PC BENCHMARK

Student: Lazar Vlad Adrian

Group: 30433

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA
ROMANIA

Contents

1. Introduction

1.1 Project Requirements

Develop a benchmark application for personal computers, which will measure the following performances of the system under test:

- The type of the processor
- The frequency of the processor
- The dimension of the memory
- The transfer speed of a data block
- The execution speed of arithmetic and logic instructions

1.2 Project Context

Computer benchmarks provide a standard and fair way to test different components of the whole computer system. They vary from graphic benchmarking, logic and arithmetic operations benchmarking, memory block transfer speed, etc.

The proposed PC benchmark application is only a prototype, which is meant for a university project. It could be used, by any PC owner wanting to stress test the power of his machine's processor, with a series of performed operations.

Also, the benchmark will provide a short description of the underlying hardware, so the user will have access to information like frequency, memory size, processor type, so it could also be used by students who want to learn to correlation of the hardware and structure of the PC to the performance it provides.

1.3Objectives

My goal is to implement a basic computer benchmark for a PC, which will address all the main aspects stated in the requirements. Also, the benchmark will state following information about the underlying hardware when opened. For implementations, I will use either C++ or Java

programming languages, and the user interface will be designed either in Qt or JavaFX, depending on the language I will chose for carrying out the development of the project.

The main characteristics of a good benchmark , that I will aim to follow, are:

- Relevance : How close the benchmark behaviour relates to the ones of interest of the consumers
- Reproducibility: produce similar results when the test is run with the same configuration
- Fairness: Allowing different test configurations to compete on their merits (very important in industry)
- Verifiability: Providing confidence that the test results is accurate.
- Usability: avoiding to much dependency on other software & hardware that the users need to run the benchmark.

Weekly plan for project development

WEEK #1 (7TH FEBRUARY)	○ Choose the project
WEEK #2 (21TH MARCH)	○ Establish: 1. Project requirements – what aspects will the benchmark address. 2. Study what is the context of benchmarking on PCs 3. Time table with plan for the future weeks 4. Study some research articles about benchmarking
WEEK #3 (4TH APRIL)	○ Decide on a language for implementations, after finding what C++ and Java can bring to the project. ○ Design a structure of the tests for each of the aspects in the requirements. ○ Test methods of time measurement in C++
WEEK #4 (18THAPRIL)	○ Begin writing the Implementation Chapter of the documentation

	○ Start the implementation of other types of tests (for speed of data transfer, speed of arithmetic & logic operation execution).
WEEK #5 (2ND MAY)	○ Analyse the results of more tests, compare them, come up with a formula to compute the score. Save the tests in some files to be able to create a history of the tests on the machine it runs on ○ Design a user interface for the benchmark.
WEEK #6 (16TH MAY)	○ Refine the interface and solve bugs if they appear during testing. Also, complete the testing and validation of the system. Possibly run on another architecture, import the tests, and compare them. ○ Write the "Test and validation" chapter.
WEEK #7 (1TH JUNE)	○ Complete the "Conclusions" chapter in the documentation.

2. Bibliographic research

Benchmarking is a performance measuring technique used in industry to compare certain products or methodologies by measuring their performance in different aspects.

As detailed in [1], beside their main functionality, benchmarks need to be standardized so to cover a wide area from the products in the same domain. Because of this, they are developed by consortia from multiple companies to assure the fairness of the benchmark tool. The most important consortia of this type on the market in the last decades are SPEC (System Performance Evaluation Consortium) and TCP.

There are two main types of benchmarks in industry, namely the specification-based benchmark and kit based benchmark. A hybrid between these two, is also accepted as basically a third type.

Specification based benchmarks begin with the specification of the business problem for which the benchmark is build, hence they may allow the use of some innovative software to build a very specific benchmark for that business.

For kit based benchmarks, the specification is used only as a design guide to building the kit, but the benchmark is not built solely on the specification, to address only a certain business are.

The workload, also briefly explained in [1], is an essential term when discussing about benchmarks. It is a "task", that the SUT (system under test) must perform, on which the performance of the system are evaluated. A good workload should always be fair and should be generally usable, meaning it should be able to be carried out on almost every CPU present on the market, or else, the benchmark using it would fall out the class of standard benchmarks.

In our case, the workload will be composed of a series of stress tests [2]. The capacities of the monitored CPU will be stressed through complex operations, and stats like cycles needed to perform the operations will be monitored.

3. Analysis

As mentioned in the previous section, stress tests are required to be carried out in order to provide an overview of the CPU performance. So, naturally, the issue of choosing good stress tests comes in mind.

To emphasize the performance of the System under Test, we have chosen the time measurement as performance evaluation method, mainly because memory usage measurement is not an issue with machines today.

The programming language in which the benchmark will be implemented will be C++, because it's a language close to the hardware, and does not depend on a VM like Java, which may influence the measurements done by interfering the task at hand with other processes from the VM.

There are 2 main approaches when measuring time of the execution of a task in C++ : using the <time.h> library and <chrono> library. The first one relies on measuring the clock cycles and then, you can convert this information to seconds. While this method is acceptably accurate, it's drawback is that, during I/O operations, there might be times when not a full clock cycle is used for your application, but it is counted as used. When performing a big loop for example with many I/O operations, the measured result might not reflect accurately the performance of your own program.

The second approach, using <chrono> library does not use the same approach, and is also more accurate on the decimal side of the measured time. It's only downside is that it is available only since C++11.

The tests of the measuring method can be found in the Experiments folder attached to the project.

Literature research and the testimonials in [2], have inspired me to run 3 kind of tests on the CPU:

Integer operation testing

Integer testing consists of operations involving large integer numbers. For this, my choice as a stress test is computing the nth Fibonacci number, as it performs operations on big numbers if the n parameter is given as a big value. Also, I will look for multiplication/division stress tests.

The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377...

1+1=2	13+21=34
1+2=3	21+34=55
2+3=5	34+55=89
3+5=8	55+89=144
5+8=13	89+144=233
8+13=21	144+233=377

The Fibonacci sequence, although working with large integer values, is very well optimised in modern processors, meaning the repeatedly addition of 2 numbers is not a problem anymore for the processors of modern computers.

To still impose a stress test on the CPU, I have implemented a Integer Matrix Multiplication test, where the matrices (quadratic) have huge dimensions from 500x500 to 1000x1000. To also include a division stress test, each matrix element from the multiplication is divided by a division factor.

Floating point operations testing

To stress the pc by demanding floating point operations, approximating the value of PI seems the best choice. There is more than one approach on approximating the famous number, as mentioned both in [2] and [3], but my choice is the one involving the Gregory-Leibniz formula:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} \cdots$$

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

The variation parameter here will be the number of terms from the Leibniz series that will be used for computation.

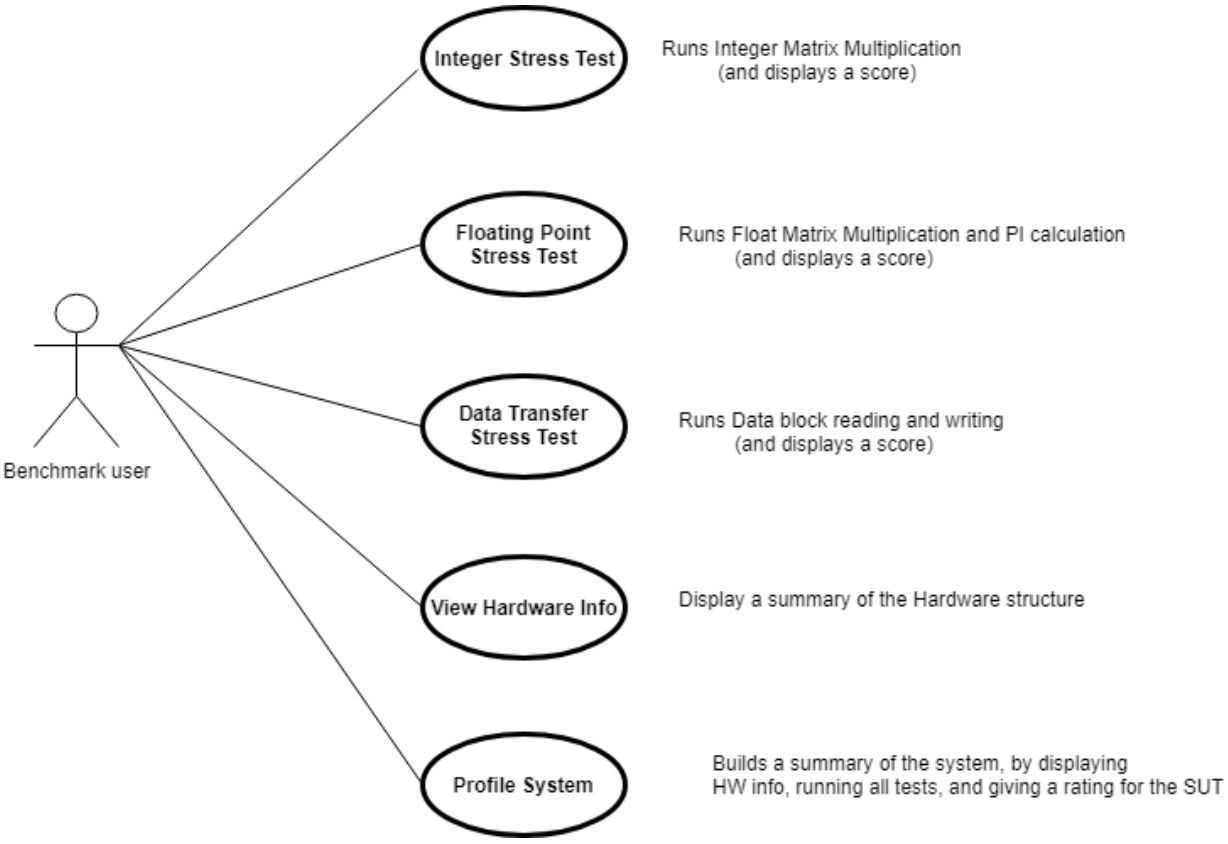
A Floating Point Matrix Multiplication test similar to the Integer Multiplication was also implemented, where the division factor is represented by the PI value, which adds an even bigger stress test for the machine running the code.

The division by PI, the large dimension of the array and the many decimals of the array elements are good elements to cause stress and show how well a machine performs on operations with floating point numbers.

Data block transfer testing

Optionally, if time allows, I will implement testing of the data block transfer speed, by reading and writing large chunks of data in files and monitoring the time taken. Here, I will vary the size of the block transferred.

All these tests will be able to be run either individually or all at the same time (when creating a profile for the current machine). Additionally when profiling the whole computer, an overview of the system hardware information will be provided. Below, a summary of the possible tests is provided in the form of a Use Case Diagram.



4. Project Design

The design of the structure should allow, as requested by the requirements, that a test is stored in memory to create a test history, and compare tests one with another.

For this, my approach is to create C++ classes for all types of tests:

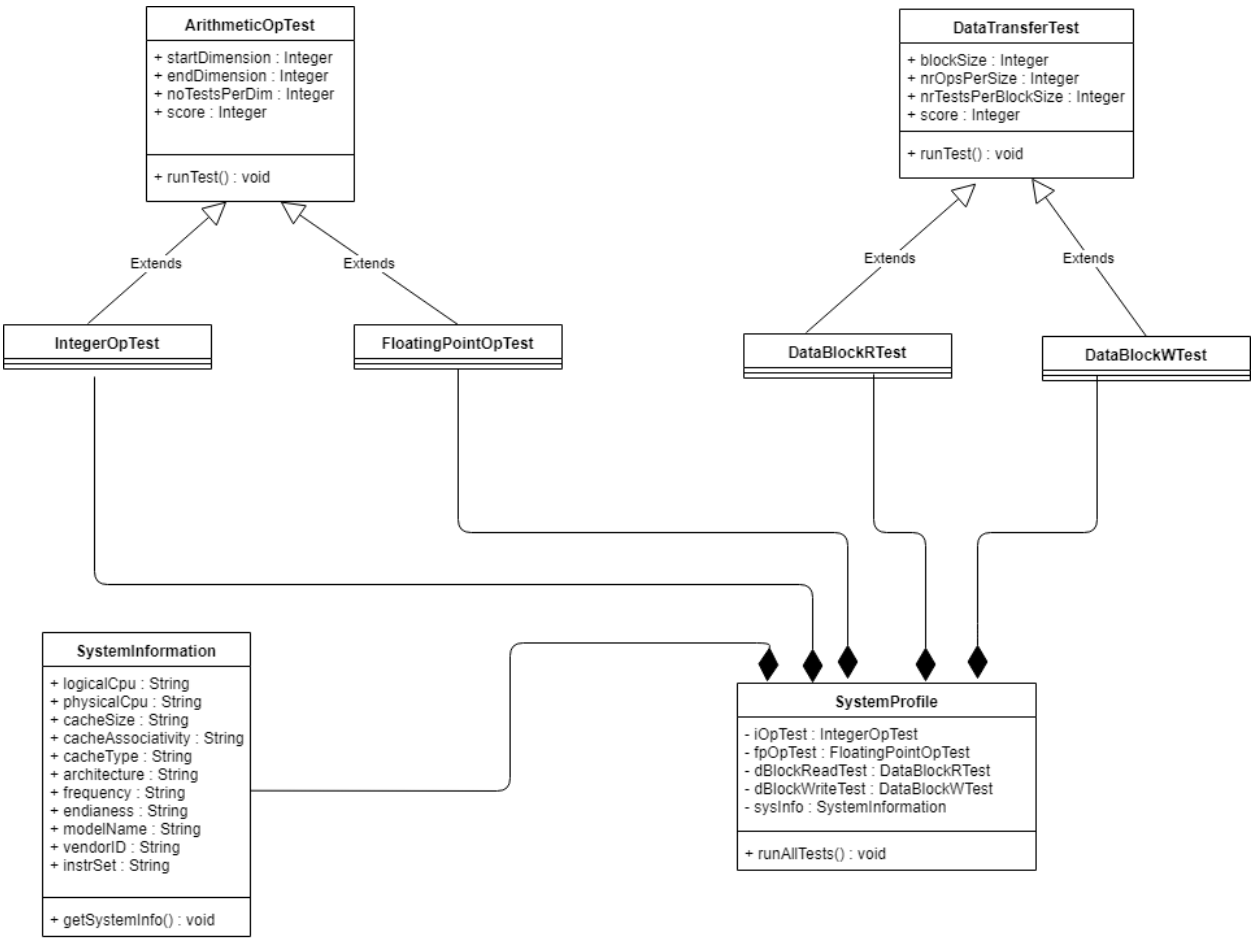
- IntegerOpTest
- FloatingOpTest
- DateBlockRTest
- DataBlockWTest
- SystemInformation
- SystemProfile

A wrapper class, of an SystemProfile test, will contain a list of tests for each of the types, along with their results and varying parameters. This structure is also helpful to then put the information in some tables, in a Qt interface.

Beside the functional tests, my plan is to show a system information summary, upon launching the application. This will be a separate SystemInformation class, which will be associated with the SystemProfile, creating a TestRun. The system information will be gathered with the infoware c++ library [4].

Initially, my goal is to display the results of a test run on the interface, to store the results of a history in a file or database, and if time allows, to create graphics to compare 2 tests.

An overview of the main components of the Benchmark application is presented below in the form of a UML Class Diagram:



TECHNICAL UNIVERSITY OF CLUJ-NAPOCA ROMANIA

5. Implementation

Time Measurement:

As discussed in the Analysis chapter, the times measurement was performed using the <chrono> C++11 library. An example of using the library to measure the execution of some instructions is:

```
auto beg = std::chrono::high_resolution_clock::now();
// some insitructions
auto end = std::chrono::high_resolution_clock::now();
auto dur = std::chrono::duration_cast<std::chrono::microseconds>(end - beg);

cout << "Total time " << (1.0 * dur.count()) * 0.000001  << " seconds" << endl;
```

Matrix operations (Integer / Floating Point):

Matrix operations, for both integer and floating point numbers was implemented using the same pattern. A method AllocateMatrix was created for initializing a matrix with a given dimension, with random numbers. A method DeallocateMatrix was also created to free the alocated space. The MultiplyMatrix is the classic matrix multiplication algorythm, with the addition that each time a

term is multiplied with another, a division operation is also performed to create a higher stress on the CPU.

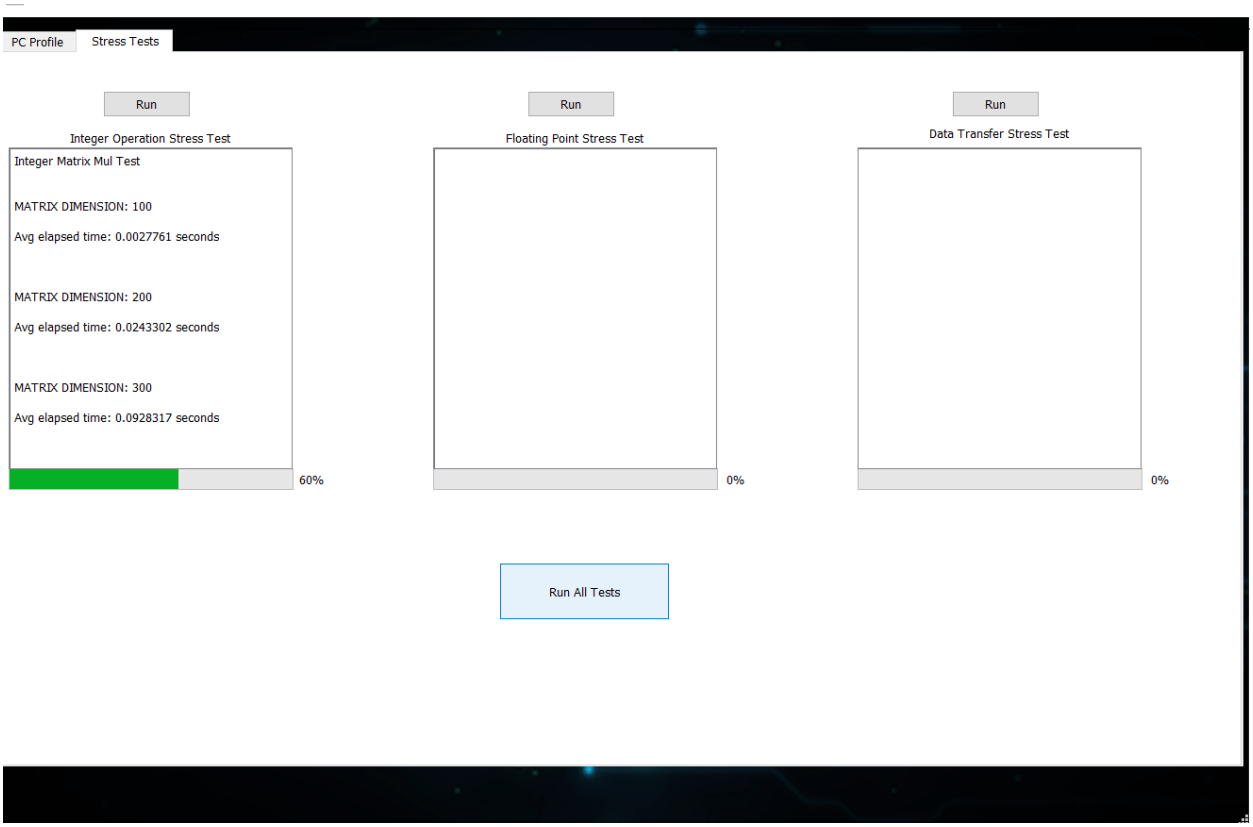
```
int** AllocateMatrix(int dim) {

	int** arr;

	arr = (int**)malloc(dim * sizeof(int*));

	for (int i = 0; i < dim; i++) {
		arr[i] = (int*)malloc(dim * sizeof(int));
		for (int j = 0; j < dim; j++) {
			arr[i][j] = d(gen);
		}
		// elemente
	}

	return arr;
}


void DeallocateMatrix(int **arr, int dim) {

	for (int i = 0; i < dim; i++) {
		free(arr[i]);
	}
	free(arr);
}


int **MatrixMultiply(int **A, int **B, int dim)
{
	int i;
	int j;
	int k;
	int sum;
	int **C;

	C = (int**)malloc(dim * sizeof(int*));
```

```
for (i = 0; i < dim; i++)
	{
		C[i] = (int*)malloc(dim * sizeof(int));
	}

	auto beg = std::chrono::high_resolution_clock::now();
	//clock_t start = clock();

	for (i = 0; i < dim; i++)
	{
		for (j = 0; j < dim; j++)
		{
			sum = 0;
			for (k = 0; k < dim; k++)
			{
				sum += A[i][k] * B[k][j] / DIVISION_FACTOR;
			}
			C[i][j] = sum;

		}
	}

	//clock_t end = clock();

	auto end = std::chrono::high_resolution_clock::now();

	auto dur = std::chrono::duration_cast<std::chrono::microseconds>(end - beg);

	//file <<  "Total time " << (1.0 * dur.count())  << " seconds" << endl;

	double elapsed_time = MICROS_TO_SECONDS * dur.count();

	file << "Dimension " << dim << " Test: " << test << " Total seconds : " <<
elapsed_time << endl;

	return(C);
}
```

Classes Implementation:

Since for the UI implementation, Qt C++ framework was used, the classes representing the Benchmark tests had to be designed around Qt Widgets, so they are able to independently print the results.

In the user interface, for each test, a test box is displayed in which information about the current state of the test and the results is generated, and a progress bar indicating the progress of the test is displayed. This look like in the image below:



Qt is an UI framework based on widgets, that are displayed using different layouts configuration. To be able to print those results while the test runs, I have declared the QProgressBar and QTextEdit in each of the test classes, so they can acces directly the UI components while they are running.

```
	class PiTest
{
	public:
		PiTest(QTextEdit *textEdit, QProgressBar *progressBar);
		PiTest(QTextEdit *textEdit, QProgressBar *progressBar, int min_nr_terms,
		int max_nr_terms, int nr_tests_per_run);

		~PiTest();

		int min_nr_terms;
		int max_nr_terms;
		int nr_tests_per_run;

		QTextEdit *textEdit;
		QProgressBar *progressBar;

		vector<float> results;

		void runTest();

};
```

While a test is in progress, at different steps in the test, the UI progress bar corresponding to each text is updated and in the text area, the average elapsed time for each operation is displayed.

```
	void PiTest::runTest() {

	this->textEdit->append("Pi Digits Calculation (Leibniz Series)\n \n");
	int progress = 50;
	this->progressBar->setValue(progress);
```


```
	for (int terms = this->min_nr_terms; terms <= this->max_nr_terms; terms += 1000000) {

		this->textEdit->append("TERMS: " + QString::number(terms) + " \n") ;

		float avg_elapsed_time = 0.0;
		int pos_elapsed_time_no = 0;

		for (int tests = 0; tests < nr_tests_per_run; tests++) {
			double pi = 0.0;

			auto beg = std::chrono::high_resolution_clock::now();

			for (int i = 0; i < terms; i++) {
				pi += (float)(pow(-1, i)) / (2 * i + 1);
			}

			auto end = std::chrono::high_resolution_clock::now();

			auto dur = std::chrono::duration_cast<std::chrono::microseconds>(end - beg);

			double elapsed_time = MICROS_TO_SECONDS * dur.count();

			cout.precision(1000);

			//cout << "Test: " << tests << " Pi: " << pi * 4 << endl << " Total seconds : " << elapsed_time << endl << endl;

			if (elapsed_time > 0.0) {
				avg_elapsed_time += elapsed_time;
				pos_elapsed_time_no++;
			}

		}

		avg_elapsed_time /= pos_elapsed_time_no;

		textEdit->append("Avg elapsed time: " + QString::number(avg_elapsed_time) + " seconds \n");

		this->results.push_back(avg_elapsed_time);

		progress += 10;
		this->progressBar->setValue(progress);

		textEdit->append(" \n");

	}

}
```

Each test runs 10 times, for each dimension / term number, after it computes an average of the positive values from those test runs and displays the result on the screen.

6. Testing and validation

7. Conclusion

Bibliography

[1] R. Longbottom, UK Government – Stress testing programs (technical report)

https://www.researchgate.net/publication/321868288_Stress_Testing_Programs

[2] Jóakim von Kistowski, University of Wuerzburg – How to Build a benchmark

https://www.researchgate.net/publication/273133047_How_to_Build_a_Benchmark

[3] Approximations for PI (wikipedia articles)

https://en.wikipedia.org/wiki/Approximations_of_%CF%80

[4] Infoware C++ library

https://github.com/ThePhD/infoware?fbclid=IwAR2noupqmHE-p8TuQF7DXxbKxnP0O5ZSfNnfDUSlZKCb-vNXzbY0kMsGRg4