The Journal of Open Source Software

Castor: A C++ library to code "à la Matlab"

Matthieu Aussal[*1], **Marc Bakry**[†1], **and Laurent Series**[‡2]

1 Ecole Polytechnique (CMAP), INRIA, Institut Polytechnique Paris, Route de Saclay 91128, Palaiseau, France **2** Ecole Polytechnique (CMAP), Institut Polytechnique Paris, Route de Saclay 91128, Palaiseau, France

DOI: 10.21105/joss.03965

Software

- Review
- Repository
- Archive

Editor: Mehmet Hakan Satman

Reviewers:

- @mkitti
- @pitsianis

Submitted: 23 November 2021
Published: 16 March 2022

Summary

The objective of the *Castor* framework is to propose high-level semantics, inspired by the Matlab language, allowing fast software prototyping in a low-level compiled language. It is nothing more than a matrix management layer using the tools of the standard C++ library (C++14 and later), in different storage formats (full, sparse and hierarchical). Linear algebra operations are built over the BLAS API and graphic rendering is performed in the VTK framework. The *Castor* framework is provided as an open source software under the LGPL 3.0, compiled and validated with clang and gcc.

Statement of need

Matlab is a software used worldwide in numerical prototyping, due to its particularly user-friendly semantics and its certified toolboxes. However, many use cases do not allow codes in Matlab format, for example multi-platform portability issues, proprieraty licensing and more generally code interfacing. To start meeting these needs, a header-only template library for matrix management has been developed, based on the standard C++14 and later library, by encapsulating the `std::vector` class. Many tools and algorithms are provided to simplify the development of prototypes:

- dense, sparse and hierarchical matrices manipulations,
- linear algebra computations (Anderson et al., 1999),
- graphical representations (Schroeder et al., 2000).

This high-level semantic/low-level language coupling makes it possible to gain efficiency in the developpement, while ensuring performance for applications. In addition, direct access to data structures allows users to optimize the most critical parts of their code. Finally, a complete documentation is available, as well as continuous integration unit tests. All of this makes it possible to meet the needs of teaching (notebooks using a C++ interpreter such as Cling), academic research and industrial applications at the same time.

State of the field

For a developer accustomed to the Matlab language, it is natural to turn to prototyping tools such as Numpy or Julia, to produce open-source codes. Indeed, these tools today offer similar semantics and performance, with well-established user communities. To illustrate this similarity, the following codes perform the same tasks, with one implementation in Matlab (MATLAB, 2010) (left) and another in Julia (Bezanson et al., 2012) (right) :

[*]matthieu.aussal@polytechnique.edu
[†]marc.bakry@polytechnique.edu
[‡]laurent.series@polytechnique.edu

Matlab	Julia
	`using LinearAlgebra`
`tic`	`function test()`
`M = [ 1  2  3 ;`	`M = [ 1  2  3 ;`
`4  5  6 ;`	`4  5  6 ;`
`7  8  9 ;`	`7  8  9 ;`
`10 11 12];`	`10 11 12];`
`disp(M);`	`display(M);`
`M = (M - 1) .* ...`	`M = (M .- 1) .*`
`eye(size(M));`	`Matrix(I,size(M));`
`M(1,1) = -1;`	`M[1,1] = -1;`
`M([2,3],1)  = -1;`	`M[[2 3],1] .= -1;`
`M(4,:) = -1;`	`M[4,:] .= -1;`
`disp(M);`	`display(M);`
`disp(sum(M,2));`	`display(sum(M,dims=2));`
`disp(abs(M));`	`display(abs.(M))`
`disp(sort(M,1));`	`display(sort(M,dims=1));`
`disp(M*M');`	`display(M*M');`
	`end`
`toc`	`@time test();`
`disp("done.");`	`display("done.");`

Despite the many advantages that these languages have and their high popularity, many codes are still developed natively in Fortran, C, and C++, for practical or historical reasons. Even if there are tools to automatically generate C/C++ code from a high-level language (as *Matlab Coder*), this work is often done manually by specialists. To find high-level semantics in native C++, we can turn to libraries like Eigen (Guennebaud et al., 2010), which offers a matrix API and efficient algebra tools. However, as the comparison below shows, the transcription from a Matlab code to an Eigen-based C++ code is not immediate:

```cpp
#include <iostream>
#include <chrono>
#include <eigen-3.4.0/Eigen/Dense>
using namespace std::chrono;
using namespace Eigen;
int main()
{
    auto tic = high_resolution_clock::now();

    MatrixXd  M(4,3);
    M << 1,  2,  3,
         4,  5,  6,
         7,  8,  9,
        10, 11, 12;
    std::cout << M << std::endl;

    M.array() -= 1;
    M.array() *= MatrixXd::Identity(M.rows(),M.cols()).array();
    M(0,0)     = -1;
    M.block<2,1>(1,0) = -MatrixXd::Ones(2,1);
```

```
    M.row(3)   = -MatrixXd::Ones(1,3);
    std::cout « M « std::endl;

    std::cout « M.rowwise().sum() « std::endl;
    std::cout « M.array().abs() « std::endl;
    MatrixXd  Ms = M;
    for(auto col : Ms.colwise())
    {
      std::sort(col.begin(), col.end());
    }
    std::cout « Ms « std::endl;
    std::cout « M * M.transpose() « std::endl;

    auto toc  = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(toc - tic);
    std::cout « "Elapsed time: " « duration.count()*1e-6 « std::endl;
    std::cout « "done." « std::endl;

    return 0;
}
```

To complete this example, other references are available on this link. This is why all the features of the Castor library have been designed and developed so that the semantics at user level are as close to Matlab as what C++ allows. Moreover, to gain in portability, the manipulations of full matrices and the main algorithms depend only on the standard library which is available on the most majority of operating systems (MacOS, Linux, Windows, Android, etc.). Only advanced linear algebra tools require an external BLAS / LAPACK API, as well as graphical visualization functionality (VTK). The example below illustrates this goal:

Matlab	Castor

	`#include "castor/matrix.hpp"`
	`using namespace castor;`
	`int main (int argc, char* argv[ ])`
	`{`

`tic`	`tic();`

`M = [ 1  2  3 ;`
`  4  5  6 ;`
`  7  8  9 ;`
`  10 11 12];`
	`M = {{ 1,  2,  3},`
	`  { 4,  5,  6},`
	`  { 7,  8,  9},`
	`  {10, 11, 12}};`

`disp(M);`	`disp(M);`

`M = (M - 1) .* eye(size(M));`
`M(1,1) = -1;`
`M([2,3],1)  = -1;`
`M(4,:) = -1;`
`disp(M);`
	`M = (M - 1) * eye(size(M));`
	`M(0,0) = -1;`
	`M({1,2},0) = -1;`
	`M(3,col(M)) = -1;`
	`disp(M);`

`disp(sum(M,2));`
`disp(abs(M));`
`disp(sort(M,1));`
`disp(M*M');`
	`disp(sum(M,2));`
	`disp(abs(M));`
	`disp(sort(M,1));`
	`disp(mtimes(M,transpose(M)));`

`toc`	`toc();`

Matlab	Castor

`disp("done.");`	`disp("done.");`
	
	`return 0;`
	`}`

Note: It is important to specify that the Castor library is far from offering today all the functionalities offered by Matlab and its many toolboxes.

Dense Matrix

The dense matrix part of the *Castor* framework implements its own templatized class `matrix<T>` in `matrix.hpp`, where `T` can be fundamental types of C++ as well as `std::complex`. This class is built over a `std::vector<T>` which holds the values (ISO/IEC, 2014). Note that the element of a matrix is stored in row-major order and that a vector is considered as a $1 \times n$ or $n \times 1$ matrix.

The class `matrix<T>` provides many useful functions and operators such as:

- builders which can be used to initialize all coefficients (`zeros`, `ones`, `eye`, etc.),
- standard algorithms over data stored in matrices (`norm`, `max`, `sort`, `argsort`, etc.),
- mathematical functions which can be applied element-wise (`cos`, `sqrt`, `conj`, etc.),
- matrix manipulations like concatenate matrices in all dimensions, find the non-zero elements or transpose them, reshape size, etc.,
- standard C++ operators which have been overloaded and work element-wise (`+`,`*`,`!`,`&&`,etc.),
- values accessors and matrix views with linear and bi-linear indexing,
- elements of linear algebra, such as the matrix product (`mtimes` or `tgemm`) and linear system resolution (multi-right-hand-side `gmres`),
- many other tools to display elements (`<<`, `disp`), save and load elements from file (ASCII or binary), etc.

The API provides more than a hundred functions and is designed such that it should feel like using Matlab. For advanced users, direct access to the data stored in the `std::vector<T>` enables all or part of an algorithm to be optimized in native C++.

This example displays the sum of two matrices with implicit cast :

```
#include <iostream>
#include "castor/matrix.hpp"
using namespace castor;
int main(int argc, char* argv[])
{
    matrix<float> A = {{ 1.0,  2.0,  3.0,  4.0},
                       { 5.0,  6.0,  7.0,  8.0},
                       { 9.0, 10.0, 11.0, 12.0}};
    matrix<double> B = eye(3,4);
    auto C = A + B;
    disp(C);
    return 0;
}
```

```
Matrix 3x4 of type 'd' (96 B):
    2.00000      2.00000      3.00000      4.00000
    5.00000      7.00000      7.00000      8.00000
    9.00000     10.00000     12.00000     12.00000
```

Linear Algebra

The linear algebra part of the framework, implemented in `linalg.hpp`, provides a set of useful functions to perform linear algebra computations by linking to optimized implementations of the BLAS and LAPACK standards (Anderson et al., 1999) (OpenBLAS, oneAPI MKL, etc.).

The BLAS part is a straightforward overlay of the C-BLAS type III API, which is compatible with row-major ordering. This is achieved by a template specialization of the `tgemm` function, which allows optimized implementations to take control of the computation using `sgemm`, `dgemm`, `cgemm` and `zgemm`. Thanks to this interface, naive implementations proposed in `matrix.hpp` for dense matrix-products `mtimes` and `tgemm` may be improved in term of performance, especially for large matrices.

The LAPACK part is a direct overlay over the Fortran LAPACK API, which uses a column ordering storage convention. This interface brings new high-level functionalities, such as a linear solver (`linsolve`), matrix inversion (`inv`, `pinv`), factorizations (`qr`, `lu`), the search for eigen or singular values decompositions (`eig` ,`svd`), aca compression (`aca`), etc. It uses templatized low-level functions following the naming convention close to the LAPACK one (like `tgesdd`, `tgeqrf`, etc.).

This example displays the product of A and A^{-1} :

```
#include <iostream>
#include "castor/matrix.hpp"
#include "castor/linalg.hpp"
using namespace castor;
int main (int argc, char* argv[])
{
    matrix<> A = rand(4);
    matrix<> Am1 = inv(A);
    disp(mtimes(A,Am1));
    return 0;
}
```

```
Matrix 4x4 of type 'd' (128 B):
 1.0000e+00   1.0408e-16  -2.7756e-17  -5.5511e-17
          0   1.0000e+00  -5.5511e-17   1.1102e-16
          0  -2.2204e-16   1.0000e+00  -1.1102e-16
-2.7756e-17            0            0   1.0000e+00
```

Note: The backslash operator (\) not being available, the `linsolve` function allows to solve linear systems with:

- LU decomposition with partial pivoting and row interchanges for square matrices (`[sdcz]gesv`),
- QR or LQ factorization for overdetermined or underdetermined linear systems (`[sdcz]gels`).

In addition, an iterative multi-right-hand-side solver `gmres` is available in `matrix.hpp`, without dependency on BLAS and LAPACK.

2D/3D Visualization

The graphic rendering part, provided by `graphics.hpp`, features 2D/3D customizable plotting and basic mesh generation. It is based on the well-known VTK library (Schroeder et al., 2000). Here again, the approach tries to get as close as possible to Matlab semantics.

First, the user creates a `figure`, which is a dynamic container of data to display. The `figure`

class is composed of a `vtkContextView` class, providing a view with a default interactor style, renderer, etc. Then, graphic representations can be added to the figure, using functions like `plot`, `imagesc`, `plot3`, `mesh`, etc. Options are available to customize the display of the results, such as the plotting style, legend, colorbar and others basic stuff. Finally, the `drawnow` function must be called to display all defined figures. The latters are displayed and manipulated in independent windows.

In addition, graphics exports are available in different compression formats (`png`,`jpg`, `tiff`, etc.), as well as video rendering (`ogg`).

This example shows a basic 2D plotting of a sine function (Figure 1):

```
#include "castor/matrix.hpp"
#include "castor/graphics.hpp"
using namespace castor;
int main (int argc, char* argv[])
{
    matrix<> X = linspace(0,10,100);
    figure fig;
    plot(fig,X,sin(X),{"r-+"},{"sin(x)"});
    plot(fig,X,cos(X),{"bx"},{"cos(x)"});
    drawnow(fig);
    return 0;
}
```

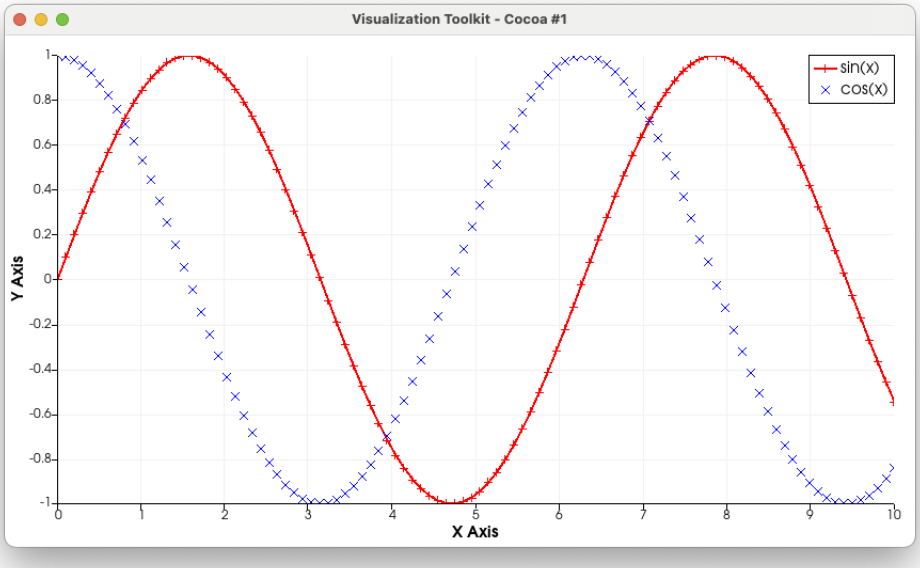

Figure 1: Basic 2D plotting from Castor (using VTK).

Sparse matrices

Some matrices have sparse structures, with many (many) zeros that do not need to be stored (Tewarson, 1973). There are adapted storage formats for this type of structure (LIL, COO, CSR, etc.), the most natural being to store the indices of rows and columns for each non-zero value, as a list of triplet $\{i, j, v\}$. For the *Castor* framework, a dedicated template class to this kind of matrix has been developed (see `smatrix.hpp`). The storage format is based on

a row major sorted linear indexing. Only non-zero values and their sorted linear indices are stored in a list of pairs $\{v, l\}$: for a $m \times n$ matrix, the following bijection is used to switch with the common bilinear indexation:

$$\{i, j\} \to l = i \cdot n + j,$$
$$l \to \{i = \frac{l}{n}; j = i \bmod n\}.$$

Accessors to all the elements are provided so that sparse matrices can be manipulated in a similar way as the dense matrices. This operation is performed by dichotomy with a convergence in $\log_2(\text{nnz})$, where nnz is the number of non-zero elements. Just like dense matrices, numerical values are stored in a templatized `std::vector<T>`. For convenience, we provide classical builders (`sparse`, `speye`, `spdiags`, etc.), standard C++ operators overloading, views, display functions (`disp`, `spy`) and some linear algebra tools (`transpose`, `mtimes`, `gmres`, etc.).

This example displays the sum of two sparse matrices, with implicit cast and sparse to dense conversion :

```
#include <iostream>
#include "castor/smatrix.hpp"
using namespace castor;
int main (int argc, char* argv[])
{
    smatrix<float> As = {{0.0,  0.0,  0.0},
                         {5.0,  0.0,  7.0}};
    As(0,1) = 2.0;
    smatrix<double> Bs = speye(2,3);
    disp(As);
    disp(As(0,1)); // bilinear accessor
    disp(As(4));   // linear accessor
    disp(Bs);
    disp(full(As+Bs));
    return 0;
}
```

```
Sparse matrix 2x3 of type 'f' with 3 elements (12 B):
(0,1)  2
(1,0)  5
(1,2)  7
2
0
Sparse matrix 2x3 of type 'd' with 2 elements (16 B):
(0,0)  1
(1,1)  1
Matrix 2x3 of type 'd' (48 B):
    1.00000      2.00000            0
    5.00000      1.00000      7.00000
```

Hierarchical matrices

To widen the field of applications, the $\mathcal{H}$-matrix format, so-called hierachical matrices (Hackbusch, 1999), have been added in `hmatrix.hpp`. They are specially designed for matrices with localized rank defaults. It allows a fully-populated matrix to be assembled and stored in a lighter format by compressing some parts of the original dense matrix using a low-rank representation (Rjasanow, 2002). They are constructed by binary tree subdivisions in a recursive way, with a

parallel assembly of the compressed and full leaves (using the OpenMP standard). This format features a complete algebra, from elementary operations to matrix inversion. An example is given in the application section that follows.

Application with a FEM/BEM simulation

As an application example, an acoustical scattering simulation was carried out using a boundary element method (BEM) tool, implemented with the *Castor* framework (see the *fembem* package (Aussal & Bakry, 2021)). We consider a smooth n-oriented surface Γ of some object Ω, illuminated by an incident plane wave u_i with wave-number k. The scattered field u satisfies the Helmholtz equation in Ω, Neumann boundary conditions (*sound-hard*) and the Sommerfeld radiation condition:

$$\begin{aligned} -(\Delta u + k^2 u) &= 0 \\ -\partial_n u_i &= 0 \\ \lim_{r\to+\infty} r\ (\partial_r u - iku) &= 0 \end{aligned}$$

The scattered field u satisfies the integral representation (Neumann interior extension, see (Terrasse & Abboud, 2013)):

$$u(\mathbf{x}) = -\left(\frac{1}{2}\mu(\mathbf{x}) + \int_\Gamma \partial_{n_y} G(\mathbf{x},\mathbf{y})\mu(\mathbf{y}) d_y\right) \quad \forall \mathbf{x} \in \Gamma^+, \tag{1}$$

for some density μ, with the Green kernel $G(\mathbf{x},\mathbf{y}) = \dfrac{e^{ik|x-y|}}{4\pi|x-y|}$. Using the boundary conditions we obtain :

$$-H\mu(\mathbf{x}) = -\partial_n u_i(\mathbf{x}) \quad \forall \mathbf{x} \in \Gamma, \tag{2}$$

where the hypersingular operator H is defined by:

$$H\mu(\mathbf{x}) = \int_\Gamma \partial_{n_x}\partial_{n_y} G(\mathbf{x},\mathbf{y})\mu(\mathbf{y}) d_y. \tag{3}$$

The operator H is assembled using a P_1 finite element discretization on a triangular mesh of the surface Γ, stored using dense matrices (`matrix.hpp`) or hierarchical matrices (`hmatrix.hpp`).

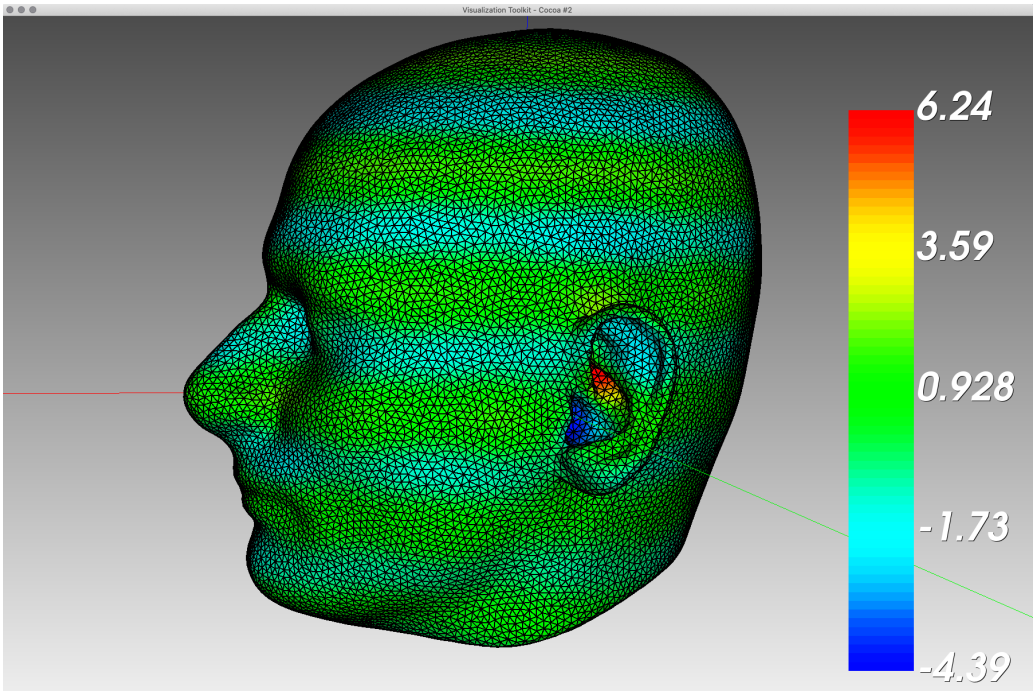

Figure 2: Resonance mode at 8kHz of the human pinna (BEM with H-Matrix).

Finally, using all the tools provided by Castor to write and solve these equations, we are able to efficiently compute the acoustic diffraction of a harmonic plane wave at 8kHz, on a human head mesh (Jin et al., 2013). As shown in Figure 2, the simulation result highlights the role of the auditory pavilion as a resonator, modifying the timbre of a sound source to allow a listener's brain to precisely locate its direction.

```
#include <castor/matrix.hpp>
#include <castor/smatrix.hpp>
#include <castor/hmatrix.hpp>
#include <castor/linalg.hpp>
#include <castor/graphics.hpp>
#include "fem.hpp"
#include "bem.hpp"
using namespace castor;
int main (int argc, char* argv[])
{
    // Load meshes
    matrix<double> Svtx;
    matrix<size_t> Stri;
    std::tie(Stri,Svtx) = triread("./","Head03_04kHz.ply");

    // Graphical representation
    figure fig;
    trimesh(fig,Stri,Svtx);

    // Parameters
    matrix<double> U = {0,0,-1};
    double f = 2000;
    double k = 2*M_PI*f/340;
    float tol = 1e-3;
```

```
    // FEM and mass matrix, sparse storage
    tic();
    femdata<double> v(Stri,Svtx,lagrangeP1,3);
    femdata<double> u(Stri,Svtx,lagrangeP1,3);
    auto Id = mass<std::complex<double>>(v);
    toc();

    // Left hand side '-H', equation (3), H-Matrix storage
    tic();
    auto LHSfct = [&v,&u,&k](matrix<std::size_t> Ix, matrix<std::size_t> Iy)
    {
        return -hypersingular<std::complex<double>>(v,u,k,Ix,Iy);
    };
    hmatrix<std::complex<double>> LHS(v.dof(),u.dof(),tol,LHSfct);
    toc();
    disp(LHS);

    // Right hand side '-dnUi', equation (2), full storage
    auto B = - rightHandSide<std::complex<double>>(v,dnPWsource,U,k);

    // Solve '-H = -dnUi', equation (2), H-matrix preconditionner
    // associated to iterative solver
    hmatrix<std::complex<double>> Lh,Uh;
    tic();
    std::tie(Lh,Uh) = lu(LHS,1e-1);
    toc();
    disp(Lh);
    disp(Uh);
    auto mu = gmres(LHS,B,tol,100,Lh,Uh);

    // Boundary radiation, equation (1)
    tic();
    auto Dbndfct = [&v,&u,&k,&Id](matrix<std::size_t> Ix, matrix<std::size_t> Iy)
    {
        return 0.5*eval(Id(Ix,Iy)) + doubleLayer<std::complex<double>>(v,u,k,Ix,Iy);
    };
    hmatrix<std::complex<double>> Dbnd(v.dof(),u.dof(),tol,Dbndfct);
    matrix<std::complex<double>> Pbnd = mtimes(Dbnd,mu);
    toc();
    Pbnd = - gmres(Id,Pbnd,tol,100) + planeWave(v.dof(),U,k);

    // Graphical representation
    figure fig2;
    trimesh(fig2,Stri,Svtx,abs(Pbnd));

    // Export in .vtk file
    triwrite("./","head.vtk",Stri,Svtx,real(Pbnd));

    // Plot
    drawnow(fig);
    disp("done !");
    return 0;
}
```

Acknowledgements

We thank Houssem Haddar for his precious help.

References

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., & Sorensen, D. (1999). *LAPACK Users' guide* (Third). Society for Industrial and Applied Mathematics. ISBN: 0-89871-447-8

Aussal, M., & Bakry, M. (2021). https://gitlab.labos.polytechnique.fr/leprojetcastor/fembem

Bezanson, J., Karpinski, S., Shah, V. B., & Edelman, A. (2012). Julia: A fast dynamic language for technical computing. *arXiv Preprint arXiv:1209.5145*.

Guennebaud, G., Jacob, B., & al., et. (2010). *Eigen v3*. http://eigen.tuxfamily.org.

Hackbusch, W. (1999). A sparse matrix arithmetic based on H-matrices. Part 1: Introduction to H-matrices. *Computing, 62(2), 89–108*.

ISO/IEC. (2014). International standard ISO/IEC 14882:2014(e) – Programming language C++. *Geneva, Switzerland: International Organization for Standardization*.

Jin, C. T., Guillon, P., Epain, N., Zolfaghari, R., van Schaik, A., Tew, A. I., & Thorpe, J. (2013). Creating the Sydney York morphological and acoustic recordings of ears database. *IEEE Transactions on Multimedia, 16(1), 37-46*. https://doi.org/10.1109/icme.2012.93

MATLAB. (2010). *Version 7.10.0 (R2010a)*. The MathWorks Inc.

Rjasanow, S. (2002). Adaptive cross approximation of dense matrices. *Int. Association Boundary Element Methods Conf., IABEM (Pp. 28-30)*.

Schroeder, W. J., Avila, L. S., & Hoffman, W. (2000). Visualizing with VTK: A tutorial. *IEEE Computer Graphics and Applications, 20(5), 20-27*. https://doi.org/10.1109/38.865875

Terrasse, I., & Abboud, T. (2013). Modélisation des phénomènes de propagation d'ondes. *École Polytechnique*.

Tewarson, R. P. (1973). Sparse matrices. *New York: Academic Press, Vol. 69*.