1 Lecture 1

1.1 Introduction and Welcome

1.1.1 The Role of Computer Science for Aerospace

This is presented orally, maybe at a later time, I will write up some of the aspects. The most important ones is that "learning" is not enough for this course.

In fact, you need to "learn" the basics, but you as well have to "practice" the topics creating routines such that you can create error-free programs without much cognitive load. With this training (like in sports: do simple things over and over again until they become natural) you can focus on the problem you want to solve with computational support rather than on the computation itself.

Many things look easy, but they usually are not. You don't learn from reading or hearing (passive interaction). Take a sheet of paper and write down things. Think with a pen. And try **not to use** your computer. You need to train your imagination, your handwriting skills.

1.2 A First Algorithm: Constructing the Bisector of a Line

An algorithm is a description of a procedure or sequence of steps that solve a certain problem. The notion of an algorithms in this generality are a key element of every biological system in which a certain behavior gives rewards, like that eating avoids dying for hunger. Animals follow quite sophisticated protocols when solving the tasks of their lives and not seldom one would be surprised by the amount of wisdom and aesthetics embedded in such behavior.

However, for the sake of this lecture it might be interesting to find examples of algorithms that the readers have already been exposed to. And while it is quite common to use the cooking recipe as an example of a formalized algorithm in which even algorithmic aspects such as input (the ingredients) and output (the dish) can be introduced, we choose an area in which most people have been trained early in school maybe not realizing that it is all about algorithms.

The first algorithm in this lecture is, therefore, an algorithm to construct the middle point of a line by creating the bisector. This is a rather traditional construction from Euclidean geometry.

Algorithm 1. *One draws a circle with the same radius r around A and B. If the radius is large enough (greater than half the distance), we are left with two intersection points of the circles which together define a line. This line is the bisector and the intersection of this respective line with the original line is the middle point p.*

When looking closely at the description, we note that certain things are "just" expected to be possible while the algorithm does not tell us how to do so. It might be that you will encounter many students or pupils that will ask you questions like "and how do I get the radius without a ruler?" or "Why is it clear that I can observe the intersection of two lines?".

The point is that the previous algorithmic description is incomplete in that it did not describe the abilities of the "system" (pupil, student) that is expected to execute the algorithm. Let us complete this a little bit:

Assume we have a device (or person, or computer, or . . .) that enables us to

- Draw a straight line between two points in space
- Observe intersections of pairs of lines or pairs of circles resolving them to points

- Draw a circle around any of the involved points such that the circle intersects an already existing point.

With such a device, we can formulate the following algorithm:

Algorithm 1: Given a line between two points A and B, we can draw a circle around A intersecting B and a circle around B intersecting A. These two circles will intersect in two points, say C and D. Now draw the line C D and observe the intersection with A B. This is the middle point.

Now, we should check whether the algorithm can be implemented on above machine by making more explicit use of the machine itself and introduce a machine language. This language contains one line for each operation we are allowed to do. With easily understandable notations, we can try to write our algorithms as follows:

Listing 1.2.1 A first program

```
1 Circle(A,B)
2 Circle(B,A)
```

But now we find ourselves in a situation in which just writing up the algorithm becomes tricky. How do we express which intersection points are meant? In fact, we just want to intersect the two circles that are already constructed in Listing 1, but how can we extend our formal way of expressing the algorithm towards this? The idea is simple and also well-known from school geometry. Giving names to things, usually by assigning various types of letters to geometric objects such that they can be referenced in scripts.

Listing 1.2.2 Bisector of a Line in Euclidean Geometry

```
1 C1 := Circle(A,B)
2 C2 := Circle(B,A)
3 C,D := Intersect(C1,C2)
4 L1 := Line(A,B)
5 L2 := Line(C,D)
6 P := Intersect(L1,L2)
```

What we have now silently introduced are two related concepts: The idea of a variable and the idea of assignment. For now, variables can be used to give names to geometric objects that have been constructed by our device. Only in this way, Intersect can be sensibly formulated.

Now, this is a programming language that is quite similar to assembler or machine code which we will learn to know a little bit later. However, it still ignores one constraint that all computers (including humans) typically have: It uses infinite memory. What if our pupil has a limited capacity of remembering things? What if he has only one sheet of paper to write up things? Well, the typical advice would be not to write up and name everything (though in school you might have heart the opposite). One construction that makes this possible is the use of expressions which are related to a central notion of algorithms known as recursion. In fact, the two lines constructed in above algorithm Line(A,B) and Line(C,D) are constructed solely for the purpose of formulating the Intersect line. Can we avoid giving names to them and storing them? How can we?

From a programming language point of view, we introduce expressions. These feel very intuitive as you are used to this approach from mathematics. Look at the following program code:

Listing 1.2.3 Bisector of a Line in Euclidean Geometry

```
1 C,D := Intersect(Circle(A,B),Circle(B,A))
2 P := Intersect(Line(A,B),Line(C,D))
```

Here, we have restructured the algorithm for a reader giving stress to the fact that in the first step the circles are not interesting (in school you did only draw a fragment of the circle to create the

intersection, do you remember?), but C and D are. They get names. So we can talk about them, maybe in explanations or proofs of correctness. Same holds for the lines that are constructed solely for formulating the intersection arguments.

The change that happened to our language is that parameters do not need to be data (e.g., points or pairs of points), but can be replaced with operations that would "return' ' data. While the change does not look too much, it is a big conceptual change from assembler-like languages to higher languages, in fact, towards procedural languages. Now you might have observed, that the above snippet is not optimally compact. Given the purpose of our algorithm, to find P, the values C and D are also useless to remember as they are used in a single place below line one. We can compress to

Listing 1.2.4 Bisector of a Line in Euclidean Geometry

```
1  P:=Intersect(Line(A,B),
2           Intersect(Circle(A,B),Circle(B,A)))
```

Again, we silently introduced a trick common in programming langauges: expressions can use more than a single line of source code and, thus, become fairly expressible and complex.

This raises the question, whether using such a feature is always a good idea. From a programming point of view, it is (in these days and with exceptions) a good strategy to give names not only to the output, but also to entities that you might want to use in debugging and testing. You might want to be able to access intermediate results when developing and trying to understand your program. And you want other people to be able to understand your program. In addition, the overhead that a naïve execution of such decomposed statements implies is often removed automatically by computer programs turning your program source code into an executable form. Such programs, that taken a given programming language convert the program into another program language (either a programming language, or machine code) are called compiler.

The idea that every program can be written as a (single) expression is the root of functional programming which has long been a useful tool in proving algorithms correctness or in special branches like physics. However, in the last decades functional programming languages or at least elements that have been invented in the context of these programming language have become more and more popular. For example, the big data system Apache Spark is written (largely) in Scala, which represents a functional language on top of the Java Virtual Machine. Or MapReduce, a programming pattern popularized by Google in the last decades for big data, goes back to early functional programming.

A third important way of expressing algorithms is by declaring the result instead of giving a notion of how to reach to the result. For our geometry example, this could look like

$$\begin{aligned} M &= A + \tau(B - A) \\ M &= C + \lambda(D - C) \\ \|C - A\| &= \|A - B\| \\ \|C - B\| &= \|A - B\| \\ \|D - A\| &= \|A - B\| \\ \|D - B\| &= \|A - B\| \end{aligned}$$

Basically, each of these lines marks a geometric constraint. The question that our computer would have to solve is to decide whether suitable τ and λ exist, whether they are unique, and to compute them. This idea leads into the domain of declarative programming languages including the important branches of logical programming and constraint programming. The idea is to describe sufficient properties of the solution not giving any hit on how to compute the solution effectively specifying a search problem.

However, these algorithms are not at the mainstream of computing (though they have a subfield in which they are essential) as the constraint-based notion is non-constructive. Go back to our bisector-of-a-line example. In the last description, the computer would have to solve it without geometric intuition or operations. This is possible and boils down to the same line of thinking, but giving certain concepts like circles and lines a name simplifies the search for a solution.

1.2.1 A second algorithm: Written Addition

Let us now consider a second algorithm you know from primary school: written addition on a sheet of paper. We assume that you are able to add all one-digit numbers with each other. This knowledge can be given to a device or computer, for example, in a table:

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

Algorithm 2. *Given two n-digit numbers $a = (a_n, \ldots, a_1)$ and $b = (b_n, \ldots, b_1)$, prepare a number as follows: Starting from $i = 1$, set the $i-th$ digit (from the end) of an output variable $c = (c_{n+1}, \ldots, c_0)$ to the sum of a_i and b_i modulo 10. If the number $a_i + b_i$ is larger than 10, remember this and add an additional 1 to the next digit. If the last input digit c_n does not lead to a carry, remove the slot c_{n+1}*

Let us write this algorithm in the C++ programming language, which we will introduce a little later in this course.

Listing 1.2.5 Adding two n-digit numbers

```
1  #include<iostream>
2  #include<vector>
3  #include<cassert>
4  // compile: g++ -std=c++11 -o add add.cpp
5
6  std::vector<int> add (std::vector<int> &a, std::vector<int> &b)
7  {
8      // simplifying assumption: same length
9      assert(a.size() == b.size());
10     std::vector<int> c(b.size()+1);
11     int carry = 0;
12     for (int i=a.size()-1; i >= 0; i--)
13     {
14         auto digit = a[i] + b[i] + carry;
15         carry = (digit >= 10)?1:0;
16         auto reduced_digit = digit % 10;
17         std::cout << "At␣"<<i << ",␣we␣have␣a[i]="<< a[i]
18                   << ",b[i]=="<<b[i]
19                   << ",digit=="<<digit
20                   << ",carry=="<<carry
21                   << "c[i]␣=␣reduced_digit=="<<reduced_digit
```

```
22                      << std::endl;
23          c[i+1] = reduced_digit;
24      }
25      c[0] = carry; // can be 0 or 1
26      return c;
27
28  }
29
30  int main(void)
31  {
32     std::vector<int> a {5,2,3}; // represents 123
33     std::vector<int> b {6,2,3}; // represents 123
34
35     auto output = add(a,b);
36     std::cout << "Result: ";
37     for (const auto digit:output)
38       std::cout << digit;
39     std::cout << std::endl;
40  }
```

The output of this program should be something like

Listing 1.2.6 Output of adding two numbers program.

```
1  At 2, we have a[i]=3,b[i]==3,digit==6,carry==0,c[i] = reduced_digit==6
2  At 1, we have a[i]=2,b[i]==2,digit==4,carry==0,c[i] = reduced_digit==4
3  At 0, we have a[i]=5,b[i]==6,digit==11,carry==1,c[i] = reduced_digit==1
4  Result: 1146
```

For now, just note that this algorithm creates a vector or array of results of size $N + 1$ for two equally-sized inputs of N digits. The first digit of the result can be zero in which case it would usually be removed.

The algorithm itself consists of a loop with $N + 1$ steps. Each of these steps creates one output digit by adding three numbers: One digit from each input and the local variable (we will learn more formally what this means) carry which takes a value of one if and only if the previous addition of two digits was larger than 10. Beyond this simple addition (maybe based on a table as above), we need to split the result into the carry (one if larger than 10, 0 otherwise?) and the digit value ($a_i + b_i \mod 10$).

In summary, this algorithms performs a loop with N steps and in each loop three additions, one modulo, one comparison each silently including the assignment of a variable or an output array slot. Counting these as five steps and assuming that they all take the same time (maybe they are implemented in a digital computer and each of them takes exactly one clock cycle), the loop contains

$$5N$$

instructions. Finally, we need to set the carry into the first slot of the output leading to

$$5N + 1$$

steps.

1.2.2 A third algorithm: Insertion Sort

Now, we introduce a very simple algorithm for a basic problem of computer science and everyday life: sorting things.

Formally speaking

Definition 1. *An algorithm $\mathcal{A}$ solves* ***the search problem*** *if given an array $A = (a_1, \ldots, a_n)$ it creates an array B with the same entries as A, but $b_i \leq bi + 1$ for all $i = 1 \ldots n - 1$. Such an algorithm*

Definition 2. *Such an algorithm $\mathcal{A}$ is called* ***in-place*** *if the input array A is used to hold the output (and typically only a small constant amount of additional memory is used).*

Insertion Sort is now a very simple algorithm which you are used to from everyday's life. Given a non-ordered set of things on a table ordered from left to right (like playing cards or pens you want to order by color), you start by realizing that the left-most item alone is a sorted subset. Then you take the next element into your hand leaving an empty slot. Now you proceed by shifting each element left from the empty slot into the empty slot as long as the current element in your hand is larger than the element you are shifting. The first time that this condition is not true anymore, you have moved the empty spot into the location where you can deposit the object from your hand. As the items left of the item you pick are already sorted, now, the elements are sorted up to and including the slot from which you picked the element. Then you continue with the slot right from it. In the end, when there is not slot to the right, everything is left from the slot and therefore sorted.

Let us now implement this function in MATLAB, a second language we are going to use a lot in this lecture.

Listing 1.2.7 Insertion Sort

```
A = [1,2,4,3,5,2]
% Insertion Sort
for j = 2:length(A)
    in_hand = A(j);
    i = j-1;
    while (i > 0 && A(i) > in_hand)
        A(i+1) = A(i); %shift right, consider A[i] empty
        i = i -1;
    end
    A(i+1) = in_hand;
    disp(A)
end
```

Let us see, how this algorithm behaves by writing step by step how the array A looks like. For clarity, note that we will write ϵ for places that would be empty in the real world. In the implementation, they are not overwritten with some empty, as they are finally overwritten either from the left neighbor or from the current value in your hand.

Hand	A_0	A_1	A_2	A_3
ϵ	4	3	2	1
3	4	ϵ	2	1
3	ϵ	4	2	1
ϵ	3	4	2	1
2	3	4	ϵ	1
2	3	ϵ	4	1
2	ϵ	3	4	1
ϵ	2	3	4	1
1	2	3	4	ϵ
1	2	3	4	ϵ
1		3	ϵ	4
1	2	ϵ	3	4
1	ϵ	2	3	4
ϵ	1	2	3	4

1.2.3 A fourth algorithm: Euklid's algorithm

One of the oldest algorithmic descriptions (that is, a description of a procedure in a very abstract context) is represented by Euklid's algorithm which we will discuss in two versions.

Euklid describes in this book "Elements" a version of the algorithm which is asking for a joint measure of two lengths. What is the longest length unit that can express two given length as multiples. In terms of integer numbers, this is known as the greatest common divisor (GCD) which also plays a fundamental role in working with rational numbers.

Definition 3. *The greatest common divisor of two positive integer numbers a and b is the largest number g such that numbers s and t exists such that $a = g \cdot s$ and $b = g \cdot t$.*

Example 1.
- $gcd(44, 4) = 4$
- *Let p, q be prime numbers. Then $gcd(p, q) = 1$.*

A baseline algorithm can be derived from the definition: We can try every number smaller than $\min(a, b)$ and check whether it is a divisor of both. During this search, we remember the largest.

Listing 1.2.8 Baseline GCD Algorithm

```
1  // written a|b
2  bool divides(int a,int b)
3  {
4      int q = b/a; // integer division
5      return q*a == b; // see if there is a remainder
6  }
7
8  // alternatively use the modulo function
9  bool divides2(int a, int b)
10 {
11    return (b%a) == 0;
12 }
13
14
15 int simplegcd(int a, int b)
16 {
17    int gcd = 1;
18    for (int i=2; i <= a && i <= b; i++)
19       if (divides(i,a) && divides(i,b))
20          gcd = i;
21    return gcd;
22 }
```

Naive Correctness

Let us now talk about whether this algorithm is *correct*. While there are quite formal frameworks for the topic of correctness, we will argue naively. Therefore, we first define

Definition 4. *An algorithm $\mathcal{A}$ is correct if for every valid input x, the algorithm* terminates *and produces an outcome $y = \mathcal{A}(x)$ as specified in the purpose of the algorithm.*

Proof of Correctness. As we want to compute the greatest common divisor, we can do this for all positive integer numbers. That is, we assume $a, b > 0$. We have complex for loop and we need to see that the breaking condition (middle part of the for) is eventually becoming true. But as we can

formulate the following loop variant: “The value of i is increased in every loop”, we conclude using that a and b are not changed in the algorithm, that i will supersede either a or b ending the for loop. This proofs the "terminates for every input" part.

The correctness part is now shown using a loop invariant: the following statement holds true before the loop, during the loop and after the loop. “gcd divides a and b and is the largest value smaller than i with this property”. As i takes only values greater or equal to two, the start value one fits to this statement. During the loop, the condition remains true (use induction!). After the loop, the invariant can be transformed by replacing i with $\min(a, b)$ showing that gcd is the largest value which divides a and b (exploiting that larger values never divide a number). □

Remark. *An **assertion** is a feature of a programming language to check such conditions at compile time or at run time.*

1.2.4 Euklid's Version

As a helpful notation, we introduce the integer definition with remainder as follows:

Lemma 1. *Given two positive integer numbers a and b, there is a representation*

$$a = q \cdot b + r$$

With $r < b$ this representation is unique.

Proof. We set $q = \lfloor \frac{a}{b} \rfloor$ to be the largest integer smaller than a divided by b. Then $r = a - qb$.

Show size of R in this setting

□

Definition 5. *In this representation, q is called the integer division of a and b and r is called the remainder. We write $q = aDIVb$, $r = amodb$, $a \equiv r(modb)$ (read a is congruent r modulo b), $a \equiv_b r$. In programming languages, we often have* `r = a %b`.

The classical version of the algorithm of Euklid is covered in Task **??**. Note that this is formulated without division or modulo operations and is, therefore, suitable for implementation on extremely constrained computers. It is a nice example that even basic arithmetic operations can sometimes be avoided and replaced with loops of simpler ones.

The modern version of this algorithm is based on the following lemma and a repeated application of computing the remainder.

Lemma 2. *Let a, b be positive integers and divide a by b with remainder as in $a = qb + r$. Then $t|a$ and $t|b \Leftrightarrow t|band t|r$*

Proof. $\Rightarrow$: Assuming $t|a$ and $t|b$, we need to show $t|r$. The divisibility can be written as

$$a = t_a t$$
$$b = t_b t$$

We substitute in $a = qb + r$:

$$t_a t = a = q(t_b t) + r$$

Hence,

$$r = t_a t - t_b t q = t(t_A + t_b q)$$

That is $t|r$.

$\Leftarrow$: Conversely, let $t|r$ and $t|b$. We need to show $t|a$. Analogously write

$$r = t_r t$$
$$b = t_b t$$

and substitute in $a = qb + r$

$$a = qt_b t + t_r t = t(qt_b + t_r),$$

hence $a|b$. □

Lemma 3. *Let $a = qb + r$ represent integer division with remainder. Then*

$$gcd(a, b) = gcd(b, r)$$

Algorithm 3 (Euclid)**.** *The following listing specifies an algorithm for computing the greatest common divisor of two integer numbers.*

Listing 1.2.9 Algorithm of Euclid

```
1  unsigned int gcd_with_modulo(unsigned int a, unsigned int b)
2  {
3      while( b!=0)
4      {
5          auto h = a%b; //
6          a = b;
7          b=h;
8      }
9      return a;
10 }
```

Theorem 4. *Algorithm 3 is correct.*

Proof. Again, we need a loop invariant which is: "Before, throughout, and after the loop, it remains true that $gcd(a, b) = gcd(a_0, b_0)$, where a_0, b_0 refer to the initial values of a and b passed as parameters. Before the loop, the invariant is obviously true as $a = a_0$, $b = b_0$. Lemma 3 proofs that it remains true in Lines 6-7 (proof by induction!). As $0 \leq h < b$ we conclude that b decreases its value ultimately reaching 0 terminating the loop. And $gcd(a, 0) = a$ concludes the proof. □

Remark. *For the complexity of this algorithm, see Task 1.2 and 1.3. For a classical version without division, see Task 1.1.*

Task 1.1. . *There is a traditional version of Euclid's algorithm which iteratively reduces the longer length by the shorter one. One formulation as (pseudo-)code could be*

```
1  unsigned int gcd_without_division(unsigned int a, unsigned int b)
2  {
3      if (a == 0)
4       return b;
5      while (b!= 0)
6      {
7          if (a>b) {
8              a -= b;
9          }else{
10             b-=a;
11         }
12     }
13     return a;
14 }
```

- *Proof correctness of this algorithm.*

- *Express the runtime exactly as well as in O-notation.*
- *Implement this in MATLAB.*

Task 1.2. *The runtime analysis of Euclids modern algorithm is a bit tricky. Therefore, implement a version which counts steps and draw an image in MATLAB (x-axis for a, y-axis for b, color for the number of steps) and interpret it.*

Task 1.3. *The theorem of Lamé gives an interesting bound formulation. If the algorithm needs n steps for $gcd(a, b)$, then $b \geq fib(n)$, where fib refers to Fibonacci numbers we will discuss a bit later. In addition, this also links the compelxity to another fundamental number, the Golden Ratio: $fib(n) \approx \Phi^n$ where $\Phi \approx 1.618$. Conclude from this information just given that the complexity of the algorithm is bound by $O(\log n)$.*

1.3 MATLAB

1.3.1 Basics

This section is currently available as slides. We learn how to formulate programs, loops and conditions, how to work with matrices, what slicing is, how images are represented as matrices, and some basic plotting.

1.3.2 Turtle Graphics

This section currently exists as slides only.

1.3.3 Discrete-Time Simulation of Movement

This section currently exists as slides only.

1.3.4 The MATLAB language

Iy this chapter, we introduce basic concepts of the MATLAB language which is a widely-used engineering language especially suited to numerical problems that can be expressed in terms of vectors and matrices. In addition to the commercial software MATLAB, there is an open source implementation of the core language with the name Octave, yet, part of the power of MATLAB comes from the domain-specific toolboxes and integrations with lab devices. For this lecture, however, as we concentrate on the programming language, Octave is sufficient and sometimes it might be easier to use due to its open source nature.

As we have already seen, programming languages evolve from sensible notations of algorithms. The first programming languages provide exactly what an underlying computer is able to do, and more complex languages are derived by introducing advanced patterns like loops or functions that can simplify the source code. The main reason for this is to trade-off between optimality in terms of performance or specificity and in terms of readability of a software. This chapter contains a walk-through of the most important concepts to write MATLAB programs, but note that it is just a reference. The only way of learning to use programming languages is by writing programs yourself. Therefore, we give a handful of tasks at the end of each chapter.

Basic Program Structure

A MATLAB program consists of a sequence of statements each of which is terminated with a newline, comma or semicolon. All statements that end with a newline or comma will print their result to the MATLAB console. If you want to suppress this output, you end the statement with a semicolon. The most basic statement is the assignment statement, where on the left hand side a name is given (usually starting with a character) and on the right hand side an expression is given. The simplest