

# A real-time benchmark comparison of the the Simplex, Interior Points and Orthogonal Matching Pursuit decoding algorithms with applications in Compressive Sensing

Bruno Vizcarra, Electrical Engineering Graduate Student  
University of Missouri Kansas City- School of Computing and Engineering  
A Directed Reading Fulfillment  
February 13, 2015

## I. Introduction

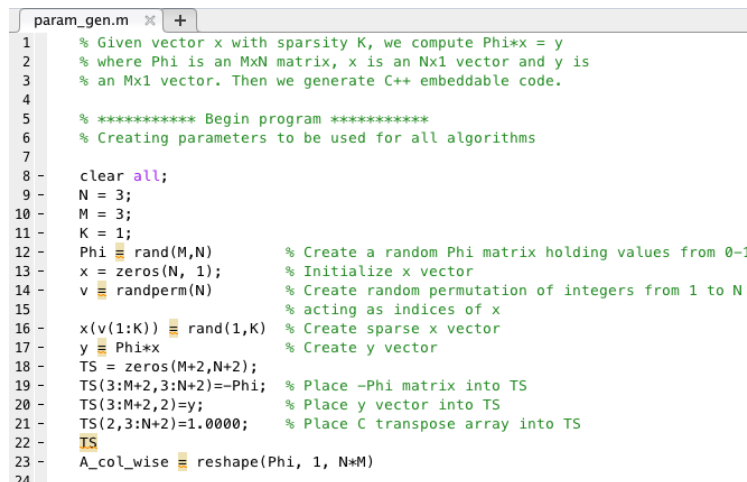
A performance benchmark was assembled for the following solution methods of real-time linear programming: Simplex, Orthogonal Matching Pursuit and Interior Points with the key parameters of interest being execution time and memory utilization.

## II. System overview

All code in this document was compiled on a 21.5 inch iMac with Intel Core i5, 2.7 GHz, 8GB memory and found in the following location: <https://github.com/bvizcarra/compressive-sensing>. The prototypes were created/compiled in Xcode/g++ then ported to Ubuntu Linux 14.04 yielding final experimental data. All theory regarding the implementation of each algorithm has been left for the reader to research although multiple URL's and sources have been included in the Appendix of this document (poster included).

## III. Matlab Matrix/Vector generation function

Included with the source files is the Matlab configuration/parameter generation file param\_gen.m partially shown below:



```
1 % Given vector x with sparsity K, we compute Phi*x = y
2 % where Phi is an MxN matrix, x is an Nx1 vector and y is
3 % an Mx1 vector. Then we generate C++ embeddable code.
4
5 % ***** Begin program *****
6 % Creating parameters to be used for all algorithms
7
8 - clear all;
9 - N = 3;
10 - M = 3;
11 - K = 1;
12 - Phi = rand(M,N) % Create a random Phi matrix holding values from 0-1
13 - x = zeros(N, 1); % Initialize x vector
14 - v = randperm(N) % Create random permutation of integers from 1 to N
15 % acting as indices of x
16 - x(v(1:K)) = rand(1,K) % Create sparse x vector
17 - y = Phi*x % Create y vector
18 - TS = zeros(M+2,N+2);
19 - TS(3:M+2,3:N+2)=-Phi; % Place -Phi matrix into TS
20 - TS(3:M+2,2)=y; % Place y vector into TS
21 - TS(2,3:N+2)=1.0000; % Place C transpose array into TS
22 - TS
23 - A_col_wise = reshape(Phi, 1, N*M)
24
```

Figure 1: Contents of param\_gen.m Matlab file

Using this file we are able to generate the vector, matrix and constant combinations for all algorithm types in a single run. For example, if  $N = 3$ ,  $M = 3$  and  $K = 1$  we get the following parameters:

$$\begin{aligned} \text{Phi} &= \begin{bmatrix} 0.8147 & 0.9134 & 0.2785 \\ 0.9058 & 0.6324 & 0.5469 \\ 0.1270 & 0.0975 & 0.9575 \end{bmatrix} \\ x &= \begin{bmatrix} 0 \\ 0.9572 \\ 0 \end{bmatrix} \\ y &= \begin{bmatrix} 0.8743 \\ 0.6053 \\ 0.0934 \end{bmatrix} \\ \text{TS} &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.0000 & 1.0000 & 1.0000 \\ 0 & 0.8743 & -0.8147 & -0.9134 & -0.2785 \\ 0 & 0.6053 & -0.9058 & -0.6324 & -0.5469 \\ 0 & 0.0934 & -0.1270 & -0.0975 & -0.9575 \end{bmatrix} \end{aligned}$$

As explained in `param_gen.m`,  $\text{Phi}$  is a  $3 \times 3$  matrix,  $x$  is a  $3 \times 1$  vector with sparsity 1,  $y$  is a  $3 \times 1$  vector computed by multiplying  $\text{Phi}$  with  $x$ .  $\text{TS}$  is a well-ordered matrix including vector  $y$  and the negative of matrix  $\text{Phi}$ .

For the Simplex algorithms, a header file named `simplexv2-Nxx-Myy-Kzz.h` is created where the `xx`, `yy` and `zz` parameters are replaced with combinations of  $N$  (length of the optimal feasible vector  $X$ ),  $M$  (length of constraint matrix  $\text{Phi}$ ) and  $K$  (sparsity of  $X$ ) desired. For example, if the above file shown in Figure 1 was run in Matlab, a file called `simplexv2-N3-M3-K1.h` would be output with all important data needed to compile and run either Simplex algorithm.

For the A\* OMP algorithm, three files are generated: `astaromp-x-Nxx-Myy-Kzz.txt`, `astaromp-y-Nxx-Myy-Kzz.txt` and `astaromp-Phi-Nxx-Myy-Kzz.txt` in a similar fashion as the example above.

For the linear points algorithm, a header file named `interiorv1-Nxx-Myy-Kzz.h` is also generated in the similar method as the above two examples.

The point of all of our algorithms is to re-acquire the original  $x$  signal solving the equation:

$$x' = \text{Phi}/y \quad (1)$$

#### IV. Compilation instructions

All code outlined in the following sections may be found in the corresponding drive subdirectires whereupon this document resides.

##### A. Simplex Method

All original code and reference material comes courtesy of Jean-Pierre Moreau [1]. Making sure that the data include file, `simplexv2-N256-M80-K8.h` for example, is present in the same directory (see Fig. 2) as well as referenced within the source file (`simplexv2_inx.cpp`),

```
bruno@bigbrotherul:~/DirectedReading$ ls
param_gen.m      simplexv2_lnx.cpp      tsimplexv2_lnx      tsimplexv2-N3-M3-K1.h
simplexv2_lnx     simplexv2-N3-M3-K1.h  tsimplexv2_lnx.cpp
bruno@bigbrotherul:~/DirectedReading$
```

Figure 2: Contents of Simplex algorithm directory

run the following commands in the source code directory to compile and run the simplex method version 1:

```
$ g++ simplexv2_lnx.cpp -o simplexv2_lnx -Wall -ansi
$ ./simplexv2_lnx
```

The -Wall switch turns on most warning flags for debugging purposes and the -ansi switch specifies the standard our code should conform to. Note that the g++ compiler switches may be modified accordingly.

To compile and run the simplex method version 2, everything remains the same as above except the algorithm originates from another source. To run and compile type:

```
$ g++ tsimplexv2_lnx.cpp -o tsimplexv2_lnx -Wall -ansi
$ ./tsimplexv2_lnx
```

Your output should appear as shown in Fig. 3 complete with Execution time, the original X signal vector, the approximate decoded X' signal, the Mean Square Error, Signal Power, Signal-to-Noise ratio and pertinent memory information. In this example, our algorithm took about 70 microseconds to complete using 204 kB of memory for the data segment (a.k.a. the heap, containing our initialized static data as well as Block Started By Symbol data), 136 kB of memory for the stack segment (storage of automatic identifiers, register variables, and function call information) and 8 kB of memory for the text segment (a.k.a the instruction segment, containing read-only data such as the executable program code and constant data).

```
bruno@bigbrotherul:~/DirectedReading$ ./simplexv2_lnx

Program simplex version 1 3-3-1 results:

M: 3
N: 3
K: 1

Execution time: 69695 nanoseconds

[0] Actual X:[0.0000000000000000] Approx X:[0.0000000000000000]
[1] Actual X:[0.9571670000000000] Approx X:[0.9571662907525118]
[2] Actual X:[0.0000000000000000] Approx X:[0.0000000000000000]

Mean Square Error, MSE:          0.0000000000001677
Signal Power, Ps:                0.3053895552963333
Signal-to-Noise Ratio, SNR:      122.6037982146705332 dB

Size of data segment - VmData:    204 kB
Size of stack segment - VmStk:    136 kB
Size of text segment - VmExe:     8 kB
bruno@bigbrotherul:~/DirectedReading$
```

Figure 3: Output from execution of simplex method version 1

## B. Orthogonal Matching Pursuit (OMP) method

The A\*OMP specific solution was used to implement the OMP algorithm although there are various methods used for OMP. All code and reference material comes courtesy of Nazim Burak Karahanoglu of Sabanci University [2]. Assuming all files are located in your source directory, as

```
bruno@bigbrotherul:~/DirectedReading/astaromp$ ls
AlgorithmInterface.cpp  astaromp-Phi-N3-M3-K1.txt  Chameleon.cpp  GlobalUtil.h  Results.txt  X.txt
AlgorithmInterface.h    astaromp-x-N3-M3-K1.txt    Chameleon.h    main_lnx.cpp  Trie.cpp     Y.txt
AStarDefinitions.h     astaromp-y-N3-M3-K1.txt    ConfigFile.cpp  OutputVectors_old.txt  Trie.h
AStarOMP               BaseAStar.cpp              ConfigFile.h    OutputVectors.txt  TrieNode.cpp
AStarOMPBuilder.h      BaseAStar.h               config_old.txt   Phi.txt          TrieNode.h
AStarOMPBuilder_lnx.cpp BaseOMP.cpp               config.txt       resource.h        VectorMath.cpp
AStarOMPDefs.h         BaseOMP.h                 GlobalUtil.cpp  Results_old.txt   VectorMath.h
bruno@bigbrotherul:~/DirectedReading/astaromp$
```

Figure 4: Contents of Orthogonal Matching Pursuit algorithm directory

shown in Fig 4, and your config.txt file (Fig. 5) has been amended with the proper values of the following six parameters (N, M, TargetFileName, MeasurementsFileName, DictFileName and K).

```
config.txt x AlgorithmInterface.cpp x AStarOMPBuilder_lnx.cpp x BaseAStar.cpp x BaseOMP.cpp x Chameleon.cpp
10
11 [Data_Parameters]
12
13 # number of test vectors
14 NoVectors = 1
15
16 # total dimensions & dictionary size
17 N = 3
18
19 # observation size
20 M = 3
21
22 # Do you want to provide a file for reading sparse target vectors?
23 # read TestFileName if ReadTargetVectors = 1, ignores it if ReadTargetVectors = 0
24 ReadTargetVectors = 1
25
26 # binary(.bin) or text(.txt) file to read sparse target vectors (x)
27 TargetFileName = astaromp-x-N3-M3-K1.txt
28
29 # binary(.bin) or text(.txt) file for measurements (y)
30 MeasurementsFileName = astaromp-y-N3-M3-K1.txt
31
32 # binary(.bin) or text(.txt) file for dictionary/observation matrix (phi)
33 DictFileName = astaromp-Phi-N3-M3-K1.txt
34
35 # Dictionary mode (single or multi). If single, only one dictionary is read from the dictionary file and
36 # is used for all test vectors. If multi, an individual dictionary is read from the dictionary file for each test
37 # vector. In latter, dictionaries should be concatenated.
38 DictMode = multi
39
40 [OutputFiles]
41
42 # binary(.bin) or text(.txt) file to write the reconstructed vectors
43 RecVectorsFileName = OutputVectors.txt
44
45 # text(.txt) file to write evaluation results
46 ResultFile = Results.txt
47
48 [A*OMP_Parameters]
49
50 # sparsity (maximum number of nonzero entries)
51 # search is terminated when K nonzero entries are reached.
52 K = 1
53
54 # allowable error tolerance in reconstruction of measurements
55 # search is terminated when (l2 norm of residue/ l2 norm of observation vector) becomes lower than Eps.
56 Eps = 0.0000009
```

Figure 5: config.txt file to be amended

Compile and run using the following commands as shown here (ignoring all warnings):

```
$ g++ -o astaromp -Wall -ansi -lm *.cpp
$ ./astaromp
```

Your output should appear as shown below in Fig. 6.

```

bruno@bigbrotherul:~/DirectedReading/astaromp$ ./astaromp
Reading parameters from config.txt...
Individual dictionary for each measurement, dictionaries will be read online...
Reading measurement vectors : Reading sparse target vectors :
Initializing A*OMP...
  Max. Non-zero components (K): 1
  Error Tolerance for termination (Eps): 9e-07
  Signal length (N):3
  No observations (M):3
  No Initial Branches (I):3
  No Branches per Extension (B): 2
  No Maximum Paths in Stack (P): 200
  Auxiliary Function Type: Adaptive-Multiplicative
  Alpha: 0.9

Running A*OMP.

Reconstruction finished, reconstructed vectors written in OutputVectors.txt

RECONSTRUCTION RESULTS:
  No. Total Vectors: 1
  Average Normalized Mean Squared Error: 2.48179e-13

  No. Exactly Reconstructed Vectors: 1
  Exact Reconstruction Rate: 100
  A* Search Analysis Results:
    Total Time: 3.891e-06 sec.
    Statistics per vector:
      Average Time: 3.891e-06 sec.
      No. Iterations: 0
      No. Added Branches: 3
      No. Replaced Branches: 0
      No. Equivalent Branches: 0
      No. Ignored Branches: 0

      Size of data segment - VmData:      204 kB
      Size of stack segment - VmStk:      136 kB
      Size of text segment - VmExe:       108 kB
Finished...
bruno@bigbrotherul:~/DirectedReading/astaromp$

```

Figure 6: Output from execution of A\*OMP algorithm

A quick look at the OutputVectors.txt file shows the approximated vector to be  $[0 \ 0.957167 \ 0]$ .

### C. Interior Points method

Although the algorithm did run according to the instruction of the Numerical Recipes sources [3,4,5] using the following commands:

```

$ g++ -o interiorpoints -Wall -ansi -lm *.cpp
$ ./interiorpoints

```

successful compilation and execution was only achieved for a few values. The error message was as follows:

**ERROR: Factorization failed since diagonal is zero in file  
/Users/brunovizcarra/Desktop/Bruno Data/Education/XCode/InteriorPoints/NR1dl.h at line 62**

Further investigation of this error is being implemented in order to obtain the proper benchmark data.

### V. Results discussion

Each algorithm was compiled and executed using increasingly different numbers for N, M and

K (sparsity) for a single, random trial as shown below.

Table 1: Results Comparison of memory usage and execution time								
	N=3,M=3,K=1	N=10,M=6,K=1	N=20,M=11,K=2	N=30,M=11,K=2	N=40,M=11,K=3	N=80,M=16,K=3	N=256,M=80,K=8	
<b>Simplex version 1</b>								
Execution time:	66.34 us	78.8 us	184.3 us	510.4 us	252.4 us	921.6 us	99.38 ms	
Virtual memory size (kB):	3156	3156	3156	3156	3160	3168	3320	
Peak resident set size (kB):	1108	1112	1112	1108	1112	1124	1276	
Resident set size (kB):	1108	1112	1112	1108	1112	1124	1276	
Size of data segment (kB):	212	212	212	212	212	212	212	
Size of stack segment (kB):	88	88	88	88	88	88	88	
Size of text segment (kB):	8	8	8	8	8	8	8	
<b>Simplex version 2</b>								
Execution time:	54 us	61.2 us	89.96 us	109.8 us	168.2 us	483.1 us	57.22 ms	
Virtual memory size (kB):	3160	3160	3160	3160	3164	3172	3328	
Peak resident set size (kB):	1108	1108	1108	1108	1108	1116	1280	
Resident set size (kB):	1108	1108	1108	1108	1108	1116	1280	
Size of data segment (kB):	212	212	212	212	212	212	216	
Size of stack segment (kB):	88	88	88	88	88	88	88	
Size of text segment (kB):	12	12	12	12	12	12	12	
<b>A*OMP</b>								
Execution time:	1.69 us	1.75 us	28.1 us	67.4 us	84.3 us	199.7 us	3.8 ms	
Virtual memory size (kB):	3260	3260	3260	3260	3260	3260	3392	
Peak resident set size (kB):	1372	1372	1372	1380	1380	1388	1532	
Resident set size (kB):	1372	1372	1372	1380	1380	1388	1532	
Size of data segment (kB):	212	212	212	212	212	212	344	
Size of stack segment (kB):	88	88	88	88	88	88	88	
Size of text segment (kB):	112	112	112	112	112	112	112	

	N=3,M=3,K=1	N=4,M=3,K=1	N=4,M=4,K=1	N=5,M=5,K=1	N=9,M=9,K=1
<b>Interior Points</b>					
Execution time:	452 us	Error	589 us	Error	Error
Virtual memory size (kB):					
Peak resident set size (kB):					
Resident set size (kB):					
Size of data segment (kB):					
Size of stack segment (kB):					

Regarding the Interior Points error, it was decided that since the execution time for a small N=3, M=3, K=1 experiment was more than two orders of magnitude of the best performing method, that following this path for results was not necessary. Once again, further investigation of this error will be needed for another comparison attempt.

Comparing the two Simplex algorithm implementations, version 2 which was adapted from the Numerical Recipes 2<sup>nd</sup> Editions source book [3] displays better execution times (roughly more than 40% from Table 2) as the N-M-K combinations grow larger at the expense of about 4K more code memory utilization.

Table 2: Execution time comparison							
	N=3,M=3,K=1	N=10,M=6,K=1	N=20,M=11,K=2	N=30,M=11,K=2	N=40,M=11,K=3	N=80,M=16,K=3	N=256,M=80,K=8
Simplex V2 vs V1	18.18%	21.79%	51.63%	78.63%	33.33%	47.56%	42.42%
A*OMP vs Simplex V2	96.87%	97.13%	68.54%	38.53%	50.00%	58.80%	93.33%

From Table 1 we clearly see that A\*OMP compared to Simplex version 2 yields more than an order of better performance with the results of Table 2 displaying more than 90% faster execution time at both ends of the N-M-K spectrum. This comes at the cost of memory utilization with A\*OMP using about 100K extra memory for code.

## VI. Conclusions

In conclusion, for a real-time system with memory constraints a Simplex version 2 solution may be desired. If minimum execution time with utmost speed is desired, the Orthogonal Matching Pursuit method is an optimal solution.

## VII. Appendix

Below are all the sources used in the document.

[1] [http://jean-pierre.moreau.pagesperso-orange.fr/c\\_linear.html](http://jean-pierre.moreau.pagesperso-orange.fr/c_linear.html)

[2] <http://myweb.sabanciuniv.edu/karahanoglu/research/>

[3] William H. Press, Saul A Teukolsky, William T. Vetterling and Brian P. Flannery, “Minimization and Maximization of Functions”, Numerical Recipes, The Art of Scientific Computing 2<sup>nd</sup> Edition, 1992

[4] William H. Press, Saul A Teukolsky, William T. Vetterling and Brian P. Flannery, “Minimization and Maximization of Functions”, pp. 537-549, Numerical Recipes, The Art of Scientific Computing 3<sup>rd</sup> Edition, 2007

[5] Numerical Recipes, The Art of Scientific Computing 3<sup>rd</sup> Edition, Source Code CD-ROM  
v3.0