Abstract. Particle accelerators are invaluable tools for research in the basic and applied sciences, in fields such as materials science, chemistry, the biosciences, particle physics, nuclear physics and medicine. The design, commissioning, and operation of accelerator facilities is a non-trivial task, due to the large number of control parameters and the complex interplay of several conflicting design goals.

We propose to tackle this problem by means of multi-objective optimization algorithms which also facilitate a parallel deployment. In order to compute solutions in a meaningful time frame we require a fast and scalable software framework.

In this paper, we present the implementation of such a general-purpose framework for simulation-based multi-objective optimization methods that allows the automatic investigation of optimal sets of machine parameters. The implementation is based on a master/slave paradigm, employing several masters that govern a set of slaves executing simulations and performing optimization tasks.

Using evolutionary algorithms and OPAL simulations as optimizer and forward solver in our framework, we present validation experiments and first results of multi-objective optimization problems in the domain of beam dynamics.

Key words. multi-objective optimization, beam dynamics, high performance computing

1. INTRODUCTION. Particle accelerators play a significant role in many aspects of science and technology. Fields, such as material science, chemistry, the biosciences, particle physics, nuclear physics and medicine depend on reliable and effective particle accelerators, both as research but as well as practical tools. Achieving the required performance in the design, commissioning, and operation of accelerator facilities is a complex and versatile problem. Today, tuning machine parameters, e.g., bunch charge, emission time and various parameters of beamline elements, is most commonly done manually by running simulation codes to scan the parameter space. This approach is tedious, time consuming and can be error prone. In order to be able to reliably identify optimal configurations of accelerators we propose to solve large multi-objective design optimization problems to automate the investigation for an optimal set of tuning parameters. Observe that multiple and conflicting optimality criteria call for a multi-objective approach.

We developed a modular multi-objective software framework (see Fig. 1.1) where the core functionality is decoupled from the “forward solver” and the optimizer. To that end, we use a master/slave mechanism where a master process governs a set of slave processes given some computational tasks to complete. This separation allows to easily interchange optimizer algorithms, forward solvers and optimization problems. A “pilot” coordinates all efforts between the optimization algorithm and the forward solver. This forms a robust and general framework for massively parallel multi-objective optimization. Currently the framework offers one concrete optimization algorithm, an evolutionary algorithm employing a NSGAII selector [1].
Normally, simulation based approaches are plagued by the trade-off between level of detail and time to solution. We address this problem by using forward solvers with different time and resolution complexity.

The framework discussed here, incorporates the following three contributions:

1. Implementation of a scalable optimization algorithm capable of approximating Pareto fronts in high dimensional spaces,
2. design and implementation of a modular framework that is simple to use and deploy on large scale computation resources, and
3. demonstrating the usefulness of the proposed framework on a real world application in the domain of particle accelerators.

Here, we will focus on a real world application arising from PSI’s SwissFEL activities, which have an immediate and a long-range impact, by optimizing important parameters of PSI’s 250 MeV injector test facility (and later for the full PSI SwissFEL).

The next section introduces the notation of multi-objective optimization theory and describes the first implemented optimizer. In Section 3 we discuss the implementation of the framework. We introduce the employed forward-solver in Section 4. A validation and a proof of concept application of a beam dynamics problem is discussed in Section 5.

2. MULTI-OBJECTIVE OPTIMIZATION. Optimization problems deal with finding one or more feasible solutions corresponding to extreme values of objectives. If more than one objective is present in the optimization problem we call this a multi-objective optimization problem (MOOP). A MOOP is defined as

$$\begin{align*}
\min_{x} & \quad f_m(x), \quad m = 1, \ldots, M \\
\text{s.t.} & \quad g_j(x) \geq 0, \quad j = 0, \ldots, J \\
& \quad x_i^L \leq x_i \leq x_i^U, \quad i = 0, \ldots, n,
\end{align*}$$

where $f$ denotes the objectives (2.1), $g$ the constraints (2.2) and $x$ the design variables (2.3).
Often, we encounter conflicting objectives complicating the concept of optimality. To illustrate this, let us consider the problem of buying a car. Naturally, we try to bargain the lowest price for the best performance, e.g. maximal horsepower or minimal fuel consumption. This can be formulated as MOOP (2.4):

\[
\begin{align*}
\text{min} \quad & [\text{price}, -\text{performance}]^T \\
\text{s.t.} \quad & \text{low}_{\text{pr}} \leq \text{price} \leq \text{high}_{\text{pr}} \\
& \text{low}_{\text{pc}} \leq \text{performance} \leq \text{high}_{\text{pc}}
\end{align*}
\]

In general, it is not possible to get the maximal performance for the lowest price and a trade-off decision between performance and price has to be reached (see Figure 2.1). Since not every choice is equally profitable for the buyer (for example, car \( x_4 \) costs as much as \( x_2^* \) but offers less performance), we pick trade-offs (red points) that are essentially “equally optimal” in both conflicting objectives, meaning, we cannot improve one point without hurting at least one other solution. This is known as Pareto optimality. The set of Pareto optimal points (blue curve) forms the Pareto front or surface. All points on this surface are considered to be Pareto optimal.

Once the shape of the Pareto front has been determined the buyer can specify preference, balancing the features by observing the effect on the optimality criteria, converging to the preferred solution. This is called a posteriori preference specification since we select a solution after all possible trade-offs have been presented to us. An alternative is to specify preference a priori, e.g., by weighting (specifying preference before solving the problem) and combining all objectives into one and applying a single-objective method to solve the problem (yielding only one solution). In many situations preference is not known a priori and an a posteriori preference specification helps conveying a deeper understanding of the solution space. The Pareto front can be explored and the impact of a trade-off decision then becomes visible.

Sampling Pareto fronts is far from trivial. A number of approaches have been proposed, e.g. evolutionary algorithms [10], simulated annealing [21], swarm methods [20], and many more [9][12][19][26]. In the next section we briefly introduce the
theory of evolutionary algorithms used in the present work.

2.1. Evolutionary Algorithms. Evolutionary algorithms are loosely based on nature’s evolutionary principles to guide a population of individuals towards an improved solution by honoring the “survival of the fittest” practice. This “simulated” evolutionary process preserves entropy (or diversity in biological terms) by applying genetic operators, such as mutation and crossover, to remix the fittest individuals in a population. Maintaining diversity is a crucial feature for the success of all evolutionary algorithms.

In general, a generic evolutionary algorithm consists of the following components:

- **Genes**: traits defining an individual,
- **Fitness**: a mapping from genes to a fitness value for each individual,
- **Selector**: selecting the \( k \) fittest individuals of a population based on some sort of ordering,
- **Variator**: recombination (mutations and crossover) operators for offspring generation.

Applied to multi-objective optimization problems, genes correspond to design variables. The fitness of an individual is loosely related to the value of the objective functions for the corresponding genes. Figure 2.2 schematically depicts the connection of the components introduced above. The process starts with an initially random population of individuals, each individual with a unique set of genes and corresponding fitness, representing one location in the search space. In a next step the population is processed by the selector determining the \( k \) fittest individuals according to their fitness values. While the \( k \) fittest individuals are passed to the variator, the remaining \( n - k \) individuals are eliminated from the population. The Variator mates and recombines the \( k \) fittest individuals to generate new offspring. After evaluating the fitness of all the freshly born individuals a generation cycle has completed and the process can start anew.

Since there already exist plenty of implementations of evolutionary algorithms, we decided to incorporate the PISA library \[\text{1}\] into our framework. One of the advantages of PISA is that it separates variator from selector, rendering the library expandable.
and configurable. Implementing a variator was enough to use PISA in our framework and retain access to all available PISA selectors. As shown in Figure 2.2, the selector is in charge of ordering a set of \( d \)-dimensional vectors and selecting the \( k \) fittest individuals currently in the population. The performance of a selector depends on the number of objectives and the surface of the search space. So far, we used an NSGA-II selector \[11\] exhibiting satisfactory convergence performance.

The task of the variator is to generate offspring and ensure diversity in the population. The variator can start generating offspring once the fitness of every individual of the population has been evaluated. This explicit synchronization point defines an obvious bottleneck for parallel implementations of evolutionary algorithms. In the worst case some MPI threads are taking a long time to compute the fitness of the last individual in the pool of individuals to evaluate. During this time all other resources are idle and wait for the result of this one individual in order to continue to generate and evaluate offspring. To counteract this effect we call the selector already when two individuals have finished evaluating their fitness, lifting the boundaries between generations and evaluating the performance of individuals. New offspring will be generated and MPI threads can immediately go back to work on the next fitness evaluation. By calling the selector more frequently (already after two offspring individuals have been evaluated) results in better populations since bad solutions are rejected faster. On the other hand, calling the selector more often is computationally more expensive.

Our variator implementation uses the master/slave architecture, presented in the next section, to run as many function evaluations as possible in parallel. Additionally, various crossover and mutation policies are available for tuning the algorithm to the optimization problem.

3. THE FRAMEWORK. Simulation based multi-objective optimization problems are omnipresent in research and the industry. The simulation and optimization problems arising in such problems are in general very big and computationally demanding. This motivated us to design a massively parallel general purpose framework. The key traits of such a design can be summarized as:

- support any multi-objective optimization method,
- support any function evaluator: simulation code or measurements,
- offer a general description/specification of objectives, constraints and design variables,
- run efficiently in parallel on current large-scale high-end clusters and supercomputers.

3.1. Related Work. Several similar frameworks, e.g. \[13, 16, 22, 23\], have been proposed. Commonly these frameworks are tightly coupled to an optimization algorithm, e.g. only providing evolutionary algorithms as optimizers. Users can merely specify optimization problems, but cannot change the optimization algorithm. Our framework follows a more general approach, providing a user-friendly way to introduce new or choose from existing built-in multi-objective optimization algorithms. Tailoring the optimization algorithm to the optimization problem at hand is an important feature due to the many different characteristics of optimization problems that should be handled by such a general framework. As an example, we show how PISA \[1\], an existing evolutionary algorithm library, was integrated with ease. Similarly, other multi-objective algorithms could be incorporated and used to solve optimization problems.

The framework presented in \[22\] resembles our implementation the most, aside from their tight coupling with an evolutionary algorithm optimization strategy. The
authors propose a plug-in based framework employing an island parallelization model, where multiple populations are evaluated concurrently and independently up to a point where some number of individuals of the population are exchanged. This is especially useful to prevent the search algorithm to get stuck in a local minimum. A set of default plug-ins for genetic operators, selectors and other components of the algorithms are provided by their framework. User-based plug-ins can be incorporated into the framework by implementing a simple set of functions.

Additionally, as with simulation based multi-objective optimization, we can exploit the fact that both the optimizer and simulation part of the process use a certain amount of resources. The ratio of work between optimizer and simulation costs can be reflected in the ratio of number of processors assigned to each task. This not only provides users with great flexibility in using any simulation or optimizer, but renders influencing the role assignment easy as well.

3.2. Components. The basic assumption in simulation-based optimization is that we need to call an expensive simulation software component present in the constraints or objectives. We divide the framework in three exchangeable components, as shown in Figure 3.1, to encapsulate the major behavioral patterns of the framework.

The Pilot component acts as a bridge between the optimizer and forward solvers, providing the necessary functionality to handle passing requests and results between the Optimizer and the Simulation modules.

We implemented the framework in C++, utilizing features like template parameters to specify the composition of the framework (shown in Listing 1).

```
typedef OpalInputFileParser Input_t;
typedef PisaVariator Opt_t;
typedef OpalSimulation Sim_t;
typedef Pilot< Input_t, Opt_t, Sim_t > Pilot_t;
```

The framework provides “default” implementations that can be controlled via command line options. Due to its modular design, all components can be completely customized.

Every available MPI thread will take up one of the three available roles (see Figure 1.1): one thread acts as Pilot, the remaining threads are divided amongst
Worker and Optimizer roles. Both, the Worker and the Optimizer can consist of multiple MPI threads to exploit parallelism. As shown in Figure 3.1, the Pilot is used to coordinate all “information requests” between the Optimizer and the Worker. An information request is a job that consists of a set of design variables (e.g., the genes of an individual) and a type of information it requests (e.g., function evaluation or derivative). The Pilot keeps checking for idle Worker and assigns jobs in the queue to any free Worker. Once the Worker has computed and evaluated the request its results are returned to the Optimizer that originally requested the information.

After a thread gets appointed a role it starts a polling loop asynchronously checking for appropriate incoming requests. To that end a Poller interface helper class has been introduced. The Poller interface consists of an infinite loop that checks periodically for new MPI messages. Upon reception a new message is immediately forwarded to the appropriate handler, the onMessage() method. The method is called with the MPI_Status of the received message and a size_t value specifying different values depending on the value of the MPI_Tag. The Poller interface allows the implementation of special methods (denoted hooks) determining the behavior of the polling process, e.g., for actions that need to be taken after a message has been handled. Every Poller terminates the loop upon receiving a special MPI tag.

3.3. Implementing an Optimizer. All Optimizer implementations have to respect the API shown in Listing 2.

Code Listing 2: Optimizer API

```cpp
virtual void initialize() = 0;
// Poller hooks
virtual void setupPoll() = 0;
virtual void prePoll() = 0;
virtual void postPoll() = 0;
virtual bool onStop() = 0;
virtual bool onMessage(MPI_Status status,
size_t length) = 0;
```

All processors running an Optimizer component call the initialize entry point after role assignment in the Pilot. The implementation of initialize must set up and start the poller and the optimization code. Since an optimizer derives from the Poller interface, predefined hooks can be used to determine the polling procedure. Hooks can be implemented as empty methods, but the onMessage implementation should reflect the optimization part of the protocol for handling events from the Pilot. A special set of communicator groups serves as communication channels to the Pilot and its job queue and if existing to threads supporting the Optimizer component.

3.4. Implementing a Forward Solver. In most cases Simulation implementations are simple wrappers to run an existing “external” simulation code using a set of design variables as input. As for the Optimizer component there exists a base class, labeled Simulation as common basis for all Simulation implementations. In addition, this component also inherits from the Worker class, already implementing the polling protocol for default worker types. As shown in the API in Listing 3 the Worker class expects an implementation to provide implementations for those three methods.

Code Listing 3: Simulation API

```cpp
```
virtual void run() = 0;
virtual void collectResults() = 0;
virtual reqVarContainer_t getResults() = 0;

First, upon receiving a new job, the Worker will call the run method on the Simulation implementation. This expects the Simulation implementation to run the simulation in a blocking fashion, meaning the method call blocks and does not return until the simulation has terminated. Subsequently, the Worker calls collectResults, where the Simulation prepares the result data, e.g., parsing output files, and stores the requested information in a reqVarContainer_t data structure. Finally, the results obtained with getResults are sent to the Pilot. As before, the serialized data is exchanged using MPI point-to-point communication using a specific set of communicators.

3.5. Specifying the Optimization Problem. We aimed at an easy and expressive way for users to specify multi-objective optimization problems. Following the principle of keeping metadata (optimization and simulation input data) together, we decided to embed the optimization problem specification in the simulation input file by prefixing it with special characters, e.g., as annotations prefixed with a special character. In some cases it might not be possible to annotate the simulation input file. By providing an extra input file parser, optimization problems can be read from stand-alone files.

To allow arbitrary constraints and objective expressions, such as

\[
\text{name: OBJECTIVE, EXPR=5 \cdot \text{average(42.0, "measurement.dat") + ENERGY}}
\]

we implemented an expression parser using Boost Spirit\(^1\). In addition to the parser, we need an evaluator able to evaluate an expression, given a parse tree and variable assignments to an actual value. Expressions arising in multi-objective optimization problems usually evaluate to booleans or floating point values. The parse tree, also denoted abstract syntax tree (AST), is constructed recursively while an expression is parsed. Upon evaluation, all unknown variables are replaced with values, either obtained from simulation results or provided by other subtrees in the AST. In this stage, the AST can be evaluated bottom-up and the desired result is returned after processing the root of the tree.

To improve the expressive power of objectives and constraints we introduced a simple mechanism to define and call custom functions in expressions. Using simple functors, e.g., as the one shown in Listing 4 to compute an average over a set of data points, enriches expressions with custom functions. Custom function implementations overload the () parenthesis operator. The function arguments specified in the corresponding expression are stored in a std::vector of Boost variants\(^2\) that can be booleans, strings or floating point values.

**Code Listing 4: Simple Average Functor**

```cpp
struct avg {

    double operator()(client::function::arguments_t args) const {
        double limit = boost::get<double>(args[0]);
        std::string filename =
```
All custom functions are registered with expression objects. This is necessary to ensure that expressions know how they can resolve function calls in their AST. As shown in Listing 5 this is done by creating a collection of Boost functions\(^3\) corresponding to the available custom functions in expressions and passing this to the PILOT.

**Code Listing 5: Creating function pointer for registering functor**

```cpp
functionDictionary_t funcs;
client::function::type ff;
ff = average();
funcs.insert(std::pair<std::string, client::function::type>(
    "my_average_name", ff));
```

A set of default operators, corresponding to a mapping to C math functions, is included in the dictionary by default. This enables an out of source description of optimization problem containing only simple math primitives.

**3.6. Parallelization.** The parallelization is defined by a mapping of the roles introduced above to available cores. Command-line options allow the user to steer the number of processors used in worker and optimizer groups. Here, we mainly use the command-line options to steer the number of processors running a forward solver.

The parallelization will be explained and benchmarked in detail in a forthcoming publication. One major disadvantage of the master/slave implementation model is the fast saturation of the network links surrounding the master node. In\(^5\) authors observe an exponential increase in hot-spot latency with increasing number of workers that are attached to one master process. Clearly, the limiting factor is the number of outgoing links of a node in the network topology and already for a few workers the links surrounding a master process are subject to congestions. This effect is amplified further by large message sizes.

To that end we implemented a solution propagation based on rumor networks (see e.g.\(^4\)) using only one-sided communication. This limits the number of messages sent over the already heavily used links surrounding the master node and at the same time helps to prevent the use of global communication. Using information about the interconnection network topology and the application communication graph the task of assigning roles helps to further improve the parallel performance.

**4. FORWARD SOLVER.** The framework contains a wrapper implementing the API mentioned in Listing 3 for using OPAL\(^2\) as the forward solver. OPAL provides different trackers for cyclotrons and linear accelerators with satisfactory parallel performance\(^3\). Recently we introduced a reduced envelope model\(^18\) into OPAL reducing time to solution by several orders of magnitude.

With access to the OPAL forward solver the framework is able to tackle a multitude of optimization problems arising in the domain of particle accelerators. By

\(^3\)www.boost.org/libs/function/
Fig. 5.1. The hypervolume for a two-objective optimization problem corresponds to the accumulated area of all dashed rectangles spanned by all points in the Pareto set and arbitrary origin $p_0$.

abiding to the API from Listing 3 it becomes trivial to add new forward solvers to the framework (see Appendix A). If the objectives and constraints are simple arithmetical expressions, the FunctionEvaluator simulator can be used. Using functors and the default expression primitives already powerful multi-objective optimization problems can be specified, i.e. the benchmark problem presented in [17]:

$$
\begin{align*}
\min & \quad \left[ 1 - \exp\left( -1 \left( \left( x_1 - \frac{1}{\sqrt{3}} \right)^2 + \left( x_2 - \frac{1}{\sqrt{3}} \right)^2 + \left( x_3 - \frac{1}{\sqrt{3}} \right)^2 \right) \right) \right], \\
& \quad \left[ 1 - \exp\left( -1 \left( \left( x_1 + \frac{1}{\sqrt{3}} \right)^2 + \left( x_2 + \frac{1}{\sqrt{3}} \right)^2 + \left( x_3 + \frac{1}{\sqrt{3}} \right)^2 \right) \right) \right]^T \\
\text{s.t.} & \quad -1 \leq x_i \leq 1, \quad i = 1, 2, 3.
\end{align*}
$$

Using the default expression primitives this can be stated in an input file as:

```plaintext
d1: DVAR, VARIABLE="x1", LOWERBOUND="-1.0", UPPERBOUND="1.0";
d2: DVAR, VARIABLE="x2", LOWERBOUND="-1.0", UPPERBOUND="1.0";
d3: DVAR, VARIABLE="x3", LOWERBOUND="-1.0", UPPERBOUND="1.0";
obj1: OBJECTIVE,
  EXPR="1- exp(-1 * (sq(x1 - 1/sqrt(3)) + sq(x2 - 1/sqrt(3)) + sq(x3 - 1/sqrt(3))))";
obj2: OBJECTIVE,
  EXPR="1- exp(-1 * (sq(x1 + 1/sqrt(3)) + sq(x2 + 1/sqrt(3)) + sq(x3 + 1/sqrt(3))))";
objs: OBJECTIVES = (obj1, obj2);
dvars: DVARS = (d1, d2, d3);
constrs: CONSTRAINTS = ();
opt: OPTIMIZE, OBJECTIVES=objs, DVARS=dvars, CONSTRAINTS=constrs;
```

5. EXPERIMENTS. In this section we present numerical results of the validation benchmark and discuss a proof of concept application in the domain of particle accelerators.

5.1. Optimizer Validation. To ensure that the optimizer works correctly we solved the benchmark problem [4.1]. To that end, we use a metric for comparing the quality of a Pareto front. Given a point in the Pareto set, we compute the $m$ dimensional volume (for $m$ objectives) of the dominated space, relative a chosen origin.
We visualize this for 2 objectives in Figure 5.1. For further information and details of the implementation see [27], Figure 5.2 and the corresponding hypervolume values in Table 5.1. The reference Pareto front is clearly very well approximated. It took a total of 1100 function evaluations to perform this computation. The hypervolume of the reference solution (0.6575) for our benchmark was computed by sampling the solution provided in [17].

Table 5.1

| tot. function evaluations | hyper volume | relative error |
|---------------------------|--------------|----------------|
| 100                       | 0.859753     | 3.076 × 10⁻¹   |
| 200                       | 0.784943     | 1.938 × 10⁻¹   |
| 500                       | 0.685183     | 4.210 × 10⁻²   |
| 900                       | 0.661898     | 6.689 × 10⁻³   |
| 1100                      | 0.657615     | 1.749 × 10⁻⁴   |

From Table 5.1 we deduce that we achieved satisfactory convergence to the sampled reference Pareto front after 1000 (plus the additional 100 evaluations for the initial population) function evaluations.

5.2. Ferrario Matching Point. As a verification and proof of concept we reproduce the Ferrario matching point discovered by Ferrario et al. [15], by formulating the problem as a multi-objective optimization problem. Using the low-dimensional and fast nature of their new simulation code Homdyn [14], an extensive beam dynam-
ics study was conducted.

One of the results of the study presented in [15] was the discovery of a novel working point. The authors noticed that the second emittance minimum can profit from the additional emittance compensation in the accelerating traveling wave structure ensuring that the second emittance minimum occurs at a higher energy. This property is attained if the beam emittance has a maximum and the root mean square (rms) beam size has a minimum at the entrance of the first accelerating traveling wave structure. This behavior is illustrated in Figure 5.3.

By artificially reproducing this working point as the solution of a multi-objective optimization problem given in equations (5.1) to (5.9), we demonstrate the automation of discovering optimal beam dynamics behaviors given a set of desired objectives.

\[
\begin{align*}
\text{min} & \quad |\Delta \text{rms}_{x, \text{peak}}| = |3.025 - \text{rms}_{x, \text{peak}}|, \\
\Delta \varepsilon_{x, \text{peak}} & = |3.025 - \varepsilon_{x, \text{peak}}|, \\
|\text{rms}_{x, \text{peak}, \text{pos}} - \varepsilon_{x, \text{peak}, \text{pos}}| & = q = 200 \text{[pC]} \\
\text{s.t.} & \quad q = 200 \text{[pC]} \\
& \quad \text{Volt}_{\text{RF}} = 100 \text{[MV/m]} \\
& \quad \sigma_L \leq \sigma_x = \sigma_y \leq \sigma_U \\
& \quad \text{KS}_L \leq \text{KS}_{\text{RF}} \leq \text{KS}_U \\
& \quad \text{LAG}_L \leq \text{LAG}_{\text{RF}} \leq \text{LAG}_U \\
& \quad \Delta z_{\text{KS}} \leq \Delta z_{\text{RF}} \leq \Delta z_{\text{UKS}}
\end{align*}
\]

The first two objectives minimize the distance from the position of the current minimum peak to the expected peak location at 3.025 m for transverse bunch size (beam waist) and emittance (see Figure 5.3). The third objective (5.3) adds a condition preferring solutions that have their emittance and rms peak locations at the same z-coordinate. Equations (5.4) and (5.5) define constraints for initial conditions for the simulation: charge, gun voltage and laser spot size. Design variables given in (5.6) to (5.9) correspond to field strengths of the first focusing magnet, its displacement, and the phase of the gun.

In order to compute the peaks, we employed an additional Python script. This script was called in the OPAL input file, after the simulation finished using the SYSTEM functionality. Once the peaks (in a given range) were located, the two objectives (5.1)
Table 5.2

| Initial conditions for the envelope tracker. |
|---------------------------------------------|
| name | initial value |
| Gun voltage | 100 MV |
| Bunch charge | 200 pC |
| DT Beamline | 1.5 ps |
| Number of slices | 400 |

and (5.2) were computed and their values written into corresponding files. The custom `fromFile` functor allows us to access the values stored in the peak finder Python script result files:

```python
rmsx: OBJECTIVE, EXPR="fromFile("rms_x-err.dat", "var")";
emitx: OBJECTIVE, EXPR="fromFile("emit_x-err.dat", "var")";
match: OBJECTIVE, EXPR="fabs(fromFile("emit_x-peak.dat", "var") -
 fromFile("rms_x-peak.dat", "var"));"
```

The design variables and the assembly of the multi-objective optimization problem can be included in the OPAL input file as shown below:

```python
d1: DVAR, VARIABLE="SIGX", LOWERBOUND="0.00025", UPPERBOUND="0.00029";
d2: DVAR, VARIABLE="FIND1_MSOL10_i", LOWERBOUND="110", UPPERBOUND="120";
d3: DVAR, VARIABLE="D_LAG_RGUN", LOWERBOUND="-0.1", UPPERBOUND="0.1";
d4: DVAR, VARIABLE="D_SOLPOS", LOWERBOUND="-0.05", UPPERBOUND="0.05";
objs: OBJECTIVES = (rmsx, emitx);
dvars: DVARS = (d1, d2, d3, d4);
constrs: CONSTRAINTS = ()
opt: OPTIMIZE, OBJECTIVES=objs, DVARS=dvars, CONSTRAINTS=constrs;
```

All numerical experiments in this sections were executed on the FELSIM cluster at PSI. The FELSIM cluster consists of 8 dual quad-core Intel Xeon processors at 3.0 GHz and has 2 GB memory per core with a total of 128 cores. The nodes are connected via Infiniband network with a total bandwidth of 16 GB/s.

5.3. Convergence Study. The envelope tracker, mentioned in the previous chapter, was chosen as the forward solver. We performed a beam convergence study in order to tune the simulation input parameters to achieve the best trade-off between simulation accuracy and time to solution. These parameters include the number of slices (NSLICE) used for the envelope-tracker simulations, simulation timestep (DT) and gun timestep (DTGUN).

Before the simulation can be executed a number of initial beam optics parameters have to be defined in an input file. Table 5.2 shows the values of these parameters for the envelope-tracker. All simulations were performed up to 12.5 m of the SwissFEL 250 MeV injector [24] beam line, with energies reaching up to 120 MeV.

The parameter that affects the performance most is the number of slices. We scanned the range from 100 to 1000 slices to determine the minimal number of slices required for stable results using various timesteps. The results (for 100, 400 and 800 slices) of this scan are shown in Figure 5.4. Using this data we settled for 400 slices – increasing the slice number only minimally improves convergence of the results, therefore using more slices is inefficient.

In a next step the influence of different time steps was examined. To that end
5.4. Optimization Results. Each of the 40 points on the Pareto front, shown in Figure 5.5, represents an optimal solution, where emittance and beamsize values are compromised to achieve the best agreement with the Ferrario matching point. We selected individual 3 based on a comparison of the emittance and beamsize characteristics of all solutions and by retaining the feasibility of the beam line optics parameters. The design variables, emittance and beamsize of the selected solution are shown in Table 5.3 and Figure 5.6. With the multi-objective optimization framework we attain the same working point as reported in [24].

Using the input parameters of the selected solution, we performed a stability analysis by varying the slice number and the time step for both the gun and the beam line. Figure 5.4 shows that the exit emittance stabilizes for 400 slices and various time steps.

Table 5.3

| name                  | value          |
|-----------------------|----------------|
| $\sigma_x$            | 0.262 mm       |
| Solenoid displacement | 28.8467 mm     |
| Gun voltage lag       | 0.0159067 MV   |
| Solenoid current      | 111.426 A      |

a series of optimization runs with 100, 400 and 800 slices and varying timestep was performed. Figure 5.5 shows the Pareto fronts for 400 slices respectively using different timesteps. As expected, increasing the number of slices while lowering the timestep produces more detailed results.
steps. No difference between 800 and 400 slices is visible as their minimum maximum extension seems to be in the same range of 0.024 mm mrad.

For validation purposes we compared the results of the envelope-tracker using
the analytical space charge model with the OPAL 3D macro particle tracker. The benchmark was run on the first 12.5 meters of the SwissFEL 250 MeV injector. The results for both rms beamsize and emittance are shown in Figure 5.7. A good agreement between the two codes can be observed. The difference of the larger emittance along the solenoids in case of 3D tracker that is not seen by the envelope-tracker is due to the different definition of the particle momenta (canonical vs. mechanical). Both trackers agree within acceptable limits [8].

6. CONCLUSIONS. We presented a general-purpose framework for solving multi-objective optimization problems. Its modular design simplifies the application to simulation-based optimization problems for a wide range of problems and allows to exchange the optimization algorithm. In a recent work, we successfully implemented and integrated another optimization algorithm, originally presented in [25], into our framework. The flexibility of being able to adapting both ends of the optimization process, the forward solver and the optimization algorithm simultaneously not only leads to broad applicability but it facilitates the ability to tailor the optimization strategy to the optimization problem as well.

In this paper we applied the framework to reproduce the important and well known Ferrario matching condition for the 250 MeV injector of PSI’s SwissFEL. The beam size and emittance optimization was successful and emittance damping is observed in the accelerating cavity. Also, the influence of the number of slices employed by the envelope tracker with respect to convergence was investigated. We found that the optimal slice number is around 400 for the problem addressed here, producing the best trade-off between computational cost and accuracy of the solution of the simulation. Similarly, the gun and beam line time step was investigated confirming the convergence of the results. This first study shows that the framework is ready to tackle problems arising in the domain of beam dynamics.

Interestingly, the computation of a good approximation of the Pareto front is
already feasible on a small cluster using only a modest number of MPI threads. For 1000 generations consisting of 2048 feasible and 8354 infeasible function evaluations (simulations) the optimizer needed 845 minutes on the FELSIM cluster on 16 threads. When we double the number of threads to 32 (FELSIM allows a maximum of 64 threads per job) the runtime improves to 302 minutes (2048 accepted and 5426 infeasible individuals). Notice that the runtime can vary due to the random nature of the process. Nevertheless we see a reasonable reduction when using more cores to solve the optimization problem and the parallel performance will be further evaluated in a forthcoming publication.

In contrast to approaches that are tightly coupled to the optimization algorithm, the range of possible applications is much wider. Even in cases where the mathematical model of the forward solver is not known exactly, fixed or on the real time measurements can be used to guide the search for the Pareto optimal solutions. Finally, combining the presented multi-objective optimization framework with a physicist long standing experience in the field provides a solid basis for better understanding and improving the decision making process in the design and operation of particle accelerators.

7. ACKNOWLEDGMENT. The authors thank the SwissFEL team for contributing to the formulation of optimization problems.

Appendix A. Forward Solver Implementation.
A simple implementation, e.g. using a Python forward solver, is given below in Listing 6. We assume the Python script executes a simulation and dumps results in the SDDS file format [6]. Utility classes are used to parse the result data and fill the appropriate structures returned to the optimizer algorithm.

```
#include <map>
#include <string>
#include <vector>
#include "Util/Types.h"
#include "Util/CmdArguments.h"
#include "Simulation/Simulation.h"
#include "boost/smart_ptr.hpp"

class SimpleSimulation : public Simulation {

public:

    SimpleSimulation(Expressions::Named_t objectives,
                     Expressions::Named_t constraints,
                     Param_t params, std::string name, MPI_Comm comm,
                     CmdArguments_t args);

    ~SimpleSimulation();

    /// Calls simulation through Python wrapper and returns when simulation
    /// has either failed or finished.
    void run() {
        // user implements wrappers
        prepare_input_file();
        run_python_simulation();
    }

    /// Parse SDDS stat file and build up requested variable dictionary.
    void collectResults() {
        Expressions::Named_t::iterator it;
        for(it = objectives_.begin(); it != objectives_.end(); it++) {
            
```
Expressions::Expr_t *objective = it->second;

// find out which variables we need in order to evaluate the
// objective
variableDictionary_t variable_dictionary;
std::set<std::string> req_vars = objective->getReqVars();

if(req_vars.size() != 0) {
    boost::scoped_ptr<SDDSReader> sddsr(new SDDSReader(fn));
    try {
        sddsr->parseFile();
    } catch(OptPilotException &e) {
        std::cout << "Exception while parsing SDDS file: "
                   << e.what() << std::endl;
        break;
    }

    // get all the required variable values from the stat file
    foreach(std::string req_var, req_vars) {
        if(variable_dictionary.count(req_var) == 0) {
            try {
                double value = 0.0;
                sddsr->getValue(1 /*atTime*/, req_var, value);
                variable_dictionary.insert(
                    std::pair<std::string, double>(req_var, value));
            } catch(OptPilotException &e) {
                std::cout << "Exception while getting value from SDDS file: "
                           << e.what() << std::endl;
            }
        }
    }

    // and evaluate the expression using the built dictionary of
    // variable values
    Expressions::Result_t result =
        objective->evaluate(variable_dictionary);
    std::vector<double> values;
    values.push_back(boost::get<0>(result));
    bool is_valid = boost::get<1>(result);
    reqVarInfo_t tmps = {EVALUATE, values, is_valid};
    requestedVars_.insert(  
        std::pair<std::string, reqVarInfo_t>(it->first, tmps));
}

// returns container containing all requested variables with results
reqVarContainer_t getResults() { return requestedVars_; }

private:

    // holds solutions returned to the optimizer
    reqVarContainer_t requestedVars_;
[1] PISA — a platform and programming language independent interface for search algorithms. In C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele, editors, Evolutionary Multi-Criterion Optimization (EMO 2003), Lecture Notes in Computer Science, pages 494–508, Berlin, 2003. Springer.

[2] A. Adelmann, C. Kraus, Y. Ineichen, S. Russell, Y. Bi, and J. Yang. The OPAL (Object Oriented Parallel Accelerator Library) Framework. Technical Report PSI-PR-08-02, Paul Scherrer Institut, 2008.

[3] A. Adelmann, C. Kraus, Y. Ineichen, S. Russell, Y. Bi, and J. Yang. The object oriented parallel accelerator library (opal), design, implementation and application. In Proceedings ICAPO9, 2009.

[4] T. Aysal, M. Yildiz, A. Sarwate, and A. Scaglione. Broadcast gossip algorithms for consensus. Signal Processing, IEEE Transactions on, 57(7):2748–2761, 2009.

[5] P. Balaji, A. Ghosh, B. Prabhakar, and D. Shah. Randomized gossip algorithms. Information Theory, IEEE Transactions on, 52(6):2508–2530, 2006.

[6] K. Deb. Multi-Objective Optimization Using Evolutionary Algorithms. Wiley, 2009.

[7] M. Ferrario. Homdyn User Guide. Technical report, LNF, 2006. http://nicadd.niu.edu/fnpl/homdyn/manual.pdf.

[8] D. Karaboga. An idea based on honey bee swarm for numerical optimization. Techn. Rep. TR06, Erciyes Univ. Press, Erciyes, 2005.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. Science, 220(4598):671–680, May 1983.

[10] C. León, G. Miranda, and C. Segura. Metco: A parallel plugin-based framework for multi-objective optimization. International Journal on Artificial Intelligence Tools, 18(04):569–588, Aug 2009.

[11] A. Liefooghe, M. Basseur, L. Jourdan, and E. Talbi. Paradiseo-moeo: A framework for evolutionary multi-objective optimization. In Evolutionary multi-criterion optimization, pages 386–400. Springer, 2007.

[12] M. Pedrozzi, V. Arsov, B. Beutner, M. Dehler, A. Falone, W. Fichte, A. Fuchs, R. Ganter, C. Hauri, S. Hunziker, R. Ischebeck, H. Jhri, Y. Kim, M. N. ands P. Pearce, J.-Y. Raguin, 19
S. Reiche, V. Schlott, T. Schietinger, T. Schilcher, L. Schulz, W. Tron, D. Vermeulen, E. Zimoch, and J. Wickström. SwissFEL Injector Conceptual Design Report. Technical Report PSI-PR-10-05, Paul Scherrer Institut, 2010. [http://www.psi.ch/swissfel/CurrentSwissFELPublicationsEN/SwissFEL Injector CDR 310810.pdf](http://www.psi.ch/swissfel/CurrentSwissFELPublicationsEN/SwissFEL Injector CDR 310810.pdf)

[25] V. Pereyra, M. Saunders, and J. Castillo. Equispaced pareto front construction for constrained bi-objective optimization. *Mathematical and Computer Modelling*, 2011.

[26] H. Shah-Hosseini. The intelligent water drops algorithm: a nature-inspired swarm-based optimization algorithm. *International Journal of Bio-Inspired Computation*, 1(1):71–79, 2009.

[27] L. While, L. Bradstreet, and L. Barone. A fast way of calculating exact hypervolumes. *Evolutionary Computation, IEEE Transactions on*, 16(1):86–95, 2012.