Contents

Introduction

This tutorial is designed to introduce students to the Julia programming language and its applications in spectroscopy. Lessons are more like lecture notes. It is intended for students who have never programmed before, or who have only done a little programming in another language. The goal is to teach the basics of programming in Julia, and to provide examples of how to use Julia for data analysis and visualization in spectroscopy. It is not intended to be a comprehensive introduction to Julia, but rather a starting point for students to learn how to use Julia for their own research projects in spectroscopy.

Getting started

What is Julia?

The Julia programming language is a high-level, general purpose programming language designed for high-performance numerical and scientific computing. It is by default a just-in-time (JIT) compiled language, meaning that it compiles code as you need it.

The Julia community is small, but it has found a niche in scientific computing. It is used by CERN to analyze data from the Large Hadron Collider, and by the NASA Jet Propulsion Laboratory for modeling spacecraft and data analysis. It is also used by many pharmaceutical companies for drug discovery and development.

Why use Julia?

Any software can of course be used to analyze data, but I choose Julia for a few reasons.

1. It is designed specifically for scientific applications and has many robust scientific libraries.
2. It is easy to learn, borrowing good ideas from Python and Matlab.
3. It is responsive and interactive. It can be used in Jupyter notebooks or in the REPL (Read Evaluate Print Loop) in the command line. The VS Code extension for Julia also provides an interactive environment for writing and running Julia code.
4. It is fast, comparable to C or Fortran. So if performance is your goal, you can stick with Julia and do not need to learn a second language.
5. It is free, open source, and has a large and growing community. A modern, free, powerful, and simple programming language with a great selection of packages is a tremendous asset for any lab.
6. Reproducibiliy is a priority, and Julia has a built-in package management system that makes it easy to share code and reproduce results. This is essential when designing code for science.
7. I was sold on the excellent plotting library, Makie.jl. It is fast, interactive, and has a clean API.

Install Julia and Visual Studio Code

1. Install the latest version of Julia following the instructions on the official website.
2. Download and install Visual Studio Code.
3. Install the Julia extension for Visual Studio Code by searching for "Julia" in the Extensions Marketplace.

We will first use the Julia REPL (Read-Eval-Print Loop) in the terminal. VS Code will be used later. There are many code editors available, but we will use VS Code because it is free and widely used for many programming languages.

The Command Palette Briefly explain the command palette and how to use it in VS Code.

Exercises

1. Open the Julia REPL in the terminal by typing `julia` in the terminal.
2. Type `versioninfo()` in the REPL to check that Julia is installed correctly.
3. Your first program in Julia: type `print("Hello, world!")` in the REPL to print "Hello, world!" to the console.

Files, folders, and environments

The concept of files and folders are reviewed and then a tour of Visual Studio Code is given. Students create a project folder to store their tutorial files and future experiment code. This ensures consistent data storage and analysis practices in the lab. We also cover environments and how to create a new environment for each project. This is important for reproducibility and to avoid package conflicts.

To the instructor: Do not show your actual experiments folder, unless it is *very* well organized (to avoid confusion). Follow along with these instructions with the students.

Files and folders on a computer

On a computer, files are stored in folders (directories), where a folder can contain both files and other folders. We want to store files and folders in a way that makes it easy to find and organize them. It is important in science to have well-organized data and code for reproducibility.

Follow these steps to create a project folder for your tutorials and experiments:

1. On macOS, go to `~/Documents/` and create a new folder called `projects`. On Windows go to `C:\Users\<username>\Documents\` and create a new folder called `projects`. This is where you will store your projects while you are in the lab.
2. Inside of `projects`, create a new folder called `tutorials`. This is where you will store code for these tutorials.

3. Open Visual Studio Code and open the `tutorials` folder that you just created. This folder will be used for all tutorials, including analysis in the optics tutorials later.
4. Click on the new folder icon and make a new folder called `programming` or something similar.
5. Create a new file called `functions.jl` in the `programming` folder you just created. This is where you will write your code for the programming tutorial, including problems.

Please continue to use this tutorials folder for your analysis when you do experiments in later tutorials (by creating an `FTIR` folder for the FTIR lesson, for example). Next we will create a new environment for these lessons.

Environments

In Julia, an environment is a collection of packages and their versions that are used for a specific project. This is important for reproducibility and to avoid package conflicts. The packages you use in these tutorials might be different from your experiments. Also, loading lots of packages can slow down compilation.

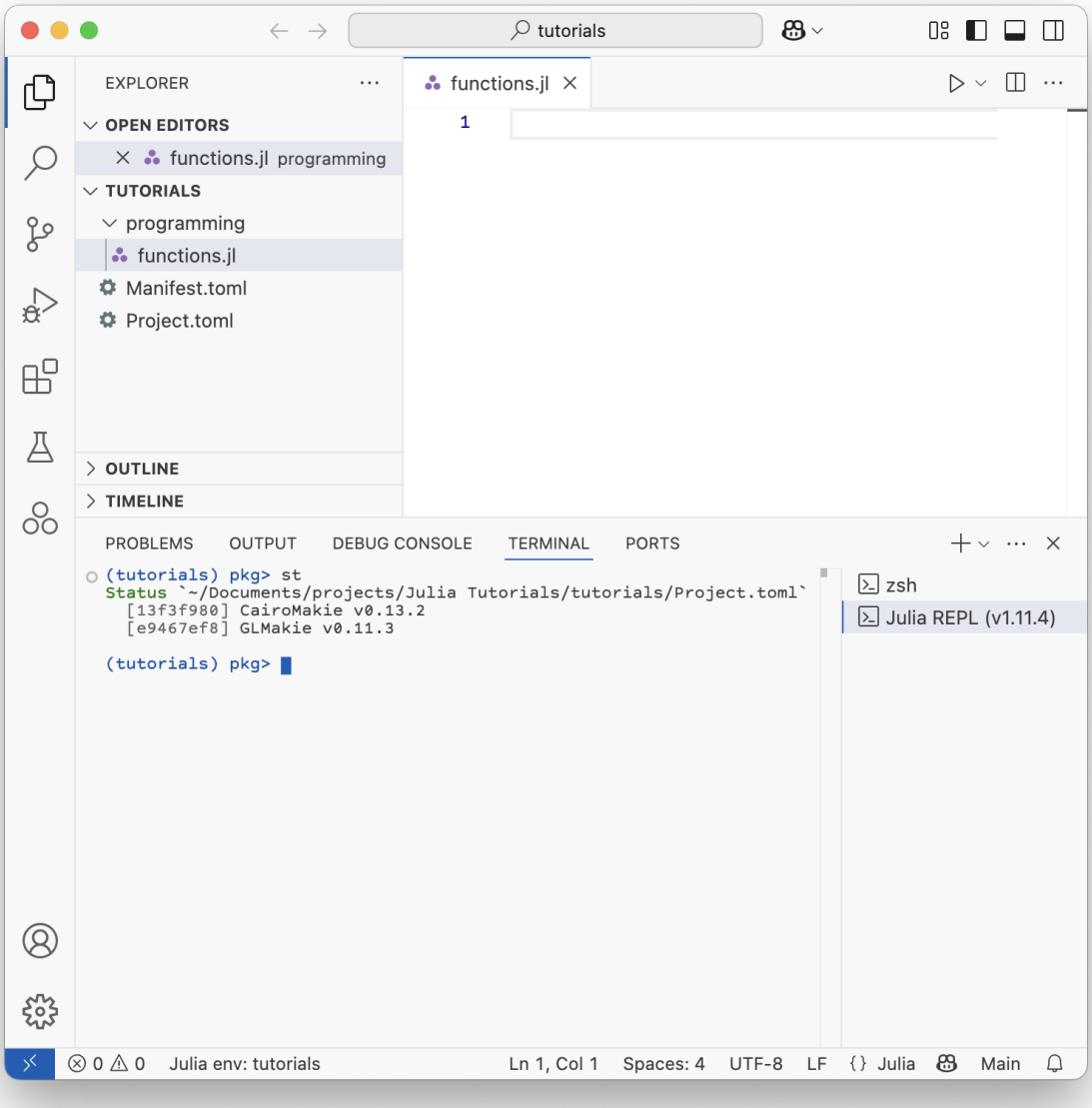
First look in the bottom left corner of the Visual Studio Code window. Notice that it say "Julia env: v1.11" or something similar. This means that you are using the default Julia environment, which is the global environment. In general, we don't want to use the global environment for our projects. Sometimes I use the global environment for quick tests, but I always create a new environment for my projects.

To create a new environment, follow these steps: 1. Open the Julia REPL in Visual Studio Code via the Command Palette (Ctrl+Shift+P) and find `Julia: Start REPL`. 2. In the REPL, type `]` to enter the package manager. (Hit backspace to exit the package manager and return to the Julia REPL.) Notice that here too, it says `(@v1.11) pkg>` or something similar, indicating that you are in the global environment. 3. Type `activate .` to create a new environment in the current folder. Now it says `(tutorials) pkg>`, indicating that you are in the new environment. 4. Now let's add the plotting package that we will use later, `GLMakie` and `CairoMakie`. I will explain what these are when we start to use them.

Notice that a `Project.toml` file and a `Manifest.toml` file were created in your project folder. These files contain information about the packages and their versions that are used in this environment. This is how Julia keeps track of the exact versions of the packages you are using and ensures that your code will work in the future. This is important for reproducibility and sharing with other researchers. If you close and reopen VS Code, the environment indicator in the lower left will say "Julia env: tutorials".

When Makie and its dependencies have finished installing and compiling, let's go back to the REPL by hitting backspace and learn about functions.

Below is what the student environment and file structure should look like after setup.



Exercises

1. Download the Japanese language extension for VS Code.
2. Find the setting to change the VS Code theme from the command palette.
3. Try to open a folder from the command palette.

Problems

1. In your `tutorials` folder, create a file called "hello_world.jl" and type `println("Hello, world!")` in the file. Hit Shift+Enter to run this line of code.
2. Check out the Julia documentation at docs.julialang.org. What kinds of things can you find there?

Variables, operators, and types

Open the REPL (the command line interface for Julia) by typing `julia` in the terminal. REPL stands for Read-Eval-Print Loop.

Follow along with the code examples below in the REPL. Adapt the code based on student feedback and progression. Remember to ask students to try things out themselves and play with the code. Use the question, "What do you think will happen?" to encourage students to think about the code before running it. Also a useful way of thinking about their optics setups later.

Assignment and basic operations

Start with basic operations on one variable. In programming, `=` is used for assignment, not equality.

For example, `a = 2` means "assign the value 2 to the variable `a`". In programming, we can write the following:

```
a = a + 1
```

This means "assign the value of `a + 1` to `a`". In mathematics, this makes no sense, but in programming, it is perfectly valid.

Here are some basic operations you can do with numbers in Julia:

```
julia> a = 2  # assignment
2

julia> a  # print the value of a
2

julia> a + 1  # addition
3

julia> a - 1  # subtraction
1

julia> a * 2  # multiplication
4

julia> a^2  # exponentiation
4

julia> a / 2  # division
1.0  # what's going on here?

julia> a % 2  # modulo
0
```

Exercises

1. Find the value of x at the end of this block of code.

   ```
   x = 3
   y = x
   x = x + 1
   x = y
   ```

2. What happens if you try to use a variable that has not been defined yet?

```
cats = 5
Cats
```

Types

We can use the `typeof` function to check the type of a variable. Now let's create some variables of different types and explore Julia's type system.

```
julia> a = 2
2

julia> b = 2.0
2.0

julia> c = 3.0 + 4.0im  # complex number
3.0 + 4.0im

julia> d = "hello"  # double quotes for strings
"hello"

julia> e = 'a'  # single quotes for characters
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> typeof(a)
Int64

julia> typeof(b)
Float64

julia> typeof(c)
ComplexF64 (alias for Complex{Float64})

julia> typeof(d)
String
```

Exercises

1. Try operations with different types (like `1 + 2.0`). What happens?
2. What is the type of operators like `*` and `+`?

Strings

Text data is represented as a sequence of characters called a string. String can be operated on, but the operations are different from those for numbers.

```
julia> s = "hello"
"hello"

julia> s * s  # string concatenation
"hellohello"

julia> s^3
"hellohellohello"
```

There are also multi-line strings, which are defined with triple quotes.

```
"""
This is a multi-line string.
It can span multiple lines.
It is useful for writing long text or documentation.
"""
```

We can also return a character from a string by indexing. We can get the third to eighth characters of a string by using the colon operator `:`.

```
julia> s = "Hello world"
"Hello world"

julia> s[1]  # first character
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)

julia> s[3:8]
"llo wo"
```

String interpolation

String interpolation is a way to include variables or expressions inside a string. You can use the `$` symbol to interpolate a variable or expression into a string.

```
julia> name = "Alice"
"Alice"

julia> age = 30
30

julia> "My name is $name and I am $age years old."
"My name is Alice and I am 30 years old."
```

You can also use `$()` syntax to include expressions inside a string.

```
julia> "In 5 years, I will be $(age + 5) years old."
"In 5 years, I will be 35 years old."
```

Exercises

1. What happens when you type `a = hello` (no quotes)? What is the meaning of the single and double quotes?
2. What happens if you try to multiply a string by a number? Read the error message.
3. What does the function `string` do? Try it on a number, a string, and a variable.

Problems

1. In a new file, calculate how many seconds there are in 3 years. Use variables to assign intermediate values.
2. In Julia, `pi` is a constant that represents the value of π (3.14159...). You can also use π (the Greek letter pi) by typing `\pi` and pressing `Tab`. Calculate the area of a circle with radius 5 using `pi` or π. Assign the radius to a variable and use it in the calculation. Make sure to assign the result to a variable, also.

Conditional execution

Boolean expressions

A boolean expression is a statement that is either `true` or `false`.

```
julia> t = true
true

julia> f = false
false
```

We can use the `typeof` function to check the type of a boolean variable.

```
julia> typeof(true)
Bool

julia> typeof(false)
Bool
```

Comparison operators

Comparison operators are used to compare values and return a boolean value. Let `a = 1` and `b = 2`. The comparison operators are:

```
julia> a, b = 1, 2
(1, 2)

julia> a == b  # equal to
false

julia> a != b  # not equal to
true

julia> a < b  # less than
true

julia> a >= b  # greater than or equal to
false
```

Exercises

1. What is the type result returned by `1 == 2`?
2. What is the difference between == and ===?
3. Make the statement `"Hello world"[i:j] == "o wo"` return `true`.

Logical operators

Logical operators are used to perform logical operations on boolean values. There are three logical operators in Julia: && (AND), `||` (OR), and `!` (NOT). The logical operators are used to combine boolean expressions.

```
julia> t && f  # logical AND
false

julia> t || f  # logical OR
true

julia> !t  # logical NOT
false
```

For complicated expressions, we can use parentheses to group operations.

```
julia> (a < b) && (b > 0)  # true
true
```

Exercises

1. Let `x = 5` and `y = 10`. What is the result of the following expression?

   ```
   !(x > 0) && y < 0
   ```

If statements

If statements allow you to execute code conditionally based on a boolean expression. For example, you can use `if` to check if a number is greater than zero.

```
x = 5
if x > 0
    println("x is positive")
end
```

This statement will print "x is positive" because `x` is greater than zero. If `x` were less than or equal to zero, the code inside the `if` block would not execute. Nothing happens. Usually, you want to do something if the condition is false, so you can use `else` to specify an alternative block of code.

```
x = -5
if x > 0
    println("x is positive")
else
    println("x is not positive")
end
```

There is a third option called `elseif` that allows you to check multiple conditions. Let's use this to check if a number is positive, negative or zero.

```
x = 0
if x > 0
    println("x is positive")
elseif x < 0
    println("x is negative")
else
    println("x is zero")
end
```

Exercises

1. Write a program that prints a message basd on the temperature. For example:

   Below 0: "It's freezing!"
   Between 0 and 20: "It's cold."
   Between 20 and 30: "It's warm."
   Above 30: "It's hot!"

Problems

1. Complete the following program so that it prints only the even numbers between 1 and 10 that are not divisible by 4.

   ```
   for i in 1:10
       if # Your code here. This should return `true`.
           println(i)
       end
   end
   ```

2. Try fixing the logic in this conditional. It's supposed to check if x is between 10 and 20, inclusive. Why doesn't the original code work as expected?

```
x = 15
if x > 10 || x < 20
    println("x is between 10 and 20")
else
    println("x is not between 10 and 20")
end
```

Iteration

While statements

While loops allow you to execute a block of code as long as a condition is true. The code block below will print the numbers from 1 to 10 by incrementing a variable `i` by 1 with each iteration.

```
n = 5
while n > 0
    println(n)
    n = n - 1
end
println("Blast off!")
```

While loops can be tricky because if the condition never becomes false, the loop will run indefinitely.

```
while true
    println("This will run forever!")
end
```

For statements

For loops allow you to iterate over a range of values or elements in a collection.

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits
    println(fruit)
end
```

In Julia, the `:` operators can be used to create ranges. For example, `1:5` creates a range from 1 to 5. Let's use this notation to print the numbers from 1 to 10.

```
for i in 1:10
    println(i)
end
```

Problems

1. Write a function that take a positive integer and returns the sum of all integers from 1 to that number using a `for` loop.

Functions

Functions are used to organize code and make it reusable. Once it is written, you don't have to think about how it works. Solving a large problem is much easier by building up from smaller functions that perform specific tasks.

Functions in programming operate similarly to mathematical functions. They take inputs (arguments) and produce outputs (return values). The difference is that programming functions can take and return more than just numerical values. They perform operations on data, manipulate variables, and control the flow of a program.

Built-in functions

Julia has many built-in functions, and you can also define your own functions. We have already seen basic functions like * and + for multiplication and addition. Other built-in mathematical functions include: `sin`, `cos`, `exp`, `log`, `sqrt`, `abs`, `round`, `floor`, `ceil`, `max`, `min`. Non-numerical functions include: `length`, `size`, `typeof`, `print`, `println`, `push!`, `pop!`, `sort`, `reverse`, and many more.

Play around with these functions in the REPL.
(To instructor: you don't have to do all of the below examples. They are just to show the variety of built-in functions available.)

```
julia> round(3.14159, digits=2)  # This function has two arguments (inputs).
3.14

julia> floor(3.14159)
3.0

julia> max(3, 5)
5

julia> reverse("Hello, world!")
"!dlrow ,olleH"
```

User-defined functions

User-defined functions are the heart of programming. This is the first step towards writing code to solve complex problems, automate tasks, and share your work with others.

You can define your own functions in Julia using the `function` keyword. Let's define a simple function that takes a number as input and returns that number plus one.

```
function add_one(x)
    return x + 1
end

y = 10 + add_one(5)
```

The `return` keyword can be omitted in Julia if it is the last line of the function. The block above can be rewritten as

```
function add_one(x)
    x + 1
end
```

You can also define a function in a single line using the = operator.

```
add_one(x) = x + 1
```

This shorthand way makes mathematical functions easier to read and write. They look just like regular mathematical notation.

```
f(x) = x^2 + 2x + 1
```

A function does not have to return anything. It can just perform an action.

```
function print_twice(x)
    print(x)
```

```
    print(x)
end

print_twice("Hello")
```

The variable x is a parameter and it is a dummy variable just like in a mathematical function, like the x in $f(x) = x^2$. When you call the function, you can pass in any value for x, and the function will use that value in its calculations. It is internal to the function and only exists within the function's scope.

Operators like * and + are also functions in Julia, but have a special syntax called *infix notation*. For example, you can write `3 * 5` or `*(3, 5)`.

Let's try another example that squares a number.

```
julia> square(x) = x^2  # function definition
square (generic function with 1 method)

julia> square(3)
9

julia> square(4.0)
16.0

julia> square(4.0 + 2.0im)
12.0 + 16.0im
```

You can see that the function automatically works with different types of inputs. This is one of the powerful features of Julia: it can automatically determine the type of the input and return the appropriate type for the output. This is allowed because of Julia's "multiple dispatch" functionality and is a key feature of Julia's design.

Exercises

1. Write a function checks if a number is even. It should return `true` if the number is even and `false` otherwise.

   ```
   is_even(x) = # your code here
   ```

2. Write a function that takes two strings and joins them together with a space in between.

   ```
   join_with_space(a, b) = # your code here
   ```

Problems

1. Calculate the energy in eV for photons of wavelengths 620 nm, 310 nm, and 1240 nm. "'julia function photon_energy(wavelength_in_nm) # Your code here end

   using Test @test wavelength_to_ev(620) ≈ 2.0 atol=1e-2 @test wavelength_to_ev(310) ≈ 4.0 atol=1e-2 @test wavelength_to_ev(1240) ≈ 1.0 atol=1e-2 "'

2. Write a function that returns the quadrant (1, 2, 3, 4) of a point (x, y) in 2D Cartesian space.

   Bonus: What should the function return if the point is on an axis or the origin?

   ```
   function quadrant(x, y)
       # add code here
   end

   using Test
   @test quadrant(1.0, 2.0) == 1
   @test quadrant(-13.0, -2) == 3
   ```

```
@test quadrant(4, -3) == 4
@test quadrant(-2, 6) == 2
```

3. There is a famous conjecture in mathematics (the Collatz conjecture) that states that any positive integer can be reduced to 1 by repeated application of these rules:

   1. If the number is even, divide it by two.
   2. If the number is odd, triple it and add one.

   Write a function that produces a sequence of numbers starting from a positive integer n and applying the rules above until it reaches 1.

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

Arrays

Arrays are a collection of elements, usually of the same type but not always. Arrays in Julia are **mutable**, which means that you can change the elements of an array after it has been created.

```
julia> v = [1, 2, 3]  # array of integers
3-element Vector{Int64}:
 1
 2
 3

julia> w = [1.0, 2.0, 3.0]  # array of floats
3-element Vector{Float64}:
 1.0
 2.0
 3.0

julia> a = ["hello", 2.0, 3, false, "□"]  # mixed array has type Any
3-element Vector{Any}:
    "hello"
    2.0
    3
false
    "□"
```

Notice that these arrays are of type `Vector{Int64}`, `Vector{Float64}`, and `Vector{Any}`. The type `Vector` is an alia for a one-dimensional `Array` in Julia.

```
julia> typeof(v)
Vector{Int64} (alias for Array{Int64, 1})

julia> typeof(w)
Vector{Float64} (alias for Array{Float64, 1})
```

Range objects

Range objects can also be used to create arrays. The object `r = 1:10` is a `UnitRange` object that represents the numbers from 1 to 10 with step size 1. You can also create a range object with a step size. For example, `r = 1:2:10` represents the numbers from 1 to 10 with a step size of 2. This is a `StepRange` object.

We can use the `dump` function to see the structure of an object. Try using `dump(r)` on the range object `r = 1:2:10` and compare it to a `Vector` object created with `v = [1, 2, 3]`. We can see that a `Vector` object is a one-dimensional

array of elements, while a `UnitRange` object is a range of numbers with a start, stop, and step size. It's kind of a lazy object that doesn't store all the numbers in memory. This makes them very memory efficient. See this StackOverflow post for a detailed explanation. When you use `collect(r)`, it creates a new array with the elements of the range object.

```
julia> v = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> dump(v)
Array{Int64}((3,)) [1, 2, 3]

julia> r = 1:2:10
1:2:9

julia> dump(r)
StepRange{Int64, Int64}:
  start: 1
  stop: 10
  step: 2

julia> dump(1:3)  # UnitRange is even lazier
UnitRange{Int64}
  start: Int64 1
  stop: Int64 3
```

You can also create a range object with more specificity by using the `range` function.

```
# create a range with a start value, end value, and step size
range(1, stop=10, step=2)

# create a range with a start value, step size, and length
range(1, step=0.5, length=5)
```

Check out `?range` in the REPL to see all the options available for creating range objects.

Indexing and slicing

You can access elements of an array using *indexing*. Julia uses 1-based indexing, which means that the first element of an array is at index 1, not 0.

```
a = ["hello", 2.0, false, "□", 6]
a[1]  # returns "hello"
a[2]  # returns 2.0
a[end-1]  # returns "□"
```

Subarrays can be extracted by *slicing* with `i:j` notation.

```
a = ["hello", 2.0, false, "□", 6]
a[1:2]  # returns ["hello", 2.0]
a[2:end]  # returns [2.0, false, "□", 6]
```

We can also step through the array with a step size.

```
a = ["hello", 2.0, false, "□", 6]
a[1:2:end]  # returns ["hello", false, 6]
```

We can also step *backwards* through the array.

```
a = ["hello", 2.0, false, "□", 6]
a[end:-1:1]  # returns [6, "□", false, 2.0, "hello"]
```

Elements in a list can be reassigned using their index.

```
julia> v = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> v[1] = 10  # change the first element of the array
10

julia> v
3-element Vector{Int64}:
 10
  2
  3
```

Exercises

1. Consider an array created as follows:

   ```
   x = 3
   my_array = [1, 2, x]
   ```

   What happens to `my_array` if `x` is changed after the array is created?

2. In the array `v = [1, 2, 3]`, what happens if you try to change the second element to `1.0`? What about if you try to change it to a string?

3. What is the last element of the range object `1:2:10`?

4. A `BoundsError` occurs when you try to access an element of an array that doesn't exist. Write a code snippet that produces a `BoundsError`. Then fix it.

If you want to iterate over a collection and keep track of the index, you can use the `enumerate` function.

```
fruits = ["apple", "banana", "cherry"]
for (index, fruit) in enumerate(fruits)
    println("Index: $index, Fruit: $fruit")
end
```

If you *only* want the index, you can use the `eachindex` function.

```
fruits = ["apple", "banana", "cherry"]
for index in eachindex(fruits)
    println("Index: $index")
end
```

There are special functions that allow you to iterate over rows and columns of a matrix

```
A = [1 2 3; 4 5 6; 7 8 9]
for row in eachrow(A)
    println(row)
end
for col in eachcol(A)
```

```
    println(col)
end
```

Operations on arrays

Simple operations can extract information about an array or modify it. The length of a `Vector` can be obtained using the `length` function.

```
a = ["hello", 2.0, false, "□", 6]
julia> length(a)
5
```

We can also use the `push!` function to add an element to the end of an array.

```
julia> push!(a, "-2")  # add a new element to the end of the array
6-element Vector{Any}:
    "hello"
    2.0
 false
    "□"
    6
    "-2"
```

Matrices

Matrices are two-dimensional arrays in Julia. You can create a matrix using the `reshape` function or by using semicolons to separate rows.

```
julia> A = [1 2 3; 4 5 6; 7 8 9]  # matrix with semicolons
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> B = reshape(1:9, 3, 3)  # matrix with reshape
3×3 reshape(::UnitRange{Int64}, 3, 3) with eltype Int64:
 1  4  7
 2  5  8
 3  6  9

julia> A[1, 2]  # indexing a matrix
2

julia> A[2, 3]
6

julia> A[1, :]  # all elements in the first row
3-element Vector{Int64}:
 1
 2
 3

julia> A[:, 1]  # all elements in the first column
3-element Vector{Int64}:
 1
```

```
 4
 7
```

You can also use the `size` function to get the dimensions of a matrix.

```
julia> size(A)
(3, 3)
julia> size(B)
(3, 3)
```

Matrix and vector operations

You can perform various operations on matrices and vectors, such as addition, subtraction, multiplication, and division.

```
julia> A + B  # matrix addition
3×3 Matrix{Int64}:
  2   6  10
  6  10  14
 10  14  18

julia> A - B  # matrix subtraction
3×3 Matrix{Int64}:
 0  -2  -4
 2   0  -2
 4   2   0

julia> A * B  # matrix multiplication
3×3 Matrix{Int64}:
 14   32   50
 32   77  122
 50  122  194

julia> A / B  # matrix division
3×3 Matrix{Float64}:
 -0.333333   0.666667  -0.0
  2.26667   -7.53333    5.6
 -6.33333    6.66667   -0.0

julia> A'  # transpose of a matrix
3×3 adjoint(::Matrix{Int64}) with eltype Int64:
 1  4  7
 2  5  8
 3  6  9

julia> A * v  # matrix-vector multiplication. What do you think will happen?
3-element Vector{Int64}:
 14.0
 32.0
 50.0

julia> v * A  # what do you think will happen?
```

There are more linear algebra operations available if you use the `LinearAlgebra` package.

Broadcasting

Broadcasting allows you to apply a function to each element of an array or collection. You can use the dot `.` operator to apply a function element-wise. This is also known as vectorizing a function. For example, you can perform element-wise operations using the `.*`, `./`, and `.^` operators.

```
julia> v = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> v .+ 1  # add 1 to each element of v
3-element Vector{Int64}:
 2
 3
 4

julia> v .+ v  # add v to itself
3-element Vector{Int64}:
 2
 4
 6
```

You can multiply a scalar with a vector without broadcasting. The notation is perhaps intuitive if you are used to mathematical notation. You can write both `2 * v` and `2v` to multiply a vector by a scalar. Actually this works for most variables.

Broadcasting also works with matrices. The `Matrix` type is an alias for a two-dimensional `Array`, denoted as `Array{T, 2}`.

```
A = [1 2 3; 4 5 6; 7 8 9]
B = A
julia> A .* B  # element-wise multiplication
3×3 Matrix{Int64}:
  1   4   9
 16  25  36
 49  64  81
```

Note the difference between `A * B` and `A .* B`. The first one is *matrix* multiplication, while the second one is *element-wise* multiplication.

Broadcasting also applies to functions. Let's start with a familiar function that adds 1 to a number.

```
julia> f(x) = 1 + x
f (generic function with 1 method)

julia> f(1)
2
```

What happens if we try to apply this function to a vector?

```
julia> v = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> f(v)
```

```
ERROR: MethodError: no method matching +(::Int64, ::Vector{Float64})
For element-wise addition, use broadcasting with dot syntax: scalar .+ array
The function `+` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...)
   @ Base operators.jl:596
  +(::Real, ::Complex{Bool})
   @ Base complex.jl:322
  +(::Array, ::Array...)
   @ Base arraymath.jl:12
  ...

Stacktrace:
 [1] f(x::Vector{Float64})
   @ Main ./REPL[95]:1
 [2] top-level scope
   @ REPL[99]:1
```

The error message is telling us that the function + does not know how to add a scalar to a vector. Broadcasting *vectorizes* the function. This is also known as "element-wise" operations, "vectorized" operations, or "broadcasting".

```
julia> f.(v)  # apply f to each element of v
3-element Vector{Int64}:
 2
 3
 4
```

You can also use broadcasting with other functions.

```
julia> sin.(v)  # apply sin to each element of v
3-element Vector{Float64}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
```

It is also possible to define a function that takes a vector as input and returns a vector as output.

```
v = [1, 2, 3]
f(x) = x .+ 1  # broadcasting with dot operator
f(v)  # returns [2, 3, 4]
```

or using the `@.` macro

```
julia> @. f(x) = x + 1  # broadcasting with @. macro
f (generic function with 1 method)
julia> f(v)
3-element Vector{Int64}:
 2
 3
 4
```

Exercises

1. Write a function that takes a vector of numbers `v` and a positive integer `n`. The function should return a new vector that rotates the elements of `v` by `n` positions. An element should go back to the beginning of the vector if it goes past the end.

```julia
function rotate(v, n)
    # add code here
end

using Test
v = [1, 2, 3]
@test rotate(v, 1) == [2, 3, 1]
```

Array comprehensions

There are many ways to programmatically create arrays in Julia. One way is to apply a function to each element of an array. For example, if we want to apply a function `square` to each element of a vector containing integers from 0 to 5, we can use the `map` function.

```julia
square(x) = x^2
map(square, 0:5)  # returns [0, 1, 4, 9, 16, 25]
```

Another method is to filter the elements of an array. For example, if we want to filter the even numbers from a vector containing integers from 0 to 5, we can use the `filter` function and the function `iseven` that returns true if a number is even.

```julia
filter(iseven, 0:5)  # returns [0, 2, 4]
```

Sometimes we don't want a lazy array. If we want each element in an array to be stored in memory, we can use the `collect` function.

```julia
collect(0:5)  # returns [0, 1, 2, 3, 4, 5]
```

Finally, array comprehensions combine mapping and filtering in a single expression. In the following example, we create a new array with the squares of the even numbers from 0 to 5.

```julia
[x^2 for x in 0:5 if iseven(x)]  # returns [0, 4, 16]
```

Finally, comprehensions can be used to create multi-dimensional arrays.

```julia
julia> [i + j for i in 1:3, j in 1:3]  # returns a 3x3 matrix
3×3 Matrix{Int64}:
 2  3  4
 3  4  5
 4  5  6
```

Exercises

1. Use a comprehension to create an array of numbers 1 to 100 that are divisible by 7.
2. Use an array comprehension to find all of the vowels in a string.

Problems

1. Make a vector with the numbers 1 to 10 using array comprehension.
2. Use the vector that you made in the previous quesion. Write code that sets every even-indexed element of a vector `v` to 0. A list of `n` zeros can be created with `zeros(n)` or `fill(0, n)`. *Challenge: Do this in one line of code without defining a new function!*

Plotting

All of the functions we have used so far have been part of the Julia standard library. In this lesson we will use our first external package: a plotting library called Makie.jl.

Makie.jl is a powerful, modern plotting library that is easy to use and produces high-quality visualizations. It has a GPU-accelerated backend for fast rendering and interactive plots, a static vector graphics backend for high-quality publication-ready plots, and a web-based backend for interactive plots in the browser.

Check out the official Makie tutorials for examples and documentation.

Before we start

Makie's Getting Started tutorial is very well written and, honestly, I can't do much better for an introduction. Before this lesson, please go through it on your own and ask questions if you run into trouble.

As a reminder, here is how to install Makie (or any package) on your system.

Install Makie

There are three backends for Makie: - **GLMakie:** a GPU-accelerated backend for fast rendering and interactive plots. - **CairoMakie:** a static vector graphics backend for high-quality publication-ready plots. - **WGLMakie:** a web-based backend for interactive plots in the browser.

In this tutorial we will use GLMakie in VS Code, but you can also use CairoMakie. Let's first install it.

1. Open the Julia REPL in VS Code via the Command Palette (`Cmd+Shift+P` on macOS) (`Ctrl+Shift+P` on Windows) and type `Julia: Start REPL`.
2. In the REPL, type `]` to enter the package manager. (Hit backspace to exit the package manager and return to the Julia REPL.)
3. Check that the current environment is set to the current directory. If not, type `activate .` to create a new environment in the current folder.
4. You can see what packages are installed by typing `status` in the package manager (`st` is a shortcut).
5. Type `add GLMakie` (or `CairoMakie`) to install.
6. Make a new file called `plotting.jl` in your tutorial folder.
7. Type `using GLMakie` inside the file at the top
8. Evaluate the line (`shift+enter`) to load the package.

Make a basic plot

Let's first define a function: a damped sine wave as a function of time.

```
function damped_sine(t, A, f, τ)
    return A * exp(-t / τ) * sin(2π * f * t)
end
```

Then we will generate the data. Start with the following values

```
A = 3.0  # amplitude
f = 1  # frequency in Hz
τ = 1  # decay time constant in seconds
```

Create the `seconds` values any way you like. Then generate the intensity values and store them in a variable called `intensity`.

```
seconds = 1:0.1:5
intensity = damped_sine.(t, A, f, τ)
```

Let's add a bit of noise to the data to make it look more realistic.

```
# Add some noise to the data
noise = 0.1 * randn(length(seconds))
intensity_noisy = intensity .+ noise
```

In Makie, the `Figure` is the top-level container object. An `Axis` is a container object for plots. We can place one or more `Axis` objects in a `Figure` to create a layout. Let's try this step by step in VS Code.

```
using GLMakie

f = Figure()
f
```

Hit `alt+enter` to run all of the code in this file. You should see a blank figure window pop up. The code might run slow at first. This is because Julia is a just-in-time (JIT) compiled language, meaning that it compiles code as you need it. The first time you run the code, it will take a bit longer to run. But subsequent runs will be much faster.

Think of a `Figure` as a blank canvas on which to place axes, plots, and labels. Now let's add an `Axis` to the figure.

```
f = Figure()
ax = Axis(f[1, 1])
f
```

The `Axis` function has many options to customize the plot, such as the title, labels, and limits. Let's make another `Axis` in the first row and second column of the figure.

```
f = Figure()
ax = Axis(f[1, 1])
ax2 = Axis(f[1, 2])
f
```

An `Axis` can span multiple rows and columns.

```
f = Figure()
ax = Axis(f[1, 1])
ax2 = Axis(f[1, 2])
ax3 = Axis(f[2, 1:2])
f
```

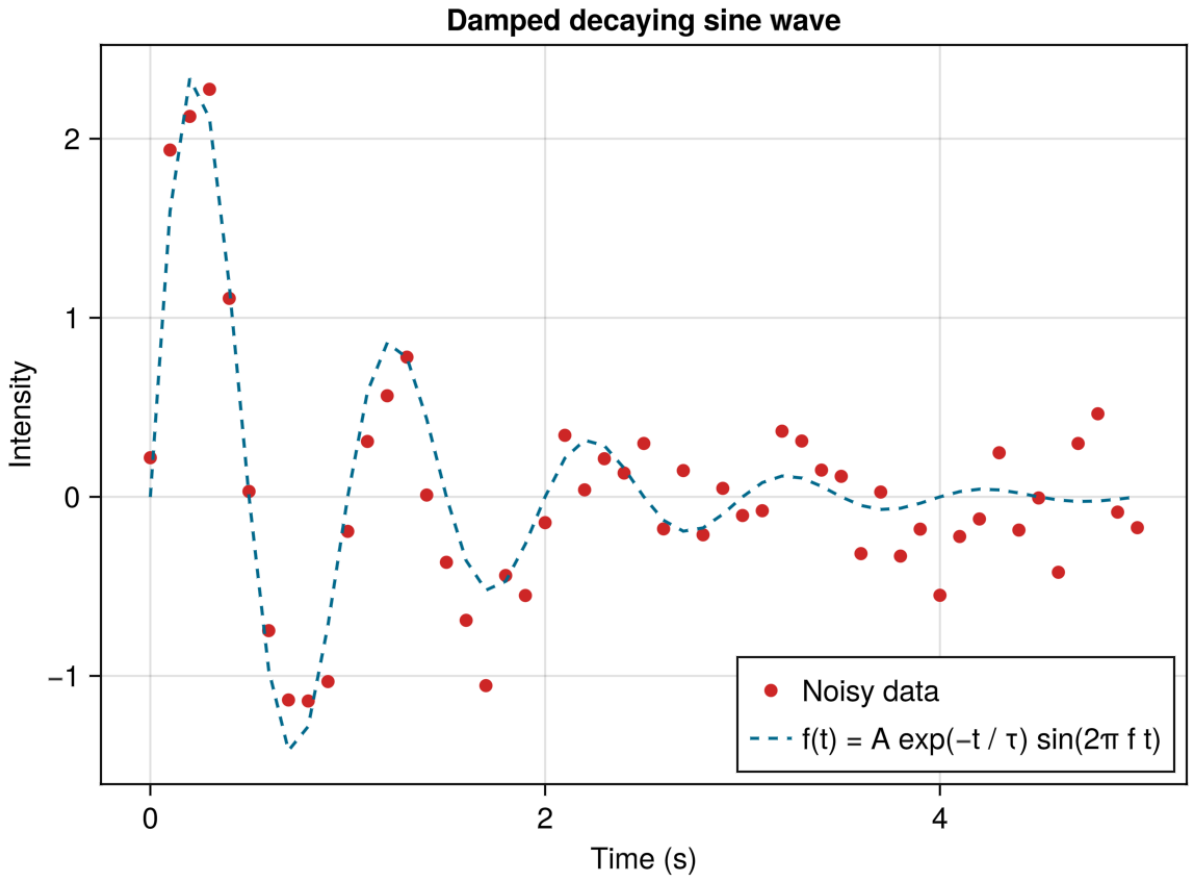
We can also change the size of the figure.

```
f = Figure(size = (800, 800))
```

These are just the basics of the powerful layout system in Makie. For now, let's just create a single `Axis` in the figure and plot our noisy data.

```
f = Figure()
ax = Axis(f[1, 1],
    title = "Damped sine wave",
    xlabel = "Time (s)",
    ylabel = "Intensity",
)
scatter!(
    seconds,
    intensity_noisy,
    color = :firebrick3
    label = "Data",
    )
lines!(
    seconds,
    intensity,
    color = :deepskyblue4,
    linestyle = :dash,
    label = "f(x) = A exp(-t / τ) sin(2π f t)",
    )
```

```
axislegend(position = :rb)
f
```



If the plot comes after the `Axis` definition, it will be drawn on top of the `Axis` and you don't need to input the `ax` variable. These are some colors that I like. There is a full color palette at Colors.jl.

Navigating a plot

You can zoom in and out of the plot by scrolling. Click while holding the `ctrl` key to reset the zoom. Holding `x` or `y` while scrolling will zoom in only on the x or y axis. Click and drag to zoom in a rectangle. This can also be done in x or y direction by holding `x` or `y` while dragging.

Commenting and working in sections in VS Code

By now you know that you can evaluate a single line of code by placing the cursor on the line and hitting `shift+enter`. You can also evaluate the entire file by pressing `alt+enter`.

It's common to also leave comments in the code to explain what the code does. You can add comments by starting a line with `#` or by adding `#` at the end of a line. A convenient way to comment out a line is to select the line and hit `cmd+/` (or `ctrl+/` on Windows). Hit `cmd+/` (`ctrl+/`) again to uncomment the line.

Try this by changing the `markersize` to `10` toggling the comment keyboard shortcut.

```
scatter!(
    seconds,
    intensity_noisy,
```

```
    color = :firebrick3
    label = "Data",
    # markersize = 10,
)
```

Sometimes you want to evaluate only a section of the code. For example, you might want to evaluate the code that generates the data and the code that creates the plot separately. You can separate chunks of code by adding `##` at the beginning of line. Then hitting `alt+enter` will evaluate the code in that section (between two `##` lines, or from the beginning of the file to the first `##` or from the last `##` to the end of the file). Try this by separating the code that generates the data and the code that creates the plot into two sections.

Modify the parameters of the function until you obtain a plot that looks something like this:

Saving

You can save the plot to a file using the `save` function.

You can save the plot to a file using the `save` function. Save this plot to a file called `damped_sine_wave.png` in a new folder called `output` in your tutorial folder.

```
save("output/damped_sine_wave.png", f)
```

Notice that the plot no longer appears in the window. You can comment out the `save` line and evaluate again to see the plot.

Problems

1. Visualize the potential of two point charges with surface plot in 3D. You will need to look up the equation for the potential of a point charge in Cartesian coordinates and how to create a surface plot in Makie.

   The potential of a point charge in Cartesian coordinates is given by the equation:

   $$\phi(x, y) = \frac{q}{4\pi\epsilon_0}\left(\frac{1}{\sqrt{(x+a)^2+y^2}} - \frac{1}{\sqrt{(x-a)^2+y^2}}\right)$$

   where q is the charge, ϵ_0 is the permittivity of free space, and a is the distance between the two charges. Place two charges, $+q$ and $-q$ on your plot at $(-a, 0)$ and $(+a, 0)$.

Basic fitting

In this chapter we will discuss the basic principles of fitting a model to data using the least squares method. We will use the LsqFit.jl package to perform the fitting.

Least squares fitting

You have some experimental data composed of n pairs of data points (x_i, y_i) where $i = 1, \ldots, n$ and y_i is the value that you observe at x_i. You want to gain some physical insight into the data by seeing how well a model explains it and adjusting the model parameters until the model function "fits" the data as best it can. For example, if you measure the absorbance for several samples of a liquid at various concentrations you expect there to be a linear relationship between the absorbance and the concentration (the famous Beer-Lambert Law). The difference between the measured absorbance and the linear model is the error. The slope and intercept of the line are the parameters to vary to find a line that best matches the data.

For whatever system you are studying, the parameters that produce a model that best fits the data hopefully say something useful about the physical system that you have measured. Aside from choosing a physically suitable mode, we have to somehow quantify how well the model "fits" the data. The model function takes the form $f(x, p)$, where p is a vector of parameters that you want to uncover via the fitting process. This vector of "best fit" parameters is what

we are trying to find. How well the model fits the data is measured by the difference between the observed values y_i and the model values $f(x_i, p)$ for a given p. The set of differences is called the residuals, and are defined by

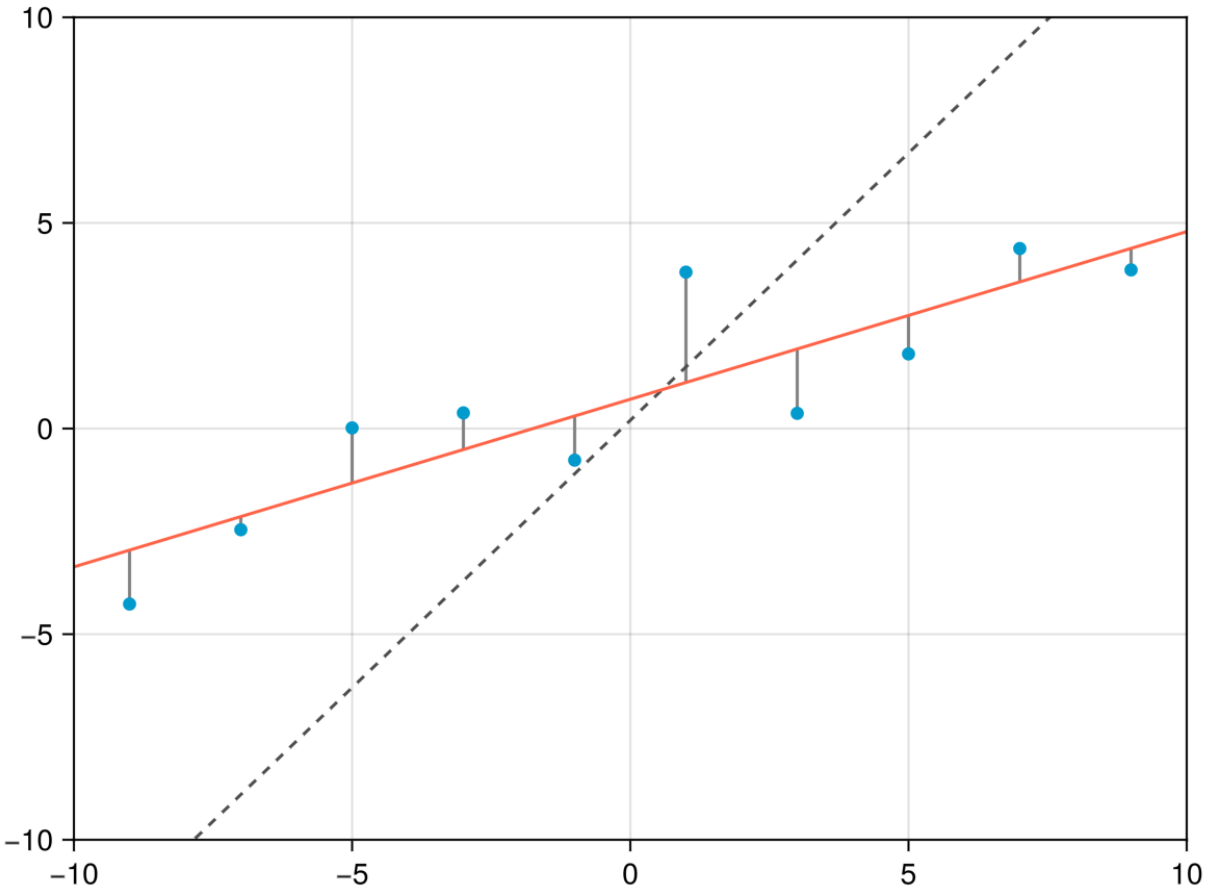
$$r_i = y_i - f(x_i, p)$$

The least squares method then squares the residuals and sums them up. Minimizing this sum of the squared residuals will return the optimal parameters values p. The sum of the squared residuals is given by

$$S = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - f(x_i, p))^2$$

This function is also known as the cost or loss function. Sometimes it is also referred to as the error function. There are many ways to minimize the cost function, but that is beyond the scope of this chapter. (Plotting the loss function is a useful way to visualize how sensitive the model is to different parameters, but this is a topic for the next chapter on optimization.)

The example below shows a linear fit to randomly generated data, together with the residuals shown as lines between the data points and the fitted line. The line representing the initial guess parameters is shown as a dashed line.



Problem

Let's say you take a noisy spectrum of a sample and observe single peak around 500 nm. You have good reason to believe that the peak is Lorentzian in shape, a common line shape in spectroscopy. A Lorentzian line shape is given by the equation

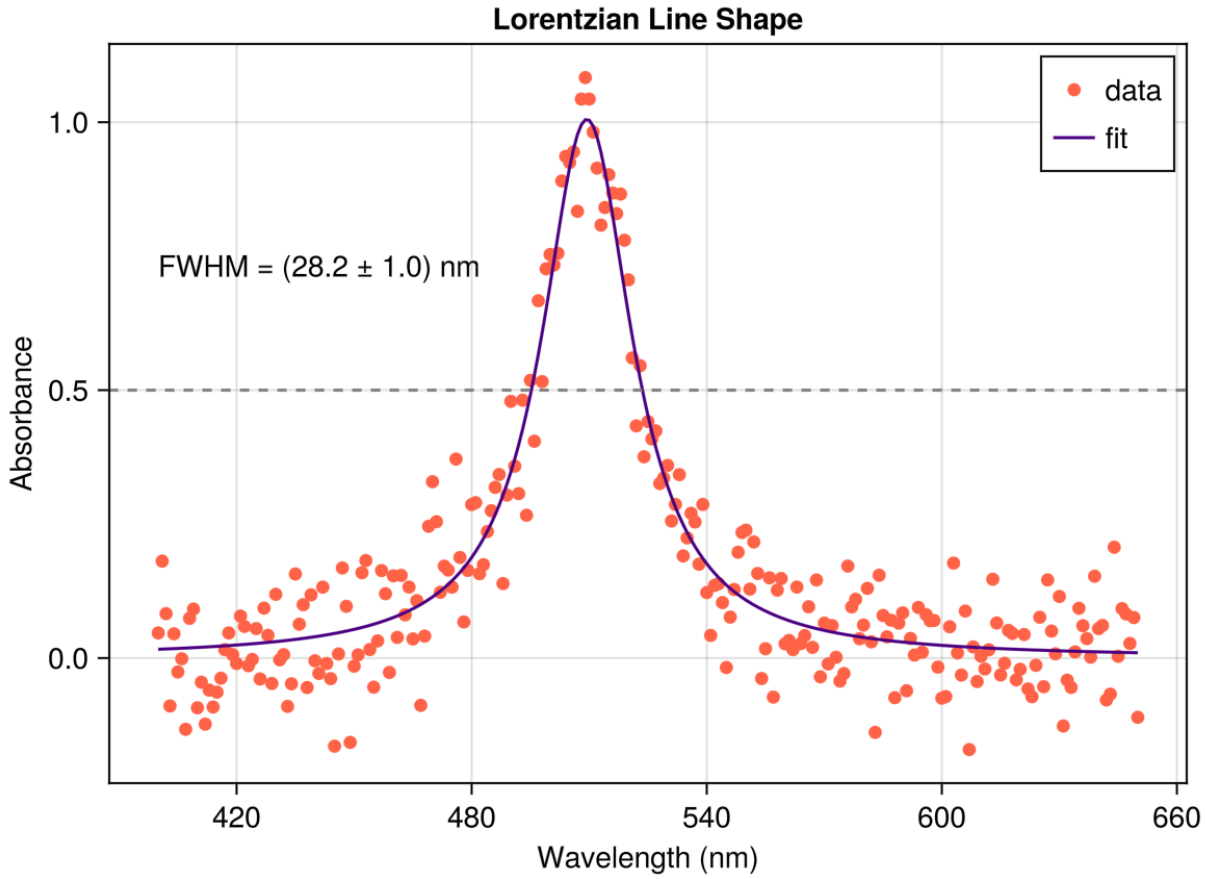
$$L(x) = \frac{I_0}{1 + \left(\frac{x - x_0}{\Gamma/2}\right)^2}$$

where I_0 is the amplitude, x_0 is the center frequency (often expressed in wavenumbers) of the peak, and Γ is the full width at half maximum (FWHM) occuring at points $x = x_0 \pm \frac{\Gamma}{2}$.

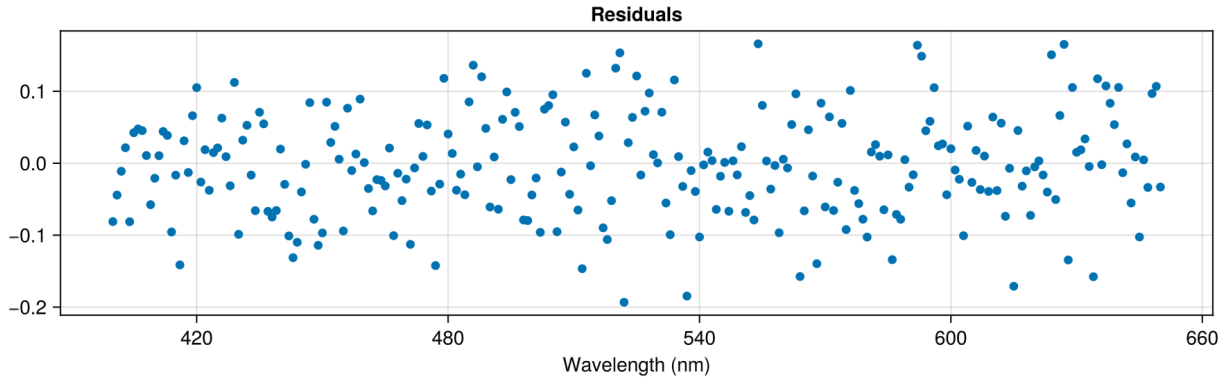
Go to the GitHub page for LsqFit.jl and follow the examples to use `curve_fit()` to fit a Lorentzian function with the following parameters:

```
A = 1.0
Γ = 28
x0 = 510
```

You should end up with something like below, depending on how many data points you create and how much noise you add to the data. Remember to include an error estimate for each parameter.



It is useful to plot the residuals of the fit to see how well the results fit the data. There should not be any systematic structure in the residuals. If there is, then the model is not a good fit to the data.



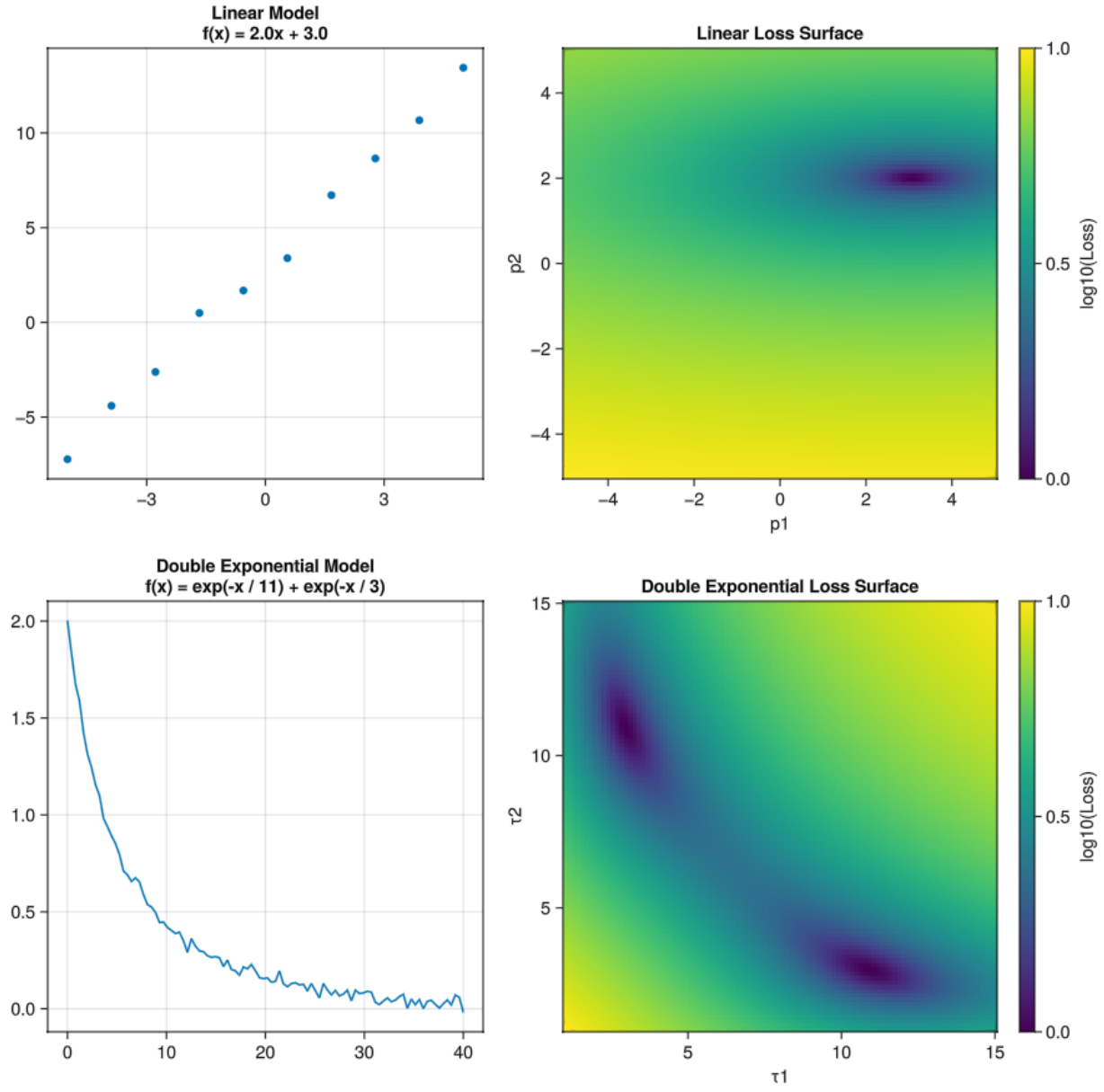
Resources

- Least Squares Fitting on Wikipedia
- Khan Academy on residuals and least squares regression
- Ledvij, M. "Curve Fitting Made Easy." Industrial Physicist 9, 24-27, Apr./May 2003.

Optimization

The fitting we did in the previous chapter is a simple example of optimization. An optimization problem maximizes or minimizes an objective function by varying the input parameters. With least squares fitting the objective function is the sum of the squared residuals, and this is then minimized.

It is often useful to visualize the loss function, which is the objective function plotted as a function of the parameters. Below are two examples of the loss functions plotted as functions of parameters for a linear model $f(x) = p_1 x + p_2$ and a double exponential model $f(x) = e^{-x/\tau_1} + e^{-x/\tau_2}$. Here, p_1 and p_2 are 2 and 3, respectively, and τ_1 and τ_2 are 11 and 3, respectively. You can see that the loss function is a simple parabola for the linear model the minimum of the parabola occurs at the point (2, 3). The double exponential loss surface has two wells, one at (11, 3) and the other at (3, 11) because it does not matter which of the two time constants is assigned to which parameter. More complicated loss functions can have multiple wells, and the optimization algorithm may get stuck in a local minimum. Thus the choice of optimization algorithm becomes important.



Later you will measure the Rabi splitting of a polariton system as a function of beam incidence angle. There will be two peaks in the transmission spectrum, one corresponding to the upper polariton and the other to the lower polariton, that will change in frequency with angle. If the two polariton peaks have the sample transmission amplitude, then the Rabi splitting is the frequency difference between the two peaks. Another way to extract the Rabi splitting is to measure the angle-resolved spectrum, plot the frequency versus incidence angle, and fit the data to a simple coupled harmonic oscillator model, given by the Hamiltonian:

$$H_{\text{total}} = \begin{pmatrix} \nu_c(\theta) & 0 \\ 0 & \nu_v \end{pmatrix} + \begin{pmatrix} 0 & \Omega_R/2 \\ \Omega_R/2 & 0 \end{pmatrix} = \begin{pmatrix} \nu_c(\theta) & \Omega_R/2 \\ \Omega_R/2 & \nu_v \end{pmatrix}$$

Diagonalizing the Hamiltonian gives the upper and lower polariton energies:

$$\nu_{\pm}(\theta) = \frac{1}{2}\left[\nu_c(\theta) + \nu_v \pm \sqrt{(\nu_c(\theta) - \nu_v)^2 + \Omega_R^2}\right]$$

The molecular vibrational transition is ν_v, and the Rabi splitting magnitude is Ω_R. The cavity mode frequency ν_c is given by the equation

$$\nu_c(\theta) = \nu_c(0)\left(1 - \frac{\sin^2\theta}{n^2}\right)^{-1/2}$$

where θ is the beam angle with respect to the normal of the cavity surface and n is the refractive index of the intracavity medium.

Let's first do this using the `LsqFit.jl` package and modify the model slightly to include both upper polariton and lower polariton curves. First, we need to write a function for the upper and lower polariton energies. This is an exercise for you to do. The function should take the parameters $\nu_c(0)$, ν_v, Ω_R, and n as input, and return the upper and lower polariton frequencies as output. Then we use the polariton function to create a model function for fitting.

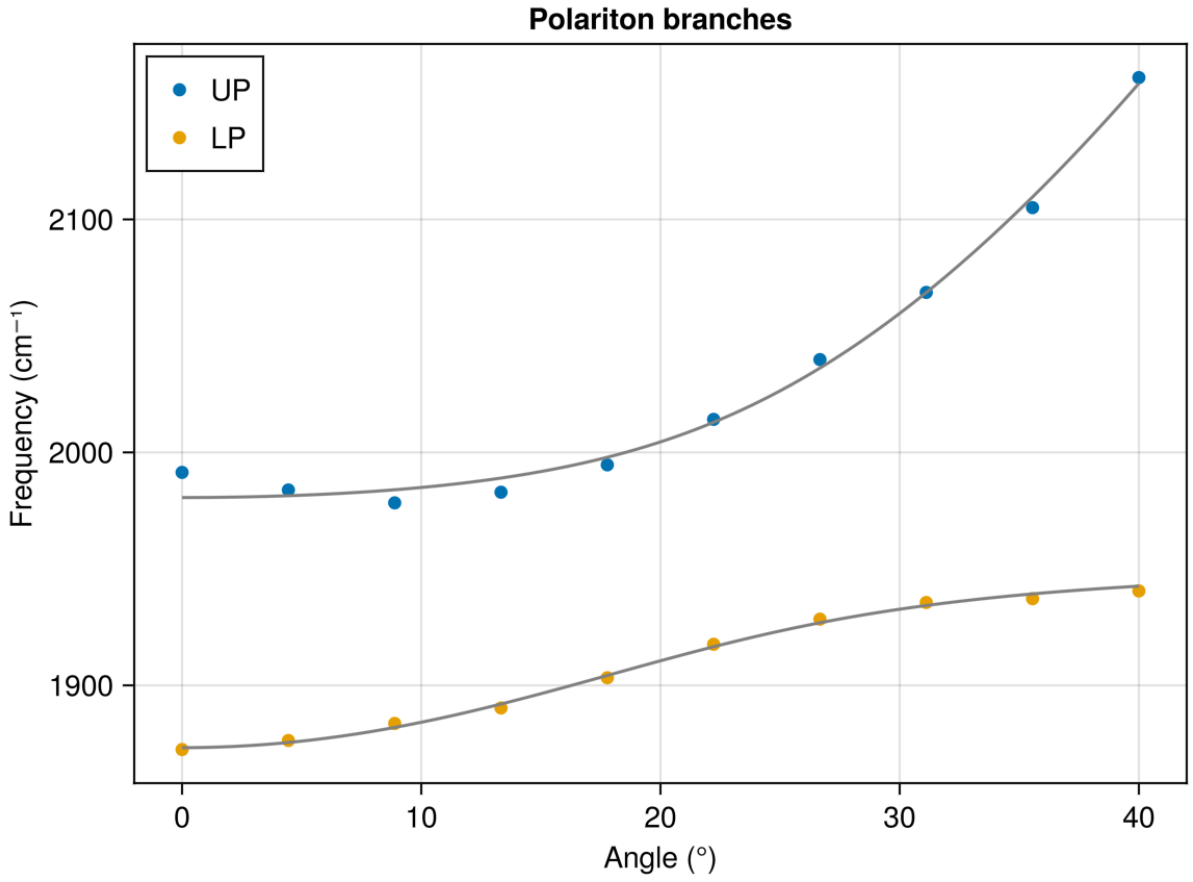
```
using LsqFit

function polariton_freqs(θs, ...)
    # Your code here
end

function model(θs, p)
    E_v, E_0, n, Ω = p
    LP = polariton_freqs(θs, ...)
    UP = polariton_freqs(θs, ...)
    return vcat(LP, UP)
end
```

Here we have used a new function called `vcat()` to concatenate the two vectors of frequencies into a single vector. This way we can use the same fitting function for both the upper and lower polariton frequencies simultaneously.

Then it's just a matter of organizing our data appropriately and calling the `curve_fit()` function.

```
fit = curve_fit(model, θs, vcat(LP, UP), [E_v, E_0, n, Ω])
```

Just as before, the parameters are passed as a vector in `fit.param`. If you plot you data and fitted curves, you should obtain something like the following.



Problems

1. Write a function to calculate the upper and lower polariton frequencies.

```julia
using LsqFit

function polariton_freqs(θs, ...)
    # Your code here
end

function model(θs, p)
    E_v, E_0, n, Ω = p
    LP = polariton_freqs(θs, ...)
    UP = polariton_freqs(θs, ...)
    return vcat(LP, UP)
end
```

2. Use the function to create data for the two polariton branches with gaussian noise and parameters $\nu_c(0)$ = 1900 cm-1, ν_v = 1950 cm-1, Ω_R = 60 cm-1, and n = 1.4.

3. Use the `curve_fit()` function to fit the model to the data. Try different initial guesses for the parameters and see how they affect the fit.

4. Try changing the number of fitting parameters. In the example above, we allowed four parameters to vary, but you can also fix some of them to known values. Often the molecular vibrational mode is known, for

example. Remember to report the standard error on each parameter and the confidence intervals. What do these quantities mean?

More advanced optimization

The `LsqFit.jl` package only uses the Levenberg-Marquardt algorithm for optimization. The Levenberg-Marquardt algorithm can be slow and may not converge to the global minimum, particularly in cases where the loss function has multiple wells. Sometimes we need a different algorithm or more powerful optimization methods. In these cases, it's usually necessary to define the loss function explicitly and use a more general optimization package. There are several alternatives and the Julia ecosystem has become a rich source of modeling, equation solving, and optimization tools. This page on fitting a simulation to a dataset provides a good example suitable for our lab.

Fourier transform and spectroscopy

Every signal has a spectrum that can be analyzed in the time domain or the frequency domain. Alternatively, you can measure in the spatial domain or spatial frequency domain. The Fourier transform is a mathematical operation that converts a signal from the time domain to the frequency domain.

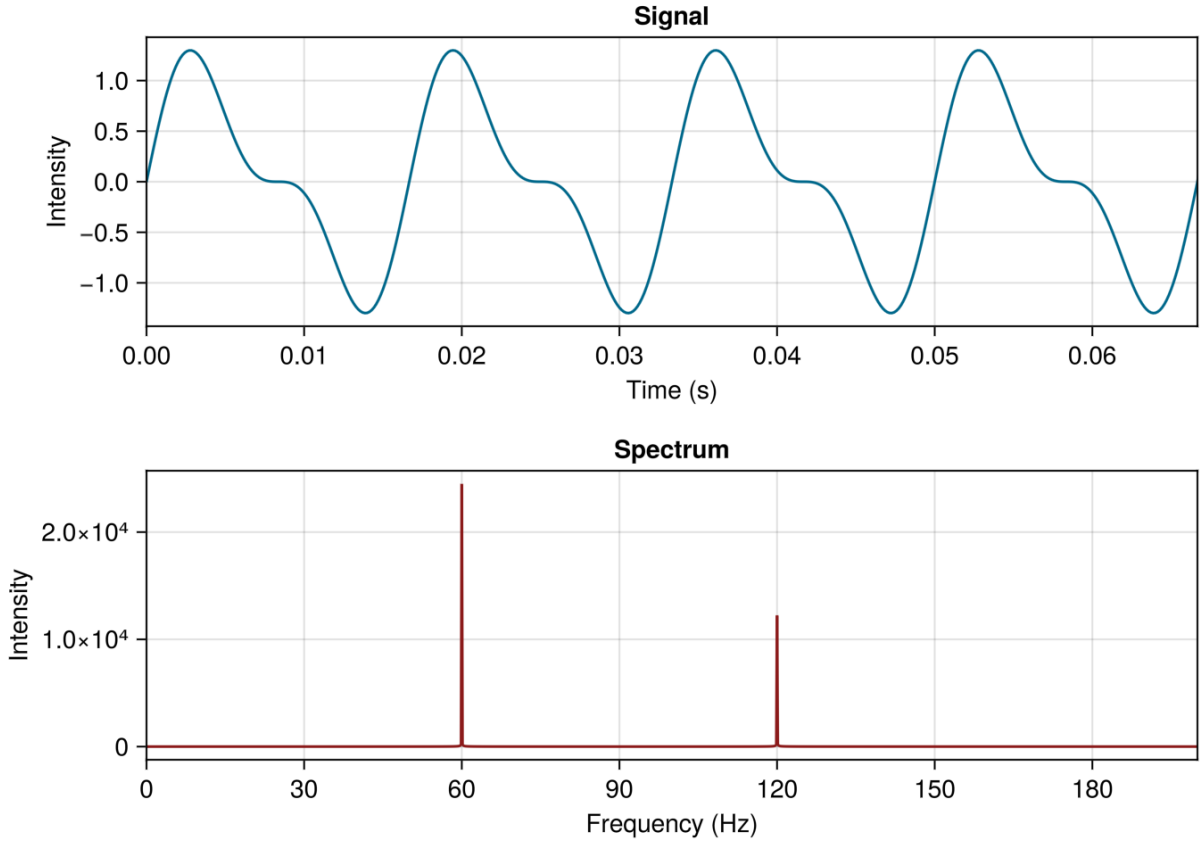
We use the Fourier transform to analyze the optical response of materials. Sometime when measuring the time-dependent nonlinear response of a material oscillatory signatures appear. The frequency of these oscillations corresponds to coherences and couplings within the system.

We also often use a spectrometer called a Fourier transform infrared (FTIR) spectrometer. This instrument measures the interference pattern of light from a sample (called the interferogram) and uses the Fourier transform to convert the interference pattern into a spectrum.

A proper Fourier transform course would take many weeks to cover. Here we will only cover the basics to understand how to compute the Fourier transform in Julia and interpret the results.

Two frequencies

Consider a sine wave at 60 Hz and its first harmonic at 120 Hz at half the amplitude of the fundamental frequency. The time-domain signal may look complicated, as shown in the figure below, the Fourier transform of the signal will show two distinct peaks at 60 Hz and 120 Hz. This simple example demonstrates the power of the Fourier transform to decompose a complex signal into its constituent frequencies, and we will use it to analyze much more complex signals arising from interactions between light and matter.



Let's implement this example in code. First we import the required packages, GLMakie for visualization and FFTW for the Fourier transform (FFTW.jl is just a wrapper for the FFTW library, the Fastest Fourier Transform in the West, written in C). Then we define the signal function.

```
using FFTW
using GLMakie

signal(t, f1, f2) = sin(2π * f1 * t) + 0.5 * sin(2π * f2 * t)
```

Next we define the sample rate and generate the time vector. The sample rate is the number of times per second we sample the signal. Here we will define the inverse of the sample rate to be the time spacing between samples. Half of the sample rate is called the Nyquist frequency, which is the highest frequency that can be accurately represented. If you sample a signal at a rate less than twice the highest frequency, you will get aliasing, which is when the signal appears to be at a lower frequency than it actually is. I encourage you to try sampling the signal at a lower rate and see what happens.

```
fs = 1000 # sample rate in Hz
t_max = 1 # max time in seconds
t = 0:1/fs:t_max

f1 = 60 # frequency in Hz
f2 = 120 # frequency in Hz
y = signal.(t, f1, f2)
```

Now we can compute the Fourier transform of the signal with the following code.

```
Y = abs.(fftshift(fft(y)))
X = fftshift(fftfreq(length(t), fs))
```

Once you successfully plot the signal and its Fourier transform let's consider what the three functions — `fft`, `fftshift`, and `fftfreq` — in the code do.

The function `fft` computes the discrete Fourier transform (DFT) of the signal in a way that is computationally efficient. The result is a complex array, which is why we take the absolute value of the result to get the amplitude spectrum.

The function `fftfreq` computes the frequencies corresponding to the Fourier transform based on the window size `n = length(t)` (the length of the time vector) and the sample spacing `fs`. It returns the positive frequencies first, then the negative frequencies. The Fourier transform is symmetric about 0; it is defined on the interval (-∞, ∞). Therefore, the second half of the Fourier transform is just a mirror image of the first half. The function `fftshift` moves the zero frequency component to the center of the spectrum to make it easier to visualize. Examples for `fftfreq` can be found by typing `?fftfreq` in the Julia REPL or hovering over the function name in VS Code. Likewise, `fft` also requires shifting to the center of the spectrum.

We can see what `fftshift` does by acting it on the numbers one through ten.

```
julia> fftshift(1:10)
10-element Vector{Int64}:
  6
  7
  8
  9
 10
  1
  2
  3
  4
  5
```

You can see that it just swaps the first half of the array with the second half. Try plotting the Fourier transform of the signal with and without using `fftshift` and see how it looks.

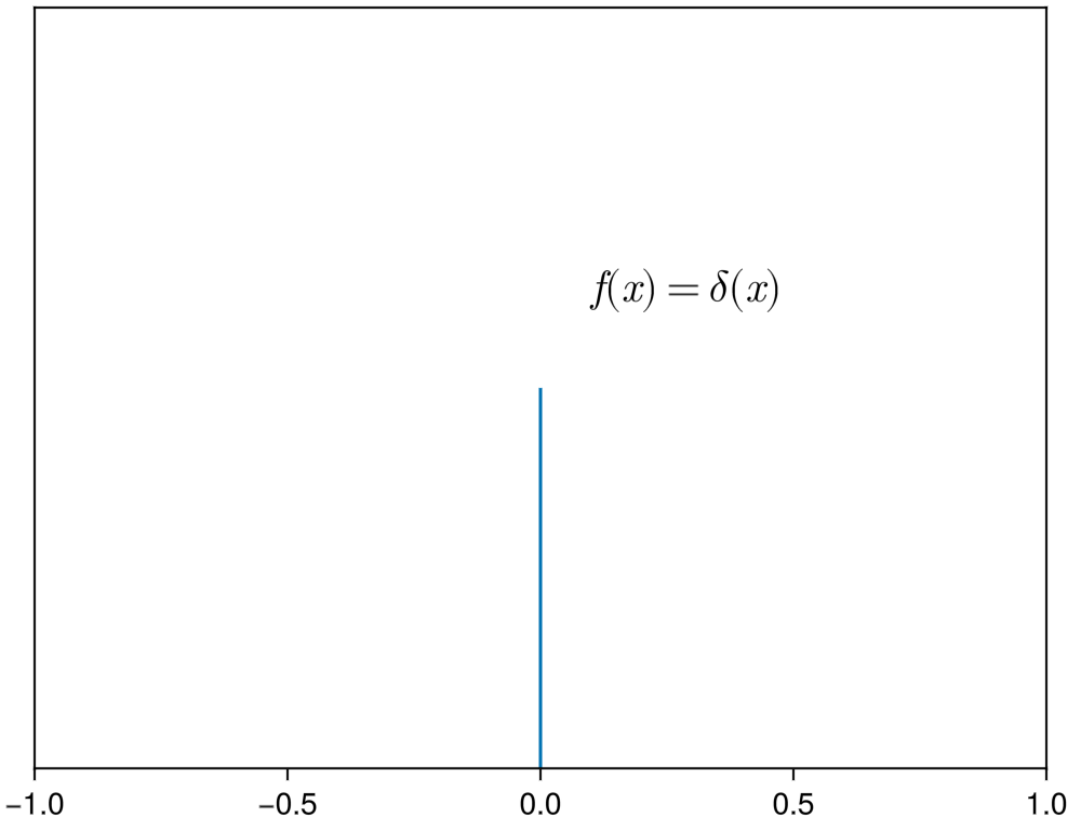
Problems

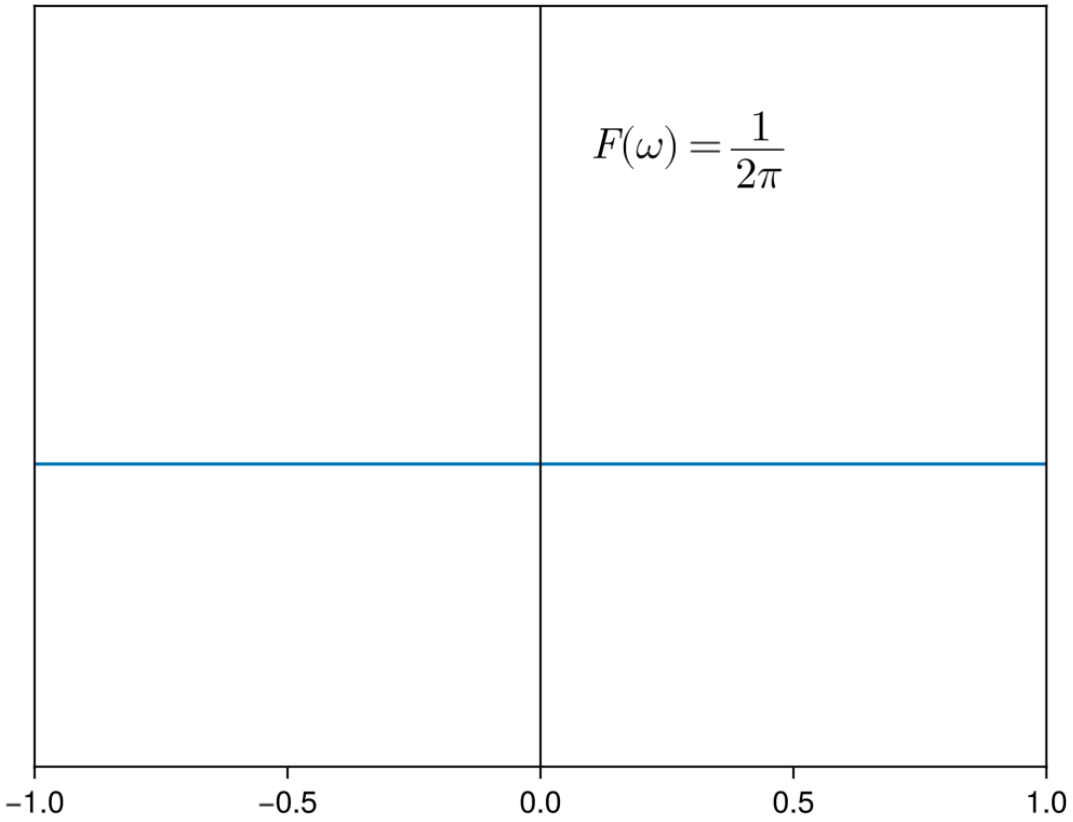
Recall that the delta function $\delta(x)$ is a curious function with the following properties

$$\delta(x) = \begin{cases} 0, & x \neq 0, \\ \infty, & x = 0, \end{cases}$$

and

$$\int_{-\infty}^{\infty} \delta(x) dx = 1.$$





The Dirac delta

function is actually a distrubution, not a function, and distributions are defined by their integration properties or actions on test functions. For a function $f(x)$ that is continuous at $x = 0$, the delta function has the property

$$\int_{-\infty}^{\infty} f(x)\delta(x)dx = f(0),$$

and if it is continuous at $x = a$, then

$$\int_{-\infty}^{\infty} \delta(x-a)f(x)dx = \int_{-\infty}^{\infty} \delta(a-x)f(x)dx = f(a).$$

Then, the Fourier transform of $\delta(x-a)$ is

$$\int_{-\infty}^{\infty} \delta(x-a)e^{-ix\xi}dx = e^{-ia\xi}.$$

1. Using the definition of the Dirac delta compute the following:

$$\frac{1}{2}\int_{-\infty}^{\infty} [\delta(x+a) + \delta(x-a)]e^{-ix\xi}dx$$

2. Write down the equation for a damped oscillator in terms of a temporal decay constant and an oscillation frequency and then implement it in code. Then compute its Fourier transform. Plot both curves. What happens to both the time-domain curve and its transform when you change the decay constant?

I have given you code the implements a square wave and it's Fourier transform. From the code, answer the following questions. **DO NOT RUN THE CODE YET!!** Just read it.

3. What is the frequency of the square wave?
4. How many harmonics are present in the Fourier transform and what are their frequencies?
5. What is the frequency resolution?
6. What is the Nyquist frequency of the given code? After which harmonic does it occur? In other words, what is the highest frequency that can be accurately represented?

Now **run the code** and answer the following questions.

7. Increase the number of harmonics and see what happens to the Fourier transform. What do you observe? Now decrease the number of points (thus decreasing the Nyquist frequency). What do you observe?

The photosynthetic light harvesting antennas in proteins are known to have electronic coherences, which are notoriously difficult to measure because of the noisiness of biological environments. The paper used in this problem measures long-lived coherences between excitons and molecular vibrations in coupled chromophores. The following problem uses raw data from Figure 2d in the paper. The raw data and basic code are provided. You will perform analysis to extract the lifetime and frequency of the oscillations.

Zhu, *et al.* Quantum Phase Synchronization via Exciton-Vibrational Energy Dissipation Sustains Long-Lived Coherence in Photosynthetic Antennas. Nat Commun 2024, 15 (1), 3171. https://doi.org/10.1038/s41467-024-47560-6.

8. Using a single exponential decay function, fit the data to extract the energy relaxation time. Plot the data and the fit.

9. Now subtract the fit from the data and compute the Fourier transform of the residuals. Plot the Fourier transform and identify the oscillation frequency. What is the physical meaning of this frequency?

Transfer Matrix Method

This chapter blends physics and programming. First we will cover strong coupling in microcavity structures and then we will use the transfer matrix method to simulate the propagation of electromagnetic waves through a series of layers.

The physics concepts will be based on the following text.

Skolnick, Fisher, and Whittaker. Strong coupling phenomena in quantum microcavity structures. *Semicond. Sci. and Technol.* **1998**, *13* (7), 645-669. https://doi.org/10.1088/0268-1242/13/7/003.

You may use outside literature to bolster your understanding. In particular, you may find the Wikipedia entry on the distributed Bragg reflector helpful.

Introduction

Read the introduction in Skolnick, *et al.* and answer the following conceptual questions.

1. What is an exciton? How does it form?
2. What is a quantum well?
3. How is a DBR mirror structured? How does this structure create a highly reflective surface?
4. How do the authors define a quantum microcavity (QMC)?
5. Why does a Fabry-Pérot cavity cause the photons to be quantized in the "growth" (vertical) direction and not in the in-plane direction?
6. What are the differences between "bulk" polaritons and "cavity" polaritons?
7. How does the dispersion of photons in cavity differ from photons in free space (the vacuum)?
8. How is the strong coupling limit defined? What phenomena occur in this limit? What about in the weak coupling limit?

Basic microcavity physics

At 630 nm wavelength, TiO2 has a refractive index of about 2.39 and SiO2 has a refractive index of about 1.46. Consider a distributed Bragg reflector (DBR) mirror made from alternating layers of TiO2 and SiO2.

1. What is the reflectivity of a DBR mirror for 2, 4, and 8 periods of TiO2/SiO2 layers?
2. What is the bandwidth of the photonic stop band of the mirror?

Now consider a microcavity with a cavity length of 1 μm and a DBR mirror on each side.

3. What is the effective cavity length if the distance between the two mirror surfaces is 1 μm? Why is the effective cavity length different than when measured as the distance between the two mirror surfaces?
4. What is the cavity length required to have a resonance at 630 nm?
5. What is the cavity mode width Δc for the cavity in the previous question? What lifetime does this correspond to? What is the Q-factor of the cavity?

Transfer matrix method

I find that one of the best ways to learn a physics concept is to implement it in code. When I was in graduate school, that is exactly what I did when I built a transfer matrix simulation first in Python and then in Julia (which is the version we will be using). Having you build an entire simulation from scratch is too much to ask, but at least we can use the transfer matrix method to simulate some simple structures with electromagnetic waves propagating through them.

The transfer matrix method simulates the propagation of electromagnetic waves through a series of layers. It can produce a wide range of static phenomena, such as the reflection, transmission, and absorption of light, Bloch surface waves, and the electric fields existing inside the layers. There are known limitations to the transfer matrix method, such as the fact that it cannot be used to simulate nonlinear effects or time-dependent phenomena, but it is a powerful

tool in our lab for designing dielectric mirrors, simulating transmission spectra of complex structures, and analyzing polariton transmission spectra. The transfer matrix implementation in TransferMatrix.jl is based on the work of Passler *et al.* (see references). It is a general 4 x 4 matrix formalism that tries to avoid some of the traditional pitfalls of the traditional Yeh formalism (notably the tendency for singularities to occur in certain cases). The code itself is modular and well-documented (I hope) so that it is easy to modify parts of the algorithm to suit your needs.

Getting started

I have written both a quick start and longer tutorial on the documentation website for TransferMatrix.jl. First go through those examples and make sure you understand what the code is doing and also how to interpret the generated spectra.

Once you have done that, you will build your own structure and simulate a transmission spectrum.

Problems

Simulate the reflection spectrum of the dielectric mirror structure from the previous problem set using the transfer matrix method and answer the following questions using the numerical approach and compare with your analytic answers from before.

1. What is the reflectivity of a DBR mirror for 2, 4, and 8 periods of TiO2/SiO2 layers?
2. What is the bandwidth of the photonic stop band of the mirror?

Now simulate a mirror with TiO2 and Ta2O5 layers for an incident wavelength of 630 nm.

3. How many layers do you need to have a higher reflectivity than 2 layers of TiO2/SiO2?
4. Which mirror has a larger stop band? Why?

Next simulate a gold-mirror cavity with a cavity length of $\lambda_0/2$ where λ_0 = 3. Let the intracavity medium be air (n = 1). (Make sure to plot the spectrum to long wavelengths.)

5. Plot the transmission spectrum in units of wavelength and frequency (cm-1). What do you notice about the mode spacing?
6. Adjust the gold thickness until you obtain a cavity mode FWHM of about 40 cm-1 for the mode near 3200-3300 cm-1. What is this thickness?
7. What photon lifetime does this FWHM correspond to? What is the Q-factor of the mode?
8. The free spectral range (FSR) is defined as the frequency spacing between adjacent cavity modes. Calculate the average FSR for this cavity with a gold thickness of 10 nm.
9. Choose any cavity mode and calculate the electric field at that wavelength. Can you find the first order mode? What is its electric field profile?
10. Change increase the cavity length to 3/2 λ_0.How does the electric field profile change? What is the new FSR?

References

- Transfer Matrix Method Wikipedia entry
- Yeh, P. *Optical Waves in Layered Media*; Wiley Series in Pure and Applied Optics; Wiley, 2005.
- Passler, N. C.; Paarmann, A. *Generalized 4 × 4 Matrix Formalism for Light Propagation in Anisotropic Stratified Media: Study of Surface Phonon Polaritons in Polar Dielectric Heterostructures.* J. Opt. Soc. Am. B 2017, 34 (10), 2128. https://doi.org/10.1364/JOSAB.34.002128.
- Passler, N. C.; Paarmann, A. *Generalized 4 × 4 Matrix Formalism for Light Propagation in Anisotropic Stratified Media: Study of Surface Phonon Polaritons in Polar Dielectric Heterostructures: Erratum.* J. Opt. Soc. Am. B 2019, 36 (11), 3246. https://doi.org/10.1364/JOSAB.36.003246.

- Garibello, B.; Avilán, N.; Galvis, J. A.; Herreño-Fierro, C. A. *On the Singularity of the Yeh 4 × 4 Transfer Matrix Formalism.* Journal of Modern Optics 2020, 67 (9), 832–836. https://doi.org/10.1080/09500340.2020.1775905.