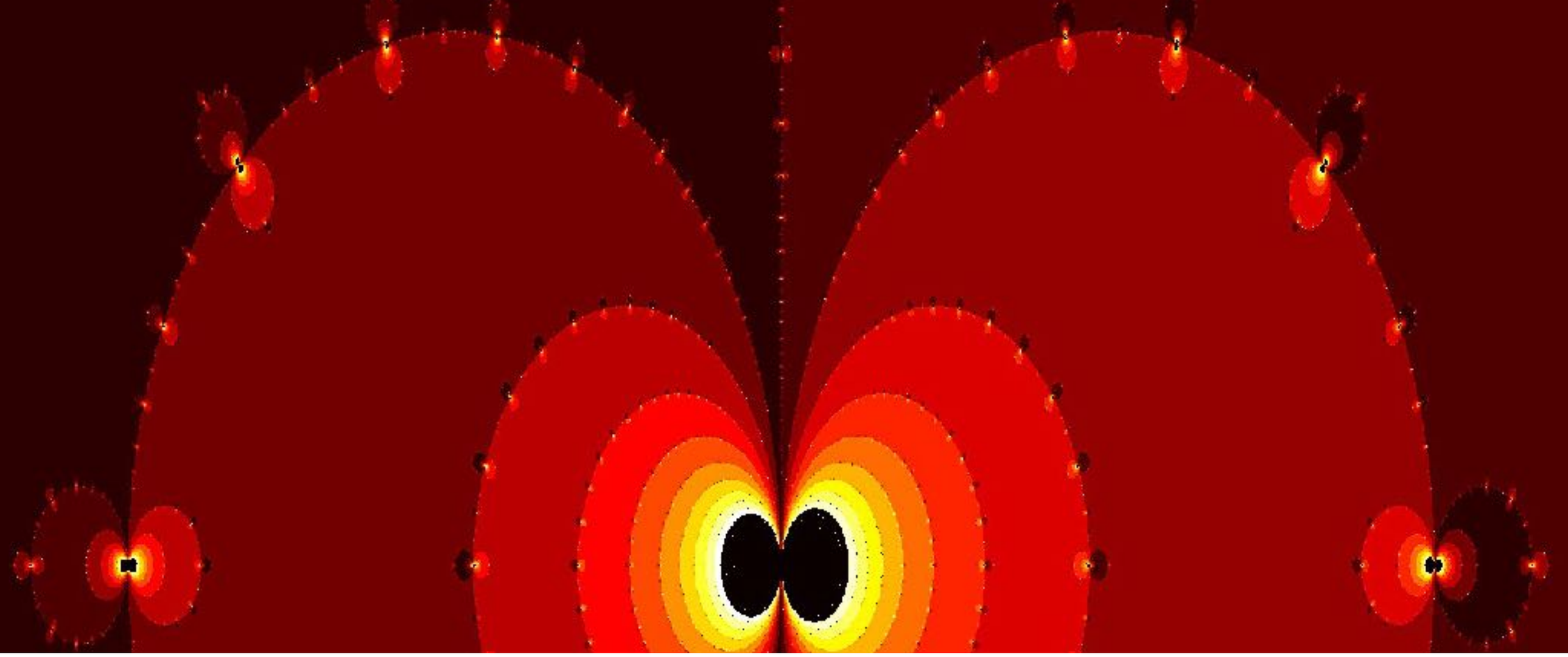


NUMERICAL METHODS IN LINEAR ALGEBRA

A. Krassnigg, Uni Graz, WS 2019

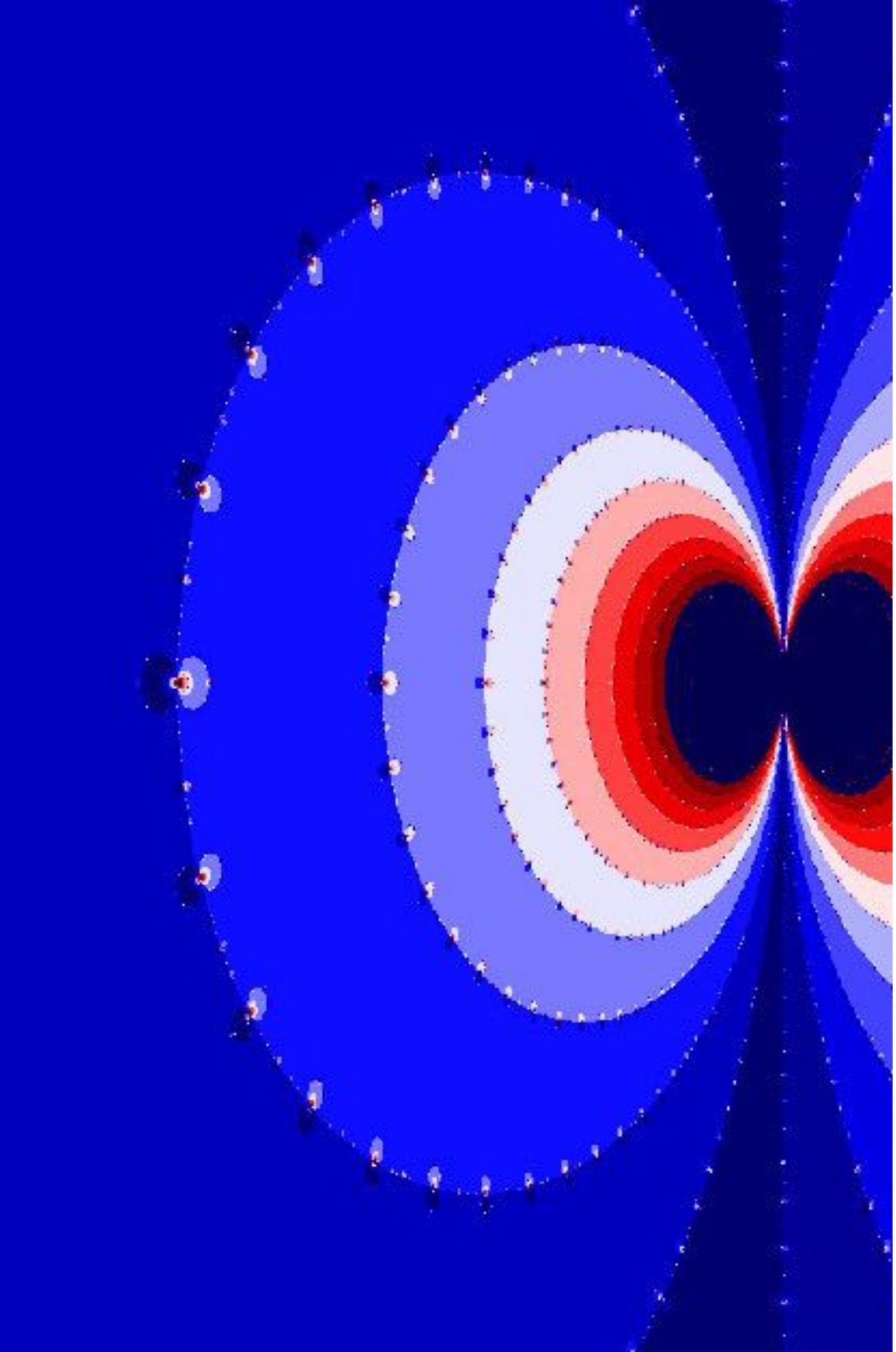
TABLE OF CONTENTS

LECTURE OVERVIEW

- What to expect
  - Language of notes/presentations will be English. In the lecture room: English or German!
  - working together towards a thorough understanding of numerical methods in linear algebra
  - first time for me (this particular lecture), so let's work together to reach a better understanding of the material
  - methods before physics (sounds weird, I know)
  - any questions? Ask! (during the lecture, per email, …)

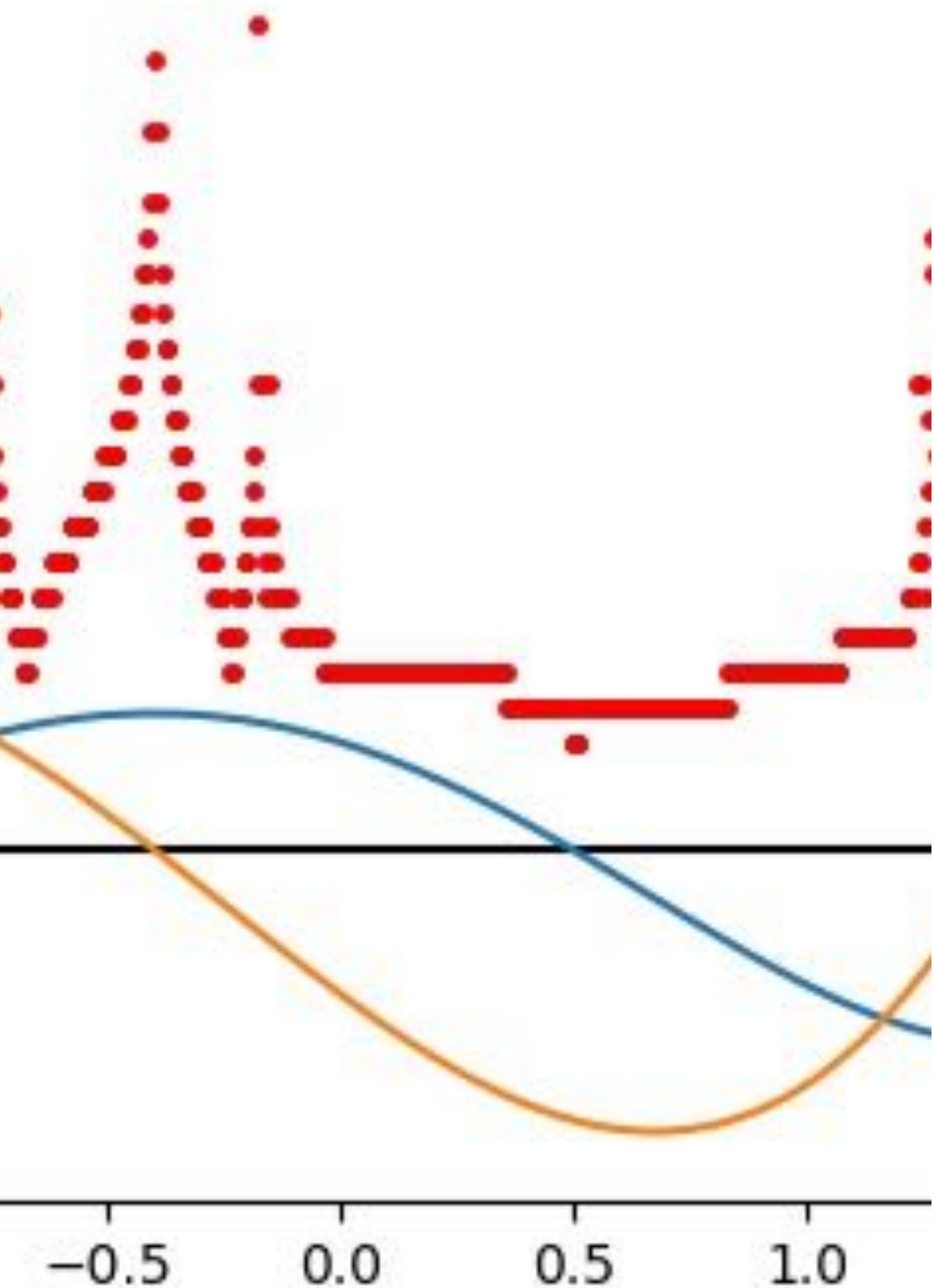
LECTURE OVERVIEW

- How do we work the best?
  - I find the lecture room adequate, in particular with laptops
  - Does everybody have a laptop?
  - The VU should have a practical component (each week)
    - Use a programming language of your choice (each)!
- Scoring?
  - I'd like to do oral exams at the end.
  - But before that, we'll have practical exercise parts each week
  - We'll also have a round of projects and presentations in January



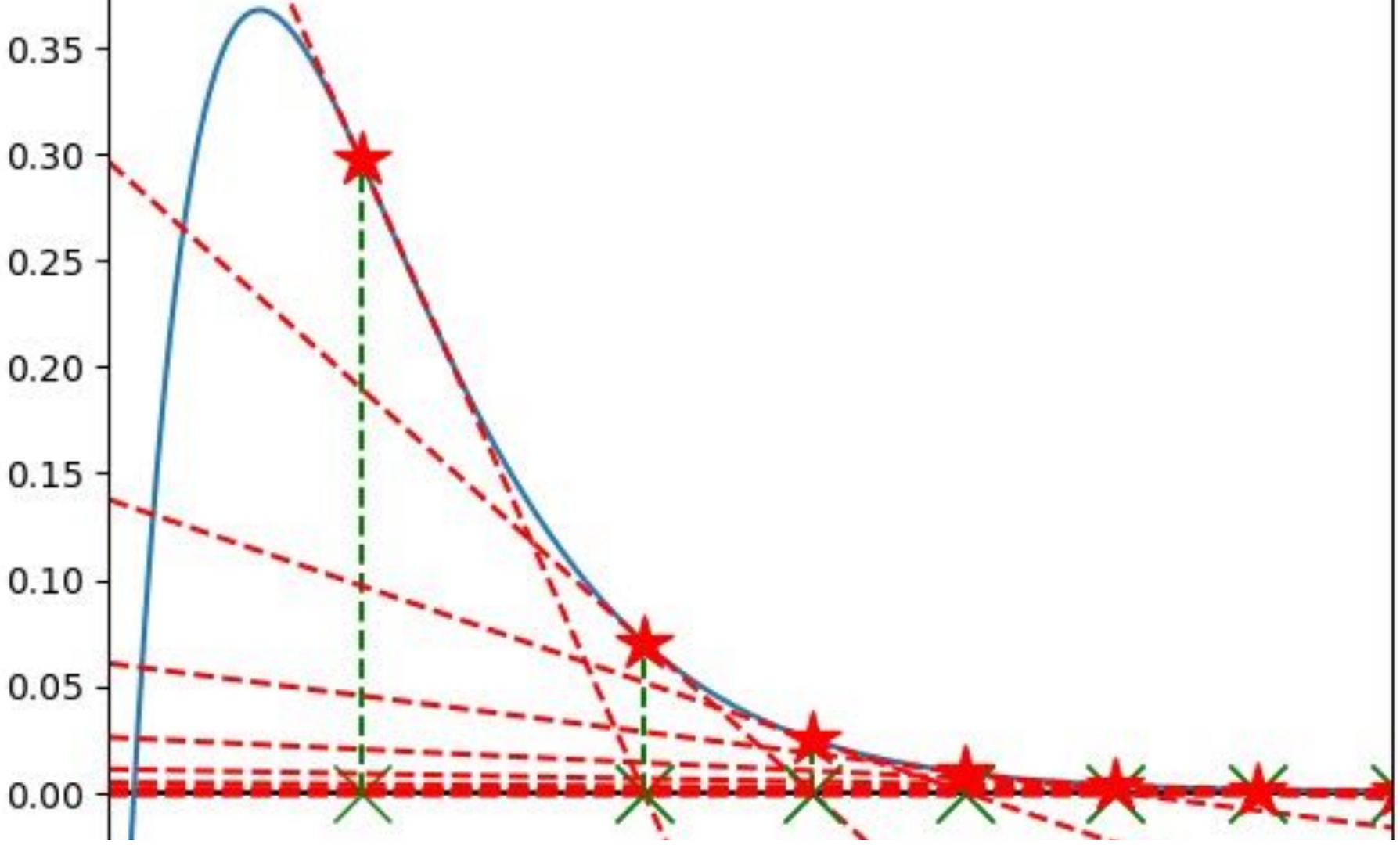
CONTENTS OF THIS SEMESTER

- Get an overview of numerically interesting problems in linear algebra
- Solving a system of linear algebraic equations
- Computing the eigensystem of a matrix
- Eigenvalues and eigenvectors
- Identifying interesting problems to solve
- In physics and outside!
- Numerical aspects!



INTRODUCTION

- Where are NMLA used?
- Everywhere!
- Characteristics:
  - direct
  - iterative
  - stability
  - efficiency
  - applicability
  - speed
  - complexity



DIVING IN, BUT NOT QUITE

A numerical method to begin with

$$f(x) = 0$$

NEWTON'S METHOD

- Used to find numerical approximations of roots of functions

NEWTON'S METHOD

$$x = x_0$$

- Used to find numerical approximations of roots of functions
- Start at a particular initial guess

$$f(x_0)$$

$$f'(x_0)$$

$$y(x) = kx + d$$

$$k = f'(x_0)$$

$$d = f(x_0) - f'(x_0)x_0$$

NEWTON'S METHOD

- Used to find numerical approximations of roots of functions
- Start at a particular initial guess
- Construct a tangent to the function at this point

NEWTON'S METHOD

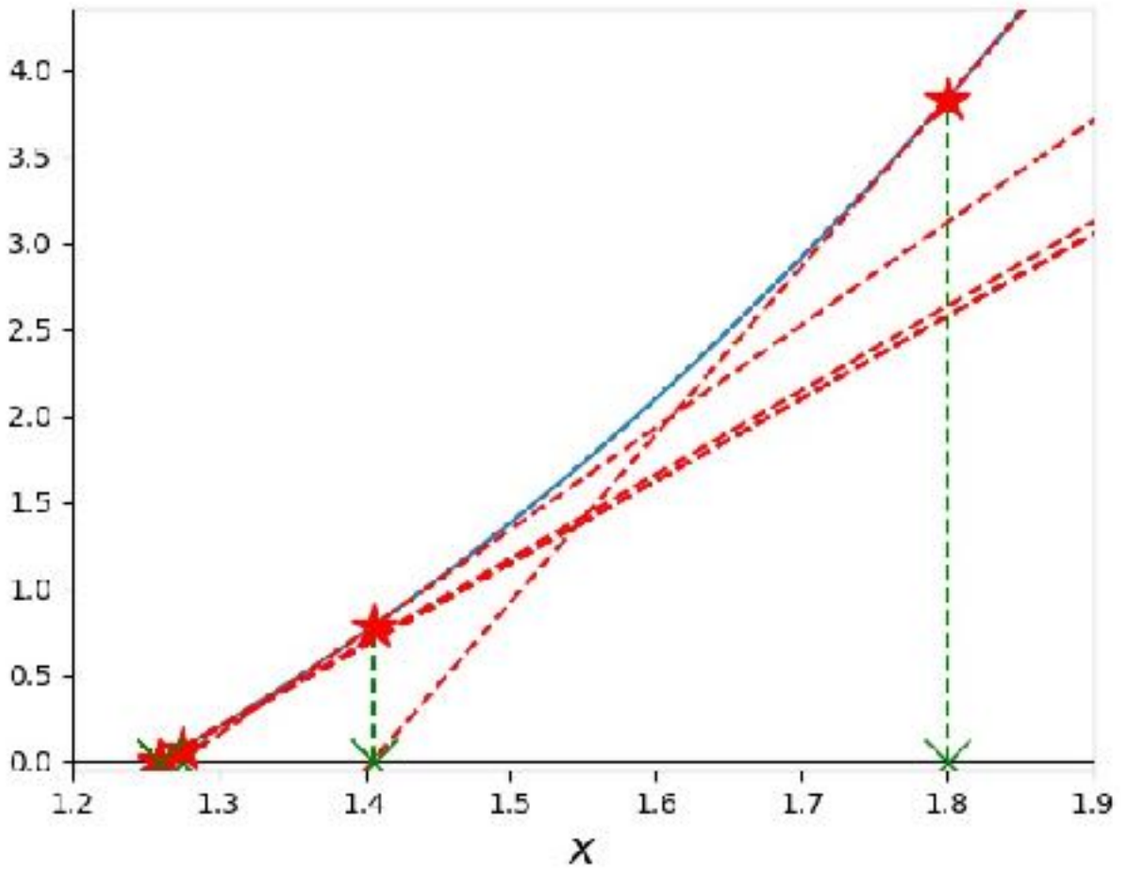
- Used to find numerical approximations of roots of functions
- Start at a particular initial guess
- Construct a tangent to the function at this point
- Intersect the tangent with x-axis and find next guess for the solution

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

NEWTON'S METHOD

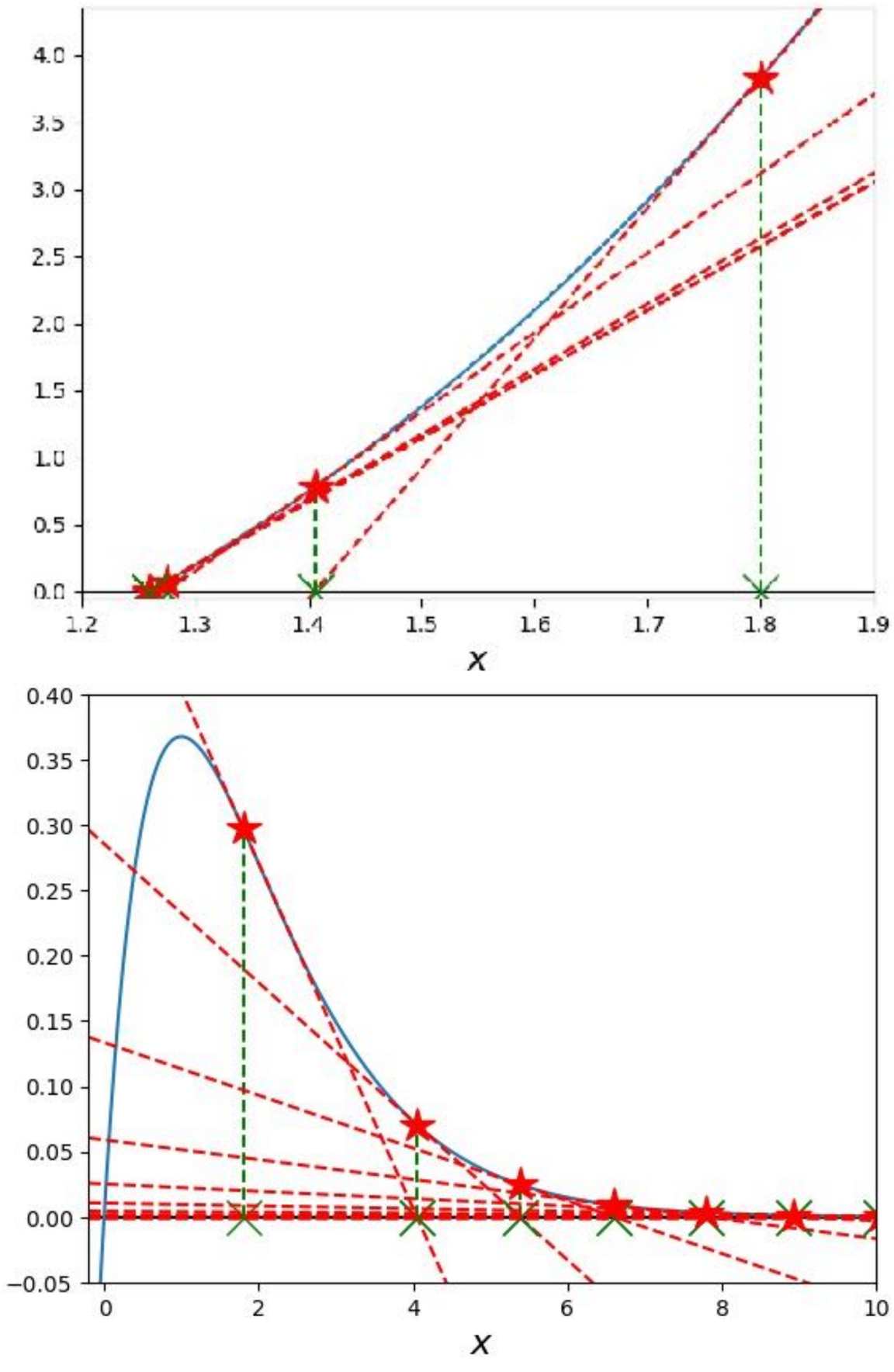
- Used to find numerical approximations of roots of functions
- Start at a particular initial guess
- Construct a tangent to the function at this point
- Intersect the tangent with x-axis and find next guess for the solution
- Repeat and wait for convergence

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$



NEWTON'S METHOD

- Used to find numerical approximations of roots of functions
- Start at a particular initial guess
- Construct a tangent to the function at this point
- Intersect the tangent with x-axis and find next guess for the solution
- Repeat and wait for convergence



NEWTON'S METHOD

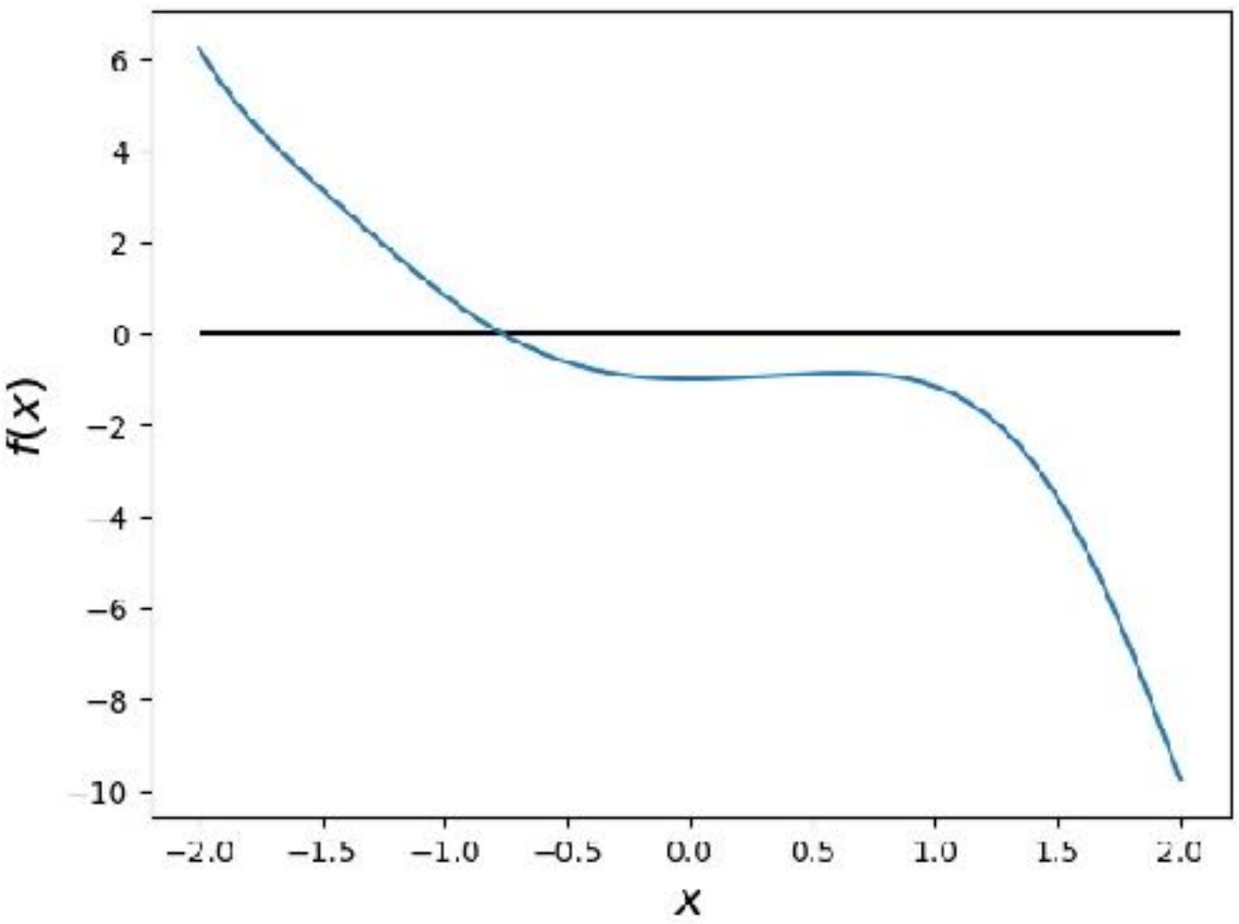
- Used to find numerical approximations of roots of functions
- Start at a particular initial guess
- Construct a tangent to the function at this point
- Intersect the tangent with x-axis and find next guess for the solution
- Repeat and wait for convergence

EXERCISE 1

- Implement Newton's method
- Find a numerical approximation to the root of a given function
- Set an initial guess/starting point
- require a relative precision of 10^{-10}
- set a maximum number of iteration steps (as an important precaution)
- produce a table of the x_i
- if you like and have the time, produce some sort of graph

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

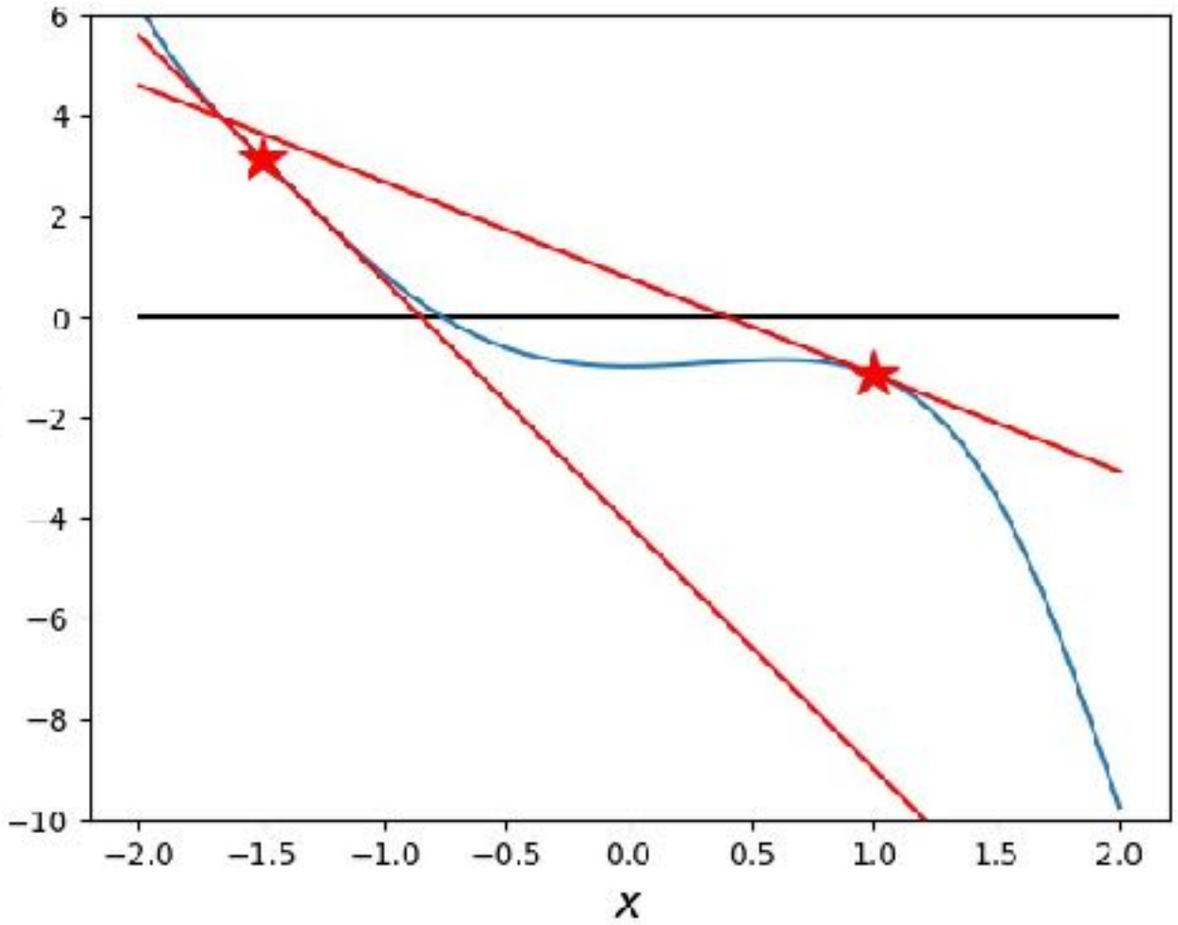
EXERCISE 1: RESULTS



- Check out what an example function looks like
- Plot the function
- Check for zeros/roots

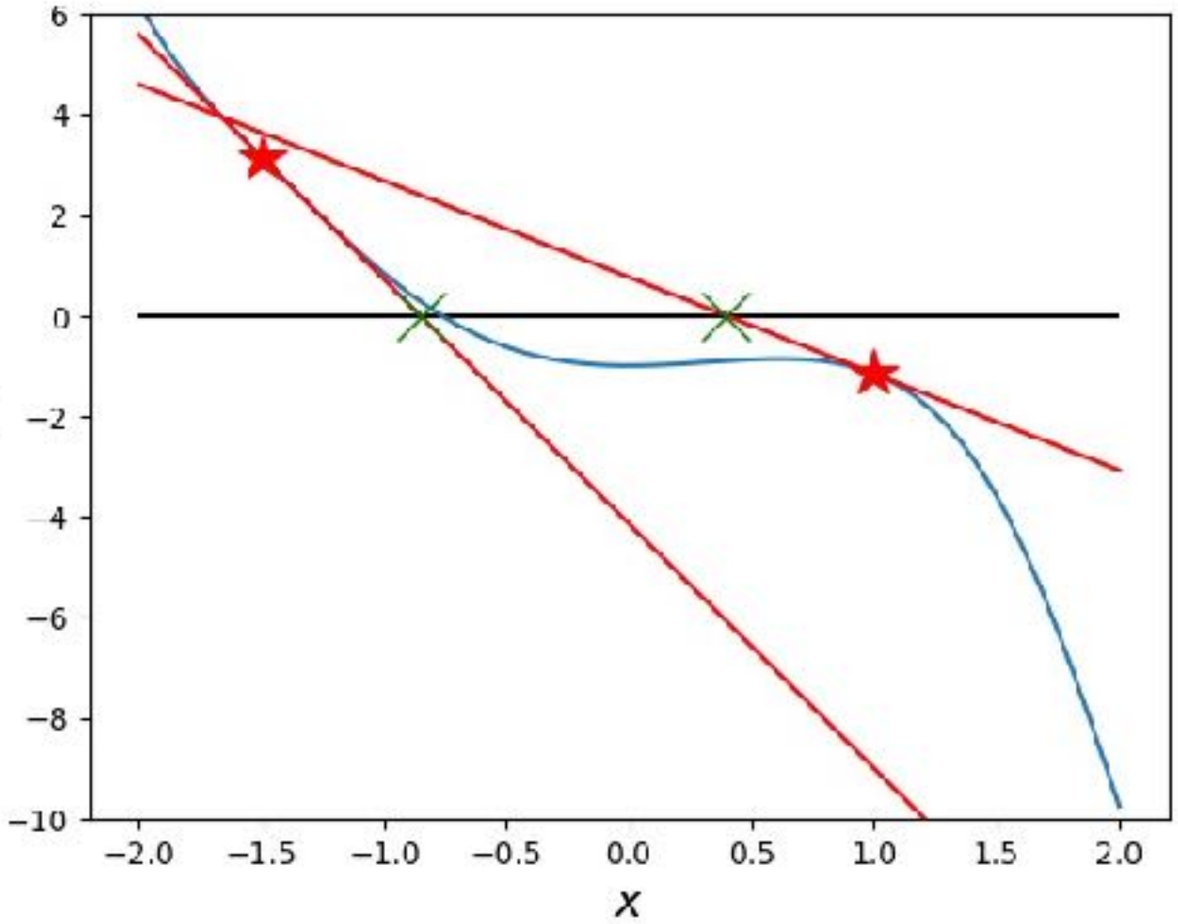
- Caveat:
  - It may not be cheap to plot the function
  - but even then, each evaluation of the function during Newton's method yields one point to plot

EXERCISE 1: RESULTS



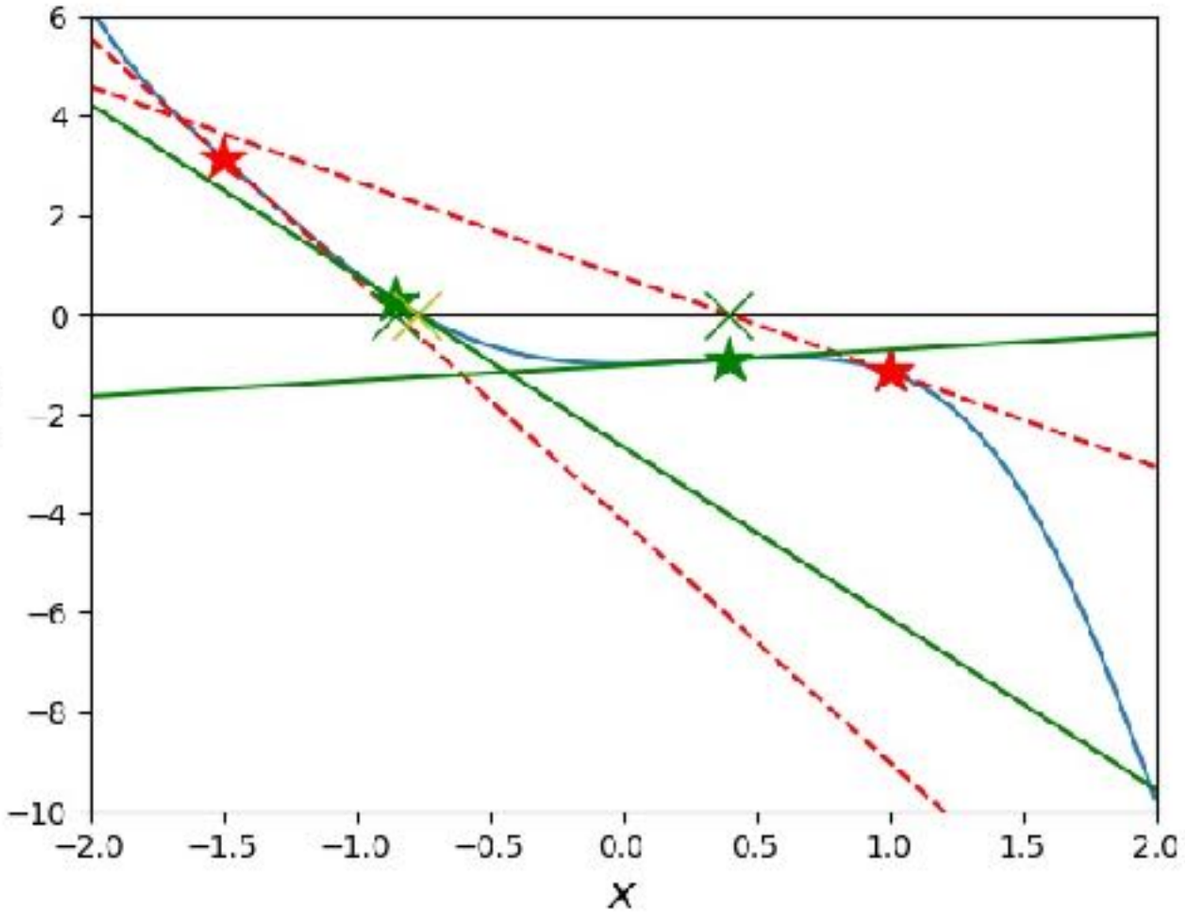
- Next step: pick a starting point and create a tangent to the function at this point.
- Here are two examples
- Each starting point on the curve is marked by a red star
- Tangent plotted as red line

EXERCISE 1: RESULTS

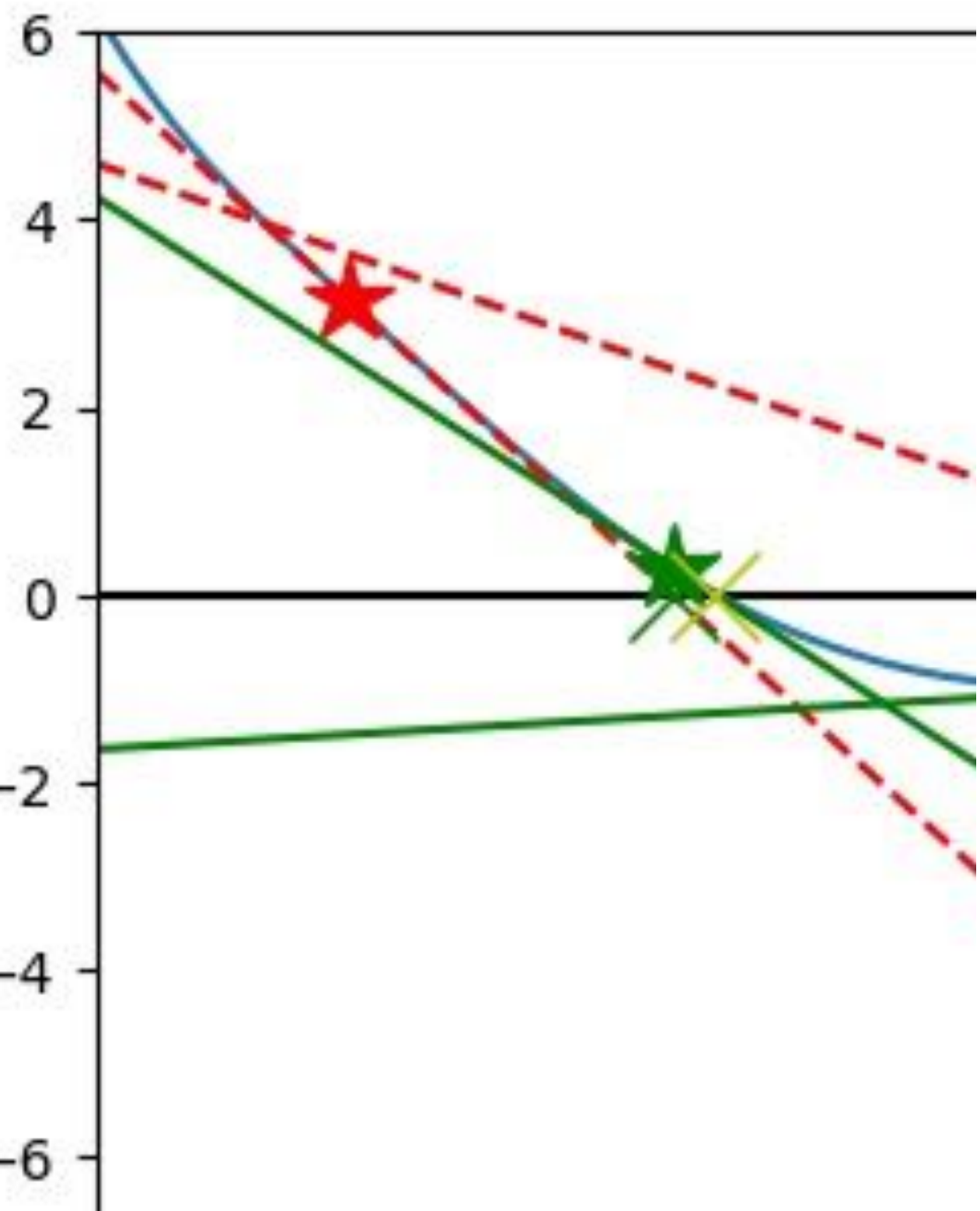


- Next step:
- Notice where these two example tangents intersect the zero line
- Differences between starting points!
- Intersections marked with green Xs
- Comparisons to the actual root are interesting, but one iteration step is nothing compared to many. Why?

EXERCISE 1: RESULTS

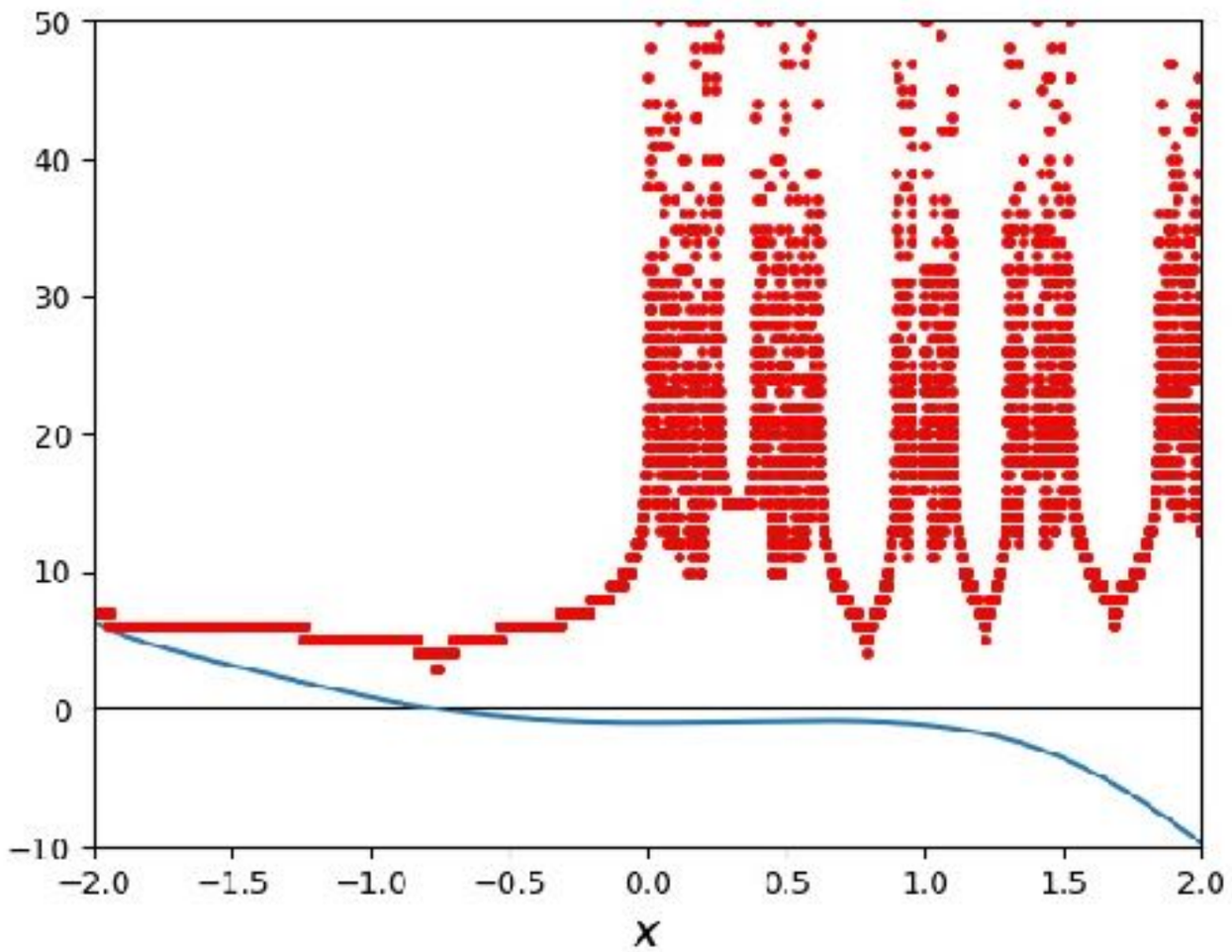


- Next step:
- Use intersection of tangent as the new starting point for the next step
- Mark new starting point on the curve by a green star
- Also draw the tangents to those green-star points
- Whoa, what a difference with regard to where these intersect with zero, depending on the first choice of the starting point



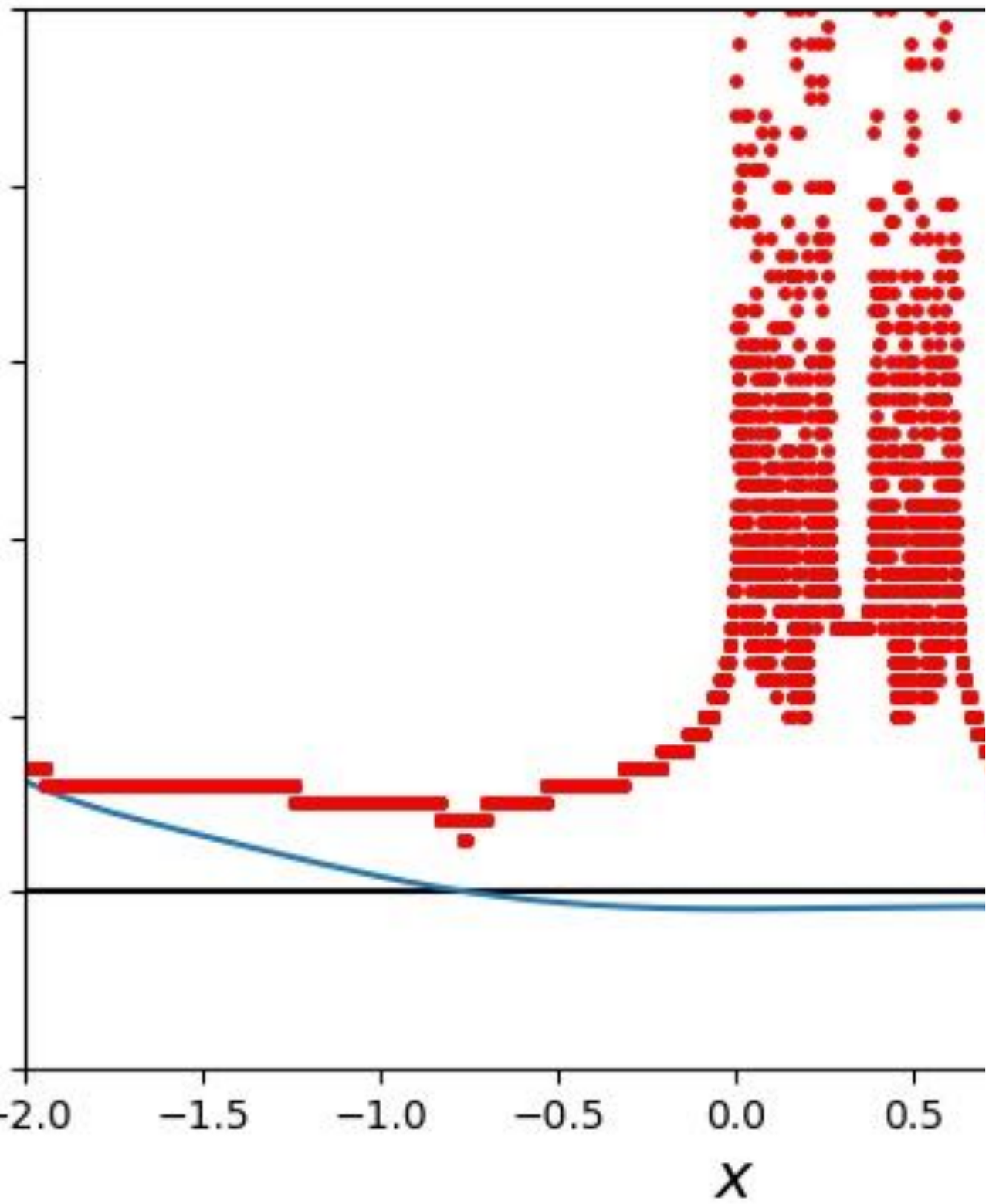
EXERCISE 1: RESULTS

- Whoa, what a difference with regard to where these intersect with zero, depending on the first choice of the starting point
- Close inspection:
- First choice is almost at the actual root (yellow X)
- Second choice has moved out (away from the root) substantially!



EXERCISE 1: RESULTS

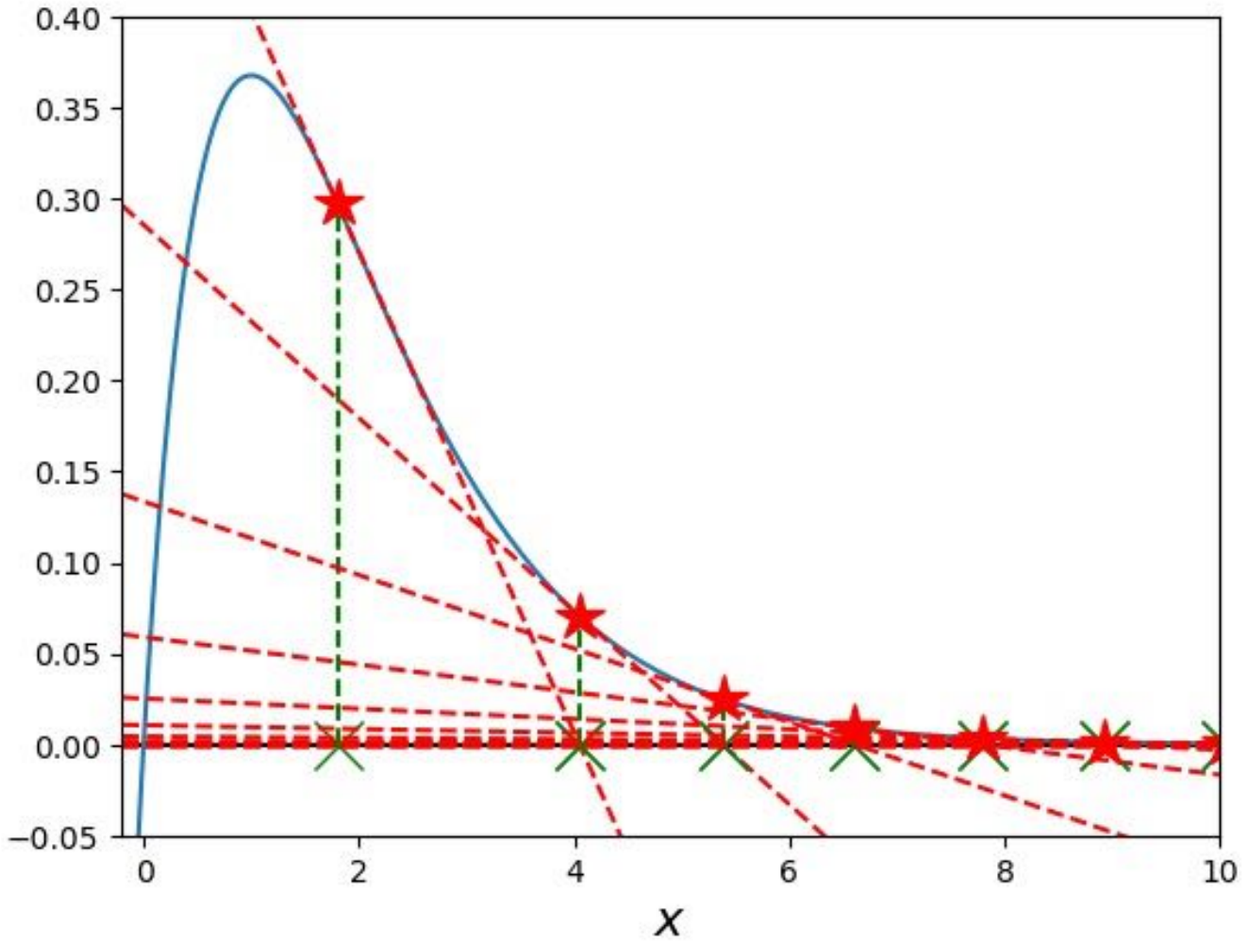
- Can we quantify this dependence somehow?
- How many iteration steps does it take to get to the root with a certain precision?
- Let’s plot the number of necessary iterations as a result of the starting point along the x-Axis
- Red dots mark number of necessary iterations
- “Towers” mark regions of instability
- Relates well to the slope of the function



EXERCISE 1: BEWARE!

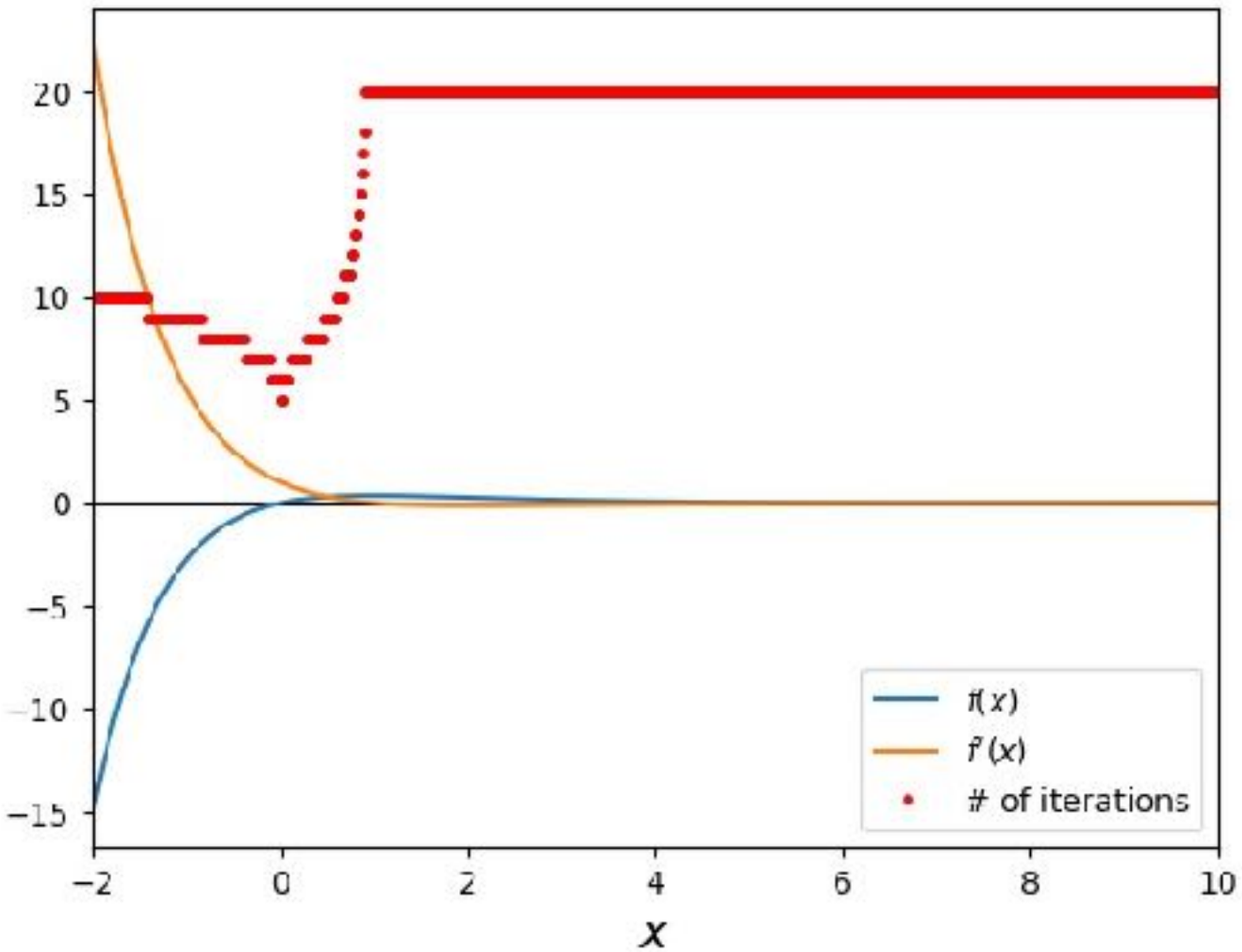
- Beware of the following:
- Points where the slope of the function vanishes (i.e., its first derivative is zero)
- The first two “Towers” (from left to right) of red dots in this plot are generated around the two zeros of the first derivative of our example

EXERCISE 1: BEWARE!



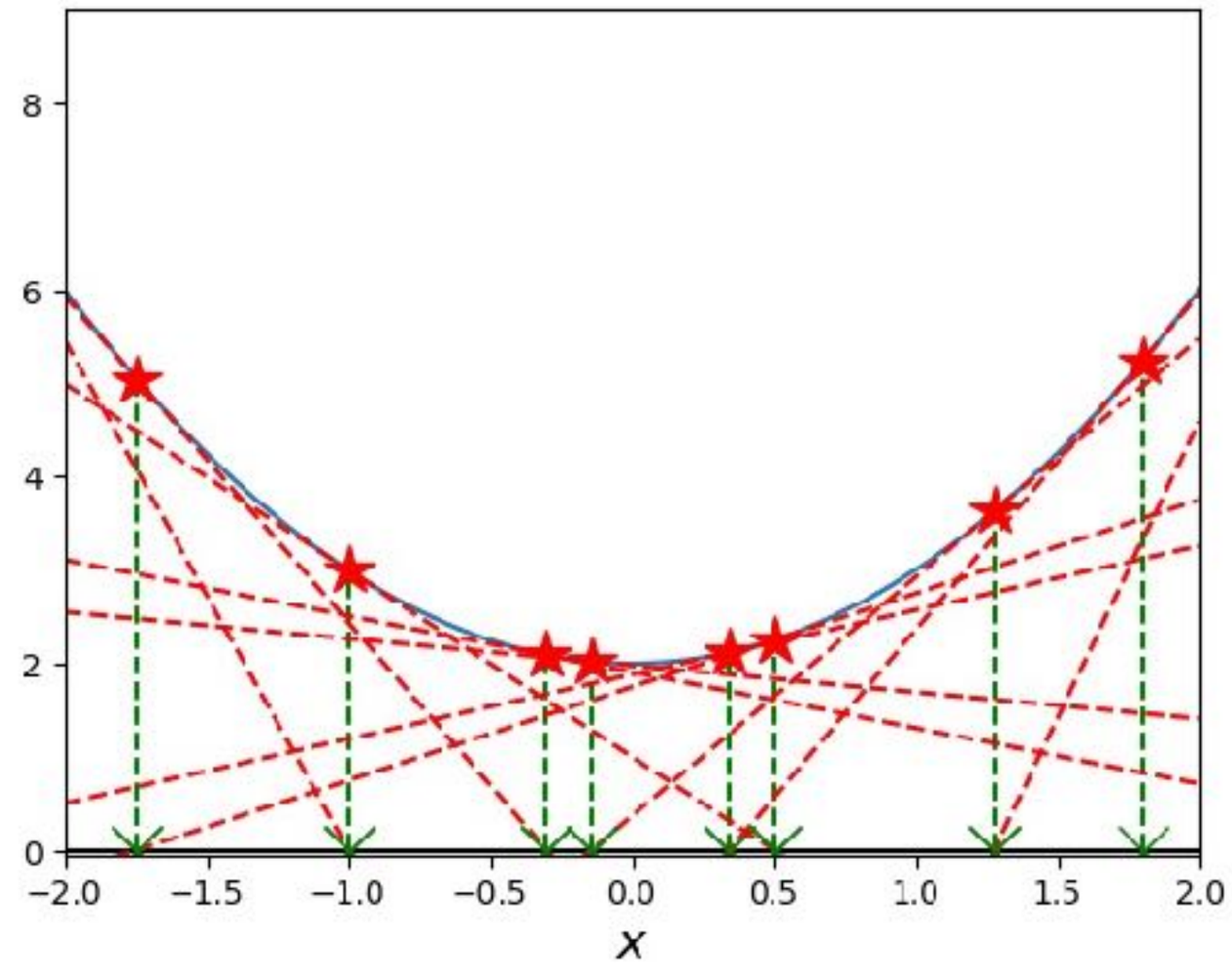
- Beware of the following:
- Regions in the figure, where function and slope fall off monotonously, but don't reach zero
- Example:

$$f(x) = x \exp(-x)$$



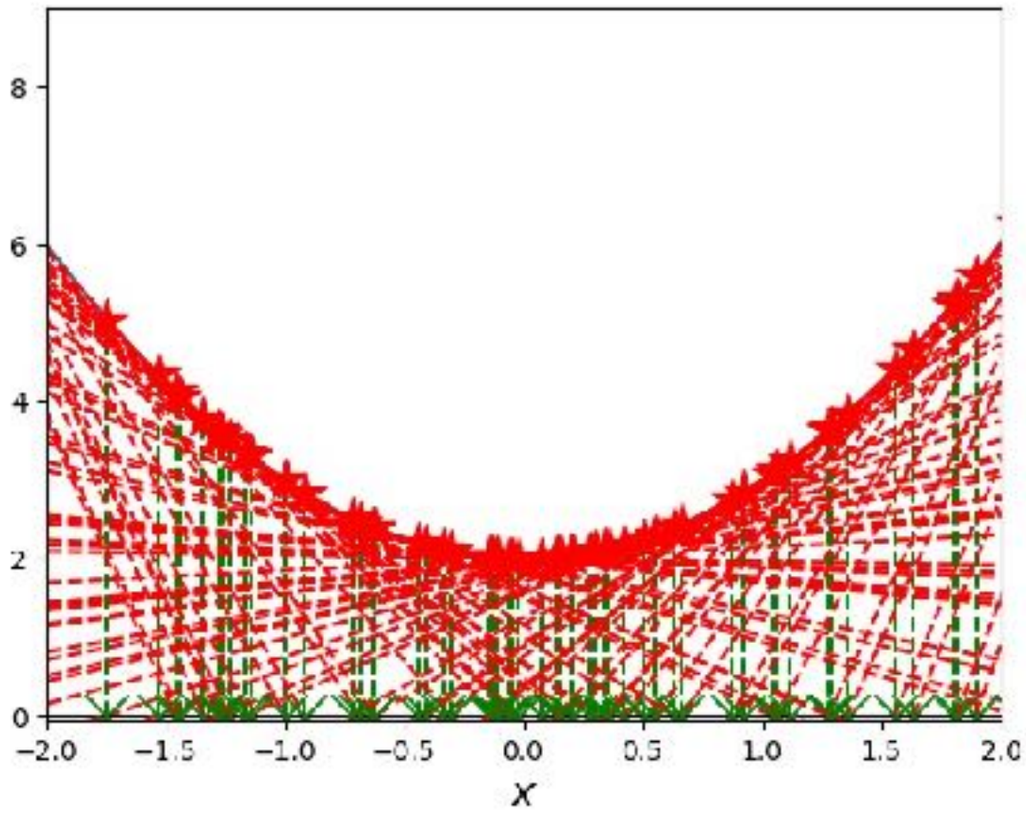
EXERCISE 1: BEWARE!

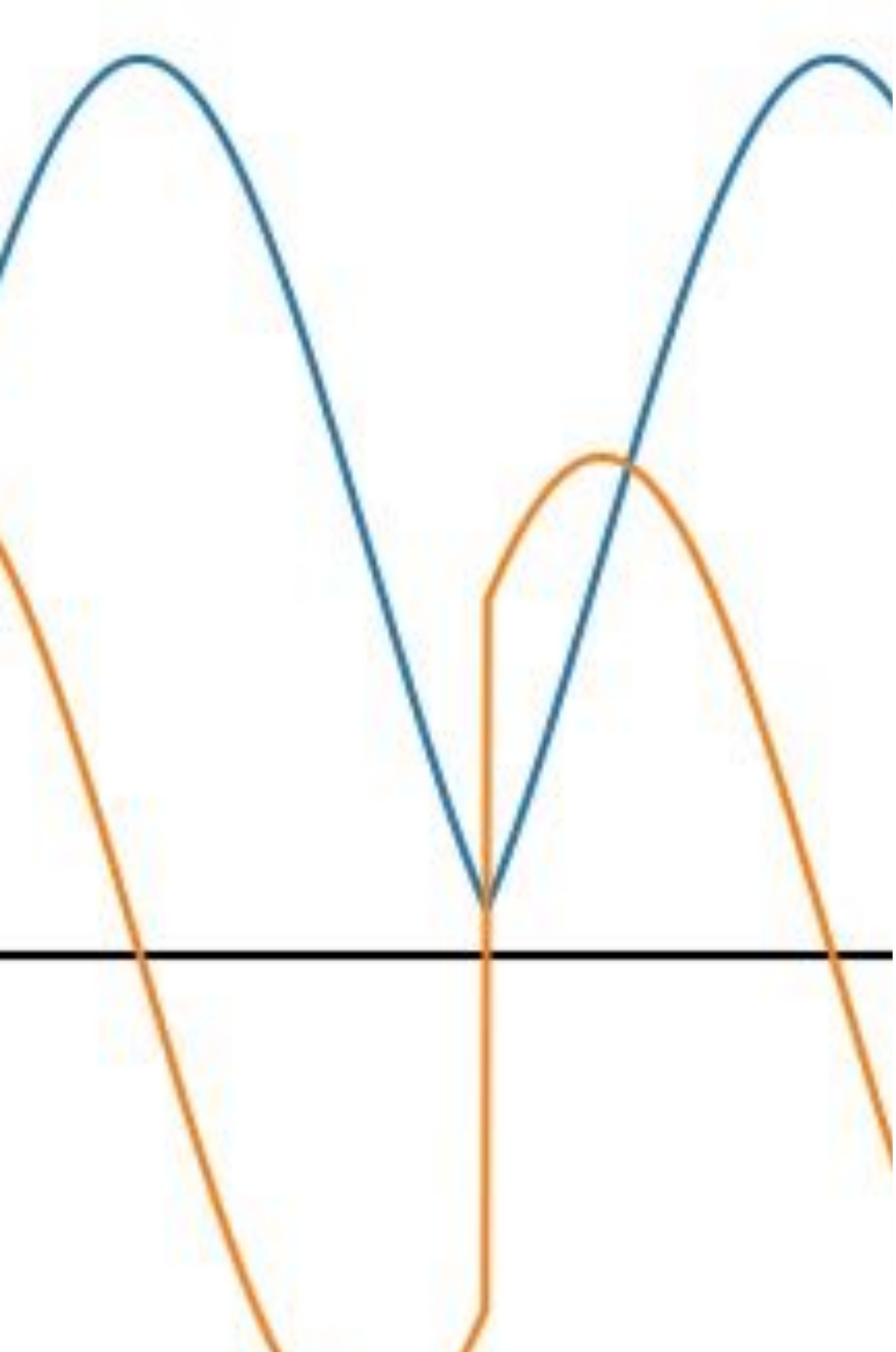
- Beware of the following:
- Regions in the figure, where function and slope fall off monotonously, but don't reach zero
- Let's back this up by our red-dot-plot
- Maximum number of iterations is capped at 20 here (would be infinite otherwise)



EXERCISE 1: BEWARE!

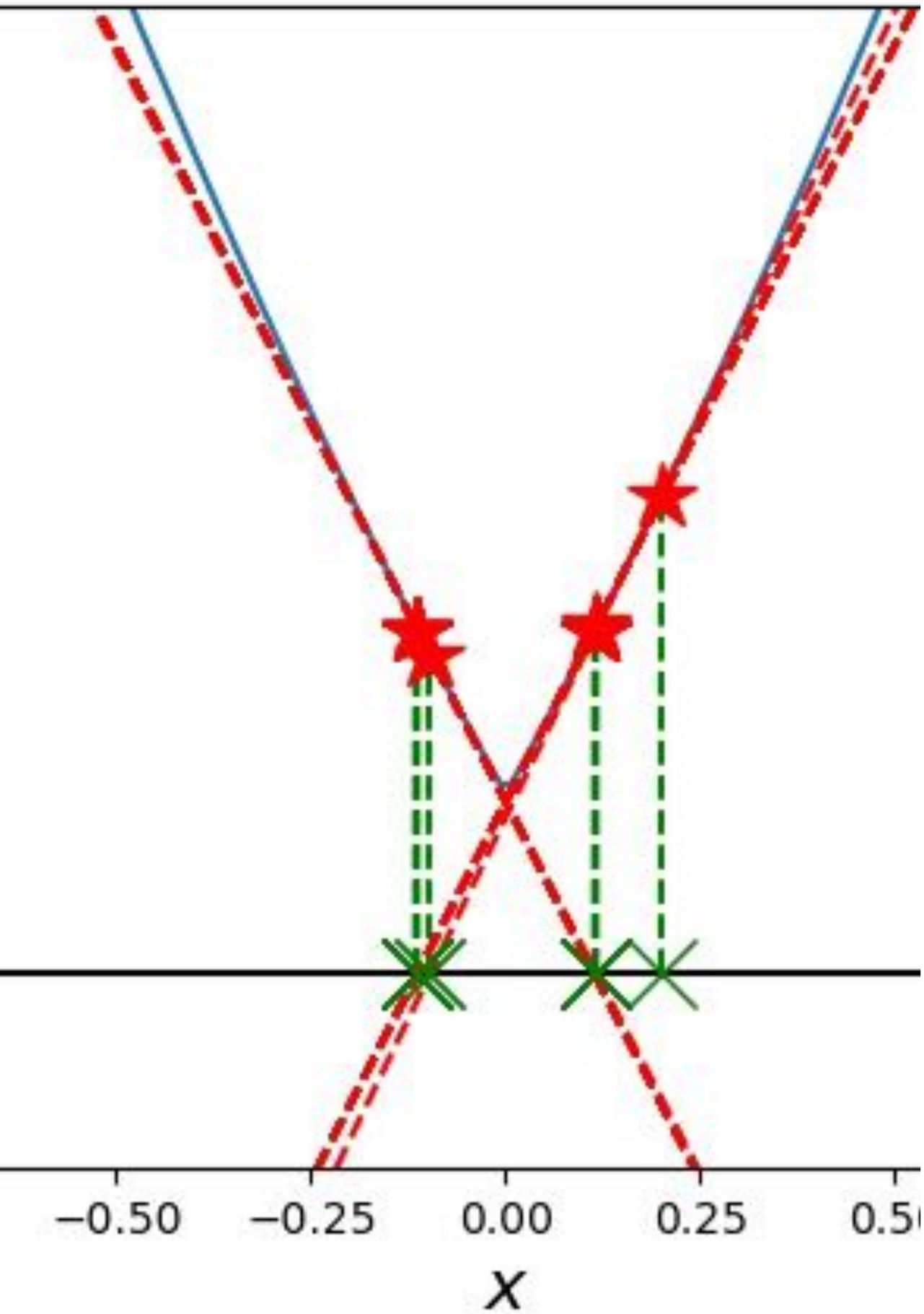
- Beware of the following:
- Functions that don't actually have a zero
- Then Newton's method will go on forever without finding anything





EXERCISE 1: BEWARE!

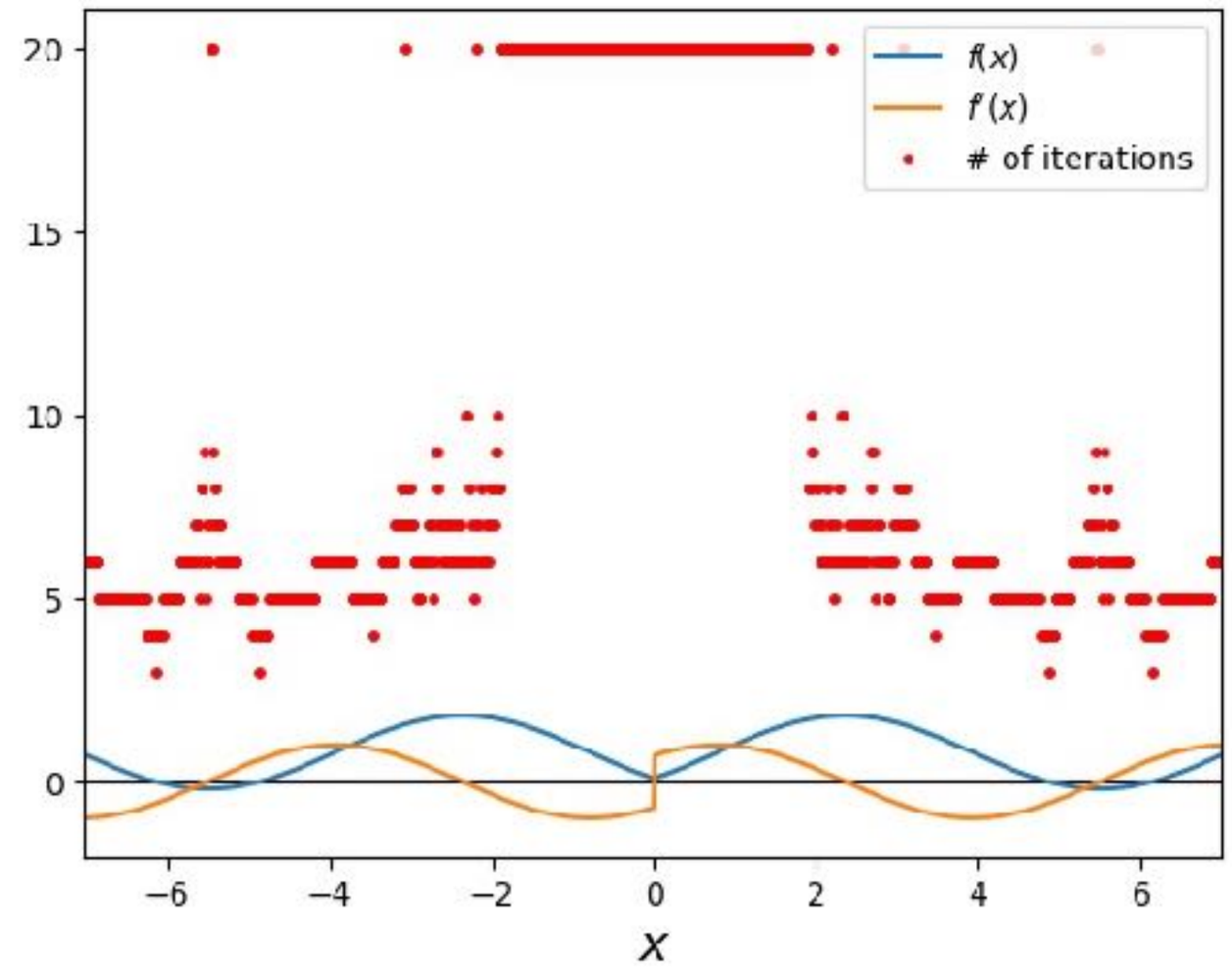
- Beware of the following:
- Regions in the figure, between which the Newton algorithm can oscillate
- Example given on the left
- The region of interest is in the middle
- Blue curve is the function, orange is the derivative
- What happens?



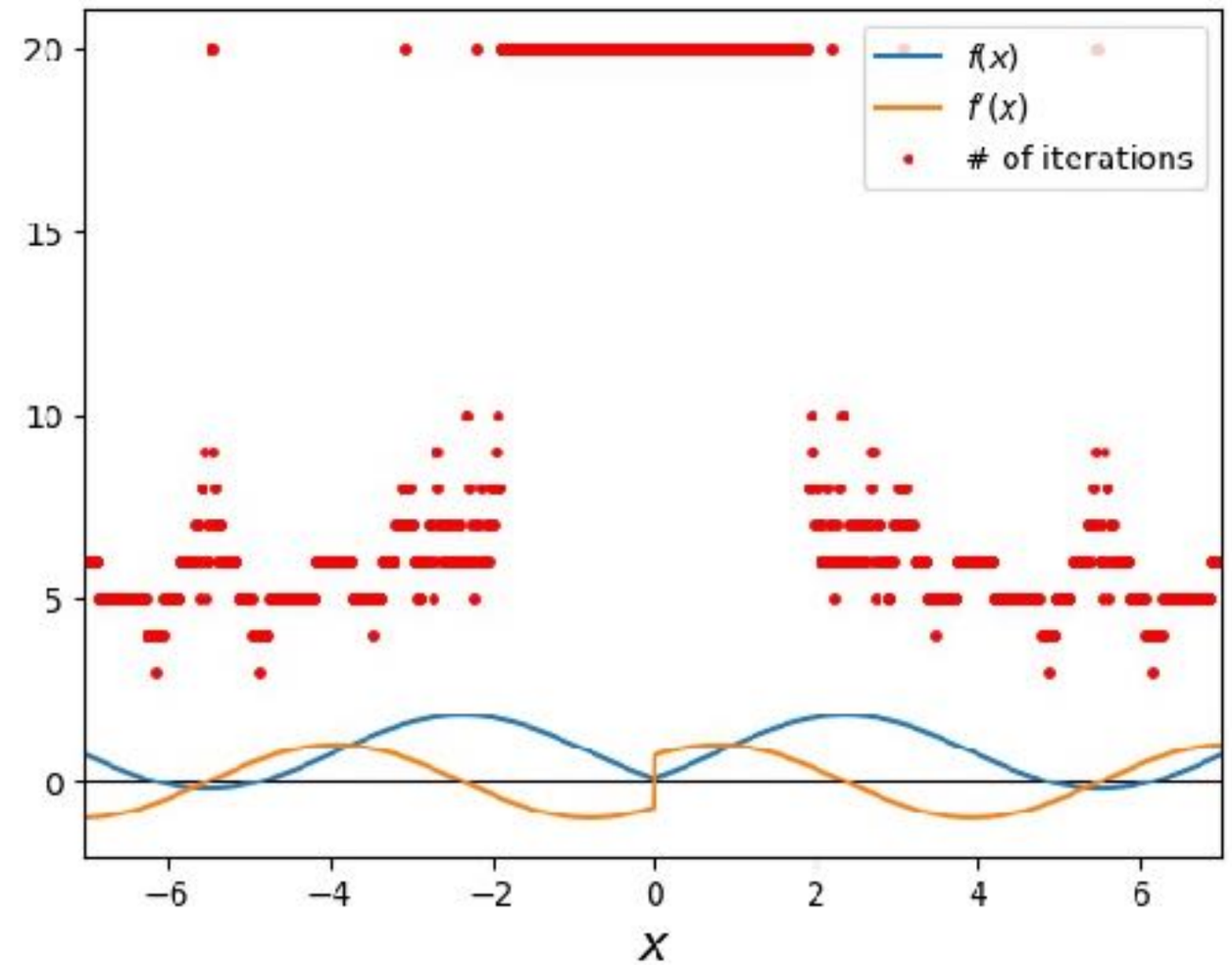
EXERCISE 1: BEWARE!

- Beware of the following:
- Regions in the figure, between which the Newton algorithm can oscillate
- Example given on the left
- The region of interest is in the middle
- Blue curve is the function, orange is the derivative
- What happens?
- Oscillation jumps back and forth, but doesn't reach a zero

EXERCISE 1: BEWARE!



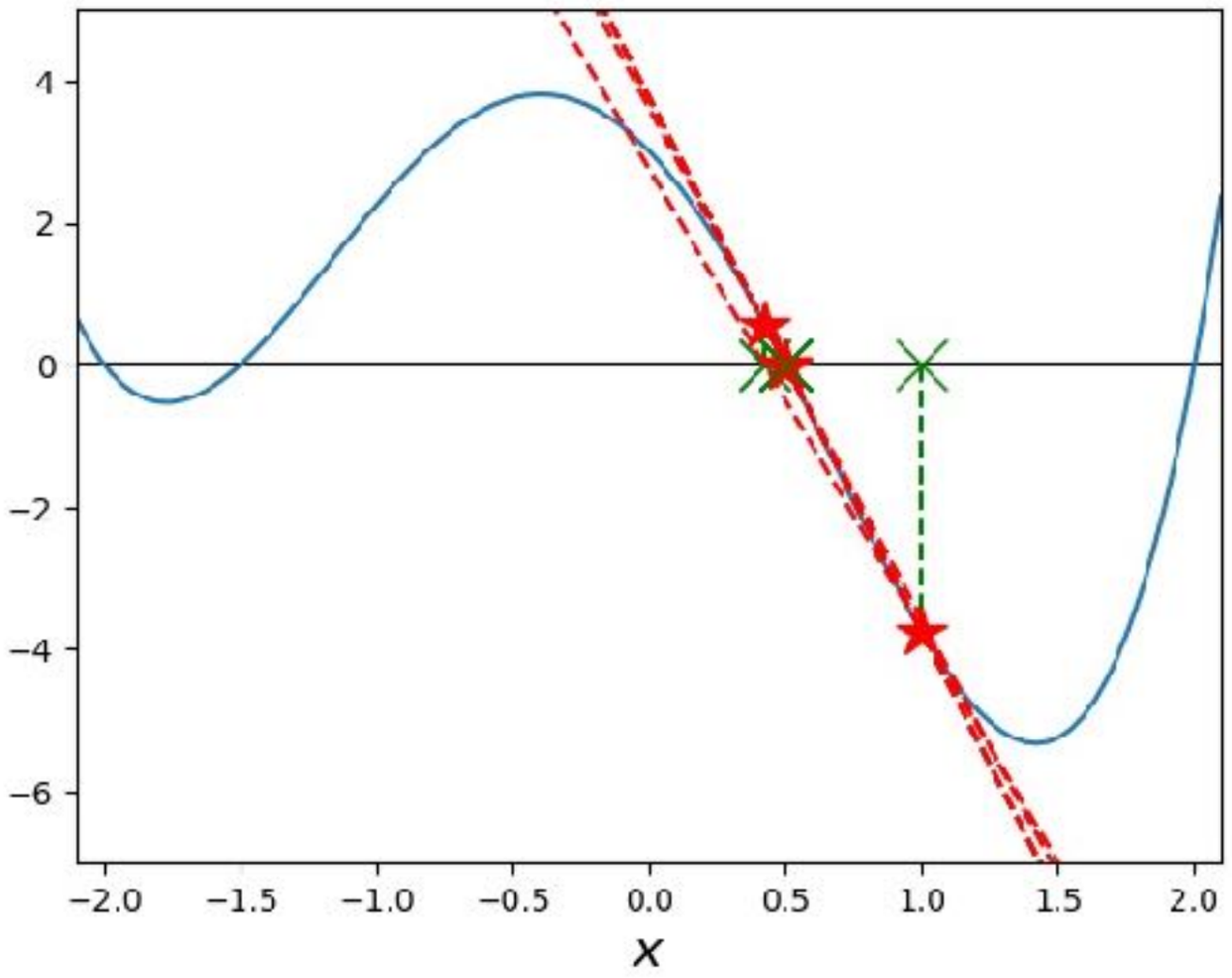
- Beware of the following:
- Regions in the figure, between which the Newton algorithm can oscillate
- Example given on the left
- The region of interest is in the middle
- What happens?
- Oscillation jumps back and forth, but doesn't reach a zero
- Check out our red-dot-plot
- Capped again at 20 iterations



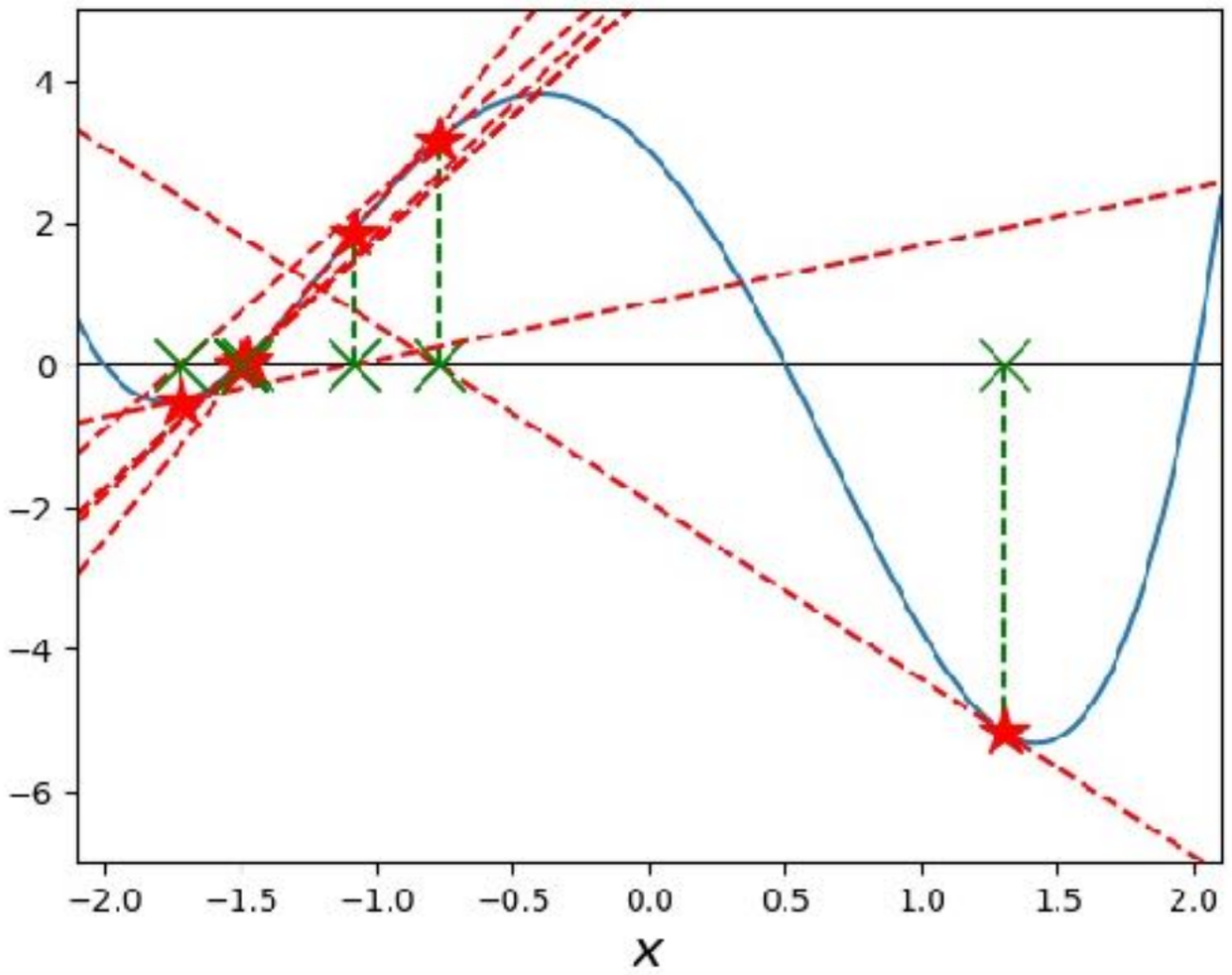
EXERCISE 1: SUMMARY

- Keep in mind about Newton's method:
- Finds roots, if there are any
- Depends on initial guess
- Can jump around via slope of the curve
- Can fail due to asymptotic region
- Can fail due to oscillation
- Fails at zero derivative

EXERCISE 1: OUTLOOK

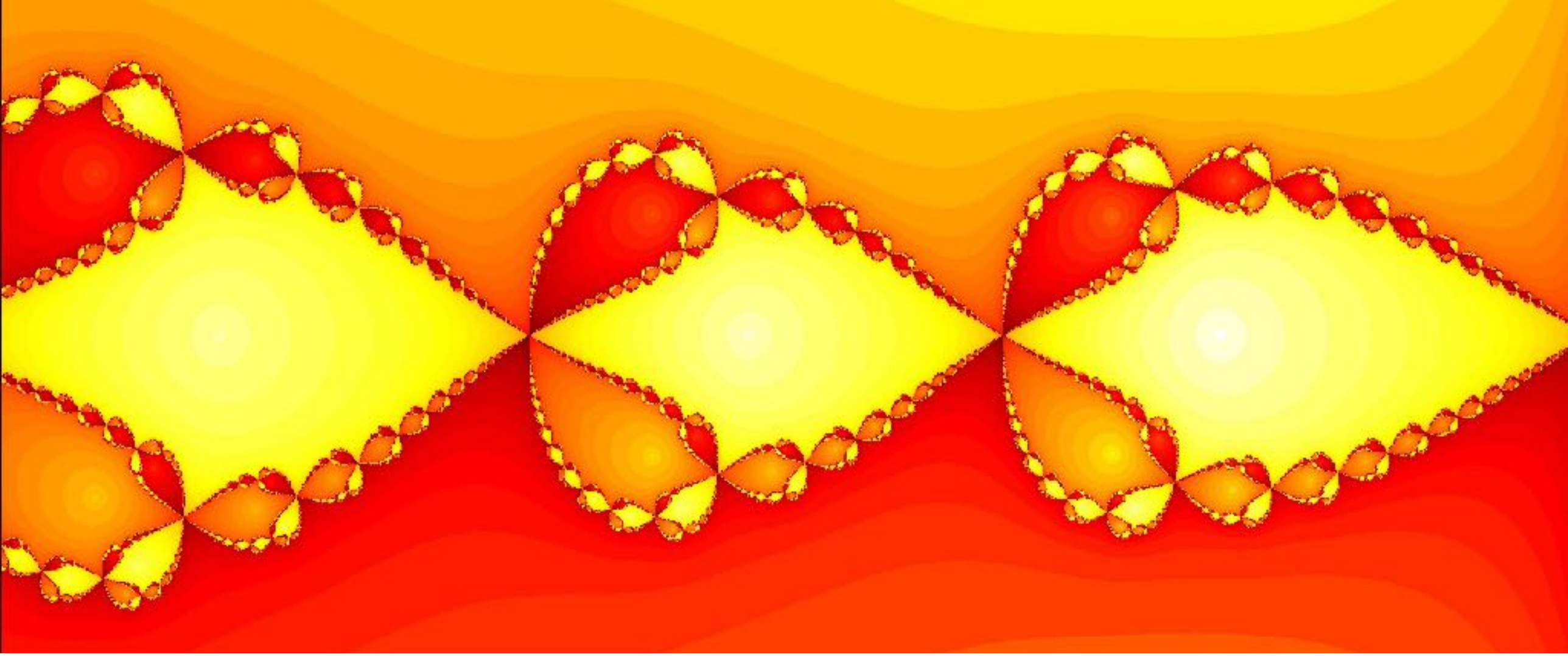


- What we haven't really looked at so far:
- Dependence on initial guess for a function with more than one root
- Why is that interesting?
- Because solution (root) that Newton's method converges to depends on the initial guess
- Look at example on the left
- Start with initial guess of 1.0



EXERCISE 1: OUTLOOK

- What we haven't really looked at so far:
- Dependence on initial guess for a function with more than one root
- Why is that interesting?
- Because solution (root) that Newton's method converges to depends on the initial guess
- Look at example on the left
- Start instead with initial guess of 1.3



TOWARDS LINEAR ALGEBRA

Numerical methods concerning vectors (matrices)

$$A_{ij} = \begin{pmatrix} a_{11} & \dots & a_{1M} \\ \vdots & \ddots & \vdots \\ a_{N1} & \dots & a_{NM} \end{pmatrix}$$

WHY MATRICES?

- Matrices arise all the time
- Different contexts
- Various applications
- We'll find enough examples to compute for a lifetime
- General structure is an array of numbers with *N* rows and *M* columns
- Used to have compact way to write equations, relations, etc.
- Defines a general type of data

EXAMPLE: SYSTEM OF LINEAR EQUATIONS

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

$$A \cdot x = b$$

$$A \cdot B = C$$

- Simplest: 2 linear equations in 2 variables
- four coefficients *a* and two terms *b*
- matrix notation is more compact already at this level
- even more so, when we start multiplying matrices, etc.
- We all know and believe this (I hope)
- Similar difference in computing

$$V_i = \begin{pmatrix} v_1 \\ \vdots \\ v_N \end{pmatrix}$$

COMPUTING WITH VECTORS

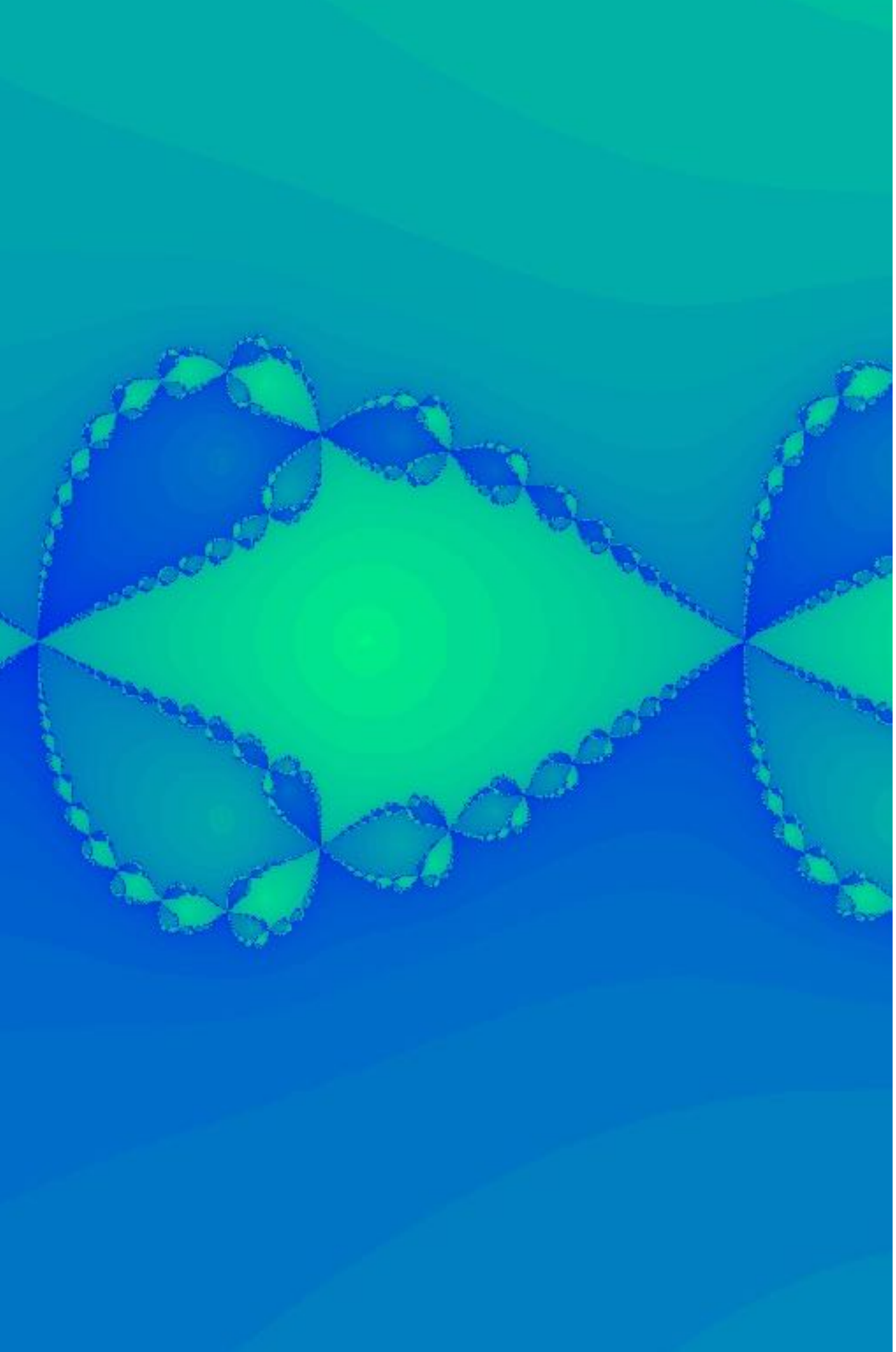
- Before we jump into matrices, let's think about vectors
- vectors are 1-D arrays
- work very similarly to matrices
- Let's check out a computational principle called *vectorisation*
- The idea is to make an operation more efficient that must be carried out on many elements at once
- Why is this even an issue?

$$A_i = \begin{pmatrix} v_1 \\ \vdots \\ v_m \\ v_{m+1} \\ \vdots \\ v_{2m} \\ v_{2m+1} \\ \vdots \\ \vdots \\ \vdots \\ v_{(k-1)m} \\ v_{(k-1)m+1} \\ \vdots \\ v_{km} \end{pmatrix}$$

$$km = N$$

VECTORISATION IN A NUTSHELL

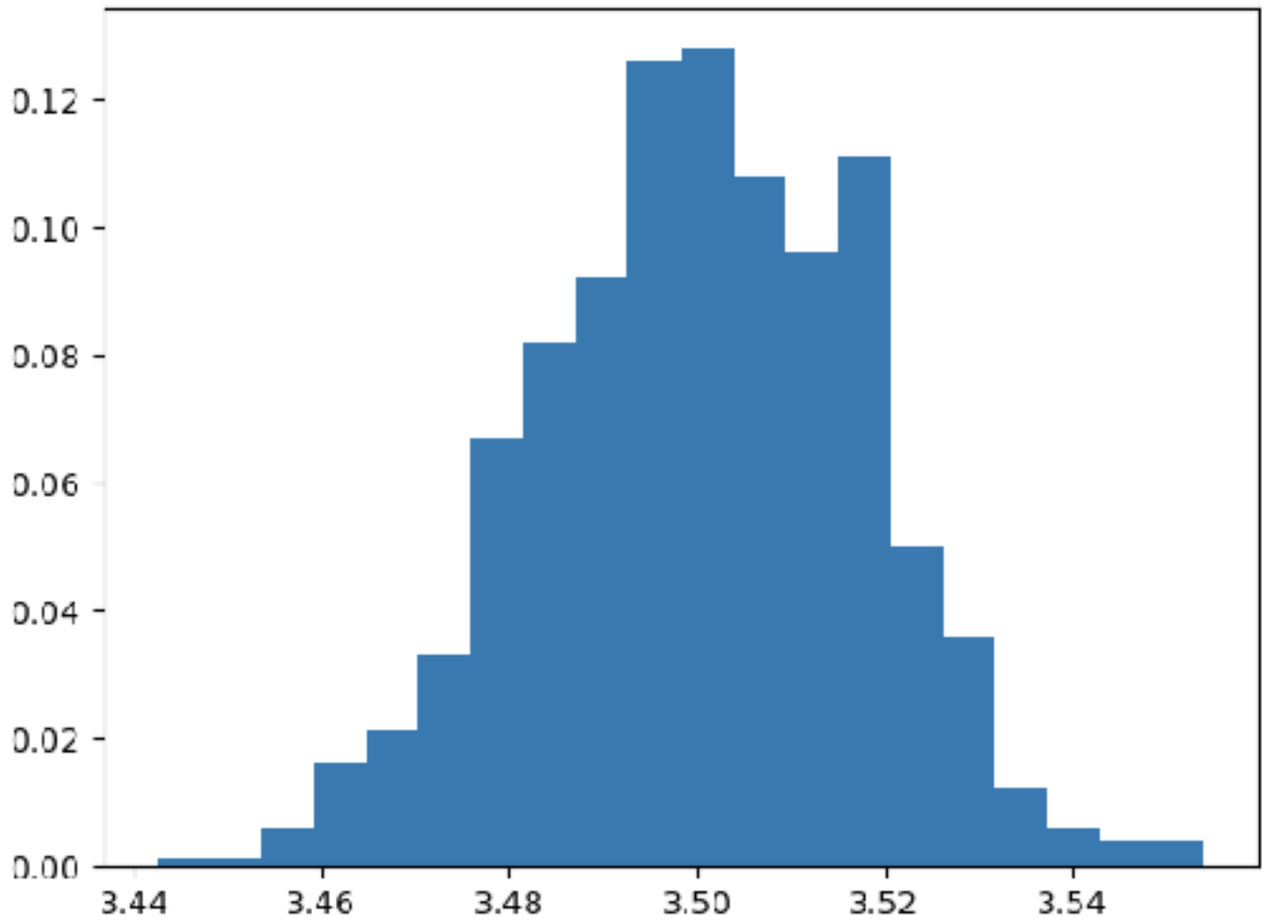
- Modern CPUs have better capabilities than early ones
- Sequential at first: Loop
- Parallel later: Loops taken apart, etc.
- Vectorised: Use CPU register to store and operate on several pieces of data at once
- Vectorisation is taken care of automatically:
  - Compiler
  - Libraries (e.g. numpy)



HOW TO VECTORIZE?

- Have it done automatically
- Provide Code that is vectorised and tell the compiler, if necessary
  - loops with compiler option
  - vectorised libraries (numpy)
- Use data such that it can be vectorised (dependencies)
- Keep in mind that arrays can get large
- It’s a game of memory vs. execution time

EXAMPLE: SIMULATE DICE



- Sample random dice
- get averages and such
- use samples of certain size
- compare loops to numpy

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

EXERCISE 2

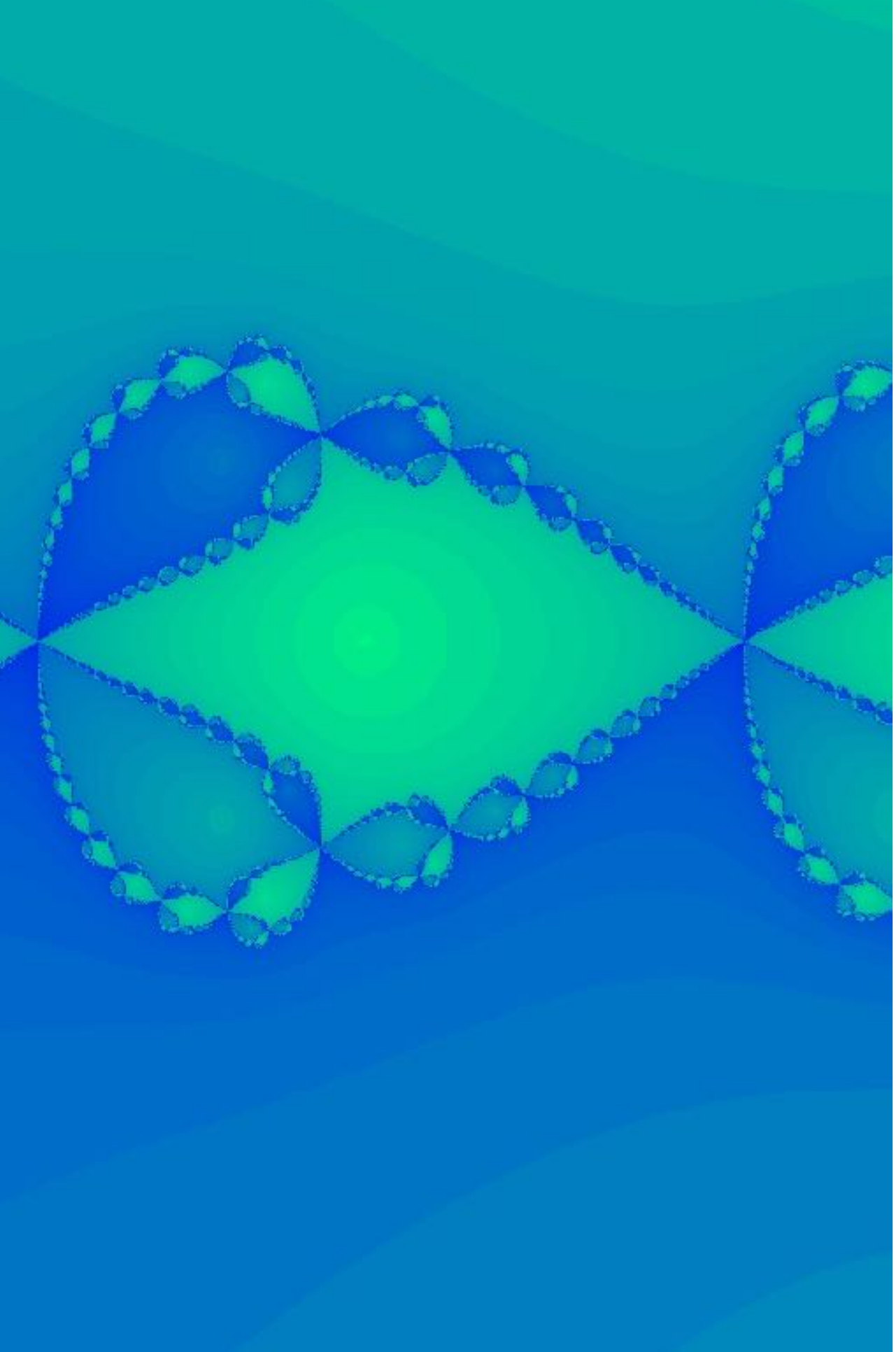
- Implement Newton's method in the complex plane
- Define a complex grid of starting points
- set a maximum number of iterations and a target relative precision of 10^{-10}
- Pick a function that has several different roots in the complex plane, list the roots numerically
- Let Newton's method run from each starting point on the grid
- Identify the converged solution with one of the roots and note the number of necessary iterations needed to get there
- Translate these numbers into colors and plot the corresponding Newton fractal.
- Typical Example functions on the left

$f(z) =$

$z^3 - 1$

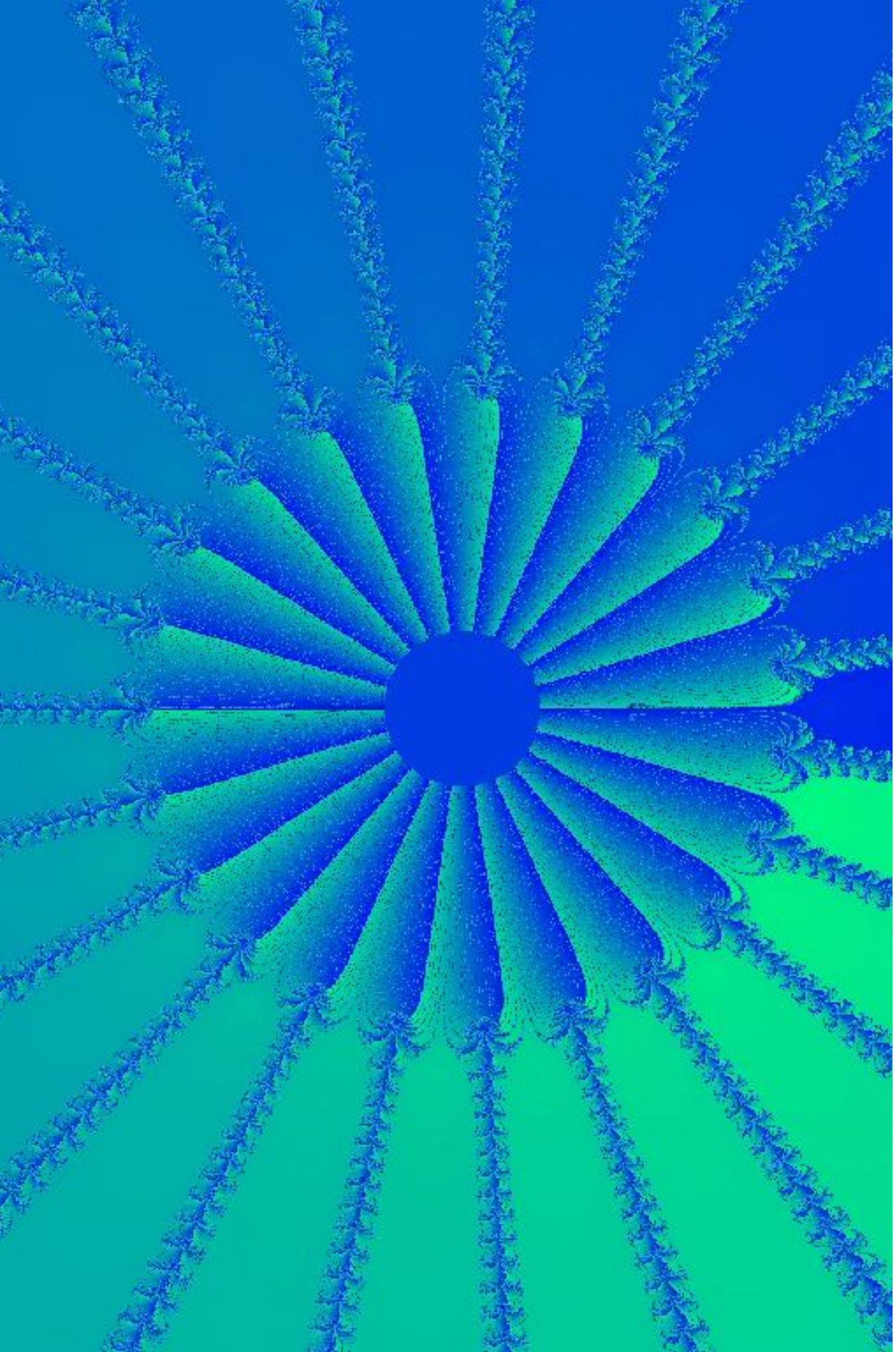
$(z^2 - 1)(z^2 + 1)$

$\sin(z)$



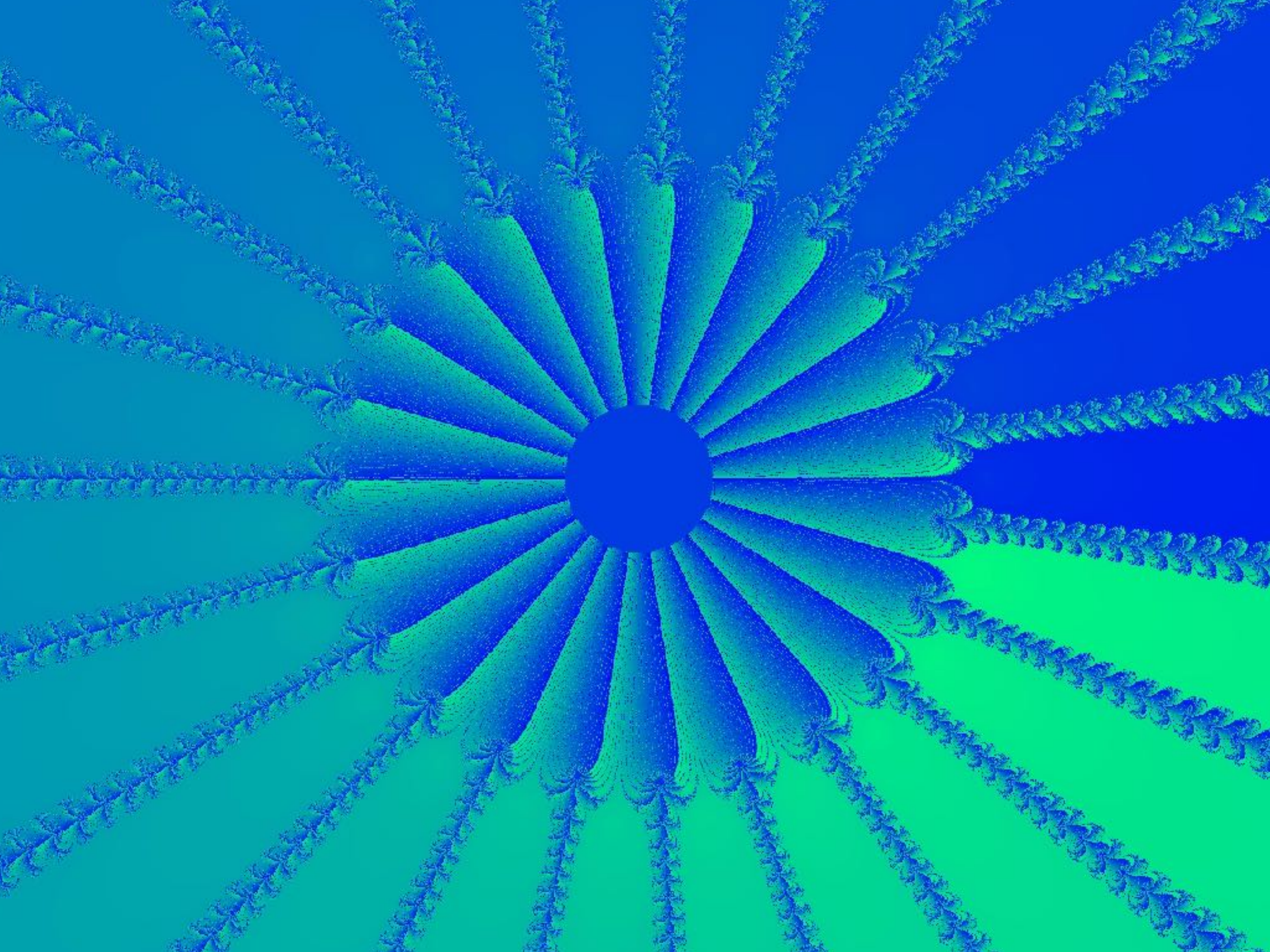
EXERCISE 2: RESULTS

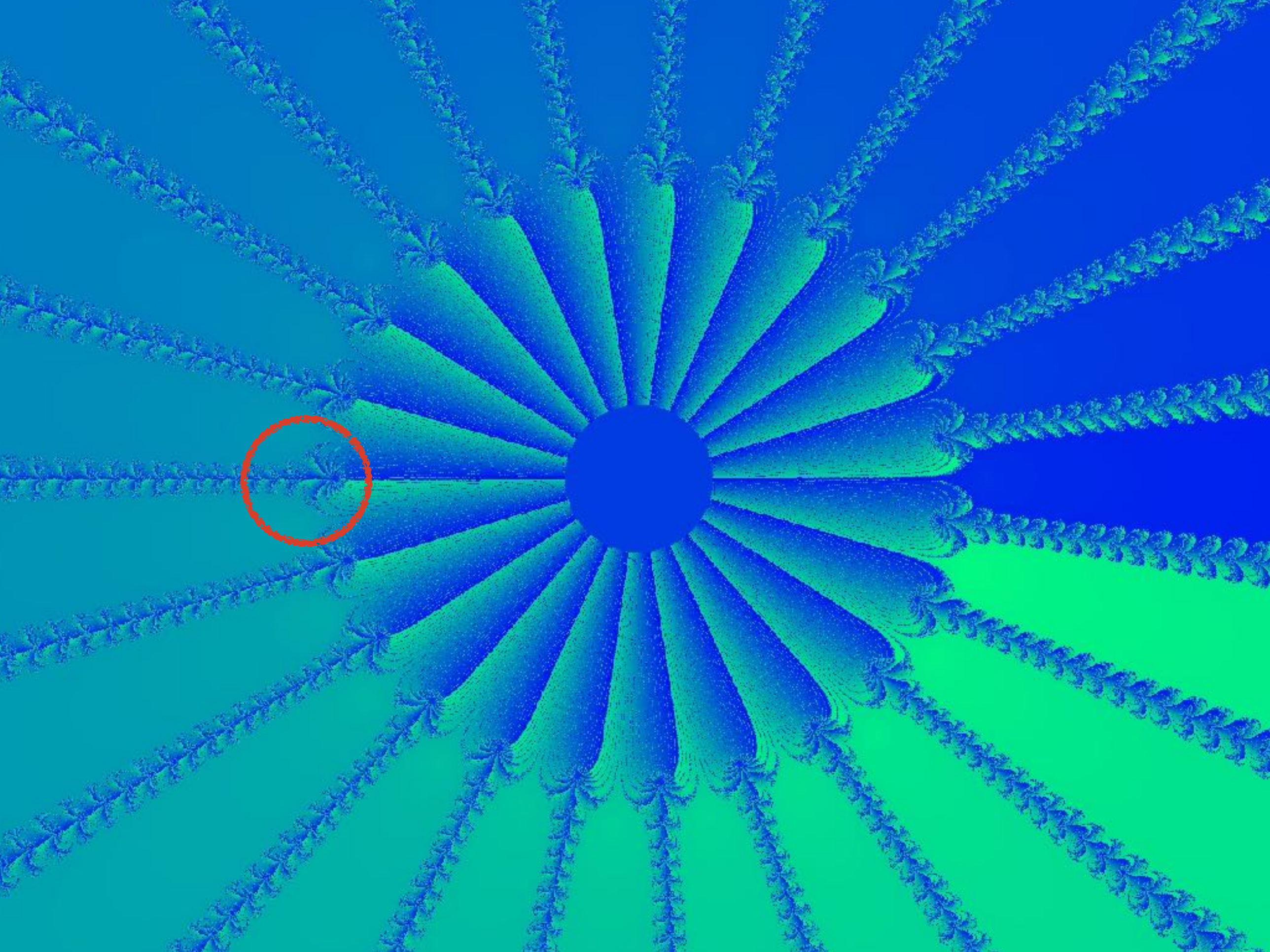
- Create a grid of complex numbers with numpy: One large vector
- For the entire vector, call the function *-f/df* (that's the increment for each iteration) and get another large vector
- Subtract the two and check for convergence by comparing relative difference to predefined small number
- Increase counter vector according to convergence (vectorized **if**)
- Compare the resulting list of complex values to a predefined or auto-generated list of roots

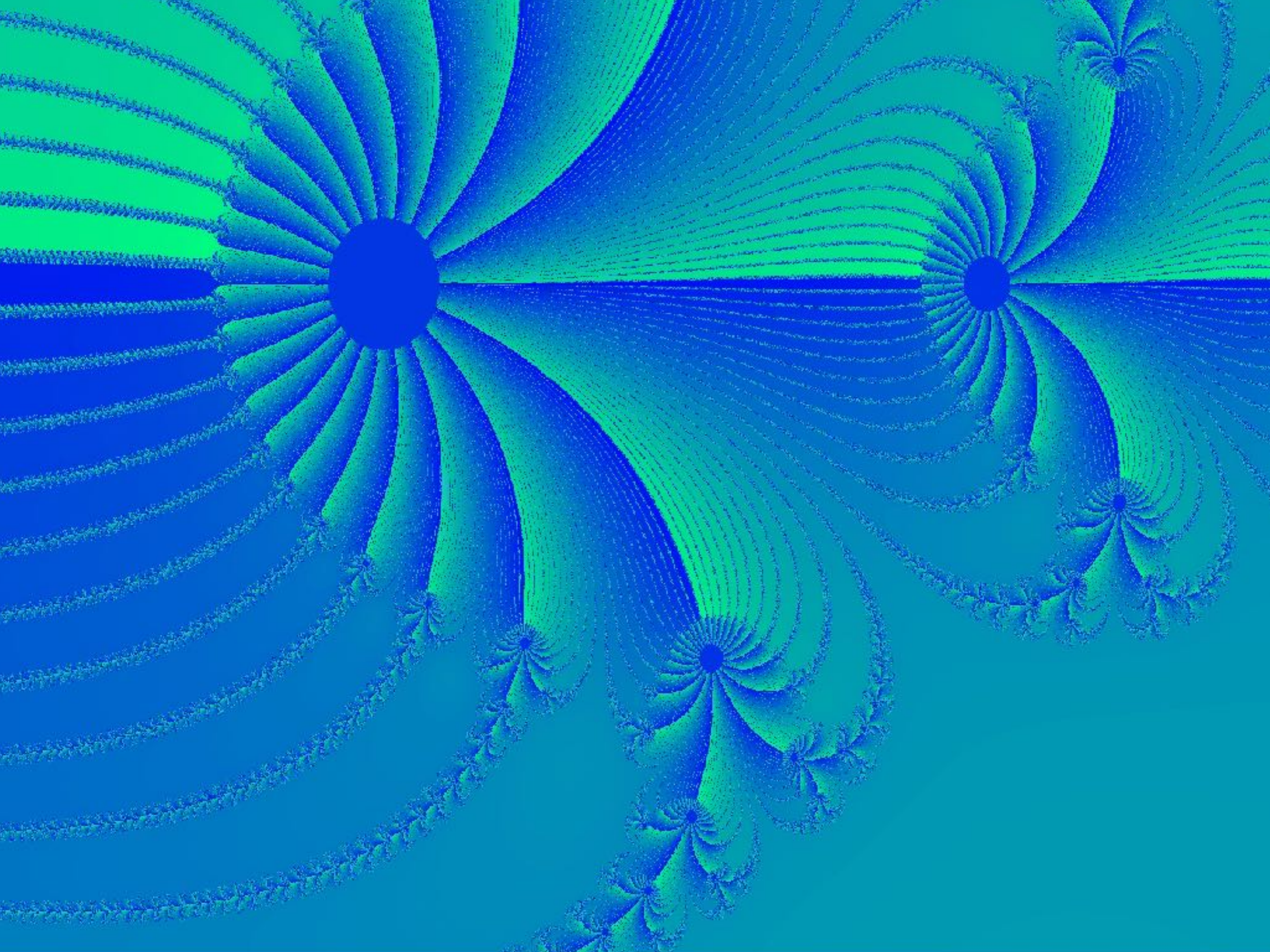


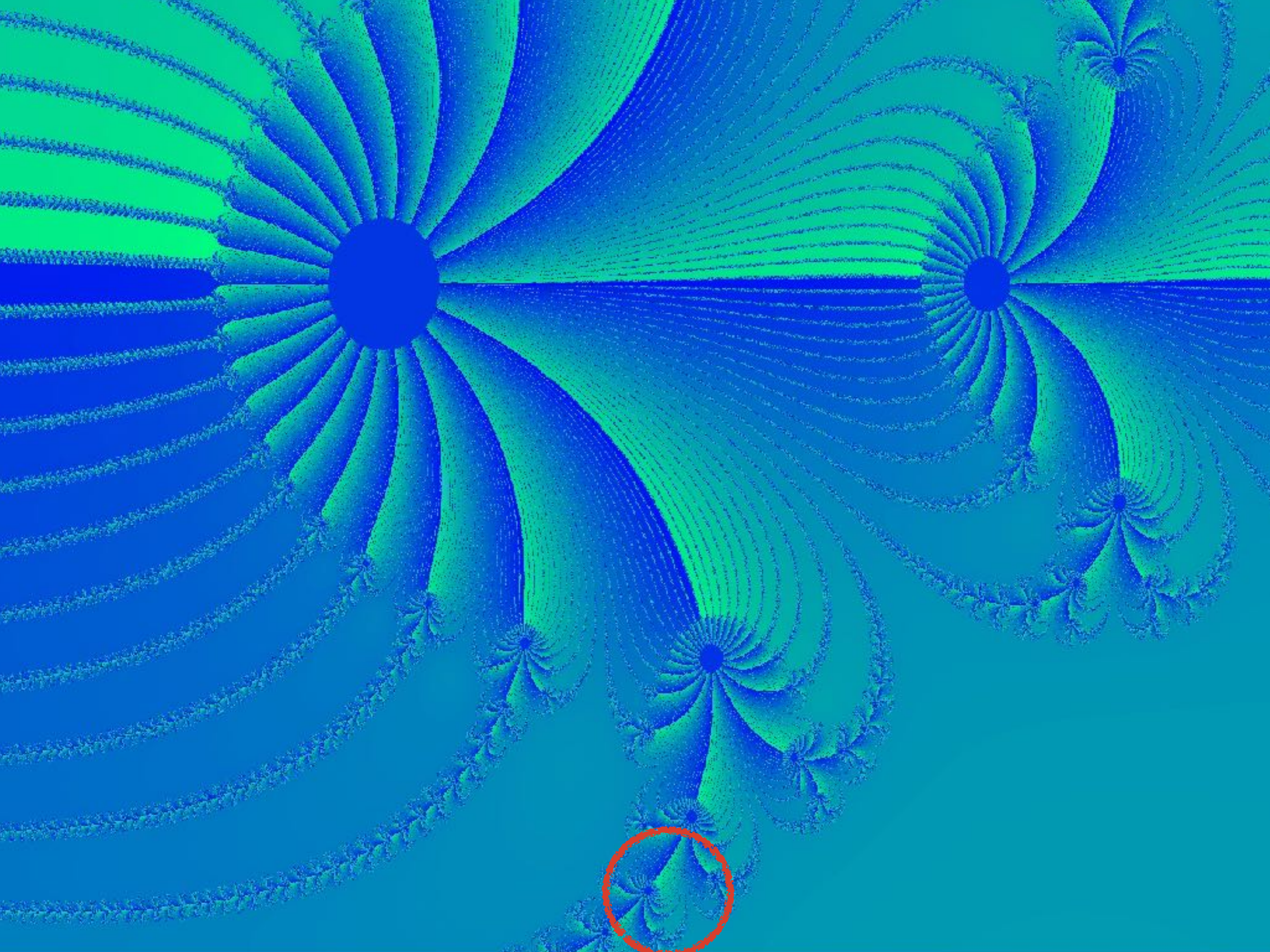
EXERCISE 2: RESULTS

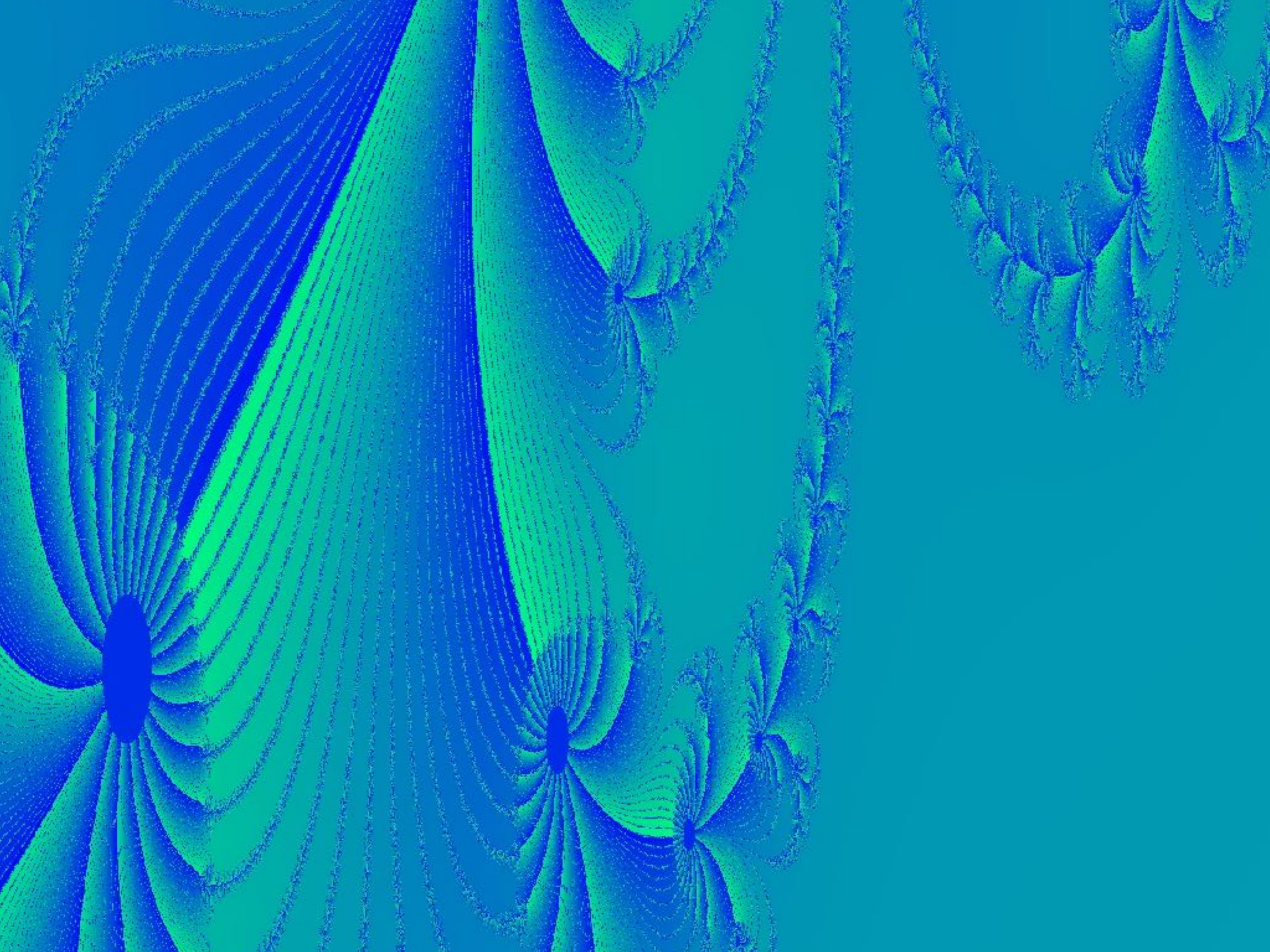
- Look at the details of the code
- numpy functions operate on vectors
- sometimes that requires the construction of additional vectors (e.g. to compare changing vectors to)
- Speed improvement over loop is remarkable
- A few example pictures:
  $f(x)=x^{23}-1$

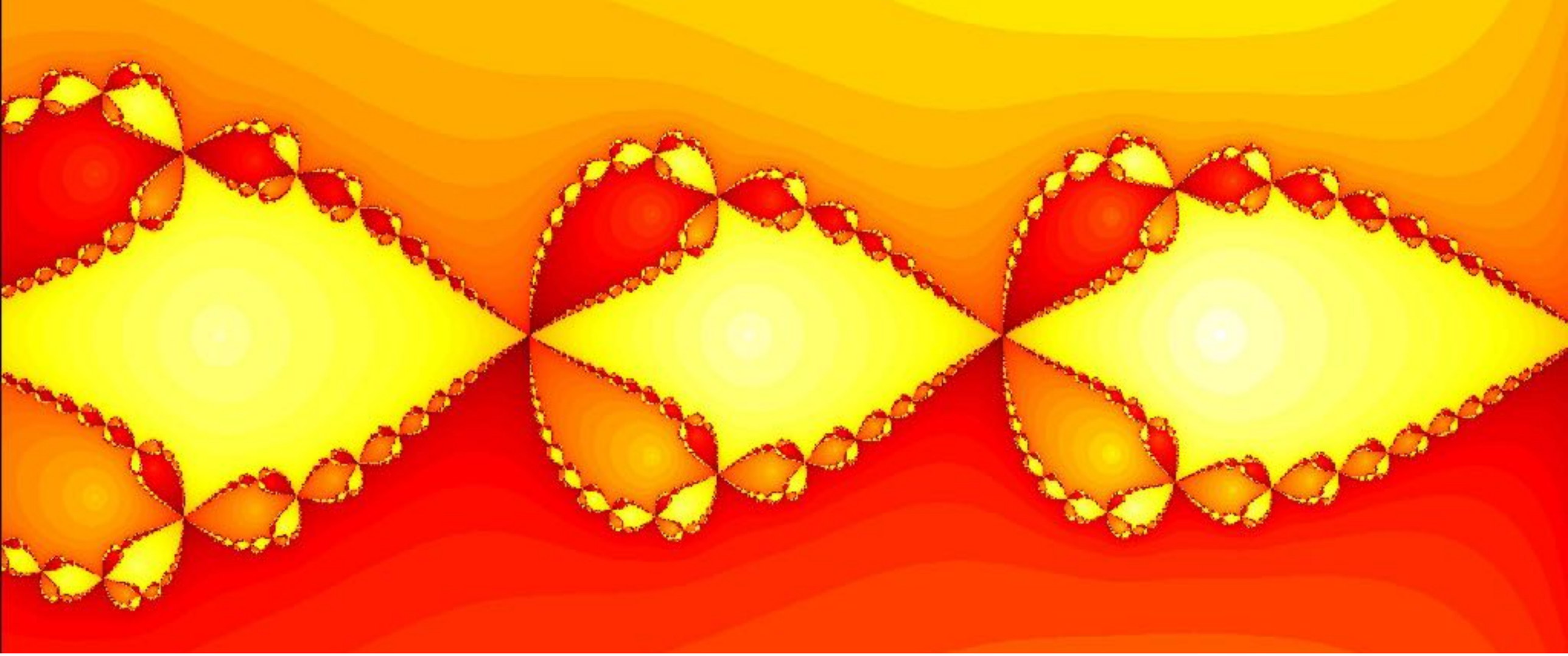












LINEAR ALGEBRA

Using actual LA methods and libraries

RECUERDOS DE ÁLGEBRA LINEAL

$$A_{ij} = \begin{pmatrix} a_{11} & \dots & a_{1M} \\ \vdots & \ddots & \vdots \\ a_{N1} & \dots & a_{NM} \end{pmatrix}$$

- Dealing with matrices and systems of linear equations
- Typical things to do:
  - compute determinant of a matrix
  - invert a matrix
  - compute eigenvalues and -vectors of a matrix
  - solve a linear system
- In practice we use numerical libraries for that, so let's get a feel for what that's like

$$A_{ij} = \begin{pmatrix} a_{11} & \dots & a_{1M} \\ \vdots & \ddots & \vdots \\ a_{N1} & \dots & a_{NM} \end{pmatrix}$$

LIBRARIES FOR LINALG

- Python:
  scipy.linalg
- Julia:
  native: **LinearAlgebra**
- Go ahead and find those libraries and documentation:
- Python:
  https://docs.scipy.org/doc/scipy/reference/linalg.html
- Julia:
  https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/

$$A_{ij} = \begin{pmatrix} a_{11} & \dots & a_{1M} \\ \vdots & \ddots & \vdots \\ a_{N1} & \dots & a_{NM} \end{pmatrix}$$

EXERCISE 3

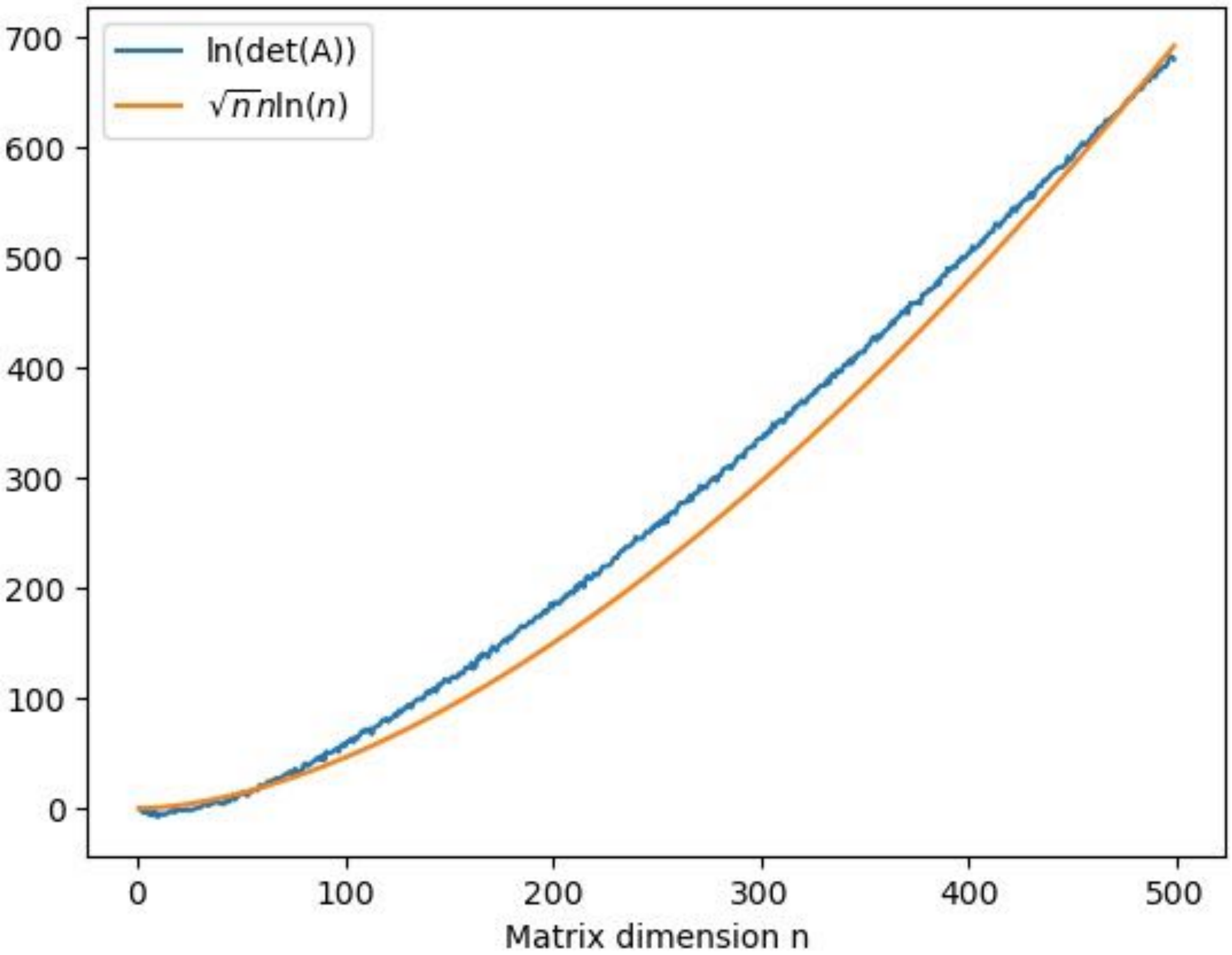
- Construct a square example matrix numerically
- Use these libraries to:
  - compute the determinant of the matrix
  - compute its eigenvalues and eigenvectors
- Construct an example for linear system of equations
- Use these libraries to:
  - solve the system
- Play around with the possibilities
- Try and benchmark something

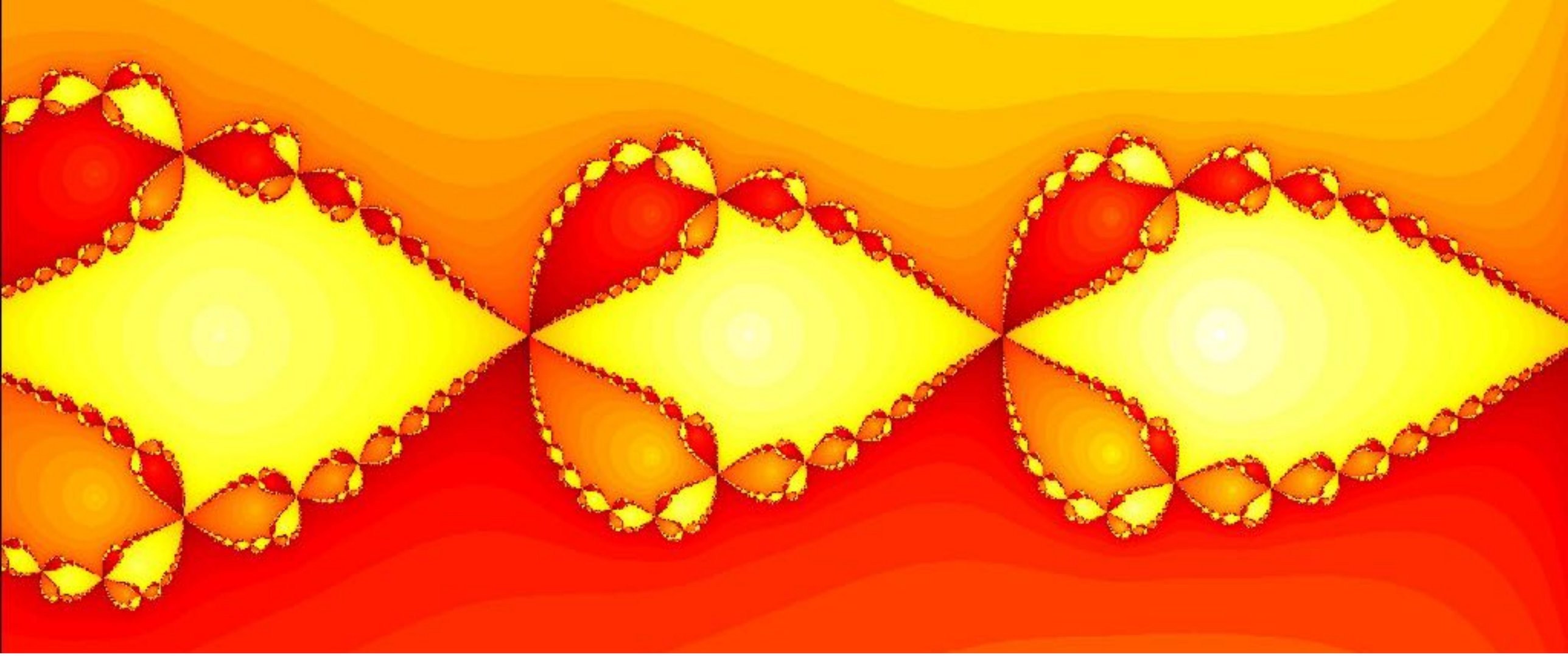
$$A_{ij} = \begin{pmatrix} a_{11} & \dots & a_{1M} \\ \vdots & \ddots & \vdots \\ a_{N1} & \dots & a_{NM} \end{pmatrix}$$

EXERCISE 3: RESULTS

- On determinants of large matrices:
  - They aren't as important as other things like, e.g. eigenvalues, figuring out the rank, etc.
  - Actually, the determinant of random matrices scales drastically with the matrix dimension:

$$\ln(\det(A)) \approx \sqrt{(n)} n \ln(n)$$


ln(det(A))
$\sqrt{n}\,n\ln(n)$
700
600
500
400
300
200
100
0
0
100
200
300
400
500
Matrix dimension n



LINEAR ALGEBRA

Direct vs. Iterative Methods

NUM. METHODS IN LINALG

$$A_{ij} = \begin{pmatrix} a_{11} & \dots & a_{1M} \\ \vdots & \ddots & \vdots \\ a_{N1} & \dots & a_{NM} \end{pmatrix}$$

- Numerical methods in linear algebra are basically twofold:
  - Direct methods
  - Iterative methods
- Both work with a number of steps (mostly, e.g., operating/doing something with a matrix)
- Direct method is done after finite number of steps
- Iterative method produces an approximation after finite number of steps

DIRECT METHOD: EXAMPLE

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 4 \\ 3 & 8 & 7 \end{pmatrix}$$

- Determinant of a matrix via Gaussian elimination
- How does that work?
- Look a the Matrix on the left
- In Gaussian elimination, you can add a multiple of one row/ column to one of the others
- Such a step does not change the determinant
- E.g., take first row times 3 and subtract it from the other two

DIRECT METHOD: EXAMPLE

$$A \to \begin{pmatrix} 1 & 2 & 3 \\ 0 & -2 & -5 \\ 0 & 2 & -2 \end{pmatrix}$$

- Determinant of a matrix via Gaussian elimination
- How does that work?
- Look a the Matrix on the left
- In Gaussian elimination, you can add a multiple of one row/ column to one of the others
- Such a step does not change the determinant
- Looking good!
- Now, take second row and add to the third

DIRECT METHOD: EXAMPLE

$$A \to \begin{pmatrix} 1 & 2 & 3 \\ 0 & -2 & -5 \\ 0 & 0 & -7 \end{pmatrix}$$

- Determinant of a matrix via Gaussian elimination
- How does that work?
- Look a the Matrix on the left
- In Gaussian elimination, you can add a multiple of one row/ column to one of the others
- Such a step does not change the determinant
- Looking even better!
- Now, the determinant is the product of all elements in the diagonal

DIRECT METHOD: EXAMPLE

$$A \to \begin{pmatrix} 1 & 2 & 3 \\ 0 & -2 & -5 \\ 0 & 0 & -7 \end{pmatrix}$$

- Determinant of a matrix via Gaussian elimination
- Now, the determinant is the product of all elements in the diagonal
- Remember that this is the case for a triangular matrix
- The result is 14
- So, it took us something like 6 steps to get there (or however many operations it actually takes)

DIRECT METHOD: EXAMPLE

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 4 \\ 3 & 8 & 7 \end{pmatrix} \qquad b = \begin{pmatrix} 4 \\ 3 \\ 7 \end{pmatrix}$$

$$A \cdot x = b$$

- Solve a linear system of equations via Gaussian elimination
- How does that work?
- Exactly the same story, but with an additional vector
- Perform the same steps as before, but include the vector b

DIRECT METHOD: EXAMPLE

$$\left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 \\ 3 & 4 & 4 & 3 \\ 3 & 8 & 7 & 7 \end{array}\right]$$

- Solve a linear system of equations via Gaussian elimination
- How does that work?
- Exactly the same story, but with an additional vector
- Perform the same steps as before, but include the vector b
- Write next to each other, remember steps from before

DIRECT METHOD: EXAMPLE

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 0 & -2 & -5 & -9 \\ 0 & 2 & -2 & -5 \end{array}\right)$$

- Solve a linear system of equations via Gaussian elimination
- How does that work?
- Exactly the same story, but with an additional vector
- Perform the same steps as before, but include the vector b
- Write next to each other, remember steps from before, but we need to do a little bit more …

DIRECT METHOD: EXAMPLE

$$\left(\begin{array}{ccc|c} 1 & 0 & -2 & -5 \\ 0 & -2 & -5 & -9 \\ 0 & 0 & -7 & -14 \end{array}\right)$$

- Solve a linear system of equations via Gaussian elimination
- How does that work?
- Exactly the same story, but with an additional vector
- Perform the same steps as before, but include the vector b
- Write next to each other, remember steps from before, but we need to do a little bit more …

DIRECT METHOD: EXAMPLE

$$\left(\begin{array}{ccc|c} 1 & 0 & -2 & -5 \\ 0 & -2 & -5 & -9 \\ 0 & 0 & 1 & 2 \end{array}\right)$$

- Solve a linear system of equations via Gaussian elimination
- How does that work?
- Exactly the same story, but with an additional vector
- Perform the same steps as before, but include the vector b
- Get rid of all coefficients except for diagonal ones
- Also, make all diagonal coefficients equal to one

DIRECT METHOD: EXAMPLE

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & -2 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{array}\right)$$

- Solve a linear system of equations via Gaussian elimination
- How does that work?
- Exactly the same story, but with an additional vector
- Perform the same steps as before, but include the vector b
- Get rid of all coefficients except for diagonal ones
- Also, make all diagonal coefficients equal to one

DIRECT METHOD: EXAMPLE

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1/2 \\ 0 & 0 & 1 & 2 \end{array}\right)$$

- Solve a linear system of equations via Gaussian elimination
- How does that work?
- Exactly the same story, but with an additional vector
- Perform the same steps as before, but include the vector b
- Get rid of all coefficients except for diagonal ones
- Also, make all diagonal coefficients equal to one

LINEAR SYSTEMS SOLVER

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1/2 \\ 0 & 0 & 1 & 2 \end{array}\right)$$

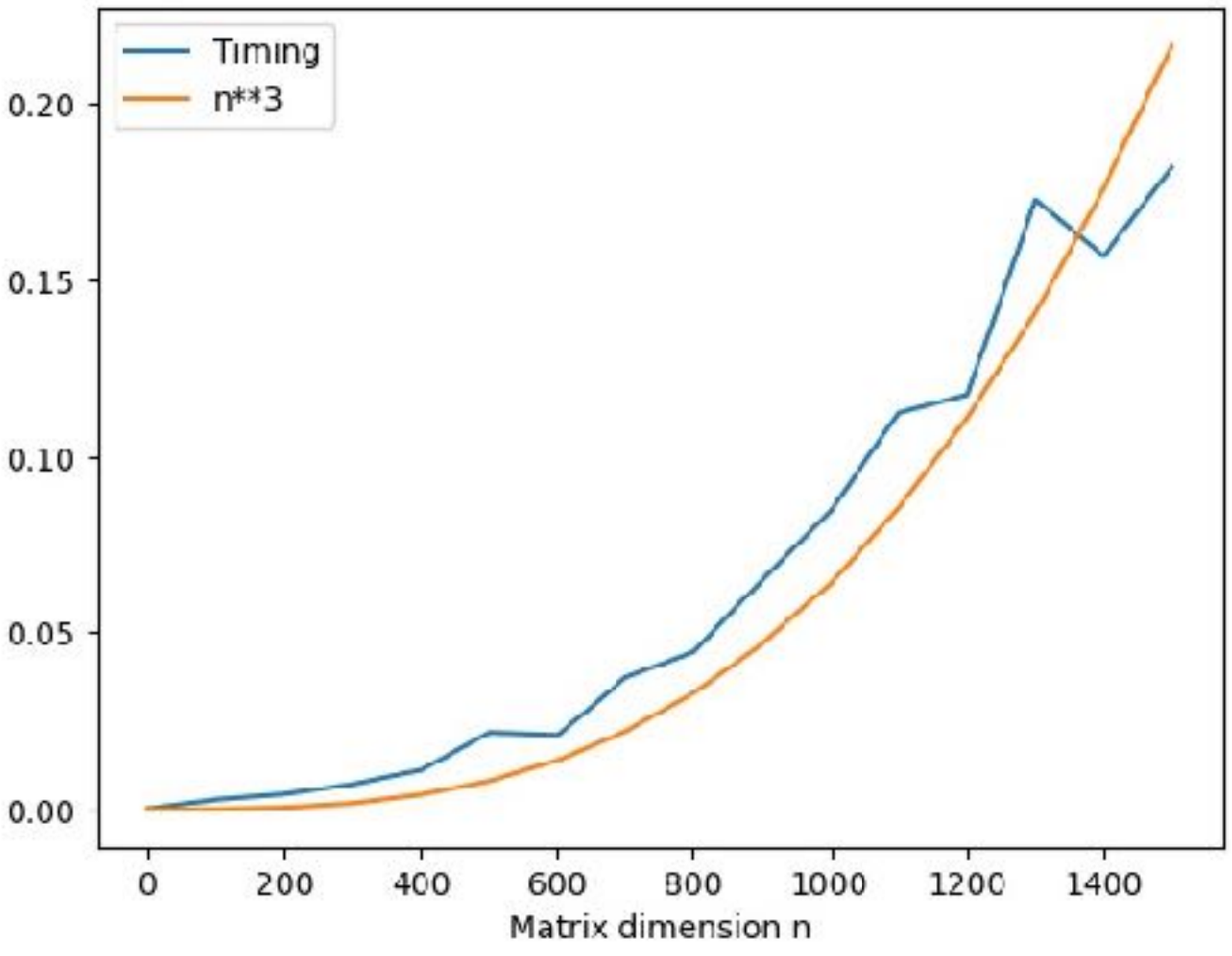
- Consider a linear system with n equations and n variables
- That gives us an nxn-matrix and an n-dimensional vector
- Gaussian elimination has a computational complexity of $\mathcal{O}(n^3)$
- We have various libraries, like scipy.linalg
- Is $\mathcal{O}(n^3)$ the actual scaling?

$$A_{ij} = \begin{pmatrix} a_{11} & \dots & a_{1M} \\ \vdots & \ddots & \vdots \\ a_{N1} & \dots & a_{NM} \end{pmatrix}$$

EXERCISE 4

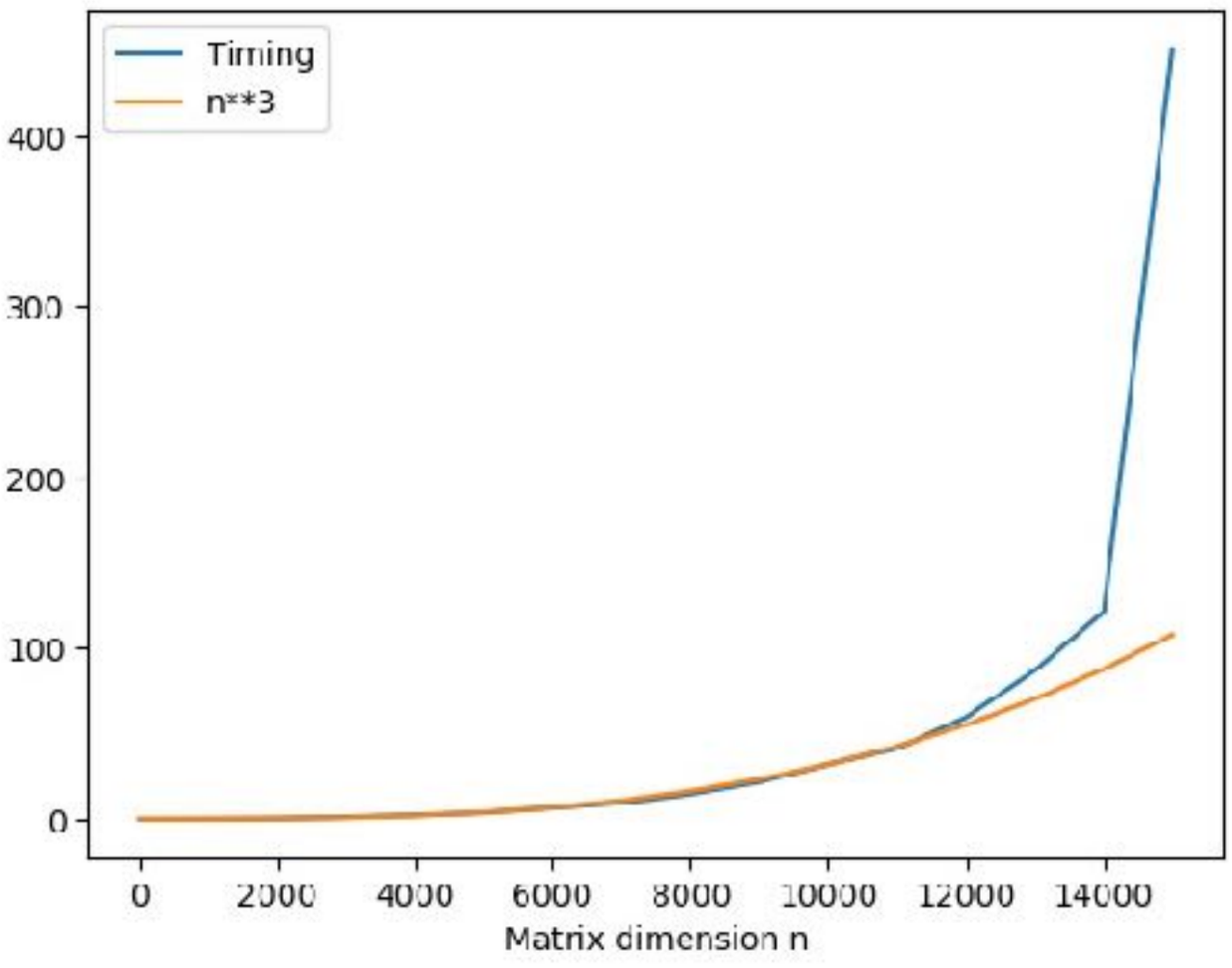
- Generate a random *nxn* matrix as well as a random *n*-dim vector
- Use something like scipy.linalg.solve() to solve the corresponding system of linear equations
- Check whether or not the algorithm used by the library scales like $\mathcal{O}(n^3)$
- Useful python functions:
- time.time()
- numpy.random.random()

EXERCISE 4: RESULTS



- Generate a random *nxn* matrix as well as a random *n*-dim vector
- Check whether or not the algorithm used by the library scales like $\mathcal{O}(n^3)$
- On the left, you can see a comparison for up to n=1500

EXERCISE 4: RESULTS



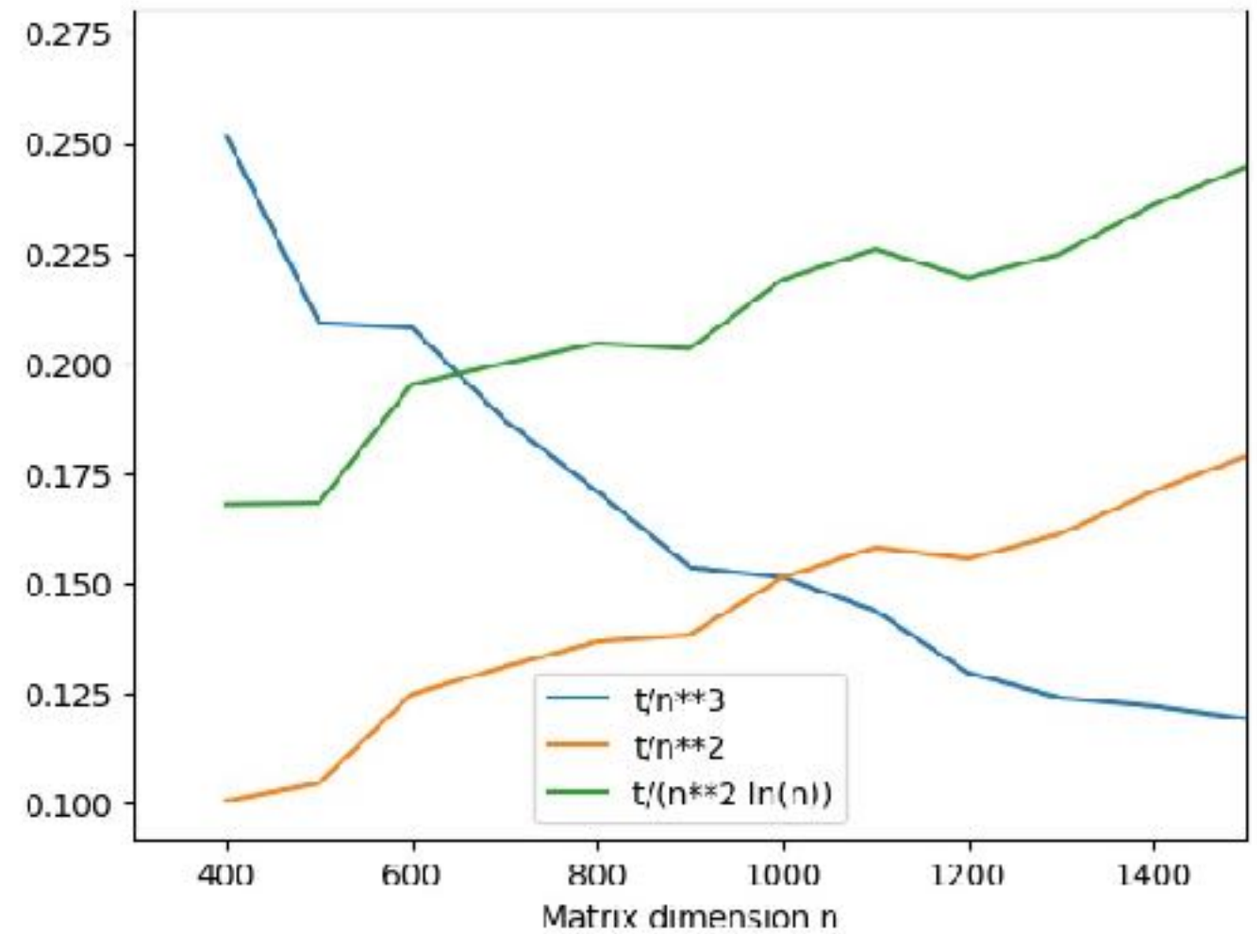
- Generate a random *nxn* matrix as well as a random *n*-dim vector
- Check whether or not the algorithm used by the library scales like $\mathcal{O}(n^3)$
- I pushed the algorithm up to n=15000
- Machine started to run into memory problems, which messes up the good performance …
- Something to keep in mind!

$$A \cdot \vec{x} = \lambda \vec{x}$$

EXERCISE 5

- Generate a random *nxn* matrix
- Use something like scipy.linalg.eig() to get the eigenvalues (and eigenvectors) of this matrix
- Try to figure out, how the computation time used by this library function scales with *n*
- Useful python functions:
- time.time()
- numpy.random.random()

EXERCISE 5: RESULTS

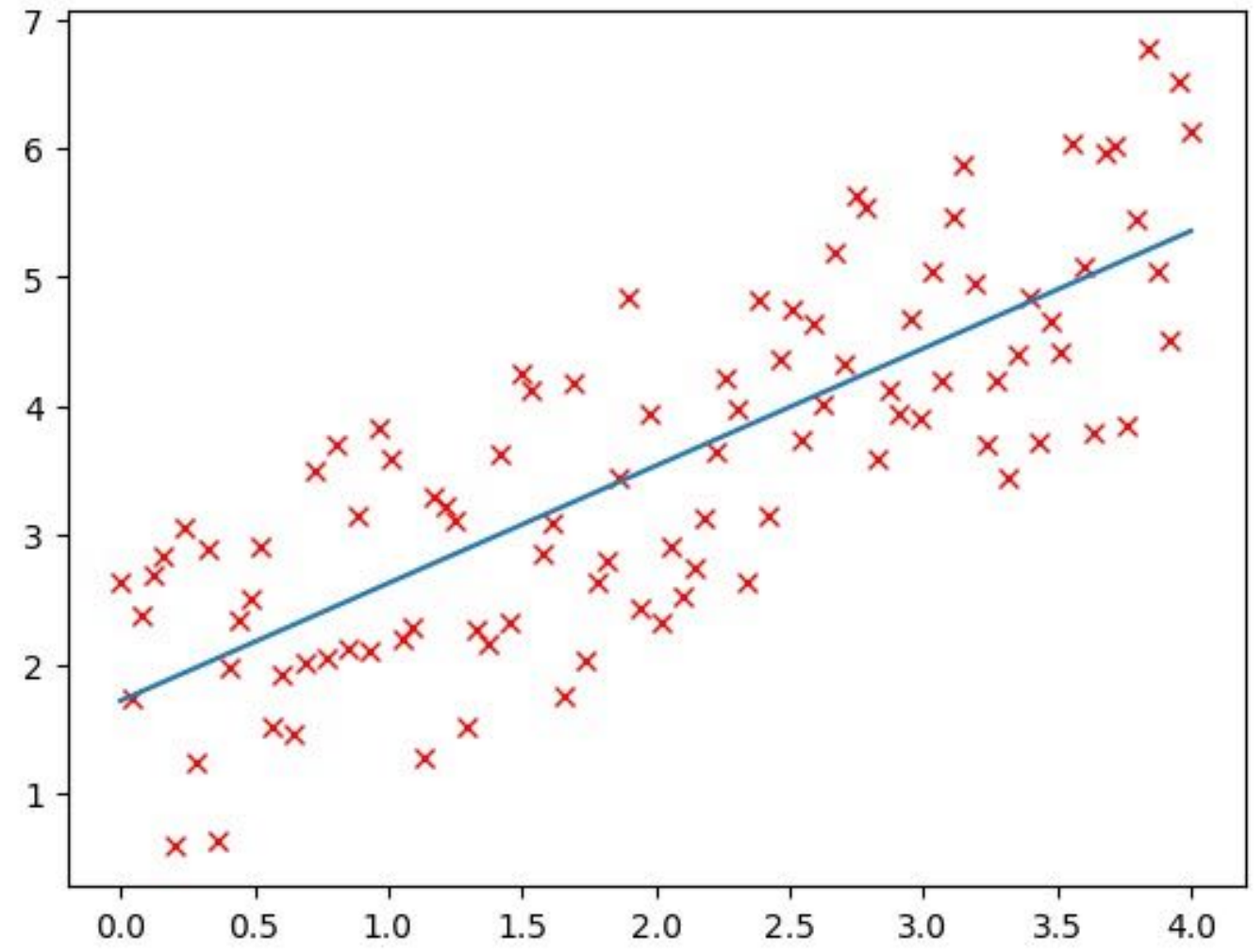


- Generate a random *nxn* matrix
- Use something like scipy.linalg.eig() to get the eigenvalues (and eigenvectors) of this matrix
- Looks like whatever this function uses goes like something in between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$

LINEAR REGRESSION

Direct vs. Iterative Methods

LINEAR REGRESSION



- This is an excellent example for direct vs. iterative methods
- So far, we have seen direct methods in action. Now, let's go compare these two approaches.
- Linear regression is a simple way to fit a (multi-)linear function to a set of data points
- The function is not necessarily linear in the available data, but in the coefficients that are assigned to them.
- What? Let's take a closer look …

LINEAR REGRESSION

- A system to be treated with linear regression has the following components:
  - a data set of *m* pieces of data, for which a linear model is to be found
  - a number *n* of independent variables *x* (in machine-learning speak, they are called *features*)
  - a resulting variable *y*
  - a number *n+1* of parameters a
  - a concept for how good a model prediction is (cost function)

LINEAR REGRESSION

- A system to be treated with linear regression has the following components:
  - a data set of *m* pieces of data, for which a linear model is to be found
  - a number *n* of independent variables *x* (in machine-learning speak, they are called *features*)
  - a resulting variable *y*
  - a number *n+1* of parameters a
  - a concept for how good a model prediction is (cost function)

$$\text{data point } i : \vec{x}^{(i)}, y^{(i)}$$

$$i = 1, \ldots, m$$

$$\vec{x}^{(i)} = (x_0^{(i)}, x_1^{(i)}, \ldots, x_n^{(i)})$$

$$x_0^{(i)} = 1 \; \forall i$$

$$\vec{a} = (a_0, a_1, \ldots, a_n)$$

$$\text{the model: } f(\vec{a}, \vec{x}) = \vec{a} \cdot \vec{x} = \sum_{j=0}^{n} a_j x_j$$

LINEAR REGRESSION

- The cost function quantifies the error, taking into account all available data points, e.g., a sum over all squared differences between model y and data y:

$$J(\vec{a}) = \frac{1}{2m}\sum_{i=1}^{m}(f(\vec{a},\vec{x}^{(i)}) - y^{(i)})^2 = \frac{1}{2m}\sum_{i=1}^{m}(\vec{a}\cdot\vec{x}^{(i)} - y^{(i)})^2$$

- In order to construct the best model, we need to minimize the cost.
- This can be done in different ways.

LINEAR REGRESSION

- To minimize the cost function, we'll use its gradient
- Consider J as a function of the vector of all *a*, then:

$$\frac{\partial J(\vec{a})}{\partial a_j} = \frac{1}{m}\sum_{i=1}^{m}(\vec{a}\cdot\vec{x}^{(i)} - y^{(i)})x_j^{(i)}$$

- Remember that this is also true for *i=0*, since

$$\frac{\partial J(\vec{a})}{\partial a_0} = \frac{1}{m}\sum_{i=1}^{m}(\vec{a}\cdot\vec{x}^{(i)} - y^{(i)})x_0^{(i)} = \frac{1}{m}\sum_{i=1}^{m}(\vec{a}\cdot\vec{x}^{(i)} - y^{(i)})$$

LINEAR REGRESSION

$$X = \begin{pmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{pmatrix} = \begin{pmatrix} (\vec{x}^{(1)})^T \\ \vdots \\ (\vec{x}^{(m)})^T \end{pmatrix}$$

$$\frac{\partial J(\vec{a})}{\partial a_j} = \frac{1}{m} \left[X^T (X\vec{a} - \vec{y}) \right]_j$$

- Now, in order to be efficient, let's write all that in matrix form, using all data points at once.
- Define the *design matrix* as given on the left.
- It is denoted by *X* and is a matrix of dimensions *m* x *(n+1)*
- Using this notation makes writing and computing the cost function's gradient easier
- The sum over *i* is now implicit in the matrix(-vector) multiplications

LINEAR REGRESSION

- In order to solve this directly, we can reformulate

$$\frac{\partial J(\vec{a})}{\partial a_j} = \frac{1}{m}\left[X^T(X\vec{a} - \vec{y})\right]_j \stackrel{!}{=} 0 \quad \forall j$$

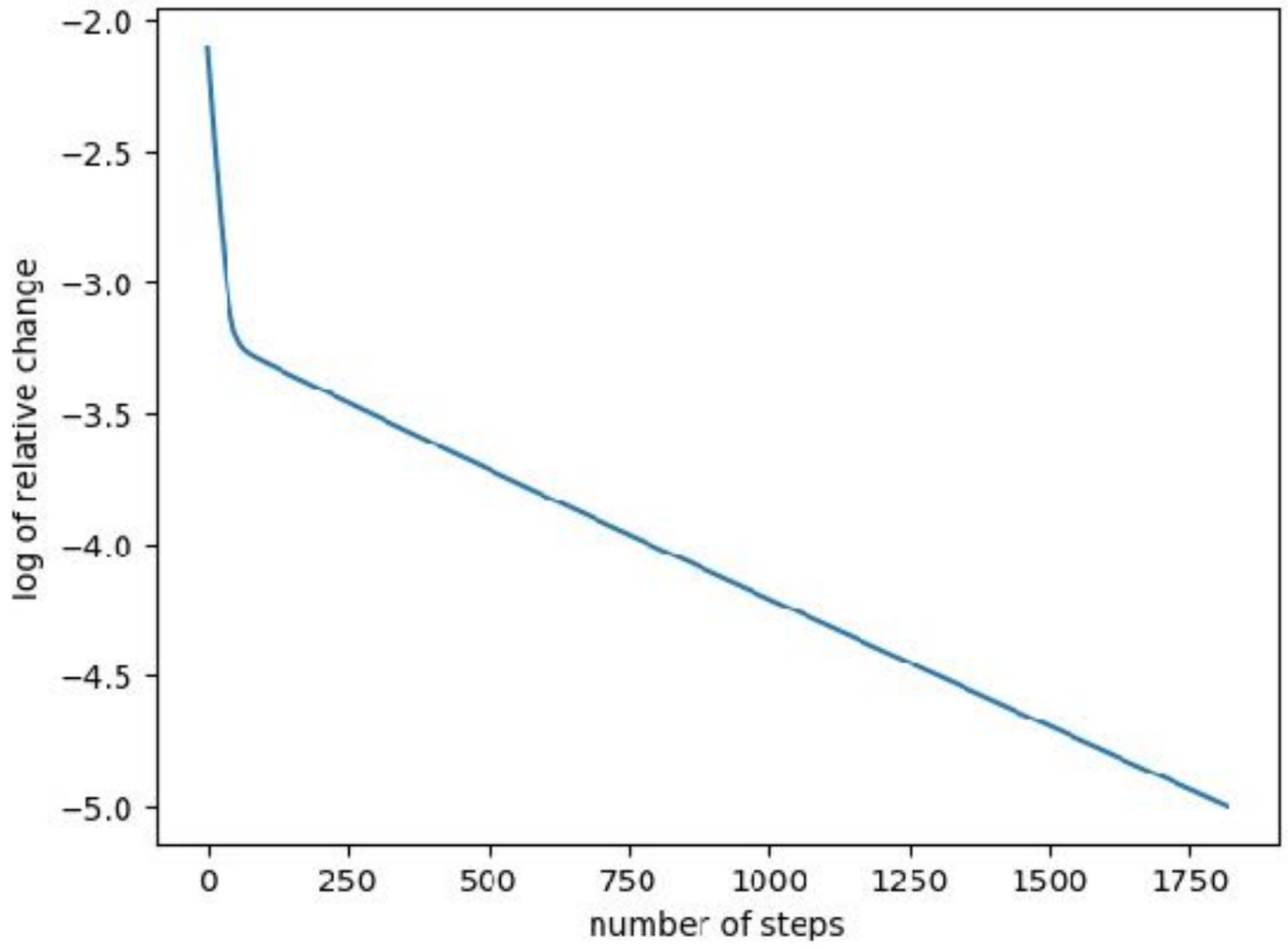
via

$$X^T X\vec{a} \stackrel{!}{=} X^T\vec{y}$$

to get the optimal values for the vector *a* as

$$\vec{a} = (X^T X)^{-1} X^T\vec{y}$$

LINEAR REGRESSION



- In order to solve this iteratively, we use the gradient-descent algorithm
- Start with an initial guess for the vector a and iterate the following condition:

$$\vec{a} \to \vec{a} - \frac{\alpha}{m} X^T (X\vec{a} - \vec{y})$$

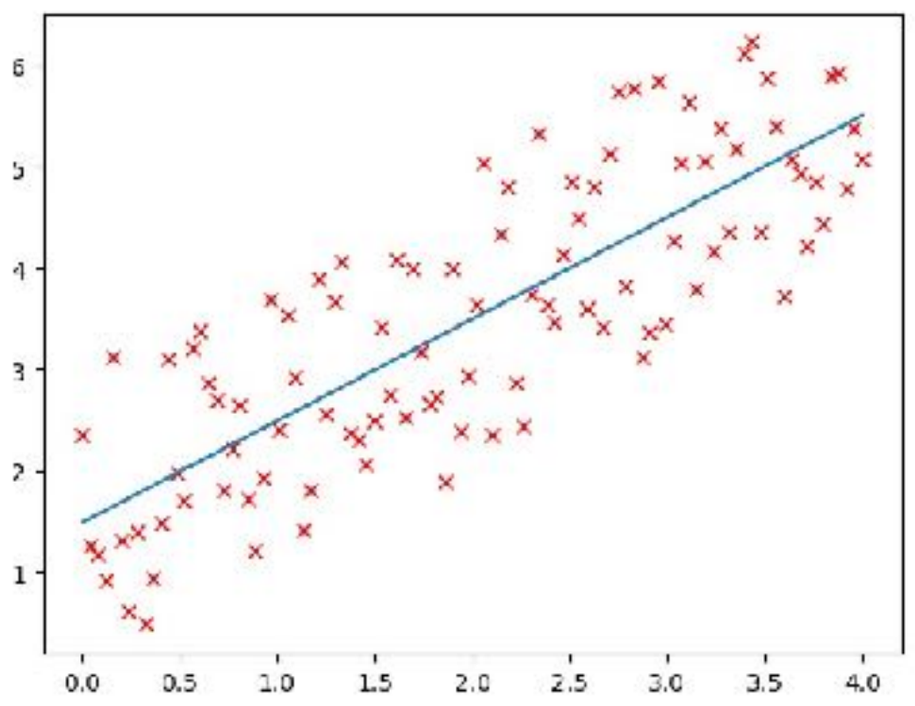
- α is called the learning rate (again in ML-speak) and is a hyper-parameter, i.e., we have to choose and test to find a good value.
- Watch relative change converge

EXERCISE 6A

- Use linear regression and the direct method $\vec{a} = (X^T X)^{-1} X^T \vec{y}$ to fit a linear function with one parameter ($n=1$) to the dataset

```
ndata = 100
data_x = np.linspace(0,4,ndata)
data_y = np.array(data_x + 3*np.random.random(size=(ndata)))
```

- Plot the data and the resulting linear function



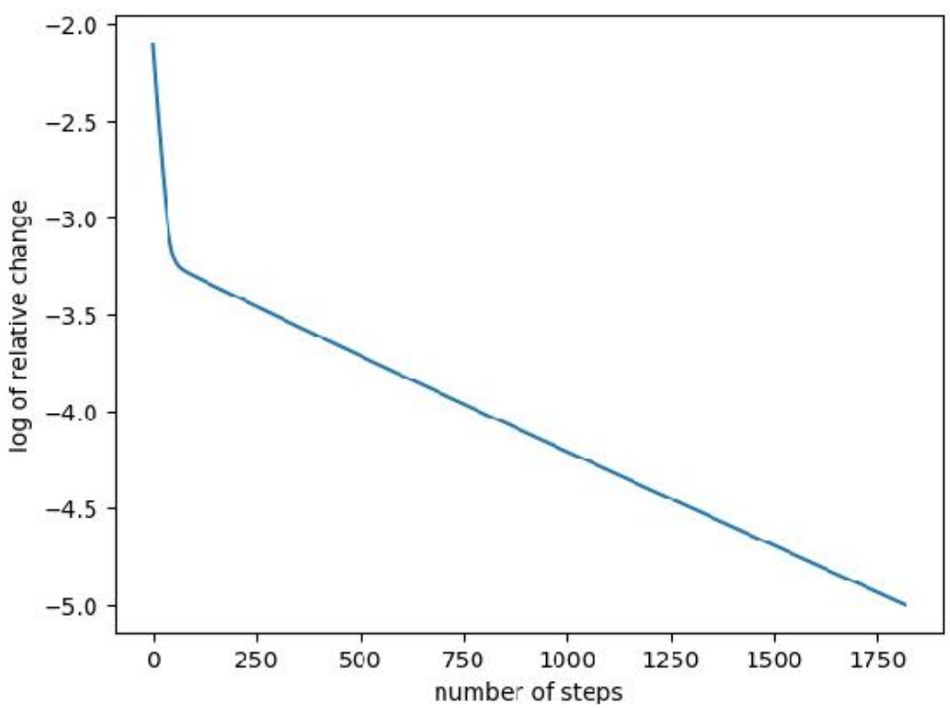
EXERCISE 6B

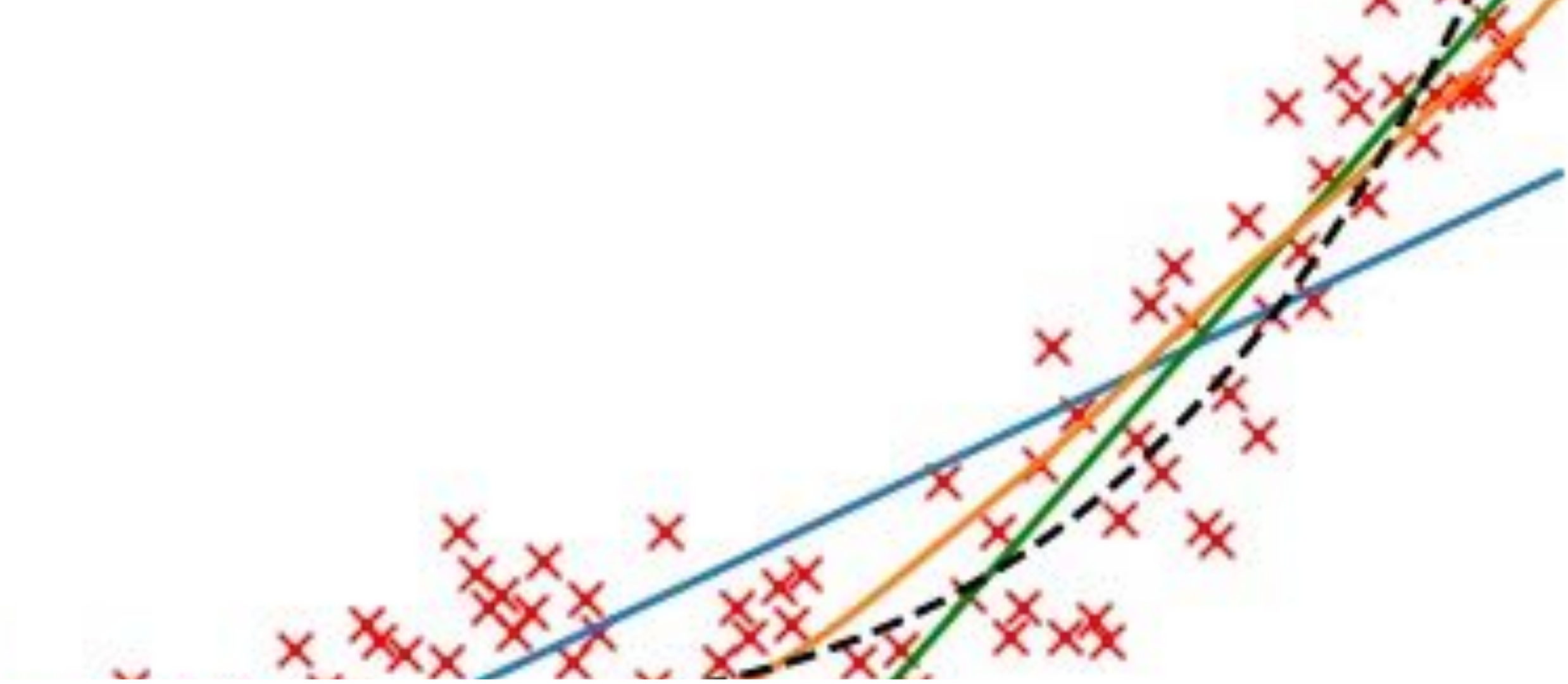
- Use linear regression and gradient descent to fit a linear function with one parameter (*n=1*) to the dataset

$$\vec{a} \to \vec{a} - \frac{\alpha}{m} X^T (X\vec{a} - \vec{y})$$

```
ndata = 100
data_x = np.linspace(0,4,ndata)
data_y = np.array(data_x + 3*np.random.random(size=(ndata)))
```

- Plot the logarithm of the relative change as a function of the number of iterations
- Compare the results from both methods
- Identify suitable values for the learning rate and the target precision of the iteration algorithm





EVEN MORE LINEAR REGRESSION

Direct vs. Iterative Methods Using Several Features

LINEAR REGRESSION

$$\text{data point } i : \vec{x}^{(i)}, y^{(i)}$$
$$i = 1, \ldots, m$$

$$\vec{x}^{(i)} = (x_0^{(i)}, x_1^{(i)}, \ldots, x_n^{(i)})$$
$$x_0^{(i)} = 1 \; \forall i$$

$$\vec{a} = (a_0, a_1, \ldots, a_n)$$

$$\text{the model: } f(\vec{a}, \vec{x}) = \vec{a} \cdot \vec{x} = \sum_{j=0}^{n} a_j x_j$$

- Look at LR again, but with several independent variables (features)
- The same as before: a data set of *m* pieces of data, for which a linear model is to be found
- actually $n>1$ independent variables x_i (in machine-learning speak, they are called *features*)
- again just one resulting variable *y*
- actually $n+1$ of parameters a_i
- The same cost function *J* as before

LINEAR REGRESSION

$$X = \begin{pmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{pmatrix} = \begin{pmatrix} (\vec{x}^{(1)})^T \\ \vdots \\ (\vec{x}^{(m)})^T \end{pmatrix}$$

$$\frac{\partial J(\vec{a})}{\partial a_j} = \frac{1}{m} \left[X^T (X\vec{a} - \vec{y}) \right]_j$$

- Define the *design matrix* as given on the left.
- It is denoted by *X* and is a matrix of dimensions *m* x *(n+1)*
- Keep in mind that the first column in *X* corresponds to any independent variable to the zeroth power
- The gradient has been written for linear dependence on the parameters in the vector *a*
- This form is more general for nonlinearities, but we'll deal with them another time

$$X = \begin{pmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{pmatrix} = \begin{pmatrix} (\vec{x}^{(1)})^T \\ \vdots \\ (\vec{x}^{(m)})^T \end{pmatrix}$$

$$\frac{\partial J(\vec{a})}{\partial a_j} = \frac{1}{m} \left[X^T (X\vec{a} - \vec{y}) \right]_j$$

FEATURE ENGINEERING

- Imagine starting with a single independent variable, like in our example, time *t*.
- If we have nothing else, we could add another column to *X* by using the values for t^2 in addition to just *t*.
- If we happen to know, which variable constructions make more sense than others, we can try and use them as well
- This is called *feature engineering*, i.e., creating data that represent useful *additional independent variables* in a *linear* combination

$$X = \begin{pmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{pmatrix} = \begin{pmatrix} (\vec{x}^{(1)})^T \\ \vdots \\ (\vec{x}^{(m)})^T \end{pmatrix}$$

$$x_i' = \frac{x_i - \bar{x}_i}{\sigma_{x_i}}$$

$$\bar{x}_i = \frac{1}{m} \sum_{j=1}^{m} x_i^{(j)}$$

$$\sigma_{x_i}^2 = \frac{1}{m} \sum_{j=1}^{m} (x_i^{(j)} - \bar{x}_i)^2$$

FEATURE SCALING

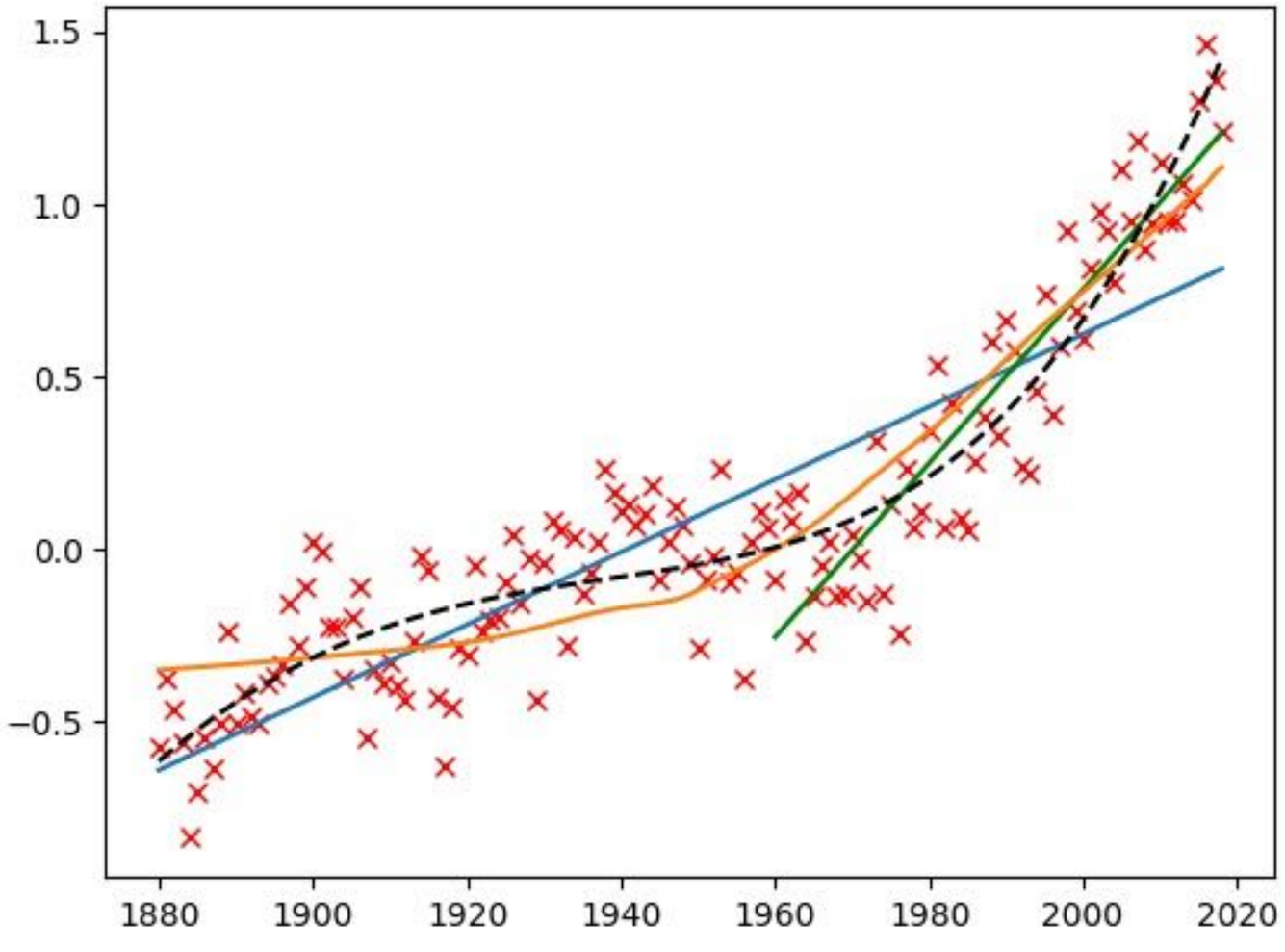
- Multiple features can have a totally different range of values
- This is problematic, since the largest values can dominate the algorithm
- To avoid this, and balance the contributions from different features, their values should be scaled.
- Use mean and standard deviation to scale and shift values for each feature separately.
- Keep track of means etc. to convert correctly (later on)

EXERCISE 7A

- Use linear regression and the direct method to fit a linear function with more than one parameter ($n>1$) to the dataset of annual temperature anomalies (see Email)

$$\vec{a} = (X^T X)^{-1} X^T \vec{y}$$

- Use provided code snippet to read in and prepare the data (also Email)
- Check out linear regression in one variable as a starting point
- Restrict the data to the time from 1960 onward
- Think about what to do with the independent data in order to create new features in addition to t (time/year at which temperatures were measured)
- Use feature scaling
- Plot the data and the resulting function
- Predict the anomalies in 2050 and 2100

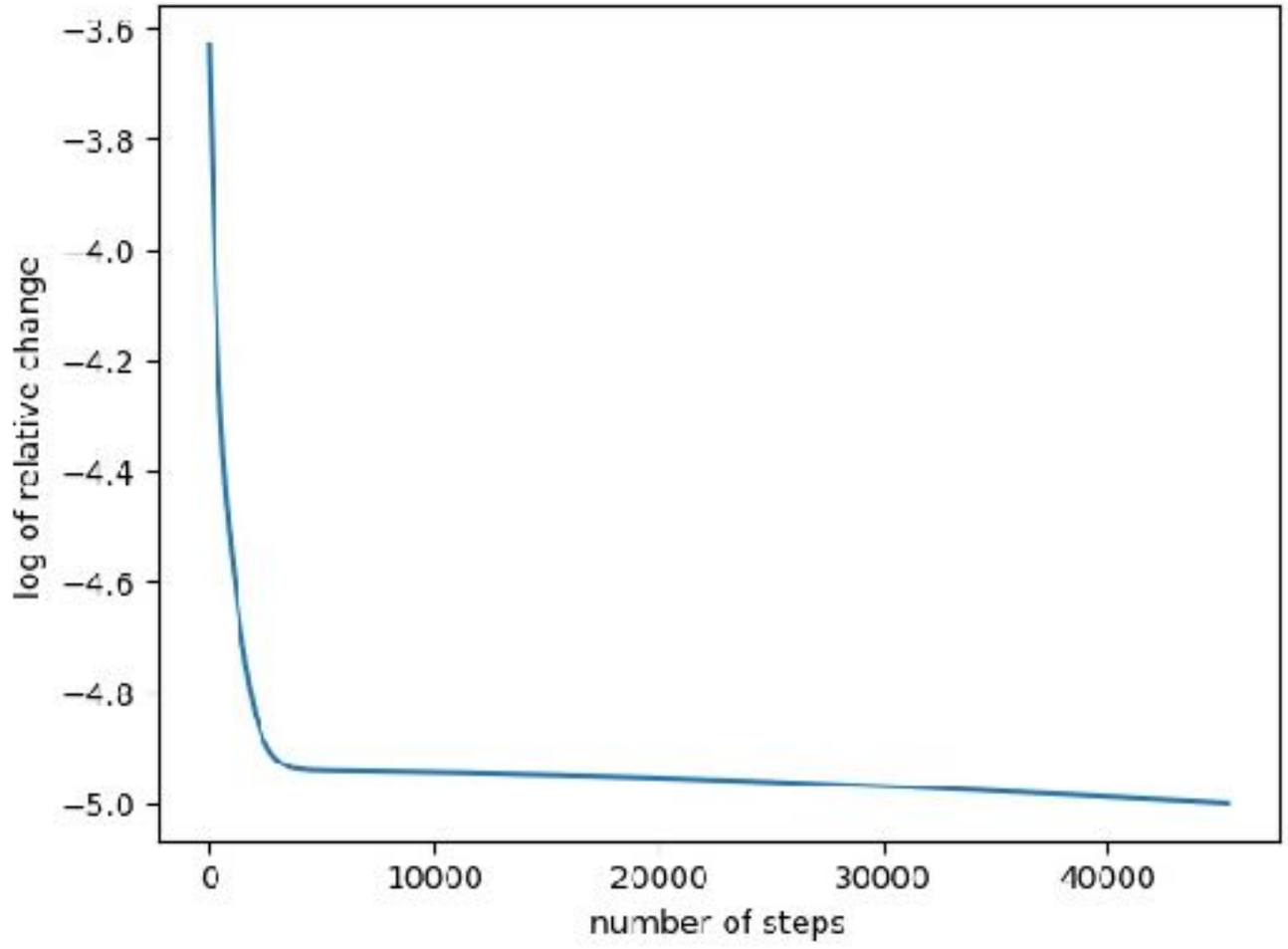


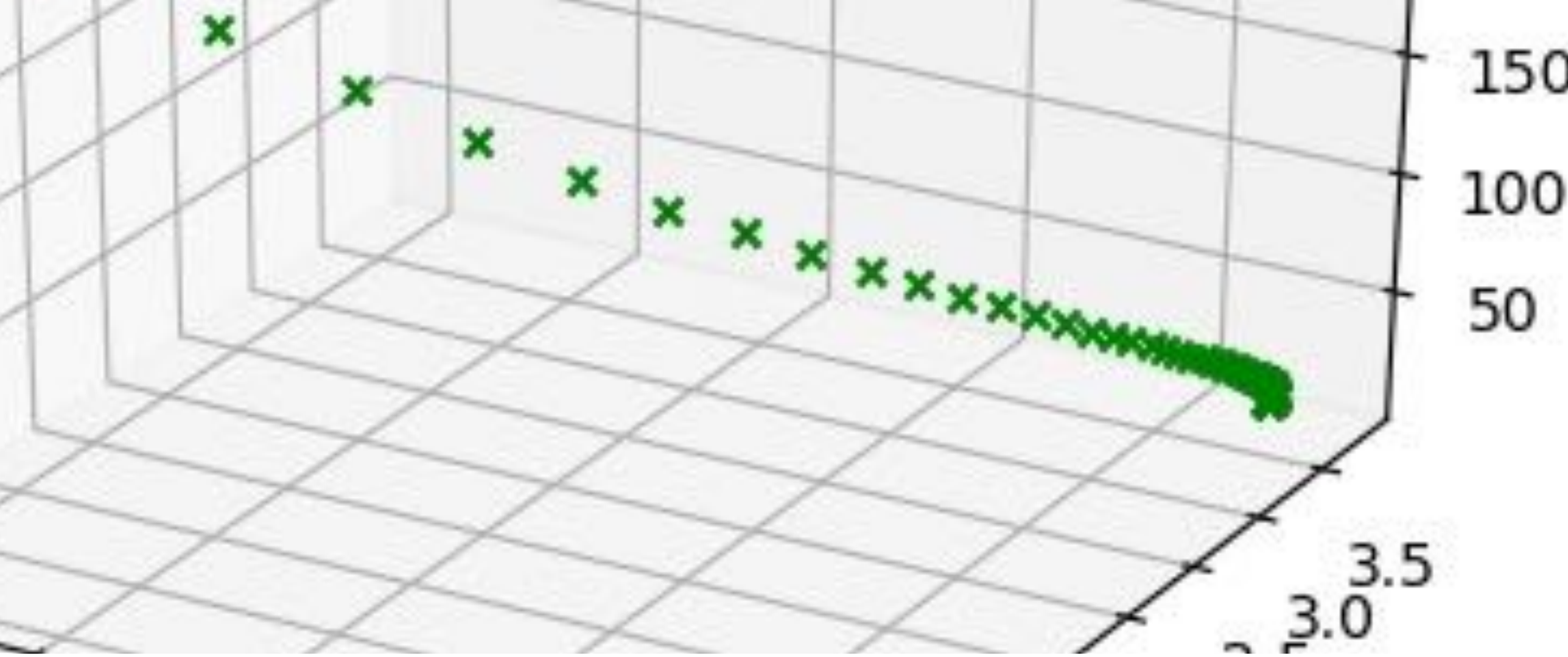
EXERCISE 7B

- Use linear regression and gradient descent to fit a linear function with more than one parameter ($n=1$) to the same dataset

$$\vec{a} \to \vec{a} - \frac{\alpha}{m} X^T (X\vec{a} - \vec{y})$$

- Try to get the algorithm to work (it won't, unless you use feature scaling)
- Plot the logarithm of the relative change as a function of the number of iterations
- Compare the results from both methods
- Identify suitable values for the learning rate and the target precision of the iteration algorithm
- Which problems appear?





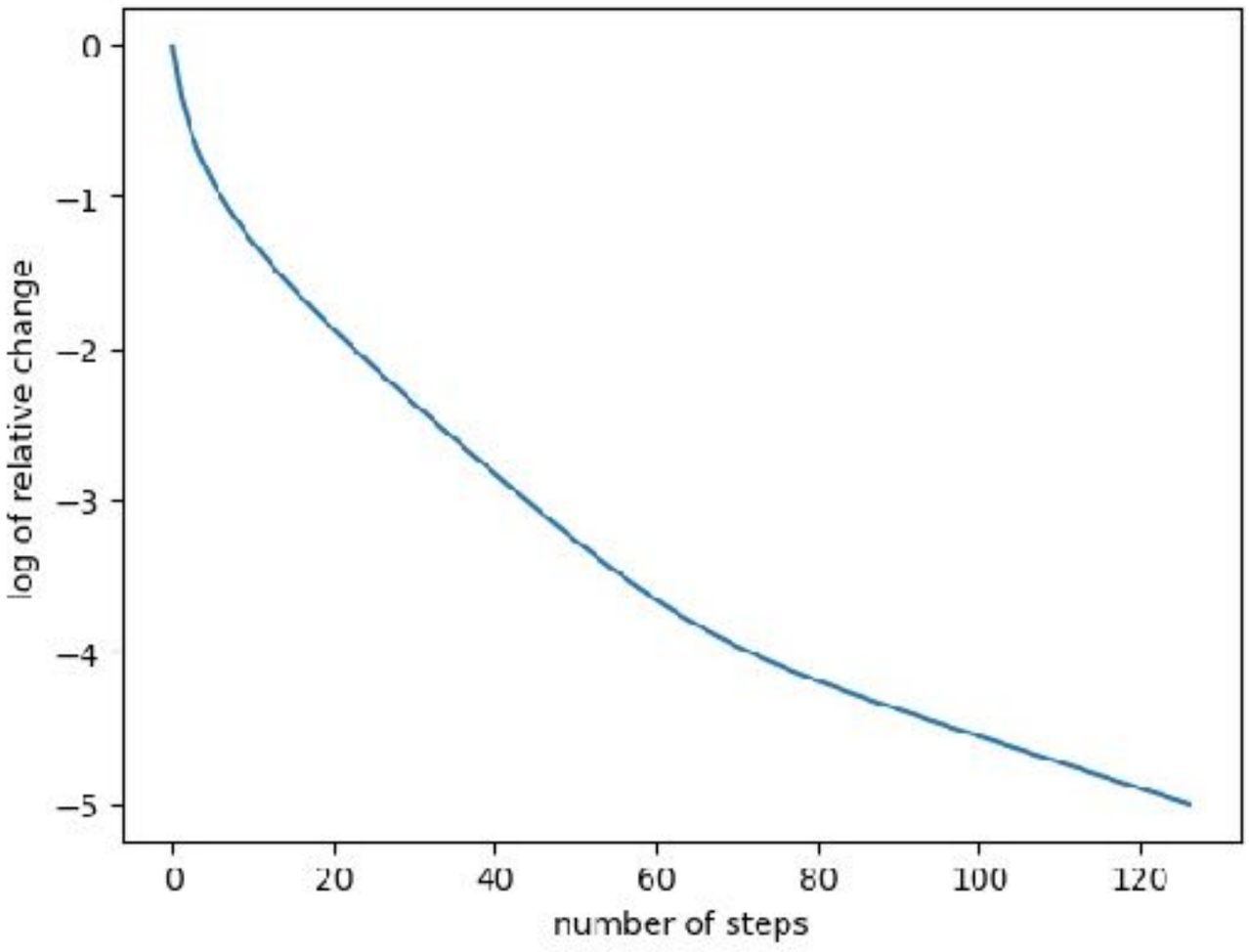
GENERAL OPTIMIZATION

Using Gradient Descent for any model

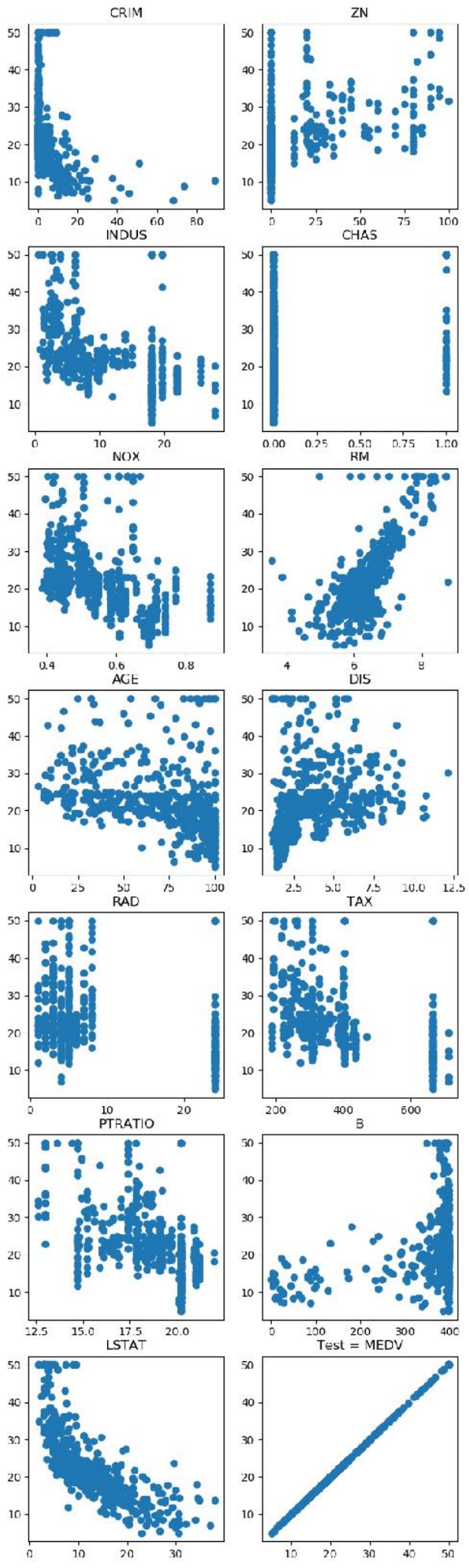
OPTIMIZING A MODEL

- So far, we have been dealing with linear models to describe our data
- In general, we'll need more than just linear coefficients
- Gradient Descent is an example for an iterative algorithm to find minima in any sort of optimisation problem
- Example: Famous dataset: Boston housing prices
- Can be used for all sorts of exercises and testing

OPTIMIZING A MODEL

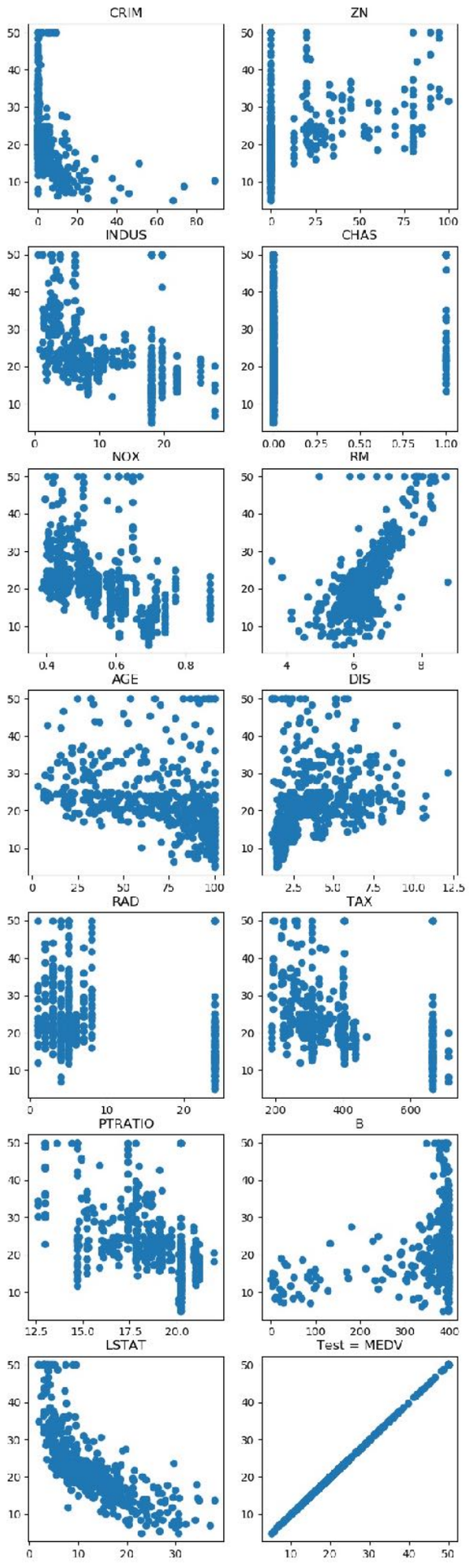


- This could be a real-world example
- Data set has ≈ 500 entries
- 13 independent variables
- one dependen variable: average price of housing
- Your task will be to:
  - Take the data and use it to build a model, however complicated you find useful
  - use as many of the variables as you like/need
  - Watch the cost (as before) decline



GETTING AN IDEA

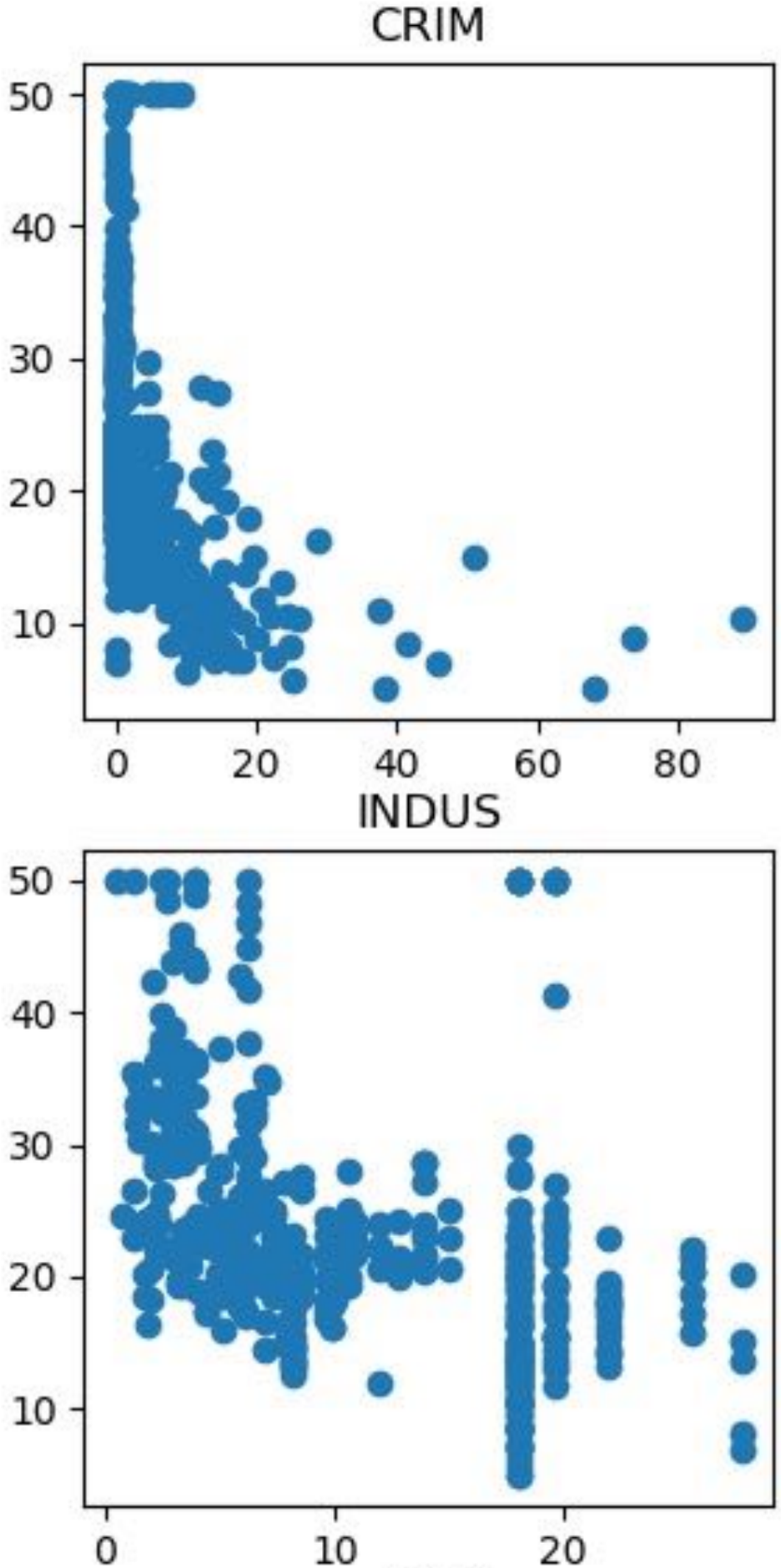
- Before we start, let's look at the data and some formulae.
- It is always a good idea to look at what might be useful and how.
- You received a piece of python code, which produces this figure for you together with the data.
- We'll check it out as well as discuss what is contained in the data set.
- Not all of this is totally useful, some of it marginally at best, but that is part of the exercise.



THE DATA SET

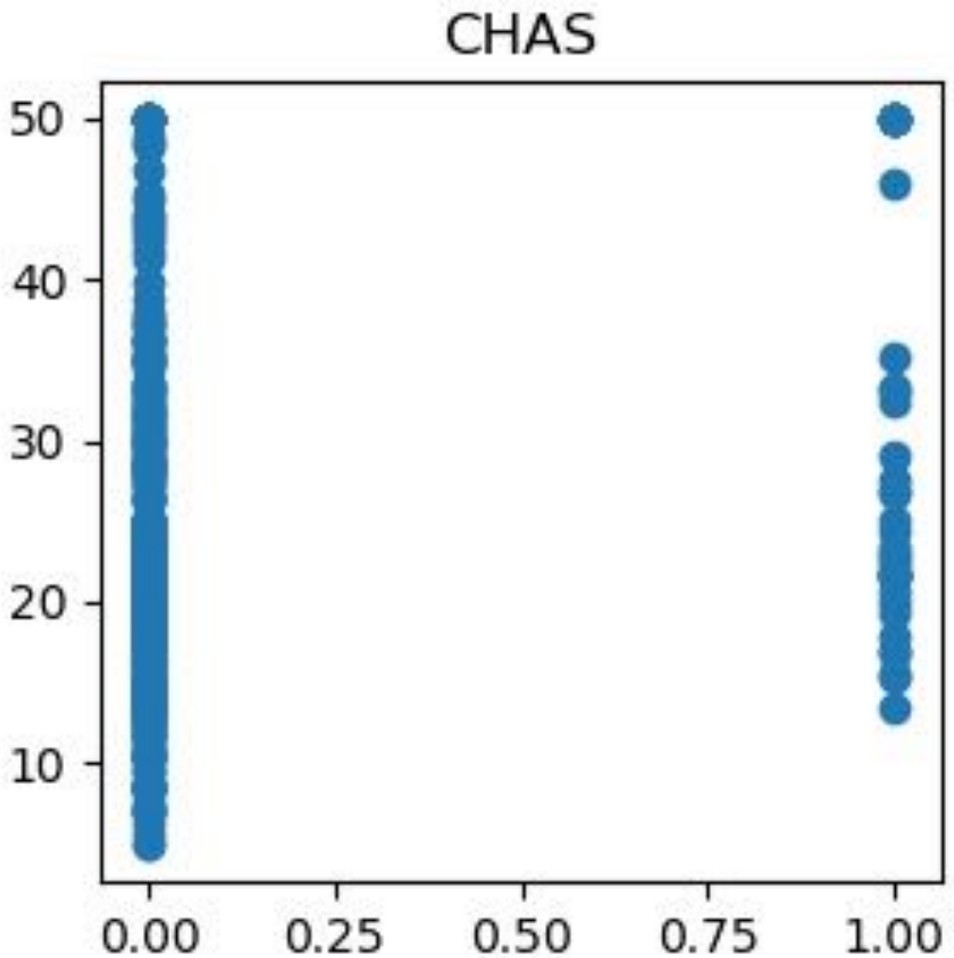
- Public data set, obtained via kaggle.com
- Each record in the database describes a Boston suburb or town. The data was drawn from the Boston Standard Metropolitan Statistical Area (SMSA) in 1970. The attributes are defined as follows (taken from the UCI Machine Learning Repository1):

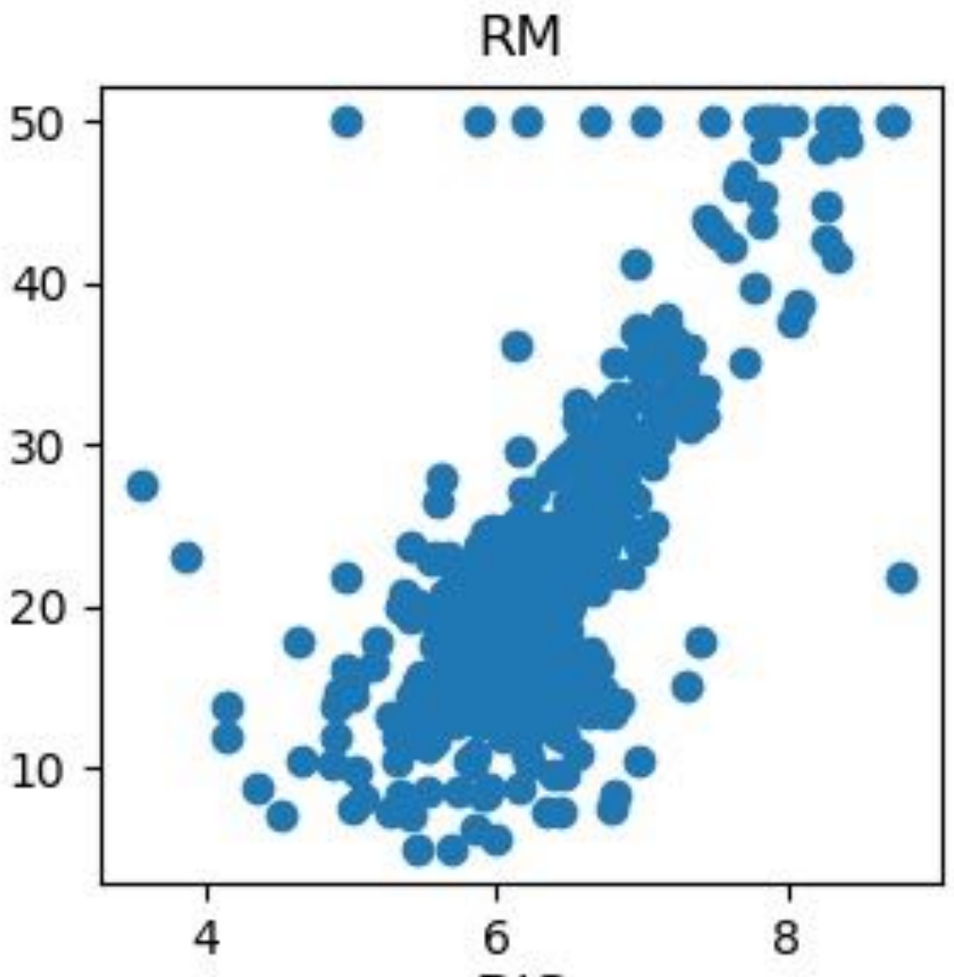
1. CRIM: per capita crime rate by town
2. ZN: proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS: proportion of non-retail business acres per town
4. CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
5. NOX: nitric oxides concentration (parts per 10 million) 1https://archive.ics.uci.edu/ml/datasets/Housing 123 20.2. Load the Dataset 124
6. RM: average number of rooms per dwelling
7. AGE: proportion of owner-occupied units built prior to 1940
8. DIS: weighted distances to five Boston employment centers
9. RAD: index of accessibility to radial highways
10. TAX: full-value property-tax rate per $10,000
11. PTRATIO: pupil-teacher ratio by town
12. B: 1000(Bk−0.63)2 where Bk is the proportion of blacks by town
13. LSTAT: % lower status of the population
14. MEDV: Median value of owner-occupied homes in $1000s



A FEW FEATURE EXAMPLES

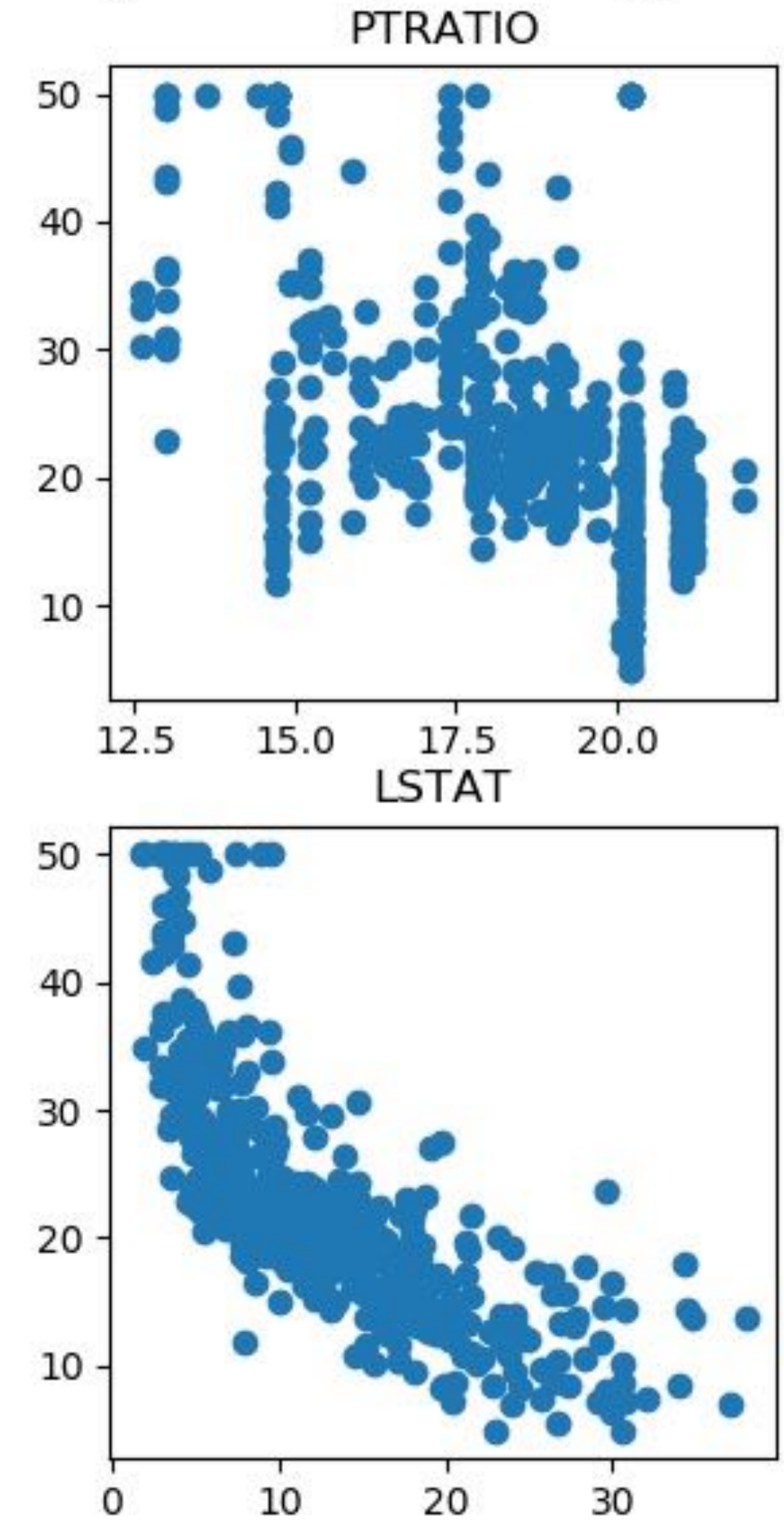
- Take a couple of features and investigate their potential usefulness
- In the figure, the feature title is above the graph
- CRIM: per capita crime rate by town
- INDUS: proportion of non-retail business acres per town





A FEW FEATURE EXAMPLES

- Take a couple of features and investigate their potential usefulness
- In the figure, the feature title is above the graph
- CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- RM: average number of rooms per dwelling



A FEW FEATURE EXAMPLES

- Take a couple of features and investigate their potential usefulness
- In the figure, the feature title is above the graph
- PTRATIO: pupil-teacher ratio by town
- LSTAT: % lower status of the population

$$\text{data point } i : \vec{x}^{(i)}, y^{(i)}$$

$$i = 1, \ldots, m$$

$$\vec{x}^{(i)} = (x_0^{(i)}, x_1^{(i)}, \ldots, x_n^{(i)})$$

$$x_0^{(i)} = 1 \; \forall i$$

$$\vec{a} = (a_0, a_1, \ldots, a_n)$$

$$J(\vec{a}) = \frac{1}{2m} \sum_{i=1}^{m} (f(\vec{a}, \vec{x}^{(i)}) - y^{(i)})^2$$

$$f(\vec{a}, \vec{x}) = \vec{a} \cdot \vec{x} = \sum_{j=0}^{n} a_j x_j$$

$$f(\vec{a}, \vec{x}) = ??$$

WHAT DO WE DO?

- The same as before: a data set of *m* pieces of data with *n>1* independent variables x_i (*features*)
- again just one resulting variable *y*
- fitting *n+1* parameters a_i
- the cost remains the same, but the model can be different
- we could use a linear model again
- or something completely different

$$J(\vec{a}) = \frac{1}{2m}\sum_{i=1}^{m}(f(\vec{a},\vec{x}^{(i)}) - y^{(i)})^2$$

$$\frac{\partial J(\vec{a})}{\partial a_j} = \frac{1}{m}\left[X^T(X\vec{a} - \vec{y})\right]_j$$

GRADIENT DESCENT

- The gradient of *J* with respect to *a* was easily writable in marix form for a linear model.
- In general, however, we need to remember what we started with, and compute the general gradient:

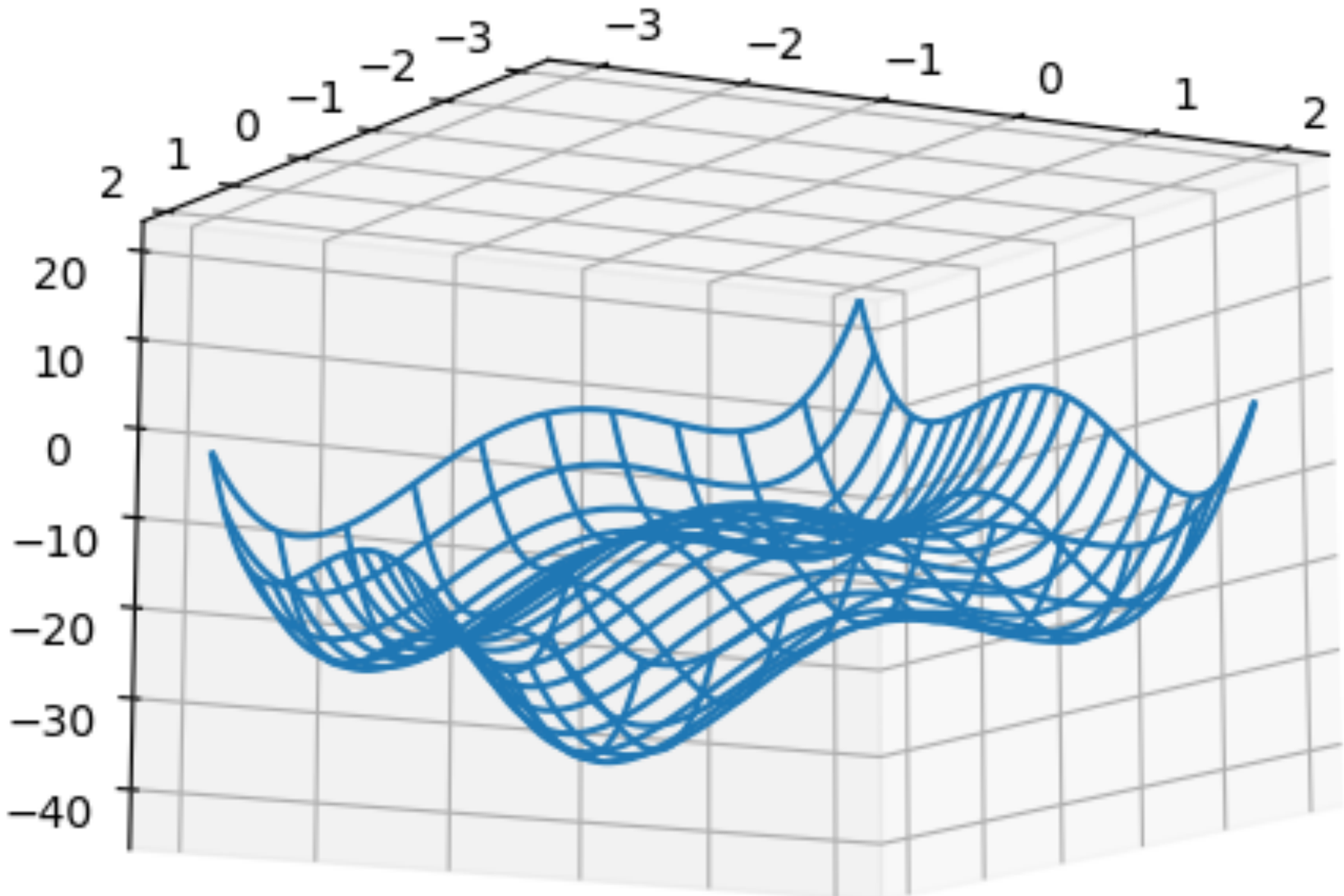
$$\frac{\partial J(\vec{a})}{\partial a_j} = \frac{1}{m}\sum_{i=1}^{m}\left[(f(\vec{a},\vec{x}^{(i)}) - y^{(i)})\frac{\partial f(\vec{a},\vec{x}^{(i)})}{\partial a_j}\right]$$

- The step for gradient descent remains the same

$$a_j \rightarrow a_j - \alpha\frac{\partial J(\vec{a})}{\partial a_j} \quad \forall a_j, j = 1,\ldots,n+1$$

GRADIENT DESCENT

- Running around with gradient descent might be adventurous

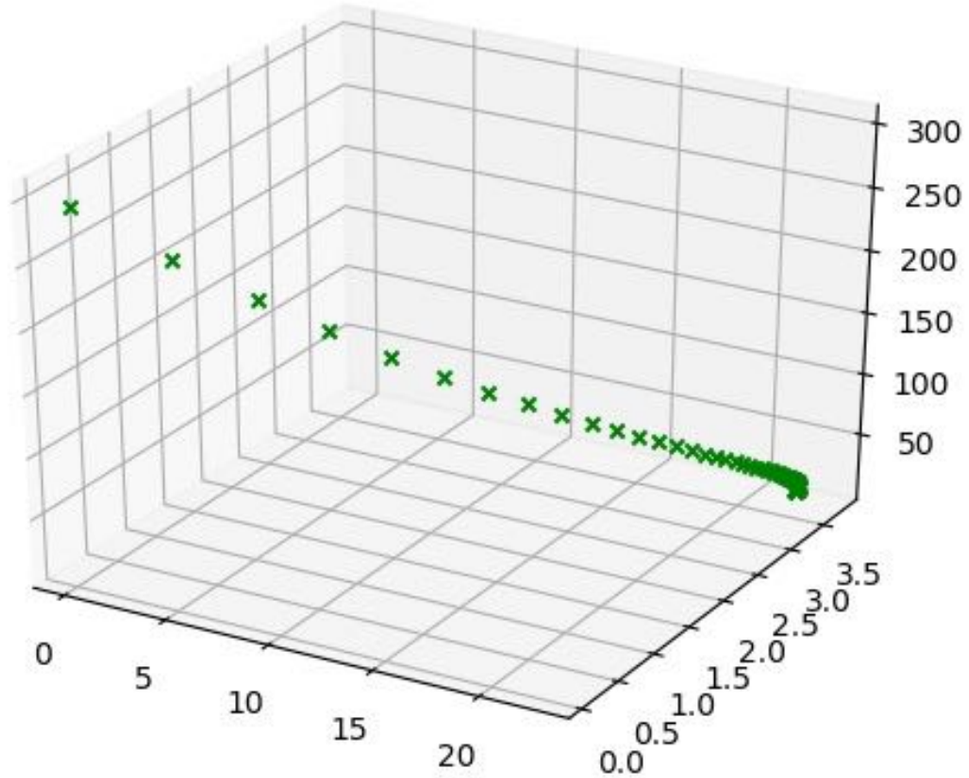


EXERCISE 8

- Use gradient descent to fit your choice of model to the housing prices data set

$$a_j \to a_j - \alpha \frac{\partial J(\vec{a})}{\partial a_j}$$

- Use feature scaling and any model function you like (may also be linear in the parameters)
- Plot the logarithm of the relative change as a function of the number of iterations
- Identify suitable values for the learning rate and the target precision of the iteration algorithm
- Which problems appear?
- No need to make 3D- or higher-dimensional plots …



$$\lambda\left(P^2\right)\vec{F}_{[h]}\left(P^2\right) = \mathbf{K}\left(P^2\right) \cdot \vec{F}_{[h]}\left(P^2\right)$$

EIGENVALUE PROBLEMS

Computing Eigenvalues of Matrices

EIGENVALUES, REMINDER

$$A\ x = \lambda\ x$$

$$A\ x = \lambda\ \mathbf{1}\ x$$

$$(A - \lambda\ \mathbf{1})\ x = 0$$

- We are used to seeing eigenvalue equations, where A is a matrix, x is a vector, and *lambda* is the eigenvalue. Let's not forget: $\mathbf{1}$ is the identity matrix.
- In general it is of interest to compute both the eigenvalues and the corresponding eigenvectors
- For simple systems (like 2x2 matrices), it is straightforward to do this
- For example, calculate the characteristic polynomial and find its solutions. Why?

EIGENVALUES, REMINDER

$$(A - \lambda\,\mathbf{1})\,x = 0$$

$$(\lambda\,\mathbf{1} - A)\,x = 0$$

$$\mathrm{Det}(A - \lambda\,\mathbf{1}) = 0$$

$$\mathrm{Det}(\lambda\,\mathbf{1} - A) = 0$$

$$\mathrm{Det}((\lambda_i - a_{ii})\,\mathbf{1}) = 0$$

$$\prod_i (\lambda_i - a_{ii}) = 0$$

- Why do we calculate the characteristic polynomial in order to find eigenvalues?
- This equation is a system of linear equations to be solved.
- A solution exists for a zero determinant of the matrix multiplying x.
- Now, it is really easy to see, why/how the eigenvalues λ_i of a diagonal matrix are the matrix elements a_{ii} in the diagonal:
- We get a characteristic poly. that is already factorised

EIGENVALUES, REMINDER

$$(\lambda\,\mathbf{1} - A)\,x = 0$$

$$\mathrm{Det}(\lambda\,\mathbf{1} - A) = 0$$

$$f(\lambda_i) = 0$$

- However, it isn't always that simple
- In general, one has to compute the determinant and the polynomial
- Then, take the actual polynomial and search for numerical values of the eigenvalues λ_i
- In principle, we already know a technique to do this: Newton's method for finding zeros of a function
- However, we'll go fo something more general

EIGENSYSTEM BY ITERATION

Eigenvalues: $\lambda^{(i)}$

Eigenvectors: $\vec{x}^{(i)}$

$$\vec{x}_{j+1} = A\,\vec{x}_j$$

$$\text{and } \vec{x}_{j+1} \to \frac{\vec{x}_{j+1}}{x_{0,j+1}}$$

with $x_{0,j+1}$ the first element of $\vec{x}_{j+1}$

$$\vec{x}_j \to \vec{x}^{(0)}$$

$$x_{0,j} \to \lambda^{(0)}$$

- Let's look at a simple iterative way to find the eigenvalue-eigenvector pairs, one by one, starting from the largest in terms of absolute value
- Take a real symmetric matrix
- Choose a test vector x_0 and multiply it by the matrix A repeatedly.
- This procedure makes the successors of x_0 in the iteration converge to the eigenvector corresponding to the largest eigenvalue
- The eigenvalue is a normalisation factor appearing in the procedure.
- What? How? Why?

EIGENSYSTEM BY ITERATION

$$\vec{x}_0 = \sum_j C_j \vec{x}^{(j)}$$

$$A\vec{x}_0 = \sum_j C_j A\ \vec{x}^{(j)}$$

$$A^n\vec{x}_0 = \sum_j C_j A^n\ \vec{x}^{(j)}$$

$$A^n\vec{x}_0 = \sum_j C_j (\lambda^{(j)})^n\ \vec{x}^{(j)}$$

$$\frac{1}{(\lambda^{(0)})^n} A^n\vec{x}_0 = \sum_j C_j \frac{(\lambda^{(j)})^n}{(\lambda^{(0)})^n}\ \vec{x}^{(j)}$$

- Expand the test vector in a sum of the eigenvectors
- Then, multiply the testmatrix A on this vector
- Remember, we are doing this over and over again, *n* times
- Then, remember that we are expanding in eigenvectors of A
- We remain with a sum over eigenvalues to some power
- Now, divide out the largest eigenvalue, and that's it
- With high enough *n*, only the largest eigenvalue term survives

EXERCISE 9A

- Use a real symmetric random matrix to test the iterative algorithm just described. Your starting point is:

```
matdim = 20
a = np.random.random(size=(matdim,matdim))-1
testmatrix = a + np.transpose(a)
```

- iterate the testmatrix on a random vector
- find the largest (in terms of absolute value) eigenvalue and the corresponding eigenvector
- observe the convergence and find out how many iterations you need
- compare your result to the one produced by a standard library, e.g. scipy.linalg.eig()

EIGENSYSTEM BY ITERATION

- Can we modify/extend this somehow to get more than one eigenvalue?
- Yes, by orthogonalisation onto all of the already known eigenvectors
- Use a tactic analogous to Gram-Schmidt procedure
- Subtract components in the directions of all known eigenvectors at each iteration step

$$\vec{x}^{(i)}_{j+1} = A\ \vec{x}^{(i)}_{j}$$

$$\vec{x}^{(i)}_{j+1} \rightarrow \frac{1}{x^{(j)}_{0,j+1}} \left(\vec{x}_{j+1} - \sum_{k<i} \vec{x}^{(k)} \left[\vec{x}^{(i)}_{j+1} \cdot \vec{x}^{(k)} \right] \right)$$

EXERCISE 9B

- Continue our investigation of a real symmetric random matrix to test the additional algorithm just described. Your starting point is again:

```
matdim = 20
a = np.random.random(size=(matdim,matdim))-1
testmatrix = a + np.transpose(a)
```

- iterate the testmatrix on a random vector
- find the largest couple of (in terms of absolute value) eigenvalues and the corresponding eigenvectors
- observe the convergence and find out how many iterations you need for the largest, next, next-to-next, etc.
- compare your result to the one produced by a standard library, e.g. scipy.linalg.eig()

EIGENVALUES: JACOBI METHOD

$$(\lambda \, \mathbf{1} - A) \, x = 0$$

$$A = (a_{ij})$$

$$A = \begin{pmatrix} a_{11} & \dots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \dots & a_{NN} \end{pmatrix}$$

- Another iterative method
- Real, symmetric matrix
- Uses many subsequent rotation matrices to
  - make diagonal elements larger
  - off-diagonal elements smaller
- This happens one element at a time
- Pick one (large) off-diagonal element
- Set that to zero by a suitable rotation

EIGENVALUES: JACOBI METHOD

$$A = \begin{pmatrix} a_{11} & \dots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \dots & a_{NN} \end{pmatrix}$$

- Say, the off-diagonal element is in row j and column i (and mirrored, since A is symmetric)
- Then, look at this particular part of A:

$$A = \begin{pmatrix} a_{11} & \dots & \dots & \dots & \dots & \dots & a_{1N} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & a_{jj} & \vdots & a_{ji} & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \vdots & a_{ij} & \vdots & a_{ii} & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{N1} & \dots & \dots & \dots & \dots & \dots & \dots a_{NN} \end{pmatrix}$$

EIGENVALUES: JACOBI METHOD

$$A' = R\ A\ R^T$$

- Now, imagine, we rotate exactly this part of A
- Use a Givens rotation matrix R:

$$R = \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \cos\vartheta & 0 & -\sin\vartheta & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & \sin\vartheta & 0 & \cos\vartheta & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \end{pmatrix}$$

EIGENVALUES: JACOBI METHOD

$$A' = R\ A\ R^T$$

$$\cos\vartheta =: c, \quad \sin\vartheta =: s$$

$$a'_{kl} = a_{kl} \quad \text{for} \quad k,l \neq i,j$$

$$a'_{jk} = c\ a_{jk} - s\ a_{ki} \quad k \neq i,j$$

$$a'_{ik} = s\ a_{ik} + c\ a_{kj} \quad k \neq i,j$$

$$a'_{jj} = c^2\ a_{jj} - 2sc\ a_{ji} + s^2\ a_{ii}$$

$$a'_{ii} = c^2\ a_{ii} - 2sc\ a_{ji} + s^2\ a_{jj}$$

$$a'_{ij} = (c^2 - s^2)\ a_{ij} + sc\ (a_{jj} - a_{ii})$$

$$a'_{ij} = (c^2 - s^2)\ a_{ij} + sc\ (a_{jj} - a_{ii}) = 0$$

- This product of three matrices can be calculated by hand straightforwardly
- a limited number of matrix elements change
- Only a few substantially
- We want one matrix element in particular to vanish after the rotation, namely a particular a_{ij} of our choosing
- to maximize the effect of the procedure, pick the a_{ij} with the largest absolute value (pivot)

EIGENVALUES: JACOBI METHOD

$$a'_{ij} = (c^2 - s^2)\, a_{ij} + sc\, (a_{jj} - a_{ii}) = 0$$

$$\vartheta = \frac{1}{2} \arctan \frac{2a_{ij}}{a_{ii} - a_{jj}}$$

$$\text{for } a_{ii} = a_{jj} : \ \vartheta = \frac{\pi}{4}$$

- We want one matrix element in particular to vanish after the rotation, namely a particular a_{ij} of our choosing
- to maximize the effect of the procedure, pick the a_{ij} with the largest absolute value (pivot)
- This way, we determine the rotation angle and use R on A.
- Then, pick the next largest, determine rotation angle, use corresponding R again, etc.
- Repeat until satisfied

EXERCISE 10

- Check the necessary formulae of he Jacobi eigenvalue method
- Implement the Jacobi method for getting the eigenvalues of a real symmeric matrix. Your starting point is again:

```
matdim = 20    # increase at will
a = np.random.random(size=(matdim,matdim))-1
testmatrix = a + np.transpose(a)
```

- think about how to cleverly find the pivot
- think about how to define convergence
- compute the eigenvalues for the test matrix
- observe the convergence and find out how many iterations you need
- benchmark and compare your result with the help of a standard library, e.g. scipy.linalg.eig()

$$\Delta\Phi(\vec{x}) = \varrho(\vec{x})$$

DIFFERENTIAL EQUATIONS

... and their numerical solution

$$\Delta\Phi(\vec{x}) = \varrho(\vec{x})$$

WHY DIFFERENTIAL EQUATIONS?

- Differential equations can be solved approximately by using numerical methods
- Various kinds, various methods
- Similarity: Space/time/coordinate is discretised
- Derivatives evaluated approximately
- E.g., as finite differences
- This yields matrix equations in terms of $\Phi(\vec{x}),\ \varrho(\vec{x})$ in our example of the Poisson equation
- Solution via linear algebra

DIFFERENTIAL OPERATORS

$$\Delta\Phi(\vec{x}) = \varrho(\vec{x})$$

- How can we approximate the Laplacian operator by finite differences?
- Take one derivative in one variable first

$$f'(x) = \frac{df}{dx} \approx \frac{f(x+h) - f(x)}{h} \quad \text{for small } h$$

- Then, the second derivative in one variable

$$f''(x) = \frac{d^2 f}{dx^2} \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \quad \text{for small } h$$

- Then, we can consider more than one dimension

$$\Delta\Phi(\vec{x}) = \varrho(\vec{x})$$

DIFFERENTIAL OPERATORS

- The Laplacian operator in two dimensions is

$$\Delta f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

$$\approx \frac{f(x+h_x,y) + f(x,y+h_y) - 4f(x,y) + f(x-h_x,y) + f(x,y-h_y)}{h^2}$$

and, in terms of grid points counted by the indices i and j, we get

$$(\Delta f)_{ij} \approx \frac{f_{i+1,j} + f_{i,j+1} - 4f_{i,j} + f_{i-1,j} + f_{i,j-1}}{h^2}$$

THE POISSON EQUATION

$$\Delta\Phi(\vec{x}) = \varrho(\vec{x})$$

- With this, the Poisson equation can be discretised to get

$$\frac{f_{i+1,j} + f_{i,j+1} - 4f_{i,j} + f_{i-1,j} + f_{i,j-1}}{h^2} = \varrho_{ij}$$

- We rewrite this as follows:

$$4f_{ij} = f_{i+1,j} + f_{i,j+1} + f_{i-1,j} + f_{i,j-1} - h^2\varrho_{ij}$$

- Finally, we write the iterable equation

$$f_{ij} = \frac{1}{4}(f_{i+1,j} + f_{i,j+1} + f_{i-1,j} + f_{i,j-1}) - \frac{h^2}{4}\varrho_{ij}$$

THE POISSON EQUATION

$$\Delta\Phi(\vec{x}) = \varrho(\vec{x})$$

- Solve this by iteration:

$$f_{ij}^{(k+1)} \to \frac{1}{4}(f_{i+1,j}^{(k)} + f_{i,j+1}^{(k)} + f_{i-1,j}^{(k)} + f_{i,j-1}^{(k)}) - \frac{h^2}{4}\varrho_{ij}$$

- Start with an (educated) guess
- Iterate each point
- Define boundary conditions for *f* and leave them untouched
- Define iteration as a matrix-vector multiplication that takes boundaries into account correctly
- Unroll the 2D-*f* into a vector
- Determine the correct matrix coefficients

EXERCISE 11

$$\Delta\Phi(\vec{x}) = \varrho(\vec{x})$$

- Solve this by iteration:

$$f_{ij}^{(k+1)} \to \frac{1}{4}(f_{i+1,j}^{(k)} + f_{i,j+1}^{(k)} + f_{i-1,j}^{(k)} + f_{i,j-1}^{(k)}) - \frac{h^2}{4}\varrho_{ij}$$

- Use the matrix form

$$\vec{f}^{(k+1)} \to A\ \vec{f}^{(k)} - \frac{h^2}{4}\vec{\varrho}$$

- Determine the coefficients in A for the Poisson equation
- Pick a density and boundary conditions
- Visualise the solution