Run time comparison of MATLAB, Scilab and GNU Octave on various benchmark programs

Roland Baudin, <roland65@free.fr>, july 2016

1. Introduction

This document presents a comparison of the run times of MATLAB, Scilab and GNU Octave (abbreviated as "Octave" in the sequel) on 50 benchmark programs found on the Internet or developed by the author.

2. Hardware and software used

2.1. Hardware

The computer used for the benchmarks is a HP Z820 PC workstation equipped with a Intel Xeon E5-2687W processor @ 3.10 GHz, with 16 cores, 32 threads and 128 GB RAM.

2.2. Software

The operating system used is Ubuntu Linux. More precisely, the system version is Ubuntu 16.04 Xenial Xerus and the architecture is amd64 (64 bits).

The MATLAB, Scilab and Octave versions used are given in the following table. It has to be noted that Octave was compiled with the experimental Just In Time (JIT) compiler enabled (see Appendix 7.1 for details). There is no such feature in Scilab, while in MATLAB the JIT feature is available by default.

Software	Version
MATLAB	R 2014a (8.3.0.532)
Scilab	5.5.2
Octave	4.0.2 (JIT enabled)

Table 2.1: Software versions

The three packages use the same linear algebra library, called LAPACK. This library, in turn, relies on the BLAS (Basic Linear Algebra Subprograms) library for its internal computations.

There are several implementations of the BLAS library that can be used within math packages like MATLAB, Scilab, Octave (and others, like Python SciPy). The four major implementations are called Reference BLAS (RefBLAS), Atlas BLAS (Atlas), OpenBLAS and Intel Math Kernel Library (MKL).

RefBLAS, Atlas and OpenBLAS are free software and are directly available through the package system of the Ubuntu Linux distribution. MKL is a commercial library developed by Intel. It is free for non commercial use and Intel provides a specific bundle that has to be manually installed in Ubuntu.

RefBLAS provides a reference single thread implementation of BLAS, but it is slow on some complex linear algebra operations. Atlas and OpenBLAS are highly optimized multithread BLAS implementations and are usually faster.

MKL is a multithread implementation of BLAS highly optimized for Intel processors and is usually very fast on such hardware.

Scilab and Octave are free software and can be compiled under Ubuntu Linux in such a way that it is easy to switch between the four implementations of the BLAS library (see Appendix 7.2 for the details of the procedure).

MATLAB, as a commercial software, is provided with a given version of the MKL, so that it is not possible to switch to another BLAS implementation.

The following table gives the version of the various BLAS libraries used in this comparison.

BLAS Library	Version	Package name in Ubuntu
RefBLAS	3.6.0	libblas3
Atlas	3.10.2	libatlas3-base
OpenBLAS	0.2.18	libopenblas-base
MKL	11.3.3.210	N/A
MKL MATLAB	11.0.5	N/A

Table 2.2: BLAS versions

3. Description of the benchmarks

We have used four different sets of benchmark programs. The first three were found on the Internet and only slightly modified for the purpose of this comparison. The fourth was specifically written by the author.

These four benchmark sets are named upon their directory name in the benchmark package. They are described below.

3.1. pincon

This benchmark set was written by Bruno Pinçon (hence the benchmark name) and is extensively described in [1].

It consists of 31 different programs that target various operations: loops, random generations, recursive calls, quick sorts, extractions and insertions, matrix operations, permutations, comparisons, cell reads and writes.

This benchmark set does not make an intensive use of the BLAS routines because its aim is mainly to test the speed of the interpreter and of the general matrix operations.

3.2. poisson

This benchmark set contains only two programs. It was developed by Neeraj Sharma and Matthias K. Gobbert and is fully described in [2].

The purpose of this benchmark is to provide a complex problem that could be encountered in the real life. The implemented problem results from a finite difference discretization of the Poisson equation (hence the name) in two spatial dimensions. The problem is solved using two algorithms: gaussian elimination (GE) and conjugate gradient (CG).

3.3. ncrunch

This benchmark set consists of 15 programs. It was designed by Stefan Steinhaus and is described in [3].

The purpose of this benchmark is to be a general “number crunching” benchmark (hence the name). It tests loops, random generation, sorting, FFT, determinants, matrix inverses, eigenvalues, Cholesky decomposition, principal components, special functions and linear regression.

This benchmark makes an intensive use of BLAS routines and thus it is very sensitive to the BLAS library implementation.

3.4. optim

This benchmark set contains two programs and was developed specifically by the author of this comparison. These programs implement two popular optimization metaheuristics, called Particle Swarm Optimization (PSO) and Differential Evolution (DE). These metaheuristics are described in [4] and [5]. Since there exists many variants of these methods, we choose to implement the “standard” versions, which are the most popular and simple ones.

The PSO method is used to find the global minimum of the Rosenbrock function in 30 dimensions. The DE method is used to find the global minimum of the Rastrigin function in 10 dimensions. Rosenbrock and Rastrigin functions are standard test functions used for the evaluation of global optimization methods.

The two methods are by nature loop intensives. However, the standard PSO method can easily be vectorized, while only some parts of the DE method can be vectorized. Thus, the DE method is slower than the PSO.

This benchmark set does not make much use of the BLAS routines and is rather insensitive to the choice of the BLAS library.

4. Results

4.1. Detailed results

The detailed results (i.e. run times in seconds, lower is better) of the four benchmark sets are given in tables 4.1 to 4.4. We comment below these results for each benchmark set.

- **pincon**

The first observation is that the MKL gives by far the worst total results for Scilab and Octave on this benchmark set.

Indeed, looking at the individual benchmark results, we see that some programs have nearly the same performances in all BLAS implementations (`fibonacci`, random generation, quick sort, `fannkuch`, etc.) while others (subsets, `crible`, loop calls, etc.) have much higher run times in MKL.

We can also see such a phenomenon in MATLAB (though there is no comparison point with another BLAS implementation in MATLAB): for example, it is quite strange that the `hmeans` (a simple class partitioning algorithm) benchmark has a run time of ~3 s in MATLAB, while it is only ~0.4 s in Scilab and ~0.1 s in Octave. The same observation can be done on the `make_perm` (permutation algorithm), loop calls, `crible` or `simplexe` algorithms.

A possible explanation could be that the MKL is well optimized for complex linear algebra operations (as will be seen later) but this optimization has some negative impact on more simple matrix operations. However, this is only a conjecture since we don't know the internals of the MKL library.

It has to be noted that Scilab gives bad results on the cell test (construction and lecture) but this should be fixed in the next generation of the program (Scilab 6).

If we disregard the Scilab and Octave results with MKL, we see that the three packages give comparable results on the `pincon` benchmark set, whatever the BLAS library. Anyway, MATLAB is slightly faster, then comes Scilab and the last is Octave.

Benchmark name	MATLAB (MKL)	Scilab (RefBLAS)	Scilab (Atlas)	Scilab (OpenBLAS)	Scilab (MKL)	Octave (RefBLAS)	Octave (Atlas)	Octave (OpenBLAS)	Octave (MKL)
fib(24)	1.11	0.52	0.54	0.59	0.67	2.04	2.06	2.03	1.99
subsets(1:20.7)	0.76	0.51	0.57	0.57	3.31	2.24	2.37	2.25	2.22
irn55_rand(1.1e6)	0.78	0.44	0.44	0.46	0.48	2.28	2.24	2.26	2.25
merge_sort. 10000 elts	0.29	0.42	0.45	0.46	0.47	2.48	2.48	2.44	2.41
qSort. 40000 elts	1.28	0.62	0.66	0.68	0.70	2.83	2.85	2.83	2.73
25 times harmvect(1e6)	0.46	0.22	0.23	0.21	0.22	0.22	0.23	0.21	0.22
harmloop(1e6)	0.02	0.47	0.51	0.51	0.50	0.01	0.01	0.02	0.01
3 times fannkuch(7)	0.46	0.29	0.30	0.31	0.30	2.37	2.39	2.34	2.30
my_lu 400x400	2.04	0.15	0.15	0.14	1.90	0.10	0.10	0.10	1.21
crible(3e6)	1.19	0.12	0.11	0.11	1.46	0.09	0.09	0.09	1.19
p = make_perm(20000)	3.80	0.44	0.44	0.49	2.28	2.21	2.24	2.21	4.50
4 times q = inv_perm(p)	0.01	0.26	0.30	0.28	0.29	0.05	0.06	0.05	0.05
fftx 2^15 elts	1.79	2.76	2.86	2.89	2.92	4.10	4.13	4.08	4.02
30 times pascal(1029)	0.20	0.34	0.36	0.36	0.36	0.50	0.50	0.51	0.50
hmeans. 8x8000. K=4	2.92	0.36	0.37	0.70	7.91	0.13	0.14	0.12	2.17
simplex 200 cstr 300 var	1.94	0.16	0.16	0.15	2.74	0.12	0.10	1.71	2.24
loop_call_f (30000 elts)	1.37	0.60	0.65	0.74	6.85	2.96	2.95	4.51	6.97
loop_call_p (50000 elts)	3.39	0.22	0.21	0.31	0.24	2.08	2.12	2.10	2.07
form_vect1(20000)	0.01	0.30	0.31	0.31	0.31	0.01	0.01	0.01	0.01
form_vect2(20000)	0.03	0.14	0.14	0.13	0.14	0.17	0.17	0.17	0.16
form_vect3(20000)	0.00	0.03	0.01	0.01	0.01	0.00	0.00	0.00	0.00
loop1(1e6)	0.01	1.17	0.84	0.36	0.36	0.00	0.01	0.01	0.00
loop2(1e6)	0.01	0.53	0.58	0.64	0.58	3.84	3.87	3.80	3.76
loop3(3e5)	0.01	0.39	0.44	0.44	0.46	0.01	0.00	0.00	0.00
test_bool_and_comp_ops	0.08	0.32	0.35	0.25	0.26	0.06	0.06	0.06	0.06
test_find	0.77	0.40	0.38	0.39	0.47	0.20	0.21	0.21	0.20
prime_factors(160001)	0.01	5.11	5.20	5.19	5.28	1.40	1.40	1.41	1.39
extraction test	0.05	0.11	0.11	0.12	0.12	0.04	0.04	0.04	0.04
insertion test	0.07	0.17	0.17	0.17	0.22	0.07	0.07	0.07	0.06
bench_cells(1000) construction	0.01	5.23	5.18	5.55	5.34	0.02	0.02	0.02	0.02
bench_cells(1000) lecture	0.00	4.06	4.12	4.18	4.26	0.00	0.00	0.00	0.00
Total	**24.87**	**26.87**	**27.17**	**27.68**	**51.42**	**32.62**	**32.94**	**35.66**	**44.78**

Table 4.1: Run times of the pincon benchmark set

- **poisson**

This benchmark set emphasizes a big difference between the MATLAB / Octave run time and the Scilab run time on the gaussian elimination implementation of the Poisson problem. This could be a problem of Scilab itself or perhaps a problem of the algorithm implementation.

We can also see there is no major differences between the different BLAS implementations.

The winner here is Octave, then comes MATLAB and the last is Scilab.

Benchmark name	MATLAB (MKL)	Scilab (RefBLAS)	Scilab (Atlas)	Scilab (OpenBLAS)	Scilab (MKL)	Octave (RefBLAS)	Octave (Atlas)	Octave (OpenBLAS)	Octave (MKL)
poisson(256) gauss	0.30	172.04	190.44	181.69	180.33	0.24	0.15	0.24	0.51
poisson(256) conj. gradient	1.29	1.45	1.48	1.42	1.56	0.66	0.68	0.67	0.84
Total	**1.59**	**173.49**	**191.92**	**183.11**	**181.89**	**0.90**	**0.83**	**0.91**	**1.35**

Table 4.2: Run times of the poisson benchmark set

- **ncrunch**

The total run time results show a strong influence of the BLAS library on Scilab and Octave performances. The run times can generally be sorted as RefBLAS > Atlas > OpenBLAS > MKL. However, since Scilab expected crashes with OpenBLAS on three benchmarks, this BLAS implementation must be discarded for this package.

The influence of the BLAS implementation is normal since a number of benchmarks belonging to this set are related to linear algebra operations (matrix inverse, eigenvalues, cross-product, principal components, linear regression).

MATLAB gives by far the shortest run times. Then comes Octave and then Scilab.

Benchmark name	MATLAB (MKL)	Scilab (RefBLAS)	Scilab (Atlas)	Scilab (OpenBLAS)	Scilab (MKL)	Octave (RefBLAS)	Octave (Atlas)	Octave (OpenBLAS)	Octave (MKL)
IO test & descriptive statistics	0.95	1.14	1.16	1.17	1.34	1.20	1.18	1.19	1.17
Loop 1000x1000	0.20	1.40	1.49	1.53	1.49	0.01	0.01	0.01	0.01
2000x2000 matrix ^1000	0.03	0.22	0.23	0.21	0.22	0.04	0.04	0.04	0.04
10000000 values sorted ascending	0.29	2.23	2.21	2.24	2.23	1.21	1.22	1.24	1.20
FFT over 1048576 values (10 times)	0.20	0.54	0.36	0.34	0.37	0.14	0.08	0.14	0.18
Determinant of a 1500x1500 matrix	0.24	0.69	0.20	**crash**	0.03	0.70	0.21	0.26	0.05
Inverse of a 1500x1500 matrix	0.10	2.03	0.58	**crash**	0.12	2.00	0.61	0.34	0.12
Eigenvalues of a 1200x1200 matrix	0.93	4.11	2.56	2.42	1.75	4.06	2.67	1.96	1.02
Cholesky decomposition of a 1500x1500 matrix	0.01	0.47	0.10	0.02	0.01	0.47	0.10	0.01	0.01
1500x1500 cross-product of a matrix	0.04	2.37	0.53	0.06	0.06	1.38	0.27	0.02	0.03
Principal components of a 5000x500 matrix	0.19	23.83	7.78	5.29	0.83	22.48	7.31	4.47	0.62
Calculation of 10000000 fibonacci numbers	0.22	0.99	0.98	1.01	1.07	1.54	1.55	1.57	1.62
Gamma function over a 1500x1500 matrix (10times)	0.10	0.53	0.53	0.54	0.53	3.31	3.34	3.44	3.30
Erf function over a 1500x1500 matrix (10 times)	0.05	0.30	0.30	0.31	0.30	0.17	0.18	0.18	0.17
Linear regression over a 2000x2000 matrix	0.10	1.65	0.50	**crash**	0.09	1.66	0.51	0.11	0.11
Total	**3.64**	**42.49**	**19.49**	**N/A**	**10.45**	**40.37**	**19.28**	**14.98**	**9.62**

Table 4.3: Run times of the ncrunch benchmark set

– **optim**

There is no influence of the BLAS library on the run times of this set. This is normal since these programs do not make use of linear algebra functions.

MATLAB gives the shortest run time, then comes Scilab and then Octave. It has to be noted that, for an unexplained reason, the Octave run times of the DE method are nearly three times higher than the Scilab ones.

Benchmark name	MATLAB (MKL)	Scilab (RefBLAS)	Scilab (Atlas)	Scilab (OpenBLAS)	Scilab (MKL)	Octave (RefBLAS)	Octave (Atlas)	Octave (OpenBLAS)	Octave (MKL)
Standard PSO	1.29	3,59	3.68	3.76	3.63	5.12	5.05	5.11	5.07
Standard DE	21.03	23,08	23.97	23.18	23.23	67.86	67.23	66.89	66.65
Total	**22.33**	**26.67**	**27.65**	**26.94**	**26.86**	**72.98**	**72.29**	**72.00**	**71.72**

Table 4.4: Run times of the optim benchmark set

4.2. Selection of the BLAS implementation

In order to obtain a clear comparison of three package performances, we need to summarize the above detailed results. For this purpose, we select below the best BLAS implementation for Scilab and Octave (for MATLAB, MKL is the only choice).

For Scilab, the `pincon` benchmark set gives some very slow run times with MKL on basic matrix manipulations. Therefore, this eliminates MKL even if the `ncrunch` MKL run times are the lowest.

OpenBLAS cannot be selected because of crashes in three benchmarks of the `ncrunch` set. Atlas gives much better results than RefBLAS on the `ncrunch` set, with only minor negative impact on the other benchmark sets. So, we select the Atlas BLAS implementation for Scilab.

For Octave, MKL gives bad results with the `pincon` benchmark set (same problem as with Scilab), so we discard MKL, even if the `ncrunch` run times are a little lower.

OpenBLAS gives much shorter run times on the `ncrunch` set than RefBLAS and Atlas and has nearly no effect on the other benchmark sets. So, we select OpenBLAS for Octave.

Table 4.5 summarizes these choices.

Package	BLAS library
MATLAB	MKL
Scilab	Atlas
Octave	OpenBLAS

Table 4.5: Selected BLAS implementations

4.3. Absolute run time comparison

A first way to summarize the above results is to compare the global run times of each benchmark set and also the total run times on all the sets. Table 4.6 shows these results.

Looking at these results, we see that MATLAB has the lowest run times for the `pincon`, `ncrunch` and `optim` benchmark sets. Octave has the lowest run time for the `poisson` set but the highest for the `pincon` and `optim` sets. Scilab is not far from MATLAB on the `pincon` and `optim` sets but has a very

high run time on the `poisson` set.

As we see, the performance of Scilab is penalized by its very bad run time on only one benchmark (Poisson problem with gaussian elimination). If we disregard this benchmark, we see that Scilab is globally 1.5 times slower than MATLAB while Octave is globally 2.5 times slower than MATLAB.

Benchmark set	MATLAB (MKL)	Scilab (Atlas)	Octave (OpenBLAS)
pincon	24.87	27.17	35.66
poisson	1.59	191.92	0.91
ncrunch	3.64	19.49	14.98
optim	22.33	27.65	72.00
Total	**52.43**	**266.23**	**123.55**

Table 4.6: Absolute run time comparison

4.4. Relative run time comparison

Taking into account the absolute run times can penalize a given package because its run time on a specific benchmark is catastrophic. Of course, such an isolated bad result does not mean that the package is globally bad.

We propose here a relative comparison of the run times on each benchmark. For that purpose, we compare the three packages:

- by counting the number of benchmarks where a package has the lowest run time,
- by counting the number of benchmarks where a package has the highest run time.

We can then sort the three packages according to these two merit figures and this also will allow checking the consistency of the results.

The results (extracted from tables 4.1 to 4.4) are shown in table 4.7 for all benchmarks. On this table, the lowest run times are colored green and the highest run times are colored red.

The number of benchmarks with lowest and highest run times are counted for each package. The results are displayed in table 4.8 and in figures 4.1 and 4.2.

The results are consistent (same order in the two merit figures) and we can see that MATLAB is the clear winner, then comes Octave and the last is Scilab.

Comparing these results with the absolute run time comparison, this confirms MATLAB as the fastest package. However, the conclusion about Scilab and Octave is different. In this relative comparison, Octave appears clearly better than Scilab, meaning that its has more often a lower run time than Scilab on individual benchmarks.

Benchmark name	MATLAB (MKL)	Scilab (Atlas)	Octave (OpenBLAS)
pincon			
fib(24)	1.11	0.54	2.03
subsets(1:20.7)	0.76	0.57	2.25
irn55_rand(1.1e6)	0.78	0.44	2.26
merge_sort. 10000 elts	0.29	0.45	2.44
qSort. 40000 elts	1.28	0.66	2.83
25 times harmvect(1e6)	0.46	0.23	0.21
harmloop(1e6)	0.02	0.51	0.016
3 times fannkuch(7)	0.46	0.30	2.34
my_lu 400x400	2.04	0.15	0.10
crible(3e6)	1.19	0.11	0.09
p = make_perm(20000)	3.80	0.44	2.21
4 times q = inv_perm(p)	0.01	0.30	0.05
fftx 2^15 elts	1.79	2.86	4.08
30 times pascal(1029)	0.20	0.36	0.51
hmeans. 8x8000. K=4	2.92	0.37	0.12
simplex 200 cstr 300 var	1.94	0.16	1.71
loop_call_f (30000 elts)	1.37	0.65	4.51
loop_call_p (50000 elts)	3.39	0.21	2.10
form_vect1(20000)	0.01	0.31	0.008
form_vect2(20000)	0.03	0.14	0.17
form_vect3(20000)	0.00	0.01	0.004
loop1(1e6)	0.01	0.84	0.008
loop2(1e6)	0.01	0.58	3.80
loop3(3e5)	0.01	0.44	0.004
test_bool_and_comp_ops	0.08	0.35	0.06
test_find	0.77	0.38	0.21
prime_factors(160001)	0.01	5.20	1.41
extraction test	0.05	0.11	0.04
insertion test	0.07	0.17	0.068
bench_cells(1000) construction	0.01	5.18	0.02
bench_cells(1000) lecture	0.00	4.12	0.004
poisson			
poisson(256) gauss	0.30	190.44	0.24
poisson(256) conj. gradient	1.29	1.48	0.67
ncrunch			
IO test & descriptive statistics	0.95	1.16	1.19
Loop 1000x1000	0.20	1.49	0.01

2000x2000 matrix ^1000	0.03	0.23	0.04
10000000 values sorted ascending	0.29	2.21	1.24
FFT over 1048576 values (10 times)	0.20	0.36	0.14
Determinant of a 1500x1500 matrix	0.24	0.20	0.26
Inverse of a 1500x1500 matrix	0.10	0.58	0.34
Eigenvalues of a 1200x1200 matrix	0.93	2.56	1.96
Cholesky decomposition of a 1500x1500 matrix	0.01	0.10	0.012
1500x1500 cross-product of a matrix	0.04	0.53	0.02
Principal components of a 5000x500 matrix	0.19	7.78	4.47
Calculation of 10000000 fibonacci numbers	0.22	0.98	1.57
Gamma function over a 1500x1500 matrix (10times)	0.10	0.53	3.44
Erf function over a 1500x1500 matrix (10 times)	0.05	0.30	0.18
Linear regression over a 2000x2000 matrix	0.10	0.50	0.11
optim			
Standard PSO	1.29	3.68	5.11
Standard DE	21.03	23.97	66.89

Table 4.7: Comparison of MATLAB, Scilab and Octave run times (green color for lowest run time, red color for highest)

Package	Number of benchmarks having the lowest run times	Number of benchmarks having the highest run times
MATLAB	23 / 50	8 / 50
Octave	17 / 50	17 / 50
Scilab	10 / 50	25 /50

Table 4.8: Relative run time comparison

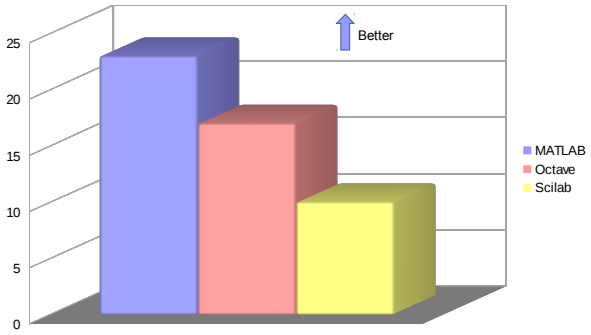

Figure 4.1: Number of benchmarks having the lowest run times

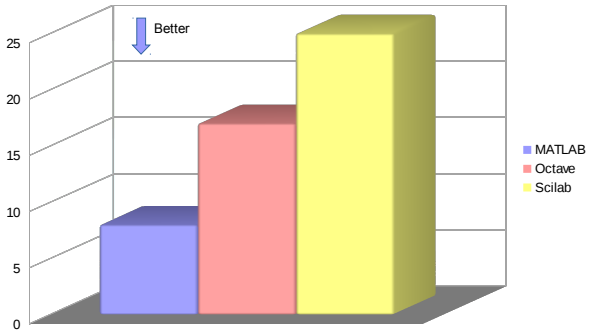

Figure 4.2: Number of benchmarks having the highest run times

4.5. Influence of the JIT compiler

It is well known that loop intensive programs have high run times in MATLAB, Scilab, Octave and other number crunching packages. Expressing the loops in a vectorized form can help to get more decent run times.

For years, MATLAB has introduced its Just In Time (JIT) compiler that tries to automatically vectorize the loops when executing the code. Octave has also such a feature, but is is experimental and only works on simple loops. Scilab does not have such a feature yet.

The JIT feature can easily be enabled or disabled (see Appendix 7.4) in MATLAB and Octave, so it is tempting to evaluate the impact of the JIT compiler on MATLAB and Octave run times using our four benchmark sets.

The results are shown in table 4.9. In this table, the run times where the JIT compiler makes a significant difference (good or bad) are colored in red.

For Octave, it just as expected: the JIT compiler reduces the run time only when simple loops (i.e. loops that contain no function call) exist in the code . For example, we see there is no improvement of the PSO and DE run times, though these benchmarks are rather loop intensives.

For MATLAB, the JIT compiler has a more deep impact: it can significantly accelerate many benchmark programs that contain complex loops. This can be checked, for example, on the PSO and DE run times.

However, it has to be mentioned the MATLAB JIT compiler can sometimes increase the run time: this is indeed the case for three benchmarks of the `pincon` set.

Benchmark name	MATLAB (MKL. JIT enabled)	MATLAB (MKL. JIT disabled)	Octave (OpenBLAS. JIT enabled)	Octave (OpenBLAS. JIT disabled)
pincon				
fib(24)	1.11	1.03	2.03	2.04
subsets(1:20.7)	0.76	1.22	2.25	2.22
irn55_rand(1.1e6)	0.78	1.62	2.26	2.27
merge_sort. 10000 elts	0.29	0.66	2.44	2.48
qSort. 40000 elts	1.28	1.25	2.83	2.80
25 times harmvect(1e6)	0.46	0.34	0.21	0.21
harmloop(1e6)	0.02	0.69	0.02	1.64
3 times fannkuch(7)	0.46	0.93	2.34	2.42
my_lu 400x400	2.04	3.51	0.10	0.10
crible(3e6)	1.19	1.18	0.09	0.08
p = make_perm(20000)	3.80	2.72	2.21	2.19
4 times q = inv_perm(p)	0.01	0.37	0.05	1.91
fftx 2^15 elts	1.79	2.14	4.08	4.03
30 times pascal(1029)	0.20	0.46	0.51	0.50
hmeans. 8x8000. K=4	2.92	2.56	0.12	0.12
simplex 200 cstr 300 var	1.94	2.69	1.71	2.00
loop_call_f (30000 elts)	1.37	6.18	4.51	4.36
loop_call_p (50000 elts)	3.39	0.20	2.10	2.12
form_vect1(20000)	0.01	0.01	0.01	0.07
form_vect2(20000)	0.03	0.08	0.17	0.17
form_vect3(20000)	0.00	0.01	0.00	0.07
loop1(1e6)	0.01	0.60	0.01	1.38
loop2(1e6)	0.01	0.66	3.80	3.84
loop3(3e5)	0.01	0.65	0.00	1.62
test_bool_and_comp_ops	0.08	0.07	0.06	0.06
test_find	0.77	0.32	0.21	0.20
prime_factors(160001)	0.01	0.67	1.41	1.35
extraction test	0.05	0.04	0.04	0.04
insertion test	0.07	0.06	0.07	0.06
bench_cells(1000) construction	0.01	0.01	0.02	0.02
bench_cells(1000) lecture	0.00	0.00	0.00	0.00

poisson				
poisson(256) gauss	0.30	0.44	0.24	0.22
poisson(256) conj. gradient	1.29	1.17	0.67	0.67
ncrunch				
IO test & descriptive statistics	0.95	1.77	1.19	1.18
Loop 1000x1000	0.20	0.71	0.01	2.07
2000x2000 matrix ^1000	0.03	0.04	0.04	0.04
10000000 values sorted ascending	0.29	0.32	1.24	1.21
FFT over 1048576 values (10 times)	0.20	0.39	0.14	0.14
Determinant of a 1500x1500 matrix	0.24	0.23	0.26	0.28
Inverse of a 1500x1500 matrix	0.10	0.22	0.34	0.33
Eigenvalues of a 1200x1200 matrix	0.93	1.00	1.96	1.94
Cholesky decomposition of a 1500x1500 matrix	0.01	0.01	0.01	0.01
1500x1500 cross-product of a matrix	0.04	0.02	0.02	0.02
Principal components of a 5000x500 matrix	0.19	0.34	4.47	4.97
Calculation of 10000000 fibonacci numbers	0.22	0.22	1.57	1.55
Gamma function over a 1500x1500 matrix (10times)	0.10	0.07	3.44	3.31
Erf function over a 1500x1500 matrix (10 times)	0.05	0.04	0.18	0.17
Linear regression over a 2000x2000 matrix	0.10	0.08	0.11	0.13
optim				
Standard PSO	1.29	1.93	5.11	5.13
Standard DE	21.03	35.59	66.89	67.91

Table 4.9: JIT compiler influence

5. Conclusion

Considering the above results, it is clear that MATLAB is the fastest of the three packages we evaluated.

When we compared the absolute run times of the benchmark sets, we found that MATLAB is 1.5 faster than Scilab and 2.5 times faster than Octave. It is also the fastest of the three packages in 50 % of the benchmarks. Moreover, thanks to its JIT compiler, it can be several times faster than Scilab or Octave when the code contains complex loops with function calls.

Scilab can be as fast as MATLAB on simple matrix computations (the `pincon` benchmark set). Surprisingly, even without a JIT compiler, it can also give fast run times in the loop intensive programs of the `optim` set. However, Scilab curiously has high run times on some benchmarks where Octave and MATLAB performs very well.

Octave is slower than Scilab on the absolute run time comparison. However, it gives the fastest run time more often than Scilab when comparing individual benchmarks. Besides, Octave is very robust and has no significant failure on any benchmark. Its JIT compiler can be useful in simple loops but is not able to vectorize loops that contains function calls.

Drawing a definitive conclusion between Scilab and Octave is actually difficult: Scilab seems faster on complex loops, while Octave is slower in those cases. However, the two packages perform equally well on linear algebra intensive problems. Nevertheless, Octave is more robust than Scilab and expected no crash and no run time outlier on all the benchmarks.

6. References

[1] Pinçon B., "Quelques tests de rapidité entre différents logiciels matriciels", University of Lorraine, http://cermics.enpc.fr/~jpc/scilab-gtk-tiddly/bench.pdf

[2] Sharma N., Gobbert M. K., "A comparative evaluation of MATLAB, Octave, FreeMat and Scilab for Research and Teaching", Department of Mathematics and Statistics, University of Maryland, Baltimore, 2010, http://userpages.umbc.edu/~gobbert/papers/SharmaGobbertTR2010.pdf

[3] Steinhaus, S., "Comparison of Mathematical Programs for Data Analysis", Munich, 2008, http://www.scientificweb.de/ncrunch/

[4] Kennedy J., Eberhart R., « Particle swarm optimization », Neural Networks, 1995, vol. 4, p. 1942- 1948, 1995.

[5] Storn R., Price K., "Differential Evolution - A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces", Journal of Global Optimization, vol. 11, n^{o} 4, p. 341-359, 1997.

7. Appendices

7.1. JIT support in Octave

At the time of writing (july 2016), the Octave version provided in Ubuntu 16.04 is 4.0.0. This version is not up to date and was not compiled with JIT support.

For convenience, the author provides the binary deb packages of Octave 4.0.2 compiled with JIT support. These packages can be installed directly in Ubuntu 16.04 (amd64 architecture).

To use these packages, download the archive file http://roland65.free.fr/benchmarks/octave-4.0.2-debs.tar.gz , then enter in a terminal:

```
sudo apt-get install octave
```

This command installs the Octave version that comes with Ubuntu and ensures that all the required dependencies are installed.

Then install the provided deb packages by typing:

```
tar zxvf octave-4.0.2-debs.tar.gz
cd octave-4.0.2-debs
sudo dpkg -i *.deb
```

After that, launch the Octave GUI using:

```
/usr/bin/octave
```

If for some reason one needs to compile Octave with JIT support, the instructions are given below.

First, to compile octave with JIT support, it is necessary to install some deb packages coming from the older Ubuntu 14.04 distribution. These packages are (for a 64 bits architecture):

```
llvm-3.3_3.3-16ubuntu1_amd64.deb
llvm-3.3-runtime_3.3-16ubuntu1_amd64.deb
llvm-3.3-dev_3.3-16ubuntu1_amd64.deb
libllvm3.3_3.3-16ubuntu1_amd64.deb
libisl10_0.12.2-1_amd64.deb
libcloog-isl4_0.18.2-1_amd64.deb
```

One can download these packages using the http://packages.ubuntu.com query interface. Once they are present in a directory, they can be installed by typing in a terminal:

```
sudo dpkg -i *.deb
```

It is also necessary to install and select the older compiler gcc 4, because compilation with gcc 5 will fail when JIT support is enabled. Thus, enter the following commands:

```
sudo apt-get install gcc-4.9 g++-4.9
sudo update-alternatives gcc
sudo update-alternatives g++
```

In each of the above `update-alternatives` commands, select the gcc-4.9 and g++-4.9 versions.

The Octave source package can be obtained from ftp://ftp.gnu.org/gnu/octave . To compile Octave, first install the building dependencies by entering in terminal:

```
sudo apt-get build-dep octave
```

Then type:

```
tar Jxvf octave-4.0.2.tar.xz
cd octave-4.0.2
./configure LLVM_CONFIG=/usr/bin/llvm-config-3.3 JAVA_HOME=/usr/lib/jvm/default-java \
CPPFLAGS=-I/usr/include/hdf5/serial LDFLAGS=-L/usr/lib/x86_64-linux-gnu/hdf5/serial \
--enable-jit --prefix=/usr/local/octave
make -j <number of cores>
sudo make install
```

If the compilation succeeded, the octave binaries are located in `/usr/local/octave`. Then, the Octave GUI can be launched by:

```
/usr/local/octave/bin/octave
```

7.2. BLAS library selection

It is assumed that Octave is installed using the packages provided by the author (see previous paragraph). Then in Ubuntu 16.04, Scilab 5.5.2 can be installed by entering in a terminal:

```
sudo apt-get install scilab
```

The RefBLAS library (package `libblas3`) is automatically installed as a dependency of Scilab or Octave. However, one also needs to install the Atlas and OpenBLAS libraries:

```
sudo apt-get install libatlas3-base libopenblas-base
```

The active BLAS library can be selected using the `update-alternatives` mechanism of Ubuntu. For that, enter in a terminal:

```
sudo update-alternatives --config libblas.so.3
```

and enter the number that corresponds to the BLAS version that is desired.

Then repeat the same procedure for the LAPACK library:

```
sudo update-alternatives --config liblapack.so.3
```

and enter the number that corresponds to the BLAS version that is desired.

After that, the Scilab and Octave programs automatically use the selected BLAS version.

The MKL library is not distributed within Ubuntu. It is free for non commercial use but must be obtained and installed from the Intel web site. See the download and install instructions at https://software.intel.com/en-us/intel-mkl .

It is assumed that the MKL library is installed in the following directory (this is the default for MKL version 11.3.3.210):

```
/opt/intel/compilers_and_libraries_2016.3.210
```

To select the MKL library in Scilab, edit a file called scilab-mkl.sh with the following contents:

```
#!/bin/sh

# Intel MKL libraries
MKL_LIBS=/opt/intel/compilers_and_libraries_2016.3.210/linux/mkl/lib/intel64_lin

# Intel MKL compiler libraries
MKL_COMP_LIBS=/opt/intel/compilers_and_libraries_2016.3.210/linux/compiler/lib/intel64_
lin

# Launch Scilab 5.x with MKL libraries
export LD_LIBRARY_PATH=$MKL_LIBS:$MKL_COMP_LIBS
LD_PRELOAD="$MKL_LIBS/libmkl_gf_lp64.so $MKL_LIBS/libmkl_gnu_thread.so
$MKL_LIBS/libmkl_core.so $MKL_COMP_LIBS/libiomp5.so" /usr/bin/scilab
```

Then make the file executable:

```
chmod +x
```

and launch it :

```
./scilab-mkl.sh
```

For Octave, copy the above shell script to `octave-mkl.sh` and replace `/usr/bin/scilab` with `/usr/bin/octave` in the last line of the file.

7.3. Running benchmarks

The benchmark archive can be downloaded at the following address: http://roland65.free.fr/benchmarks/benchmarks-0.1.tar.gz .

The archive contains all the source codes (`.m` files for MATLAB / Octave and `.sce`, `.sci` files for Scilab) plus some documentation.

The commands used to launch the benchmarks are given below.

- **in MATLAB / Octave**

For the `pincon` set:

```
>> cd pincon/m/
>> bench
>> bench_cells
```

For the `poisson` set:

```
>> cd poisson/m/
>> driver_ge(256)
>> driver_cg(256)
```

For the `ncrunch` set:

```
>> cd ncrunch/m/
>> ncrunch
```

For the `optim` set:

```
>> cd optim/m/
>> standard_PSO
>> standard_DE
```

- **in Scilab**

For the `pincon` set:

```
--> cd pincon/sci/
--> exec('bench.sce', -1)
--> exec('bench_cells.sce', -1)
```

For the `poisson` set:

```
--> cd poisson/sci/
--> driver_ge(256)
--> driver_cg(256)
```

For the `ncrunch` set:

```
--> cd ncrunch/sci/
--> exec('ncrunch.sce', -1)
```

For the `optim` set:

```
--> cd optim/sci/
--> exec('standard_PSO.sce', -1)
--> exec('standard_DE.sce', -1)
```

7.4. Enabling / disabling the JIT compiler in MATLAB and Octave

In MATLAB, the JIT compiler can be enabled and then disabled by entering the following commands:

```
>> feature accel on
>> feature accel off
```

In Octave, the JIT compiler can be enabled and then disabled by entering the following commands:

```
>> enable_jit(1)
>> enable_jit(0)
```