Making simulations with the MNPBEM toolbox big: Hierarchical matrices and iterative solvers

Ulrich Hohenester

Institute of Physics, University of Graz, Universitätsplatz 5, 8010 Graz, Austria

Abstract

MNPBEM is a Matlab toolbox for the simulation of metallic nanoparticles using a boundary element method (BEM) approach [Comp. Phys. Commun. 183, 370 (2012)], which is currently used by many research groups in the field of plasmonics. In this paper we introduce an extension for more efficient and faster simulations of large nanoparticles with several thousand to ten thousand boundary elements. Our approach is based on hierarchical matrices, for matrix compression and faster matrix manipulations, as well as iterative solvers for the BEM working equations. We discuss implementation details and present results for a few selected plasmonics applications.

Keywords: Plasmonics, metallic nanoparticles, boundary element method, hierarchical matrices, iterative solver

Program summary

Program title: `MNPBEM` toolbox
Programming language: Matlab 8.6.0 (R2015b)
Computer: Any which supports Matlab 8.6.0 (R2015b)
Operating system: Any which supports Matlab 8.6.0 (R2015b)
RAM required to execute with typical data: $\geq$ 8 GByte
Has the code been vectorised or parallelized?: yes
Keywords: Plasmonics, boundary element method, hierarchical matrices and iterative solvers
CPC Library Classification: Optics
External routines/libraries used: no
Nature of problem: Simulation of plasmonic nanoparticles using hierarchical matrices and iterative solvers
Solution method: Boundary element method using electromagnetic potentials
Running time: Depending on surface discretization between minutes and hours

1. Introduction

Computational electrodynamics is concerned with the numerical solution of Maxwell's equations in presence of dielectric or metallic bodies. The most popular simulation aproaches include the finite difference time domain (FDTD) [1, 2] scheme, the finite element method (FEM) [3, 4], and the discrete dipole approximation (DDA) [5]. In contrast to these general solvers, which make practically no assumptions about the dielectric bodies under study, the boundary element method (BEM) [6, 7] approach starts from the outset with an assumption and considers only dielectric bodies with homogeneous dielectric functions that are

Email address: `ulrich.hohenester@uni-graz.at` (Ulrich Hohenester)
URL: `http://physik.uni-graz.at/~uxh` (Ulrich Hohenester)

separated by abrupt interfaces. Whether this assumption is applicable or not depends on the system under study. However, if applicable BEM improves on the other schemes as only the boundaries of the dielectric bodies need to be discretized, and not their entire volumes.

In the field of plasmonics [8], which investigates light interactions with metallic nanoparticles, the BEM approach has a longstanding tradition [9–11], probably because of two main reasons: firstly, the assumption of homogeneous dielectric functions is well met for metallic nanoparticles embedded in a dielectric background; secondly, the surface charges and currents entering the BEM approach can be interpreted in terms of surface plasmons, these are coherent electron charge oscillations at the metal interface which are the workhorse of plasmonics [8]. Although BEM solvers have been developed and routinely used by several research groups in the field of plasmonics [11–13], there are unfortunately very few solvers freely available.

Back in 2012 we published our Matlab toolbox MNPBEM for the simulation of plasmonic nanoparticles [14], which builds on the methodology developed by Javier García de Abajo and Archie Howie [7]. The toolbox was extremely well received by the plasmonics community, and has since then been used by numerous research groups. In the last couple of years we have added further features to the toolbox, such as the simulation of electron energy loss spectroscopy (EELS) [15] or the consideration of substrate and layer effects [16]. Currently an alternative open-source Python and C++ library, named BEM++, is developed at the University College London [17, 18]. Conceptually this approach is probably more ambitious than ours, as the authors aim for a generic BEM solver that can cope with problems ranging from acoustics over electrostatics to the simulation of Maxwell's equations, and also computationally the Galerkin scheme [19] underlying the BEM++ library improves on our more simple collocation approach [14]. On the other hand, several BEM++ features are still under development and simulations of EELS or of stratified background media are currently not implemented, so it has to be seen how this software will develop in comparison to our toolbox.

The philosophy behind our MNPBEM toolbox is to provide a simulation software with easy installation and transparent coding, which also profits from the Matlab environment enabling simple plotting and data manipulation features. Typical applications for our toolbox use particle boundaries with a few thousand boundary elements, such that the solution of the BEM equations is fast, say of the order of seconds to minutes. Thus, even if one is interested in scattering or extinction spectra, where one has to solve the BEM equations for various light wavelengths, computer time is not an overly critical issue. However, from time to time it becomes necessary to simulate larger nanoparticles, such as nanowires [20], tapered films [21], or realistic nanoparticles exhibiting surface roughness [22], with several 1000 to 10 000 boundary elements, in which cases the direct matrix inversions performed in our MNPBEM toolbox can become painfully slow.

In this paper, we present a novel version of our MNPBEM toolbox aiming at a more efficient simulation of large nanoparticles. To make the simulations faster and less memory consuming, we introduce several novel features: firstly, we use hierarchical matrices [23, 24], or H-matrices in short, for the compression of Green functions; secondly, we employ iterative solvers, such as the conjugate gradient or GMRES ones, to solve the BEM equations. In particular around plasmonic resonances these iterative solvers are found to be extremely slow, and it becomes compulsory to use suitable preconditioners. Here we exploit the possibility that H-matrices can be manipulated similarly to normal matrices to implement an efficient preconditioner based on an approximate solution of the full BEM equations.

These new developments challenge Matlab in the field where it performs best, linear algebra and solution of systems of linear equations. To be compatible and ultimately better than the built-in Matlab functions, the current MNPBEM version uses C++ code embedded in a MEX environment. As a side effect, this code for the implementation of H-matrix operations, which makes excessive use of the BLAS and LAPACK routines, can be also used outside the Matlab environment for a simple and puristic standalone library.

We have organized this paper as follows. In Sec. 2 we discuss how to install the toolbox and give an example for plasmonics simulations with H-matrices and iterative solvers. Sec. 3 gives a brief account of our BEM approach and introduces to hierarchical matrices. Our H-matrix implementation in the toolbox is discussed in Sec. 4. A short summary is given in Sec. 5. To keep the main text short and self-contained, several details about the BEM working equations and H-matrix manipulations are presented in the various appendices.

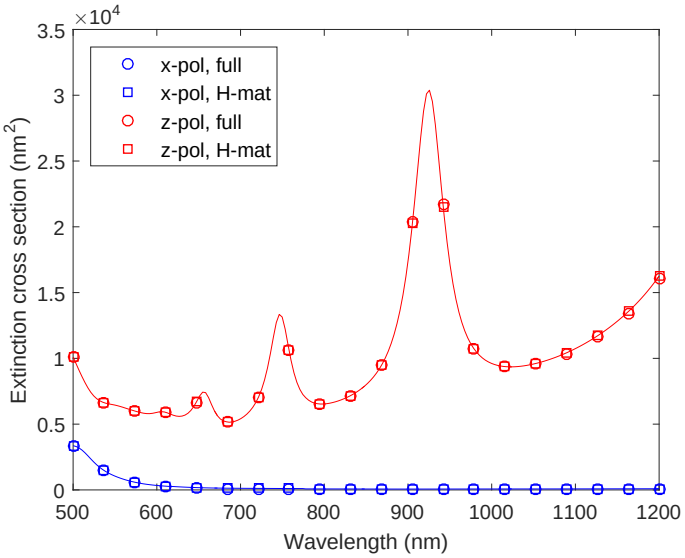


Figure 1: Extinction cross section for gold nanorod produced with `p=trirod(20,800,[15,15,500])` consisting of 7378 boundary elements. The circles report results of a full BEM simulation for x (blue) and z (red) polarization, where z is the long axis of the rod, the squares report results of simulations using H-matrices. The H-matrix simulations are about a factor of three faster and use significantly less memory.

2. Getting started

2.1. Installation of the toolbox

To install the toolbox, one must simply add the path of the main directory `mnpbemdir` of the `MNPBEM` toolbox as well as the paths of all subdirectories to the Matlab search path. This can be done, for instance, through

```
addpath(genpath(mnpbemdir));
```

To set up the help pages, one must once change to the main directory of the MNPBEM toolbox and run the program `makemnpbemhelp`

```
>> cd mnpbemdir;
>> makemnpbemhelp;
```

Once this is done, the help pages, which provide detailed information about the toolbox, are available on the start page of the help browser under *Supplemental Software*. The toolbox is similar to our previously published version [16].

2.2. A simple example

Details about BEM simulations with our toolbox can be found in Refs. [14–16] and in the help pages. In the following we discuss the example of light scattering at a nanorod.

```
op = bemoptions( 'sim', 'ret' );                    %  options for BEM simulation
epstab = { epsconst( 1 ), epstable( 'gold.dat' ) };    %  table of dielectric functions
%  nanorod with diameter 20 nm, length 800 nm, number of boundary elements 7378
p = trirod( 20, 800, [ 15, 15, 500 ] );
p = comparticle( epstab, { p }, [ 2, 1 ], 1, op );     %  initialize nanorod
```

Table 1: Demo programs for simulations with iterative BEM solvers provided by the MNPBEM toolbox. We list the names of the programs, typical runtimes, and give brief explanations. `stat` refers to demo files using the quasistatic approximation, and `ret` to simulations of the full Maxwell equations. The programs were tested on a standard PC (Intel i7–2600 CPU, 3.40 GHz, 16 GB RAM).

Demo program	Runtime	Description
`demospecstat17.m`	4 min	Light scattering of nanorod using iterative BEM solver
`demospecstat18.m`	30 sec	Auxiliary information for iterative BEM solver
`demospecret17.m`	35 min	Light scattering of nanorod using iterative BEM solver
`demospecret18.m`	70 min	Timing for iterative BEM solver
`demospecret19.m`	19 min	Light scattering of sphere chain using iterative BEM solver
`demospecret20.m`	2 h	Nanorod above substrate using itertive BEM solver
`demoeelsret9.m`	60 min	EEL spectra for coupled nanorods using iterative BEM solver

We first set up an options structure, which selects a BEM solver for the simulation of the full Maxwell's equations, and define a table of dielectric functions, here $\varepsilon_b = 1$ for the embedding medium and a gold dielectric function using values tabulated from optical experiments [25]. Next, we define a nanorod with a diameter of 20 nm and a length of 800 nm, which can be plotted with `plot(p,'EdgeColor','b')`, see Fig. 2. We then compute the extinction cross section

```
bem = bemsolver( p, op );                                          %  set up BEM solver
exc = planewave( [ 1, 0, 0; 0, 0, 1 ], [ 0, 0, 1; 1, 0, 0 ], op );  %  plane wave excitation
enei = linspace( 500, 1200, 20 );                    %  light wavelengths in vacuum
ext = zeros( length( enei ), 2 );                    %  allocate extinction cross section

for ien = 1 : length( enei )                         %  loop over wavelengths
  sig = bem \ exc( p, enei( ien ) );                 %  surface charge
  ext( ien, : ) = exc.ext( sig );                    %  extinction cross section
end
```

First, we set up a BEM solver for the solution of the full Maxwell's equations. We next define a plane wave excitation with light polarization along x or z (the long axis of the rod), and orthogonal light propagation directions, define the light wavelengths `enei`, allocate an array for the extinction cross sections `ext`, and then compute within a loop over the different wavelengths the surface charges `sig` and cross sections `ext`. More details about the different objects and how to control them can be found in our previous papers [14–16] and in the help pages of the toolbox. Fig. 1 shows the extinction cross sections computed for the nanorod.

For the particle boundary consisiting of 7378 boundary elements the above simulation takes about 90 minutes. To save computer time and memory, in the new version of the toolbox one can switch to iterative BEM solvers which internally use hierarchical matrices. To this end, we simply have to modify the options structure according to

```
op = bemoptions( 'sim', 'ret' );     %  options for BEM simulation
op.iter = bemiter.options;           %  use iterative BEM solver
```

One can set some option parameters in the `bemiter.options` call, as we will discuss further below. With the iterative BEM solver the simulation takes about 36 minutes, which is roughly a factor of three faster than with the conventional solver. Also the memory used by the program is significantly reduced, say by a factor between 5 and 10. In Fig. 1 we also show the extinction cross sections computed with the iterative solver. As can be seen, the spectra of the full and iterative solvers are almost indistinguishable. Quite generally, we expect that the performance of the iterative solver will further improve with increasing number of boundary elements. In the toolbox we provide a number of demo files listed in Table 1.

2.3. Compiling the MEX files

The toolbox comes along with a C++ library for the manipulation of H-matrices and a number of MEX files. We distribute precompiled versions for the most common operating systems. However, in some cases

it might be needed to recompile the files, or one might like to improve the performance of the MEX files by using different compilers. To this end, we provide in the directory `MNPBEM/mex` the file `makemex.m` which must be executed

```
>> makemex
```

to create the MEX files. Experienced users might like trying to improve the performance of the H-matrix library by selecting different compilers and/or modifying the compilation options.

3. Theory

3.1. The BEM working equations

The methodology of our BEM approach has been developed by Javier García de Abajo and Archie Howie [7]. In the following we briefly summarize the central steps underlying this approach. The systems we have in mind consist of a single or several bodies V_j with homogeneous permittivities ε_j, which are embedded in a dielectric background ε_b. A generalization for a background consisting of stratified media has been presented in Ref. [16]. The central quantities of our approach are the scalar potential $\phi_j(\boldsymbol{r})$ and vector potential $\boldsymbol{A}_j(\boldsymbol{r})$, which are related to the electromagnetic fields according to

$$\boldsymbol{E}_j = ik\boldsymbol{A}_j - \nabla\phi_j\,, \quad \boldsymbol{B}_j = \nabla\times\boldsymbol{A}_j\,. \tag{1}$$

Note that we work in the frequency domain and use Gauss units. Here $k=\omega/c$ and c are the wavenumber and speed of light in vacuum, respectively. The potentials are connected through the Lorenz gauge condition $\nabla\cdot\boldsymbol{A}_j = ik\varepsilon_j\phi_j$. Throughout we set the magnetic permeability $\mu_j = 1$. Within each medium, we introduce the Green function for the Helmholtz equation defined through

$$\left(\nabla^2 + k_j^2\right) G_j(\boldsymbol{r},\boldsymbol{r}') = -4\pi\delta(\boldsymbol{r}-\boldsymbol{r}')\,, \quad G_j(\boldsymbol{r},\boldsymbol{r}') = \frac{e^{ik_j|\boldsymbol{r}-\boldsymbol{r}'|}}{|\boldsymbol{r}-\boldsymbol{r}'|}\,, \tag{2}$$

where $k_j = \sqrt{\varepsilon_j}k$ is the wavenumber in the medium $\boldsymbol{r}\in V_j$. For an inhomogeneous dielectric environment, the solutions of Maxwell's equations can be written in the *ad-hoc* form [7, 10]

$$\phi_j(\boldsymbol{r}) = \phi_j^e(\boldsymbol{r}) + \oint_{\partial V_j} G_j(\boldsymbol{r},\boldsymbol{s})\sigma_j(\boldsymbol{s})\,da \tag{3a}$$

$$\boldsymbol{A}_j(\boldsymbol{r}) = \boldsymbol{A}_j^e(\boldsymbol{r}) + \oint_{\partial V_j} G_j(\boldsymbol{r},\boldsymbol{s})\boldsymbol{h}_j(\boldsymbol{s})\,da\,. \tag{3b}$$

Here ϕ_j^e and $\boldsymbol{A}_j^e$ are the scalar and vector potentials characterizing the external perturbation (e.g., plane wave or oscillating dipole) within a given medium j. Because of Eq. (2), these expressions fulfill the Helmholtz equations everywhere except at the particle boundaries. σ_j and $\boldsymbol{h}_j$ are surface charge and current distributions, which are chosen such that the boundary conditions of Maxwell's equations at the interfaces between regions of different permittivities ε_j hold.

We next introduce in accordance to Refs. [7, 12, 14] matrix notations of the form $G\sigma$ instead of the integration given in Eq. (3a). This also allows us to immediately change to a boundary element method (BEM) approach, where the boundary is split into elements of finite size suitable for a numerical implementation. With σ_1 and $\boldsymbol{h}_1$ denoting the surface charges and currents at the particle *insides*, and σ_2 and $\boldsymbol{h}_2$ the corresponding quantities at the particle *outsides*, we obtain from the continuity of the scalar and vector potentials at the particle boundaries the expressions

$$G_1\sigma_1 - G_2\sigma_2 = \phi_2^e - \phi_1^e\,, \quad G_1\boldsymbol{h}_1 - G_2\boldsymbol{h}_2 = \boldsymbol{A}_2^e - \boldsymbol{A}_1^e\,. \tag{4}$$

Note that this approach can be also generalized to the systems consisting of multiple dielectric materials [7]. From the continuity of the Lorentz gauge condition and the dielectric displacement at the particle boundary, we get

$$\varepsilon_1 H_1\sigma_1 - \varepsilon_2 H_2\sigma_2 - ik\hat{\boldsymbol{n}}\cdot(\varepsilon_1 G_1\boldsymbol{h}_1 - \varepsilon_2 G_2\boldsymbol{h}_2) = D^e\,, \quad D^e = \hat{\boldsymbol{n}}\cdot[\varepsilon_1(ik\boldsymbol{A}_1^e - \nabla\phi_1^e) - \varepsilon_2(ik\boldsymbol{A}_2^e - \nabla\phi_2^e)] \tag{5a}$$

$$H_1\boldsymbol{h}_1 - H_2\boldsymbol{h}_2 - ik\hat{\boldsymbol{n}}(\varepsilon_1 G_1\sigma_1 - \varepsilon_2 G_2\sigma_2) = \boldsymbol{\alpha}\,, \quad \boldsymbol{\alpha} = (\hat{\boldsymbol{n}}\cdot\nabla)(\boldsymbol{A}_2^e - \boldsymbol{A}_1^e) + ik\hat{\boldsymbol{n}}(\varepsilon_1\phi_1^e - \varepsilon_2\phi_2^e)\,. \tag{5b}$$

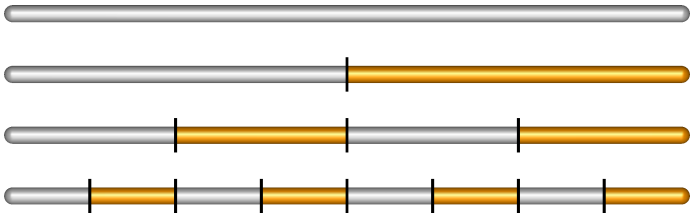


Figure 2: Schematics for creation of cluster tree. We start with a nanorod consisting of 7378 boundary elements, which we create in the MNPBEM toolbox through `trirod(20,800,[15,15,500])`, and partition the boundary elements in a cluster tree through bisection, `tree=clustertree(p,'cleaf',200)`. Starting with the parent cluster consisting of all boundary elements, we place a bounding box around the nanoparticle and split the box along the largest dimension: elements in the first (second) half are assigned to the first (second) cluster son. Splitting through bisection is repeated for the cluster sons, until we end up with a *leaf*, which is a cluster consisting of less than C_{leaf} boundary elements.

where $\hat{\boldsymbol{n}}$ is the outer surface normal of the boundary ∂V, and we have introduced the surface derivatives of the Green function $H_{1,2} = (\hat{\boldsymbol{n}} \cdot \nabla) G_{1,2} \pm 2\pi$. Eqs. (4) and (5) form a set of eight coupled equations that can be solved within a boundary element method (BEM) approach in order to obtain the surface charges and currents, which provide the unique solution for a given external excitation [7, 10, 12]. The central steps of this solution are briefly sketched in Appendix A.

3.2. Iterative solution of BEM equations

In case of large particle boundaries, consisting of several thousand to ten thousand boundary elements, it becomes difficult to store the matrices G_1, G_2, H_1, and H_2, and also the multiplication and inversion of matrices (see Appendix A) becomes computationally costly. H-matrices, to be discussed further below, provide an efficient means for the compression of the Green function matrices, controlled through some accuracy parameter η, and allow for fast matrix-vector multiplications.

This suggests solving Eqs. (4,5) with an iterative solver for the set of linear equations, such as the conjugate gradient or generalized minimal residual (GMRES) methods [26]. The set of equations can be interpreted as a matrix equation $A\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{x}$ is the solution vector containing the surface charge and current distributions σ and $\boldsymbol{h}$, and $\boldsymbol{b}$ is the inhomogeneity of the external electromagnetic potentials. The matrix equation is solved by starting with some initial guess for $\boldsymbol{x}^{(0)}$, and using the residuum $\boldsymbol{r}^{(i)} = \boldsymbol{b} - A\boldsymbol{x}^{(i)}$ to determine an improved guess for $\boldsymbol{x}^{(i+1)}$. The iterative solution comes to an end when $\|\boldsymbol{r}^{(i+1)}\|$ drops below a user-defined tolerance.

Unfortunately, for the matrix A defined through Eqs. (4,5) it is not guaranteed that the iterative solution converges, and indeed one finds that convergence of the BEM equations can be extremely slow, in particular for frequencies close to plasmonic resonances. It is well known that the convergence behavior can be drastically improved by using a *preconditioner* matrix M, where one solves the equation $M^{-1}A\boldsymbol{x} = M^{-1}\boldsymbol{b}$ that possesses the same solution vector $\boldsymbol{x}$. If M^{-1} is close to the inverse of A, convergence of the iterative solver can be extremely fast.

In our approach we exploit for the preconditioner matrix M a second feature of H-matrices, namely that they can be manipulated similarly to normal matrices, with the only difference that the manipulations (such as matrix multiplications and inversions) are significantly faster but bound by some accuracy η [23, 24]. In general, H-matrix manipulations become faster with decreasing accuracy. This suggests the following strategy for the iterative solution of the BEM equations: in the iterative solution of the BEM equations we introduce two tolerance parameters, η_1 and η_2. η_1 controls the compression of the Green function matrices G_1, G_2, H_1, and H_2, and is chosen low (to achieve high accuracy of the final solution). η_2 controls the accuracy of the auxilary matrices Σ_1, Σ_2, Δ^{-1} and Σ^{-1}, see Appendix A, which are used as a preconditioner for the approximate solution of the BEM equations, and is chosen sufficiently large (to achieve fast matrix multiplications and inversions). In combination, the accuracy of the iterative solver can be high for a small η_1 value (as well as a small tolerance for the termination of the iterative solver), whereas the evaluation of the preconditioner $M^{-1}\boldsymbol{x}^{(i)}$ can be fast (but relatively inaccurate) for a sufficiently large η_2 value. However, even with a relatively crude approximation for the preconditioner typically only a very small number of

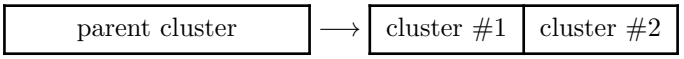
iterations, say ten, is needed for the computation of the surface charges and currents. Below we will discuss the choice of the different parameters and other implementation technicalities in more detail.

3.3. Hierarchical matrices

Hierarchical matrices, or H-matrices in short, have been described in some detail elsewhere [23, 24]. In the following we provide a brief introduction to the topic, mainly to set the stage for discussing our H-matrix implementation. Some details about our C++ library are provided in Appendix C.

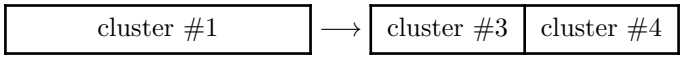
3.3.1. Cluster tree and admissibility

We consider a discretized particle boundary consisting of boundary elements of finite size, as schematically shown in Fig. 2. In a first step, we partition the boundary into a cluster tree. We start with the *parent* cluster consisting of all boundary elements, and place a bounding box around the particle. In the next step, we split the bounding box along the largest dimension: the boundary elements in the first half belong to the first cluster, and the elements in the second half to the second cluster,



.

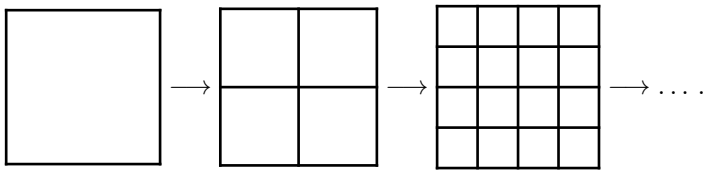
We then submit the newly created clusters to an analogous bisection procedure,



and similarly for the second cluster, thereby setting up a so-called *cluster tree*. The splitting comes to an end when the number of boundary elements in a given cluster drops below a given threshold value C_{leaf}, typically of the order of hundred. In the cluster tree each cluster has either two sons, created through bisection, or is a leaf. The algorithm for setting up the cluster tree starts with an empty cluster tree T and the cluster index τ set to the parent cluster, which contains all boundary elements. We then call the procedure:

procedure BISECTION(var T, τ) ▷ create cluster tree through bisection
  $T \leftarrow T \cup \tau$ ▷ add cluster τ to cluster tree T
  if $|\tau| > C_{\text{leaf}}$ **then** ▷ number of elements of cluster τ larger than C_{leaf}
    split cluster τ through bisection into sons τ_1 and τ_2
    BISECTION(T, τ_1) ▷ continue with bisection of son τ_1
    BISECTION(T, τ_2) ▷ continue with bisection of son τ_2
  end if
end procedure

The basic philosophy behind H-matrices is that for clusters sufficiently far apart we don't have to compute the Green function elements for all boundary elements exactly, but can submit the interaction matrix to some sort of approximation, such as a multipole expansion [27] or a low-rank approximation, as we will do. First, we notice that also the interaction matrices G_1, G_2, H_1, and H_2 of our BEM approach become partitioned when submitting the clusters to bisection splitting,



Subdividing the matrix by moving down the cluster tree we obtain a *hierarchical* matrix composed of many *submatrices*. As we will discuss below, through this checkerboard structure it is possible to perform H-matrix operations such as multiplication or inversion similarly to normal matrices, with the only difference that instead of matrix elements we have to deal with submatrices.

In the next step we have to determine when the interaction matrix between two clusters τ_1 and τ_2 can be submitted to a low-rank approximation. We shall call this the *admissibility* criterion. Let $\text{rad}(\tau)$ be the

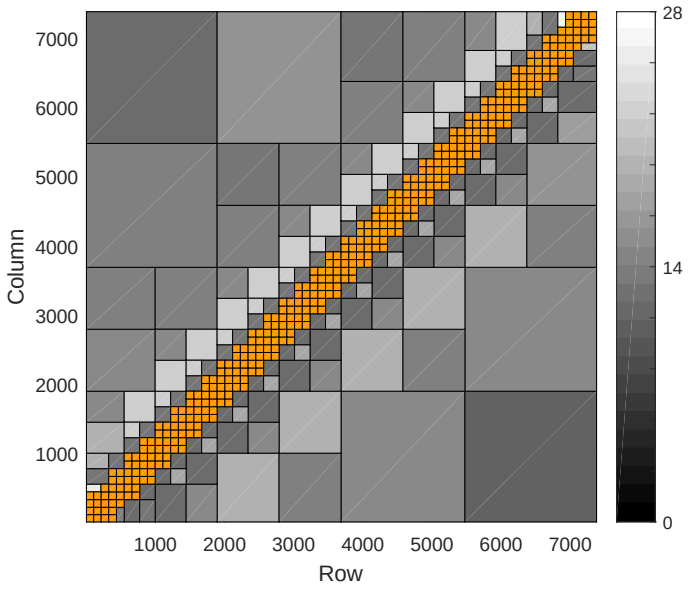
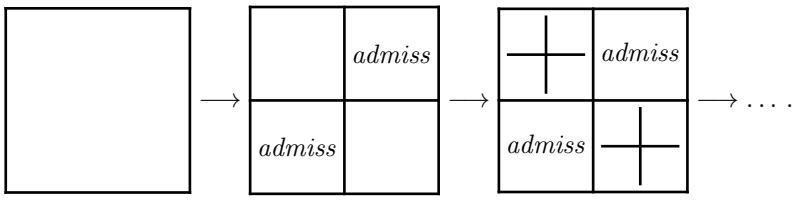


Figure 3: Rank of submatrices. The H-matrix is partitioned into admissible submatrices (gray), which can be approximated by low-rank matrices, and full matrices (orange) along the diagonal of the hierarchical matrix. In the figure we show the Green function G_2 for the gold nanorod shown in Fig. 2 using a light wavelength of 600 nm. The gray level corresponds to the rank needed for a truncation error of 10^{-6}. The compression factor for the matrix is 0.12.

radius of a sphere enclosing cluster τ, and $\mathrm{dist}(\tau_1, \tau_2)$ the distance between the center positions of clusters τ_1 and τ_2. A typical choice for the admissibility criterion would be [24]

$$\min\{\mathrm{rad}(\tau_1), \mathrm{rad}(\tau_2)\} \leq \lambda\, \mathrm{dist}(\tau_1, \tau_2)\,, \tag{6}$$

where λ is a parameter controlling the trade-off between the number of admissible blocks and the accuracy of the approximation. To split the matrix into admissible submatrices and submatrices that need to be further subdivided, we move down the cluster tree and test after each subdivision for admissibility of the different cluster pairs,



If they are admissible, no furher subdivision is done. Otherwise we continue with subdivision until we either find admissibility or end up with two clusters that are leaves of the cluster tree. In the latter case, we make no approximation for the interaction matrix and treat the submatrix as a full matrix. Through this procedure we partition the entire matrix into a hierarchical matrix consisting of submatrices. Figure 3 shows the rank structure for the Green function matrix computed for the nanorod of Fig. 2.

To set up a block tree for admissibility, we start with an empty admissibilty matrix admiss and set the cluster indices τ and σ to the parent cluster. We then call the following procedure:

procedure ADMISSIBILITY(var admiss, τ, σ) ▷ determine admissibility of submatrices
  if $|\tau| < C_{\mathrm{leaf}}$ and $|\sigma| < C_{\mathrm{leaf}}$ **then**
    admiss($\tau \times \sigma$) $\leftarrow$ full matrix ▷ use full matrix for cluster pair $\tau \times \sigma$
  else if $\tau \times \sigma$ is admissible **then**

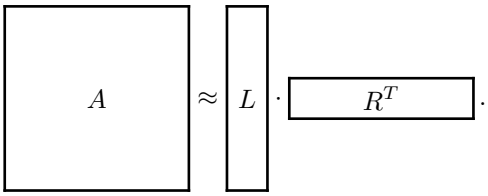
```
        admiss(τ × σ) ← low-rank matrix        ▷ use low-rank approximation for cluster pair τ × σ
    else                                        ▷ subdivide clusters
        for τ′, σ′ ← sons of τ, σ do
            ADMISSIBILITY(admiss, τ′, σ′)
        end for
    end if
end procedure
```

3.4. Low-rank approximation

To achieve memory compression and faster manipulations with hierarchical matrices, it is necessary to perform some kind of approximation for the admissible submatrices. Let $A_{m\times n}$ be a submatrix of dimension $m\times n$ which we assume to be admissible. In the H-matrix approach we approximate the matrix by a product of two low-rank matrices

$$A_{m\times n} \approx L_{m\times k} R^T_{k\times n}\,, \tag{7}$$

where T denotes the transpose of the matrix. Obviously, if k is significantly smaller than m and n one can drastically reduce the number of stored matrix elements, as can be inferred from the graphical matrix representation



The compression factor, defined as the number of stored matrix elements $mk+nk$ divided by the number of matrix elements mn, depends on the details of the matrix A. In case of the Green function matrices of the BEM approach one can achieve compression factors of the order of 0.1 or less. High compressions are also accompanied by faster matrix manipulations, owing to the reduced number of floating point operations.

As a first example for achieving matrix compression we mention the singular value decomposition, which factorizes the matrix into

$$A_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^\dagger_{n\times n}\,, \tag{8}$$

where $U_{m\times m}$ is a unitary matrix, $\Sigma_{m\times n}$ is a diagonal matrix with non-negative real numbers on the diagonal, and $V^\dagger_{n\times n}$ is a unitary matrix. In many cases of interest we do not have to consider all diagonal entries of Σ but can keep only values larger than a given cutoff value, say 10^{-6} times the largest element. The matrix A can then be approximated by the low-rank version

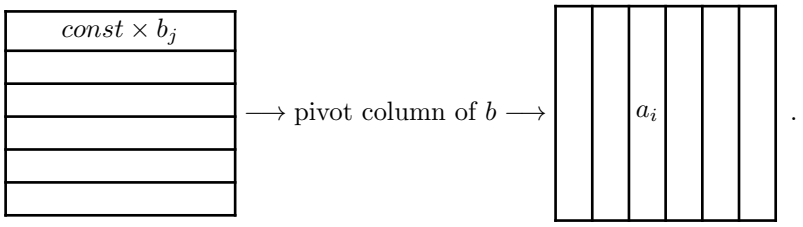
$$M_{m\times n} \approx U_{m\times k}\Sigma_{k\times k}V^\dagger_{k\times n}\,, \tag{9}$$

where k is the number of singular values kept that is usually much smaller than m and n. Setting $L = U\Sigma$ and $R^T = V^\dagger$ we have computed the low-rank approximation of Eq. (7).

3.5. Adaptive cross approximation

Although the singular value decomposition is an extremely powerful algorithm, it has two shortcomings. Firstly, one has to compute all matrix elements of A prior to submitting the matrix to the decomposition. Secondly, it is very slow. A way out is an alternative algorithm that is called *adaptive cross approximation* [28]. It has the advantages that only few matrix elements have to be computed, and the low-rank approximation of A is set up with with very little additional cost. We start by computing the matrix elements of

the first row of the matrix $b_j = A_{1j}$,



Next, we determine the pivot column c where the modulus of b_j is largest, divide b by the pivot element, and compute the c'th column $a_i = A_{ic}$. Once this is done, we set $L_{i1} = a_i$ and $R_{j1} = b_j$ and obtain the lowest-order approximation $A \approx LR^T$. In the next iteration, we subtract the lowest-order approximation from A, to obtain the residuum matrix $A - LR^T$, determine the pivot row r of a_i, and continue with the above scheme until the product norm $|a|\,|b|$ of the newly computed vectors drops below a given threshold value. The adaptive cross approximation algorithm can be summarized as follows:

procedure ACA(var L, var R, A) ▷ adaptive cross approximation for $A \approx LR^T$
  $r \leftarrow 1$ ▷ initial guess for pivot row
  $k \leftarrow 1$ ▷ rank of matrices L and R
  repeat
    $b_j \leftarrow A_{rj}$ ▷ compute row r of matrix A
    $b_j \leftarrow \sum_{\ell=1}^{k-1} L_{r\ell} R_{\ell j}^T$ ▷ b_j gets values of residuum matrix $A - LR^T$
    $c \leftarrow$ column index where $|b_j|$ is largest, $b_{\text{piv}} \leftarrow b_c$
    $b_j \leftarrow b_j / b_{\text{piv}}$
    $a_i \leftarrow A_{ic}$ ▷ compute column c of matrix A
    $a_i \leftarrow \sum_{\ell=1}^{k-1} L_{i\ell} R_{\ell c}^T$ ▷ a_j gets values of residuum matrix $A - LR^T$
    $L_{ik} \leftarrow a_i$, $R_{jk} = b_j$ ▷ update low-rank matrices
    $r \leftarrow$ row index where $|a_i|$ is largest ▷ determine new pivot row for next iteration
      assert that row r has not been searched before
      by discarding previous r values from the i list
    $k \leftarrow k + 1$ ▷ increment iteration counter
  until $|a|\,|b| >$ `htol` and $k <$ `kmax`
end procedure

The accuracy of the above scheme is determined by the tolerance `htol` assuming that the maximal rank `kmax` is chosen sufficiently large (see discussion below). In general, the adaptive cross approximation converges extremely fast. The algorithm may fail for a block-diagonal form of A, in which case only the first block will be approximated by LR^T. This could be avoided by computing within each iteration a few random elements of the matrix, which become pivot elements if they are larger than the pivot elements of a and b. However, for all problems we have investigated so far no such difficulties were encountered and we have thus refrained from this approach.

3.6. H-matrix operations

Because of the checkerboard structure of H-matrices it is possible to perform matrix operations similarly to normal matrices. A few examples, including summation, multiplication, and inversion are sketched in Appendix B. See also Appendix C for a brief discussion of our C++ implementation. Importantly, the operations can be performed significantly faster than for normal matrices owing to the matrix compression using low-rank matrices. On the other hand, the accuracy of the matrix operations is bound by the parameters `htol` and `kmax` used in the adaptive cross approximation.

4. Iterative solvers and H-matrices in the MNPBEM toolbox

4.1. The options structure

In principle, all details about the H-matrices and iterative solvers are hidden in the Matlab classes and the user usually only has to deal with the options structure set through

```
op.iter = bemiter.options;
op.iter = bemiter.options( PropertyName, PropertyValue );
```

The structure contains the following fields:

```
>> bemiter.options

     solver: 'gmres'
        tol: 1.0000e-06
      maxit: 100
    restart: []
    precond: 'hmat'
     output: 0
      cleaf: 200
       htol: 1.0000e-06
       kmax: [4 100]
    fadmiss: @(rad1,rad2,dist)2.5*min(rad1,rad2)<dist
```

These fields, which can be set in the `bemiter.options` through property pairs, control the simulation performance as follows:

- `solver` selects among the iterative solvers `cgs` (conjugate gradient), `bicgstab` (biconjugate gradients stabilized method), and `gmres` (generalized minimum residual method). For details see the Matlab help pages.
- `tol`, `maxit`, `restart` control the performance of the iterative solvers, see Matlab help pages.
- `precond` selects the preconditioner for the iterative solution of the BEM equations. Currently only `'hmat'` is implemented.
- `output` controls the output during the BEM solution, the default value 0 gives no output, 1 gives some intermediate output.
- `cleaf` is the minimum number of boundary elements for the cluster tree. Values should be typically in the range between 100 and 400.
- `htol` and `kmax` control the performance of the matrix compression using the adaptive cross approximation. In our implementation both variables can be arrays with two elements. The values

  ```
  htol1 = min( htol );  kmax1 = max( kmax );
  ```

  are used for the comression of the Green function matrices G_1, G_2, H_1, and H_2 and control (together with `tol`) the overall accuracy of our iterative BEM solution. The values

  ```
  htol2 = max( htol );  kmax2 = min( kmax );
  ```

  control the accuracy of the H-matrix manipulations used for the approximate solution of the BEM working equations, see Appendix A, by the preconditioner. With the standard setting `kmax2=4` we get H-matrices with a maximal rank of four, resulting in a not overly accurate but sufficiently fast preconditioner.
- `fadmiss` is a function handle controlling the admissibility criterion for low-rank approximations.

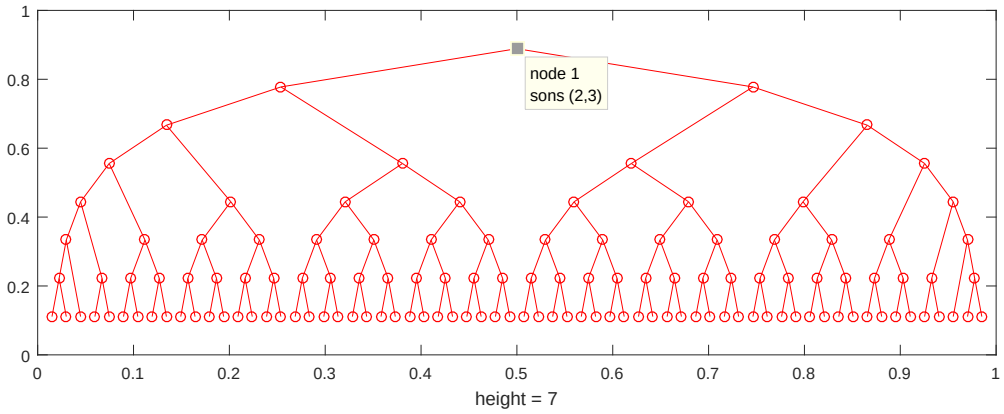


Figure 4: Cluster tree for nanorod shown in Fig. 2. The figure was produced in the MNPBEM toolbox with `plot(tree)`, the node and cluster sons can be accessed with the data cursor. Visualization of the different clusters, similar to Fig. 2, can be achieved via `plotcluster(tree)`.

4.2. Cluster tree

In most cases of interest it suffices to use the default values for the options and to pass them to the BEM solvers. In what follows, we describe some commands that allow inspecting the cluster tree and the performance of the iterative solver in more detail. We emphasize that in general these commands are only used internally by the BEM solvers. Setting up the cluster tree is achieved through

```
%  table of dielectric functions
epstab = { epsconst( 1 ), epstable( 'gold.dat' ) };
%  initialize nanorod
p = trirod( 20, 800, [ 15, 15, 500 ] );
p = comparticle( epstab, { p }, [ 2, 1 ], 1, bemoptions );
%  compute cluster tree
tree = clustertree( p, 'cleaf', 200 );
```

In the last line we partition the boundary elements into a cluster tree

```
>> tree

clustertree :
      p: [1x1 comparticle]
    son: [131x2 double]
```

The `clustertree` object contains the `comparticle` object defining the particle boundaries and the details of the dielectric environment (see help pages of the toolbox for a more detailed discussion) and the tree structure of the sons. The cluster tree can be visualized with the plot commands

```
%  plot cluster tree
plot( tree );
%  plot clusters
plotcluster( tree );
```

The first `plot` command generates the figure shown in Fig. 4. The parent node has two sons, which are again split into sons through bisection and so on, until the number of boundary elements in a cluster drops below a given threshold value `cleaf` and one ends up with a *leaf*. One can use the data cursor of the Matlab figure panel to explore the cluster and son indices, as shown in the figure for the parent cluster. The `plotcluster` command plots the different clusters similar to Fig. 2.

The *clustertree* class has a number of methods and properties. Most importantly, within the cluster tree the boundary elements are ordered differently than in the `comparticle` object, namely in ascending order for the different clusters. With the commands

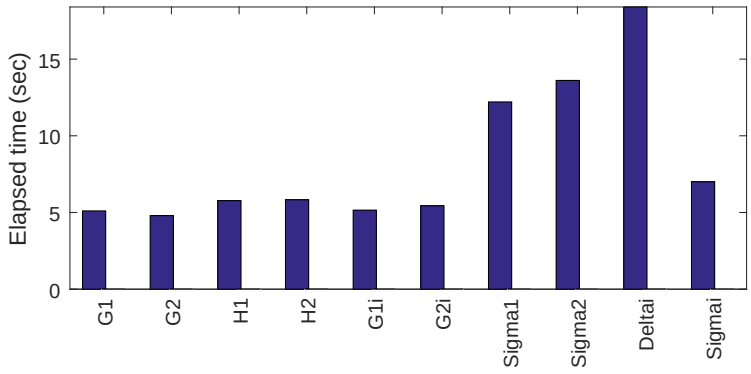


Figure 5: Output of iterative BEM solver during initialization `bem(800)` for gold nanorod shown in Fig. 2. It reports the runtimes for setting of the H-matrices G_1, G_2, H_1, and H_2, and for the evaluation of the auxiliary matrices introduced in Appendix A.

```
%  convert between particle and cluster indices
ind1 = part2cluster( tree, reshape( 1 : p.n, [], 1 ) );
%  convert between cluster and particle indices
ind2 = cluster2part( tree, ind1 );
```

it is possible to switch between normal `comparticle` and cluster ordering. The variable

```
%  index range for clusters [ncluster,2]
tree.cind
```

contains the size for a given cluster `ic`: `tree.cind(ic,1)` points to the first element of a cluster, `tree.cind(ic,2)` to the last element of a cluster. For instance, the indices for the parent cluster 1 and its sons 2, 3 can be printed through

```
>> tree.cind(1:3,:)

           1        7378
           1        3682
        3683        7378
```

As can be seen, the parent cluster contains all boundary elements, while its two sons contain the first and second half of the cluster-ordered boundary elements.

4.3. H-matrices and BEM solver

The initialization of H-matrices and their manipulation is done in the BEM solver classes which are selected according to the options setting. Let us consider for instance the following code with a run time of about two minutes

```
%  options for BEM simulation
op = bemoptions( 'sim', 'ret' );
op.iter = bemiter.options( 'output', 1 );
%  set up BEM solver and perform initialization
bem = bemsolver( p, op );
bem = bem( 800 );
```

The `output` flag opens a window, see Fig. 5, that shows the computer run times for setting up and manipulating the H-matrices. As a side remark, internally we do not invert the G_1 and G_2 matrices directly, but use a faster LU-decomposition instead. For this reason, the H-matrix manipulations are significantly faster than those for Σ_1, Σ_2, and Δ^{-1} which invoke multiple matrix multiplications and inversions.

To inspect the BEM solver object we type at the Matlab prompt

```
>> bem

bemretiter :
         p: [1x1 comparticle]
         g: [1x1 aca.compgreenret]
    solver: 'gmres'
       tol: 1.0000e-06
```

As can be seen, the `bemretiter` class has its own Green function object `aca.compgreenret` that initializes the Green functions using the adaptive cross approximation. The Green functions can be inspected through

```
>> bem.G2

hmatrix :
    tree: [1x1 clustertree]
    htol: 1.0000e-06
    kmax: 100
     val: {386x1 cell}
     lhs: {160x1 cell}
     rhs: {160x1 cell}
```

`htol` and `kmax` are the tolerance and the maximal rank used for the Green function. `val` stores the full matrices, and `lhs` and `rhs` are the low-rank matrices. With `plotrank(bem.G2)` we can plot the rank of the low-rank matrices, as shown in the left panel of Fig. 6 (see also Fig. 3). Note that here the full matrices along the diagonal of the matrix are plotted with a rank of zero. The preconditoner matrices can be inspected through

```
>> bem.sav

         k: 0.0079
      nvec: [7378x3 double]
       G1i: [1x1 hmatrix]
       G2i: [1x1 hmatrix]
      eps1: [7378x7378 double]
      eps2: [7378x7378 double]
    Sigma1: [1x1 hmatrix]
    Deltai: [1x1 hmatrix]
    Sigmai: [1x1 hmatrix]
```

The right panel of Fig. 6 shows the inverse of the G_2 matrix, which is computed through a LU-decomposition of the H-matrix. Through `kmax=[4,100]` we select for the H-matrix manipulations the lower rank of 4. As can be seen, all low-rank matrices have the same rank of 4, and the H-matrix inversion thus has an uncontrolled truncation error. However, as discussed in some detail above this is not a problem because these auxiliary matrices are only used for the preconditioner, and the accuracy of the entire scheme is governed by the larger value of `kmax` and the smaller value of `htol`.

4.4. Iterative solver

For a BEM simulation we have to specify some external excitation, such as a plane wave with z-polarization propagating along the x-direction

```
%  plane wave excitation with z-polarization
exc = planewave( [0,0,1], [1,0,0], op );
%  iterative BEM solution with timing
tic; sig = bem \ exc( p, 800 ); toc

gmres(100), it=  8(1), res= 6.429e-07, flag=0
Elapsed time is 14.421547 seconds.
```

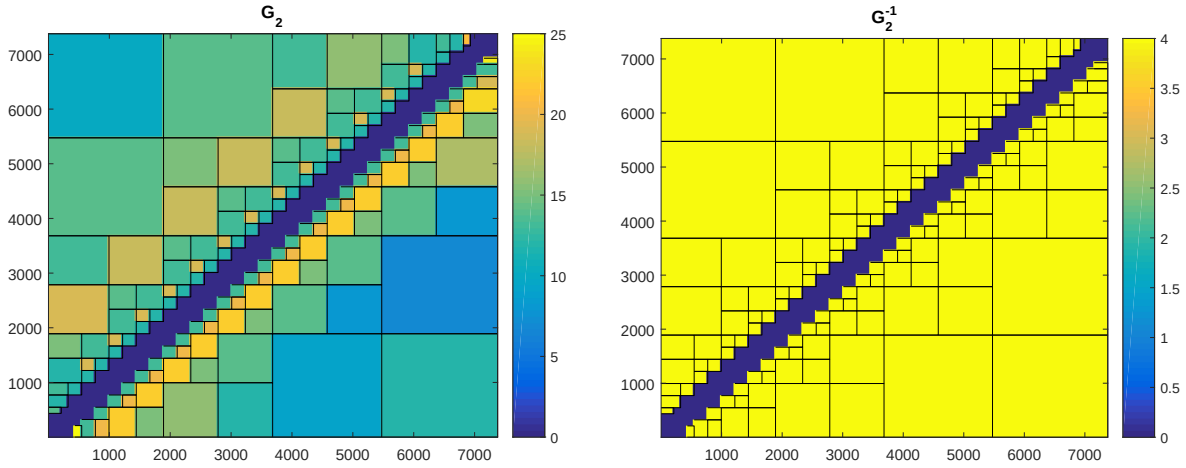


Figure 6: Rank of H-matrices as produced with the commands `plotrank(bem.G2)` (left) and `plotrank(bem.sav.G2i)` (right). The Green functions are computed with a sufficiently large maximal rank, whereas the matrices for the preconditioner are computed with a small `kmax` of four.

As can be seen, only 8 iterations are needed by the iterative solver to reach an accuracy of less than 10^{-6}. In general, the time needed for the iterative solution is much less than the time needed for initializing the Green functions and the preconditioner. Finally, some statistics about the performance of the iterative BEM solver can be obtained by typing

```
>> hinfo(bem)

Compression Green functions        :  0.141079
Compression auxiliary matrices     :  0.098572

Total time for H-matrix operations :  58.269    sec
  aca           :  18.26 %
  add           :   1.87 %
  dcopy         :  15.57 %
  lu            :   2.03 %
  mul_BLAS      :  19.27 %
  inv_LAPACK    :   0.07 %
  rest          :  42.94 %
```

The first lines show the compression for the Green functions and the auxiliary preconditioner matrices. We then report the computer times for the different H-matrix operations.

4.5. Scaling behavior

One of the advantages of H-matrices particulary discussed in the mathematical literature [23, 24] is the scaling of the matrix manipulations with respect to the number of boundary elements n: while the number of operations needed for direct matrix multiplication or matrix inversion is of the order n^3, for H-matrices it can scale linearly, $const \times n$. Unfortunately, the prefactor depends on the ranks of the admissible submatrices. For H-matrices of moderate size (which are of relevance for our BEM solvers) this additional factor often spoils the linear scaling. In Fig. 7 we show the time needed for setting up and initializing the BEM solvers for nanorods of increasing length produced with

```
p = trirod( diameter, len, [ 15, 15, n ] );
```

where `len=[1000,1500,2000,2500,3000]` and `n=[500,750,1000,1250,1500]`. As can be seen, the scaling for the BEM solver is certainly better than n^3 but not yet linear. We note that the comparison has to

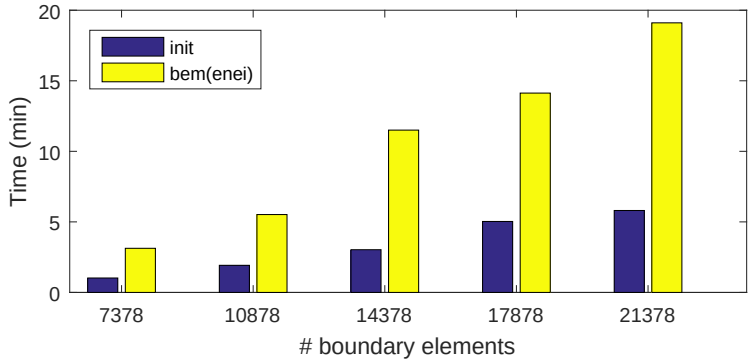


Figure 7: Computer time for BEM solver initializations `bem=bemsolver(p,op)` (blue) and `bem=bem(800)` (yellow) using gold nanorods of increasing size. The rods are created through `p=trirod(50,len,[15,15,n])`, where `len=[1000,1500,2000, 2500,3000]` and `n=[500,750,1000,1250,1500]`.

be considered with some care because in the above example we not only modify the number of boundary elements but at the same time also the geometry of the nanorods.

5. Summary and discussion

To summarize, in this paper we have presented an extension of the MNPBEM toolbox that makes possible simulations of large nanoparticles with several 1000 to 10 000 boundary elements. Our approach builds on hierarchical matrices for matrix compression and fast matrix manipulations, as well as iterative solvers for the solution of the BEM working equations. The strategy behind our approach can be summarized as follows:

- The Green function matrices G_1, G_2, H_1, and H_2 are initialized as H-matrices using the adaptive cross approximation. We use a sufficiently small accuracy parameter η_1 which controls the accuracy of our entire approach.
- The working equations of our BEM approach, see Appendix A, serve as a preconditioner and are solved with a larger accuracy parameter η_2 (iterative BEM simulations with the standard options setting use a small `kmax` value instead). With low accuracy we can perform H-matrix operations sufficiently fast.
- The BEM equations of Eqs. (4,5) are solved using an iterative solver for systems of linear equations, such as conjugate gradients or GMRES. The tolerance for the iterative solver should be set to a value similar to η_1. With this setting, the overall accuracy of our approach is governed by η_1. The approximate BEM solution with accuracy η_2 is used as a preconditioner, which is needed to achieve convergence of the iterative solver in the vicinity of plasmonic resonances. Otherwise the η_2 value has no impact on the accuracy of the BEM solutions.

BEM simulations with iterative solvers can in principle be run like normal simulations, with the only difference that one has to add

```
op.iter = bemiter.options
```

to the options structure. We have discussed that the performance of the iterative BEM solvers can be controlled through a number of additional parameters, although we expect that the default settings will do a reasonable job in most cases of interest and in general no fine-tuning of parameters is needed. In this paper we have mainly discussed the implementation of the above scheme within the MNPBEM toolbox, and have presented only a few representative examples. More examples can be found in the example section of the toolbox help pages. We think that the new toolbox features might be helpful in several respects.

- Iterative BEM solvers using H-matrices should be faster for large nanoparticles. The expected speedup is not dramatic, say a factor between three and ten, which, however, might be a significant improvement for simulations running for several hours.
- Probably more importantly, because of H-matrix compression significantly less memory is needed by the iterative BEM solvers in comparision to the normal ones. Typical compression factors are of the order of 0.1 or less.
- The toolbox comes with its own H-matrix library written in C++ which can be also used outside of Matlab. In the future we might look for more efficient preconditioners, which can be easily implemented with the generic H-matrix library at hand.

Whether the iterative BEM solvers will be of great importance to plasmonics applications has to seen. We are currently using the iterative BEM solvers for several large scale problems where they seem to work significantly faster. In the toolbox we additionally provide iterative BEM solvers for quasistatic simulations and for simulations using stratified media, which have not been discussed in this paper. When using the latter simulations one should carefully check the convergence properties with respect to `htol` and `kmax`. While efficient H-matrix compression has been demonstrated for free-space Green functions, less is known about the reflected Green functions of Ref. [16].

Where do we stand with our toolbox? Contrary to other simulation software packages, such as BEM++ mentioned in the introduction, the MNPBEM toolbox has been developed for applications in the field of plasmonics. Our primary interest is to provide a simulation software that can be applied easily and flexibly to various plasmonics problems. In this respect, we have been happy to see that our toolbox has been accepted extremely well by the plasmonics community (on the web of science our original paper [14] is indicated as a highly cited paper). On the other hand, our approach takes a few non-standard steps, such as the use of electromagnetic potentials instead of fields, or the use of a collocation rather than the more accurate but slower Galerkin scheme. Thus, there is still plenty of room for future improvements. Nevertheless, we hope that our toolbox will continue to serve the plasmonics community as a helpful simulation software.

Acknowledgment

I am indebted to Andi Trügler who over all the years has been the most persistent and critical user of the toolbox. Without him the program would not be what it now is. I gratefully acknowledge most helpful discussions with Gerhard Unger. This work has been supported in part by the Austrian Science Fund FWF under SFB NextLite F4906-N23 and by NAWI Graz.

Appendix A. Deriving the BEM working equations

In this Appendix we show how to derive the BEM working equations. First, we rewrite Eq. (4) in the form

$$G_1\sigma_1 = G_2\sigma_2 + \phi^e\,, \quad G_1\boldsymbol{h}_1 = G_2\boldsymbol{h}_2 + \boldsymbol{A}^e\,, \tag{A.1}$$

where we have introduced the shorthand notations $\phi^e = \phi_2^e - \phi_1^e$ and $\boldsymbol{A}^e = \boldsymbol{A}_2^e - \boldsymbol{A}_1^e$. From Eq. (5a) we obtain

$$\varepsilon_1 H_1 G_1^{-1}\left(G_2\sigma_2 + \phi^e\right) - \varepsilon_2 H_2 G_2^{-1} G_2\sigma_2 - ik\hat{\boldsymbol{n}}\cdot\left\{\varepsilon_1\left(G_2\boldsymbol{h}_2 + \boldsymbol{A}^e\right) - \varepsilon_2 G_2\boldsymbol{h}_2\right\} = D^e$$

through elimination of σ_1 and $\boldsymbol{h}_1$, using Eq. (A.1). Similarly, from Eq. (5b) we obtain

$$H_1 G_1^{-1}\left(G_2\boldsymbol{h}_2 + \boldsymbol{A}^e\right) - H_2 G_2^{-1} G_2\boldsymbol{h}_2 - ik\hat{\boldsymbol{n}}\left\{\varepsilon_1\left(G_2\sigma_2 + \phi^e\right) - \varepsilon_2 G_2\sigma_2\right\} = \boldsymbol{\alpha}\,.$$

We next introduce the auxiliary matrices $\Sigma_1 = H_1 G_1^{-1}$ and $\Sigma_2 = H_2 G_2^{-1}$ and bring the above expressions to the form

$$\begin{aligned}
\left(\varepsilon_1\Sigma_1 - \varepsilon_2\Sigma_2\right) G_2\sigma_2 - ik\hat{\boldsymbol{n}}\cdot\left(\varepsilon_1 - \varepsilon_2\right) G_2\boldsymbol{h}_2 &= D^{e\prime}\,, \quad D^{e\prime} = D^e - \varepsilon_1\Sigma_1\phi^e + ik\hat{\boldsymbol{n}}\cdot\varepsilon_1\boldsymbol{A}^e && \text{(A.2a)}\\
\left(\Sigma_1 - \Sigma_2\right) G_2\boldsymbol{h}_2 - ik\hat{\boldsymbol{n}}\left(\varepsilon_1 - \varepsilon_2\right) G_2\sigma_2 &= \boldsymbol{\alpha}'\,, \quad \boldsymbol{\alpha}' = \boldsymbol{\alpha} - \Sigma_1\boldsymbol{A}^e + ik\hat{\boldsymbol{n}}\cdot\varepsilon_1\phi^e\,. && \text{(A.2b)}
\end{aligned}$$

Note that this set of equations can be also used in case of multiple materials, without introducing the L matrices as in Ref. [7], if one interprets ε_1 and ε_2 as diagonal matrices. Such approach has the advantage that one avoids the computation of the L matrices which becomes costly in case of hierarchical matrices. From Eq. (A.2b) we get

$$G_2 \boldsymbol{h}_2 = \Delta^{-1} \left\{ ik\hat{\boldsymbol{n}} \left(\varepsilon_1 - \varepsilon_2\right) G_2 \sigma_2 + \boldsymbol{\alpha}' \right\} , \qquad \Delta = \Sigma_1 - \Sigma_2 \, . \tag{A.3}$$

Inserting this expression into Eq. (A.2a) gives

$$G_2 \sigma_2 = \Sigma^{-1} \left\{ D^{e\prime} + ik\hat{\boldsymbol{n}} \left(\varepsilon_1 - \varepsilon_2\right) \Delta^{-1} \boldsymbol{\alpha}' \right\} , \qquad \Sigma = \varepsilon_1 \Sigma_1 - \varepsilon_2 \Sigma_2 + k^2 \left(\varepsilon_1 - \varepsilon_2\right) \hat{\boldsymbol{n}} \cdot \Delta^{-1} \hat{\boldsymbol{n}} \left(\varepsilon_1 - \varepsilon_2\right) \, . \tag{A.4}$$

Eqs. (A.1,A.3,A.4) are the working equations of our BEM approach. We first solve Eq. (A.4) to compute the surface charge distribution σ_2, which is then used to compute $\boldsymbol{h}_2$ from Eq. (A.3). Finally, the surface charge and current distributions σ_1 and $\boldsymbol{h}_1$ at the particle insides are obtained from Eq. (A.1). The set of matrix equations can be solved with only four matrix inversions and two matrix multiplications, if one neglects addition of matrices, multiplication by diagonal matrices, and multiplication of matrices with vectors.

Appendix B. Selected *H*-matrix manipulations

In this Appendix we briefly describe a few algorithms for H-matrix operations, including matrix-vector multiplication, H-matrix summation, multiplication of H-matrices, and inversion of H-matrices.

Appendix B.1. Matrix-vector multiplication

Consider a hierarchical matrix $A_{m\times m}$, consisting of either full or low-rank submatrices, and a full matrix $x_{m\times n}$ reducing in case of $n = 1$ to a normal vector. In the following we refer to x as a vector and use the shorthand notation $A_{\tau_1\times\tau_2}$ for the submatrix composed of clusters τ_1 and τ_2. The matrix-vector multiplication $y = Ax$ can then be performed by setting $y_{m\times n} \leftarrow 0$ and calling the following procedure with τ_1 and τ_2 initially set to the parent cluster:

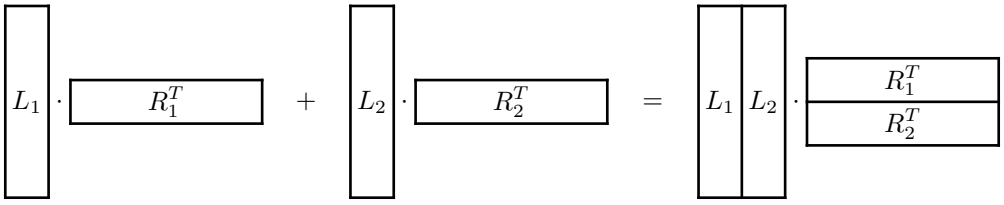
procedure MVMUL(var y, A, x, τ_1, τ_2) ▷ compute H-matrix vector product
  if admiss($\tau_1 \times \tau_2$) is low-rank matrix **then** ▷ low-rank approximation $A_{\tau_1\times\tau_2} \approx L_{\tau_1\times k} R^T_{k\times\tau_2}$
    $z_{k\times n} \leftarrow R^T_{k\times\tau_2} x_{\tau_2\times n}$ ▷ auxiliary vector
    $y_{\tau_1\times n} \leftarrow y_{\tau_1\times n} + L_{\tau_1\times k} z_{k\times n}$
  else if admiss($\tau_1 \times \tau_2$) is full matrix **then**
    $y_{\tau_1\times n} \leftarrow y_{\tau_1\times n} + A_{\tau_1\times\tau_2} x_{\tau_2\times n}$ ▷ full matrix $A_{\tau_1\times\tau_2}$
  else
    for $\tau_1', \tau_2' \leftarrow$ sons of τ_1, τ_2 **do** ▷ loop over sons of clusters τ_1 and τ_2
      MVMUL(y, A, x, τ_1', τ_2')
    end for
  end if
end procedure

For low-rank approximations the matrix-vector product can be computed much faster than for full matrices, and the speed up of the above algorithm depends on the compression factor for the H-matrix. A slight variant of the above algorithm also allows to compute H-matrices with diagonal matrices, as needed for our working BEM equations presented in Appendix A.

Appendix B.2. Addition of H-matrices

Contrary to other compression techniques for Green functions, such as the multipole expansion [27], H-matrices can be manipulated in a similar fashion to normal matrices. In the following we discuss a few

algorithms, a more exhaustive description can be found elsewhere [23, 24]. Adding two low-rank matrices of ranks k_1 and k_2

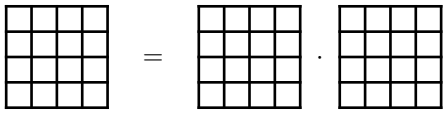


leads again to a low-rank matrix with rank $k_1 + k_2$. In principle, we could recompress the resulting low-rank matrix either through a QR-decomposition [24] or by calling the adaptive-cross approximation procedure ACA, as we do in our approach. The H-matrix summation without recompression can be done by initializing C with an empty H-matrix, and calling the following function with τ_1 and τ_2 set to the parent cluster:

procedure ADD(var C, A, B, τ_1, τ_2) ▷ add H-matrices $C = A + B$
  if admiss($\tau_1 \times \tau_2$) is low-rank matrix **then**
    $L^C_{\tau_1 \times (k_1+k_2)} = [L^A_{\tau_1 \times k_1} \; L^B_{\tau_1 \times k_2}]$ ▷ concatenate L-matrices horizontally
    $R^C_{\tau_2 \times (k_1+k_2)} = [R^A_{\tau_2 \times k_1} \; R^B_{\tau_2 \times k_2}]$ ▷ concatenate R-matrices horizontally
  else if admiss($\tau_1 \times \tau_2$) is full matrix **then**
    $C_{\tau_1 \times \tau_2} = A_{\tau_1 \times \tau_2} + B_{\tau_1 \times \tau_2}$ ▷ add full matrices
  else
    for $\tau_1', \tau_2' \leftarrow$ sons of τ_1, τ_2 **do** ▷ loop over sons of clusters τ_1 and τ_2
      ADD(C, A, B, τ_1', τ_2')
    end for
  end if
end procedure

Appendix B.3. Multiplication of H-matrices

The multiplication of H-matrices is doable but more difficult. Consider the multiplication of two matrices that are subdivided down to the smallest clusters, the leaves of the cluster tree



This multiplication can be performed by matrix multiplications of the smallest submatrices. Unfortunately things are not that easy because admissible matrices combine many of these smallest submatrices to form a single low-rank matrix. It is still possible to formulate a (surprisingly simple) algorithm for H-matrix multiplication, but we have to deal with a number of situations:

- Multplications of the form $\boxplus = \boxplus \cdot \boxplus$, all submatrices are subdivided and multiplication and summation is done in the subblocks, see procedure MULADD1.
- A low-rank matrix needs to be further split $\square \rightarrow \boxplus$. This can be done because of the structure of the cluster-tree underlying the H-matrices. In the computer code, splitting can be achieved through a suitable masking of the submatrices without copying any contents.
- Multplications of the form $\boxplus = \square \cdot \boxplus$ or $\boxplus \cdot \square$ can be performed by subdividing one of the matrices, see procedure MULADD1.
- Multplications of the form $\square = \boxplus \cdot \boxplus$ has to be treated separately, see procedure MULADD2.
- To deal with multiplications of the form $\boxminus \cdot \sqsubset\sqsupset$ or similar we have to modify the loops over a cluster τ such that it runs over the sons τ' if the cluster can be further split, and keeps ths value τ if the cluster is a leaf. We shall denote this with "$\tau' \leftarrow$ (sons of) τ".

- Multiplication of a full and a low-rank matrix

$$A_{m\times n}\Big(L_{n\times k}R_{k\times p}^T\Big) = \Big(A_{m\times n}L_{n\times k}\Big)R_{k\times p}^T \equiv \tilde{L}_{n\times k}R_{k\times p}^T$$

  results again in a low-rank matrix.

- Multiplication of two low-rank matrices

$$\Big(L^1_{m\times k}R^{1\,T}_{k\times n}\Big)\Big(L^2_{n\times k'}R^{2\,T}_{k'\times p}\Big) = L^1_{m\times k}\Big(\Big[R^{1\,T}_{k\times n}L^2_{n\times k'}\Big]R^{2\,T}_{k'\times p}\Big) \equiv L^1_{m\times k}\tilde{R}^{1\,T}_{k\times p}$$

  results again in a low-rank matrix.

- We have to ensure that the final matrix has proper storage format, and thus provide functions that convert a low-rank matrix to a full matrix through $A = LR$, and a full matrix to a low-rank approximation by calling the adaptive cross approximation ACA.

- Because of the many multiplications and summations performed it is necessary to recompress from time to time the H-matrices, which we do by calling the ACA procedure.

Multiplication of two H-matrices $C = AB$ is achieved by initializing C as an empty H-matrix and setting the cluster indices τ_1, τ_2, and τ_3 to the parent cluster. We then call:

```
procedure ADDMUL1(var C, A, B, τ1, τ2, τ3)                    ▷ C_{τ1×τ2} = A_{τ1×τ3} B_{τ3×τ2}
    if C_{τ1×τ2} is neither a full nor low-rank matrix then   ▷ ⊞ = □·⊞ or ⊞·□ or ⊞·⊞
        for τ1', τ2', τ3' ← (sons of) τ1, τ2, τ3 do
            ADDMUL(C, A, B, τ1', τ2', τ3')
        end for
    else                                                       ▷ C_{τ1×τ2} is a full or low-rank matrix
        if A_{τ1×τ3} and B_{τ3×τ2} are both full or low-rank matrices then
            C_{τ1×τ2} ← C_{τ1×τ2} + A_{τ1×τ3} B_{τ3×τ2}        ▷ □ = □·□
        else
            ADDMUL2(C_{τ1×τ2}, A, B, τ1, τ2, τ3)              ▷ □ = ⊞·⊞ or ⊞·□ or ⊞·□
        end if
    end if
end procedure
```

Finally, we have to deal with the situation that C is a full or low-rank matrix, and either A, or B, or both are not. If this is the case, we split the matrix $\tilde{C}$ according to the structure of the matrices A and B, compute the matrix product

$$\begin{array}{|c|c|}\hline C_{11} & C_{12} \\ \hline C_{21} & C_{22} \\ \hline \end{array} = \begin{array}{|c|c|}\hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} \cdot \begin{array}{|c|c|}\hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array},$$

and finally assemble together the blockmatrix C. Again we have to be careful about the fact that A and B can have also forms such as ⊟ or ▭. We then arrive at the following procedure:

```
procedure ADDMUL2(var C, A, B, τ1, τ2, τ3)                    ▷ C is full or low-rank matrix
    if A_{τ1×τ3} and B_{τ3×τ2} are both full or low-rank matrices then
        C ← C + A_{τ1×τ3} B_{τ3×τ2}                           ▷ multiplication of matrices
        ensure proper storage format
        (full or low-rank) for target matrix C
    else
        allocate C_{τ1×τ2} with proper block format           ▷ ⊞ or ⊟ or ▭
```

for $\tau_1', \tau_2', \tau_3' \leftarrow$ (sons of) τ_1, τ_2, τ_3 **do**
ADDMUL(C, A, B, τ_1', τ_2', τ_3') ▷ multiplication and summation in subblocks
end for
assemble block matrix C to single full or low-rank matrix
end if
end procedure

Appendix B.4. Inversion of H-matrices

The checkerboard structure of the H-matrices can be exploited to invert an H-matrix by using the relation

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}^{-1} = \begin{pmatrix} Y + YA_{12}S^{-1}A_{21}Y & -YA_{12}S^{-1} \\ -S^{-1}A_{21}Y & S^{-1} \end{pmatrix}, \quad Y = A_{11}^{-1}, \; S = A_{22} - A_{21}YA12,$$

For inversion we set the cluster index τ to the parent cluster and call the following procedure with an empty H-matrix C:

procedure INV(var C, A, τ) ▷ inversion $C_{\tau\times\tau} = A_{\tau\times\tau}^{-1}$
if $A_{\tau\times\tau}$ is full matrix **then**
$C \leftarrow A_{\tau\times\tau}^{-1}$ ▷ matrix inversion using LAPACK routine
else
$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$, $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ ▷ decompose matrix into blocks
INV(Y, A_{11}, τ_1) ▷ auxiliary matrix $Y \leftarrow A_{11}^{-1}$
$S \leftarrow A_{22} - A_{21}YA_{12}$ ▷ auxiliary matrix S
$C_{22} \leftarrow S^{-1}$, $C_{11} \leftarrow Y + YA_{12}C_{22}A_{21}Y$
$C_{12} \leftarrow -YA_{12}C_{22}$, $C_{21} \leftarrow -C_{22}A_{21}Y$
end if
end procedure

Appendix C. The *H*-matrix library of the MNPBEM toolbox

The MNPBEM toolbox provides its own H-library written in C++. This library has to be compiled to be used as MEX functions within Matlab (we distribute with the toolbox precompiled files for the most common operating systems), as discussed in more detail in the main text. Our library only contains the most basic features and is primarily considered for use within the Matlab environment. In the following we briefy discuss its main ingredients.

Appendix C.1. Options

The file `hoptions.h` contains the definition for some global parameters.

```
#define MEX
typedef std::complex<double> dcmplx;
typedef std::pair<size_t,size_t> pair_t;

//  option array for H-matrices
struct hoptions
{
  double tol;     //  tolerance for truncation of Rk matrices
  size_t kmax;    //  maximum rank for low-rank matrices
};
extern struct hoptions hopts;
extern std::map<std::string,double> timer;
```

By using `#undef MEX` one can remove the MEX definitions and use the library outside the Matlab environment. The structure `hopts` contains the tolerance for H-matrix operations and the maximum rank for the storage of H-matrices. The map `timer` is used for internal timing of the H-matrix library.

Appendix C.2. The `basemat` class

We provide our own class for basic matrix operations. Most importantly, the class (i) directly uses the BLAS and LAPACK libraries, and (ii) allows matrix masking. The latter feature is particularly important for H-matrix manipulations where sometimes full or low-rank matrices need to be further subdivided. Initialization of a matrix is done through

```
matrix<double> a(m,n), b(m,n,0.);   // initialization
a=matrix<double>::getmex(rhs);      // convert Matlab array to C++ matrix
```

The `getmex` function call allows importing the matrix through a MEX interface. The most important properties of the `matrix` class include

```
lhs=setmex(a); //  copy C++ matrix into Matlab array
a.nrows();     //  number of rows
a.ncols();     //  number of columns
a(i,j);        //  reference
a[i]           //  access elements (FORTRAN storage, rows first)
c=a+b; c=a*b;  //  basic matrix operations

mask_t amask=a.size();        //  get size of matrix (0,mrows,0,ncols)
at=transp(a);                 //  transpose of matrix
b=mask(a,amask)               //  matrix masking
copy(a,amask,b,bmask);        //  copy contents from a to b using masking
add(a,amask,b,bmask,c,cmask); // summation c=a+b using masking
                              //  multiplication c=a*b using masking, type="N", "T"
add_mul(a,amask,atype,b,bmask,btype,c,cmask);
c=add_mul(a,amask,atype,b,bmask,btype);
c=cat(a1,a2);                 //  concatenate matrices horizontally
c=cat(nrow,ncol,a1,a2,...);   //  concatenate multiple matrices
ai=inv(a);                    //  inverse of matrix (using LAPACK)
```

Appendix C.3. Cluster tree

The `clustertree.h` file defines one global cluster tree and two matrices `ind1` and `ind2` specifying the cluster pairs for the full and low-rank matrices

```
//  flag for low-rank and full matrices
#define flagRk 1
#define flagFull 2
extern clustertree tree;          //  one tree accessible for everyone
extern matrix<size_t> ind1, ind2; //  index to full and low-rank matrices
```

`flagRk` is the flag for low-rank matrices and `flagFull` the flag for full matrices. `ind1(i,0)` and `ind2(i,1)` give the cluster row and column index for the i'th full matrix (same for `ind2` and low-rank matrices). The most important properties and methods of this class include

```
tree.getmex(rhs,ind1,ind2); // initialize cluster tree from MEX call
tree.sons;                  //  cluster sons, matrix object
tree.size(i);               //  size of cluster i, (ibegin,iend)
tree.size(i,j);             //  size of subcluster i wrt cluster j, used for masking
tree.leaf(i);               //  determine whether cluster i is leaf
tree.admiss(i,j);           //  flagFull, flagRk, or NULL
```

`tree.size(i)` returns a pair where the first element points to the first boundary element of cluster i and the second element to the past-the-end element. `tree.size(i,j)` returns the same pair but for elements measured with respect to the parent cluster `j`. We also provide two iterator classes for looping through the cluster tree

```
// loop over sons of cluster ic
size_t ic=1;
// index of starting cluster
for (tree:iterator it=tree.begin(ic); it!=tree.end(); it++)
{
  *it;         // current cluster
  it->num;     // 0 for first son, 1 for second son
}
```

and

```
// loop through cluster trees starting from (ic0,ic1)
for (tree:iterator it=tree.pair_begin(ic0,ic1); it!=tree.pair_end(); it++)
{
  it->first;       // row index
  it->second;      // column index
}
```

Appendix C.4. Submatrices

The `submatrix` class holds the submatrices for the hierarchical matrix

```
template<class T>
class submatrix
{
public:
  matrix<T> mat, lhs, rhs; // either full matrix or low-rank matrix
  size_t row, col;         // row and column index of matrix
  ...
};
```

`row` and `col` are the cluster row and column, respectively. For a full matrix only the `mat` matrix is set and `lhs` and `rhs` are empty, whereas for a low-rank matrix `mat` is empty and `lhs` and `rhs` are the low-rank parts of the matrix `lhs*transp(rhs)`. The class contains the following important functions:

```
submatrix<double> A(row,col,mat);    // initialization with full matrix
submatrix<double> A(row,col,lhs,rhs); // initialization with Rk matrices

A.nrows();          // number of rows of submatrix
A.ncols();          // number of columns of submatrix
A.empty();          // matrix initialized ?
A.flag();           // flagFull or flagRk
A.convert(flag);    // convert storage format to flagFull or flagRk

C=A+B; C=A-B;       // basic arithmetic operations
add_mul(A,x,y);     // y = y + A*x
C=add(A,B);         // summation of sub-matrices
A+=B;               // add sub-matrices and compress low-rank matrices using ACA
C=mul(A,B,i,j,k);   // multiplication C(i,j) = A(i,k)*B(k,j)
A=cat(Amatrix,i,j); // concatenate matrix with sub-matrices to larger sub-matrix
```

The `mul` function can be used to subdivide a full or low-rank matrix, as needed for H-matrix multiplications.

Appendix C.5. Hierarchical matrices

The `hmatrix` class uses the above classes to define hierarchical matrices. In the class definition we save the submatrices in the form of a `std::map<pair_t,submatrix<T> >`, where the first and second entry of the `pair_t` variable hold the row and column of the submatrix, respectively.

```
template<class T>
class hmatrix
{
public:
  typedef typename std::map<pair_t,submatrix<T> >::iterator iterator;
  typedef typename std::map<pair_t,submatrix<T> >::const_iterator const_iterator;

  std::map<pair_t,submatrix<T> > mat;

  // find submatrix, NULL if not present
  submatrix<T>* find(size_t row, size_t col)
    { iterator it=mat.find(pair_t(row,col));
      return (it!=mat.end()) ? &it->second : 0;
    }
  // initialize H-matrix from MEX call
  static hmatrix<T> getmex(const mxArray* prhs[]);
  ...
};
```

We also provide iterators for accessing the submatrices. The function `find` returns a pointer to a requested submatrix and zero if the submatrix is not found in `mat`. The function `getmex` provides a gateway between Matlab and C++.

Muliplication of a H-matrix A with a vector x is performed in the following function:

```
// multiplication with matrix, y = A*x
template<class T>
matrix<T> hmatrix<T>::operator* (const matrix<T>& x) const
{
  matrix<T> y=matrix<T>(x.nrows(),x.ncols(),(T)0);

  // loop over all submatrices and perform submatrix-vector multiplication
  for (const_iterator it=mat.begin(); it!=mat.end(); it++) add_mul(it->second,x,y);
  return y;
}
```

The function loops over all submatrices and calls the function `add_mul` defined in `submatrix.h`. Summation of H-matrices is achieved through

```
// add two H-matrices using tree, C(H) += A(H) + B(H)
template<class T>
void add(const hmatrix<T>& A, const hmatrix<T>& B, hmatrix<T>& C, size_t i=0, size_t j=0)
{
  // start at cluster pair (i,j) and move down the tree
  for (pairiterator it=tree.pair_begin(i,j); it!=tree.pair_end(); it++)
    C[*it]+=*A.find(it->first,it->second)+*B.find(it->first,it->second);
}
```

In the `for` loop we move through all submatrices, starting from the cluster pair (i, j), and add the sum defined in `submatrix.h` to the target matrix C. Note that the `+=` operator call for submatrices also invokes a recompression using the adaptive cross approximation, with the tolerance and maximal rank `kmax` defined in the `hopts` structure.

We finally demonstrate that the implementation for H-matrix multiplications is relatively simply with our `submatrix` definition.

```
// recursive function for H-matrix multiplication, C(H) = A(sub,H)*B(sub,H)
template<class T, class Amat, class Bmat>
void add_mul(const Amat& A, const Bmat& B, hmatrix<T>& C, size_t i, size_t j, size_t k)
{
```

```
  // are matrices of type submatrix ?
  const submatrix<T> *pA=A.find(i,k), *pB=B.find(k,j);
  // admissibility of C matrix
  short adC=tree.admiss(i,j);

  if (adC==0)
    // all matrices are subdivided
    for (treeiterator ii=tree.begin(i); ii!=tree.end(); ii++)
    for (treeiterator jj=tree.begin(j); jj!=tree.end(); jj++)
    for (treeiterator kk=tree.begin(k); kk!=tree.end(); kk++)
    {
      if (pA==0 && pB==0)
        add_mul(A,B,C,*ii,*jj,*kk);                    // C(H) = A(H)*B(H)
      else if (pA!=0 && pB==0)
        add_mul(*pA,B,C,*ii,*jj,*kk);                  // C(H) = A(sub)*B(H)
      else if (pA==0 && pB!=0)
        add_mul(A,*pB,C,*ii,*jj,*kk);                  // C(H) = A(H)*B(sub)
      else
        add_mul(*pA,*pB,C,*ii,*jj,*kk);                // C(H) = A(sub)*B(sub)
    }
  else if (adC!=0 && pA!=0 && pB!=0)
    C.mat[pair_t(i,j)]+=mul(*pA,*pB,i,j,k).convert(adC);  // C(sub) = A(sub)*B(sub)
  else
    C.mat[pair_t(i,j)]+=add_mul2<T>(A,B,i,j,k,adC);       // C(sub) = A(H)*B(H)
}
```

In the first two lines of the function we determine whether the matrices are full or low-rank matrices, or can be further subdivided. If the H-matrix `C` can be further subdivided, we loop over all subclusters and perform the multiplication $C_{\tau_i \times \tau_j} = A_{\tau_i \times \tau_k} B_{\tau_k \times \tau_j}$ using matrix splitting. Whenever either `A` or `B` is a submatrix, we recall the function `add_mul` by replacing the H-matrix argument by the corresponding submatrix. Further splitting is then done through masking of the submatrices. Finally, when `C` is a submatrix and `A`, `B` not, we call the function

```
// recursive function for H-matrix multiplication, C(sub) = A(sub,H)*B(sub,H)
template<class T, class Amat, class Bmat>
submatrix<T> add_mul2(const Amat& A,
                      const Bmat& B, size_t i, size_t j, size_t k, short flag)
{
  // are matrices of type submatrix ?
  const submatrix<T> *pA=A.find(i,k), *pB=B.find(k,j);

  if (pA && pB)
    return mul(*pA,*pB,i,j,k).convert(flag);
  else
  {
    // subdivide matrices
    treeiterator ii=tree.begin(i), jj=tree.begin(j), kk;
    matrix<submatrix<T> > C(ii->size(),jj->size());

    // matrix multiplication with subdivided matrices
    for (ii=tree.begin(i); ii!=tree.end(); ii++)
    for (jj=tree.begin(j); jj!=tree.end(); jj++)
    for (kk=tree.begin(k); kk!=tree.end(); kk++)
      if (pA)
        C(ii->num,jj->num)+=add_mul2<T>(*pA,B,*ii,*jj,*kk,flag);  // A(sub)*B(H)
      else if (pB)
        C(ii->num,jj->num)+=add_mul2<T>(A,*pB,*ii,*jj,*kk,flag);  // A(H)*B(sub)
```

```
      else
        C(ii->num,jj->num)+=add_mul2<T>(A,B,*ii,*jj,*kk,flag);    // A(H)*B(H)

    // assemble matrix
    return cat(C,i,j);
  }
}
```

The function again determines whether `A` and `B` are admissible matrices or H-matrices, and then fills the auxiliary `C` matrix according to the algorithm given in Appendix B. At the end, the target matrix is assembled together and returned to the calling function `add_mul`. We finally give the code for matrix inversion which closely follows the algorithm previously given:

```
// recursive inversion of H-matrix
template<class T>
hmatrix<T> inv(const hmatrix<T>& A, size_t i=0)
{
  hmatrix<T> C;                                          // return matrix
  size_t i0=tree.sons(i,0), i1=tree.sons(i,1);           // sons of cluster

  if (i0==0 && i1==0)                                    // full matrix ?
  {
    const submatrix<T>* it=A.find(i,i);                  // get submatrix pointer
    C[pair_t(i,i)]=submatrix<T>(i,i,inv(it->mat));       // calculate the inverse exactly
  }
  else
  {
    // working matrices
    hmatrix<T> Y,S,t;
    // Schur decomposition of subdivided matrix, Y = inv(A00)
    Y=inv(A,i0);
    // S = A11 - A10 * Y * A01
    S=subtract(A,mul(A,mul(Y,A,i0,i1,i0),i1,i1,i0),i1,i1);
    // invert S, C11 = inv(S)
    C=inv(S,i1);

    // C00 = Y + Y * A01 * C11 * A10 * Y
    t=mul(Y,mul(A,mul(C,mul(A,Y,i1,i0,i0),i1,i0,i1),i0,i0,i1),i0,i0,i0);
    add(Y,t,C,i0,i0);
    // C01 = - Y * A01 * C11
    add_mul(uminus(Y,i0,i0),mul(A,C,i0,i1,i1),C,i0,i1,i0);
    // C10 = - C11 * A10 * Y
    add_mul(C,mul(A,uminus(Y,i0,i0),i1,i0,i0),C,i1,i0,i1);
  }

  return C;
}
```


[1] A. Taflove, S. C. Hagness, Computational electrodynamics, Artech House, Boston, 2005.
[2] U. S. Inan, R. A. Marshall, Numerical Electromagnetics, Cambridge, Cambridge, 2011.
[3] J. S. Hesthaven, High-order accurate methods in time-domain computational electromagnetics: A review, Advances in Imaging and Electron Physics 127 (2003) 59.
[4] K. Busch, M. König, J. Niegemann, Discontinuous galerkin method in nanophotonics, Laser Photonics Rev. 5 (2011) 773–809.
[5] B. T. Draine, P. J. Flateau, Discrete-dipole approximation for scattering calculations, J. Opt. Soc. Am. A 11 (1994) 1491.
[6] J. A. Stratton, L. J. Chu, Diffraction theory of electromagnetic waves, Phys. Rev. 56 (1939) 99–107.
[7] F. J. Garcia de Abajo, A. Howie, Retarded field calculation of electron energy loss in inhomogeneous dielectrics, Phys. Rev. B 65 (2002) 115418.

[8] S. A. Maier, Plasmonics: Fundamentals and Applications, Springer, Berlin, 2007.
[9] R. Fuchs, S. H. Liu, Sum rule for the polarizability of small particles, Phys. Rev. B 14 (1976) 5521.
[10] F. J. Garcia de Abajo, Optical excitations in electron microscopy, Rev. Mod. Phys. 82 (2010) 209.
[11] B. Gallinet, J. Butet, O. J. F. Martin, Numerical methods for nanophotonics: standard problems and future challenges, Laser Photonics Rev. 9 (2015) 577.
[12] V. Myroshnychenko, E. Carbo-Argibay, I. Pastoriza-Santos, J. Perez-Juste, L. M. Liz-Marzán, F. Javier Garcia de Abajo, Modeling the optical response of highly faceted metal nanoparticles with a fully 3d boundary element method, Adv. Mater. 20 (2008) 4288.
[13] R. Esteban, G. Aguirregabiria, A. G. Borisov, Y. M. Wang, P. Nordlander, G. W. Bryant, J. Aizpurua, The morphology of narrow gaps modifies the plasmonic response, ACS Photon. 2 (2015) 295.
[14] U. Hohenester, A. Trügler, MNPBEM - A Matlab Toolbox for the simulation of plasmonic nanoparticles, Comp. Phys. Commun. 183 (2012) 370.
[15] U. Hohenester, Simulating electron energy loss spectoscopy with the mnpbem toolbox, Comp. Phys. Commun. 185 (2014) 1177.
[16] J. Waxenegger, A. Trügler, U. Hohenester, Plasmonics simulations with the mnpbem toolbox: Consideration of substrates and layer structures, Comp. Phys. Commun. 193 (2015) 138.
[17] E. van 't Wout, P. Gélat, T. Betcke, S. Arridge, A fast boundary element method for the scattering analysis of high-intensity focused ultrasound, J. Acoust.Soc. Am. 138 (2015) 2726.
[18] S. P. Groth, A. J. Baran, T. Betcke, S. Havemann, W. Śmigaj, The boundary element method for light scattering by ice crystals and its implementation in bem++, Journal of Quantitative Spectroscopy and Radiative Transfer 167 (2015) 40–52.
[19] O. C. Zienkiewicz, R. L. Taylor, The finite element method, McGraw Hill, Maidenhead, 1989.
[20] C. Gruber, A. Hirzer, V. Schmidt, A. Trügler, U. Hohenester, H. Ditlbacher, A. Hohenau, J. R. Krenn:, Imaging nanowire plasmon modes with two-photon polymerization;, Appl. Phys. Lett. 106 (2015) 081101.
[21] F. P. Schmidt, H. Ditlbacher, A. Tügler, U. Hohenester, A. Hohenau, F. Hofer, J. R. Krenn, Plasmon modes of a silver thin film taper probed with stem-eels, Opt. Lett. 40 (2015) 5670.
[22] G. Haberfehlner, A. Trügler, F. P. Schmidt, A. Hörl, F. Hofer, U. Hohenester, G. Kothleitner, Correlated 3d nanoscale mapping and simulation of coupled plasmonic nanoparticles, Nano Lett. 15 (2015) 7726.
[23] W. Hackbusch, A sparse matrix arithmetic based on h-matrices. part i: Introduction to h-matrices, Computing 62 (1999) 89.
[24] S. Börm, L. Grasedyck, W. Hackbusch, Introduction to hierarchical matrices with applications, Engineering analysis with boundary elements 27 (2003) 405.
[25] P. B. Johnson, R. W. Christy, Optical constants of the noble metals, Phys. Rev. B 6 (1972) 4370.
[26] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Numerical Recipes in C++: The Art of Scientific Computing, 2nd Edition, Cambridge Univ. Press, Cambridge, 2002.
[27] R. Coifman, V. Rokhlin, S. Wandzura, The fast multipole method for the wave equation: A pedestrian prescription, IEEE Antennas and Propagation Magazine 35 (1993) 7.
[28] M. Bebendorf, Approximation of boundary element matrices, Numer. Math. 86 (2000) 565.