Basic Comparison of Python, Julia, Matlab, IDL and Java (2018 Edition)

Announcement: We have started the process of making this project open source. The source codes are being rewritten for clarity, simplicity and consistency. As soon as the process is completed, all the new codes and running scripts will be made available.

HISTORY:

- September 13, 2018: Added R numbers for the Fibonacci Number test case (Problem 1)
- September 13, 2018: Corrected R numbers for the Laplace Equation test case (Problem 5)

This report is the continuation of the work done in:

Basic Comparison of Python, Julia, R, Matlab and IDL

Here we:

1. Add new versions of languages
2. Add JAVA
3. Add more test cases.
4. For each language, consistantly use the same method to measure the elapsed time.
5. Provide source codes for all the test cases.
6. Present all the timing results to the fourth digit accuracy (any number less tha 0.0001 is rounded to 0).

While reading this report, be mindful of the following:

- Our intention is not to claim that one language is better than the other.
- In our work, we are often asked to address users' issues on the computing languages Python, Matlab, IDL, etc. We only have few hours to understand the coding principles of those languages and quickly write codes that resolve users' issues. We present results in the point of view of a novice programmer.
- If you are an advanced programmer or a language developer and you have results (obtained with optimization techniques) you want to share, feel free to contact us (with a web link) and we wil provide a link to your results here.

All the experiments presented here were done on Intel Xeon Haswell processor node. Each node has 28 cores (2.6 GHz each) and 128 Gb of available memory. We consider the following versions of the languages:

Language	**Version**	**Free?**
Python	2.7.1	Yes
Julia	0.6.2	Yes
JAVA	1.8.0_92	Yes
IDL	8.5	No
Matlab	R2017a	No
R		Yes
GNU Compilers	7.3	Yes
Intel Compilers	18.0.1.163	No
Scala	2.12.4	Yes

Problem 1: Fibonacci Number

The Fibonacci numbers are the sequence of numbers defined by the linear recurrence equation:

with .

Fibonacci numbers find applications in the fields of economics, computer science, biology, combinatoric, etc.

We implemented both the iterative method and the recursive one, and we record the elepased time for generating the Fibonacci numbers for a given n.

Language	**Option**	**n=25**	**n=35**	**n=45**
Python		0	0	0
Python + Numba		0	0	0
Julia		0	0	0
IDL		0	0	0
Matlab		0.0098	0.0032	0.0025
R		0.034	0.034	0.034
JAVA		0	0	0
Scala		0	0	0
Fortran	gfortran	0	0	0
	gfortran -O3	0	0	0
	ifort	0	0	0
	ifort -O3	0	0	0
C	gcc	0	0	0
	gcc -Ofast	0	0	0
	icc	0	0	0
	icc -Ofast	0	0	0

Table 1.1: Elapsed times (in seconds) obtained by computing Fibonacci numbers using then iterative method.

Language	Option	n=25	n=35	n=45
Python		0.0211	2.5284	311.2046
Python + Numba		0.03	0.1	8.82
Julia		0	0.0335	4.130
IDL		0.0301	2.2573	304.2285
Matlab		0.0128	0.5149	58.9283
R		0.008	0.008	0.008
JAVA		0.0016	0.0414	4.8609
Scala		0.001	0.045	5.289
Fortran	gfortran	0	0.0840	10.4326
	gfortran -O3	0	0.0280	3.4602
	ifort	0	0	0
	ifort -O3	0	0	0
C	gcc	0	0.04	5.07
	gcc -Ofast	0	0.01	1.66
	icc	0	0.02	3.15
	icc -Ofast	0	0.02	3.07

Table 1.2: Elapsed times (in seconds) obtained by computing Fibonacci numbers using the recursive method.

Problem 2: Copy Arrays

This test case is meant to show how fast languages access non-contiguous memory locations.

Consider an arbitrary nxnx3 matrix A. We want to perform the following operations on A:

```
A(i,j,1) = A(i,j,2)
A(i,j,3) = A(i,j,1)
A(i,j,2) = A(i,j,3)
```

For instance, in Python the code looks like:

```
for i in range(n):
    for j in range(n):
        A[i,j,1] = A[i,j,2]
        A[i,j,3] = A[i,j,1]
        A[i,j,2] = A[i,j,3]
```

The above code segment uses loops. We are also interested on how the same operations are done using vectorization:

```
A[:,:,1] = A[:,:,2]
A[:,:,3] = A[:,:,1]
A[:,:,2] = A[:,:,3]
```

The problem allows us to see how each language handles loops and vectorization. We record the elapsed time needed to do the array assignments.

Language	Option	n=5000	n=7000	n=9000
Python		18.6055	37.1279	61.0172
Python + Numba		0.26	0.26	0.34
Julia		0.0907	0.1386	0.2274
IDL		6.8773	13.2422	21.9349
Matlab		0.2787	0.5223	0.8437
R		19.750	38.635	63.820
JAVA		0.1420	0.2680	0.4350
Scala		0.204	0.349	0.51
Fortran	gfortran	0.1760	0.3480	0.5760
	gfortran -O3	0.0720	0.1360	0.2200
	ifort	0.0680	0.1360	0.2120
	ifort -O3	0.0680	0.1320	0.2120

C	gcc	0.1700	0.3400	0.5700
	gcc -Ofast	0.0900	0.1800	0.2900
	icc	0.0900	0.1800	0.3000
	icc -Ofast	0.0900	0.1800	0.3000

Table 2.1: Elapsed times (in seconds) obtained by copying a matrix using loops.

Language	Option	n=5000	n=7000	n=9000
Python		0.4953	0.9689	1.5962
Python + Numba		0.834	1.29	1.96
Julia		0.2926	0.5471	0.8964
IDL		0.4091	0.8093	1.3315
Matlab		0.2845	0.5841	0.9193
R		2.956	5.785	9.566
Fortran	gfortran	0.0960	0.2480	0.3080
	gfortran -O3	0.0920	0.1840	0.3040
	ifort	0.1200	0.2320	0.3760
	ifort -O3	0.1200	0.2320	0.3880

Table 2.2: Elapsed times (in seconds) obtained by copying a matrix using vectorization.

Problem 3: Matrix Multiplication

We multiply two randomly generated *nxn* matrices A and B:

C=AxB

This problem shows the importance of taking advantage of built-in libraries available in each language. The elapsed times presented here only measure the times spent on the multiplication (as the size of the matrix varies).

Language	Option	n=1500	n=1750	n=2000
Python	intrinsic	0.58	0.96	0.97
Python + Numba	(loop)	3.64	6.33	13.57
Julia	intrinsic	0.1494	0.2391	0.3497
IDL	intrinsic	0.3028	0.3613	0.4797
Matlab	intrinsic	0.9567	0.2575	0.2943
R		0.920	1.158	0.951
JAVA	(loop)	6.8530	13.4700	29.2320
Scala	(loop)	9.258	14.482	23.363
Fortran	gfortran (loop)	17.2450	31.2299	60.1837
	gfortran -O3 (loop)	3.3202	5.3043	12.3367
	gfortran (matmul)	0.3520	0.5600	0.8280
	gfortran -O3 (matmult)	0.3480	0.5560	0.7840
	ifort (loop)	1.1400	1.8081	3.1001
	ifort -O3 (loop)	0.5200	0.8240	1.2760
	ifort (matmul)	1.1400	1.8121	2.9001
	ifort -O3 (matmul)	1.1400	1.8121	2.9881
	ifort (DGEMM)	0.2120	0.2280	0.3320
C	gcc (loop)	13.4900	20.9600	31.4800
	gcc -Ofast (loop)	1.3500	2.3900	4.3700
	icc (loop)	1.2100	2.1600	4.0200
	icc -Ofast (loop)	1.1500	1.7000	2.6600

Table 3.1: Elapsed times (in seconds) obtained by multiplying two randomly generated matrices.

Problem 4: Gauss-Legendre Quadrature

The Gauss-Legendre quadrature formulas approximate the integral of a functionby a weighted sum of function-values. When *m* function-values are used, the formula is exact for polynomials of degree zero through *2m — 1*.

Language	Option	n=50	n=75	n=100
Python		0.1345	0.0183	0.0186
Julia		1.2962	1.3553	1.3556
IDL		0.0006	0.0009	0.0014
R				
JAVA				
Matlab		0.7739	0.7197	0.0853
Fortran	gfortran	0	0.004	0.008
	gfortran -O3	0	0.004	0.008
	ifort	0	0.004	0.008
	ifort -O3	0	0.004	0.008
C	gcc			
	gcc -Ofast			
	icc			
	icc -Ofast			

Table 4.1: Elapsed times (in seconds) obtained by performing the Gauss-Legendre qudrature.

Problem 5: Numerical Approximation of the 2D Laplace Equation

We find the numerical solution of the 2D Laplace equation:

$$U_{xx} + U_{yy} = 0$$

We use the Jacobi iterative solver. We are interested in fourth-order compact finite difference scheme (Gupta, 1984):

$$U_{i,j} = (4(U_{i-1,j} + U_{i,j-1} + U_{i+1,j} + U_{i,j+1}) + U_{i-1,j-1} + U_{i+1,j-1} + U_{i+1,j+1} + U_{i-1,j+1})/20$$

The Jacobi iterative solver stops when the difference of two consecutive approximations falls below 10^{-6}.

Language	**Option**	**n=100**	**n=150**	**n=200**
Python		142.7886	705.268	2188.007
Python + Numba		1.2764	5.4262	16.396
Julia		1.0309	5.1724	16.1657
	optimized	1.0987	5.5039	17.1473
	optimized_smind	0.6215	3.0289	9.4964
IDL		83.6360	416.5523	1298.777
Matlab		1.8199	4.9914	9.1465
R		128.131	635.674	1971.329
JAVA		0.4850	2.0210	5.5980
Scala		0.545	2.289	6.202
Fortran	gfortran	0.840	3.800	10.945
	gfortran -O3	0.668	3.068	8.881
	ifort	0.5360	2.4680	7.1520
	ifort -O3	0.5360	2.4640	7.1520
C	gcc	0.500	2.4200	7.7000
	gcc -Ofast	0.2100	1.0400	3.1800
	gcc -fPIC -Ofast -O3 -xc -shared	1.1410	5.5953	17.3381

	icc	0.4500	2.2300	6.7900
	icc -Ofast	0.3200	1.6000	4.8700

Table 5.1: Elapsed times (in seconds) obtained by numerically solving the Posson equation using a Jacobi iterative solver with loops.

Language	Option	n=100	n=150	n=200
Python		2.3209	10.7638	41.2477
Python + Numba		3.5021	12.5186	36.1285
Julia	optimized_vectorized	2.3787	14.0944	42.1255
IDL		1.9159	10.1320	32.2211
Matlab		3.5102	6.4710	16.4999
R		21.177	102.229	333.366
Fortran	gfortran	0.876	3.948	11.329
	gfortran -O3	0.3560	1.7880	5.0880
	ifort	0.3000	1.5440	4.4400
	ifort -O3	0.2840	1.5680	4.4520

Table 5.2: Elapsed times (in seconds) obtained by numerically solving the Posson equation using a Jacobi iterative solver with vectorization.

Problem 6: Belief Propagation

The Belief Propagation can be applied to fields such as speech recognition, computer vision, image processing, medical diagnostics, parity check codes, etc. Its calculations involve a repeated sequence of matrix multiplications, followed by normalization. The Matlab, C and Julia codes are shown in the Justin Domke's weblog (Domke 2012). We report the computing times for various values of the number of iterations (N) when the matrix dimension is 5000x5000.

Language	Option	n=250	n=500	n=1000
Python		4.8186	7.3240	13.9176
Julia		3.957	7.684	14.855
IDL		18.3229	35.8977	71.0820
Matlab		2.6299	4.0708	6.8691
R		25.463	46.985	92.654
JAVA		321.403	642.395	1284.106
Fortran	gfortran	22.5574	39.9224	89.9696
	gfortran -O3	5.1603	9.5885	18.7051
	ifort	4.6082	8.8605	17.3810
	ifort -O3	4.6322	8.7325	17.4130
C	gcc	2.6400	5.2800	10.5700
	gcc -Ofast	2.3500	4.7200	9.4400
	icc	1.4500	2.9000	5.8000
	icc -Ofast	1.4400	2.9000	5.8100

Table 6.1: Elapsed times (in seconds) obtained by doing the Belief Propagation computations.

Problem 7: Metropolis-Hastings Algorithm

The Metropolis–Hastings (M–H) algorithm is a method for obtaining random samples from a probability distribution. We perform calculations for the implementation of a Metropolis-Hastings algorithm using a two dimensional distribution (Domke 2012). Results are shown when the number of iterations (N) varies.

Language	Option	n=5000	n=10000	n=15000
Python		0.02642	0.0637	0.0937

Julia		0.0002	0.0004	0.0006
IDL		0.0058	0.0219	0.0291
Matlab		0.0164	0.0194	0.0276
R		0.105	0.166	0.24
JAVA		0.006	0.007	0.009
Scala		0.009	0.012	0.014
Fortran	gfortran	0	0.0040	0.0040
	gfortran -O3	0	0	0
	ifort	0	0	0
	ifort -O3	0	0.0040	0
C	gcc	0	0	0
	gcc -Ofast	0	0	0
	icc	0	0	0
	icc -Ofast	0	0	0

Table 7.1: Elapsed times (in seconds) obtained by doing the Metropolis algorithm computations.

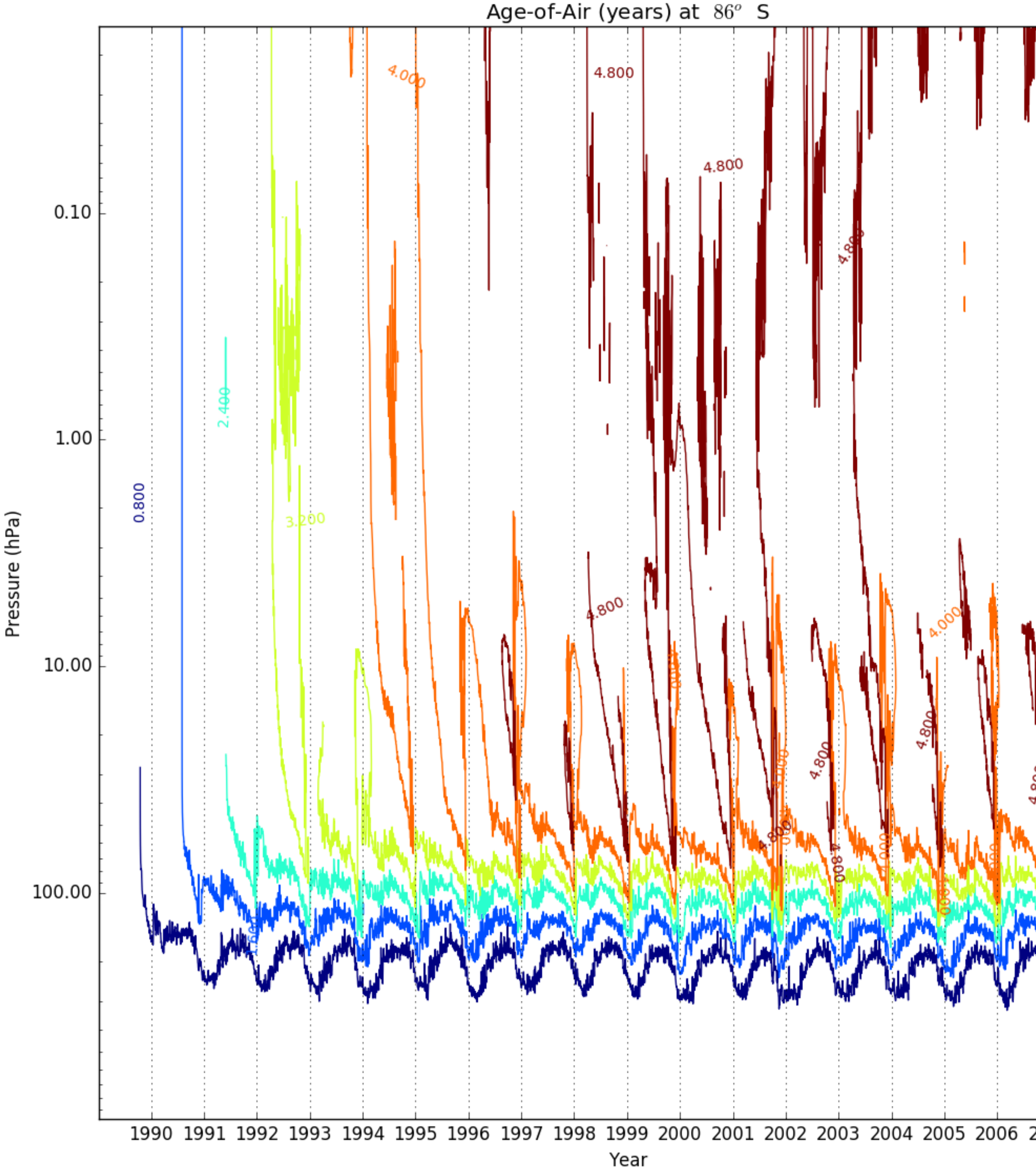
Problem 8: Manipulation of netCDF Files

We have a set of daily NetCDF files (7305) covering a period of 20 years (1990-2009). The files for a given year are in a sub-directory labeled YYYY (for instance 1990, 1991, 1992, etc.). We want to write a script that opens each file, reads a three-dimensional variable (longitude/latitude/level), and manipulates it. A pseudo code for the script reads:

```
Loop over the years
    Obtain the list of NetCDF files
    Loop over the files
        Read the variable (longitude/latitude/level)
```

```
        Compute the zonal mean average (new array of latitude/level)
        Extract the column array at latitude 86 degree South
        Append the column array to a "master" array (or matrix)
```

The goal here is to be able to do a generate the data to do a contour plot that looks like (obtained with Python):


Age-of-Air (years) at 86^o S
Pressure (hPa)
0.10
1.00
10.00
100.00
Year
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
0.800
2.400
3.200
4.000
4.800

This is the kind of problems that a typical user we support faces: a collection of thousands of files that needs to be manipulated to extract the desired information. Having tools that allow us to quickly read data from files (in formats such as NetCDF, HDF4, HDF5, grib) is critical for the work we do.

Note that unlike in Problem 4 of the previous report (where the daily files are in directories associated with months), the daily files to be read in in this case are stored in directories associated with the years. The access to the files is easier in this current problem and we expect the timing numbers to be reduced.

We report in Table 8.1 the elapsed times it took to solve Problem 8 with the various languages.

Language	**Elapsed Time (s)**
Python	558.4496
Julia	580.5683
IDL	504.5634
Matlab	646.2261

Table 8.1: Elapsed time (in seconds) obtained by manipulating 7305 NetCDF files on a single processor.

All the above runs were conducted on a node that has 28 cores. Basically, only one core was used. We want to take advantage of all the available cores by spreading the reading of the files and making sure that the data of interest are gathered in the proper order. We use the multi-processing capabilities of the various languages to slightly modify the scripts. For each year, the daily files are read in by different threads (cores).The results are shown in Table 8.2.

Language	numThreads=2	numThreads=4	numThreads=8	numThreads=16
Python	352.7964	238.1065	170.9945	105.3949

Table 8.2: Elapsed time (in seconds) obtained by manipulating 7305 NetCDF files using multiple threading.

Problem 9: Function Evaluations

We create an array x of length n and loop several times to perform the six operations:

```
y = sin(x)
x = asin(y)
y = cos(x)
x = acos(y)
y = tan(x)
x = atan(y)
```

Language	n=80000	n=90000	n=100000
Python	52.1014	58.4591	64.8276
Julia	55.5550	62.3450	69.2350
IDL	37.4798	42.0187	34.8829
Matlab	5.1866	5.6523	4.6116
R	89.500	101.439	112.269

Problem 10: Simple FFT

We create a nxn random complex matrix M and compute the following:

```
r = fft(M)
r = abs(r)
```

Language	n=10000	n=15000	n=20000
Python	10.5087	25.5764	45.1959
Julia	3.916	11.489	20.632
IDL	16.6154	36.5711	73.3394
Matlab	2.6606	6.0293	10.7011
R	60.722	157.626	269.651

Problem 11: Square Root of a Matrix

We consider an nxn matrix A with 6s on the diagonal and 1s everywhele else. We are lloking for the matrix B such that BxB = A. We record the time for determining B.

Language	Option	n=1000	n=2000	n=4000
Python	SciPy sqrtm	2.2227	5.2814	45.7643
Julia	sqrtm	0.4129	2.511	19.111
Matlab	sqrtm	0.9683	1.3916	2.3767
R		1.057	3.602	19.122

Problem 12: Look and Say Sequence

We write codes to determine the look and say number of order n. Instead of starting with a single digit, we begin with 1223334444.

This test case highlights how languages manipulate strings of arbitrary length.

Language	Options	n=40	n=45	n=48
Python		2.2921	37.4429	224.4362
Julia		2.769	44.333	345.069
IDL		19.9563	296.4768	1570.4234
Matlab		412.5993	4501.6751	
R		0.509	1.678	3.611
Java		0.0487	0.0947	0.1582
Scala		0.0390	0.1020	0.1720
Fortran	gfortran	0.0160	0.0200	0.0200
	gfortran -O3	0.0200	0.0240	0.0240
	ifort	0.0120	0.0160	0.0120
	ifort -O3	0.0160	0.0200	0.0080
C	gcc	0.0800	0.2600	0.5300
	gcc -Ofast	0.0400	0.2500	0.5000
	icc	0.0700	0.2600	0.4800
	icc -Ofast	0.0700	0.2100	0.4600

References

1. Justin Domke, Julia, Matlab and C, September 17, 2012.
2. Michael Hirsch, Speed of Matlab vs. Python Numpy Numba CUDA vs Julia vs IDL, June 2016.
3. Murli M. Gupta, A fourth Order poisson solver, Journal of Computational Physics, 55(1):166-172, 1984.
4. Jean Francois Puget, A Speed Comparison Of C, Julia, Python, Numba, and Cython on LU Factorization, January 2016.
5. Alex Rogozhnikov, Log-likelihood benchmark, September 2015.
6. Sebastian Raschka, Numeric matrix manipulation - The cheat sheet for MATLAB, Python Nympy, R and Julia, June 2014.
7. Yousef Saad, Iterative Methods for Sparse Linear Systems (2 ed.), SIAM, ISBN 0898715342, 200366

Source Files

All the source files for the problems presented here are in the attached file: *sourceFiles2018.tar.gz*

If you have a comment/suggestion/question, contact Jules Kouatchou (Jules.Kouatchou@nasa.gov)