Mathematical Modelling and Numerical Analysis ESAIM: M2AN
Modélisation Mathématique et Analyse Numérique M2AN, Vol. 36, N° 5, 2002, pp. 747–771
DOI: 10.1051/m2an:2002035

A MATHEMATICAL AND COMPUTATIONAL FRAMEWORK FOR RELIABLE REAL-TIME SOLUTION OF PARAMETRIZED PARTIAL DIFFERENTIAL EQUATIONS

Christophe Prud'homme[1], Dimitrios V. Rovas[1], Karen Veroy[1] and Anthony T. Patera[1]

Abstract. We present in this article two components: these components can in fact serve various goals independently, though we consider them here as an ensemble. The first component is a technique for the *rapid and reliable evaluation* prediction of linear functional outputs of elliptic (and parabolic) partial differential equations with affine parameter dependence. The essential features are (*i*) (provably) rapidly convergent global reduced–basis approximations — Galerkin projection onto a space W_N spanned by solutions of the governing partial differential equation at N selected points in parameter space; (*ii*) *a posteriori* error estimation — relaxations of the error–residual equation that provide inexpensive yet sharp and rigorous bounds for the error in the outputs of interest; and (*iii*) off–line/on–line computational procedures — methods which decouple the generation and projection stages of the approximation process. This component is ideally suited — considering the operation count of the online stage — for the repeated and rapid evaluation required in the context of parameter estimation, design, optimization, and real–time control. The second component is a framework for distributed simulations. This framework comprises a library providing the necessary abstractions/concepts for distributed simulations and a small set of tools — namely SimTEX and SimLaB— allowing an easy manipulation of those simulations. While the library is the backbone of the framework and is therefore general, the various interfaces answer specific needs. We shall describe both components and present how they interact.

Mathematics Subject Classification. 65N15, 65N30, 68U01, 68U20, 68M14, 68M15.

Received: 23 January, 2002. Revised: 4 June, 2002.

1. Introduction to reduced basis output bound methods

The optimization, control, and characterization of an engineering component or system requires the prediction of certain "quantities of interest", or performance metrics, which we shall denote *outputs* — for example deflections, maximum stresses, maximum temperatures, heat transfer rates, flowrates, or lift and drags. These outputs are typically expressed as functionals of field variables associated with a parametrized partial differential equation which describes the physical behavior of the component or system. The parameters, which we

Keywords and phrases. Mathematical framework, reduced-basis methods, error bounds, computational framework, simulations repository, distributed and parallel computing, CORBA, C++.

[1] Massachusetts Institute of Technology, Department of Mechanical Engineering, Room 3-266, 77 Massachusetts Ave., Cambridge, MA 02139, USA. e-mail: prudhomm@MIT.EDU

© EDP Sciences, SMAI 2002

shall denote *inputs*, serve to identify a particular "configuration" of the component: these inputs may represent design or decision variables, such as geometry — for example, in optimization studies; control variables, such as actuator power — for example in real–time applications; or characterization variables, such as physical properties — for example in inverse problems. We thus arrive at an implicit *input–output* relationship, evaluation of which demands solution of the underlying partial differential equation.

Our goal is the development of computational methods that permit *rapid* and *reliable* evaluation of this partial-differential-equation-induced input-output relationship *in the limit of many queries* — that is, in the design, optimization, control, and characterization contexts. The "many query" limit has certainly received considerable attention: from "fast loads" or multiple right-hand side notions (*e.g.* [5,7]) to matrix perturbation theories (*e.g.* [1,25]) to continuation methods (*e.g.* [2,20]). Our particular approach is based upon the reduced–basis method, first introduced in the late 1970s for nonlinear structural analysis [3, 13], and subsequently developed more broadly in the 1980s and 1990s [4,8,16,17,21]. The reduced–basis method recognizes that the field variable is not, in fact, some arbitrary member of the infinite-dimensional space associated with the partial differential equation; rather, it resides, or "evolves", on a much lower–dimensional manifold induced by the parametric dependence.

The reduced–basis approach as earlier articulated is local in parameter space in both practice and theory. To wit, Lagrangian or Taylor approximation spaces for the low–dimensional manifold are typically defined relative to a particular parameter point; and the associated *a priori* convergence theory relies on asymptotic arguments in sufficiently small neighborhoods [8]. As a result, the computational improvements — relative to conventional (say) finite element approximation — are quite modest [17]. Our work differs from these earlier efforts in several important ways: first, we develop (in some cases, probably) *global* approximation spaces; second, we introduce rigorous *a posteriori error estimators*; and third, we exploit *off–line/on–line* computational decompositions. These three ingredients allow us — for a restricted but important class of problems — to reliably decouple the generation and projection stages of reduced–basis approximation, thereby effecting computational economies of several orders of magnitude.

In this expository review paper we focus on these new ingredients. We begin in Section 2 by introducing an abstract problem formulation and several illustrative instantiations. In Section 3 we describe the reduced–basis approximation for coercive symmetric problems and "compliant" outputs; associated *a posteriori* estimators are then developed in Section 4.

2. Problem statement

2.1. **Abstract formulation**

We consider a suitably regular domain $\Omega \subset \mathbb{R}^d$, $d = 1,\ 2$, or 3, and associated function space $X \subset H^1(\Omega)$, where $H^1(\Omega) = \{v \in L^2(\Omega),\ \nabla v \in (L^2(\Omega))^d\}$, and $L^2(\Omega)$ is the space of square integrable functions over Ω. The inner product and norm associated with X are given by $(\cdot\,,\cdot)_X$ and $\|\cdot\|_X = (\cdot\,,\cdot)^{1/2}$, respectively. We also define a parameter set $\mathcal{D} \in \mathbb{R}^P$, a particular point in which will be denoted μ. Note that Ω does *not* depend on the parameter.

We then introduce a bilinear form $a\colon\ X \times X \times \mathcal{D} \to \mathbb{R}$, and linear forms $f\colon\ X \to \mathbb{R}$, $\ell\colon\ X \to \mathbb{R}$. We shall assume that a is continuous, $a(w, v; \mu) \ \leq\ \gamma(\mu)\ \|w\|_X\ \|v\|_X \leq \gamma_0\ \|w\|_X\ \|v\|_X$, $\forall \mu \in \mathcal{D}$; furthermore, in Sections 3 and 4, we assume that a is coercive,

$$0 < \alpha_0 \leq\ \alpha(\mu) = \inf_{w \in X} \frac{a(w, w; \mu)}{\|w\|_X^2}, \quad \forall\, \mu \in \mathcal{D}, \tag{1}$$

and symmetric, $a(w, v; \mu) = a(v, w; \mu)$, $\forall\, w, v \in X$, $\forall\, \mu \in \mathcal{D}$. We also require that our linear forms f and ℓ be bounded; in Sections 3 and 4 we additionally assume a "compliant" output, $f(v) = \ell(v)$, $\forall\, v \in X$.

We shall also make certain assumptions on the parametric dependence of a, f, and ℓ. In particular, we shall suppose that, for some finite (preferably small) integer Q, a may be expressed as

$$a(w,v;\mu) \;=\; \sum_{q=1}^{Q} \sigma^q(\mu)\; a^q(w,v), \quad \forall\; w,v \in X,\; \forall\; \mu \in \mathcal{D}, \tag{2}$$

for some $\sigma^q\colon\; \mathcal{D} \to \mathbb{R}$ and $a^q\colon\; X \times X \to \mathbb{R}$, $q = 1,\dots,Q$. This "separability", or "affine", assumption on the parameter dependence is crucial to computational efficiency; however, certain relaxations are possible — see [19]. For simplicity of exposition, we assume that f and ℓ do not depend on μ; in actual practice, affine dependence is readily admitted.

Our abstract problem statement is then: for any $\mu \in \mathcal{D}$, find $u(\mu) \in X$ such that

$$a(u(\mu),v;\mu) \;=\; f(v), \quad \forall\; v \in X; \tag{3}$$

and $s(\mu) \in \mathbb{R}$ given by

$$s(\mu) = \ell(u(\mu)). \tag{4}$$

In the language of the introduction, a is our partial differential equation (in weak form), μ is our parameter, $u(\mu)$ is our field variable, and $s(\mu)$ is our output.

For simplicity, we may suppress the μ-dependence along the article when there is no possible confusion.

2.2. **Particular instantiations**

We indicate here a few instantiations of the abstract formulation; these will serve to illustrate the methods (for coercive, symmetric problems) of Sections 3 and 4.

2.2.1. *A thermal fin*

In this example we consider the two- and three-dimensional thermal fins shown in Figure 1; these examples may be (interactively) accessed on our web site[1]. The fins consist of a vertical central "post" of conductivity $\tilde{k}_0$ and four horizontal "subfins" of conductivity $\tilde{k}^i,\ i = 1,\dots,4$; the fins conduct heat from a prescribed uniform flux source, $\tilde{q}''$, at the root, $\tilde{\Gamma}_{\rm root}$, through the post and large-surface-area subfins to the surrounding flowing air; the latter is characterized by a sink temperature $\tilde{u}_0$, and prescribed heat transfer coefficient $\tilde{h}$. The physical model is simple conduction: the temperature field in the fin, $\tilde{u}$, satisfies

$$\sum_{i=0}^{4} \int_{\tilde{\Omega}_i} \tilde{k}^i\; \tilde{\nabla}\tilde{u} \cdot \tilde{\nabla}\tilde{v} \;+\; \int_{\partial\tilde{\Omega}\backslash\tilde{\Gamma}_{\rm root}} \tilde{h}\; (\tilde{u} - \tilde{u}_0)\; \tilde{v} \;=\; \int_{\tilde{\Gamma}_{\rm root}} \tilde{q}''\; \tilde{v}, \quad \forall\; \tilde{v} \in \tilde{X} \equiv H^1(\tilde{\Omega}), \tag{5}$$

where $\tilde{\Omega}_i$ is that part of the domain with conductivity $\tilde{k}^i$, and $\partial\tilde{\Omega}$ denotes the boundary of $\tilde{\Omega}$.

We now (*i*) nondimensionalize the weak equations (5), and (*ii*) apply a continuous piecewise-affine transformation to map $\tilde{\Omega}$ to a fixed reference domain Ω [10]. The abstract problem statement (3) is then recovered [22] for $\mu = \{k^1,\ k^2,\ k^3,\ k^4,\ {\rm Bi},\ L,\ t\}$, $\mathcal{D} = [0.1, 10.0]^4 \times [0.01, 1.0] \times [2.0, 3.0] \times [0.1 \times 0.5]$, and $P = 7$; here $k^1,\dots,k^4$ are the thermal conductivities of the "subfins" (see Fig. 1) relative to the thermal conductivity of the fin base; Bi is a nondimensional form of the heat transfer coefficient; and, L, t are the length and thickness of each of the "subfins" relative to the length of the fin root $\tilde{\Gamma}_{\rm root}$. It is readily verified that a is continuous, coercive, and symmetric; and that the "affine" assumption (2) obtains for $Q = 16$ (two-dimensional case) and $Q = 25$ (three-dimensional case). Note that the geometric variations are reflected, *via* the mapping, in the $\sigma^q(\mu)$.

[1]FIN3D: `http://augustine.mit.edu/fin3d_1/fin3d_1.pdf` and FIN2D: `http://augustine.mit.edu/fin2d/fin2d.pdf`

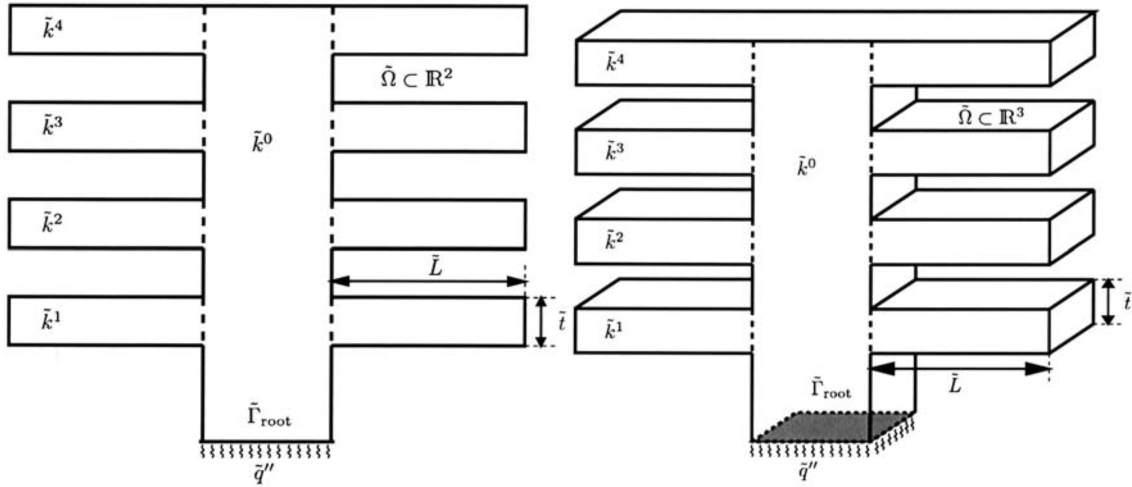

FIGURE 1. Two- and three-dimensional thermal fins.

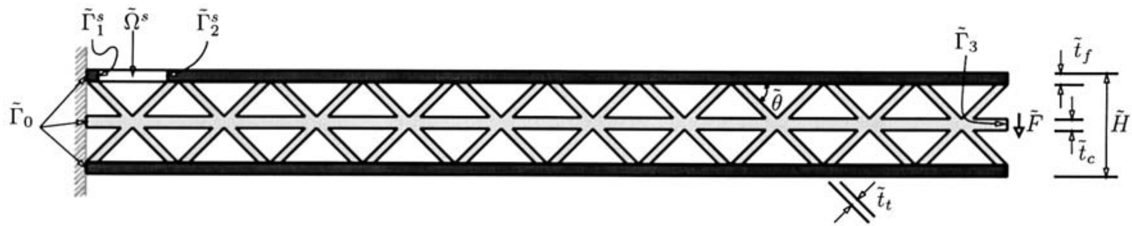

FIGURE 2. A truss structure.

For our output of interest, $s(\mu)$, we consider the average temperature of the root of the fin nondimensionalized relative to $\tilde{q}''$, $\tilde{k}^0$, and the length of the fin root. This output is calculated as $s(\mu) = \ell(u(\mu))$, where $\ell(v) = \int_{\Gamma_{\text{root}}} v$. It is readily shown that this output functional is bounded and also "compliant": $\ell(v) = f(v)$, $\forall v \in X$.

2.2.2. *A truss structure*

We consider a prismatic microtruss structure [6,24] shown in Figure 2; this example may be (interactively) accessed on our web site[2]. The truss consists of a frame (upper and lower faces, in dark gray) and a core (trusses and middle sheet, in light gray); the structure transmits a force per unit depth $\tilde{F}$ uniformly distributed over the tip of the middle sheet, $\tilde{\Gamma}_3$, through the truss system to the fixed left wall, $\tilde{\Gamma}_0$. The physical model is simple plane–strain (two-dimensional) linear elasticity: the displacement field u_i, $i = 1, 2$, satisfies

$$\int_{\tilde{\Omega}} \frac{\partial \tilde{v}_i}{\partial \tilde{x}_j} \tilde{E}_{ijkl} \frac{\partial \tilde{u}_k}{\partial \tilde{x}_l} = -\left(\frac{\tilde{F}}{\tilde{t}_c}\right) \int_{\tilde{\Gamma}_3} \tilde{v}_2, \quad \forall\, v \in \tilde{X}, \tag{6}$$

where $\tilde{\Omega}$ is the truss domain, and $\tilde{X}$ refers to the set of functions in $H^1(\tilde{\Omega})$ which vanish on $\tilde{\Gamma}_0$. We assume summation over repeated indices.

We now (i) nondimensionalize the weak equations (6), and (ii) apply a continuous piecewise-affine transformation to map $\tilde{\Omega}$ to a fixed reference domain Ω. The abstract problem statement (3) is then recovered [23] for $\mu = \{t_f,\ t_t,\ H,\ \theta\}, \mathcal{D} = [0.08, 1.0] \times [0.2, 2.0] \times [4.0, 10.0] \times [30.0°, 60.0°]$, and $P = 4$; here t_f and t_t are

[2]TRUSS: `http://augustine.mit.edu/virg_1/virg_1.pdf`

the thicknesses of the frame and trusses, respectively; H is the total height of the microtruss; and θ is the angle between the trusses and the faces. The Poisson's ratio, $\nu = 0.3$, and the frame and core Young's moduli, $E_f = 75$ GPa and $E_c = 200$ GPa, respectively, are held fixed. It is readily verified that a is continuous, coercive, and symmetric; and that the "affine" assumption (2) obtains for $Q = 44$; Q is larger than for the fin examples due to the more complex (sheared) affine geometry mappings.

Our outputs of interest are (*i*) the average downward deflection (compliance) at the core tip, Γ_3, nondimensionalized by $\tilde{F}/\tilde{E}_f$; and (*ii*) the average normal stress across the critical (yield) section denoted Γ_1^s in Figure 2. These compliance and noncompliance outputs are written as $s^1(\mu) = \ell^1(u(\mu))$ and $s^2(\mu) = \ell^2(u(\mu))$, respectively, where $\ell^1(v) = -\int_{\Gamma_3} v_2$, and

$$\ell^2(v) = \frac{1}{t_f}\int_{\Omega^s} \frac{\partial \chi_i}{\partial x_j} E_{ijkl} \frac{\partial u_k}{\partial x_l}$$

are bounded linear functionals; here χ_i is any suitably smooth function in $H^1(\Omega^s)$ such that $\chi_i \hat{n}_i = 1$ on Γ_1^s and $\chi_i \hat{n}_i = 0$ on Γ_2^s, where $\hat{n}$ is the unit normal.

3. Reduced-basis approach

We recall that in this section, as well as in Section 4, we assume that a is continuous, coercive, symmetric, and affine in μ — see (2); and that $\ell(v) = f(v)$, which we denote "compliance".

3.1. Reduced-basis approximation

We first introduce a sample in parameter space, $S_N = \{\mu_1, \dots, \mu_N\}$, where $\mu_i \in \mathcal{D}$, $i = 1, \dots, N$; see Section 3.2 for a brief discussion of point distribution. We then define our Lagrangian [17] reduced–basis approximation space as $W_N = \text{span}\,\{\zeta_n \equiv u(\mu_n),\ n = 1, \dots, N\}$, where $u(\mu_n) \in X$ is the solution to (3) for $\mu = \mu_n$. In actual practice, $u(\mu_n)$ is replaced by a finite element approximation on a suitably fine truth mesh; we shall discuss the associated computational implications in Section 3.3. Our reduced–basis approximation is then: for any $\mu \in \mathcal{D}$, find $u_N(\mu) \in W_N$ such that

$$a(u_N(\mu), v; \mu) \;=\; \ell(v), \quad \forall\, v \in W_N; \tag{7}$$

we then evaluate $s_N(\mu) = \ell(u_N(\mu))$. (Non-Galerkin projections are briefly described in [19].)

3.2. *A priori* convergence theory

3.2.1. *Optimality*

We consider here the convergence rate of $u_N(\mu) \to u(\mu)$ and $s_N(\mu) \to s(\mu)$ as $N \to \infty$. To begin, it is standard to demonstrate optimality of $u_N(\mu)$ in the sense that

$$\|u(\mu) - u_N(\mu)\|_X \le \sqrt{\frac{\gamma(\mu)}{\alpha(\mu)}} \inf_{w_N \in W_N} \|u(\mu) - w_N\|_X. \tag{8}$$

We note that, in the coercive case, stability of our "conforming" discrete approximation is not an issue; the noncoercive case is decidedly more delicate (see [19]). Furthermore, for our compliance output,

$$s(\mu) = s_N(\mu) + \ell(u - u_N) = s_N(\mu) + a(u, u - u_N; \mu) = s_N(\mu) + a(u - u_N, u - u_N; \mu) \tag{9}$$

from symmetry and Galerkin orthogonality. It follows that $s(\mu) - s_N(\mu)$ converges as the square of the error in the best approximation and, from coercivity, that $s_N(\mu)$ is a lower bound for $s(\mu)$.

3.2.2. *Best approximation*

It now remains to bound the dependence of the error in the best approximation as a function of N. At present, the theory is restricted to the case in which $P = 1$, $\mathcal{D} = [0, \mu_{\max}]$, and

$$a(w, v; \mu) = a_0(w, v) + \mu a_1(w, v), \tag{10}$$

where a_0 is continuous, coercive, and symmetric, and a_1 is continuous, positive semi-definite ($a_1(w, w) \geq 0$, $\forall w \in X$), and symmetric. This model problem (10) is rather broadly relevant, for example to variable orthotropic conductivity, rectilinear geometry variations, piecewise-constant conductivity variations, and variable Robin boundary conditions.

We now suppose that the μ_n, $n = 1, \ldots, N$, are logarithmically distributed in the sense that

$$\ln\left(\mu_n + \overline{\lambda}^{-1}\right) = \ln \overline{\lambda}^{-1} + \frac{n-1}{N-1} \ln\left(\frac{\mu_{\max} + \overline{\lambda}^{-1}}{\overline{\lambda}^{-1}}\right), \quad n = 1, \ldots, N, \tag{11}$$

where $\overline{\lambda}$ is the maximum eigenvalue of a_0 relative to a_1. (Note $\overline{\lambda}$ is perforce bounded thanks to our assumption of continuity and coercivity; the possibility of a continuous spectrum does not, in practice, pose any problems.) We can then prove [12] that, for $N > N_{\text{crit}} \equiv 2e \ln(\overline{\lambda}\, \mu_{\max} + 1)$,

$$\inf_{w_N \in W_N} \|u(\mu) - w_N(\mu)\|_X \leq \sqrt{\frac{\gamma}{\alpha}}\, \|u(0)\|_X \, \exp\left\{\frac{-N}{2e\, \ln(\overline{\lambda}\, \mu_{\max} + 1)}\right\}, \quad \forall\, \mu \in \mathcal{D}. \tag{12}$$

We observe exponential convergence, uniformly (globally) for all μ in $\mathcal{D}$, with only very weak (logarithmic) dependence on the range of the parameter ($\mu_{\max}$).

The proof exploits the (parameter–space) interpolant as a surrogate for the Galerkin approximation. As a result, the bound is not "sharp": we observe many cases in which the Galerkin projection is considerably better than the associated interpolant; optimality (8) chooses to "illuminate" only certain points μ_n, automatically selecting a best "sub–approximation" amongst all possibilities — we thus see why reduced–basis *state-space* approximation of $s(\mu)$ *via* $u(\mu)$ is preferred to simple *parameter-space* interpolation of $s(\mu)$ ("connecting the dots") *via* $(\mu_n, s(\mu_n))$ pairs. Nevertheless, the logarithmic point distribution (11) implicated by our interpolant–based arguments is *not* simply an artifact of the proof: in numerous numerical tests, the logarithmic distribution performs considerably better than other obvious candidates, in particular for large ranges of the parameter. Fortunately, the convergence rate is not *too* sensitive to point selection: the theory only requires a log "on the average" distribution [12]; and, in practice, $\overline{\lambda}$ in (12) may be replaced with any "reasonable" value.

The result (12) is certainly tied to the particular form (10) and associated regularity of $u(\mu)$. However, we do observe similar exponental behavior for more general operators; and, most importantly, the exponential convergence rate degrades only very slowly with increasing parameter dimension, P. We present in Table 1 the error $|s(\mu) - s_N(\mu)|/s(\mu)$ as a function of N, at a particular representative point μ in $\mathcal{D}$, for the two-dimensional thermal fin problem of Section 2.2.1; we present similar data in Table 2 for the truss problem of Section 2.2.2. In both cases, since tensor-product grids are prohibitively profligate as P increases, the μ_n are chosen "log-randomly" over $\mathcal{D}$: we sample from a multivariate uniform probability density on $\log(\mu)$. We observe that for both the thermal fin ($P = 7$) and truss ($P = 4$), the error is remarkably small even for very small N; and, in both cases, very rapid convergence obtains as $N \to \infty$. We do not yet have any theory for $P > 1$. But certainly the Galerkin optimality plays a central role, automatically selecting "appropriate" scattered-data subsets of S_N and associated "good" weights so as to mitigate the curse of dimensionality as P increases; and the log–random point distribution is also important — for example, for the truss problem of Table 2, a (non–log) uniform random point distribution yields errors which are larger by factors of 20 and 10 for $N = 30$ and 80, respectively.

TABLE 1. Error, error bound, and effectivity as a function of N, at a particular representative point $\mu \in \mathcal{D}$, for the two-dimensional thermal fin problem (compliant output).

N	$\lvert s(\mu) - s_N(\mu)\rvert / s(\mu)$	$\Delta_N(\mu)/s(\mu)$	$\eta_N(\mu)$
10	1.29×10^{-2}	8.60×10^{-2}	2.85
20	1.29×10^{-3}	9.36×10^{-3}	2.76
30	5.37×10^{-4}	4.25×10^{-3}	2.68
40	8.00×10^{-5}	5.30×10^{-4}	2.86
50	3.97×10^{-5}	2.97×10^{-4}	2.72
60	1.34×10^{-5}	1.27×10^{-4}	2.54
70	8.10×10^{-6}	7.72×10^{-5}	2.53
80	2.56×10^{-6}	2.24×10^{-5}	2.59

TABLE 2. Error, error bound, and effectivity as a function of N, at a particular representative point $\mu \in \mathcal{D}$, for the truss problem (compliant output).

N	$\lvert s(\mu) - s_N(\mu)\rvert / s(\mu)$	$\Delta_N(\mu)/s(\mu)$	$\eta_N(\mu)$
10	3.26×10^{-2}	6.47×10^{-2}	1.98
20	2.56×10^{-4}	4.74×10^{-4}	1.85
30	7.31×10^{-5}	1.38×10^{-4}	1.89
40	1.91×10^{-5}	3.59×10^{-5}	1.88
50	1.09×10^{-5}	2.08×10^{-5}	1.90
60	4.10×10^{-6}	8.19×10^{-6}	2.00
70	2.61×10^{-6}	5.22×10^{-6}	2.00
80	1.19×10^{-6}	2.39×10^{-6}	2.00

3.3. **Computational procedure**

The theoretical and empirical results of Sections 3.1 and 3.2 suggest that N may, indeed, be chosen very small. We now develop off–line/on–line computational procedures that exploit this dimension reduction.

We first express $u_N(\mu)$ as

$$u_N(\mu) = \sum_{j=1}^{N} u_{N\,j}(\mu)\, \zeta_j = (\underline{u}_N(\mu))^T\, \underline{\zeta}, \tag{13}$$

where $\underline{u}_N(\mu) \in \mathbb{R}^N$; we then choose for test functions $v = \zeta_i$, $i = 1, \ldots, N$. Inserting these representations into (7) yields the desired algebraic equations for $\underline{u}_N(\mu) \in \mathbb{R}^N$,

$$\underline{A}_N(\mu)\, \underline{u}_N(\mu) \;=\; \underline{F}_N \tag{14}$$

in terms of which the output can then be evaluated as $s_N(\mu) = \underline{F}_N^T\, \underline{u}_N(\mu)$. Here $\underline{A}_N(\mu) \in \mathbb{R}^{N\times N}$ is the SPD matrix with entries $A_{N\,i,j}(\mu) \equiv a(\zeta_j, \zeta_i; \mu)$, $1 \le i, j \le N$, and $\underline{F}_N \in \mathbb{R}^N$ is the "load" (and "output") vector with entries $F_{N\,i} \equiv f(\zeta_i)$, $i = 1, \ldots, N$.

We now invoke (2) to write

$$A_{N\,i,j}(\mu) = a(\zeta_j, \zeta_i; \mu) = \sum_{q=1}^{Q} \sigma^q(\mu)\, a^q(\zeta_j, \zeta_i), \tag{15}$$

or

$$\underline{A}_N(\mu) = \sum_{q=1}^{Q} \sigma^q(\mu)\, \underline{A}_N^q,$$

where $A^q_{N\,i,j} = a^q(\zeta_j, \zeta_i)$, $i \le i, j \le N$, $1 \le q \le Q$. The off–line/on–line decomposition is now clear:

In the *off–line* stage, we compute the $u(\mu_n)$ and form the $\underline{A}_N^q$ and $\underline{F}_N$: this requires N (expensive) "a" finite element solutions and $O(QN^2)$ finite-element-vector inner products.

In the *on–line* stage, for any given new μ, we first form $\underline{A}_N$ from (15), then solve (14) for $\underline{u}_N(\mu)$, and finally evaluate $s_N(\mu) = \underline{F}_N^T\, \underline{u}_N(\mu)$: this requires $O(QN^2) + O(\frac{2}{3}N^3)$ operations and $O(QN^2)$ storage.

Thus, as required, the incremental, or marginal, cost to evaluate $s_N(\mu)$ for any given new μ — as proposed in a design, optimization, or inverse-problem context — is very small: first, because N is very small, typically $O(10)$ — thanks to the good convergence properties of W_N; and second, because (14) can be very rapidly assembled and inverted — thanks to the off–line/on–line decomposition. For the problems discussed in this paper, the resulting computational savings relative to standard (well-designed) finite-element approaches are significant — at least $O(10)$, typically $O(100)$, and often $O(1000)$ or more.

4. *A posteriori* error estimation: Output bounds

From Section 3 we know that, in theory, we can obtain $s_N(\mu)$ very inexpensively: the on–line stage scales as $O(N^3) + O(QN^2)$; and N can, *in theory*, be chosen quite small. However, *in practice*, we do not know *how* small N can be chosen: this will depend on the desired accuracy, the selected output(s) of interest, and the particular problem in question; in some cases $N = 5$ may suffice, while in other cases, $N = 100$ may still be insufficient. In the face of this uncertainty, either too many or too few basis functions will be retained: the former results in computational inefficiency; the latter in unacceptable uncertainty — particularly egregious in the decision contexts in which reduced–basis methods typically serve. We thus need *a posteriori* error estimators for s_N. Surprisingly, *a posteriori* error estimation has received relatively little attention within the reduced–basis framework [13], even though reduced–basis methods are particularly in need of accuracy assessment: the spaces are *ad hoc* and pre-asymptotic, admitting relatively little intuition, "rules of thumb," or standard approximation notions.

Recall that, in the section, we continue to assume that a is coercive and symmetric, and ℓ is "compliant".

4.1. **Method I**

The approach described in this section is a particular instance of a general "variational" framework for *a posteriori* error estimation of outputs of interest. However, the reduced-basis instantiation described here differs significantly from earlier applications to finite element discretization error [9, 11] and iterative solution error [14, 15] both in the choice of (energy) relaxation and in the associated computational artifice.

4.1.1. *Formulation*

We assume that we are given a function $g(\mu) : \mathcal{D} \to \mathbb{R}_+$, and a continuous, coercive, symmetric (μ-independent) bilinear form $\hat{a} : X \times X \to \mathbb{R}$, such that

$$\underline{\alpha}_0 \|v\|_X^2 \;\le\; g(\mu)\; \hat{a}(v,v) \;\le\; a(v,v;\mu), \quad \forall\, v \in X, \forall\, \mu \in \mathcal{D}. \tag{16}$$

We then find $\hat{e}(\mu) \in X$ such that

$$g(\mu)\ \hat{a}(\hat{e}(\mu), v) \ = \ R(v; u_N(\mu); \mu), \quad \forall v \in X \tag{17}$$

where for a given $w \in X$, $R(v; w; \mu) = \ell(v) - a(w, v; \mu)$ is the weak form of the residual. Our lower and upper output estimators are then evaluated as

$$s_N^-(\mu) \ \equiv \ s_N(\mu), \text{ and } s_N^+(\mu) \equiv s_N(\mu) + \Delta_N(\mu), \tag{18}$$

respectively, where

$$\Delta_N(\mu) \ \equiv \ g(\mu)\ \hat{a}(\hat{e}(\mu), \hat{e}(\mu)) \tag{19}$$

is the estimator gap.

4.1.2. *Computational procedure*

Finally, we turn to the computational artifice by which we can efficiently compute $\Delta_N(\mu)$ in the on–line stage of our procedure. To begin, we rewrite the "modified" error equation, (17), as

$$\hat{a}(\hat{e}(\mu), v) \ = \ \frac{1}{g(\mu)} \left(\ell(v) - \sum_{q=1}^{Q} \sum_{j=1}^{N} \sigma^q(\mu) u_{N\,j}(\mu) a^q(\zeta_j, v) \right), \quad \forall\ v \in X$$

where we have appealed to our reduced–basis approximation (13) and the affine decomposition (2). It is immediately clear from linear superposition that we can express $\hat{e}(\mu)$ as

$$\hat{e}(\mu) = \frac{1}{g(\mu)} \left(\hat{z}_0 + \sum_{q=1}^{Q} \sum_{j=1}^{N} \sigma^q(\mu) u_{N\,j}(\mu) \hat{z}_j^q \right); \tag{20}$$

where $\hat{z}_0 \in X$ satisfies $\hat{a}(\hat{z}_0, v) = \ell(v)$, $\forall\ v \in X$, and $\hat{z}_j^q \in X, j = 1, \ldots, N$, $q = 1, \ldots, Q$, satisfies $\hat{a}(\hat{z}_j^q, v) = -a^q(\zeta_j, v)$, $\forall\ v \in X$. Inserting (20) into our expression for the upper bound, $s_N^+(\mu) = s_N(\mu) + g(\mu)\hat{a}(\hat{e}(\mu), \hat{e}(\mu))$, we obtain

$$s_N^+(\mu) = s_N(\mu) + \frac{1}{g(\mu)} \left(c_0 + 2 \sum_{q=1}^{Q} \sum_{j=1}^{N} \sigma^q(\mu) u_{N\,j}(\mu) \Lambda_j^q \ + \ \sum_{q=1}^{Q} \sum_{q'=1}^{Q} \sum_{j=1}^{N} \sum_{j'=1}^{N} \sigma^q(\mu) \sigma^{q'}(\mu) u_{N\,j}(\mu) u_{N\,j'}(\mu) \Gamma_{jj'}^{qq'} \right) \tag{21}$$

where $c_0 = \hat{a}(\hat{z}_0, \hat{z}_0)$, $\Lambda_j^q = \hat{a}(\hat{z}_0, \hat{z}_j^q)$, and $\Gamma_{jj'}^{qq'} = \hat{a}(\hat{z}_j^q, \hat{z}_{j'}^{q'})$. The off–line/on–line decomposition should now be clear.

In the *off–line* stage we compute $\hat{z}_0$ and $\hat{z}_j^q$, $j = 1, \ldots, N$, $q = 1, \ldots, Q$, and then form c_0, Λ_j^q, and $\Gamma_{jj'}^{qq'}$: this requires $QN + 1$ (expensive) "$\hat{a}$" finite element solutions, and $O(Q^2N^2)$ finite-element-vector inner products.

In the *on–line* stage, for any given new μ, we evaluate s_N^+ as expressed in (21): this requires $O(Q^2N^2)$ operations; and $O(Q^2N^2)$ storage (for c_0, Λ_j^q, and $\Gamma_{jj'}^{qq'}$).

As for the computation of $s_N(\mu)$, the marginal cost for the computation of $s_N^{\pm}(\mu)$ for any given new μ is quite small — in particular, independent of the dimension of the truth finite element approximation space X.

There are a variety of ways in which the off–line/on–line decomposition and output error bounds can be exploited. A particularly attractive mode incorporates the error bounds into an on–line adaptive process, in which we successively approximate $s_N(\mu)$ on a sequence of approximation spaces $W_{N'_j} \subset W_N, N'_j = N_0 2^j$ —

for example, $W_{N'_j}$ may contain the N'_j sample points of S_N closest to the new μ of interest — until $\Delta_{N'_j}$ is less than a specified error tolerance. This procedure both minimizes the on–line computational effort and reduces conditioning problems — while simultaneously ensuring accuracy and certainty.

4.2. **Method II**

As already indicated, Method I has certain limitations; we discuss here a Method II which addresses these limitations — albeit at the loss of complete certainty.

4.2.1. *Formulation*

To begin, we set $M > N$, and introduce a parameter sample $S_M = \{\mu_1, \ldots, \mu_M\}$ and associated reduced–basis approximation space $W_M = \text{span}\{\zeta_m \equiv u(\mu_m),\ m = 1, \ldots, M\}$; both for theoretical and practical reasons we require $S_N \subset S_M$ and therefore $W_N \subset W_M$. The procedure is very simple: we first find $u_M(\mu) \in W_M$ such that $a(u_M(\mu), v; \mu) = f(v), \forall\, v \in W_M$; we then evaluate $s_M(\mu) = \ell(u_M(\mu))$; and, finally, we compute our upper and lower output estimators as

$$s^-_{N,M}(\mu) = s_N(\mu), \qquad s^+_{N,M}(\mu) = s_N(\mu) + \Delta_{N,M}(\mu), \tag{22}$$

where $\Delta_{N,M}(\mu)$, the estimator bound gap, is given by

$$\Delta_{N,M}(\mu) = \frac{1}{\tau}\left(s_M(\mu) - s_N(\mu)\right) \tag{23}$$

for some $\tau \in (0, 1)$. The effectivity of the approximation is defined as

$$\eta_{N,M}(\mu) = \frac{\Delta_{N,M}(\mu)}{s(\mu) - s_N(\mu)}\,\cdot \tag{24}$$

For our purposes here, we shall consider $M = 2N$.

4.2.2. *Computational procedure*

Since the error bounds are based entirely on evaluation of the output, we can directly adapt the off–line/on–line procedure of Section 3.3. Note that the calculation of the output approximation $s_N(\mu)$ and the output bounds are now integrated: $\underline{A}_N(\mu)$ and $\underline{F}_N(\mu)$ (yielding $s_N(\mu)$) are a sub-matrix and sub-vector of $\underline{A}_{2N}(\mu)$ and $\underline{F}_{2N}(\mu)$ (yielding $s_{2N}(\mu)$, $\Delta_{N,2N}(\mu)$ and $s^{\pm}_{N,2N}(\mu)$) respectively.

In the *off–line* stage, we compute the $u(\mu_n)$ and form the $\underline{A}^q_{2N}$ and $\underline{F}_{2N}$: this requires $2N$ (expensive) "a" finite element solutions, and $O(4QN^2)$ finite-element-vector inner products.

In the *on–line* stage, for any given new μ, we first form $\underline{A}_N(\mu)$ and $\underline{A}_{2N}(\mu)$ then solve for $\underline{u}_N(\mu)$ and $\underline{u}_{2N}(\mu)$, and finally evaluate $s^{\pm}_{N,2N}(\mu)$: this requires $O(4QN^2) + O(\frac{16}{3}N^3)$ operations and $O(4QN^2)$ storage.

The on–line effort for this Method II predictor/error estimator procedure (based on $s_N(\mu)$ and $s_{2N}(\mu)$) will require eightfold more operations than the predictor procedure of Section 4.1.

Method II is in some sense very naive: we simply replace the true output $s(\mu)$ with a finer–approximation surrogate $s_{2N}(\mu)$. (There are more obscure ways to describe the method — in terms of a reduced–basis approximation for the error — however there is little to be gained from these alternative interpretations.) The essential computation enabler is again exponential convergence, which permits us to choose $M = 2N$ — hence controlling the additional computational effort attributable to error estimation — while simultaneously ensuring that $\varepsilon_{N,2N}(\mu)$ tends rapidly to zero. Exponential convergence also ensures that the cost to compute both $s_N(\mu)$ and $s_{2N}(\mu)$ is "negligible". In actual practice, since $s_{2N}(\mu)$ is available, we can of course take $s_{2N}(\mu)$, rather than $s_N(\mu)$, as our output prediction; this greatly improves not only accuracy, but also certainty — $\Delta_{N,2N}(\mu)$ is almost surely a bound for $s(\mu) - s_{2N}(\mu)$, albeit an exponentially conservative bound as N tends to infinity.

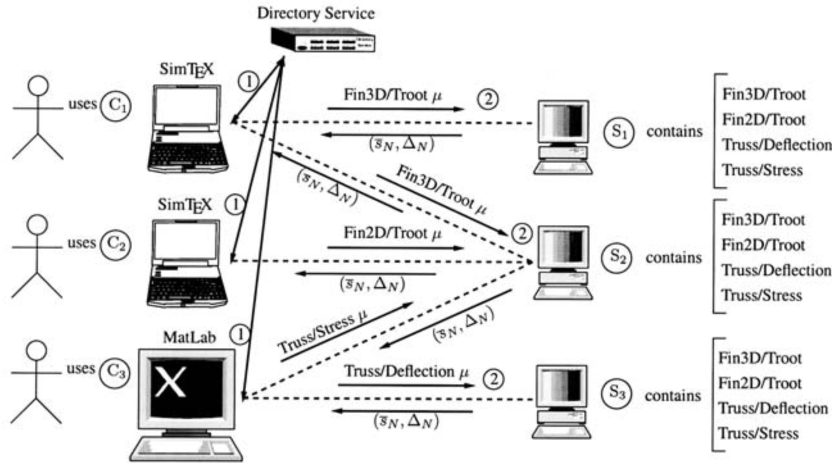

FIGURE 3. A sample use case of the framework.

5. System architecture

5.1. Introduction

The numerical methods proposed are rather unique relative to more standard approaches to partial differential equations. Reduced–basis output bound methods — in particular the global approximation spaces, *a posteriori* error estimators, and off–line/on–line computational decomposition — are intended to render partial–differential-equation solutions truly "useful": essentially real–time as regards operation count; "blackbox" as regards reliability; and directly relevant as regards the (limited) input–output data required. But to be truly useful, the methodology — in particular the inventory of on–line codes — must reside within a special framework. This framework must permit a User to specify — within a native applications context — the problem, output, and input value of interest; and to receive — quasi–instantaneously — the desired prediction and certificate of fidelity (error bound). We describe such a (fully implemented, fully functional) framework here: we focus primarily on the User point of view; see [18] for a more detailed description of the technical foundations and ingredients.

5.2. Overview of framework

We show in Figure 3 a virtual schematic of the framework. The key components are the User, Computers, Network, Client software, Server software, and Directory Service. Each User interacts with the system through a selected Client (interface) which resides, say, on the User's Computer; we shall describe briefly below two Clients. Based on directives from the User, the Client broadcasts over the Network a Problem Label (*e.g.* Fin3D), Output Label (*e.g.* Troot) Pair. This Pair is received by the Directory Service — a White Pages — which informs the Client of the Simulation Resource Locator "SRL" — physical location on a particular Computer — of a Server which can respond to the request. The Client then sends the Input (μ P–tuple Value) to the designated SRL. The Server — essentially a suite of on–line codes and associated input–output utilities — is awaiting queries at all times; upon receipt of the Input it executes the on–line code for the designated Output Label and Input Value, and responds to the Client with the Output Value ($\bar{s}_N$) and Error Bound Gap (Δ_N). The Client then displays or acts upon this information, and the cycle is complete.

Typically many identical (as well as different) Servers will be available, typically on many different Computers: there are multiple instances of the on–line codes. The Directory Service indicates to the Client the least busy Server so as to provide the fastest response possible. In some cases Clients may issue Input Value which are in fact a vector of input values — that is, L P–tuples. In this case the Directory Service will distribute

the calculations over multiple (*e.g.* as many as L) Servers — in particular Servers on multiple Computers — so as respond more quickly to this multiple–input query. Our framework is clearly an example of "grid" computing, similar to GLOBUS, NetSolve, and Seti@HOME, to name but a few. Indeed, we exploit several generic tools upon which grid and network computing applications may be built; for example, we appeal to CORBA[3] (standardized by OMG[4]) to seamlessly manipulate the Server software *as if* it resided on the Client Computer. We remark that our reduced–basis output bound application is particularly well–suited to grid computing: the computational load on participating Computers (on which the Servers reside) is very light; and the Client–Server input/output load on the Network is very light. The network computing paradigm also serves very well the archival, collaboration, and integration aspects of standardized input–output objects.

5.3. **Clients**

We describe here two Clients: SimTeX, which is a PDF–based "dynamic text" interface for interrogation, exploration, and display; and SimLab, which is a MATLAB–based "mathematical" interface for manipulation and integration. A third client has been developed: WebLab; however it is a direct application of the MatLab client using the MatLab Web Server Toolbox.

5.3.1. SimTeX

SimTeX combines several standardized tools so as to provide a very simple interface by which to access the Servers. A particularly nice feature of SimTeX is the natural context which it provides — in essence, defining the input–output relationship and problem definition in the language of the application. The SimTeX Client should prove useful in a number of different contexts: textbooks and technical manuscripts; handbooks; and product specification and design sheets.

The SimTeX Client consists of an authoring component, a display and interface component, and an "intermediary" component. The authoring component is — a standard in scientific typesetting — enhanced (*via* `hyperref`) with a new `acteq` environment which permits the inclusion of actionable equations. The `acteq` environment links an equation to a Problem Label, Output Label(s) and Input Value template. The output is a PDF document: the PDF document serves as the display, graphics, and (rudimentary) interface component of SimTeX. The PDF document contains a form which accepts the Input Values, and an "equal sign" button which initiates the Client–Directory Services Client–Server dialogue described in the previous document. Upon completion of the cycle, the PDF document is updated to display the values of the output and error bound for the Input Values submitted; in cases in which multiple input values or outputs are selected, appropriate graphics are presented using the Figure button. Finally, since PDF is not a programming language, and Client–Server intermediary is required: a CGI script serves to parse the PDF form, communicate with the Server, and finally update the Client.

As an example, we include here an actionable equation — the actual SimTeX user interface — for the several outputs (the root temperature, tip temperature, volume) associated with the three–dimensional thermal fin example:

$$\begin{pmatrix} T_{\text{root}} \; \square \\ T_{\text{tip}} \; \square \\ \text{Volume} \; \square \end{pmatrix} = \mathcal{F} \begin{pmatrix} k^1 = & 0.4 \\ k^2 = & 0.6 \\ k^3 = & 0.8 \\ k^4 = & 1.2 \\ \text{Bi} = & 0.1 \\ t = & 0.3 \\ L = & 2.8 \end{pmatrix} \equiv \qquad \pm \qquad$$

[3]Common Object Request Broker Architecture — `http://www.corba.org`
[4]Object Management Group — `http://www.omg.org`

The input list corresponds to the μ vector described in Section 2.2.1; the input values must lie in the parameter domain $\mathcal{D}$ described in Section 2.2.1. The notation Output = $\mathcal{F}$(Input) is a description of the input–output relationship $s(\mu)$ implied by $s = \ell(u(\mu))$. The actionable PDF version of this entire paper (in which is embedded the actionable equation) may be found on our web site[5]; readers are encouraged to access this electronic version of the paper and exercise the SimTEX interface, a brief users manual for which may be found again on our web site[6].

5.3.2. SimLaB

The main drawback of SimTEX is the inability to manipulate the on–line codes. SimLaB is a suite of tools that permit Users to incorporate Server on–line codes as MATLAB functions within the standard MATLAB interface; and to generate new Servers and on–line codes from standard MATLAB functions (which themselves may be built upon other on–line codes). In short, SimLaB permits the User to treat the inputs and outputs of our on–line codes as mathematical objects that are the result of, or an argument to, other functions — graphics, system design, or optimization — and to archive these higher level operations in new Server objects available to all Clients once registered in the Directory Service.

For example, to incorporate the Fin3D input–output relationship into MATLAB, we first generate the needed MATLAB functions using a MATLAB script called `st2m`. This script, by default, generates automatically a MATLAB function for each Output registered in the Directory Service. It is also possible to ask for a specific Problem and Output using the following command in MATLAB: `st2m --model fin3d --output Troot` — however it implies that one knows the name of the Problem and Output. Then, to set the values for the seven components of the parameter vector, we enter

```
p.values(1).value=0.8;
p.values(1).name='k1';
p.values(2).value=2;
p.values(2).name='k2';
p.values(3).value=14;
p.values(3).name='k3';
p.values(4).value=3;
p.values(4).name='k4';
p.values(5).value=0.2;
p.values(5).name='Bi';
p.values(6).value=0.1;
p.values(6).name='t';
p.values(7).value=2.5;
p.values(7).name='L';
```

within the MATLAB command window. To determine the output value and bound gap for this value of the 7–tuple parameter, we then enter

```
[Troot, Bound_Troot] = fin3d_Troot( p )
```

which returns

```
Troot = 1.06869419906058
Bound_Troot = 1.06869419906058
```

It is also now possible of course to find all values of `Troot` greater than 1.05 for t in the range $[0.1, 0.5]$ and all other parameters fixed as in the list above. To wit, we enter

```
i=1;
while( i < 1000 )
  p.values(6).value=0.1+i*(0.4)/1000;
```

[5] http://augustine.mit.edu/jfe/jfe.pdf
[6] http://augustine.mit.edu/guided_tour.pdf

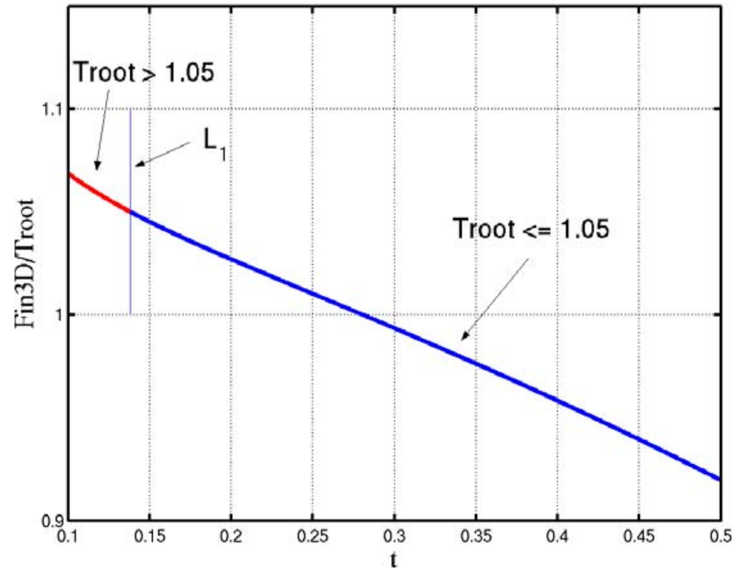

FIGURE 4. Plot.

```
  t(i)=p.values(6).value;
  [o(i),e(i)] = fin3d_Troot( p );
  i=i+1;
end
plot(x(o<1.05),o(o<1.05),'b'); hold on; grid on;
plot(t(o>=1.05),o(o>=1.05),'r');
line([max(t(o>1.05)) max(t(o>1.05))], [1  1.1]); %% L1
```

which generates Figure 4. Where the line L_1 splits the domain $[0.1; 0.5]$ at $t_{\max} = \max(t(o > 1.05)) = 0.1380$ between the values of `Troot` greater than 1.05 ($t < t_{\max}$) and the values of `Troot` less then 1.05 ($t \geq t_{\max}$). To be more certain that `Troot` was *truly* greater than 1.05, we could easily ask for those values of t for which $\overline{s}_N - \Delta_N \geq 1.05$; again readily effected by a simple function call. Obviously, once the on-line code is within the MATLAB environment, we have the full functionality of MATLAB at our disposal; and the rapid response of the reduced–basis output bounds maintains the immediate response expected of an interactive environment, even though we are in fact solving — and solving reliably — three-dimensional partial differential equations.

5.4. **Overview of the framework**

5.4.1. *Introduction*

The main current trend in computing and scientific computing is Distributed Objects — see for example the .NET, Mono, Globus, Netsolve, Seti@Home technologies and as mentioned earlier, we use CORBA as the underlying technology for our Framework.

CORBA is the Distributed Objects technology developed and standardized by the OMG. As envisioned by CORBA, Distributed Objects are the melding of concepts from two paradigms, Client-Server (or more precisely Distributed Computing) and Object Orientation (OO) with some slight differences: (*i*) a Client knows an object by its interface; (*ii*) objects are not always local with respect to their Clients; (*iii*) dynamic composition may compose objects into new application; (*iv*) objects hide many of the underlying differences (between Client and Server) in architecture through encapsulation.

The combination of the Client-Server and OO models gives us the best features: ability to distribute risk (fault tolerance), rightsizing system development with small composable subtasks and having looser coupling

thanks to well-defined integration and interfaces. Another advantage is also that we can have much more complicated topologies — see, for example, Figure 3 — than typically found in the Client-Server paradigm: a Client request computation from a Server which is itself a composition of several other Servers; in this context, our requested object is a Client-Server — a Server for the Client and a Client for the Servers composing it.

It is interesting to note that although there is generally a many-to-one relation for Client to Server, Clients may want to have access to more than one Server for a given purpose. In the overview of our Framework we have seen that it was effectively the case (see Fig. 3). Indeed, if a single source can be beneficial, it can also be expensive: risk of central outage (single point failure), too little specialization (resource utilization is suboptimal) long queues for services and large distances over which products chip and so on.

Development costs may rise also: distribution introduces more difficult problems such as, for example, the logistics of coordinating multiple sites. Two other particularly serious issues are the *network latency* and the *scalability*. They are difficult to determine beforehand and they can undermine gravely the deployment of the Framework.

Those issues are partly addressed by the numerical methods proposed (see Sect. 5.1). Only partly because the *network latency* is a difficult issue and reducing it is by no means easy — see the Akamai technology[7]. And regarding the *scalability*, the numerical methods proposed are not sufficient — although lots of problems (scheduling, monitoring, ...) arising when using more conventional methods are of no concern in the Reduced-Basis Output Bound methods context — therefore an adequate design for our Framework is also a requirement to ensure *scalability*.

The Framework relies on a library, St [8], which sits on top of CORBA (see Fig. 5) and its associated services. Three Clients — SimTeX, SimLaB, WebLaB— have been developed with St.

5.4.2. *The main actors of* St

The design of St shares similar concepts to the one that can be found in modern Graphical User Interface (GUI) libraries: the `SApplication` class and the `SSimget` class and their respective subclasses. Corba is a complex middle-ware specification and through simple coarse grain interfaces and high level concepts St encapsulates all Corba aspects — standard Corba calls or Corba Services — inside its classes. In the following section, we shall describe briefly some aspects of the Framework.

SApplication. As shown in Figure 5, St sits on top of Corba and the Corba Services: it encapsulates all Corba and associated components into a small set of classes with well defined behavior. Central to this design is the `St::SApplication` which encapsulates initialization of Corba, determines the available services and, in Server mode using `St::SApplicationServer` subclass, drives the execution flow of the application through its `St::SApplicationServer::run()` method which is basically an infinite loop waiting for new requests from Clients, see Section 5.5.1. A `SApplication`, and subclasses, follows the Singleton pattern to ensure that there exists only one instance of this class per process. It is possible to check for the availability on the Directory Service Server (see Fig. 3) of three standard Corba Services through the member functions `bool hasNamingService()`, `bool hasTradingService()` and `bool hasImplementationRepository()` and have access to each to these Services through there associated class, `SNamingService`, `STradingService` and `SImplementationRepository`.

In practice, accessing the Corba Services directly from a Client or a Server is neither needed nor recommended, it is usually taken care of by the objects that represent the Simulation objects, the Simgets.

An issue arising inevitably in Distributed Computing is Security. The most basic Security action that can be taken is related to the protection of the computers running the Servers: the processes associated to the Servers must have the lowest permissions on the system. On a Unix system for example, such a process would belong to the `nobody` user and group. By doing so, if someone with ill-intent manages to enter the operating system using those processes, he cannot do any harm. That is the least that has to be done. Unfortunately, ill-intended people can still do harm to the Framework. In order to avoid this, we use an authentication layer on top of the Framework using the Secure Sockets Layer (SSL). It has also the advantage to have some statistics on the

[7] `http://www.akamai.com`
[8] Simulation Toolkit.

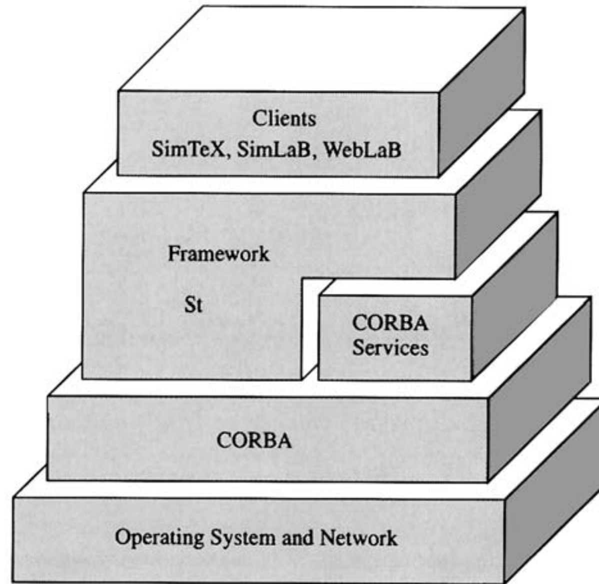

FIGURE 5. ST and CORBA.

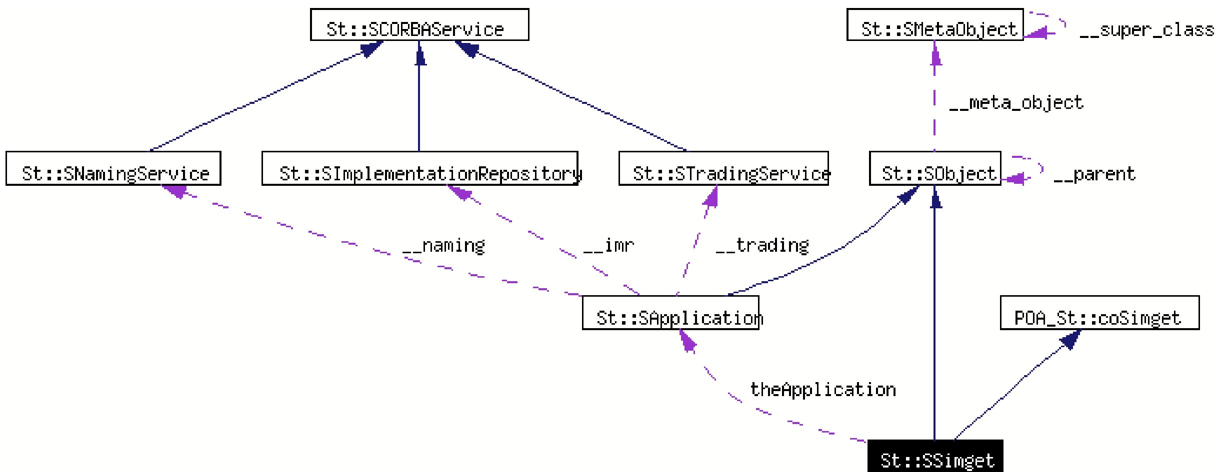

FIGURE 6. The Simget collaboration diagram.

Framework usage. The inclusion of the MICOSEC which is an implementation of the CORBA Security Services (CORBASec) Level 2 version 1.7[9] is underway. The current and future security features are/will be built-in `SApplication`.

SSimget. As described in the collaboration diagram (Fig. 6), a Simget is a SObject, base class for all ST objects, and also a CORBA object through the interface `POA_St::coSimget`. It contains a reference to the reference of the `SApplication` running in order to have access to the CORBA Services to be able to register itself automatically in each of them, provided that some of the Services are available. A Simget is a rather

[9]`http://www.micosec.org/`

complex object which is following the Composite pattern: it can be either a standalone computational object or a composite object of Simgets.

The constructor of the SSimget class and subclasses follows the prototype:

```
#include <SSimget.hpp>
SSimget( SSimget* parent, const char* name );

class A: public St::SSimget
{
  public:
  A( St::SSimget* parent, const char* name )
   : St::SSimget( parent, name )
 {}
};
```

where parent is the parent Simget and name is the name of the object passed to SObject — every SObject has a name. Then, using this composite or tree, it is possible to define a directory of Simgets in the various CORBA Services. The name of the Simget and its relationship with the other ones will be used to define its location in the Services. For example, if the following code is executed:

```
St::SApplicationServer *app = new St::SApplicationServer;
A * a = new A( 0, "a");
A * b = new A( a, "b");
A * c = new A( b, "c");
A * d = new A( a, "d");
app->setMainSimget( a );
```

then the Naming Service will contain the following references

```
console >  nsadmin -ORBNamingAddr inet:<directory service computer>:<port of the service>
> ls
a.a/
> ls a.a
b.b/
d.d/
> ls a.a/b.b
c.c/
```

nsadmin is a tool provided by MICO that allows to browse the Naming Service using the UNIX-like tools like ls or rm.

The SSimget is very general and is the base class for all Simgets. In the context of the Reduced-Basis methods and the Framework we built for them, we added other concepts which could be used to other kind of problems. First, we defined a *model* which is an object describing the model or problem being considered, for example Fin3D. The associated class is St::SModel. Then, we defined a model component which represents a general concept of model part. The associated class is SModelComponent. Finally, various subclasses of SModelComponent arise like SModelOutput, SModelOutputSet, and SModelField which are associated with the computation respectively of one output and associated error estimation if available, of a set of different outputs and associated error estimations, or a field — say a Finite Element field associated with a Finite Element simulation. Each of these objects are CORBA objects and follow a relatively simple IDL[10] interface that allows their remote manipulation from a Client program. Here is the IDL interface for a SModelComponent

[10]Interface Definition Language.

```
interface coModelComponent:  coSimget
{
  //! get the name of the Model
  string getModelName();

  //! set the new parameter set
  void setParameterSet( in coParameterSeq  pset );

  //! set a new Range
  void setRange( in coRangeSeq range );
};
```

and `SModelComponent` is defined as a subclass of `SSimget` which implements the interface `coModelComponent`. The other classes, `SModelOutput`, `SModelOutput` and `SModelField`, have similar simple CORBA interfaces with further specification. For example a `SModelComponent` and subclasses have the following constructor

```
SModelComponent( SModel* parent, const char* name );
```

which means that a model component has to have a model as a parent Simget. The relationship model/component is enforced through this explicit constructor. For an implementation example see Section 5.5.1.

SMonitor. While monitoring is not an important issue for our Framework since we deal with black-box real-time response simulations, it is interesting however to collect statistics or to provide such a tool for future Simgets that will need monitoring. Monitors are Clients and Servers for the Framework, they are implemented using the Observer and Memento design pattern. The base class for monitors is `SMonitor` which encapsulates the commonalities for all monitors which are here the interfaces following the above-mentioned design patterns.

The abstraction is that a Simget sends messages stored in a Memento upon special Events to the current Observers/monitors of the Simget. Monitors are Simgets which are Clients to the Simulation Simgets. Here is a short code snippet to describe how it works

```
St::SApplicationServer* app = new St::SApplicationServer;
St::SMonitorOutput* mon = new St::SMonitorOutput( 0, "monitor", SYSLOG );
mon->setMonitor( St::MONITOR_TIME | St::MONITOR_RESULT | St::MONITOR_PARAMETERS );
app->setMainSimget( mon );
app->run();
```

the second line creates a monitor for all Simgets in the Directory Services which will use syslog to log events like computation time, results from the Simgets and parameters passed to the Simgets. In order to create a model specific monitor for the Fin3D model and its outputs, we use the following code for example:

```
St::SApplicationServer* app = new St::SApplicationServer;
St::SModel* model = new SModel( 0, "fin3d" );
St::SMonitorOutput* mon = new St::SMonitorOutput( fin3d, "monitor", SYSLOG );
mon->setMonitor( St::MONITOR_TIME | St::MONITOR_RESULT | St::MONITOR_PARAMETERS );
app->setMainSimget( model );
app->run();
```

A monitor can just log events but it provides also a CORBA interface that allows Clients like SimTeX for example to access statistics of the Simget being called. Note that this Monitor design pattern is not only used by St to provide monitoring of the Simget but it is also used for all kind of objects that requires monitoring like solvers, preconditioners, iterative processes in general. The monitor design pattern is a non trivial application of

several design patterns and is a reusable component/strategy in scientific computing libraries. We just provided an application in the context of the Framework and Distributed Computing.

XML. XML[11] is another trendy technology and it is often associated with Distributed Computing — see SOAP[12] or XML-RPC[13]. The eXtended Markup Language is used in the Framework as a meta-data language to describe the Simgets. If it exists at the creation of a Simget, an associated XML file is loaded and is available through the CORBA interface of the Simget. In the current implementation, the XML data contain only static information like the name of the Simget, the model name, some parameter set data — like the name of the parameters, their minimum and maximum values, and a description — and a description of the Simget. Here is a simple XML example:

```
  <!DOCTYPE St>
<St>
  <SModelOutput name="Troot" parent="fin3d" model="fin3d">
    <Author firstname="Christophe" lastname="Prud'homme"></Author>
    <Basis db="fin3d_Troot.bb" N="20" Nused="20"></Basis>
    <ParameterSet name="D" dimension="7">
      <Parameter ub="10" lb=".1"  name="k1">k1</Parameter>
      <Parameter ub="10" lb=".1"  name="k2">k2</Parameter>
      <Parameter ub="10" lb=".1"  name="k3">k3</Parameter>
      <Parameter ub="10" lb=".1"  name="k4">k4</Parameter>
      <Parameter ub="1"  lb=".01" name="Bi">Bi</Parameter>
      <Parameter ub="3"  lb="2"   name="L">L</Parameter>
      <Parameter ub=".5" lb=".1"  name="t">t</Parameter>
    </ParameterSet>
    <Description>
      Troot computes the temperature at the root of the 3D Thermal Fin.

      BlackBox: O(Q^2) in compliant case.

      See http://augustine.mit.edu/fin3d_1/fin3d_1.pdf  for more details.
    </Description>
  </SModelOutput>
</St>
```

This is particularly helpful for automatic generation of Clients for the Framework. For example SimLab uses this feature to automatically generate `.m` files for the Simgets registered in the Directory Service (see Sect. 5.3.2). Without having to communicate with the Server, the Client can for instance provide documentation about the Simget, and checks that the parameter set which will be sent to the Server is contained in $\mathcal{D}$ (see Sect. 3.1).

In the future an automatic C++ code generator for the Clients will be implemented using the XML meta-data provided by the Simgets registered in the Directory Service. At present only SimLab does automatic code generation using the XML meta-data.

5.5. A simple Client/Server implementation in C++

In a Client-Server paradigm, we have a Server side and a Client Side. In the next sections, we are going to present a sample code of a Server running the Simget computing the temperature at the root of the 3D thermal fin and one possible — while very simple — Client code in C++ accessing the code and the equivalent

[11]http://www.w3.org/XML/
[12]http://www.w3.org/TR/SOAP/
[13]http://www.xmlrpc.com/spec

in MatLaB using SimLaB in order to compare the amount of work needed. Some knowledge of C++ is required, however some of the general ideas and concepts appear in the code.

5.5.1. *Server side*

First look at what the main program looks like from the server point of view:

```
#include <SApplicationServer.hpp>
using namespace St;

int main(int argc, char** argv)
{
  SCommandLineArguments::init( argc, argv );
  SApplicationServer* server = new SApplicationServer();

  server->run();
}
```

The line `6` initializes the command line parsing system. Then, we define the St Server application using the `SApplicationServer` class. Unfortunately this example does nothing but running forever. Indeed the `server->run()` is an infinite loop, where the `server` is waiting for new requests.

Now we need to feed the server with *Simgets*.

```
SApplicationServer* server = new SApplicationServer();
SModel* fin3d = new SModel( 0, "fin3d");
server->setMainSimget( fin3d );
server->run();
```

With the two new lines `2` and `3`, we have created a possibly new Model in the St Services (Naming and Trading). First we define the `"fin3d"` model — second argument — which has no parent in the current `SApplicationServer` — first argument. Again this server is not very helpful, since it does not do any real computation.

We can add for example a `Simget` which compute the temperature at the root of the three-dimensional thermal fin — see line `2` in the next listing.

```
SModel* fin3d = new SModel( 0, "fin3d");
SModelOutput* troot = new Troot( fin3d, "Troot");
server->setMainSimget( fin3d );
```

The work in the main program is finished and doesn't need further work. when `server->run()` is executed the server will provide the `Troot` Simget associated with the model `fin3d`.

Now let us see what the `Troot` looks like:

```
#include <SModelOutput.hpp>
using namespace St;
class Troot: public SModelOutput
{
public:
  Troot( SModel* parent, const char* name ): SModelOutput( parent, name ) {<snip> }
  virtual SOutput* run( SParameterSet const& pset )
   {
     SOutput* output = new SOutput;
     // do the computation here for the parameter set pset
```

```
      . . .
      output->output = <prediction value>;
      output->error = <associated error estimator value>;
      return output;
    }
};
```

The last step is the execution of the Server:

```
fin3d_Troot --server augustine.mit.edu --daemon
```

The `--server` option tells the `SApplicationServer` to register the Simgets in `augustine.mit.edu`. Whereas the `--daemon` tells `SApplicationServer` to detach from its parent process and run forever in the background.

Under Unix/Linux, a database and the `init.d` system are used to start and stop automatically the Servers used and registered on one computer. The advantage is that if a computer containing Simgets is rebooted or shutdown for some reason then the Simgets will be cleanly shutdown. Indeed, one of the major difficulty with Distributed Computing is the fact that the Object have *external* references and they need to be deleted properly if the corresponding is being shutdown.

Looking back at the example presented above, it would seem that it is possible to have only one Simget per process. Recall the tree structure of the SObject/Simget classes plugged into `SApplicationServer` using its `setMainSimget()` member function, then enabling new Simgets in the same process as the one presented earlier is a one-liner per Simget provided that each Simget has been implemented like `Troot` for example.

```
SModel* fin3d = new SModel( 0, "fin3d");
SModelOutput* troot = new Troot( fin3d, "Troot");
SModelOutput* ttip = new Troot( fin3d, "Ttip");
SModelOutput* volume = new Troot( fin3d, "Volume");
server->setMainSimget( fin3d );
```

The code shown above adds two Simgets to the Framework — in this case the temperature at the tip of the 3D fin and the volume of the 3D fin. As one can see, in one process we can register one or more components for a given Model in the various services available and the associated code and programming time are reduced.

A note on parallelism. Finally let us present a possible extension for the Server part: the various registered Simgets in one process are often independent from each other — eventually a Simget could depend on the computation done by another one. For example a Simget providing a dimensional `Troot` would depend on the non-dimensional `Troot` counterpart: this kind of dependency requires generally simple algebra while the non-dimensional `Troot` would require an Online Code as described in Sections 3 and 4. The Simget could be accessed by several users at the same time, and it appears that exploiting parallelism is important in this case to make sure that the Framework is responsive enough with respect to multiple Clients and provide Real-Time response for each Client. We implement parallelism at two levels on the Server side. Firstly, we use a multi-threaded CORBA implementation — MICO/MT[14] which is about to be or already merged with MICO[15] — that enables us to answer "simultaneously" to concurrent requests. Secondly, we implement thread-safe Simgets so that when the Client asks for multiple evaluations — say a thousand randomly chosen parameters — we use the interface of `SModelOutput` which provides multiple parallel evaluations — see `SModelOutput::run(MultiParameter)` in the next section. In order to control the level of parallelism for many outputs, the `SApplication` base class provides a way of defining the number of possible multiple threads.

```
SApplicationServer* __app = new SApplicationServer;
```

[14] http://micomt.sf.net
[15] http://www.mico.org

```
// tell thread-aware classes (SModelOutput for example)
// and their subclasses  to use up to 3 threads for
// many outputs evaluations
__app->setNumberOfProcessors( 3 );
```

The Framework provides a simple way to allow multi-threaded classes. The developer has to use the class `SThread` as the base class for a multi-thread enabled class. Here is an example:

```
#include <iostream>
#include <SThread.hpp>

class A: public St::SThread
{
public:
 // a possible default constructor
 A( int __no = 1 ): St::SThread(), _M_no( __no ) {}

protected:
 // run() method has to be re-implemented
 void run()
 {
   St::SMutex __mutex;
   if ( __mutex.tryLock() )
     {
       std::cerr << "run thread number " << _M_no << std::endl;
     }
   else
     {
       ;//locking did not work
     }
  // __mutex.unlock() is called by destructor automatically
  // so no need to call it here
 }
private:
 int _M_no;
};
```

Note the utilization of the mutex class `SMutex` — which is provided by `SThread.hpp` — to ensure that the output will be written sequentially in the console instead of having both threads outputs being mixed up. Then the controller program can do the following:

```
int main( int argc, char** argv )
{
  St::SCommandLineArguments::init( argc, argv );
  St::SApplication* __app = new St::SApplication;
  // can either use NPROCS environment variable or
  // SApplication::setNumberOfProcessors( n ) to set the number of threads
  int __nprocs = __app->getNumberOfProcessors();
  A** __threads = new A*[ __nprocs ];
  for( int __i = 0; __i < __nprocs; ++__i )
    {
```

```
      // start a new thread and call thread1.run()
      __threads[ __i ] = new A( __i );
      __threads[ __i ]->start();
    }
  // wait for all threads to finish
  for( int __i = 0; __i < __nprocs; ++__i )
    {
      __threads[ __i ]->wait();
    }
}
```

The environment variable `NPROCS` controls the number of threads than can be used, it is overridden by the `SApplication::setNumberOfProcessors( int )` member function. When running the above code, we get the following expected console output:

```
console > export NPROCS=4
console > test_thread
run thread number 0
run thread number 1
run thread number 2
run thread number 3
```

The various classes handling the Online Codes computation uses `SThread` which eases the use of multi-thread programming by encapsulating all the `pthread` library.

To summarize, parallelism is achieved at two levels on the Server side: at the ORB level using a multi-threaded ORB and at the Simget level. Note that it is transparent to the Framework user except for the part he has to provide the number of threads and that his code has to be thread-safe in order to be used in this multiple threads (> 1) environment. If it is not thread-safe then the Framework user can still fall back to a single thread environment which is the default and conservative behavior. There is a Client parallel counterpart which is discussed in the next section which uses yet another feature of the Framework.

5.5.2. *Client side*

On the client side, the Client developer has access to the Naming and Trading services in order to find the object he is looking for. In the next listing, we present a client using the Naming service. The way the Simget in general are stored in the Naming service resembles a directory tree where `/` is the root of the tree. Under the `fin3d` model appears the `Troot`, `Ttip` and `Volume` Simgets. In the following example, we retrieve a reference to the `Troot` Simget and use its interface to do some computations:

```
#include <SApplication.hpp>
#include <SModelOutput.hpp>
using namespace St;


int main(int argc, char** argv)
{
  SCommandLineArguments::init( argc, argv );
  SApplication* client = new SApplication();

  SModelOutput_var fin3d_troot = client->resolve( "fin3d/Troot" );
  fin3d_Troot->setNewParameterSet( pset );
  fin3d_Troot->setRange( range );
```

```
  fin3d_troot->run( MultiParameter );

  coOutputArray_var results = fin3d_Troot->getOutputs();
  for( ULong  __i = 0;__i < __output_pl->length();++__i)
    {
      std::cerr << "output( " << __i << ")=" << result( __i ).output << std::endl;
      std::cerr << "error( " << __i << ")=" << result( __i ).error << std::endl;
    }
}
```

The only *non standard* C++ statement is the call to `client->resolve()` which will lookup in the naming service in order to get an handle on the corresponding simulation which was defined earlier. The other statements are classical object manipulation, setting the parameter set, set the range if needed, run the simulation and then retrieve the results.

Writing a client for the framework is relatively easy task, however providing a graphical user interface to the client code certainly comes with some efforts, that one of the reason why we based SimTEX and SimLaB Clients upon existing standards(MatLaB and PDF) — to alleviate the work in programming user interfaces.

Now it remains to execute the Client program:

```
fin3d_Troot_client --server augustine.mit.edu
```

This command line tells the Client to contact `augustine.mit.edu` for Simget lookups.

A note on parallelism. Parallelism is/can be also implemented on the Client side. However it is not an automatic builtin feature of the Client. The Framework allows to implement a Client which takes advantage of certain of its properties. The starting point is that, as mentioned in the previous section, the ORB is multi-threaded and recall that several instances of the same Model/Simget can exist in the CORBA Services — namely the Naming or the Trading Services — so with those two features one can build a Client which access several instances of the same Simget or several different Simgets at the same time in different threads. The `SThread` class presented in Section 5.5.1 can be used to implement such a multi-threaded Client. Another important part here is the load balancing system of the Services which ensures that the least used instance will be selected by the Service to answer each new request.

To summarize the parallelism aspects of the Framework, both Server and Client sides, we achieved a three levels parallelism which ensures real-time response — with the certificates of fidelity developed in the first sections of the paper — in the context of many evaluations and many users.

6. Conclusion

This framework is likely to be extended in the next few months, and will certainly enjoy some improvements. It has already been tested on many problems already available online on our web site `http://augustine.mit.edu`. The component-wise design of the overall system is very flexible, scales well as the number of Online Codes increases and is an elegant solution as a platform for engineering design, research and education. Regarding the mathematical framework, it has been extended to several problem categories: non-coercive partial differential equations, parabolic equations, generalized eigenvalue problems, non-symmetric and non-compliant cases, see [19].

Acknowledgements. We would like thank Thomas Leurent (formerly of MIT), Shidati Ali of the Singapore-MIT Alliance and Yuri Solodukhov for very helpful discussions. We would also like to acknowledge our longstanding collaborations with Professor Jaime Peraire of MIT and Professor Einar Rønquist of the Norwegian University of Science and Technology. This work was supported by the Singapore-MIT Alliance, by DARPA and AFOSR under Grant F49620-01-1-0458, by DARPA and ONR under Grant N00014-01-1-0523 (Subcontract # 340-6218-3), and by NASA under Grant # NAG-1-1978.

References

[1] M.A. Akgun, J.H. Garcelon and R.T. Haftka, Fast exact linear and non-linear structural reanalysis and the Sherman-Morrison-Woodbury formulas. *Int. J. Numer. Methods Engrg.* **50** (2001) 1587–1606.

[2] E. Allgower and K. Georg, Simplicial and continuation methods for approximating fixed-points and solutions to systems of equations. *SIAM Rev.* **22** (1980) 28–85.

[3] B.O. Almroth, P. Stern and F.A. Brogan, Automatic choice of global shape functions in structural analysis. *AIAA Journal* **16** (1978) 525–528.

[4] A. Barrett and G. Reddien, On the reduced basis method. *Z. Angew. Math. Mech.* **75** (1995) 543–549.

[5] T.F. Chan and W.L. Wan, Analysis of projection methods for solving linear systems with multiple right-hand sides. *SIAM J. Sci. Comput.* **18** (1997) 1698–1721.

[6] A.G. Evans, J.W. Hutchinson, N.A. Fleck, M.F. Ashby and H.N.G. Wadley, The topological design of multifunctional cellular metals. *Prog. Mater. Sci.* **46** (2001) 309–327.

[7] C. Farhat, L. Crivelli and F.X. Roux, Extending substructure based iterative solvers to multiple load and repeated analyses. *Comput. Methods Appl. Mech. Engrg.* **117** (1994) 195–209.

[8] J.P. Fink and W.C. Rheinboldt, On the error behavior of the reduced basis technique for nonlinear finite element approximations. *Z. Angew. Math. Mech.* **63** (1983) 21–28.

[9] L. Machiels, J. Peraire and A.T. Patera, *A posteriori* finite element output bounds for the incompressible Navier-Stokes equations; Application to a natural convection problem. *J. Comput. Phys.* **172** (2001) 401–425.

[10] Y. Maday, L. Machiels, A.T. Patera and D.V. Rovas, Blackbox reduced-basis output bound methods for shape optimization, in *Proceedings* 12th *International Domain Decomposition Conference*, Chiba, Japan (2000) 429–436.

[11] Y. Maday, A.T. Patera and J. Peraire, A general formulation for *a posteriori* bounds for output functionals of partial differential equations; Application to the eigenvalue problem. *C. R. Acad. Sci. Paris Sér. I Math.* **328** (1999) 823–828.

[12] Y. Maday, A.T. Patera and G. Turinici, Global *a priori* convergence theory for reduced-basis approximation of single-parameter symmetric coercive elliptic partial differential equations. *C. R. Acad. Sci. Paris Sér. I Math.* **335** (2002) 1–6.

[13] A.K. Noor and J.M. Peters, Reduced basis technique for nonlinear analysis of structures. *AIAA Journal* **18** (1980) 455–462.

[14] A.T. Patera and E.M. Rønquist, A general output bound result: Application to discretization and iteration error estimation and control. *Math. Models Methods Appl. Sci.* **11** (2001) 685–712.

[15] A.T. Patera and E.M. Rønquist, A general output bound result: Application to discretization and iteration error estimation and control. *Math. Models Methods Appl. Sci.* (2000). MIT FML Report 98-12-1.

[16] J.S. Peterson, The reduced basis method for incompressible viscous flow calculations. *SIAM J. Sci. Stat. Comput.* **10** (1989) 777–786.

[17] T.A. Porsching, Estimation of the error in the reduced basis method solution of nonlinear equations. *Math. Comp.* **45** (1985) 487–496.

[18] C. Prud'homme, *A Framework for Reliable Real-Time Web-Based Distributed Simulations.* MIT (to appear).

[19] C. Prud'homme, D. Rovas, K. Veroy, Y. Maday, A.T. Patera and G. Turinici, Reliable real-time solution of parametrized partial differential equations: Reduced-basis output bounds methods. *J. Fluids Engrg.* **124** (2002) 70–80.

[20] W.C. Rheinboldt, Numerical analysis of continuation methods for nonlinear structural problems. *Comput. Structures* **13** (1981) 103–113.

[21] W.C. Rheinboldt, On the theory and error estimation of the reduced basis method for multi-parameter problems. *Nonlinear Anal.* **21** (1993) 849–858.

[22] D. Rovas, *Reduced-Basis Output Bound Methods for Partial Differential Equations.* Ph.D. thesis, MIT (in progress).

[23] K. Veroy, *Reduced Basis Methods Applied to Problems in Elasticity: Analysis and Applications.* Ph.D. thesis, MIT (in progress).

[24] N. Wicks and J. W. Hutchinson, Optimal truss plates. *Internat. J. Solids Structures* **38** (2001) 5165–5183.

[25] E.L. Yip, A note on the stability of solving a rank-p modification of a linear system by the Sherman-Morrison-Woodbury formula. *SIAM J. Sci. Stat. Comput.* **7** (1986) 507–513.