Spectral Elements for Two-Dimensional Resistive MHD

Bernhard Hientzsch
Courant Institute of Mathematical Sciences
New York University
mailto:Bernhard.Hientzsch@na-net.ornl.gov
http://www.math.nyu.edu/~hientzsc

April 10, 2005

CEMM Meeting April 2005
2005 International Sherwwod Fusion Theory Conference
April 11-13, 2005 in Stateline, Nevada

Postdoc under H.R. Strauss at NYU.

Overview

- Why spectral elements: the approach
- Incompressible MHD in 2D: in primitive variables and potentials
- Time discretization (FD), space discretization (SEM), complete algorithm
- Tilting mode examples. Setup (IV/BV). Variables, energies, peak currents, growth rates
- C1 continuous elements and mapped elements
- Implementation in C/LAPACK/ATLAS resp. SUNPERFLIB
- Numerical observations and future work

Why spectral elements: the approach

- Exponential convergence for (standard) problems with (piecewise) smooth solutions
- Even if the solution is only piecewise smooth, alignment of elements, postprocessing, or filtering can possibly restore higher order convergence
- Faster solvers/application of element matrices and of subassembled system for rectangular array of elements (Helmholtz: generalized Sylvester equation)
- Implementation is relatively straightforward, can be expressed as LAPACK calls and some self-written modules (or as script in MATLAB)
- Runs at (relatively) high percentage of peak at modern computer architectures. (Sparse block matrix with dense blocks. Usually degree 15-20 resolves space enough)

General philosophy

- Try out spectral element type discretizations on more complex problems
- Rapid prototype the discretizations and methods. Choose simple and straightforward approaches first to get to into the problem as soon as possible
- Rapid prototype the programming (First use MATLAB, then modular C/PETSc code using LAPACK/ATLAS or SUNPERFLIB or similar and possibly inserting into M3D or other packages)
- Try out methods first on structured grids where one has fast solvers etc. so that quick turnaround, extensive testing possible, and a slightly higher chance to find bugs and understand what is going on

Incompressible MHD: primitive variables

∇u is the standard gradient $\nabla u := \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\right)$

$\nabla^2 u$ is the standard Laplacian $\nabla^2 u := \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$

$\mathbf{B}$: magnetic field.$\mathbf{v}$: velocity. ρ: density, assumed constant. μ: viscosity

$$\frac{\partial \mathbf{B}}{\partial t} = \mathbf{curl}(\mathbf{v} \times \mathbf{B}) + \eta \nabla^2 \mathbf{B} \quad (1)$$

$$\rho \frac{\partial \mathbf{v}}{\partial t} = -\rho \mathbf{v} \cdot \nabla \mathbf{v} + \mathbf{curl}\, \mathbf{B} \times \mathbf{B} + \rho \mu \nabla^2 \mathbf{v} \quad (2)$$

$$\nabla \cdot \mathbf{v} = 0 \quad (3)$$

$$\nabla \cdot \mathbf{B} = 0 \quad (4)$$

Incompressible MHD: potential form in 2D (vorticity-flux)

$[a,b] := \frac{\partial a}{\partial x}\frac{\partial b}{\partial y} - \frac{\partial b}{\partial x}\frac{\partial a}{\partial y}$. $\mathbf{v} = \mathbf{curl}\,\phi$ with ϕ (velocity flux) and $\mathbf{B} = \mathbf{curl}\,\psi$ with ψ (magnetic flux). Ω: vorticity. C: current density. η: resistivity (new here)

$$\frac{\partial \Omega}{\partial t} = [C,\psi] - [\Omega,\phi] + \mu\nabla^2\Omega \quad (5)$$

$$\frac{\partial \psi}{\partial t} = -[\psi,\phi] + \eta\nabla^2\psi \quad (6)$$

$$\nabla^2\phi = \Omega \quad (7)$$

$$C = \nabla^2\psi \quad (8)$$

Incompressible MHD: potential form in 2D (vorticity-current)

$$\frac{\partial \Omega}{\partial t} = [C, \psi] - [\Omega, \phi] + \mu \nabla^2 \Omega \tag{9}$$

$$\frac{\partial C}{\partial t} = [\phi, C] + 2\left[\frac{\partial \phi}{\partial x}, \frac{\partial \psi}{\partial x}\right] + 2\left[\frac{\partial \phi}{\partial y}, \frac{\partial \psi}{\partial y}\right] + [\Omega, \psi] + \eta \nabla^2 C \tag{10}$$

$$\nabla^2 \phi = \Omega \tag{11}$$

$$\nabla^2 \psi = C \tag{12}$$

Time discretization (vorticity-flux, semi-implicit)

$$\frac{\Omega^{n+1}-\Omega^n}{\Delta t} = [C^n,\psi^n]-[\Omega^n,\phi^n]+\mu\nabla^2\Omega^{n+1} \quad (13)$$

$$\nabla^2\phi^{n+1} = \Omega^{n+1} \quad (14)$$

$$\frac{\psi^{n+1}-\psi^n}{\Delta t} = -[\psi^n,\phi^{n+1}]+\eta\nabla^2\psi^{n+1} \quad (15)$$

$$C^{n+1} = \nabla^2\psi^{n+1} \quad (16)$$

PDEs to be solved in each time step (vorticity-flux)

$$\Omega^{n+1} - \mu\Delta t\nabla^2\Omega^{n+1} = \Omega^n + \Delta t\left\{[C^n,\psi^n] - [\Omega^n,\phi^n]\right\} \quad (17)$$

$$\nabla^2\phi^{n+1} = \Omega^{n+1} \quad (18)$$

$$\psi^{n+1} - \eta\Delta t\nabla^2\psi^{n+1} = \psi^n - \Delta t[\psi^n,\phi^{n+1}] \quad (19)$$

$$C^{n+1} = \nabla^2\psi^{n+1} \quad (20)$$

(17) and (19) are Helmholtz solves, for the operator $(I + \alpha\nabla^2)$ with $\alpha = -\mu\Delta t$ and $\alpha = -\eta\Delta t$, respectively. For zero viscosity and zero resistivity, respectively, the Helmholtz solves simplify to direct formulae for the new values taking into account possible boundary conditions. (18) and (20) are standard Laplace solves resp. applications of Laplace operator

Spectral elements

- Approximate u by nodal values on GLL grid $\underline{u}.\underline{u}$ also expansion in interpolatory basis. Interpolation between GLL grids of different degrees, differentiation, and integration of SEM function on the grid can be written as matrices I_n^k, D_x, D_y, M

- $(u,v) = \int uv$ can be approximated by a mass matrix: $(u,v) \approx \underline{v}^T M \underline{u}$. For one dimension, and integration on same grid, $(u,v) = \int uv \approx \sum_i u_i v_i \rho_i = \underline{v}^T M \underline{u}$ with diagonal M.

- $(\nabla u, \nabla v)$: this is a sum of inner products (component by component) $(\nabla u, \nabla v) = \left(\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}\right) + \left(\frac{\partial u}{\partial y}, \frac{\partial v}{\partial y}\right)$. Approximate $(\cdot,\cdot)$ as before: $(\nabla u, \nabla v) \approx (D_x \underline{v})^T M (D_x \underline{u}) + (D_y \underline{v})^T M (D_y \underline{u}) = \underline{v}^T D_x^T M D_x \underline{u} + \underline{v}^T D_y^T M D_y \underline{u} =: \underline{v}^T K \underline{u}$ with $K = D_x^T M D_x + D_y^T M D_y$

Different approximations for Poisson Brackets

$$([a,b],v) = \left(\frac{\partial a}{\partial x}\frac{\partial b}{\partial y} - \frac{\partial a}{\partial y}\frac{\partial b}{\partial x}, v\right)$$

Approximate $[a,b]$ pointwise by pointwise multiplication $\circledast$: $([a,b],v) \approx \underline{v}^T M\left((D_x\underline{a}) \circledast (D_y\underline{b}) - (D_y\underline{a}) \circledast (D_x\underline{b})\right) =: \underline{v}^T M P(\underline{a},\underline{b})$ with $P(\underline{a},\underline{b}) = (D_x\underline{a}) \circledast (D_y\underline{b}) - (D_y\underline{a}) \circledast (D_x\underline{b})$

Now there are different choices to approximate this: either compute the derivative as an average or projected continuous function on the entire domain or as a piecewise continuous function. The inner products can be approximated on the same grid, resulting in underintegration, or the derivatives can be interpolated to a finer grid and then the inner product can be computed exactly.

Time stepping algorithm

A time-stepping algorithm can be implemented like

$$
\begin{aligned}
\underline{\Omega}^{n+1} &= \text{HHSolve}(-\mu\Delta t, \underline{\Omega}^n + \Delta t P(\underline{C}^n, \underline{\psi}^n) - \Delta t P(\underline{\Omega}^n, \underline{\phi}^n)) \\
\underline{\phi}^{n+1} &= \text{LapSolve}(\underline{\Omega}^{n+1}) \\
\underline{\psi}^{n+1} &= \text{HHSolve}(-\eta\Delta t, \underline{\psi}^n - \Delta t P(\underline{\psi}^n, \underline{\phi}^{n+1})) \\
\underline{C}^{n+1} &= \text{ApplyLap}(\underline{\psi}^{n+1})
\end{aligned}
$$

Some optimization is possible in the computation of the right hand side by saving terms occuring at several places. Right hand side assembly needs only modules for the two types of terms if modularity is more important. Different BC/DOF/continuity only influence HHSolve, LapSolve, ApplyLap, and P, can reuse the same frame.

Tilting mode problem - Setup

Introduce polar coordinates $x = r\cos\theta$, $y = r\sin\theta$ and use seperable form in polar coordinates,

$$\psi(t=0) = \psi_{0,rad}(r)\cos\theta \qquad \Omega(t=0) = \epsilon\Omega_{0,pert}(r)$$

with the radial functions (k being the first positive zero of J_1, $k \approx 3.8317$)

$$\psi_{0,rad}(r) = \begin{cases} \frac{2J_1(kr)}{kJ_0(k)} & \text{for } r \leq 1 \\ \frac{r^2-1}{r} & \text{for } r \geq 1 \end{cases} \qquad \Omega_{0,pert}(r) = 4(r^2-1)\exp(-r^2)$$

The following boundary conditions were used in this problem:

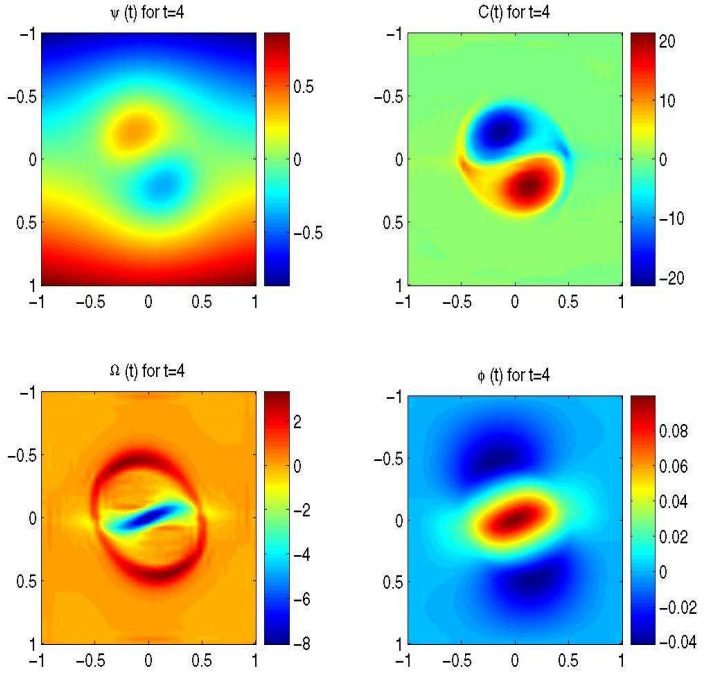
$$\phi = 0 \qquad C = 0 \qquad \frac{\partial\psi}{\partial t} = 0 \qquad \Omega = 0$$

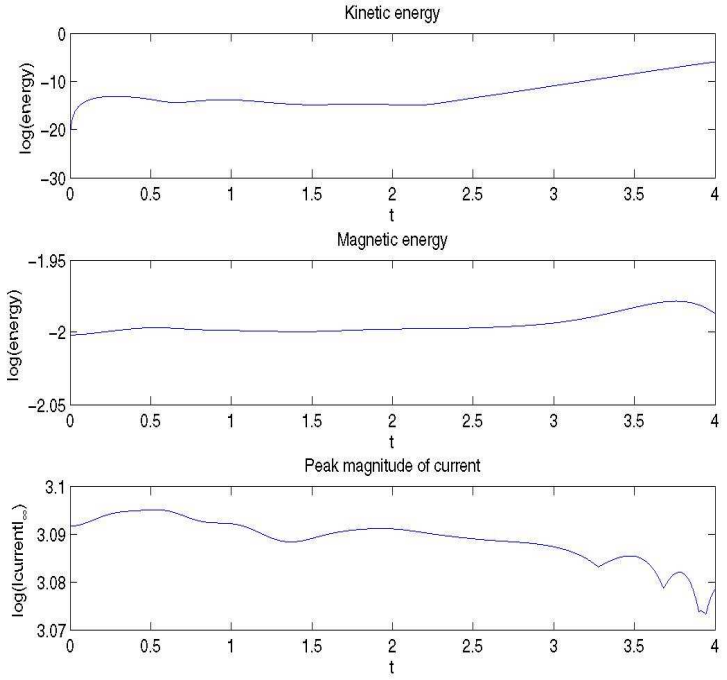
General setting, algorithmic choices

- MATLAB: time-integrating on one element (spectral method) or on a rectangular array: explicit, *semi-implicit*, ODE suite
- Sometime scaled version of problem to compute only on $[-1,1]^2$
- Split rectangle into $M \times M$ elements of degree $N \times N$
- Use the tensor product structure of Helmholtz and Laplace equations for fast solvers (Leftovers: Fast diagonalization methods/block diagonalization methods/Hessenberg-Schur methods for Generalized Sylvester equations)
- Typical: $\mu = 0.005,\ \eta = 0$ or $\eta = 0.005,\ \epsilon = 0.0001$ or $\epsilon = 0.001$, integration up to time $t = 2, 4, 10$

50×50 **elements of degree** 2×2

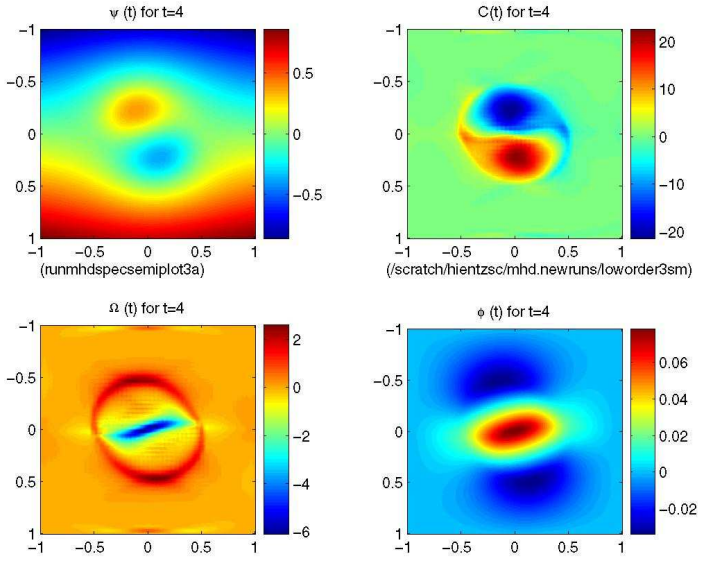
$$PPE = 10,\ \Delta = 0.001,\ t_{final} = 4.0.$$

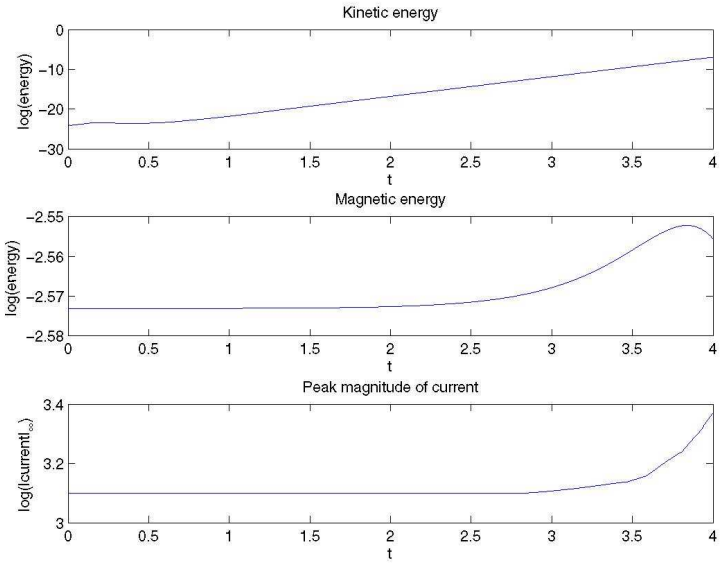




50×50 **elements of degree** 3×3

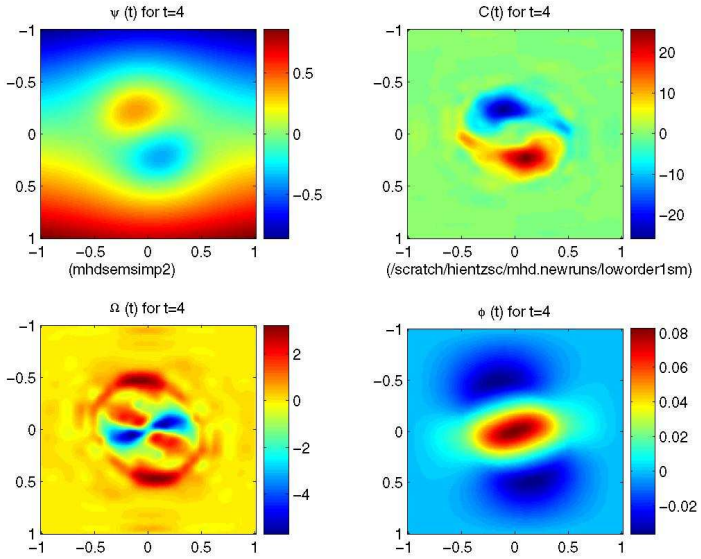
$$PPE = 10,\ \Delta = 0.001,\ t_{final} = 4.0.$$

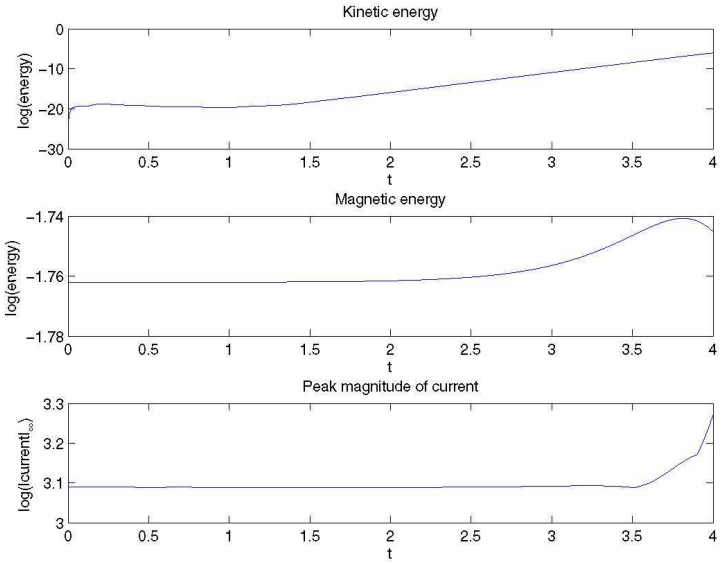




15×15 **elements of degree** 4×4

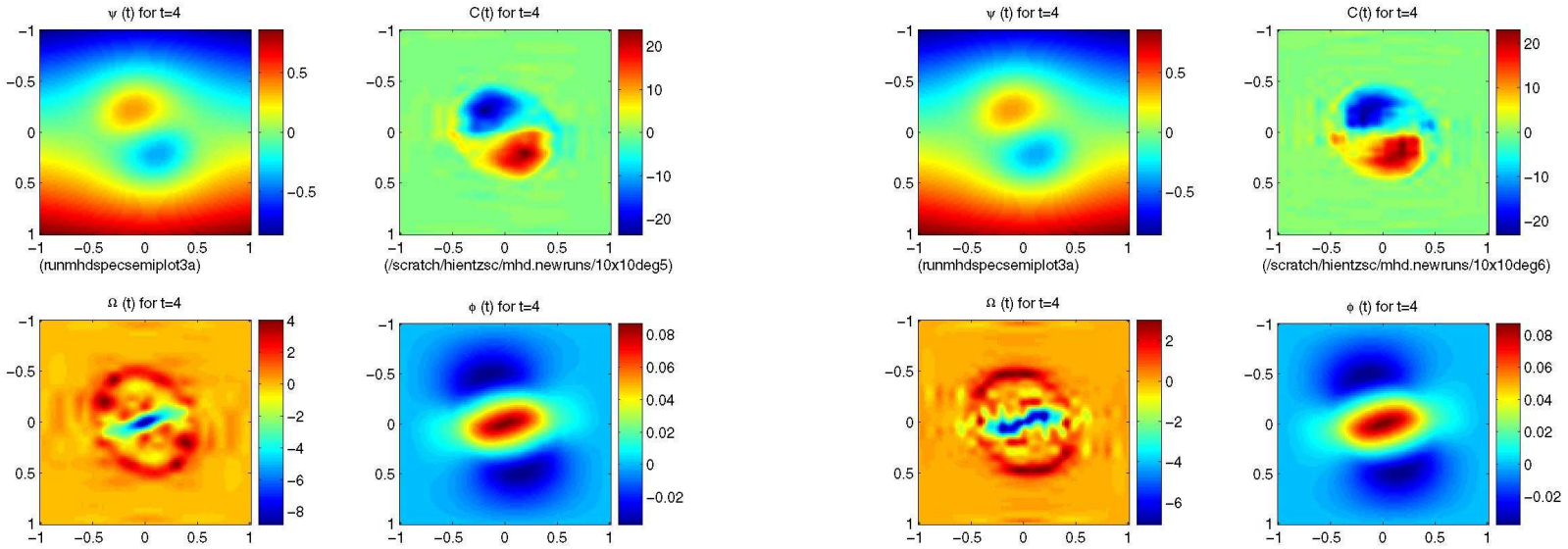
$$PPE = 10,\ \Delta = 0.001,\ t_{final} = 4.0.$$



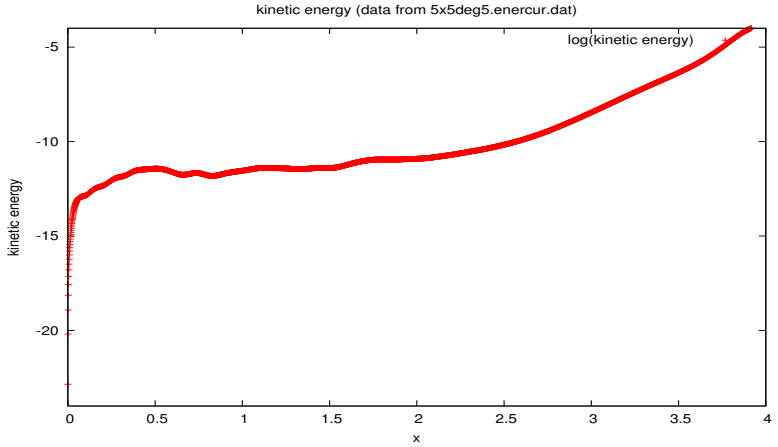


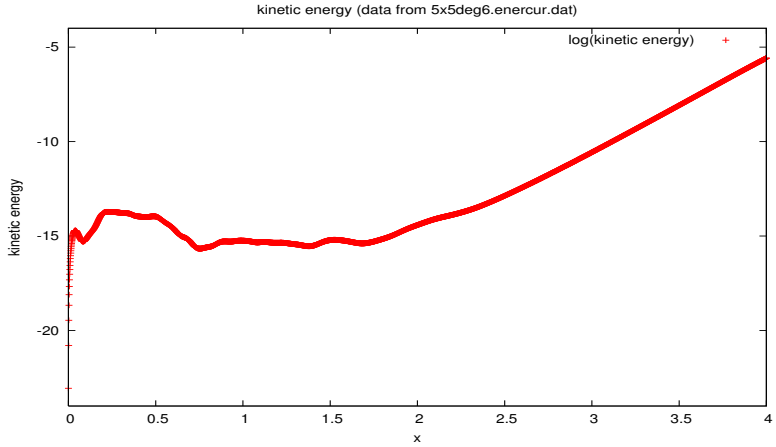
10×10 **elements: degrees** 5×5**,** 6×6

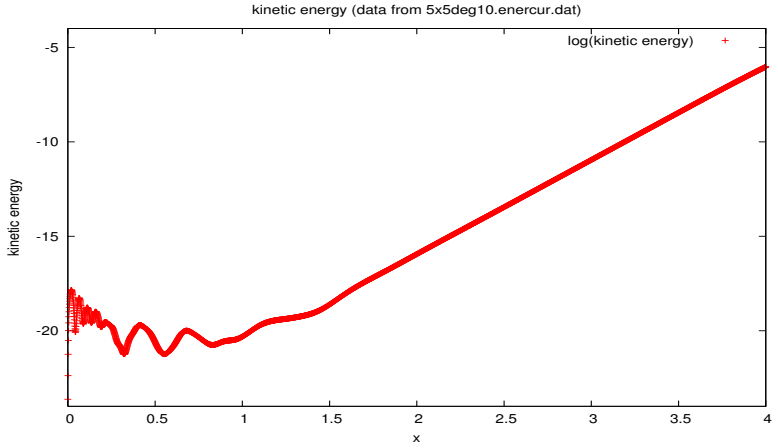
$$PPE = 10,\ \Delta = 0.001,\ t_{final} = 4.0.$$

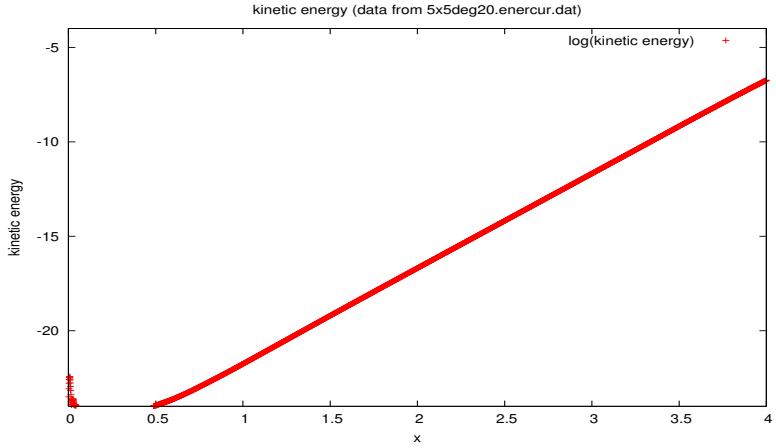


5×5 elements: Kinetic energies









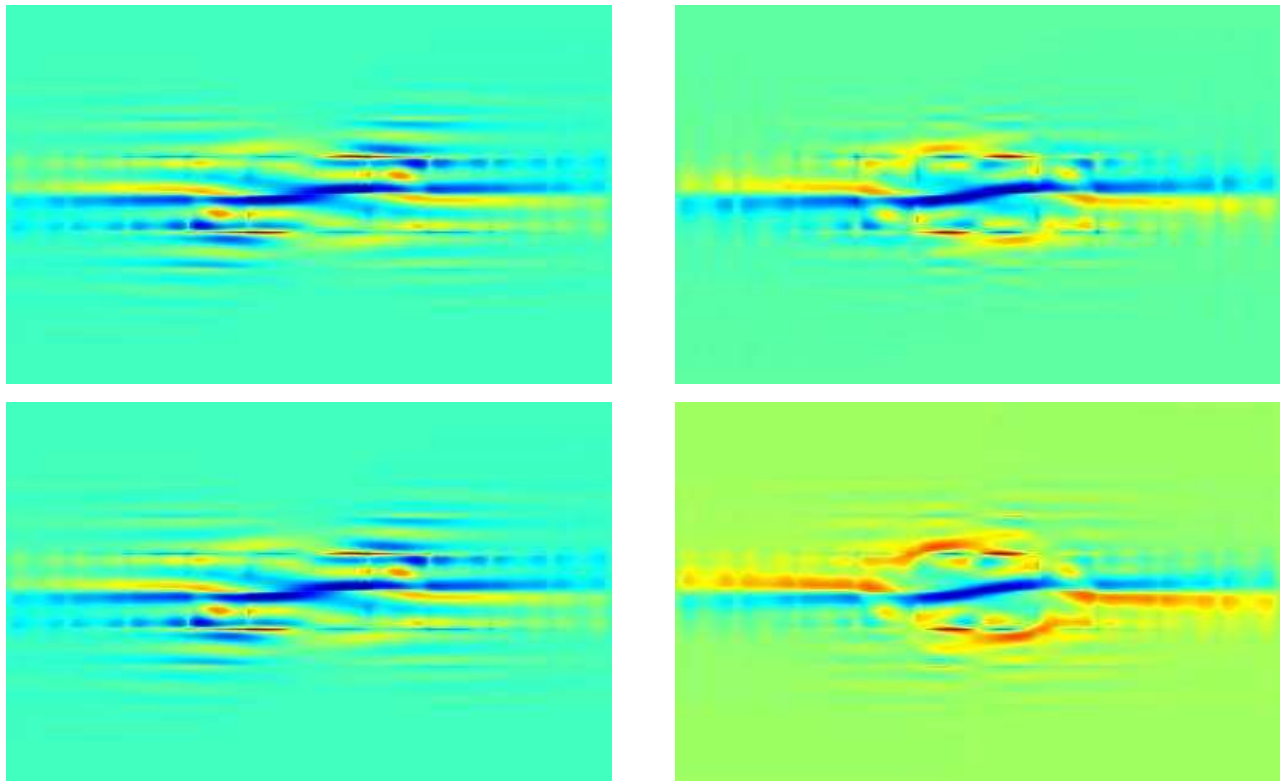
5×5 elements: Growth rates (+elapsed time)

4000 time steps. MATLAB on Sun Blade workstation.

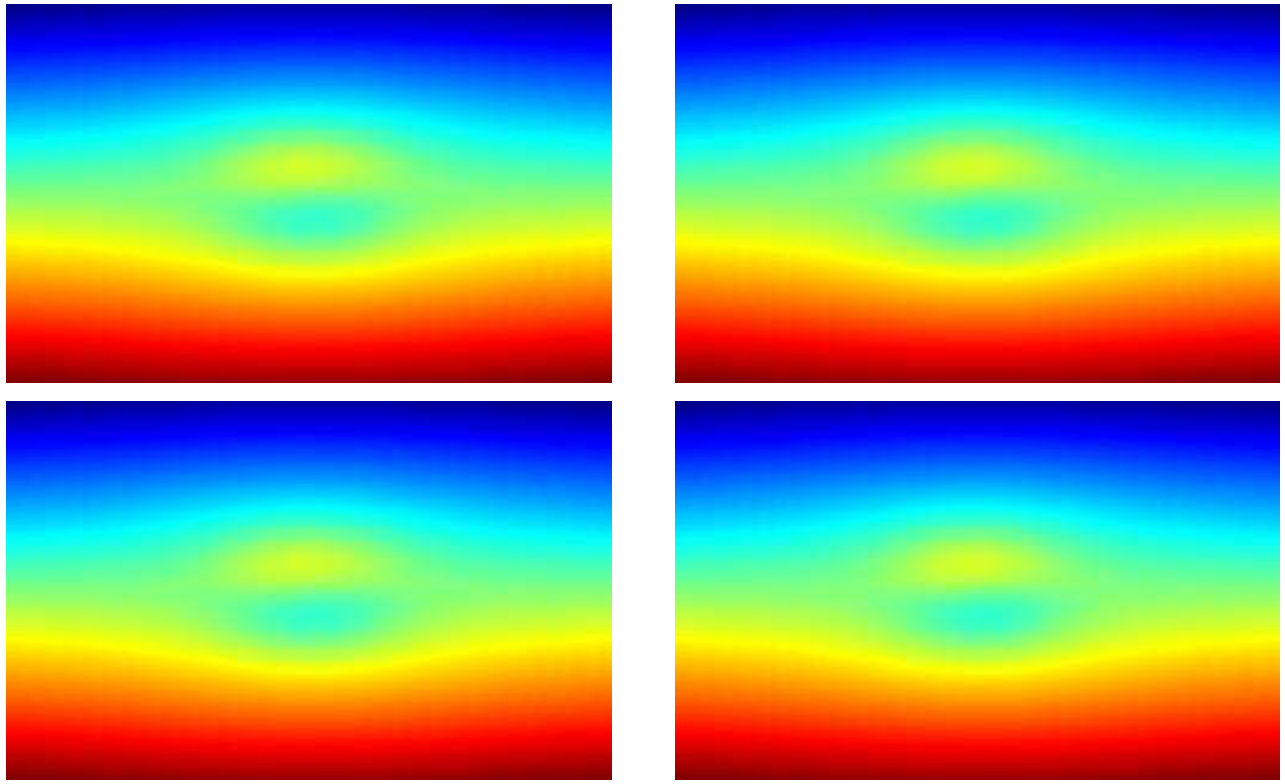
degree	estimated growth rate	elapsed time
5	1.2065	243.3s
6	1.2543	265.4s
10	1.2398	494.4s
20	1.2417	8843s

10×10 elements of degree 5, Poisson brackets, Ω

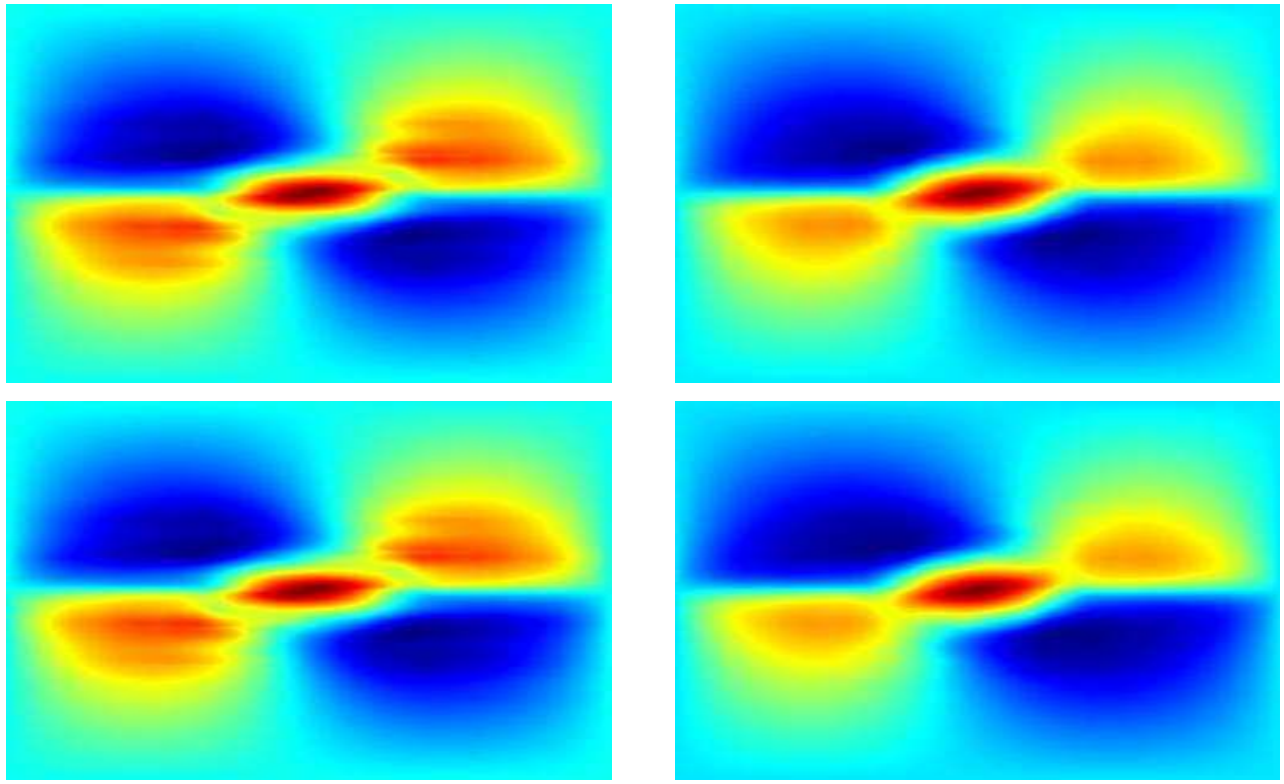
At time $t = 6.0$. (Projected derivative underintegrated, exactly integrated; discontinuous derivative underintegrated, exactly integrated.)



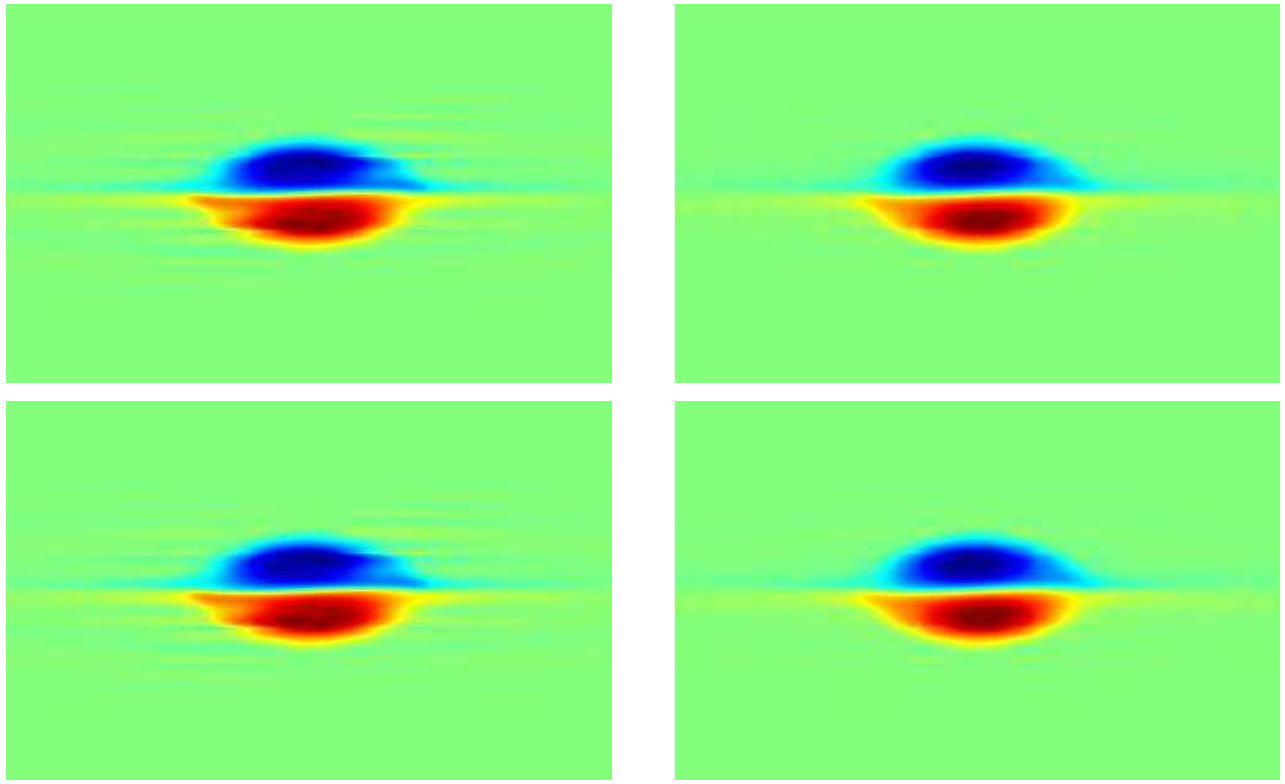
10×10 elements of degree 5, Poisson brackets, Ψ



10×10 elements of degree 5, Poisson brackets, Φ

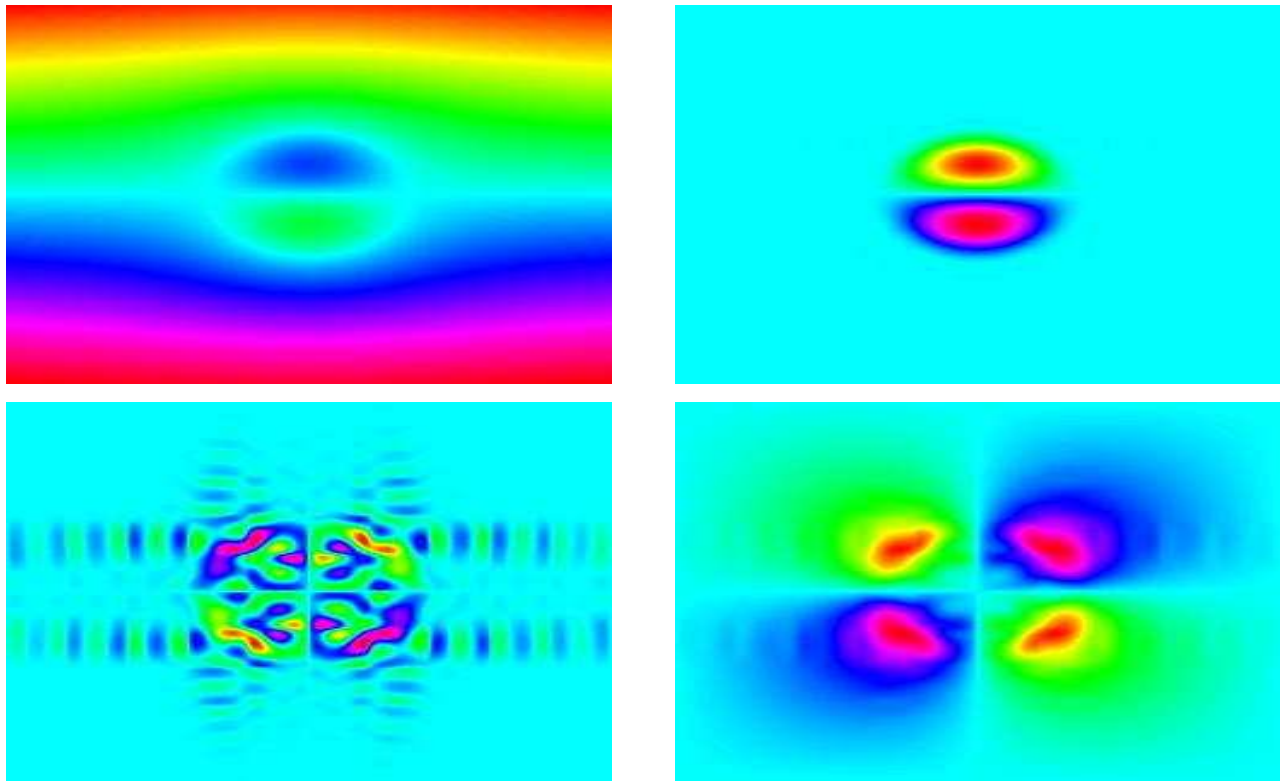


10×10 elements of degree 5, Poisson brackets, C



10×10 elements of degree 5, C^1 filtering

Time $t = 0.5$



C1 continuous elements - degrees of freedom

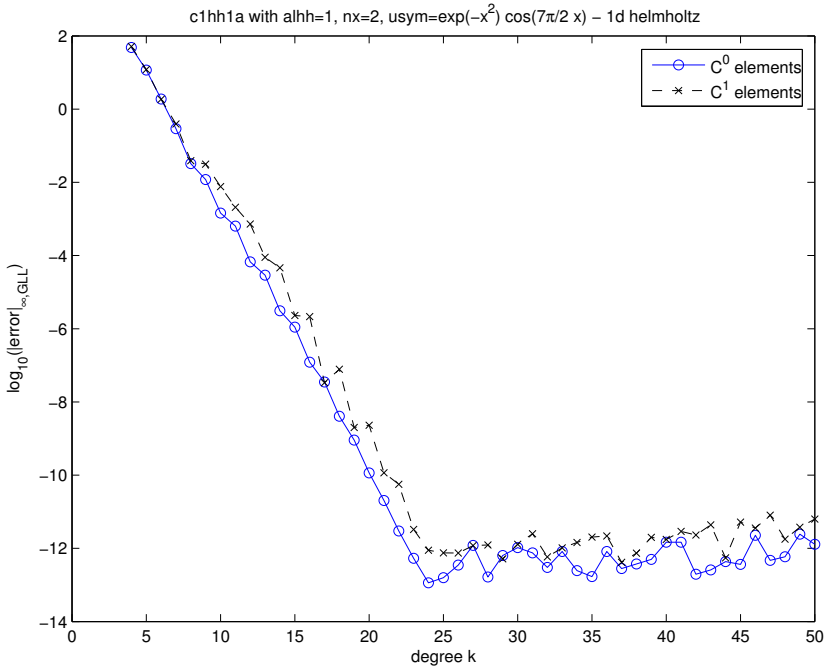
We work on rectangles or mapped rectangles. The degrees of freedom in the interior are the standard nodal degrees of freedom, while on the boundary we have both function values and a normal derivative (at least in the unmapped case). Taking tensor products of these one-dimensional degrees of freedom gives us degrees of freedom involving u_x, u_y and u_xy in the corners.

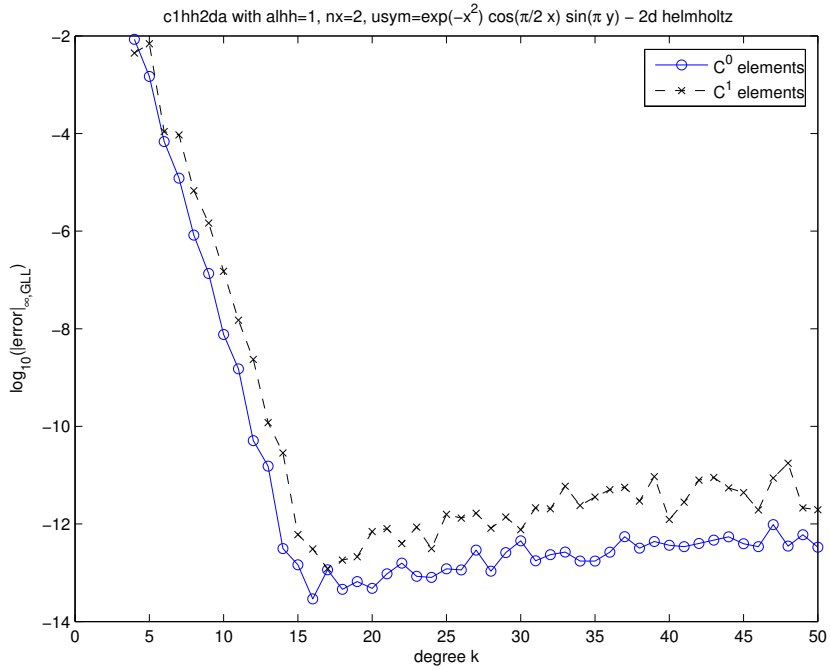
Can write down relatively easily mapping to normal spectral element degrees of freedom and can use the same machinery. In the regular case (rectangle split into smaller rectangles), the subassembled form for Laplace or Poisson is again a generalized Sylvester equation which can be solved fast in the same way.

On mapped elements, need either to fix directional derivatives like u_x, u_y, and/or u_{xy}, or need to fix normal derivatives for each degree of freedom. This is going to involve an extra chain-rule, and mess up some structure on the boundaries.

Some examples for modules for C1 elements on the next few slides. I am close to have the complete environment that I have for C0 for C1, but, unfortunately, no results for complete runs with nice pictures yet.

C1 continuous elements - example - 1D, 2D





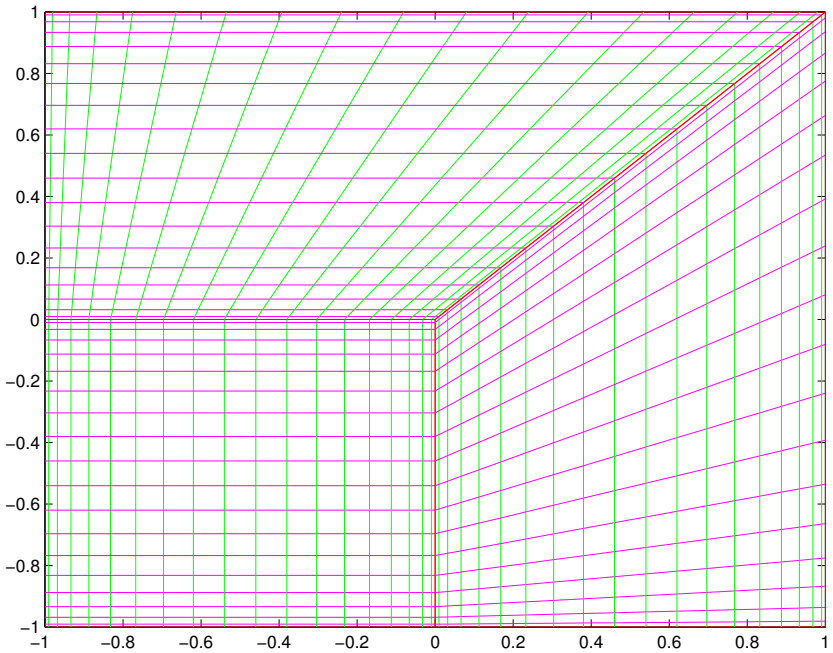
Mapped elements

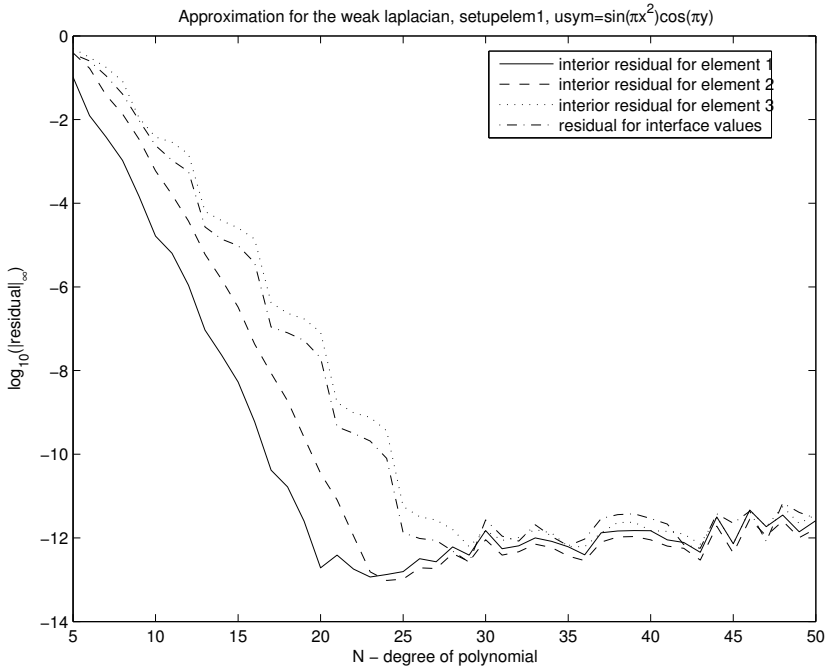
Using chain rule and a bilinear respresentation of the element mapping (straight-line quadrilaterals) or a representation of the element mapping in the same basis as the spectral elements, one can write down explicitly the Jacobian and all the needed derivatives of the mapping and such as combinations of tensor-product operators and point-wise multiplication. The expressions look more involved and somewhat harder to code, but they still execute close to peak for the Matrix-Vector product. Element-wise preconditioners etc. are also relatively easy.

To solve, one needs to perform static condensation (compute the Schur complement) and then solve the Schur complement with some standard solver, such as SuperLU or so.

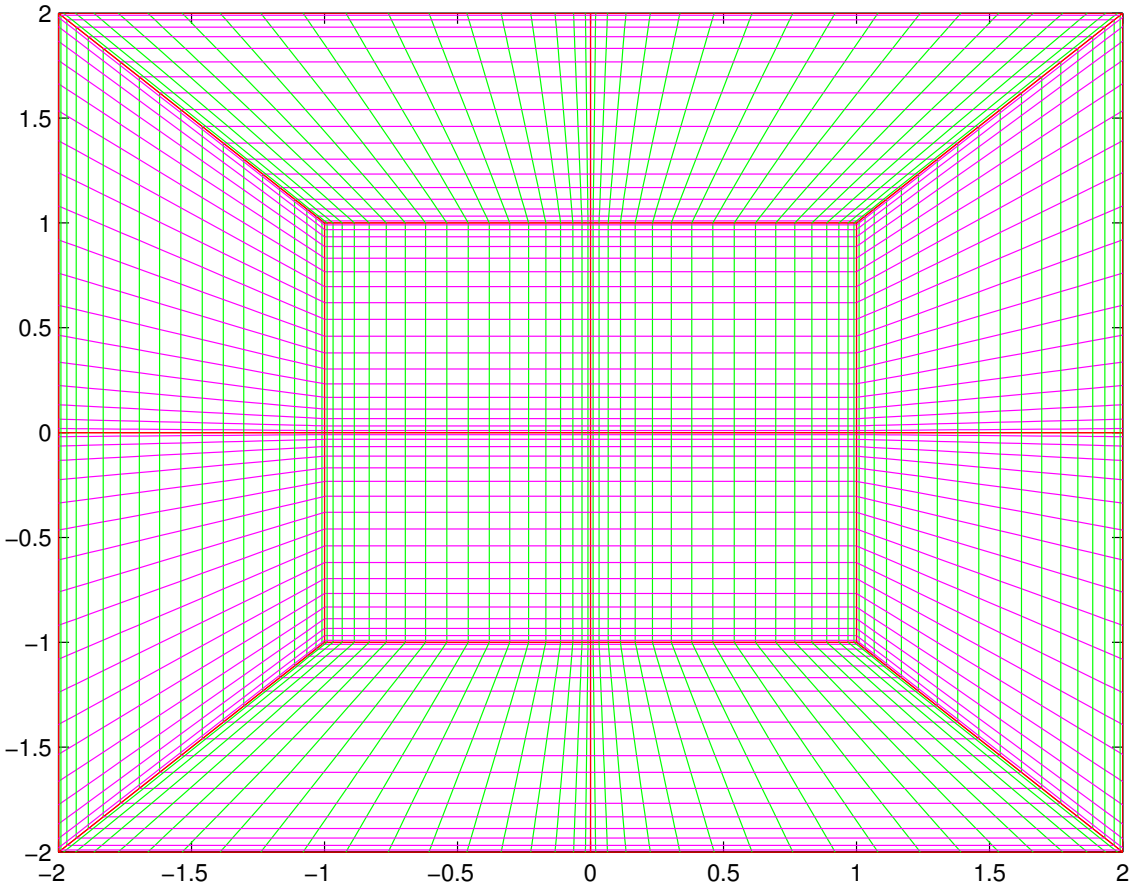
I have the pieces of the code, but development is more painful, since solution takes more time, and I cannot develop and debug code in MATLAB as fast as for the other parts, and writing fast C code is tricky as well (also coding interfaces to all the libraries one needs).

Example 1 - 3 elements, approximation of Laplacian





Example 2 - 12 elements



Implementation in C

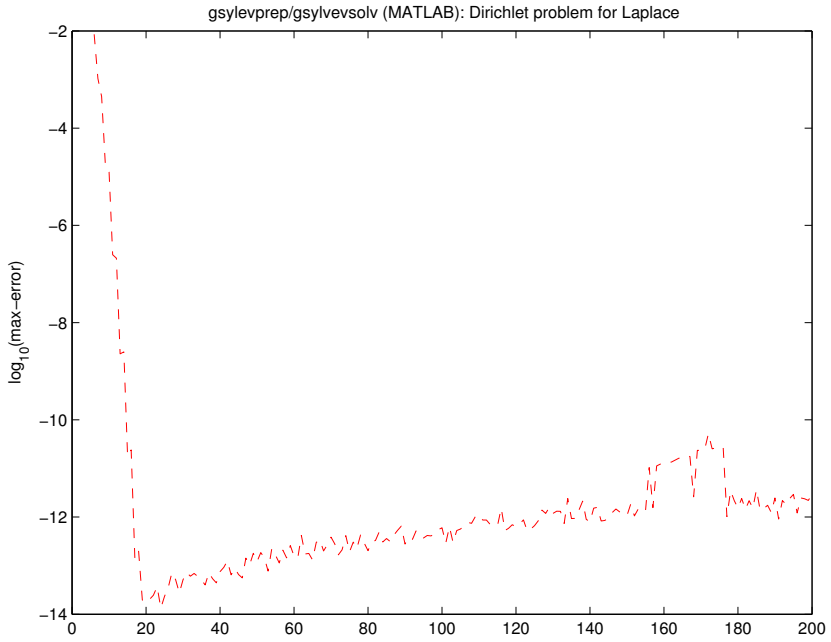
The idea of the implementation is very simple: essentially code all the matrix operations and algorithms that are written in high-level form in my MATLAB code in C using LAPACK, BLAS and other libraries as needed.

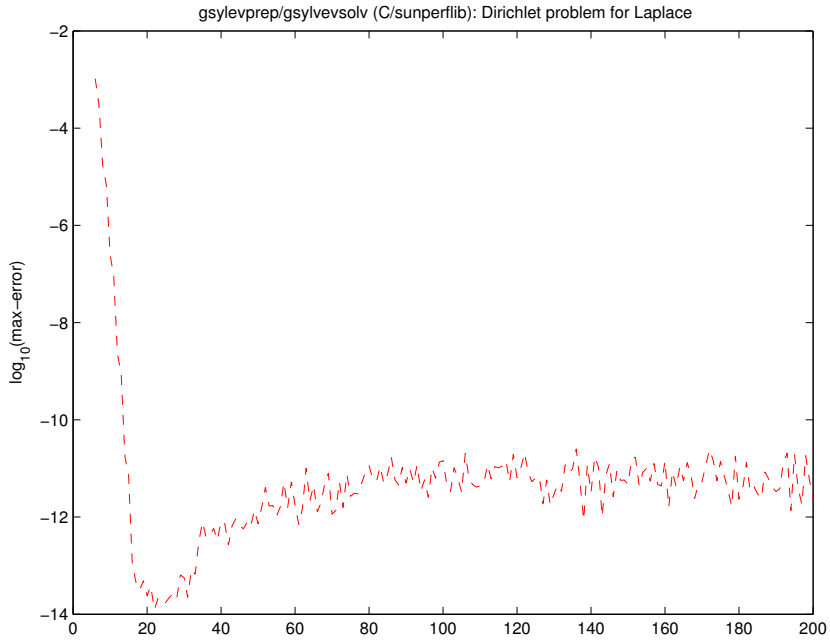
Figuring out compilation and optimization for vendor specific libraries is interesting, and MATLAB does a lot of things (like eigenvalue computations and other matrix operations) very efficient and with lot of hidden details that have to be coded explicitly when using LAPACK/SUNPERFLIB/ATLAS etc.

By now, there is a reasonable code base for SUN and for Linux Pentium, but I am still working on the optimization and on more complete runs. (A sketchy version of the MATLAB code runs, but I am not able to show complete runs yet.)

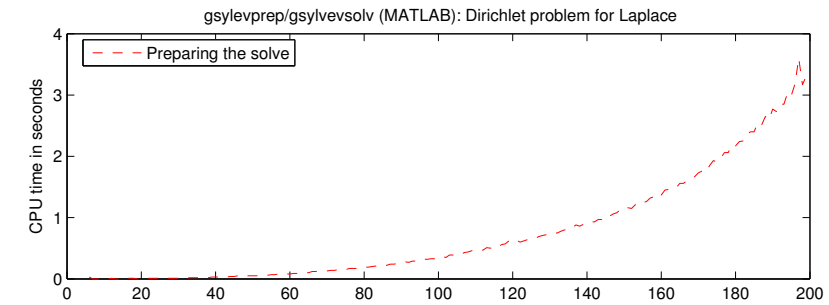
I will show a few results for modules on the next slides.

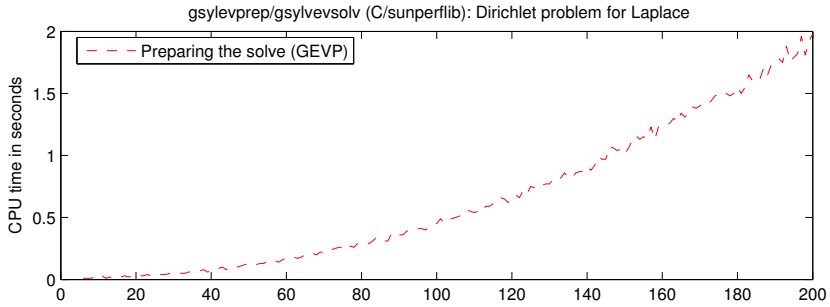
Implementation in C - comparing errors (Sun Blade)

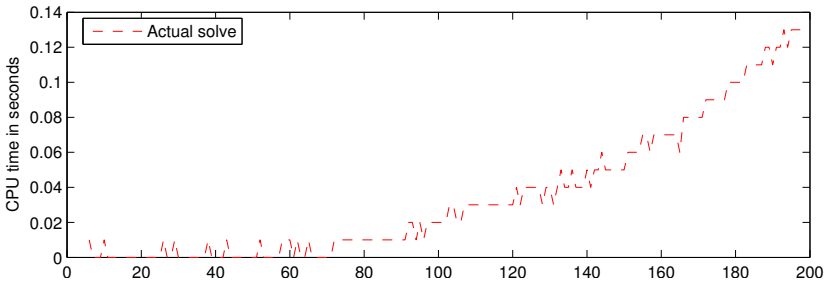


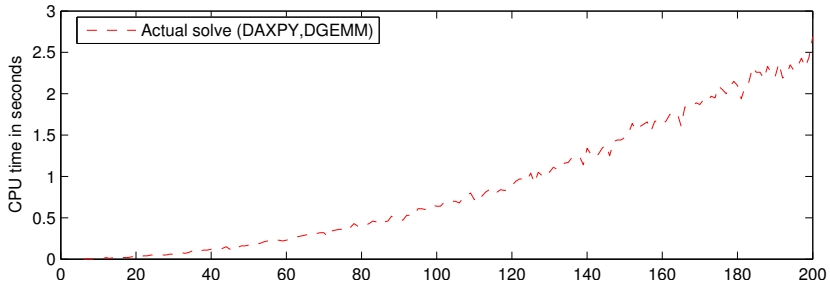


Implementation in C - comparing timing (Sun Blade)

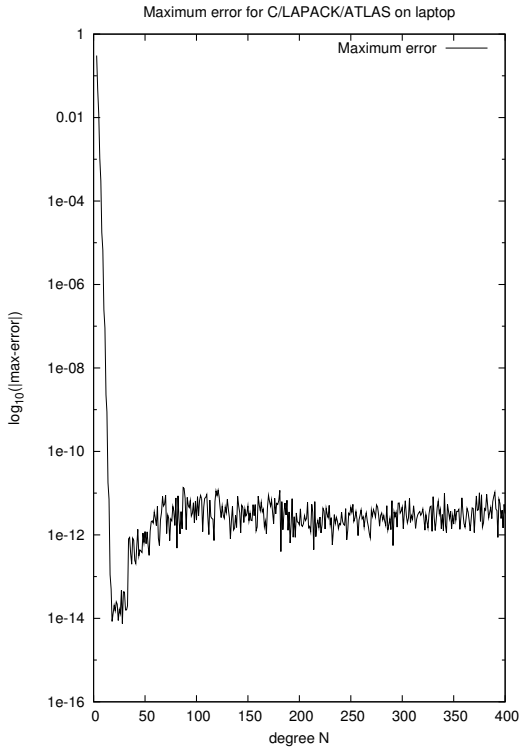


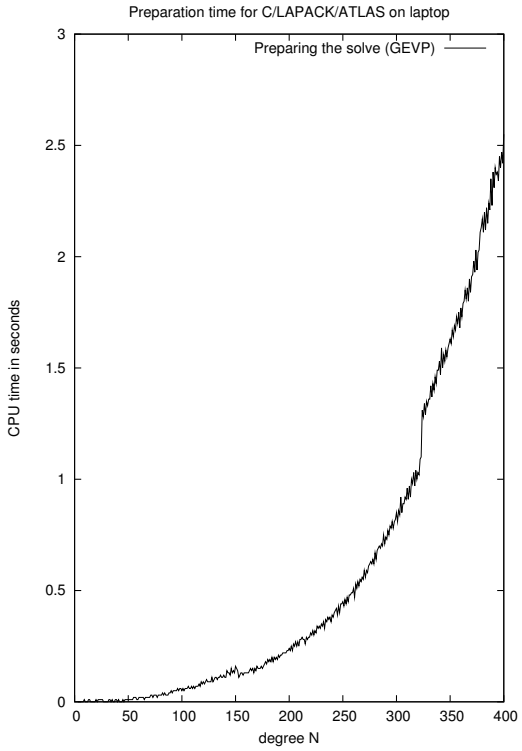


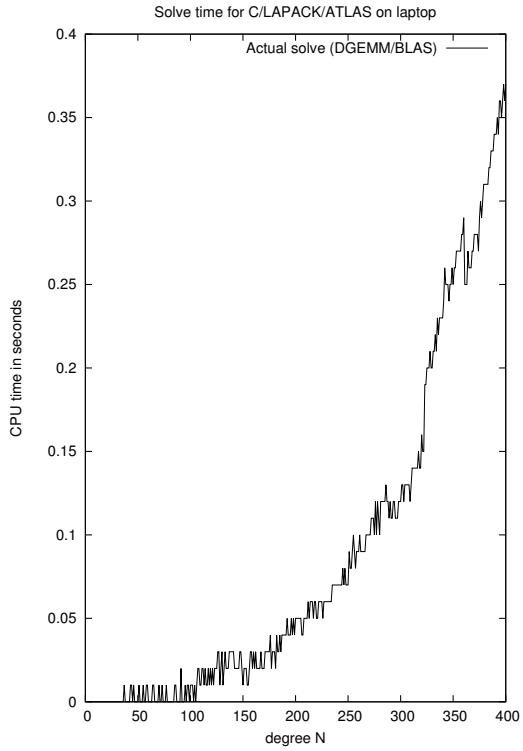




Implementation in C - results from my laptop







Implementation in C - observations

MATLAB is not only easier to program, but since it uses ATLAS and whatever else it can to optimize its matrix operations, it is actually quite competetive against a straigthforward coded C equivalent, even though that equivalent uses the vendor-specific, vendor-optimized library SUNPERFLIB.

On the other hand, I am very pleased with the speed of the C version on my laptop. ATLAS seems to make a real difference. Still playing around with some possibilities for optimization.

If speed for pentium becomes an issue, and if I want to use C on the Sun workstations at work, I will need to profile my code and find the performance bottleneck.

Numerical observations

- For some degrees and some kinds of interpolation, see stronger gradients in the vicinity of element interfaces. Stable, relatively local effect. Different kinds of filering are more or less successfull, a L^2 projection into the C^1 finite element space seems to get rid of the edge effects, but makes the results more blurry.

- Representing some intermediate variables as continuous or discontinuous (such as the derivatives in the poisson brackets) does not seem to make a difference. What about other treatments of this nonlinear term? - Higher order integration does same to make a difference and give more detailed results, there is not much difference between projected continuous derivatives and piecewise continuous in the Poisson brackets.

- "Real" C^1 elements seem to give quite encouraging preliminary results in the regular case.

Extension of algorithms

- Fully implicit? Use some leftover tricks for fast implementation of Jacobian for Newton method?
- Other problems: tearing mode (on it), coalescence ..
- More oriented toward inclusion into M3D?
- Full C^1 version, full mapped version, full C version, and then parellel version