UiO : Department of Informatics
University of Oslo

Exercises for IN1900

Cecilie Glittum and Simen Moe

September 21, 2018

Preface

This document contains a number of programming exercises made for the course IN1900. The chapter numbers and titles correspond to the chapters of the book "A primer on Scientific Programming with Python" by Hans Petter Langtangen. The exercises are meant to be a supplement to the exercise collection in the book, and most are motivated by applications in science and applied mathematics. The exercise collection is used for the first time in 2018, and there may be typos and small errors. If you find any errors, or have other comments or questions about the exercises, please send them to Joakim Sundnes: *sundnes@simula.no*.

Chapter 1

Computing with Formulas

Problem 1.1. Throw a ball
When throwing a ball in the air, it has a constant acceleration $-g$ from gravity (we're excluding air resistance). The height of the ball relative to its starting point is

$$y(t) = v_0 t - \frac{1}{2} g t^2,$$

where v_0 is the initial velocity of the ball and t is the time after the throw. The ball reaches its maximum height at time

$$t_{\max} = \frac{v_0}{g}.$$

Write a program computing the maximum height of the ball, that is $y(t_{\max})$, when $v_0 = 8.2\text{m/s}$ and $g = 9.81\text{m/s}^2$. Print the result.

Filename: `ball.py`

Problem 1.2. Population growth
The growth of a population can often be described by a logistic function

$$N(t) = \frac{B}{1 + C\mathrm{e}^{-kt}},$$

where B is the carrying capacity of the species in the enviroment, i.e., the maximum size of the population that the environment can sustain indefinitely. The constant k tells us something about how fast the population grows, while C is given by the initial conditions. Let us consider a bacterial colony where we take the carrying capacity to be $B = 50000$ and $k = 0.2\text{h}^{-1}$. If the population is 5000 at $t = 0$, find C and write a code that finds the number of bacteria in the colony after 24 hours.

Filename: `population.py`

Problem 1.3. Solve the quadratic equation
Given a quadratic equation

$$ax^2 + bx + c = 0,$$

the two roots are

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

Make a program evaluating the roots of

$$3x^2 + x + 2 = 0.$$

Filename: `find_roots.py`

Problem 1.4. Forces in the hydrogen atom
There are two kinds of forces acting between the proton and the electron in the hydrogen atom; Coulomb force and gravitational force. The Coulomb force can be expressed as

$$F_C = k_e \frac{e^2}{r^2},$$

where k_e is Coulomb's constant, e is the elementary charge, and r is the distance between the proton and the electron.

The gravitational force can be expressed as

$$F_G = G \frac{m_p m_e}{r^2},$$

where G is the gravitational constant, m_p is the mass of the proton, m_e is the mass of the electron, and r is the distance between the particles. We can use these expressions for F_C and F_G to illustrate the difference in strength of these two forces, i.e., the electromagnetic and gravitational force. Use the values $k_e = 9.0 \cdot 10^9 \mathrm{Nm^2C^{-2}}$, $e = 1.6 \cdot 10^{-19}\mathrm{C}$, $G = 6.7 \cdot 10^{-11}\mathrm{Nkg^{-2}m^2}$, $m_p = 1.7 \cdot 10^{-27}\mathrm{kg}$ and $m_e = 9.1 \cdot 10^{-31}\mathrm{kg}$. You can take the distance between the proton and electron to be approximately the Bohr radius $r = a_0 = 5.3 \cdot 10^{-11}\mathrm{m}$.

Make a program that computes both the Coulomb force and the gravitational force between the proton and the electron. Write out the forces in scientific notation with one decimal in units of Newton ($\mathrm{N} = \mathrm{kgm/s^2}$). Also print the ratio between the two forces.

Filename: `hydrogen.py`

Chapter 2

Loops and Lists

Problem 2.1. Multiply by five
Write a code printing out $5 \cdot 1$, $5 \cdot 2$, ..., $5 \cdot 10$, using either a `for` or a `while` loop.
Filename: `multiplication.py`

Problem 2.2. Multiplication table
Write a new code based on the one from Problem 2.1. This code should print the whole miltiplication table from $1 \cdot 1$ to $10 \cdot 10$.
Hint: You may want to consider using one loop inside another.
Filename: `mult_table.py`

Problem 2.3. Stirling's approximation
Stirling's approximation can be written $\ln(x!) \approx x \ln x - x$. This is a good approximation for large x. Write out a nicely formatted table of integer x values, the actual value of $\ln(x!)$, and Stirling's approximation to $\ln(x!)$.
Filename: `stirling.py`

Problem 2.4. Errors in summation
The following code is supposed to compute the sum $s = \sum_{k=1}^{M} \frac{1}{(2k)^2}$.

```
s = 0; M = 3

for i in range(M):
    s += 1/2*k**2

print(s)
```

This program does not work. Find the errors and write a correct program. Lastly, write a similar code evaluating the same sum using a `while` loop. Check that you get the same answers.
Filename: `sum_for.py`

Problem 2.5. Binomial coefficient
The binomial coefficient is indexed by two integers n and k and is written $\binom{n}{k}$. It is given by the formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{2.1}$$

We can write this out, and get

$$\binom{n}{k} = \prod_{j=1}^{n-k} \frac{k+j}{j}. \tag{2.2}$$

Use Eq. (2.2) and a `for` loop to find the binomial coefficient for $n = 14$ and $k = 3$. Compute the same value using Eq. (2.1) and check that the results are correct.

Hint: The $\prod$ sign is a product sign. Thus $\prod_{j=1}^{n-k} \frac{k+j}{j} = \frac{k+1}{1}\frac{k+2}{2}\cdots\frac{k+(n-k)}{(n-k)}$. When checking the result you will need `math.factorial`.

Filename: `binomial.py`

Problem 2.6. Table showing population growth
Consider again the bacterial colony from Problem 1.2. Let us study the number of individuals for $n+1$ uniformly spaced t values throughout the interval $[0, 48]$. First store the t and N values in two lists `t` and `N`. Thereafter, write out a nicely formatted table of t and N values by traversing the two lists with a `for` loop.

Filename: `population_table.py`

Problem 2.7. Nested list

a) Compute two lists of t and N values as explained in Problem 2.5. Store the two lists in a new nested list `tN1` such that `tN1[0]` and `tN[1]` correspond to the two lists. Write out a table with t and N values in two columns by looping over the data in the `tN1` list. Each t value should be written with two decimals, while each N value should be written as integer.

b) Make a nested list `tN2` which holds each row in the table of t and N values. Loop over the `tN2` list and write out the t and N values with two decimals for the t values and integers for the N values.

Filename: `population_table2.py`

Problem 2.8. Calculate Cesaro mean
Let $(a_n)_{n=1}^{\infty}$ be a sequence of numbers, $s_k = \sum_{n=0}^{k} a_n = a_0 + \ldots, +a_k$, and

$$S_N = \frac{1}{N-1}\sum_{k=0}^{N-1} s_k.$$

Let $(a_n)_{n=1}^{\infty}$ be the sequence with $a_n = (-1)^n$. Calculate S_N for $N = 1, 2, 3, 4, 5, 10, 50$ and print the results in a table.

Filename: `cesaro_mean.py`

Problem 2.9. Catalan numbers
A number on the form

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

is called a *Catalan number*. Compute and print the first 10 Catalan numbers.

Filename: `catalan.py`

Chapter 3

Functions and Branching

Problem 3.1. Implement a function for population growth
Consider again the function

$$N(t, k, B, C) = \frac{B}{1 + C\mathrm{e}^{-kt}}.$$

Implement N as a python function `population(t, k, B, C)` that returns the number of individuals in a population after a time t.

Write out a nicely formatted table of t and N values for the time interval $t \in [0, 48]$ using the values from Problem 2.6.

Filename: `pop_func.py`

Problem 3.2. Sum of integers
We consider the sum $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n$ of positive integers up to n. It can be shown that the sum is equal to $\frac{n(n+1)}{2}$.

a) Write a function `sumint(n)` that returns the sum of all positive integers up to n.

b) Write a function implementing $\frac{n(n+1)}{2}$.

c) Write test functions for both a) and b) testing for specific known values.

Filename: `sumint.py`

Problem 3.3. Implement the factorial

a) The factorial can be implemented by a so called recursive function call. Use a recursive function call to implement a function `myfactorial(n)` that returns $n!$.

Hint: In this case, the recursive function call can be implemented as a function taking a value `n` *and returning* `n*myfactorial(n-1)`*. Include a test to check if* $n = 0$*, in that case return* 1.

b) Write a test function where you call the `myfactorial` function and check the value of the returned object for one value of n using `math.factorial`.

Filename: `factorial.py`

Problem 3.4. Half-wave rectifier

In a half-wave rectifier the positive part of a signal passes, while the negative part is blocked. Thus, for a signal passing through a half-wave rectifier, the negative values are set to zero. Let us look at a sine signal that has passed through a half-wave rectifier:

$$f(x) = \begin{cases} \sin x & \text{if } \sin x > 0 \\ 0 & \text{if } \sin x \leq 0. \end{cases}$$

Implement $f(x)$ as a Python function `f(x)` and make a test function for testing the implementation of `f(x)` in both cases.

Filename: `half_wave.py`

Problem 3.5. Primality checker

Recall that a prime number is a number greater than 1 that has exactly 2 divisors. Said differently, a number greater than one is a prime if it is divisible by only itself and one. A number that is not prime is called composite. Every number n can be written as a unique product of primes (e.g. $12 = 2 \cdot 2 \cdot 3$), this is called the prime factorization of n.

a) Make a function that takes a number n, and returns true if it's prime, and false if it's not. Use the program to find all prime numbers up to 100.

Hint: You will only need to check divisibility for numbers up to and including $\sqrt{(n)}$, because any greater divisor will imply that there is a divisor less than this.

b) Make a function that instead finds the prime factorization of the input number. It should print "prime" and return nothing if the number is prime, and both print and return the factorization if it's composite. Find the prime factorization of 5525612.

c) Make test functions for the two functions above where you check for small values of n.

d) Compare the runtime of the two functions with the number 33425626272. Is the difference big? If so, why do you think one is faster than the other? The following code returns the mean time it takes for your program to run once:

```
import timeit
timeit.timeit('your_func(args)', 'from __main__ import your_func',number=1)
```

Filename: `prime.py`

Problem 3.6. Eulers totient function

Two numbers n and m are called relatively prime if they have no common divisors except for 1. That is, no number greater than one should divide both numbers with no residue.

a) Make a function that takes two numbers and returns true if they're relatively prime and false if they're not.

b) Euler's totient function is defined as

$$\phi(d) = \#\{\text{Numbers less than d which are relatively prime to d}\}.$$

Implement Eulers totient function and print $\phi(d)$ for $d = 10, 50, 100, 200$.

c) Make a test function for both a) and b).
Filename: `euler.py`

Chapter 4

User Input and Error Handling

Problem 4.1. Quadratic with user input
Consider the usual formula for computing solutions to the quadratic equation $ax^2 + bx + c = 0$ given by

$$x_{\pm} = \frac{b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Write a program that asks the user for values of a,b, and c through the users keyboard. Print the solutions.

Filename: `quadratic_roots_input.py`

Problem 4.2. Quadratic with exceptions
Modify the program in Problem 4.1 to read values for a,b, and c from the command line. Use exceptions (IndexError) to handle missing arguments.

Filename: `quadratic_roots_error.py`

Problem 4.3. Quadratic fixed

In this exercise, use the `sqrt` *function imported from* `math`*.*

Consider the program from Problem 4.2. Not all inputs yield valid real solutions (e.g. $a = 1$, $b = 1$, $c = 1$). Modify the program using exceptions or if-tests to handle invalid input and provide a suitable message for the user.

Filename: `quadratic_roots_error2.py`

Problem 4.4. Estimating harmonic series
Let $f(x)$ be the function

$$f(x) = \sum_{n=1}^{\infty} \frac{x^n}{n} = x + \frac{x^2}{2} + \frac{x^3}{3} + \dots$$

Write a program that approximates $f(x)$ (that is, evaluates $f_N(x) = \sum_{n=1}^{N} \frac{x^n}{n}$) with values of x and N given as command line arguments. Run the program for $x = 0.9$, $x = 1$, and $N = 10000$. Print the results.

Remark. For $x = 1$ this is known as the *harmonic series*. Despite the low values for large N, the series does not converge, but diverges very slowly. Try to run the program for different values of N to see how big you can get the value of $f(1)$.

Filename: `harmonic.py`

Problem 4.5. Estimating harmonic series extended
Using the program from Problem 4.4, consider the following values for x and N in a text file

```
x: 0.9 1
N: 500 1000 10 100 50000 10000 5000
```

a) Write a function to read a file containing information in the above format that returns two lists containing the values of x and N.

b) Write a test function for a) that generates a file in the given format and checks that the values returned by the function is correct.

c) Use the program from Problem 4.4 to evaluate $f_N(x)$ for the different values of x and N. Create a function that writes the information to a file in a table format with the first column containing the values of N in increasing order, and the second and third the values of $f_N(x)$ at 0.9 and 1 respectively.

Filename: `harmonic_table.py`

Problem 4.6. A result on prime numbers
A famous result concerning prime numbers states that the number of primes below a natural number n, denoted $\pi(n)$, is approximately given by

$$\pi(n) \approx \frac{n}{\log(n)}.$$

That is, the fraction $p(n) = \pi(n)/\frac{n}{\log(n)}$ tends to 1 as $n \to \infty$. The following table contains the exact values of $\pi(n)$ for some values of n.

```
n: 10**20 10**4 10**2 10**1 10**12 10**4 10**6 10**15
pi(n): 2220819602560918840 1229 25 4 37607912018 168 78498 29844570422669
```

a) Write a function that reads the file given above and returns two tuples containing sorted values of n and $\pi(n)$. It is important that the correspondence in the orderings are correct, that is, the same as in the table above.

b) Write a test function that generates a file with the format above and tests that the returned values are correct. It should test that the order of the elements are in correspondence as in the file.

Hint: The `==` *operator on tuples will take the order into account. The same operator on lists will not.*

c) Create a function that writes the values of n and $p(n)$ to a file in a table format in increasing order with the values of n in the first column and the corresponding values of $p(n)$ in the second column.

Bonus problem There are better approximations to $\pi(n)$, for example the function

$$\mathrm{Li}(n) = \int_2^n \frac{1}{\log(t)} dt$$

Approximate the integral for different values of n and modify the program to write these into a third column.

Hint: Implement an algorithm for approximating the integral (e.g. the trapezoidal rule) and compute the difference as before.

Filename: `primes.py`

Problem 4.7. Conversion from other bases

Recall that a binary number is a sequence of zeros and ones which converted to the decimal system becomes $\sum_i 2^i$ where i is a term in the sequence containing a 1 (e.g. $100101 = 2^5 + 2^2 + 2^0 = 37$).

a) Write a function that takes a binary number and converts it to a decimal number. If the argument is not a binary number, a message should be printed and nothing returned.

Hint: Let the number in the argument be of type string to avoid problems with numbers starting with a zero.

b) Let the binary number from a) be taken as a command line argument. Use exceptions (IndexError) to handle missing input. Print the convertion of 100111101.

c) Extend the program with a function to also handle numbers written in base 3.

Hint: An example of a ternary number(a number in base 3) converted to a decimal number: $1201 = 1 \cdot 3^3 + 2 \cdot 3^2 + 0 \cdot 3^1 + 1 \cdot 3^0$.

Filename: `base_convertion.py`

Chapter 5

Array Computing and Curve Plotting

Problem 5.1. Fill arrays; loop version
We study the function

$$f(x) = \ln(x).$$

We want to fill two arrays `x` and `y` with x and $f(x)$ values, respectively. Use 101 uniformly spaced x values in the interval $[1, 10]$. Create empty `x` and `y` arrays and compute each element in `x` and `y` with a `for` loop.

Filename: `fill_log_arrays_loop.py`

Problem 5.2. Fill arrays; vectorized version
Vectorize the code in Problem 5.1 by creating the x values using the `linspace` function from the `numpy` package and evaluating $f(x)$ with an array argument.

Filename: `fill_log_arrays_vectorized.py`

Problem 5.3. Plot the population growth
Again, we're considering a population undergoing logistic growth. The number of individuals in the population is given by

$$N(t, k, B, C) = \frac{B}{1 + C\mathrm{e}^{-kt}}.$$

Plot this function for $t \in [0, 48]$ with a carrying capacity $B = 50000$, $C = 9$ from the initial condition that we have 5000 individuals at $t = 0$ and a steepeness of $k = 0.2$.

Filename: `population_plot.py`

Problem 5.4. Plot Stirling's approximation
Stirling's approximation is

$$\ln(x!) \approx x \ln x - x.$$

a) Make two functions `stirling(x)` and `exact(x)`, returning Stirling's approximation and the exact value of $\ln(x!)$, respectively. Plot both the approximation and the exact curve in the same figure.

Hint: To implement a vectorized version of the `exact` *function, you can use scipy.special.gamma(x). This function is a "generalized factorial" which can find the "factorial" of float numbers. It works such that* $n! = \text{gamma}(n+1)$*. You can also just consider integer values and plot the value of* $\ln(x!)$ *for each integer* x *in the interval you're considering. Keep in mind that* `math.factorial` *is not vectorized.*

b) Use a `while` loop and find the minimal value of x for the relative error to be less than 0.1%.

Hint: Relative error is given as $(a - \tilde{a})/a$*, where* a *is the exact value and* $\tilde{a}$ *is the approximation. Also, do not start with* x *smaller than or equal to* 1*, why?*

Filename: `stirling_plot.py`

Problem 5.5. Fermi-Dirac distribution

The Fermi-Dirac distribution says something about the probability of an energy state being occupied by a particle, or more precisely a fermion, e.g. an electron. It is a function of energy and temperature given by

$$f(E,T) = \frac{1}{1 + e^{(E-\mu)/kT}}, \tag{5.1}$$

where E is energy, T is temperature, k is Boltzmann's constant and μ is the so-called chemical potential. Use $k = 8.6 \cdot 10^{-5}\text{eVK}^{-1}$ and $\mu = 4.74\text{eV}$ and make a program that visualizes the Fermi-Dirac distribution on the interval $E \in [0, 10]\text{eV}$ when $T = 0.1\text{K}$. (eV is a unit of energy, $1\text{eV} = 1.6 \cdot 10^{-19}\text{J}$.)

Filename: `Fermi_Dirac.py`

Problem 5.6. Animate the temperature dependence of the Fermi-Dirac distribution

Make an animation of the Fermi-Dirac distribution $f(E,T)$ from Problem 5.5 We're interested in studying how the distribution changes when we raise the temperature. Plot f as a function of E on $[0, 10]$ for a set of temperatures $T \in [0.1, 3 \cdot 10^4]$. Also make an animated GIF file. Remember to label your axes and include a legend to show the value of the temperature.

Hint: A suitable resolution can be 1000 *intervals (*1001 *points) along the* E *axis,* 60 *intervals (*61 *points) in temperature, and* 6 *frames per second in the animated GIF file. Use the recipe in Section 5.3.4 and remember to remove the family of old plot files in the beginning of the program.*

Stemmer dette i siste utgave?

Filename: `Fermi_Dirac_movie.py`

Problem 5.7. Bump functions

Consider the function

$$f(x) = \begin{cases} ke^{-\frac{1}{1-x^2}} & -1 < x < 1 \\ 0 & \textit{otherwise.} \end{cases}$$

a) Plot the function with $k = 1$ on the interval $-2 \leq x \leq 2$ by implementing a vectorized version in your program.

b) Animate the function on the same interval as above when k decreases from 1 to 0.

Filename: `bump.py`

Problem 5.8. Band structure of solids
Electrons in solids are waves. These waves have different wave lengths λ. Often, waves are characterised by their wave number $k = 2\pi/\lambda$, and the wave number is associated with the energy of the electron. The energies of electrons in solids have a band structure, i.e., there are different bands of energies separated by a band gap.

The file `bands.txt` contains k-values and corresponding energies for the three first bands of a solid. Have your program read the values for k and the energies and plot the energy bands as functions of k in the same figure. You will see that some energies never can be obtained by electrons in the solids. These areas of non-allowed energies are called the bad gaps.

Filename: `band_structure.py`

Problem 5.9. Half-wave rectifier vectorized
In Problem 3.4, we implemented a function illustrating a sine signal after it had passed through a half-wave rectifier. Vectorize this function and plot $f(x)$ for $x \in [0, 10\pi]$.

Hint: The `numpy.where(condition, x1, x2)` *function returns an array of the same length as* `condition`*, whose element number* `i` *equals* `x1[i]` *if* `condition` *is* `True`*, and* `x2[i]` *otherwise.*

Filename: `half_wave_vec.py`

Problem 5.10. Singularity plot
In this problem we consider the function

$$f(r,\theta) = \left(e^{\frac{1}{r}\cos\theta} \cos\left(-\frac{1}{r}\sin\theta \right), -e^{\frac{1}{r}} \sin\left(-\frac{1}{r}\sin\theta \right) \right)$$

with $0.01 \leq r \leq 1$ and $0 \leq \theta \leq 2\pi$. Create arrays of r and θ values on the unit circle centered at the origin with n uniformly spaced values. Fix axes between -0.5 and 0.5 for x and y and visualize the function for $n = 10, 50, 100, 500$. You can use the following to generate the correct values for r and θ:

```
theta = np.linspace(0,2*np.pi,100)
r = np.linspace(0.01,1,100)
r, theta = np.meshgrid(r,theta)
```

Remark. If we had an ideal computer that could calculate every value in an interval and plot it, then the image we have plotted would touch every single value in the plane, except for at most one! In our program we have $0.01 < r < 1$. The remarkable thing is that the same is true if we replace the inequality with $0 < r < \epsilon$ for any $\epsilon > 0$. Not only that, but all those points are hit an infinite number of times!

Filename: `ess_sing.py`

Problem 5.11. Approximate $|x|$
The absolute value $f(x) = |x|$ can be written as a sum

$$f(x) = \frac{\pi}{2} - \frac{4}{\pi} \sum_{n=1}^{\infty} \frac{\cos(2n-1)x}{(2n-1)^2}.$$

Write a program that calculates the first N terms for $N = 1, 2, 3, 4$ and plots it against the exact function. Let the x-axis be $[-\pi, \pi]$ with a suitable y-axis.

Filename: `approx_abs.py`

Problem 5.12. Plotting graphs
A graph is a collection of lines and points in the plane such that each line connects two points. For two points $[x_1, y_1], [x_2, y_2]$, the line between them consists of the points

$$[tx_1 + (1-t)x_2, ty_1 + (1-t)y_2], \quad 0 \leq t \leq 1.$$

a) Make a function that takes two points and plots the line between them. Plot a vertical and a horizontal line.

b) A complete graph is a graph such that any two points has a line that connects them. Make a function that takes a list of points and plots the complete graph on those points. Choose some points of your choosing in the 1x1 square and plot the graph on those points.

c) Given a natural number n, make a function that plots the following graph:

- Two vertical rows with n points should be placed side by side.
- Each point on the left side should have a line to every point on the right side and vice versa.
- No two points on the same side should be connected by a single line.

Filename: `graph.py`

Problem 5.13. Inefficiency of primality checker
Consider the program from Problem 3.5. Use the `timeit` module and run the program to find the time it takes to find a factorization of an n digit number. Plot the time against the number of digits for the numbers in the file `prime_check.dat`. You can use the following code to time the function for different numbers:

```
str1 = "f(" + str(n) + ")"
str2= "g(" + str(n) + ")"
time1 = timeit.timeit(str1, 'from __main__ import f',number=100)
time2 = timeit.timeit(str2, 'from __main__ import g',number=100)
```

Filename: `prime_ineff.py`

Problem 5.14. Animating a cycloid
One may create a curve by placing a circle on the x-axis, fixing a point on the circle, and then drawing the trace of the point as the circle is rolling. The resulting curve is called a cycloid. In mathematical language it is given as

$$r(\theta) = [R(\theta - \sin\theta), R(1 - \cos\theta)]$$

where R is the radius of the rotating circle and θ is the angle starting at 0 and increasing.

a) Animate the cycloid as a function of θ starting at 0, ending at 15. Draw a point at the end of the cycloid that varies with the animation.

Hint: A point can be added through a new plot using for example

```
point, = axes.plot([],[],'o')
```

and updating during the animation.

b) Add the rolling circle defining the cycloid to the plot. You may use that at a given time θ, the circle is given as $s(\theta) = (R \cdot \theta + \cos\theta, R + \sin\theta)$.

Filename: `cycloid.py`

Chapter 6

Dictionaries and Strings

Problem 6.1. A result on primes "dictionarized"
Consider the program from Problem 4.6. Since the entries correspond to each other, working with two seperate lists is cumbersome. We may avoid that using dictionaries. Modify the program such that the values are saved in a dictionary instead of a list. Let the values of n be keys with values $\pi(n)$.

Filename: `primes_dict.py`

Problem 6.2. Representation of polynomials
Let $f(x) = \sum_{i=0}^{n} a_i x^i$ and $g(x) = \sum_{j=0}^{m} b_m x^m$ be two polynomials. Recall that a polynomial can be expressed as a dictionary with keys equal to the degree of a term with value the corresponding coefficient(So $3x^2 + 1/2$ is represented by the dictionary $\{2 : 3, 0 : 1/2\}$).

a) Create a function that takes two dictionaries (corresponding to two polynomials f and g) as arguments and returns a dictionary corresponding to the sum of the two.

b) Create a function as above that returns the dictionary corresponding two the product of two polynomials.

Hint: $fg = \sum_{k=0}^{n+m} c_k x^k$ *where* $c_k = \sum_{i+j=k} a_i b_j$

c) Add a function that evaluates a polynomial dictionary at a point. Make test functions for all three.

Filename: `poly_dict.py`

Problem 6.3. Saving information in a nested dictionary
The file below contains information about various people. The first column is the name, the second is the age, and the third is the gender.

```
John, 55, Male
Toney, 23, Male
Karin, 42, Female
Cathie, 29, Female
Rosalba, 12, Female
Nina, 50, Female
Burton, 16, Male
Joey, 90, Male
```

a) Create a function that reads the file and returns the information in a nested dictionary. For example the key 'John' has the dictionary {'Age': 55, 'Gender': 'Male'} as value. When reading the file, the name should read "John", not "John, ".

b) Create a function that takes as an argument a nested dictionary as above, a person name, and optional arguments: a number (age), and a string (gender) which returns the same dictionary with the new age and gender. Note that neither should be changed if no age or gender is given.

c) Extend the function with the possibility to change the name of a person. One should not be allowed to change a name to one that is taken (Can you think of a way to allow this without overwriting another person?). Read the file above and change the gender and name of John. Iterate the dictionary and print the new information in a table format.

Remark. The nested dictionary here is a prototype of what is known as a class, and the functions from b) is a prototype for what will be known as methods in that class.

Filename: `people_dict.py`

Problem 6.4. Finding the frequency of words in a text

a) Write a function that reads the file *RandomWords.dat* and finds the frequency of words of length n. Save the information in a dictionary with the length as keys and the number of words of that length as values. You may assume that all words are separated by spaces and that only punctuation marks appear in the text.

Hint: For your program to be compatible with words of any length, it might be helpful to use defaultdict imported from collections. See page 306 in the book. Use the function dict() on such an object to convert it to an ordinary dictionary

Sjekk at sidetallet er rett.

b) Write a test function that generates a file of words and checks that the function returns the correct values.

Filename: `word_length.py`

Problem 6.5. Compute digital roots

Given a number, say 5282, we can compute the sum of the digits. In this case $5 + 2 + 8 + 2 = 17$, and doing this again gives $1 + 7 = 8$. The one digit number we get by doing this is called the digital root of the number.

a) Make a function that calculates the digital root of a number.

Hint: Convert the number to a string in order to work with it.

b) Plot the digital root of numbers up to 500 with the digital root on the x-axis and the frequency of digital roots on the y-axis. Use `plt.scatter(x,y)` for the plot.

Filename: `dig_root.py`

Problem 6.6. Münchhausen Numbers
A Münchhausen number is a number such that the sum of every digit to the power of itself equals the original number. E.g. $1^1 = 1$ is a Münchhausen numbera, and $5^5 + 3^3 + 2^2 = 3156 \neq 532$, so 532 is not.

Make a function that checks if a number is Münchhausen. Find a Münchhausen number different from one.

Hint: There is only one such number different from 1.

Filename: `m_numbers.py`

Problem 6.7. Timezone converter
In the file `timezones.dat` you will find places and their timezone in GMT format.

a) Make a function that reads the file and saves the information in a dictionary.

b) Create a function that takes local Norwegian time (GMT +1) in the string format 'ddmmyy-hhmm', a place, and returns the local time at that place. Your program should display a message to the user if a place that is not saved in the dictionary is used. Do the following conversions:

- March 21st 2018 05.34 in Vancouver
- December 31th 2017 20.03 in Sydney
- January 1st 2018 00.15 in London

Filename: `timezones.py`

Chapter 7

Introduction to Classes

Problem 7.1. Saving information in a class
In this problem you can use the program from 6.3.

a) Create a class `Person` with name, age, and gender as initial arguments.

b) Add methods for changing a persons name, age, and gender.

c) Add a method `__str__` that returns a string with all the information of that person. Create an instance of John as with the information in the table from Problem 6.3. Change the name and gender of John. Print the information of the instance before and after changing.

Filename: `class_people.py`

Problem 7.2. Extending the `AccountP` class
Modify the class `AccountP` in the book to include a method `transfer` that transfers an amount between two accounts. The method should take an amount and the account you want to transfer to as arguments. Write a test function that checks that the methods `deposit`, `withdraw`, `transfer` and `get_balance` works properly.

Filename: `AccountP.py`

Problem 7.3. Approximating the square root of two
The square root of two can be represented by a so called *continued fraction* on the following form:

$$\begin{aligned}\sqrt{(}2) &= 1 + \frac{1}{1+\sqrt{(}2)} \\ &= 1 + \frac{1}{2+\frac{1}{1+\sqrt{(}2)}} \\ &= 1 + \frac{1}{2+\frac{1}{2+\frac{1}{1+\sqrt{(}2)}}}\end{aligned}$$

In this exercise we will exhibit two possibilities for approximating the number $\sqrt{(}2)$.

a) Make a class `Square` with a method `approx_frac` that takes an integer n, an initial value and returns the first n fractions as above with initial value x_0. This can be done by iterating the function

$$f(x) = 1 + \frac{1}{1+x}$$

starting at x_0. For $n = 2$ and $x_0 = 1$ this gives

$$f(f(x_0)) = 1 + \frac{1}{2 + \frac{1}{1+1}}.$$

b) Another way to approximate the square is by iterating the function $f(x) = \frac{1}{2}\left(x + \frac{2}{x}\right)$. Add a method `approx_iter` that takes a number x_0, an integer n, and returns the value of the function at x_0 iterated n times. For $n = 2$ we would have $f(f(x_0))$. From here on we assume for simplicity that $0.1 \leq x_0 \leq 2$.

c) Create a method that returns a nicely formatted table with the two approximations and their difference ϵ along with the exact value for $n = 1, 2, 5, 10$. Run the program, which approximation is best?

d) To visualize the approximation plot the exact value as a line in the plane and the two approximations as points (n, y_n), where y_n is the approximation. Use $n = 1, 2, 5, 10$.

Filename: `square_iteration.py`

Problem 7.4. Tangent lines on a quadratic curve
Consider a quadratic polynomial on the form $f(x) = x^2 + bx + c$. At a point x_0 the tangent line is given by $l(x) = (2x_0 + b)x + C$ where $C = f(x_0) - (2x_0 + b)x_0$.

a) Make a class `Quadratic` with a function $f(x)$ (a quadratic polynomial as above) as initial argument. Make a method that computes the tangent at a point and returns the function $l(x)$.

Hint: You will need to extract the coefficients $b = f(1) - f(0) - 1$ *and* $c = f(0)$.

b) Create a method that plots the function along with its tangent at a point.

c) Make a method that animates the tangent line moving over the curve $f(x)$. Make the animation for uniformly distributed x values in the interval $-5 \leq x \leq 5$. Test the program with the function $f(x) = x^2$.

Filename: `quadratic_tangents.py`

Problem 7.5. Numerical approximations for the derivative
Let $f(x)$ be a function and $f'(x)$ its derivative. There are many ways to approximate the derivative, some of which are:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$
$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$
$$f'(x) \approx \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}$$

a) Make a class `Diff` with a function f as initial argument and implement three methods `diff1`, `diff2`, and `diff3` approximating the derivative using the above formulas.

b) Create a method `plot_diff(self, h, exact=0)` (where exact is the exact derivative if possible) which plots the different approximations in the same figure. Use the program to approximate $f'(x)$ where $f(x) = e^x$ for $h = 1$, 0.8, 0.6, 0.4, 0.2, 0.1. Which approximation is better? Is the difference big?

Filename: `class_diff.py`

Problem 7.6. Visualizing functions

For a function $f(x)$ we can plot the graph of the function as points $(x, f(x))$. This results in a curve in the plane. Suppose we have a function

$$f(x, y) = (u(x, y), v(x, y)).$$

The graph of this function lives in four dimensions and is not easily visualized. On way to visualize these functions is to instead of looking at the graph we look at how f act on points. For example, how the grid lines in the plane look after f is applied.

a) First we consider a specific function $f(x, y) = (x^2 - y^2, 2xy)$. Write a program where you define the function f and make a figure with x and y axis from -2 to 2 where you plot a number of the grid lines in x and y direction in the same plot. You will need around 15 lines in each direction. Make a another plot side-by-side in the same figure of all points $(x^2 - y^2, 2xy)$ where x and y are the points in the first plot.

b) To make the construction more flexible, modify your program to be a class `Visualize` taking a function $f(x, y) = (u(x, y), v(x, y))$ as initial argument. It should contain a method `grid(self, n)` that generates two plots, one of grid lines, and one of the image as in a).

c) We used grid lines of the plane to see how the function f behaved. We could have used any curves in the plane. Extend the class with a method `circ` that instead of using points corresponding to grid lines, uses circles with expanding radii. Let the axes go from -5 to 5 and the radii be uniformly distributed between 0 and 10 (15 circles should be sufficient). Test with the function $f(x, y) = (x^2 - y^2 + x + 1, 2xy + y)$. The second plot should consist of circular like objects with a self-crossing.

d) Add a method `grid_anim` that shows an animation of the image of the functions

$$f_\epsilon(x, y) = [(1 - \epsilon)x + \epsilon u(x, y), (1 - \epsilon)y - \epsilon v(x, y)]$$

where ϵ varies from 0 to 1.

e) Using the functions

$$f(x,y) = (x^2 - y^2, 2xy) \quad \text{and} \quad g(x,y) = (x^2 - y^2 + x + 1, 2xy + y),$$

test the `grid` and `grid_anim` methods on f, and the `circ` method on g. Use 15 gridlines and 15 circles.

Remark. For the student familiar with complex numbers, this is exactly how one would visualize a complex function $f(z)$. In our case we can use this for any function $f(x,y)$, but we usually restrict ourself to look at functions corresponding to certain complex functions, namely the differentiable ones.

Filename: `plot_functions.py`

Chapter 8

Random Numbers and Simple Games

Problem 8.1. Throw a die
Compute the probability of getting a 6 when throwing a die. Write a program that throws a die N times and count how many times the die shows 6, let this number be M. Then compute the probability of getting a 6 when throwing a die as M/N.

Filename: `die.py`

Problem 8.2. Telephone number
A Norwegian telephone number consists of eight digits. We assume that all digits from 0 to 9 are equally probable in every place of the telephone number. Make a program that finds the probability of having a telephone number where the digit 1 appears at least four times.

Filename: `telephone.py`

Problem 8.3. Coin-flip game
Two persons are playing a simple coin-flip game. They flip a coin in turn, and whoever first gets a heads wins the game. Make a program to model 100 such games. Estimate the probability for the first person to flip to win the game.

Filename: `coin.py`

Problem 8.4. Approximate π by throwing darts
You are throwing darts at a square shaped target with an inscribed circle. Let the length of the sides of the square be 2, which means that the circle has radius 1. Assume that you throw the darts such that the darts gets uniformly distributed on the target. Then, the number of darts which hits the target inside the circle divided by the total number of darts that hits the target is approximately the area of the circle divided by the total area of the target. This approximation gets more accurate the more darts you throw.

$$\frac{\text{number of darts inside circle}}{\text{number of darts that hits target}} \approx \frac{\text{area of circle}}{\text{area of target}} = \frac{\pi}{4}.$$

Thus, π can be approximated by

$$\pi \approx 4\frac{\text{number of darts inside circle}}{\text{number of darts that hits target}}.$$

Write a program that throws M darts uniformly on the target. Then approximate π. Read M from the command line.

Filename: `approximate_pi.py`

Problem 8.5. Wheel of fortune
At an amusement park they have a wheel of fortune where you can win 2kg of chocolate. You get to choose one number between 1 and 20 for 20NOK. Assume that you play on the same number until you win.

a) Write a program that finds the average number of times you have to play before you win and check if you earn or loose money, compared to buying 2kg of chocolate in the store.

b) Modify your program so that every time you lose you move one place to the right, i.e. you increase n by one. If you are at $n = 20$ you go back to $n = 1$. Does this make any difference to the result?

Filename: `wheel_of_fortune.py`

Problem 8.6. Birthday probability
Make a function that generates a string of random integers between 0 and 9. Estimate the probability that your birthday is contained in a string of random numbers of length 1000. Let the format of the date be on the form `ddmmyy`. Print the estimates in %.

Filename: `birthday_prob.py`

Chapter 9

Object-Oriented Programming

Problem 9.1. Implement Newtons method

a) Make a subclass `Function` of the class `Diff` in problem 7.5 that takes a function f as an initial variable. It should contain a method such that the following code is compatible with your program and prints the value of f at 2.

```
def f(x):
    return x**2
func = Function(f)
print(f(2))
```

b) We would like the class to give estimated values for roots of f. That is, points such that $f(x) = 0$. To do this we implement Newton's formula. It is given recursively as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

where we give a starting point x_0. In some cases(not all) x_n will approach a root of f. Implement this in a method `approx_root` that takes a starting point and a bound $\epsilon < 1$ as arguments and approximates x_n such that $f(x_n) < \epsilon$.

Hint: Implement a simple convergence test. Check that $f(x_n) < 1$ after 100 iterations. If not terminate the loop and inform the user that there is no convergence for that starting point. It is still a possibility for convergence, but unlikely.

c) Test the program with the function $f(x) = x^2 - 1$ and starting value 5 with bounds 10^{-i}, for $i = 1, 2, 3, 4, 5, 6$. Print the approximated value for x, $f(x)$ and the bound in a table format. Try to run the program with starting value 0. What happens, can you see why?

Filename: `newton.py`

Appendix A

Sequences and Difference Equations

Problem A.1. Computing Bell numbers
Let $B_0 = B_1 = 1$, the n'th *Bell number* is defined recursively as

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k.$$

Make a function that returns the n first Bell numbers and print the first 10.
Filename: `bell.py`

Problem A.2. Solve a difference equation numerically
We study the difference equation

$$x_n = x_{n-1} + x_{n-2}.$$

Write a program that writes out the first 15 elements of the sequence for $x_0 = x_1 = 1$.
Filename: `fibonacci.py`

Problem A.3. The spreading of a disease
We want to study the spreading of a disease. Assume that 25% of the people that are ill this week are still ill next week. It takes two weeks from when you get infected until you become ill, and a person who is ill will on average infect 5/7 persons each week, who will become ill two weeks later. Let x_n be the number of ill people in week n. Thus, the number of ill people is given by the following difference equation

$$x_n = \frac{1}{4} x_{n-1} + \frac{5}{7} x_{n-2}.$$

a) Let $x_0 = 100$ and $x_1 = 150$. Write a program that stores the number of ill persons in an array up to week $N = 50$ and plot the result. What happens if an ill person on average infects 3/4 persons each week?

b) We do not need to store all the $N+1$ values. Since x_n only depends on x_{n-1} and x_{n-2}, these are the only values we need to store. Modify the program from a) to use two variables and not an array for the entire sequence. Print the number of ill persons each week in a nicely formatted table.

Filename: `disease.py`

Problem A.4. Find difference equations for computing $\ln x$
The Taylor expansion of $\ln x$ is

$$\ln x = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{(x-1)^n}{n},$$

for $x \in (0, 2]$.

We can define the sum as

$$\ln x \approx S(x; n) = \sum_{j=0}^{n} (-1)^{j+1} \frac{(x-1)^j}{j},$$

so that $S(x, n) = \sum_{j=1}^{n} a_j$ and

$$a_j = -\frac{(j-1)}{j}(x-1)a_{j-1}.$$

Introduce $s_j = S(x, j-1)$ and a_j as the two sequences to compute. We have the initial values $s_1 = 0$ and $a_1 = (x-1)$.

a) Find the set of difference equations for s_j and a_j.

Hint: You can find an example on how this is done for e^x in section A.1.8 in the book.

b) Implement the system of difference equations in a function `ln_Taylor(n, x)`, which returns s_{n+1} and $|a_{n+1}|$. The term $|a_{n+1}|$ is the first neglected in the sum and may act as a rough estimate of the size of the error in the Taylor polynomial approximation.

c) Verify the implementation by computing the difference equations for $n = 3$ by hand and comparing with the output from the `ln_Taylor` function. Automate this comparison in a test function.

d) Check that the accuracy of the Taylor polynomial improves as n increaces and x is close to 1. What happens when $x > 2$?

Filename: `ln_Taylor_series_diffeq.py`

Problem A.5. Lotka-Volterra two species model
We have previously studied the logistic model for poulation growth. This is a model showing the growth of a population in the abscence of preditors. The Lotka-Volterra model describes interactions between two species in an ecosystem, a predator and a prey. We will in the following take the preys to be rabbits and

the predators to be foxes. The number of rabbits and foxes in week n is denoted by R_n and F_n respectively, and the population is modelled by the equations

$$R_{n+1} = R_n + aR_n - cR_nF_n$$
$$F_{n+1} = F_n + ecR_nF_n - bF_n,$$

where a is the natural growth rate of rabbits in the absence of predation, b is the natural death rate of foxes in the absence of food (rabbits), c is the death rate per encounter of rabbits due to predation and e is the efficiency of turning predated rabbits into foxes.

Write a program that computes the number of rabbits and foxes up to $n = 500$. Use $a = 0.04$, $b = 0.1$, $c = 0.005$ and $e = 0.2$. In the begining we have $R_0 = 100$ and $F_0 = 20$. Plot how the number of individuals in the populations vary with time.

Filename: `Lotka_Volterra.py`

Appendix E

Programming of Differential Equations

Problem E.1. Decrease the length of the time steps
We have the following differential equation

$$\frac{dx}{dt} = \frac{\cos(6t)}{1+t+x}.$$

Use Forward Euler to solve this differential equation numerically. You should solve it on the interval $t \in [0, 10]$ for $n = \{20, 30, 35, 40, 50, 100, 1000, 10000\}$. Plot all the solutions in the same plot.

Filename: `decrease_dt.py`

Problem E.2. Implement Euler's midpoint method
Make a subclass `Midpoint` in the `ODESolver` hierarchy from Section E.3 for solving ordinary differential equations with Euler's midpoint method. This method computes

$$\begin{aligned} u_{k+1/2} &= u_k + dtf(u_k, t_k)/2, \\ u_{k+1} &= u_k + dtf(u_{k+1/2}, t_k + dt/2). \end{aligned}$$

Test your implementation on $y = x\cos(x)$ and plot the numerical solutions obtained from both Euler's midpoint method and Forward Euler together with the analytical solution. Use 15 time steps on the interval $x \in [0, 10]$.

Filename: `Midpoint.py`

Problem E.3. Modeling war between nations
We consider the interaction between two nations C1 and C2 and a system of equations for modeling a conflict between these [**braun**]. Assuming that each nation is determined to defend itself against a possible attack, let $x(t)$ and $y(t)$ denote the armaments of the first and second nation respectively. The change $x'(t)$ depends on the armaments of $y(t)$. We assume that it's proportional to it, say $ky(t)$ for some positive constant k. It also depends on the relationship of the two. Assuming anger leads to increased armaments, let g measure the relationship between them, positive numbers meaning anger towards the other nation and 0 means neutral, and negative numbers meaning disarmament. The cost of having

an army will restrain $x(t)$, represented by a term $-\alpha x$ for some positive constant α. Similar setup for $y(t)$ yields a system of differential equations:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = ky(t) - \alpha x(t) + g, \quad \frac{\mathrm{d}y}{\mathrm{d}t} = lx(t) - \beta y(t) + h. \tag{E.1}$$

In the case where $x'(t) = y'(t) = 0$ we have reached a stable point where neither nation is increasing armies. We interpret such a fixed point as peace. In the case were $x(t)$ and $y(t)$ diverges we have an arms race, and we interpret this as war.

a) Make a function that solves the system (E.1) with a numerical method of your own choice (you may use ODESolver to do this) and a function that plots the solution curves of $x(t)$ and $y(t)$ for given initial values. Your program should not solve beyond the point where either x or y is zero. We want to allow the value zero, so have your program check whether x and y are larger than a very small negative number. If you use ODESolver this can be done by defining a terminate function to send with the solve method. Until otherwise specified, we let t be the time measured in years.

Filename: `C_model.py`

b) Modify your program to instead consist of two classes. The first class `ProblemConflict` should contain the following:

- An init method saving all information relevant to the problem (parameters etc)
- A call method such that the class can be called as a function. It should take an array $[x, y]$ of specific values of x and y at time t, the time t, and return the right hand side of the ODE system.

The second class `Solver` should consist of the following:

- An init method that takes a problem instance on the form above, and a step length dt.
- A method that solves the problem, with the same restrictions as in Problem a). It should solve any problem on the same form as `ProblemConflict` that is given by two differential equations.
- A method that plots the solutions as in Problem a).
- A method that saves an image of the plot in `.png` format. When this is called, no plot should be visible to the user.

Use the parameter values $\alpha = \beta = 0.2$, $g = h = 0$, $x_0 = 10000$, $y_0 = 20000$. Run the program once with $k = l = 0.2$, and once for $k = l = 0.3$ plotting the first 10 years. What is the interpreted difference between these two?

Hint: You may need to convert step length to the number of time points to use. This can be done as

```
n = int(round( "Last time step" / "Step length")) + 1.
```

Filename: `C_model_class.py`

c) Let us consider the parameter values $k = l = 0.9$, $\alpha = \beta = 0.2$, and $g = h = 0$. One can argue that these give rough estimates for the arms race between 1909 to 1914 between the alliances of France and Russia, and Germany and Austria-Hungary [**braun**]. Assuming stability prior to this and negligible armies, we assume $x_0 \approx 0$ and $y_0 \approx 0$ (This does not mean that neither nation had armies, but that they were much smaller prior to the arms race). Solve the problem when x_0 and y_0 are zero versus when they are small positive numbers. Plot the next 5 years of both in the same figure. What happens?

Filename: `C_model_c.py`

d) So far we have seen a model intended to describe a conflict situation prior to war. The preceding model doesn't describe what happens during a war, but similar equations can.

First of all, we will work with two types of warfare. The conventional one, what we know as regular warfare, and guerrilla warfare, where groups of combatants use military tactics such as ambush, raids, hit-and-run, among others.

We first consider two conventional armies engaging. Let $x(t)$ and $y(t)$ denote the respective forces (the number of soldiers) and t denote the time measured in days. The rate of change of $x(t)$ is affected by combat loss, operational loss (non-combat related. e.g. disease, accidents), and reinforcements. Combat loss should be proportional to the size of the opponent, represented by a term $-\alpha y(t)$, $\alpha > 0$. The operational losses should depend only on $x(t)$, represented by a term $-kx(t)$, $k > 0$. The reinforcements are represented by a function $f(t)$. In short-term warfare, the operational losses are negligible, and we will assume it to be zero. We get equations

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\alpha y(t) + f(t), \quad \frac{\mathrm{d}y}{\mathrm{d}t} = -\beta x + g(t).$$

For a conventional-guerrilla combat, $y(t)$ representing the guerrilla army, we assume that the combat losses also depend on the size of its own army. As guerrilla armies often use strategies of surprise and hidden attacks, it is safe to assume that bigger losses are experienced when the army is larger. Let $-\beta x(t)y(t)$ denote the combat loss of a guerrilla army. By the same arguments as above, we get equations

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\alpha y(t) + f(t), \quad \frac{\mathrm{d}y}{\mathrm{d}t} = -\beta x(t)y(t) + g(t).$$

Make two classes `ProblemCCWar` and `ProblemGCWar` on the same form as `ProblemConflict` representing the new problems. Note that f and g are now functions. To handle the case when they are constant, you may need the commands `callable(f)` that checks if f a callable, and `isinstance(f, (float, int))` that checks if f is a float or int, in order to convert a constant to a constant function.

Filename: `CW_model.py`

e) The battle of Iwo Jima is a famous battle during World War II. It was fought on an island just outside of Japan. America invaded the island on February 19, 1945, and the fight lasted for 36 days. The Japanese army consisted of around 21500 soldiers, while the Americans had a number above 50000 by the 36th day.

During the war, the Japanese had no reinforcements. The Americans started with no soldiers, but landed 54000 soldiers the first day, 6000 the third, 13000 the sixth, and none for the remaining. The reinforcements is therefore given as

$$f(t) = \begin{cases} 54000 & 0 \leq t < 1 \\ 0 & 1 \leq t < 2 \\ 6000 & 2 \leq t < 3 \\ 0 & 3 \leq t < 5 \\ 13000 & 5 \leq t < 6 \\ 0 & t \geq 6 \end{cases}$$

It can be shown that good estimates for the parameter values are $\alpha = 0.0544$ and $\beta = 0.0106$ [**braun**]. The exact values on a day to day basis is given in the file `Casualties.dat`. Plot the modeled American army vs the exact numbers, and $y(t)$ vs $x(t)$. Both plots should have the x-axis corresponding to the first $T = 36$ days.

Filename: `iwo_jima.py`

f) Find the least number of soldiers Japan would need (according to the model) to have won the fight. You may round to the nearest hundred. *Hint: Check which army decreases to zero first. You might want to extend the variable T for this.*

Filename: `least_number.py`

g) Suppose the Japanese army was interpreted as a guerrilla army. Find a value for β such that the fight is close. Is it likely that the outcome would be different if America met a large guerrilla army?

Filename: `guerrilla.py`