GASpAR User's Guide

Duane Rosenberg, Aime´ Fournier,
&
Annick Pouquet

Geophysical Turbulence Program
Turbulence Numerics Team (TNT)
of the
National Center for Atmospheric Research
Boulder, Colorado USA

Abstract

GASpAR is a an object-oriented, operator-based spectral element code, whose primary purpose is to solve a variety of PDEs in multiple spatial dimensions using adaptive high order spectral element methods. This document will guide the user through the steps required to run the code on serial and parallel platforms, and provide a detailed description of the code design and many of the underlying algorithms. While the focus of the guide will be on the solution of the time-dependent Navier-Stokes and advection-diffusion equations, the design considerations will enable one to create other types of solvers (*e.g.* steady-state and time-dependent) using the existing operator objects and methodologies.

Contents

Chapter 1

Introduction

The GASpAR(**G**eophysical and **A**strophysical **Sp**ectral element **A**daptive **R**efinement) code is designed to solve a variety of PDEs using adaptive high-order spectral element methods. We are strongly motivated by the following requirements: The code must

- allow for a variety PDE solvers
- allow for a variety of boundary conditions
- provide adaptive mesh capability
- provide a variety of time-stepping schemes
- be capable of running serially and on a distributed or shared memory parallel system.

These considerations suggest that an (native) object-oriented approach be taken for the code development. The code is written in C++, C, and Fortran. The goal of an object-oriented approach is that much of the code can be re-used. In our case, the objects comprise *methods* for carrying out tasks, and insofar as they are related, they are grouped into *classes*. But a by-product of this concept, which emerges from an examination of the spectral-element method (SEM) [31], is that the primary objects are *operators* which are used in the solution of the equations (cf. Chapter 8). While we use these operators

for solving the Navier-Stokes and advection-diffusion equations in this Guide, they may be re-used to solve entirely different classes of problems, with only modest additional work.

This guide will help the user to run the GASpAR code. The focus will be on the solution of the time-dependent advection-diffusion and Navier-Stokes equations. The document will also attempt to help the user become familiar with the code architecture sufficiently to make modifications to the existing code, or even to determine how the code might be altered to accommodate different physics, if desired.

We will assume that the reader is at least not repulsed by the sight of C and C++, and that he/she knows about makefiles and basic Unix commands. Most of the presentation on serial processing will, if details are required, center on the Linux operating system, and for parallel processing, will refer mainly distributed memory cluster-type systems. So, some basic knowledge of these systems may be helpful.

Throughout this Guide, we adopt the conventions that text entered on the command line will appear as bold typewriter text; file names will appear in quotes, and stand-alone executables or packages will appear in italics; and function/method/class and parameter names will appear in boldface.

The first chapter will enable the user to run GASpAR quickly on canned test problems, and will provide a description of operational basics. The user will also be guided quickly through the procedure for setting up a new problem in Chapter 3. Subsequent chapters will look in more detail at I/O (Chapter 6), and utility classes and functions used in earlier chapters (Chapter 7). In Chapter 5 we present some of the most important MATLAB ingestion and analysis tools that are supplied with the distribution. Finally, in Chapter 8 we offer an in-depth look at the our adaptive spectral element method as applied to advection-diffusion problems, and offer some observations about the application of the method to studies of turbulent flows.

If you have any questions, comments or concerns, please feel free to contact us using the contact information provided in Appendix D. While we cannot at this time offer formal support to users, we would appreciate any feedback.

Chapter 2

Operational basics

This chapter looks at basic code operation, and, in general, is intended to enable the user to run a problem as quickly as possible. It should be recognized that while some of the discussion deals with interactive use of the code, not all operating systems will easily allow this type of operation. This chapter will not necessarily distinguish between interactive and non-interactive (batch) operation; however, all the examples will work on a system that allows interactive jobs.

2.1 A look at the source tree

We assume that you have downloaded the GASpAR code tarball, **gaspar.tar**. You should simply expand the tarball, and enter the directory, "gaspar" that is created:

> **tar xvf gaspar.tar**
> **cd gaspar**

You will notice several new subdirectories are also created. These are the "src", "bin", "doc", and "test" subdirectories. The basic contents of the "src" directory is described in Table 2.1.

Table 2.1: Description of source code subdirectories

blas	:	basic linear algebra routines and interfaces to GASpAR objects
comm	:	all inter-processor communication code and data
exec	:	upper-level executive, output, and restart files
include	:	basic data types, and misc other global definitions
io	:	all I/O classes
matlab	:	all MATLAB ingestion and analysis scripts
mesh	:	mesh generator code and makefiles
sem	:	SEM operators and basis classes
solvers	:	linear solvers, and preconditioners
utils	:	generic, and GASpAR-related utility namespaces, utility classes

The "bin" directory contains the stand-alone utilities described in Chapter 4. To make these utilities, simply enter the bin directory, and type

> **make all**

The "doc" subdirectory contains this Guide, as well as comprehensive documentation on all the sources in the distribution. One may simply point a browser a the "doc/html/index.html" file to find the method interfaces for all the classes, namespaces and functions in the source tree.

Finally, the "test" directory contains the test source codes and scripts and is the directory from which we will begin running problems.

2.2 Running test problems: A quick start

We are interested in the "test" subdirectory:

> **cd test**
> **ls**

There is a makefile in this directory ("makefile") that can be modified for the local system. In §3.8 we discuss the systems on which the code has been

tested. The preprocessor definitions that may be set in the makefile are provided in Appendix B. For now, these should not be changed.

Once the makefile has been modified to give the proper paths for the MPI libs (if desired) and the compilers, build and run the code:

> **make**
> **./gauss_test.csh**

The script "gauss_test.csh" runs the Gaussian sphere test discussed in §8.2.2. This test uses the advection-diffusion solver. The script creates a directory "gauss_test" which contains subdirectories of each run, labeled "gauss_test_nrX_nY", where X is the number of grid refinement levels, and $Y = P + 1$, where P is the polynomial expansion order. Before each run, the data file "gaspar_gauss.dat" is copied to the run subdirectory. This file contains the parameter data, and is read before execution.

The shell script shows that the line starting the code execution contains command line parameters as well. In all cases, the command line parameters override those parameters set in the parameter file (see Sec.7.2). The output resides in binary files prefixed with the subdirectory name, and suffixed with the time indices, and a log file is created that is prefixed with "log.", for each run. The log file contains system-wide ("static") data and time-dependent ("dynamic") data. The system specifies which parameters are placed in the log file, but the user can add to it any additional static or dynamic data desired (cf.§3.5). In this problem, the final column in the log file gives the error of the computed solution with respect to the analytic solution. We will consider how to view the binary file output in §2.6, and in Chapter 5.

In addition, for convenience, there is a user-defined log file called "gauss.conv" that gives the expansion order vs. the final error for all the runs that are computed by the script. The user may plot the two columns in this file to arrive at a plot similar to that in Fig. 8.3(b). The dynamic log file data can be compared with that in Figs. 8.4(c-d).

There are a number of other advection-diffusion problems that can be run as well. Each problem is set up in a file called a *user* file, and is named with a prefix "guser_". These are located in the "gaspar/src/user" directory. For

example, if we want to run the N-wave example discussed in §8.2.3, we do the following:

> **cd ../src/user**
> **cp guser_nwave_test.cpp guser.cpp**
> **cp guser_nwave_test.h guser.h**
> **cd ../../test**
> **make clean**
> **make**

For this problem, there is also a script, "nwave_test.csh", that can be run to reproduce some of the results discussed in §8.2.3, particularly Fig. 8.6. As with all the scripts, the variable "HOME" in this script tells where the output from this run can be found. The output file name conventions follow those in the above example. Also, a user-defined log file, "nwave.conv", provides convergence data in a form convenient for comparing with results in Fig. 8.6.

The final advection-diffusion test setup is in "guser_heat_test.cpp", and this can be run in the same way as the above test problems; output can be compared with that in Fig. 8.3(a) and Figs. 8.4(a-b).

The user file "guser_kovaszny.cpp" runs a test of the Navier-Stokes solver. To run this test, you must first modify the "build/makefile" by commenting out the "SOLVER=BURGERS" line and replacing it with "SOLVER=NS". Then type

> **cd ../src/user**
> **cp guser_kovasznay.cpp guser.cpp**
> **cp guser_kovasznay.h guser.h**
> **cd ../../test**
> **make clean**
> **make**

as before. There is also a script ("kov_test.csh") that will make a series of runs at different orders in order to compute the solution and its error as a function of expansion order. The directory "kov_test" contains subdirectories with the results from the different runs. In each of these subdirectories,

the log file, prefixed with the word "log" gives the run diagnostics. The final column of the log files gives the solution error as a function of time. The polynomial expansion order is gotten from the run subdirectory name as above. For example, "kov_test_nX" means that the expansion order is $P = X - 1$. A plot of the error vs. polynomial expansion order should look like Fig. (2.1). The data for making this plot is provided in the user-defined log file "kov.conv" for convenience.

The straight line indicates spectral convergence, one of the nice properties of the spectral element method. [Note that for $P > 12$, the error norm in this plot is limited by the iterative solver's error tolerance, which was set low.]

This problem is a steady-state test, that is nevertheless achieved by using the time marching algorithms. The user is free to modify the **NR** parameter in the "kov_test.csh" script in order to investigate the effects of different levels of refinement on convergence. Be warned that the fixed timestep (set on the command line invocation of the executable within the script, with a **-dt**) may have to be reduced for large **NR**.

We have encapsulated much of operation discussed in this section within scripts, and set up the directories in order to facilitate getting the user up and running on some canned problems. But note that there is nothing magical about these scripts or the directory structure. The user is free place the sources, binaries, scripts and data in whatever locations relative to each other that she feels comfortable with.

2.3 How to use the command line and the parameter file

The shell scripts in the test problems indicate that a number of parameters are altered on the command line from those specified in the parameter file, "gaspar*.dat". For example, the options

> ./gaspar -if gaspar_gauss.dat -m data/cone_mesh_4x4_n5_p.dat -nr 3 -of tst -lf log.tst -c 0.2 -time 0.2

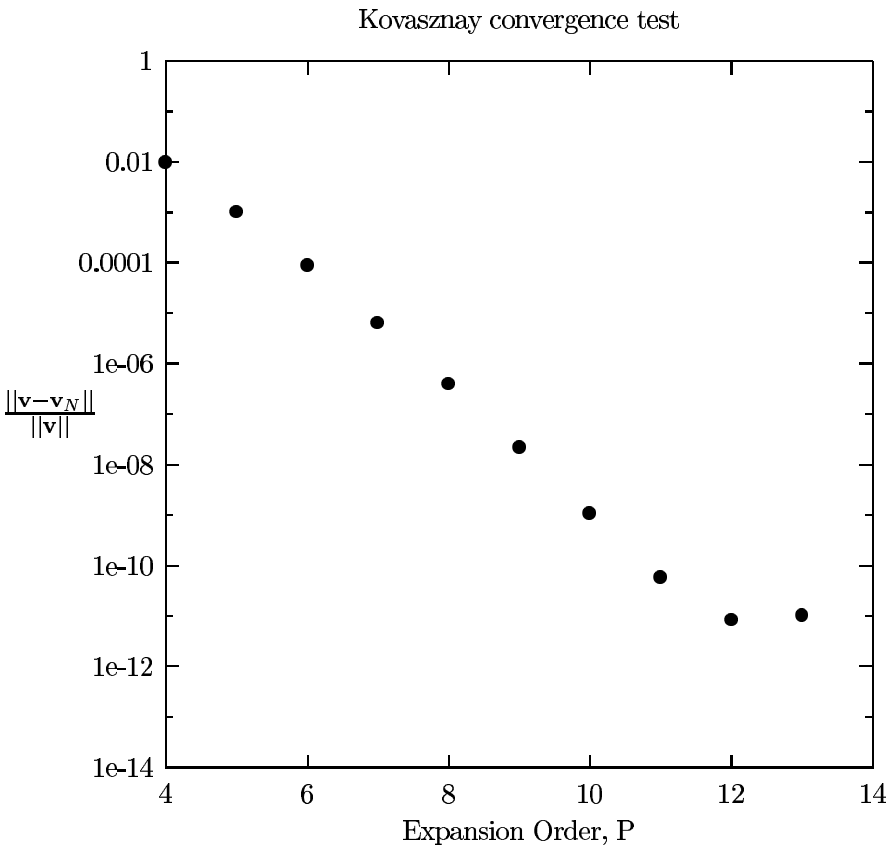

Figure 2.1: Plot of L^2 error norm vs. expansion order in Kovasznay flow problem

tell GASpAR to read parameter data from the file "gaspar_gauss.dat"; use the mesh file **data/cone_mesh_4x4_n5_p.dat** for creating the grid; to allow a maximum of 3 refinement levels (**-nr 3**); to place the output in files prefixed with the name "tst" (**-of tst**); to place log information in the file **log.tst** (**-lf log.tst**); to set the Courant number for a variable timestep (**-c 0.2**); and to integrate to a time of 0.2 (**-time 0.2**).

Most of the parameters specified in this file can be changed on the command line. By typing

> **./gaspar -h**

one can view the list of parameters that may be entered on the command line. Command line parameters always override parameter file settings. There is no ordering required for specifying command line parameters. If a parameter is repeated on the command line, the final specification is the one that takes effect. Some command line parameters have no parameter file equivalent. At least one command line option (**-if paramfile**) can have no parameter file equivalent (this changes the parameter filename). The complete list of command line options is provided here; it is largely self-explanatory. This list is just the screen output from issuing the **./gasp -h** directive.

```
Enter:

 ./gaspar [{-Nx}{-Ny} #Elems] [{-xNx}{-xNy} Exp_Order] [-P0 x0 {y0}] [-P1 x1 {y1}] [-(no)u1(2)pc]
        [-(no)ppc] [-upctype Type] [-stokestype Type] [-spectralap YN] [-deriv1ap YN] [deriv2ap YN]
        [-aptol[1][2] Tol] [-apmult[1][2] Mult] [-p(u)iter #Iter] [-p(u)tol Tol]
        [-bdy Face# Type] [-c Courant#] [-dt TimeStep] [-dd Text] [-nu val] [-nu1 val] [-nu2 val]
        [-rho val] [-time MaxTime] [-cycles MaxCycles] [-r] [-rg] [pgrid] [-rc YN] [-noadvect] [-linadvect]
        [-bal Y/N] [-filter Y/N] [-filter_strength alpha] [-filter_delta d] [-no_outongridchg]
        [-ocb OutCyc_Beg] [-oce OutCyc_End] [-ocd OutCyc_Skp] [-otb OutTim_Beg] [-ote OutTim_End]
        [-otd OutTim_Skp] [-acd AMRCyc_Skp] [-atb AMRCyc_Beg] [-atd AMRTim_Skp] [-atb AMRTim_Beg]
        [-lcd LogCyc_Skp] [-dcd DmpCyc_Skp] [-nfit NspFit] [-sigtol Tol] [-if Cmd_File]
        [-of Out_File] [-lf Log_File] [-uf User_File] [-rf Rst_File] [-df Dmp_File]
        [-m Mesh_File] [-ub BlkName] [-noadapt] [-dealias Y/N] [-nr nLevels] [-h]

 Where:
      -Nx #Elems      : Number of elements in x-direction. Default is 1.
      -Ny #Elems      : Number of elements in y-direction. Defaulst is 1.
      -xNx Exp. Order : Expansion order in x-direction. Default is 4.
      -xNy Exp. Order : Expansion order in y-direction. Default is 4.
      -P0 x0 {y0}     : Bottom left corner. There must be dim of these. Default is (0,0,0).
      -P1 x0 {y0}     : Top right diagonal corner. There must be dim of these. Default is (1, 1,1).
      -(no)upc        : Use (don't use) preconditioner for u1 and u2.
```

```
-(no)ppc        : Use (don't use) preconditioner for p.
-upctype Type   : Use preconditioner type Type for u1 and u2. Type may be:
                    0: GPC_BLOCKJAC_HELM
                    2: GPC_POINTJAC_HELM
                    3: GPC_NONE
                    Default is GPC_NONE.
-stokestype Type: Use Stokes solver type Type. Type may be:
                    0: STOKES_SCHUR_DELP--Perot decomp. with pressure correction
                    1: STOKES_UZAWA     --Uzawa splitting
-spectralap YN  : Use spectral a-posteriori error criterion (1); else don't (0). Default is 1.
-deriv1ap   YN  : Use 1-derivative a-posteriori error criterion (1); else don't (0). Default is 1.
-deriv2ap   YN  : Use 2-derivative a-posteriori error criterion (1); else don't (0). Default is 1.
-aptol     Tol  : Set spectral error refinement tolerance.
-apmult   Mult  : Set spectral coarsening scaling factor.
-aptol1    Tol  : Set 1-derivative refinemtn tolerance.
-apmult1  Mult  : Set 1-derivative coarsening scaling factor.
-aptol2    Tol  : Set 2-derivative refinemtn tolerance.
-apmult2  Mult  : Set 2-derivative coarsening scaling factor.
-p(u)iter Iter  : Max no. iterations for pressure (velocity) solve.
-p(u)tol Tol    : Error tolerance for pressure (velocity) solve.
-bdy Seg Type   : Set boundary condition Type on face (edge, point) given by Seg.
                  Seg is given in 1d by X0; in 2d by segment endpoints X0 Y0 X1 Y1.
                  Type is one of: 0 (Dirichlet),
                  1 (Neumann), 2 (Periodic), 3 (No-slip), 4 (None). Default is 4.
-c   Courant#   : Set Courant number to Courant.
-dt  TimeStep   : Set time step to TimeStep. Courant number not used.
-bscaled_dt y/n : If fixed time step, scale it due to refinement or not (0 or 1). Default is 1.
-dd  Text       : Pass Text to grid partitioner
-ab  N          : Set Adams-Bashforth order for adv. term (1, 2, 3, or 4)
-ext N          : Set extrapolation order for adv. term (1, 2, or 3)
-bdf N          : Set BDF order for derivative (1, 2, 3, or 4)
-et  N          : Set time stepping method: 0=> OIFS; 1=>ABBDF; 2=>EXBDF.
-nu  val        : Set kinematic viscosity to val.
-nu1  val       : Set 1- kinematic viscosity to val.
-nu2  val       : Set 2- kinematic viscosity to val.
-rho val        : Set density to val.
-time val       : Set max evolution time to val.
-cycles val     : Set max evolution cycles to val.
-r              : Do a restart with no regridding (default restart file is 'gaspar.dmp').
-rg             : Do a restart with a regrid (default restart file is 'gaspar.dmp').
-pgrid          : Construct a PERIODIC [0,1]^2  grid with Nx x Ny elements, each
                  of expansion order (xNx x xNy).
-rc             : Read command file on startup? (0, 1; default is 1).
-noadvect       : Do not perform advection.
-no_outongridchg: Do not do SDS output when grid changes.
-linadvect      : Do linear advection with specified advection velocity.
-bal Y/N        : Do load balancing (1/0). Default=0.
-filter Y/N     : Use filtering? (1/0). Default=0.
-filter_strength alpha
                : Set filter strength = alpha. Default=0.0.
-filter_delta d : Set filter smoothing delta = d.Default=1.
-ocb OutCyc_Beg : Output cycle-begin.
-oce OutCyc_End : Output cycle-end.
-ocd OutCyc_Skp : Output cycle-delta.
-otb OutTim_Beg : Output begin time.
-ote OutTim_End : Output endtime.
-otd OutTim_Skp : Output time-delta.
```

```
-acd AMRCyc_Skp : AMR cycle-delta.
-acb AMRCyc_Beg : AMR cycle-begin.
-ace AMRCyc_End : AMR cycle-end.
-atd AMRTim_Skp : AMR time-delta.
-atb AMRTim_Beg : AMR time-begin.
-ate AMRTim_End : AMR time-end.
-bcd LBalCyc_Skp: Load balance cycle-delta.
-bcb LBalCyc_Beg: Load balance cycle-begin.
-bce LBalCyc_End: Load balance cycle-end.
-btd LBalTim_Skp: Load balance time-delta.
-btb LBalTim_Beg: Load balance time-begin.
-bte LBalTim_End: Load balance time-end.
-lcd LogCyc_Skp : Logging cycle-delta.
-dcd DmpCyc_Skp : Dump cycle-delta.
-nfit nSpFit    : Number of coefficients to use in spectral fit.
-sigtol Tol     : Spectral decay rate tolerance.
-if Cmd_File    : Get system options from Cmd_File, instead of from gaspar.dat (default).
-of Outut_File  : Place output in Output_File. Default is results.######.
-lf Log_File    : Log to file Log_File. Default is gaspar.log.
-uf User_File   : Get user data from User_File. Default is gaspar.user.
-rf Restart_File: Use 'Restart_File' as restart file. Default is gaspar.dmp.
-df Dump_File   : Change default dump file name from 'gaspar.dmp' to Dump_File.
-m Mesh_File    : Use 'Mesh_File' to generate the mesh.
-ub BlkName     : Use BlkName as parameter block in the command file, instead of UserBlk (default).
-noadapt        : Turn off grid adaption.
-dealias        : Set Dealias flag (0,1). Default is 0.
-nr nLevels     : Maximum number of refinement levels. Default is 0 (no adaption).
-h              : Print this help text.
```

If the parameter file is not found, then all tunable parameters revert to their defaults, and a message is displayed. The default values of the tunables are contained in the "src/exec/gaspar_t.h" file. The lack of a parameter file is not considered an error. It is inadvisable to use any of the grid-altering command line parameters (*e.g.* , **-Nx(y)**,**-xNx(y)**, **-P0(1)**, or **-bdy**), since the ability to generate "on-the-fly" grids will be removed entirely soon.

If you look at the parameter file, you'll notice several things. First, there are two sections (parameter blocks) within it. [Note: a parameter file with a full list of allowed parameters is included in Appendix A.] The first is called **GASpAR_MAIN**, and the second, **GASpAR_AUX**. Both sections are read on initialization, but only **GASpAR_MAIN** is read if the restart flag (or command line equivalent) is set. If either one of these parameter blocks is missing on initialization, or if the **GASpAR_MAIN** group is missing upon restart, then an error is issued, and the code halts.

Also, note the structure of a parameter file block:

```
GASpAR_MAIN
{
//_____________________________________________________________________________________________
// General time and output control
//_____________________________________________________________________________________________
  Do_Restart                          : 0;          //0->no restart; 1->restart
  Restart_Using_File_Name             : gaspar.dmp; // Used if Do_Restart = 1
  Output_Time_Specification           : 0;          // 0->Time-based; 1->Cycle-based
  Output_Time_Begin                   : 0.0;        // Begin output at this time
  Output_Time_End                     : 20.0 ;      // End output at this time (also run termination time)
  Output_Time_Delta                   : 0.1 ;       // Output at this time interval
  Output_Cycle_Begin                  : 0;          // Begin output at this cycle number
//_____________________________________________________________________________
//_____________________________________________________________________________
 ...
}
```

The parameter block must have a name (**GASpAR_MAIN** here), be bracketed by the pair {}, and for each keyword, the format must be

```
keyword: value;
```

Neither the spaces nor, strictly speaking, the carriage returns are important. Also, any text starting with "//" is ignored; these indicate that a comment follows, and may be placed anywhere in the file. These rules are general for any parameter file read by GASpAR. For example, a user may want to initialize a problem using a parameter file (this is discussed more in section 7.2). If so, then the same rules apply. The parameters in the default ("gaspar.dat") parameter file is the full list of parameters that may be modified. However, the file need not contain all of the parameters; those that are not found in the file will revert to the defaults specified in the "gaspar_t.h" file.

2.4 Mesh Generation

In the test problems above, a mesh file was specified for each run, by using the **-m** option on the command line. At one time, GASpAR was set up to produce grids itself if none was specified. However, this capability will soon be deprecated, and the code will *require* a mesh file in order to generate a grid.

The mesh reader used by GASpAR requires that the mesh file conform to a certain format (described in section 7.3), but there is no prescription for creating a mesh, so that one may be provided from any source, so long as it has a format that the **MeshReader** object can handle. In fact, the mesh files in the Kovasznay test were created using the mesh-generation utility *gdd*, and then modified to produce the unregularized meshes that are used. More on the *gdd* utility can be found in Chapter 4.

2.5 Restarts

GASpAR has the ability to restart a problem and continue time-marching. A special binary dump ("**.dmp**") file is created at specified intervals during a run, which contains all of the information required to restart the run from the point where it left off. In reality, any ordinary output file *can* serve the purpose of the dump file, however, information, such as the time level *history* of the state, is not currently included in the output files. So there will be some start-up error on restart if an ordinary output file is used.

The dump file name, and dump cycle interval are set in the parameter file, or on the command line. The default dump file name is "gaspar.dmp", and it is (re)-created every 100 cycles, by default. Currently, an existing dump file is moved to a new file, tagged by a "%", and the new dump file created. Thus, at any time, there will be two dump files separated in time by the dump cycle interval. The dump file name can be changed in the parameter file (using the **Default_Dump_File_Name parameter**) or on the command line (with the **-df** option); the dump interval is governed by the **Dump_Cycle_Delta** parameter and the **-dcd** command line option.

2.6 Output preliminaries

This section discusses, briefly, how output is managed, and what formats GASpAR supports. Details of the output objects can be found in Chapter 6. By "output", it is meant the placing of the current time-stamped state, in some form, into a file. Currently, this is distinguished from the process of

"dumping" data from which the code can be restarted, and time marching continued (see section 2.5). The output file name is set in the parameter file ("gaspar.dat") using the quantity **Output_File_Prefix** or on the command line with the **-of** option. By default, the output routine will append a cycle index to this string in order to time-tag it. The file contains all physical quantities (velocity components, pressure), their grids, and time (physical and cycle number) information. Additional information (e.g., an analytic solution) may be included easily (see §3.2).

The frequency of the outputs is governed by the **Output_Time_Specification** parameter in the parameter file. If this parameter is 0, then the **Output_Time_Begin, Output_Time_End**, and **Output_Time_Delta** parameters in the file are read, and output occurs starting at **Output_Time_Begin**, ending at (or close to) **Output_Time_End**, with an interval **of Output_Time_Delta**. This interval may not be exact; it will depend on the timestep, which, in general is changing at each cycle, unless the **Use_Fixed_Timestep** parameter is set to 1, or the **-dt** command line parameter is set. If the **Output_Time_Specification** is 1, then output will occur by cycle number, starting at **Output_Cycle_Begin**, ending at **Output_Cycle_End**, with an interval of **Output_Cycle_Delta**. On the command line, the time specification is determined by which set of commands is given (**-otb**, **-ote**, **-otd** for time-determined, or **-ocb**, **-oce**, and **-ocd** for cycle-determined). It is best not to mix these sets of command line options.

GASpAR supports, ostensibly, two different binary output formats. One is based on the Hierarchical Data Format-5 (HDF5; see URL http::/hdf.ncsa.uiuc.edu). This format has not been fully tested, so it has not been fully integrated into the code. The second format is the GASpAR-binary, or *GBin*, format. *GBin* is a fully self-describing platform-independent binary format, whose data is specified by offsets within the file. It, like the HDF5 package, allows for parallel output using MPI-IO, and will revert to POSIX calls if MPI-IO is unavailable. The construction API for *GBin* is similar to that for the HDF5 package, so that the object instantiation is the same. *GBin* retains an HDF3-like API for setting data dimensions, grids, tags and metadata, and can accommodate all data types used in GASpAR, with no repetition of data. *GBin* files (including the dump files) can be profiled with the utility, *ginfo* (see Chapter 4).

Currently, only the *GBin* format is selectable, but the HDF format will be available soon. Restart (dump) files will continue to use the *GBin* format, however.

At present, a full printed explanation of the *GBin* file format is not available. However, one will be published in a future document. In the meantime, you can go directly to the sources ("src/io/gbin_writer.cpp" or "src/io/gbin_reader.cpp") if you want to see the details.

Output may be examined using a variety of MATLAB scripts that have been provided with the distribution. The script "src/matlab/gbin_input.m", with its components "src/matlab/gbin_fileinfo" and "src/matlab/gbin_dsread", in addition to being required to ingest *GBin* data may also be useful for eliciting the file format. The script "src/matlab/meshelem.m" is arguably the most useful analysis script in that it will perform visualization of *GBin* datasets. These and other important MATLAB scripts are described in Chapter 5.

Chapter 3

Setting up your own problem

We have attempted to make problem setup as painless as possible. In this chapter we provide in some detail some of the procedures used in setting up and running a problem from scratch. The code is designed so that all problem set-up, and in fact, all user-defined quantities, reside in a single file, called a *user file*. In Sec. §2.2 we saw that this file is, by default, called, "guser.cpp", with a header called "guser.h". The user file header contains all of the data that the user wants to globalize, and make available to other parts of the code. It is also a good place to pre-define user-defined functions that may also reside in the user file.

A user file *must* contain several functions (methods) that are recognized by the system. Comprising this section is a list of theses methods. Perhaps the best way to familiarize yourself with these is to look at them in the context of the examples described in Sec. 2.2. In the following subsections, we provide detail on each of the required functions, and what they can do for you.

3.1 GUserConfig

The **GUserConfig** method in "guser.cpp" is the method where system configuration parameters are set. The method is called in the **GInit** method after the parameter file and command line parameters have been parsed. It

is intended to be the place where variables representing the initialization of system-wide memory or solver characteristics can be set. For example, if the advection-diffusion solver is being used, one might set

```
bLinAdvection_ = TRUE;
nEvolvedFields_ = 1;
```

in order to indicate that linear advection is to be done, and that the number of fields that the user file will initialize, and that GASpAR will evolve, is 1.

3.2 GUserInit

This method is where the problem is initialized. This is called after all variables and the solver(s) are created. This method is *not* called if the problem is a restart (**-r** or **-rf** command line option specified, or **Do_Restart** file parameter is set to 1. In this method, the problem time initialization should be specified. This includes adaption at $t = 0$, if required. In Chapter 2, we described a number of user files that show a variety of types of initialization. Many of the methods that are provided in these files are not the most efficient way to perform the initialization. This was done not only for expediency, but also to present the variety of access methods that are allowed. For efficiency, however, intialization or restart should adhere to good vectorization methods; the access methods exist to insure this.

Also, user-defined parameters or variables may be defined here. By "user-defined" it is meant that the system will manage these variables. That is, the system will output them together with the system quantities, and if adaption is done, the system will take care of interpolating these quantities to their new grids. Note that if user-defined parameters/variables are defined in this method, then, in general, they won't be defined upon restart. So, the user is encouraged to put definitions of new variables in a user-defined method (function). The name of the function must appear in the "guser.h" header file, and all user-defined variables must be globalized by placing them also in this same header file. An example is of this technique is provided presently.

In order to specify user-defined variables, we use a call(s) to the method

GRegisterUserFields. For example, we declare the fields for vorticity and energy in "guser.h" together with some other data:

```
#if defined(_GASPAR_H_INIT)
GBOOL bSkipTime;
GINT iSkipCycle;
GDOUBLE kx;
GDOUBLE ky;
GDOUBLE sigma;
GDOUBLE KE;
GDOUBLE Enstrophy;
GFieldList vorticity;
GFieldList energy;
#else
extern GBOOL bSkipTime;
extern GINT iSkipCycle;
extern GDOUBLE kx;
extern GDOUBLE ky;
extern GDOUBLE sigma;
extern GDOUBLE KE;
extern GDOUBLE Enstrophy;
extern GFieldList vorticity;
extern GFieldList energy;
#endif
void MyInit();
```

Then in the function **MyInit** we make the following call:

```
if ( GRegisterUserFields(2, &vorticity, "Zeta", &energy, "KE")
== 0 )
cout << "MyInit: user field registration failed" ¡¡ endl;
exit(1);
```

The user may choose to make a single call to **GRegisterUserFields** with all the user-defined variables to register, or he can split the registration up

over several calls to the function. Each variable registered must contain a pointer to the variable (*e.g.* "&vorticity") and a user-provided name (*e.g.* , "Zeta"), in that order. In this example, then, the function **MyInit** would be called from both **GUserInit** and **GUserStart** so that the user-defined variables are available both at initialization and restart.

The user is encouraged not to modify the

#if defined(_GASPAR_H_INIT)
#else
#endif

preprocessor definition in the "guser.h" file. This construct is used in the main driver so that data is not multiply defined.

3.3 GUserStart

If the run is a restart, then **GUserInit** is not called, but **GUserStart** is. For example, if there are time-dependent boundary conditions that must be computed, then these can be computed initially with a call to a function from withing **GUserStart**, and can be computed subsequently with a call to this same function from within **GUserTimeDep**.

3.4 GUserTimeDep

This method is intended to compute *anything* the user wants to compute at *each* timestep. It is called after grid adaption (if any), and *before* a solver step; hence, *before* the time is updated. As mentioned above, time-dependent boundary conditions may be computed in this function, but also, any user-defined time dependent quantity can also be computed here (*e.g.* , vorticity, point-wise kinetic energy). User-defined quantities that do not need to be recomputed at every timestep should not be placed in this method.

Since this method is called at each timestep, it is important to code it as

efficiently as possible. We have not done this in the above examples in all cases, mainly to illustrate the variety of access methods available. In general, this method should use the rules for vectorization as often as possible.

3.5 GUserLogConfig

GUserLogConfig provides the user with a place to configure static and dynamic components of the log file. The log file is intended to contain run parameters (static) and time-varying diagnostics (dynamic), a sort of quick-check of the run's progress. The log file is updated at a user-specified interval (command line parameter **-dcd** or file parameter **Logging_Cycle_Delta**). Any user-defined parameters that are to be added to the log file *must* be globalized by placing them in the "guser.h" file. The logging is handled by the class **GLogger**. Static and dynamic parameters are configured separately:

```
// Set static log data (doesn't change with time cycle)
bOk = glogger_.SetStaticParamDesc("%f %f %f %b %i", "kx"
                                                  , "ky"
                                                  , "sigma"
                                                  , "NewbSkipTime"
                                                  , "NewiSkipCycle");
bOk = bOk &
      glogger_.SetStaticData(5   , &kx, &ky   , &sigma
                                 , &bSkipTime, &iSkipCycle);
                             );


GBOOL bOk;
// Set dynamic log data (changes with time):
bOk = glogger_.SetDynamicParamDesc("%f %f"
                                  , "KE"
                                  , "Enstrophy"
                                   );
bOk = bOk &
      glogger_.SetDynamicData(3  , &KE, &Enstrophy
                                );
```

Here, we assume that the user-defined variables are defined in the "guser.h" file, as in §3.2.

The parameter description setups have the same format: specify a format for the parameter, and then provide a text description of the variable. The specification of the data (variables) are also the same: indicate how many variables you're setting, and then give the pointers to each of them. In the call to the method **SetStatic(Dynamic)Data**, the variables must agree in type and number to those given in the **SetStatic(Dynamic)ParamDesc** call. It is because the **SetStatic(Dynamic)Data** methods take pointers to the variables that are used in the system that the variables must be globalized by setting them in the "guser.h" file. The user can set as many or as few parameters as desired in the file, and can split them over several calls to the **Set** methods. The order in which they appear in the log file, however, is the order in which they are set.

In this example, the static data section of the log file contains, in addition to the data set by the system, the user-defined quantities labeled "kx", "ky", "sigma", "NewbSkipTime", and "NewiSkipCycle". The system dynamic parameters will be augmented by the user-defined quantities "KE" and "Enstrophy" in that order. All time-dependent user-defined variables can be updated in the call to **GUserLogUpdate**. This is distinguished from **GUserTimeDep** simply because the user is free to specify the log cycle interval, so logging may not be done at every time cycle.

The allowed format specifiers are b, i, d, l, f, and s, representing, respectively, the system-defined data types, GBOOL, GINT, GINT, GLONG, GDOUBLE, and char *. There is currently no format specifier for GFLOAT, so all user-defined floats are expected to to be of type GDOUBLE.

The list of static and dynamic system parameters that are provided in the log file is given in the method **GInitLogger**, in the file "src/exec/gaspar_t.cpp".

3.6 GUserLogUpdate

This method is intended to be a place where the user can compute the user-defined variables that have been set to appear in the log file. It is called at the end of a timestep, after any data is output, and before the method that creates a restart file. It is only called at a user-defined cycle interval (see §3.5). All quantities computed in **GUserLogUpdate** must have been globalized by placing them in the "guser.h" file.

3.7 GUserTerm

The **GUserTerm** simply provides a place where the user may perform some activity after time stepping is complete. For example, the user might wish to compute a terminal variable, such as cpu time, and write this to a user-defined log file.

3.8 Systems that have been tested

In the following table 3.1 we present a list of characteristics of some of the systems that GASpAR has been run on. This is not a complete list, as various component tests have been done on a variety of other systems. The idea is to make the code as portable as possible. If you run the code on a system that is not described here, we would appreciate it if you would let us know, so that we can update this list. [Note that PGI refers to the Portland Group compiler suite.]

Table 3.1: List of some of the systems that GASpAR has been tested on

Description	OS	Compiler(s)
Linux single-cpu, x_86	Fedora Core 1	gcc2.9.6 PGI 5.2
Linux multi-cpu, x_86	Fedora Core 3	PGI 5.2 PGI 6
IBM SP	AIX	Visual Age 6.0
Linux SMP, EMT64	Linux	PGI 5.2 PGI 6

Chapter 4

Command line utilities

There are a couple of off-line utilities that have been provided to produce grids and inspect output data. These utilities are provided in the distribution under the

./bin

subdirectory. To make these utilities, simply enter the bin directory, and type

> **make all**

Each of these utilities is described, in turn, here.

4.1 Grid generation with *gdd*

This utility was introduced in section 2.4. This mesh generator is a basic rectangular mesh generation utility that creates conforming meshes from a specified global rectangular domain, and labels nodal points if directed to do so. Boundary indices are also provided; these index the labeled nodes. Boundary

condition types may be specified node-by-node using *gdd*. There are two formats into which the generator will put the mesh data: one sorts the element data (including nodes and boundary data) for each processor; the other sorts the data by element id. Currently, the elements are assigned to processors by an even distribution method, but hooks have been placed in the code to access a standard interface routine, so that other packages, such as Chaco (see http://www.cs.sandia.gov/%7Eweb9200/9200_download.html), can be used by *gdd* as dynamic libraries to carry out the element distribution among processors. This work is on-going.

The *gdd* utility is designed primarily as an interactive stand-alone program for generating grids, which can then be used by GASpAR with the **-m** command line option. The options that *gdd* understands are given by the following, which is the output obtained by issuing the command

> **gdd -h**

```
Enter:

 ../gdd [-d dim] [{-Nx}{-Ny}{-Nz} #Elems] [{-xNx}{-xNy}{-xNz} Exp. Order]
        [-P0 x0 {y0} {z0}] [-P1 x1 {y1} {z1}] [-b Face# Type] [-Np #Procs]
        [-o Output_format] [-l filename] [-f filename] [-quiet] [-h]

 Where:
      -d dim          : problem dimensionality (1, 2, or 3). Default is 2.
      -Nx #Elems      : Number of elements in x-direction. Default is 1.
      -Ny #Elems      : Number of elements in y-direction. Default is 1.
      -Nz #Elems      : Number of elements in z-direction. Default is 1.
      -xNx Exp. Order : Expansion order in x-direction. Default is 4.
      -xNy Exp. Order : Expansion order in y-direction. Default is 4.
      -xNz Exp. Order : Expansion order in z-direction. Default is 4.
      -P0 x0 {y0 {z0}}: Bottom left corner. There must be dim of these. Default is (0,0,0).
      -P1 x0 {y0 {z0}}: Top right diagonal corner. There must be dim of these. Default is (1,1,1)
      -b   Seg Type   : Set boundary condition Type on face (edge, point) given by Seg.
                        Seg is given in 1d by X0; in 2d by segment endpoints X0 Y0 X1 Y1;
                        in 3d by diag corners X0 Y0 Z0 X1 Y1 Z1. Type is one of: 0 (Dirichlet),
                        1 (Neumann), 2 (Periodic), 3 (No-slip), 4 (None). Default is 4.
      -Np  #Procs     : Number of processors. Default is 1.
      -cb  N Type     : Set common bdy node, N, to type, where Type is one of: 0 (Dirichlet),
                        1 (Nuemann), 2 (Periodic), 3 (No-slip), 4 (None). No default.
                        Corners, N, are s.t. 0=>bottom left; 1=>bottom right; 2=>top right; 3=>top left in 2d.
      -o  Output form.: Output format: By_Proc==>sort by proc id; By_Elem==>sort by element id.
                        Default is By_Proc.
      -l filename     : Partition library file name. Default is NULL.
      -f filename     : Output file name. Default is mesh.dat.
      -quiet          : run without echoing parameters. Default is full echo.
      -(no)nodes      : (do not) print nodal information to file. Default is -nodes.
```

```
      -h              : Print this help list.


  Currently, only rectangular domains are supported. In the
future, a GUI will be added to decompose grids and provide
mesh information for general domains.
  Output data is placed in the 'filename' file, or in 'mesh.dat'
if no filename is provided. All data for constructing the finite
element-based mesh on a multi-processor system is supplied. By default,
a simple cyclic distribution of the elements among the processors is
performed. If a shared library (SO) is specified, then the processor
distribution is performed by that library. The SO must contain the
entry point 'GDDInterface', whose format is given in gdd.h
```

4.2 Data profiling with *ginfo*

The profiling utility, *ginfo*, simply provides at-a-glance information on a *GBin* file. Summary information only is provided. This includes the metadata, number of datasets, and for each dataset, the label, rank, dimensions, tag data, basic geometry information, if provided, and data max/min. The call is simply

> ginfo file1 file2 file3 ...

The following is a sample output from *ginfo* acting on a dump file:

```
File summary for 'gaspar.dmp':

File descriptor      : \gaspar_DUMP_FILE
Endian swapped?      : 0
Number of datasets   : 3
Meta Data:     (0): 4400
    (1): 13
    (2): 0
    (3): 13
    (4): 100
    (5): 0
    (6): 0
    (7): 100
    (8): 10
    (9): 1e-09
    (10): 4.4e-06
    (11): 1e-05
    (12): 0
```

```
    (13): 1e-05
    (14): 5
    (15): 0
    (16): 0
    (17): 0
    (18): 1
    (19): 1
    (20): 2
    (21): 0
    (22): 0.025
    (23): 0.025
    (24): 1
    (25): 0
    (26): 0
    (27): 0
    (28): 0

.....................................Dataset: 0:

        Label         : V1
        Rank          : 2
        Dims          : (13, 13 )
        Tags          :                (0): 4.4e-06
             (1): 4400
        No. Verts     : 4
        Elem_Type     : 1
        Elem_Vert     : (-0.5, -0.5) (0, -0.5) (0, 0) (-0.5, 0)
        Coord dims    : (169,169)
        Data (Max,Min): (2.6191, -0.6191)

.....................................Dataset: 1:

        Label         : V2
        Rank          : 2
        Dims          : (13, 13 )
        Tags          :                (0): 4.4e-06
             (1): 4400
        No. Verts     : 4
        Elem_Type     : 1
        Elem_Vert     : (-0.5, -0.5) (0, -0.5) (0, 0) (-0.5, 0)
        Coord dims    : (169,169)
        Data (Max,Min): (0.248344, -1.62852e-11)

.....................................Dataset: 2:

        Label         : Pr
        Rank          : 2
        Dims          : (11, 11 )
        Tags          :                (0): 4.4e-06
             (1): 4400
        No. Verts     : 4
        Elem_Type     : 1
        Elem_Vert     : (-0.5, -0.5) (0, -0.5) (0, 0) (-0.5, 0)
        Coord dims    : (121,121)
        Data (Max,Min): (-0.0910632, -0.882867)
```

Chapter 5

Manipulating output: MATLAB utilities

The "src/matlab" subdirectory of the distribution contains a variety of MATLAB scripts for use in manipulating **GBin** data. Many of these are esoteric, but may nevertheless provide some useful capability. Some of them are indispensable or otherwise very useful. It is the scripts from this latter category that we focus on in this chapter.

The goal is to present in one location a comprehensive list of these utilities that the user may refer to. The user is always free to type

>> **help <script_name>**

at the MATLAB command line to see the description of the utility. We do not look in detail at which other MATLAB functions may be called from the script interfaces.

5.1 gbin_input

`Usage:`

```
[fileinfo elems u]=gbin_input(filename);

Description:

Given input GBin type filename, return
fileinfo (structure for file information),
elems (structure for element mesh) and
u (structure for scientific data on elems).
To view a structure, e.g., "u", type "u<Enter>".
Try "help meshelem" to view these data.
```

5.2 meshelem

```
Usage:

h = meshelem(elems,u,ifig,'lab',lw);

Description:

Given structures elems & u (e.g., from gbin_input),
create a mesh plot in figure ifig (default 1), of variable
labeled 'u.lab' (default 'u.V1').
Return graphics objects handle structure h.
h.surf contains a patch-object handle, h.text is a
element-no. text-handle vector, and h.edge(1:4) are element-edge handles,
to lines of width lw (default 3).
Enter set(h.text,'Visible','on') etc. to make objects visible.
```

5.3 plotlogf

```
Usage:

[h A labs]=plotlogf('fname',icols,jcols,'pars',pltflag,nbackup,llist);
 h     =plotlogf({A labs},icols,jcols,'pars',pltflag,nbackup,llist);
```

```
Description:

Read data matrix A from file 'fname' and return handle h from
eval(['h=plot(A(:,icols),A(:,jcols)' 'pars' ');']) and cell
array labs of legend strings.  In 'fname'
lines starting with '#' are ignored, lines starting with ' ',
'!*' or a digit are read for equal numbers of numeric data columns.

Defaults:
   'fname'    'test_log.txt'
   icols      2               time column
   jcols      9               dU1/dx_max
   'pars'     []              e.g., ','r--''' for red dashed
   pltflag    true            do plot
   nbackup    1               backup to labels line before data
   llist      '1:ndl'         line number range to read

If 'fname' contains a '*' (no directory path allowed) then
loop over the wildcard-expanded file list, and automatically
assign pars for each file and each data column.

Once A & labs have been returned you can enter them to avoid rereading fname.
```

5.4 biopelem

```
Usage:

w=biopelem(u,v,op,'labu','labv','labw');

Description:

Assign to w field 'labw' (default 'labuoplabv') a binary operation on
the u & v fields 'labu' and 'labv'.  The op can be a string
'plus', 'minus', 'times', 'rdivide' etc. (do "help ops" for examples)
or more generally an expression like 'x/y-1' (do "help inline" for
```

examples). In the latter case 'labw' must be an acceptable field name.

5.5 unopelem

Usage:

```
w=unopelem(elems,u,'op(x,u)','labu','labw');
```

Description:

Assign to w field 'labw' a unary operation on the u field 'labu'.
The 'op(x,u)' can be an expression like 'sin(u)+x(:,:,1)-x(:,:,2)'.
Note 'op(x,u)' uses 'u' for any 'labu',
and 'labw' must be an acceptable field name.

5.6 manycall

Usage:

```
h=manycall(filepat);
h=manycall(filepat,call);
F=manycall(filepat,call,domovie);
F=manycall({fileinfo elems u},call,domovie);
```

Description:

Given filepat, a string with a file specification
pattern (including path and * wildcards) like 'abc*xyz',
execute [fileinfo elems u]=gbin_input(filename); for every
filename (labeled j) that matches filepat, and execute
eval([call{1} 'elems,u,j,' call{2}]); for each j.

If nargin<2 or call==[] use the default call={'h(j)=meshelem('
''''V1'');view([40 60]),axis([0 1 0 1 b])'};

```
where b contains the ZMIN ZMAX of the 1st call.

If filepat is a structure than it is of mkfnames output type.

If the 1st argument is a cell then loop over its columns' contents
instead of using gbin_input.

Use fig_flip(1:Nfigs,s) to cycle all figs at s seconds each.

With "F=" variant (domovie a 3rd argin), save memory by
overwriting figure j=1, but instead of h,
return an array F of movie frames from getframe
(that can be played using movie(1,F)),
and if domovie=1 print each figure to a .png file named 'fig_abc*xyz.png'
(with any subdirectory and 3-character extension info clipped)
for later conversion using e.g., the UNIX command

convert -delay 50 fig_abc*xyz.png abcxyz.mng

If domovie is a string then it names a .avi animation file and no
.png files are created.
```

5.7 ntrpelem

```
Usage:

zi=ntrpelem(elems,u,'lab',   xi,         yi,'method')
ui=ntrpelem(    x,y,    z,'lab',{elemsi ic},'method')

Description:

Interpolate from values u.lab on elems.x to zi on xi & yi, or
from values z on x & y to ui.lab on elemsi.x{ic}, using
'method' (default 'linear', accepts 'cubic', 'nearest', 'spline').
Pass xi & yi as vectors or matrices as in INTERP2.
The default ic is 1:elemsi.n but for staggered grids ic
```

```
should be specified, e.g., u.V1ic, u.Pic etc.
```

5.8 streelem

```
Usage:

[h fx fy xt yt] = streelem(elems,u,ifig,ilist,sx,sy,xt,yt,stepsize,maxverts);

Description:

Given structures elems & u (e.g., from gbin_input),
create streamline plot in figure window ifig,
for elements i=ilist (default 1:elems.n),
using (and updating) tickmarks (xt,yt) (default ([],[])),
with steps of length stepsize (default .1) and
at most maxverts (default 10000) vertices per streamline.
Start vertices for element i=1:length(ilist) are (sx(:,:,i),sy(:,:,i))
(defaults to uniform element-interior grid).
Label x-y axes by l2 errors in u-v.
Return h{1:length(ilist)}, the graphics objects handles,
and (fx{:,:,i},fy{:,:,i}), the streamline finish vertices.

Example: see Kovasznay.m.
```

5.9 specelem

```
Usage:

h = specelem(elems,u,ifig,lab);
h = specelem(elems,u,ifig,lab,invis);
h = specelem(elems,u,ifig,lab,invis,lw);

Description:
```

```
As in meshelem, given structures elems & u (e.g., from gbin_input),
create a spectrum plot in figure ifig, of variable
labeled lab; return graphics objects handles h{1:elems.n,1:6}.
The 1st column of h contains patch-object handles, column 2 are
element-no. handles, and columns 3:6 are element-edge handles,
to lines of width lw (default 3).
If included, invis is a list (e.g., 1 or 3:6 or [1 4:5] etc.)
of h columns to be invisible.
Enter "u<Enter>" to see the labels of u.
```

Chapter 6

I/O with *GBin*

In this chapter we consider a more administrative aspect of the GASpAR code in more detail. By *administrative* we distinguish the classes/functions from those which perform purely numerical computation, such as those in Chapter 8. Some administrative capabilities were introduced in Chapter 2. Here, we try to present somewhat less of an operational and more of a design-oriented perspective of I/O in GASpAR.

The main I/O format, *GBin*, used in GASpAR was introduced in section 2. Here, we expand on what was presented there. *GBin* actually consists of a reader and writer, **GBinWriter** and **GBinReader**, respectively. Each of these classes is derived from a *GBin* base class, called **GBinStream**, which handle opening and closing of *GBin* files, file integrity checks, common data offset calculations, and also provides some basic information about the file required by both the reader and writer. This class is, in turn, derived from a more basic I/O class, **GStream**, which handles the lower–level file operations, for both POSIX and MPI–IO libraries. Thus **GStream** forms a common interface to either serial or parallel I/O. The MPI–IO and actual POSIX calls are contained only in the **GStream** class, in order to localize them. If other parallel I/O message–passing libraries are required, they should be localizable in largely the same way. If the code is compiled with the MPI_IO_DEFAULT option, then MPI–IO is used; otherwise the POSIX calls are used.

The basic unit of the reader and writer is the *dataset*. All data that is set before writing applies to the current dataset under consideration until they are changed. For reading, a dataset id must be supplied in order to retrieve dataset information. In the next two subsections, we discuss the more prominent *GBin* classes, the **GBinWriter** and the **GBinReader**.

6.1 GBinWriter

In this section we present the public interface methods for the **GBinWriter** classes. **GBinWriter** accepts a variety of information required to define the dataset, before actually writing it. Most of this associated data is not required. If specified for any dataset, however, it remains in effect for all future datasets, unless it is changed explicitly. A final call to **WriteData** will write the dataset and its associated data to the file.

Unless otherwise indicated, the method is not required to be called.

GBinWriter(GBOOL isCollective=FALSE, GBOOL isIndependent=TRUE, GINT ioTaskID=0);

```
DESCRIPTION:  Required. Constructor.
ARGUMENTS   : isCollective : flag that tells if writer is to operate in collective mode,
                             meaning that all processors write collectively to the same file.
                             If FALSE, then the file used in the  GBinWriter::Open
                             call must be distinct for each processor.
              isIndependent: flag that tells if writer is to operate independently. If TRUE,
                             then host processor carries out the writes. If FALSE, then writer
                             will send all data to the task id specified by ioTaskID.
              ioTaskID     : specifies the processor id to which to send data if isIndependent=FALSE.
```

GBOOL Open(const char *filename, GIOS_MODE iomode);

```
DESCRIPTION:  Required. Opens file for writing. Should be the first call made.
ARGUMENTS   : filename      : name of file to which to write data.
              iomode        : open mode, one of: gios::in, gios::out, gios::inout, gios::app,
                              gios::binary, gios::ate ('at end'). Multiple modes can be specified
                              by using the '|' ('or' operator), e.g. gios::binary | gios::ate to
                              open as a binary file with file pointer at the end of the file.
                              Open automatically adds the gios::binary | gios::ate flags.
RETURNS     : TRUE on success; else FALSE.
```

void Close();

```
DESCRIPTION:  Required. Closes GBinStream to file, and frees up all data.
```

```
ARGUMENTS  : none.
RETURNS    : none.
```

GBOOL SetMeta(GINT nmeta, GDOUBLE *meta˙data, const char *descriptor);

```
DESCRIPTION: Sets file meta data.
ARGUMENTS  : nmeta     : number of elements in array meta_data.
             meta_data : meta data array, allocated by caller.
             descriptor: meta data description string.
RETURNS    : TRUE on success, else FALSE. Failure occurs if Open has not been called.
             If nmeta = 0 or meta_data = NULL, TRUE is returned (not considered an error).
```

GBOOL SetDims(GINT rank, GINT *dims);

```
DESCRIPTION: Required. Sets data dimensions.
ARGUMENTS  : rank : rank of dataset
             dims : array of length rank, giving the size of each dimension.
RETURNS    : TRUE on success; else FALSE. Failure occurs if Open has not been called,
             or the rank or dimensionality is invalid.
```

GBOOL SetVertices(const GINT num, GDOUBLE *vertices, ELEMTYPE etype);

```
DESCRIPTION: Sets the vertices for the element corresponding to this dataset.
ARGUMENTS  : num     : number of vertices.  This is checked against the rank.
                       There must be $ rank 2^ rank$ of these.
             vertices: array of vertices from the element.
             et      : element type id.
RETURNS    : TRUE on success; else FALSE. Failure occurs if Open has not been called, or
             if the number of vertices is inconsistent with dataset rank.
```

GBOOL SetCoord(GINT idir, GINT dim, GDOUBLE *coord, const char *descriptor);

```
DESCRIPTION: Sets coordinate information, if any.
ARGUMENTS  : idir      : dimension referred to by call. If called multiple times with the same
                         value, then data will be overwritten.
             dim       : size of the coordinate array.
             coord     : coordinate values.
             descriptor: coordinate description. May be NULL.
RETURNS    : TRUE on success; else FALSE. Failure occurs if Open has not been called, or
             if the idir is inconsistent with rank
```

WriteData(GINT rank, GINT *dims, GDOUBLE *data , GINT ntags, GDOUBLE *ftags, const char *descriptor, GFPOS *datablk);

```
DESCRIPTION: Required. Puts the data to the file.
ARGUMENTS  : rank      : rank of data set. Duplicate of that used in SetDims. May be deprecated.
             dims      : dims of data set. Duplicate of that used in SetDims. May be deprecated.
             data      : data to be written. Must be of length
                         ( dims[0] * dims[1] ... * dims[rank-1].
             ntags     : number of data descriptor tags.
             ftags     : array of data descriptor tags, of length ntags.
             descriptor: data descriptor.
             datablk   : (returned) offset of the dataset in the file.
RETURNS    : TRUE on success; else FALSE. Failure occurs if Open has not been called, or
             if any of the internal writes fails. Error number is set.
```

GINT GBinWriter::GetTotalWritten();

```
DESCRIPTION: Gets total number of data items written to file by  WriteData.
ARGUMENTS  : none.
RETURNS    : total number of items written.
```

const char *Error();

```
DESCRIPTION: Provides error string if the error condition has been set.
ARGUMENTS  : none.
RETURNS    : error string.
```

GINT ErrorID();

```
DESCRIPTION: provides error condition number if an error is encountered.
ARGUMENTS  : none.
RETURNS    : condition number
```

6.2 GBinReader

In this section we present the public interface methods for the **GBinReader** class. Unless otherwise indicated, the method is not required to be called.

The first set of methods apply to the entire file, or to the meta-data. The second set apply to individual datasets within the file.

GBinReader(GBOOL isCollective=FALSE, GBOOL isIndependent=TRUE, GINT ioTaskID=0);

```
DESCRIPTION:  Required. Constructor.
ARGUMENTS  : isCollective : flag that tells if writer is to operate in collective mode,
                            meaning that all processors write collectively to the same file.
                            If FALSE, then the file used in the  GBinWriter::Open
                            call must be distinct for each processor.
             isIndependent: flag that tells if reader is to operate independently. If TRUE,
                            then host processor carries out the reads. If FALSE, then reader
                            will receive all data from the task id specified by ioTaskID.
             ioTaskID     : specifies the processor id from which to receive data if isIndependent=FALSE.
```

GBOOL Open(const char *filename, GIOS_MODE iomode);

```
DESCRIPTION:  Required. Opens file for writing. Should be the first call made.
ARGUMENTS  : filename      : name of file to which to write data.
             iomode        : open mode, one of: gios::in, gios::out, gios::inout, gios::app,
```

```
                              gios::binary, gios::ate ('at end'). Multiple modes can be specified
                              by using the '|' ('or' operator), e.g. gios::binary | gios::ate to
                              open as a binary file with file pointer at the end of the file.
                              Open automatically adds the gios::binary flag.
RETURNS    : TRUE on success; else FALSE.
```

void Close();

```
DESCRIPTION:  Required. Closes GBinStream to file, and frees up all data.
ARGUMENTS  : none.
RETURNS    : none.
```

char *GetMetaDesc()

```
DESCRIPTION: Gets meta-data (file) descriptor.
ARGUMENTS  : none.
RETURNS    : char * to descriptor string.
```

GINT GetNumMeta();

```
DESCRIPTION: Gets the number of meta-data items
ARGUMENTS  : none.
RETURNS    : integer number of items. May be 0.
```

GDOUBLE *GetMeta();

```
DESCRIPTION: Gets a pointer to the meta-data itself.
ARGUMENTS  : none.
RETURNS    : GDOUBLE pointer to meta-data.
```

GINT GetNumDataSets();

```
DESCRIPTION: Gets the total number of dataset within the file.
ARGUMENTS  : none.
RETURNS    : GINT number of datasets. May be 0.
```

const char *Error();

```
DESCRIPTION: Provides error string if the error condition has been set.
ARGUMENTS  : none.
RETURNS    : error string.
```

GINT ErrorID();

```
DESCRIPTION: provides error condition number if an error is encountered.
ARGUMENTS  : none.
RETURNS    : condition number
```

The following **GBinReader** methods apply to individual datasets. The

datasets are referenced by an id (or by a label). The dataset id is an GINT integer in the range 0 to **GBinReader::GetNumDataSets()** - 1.

const char *GetLabel(GINT idataset);

```
DESCRIPTION: Gets label to dataset, if there is one.
ARGUMENTS  : idataset: dataset id.
RETURNS    : char * to label; else NULL.
```

GINT GetNumTags(GINT idataset);

```
DESCRIPTION: Gets the number of data tags associated with dataset.
ARGUMENTS  : idataset: dataset id.
RETURNS    :
```

GDOUBLE *GetTags(GINT idataset);

DESCRIPTION: Gets the data tags associated with dataset
ARGUMENTS : idataset: dataset id.
RETURNS : GDOUBLE pointer to tag data. Array is of size **GetNumTags()**

GINT GetRank(GINT idataset);

DESCRIPTION: Gets dataset rank
ARGUMENTS : idataset: dataset id.
RETURNS : GINT *rank*

GINT GetDims(GINT idataset, GINT idir);
GINT *GetDims(GINT idataset);

DESCRIPTION: Gets the dimensions of the data.
ARGUMENTS : idataset: dataset id.
idir : coordinate direction of interest.
RETURNS : First method gives the size of data in *idir* coordinate direction;
second method gives an array of size *rank*, containing each of the
coordinate dimensions. Error if NULL.

GINT GetCoordDims(GINT idataset, GINT idir);

DESCRIPTION: Gets the dimensions of the coordinate associated with the dataset.
ARGUMENTS : idataset: dataset id.
idir : coordinate direction of interest.
RETURNS : dimension of the *idir* coordinate. May be 0.

GINT GetNumVert(GINT idataset);

DESCRIPTION: Gets the number of vertices associated with dataset
ARGUMENTS : idataset: dataset id.
RETURNS : Number of vertices. May be 0.

Point3D *GetVert(GINT idataset);

DESCRIPTION: Gets vertices associated with dataset
ARGUMENTS : idataset: dataset id.
RETURNS : Point3D array of length **GetNumVert()** containing the vertices.

```
            If NULL, and GetNumVert() is nonzero, then there is an error.
```

GDOUBLE *GetGridData (GINT idataset, GINT idir, GDOUBLE *x, GINT n);

```
DESCRIPTION: Gets the coordinate grid data for direction idir.
ARGUMENTS  : idataset: dataset id.
             idir    : coordinate direction of interest.
             x       : pointer to array of coordinate values.
             n       : number of coordinate values.
RETURNS    : On success, returns x pointer; else returns NULL.
```

GDOUBLE *GetFieldData(GINT idataset, GDOUBLE *data, GINT n);
GDOUBLE *GetFieldData(const char *label, GINT ids˙start, GDOUBLE *data, GINT n);

```
DESCRIPTION: Retrieves the dataset data. The first method retrieves data based on
             dataset id. The second retrieves data based on the dataset label (if any),
             and begins searching starting from, and including, the specified
             dataset id.
ARGUMENTS  : idataset  : dataset id.
             (label    : dataset label)
             (ids_start: tells reader to start searching from this dataset id.a)
             data      : pointer to retrieved data.
             n         : total number of GDOUBLE values in data.
RETURNS    : On success, returns data pointer; else returns NULL.
```

6.2.1 Some examples

GBinWriter example

The following is an code fragment that instantiates a writer object, and performs a parallel file write:

```
GINT      idims[2], rank;
GDOUBLE   meta [3], u[100], x[10], y[10];

GBinWriter wgbin(TRUE,TRUE,0);                // Create writer in collective mode, do independent writes,
                                              // Since independent, last parameter irrelevant.

if ( !wgbin.Open("foo.dat", FALSE) ) exit(1); // Opens 'foo,dat', checks for valid file type
                                              // Appends if it exists; else creates a new file.
                                              // FALSE says not to truncate if file exists.

meta[0] = (GDOUBLE)icycle_;
meta[1] = time_;
meta[2] = dt_;
wgbin.SetMeta(3, meta, "FOO_TEST");           // Set meta-data

rank     = 2;
```

```
idims[0] = 10;
idims[1] = 10;

//  fill u, x, y, ...

wgbin.SetDims(rank, idims) )           ;       // rank, Set dimensions
wgbin.SetCoord(1, idims[0], x, "X1") ) ;       // Set 1-coordinate
wgbin.SetCoord(2, idims[1], y, "X2") ) ;       // Set 2-coordinate
wgbin.WriteData(rank, idims, u, 0, NULL,
                  "Field_Data", NULL) );       // Write dataset to file

wgbin.Close();                                 // Close file
```

GBinReader example

The following is an code fragment that instantiates a reader object, and performs a parallel file read:

```
GINT       j, n, NN, *idims, rank;
GDOUBLE    pmax=0.0;
GVector    p();
char       label[80];
GBinReader rgbin(TRUE,TRUE,0);                 // Create reader in collective mode, do independent reads,
                                               // Since independent, last parameter irrelevant.

if ( !rgbin.Open("foo.dat") ) exit(1);         // Opens 'foo,dat', checks for validity, reads header.

for ( n=0; n<rgbin.GetNumDataSets(); n++ ) {   // Loop over all datasets in file.
                                               // If dataset label is not "Pressure", go to next one.
  if ( strcmp(rgbin.Getlabel(n), "Pressure") != 0 ) continue;
  rank  = rgbin.GetRank(n);                    // Get dataset rank.
  idims = rgbin.GetDims(n);                    // Get dataset dimensions.
  for ( j=0,NN=1; j<rank; j++ ) NN *= idims[j];
  p.Resize(NN);
                                               // Get the field data for this dataset.
  if ( rgbin.GetFieldData(n, p,Data(), NN,) == NULL ) {
    cout << "Reader error: " << rgbin.Error() << endl;
    exit(1);
  }
  pmax = MAX( umax, fabs(p.Max()) );
}
rgbin.Close();                                 // Close file

cout << "Max Pressure = " << pmax << endl;
```

Chapter 7

GASpAR Utility classes and functions

We continue our consideration of GASpAR administrative capabilities in this chapter by highlighting some classes or functions that are particularly useful when writing solvers or setting up a problem. This list is hardly complete, as there are many "utility" features in the code that are not described here. Most of these features, however, require a lot of additional explanation that is not relevant for a user, whereas the classes and function presented here may well be.

In no particular order we present...

7.1 Linked lists for managing fields and other things

There are a number of linked–list classes that help to manage lists of objects: the **GElemList()** manages lists of elements, and the **GFieldList()** manages lists of field objects, and the class **GTList** is a template list class that manages lists of vectors, and various buffers. Typically, linked–lists are used in the upper level tie (driver)–code, or in the upper–level equation solvers (e.g. **StokesSolver**). It is easy to navigate a linked–list ("list"), which can be

done by indices or in other ways. It is also easy to remove lists elements or add to the list. The lists are important for managing quantities on a dynamic grid.

The user must remember that there is both a random–access feature to all lists, and a list–directed (ordered) access feature. The random–access feature is called by specifying a list-member's id, while the ordered access is achieved by calls without an id, so that the current member is used. The following are examples illustrating the use of both types of access.

This fragment uses random access to access all members of the element list:

```
RectQuad2D *elem;

for ( i=0; i<my_element_list->size(); i++ ) {
  elem = (*my_element_list)[i];  // or my_element_list->member(i);
  {do something with element...}
}
```

This fragment uses ordered–access for the same thing:

```
RectQuad2D *elem;

my_element_list->start(NULL);
for ( i=0; i<my_element_list->size(); i++ ) {
  elem = my_element_list->member();
  {do something with element...}
  my_element_list->next();
}
```

Note that with ordered–access, we must initialize the list with a call to **start(NULL)** (**NULL** indicates the beginning of the list; a pointer to any list member can be specified), and continue, in order, to the next element pointer with a call to **next()**.

7.2 ParamReader

The **ParamReader** was introduced in Chapter 2, mainly as a way of initializing system–wide parameters. However, it can be configured easily to be used anywhere that external data is required or desired (e.g., in **guser.cpp**).

ParamReader is derived from **GStream**, so it has a constructor similar to that of **GBinReader** and **GBinWriter**. The user–API is very simple:

ParamReader(GBOOL isCollective=FALSE, GBOOL isIndependent=TRUE, GINT ioTaskID=0);

```
DESCRIPTION:  Required. Constructor.
ARGUMENTS   : isCollective : flag that tells if writer is to operate in collective mode,
                             meaning that all processors write collectively to the same file.
                             If FALSE, then the file used in the  GBinWriter::Open
                             call must be distinct for each processor.
              isIndependent: flag that tells if reader is to operate independently. If TRUE,
                             then host processor carries out the reads. If FALSE, then reader
                             will receive all data from the task id specified by ioTaskID.
              ioTaskID     : specifies the processor id from which to receive data if isIndependent=FALSE.
```

void SetParams(char *format, ...);

```
DESCRIPTION: Sets the parameter names which may exist in the file.
ARGUMENTS   : format : printf--like format specification string. This string
                       establishes the data-types for the parameter (names) that follow.
                       Valid data types are: %i, %l (GINT), %f (GDOUBLE),  %s (string).

              ...    : comma--separated list of parameter-name strings. The parameter
                       in the file that corresponds to this name will be interpreted as
                       having a data type corresponding to that in the position of the name in the
                       format string. All parameters must have a data type.
RETURNS     : none.
```

GINT GetParams(char *filename, char *blk˙name, ...);

```
DESCRIPTION: Gets the parameter data from the specified block in the
             the specified file.
ARGUMENTS   : filename: name of parameter file
              blk_name: name of parameter block within file. All parameters must reside
                        within a named block.
              ...     : variable--list argument, one argument for each of the parameter
                        labels specified in  SetParams. Addresses of arguments only
                        must be passed, and they should be of the data type specified
                        in the format string used in the call to  SetParams.
RETURNS     : TRUE on success; else FALSE;  ParamReader::Error() gives
              an error string.
```

void SetBuffSize(GINT n);

```
DESCRIPTION: Sets the size of the read--buffer for this configuration.
ARGUMENTS   : n: Maximum number of bytes that can be read into buffer.
RETURNS     : none.
```

As an example, we configure a parameter file with 4 parameters:

```
  GINT        ion_state;
  GDOUBLE     time, density_0, temperature_0;
  char        finit[80];
  ParamReader MyInit(TRUE, TRUE, 0);

  strcpy(finit,"helium.dat");
  MyInit.SetParams("%i %f %f %f %s"
                                   , "Ionization_State"
                                   , "Time"
                                   , "Initial_Density"
                                   , "Initial_Temperature"
                                   , "Density_file");
```

Then, we parse the parameter file and place any parameters found into the appropriate variables:

```
  if ( !MyInit.GetParams("user.dat", "HELIUM_DATA"
                                                 , &ion_state
                                                 , &time
                                                 , &density_0
                                                 , &temperature_0;
                                                 , finit ) )
    cout << "Initialization failed: error: " << MyInit.Error() << endl;
    exit(1);
```

Note that the variable names in the call to **GetParams** must match the order in which they are issued parameter names in the call to **SetParams**. The data file for this configuration, stored in the file "user.dat", might look like

```
HELIUM_DATA
{
//_____________________________________________________________________________
//_____________________________________________________________________________
  Time               : 0.0   ; // In seconds
  Initial_Temperature: 1.0e4 ; // In Kelvin
  Initial_Density    : 1.0e-8; // In g/cm^3
  Ionization_State   : 1     ; // 0->no ionization, ...
}
```

In this example, there is no parameter data for the "Density_file" parameter; therefore, its value is unchanged after the **GetParams** call. Also, note that the order of the parameters within the file does not matter. There is no rule

for constructing parameter "names"; in reality, these are parameter "string" descriptions. However, they must appear in the file *exactly* as they appear in the **::SetParams** call. The file structure and the comments (everything after a "//") were already discussed in section 2.3.

There is limited ability within the **ParamReader** class to alter the file structure. For example, the block delimiters, end-of-line characters, parameter name delimiter, and comment characters may all be changed on construction, by providing "Set" methods for these quantities in the class specification; the member data corresponding to these quantities already exists. Another possible (and easy) modification would provide a "Set" method (or constructor) that tells the reader to make the parameter names (strings) case-insensitive.

7.3 MeshReader

The **MeshReader** was introduced in section section 2.4. Here we present the public–access methods provided by this class. At a future date, we will provide the file format(s) that **MeshReader** can read.

MeshReader(GINT maxbuff=65536)

```
DESCRIPTION:  Required. Constructor.
ARGUMENTS   : maxbuff : sets the maximum buffer size that class can handle.
                        Larger meshes will require a larger buffer size.
```

GBOOL Open(const char *filename);

```
DESCRIPTION: Opens mesh file for reading
ARGUMENTS   : filename: name of mesh file.
RETURNS     : TRUE on success; else FALSE.
```

void Close();

```
DESCRIPTION: closes mesh file stream
ARGUMENTS   : none.
RETURNS     : none.
```

void SetProc(GINT id);

```
DESCRIPTION: Sets processor id for subsequent reads. Only data
             designated in file for this processor will be retrieved.
ARGUMENTS   : id: processor id.
RETURNS     : none.
```

GINT GetNumRetrieved();

```
DESCRIPTION: Gets number of element blocks retrieved
ARGUMENTS  : none.
RETURNS    : integer number of elements
```

GINT GetDynRange();

```
DESCRIPTION: If nodes are listed, gives the maximum node id.
ARGUMENTS  : none.
RETURNS    : integer dynamic range
```

GINT GetNumProcs();

```
DESCRIPTION: Gets total number of processors that have data represented in file.
ARGUMENTS  : none.
RETURNS    : integer number of processor
```

GINT GetDim();

```
DESCRIPTION: Gets data dimensionality or rank.
ARGUMENTS  : none.
RETURNS    : integer rank
```

**GINT GetElem(ELEMTYPE &etype, Point3D *&vertex, GINT *&xN,
GIBuffer *&node˙ids, GIBuffer *&bdy˙nodes, GBTBuffer *&bct);**

```
DESCRIPTION: Gets data for the next element in the file corresponding to the
             processor set in SetProc. For allocated quantities, caller is
             responsible for deleting.
ARGUMENTS  : etype    : element type, of type ELEMTYPE.
             vertex   : array of length $2^it rank$ containing vertices. Allocated in reader
             xN       : array of length rank containing expansion order in each
                        coordinate direction. Allocated.
             node_ids : node ids (if in file) for each of (xN[0]*xN[1]...*xN[rank-1]) nodes
                        in element. Allocated.
             bdy_nodes: indices into the node˙ids array of the boundary nodes. Allocated.
             bct      : for each of the bdy˙nodes indices, the boundary condition type.
                        Allocated.
RETURNS    : 0 on success; else error id is set. User may retrieve error condition using
             ErrorID(), and error string using Error().
```

const char *Error();

```
DESCRIPTION: Provides error string if the error condition has been set.
ARGUMENTS  : none.
RETURNS    : error string.
```

GINT ErrorID();

```
DESCRIPTION: provides error condition number if an error is encountered.
ARGUMENTS  : none.
RETURNS    : condition number.
```

7.4 MTK

The **MTK** (for **M**ath **T**ool**K**it) is a namespace that includes all of the basic math operations allowed for vectors and matrices. These are set off from the vector and matrix methods so that they can be "tuned" in a platform–dependent way, in order to achieve better performance. Since most of a time cycle is spent in the iterative solver doing operator–vector products, this separation of the underlying operations seems beneficial. In fact, all math operations in **GTVector** and **GTMatrix** are carried out using the **MTK** functions.

The routines that actually carry out the operations in the **MTK** are written in Fortran, and a wrapper provides the C-Fortran interface. The decision was made to write the basic linear algebra operations in Fortran because it was found early on that there was some difficulty in vectorizing the inner loops for the comparable C functions. These generic linear algebra routines are written to be "cache-friendly"; that is, they are written to take advantage of a specified cache size, the preprocessor variable **szCACHE**, (platform dependent), but are otherwise not tuned for a particular architecture.

Chapter 8

GASpAR: In depth

We describe in this chapter our motivation for developing the code, and we present details on the formulation of computational operators, communication objects, and adaption as used in the code. In the process, moreover, we attempt to shed some light on various design considerations. Most of this chapter is adapted from the paper "Geophysical-astrophysical spectral-element adaptive refinement (GASpAR): Object-oriented h-adaptive fluid dynamics simulation", *J. Comp. Phys., in press* (2005), by D. Rosenberg, A. Fournier, P. Fischer, and A. Pouquet. This paper is also available at www.arxiv.org: math.NA/0507402

Accurate and efficient simulation of strongly turbulent flows is a prevalent challenge in many atmospheric, oceanic, and astrophysical applications. New simulation codes are needed to investigate such flows in the parameter regimes that interest the geophysics communities. Turbulent flows are linked to many issues in the geosciences, for example, in meteorology, oceanography, climatology, ecology, solar-terrestrial interactions, and solar fusion, as well as dynamo effects, specifically, magnetic-field generation in cosmic bodies by turbulent motions. Nonlinearities prevail when the Reynolds number Re is large. The number of 3-dimensional degrees of freedom (d.o.f.) increases as $\mathsf{Re}^{9/4}$ as $\mathsf{Re} \to \infty$ in the Kolmogorov 1941 framework [16, §7.4]. For aeronautic flows often $\mathsf{Re} > 10^6$, but for geophysical flows often $\mathsf{Re} \gg 10^8$ [11, 28]. Also, computations of turbulent flows must contain enough scales to encompass the energy-containing and dissipative scale ranges *distinctly*. Uniform-

grid convergence studies on 3D compressible-flow simulations show that in order to achieve the desired scale content, uniform grids must contain at least 2048^3 cells [34]. Today such computations can barely be accomplished. A pseudo-spectral Navier-Stokes code on a grid of 4096^3 uniformly spaced points has been run on the Earth Simulator [19], but the Taylor Reynolds number ($\propto \sqrt{\mathsf{Re}}$) is still no more than ≈ 700, very far from what is required for most geophysical flows. The *main goal of the present code development* is to ask, if the significant structures of the flow are indeed sparse, so that their dynamics can be followed accurately even if they are embedded in random noise, then does dynamic adaptivity offer a means for achieving otherwise unattainable large Re values. Thus, we have developed a dynamic geophysical and astrophysical spectral-element adaptive refinement (GASpAR) code for simulating and studying turbulent phenomena.

Several properties of spectral-element methods [SEMs, 9, 29] make them desirable for direct numerical simulation of geophysical turbulence. Perhaps most significant is the fact that SEMs performed at high polynomial degree are inherently minimally diffusive and dispersive. This property is clearly important when trying to simulate high-Re flows with multiple spatial and temporal scales that characterize turbulence. Also, because SEMs use finite elements, they can be used in very efficient high-resolution turbulence studies in domains with complicated boundaries. It is an important feature that SEMs are naturally parallelizable [*e.g.* , 15]. Equally important, SEMs not only provide spectral convergence when the solution is smooth (see Appendix C eq. C.3), but are also effective when the solution is not smooth.

Our goal in this chapter is to describe GASpAR and, in particular, the procedures used in our dynamic adaptive refinement (DARe) technique. We provide SEM and DARe algorithm *details* here that are not available elsewhere, in the hope of supporting readers who wish to create their own codes. Furthermore, we propose several linear and nonlinear problems as standards to test fundamental aspects of flows that are encountered in turbulence studies, and use these to test our DARe algorithms. Because these problems have known exact time-dependent solutions, quantitative errors can be reported for DARe simulations. Our code is object-oriented, and we will describe how object-oriented programming serves our purposes. The code is parallelized, but we will discuss this aspect only when it is intrinsic to the algorithms. While we are motivated by the performance potential of SEMs generally,

([8], [35]) we do not emphasize performance metrics in the present chapter, in favor of focusing on algorithmic detail and solution accuracy.

First we describe (§8.1.2) SEM discretization on a particular class of problems and introduce many of the required formulas, operators, and so forth. We explain (§8.1.4) how continuity is maintained between nonconforming elements. We provide linear-solver details in §8.1.5, and introduce innovations required to solve on nonconforming elements. In §8.1.6 we present our new adaptive-mesh algorithms: how neighboring elements are found, how conformity is established, and the procedures for refinement and coarsening. In §8.1.6 we describe a new implementation of element-boundary communication. DARe criteria are discussed in §8.1.6. Then, in §8.2 we propose and perform examples from two test-problem classes with time-dependent analytic solutions: the linear advection-diffusion equation (§8.2.2), demonstrating feature tracking of smooth and isolated features; and the 2D Burgers equation (§8.2.3), testing the ability of DARe to track well-defined increasingly sharp structures arising from nonlinear dynamics.

8.1 Temporal and dynamically adaptive spatial discretizations

8.1.1 Adaptive-mesh geometry

Conforming adaptive methods (where entire element boundaries geometrically coincide, as in Fig. 8.1a) on quadrilaterals and hexahedra are gradually being replaced by nonconforming adaptive methods. One reason is that locally adaptive mesh generation for conforming methods is complicated [30]. Another reason is that adaptive conforming meshes can lead to high-aspect-ratio elements that can cause difficulties for a linear solver [13]. Moreover, the fact that nonconforming elements can better localize mesh refinement implies that the computational cost over all elements can be reduced [24].

Nonconforming elements can be *geometrically* and/or *functionally* nonconforming. In the former case (Fig. 8.1b), neighboring-element boundaries do not entirely coincide; in the latter, the polynomial expansion degree

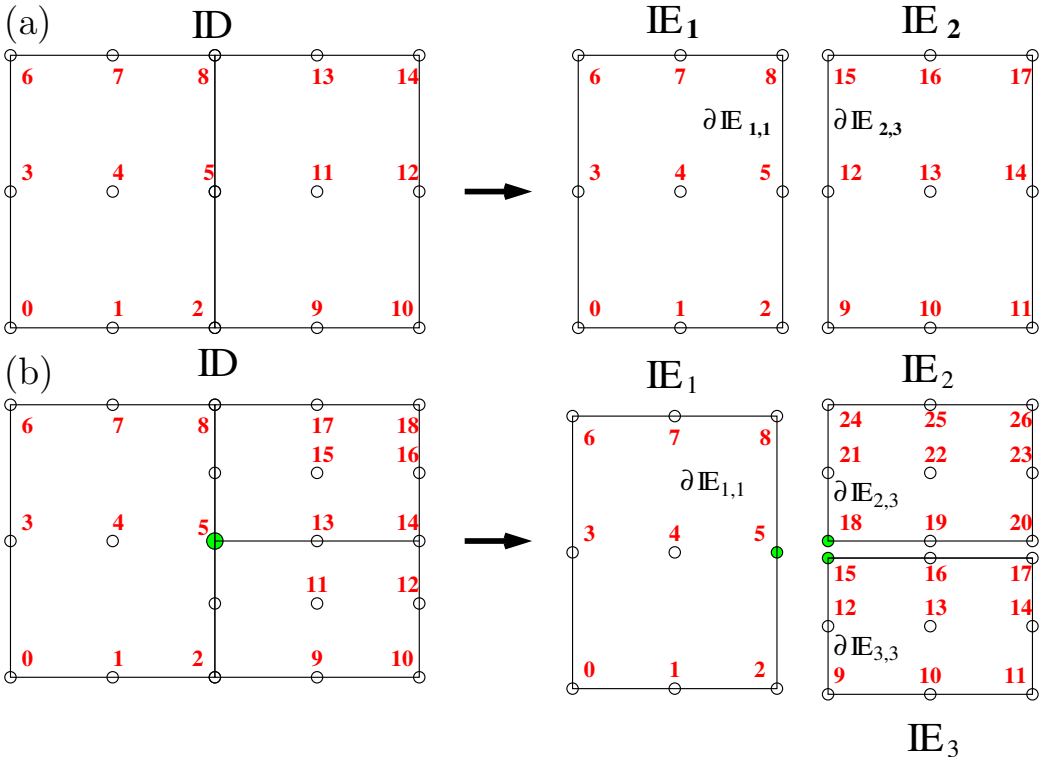

Figure 8.1: (a) Conforming degree $p = 2$ mesh showing the mapping of global (i.e., unique) d.o.f. in the domain $\bar{\mathbb{D}}$ to local (i.e., redundant) d.o.f. in the elements $\mathbb{E}_k$. Edge subscripts give element key k and edge index from $s = 0$ counterclockwise to $s = 3$. Element $\mathbb{E}_1$ is bounded at the east by $\partial\mathbb{E}_{1,1}$ and $\mathbb{E}_2$ at the west by $\partial\mathbb{E}_{2,3} = \partial\mathbb{E}_{1,1}$. Interface matching occurs by assignment, so the assembly matrix $\mathbf{A}_c$ is Boolean. (b) Geometrically *non*conforming (functionally conforming) mesh. Here $\mathbb{E}_2$ and $\mathbb{E}_3$ are bounded at the west by "child" edges $\partial\mathbb{E}_{2,3}$ and $\partial\mathbb{E}_{3,3}$, and $\mathbb{E}_1$ is bounded at the east by the "parent" edge $\partial\mathbb{E}_{1,1} = \partial\mathbb{E}_{2,3} \bigcup \partial\mathbb{E}_{3,3}$. Interface matching occurs by *interpolation* of global d.o.f. from the function space associated with $\partial\mathbb{E}_{1,1}$ onto the union of those associated with the $\partial\mathbb{E}_{k,3}$, which contains the function space of $\partial\mathbb{E}_{1,1}$.

p in neighboring elements differs. Several SEM researchers have adopted a method that simultaneously alters element size h and configuration (h-refinement) *and* the polynomial degree p across neighboring elements (p-refinement), providing for a so-called h-p-refinement strategy. The *mortar element method* (MEM) [1, 4, 10, 26] variationally minimizes the Lebesgue $\mathbb{L}_2$ norms of the discontinuities across nonconforming-element boundaries. MEM has been shown to produce optimal convergence in solving the incompressible Stokes equation [3], and has been demonstrated experimentally to produce excellent results when used as a basis for DARe in 1D [27]. Nonconforming h-p (not always *dynamic*) adaptive MEMs have been developed for studying turbulence [17, 18], ocean simulation [20, 25], flame front deformation [12], electromagnetic scattering [23], wave propagation [6], seismology [7] and other topics. However, MEM for p-type refinement has been cited as sometimes causing instability [30]. Also, in most flows of interest to us, it is the nonlinear interaction of the different scales that determines not only the structures that form but also their statistics and time evolution. This suggests that reasonably high-order approximations are required *in each element* during much of the evolution. Thus, in the present work we restrict ourselves to a nonconforming fixed-p, h-refinement strategy only and use an *interpolation-based* scheme to maintain continuity between nonconforming elements. This method [13, 24] is akin to the formulation developed in [5]; however, the latter deals with functionally nonconforming elements, while the former relates to the geometrically nonconforming elements of interest here. We contrast this choice with other familiar DARe codes [*e.g.* , 10], which, while object-oriented, uses the MEM as the basis of its dynamic adaptivity, but does not accommodate h-refinement. While the interpolation-based matching scheme has been widely used for functionally nonconforming meshes, to the best of our knowledge, our implementation of it in the context of fully dynamic adaptivity is unique and new.

8.1.2 Discretization of a nonlinearly coupled dynamical PDE system

In order to focus on DARe methodology, we concentrate on the simplest nonlinearly coupled PDE system that encompasses many of the difficulties in simulating fluid turbulence. Thus we discretize the 2D Burgers equation,

presenting in turn the spatial operators and the time discretizations. These sections are in part a review of well established methods but also provide implementation details unavailable elsewhere, and enable us to discuss code design motivations.

The equation considered in this work is the advection-diffusion equation for velocity $\bar{u}(\vec{x}, t)$:

$$\partial_t \bar{u} + \vec{c} \cdot \vec{\nabla} \bar{u} = \nu \nabla^2 \bar{u}, \tag{8.1}$$

where $\vec{c}$ may be $\bar{u}$ (so that (8.1) is the Burgers equation), or $\vec{c} = \vec{c}(t)$ (a prescribed uniform linear-advection velocity) and $\nu \propto \mathsf{Re}^{-1}$ is the kinematic viscosity. This is to be solved in a spatiotemporal domain $(\vec{x}, t) \in \mathbb{D} \times]0, t_\mathrm{f}]$ subject to the boundary and initial conditions

$$\bar{u}(\vec{x}, t) = \vec{b}(\vec{x}, t) \quad \text{for} \quad (\vec{x}, t) \in \partial\mathbb{D} \times]0, t_\mathrm{f}], \tag{8.2}$$

$$\bar{u}(\vec{x}, 0) = \bar{u}^0(\vec{x}) \quad \text{for} \quad \vec{x} \in \mathbb{D}. \tag{8.3}$$

Variational approach to spatial discretization

Then the discretization of (8.1) starts from the following "weak" variational form: Find the trial function $\bar{u}(\cdot, t) \in \mathbb{U}_{\vec{b}}$ such that for any test function $\vec{v} \in \mathbb{U}_{\vec{0}}$,

$$\langle \vec{v}, \partial_t \bar{u} \rangle + \langle \vec{v}, \mathcal{C}\bar{u} \rangle = -\nu \langle \vec{\nabla} v^\mathrm{T}, \vec{\nabla} u \rangle, \tag{8.4}$$

where $\mathcal{C} := \vec{c} \cdot \vec{\nabla}$ is the advection operator and the inner product is (C.8). (See the Appendix C for the complete mathematical details.) The treatment of (8.3) will not be made explicit but may be easily inferred from our general discussion.

Assume that $\bar{\mathbb{D}}$ can be partitioned as in Table C.1. Adopt a Gauss-Lobatto-Legendre (GLL) basis, that is, expand u^μ and v^μ using (C.6). Inserting these expansions into (8.4), we arrive at the semi-discrete ODE system problem: Find the numerical solution $\bar{u}_\mathrm{n}(\cdot, t) = \vec{\boldsymbol{\phi}}^\mathrm{T} \boldsymbol{u}(t) \in \mathcal{P}_{\boldsymbol{h},\vec{p}} \mathbb{U}_{\vec{b}}$ such that for all $\vec{v} = \vec{\boldsymbol{\phi}}^\mathrm{T} \boldsymbol{v} \in \mathcal{P}_{\boldsymbol{h},\vec{p}} \mathbb{U}_{\vec{0}}$,

$$\boldsymbol{v}^\mathrm{T} \mathbf{M} \frac{d\boldsymbol{u}}{\mathrm{d}t} + \boldsymbol{v}^\mathrm{T} \mathbf{C} \boldsymbol{u} = -\nu \boldsymbol{v}^\mathrm{T} \mathbf{L} \boldsymbol{u}, \tag{8.5}$$

collocated at $K(p+1)^d$ mapped Lagrange node points (Table C.1), where $\mathbf{M} = \mathrm{diag}_k \mathbf{M}_k$, $\mathbf{C} = \mathrm{diag}_k \mathbf{C}_k$, and $\mathbf{L} = \mathrm{diag}_k \mathbf{L}_k$ are the unassembled block-diagonal mass matrix, linear or nonlinear advection matrix [cf. 9, ch. 6], and

diffusion matrix, respectively. The respective $d(p+1)^d$-square matrix blocks for element $\mathbb{E}_k$ are formulated in Appendix C.

Note that after assembly as discussed in §8.1.4, (8.5) must hold for the restriction $\vec{v}|_{\bar{\mathbb{E}}_k} = \vec{\boldsymbol{\phi}}_k^{\mathrm{T}} \boldsymbol{v}_k$ of $\vec{v}$ to the kth element $\mathbb{E}_k$, so that a coupled ODE system for $\bar{u}_{\mathrm{n}}|_{\bar{\mathbb{E}}_k} = \vec{\boldsymbol{\phi}}_k^{\mathrm{T}} \boldsymbol{u}_k$ would in an assembled state be

$$\mathsf{M}_k \frac{d\boldsymbol{u}_k}{dt} + \mathsf{C}_k \boldsymbol{u}_k = -\nu \mathsf{L}_k \boldsymbol{u}_k. \tag{8.6}$$

Assembly guarantees continuity of $\bar{u}_{\mathrm{n}}$ across all elements, which in turn is sufficient to keep $u_{\mathrm{n}}^{\mu} \in \mathbb{H}^1(\mathbb{D})$. There are conforming and nonconforming element configurations, as illustrated in Fig. 8.1, and an interpolation-based scheme to enforce continuity along a nonconforming interface is the subject of §8.1.4. (Throughout the remainder of this chapter "nonconforming" will refer to geometrically nonconforming elements, keeping the polynomial degree p fixed in all elements.)

Semi-implicit multistep time discretization

GASpAR employs semi-implicit multistep time discretization schemes. The diffusion is always solved fully implicitly, the time derivative is approximated using a backward-difference formula (BDF) of order M_{bdf} [9, 21] and the advection term is approximated by an explicit extrapolation-based method (Ext) of order M_{ext} [22]. Then the integral of (8.6) from t^{n-1} to t^n is approximated by

$$\mathsf{H}_k^n \boldsymbol{u}_k^n = \sum_{m=n-M_{\mathrm{bdf}}}^{n-1} \beta_{\mathrm{bdf}}^{m,n} \mathsf{M}_k^m \boldsymbol{u}_k^m - \sum_{m=n-M_{\mathrm{ext}}}^{n-1} \beta_{\mathrm{ext}}^{m,n} \mathsf{C}_k^m \boldsymbol{u}_k^m, \tag{8.7}$$

where

$$\mathsf{H}_k^n := \beta_{\mathrm{bdf}}^{n,n} \mathsf{M}_k^n + \nu \mathsf{L}_k^n \tag{8.8}$$

is the spectral-element Helmholtz matrix. Although the matrices L_k and M_k in (8.6) were t-independent, they are time-indexed in (8.7) and (8.8) because DARe will, in general, reconfigure the partition (Table C.1) over time. For this reason the coefficients $\beta^{m,n}$ are re-computed for each t^n after a reconfiguration, as in the traditional schemes cited, except that the timestep Δt^m

may vary with m as the smallest spectral-element diameter $h^m := \min_k h_k^m$ (Table C.1) changes. The accuracy of solving (8.7) follows from many known SEM error estimates, e.g., for the Helmholtz problem on conforming meshes [21, §2.3.6] or the Poisson problem on non-conforming meshes [21, §5.5.2.1]. In §8.1.5 the solution of (8.7) is explained.

8.1.3 Implications for code design

The fully discretized advection-diffusion equation (8.7) brings up several issues impinging on code design. First, all mesh information is separated from all other code objects, since element type information can be encoded easily into the objects that require this distinction. Second, solution data must be available at multiple times t^m, so this information is provided in a data structure. Thus arise both *element* and *field* objects. The former contains all d-dimensional mesh information, including the Gauss-quadrature nodes and weights (Table C.1). The *element* object also contains neighbor-list information and the hierarchical element refinement level $\propto -\log_2 h_k$ of each element $\mathbb{E}_k$. The *field* object contains the data $\boldsymbol{u}^m$ quantifying the physical system of interest at each t^m.

The 1D basis functions, the derivative matrices and Gauss-quadrature nodes and weights (Table C.1) are encapsulated in *basis* classes (objects), and the 1D matrices such as (C.9,C.10,8.8) are objects that contain pointers to the basis objects and to a local element object. Generally d-dimensional SEM matrices are not constructed but are applied using 1D tensor-product matrix factors. High-level objects encapsulate the solution of (8.6) or other equations, and have common interfaces that allow the equations to take a single time integration step. In other words, all high-level equation-solver classes are used in the same way; they are constructed using linked lists of elements, fields and multidimensional SEM objects that depend only on the underlying mesh. Hence, the classes that handle DARe and enforce continuity between elements are independent of the system being solved.

8.1.4 Continuity and global assembly of nonconforming elements

Conforming discretizations enforce continuity simply by assigning the same weighted-averaged $\bar{u}_{\rm n}$ values to the coinciding node points $\vec{x}_{\vec{\jmath},k} = \vec{x}_{\vec{\jmath}',k'}$ along element edges $\partial\mathbb{E}_{k,s} = \partial\mathbb{E}_{k',s'}$ (Fig. 8.1a). This matching condition consists of expressing the $N_{\rm g}$ global (unique) d.o.f. $\boldsymbol{u}_{\rm g}$ in terms of the local (redundant) d.o.f. as $d(p+1)^d$-vectors $\boldsymbol{u}_k$, $k \in \{1, \cdots K\}$. Generally $N_{\rm g} < Kd(p+1)^d$. This expression is accomplished by using a $Kd(p+1)^d \times N_{\rm g}$ Boolean assembly matrix $\mathbf{A}_{\rm c}$ (also called a *scatter* matrix):

$$\boldsymbol{u} = \mathbf{A}_{\rm c}\boldsymbol{u}_{\rm g}. \tag{8.9}$$

The transpose $\mathbf{A}_{\rm c}^{\rm T}$ performs the *gather* operation associated with the $\mathbf{A}_{\rm c}$ scatter. In practice, $\mathbf{A}_{\rm c}$ is never formed explicitly but is instead *applied*.

In the nonconforming case $\partial\mathbb{E}_{k,s} \subsetneq \partial\mathbb{E}_{k',s'}$ and most boundary-node points are not coinciding (Fig. 8.1b). In the present work, unlike in MEM, the interface matching does not alter the underlying function space $\mathbb{U}_{\vec{b}}$ (§8.1.2). To illustrate, consider the nonconforming mesh in Fig. 8.1b. For the moment denote the global nodes, those nodes residing on the east *parent* edge $\partial\mathbb{E}_{1,1}$, by $\vec{x}_{{\rm g},i}$, $i \in \{2, 5, 8\}$, and denote the nodes on the west *child* edges, $\partial\mathbb{E}_{2,3}$ and $\partial\mathbb{E}_{3,3}$ by $\vec{x}_j$, $j \in \{9, 12, 15, 18, 21, 24\}$. A globally continuous function can always be found in $\mathbb{U}_{\vec{0}}$ in a proper subspace of the span of globally discontinuous functions $\phi_j(\vec{x})$ that interpolate from the local nodes $\vec{x}_j$. Therefore the weak formulation of (8.1) implies functions $\phi_{{\rm g},i}(\vec{x})$ exist that are globally continuous across $\mathbb{D}$, span $\mathbb{U}_{\vec{0}}$, and interpolate from the global nodes $\vec{x}_{{\rm g},i}$. Therefore the matrix $\mathbf{A}$, that generalizes the Boolean scatter matrix $\mathbf{A}_{\rm c}$ used in the conforming-element formulation, can be conceived as having entries $\phi_{{\rm g},i}(\vec{x}_j)$, and accommodates both conforming *and* nonconforming elements. It is convenient to factor $\mathbf{A} = \mathbf{\Phi}\mathbf{A}_{\rm c}$, where $\mathbf{\Phi}$ is the interpolation matrix from global to local d.o.f. and $\mathbf{A}_{\rm c}$ is locally conforming. Another illustration appears in [32, (14–16)].

To accommodate Dirichlet boundary conditions (8.2) into the solution, we employ a *masking projection* $\mathbf{\Pi}$, which is diagonal with unit entries everywhere except corresponding to nodes on Dirichlet boundaries, where there are zero entries. Any field $\vec{\boldsymbol{\phi}}^{\rm T}\boldsymbol{u} = \bar{u} \in \mathbb{U}_{\vec{b}}$ may be analyzed as $\bar{u} = \bar{u}_{\rm h} + \bar{u}_{\rm b}$, where $\boldsymbol{u}_{\rm h} := \mathbf{\Phi}\mathbf{\Pi}\mathbf{A}_{\rm c}\boldsymbol{u}_{\rm g}$ constructs the projection $\bar{u}_{\rm h} := \vec{\boldsymbol{\phi}}^{\rm T}\boldsymbol{u}_{\rm h} \in \mathbb{U}_{\vec{0}}$ of $\bar{u}$, that is,

Table 8.1: PCG algorithm modified for nonconforming element meshes.

$\boldsymbol{u}_\mathrm{h} = \boldsymbol{0}$ // initialize homogeneous term
$\boldsymbol{r} = \boldsymbol{\Sigma}(\boldsymbol{f} - \mathsf{H}\mathsf{S}\boldsymbol{u}_\mathrm{b})$ // initialize residual
$\boldsymbol{w} = \boldsymbol{0}$ // initialize search vector
$\rho_1 = 1$ // initialize parameter
while not converged:
  $\boldsymbol{e} = \mathsf{S}\mathsf{P}^{-1}\boldsymbol{r}$ // error estimate
  $\rho_0 = \rho_1$, $\rho_1 = \boldsymbol{r}^\mathrm{T}\mathsf{W}\boldsymbol{e}$ // update parameters
  $\boldsymbol{w} \leftarrow \boldsymbol{e} + \boldsymbol{w}\rho_1/\rho_0$ // increment search vector
  $\boldsymbol{r}' = \boldsymbol{\Sigma}\mathsf{H}\boldsymbol{w}$ // image of $\boldsymbol{w}$
  $\alpha = \rho_1/\boldsymbol{w}^\mathrm{T}\mathsf{W}\boldsymbol{r}'$ // component of $\boldsymbol{u}_\mathrm{h}$ increment
  $\boldsymbol{u}_\mathrm{h} \leftarrow \boldsymbol{u}_\mathrm{h} + \alpha\boldsymbol{w}$ // increment $\boldsymbol{u}_\mathrm{h}$ along $\boldsymbol{w}$
  $\boldsymbol{r} \leftarrow \boldsymbol{r} - \alpha\boldsymbol{r}'$ // increment residual
end
$\boldsymbol{u} = \mathsf{S}_\mathrm{f}(\boldsymbol{u}_\mathrm{h} + \boldsymbol{u}_\mathrm{b})$.

its homogeneous part, and $\boldsymbol{u}_\mathrm{b} := \boldsymbol{u} - \boldsymbol{u}_\mathrm{h}$ constructs $\bar{u}_\mathrm{b} \in \mathbb{U}_{\vec{0}}$, which vanishes at the interior nodes $\vec{x}_{\vec{\jmath},k} \in \mathbb{D}\backslash\partial\mathbb{D}$. Inserting this analysis into (8.5) (noting $\vec{v} \in \mathbb{U}_{\vec{0}} \Rightarrow \boldsymbol{v} = \boldsymbol{\Phi}\boldsymbol{\Pi}\mathsf{A}_\mathrm{c}\boldsymbol{v}_\mathrm{g}$) and repeating the time discretization leading to (8.7), we arrive at the following linear equation to solve for $\boldsymbol{u}_\mathrm{g}$ at each time step:

$$\boldsymbol{v}^\mathrm{T}\mathsf{H}\boldsymbol{u} = \boldsymbol{v}^\mathrm{T}\boldsymbol{f} \quad \forall \boldsymbol{v}_\mathrm{g} \Longrightarrow \mathsf{A}_\mathrm{c}^\mathrm{T}\boldsymbol{\Pi}\boldsymbol{\Phi}^\mathrm{T}\mathsf{H}\boldsymbol{\Phi}\boldsymbol{\Pi}\mathsf{A}_\mathrm{c}\boldsymbol{u}_\mathrm{g} = \mathsf{A}_\mathrm{c}^\mathrm{T}\boldsymbol{\Pi}\boldsymbol{\Phi}^\mathrm{T}(\boldsymbol{f} - \mathsf{H}\boldsymbol{u}_\mathrm{b}), \quad (8.10)$$

where $\mathsf{H} := \mathrm{diag}_k\, \mathsf{H}_k$ is symmetric positive-definite (8.8) and we have denoted all past-time terms from time-derivative expansion and advection in (8.7) by $\boldsymbol{f}$. The preconditioned conjugate-gradient [PCG, 33, 37] algorithm is used to solve (8.10). While (8.10) shows explicitly that the l.h.s. matrix is symmetric nonnegative-definite, it is not in a form easily solved in parallel. Left-multiplying (8.10) by $\boldsymbol{\Phi}\boldsymbol{\Pi}\mathsf{A}_\mathrm{c}$, we get the following local problem to solve for $\boldsymbol{u}_\mathrm{h}$:

$$\boldsymbol{\Sigma}\mathsf{H}\boldsymbol{u}_\mathrm{h} = \boldsymbol{\Sigma}(\boldsymbol{f} - \mathsf{H}\boldsymbol{u}_\mathrm{b}), \qquad \text{where} \qquad \boldsymbol{\Sigma} := \boldsymbol{\Phi}\boldsymbol{\Pi}\mathsf{A}_\mathrm{c}\mathsf{A}_\mathrm{c}^\mathrm{T}\boldsymbol{\Pi}\boldsymbol{\Phi}^\mathrm{T}. \quad (8.11)$$

The *direct stiffness summation* (DSS) matrix $\boldsymbol{\Sigma}$ is coded so that the gather and scatter are performed in one operation (§8.1.6), which reduces parallel communication overhead [35].

Two other operators must be introduced that help maintain $\mathbb{H}^1(\mathbb{D})$ continuity. The inverse *multiplicity matrix* W is diagonal, computed by initializing

a collocated vector $g^{\mu}_{\vec{\jmath},k} = 1\ \forall \vec{\jmath}, k, \mu$, setting child boundary nodes to 0, performing $\boldsymbol{g} \leftarrow \boldsymbol{\Phi}\mathbf{A}_{\mathrm{c}}\mathbf{A}_{\mathrm{c}}^{\mathrm{T}}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{g}$, then setting

$$W^{\mu,\mu'}_{\vec{\jmath},k,\vec{\jmath}',k'} = \begin{cases} \delta^{\mu,\mu'}/g^{\mu}_{\vec{\jmath},k}, & \text{if } \vec{x}_{\vec{\jmath},k} = \vec{x}_{\vec{\jmath}',k'} \text{ coincides with a global node,} \\ 0, & \text{otherwise.} \end{cases}$$

For example, corresponding to Figs. 8.1a & b the diagonals of $\mathbf{W}$ are

$$\begin{aligned} & (\quad 1, 1, \tfrac{1}{2}, 1, 1, \tfrac{1}{2}, 1, 1, \tfrac{1}{2}, \tfrac{1}{2}, 1, 1, \tfrac{1}{2}, 1, 1, \tfrac{1}{2}, 1, 1) \\ \text{and} \quad & (\quad 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, \tfrac{1}{2}, \tfrac{1}{2}, 0, \tfrac{1}{2}, \tfrac{1}{2}, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1) \end{aligned} \tag{8.12}$$

respectively. After a DSS operation (8.11) the true global d.o.f., nodes 2, 5, and 8, carry all the information held by nodes 9, 12, 15, 18, 21, and 24, so for the purpose of the PCG solve the latter give zero $\mathbf{W}$ entries in (8.12). Given that global inner products in the PCG solve are collected from local contributions from each element (i.e., Table 8.1, the lines involving $\mathbf{W}$), the $\mathbf{W}$ zeros prevent double counting when computing these products, and prevent non-global d.o.f. (e.g., child edge nodes) from contributing. Note also that in Fig. 8.1b the $\mathbf{W}$ entries for nodes 17 and 20 have value $\frac{1}{2}$, as expected for nodes such as these that lie on conforming edges. The $\mathbb{H}^1$ "smoothing" operation in the PCG algorithm also uses $\mathbf{W}$. In smoothing, we have that $\bar{\boldsymbol{g}} = \mathbf{S}\boldsymbol{g}$, where $\mathbf{S} := \boldsymbol{\Phi}\boldsymbol{\Pi}\mathbf{A}_{\mathrm{c}}\mathbf{A}_{\mathrm{c}}^{\mathrm{T}}\mathbf{W}$. Smoothing acts only on quantities all of whose d.o.f. have already been distributed to global d.o.f. using DSS. The result of smoothing is a quantity that is interpolated properly to the child edges and that is expressed without multiple counting at multiple local nodes that represent the same physical location. The $\mathbf{W}$ matrix weights the operand $\boldsymbol{g}$ so that the respective sums on the parent (global) edge nodes (nodes 2, 5, and 8 in the case above) contribute to the result $\bar{\boldsymbol{g}}$ just once each, and the child edge nodes receive their $\bar{\boldsymbol{g}}$ values from the parent edge nodes by interpolation.

8.1.5 Modified preconditioned conjugate-gradient algorithm

It is important to modify the well known PCG algorithm in order to solve (8.11) in the nonconforming case. The modifications stem from the requirement that the iteration residuals $\boldsymbol{r}$ and the search directions $\boldsymbol{w}$ correspond

to functions $\vec{r} \equiv \vec{\boldsymbol{\phi}}^{\mathrm{T}}\boldsymbol{r}$ and $\vec{w} \equiv \vec{\boldsymbol{\phi}}^{\mathrm{T}}\boldsymbol{w}$ belonging to $\mathbb{H}^1(\mathbb{D})^d$. The CG algorithm searches the global d.o.f. space for the solution to the linear equation. So that we may continue to use the local matrix forms, however, we must also mask off all Dirichlet nodes (if any exist), which are not solved for. The $\boldsymbol{\Sigma}$ matrix (8.11) masks off these nodes in such a way that the new search direction $\vec{w} \in \mathbb{H}^1(\mathbb{D})^d$. Additionally, in all cases in the CG iteration where a quantity $\vec{g}$ must remain in $\mathbb{H}^1(\mathbb{D})^d$, we explicitly "smooth" it by using the smoothing operator, $\mathbf{S}$ (cf.§8.1.4). Note that it is critical that the inhomogeneous boundary term $\bar{u}_{\mathrm{b}}$ belong to $\mathbb{H}^1(\mathbb{D})^d$ in (8.11); thus, the smoothing matrix $\mathbf{S}$ is applied to $\boldsymbol{u}_{\mathrm{b}}$ before $\mathbf{H}$ is. However, the non-smoothed boundary term must be added after the convergence loop in order to complete the solution. Note also that the final smoothing operation follows the addition of the boundary condition and therefore *cannot* be masked; hence the distinction of the final matrix $\mathbf{S}_{\mathrm{f}} := \boldsymbol{\Phi}\mathbf{A}_{\mathrm{c}}\mathbf{A}_{\mathrm{c}}^{\mathrm{T}}\mathbf{W}$.

With these considerations we present in Table 8.1 the PCG algorithm for the assembled local problem (8.11) modified from the conforming-elements case, here for nonconforming elements. Preconditioning is handled by the matrix $\mathbf{P}^{-1}$. GASpAR includes block- and point-Jacobi preconditioners. For the test problems presented in §8.2, a point Jacobi preconditioner has proven to be adequate. In general, the preconditioned quantity must be smoothed, as indicated in Table 8.1.

8.1.6 Adaptive mesh formulation

Element-mesh hierarchical configuration

We now employ nonconforming connectivity to carry out dynamic adaptivity. Recall that the global domain $\mathbb{D}$ is initially covered (Table C.1) by a set of disjoint (non-overlapping) elements $\mathbb{E}_k$. Each of these initial elements becomes a tree *root* element, identified by a unique root *key* k_{r} for that tree. At each level $\ell \in \{\ell_{\min}, \cdots \ell_{\max}\}$, an element data structure provides both its own key k and its root key k_{r}. For any level ℓ, the range of $2^{d\ell}$ valid element keys will be $k \in [2^{d\ell}k_{\mathrm{r}}, 2^{d\ell}(k_{\mathrm{r}}+1)-1]$ because the refinement is *isotropic* (that is, it splits an element at the midpoints of all its edges to produce its 2^d child elements). Conversely, we obtain the level index from the element

key using

$$\ell = \lfloor \log_{2^d}(k/k_{\rm r}) \rfloor. \tag{8.13}$$

In order to ensure all keys are unique, the first $k_{\rm r} := 1$ and the next is $k'_{\rm r} := 2^{d\ell_{\max}}(k_{\rm r}+1)$, and so on.

After elements $\mathbb{E}_k$ are identified ("tagged") for refinement or coarsening at level ℓ, three steps are involved in performing DARe: (1) performing *refinement* by adding a new level of 2^d child elements $\mathbb{E}_{2^d k}, \cdots \mathbb{E}_{2^d(k+1)-1}$ at level $\ell+1$ to replace each $\mathbb{E}_k$, or else *coarsening* 2^d existing children $\mathbb{E}_k, \cdots \mathbb{E}_{k+2^d-1}$ into a new parent $\mathbb{E}_{\lfloor k/2^d \rfloor}$; (2) building data structures for all element *boundaries*, which hold data representing global d.o.f. and accept gathers ($\mathbf{A}^{\rm T}\boldsymbol{u}$ segments) or perform scatters ($\mathbf{A}\boldsymbol{u}_{\rm g}$ segments); and (3) determining neighbor lists for data exchange. Neighbor lists consist of records (structures) that each contain the computer processor id, element key k, root key $k_{\rm r}$ and boundary id $s \in \{0, \cdots 2^d - 1\}$ of each neighbor element that adjoins every interface. In refining or coarsening, the field values for each child (parent) elements are interpolated from the parent (child) fields. For simplicity, the interior of each element boundary (i.e., excluding the vertices) is restricted to an interface between one coarse and at most 2^{d-1} refined neighbors. Thus, at most one refinement-level difference will exist across the interior of an interface between neighboring elements.

In GASpAR, the data structures that represent global d.o.f. at the inter-element interfaces are referred to as "mortars." These structures are not to be confused with the mortars used in MEM; however, they serve as templates for that more general method. Recalling Fig. 8.1b as a paradigm, in general the mortars contain node locations and the basis functions of the parent element boundary (edge in 2D, or face in 3D). The mortar structures represent the same field information for the parent and child edges; their nodes coincide with the nodes of the parent edge, and they interpolate global d.o.f. data to the child edges, as described above. The mortar data structures are determined by communicating with all neighbors to determine which interfaces are nonconforming. This communication uses a *voxel database* (VDB) [17]. A VDB consists of records containing geometric point locations, a component id that tells what part of the element $\mathbb{E}_k$ (in 2D, edge $\partial\mathbb{E}_{k,s}$, vertex $\in \partial^2\mathbb{E}_{k,s}$, etc.) the point represents, an id of the element that contains the point, the root id of that element, and some auxiliary data. Two VDBs are constructed: one consists of all element vertices, and one consists of all

element edge midpoints. With these two VDBs, we are able to determine whether a relationship between neighbor edges is conforming and also determine the geometrical extent of the mortar. The VDB approach can also be used for general deformed geometries in two and three dimensions, as long as adjacent elements share well-defined common node points.

The algorithm classes that carry out DARe operate only on the element and field lists. The SEM solvers adjust themselves automatically to accommodate the dynamic addition and removal of elements that occurs as a result of DARe.

Refinement and coarsening rules

The refinement and coarsening method takes as input only the local indexes of the elements to be refined and coarsened. Before refinement or coarsening is done, the tagged elements are checked for compliance with several rules. For refinement, the rules are: (R1) the refinement level must not exceed a specified limit $\ell_{\max}$; and (R2) at most one level may separate neighbor elements. These rules must be followed also for interfaces at periodic boundaries. Rule R2 is enforced by tagging a coarse element for refinement too, if it has an already refined neighbor tagged for further refinement. Enforcement of R1 and R2 is most easily effected by building a global list of keys of all elements tagged for refinement, and comparing the local refinement lists with it.

We may not coarsen an element under any of the conditions: (C1) it is a root; (C2) any of its $2^d - 1$ siblings are not tagged for coarsening; (C3) it appears in a refinement list; or (C4) rule R2 would be violated. To enforce C4, we use a *query-list*, i.e., a global list of each element key k, its parent key $\lfloor k/2^d \rfloor$, and its level ℓ (8.13). The query-list contains keys gathered from all processors. The following procedure is then used.

1. Build a global "refinement" query-list (RQL) from the keys in the local refinement list.
2. Find level limits $\ell_{\max}$ and $\ell_{\min}$ from the coarsen list.
3. Reorder the current local coarsen list from $\ell_{\max}$ down to $\ell_{\min}$.

4. Looping from $\ell = \ell_{\max}$ down to $\ell_{\min}$: build a global "coarsen" query-list (CQL) from the keys in the current local coarsen list; and for all keys k in the local coarsen list at the current ℓ, if any refined neighbor is in the CQL and no refined neighbors are in the the RQL, then k is retained in the current coarsen list; otherwise it is deleted.

5. Check finally that all elements in the local coarsen list have all their siblings also tagged for coarsening. The sibling elements of k are identified by having the same parent key $\lfloor k/2^d \rfloor$.

Note that the local refinement lists are checked and possibly modified *before* checking and modifying the coarsen lists.

Communicating boundary data

The mortar data structures contain all the data to be communicated between elements during each application of the DSS $\mathbf{\Sigma}$ (8.11) or smoothing operation $\mathbf{S}$. Communication of element-boundary data requires network communication on parallel computers. This involves *initialization* and *operation* steps. Initialization establishes element-processor connectivity by bin-sorting global node indexes and having each processor examine the nodes from one bin, to determine element-neighbor lists. This method has been suggested in [9, §8.5.2] but to our knowledge has never before been implemented. All coinciding mortar-structure nodes $\vec{x}_{\mathrm{g},i} = \vec{x}_{\mathrm{g},i'}$ are uniquely labeled by their Morton index $\mathrm{M}(\vec{x}_{\mathrm{g},i})$, computed by digitizing the d coordinates and partially interleaving the B bits along each coordinate μ. So for $a^\mu := \min_{\vec{x}\in\mathbb{D}} x^\mu$:

$$\mathrm{M}^\mu(\vec{x}) := \lfloor \frac{x^\mu - a^\mu}{\Delta x} + \tfrac{1}{2} \rfloor \quad \Rightarrow \quad \mathrm{M}(\vec{x}) := \sum_{\mu=1}^{d} 2^{(\mu-1)B} \mathrm{M}^\mu(\vec{x}) \in \left\{0, \cdots 2^{dB} - 1\right\},$$

where Δx is chosen so that $\mathrm{M}^\mu(\vec{x}) \in \left\{0, \cdots 2^B - 1\right\}\ \forall \vec{x}$. For P processors, a collection of P bins $\mathbb{B}_l$, $l \in \{0, \cdots P-1\}$, is generated that partitions the dynamic range (over all processors) of the Morton indexes. Processor l partitions its list of indexes into the bins, sending the contents of $\mathbb{B}_{l'}$ to processor l', where the information is combined with those from other processors and then sent back to processor l. After this initialization step, every processor is informed of which other processors share which mortar nodes.

The operation step communicates the data at any node point $\vec{x}_{\mathrm{g},i}$ with all other processors that share it. These data are extracted from the containing element by using the pointer indirection provided by $\mathrm{M}(\vec{x}_{\mathrm{g},i})$. The field values at $\vec{x}_{\mathrm{g},i}$ are summed during DSS or smoothing and reassigned to $\vec{x}_{\mathrm{g},i}$ also by indirection. To reduce communication, shared $\vec{x}_{\mathrm{g},i}$ residing on the same processor are summed before being transmitted to the other processors that share that $\vec{x}_{\mathrm{g},i}$. At the end of the operation step, the field values at multiply-represented global nodes are identical. This gather-scatter procedure ensures that the DSS output are locally available immediately after communication. One benefit of this gather-scatter method is that it allows communication to be separated from the geometry, because Morton indexes are essentially unstructured lists of local data locations. However, a future upgrade of GASpAR will use VDBs to obviate the need for the bin-sort initialization step, which requires information already provided in the VDBs.

Error estimators

Elements are tagged for DARe by the use of an a posteriori criterion. The *spectral estimator* criterion, modified from [18, 27], uses local Legendre spectra to estimate the quadrature and truncation errors and the spectral convergence rate in each element $\bar{\mathbb{E}}_k = \vec{\vartheta}_k([-1,1]^d)$. First, the mapping $u^\mu \circ \vec{\vartheta}_k(\vec{\xi})$ of each solution component $u^\mu(\vec{x})$ is transformed to spectral coefficients $u_j^{\mu,\mu'}$ along a 1D line in coordinate $\xi^{\mu'}$ by [9, (B.3.13)], averaging over all the $\xi^{\mu''\neq\mu'}$. The convergence rates $\lambda^{\mu,\mu'}$ are fit using $|u_j^{\mu,\mu'}| \approx C^{\mu,\mu'} \exp^{-\lambda^{\mu,\mu'} j}$ [18, (18)] with $j \in \{p-3, \cdots p\}$, except that instead of "equivalent" 1D coefficients [18, (17)], we combine fits using $\lambda^\mu := \min_{\mu'=1}^d \lambda^{\mu,\mu'}$. The solution error $\varepsilon_{\mathrm{est}}^\mu$ is estimated using [18, (19) l.h.s.], except again instead of "equivalent" 1D coefficients, we estimate the first term of [18, (19)] by $\sum_{\mu'=1}^d (u_p^{\mu,\mu'})^2$ and the second term by $(\prod_{\mu'=1}^d (C^{\mu,\mu'})^2 \int_{p+1}^\infty \mathrm{d}j \exp^{-2\lambda^{\mu,\mu'} j})^{1/d}$. Thus, $\bar{\mathbb{E}}_k$ is *refined*, if for some μ, $\varepsilon_{\mathrm{est}}^\mu$ is above a threshold value ε_{t} or if λ^μ is below another threshold λ_{t}. For *coarsening*, for all μ, all 2^d sibling elements must have their $\varepsilon_{\mathrm{est}}^\mu$s below some value $\gamma_{\mathrm{c}}\varepsilon_{\mathrm{t}} < \varepsilon_{\mathrm{t}}$, computed by multiplying by a "coarsening multiplier" γ_{c}. This prevents "blinking," i.e., refined elements being immediately coarsened again. In conjunction with the spectral estimator, we can often obtain better overall accuracy convergence by thresholding on the $\bar{\mathbb{E}}_k$-maximum second derivative magnitude in any coordinate and taking

a logical OR of that criterion with the spectral estimator. While the high polynomial degrees will help the spectral estimator, given the variety of our future applications, new refinement criteria may be more effective. The investigation of refinement criteria appropriate, e.g., for intermittent features is a major outstanding problem in adaptive numerical solution of PDEs that we will consider in future work.

8.2 Results for adaptive (non)linear advection-diffusion simulation

Our test problems examine various aspects of (8.1). The primary goal is to investigate the solution temporal and spatial convergence when adaption is used. Thus we have selected problems with analytic solutions, so that errors may be determined *exactly*, instead of only by comparison e.g., to a uniformly highly refined control solution. Tests begin with the simplest aspect of (8.1) and progress through more difficult problems until the behavior of the full 2D nonlinear, multi-component version of (8.1) is considered. We do not use filtering for any of these test problems.

For each test the BDF3 and Ext3 schemes are used for the time-derivative and the advection terms in (8.7), respectively, unless stated otherwise. This requires that all the required time levels t^{m-1} be initialized, $m \in \{1, \cdots \max(M_{\rm bdf}, M_{\rm ext})\}$. A logical OR of the spectral and second-derivative error estimators or just the second-derivative estimator is used for the adaption criterion. The spectral estimator is normalized by the initial-condition norm $||\bar{u}^0||_\infty$, and the second derivative is normalized by $||\bar{u}^0||_\infty/L^2$, where L is the longest global domain length. The threshold $\lambda_{\rm t}$ is always set at 1 when used.

Except where we compare with published results, the viscosities are somewhat arbitrary. We reiterate that one of our motivations in considering (8.1) is that it exemplifies many of the characteristics of the Navier-Stokes equations of interest in simulating turbulence, including the dependence on ν via Re. However, we note that a recent paper [32] concludes that the MEM and the interpolation-based connectivity for nonconforming elements may manifest inconsistencies that affect convergence, which a small viscosity can prevent.

For the purposes of our tests, we perform adaption after every 10 timesteps except if stated otherwise. In practice, this is not optimal as the adaptivity overhead can overtake the computational savings achieved by reducing the required number of d.o.f.. In general, it is more meaningful and efficient to adapt at a fraction of a fiducial timescale, say an eddy turnover time. The refinement criteria are applied to each component of (8.1) that is solved for.

In order to compare an adaptive solution, we use an ℓ-*control* grid. This is a grid that uniformly covers the domain with elements at the finest resolution $\ell_{\max} = \ell$. For all spatial convergence tests that have control solutions, we will also provide a single processor speed-up factor representative of the adaptive solutions, by giving the ratio $T_{\text{control}}/T_{\text{adaptive}}$ of the total control and adaptive cpu run times. Naturally, this factor is only to be used for reference since the speed-up will, in general, depend not only on the solution and its refinement criteria and thresholds, but also the adaption interval, and expansion degree, p.

8.2.1 Adaptive heat-equation solution results

For the linear case $\vec{c} = \vec{c}(t)$ the fundamental solution of (8.1) is a Gaussian d-periodized in $\mathbb{D} = [0,1]^d$:

$$u_{\text{a}}^{\mu}(\vec{x},t) := \frac{\sigma(0)^d}{\sigma(t)^d} \sum_{\imath^1,\cdots\imath^d=-\infty}^{\infty} \exp - \left(\frac{\vec{x} - \vec{x}^{\,0} + \vec{\imath} - \int_0^t \vec{c}(t')\mathrm{d}t'}{\sigma(t)} \right)^2 \tag{8.14}$$

for $t > -\sigma(0)^2/4\nu$ ($u_{\text{a}}^{\mu}(\vec{x},t) := 0$ otherwise), where $\sigma(t) := \sqrt{\sigma(0)^2 + 4\nu t}$, $\sigma(0) = \sqrt{2}/20$ is the initial e-folding width and $\vec{x}^{\,0} = \sum_{\mu=1}^{d} \vec{e}^{\,\mu}/2$ is the initial peak location. To compute (8.14), we truncate summands of value less than 10^{-18} of the partial sum. The simplest version of (8.1) is the heat equation, where $\vec{c} = \vec{0}$. The goal here is to determine the temporal and spatial convergence when there is no advection. The initial condition (8.3) is computed on $K = 4 \times 4$ elements from (8.14) at $t = 0$ and $d = 2$, and the mesh is refined until refinement level $\leq \ell_{\max}$. Both the spectral estimator with threshold $\varepsilon_{\text{t}} = 10^{-3}$ and second-derivative estimator with threshold of 0.25 were used. The coarsening multipliers (to prevent blinking) for each were set to $\gamma_{\text{c}} = 0.5$ and 0.25, respectively. A BDF2 scheme is used here for the time derivative.

Temporal convergence of the adaptive heat-equation solution

We examine time convergence by advancing to $t_\mathrm{f} = 0.05$ for various constant Δt. From (8.14) curves of relative $\mathbb{L}_2$ error $\varepsilon = ||\bar{u}_\mathrm{n} - \bar{u}_\mathrm{a}||_2/||\bar{u}_\mathrm{a}^0||_2$ vs Δt are plotted for several maximum-refinement levels ℓ_max and for degrees p, in Fig. 8.2a-d, The control grid here consists of 16×16 elements. The BDF2 and Ext2 are globally second-order schemes, so if the solution is well resolved spatially, we expect to find a slope of ≈ 2 in a log-log plot of error vs Δt. Indeed this is seen in Fig. 8.2a-d; each panel shows a sequence of three curves for the refinement levels $\ell_\mathrm{max} \in \{0, \cdots 2\}$, where $\ell_\mathrm{max} = 0$ implies that no refinement is done. For the curves that are spatially resolved, the error is linear with slope 2.04. Even at low p, the solution is well resolved if DARe is used, even at $\ell_\mathrm{max} = 1$. If the refinement thresholds ε_t where increased slightly, we would see a larger reduction in the the number of d.o.f. required, but our accuracy would decrease, requiring a higher ℓ_max before accuracy (at small Δt) is restored. As p increases, there is less need for DARe, as is expected due to the smoothness of the solution.

Spatial convergence of the adaptive heat-equation solution

We now consider the effects of polynomial degree p. The maximum refinement is fixed at $\ell_\mathrm{max} = 2$. At time t^n a dynamic Courant-limited timestep

$$\Delta t^n \leq \mathsf{Co} \Bigg/ \max_{\vec{\jmath} \in \{1,\cdots p\}^d; k \in \{1,\cdots K^n\}; \mu,\mu' \in \{1,\cdots d\}} \left(\frac{4\nu}{(\Delta^n_{\vec{\jmath},k})^2} + \frac{|u^{\mu' n}_{\vec{\jmath}-\vec{e}^{\,\mu},k} + u^{\mu' n}_{\vec{\jmath},k}|}{2\Delta^n_{\vec{\jmath},k}} \right) \tag{8.15}$$

is used with a fixed Courant number $\mathsf{Co} = 1.0$, where $\Delta^n_{\vec{\jmath},k} := \min_{\mu \in \{1,\cdots d\}} |\vartheta^{\mu n}_k(\vec{\xi}_{\vec{\jmath}-\vec{e}^{\,\mu}}) - \vartheta^{\mu n}_k(\vec{\xi}_{\vec{\jmath}})|$ (Table C.1). We can set Co to a reasonably high value because a semi-implicit scheme is used. The solution is advanced to $t_\mathrm{f} = 0.5$, enough to observe the solution coarsening as it decays. Only the control runs use the variable timestep; the adaptive runs use as a fixed timestep the Courant-limited value of the corresponding control case at $t = t_\mathrm{f}$. The initial mesh is the same as §8.2.1.

Figure 8.3a shows the exponential spatial convergence characteristic of all our tests. We expect from (C.3) that an infinitely smooth solution will spectrally

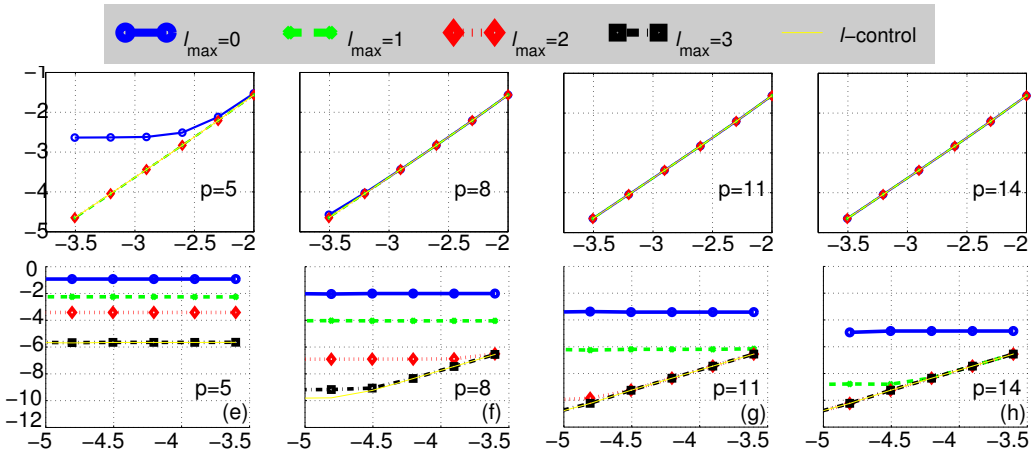

Figure 8.2: Plots of normalized error $\log_{10}(||\bar{u}_{\mathrm{n}} - \bar{u}_{\mathrm{a}}||_2/||\bar{u}_{\mathrm{a}}^0||_2)$ vs $\log_{10}\Delta t$ for (a-d) the heat equation and (e-h) advection-dominated flow (8.1), for different polynomial degrees p as labeled. Each upper panel shows curves for up to three maximum refinement levels $\ell_{\max}$ indicated in the legend; each lower panel shows four refinement levels. For the heat equation, The 2-control solutions (thin curves) overlie the $\ell_{\max} \geq 1$ adaptive curves. As p increases, the curves converge.

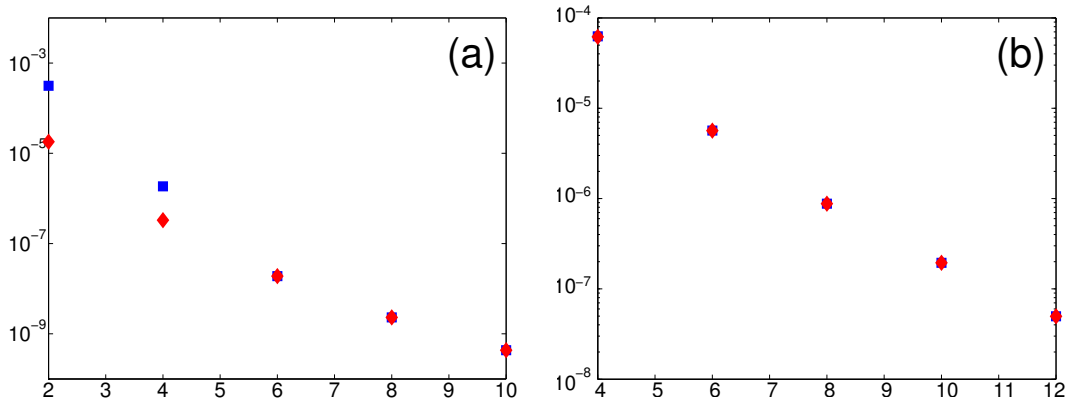

Figure 8.3: Semilog plots of $||\bar{u}_\mathrm{n} - \bar{u}_\mathrm{a}||_2/||\bar{u}_\mathrm{a}^0||_2$ vs p for (a) the diffusion, (b) advection-dominated flow. Square and diamond markers indicate the adaptive and ℓ_max-control runs, respectively. For diffusion, $\ell_\mathrm{max} = 2$, and for the advection-dominated cases, $\ell_\mathrm{max} = 3$.

converge along a straight-line plot of $\log_{10}(||\bar{u}_\mathrm{n} - \bar{u}_\mathrm{a}||_2/||\bar{u}_\mathrm{a}^0||_2)$ vs p. For lower p, the 2-control solutions are better than the adaptive runs, but the curves merge quickly, as we would expect for such a smooth problem. The adaptive curves show some slight concavity for this problem. The low-p error source is likely the elliptic nature of (8.10), so that coarse elements propagate their error throughout the mesh. Figure 8.4b shows that even for varying K (Fig. 8.4a), the error over time behaves monotonically, agreeing very closely with the control profile. We find that the adaptive cases for all but $p = 2$ case run significantly faster ($T_\mathrm{control}/T_\mathrm{adaptive} \approx 3$) than the controls for this problem.

8.2.2 Adaptive linear-advection simulation results

Next we consider the linear advection-dominated equation (8.1) with $d = 2$, $\nu = 10^{-4}$ and $\vec{c} = \vec{e}^1$. This tests the ability of the code to follow a localized translating distribution. The initial state (8.3) is given by (8.14) at $t = 0$. The spectral estimator in this problem is turned off. The second-derivative criterion is set to $\varepsilon_\mathrm{t} = 1$ with a coarsening multiplier of $\gamma_\mathrm{c} = 0.5$.

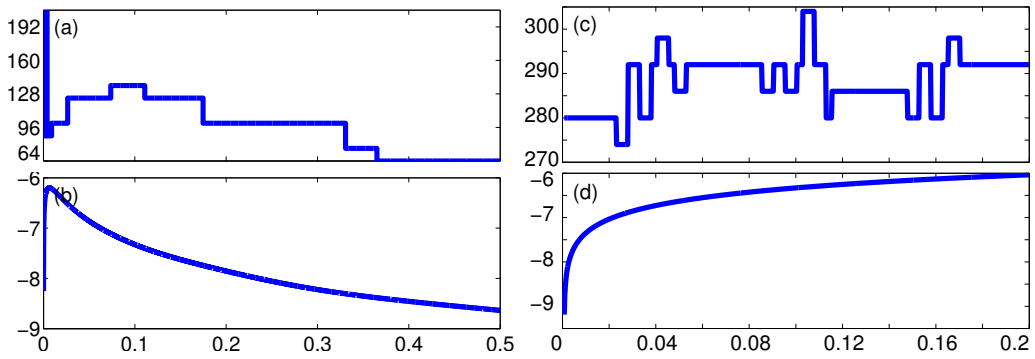

Figure 8.4: For the 2D adaptive (a-b) $p = 6$ heat-equation, and (c-d) $p = 8$ linear advection tests, time series of (a,c) number K of elements, and (b,d) $\log_{10}(||\bar{u}_\mathrm{n} - \bar{u}_\mathrm{a}||_2/||\bar{u}_\mathrm{a}^0||_2)$. The errors for the adaptive and control meshes lie on top of one another.

Temporal convergence for adaptive linear advection

Temporal convergence is tested as in §8.2.1, except that only the second-derivative criterion is used. The final $t_\mathrm{f} = 0.06$, and we begin with a $K = 4\times4$ element mesh. We present the results in 8.2e-h. The spatially resolved curves in each plot have an average slope of 2.95. Even at high degree p, the error is Δt-independent for the unrefined mesh. For lower p, the error decays at the order of the time-stepping method only if there are several refinement levels, indicating that the solution is well resolved spatially only at higher ℓ_max. Thus, in order to achieve a temporal error $\mathcal{O}(\Delta t^3)$, refinement is necessary.

Figure 8.2e-h also shows 3-control runs corresponding to the adaptive solutions, indicated by thin curves that all overlie the $\ell_\mathrm{max} = 3$ curves. As p increases, less refinement is required to achieve the same accuracy that 3-control does.

Spatial convergence for adaptive linear advection

We turn to the effects of polynomial degree p on the solution error. The maximum refinement level is fixed to $\ell_\mathrm{max} = 3$. Here, a Courant-limited timestep (8.15) is again used with $\mathsf{Co} = 0.2$. The solution is advanced to

$t_f = 0.2$, enough to see several DARe cycles occur (Fig. 8.4c). The initial mesh is the same as in §8.2.2. Spectral error decay can be seen in Fig. 8.3b, which also shows the 3-control solutions. The adaptive solution error decays nearly identically as does the 3-control, suggesting again that interpolation introduces no deleterious effects for this problem.

Figure 8.4c-d shows typical time series of the element count K and the error. Clearly, adaptivity does not alter the monotonic error behavior. The 3-control grid ($K = 32 \times 32$ elements) error for $p = 8$ is plotted in Fig. 8.4d and is nearly identical to the adaptive error. Adaptivity clearly provides a significant savings in the number of d.o.f. required for a given accuracy. Indeed, the single processor time savings is significant too; we find that $(T_{\text{control}}/T_{\text{adaptive}} \approx 10$ for most p.

Note that when we set $\nu = 0$ for this problem, we obtain energy conservation to about six digits for the $\ell_{\max} = 3$ adaptive case, and to about seven digits in the $\ell_{\max} = 3$ control run, up to $t_f = L/|\vec{c}| = 1$.

8.2.3 2D Burgers equation

We now examine the nonlinear ($\vec{c} = \bar{u}$) version of (8.1). The goal is to investigate the solution errors as the mesh resolves and tracks the stationary or propagating fronts generated and sustained by the nonlinear coupling of the system. We introduce a class of exact 2D solutions as follows. Note that any d solutions $q^\mu(y,t)$ to the 1D Burgers equation can be cast into d dimensions by substituting

$$\bar{u}(\vec{x},t) = \sum_{\mu=1}^{d} \vec{\kappa}^\mu q^\mu(\vec{\kappa}^\mu \cdot \vec{x}, \vec{\kappa}^\mu \cdot \vec{\kappa}^\mu t), \qquad \text{where} \quad \vec{\kappa}^\mu \cdot \vec{\kappa}^{\mu'} := \vec{\kappa}^\mu \cdot \vec{\kappa}^\mu \delta^{\mu,\mu'}, \tag{8.16}$$

into (8.1) [14]. If q^μ has period Y^μ w.r.t. y, then taking integer $2\kappa^{\mu,\mu'}/Y^\mu$ makes periodic boundary conditions for $\vec{x} \in [-1,1]^d$ appropriate. An initial condition (8.3) for a kind of straight $\vec{\kappa}^\mu$-perpendicular front is derived from

$$q^\mu(y,0) := -\sin(\pi y) + \hat{u}_2^\mu \sin(2\pi y), \tag{8.17}$$

The first problem is the classical Burgers stationary front, which is compared with and without adaptivity to previous results. The second problem will

consider the vector nature of (8.1) by simulating the collision of two oppositely translating oblique fronts. The third case is a curved front, i.e., a propagating radial N-wave.

Stationary Burgers front

The stationary Burgers front is the classical solution to (8.1), exhibiting a straight front developing across the x^1 direction. We compare with analytic values the maximum derivative magnitude $|\partial_{x^1} u^1|_{\max}$ and the time $t_{\max}$ at which the maximum occurs. To compare with the literature [2], we set $\nu = 0.01/\pi$, $\hat{u}_2^\mu = 0$ and $\vec{\kappa}^\mu = \vec{e}^{\,1}\delta^{\mu,1}$. The problem is initialized with $K = 4 \times 1$ grid of a specified degree p. A BDF3/Ext3 scheme is used for the time-derivative and advective terms, respectively. We initialize from (8.17) only at $t = t^0$, and integrate using a BDFM/Ext$M$ scheme to provide values at t^M ($M = 1, 2$). A nonadaptive and an adaptive case with maximum refinement $\ell_{\max} = 3$ are considered. In the nonadaptive case, the element edges lie along $x^1 = 0, \pm 0.05, \pm 1$, whereas in the adaptive case, the elements are initially uniform. The second-derivative error criterion is used in this problem applied to $\bar{u}$, and the threshold and coarsening multiplier are $\varepsilon_{\mathrm{t}} = 1$ and $\gamma_{\mathrm{c}} = 0.5$, respectively.

Table 8.2a presents the nonadaptive results from GASpAR and from [27]. Besides the comparison in Tables 8.2a and 8.2b, we obtained analytic solutions using (8.16) combined with the 1D formula [38, (4.10)] computed using Gauss-Hermite quadrature, and verified $|\partial_{x^1} u^1|_{\max}$ to seven digits against the reported value [2]. Thus, we have also verified that the $\mathbb{L}_2$ accuracy of the solution is consistent with the derivative accuracy implied by Tables 8.2a and 8.2b. We note that the $p = 5$ case is comparatively poor [cf. 27], possibly due to differences between the basis functions in the two methods [2], but our nonadaptive errors in $t_{\max}$ for our case are consistently better, while for $p > 5$ the $|\partial_{x^1} u^1|_{\max}$ errors are comparable [cf. 27].

Table 8.2b shows the results from the adaptive case and the reference and control solutions, where *reference* refers to a solution on a nonadaptive grid with K fixed as at the adaptive solution at $t = t_{\max}$. Thus, it offers a solution computed with roughly as many d.o.f. as the adaptive solution, and hence requiring about the same computational effort, disregarding adaptivity

Table 8.2: For the stationary Burgers front: (a) nonadaptive results; (b) adaptive, reference, and control results. The analytic solution [2] is denoted by $p = \infty$.

(a)

	Mavriplis [27]		GASpAR	
p	$t_{\max}$	$\lvert\partial_x u\rvert_{\max}$	$t_{\max}$	$\lvert\partial_{x^1} u^1\rvert_{\max}$
5	0.53745	167.227	0.5320	228.38977
9	0.50611	154.019	0.51074	148.04258
13	0.51103	151.496	0.51072	151.69874
17	0.51071	152.076	0.51045	152.09104
21	0.51023	152.004	0.51047	151.99624
∞	0.51047	152.00516		

(b)

	adaptive		reference		control	
p	$t_{\max}$	$\lvert\partial_{x^1} u^1\rvert_{\max}$	$t_{\max}$	$\lvert\partial_{x^1} u^1\rvert_{\max}$	$t_{\max}$	$\lvert\partial_{x^1} u^1\rvert_{\max}$
5	0.52679	224.36164	—	—	0.52674	224.37214
9	0.51095	153.39634	0.52635	227.53596	0.51095	153.39633
13	0.51030	150.03130	0.51219	181.02024	0.51030	150.03130
17	0.51048	152.25110	0.51082	149.57372	0.51048	152.25110
21	0.51047	152.00556	0.51021	147.22940	0.51047	152.00565
∞	0.51047	152.00516				

overhead. Clearly, resolving the front is very challenging as evidenced by the reference solution for $p = 5$ actually diverging, and good solutions not being obtained until $p > 13$. The control solutions are all nearly identical to the adaptive ones, suggesting that our refinement criteria enable DARe to capture the formation of the front accurately, at a significantly reduced number of d.o.f.. Indeed, on one processor, the computational times for the DARe cases are also reduced by a factor of about 7 compared with the control runs. Keeping in mind that on a single processor, no load balancing is required, we do not expect this level of efficiency for most turbulence problems. However, for the case where we are resolving largely isolated structures in an otherwise noisy background, we expect to see significant reductions in overall computational costs using DARe.

N-wave problem

The radial N-wave solution combines a d-dimensional Cole-Hopf transformation of (8.1) and a heat-eq. solution [generalizing 38, (4.6) & (4.40)]

$$\bar{u} = -2\nu\vec{\nabla}\ln\chi \qquad \longleftarrow \qquad \chi(\vec{x},t) = 1 + \frac{a}{t^{d/2}}\exp -\frac{(\vec{x}-\vec{x}^{\,0})^2}{4\nu t}, \qquad (8.18)$$

The N-wave emanates from $\vec{x}^0 = (\vec{e}^1 + \vec{e}^2)/2$. For this test, we initialize at $t^0 = 5 \times 10^{-2}$ and set $\nu = 5 \times 10^{-3}$ and $a = 10^4$. Dirichlet boundary conditions (8.2) on $\mathbb{D} = [0,1]^2$ are imposed at each time by evaluating (8.18) on $\partial\mathbb{D}$. The initial grid has $K = 4 \times 4$ elements, and we consider only the adaptive case with $\ell_{\max} = 4$. The refinement criteria are the same as in §8.2.2.

Fig. 8.5 presents six snapshots of the u^1 component of a typical N-wave system numerical solution, and illustrates the refinement patterns characteristic of all the runs. The solution has reflection symmetries, so for simplicity only one quadrant is shown. As the semicircular front propagates outward, the mesh refines to track it; while in the center the velocity components grow more planar, and the mesh coarsens. The front does not steepen in this problem, as it does in the planar front problem (§8.2.3); it simply decays as it propagates outward.

We set $p = 14$ and advance from $t = t^0$ to $t_\mathrm{f} = 0.11$ for various constant Δt to produce the timestep error-convergence curve in Fig. 8.6a. This time interval was enough to provide a number of DARe events; nevertheless, the solution converges with Δt, at order (slope) 3.01.

To check spatial convergence, the solution is advanced from $t = t^0$ to $t_\mathrm{f} = 0.11$ by using variable p and Δt (8.15) but fixed $\mathsf{Co} = 0.15$. Figure 8.6b shows the final $\mathbb{L}_2$ error vs p. As with the linear advection case, the error behaves spectrally for a finite time integration.

Colliding front problem

Here we take (8.17) with $\hat{u}_2^\mu = \frac{1}{2}\delta^{\mu,1}$, an initial condition that develops two translating, colliding fronts, and use (8.16) to get a 2D bi-periodic vector solution to the system (8.1). We retain $\nu = 0.01/\pi$, and to set the fronts oblique to the axes put $\vec{\kappa}^\mu = (\vec{e}^1 + 2\vec{e}^2)\delta^{\mu,1}$. The mesh initially has $K = 4 \times 4$ elements of degree $p = 8$. Initialization is as in §8.2.3, except that we use a BDF2/Ext2 scheme for time integration. The second-derivative error criterion is used in this test, with threshold and coarsening multiplier $\varepsilon_\mathrm{t} = 8$ and $\gamma_\mathrm{c} = 0.2$, respectively. The maximum refinement is $\ell_{\max} = 5$. Because the mesh only has to resolve discrete fronts as they develop,

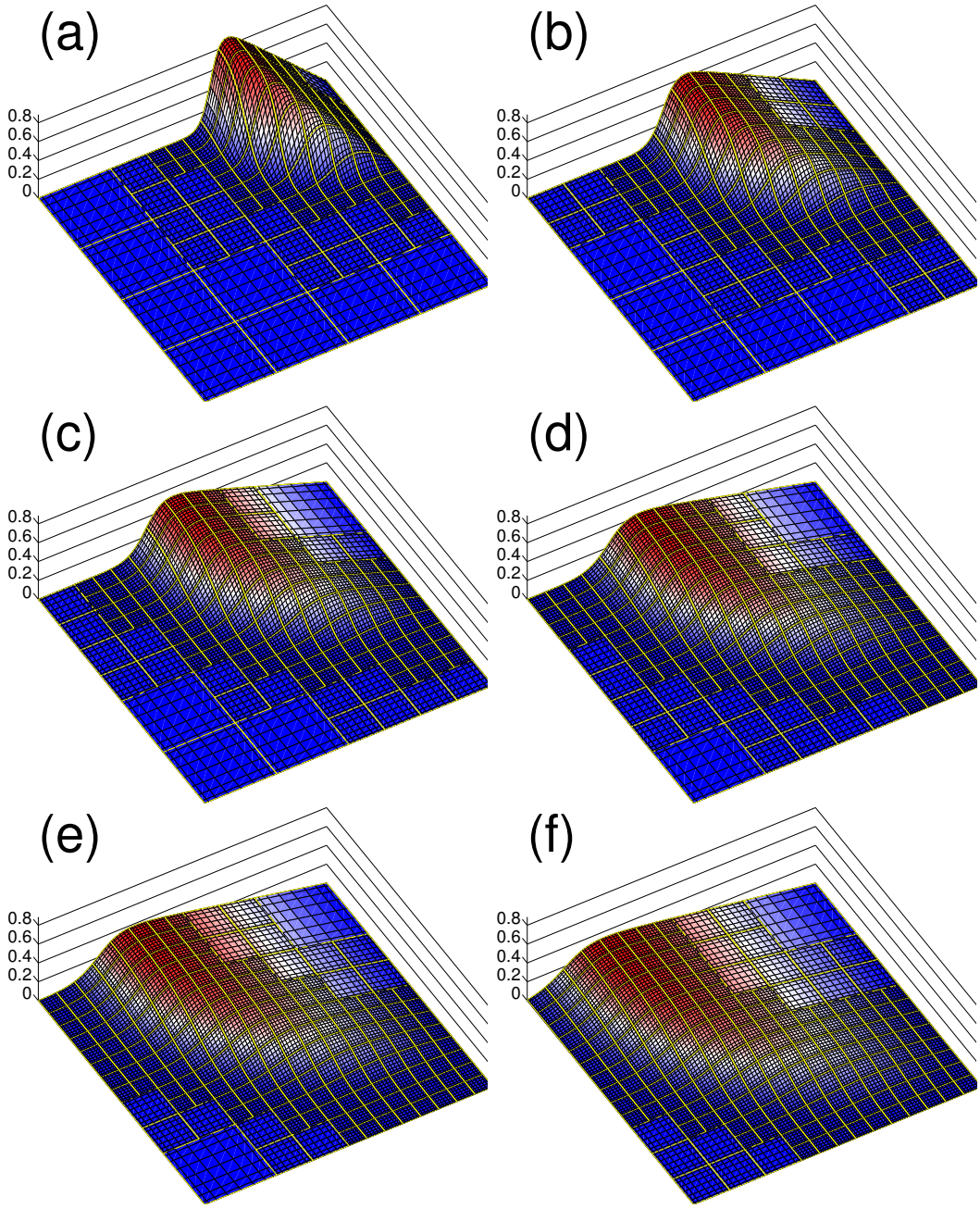

Figure 8.5: For the $p = 8$ adaptive radial N-wave solution of (8.1) with $\vec{c} = \bar{u}$ and $\nu = 5 \times 10^{-3}$, initialized by (8.18), surface plots of $u^1(\vec{x}, t)$, showing $\vec{x} \in [\frac{1}{2}, 1]^2$ and $K/4 = 88$, 121, 139, 172, 181, 190 as $t = 0.18$, 0.33, 0.48, 0.65, 0.81, 1.00. Black and yellow curves show nodes and element edges, respectively.

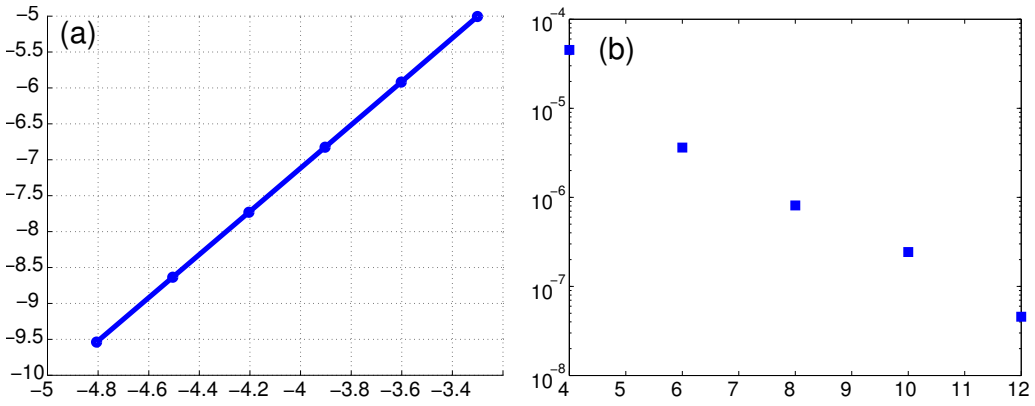

Figure 8.6: For the adaptive radial N-wave solution of (8.1) with $\vec{c} = \bar{u}$ and $\nu = 5 \times 10^{-3}$, initialized by (8.18), plots of $\log_{10}(||\bar{u}_{\rm n} - \bar{u}_{\rm a}||_2/||\bar{u}^0_{\rm a}||_2)$ vs: (a) $\log_{10} \Delta t$ for $p = 14$, with slope 3.01; and (b) p.

translate, merge and decay there is clear potential for *computational* savings by using adaptivity: simply reducing the number of elements on which to compute. Here, we wish to illustrate this potential and to verify that the error in the solution is consistent with the results in §8.2.3. We do not consider a control run for this problem.

In Fig. 8.7 are presented six snapshots during the evolution of the u^1 component of the colliding-fronts system, zoomed to one quadrant of the domain. The mesh refines around each of the oppositely-propagating fronts as they steepen, merge and begin to decay. The dash-dotted curve of Fig. 8.8 shows the number K of elements increasing monotonically before and during the merger, and decreasing, as expected, after the merger is complete at about $t = 0.12$. Moreover, Fig. 8.7 shows that DARe occurs only in regions localized around the steepening or translating fronts. The maximum number of adaptive elements is $\max_t K = 3136$, while the control solution would require $K = 16384$. This is a coverage fraction of about 19%, suggesting that adaptivity in this problem certainly offers a huge reduction in the required number of d.o.f..

Figure 8.8 provides the time series for the maximum-magnitude and $\mathbb{L}_2$ solution errors (unnormalized) of u^1, as well as the relative error of $|\vec{\kappa}^1 \cdot \vec{\nabla} u^1|$. The solution errors are reasonably well behaved. As expected, there is much

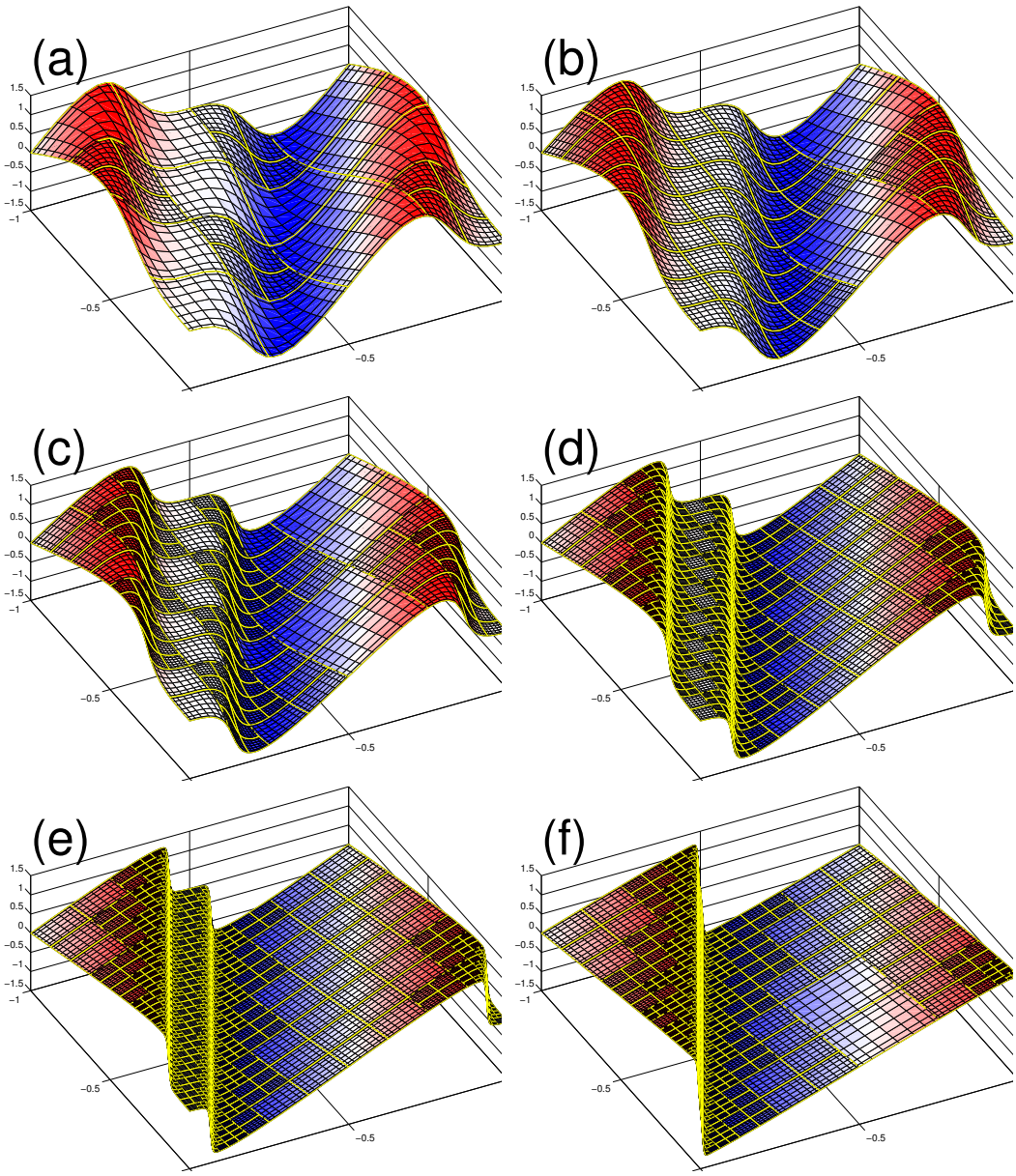

Figure 8.7: For the $p = 8$ adaptive colliding front solution of (8.1) with $\vec{c} = \bar{u}$ and $\nu = 10^{-2}/\pi$, initialized by (8.16) and (8.17) with $\vec{\kappa}^\mu = (\vec{e}^{\,1} + 2\vec{e}^{\,2})\delta^{\mu,1}$ and $\hat{u}_2^\mu = \frac{1}{2}\delta^{\mu,1}$, surface plots of u^1, showing $\vec{x} \in [-1, 0]^2$ and $K/4 = 28$, 52, 112, 352, 784, 481 at the time abscissas noted in Fig. 8.8. Black and yellow curves show nodes and element edges, respectively.

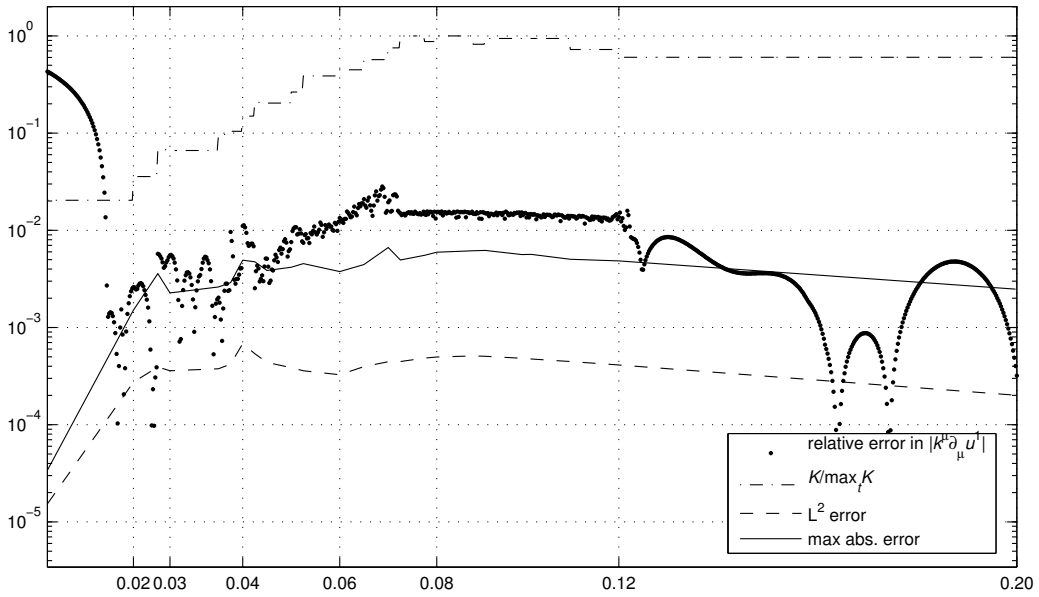

Figure 8.8: For the $p = 8$ adaptive solution of (8.1) with $\vec{c} = \bar{u}$ and $\nu = 10^{-2}/\pi$, initialized by (8.16) and (8.17) with $\vec{\kappa}^\mu = (\vec{e}^{\,1} + 2\vec{e}^{\,2})\delta^{\mu,1}$ and $\hat{u}_2^\mu = \frac{1}{2}\delta^{\mu,1}$, time series of fraction $K/\max_t K$ of elements (dash-dotted curve) and magnitude of relative maximum error in $|\vec{\kappa}^1 \cdot \vec{\nabla} u^1|$ (dot markers) vs time. Also shown are the maximum-absolute (solid curve) and $\mathbb{L}_2$ (dashed curve) errors for u^1. The abscissa is marked at the six times of Fig. 8.7.

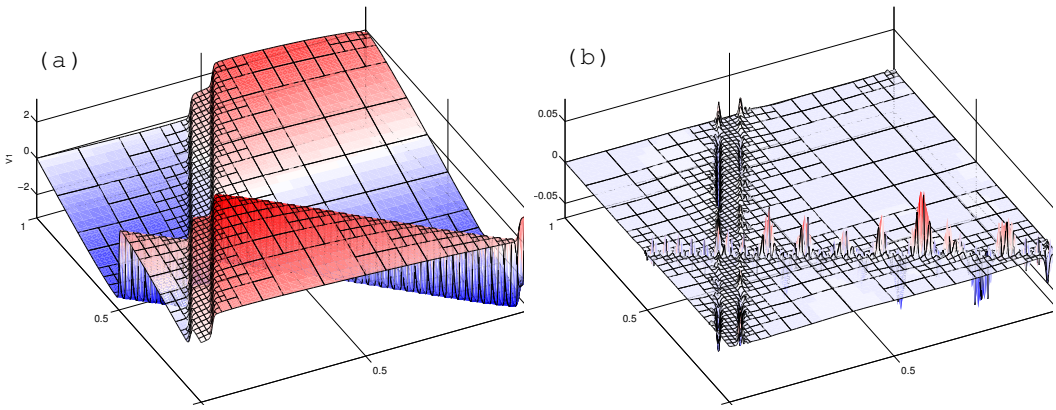

Figure 8.9: For the $p = 6$ adaptive double colliding fronts solution of (8.1) with $\vec{c} = \bar{u}$ and $\nu = 10^{-2}/\pi$, initialized by (8.16) and (8.17) with $\vec{\kappa}^1 = \vec{e}^{\,1} + 2\vec{e}^{\,2}$, $\vec{\kappa}^2 = \vec{e}^{\,2} - 2\vec{e}^{\,1}$, $\hat{u}_2^1 = \frac{1}{2}$ and $\hat{u}_2^2 = 0$, surface plots of (a) $u_{\rm n}^1$ and (b) $(u_{\rm n}^1 - u_{\rm a}^1)/||u_{\rm n}^1||_\infty$, showing $\vec{x} \in [0,1]^2$ and $K/4 = 1018$ at $t = 0.10$. For clarity, the node lines are not shown, but the element boundaries are now black.

more variation of the derivative error. The analytic values for $|\partial_{x^1} u^1|_{\max}$ and the time, $t_{\max}$ at which this maximum occurs are 213 and 0.1280, respectively. From our results, we find that $|\partial_{x^1} u^1|_{\max} = 222$ and $t_{\max} = 0.1283$, which is entirely consistent with the stationary results presented in Table 8.2b.

Finally, Fig. 8.9 shows a snapshot solution and relative error field of an even more challenging problem, namely the same two colliding fronts orthogonally crossed by a stationary front. Also, to better exercise h-refinement, the degree was reduced to $p = 6$ from $p = 8$ in the previous test. The reduction in overall accuracy is consistent with the p-convergence results in Fig. 8.6b. The relative $\mathbb{L}_2$ error is $||u_{\rm n}^1 - u_{\rm a}^1||_2/||u_{\rm n}^1||_2 = 5.8 \times 10^{-3}$. The element distribution in Fig. 8.9b shows that the error estimation coincides well with the actual point-wise error field.

Appendix A

Sample GASpAR parameter file

This appendix contains a complete list of parameters currently allowed in a start-up parameter file. The keywords on the left within a block may be modified easily by changing the ParamReader configuration in the **CmdFileParse()** method in the **gaspar_t.cpp** module.

```
GASpAR_MAIN
{
//_____________________________________________________________________________________________
// General time and output control
//_____________________________________________________________________________________________
  Do_Restart                          : 0;          //0->no restart; 1->restart
  Restart_Using_File_Name             : gaspar.dmp; // Used if Do_Restart = 1
  Output_Time_Specification           : 0;          // 0->Time-based; 1->Cycle-based
  Output_Time_Begin                   : 0.0;        // Begin output at this time
  Output_Time_End                     : 20.0 ;      // End output at this time (also run termination time)
  Output_Time_Delta                   : 0.1 ;       // Output at this time interval
  Output_Cycle_Begin                  : 0;          // Begin output at this cycle number
  Output_Cycle_End                    : 13000;      // End output, and run, at this cycle number
  Output_Cycle_Delta                  : 100;        // Output at this cycle interval
  Logging_Cycle_Delta                 : 10;         // Do logging at this cycle interval
  Dump_Cycle_Delta                    : 100;        // Create dump files at this cycle interval
  AMR_Cycle_Delta                     : 100;        // Perform AMR at this cycle interval
  Output_File_Prefix                  : tst;        // Use this as the output file prefix, and tag with cycle no.
  Use_Fixed_Timestep                  : 1  ;        // Use fixed timestep?
  Timestep                            : 5e-4;       // Set if Use_Fixed_Timestep = 1;
//_____________________________________________________________________________________________
// Mesh and solver specifications
//_____________________________________________________________________________________________
```

```
  Is_External_Mesh                     : 1;          // Flag indicating external mesh
  Mesh_File_Name                       : mesh.dat;   // Mesh data file
  Log_File_Name                        : gaspar.log; // Default log file
  Default_Dump_File_Name               : gaspar.dmp; // Default dump file name
  Scale_TimeStep                       : 0;          // Scale timestep if fixed (0 or 1)?
  Courant_Number                       : 0.2;        // Courant number for variable timestep
  Use_V-Preconditioner                 : 1;          // Vel. precond. if == 1
  Use_P-Preconditioner                 : 1;          // Press. precond. if == 1
  V-Preconditioner_Type                : 2;          // 0=GPC_BLOCKJAC_HELM; 2=GPC_POINTJAC_HELM
  Stokes_Solver_Splting_Type           : 0;          // 0=SCHUR_DELP; 1=CLASSICAL_UZAWA
  Time_Evolution_Scheme                : 2;          // 0=OIFS; 1=ABBDF; 2=EXBDF; 3=AMFE; 4=RKK
  Time_Derivative_Order                : 2;          // Time derivative order for now
  Advection_Order                      : 2;          // AB or EXT order dep. on Time_Evolution_Scheme
  Max_Number_of_Velocity_Iterations    : 256;
  Max_Number_of_Pressure_Iterations    : 256;
  Velocity_Solver_Tolerance            : 1.0e-15;
  Pressure_Tolerance                   : 1.0e-15;
//_____________________________________________________________________________________
// AMR a-posteriori check types, tolerances
//_____________________________________________________________________________________
  Do_Spectral_Check_AP                   : 0;      // Use spectral error method
  Do_1Derivative_Check_AP                : 0;      // Use 1-deriv. in x, y, or z
  Do_2Derivative_Check_AP                : 1;      // Use 2-deriv in x, y, z
  a-Posteriori_Tolerance                 : 1.0e-2; // Refinement tolerance
  a-Posteriori_ToleranceMult             : 1.0e-3 ;// Multiplies tol to get coarsening tol.
  Decay_Rate_Tolerance                   : 1.0;    // Decay rate tolerance
  Num_Spectral_Fit_Points                : 4  ;    // Number spectral fit points
  a-Posteriori_ToleranceD1               : 0.5;    // Refine tolerance on 1st derivative
  a-Posteriori_ToleranceMultD1           : 0.5 ;   // Multiplies 1nd deriv. tol to get coarsening tolerance
  a-Posteriori_ToleranceD2               : 1.0;    // Refine tolerance on 2nd derivative
  a-Posteriori_ToleranceMultD2           : 0.5 ;   // Multiplies 2nd deriv. tol to get coarsening tolerance
}

GASpAR_AUX
{
  1-Viscosity                          : 1.0e-4;      // viscosity in the x-direction
  2-Viscosity                          : 1.0e-4;      // viscosity in the y-direction
  Density                              : 1.0;         // Scalar density
  User_Parameter_File_Name             : gaspar.user; // Specify user parameter file
  User_Parameter_Block_File_Name       : UserBlk;     // User parameter block w/in user parameter file
}
```

Appendix B

Preprocessor definitions

Presented here is a complete list of preprocessor definitions (set using #define directives).

BURGERS	:	If defined, compiles code for Burger's equation.
NS	:	If defined, compiles code for Navier Stokes equation.
_LINUX	:	Compiles code specific to Linux operating system.
_AIX	:	Compiles code specific to the AIX operating system.
_INT32	:	Compiles code specific to a 32-bit platform.
_INT64	:	Compiles code specific to a 64-bit platform.
IS2D	:	Compiles code required for a 2d run.
IS3D	:	Compiles code required for a 3d run.
MPI_GENERIC_DEFAULT	:	If defined, compiles using MPI library commands.
G_MPI1	:	If defined, compiles using MPI-1 functions. There are few of these.
MPI_IO_DEFAULT	:	If defined, compiles using MPI_IO library commands.

Appendix C

Spectral-element formalism

In this appendix we summarize results from the SEM literature, and our notation. Table C.1 shows the hierarchy of basic formulas progressing from one 1D element, through K^1 1D elements, to K d-dimensional elements. Any dependent variable $u = u(\xi)$ may be approximated by its projection $\mathcal{P}_p u$ on the space $\mathbb{V}_p$ of polynomials of degree p, using u-values on any $p+1$ distinct nodal points ξ_j:

$$u = \mathcal{P}_p u + \mathcal{E}_p u \approx \mathcal{P}_p u := \sum_{j=0}^{p} u(\xi_j)\phi_j, \tag{C.1}$$

where $\mathcal{E}_p u$ is the point wise error and $\phi_j(\xi) := \prod_{j' \neq j}(\xi - \xi_{j'})/(\xi_j - \xi_{j'})$ denotes the Lagrange interpolating polynomials. Taking ξ_j and w_j from Table C.1 implies the quadrature

$$\langle u \rangle_1 := \int_{-1}^{1} u(\xi)\mathrm{d}\xi = \sum_{j=0}^{p} w_j u(\xi_j) + \mathcal{R}_p u(\xi'), \tag{C.2}$$

where $\mathcal{R}_p := -2^{2p+1}\frac{p^3(p+1)(p-1)!^4}{(2p+1)(2p)!^3}(\mathrm{d}/\mathrm{d}\xi)^{2p}$ is the residual operator [36] and $\xi' \in \,]-1, 1[$. Then the mean-square error is bounded as

$$\langle (\mathcal{E}_p u)^2 \rangle_1 \propto p^{1-2Q} \sum_{q=0}^{Q} \langle u^{(q)2} \rangle_1 \tag{C.3}$$

Table C.1: Hierarchy of spectral-element formulas, where L_j is the standard Legendre polynomial of degree j and norm $(j+\frac{1}{2})^{-\frac{1}{2}}$, $f\circ g(x) := f(g(x))$ and $1_{\mathbb{S}}(x) := \left\{ {1 \ (x\in\mathbb{S})} \atop {0 \ (\text{else})} \right.$.

Domain:	$\xi \in [-1,1]$;
	$x \in [-1,1] = \bigcup_{k=1}^{K^1} \bar{\mathbb{E}}^1_k$, where $]x_{k-1}, x_k[\equiv \mathbb{E}^1_k := \vartheta_k(]-1,1[)$ has length $h^1_k := x_k - x_{k-1} > 0 \Longrightarrow \mathbb{E}^1_k \bigcap \mathbb{E}^1_{k'} = \varnothing$ if $k \neq k'$;
	$\vec{x} \in \bar{\mathbb{D}} = \bigcup_{k=1}^{K} \bar{\mathbb{E}}_k$, where $\mathbb{E}_k := \vec{\vartheta}_k(]-1,1[^d)$ has diameter $h_k := \max_\mu \max_{\vec{x},\vec{x}'\in\bar{\mathbb{E}}_k} \lvert x^\mu - x'^\mu \rvert$ and $\mathbb{E}_k \bigcap \mathbb{E}_{k'} = \varnothing$ if $k \neq k'$.
Nodes:	$\xi_j :=$ $(j+1)$th least root of $(1-\xi^2)\frac{\mathrm{d}}{\mathrm{d}\xi}\mathrm{L}_p$;
	$x_{j,k} := \vartheta_k(\xi_j)$, where $\vartheta_k(\xi) := x_{k-1} + \frac{1}{2}h^1_k(1+\xi)$, $k \in \{1, \cdots K^1\}$;
	$\vec{x}_{\vec{\jmath},k} := \vec{\vartheta}_k(\vec{\xi}_{\vec{\jmath}})$, where $\xi^\mu_{\vec{\jmath}} := \xi_{\jmath^\mu}$ and $\vec{\vartheta}_k(\vec{\xi})$ is invertible but not necessarily linear.
Weights:	$w_j := 2/p(p+1)\mathrm{L}_p(\xi_j)^2$;
	$w_{j,k} := \lvert\frac{\mathrm{d}}{\mathrm{d}\xi}\vartheta_k(\xi_j)\rvert w_j$;
	$w_{\vec{\jmath},k} := \lvert \det \vec{\nabla}_{\vec{\xi}} \vec{\vartheta}_k(\vec{\xi}_{\vec{\jmath}})\rvert \prod_{\mu=1}^d w_{\jmath^\mu}$.
Basis:	$\phi_{j'}(\xi) = w_{j'} \sum_{j=0}^p \mathrm{L}_j(\xi_{j'})\mathrm{L}_j(\xi) / \sum_{j''=0}^p w_{j''}\mathrm{L}_j(\xi_{j''})^2 \xrightarrow[\xi\to\xi_{j''}]{} \delta_{j,j''}$;
	$\phi_{j,k}(x) := 1_{\bar{\mathbb{E}}^1_k}(x)\phi_j \circ \vartheta_k^{-1}(x) \xrightarrow[x\to x_{j',k'}]{} \begin{cases} 1, & x_{j,k} = x_{j',k'}, \\ 0, & \text{otherwise;} \end{cases}$
	$\phi_{\vec{\jmath},k}(\vec{x}) := 1_{\bar{\mathbb{E}}_k}(\vec{x})\phi_{\vec{\jmath}} \circ \vec{\vartheta}_k^{-1}(\vec{x}) \xrightarrow[\vec{x}\to\vec{x}_{\vec{\jmath}',k'}]{} \begin{cases} 1, & \vec{x}_{\vec{\jmath},k} = \vec{x}_{\vec{\jmath}',k'}, \\ 0, & \text{otherwise,} \end{cases}$ where $\phi_{\vec{\jmath}}(\vec{\xi}) := \prod_{\mu=1}^d \phi_{\jmath^\mu}(\xi^\mu)$.

for any order Q of square-integrable derivative [9, (B.3.59)]. Thus if u is infinitely smooth then $\mathcal{P}_p u$ converges to u *spectrally*.

Now let $[-1,1]$ be covered by K^1 disjoint 1D elements $\mathbb{E}^1_k$ as in Table C.1 (noting that nonlinear invertible ϑ_k may sometimes be preferable). Then u may be approximated by its projections $\mathcal{P}_{k,p}u$ on the space $\mathbb{V}_{\boldsymbol{h}^1,p}$ of piecewise polynomials of degree p on the $\mathbb{E}^1_k$. That is, (C.1) generalizes to

$$u = \sum_{k=1}^{K^1} \left(\mathcal{P}_{k,p}u + \mathcal{E}_{k,p}u\right), \qquad \mathcal{P}_{k,p}u := \sum_{j=0}^{p} u(x_{j,k})\phi_{j,k}, \tag{C.4}$$

where $\mathcal{E}_{k,p}u := \mathcal{E}_p(u\circ\vartheta_k)\circ\vartheta_k^{-1}$. Then (C.2) generalizes to

$$\langle u\rangle_1 = \sum_{k=1}^{K^1}\int_{x_{k-1}}^{x_k} u(x)\mathrm{d}x, \quad \int_{x_{k-1}}^{x_k} u(x)\mathrm{d}x = \sum_{j=0}^{p} w_{j,k}u(x_{j,k}) + \mathcal{R}_{k,p}u(x'_k), \tag{C.5}$$

where $\mathcal{R}_{k,p}u := (h^1_k/2)^{2p+1}\mathcal{R}_p(u\circ\vartheta_k)\circ\vartheta_k^{-1}$ and $x'_k \in \mathbb{E}^1_k$.

Generalizing further, assume a d-dimensional problem domain $\mathbb{D}$ can be partitioned as in Table C.1. Now generalizing (C.4), one may approximate a field $u(\vec{x})$ by its projections $\mathcal{P}_{k,\vec{p}}u$ on the space $\mathbb{V}_{\boldsymbol{h},\vec{p}}$ of piecewise polynomials of degree p^μ in coordinate x^μ on the $\mathbb{E}_k$. That is, (C.4) generalizes to

$$u \approx \mathcal{P}_{\boldsymbol{h},\vec{p}}u := \sum_{k=1}^{K} \mathcal{P}_{k,\vec{p}}u, \qquad \mathcal{P}_{k,\vec{p}}u := \sum_{\vec{\jmath}\in\mathbb{J}} u(\vec{x}_{\vec{\jmath},k})\phi_{\vec{\jmath},k}, \tag{C.6}$$

where $\mathbb{J} := \left\{\vec{\jmath} \;\middle|\; \jmath^\mu \in \{0,\cdots p^\mu\}\right\}$. The appropriate approximation of a vector

$$\bar{u} = \sum_{\mu=1}^{d} u^\mu \vec{e}^{\,\mu} \approx \mathcal{P}_{\boldsymbol{h},\vec{p}}\bar{u} = \vec{\boldsymbol{\phi}}^{\mathrm{T}}\boldsymbol{u}$$

uses $\vec{\boldsymbol{\phi}}$ with entries $\vec{\phi}^{\,\mu}_{\vec{\jmath},k} := \phi_{\vec{\jmath},k}\vec{e}^{\,\mu}$ and $\boldsymbol{u}$ with entries $u^\mu_{\vec{\jmath},k} := u^\mu(\vec{x}_{\vec{\jmath},k})$, where $\vec{e}^{\,\mu}$ denotes the Cartesian unit vectors. For scalars u (C.5) generalizes to

$$\begin{aligned}\langle u\rangle &:= \int_{\mathbb{D}}\cdots\int u(\vec{x})\mathrm{d}^d\vec{x} = \sum_{k=1}^{K}\int_{\mathbb{E}_k}\cdots\int u(\vec{x})\mathrm{d}^d\vec{x}\\ &\approx \sum_{k=1}^{K}\sum_{\vec{\jmath}\in\mathbb{J}} w_{\vec{\jmath},k}u(\vec{x}_{\vec{\jmath},k}) =: \langle u\rangle_{\mathrm{GL}}.\end{aligned} \tag{C.7}$$

Finally, variational formulation depends on the inner product from (C.7):

$$\langle u, v\rangle := \langle uv\rangle \approx \sum_{k=1}^{K}\sum_{\vec{\jmath}\in\mathbb{J}} w_{\vec{\jmath},k}\, u(\vec{x}_{\vec{\jmath},k})\, v(\vec{x}_{\vec{\jmath},k}) =: \langle u, v\rangle_{\mathrm{GL}} \tag{C.8}$$

for scalars, $\langle \bar{u}, \bar{v}\rangle := \sum_{\mu=1}^{d}\langle u^{\mu}, v^{\mu}\rangle$ for vectors, $\langle \vec{\bar{u}}, \vec{\bar{v}}\rangle := \sum_{\mu,\mu'=1}^{d}\langle u^{\mu,\mu'}, v^{\mu,\mu'}\rangle$ for tensors, and so forth. This implies a norm $||u||_2 := \langle u, u\rangle_{\mathrm{GL}}^{1/2}$. The norm $||u||_\infty := \max_{k=1}^{K}\max_{\vec{\jmath}\in\mathbb{J}} |u(\vec{x}_{\vec{\jmath},k})|$ is also used.

Now define the function spaces

$$\mathbb{U}_{\vec{b}} := \left\{ \bar{u} = \sum_{\mu=1}^{d} u^{\mu}\vec{e}^{\,\mu} \;\middle|\; u^{\mu} \in \mathbb{H}^1(\mathbb{D})\ \forall\mu \quad \& \quad \bar{u} = \vec{b} \text{ on } \partial\mathbb{D} \right\}$$

$$\text{and} \qquad \mathbb{H}^1(\mathbb{D}) := \left\{ f \;\middle|\; f \in \mathbb{L}_2(\mathbb{D}) \quad \& \quad \partial_{x^{\mu}} f \in \mathbb{L}_2(\mathbb{D})\ \forall\mu \right\}.$$

Searching the piecewise polynomial subspace $\mathcal{P}_{\boldsymbol{h},\vec{p}}\mathbb{U}_{\vec{b}} \subsetneq \mathbb{U}_{\vec{b}}$ for a solution to (8.4) leads to (8.5), where

$$M^{\mu,\mu'}_{\vec{\jmath},\vec{\jmath}';k} := \langle \vec{\phi}^{\mu}_{\vec{\jmath},k}, \vec{\phi}^{\mu'}_{\vec{\jmath}',k}\rangle_{\mathrm{GL}} = \delta_{\vec{\jmath},\vec{\jmath}'}\delta^{\mu,\mu'} w_{\vec{\jmath},k}, \tag{C.9}$$

$$C^{\mu,\mu'}_{\vec{\jmath},\vec{\jmath}';k} := \langle \vec{\phi}^{\mu}_{\vec{\jmath},k}, \mathcal{C}\vec{\phi}^{\mu'}_{\vec{\jmath}',k}\rangle_{\mathrm{GL}} = \delta^{\mu,\mu'} w_{\vec{\jmath},k}\vec{c}_{\vec{\jmath},k}\cdot\vec{\nabla}\phi_{\vec{\jmath}',k}(\vec{x}_{\vec{\jmath},k}), \tag{C.10}$$

$$L^{\mu,\mu'}_{\vec{\jmath},\vec{\jmath}';k} := \langle \vec{\nabla}\vec{\phi}^{\mu}_{\vec{\jmath},k}, \vec{\nabla}\vec{\phi}^{\mu'}_{\vec{\jmath}',k}\rangle_{\mathrm{GL}} = \delta^{\mu,\mu'}\sum_{\vec{\jmath}''\in\mathbb{J}} w_{\vec{\jmath}'',k}\vec{\nabla}\phi_{\vec{\jmath},k}(\vec{x}_{\vec{\jmath}'',k})\cdot\vec{\nabla}\phi_{\vec{\jmath}',k}(\vec{x}_{\vec{\jmath}'',k}),$$

and $\vec{c}_{\vec{\jmath},k}(t) := \vec{c}(\vec{x}_{\vec{\jmath},k}, t)$. The matrix $\mathbf{L}$ for deformed $\mathbb{E}_k$ (nonlinear $\vec{\vartheta}_k$) can also be constructed [*e.g.* , 9], and is supported in GASpAR.

As an example of global assembly, for the mesh partition in Fig. 8.1a, (8.9) takes the following explicit form (suppressing zero-valued and $\mu > 1$ blocks):

$$\boldsymbol{u} = \begin{pmatrix} u_0 \\ \vdots \\ u_{17} \end{pmatrix} = \left(\begin{array}{c} u_{0,1} \\ \vdots \\ u_{8,1} \\ \hline u_{0,2} \\ \vdots \\ u_{8,2} \end{array}\right) = \left(\begin{array}{cccccccccccccccc} 1&0&0 & & & & & & & & & & & & & \\ 0&1&0 & & & & & & & & & & & & & \\ 0&0&1 & & & & & & & & & & & & & \\ & & & 1&0&0 & & & & & & & & & & \\ & & & 0&1&0 & & & & & & & & & & \\ & & & 0&0&1 & & & & & & & & & & \\ & & & & & & 1&0&0 & & & & & & & \\ & & & & & & 0&1&0 & & & & & & & \\ & & & & & & 0&0&1 & & & & & & & \\ \hline & & & & & & & & & & & & & & & \\ & & & & & & & & & & & & & & & \\ & & & & & & & & & & & & & & & \\ & & & & & & & & & & & & & & & \\ & & & & & & & & & & & & & & & \\ & & & & & & & & & & & & & & & \\ & & & & & & & & & & & & & & & \\ & & & & & & & & & & & & & & & \\ \hline 0&0&1 & 0&0&1 & & & & & & & & & & \\ 0&0&0 & 0&0&0 & & & & & & & & & & \\ & & & & & & 0&0&1 & & & & & & & \\ & & & & & & 0&0&0 & & & & & & & \\ & & & & & & & & & 0&0&1 & & & & \\ & & & & & & & & & 0&0&0 & & & & \\ & & & & & & & & & & & & 0&0&1 \\ & & & & & & & & & & & & 0&0&0 \end{array}\right) \begin{pmatrix} u_{\mathrm{g},0} \\ \vdots \\ u_{\mathrm{g},14} \end{pmatrix}.$$

For the mesh in Fig. 8.1b, the explicit form of (8.9) for the nonconforming assembly matrix $\mathbf{A} = \mathbf{\Phi}\mathbf{A}_\mathrm{c}$ is (suppressing zero-valued and $\mu > 1$ blocks)

$$\boldsymbol{u} = \begin{pmatrix} u_0 \\ \vdots \\ u_{26} \end{pmatrix} = \begin{pmatrix} u_{0,1} \\ \vdots \\ u_{8,1} \\ \hline u_{0,2} \\ \vdots \\ u_{8,2} \\ \hline u_{0,3} \\ \vdots \\ u_{8,3} \end{pmatrix} = \left(\begin{array}{ccccccccc} 1&0&0 & & & & & & \\ 0&1&0 & & & & & & \\ 0&0&1 & & & & & & \\ & & & 1&0&0 & & & \\ & & & 0&1&0 & & & \\ & & & 0&0&1 & & & \\ & & & & & & 1&0&0 \\ & & & & & & 0&1&0 \\ & & & & & & 0&0&1 \\ \hline 0&0&1 & & & & 0&0 & \\ 0&0&0 & & & & 1&0 & \\ 0&0&0 & & & & 0&1 & \\ 0&0&3/8 & 0&0&3/4 & 0&0&-1/8 \\ 0&0&0 & 0&0&0 & 0&0&0 \\ 0&0&0 & 0&0&0 & 0&0&0 \\ & & & 0&0&1 & & & \\ & & & 0&0&0 & & & \\ & & & 0&0&0 & & & \\ \hline & & & & & & 0&0 & \\ & & & & & & 1&0 & \\ & & & & & & 0&1 & \\ 0&0&-1/8 & 0&0&3/4 & 0&0&3/8 \\ 0&0&0 & 0&0&0 & 0&0&0 \\ 0&0&0 & 0&0&0 & 0&0&0 \end{array}\right) \begin{pmatrix} u_{\mathrm{g},0} \\ \vdots \\ u_{\mathrm{g},18} \end{pmatrix}$$

$$= \left(\begin{array}{ccccccccc} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ 0&0&3/8 & 0&0&3/4 & 0&0&-1/8 \\ 0&0&0 & 0&0&0 & 0&0&0 \\ & & & & & & 1 & & \\ 0&0&0 & 0&0&0 & 0&0&0 \\ & & & & & & & \ddots & \\ & & & & & & & & 1 \\ 0&0&-1/8 & 0&0&3/4 & 0&0&3/8 \\ 0&0&0 & 0&0&0 & 0&0&0 \\ & & & & & & & 1 & \\ 0&0&0 & 0&0&0 & 0&0&0 \\ & & & & & & & & \ddots \\ & & & & & & & & 1 \end{array}\right) \left(\begin{array}{ccccccccc} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ \hline 0&0&1 & & & & 0&0 & \\ 0&0&0 & & & & 1&0 & \\ 0&0&0 & & & & 0&1 & \\ & & & 1 & & & & & \\ & & & & 1 & & & & \\ 0&0&1 & & & & 0&0 & \\ 0&0&0 & & & & 1&0 & \\ 0&0&0 & & & & 0&1 & \\ \hline 0&0&1 & & & & 0&0 & \\ 0&0&0 & & & & 1&0 & \\ 0&0&0 & & & & 0&1 & \\ & & & & & 1 & & & \\ & & & & & & 1 & & \\ & & & 0&0&1 & & 0&0 \\ & & & 0&0&0 & & 1&0 \\ & & & 0&0&0 & & 0&1 \end{array}\right) \begin{pmatrix} u_{\mathrm{g},0} \\ \vdots \\ u_{\mathrm{g},18} \end{pmatrix}.$$

Note that the $\mathbf{A}$ entries corresponding to the child-node rows (see Fig. 8.1b) are not Boolean but that every row sum is unity. This result is to be expected because $\mathbf{A}$ must accommodate interpolation of a constant solution (*e.g.* , $u_{\mathrm{g},i} = 1\ \forall i$) across a nonconforming interface.

Appendix D

Contact information

If you would like to offer comments, or have questions or concerns, you may contact us using the following information. While we are not able to offer formal support to users at this time, we are very interested in your comments, and we will try to get back to you as soon as possible.

You may contact the authors by surface mail, phone or fax at

Turbulence Numerics Team,
Institute for Mathematics Applied to Geosciences
NCAR
P.O. Box 3000 Boulder, CO 80307-3000, USA
tele: 303 497 1636; FAX: 303 497 2180

In addition, you may contact us by email at the following addresses:

Duane Rosenberg : duaner@ucar.edu
Aime´ Fournier : fournier@ucar.edu
Annick Pouquet : pouquet@ucar.edu

Bibliography

[1] Anagnostou, G., Y. Maday, C. Mavriplis, and A. T. Patera, "On the mortar element method: Generalizations and implementation," in *Third International Symposium on Domain Decomposition Methods,* pp. 157–173, SIAM, (1989).

[2] Basdevant, C, Deville, M., Haldenwang, P., Lacroix, J. M., Ouazzani, J., Peyret, R., Orlandi, P, and Patera, A., "Spectral and finite difference solutions of the Burgers equation," *Comp. Fluids*, **14**, pp., 23–41 (1986).

[3] Belgacem, F. B., "The mixed mortar finite element method for the incompressible Stokes problem: Convergence analysis," *SIAM J. Numer. Anal.,* **37,**(4) pp. 1085-1100 (2000).

[4] Bernardi, C., Y. Maday, C. Mavriplis, and A. T. Patera, "The mortar element method applied to spectral discretizations", *Proceedings of the Seventh International Conference on Finite Element Methods in Flow Problems,*, T. J. Chung and G. R. Karr, eds., University of Alabama, Huntsville (1989)

[5] Chang, Rong-Yeu, Hsu, Chia-Hsiang, "A variable-order spectral element method for incompressible viscous flow simultaion". *Int. J. Num. Meth. Eng.*, **39**, 2865–2887, 1996.

[6] Casadei F., E. Gabellini, G. Fotia, F. Maggio, and A. Quarteroni, "A mortar spectral/finite element method for complex 2D and 3D elastodynamic problems," *Comp. Meth. Appl. Mech. Eng.*, **191**, 5119–5148 (2002).

[7] Chaljub, E., Y. Capdeville, and J. P. Vilotte, "Solving elastodynamics in a fluid-solid heterogeneous sphere: A parallel spectral element ap-

proximation on non-conforming grids," *J. Comp. Phys.*, **187**, 457–491 (2003).

[8] Dennis, J., Fournier, A., Spotz, W., St.-Cyr, A. Taylor, M., Thomas, S., and Tufo, H., *High Resolution Mesh Convergence Properties and Parallel Efficiency of a Spectral Element Atmospheric Dynamical Core, Int. J. High Perf. Computing Appl.*, **19** pp. 225–235 (2005).

[9] Deville, M. O., P. F. Fischer and E. H. Mund, *High-Order Methods for Incompressible Fluid Flow*. Cambridge, Cambridge University Press (2002).

[10] Dubois-Pelerin, Y., V. Van Kemenate, M. O. Deville, "An Object-Oriented Toolbox for Spectral Element Analysis", it J. Sci Comp., **14**, pp. 1-29 (1999)

[11] Elmegreen, B. G., & J. Scalo "Interstellar turbulence, I: Observations and Processes" *Ann. Rev. Astron. Astrophys.* **42**, 211–273 (2004).

[12] Feng, H. and C. Mavriplis, "Adaptive spectral element simulations of thin flame sheet deformations," *J. Sci. Comp.*, **17**, pp. 1–3 (2002).

[13] Fischer, P. F., G. W. Kruse, and F. Loth, "Spectral element methods for transitional flows in complex geometries," *J. Sci. Comput.,* **17**, 1, pp. 81–98 (2002).

[14] Fournier, A., G. Beylkin and V. Cheruvu, "Multiresolution adaptive space refinement in geophysical fluid dynamics simulation," *Lecture Notes Comp. Sci. Eng.,* **41,** pp. 161–170 (2005).

[15] Fournier, A., M. A. Taylor, and J. J. Tribbia, "The spectral element atmosphere model (SEAM): High-resolution parallel computation and localized resolution of regional dynamics," *Mon. Wea. Rev.*, **132**, pp. 726–748 (2004).

[16] Uriel Frisch, *Turbulence: The legacy of A.N. Kolmogorov*, Cambridge University Press, 1995.

[17] Henderson, R. D., "Unstructured spectral element methods for simulation of turbulent flows," *J. Comp. Phys.* **122**, pp. 191–217 (1995).

[18] Henderson, R. D., "Dynamic refinement algorithms for spectral element methods," *Comput. Methods Appl. Mech. Engrg.* **175**, pp. 395–411 (1999).

[19] Isihara T., Y. Kaneda, M. Yokokawa, K. Itakura, and A. Uno, "Spectra of energy dissipation, enstrophy and pressure by high-resolution direct numerical simulations of turbulence in a periodic box," *J. Phys. Soc. Japan* **72**, pp. 983–986 (2003).

[20] Iskandarani, M., D. B. Haidvogel, and J. C. Levin, "A three-dimensional spectral element model for the solution of the hydrostatic primitive equations" *J. Comp. Phys.* **186**, pp. 397–426 (2003).

[21] Karniadakis, G.E. and S.J. Sherwin, *Spectral/hp Element Methods for CFD*. New York, Oxford Iniversity Press (1999).

[22] Karniadakis, G.E., M. Israeli, and S.A. Orszag, "High–Order splitting methods for the incompressible Navier-Stokes equations " *J. Comp. Phys.* **97**, pp. 414–443 (1991).

[23] Kopriva D. A., S. L. Woodruff, and M. Y. Hussaini, "Computation of electromagnetic scattering with a non-conforming discontinuous spectral element method," *Int. J. Num. Meth. Eng.*, **53**, pp. 105–122 (2002).

[24] Kruse, G. W., "Parallel nonconforming spectral element solution of the incompressible Navier–Stokes equations in three dimensions," Ph.D. Dissertation, Division of Applied Mathematics, Brown University (1997).

[25] Levin, J. G., M. Iskandarani, and D. B. Haidvogel, "A nonfoncorming spectral element ocean model," *Intl. J. Numer. Meth. Fluids* **34**, pp. 495–525 (2000).

[26] Maday, Y., C. Mavriplis, and A. T. Patera, "Nonconforming mortar element methods: Application to spectral discretizations," in *Domain Decomposition Methods,* pp. 392–418, SIAM (1989). Also ICASE Report 88-59.

[27] Mavriplis, C., "Adaptive mesh strategies for the spectral element method," *Comput. Methods Appl. Mech, Engrg.* **116**, pp. 77–86 (1994).

[28] C. Meneveau and J. Katz, "Scale-invariance and turbulence models for large-eddy simulation," *Annu. Rev. Fluid Mech.* **32**, pp. 1–32 (2000).

[29] Patera, A., "A spectral element method for fluid dynamics: laminar flow in a channel expansion," *J. Comp. Phys.* **54**, pp. 468–488 (1984).

[30] Rønquist, E. "Convection treatment using spectral elements of different order," *Intl. J. Num. Meth. Fluids* **22**, pp. 241–264 (1996).

[31] Rosenberg, D., Fournier, A., Fischer, P., and Pouquet, A., "Geophysical-astrophysical spectral-element adaptive refinement (GASpAR): Object-oriented *h*-adaptive fluid dynamics simulation", *J. Comp. Phys., in press* (2005). Also available at www.arxiv.org: math.NA/0507402

[32] Sert, C. & Beskok, A., "Spectral element formulation on non-conforming grids: A comparative study of pointwise matching and integral projection methods", *J. Comp. Phys.*, in press (2005).

[33] Shewchuck, Richard J. "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain," http://www-2.cs.cmu.edu/jrs/jrspapers.html (1994).

[34] I. Sytine, D. Porter, P. Woodward, S. Hodson and K-H Winkler, "Convergence tests for the Piecewise Parabolic Method and Navier-Stokes solutions for homogeneous compressible turbulence", *J. Comp. Phys.* **158**, pp. 225–238 (2000).

[35] Tufo, H.M., Fischer, P.F., "Terascale Spectral element Algorithms and Implementations", Proceedings of the ACM/IEEE SC99 Conference on High Performance Networking and Computing, IEEE Computer Soc. (1999).

[36] Eric W. Weisstein. "Lobatto Quadrature." From MathWorld—A Wolfram Web Resource. http://mathworld.wolfram.com/LobattoQuadrature.html.

[37] Eric W. Weisstein. "Conjugate Gradient Method." From MathWorld—A Wolfram Web Resource. http://mathworld.wolfram.com/ConjugateGradientMethod.html.

[38] Whitham, G.B., *Linear and Nonlinear Waves* New York, Wiley (1974).