Introduction to data analysis with Julia

Chris Waudby
c.waudby@ucl.ac.uk

Introduction to data analysis with Julia

- Basic introduction to programming and scripting
- Publication-quality scientific plotting
- Automatic uncertainty propagation in calculations
- Nonlinear curve fitting, e.g. for binding studies
- Tips for handling datasets

Introduction to data analysis with Julia

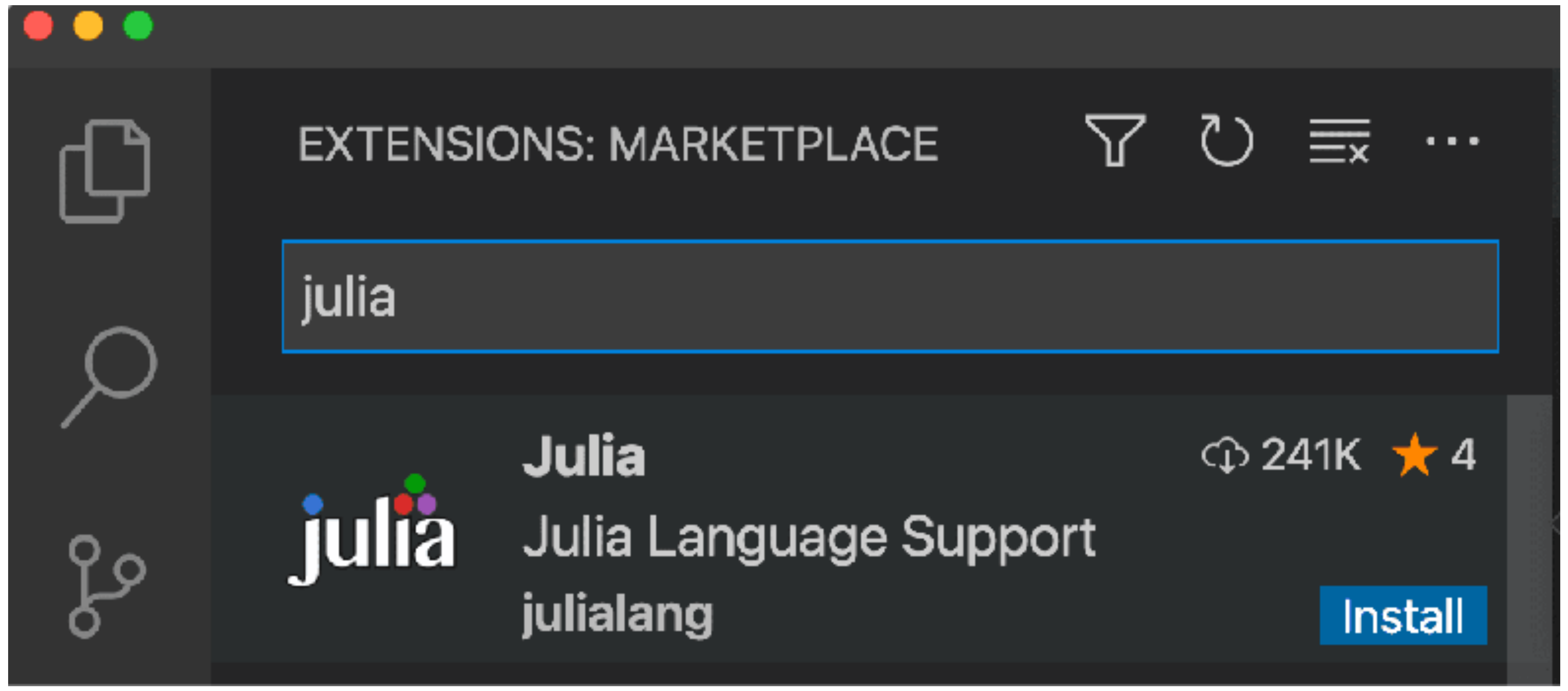
Why Julia?

- Designed for scientific computing from day one
- Easy syntax but high performance
- Excellent statistical and plotting packages
- Growing use worldwide
- Concepts readily transferrable to other languages, e.g. Python

Getting started

Installing Julia

- You should have already downloaded and installed Julia, from julialang.org/install/
- You should also have downloaded and installed Visual Studio Code, from code.visualstudio.com/download
- You should have installed the Julia extension for VSCode. If not, open **VS Code Marketplace**, find the Julia extension and press **Install**



Getting started

Opening Julia

- In Terminal/Command Prompt: type **julia** and press Enter
- In VSCode: Ctrl+Shift+P (Mac: Command+Shift+P) → “Julia: Start REPL”
- VSCode gives you the best of both worlds: interactive coding + script files

- Tip: You can register for an educational licence for GitHub Copilot and integrate this within VSCode for AI assistance

Getting started

Using Julia as your calculator

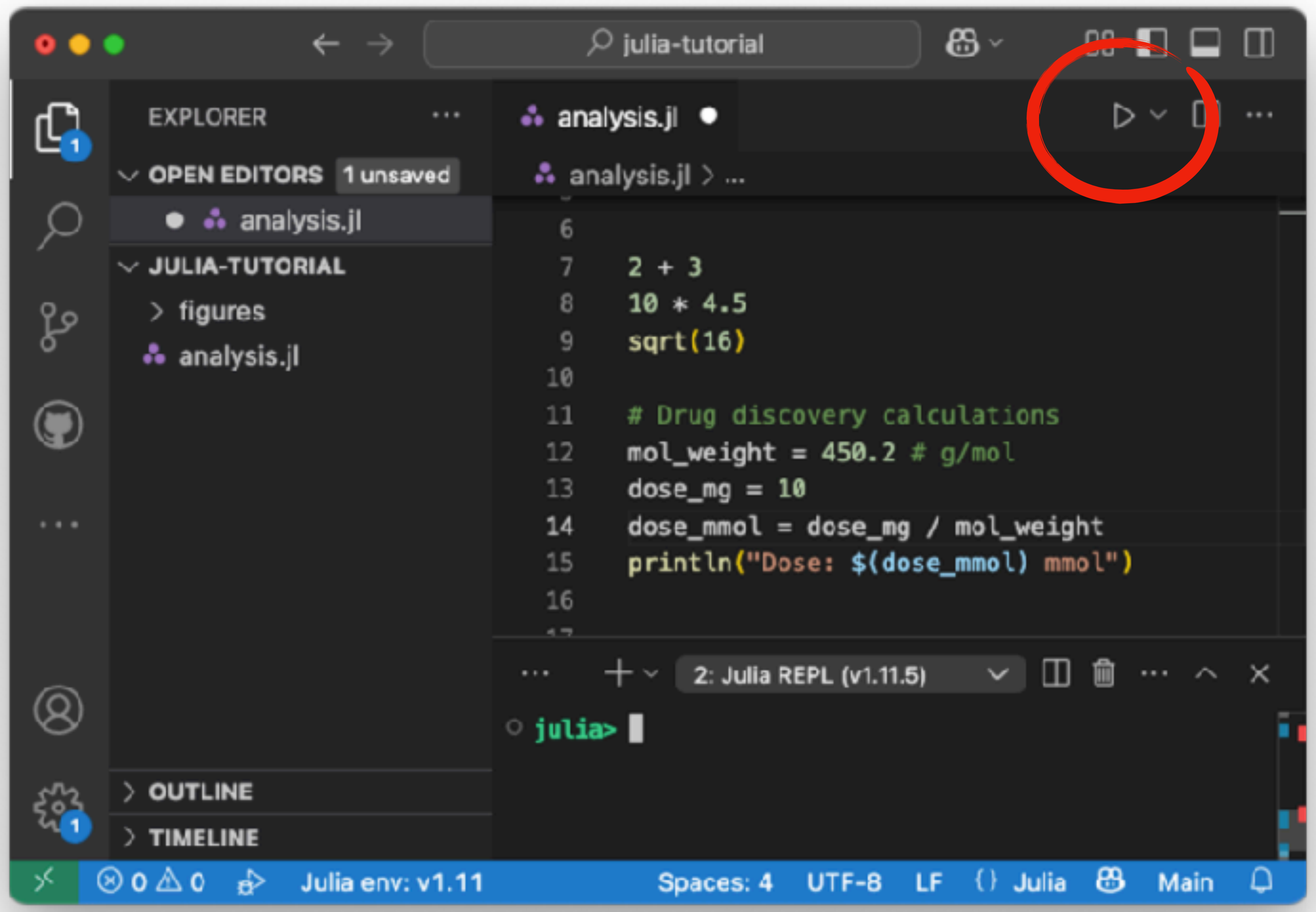
- Basic arithmetic
- Drug discovery calculations with proper units in comments
- Greek letters: type \alpha then Tab to get α
- Special symbols: type \pm then Tab to get ±

```julia
2 + 3
10 * 4.5
sqrt(16)

# Drug discovery calculations
mol_weight = 450.2 # g/mol
dose_mg = 10
dose_mmol = dose_mg / mol_weight
println("Dose: $(dose_mmol) mmol")
```

Setting up your workspace

Organising projects

- Create a folder for your project, e.g. "julia_tutorial"
- Open this folder in VSCode
- Create a subfolder "figures" to save figures
- Create a script file: "analysis.jl"
- Run scripts with Alt/Option+Enter or press play



```

2 + 3
10 * 4.5
sqrt(16)

# Drug discovery calculations
mol_weight = 450.2 # g/mol
dose_mg = 10
dose_mmol = dose_mg / mol_weight
println("Dose: $(dose_mmol) mmol")

```

```
julia> 
```

Setting up your workspace

Loading additional packages

- Julia has lots of additional packages available - these can be downloaded and installed automatically
- **CairoMakie**: High-quality scientific plotting
- **Measurements**: Automatic uncertainty propagation
- **LsqFit**: Professional curve fitting
- **Statistics**: Basic statistical functions

```
using CairoMakie
using Measurements
using LsqFit
using Statistics
```

If you haven't downloaded a package, you'll normally be prompted by the package manager to download it the first time you try to use it.

If necessary you can add packages manually, e.g.:
import Pkg; Pkg.add("CairoMakie")

Variables and lists – storing your data!

Variables

- Store single experimental values with descriptive names
- Always include units in comments for clarity
- Use meaningful names: **protein_conc** not **x1**
- Julia automatically handles different number types (integers, decimals)

```
# creating variables
protein_conc = 50.0  # μM – clear, descriptive
Kd_estimate = 2.3    # μM – units matter!
```

Variables and lists – storing your data!

Arrays (Lists) - Multiple Data Points

- Store entire datasets: e.g. dose ranges, time series, replicate measurements

- Square brackets create arrays:
  [0, 1, 2, 5, 10]

- Entries are separated by commas

- Perfect for experimental data series

- Each element accessed by position:
  concentrations[1] gives first value

```
# experimental examples
concentrations = [0, 1, 2, 5, 10, 20, 50]  # µM
times = [0, 5, 10, 30, 60, 120]  # minutes
replicates = [0.45, 0.47, 0.46]  # absorbance
```

If you've worked in other languages (e.g. C or Python), be aware – Julia arrays start at one and not zero!

Variables and lists – storing your data!

Useful array operations

- There are many **functions** that operate of arrays
- e.g. to calculate the length of the list, maximum values, statistics…
- Arrays are the foundation for all plotting and data analysis!

```
# experimental examples
concentrations = [0, 1, 2, 5, 10, 20, 50]  # µM
times = [0, 5, 10, 30, 60, 120]  # minutes
replicates = [0.45, 0.47, 0.46]  # absorbance

length(concentrations) # how many data points?
maximum(absorbance)  # highest value
mean(replicates)  # average of measurements
std(replicates) # standard deviation
```

Variables and lists – storing your data!

Transferring data from Excel

- Copy a column from Excel (not rows with commas!)
- Paste as vertical array in Julia
- Always add units in comments
- There are more sophisticated ways to read data from files but this is a quick and easy way to get started!

```julia
# create a list with commmas
my_list = [0.45, 0.47, 0.31, 0.38]

# or paste as a column directly from Excel
data_from_excel = [
    0.45
    0.47
    0.31
    0.38
]
```

Variables and lists – storing your data!

Two-dimensional arrays

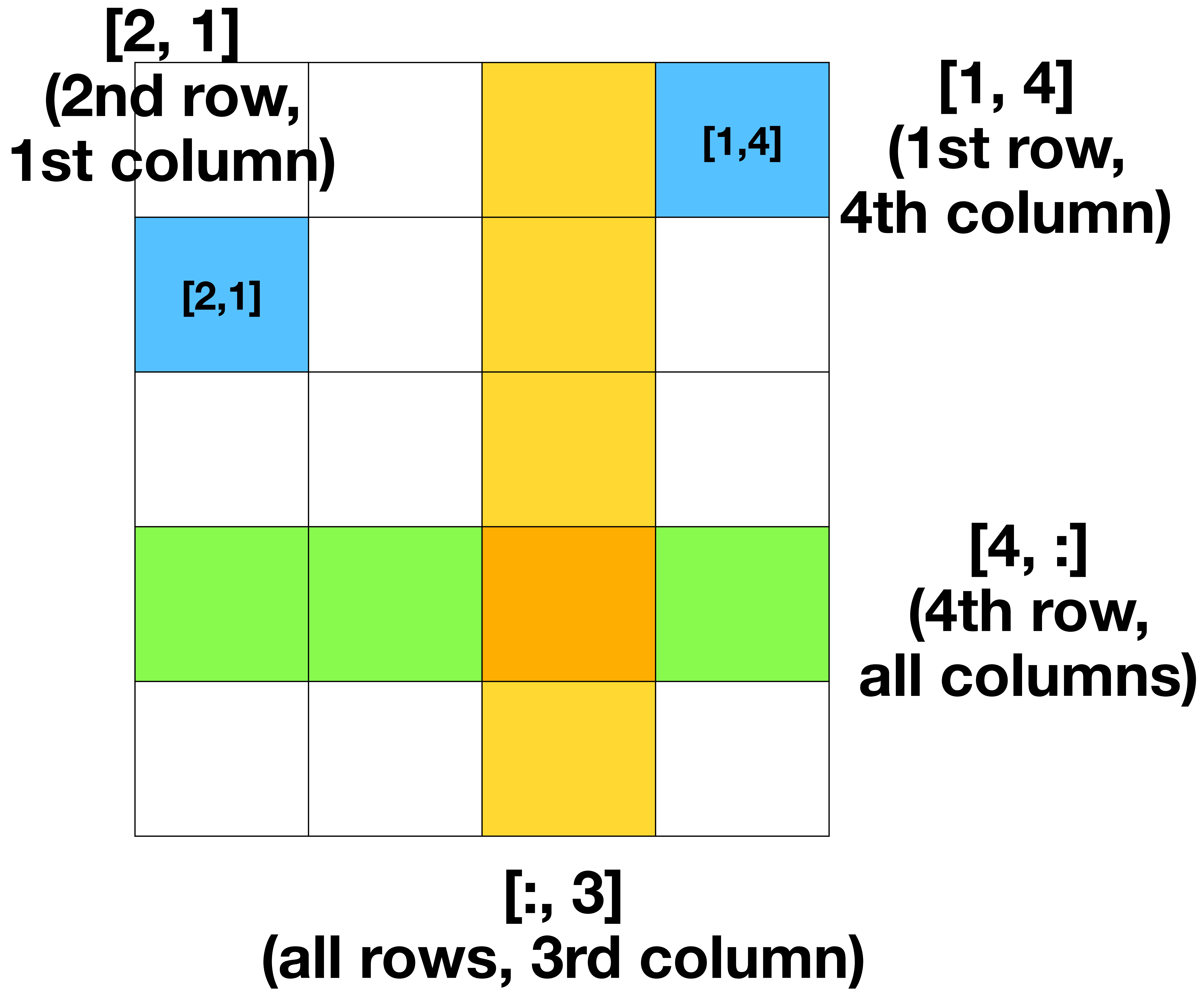
- Store data in rows and columns, like a spreadsheet or data table
- Perfect for multiple measurements across different conditions
- Example: CSP values for different residues (columns) at different ligand concentrations (rows)

```
# two–dimensional data (units: ppm)
residue_csps = [
    –0.014 0.016 –0.005
    0.040 0.008 0.037
    0.087 0.017 0.069
    0.132 0.024 0.103
    0.214 0.093 0.177
]

# accessing data
residue_csps[2, 1] # row 2, column 1
residue_csps[:, 1] # entire first column
residue_csps[3, :] # entire third row
```

Variables and lists – storing your data!

Two-dimensional arrays

- Store data in rows and columns, like a spreadsheet or data table
- Perfect for multiple measurements across different conditions
- Example: CSP values for different residues (columns) at different ligand concentrations (rows)



Variables and lists – storing your data!

Best practices

- One array per experimental variable
- Keep related data together
- Comment everything with units
- Use descriptive variable names
  you'll remember next month!

Calculations with uncertainty

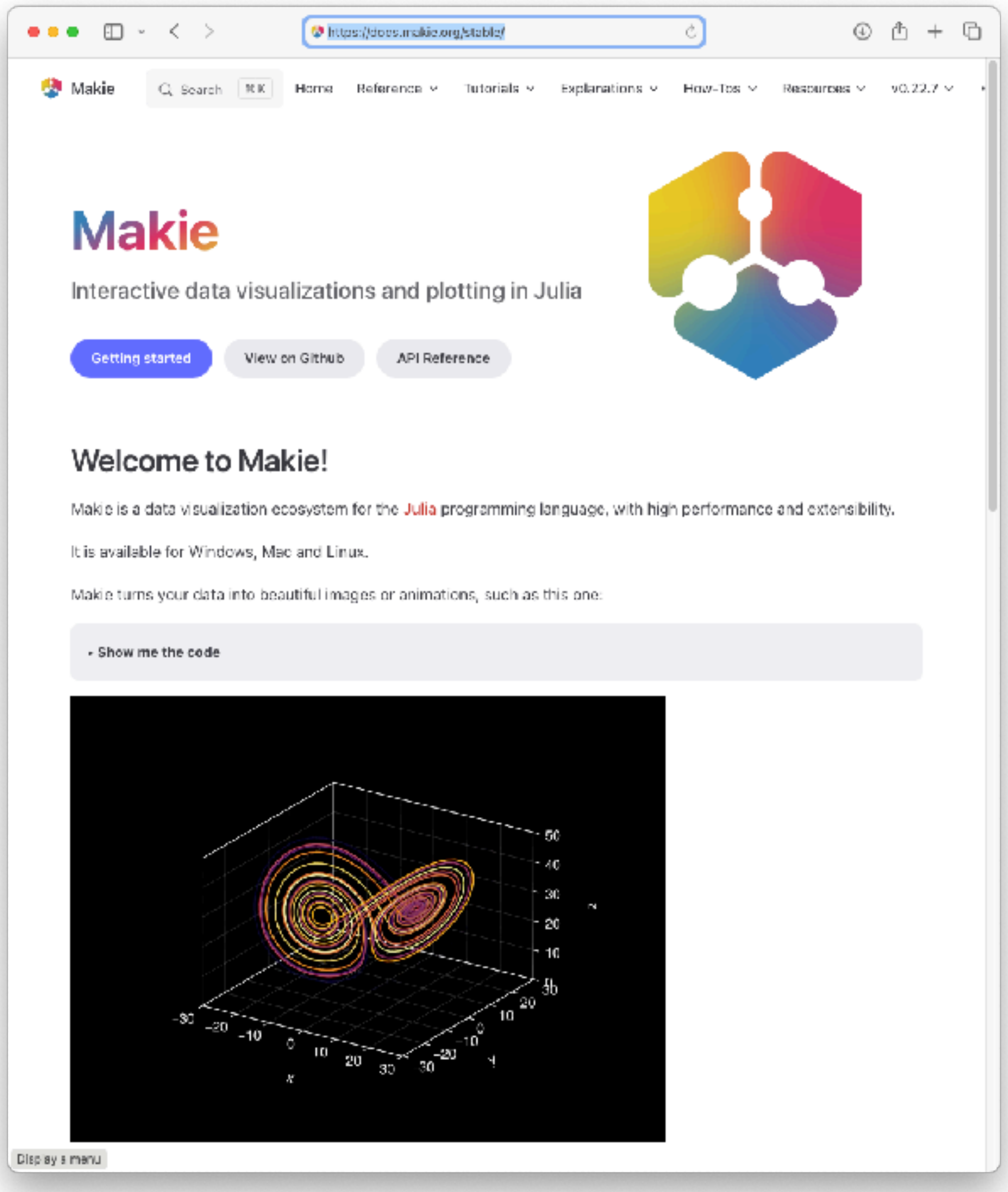
Let Julia do the maths!

- Every lab measurement has error
- Manual error propagation is tedious and error-prone
- One mistake ruins your entire calculation chain
- Julia's **Measurements.jl** package does this automatically and correctly
- Get ± symbol: type \pm then Tab

```julia
# Pipetting errors, instrument precision
volume = 100.0 ± 2.0     # µL
concentration = 10.5 ± 0.3  # mM

# Automatic error propagation
total_amount = volume * concentration
println("Total: $(total_amount) nmol")

# More complex example
log_ic50 = -2.2 ± 0.05
ic50 = 10^log_ic50
println("IC50 = $(ic50) M")
```

Plotting with CairoMakie

- CairoMakie is a package for publication-quality plotting
- It's part of the Makie ecosystem – other packages like GLMakie provide features like interactive plots and 3D plots
- Makie is based on the concept of **Figures**, **Axes** and **Plots** – similar to how you would draw a graph on paper:
  - **Figure**: Choose your paper size and layout
  - **Axis**: Draw and label your coordinate system on the figure
  - **Plot**: Add your experimental data points or lines to the axes



Makie

Interactive data visualizations and plotting in Julia

Welcome to Makie!

Makie is a data visualization ecosystem for the Julia programming language, with high performance and extensibility.

It is available for Windows, Mac and Linux.

Makie turns your data into beautiful images or animations, such as this one:

http://docs.makie.org

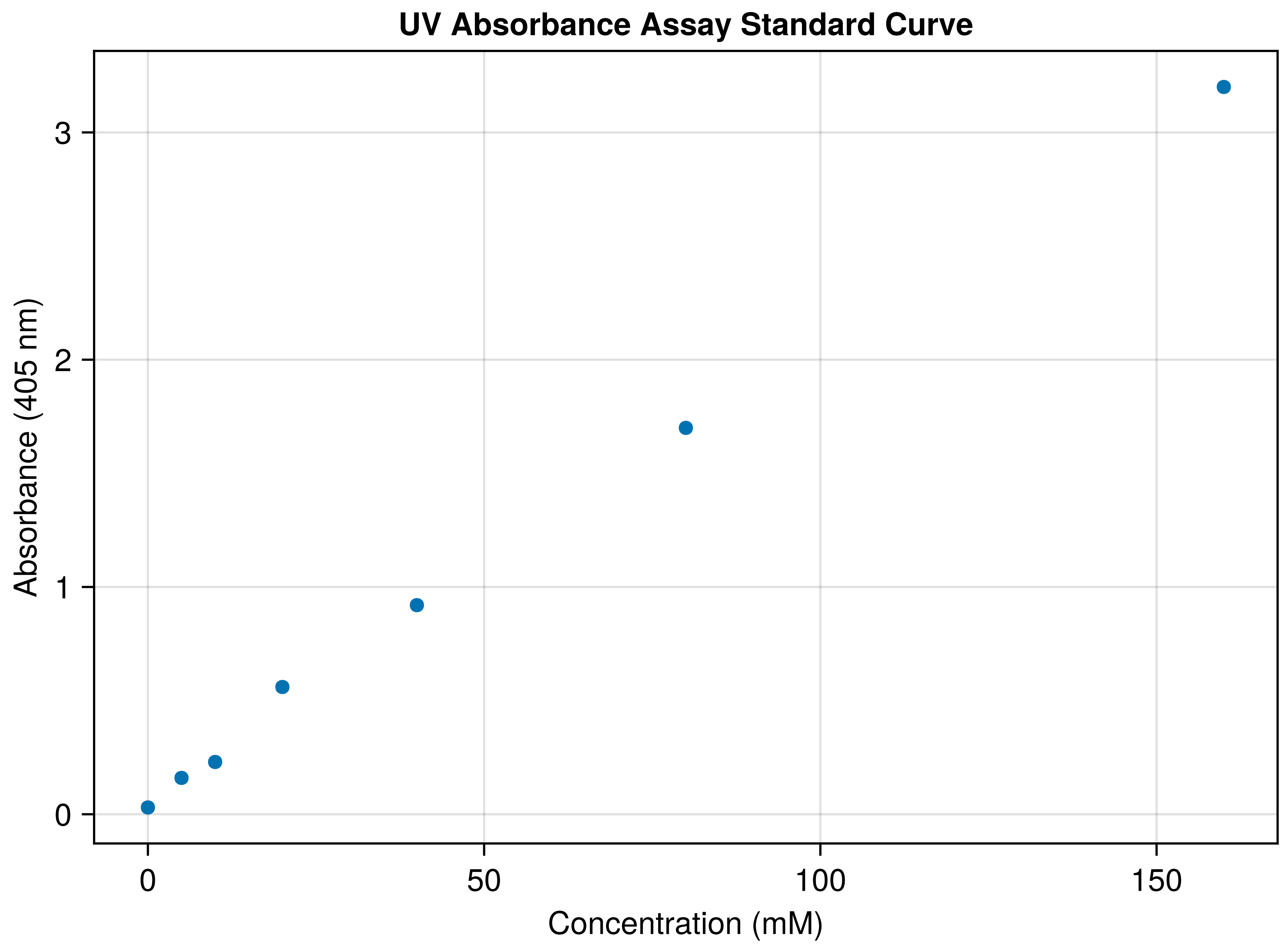
Plotting a standard curve

- Step-by-step construction:
  - Figure → Axis → Data → Display
- Build good habits:
  - Proper axis labels with units
  - Descriptive titles
  - Professional appearance from day one
- Pro tip – save early, save often!

```julia
# UV standard curve data
concentration = [0, 5, 10, 20,
                 40, 80, 160]  # mM
absorbance = [0.03, 0.16, 0.23, 0.56,
              0.92, 1.70, 3.20]  # A405nm

# Step 1: Create the figure (paper)
fig = Figure()

# Step 2: Create an axis (draw the axes)
ax = Axis(fig[1, 1],
    xlabel="Concentration (mM)",
    ylabel="Absorbance (405 nm)",
    title="UV Absorbance Assay Standard Curve")

# Step 3: Add data to the axes
scatter!(ax, concentration, absorbance)

# Step 4: Display the plot and save as a pdf
display(fig)
save("figures/standard_curve.pdf", fig)
```

Plotting a standard curve

- Step-by-step construction:
  - Figure → Axis → Data → Display
- Build good habits:
  - Proper axis labels with units
  - Descriptive titles
  - Professional appearance from day one
- Pro tip – save early, save often!



Plotting a standard curve

Adding error bars

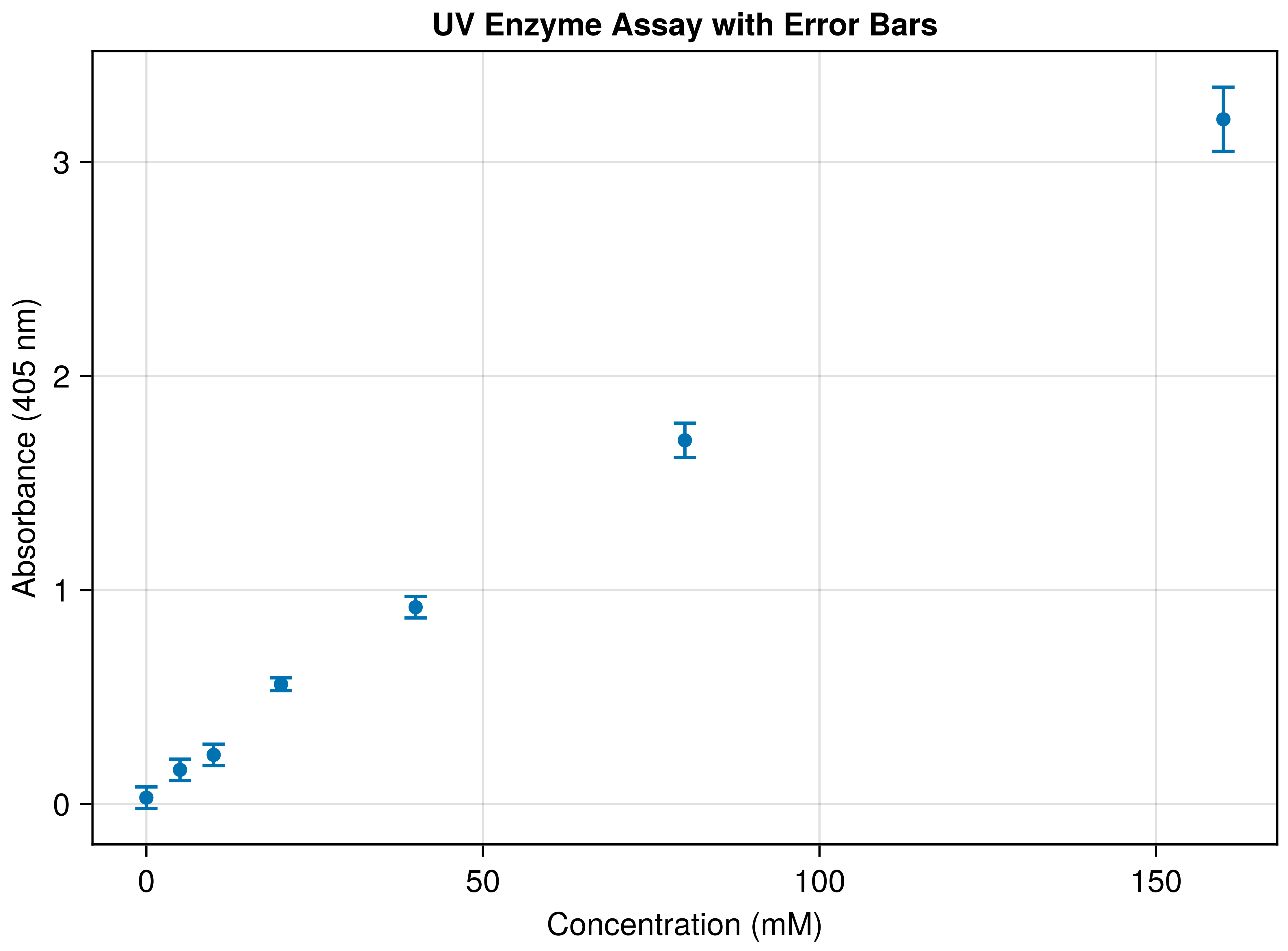
- Why error bars matter:
    - Communicate measurement precision to readers
    - Show where uncertainty actually exists
    - Distinguish reliable from unreliable data points
- Error bars can indicate either the **standard deviation** – a **descriptive** measure of the spread of values – or the standard error of the mean (usually just referred to as '**standard error**'), which is a **statistical** measure of certainty about the true value.
    - It's critical to specify in a figure legend which of these you show!

$$\text{standard error} = \frac{\text{standard deviation}}{\sqrt{N}}$$

- Plotting strategy
    - Show vertical error bars for measurement uncertainty (y-axis)
    - Include horizontal bars only if x-values are truly uncertain
    - Focus on showing error bars where the real uncertainty lies

Plotting a standard curve

Adding error bars



```
# UV standard curve data
concentration = [0, 5, 10, 20,
                 40, 80, 160]  # mM
absorbance = [0.03, 0.16, 0.23, 0.56,
              0.92, 1.70, 3.20]  # A405nm
error_values = [0.05, 0.05, 0.05, 0.03, 0.05, 0.08, 0.15]

# Step 1: Create the figure (paper)
fig = Figure()

# Step 2: Create an axis (draw the axes)
ax = Axis(fig[1, 1],
    xlabel="Concentration (mM)",
    ylabel="Absorbance (405 nm)",
    title="UV Enzyme Assay with Error Bars")

# Step 3: Add data to the axes
errorbars!(ax, concentration, absorbance, error_values,
           whiskerwidth=10)
scatter!(ax, concentration, absorbance)

# Step 4: Display the plot and save as a pdf
display(fig)
```

https://gist.github.com/chriswaudby/ab553c2fc3090cc78ff77e22cc35e8f3

Understanding functions

What is a function?

- Takes inputs (e.g. substrate concentration) → gives outputs (reaction rate)
- Describes your experimental relationships mathematically
- Foundation for all curve fitting and analysis

$$I = I_0 \exp\left(-kt\right) \qquad p_B = \frac{[L]}{[L] + K_d}$$

```julia
function intensity(t, I0, k)
    # Exponential decay model
    return I0 * exp(-k * t)
end

function pB(L, Kd)
    # Fraction ligand bound
    # in a 1:1 binding model
    return L / (L + Kd)
end

julia> intensity(0.1, 1, 2)
0.8187307530779818

julia> pB(100, 100)
0.5
```

Understanding functions

Applying functions to lists of data (‘vectorisation’)

- Functions can be applied to whole lists of data simultaneously

- Use a dot after the function name to indicate this:

  e.g. **pB.(concentrations, Kd)**

- This is very commonly used when plotting functions or fitting data

$$p_B = \frac{[L]}{[L] + K_d}$$

```julia
concentration = [0, 5, 10, 20,
                 40, 80, 160]  # mM

function pB(L, Kd)
    # Fraction ligand bound
    # in a 1:1 binding model
    return L / (L + Kd)
end

julia> pB.(concentration, 100)
7-element Vector{Float64}:
 0.0
 0.047619047619047616
 0.09090909090909091
 0.16666666666666666
 0.2857142857142857
 0.4444444444444444
 0.6153846153846154
```

Plotting functions

Create a list of x-values then calculate corresponding y-values

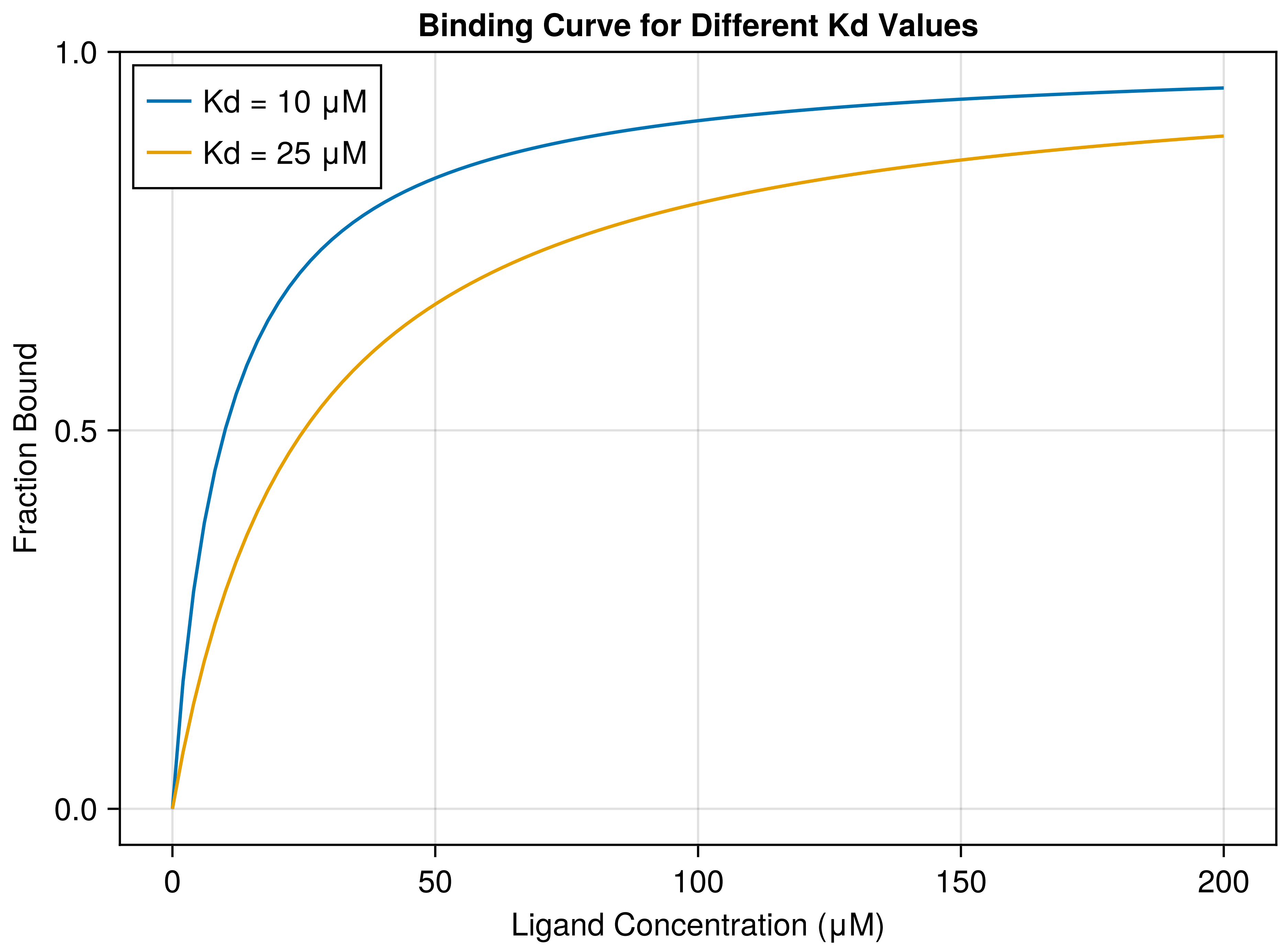
- Plotting functions is very similar to plotting data!
- Create a series of x-values using the **range** function, then calculate the corresponding y-values using your function
- Add line plots to an axis using ‘**line!**’

$$p_B = \frac{[L]}{[L] + K_d}$$

```
# Generate data for plotting
pB(L, Kd) = L / (L + Kd)  # fraction bound function
L_values = range(0, 200, length=100)  # ligand concentrations
pB_values_10 = pB.(L_values, 10)  # Kd = 10 μM
pB_values_25 = pB.(L_values, 25)  # Kd = 25 μM

# Create a new figure for the binding curve
fig = Figure()
ax = Axis(fig[1, 1],
    xlabel="Ligand Concentration (μM)",
    ylabel="Fraction Bound",
    title="Binding Curve for Different Kd Values")

# Plot the binding curves
lines!(ax, L_values, pB_values_10, label="Kd = 10 μM")
lines!(ax, L_values, pB_values_25, label="Kd = 25 μM")

axislegend(ax, position=:lt)  # add legend to left-top corner
display(fig)
```

Plotting functions

Create a list of x-values then calculate corresponding y-values



$$p_B = \frac{[L]}{[L] + K_d}$$

```julia
# Generate data for plotting
pB(L, Kd) = L / (L + Kd)  # fraction bound function
L_values = range(0, 200, length=100)  # ligand concentrations
pB_values_10 = pB.(L_values, 10)  # Kd = 10 μM
pB_values_25 = pB.(L_values, 25)  # Kd = 25 μM

# Create a new figure for the binding curve
fig = Figure()
ax = Axis(fig[1, 1],
    xlabel="Ligand Concentration (μM)",
    ylabel="Fraction Bound",
    title="Binding Curve for Different Kd Values")

# Plot the binding curves
lines!(ax, L_values, pB_values_10, label="Kd = 10 μM")
lines!(ax, L_values, pB_values_25, label="Kd = 25 μM")

axislegend(ax, position=:lt)  # add legend to left-top corner
display(fig)
```

Other types of plots…

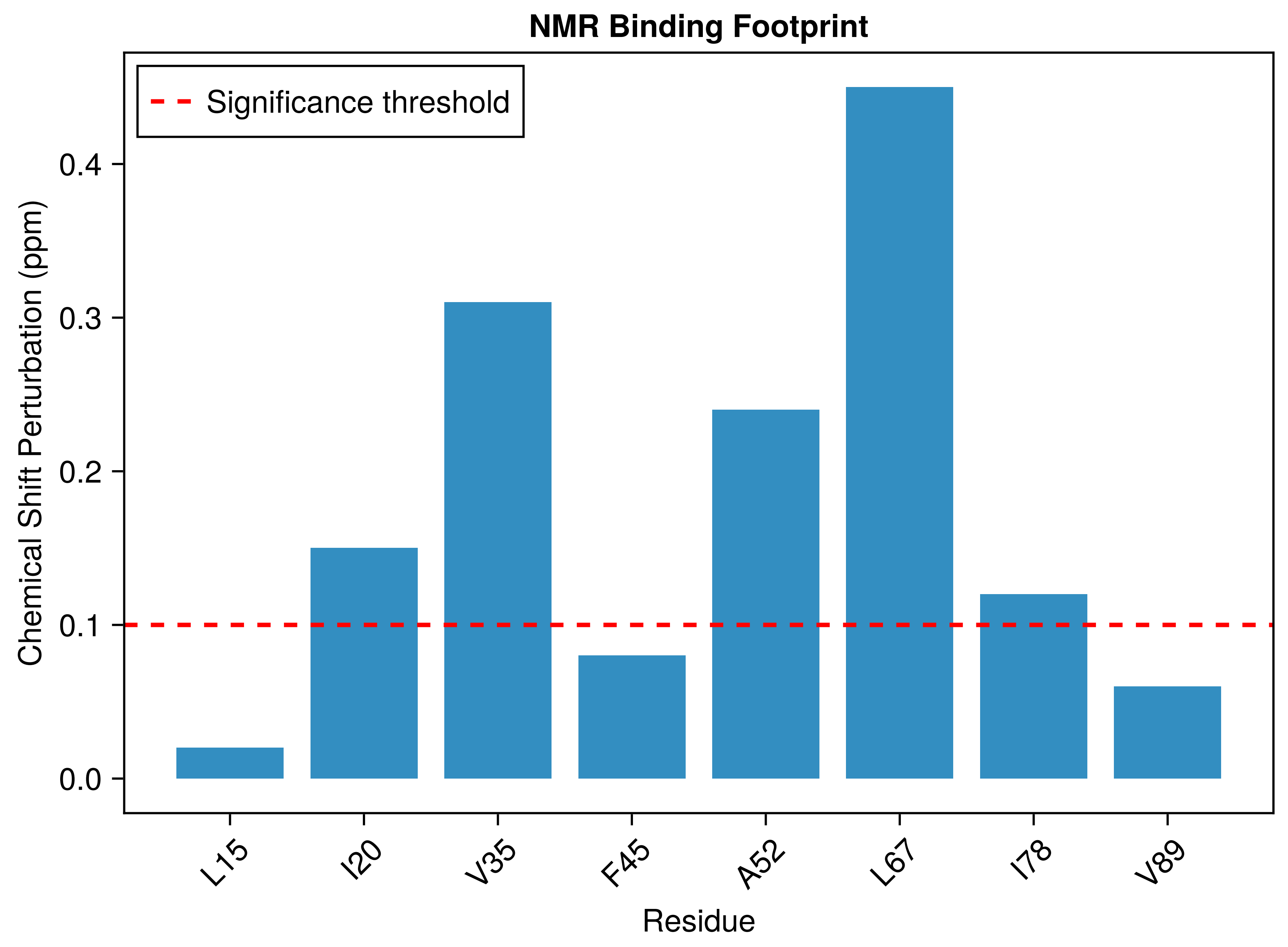
Bar plots

- NB. Syntax to create categorical (labelled) tick marks along the x axis
- Rotate labels for readability
- Add a significance threshold as a horizontal line

```
residues = ["L15", "I20", "V35", "F45", "A52", "L67", "I78", "V89"]
csp_values = [0.02, 0.15, 0.31, 0.08, 0.24, 0.45, 0.12, 0.06]  # ppm

fig = Figure()
ax = Axis(fig[1, 1],
    xlabel="Residue",
    ylabel="Chemical Shift Perturbation (ppm)",
    title="NMR Binding Footprint",
    xgridvisible=false, ygridvisible=false,
    xticklabelrotation=π / 4)

# Create bar chart
barplot!(ax, csp_values)

# Customise x-axis labels
ax.xticks = (1:length(residues), residues)

# Add significance threshold line at 0.1 ppm
hlines!(ax, 0.1, color=:red, linestyle=:dash, linewidth=2,
    label="Significance threshold")

axislegend(ax, position=:lt)
display(fig)
```

Other types of plots…

Bar plots



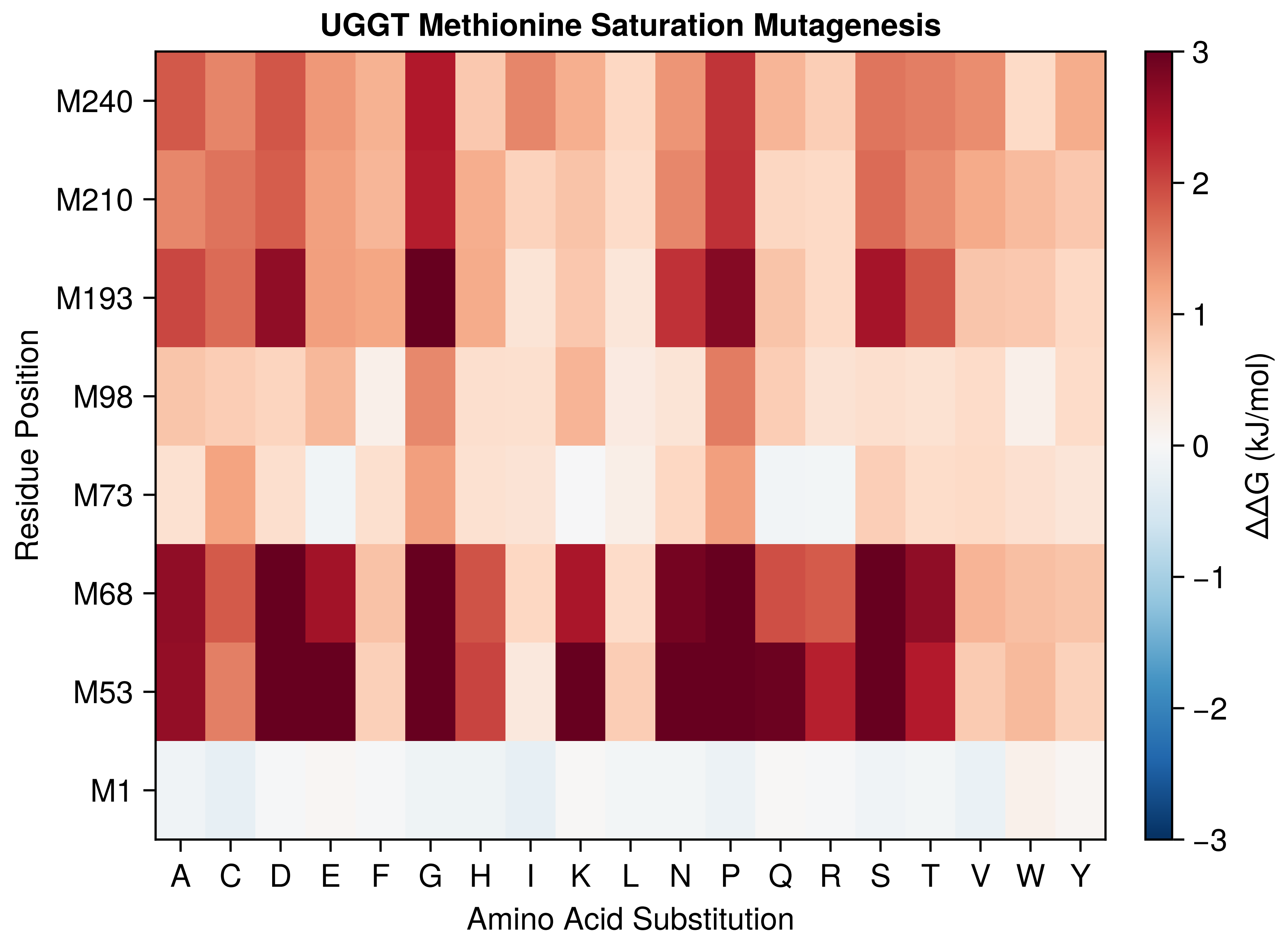
```
residues = ["L15", "I20", "V35", "F45", "A52", "L67", "I78", "V89"]
csp_values = [0.02, 0.15, 0.31, 0.08, 0.24, 0.45, 0.12, 0.06]  # ppm

fig = Figure()
ax = Axis(fig[1, 1],
    xlabel="Residue",
    ylabel="Chemical Shift Perturbation (ppm)",
    title="NMR Binding Footprint",
    xgridvisible=false, ygridvisible=false,
    xticklabelrotation=π / 4)

# Create bar chart
barplot!(ax, csp_values)

# Customise x-axis labels
ax.xticks = (1:length(residues), residues)

# Add significance threshold line at 0.1 ppm
hlines!(ax, 0.1, color=:red, linestyle=:dash, linewidth=2,
    label="Significance threshold")

axislegend(ax, position=:lt)
display(fig)
```

https://gist.github.com/chriswaudby/41c0c6b97365f34dc0604e90ed0b5978

Other types of plots…

Heat maps

- Useful for visualising large arrays of data
- Example: saturation mutagenesis:
    - Rows = residue positions, columns = amino acid substitutions
    - Colour = ΔΔG stability changes
- Design principles:
    - Diverging colourmap: red (destabilising) ↔ blue (stabilising)
    - Centre colour scale on zero



https://gist.github.com/chriswaudby/1c4f05849f7b978a5f343120aeabc503

Other types of plots...

Heat maps

```
# Data structure: rows = residue positions, columns = amino acid substitutions
residue_positions = ["M1", "M53", "M68", "M73", "M98", "M193", "M210", “M240"]

amino_acids = ["A", "C", "D", "E", "F", "G", "H", "I", "K", "L", "N", "P", "Q", "R", "S", "T", "V", "W", "Y"]

# ΔΔG matrix – copy and paste table from Excel (rows = residue, columns = amino acid)
ΔΔG_matrix = [
    -0.12 -0.26 -0.04 0.05 -0.03 -0.15 -0.15 -0.26 0.02 -0.08 -0.08 -0.17 0.03 -0.04 -0.12 -0.08 -0.2 0.16 0.06
    2.65 1.53 4.05 3.99 0.72 4.16 2.03 0.33 3.64 0.76 3.35 4.61 2.96 2.34 3.21 2.39 0.78 0.97 0.7
    2.67 1.85 3.36 2.53 0.88 4.19 1.91 0.63 2.46 0.57 2.89 3.68 1.94 1.84 3.33 2.68 1.03 0.91 0.86
    0.48 1.21 0.51 -0.11 0.5 1.25 0.48 0.42 -0.02 0.2 0.63 1.25 -0.09 -0.08 0.74 0.56 0.6 0.49 0.39
    0.85 0.76 0.67 0.99 0.17 1.45 0.51 0.5 1.03 0.27 0.41 1.56 0.76 0.45 0.51 0.46 0.58 0.17 0.58
    2.01 1.7 2.67 1.25 1.18 3.48 1.12 0.41 0.82 0.37 2.19 2.76 0.86 0.61 2.5 1.88 0.84 0.81 0.62
    1.46 1.63 1.82 1.24 1.01 2.36 1.1 0.69 0.87 0.58 1.46 2.19 0.64 0.61 1.7 1.42 1.13 0.96 0.82
    1.86 1.48 1.88 1.31 1.06 2.41 0.81 1.47 1.09 0.63 1.34 2.17 1.02 0.75 1.62 1.54 1.41 0.6 1.11
]
```

- Prepare data with lists of axis labels (residue, amino acid) and ΔΔG values

Other types of plots…

Heat maps

```
fig = Figure()
ax = Axis(fig[1, 1],
    xlabel="Amino Acid Substitution",
    ylabel="Residue Position",
    title="UGGT Methionine Saturation Mutagenesis")

# Create heat map with diverging colormap
hm = heatmap!(ax, ΔΔG_matrix', # note the transpose to match dimensions
    colormap=Reverse(:RdBu), colorrange=(-3, 3))

# Set axis labels
ax.xticks = (1:length(amino_acids), amino_acids)
ax.yticks = (1:length(residue_positions), residue_positions)

# Add colorbar
Colorbar(fig[1, 2], hm, label="ΔΔG (kJ/mol)")

display(fig)
save("figures/heatmap.pdf", fig)
```

Our ΔΔG array was defined as:

residues (rows)
x
amino acids (columns)

If we want to plot with:

x = amino acids
y = residues

then we need to swap rows/ columns around… this is called **transposition**

See available colour maps at https://docs.makie.org/stable/explanations/colors

Multi-panel plots

Using subplots

- Step-by-step figure construction:
    - Figure → Axis → Data → Display
- You can put more than one axis into a figure!
- e.g.
    - fig[2,1] = 2nd row, 1st column
    - fig[1,2] = 1st row, 2nd column

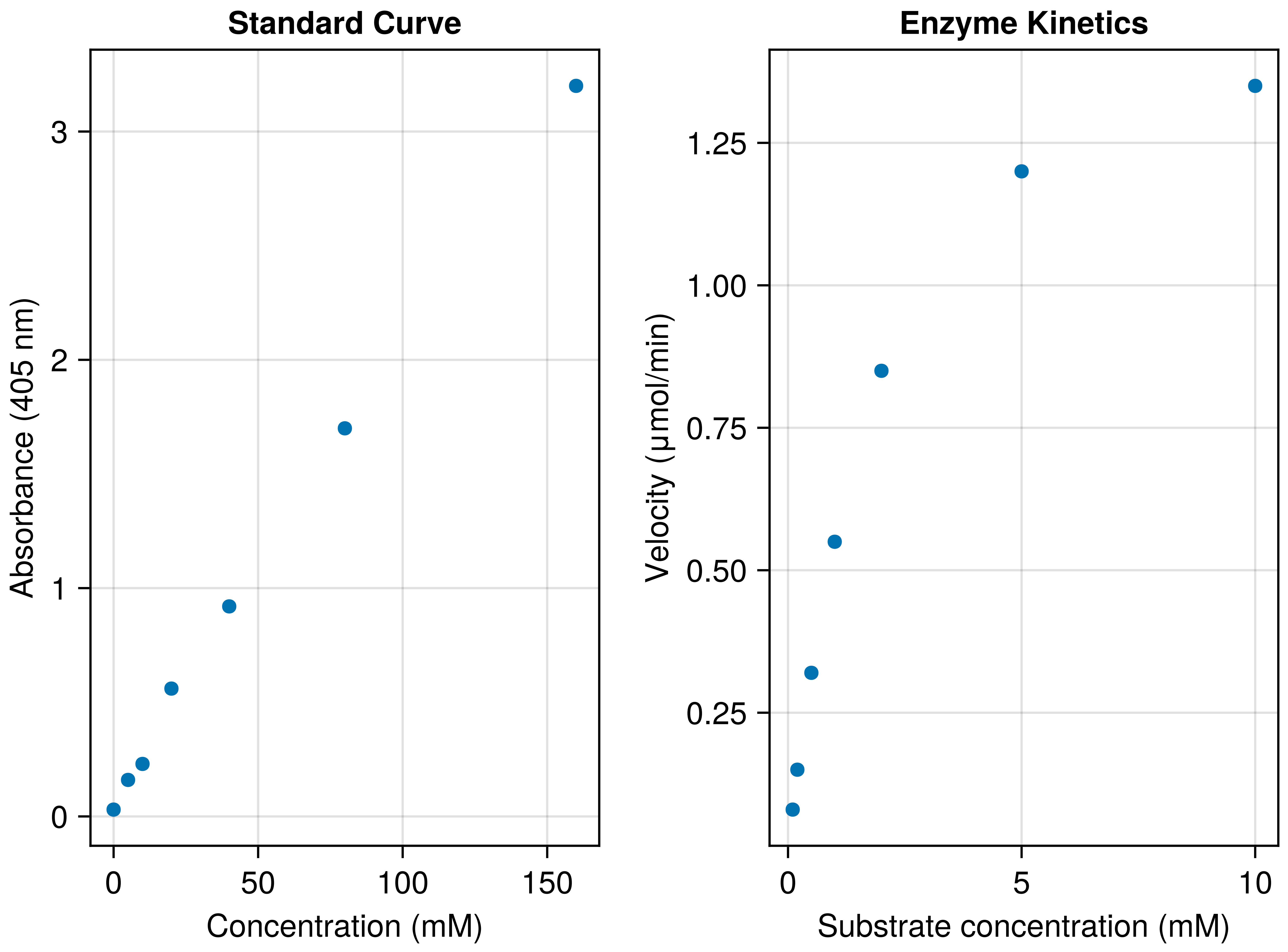
```
# Same UV standard curve data
concentration = [0, 5, 10, 20, 40, 80, 160]  # mM
absorbance = [0.03, 0.16, 0.23, 0.56, 0.92, 1.70, 3.20]  # A405

# Additional dataset - enzyme kinetics
substrate = [0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0]  # mM
velocity = [0.08, 0.15, 0.32, 0.55, 0.85, 1.20, 1.35]  # μmol/min

# Step 1: Create the figure (paper)
fig = Figure()

# Step 2: Create two axes (draw two sets of axes)
ax1 = Axis(fig[1, 1],
    xlabel="Concentration (mM)",
    ylabel="Absorbance (405 nm)",
    title="Standard Curve")

ax2 = Axis(fig[1, 2],
    xlabel="Substrate concentration (mM)",
    ylabel="Velocity (μmol/min)",
    title="Enzyme Kinetics")

# Step 3: Add data to each axis
scatter!(ax1, concentration, absorbance)
scatter!(ax2, substrate, velocity)

# Step 4: Display the plot and save
display(fig)
```

https://gist.github.com/chriswaudby/e349928fc3d42f3a207b57c88d7baf3f

Multi-panel plots

Using subplots

- Step-by-step figure construction:
  - Figure → Axis → Data → Display
- You can put more than one axis into a figure!
- e.g.
  - fig[2,1] = 2nd row, 1st column
  - fig[1,2] = 1st row, 2nd column

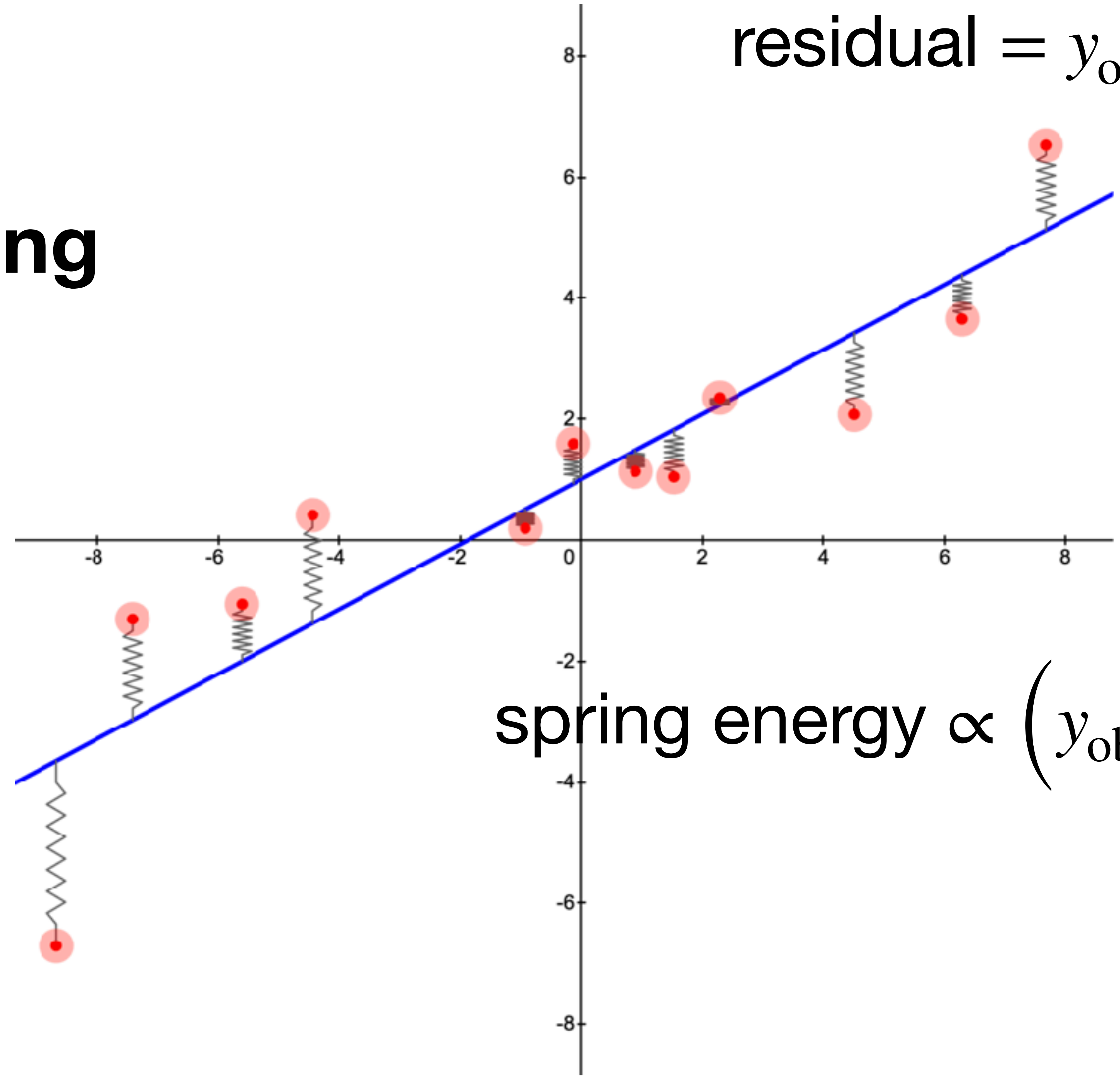


Curve fitting

Curve fitting

Principles of least-squares fitting

- Imagine tiny springs connecting your theoretical curve to each data point
- Each spring pulls with force proportional to the distance (residual)
- Least-squares fitting finds the curve position where total spring energy is minimised
- Closer fit = less stretched springs = lower total energy
- The total energy is the **sum of the squares of the residuals** – we call this quantity χ^2 (chi-squared)



residual $= y_{\text{obs}} - y_{\text{pred}}$

spring energy $\propto \left(y_{\text{obs}} - y_{\text{pred}}\right)^2$

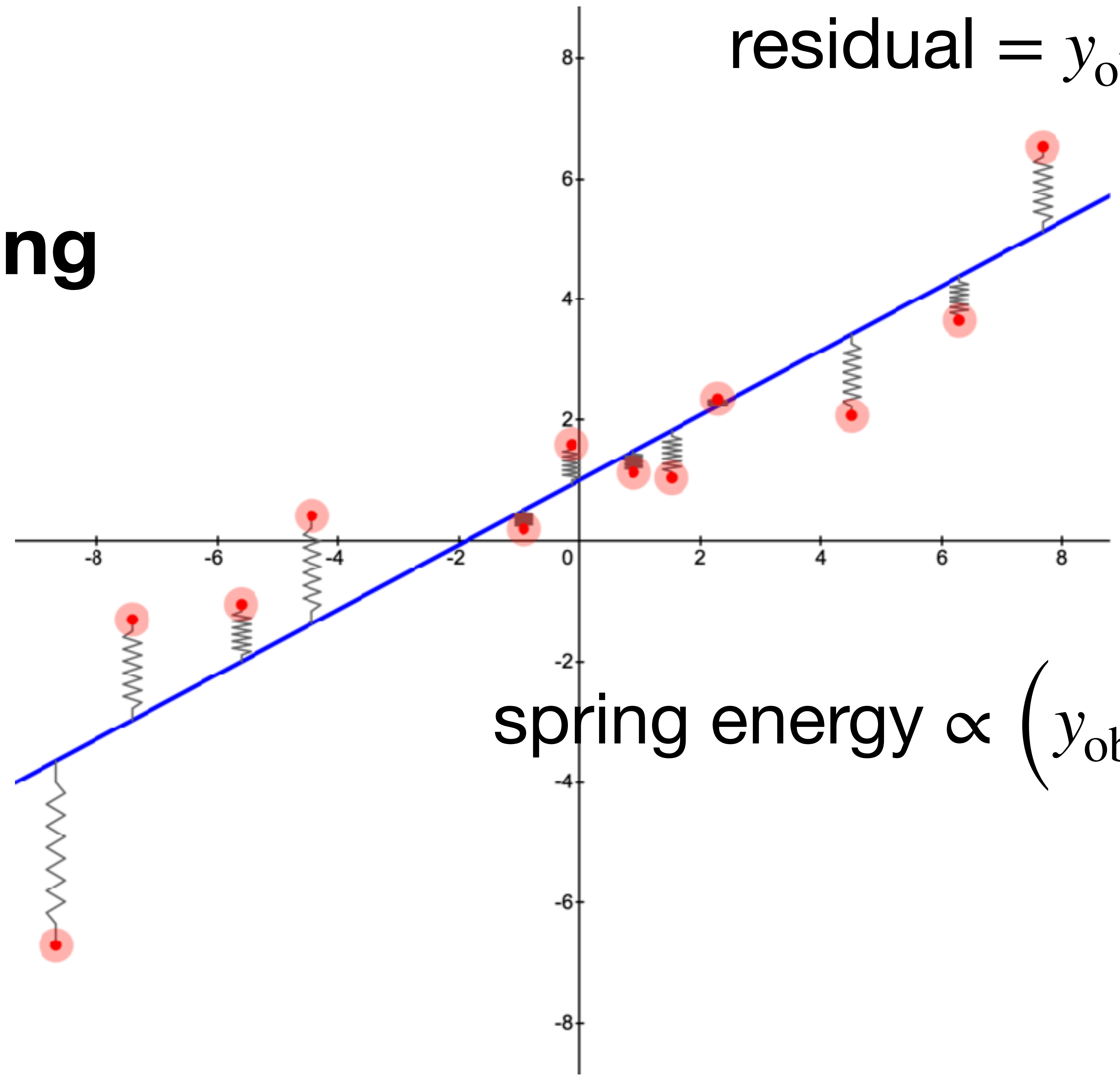
$$\chi^2 = \text{total energy} \propto \sum_{\text{points}} \left(y_{\text{obs}} - y_{\text{pred}}\right)^2$$

https://www.desmos.com/calculator/90vaqtqpx6

Curve fitting

Principles of least-squares fitting

- Fitting algorithms systematically adjust curve parameters (e.g. slope, intercept, K_d…)
- At each point, we calculate the total energy
- We look for the parameter combinations that minimise this – and report this as our fit results



residual $= y_{\text{obs}} - y_{\text{pred}}$

spring energy $\propto \left(y_{\text{obs}} - y_{\text{pred}}\right)^2$

$$\chi^2 = \text{total energy} \propto \sum_{\text{points}} \left(y_{\text{obs}} - y_{\text{pred}}\right)^2$$

https://www.desmos.com/calculator/90vaqtqpx6

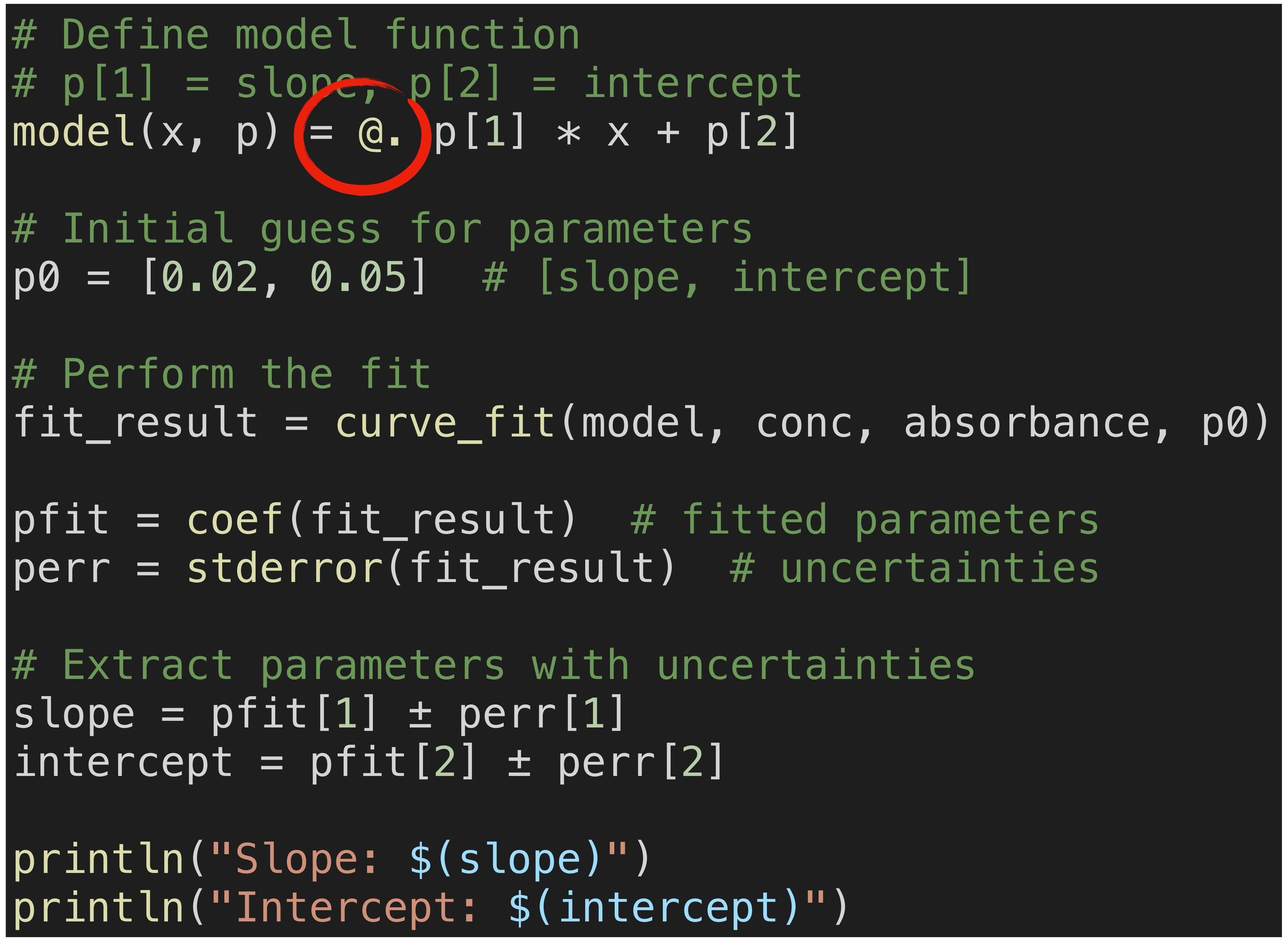
Curve fitting

Fitting a line in Julia with LsqFit

$$y = ax + b$$

$$y = p[1]x + p[2]$$

- LsqFit is a package that implements least-squares fitting
- We need to define a **model** that predicts *y* as a function of *x*, and also depends on a list of parameters *p*.
  - We use the **@.** symbol to automatically vectorise it.
- We need to give a starting guess for the parameters, p_0.
- Fitted parameters can be extracted with the **coef** function and parameter uncertainties with the **stderror** function



```julia
# Define model function
# p[1] = slope, p[2] = intercept
model(x, p) = @. p[1] * x + p[2]

# Initial guess for parameters
p0 = [0.02, 0.05]  # [slope, intercept]

# Perform the fit
fit_result = curve_fit(model, conc, absorbance, p0)

pfit = coef(fit_result)  # fitted parameters
perr = stderror(fit_result)  # uncertainties

# Extract parameters with uncertainties
slope = pfit[1] ± perr[1]
intercept = pfit[2] ± perr[2]

println("Slope: $(slope)")
println("Intercept: $(intercept)")
```

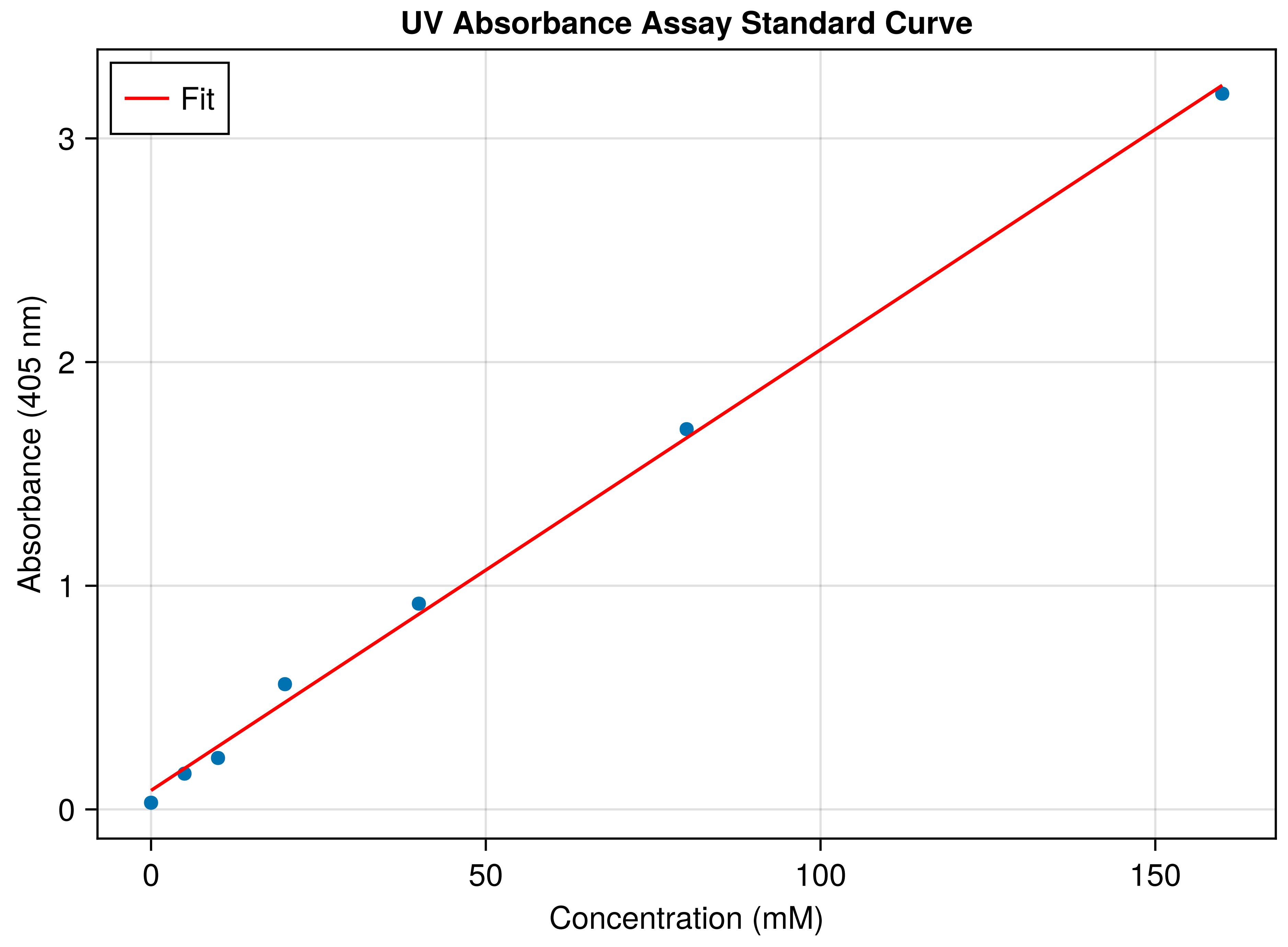
https://gist.github.com/chriswaudby/d4f16335f8e964a0f64ec8630c39edde

Curve fitting

Fitting a line in Julia with LsqFit

- Once we have our fitted parameters, we can plot the fitted model like any other function

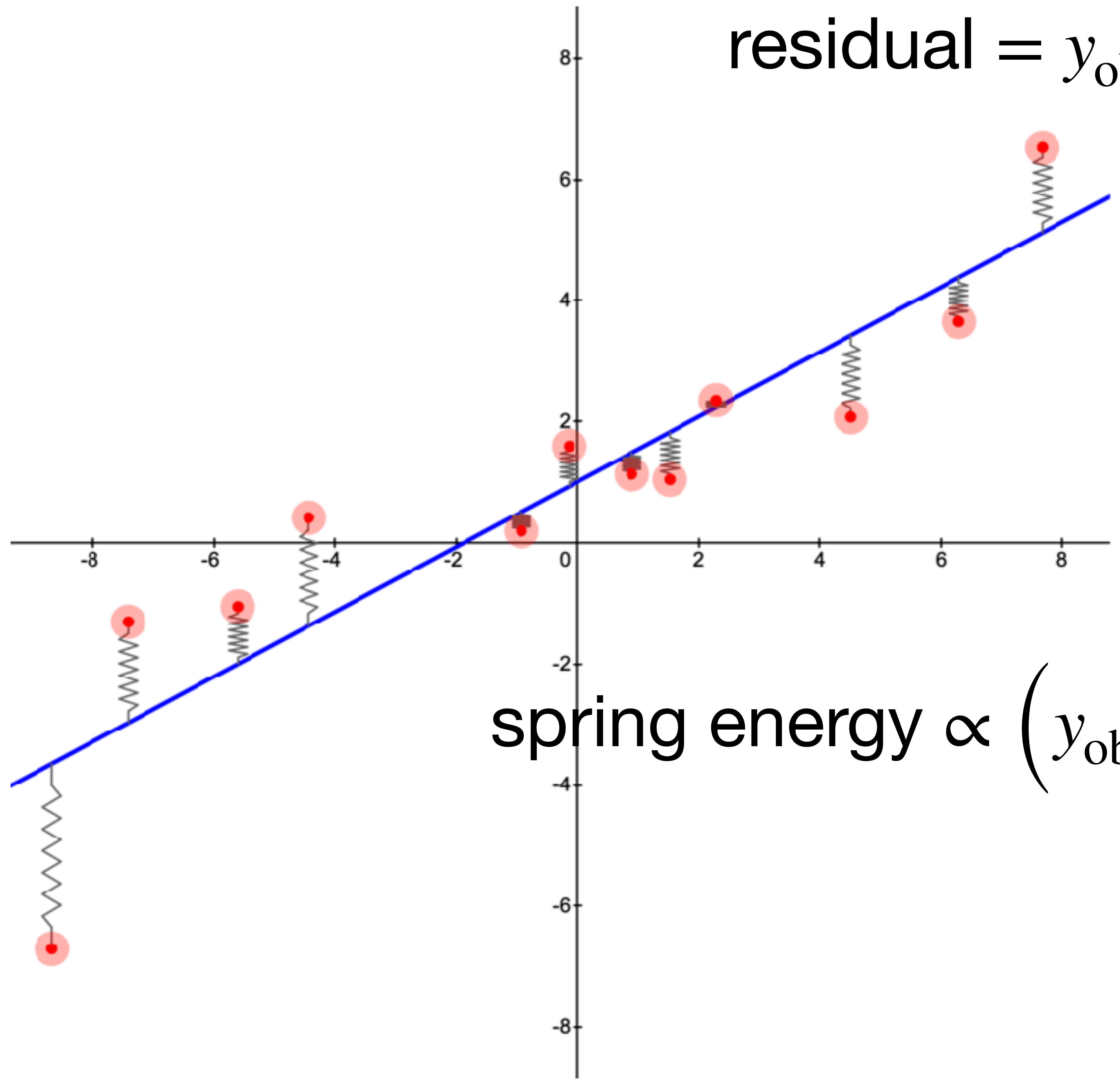
```
# add the fitted line to the plot
x_line = range(0, 160, length=100)
y_fitted = model(x_line, coef(fit_result))

lines!(ax, x_line, y_fitted, color=:red,
    label="Fit")

axislegend(ax, position=:lt)
display(fig)
```



Curve fitting

Chi-squared surfaces

- We can calculate the 'total energy' or chi-squared (χ^2) as a function of our unknown parameters (e.g. slope, intercept)
- Plotting χ^2 as a function of possible parameter values gives us a χ^2 surface



residual $= y_{\text{obs}} - y_{\text{pred}}$

spring energy $\propto \left(y_{\text{obs}} - y_{\text{pred}}\right)^2$

$$\chi^2 = \text{total energy} \propto \sum_{\text{points}} \left(y_{\text{obs}} - y_{\text{pred}}\right)^2$$

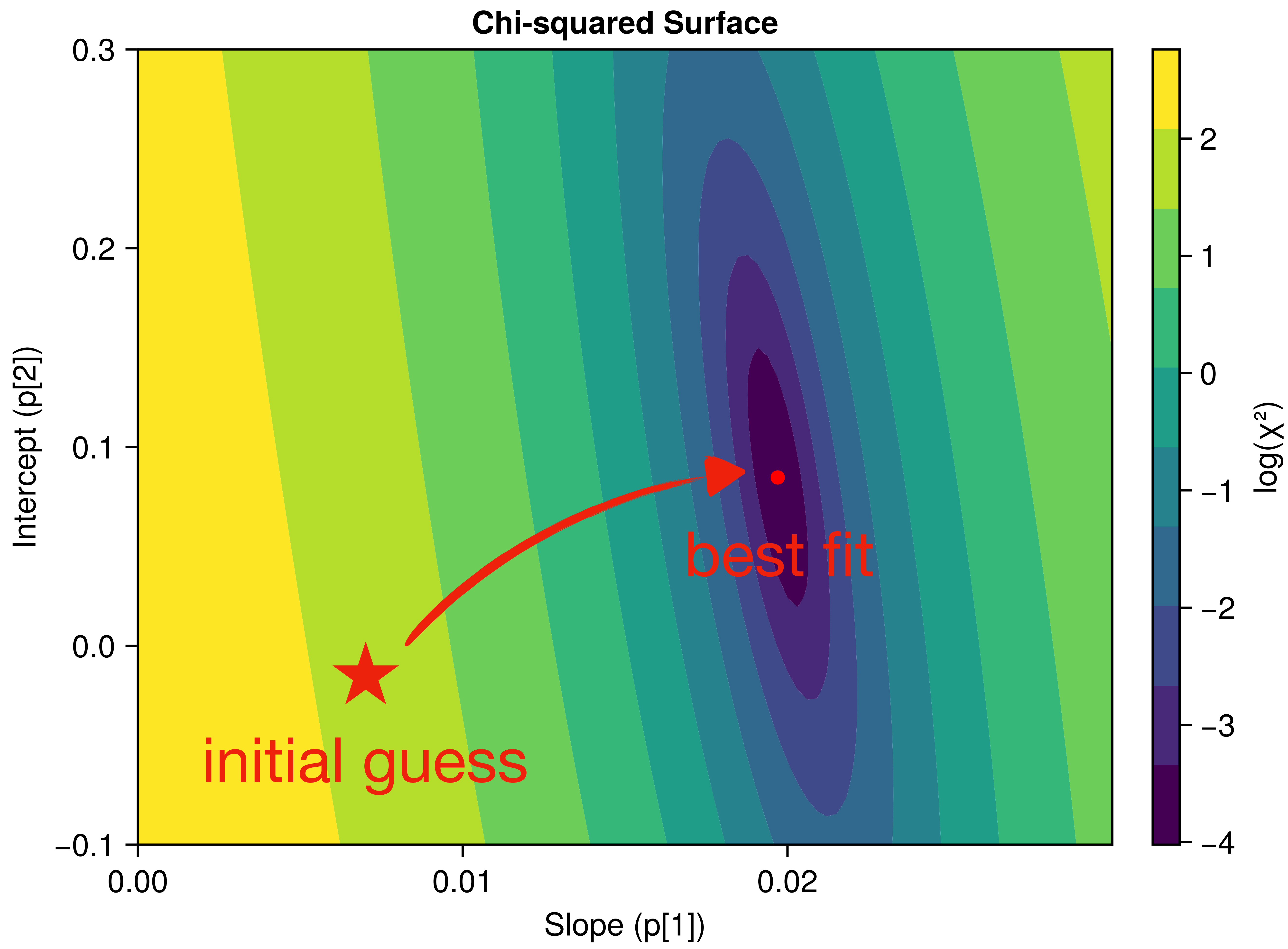
https://www.desmos.com/calculator/90vaqtqpx6

Curve fitting

Chi-squared surfaces

$$\chi^2 = \text{total energy} \propto \sum_{\text{points}} \left(y_{\text{obs}} - y_{\text{pred}}\right)^2$$

- We can plot the chi-squared surface for our line-fitting problem using Julia
- Best-fit point corresponds to the minimum on this surface ('lowest energy')
- Fitting algorithms 'explore' this surface to find a route from your starting guess to the minimum



This is also an example of how to make contour plots!

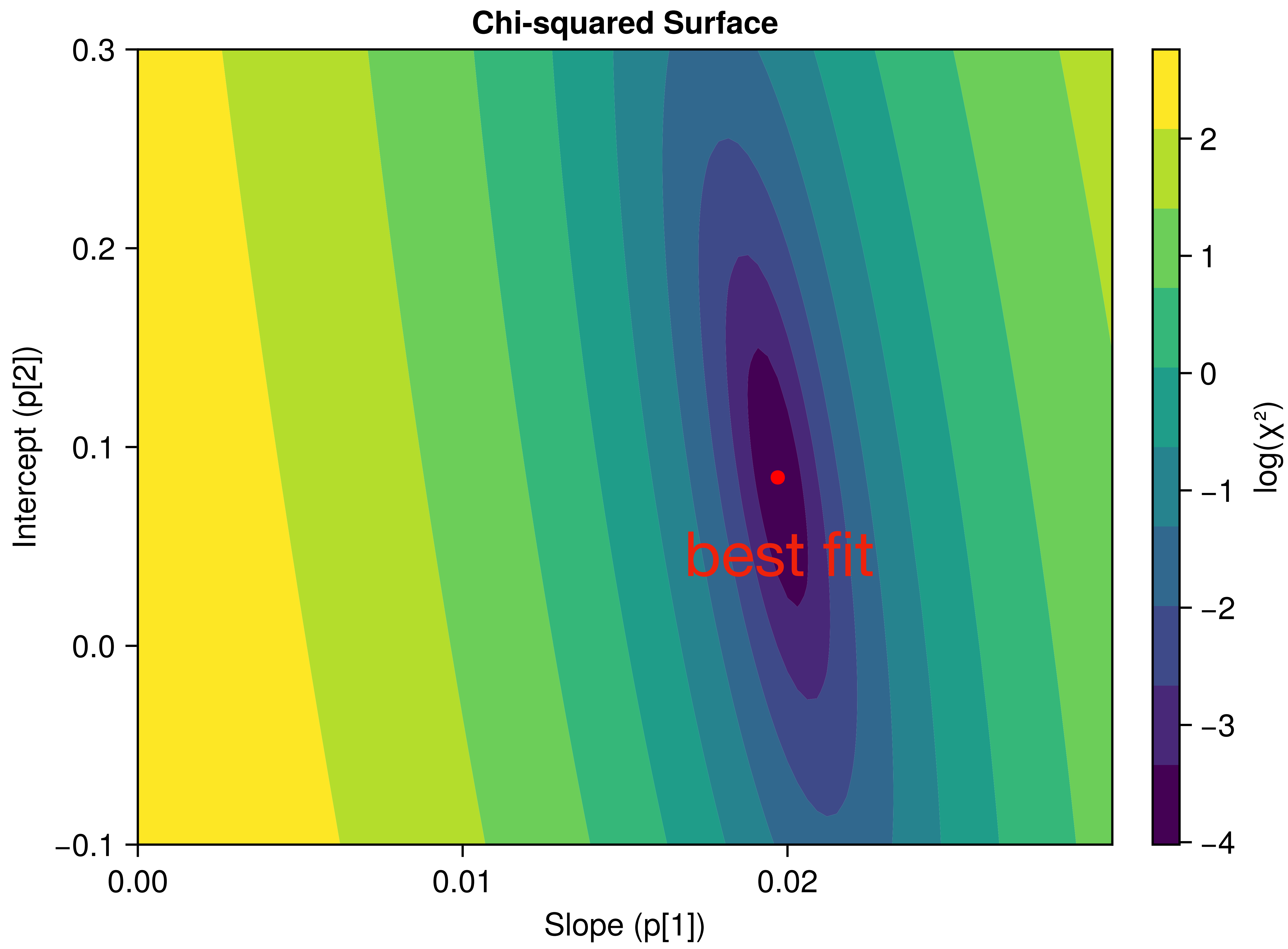
https://gist.github.com/chriswaudby/12db55537811f520a09c4128b5714c27

Curve fitting

Chi-squared surfaces & parameter uncertainty

$$\chi^2 = \text{total energy} \propto \sum_{\text{points}} \left(y_{\text{obs}} - y_{\text{pred}}\right)^2$$

- How confident are we in our fitted parameters?
- Intuitively, this depends on how 'steep' the sides of the valley are around the best fit point
- The curvature of the surface is used to calculate the uncertainty of fitted parameters
- Be aware of parameter covariance!

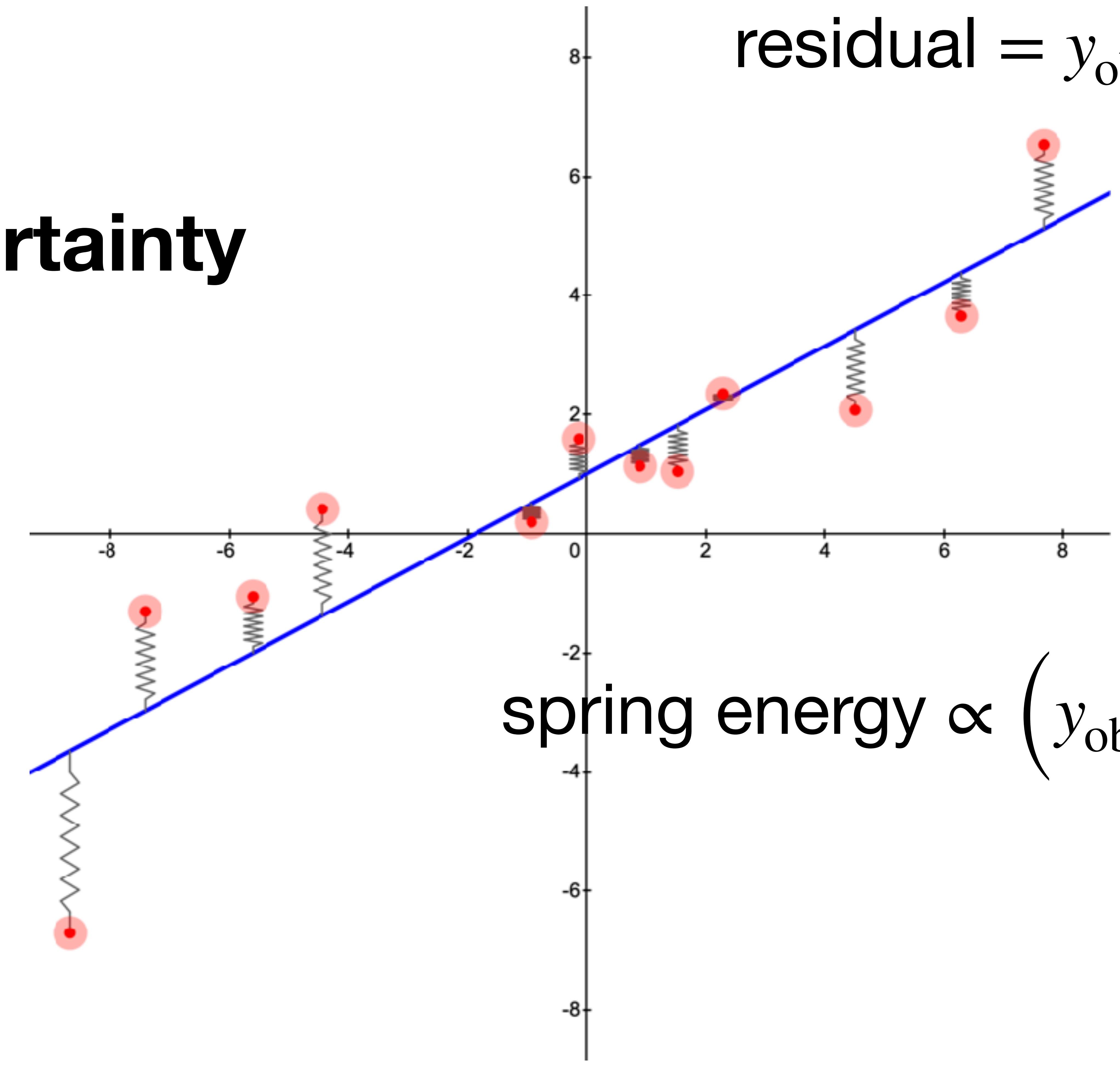


https://gist.github.com/chriswaudby/12db55537811f520a09c4128b5714c27

Curve fitting

Weighting data points by uncertainty

- How do we incorporate measurement error into fitting?
- In our previous analogy, uncertainty tells us how 'stiff' the spring is!



residual $= y_{\text{obs}} - y_{\text{pred}}$

spring energy $\propto \left(y_{\text{obs}} - y_{\text{pred}}\right)^2$

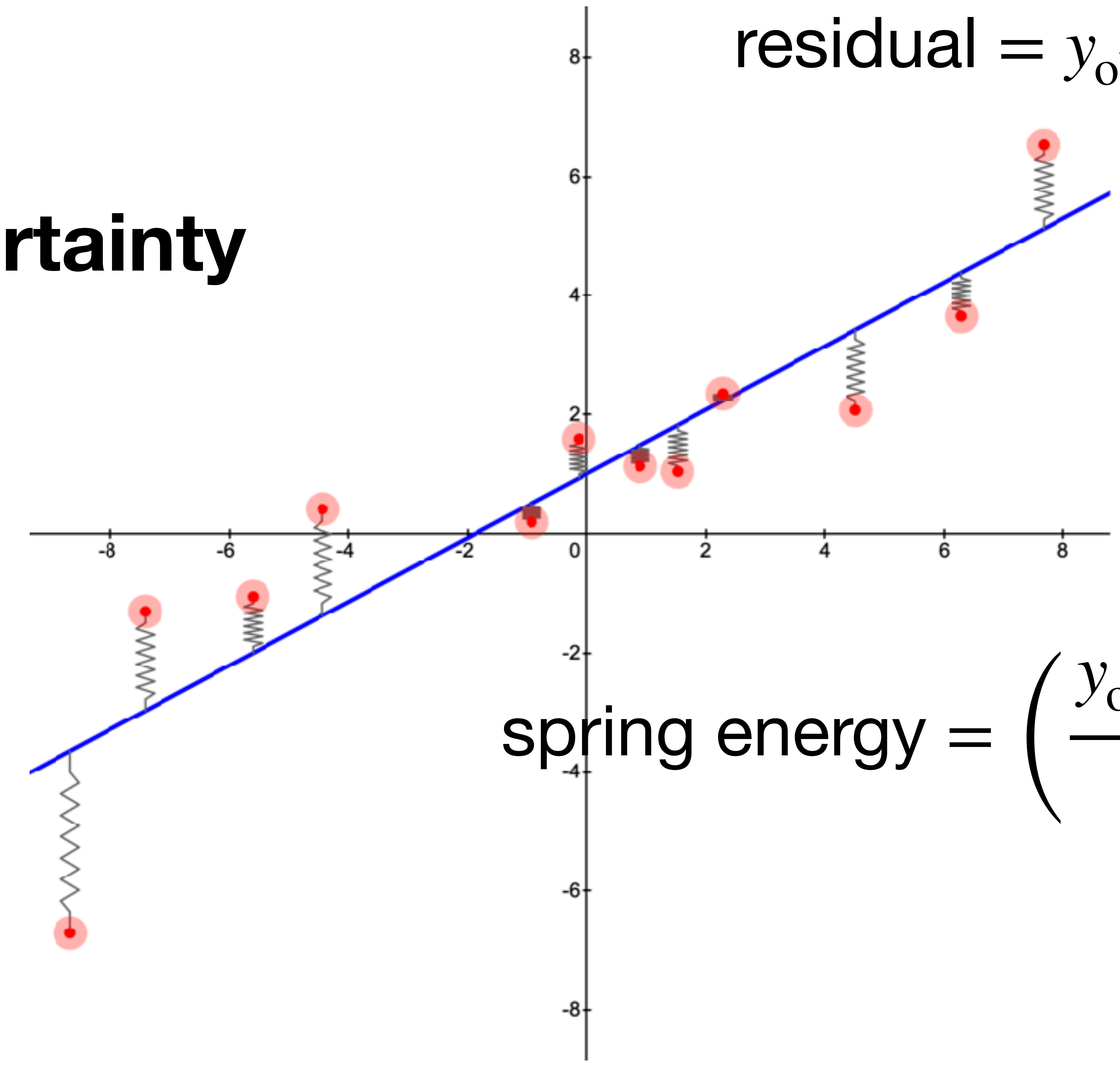
$$\chi^2 = \text{total energy} \propto \sum_{\text{points}} \left(y_{\text{obs}} - y_{\text{pred}}\right)^2$$

https://www.desmos.com/calculator/90vaqtqpx6

Curve fitting

Weighting data points by uncertainty

- How do we incorporate measurement error into fitting?
- In our previous analogy, uncertainty tells us how ‘stiff’ the spring is!
- Residuals should be normalised according to their uncertainty
- This defines an **exact** χ^2 score



$$\text{residual} = y_{\text{obs}} - y_{\text{pred}}$$

$$\text{spring energy} = \left(\frac{y_{\text{obs}} - y_{\text{pred}}}{\sigma}\right)^2$$

$$\chi^2 = \text{total energy} = \sum_{\text{points}} \left(\frac{y_{\text{obs}} - y_{\text{pred}}}{\sigma}\right)^2$$

https://www.desmos.com/calculator/90vaqtqpx6

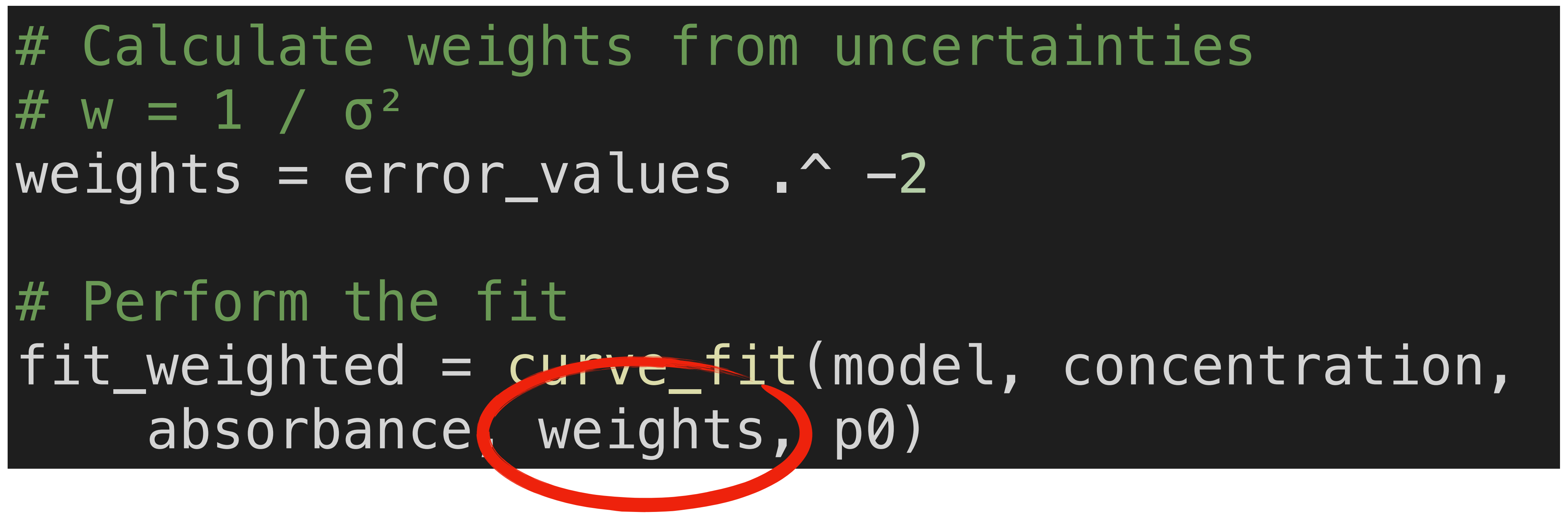
Curve fitting

Incorporating measurement error

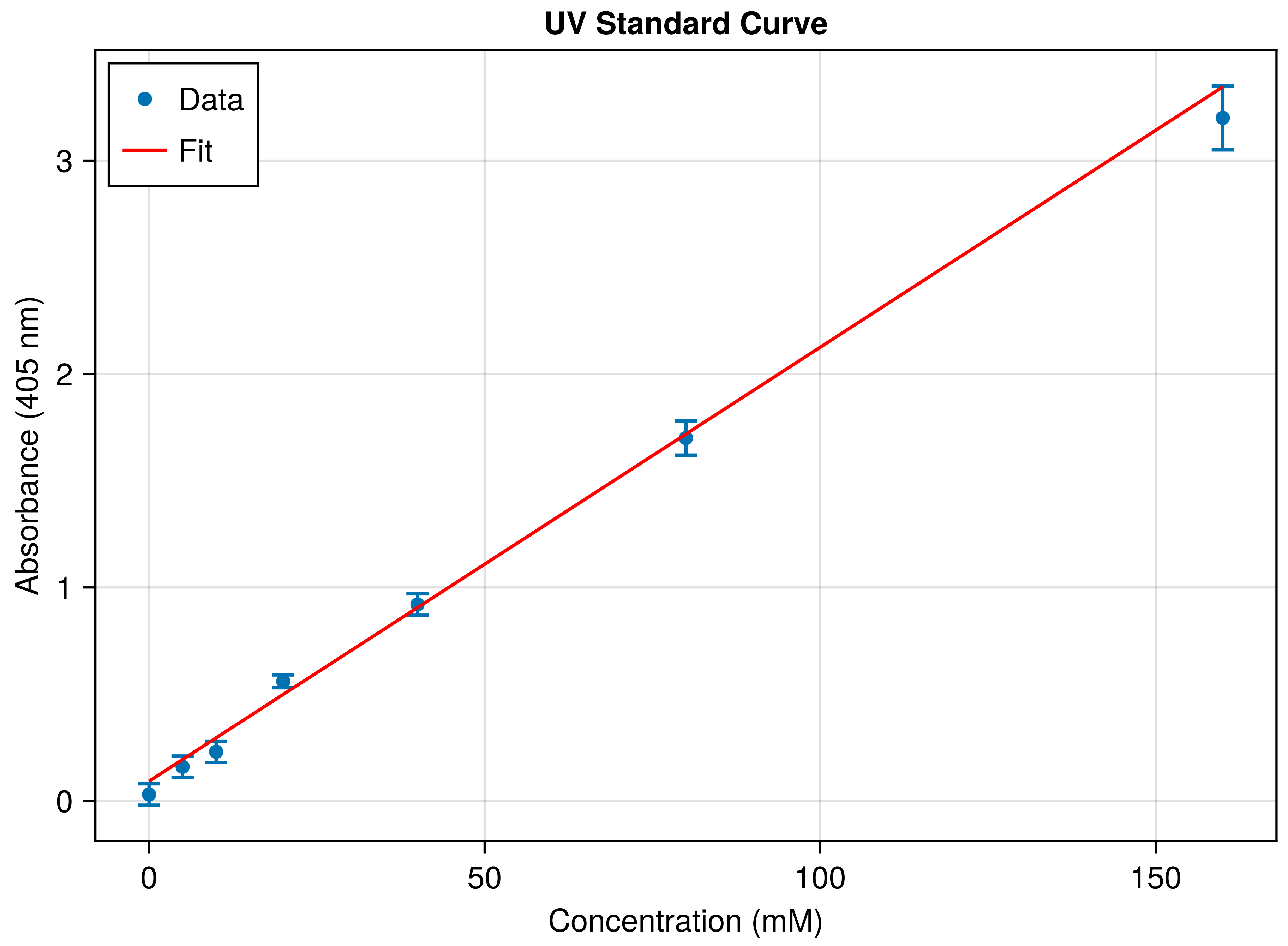
- LsqFit easily handles measurement uncertainty
- Calculate **weights** for each data point:

$$\text{weight} = \frac{1}{\text{error}^2}$$

- Pass these weights to the **curve_fit** command
- That's it!



```
# Calculate weights from uncertainties
# w = 1 / σ²
weights = error_values .^ -2

# Perform the fit
fit_weighted = curve_fit(model, concentration,
    absorbance, weights, p0)
```

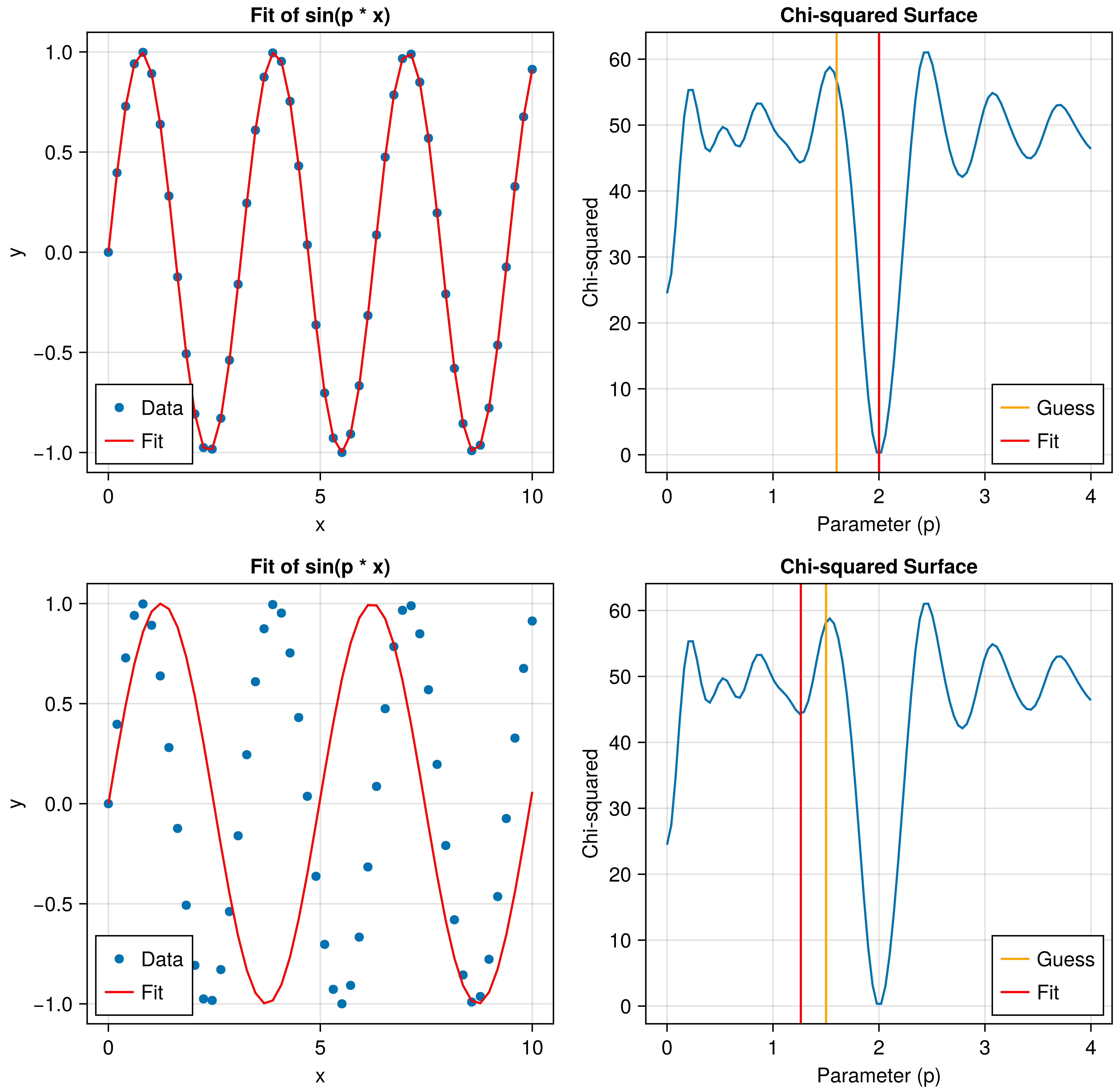


https://gist.github.com/chriswaudby/849590a39dcf1f595642a247433a8a77

Curve fitting

Global and local minima

- Not all chi-squared surfaces are as nice as for a linear fit!
- **Non-linear** models can have multiple ‘local’ minima
- Fitting algorithms can erroneously converge to these rather than the true ‘global’ minimum
- **Fitting can be very sensitive to the initial guess**



https://gist.github.com/chriswaudby/9696357f136dfe2b152739fb02abbc60

Curve fitting

Tips and tricks

- Fitting algorithms generally work better if all parameters have a similar order of magnitude

$$y(t) = A \cdot \exp(kt) \xrightarrow{\text{rescale model function}} y(t) = 10^6 \cdot A \cdot \exp(10^{-5} \cdot kt)$$

$A = 123{,}000{,}000$

$k = 0.000789$

$A = 123$

$k = 78.9$

Curve fitting

Tips and tricks

- If parameters could span a large but unknown range of values (and are always positive), consider fitting logarithms of these parameters

$$y(t) = A \cdot \exp(kt) \xrightarrow{\text{rewrite with logarithms}} y(t) = e^{lnA} \cdot \exp(e^{lnk}t)$$

A = 123,000,000

k = 0.000789

ln$A$ = 18.6

ln$k$ = -7.14

Curve fitting

Tips and tricks

- Reparameterise models to reduce covariance
- Example: it's more robust to fit a peak in terms of height and width than area and width

$$y(x) = \frac{A}{\sigma^2 + x^2} \xrightarrow{\text{reparameterise}} y(x) = \frac{H \cdot \sigma^2}{\sigma^2 + x^2}$$

A = peak area

σ = peak width

H = peak height

$$A = H \cdot \sigma^2$$

Final thoughts

Final thoughts

- **Start small:** every expert was once a beginner who kept experimenting
- **Start today:** one plot, one calculation, one “what if…”
- **Correct handling of uncertainties = better science** (more reproducible, more sustainable!)
- **Embrace the AI revolution:** GitHub Copilot + Julia = supercharged productivity
- Use Claude/ChatGPT to explain error messages and suggest approaches
- “How do I plot X in Julia?” → instant, working code
- AI excels at boilerplate code – **you bring the scientific insight!**
- Remember to report use of AI in your write-ups!
- **Your research advantage:** Julia is increasingly popular in research, and these skills and concepts are easily transferred to Python, R, MATLAB – and they’re in demand!