Scientific Programming 11 (2003) 263–272
IOS Press

An interactive environment for supporting the transition from simulation to optimization[1]

Christian H. Bischof, H. Martin Bücker, Bruno Lang and Arno Rasch

Abstract. Numerical simulation is a powerful tool in science and engineering, and it is also used for optimizing the design of products and experiments rather than only for reproducing the behavior of scientific and engineering systems. In order to reduce the number of simulation runs, the traditional "trial and error" approach for finding near-to-optimum design parameters is more and more replaced with efficient numerical optimization algorithms. Done by hand, the coupling of simulation and optimization software is tedious and error-prone. In this note we introduce a software environment called EFCOSS (Environment For Combining Optimization and Simulation Software) that facilitates and speeds up this task by doing much of the required work automatically. Our framework includes support for automatic differentiation providing the derivatives required by many optimization algorithms. We describe the process of integrating the widely used computational fluid dynamics package FLUENT and a MINPACK-1 least squares optimizer into EFCOSS and follow a sample session solving a data assimilation problem.

1. Introduction

Traditionally, simulation software has been used mainly for reproducing–as exactly as possible–the behavior of scientific and engineering systems, thus complementing the two classical categories of scientific methodology: theory and experiment. While this kind of use certainly will remain important, there is increasing demand for embedding the simulation in a larger optimization framework.

A prominent example is *design optimization*, where one is looking for parameters $\mathbf{x}$ for a system such that some cost function $f(\mathbf{x})$ is minimized. Replacing experiments with simulation in the optimization process can drastically reduce the number of prototypes to be built before the final product emerges, and thus leads to considerable savings in money and time. Another optimization problem, *data assimilation*, stems from modeling and simulation itself. Here, the task is to try to adjust the values of certain model or simulation parameters such that the computed values $f(\mathbf{x})$ best match the data $\mathbf{d}$ obtained from actual experiments. Thus, the objective function of the corresponding optimization problem is given by $\|f(\mathbf{x})-\mathbf{d}\|$ in data assimilation and $f(\mathbf{x})$ in design optimization. In both cases, weighted norms may be used to emphasize selected components.

Traditionally, such optimization problems are often tackled by running the simulation over and over again with varying parameter sets and selecting the set leading to the "best" results. While easy to implement from a programming point of view, this procedure is not very efficient with respect to the number of simulation runs. Moreover, the selection of an "appropriate" parameter set may require experience. Numerical optimization routines, by contrast, typically make better use of available information and can also be used by non-experts. Unfortunately, optimization software and simulation software often follow different conventions for passing parameters, and therefore it is a tedious and error-prone task to combine these two components. In particular, switching to another optimizer or another simulation package requires rewriting the interfacing software, often from scratch.

This situation is addressed in [10], where a preliminary version of a framework for *automatically* combining large-scale simulation and optimization software via the Common Object Request Broker Architecture (CORBA) is proposed.

[1] This research is partially supported by the Deutsche Forchungsgemeinschaft (DFG) within SFB540 "Model-based experimental analysis of kinetic of phenomena in fluid multi-phase reactive systems," Aachen University, Germany.

ISSN 1058-9244/03/$8.00

The structure of this note is as follows. Sophisticated optimizers make use of derivative information in order to reduce the number of iterations. In Section 2 we discuss several methods for computing these derivatives. In particular, the so-called forward mode of Automatic Differentiation (AD) is reviewed. The structure of the Environment For Combining Optimization and Simulation Software (EFCOSS) is described in Section 3. Section 4 presents a detailed case study showing the steps that are necessary to integrate new simulation and optimization codes into EFCOSS and to run the optimization, including simple and efficient access to derivatives via AD. This section also includes results from numerical experiments. Our findings are summarized in Section 5.

2. Providing derivative information

This section addresses the problem of providing derivative information for large-scale simulations, as required by sophisticated optimization algorithms. We will briefly discuss the problems with three well-known methods for computing derivatives: analytic, symbolic, and numerical differentiation. Automatic differentiation is presented as a fourth alternative offering crucial advantages in the context of large-scale simulations.

If a function is given by an explicit formula or defined by simple differential or integral equations then the derivatives of that function may also be described in terms of formulæ. The mathematical description is then easily turned into code by hand. For functions given in the form of large-scale computer programs, complex effects such as turbulence typically preclude this *analytic differentiation* technique.

Due to the sheer size of the simulation packages totaling hundreds of thousands of lines and because of the complexity of the codes with their numerous branches, loops, and subroutine calls, current tools for *symbolic differentiation*–aimed at expressing derivatives as explicit formulæ, and then automatically producing code for evaluating the derivatives at arbitrary points $\mathbf{x}$ of interest–also fail [22].

If applicable, computer programs obtained from analytic and symbolic differentiation generate derivatives that are accurate except for rounding errors. *Numerical differentiation* based on Divided Differences (DD), by contrast, always involves an approximation error which grows with increasing step size, say h. Taking the first-order forward divided difference

$$\frac{\partial}{\partial x_j} f(\mathbf{x}) \tag{1}$$
$$\approx \frac{f(x_1, \ldots, x_{j-1}, x_j + h, x_{j+1}, \ldots, x_n) - f(\mathbf{x})}{h}$$

as an example, it is evident that this problem cannot be solved by using smaller step sizes h because, then, catastrophic cancellation in the numerator of Eq. (1) reduces the quality of the computed values. As a result, derivatives approximated via DD are often only valid to one half of the available digits, even with an optimal choice for h. The main advantage of the DD approach is that it is independent of the complexity of the function f, which is used only in a black-box fashion to be evaluated at certain points $\mathbf{x}$.

Automatic Differentiation (AD) is a fourth technique for the evaluation of derivatives capable of providing accurate values even for functions defined by arbitrarily complex codes. AD comprises a set of techniques for automatically augmenting a given computer program with statements for the computation of derivatives. That is, given a computer program C that computes a function

$$f(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x}))^T \in \mathbb{R}^m,$$

automatic differentiation generates another program C' that, at any point of interest $\mathbf{x} \in \mathbb{R}^n$, not only evaluates f but also its Jacobian

$$J(\mathbf{x}) := \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} f_1(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} f_m(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} f_m(\mathbf{x}) \end{pmatrix} \in \mathbb{R}^{m \times n}$$

at the same point $\mathbf{x}$.

The AD technology is applicable whenever derivatives of functions given in the form of a high-level programming language, such as Fortran, C, or C++, are required. The reader is referred to the recent book by Griewank [23] and the proceedings of AD workshops [7,12,24] for details on this technique. In automatic differentiation the program is treated as a–potentially very long–sequence of elementary statements such as binary addition or multiplication, for which the derivatives are known. Then the chain rule of differential calculus is applied over and over again, combining these step-wise derivatives to yield the derivatives of the whole program. This mechanical process can be automated, and several AD tools are available for transforming a given code C to the new *differentiated* code C' [8,9,20]. Other AD tools use operator overloading as implementation mechanism [4, 25]. In this way, AD requires little human effort and

produces derivatives that are accurate up to machine precision.

When accumulating the derivatives of elementary operations step by step, the associativity of the chain rule offers several alternatives, all leading to the same overall derivatives for the whole program, but at different cost with respect to computation and storage. One well-known strategy for applying the chain rule is the so-called forward mode of AD. If one is interested in obtaining derivatives of f with respect to n scalar variables (called *independent variables* hereafter), a gradient object $\mathbf{u}^\nabla \in \mathbb{R}^n$ is associated to every intermediate scalar variable u involved in the evaluation of the function f. The pair $u_d = [u, \mathbf{u}^\nabla]$ is called a *doublet*. In the sequel, u is called *function part* and its associated $\mathbf{u}^\nabla$ is referred to as *gradient part*. Note that the gradient part of a doublet stores the gradient of the function part with respect to the independent variables.

For every operation of the original code C involving a scalar variable u, there is a corresponding operation on the doublet $u_d = [u, \mathbf{u}^\nabla]$ in the differentiated code C'. For instance, a binary addition statement $u = v+w$ in C is transformed in C' into

$$\begin{aligned} u &= v + w \\ \mathbf{u}^\nabla &= \mathbf{v}^\nabla + \mathbf{w}^\nabla; \end{aligned}$$

that is, the separate addition of the function and gradient parts. A multiplication statement $u = v \cdot w$ is transformed into

$$\begin{aligned} u &= v \cdot w \\ \mathbf{u}^\nabla &= v\mathbf{w}^\nabla + w\mathbf{v}^\nabla, \end{aligned}$$

where the gradient part is defined in a product rule-like manner. Since any programming language consists of only a small set of operations, the set of corresponding operations on doublets is easily constructed.

Except for very simple cases, the function f cannot be evaluated within a single routine. Instead, evaluating f typically involves a large subtree of the whole program, the "top-level" routine of this tree invoking a multitude of lower-level routines and finally providing the function value. For example the computation of some characteristic number of a stationary flow may involve a nonlinear solver, which in turn calls a preconditioned linear solver, and so on. In such a case automatic differentiation must be applied to the whole subtree, often totaling several hundreds of routines and tens or even hundreds of thousands of lines of code, in order to obtain the function's derivatives.

In contrast to the forward mode, the so-called reverse (or backward) mode of AD generates derivative objects whose length is equal to the number of output or *dependent variables*, m, and can be more efficient than the forward mode if $m < n$.

Sophisticated forward mode AD tools are capable of generating code for the computation of $J(\mathbf{x}) \cdot S$, where S is a so-called *seed matrix* of appropriate dimension [23]. That is, besides computing the Jacobian matrix explicitly by setting S to the $n \times n$ identity, the seed matrix offers the option to compute any linear column combination of the Jacobian at a cost that is proportional to the number of columns of S. Thus, appropriately initializing S, called *seeding*, is often critical in terms of performance, provided that the full Jacobian J is not needed explicitly. Reverse mode AD tools typically allow the computation of any linear row combinations $S^T \cdot J(\mathbf{x})$ of the Jacobian.

3. The structure of EFCOSS

To give a better understanding of the structure of EFCOSS we first review the steps necessary for the solution of a typical data assimilation problem. Here, `f` denotes a subroutine evaluating the simulation function, and `opt` is used to refer to some optimization routine. We further assume that `opt` takes as input a user-supplied function `u` which is needed to compute the objective function. For instance, `u` might represent the difference vector $f(\mathbf{x}) - \mathbf{d}$ or its norm.

- First, a driver for `opt` has to be implemented, which provides an initial guess $\mathbf{x}_0$ and potentially additional control parameters specifying stopping criteria, constraints, etc.
- The function `u` has to be implemented such that `f` is called, then the return values $f(\mathbf{x})$ are compared with the measurements $\mathbf{d}$, and finally the results are returned. Note that the calling sequence of `u` is usually prescribed by the developers of the optimization software. Hence, the user has to fit his particular optimization problem into a given scheme.
- There is often a similar scheme for the routine `du` computing the derivative of `u`. Therefore a mechanism for evaluating the derivatives of `f` is needed.
- To increase flexibility it is often desired to keep some of the input parameters of `f` at fixed values, i.e., only a subset of the parameters of `f` should be optimized. Hence, there is need for a mechanism to specify fixed parameters $\mathbf{p}$.

The computational scientist often considers various simulation packages and, more importantly, different optimization codes in order to validate the robustness of the numerical solution. Instead of implementing the above requirements several times by hand, we suggest an automated way for easily combining different software packages and experimenting with varying problem configurations.

A related approach is the Toolkit for Advanced Optimization (TAO) [5,6] designed as a component-based optimization software for the solution of large-scale optimization applications. It is based on the Portable, Extensible Toolkit for Scientific Computation (PETSc) [1, 2], an object-oriented implementation for the parallel solution of large problems arising from partial differential equations. A different approach is the NEOS project [14]. The NEOS environment allows users to solve an optimization problem remotely on an optimization server, which offers a rich variety of optimization algorithms. The user submits an initial guess, possibly some control parameters, and a subroutine for evaluating the objective function to the NEOS server, where the problem is solved with the selected optimizer. This approach is easy to use, highly flexible, and adjustable to large heterogeneous clusters of workstations using a batch processing mechanism [16].

However, it is not applicable to our class of problems, where the evaluation of the objective function involves a complete run of a typically very large simulation code. In addition to sheer size, the simulation code might be tuned for a specific architecture, e.g., for a parallel or vector supercomputer; or it might be protected by copyright laws and therefore cannot be submitted via the Internet. There is a recent attempt to circumvent these difficulties for optimization problems where the objective function is evaluated locally while the optimization algorithm runs on a remote server [21].

Other Projects involving object-orientation and interoperability in the context of computational science include the Parallel Object-Oriented Methods and Applications (POOMA) framework [28] and Overture [11]; see also the contributions in [26] for more details. The CAPE-OPEN initiative provides an open standard for software interoperability in chemical process engineering. More details about this project can be found in [3]. Commercial process simulation environments like Aspen Plus provide integrated optimization routines and allow users to hook up their own optimizers. Furthermore, design optimization tools, e.g., iSIGHT, can be used together with several different commercial simulation codes.

In [10] we proposed a preliminary version of EFCOSS, a software environment especially designed for *automatically* combining optimization routines and large-scale simulation codes. Our approach treats the evaluation of the function `f`, the execution of one iteration of the optimizer `opt`, and the computation of the user-defined function `u` as basic tasks and provides an infrastructure for controlling the interplay of these tasks. To achieve maximum flexibility in supporting different platforms and languages, EFCOSS is based on the CORBA technology. The overall structure is depicted in Fig. 1.

The user specifies the input and output variables of the simulation code as well as the variables needed to optimize the objective function. From this specification, C++ interfaces for the evaluation of the function `f` and its derivatives are generated automatically. They are needed to set up a so-called "simulation server", which is able to drive the evaluation of the simulation function as well as the specified gradient or Jacobian. All requests to the simulation server come from a standard module called "wrapper" hereafter. The wrapper is responsible for transferring data between the simulation and optimization components and also for the computation of the user-defined function `u`. Also making use of the specification mentioned above, the wrapper sends a request to the simulation server for evaluating either $f(\mathbf{x}, \mathbf{p})$ or the derivative of $f(\mathbf{x}, \mathbf{p})$ w.r.t. $\mathbf{x}$. Here, the variables needed for optimization are denoted by $\mathbf{x}$, whereas $\mathbf{p}$ represents additional fixed parameters of the simulation. The specification of the simulation's input parameters also includes the values for such fixed parameters.

The whole computation is driven by the optimization component which repeatedly calls the user-defined subroutine `u` and, possibly, `du` for the evaluation of the objective function and its derivative, respectively. In EFCOSS, these two routines are just stubs that perform no computational work on their own but only call a standardized C++ interface, which in turn makes use of CORBA to send an evaluation request, together with an argument $\mathbf{x}$, to the wrapper. The wrapper then forwards this request to the simulation server, complementing the free simulation parameters $\mathbf{x}$ with the fixed values $\mathbf{p}$.

The complete optimization process is invoked via a user interface, which also makes the measurements $\mathbf{d}$ available to the wrapper. The user interface is implemented via the Python shell and provides some auxiliary functions, e.g., for communication with the wrapper module and for specifying input and output param-

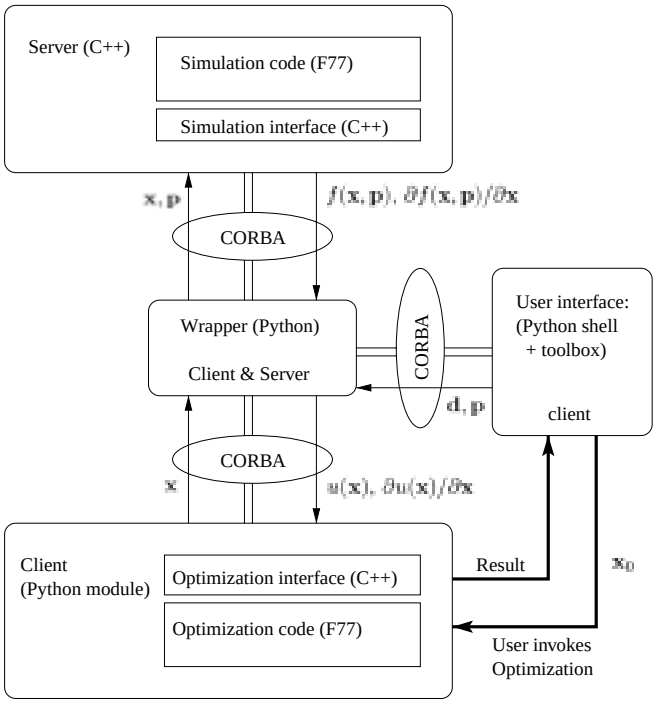

Fig. 1. Overall structure of EFCOSS.

eters of the simulation. This allows users to experiment with different problem configurations either interactively or through scripts. The actual implementation of EFCOSS uses omniORB3.0 [27] for C++ and Fnorb [13] for Python.

4. Case studies

In this section we describe the integration of particular software components into EFCOSS, namely the widely used FLUENT computational fluid dynamics (CFD) package [17], and an optimization routine from the MINPACK-1 library [18]. Then we show the use of EFCOSS by means of a test example taken from the FLUENT tutorial guide [17].

4.1. Integration of the FLUENT solver

In order to integrate any simulation package into EFCOSS, we must be able to control the simulation through one single routine, the so-called "top-level routine". This does not mean that the complete simulation code must be contained in one routine but that the input and output values of the simulation are accessible within that routine.

For FLUENT, we had to turn off the graphical user interface. The remaining text-based version of FLUENT can be controlled via a so-called *log file* providing a complete specification of the simulation problem. Furthermore, we had to replace the main program by a top-level routine, which takes the relevant input parameters, executes the commands given in the log file, and returns the results.

As a test problem, we consider the flow through a filter cartridge as described in the FLUENT tutorial. Here, we want to adjust the five scalar model parameters $c_{1\varepsilon}, c_{2\varepsilon}, c_\mu, \sigma_k, \sigma_\varepsilon$ of the k-ε turbulence model such that the pressure distribution in the filter best matches some given experimental data. Therefore, the top-level routine `filter` calculating the simulation function has the following structure:

```
      subroutine filter(c1,c2,cmu,
     + eprnd,dprnd,
     + ldpressure,pressure)
      integer ldpressure
      double precision c1,c2,cmu,
     + eprnd,dprnd
     + pressure(ldpressure)

c- open log file: channel 5 (= stdin)
c- is redirected to file.
```

```
c- Problem specification is given
c- in FILTER_LOGFILE
       open(unit=5,
      + file='FILTER_LOGFILE')

c- original FLUENT code

c- close channel 5 (stdin)
       close(5)
       end
```

Here, the input variables `c1`, `c2`, `cmu`, `eprnd`, and `dprnd` correspond to the turbulence parameters $c_{1\varepsilon}$, $c_{2\varepsilon}$, c_μ, σ_k, and σ_ε. Note that we further changed the original FLUENT code such that it uses the values of these input variables instead of the internal default values for the turbulence parameters. On exit, the routine `filter` computes the output variable `pressure` by executing the FLUENT code.

The next step is to provide a specification of the input and output variables. For simplicity, all variables are assumed to be `double precision`. Hence, the specification just consists of array size information. Furthermore, the top-level routine of the simulation and a set of variables to optimize, which is a subset of the simulation's input variables, must be defined. In our test example we want to enable all input variables for optimization.

The specification is given via the user interface, and is needed to generate the CORBA/C++ interface connecting the simulation routine to the wrapper. It will be also used later during the optimization process.

The generated CORBA/C++ routine takes a sequence of input variables (possibly vectors) from the wrapper, allocates memory for the output variables, then calls the simulation's top-level function with the input values, and finally returns one or more solution vectors to the wrapper. An excerpt from the CORBA/C++ interface generated for our particular test example is given below:

```
// input:  sequence of input
// vectors ''x_in''
// output: sequence of solution
// vectors ''all_solutions''
  double c1,c2,cmu,eprnd,dprnd;
  int ldpressure = 336;
  double *pressure =
            new double[ldpressure];
  c1 = x_in[0][0];
  c2 = x_in[1][0];
  cmu = x_in[2][0];
  eprnd = x_in[3][0];
  dprnd = x_in[4][0];
  filter(&c1,&c2,&cmu,&eprnd,
       &dprnd,&ldpressure,pressure);
// copy pressure to
// all_solutions[0]
  ...
  delete[] pressure;
  return all_solutions._retn();
```

If desired, EFCOSS additionally creates a control script which can be used by the automatic differentiation tool ADIFOR [8] to transform the simulation source code into new code with additional statements for the computation of the derivatives of the simulation's outputs w.r.t. the input variables selected for optimization. In the test example, we are interested in the derivatives of the dependent variable `pressure` w.r.t. the five independent variables representing the turbulence parameters. This information is passed to the AD tool by specifying the directives `AD_DVARS`, `AD_IVARS`, and `AD_PMAX`. The top-level routine `filter` is indicated by the directive `AD_TOP`. EFCOSS generates a control script for ADIFOR containing the following directives:

```
AD_TOP=filter
AD_PMAX=5
AD_IVARS=c1,c2,cmu,eprnd,dprnd
AD_DVARS=pressure
AD_PROG=filter.cmp
```

Here, `filter.cmp` is the name of a so-called composition file listing the names of all source files of the simulation. For programs fully adhering to the Fortran 77 standard, applying the ADIFOR tool results in Fortran code for the desired derivatives. However, in practice, legacy code may need some manual rearrangements before the ADIFOR tool can be successfully applied. For the FLUENT code with its approximately $1,500,000$ lines of mostly Fortran 77, some additional code massaging was necessary in order to obtain standard Fortran 77. Note that this preparation of the code has to be done only once, even if many different optimization problems are considered later on.

The calling sequence of the differentiated top-level routine generated by ADIFOR is given below:

```
      subroutine g_filter(g_p_,
     + c1,g_c1,ldg_c1, c2,g_c2,
     + ldg_c2, cmu,g_cmu,ldg_cmu,
     + eprnd,g_eprnd,ldg_eprnd,
     + dprnd,g_dprnd,ldg_dprnd,
     + ldpressure,pressure,
     + g_pressure,ldg_pressure)
```

It is easy to see how the calling sequence of the augmented routine `g_filter` is determined by the original top-level routine: all variables appearing in the parameter list of `filter` reappear in `g_filter`; each variable `v` corresponding to an independent or dependent variable (as listed in `AD_IVARS` and `AD_DVARS` in the ADIFOR script) is immediately followed by two additional variables `g_v` and `ldg_v` where the derivatives of `v` are stored in an array `g_v` with leading dimension `ldg_v`. In the above parameter list, the seed matrix S, introduced in Section 2, consists of the input variables `g_c1`, `g_c2`, `g_cmu`, `g_eprnd`, and `g_dprnd` with corresponding leading dimensions `ldg_c1`, `ldg_c2`, `ldg_cmu`, `ldg_eprnd`, and `ldg_dprnd`. The derivative of the pressure field is returned in `g_pressure`, a two dimensional array of size (`ldg_pressure`, `ldpressure`).

Similar to the interface of `filter`, EFCOSS generates the CORBA/C++ interface for `g_filter` which also performs the proper seeding of the independent variables according to the user's specification. For this reason, the CORBA/C++ interface routine for the differentiated code gets not only the input values for the simulation from the wrapper but also the actual seed matrix in a condensed representation. The seeding is then done automatically within the interface routine. The other tasks of this routine are similar to the one described above for the simulation–wrapper interface, except that now the *differentiated* top-level routine is called, and the derivatives are returned instead of the corresponding solution vectors. This "automatic seeding" allows the user to interactively reduce the set of parameters to optimize, even *after* the generation of the AD code and the CORBA interfaces.

To further illustrate this issue, we consider two basic strategies for choosing parameters for optimization.

Method 1: The user specifies only those parameters that are definitely to be optimized, and generates the differentiated program suitable for this particular case. If, later on, it turns out that more (or other) parameters should be considered for optimization then the AD code must be regenerated, and the CORBA interface for the new differentiated code must be generated as well. Since both tasks are carried out in a completely mechanical fashion, the only disadvantage of this approach is the processing time to be invested. In the case of very large codes like FLUENT, automatic differentiation and compilation may take several hours.

Method 2: The user specifies all input parameters that might possibly become relevant for optimization, and generates AD code as well as CORBA interfaces for this configuration. The number of parameters to optimize may now be reduced. Since the current set of optimization parameters is always known to the wrapper module, the seed matrix can be used to filter out the corresponding partial derivatives. Thus, whenever the set of optimization parameters is changed, the seed matrix is changed respectively. This approach does not need any recompilation at all, i.e., the set of optimization parameters can be changed interactively. On the other hand, this approach typically requires more memory at run-time.

Of course, mixing the two strategies is possible.

In order to build the complete *simulation server* shown in Fig. 1, the simulation function, the differentiated code, and the corresponding CORBA/C++ interfaces have to be linked with a small main program, which is independent from the actual simulation.

4.2. Integration of a MINPACK-1 optimization routine

Typical optimization routines require subroutines for the evaluation of the objective function or its derivative. These subroutines must be provided by the user. Thus, these two subroutines provide the means for coupling the optimizers with the simulation software. In EFCOSS, this is done via the wrapper; see Fig. 1.

To give an example, we discuss the process of integrating the least squares optimization routine `lmder1` from the MINPACK-1 library into EFCOSS. The MINPACK package is publicly available, e.g., from `http://www.netlib.org`. According to [18] the user-defined subroutine must have the following structure:

```
      subroutine fcn (m,n,x,fvec,
     + fjac,ldfjac,iflag)
      integer n, m, ldfjac, iflag
      double precision x(n), fvec(m),
     + fjac(ldfjac,n)
      if (iflag .eq. 1) then
c --- calculate the functions at x
c --- return this vector in fvec
      end if
      if (iflag .eq. 2) then
c --- calculate the Jacobian at x
c --- return this matrix in fjac
      end if
      end
```

Usually the user implements the code for the required computations in this routine. In EFCOSS, by contrast, the user only needs to put the following subroutine calls at the appropriate places:

```
call calcfvec(n,x,m,fvec)
```

for the function, and

```
call calcjac(n,x,m,fjac)
```

for the Jacobian. These external routines perform no computational work, but send a request to the wrapper for evaluating the objective function and the Jacobian, respectively. The routines can be used whenever a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ or its Jacobian has to be computed. For convenience, we also provide specialized interface routines tailored to the evaluation of scalar-valued functions and gradients.

The header of the main optimization routine is given below:

```
 subroutine lmder1(fcn,m,n,x,
+ fvec,fjac,ldfjac,tol,
+ info,ipvt,wa,lwa)
 integer m,n,ldfjac,lwa,info
 integer ipvt(n)
 double precision tol
 double precision x(n), fvec(m),
+ fjac(m,n), wa(lwa)
 external fcn
```

To make this routine available from within Python we employ the Pyfort tool [15]. Since Pyfort cannot handle function names in a subroutine's calling sequence, we remove the first argument. After this modification, we utilize Pyfort to create a shared library `minpackmodule.so`, which can be accessed from Python. Note that the resulting optimization module is independent from the actual problem configuration because the evaluation of the objective function and the Jacobian is separated from the optimization part. Therefore, the steps described in this subsection have to be carried out only once, and from this point on the generated Python module is usable for solving arbitrary optimization problems within EFCOSS.

4.3. A sample optimization session

In the following we will show how EFCOSS can be used to solve a typical data assimilation problem. For the sample problem of a flow through a filter cartridge [17], we consider a turbulent flow at a Reynolds number $\mathrm{Re} \approx 10^5$ using the k-ε turbulence model.

We want to determine values for three of the turbulence parameters, $c_{1\varepsilon}$, $c_{2\varepsilon}$, and c_μ, such that the pressure distribution computed with FLUENT best matches given experimental data at certain grid points. For the time being, the remaining two parameters $\sigma_k = 1.0$ and $\sigma_\varepsilon = 1.3$ are considered fixed. But as we do not know if these two values are correct, they might be included in later optimization problems.

In our example, the test data $\mathbf{d}$ has been generated artificially by running the simulation with the parameter set

$$c_{1\varepsilon} = 1.44, c_{2\varepsilon} = 1.92, c_\mu = 0.09, \sigma_k = 1.0,$$
$$\sigma_\varepsilon = 1.3$$

and saving the results for the pressure distribution to a file.

The following extract from a Python session shows how we set up the optimization problem:

```
1. c1_eps = newInputVar(wrapper,
   "c1", 1,[1.44])
2. c2_eps = newInputVar(wrapper,
   "c2",1,[1.92])
3. cmu = newInputVar(wrapper,
   "cmu", 1,[0.09])
4. sigma_k = newInputVar(wrapper,
   "eprnd",1,[1.0])
5. sigma_eps = newInputVar(wrapper,
   "dprnd",1,[1.3])
6. setInputVars(wrapper,[c1_eps,
   c2_eps,cmu,sigma_k,sigma_eps])
7. pressure = newOutputVar(wrapper,
   "pressure",336,1)
8. setOutputVars(wrapper,[pressure])
9. setOptVars(wrapper,[c1_eps,
   c2_eps, cmu,sigma_k,sigma_eps])
```

In lines 1–6 the input variables are specified. For each input variable we indicate its name in the simulation code, its size, and its default value. In lines 7 and 8 we specify the output variable. The solution vector `pressure` is defined as an array of size 336, corresponding to the number of cells in the underlying grid. In line 9 we specify all the input variables that might be optimized later on.

At this point we can generate a control script for ADIFOR as well as the CORBA interfaces for the routines `filter` and `g_filter`. Then, ADIFOR is applied to obtain the differentiated code for `g_filter`. Note that, due to our definition of the optimization variables in line 9, `g_filter` will be able to compute derivatives with respect to all five parameters or any subset of them, depending on the actual seeding. Considering σ_k and σ_ε to be fixed, we are going to optimize the subset $c_{1\varepsilon}, c_{2\varepsilon}$, and c_μ. Hence, we redefine the set of optimization variables and provide an initial guess for these three variables:

```
setOptVars(wrapper,
           [c1_eps,c2_eps,cmu])
x = array([1.1,1.2,0.1],Float64)
```

After setting further control parameters for the MINPACK optimizer and providing the test data **d** to the wrapper, we call the optimization routine `lmder1` from the MINPACK module. In addition, we print the result `x` and the final Euclidean norm of the residual vector.

```
import minpack
fvec,fjac,info,ipvt=
  minpack.lmder1(336,3,x,ldfjac,
                 tol,wa,lw)
print x,minpack.enorm(336,fvec)
```

After six function calls and five evaluations of the Jacobian, the optimization algorithm produces the output

```
[1.44000001 1.91999998 0.09000001]
3.39489005512e-06
```

Here, the first three numbers are the final approximations for the variables $c_{1\varepsilon}, c_{2\varepsilon}$, and c_μ whereas the fourth number is the Euclidean norm of the residual vector. Indeed, the solution shows a good agreement with the parameter set that was used to generate the test data. However, in general, users may want to verify the robustness of the solution, e.g., by trying a different optimization package. Thus we let the same problem be solved again by another optimizer, namely the bound-constrained least squares optimization routine `dn2gb` from the PORT [19] library, which has been integrated in EFCOSS as well. We import the corresponding Python module, and provide the initial guess as above. The array `bounds` will be used to pass the constraints to the optimization routine. For sake of brevity the initialization of the remaining variables is omitted here.

```
import port
x = array([1.1,1.2,0.1],Float64)
lower_bounds = [0.01]*3
upper_bounds = [2.0]*3
bounds = array([lower_bounds,
                upper_bounds],
                Float64)
port.dn2gb(336,3,x,bounds,iv,liv,
                   lv,v,ui,ur)
```

In this optimization problem, the solution found by `dn2gb` agrees with the solution computed by `lmder1`. The crucial point here is that another optimizer can be used with minimal human effort. More precisely, any optimization algorithm accessible from within EFCOSS can be used by interactively specifying its control parameters.

5. Conclusions

We have presented EFCOSS, a software environment for facilitating the combination of simulation and optimization software by enabling much of the interfacing work to be done automatically. Our approach treats the evaluation of the simulation function, one iteration of the optimizer, and the computation of the optimizer's objective function as basic tasks and provides an infrastructure for the interplay of these tasks.

A considerable number of languages and tools play together in EFCOSS. Object-oriented languages are used for managing the data and control flows between the different components at the executable level, whereas we rely on highly optimizable imperative languages, e.g., Fortran, for the computationally intensive tasks. One reason for selecting CORBA is that it greatly simplifies distributed execution. For example, in our experiments the simulation ran on a SUN Fire 6800 high-end compute server, whereas a standard PC was used for the optimizer and the user interface. The C++ interfaces to the Fortran codes are added to simplify the remote calls via CORBA. The wrapper and the user interface are written in Python because of its flexibility and ease of use. Finally, the ADIFOR automatic differentiation tool is employed for automatically augmenting the simulation code such that it computes derivative information together with the function values. Note that, due to the use of CORBA, the simulation and the optimization can be implemented in different languages. The optimizer is required to be callable from Python which holds at least for Fortran, C, and C++. In order to use automatic differentiation, the simulation code should be written in a language that is supported by some AD tool. Otherwise, one can still use EFCOSS, but with a loss of accuracy when approximating the derivatives by numerical differentiation.

In addition to the FLUENT simulation package and the least squares optimizers from the MINPACK-1 and PORT libraries which are mentioned in this note, we have integrated another large simulation package, SEPRAN [30], as well as optimizers for scalar-valued objective functions like, e.g., L-BFGS-B [31] and UNCMIN [29], thus showing the versatility of EFCOSS. Due to its modular structure, components can

easily be replaced with others providing comparable functionality. Thus, experimenting with different simulation packages and/or different optimization algorithms is greatly simplified.

References

[1] S. Balay, W.D. Gropp, L.C. McInnes and B.F. Smith, PETSc 2.0 users manual, Technical Report ANL-95/11-Revision 2.0.24, Argonne National Laboratory, 1999.

[2] S. Balay, W.D. Gropp, L.C. McInnes and B.F. Smith, PETSc home page, http://www.mcs.anl.gov/petsc, 1999.

[3] J.-P. Belaud, B. Braunschweig and M. White, The CAPE-OPEN standard: Motivations, development process, technical architecture and examples, in: *Software Architectures and Tools for Computer Aided Process Engineering*, B. Braunschweig and R. Gani, eds, Elsevier, Amsterdam, Netherlands, 2002, pp. 303–332.

[4] C. Bendtsen and O. Stauning, FADBAD, a flexible C++ package for automatic differentiation, Technical Report IMM-REP-1996-17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, August, 1996.

[5] S. Benson, L. Curfman McInnes and J. Moré, TAO users manual, Technical Report ANL/MCS-TM-242 Revision 1.2, Mathematics and Computer Science Division, Argonne National Laboratory, 2001.

[6] S.J. Benson, L.C. McInnes and J.J. Moré, A case study in the performance and scalability of optimization algorithms, *ACM Trans. Math. Softw.* **27**(3) (2001), 361–376.

[7] M. Berz, C. Bischof, G. Corliss and A. Griewank, Computational Differentiation: Techniques, Applications, and Tools. SIAM, Philadelphia, PA, 1996.

[8] C. Bischof, A. Carle, P. Khademi and A. Mauer, ADIFOR 2.0: Automatic differentiation of Fortran 77 programs, *IEEE Computational Science and Engineering* **3**(3) (1996), 18–32.

[9] C. Bischof, L. Roh and A. Mauer, ADIC – An extensible automatic differentiation tool for ANSI-C, *Software: Practice and Experience* **27**(12) (1997), 1427–1456.

[10] C.H. Bischof, H.M. Bücker, B. Lang, A. Rasch and J.W. Risch, A CORBA-based environment for coupling large-scale simulation and optimization software, in: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, H.R. Arabnia, ed., PDPTA 2001, Las Vegas, USA, CSREA Press, Vol. 1, June, 25–28, 2001, pp. 68–72.

[11] D.L. Brown, G.S. Chesshire, W.D. Henshaw and D.J. Quinlan, *OVERTURE: An object-oriented software system for solving partial differential equations in serial and parallel environments*, in Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March, 1997, pp. 14–17.

[12] G. Corliss, C. Faure, A. Griewank, L. Hascoët and U. Naumann, eds, Automatic Differentiation of Algorithms: From Simulation to Optimization, Springer, New York, 2002.

[13] CRC for Distributed Systems Technology, The University of Queensland, Australia, The Python CORBA ORB, 1999, http://www.fnorb.org.

[14] J. Czyzyk, M.P. Mesnier and J.J. Moré. The NEOS server, *IEEE Computational Science and Engineering* **5**(3) (1998), 68–75.

[15] P.F. Dubois and T.-Y. Yang, Extending Python with Fortran, *Computing in Science and Engineering* **1**(5) (1999), 66–73.

[16] M.C. Ferris, M.P. Mesnier and J.J. Moré, NEOS and Condor: Solving optimization problems over the internet, *ACM Trans. Math. Softw.* **26**(1) (2000), 1–18.

[17] Fluent Inc., Lebanon, NH. FLUENT Tutorial Guide, 1995.

[18] B.S. Garbow, K.E. Hillstrom and J.J. Moré, User Guide for MINPACK-1, Report ANL-80-74, Argonne National Laboratory, Argonne, 1980.

[19] D.M. Gay, Usage summary for selected optimization routines, Computing Science Technical Report 153, AT&T Bell Laboratories, Murray Hill, 1990.

[20] R. Giering and T. Kaminski, Recipes for adjoint code construction, *ACM Trans. Math. Softw.* **24**(4) (1998), 437–474.

[21] M. Good, J.-P. Goux, J. Nocedal and V. Pereyra, iNEOS: An interactive environment for nonlinear optimization, *Applied Numerical Mathematics* **40**(1–2) (2002), 49–57.

[22] A. Griewank, On automatic differentiation, in: *Mathematical Programming: Recent Developments and Applications*, M. Iri and K. Tanabe, eds, Kluwer Academic Publishers, Dordrecht, 1989, pp. 83–108,

[23] A. Griewank, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, SIAM, Philadelphia, PA, 2000.

[24] A. Griewank and G. Corliss, Automatic Differentiation of Algorithms, SIAM, Philadelphia, PA, 1991.

[25] A. Griewank, D. Juedes and J. Utke, ADOL-C, a package for the automatic differentiation of algorithms written in C/C++, *ACM Trans. Math. Softw.* **22**(2) (1996), 131–167.

[26] M.E. Henderson, C.R. Anderson and S. L.Lyons, eds, Proceedings of the 1998 SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing, Philadelphia, PA, 1999, SIAM.

[27] S.-L. Lo, D. Riddoch and D. Grisby, The omniORB Version 3.0 User's Guide, AT&T Bell Laboratories, Cambridge, May 2000, Available from http://www.uk.research.att.com/omniORB.

[28] J.V.W. Reynders et al., POOMA: A framework for scientific simulations on parallel architectures, in: *Parallel Programming Using C++*, G.V. Wilson and P. Lu, eds, MIT Press, Cambridge, 1996, pp. 547–588.

[29] R.B. Schnabel, J.E. Koontz and B.E. Weiss, A modular system of algorithms for unconstrained minimization, *ACM Trans. Math. Softw.* **11** (1985), 419–440.

[30] G. Segal, SEPRAN Users Manual, Ingenieursbureau Sepra, Leidschendam, NL, 1993.

[31] C. Zhu, R.H. Byrd, P. Lu and J. Nocedal, Algorithm 778: L-BFGS-B, Fortran subroutines for large-scale bound constrained optimization, *ACM Trans. Math. Softw.* **23** (1997), 550–560.