

FIRST LINE OF TITLE

SECOND LINE OF TITLE

THIS IS A TEMPORARY TITLE PAGE
It will be replaced for the final print by a version
provided by the service academique.



Thèse n. 1234 2011
présenté le 12 Mars 2011
à la Faculté des Sciences de Base
laboratoire SuperScience
programme doctoral en SuperScience
École Polytechnique Fédérale de Lausanne
pour l'obtention du grade de Docteur ès Sciences
par

Paolino Paperino

acceptée sur proposition du jury:

Prof Name Surname, président du jury
Prof Name Surname, directeur de thèse
Prof Name Surname, rapporteur
Prof Name Surname, rapporteur
Prof Name Surname, rapporteur

Lausanne, EPFL, 2011

Contents

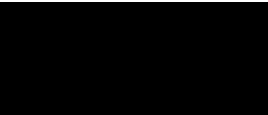
List of figures	iii
List of tables	v
1 Introduction	1
I Torsion	5
2 Geometry of smooth and discret curves	7
2.1 Introduction	7
2.2 Parametric Curves	7
2.3 Frenet's Trihedron	9
2.4 Curvature	10
2.5 Torsion	11
2.6 Curve Framing	12
2.7 Discrete Curvature	18
3 Elastic rod : variational approach	25
3.1 Introduction	25
3.2 Kirchhoff rod	26
3.3 Curve-angle representation	28
3.4 Strains	29
3.5 Elastic energy	29
3.6 Quasistatic assumption	30
3.7 Energy gradient with respect to θ : moment of torsion	30
3.8 Energy gradient with respect to x : internal forces	33
3.9 Numerical Model	42
3.10 Discretization	42
3.11 Connection	42
4 Elastic rod : a novel element from Kirchhoff equations	45
4.1 Introduction	45
4.2 Kirchhoff's law	45

Contents

4.3	Constitutive equations	47
4.4	Internal forces and moments	48
4.5	Equations for the dynamic of rods	48
4.6	Main hypothesis	48
II	Connection	51
5	Calculus of variations	53
5.1	Introduction	53
5.2	Spaces	53
5.3	Derivative	55
5.4	Gradient vector	58
5.5	Jacobian matrix	58
5.6	Hessian	59
5.7	Functional	59
6	Bench for HPC	61
6.1	Introduction	61
6.2	Languages	61
6.3	From syntax to processor	61
6.4	Data Structure	62
6.5	Memory allocation and garbage collection	63

List of Figures

2.1	Different osculating circles for a spiral.	11
2.2	Differential definition of Frenet's trihedron at given point P_0	12
2.3	Geometric torsion and rotation of the osculating plane	13
2.4	Adapted moving frame $F(s) = \{e_3(s), e_1(s), e_2(s)\}$ where $e_3(s) = t(s)$. . .	14
2.5	Geometric interpretation of the Darboux vector of a moving frame.	15
2.7	Variation of the vertex-based discrete curvature.	19
2.8	Variation of the edge-based discrete curvature.	19
2.9	Definition of the osculating circle for discrete curves.	20
2.10	Another definition of the osculating circle for arc-length parametrized curves.	20
2.11	Discrete curvature comparison for $\alpha \in [0.5, 2]$	21
2.12	Another definition of the osculating circle for arc-length parametrized curves.	22
2.13	Discrete curvature comparison for $\alpha \in [0.5, 2]$	23
2.14	Another definition of the osculating circle for arc-length parametrized curves.	23
2.15	Another definition of the osculating circle for arc-length parametrized curves.	24
3.1	Repères de Frenet attachés à γ	34
3.2	\tilde{F} is obtained by rotating \tilde{F}_ϵ around t of an angle Ψ_ϵ	35
3.3	\tilde{F}_ϵ is obtained by parallel transporting F_ϵ from t_ϵ to t . This operation could be seen as a rotation around $t_\epsilon \times t$ of an angle α_ϵ	35
6.1	Nonlinear cost of CPU time in ns/el of memory allocation for arrays (Float64).	65
6.2	Absolute CPU time in <i>ns/element</i> for $n = 104$ elements. Error bars indicate 95% confidence interval.	66
	(a) Float32	66
	(b) Float64	66
6.3	CPU time relative to Intel MKL for $n = 104$ elements.	67
	(a) Float32	67
	(b) Float64	67
6.4	relative CPU time performance for double versus single precision numbers for $n = 10^4$ elements.	68



List of Tables

6.1	Memory allocations for various methods computing $\text{sqrt}(a)$ for $n = 10^4$. . .	63
-----	----------------------------------------------------------------------------------------	----

Torsion Part I

Connection Part II

6 Bench for HPC

6.1 Introduction

In this section aims at providing basic but reliable guidelines to produce fast and mannagable code for our algorithms

Most compilers with which you are probably familiar are standalone programs which take as input some source code text and compile it into machine code (or some other target representation).

cach miss : une donnée n'est pas dans le cache

[Dre07] [Aka12]

L1 = 64kB L2 = 512kB L3 = 4096kB

6.2 Languages

- Csharp - Julia - C++ - Intel MKL - OpenBLAS

Parallelization vs. Vectorization (SIMD)

SIMD :

<http://www.drdobbs.com/windows/64-bit-simd-code-from-c/240168851>

6.3 From syntax to processor

A short story about how a code is translated to get machin instructions

6.4 Data Structure

Array of Structures (AOS) vs Structure of Arrays (SOA)

The most common and likely well-known data structure is the array, which contains a contiguous collection of data items that can be accessed by an ordinal index. This data can be organized as an Array Of Structures (AOS) or a Structure Of Arrays (SOA). While AOS organization is excellent for encapsulation it can be poor for use of vector processing. Selecting appropriate data structures can also make vectorization of the resulting code more effective. To illustrate this point, compare the traditional array-of- structures (AoS) arrangement for storing the r, g, b components of a set of three- dimensional points with the alternative structure-of-arrays (SoA) arrangement for storing this set.

<http://hectorgon.blogspot.fr/2006/08/array-of-structures-vs-structure-of.html>

<http://arxiv.org/pdf/1402.4986.pdf>

sqrt Float64	CPU (ns/el)	Allocation (Bytes)
Allocation	4	8,112
Julia vectorized	9	8,080
Julia broadcast	13	8,352
Julia broadcast!	7	16
Julia map	100	48,000
Julia map!	92	48,000
VML (allocation)	6	8,080
VML (in-place)	4	0

Table 6.1 – Memory allocations for various methods computing $\text{sqrt}(a)$ for $n = 10^4$

6.5 Memory allocation and garbage collection

Toutes les syntaxes ne sont pas égales en terme de gestion de la mémoire. Le problème, c'est le passage de la GC qui est couteux en temps. Donc il faut essayer de minimiser l'utilisation mémoire. Idéalement le problème peut rester dans le cache du processeur (mémoire d'accès bcp plus rapide que la RAM). Donc la meilleur stratégie consiste à pré-allouer les tableaux et à faire des opération "in-place" au maximum, c'est à dire d'écraser les donner au fur et à mesure du calcul.

Par ailleurs les accès mémoires sont lents. Plus la taille du problème reste petite, plus le problème peut être résolu en restant dans le cache. (latency).

<http://stackoverflow.com/questions/4087280/approximate-cost-to-access-various-caches-and-main-memory>

MOST CPU's today uses the memory on multiple level. Generally the memory at the proximity of CPU is costly and less, whereas the memory at the distance (wire distance) is bigger, slower and cheap [1]. Today getting the computer in market with 8GB DRAM is cheap, but L1/L2 cache of such computer is very small in terms of 10's of KB's and few MB's respectively. The access time of L1 (that is generally SRAM) is few cycles whereas L2 is few 10's cycles and accessing main memory is considered a bad programming if accessed too frequently. The access time is huge and in terms of 100's of cycles. So optimizing the code to run and access L1 Instruction and Data cache is the simplest way to optimizing the code.

On remarque que l'allocation mémoire est très différente d'une fonction à l'autre. Il est important de privilégier des opération "in-place" pour contenir l'allocation mémoire, sinon on risque de déclencher la GC qui est couteuse.

Les temps CPU sont indicatifs car le bench est fait sur une durée caractéristique trop courte

Ici on met en évidence la non linéarité du coût d'allocation par élément d'un tableau de taille n . On remarque que la différence entre le coût de sqrt et le coût de l'allocation est constante : c'est le coût de sqrt hors allocation. Attention, cette notion est "language dependent" car les allocations sont gérées par la GC.

Remarque, on trouverait sans doute la même chose pour MKL, à cause du marshalling : le coût d'appel à une fonction C est supérieur à celui d'une fonction managée (cf HPC .Net)

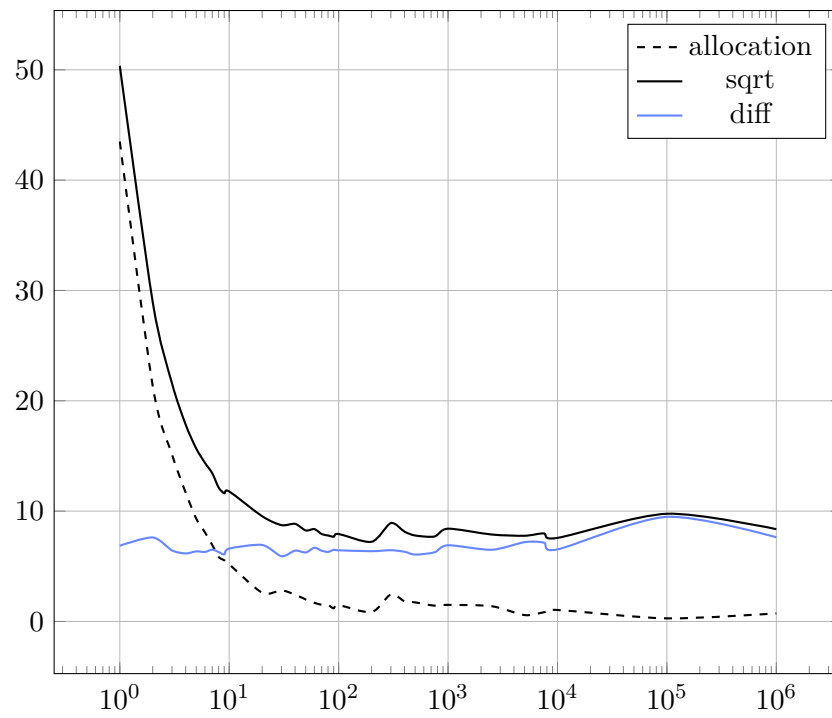


Figure 6.1 – Nonlinear cost of CPU time in ns/el of memory allocation for arrays (Float64).

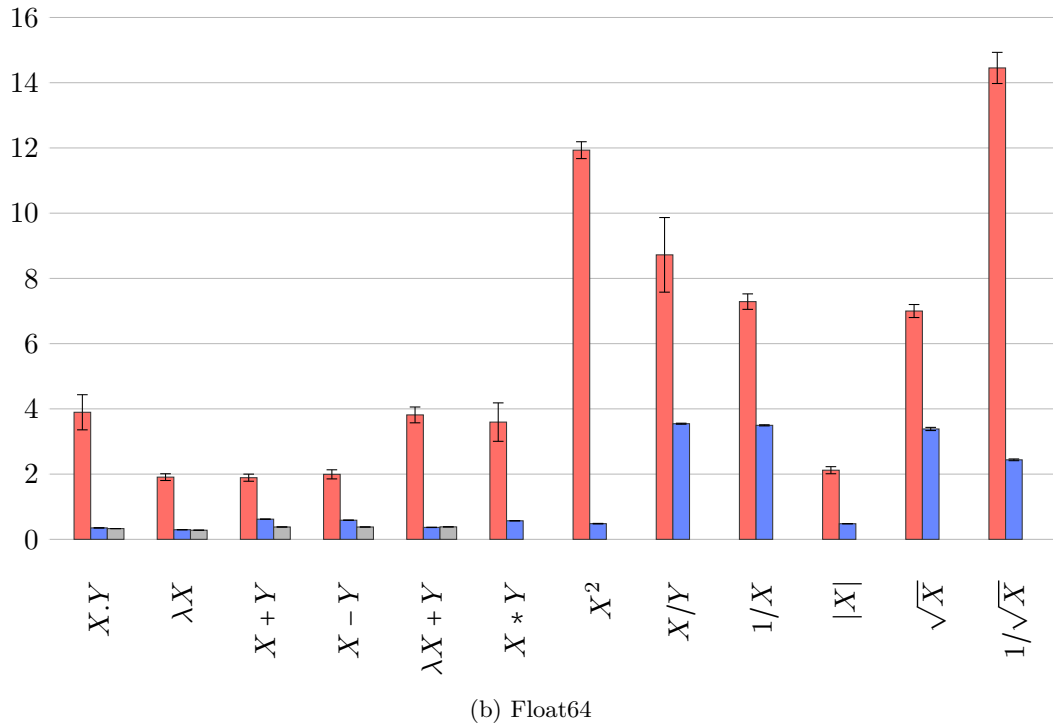
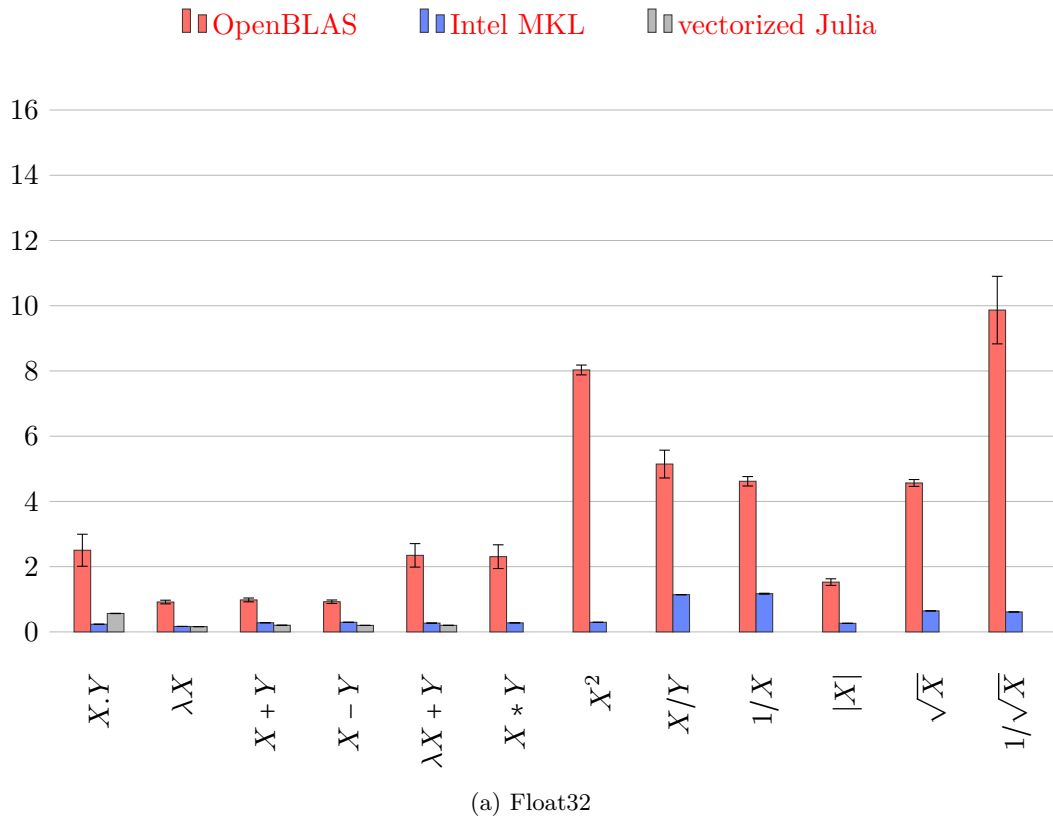


Figure 6.2 – Absolute CPU time in $ns/element$ for $n = 104$ elements. Error bars indicate 95% confidence interval.

6.5. Memory allocation and garbage collection

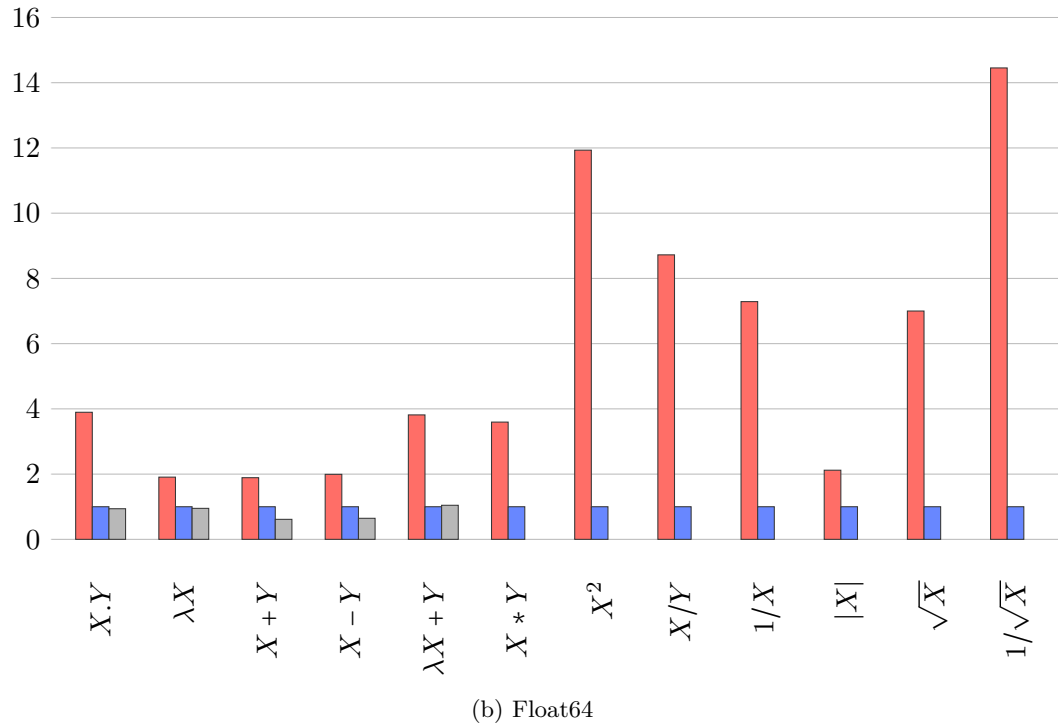
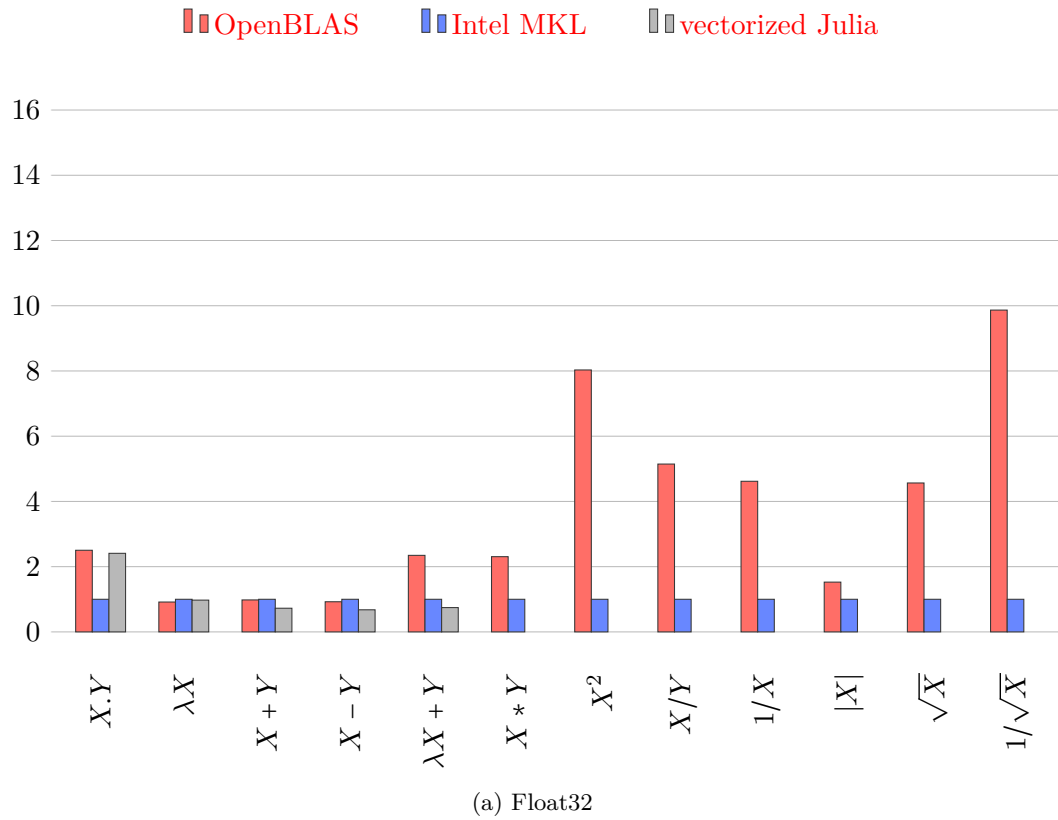


Figure 6.3 – CPU time relative to Intel MKL for $n = 104$ elements.

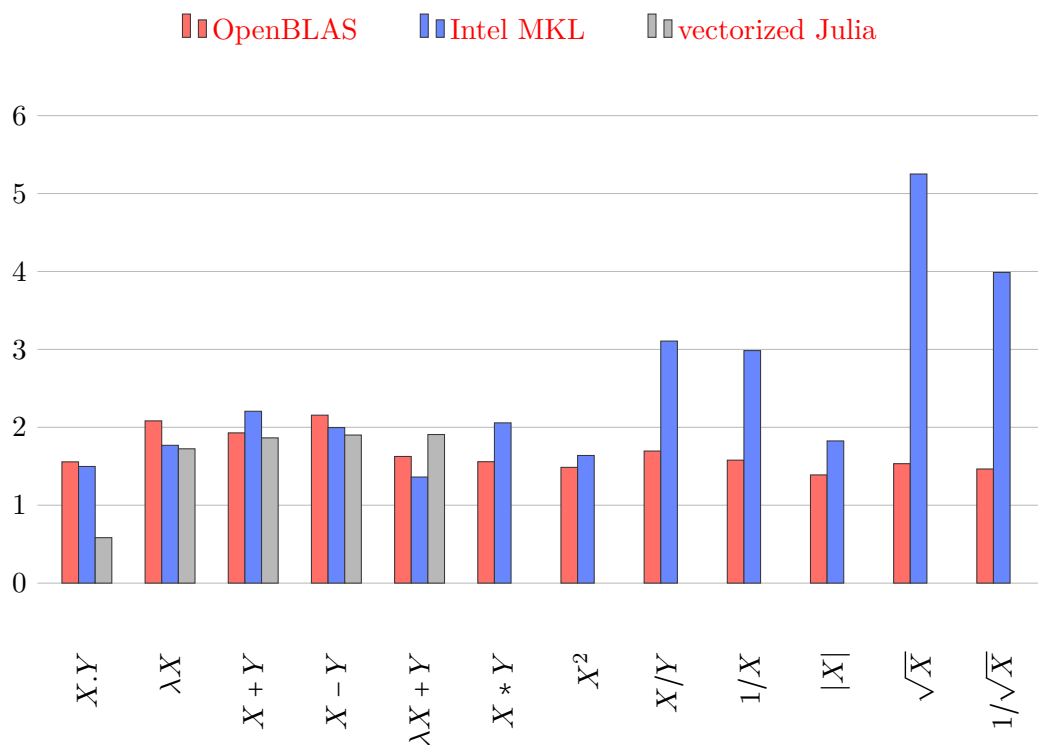


Figure 6.4 – relative CPU time performance for double versus single precision numbers for $n = 10^4$ elements.

profiling : <https://software.intel.com/en-us/intel-vtune-amplifier-xe>

SIMD : <http://www.drdobbs.com/architecture-and-design/simd-enabled-vector-types-with-c/240168888>

<https://software.intel.com/en-us/articles/optimize-for-intel-avx-using-intel-math-kernel-librarys-basic-linear-algebra-subprograms-blas-with-dgemm-routine>

- <https://msdn.microsoft.com/en-us/library/ms973852.aspx>
- <http://www.sebastiansylvan.com/post/why-most-high-level-languages-are-slow/>
- <http://creamysoft.blogspot.fr/2013/05/c-vs-c-performance.html>
- <http://www.codeproject.com/Articles/212856/Head-to-head-benchmark-Csharp-vs-NET>
- <https://software.intel.com/en-us/articles/speeding-up-c-code-with-the-vtune-amplifier-xe-performance-profiler>
- <http://jonathankinlay.com/index.php/2015/02/comparison-programming-languages/>

Bibliography

[Aka12] Eisha Akanksha. Modern CPU ' s memory architecture. *Intelligent Systems and Applications*, 4(April):26–32, 2012.

[Dre07] Ulrich Drepper. What every programmer should know about memory. *Red Hat Magazine*, 2007.

Bibliography

```
1 using VML
2
3 # wrap functions to avoid global scoping while testing
4 function sqrt_jvectorized{T<:Number}(a::Vector{T})
5     sqrt(a)
6 end
7 function sqrt_jbroadcast{T<:Number}(a::Vector{T})
8     broadcast(sqrt,a)
9 end
10 function sqrt_jbroadcast!{T<:Number}(dest::Vector{T}, a::Vector{T})
11     broadcast!(sqrt,dest,a)
12 end
13 function sqrt_jmap{T<:Number}(a::Vector{T})
14     map!(sqrt,a)
15 end
16 function sqrt_jmap!{T<:Number}(dest::Vector{T}, a::Vector{T})
17     map!(sqrt,dest,a)
18 end
19 function sqrt_jloop{T<:Number}(dest::Vector{T},a::Vector{T})
20     @inbounds for i in eachindex(a) dest[i]=sqrt(a[i]) end
21 end
22
23 function sqrt_bench()
24     # define vector size and floating precision
25     n = 1_000
26     T = Float64
27     # allocate vectors
28     dest = ones(T,n)
29     @time a = rand(T,n)
30     # bench
31     gc()
32     gc_enable(false)
33     @time sqrt_jvectorized(a)
34     @time sqrt_jbroadcast(a)
35     @time sqrt_jbroadcast!(dest,a)
36     @time sqrt_jmap(a)
37     @time sqrt_jmap!(dest,a)
38     @time VML.sqrt(a)
39     @time VML.sqrt!(dest,a)
40     gc_enable(true)
41 end
42
43 sqrt_bench()
```

Listing 1 – Example from external file

```

1 using DataFrames, VML
2
3 function sqrt_bench()
4
5     # vector size
6     N = [1,2,3,4,5,6,7,8,9,
7          10,20,30,40,50,60,70,80,90,100,200,300,400,500,750,
8          1_000,2_500,5_000,7_500,10_000,100_000,1_000_000]
9
10    # dataframe for results
11    df = DataFrame(N=[],ALLOC=Float64[],JULIA=Float64[],MKL=Float64[])
12
13    @inbounds for i in 1:length(N)
14        T = Float64
15        n = N[i]
16        a = rand(T,n)
17        dest = zeros(T,n)
18
19        # evaluate sqrt and allocation
20        # for small n @elapsed applies to a bunch of evaluations
21        nrep = 1000
22        ncycle = 10_000 ÷ n + 1
23
24        # trigger garbage collection
25        gc()
26        talloc = 0.0 ; talloc = 0.0 ; tsqrt = 0.0
27        for j in 1:nrep
28            talloc += @elapsed for k in 1:ncycle Vector{T}(n) end
29            tsqrt += @elapsed for k in 1:ncycle sqrt(a) end
30        end
31
32        # scale results (ns/element)
33        talloc = talloc / nrep / ncycle / n * 1e9
34        tsqrt = tcu / nrep / ncycle / n * 1e9
35
36        # write results
37        push!(df,[n,talloc, tsqrt])
38    end
39    df
40 end
41
42 sqrt_bench()

```

Listing 2 – Example from external file

