Programming with Maxima

Prof. Dr. Wolfram Koepf

Universität Kassel

http://www.mathematik.uni-kassel.de/~koepf

October 7, 2017, 9:30
Douala, Cameroon

Abstract

Topics of this lecture

Abstract

Topics of this lecture

- In this lecture, we will discuss programming techniques using Maxima.

Abstract

Topics of this lecture

- In this lecture, we will discuss programming techniques using Maxima.
- We start with iterative constructs like loops and conditions, and continue with recursive programming.

Abstract

Topics of this lecture

- In this lecture, we will discuss programming techniques using Maxima.
- We start with iterative constructs like loops and conditions, and continue with recursive programming.
- Remember programming and the divide-and-conquer paradigm will be discussed.

Abstract

Topics of this lecture

- In this lecture, we will discuss programming techniques using Maxima.
- We start with iterative constructs like loops and conditions, and continue with recursive programming.
- Remember programming and the divide-and-conquer paradigm will be discussed.
- The efficiency of a program and of an algorithm is an important issue.

Abstract

Topics of this lecture

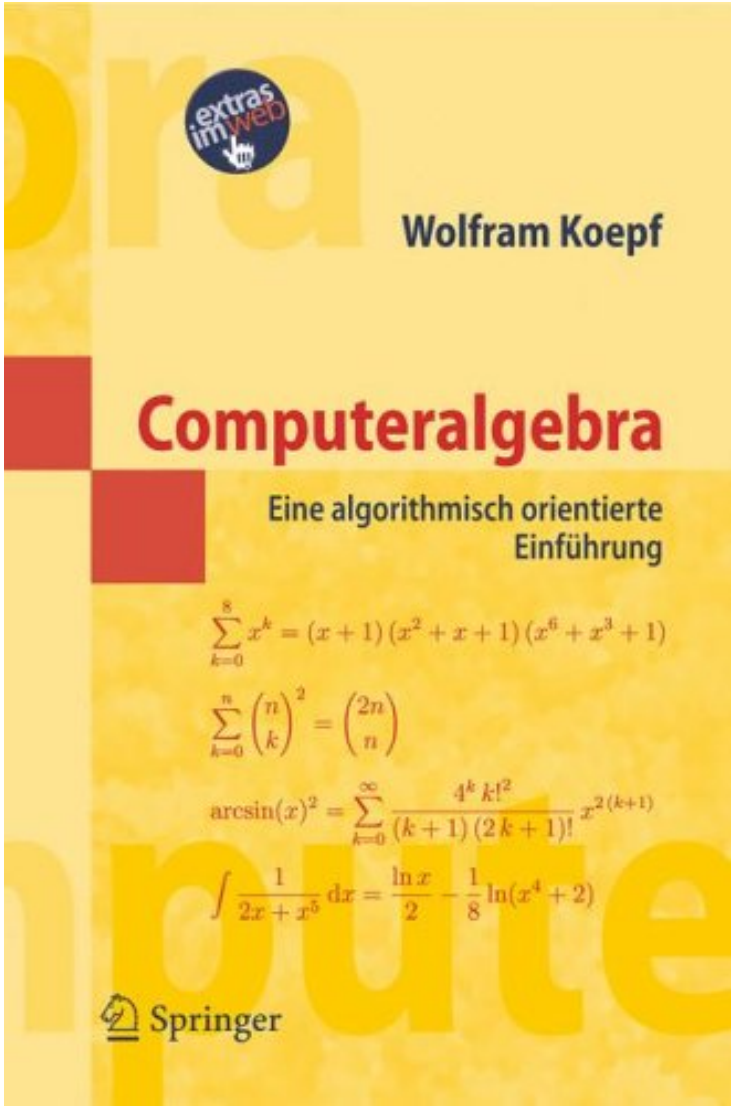
- In this lecture, we will discuss programming techniques using Maxima.
- We start with iterative constructs like loops and conditions, and continue with recursive programming.
- Remember programming and the divide-and-conquer paradigm will be discussed.
- The efficiency of a program and of an algorithm is an important issue.
- The programming techniques considered will be practised by the participants in the subsequent tutorial.

Text Book on Computer Algebra

- This talk corresponds essentially to Chapter 2 of my text book

  Computeralgebra, Springer, Berlin/Heidelberg, 2006
- The text book uses *Mathematica*, but all sessions are also given in *Maple* and Maxima.
- Web page

  www.computer-algebra.org
- This German language book is in the translation process to English.



Iterative Programming

Conditions

Iterative Programming

Conditions

- We begin with the case distiction.

Iterative Programming

Conditions

- We begin with the case distiction.
- In Maxima this is the if-then-else construct.

Iterative Programming

Conditions

- We begin with the case distiction.
- In Maxima this is the if-then-else construct.
- Start Maxima

Iterative Programming

Conditions

- We begin with the case distiction.
- In Maxima this is the if-then-else construct.
- Start Maxima

Loops

- The for loop is used if the number of steps is known.

Iterative Programming

Conditions

- We begin with the case distiction.
- In Maxima this is the if-then-else construct.
- Start Maxima

Loops

- The for loop is used if the number of steps is known.
- Otherwise one can use while.

Iterative Programming

Conditions

- We begin with the case distiction.
- In Maxima this is the if-then-else construct.
- Start Maxima

Loops

- The for loop is used if the number of steps is known.
- Otherwise one can use while.
- Examples of high level routines are makelist, apply, product, sum, map, ...

Iterative Programming

Conditions

- We begin with the case distiction.
- In Maxima this is the if-then-else construct.
- Start Maxima

Loops

- The for loop is used if the number of steps is known.
- Otherwise one can use while.
- Examples of high level routines are makelist, apply, product, sum, map, ...

Programs

One can combine those computation blocks in a program for general use.

Recursive Programming

Factorial

Recursive Programming

Factorial

- The factorial function can also be defined by the recurrence

$$n! = n \cdot (n-1)! \quad \text{and} \quad 0! = 1 \ .$$

Recursive Programming

Factorial

- The factorial function can also be defined by the recurrence

$$n! = n \cdot (n-1)! \quad \text{and} \quad 0! = 1 \ .$$

- A recursive function calls itself and needs an initial value as terminating condition.

Recursive Programming

Factorial

- The factorial function can also be defined by the recurrence

$$n! = n \cdot (n-1)! \quad \text{and} \quad 0! = 1 \ .$$

- A recursive function calls itself and needs an initial value as terminating condition.

Fibonacci numbers

- The Fibonacci numbers F_n are given by

$$F_n = F_{n-1} + F_{n-2} \quad \text{and} \quad F_0 = 0, \quad F_1 = 1 \ .$$

Recursive Programming

Factorial

- The factorial function can also be defined by the recurrence

$$n! = n \cdot (n-1)! \quad \text{and} \quad 0! = 1 \ .$$

- A recursive function calls itself and needs an initial value as terminating condition.

Fibonacci numbers

- The Fibonacci numbers F_n are given by

$$F_n = F_{n-1} + F_{n-2} \quad \text{and} \quad F_0 = 0, \quad F_1 = 1 \ .$$

- If implemented similarly as the factorial function, the computation times are very slow. Why?

Remember Programming

Fibonacci numbers

Remember Programming

Fibonacci numbers

- The reason for the low efficiency is that this program computes each previous Fibonacci number several times!

Remember Programming

Fibonacci numbers

- The reason for the low efficiency is that this program computes each previous Fibonacci number several times!
- Assume, we want to compute the number F_{30}. Then the number F_{29} is called exactly once. However, already the number F_{28} is computed twice, and F_{27} is computed three times! And this goes on.

Remember Programming

Fibonacci numbers

- The reason for the low efficiency is that this program computes each previous Fibonacci number several times!
- Assume, we want to compute the number F_{30}. Then the number F_{29} is called exactly once. However, already the number F_{28} is computed twice, and F_{27} is computed three times! And this goes on.

Number of calls

- For the computation of F_n the value F_0 is called F_n times!

Remember Programming

Fibonacci numbers

- The reason for the low efficiency is that this program computes each previous Fibonacci number several times!
- Assume, we want to compute the number F_{30}. Then the number F_{29} is called exactly once. However, already the number F_{28} is computed twice, and F_{27} is computed three times! And this goes on.

Number of calls

- For the computation of F_n the value F_0 is called F_n times!
- One can prove that F_n grows exponentially: $F_n \sim \left(\frac{1+\sqrt{5}}{2}\right)^n$.

Remember Programming

Fibonacci numbers

- The reason for the low efficiency is that this program computes each previous Fibonacci number several times!
- Assume, we want to compute the number F_{30}. Then the number F_{29} is called exactly once. However, already the number F_{28} is computed twice, and F_{27} is computed three times! And this goes on.

Number of calls

- For the computation of F_n the value F_0 is called F_n times!
- One can prove that F_n grows exponentially: $F_n \sim \left(\frac{1+\sqrt{5}}{2}\right)^n$.
- This explains the low efficiency.

Remember Programming

Fibonacci numbers

- The reason for the low efficiency is that this program computes each previous Fibonacci number several times!
- Assume, we want to compute the number F_{30}. Then the number F_{29} is called exactly once. However, already the number F_{28} is computed twice, and F_{27} is computed three times! And this goes on.

Number of calls

- For the computation of F_n the value F_0 is called F_n times!
- One can prove that F_n grows exponentially: $F_n \sim \left(\frac{1+\sqrt{5}}{2}\right)^n$.
- This explains the low efficiency.
- Therefore, to get linear efficiency, one should remember the previously computed numbers. This resolves this issue.

Divide-and-Conquer Programming

Fibonacci numbers revisited

Divide-and-Conquer Programming

Fibonacci numbers revisited

- The internal Maxima function fib is still much faster! Why?

Divide-and-Conquer Programming

Fibonacci numbers revisited

- The internal Maxima function fib is still much faster! Why?
- The previously used algorithms using the given recurrence equation cannot do better.

Divide-and-Conquer Programming

Fibonacci numbers revisited

- The internal Maxima function fib is still much faster! Why?
- The previously used algorithms using the given recurrence equation cannot do better.
- Hence for higher efficiency a faster algorithm is needed!

Divide-and-Conquer Programming

Fibonacci numbers revisited

- The internal Maxima function fib is still much faster! Why?
- The previously used algorithms using the given recurrence equation cannot do better.
- Hence for higher efficiency a faster algorithm is needed!
- The Divide-and-conquer paradigm is to compute a problem of size n by computing several of size $n/2$.

Divide-and-Conquer Programming

Fibonacci numbers revisited

- The internal Maxima function fib is still much faster! Why?
- The previously used algorithms using the given recurrence equation cannot do better.
- Hence for higher efficiency a faster algorithm is needed!
- The Divide-and-conquer paradigm is to compute a problem of size n by computing several of size $n/2$.

Divide-and-conquer algorithm for Fibonacci numbers

The formulas that can be used for this purpose, are given by

$$F_{2n} = F_n \cdot (F_n + 2\,F_{n-1}) \quad \text{for even } k = 2n$$

and

$$F_{2n+1} = F_{n+1}^2 + F_n^2 \quad \text{for odd } k = 2n+1\ .$$

Efficient Computation of Modular Powers

Divide-and-conquer algorithm

Efficient Computation of Modular Powers

Divide-and-conquer algorithm

- The modular power is defined as

$$a^n \pmod{p} .$$

Efficient Computation of Modular Powers

Divide-and-conquer algorithm

- The modular power is defined as

$$a^n \pmod{p} .$$

- The critical value is of course the exponent n. Even for moderate size n the number a^n is larger than the computer memory although the modular power is smaller than p.

Efficient Computation of Modular Powers

Divide-and-conquer algorithm

- The modular power is defined as

$$a^n \pmod{p} .$$

- The critical value is of course the exponent n. Even for moderate size n the number a^n is larger than the computer memory although the modular power is smaller than p.
- Recursive formulation of the divide-and-conquer computation of the modular power:

Efficient Computation of Modular Powers

Divide-and-conquer algorithm

- The modular power is defined as

$$a^n \pmod{p} .$$

- The critical value is of course the exponent n. Even for moderate size n the number a^n is larger than the computer memory although the modular power is smaller than p.
- Recursive formulation of the divide-and-conquer computation of the modular power:
  - $a^0 \mod p = 1$

Efficient Computation of Modular Powers

Divide-and-conquer algorithm

- The modular power is defined as

$$a^n \pmod{p} .$$

- The critical value is of course the exponent n. Even for moderate size n the number a^n is larger than the computer memory although the modular power is smaller than p.
- Recursive formulation of the divide-and-conquer computation of the modular power:
  - $a^0 \mod p = 1$
  - $a^n \mod p = (a^{n/2} \mod p)^2 \mod p$ for even n

Efficient Computation of Modular Powers

Divide-and-conquer algorithm

- The modular power is defined as

$$a^n \pmod{p} .$$

- The critical value is of course the exponent n. Even for moderate size n the number a^n is larger than the computer memory although the modular power is smaller than p.
- Recursive formulation of the divide-and-conquer computation of the modular power:
  - $a^0 \mod p = 1$
  - $a^n \mod p = (a^{n/2} \mod p)^2 \mod p$ for even n
  - $a^n \mod p = (a^{n-1} \mod p) \cdot a \mod p$ for odd n

Efficient Computation of Modular Powers

Divide-and-conquer algorithm

- The modular power is defined as

$$a^n \pmod{p} .$$

- The critical value is of course the exponent n. Even for moderate size n the number a^n is larger than the computer memory although the modular power is smaller than p.
- Recursive formulation of the divide-and-conquer computation of the modular power:
  - $a^0 \mod p = 1$
  - $a^n \mod p = (a^{n/2} \mod p)^2 \mod p$ for even n
  - $a^n \mod p = (a^{n-1} \mod p) \cdot a \mod p$ for odd n
- Question: How does an iterative version of this algorithm work?

Application: Prime Number Test

Fermat's Little Theorem

Application: Prime Number Test

Fermat's Little Theorem

For a prime number $p \in \mathbb{P}$ and $a \in \mathbb{Z}$ the relation

$$a^p = a \pmod{p}$$

is valid, or respectively

$$a^{p-1} = 1 \pmod{p}\,, \quad \text{if } \mathrm{GCD}(a, p) = 1\,.$$

Application: Prime Number Test

Fermat's Little Theorem

For a prime number $p \in \mathbb{P}$ and $a \in \mathbb{Z}$ the relation

$$a^p = a \pmod{p}$$

is valid, or respectively

$$a^{p-1} = 1 \pmod{p}\,, \quad \text{if } \mathrm{GCD}(a, p) = 1\,.$$

Fermat test

- If Fermat's relation is *not* valid for a number $a \in \mathbb{Z}$ (called a witness for non-primality of p), then p cannot be a prime number!

Application: Prime Number Test

Fermat's Little Theorem

For a prime number $p \in \mathbb{P}$ and $a \in \mathbb{Z}$ the relation

$$a^p = a \pmod{p}$$

is valid, or respectively

$$a^{p-1} = 1 \pmod{p}\,, \quad \text{if } \mathrm{GCD}(a, p) = 1\,.$$

Fermat test

- If Fermat's relation is *not* valid for a number $a \in \mathbb{Z}$ (called a witness for non-primality of p), then p cannot be a prime number!
- Note that we get this information without any knowledge about a concrete factor of p!

Many thanks for your interest!