Gerson J. Ferreira
Nanosciences Group www.infis.ufu.br/gnano

Introduction to Computational Physics
with examples in Julia

This is a draft!
Please check for an updated version on
`http://www.infis.ufu.br/gerson`
Currently, I'm not working on this text. I'll start to review and improve/finish some sections next semester.

Institute of Physics, Federal University of Uberlândia
October 25, 2017

Contents

Introduction **1**

1 Introduction to the Julia language **3**

1.1 Installing Julia (needs update) 3

1.1.1 JuliaBox 3

1.1.2 Juno IDE 6

1.1.3 iJulia 6

1.2 Trying Julia for the first time 7

1.2.1 The command line interface 7

1.2.2 Using scripts 9

1.2.3 Other interfaces 10

1.3 Constants and variable types 10

1.3.1 Assertion and Function Overloading 11

1.3.2 Composite Types (struct) 12

1.3.3 Tuples 13

1.3.4 Arrays: vectors and matrices 14

1.3.5 Scope of a variable 17

1.4 Control Flow: if, for, while and comprehensions 18

1.4.1 Conditional evaluations: if 19

1.4.2 Ternary operator ? : 19

1.4.3 Numeric Comparisons 20

1.4.4 For and While Loops 20

1.4.5 Comprehensions 21

1.5 Input and Output 22

1.6 Other relevant topics 23

1.6.1 Passing parameters by reference or by copy (not finished) 23

1.6.2 Operator Precedence 23

1.7 Questions from the students 23

1.8 Problems 24

2 Differential and Integral Calculus 27
2.1 Interpolation / Discretization 27
2.2 Numerical Integration - Quadratures 30
2.2.1 Polynomial Interpolations 31
2.2.2 Adaptive and multi-dimensional integration 33
2.2.3 Monte Carlo integration 34
2.3 Numerical Derivatives 35
2.3.1 Finite differences and Taylor series 35
2.3.2 Matrix Representation 37
2.3.3 Derivatives via convolution with a kernel 38
2.3.4 Other methods, Julia commands and packages for derivatives . 39
2.4 Problems 39

3 Ordinary Differential Equations 43
3.1 Initial value problem: time-evolution 44
3.1.1 The Euler method 46
3.1.2 Runge-Kutta Methods 47
3.1.3 Stiff equations 49
3.1.4 Julia's ODE package 52
3.2 Boundary-value problems 53
3.2.1 Boundary conditions: Dirichlet and Neumann 53
3.2.2 The Sturm-Liouville problems 55
3.2.3 The Wronskian method 57
3.2.4 Schroedinger equations: transmission across a barrier 59
3.2.5 Non-linear differential equations 61
3.3 The eigenvalue problem 62
3.3.1 Oscillations on a string 63
3.3.2 Electron in a box 63
3.3.3 Method of finite differences 64
3.4 Problems 65

4 Fourier Series and Transforms 69
4.1 General properties of the Fourier transform 71
4.2 Numerical implementations 72
4.2.1 Discrete Fourier Transform (DFT) 72
4.2.2 Fast Fourier Transform (FFT) 72
4.2.3 Julia's native FFT 73
4.3 Applications of the FFT 75
4.3.1 Spectral analysis and frequency filters 75

4.3.2 Solving ordinary and partial differential equations 76
4.4 Problems 79

5 Statistics (TO DO) 81
5.1 Random numbers 81
5.2 Random walk and diffusion 81
5.3 The Monte Carlo method 81

6 Partial Differential Equations (TO DO) 83
6.1 Separation of variables 83
6.2 Discretization in multiple dimensions 83
6.3 Iterative methods, relaxation 83
6.4 Fourier transform (see §4.3.2) 83

7 Plotting (not finished) 85
7.1 PyPlot 85
7.1.1 Installing PyPlot 85
7.1.2 Using PyPlot 86
7.1.3 Most useful features and examples 86

8 Other topics (TO DO) 91
8.1 Linear algebra 91
8.2 Root finding 91
8.3 Linear systems 91
8.4 Least squares 91

Bibliography 93

Introduction

These are the class notes for the course “Computational Physics”, lectured to the students of *Bacharelado em Física de Materiais*, *Física Médica*, and *Licenciatura em Física* from the *Instituto de Física, Universidade Federal de Uberlândia* (UFU). Since this is an optional class intended for students with diverse knowledge and interest in Computational Physics and computer languages, I choose to discuss selected topics superficially with practical examples that may interest the students. Advanced topics are proposed as *home projects* to the students accordingly to their interest.

The ideal environment to develop the proposed lessons is the Linux operational system, preferably Debian-based distributions[1]. However, for the lessons and examples of this notes we use the recently developed `Julia`[2] language, which is open-source and available to any OS (GNU/Linux, Apple OS X, and MS Windows). This language has a syntax similar to MATLAB, but with many improvements. Julia was explicitly developed for numerical calculus focusing in vector and matrix operations, linear algebra, and distributed parallel execution. Moreover, a large community of developers bring extra functionalities to Julia via packages, *e.g.:* the `PyPlot` package is plotting environment based on `matplotlib`, and the `ODE` package provides efficient implementation of adaptive Runge-Kutta algorithms for ordinary differential equations. Complementing Julia, we may discuss other efficient plotting tools like `gnuplot`[3] and `Asymptote`[4].

Home projects. When it comes to Computational Physics, each problem has a certain degree of difficulty, computational cost (processing time, memory, ...) and challenges to overcome. Therefore, the *home projects* can be taken by group of students accordingly to the difficulty of the project. It is desirable for each group to choose a second programming language (`C/C++`, `Python`, ...) for the development, complementing their studies. The projects shall be presented as a short report describing the problem, computational approach, code developed, results and conclusions. Preferably, the text should be written in LaTeX to be easily attached to this notes.

Text-books. Complementing these notes, complete discussions on the proposed topics can be found on the books available at UFU's library[1, 2, 3, 4, 5, 6], and other references presented throughout the notes.

Please check for an updated version of these notes at `www.infis.ufu.br/gerson`.

[1]Debian: `www.debian.org`, Ubuntu: `www.ubuntu.com`.
[2]Julia: `www.julialang.com`.
[3]gnuplot: `www.gnuplot.info`.
[4]Asymptote: `asymptote.sourceforge.net`.

CHAPTER 1

Introduction to the Julia language

`Hello world!`

JULIA is a recently developed (~ 2012) high-level computer language designed for high-performance numerical and scientific computing. The language has many advanced features that we will not cover on this introductory class. Please check the official website[1] and the Wikipedia[2] for more information. Here we will focus numerical calculus, Fourier transforms, systems of equations, eigenvalue problems, random number generators and statistics, and plotting figures, which constitutes the core of an introduction to scientific computing.

Keep in mind that Julia is a high-level language. Its performance is typically worse than low-level languages like C by a factor of two. Still, this is an impressive performance for a high-level language. Therefore Julia combines the easy-to-use features of high-level languages like Matlab with the performance of low-level languages. Add to this the fact that Julia is open-source and has a variety of available and useful packages (*e.g.:* ODE, PyPlot) for beginners and experts, and Julia becomes the ideal language for an introductory class.

In this Chapter we will cover only the fundamental aspects of the Julia language as a introduction for the **students with zero or little experience** with computer languages. For the more experienced student, I suggest you check the Tutorials on the Learning section of Julia's webpage.

1.1 Installing Julia (needs update)

1.1.1 JuliaBox

You don't even need to install Julia! `JuliaBox`[3] is an online server that allows you to run your Julia codes on a remote machine hosted by the the `Amazon WebServices`. To access this server you'll need a Google account. Once you are in, there will be an iJulia

[1]Julia: `www.julialang.org`
[2]Wikipedia: `https://en.wikipedia.org/wiki/Julia_(programming_language)`
[3]JuliaBox: `www.juliabox.org`

notebook interface running on Jupiter (iPython). You can easily upload, download and edit your codes. Moreover, you can sync your files with Google Drive or GIT.

In JuliaBox you can run your code on the Amazon server for free up to six hours. After that your virtual machine will be shut down and you have to login again.

Unfortunately, my personal experience with JuliaBox is that it is not stable yet. It's still beta. The kernel seems to die often, but it could be a problem with my network (proxy?). Despite this, I encourage you to try it. Once this is stable, it will be a great tool.

MS Windows and Apple OS X

I highly recommend you to use `Linux`. However, if you are stubborn enough I allow you to waste time with another OS. For MS Windows and Apple OS X (at least this one is UNIX-based) the installation follows usual procedures. Please visit the Download section on Julia's webpage to find the newest version of the installation files and instructions.

Debian/Ubuntu/Fedora Linux

You can also find the installation files for Linux distributions on the Downloads section of Julia's webpage. However, one of the great advantages of using Linux is the advanced packaging system that helps you keep your software always up to date. Many third-party applications will have PPA repositories for Ubuntu Linux. So here I'll quick teach you how to use them.

If you are new to Linux. Try Ubuntu. Their PPA repositories are great.

Ubuntu Linux

Julia's PPA repository [4] is `ppa:staticfloat/juliareleases`. To add it to your list of repositories and install Julia, please open the Terminal and follow the commands below. Please read the output of each command as it may request you to accept the changes.

Example 1.1: Installing Julia in Ubuntu Linux

```
sudo apt-add-repository ppa:staticfloat/juliareleases
sudo apt-get update # this command updates the package list
sudo apt-get install julia
```

Depending on the Ubuntu version you are using, you may also need to add the PPA for Julia's dependent packages: `ppa:staticfloat/julia-deps`.

That is it! If all goes well, Julia is already installed. If there is a new version and you want to upgrade, go again to the terminal and run:

[4]Julia's PPA: https://launchpad.net/~staticfloat/+archive/ubuntu/juliareleases

Example 1.2: Upgrading your installed software

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

As a suggestion for the new Linux users, there are a few graphical interfaces for the APT packaging system. I recommend you to use `Synaptic`. If it is not already installed, simply update your package list as in the examples above, and run `sudo apt-get install synaptic`.

If you are new to Linux, get ready to be comfortable using the Terminal as it makes your life much easier. For instance, many of the commands above are probably new to you. To learn more about them, you can check the `man-pages` on the Terminal. Please check and read the content of the commands below (to exit press `q`).

Example 1.3: Using the man-pages

```
$ man sudo # execute a command as another user
$ man apt-get # APT package handling utility -- command-line interface
$ man julia # launch a Julia session
$ man synaptic # graphical management of software packages
$ man man # an interface to the on-line reference manuals
$ man bash # GNU Bourne-Again SHell
```

Debian Linux

Unfortunately you cannot find Debian packages of Julia in their main webpage. In Debian Jessie (stable) repository you will find an old version of Julia (0.3.2 the last time I've checked). To get the newest version of Julia, you will need Debian Stretch (testing) or Sid (unstable). It could be tricky to install a Stretch package in Jessie, so I don't recommend this. If you use Debian Jessie, the best choice is to use the Generic Linux binaries from Julia's webpage Download section.

Here's my suggestion on how to install the **Generic Linux Binaries**:

First go to Julia's webpage Download section and get the file for the 64bit or 32bit version, depending on your computer architecture. The file name will be something like `julia-0.4.3-linux-x86_64.tar.gz`. The `.tar.gz` extension indicates a compressed file (like a zip file). You may save it to your `Downloads` folder.

Open the Terminal. Next we will extract the files from the `.tar.gz` file and move it to a better location. Follow these commands:

Example 1.4: Julia: manual installation, part 1

```
# Change Directiory to where you have saved the .tar.gz file:
$ cd Downloads
# extract the contents:
$ tar -xzf julia-0.4.3-linux-x86_64.tar.gz
# list the files within the current directory:
$ ls
```

The last command, `ls`, will list the files on the directory. Among other personal files of yours, you'll see the `.tar.gz` file and a new folder with a name similar to `julia-a2f713dea5`. Let's now move it to a better location, back to the Terminal:

Example 1.5: Julia: manual installation, part 2

```
# let's move it to a different place. The dot makes it a hidden folder
$ mv julia-a2f713dea5 ~/.julia

# open the .bashrc with your favorite text editor
$ gedit ~/.bashrc

# to add Julia's binary to the path where the system looks for binaries,
# add the following line at the end of the file
export PATH=$PATH:$HOME/.julia/bin
# save the file and close the text editor
```

Great! If all goes well, close your Terminal and open it again. Every time the Terminal (BASH) starts, it runs the content of `.bashrc`. The line we have added to this file tell the system to search Julia's folder for binaries. Therefore, now you can start Julia in bash running `$ julia`.

1.1.2 Juno IDE

Juno[5] is a powerful IDE (Integrated Development Environment) built specifically for Julia. Please check its webpage for installation instructions for MS Windows, Apple OS X, and Linux. Unfortunately, up to now there's no official repository for Juno on Linux, so you have to install it manually.

1.1.3 iJulia

iJulia[6] provides the Jupiter interactive environment for Julia (similar to iPhyton). This is equivalent to the interface of JuliaBox, but this one runs locally on your computer. To install it, go to the Terminal, start Julia's command line interface and run:

[5]Juno: `junolab.org`
[6]iJulia: `github.com/JuliaLang/IJulia.jl`

Example 1.6: Installing iJulia

```
 First, open the Terminal and start Julia
$ julia
 Next, install iJulia with the Pkg.add command
julia> Pkg.add("IJulia")
```

To start and use iJulia, follow these commands:

Example 1.7: Using iJulia

```
 First, open the Terminal and start Julia
$ julia
 Next, start iJulia
julia> using IJulia; notebook();
```

The commands above will start the iJulia server, which you have to access via you Internet browser at `127.0.0.1:8888`.

1.2 Trying Julia for the first time

First of all, I'm old fashion. So I prefer to run my codes on the command line interface, while I edit my code using Emacs[7]. Emacs can be complicated for a beginner, while it becomes a incredible tool as you get experience. For now, you can use simpler text editors like `gEdit` or `Kate`. Of course, if you are using Juno, JuliaBox, or iJulia interface, you won't need a text editor. But first let's learn how to get around the command line interface, and later on we'll start with the codes and scripts.

1.2.1 The command line interface

Open the Terminal and start Julia.

Example 1.8: Julia's initial screen

```
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.4.2 (2015-12-06 21:47 UTC)
 _/ |\__'_|_|_|\__'_|  |  Official http://julialang.org release
|__/                   |  x86_64-linux-gnu

julia>
```

[7]Emacs: http://www.gnu.org/software/emacs/tour/.
To install Emacs on Ubuntu: `sudo apt-get install emacs`.

The first thing you see is a message on how to ask for help. Very encouraging! Since we don't know anything, let's ask for help. Follow the instruction on the screen. Type "`?help`" for help. Great! From the command line there's always an easy access to a short documentation that you can use to check what an specific Julia command does and what parameters does it take.

Try some simple commands:

Example 1.9: Simple commands

```
julia> 2+2
julia> sin(30)
julia> sin(30*pi/180)
julia> sin(30pi/180)
julia> pi
julia> a = 2^4;
julia> a
julia> println("The value of a is ", a);
julia> ?println
```

Above you can see that Julia (as any other computing language) takes angles in radians, there's a pre-defined constant for π, the exponentiation is done with the circumflex symbol (x^y = `x^y`), you can assign value to variables with =, print text and values on the screen with `println`, and if you finish a line with a semicolon (`;`), it suppresses the output. That's a good start.

In the examples above, note that you don't need to type the multiplication symbol $*$ between numbers and variables. Julia understands the implied multiplication. This is a nice trick to keep your expressions short and easy to read.

Let's try another basic set of commands:

Example 1.10: Functions and memory use

```
julia> a = 0; # defines and attributes value to the variable
julia> f(x,a) = exp(-x/a)sin(x); # defines $f(x,a) = e^{-x/a}\sin(x)$
julia> f(1,1) # calculates the function for the specific values
julia> whos() # prints the memory use
julia> workspace(); # cleans the user-defined varibles
julia> f(1,1) # gives an error, since f(x,a) is not defined anymore...
julia> whos() # ... as you can see here.
```

Here we are defining a function `f(x,a)` that takes two parameters. Note that even tough we have set the variable `a=0` on the previous line, the function definition does not use this value, since it belongs to a different scope.

The commands `whos()` and `workspace()` are very helpful to keep track of memory use, and to clean all user-defined values and start again when needed.

Essential commands, auto-complete, and command history

Here's a list of other essential commands to keep in mind:

- Everything that follows # is a comment and it is ignored;
- To exit Julia's command line interface: `quit()`, or `exit()`, or press `CTRL+D` (= `^D`). As usual, let's use the circumflex symbol ^ to refer to `CTRL` key for short notation.
- To interrupt a computation, press `^C`;
- Press `;` at an empty line to go back to `BASH` temporally.
- To run a script file, run `include("file.jl")`. Always use the `.jl` extension so your text editor recognizes that your are writing a Julia script;
- Use the `TAB` key to complete what you are typing (or a list of suggestions). Try to type `"in"` and press `TAB` twice, you'll see a list of possible commands that start with `"in"`. Continue the command pressing `c` to form `"inc"` and press `TAB` again. It's magic!
- You can go through your **commands history** pressing the up and down arrows of your keyboard. Also, you can search the command history pressing `^R` and typing any part of the command you want to recall.

1.2.2 Using scripts

The command line interface is great for quick calculations, test codes, plot figures, etc. However, in practice we should always have our code stored in a file somewhere. This is the script. While in low-level compiled languages we refer to the codes as the "source code", for high-level interpreted languages, it is called a "script".

There's no secret here. Through the classes we will learn Julia commands and structures, try some codes and solve some problems. A script file is just a text file with the sequence of commands you want to run. Julia scripts should end with a `.jl` extension, so that your script editor recognizes it is a Julia script. In Linux, I recommend the beginners to use one of these editors: Kate or gEdit. The first is part of the KDE ensemble, while the second belongs to Gnome. For those with more experience, try to learn Emacs.

In the next section we will start to write more complicated functions, so I invite you to code them in a script file instead of the command line as we were doing so far. Simply open your favorite editor, type the code, and save it with the `.jl` extension. Let's say your file is `script.jl`. There's two ways to run the script:

1. In the Terminal, start Julia and run: `julia> include("script.jl");`
2. In the Terminal, do not start Julia! From the shell, run: `$ julia script.jl`

What's the difference between these two cases?

1.2.3 Other interfaces

During the classes we will use mostly the command line interface and scripts. Therefore I will not cover here a guide to use the other possible interfaces: JuliaBox, Juno IDE, and iJulia. These are easy-to-use interfaces, so after learning about the command line and scripts, you will probably have no difficult using the others.

1.3 Constants and variable types

Julia comes with a few predefined mathematical constants:

- π = `pi` = 3.1415926535897...;
- e = `e` or `eu` = 2.7182818284590..., Euler's number, base for the natural logarithm;
- For complex numbers, use `im` for the imaginary part. For example: `z = 5 + 3im`;
- True and false constants are `true=1`, and `false=0`;

The variable types in Julia are dynamic. This means that you don't have to declare it beforehand. For instance:

Example 1.11: Variable types

```
julia> a = 8
julia> typeof(a) # will tell you that "a" is 64bit integer
julia> sizeof(a) # used memory in bits
julia> a = 8.0
julia> typeof(a) # now "a" is a 64bit floating-point number
julia> sizeof(a)
julia> a = 8.0 + 3.0im
julia> typeof(a) # is a complex floating-point number
julia> sizeof(a)
julia> a = 8 + 3im
julia> typeof(a) # is an integer floating-point number
julia> sizeof(a)
julia> a = true
julia> typeof(a) # is a Boolean
julia> sizeof(a)
```

The difference between the types is how the variable is stored in memory. A floating-point number is the computer version of the Real numbers[8]. `Float64` means that Julia is using the double precision format (64 bits), which uses one bit for the sign, 11 bits for the exponent, and 52 bits for the fraction. Since 1 byte = 8 bits, each `Float64` variable will need 8 bytes memory. For instance, if you have a 100×100 real matrix, there will

[8]For more details, check Wikipedia's entry on the double precision (64bits) floating-point numbers: https://en.wikipedia.org/wiki/Double-precision_floating-point_format, and a general discussion on floating-point numbers https://en.wikipedia.org/wiki/Floating_point.

be 100^2 `Float64` numbers to store, consuming 80.000 bytes = 80 kB. You can check the memory use of any variable with the command `sizeof(...)`. Evidently, a 64 bit integer, *e.g.* Julia's `Int64`, also needs 8 bytes of memory. So, what's the advantage of having both `Float64` and `Int64` types of variables? I'll leave this to you as a problem.

Note that the complex variable takes twice the memory of a real or integer variable. It has to store both the real and imaginary part. Boolean variables uses only a single bit! Since it can be only true or false, a single bit suffices.

1.3.1 Assertion and Function Overloading

Type assertion is particularly useful when defining functions. It can be used to assure that the functions receives an argument of a certain type. For instance, let's define a function that calculates the factorial using a loop:

Example 1.12: Factorial function with assertion

```
function fact(n::Int64)
   res = 1; # initialize the result as 1
   for i=n:-1:1 # loop the variable i from N to 1 in steps of -1
       res = res*i; # since n! = n·(n−1)·(n−2)···(1)
   end # end loop
   return res; # return the result
end
```

In the first line the argument of the function `fact(n::Int64)` is declared as an `Int64`. If you try to call this function as `fact(4)`, you will get the correct result, 24. But if you call `fact(4.0)`, it fails. The function is not defined for an arbitrary argument.

If you have a function that for different types of parameters may take different forms, you can use **function overloading**. This is an extremely useful feature of modern computer languages. For instance, the factorial of a real number can be defined via the Riemann's Γ function,

$$\Gamma(n) = (n-1)!, \quad \text{for } n \in \mathbb{Z}^+, \tag{1.1}$$

$$\Gamma(t) = \int_0^\infty x^{t-1} e^{-x} dx, \quad \text{for } t \in \mathbb{R}. \tag{1.2}$$

We, humans, use the same letter Γ for these functions. But the expression used to calculate the result depends on the parameter being integer or real. Function overloading is the computer language way of making the decision between the two forms of Γ above. Check the next example.

Example 1.13: Function Overloading

```julia
function Gamma(n::Int64)
   return fact(n-1);
end

function Gamma(t::Float64)
   println("We'll learn how to calculate numerical integrals later...");
end
```

Try to define all these functions in Julia. Simply copy and paste the code from here. Now, if you call `Gamma(5)` the result will be 24 again. But for `Gamma(5.0)` you will get the message above from `println`. Simply call `Gamma` with no parameters or parenthesis, and you see the message "`Gamma (generic function with 2 methods)`". The different methods are the different implementations of `Gamma` with distinct type or number of parameters.

Please note that this is a simple illustrative example. In practice Julia has built-in functions to calculate both the factorial and Riemann's Γ. Built-in are always more efficient.

1.3.2 Composite Types (struct)

Composite types are equivalent to the `structs` in C language. It is usually useful when you want to store different information about an object in a single variable.

Since we are taking a Computation Physics class... let's exemplify the composite types as a way to store the position and momenta of particles.

Example 1.14: Composite Types: particle position and momenta

Say that you need to store the position (x, y) and momenta (p_x, p_y) of all habitants of flatland[7]. We can define a type of variable that we choose to name "Particle":

```julia
type Particle
  x::Float64
  y::Float64
  px::Float64
  py::Float64
end
```

Now we can create many particles calling

```julia
q1 = Particle(0.0, 0.0, 0.0, 0.0); # resting at origin
q2 = Particle(10.0, 10.0, -1.0, -1.0); # in a collision course with q1
```

To avoid the collision, let's update the position of particle q_1:

```julia
q1.x = 1; # now q1 is still at rest, but now on (x,y) = (1,0)
```

In the example above we have defined a new type of variable (`Particle`), and we create two variables of this type representing particles q_1 and q_2. The parameters (position and momenta) are the “fields” of this new variable type. The parameters passed to these variables when defining q_1 and q_2 are called the “initialization”. Once the variables exist, you can access and modify its contents using the construct `variable.field`, as in the example where we update the position of particle q_1.

This is a silly example, but eventually we will use this to help us solve the motion of planets using Newtonian dynamics.

Instead of the declaring the composite type using the `type` keyword, you could also work with the `immutable`, *i.e.* `immutable Particle ··· end`. Immutable types work as constants, they cannot be modified. The main difference between “mutable” and immutable types is that the first is passed by reference to a function, while the second is passed by copy[9].

1.3.3 Tuples

In mathematics, a tuple is a ordered list of elements. For instance, a vector $\boldsymbol{r}$ can be written as 3-tuple: (x, y, z). Julia's documentation describes this type of data as an abstraction of the arguments of a function.

```
Example 1.15: Tuples: square root

function squareroot(x::Number)
   return (+sqrt(x), -sqrt(x)) # returns a tuple
end

sols = squareroot(9);

println("The square roots are ", sols[1], " and ", sols[2]);

println("The number of square roots is ", length(sols));

sols[1] = 0; # will return an error
```

In the example above, while the `sqrt(···)` native command returns only the positive square root, our example function returns a tuple with all square roots. The tuple is stored in the variable `sols`, and each value can be accessed via the construct `sols[1]`, `sols[2]`, ··· `sols[i]` ··· `sols[n]`, where the number of elements in the tuple can be checked with the `length` command. In the simple example above, we will always have two entries in the tuple. However, some codes may return a tuple of an arbitrary number of entries. In this case you may use the command `length` to check the number of elements.

In Julia, a tuple is an immutable type: you cannot change its contents, and it will be passed to a function by copy. Therefore tuples are appropriated when you need to pass

[9] If you are a beginner, you probably don't know what “passing by reference/copy” means. Don't worry, we'll get to that.

vector or matrix parameters by copy, instead of by reference. Apart from this, they are equivalent to arrays.

1.3.4 Arrays: vectors and matrices

As we have seen above, mathematically and array is a tuple. In Julia an array is a mutable tuple. To distinguish these two, tuples are defined by parenthesis as `pos = (x,y)`, while arrays are set by square brackets as `pos = [x, y]`. Essentially, both tuples and arrays could be used to represent vectors and matrices. However, when we are working with vectors and matrices, we will probably need to update their values along the calculation. Therefore we need a mutable type: the array.

Let's define a vector and a matrices and try some common operations.

Example 1.16: Arrays: vectors and matrices

```
veclin = [1 2 3] # this is a line vector (1×3)
veccol = [1; 2; 3] # this is a column vector (3×1)
matA = [4 5 6; 7 8 9; 10 11 12] # a 3×3 matrix
matB = [3 2 1; 6 5 4; 9 8 7] # another 3×3 matrix

matA*veccol # (3×3)·(3×1) = (3×1)

veclin.' # transpose the vector
veccol' # conjugate transpose the vector

matA' # conjugate transpose the matrix

inv(matA) # calculates the inverse

matA*matB # mathematical product of the matrices

matA.*matB # direct element by element product
```

Run the example above line by line and check the outputs to understand what each command does. You will see that there's no difference between the conjugate and the transpose conjugate here, because we are dealing with real values. Try defining a complex matrix or vector and check again the difference between these two commands.

There's a lot more you can do with vectors and matrices of course. We'll learn more tricks as we go along. For instance, a nice trick to construct matrices is the `comprehension` construct that we present in Section 1.4.5. Other relevant commands that we may use later on are:

- `zeros(n)`: creates an array of zeros with n elements;
- `zeros(n,p)`: creates a $n \times p$ matrix with all elements set to zero;
- `ones(n)`, and `ones(n,p)`: equivalent to `zeros`, but all elements are set to 1;

- `eye(n)`: creates the identity matrix of order n. This is different than `ones(n,n)`;
- `rand(n)`, and `rand(n,p)`: array or matrix of random values uniformly distributed between $[0,1)$;
- `randn(n), and randn(n,p)`: equivalent to `rand`, but with a standard normal distribution (mean value = 0, and standard deviation = 1);
- `linspace(start, stop, n)`: creates a range from `start` to `stop` with `n` elements. To convert the range into an array, use the `collect` command;
- `norm(v)`: returns the norm of vector v;
- `cross(x,y)`: calculates the cross product $x \times y$;
- `diagm(v[, k])`: constructs a matrix with the k-th diagonal set to the vector v and all other elements to zero;
- `trace(M)`: returns the trace of the matrix M;
- `det(M)`: returns the determinant of the matrix M;
- `inv(M)`: inverse of the matrix M;
- `eig(M)`: calculate the eigenvalues and eigenvectors of the matrix M;

Indexing an array or matrix

To access or change the value of a matrix or array element, use `matA[i,j]` and `veclin[i]`, respectively. The first access the element (i, j) of matrix `matA`, the second access the element i of vector `veclin` from the example above. You may also use ranges to access multiple elements. Check the example:

Example 1.17: Indexing arrays and matrices

```
# creates a random 6×6 matrix. Check the output
M = randn(6,6)

M[2,3] # returns element of the 2nd row, 3rd column

M[2:4, 5] # returns rows 2 to 4 in column 5

M[:, 2] # returns column 2

M[3, :] # returns row 3

M[1, end] # returns element in the last column of row 1
```

Concatenation

Array concatenation in Julia follow the vector and matrix constructions as shown in the example below. Additionally, one may also explicitly call the `*cat`-functions: `cat`, `vcat`, `hcat`, `hvcat`.

In the next example we start with a vector x and add more elements to it, either increasing its size, or transforming it into a matrix. Please check carefully the outputs of each line to understand the different types of concatenations.

Example 1.18: Concatenation

```julia
x = rand(3); # initialized as a column vector 3x1

# repets x and adds two more elements, making it 5x1
y = [x; 10; 20];

# the line above is equivalent to the vcat: vertical concatenation
y = vcat(x, [10; 20]);

# creates another 5x1 vector
z = rand(5);

# creates a 5x2 matrix setting y as the first column and z as the second
m = [y z];

# the line above is equivalent to hcat: horizontal concatenation
m = hcat(y, z);
```

Keep in mind that concatenation can be useful to store results in a list of initially unknown size. You can concatenate new results of some calculation into the list as needed. However, this is not efficient for large vectors, in which case it is better to known the size of the vector and initialize it beforehand.

In the next example we calculate the average and standard deviation of a list of random numbers as a function of the number of random values sampled. This is not the most efficient way to calculate and store the results, but exemplifies an interesting way of using concatenation.

— Example 1.19: Concatenation to store results on demand —

```julia
# using the PyPlot package that will be introduced in Section 7.1
using PyPlot

# initialize empty arrays of Float64 to store the resuts
average = Float64[];
stddev  = Float64[];

n = 1; # initialize counter of sampled random numbers
# the average of a uniform random distribution should be 0.5
# let's run the loop until we reach this limit
while length(average)==0  abs(average[end]-0.5) > 1e-6
  n += 1; # increment the counter
  list = rand(n); # sample a new list
  average = [average; mean(list)]; # concatente new average to the list
  stddev  = [stddev;  std(list)]; # and the same for the std. deviation
end

# plot the average and standard deviation as a function of
# the number of sampled points
subplot(211)
plot(average)

subplot(212)
plot(stddev)
```

1.3.5 Scope of a variable

The scope of a variable refers to the region of the code where the variable is accessible. You may, for instance, define two functions of x, say $f(x)$ and $g(x)$. Here x is suppose to be a free variable, being assigned to a value only when you call the functions; *e.g.* if you call $f(3)$ the code inside the function $f(x)$ uses $x = 3$. From a mathematical point of view this is quite obvious, right?

What about for a computer? Is it obvious? Remember that a computer does whatever you code him to do. Therefore the definition of the scope of a variable is extremely important. Otherwise you and the computer could have some misunderstandings. Check the example:

Example 1.20: Scope of a variable

```
x = 3; # global definition of x

# definition of f(x): here x is local. The scope is the function.
f(x) = exp(-x)sin(x);

# definition of g(y): here x takes the global value, y is local.
g(y) = sin(x*y);

# definition of h(y): x, dx, and ans are local variables
function h(y)
  x = 0;
  dx = 0.01;
  res = 0;
  while x <= y
     res = res + x*dx;
     x = x + dx;
  end
  return res;
end

x # check the value of x

f(0) # will return 0
f(1) # will return 0.3095598756531122
f(x) # will return 0.00702595148935012
h(2) # will return 1.99

x # still has the same value. Calling h(2) didn't change it.

g(10*pi/180) # will return 0.49999999999999994

x = pi/180;

f(x) # now returns 0.017150447238797207
g(30) # returns 0.49999999999999994
h(2) # still returns 1.99
```

Do you understand all outputs of the example above? Please try to follow what happens at each line, and what value of x is used. The variable x is first initialized in the global scope ($x = 3$). However, when we define a function, its arguments are threated as local variables. Therefore the variable x that appears in $f(x)$ is not the same as the global x. It has the same name, but refers to a different scope. Check Problem 1.4.

Besides the functions, new scopes are initiated every time you start a new code block; *e.g.* `for` and `while` loops.

1.4 Control Flow: if, for, while and comprehensions

In a vector oriented language one should always try to avoid using loops (*e.g.:* `for`, `while`). Also, it's always better to avoid conditional evaluations (`if`). But it is not always possible. So let's check how to use them in Julia.

1.4.1 Conditional evaluations: if

The definition of the `if` construction follows the common `if-elseif-else` syntax:

```
if x < y
  println("x is less than y")
elseif x > y
  println("x is greater than y")
else
  println("x is equal to y")
end
```

As usual, it checks if the first statement is true (`x < y`), if so, it runs the first block. Otherwise it goes to the second condition (`elseif`). If all conditions are false, it runs the `else` block. You may have many `elseif` conditions if you need it... or none. Both `elseif` and `else` blocks are optional.

One can also use Short-Circuits (AND: `&&`, OR: `||`). The `&&` is the boolean AND operator. For instance, `(testA && testB)` will return TRUE only if both `testA` **and** `testB` are TRUE, and it returns FALSE otherwise. For efficiency it first evaluate `testA`, if it returns FALSE, Julia does not have to evaluate `testB`. Do you agree?

What about the OR operator `||`? What is the outcome of `(testA || testB)` for different results of `testA` and `textB`?

Write and run the following code. It uses a `comprehension` to create a matrix. We'll learn about `comprehensions` later, but you can probably guess what it does just by reading the code. Try to understand the content of the matrices `Tand` and `Tor`.

Example 1.21: Truth tables

```
tf = [true; false];

Tand = [ x && y for x in tf, y in tf ]

Tor = [ x || y for x in tf, y in tf ]
```

1.4.2 Ternary operator ?:

The ternary operator (`a ? b : c`) is simply a short syntax for `if` statements that can be put in a single line. Here `a` is the boolean test, `b` is the code to run if `a` is TRUE, and `c` is the code if `a` is FALSE. For instance, the next example shows two different implementation of a function that calculates the absolute value of *x*.

Example 1.22: Ternary operator: absolute value

```
function absolute(x)
  if x >= 0
    return x
  else
    return -x
  end
end

ternaryabsolute(x) = (x>=0) ? x : -x;
```

1.4.3 Numeric Comparisons

Comparison operators follow standard notation:

- Equality `a == b`, tests if `a` is equal to `b`;
- Inequality `a != b`, tests if `a` and `b` are different;
- Less than: `a < b`;
- Less than or equal to: `a <= b`;
- Greater than: `a > b`;
- Greater than or equal to: `a >= b`;

It's possible to compare numbers to infinity (`Inf` $= \infty$), or undefined results (`NaN` = not a number). For instance, `(1/0 == Inf)` returns TRUE, and `(0/0 == NaN)` also returns TRUE. Other comparisons can be made with the help of predefined functions:

- `iseven(a)`, returns TRUE if `a` is even;
- `isodd(a)`, returns FALSE if `a` is odd;
- Other useful tests: `isinteger(a)`, `isnumber(a)`, `isprime(a)`, `isreal(a)`, `isfinite(a)`, ...

Essentially, every predefined function that starts with `is...` is a boolean test. To find others, open the command line interface, type `?is` and press TAB twice.

1.4.4 For and While Loops

In Julia you can create repeated evaluations using `for` and `while` block codes. They are both quite intuitive, but let's take a chance to introduce some new commands frequently used together with loops.

The `while` loop simply runs the block **while** the test remains true. For instance:

Example 1.23: While loop

```
n = 0; # initialize n
while n <= 10
  println("Running for n=", n);
  n += 2; # update n, incrasing by 2
end
```

When we use `while`, usually a variable used for the stop condition must be initialized outside, and updated inside the block. Here we initialize with `n=0`, and update it incrementing as `n += 2`, which is a short notation for `n = n + 2`.

The code above could be better written with a `for` statement:

Example 1.24: For loop

```
for n=0:2:10
  println("Running for n=", n);
end
```

Here we use the colon operator `(start:step:end)` to initialize $n = 0$, update its value in steps of 2 until it reaches 10. We have used this form in the implementation of the factorial function in Example 1.12.

You may also use `for` to run through lists. The example is self-explanatory:

Example 1.25: For each in list

```
# using an array to store a list of values
numericlist = [ 1, 10, 2, 8, 20, 0 ];

for each in numericlist
  println("each = ", each);
end

stringlist = [ "this", "may", "also", "be text" ];

for txt in stringlist
  println(txt);
end
```

1.4.5 Comprehensions

Comprehensions use the structure of the `for` loop to easily create arrays. For instance, Example 1.21 uses comprehensions to create the Truth Tables. The general form is

```
A = [ f(x,y) for x=rangeX, y=rangeY ];
A = [ f(x,y) for x in listX, y in listY ];
```

The first variable after the `for` refers to the line, and the second to the column. It is possible to go further into multi-dimensional arrays (tensors), but let's keep it simple for now. The extension to n-dimensions is evident.

1.5 Input and Output

Input and Output (I/O) is essential for any code. You will almost always need to save your data on a file, and sometime you need to read some data from somewhere. The basic commands to read files are: `readdlm` and `readcsv`. To write a file: `writedlm` and `writecsv`. Check their help-description in Julia.

Here `...dlm` stands for "delimited", while `...csv` is "comma-separated values".

Let's first assume you have your data stored in the file "foo.dat" as a matrix, such each line of the files correspond to a line of the matrix, and the matrix columns are separated by white spaces. You can read it running: `julia> a = readdlm("foo.dat")`. The data will be stored in the matrix (array) `a`.

Example 1.26: Reading and writing files

Using your favorite text editor, create a data file "foo.dat" and fill with a 5×5 numerical matrix of your choice. This matrix may have integers or real numbers. Separate the numbers in each line using spaces. Now, in Julia's command line interface, run:

```
julia> data = readdlm("foo.dat")
```

You should see something like this:

```
julia> data = readdlm("foo.dat")
5x5 Array{Float64,2}:
  1.0   2.0   3.0   4.0   5.0
  6.0   7.0   8.0   8.0   9.0
 10.0  11.0  12.3  13.0  14.0
 15.0  16.0  17.3  18.2  19.0
 20.1  21.0  22.3  24.0  25.0
```

Now the data is stored in the variable `data`. Let's multiply this data by 2, and save it in a different file. But now the columns will be separated by commas.

```
julia> writedlm("bar.dat", 2*data, ',');
```

Open the file "bar.dat" in the text editor to see the result. This is the format of a CSV file. You would get the same result running:

```
julia> writecsv("bar.dat", 2*data)
```

1.6 Other relevant topics

1.6.1 Passing parameters by reference or by copy (not finished)

1.6.2 Operator Precedence

The operator precedence rules define the order that operations are evaluated. For instance, what is the result of $2+3\times 4$? You are probably getting 14 and not 20, right? You know that you should multiply 3×4 first, and them sum 2. How do you know that? Someone has told you that multiplication takes precedence over addition. If we want to add 2 and 3 first, and then multiply the result by 4, you should write $(2+3)\times 4$. This one give us 20.

Sometimes it might be difficult to remember the precedence rules. For instance, what's the result of $(10/2\times 5)$? Is it 1 or 25? To avoid problems, it is always better to use parenthesis: $(10/(2\times 5))$ or $((10/2)\times 5)$.

In Julia, the most common operations follow this order of precedence

1. Exponentiation ^ or the elementwise equivalent .^
2. Multiplication $*$ and division / or the elementwise equivalents $.*$ and ./
3. Addition $+$ and $-$
4. Comparisons: $> \quad < \quad >= \quad <= \quad == \quad !=$

For more details and the full list of operations, please check Julia's documentation.

1.7 Questions from the students

Here I'll list some questions from the students that I couldn't answer during the class.

Question [1] How to read a data file that has complex numbers?

Answer: The `readdlm` command will return a string `"3+1im"` instead of the complex number `3+1im`. In this case the data file will be read as type `Any`. Below I wrote two codes to overcome this.

In the first code I read the data as `ASCIIString` and use `parse` and `eval` to evaluate the expression in Julia as if it were a code. For instance, if one entry of the data file is `"2+2"`, it will actually be evaluated to 4. Here's the code:

```
data = map(x->eval(parse(x)), readdlm("data.dat", ASCIIString));
data = convert(Array{Complex{Float64},2}, data);
```

The problem with the code above is that it first reads everything as Strings. A second choice would be:

```
myparse(x) = (typeof(x) == SubString{ASCIIString}) ? eval(parse(x)) : x;
data = map(x->myparse(x), readdlm("data.dat"));
data = convert(Array{Complex{Float64},2}, data);
```

This version reads the data as type `Any` and uses the user-defined function `myparse(x)` to convert to a number only the Strings.

However, a better choice would be to save the real and imaginary parts separated in the first place. If your data is a n-columns file, save real part in one column and the imaginary in the next. If you are saving matrices, save the real and imaginary in different files. This way is probably more efficient, since for large files the `parse(x)` command will probably be too slow.

Question [2] How to choose the line color in PyPlot?

Answer: Please check the Python and Matplotlib installation. The simple code tested in class should work:

```
using PyPlot
x = linspace(0,2pi,50);
plot(x, sin(x), color="red", linewidth=2.0, linestyle="-")
plot(x, cos(x), color="blue", linewidth=2.0, linestyle="--")
```

Please check the updated PyPlot installation section above.

Question [3] What's the difference between Tuples and Arrays?

Answer: Tuples are immutable and array are mutable. The example in the text and problem proposed below do not explore this difference. I'll replace them with a better discussion soon. The main difference is that a Tuple will be passed to functions by copy, while arrays are passed by reference. But I still have to confirm this with some example.

1.8 Problems

Problem 1.1: `Int64` and `Float64`

If both formats `Int64` and `Float64` use the same amount of memory, why having both defined instead of always using `Float64`? Check, for instance, Ref. [2] and the Wikipedia pages mentioned earlier.

Problem 1.2: Bhaskara

Write a function named `bhaskara` that receives three parameters `a`, `b` and `c` representing the coefficients of the quadratic polynomial equation $ax^2 + bx + c = 0$. Calculate the roots `r1` and `r2`, returning them as a **Tuple**. Try the code below to test your implementation:

```
julia> root1, root2 = bhaskara(2.0, -2.0, -12.0);
julia> println("The first root is ", root1)
The first root is 3.0
julia> println("The second root is ", root2)
The second root is -2.0
```

Problem 1.3: Scripts

Write a code of your choice as a script (could be one of the examples or problems here). Try to run it using both methods described in Section 1.2.2. What is the difference between these two methods?

Problem 1.4: Scope of a variable

a) Go back to Example 1.20. Save it as a script file. Run it and check that the values match the numbers in the example. Can you explain the outputs?

b) In the first line of the definition of the function $h(y)$

replace: `x = 0`

with: `global x = 0`

Run the script again. Explain the new output of each line.

c) Still on $h(y)$, what happens if you also initialize the variable `res` as global?

Problem 1.5: Conditional evaluations

a) Write a code to represent the following function:

$$\psi(x) = \begin{cases} 1 & \text{if } x < -a \\ (x/a)^2 & \text{if } -a \le x \le +a \\ 1 & \text{if } x > +a \end{cases} \tag{1.3}$$

b) Use `linspace` to create a range of 100 values for x between `-2a` and `2a`. Calculate $\psi(x)$ over this range using your function. Use `PyPlot` or any other plotting package to plot your function.

Problem 1.6: While and For Loops

Rewrite the factorial code of Example 1.12 using a `while` loop instead of a `for`.

Problem 1.7: Input and Output

a) Go back to Example 1.26. Write a code to read the CSV data in "bar.dat", and save it as "semicolon.dat" using semicolon (;) as a separator.

b) Write a code to read the "semicolon.dat", and save it now as "text.dat" with the data type set to `String`.

Problem 1.8: Random numbers and histograms

Write a code to generate a large set of random numbers (start with 1000) and use the `hist(...)` command to generate a histogram, and use `PyPlot` to plot it. The output of `hist(...)` is tricky. Pay attention to it and check Julia's documentation.

a) Use the `rand(...)` function to get an uniform distribution between $[0,1)$;

b) Use the `randn(...)` function to get a standard normal distribution.

Problem 1.9: Cross and dot products

Given a parallelepiped defined by the vectors $\boldsymbol{a} = (1,0,0)$, $\boldsymbol{b} = (0,1,0)$ and $\boldsymbol{c} = (5,5,1)$, write a code to calculate its volume $V = \boldsymbol{a}\cdot(\boldsymbol{b}\times\boldsymbol{c})$ using Julia's definitions of dot and cross products.

Problem 1.10: Calculate π using random numbers

a) Consider a square of side 1 centered at origin, and a circle of radius 1 inscribed into the square. Draw two random numbers x and y from an uniform distribution between $[0,1)$ and calculate its norm. Count the number of pairs (x,y) that falls inside and outside the circle. You'll see that a good approximation for π is achieve after you draw many pairs (x,y) using the expression `pi = 4*inside/total`.

b) Explain why this code gives you π.

c) Use PyPlot or any other plotting package to show the random numbers filling the circle and square (use different colors if inside or outside the circle) as you draw random pairs (x,y).

CHAPTER 2

Differential and Integral Calculus

`Step by step.`

IN A TRADITIONAL introductory class of calculus, your teacher has probably started the discussion defining a function $x(t)$ and a pair of points t_i and t_f. If you are a physicist, $x(t)$ was the trajectory defined by the position at time t. Within the finite interval $\Delta t = t_f - t_i$, the distance traveled is $\Delta x = x(t_f) - x(t_i)$. From these we can define the mean velocity within this interval as $\langle v \rangle = \Delta x / \Delta t$. Moreover, in the limit where t_f and t_i are infinitesimally close, we get the instantaneous velocity:

$$v(t) = \frac{\partial}{\partial t} x(t) = \lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t}. \tag{2.1}$$

Using similar considerations your teacher has shown you that the integral is a sum of infinitesimal contributions:

$$x(t) = x(0) + \int_0^t v(t') dt' = x(0) + \lim_{\Delta t' \to 0} \sum_{n=0}^{N} v\big(t'_n\big) \Delta t', \tag{2.2}$$

where the sum runs over a discrete set labeled by the integer $0 \le n \le N$ that maps the time axes $0 \le t'_n \le t$.

If you remember all that, great! The main idea behind the numerical differential and integral calculus is to drop the infinitesimal limit and simply work with the finite differences. Of course this is too simple. It would be a crude approximation[1]. We can do better! But let's take it step by step... discrete steps.

2.1 Interpolation / Discretization

A numerical calculation usually require us to represent Real (continuous) axis, planes or volumes on the computer. This is impossible, right? Between 0.0 and 1.0 there's already ∞ numbers. We have to represent them in a discrete set, as introduced above.

[1]Before continuing to the next section, please check Problem 2.1 for some motivation.

The interpolation then deals with the unknown data in between discrete points. Let's start plotting `sin(x)` between $0 \le x \le 2\pi$ with a few points only.

Example 2.1: Discrete plot of sin(x)

```
using PyPlot
x = linspace(0, 2pi, 7);
plot(x,  sin(x), "o")
plot(x,  sin(x), "-")
```

In the example above, the first plot draws the circles, while the second draws the lines connecting them, equivalent to the green dots in Fig. 2.1(a). The lines are a linear interpolation of the points. Try again with more or less points to see how it goes.

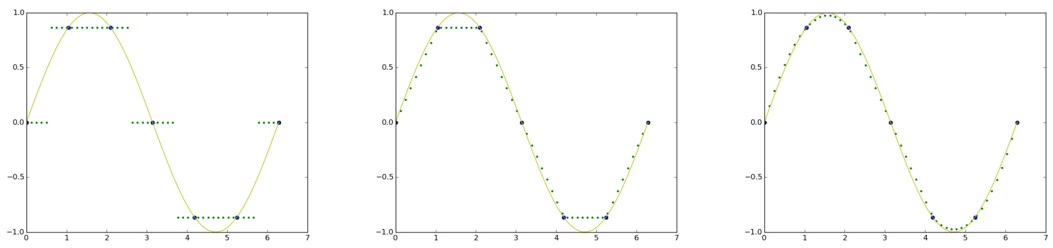

Figure 2.1: Starting with the $\sin(x)$ defined at only 7 points (blue dotes), we calculate the (a) constant (rectangle) (b) linear (trapezoidal) and (c) quadratic interpolation points (green dots). In both panels the yellow lines show the exact $\sin(x)$ for comparison.

Linear interpolation

Say you have a finite set of points $\{x_n\}$ labeled by the integer $0 \le n \le N$, and you known the function $f(x)$ at these points. You may use the data from the example above with $f(x) = sin(x)$, or any other function of your choice.

Now, assume you want the value of the function at an general point x that does not belong to your discrete set $\{x_n\}$, but lies within $x_a \le x \le x_b$. Here $b = a+1$ labeling consecutive points of the set. The most general expression for a linear interpolation connecting the points $\{x_a, f(x_a)\}$ and $\{x_b, f(x_b)\}$ is $\tilde{f}(x) = C_1 x + C_0$. We'll use the $\sim$ symbol to refer to the interpolated function. We want $\tilde{f}(x)$ to match $f(x)$ at x_a and x_b, therefore we have

$$\tilde{f}(x_a) = C_1 x_a + C_0 = f(x_a), \tag{2.3}$$

$$\tilde{f}(x_b) = C_1 x_b + C_0 = f(x_b), \tag{2.4}$$

which defines a system of two equations and two unknowns (C_1 and C_0). This set of equations can be cast in a matrix form $M \cdot C = F$, where the matrix M, coefficient vector C and function vector F are show explicitly below:

$$\begin{pmatrix} x_a & 1 \\ x_b & 1 \end{pmatrix} \begin{pmatrix} C_1 \\ C_0 \end{pmatrix} = \begin{pmatrix} f(x_a) \\ f(x_b) \end{pmatrix}. \tag{2.5}$$

This equation can be easily solved as $C = M^{-1} \cdot F$. Evidently, you are able to express the result with paper & pencil in this case. However, the matrix form makes it easy to generalize to higher order interpolation. The next example shows a code for linear interpolation, see Fig. 2.1(a). In Problem 2.2 I ask you to generalize this code for a quadratic interpolation that should result in Fig. 2.1(b).

As you can see, the quadratic interpolation is already very close to the exact $\sin(x)$ function. The interpolations usually work well because we are almost always dealing with analytical functions, smooth functions. Complicated functions will probably require advanced methods. For instance, functions with singularities.

Example 2.2: Code for Linear Interpolation

```julia
using PyPlot

xlst = linspace(0, 2pi, 7); # create data for the example
flst = sin(xlst);

# receives number of points n (odd) to interpolate
# and the original data via x and f
function interp1(n, x, f)
    xnew = linspace(x[1], x[end], n); # creates new axes
    fnew = zeros(n); # and initialize new data as zero

    i = 1; # label new sites
    for j=1:(length(x)-1) # runs over old sites
        xa = x[j]; # known points
        xb = x[j+1];
        fa = f[j]; # known data
        fb = f[j+1];

        # matrix form to find the coefficients
        M = [xa 1.0; xb 1.0];
        C1, C0 = inv(M)*[fa; fb];

        # calculate the new data within every two points interval
        while i <= n && xnew[i] <= xb
            fnew[i] = C1*xnew[i] + C0;
            i += 1;
        end
    end
    return xnew, fnew; # return interpolated data
end

# calls function to interpolate
xnew, fnew = interp1(51, xlst, flst);

# plot old data, new data, and exact function
plot(xlst, flst, "o");
plot(xnew, fnew, "g.");
plot(xnew, sin(xnew), "y-");
```

2.2 Numerical Integration - Quadratures

Common numerical integration schemes rely on simple interpolations (Newton–Cotes rules). Let's discuss these methods first. Later we give an overview of adaptive integration schemes, which is natively implemented in Julia as the function `quadgk`, and more sophisticated implementations for multi-dimensional integrals can be found on the `Cubature` package. We finish this section discussing the Monte Carlo integration technique.

2.2.1 Polynomial Interpolations

We have briefly discussed polynomial interpolations in the previous section. Now we'll use different interpolations to define common quadrature schemes. To establish a notation, we'll refer to the discrete points as x_n, where the integer n labels the points as in Fig. ?. We'll assume that the set of points $\{x_n\}$ is equally spaced, so we can define a constant step $\Delta x = x_{n+1} - x_n$.

Rectangle rule

The simplest quadrature method is the *rectangle rule*. Here the function is assumed to be constant and equal to $f(x_n)$ within the interval $x_n - \Delta x/2 \le x \le x_n + \Delta x/2$. The interpolation resulting from this rule can be seen in Fig. 2.1(a). Consequently, the integral over this range is

$$\int_{x_n-\frac{\Delta x}{2}}^{x_n+\frac{\Delta x}{2}} f(x)dx \approx f(x_n)\Delta x + \mathcal{O}(\Delta x). \tag{2.6}$$

Trapezoidal rule

The trapezoidal rule uses the linear interpolation shown in Fig. 2.1(b). It is easy to see (Prob. 2.3) that this leads to

$$\int_{x_n}^{x_{n+1}} f(x)dx \approx \frac{f(x_{n+1})+f(x_n)}{2}\Delta x + \mathcal{O}(\Delta x^2). \tag{2.7}$$

Simpson's rule

If we use the parabolic interpolation, we get Simpson's rule:

$$\int_{x_n}^{x_{n+2}} f(x)dx \approx \Big[f(x_n) + 4f(x_{n+1}) + f(x_{n+2})\Big]\frac{\Delta x}{3} + \mathcal{O}(\Delta x^4). \tag{2.8}$$

From Fig. 2.1 one can already expect that the Simpson's rule to give better results and the rectangular or trapezoidal rules.

General remarks on the rules above

Note that the trapezoidal rule is defined over a range set by two consecutive points, while the Simpson's rule runs over three points, from x_n to x_{n+2}. As a consequence the Simpson's rule can only be used if you have an odd number of known points.

Higher order polynomials leads to more precise quadrature rules. Cubic interpolation leads to the Simpson's 3/8 rule, which uses 4 points. For polynomial interpolations of degree 4 we get Boole's rule, which uses 5 points.

In the next example I assume we have a discrete set of points given by `xlst` and `flst` that correspond to the output of some previous calculation. The function `trapezoidal` is then used to integrate the function. Problem 2.1 asks you to implement other rules. Try to change the number of points in the example, as well as other more complicated functions.

```
— Example 2.3: Numerical quadrature of a discrete function —

# receives two vectors for the axis x and function f
function trapezoidal(x, f)
   delta = x[2]-x[1]; # assuming constant step
   res = 0.0; # initializes the sum
   for n=1:(length(x)-1)
       res += 0.5*(f[n+1]+f[n])*delta; # traditional rule
   end
   return res; # returns the result
end

# create data for the example
xlst = linspace(0, 2pi, 7); # x axis
flst = sin(xlst); # discretized function f(x)

trapezoidal(xlst, flst) # calls the function and shows the result
```

We may also need to integrate an exact function $f(x)$. This is done in the next example. Now the `trapezoildal` function receives the function to be integrated, the limits of integration, and the number of points to consider. Here the function is never written as a vector. Instead, it is calculated as needed. This will be more efficient for integration with a large number of points, as it doesn't store the function as the vector `flst` as above.

```
—— Example 2.4: Numerical quadrature of a function f(x) ——

# receives the function g(x) to be integraded
# from x=a to x=b with n points
function trapezoildal(g, a, b, n)
   dx = (b-a)/(n-1.0);
   xi(i) = a + (i-1.0)*dx;
   res = 0.0;
   for i=1:(n-1)
      res += 0.5*(g(xi(i+1))+g(xi(i)))*dx
   end
   return res;
end

f(x) = sin(x); # chosen function

trapezoildal(f, 0, pi, 3) # test with 3 points

trapezoildal(f, 0, pi, 20) # 20 points

trapezoildal(f, 0, pi, 100) # 100 points
```

Note that these examples illustrate two distinct cases. In Example 2.3 we assume that some previous calculation has given us the axis discretized into the vector `xlst` as well as the function `f(x)` calculated at these points and stored in the vector `flst`. This means that we are assuming that we don't have direct access to an exact function `f(x)`.

Example 2.4 presents the opposite case. Here we do have access to the exact function `f(x)`. In this case one may use more sophisticated quadrature schemes. We present some of them in the next section.

2.2.2 Adaptive and multi-dimensional integration

Julia already has a native quadrature code implemented (`quadgk`). It uses an adaptive Gauss-Kronrod integration technique. Since this is an introductory class, I will not go into details of this method[2].

But to get an idea of how the code works, imagine you have two quadrature of different order implemented, say the trapezoidal and Simpson's rules, and you want to integrate `f(x)` from `x=a` to `x=b`. First you split your integration into subintervals, say from `x=a` to `x=c`, and from `x=c` to `x=b`, *i.e.*

$$\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx. \tag{2.9}$$

Then you evaluate each integral on the right hand side independently. You can estimate the error of each subinterval comparing the result of the trapezoidal and Simpson's rules. If the error of a subinterval is too big, you split it into a new pair of subintervals and repeat the test until you converge to the desired error.

An efficient implementation of such methods can be difficult, and I leave it as a challenge for the experienced programmers in Problem 2.4.

Luckily, Julia has already the function `quadgk`. Check its documentation for details on the parameters. The next example shows a basic usage.

Example 2.5: Numerical quadrature with quadgk

```
# let's start with the same function from the previous example
f(x) = sin(x);
# quadgk receives the function and the limits of integration
quadgk(f, 0, pi)

# you may also try functions with integrable singularities...
f(x) = 1/sqrt(x);
quadgk(f, 0, 16) # exact result is 8

# and integrate all the way to infinity (Inf)
f(x) = exp(-(x^2)/2)/sqrt(2);
res = quadgk(f, -Inf, Inf) # exact result is √π
pi-res[1]^2 # compare to π to check the error
```

[2] Those who are interested in mode details, please check the Wikipedia page for Adaptive quadrature and references within: https://en.wikipedia.org/wiki/Adaptive_quadrature

For more advanced routines and multi-dimensional integration, please check Julia's `Cubature` package[3].

2.2.3 Monte Carlo integration

While the previous methods integrate functions sampling them along a regular grid, the Monte Carlo integration scheme samples the integration interval using random numbers. We have used this method already to calculate π back in Problem 1.10, which can be recast as a two-dimensional integral:

$$f(x,y) = \begin{cases} 4 & \text{if } x^2+y^2 \leq 1, \\ 0 & \text{otherwise,} \end{cases} \tag{2.10}$$

$$\pi = \int_\Omega f(x,y)dxdy, \tag{2.11}$$

where the integration area Ω is set by the ranges $0 \leq x \leq 1$ and $0 \leq y \leq 1$.

The Monte Carlo implementation of this integral reads

$$R_N = \frac{1}{N}\sum_{i=1}^{N} f(x_i, y_i). \tag{2.12}$$

The next example shows an implementation of this integral

Example 2.6: Monte Carlo integral to calculate π

```
function Rn(f, n) # Monte Carlo integration of f(x,y)
   res=0.0;
   for i=1:n
      res += f(rand(), rand());
   end
   return res/n;
end

# define the function using the ternary operator
f(x,y) = (x^2+y^2 <= 1.0)?4.0:0.0;

Rn(f, 10) # run the integral with 10 points
Rn(f, 1000) # run the integral with 100 points
```

The main advantage of the Monte Carlo integral is that its error decays with $1/\sqrt(n)$ independently of the dimensions of the integral. Therefore this method becomes very interesting for high-dimensional integrals.

[3]Cubature package: https://github.com/stevengj/Cubature.jl

2.3 Numerical Derivatives

At the beginning of this Chapter, our overview of the numerical calculus introduced the idea of a numerical derivative simply as dropping the infinitesimal limit on the definition of a derivative. Indeed this leads to a familiar expression:

$$\frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}. \tag{2.13}$$

As far as Δx is small, this should give us a good estimate of the derivative. This formula is actually known as the **forward** derivative, as it gives the derivative of $f(x)$ at x using two points: x itself, and the one step forward $x + \Delta x$.

2.3.1 Finite differences and Taylor series

The finite differences scheme is the most common method for numerical differentiation. Its arises from the Taylor expansion of a function $f(x)$:

$$f(x_i + h) = f(x_i) + h \left.\frac{\partial f}{\partial x}\right|_{x_i} + \frac{h^2}{2!} \left.\frac{\partial^2 f}{\partial x^2}\right|_{x_i} + \frac{h^3}{3!} \left.\frac{\partial^3 f}{\partial x^3}\right|_{x_i} + \cdots \tag{2.14}$$

Here x_i is an arbitrary point of our discrete grid labeled by the integer i, and h is the step size between discrete points.

Let's use the Taylor expansion to get the 1^{st} and 2^{nd} derivatives of $f(x)$ using two or three points of the discrete axis $x \to x_i$.

Forward 1^{st} derivative

If we truncate the Taylor expansion on the h^2 term, we can rearrange the remaining terms to read:

$$\left.\frac{\partial f}{\partial x}\right|_{x_i} \approx \frac{f(x_i + h) - f(x_i)}{h} + \mathcal{O}(h) \tag{2.15}$$

Backward 1^{st} derivative

Another choice is to start with the Taylor expansion for a negative step

$$f(x_i - h) = f(x_i) - h \left.\frac{\partial f}{\partial x}\right|_{x_i} + \frac{h^2}{2!} \left.\frac{\partial^2 f}{\partial x^2}\right|_{x_i} - \frac{h^3}{3!} \left.\frac{\partial^3 f}{\partial x^3}\right|_{x_i} + \cdots \tag{2.16}$$

and once again truncate the expansion on the h^2 term to get

$$\left.\frac{\partial f}{\partial x}\right|_{x_i} \approx \frac{f(x_i) - f(x_i - h)}{h} + \mathcal{O}(h) \tag{2.17}$$

Symmetric (or central) 1^{st} derivative

Subtracting Eq. (2.14) from Eq. (2.16) we eliminate the h^2 term

$$f(x_i+h)-f(x_i-h)=2h\left.\frac{\partial f}{\partial x}\right|_{x_i}+2\frac{h^3}{3!}\left.\frac{\partial^3 f}{\partial x^3}\right|_{x_i}+\cdots \tag{2.18}$$

and now we can truncate the sum on the h^3 term to get

$$\left.\frac{\partial f}{\partial x}\right|_{x_i}\approx\frac{f(x_i+h)-f(x_i-h)}{2h}+\mathcal{O}(h^2) \tag{2.19}$$

Symmetric (or central) 2^{nd} derivative

This time let's sum Eqs. (2.14) and (2.16) to eliminate the first derivative:

$$f(x_i+h)+f(x_i-h)=2f(x_i)+h^2\left.\frac{\partial^2 f}{\partial x^2}\right|_{x_i}+\cdots \tag{2.20}$$

Rearranging the expansion truncated on the h^4 term give us

$$\left.\frac{\partial^2 f}{\partial x^2}\right|_{x_i}\approx\frac{f(x_i+h)-2f(x_i)+f(x_i-h)}{h^2}+\mathcal{O}(h^2) \tag{2.21}$$

Run the next example to get a plot comparing the first derivatives with the exact result. Try changing the step size from `h=pi/5` to `pi/50` and `pi/500`.

Example 2.7: Derivative using finite differences

```
# simple implementation of the finite differences
diff1_forward(f, x, h) = (f(x+h)-f(x))/h;
diff1_backward(f, x, h) = (f(x)-f(x-h))/h;
diff1_symmetric(f, x, h) = (f(x+h)-f(x-h))/(2h);

f(x) = sin(x); # chosen function to test

xgrid = 0:(pi/50):pi; # discrete x axis
h = pi/5; # step for the derivatives

# using comprehensions to calculate the derivatives along xgrid
fwd = [diff1_forward(f, x, h) for x=xgrid]
bwd = [diff1_backward(f, x, h) for x=xgrid]
sym = [diff1_symmetric(f, x, h) for x=xgrid]

# plot the exact result and the approximate derivatives for comparison
clf();
plot(xgrid, cos(xgrid); label="cos(x)")
plot(xgrid, fwd; label="forward")
plot(xgrid, bwd; label="backward")
plot(xgrid, sym; label="symmetric")
legend()
```

In the example above we are assuming that we have access to the exact function $f(x)$ at any point. This is similar to what we saw in Example 2.4. What happens if we consider a situation similar to the one in Example 2.3? There we assume that we only known the function $f(x)$ via the discrete points set by `xlst` and `flst`. Check Problem 2.7.

2.3.2 Matrix Representation

Sometimes it is useful to represent the derivatives in matrix forms, such that it becomes an operator. Say that we have a discrete x axis labeled by x_i for $1 \le i \le N$, and a function defined at these points by $f_i = f(x_i)$. These are exactly the `xlst` and `flst` vectors from the previous Examples of this Chapter. Let's assume that $f_i = 0$ for $i \le 0$ and $i \ge N+1$. Let's write the expressions for the symmetric finite differences derivative $f'_i = \left.\frac{\partial f(x)}{\partial x}\right|_{x_i}$ at each point x_i:

$$\text{for i=1,} \quad f'_1 = \frac{f_2 - 0}{2h}, \tag{2.22}$$

$$\text{i=2,} \quad f'_2 = \frac{f_3 - f_1}{2h}, \tag{2.23}$$

$$\text{i=3,} \quad f'_3 = \frac{f_4 - f_2}{2h}, \tag{2.24}$$

$$\text{i=4,} \quad f'_4 = \frac{f_5 - f_3}{2h}, \tag{2.25}$$

$$\cdots \quad f'_i = \frac{f_{i+1} - f_{i-1}}{2h}, \tag{2.26}$$

$$\text{i=N-1,} \quad f'_{N-1} = \frac{f_N - f_{N-2}}{2h}, \tag{2.27}$$

$$\text{i=N,} \quad f'_N = \frac{0 - f_{N-1}}{2h}. \tag{2.28}$$

The zero in the first and last lines refer to $f_0 = 0$ and $f_{N+1} = 0$, respectively.

The set of equations above can be put in a matrix form as

$$\begin{pmatrix} f'_1 \\ f'_2 \\ f'_3 \\ f'_4 \\ \cdots \\ f'_i \\ \cdots \\ f'_{N-1} \\ f'_N \end{pmatrix} = \frac{1}{2h} \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ \cdots \\ f_i \\ \cdots \\ f_{N-1} \\ f_N \end{pmatrix} \tag{2.29}$$

In Julia this matrix can be implemented using the `diagm` function:

- Example 2.8: Matrix representation of the first derivative -

```
x = linspace(-10, 10, 101); # discrete x axis
f = exp(-(x.^2)/2.0); # function sampled at x

n = length(x); # number of points
h = x[2] - x[1]; # step size

matdiff = (diagm(ones(n-1), 1) - diagm(ones(n-1), -1))/(2h);

# calculates the derivative of f as a product of matrix and vector
dfdx = matdiff*f;

plot(x, dfdx);
```

2.3.3 Derivatives via convolution with a kernel

A one-dimensional (1D) discrete convolution of the vectors `f` and `g` read

$$(g*f)[n] = \sum_{m=1}^{N} g[m]f[n+N-1-m], \tag{2.30}$$

where the index of our arrays $g[m]$ and $f[m]$ start at 1 and N is the length of the array g.

Let's refer to the vector `g` as the kernel, and `f` as our function. If the kernel is $g = \frac{1}{2h}(1,0,-1)$, the convolution above becomes

$$(g*f)[n] = \frac{1}{2h}\Big(f[n+1]-f[n-1]\Big), \tag{2.31}$$

which is exactly our definition of the symmetric first derivative.

In Julia we can use the `conv` function for 1D convolutions and the `conv2` function for two-dimensional (2D) convolutions. The next example can be compared with the previous one. Note that after the convolution we remove end points to keep `dfdx` with the same size of `x`.

– Example 2.9: Derivatives via convolution with a kernel 1D –

```
x = linspace(-10, 10, 101); # discrete x axis
f = exp(-(x.^2)/2.0); # function sampled at x

n = length(x); # number of points
h = x[2] - x[1]; # step size

# kernel for the symmetric first derivative 1D
kernel = [1.0; 0.0; -1.0]/(2h);

dfdx = conv(kernel, f); # convolution
dfdx = dfdx[2:end-1]; # remove end points

plot(x, dfdx);
```

In 2D the kernel becomes a matrix. For instance, the kernel for $\frac{\partial^2}{\partial x \partial y}$ represented by the symmetric differences with step sizes h_x and h_y in each direction is

$$g = \frac{1}{4h_x h_y} \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}. \tag{2.32}$$

Example 2.10: Derivatives via convolution with a kernel 2D

```
# x and y axes
# transpose y to get matrix mesh for f later on
x = linspace(-5, 5, 101);
y = transpose(linspace(-5, 5, 101));
hx = x[2]-x[1]; # step sizes
hy = y[2]-y[1];

# 2D function as a matrix (lines are x, columns are y)
f = exp(-(x.^2)/2).*exp(-(y.^2)/2);

# 2D kernel for ∂²/∂x∂y
kernel = (1.0/(4*hx*hy))*[0.0 1.0 0.0; 1.0 0.0 -1.0; 0.0 -1.0 0.0];

df = conv2(kernel, f); # 2D convolution
df = df[2:end-1, 2:end-1]; # remove end points

surf(x, y', df) # surface plot
# requires y to be transposed back into a column vector
```

2.3.4 Other methods, Julia commands and packages for derivatives

Later on we'll see how to take derivatives using the properties of Fourier transforms. This will be useful to solve differential equations, for instance with the split-operator or split-step methods.

Julia has a native command for the first order derivatives: `gradient`. This command uses the forward rule to evaluate the derivative at the first point, the backward rule at the last point, and the symmetric rule at the internal points.

There exists also packages, like the `ForwardDiff`[4]. However there's a warning on the webpage of `ForwardDiff` and at this moment I'm not familiarized with the issues they are facing.

2.4 Problems

Problem 2.1: Checking the simplest numerical integration

Consider the functions $f(x) = \sin(x)$, $g(x) = \cos(x)$, and $h(x) = e^x$.

[4]ForwardDiff package: https://github.com/JuliaDiff/ForwardDiff.jl

a) What is the exact result for the integration of $f(x)$ and $g(x)$ on the interval $0 \leq x \leq 2\pi$? Try also the interval $0 \leq x \leq \pi$. What about the integration of $h(x)$ over $0 \leq x \leq 1$?

b) Try to numerically integrate these functions as initially described at the introduction of this chapter. Simply dropping the infinitesimal limit. Compare the deviation from the exact result as a function of the number of points used in the integration in a log-plot. Useful Julia commands: `linspace`, `sum`.

c) Complement item **(b)** as you learn new methods: compare the precision of different methods (trapezoidal, Simpson's, Boole's, Monte Carlo, ...) as a function of the number of points being used.

Problem 2.2: Interpolation

Go back to Example 2.2, which shows a code for a linear interpolation $\tilde{f}(x) = C_1 x + C_0$. Following the example, implement (a) a code to get a quadratic interpolation $\tilde{f}(x) = C_2 x^2 + C_1 x + C_0$, (b) a code for the rectangle interpolation. You must reproduce Fig. 2.1.

Problem 2.3: Quadrature equations

(a) Show that the linear interpolation leads to the trapezoidal quadrature rule.

(b) Show that the quadratic interpolation leads to the Simpson's rule.

Problem 2.4: Adaptive Integration

This problem might not be easy. I'll leave it here as a challenge for the more experienced programmers among the students: Try to implement an adaptive integration code using the trapezoidal and Simpson's rule discussed in the text.

Problem 2.5: Monte Carlo integration and π

Edit the Monte Carlo Example 2.6 or your code from Problem 1.10 to calculate π as a function of the number n of randomly sampled points. Let's say that R_n is the result of the calculation with n points. The relative error is $E_n = (R_n - \pi)/\pi$. Do a log-log plot of E_n vs n to see that the error goes with $1/\sqrt(n)$, which is the typical behavior of Monte Carlo integrals.

Problem 2.6: Derivatives via finite differences with an exact function

(a) Following Example 2.7, write a code to calculate the second derivative of a function $f(x)$ and compare with the exact result.

(b) Run the Example 2.7 and the code you wrote on item (a) with the following functions

- $f(x) = e^{-x^2}$, for $-5 < x < 5$;
- $f(x) = \sqrt{x}$, for $0 < x < 10$;
- $f(x) = 4x^4 + 3x^3 + 2x^2 + x$, for $-10 < x < 10$.

Problem 2.7: Derivatives via finite differences with a discrete axis

Let's assume that we only have the function $f(x)$ sampled at discrete points set by `xlst` and stored at the vector `flst`:

```
h = pi/5; # grid step
xlst = 0:h:pi; # discrete x axis
flst = sin(xlst); # function evaluated at discrete points
```

(a) Write a code to calculate the first and second derivatives using this discrete data. Here the derivative step size has to be equal to the grid step size. You will face a difficulty at the end points. What can you do?

(b) Test your code with the functions of the previous Problem. Try to vary the grid step to see how the precision changes.

Problem 2.8: Matrix representation of the finite differences

Following Example 2.8, implement the matrix representation for the finite differences derivatives:

(a) First derivative: forward, backward and symmetric. Change the step size trying to visualize the difference between these implementations.

(b) Second derivative: symmetric.

Always test your implementations with well known functions that you can differentiate analytically for comparison.

Problem 2.9: Derivatives via convolution with a kernel

Implement the forward and backward first derivatives, and the second symmetric derivative in 1D using convolution with a kernel following the Example 2.9.

CHAPTER 3

Ordinary Differential Equations

`ENIAC, since 1946`

ENIAC, **E**lectronic **N**umerical **I**ntegrator **A**nd **C**omputer[8], the first electronic general-purpose computer was designed by the US Army and finished in 1946[1]. The goal was to numerically calculate artillery firing tables by solving sets of differential equations. Essentially, $\boldsymbol{F} = m\boldsymbol{a}$ with quite a few complications.

Indeed we are surrounded by differential equations...

$$m\frac{\partial^2 \boldsymbol{r}}{\partial t^2} = \boldsymbol{F}, \tag{3.1}$$

$$i\hbar\frac{\partial\psi(\boldsymbol{r},t)}{\partial t} = H\psi(\boldsymbol{r},t), \tag{3.2}$$

$$\boldsymbol{\nabla}\cdot\boldsymbol{D} = \rho, \quad \boldsymbol{\nabla}\cdot\boldsymbol{B} = 0, \quad \boldsymbol{\nabla}\times\boldsymbol{E} = -\frac{\partial\boldsymbol{B}}{\partial t}, \quad \boldsymbol{\nabla}\times\boldsymbol{H} = \frac{\partial\boldsymbol{D}}{\partial t} + \boldsymbol{J}, \tag{3.3}$$

$$\frac{\partial u}{\partial t} - \alpha^2\boldsymbol{\nabla}^2 u = 0, \tag{3.4}$$

$$\frac{\partial^2 u}{\partial t^2} - c^2\boldsymbol{\nabla}^2 u = 0, \tag{3.5}$$

... to list a few. We also have Navier–Stokes, general relativity, etc.

The unknown quantity in a differential equation can be a function of a single variable, like the position that depends on time on Newton's second law above ($\boldsymbol{r} \equiv \boldsymbol{r}(t)$). These cases are labeled `ordinary differential equations` (ODE). If your function depends upon two or more variables and your differential equation has derivatives on these variables, it becomes a `partial differential equation` (PDE). This is the case of all other examples above. PDEs and ODEs are intrinsically different. Important theorems of ODEs do not apply for PDEs.

[1]ENIAC: `https://en.wikipedia.org/wiki/ENIAC`

Let us compare two simple cases, the Laplace equation in one and two dimensions:

$$\frac{\partial^2 f(x)}{\partial x^2} = 0, \tag{3.6}$$

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) f(x,y) = 0. \tag{3.7}$$

The 1D equation is a second order ODE. This means that its most general solution is defined up to two unknown constants. These are set by two initial or boundary conditions. In this case the general solution is a straight line,

$$f(x) = c_1 x + c_0, \tag{3.8}$$

where c_1 and c_0 are the unknown coefficients.

Contrasting the simplicity of the 1D Laplace equation, the 2D version, which is a PDE, has an infinite number of solutions. Introducing complex variables, it is easy to verify that the most general solution is

$$f(x,y) = p(z) + q(\bar{z}), \tag{3.9}$$

where $p(z)$, $q(\bar{z})$ are arbitrary analytical functions of the complex variables $z = x + iy$, $\bar{z} = x - iy$.

For now, we will focus on `ordinary differential equations` (ODE). These can be classified into three main categories: (i) initial value problems; (ii) boundary-value problems; and (iii) eigenvalue problems. Here we shall restrict ourselves to introductory discussion on the methods to solve ODEs, for more details please check Ref. [4, 6].

3.1 Initial value problem: time-evolution

Initial value problems usually deal with the dynamics of a system, where the derivatives are taken with respect to time. Newton's second law is the paradigmatic example of an initial value ODE,

$$\frac{d^2\boldsymbol{r}}{dt^2} = \frac{\boldsymbol{F}}{m}, \tag{3.10}$$

requiring the `initial` position $\boldsymbol{r}(0)$ and velocity $\boldsymbol{v}(0)$ to be solved.

Let's consider the one dimensional case to guide our discussion. Just keep in mind that a generalization to more dimensions is immediate. First we write Newton's second law in a more general form of an arbitrary second order initial value problem ODE,

$$\frac{d^2x}{dt^2} = g(x,t). \tag{3.11}$$

This second order ODE can be split into a pair of coupled first order ODEs using $v = \frac{dx}{dt}$, yielding

$$\frac{dv}{dt} = g(x,t), \quad \text{and} \quad \frac{dx}{dt} = v. \tag{3.12}$$

Finally, we can define vectors $\boldsymbol{y} = [v(t), x(t)]$ and $\boldsymbol{f} = [g(x,t), v(t)]$, such that the equations above can be cast as

$$\frac{d\boldsymbol{y}}{dt} = \boldsymbol{f}. \tag{3.13}$$

Example: the pendulum

Any physics text-book will tell you that the equation of motion for the pendulum is

$$\frac{d^2\theta(t)}{dt^2} = -\omega^2 \sin[\theta(t)], \tag{3.14}$$

where $\theta(t)$ is the angular displacement, $\omega^2 = g/\ell$, $g \approx 9.8\ \mathrm{m/s}^s$ is the gravity, and ℓ is the pendulum length.

For small oscillations we can use Taylor's expansion to write $\sin\theta \approx \theta$. This leads to the usual harmonic solution $\theta(t) = A\cos(\omega t + \phi)$, where the amplitude A and the phase constant ϕ are set by initial conditions.

Let us try to go beyond the small oscillations approximation and numerically solve Eq. (3.14) with the content of the next sections. To do this, first we have to rewrite Eq. (3.14) in the form of a set of coupled first order ODEs as we generically did above. With this purpose, define[2] $x(t) = \theta(t)$ and $v(t) = \dot{\theta}(t)$. The pendulum equation of motion becomes

$$\frac{dv}{dt} = -\omega^2 \sin[\theta(t)], \quad \text{and} \quad \frac{dx}{dt} = v, \tag{3.15}$$

which can be cast as Eq. (3.13) setting $\boldsymbol{y} = [\dot{\theta}(t), \theta(t)]$ and $\boldsymbol{f} = [-\omega^2 \sin[\theta(t)], \dot{\theta}(t)]$. To solve these equations we will have to specify the initial position $x(0)$ and velocity $v(0)$.

In the next sections we will learn methods to solve differential equations and apply them to this pendulum example. Note that replacing $\sin\theta \to \theta$ in Eq. (3.15) allows you to compare the results with the exact solution for small oscillations. Once you have confidence that your code works, put back the full $\sin\theta$ dependence and compare the small oscillations solution with the numerical solution for large amplitudes.

[2]Within the text I'll use the dot notation to refer to time derivatives, *i.e.* $\dot{\theta} = \dfrac{d\theta}{dt}$

3.1.1 The Euler method

Go back to the previous Chapter and check the expression for the forward first derivative. Applying it to Eq. (3.13) gives

$$\frac{\boldsymbol{y}(t+\tau)-\boldsymbol{y}(t)}{\tau} = \boldsymbol{f}(t), \tag{3.16}$$

where τ is the discrete time step. Labeling the discrete time $t_n = t_0 + n\tau$ with integers n, we can define $\boldsymbol{y}(t_n) = \boldsymbol{y}_n$ and $\boldsymbol{f}(\boldsymbol{y}_n, t_n) = \boldsymbol{f}_n$. Rewriting the equation above give us

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \tau \boldsymbol{f}_n. \tag{3.17}$$

Since in principle we known $\boldsymbol{y}_0$ and $\boldsymbol{f}_0$ (initial values), the equation above can be used to iterate the solution from $n = 0$ to all n. The global error of the Euler method is $\mathcal{O}(\tau)$.

Example 3.1: Euler method - Pendulum

```
using PyPlot

w = 2pi; # frequency, period for small oscillations T = 2pi/w
g(x,t) = -(w^2)*sin(x); # r.h.s. of Newton's 2nd law
f(x,v,t) = [g(x,t); v]; # r.h.s. of linearized ODE

x0 = 0.5pi; # initial angular displacement
v0 = 0.0; # inicial angular velocity
y0 = [v0; x0]; # initial vector for linearized ODE

tau = 1e-4; # time step
tspan = 0:tau:5; # time range

yt = y0; # we will store the data for each t in yt
y = y0; # y at current t for the calculation
for t=tspan[1:end-1]
    y = y + tau*f(y[2], y[1], t); # Euler iteration
    yt = [ yt y ]; # store solution for each time step
end
# data stored in lines, transpose to use with PyPlot
v = transpose(yt[1,:]); # first line is v(t)
x = transpose(yt[2,:]); # second line is x(t)

small = x0*cos(w*tspan); # exact solution for small oscillations

# plot numerical and exact solutions for comparison
clf();
plot(tspan, x; label="numerical");
plot(tspan, small; label="small oscillations");
legend();
```

In the example above, the initial displacement of the pendulum is set by $-\pi < x_0 < \pi$. For $|x_0| \ll \pi$ you should see a good agreement with the exact solution for small oscillations. For large $|x_0| \lesssim \pi$ the numerical simulation will show plateaus as the pendulum slows down near $x_0 = \pm\pi$. However, to see this you will probably have to reduce τ to recover stability. Try to play with the parameters.

3.1.2 Runge-Kutta Methods

The Runge-Kutta methods are the most popular methods for solving ODE due to its high-precision and stability. These methods can be classified as predictor-corrector methods, see Ref. [6]. The most used version is the 4th order Runge-Kutta method, or simply RK4. But let's start with the RK2 for simplicity.

2nd order Runge-Kutta method (RK2)

Our prototype for a differential equation, Eq. (3.13), can be written in a integral form as

$$\boldsymbol{y}(t+\tau) = \boldsymbol{y}(t) + \int_t^{t+\tau} \boldsymbol{f}\Big(y(t'), t'\Big) dt'. \tag{3.18}$$

If we approximate the integrand by a constant value evaluated at the lower limit of the integral, *i.e.* $\boldsymbol{f}(y(t'), t') = \boldsymbol{f}(y(t), t)$, we get the Euler method again: $\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \tau \boldsymbol{f}_n$.

Instead, if we approximate the integrand by the midpoint, *i.e.* $\boldsymbol{f}(y(t'), t') = \boldsymbol{f}(y(t+\tau/2), t+\tau/2)$, we get

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \tau \boldsymbol{f}\Big(\boldsymbol{y}_{n+\frac{1}{2}}, t_n + \frac{\tau}{2}\Big). \tag{3.19}$$

But we don't known $\boldsymbol{y}_{n+1/2}$. To proceed, there's two possibilities:

(i) The explicit RK2 is obtained using the Euler method to express $\boldsymbol{y}_{n+1/2} = \boldsymbol{y}_n + \frac{1}{2}\tau \boldsymbol{f}_n$, yielding

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \tau \boldsymbol{f}\Big(\boldsymbol{y}_n + \frac{\tau}{2}\boldsymbol{f}(\boldsymbol{y}_n, t_n),\ t_n + \frac{\tau}{2}\Big). \tag{3.20}$$

(ii) The implicit RK2 is obtained if we use the midpoint rule to express $\boldsymbol{y}_{n+1/2} = \frac{1}{2}(\boldsymbol{y}_n + \boldsymbol{y}_{n+1})$,

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \tau \boldsymbol{f}\Big(\frac{\boldsymbol{y}_n + \boldsymbol{y}_{n+1}}{2},\ t_n + \frac{\tau}{2}\Big). \tag{3.21}$$

Both version result in global errors $\mathcal{O}(\tau^2)$.

Notice that the **explicit RK2** is slightly more complicated than the Euler method, but still very similar: on the left hand side we have the unknown $\boldsymbol{y}_{n+1}$ at the next time step, and on the right hand side we have all quantities evaluated at the current time step. Therefore it is easy to generalize the Euler code from Example 3.1 to implement the explicit RK2.

In contrast, the **implicit RK2** has the unknown $\boldsymbol{y}_{n+1}$ on both sides of the equation. To solve Eq. (3.20) one can use the *fixed-point iteration* method. Start with an initial guess for $\boldsymbol{y}_{n+1}$, which could be from the Euler method: $\boldsymbol{y}_{n+1}^{[0]} = \boldsymbol{y}_n + \tau \boldsymbol{f}_n$. For now on the superscript $[k]$ in $\boldsymbol{y}_{n+1}^{[k]}$ refers to the level of the iteration. Then iterate the solution until convergence,

$$\boldsymbol{y}_{n+1}^{[0]} = \boldsymbol{y}_n + \tau \boldsymbol{f}_n, \tag{3.22}$$

$$\boldsymbol{y}_{n+1}^{[1]} = \boldsymbol{y}_n + \tau \boldsymbol{f}\Big(\frac{\boldsymbol{y}_n + \boldsymbol{y}_{n+1}^{[0]}}{2},\ t_n + \frac{\tau}{2}\Big), \tag{3.23}$$

$$\boldsymbol{y}_{n+1}^{[2]} = \boldsymbol{y}_n + \tau \boldsymbol{f}\Big(\frac{\boldsymbol{y}_n + \boldsymbol{y}_{n+1}^{[1]}}{2},\ t_n + \frac{\tau}{2}\Big), \tag{3.24}$$

$$\vdots \tag{3.25}$$

$$\boldsymbol{y}_{n+1}^{[k+1]} = \boldsymbol{y}_n + \tau \boldsymbol{f}\Big(\frac{\boldsymbol{y}_n + \boldsymbol{y}_{n+1}^{[k]}}{2},\ t_n + \frac{\tau}{2}\Big). \tag{3.26}$$

Convergence is achieved for large k when $\boldsymbol{y}_{n+1}^{[k+1]} - \boldsymbol{y}_{n+1}^{[k]}$ is sufficiently small.

It is also possible to solve Eq. (3.20) using root-finding algorithms (e.g. Newton's method). Notice that the only unknown quantity is $\boldsymbol{y}_{n+1}$, so let's define an auxiliary function $\boldsymbol{R}(\boldsymbol{y}_{n+1})$,

$$\boldsymbol{R}(\boldsymbol{y}_{n+1}) = \boldsymbol{y}_{n+1} - \boldsymbol{y}_n - \tau \boldsymbol{f}\Big(\frac{\boldsymbol{y}_n + \boldsymbol{y}_{n+1}}{2},\ t_n + \frac{\tau}{2}\Big), \tag{3.27}$$

such that the possible solutions of Eq. (3.20) are the roots of $\boldsymbol{R}(\boldsymbol{y}_{n+1}) = 0$.

Explicit 4th order Runge-Kutta method (RK4)

The RK4 method is by far the most popular method to solve ODEs. Its implementation is not much more complicated than the simples Euler method, but its precision is far superior with a global error $\mathcal{O}(\tau^4)$. The iteration rule for the RK4 is

$$\boldsymbol{k}_1 = \boldsymbol{f}(\boldsymbol{y}_n, t_n), \tag{3.28}$$

$$\boldsymbol{k}_2 = \boldsymbol{f}\Big(\boldsymbol{y}_n + \frac{\tau}{2}\boldsymbol{k}_1, t_n + \frac{\tau}{2}\Big), \tag{3.29}$$

$$\boldsymbol{k}_3 = \boldsymbol{f}\Big(\boldsymbol{y}_n + \frac{\tau}{2}\boldsymbol{k}_2, t_n + \frac{\tau}{2}\Big), \tag{3.30}$$

$$\boldsymbol{k}_4 = \boldsymbol{f}\Big(\boldsymbol{y}_n + \tau\boldsymbol{k}_3, t_n + \tau\Big), \tag{3.31}$$

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \frac{\tau}{6}\Big(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4\Big). \tag{3.32}$$

Since this is the most used method for differential equations, I will not show an implementation example here. Instead, I'll leave its implementation as a problem for the students.

3.1.3 Stiff equations

Stiff differential equations are those where the numerical solution require a step size excessively small when compared with the actual smoothness of the solution. Typically, a stiff equation solved with a large step size show spurious oscillations.

A stiff differential equation can be as simple as

$$\frac{dy(t)}{dt} = -15y(t), \tag{3.33}$$

which has an exact solution $y(t) = e^{-15t}$ for the initial condition $y(0) = 1$. Note that $y(t) > 0$ for any $t > 0$. However, if you apply the Euler method with $f(y) = -15y$, you will get $y_{n+1} = y_n - 15\tau y_n$. For a large τ, the Euler method may give negative values for y_{n+1}. Indeed, if you solve this equation with the Euler method with a large τ, you will get the spurious oscillations.

A better choice is to use the implicit RK2 method. In this particular case you can actually solve Eq. (3.21) analytically for y_{n+1} to obtain

$$y_{n+1} = \frac{1 - \frac{15}{2}\tau}{1 + \frac{15}{2}\tau} y_n. \tag{3.34}$$

The next example compares the numerical solutions of this stiff equation obtained with the explicit and implicit RK2 methods. Try running it with different step sizes τ and compare the results.

— Example 3.2: Stiff ODE, implicit vs explicit RK2 methods —

```
# explicit RK2 solver receives the right hand side function,
# initial (t0) and final (t1) times, time-step (tau),
# and initial condition (y0)
function explicitRK2(rhs, t0, t1, tau, y0)
   tspan = t0:tau:t1; # sets the time span
   y = y0; # stores solutions in y
   yt = y0; # yt is the auxiliary for the iterations
   for t=tspan[1:end-1]
      # explicit RK2 rule:
      yt = yt + tau*rhs(t+tau/2, yt+0.5*tau*rhs(t, yt));
      y = [ y; yt ]; # stores solutions
   end
   return tspan, y;
end

# does not need the rhs as it is implemented specifically
# for the ODE: dy/dt = -15y
function implicitRK2(t0, t1, tau, y0)
   tspan = t0:tau:t1;
   y = y0;
   yt = y0;
   for t=tspan[1:end-1]
      # explicit solution of the implicit rule for this particular ODE
      yt = yt*(1.0-15*tau/2.0)/(1.0+15*tau/2.0);
      y = [ y; yt ];
   end
   return tspan, y;
end

rhs(t,y) = -15*y; # defines the right hand side of the ODE
y0 = 1.0; # initial condition

tau = 1.0/10; # same tau for both methods
te, ye = explicitRK2(rhs, 0.0, 1.0, tau, y0); # calls explicit RK2
ti, yi = implicitRK2(0.0, 1.0, tau, y0); # calls implicit RK2

texact = 0:0.01:1;
exact = exp(-15*texact); # exact solution for comparison

clf(); # plot the results
plot(texact, exact; label="Exact");
plot(te, ye; label="Explicit");
plot(ti, yi; label="Implicit");
legend();
axis([0.0, 1.0, -1.5, 1.5]);
```

A more complicated stiff ODE is

$$\frac{dy(t)}{dt} = y^2(t) - y^3(t). \tag{3.35}$$

If you put $f(y) = y^2 - y^3$ into the implicit RK2 Eq. (3.21), you will have three roots. Which one should you use? In this case it is better to use the fixed-point iteration method.

In the next example implement the implicit RK2 method using the fixed-point iteration for an initial condition $y(0) = y_0$ within a time range $0 \le t \le 2/y_0$. For small y_0 the ODE become stiff and the solution will require a very small τ to converge. You may use the implementation of the explicit RK2 from the previous example to compare with the implicit RK2 again.

— Example 3.3: Stiff ODE, implicit vs explicit RK2 methods —

```
# implicit RK2 solver receives the right hand side function,
# initial (t0) and final (t1) times, time-step (tau),
# initial condition (y0), and relative tolerance (reltol)
function implicitRK2(rhs, t0, t1, tau, y0, reltol)
   tspan = t0:tau:t1;
   y = y0;
   yt = y0;
   for t=tspan[1:end-1]
      yold = yt; # previous value
      ynext = yt + tau*rhs(t, yt); # next value
      k = 0; # loop counter
      # check convergence, while loops
      while abs(ynext-yold) > reltol*ynext && k < 50000
         yold = ynext; # update old
         ynext = yt + tau*rhs(t+tau/2.0, (yt+ynext)/2.0); # update new
         k += 1;
      end
      yt = ynext; # final result
      y = [ y; yt ];
   end
   return tspan, y;
end

rhs(t,y) = y^2 - y^3;
y0 = 0.1;

tau = (2.0/y0)/5;
te, ye = explicitRK2(rhs, 0.0, 2/y0, tau, y0, 1e-6);
ti, yi = implicitRK2(rhs, 0.0, 2/y0, tau, y0, 1e-6);

clf();
plot(te, ye; label="Explicit");
plot(ti, yi; label="Implicit");
legend();
axis([0.0, 2/y0, -0.1, 1.5]);
```

After checking the results of the code above, try reducing the step size to `tau = (2.0/y0)/10` and `tau = (2.0/y0)/100` to see how the solution improves. Next, reduce y_0 to `y0 = 0.01` and the solutions will split again. Reduce τ until they match. Now try for `y0 = 0.0001`. As you try different parameters, zoom in into the $y = 1$ plateau to see the oscillations on the solutions.

3.1.4 Julia's ODE package

The ODE package[3] provides efficient implementations of adaptive Runge-Kutta methods, including a method for stiff ODEs. To install it, simply call `Pkg.add("ODE")`. The ODE solvers are labeled `odeXY`, where X is the main order of the method, and Y is the order of the error control. Let's focus on the `ode45` solver.

All methods of the ODE package are implemented to solve the differential equation

$$\frac{d\boldsymbol{y}}{dt} = \boldsymbol{F}(t, \boldsymbol{y}), \tag{3.36}$$

and the methods obey the function prototype:

```
tout, yout = odeXX(F, y0, tspan; keywords...)
```

Input: `F` is a function that receives the current time `t` and the corresponding vector `y`, and returns a vector as defined by the right hand side of Eq. (3.36). Additionally, `odeXX` receives a vector with the initial conditions `y0` (= $\boldsymbol{y}(0)$), and the range of time over which the solution is desired (`tspan`). The last parameter, `keywords`, are allow you to set extra configurations, like the error tolerance. I suggest you use the keyword `points=:specified`, so that the output is returned only for each time instant in `tspan = 0:dt:tmax`.

Output: the package returns `tout` and `yout`. The first, `tout`, is the list of time instants t at which the solution $\boldsymbol{y}(t)$ was calculated. If the keyword `points=:specified` was passed to `odeXX`, then `tout = tspan`, otherwise it may vary. The solutions $\boldsymbol{y}(t)$ are returned in `yout`.

Next I adapt Example 3.1 to run with the `ODE` package.

[3]ODE: https://github.com/JuliaLang/ODE.jl

Example 3.4: ODE Package - Pendulum

```
using PyPlot
using ODE

w = 2pi; # frequency, period for small oscillations T = 2pi/w
g(x,t) = -(w^2)*sin(x); # r.h.s. of Newton's 2nd law
f(x,v,t) = [g(x,t); v]; # r.h.s. of linearized ODE

x0 = 0.5pi; # initial angular displacement
v0 = 0.0; # inicial angular velocity
y0 = [v0; x0]; # initial vector for linearized ODE

tau = 1e-4; # time step
tspan = 0:tau:5; # time range

# the rhs function is our implementation of F(t,y) from Eq. (3.36)
rhs(t, y) = f(y[2], y[1], t);
tout, yout = ode45(rhs, y0, tspan; points=:specified);
x = map(k-> k[2], yout); # extract x from yout

small = x0*cos(w*tspan); # exact solution for small oscillations

# plot numerical and exact solutions for comparison
clf();
plot(tspan, x; label="numerical");
plot(tspan, small; label="small oscillations");
legend();
```

3.2 Boundary-value problems

While the differential equations of initial value problems require initial conditions set on a single point (*e.g.:* position and velocity at $t = 0$), the boundary-value problem (BVP) applies when the constraints are specified at two or more points (*e.g.:* the potential of the source and drain electrodes on a sample). Therefore, typical boundary-value problems involve second oder differential equations.

We can split the boundary-value problems into two categories. First we will discuss differential equations in the Sturm-Liouville form, including both homogeneous and inhomogeneous cases. These are linear equations and we shall use its well known properties to optimize our numerical implementations. Later we discuss non-linear differential equations for which the properties of the Sturm-Liouville case do not apply.

The methods discussed here are also valid for the initial-value problems of the preceding section, since these are simply a particular case of the class of boundary-value problems.

3.2.1 Boundary conditions: Dirichlet and Neumann

The boundary conditions of a differential equation reflect an external constraint that must be imposed on the general solution in order to find the specific solution of the problem at hand. For instance, on an oscillating string one might have the ends of the string fixed. In this case the string oscillation obey the wave equation, while fixed ends conditions must be imposed on the solution, constraining it to have vanishing oscillation at these end points. The solution of an differential equation that satisfy the specified boundary conditions is unique. For ordinary differential equations, there's two types of possible cases: (i) Dirichlet; and (ii) Neumann boundary conditions.

To guide our discussion, let's consider a general second order differential equation of the form

$$y''(x) = f[y'(x), y(x), x], \tag{3.37}$$

where we use the prime notation for the derivatives (i.e. $y'(x) = dy/dx$, and $y''(x) = d^2y(x)/dx^2$), and $f(y', y, x)$ is a function to be specified by the problem at hand.

A boundary condition is said to be of the **Dirichlet** type when it constrains the value of the solution at specific points. For instances, it requires that at the points $x = x_0$ and $x = x_1$ the solution $y(x)$ must satisfy $y(x_0) = y_0$, and $y(x_1) = y_1$, where y_0 and y_1 are constants. If we are dealing with the heat equation, these boundary conditions could be specifying the constant temperatures at two reservoirs. On a electrostatic problem (see the Poisson equation example below), the Dirichlet boundary condition is used to impose the the potential difference between two distant electrodes.

A **Neumann** boundary condition constrains the derivative of the solution at specific points. For instance, it may require $y'(x_0) = v_0$ and $y'(x_1) = v_1$, where v_0 and v_1 are constants. On an electrostatic problem, these could be specifying the electric fields at the end points. On the wave equation the boundary condition $y'(L) = 0$ is used when the point L is not fixed.

One can always mix these types of boundary conditions. These are called **mixed** boundary conditions. For instance, we may require $y(x_0) = y_0$ and $y'(x_1) = v_1$. Notice that these are specified at distinct points x_0 and x_1. If they were both specified at the same point, i.e. $y(x_0) = y_0$ and $y'(x_0) = v_0$, we would have the initial-value problem discussed in the previous section. Therefore, one may see the initial-value problem as a particular case of the boundary-value problem. Nonetheless, it is important to distinguish these cases because the techniques used to solve them are distinct.

For ordinary differential equations, the boundary condition that leads us back to the initial-value problem is called **Cauchy** boundary condition. However, this nomenclature is irrelevant in one dimension (which is always the case for ordinary differential conditions). Later on we will discuss partial differential equations, where we'll come back to this discussion to properly present the **Cauchy** and the **Robin** boundary conditions.

Example: the Poisson equation

As a paradigmatic example of the boundary-value problem, consider the Poisson equation for the electrostatic potential in one-dimension. Let us start from the electrostatic Gauss law, $\nabla \cdot \boldsymbol{E}(\boldsymbol{r}) = \rho(\boldsymbol{r})/\epsilon_0$, where ϵ_0 is vacuum dielectric constant, $\rho(\boldsymbol{r})$ is the charge density at $\boldsymbol{r}$, and $\boldsymbol{E}(\boldsymbol{r})$ is the electric field at $\boldsymbol{r}$. In electrostatics the electric field can be written in terms of the scalar potential $\phi(\boldsymbol{r})$ as $\boldsymbol{E}(\boldsymbol{r}) = -\nabla\phi(\boldsymbol{r})$, such that the Gauss law takes the form of the Poisson equation, $\nabla^2\phi(\boldsymbol{r}) = -\rho(\boldsymbol{r})/\epsilon_0$. In three-dimensions the Poisson equation is a partial differential equation (PDE), as it has partial derivatives in x, y and z. In this Chapter we are interested in ordinary differential equations (ODE) only, therefore we will consider the one-dimensional case of the Poisson equation,

$$\frac{\partial^2\phi(z)}{\partial z^2} = -\frac{\rho(z)}{\varepsilon_0}. \tag{3.38}$$

Let's say that we have metallic contacts setting the electrostatic potential at $z = \pm L$ as $\phi(\pm L) = \pm\phi_0$, and there's a narrow charge distribution at $z = 0$ set as $\rho(z) = q\delta(z)$. You can solve this problem analytically to get

$$\phi(z) = -\frac{q}{2\epsilon_0}(|z|-1) + \frac{\phi_0}{L}z. \tag{3.39}$$

This solution is shown in Fig. 3.1.

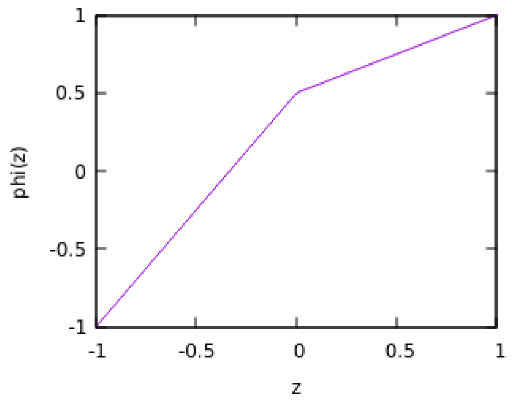

Figure 3.1: Illustration of the solution of the Poisson equation given by Eq. (3.39) for $\phi_0 = L = q = \varepsilon_0 = 1$.

In this section we will learn how to solve this equation numerically for an arbitrary charge distribution $\rho(z)$. Since it might be difficult to deal with δ distributions numerically, we shall consider, for instance, a Gaussian charge distribution $\rho(z) = qe^{-\frac{(z-z_0)^2}{2\Gamma^2}}$, where z_0 is the center of the distribution and Γ is the broadening. For $z_0 = 0$ and small Γ you should be able to reproduce the analytical solution above.

3.2.2 The Sturm-Liouville problems

In physics, many problems fall into differential equations that take the Sturm-Liouville form,

$$\left\{\frac{d}{dx}\left[p(x)\frac{d}{dx}\right] + q(x)\right\}y(x) = \lambda w(x)y(x) + r(x), \tag{3.40}$$

where $p(x)$, $q(x)$, $w(x)$ and $r(x)$ are functions specified by the problem at hand, while $y(x)$ is the unknown function that we want to find. For instance, the Poisson equation in Eq. (3.38) is set by using $p(x) = 1$, $q(x) = w(x) = 0$, and $r(x) = -\rho(x)/\epsilon_0$. If $w(x) \neq 0$, λ is the eigenvalue of the equation. We will discuss the class of eigenvalue problems later on in this Chapter, therefore, for now we shall take $w(x) = 0$. Moreover, let's assume that $p(x)$, $q(x)$ and $r(x)$ are analytical functions over the domain of interest.

Let's first present useful properties of the Sturm-Liouville equation without properly deriving them. For the derivations and more details please check specialized books on Mathematical Physics [refs] and Problem 3.4. Next, we use these properties to establish the Wronskian method to solve Sturm-Liouville problems.

The homogeneous Sturm-Liouville equation

If $r(x) = 0$, Eq. (3.40) is said homogeneous as it takes the form $\mathscr{L}y(x) = 0$, where $\mathscr{L}$ is the Sturm-Liouville operator, which is given by the terms between curly brackets, $\{\cdots\}$, in Eq. (3.40). It follows the properties:

(1) Principle of superposition. Since this equation is linear, if $y_a(x)$ and $y_b(x)$ are solutions, then the linear combination $y_c(x) = ay_a(x) + by_b(x)$ also satisfies $\mathscr{L}y_c = 0$. Here a and b are arbitrary constants that shall be defined by the boundary conditions, i.e. the linear combination coefficients.

(2) Linear independence. Two solutions $y_a(x)$ and $y_b(x)$ are linearly independent if their Wronskian $W(x) \neq 0$ over the domain of x. The Wronskian of a pair of solutions $y_a(x)$ and $y_b(x)$ is $W(x) = y_a'(x)y_b(x) - y_b'(x)y_a(x)$. For the class of Sturm-Liouville homogeneous equations it is easy to show that

$$W(x) = W(x_0)\exp\left\{-\int_{x_0}^{x}\frac{p_1(x')}{p(x')}dx'\right\}, \tag{3.41}$$

where $p_1(x) = \frac{d}{dx}p(x)$, and x_0 belongs to the domain of x. Therefore, if the Wronskian $W(x_0) \neq 0$ in a specific point $x = x_0$, it will be non-zero over the whole domain.

(3) Uniqueness of the solution. Since the Sturm-Liouville equation is a second-order ordinary differential equation, its solution is unique if it satisfies the ODE and two boundary (or initial) conditions. As a consequence, its most general solution can be written as a linear combination of two linearly independent solutions.

The properties above form the basis need to solve the homogeneous Sturm-Liouville problem using the Wronskian method. But before discussing this method, let's check the inhomogeneous Sturm-Liouville problem.

The inhomogeneous Sturm-Liouville equation

For $r(x) \neq 0$ the Sturm-Liouville problem takes the form $\mathscr{L}y(x) = r(x)$, and is said to be inhomogeneous as it has a term independent of $y(x)$. In this case, the principle of superposition as stated above is not valid (see Problem 3.4). Instead, the generalized principle of superposition of inhomogeneous equations states that the most general solution can be written as

$$y(x) = ay_a(x) + by_b(x) + y_p(x), \tag{3.42}$$

where $y_a(x)$ and $y_b(x)$ are linearly independent solutions of the homogeneous equation $\mathscr{L}y_{a/b}(x) = 0$, and $y_p(x)$ is the particular solution of the inhomogeneous equation $\mathscr{L}y_p(x) = r(x)$. Here again, a and b are arbitrary constants to be set by the boundary conditions.

Since $y_a(x)$ and $y_b(x)$ are solutions of the homogeneous case, property **(2)** above follows and the Wronskian criteria can be used to verify or assure that $y_a(x)$ and $y_b(x)$ are linearly independent solutions. Property (3) also follows, and the solution $y(x)$ is unique if and only if it satisfies the ODE $\mathscr{L}y(x) = r(x)$ and the required boundary conditions.

3.2.3 The Wronskian method

The Wronskian method applies to both homogeneous and inhomogeneous cases of the Sturm-Liouville equation. Essentially, it provides a method to obtain a pair of linear independent solutions $y_a(x)$ and $y_b(x)$. We will state the problem for the inhomogeneous case, since the homogeneous case is simply a particular case in which $r(x) = 0$.

To guide our description let's consider Dirichlet boundary conditions, i.e. $y(x_0) = y_0$ and $y(x_1) = y_1$. Generalizations to Neumann or mixed boundary conditions will be simple enough and we leave it to the reader as an exercise. We want to solve the Sturm-Liouville equation $\mathscr{L}y(x) = r(x)$ in the domain $x_0 \le x \le x_1$.

First step: find two linearly independent solutions of the homogeneous equation

Our first step is to find a pair of linearly independent solutions $y_a(x)$ and $y_b(x)$ that satisfy the *homogeneous* equation $\mathscr{L}y_{a/b}(x) = 0$. Accordingly to property **(2)** above, $y_a(x)$ and $y_b(x)$ will be linearly independent if their Wronskian $W(x) \neq 0$ at any point x. For practical purposes, we chose to analyze the Wronskian at $x = x_0$, the left end point of the domain. Namely, we want

$$W(x_0) = y_a'(x_0)y_b(x_0) - y_a(x_0)y_b'(x_0) \neq 0. \tag{3.43}$$

Notice that the final solution will be given by $y(x)$ in Eq. (3.42). Therefore the boundary conditions must apply to $y(x)$, and not to the auxiliary functions $y_a(x)$, $y_b(x)$ or $y_p(x)$. Therefore we can attribute auxiliary boundary conditions to $y_a(x)$ and $y_b(x)$ in order to assure that the condition above is satisfied and the pair $[y_a(x), y_b(x)]$ forms a set of linearly independent solutions. For instance, we may choose

$$y_a(x_0) = 0, \text{ and } y_a'(x_0) = A_1, \tag{3.44}$$

$$y_b(x_0) = A_2, \text{ and } y_b'(x_0) = 0, \tag{3.45}$$

such that $W(x_0) = A_1A_2$, where $A_1 \neq 0$ and $A_2 \neq 0$ are arbitrary nonzero constants.

The auxiliary boundary conditions above actually transforms the problem of finding $y_{a/b}(x)$ into a pair of independent **initial-value problems**, with the initial values for $y_a(x)$, $y_b(x)$ (and their derivatives) set above in terms of the arbitrary A_1 and A_2. We can use any initial-value problem method to obtain $y_{a/b}(x)$ satisfying the homogeneous equation $\mathscr{L} y_{a/b}(x) = 0$.

Second step: find the particular solution of the inhomogeneous equation

Now we need to find $y_p(x)$. Since this is also an auxiliary function, its boundary or initial conditions are arbitrary. Let's use $y_p(x_0) = B_1$ and $y'_p(x_0) = B_2$, where B_1 and B_2 are arbitrary constants. We can use any method of the initial-value problems to find $y_p(x)$.

Third step: impose the physical boundary conditions

In this final step we must impose the physical boundary conditions stated by our problem. Namely, we want $y(x_0) = y_0$ and $y(x_1) = y_1$. From Eq. (3.42) we have

$$y(x_0) = a y_a(x_0) + b y_b(x_0) + y_p(x_0) = b A_2 + B_1, \tag{3.46}$$
$$y(x_1) = a y_a(x_1) + b y_b(x_1) + y_p(x_1), \tag{3.47}$$

where $y_a(x_0) = 0$, $y_b(x_0) = A_2$, and $y_p(x_0) = B_1$ where the auxiliary initial conditions set above. The quantities $y_a(x_1)$, $y_b(x_1)$, and $y_p(x_1)$ are known from the solution of the auxiliary initial-value problems. To satisfy the boundary conditions, the coefficients a and b must be

$$b = \frac{y_0 - B_1}{A_2}, \tag{3.48}$$
$$a = \frac{y_1 - b y_b(x_1) - y_p(x_1)}{y_a(x_1)}. \tag{3.49}$$

Combining these quantities, we have the all terms of Eq. (3.42) to compose our final solution $y(x)$.

Example: the Poisson equation via the Wronskian method

Let's apply the Wronskian method to the Poisson equation to reproduce numerically the example of the beginning of this section; see Fig. 3.1. Let's write the Poisson equation, Eq. (3.38), using the notation of the Sturm-Liouville problem above. It reads

$$\frac{d^2}{dx^2} y(x) = -\rho(x). \tag{3.50}$$

Next we use the ODE package to numerically find the auxiliary functions $y_a(x)$, $y_b(x)$ and $y_p(x)$ with appropriate initial conditions, and combine them at the end to satisfy the physical boundary condition.

The homogeneous version of the 1D Poisson's equation ($\rho(x) = 0$) is actually Laplace's equation in one dimension. Therefore it would be quite easy to find the solutions $y_a(x)$ and $y_b(x)$ of the homogeneous equation. It's simply $y_a(x) = (x - x_0) A_1$ and $y_b(x) = A_2$. However, in the numerical code below we choose to find these simple solutions numerically just to exemplify how to proceed in a more difficult scenario.

Example 3.5: Poisson via Wronskian method

```
using ODE
using PyPlot

z = linspace(-1.0, 1.0, 1000); # z axes
eps0 = 1.0; # ε0

# charge density ρ(z)
g = 0.01; # broadening Γ
rho(z) = exp(-(z.^2)/(2*g^2))/(sqrt(2pi)*g);

# physical boundary conditions: y0 and y1
y0 = -1.0;
y1 = +1.0;

# find ya(z)
A1 = 1.0;
ic = [0.0; A1]; # = (0, A1)
rhs(z, y) = [y[2]; 0.0];
za, ya = ode45(rhs, ic, z; points=:specified);
ya = map(k->k[1], ya);

# find yb(z)
A2 = 1.0;
ic = [A2; 0.0]; # = (A2, 0)
rhs(z, y) = [y[2]; 0.0];
zb, yb = ode45(rhs, ic, z; points=:specified);
yb = map(k->k[1], yb);

# find yp(z)
B1 = 0.0;
B2 = 0.0;
ic = [B1; B2]; # = (B1, B2)
rhs(z, y) = [y[2]; -rho(z)/eps0];
zp, yp = ode45(rhs, ic, z; points=:specified);
yp = map(k->k[1], yp);

# coefficients and final solution
b = (y0 - B1)/A2;
a = (y1 - b*yb[end] - yp[end])/ya[end];
y = a*ya + b*yb + yp;

plot(z, y);
```

3.2.4 Schroedinger equations: transmission across a barrier

Let's consider a free electron in one dimension colliding with a general shaped barrier set by a potential $V(x)$. The dynamics of this electron is tunnel or scatter back as it collides with the barrier. Here we want to calculate the transmission T and reflection R probabilities.

The Hamiltonian of the system is

$$H = -\frac{1}{2}\frac{\partial^2}{\partial x^2} + V(x). \tag{3.51}$$

Here we use atomic units ($\hbar = 1$, $m = 1$), and $V(x)$ is a general potential within the scattering region $0 \le x \le L$, and $V(x) = 0$ outside these limits.

Referring to the region $x < 0$ as I, let's assume that the state of the electron is a linear combination of the injected and reflected waves,

$$\psi_I(x) = e^{ikx} + re^{-ikx}, \tag{3.52}$$

where r the reflection coefficient, and $k = \sqrt{2\varepsilon}$ and ε is the electron energy.

We'll refer to the region $x > L$ as III. There the electron state is composed solely by the transmitted wave,

$$\psi_{III}(x) = te^{ikx}, \tag{3.53}$$

where t is the transmission coefficient.

At the central region (II), where $0 \le x \le L$, we need to find two linear independent solutions, $\psi_a(x)$ and $\psi_b(x)$, of the Schroedinger equation $H\psi(x) = \varepsilon\psi(x)$. I'll leave this as a Problem for the reader. These solutions can be combined in a general linear combination

$$\psi_{II}(x) = a\psi_a(x) + b\psi_b(x), \tag{3.54}$$

where a and b are the linear combination coefficients.

The final solution and its derivative must be continuous at the interfaces $x = 0$ and $x = L$. Therefore, we impose the conditions: (i) $\psi_I(0) = \psi_{II}(0)$; (ii) $\psi'_I(0) = \psi'_{II}(0)$; (iii) $\psi_{II}(L) = \psi_{III}(L)$; (iv) $\psi'_{II}(L) = \psi'_{III}(L)$. These give us the following equation for the coefficients a, b, r and t:

$$\begin{pmatrix} \psi_a(0) & \psi_b(0) & -1 & 0 \\ \psi'_a(0) & \psi'_b(0) & ik & 0 \\ \psi_a(L) & \psi_b(L) & 0 & e^{ikL} \\ \psi'_a(L) & \psi'_b(L) & 0 & ike^{ikL} \end{pmatrix} \begin{pmatrix} a \\ b \\ r \\ t \end{pmatrix} = \begin{pmatrix} 1 \\ ik \\ 0 \\ 0 \end{pmatrix}. \tag{3.55}$$

From the equation above one can easily extract t. The transmission probability is then $T = |t|^2$, and the reflection probability is $R = |r|^2$. You can check that $T + R = 1$.

In Fig. 3.2 we consider the transmission over a Gaussian barrier

$$V(x) = e^{-\frac{1}{2}(x-x_0)^2}, \tag{3.56}$$

where $x_0 = L/2$ is the center of the barrier, and $L = 20$.

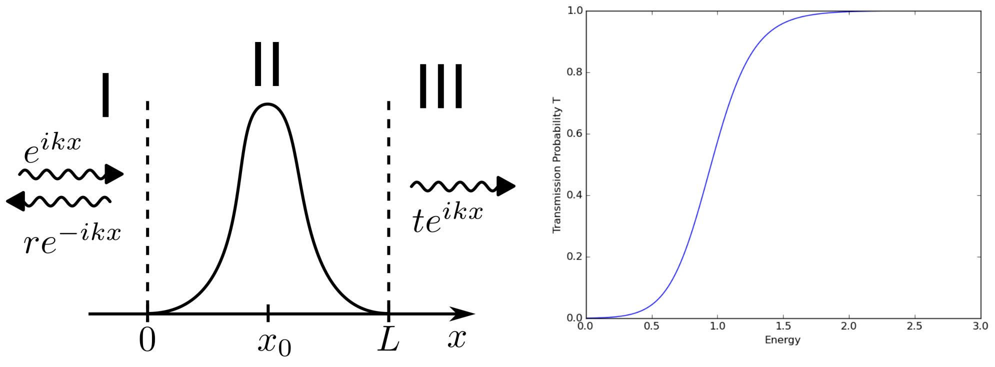

Figure 3.2: (left) Illustration of the Gaussian barrier with the injected, reflected and transmitted waves at outer regions. (right) Transmission probability T as a function of the energy ε.

3.2.5 Non-linear differential equations

For non-linear differential equations, the properties of Sturm-Liouville operator discussed in the previous sections does not hold. Particularly, there is no superposition principle for non-linear ODEs. As a consequence, the task to solve a boundary value problem for a non-linear ODE can be quite difficult.

The non-linearity implies that a ODE with a specific set of boundary conditions may have more than one solution. This is again in direct contrast with the Sturm-Liouville case, where the uniqueness theorem (by Picard–Lindelöf) states that given the boundary or initial conditions, the ODE has only one solution.

Let's illustrate this with a very simple differential equation:

$$\frac{d^2y(x)}{dx^2} + |y(x)| = 0, \tag{3.57}$$

which is non-linear due to the absolute value in the second term. Consider that the boundary conditions are $y(0) = 0$ and $y(4) = -2$.

The shooting method

One way of solving this problem is the shooting method. First, convert the second order differential equations into a pair of coupled first order differential equations. This is the same procedure used at the begging of this chapter. We know how to solve initial value problems easily. Therefore, consider the auxiliary initial conditions $y(0) = y_0$ and $y'(0) = v_0$. Clearly, it is useful to set $y_0 = 0$ to automatically satisfy our desired boundary condition. The problem is v_0. What is the appropriate value of v_0 that satisfies our boundary conditions? Namely, we want v_0 set such the evolution of the initial value problem yields $y(4) = -2$.

In the general there's no direct approach to find the appropriate value of v_0. In our example there's actually two possible values of v_0 that yield $y(4) = -2$. To see this, I invite the reader to write a code using the Runge-Kutta method, or Julia's ODE package to solve the initial value problem for the example above for different values of the initial condition $y'(0) = v_0$. Evolving the solution from $x = 0$ to $x = 4$ we can extract $y(4)$ for each value of v_0 and plot $y(4) \times v_0$. This is shown in Fig. 3.3.

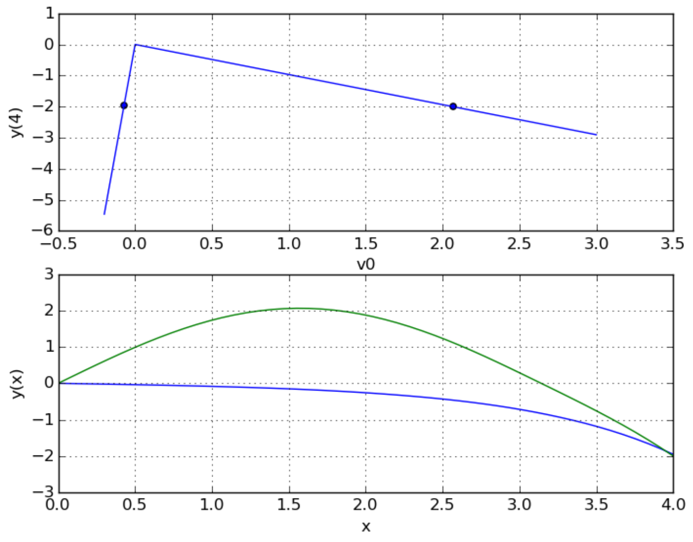

Figure 3.3: (top) Values of $y(4)$ obtained propagating the non-linear differential equation from $x = 0$ to $x = 4$ with the initial condition set to $y'(0) = v_0$, as a function of v_0. There are two values of v_0 that satisfy $y(4) = -2$, as indicated by the circles. (bottom) The two possible solutions $y(x) \times x$ for the values of v_0 found to satisfy the boundary conditions.

3.3 The eigenvalue problem

Here we'll consider only eigenvalue problems of linear and homogeneous differential equations. Particularly, we are only interest again in the Sturm-Liouville problems. In the Sturm-Liouville Eq. (3.40), the differential equation is an eigenvalue problem if $w(x) \neq 0$ and $r(x) = 0$. Therefore we have

$$\left\{\frac{d}{dx}\left[p(x)\frac{d}{dx}\right] + q(x)\right\} y(x) = \lambda w(x) y(x), \tag{3.58}$$

where $p(x)$, $q(x)$ and $w(x)$ are given functions set by the problem at hand, while λ and $y(x)$ are the unknowns that we want to find. For a set of boundary conditions, this differential equation have solutions only for particular values of λ, which are called the eigenvalues. The solutions $y(x)$ associated with each particular λ are the eigenfunctions. If we vectorize this equation using finite differences, the function $y(x)$ becomes a vector: the eigenvector.

There are many ways of solving an eigenvalue problem numerically. But let's first check two analytical cases to serve as examples where we can try our numerical approaches.

3.3.1 Oscillations on a string

A string fixed at its end points separated by a distance ℓ satisfy the wave equation,

$$\frac{\partial^2 y(x,t)}{\partial x^2} - \frac{1}{v^2}\frac{\partial^2 y(x,t)}{\partial t^2} = 0, \tag{3.59}$$

where $y(x,t)$ is the string profile as a function of space x and time t, and v is the wave velocity on the string. This is a partial differential equation, but in this chapter we are dealing with ordinary differential equations only. Since for now we are only interested in the normal modes of oscillation, we can Fourier transform the equation from time t to frequency w, yielding

$$\frac{\partial^2 y(x,\omega)}{\partial x^2} + \frac{\omega^2}{v^2} y(x,\omega) = 0. \tag{3.60}$$

This equation takes the form of eigenvalue Sturm-Liouville problem if we set $p(x) = 1$, $q(x) = 0$, $w(x) = 1$, and $\omega^2 = v^2\lambda$.

If the string is fixed at its end points, the boundary conditions are $y(0,t) = 0$, and $y(\ell,t) = 0$ for any t. Equivalently, the Fourier transformed $y(x,\omega)$ also satisfy the same boundary conditions: $y(0,\omega) = 0$ and $y(\ell,\omega) = 0$ for any ω.

You have probably seen already in a theoretical physics class that the solution to this problem is

$$y(x,\omega_n) = A\sin\left(\frac{\omega_n}{v}x\right), \tag{3.61}$$

where $k_n = \omega_n/v$ are the quantized wave-numbers, $\omega_n = n\pi v/\ell$ are the quantized frequencies, and the integer n labels the normal modes of oscillation. The amplitude A depends on the initial condition, which we will not consider yet.

3.3.2 Electron in a box

An electron in a box is described by the Schroedinger equation. It's dynamics is set by the time-dependent Schroedinger equation. But similarly to the string oscillations, its normal modes are given by a static equation, the time-*independent* Schroedinger equation,

$$-\frac{1}{2}\frac{\partial^2}{\partial x^2}\psi(x) + V(x)\psi(x) = \varepsilon\psi(x), \tag{3.62}$$

where we use atomic units ($\hbar = 1$, $m = 1$) for simplicity. Here $V(x)$ is the electrostatic potential that confines the electron, $\psi(x)$ is the wave-function and ε is the energy. This is indeed a Sturm-Liouville eigenvalue problem where the eigenvalue is the energy ε and the eigenfunction is the wave-function $\psi(x)$.

If the electron is trapped in a box of width ℓ with hard walls, such that $V(x) = 0$ for $0 < x < \ell$, and $V(x) = \infty$ outside, the wave-function must satisfy the boundary-conditions $\psi(0) = 0$ and $\psi(\ell) = 0$.

Since $V(x) = 0$ inside the box, the problem is very similar to the oscillations on a string. The solutions are

$$\psi_n(x) = A\sin(k_n x), \tag{3.63}$$

where the wave-numbers $k_n = n\pi/\ell$, the eigenenergies $\varepsilon_n = \frac{1}{2}k_n^2$, and the integer n labels the quantized eigenstates of the electron.

3.3.3 Method of finite differences

For simplicity let's consider the Sturm-Liouville eigenvalue problem with $p(x) = 1$ and $w(x) = 1$,

$$\left\{\frac{d^2}{dx^2} + q(x)\right\} y(x) = \lambda y(x). \tag{3.64}$$

We have seen in Chapter 2.3.2 that we can represent the derivative operators as matrices. Particularly, the second derivative becomes

$$\frac{d^2}{dx^2}y(x) = \begin{pmatrix} y_1'' \\ y_2'' \\ y_3'' \\ y_4'' \\ \cdots \\ y_i'' \\ \cdots \\ y_{N-1}'' \\ y_N'' \end{pmatrix} = \frac{1}{h^2}\begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \cdots \\ y_i \\ \cdots \\ y_{N-1} \\ y_N \end{pmatrix}, \tag{3.65}$$

where $y_i = y(x_i)$ and x_i are the discretized coordinates. I leave as an exercise to the reader to check that this representation is compatible with the boundary conditions $y(0) = 0$ and $y(\ell) = 0$. Here the discretization of x is such that $x_0 = 0$ and $x_{N+1} = \ell$, and h is the discrete step.

The $q(x)$ term is local, therefore its matrix representation is diagonal,

$$q(x)y(x) = \begin{pmatrix} q_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & q_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & q_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & q_4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \ddots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & q_i & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & q_{N-1} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & q_N \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \cdots \\ y_i \\ \cdots \\ y_{N-1} \\ y_N \end{pmatrix}. \tag{3.66}$$

where $q_i = q(x_i)$.

With these matrix representations, the differential equation above can be put in a matrix form $Hy = \lambda y$. The matrix H is the sum of matrices above, y is the vector composed by y_i, and λ is the eigenvalue.

The most used numerical package to solve eigenvalue problems in a matrix form is Lapack[4]. The language Julia has this package natively implemented in the command `eig`. Try to run the next example.

Example 3.6: Eigenvalues and eigenvectors

```
using PyPlot

# creates the matrix H = -∂_x^2
H = 2*diagm(ones(10)) - diagm(ones(9),1) - diagm(ones(9),-1);

# calculates the eigenvalues and eigenvetors
evals, evecs = eig(H);

subplot(311)
plot(evecs[:,1]) # plot the first eigenvector

subplot(312)
plot(evecs[:,2]) # plot the second eigenvector

subplot(313)
plot(evecs[:,3]) # plot the third eigenvector

# print the first three eigenvalues
println("E1 = ", evals[1]);
println("E2 = ", evals[2]);
println("E3 = ", evals[3]);
```

As you can see in the example, the `eig` command receives the matrix H and returns its eigenvalues in a vector and the eigenvetors as a matrix. In this eigenvector matrix, each eigenvector is set as a column. Therefore `evecs[:,n]` returns the eigenvetor n.

3.4 Problems

Problem 3.1: Damped harmonic oscillator

Implement a code to solve the damped harmonic oscillator problem,

$$\frac{d^2x(t)}{dt^2} = -\omega_0^2 x(t) - \gamma \frac{dx(t)}{dt}. \tag{3.67}$$

Fix $\omega_0 = 2\pi$ and solve the equation for different values of γ to show the solutions for the three possible regimes: (i) subcritical; (ii) critical; and (iii) supercritical. You will find the exact solutions on the volume 2 of Ref. [9].

[4]Lapack: http://www.netlib.org/lapack/

Problem 3.2: Compare Euler and RK for the pendulum

Compare the results of Example 3.1 and Example 3.4 for small and large time steps `tau`, and for small and large oscillations (amplitude `x0`).

Problem 3.3: Runge Kutta RK4

Implement your own version of the RK4 code in Julia to solve the pendulum problem or any other different equation you may prefer. Your code won't be more efficient than the adaptive code in the ODE package. However it is a good practice for you to implement the RK4 by yourself, since it is the most well known and used method for differential equations.

Problem 3.4: Sturm-Liouville equation

Tip: Check Moysés Nussenzveig's book, volume 2, chapter on oscillations [9].

(a) Derive the three properties listed for the homogeneous Sturm-Liouville problem: (1) principle of superposition; (2) linear independence; and (3) Uniqueness of the solution.

(b) Derive the generalization of the principle of superposition for inhomogeneous equations.

Problem 3.5: Wronskian

(a) Show that for an homogeneous Sturm-Liouville differential equation the Wronskian $W(x) = y_a'(x)y_b(x) - y_a(x)y_b'(x)$ can be written as

$$W(x) = W(x_0)\exp\left\{-\int_{x_0}^{x} \frac{p_1(x')}{p(x')}dx'\right\}, \tag{3.68}$$

(b) Show that if the Wronskian is nonzero over the whole domain, the solutions $y_a(x)$ and $y_b(x)$ are linearly independent.

Problem 3.6: Quantum transmission probability for electrons

(a) Why do we identify the positive exponentials in $\psi_I(x)$ and $\psi_{III}(x)$, Eqs. (3.52)-(3.53), as forward moving, and the negative one as a backwards motion?

(b) Show that matching the continuity conditions at the interfaces $x = 0$ and $x = L$ leads to Eq. (3.55).

(c) Implement a code using the Wronskian method to obtain $\psi_a(x)$ and $\psi_b(x)$ and calculate t.

(d) Transform the code of item (c) into a function that receives the energy ε and returns the transmission $T = |t|^2$. Use this code to reproduce the transmission probability plot shown in Fig. 3.2.

(e) Try now with two Gaussian barriers, and run for a thin energy grid. You will see the Fabry–Pérot resonances at the quasi-confined states of a resonant-tunneling diode.

Problem 3.7: The Schroedinger equation and the Sturm-Liouville parameters

(a) What are the expressions for the Sturm-Liouville parameters $p(x)$, $q(x)$, $w(x)$ and $r(x)$ that represents the time-independent Schroedinger equation, Eq. (3.62)?

(b) Show that the solution of electron in a box problem is given by Eq. (3.63) with eigenenergies $\varepsilon_n = \frac{1}{2}(n\pi/\ell)^2$.

Problem 3.8: Matrix representation of the derivative operator

(a) The matrix representation of the derivative operator using finite differences faces a problem at the edges. Show that this is not a problem when we consider hard wall boundary conditions: $y(0) = 0$ and $y(\ell) = 0$.

(b) What changes in the matrix representation if we consider periodic boundary conditions? Namely $y(\ell) = y(0)$.

Problem 3.9: The potential well

(a) Implement a code to solve the time-independent Schroedinger equation for a Gaussian potential well,

$$V(x) = V_0 e^{-\frac{1}{2}(x-c)^2}, \tag{3.69}$$

with amplitude $V_0 = -10$, centered at $c = \ell/2$, hard-wall boundary conditions $\psi(0) = \psi(\ell) = 0$, and $\ell = 10$.

(b) Make a plot of the potential well and the first three eigenstates.

(c) Compare the solution with a discretization of x with 100 points and with 500 points. The result (eigenvalues and eigenvectors) must not change much if the implementation is correct.

CHAPTER 4

Fourier Series and Transforms

`2π? Or not 2π?`

The Fourier series and transform are among the most essential mathematical tools for physicists. TO DO: Finish introduction...

Hereafter we assume that we are dealing with well behaved functions on their relevant domains. We shall restrict ourselves to the one-dimensional cases. Generalizations are imediate.

Every periodic function $f(x)$ can be expanded as a sum of trigonometric functions. If the period of the function is λ, i.e. $f(x+\lambda) = f(x)$, the **Fourier series** reads

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{-ik_n x}, \tag{4.1}$$

where $k_n = 2\pi n/\lambda$, $n \in \mathbb{Z}$, and $e^{i\theta} = \cos\theta + i\sin\theta$. The coefficients c_n are obtained from the orthogonality of the trigonometric functions, yielding

$$c_n = \frac{1}{\lambda}\int_{x_0}^{x_0+\lambda} f(x) e^{ik_n x} dx. \tag{4.2}$$

Here x_0 is arbitrary and the integral runs over one period of the function $f(x)$.

If the function $f(x)$ is not periodic, but we are interested in a finite domain $x \in [x_i, x_f]$, one can extend consider a periodic extension of $f(x)$ outside the domain, such that the Fourier series applies with the replacements $x_0 \to x_i$ and $\lambda \to x_f - x_i$. The periodic extension of $f(x)$ shall match the original function on the desired domain.

Consider now that the function $f(x)$ lives in the symmetric domain $x \in [-\frac{\lambda}{2}, \frac{\lambda}{2}]$, and we take the limit $\lambda \to \infty$. The discrete k_n now become infinitesimally close, $\Delta k = k_{n+1} - k_n = \frac{2\pi}{\lambda} \to 0$, and become a continuous variable $k_n \to k$. Converting the sum in Eq. (4.1) into an integral with $\sum_n \to \frac{\lambda}{2\pi}\int dk$, and replacing $c_n \to (\sqrt{2\pi}/\lambda)\tilde{f}(k)$, the Fourier series, Eq. (4.1), becomes

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \tilde{f}(k) e^{-ikx} dk, \tag{4.3}$$

$$\tilde{f}(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{ikx} dx, \tag{4.4}$$

where $\tilde{f}(k) = F[f(x)]$ is the Fourier transform of $f(x)$, and reciprocally, $f(x) = F^{-1}[\tilde{f}(k)]$ is the inverse Fourier transform of $\tilde{f}(k)$.

The next example shows the Fourier series. Play with the function and parameters.

Example 4.1: Fourier Series

```
using PyPlot

x = linspace(0.0, 30.0, 300); # domain
f(x) = sin(x)./x; # example function

lambda = x[end]-x[1]; # period
k(n) = 2pi*n/lambda; # wave number

# define a function to caculate the coefficients c_n
c(n) = quadgk(x->f(x)*exp(1im*k(n)*x), x[1], x[end])[1]/lambda;

N = 10; # the sum will be $\sum_{n=-N}^{N}$
Ni = -N:N; # array of values of n
Ci = [c(n) for n=Ni]; # calculate the coefficients and store in an array

# combine the arrays to recover the approximate function
fa=sum(i-> Ci[i]*exp(-1im*k(Ni[i])*x), 1:(2*N+1))

clf(); # the plot is shown in Fig. 4.1
subplot2grid((2,2),(0,0),colspan=2)
plot(x,f(x)); # plot the original function
plot(x, real(fa)) # and the approximated one
xlabel(L"$ x $");
ylabel(L"$ f(x) $");

subplot2grid((2,2),(1,0))
vlines(Ni, 0, real(Ci)); # real part of c_n
scatter(Ni, real(Ci));
xlabel(L"$ n $")
ylabel(L"$ Re\{c_n\} $")

subplot2grid((2,2),(1,1))
vlines(Ni, 0, imag(Ci)); # imaginary part of c_n
scatter(Ni, imag(Ci));
xlabel(L"$ n $")
ylabel(L"$ Im\{c_n\} $")

tight_layout();
```

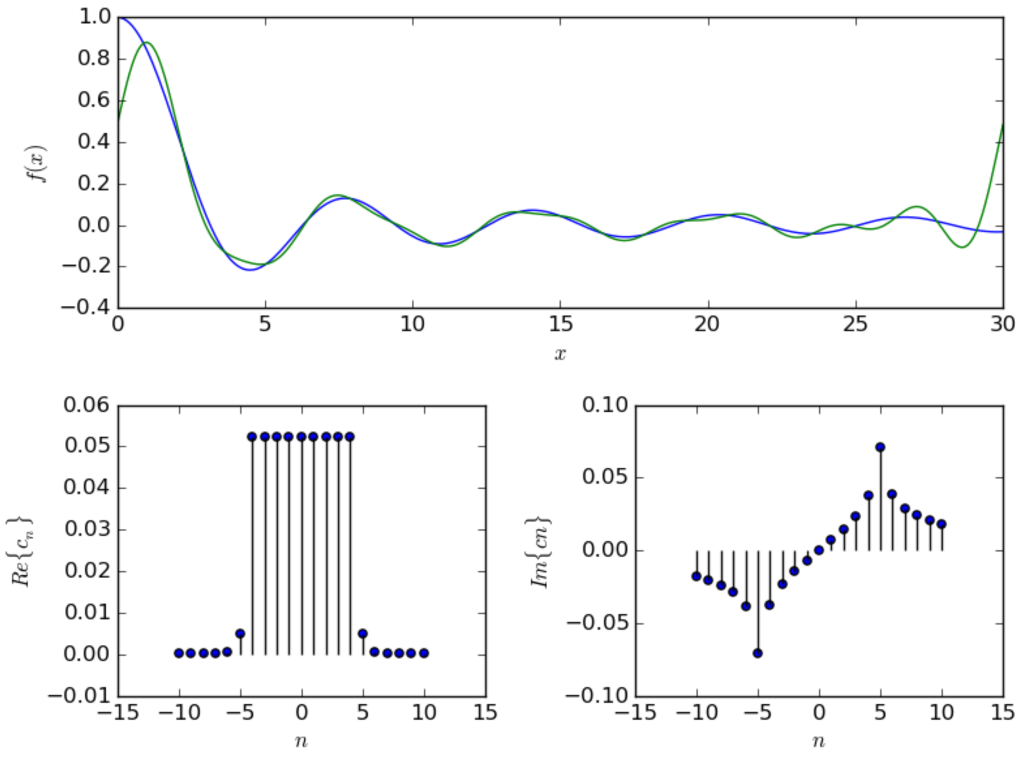

Figure 4.1: Output of Example 4.1.

Typically, when we work with a function $f(x)$, where x is a space coordinate, the Fourier transform is set in terms of the wave-number k as above. If our signal is a function of time, i.e. $f(t)$, the Fourier transform is usually set in terms of the angular frequency ω as $\tilde{f}(\omega) = F[f(t)]$.

4.1 General properties of the Fourier transform

There are many important properties of the Fourier transform that can be applied to solve physics problems. Here I'll list (without demonstrations) a few properties that will be useful for us in this Chapter.

Linearity

Let $f(x) = Ag(x) + Bh(x)$, the Fourier transform of $f(x)$ is $\tilde{f}(k) = A\tilde{g}(k) + B\tilde{h}(k)$.

Translation

For a constant shift x_0 in $f(x) = g(x-x_0)$, we get $\tilde{f}(k) = e^{-ix_0k}\tilde{g}(k)$. Equivalently, a phase set by k_0 as $f(x) = e^{ik_0x}g(x)$ leads to a Fourier transform shifted in the reciprocal space $\tilde{f}(k) = g(k-k_0)$.

Scaling

For a real number A in $f(x) = g(Ax)$, the Fourier transform scales as $\tilde{f}(k) = \frac{1}{|A|}\tilde{g}(k/A)$.

Derivatives

Let $f_n(x) = \frac{d^n}{dx^n} g(x)$ denote the n-th derivative of $g(x)$, the transform is $\tilde{f}_n(k) = (ik)^n \tilde{g}(k)$. Equivalently, for $f(x) = x^n g(x)$, the transform is $\tilde{f}(k) = i^n \frac{d^n}{dk^n}\tilde{g}(k)$.

Convolution

The convolution of the functions $g(x)$ and $h(x)$ is

$$f(x) = (g * h)(x) = \int_{-\infty}^{\infty} g(x')h(x-x')dx'. \tag{4.5}$$

The Fourier transform of a convolution is $\tilde{f}(k) = \sqrt{2\pi}\tilde{g}(k)\tilde{h}(k)$, which is simply the product of Fourier transformed functions. Conversely, if the function is the product of $g(x)$ and $h(x)$, i.e. $f(x) = g(x)h(x)$, the Fourier transform is $\tilde{f}(k) = (\tilde{g} * \tilde{h})/\sqrt{2\pi}$.

4.2 Numerical implementations

4.2.1 Discrete Fourier Transform (DFT)

The Discrete Fourier Transform is equivalent to the Fourier series applied on a discrete x lattice. It can be obtained from Eqs. (4.1)-(4.2) for $x_0 = 0$, $dx = \lambda/N$, $x \to x_n = n\lambda/N$, and converting the integral in Eq. (4.2) into a sum.

Let f be an array with N elements representing the function $f(x)$. The Fourier transform $\tilde{f}(k) = F[f(x)]$ is represented by an array $\tilde{f}$ whose elements are

$$\tilde{f}_m = \sum_{n=1}^{N} f_n \exp\left[-i\frac{2\pi(n-1)(m-1)}{N}\right]. \tag{4.6}$$

Conversely, given a vector $\tilde{f}$ representing a function $\tilde{f}(k)$ in k-space, the inverse Fourier transform $f(x) = F[\tilde{f}(k)]$ is represented by the array f with elements

$$f_n = \frac{1}{N}\sum_{m=1}^{N} \tilde{f}_m \exp\left[+i\frac{2\pi(m-1)(n-1)}{N}\right]. \tag{4.7}$$

Since we have to calculate all elements to compose the full array $\tilde{f}$ or f, the DFT or the inverse DFT takes N^2 operations. In contrast, the `Fast Fourier Transform` (FFT) algorithm takes $N\log_2 N$ operations, which is much smaller than N^2 for large N.

4.2.2 Fast Fourier Transform (FFT)

(TO DO) Here I'll follow the Radix-2 code of Cooley-Tukey: https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm

4.2.3 Julia's native FFT

Julia comes with a native implementation of FFT via the efficient FFTW library[1]. We have implicitly used the FFT before when using the command `conv` to perform convolutions. The main commands to remember are `fft`, `ifft`, `fftshift`, and `ifftshift`.

These functions act on arrays. Therefore, hereafter consider a discrete domain (`linspace`) x with spacing Δx and the array $f(x) \to f$ set on this domain.

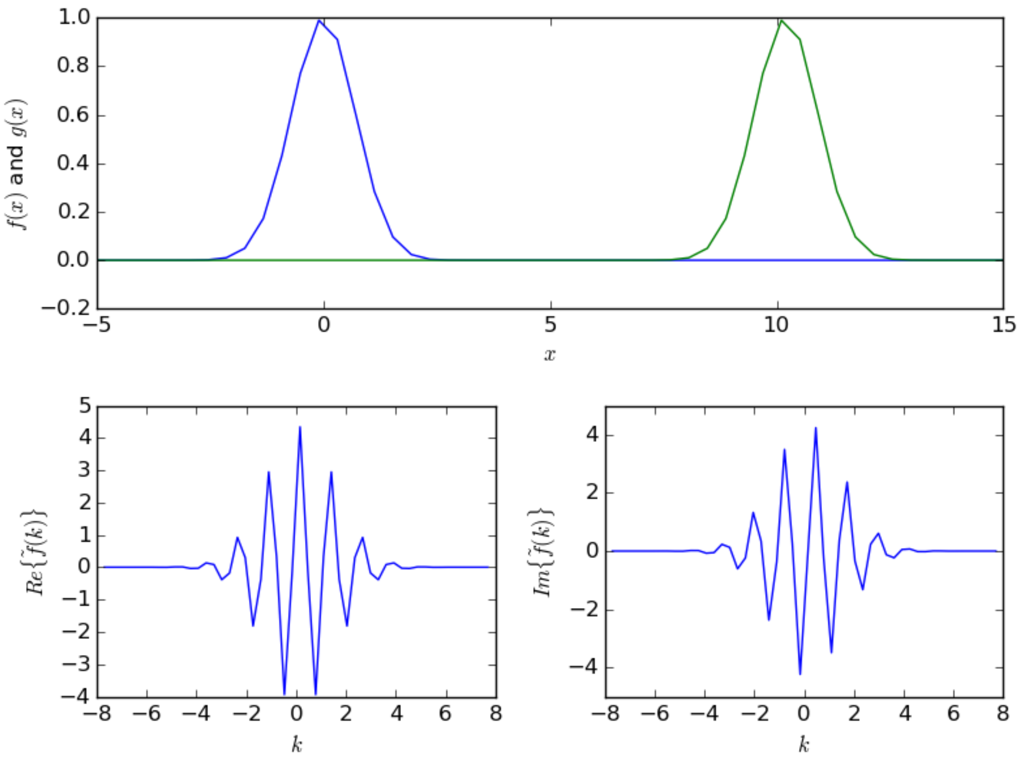

Figure 4.2: Output of Example 4.2.

First, the command `fft` acts on an array f and returns its Fourier transform $\tilde{f}$. Reciprocally, the command `ifft` acts on the array $\tilde{f}$ and returns its inverse Fourier transform f.

Notice that the array f represents the function $f(x)$ over the discrete domain set by the array x. Conversely, the array $\tilde{f}$ represents the Fourier transformed function $\tilde{f}(k)$. The FFT implementation defines the domain of reciprocal space as $k \in [0, \frac{2\pi}{\Delta x})$. However, due to the periodicity of the Fourier transforms, the region $\frac{\pi}{\Delta x} \le k < \frac{2\pi}{\Delta x}$ is actually equivalent to the region $-\frac{\pi}{\Delta x} \le k < 0$. Typically, it is more interesting to work on the reciprocal space in the symmetric domain $k \in [-\frac{\pi}{\Delta x}, \frac{\pi}{\Delta x})$.

One can use the commands `fftshift` and `ifftshift` to switch between these choices of domains above. The `fftshit` converts the domain from $k \in [0, \frac{2\pi}{\Delta x})$ into $k \in [-\frac{\pi}{\Delta x}, \frac{\pi}{\Delta x})$, while `ifftshift` performs the inverse operation.

[1]FFTW: http://www.fftw.org/

The next example uses the translation property of the Fourier transform to shift a function by x_0. Check also Problem 4.2.

Example 4.2: Fourier transform using the FFT

```
using PyPlot

n = 50; # using a small number of points for better visualization
x = linspace(-5.0, 15.0, n); # x domain
dx = x[2]-x[1]; # spacing Δx
k = linspace(-pi/dx, pi/dx, n); # k domain (symmetric)

# function in x-space
f = exp(-x.^2); # example function

# function in k-space
# and using fftshift to put ft in the symmetric domain
ft = fftshift(fft(f));

x0 = 10; # let's shift the function
gk = ft.* exp(-1im*x0*k); # see the Fourier transform properties
# goes back to the [0, 2pi/dx) k-domain and apply ifft
g = ifft(ifftshift(gk));
# g(x) = F^{-1}[e^{-i x_0 k} \tilde{f}(k)]

clf(); # the plot is shown in Fig. 4.2
subplot2grid((2,2),(0,0),colspan=2)
plot(x,f); # plot the original function
plot(x, real(g)) # and shifted one
xlabel(L"$x$");
ylabel(L"$f(x)$ and $g(x)$")

# plot the real part of the Fourier transform
# shifting to the symmetric k-domain
subplot2grid((2,2),(1,0))
plot(k, real(ft));
xlabel(L"$k$")
ylabel(L"$Re\{\tilde{f}(k)\} $")

# plot the imaginary part of the Fourier transform
# shifting to the symmetric k-domain
subplot2grid((2,2),(1,1))
plot(k, imag(ft));
xlabel(L"$k$")
ylabel(L"$Im\{\tilde{f}(k)\} $")

tight_layout();
```

4.3 Applications of the FFT

4.3.1 Spectral analysis and frequency filters

Let $\tilde{f}(\omega) = F[f(t)]$ be the Fourier transform of $f(t)$. The power spectrum of $f(t)$ is $S(\omega) = |\tilde{f}(\omega)|^2$. The plot of $S(\omega)$ *vs* ω shows us which components (frequencies ω) of the Fourier transform contribute to the signal $f(t)$. The power spectrum characterizes the color of a light beam, and the timbre of a musical instrument.

If our signal is simply $f(t) = A\sin(\omega_0 t)$, its Fourier transform is

$$\tilde{f}(\omega) = iA\sqrt{\frac{\pi}{2}}\Big[\delta(\omega-\omega_0) - \delta(\omega+\omega_0)\Big], \tag{4.8}$$

and the power spectrum will show two δ-peaks at $\omega = \pm\omega_0$. This is the expected solution of the simple pendulum in the harmonic regime. Check Problem 4.3.

Noise and frequency filters

What if our signal was noisy? There are many types of noise depending on their source[2]. Musical instruments may suffer from high frequency noises due to electronic systems, bad cables, etc. The type of noise is defined by its power spectrum. For instance, a *white noise* shows a constant power spectrum will all frequencies contributing equally. Hence the name *white noise* as a reference to white light. The power spectrum of a *pink noise* decays with $1/\omega$ and appears in most natural phenomena. The types of noise got their names as colors due to the initial analogy of white noise and white light.

In the next example, let's artificially generate and eliminate a white noise. The result is shown if Fig. 4.3. The white noise in this example have a power spectrum $S(\omega) \approx 10$, and two signal peaks show at $\omega = \pm 2\pi$. To filter the white noise we simply collect only the fft components that are above the white noise power spectrum.

[2]The colors of noise (Wikipedia)

Example 4.3: White noise

```
using PyPlot

N=300000; # generate signal with large number of points
t=linspace(0, 30, N); # time domain
dt=t[2]-t[1];
w=linspace(0, 2pi/dt, N); # frequency domain

w0 = 2pi; # original frequency of the signal
# signal + random fluctations:
f=(1+0.5*randn(N)).*sin((1+0.001*randn(N)).*w0.*t);

g = fft(f); # fft of the signal
s = log(abs2(g)); # power spectrum of the signal

g2 = g.*(s .> 15); # filters the spectrum above the white noise (~10)
s2 = log(1e-10+abs2(g2)); # spectrum of the filtered signal
f2 = ifft(g2); # recovers filtered signal with inverse FFT

clf();
subplot(211)
# first half shows original signal with noise
# second half shows filtered signal, now clean
n2 = Int(N/2);
plot(t[1:n2], f[1:n2])
plot(t[n2:end], f2[n2:end]);
xlabel(L"t")
ylabel(L"f(t)");

subplot(212)
# plot the power spectra
plot(w-pi/dt, fftshift(s));
plot(w-pi/dt, fftshift(s2));
xlim([-3w0, 3w0]);
xlabel(L"\omega")
ylabel(L"\logS(\omega)");

tight_layout();
```

In practice, each noise color requires a different method to be eliminated. Typical cases are the high-pass and low-pass filters, that apply frequency cuts to allow only high and low frequencies, respectively.

4.3.2 Solving ordinary and partial differential equations

There are many different approaches to solve partial differential equations (PDE), which we will discuss in Chapter 6. For now, we shall discuss only the a method using the Fourier transform. As paradigmatic examples, we will use the diffusion equation for the homogeneous case, and the Schrödinger equation for the inhomogeneous case. However, keep in mind that the method used in the Schrödinger equation is also valid for an inhomogeneous diffusion equation.

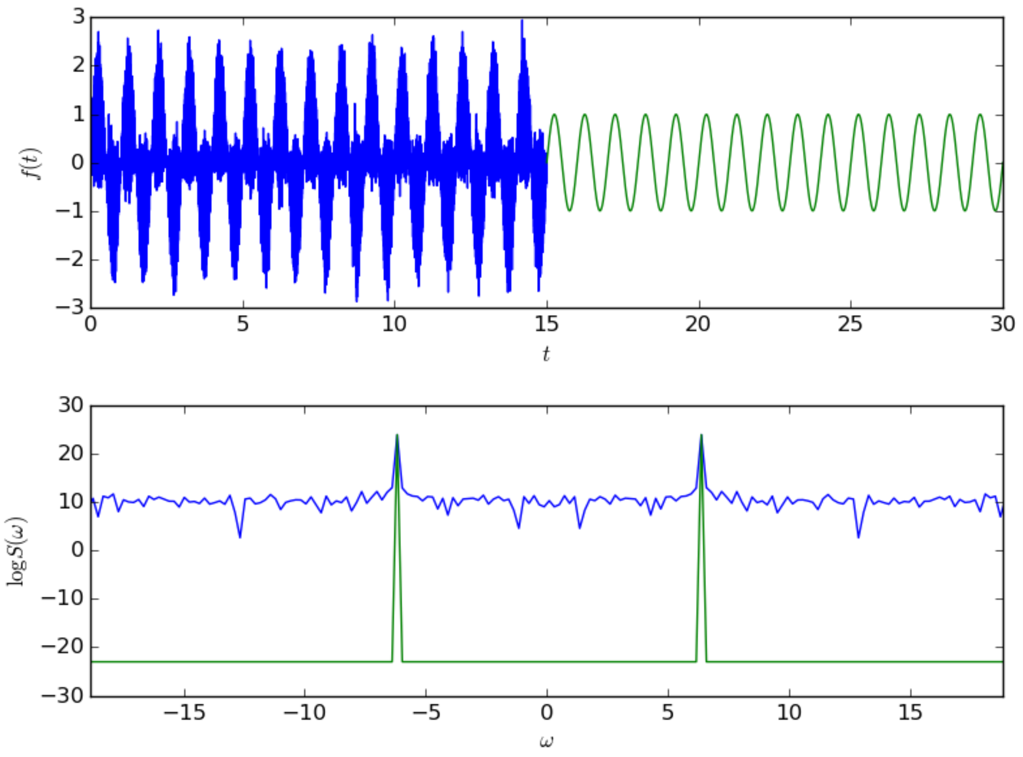

Figure 4.3: Output of Example 4.3.

Diffusion equation

Let's consider the homogeneous diffusion equation in one dimension,

$$\frac{\partial}{\partial t}u(x,t) - D\frac{\partial^2}{\partial x^2}u(x,t) + v\frac{\partial}{\partial x}u(x,t) = 0, \tag{4.9}$$

where D is the diffusion coefficient, v is the drift velocity and $u(x,t)$ is density of the diffusing material (mass, temperature, ...).

In this very simple homogeneous case, the diffusion equation has an analytical solution. For an initial condition $u(x,0) = \delta(x)$,

$$u(x,t) = \frac{1}{2\sqrt{\pi D t}}\exp\left[-\frac{(x-vt)^2}{4Dt}\right], \tag{4.10}$$

which is a Gaussian packet with a center that moves as $x = vt$, and broadens with time.

To find this solution, we first take the Fourier transform $x \to k$ of the diffusion equation,

$$\frac{\partial}{\partial t}\tilde{u}(k,t) + Dk^2\tilde{u}(k,t) + ivk\tilde{u}(k,t) = 0. \tag{4.11}$$

It is easy to show that the solution in k-space is $\tilde{u}(k,t) = e^{(-ivk-dk^2)t}\tilde{u}(k,0)$, where $\tilde{u}(k,0)$ is the Fourier transform of the initial condition $u(x,0)$. If the initial condition is $u(x,0) = \delta(x)$, then $\tilde{u}(k,0) = 1/\sqrt{2\pi}$. The inverse Fourier transform can be easily calculated analytically to obtain the solution $u(x,t)$ above.

In this simple case we don't really need any numerical calculation. However, you could eventually face a more general diffusion equation (inhomogeneous, vectorial, coupled, ...) for which numerical methods will be necessary. Therefore, let's illustrate the numerical method assuming that you are able to obtain the solution in k-space $\tilde{u}(k,t)$ analytically, but you are not able to calculate the Fourier transform of the initial condition $\tilde{u}(k,0) = F[u(x,0)]$ and the inverse Fourier transform to recover $u(x,t) = F^{-1}[\tilde{u}(k,t)]$. The next example solves this problem.

Example 4.4: Diffusion equation

```
using PyPlot

D = 1; # diffusion coefficient
v = 1; # drift velocity

N = 100; # number of points
L = -10;
x = linspace(-L, L, N); # x domain
dx = x[2]-x[1]; # x step
k = linspace(-pi/dx, pi/dx, N); # k domain (symmetric)

ux0 = exp(-x.^2); # initial condition u(x,0)
uk0 = fftshift(fft(ux0)); # initial condition in k-space ũ(k,0)

# analytical solution in k-space written with the numerical k vector
ukt(t) = exp((-1im*v*k-D*k.^2)*t).*uk0;

# the solution u(x,t) as a function that uses the inverse FFT
uxt(t) = ifft(ifftshift(ukt(t)));

# plot the resuts for different times t=0,1,2,3
clf();
plot(x, ux0);
plot(x, uxt(1));
plot(x, uxt(2));
plot(x, uxt(3));
xlabel(L"$x$");
ylabel(L"$u(x,t)$"];
```

Quantum operators in k-space, the split-step method

Trying to find an easy way to prove the $\mathcal{O}(\tau^3)$

Let $H(x,p,t) = T(p,t) + V(x,t)$ be a general Hamiltonian that can be written as a sum of two terms, where the first, $T(p,t)$, is a function of the momentum operator $p = -i\hbar\partial_x$ and the time t, while the second, $V(x,t)$, is a function of the position $x$ and time t. In general the commutator $[T,V] \neq 0$. We want to solve the time-dependent

Schrödinger equation

$$i\hbar\frac{\partial}{\partial t}\psi(x,t) = H(x,p,t)\psi(x,t). \tag{4.12}$$

For a small time step τ, an approximate numerical solution can be computed as

$$\psi(x,t+\tau) \approx e^{-i\frac{V\tau}{2\hbar}} e^{-i\frac{T\tau}{\hbar}} e^{-i\frac{V\tau}{2\hbar}} \psi(x,t). \tag{4.13}$$

Here we start with half of a time step, $\tau/2$ using V, followed by a full time step τ using T, and another $\tau/2$ to complete the evolution from t to $t+\tau$. This spitting of the operators generates an error of order $\mathcal{O}(\tau^3)$.

More interestingly, this approach can be extremely efficient if we notice that T is a function of momentum p, and, therefore, can be better represented in Fourier space (see the derivative property of the Fourier transform). The trick is to carry all operations using V in coordinates space, while those involving T are done in k-space. In terms of Fourier transforms, the evolution reads

$$\psi(x,t+\tau) \approx \exp\left(-i\frac{V\tau}{2\hbar}\right) F^{-1}\left[\exp\left(-i\frac{\tilde{T}\tau}{\hbar}\right) F\left[\exp\left(-i\frac{V\tau}{2\hbar}\right)\psi(x,t)\right]\right], \tag{4.14}$$

where $\tilde{T} \equiv \tilde{T}(k,t)$ is the Fourier transform $p \to \hbar k$ of the $T(p,t)$ operator.

Notice that $V(x,t)$ becomes a diagonal matrix when we discretize x, but $T(p,t)$ is non-diagonal, as it contains derivatives in the momentum operator. However, in k-space, $\tilde{T}(k,t)$ is diagonal. Therefore, using the Fourier transform as in the equations above, all exponentials will be defined in terms of diagonal matrices, which is easy to calculate. In fact, calculating the exponential of a general matrix is much less efficient than the FFT. Hence the high efficiency of the split-step method. The next example illustrates the difference between the coordinate- and k-space calculations of $\phi = \exp\left(-i\frac{T\tau}{\hbar}\right)\phi_0$, where $T = -\frac{1}{2}p^2$.

4.4 Problems

Problem 4.1: Discrete Fourier Transform

Implement your own version of the Discrete Fourier Transform (DFT) in Julia and compare the results with Example 4.2. Your implementation will not be efficient and you shall use Julia's native `fft` function always. However, implementing the DFT will help you learn more about coding.

Problem 4.2: Derivatives with the FFT package

Check again Example 4.2 and adapt it to use the derivative property of the Fourier transforms to calculate the first, second, and third derivatives of a function $f(x)$ set on a discrete domain. Choose a simple function $f(x)$, so that you can find its derivatives analytically and compare with the numeric results.

Problem 4.3: Pendulum: small vs large amplitudes

In the previous chapter, you have written a code to solve the differential equation of a pendulum,

$$\frac{d^2}{dt^2}x(t) = -\omega_0^2 \sin[x(t)] - \gamma v_x(t). \tag{4.15}$$

For now, let's neglect the damping term, $\gamma = 0$. Consider $\omega_0 = 2\pi$ and the initial conditions $x(0) = x_0$, and $v(0) = 0$. Prepare your code to run over $0 \le t \le 100$.

(a) Find $x(t)$ for small amplitudes set by $x_0 = 0.01\pi$ and use the `fft` command to obtain $\tilde{x}(\omega)$. Plot the spectral function $S(\omega) = |\tilde{x}(\omega)|^2$ for $-2\omega_0 \le \omega \le 2\omega_0$.

(b) Do the same for large amplitudes set by $x_0 = 0.99\pi$.

(c) Discuss the differences between the power spectra of cases (a) and (b).

Problem 4.4: Pendulum: large amplitudes and damping

Consider the same equation an parameters from the previous problem. But now let's start with a large amplitude $x_0 = 0.99\pi$, and a finite damping $\gamma = 0.001$. Solve the differential equation for $0 \le t \le 6000$ with a large number of points ($\sim 10^5$). As a function of t you shall see the oscillations loosing amplitude due to the damping.

(a) Plot the spectral function $S(\omega)$ using only the numerical data of $x(t)$ for small t, say in a window $0 \le t \le 500$.

(b) Plot $S(\omega)$ for the data at large t, say in a window $5500 \le t \le 6000$.

(c) Explain the difference between the spectral functions.

Problem 4.5: Driven pendulum

Let's analyze the power spectrum of a driven pendulum[6]. Consider a driven pendulum set by

$$\frac{d^2}{dt^2}x(t) = -\omega_0^2 \sin[x(t)] - \gamma v_x(t) + f_0 \cos(\omega_1 t). \tag{4.16}$$

(a) Solve this differential equation for $x(0) = 0$, $v_x(0) = 2$, and $0 < t < 3000$, using the parameters $\omega_0 = 1$, $\omega_1 = 2/3$, $\gamma = 1/2$, and $f_0 = 0.9$.

(b) Now try it with the same parameters, except for $f_0 = 1.15$.

(c) The power spectrum is $S(\omega) = |\tilde{x}(\omega)|^2$, where $\tilde{x}(\omega)$ is the Fourier transform ($t \to \omega$) of $x(t)$. Compare the power spectra of cases (a) and (b). The first is periodic, and will show narrow peaks at the motion frequencies. The second is chaotic, almost all frequencies contribute and the power spectrum is fractal.

CHAPTER 5

Statistics (TO DO)

`What are the odds of having a silly quote here?`

5.1 Random numbers

5.2 Random walk and diffusion

5.3 The Monte Carlo method

CHAPTER 6

Partial Differential Equations (TO DO)

6.1 Separation of variables

6.2 Discretization in multiple dimensions

6.3 Iterative methods, relaxation

6.4 Fourier transform (see §4.3.2)

CHAPTER 7

Plotting (not finished)

`... is worth a thousand words`

JULIA has a few interesting packages for plotting:

- `PyPlot`[1]: provides an interface to Python's MatPlotLib;
- `Gaston`[2]: provides a `gnuplot` interface to Julia;
- `Gadfly`[3]: "The Grammar of Graphics";
- `Winston`[4]: brings the MATLAB syntax to Julia plots.

Here we will discuss `PyPlot` to take advantage of its extensive documentation and large number of users, making it easy to find examples online.

7.1 PyPlot

7.1.1 Installing PyPlot

To install `PyPlot`, first you need Python and Matplotlib. If you use Ubuntu Linux, run

```
$ sudo apt-get update
$ sudo apt-get install python python-matplotlib
```

Now you can install PyPlot. Start Julia and run: `julia> Pkg.add("PyPlot")`.

[1]PyPlot: https://github.com/stevengj/PyPlot.jl
[2]Gaston: https://github.com/mbaz/Gaston.jl
[3]Gadfly: http://dcjones.github.io/Gadfly.jl/
[4]Winston: https://github.com/nolta/Winston.jl

7.1.2 Using PyPlot

To use `PyPlot`, first you have to initialize it by calling: `julia> using PyPlot`.

```
Example 7.1: Using PyPlot

using PyPlot
x = linspace(-3pi,3pi,200);
plot(x,  sin(x)./x, color="red", linewidth=2.0, linestyle="-");
plot(x, -sin(x)./x, color="blue", linewidth=2.0, linestyle="--");
xlabel("x");
ylabel("f(x)");
```

Since the `PyPlot` package is an interface to Python's Matplotlib, one may use its extensive documentation[5].

7.1.3 Most useful features and examples

Calling non-ported functions from Matplotlib

PyPlot's documentation says that only the documented Python's `matplotlib.pyplot` API is exported to Julia. Nonetheless, other functions from Python's `PyPlot` can be accessed as

$$\texttt{matplotlib.pyplot.foo(...)} \longrightarrow \texttt{plt[:foo](...)},$$

where on the left we have Python's notation, and on the right Julia's.

For instance, you can generate an histogram using:

```
Example 7.2: Using non-ported functions from Matplotlib

using PyPlot
x = randn(100000);
plt[:hist](x);
xlabel("Random number");
ylabel("Number of occurrences");
```

Latex

It is very easy to use Latex with Julia's PyPlot package. Implicitly, PyPlot uses the LaTexStrings package[6]. Therefore on can use Latex commands on a string simply constructing it prepending with a L. For instance:

[5] Matplotlib's documentation: http://matplotlib.org/
[6] LaTeXStrings: https://github.com/stevengj/LaTeXStrings.jl

Example 7.3: Using Latex

```
using PyPlot

x = linspace(0, 2pi, 100);
f = sin(x);

plot(x, f);
xlabel(L"$\theta$ [rads]");
ylabel(L"$\sin\theta$ [a.u.]");
```

This Latex notation can be used in all text elements of PyPlot.

Subplot

The `subplot` command allows you to break the plot window into a grid. Say that the desired grid has *N* lines and *M* columns, the command `subplot(N, M, j)` specifies that you want to plot the next figure into the *j*-th element of the grid, with the elements sorted in a "Z" shape. Try the next example.

Example 7.4: Using subplot

```
using PyPlot

x = linspace(-5pi, 5pi, 1000);

clf();

subplot(2,2,1);
plot(x, sin(x));
xlabel(L"$x$");
ylabel(L"$\sin(x)$");

subplot(2,2,2);
plot(x, cos(x));
xlabel(L"$x$");
ylabel(L"$\cos(x)$");

subplot(2,2,3);
plot(x, exp(-x.^2));
xlabel(L"$x$");
ylabel(L"Gaussian");

subplot(2,2,4);
plot(x, sin(x)./x);
xlabel(L"$x$");
ylabel(L"$sinc(x)$");

tight_layout(); # adjust the subplots to better fit the figure
```

In the example above we are using Latex notation for the labels. The final command **tight_layout(...)** allows you to adjust the subplots spacings to better adjust them within the figure. This avoids overlapping of the labels and plots.

Even more control can be achieved with the command **subplot2grid(...)**, which we use in Fig. 4.1, for instance. Here the shape is passed as a tuple (N, M) to the first parameter, the second parameter takes the location of the desired plot an ordered pair (i, j), where $0 \le i \le (N-1)$ and $0 \le j \le (M-1)$. The top-left corner grid element is $(i, j) = (0, 0)$, and the bottom-right is $(i, j) = (N-1, M-1)$. The `subplot2grid` becomes more interesting as the next parameters allow you to set a `rowspan` and a `colspan` to span over cells. Check again Example 4.1.

Labels

Relevant commands: text

Legends

Relevant commands: legend

Other plot elements

Relevant commands: annotate, arrow, axes, axis, axhline, axvline, contour,

Saving into files (PNG, PDF, SVG, EPS, ...)

Figures are shown by default on screen as PNG. However, you can save it to files with many different formats using the `savefig(...)` command. Check the example:

Example 7.5: Using subplot

```
using PyPlot

x = linspace(-5pi, 5pi, 1000);
f = sin(x);
plot(x, f);
xlabel(L"$\theta$ [rads]");
ylabel(L"$f(\theta) = \sin\theta$");

savefig("myplot.png");
savefig("myplot.pdf");
savefig("myplot.svg");
savefig("myplot.eps");
```

Please check the full documentation of the `matplotlib` for more details. You can choose the paper size, dpi, etc.

Animations

Here's an example of using Julia and `matplotlib` to create animations[7]. The `PyCall` package is used to import the `animation` library from Python, since it is not natively implemented in `PyPlot`.

Example 7.6: Animated plot

```julia
using PyCall # package used to call Python functions
using PyPlot

# import the animation library
@pyimport matplotlib.animation as anim

# this function must plot a frame of the animation
# the parameter t will be iterated by the matplotlib
function animate(t)
  # in this example we willl plot a Gaussian moving around
  c = 3*sin(2pi*t);
  x = linspace(-10, 10, 200);
  y = exp(-(x-c).^2/2);

  # clear the figure before ploting the new frame
  clf();

  # plot the new frame
  plot(x,y)

  # for this basic usage, the returned value is not important
  return 0
end

# matplotlib's FuncAnimation require an explicit figure object
fig = figure();

tspan = linspace(0, 5, 100); # will be used to iterate the frames

# animation with a interval of 20 miliseconds between frames
video = anim.FuncAnimation(fig, animate, frames=tspan, interval=20)

# save the animation to a MP4 file using ffmpeg and the x264 codec
video[:save]("anim.mp4", extra_args=["-vcodec", "libx264"])
```

[7]Adapted from Andee Kaplan's presentation at http://heike.github.io/stat590f/gadfly/andee-graphics/

CHAPTER 8

Other topics (TO DO)

`. . . and yet another silly quote`

8.1 Linear algebra

8.2 Root finding

8.3 Linear systems

8.4 Least squares

Bibliography

[1] Cláudio Scherer. *Métodos computacionais da Física.* Editora Livraria da Física, São Paulo, 2010.

[2] Neide Bertoldi Franco. *Cálculo numérico.* Pearson Prentice Hall, São Paulo, 2006.

[3] Selma Arenales and Artur Darezzo. *Cálculo numérico: aprendizagem com apoio de software.* Thomson Learning, São Paulo, 2008.

[4] J. C. Butcher. *Numerical methods for ordinary differential equations.* John Wiley & Sons Ltd., Chichester, England, 2008.

[5] Jos Thijssen. *Computational physics.* Cambridge University Press, 2007.

[6] Tao Pang. *An introduction to computational physics.* Cambridge University Press, 2006.

[7] Edwin Abbott Abbott. *Flatland: A romance of many dimensions.* Princeton University Press, 2015.

[8] Thomas Haigh, Mark Priestley, and Crispin Rope. *Eniac in Action: Making and Remaking the Modern Computer.* Mit Press, 2016.

[9] Herch Moysés Nussenzveig. *Curso de física básica.* Edgard Blücher, 2002, 2007.