Advanced practical **Programming** for Scientists

Thorsten Koch

Zuse Institute Berlin

TU Berlin

SS2017

Planned topics

Overview: Imperative, OOP, Functional, side effects, thread safe, Design by contract

Design: Information hiding, Dependencies, Coding style, Input checking, Error Handling

Design: Overall program design, Data structures, Memory allocation

Tools: git, gdb, undodb, doxygen

Languages and correctness: Design errors/problems C/C++, C89 / C99 / C11, Compiler switches, assert, flexelint, FP, How to write correct programs

Testing: black-box, white-box, unit tests, regression tests, error tests, speed tests

Tools: gcov, jenkins, cmake, doxygen, make, gprof, valgrind, cppcheck, clang

Software metrics: Why, Examples, Is it useful? Control flow complexity

Parallel programming: OpenMP, MPI, pthreads, OpenCL

Program optimization: Code optimization, linking libraries

How to design large programs

Code-shootout and comparison: Documentation, Release management

All information is subject to change.

Vorlesung 12.05: Digital Future

Please register yourself via

https://science-match.tagesspiegel.de/the-digital-future-may-2017

Voucher/VIP Code: *future17-1*

Administrative affairs

All emails related to this lecture should start with **APPFS** in the subject.

Everybody participating in this lecture, please send an email to <rehfeldt@zib.de> with your Name and Matrikel-Nr.

We will setup a mailing list for announcements and discussion.

Everything is here http://www.zib.de/koch/lectures/ss2017_appfs.php

If you need the certificate, regular attendance, completion of homework assignments, and in particular participation in our small programming project is expected. Grades will be based on the outcome and a few questions about it ☺ .

No groups.

Exercises (plan)

Exercises part I: Reading and writing Data

1. 21.04. Reading csv
2. 28.04. Reading xml
3. 05.05. Reading binary
4. 05.05. Checking input data

Exercise part II: Time series prediction

5. 26.05. Making it work 1 (make)
6. 02.06. Making it work 2 (Documentation, fixing input data)
7. 09.06. Testing
8. 16.06. Measuring performance, serializing data structures
9. 23.06. Analyzing code quality
10. 30.06. Linking third party libraries (static vs. shared)
11. 07.07. Making it better

How to submit exercises

- **You can submit the exercises in any programming language.**
- It has to run on my Rasberry-pi running Linux
- Example solutions will be provided at least in one of the following languages: C99, C++14, python3, Ada2012.
- If you use anything else, you are on your own.
- I will judge your code nevertheless.

Register at github.com

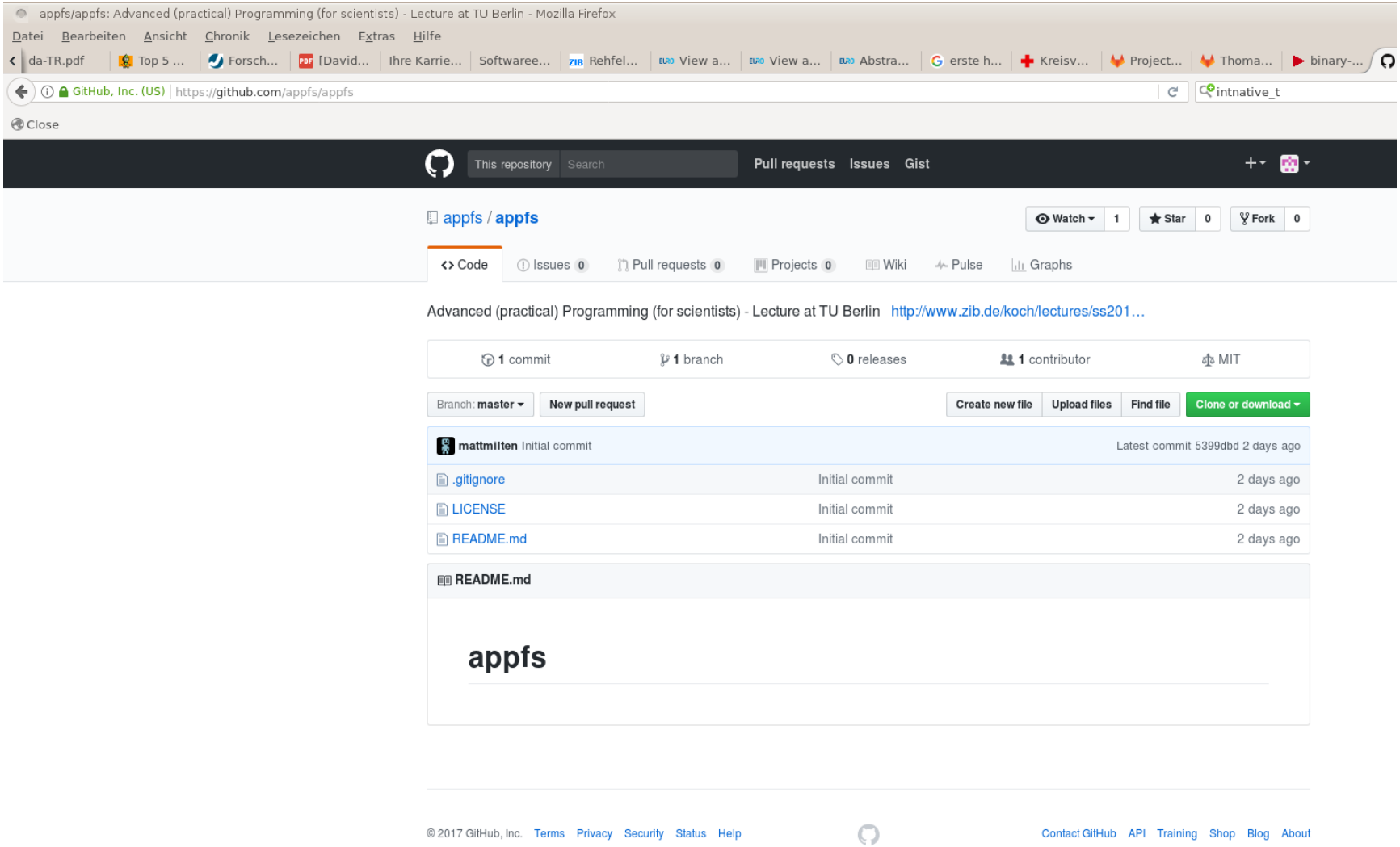
Go to https://github.com/mattmilten/appfs

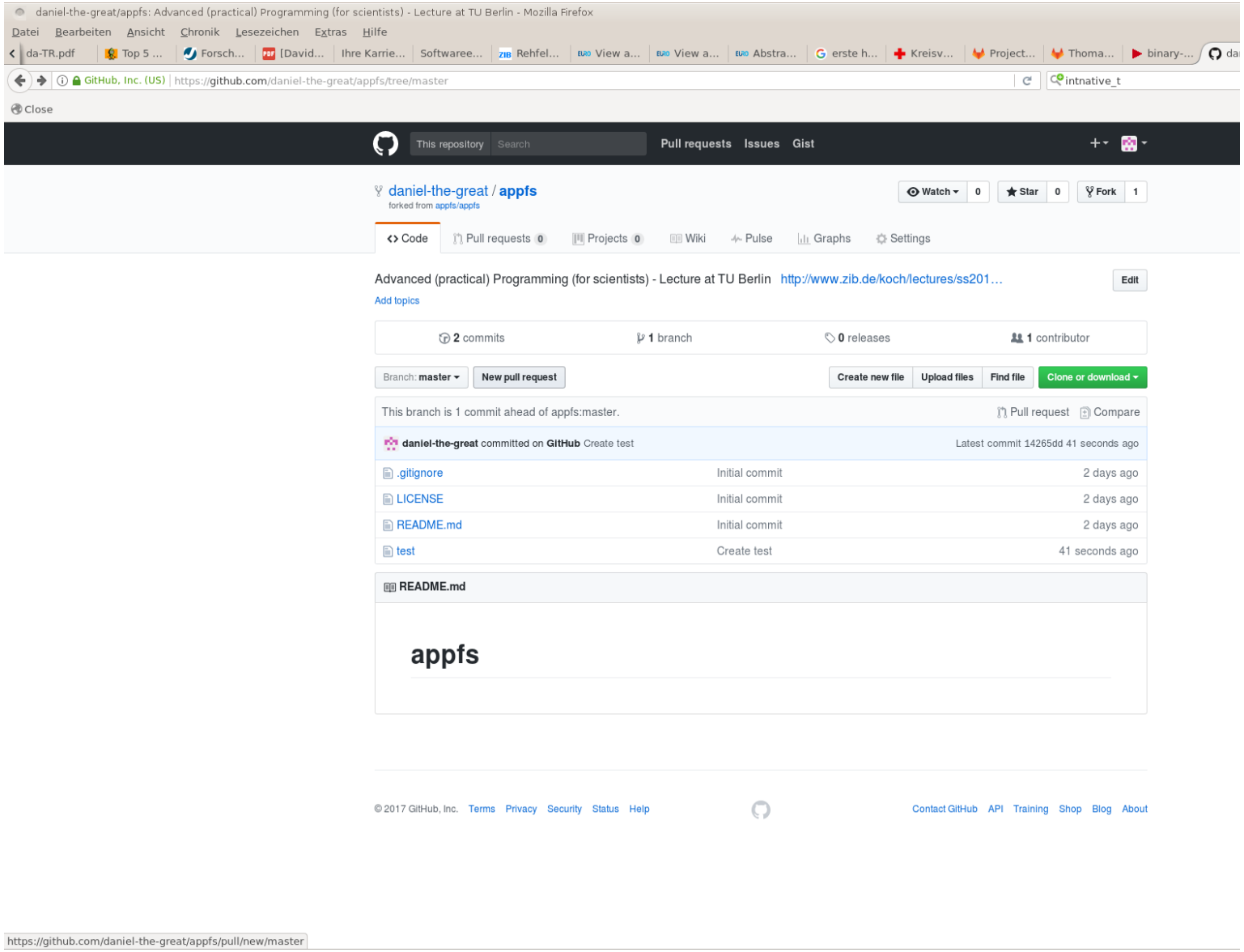
Fork

Change

Pull request

Create a new folder with your name and then make a pull request


appfs/appfs: Advanced (practical) Programming (for scientists) - Lecture at TU Berlin - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
GitHub, Inc. (US) https://github.com/appfs/appfs
Close
This repository Search
Pull requests Issues Gist
appfs / appfs
Watch 1 Star 0 Fork 0
Code Issues 0 Pull requests 0 Projects 0 Wiki Pulse Graphs
Advanced (practical) Programming (for scientists) - Lecture at TU Berlin http://www.zib.de/koch/lectures/ss201…
1 commit 1 branch 0 releases 1 contributor MIT
Branch: master New pull request
Create new file Upload files Find file Clone or download
mattmilten Initial commit
Latest commit 5399dbd 2 days ago
.gitignore Initial commit 2 days ago
LICENSE Initial commit 2 days ago
README.md Initial commit 2 days ago
README.md
appfs
© 2017 GitHub, Inc. Terms Privacy Security Status Help
Contact GitHub API Training Shop Blog About


daniel-the-great/appfs: Advanced (practical) Programming (for scientists) - Lecture at TU Berlin - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
GitHub, Inc. (US) https://github.com/daniel-the-great/appfs/tree/master
Close
This repository Search
Pull requests Issues Gist
daniel-the-great / appfs
forked from appfs/appfs
Watch 0 Star 0 Fork 1
Code Pull requests 0 Projects 0 Wiki Pulse Graphs Settings
Advanced (practical) Programming (for scientists) - Lecture at TU Berlin http://www.zib.de/koch/lectures/ss201…
Edit
Add topics
2 commits 1 branch 0 releases 1 contributor
Branch: master New pull request
Create new file Upload files Find file Clone or download
This branch is 1 commit ahead of appfs:master.
Pull request Compare
daniel-the-great committed on GitHub Create test
Latest commit 14265dd 41 seconds ago
.gitignore Initial commit 2 days ago
LICENSE Initial commit 2 days ago
README.md Initial commit 2 days ago
test Create test 41 seconds ago
README.md
appfs
© 2017 GitHub, Inc. Terms Privacy Security Status Help
Contact GitHub API Training Shop Blog About
https://github.com/daniel-the-great/appfs/pull/new/master

Attitudes

Algorithm engineering refers to the process required to transform a pencil-and-paper algorithm into a robust, efficient, well tested, and easily usable implementation.

— Bader, Moret, Sanders

Real Programmers don't comment their code. If it was hard to write, it should be hard to understand and harder to modify.

— Fortune (6)

Beware of bugs in the above program. I have only proved it correct, not tried it.

— D.E.Knuth

*The single most important rule of testing is to **do** it.*

— Kernighan, Pike

Laws, Rules, and Objectives

- Laws describe the feasible region of a solution
- Objectives (are directions) describing which solution to prefer
- Rules try to exclude (rule out) bad solutions

“Look, that's why there's rules, understand?
So that you think before you break 'em.”
— Terry Pratchett, Thief of Time

In general I recommend to follow rules, until you are sure you understand what the rules wanted to achieve and you believe your objectives are served better by not obeying the rule.

But think first!

If the union of all rules defines an empty space, resist the temptation to change the objective to look for a solution that violates the least number of rules. Slack solutions usually have a horrible objective value.

Examples

The speed of light *c* in vacuum is a universal physical constant. Its exact value is 299792458 m/s. According to special relativity, *c* is the maximum speed at which all conventional matter and hence all known forms of information in the universe can travel. (Wikipedia)

"**Everything should be made as simple as possible, but no simpler.**"

— Albert Einstein

Don't use goto!

See https://softwareengineering.stackexchange.com/questions/125715/do-we-still-have-a-case-against-the-goto-statement

A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.

— Antoine de Saint-Exupery

Imperative Programming

In computer science terminology, **imperative programming** is a programming paradigm that describes computation in terms of statements that change a program state. In much the same way that imperative mood in natural languages expresses commands to take action, imperative programs define sequences of commands for the computer to perform.

Imperative programming, http://en.wikipedia.org/w/index.php?title=Imperative_programming&oldid=624302389 (last visited Sept. 21, 2014)

.

Imperative Programmierung ist ein Programmierparadigma. Danach werden Programme so entwickelt, dass „ein Programm aus einer Folge von Anweisungen besteht, die vorgeben, in welcher Reihenfolge was vom Computer getan werden soll “.

Die imperative Programmierung ist das am längsten bekannte Programmierparadigma. Diese Vorgehensweise war, bedingt durch den Sprachumfang früherer Programmiersprachen, ehemals die klassische Art des Programmierens. Sie liegt dem Entwurf von vielen Programmiersprachen, zum Beispiel ALGOL, Fortran, Pascal, Ada, PL/I, Cobol, C und allen Assemblersprachen zugrunde.

Seite „Imperative Programmierung“. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 20. September 2014, 14:27 UTC. URL: http://de.wikipedia.org/w/index.php?title=Imperative_Programmierung&oldid=134202110 (Abgerufen: 21. September 2014, 20:41 UTC)

Imperative Programming

I am in command!

input->putput->output

Data separated from instructions (more or less ☺ as instructions are data)

Von-Neumann Architecture/Stored-Program-Computer

Access Memory, do computations incl. conditional, PC program counter

-> allows: goto (jump), if (conditional), while (loop)

1. Do it!
2. Anyway!

Structured programming vs. goto

Blocks, subroutines, scopes.

Object-Oriented Programming

Building your own world of objects

Object-oriented programming attempts to provide a model for programming based on objects. OO programming integrates code and data using the concept of an “object”. An object is an abstract data type with the addition of polymorphism and inheritance.

An object has both state (data) and behavior (code).

-> Information hiding

-> polymorphism comes naturally

-> single vs. multiple inheritance.

-> templates and generics

Functional Programming

The way mathematicians think.

Functional programming is a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data.

In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result f(x) both times.

Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

-> side effects / mutable state -> rand(), getchar(), putchar()

-> call by value, call by reference

-> thread safeness -> errno

Programming Paradigms

- **Imperative programming** – defines computation as statements that change a program state (Assembler)
- Procedural programming, structured programming – specifies the steps the program must take to reach the desired state (C, Pascal, Fortran 77)
- **Functional programming** – treats computation as the evaluation of mathematical functions and avoids state and mutable data (Lisp, ML, Haskell, Erlang, Ocaml)
- **Object-oriented programming** (OOP) – organizes programs as objects: data structures consisting of datafields and methods together with their interactions (Smalltalk, C++, Java, Eiffel)
- **Declarative programming** – defines computation logic without defining its control flow (Prolog)
- **Event-driven programming** – the flow of the program is determined by events, such as sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads (JavaScript)

Have you ever ...

- wasted a lot of time coding the wrong algorithm?
- used a data structure that was much too complicated?
- tested a program but missed an obvious problem?
- spent a day looking for a bug you should have found in five minutes?
- needed to make a program run three times faster and use less memory?
- struggled to move a program from one architecture to another?
- tried to make a modest change in someone else's program?
- rewritten a program because you couldn't understand it?

Was it fun?

From: Kernighan,Pike „The practise of programming“

Design goals

These include

- **simplicity**, which keeps programs short and manageable;
- **clarity**, which makes sure they are easy to understand, for people as well as machines;
- **generality**, which means they work well in a broad range of situations and adapt well as new situations arise; and
- **automation**, which lets the machine do the work for us, freeing us from mundane tasks.

From: Kernighan,Pike „The practise of programming“

Winner OSC1990 Best small programm

```
v,i,j,k,l,s,a[99];
main()
{
for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,
j+=(v=j<s&&(!k&&!!printf(2+"\n\n%c"-(!l<<!j),
" #Q"[l^v?(l^j)&1:2])&&++l||a[i]<s&&v&&v-i+j&&
v+i-j))&&!(l%=s),
v||(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);
}
```

What might it possibly do?

How to achieve this

- Be able to follow control flow (-imperative, +structured, -OO, +functional)
- Structuring programs into units (-imperative, +structured, +OO)
- Minimize dependencies (between components)
  - Minimize scope
  - Minimize side effects (+functional)
  - Data hiding (+OO)
  - How about things happening automatic?
    (member functions in C++, Garbage collection)
  - Being clever?: while(*s++ = *t++);
  - DbC

Something to think about

```
int data[10000]; // all 0..255
long long fun = 0;
unsigned i;
[…]
int t = (data[i] - 128) >> 31;
fun += ~t & data[i];
```

This is not the way

```
/* The Computer Language Benchmarks Game http://benchmarksgame.alioth.debian.org/
Contributed by Dmitry Vyukov
*/
#define _GNU_SOURCE
#include <stdlib.h>
[…]
#define CL_SIZE 64

void* cache_aligned_malloc(size_t sz)
{
    char*                       mem;
    char*                       res;
    void**                      pos;

    mem = (char*)malloc(sz + 2 * CL_SIZE);
    if (mem == 0)
        exit(1);
    res = (char*)((uintptr_t)(mem + CL_SIZE) & ~(CL_SIZE - 1));
    pos = (void**)(res - sizeof(void*));
    pos[0] = mem;
    return res;
}
```

Design by Contract

(DbC), is an approach for designing software.

It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, post-conditions and invariants.

These specifications are referred to as “contracts”, in accordance with a conceptual metaphor with the conditions and obligations of business contracts.

Pre-conditions

Post-conditions

Invariants

Exercise 1: Setup

Please check out the data for this exercise located here:

https://github.com/mattmilten/appfs

You will find a program named **`ex1_gen`** used to generate the input data.

Run this program as follows:

```
./ex1_gen 100000000 >ex1.dat
```

The file **`ex1.dat`** should then contain roughly 100 million lines.

You can check by

```
wc -l ex1.dat
```

Size should be around 2GB.

The first number is 123456789.

Exercise 1: File Format

Each line should consists of

- a sequence-number,
- a location (1 or 2), and
- a floating point value > 0.

Empty lines are allowed. Comment lines start with a ”#”.

Everything after a ”’#” on a line should be ignored.

Read in the data and compute the geometric mean for each location.

Be aware that there might be some errors in the data.

Exercise 1: What to do

Write a program named **`ex1`** in C or your favorite language, which

1. Reads in the data from **`ex1.dat`**

2. Compute the **Geometric Mean** for both locations
   Output should look like:

   File: ex1.dat with 100001235 lines
   Valid values Loc1: 50004598 with GeoMean: 36.7817
   Valid values Loc2: 49994703 with GeoMean: 36.7825

Exercise 3: How to report

1. Check in the source code into github as explained

2. use **`time ex1 ex1.dat`**
   to get the runtimes of your program

3. Copy the output of your program

Send the output of `time`, and your `ex1` with

subject of APPFS ex1 vorname nachname

per email to <koch@zib.de>

Deadline: 27.04. 16 Uhr (earlier would be better)