Julia and the numerical homogenization of PDEs

Clemens Heitzinger
School of Mathematical and Statistical Sciences
Arizona State University (ASU)
P.O. Box 871804
Tempe, Arizona 85287-1804
and
Institute for Analysis and Scientific Computing
Vienna University of Technology (TU Wien)
Wiedner Hauptstraße 8–10
A-1040 Vienna, Austria
Clemens.Heitzinger@ASU.edu

Gerhard Tulzer
Institute for Analysis and Scientific Computing
Vienna University of Technology (TU Wien)
Wiedner Hauptstraße 8–10
A-1040 Vienna, Austria
Gerhard.Tulzer@TUWien.ac.at

ABSTRACT
We discuss the advantages of using JULIA for solving multiscale problems involving partial differential equations (PDEs). Multiscale problems are problems where the coefficients of a PDE oscillate rapidly on a microscopic length scale, but solutions are sought on a much larger, macroscopic domain. Solving multiscale problems requires both a theoretical result, i.e., a homogenization result yielding effective coefficients, as well as numerical solutions of the PDE at the microscopic and the macroscopic length scales.

Numerical homogenization of PDEs with stochastic coefficients is especially computationally expensive. Under certain assumptions, effective coefficients can be found, but their calculation involves subtle numerical problems. The computational cost is huge due to the generally large number of stochastic dimensions.

Multiscale problems arise in many applications, e.g., in uncertainty quantification, in the rational design of nanoscale sensors, and in the rational design of materials.

Our code for the numerical stochastic homogenization of elliptic problems is implemented in JULIA. Since multiscale problems pose new numerical problems, it is in any case necessary to develop new numerical codes. JULIA is a dynamic language inspired by the Lisp family of languages [1, 8, 16] with a syntax that is more common in scientific computing. It is available under the MIT License and it provides advanced features such as macros and multiple dispatch similar to CLOS [14,15]. Second, JULIA provides compilation to native-code and a language design that promises excellent performance for numerical problems.

This combination of advanced programming features, generation of fast native code, and support for scientific computing is attractive when solving non-trivial numerical problems.

In this work, we discuss our experience with JULIA so far. We start by introducing the problem domain of multiscale problems involving PDEs in Section 2 in order to provide background information about our requirements. Then, in Section 3, we describe how the features of JULIA have proven to be highly useful in this area so far. We also give some concrete code examples. Finally, conclusions are drawn in Section 4.

1. INTRODUCTION
The numerical homogenization of PDEs leads to interesting and subtle numerical problems that require new algorithms and new codes. As we work on various multiscale problems with large computational requirements, we are developing numerical codes for their solution. We have recently begun to use the rather new dynamic language JULIA.

The reasons are twofold: First, JULIA can be viewed as continuing the tradition of the Lisp family of languages [1, 8, 16] with a syntax that is more common in scientific computing. It is available under the MIT License and it provides advanced features such as macros and multiple dispatch similar to CLOS [14,15]. Second, JULIA provides compilation to native-code and a language design that promises excellent performance for numerical problems.

This combination of advanced programming features, generation of fast native code, and support for scientific computing is attractive when solving non-trivial numerical problems.

In this work, we discuss our experience with JULIA so far. We start by introducing the problem domain of multiscale problems involving PDEs in Section 2 in order to provide background information about our requirements. Then, in Section 3, we describe how the features of JULIA have proven to be highly useful in this area so far. We also give some concrete code examples. Finally, conclusions are drawn in Section 4.

2. MULTISCALE PROBLEMS AND COMPUTATIONAL CHALLENGES
We describe the problem domain of the homogenization of PDEs in Section 2.1. The challenges stemming from the numerical homogenization of PDEs are then discussed in Section 2.4.

2.1 Multiscale problems
Multiscale problems arise in applications whenever the material or device to be modeled has a fine structure, but we are interested in the macroscopic properties of the material or the behavior of a whole device. The microscopic structure is modeled by the coefficients of a PDE, which can be deterministic (i.e., periodic) or stochastic. The microscopic length scale is denoted by $\epsilon \ll 1$ and the solution $u_\epsilon$ of the PDE depends on $\epsilon$ via the fast oscillations of the coefficient.
functions \( f_\epsilon(x) = f(x/\epsilon) \).

Homogenization means that the limit of \( u_\epsilon \) as \( \epsilon \to 0 \) is sought. If homogenization is possible, the homogenized solution \( u_0 \) satisfies the homogenized equation. The coefficients of the homogenized equation are the effective coefficients that do not depend on the microscopic length scale \( \epsilon \) anymore.

Without homogenization, it is practically impossible to solve the original PDE on a large domain and on a mesh that is fine enough to reflect the microscopic length scale. After homogenization, the homogenized equation can be solved for the limit \( u_0 \) on the macroscopic domain without considering the microscopic length scale, since the effective coefficients do not depend on \( \epsilon \) anymore. Higher-order corrections can also be calculated contingent on theoretic results. This is the computational significance of homogenization.

In summary, homogenization of PDEs is the mathematically sound way to solve multiscale problems. Solving a multiscale problems consists of two parts:

- A homogenization result yields the homogenized equation and the effective coefficients.

- The effective coefficients must be calculated numerically. This is computationally expensive especially in the case of stochastic coefficients, since in general many solutions of a deterministic problem must be calculated.

2.2 Elliptic problems

Elliptic PDEs are one of the most important classes of PDEs, since they occur in many physical contexts. For example, they describe diffusive transport and arise in electrostatics. They are among the best studied PDEs and therefore efficient algorithms for their numerical homogenization are an obvious goal.

Numerical stochastic homogenization of elliptic problems means finding effective coefficients for the limit \( \epsilon \to 0 \) for the equation

\[ -\nabla \cdot (A_\epsilon(x, \omega)\nabla u_\epsilon(x, \omega)) = f_\epsilon(x, \omega), \]

where \( u \) is the unknown, \( A \) is a coefficient matrix, \( f \) is a forcing term, \( x \) denotes position, \( \omega \) is a random variable, and the dependence on \( \epsilon \) signifies fast oscillations in \( A_\epsilon(x,\omega) = A(x/\epsilon, \omega) \) and \( f_\epsilon(x, \omega) = f(x/\epsilon, \omega) \).

The dependence on the random variable \( \omega \) includes many realistic applications, but also means that the problem does not only depend on up to three spatial dimensions, but on many stochastic dimensions. Although theoretic results are available under certain assumptions, numerical calculations pose several subtle numerical problems and efficient numerical algorithms still need to be developed.

In summary, there are many open questions even in the case of elliptic problems, and many considerations must be taken into account when developing numerical codes. The challenges are discussed in more detail in Section 2.4.

2.3 Applications

There are many applications where multiscale problems arise. We have recently focused on new applications in nanotechnology, where different length scales are common due to fine structures at the nanometer scale, but simulations at larger scales are required in order to understand functional devices.

We have recently solved multiscale problems arising from nanoscale bio- and gas sensors. We have worked on the homogenization of deterministic and stochastic PDEs [2, 9, 10, 12, 13, 19], we have developed efficient algorithms based on the theoretic results, we have implemented parallel numerical codes [3], and we have used them for the rational design of these sensors [4–7, 11, 18, 20].

Of course, there are many other applications, for example in geology, uncertainty quantification, and materials science.

2.4 Challenges: where dynamic languages help

The discussion in this section shows that there are various requirements in numerical homogenization that necessitate the development of new numerical codes. Among the reasons and requirements are:

- The method of choice for the spatial dimensions is the finite-element method. However, there are special requirements: In important applications, the permittivity \( \varepsilon \) has high contrast so that structurally aligned meshes are highly desirable. This requires control over mesh generation and discretization.

- In contrast to periodic homogenization of elliptic problems, the cell problem in stochastic homogenization is posed on the whole space \( \mathbb{R}^3 \). This requires a nonstandard discretization.

- Because of the interplay of the spatial and stochastic dimensions, complete control over the discretizations of both is needed.

- Stochastic processes that generate the underlying geometry must be implemented and affect the discretization. The stochastic processes also vary from application to application.

- Due to the large computational cost of stochastic homogenization, parallelization is indispensable. There are various opportunities for parallelization that should be explored. At this point, it is largely unknown which approaches will be most promising.

Because of these reasons, we need complete control over all aspects of discretization, while still being able to explore various algorithmic approaches. In other words, we would ideally like to use a high-level language that provides both fast native-code generation and dynamic features.

In the next section, we discuss how a dynamic language such as JULIA can help to address these challenges and give some concrete examples.

3. JULIA AND ITS ADVANTAGES

3.1 Julia in numerical homogenization

We have seen in the previous section that in order to make progress in the area of numerical homogenization, it is not sufficient to implement well-understood numerical algorithms, but it is necessary to be able to explore the interactions between algorithms and error estimates in the various steps involved in numerical homogenization. Therefore any support from modern programming concepts and fast native-code generation are highly welcome when solving this type of problems.
JULIA has proven to be a very productive language. Many design decisions were made with scientific computing in mind. Among the advantages of JULIA are the following:

- Development in a REPL is interactive.
- The code base is relatively small thanks to expressive language features.
- This is also important in view of the notion that the number of defects per line of code (or per number of statements) is relatively constant regardless of programming language.
- Due to a modern choice of data structures, JULIA is useful for general programming tasks in addition to native-code compilation and access to highly optimized linear algebra routines.
- This is especially useful, since in this problem domain various different types of numerical problems must be solved ideally within a single code base.
- Parallelization is supported.
- Sparse matrices are supported.
- Macros can be used to shift computations from run time to compile time and are generally useful to design a domain specific language.
- Many students have already been exposed to Matlab or Octave so that they are already familiar with the general syntax.
- Still, despite the syntax that is unlike Lisp, JULIA is homoiconic and supports metaprogramming.
- JULIA is distributed under an open-source license, namely the MIT License.

The domain of the cell problem and realistic stochastic processes pose requirements on the mesh generation that we have solved so far by constructing realizations of the geometry in JULIA and then using gmsh as a backend. We use periodic boundary conditions on domains of variable size. The permittivity $A$ has inclusions, i.e., it is piecewise constant, and the precise geometry of each realization is constructed in JULIA first according to a given stochastic process. Then the geometry is transformed into a valid gmsh input file. JULIA acts as a general-purpose programming language here offering much better algorithmic support compared to languages with only basic data structures.

Finally, we would also like to mention that there is at least another example in numerical analysis or in finite elements in particular where a high-level dynamic language was used to implement a large numerical code base. It is FEMLISP [17], a Common Lisp framework for solving PDEs using the finite-element method.

### 3.2 Examples

We discuss two code examples.

The first example stems from the assembly of the finite-element system matrix. Assembly is, in fact, a time critical step. Iterative solvers for linear systems are very advanced nowadays so that the computational cost of the assembly step is significant. It is not sufficient to construct the sparse system matrix in the naive manner by inserting the coefficients one at a time in a loop over all finite elements. A much better method is to collect the entries of the system matrix in three vectors containing the two indices and the corresponding value. The macro that implements this method is shown in Figure 1. Macros are even more useful for the efficient implementation of higher-order hp-FEMs.

The second example concerns parallelization. We have calculated many tens of thousands solutions of PDEs and are now using up to 400 nodes of a cluster concurrently. Figure 2 shows how results of a straightforward Monte-Carlo approach are collected. The function $\text{hom}$ distributes the calculations, and the function $\text{hhcat}$ provides a sanity check when collecting the results.

### 3.3 Outlook

JULIA is a young language and there are many open homogenization problems. Therefore, we are looking forward to future improvements both in the JULIA language and in our handling of numerical problems using JULIA.

We mention some areas where improvements are expected or would be very welcome:

- Lazy sequences seem to be the state-of-the-art data structure to handle Monte-Carlo or quasi-Monte-Carlo sequences.
- Clojure provides excellent support for parallel computations among any dynamic language. It could be beneficial to introduce some of its constructs into JULIA.
- Support of sparse matrices is essential for the finite-element methods.
- A package for flexible mesh generation in JULIA would be a highly useful addition. However, specific demands in the area of numerical homogenization will probably always require custom code.

### 4. CONCLUSIONS

Programming languages for scientific computing in the widest sense have historically been divided into the Fortran and the Lisp family of languages. Regarding scientific computing, the Fortran family of languages emphasizes the compilation of functions and matrix computations into fast native code, while the Lisp family of languages is multiparadigm and emphasizes homoiconicity, functions as first-class objects (including multiple inheritance in CLOS), and introspection.

It is unfortunate that these two strands of languages have developed separately for such a long time. The promise of the JULIA language is to combine dynamic language concepts as known from the Lisp family of languages with the ability to generate fast native code without the need of an abundance of declarations by employing modern compiler technology. JULIA could thus become the proverbial “sufficiently smart compiler.”

Our experience while developing numerical algorithms for numerical homogenization has been very positive so far. Language features such as support for parallel computations, for sparse matrices, and the macro system have been especially useful. Numerical homogenization poses a variety of numerical problems and therefore any help from dynamic and efficient languages such as JULIA is highly welcome to advance the state of the art.
macro set_IJV!(i_sparse, nodes, MM)
    blk = Expr(:block)
    for j in 1:3, i in 1:3
        push!(blk.args,
            quote
                @inbounds I[$i_sparse] = $nodes[$i]
                @inbounds J[$i_sparse] = $nodes[$j]
                @inbounds V[$i_sparse] = $MM[$i, $j]
                @inbounds $i_sparse += 1
            end)
    end
    blk
end

## While iterating over the elements in the inner loop,
## the local system matrix is recorded.
@set_IJV!(i_sparse, node_indices, M0)

## After the inner loop, the sparse system matrix is constructed.
## Note that the default combine function is +.
M = sparse(I, J, V)

Figure 1: Macro used in the inner loop over all elements when assembling the system matrix.

function hhcat(a, b)
    ## Ignore vectors that are not finite.
    a_ok = isa(a, Array{Float64}) && all(isfinite(a))
    b_ok = isa(b, Array{Float64}) && all(isfinite(b))
    if a_ok && b_ok
        hcat(a, b)
    elseif a_ok
        a
    elseif b_ok
        b
    end
end

function hom(N)
    solutions = @parallel hhcat for i in 1:N
        info("[$(myid())] Calculating solution #$i.")
        solve(1, 3, 9)
    end
    if isa(solutions, Array{Float64, 2}) && size(solutions, 2) >= 1
        info("Calculated $(size(solutions, 2)) solutions.")
        solutions, mean(solutions, 2), std(solutions, 2)
    else
        solutions
    end
end

Figure 2: This function distributes the calculation of \( N \) solutions of a PDE on a cluster. Here a straightforward Monte-Carlo method is implemented implicitly in solve.
5. ACKNOWLEDGMENTS

The authors acknowledge support by the FWF (Austrian Science Fund) START project no. Y660 PDE Models for Nanotechnology. The authors also acknowledge Renate Mittelmann’s support in running simulations on the cluster of the School of Mathematical and Statistical Sciences at ASU.

6. REFERENCES