Contents

Introduction to Julia for spectroscopy

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.

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.

Julia has a small community, but 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.
8. Programming basics in Julia

This lesson is a overview of basic programming concepts using the Julia language and the REPL. The intended audience is people who have never programmed before, or who have only done a little programming in another language.

Install Julia and Visual Studio Code

1. Download the latest version of Julia from the official website.
2. Follow the installation instructions for your operating system.
3. Download and install Visual Studio Code.
4. 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 and VS Code will be introduced later. There are many code editors available, but we will use VS Code because it is free and widely used for many programming languages.

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.

```
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?
```

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?

```
num_cats = 5
num_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
```

Going back to the division example, we can see that the type of the result depends on the operator. The `/` operator performs floating-point division, which means it returns a `Float64` type. If we want to perform integer division, we can use the `div` function or the `÷` operator.

```
julia> a = 2  # assign a
2

julia> a / 2  # floating-point division
1.0

julia> a ÷ 2  # integer division
1

julia> a // 4  # rational number
1//2
```

Exercises

1. Check the definitions of `div`, `÷`, and `//` by typing `?div`, `?÷`, and `?//` in the REPL.
2. Can you make a complex *integer*?

3. Try operations with different types. What happens?
4. What type is operators like * and +?
5. Evaluate the following expressions without executing the code:

```
1 + 5÷3 + 2^3
```

```
11/2-11÷2-3
```

6. Find the value of x at the end of this block of code:

```
x = 3^2
x = x + 1
x = x + 1
y = x÷2
x = y*y
z = 2*x
```

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"
```

Questions

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 funtion `length` do? Try it on a string and a number.
4. What does the function `string` do? Try it on a number, a string, and a variable.
5. Make the statement `"Hello world"[i:j] == "o wo"` return `true`.

6. What happens if `j` is replaced by `end` in the above expression?

Logical operators

Logical operators are used to perform logical operations on boolean values. Booleans are a type that can be either `true` or `false`.

```
julia> t = true
true

julia> f = false
false

julia> t && f  # logical AND
false

julia> t || f  # logical OR
true

julia> !t  # logical NOT
false

julia> t == f  # equality
false

julia> t != f  # inequality
true

julia> t === f  # strict equality
false

julia> t !== f  # strict inequality
true
```

Question

1. What is the type result returned by `1 == 2`?

Comparison operators

Comparison operators are used to compare values.

```
julia> a = 1
1

julia> b = 2
2

julia> a < b  # less than
true

julia> a <= b  # less than or equal to
true

julia> a > b  # greater than
false
```

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

julia> a == b  # equal to
false

julia> a != b  # not equal to
true
```

Questions

1. Check how to use the funciton `isa` by typing `?isa` in the REPL.
2. Use `isa` to check the type of `a`, and `b`. What does it return?
3. What type is `isa`? How can you check this?
4. What is the difference between == and ===?

Finding help

- Official Julia Documentation
- Julia tutorials list
- The Julia WikiBook
- The Julia Express

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.

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.
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 this lesson (on functions).

Please continue to use this tutorials folder for you 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.

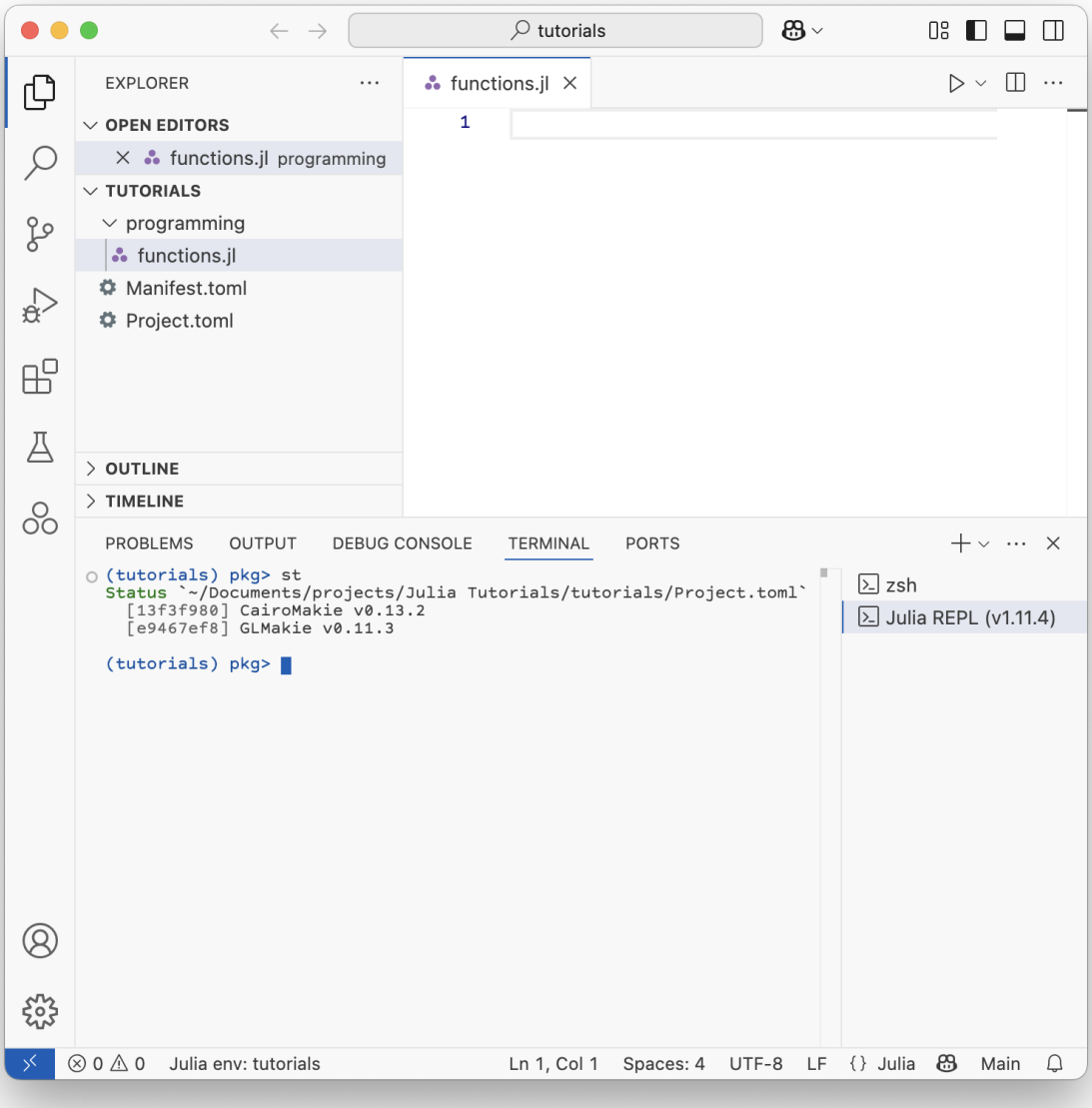
First look in the bottom left corner of the Visual Studio Code window. Notice that it say "Julia env: v1.11" or something similar. This means that you are using the default Julia environment, which is the global environment. In general, we don't want to use the global environment for our projects. Sometimes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



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 and useful 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.

Activity

Write a function that takes a positive integer as input and return the nth positive odd integer.

```
function nth_odd(n)
    # Your code here
end

using Test
@test nth_odd(3) == 5
@test nth_odd(5) == 9
@test nth_odd(19) == 37
```

Conditionals and iteration

Conditionals allow you to control the flow of execution in your code. This includes `if` statements, `for` loops, and `while` loops. It executes different blocks of code based on the value of a boolean expression.

If statements

If statements allow you to execute code conditionally based on a boolean expression.

Here's a simple example that returns the absolute value of the input argument.

```
function make_abs(x)
    if x < 0
        return -x
    else
        return x
    end
end

println(make_abs(-5))  # output: 5
```

Here's an example that includes an `elseif` statement to check whether a number is positive, negative or zero.

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

x = 5
check_number(x)  # output: x is positive
```

Now change the value of x to a negative number and run the code again.

Activity

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
```

For statements

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

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

To iterate over a collection, you can use the `for` loop as follows:

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

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
```

Exercise

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

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.

```
i = 1
while i <= 10
    println(i)
    i += 1
end
```

Activity

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.

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

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

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.

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

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

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`?

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.

   For example: `julia  v = [1, 2, 3]  rotate(v, 1) # returns [3, 1, 2]`

2. Write a line of 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)`.

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.

```
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.

```
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.

```
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.

```
[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> [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
```

Activity

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.

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.

```julia
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

```julia
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`.

```julia
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.

```julia
# 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.

```julia
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
```

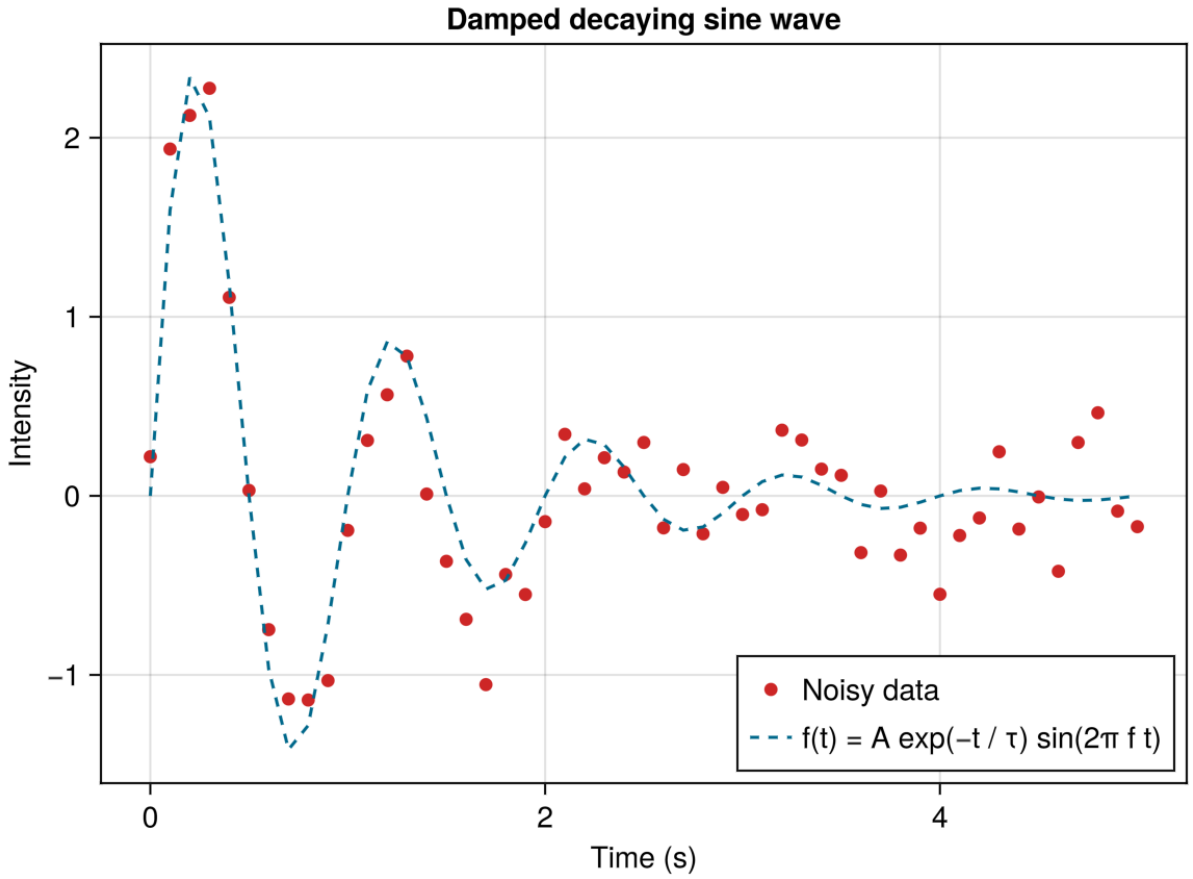
We can also change the size of the figure.

```
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.

Exercise

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 data to a model using the least squares method. We will use the LsqFit.jl package to perform the fitting, which uses the Levenberg-Marquardt algorithm.

Nonlinear least squares

You are a spectroscopist and you have taken a spectrum of a sample with n pairs of data points (x_i, y_i) where $i = 1, ..., n$ and y_i is the value that you observe at x_i. You want to gain some physical insight into the spectrum by overlaying a model on the data and adjusting the model parameters until the curve fits as best it can. The parameters of the model that best fit the data hopefully say something useful about the physical system that you have just measured. 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 $f(x_i, p)$. 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

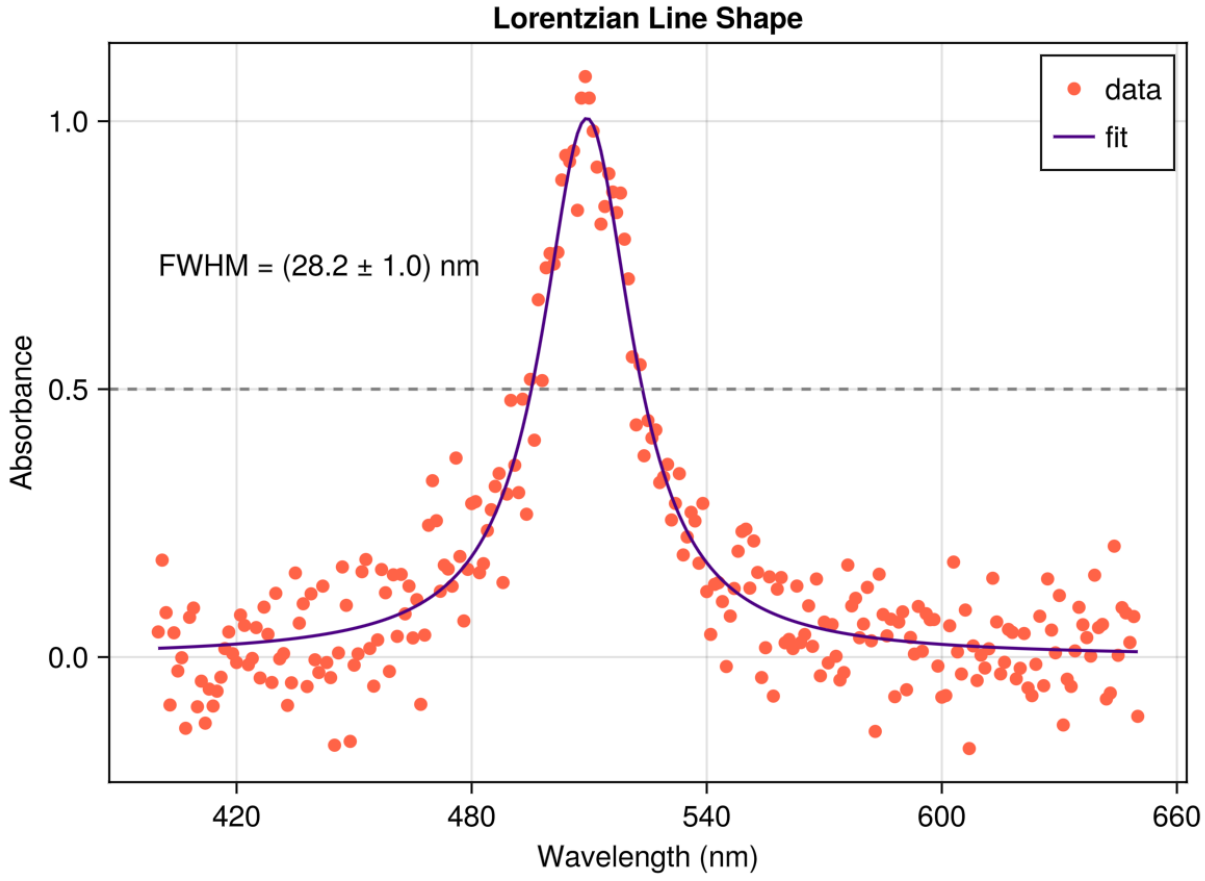
$$S = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - f(x_i, p))^2$$

This function is also known as the cost or loss function. Sometimes it is also referred to as the error function. 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.

Let's say you have good reason to believe that the peak is Lorentzian in shape. (You must have some physical justification for any model you choose.) A Lorentzian line shape is given by the equation

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

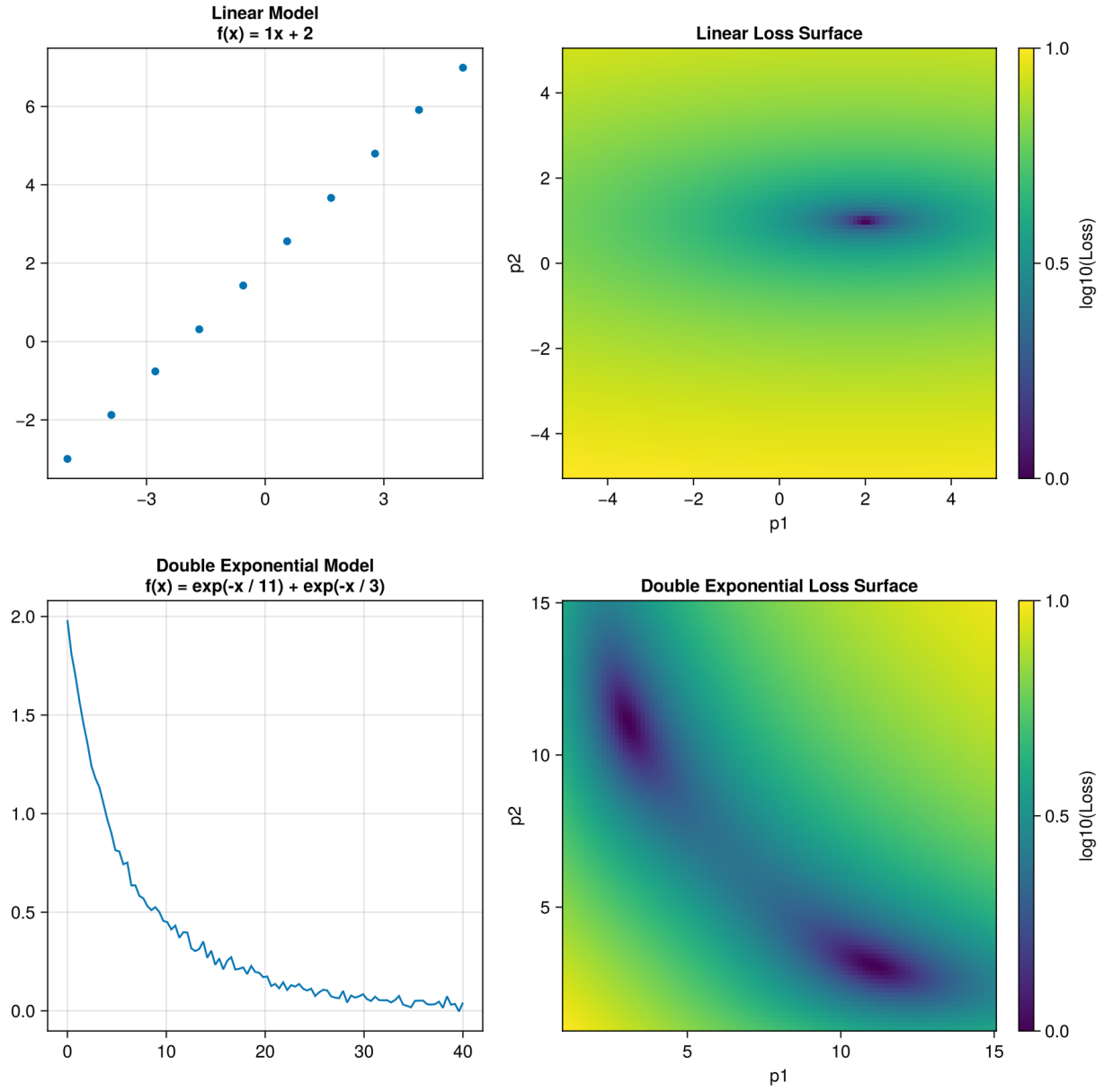
where I_0 is the amplitude, x_0 is the center frequency (often expressed in wavenumbers) of the peak, and Γ is the full width at half maximum (FWHM) occuring at points $x = x_0 \pm \frac{\Gamma}{2}$.



Resources

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

Optimizing a function



In the final experiment tutorial, you measured the spectrum for a polariton system. If the two polariton peaks have the sample transmission amplitude, then the Rabi splitting is the frequency (energy) difference between the two peaks. Another way to extract the Rabi splitting is to measure the angle-resolved spectrum, plot the energy versus incidence angle, and fit the data to a simple coupled harmonic oscillator model. We did not measure the angle dependence, so this lesson provides some fake data to work with.

Since the data is more complicated than a simple curve, we will write our own loss function and minimize it using the `Optim.jl` package. This is often required if the data do not fit to a simple model, or you wish to use a specific algorithm. (Julia has a rich ecosystem of machine learning and optimization packages that you may wish to take advantage of during your studies).

As a reminder, we want to minimize the square of the errors between the model and the raw data:

$$\min \sum_i \left(y_{\text{model}}(x_i) - y_{\text{i, data}}\right)^2.$$

In this case, we will use generated data representing the polariton dispersion curve from an angle-resolved measurement.

To fit this data, we use a simple coupled-oscillator Hamiltonian:

$$H_{\text{total}} = H_0 + H'$$

$$H_{\text{total}} = \begin{pmatrix} E_{\text{cavity}} & 0 \\ 0 & E_{\text{vibration}} \end{pmatrix} + \begin{pmatrix} 0 & \Omega_R/2 \\ \Omega_R/2 & 0 \end{pmatrix} = \begin{pmatrix} E_{\text{cavity}} & \Omega_R/2 \\ \Omega_R/2 & E_{\text{vibration}} \end{pmatrix}$$

The cavity mode energy is E_{cavity}, which depends on the angle θ of the beam with respect to the normal of the cavity surface. The molecular vibrational transition is $E_{\text{vibration}}$, and the Rabi splitting is Ω_R. Diagonalize the Hamiltonian to find the upper and lower polariton energies.

Finally, write an error function to minimize the lower and upper polariton energy equations.

Fourier transform