TECHNISCHE UNIVERSITÄT DRESDEN

**Faculty of Computer Science** Center for Advancing Electronics Dresden, Chair for Compiler Construction

# Multi-Objective Autotuning Targeting a Domain Specific Language for Particle Simulations

Friedrich Michel

Matriculation number: 4084573

## Master Thesis

to achieve the academic degree

## Master of Science (M.Sc.)

First referee

Prof. Dr.-Ing. Jeronimo Castrillon

Second referee

Prof. Dr. sc. techn. Ivo F. Sbalzarini

Supervisor

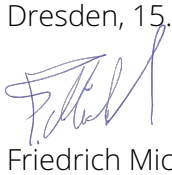
M.Sc. Nesrine Khouzami

Submitted on: 15.05.2020

# Statement of authorship

I hereby certify that I have authored this Master Thesis entitled *Multi-Objective Autotuning Targeting a Domain Specific Language for Particle Simulations* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 15.05.2020



Friedrich Michel

# Abstract

Simulations have become well established means in scientific research since they are capable of predicting the behavior for a variety of problems, e.g. fluid flows, diffusion of gases or chemical reactions. Particle methods in particular are able to simulate discrete as well as continuous models where continuous simulations require discretization. The particle methods domain-specific language (DSL) OpenPME allows to directly formulate simulations of continuous fields without explicit discretization. The choice of discretization schemes and their parameters traditionally required a lot of domain knowledge combined with tedious manual optimization effort. Choosing the right schemes is of utmost importance since it has a considerable impact on both accuracy and computational cost of the simulation. This work presents a multi-objective autotuning approach for numerical discretization schemes used in particle methods to automate this process. The proposed approach strives for optimization regarding the accuracy and computational cost of the target simulations while also considering numerical stability. The implementation of the autotuning approach involves efficient variant generation and means to measure the tuning objectives. The autotuning framework OpenTuner is used to implement general optimization techniques while domain knowledge is utilized to design model-based search and prediction approaches. The evaluations show the advantage of the model-based approaches where the search is consistently able to find highly performant configurations within 14 to 17 steps in a search space containing hundreds of thousands of configurations. The model-based prediction is able to predict similar performant configurations while conducting only a few swift initialization measurements with no further search steps required.

# Contents

# 1. Introduction

Autotuning systems have been used since the late 1990s to tune performance critical application kernels and libraries for new platforms and problems. They select the most desirable configuration out of large search spaces of possible implementations through models and empirical measurements. While the early systems mainly targeted algebra libraries, they have been used to optimize a wide variety of problems ever since. They are of particular importance in High-Performance Computing (HPC) environments.

Since the early 2000s, the computing power in HPC environments has exponentially increased due to the use of multi-core architectures with ever-growing numbers of cores. Such growth gave birth to a large number of programming models and tools to benefit from the prominent advancements.

There are classical parallel models like OpenMP's shared memory, MPI's message passing or Pthreads' thread model which are universal and hardware independent. Additionally, hardware specific languages are used such as CUDA for NVIDIA GPUs [22]. Furthermore, implicit parallel programming can be heavily utilized by functional programming languages that do not impose an explicit order of operations. Many scientific applications, which are greedy in terms of computing power, showed a pertinent need to exploit HPC systems. A known example of such applications are particle simulations. They are formulated in terms of interacting and evolving point-like objects called particles. Since the particles evolve individually over time, particle simulations are well suited for parallelization in HPC environments over potentially thousands of CPU cores.

However, using the aforementioned programming models and tools to efficiently develop these applications is a cumbersome task for domain experts who may lack specialized skills to implement highly scalable parallel applications. This problem is known as the knowledge gap [53].

Domain-specific languages (DSLs) have the potential of closing this knowledge gap by using high-level abstractions which are easily understood by domain experts on the one hand and by generating highly performant parallel code on the other hand. The OpenPME [40] DSL for instance allows the formulation of discrete and continuous particle-mesh simulations. It generates C++ code on top of the OpenFPM library [37].

OpenPME does not only use abstractions to ease the parallelization of simulations but also to describe continuous fields and the application of partial differential equations (PDEs) over them without explicit discretization. Still, the abstraction of continuous fields have to be discretized onto particles using concrete numerical methods. The choice of which method and other discretization parameters to use, has a substantial impact on the computational costs, accuracy and even stability of the resulting simulation.

The question investigated in this work is how to choose the best discretization method and its parameters given a specific simulation. There is no universally best choice that could be applied for all simulations since the requirements are problem specific. For a subclass of problems it has been shown that analytical solutions for some parameters are possible [57]. However, in the general case, a purely analytical solution is beyond reach [57]. This thesis presents a multi-objective autotuning approach which combines analytical knowledge with empirical measurements to find the best discretization method considering both accuracy and computational costs for continuous particle simulations written in OpenPME. After giving an overview of the general field of autotuning, optimization algorithms, particle methods and the DSL OpenPME in particular in Chapter 2, Chapter 3 will introduce two particle simulations that will be used throughout this work and identify the exact tuning opportunities. Chapter 4 and Chapter 5 describe the design and implementation of this thesis' autotuning approach including the applied general purpose and model-based domain specific optimization techniques. In Chapter 6 the optimization techniques are evaluated and the general quality and feasibility of the autotuning system is investigated. The work is concluded in Chapter 7 followed by an outlook on future work.

# 2. Background

This chapter will present a general overview over existing autotuning systems, describe a selection of general purpose optimization algorithms, introduce the basics of particle methods needed throughout this thesis and briefly discuss OpenPME.

## 2.1. Autotuning

Autotuning is the automatic generation of a search space of possible valid implementations combined with a search for the most suitable one. Suitability is often defined by runtime but may, as in this case, be dependent on multiple criteria. This is referred to as multi-objective autotuning. The automated search for the best implementation offers huge potential for performance improvements and relieves the programmer from a lot of manual optimization work. This section provides a general overview of different autotuning approaches and what decisions have to be made when implementing one.

### 2.1.1. Autotuning Systems

This section presents a selection of successful autotuning systems and briefly describes the areas they are applied in as well as their individual approaches.

ATLAS is an autotuning approach for linear algebra libraries from 1998. It is often regarded as one of the earliest applications of autotuning. The Application Programming Interface (API) in question was the Basic Linear Algebra Subprograms (BLAS) that contained a small number of widely used and performance critical routines like vectors and matrix operations. Usually, hardware vendors provided implementations which were optimized by hand for the specific machine architecture. The idea of Automatically Tuned Linear Algebra Software (ATLAS) [68] was to automate this process. A number of target architecture dependent optimizations are applied. These include loop unrolling (enough to decrease overhead but not too much so that the instruction cache does not

overflow), instruction reordering for latency hiding, adjusting blocking factors and loop reordering. Depending on the architecture properties such as cache size, register count and the number of floating point units, a search space of possible implementations is created. The search space is explored by testing each value of a single tuning parameter while leaving all others unchanged. This is done for all parameters and repeated multiple times. While this strategy does not guarantee to find the globally best value, it does find a local minimum.

FFTW is a widely used C library implementing discrete Fourier transformation. It adapts to different hardware and problems by conducting empirical measurements at runtime before executing the actual transformation. To ensure that the improvement is not overshadowed by the tuning cost, FFTW [28] provides multiple methods with different trade-offs between tuning time and result quality.

OSKI provides automatically tuned sparse matrix operations as sparse matrix-vector multiply and sparse triangular solve. The tuning is conducted at runtime to not only adapt to the specific hardware but also to the given matrix. OSKI [67, 66] exposes the decision of if and when to tune to the user to ensure that the overhead is amortized by the improved performance since the tuning can cost 40 times as much as a single sparse matrix operation. The literature introducing it focuses strongly on the interface for the user and does not describe the implementation of the actual optimization in detail.

CHiLL is a polyhedral loop transformation framework that is enhanced with automatic parameter tuning [20, 60]. The optimization utilizes a parallel simplex method and allows for user interaction and constraint specification.

PetaBricks is a language that allows to define multiple implementation variants for a single problem. To decide which variant to execute, PetaBricks [4] deploys autotuning techniques. In the optimization phase, smaller sub-problems are tuned using a genetic optimization approach. The best solutions of the sub-problems are then combined over multiple stages and optimized again until they form a global selection.

Orio tunes the performance of annotated C and C++ codes by performing source-to-source transformations as loop unrolling, loop tiling and loop permutation. Structured comments containing performance tuning directives are processed in a pre-compilation phase defining the search space. To explore the search space, Orio [33] implements exhaustive search, random search, the Nelder-Mead simplex method and simulated annealing. Each global optimization is followed by a greedy local search. The system is implemented in Python.

Apollo dynamically optimizes input sensitive C++ kernels at runtime. To achieve this, a decision tree classifier is trained offline, meaning before the actual execution. Using this pre-trained model, tuning decisions are made at runtime leading to performance improvements that were not possible with statically optimized code. Apollo [10] is implemented in Python using the scikit-learn [50] package to implement the decision tree.

### 2.1.2. Autotuning Frameworks

Autotuning frameworks provide the means for implementing autotuners for new languages, libraries and even single programs. They may provide tools to integrate autotuning directives directly into program code via annotations and automatically generate the search space from it. They also usually provide the implementation of an optimization algorithm or even a large collection to choose from. This section will briefly introduce Nitro, which allows the construction of input sensitive online autotuners. The two general purpose frameworks to build offline autotuners OpenTuner and ATF will be discussed in a bit more detail.

#### Nitro

Nitro is an autotuning framework that utilizes supervised learning in an offline training phase to generate an input sensitive model [46]. The autotuner itself is implemented in Python and deploys Support Vector Machines (SVMs) as means of supervised learning. The learned model is exported as C++ header file and the prediction is performed through the `libSVM` [19] library.

#### OpenTuner

OpenTuner [3] is a framework for building domain-specific, multi-objective autotuners and focuses on the optimization part of an autotuner. It provides the user with the search algorithm and leaves the integration and measurement to the user. This increases the flexibility while decreasing the ease of use, especially for single program autotuning, compared to solutions like ATF (2.1.2).

OpenTuner's core idea is to combine multiple search techniques (ensembles) by using them in conjunction. Those, that are performing better are assigned bigger parts of the tuning time. Results are shared between techniques via a common database.

The main benefit of this is that OpenTuner's resulting search technique is able to work in "search spaces [that] are much more complex, with discontinuities, high dimensionality, plateaus, hills with some of the configuration parameters strongly coupled and some others independent from each other" [3, p.304].

The problem of how much time should be allocated to which search algorithm is modeled by the multi-armed bandit problem. This problem is an instance of the exploration

versus exploitation dilemma. Each gambling machine (optimization technique in this case) has an unknown expectation of reward (improvement in this case). So an allocation strategy that maximizes the overall reward is sought [5]. The strategy used is the area under the curve credit assignment (AUC Bandit) meta technique with sliding window, where the sliding window ensures that only the most recent, and by that most significant history is considered. As area under the curve, OpenTuner essentially uses the number of new best values each technique found as proposed in [27].

Available search techniques in OpenTuner include differential evolution, Nelder-Mead search, Torczon hillclimbers, evolutionary mutation, pattern search and particle swarm optimization among others. The meta techniques may use a selection of these given techniques.

### ATF

The Auto-Tuning Framework (ATF) [52] is a generic directive-based approach to automatic program optimization. It emphasizes ease of use strongly and focuses on dependent tuning parameters.

ATF is used by annotating the source code with tuning directives. They define tuning parameters and their possible values, the cost function, the search technique and abort conditions to use. ATF parses this file and automatically generates the defined search space, does measurements and tunes the generated program variants according to the directives. This is especially useful when creating an autotuner for a single program as only the regular source file enriched with annotations is necessary.

Tuning parameters may define constraints that express dependencies between parameters. For example, one could define that one parameter always has to divide another one. In this way, ATF is able to define a search space more precisely.

As optimization techniques it provides exhaustive search, simulated annealing and OpenTuner search. To use OpenTuner's optimization, ATF has to work with the fact that its own search space is defined using constraints while OpenTuner does not support them. To handle this, ATF uses a range of integers as OpenTuner parameter and creates a mapping into its own search space. This mapping seems subpar for multi dimensional search spaces where relevant correlations between neighboring configurations exist, since those will be lost in the flattened mapping. When no dependent parameters are used, the search space of ATF is directly translated into the equivalent for OpenTuner.

It is demonstrated that ATF shows speedups over OpenTuner of up to a factor of 5.31 for a case of general matrix multiplication where the tuning parameters have a lot of interdependencies. Since OpenTuner can not inherently represent them, it is given the full search space where only one in $10^7$ configurations is valid. For cases without interdependencies, ATF and OpenTuner show equal performances as expected since their search spaces coincide and both use OpenTuner's optimization technique.

### 2.1.3. Differences in Autotuning Systems

Although the main goal of all autotuning systems is to find the best configuration automatically by evaluating the search space, they differ in many other aspects. This section categorizes the previously introduced approaches according to Balaprakash et al. [8].

#### Packaging

Many autotuners are packaged as library optimizations and aim at improving codes for specific hardware architectures to achieve performance portability. ATLAS [68] is an early example from this category tuning algebra routines like matrix and vector operations. OSKI [20, 60] tunes sparse matrix kernels for specific machines. The widely used library FFTW [28] optimizes the discrete Fourier transform (DFT) to adapt to changing hardware and problems at runtime.

Autotuners have also been integrated into compilers to aid optimization decisions. CHiLL [20, 60] tunes loops by combining empirical optimization techniques with compiler analysis to prune the search space. The language PetaBricks [4] is even build around the idea of defining multiple implementation variants of functions and letting the compiler decide for the best one.

The third way, autotuners might be packaged at application level. They are usually designed as tuning frameworks which can be applied by users to tune specific applications. Some of these frameworks, like CLTune [48], target specific areas working on user defined search spaces of OpenCL code. More general frameworks, as OpenTuner [3] and ATF [52], are generic regarding the programming language, application domain and tuning objective. These general frameworks can also be used to build compiler directed autotuners which will be discussed more in section 5.

#### Selection Approach

The variant selection approaches can roughly be divided into model-based, search-based and hybrid approaches.

Most autotuners go under the category of search-based approaches. A large set of search algorithms is applied ranging from exhaustive search to different local search algorithms such as gradient methods and global strategies like simulated annealing. The subsequent section presents a more detailed overview.

Model-based tuning reduces tuning cost by carefully designing an analytical model of the search space based on system and problem characteristics. The model is then enriched with empirical measurements. This can be done by conducting a small number of samplings in the beginning to fit the model to the actual problem and the target architecture and then using this model for the final prediction. The model could alternatively be used as a search method by predicting the best configuration according

to the current model, measuring this configuration and improving the model with the new measurement point. The biggest limitation of model-based approaches is that they require comprehensive domain knowledge and a lot of manual design decisions by the person implementing the autotuner. If the model is not general enough, it may also occur that the search space at hand does not fit the model which would render it unusable.

Hybrid approaches use some degree of modeling to narrow the search space using systems characteristics as done by ATLAS and then conduct a regular search on the remaining configurations.

#### Time of Application

The time of conducting the tuning varies a lot between different autotuners. The two overarching categories here are online and offline tuning referring to tuning before and during the execution of the target program respectively.

Offline tuning can be applied when a library or application is ported to a new architecture as done in ATLAS and CHiLL. In other cases, especially in case of application level tuning as in CLTune, ATF or OpenTuner, the tuning has to be repeated after each change of the program or at a time chosen by the user.

Online tuning can be accomplished in two manners. The first possibility is that general measurements are conducted offline to train an input sensitive predictive model which chooses the configuration to use. This approach is used by Apollo and Ding [23]. The second way is that actual empirical evaluations are performed at runtime as practiced by FFTW and OSKI. They are able to adapt more precisely to new problems which are unknown before runtime.

#### Integration into Application

Usually, the configuration found by the tuner is directly integrated into the final code either through compilation or linking. In the case of online tuning, different code variants might be created and an implemented selector chooses which one to use. If the tuning parameters are primitive values not needed at compile time, they might perfectly be integrated through a configuration file. The latter approach is also preferable if re-tuning should afterwards be an option as long as they have no relevant negative impact on the runtime.

## 2.2. Search and Optimization Algorithms

The words *search method* and *optimization method* are often used interchangeably in the field of objective function tuning. They refer to techniques that estimate the best element out of a set of possible configurations (search space). More generally, *search methods* also refer to a wider field of information retrieval techniques, e.g. the check

whether a text contains a specific word. This section will briefly discuss techniques that are widely used in the field of autotuning.

### Exhaustive Search

Exhaustive search is the technique of thoroughly measuring every configuration in the search space. It is the only technique that can guarantee to find the best configuration for arbitrary problems but also represents the slowest of optimization method. In many cases it is not feasible because either the search space is too large, the measurement of single configurations occupies substantial amounts of time or a combination of both.

### Random Search

Random search refers to various different search techniques that involve iterative selection of randomly chosen configurations in the search space. They differ in the question whether all configurations are eligible [11], or only those inside a hypersphere around the current best configuration [59]. The eligible configurations are either selected equally likely or according to a normal distribution around the best configuration [45].

### Simulated Annealing

Simulated annealing [1] is an optimization approach which tries to approximate the global optimum of a given problem. Its idea is inspired by the process of annealing metals. The search uses a decreasing *temperature* value for exploration decisions. High temperatures at the beginning allow large changes to the configuration and also to accept worse ones. As the temperature cools down over time the steps get smaller and the probability of choosing worse configurations decreases. This process enables simulated annealing to overcome local minima in the beginning and do local optimizations in the end. To avoid losing priorly found good configurations, simulated annealing can be enhanced with restarts which sets them back to the best configuration found earlier.

### Genetic Algorithm

Genetic algorithms [34] use optimization strategies based on Darwin's theory of evolution. A set of active configurations is maintained and used as base for operations like crossover and mutation to create a new generation. Using selection, the active configuration base is pruned. These means are repeated until a maximum number of generations or timeout is reached.

### Nelder-Mead

The Nelder-Mead [47] method approximates a local optimum in a n-dimensional search space. It uses a simplex[1] with $n + 1$ vertices. The vertices represent configurations. The search progresses using repeated movement of single vertices. Usually the worst vertex is moved either by reflection through the remaining points or by expansion or retractions of the simplex. Many improvements and variations of the method have been developed since it was proposed in 1965 [9, 41, 44, 29].

### Hill Climbing

Hill climbing describes the class of local optimization techniques that start with an arbitrary initial configuration and apply incremental changes. If a change leads to a better solution, this new configuration becomes the base for further changes. This process is repeated until a local optimum is found or a timeout is reached. It naturally will only reach a local optimum and not necessarily a global one for non convex problems. A countermeasure to this would be the use of repeated restarts [17].

### Gradient Descent

Gradient decent is a local optimization technique similar to hill climbing with the difference that it is only applicable for differentiable functions. In each step the configuration is altered in the direction of its negative gradient whereas hill climbing approaches accept any changes that lead to improvements. Hence, the gradient descent method is not, in contrast to all other algorithms discussed in this section, a direct search algorithm meaning it utilizes information about the gradient. However, this does not imply that the gradient has to be known analytically. It can also be estimated by measuring surrounding configurations.

### Bayesian Optimization

Bayesian search is based on the idea of building a probabilistic model of the objective function based on already observed data points. The most promising unobserved data point gets evaluated and refits the model [25]. This approach is especially promising when the evaluation of individual points takes a relatively long time as it has the potential to predict the optimal configuration early on, if the model closely resembles the reality.

[1] A simplex is a generalized triangle for arbitrary dimension.

## 2.3. Particle Methods

Nowadays, simulations belong to the fundamental assets in science alongside experiments and theory. Whenever a problem is too complex to analyze analytically, simulations enable predictions of the behavior for a wide spectrum of problems such as fluid flows, diffusion of gases or chemical reactions. Particle methods are numerical simulations schemes that are formulated as interacting and evolving particles [37]. This section will introduce the fundamentals of particle methods following the elaborations in [54].

A particle simulation consists of many simulation time steps during which the particles interact with each other and evolve their properties and positions. In theory, each particle $p \in P$ could interact with each other particle in each step, resulting in a time complexity of $\mathcal{O}(steps \cdot |P|^2)$.

```
for (step : steps) {
    for (p1 : particles) {
        for (p2 : particles) {
            p1.interact(p2);
        }
    }

    for (p : particles) {
        p1.evolve();
    }
}
```

However, in many cases it is only necessary to interact with particles that are nearby. This reduces the complexity to $\mathcal{O}(steps \cdot |P| \cdot \mathcal{N}_{max})$. Assuming that the neighborhood $\mathcal{N}(p)$ of each particle is significantly smaller than the total number of particles, this also leads to a substantial reduction in runtime.

To find particles in the neighborhood, many different solutions and data structures exist. Depending on the particle arrangement, e.g. (regular) grid-like or non grid-like formation, particles can be simply indexed by their position or may require more laborious approaches.

**Cell lists** One way to iterate particle neighborhoods is to use cell lists [2]. Cell lists are buckets arranged in a grid formation containing the particles. Each particle interacts with the particles in its own and neighboring cells (buckets). The cell size has to be chosen such that all relevant interaction partners are guaranteed to be in neighboring cells. Smaller cell sizes reduce the runtime, since the number of neighbors is defined by the number of particles per cell and the number of neighboring cells. The number of neighboring cells only depends on the dimensionality of the problem. It includes the own cell and is given by $3^{dim}$.

**Verlet lists** Another approach are Verlet lists [63]. In Verlet lists, the neighboring particles are stored as direct references for each particle. The resulting major disadvantage

is that moving particles may come in and out of the neighborhood. This leads to the necessity of rebuilding the Verlet lists each time step which would negate all advantages but can be counteracted to a certain extent by adding a margin. Now, the Verlet lists only have to be rebuild when particles moved far enough that they might have overcome this margin. Thus, they work well for slowly moving or only oscillating particles but still perform poorly for fast moving particles.

### 2.3.1. Discretization Methods

This thesis is focused on problems that are continuous in time as well as in space. To handle those cases, a discretization is necessary.

#### Temporal discretization

In the continuous-time case, the particles change according to

$$\frac{d\mathbf{x}_p(t)}{dt} = \mathbf{v}_p(t).$$

So they change their position $\mathbf{x}_p$ at each point in time $t$ according to their velocity $\mathbf{v}_p$ at that moment $t$.

$$\frac{d\omega_p(t)}{dt} = \mathbf{g}_p(t)$$

The properties $\omega$ change according to the property rate $\mathbf{g}$.

**Explicit Euler scheme** The explicit Euler scheme is one of the simplest time discretization schemes. To approximate a function of the form $\frac{dy}{dt}(t) = f(y(t))$, a finite time step size $\delta t = t_{n+1} - t_n$ is chosen, leading to

$$\frac{y(t_{n+1}) - y(t_n)}{\delta t} \approx f(y(t_n))$$

so in each time step, the new value of $y$ is given by

$$y(t_{n+1}) \approx y(t_n) + f(y(t_n)) \cdot \delta t.$$

Since the discretization is only an approximation, it introduces an error to the simulation. The smaller $\delta t$, the smaller the error. Explicit Euler is first-order accurate meaning that the doubling of $\delta t$ leads to the doubling of the error.

**Leapfrog scheme** The leapfrog scheme is an alternative time stepping method that provides a higher order of accuracy than explicit Euler. Here continuous time is approximated with

$$\frac{y(t_{n+1}) - y(t_{n-1})}{2\delta t} \approx f(y(t_n))$$

and new values of $y$ are given by

$$y(t_{n+1}) \approx y(t_{n-1}) + 2\delta t \cdot f(y(t_n)).$$

In contrast to explicit Euler, the Leapfrog scheme does not only depend on the current value of $y$ but also its previous one. This essentially doubles the memory requirements but in exchange it is second-order accurate meaning the error decreases with the square of the time step size for small enough $\delta t$. This is particularly beneficial for achieving high levels of accuracy while keeping the computational costs reasonably low.

**Runge-Kutta-4 and implicit Euler** In this work, only the explicit Euler scheme is used but there is a great variety of other methods. Runge-Kutta-4 divides each time step into four stages and achieves fourth-order accuracy.

All methods discussed thus far are explicit schemes meaning they only rely on the present and past states to calculate future ones. Implicit schemes like implicit Euler in turn include the future state's value for calculating the current one. This leads to a system of linear equations that has to be solved. While implicit Euler is still only first-order accurate it is unconditionally stable in contrast to all aforementioned schemes.

### Spatial discretization of differential operators

Many continuous particle simulations involve the calculation of differential operators over fields. The following methods allow to numerically solve them over discretized fields on particles.

**SPH** Consider spatial differential operators of the form

$$D^\beta = \frac{\partial^{|\beta|}}{\partial x_1^{\beta_1} \partial x_2^{\beta_2} \cdots \partial x_d^{\beta_d}}$$

where $d$ is the number of dimensions in space and $\beta$ defines the derivative in each dimension.

Using smoothed particle hydrodynamics (SPH) [42], the differential operator applied to a function $f$ describing the field is approximated with

$$D^\beta f(\mathbf{x}) \approx \sum_p f(\mathbf{x_p}) \cdot [D^\beta W_\epsilon](\mathbf{x} - \mathbf{x}_p) \cdot V_p.$$

$[D^\beta W_\epsilon]$ is the differential operator applied to the smoothing kernel $W_\epsilon$. $V_p$ represents the volume of interacting particles. A typical smoothing kernel is the Gaussian kernel

$$W_\epsilon(z) = \frac{1}{\epsilon\sqrt{2\pi}} \cdot e^{\frac{-z^2}{2\epsilon^2}}$$

where $\epsilon$ is the kernel width.

**PSE** The discretization method particle strength exchange (PSE) [24] uses symmetric particle interaction meaning that quantities are always exchanged between particles, hence the name. The differential operator $D^\beta$ applied to $f$ is approximated by

$$D^\beta f(\mathbf{x}) \approx \frac{V_p}{\epsilon^{|\beta|}} \cdot \sum_p (f(\mathbf{x}_p) \pm f(\mathbf{x})) \cdot \eta_\epsilon^\beta(\mathbf{x} - \mathbf{x}_p).$$

In contrast to SPH, PSE requires a different kernel $\eta_\epsilon$ for each differential operator. Gaussian kernels for the Laplacian operator can be found in Section 4.2.1.

**DC-PSE** The discretization method discretization corrected PSE (DC-PSE) [58, 13] generally works similar to PSE with the difference that the kernel $\eta$ now is not fixed anymore but has to be calculated for each particle separately. The general scheme is given by

$$\mathbf{D}^{m,n} f(x_p) = \frac{1}{\epsilon(x_p)^{m+n}} \sum_{x_q \in N(x_p)} (f(x_q) \pm f(x_p)) \eta\left(\frac{x_p - x_q}{\epsilon(x_p)}\right). \tag{2.1}$$

To generate a kernel, Bourantas et al. [13] proposed the following template:

$$\eta(\mathbf{x_p}) \begin{cases} \sum_{i,j}^{i+j<r+m+n} a_{i,j} x^i y^j e^{-x^2-y^2} & \sqrt{x^2+y^2} < r_c \\ 0 & otherwise \end{cases} \tag{2.2}$$

In this template, the coefficients $a_{i,j}$ have to be determined. The vector $\mathbf{a}(\mathbf{x}_p)$ puts them in an arbitrary but fixed order. The vector $\mathbf{p}(\mathbf{x}_p)$ contains the corresponding monomial basis such that if $a_k = a_{i,j}$ then $p_k = x^i y^j$. Both are of size $l = \frac{(r+m+n)(r+m+n+1)}{2}$ and the number of neighboring particles $k$ has at least the value of $l$. The value $r$ allows to specify the convergence rate. Since it directly influences $l$ (which defines the size of the following matrices and vectors), it also has a substantial effect on the computational cost. Vector $\mathbf{a}(\mathbf{x}_p)$ is given by the equation

$$\mathbf{A}(\mathbf{x}_p)\mathbf{a}(\mathbf{x}_p)^T = \mathbf{b} \tag{2.3}$$

where

$$\mathbf{A}(\mathbf{x}_p) = \mathbf{B}(\mathbf{x}_p)^T \mathbf{B}(\mathbf{x}_p) \in \mathbb{R}^{lxl} \tag{2.4}$$

$$\mathbf{B}(\mathbf{x}_p) = \mathbf{E}(\mathbf{x}_p)^T \mathbf{V}(\mathbf{x}_p) \in \mathbb{R}^{kxl} \tag{2.5}$$

$$\mathbf{b} = (-1)^{m+n} D^{m,n} \mathbf{p}(\mathbf{x})|_{x=0} \in \mathbb{R}^{lx1}. \tag{2.6}$$

Since $D^{m,n} p_i(\mathbf{x})|_{x=0}$ equals zero unless $p_i(\mathbf{x}) = x^m y^n$, the equation 2.6 essentially states

$$b_i = \begin{cases} (-1)^{m+n} \cdot m! \cdot n! & p_i(x) = x^m y^n \\ 0 & otherwise \end{cases}. \tag{2.7}$$

$\mathbf{V}(\mathbf{x}_p)$ is the Vandermonde matrix containing the monomial basis evaluated over the neighboring particles and $\mathbf{E}(\mathbf{x}_p)$ a diagonal matrix, meaning all entries outside the main diagonal are all zero.

$$\mathbf{V}(\mathbf{x}_p) = \begin{pmatrix} p_1\left(\frac{\mathbf{x}_p - \mathbf{x}_1}{\epsilon(\mathbf{x}_p)}\right) & p_2\left(\frac{\mathbf{x}_p - \mathbf{x}_1}{\epsilon(\mathbf{x}_p)}\right) & \dots & p_l\left(\frac{\mathbf{x}_p - \mathbf{x}_1}{\epsilon(\mathbf{x}_p)}\right) \\ p_1\left(\frac{\mathbf{x}_p - \mathbf{x}_2}{\epsilon(\mathbf{x}_p)}\right) & p_2\left(\frac{\mathbf{x}_p - \mathbf{x}_2}{\epsilon(\mathbf{x}_p)}\right) & \dots & p_l\left(\frac{\mathbf{x}_p - \mathbf{x}_2}{\epsilon(\mathbf{x}_p)}\right) \\ \vdots & \vdots & \ddots & \vdots \\ p_1\left(\frac{\mathbf{x}_p - \mathbf{x}_k}{\epsilon(\mathbf{x}_p)}\right) & p_2\left(\frac{\mathbf{x}_p - \mathbf{x}_k}{\epsilon(\mathbf{x}_p)}\right) & \dots & p_l\left(\frac{\mathbf{x}_p - \mathbf{x}_k}{\epsilon(\mathbf{x}_p)}\right) \end{pmatrix} \in \mathbb{R}^{lxl} \tag{2.8}$$

$$\mathbf{E}(\mathbf{x}_p) = diag\left(\left\{ e^{\frac{-|\mathbf{x}_p - \mathbf{x}_q|^2}{2 \cdot \epsilon(\mathbf{x}_p)^2}} \right\}_{q=1}^{k}\right) \in \mathbb{R}^{kxk} \tag{2.9}$$

Using equations 2.4 to 2.9, $\mathbf{A}(\mathbf{x}_p)$ and $\mathbf{b}$ are well defined. This means, that equation 2.3 can be solved for $\mathbf{a}(\mathbf{x}_p)$ using LU decomposition. While $\mathbf{a}(\mathbf{x}_p)$ has to be recomputed at each time step for moving particles, it can be re-used if particles are not moving or are re-meshed each step. As stated before, the convergence rate $r$ is an open parameter which greatly influences the computational cost but is worthwhile for smaller particle distances $h$.

### 2.3.2. OpenFPM

OpenFPM [37] is a C++ library targeting the implementation of scalable particle and hybrid particle-mesh simulations in HPC environments. It provides abstractions that allow to spread particles over processors with distributed memory, enable transparent cross-processor particle interactions and offers methods for dynamic load balancing and communication. The data structures can be used flexibly for different dimensionalities of space and floating-point precisions of particle positions and properties due to the heavy use of template parameters.

The two main data structures `vector_dist<...>` and `grid_dist_id<...>` represent particle sets and meshes respectively. Both are distributed over multiple processors transparently for the user. This is done by domain decomposition and assignment of the subdomains to processors. Each processor only stores and computes the interactions for its assigned particles. For particles to be able to interact across those domain borders, the sub-domains are extended by a *ghost layer* containing all particles within

interaction radius of the sub-domain particles. Communications between processors to keep the *ghost layers* updated are implemented using non-blocking MPI communication.

Furthermore, OpenFPM provides implementations of the neighborhood lists `VerletList<...>` and `CellList<...>` to allow neighborhood iteration.

Additionally, OpenFPM provides an ever-growing collection of numerical solvers frequently used in particle methods. This includes an implementation of the finite difference method but currently no general interface for the spatial discretization methods SPH, PSE and DC-PSE. An interface for DC-PSE is currently in the process of being implemented but has not made it into the official release [35].

## 2.4. OpenPME

Even though OpenFPM partly bridges the knowledge gap, it is still not easy for domain experts to handle its complexity and to understand error messages caused by incorrect use of template parameters. Those problems can be solved by using a DSL. OpenPME [40] forms an intermediate layer between the user and the particle library. It provides an user-friendly interface that should be intuitively comprehensible for domain experts. Error messages could express problems in a intelligible manner, e.g. by informing about the mismatching use of spatial dimensionality rather than cryptic C++ template errors. Additionally, it has the potential of warning the user about problems that do not involve semantic incorrectness but will likely lead to unintended behavior as missing or duplicated position and property updates during the same time step or the use of unreasonably small or large neighborhood radii.

OpenPME is the successor of the Parallel Particle-Mesh Environment (PPME) [38] which uses similar abstractions but generates Parallel Particle-Mesh (PPM) [6] code. Parallel Particle-Mesh Language (PPML) in turn is a particle language embedded into Fortran2003 and uses the PPM library [56]. As PPME, OpenPME uses Meta Programming System (MPS) [18] to build a language workbench providing advanced features such as code highlighting and completion.

OpenPME programs are structured in three phases: *initialization*, *simulation* and *visualization*. In the *initialization* phase, all properties and parameters of the simulation are defined. This includes dimensionality of space, shape and size of the simulation domain and boundary conditions. Additionally, global variables like physical constants can be defined in this phase. During *simulation* phase particle and grid points are initialized and evolve in the *time loop*. The *visualization* phase specifies how to handle the results of the simulation. A typical way would be to save the particle positions and properties in VTK files that can e.g. be imported into Paraview [7] for data analysis and visualization.

The *time loop* describing the diffusion of a continuous field $u$ in OpenPME would be expressed as follows.

```
time loop
    start: 0 dt: 0.1 stop: 5000
    ode method: explicit_euler
        ∂u/∂t = D · ∇²u
```

It is clearly visible that the expressions are on a very high level and describe how the field $u$ evolves using a PDE. However, this leaves open how it should be calculated on the discretized field. The code only hints that for temporal discretization the *explicit Euler* scheme should be used with a time step size of $\delta t = 0.1$. It does not specify, though, which spatial discretization scheme should be used, e.g. whether a SPH with a gaussian kernel of width $\epsilon = h$ or a DC-PSE with a polynomial kernel of some other width should be used. Currently, a default choice is made which is anything but ideal. The choice of discretization method and its properties has a significant influence on the resulting accuracy and computational cost of the simulation. This problem could be approached by providing the user with the possibility to specify those information as he currently already specifies the temporal discretization scheme. This however would increase the burden on the user and would most likely lead to trial and error approaches until a fitting configuration was found. For this reason, autotuning approaches are applied in this work.

# 3. Case Studies and Identification of Autotuning Opportunities

This chapter examines the diffusion and Gray-Scott reaction-diffusion system simulations. It introduces their general idea, illustrates how they would be implemented in OpenPME and defines the specific implementation characteristics that will be used in the rest of this thesis. OpenPME is currently in the early stages of development and the code generation not fully implemented. Due to this, the actual OpenFPM code which later will be generated for the following simulations, is written by hand for the purposes of this thesis.

## 3.1. Case Study: Diffusion

The first example is the diffusion of a field value $u$. This field could for example represent the temperature change in a material (heat transfer) or the change of concentration in a gas or fluid (mass transfer). Particles do not represent individual molecules here but they are sampling the field.

Fick's second law of diffusion states how the concentration changes with respect to time:

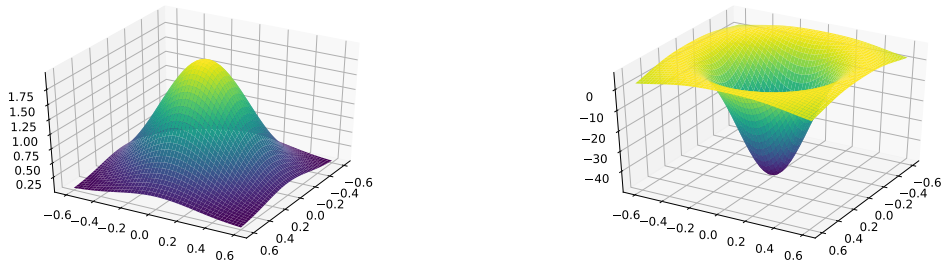
$$\frac{\partial u}{\partial t} = D\nabla^2 u$$

where D is the diffusion coefficient and $\nabla^2 = \triangle$ is the Laplace operator which is a generalization of the second derivative into the $n$-dimensional space. For 2D it equals

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

Figure 3.3 shows the OpenPME code that would generate a diffusion simulation. Lines 5 and 6 define the number of dimensions and the size of the simulation area. Line 7 states the boundary conditions for the simulation's border areas as periodic. Thereby,

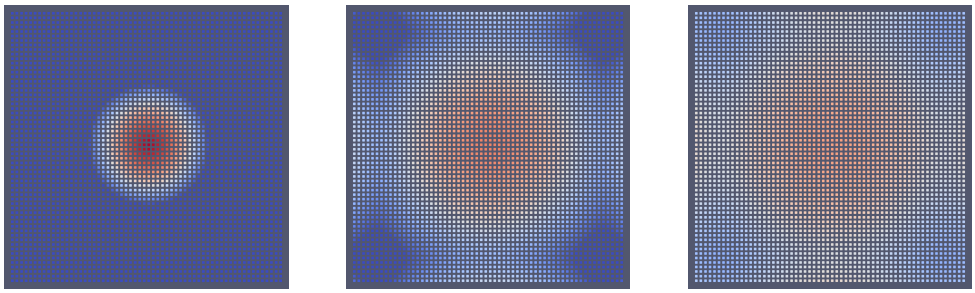
periodic indicates that the top and bottom as well as left and right borders are connected to each other so that particles near the borders interact with particles on the opposite side. Thus, the simulation essentially takes place on the surface of a torus. The uniform conditions in line 8 define that the particles are initialized on a regular Cartesian grid inside the domain. The subsequent parameters such as the number of particles, the spacial discretization method (SPH, PSE or DC-PSE including a corresponding kernel), the cutoff radius and the kernel width (epsilon) are subjects to tuning (lines 9-11). Lines 18-20 initialize the property *u* for all particles according to their position such that the field describes a Gaussian bell curve as shown in Figure 3.1 (a). Lines 23-27 describe the main time loop of the simulation in which the field *u* evolves according to the given PDE. The visualization phase (lines 32-33) defines the output of the simulation which is shown in Figure 3.2.



(a) Gaussian (b) Laplacian of Gaussian

Figure 3.1.: Initial condition used in the diffusion simulation along with its Laplacian.

The value of $\nabla^2 u$ at $t = 0$ is shown in Figure 3.1 (b). Its interpretation is quite intuitive since at the point of highest concentration, the largest negative change is shown which would be expected for diffusion.



(a) $t = 0$ (b) $t = 2500$ (c) $t = 5000$

Figure 3.2.: Diffusion simulation with $50^2$ particles on a regular Cartesian grid. The quantity *u* is visualized by the color on a logarithmic scale.

```
module Diffusion

    initialization

        dimension: 2
        domain_size: box(( 0 , 0 , 0 ),( 5 , 5 , 5))
        boundary_conditions: periodic
        initial_conditions: uniform
        num_particles = tune
        cutoff_radius = tune
        epsilon = tune
        D = 0.0001

    simulation

        type of simulation: continuous
            // initialize particle values
            foreach particle p
                distance_squared = (2.5 − p.x)^2 + (2.5 − p.y)^2
                p.u = 1 / (0.16 * π) * exp(-distance_squared / 0.16)

            // main timestepping
            time loop
                start: 0 dt: tune stop: 5000
                temporal ode method: explicit_euler
                spatial pde method: tune
                    ∂u/∂t = D * ∇^2 u


    visualization

        visualize particles:
            output file: "diffusion"

end module
```

Figure 3.3.: OpenPME code for diffusion simulation.

## 3.2. Case Study: Gray-Scott Reaction-Diffusion System

The second considered case study is the Gray-Scott reaction-diffusion system [31, 49]. While it may model a variety of physical phenomena, it is most commonly used to simulate two chemicals reacting with each other and diffusing individually. The system is described by the following two PDEs:

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u - uv^2 + F(1-u)$$
$$\frac{\partial v}{\partial t} = D_v \nabla^2 v + uv^2 - (F+k)v.$$

$D_u$ and $D_v$ are the diffusion constants of the chemicals $u$ and $v$. The constants $F$ and $k$ are additional parameters that determine properties of the system and by that the pattern that is formed. Figure 3.8 shows the OpenPME code that implements the Gray-Scott system. It reflects an abstraction of the Gray-Scott implementation in OpenFPM [37, 36].

The 2D implementation in OpenFPM uses a square plateau as initial condition as shown in Figure 3.4. While this is acceptable for low spatial resolutions, the used discretization schemes assume sufficiently smooth fields. For the Laplace operator, this means that the field has to be two times continuously differentiable which is not the case for the discontinuous plateau as initial condition.

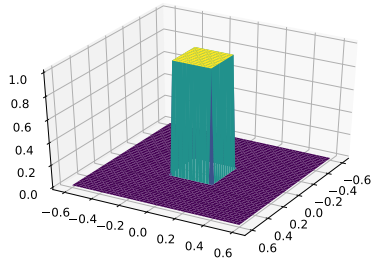
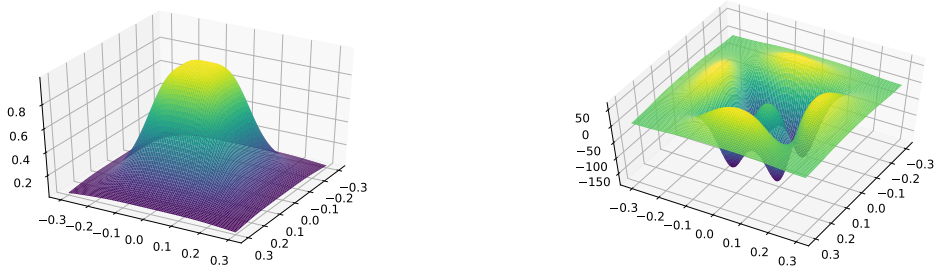



Figure 3.4.: Square plateau used as initial condition in OpenFPM.

For this reason, a smoothed square plateau function is used.

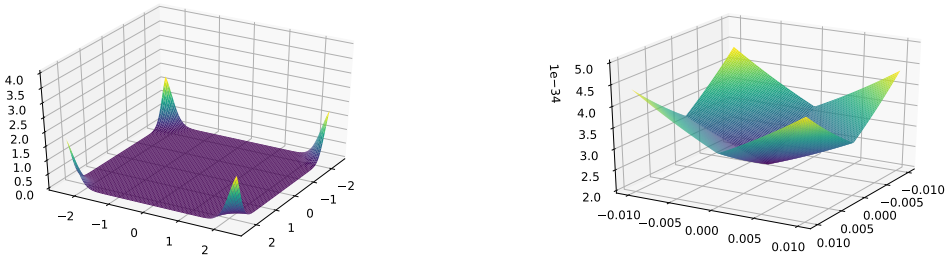
$$q(x,y) = \frac{1}{\frac{x^n+y^n}{r^n}+1}$$

The exponent $n$ specifies the steepness. For $n \to \infty$ the function approaches the square plateau. By $2r$ the side length of the square is defined and adjusted. The simulation uses $n = 4$ and $r = 0.15$ and adjusts the height with a prefactor. The smoothed square plateau function with these parameters is shown in Figure 3.5 next to its Laplacian.




(a) Smoothed square plateau (b) Laplacian

Figure 3.5.: Smoothed square plateau with its corresponding Laplacian. It is used as initial condition in the OpenPME Gray-Scott simulation.

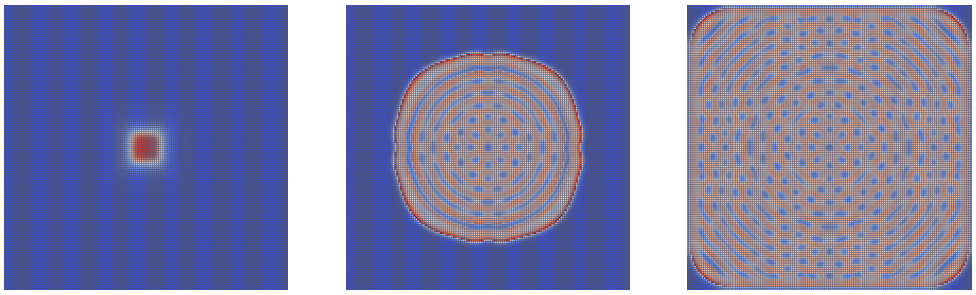



(a) Far (b) Near

Figure 3.6.: Discontinuity at the borders due to connected edges. It is visualized by moving the actual edges into the center and vice versa.

Due to the continuous boundary conditions, the field is not smooth along the edges as visualized in Figure 3.6. Those incontinueties are negligibly tiny though.

The initialization of the particles is implemented in lines 22-25. As in the diffusion simulation, this is followed by the main time loop (lines 28-33) in which the particles evolve according to the previously defined systems of equations. In the visualization phase in the end is responsible for generating an output as displayed in Figure 3.7.



(a) $t = 0$ (b) $t = 5000$ (c) $t = 10000$

Figure 3.7.: Gray-Scott simulation with $150^2$ particles on a regular Cartesian grid. The quantity $v$ is visualized by the color on a linear scale.

```
module GrayScott

    initialization

        dimension: 2
        domain_size: box(( 0 , 0 , 0 ),( 5 , 5 , 5))
        boundary_conditions: periodic
        initial_conditions: uniform
        num_particles = tune
        cutoff_radius = tune
        epsilon = tune
        Du = 0.0001
        Dv = 0.0001
        F = 0.055
        k = 0.03


    simulation

        type of simulation: continuous
            // initialize particle values
            foreach particle p
                distance4 = (2.5 − p.x)^4 + (2.5 − p.y)^4
                p.u = 1 - 0.5 / ((distance4 / 0,00050625) + 1)
                p.v = 0.25 / ((distance4 / 0,00050625) + 1)

            // main timestepping
            time loop
                start: 0 dt: tune stop: 10000
                temporal ode method: explicit_euler
                spatial pde method: tune
                    ∂u/∂t = Du * ∇²u - u * v² + F * (1 - u)
                    ∂v/∂t = Du * ∇²u + u * v² - v * (F + k)


    visualization

        visualize particles:
            output file: "gray_scott"

end module
```

Figure 3.8.: OpenPME code for Gray-Scott reaction-diffusion simulation.

## 3.3. Tuning Opportunities

The aim of this work is the autotuning of the (spatial) discretization method. Therefor, the choice between SPH, PSE and DC-PSE is considered. Changing the discretization scheme also includes changing the smoothing kernel and its width $\epsilon$. That is because the smoothing kernels are method specific, although there might be similar ones for all of them. Specifying $\epsilon$ regardlessly of both kernel and method would ignore the differences among the methods, and defining a fixed $\epsilon$ per spatial discretization scheme would make adaptions to higher or lower target accuracies impossible.

The spatial discretization scheme and smoothing kernel influence the order of error convergence and thus, the best choice for the spatial resolution $h$ for a given target accuracy. Therefore, it would be beneficial to also tune it in conjunction with the aforementioned parameters. Moreover, including the temporal resolution $\delta t$ as tuning parameter enables full control over the achieved accuracy and computational cost.

All the mentioned parameters (discretization scheme, smoothing kernel, $\epsilon$, $h$ and $\delta t$) have in common that they do not define what a continuous simulation represents but only its accuracy, stability and computational costs. Contrary to this, for discrete simulations a change of the spatial resolution, and by that the number of particles, would alter the problem. The same is true for the temporal resolution.

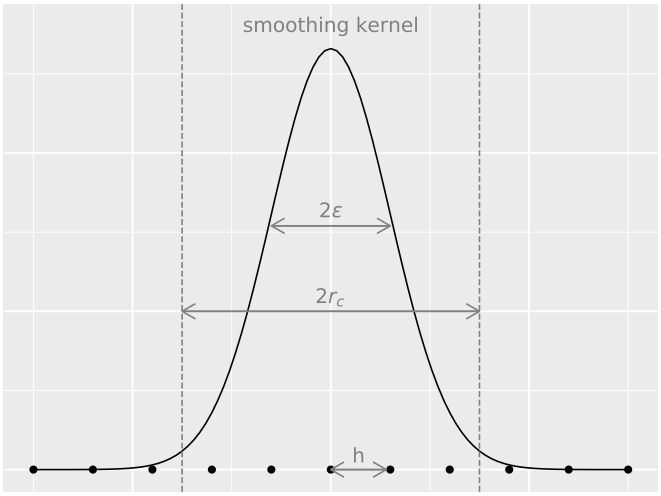



Figure 3.9.: Visualization of tuning parameters. The vertices at the bottom represent particles in 1D with inter-particle distance $h$. The function shows a Gaussian smoothing kernel with width $\epsilon$. The vertical bars show the neighborhood radius. The influence of the particles beyond is truncated.

This work is limited to the aforementioned tuning parameters. Nonetheless, it would also be possible to integrate others as long as they do not alter the basic characteristic of the simulation defined by the programmer. Possible additions will be discussed in the Section 7.1.3.

# 4. Autotuning Numerical Discretization Schemes

This chapter introduces the autotuning approach developed in this work. It presents and justifies the design decision regarding variant generation, accuracy and runtime measurement and reduction of the multi-objective problem into single-objective one. The optimization techniques used to tune for this objective are presented in the following chapter.

## 4.1. General Autotuning Approach

In Section 2.1.3, an overview of autotuning systems in the literature was presented and also highlighted the differences in their approaches. This section classifies the chosen system according to that overview and thus describes the general structure of the autotuner as shown in Figure 4.1.

Since this autotuner will be integrated into OpenPME, it is packaged as a compiler-directed tuner. However, this does not imply that the tuning will necessarily be applied at compile time but rather that the compiler will provide necessary hints to carry it out. It will generate the search space optionally defined by the user in the OpenPME code. If not, a reasonable default range for the different tuning parameters will be assumed. The compiler will provide the means to generate the code variants in the search space. Additionally, it will insert the necessary measurement facilities during code generation and generate ultimately the final version.

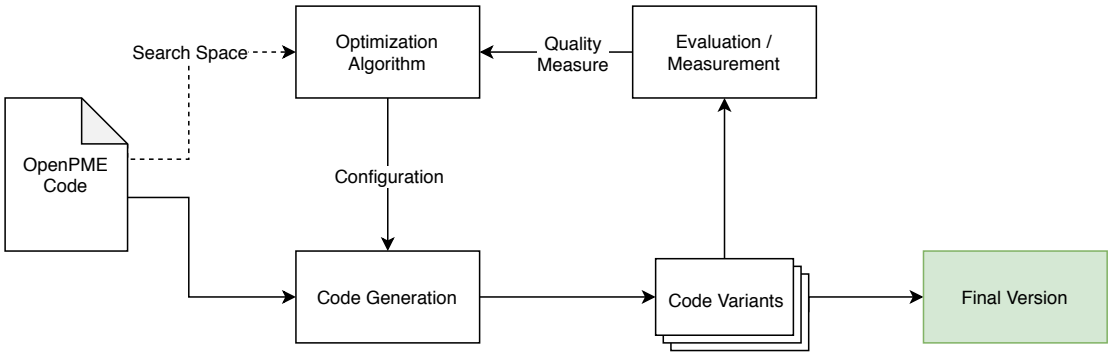



Figure 4.1.: General structure and components of the implemented autotuner.

## 4.2. Variant Generation

In Section 3.3, the open parameters affecting only the tuning goals but not the general purpose of the simulation are identified. A concrete instantiation of all parameters is called a *configuration* of the simulation. An optimization algorithm trying to find the best configuration must be able to generate code for all configurations. Thereby, numerical parameters such as the kernel width $\epsilon$, the number of particles and size of the neighborhood can be easily translated into the target code. However, parameters like the discretization method including its kernel are more complicated. For SPH and PSE this involves iterating over the neighborhood of all particles and accumulating a value according to the fitting kernel version, to the number of dimensions of the simulation and the order of the differential operator. For DC-PSE this even involves building up a system of linear equations and solving it for each particle.

Currently, OpenPME is not able to generate code for an arbitrary discretization method, kernel function, dimensionality of space and differential operator. The predecessor PPME would in case of diffusion generate code as shown in Figure 4.2. In line 1, the discretized differential operator is defined to be a Laplacian ($\nabla^2$) over the particles `c` using second order accurate DC-PSE with a ratio $c = \frac{h}{\epsilon} = 1$. As the decision for this discretization method is not defined in the PPME code and no autotuning is conducted, it is a static choice. In line 4, this discretized operator is applied to the field *U* which is discretized over the particles in *c*. The result is saved on each particle and can now be accessed in line 7 as `dU_p`.

Since OpenPME is compiling down to C++ and linking against OpenFPM, these constructs from PPML are not available. At this point, OpenFPM does not provide comparable constructs but as stated before it is not a trivial task to simply generate the whole code for the discretization method directly from OpenPME. This is the reason why a new interface has to be created providing comparable ease of code generation to the PPML constructs.

PPME

```
1  deqn method "rk4" on c
2      ∂c → U / ∂t  = constDU * ∇²c → U
3  end deqn
```

PPML

```
1  W = discretize_op(Lap, c, ppm_param_op_dcpse, [order=>2,c =>1.0_mk])
2  ...
3  get_fields(dU)
4  dU = apply_op(W, U)
5
6  foreach p in particles(parts) with sca_fields(U, V, dU)
7      dU_p = D_u * dU_p
8  end foreach
```

Figure 4.2.: Code generation for PDEs in PPME. Adapted from [39].

This task is fulfilled by the new C++ class `Differential_Operator` which takes as template arguments the dimensionality of the space, the datatype used for particle position, the order of the differential operator to be used, the discretization method and its kernel that should be utilized. Different ways of how it could be instantiated for diffusion in two dimensional space are shown in Figure 4.3.

```
Differential_Operator<2, double, 2, OP_SPH, KERN_GAUSSIAN> sph_operator(eps);
Differential_Operator<2, double, 2, OP_PSE, KERN_GAUSSIAN> pse_operator(eps);
Differential_Operator<2, double, 2, OP_DCPSE, KERN_POLY> dcpse_operator(eps);
```

Figure 4.3.: Instantiations of the `Differential_Operator` class for SPH, PSE and DC-PSE

As shown in Figure 4.4 line 3, the class provides the method `apply_op` which does, as the identically named PPML function, iterate over all particles in the distributed vector `vd` and calculates the result of the discretized differential operator over the property given by the template parameter `Diff_U`. The result is saved in the property given by `Lap_U`. This value then can be used as any scalar value in the main loop (line 7-18) where it is accessed in line 12. Line 13 reassembles line 2 of the OpenPME code with $\nabla^2 u$ now available as simple number and temporal discretization handled via Euler time stepping as the factor $\delta t$ shows.

### 4.2.1. Implementation of `Differential_Operator`

The class `Differential_Operator` is partially defined over its template parameters. This is due to the fact that the implementation is done through partial specialization of the template parameters fixing e.g. the discretization method and dimensionality. The primary template without specialization only writes error messages and exits. So

OpenPME

```
ode method: explicit_euler
    ∂U/∂t = Du * ∇²u
```

C++ / OpenFPM

```
Differential_Operator<2, double, 2, OP_DCPSE, KERN_POLY> dcpse_operator(eps);
...
dcpse_operator.template apply_op<Diff_U, Lap_U>(spacing, r_cut, nl, vd);

// Iterate all particles to calculate the new value
it_p = vd.getDomainIterator();
while (it_p.isNext()) {
    // Get particle p
    auto p_key = it_p.get();

    // Integrate with Euler step
    double laplacian_u = vd.template getProp<Lap_U>(p_key);
    double p_prp_change = diff_D * laplacian_u * dt;
    vd.template getProp<Diff_U>(p_key) += p_prp_change;

    // Next particle
    ++it_p;
}
```

Figure 4.4.: Advised code generation in OpenPME.

useful implementations only exist where specializations have been implemented. With the current interface, this can not be the full argument space since arbitrary integers are possible for the dimensionality and the order of the operator but PSE would require an individually crafted kernel for each of these combinations.

The following paragraphs will briefly talk about the specializations for SPH, PSE and DC-PSE without going too much into detail since the implementation is merely the careful following of the mathematical definitions and the conduction of extensive unit testing to detect numerical errors caused by mistakes in implementation, interpretation and generalization of the equations at hand. The implementation primarily targets functionality and not maximal performance. It does not utilize symmetries to reduce computational costs and in case of DC-PSE, no caching or lookup tables are used to prevent the necessity of constructing the smoothing kernel in each iteration for each particle. This implies, that the relative computational costs of the discretization methods measured throughout this work do not necessarily arise from general advantages of the methods. However, for the design and evaluation of an autotuning system this is no major concern.

**SPH** The definition of SPH in Section 2.3.1 is general over all possible numbers of dimensions but requires $D^\beta W_\epsilon$ to be known. This work implements SPH with a Gaussian kernel. The Gaussian kernel in n-D equals

$$W_\epsilon(\mathbf{x}) = \frac{1}{(\epsilon\sqrt{2\pi})^n} e^{-\frac{x_1^2+x_2^2+\cdots+x_n^2}{2\epsilon^2}}.$$

So $\nabla^2 W_\epsilon$ resolves to

$$\begin{aligned}
\nabla^2 W_\epsilon(\mathbf{x}) &= \frac{\partial^2 W_\epsilon(\mathbf{x})}{\partial \mathbf{x}_1^2} + \frac{\partial^2 W_\epsilon(\mathbf{x})}{\partial \mathbf{x}_2^2} + \cdots + \frac{\partial^2 W_\epsilon(\mathbf{x})}{\partial \mathbf{x}_n^2} \\
&= \frac{\mathbf{x}_1^2 - \epsilon^2}{\epsilon^4} \cdot \frac{1}{(\epsilon\sqrt{2\pi})^n} e^{-\frac{\mathbf{x}_1^2+\mathbf{x}_2^2+\cdots+\mathbf{x}_n^2}{2\epsilon^2}} + \cdots + \frac{\mathbf{x}_n^2 - \epsilon^2}{\epsilon^4} \cdot \frac{1}{(\epsilon\sqrt{2\pi})^n} e^{-\frac{\mathbf{x}_1^2+\mathbf{x}_2^2+\cdots+\mathbf{x}_n^2}{2\epsilon^2}} \\
&= \frac{\mathbf{x}_1^2 + \mathbf{x}_2^2 \cdots + \mathbf{x}_n^2 - n\epsilon^2}{\epsilon^4(\epsilon\sqrt{2\pi})^n} e^{-\frac{\mathbf{x}_1^2+\mathbf{x}_2^2+\cdots+\mathbf{x}_n^2}{2\epsilon^2}}.
\end{aligned}$$

Using this equations, the discretization of the Laplacian operator with a Gaussian SPH kernel in n-D does not involve any other complicated steps and can be implemented as such.

**PSE** As for SPH, the general equation for PSE defined in Section 2.3.1 is applicable for each dimensionality and differential operator but requires the use of the correct kernel for a given combination. In contrast, the kernels are not retrieved by applying the differential operator to a base kernel but are created for given dimension and differential operator by hand such that they fulfill specific analytical properties. This work will use kernel functions found in literature [54]. In this case a Gaussian kernel for the Laplacian operator in 1D, 2D and 3D.

$$\eta_\epsilon(x) = \frac{1}{2\epsilon\sqrt{\pi}} e^{-\frac{x^2}{4\epsilon^2}} \qquad x \in \mathbb{R}$$

$$\eta_\epsilon(\mathbf{x}) = \frac{1}{4\epsilon^2\pi} e^{-\frac{|\mathbf{x}|^2}{4\epsilon^2}} \qquad \mathbf{x} \in \mathbb{R}^2$$

$$\eta_\epsilon(\mathbf{x}) = \frac{1}{8\epsilon^3\sqrt{\pi^3}} e^{-\frac{|\mathbf{x}|^2}{4\epsilon^2}} \qquad \mathbf{x} \in \mathbb{R}^3$$

A collection of polynomial PSE kernels of several orders of accuracy for first and second derivatives in 1D and 2D can be found in Eldredge, Leonard and Colonius [24]. Various kernels for the discretization of the Laplacian operator can be found in Schrader et al. [57].

**DC-PSE** In DC-PSE, the kernel is calculated at runtime following the scheme presented in Section 2.3.1. The matrices and vectors are implemented using the C++ Eigen template library [32] which also provides the functionality to compute the decomposition needed to solve the linear system shown in Equation 2.3.

**Unit Testing**

Implementing numerical methods easily results in numerical errors that are often hard to detect as the sources of these errors can be manifold. They can be the result of mistakes made during the analytical calculation e.g. of SPH kernels, misunderstandings in reading the source material describing the discretization method, incorrect generalizations for other dimensions or differential operators and also normal implementation errors. Usually, the compiler is also not very helpful in finding these since they neither involve incorrect types nor other syntactic or semantic errors.

For these reasons, it is even more important than usually to conduct strong unit testing. The different discretization schemes of `Differential_Operator` are tested using Google Test [30] in the 1D, 2D and 3D case for first and second derivatives. The initial conditions for the testing are chosen so that an analytical solution for comparison is easily found. The testing is conducted against these analytical solutions and between discretization schemes.

## 4.3. Measurements

The previous section described how OpenPME could generate multiple variants of the same simulation with different tuning parameter configurations. The next step according to Figure 4.1 is to evaluate the quality of a configuration so that in the end a quality measure can be provided to the optimization algorithm. This Section will answer the questions of when to conduct measurements, how to measure the different objectives (accuracy, computational cost, stability) and how to combine them into a single quality measure. Additionally, it will discuss whether combining them is even favorable.

### 4.3.1. When to Conduct Measurements

The natural point in time to take measurements is at the end of a simulation. Before that it is not possible to measure the actual runtime of the simulation, the final accuracy and to decide on stability at some point. In general this is not feasible though. Moreover, the individual tuning runs would take as long as the final simulation and depending on the tested configuration even longer. This could not be amortized by the improved computational costs.

To circumvent this problem, one has to measure only over a small portion of the simulation. Since tuning requires many benchmark runs, this portion should presumably be less than one percent to make sure that the tuning costs are worthwhile.

Effect on compuational costs The smallest drawback of the reduced measuring time is in determining the computational cost. From the runtime of the measured portion one can extrapolate for the whole simulation by plain multiplication. This does not always

work perfectly since moving particles might come closer together which leads to an increased number of particles in the neighborhood of each other which implies more interactions and by that higher computational costs. However, this effect is limited in real scenarios since bulking particles usually require remeshing which sets them back to grid positions.

**Effect on accuracy** The effect on the measured error is more problematic. It strongly depends on how much change is currently happening in the simulation. If quantities are changing a lot, one should also expect a higher absolute error. Additionally, errors might accumulate or cancel out over time. The second is plausible for a diffusion reaction where one should expect an even distribution after a sufficiently long time period. Since PSE ensures that quantities are only exchanged between particles, this final distribution will be the same as the accurate one even if the simulation was inaccurate in between[1]. This essentially means that the tuning will provide a configuration with a specific error at a chosen point in time. It will and cannot claim a specific error for the end of the simulation. While this is certainly limiting it is, if communicated correctly to the user, still very valuable to tune for a target accuracy at an early point in time.

**Effect on stability** However, the biggest downside of measuring only over a short period of time is in identifying instabilities. If they occur within the measured period, they are easily detected. If not, it is not possible in the general case to know whether the simulation is stable or not. Still, for both simulations introduced in Chapter 3, unstable simulations showed their instability within the first few time steps. If the chosen configuration by the tuning reveals to be unstable, one would have to decrease $\delta t$ until it reaches stability.

At large, measuring over a reduced period of time is a clear engineering decision that does come with crucial disadvantages but it is still necessary.

### 4.3.2. How to Conduct Measurements

This section will describe how the quality measures are measured.

#### Computational Cost

The computational cost is measured by inserting measurement points into the target C++ code. The start point is measured just before the main time loop starts. This means, that the setup time for creation, initialization and distribution of particles between processors is not captured. The main reason for this is that the setup time is negligibly short compared to the full simulation time but might not be in comparison to the

[1] This is only true if the simulation is stable.

measured time period. The time is naturally measured up at the end of the tuning period. The runtime is measured using `std::chrono::steady_clock` in milliseconds, which is adequate since the measured time periods usually span over multiple seconds or minutes.

It is also possible to estimate the runtime using predictive methods that are trained on a small set of sample measurements. This way it would not be necessary to run a simulation for a given configuration at all to get a measure of the computational costs. The prediction approach and how it can be utilized is presented in Section 5.4.

#### Accuracy / Error

The evaluation of the error requires some more considerations than the computational cost. It basically involves two steps: Measuring the error at the individual particles and accumulating the individual errors to a global one.

The individual error is measured by subtracting the measured value from the correct one and taking the absolute value of the result $|u_{correct} - u_{measured}|$. The main problem here is that in the general case the correct solution is unknown. Otherwise, one would not need to simulate the problem in the first place but could use the correct analytical solution.

As first step, it is possible to calculate the correct analytical solution of the Laplacian for the diffusion problem with the given initial condition of the field value $u$ by

$$u = \frac{1}{4\sigma^2\pi} * \mathrm{e}^{-\frac{x^2+y^2}{4\sigma^2}}.$$

The Laplacian of this evaluates to

$$\Delta\left(\frac{1}{4\sigma^2\pi} \cdot \mathrm{e}^{-\frac{x^2+y^2}{4\sigma^2}}\right) = \frac{x^2+y^2-4\sigma^2}{16\sigma^6\pi} \cdot \mathrm{e}^{-\frac{x^2+y^2}{4\sigma^2}}.$$

This allows first measurements for the diffusion problem without time stepping. To measure the error over multiple time steps for a general problem, one has to use a comparison configuration which is assumed to be more accurate as done by Bourantas et al. [13]. For this purpose, a configuration is used which halves the spatial and temporal resolution of the lowest value in the search space and takes the maximum neighborhood radius for the chosen discretization method and kernel. In this work, PSE with a Gaussian kernel and a neighborhood radius of nine is used.

Since both considered problems use a regular Cartesian particle distribution, it is possible to measure the accuracy at common grid points. To ensure a high number of overlapping grid points between the configurations in the search space and the reference configuration, only particle numbers that are a multiple of 50 per dimension are considered. By that, it is ensured that there are always 2 500 common particle positions.

Achieving the same for non Cartesian distributions or measuring the accuracy for all particles and not only the overlapping subset would require additional steps. One possible approach would be to use a smoothing function to calculate values between particles on the reference solution. The existing SPH implementation without a differential operator applied, could fulfill this purpose. This approach would take considerably more computational resources for the evaluation and introduce a new source of inaccuracies that would have to be compensated by higher numbers of particles. So in the considered cases it is less favorable but will be necessary to extend the tuning for a wider range of problems.

The individual errors at particles have to be accumulated in order to form a global accuracy that can be compared between configurations. In the particle methods literature, the $L_2$ error norm often fulfills this purpose [62, 69, 13, 43, 51]. For $N$ particles it is defined as

$$L_2 = \sqrt{\frac{1}{N}\sum_{p=1}^{N}\left(u_{correct}(x_p) - u_{measured}(x_p)\right)^2}.$$

It equals the quadratic mean over the individual errors. The $L_p$ norm describes the generalization of the $L_2$ norm for other exponents $p$

$$L_p = \sqrt[p]{\frac{1}{N}\sum_{p=1}^{N}\left(u_{correct}(x_p) - u_{measured}(x_p)\right)^p}$$

and resembles the generalized mean over the individual errors. The limit case of the $L_p$ norm, the $L_\infty$ norm is defined by the maximum individual error

$$L_\infty = \max_{p=1}^{N}\left(u_{correct}(x_p) - u_{measured}(x_p)\right).$$

While other values for $p$ could be used, the only two relevant norms in the literature are $L_2$ and $L_\infty$.

### Measuring stability

Deciding stability over the considered time period is accomplished by checking whether the discretized field values stay bounded. Usually this would imply the further question where to set these boundaries. Since unbounded values result in huge errors for bounded correct solutions, the check can be omitted. High errors are already regarded as unfavorable when tuning the accuracy. This observation reduces the tuning task to two-objective optimization problems.

To decide if the final simulation is stable or not, it suffices to check whether the values stay within the boundaries of the used datatype since it usually takes only few time steps to reach this condition after crossing any other boundaries.

### 4.3.3. Multi-Objective Autotuning

When tuning for multiple objectives, there most often is no such thing as a single best solution. Trade-off optimal solutions, known as Pareto-optimal solutions exist. The goal is either to transform the multi-objective optimization via a quality measure into a single-objective problem, or to find Pareto-optimal solutions and then decide the trade-off [21, p. 407-408].

"The classical means of solving such problems were primarily focused on scalarizing multiple objectives into a single-objective, whereas the evolutionary means have been to solve a multi-objective optimization problem as it is" [21, p. 403].

#### Dominance and Pareto-Optimality

This section will formally define dominance and pareto-optimality according to Sbalzarini et al. [55].

A configuration $\mathbf{c} = (x_1, \ldots, x_m) \in X$ with parameters $x_i$ has the objective vector

$$\mathbf{f}(\mathbf{c}) = (f_1(\mathbf{c}), \ldots, f_n(\mathbf{c}))$$

A configuration $\mathbf{a}$ is said to dominate a configuration $\mathbf{b}$ (written as $\mathbf{a} \succ \mathbf{b}$) if and only if:

$$\begin{aligned} &\forall i \in \{1, \ldots, n\} : f_i(\mathbf{a}) \geq f_i(\mathbf{b}) \\ \wedge \quad &\exists j \in \{1, \ldots, n\} : f_j(\mathbf{a}) > f_j(\mathbf{b}) \end{aligned}$$

A configuration $\mathbf{a}$ is called nondominated regarding a set of configurations $X$ if and only if there is no configuration in $X$ that dominates $\mathbf{a}$. Formally

$$\nexists \mathbf{b} \in X : \mathbf{b} \succ \mathbf{a}.$$

Nondominated configurations regarding the whole search space are called *Pareto-optimal*. A set of Pareto-optimal configurations is called *Pareto-front*. Figure 4.5 shows the Pareto-front for the two objective problems with the trade-off between runtime and error.

#### Optimization Procedures

It is clear from the definition of Pareto elements that the best configuration should be one. Otherwise it is dominated by a Pareto element which hence is preferable. In the book *Search Methodologies*, Kalyanmoy Deb describes the two main approaches of finding a desired solution [21]. Both are visualized in Figure 4.6. The approaches differ in the point of deciding the trade-off between the objectives to find a single best solution.

Approach A (Figure 4.6, bottom left) starts by finding a set of nondominated solutions and then chooses among them. This choice may involve user feedback.

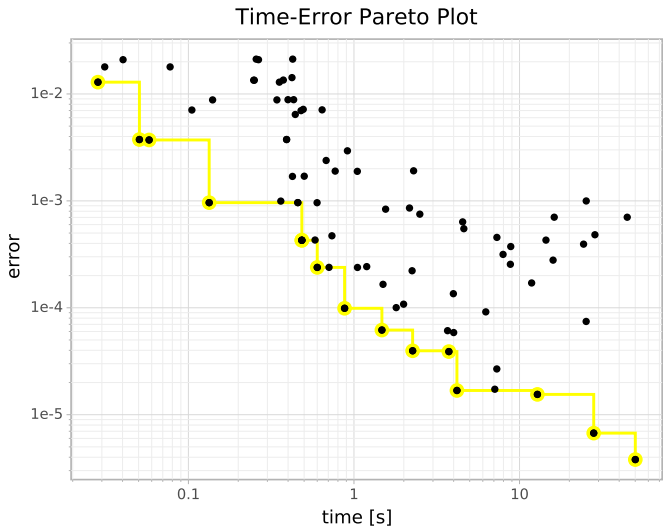



Figure 4.5.: Nondominated configurations in the objectives time and error are connected via the yellow line. Assuming that the whole search space is visualized, those configurations are called Pareto-optimal. Any position above the yellow line is dominated by one of the Pareto elements. The plot uses measurements of the diffusion problem for t=0, with varying $\epsilon$, $h$, $r_{cut}$ and discretizations methods.

Approach B (Figure 4.6, top right) starts by defining a composite objective function that reduces the problem to a single-objective one. This is then optimized by a single-objective approach.

In this thesis, only approach B will be used. It promises to take fewer steps since it searches only for one solution to begin with. Lowering the number of tuning steps is crucial since they are unusually extraordinarily costly as discussed in 4.3.2. Additionally, approach A needs user feedback at the end of the tuning to be truly beneficial. Without user feedback, some score has to be calculated to decide which single configuration should be used. That function then could have also been used to reduce the multi-objective problem to a single-objective one immediately. Since this autotuning approach is meant to be integrated into the OpenPME compiler later, user interaction would at least be unusual but not inconceivable.

However, the biggest disadvantage of approach B is that the decision has to be made at a point in time when the achievable trade-offs are unknown. As a result this, the user might set a different focus than he would have in approach A. For this reason, it would be interesting to investigate how well approach A would perform in practice.

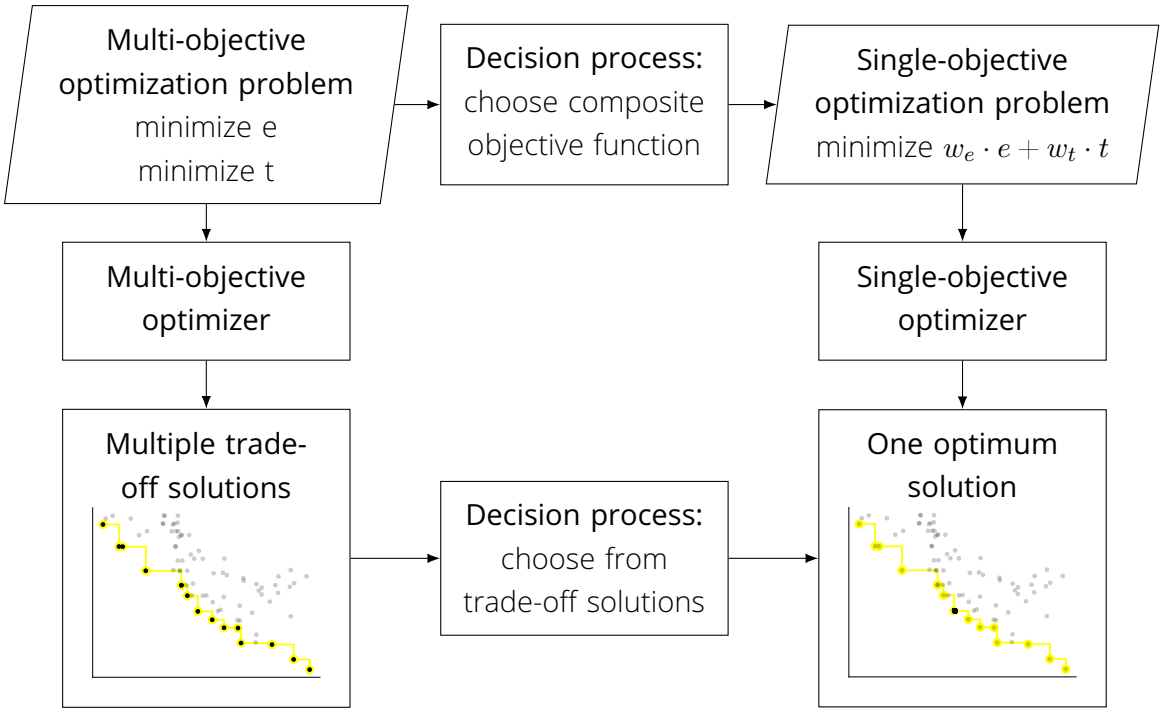



Figure 4.6.: Schematic of the two possibilities for multi-objective optimization procedures based on Deb [21, pp. 408-409].

### Weighted sum

One of the most widely used approaches of combining a set of objectives into a single scalar value are weighted sums. The composite objective function $F(c)$ is formed by summing up the weighted objectives.

$$F(c) = \sum_{m=1}^{M} w_m f_m(c)$$

Usually, the objectives are normalized for comparability and the weights are adjusted according to preferences. Since the absolute value of $F(x)$ does not change the problem, the weights are usually chosen such that their sum is one.

Even though the grand scheme of weighted sums is easy, it has one major difficulty on how to choose the weights and performing the normalization. These choices define whether the configurations are good or not. In the case of optimizing computational cost $t$ and error $e$, both objectives have possible values for the whole interval $[0, \infty)$. Thus, there is no obvious choice even for normalization. For this reason, this work proposes the use of a goal objective vector $(t_g, e_g)$ which defines the target relation between the individual objectives. The values in the target vector are used as normalization quotients.

$$F(c) = 0.5 \cdot \frac{t(c)}{t_g} + 0.5 \cdot \frac{e(c)}{e_g}$$

Figure 4.7a displays the effect of this composite function. All configurations on the gray line share the same quality measure as the goal objective vector $(t_g, e_g)$ of $F(c) = 1$. All

points below the curve have a quality value of less than one and are considered superior. The inverse is true for the points above the curve. One can easily imagine how moving the point $(t_g, e_g)$ would redefine the target trade-off. Notice that actual weights $w$ do not have to define the relation between $e$ and $t$ since this is already done by the goal vector. Therefore, they are set to $0.5$ to ensure that a configuration with the same quality as the target vector has a quality value of one. If not for this reason, or to stick to the convention that $\sum_{m=1}^{M} w_m = 1$, both could be set to one and by that can be omitted.

The weighted sums approach allows to trade some degree of error and runtime. If the relative error $\frac{e(c)}{e_g}$ decreases by $x$, this might be compensated by an increase of $x$ in the relative runtime $\frac{t(c)}{t_g}$. Whether this is regarded favorable or not largely depends on the preferences of the user. To enforce a stricter compliance to the relation between $t_g$ and $e_g$, an exponent can be introduced into the normalization. By that, deviations in both directions have increased impact as visualized in Figure 4.7b.

$$F_2(c) = 0.5 \cdot \left(\frac{t(c)}{t_g}\right)^2 + 0.5 \cdot \left(\frac{e(c)}{e_g}\right)^2$$

The same is possible in the other direction by using an exponent smaller one.

$$F_{0.5}(c) = 0.5 \cdot \sqrt{\frac{t}{t_g}} + 0.5 \cdot \sqrt{\frac{e}{e_g}}$$

This reduces the effects of the exact choice of $(t_g, e_g)$ as shown in Figure 4.7c.

The principle can be generalized for any exponent $p$ to

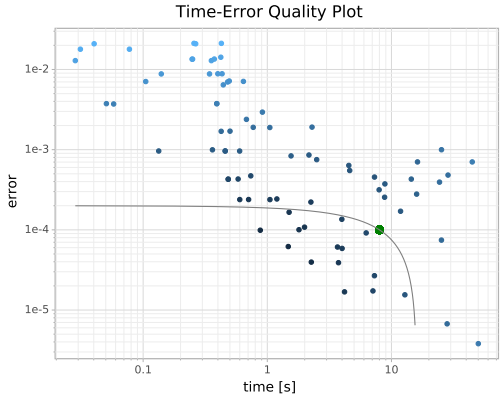
$$F_p(c) = 0.5 \cdot \left(\frac{t(c)}{t_g}\right)^p + 0.5 \cdot \left(\frac{e(c)}{e_g}\right)^p.$$

### Objective Product

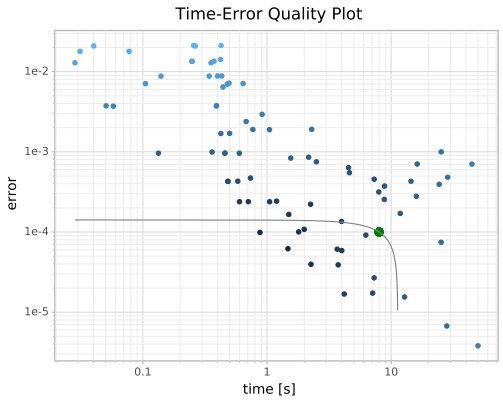
Even though weighted sums are the most popular methods of combining multiple objectives into a single one, they are not the only option. A simple product of the objectives could be used.

$$F_{product}(c) = t(c) \cdot e(c)$$

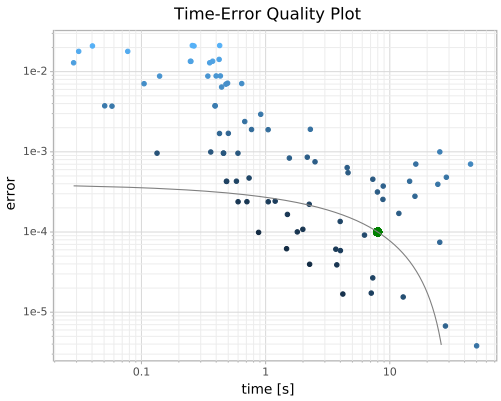
This would essentially describe a combined objective where the halving of one original objective would allow the duplication of another. One advantage of this method is that no goal vector is needed. The method is visualized in Figure 4.8a. As one can see in the graph, this method closely resembles the form of the Pareto front. While this seems to be good at first, it essentially means that it is not effective at select which element along the Pareto front should be chosen. The method may select elements at both extremes, very fast and inaccurate and the inverse, which is undesirable. At one extreme would be an imaginary method that does nothing, and would lead to a finite error and a runtime of $t = 0$. With the product method this would result in a perfect score of $F_{product} = 0$.




(a) Normalized weighted sum




(b) Square-normalized weighted sum




(c) Root-normalized weighted sum

Figure 4.7.: Weighted sum objective functions. Points along the line share the same quality while the points below have a higher and the points above a lower quality.

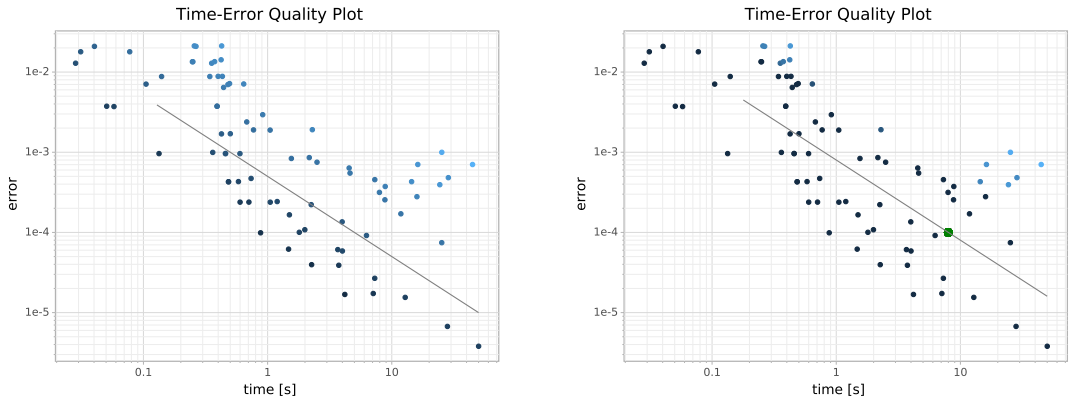
It should be noted that the objective product could be expressed using the weighted sum approach as one can see by applying the logarithm. This is valid because the logarithm is strictly increasing for values greater than zero and consequently does not change the order induced by the combined objective function.

$$F_{log}(c) = \log t(c) + \log e(c), \quad t(c), e(c) > 0$$

While the imposed order by the weighted sum of logarithms and the objective product is the same, they might be perfectly handled in different ways by single-objective optimization algorithms. The weighted sum of logarithms is depicted in Figure 4.8b.

### Threshold Objectives

Besides the weighted sum approach and the objective product, the idea of defining a threshold for all but one objective and optimizing only the remaining one is a popular approach of transforming a multi-objective problem into a single-objective one. In *Search*




(a) Objective product

(b) Logarithmic normalized weighted sum

Figure 4.8.: Objective product and the equivalent using weighted sums of logarithms. Points along the line share the same quality while the points below have a higher and the points above a lower quality.

*Methodologies*, Kalyanmoy Deb also refers to this principle as the $\epsilon$-constraint method where $\epsilon_m$ is the threshold of the objective function $f_m$ [21, p. 420-421]. The problem can be formulated as

$$\begin{aligned} &\text{minimize } f_k(c) \\ &\text{subject to } f_m(c) \leq e_m, \quad m = 1, 2, \ldots, M \wedge m \neq k. \end{aligned}$$

If one of the objectives exceeds its threshold, the configuration is invalid. One major advantage of this method is that it closely resembles methods used in benchmarking discretization schemes in particle methods. There, the error level is fixed to different values and the runtime of the fastest configuration within these error limits is measured [57].

To translate the aforementioned formalism into a single-objective function, the invalidity can be interpreted as an infinite objective value. Applying this idea to the problem at hand, one obtains

$$F_\epsilon(c) = \begin{cases} t(c) & e(c) \leq \epsilon_e \\ \infty & \text{otherwise} \end{cases}$$

as a composite function. This might be disadvantageous though, since the single-objective tuner does not receive any feedback on improvements or degradation for invalid configurations. This is particularly problematic if a significant portion of the search space is invalid according to the threshold, since the optimization algorithm would not have guidance to find any valid configuration at all. This problem can be solved by replacing $\infty$ with a value that is larger than the maximum value for the open objective.

$$F_\epsilon(c) = \begin{cases} t(c) & e(c) \leq \epsilon_e \\ t_{max} + e(c) & \text{otherwise} \end{cases}$$

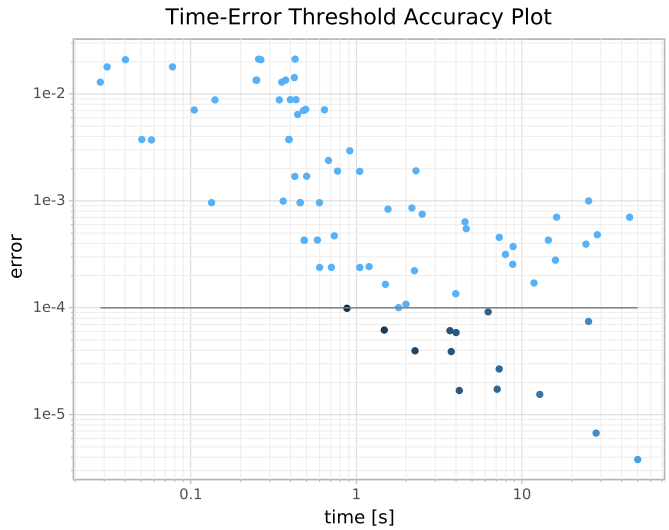



Figure 4.9.: Error threshold. All configurations above the line are considered invalid. Among the remaining, the ones with the lowest runtime are considered to have the highest quality.

In general, there is no upper limit on the computational costs $t_{max}$ in particle methods, but it exists for a given finite search space. The $t_{max}$ could either be approximated or a general upper limit for realistic runtimes might be used.

# 5. Optimization Approaches

This chapter introduces the optimization approaches applied in this work. A selection of general search algorithms are available through utilizing the OpenTuner framework. Additionally, a detailed search space analysis for the diffusion and Gray-Scott simulations is presented. The obtained results are used to design a model-based optimization approach.

## 5.1. OpenTuner

Autotuning frameworks are the best approach when implementing the optimization part of an autotuner. In particular, they provide a wide selection of optimization algorithms which help to overcome the burden of implementing them by hand. Besides the fact of saving a lot of time, widely used frameworks provide good code quality and resilience as they have been tested on real world cases for a longer time. In contrast to general optimization frameworks like Pagmo [12] and SciPy.optimize [64], autotuning frameworks are well fitted for code optimization problems. They combine different parameter types such as integer, floating point and categorical types and do not rely on further knowledge like gradients which cannot be easily determined by a single measurement.

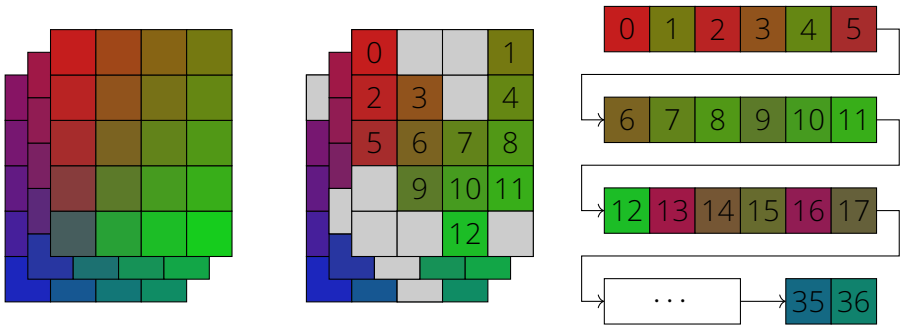
### 5.1.1. Why Use OpenTuner

Unfortunately, the number of available general-purpose autotuning frameworks is fairly limited. Section 2.1.2 introduced OpenTuner and ATF. This work utilizes OpenTuner. The decision for choosing it over ATF will be discussed in the following paragraphs taking into account their provided algorithms, features, interface, development state and availability.

**Provided Algorithms** OpenTuner contains a wide selection of local and global optimization algorithms and is capable of using them in conjunction. ATF provides per default

exhaustive search, simulated annealing and a binding to OpenTuner's search engine. Since OpenTuner also implements simulated annealing, they only differ in the exhaustive search provided by ATF which is not feasible for the problem at hand. Additionally, both frameworks allow the implementation of new search techniques. Consequently, there is no reason to choose one framework over the other due to the selection of search algorithms.

**Parameter constraints** One of the key improvements of ATF over OpenTuner is the possibility to define dependencies between tuning parameters. Indeed, this would be beneficial for autotuning the discretization parameters of continuous particle simulations since the range of reasonable cutoff radii depends on the chosen discretization method and kernel. However, the main concern of ATF's constraints is their implementation. The constrained search space gets mapped to a single OpenTuner integer tuning parameter and specifies the index in ATF's search space. A visualization and an explanation can be found in Figure 5.1. This approach loses almost all locality information. Slight changes in the cutoff radius, particle distance, time step size or kernel width could also result in only small performance changes. Therefore, locality is of great importance in guiding essentially all optimization algorithms. Nonetheless, ATF demonstrated performance improvements using constraints over unconstrained OpenTuner, but just for a case study where only one in $10^7$ unconstrained configurations was valid. Since this is not the case here, ATF constraints could lead to worse performance. Instead, a subdivision of the search space will be performed and discussed in Section 5.1.2.




(a) Full search space (b) Search space with constraints (c) ATF's mapped search space

Figure 5.1.: Problem of ATF constraints. Configurations with similar parameters can be expected to have a comparable performance. Configuration similarity is visualized by color. Constraints invalidate certain configurations of the full search space as shown in (b). When mapping the whole search space into a single integer as shown in (c), most similarities are lost as shown by the mismatching colors. Even if the neighbors are still similar as in row two of (c), the total number of neighbors strongly decreased.

**Interface** New autotuners are created with OpenTuner by implementing a child class of the `MeasurementInterface` class providing methods for search space creation and configuration evaluation. ATF utilizes annotated source files to generate autotuners automatically. While this is very convenient for application level autotuning, it does not provide enough flexibility and is not expressive enough to implement the sophisticated accuracy evaluations that are needed.

**Development state and availability** OpenTuner is available in GitHub and provides tutorials on how to implement tuners and new optimization techniques on opentuner.org. It has successfully been used to optimize GPU compiler parameters [15, 14, 26], high-level synthesis for FPGAs [16] and image processing schedules [70]. ATF, in contrast, is currently not available online.

In conclusion, ATF and OpenTuner provide the same variety of optimization algorithms but OpenTuner's interface fits better for the purpose of this work. Furthermore, this framework is easily available and widely used in practice. ATF's support for constraints is highly beneficial but would be disadvantageous for the tuning performance of the given problem.

### 5.1.2. Defining the Search Space

As mentioned when discussing ATF constraints, the considered range of cutoff radii depends on the discretization method and kernel function. Additionally, they influence the effect of the kernel width, the error convergence order and time stepping stability. Hence, there is barely similarity between two configurations that are identical in all parameters but the discretization method and kernel function. Considering that, the full optimization is divided into multiple smaller optimizations. One for each discretization method and kernel function pair. Since there is currently only one kernel function implemented per discretization method, this amounts to three optimizations. One for SPH, PSE and DC-PSE respectively.

The remaining open parameters are the normalized cutoff radius $\frac{r_c}{h}$, normalized kernel width $\frac{\epsilon}{h}$, the spatial resolution $h$ represented by the number of particles per dimension and the time step size $\delta t$. All of them are represented as fixed-point numbers and all value ranges have a fixed step size. Since OpenTuner does not support fixed-point numbers with a given step size, the ranges are transformed into integer ranges with step size one which is done by plain multiplication with the step size. For example, the range of values for $\delta t$ of $\{0.05, 0.06, \ldots, 0.99, 1.00\}$ is represented as the integer range $[5, 100]$. The corresponding translations between ranges have to be done before and after interactions with the OpenTuner API.

```
IntegerParameter('cutoff_radius', 3, 10)  # factor 1
IntegerParameter('epsilon', 6, 20)        # factor 0.1
IntegerParameter('x_particles', 1, 40)    # factor 50
IntegerParameter('dt', 5, 100)            # factor 0.01
```

Figure 5.2.: OpenTuner Integer parameter specification.

### 5.1.3. Evaluating a Configuration

The general approach on conducting measurements and combining them into a single quality measure has been discussed in Section 4.3. Here, only OpenTuner-specific issues are discussed.

Implementing an instance of OpenTuner's `MeasurementInterface` involves the specification of the `run` method which maps a configuration dictionary to a `Result` containing the quality measure. A possible configuration dictionary is shown in Figure 5.3

```
{
    'cutoff_radius': 5,
    'epsilon': 10,
    'x_particles': 5,
    'dt': 10
}
```

Figure 5.3.: Configuration as passed by OpenTuner.

The OpenTuner configuration is then translated into the corresponding configuration with correct values ($\frac{r_c}{h} = 5, \frac{\epsilon}{h} = 1.0, \frac{1}{h} = 250, \delta t = 0.1$). The configuration is run as described in Section 4.2 and the measurements are conducted as in Section 4.3. The calculated quality measure is afterwards returned inside a `Result` object. OpenTuner provides also the possibility to define other objectives than minimizing a single value. The `ThresholdAccuracyMinimizeTime` objective allows to use the $\epsilon$-constraint method by returning a time and accuracy measure inside the `Result`.

## 5.2. Model-Based Search

Model-based search approaches require more effort to design as they entail good understanding of the search space. Fortunately, particle methods are analytically well understood. For all used spatial discretization methods the error convergence rate is known. The same is true for the used temporal discretization method. The main problem is that these theoretical convergence rates assume otherwise perfect conditions. The spatial error convergence for example assumes infinite neighborhood radius, arbitrary accurate floating point values and time step sizes approaching zero. To investigate the

influence of the interaction of tuning parameters on accuracy and computational cost, a parameter study is conducted.

### 5.2.1. Parameter Study

Since not all tuning parameters can be displayed in a single diagram, a series of them is used. For this purpose, exhaustive measurements are conducted over a reduced parameter space for both case study simulations. The used parameter spaces for Gray-Scott and diffusion are shown in Table 5.1 and Table 5.2 respectively.

| Discretization method | Smoothing kernel | Parameter | Measured values |
|---|---|---|---|
| | all | $\frac{1}{h}$ | $\{50, 100, 150, \ldots, 800\}$ |
| | | $\frac{1}{\delta t}$ | $\{1, 2, 4, 8, \ldots, 512\}$ |
| | | $\frac{\epsilon}{h}$ | $\{1.0\}$ |
| SPH | Gaussian | $\frac{r_c}{h}$ | $\{4, 5, 6, 7\}$ |
| PSE | Gaussian | $\frac{r_c}{h}$ | $\{4, 5, 6, 7\}$ |
| DC-PSE | Polynomial | $\frac{r_c}{h}$ | $\{2, 3, 4\}$ |

Table 5.1.: Parameter space for Gray-Scott parameter study.

Figure 5.4 shows multiple aspects. Firstly, if $\delta t$ is not small enough, $h$ does not matter and the error does not converge to zero. The smaller the time step, the larger the range of particle distances for which the error converges. Since there is basically no change from $\delta t = \frac{1}{256}$ to $\frac{1}{512}$ it can be assumed that the former is small enough for the measured interval. Secondly, the convergence rate equals two, as should be expected for second order accurate kernels, assuming small enough $\delta t$ and $h$. This is visualized by a comparison line with slope $-2$ which corresponds to the actual quadratic function for the accuracy on a nonlogarithmic scale. Thirdly, the used neighborhood radius does set a limit on the minimal error for SPH and PSE. While a low $r_c$ does not make a difference for high $\delta t$ and $h$, increasingly higher neighborhoods are required for smaller spatial and temporal resolutions. This does not apply to DC-PSE since the discretization correction eliminates the cutoff error.

For the diffusion simulation, a slightly different parameter space is analyzed (Table 5.2). Since the requirements for time step sizes are lower, a larger range of spatial resolutions could be measured in an adequate amount of time.

In comparison to Gray-Scott, the measurements for the diffusion simulation show very low errors even for relatively large $\delta t$ (Figure 5.5). This is most likely due to the fact that fewer interactions occur and the pattern formed by the simulation is less complex.

| Discretization method | Smoothing kernel | Parameter | Measured values |
|---|---|---|---|
| | all | $\frac{1}{h}$ | $\{50, 100, 150, \ldots, 1000\}$ |
| | | $\frac{1}{\delta t}$ | $\{1, 2, 4, 8, \ldots, 32\}$ |
| | | $\frac{\epsilon}{h}$ | $\{1.0\}$ |
| SPH | Gaussian | $\frac{r_c}{h}$ | $\{4, 5, 6, 7\}$ |
| PSE | Gaussian | $\frac{r_c}{h}$ | $\{4, 5, 6, 7\}$ |
| DC-PSE | Polynomial | $\frac{r_c}{h}$ | $\{2, 3, 4\}$ |

Table 5.2.: Parameter space for diffusion parameter study.

Especially for PSE with large $\delta t$ the error spikes to extremely low levels (Figure 5.5). This behavior can be assumed to be an artifact of the small measurement period and absolute chance. Due to its relative simplicity, the diffusion simulation is prone for these kind of measurement errors. The effect is especially articulated for PSE presumably due to its symmetry. These spikes are a big problem for autotuning though, since they seem to have a very good trade-off between runtime and accuracy according to the measurements. To prevent these, longer measurement periods could be used or model-based approaches that do not consider them in the first place.

Other than that, the observations coincide with the Gray-Scott case. The relevance of the neighborhood size becomes even more apparent caused by the larger range of spatial resolutions.

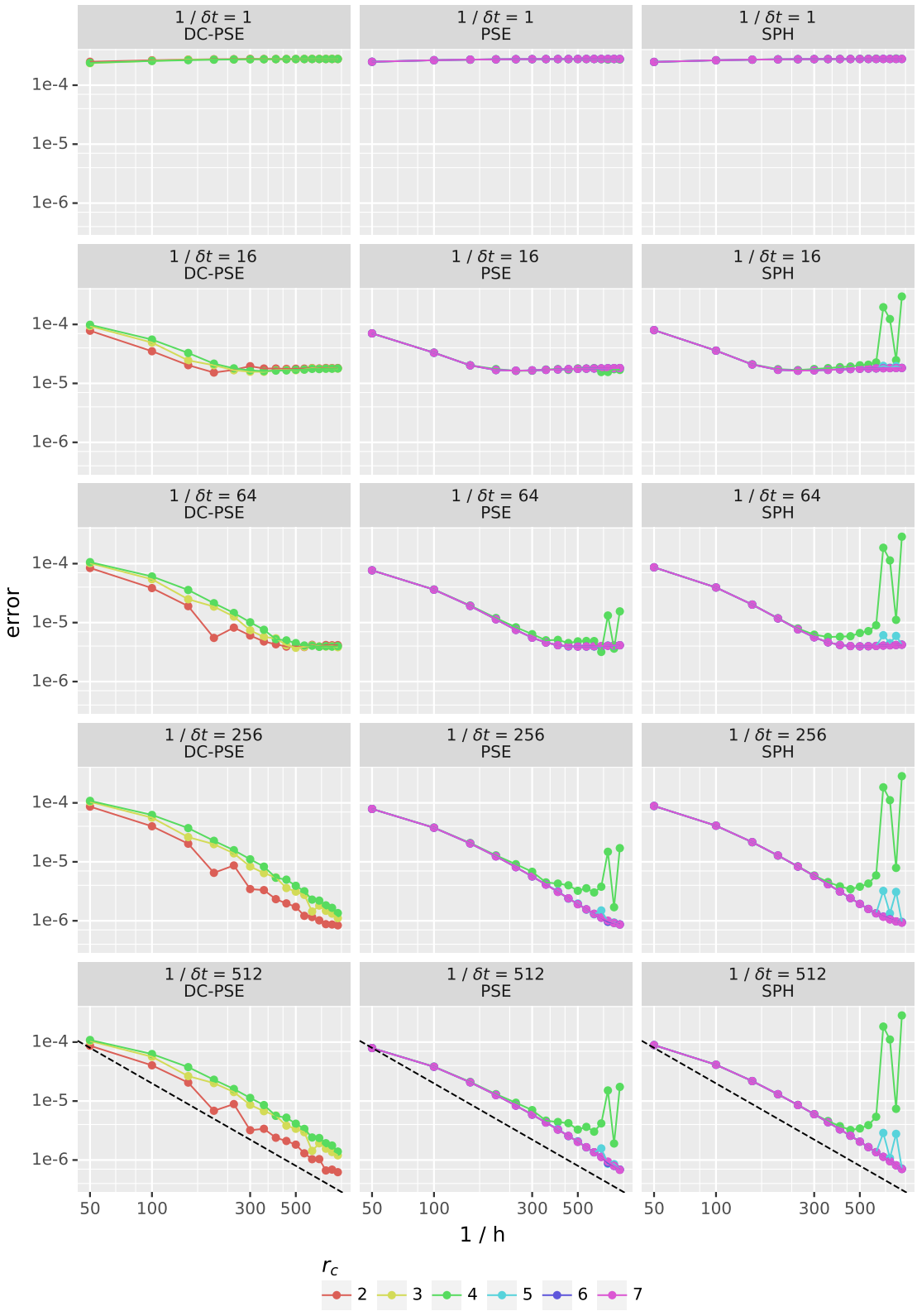



Figure 5.4.: Gray-Scott parameter study relating the error, spatial and temporal resolution and discretization method. The dashed line has a slope of -2 representing quadratic relations in the logarithmic diagram.

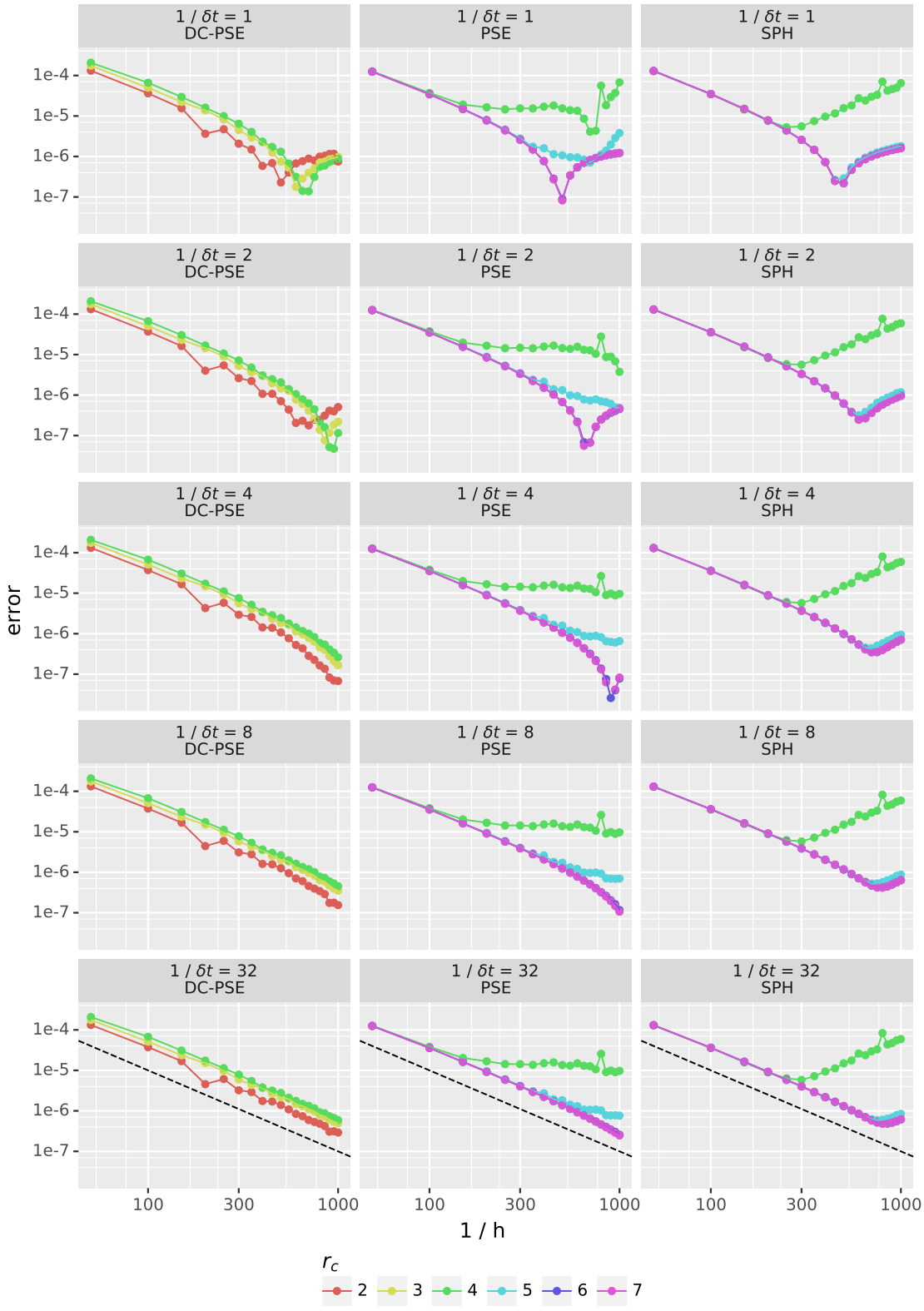



Figure 5.5.: Diffusion parameter study relating the error, spatial and temporal resolution and discretization method. The dashed line has a slope of -2 representing quadratic relations in the logarithmic diagram.

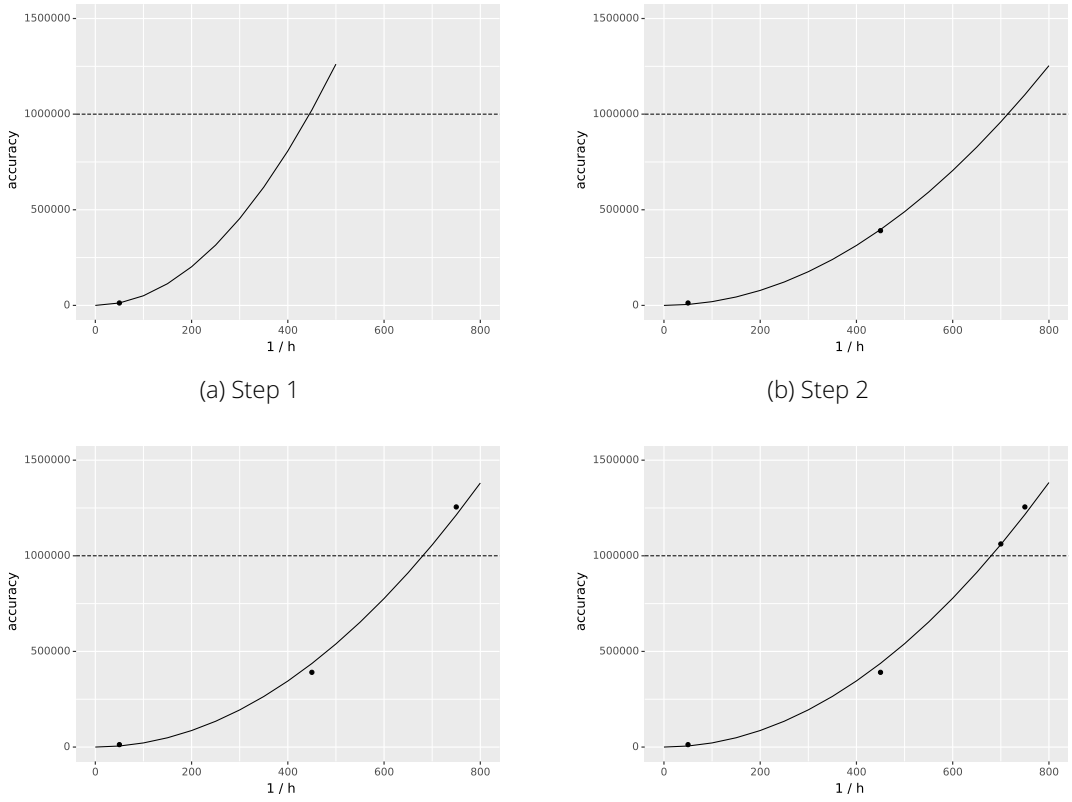
### 5.2.2. Accuracy Regression Based on Number of Particles

The previous measurements encourage the idea of determining the exact relation between the error and spatial resolution. For second order accurate discretization methods, one should expect the error to shrink quadratically with decreasing particle distances which coincides with the measurements when all parameters where large or small enough. Measuring a small set of values, the function $e = a \cdot h^2$ can be estimated using regression. By considering the root of the measured error, it becomes a case of linear regression.

Based on these observations the following algorithm is proposed. It uses a threshold error optimize runtime approach.

1. Set $r_c$ to the maximum value in the search space $r_{c_{max}}$
   Set $\delta t$ to the minimum value in the search space $t_{min}$
   Set $h$ to the minimum value in the search space $h_{min}$
   Set $\frac{\epsilon}{h}$ to 1
   $C = \emptyset$
2. Measure configuration and place in the set $C$
3. Use linear regression on $C$ to estimate $a$ in $e = a \cdot h^2$
4. Use $h = \sqrt{\frac{e}{a}}$ to estimate $h_{estimate}$ for $e_{threshold}$
5. $h_{next} := \max(h_{estimate}, h_{min})$
6. Create new configuration $c$ by setting $h$ to $h_{next}$
7. $$\begin{cases} \text{Go to 2.} & \text{if } c \notin C \\ \text{Continue at 8.} & \text{if } c \in C \text{ and } e(c) \leq e_{threshold} \\ \text{Decrease } h_{estimate} \text{ by one step and go to 5.} & \text{if } c \in C \text{ and } e(c) > e_{threshold} \text{ and } h_{next} > h_{min} \\ \text{Report } e_{threshold} \text{ could not be reached} & \text{otherwise} \end{cases}$$
8. Use bisection search to find the lowest $\epsilon$ that does not change $e$ significantly
9. Use bisection search to find the lowest $r_c$ that does not change $e$ significantly
10. Use bisection search to find the largest $\delta t$ that does not change $e$ significantly

Steps 1. to 7. find the most suitable particle distance for the given error threshold. There, the *best* values for $r_c$ and $\delta t$ in the search space are used. Since neither the minimal nor the maximal $\epsilon$ in the search space can be expected to perform best, the value of 1 is used. The minimum kernel might not take neighboring particles sufficiently into account whereas the maximum might lead to not large enough $r_c$. Steps 2 to 7 repeatedly try to estimate the value of $h$. This process is visualized in Figure 5.6.




(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

Figure 5.6.: Model-based search steps used to optimize $h$. The plots use an accuracy value (one divided by the error) rather the error to better visualize the quadratic dependency between error and resolution.

The regression search succeeds when the estimated configuration was already measured and is below the error threshold as in Figure 5.6d. The regression based search is followed by individual bisection searches on the remaining three parameters. In the bisection search the valid range of the parameters is repeatedly split in two, keeping the lower or upper half depending on whether the measurement of the midpoint lead to a significant change of the error or not. To prevent to the selection of the seemingly low error spikes seen in diffusion, the requirement is *insignificant change* rather than *no significant increase*.

The major disadvantages are on the one hand that $\epsilon$ is handled subpar since no obvious most accurate value is available as for the other parameters and on the other hand that by choosing the extreme values for $r_c$ and $h$, the required measurement time of the configurations can be expected to be above average.

The model-based search algorithm is implemented as new OpenTuner optimization technique using the `SequentialSearchTechnique` interface. While no features of OpenTuner are needed for the implementation, the integration into OpenTuner simplifies the

optimization technique selection. It can be accessed as any other technique by specifing it in OpenTuner's `--technique` parameter.

The linear regression is implemented using `scikit-learn` [50].

## 5.3. Model-Based Prediction

The previous section utilized the knowledge about the error convergence rate of the spatial discretization method in conjunction with empirical measurements to predict the best number of particles to be used for a specified error threshold. The same idea can be expanded to also predict the best choice for $\delta t$. As for the relation between error and spatial resolution, measurements are conducted to validate the analytical properties in actual simulations. Since the diffusion simulation did not require small time step sizes as in Gray-Scott, the measurements are done for the latter.

Figure 5.7 clearly shows that for small enough $h$, the error converges with the expected rate of one. The convergence rate depends on the temporal discretization scheme, which is fixed to explicit Euler. The graphs also visualizes that for small enough $h$, the spatial discretization method does not matter. This implies that the predictive model can be used for all of them together and does not have to be trained individually.

Combining the idea of using regression to determine the actual dependency between the spatial resolution and the error with the observation that the same can be done for the temporal resolution, a fully predictive model can be created. The main idea of this predictive model is to only do a few swift measurements to adapt it to the given simulation. The accuracy of the predicted configuration is then never evaluated accepting the chance of slight inaccuracies and gaining the advantage of not requiring a highly accurate comparison configuration. This significantly reduces the computational costs of the tuning phase.

To integrate the neighborhood radius into the prediction, a simple heuristic is created choosing $r_c$ depending on the target accuracy. Based on the observations in this chapter, for a target accuracy below $10^{-n}$, a value of $r_c = n \cdot h$ is selected for SPH and PSE while for DC-PSE always $r_c = 2 \cdot h$ is used. This heuristic is not guaranteed to generalize well and should be reconsidered in the future.

For the sake of simplicity, the value $\epsilon$ is always set to $h$ accepting the possiblity of missing the optimal configuration.

To summarize, the model-based prediction method works as follows: A small number of configurations with maximum neighborhood radius, minimum $\delta t$, $\epsilon = 1$ and varying $h$ are measured for each discretization method. The results are used to determine $a$ in $e = a \cdot h^2$ using linear regression for all methods. Additionally, a small number of configurations with maximum neighborhood radius, minimum $h$, $\epsilon = 1$ and varying $\delta t$ is conducted for PSE. The results are used to determine $a$ in $e = a \cdot \delta t$. The functions determined this way are reversed and used to calculate a value for $\delta t$ and one for $h$ for each discretization method using the error threshold. The neighborhood radius $r_c$ is

determined using the aforementioned heuristic and the kernel width is set to one. This way, there exists one candidate configuration per discretization method. For all of them, the runtime is measured over the evaluation period and the fastest one is chosen.

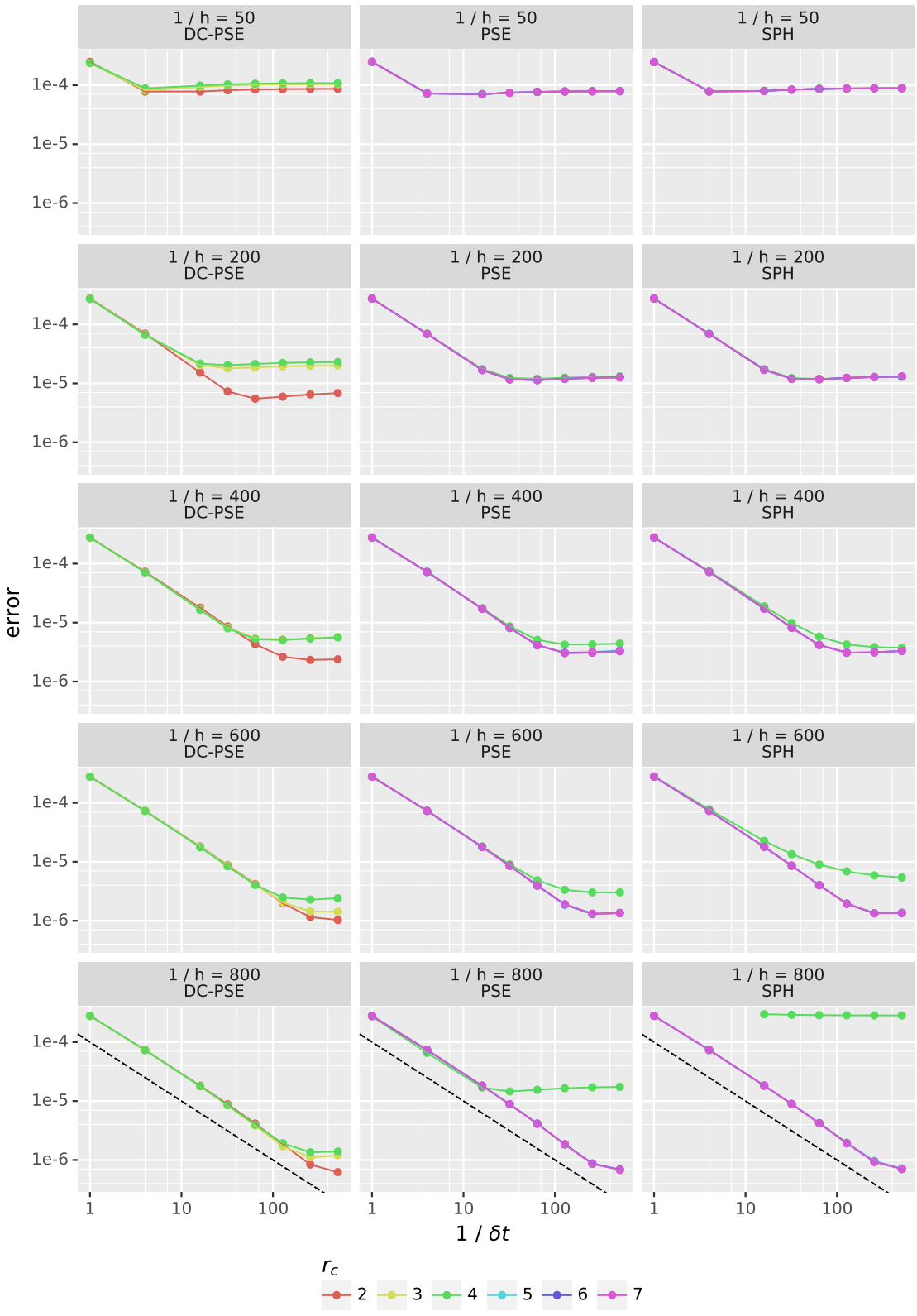



Figure 5.7.: Gray-Scott parameter study relating the error, temporal and spatial resolution and discretization method. The dashed line has a slope of -1 representing linear relations in the logarithmic diagram.

## 5.4. Runtime Regression

The previous sections explored the idea of using an analytical model in conjunction with regression over sample measurements to determine the relation between the configurations and the error. The same can be done for the computational costs. Essentially, the runtime scales with the number of particle interactions $i$. This in turn depends on the the total number of particles, the number of particles within the neighborhood of each other and the number of time steps. It can be calculated with

$$i = N^n \cdot \left(\frac{3 \cdot r_c}{h}\right)^n \cdot \frac{t}{\delta t}$$

where $N$ is the number of particles in each dimension in a $n$-dimensional space, so $N^n$ denotes the total number of particles. Figure 5.8 explores the actual measured relation between the number of particle interactions and runtime. The configurations measured vary in the number of particles, time steps, and neighborhood radii.

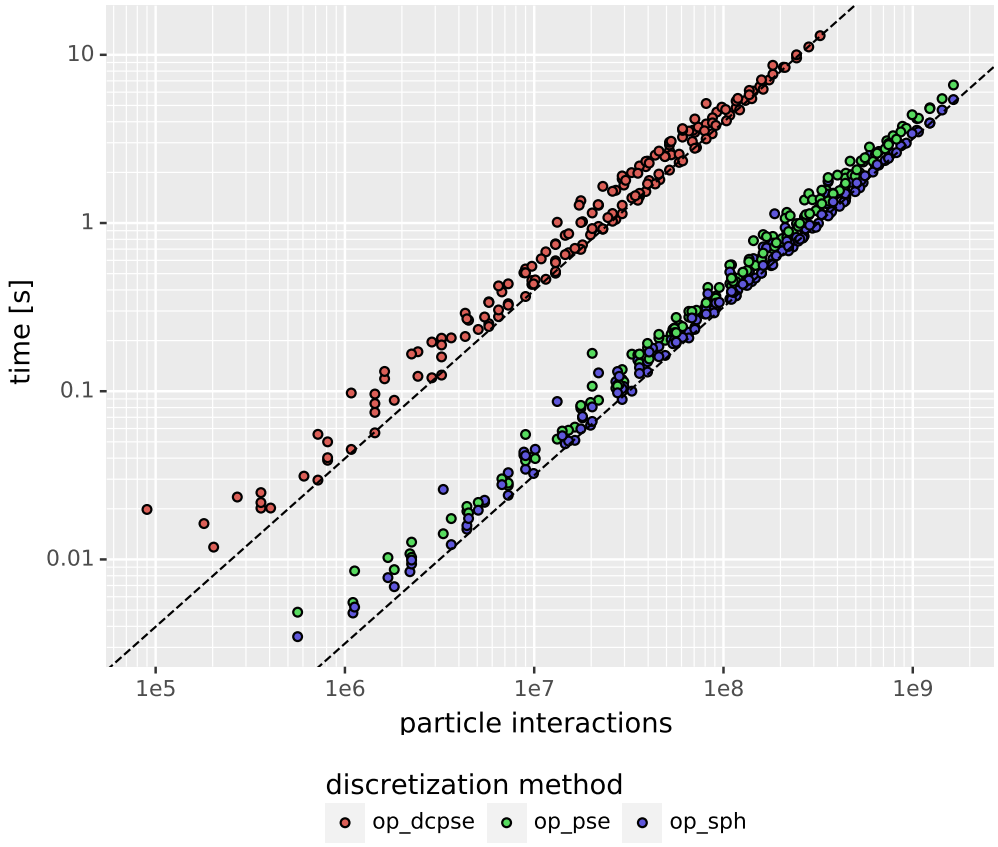



Figure 5.8.: Relation between the number of particle interactions and runtime. The dashed lines have a slope of one and denote linear relationships in the logarithmic diagram.

SPH and PSE show a clear linear growth. For DC-PSE, one should not expect a full linear growth, since the computation of the weights (solving the system of linear equations) does not linearly depend on the number of particle interactions per particle, so the influence of $r_c$ is not linear. Nevertheless, the measurements show to be close enough for an approximation.

Using these observations, the runtime can be predicted using linear regression on a small set of measurements for each discretization method. A similar approach was proposed by Schrader et al. [57]. To reduce the effect of time measurement inaccuracies, only configurations with a runtime above one seconds are considered.

The resulting runtime model could be used in conjunction with the diffrent methods described in this thesis. The empirical autotuners could benefit from skipping the actual evaluation of time consuming simulations. If used together with the model-based prediction approach, the whole trade-off between runtime and accuracy could be presented to the user.

Although the runtime regression approach is implemented and performs perfectly, it is currently not utilized by any of the aforementioned optimization techniques.

## 5.5. Separate Spatial Optimization

Evaluating the influence of the spatial discretization of a simulation separately from the temporal discretization enables a rapid tuning approach. The idea here is not to measure over multiple time steps but only the result of the discretized differential operator. Since the actual result is not needed for subsequent time steps, it is possible to approximate the accuracy by evaluating and measuring it for a representative subset of particles. The approach prohibits the measurement of the computational costs which can be compensated by the previously discussed runtime regression approach. The configurations are then optimized using any general optimization technique.

The advantage of this approach is that it does not require any analytical knowledge about the spatial discretization method and is by that easily transferred to changing tasks. To be used for the optimization problems at hand, it has to be combined with a technique determining a fitting temporal discretization method which is not in the scope of this thesis. Indeed, using the techniques from the model-based prediction approach (Section 5.3) is promising.

# 6. Evaluation

The empirical evaluation aims at answering multiple questions. How well do the available optimization algorithms perform in terms of the found solution and required tuning time? How profitable are the autotuning approaches in general comparing the tuning time with the gain in simulation time improvement?

All measurements are conducted on the HPC cluster of the Centre for Information Services and High Performance Computing (ZIH) of TU Dresden. Intel Haswell nodes with two Intel Xeon E5-2680v3 CPUs with 12 cores each at 2.50 GHz were used. The 24 core nodes have 64 GB of RAM available and are connected by an InfiniBand network with a 40Gb/s bandwidth. Even though OpenFPM code has shown to scale well for large amounts of cores [37], single nodes are used for the measurements in this chapter. This allows for a wider evaluation within the limited access to the HPC environment.

The DC-PSE implementation used in this thesis does not employ any optimizations like remembering kernel functions for stationary particles or lookup tables to avoid the construction of the smoothing kernel for each particle in each time step. For this reason, the computational costs are considerably higher than for SPH and PSE. Due to this, one cannot expect to find configurations using DC-PSE which outperform any configurations found by the other two methods. Besides, the tuning takes significantly (up to four times) longer due to slow evaluations of single configurations. Therefore, the tuning runtimes of DC-PSE where neglected in this evaluation to not obfuscate the times measured for SPH and PSE.

## 6.1. Comparison of Optimization Techniques

This section will compare the achieved optimizations from the different available techniques. These are the model-based search described in Section 5.2, OpenTuner's main composed optimization technique and a wide selection of individual techniques implemented in the OpenTuner framework as listed in Table 6.1. The model-based prediction

described in Section 5.3 is evaluated separately as it does not guarantee to find a configuration with an error below the threshold. The presented runtime measurements are the mean over ten executions of the optimization algorithms.

| Technique | Short name | OpenTuner name |
|---|---|---|
| Model-based search | AccReg | AccuracyRegression |
| OpenTuner default search | AUC | AUCBanditMetaTechniqueA |
| Differential Evolution | DE | DifferentialEvolution |
| Genetic Algorithm | GA | ga-base |
| Greedy Mutation | UniGM | UniformGreedyMutation10 |
| | NormGM | NormalGreedyMutation10 |
| Nelder Mead | RegNM | RegularNelderMead |
| | MulNM | MultiNelderMead |
| Particle Swarm Optimization | PSO | pso-PMX |
| Pseudo Annealing | Anneal | PseudoAnnealingSearch |
| Torczon Hill Climber | RTorc | RegularTorczon |
| | MTorc | MultiTorczon |
| Random search | Rand | PureRandom |

Table 6.1.: Overview of the evaluated optimization techniques. The first column names the general technique. Second column states the short name used in diagrams. Third column states the technique name passed to OpenTuner via `--technique`.

The search space used for the evaluations is given in Table 6.2. It contains 798 720 configurations. The measurement of a single configuration at $t = 2$ takes between 0.003 and 735 seconds. This is what renders optimization challenging as in the worst case an optimizer is not able to measure even five configurations within an hour. For the accuracy measurement, a configuration using PSE with a Gaussian smoothing kernel of width $\frac{\epsilon}{h} = 1$ with $h = \frac{1}{1600}$, $\delta t = \frac{1}{1024}$ and $\frac{r_c}{h} = 9$ is employed.

### 6.1.1. Diffusion Simulation

For the diffusion simulation, the aforementioned optimization algorithms where used to tune the runtime while keeping the measured error below a threshold of $10^{-5}$ as shown in Figure 6.1 and $10^{-6}$ as in Figure 6.2. As measurement point $t = 2$ was chosen and the optimization was limited to 0.5, 1 and 2 times the runtime of the model-based search approach for the given problem. Since a separate tuning is conducted for all

| Discretization method | Smoothing kernel | Parameter | Measured values |
|---|---|---|---|
| | all | $\frac{1}{h}$ | $\{50, 100, 150, \ldots, 800\}$ |
| | | $\frac{1}{\delta t}$ | $\{1, 2, 3, 4, \ldots, 256\}$ |
| | | $\frac{\epsilon}{h}$ | $\{0.6, 0.7.0.8, \ldots, 2.0\}$ |
| SPH | Gaussian | $\frac{r_c}{h}$ | $\{5, 6, 7, 8, 9\}$ |
| PSE | Gaussian | $\frac{r_c}{h}$ | $\{5, 6, 7, 8, 9\}$ |
| DC-PSE | Polynomial | $\frac{r_c}{h}$ | $\{2, 3, 4\}$ |

Table 6.2.: Search space used for autotuning the diffusion and Gray-Scott simulations.

discretization methods, these limitations apply per method. The runtime of the model-based search approach is defined by $t_{AccReg} = \max(t_{AccReg\_SPH}, t_{AccReg\_PSE})$.

**Threshold $10^{-5}$**

In the case of an error threshold of $10^{-5}$, the model-based search approach took on average 277 s for PSE and 166s for SPH. For PSE and DC-PSE, the tuning consisted of 15 measurements and 14 for SPH. Since the method is deterministic, the number of steps did not vary. The found configuration (SPH, $\frac{1}{h} = 200$, $\frac{\epsilon}{h} = 0.9$, $\delta t = 1$, $\frac{r_c}{h} = 5$) took 0.06 s for period $t = 0$ to $t = 2$, so the whole simulation would take 155 s. As Figure 6.1 shows, the other optimization algorithms found considerably slower configurations even within double the tuning time. There, OpenTuner's default search technique performed best with an average runtime over the tuning period of 3.84 s (9 598 s simulation runtime). Since the other optimization techniques are nondeterministic, their performance varied between tuning runs. Normal greedy mutation found the same configuration in one run and found configurations taking less than 0.1 s in four out of ten runs. To get a better insight on the variation in between the individual tuning runs, a box plot can be found in the appendix (Figure B.1).

**Threshold $10^{-6}$**

With an error threshold of $10^{-6}$, the model-based search for PSE took 1 765 s on average whereas SPH needed 1 216 s for 14 to 15 steps, respectively. The found configuration (SPH, $\frac{1}{h} = 600$, $\frac{\epsilon}{h} = 1.0$, $\frac{1}{\delta t} = 4$, $\frac{r_c}{h} = 5$) takes 1.00 s for the tuning period (2 500 s full simulation). In this case, some other optimization algorithms were consistently able to outperform the model-based search when given double the tuning time, but never achieved this within the same amount of time. Particle swarm optimizations scored an average found runtime of 0.19 s (467 s full simulation). Despite the fact that this is five times faster, it does not amortize the increased tuning time.

The best individual configuration found by other algorithms was (PSE, $\frac{1}{h} = 250$, $\frac{\epsilon}{h} = 0.6$, $\delta t = 1$, $\frac{r_c}{h} = 5$). There are two aspects that should be noted about this configuration. Firstly, a very high time steps size was chosen despite the low target error. In the parameter study, Figure 5.5 showed low error spikes for high values of $\delta t$. Since they were interpreted as unfavorable, the model-based search actively avoids them. Secondly, the width of the smoothing kernel is surprisingly small but does seem to be the main reason for the low error that would not be explainable with other parameters. This however contradicts the parameter study done by Schrader et al. [57] who found that higher kernel widths lead to higher accuracies (assuming sufficiently large $r_c$ which is given in this case). Based on these observations one would assume that the high accuracies are only artifacts of the measurement at an early point during the simulation. However, first tests inspecting other time points were not able to confirm this theory. Further evaluations are necessary to investigate this matter.

### Model-Based Prediction

Besides the search based optimization techniques, this thesis presented a model-based prediction approach. As data points for the regression, three values per prediction (temporal, spatial SPH, spatial PSE, spatial DC-PSE) were measured evenly distributed within the lower quarter of the search space and took a total of 98 s for all 12 measurements. The configuration used to measure the accuracy was also reduced to a quarter for the spatial resolution and time step size which took 133 s to be created. After this initialization, predictions for different target accuracies barely take any noticeable time. For the spatial resolution, the regression found a function with a coefficient of determination of $R^2 = 0.998$ for all three discretization methods[1]. For the temporal resolution however, the regression was not able to find a function at all ($R^2 = -47$). This can be attributed to the fact that $\delta t$ did not seem to have a large influence on the accuracy for diffusion as reported in the parameter studies (Figure 5.5). Interpreting the negative coefficient of determination as no influence by the temporal resolution, a constant value of $\delta t = 1$ is chosen. The predicted configurations for an error threshold of $10^{-5}$ took 0.08 s for the tuning period and 0.50 s with a threshold of $10^{-6}$. In both cases the error threshold was met. The resulting runtimes were close to or even superior to those of the model-based search. However, it should be noted that this might be attributed to the fact that the chosen configurations partially fall into the previously discussed low error spikes whose correctness stays an open question.

[1] A coefficient of determination of one denotes a perfect fit. The value can also be negative.

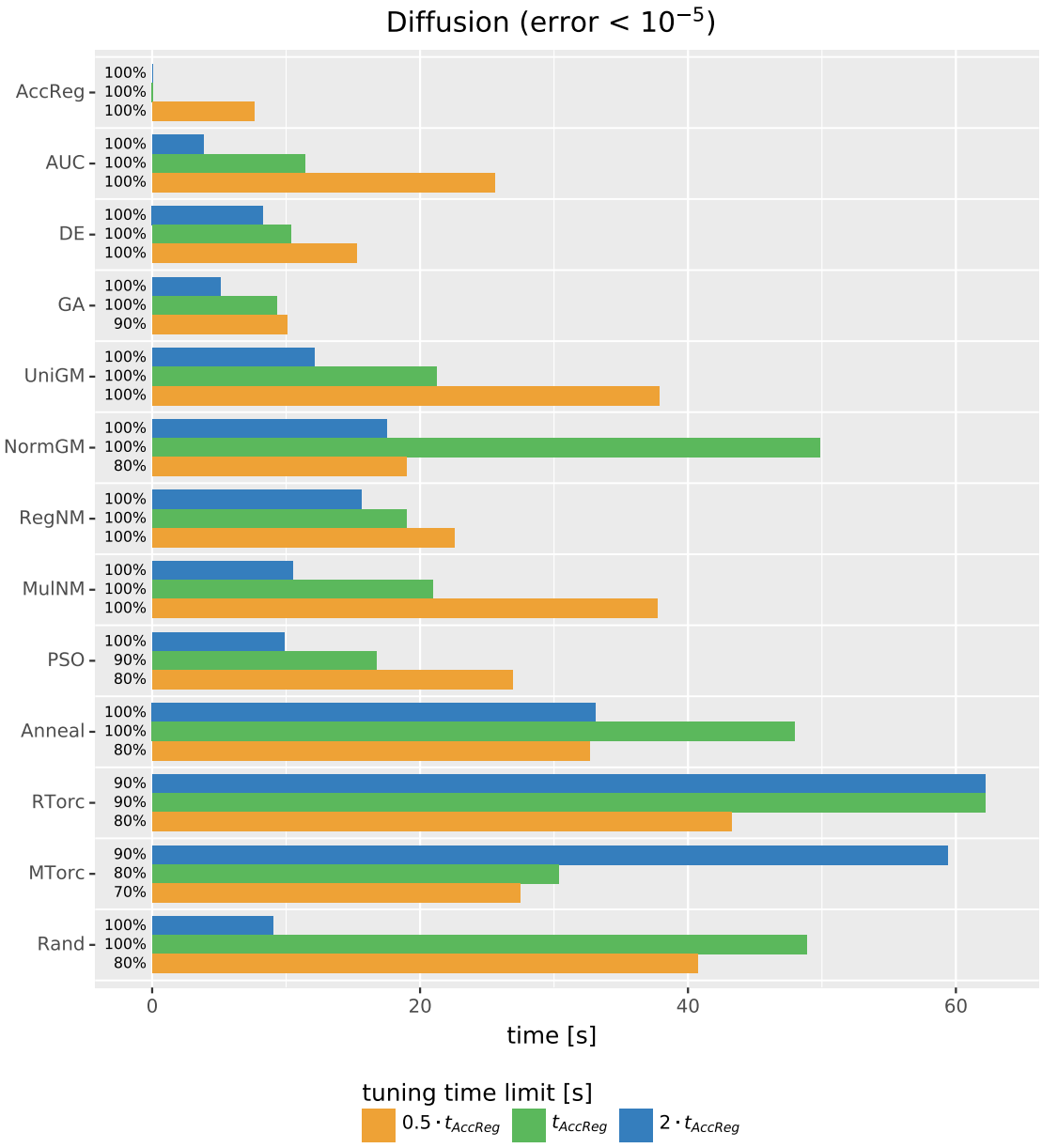



Figure 6.1.: Achieved simulation runtimes by different optimization algorithms for the diffusion simulation with an error threshold of $10^{-5}$. The runtime (x-axis) refers to the measured time over the tuning period ($t = 0$ to $t = 2$). The bars represent the mean over 10 runs. Each algorithm is evaluated with an upper tuning time limit of 0.5, 1 and 2 times the tuning time of the model-based search. The percentage on the left indicates how often any valid configuration below the error threshold was found.

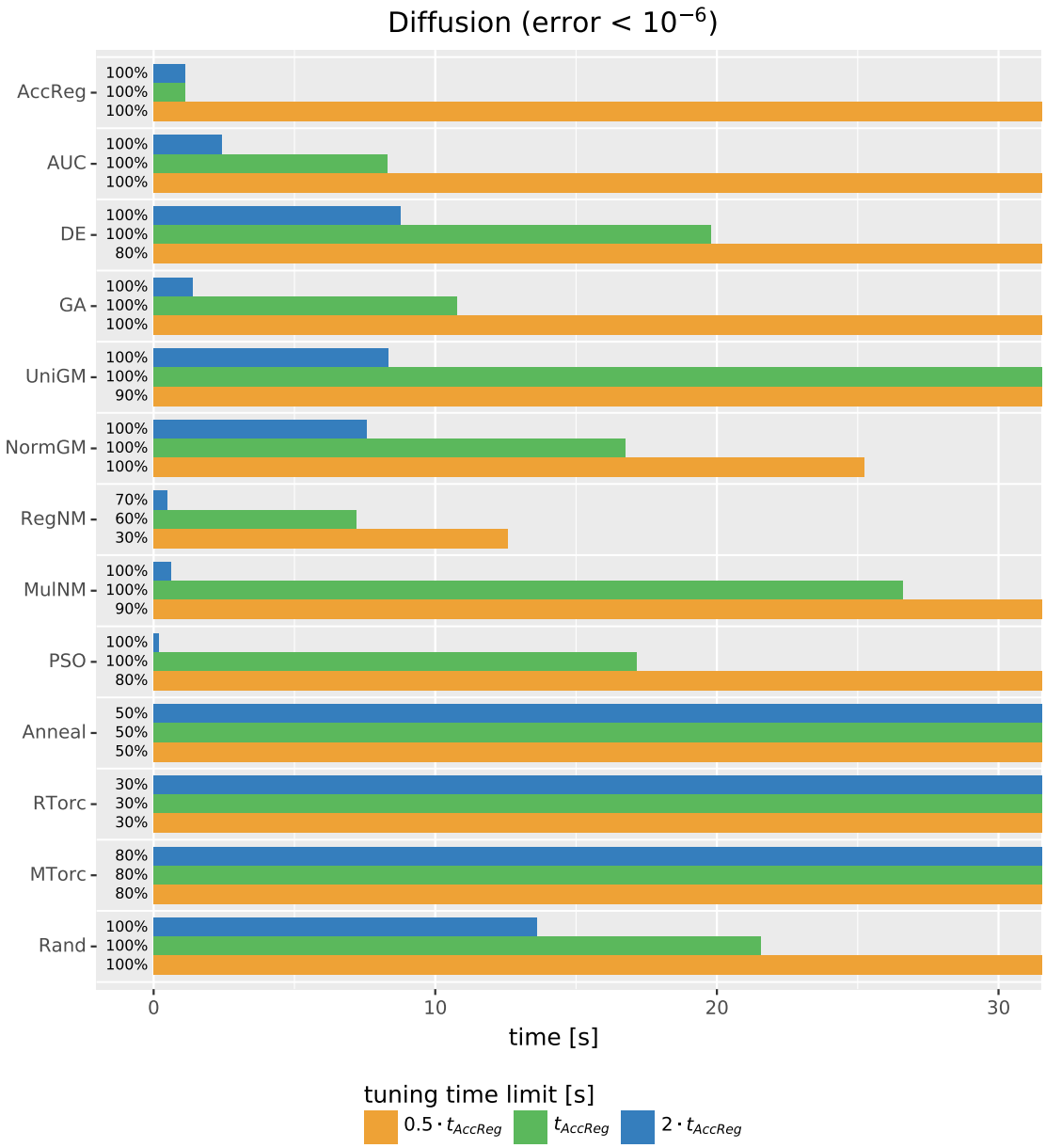



Figure 6.2.: Achieved simulation runtimes by different optimization algorithms for the diffusion simulation with an error threshold of $10^{-6}$. The runtime (x-axis) refers to the measured time over the tuning period ($t = 0$ to $t = 2$). The bars represent the mean over 10 runs. Each algorithm is evaluated with an upper tuning time limit of 0.5, 1 and 2 times the tuning time of the model-based search. The percentage on the left indicates how often any valid configuration below the error threshold was found.

### 6.1.2. Gray-Scott Simulation

As in the diffusion simulation, the optimization algorithms for the Gray-Scott simulations are evaluated for error thresholds of $10^{-5}$ and $10^{-6}$ and the evaluation point is $t = 2$. Additionally, the tuning times were again limited to 0.5, 1 and 2 times the tuning time of the model-based search.

#### Threshold $10^{-5}$

For an error threshold of $10^{-5}$, the model-based search took on average 1 114 s over 16 steps for PSE and 584 s over 15 steps for SPH. The found configuration (SPH, $\frac{1}{h} = 250$, $\frac{\epsilon}{h} = 0.9$, $\frac{1}{\delta t} = 30$, $\frac{r_c}{h} = 5$) took 2.94 s (14 700 s full simulation). The general optimization algorithms performed considerably worse on average, even if given the double amount of tuning time. The best average runtime found is 7.95 s (39 750 s full simulation) by normal greedy mutation. In individual runs, normal greedy mutation was able to outperform the model-based search in six out of ten runs given double the tuning time. The best configuration found this way takes 1.18 s (5 917 s full simulation). Hence, it would be able to amortize the increased tuning costs if the result was found consistently. Nonetheless, it has to be noted that these superior configurations have all used PSE with $\frac{\epsilon}{h} = 0.6$. The issue with this was discussed before in Section 6.1.1. For more insights on the variation between the individual tuning runs, a box plot can be found in Figure B.3.

#### Threshold $10^{-6}$

For an error threshold of $10^{-6}$, the tuning runtimes and resulting computational cost for the simulations were infeasibly long. For this reason, the evaluation was only performed five (and not ten) times and excluded the tuning time limit of $2 \cdot t_{AccReg}$. The model-based search took 13 426 s over 15 steps for PSE and 12 340 s over 17 steps for SPH. The found configuration (PSE, $\frac{1}{h} = 750$, $\frac{\epsilon}{h} = 0.6$, $\frac{1}{\delta t} = 219$, $\frac{r_c}{h} = 5$) took 211 s (12 days full simulation). As Figure 6.4 shows, multiple general search algorithms were able to outperform the model-based search, although most of them were not able to consistently find a configuration which fulfills the threshold and even none of them outperformed the model-based search in all runs.

#### Model-Based Prediction

Similarly to diffusion, the regression used three measured values per prediction which were evenly spread across the lower quarter of the search space. Their evaluation took a total of 196 s. This is double the time required for diffusion since two fields ($u$ and $v$) are simulated, whereas in diffusion it is only one. The configuration used to measure accuracy was created in 268 s. The temporal model was fitted with a coefficient of determination of $R^2 = 0.99$ and the spatial models for SPH, PSE and DC-PSE one with $0.95$, $0.96$ and $0.84$, respectively. The predicted configuration for an error threshold of

$10^{-5}$ takes 2.00 s but only achieves an error of $1.4 \cdot 10^{-5}$. For the error threshold $10^{-6}$, the predicted configuration has a runtime of 235 s but also exceeds the error threshold with an error of $1.4 \cdot 10^{-6}$. The achieved performance compares well with the considerably more computationally expensive search methods. The error levels above the threshold are possible since on the one hand the method is not exact and on the other hand, the spatial and temporal errors may add up. To counteract this, one could target a lower error level to begin with.

### 6.1.3. General Observations

The model-based search showed consistency in the number of needed tuning steps which varied between 14 and 17. In contrast to this, the tuning time varied strongly depending on the computational costs of the final simulation and this is because most of the measurements are performed on configurations that rapidly approach the final one. The quality of the general optimization algorithms seemed to depend less on the specified target since they performed better in comparison when given long tuning times to find long running configurations.

Out of the general optimization algorithms, none was able to compete with the model-based search in all experiments. Among of them, OpenTuner's AUC Bandit technique and the genetic algorithm were able to achieve good results most reliably regarding all experiments.

The model-based prediction performed astonishingly well considering its strongly reduced tuning time. Nonetheless, one has to keep in mind that it does not guarantee to stay below the error threshold.

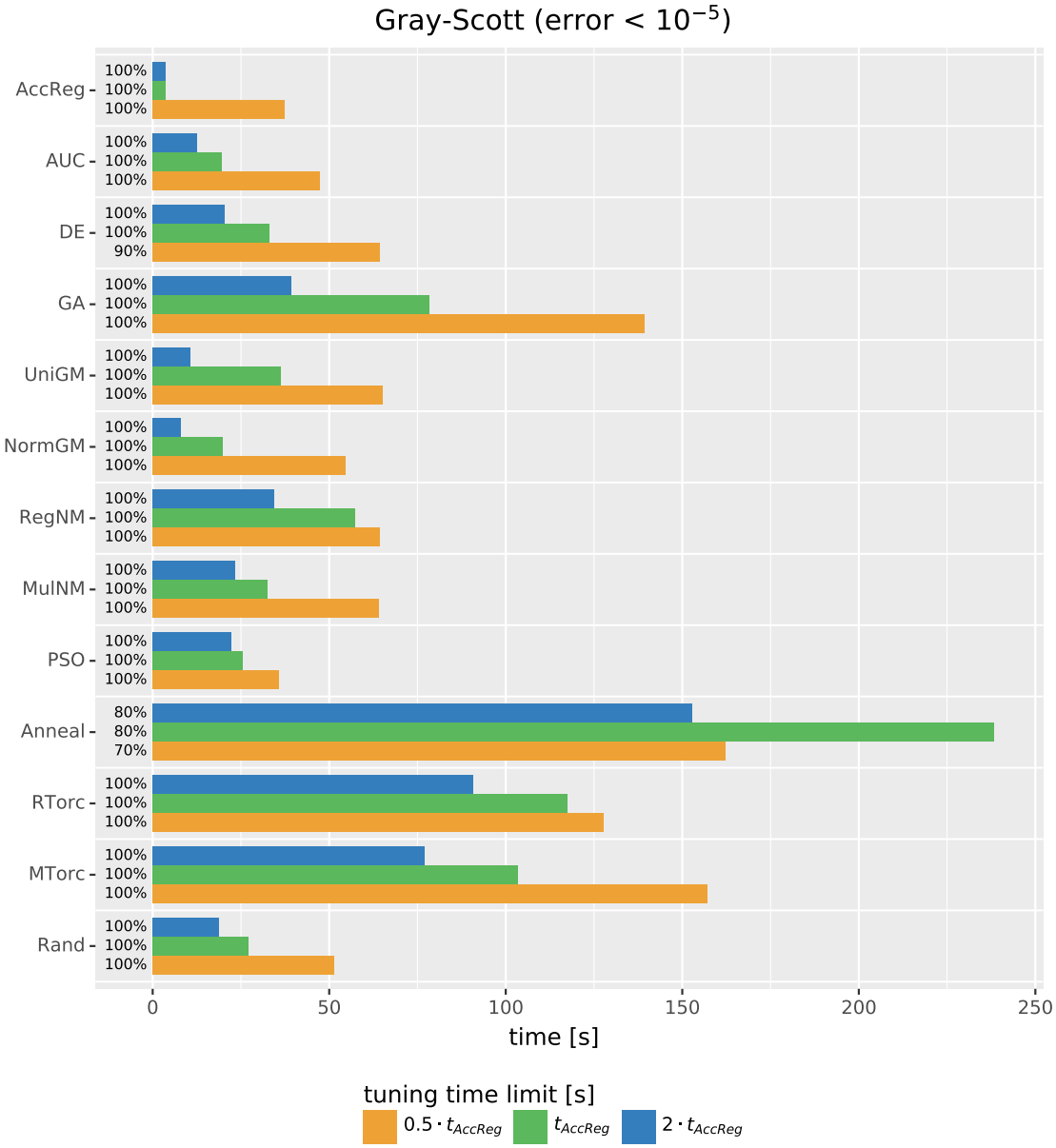



Figure 6.3.: Achieved simulation runtimes by different optimization algorithms for the Gray-Scott simulation with an error threshold of $10^{-5}$. The runtime (x-axis) refers to the measured time over the tuning period ($t = 0$ to $t = 2$). The bars represent the mean over 10 runs. Each algorithm is evaluated with an upper tuning time limit of 0.5, 1 and 2 times the tuning time of the model-based search. The percentage on the left indicates how often any valid configuration below the error threshold was found.

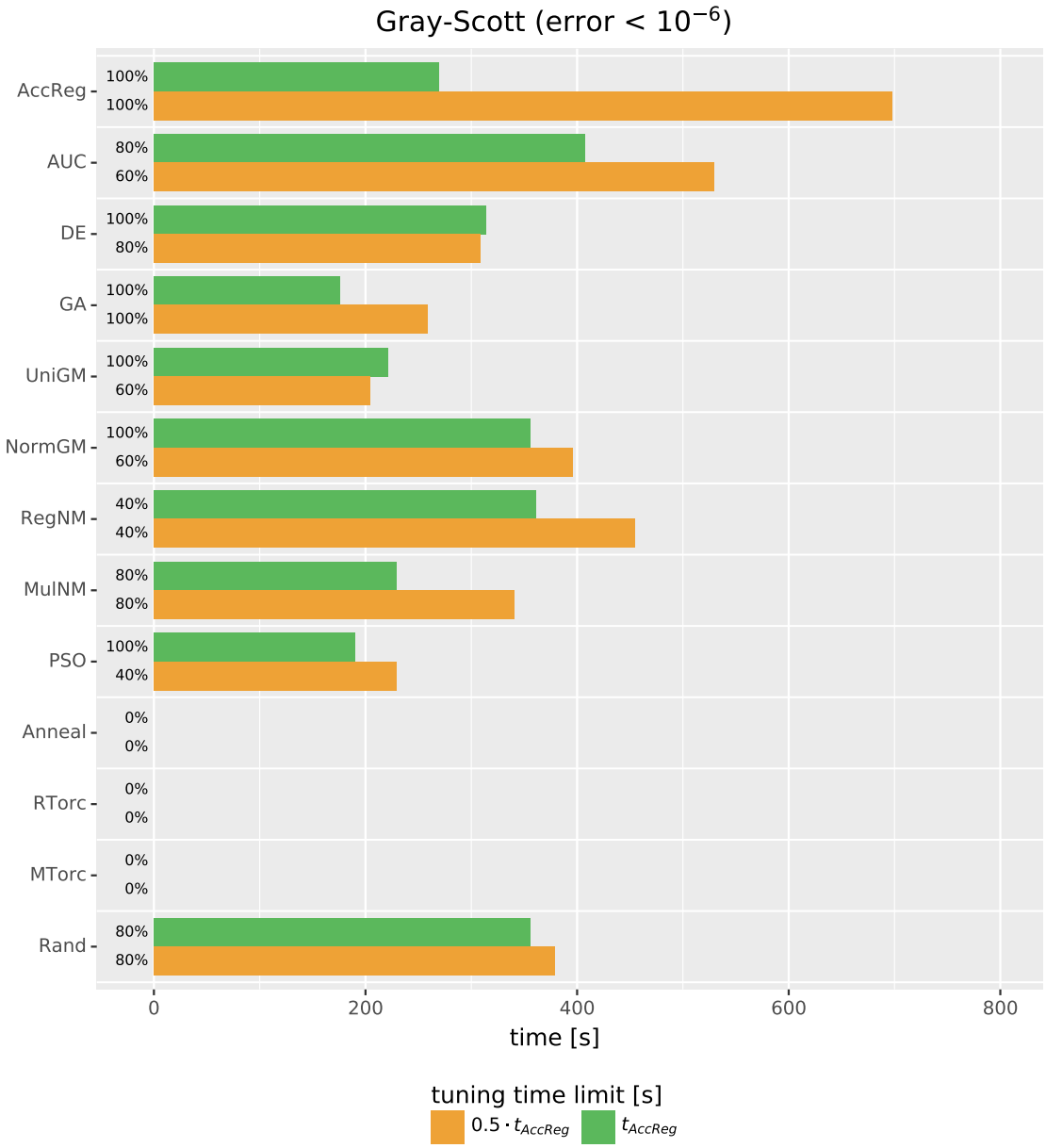



Figure 6.4.: Achieved simulation runtimes by different optimization algorithms for the Gray-Scott simulation with an error threshold of $10^{-6}$. The runtime (x-axis) refers to the measured time over the tuning period ($t = 0$ to $t = 2$). The bars represent the mean over 5 runs. Each algorithm is evaluated with an upper tuning time limit of 0.5 and 1 times the tuning time of the model-based search. The percentage on the left indicates how often any valid configuration below the error threshold was found.

## 6.2. Profitability Analysis

The previous section showed that some of the available optimization algorithms are capable to find highly suitable configurations according to the optimization goals. This section will investigate if the tuning is generally profitable. The main question here is whether the performance gains are able to compensate for the costs introduced by the required tuning phase. This question is not easily answered as it is not obvious what should be compared here, since there exists no base configuration that could be used to calculate the performance improvement. Additionally, the computational costs of the tuning highly depend on the chosen search space, the reference configuration for accuracy measurements and the size of the measurement period used for the tuning. These parameters are assumed to be as in the previous section for the evaluation.

Since no single obvious comparison is available, multiple ones will be considered. On the tuning time side, the time needed by the best performing algorithms will be evaluated for both simulations and target accuracies. On the simulation runtime side, the best configuration consistently found the accuracy reference configuration and the best configuration for the respectively other problem will be considered. The reasoning for this choice is argued in the following paragraphs.

The best configuration consistently found represents the lowest simulation runtime available. By that, it sets the lower bound on the simulation runtime spectrum. However, one should not forget that this configuration would most likely not be available without autotuning or be the result of even longer manual tuning work made by the programmer.

The accuracy reference configuration sets an upper bound on the simulation runtime. It is the only configuration that is assumed to fulfill the accuracy requirements beforehand by the autotuner. To ensure that it is sufficiently accurate as reference for accuracy evaluations, it is computationally very costly and most likely not feasible as configuration for the whole simulation. Otherwise, the tuning could be omitted since this configuration is already assumed to be accurate enough.

The best configuration for the other simulation represents an intuitive choice by a human trying to transfer knowledge from a known problem to a new one. For this comparison, not only the resulting runtime has to be considered but also the achieved accuracy since it is not guaranteed that this configuration fulfills the accuracy requirements.

Table 6.3 lists the tuning time requirements and the time to create the reference configurations for the given search space. Table 6.4 lists the resulting runtime of the full simulations in the three previously defined categories.

For the diffusion simulation, the tuning is strongly dominated by the time to run the reference configuration which is also significantly longer than the final runtime of the best

| | Threshold | Search | | | Prediction | |
|---|---|---|---|---|---|---|
| | | Reference | AccReg | Others | Reference | Prediction |
| Diffusion | $10^{-5}$ | 8441 s | 443 s | 1108 s | 133 s | 98 s |
| | $10^{-6}$ | 8441 s | 1216 s | 3530 s | 133 s | 98 s |
| Gray-Scott | $10^{-5}$ | 17015 s | 1698 s | 4456 s | 268 s | 196 s |
| | $10^{-6}$ | 17015 s | 25767s | 26853 s | 268 s | 196 s |

Table 6.3.: Tuning runtimes.

| | Threshold | Reference | Best own | Best other | |
|---|---|---|---|---|---|
| | | | | Time | Error |
| Diffusion | $10^{-5}$ | 244 days | 155 s | 3707 s | $4.3 \cdot 10^{-6}$ |
| | $10^{-6}$ | 244 days | 467 s | 5 days | $5.4 \cdot 10^{-7}$ |
| Gray-Scott | $10^{-5}$ | 984 days | 14700 s | 590 s | $2.6 \cdot 10^{-4}$ |
| | $10^{-6}$ | 984 days | 12 days | 834 s | unstable |

Table 6.4.: Simulation runtimes.

configuration. For an error threshold of $10^{-5}$ it exceeds the runtime of the suboptimal configuration representing a user choice (best configuration from the other simulation). For an error threshold of $10^{-6}$ it does not. Only the model-based prediction approach compares well even including its accuracy reference configuration.

For Gray-Scott, the best simulation runtimes found, were similar to the tuning times for a threshold of $10^{-5}$ but considerably longer for $10^{-6}$. The representation of the user choice did lead to configurations that were either inaccurate or even unstable. Nonetheless, the low accuracy might not have been detected without the measurement of the comparison configuration. This shows that in addition to its necessity for the tuning, the comparison configuration or something similar would be needed to measure the accuracy when optimizing by hand.

Comparing the tuning time with the simulation runtime of the accuracy reference configuration (*Reference* column in Table 6.4) visualizes how much the choice of an appropriate configuration matters since the runtime does not only vary by small margins but over multiple orders of magnitude.

Finally, it should be noted that all tuning methods did not take more than few hours, even including the creation of the reference configuration which is certainly fast in comparison to manual optimization.

# 7. Conclusion

This thesis designed, implemented and evaluated a multi-objective autotuning approach for particle simulations written in OpenPME. It implemented an interface to easily generate C++ code linked with the OpenFPM library for the spatial discretization methods SPH, PSE and DC-PSE. Multiple methods to measure accuracy and runtime as well as to combine them into a single-objective were explored and implemented. Ultimately, the method of fixing the error while optimizing the computational cost is used.

To find the best performing configuration regarding this objective, this work utilized the autotuning framework OpenTuner which provides a large variety of general purpose optimization algorithms. Additionally, multiple approaches involving domain specific knowledge were designed. The model-based search uses the knowledge of the convergence order of spatial discretization methods in conjunction with regression to guide an empirical search. The model-based prediction uses the same idea combined with the knowledge about the convergence order of the temporal discretization method to predict a configuration without conducting a search.

The performance of the implemented optimization algorithms were evaluated for diffusion and Gray-Scott simulations with different target accuracies. The evaluations showed that especially the model-based approaches were able to perform well throughout all considered scenarios. The model-based prediction approach was able to conduct the autotuning within a few minutes while the search-based optimizations took considerably longer due to the need of a computationally intensive reference configuration to evaluate the accuracy. However, the search itself showed to reliably only require 14 to 17 steps for the evaluated search space. While the general optimization algorithms were outperformed by the model-based ones, they showed some potential when given enough time. Their major advantage is that they are, due to their generality, easily adapted to changing problem spaces. The evaluations also showed the large potential of autotuning by quantifying how strongly the choice of configuration parameters influences the resulting runtime and error. Both vary over multiple orders of magnitude where the final decision is partially a trade-off but most importantly involves choosing the individual

parameters in the correct proportions to each other since not every increase in computational cost leads to an improved accuracy. The correct proportions showed to be strongly dependent on the problem which justifies the applications of autotuning over default decisions.

Conclusively, the developed autotuning system proved to be highly valuable as an addition for particle DSLs in general and OpenPME specifically. Nonetheless, autotuning should never replace the possibility of direct specification of the parameters if the programmer intends to do so. The autotuner can not guarantee to find the best configuration, the tuning takes time and the parameters may have other properties which are not captured by the error and runtime measurements. One such feature is the symmetry of the PSE operator which guarantees that quantities are always only transferred between particles and neither created nor lost.

## 7.1. Future Work

While the autotuning system described in this work is already highly performant, there are still optimization opportunities open for exploration, new problem spaces to be adapted to and steps open in the full integration into OpenPME.

### 7.1.1. Integration into OpenPME

This thesis designed the autotuning system in a way, that it is easily integrated into OpenPME but could not do the last steps since OpenPME is under heavy development currently and not quite ready for the integration. The three steps OpenPME would have to perform are:

1. Generate the tuning configuration json file. It includes the search space definition and the accuracy reference configuration. Additionally, it defines the length of the measurement period and the error threshold. This json file is parsed and interpreted by the optimization implementations used in this work.

2. Read a runtime configuration file defining the chosen values for the tuning parameters in the generated C++ code as done in the implementation of the diffusion and Gray-Scott simulations in this work.

3. Deploy the found final configuration either through direct integration or in a final runtime configuration file.

Optionally, the call to the optimizers could be integrated into the OpenPME build process but it would also be conceivable to leave it as a separate call.

### 7.1.2. Evaluations on a Larger Number of Simulations

In this thesis, the optimization techniques were evaluated for diffusion and Gray-Scott simulations. Evaluating them on a wider range of problems would be highly valuable to test their generality or to expand them if necessary. In addition to completely different simulation types, it would be interesting to include diffusion and Gray-Scott simulations in 3D and with moving particles leading to irregular distributions. Schrader et al. [57] showed that parameter studies made on uniform Cartesian particle distributions do indeed have the potential to transfer well onto irregular ones.

### 7.1.3. Integration of More Tuning Parameter

This work is limited to tuning the discretization scheme, smoothing kernel, $\epsilon$, $h$ and $\delta t$. Nonetheless, it would also be possible to include more parameters. One could tune between multiple temporal discretization schemes as the explicit Euler, Leapfrog, Runge-Kutta-4 or implicit Euler. Furthermore, it would also be imaginable to include the floating point accuracy of particle positions and attributes as tuning parameter. Especially since OpenPME currently does not have a typesystem that would allow to define it manually.

### 7.1.4. Threshold Computational Costs

The optimization techniques discussed in this thesis assumed the strategy of fixing the error and optimizing the simulation runtime. Exchanging the roles would also be a viable option. Fixing a target runtime and optimizing for the best possible accuracy does pay attention to the fact that programmers usually have relatively clear expectations on how long a simulation should take. The possible computational costs, however, vary over multiple orders of magnitude.

An optimization technique that searches for the best accuracy while putting a threshold on the allowed computational costs could use the runtime regression approach presented in Section 5.4 to prune the search space.

### 7.1.5. Search Space Pruning Using Predictive Models

In this thesis, predictive models were used to guide a search approach and to directly predict the best configuration. Alternatively, they could be used to narrow the search space and follow up with a classic search approach. This combines the speed of a predictive approach with the exactness of an actual search. The idea additionally has the potential of eliminating issues the model-based approaches have with parameters which are challenging to predict because either no model for them was implemented yet or their interactions are generally more difficult or even impossible to predict. In this work, the fully predictive method fixes $\epsilon$ to a constant value potentially missing improvements. In an approach that prunes the search space, all values for $\epsilon$ could be considered while the ranges for $h$ and $\delta t$ are narrowed to a few possible values.

### 7.1.6. Transfer Knowledge between Simulations

With a wider range of simulations, it could be possible to correlate problem characteristics with performance behavior depending on the parameters. Characteristics could include gradients and rates of change of particle properties. Using a sufficiently large set of exhaustively benchmarked simulations (within some limits on the parameters), one could explore the idea of transferring knowledge between simulations. This could be used to predict optimal configurations without performing any measurements at all or choosing more narrow search spaces.

# A. Acronyms

**API** Application Programming Interface

**ATLAS** Automatically Tuned Linear Algebra Software

**AUC Bandit** area under the curve credit assignment

**ATF** Auto-Tuning Framework

**CUDA** Compute Unified Device Architecture

**DFT** discrete Fourier transform

**DSL** domain-specific language

**DC-PSE** discretization corrected PSE

**BLAS** Basic Linear Algebra Subprograms

**GPU** Graphics Processing Unit

**HPC** High-Performance Computing

**MPI** Message Passing Interface

**MPS** Meta Programming System

**OpenFPM** Open Framework for Particle Methods

**OpenPME** Open Particle Mesh Environment

**OpenMP** Open Multi-Processing

**PDE** partial differential equation

**PPM** Parallel Particle-Mesh

**PPML** Parallel Particle-Mesh Language

**PPME** Parallel Particle-Mesh Environment

**PSE** particle strength exchange

**Pthreads** POSIX Threads

**SPH** smoothed particle hydrodynamics

**SVM** Support Vector Machine

**ZIH** Centre for Information Services and High Performance Computing

# Bibliography


[1] Emile Aarts and Jan Korst. “Simulated annealing and Boltzmann machines”. In: (1988).

[2] Michael P Allen and Dominic J Tildesley. *Computer simulation of liquids*. Oxford university press, 1987.

[3] Jason Ansel et al. “Opentuner: An extensible framework for program autotuning”. In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pp. 303–316.

[4] Jason Ansel et al. “PetaBricks: a language and compiler for algorithmic choice”. In: *ACM Sigplan Notices* 44.6 (2009), pp. 38–49.

[5] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. “Finite-time analysis of the multiarmed bandit problem”. In: *Machine learning* 47.2-3 (2002), pp. 235–256.

[6] Omar Awile et al. “A domain-specific programming language for particle simulations on distributed-memory parallel computers”. In: *Proc. III Intl. Conf. Particle-based Methods (PARTICLES). Stuttgart, Germany*. 2013, p52.

[7] Utkarsh Ayachit. *The paraview guide: a parallel visualization application*. Kitware, Inc., 2015.

[8] Prasanna Balaprakash et al. “Autotuning in high-performance computing applications”. In: *Proceedings of the IEEE* 106.11 (2018), pp. 2068–2083.

[9] Russell R Barton and John S Ivey Jr. “Nelder-Mead simplex modifications for simulation optimization”. In: *Management Science* 42.7 (1996), pp. 954–973.

[10] David Beckingsale et al. “Apollo: Reusable models for fast, dynamic tuning of input-dependent code”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 307–316.

[11] James Bergstra and Yoshua Bengio. “Random search for hyper-parameter optimization”. In: *Journal of machine learning research* 13.Feb (2012), pp. 281–305.

[12] Francesco Biscani and Dario Izzo. *Pagmo*. 2020. URL: `https://esa.github.io/pagmo2/` (visited on Apr. 11, 2020).

[13] George C Bourantas et al. "Using DC PSE operator discretization in Eulerian meshless collocation methods improves their robustness in complex geometries". In: *Computers & Fluids* 136 (2016), pp. 285–300.

[14] Pedro Bruel, Marcos Amarís, and Alfredo Goldman. "Autotuning cuda compiler parameters for heterogeneous applications using the opentuner framework". In: *Concurrency and Computation: Practice and Experience* 29.22 (2017), e3973.

[15] Pedro Bruel, M Gonzalez, and Alfredo Goldman. "Autotuning gpu compiler parameters using opentuner". In: *XXII Symposium of Systems of High Performance Computing*. 2015.

[16] Pedro Bruel et al. "Autotuning high-level synthesis for FPGAs using OpenTuner and LegUp". In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2017, pp. 1–6.

[17] Edmund K Burke, Graham Kendall, et al. *Search methodologies*. Springer, 2005.

[18] Fabien Campagne. *The MPS language workbench: volume I*. Vol. 1. Fabien Campagne, 2014.

[19] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: A library for support vector machines". In: *ACM transactions on intelligent systems and technology (TIST)* 2.3 (2011), pp. 1–27.

[20] Chun Chen. *Model-guided empirical optimization for memory hierarchy*. University of Southern California, 2007.

[21] Kalyanmoy Deb. "Multi-objective optimization". In: *Search methodologies*. Springer, 2014, pp. 403–449.

[22] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. "A survey of parallel programming models and tools in the multi and many-core era". In: *IEEE Transactions on parallel and distributed systems* 23.8 (2012), pp. 1369–1386.

[23] Yufei Ding et al. "Autotuning algorithmic choice for input sensitivity". In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 379–390.

[24] Jeff D Eldredge, Anthony Leonard, and Tim Colonius. "A general deterministic treatment of derivatives in particle methods". In: *Journal of Computational Physics* 180.2 (2002), pp. 686–709.

[25] Stefan Falkner, Aaron Klein, and Frank Hutter. "BOHB: Robust and efficient hyperparameter optimization at scale". In: *arXiv preprint arXiv:1807.01774* (2018).

[26] Wilson Feng and Tarek S Abdelrahman. "A sampling based strategy to automatic performance tuning of GPU programs". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2017, pp. 1342–1349.

[27] Álvaro Fialho et al. “Comparison-based adaptive strategy selection with bandits in differential evolution”. In: *International Conference on Parallel Problem Solving from Nature*. Springer. 2010, pp. 194–203.

[28] Matteo Frigo and Steven G Johnson. “The design and implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231.

[29] Fuchang Gao and Lixing Han. “Implementing the Nelder-Mead simplex algorithm with adaptive parameters”. In: *Computational Optimization and Applications* 51.1 (2012), pp. 259–277.

[30] Google. *Google Test*. https://github.com/google/googletest/. 2020.

[31] P Gray and SK Scott. “Autocatalytic reactions in the isothermal, continuous stirred tank reactor: isolas and other forms of multistability”. In: *Chemical Engineering Science* 38.1 (1983), pp. 29–43.

[32] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2020.

[33] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. “Annotation-based empirical performance tuning using Orio”. In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–11.

[34] John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[35] Pietro Incardona. *OpenFPM DCPSE branch*. 2019. URL: `https://github.com/incardon/openfpm_pdata/tree/DCPSE/src/DCPSE` (visited on Mar. 22, 2020).

[36] Pietro Incardona. *Pagmo*. 2020. URL: `http://ppmcore.mpi-cbg.de/doxygen/openfpm/Grid_3_gs.html` (visited on May 3, 2020).

[37] Pietro Incardona et al. “OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers”. In: *Computer Physics Communications* 241 (2019), pp. 155–177.

[38] Sven Karol et al. “A domain-specific language and editor for parallel particle methods”. In: *ACM Transactions on Mathematical Software (TOMS)* 44.3 (2018), pp. 1–32.

[39] Sven Karol et al. “A language and development environment for parallel particle methods”. In: *International Conference on Particle-based Methods–Fundamentals and Applications, Hanover, Germany*. 2017, pp. 1–12.

[40] Nesrine Khouzami. *OpenPME*. URL: `https://github.com/Nesrinekh/OpenPME` (visited on May 14, 2020).

[41] Tamara G Kolda, Robert Michael Lewis, and Virginia Torczon. “Optimization by direct search: New perspectives on some classical and modern methods”. In: *SIAM review* 45.3 (2003), pp. 385–482.

[42] MB Liu and GR Liu. “Smoothed particle hydrodynamics (SPH): an overview and recent developments”. In: *Archives of computational methods in engineering* 17.1 (2010), pp. 25–76.

[43] Wing Kam Liu, Sukky Jun, and Yi Fei Zhang. “Reproducing kernel particle methods”. In: *International journal for numerical methods in fluids* 20.8-9 (1995), pp. 1081–1106.

[44] Marco A Luersen and Rodolphe Le Riche. “Globalized Nelder–Mead method for engineering optimization”. In: *Computers & structures* 82.23-26 (2004), pp. 2251–2260.

[45] J Matyas. “Random optimization”. In: *Automation and Remote control* 26.2 (1965), pp. 246–253.

[46] Saurav Muralidharan et al. “Nitro: A framework for adaptive code variant tuning”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 501–512.

[47] John A Nelder and Roger Mead. “A simplex method for function minimization”. In: *The computer journal* 7.4 (1965), pp. 308–313.

[48] Cedric Nugteren and Valeriu Codreanu. “CLTune: A generic auto-tuner for OpenCL kernels”. In: *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE. 2015, pp. 195–202.

[49] John E Pearson. “Complex patterns in a simple system”. In: *Science* 261.5118 (1993), pp. 189–192.

[50] Fabian Pedregosa et al. “Scikit-learn: Machine learning in Python”. In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[51] Nathan J Quinlan, Mihai Basa, and Martin Lastiwka. “Truncation error in mesh-free particle methods”. In: *International Journal for Numerical Methods in Engineering* 66.13 (2006), pp. 2064–2085.

[52] Ari Rasch and Sergei Gorlatch. “ATF: A generic directive-based auto-tuning framework”. In: *Concurrency and Computation: Practice and Experience* 31.5 (2019), e4423.

[53] Ivo F Sbalzarini. “Abstractions and middleware for petascale computing and beyond”. In: *International Journal of Distributed Systems and Technologies (IJDST)* 1.2 (2010), pp. 40–56.

[54] Ivo F Sbalzarini. *Particle Methods Lecture Notes*. URL: `https://mosaic.mpi-cbg.de/docs/PM/script.pdf` (visited on May 15, 2020).

[55] Ivo F Sbalzarini, Sibylle Müller, and Petros Koumoutsakos. “Multiobjective optimization using evolutionary algorithms”. In: *Proceedings of the summer Program*. Vol. 2000. 2000, pp. 63–74.

[56] Ivo F Sbalzarini et al. “PPM–A highly efficient parallel particle–mesh library for the simulation of continuum systems”. In: *Journal of Computational Physics* 215.2 (2006), pp. 566–588.

[57] Birte Schrader, Sylvain Reboux, and Ivo F Sbalzarini. “Choosing the best kernel: performance models for diffusion operators in particle methods”. In: *SIAM Journal on Scientific Computing* 34.3 (2012), A1607–A1634.

[58] Birte Schrader, Sylvain Reboux, and Ivo F Sbalzarini. “Discretization correction of general integral PSE Operators for particle methods”. In: *Journal of Computational Physics* 229.11 (2010), pp. 4159–4182.

[59] Francisco J Solis and Roger J-B Wets. “Minimization by random search techniques”. In: *Mathematics of operations research* 6.1 (1981), pp. 19–30.

[60] Ananta Tiwari et al. “A scalable auto-tuning framework for compiler optimization”. In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–12.

[61] TOP500.org. *TOP500 List*. 2019. URL: `https://www.top500.org/lists/2019/11/` (visited on Mar. 9, 2020).

[62] Rüdiger Verfürth. “Error estimates for a mixed finite element approximation of the Stokes equations”. In: *RAIRO. Analyse numérique* 18.2 (1984), pp. 175–182.

[63] Loup Verlet. “Computer” experiments” on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules”. In: *Physical review* 159.1 (1967), p. 98.

[64] Pauli Virtanen et al. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. In: *Nature methods* (2020), pp. 1–12.

[65] Markus Voelter and Vaclav Pech. “Language modularity with the MPS language workbench”. In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 1449–1450.

[66] Richard Vudac, James W Demmel, and Katherine A Yelick. “The optimized sparse kernel interface (OSKI) library: user’s guide for version 1.0. 1b”. In: *Berkeley Benchmarking and Optimization (BeBOP) Group* 15 (2006).

[67] Richard Vuduc, James W Demmel, and Katherine A Yelick. “OSKI: A library of automatically tuned sparse matrix kernels”. In: *Journal of Physics: Conference Series*. Vol. 16. 1. IOP Publishing. 2005, p. 521.

[68] R Clinton Whaley and Jack J Dongarra. “Automatically tuned linear algebra software”. In: *SC’98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE. 1998, pp. 38–38.

[69] N-E Wiberg and Xiang Dong Li. “Superconvergent patch recovery of finite-element solution and a posteriori L2 norm error estimate”. In: *Communications in Numerical Methods in Engineering* 10.4 (1994), pp. 313–320.

[70] Giuseppe Paolo Ernesto Toffanin Zingales. “HalideTuner: generating and tuning halide schedules with Opentuner”. PhD thesis. Massachusetts Institute of Technology, 2015.

# B. Appendix

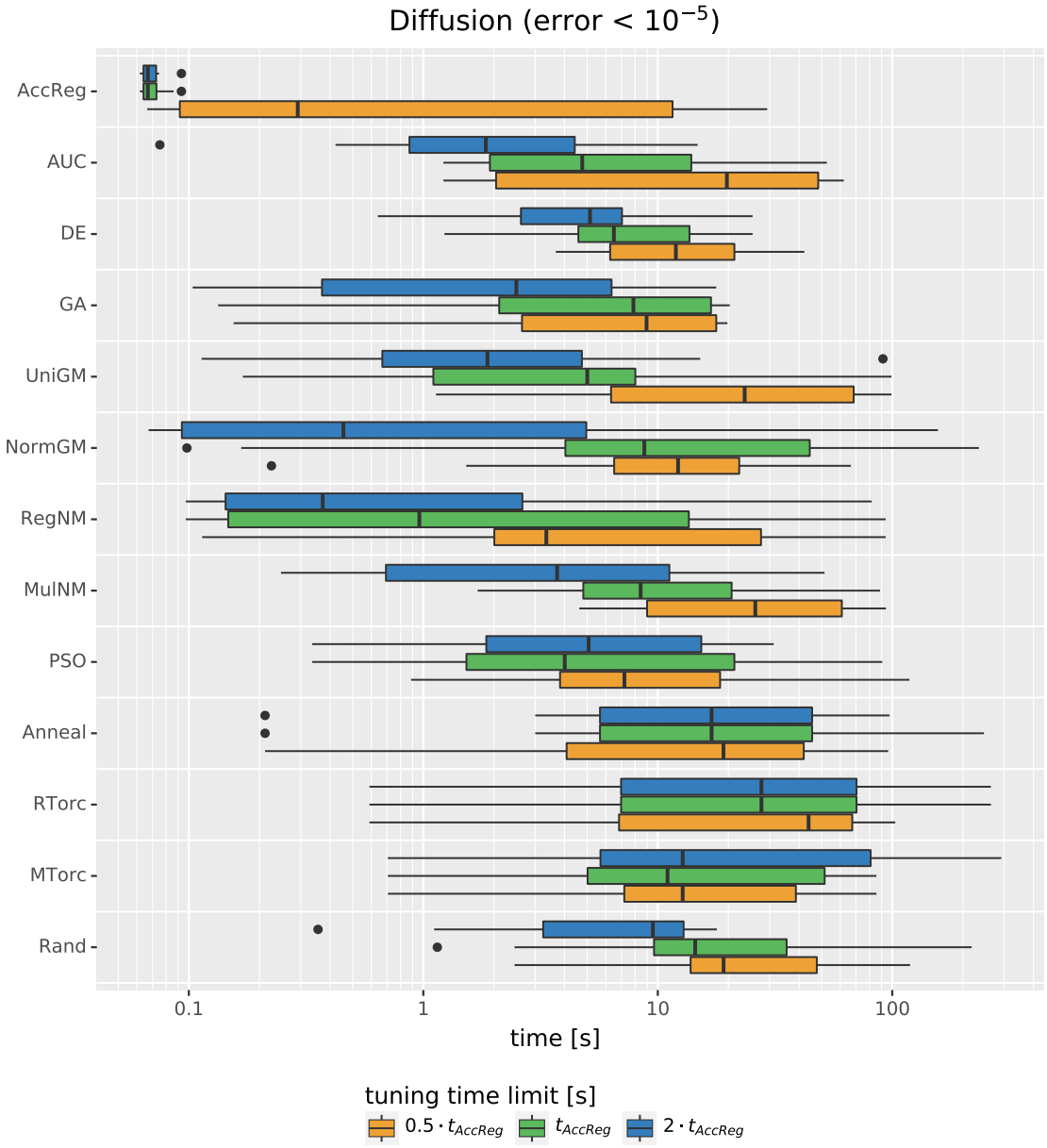



Figure B.1.: Achieved simulation runtimes by different optimization algorithms for the diffusion simulation with an error threshold of $10^{-5}$. The runtime (x-axis) refers to the measured time over the tuning period ($t = 0$ to $t = 2$). Each algorithm is evaluated with an upper tuning time limit of 0.5, 1 and 2 times the tuning time of the model-based search.

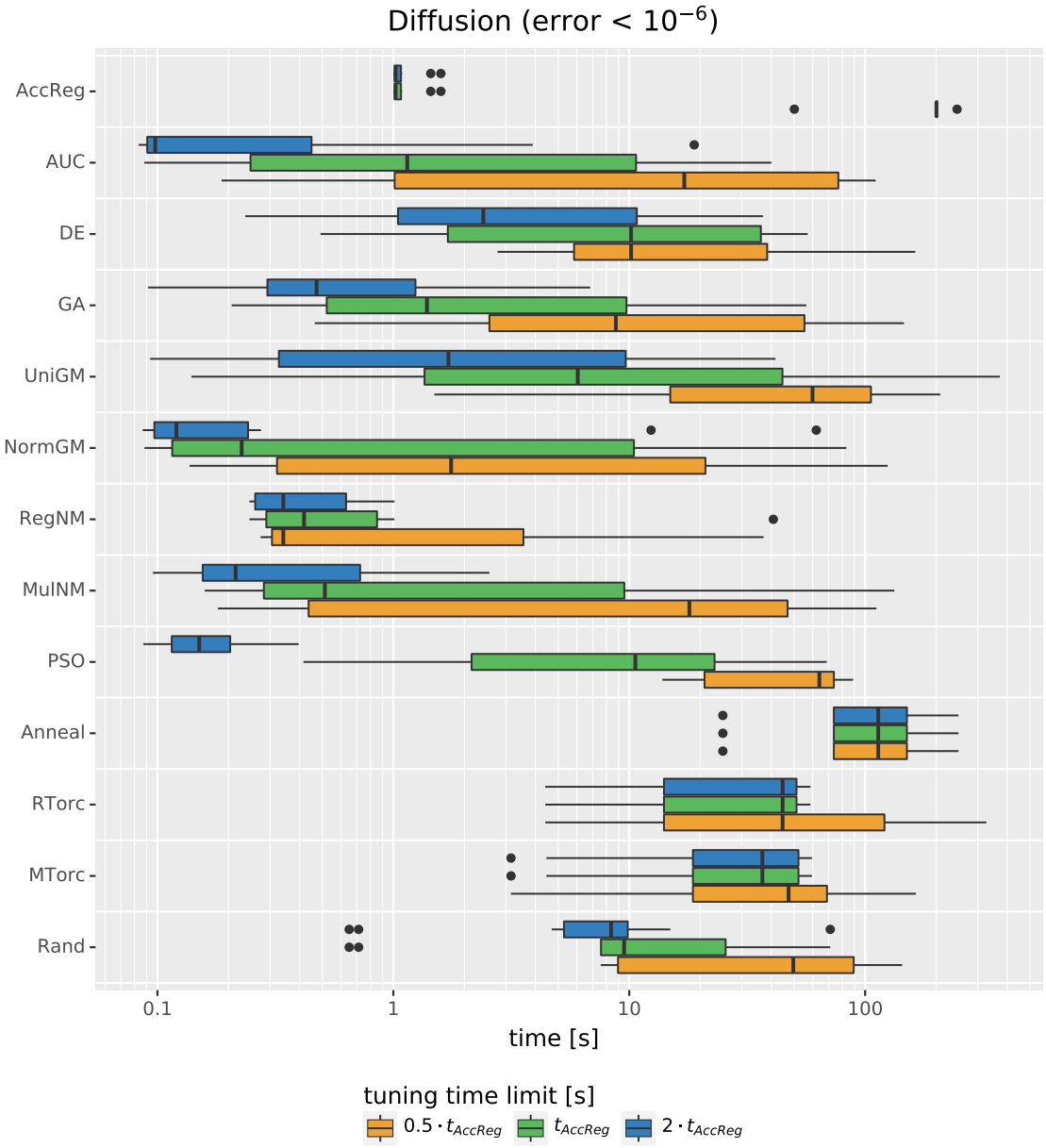



Figure B.2.: Achieved simulation runtimes by different optimization algorithms for the diffusion simulation with an error threshold of $10^{-6}$. The runtime (x-axis) refers to the measured time over the tuning period ($t = 0$ to $t = 2$). Each algorithm is evaluated with an upper tuning time limit of 0.5, 1 and 2 times the tuning time of the model-based search.

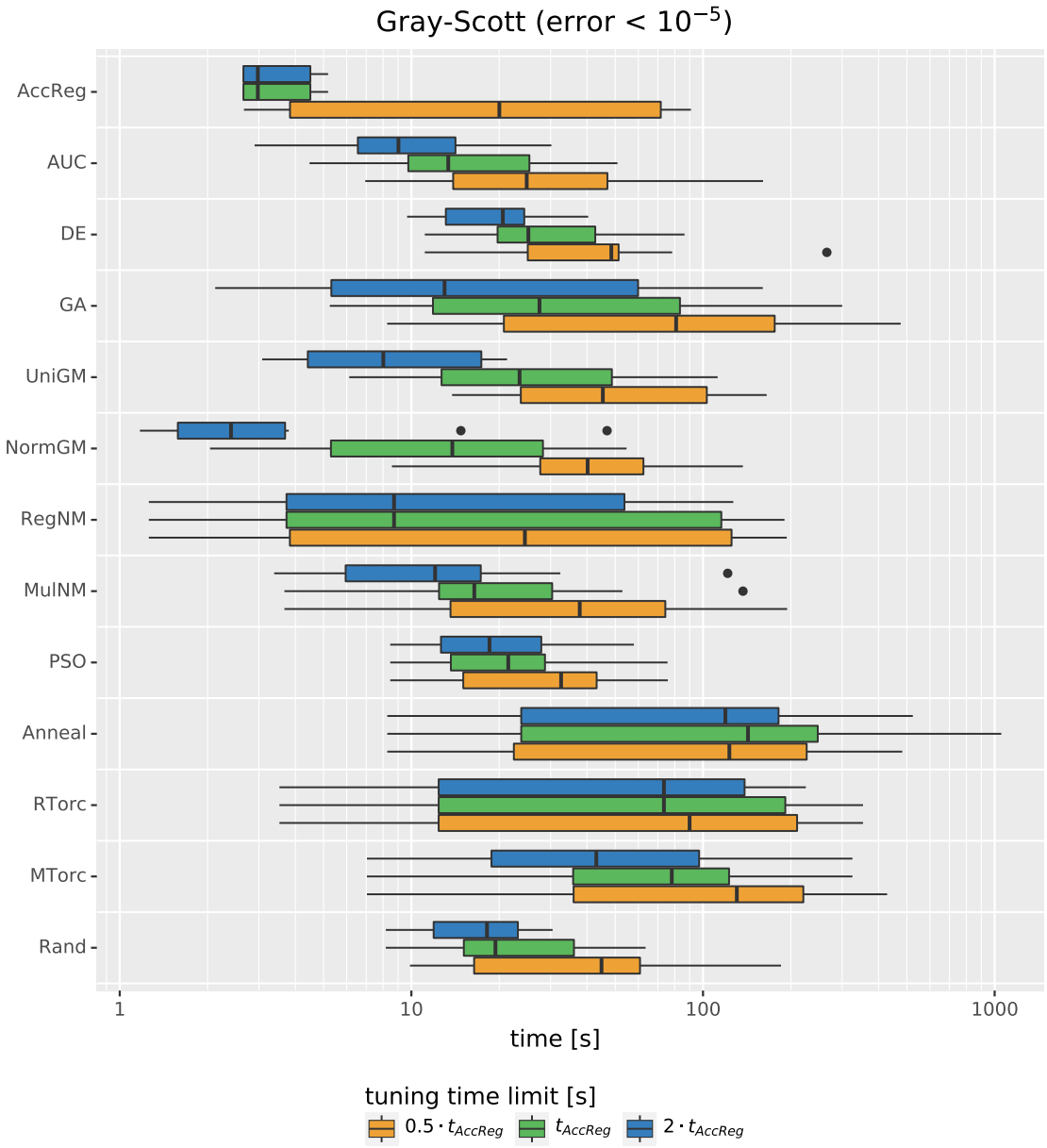



Figure B.3.: Achieved simulation runtimes by different optimization algorithms for the Gray-Scott simulation with an error threshold of $10^{-5}$. The runtime (x-axis) refers to the measured time over the tuning period ($t = 0$ to $t = 2$). Each algorithm is evaluated with an upper tuning time limit of 0.5, 1 and 2 times the tuning time of the model-based search.

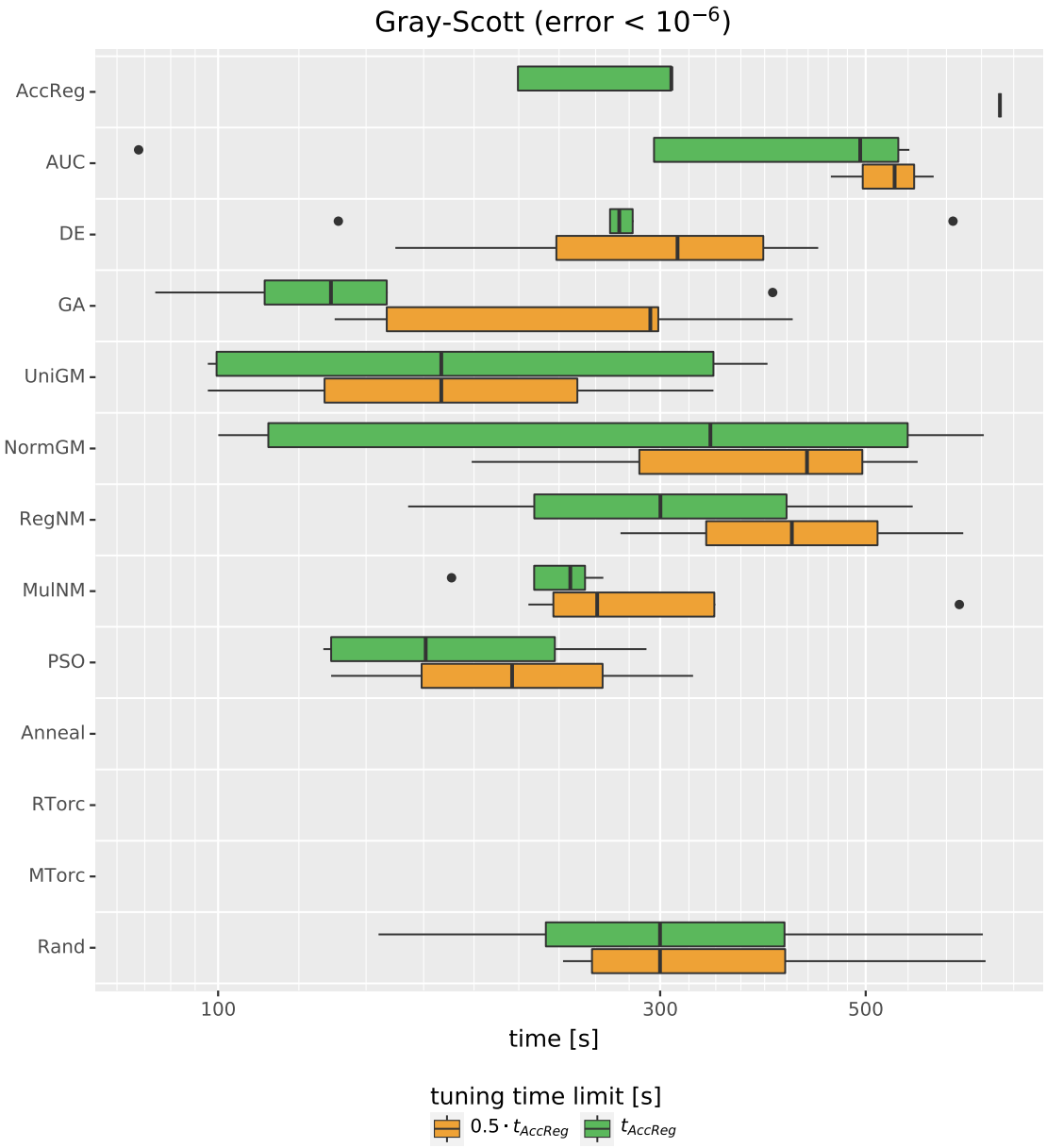



Figure B.4.: Achieved simulation runtimes by different optimization algorithms for the Gray-Scott simulation with an error threshold of $10^{-6}$. The runtime (x-axis) refers to the measured time over the tuning period ($t = 0$ to $t = 2$). Each algorithm is evaluated with an upper tuning time limit of 0.5 and 1 times the tuning time of the model-based search.