Higher Order Spectral Elements in M3D

Bernhard Hientzsch
Courant Institute of Mathematical Sciences
New York University
mailto:BernHien@gmail.com
http://www.math.nyu.edu/~hientzsc

March 20, 2007

Talk at the
Future Directions for M3D Workshop
Princeton Plasma Physics Lab, Princeton, New Jersey

Working with H.R. Strauss at NYU.

Why spectral elements, I

- Exponential convergence for many problems – very efficient discretization.
- p-refinement (increasing the degree) is very easy to implement.
- Works on any quadrilateral mesh, even with curved elements.
- Fast application and assembly of stiffness and mass matrix. Fast solvers for special situations. Direct solvers for symmetric positive (or negative) definite problems. Much work on preconditioning for standard problems such as Poisson equation, Helmholtz equation, and Maxwell equation. (But that work needs to be adapted or applied to the M3D problems.)
- Straightforward discretization and implementation.
- Fast solvers and methods for regular domains/problems can be used for the development and debugging of methods for mapped and/or curved domains.

Why spectral elements, II

- General, variable order discretization that can be implemented using a few different types of matrices and modules (don't need to implement different elements and modules for different orders and problems)

- Easy derivation and computation of discretizations and matrices (PDE $\mapsto$ weak $\mapsto$ discretization using standard matrices $\mapsto$ optimized formulation).

- Lends itself to a modular implementation.

- A nodal representation is especially adapted to operations in the physical domain (Poisson bracket as a pointwise operation can be discretized easily). Can be more easily debugged and understood. Can be transformed easily into modal representation. Modal representation might lend itself to easier frequency-space filtering, resolution or truncation error analysis.

Spectral elements in words

- Discrete problem is a subassembled version of element-wise discretizations, consisting of combinations of derivatives (differentation matrices), interpolation (interpolation matrices), and integrations (mass matrices).

- The fundamental matrices (derivative, interpolation, mass, even mapping to C^1 or modal representation) on the reference element are tensor product matrices or sums thereof – easy to implement, fast to apply or assemble. Application of tensor product matrices turns into dense matrix-matrix (respective tensor-matrix) multiplications which run at close to peak.

- Mappings and curved elements bring in geometry factors inside of diagonal matrices (or, in the application, pointwise multiplications). The geometry factors themselves can be computed using the standard matrices.

Mapped spectral elements in symbols,I

- *Integration/mass matrices:* $(u, v) \approx \sum_E v^T R_E^T \tilde{M}_E R_E u$

  For diagonal mass matrix: (J-Jacobian, $\Lambda(A)$ diagonal matrix with diagonal A):

$$\tilde{M}_E = \Lambda(\tilde{m}_E) = \Lambda(J)\Lambda(m_{2D}) = \Lambda(J)(\Lambda(m_{1D}^x) \otimes (\Lambda(m_{1D}^y)$$

  For general mass matrix: $\tilde{M}_E = \Lambda(\sqrt{J})(M^x \otimes M^y)\Lambda(\sqrt{J})$.

- *Differentiation matrices:*

$$\begin{aligned} D_x &= \Lambda\left(\frac{\partial s}{\partial x}\right)(D_s \otimes I) + \Lambda\left(\frac{\partial t}{\partial x}\right)(I \otimes D_t) \\ D_y &= \Lambda\left(\frac{\partial s}{\partial y}\right)(D_s \otimes I) + \Lambda\left(\frac{\partial t}{\partial y}\right)(I \otimes D_t) \end{aligned}$$

Mapped spectral elements in symbols,II

- *Geometry factors:* If X and Y are the matrices containing the images of the GLL points from the reference element,

$$J = ((D_s \otimes I)X) \circledast ((I \otimes D_t)Y) - ((D_s \otimes I)Y) \circledast ((I \otimes D_t)X) = [X, Y]_{s,t}$$

$$\frac{\partial s}{\partial x} = ((I \otimes D_t)Y)./J \qquad \frac{\partial s}{\partial y} = -((I \otimes D_t)X)./J$$

$$\frac{\partial t}{\partial x} = -((D_s \otimes I)Y)./J \qquad \frac{\partial t}{\partial y} = ((D_s \otimes I)X)./J$$

- *Laplacian:*

$$\begin{aligned}\tilde{K}_E &= (D_s^T \otimes I)\Lambda(w_{11})(D_s \otimes I) + (D_s^T \otimes I)\Lambda(w_{12})(I \otimes D_t) + \\ &\quad (I \otimes D_t^T)\Lambda(w_{12})(D_s \otimes I) + (I \otimes D_t^T)\Lambda(w_{22})(I \otimes D_t)\end{aligned}$$

Mapped spectral elements in C (Poisson bracket)

```
// derivatives on reference element
matmul(GlobCtx->dm,thisa,ax,k+1,k+1,k+1);matmul_nt(thisa,GlobCtx->dm,ay,k+1,k+1,k+1);
matmul(GlobCtx->dm,thisb,bx,k+1,k+1,k+1);matmul_nt(thisb,GlobCtx->dm,by,k+1,k+1,k+1);
for (j=0;j<(k+1)*(k+1);j++) { // geometry factors
  axm[j]=thisctx->d1dx[j]*ax[j]+thisctx->d2dx[j]*ay[j];
  bxm[j]=thisctx->d1dx[j]*bx[j]+thisctx->d2dx[j]*by[j];
  aym[j]=thisctx->d1dy[j]*ax[j]+thisctx->d2dy[j]*ay[j];
  bym[j]=thisctx->d1dy[j]*bx[j]+thisctx->d2dy[j]*by[j];}
// interpolate to GLL grid
matmul(GlobCtx->ip,axm,temp,kg+1,k+1,k+1);matmul_nt(temp,GlobCtx->ip,axi,kg+1,k+1,kg+1);
matmul(GlobCtx->ip,aym,temp,kg+1,k+1,k+1);matmul_nt(temp,GlobCtx->ip,ayi,kg+1,k+1,kg+1);
matmul(GlobCtx->ip,bxm,temp,kg+1,k+1,k+1);matmul_nt(temp,GlobCtx->ip,bxi,kg+1,k+1,kg+1);
matmul(GlobCtx->ip,bym,temp,kg+1,k+1,k+1);matmul_nt(temp,GlobCtx->ip,byi,kg+1,k+1,kg+1);
// compute point-wise poisson bracket
for (j=0;j<(kg+1)*(kg+1);j++) pbgll[j]=thisctx->jacg[j]*(axi[j]*byi[j]-ayi[j]*bxi[j]);
// mass and transpose of interpolation
left_scale_mat(pbgll,pbgll,GlobCtx->mg,kg+1,kg+1);
right_scale_mat(pbgll,pbgll,GlobCtx->mg,kg+1,kg+1);
matmul(pbgll,GlobCtx->ip,temp,kg+1,kg+1,k+1);
matmul_tn(GlobCtx->ip,temp,thisp,k+1,kg+1,k+1);
```

Available framework/implemented modules

- Subassemble the global matrix
- Apply the global matrix in a matrix-vector product
- Static condensation (i.e., subassemble the global Schur complement system)
- Global sparse solve [block-sparse methods could be useful] for the Schur complement system and the needed local solves for the static condensation.
- Computing element matrices for Laplacian, Helmholtz, including weighting etc.
- Computing geometry factors for straightline/curved elements.
- Standard spectral element matrices and algorithms.

Integration of a spectral element module into M3D

- Proof of concept, algorithmic development in MATLAB. First serial production version in C.
- Refactoring modules and routines and providing two interfaces: opaque pointers for original data structures (SWIG and C) and flat vectors (for FORTRAN interface).
- Reimplement C version, and implement PYTHON and FORTRAN versions using these modules and interfaces (easy scripting).
- Define interface to M3D for discretization modules.
- Implement this interface for spectral element discretizations based on the previous C version.
- Integrate, test, and use this module with serial, OpenMP, PETSc and/or other distributed memory version of M3D.

Plans

- Completely implement curved elements.
- Implement iterative methods and preconditioners (iterative substructuring methods and overlapping methods).
- Parallelization for distributed memory machines.

M3D interface

- M3D implements the assembly of the right hand side and the elliptic solves in terms of a number of functions/routines defined originally in `mpp3.F`.
- The M3DSEL module provides an implementation of that interface in terms of solvers for specific symmetric statically condensed systems (in a new file `mpp4bh.F`) and spectral element operators.
- The functions in `mpp4bh.F` are implemented in terms of the functions of the original C version (plus some necessary extensions) in `mpp4interface.c`.
- To initialize the discretization and mesh structures and some structures that will be needed in the solvers, we implement a call back function `getselmesh` in getmesh.c which initializes the needed structures, computes the GLL mesh, and sets up the environment for the C module.

The mpp3/mpp4bh interface

`poiss`	Solves $\Delta u = f$
`delsq`	Computes Δu
`gcro`	Computes $[u, v]$
`gradsq`	Computes $\nabla u \cdot \nabla v$
`agrad`	Computes $\nabla u \cdot \vec{v}$
`dxdr` , `dxdz`, `wgrad`	Compute derivatives
`lowpois`	Solves $\frac{1}{R}(\nabla \cdot (R\nabla))u = f$
`lowpoisa`	Solves $R(\nabla \cdot (\frac{1}{R}\nabla))u = f$
`poisvmu`	Solves $(\Delta - 1/(dt * ss(.)))u = f$
`poisdmd`	Solves $(\nabla \cdot (ss(.)\nabla) - 1/(dt * hmt))u = f$
`poisvmu3`	Solves $(\Delta - 1/(dt * ss(.))u = f$
`lopoismu`	Solves $(\nabla \cdot (\frac{1}{R}\nabla) - 1/(dtt * R * ss(.)))u = f$
`load3`	Computes load vector
`pvol`	Computes volume integral
`intgrsq`	Computes square integral
`*fft*`	Some FFT functions

The mpp4interface.c implementation

- Differential and integral operators are straightforward to implement once some basic structures have been initialized and some fields have been computed.
- The elliptic solvers are all symmetric and positive definite (or negative definite).
- Assembling local matrices straightforward. Schur complement matrix can be assembled in general function with a function argument (that function computes parametrized local matrix).
- Solvers can be initialized and solve can be implemented as general Schur complement solvers for symmetric systems, using sparse direct solvers for global block-sparse system, dense direct solvers for local systems, and fast BLAS. Use abstract interface for these linear algebra algorithms. Has been implemented with several different packages.

Status of the M3D integration: serial, OpenMP

- Serial M3D runs for several test problems. Could be more stable and cleaner, but it works with M3D if done/used right.
- OpenMP version runs for the same test problems. No thread issues anymore. In the moment, we are trying to apply it to more and more problems to find hidden bugs/problems/issues. The interface is completely implemented, but not completely tested.
- The module has reached some stable state, except necessary fixes every now and then. Started to work on documentation and examples to verify the implementation.

... onto distributed memory, massively parallel

- In the moment, I am considering several possible distributed memory parallelizations of the module and M3D.
- The approach with the least amount of rewriting necessary for the module would be to implement a parallelization of that Schur complement approach, maybe two-level or multi-level, and see if this allows massively parallel implementations. In this case, one only needs to communicate the data on the interfaces, which can be implemented in a straightforward way.
- The approach more similar to PETSc-M3D and possibly optimal with respect to complexity would be to use iterative solvers and optimal domain decomposition/multigrid preconditioners. Alas, Schur complement methods for spectral elements require good coarse spaces but (relatively little) communication since only on the interface; overlapping methods are somewhat easier but require more communication.

More details: description of the skeleton mesh

- Elements are described as 4 x nt array of indices of their vertices. [`elempt` in C code, `itnode` in FORTRAN]
- The coordinates of the points are given in arrays indexed by the point index. [`pos_x`,`pos_y` in C]
- My code: boundary edges are described as 2 x nb array of the indices of their vertices [`bdedge` in C]. M3D: points are considered boundary points if their index is higher than a threshold, `ldb`. Any edge between two boundary points is a boundary edge. (Does not work for thin domains.) My code transforms one into the other.

Generating the GLL mesh points and meshes [setupelem.c]

- Starting from the indices and the coordinates of the corners, my code computes the coordinates of all GLL points.
- It also determines the correct indices for points on the element interfaces, on the boundary, and in the interior of elements.
- This is needed to be able to store the data in a flat vector, to work on distributed datastructures, and also to implement static condensation.
- Might also be needed for other geometric operations and implementations of certain other algorithms.

Representations of the variables

- Flat FORTRAN vectors as used in M3D.
- *Distributed variables* [`distvar.c`]: variable is described by the collection of its values on all elements. This allows the fast implementation of many spectral element operations (see for instance earlier slide for Poisson bracket implementation). (Also, later on, for the implementation of iterative methods with overlapping preconditioners.)
- Vectors of interface values or interface and boundary values as needed in the solution of the Schur complement system for different kinds of boundary conditions. (Also, later on, for implementation of iterative substructuring methods.)
- My code provides mappings between these [`flatvar.c` and elsewhere]

Global operations based on element-wise operations [assemble.c,mapdifop.c,...]

- `assemble.c` implements different kinds of subassembly for elementwise results to give global results, corresponding to different treatments on the boundary (this is essentially just summing up the correct values and treating the boundary correctly).

- This is used to implement differential operators in `mapdifop.c` (Laplace+0 B.C.: `ApplyLapSub`; Strong Laplace averaged: `ApplyLapSubStrongBdry`; `mappoisbrgll`: ([a,b],v), also for x- and y-derivatives; `mapgradsq`: $(\nabla a \cdot \nabla b, v)$, `mapagrad`: $(\nabla a \cdot \vec{b}, v)$; `mapdxdr`: (a_x, v); `mapdxdz`: (a_y, v)) *and*

- also to implement integral operators in `mapdifop.c` (`maploadvec`: $< bc, v >_{\partial\Omega}$; `mapintgrsq`: $(\nabla a, \nabla b)$; and `mappvol`: $\int a \quad d\Omega$.

Direct solvers for symmetric positive definite problems [symschursolve.c]

- The function `prepsymschur` takes a pointer to a function and an opaque parameter vector (used to generate element-wise matrix for those parameter values), and subassembles the global Schur complement matrix for essential (Dirichlet) and natural (Neumann) boundary conditions, possibly non-singularized, and also the matrix for the right hand side for nonhomogeneous Dirichlet boundary conditions.
- To do that, it needs implementations of dense symmetric factorizations and solves (for the interior of the elements) and sparsification and factoring for sparse symmetric solve for the global Schur complement.
- Using these pre-computed factorizations and implemented solves, the different stages of the Schur complement solver are implemented for symmetric positive and negative definite problems with different boundary conditions, which are: computing the right hand side of the Schur complement system, solve the Schur complement system, and solving the local systems in the interior of the elements.

The abstract interface for the element-wise and global solvers

- For the global sparse problem, `spsymsolve.h` defines an abstract interface consisting of `init_sparse`, `exit_sparse`, `sparsify`, `sparse_sym_factor`, `sparse_sym_solve`, `freeSpMatrix`, and `freeSpMatSlv`.

- For the local dense problems, `densymsolve.h` defines an abstract interface consisting of `dense_sym_factor`, `dense_sym_solve`, and `dense_sym_solve_mult`.

- There are a few implementations, mainly trying to use different program packages. In the moment, I am using an LAPACK implementation for the dense symmetric solves (see `densymslv_lapack1.c`) and an LDL implementation for the global sparse symmetric solves (see `spsymslv_ldl.c`).

The actual problems assembled and implemented [hhschurprep.c]

`prepsymschur` is called from `hhschurprep.c` to assemble the correct matrices and to set up the solvers for the following problems:

- $\alpha(\nabla u, \nabla v) + \beta(u, v) = (f, v)$
- $\alpha(\nabla u, \nabla v) + (\beta(\cdot)u, v) = (f, v)$
- $\alpha(\gamma(\cdot)\nabla u, \nabla v) + \beta(u, v) = (f, v)$
- $\alpha(\gamma(\cdot)\nabla u, \nabla v) + \beta(\delta(\cdot)u, v) = (f, v)$

It is easy to extend this list to others, the listed ones can all be computed easily from the fundamental matrices already computed.

Supporting cast, I

- Interfaces to libraries: `cblasdegemm.c` and `cblasdgemv.c`: very rudimentary implementation of CBLAS in terms of BLAS. `blasmacros.h`: useful macros to express matrix-matrix multiplication shorter, which then expands to the correct BLAS calls.
- Interface to my tools: `dvarbinary.c` and `readelem.c`: write and read descriptions of the triangulation and also variables in a binary format (for debugging and my visualization).
- Standard matrices: `discelem.c`: compute the discretization matrices (mass, differentiation, interpolation, Jacobian, geometry factors, Laplacian, grid points) and the lengths of the boundary edges. `semmat.c`: compute standard spectral element matrices for the reference interval or reference element. `gllgrid.c`: compute the one-dimensional GLL grid. `legpoly.c`: values of Legendre polynomials and their derivatives.

Supporting cast, II

- Fast operations: `scalemat.c`: utility functions to scale matrices (faster implementations in newer BLAS??). `tensormult.c`: tensor-product-matrix operations.
- Others: `chkptr.c`: tracking down zero pointers. `prepimass.c`: compute inverse diagonal mass matrix. `rdistvar.c`: various distributed variables and operations involving expression in R.

General control flow of the module

- `getselmesh_` is called during initialization. In it, the GLL mesh points, their indices, and whatever data structures and global variables are computed respective allocated.

- M3D is implemented in terms of calls to functions formerly defined in `mpp3.h`, but now implemented in `mpp4bh.F`.

- `mpp4bh.F` implements this FORTRAN interface, mostly by calling the appropriate C functions implemented in `mpp4interface.c`.

- The functions in `mpp4interface.c` first map the flat vector to distributed variables, and then call functions implemented in various files in my code, map the results back to flat vectors, and multiply by the inverse of the mass matrix, if appropriate.

Control flow example

- poiss in mpp4bh.F calls slvpoiss0bh_ in each poloidal plane.
- slvpoiss0bh_ is implemented in mpp4interface.c. In its first call, it assembles the appropriate matrices and factors them with prepMapHHSchur from hhschurprep.c, which in turn call prepSymSchur from symschursolve.c, which in the current version calls LAPACK and LDL to do the actual factorizations and solves.
- In all calls, these factorizations are used to solve the Poisson problem with different types of boundary conditions (solveSymSchurNeg, solveSymSchurNHNeg, solveSymSchurNeuNeg) in symschursolve.c, which again uses the abstract solver interface to finally have LAPACK, LDL, and BLAS do the main computational work.

The end

- This was a short overview of the implementation. More on request. Also, I would be happy to explain more about the algebraic or mathematical details.
- For examples of results of my modules, see previous talks at the CEMM meetings.
- For first examples of M3D running with my module, see my talk at Philadelphia CEMM meeting.
- For more examples of M3D running with my module, see Hank Strauss's presentation.
- Thanks for your attention.