EasyChair Preprint

№ 778

Towards the automatic optimization of geometric multigrid methods with evolutionary computation

Jonas Schmitt, Sebastian Kuckuk and Harald Köstler

February 10, 2019

TOWARDS THE AUTOMATIC OPTIMIZATION OF GEOMETRIC MULTIGRID METHODS WITH EVOLUTIONARY COMPUTATION *

JONAS SCHMITT †

In collaboration with: Sebastian Kuckuk, Harald Köstler

Abstract. For many linear and nonlinear systems that arise from the discretization of partial differential equations the construction of an efficient multigrid solver is a challenging task. Here we present a novel approach for the optimization of geometric multigrid methods that is based on evolutionary computation, a generic program optimization technique inspired by the principle of natural evolution. A multigrid solver is represented as a tree of mathematical expressions which we generate based on a tailored grammar. The quality of each solver is evaluated in terms of convergence and compute performance using automated Local Fourier Analysis (LFA) and roofline performance modeling, respectively. Based on these objectives a multi-objective optimization is performed using strongly typed genetic programming with a non-dominated sorting based selection. To evaluate the model-based prediction and to target concrete applications, scalable implementations of an evolved solver can be automatically generated with the ExaStencils code generation framework. We demonstrate our approach by constructing multigrid solvers for Poisson's equation with constant and variable coefficients.

Key words. geometric multigrid, automatic program optimization, genetic programming, evolution strategy, code generation, local Fourier analysis

1. Introduction. Solving the linear or nonlinear systems that arise from the discretization of partial differential equations (PDEs) is an outstanding challenge. The huge number of unknowns in many of these systems necessitates the design of efficient and scalable solvers. Unfortunately, the optimal solution method highly depends on the system itself and it is therefore infeasible to formulate a single algorithm that works efficiently in all cases. Multigrid methods are a class of asymptotically optimal solution algorithms for (non-)linear systems. Although in the last decades great effort has been put into the design of efficient multigrid methods, for many important cases, such as the Navier-Stokes or the Schrödinger equation, this task is still not fully solved. Within this paper we propose a novel approach for the automatic optimization of multigrid solvers through the use of evolutionary computation. Our approach builds on the work by Mahmoodabadi and Köstler [15], in which it was first demonstrated how genetic programming (GP) [12] can be used to optimize the iteration matrices of stationary iterative methods. In contrast to [15], where the iteration matrix was assembled explicitly, we represent iterative methods as symbolic expressions, which are independent of the size of the linear system. For this purpose we propose a new formal grammar for the automatic generation of multigrid expressions. We furthermore show how we can automatically obtain convergence and performance estimates for geometric multigrid solvers on rectangular grids of arbitrary size and how based on these metrics a multi-objective optimization can be performed using genetic programming and evolution strategies (ES) [1]. In addition, the evolved multigrid solvers are emitted in form of a domain specific language (DSL) specification and the ExaStencils code generation framework is employed to automatically generate a

*This work is supported by the German Research Foundation (DFG), as part of the Priority Programme 1648 "Software for Exascale Computing" in the project under Contract RU 422/15-2.

†Friedrich-Alexander-Universität Erlangen-Nürnberg, (jonas.schmitt@fau.de, https://www.cs10.tf.fau.de/person/jonas-schmitt).

scalable implementation on the target platform. Our approach is therefore similar to the work by Thekale et al [22], where the optimal number of full multigrid cycles was optimized based on a cost model, but aims to achieve more generality by considering the construction of multigrid expressions as program optimization task. Finally, we demonstrate our approach by solving Poisson's equation in two dimensions with constant and variable coefficients.

2. A Formal Grammar for Generating Multigrid Solvers. The task of constructing a multigrid solver for a certain problem is typically performed by a human expert with profound knowledge in numerical mathematics. To automate this task, we first need a way to represent arbitrary multigrid solvers in a formal language that we can then use to automatically construct different instances on a computer. The rules of this formal language must ensure that only valid solver instances can be defined, which means that we can both automatically determine their convergence and runtime behavior. Additionally, we want to enforce that the generated method works on a hierarchy of grids, which requires the availability of inter-grid operations that allow to obtain approximations of the same operator or grid on a finer or coarser level. Consider the general system of linear equations defined on a grid with spacing h

$$A_h u^h = f^h. \tag{2.1}$$

Each step of an arbitrary multigrid method can be written in the following form:

$$u_{i+1}^h = u_i^h + \omega B_h (f^h - A_h u_i^h), \tag{2.2}$$

where u_i^h and f^h are the current approximate solution and right-hand side, respectively, $\omega \in \mathbb{R}$ the relaxation factor and B_h an operator defined on the given level h. For example, with the splitting $A_h = D_h - U_h - L_h$, we can define the Jacobi

$$u_{i+1}^h = u_i^h + D_h^{-1} (f^h - A_h u_i^h) \tag{2.3}$$

and the lexicographical Gauss-Seidel method

$$u_{i+1}^h = u_i^h + (D_h - L_h)^{-1} U_h (f^h - A_h u_i^h). \tag{2.4}$$

If we assume the availability of a prolongation operator P_H, a restriction operator R_h and an approximation for the inverse of A_h on the coarser grid, a coarse grid correction can be defined as

$$u_{i+1}^h = u_i^h + P_H A_H^{-1} R_h (f^h - A_h u_i^h). \tag{2.5}$$

Furthermore, we can substitute u_i^h in (2.5) with (2.3) and obtain a two grid with Jacobi pre-smoothing

$$u_{i+1}^h = (u_i^h + D_h^{-1}(f^h - A_h u_i^h)) + P_H A_H^{-1} R_h (f^h - A_h (u_i^h + D_h^{-1}(f^h - A_h u_i^h))). \tag{2.6}$$

By repeatedly substituting subexpressions we can automatically construct a single expression for any multigrid solver. If we take the set of possible substitutions as a basis, we can define a list of rules according to which we can generate such an expression. We specify these rules in the form of a context-free grammar, which is described in figure 2.1 for three multigrid levels. Figure 2.1a contains the production rules while figure 2.1b describes their semantics. Each rule defines the set of

$$
\begin{aligned}
\langle S \rangle &\models \textsc{iterate}(\langle c^h \rangle,\ \omega,\ \langle \mathcal{P} \rangle) \\
\langle c^h \rangle &\models \textsc{apply}(\langle B_h \rangle,\ \langle c^h \rangle) \mid \textsc{residual}(A_h,\ \langle s^h \rangle) \\
\langle s^h \rangle &\models \textsc{iterate}(\langle c^h \rangle,\ \omega,\ \langle \mathcal{P} \rangle) \mid \textsc{iterate}(\textsc{coarse-grid-correction}(P_{2h},\ \langle c^{2h} \rangle,\ \omega),\ 1,\ \lambda)\ \lambda) \\
\langle s^h \rangle &\models (u_0^h,\ f_0^h,\ \lambda,\ \lambda) \\
\langle B_h \rangle &\models \langle B_h \rangle + \langle B_h \rangle \mid \langle B_h \rangle \cdot \langle B_h \rangle \mid -\langle B_h \rangle \mid \langle D_h \rangle \mid A_h \\
\langle D_h \rangle &\models \langle D_h \rangle + \langle D_h \rangle \mid \langle D_h \rangle \cdot \langle D_h \rangle \mid -\langle D_h \rangle \mid \textsc{inverse}(\langle D_h \rangle) \mid \textsc{diagonal}(\langle B_h \rangle) \\
\langle c^{2h} \rangle &\models \textsc{apply}(\langle B_{2h} \rangle,\ \langle c^{2h} \rangle) \mid \textsc{coarse-cycle}(A_{2h},\ u_0^{2h},\ \textsc{apply}(R_h,\ \langle c^h \rangle)) \\
\langle s^{2h} \rangle &\models \textsc{iterate}(\langle c^{2h} \rangle,\ \omega,\ \langle \mathcal{P} \rangle) \mid \textsc{iterate}(\textsc{apply}(P_{4h},\ \langle c^{4h} \rangle),\ 1,\ \lambda) \\
\langle c^{4h} \rangle &\models \textsc{apply}(A_{4h}^{-1},\ \textsc{apply}(R_{2h},\ \langle c^{2h} \rangle)) \\
\langle \mathcal{P} \rangle &\models \text{red-black partitioning} \mid \lambda
\end{aligned}
$$

(a) Syntax

function ITERATE($(u,\ f,\ \delta,\ pred)$, ω, $\mathcal{P}$)
  $\tilde{u} \leftarrow u + \omega \cdot \delta$ with $\mathcal{P}$
  return $(\tilde{u},\ f,\ \lambda,\ pred)$
function APPLY(B, $(u,\ f,\ \delta,\ pred)$)
  $\tilde{\delta} \leftarrow B \cdot \delta$
  return $(u,\ f,\ \tilde{\delta},\ pred)$
function RESIDUAL(A, $(u,\ f,\ \lambda,\ pred)$)
  $\delta \leftarrow f - Au$
  return $(u,\ f,\ \delta,\ pred)$
function COARSE-CYCLE(A_H, u_0^H, $(u^h,\ f^h,\ \delta^H,\ pred^h)$)
  $u^H \leftarrow u_0^H$
  $f^H \leftarrow \delta^H$
  $\tilde{\delta}^H \leftarrow f^H - A_H u_0^H$
  $pred^H \leftarrow (u^h,\ f^h,\ \delta^H,\ pred^h)$
  return $(u^H,\ f^H,\ \tilde{\delta}^H,\ pred^H)$
function COARSE-GRID-CORRECTION(P_H, $(u^H,\ f^H,\ \delta^H,\ pred^H)$, ω)
  $(u^h,\ f^h,\ \delta^h,\ pred^h) \leftarrow pred^H$
  $\tilde{\delta}^h \leftarrow P_H \cdot (u^H + \omega \cdot \delta^H)$
  return $(u^h,\ f^h,\ \tilde{\delta}^h,\ pred^h)$

(b) Semantics

Fig. 2.1: A formal grammar for the generation of multigrid solvers with a hierarchy of three levels.

expressions by which a certain production symbol, denoted by $\langle \cdot \rangle$, can be replaced. To generate an expression, starting with the symbol $\langle S \rangle$, this process is recursively repeated until the produced expression contains only terminal symbols or the empty string λ. The construction of a multigrid solver comprises the recursive generation of cycles on multiple levels. Consequently, we must be able to create a new system of linear equations on a coarser level, including a new initial solution, right-hand side

and system matrix.[1] Moreover, if we decide to finish the computation on a certain level, we must be able to restore the state of the next finer level, i.e. the current solution and right-hand side, when applying the coarse grid correction. The current state of a multigrid solver on a certain level with grid spacing h is represented as a tuple (u^h, f^h, δ^h), where u^h represents the current iterate, f^h the right-hand side and δ^h a correction expression. To restore the current state on the next finer level, we additionally include a pointer *pred* to the corresponding state tuple. According to figure 2.1a the construction of a multigrid solver always ends when the tuple $(u_0^h, f_0^h, \lambda, \lambda)$ is reached. This tuple contains the initial solution and the right-hand side on the finest level and therefore corresponds to the original system of linear equations that we aim to solve. Here we have neither yet computed a correction, nor do we need to restore any state and hence both δ^h and *pred* contain the empty string. To conclude, it must be mentioned that this grammar comprises certain restrictions on the structure of the generated solver. Even though we can not preclude that it is possible to generate improved multigrid methods without these restrictions, this work only represents a first step towards the automatic generation and optimization of these methods and we do not claim to consider all possible variations, but instead focus on the classical multigrid formulation, as presented in [7, 3, 23]. Since we have shown how it is possible to generate expressions that represent arbitrary multigrid solvers using the formal grammar defined in figure 2.1, the remainder of this paper focuses on the evaluation and optimization of the resulting algorithms based on this representation.

3. Multi-Objective Optimization with Evolutionary Computation.

The fundamental requirement for the optimization of an iterative method is to have a way to evaluate both its rate of convergence and performance on a computer. As we want to fully automate this process, we must be able to perform all steps from the generation of a solver to its evaluation without requiring any human intervention. In general there are two possibilities to automatically evaluate an algorithm. Assuming there exists a code generator that is able to generate machine instructions from a high-level algorithm, one could first translate it to this representation, then employ the generator to emit an executable and finally run it to evaluate both objectives. The main disadvantage of this approach is that, depending on the runtime of the code generator, this can be time consuming. The second possibility is to use predictive models to obtain an approximation for both objectives in significantly less compute time. This work focuses on the automatic optimization of geometric multigrid solvers on rectangular grids. In this case, we can represent all matrices as one or multiple stencil codes and there exist models that allow us to predict the quality of a multigrid solver with respect to both objectives. Although, as it can not be expected that these predictions are always accurate, we still employ code generation to evaluate the best solver of each optimization run.

3.1. Convergence estimation.

The quality of an iterative method is first and foremost determined by its rate of convergence, i.e. the number of iterations that is required until the residual approaches zero. One iteration of an arbitrary multigrid solver can be expressed in the general form

$$u_{i+1}^h = M_h u_i^h + g^h, \tag{3.1}$$

[1] Note that we choose the initial solution $u_0^h = 0$ on all levels.

where M_h is the iteration matrix, u_i^h the solution vector in iteration i on the finest level, i.e. the current iterate, and g^h a vector that is obtained by transforming the right-hand side f^h. Essential for the convergence of stationary iterative methods is the spectral radius ρ of the iteration matrix M_h defined by

$$\rho(M_h) = \max_{1 \leq j \leq n} |\lambda_j(M_h)|, \tag{3.2}$$

where $\lambda_j(M_h)$ are the eigenvalues of M_h. Assume u_*^h is the exact solution of the system, the error $e_i^h = u_i^h - u_*^h$ in iteration i then satisfies,

$$e_i^h = M_h^i e_0^h, \tag{3.3}$$

where M_h^i is the ith power of M_h. The *convergence factor* of this sequence is the limit

$$\rho = \lim_{i \to \infty} \left(\frac{\left\| e_i^h \right\|}{\left\| e_0^h \right\|} \right)^{1/i}, \tag{3.4}$$

which is equal to the spectral radius of the iteration matrix M_h [19]. In general, the computation of the spectral radius is of complexity $\mathcal{O}(n^3)$ for $M_h \in \mathbb{R}^{n \times n}$. Although if we restrict ourselves to geometric multigrid solvers on rectangular grids, we can employ local Fourier analysis (LFA) to obtain an estimate for ρ [24]. LFA considers the original problem on an infinite grid while the boundary conditions are neglected. Recently LFA has been automated through the use of software packages [24, 18]. LFA Lab[2] is a library for the automatic local Fourier analysis of constant and periodic stencils [2] on rectangular grids. To automatically estimate the convergence factor of a multigrid solver using this tool, we first need to obtain the iteration matrix. Using the grammar described in the last section, we always generate expressions of the form of equation 2.6 from which we can extract the iteration matrix through a number of simple transformations. First we replace all occurrences of the initial solution u_0^h with the unit matrix and those of the right-hand side f_0^h with the zero matrix, then we remove all obsolete subexpression, i.e. the ones that become the zero matrix. Both can be accomplished in a single bottom up traversal of the corresponding expression tree. Finally, we transform the resulting expression, which represents the iteration matrix of our multigrid solver, to an LFA Lab expression, for which we can automatically estimate the spectral radius.

3.2. Performance estimation. A popular yet simple model for estimating the performance of an algorithm on modern computer architectures is the *roofline model* [25]. Based on the operational intensity of a compute kernel, i.e. the ratio of floating point operations to words loaded from and stored to memory, it gives an estimate for the maximum achievable performance, which is either limited by the memory bandwidth or the compute capabilities of the machine. The basic roofline formula is given by

$$P = \min(P_{max},\ I \cdot b_s), \tag{3.5}$$

where P is the attainable performance, P_{max} the peak performance of the machine, i.e. the maximum achievable amount of floating point operations per second, I the arithmetic intensity of the kernel and b_s the peak memory bandwidth, i.e. the amount

[2]LFA Lab: https://github.com/hrittich/lfa-lab

of words that can be moved from and to main memory per second. Within a geometric multigrid solver each kernel either represents a matrix-vector or vector-vector operation, where each vector corresponds to a rectangular grid and each matrix to one or multiple stencil codes. If we explicitly represent the stencil code of each operation, the computation of the operational intensity is straightforward.

3.3. Optimization. In case we want to find the optimal geometric multigrid solver for a certain problem, first the question about the size of the search space arises. With a sufficiently small search space one could attempt to simply enumerate all possible solutions. The infeasibility of this approach becomes obvious when looking at the grammar in figure 2.1. Assume our goal is to find a multigrid solver that operates on three levels, but the only allowed operation on the coarsest level is the application of a direct solver. Besides the start symbol $\langle S \rangle$ and the production resulting in the application of a direct solver on the coarsest level, each non-terminal symbol produces at least two alternatives. Now assume we perform on average twenty productions per level. This means we must consider more than 2^{40} alternatives that must be all evaluated with respect to both objectives, which is already infeasible on a standard desktop computer. In practice this number will be even larger, especially if we consider more levels. Furthermore, we need to choose a value for all occurrences of the relaxation parameter ω, which yields an additional continuous optimization problem. In case the search space is too large to be directly enumerated, a remedy is to use heuristics that aim to search efficiently through the space of possible solutions and still find the global or at least a local optimum. Evolutionary algorithms are a class of search heuristics inspired by the principle of natural evolution that have been successfully applied to numerous domains [13]. All of these methods have in common that they evolve a population of solutions (called individuals) through the iterative application of so-called genetic operators. The exact implementation of each genetic operator depends on the class of problem, i.e. the structure of the solution. Within this work we consider two different optimization problems. First of all, we want to find the list of productions that, according to the context-free grammar presented in section 2, leads to the optimal multigrid solver. The class of evolutionary algorithms that evolve expressions according to a context-free grammar are summarized under the term genetic programming [17, 12]. To evolve a Pareto front of multigrid solvers with respect to both objectives, we employ strongly typed genetic programming [16] with a non-dominated sorting based selection [4]. Because the computation of the spectral radius significantly slows down for expressions consisting of more than two levels, we split each optimization into multiple runs. Starting on the three coarsest levels, we perform the optimization assuming that we can obtain the correct solution on the coarsest level. During this process the value 1 is chosen for each relaxation factor ω that occurs within an expression. After we have evolved a Pareto front of multigrid expressions, we choose the best individual with respect to its estimated time-to-solution

$$T = \frac{\ln \varepsilon}{\ln \rho} \cdot t \tag{3.6}$$

among those with a spectral radius $\rho < 0.1$. Here t is the estimated runtime of one iteration and ε the error reduction until convergence, i.e. the ratio between the maximum tolerable and initial error, for which we choose $\varepsilon = 10^{-20}$. In the second step, we turn our attention to the list of relaxation factors in order to improve the convergence of the best individual evolved in the first step, which corresponds to a

single-objective continuous optimization problem[3], which we solve using a covariance matrix adaptation evolution strategy (CMA-ES) [9]. The resulting individual is then employed as direct solver for the performance estimation on the next two levels. We repeat this procedure until the optimization on the finest level is finished. By recursively deploying the best individual of a run as direct solver for the next run, a single multigrid expression is obtained which operates on the complete range of levels. We implement our optimization approach in the Python programming language using the framework DEAP [6] for the implementation of the evolutionary algorithms.[4]

3.4. Code Generation and Evaluation. In order to evaluate the solver whose components have been evolved within multiple stages of optimization according to the metric defined in equation 3.6, we employ the ExaStencils code generation framework [14], which was specifically designed for the generation of geometric multigrid implementations that run on parallel and distributed systems. To employ the code generation capabilities of this framework, we transform the evolved multigrid expression to an algorithmic representation, which we then emit in form of a specification in ExaStencil's DSL [20]. Based on this specification the framework generates a C++ implementation of the solver, including a default application, which we finally run to measure both its runtime $\tilde{t}$ and convergence factor

$$\tilde{\rho}_i = \frac{\left\| f^h - A_h u_i^h \right\|}{\left\| f^h - A_h u_{i-1}^h \right\|} \tag{3.7}$$

per iteration i on the target platform. We then obtain an approximate for the asymptotic convergence factor

$$\tilde{\rho} = \left(\prod_{i=1}^{n} \tilde{\rho}_i \right)^{1/n}, \tag{3.8}$$

where n is the number of iterations until convergence.

4. Evolving Solvers for Poisson's Equation. For our experiments we consider the steady-state heat equation with Dirichlet boundary conditions on a unit square, which is given by

$$\begin{aligned} -\boldsymbol{\nabla} \cdot (a \boldsymbol{\nabla} u) &= f \quad \text{in} \quad \Omega \,, \\ u &= g \quad \text{on} \quad \partial\Omega \,. \end{aligned} \tag{4.1}$$

where $\Omega = (0,1)^d$, $\boldsymbol{\nabla} \cdot v : \mathbb{R}^d \to \mathbb{R}$ is the divergence of v and $\boldsymbol{\nabla} u : \mathbb{R} \to \mathbb{R}^d$ is the gradient of u. The function $a : \mathbb{R}^d \to \mathbb{R}$ describes the thermal conductivity of the material. We discretize equation 4.1 using finite differences on a two-dimensional cartesian grid with a step size of h to obtain the system of linear equations $A_h u^h = f^h$. Our goal is to evolve optimal multigrid methods for solving this system. For this purpose we consider two different cases which are summarized in table 4.1. To obtain a Pareto front of multigrid expressions, we perform a multi-objective optimization for 100 generations using genetic programming (GP) with a $(\mu + \lambda)$ ES [1] with $\mu = \lambda = 1000$, an initial population of 10μ and the non-dominated sorting procedure presented in [5]. This means that in each generation we create λ individuals based

[3]The choice of these values will only affect the convergence of the method.
[4]EvoStencils: https://github.com/jonas-schmitt/evostencils

Table 4.1: Test cases

(a) Poisson 2D with constant coefficients

$$f(x,y) = \pi^2 \cos(\pi x) - 4\pi^2 \sin(2\pi y)$$
$$a(x,y) = 1$$
$$g(x,y) = \cos(\pi x) - \sin(2\pi y)$$

(b) Poisson 2D with variable coefficients

$$f(x,y) = 2\kappa((x - x^2) + (y - y^2))$$
$$a(x,y) = e^{\kappa(x-x^2)(y-y^2)}$$
$$g(x,y) = 1 - e^{(-\kappa)(x-x^2)(y-y^2)}$$

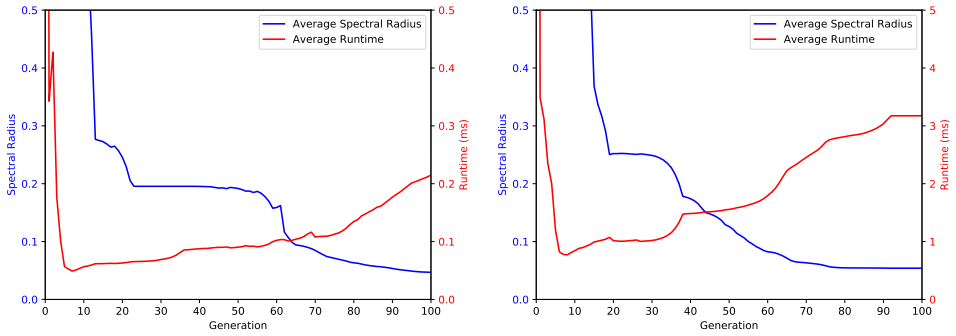
on an existing population of size μ and then select the best μ individuals for the next generation from the combined set. The fitness of each individual consists of two objectives, the spectral radius of its iteration matrix, estimated with LFA, and its runtime on the target platform, an Intel Xeon E3-1275 v5 (Skylake) machine with a clock frequency of 3.6 GHz, four cores and a peak memory bandwidth of $34,1$ GB/s, estimated with the roofline model. To estimate the spectral radius in the case of variable coefficients, we approximate the coefficient function with a constant stencil at the center of the domain. Individuals are selected for crossover and mutation using a dominance-based tournament selection as described in [5]. New individuals are created by either crossover with a probability of 0.7, whereby we employ single-point crossover with a probability of 0.2 to choose a terminal as crossover point or by mutation, through replacement of a certain subexpression with a new randomly created expression. To optimize the relaxation parameters of a multigrid solver, we employ a $(1 + \lambda)$ CMA-ES [10] with $\lambda = 50$ and 200 generations. Additionally, we restrict the set of productions for the generation of operator expressions to $\langle B_h \rangle \models$ INVERSE(DIAGONAL(A_h)), which means that we only allow Jacobi- or red-black Gauss-Seidel-type smoothers. Without this restriction we were not able to consistently predict the convergence behavior of the resulting multigrid methods in the given cases. The optimization is performed with a step size of $h = 1/2^l$ on each level l. For both cases we use a level range of $l \in [0, 8]$.[5] The results of the optimization for Poisson with variable coefficients are summarized in table 4.2. Each row contains the average and minimum of the spectral radius ρ, the runtime t and additionally the minimal time-to-solution T according to equation 3.6 before and after the optimization of the relaxation parameters ω. Figure 4.1 contains a plot of the average value of both objectives in the two GP-optimization runs on the four finest levels. Moreover, in table 4.3 we compare

Table 4.2: Optimization results for Poisson 2D with variable coefficient.

$l \in$	mean(ρ)	min(ρ)	mean(t)	min(t)	min(T)	min$_{1+\lambda}(T)$
$\{1,2\}$	8.09×10^{-3}	1.08×10^{-9}	2.11×10^{-3}	1.32×10^{-3}	5.27×10^{-3}	5.27×10^{-3}
$\{3,4\}$	1.21×10^{-2}	1.13×10^{-5}	1.35×10^{-2}	4.74×10^{-3}	6.41×10^{-2}	5.05×10^{-2}
$\{5,6\}$	4.68×10^{-2}	2.91×10^{-2}	0.215	4.62×10^{-2}	1.81	1.26
$\{7,8\}$	5.40×10^{-2}	4.01×10^{-2}	3.17	0.719	27.78	24.46

the predicted convergence rate ρ and runtime t for the evolved multigrid solver with the measured values of $\tilde{\rho}$ and $\tilde{t}$ when running the generated implementation. For

[5] This means that $l = 0$ is the coarsest possible level.



(a) $l \in \{5, 6\}$ (b) $l \in \{7, 8\}$

Fig. 4.1: Average Fitness for Poisson 2D with variable coefficients.

comparison, we also measure both metrics for a standard V-Cycle with two pre- and post-smoothing steps of red-black Gauss-Seidel (V22). In all cases we employ one thousand steps of red-black Gauss-Seidel as a direct solver on the coarsest grid. Even though the generated implementations do not achieve the predicted convergence rates and runtimes, our approach is able to evolve competitive multigrid solvers, whereby in case of Poisson with variable coefficients convergence is achieved faster than with a V22-cycle, although at the price of a slightly higher runtime. All evolved solvers are of type V-Cycle, though a variety of different smoothing steps is employed on each level.

Table 4.3: Measurement based evaluation.

Case	ρ	t	$\tilde{\rho}$	$\tilde{t}$	$\tilde{\rho}(V22)$	$\tilde{t}(V22)$
(a)	0.0405	1.69	0.144	10.71	0.0929	10.33
(b)	0.0405	1.70	0.168	27.15	0.186	22.53

5. Conclusion. In this work we have presented a novel approach for the automatic optimization of geometric multigrid methods based on a tailored context-free grammar for the generation of multigrid solvers and the use of evolutionary algorithms guided by a model-based prediction for the convergence and compute performance. Even though we have demonstrated by solving Poisson's equation that our approach in principle works, there is still room for improvement and extensions. Instead of considering a case that is well researched and for which efficient solution methods are available, a more challenging task would be the solution of partial differential equations where a functioning geometric multigrid solver has not been developed, which is for instance the case for many nonlinear PDEs. Furthermore, we aim to improve the accuracy of our model-based convergence prediction through an evaluation over the complete range of levels and of its compute performance through the use of the Execution-Cache-Memory (ECM) model [8]. Finally, one could consider different algorithms for the optimization of programs generated by our multigrid grammar, such as reinforcement learning [21] or Monte Carlo tree search [11] based techniques.

REFERENCES

[1] H.-G. Beyer and H.-P. Schwefel, *Evolution strategies–a comprehensive introduction*, Natural computing, 1 (2002), pp. 3–52.

[2] M. Bolten and H. Rittich, *Fourier analysis of periodic stencils in multigrid methods*, SIAM Journal on Scientific Computing, 40 (2018), pp. A1642–A1668.

[3] W. L. Briggs, S. F. McCormick, et al., *A multigrid tutorial*, vol. 72, Siam, 2000.

[4] C. A. C. Coello, G. B. Lamont, D. A. Van Veldhuizen, et al., *Evolutionary algorithms for solving multi-objective problems*, vol. 5, Springer, 2007.

[5] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, *A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii*, in International Conference on Parallel Problem Solving From Nature, Springer, 2000, pp. 849–858.

[6] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, *DEAP: Evolutionary algorithms made easy*, Journal of Machine Learning Research, 13 (2012), pp. 2171–2175.

[7] W. Hackbusch, *Multi-grid methods and applications*, vol. 4, Springer Science & Business Media, 2013.

[8] G. Hager, J. Treibig, J. Habich, and G. Wellein, *Exploring performance and power properties of modern multi-core chips via simple machine models*, Concurrency and Computation: Practice and Experience, 28 (2016), pp. 189–210.

[9] N. Hansen, S. D. Müller, and P. Koumoutsakos, *Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es)*, Evolutionary computation, 11 (2003), pp. 1–18.

[10] C. Igel, N. Hansen, and S. Roth, *Covariance matrix adaptation for multi-objective optimization*, Evolutionary computation, 15 (2007), pp. 1–28.

[11] L. Kocsis and C. Szepesvári, *Bandit based monte-carlo planning*, in European conference on machine learning, Springer, 2006, pp. 282–293.

[12] J. R. Koza, *Genetic programming as a means for programming computers by natural selection*, Statistics and computing, 4 (1994), pp. 87–112.

[13] J. R. Koza, *Human-competitive results produced by genetic programming*, Genetic Programming and Evolvable Machines, 11 (2010), pp. 251–284.

[14] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, et al., *Exastencils: advanced stencil-code engineering*, in European Conference on Parallel Processing, Springer, 2014, pp. 553–564.

[15] R. G. Mahmoodabadi and H. Köstler, *Genetic programming meets linear algebra: How genetic programming can be used to find improved iterative numerical methods*, in Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '17, New York, NY, USA, 2017, ACM, pp. 1403–1406.

[16] D. J. Montana, *Strongly typed genetic programming*, Evolutionary computation, 3 (1995), pp. 199–230.

[17] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming*, Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008, http://www.gp-field-guide.org.uk. (With contributions by J. R. Koza).

[18] H. Rittich, *Extending and Automating Fourier Analysis for Multigrid Methods*, PhD thesis, University of Wuppertal, June 2017.

[19] Y. Saad, *Iterative methods for sparse linear systems*, vol. 82, siam, 2003.

[20] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich, *Exaslang: a domain-specific language for highly scalable multigrid solvers*, in Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2014 Fourth International Workshop on, IEEE, 2014, pp. 42–51.

[21] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, MIT press, 2018.

[22] A. Thekale, T. Gradl, K. Klamroth, and U. Rüde, *Optimizing the number of multigrid cycles in the full multigrid algorithm*, Numerical Linear Algebra with Applications, 17 (2010), pp. 199–210.

[23] U. Trottenberg, C. W. Oosterlee, and A. Schuller, *Multigrid*, Elsevier, 2000.

[24] R. Wienands and W. Joppich, *Practical Fourier analysis for multigrid methods*, Chapman and Hall/CRC, 2004.

[25] S. Williams, A. Waterman, and D. Patterson, *Roofline: an insightful visual performance model for multicore architectures*, Communications of the ACM, 52 (2009), pp. 65–76.