REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including t
gathering and maintaining the data needed, and completing and reviewing the collection of information. Send cc
collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, [
Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork F

AFRL-SR-BL-TR-98-
0485

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	18 May 1998	FINAL 27 May 1997 - 26 May 1998

4. TITLE AND SUBTITLE
An Object-Oriented Toolbox for Distributed Parameter Control Design With Application to JSF

5. FUNDING NUMBERS
F49620-97-C-0025

6. AUTHOR(S)
Rajesh Bhaskaran (PI)
Kevin Long
Gal Berkooz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Beam Technologies, Inc.
110 N. Cayuga
Ithaca, NY 14850

8. PERFORMING ORGANIZATION REPORT NUMBER
0002AA

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR
110 Duncan Ave. Room B115
Bolling, AFB DC 20332

nm

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
0002AA

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unlimited

12b. DISTRIBUTION CODE

19980615 021

13. ABSTRACT *(Maximum 200 words)*
We have developed the core functionality for an object-oriented toolbox for modeling, control design and analysis of distributed parameter systems. The toolbox is based on PDESOLVE, our C++ class library for simulating partial differential equation (PDE) systems. The capabilities developed are: (1) High-level specification and analysis of the PDE control problem (2) Calculation of basic system matrices using a finite-element approximation of PDE control problems (3) Interface with MATLAB providing access to MATLAB functionality from within the control toolbox (4) Simulation of open and closed-loop response (5) Model reduction at the PDE level using a modal basis. We have successfully demonstrated the toolbox on the LQR control of plate vibration using a distributed actuator. This problem takes about 0.5 man-days to solve using the toolbox whereas it is estimated that it will take at least a man-month to solve it by modifying a conventional finite-element code.

14. SUBJECT TERMS
Distribution parameter systems
Control Design
Low-order controllers

15. NUMBER OF PAGES
11

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

DTIC QUALITY INSPECTED 3

1 Objectives

The overall goal of the Phase I effort was to develop the basic software framework for an object-oriented toolbox for modeling, control design and analysis of distributed parameter systems based on PDESOLVE, our C++ class library for simulating partial differential equation (PDE) systems. Specific technical objectives are listed below:

1. Develop the capability to calculate the basic system matrices through spatial discretization of linear PDE control problems using the finite-element method.
2. Develop an interface with MATLAB to allow the user easy access to the control methodologies in MATLAB such as LQG, $\mathcal{H}_\infty$ and μ-synthesis.
3. Develop the capability to perform model order reduction at the PDE level by using an appropriate eigenfunction family as the basis for the spatial discretization.
4. Develop the capability for simulating the open-loop and closed-loop response.
5. Demonstrate the Phase I product on a model PDE control problem.

2 Work Performed and Results

2.1 Approach

The approach taken in Phase I was to leverage PDESolve to rapidly develop the ability to perform PDE-based control. PDESolve is a C++ (object-oriented) class library that enables a high level expression of ordinary and partial differential equations, and interfaces with real-world engineering tools such as CAD systems. PDESolve was extended with new objects that are specific to control.

PDESolve is a new class of tool, offering mixed symbolic and numeric computing, open C++ architecture, easy interface with engineering tools and the potential for scalable performance. Existing tools such as MATLAB and Mathematica have some but not all of the above characteristics. The reusability and flexibility that come with the object-oriented and math-based architecture of PDESolve allows dramatically faster development of engineering solutions. These solutions are often one to two orders of magnitude shorter in terms of line count compared with programming in C or FORTRAN, and take accordingly less time to develop. More than $4 million dollars have been invested in PDESolve and related projects to date.

2.2 Results

In the Phase I effort, we have developed the core functionality for the control toolbox and applied it to various PDE control problems, thus demonstrating the technical feasibility of the proposed Phase II effort. The main accomplishments in Phase I are listed below:

1. High-level specification of PDE control problem
2. Calculation of system matrices
3. Interface with MATLAB providing access to MATLAB functionality from within the control toolbox

DTIC QUALITY INSPECTED 3

4. Simulation of open and closed-loop response
5. Demonstrations on model PDE problems in heat and vibration attenuation
6. Model reduction at the PDE level using the modal basis

Each of the above items is discussed in detail below. This discussion is illustrated using a simple physical example. Let $U(t,x,y)$ be the temperature distribution in a thin, homogeneous plate. The objective is to regulate the temperature of the plate using a distributed actuator $u(t,x,y)$ that can source and sink heat. The optimal control problem can be stated as follows: Find the control u that minimizes the performance index

$$J(u) = \int_{\Omega} \int_{t=0}^{\infty} \left\{ |y(t,x,y)|^2 + R|u(t,x,y)|^2 \right\} dt\, dx\, dy \tag{1}$$

subject to the PDE system

$$\begin{aligned} \frac{\partial U}{\partial t} &= k\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}\right) + u & (2)\\ y &= U & (3)\\ U(t,0,y) &= U(t,20,y) = U(t,x,0) = U(t,x,20) = 0 & (4)\\ U(0,x,y) &= \begin{cases} 2 & 8 \le x \le 12,\ 8 \le y \le 12 \\ 0 & \text{everywhere else} \end{cases} & (5) \end{aligned}$$

where $y(t,x,y)$ is the observed variable (a distributed sensor).

High-level specification of PDE control problem

We have developed the capability to specify the PDE control problem at the continuous level. Since we use the finite-element method for spatial discretization, the problem is input into the toolbox in the weak form which for eqs. (2)-(3) is given by

$$\int_{\Omega} \frac{\partial U}{\partial t} v\, d\Omega + k \int_{\Omega} \nabla U \cdot \nabla v\, d\Omega - \underline{\int_{\Gamma} \frac{\partial U}{\partial n} v\, d\Gamma} - \int_{\Omega} u\, v\, d\Omega = 0 \tag{6}$$

$$\int_{\Omega} (y - U)\, d\Omega = 0 \tag{7}$$

where v is the test function. Since the boundary conditions are Dirichlet, the underlined term is evaluated using Lagrange multipliers and is set to

$$\int_{\Gamma} \frac{\partial U}{\partial n} v\, d\Gamma = \int_{\Gamma} \lambda\, v\, d\Gamma + \int_{\Gamma} \delta\lambda\, U\, d\Gamma \tag{8}$$

$\delta\lambda$ and v being linearly independent test functions. The above weak form can be input directly into the toolbox as shown in the code segment below.

```
Function U(2, Scalar); // Unknown function
Function v(2, Scalar, VARIATIONAL); // Test function
Function lambda(2, Scalar); // Constraint dofs
Function varLambda(2, Scalar, VARIATIONAL); // Constraint test func
Function uControl(2, Scalar); // Control variable
Function y(2, Scalar); // sensor variable
Real kDiffusion = 0.25;
```

```
// Form cell complex and FE mesh using third-party mesher.
CellComplex cc = rectMeshGen(nx, ny, 0.0, 20.0, 0.0, 20.0);
FEMesh mesh(cc, 2);

// Weak formulation of the PDE.
Equations w(mesh);
w = Integral(v*(dt*U) + kDiffusion*(grad*U)*(grad*v))
  - Integral(uControl*v);
//Weak formulation of the sensor equation.
Equations s(mesh);
s = Integral( v*y - v*U);

// Impose essential (Dirichlet) boundary conditions using Lagrange
// multipliers
w["x=0"] = lambda*v + varLambda*U;
w["x=L"] = lambda*v + varLambda*U;
w["y=0"] = lambda*v + varLambda*U;
w["y=L"] = lambda*v + varLambda*U;
```

Note the one-to-one correspondence between the weak form and its specification in the control toolbox.

If instead of the above continuously distributed control, we had actuators at specified locations on the plate, we would identify the actuator locations with labels while generating the geometry/cell-complex. We then apply the control term only over the region represented by the actuator labels as shown below:

```
w = Integral(v*(dt*U) + kDiffusion*(grad*U)*(grad*v))
          - Integral("actuator", uControl*v);
```

where the location of heat source/sink is identified by the label "actuator".

Calculation of system matrices

The discrete control problem on the given finite-element mesh is formed using the `ControlProblem` class and the system matrices are obtained using the `getMatrices` method on it as shown in the following code excerpt:

```
ControlProblem problem(mesh, dt, w, s, List(U,lambda), List(v,varLambda),
                       uControl, y);
DenseMatrix A;
DenseMatrix B;
DenseMatrix C;
problem.getMatrices(A, B, C);
```

The arguments passed to the `ControlProblem` constructor are the finite-element mesh `mesh`, the time derivative `dt` to indicate the time direction, the weak form of the governing equations `w` and `s`, the list of unknown functions `List(U,lambda)`, the corresponding list of variational functions `List(v,varLambda)`, the control variable `uControl` and the sensor variable `y`. Most of this is specific to LQR problems and was done to simplify the description of the problem; a D matrix and a partition of B and C will be necessary for general control problems.

Interface with MATLAB

We have built an interface to MATLAB that allows easy access to all of MATLAB's capabilities from within our control toolbox. Users can perform optimal control design (LQG, $\mathcal{H}_\infty$, μ-synthesis, $\mathcal{L}_1$ optimal control etc.), plant and controller model reduction, simulate the designed controller on the discretized plant, and in general, perform arbitrary MATLAB operations based on the system information passed along through the interface.

The software architecture of the interface is interprocess communication (IPC). The MATLAB process and the toolbox operate as two separate processes, possibly running on different machines. Data is passed between them via IPC and a C-language interface provided by Mathworks Inc. Several C++ wrapper classes to the C-language interface have been implemented, in order to simplify use and provide a consistent interface with our data types. The resulting system allows both applications to operate on data in their native formats, resulting in a highly efficient system.

In the heat control example considered above, the feedback gains are calculated using the `lqry` function in MATLAB which performs linear-quadratic regulator design for continuous-time systems. The syntax for using the `lqry` function is

```
K = lqry(SYS,Q,R)
```

where `SYS` is the continuous-time system, `K` is the optimal gain matrix such that the state-feedback law `u = -KU` minimizes the cost function

$$J(u) = \int_0^\infty \left\{ y^T Q y + u^T R u \right\} dt \tag{9}$$

In the above, $y(t)$ and $u(t)$ are spatially discretized versions of $y(t,x,y)$ and $u(t,x,y)$. The following code segment illustrates the calculation of the gain matrix using the MATLAB interface:

```
//Start a MATLAB process on remote machine and display output on
//local machine:
MatlabShell matlab("rsh redfish matlab5 -display seneca:0.0");
//Transfer A and B matrices to MATLAB:
matlab.put(A, ``A''); // Pass matrix to MATLAB, associate it with name "A"
matlab.put(B, ``B''); // Pass matrix to MATLAB, associate it with name "B"
//Calculate gain in MATLAB.
matlab.eval("sys = ss(A, B, C, 0);"); //Form continuous-time system
int nSens = C.numberOfRows(); //Number of sensors
int nAct = B.numberOfCols(); //Number of actuators
char cmdStr[100];
sprintf(cmdStr, "K=-lqry(sys, eye(%d), 0.1*eye(%d));", nSens, nAct);
matlab.eval(cmdStr);
matlab.eval("save K");
//Import gain from MATLAB.
DenseMatrix gain;
matlab.get(gain, ``K'');
```

Note that in `matlab.eval(...)`, the part within the parentheses is what the user would normally type in within MATLAB. The output weighting matrix `Q` has been set to identity and the control weighting matrix `R` has been set to 0.1 times identity. Alternately, an interactive MATLAB shell can be opened using

```
matlab.interact();
```

and the feedback gain calculated interactively:

```
> K = -lqry(sys, eye(nSens), 0.1*eye(nAct))
```

Simulation of open and closed-loop response

We have implemented the capability to simulate open-loop and closed-loop response using the `openLoopSolve` and `closedLoopSolve` methods on the `ControlProblem` class. The use of these methods for the heat control example being considered is shown below. Note that the feedback gains are calculated in MATLAB as discussed earlier.

```
//Specify time stepping:
TimestepSpec step(initialTime, finalTime, nSteps);

// Simulate open-loop response:
Array<Function> solnOpen = problem.openLoopSolve(u0, step,
                                BackwardsEuler(), Filter(filterFunc));

//Ouput in MATLAB movie format:
outputAnimate("openLoop", animateTimeSeries(solnOpen));

//Simulate closed-loop response with the previously calculated gain:
Array<Function> solnClosed = problem.closedLoopSolve(gain, u0, step,
                                BackwardsEuler(), Filter(filterFunc));

//Ouput in MATLAB movie format:
outputAnimate("closedHeat2D", animateTimeSeries(solnClosed));

// Output L2Norm(T) vs time.
ofstream of("L2norm.dat");
for (int i=0; i<solnOpen.length(); i++) {
of << i << " " << l2Norm(solnOpen[i]) <<
          " " << l2Norm(solnClosed[i]) << endl;
}
```

The arguments to `closedLoopSolve` are the gain, the discretized initial condition u0, the time-step specification `step`, the integration method `BackwardsEuler` and the output filter `Filter(filterFunc)`. The discretized form of the initial condition (5) is generated with the following code:

```
Function u0(2, Scalar, u0Func);
FEDiscretizer disc(mesh);
u0 = disc.discretize(u0); //Discretize u0 on mesh
```

with the function definition for u0Func being given by

```
Real u0Func(const Coords& x) {
  if( x[0] >= 8 && x[0] <= 12 && x[1] >= 8 && x[1] <= 12)
    return 2.;
  else
    return 0.;
}
```

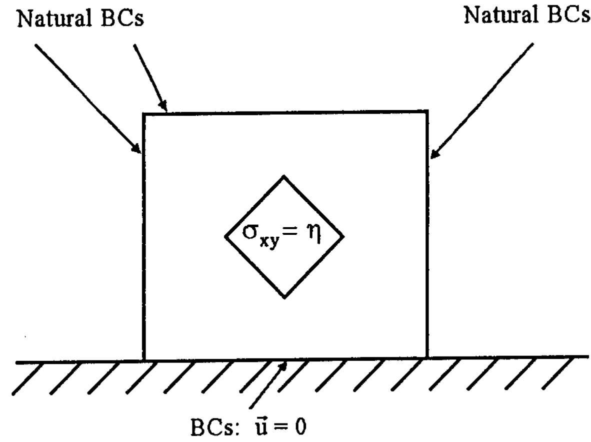

Figure 1: Schematic of the plane vibration demonstration problem. The control variable is σ_{xy} over the diamond-shaped region in the center.

The output filter allows any general mathematical operation to be performed on the solution field at each time-step and only the result from applying the filter is stored. This provides the user with the flexibility of storing only pertinent data (say, for example, the solution at certain critical points).

Demonstration on model problem: LQR control of plate vibration

We next demonstrate our code on the LQR control of the vibration of a plate. We model planar motion of a thin square plate using the plane stress approximation. The system is controlled by a stress actuator on a square at the center as shown in Figure 1; the control parameter is the xy component of the stress at the actuator.

The weak form with respect to a test function $\boldsymbol{v}$ of the governing equation is

$$\int_\Omega \rho \mathbf{v} \cdot \ddot{\mathbf{u}} d\Omega + \int_\Omega \gamma \mathbf{v} \cdot \dot{\mathbf{u}} d\Omega + \int_\Omega \boldsymbol{\sigma} \cdot \delta \epsilon d\Omega + \int_\Omega \frac{1}{2} \eta \left[\frac{\partial \delta v_x}{\partial y} + \frac{\partial v_y}{\partial x} \right] d\Omega = 0 \qquad (10)$$

We impose the constraint $\boldsymbol{u} = 0$ at the surface $y = 0$, and assume natural boundary conditions at all other surfaces. The plate is initially displaced in the plane as shown in Figure 2; the initial velocity field is zero everywhere.

Time series for the open and closed loop solutions are shown in Figure 3.

Fragments of the code used to solve this problem are presented below. For brevity, we have edited out some initialization code, the calls to the MATLAB interface, and PDESOLVE postprocessing steps; that code is similar to that in the heat equation example above. Note that although the dynamic elasticity equations are considerably more complex that the heat equation, the overall structure of the code is nearly as simple as in the heat equation example.

```
// geometry initialization code deleted for brevity.
// We have created a cell complex with labeled actuators.
```

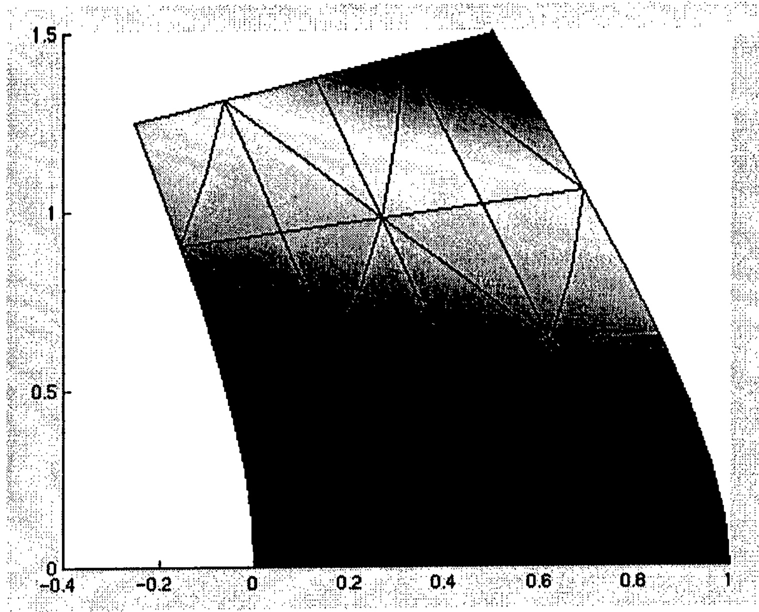

Figure 2: Initial displacement of the elastic plate. The initial velocity is zero.

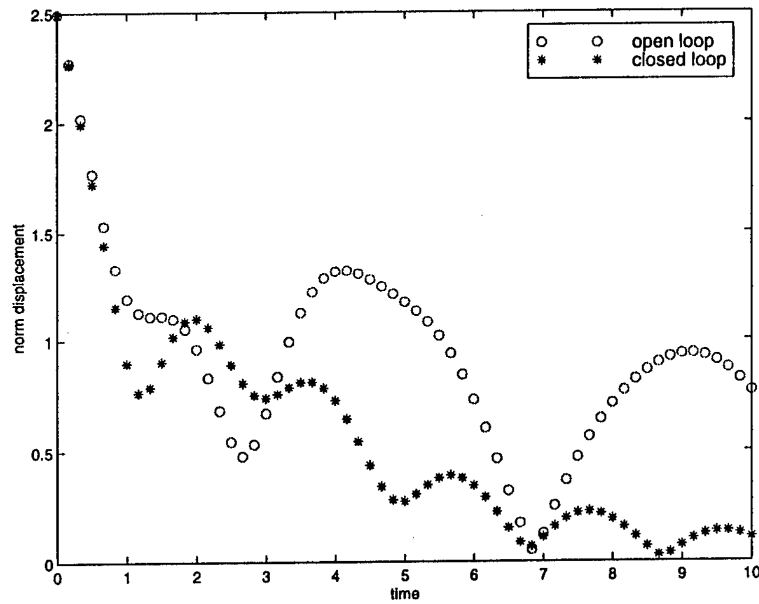

Figure 3: Time evolution of L^2 norm of displacement field for open loop and closed loop solutions of the plate vibration control problem.

```
FEMesh mesh(cc, 2);

DiffOp dx(1,0);
DiffOp dy(1,1);
DiffOp dt(1,2);
DiffOp dtt(2,2);

Function u(2,Tensor(2,1));                          // displacement field
Function v(2,Tensor(2,1),VARIATIONAL);              // test function
Function lagr(2,Tensor(2,1));                       // lagrange multiplier for BCs
Function varLagr(2,Tensor(2,1),VARIATIONAL);        // test function for BCs
Function z(2, Scalar);                              // control variable

Real damp = 0.05;                                   // damping constant
Real mu = E/(2.0*(1+nu));                           // Lame' constants
Real lambda = E*nu/((1+nu)*(1-2*nu));

// form stress-strain relation
Tensor material(3,2);
material = List( List(lambda+2*mu, lambda,      0 ),
                 List(lambda,      lambda+2*mu, 0 ),
                 List(0,           0,           mu));

Expr strainDisp = List(List(dx,0), List(0,dy), List(dy, dx));
Expr strain     = strainDisp * u;
Expr varStrain  = strainDisp * v;
Expr stress     = material * strain;

// set up weak form of equations
Equations w(mesh);
w       = Integral(v*dtt*u +  0.05*v*dt*u + stress*varStrain) +
Integral("actuator", v*dtt*u +  0.05*v*dt*u + stress*varStrain +
z*(dx*v[1] + dy*v[0]));

// variational enforcement of dirichlet BCs
w["y=0"]= lagr*v + varLagr*u;

// assemble control problem
ControlProblem problem(mesh, dt, w, List(u,lagr), List(v,varLagr), z);

// form discrete initial conditions
Function u0(2, Tensor(2,1), initPosFunc);
FEDiscretizer disc(mesh);
u0 = disc.discretize(u0);
Function v0(2, Tensor(2,1), initVelFunc);
v0 = disc.discretize(v0);
```

```
//
// calls to matlab deleted for brevity.
// see the first example in this section for matlab interface example code.
//

TimestepSpec step(0.0, 10.0, 60);
Array<Function> solnOpen = problem.openLoopSolve(List(u0,v0),
step, BackwardsEuler());

Array<Function> solnClosed = problem.closedLoopSolve(gain, List(u0,v0),
step, BackwardsEuler());

// postprocessing and animation calls deleted for brevity
}
```

Model reduction at the PDE level using the modal basis

The examples presented so far have used a finite-element discretization of the PDE. That is acceptable for control of small demonstration problems, but for real-world PDE control problems the number of degrees of freedom becomes prohibitively large. A useful alternative is to represent the system in terms of a small, dynamically meaningful set of basis functions such as an eigenmode decomposition or a proper orthogonal decomposition. Determination of an appropriate set of functions through an eigenmode analysis or POD is itself a computationally intensive problem, but once done, it allows the control designer to work with a far simpler system.

We have implemented the infrastructure needed to perform dimension reduction via a modal representation. In the example below, we consider control of the heat equation on a 2D square. Control is accomplished by four point source actuators inside the square. We represent the solution by the 16 lowest frequency modes on that domain; we obtain those modes though a finite-element analysis using PDESOLVE. Using the same finite-element mesh to do the controls problems results in a system with 225 degrees of freedom; the modal representation has reduced that to 16.

The changes in user code needed to switch from a full FE model to a modal representation are minimal. As shown in the code fragment below, essentially the only changes are (a) to use a `SpectralMesh` rather than an `FEMesh` object, and (b) the boundary conditions are built into the modes and are thus not needed in the modal calculation. The boundary conditions were of course used in the computation of the modes.

```
// initialization omitted; mode computation shown below

SpectralMesh mesh(modes);

Equations w(mesh);

w = Integral(v*(dt*u) + (grad*u)*(grad*v)) + Integral("actuator", z*v) ;

ControlProblem problem(mesh, dt, w, u, v, z);

// matlab calls, ODE solution, and postprocessing omitted
```

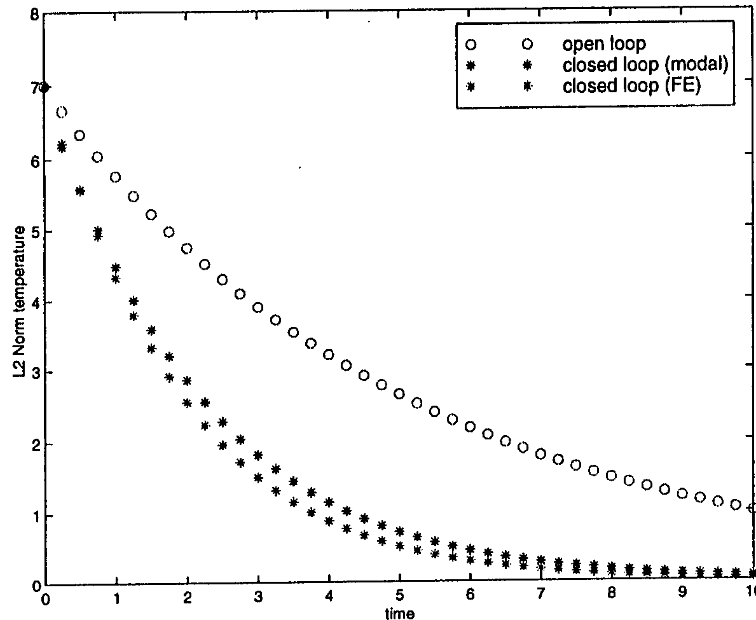

Figure 4: Comparison of full FE and reduced-dimension models of heat equation. Plotted are times series for L^2 norm of temperature for open loop and closed loop as solved with FE and modal methods.

In Figure 4 we show results for both the reduced-dimension model and the full FE simulation of the point-actuator controlled heat equation example. The open-loop solutions are indistinguishable in the plot. The closed-loop solutions with the two methods compare well; the differences are explained by the inability of the truncated modal basis to exactly represent the effect of point source terms.

We next present a code sample showing the mode computation in PDESolve. The PDESolve code sets up a general linear eigensystem that is then solved using the ARPACK family of iterative eigensolver routines. This, together with the control code, totals to under one hundred lines of user code for specification and solution of a dimension-reduced PDE control problem.

```
// initialization code omitted

DiffOp dx(1,0);
DiffOp dy(1,1);

Expr grad = List(dx,dy);

Function u(dim,Scalar);
Function v(dim,Scalar,VARIATIONAL);
Function invAlpha(dim, Scalar);
Function lambda(dim, Scalar);
Function varLambda(dim,Scalar,VARIATIONAL);

FEMesh feMesh(cc,order);

Equations heatModeEqn(feMesh);
```

```
heatModeEqn = Integral(invAlpha*(grad*u)*(grad*v) + u*v) ;
heatModeEqn["x=0"] = varLambda*u + lambda*v;
heatModeEqn["x=L"] = varLambda*u + lambda*v;
heatModeEqn["y=0"] = varLambda*u + lambda*v;
heatModeEqn["y=L"] = varLambda*u + lambda*v;

EigenProblem eigen(mesh, heatModeEqn, List(u, lambda),
   List(v, varLambda), invAlpha, ConstrainedARPACKSym(EIG_LARGEST, 16));

Array<Function> modes(0);
Array<Real> eigenvalues(0);

eigen.solve(eigenvalues, modes);
```

3 Personnel Involved

Dr. Gal Berkooz
Dr. Rajesh Bhaskaran
Dr. Kevin Long

4 Publications

None.