MORTestbed User Guide

Matthew J. Zahr[1]

[1] *Graduate Student. Institute for Computational and Mathematical Engineering. Stanford University*

May 11, 2013

Abstract

This document presents a brief outline of how to use and program the model order reduction (MOR) testbed, MORTestbed, initially created by the author during an internship for the Army High Performance Computing Research Center (AHPCRC) at Stanford University during the summer of 2010. Extensive revisions were made during the summer of 2011 to enable its use as a research tool.

Since this is a research code, it is being constantly updated and the updates are not usually documented. Any questions should be directed to the author directly (phone: 209-652-1251 or email: mzahr@stanford.edu)

Location of MORTestbed (FRG members): ssh://hg@ahpcrcfe.stanford.edu/MORTestbed

Contents

I Nonlinear Module **3**

1 Overview **4**

2 Code Organization **5**
2.1 Flow of MORTestbed 5
2.2 Directory Hierarchy 5

3 MORTestbed Core **6**
3.1 Models 6
3.2 Problems 6
3.2.1 1D Burger's Equation 6
3.2.2 Nonlinear Transmission Line 7
3.2.3 FitzHugh-Nagumo Equations - Neuron Modeling 7
3.2.4 Highly Nonlinear 2D Steady State Problem 7
3.2.5 Convection-Diffusion-Reaction 7
3.2.6 Micromachined Switch 7
3.2.7 Potential Nozzle 7
3.2.8 Quasi-1D Euler Equations 8
3.2.9 1D KdV Equations 8
3.2.10 3D Structural Dynamics 8
3.2.11 Lid-Driven Cavity (Incompressible Navier-Stokes Equations) 8

4 MORTestbed Access **9**
4.1 The Input Files: CFG, ROM, and PP 9
4.1.1 The Configuration (CFG) File 10
4.1.2 The Reduced Order Model (ROM) file 12
4.1.3 The Postprocessing (PP) File 14
4.2 The Workflow File 17

5 Future of MORTestbed **18**
5.1 ROM-based Optimization 18
5.2 Linear Module 18
5.3 Future of MORTestbed 18
5.3.1 ROM-based Optimization 18
5.3.2 Linear Module 18

Part I

Nonlinear Module

Chapter 1

Overview

MORTestbed was primarily developed by M. Zahr, under the guidance of K. Carlberg and D. Amsallem, in 2010 with funding provided by the Army High Performing Computing Research Center. Extensive revisions were made in 2011 by M. Zahr. D. Amsallem provided the core functions for the GNAT model reduction procedure used in the original version of MORTestbed and K. Carlberg offered extensive advice on techniques to accelerate the model reduction methods. Both individuals made many other significant contributions to the MORTestbed.

MORTestbed is a MATLAB program that provides researchers in the field of nonlinear Model Order Reduction (MOR) the ability to compare common MOR techniques and ideas on a set of benchmark problems. The code was designed to be dynamic; an object oriented programming approach was used to modularize the code in such a way that researchers can add their own MOR techniques and/or benchmark problems (and then easily compare them to the builtin techniques/problems). This adaptivity of the code is possible because of the complete separation between the MOR methods and the Problem formulation.

This document is organized as follows: Chapter 2 discusses the organization of the code including the directory hierarchy; Chapter 3 discusses the core features that define the MORTestbed including the MOR techniques, benchmark problems, and functions; Chapter 4 presents all information necessary for using the builtin MORTestbed features to perform an investigation regarding ROMs; Chapter **??** presents necessary information for successfully hacking MORTestbed; and Chapter 5 presents future directions of the MORTestbed.

Chapter 2

Code Organization

2.1 Flow of MORTestbed

The MORTestbed is a program designed in an object-oriented paradiagm whose functionality is determined by two major categories of classes: the MODEL classes and the PROBLEM classes. A class in the MODEL category is an reduced order model technique or the full order model; however, the specifics will be discussed later. At this point, we will view a class in the MODEL category simply as a class with a nonlinear solver and time stepping algorithm. A class in the PROBLEM category defines one of the benchmark problems either builtin to the MORTestbed or defined by a user. Again, the details will be discussed later; for now, we view a class in the PROBLEM category as a class with a method that returns the residual and jacobian given a state and time.

With this level of abstraction, the organization of the MORTestbed is relatively simple. Via input files to be discussed later, the user defines one or more PROBLEM s (instances of a class in the PROBLEM category) and one or more MODEL s (instances of a class in the MODEL category). These are defined *independently* of each other. At some point, the model and problem will linked together by storing a pointer to the appropriate problem in the model class. Now, simulations can be run because the nonlinear solver, time stepping algorithm, and residual/jacobian generator are in place. I have neglected to mention where the contributions to the nonlinear residual/jacobian from the time integration scheme come into play. These are defined in classes distinct from the MODEL and PROBLEM classes.

2.2 Directory Hierarchy

After cloning the MORTestbed to any directory on your local machine, there is a predefined directory hierarchy should not be altered. I will assume that the MORTestbed is cloned to the directory MORTestbed. To access the nonlinear module, navigate to MORTestbed/Nonlinear, which will be referred to as the Nonlinear Base Directory. From the Nonlinear Base, the following directory heiarchy defines the MORTestbed:

1. Classes - Contains all classes defining the MORTestbed including CONFIG, PROBLEM classes, MODEL classes, and POSTPROCESSING classes
2. InputFileFuncs - Contains functions that are recommended to use WITHIN input files
3. Miscellaneous - Contains functions that did not fit well into any other directory
4. ReadFunctions - Contains the functions that are used to parse the input files
5. TimeIntegrate - Contains functions that return the residual and jacobian contributions for all supported time integration schemes
6. usr - Contains the home directory of all current MORTestbed users
7. WorkflowFuncs - Contains functions that are recommended to use in workflow scripts

Chapter 3

MORTestbed Core

3.1 Models

3.2 Problems

The MORTestbed solves systems of nonlinear initial-value ODEs of the form

$$\frac{d\mathbf{\Phi}(t)}{dt} = f(\mathbf{\Phi}(t), t) + \mathbf{g} + \mathbb{B}\mathbf{u}(t) \tag{3.1}$$

$$\mathbf{\Phi}(\mathbf{0}) = \mathbf{q} \tag{3.2}$$

where $\mathbf{\Phi} : \mathbb{R}_+ \to \mathbb{R}^N$ is the unknown, $\mathbf{g} \in \mathbb{R}^N$ is the source term, $\mathbb{B} \in \mathbb{R}^{N\times\mu}$ is the input matrix, $\mathbf{u} : \mathbb{R}_+ \to \mathbb{R}^\mu$ is the input vector, $\mathbf{q} \in \mathbb{R}^N$ is the initial condition, and $f : \mathbb{R}^N \to \mathbb{R}^N$ nonlinear. Here, N is the dimension of the problem and there are μ inputs. Nearly all PROBLEM s in the MORTestbed are ODEs that arise from the semi-discretization of a PDE.

3.2.1 1D Burger's Equation

Remarks. *The 1D Burger's Equation is Problem 1 in the MORTestbed and the code for this equation can be found in the* PROBLEM *class OneDBurgers.m.*

Governing Equation

Burger's equation is a model problem for modeling the nonlinear effects of shock propagation in fluid dynamics. In one dimension, the governing equation is

$$\begin{aligned} \frac{\partial U(x;t)}{\partial t} + \frac{\partial f(U(x;t))}{\partial x} &= g(x) \\ U(0;t) &= u(t) \\ U(x;0) &= q(x) \end{aligned}$$

for all $t > 0$ and $x \in [0, L]$. In this problem, U is the unknown conserved quantity (mass, density, heat, etc.) and $f(U) = 0.5U^2$.

Discretization - Finite Volumes

This partial differential equation is spatially discretized using Gudnov's scheme (finite volumes) to yield a nonlinear ODE in the form of (3.1).

$$\frac{d\mathbf{U}(t)}{dt} = F(\mathbf{U}(t), t) + \mathbf{g} + \mathbb{B}u(t)^2 \tag{3.3}$$

$$\mathbf{U}(\mathbf{0}) = \mathbf{q} \tag{3.4}$$

See [1] for additional information on governing equations and discretization.

3.2.2 Nonlinear Transmission Line

Remarks. *The Nonlinear transmission line is Problem 2 in the MORTestbed and the code for this equation can be found in the* PROBLEM *class NLTransLine.m.*

Governing Equation

See [1] for governing equations and discretization.

3.2.3 FitzHugh-Nagumo Equations - Neuron Modeling

Remarks. *The FitzHugh-Nagumo Equations are Problem 3 in the MORTestbed and the code for this equation can be found in the* PROBLEM *class FHN.m.*

Governing Equation

See [2] for governing equation and discretization.

3.2.4 Highly Nonlinear 2D Steady State Problem

Remarks. *The highly nonlinear 2D steady state problem is Problem 4 in the MORTestbed and the code for this equation can be found in the* PROBLEM *class HNL2dSS.m.*

Governing Equation

See [2] for governing equation and discretization.

3.2.5 Convection-Diffusion-Reaction

Not currently supported...

Remarks. *The convection-diffusion-reaction problem is Problem 5 in the MORTestbed and the code for this equation can be found in the* PROBLEM *class ConvDiffReact.*

Governing Equation

See [3] for governing equation. Discretization is with FEM.

3.2.6 Micromachined Switch

Remarks. *The micromachined switch is Problem 6 in the MORTestbed and the code for this equation can be found in the* PROBLEM *class MEMS.m.*

Governing Equation

See [1] for governing equations and discretization.

3.2.7 Potential Nozzle

Remarks. *The potential nozzle is Problem 7 in the MORTestbed and the code for this equation can be found in the* PROBLEM *class SteadyNozzle.m.*

Governing Equation

See [4] for governing equations and discretization.

3.2.8 Quasi-1D Euler Equations

Remarks. *The quasi-1d Euler equations is Problem 8 in the MORTestbed and the code for this equation can be found in the* PROBLEM *class quasi1dEuler.m.*

Governing Equation

See [5] for governing equations and discretization.

3.2.9 1D KdV Equations

Remarks. *The 1d KdV equations is Problem 14 in the MORTestbed and the code for this equation can be found in the* PROBLEM *class OneDKdV.m.*

3.2.10 3D Structural Dynamics

This is a 3D, unstructured code for nonlinear structural dynamics discretized with the Finite Element Method. For now, the mesh must be 4 node tetrahedral elements; very straightforward extension to other solid elements coming soon. Also restricted to St. Venant-Kirchoff material (linear material), but the kinematics are nonlinear. All assembly type operations are done in C++ (internal force, body force, mass matrix computations). Only supports nodal external loads for now.

Remarks. *The structural dynamics equations are Problem 15 in the MORTestbed and the code for this equation can be found in the* PROBLEM *class structuralFEM.m. Code needs to be compiled before it can be used. There is a Makefile.m in Nonlinear/Classes/Problem/zFEM/ that should be run* from within MATLAB *as follows:*

```
>> Makefile(2);  Makefile(3);  Makefile(4);  Makefile(5);  Makefile(6);
```

3.2.11 Lid-Driven Cavity (Incompressible Navier-Stokes Equations)

Not currently supported

Remarks. *The lid-driven cavity is Problem 16 in the MORTestbed and the code for this equation can be found in the* PROBLEM *class lidINS.m.*

Chapter 4

MORTestbed Access

The nonlinear portion of the MOR testbed is accessed through a series of input text files and the ultimate flow of the program is specified in an M-file (script, not a function) by the researcher. This drastically improves the robustness of the MORTestbed from its original version.

At a high level, the code is organized into three disjoint collections of classes, the MODEL classes, the PROBLEM classes, and the POSTPROCESSING classes. The MODEL classes (FOM.m, ROM.m, GNAT.m, locGNAT.m, and TPWL.m) contain all of the properties and methods of a particular model (whether it be the high fidelity model or one of the reduced models). A method of a particular MODEL class may contain as much model-specific code as the researcher wishes to include (this statement will be apparent if the reader looks at the code contained in GNAT.m and locGNAT.m). The PROBLEM classes (OneDBurgers.m, NLTransLine.m, FHN.m, HNL2dSS.m, and MEMS.m) contain all of the properties and methods for a particular benchmark problem; and each of these classes may contain as much problem-specific code as the researcher wishes. To maintain this distinction between MODEL and PROBLEM classes, problem-specific code must never be put in a MODEL class; model-specific code in a PROBLEM class is OK and at times necessary (particularly when dealing with GNAT). The POSTPROCESSING classes (POSTPROCESS.m, ppFIGURE.m, ppAXES.m, and ppTABLE.m) are used to access and display the results of the simulations that were executed. There is actually one more class (CONFIG.m) that doesn't fit so nicely into this breakdown because it stores the information for a particular problem configuration, yet does not contain problem specific code.

Remarks. *The TPWL class has not been maintained and will probably not work. The* POSTPROCESSING *class has also not been maintained in quite awhile, but most features will still probably work. Most MORTestbed users find it more convenient to simply their postprocessing commands in the workflow file and omit the PP files. The PP files are useful in that they can significantly unclutter a workflow file by putting all postprocessing commands in an input file.*

Remarks. *This complete distinction between* MODEL *classes and* PROBLEM *classes is the main difference (although there are many) between this version of MORTestbed and the original that makes this version adaptable for researchers looking to incorporate new methods and problems.*

A suite of input files are used to define the properties of the above classes and a MATLAB *script* is used to define the flow of the MOR comparison (see 4.2 for details on this script).

4.1 The Input Files: CFG, ROM, and PP

At a high level, the MORTestbed needs two collections of information to perform a given analysis: the data and the instructions. The data for the analysis is specified in a configuration file (CFG), a reduced order model file (ROM), and a postprocessing file (PP). These files are text files with the extensions .cfg, .rom, and .pp, respectively. Each of these files will be discussed individually below, but first, it is necessary to make a couple remarks about syntax.

Remarks. *For the most part, the input files accept standard MATLAB notation with two notable exceptions.*

1. *Comments must begin with /* and end with */ (nested comments are supported)*
2. *Commands must be separate by a line break*

Remarks. *In the subsequent sections, I will discuss the fields available to the user to specify in the input functions. Nearly all fields in the CFG and ROM files have well-defined defaults (although some default values depend on the specific problem choosen), so if certain fields are not specificed in the input file, the default value will be used.*

Finally, let me introduce a definition that will be used throughout this document.

Definition 1. *A* **type-*block*** *of text is all of the text contained between* **type**: and the next block declaration, i.e. a FOM-block is all of the text contained between FOM: and the next block declaration.

4.1.1 The Configuration (CFG) File

The CFG file is used to specify the problem you will be analyzing and all of its parameters as well as information regarding the FOM. There are four types of blocks in this file: a GVAR-block, a VAR-block, a FOM-block, and (possibly multiple) CONFIG-blocks. The FOM-block is used to define properties of the high fidelity simulation as well as define the problem to be solved in the subsequent simulaion. The (possibly) multiple CONFIG-blocks define the parameters for the problem (defined in the FOM-block), which represent different configurations of your problem.

The GVAR-block
The GVAR-block is used to defined "global" variables that may be used in ANY input file (in the scope of a particular analysis) or in the workflow file. The variable names can be any valid MATLAB name EXCEPT x, y, z, or t because these are reserved. For example, if a variable is defined in the GVAR block in the CFG file, it may be used in *any* block in the ROM or PP files OR in the the workflow file.

The VAR-block
The VAR-block is used to defined variables that may be used ONLY in the input file where the block is defined. The variable names can be any valid MATLAB name EXCEPT x, y, z, or t because these are reserved.

The FOM-block
There must be one and only one of these in each CFG file

- fileRoot - Used as the root of the filenames for all binary files that may get saved (usually residual snapshots if saveNL = true)
- problem $\in \{1, 2, 3, 4, 6, 7, 8, 14, 15\}$ specifies the nonlinear system to solve
- T - 1×2 vector specifying the boundaries of the time domain
- nstep - scalar defining the number of time steps to use in the simulations (must be empty if dt nonempty)
- dt - scalar defining the time step size to use in the simulations (must be empty if nstep nonempty)
- maxIter - scalar indicating the maximum number of Newton iterations that may be performed per time step
- eps - vector indicating the tolerance parameters for convergence of the Newton solver
  - if eps is a scalar, its value indicates the relative residual tolerance for convergence of Newton's method, i.e. convergence when $R(u_k) < \text{eps} * R(u_0)$
  - if eps is a 1×2 vector, the first entry is the absolute residual tolerance for convergence of Newton's method and the second is a tolerance on the distance between iterates, i.e. convergence when $R(u_k) < \text{eps}(1)$ or $||u_k - u_{k-1}|| < \text{eps}(2)$
  - if eps is a 1×3 vector, the first entry is the relative residual tolerance (described in first bullet) and the next two entries are the absolute residual tolerance and absolute iterate tolerance (described in second bullet)
- timeQuiet - boolean indicating whether to print the progress of the simulation
- newtQuiet - boolean indicating whether to print warnings when the maximum number of Newton iterations is reached

The CONFIG-block

There may be as many of these as desired in a given CFG file. Each defines a new problem configuration.

- id - scalar used to identify the problem configuration
- type - string indicating whether this configuration is meant as a training configuration or an online configurations. This is not used by the MORTestbed, it is mainly for the user benefit.
- desc - string that describes the configuration. Again, this is not used by the MORTestbed, it is mainly for the user benefit.
- DLim - $n \times 2$ matrix, where n is the number of dimensions in your problem. The (k,1) entry specifies the lower bound of the k-th dimension of your domain. Similarly, the (k,2) specifies the upper bound of the k-th dimension of your domain.
- ndof - scalar indicating the number of unknowns in your system (i.e. after removing dirichlet BCs from the problem)
- nNodes - $1 \times k$ vector whose k-th indicates the number of nodes in the k-th dimension
- altFile - a filename (not as a string so don't use single quotes) that contains the relevant information for a given problem. This is currently only availabe for the MEMS problem (it is actually necessary for this problem due to the large number of parameters).
- BFunc - a single line of MATLAB code *OR* a filename (this file must be a M-file function that takes no inputs and returns a single output). This line of code or file will specify how to make the input matrix B.
- CFunc - a single line of MATLAB code *OR* a filename (this file must be a M-file function that takes no inputs and returns a single output). This line of code or file will specify how to make the output matrix C.
- DFunc - a single line of MATLAB code *OR* a filename (this file must be a M-file function that takes no inputs and returns a single output). This line of code or file will specify how to make the feedthrough matrix D.
- GFunc - a single line of MATLAB code *OR* a filename (this file must be a M-file function that takes up to k inputs, where k is the dimension of the problem, each input corresponds to the position vector along a particular dimension and returns a single output). This line of code or file will specify how to make the source term vector G.
- icFunc- a single line of MATLAB code *OR* a filename (this file must be a M-file function that takes up to k inputs, where k is the dimension of the problem, each input corresponds to the position vector along a particular dimension and returns a single output). This line of code or file will specify how to make the initial condition.
- param - vector containing the parameters of the problem (problem specific and currently only has meaning for the HNL2dSS problem)
- saveNL - boolean indicating whether to save nonlinear terms during the simulation
- inFunc = filename (not as a string) that contains the input function. This file must be an M-file function that accepts 1 input (the time) and returns 1 output (the input vector - the number of entries must be equal to the number of columns in B).

Remarks. *Even though in the above, I made a distinction between the FOM-block and the CONFIG-block fields, there is actually not much of a difference. Any property specified in the FOM-block can be specified in any of the CONFIG-block and vice versa. However, there may be no repeats, i.e. if there is a property specified in the FOM-block, then it may not be specified in the CONFIG-block (an error will be produced if this is attempted). The distinction between the FOM-block and CONFIG-block comes into play during the assignment of a problem configuration to a high-fidelity simulation. Any property specified in the FOM-block will be common to ALL high-fidelity simulations specified, whereas properties specifed in the CONFIG-block will be common to that configuration only. This distinction is more easily understood in an example: Suppose I want to simulate 10 high-fidelity model that have all of the FOM-block/CONFIG-block properties in common EXCEPT the input function. In my FOM-block, I would specify all of the above properties expect inFunc. Then, I would have 10 CONFIG-blocks, each with their own inFunc line. This feature was meant to alleviate as much code copying as possible.* The point

here is to specify everything common to ALL your simulations in the FOM-block and those specific to a particular configuration in the CONFIG-blocks. *However, keep in mind that you will eventually be linking your configurations to ROMs, so some foresight will be necessary in your organization (i.e. don't plan everything based on your FOM simulations).*

Remarks. *In the above, nstep and dt cannot both be specified (even if they are consistent). Specifying both will results in an error. This is a convention.*

4.1.2 The Reduced Order Model (ROM) file

The ROM file is used to declare the parameters for any reduced order models (that are supported by MORTestbed) that the user wishes to generate. There is an important syntactic difference between the blocks in this file vs. those in the CFG file: the user can now specify a parameter for MULTIPLE models in a single line. As before, these lines adhere to MATLAB syntax and therefore, any MATLAB function can be used to generate these multiple ROM parameters. This capability becomes important when one wants to compare many ROMs (which is the whole point of the MORTestbed).

The rules for specifying multiple ROMs in one line are as follows: create a COLUMN vector (either a double vector or cell array) and each entry of the column vector will correspond to the parameter for a different ROM. For fields that have more than one entry per ROM (i.e. the snapDist field below), then you will specify a matrix (double or cell), where each row corresponds to a different ROM and the columns of a particular row will represents the different entries of the field for a given ROM.

There will be zero or one of the following blocks in a ROM file: GROM-block (Galerkin projection), PGROM-block (Least Square Petrov-Galerkin), GNAT-block, and TPWL-block.

The GROM-block and PGROM-block

The field dimensions given below are based on the assumption that we are generating only 1 ROM object in the block. Generalization to multiple ROM objects in the block can be acheived by adhering to the convention defined above.

- fileRoot - Used as the root of the filenames for all binary files that may get saved (usually residual snapshots if saveNL ¿ 0)
- id - scalar used to identify the ROM
- snapcoll $\in$ {'ref_init', 'ref_none', 'ref_prev' } indicating the snapshot collection to use: 'ref_init' = state vectors referencing the initial condition, 'ref_none' = unreferenced state vectors, and 'ref_prev' = state vectors referencing the state vector at the previous step. nsnap - scalar OR the string 'all' indicating how many snapshots to take in the snapshot interval (defined below)
- snapInt - 1×2 vector indicating the time interval where it is valid to collect snapshots. If left empty (or omitted), the entire time interval will be used.
- snapDist - this is used to select snapshots based on a distribution, where the format is a 1×3 cell array where the first entry is the random seed number (this is included for repeatability of an experiement), the second entry is a string containing the name of the distribution, and the last entry is a vector of doubles (of length p) defining parameters of the distribution (i.e. if we have the normal distribution, p = 2 and the parameters are the mean and standard deviation). This is a pedantic option and is not going to be maintained or updated unless I am contacted with interest in this option. For more information, contact the author.
- nY - scalar indicating the dimension of the ROM
- saveNL $\in \{0, 1, 2, 3\}$ indicating the nonlinear terms. saveNL = 0 will not save nonlinear terms (makes ROM computation faster) and saveNL = j (j = 1, 2, or 3) will perform snapshot collection j from Carlberg 2010 et. al).
- maxIter - scalar indicating the maximum number of Newton iterations that may be performed per time step during the ROM simulation
- eps - scalar indicating the tolerance parameter for convergence of the Newton solver for the ROM simulation
  - if eps is a scalar, its value indicates the relative residual tolerance for convergence of Newton's method, i.e. convergence when $R(u_k) < \text{eps} * R(u_0)$

  - if eps is a 1×2 vector, the first entry is the absolute residual tolerance for convergence of Newton's method and the second is a tolerance on the distance between iterates, i.e. convergence when $R(u_k) < \text{eps}(1)$ or $||u_k - u_{k-1}|| < \text{eps}(2)$
  - if eps is a 1×3 vector, the first entry is the relative residual tolerance (described in first bullet) and the next two entries are the absolute residual tolerance and absolute iterate tolerance (described in second bullet)
- timeQuiet - boolean indicating whether to print the progress of the ROM simulation
- newtQuiet - boolean indicating whether to print warnings when the maximum number of Newton iterations is reached during the ROM simulation
- nBases - scalar indicating the number of local Bases to generate
- addElem - boolean indicating whether to add vectors from adjacent bases so transition between bases is smooth (unused if nBases = 1)
- addElemTol - scalar indicating the percentage of vectors to add to the each basis from the surrounding bases (unused if nBases = 1)
- basisSelect $\in$ {'closest', 'interpolate'} determines which algorithm to use in selecting an appropriate basis online (only closest is supported at this time).

The GNAT-block

The field dimensions given below are based on the assumption that we are generating only 1 GNAT object in the block. Generalization to multiple GNAT objects in the block can be acheived by adhering to the convention defined above.

- id - scalar used to identify the ROM
- nR - dimension of the subspace we constrain the residual to lie in
- nJ - dimension of the subspace we constrain the jacobian to lie in
- nI - number of sample INDICIES to use for GNAT simulation (must be empty if nSample and nGreed specified)
- nSample - number of sample NODES to use (must be empty if nI specified) for GNAT simulation
- nGreed - number of vectors of phiR and phiJ to use when selecting sample nodes (must be empty if nI specified)
- addInd = string indicating which algorithm to use when adding residual entries that come for "free". Must be either 'all' or 'samevar'. Contact D. Amsallem for details.
- maxIter - scalar indicating the maximum number of Newton iterations that may be performed per time step during GNAT simulation
- eps - scalar indicating the tolerance parameter for convergence of the Newton solver during GNAT simulation
  - if eps is a scalar, its value indicates the relative residual tolerance for convergence of Newton's method, i.e. convergence when $R(u_k) < \text{eps} * R(u_0)$
  - if eps is a 1×2 vector, the first entry is the absolute residual tolerance for convergence of Newton's method and the second is a tolerance on the distance between iterates, i.e. convergence when $R(u_k) < \text{eps}(1)$ or $||u_k - u_{k-1}|| < \text{eps}(2)$
  - if eps is a 1×3 vector, the first entry is the relative residual tolerance (described in first bullet) and the next two entries are the absolute residual tolerance and absolute iterate tolerance (described in second bullet)
- timeQuiet - boolean indicating whether to print the progress of the GNAT simulation
- newtQuiet - boolean indicating whether to print warnings when the maximum number of Newton iterations is reached during the GNAT simulation

Remarks. *If nBases > 1 in the ROM file, then the GNAT model built on top of this ROM will be a local GNAT model (with the same number of bases as the ROM).*

The TPWL-block
The field dimensions given below are based on the assumption that we are generating only 1 TPWL object in the block. Generalization to multiple TPWL objects in the block can be acheived by adhering to the convention defined above.

4.1.3 The Postprocessing (PP) File

The postprocessing file is used to create figures and tables based on the results of your simulation. The blocks for this function are FIGURE-block, TABLE-block, and AXES-block. Unlike the other input files, there is a hierarchy in this function in that TABLE-blocks and AXES-blocks must be nested inside FIGURE-blocks.

The FIGURE-block
The FIGURE-block is very simple with only 3 fields:

- id - scalar used to identify the ppFIGURE object
- axesLayout - 1×2 vector defining the layout of ppAXES objects and ppTABLE objects in the figure. The first entry denotes the number of rows of subplots/subtables and the second entry denotes the number of columns of subplots/subtables.
- setfig - $1 \times 2k$ ($k \in \mathbb{N} \cup \{0\}$) cell array of strings indicating any figure properties that the user wishes to set manually.

The AXES-block
The AXES-block is more complicated with quite a few more fields:

- id - scalar used to identify the ppAXES object
- subplotNum - scalar or vector indicating the suplot numbers to plot in (same convention as MATLAB)
- xData - $M \times 1$ cell column array (or 1×1 cell array) of strings where each entry contains the key word of the data the user wants to be display on the x-axis
- yData - $M \times 1$ cell column array of strings where each entry contains the key word of the data the user wants to be display on the y-axis
- zData - $M \times 1$ cell column array of strings where each entry contains the key word of the data the user wants to be display on the z-axis
- scaleData - 1×3 array of real numbers containing the factors by which to scale xData, yData, and zData
- xlabel - cell row array where specifying the xlabel and its formatting for the current axes object
- ylabel - cell row array where specifying the ylabel and its formatting for the current axes object
- zlabel - cell row array where specifying the zlabel and its formatting for the current axes object
- title - cell row array where specifying the title and its formatting for the current axes object
- plotspec - $M \times N$ (or $1 \times N$ cell array) specifying the formatting of each plotted object, i.e. the jth row corresponds to the formatting the plot generated with xData{j}, yData{j}, zData{j}, etc.
- model - $M \times 1$ cell column array (or 1×1 cell array) of strings where each entry contains the key word of the model to be used in the plotting (i.e. which model to extract data from). Must be either fom#, rom#, gnat#, or tpwl#. The # indicates which entry in the model array that this corresponds to (this will be demonstrated with an example).
- modelAux - $M \times 1$ cell column array (or 1×1 cell array) of strings where each entry contains the key word of the auxilary model to be used in the plotting of the model results (this is only necessary to reconstruct the full vectors from the GNAT method). Must be either ' ' (empty string) or rom#. The # indicates which entry in the model array that this corresponds to (this will be demonstrated with an example).

- modelFom - $M \times 1$ cell column array (or 1×1 cell array) of strings where each entry contains the key word of the FOM that the model to be used in the plotting was based on (mainly for speedup and error computations). The # indicates which entry in the model array that this corresponds to (this will be demonstrated with an example).
- pType = $M \times 1$ cell column array of strings where each entry contains the key word of the type of plot to generate: animate (only valid if xData = animate**I**, see below), plot, semilogx, semilogy, loglog, plot3, surf, surfc, contour, or contourf.
- normType - scalar indicating the norm type to use when computing errors
- numPlotObj - scalar indicating the number of object to include in this axes object (equal to M in the convention used here)
- legend - $1 \times (h+1)$ ($h \in \mathbb{N}$) cell array of cells used to define the legend for the current axes, where h is the number of objects that the user wishes to include in the legend. The first cell is a cell array of strings defining the properties of the legend that the user wishes to set manually (i.e. the interpreter or location) and the remaining cells are 1×2 cell arrays containing the information to be displayed in the legend. The first entry of this $1x2$ cell contains the object number that their legend entry will be defining and the second entry is the name that the user wishes to assign to that object.
- connWithCurve - only used if the plotted objects are POINTS - cell column array of double vectors indicating the points to connect with a line
- connCurveSpec - only used if the plotted objects are POINTS - cell matrix array of strings (must have either 1 column or the same number of columns as connCurveSpec) that will define the formatting of the lines that connect the points indicated in connWithCurve

Remarks. *As with the ROM input file, if $M \geq 1$ objects (in this case plots) are specified, then every field must have either M or 1 rows. In only 1 row is used, then the same value of the field will be used for all objects.*

The TABLE-block

The TABLE-block is yet more complicated with fewer fields, but an implicit structure defined:

- id - scalar used to identify the ppTABLE object
- subtableNum - identical functionality of subplotNum above for tables
- normType - scalar indicating the norm type to use when computing errors
- numRows - scalar indicating the number of rows the table will have
- numCols = scalar indicating the number of rows the table will have
- rowData - This must be a column cell array of strings containing key words (see below for key words) indicating the data to be included in each row. In general, key word evaluates to a vector (may be of size 1). The result is that numCols will be unused and each vector will be placed in the appropriate row. Should be empty if colData or elemData nonempty.
- colData - This must be a row cell array of strings containing key words (see below for key words) indicating the data to be included in each column. In general, key word evaluates to a vector (may be of size 1). The result is that numRows will be unused and each vector will be placed in the appropriate column. Should be empty if rowData or elemData nonempty.
- elemData - This must be a 1×1 cell array, numRows $\times$ numCols cell array, $1 \times$ numCols cell array, or numRows $\times 1$ cell array of strings containing key words. Each keyword must evaluate to a scalar. Must be empty if colData or rowData nonempty. The functionality is defined as follows: (1) if elemData is a 1×1 cell array, the scalar data indicated by keyword will be put in every row and column of the table, (2) if elemData is a numRows $\times$ numCols cell array, then the scalar data corresponding to entry (i, j) of elemData will be placed in entry (i, j) of the table, (3) if elemData is a numRows $\times 1$ cell array, then the scalar data corresponding to entry $(i, 1)$ of elemData will be placed in entry (i, j) for $j = 1, \cdots$, numCols of the table, and (4) if elemData is a $1 \times$ numCols cell array, then the scalar data corresponding to entry $(1, j)$ of elemData will be placed in entry (i, j) for $i = 1, \cdots$, numRows of the table .

- title - Not currently used
- rowNames - cell array containing the rows names (can be either a row cell array or column cell array). This may be empty ([] or {}) or omitted and row names will not be assigned.
- colNames - cell array containing the column names (can be either a row cell array or column cell array). This may be empty ([] or {}) or omitted and column names will not be assigned.
- model - see AXES-block for definition. May be a 1×1 cell array of model strings (same model will be used for all entries of table), numRows $\times$ 1 cell array (same model will be used for each row of the table), 1 $\times$ numCols cell array (same model will be used for each column of the table, or a numRows $\times$ numCols cell array (each entry of the table will have its own model).
- modelAux - see AXES-block for definition. May be a 1×1 cell array of model strings (same model will be used for all entries of table), numRows $\times$ 1 cell array (same model will be used for each row of the table), 1 $\times$ numCols cell array (same model will be used for each column of the table, or a numRows $\times$ numCols cell array (each entry of the table will have its own model).
- modelFom - see above AXES-block for definition. May be a 1×1 cell array of model strings (same model will be used for all entries of table), numRows $\times$ 1 cell array (same model will be used for each row of the table), 1 $\times$ numCols cell array (same model will be used for each column of the table, or a numRows $\times$ numCols cell array (each entry of the table will have its own model).

Keywords

In the following, I will define the keyword that are to be used to request outputs in the PP file. I will use the convention that the state vector (or state vector-like quantities such as local basis cluster centers or POD vectors) is generated by stacking multiple variables on one another in a vector and thus the state vector-like quantity is: sv = [sv1, sv2, ..., svN]^T. In the below, we also use the convention that a **bold** letter indicates a variable integer or real number.

Keywords to be used in the xData, yData, zData fields of the PP File

- animate**I**: This will generate the animation number **I** for the current problem (see section on problem specifics for information on what **I** represents for each problem). This must be specified as xData AND it is assumed that this is the only plot specified for the current figure. Once this key is encountered, the animation plot will consume the entire figure, erasing all previous axes in the figure and never generating later axes (in the current figure).
- time: This returns a vector of times (length = nstep+1)
- timestep: This returns a vector of timesteps (= [1:nstep])
- xdomain: This returns the mesh points in the 1st dimension
- ydomain: This returns the mesh points in the 2nd dimension
- sv**I**@x=**J**: This returns sv**I** at position x= **J** (only works for 1D) of the Model using ModelAux (only needed for GNAT)
- sv**I**@t=**J**: This returns sv**I** at time t= **J** of the Model using ModelAux (only needed for GNAT)
- clusCenter**I**@b=**J**: This returns the cluster center **I** for the **J**th local basis of Model using ModelAux (only needed for GNAT)
- pod**I**@n=**J**,b=**K**: This returns the **J**th pod vector **I** for the **K**th local basis of Model using ModelAux (only needed for GNAT)
- output: This returns the output (C*sv) at each time, including initial time, of the Model using ModelAux (only needed for GNAT)
- ontime: This returns the online time of the Model
- speedup: This returns the online speedup of the Model over the ModelFom
- Aerr[%]: This returns the absolute error of Model in the *normType* norm at each time (including initial time). If % is included, the error will be given as a percent; if it is omitted, the error will not be a percent.

- Rerr[%]: This returns the relative error of Model in the *normType* norm at each time (including initial time). If % is included, the error will be given as a percent; if it is omitted, the error will not be a percent.
- MAerr[%]: This returns the maximum absolute error over all time steps. If % is included, the error will be given as a percent; if it is omitted, the error will not be a percent.
- MRerr[%]: This returns the maximum relative error over all time steps. If % is included, the error will be given as a percent; if it is omitted, the error will not be a percent.
- AvgAerr[%]: This returns the average absolute error over all time steps. If % is included, the error will be given as a percent; if it is omitted, the error will not be a percent.
- AvgRerr[%]: This returns the average relative error over all time steps. If % is included, the error will be given as a percent; if it is omitted, the error will not be a percent.
- linpt: This returns the linearization points used for TPWL (only valid for TPWL model; will error if you try to plot this for other models)
- prop.*field*: This returns the *field* property of the Model.

Remarks. All of the fields specified in all of the input files are evaluated using MATLAB eval command (since they are read as strings). Therefore, instead of specifying a command inline, the user may define a separate file that returns the appropriately formatted entry (they may even take literals as inputs). This can become quite important when you want to generate a plot from 100 different gnat simulations. Instead of typing {'gnat1'; 'gnat2'; 'gnat3'; ... ; 'gnat100'}, one could simply have a separate function that loops from 1 to 100 and stores the string ['gnat', num2str(j)] in the jth entry of a cell array and returns this cell array. This function is included in the MORTestbed with file name: 'createModelField.m'. I have found the most use for this in the PP files, but could also be employed in the ROM files.

4.2 The Workflow File

The workflow files are used to run simulations. These can be either functions or scripts, but scripts are highly recommended. These allow MORTestbed users to read the data from their input files and turn them in to model and problem objects that are used to run their simulations. Additional details can be found in comments of example files.

Remarks. *Before the MORTestbed can be used, the command*

```
>> init
```

must be run from the command line in the folder:

```
MORTestbed_Dist/Nonlinear/
```

Chapter 5

Future of MORTestbed

5.1 ROM-based Optimization

A future direction of the testbed is to include capabilities for PDE-constrained optimization. This will hopefully be a useful research tool to those interested in determining how one would used ROMs instead of high-fidelity models in an optimization loop. After usability aspects of the code are addressed, this will be the next major enhancement.

5.2 Linear Module

The underlying code for the linear module is complete, but it will undergo extensive revision to enhance performance, usability, and adaptability for use as a research tool. A front-end text file interface does not currently exist, but will be created using similar syntax as the nonlinear module. The focus of this project has been nonlinear problems, so they were given precedence. This will be completed at a later time. One interesting addition that will be made to this section is the ability to include the techniques developed by D. Amsallem and C. Farhat for parametric ROMs (manifold interpolation).

Bibliography

[1] M. J. Rewienski, *A trajectory piecewise-linear approach to model order reduction of nonlinear dynamical systems*. PhD thesis, Citeseer, 2003.

[2] S. Chaturantabut and D. Sorensen, "Discrete empirical interpolation for nonlinear model reduction," in *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, pp. 4316–4321, IEEE, 2009.

[3] M. F. Barone, I. Kalashnikova, M. R. Brake, and D. J. Segalman, "Reduced order modeling of fluid/structure interaction," *SAND2009-7189, Sandia National Laboratories Report*, 2009.

[4] L. T. Biegler, O. Ghattas, M. Heinkenschloss, and B. van Bloemen Waanders, *Large-scale PDE-constrained optimization: an introduction*. Springer, 2003.

[5] R. W. MacCormack, "Numerical coomputation of compressible viscous flow." Lecture Notes for AA214b and AA214c (Stanford University).