Documentation: Matlab code for DG solver for elastic and poroelastic wave propagation using the Hesthaven-Warburton library

Nick Dudley Ward and Simon Eveson

March 26, 2021

1 Introduction

We describe a Matlab implementation of a DG solver for 3 dimensional elastic and poroelastic wave propagation. This code has been developed using the Hesthaven and Warburton DG library that accompanies their text *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*, [1]. The current release includes, with permission, their DG library, version 1.1, [2]. Derivations for our implementation can be found available on arXiv, [3]:

A Discontinuous Galerkin method for three-dimensional poroelastic wave propagation: forward and adjoint problems (Nick Dudley Ward, Simon Eveson and Timo Lähivaara)

It is our intention that the code may be helpful to people who want to improve their skills with scientific coding beyond standard undergraduate courses. For this reason the code has been written with considerable care and we have provided extensive commentary in the code. (In our experience, very little code written to support articles is fit for general consumption.) While Matlab permits rapid code development, one soon reaches a stage when Matlab code becomes opaque due to the need to carry out, e.g., nested transposing and reshaping. (It is never much fun returning to such code that one has written, and presumably even less fun for others who are trying to figure out what you have done.)

The Hesthaven and Warburton book and accompanying Matlab library is a really excellent entry route for learning about the discontinuous Galerkin method. However, the set-up cost for the DG method is high and could require a large number of variables to be passed as function arguments (see Appendix C, p460 of their book). This would soon become cumbersome, so the HW library uses global variables: data describing the computational mesh and polynomial basis functions are stored in the global namespace and directly accessible to any function. In our code, to avoid potential conflicts with the global namespace, we used what turned out to be a very large Matlab structure `param` to store all the state information associated with our solver. This is discussed in detail below.

The main directory contains three sub-directories, viz:

- `src`
- `lib`
- `examples`

`src` contains all functions related with the DG solver, `lib` contains the Hesthaven-Warton library and `examples` contains various example problem files as well as a generic driver file:

driver_biot_3d.m. This driver file calls a specific problem file, carries out the various stages of assembly required by the DG library and then executes the principal solver function biot_3d.m. The assembly process for the DG solver is a little involved and requires three assembly stages. Each example problem file has therefore been split into three blocks:

- PRE_MESH_LOAD
- POST_MESH_LOAD
- POST_DOMAIN_SETUP

corresponding to the various stages of assembly. PRE_MESH_LOAD, for example, specifies quantities like the basis order and computational mesh to be used, so that that DG library can be initialised; this is described in detail in the next section. Through this process a structure called param is gradually enlarged until it contains enough information to run the main solver output = biot_3d(param) which appears on line 571 of the driver file! Note that param contains a lot of information (over 150 fields, depending on the problem). In addition to containing a large number of flags corresponding to the user-defined model set-up and solver choices, it also contains large matrices of physical parameters defined on an element or nodal basis as necessary. During the assembly process, wavespeeds (i.e. eigenvalues) and corresponding eigenvectors used to compute upwind fluxes are also stored on an element by element basis in param.

Note that computed quantities like the wave field and interpolated seismograms appear in another struct called output. The driver file permits computed values to be saved to an HDF5 file (https://www.hdfgroup.org/solutions/hdf5), and for a simulation to be restarted from the last computed values.

A wavefield computation is executed by calling the main solver function biot_3d which, despite its name, is a fairly generic Runge-Kutta (RK) loop. Upwind fluxes are computed by calling biot_rhs_3d.m; this function is a junction box that implements boundary conditions and splits the current state data into pieces according to element and face properties (for example, elastic or poroelastic). Appropriate functions to compute fluxes on face nodes are then called, for example, get_face_flux_ep.m computes the face fluxes on an elastic/poroelastic boundary.

We have implemented both a conservative velocity/strain formualtion and a non-conservative velocity/stress formulation; the latter we call an adjoint wavefield solver, since one may identify the velocity/stress formulation with the formal adjoint (see section 6.1 of our paper). This should not be confused with the adjoint method for computing gradients which we have also implemented. Thus one may choose whether to compute a wavefield using a velocity/strain param.run_forward = true or a velocity stress param.run_forward_adj = true formulation (or both, for comparison, say). Thus, to use the language of inverse problems, we have implemented two forward solvers or maps (which may be run backwards as well as forwards in time).

This release includes several specific problem files, which are probably the best point of entry to understanding our implementation. We have included problem files to repeat the convergence tests documented in our paper (comparing numerical solutions for a given basis order with analytic plane waves), as well as some small scale elastic and poroelastic examples with explosive sources. Notes on specific files are given in section 3.

It is a slightly frightening prospect to release code for public consumption, especially when the intention is to provide something of pedagogic value. We wrote the main bulk of this code in the first quarter of 2018 together with a parallel C implementation to carry out many of the computations for our paper. We then returned to the code in 2020 and carried out a major

overhaul of the code, mainly by Zoom, which has resulted in the current form. Unfortunately, we haven't bleen able to use the code in teaching and hence rigorously test it, and apologise in advance for any errors (we would certainly appreciate hearing about any). Obviously we could have developed the code further and made it more user-friendly; we are aware that we have not been 100% consistent with our conventions. But this is only worth doing in a classroom environment and being able to benefit from feedback. There is another reason for not developing the code too far: large scale computing requires distributed computing power, and Matlab simply isn't suited to that. It is excellent, though, for teaching and development.

On balance, it is perhaps better, from a pedagogical perspective, to intentionally leave the code in an imperfect state and simply wish the interested reader 'good luck!'.

1.1 Origin of work

Our interest in computational wave propagation, and especially poroelastic wave propagation, began shortly after the Canterbury earthquakes of 2010-11, when we began to think about using New Zealand's passive seismic activity to better quantify Canterbury's extensive groundwater system. The main city of Christchurch is built on a sequence of coastal artesian aquifers which are, surprisingly, not very accurately characterised. While we have largely moved away from using passive data, the general problem of mapping often poorly-resolved groundwater resources usually in very remote areas from active source tests remains a problem of considerable interest. This includes instrumentation, field testing and, of course, inversion. The current work, begun in 2015, and significantly delayed due to other commitments, is one more piece of this jigsaw.

2 General description of code

Figure 1 shows the dependencies between the main Matlab functions. These are described in detail below.

2.1 The point of entry: `driver_biot_3d.m`

The outermost level of the system is the Matlab function file `driver_biot_3d.m`. This is called by the user and makes one simulation run, which could be of the main PDE system or of its adjoint, with forward or backward time-stepping and a choice from several Runge-Kutta schemes. The function defined in this file takes one argument, the name of a Matlab function (for convenient interactive use in the IDE, this can also be declared at the top of the file). This function is normally contained in an m-file that we call a "problem file" and which specifies all the aspects of the required simulation: the mesh, the physical parameters, time ranges, Runge-Kutta scheme, etc. Typically, each simulation scenario is described in its own problem file, but there are ways to describe a collection of related problems in one file.

To understand the role of the problem file, we need to explain a little about how the system holds its configuration data. The state of the system at any moment before, during or after the simulation run is described by literally hundreds of pieces of data, ranging from Boolean flags to multi-dimensional arrays containing millions of entries. The Matlab code is, of course, broken down into many different functions, most of which require access to some of this state information. Passing the necessary data as individual function arguments would be extremely messy and error-prone, so a different approach is neeeded. The underlying Discontinuous Galerkin (DG) library (Hesthaven and Warburton) adopts one solution: global variables. In this

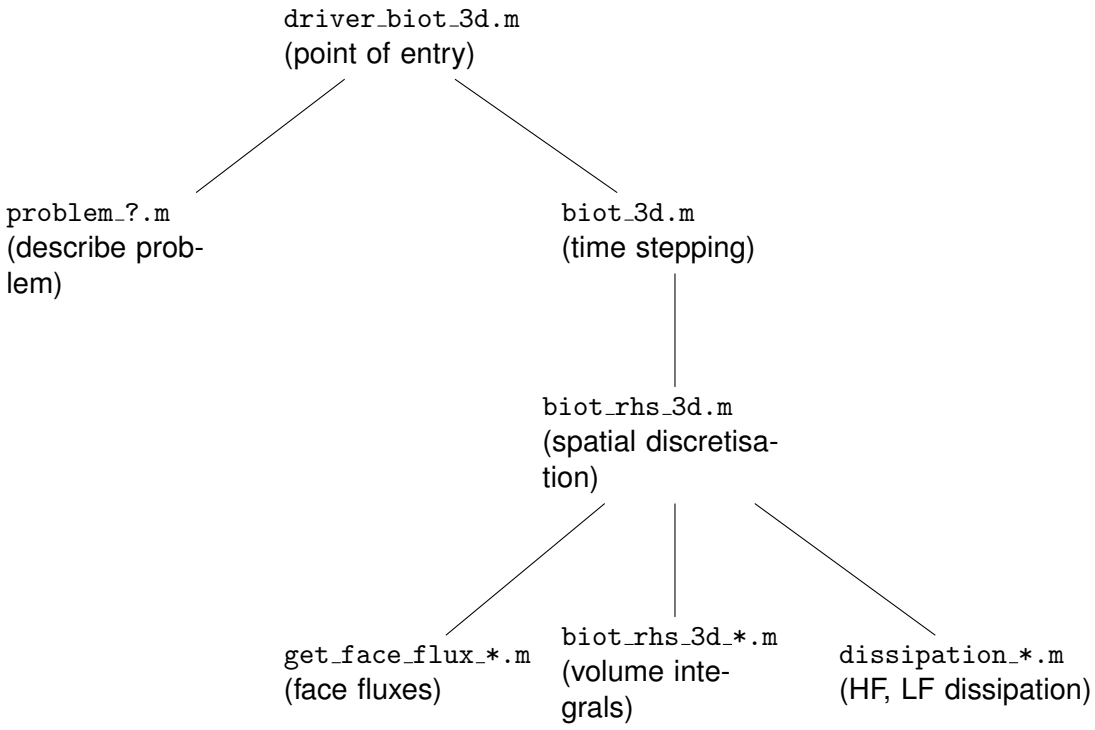

Figure 1: Dependencies between parts of the system

approach, data describing the computational mesh and the polynomial basis functions defined on its elements is declared in the global namespace, making it accessible to any function. This is effective as far as it goes, but major problems with name clashes can arise if two different systems (here, the Biot system and the DG library) both attempt to use the single global namespace. To avoid this, the Biot system stores (almost) all of its state information in a single Matlab struct, called `param`. This is initially declared in `driver_biot_3d.m` and is passed to most functions called from there; they, in turn, pass it on to functions that they call.

In this design, the process of describing the simulation to be run is effectively the same as the process of initialising the `param` structure. To avoid unnecessary duplication of code, the initialisation is broken down into three main stages, between which some standardised processing is performed by the system, to provide information needed for later stages. The sequence of events is as follows (the capitalised tokens are used in the problem file to distinguish the stages; the setup tasks listed are far from exhaustive):

problem file `(PRE_MESH_LOAD)` Specify the computational mesh, the degree of polynomials used in the DG setup, the time range to be simulated, the Runge-Kutta algorithm to be used for time-stepping and other global information about the simulation.

system Load mesh data and initialise DG library

problem file `(POST_MESH_LOAD)` Assign domain codes to elements and physical properties to domains

system Map physical properties from domains to elements, compute lots of derived quantities on elements and for the whole mesh.

problem file (POST_DOMAIN_SETUP) Specify initial values, boundary conditions, sources, receivers.

system Set up some further data structures associated with boundary conditions, sources and receivers. Calculate time step using CFL condition.

This is all implemented using the param struct mentioned above. At each setup stage, the function defined in the problem file is called with two arguments: a token representing the setup stage (PRE_MESH_LOAD, POST_MESH_LOAD, POST_DOMAIN_SETUP; these are actually numeric constants stored in param!) and the partially initialised param struct; its job is return an enlarged copy of the struct, with further data added. The system then uses this to add more data before invoking the problem file again . . . after the whole sequence above is finished, the param struct is complete.

There are also PRE_SIMULATION and POST_SIMULATION tokens; POST_SIMULATION, which is invoked with a third argument, another struct containing the results of the simulation, is useful for e.g. plotting graphs.

Once the simulation has been set up, it can be run. A good deal of housekeeping surrounds this (e.g. saving the setup for the parallel C framework, save-and-restore options for breaking the Matlab run into pieces, saving the output data at the end of the simulation) but, apart from this, control is now passed to biot_3d.m, which runs the main time-stepping loop.

2.2 The time-stepping loop: biot_3d.m

biot_3d.m is called by driver_biot_3d.m. It serves as an illustration of the use of the param struct: param is its only argument.

Despite its name, this file has little connection with the Biot model, or even with the DG method: it is, at heart, a fairly generic Runge-Kutta (RK) loop. This is because all the spatial discretisation is taken care of in biot_rhs_3d.m, which biot_3d.m calls; within biot_3d.m, we are simulating an ODE system with a large number of variables. To keep memory usage under control, only two-register low-storage RK schemes are implemented; both explicit (Carpenter and Kennedy 1994) and IMEX (Cavaglieri and Bewley 2015) are supported.

biot_3d.m begins with some housekeeping: strorage associated with the intermediate values (residuals) in the RK loop has to be initialised, the coefficients of the specified RK method must be set up (from param.RK_method, which can be set in the problem file to the name of a group in RK.h5, the data file holding the coefficients) and the the time-stepping data in param must be made consistent: typically, this involves reducing the time step (from the CFL condition) so an integer number of time steps matches the required time-integration width.

The Runge-Kutta loop then runs. Within this loop, each individual time step can follow one of several paths, depending on the way the potentially stiff low-frequency dissipation terms are handled.

The simplest option is to use a low-storage explicit RK (LSERK) scheme. Here, the loop is quite simple: at each stage of the scheme, biot_rhs_3d.m is called and the returned value is used to update the two registers representing the RK scheme. One extra complication is the use of “memory variables” to simulate dissipation in Biot’s high-frequency (HF) regime. Because these are defined only on HF elements, they are stored separately from the array containing the 13 fields in the main poroelastic model; from the point of view of the RK method, though, they are just more ODEs to simulate, so they can be time-stepped in exactly the same way as everything else. This leads to a certain amount of code duplication; it is very small, though, and one convenient Matlab feature works with us here. The obvious implementations of the register update formulae work vacuously with empty arrays, so there is no need to wrap

everything up in conditionals to test if there are any HF elements: if the memory variables and the associated residual storage are initialised empty then, harmlessly, they remain empty throughout the loop.

Within the RK loop, the other two options both address the problem of the potentially stiff low-frequency (LF) dissipation terms by removing them from the values returned by `biot_rhs_3d.m` and handling them elsewhere.

The first possibility is to use an LSERK scheme, as above, in combination with operator-splitting. Here, for each step, we use the (very simple) analytic solution to the stiff terms, independent of the other terms. This can be used at the beginning of the step (simple splitting, analytical followed by numerical, or AN), at the end of the step (simple splitting, NA) or, using half the step width, at both the beginning and the end (Strang splitting, ANA). The remaining possibility (Strang splitting, NAN) is not implemented because it would involve two expensive numerical steps and would be unlikely to repay those costs. This is all controlled by `param.splitting` (`SPL_NONE`, `SPL_SIMPLE_AN`, `SPL_SIMPLE_NA`, `SPL_STRANG_ANA`)

The other possibility is to replace the explicit (LSERK) method by an IMEX method. Here, the main body of the equations are evolved using the output from `biot_rhs_3d.m` and an explicit method similar to an LSERK method, but the LF dissipation terms are evolved by an implicit method. This turns out to be quite simple because the dissipation terms contain no spatial derivatives; in consequence, the equations that must be solved for the implicit scheme have no coupling between nodes and in fact turn out to have a simple, explicit and computationally very cheap solution.

2.3 The right-hand-side: `biot_rhs_3d.m`

`biot_rhs_3d.m` computes the "right-hand side" of the PDE system: effectively, the nodal values of the spatial derivative terms. Just as all the time discretisation is implemented in `biot_3d.m`, all the spatial discretisation is implemented in `biot_rhs_3d.m` or, more accurately, in the functions that it calls: its main job is to split up the current state data into pieces according to element properties, send these pieces to other functions and reassemble the return values. It can run in two modes, according to the system being simulated: adjoint or non-adjoint. This is specified by a Boolean value passed as an argument by `biot_3d.m` and depending on various configuration options (`param.run_forward`, `param.run_forward_adj`, `param.run_adjoint_method`).

The first part of the process is to implement boundary value rules.

The second part of the process involves fluxes on face nodes. Those not on the boundary are broken up according to the type (elastic or poroelastic) of element on which they belong and the type of element to which they are adjacent. This gives four possibile sets of nodes, each of which is processed by its own function: in the non-adjoint case, these are `get_face_flux_ee.m`, `get_face_flux_ep.m`, `get_face_flux_pe.m`, `get_face_flux_pp.m`. In the adjoint case, they are `get_face_flux_ee_adj.m`, etc. Once all the relevant functions have been called, all their return values are assembled into a single array of flux values.

The third part of the process involves the element nodes. Spatial derivatives of all fields are calculated and, along with the face fluxes already found, split up by element type (elastic or poroelastic) and passed to `biot_rhs_3d_e.m` and `biot_rhs_3d_p.m` to compute the relevant volume integrals. There are no separate adjoint versions of these; instead, the Boolean controlling (non)adjoint calculation is passed on and subsequently used to select the appropriate matrices.

The fourth part of the process involves dissipation. This occurs separately on high-frequency and low-frequency elements; the data associated with these elements is sliced out and passed to `dissipation_hf.m` and `dissipation_lf.m` and the return values combined with the re-

sults already obtained. As described under `biot_3d.m` above, the low-frequency dissipation terms are prone to stiffness and can require special handling by operator-splitting or an IMEX scheme; if either of these is in operation, `dissipation_lf.m` is not called.

The fifth and final part of the process is to add on source terms.

3 Notes on certain problem files

3.1 Directory structure and other housekeeping

To avoid overloading online storage like Dropbox the code has an option to set a path to a local directory for data storage. This is set as default to the current working directory `param.problem_data_dir = pwd` in the driver file `driver_biot_3d`, but is usually overidden in the problem files.

`driver_biot_3d` also contains many default options including default filenames for saving data, switches, initial states and other configuration parameters, which can all be overridden in a problem files. Usually initial states are initialised as being empty and are overwritten as necessary in a problem file.

The subdirectory `meshes` in the examples directory contains some simple regular meshes of a $5\times5\times5$ metre cube. The meshes were constructed by dividing the cube into equal subcubes, with each subcube divided into 6 tetrahedra of equal size. `mesh_0` is the coarsest mesh, with each axis divided into 16, while `mesh_1`, `mesh_2`, `mesh_3` are steadily finer, with respectively 19, 24 and 28 subdivisions. Element and vertex coordinate data is held in separate files, e.g. `mesh_0_element.txt` and `mesh_0_vertex.txt` contains the node coordinates. There are also some slighly modified versions, `mesh_4`...`mesh_7`, which are combinatorially the same but scaled to a $1 \times 1 \times 1$ metre cube.

3.1.1 Magic numbers

The code contains many flags which control the way the system runs, for example, model type (elastic/poroelastic or mixed), source type, integration scheme. These are given as explicity named strings; to avoid string comparisons we have assigned 'magic numbers', so each flag name is assigned to an integer, see `setup_magic`. This isn't necessary in Matlab (except that it gives some clarity), but was done to harmonise the Matlab code with a parallelised C version (not released with the Matlab code). In this case magic numbers are assigned during compilation, so that integers and not strings are compared during execution.

3.2 Assembly of basic structures for DG method

The function `setup_domains` carries out some basic housekeeping once the model has been specified in the problem file (model type by subdomain): each mesh element is associated with a model type (e.g elastic or poroelastic low frequency), which is then inflated to element nodes to identify the interface boundaries (by node) between different model types. Various masks are defined which are used by the `get_face_?` functions discussed in section 2.

In particular, a mask identifying the interfaces between elastic and poroelastic models is used to implement the correct upwind flux at these model interfaces.

The functions `build_param` and `build_param_adj` are then called to assemble elastic and poroelastic mass and stiffness matrices as well as eigenvectors on an element-by-element basis.

3.3 Convergence tests

The following problem files corresponding to elastic and poroelastic (inviscid, Biot low frequency and high frequency regimes, respectively) have been included.

- `problem_ECT`
- `problem_ICT`
- `problem_LFCT`
- `problem_HFCT`

They are the simplest driver files and for a given mesh, physical parameters and and plane wave solution, compute the wavefield (using the plane wave solution as Dirichlet boundary values) at a given final time. Summary data and errors between the numerical and analytical solution are displayed in the Matlab workspace at prescribed time steps as the solution is computed. Although a C version of the code was used to generate the results in Tables 5–7 of our paper, this code will also generate the results.

3.4 Forward and adjoint forward solvers

The problem file `problem_adjoint_forward_solver_test` simply runs a forward and forward adjoint wavefield computation for a 2-layer poroelastic model, and prints to the Matlab workspace summary wavefield data. The frequency with which that information is printed to the screen can be changed by setting `steps_per_report` in the problem file.

3.5 Adjoint method

The problem file `problem_adjoint_method_test2.m` carries out the computation of [3, section 7.3]. In this example, data is generated on a grid of receivers using an (almost) homogeneous model in which the material parameters have been perturbed on a single element (in this case the solid and fluid densities were doubled) using an explosive source. To generate an adjoint wavefield, a completely homogeneous model is specified by setting the perturbed element to have the same parameters as the rest of the domain.

To compute the adjoint method wavefield requires running the homogeneous wavefield forwards in time in order to assemble the adjoint wavefield source, which is the difference between the receiver measurements for the perturbed wavefield and for the homogeneous wavefield.

In the final step the adjoint wavefield is computed together with the sensibility kernels given in equations [3, (205)–(210)].

Since there are three steps in this model we have introduced a flag `run_type` which has the options:

- `MAKE_ADJOINT_DATA`
- `SETUP_ADJOINT_PROBLEM`
- `SOLVE_ADJOINT_METHOD`

To generate the data using the perturbed model, i.e. the first step, requires setting the `run_type =MAKE_ADJOINT_DATA`, &c.

In this driver code some output is plotted using the plotting function `plot_section_3d.m`. In this, a rectangular grid is constructed, parallel to two of the coordinate axes, and the wave field is interpolated (using the DG basis polynomials) onto the grid; the resulting data is then presented as a heat map.

We remark briefly on the computation of the Fréchet kernels. To compute the kernels of parameters arising in the mass matrix requires computing an integral with respect to time of an adjoint wavefield value multiplied by a time derivative of the forward wavefield, see [3, equations (205)–(207)]. As with the computation of the memory variables in the Biot high frequency case, this can be replaced by the right hand side of the variables in question, and thus becomes incorporated in the RK loop in `biot_3d`. The right hand side computation for all the kernels is bundled into `kernel_rhs`.

References

[1] J. Hesthaven and T. Warburton, Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications, Springer 2008.

[2] J. Hesthaven and T. Warburton, Library code from above book, `https://github.com/tcew/nodal-dg`

[3] Nick Dudley Ward, Simon Eveson and Timo Lähivaara, A Discontinuous Galerkin method for three-dimensional elastic and poroelastic wave propagation: forward and adjoint problems. `https://arxiv.org/abs/2001.09478`