Week 46

Table of contents


```
try:
    import fysisk_biokemi
    print("Already installed")
except ImportError:
    %pip install -q "fysisk_biokemi[colab] @ git+https://github.com/au-mbg/fysisk-biokemi.
git"
```

Python: Introduction to curve fitting

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
```

Curve fitting is a fundamental skill in biochemistry and biophysics for analyzing experimental data. We use curve fitting to determine parameters in mathematical models that describe biological processes like enzyme kinetics, binding affinity, and reaction rates.

The basic idea is to find the parameters of a mathematical function that best describes our experimental data.

(a) Understanding the problem
Let's start with some simple data that follows a linear relationship. The data below represents a theoretical experiment where we measure some output y at different input values x:

```
# Generate some data with noise
x_data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
y_data = np.array([2.1, 4.2, 6.0, 7.8, 10.1, 12.2, 13.9, 16.1, 18.0, 20.2])
```

Looking at this data, we can see it roughly follows a straight line. Let's assume our model is:

$$y = ax + b$$

where a and b are parameters we want to determine from the data.

(b) Defining a fitting function

To use `scipy.optimize.curve_fit`, we need to define a function where:

- The **first** argument is the independent variable (x)
- The remaining arguments are the parameters to be fitted

Define a linear function for fitting:

```
def linear_function(x, a, b):
    return ... # Complete this line
```

Always a good idea to check that your function works as expected

```
linear_function(1, 2, 0) # Should give 2
```

```
2
```

Come up with another test case to check

```
print(linear_function(0, 2, 3)) # Should give 3
print(linear_function(1, 2, 3)) # Should give 5
```

(c) Your first curve fit

Now we can use `curve_fit` to find the best parameters. The basic syntax is:

```
popt, pcov = curve_fit(function, x_data, y_data, p0=initial_guess)
```

Where:

- `function`: The function we defined above
- `x_data`, `y_data`: Our experimental data
- `p0`: Initial guess for parameters (optional but recommended)
- `popt`: The optimized parameters (what we want!)
- `pcov`: Covariance matrix (contains information about parameter uncertainties)

Finish the cell below to perform the curve fit by adding the three missing arguments to the call to `curve_fit`.

```
# Initial guess: a=2, b=0
p0 = [2, 0]

# Perform the fit
popt, pcov = curve_fit(..., ..., ..., p0=p0)

# Extract parameters
a_fit, b_fit = popt
print(f"Fitted parameters: a = {a_fit:.3f}, b = {b_fit:.3f}")
```

(d) Visualizing the fit

It's crucial to always plot your fit to see how well it describes the data. To do this we evaluate the function with the fitted parameters and a densely sampled independent variable.

Then we can plot it

```
fig, ax = plt.subplots(figsize=(6, 4))
ax.plot(x_data, y_data, 'o', markersize=8, label='Experimental data')

# Then we can plot the fit
ax.plot(..., ..., '-', linewidth=2, label=f'Fit: y = {a_fit:.2f}x + {b_fit:.2f}')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()
ax.set_title('Linear curve fit')
```

(e) Nonlinear fitting: Exponential decay

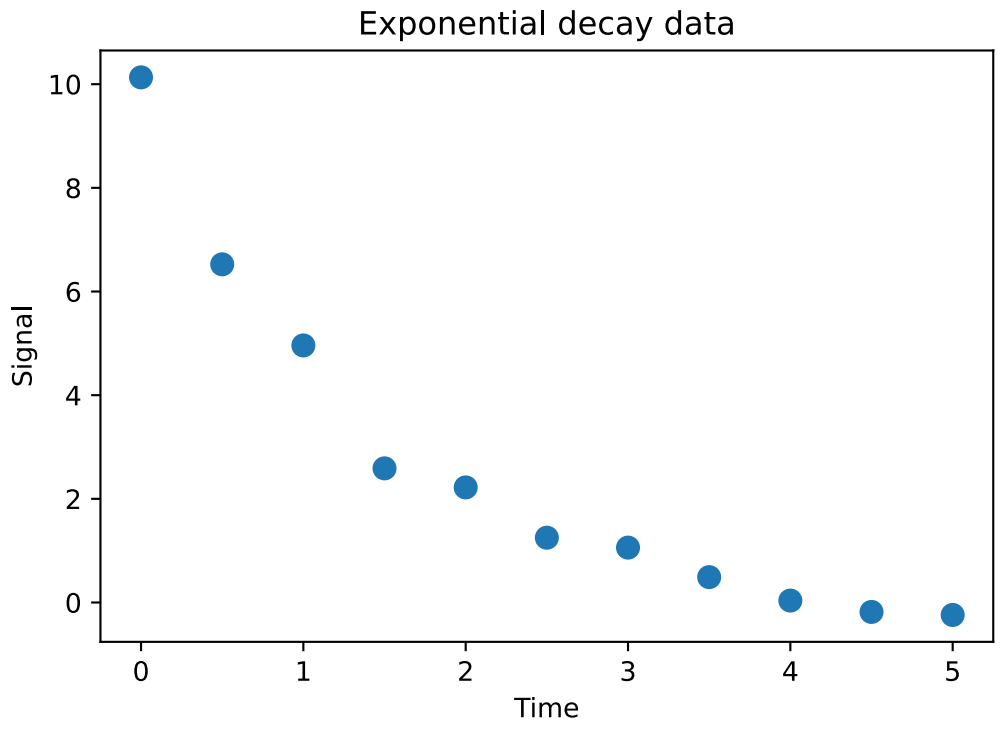
Many biological processes follow nonlinear relationships. Let's work with exponential decay, which is common in biochemistry (e.g., radioactive decay).

First, let's generate some exponential data:

```
# Generate exponential decay data
t = np.array([0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0])
# True parameters: A=10, k=0.8
A_true, k_true = 10, 0.8
signal = A_true * np.exp(-k_true * t) + np.random.normal(0, 0.3, len(t))

fig, ax = plt.subplots(figsize=(6, 4))
ax.plot(t, signal, 'o', markersize=8)
ax.set_xlabel('Time')
ax.set_ylabel('Signal')
ax.set_title('Exponential decay data')
```

```
Text(0.5, 1.0, 'Exponential decay data')
```



The model we want to fit is:

$$\text{signal} = Ae^{-kt}$$

Define the exponential decay function:

```
def exponential_decay(t, A, k):
    return ... # Complete this line
```

Now fit the exponential function to the data:

```
# Initial guess: A=8, k=1
p0 = [8, 1]

# Perform the fit
popt, pcov = curve_fit(..., ..., ..., p0=p0)

# Extract parameters
A_fit, k_fit = popt
print(f"Fitted parameters: A = {A_fit:.3f}, k = {k_fit:.3f}")
print(f"True parameters:   A = {A_true:.3f}, k = {k_true:.3f}")
```

Again we should plot to check that it looks as expected

```
t_smooth = np.linspace(0, 5, 100)
signal_fit = ... # Evaluate using the exponential_decay funciton
```

Key points for successful curve fitting:

1. **Always plot your data first** to understand what kind of function might fit
2. **Provide good initial guesses** (`p0`) - poor guesses can lead to fitting failure
3. **Use parameter bounds** when you know physical constraints
4. **Always plot the fit** to visually check if it makes sense
5. **Check if parameters are reasonable** based on your biochemical knowledge
6. **Use logarithmic plotting** for data spanning many orders of magnitude

Common pitfalls:

- **Overfitting**: Using too many parameters for the amount of data you have
- **Poor initial guesses**: Can cause the fit to fail or find a local minimum
- **Ignoring physical constraints**: Fitted parameters should make biological sense
- **Not checking the fit visually**: Always plot to see if the fit is reasonable

You now have the fundamental skills needed to fit curves to biochemical data! In the exercises, you'll apply these techniques to analyze real experimental data and extract meaningful biological parameters.

Estimation of binding affinity

(a) Train estimation skills

Train your estimation skills using the widget below.

```
from fysisk_biokemi.widgets.utils.colab import enable_custom_widget_colab
from fysisk_biokemi.widgets import estimate_kd

enable_custom_widget_colab()
estimate_kd()
```

(b) Compare to quadratic

The widget below shows the curves for θ using both the simple expression and the quadratic binding expression. Use it to determine how large [P] has to be for them to be notably different.

```
from fysisk_biokemi.widgets.utils.colab import enable_custom_widget_colab
from fysisk_biokemi.widgets import visualize_simple_vs_quadratic

enable_custom_widget_colab()
visualize_simple_vs_quadratic()
```

Dialysis experiment

```
import numpy as np
```

A dialysis experiment was set up where equal amounts of a protein were separately dialyzing against buffers containing different concentrations of a ligand – each measurement was done in triplicate. The average number of ligands bound per protein molecule, $\bar{n}$ were obtained from these experiments. The corresponding concentrations of free ligand and values are given in dataset `dialys-exper.xlsx`.

(a) Load the dataset

```
from fysisk_biokemi.widgets import DataUploader
from IPython.display import display
uploader = DataUploader()
uploader.display()
```

Run the next cell **after** uploading the file

```
df = uploader.get_dataframe()
display(df)
```

(b) Explain calculation of $\bar{n}$

Explain how the values of $\bar{n}$ is calculated when knowing the concentrations of ligand inside and outside the dialysis bag, as well as the total concentration of the protein, $[\mathrm{P}_{\mathrm{tot}}]$.

(c) Molar concentrations

Convert the concentrations of free ligand to SI-units given in M, add it as a row to the `DataFrame`.

(c) Plot the data

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

# The 'o' means we plot just the points and don't connect them.
ax.plot(..., ..., 'o') # Replace ... with your code.

ax.set_xlabel('Free ligand [L] (M)', fontsize=16)
ax.set_ylabel(r'$\bar{n}$', fontsize=16)
```

(d) Prepare for fitting

Now we want to fit the data to extract K_D and ν_{max}, by using the equation

$$\nu([L_{\mathrm{free}}]) = \nu_{\mathrm{max}} \frac{[L_{\mathrm{free}}]}{K_D + [L_{\mathrm{free}}]}$$

To do so we need to implmenet it as a Python function

```
def nu(L, nu_max, K_D):
    # Replace ... with your code that calculates the above equation.
    # Be careful with parenthesis!
    result = ...
    return result
```

```
print(f"{nu(1, 1, 1) = }") # Should give 1/2
print(f"{nu(21, 47, 2.5) = }") # Should give 42
```

(e) Actually fitting

> ! Important
>
> Fitting refers to finding the parameters that make an assumed functional form best 'fit' the data. Programmatically we will use the `curve_fit` from the `scipy` package to do so. The signature of this function looks like this
>
> ```
> curve_fit(function,
>           x_data,
>           y_data,
>           p0=[param_1, param_2, ...])
> ```
>
> The arguments are
>
> - `function`: A python function where the **first** argument is the independent variable, and other arguments are the parameters of the functions.
> - `x_data`: The observed values of the independent variable.
> - `y_data`: The observed values of the dependent variable.
> - `p0`: Initial guesses for the parameters.
>
> When called `curve_fit` starts by calculating how well the functions fits the data with the initial parameters in `p0` and then iteratively improves the fit by trying new values for the parameters in an intelligent way.
>
> The found parameters will generally depend on `p0` and it is therefore necessary to provide a good (or good enough) guess for `p0`.
>
> Generally, the best way to learn more about a function is to read it's documentation and then play around with it. The documentation is in this case on the SciPy website. You don't need to read it, unless you want more details.

Finish the code to perform the fitting in the cell below.

```
from scipy.optimize import curve_fit

# Choose the variables from the dataframe
x = df['Free Ligand [L](M)']
y = df['n-bar']

# Initial guess
K_D_guess = ... # Your initial guess for K_D
nu_max_guess = ... # Your initial guess for nu_max
p0 = [K_D_guess, nu_max_guess]

# Curve fit
```

```
# Replace the four ... with the correct arguments in the correct order.
popt, pcov = curve_fit(..., ..., ..., ...)

# Print the parameters
nu_max_fit, K_D_fit = popt
print(f"{nu_max_fit = :1.3f} ")
print(f"{K_D_fit = :e}")
```

Are the parameters you find reasonable? How can you tell if they are reasonable by looking at the plot you made earlier?

(f) Plot with fit

When we have the fitted parameters we can calculate and plot the function. To do so we make an array of values for the independent variable and use our function to calculate the dependent variable

```
# This makes 50 equally spaced points between 0 and the highest concentration x 1.2
L = np.linspace(0, x.max()*1.2, 50)

# Calculate by plugging L and the found parameters into the function.
nu_calc = ...
```

Now that we calculated the dependent variable we can plot the fit along with the data.

ADP binding to pyruvate kinase.

The binding of ADP to the enzyme pyruvate kinase was measured by fluorescence. The enzyme concentration was 4 µM throughout the titration, and each measurement was done in triplicate. The binding results were obtained at 310 K and are given in the `.csv`-file `adp-bindin-pyruva-kinase.csv`.

(a) Load the dataset

As always, use the widget to load the dataset

```
from fysisk_biokemi.widgets import DataUploader
from IPython.display import display
uploader = DataUploader()
uploader.display()
```

Run the next cell **after** uploading the file

```
df = uploader.get_dataframe()
display(df)
```

(b) Units

The concentrations in the dataset are given in mM, add a new column to the `DataFrame` with the units given in M.

```
df['[ADPtot](M)'] = ...
display(df)
```

(c) Free ADP concentration

For each value of $\bar{n}$ calculate the concentration of $[\text{ADP}_{\text{free}}]$ from $[\text{ADP}_{\text{tot}}]$ and [enzyme].

```
enzyme_conc = ...
df['[ADPfree](M)'] = ...
display(df)
```

(d) Make a plot

Make a plot of the free ligand concentration versus $\bar{n}$.

```
fig, ax = plt.subplots(figsize=(7, 4))

# Write the code to make the plot.
# Add the arguments:  marker='o', linestyle='none' to only plot points and not line
segments.
... # Add your code here!

ax.set_xlabel(r'$[\text{ADP}_{\text{free}}]$', fontsize=14)
ax.set_ylabel(r'$\bar{n}$', fontsize=14)
```

(f) Preparing for fitting

To fit we need a implement the function we want to fit the parameters of, the functional form is

$$n = n_{\max} \frac{[L]^n}{K_D + [L]^n}$$

```
def n_bound(L, n_max, K_D, n_exp):
    return ...

print(f"{n_bound(1, 1, 1, 1) = }") # Should give 1/2
print(f"{n_bound(21, 47, 2.5, 1) = }") # Should give 42
print(f"{n_bound(21, 47, 2.5, 2) = }") # Should give 46.73..
```

(e) Fitting

Finish the code below to create a fit.

```
from scipy.optimize import curve_fit

# Choose the variables from the dataframe
x = df['[ADPfree](M)']
y = df['nbar']

# Initial guess
K_D_guess = ... # Your initial guess for K_D
```

```
nu_max_guess = ... # Your initial guess for nu_max
n_exp = ... # Your initial guess for the exponent.
p0 = [K_D_guess, nu_max_guess, n_exp]

# Bounds
bounds = (0, np.inf) # We limit the parameters to be positve.

# Curve fit
popt, pcov = curve_fit(n_bound, x, y, p0=p0, bounds=bounds)

# Print the parameters
n_max_fit, K_D_fit, n_exp_fit = popt
print(f"{n_max_fit = :1.3f} ")
print(f"{K_D_fit = :e}")
print(f"{n_exp_fit = :1.3f} ")
```

Once we've obtained the fitted parameters we can plot the fit together with the data.

```
L = np.linspace(0, 1.2 * x.max(), 50)
n = n_bound(L, n_max_fit, K_D_fit, 1)

fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(df['[ADPfree](M)'], df['nbar'], 'o', label='Observations')
ax.plot(L, n)
ax.axvline(K_D_fit, color='k', linewidth=0.5, linestyle='--')
ax.axhline(n_max_fit, color='k', linewidth=0.5, linestyle='--')


ax.set_xlabel(r'$[\text{ADP}_{\text{free}}]$', fontsize=14)
ax.set_ylabel(r'$\bar{n}$', fontsize=14)
ax.set_ylim([0, n.max()*1.1])
```

Free energy

Calculate the free energy for the association of the ADP-pyruvate kinase complex assuming $R = 8.314472 \times 10^{-3}\ \frac{\text{kJ}}{\text{mol}\cdot\text{K}}$ and $T = 310\ \text{K}$.

> **Tip**
>
> Consider the difference between association and dissociation

Start by defining the two given constants as variables

```
R = ...
T = ...
```

And do the calculation

```
delta_G = ...
print(f"{delta_G = :.3f} kJ/mol")
```

Interpretation of binding data.

```
import numpy as np
```

The `inter-bindin-data.xlsx` contains a protein binding dataset.

(a) Load the dataset

Load the dataset using the widget below

```
import numpy as np
from fysisk_biokemi.widgets import DataUploader
from IPython.display import display
uploader = DataUploader()
uploader.display()
```

Run the next cell **after** uploading the file

```
df = uploader.get_dataframe()
display(df)
```

(b) SI Units

Add a new column to the `DataFrame` with the ligand concentration in SI units.

```
... # Replace ... with your code.
display(df)
```

(c) Plot the data

Make plots of the binding data directly with a linear and logarithmic x-axis.

Estimate K_D by visual inspection of these plots!

```
import matplotlib.pyplot as plt

# This makes a figure with two axes.
fig, axes = plt.subplots(1, 2, figsize=(9, 4))

# Can with [0] to plot in the first axis.
ax = axes[0]
ax.plot(..., ..., 'o') # Replace ... with your code.
ax.set_xlabel('[L](M)', fontsize=14)
ax.set_ylabel(r'$\bar{n}$', fontsize=14)

ax = axes[1]
```

```
... # Put some code here - perhaps you can copy it from somewhere?
ax.set_xscale('log') # This make the x-axis logarithmic.
```

> **Note**
>
> Ths command `ax.set_xscale('log')` tells `matplotlib` that we want the x-axis to use a log-scale.

```
k_d_estimate = ...
```

(d) Make a fit

Make a fit to determine K_D, as always we start by implementing the function to fit with

```
def ... # Give the function an appropriate name.
    return ... # Implement the expression for nbar
```

And then we can make the fit

```
from scipy.optimize import curve_fit

# Choose the variables from the dataframe
x = ... # Choose x-data from the dataframe
y = ... # Choose y-data from the dataframe

# Initial guess
p0 = [k_d_estimate]

# Bounds
bounds = (0, np.inf) # We limit the parameters to be positve.

# Curve fit
popt, pcov = ... # Call the curve_fit function.

# Print the parameters
k_d_fit = popt[0]
print(f"{k_d_fit = :e}")
```

Compare

Use the figure below to compare your guess with the fitted value.

Saturation

Based on the value of K_D found from the fit,

- At which concentration do you expect 10% saturation?
- At which concentration do you expect 90% saturation?

Determination of type and strength of cooperativity

```
import matplotlib.pyplot as plt
import numpy as np
```

The binding of NAD+ to the protein yeast glyceraldehyde 3-phosphate dehydrogenase (GAPDH) was studied using equilibrium dialysis. The enzyme concentration was 71 μM. The concentration of $[\mathrm{NAD}^+_{\mathrm{free}}]$ and the corresponding values of $\bar{n}$ were determined with the resulting data found in the dataset `deter-type-streng-coope.xlsx`.

(a) Load the dataset

Load the dataset using the widget below

```
from fysisk_biokemi.widgets import DataUploader
from IPython.display import display
uploader = DataUploader()
uploader.display()
```

Run the next cell **after** uploading the file

```
df = uploader.get_dataframe()
display(df)
```

(b) Averaging and units.

Start by adding a new column to the `DataFrame` with the average value of $\bar{n}$ across the three series

> Tip
>
> Remember that you can set a new column based on a computation using one or more other columns, e.g.
>
> ```
> df['new_col'] = df['col1'] + df['col2']
> ```

```
df['nbar_avg'] = ...
```

Now also add a column with the ligand concentration in SI units with the column-name `[NAD+free]_(M)`.

```
... # Your code here.
display(df)
```

Finally, set the concentration of the GADPH in SI units

```
c_gadph = 71 * 10**(-6)
```

(c) Plot

Make a plot of the average $\bar{n}$ as a function of $[\mathrm{NAD}^+_{\mathrm{free}}]$.

```
fig, ax = plt.subplots(figsize=(8, 4))

# Your code to plot here.
...

# This sets the labels.
ax.set_xlabel(r'$[\text{NAD}^{+}_\text{free}]$', fontsize=14)
ax.set_ylabel(r'$\bar{n}$', fontsize=14)
plt.show()
```

(d) Scatchard plot

Make a Scatchard plot based on the average $\bar{n}$.

```
# Calculate nbar / L
nbar_over_L = ...

fig, ax = plt.subplots(figsize=(6, 4))

ax.plot(..., ... 'o') # Plot the right thins.

ax.set_xlabel(r'$\bar{n}$', fontsize=14)
ax.set_ylabel(r'$\bar{n}/L$', fontsize=14)
ax.set_xlim([0, df['nbar_avg'].max()*1.2])
ax.set_ylim([0, nbar_over_L.max() * 1.2])
```

(e) Binding sites

How many binding sites does GAPDH contain for NAD^+?

(f) Cooperativity

What type of cooperativity do the plots indicate?

(g) Fit

Make a fit using the functional form

$$\bar{n} = N\frac{[L]^h}{K_D + [L]^h}$$

As usual, start by defining the function in Python

```
def n_bar(L, N, k_d, h):
    # Replace ... with your code.
    # Be careful with parentheses.
    result = ...
    return result
```

Now we can fit

```
from scipy.optimize import curve_fit
```

```
# This selects the '[NAD+free]_(m)'-column three times and stitches it together.
x = np.concatenate([df['[NAD+free]_(M)'], df['[NAD+free]_(M)'], df['[NAD+free]_(M)']])
# Do the same to stitch together the nbar1, nbar2 and nbar3 columns.
y = ...

# Initial guess
p0 = [..., ..., ...]

# Bounds
bounds = (0, np.inf) # We limit the parameters to be positve.

# Curve fit
popt, pcov = curve_fit(n_bar, x, y, p0=p0, bounds=bounds)

# Print the parameters
N_fit, k_d_fit, h_fit = popt
print(f"{N_fit = :.3f}")
print(f"{k_d_fit = :e}")
print(f"{h_fit = :.3f}")
```

Note

The function `np.concatenate` takes a number of arrays and makes a new array that consisting of the originals one after each other. For example,

```
A = np.array([1, 2, 3])
B = np.array([4, 5, 6])
C = np.concatenate([A, B])
print(C)
```

```
[1 2 3 4 5 6]
```

(h) Plot with fit

```
L = np.linspace(0, df['[NAD+free]_(M)'].max()*1.5)
n_bar_fit = n_bar(L, N_fit, k_d_fit, h_fit)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(x, y, 'o', label='Data')
ax.plot(L, n_bar_fit, '-', label='Fit')
ax.axhline(N_fit, color='C2', label='N from fit')
ax.axvline(k_d_fit, color='C3', label=r'$K_D$ from fit')
ax.set_xlabel(r'$[\text{NAD}^{+}_\text{free}]$', fontsize=14)
ax.set_ylabel(r'$\bar{n}$', fontsize=14)
ax.legend()
plt.show()
```

Competition in ligand binding

Below is given the general expression for saturation of a binding site by one of the ligands, L, when two ligands L and C are competing for binding to the same site on a protein. Assume that $[P_T] = 10^{-9}$ M.

$$\theta = \frac{[PL]}{[P_T]} = \frac{1}{\frac{K_D}{[L]}\left(1 + \frac{[C]}{K_C}\right) + 1}$$

Consider these four situations

#	K_D	$[L_T]$	K_C	$[C_T]$
1	$1 \cdot 10^{-5}$ M	$1 \cdot 10^{-3}$ M		0
2	$1 \cdot 10^{-5}$ M	$1 \cdot 10^{-3}$ M	$1 \cdot 10^{-6}$ M	$1 \cdot 10^{-2}$ M
3	$1 \cdot 10^{-5}$ M	$1 \cdot 10^{-3}$ M	$1 \cdot 10^{-5}$ M	$1 \cdot 10^{-3}$ M
4	$1 \cdot 10^{-5}$ M	$1 \cdot 10^{-5}$ M	$1 \cdot 10^{-6}$ M	$1 \cdot 10^{-6}$ M

(a) Explain simplification

Explain how the fact that $[P_T]$ is much smaller than $[L_T]$ and $[C_T]$ simplifies the calculations using the above equation.

(b) Ligand degree of saturation

Calculate the degree of saturation of the protein with ligand L in the four situations.

Start by writing a Python function that calculates the degree of saturation θ.

```
def bound_fraction(L, C, Kd, Kc):
    # Write the equation for binding saturation. Be careful with parentheses!
    theta = ...
    return theta
```

Then use that function to calculate θ for each situation.

```
theta_L_1 = bound_fraction(..., ..., ..., ...)
theta_L_2 = ...
theta_L_3 = ...
theta_L_4 = ...
print(f"{theta_L_1 = :.3f}")
print(f"{theta_L_2 = :.3f}")
print(f"{theta_L_3 = :.3f}")
print(f"{theta_L_4 = :.3f}")
```

(c) Competitor degree of saturation

What is the degree of saturation with respect to the competitor C in #1, #2 and #4?

```
theta_C_1 = ...
theta_C_2 = ...
theta_C_3 = ...
theta_C_4 = ...

print(f"{theta_C_1 = :.3f}")
print(f"{theta_C_2 = :.3f}")
print(f"{theta_C_3 = :.3f}")
print(f"{theta_C_4 = :.3f}")
```

(d) Fraction of $[P_{\text{free}}]$

What is the fraction of $[P_{\text{free}}]$ in #2, #3, #4?

> Tip
>
> Consider how to express the fraction of $[P_{\text{free}}]$ in terms of `theta_L_X` and `theta_C_X`.

```
theta_free_1 = ...
theta_free_2 = ...
theta_free_3 = ...
theta_free_4 = ...

print(f"{theta_free_1 = :.3f}")
print(f"{theta_free_2 = :.3f}")
print(f"{theta_free_3 = :.3f}")
print(f"{theta_free_4 = :.3f}")
```