $e_n = 1$ for all time, then 2.30 solves 2.28 exactly. But if $e_n \neq 1$ then the forward Euler iteration introduces errors to the solution.

Inserting 2.30 into 2.28 we get

$$e_{n+1} \begin{pmatrix} e^{i\omega(t+h)} \\ i\omega e^{i\omega(t+h)} \end{pmatrix} = e_n \begin{pmatrix} 1 & -\frac{hk}{m} \\ h & 1 \end{pmatrix} \begin{pmatrix} e^{i\omega t} \\ i\omega e^{i\omega t} \end{pmatrix}$$

Now divide out by the common factor $e^{i\omega t}$ and define the so-called "growth factor", $g_n = e_{n+1}/e_n$ to get

$$g_n e^{i\omega h} \begin{pmatrix} 1 \\ i\omega \end{pmatrix} = \begin{pmatrix} 1 & -\frac{hk}{m} \\ h & 1 \end{pmatrix} \begin{pmatrix} 1 \\ i\omega \end{pmatrix}$$

Now we see that this expression has the form of an eigenvalue problem, $\lambda x = Ax$, where the scalar $g_n e^{i\omega h}$ plays the role of the eigenvalue. We can solve for $g_n e^{i\omega h}$ the usual way by subtracting the LHS from the RHS,

$$\begin{pmatrix} 1 - g_n e^{i\omega h} & -\frac{hk}{m} \\ h & 1 - g_n e^{i\omega h} \end{pmatrix} \begin{pmatrix} 1 \\ i\omega \end{pmatrix} = 0$$

and observing that for this to hold, we need values of $g_n e^{i\omega h}$ which make the matrix singular. Therefore, we require

$$\begin{vmatrix} 1 - g_n e^{i\omega h} & -\frac{hk}{m} \\ h & 1 - g_n e^{i\omega h} \end{vmatrix} = 0$$

or

$$(1 - g_n e^{i\omega h})^2 + h^2 \frac{k}{m} = 0$$

Next, again recall the definition of the "natural frequency" of the SHO, $\omega_0^2 = k/m$. Then rearrange to solve for g_n and get

$$(1 - g_n e^{i\omega h})^2 = -h^2 \omega_0^2$$

or

$$g_n e^{i\omega h} = 1 \pm ih\omega_0$$

Now we ask, under which conditions does the error factor e_n remain constant, or at least not grow or shrink? e_n remains constant as long as $|g_n| = 1$. That is, we require

$$|g_n| = |1 \pm ih\omega_0| \quad (2.31)$$

However, this clearly never happens – the RHS of Eq. 2.26 is always greater than one. This is evident by inspecting the RHS in the complex plane as shown in fig. 2.11. Therefore, the solution computed by forward Euler always grows. For increasingly small step size h the growth factor tends to 1, but never reaches 1 until h is exactly zero – a situation which makes the forward Euler method useless since you can't step forward in time if $h = 0$. This means the forward Euler method is never useful for solving the simple harmonic oscillator problem – we need to find better methods.

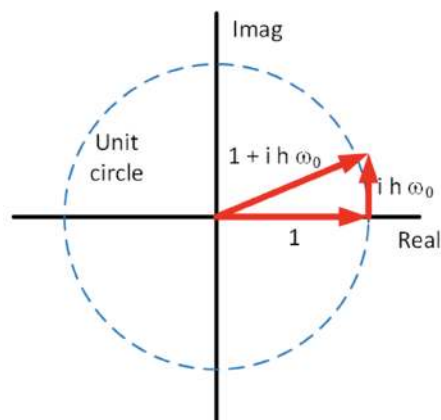


Figure 2.11: The RHS of 2.31. The real part is always 1, while the imaginary part can take any value. However, as long as the imaginary part is not identically zero, then the magnitude of $1 \pm i h \omega_0$ is always greater than 1. Therefore, the error factor e_n will always grow, and the SHO is always unstable for any non-zero stepsize h .

Chapter summary

Here are the important points made in this chapter:

- The forward Euler method is derived from the simple forward difference expression for the derivative, $y' = (y_{n+1} - y_n)/h$.
- The forward Euler method is an iterative method which starts at an initial point and walks the solution forward using the iteration $y_{n+1} = y_n + h f(t_n, y_n)$.
- Since the future is computed directly using values of t_n and y_n at the present, forward Euler is an explicit method.
- The forward Euler method is defined for 1st order ODEs. It works for both one- and many-variable ODEs. To solve an Nth order ODE one breaks the ODE into N first order ODEs.
- The RMS error expected when using the forward Euler method scales as $e \sim O(h)$.
- Solver stability is not guaranteed for all step sizes h . The stability criterion depends upon the exact ODE under consideration, but in general smaller step sizes are required for stability when using explicit methods.

This page titled 2: [Forward Euler method](#) is shared under a [CC BY-SA 3.0](#) license and was authored, remixed, and/or curated by [Stuart Brorson](#).