Modeling Chemical Transport with Spectral Element Method

Cynthia Chan and Sam Loomis

August 8, 2014

Abstract

Climate and chemical transport modeling on a globe requires the use of unstructured grids, as well as conservation of mass and energy for a stable and accurate simulation. Chemical transport is often simulated using the finite volume method (FVM), which is explicitly conservative. However, the CESM and HOMME equations are formulated in the spectral element method (SEM), which is a variant of continuous Galerkin methods. Due to the interplay between chemical pollution and climate behavior, it is desirable for our formulation of both climate and chemical transport to be based on the same method. We tested the SEM for chemical transport models through both a serial and a parallel code. We found that it accurately and efficiently models chemical transport, though more in-depth comparison testing is necessary in future research.

Project Overview

The Finite Element Method (FEM) is a popular method for solving differential equations be dividing the domain into small regions and approximating function values on those elements separately. Until recently, it has not been believed to be locally conservative. Chemical transports are typically done using the Finite Volume Method (FVM), which is explicitly locally conservative. However, climate models such as CESM and the HOMME equations use the FEM to simulate meteorological phenomena on the globe. To allow for a more complete model which includes atmospheric interactions with chemicals, it is desirable to also use the same method for both climate and chemical transport.

Mark Taylor [2] has shown relatively recently that the Spectral Element Method, a type of FEM, is explicitly locally conservative. It is also simpler to solve than most FEM problems, because the mass matrix is diagonal and easy to invert.

Our goal is to test the advantage of using SEM to model chemical transport in the form of the equation:

$$\frac{\partial u_{(\alpha)}}{\partial t} = D_{(\alpha)}\nabla^2 u_{(\alpha)} - \vec{v}\cdot\vec{\nabla}u_{(\alpha)} + R_{(\alpha)}\left(u,\vec{x},t\right)$$

where $u_{(\alpha)}$ represent various chemical species, indexed by α. $D_{(\alpha)}$ is the diffusion coefficient, $\vec{v}$ the wind velocity and $R_{(\alpha)}$ represents the production rate of the species due to chemical reactions or manmade sources. We want to compare the efficiency and accuracy of the SEM to the FVM on these sorts of problems.

The ultimate goal is to integrate SEM chemical transport models with CESM. Our plan is to take two simultaneous paths, writing a serial code to test the 1D model and a parallel code to generalize to higher dimensions.

Part I

Serial code

THE MODEL

Based on the chemical equation on two species Cl_2 and Cl,

$$Cl_2 \rightleftharpoons Cl + Cl$$

we can construct a differential equation on $u_1 = [Cl_2]$ and $u_2 = [Cl]$.

$$\frac{\partial [Cl_2]}{\partial t} = d_1 \frac{\partial^2 [Cl_2]}{\partial x^2} - 0.001[Cl_2] + 0.05[Cl]^2$$
$$\frac{\partial [Cl]}{\partial t} = d_2 \frac{\partial^2 [Cl]}{\partial x^2} + 0.002[Cl_2] - 0.1[Cl]^2$$

Generalize it as an 1-dimensional time-dependent differential equation, which is exactly the equation that will be used to test the serial code:

$$\frac{\partial u_{(\alpha)}}{\partial t} = d_{(\alpha)} \frac{\partial^2 u_{(\alpha)}}{\partial x^2} + R_{(\alpha)}(u)$$

(1) $u_{(\alpha)}$ represent various chemical species, indexed by α.

(2) $d_{(\alpha)}$ is the diffusion coefficient.

(3) $R_{(\alpha)}$ represents the production rate of the species due to chemical reactions or manmade sources. In this chemical equation, $R_{(\alpha)} = k_{1(\alpha)} u_1 + k_{2(\alpha)} u_2^2$.

Basically, the model is formulated as a continuous Galerkin polynomial-based sepectral finite-element method. An important key of formulating this model is that using Gauss-Lobatto quadrature to approximate integration. When combined with a nodal basis that interpolates the quadrature nodes, one obtains a diagonal mass matrix.

More detailed information of using these methods in the serial code is as following:

0.1 discrete spaces for the SEM

Let x denotes the coordinates of the points in the domain $[a, b]$. Decompose the domain Ω into M elements $[a = a_0, b_0]$, $[a_1, b_0]$, ..., $[a_{M-1}, b_{M-1} = b]$. Denote the m-th element $[a_m, b_m]$ by Ω_m, and each element can be mapped to a reference interval $[-1, 1]$. Let r denotes the coordinates in the reference interval, then we can denote the map by

$$x = x(r, m) \tag{1}$$

, where $m = 0, 1, 2, ..., M-1$ denotes the m-th element.
Of course, if the domain is decomposed evenly into M elements, the mapping function of each element will be the same as

$$x(r, m) = \frac{b-a}{2M}(r+1) + \frac{b-a}{M} m \tag{2}$$

0.2 construct the basis functions

0.2.1 Gauss-Lobatto Quadrature

By Gaussian-Lobbato quadrature rule with 4 points, the nodal points are obtained as $\{-1, -\frac{1}{\sqrt{5}}, \frac{1}{\sqrt{5}}, 1\}$, denoted by $\{\xi_0, \xi_1, \xi_2, \xi_3\}$. And the corresponding weights of the points are $\{\frac{1}{6}, \frac{5}{6}, \frac{5}{6}, \frac{1}{6}\}$, denoted by w_0, w_1, w_2, w_3.

These nodal points are not only used to approximate the integration, but also to construct the basis functions.

0.2.2 Basis Functions

On the reference interval $[-1, 1]$, construct the basis functions $\{\phi_0(r), \phi_1(r), \phi_2(r), \phi_3(r)\}$ as Lagrange polynomials that interpolate the 1D degree-3 Gauss-Lobatto nodes $\{\xi_0, \xi_1, \xi_2, \xi_3\}$ mentioned above, such that

$$\phi_i(r) = \prod_{j \neq i} \frac{x - \xi_j}{\xi_i - \xi_j} \tag{3}$$

$$\phi_i(\xi_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{4}$$

0.2.3 Spanning Basis Functions

By mapping the basis functions from the reference domain to each element using the mapping function(1), we obtain a set of basis functions $\{\phi_i(x(r,m)) : i = 0,1,2,3 \text{ and } m = 0,1,...,M-1\}$ on the domain $[a,b]$.
Then we cam approximate the value of $u_\alpha(x)$ by linear combination of the polynomial basis functions on $[a,b]$.

$$u_a(x) = u_a(x(r,m)) = \sum_{i=0}^{3} u_\alpha(x(\xi_i,m))\phi_i(x(r,m)) \tag{5}$$

0.3 Formulate the Differential Equation

0.3.1 Weak Form

By following 1D continuous Galerkin Method and using the basis function mentioned above, we can formulate the differential equation as the following steps.

$$\int_\Omega v\frac{\partial u_\alpha}{\partial t}\ dx = \int_\Omega d_\alpha v\frac{\partial^2 u_\alpha}{\partial x^2}\ dx + \int_\Omega vR_\alpha(u)\ dx \tag{6}$$

$$\int_\Omega v\frac{\partial u_\alpha}{\partial t}\ dx = -\int_\Omega d_\alpha\frac{\partial v}{\partial x}\frac{\partial u_\alpha}{\partial x}\ dx + \oint_{\partial\Omega} d_\alpha v\frac{\partial u_\alpha}{\partial x}\ dx + \int_\Omega vR_\alpha(u)\ dx \tag{7}$$

$$\sum_m\int_{\Omega_m} v\frac{\partial u_\alpha}{\partial t}\ dx = -\sum_m\int_{\Omega_m} d_\alpha\frac{\partial v}{\partial x}\frac{\partial u_\alpha}{\partial x}\ dx + \oint_{\partial\Omega} d_\alpha v\frac{\partial u_\alpha}{\partial x}\ dx + \sum_m\int_{\Omega_m} vR_\alpha(u)\ dx \tag{8}$$

(1)$m = 0,1,...,M-1$ denote the order of each element.
(2)$\Omega_m = I_m = [a_m, b_m]$ is the domain of m-th element.
(3)v is a continuous test function on the domain.

0.3.2 Bilinear Form

By mapping each element to the reference domain, the following equation is obtained.

$$\sum_m\int_{-1}^{1} v\frac{\partial u_\alpha}{\partial t}[det]_m\ dr = -d_\alpha\sum_m\int_{-1}^{1}\frac{\partial v}{\partial r}\frac{\partial u_\alpha}{\partial r}\frac{1}{[det]_m}\ dr + d_\alpha\oint_{\partial\Omega} v\frac{\partial u_\alpha}{\partial x}\ dx + \sum_m\int_{-1}^{1} vR_\alpha(u)[det]_m\ dr \tag{9}$$

where $[det]_m = \frac{\partial x(r,m)}{\partial r}$.

Firstly, approximate the function $u_a(x)$ as the equation(4).

$$u_a(x) = u_a(x(r,m)) = \sum_{i=0}^{3} u_\alpha(x(\xi_i,m))\phi_i(x(r,m))$$

Secondly, take the test function v as each basis function.

$$v = \phi_{i^*}(x(r,m^*)) \tag{10}$$

where $i^* = 0,1,2,3$ and m^* varies from 0 to $M-1$.

Then use Guass-Lobbato quadrature to approximate the integration. That is to say, for a function $f(x)$, approximate the integration as follows.

$$\int_{-1}^{1} f(x)dx = \sum_{j=0}^{3} w_j f(\xi_j) \tag{11}$$

As a result, we can obtain the following equation.

$$\sum_{j=0}^{3} M_{i^*j} \frac{\partial u_a(x(\xi_j, m^*))}{\partial t} = -d_\alpha \sum_{j=0}^{3} D_{i^*j} u_a(x(\xi_j, m^*)) + \sum_{j=0}^{3} M_{i^*j} R_\alpha(\vec{u}(x(\xi_j, m^*)) + d_\alpha \oint_{\partial\Omega} v \frac{\partial u_\alpha}{\partial x}\, dx \quad (12)$$

$$[M] \frac{\partial \vec{u}_a(x(\vec{\xi}, m^*))}{\partial t} = -d_\alpha [D] \vec{u}_a(x(\vec{\xi}, m^*)) + [M] R_\alpha(\vec{u}) + d_\alpha \oint_{\partial\Omega} v \frac{\partial u_\alpha}{\partial x}\, dx \quad (13)$$

where

$$\oint_{\partial\Omega} v \frac{\partial u_\alpha}{\partial x}\, dx = \begin{cases} -\frac{\partial u_\alpha}{\partial r}(x(\xi_0, 0)) = -\sum_j u_\alpha(x(\xi_j, 0)) \phi_j'(\xi_0) \quad if\ v = \phi_0(r, 0); \\ \frac{\partial u_\alpha}{\partial r}(x(\xi_3, M-1)) = \sum_j u_\alpha(x(\xi_j, M-1)) \phi_j'(\xi_3) \quad if\ v = \phi_3(r, M-1); \\ 0 \quad otherwise \end{cases} \quad (14)$$

and $\vec{u}_a(x(\vec{\xi}, m)) = (u_a(x(\xi_0, m)), u_a(x(\xi_1, m)), u_a(x(\xi_2, m)), u_a(x(\xi_3, m)))^T$.

$[M]$ is considered as the Mass Matrix of the m-th element. The entries $M_{ij} = \int_{-1}^{1} \phi_i(r)\phi_j(r)[det]_m\, dr$, where $[det]_m$ is the determinate dependent on the mapping from the reference domain to the m-th element $[m-1, m]$. And in this case, $[det]_m = \frac{b-a}{2M}$ for each element. Because we are using Gaussian-Lobatto quadrature to approximate the integration, and at the same time, the basis function $\phi_i(r)$ are Lagrange polynomials on the nodes ξ_i, we can conclude that the Mass Matrix is a diagonal matrix such that

$$M_{ij} = \int_{-1}^{1} \phi_i(r)\phi_j(r)[det]_m dr = \begin{cases} [det]_m w_i \quad i = j \\ 0 \quad i \neq j \end{cases} \quad (15)$$

(4) [D] is considered as the diffusion matrix. The entries are valued as the following equation.

$$D_{ij} = \int_{-1}^{1} \phi_i'(r)\phi_j'(r) \frac{1}{[det]_m}\, dr \quad (16)$$

Reference to the code

The quadrature points with its weights, element mass matrix and element diffusion matrix are all stored in `templs.h`.

0.4 assembly

Construct the element equation(13) on each element $m^* = 0, 1, 2, ..., M-1$, and then assemble them into a global equation.

$$[GM] \frac{\partial U_\alpha}{\partial t} = -[GD] U_a + [GM] R_a(\vec{U}) \quad (17)$$

(1) $U_a = (u_a(\xi_0, 0), u_a(\xi_1, 0), u_a(\xi_2, 0), u_a(\xi_0, 1), u_a(\xi_1, 1), u_a(\xi_2, 1),\ ...\ , u_a(\xi_0, M-1), u_a(\xi_1, M-1), u_a(\xi_2, M-1), u_a(\xi_3, M-1))^T$. $\vec{U} = (U_0, U_1, ..., U_{spi-1})$. Note that: $u_a(\xi_3, m) = u_a(\xi_0, m+1)$.

(2) $[GM]$ is the global Mass Matrix assembled by the element Mass Matrix $[M]$, while $[GD]$ is the global Diffusion Matrix assembled by the element Diffussion Matrix $[D]$ combined with the diffussion coefficient d_α and the integration on boundary.

```
void GlobalMass(double *diaGM, int M,double det);
/*output: the Global Mass Matrix for each species*/
```

`Algorithm of assembling the Global Mass Matrix`:

Initialize `[DiaGM]=0`.

Do the following for `m=0:(M-1)`

1. Obtain diagonal entries `[DiaEM]` of the m-th element by the function ElementMass();

2. do for `i=0:(NNODL-1)`

```
DiaGM[i+(NNODL-1)*m]=DiaEM[i+(NNODL-1)*m]+DiaEM[i];
```

Then the diagonal entries of the Global Mass Matrix is obtained as `[DiaGM]`.

```
void GlobalDiffusion(double** GM, int s,int element,double det,int spi);
/*input: the order of the species*/
/*outpur: diffussion Global Matrix for the s-th species*/
```

`Algorithm of assembling the Global Diffusion Matrix:`

1. Apply the same algorithm mentioned above as assembling the Global Mass Matrix to assemble `[D]` into a Global Diffusion Matrix `[GD]`;

2. Do with the integration on the boundary referring to equation(13):

do the following for `i=0:(NNODL-1)` :

`GD[0][i]`=`GD[0][i]`+$\phi_i'(\xi_0)$;

`GD[n][i]`=`GD[n][i]`$-\phi_i'(\xi_3)$; where `n=(NNODL-1)*M+1`|

3. Multiply each entry of the Global Diffusion Matrix with the diffusion coefficient d_α.

As a result, the Global Diffusion Matrix $[GD]$ is obtained, which is dependent on the order of the species.

Multiply the equation(17) with the matrix $[GM]^{-1}$ from the left. The following equation is obtained.

$$\frac{\partial U_\alpha}{\partial t} = -[GM]^{-1}[GD]U_a + R_\alpha(\vec{U}) \tag{18}$$

```
void ivsM_D(double** ivsMD, int s, int M, double det, int spi);
/*input: the order of the species*/
/*output: "inverse of Global Mass Matrix" multiply the Global diffusion Matrix */
```

i.e. output the matrix $[GM]^{-1}[GD]$ for the s-th species

0.5 Euler Backward Method

To solve the value of U_a at time $T = hN$. We use Euler Backward Method on the equation(11) with time step h.

$$U_a^{(n+1)} = U_a^{(n)} - h[GM]^{-1}[GD]U_a^{(n+1)} + hR_\alpha(\vec{U}^{(n+1)}) \tag{19}$$

1. For n=0, input the initial value $U_a^{(0)}$.

2. Solve equation(19) and update $U_a^{(n+1)}$ for N many times. Then we can obtain the value of U_a at time $T = hN$.

0.6 Newton Method

As the equation(19) is nonlinear because of the term $R_\alpha(\vec{U})$, we need to apply Newton Method the solve equation(18). A function $\vec{F}$ is construct for Newton Method.

$$\vec{F} = \begin{pmatrix} F_1(\vec{x}) \\ F_2(\vec{x}) \\ \dots \\ F_{SPI}(\vec{x}) \end{pmatrix} = 0 \tag{20}$$

where

$$F_\alpha(\vec{x}) = x_\alpha - U_\alpha + h([GM]^{-1}[GD]x_\alpha - R_\alpha(\vec{x})) \tag{21}$$

In Newton Method, we solve the equation(20) instead of solving the equation(19) in Euler Method. To solve the equation(20), we need to construct the Jacobian Matrix of the function $\vec{F}$.

$$J = \frac{\partial \vec{F}}{\partial \vec{x}} = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots \frac{\partial F_1}{\partial x_{SPI}} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots \frac{\partial F_2}{\partial x_{SPI}} \\ \cdots & \cdots & \cdots \\ \frac{\partial F_{SPI}}{\partial x_1} & \frac{\partial F_{SPI}}{\partial x_2} & \cdots \frac{\partial F_{SPI}}{\partial x_{SPI}} \end{pmatrix} \tag{22}$$

By equation(21), it is easy to obtain the partial derivative.

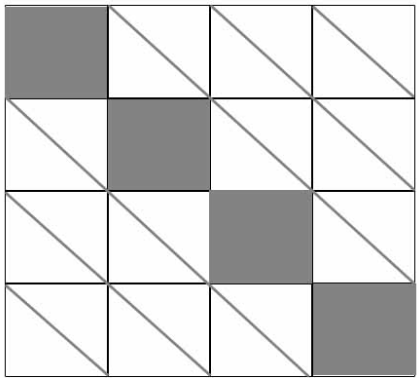
$$\frac{\partial F_j}{\partial x_i} = \begin{cases} I + h[GM]^{-1}[GD] - h\frac{\partial R_j(\vec{x})}{\partial x_i} I & i = j \\ -h\frac{\partial R_j(\vec{x})}{\partial x_i} I & i \neq j \end{cases} \tag{23}$$

Note that: if $i \neq j$, then $\frac{\partial F_j}{\partial x_i}$ will be a diagonal block.

While applying the Newton Method, the following Linear Algebra Equation need to be solved.

$$J(\Delta \vec{x}) = -F(\vec{x}^*) \tag{24}$$

And the structure of the Jacobian Matrix J is as the following graph,



which is not banded enough. Hence, it is essential to refine the matrix. To achieve this, we need to rearrange the rows and columns in the matrix.

In equation(24), $\vec{x}^* = \vec{U} = (U_0, U_1, ..., U_{spi-1})^T$, where $U_a = (u_a(\xi_0, 0), u_a(\xi_1, 0), u_a(\xi_2, 0), u_a(\xi_0, 1), u_a(\xi_1, 1), u_a(\xi_2, 1), \ ... \ , u_a($
$1), u_a(\xi_1, M-1), u_a(\xi_2, M-1), u_a(\xi_3, M-1))^T$. To make the matrix banded, rearrange the elements in $\vec{x}^*$ to obtain a new vector $\vec{x}'^*$ such that:

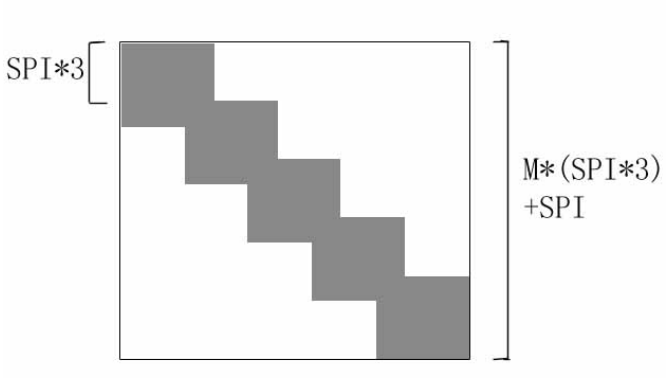
$$\vec{x}'^* = (\vec{u}(\xi_0, 0), \vec{u}(\xi_1, 0), \vec{u}(\xi_2, 0), \vec{u}(\xi_0, 1), \vec{u}(\xi_1, 1), \vec{u}(\xi_2, 1), \ ... \ , \vec{u}(\xi_0, M-1), \vec{u}(\xi_1, M-1), \vec{u}(\xi_2, M-1), \vec{u}(\xi_3, M-1))^T \tag{25}$$

where

$$\vec{u}(\xi_i, m) = (u_0(\xi_i, m), u_1(\xi_i, m), ..., u_{spi-1}(\xi_i, m))^T \tag{26}$$

For convenience, denote the RHS of equation(25) by $\overrightarrow{U'}$. Then $\vec{x}'^* = \overrightarrow{U'}$.

There is a permutation π on $\{0, 1, 2, ..., N-1\}$, which maps the elements in $\vec{x}^*$ to the elements in $\vec{x}'^*$, where $N = spi * ((NNODL - 1) * M + 1)$. Using the same permutation π, we can rearrange the rows and columns of the matrix J to obtain a new matrix J'. And the structure of the refined matrix J' should be as the following graph:



As the Jacobian Matrix can be huge, it is a good property that J' is banded. Hence, we can apply `lapack` to solve the linear equation.

Usually there should be a boundary condition while solving the differential equation. I am going to apply the boundary condition to the Jacobian Matrix by making $\Delta\vec{x}$ equals to zero at the boundary points.

```
void JacobF(double** J, double *u, double h, int M, double det, int spi);
/*input: an array u */
/*output: the refined Jacobian Matrix J' at u */
```

`Algorithm of constructing the refined Jacobian Matrix:`

SET: `N=spi*((NNODL-1)*M+1); n=(NNODL-1)*M+1;`

1. Initialize `J[i][j]=0;`

2. Referring to equation(23), input the diffusion part $I + h[GM]^{-1}[GD]$ to J.

(1) do `i=0:(N-1)`, `J[i][i]=1;`

(2) do `s=0:(spi-1)`, `i=0:(n-1)`, `j=0:(n-1)`,

`J[s+i*spi][s+j*spi]=J[s+i*spi][s+j*spi]+h(`$[GM]^{-1}[GD]$`[i][j])`.

Note that: the matrix $[GM]^{-1}[GD]$ is dependent on the order of the species `s`.

3. Referring to equation(23), input the source part $-h\frac{\partial R_j(\vec{x})}{\partial x_i}I$ to `J`.

4. Add the boundary condition.

do `s=0:(spi-1)`

(1) do `j=0:(N-1)`

`J[s][j]=0; J[N-(s+1)][j]=0;`

(2) `J[s][s]=1; J[N-(s+1)][N-(s+1)]=1;`

As J' is a banded matrix, we can call `LAPACK` solver to solve the equation

$$J'(\Delta\vec{x}') = -F(\vec{x}'^{*}) \tag{27}$$

Now, the equation(19) at n=k can be solved applying typical Newton Method Algorithm :

1. Set the tolerance of error to be `tolerance=`ϵ.

2. Initialize $\vec{x}'^{*} = \overrightarrow{U'^{(n)}}$.

3. Solve $\Delta\vec{x}'$ in the equation(27).

4. Update $\vec{x}'^{*}$: $\vec{x}'^{*} = \vec{x}'^{*} + \Delta\vec{x}'$.

5. If $\|\Delta\vec{x}'\| > \epsilon$, go back to step 3.

6. Obtian the result $\overrightarrow{U'^{(n+1)}} = \vec{x}'^{*}$ with `tolerance=`ϵ.

```
void NEWTON(double *u_0, double *u_1, double h, double tolerance, int M, double det, int spi)
/*input: an array u_0 */
/*output: the solution u_1 for equation(19)*/
```

Note: the array we use in Newton Method the a refined vector $\vec{U}'$, but in equation(19) referring to Euler Method, the vector is the unrefined one $\vec{U}$. In the serial code, of course we only use the refined vector $\vec{U}'$.

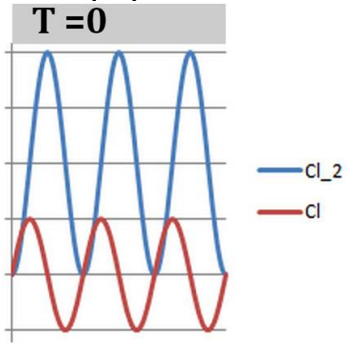
0.7 Solution

By all the method metioned above, we can solve the value of $u_\alpha(\xi_i, m)$ for $\alpha = 0, 1, ..., spi-1$, $i = 0, 1, 2, 3$ and $m = 0, 1, 2, ..., M-1$ at time $T = hN$. Apply these coefficient to the basis functions, then we can obtain the approximation function on the domain $[0, M]$.

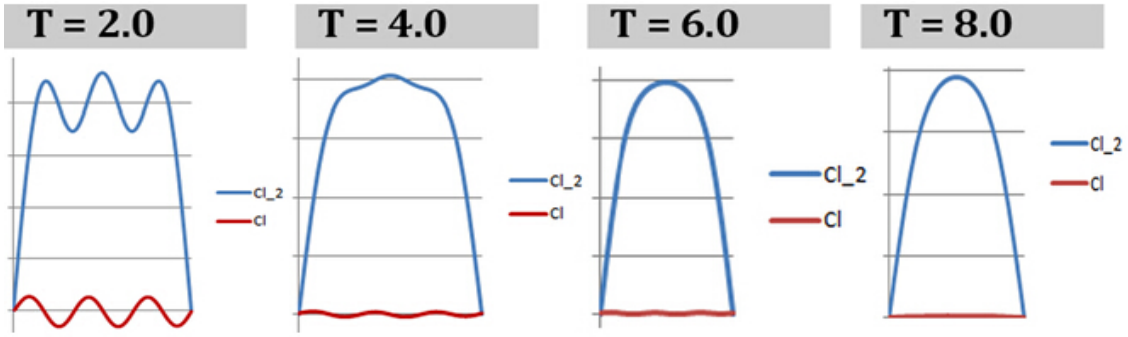
$$f(x_\alpha) = f(x_\alpha(r, m)) = \sum_{j=0}^{3} u_\alpha(\xi_j, m)\phi_j(r) \tag{28}$$

0.8 Simple example

Here is a simple example for testing. The following graph shows the initial value of 2 species on the domain $[0, 6]$ at time T=0.



Let the number of elements `M=30`, `tolerance=0.000001` for Newton Method and do 100 steps in the Euler Method. The following graph shows the quantity of those 2 species at time T=2.0, 4.0, 6.0, 8.0.

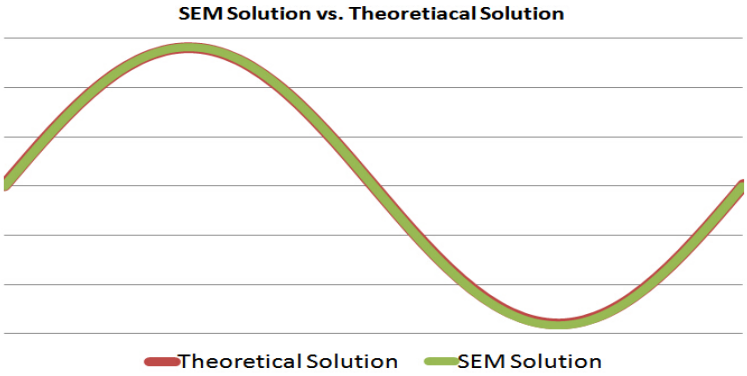


0.9 Testing of accuracy

To know whether the code is giving us a basically correct answer, a good way is to compare the result given by the code with the analytical result.

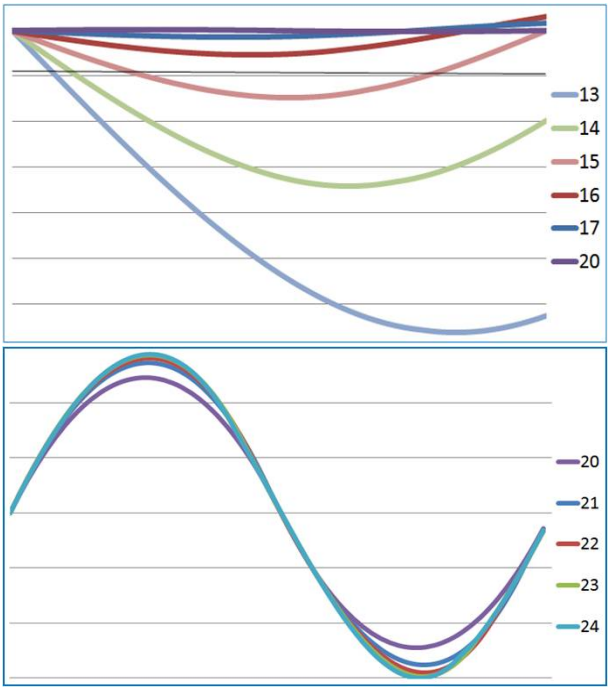
Differential Equation: $\frac{\partial u}{\partial t} = 0.1\frac{\partial^2 u}{\partial x^2} + 0.1u$;
Initial Value: $u(x,0) = sin(\pi x)$ at $T = 0$;
Analytical Solution: $u(x,t) = e^{0.1-0.1\pi^2 T} sin(\pi x)$;

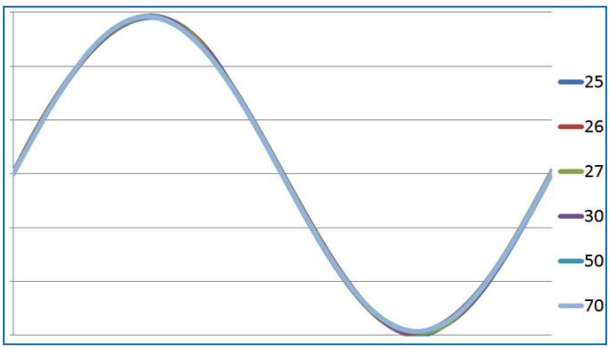
On the basis of above information, the comparison graph is shown as follows.



0.10 Convergence

By varying the number of element from 13 to 70, the convergence process is show as the following graph. Of course, it is a very rough way of testing the convergence by graphs. There should be further exploration on this part.





Part II
Parallel code

This code is a set of FORTRAN modules designed to help solve and integrate a general multi-dimensional SEM problem.

The code will consist of two main pieces: HESIOD and HOMER. HESIOD is a module which allows the user to package the mesh, fields and equation data. HOMER is the module with subroutines that will actually evolve the fields in time, using explicit or implicit methods, and with options to use splitting and different types of Newton's methods.

0.11 HESIOD

HESIOD contains four types: `mesh`, `fields`, `fn_ptr`, and `equation`.

`mesh`: Requires input of integer variables `num_elem`, `num_pt`, `num_lyr`, `deg`, and `num_dim`, which correspond to the number of elements, number of nodal points, number of layers, degree of polynomial interpolation, and the dimension of the embedding space, respectively. From these we can also calculate `num_quad` $=$ $(\texttt{deg}+1)^{\texttt{num\_dim}}$, which is the total number of quadrature points in the reference cube $[-1,1]^{\texttt{num\_dim}}$.

Also requires the arrays `structure` (`num_elem`, `num_quad`), `quad` (`deg` $+1$), `weights` (`deg` $+1$), and `coords` (`num_pt`, `num_dim`), which represent the structure of the matrix (assigning each quadrature point in each element to a nodal point), the quadrature points on $[-1,1]$, the weights for these points, and the coordinates of each nodal point in the embedded space. From these a number of arrays are calculated, the most important being: `diffs` (`num_dim`, `num_quad`, `num_quad`), which allows for the calculation of derivatives w.r.t. the reference cube; `grad` (`num_elem`, `num_dim`, `num_quad`, `num_quad`), which allows for calculation of derivatives w.r.t. the embedding space; and `jac` (`num_elem`, `num_quad`), which calculates the Jacobian determinant for each element.

`fields`: Requires as input: the integer `num_fields`, which corresponds to the total number of fields (counting different components of a vector/tensor field separately); `type` (`mesh`) `base_mesh`, which represents the mesh on which the fields are defined; and array `values` (`num_fields`, `base_mesh%num_pt`), which stores the values of the field at each nodal point. Upon declaration, this must be set to the initial values of the function.

We can switch between the nodal-point representation of the fields in `values` and a elemental-quadrature-point representation `elem_values` (`num_fields`, `base_mesh%num_elem`, `base_mesh%num_quad`) using the data in `base_mesh%structure`. This representation is redundant, containing the values at boundary points twice, but it makes a convenient representation such that each sub-array `elem_values` (:, `m`, :) can be passed on to the processor assigned to element `m` during the local evolution.

`fields` also contains a subroutine, `projectElemValues`, to perform the weighted sum required for the local evolution method, turning a discontinuous `elem_values` representation into a continuous one, where redundant points are made to agree by averaging them over the quadrature weights.

`fn_ptr:` Creates a general function format to represent the right-hand-side of the evolution equation. The required format of these functions is

```
function f(dim, num_fields, num_quad, diff, grad, Jw, u, q, t) result(ans)
    integer, intent(in) :: dim, num_fields, num_quad, q
    real(kind(0.0d0)), dimension(num_fields, num_quad), intent(in) :: u
    real(kind(0.0d0)), dimension(num_dim, num_quad, num_quad), intent(in) :: diff, grad
    real(kind(0.0d0)), dimension(num_quad), intent(in) :: Jw
    real(kind(0.0d0)), dimension(num_quad) :: ans
    real(kind(0.0d0)), intent(in) :: t
end function
```

`equation:` Requires as input: `num_dim`, `num_fields`, and `num_quad`; `num_split`, the number of split operators (an input of 0 means no splitting); the `type(fn_ptr(num_dim, num_fields, num_quad))` array `operators(num_fields, num_split)` which stores the info for $F^{(\alpha)}$ as a set of split operators; and `theta`, which represents the implicit parameter θ used in implicit evolution.

0.12 HOMER

For what follows, we consider a general time-dependent first-order PDE over the fields $u^{(\alpha)}$:

$$\frac{\partial u^{(\alpha)}}{\partial t} = F^{(\alpha)}[u]$$

No code is written for HOMER yet, but the general routines that must take place are set out below.

Local versus global evolution.

The first step of the Spectral Element Method is to take the weak form of the equations by taking an inner product:

$$\left\langle \Phi_l \frac{\partial u^{(\alpha)}}{\partial t} \right\rangle = \left\langle \Phi_l F^{(\alpha)}[u] \right\rangle$$

$$\left\langle \Phi_l \frac{\partial u^{(\alpha)}}{\partial t} \right\rangle = \sum_{m=1}^{M} \left\langle \Phi_l \frac{\partial u^{(\alpha)}}{\partial t} \right\rangle_{\Omega_m}$$

On a point interior to element $\bar{m}$, with $\vec{r}_l = \vec{r}\left(\vec{\xi}_{\vec{\imath}}, \bar{m}\right)$ this is

$$\left\langle \Phi_l \frac{\partial u^{(\alpha)}}{\partial t} \right\rangle = \left\langle \phi_{\vec{\imath}} \frac{\partial u^{(\alpha)}}{\partial t} \right\rangle_{\Omega_{\bar{m}}} = \sum_{\vec{\jmath}} \frac{\partial \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\jmath}}}{\partial t} \phi_{\vec{\imath}}\left(\vec{\xi}_{\vec{\jmath}}\right) w_{\vec{\jmath}} J_{\bar{m}}\left(\vec{\xi}_{\vec{\jmath}}\right) = \frac{\partial \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}}{\partial t} w_{\vec{\imath}} J_{\bar{m}}\left(\vec{\xi}_{\vec{\imath}}\right)$$

Meanwhile, we write

$$\left\langle \Phi_l F^{(\alpha)}[u] \right\rangle = \left\langle \Phi_l F^{(\alpha)}[u] \right\rangle_{\Omega_{\bar{m}}} = \mathbf{F}^{(\alpha)}_{\bar{m}|\vec{\imath}}$$

We expect that we can perform integration by parts on second derivatives in the calculation of this term. Resulting boundary terms are zero because $\vec{r}_l$ is interior to element $\bar{m}$.

Now

$$\frac{\partial \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}}{\partial t} = \frac{\mathbf{F}^{(\alpha)}_{\bar{m}|\vec{\imath}}}{w_{\vec{\imath}} J_{\bar{m}}\left(\vec{\xi}_{\vec{\imath}}\right)}$$

However, if $\vec{r}_l = \vec{r}\left(\vec{\xi}_{\vec{\imath}}, \bar{m}\right) = \vec{r}\left(\vec{\xi}_{\vec{\jmath}}, \bar{n}\right)$ with $\bar{m} \neq \bar{n}$, then

$$\left\langle \Phi_l \frac{\partial u^{(\alpha)}}{\partial t} \right\rangle = \sum_{m=1}^{M} \left\langle \Phi_l \frac{\partial u^{(\alpha)}}{\partial t} \right\rangle_{\Omega_m} = \left\langle \Phi_l \frac{\partial u^{(\alpha)}}{\partial t} \right\rangle_{\Omega_{\bar{m}}} + \left\langle \Phi_l \frac{\partial u^{(\alpha)}}{\partial t} \right\rangle_{\Omega_{\bar{n}}} = \frac{\partial \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}}{\partial t} w_{\vec{\imath}} J_{\bar{m}}\left(\vec{\xi}_{\vec{\imath}}\right) + \frac{\partial \mathbf{u}^{(\alpha)}_{\bar{n}|\vec{\jmath}}}{\partial t} w_{\vec{\jmath}} J_{\bar{n}}\left(\vec{\xi}_{\vec{\jmath}}\right)$$

$$\langle \Phi_l, F[u] \rangle = \langle \Phi_l, F[u] \rangle_{\Omega_{\bar{m}}} + \langle \Phi_l, F[u] \rangle_{\Omega_{\bar{n}}} = \mathbf{F}^{(\alpha)}_{\bar{m}|\vec{\imath}} + \mathbf{F}^{(\alpha)}_{\bar{n}|\vec{\jmath}}$$

Between the two $\mathbf{F}$ terms, boundary flux terms cancel out.

The continuity requirement means

$$\frac{\partial \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}}{\partial t} = \frac{\partial \mathbf{u}^{(\alpha)}_{\bar{n}|\vec{\jmath}}}{\partial t} = \frac{\partial \mathbf{u}^{(\alpha)}_{l}}{\partial t}$$

so

$$\frac{\partial \mathbf{u}^{(\alpha)}_{l}}{\partial t} = \frac{\mathbf{F}^{(\alpha)}_{\bar{m}|\vec{\imath}} + \mathbf{F}^{(\alpha)}_{\bar{n}|\vec{\jmath}}}{w_{\vec{\imath}} J_{\bar{m}}\left(\vec{\xi}_{\vec{\imath}}\right) + w_{\vec{\jmath}} J_{\bar{n}}\left(\vec{\xi}_{\vec{\jmath}}\right)}$$

This is the global approach to evolving the fields. However, it requires having global knowledge of the mesh, which requires large matrices to be passed on to the various processors. An equivalent but computationally less expensive method is the local approach, which temporarily discards continuity and sets

$$\frac{\partial \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}}{\partial t} = \frac{\mathbf{F}^{(\alpha)}_{\bar{m}|\vec{\imath}}}{w_{\vec{\imath}} J_{\bar{m}}}, \quad \frac{\partial \mathbf{u}^{(\alpha)}_{\bar{n}|\vec{\jmath}}}{\partial t} = \frac{\mathbf{F}^{(\alpha)}_{\bar{n}|\vec{\jmath}}}{w_{\vec{\jmath}} J_{\bar{n}}}$$

In this case we discard boundary terms within each $\mathbf{F}$ term. Using these separate and not necessarilly equal evolution terms, we calculate

$$\mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(t) \to \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(\star), \quad \mathbf{u}^{(\alpha)}_{\bar{n}|\vec{\jmath}}(t) \to \mathbf{u}^{(\alpha)}_{\bar{n}|\vec{\jmath}}(\star)$$

using whatever evolution method we want. Each of these calculations can be done on a separate processor, which only requires information about that individual element. The HOMER method for this process is `homer_integrate_elem`, after the information about each element is sent from the root processor with `homer_init`.

Matrices involved in the calculation may be dense and relatively small (compared to the global calculation). These routines can then be performed with the help of packages such as LAPACK or ScaLAPACK.

Since $\mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(\star) \neq \mathbf{u}^{(\alpha)}_{\bar{n}|\vec{\jmath}}(\star)$, we must now perform a weighted sum to preserve continuity. To do this we set

$$\mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(t+\Delta t) = \mathbf{u}^{(\alpha)}_{\bar{n}|\vec{\imath}}(t+\Delta t) = \frac{w_{\vec{\imath}} J_{\bar{m}}\left(\vec{\xi}_{\vec{\imath}}\right) \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(\star) + w_{\vec{\jmath}} J_{\bar{n}}\left(\vec{\xi}_{\vec{\jmath}}\right) \mathbf{u}^{(\alpha)}_{\bar{n}|\vec{\jmath}}(\star)}{w_{\vec{\imath}} J_{\bar{m}}\left(\vec{\xi}_{\vec{\imath}}\right) + w_{\vec{\jmath}} J_{\bar{n}}\left(\vec{\xi}_{\vec{\jmath}}\right)}$$

This is done once the processors return their data to the head processor, with subroutine `homer_integrate_root`.

Implicit versus explicit evolution

The evolution will be done either implicitly or explicitly. Generally the equation will be

$$\mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(\star) = \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(t) + \Delta \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}$$

$$\Delta \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}} = \frac{\partial \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}}{\partial t}\left[\mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(t)\right] \Delta t \cdot \theta + \frac{\partial \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}}{\partial t}\left[\mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(\star)\right] \Delta t \cdot (1-\theta)$$

This can be solved by a Newton's method, using

$$\Delta \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(0) = \frac{\partial \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}}{\partial t}\left[\mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(t)\right] \Delta t$$

$$\Delta \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(n+1) = \Delta \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(n) - \frac{1}{\Delta t \cdot (1-\theta)} \sum_{J} \left[\mathbf{J}\left[\Delta \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(n)\right]\right]^{-1}_{IJ} \frac{\partial \mathbf{u}^{(\beta)}_{\bar{m}|\vec{\jmath}}}{\partial t}\left[\Delta \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(n)\right]$$

where the Jacobian matrix is

$$\mathbf{J}_{IJ}\left[\Delta \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(n)\right] = \frac{1}{w_{\vec{\imath}} J_{\bar{m}}} \frac{\partial \mathbf{F}^{(\alpha)}_{\bar{m}|\vec{\imath}}}{\partial \mathbf{u}^{(\beta)}_{\bar{m}|\vec{\jmath}}}\left[\mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(t) + \Delta \mathbf{u}^{(\alpha)}_{\bar{m}|\vec{\imath}}(n)\right]$$

is a matrix indexed by the double indices $I = (\alpha, \vec{\imath})$ and $J = (\beta, \vec{\jmath})$. For a problem with, say, ten fields $\mathbf{u}$, three dimensions and a polynomial degree of 5, this matrix has dimensions $\left(10 \cdot 6^3\right) \times \left(10 \cdot 6^3\right) = 2160 \times 2160$, or on the order of 10^6 entries, about 1 MB of data. Generally derivative terms will require

$\mathbf{J}_{(\alpha,\vec{\imath})(\alpha,\vec{\jmath})} \neq 0$, and so $\mathbf{J}$ will at least have a block diagonal structure. This generates about $10 \cdot 6^6 \approx 10^5$ nonzero terms. Furthermore, if field α interacts with field β, then $\mathbf{J}_{(\alpha,\vec{\imath})(\beta,\vec{\imath})} \neq 0$. This generates about $6^3 \approx 10^2$ terms per each interaction, which are negligible in counting terms unless the number of interactions is on the order of 10^2 (all possible interactions), at which point total the number of nonzero terms reaches nearer to 10^6 in order. Therefore, for a small number of interactions between fields, $\mathbf{J}$ can be approximated as sparse, but for a large number of interactions, it is dense.

Therefore, we will require sparse matrices when there are a large number of fields or a high degree of interpolation, but a low number of interactions between the chemicals. Otherwise, we can represent $\mathbf{J}$ as a dense matrix.

θ is set through the`theta` variable in `equation`. When $\theta = 0$, the evolution is done through a simple forward Euler's method without any use of Newton's method or the Jacobian. **Currently, the code is only designed to handle $\theta = 0$.**

Strang Splitting

It may be desired to use operator splitting; that is, writing the evolution as

$$\frac{\partial u^{(\alpha)}}{\partial t} = F^{(\alpha)}\left[u\right] = F_1^{(\alpha)}\left[u\right] + F_2^{(\alpha)}\left[u\right]$$

For example, in our model, we can write $F_1^{(\alpha)}\left[u\right] = D\nabla^2 u^{(\alpha)}$ and $F_2^{(\alpha)}\left[u\right] = R^{(\alpha)}\left(u\right)$. Suppose that we represent an evolution step with

$$u^{(\alpha)}\left(t + \Delta t\right) = \exp\left[\Delta t \cdot F^{(\alpha)}\left[\cdot\right]\right] u^{(\alpha)}\left(t\right)$$

where exp is whatever method we use to evolve a step forward through Δt. In our implicit Euler integration, $\exp\left[\cdot\right]$ is given by solving

$$\exp\left[\Delta t \cdot F\left[\cdot\right]\right] u\left(t\right) = u\left(t\right) + F\left[u\left(t\right)\right] \Delta t \cdot \theta + F\left[\exp\left[F\left[\cdot\right]\right] u\left(t\right)\right] \Delta t \cdot \left(1 - \theta\right)$$

Strang splitting means we evolve as [1]

$$u^{(\alpha)}\left(t + \Delta t\right) = \exp\left[\frac{1}{2}\Delta t \cdot F_1^{(\alpha)}\left[\cdot\right]\right] \exp\left[\Delta t \cdot F_2^{(\alpha)}\left[\cdot\right]\right] \exp\left[\frac{1}{2}\Delta t \cdot F_1^{(\alpha)}\left[\cdot\right]\right] u^{(\alpha)}\left(t\right)$$

which sacrifices some accuracy but is computationally less expensive. The option to use Strang splitting can exercised through the `equation` type.

Currently, the code is NOT designed to handle Strang splitting.

0.13 Testing

The form of equations that will be used to test the parallel code is

$$\frac{\partial\left[A\right]}{\partial t} = D\nabla^2\left[A\right] - \vec{v} \cdot \vec{\nabla}\left[A\right] - k_1\left[A\right]^2 + 2k_2\left[A_2\right]$$

$$\frac{\partial\left[A_2\right]}{\partial t} = D\nabla^2\left[A_2\right] - \vec{v} \cdot \vec{\nabla}\left[A_2\right] + \frac{1}{2}k_1\left[A\right]^2 - k_2\left[A_2\right]$$

modeled after a monatomic to diatomic chemical reaction, $2A \leftrightharpoons A_2$, occuring with diffusion and convection. This model is tested in both one and two dimensions, and is used to determine convergence as well as scalability results.

One-dimensional convergence testing

The particular model was chosen:

$$\frac{\partial u\left(x,t\right)}{\partial t} = 0.1\frac{\partial^2 u\left(x,t\right)}{\partial x^2} - 1.0u\left(x,t\right), \quad x \in \left[0,1\right], \quad t \in \left[0,1\right]$$

$$u\left(x,0\right) = 1 + \cos\left(\pi x\right)$$

with theoretical solution

$$u\left(x,t\right) = e^{-1.0t}\left(1 + e^{-0.1\pi^2 t}\cos\left(\pi x\right)\right)$$

The code set to solve this problem with 5, 10, 20 and 40 processors in order to study the order of convergence. The results at $t = 1$, on the whole domain and on a small interval near the origin, are shown in Figure 1.

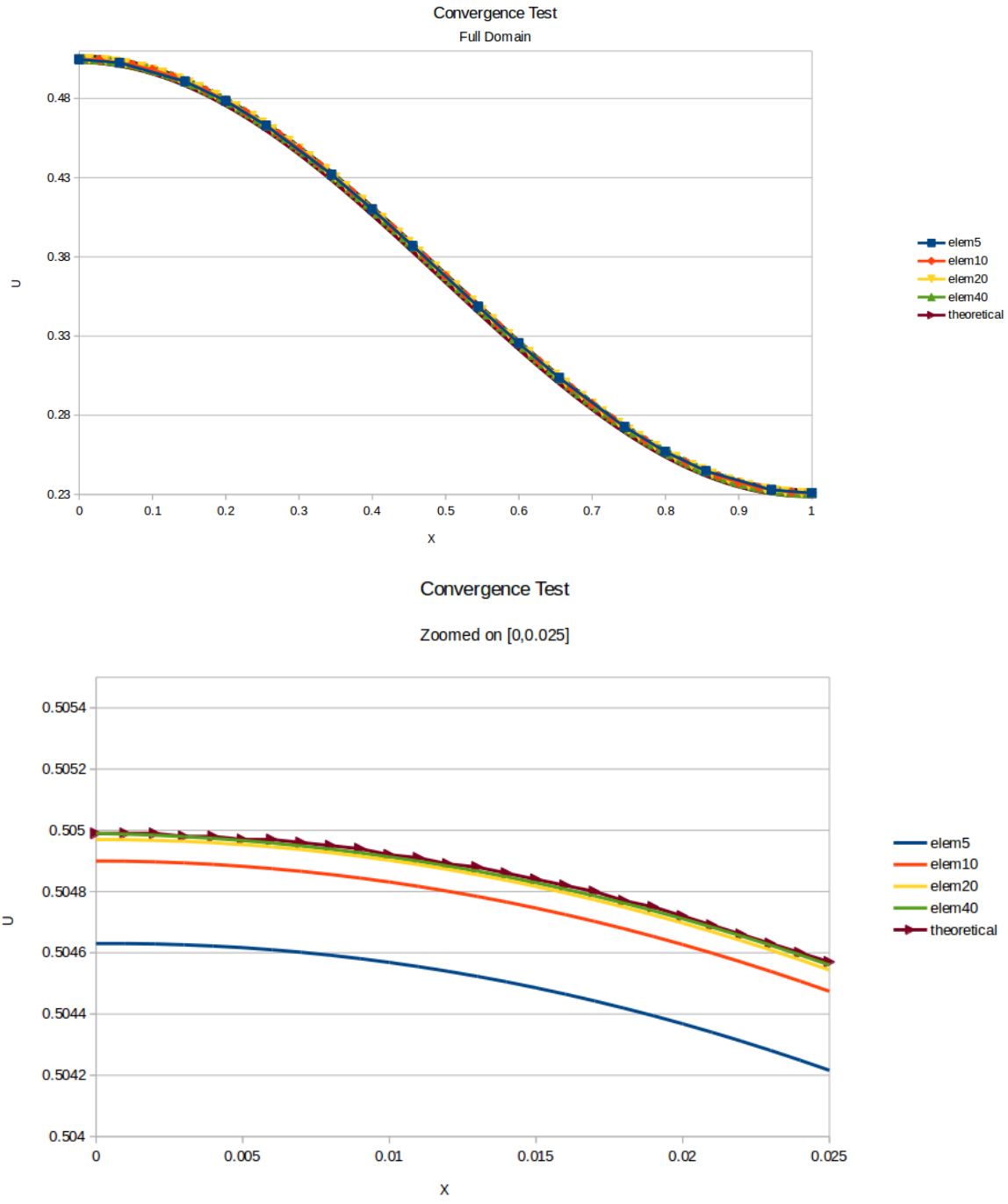


Figure 1: The graph of convergence for $0 < x < 1$ and $0 < x < 0.025$

The particular values at $x = 0$ for these tests can be numerically examined to calculate the order of convergence. It is seen clearly that the difference between each consecutive resolution decreases by one-fourth for every doubling in the number of elements. This indicates 2nd-order convergence, with an

error of $O\left(\Delta x^2\right)$.

Two-dimensional convection, diffusion and reaction testing

The equations (x) and (y) were chosen as the two-dimensional model, with $D = 0.00625$, $k_2 = 2.0$, $k_1 = 1.0$, and $\vec{v} = (0.5, 0)$. The initial conditions were chosen as

$$[A_2](x, y, 0) = \begin{cases} \frac{1}{4}(1 - \cos(4\pi x))(1 - \cos(4\pi y)) & 0 < x < \frac{1}{2},\ 0 < y < \frac{1}{2} \\ 0 & \text{otherwise} \end{cases}$$

$$[A](x, y, 0) = \begin{cases} \frac{1}{4}(1 - \cos(4\pi x))(1 - \cos(4\pi y)) & 0 < x < \frac{1}{2},\ \frac{1}{2} < y < 1 \\ 0 & \text{otherwise} \end{cases}$$

The code was run on a grid of $5 \times 5 = 20$ elements. The results are shown in Figure 2. As the above equations being tested are nonlinear, we do not have a theoretical solution to compare with. However, we can examine some quantitative and qualitative features of these results.

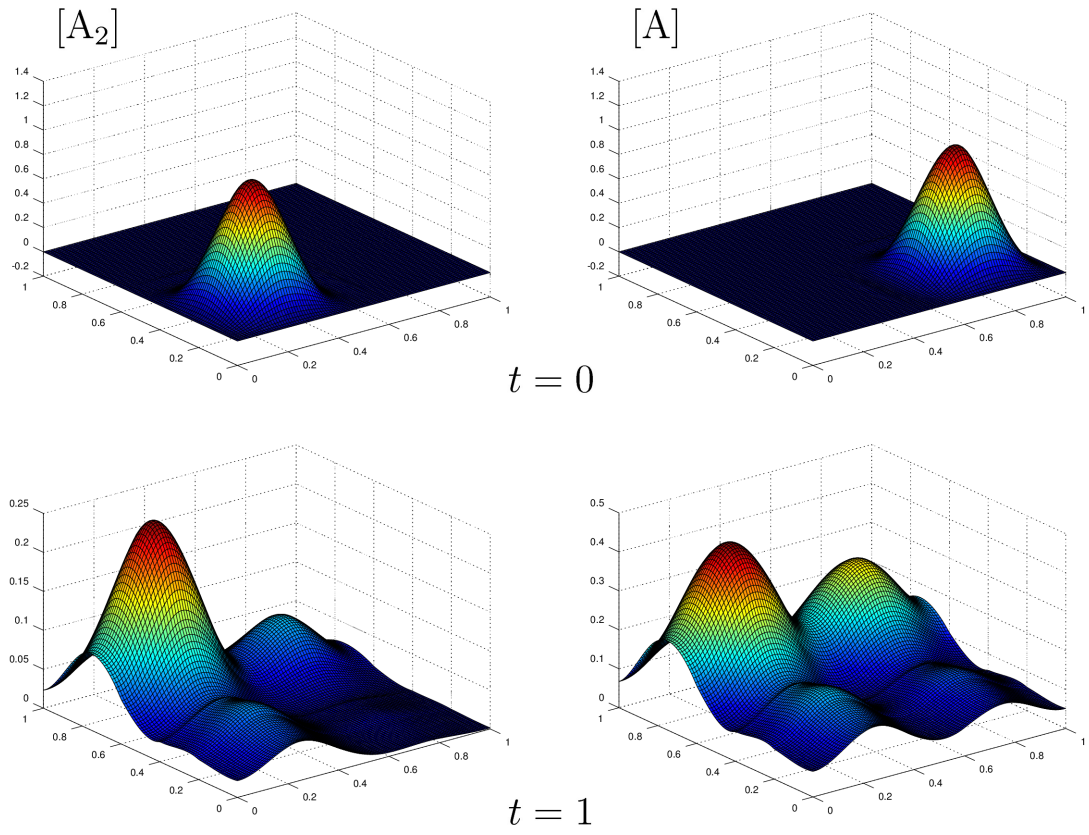


Figure 2: The results of testing in two dimensions, at $t = 0$ and $t = 1$. Graphs on the left are of $[A_2]$ and on the right are of [A].

It is clear that the peak of the initial distributions move in the direction of the convection current; in fact, they move from $x = 0.25$ at $t = 0$ to $x = 0.75$ at $t = 1$, which matches the theoretical $v_x = 0.5$. We see that the values of the function are being averaged out over the domain, as expected in diffusion, and we also note that each chemical is being produced in the domain of the other chemical due to the reactions. Particularly, we note that A is produced much more quickly than A_2, whichis in concordance with the fact that $k_2 > k_1$.

One odd behavior of this solution is the appearance of small peaks around $x = 0.25$ at $t = 1$, which seem to have "moved in" from the $x < 0$ region. This is likely due to the lack of boundary conditions on the problem.

The simulation was also run far beyond $t = 1$ to $t = 100$ (now with $\vec{v} = 0$) when the system has approached equilibrium. The equilibrium values are compared to the equilibrium constant

$$K \equiv \frac{2k_1}{k_2} = \frac{[A_2]}{[A]^2}$$

In our theoretical example, this equilibrium should be set to $K = 1.0$. The results from the code give an agreeing value of $K = 1.0$ at each point within an error of about 10^{-10} %.

Two-dimensional scalability testing

A similar example to the above was run with a grid of $32 \times 32 = 1024$ elements, run from $t = 0$ to $t = 0.01$ with a timestep of $\Delta t = 0.0001$. This code was run on Darter by dividing the elements between 2^n processors for $n = 1, 2, \ldots, 10$. The time it took to run each simulation is plotted in a log-log graph in Figure 3.

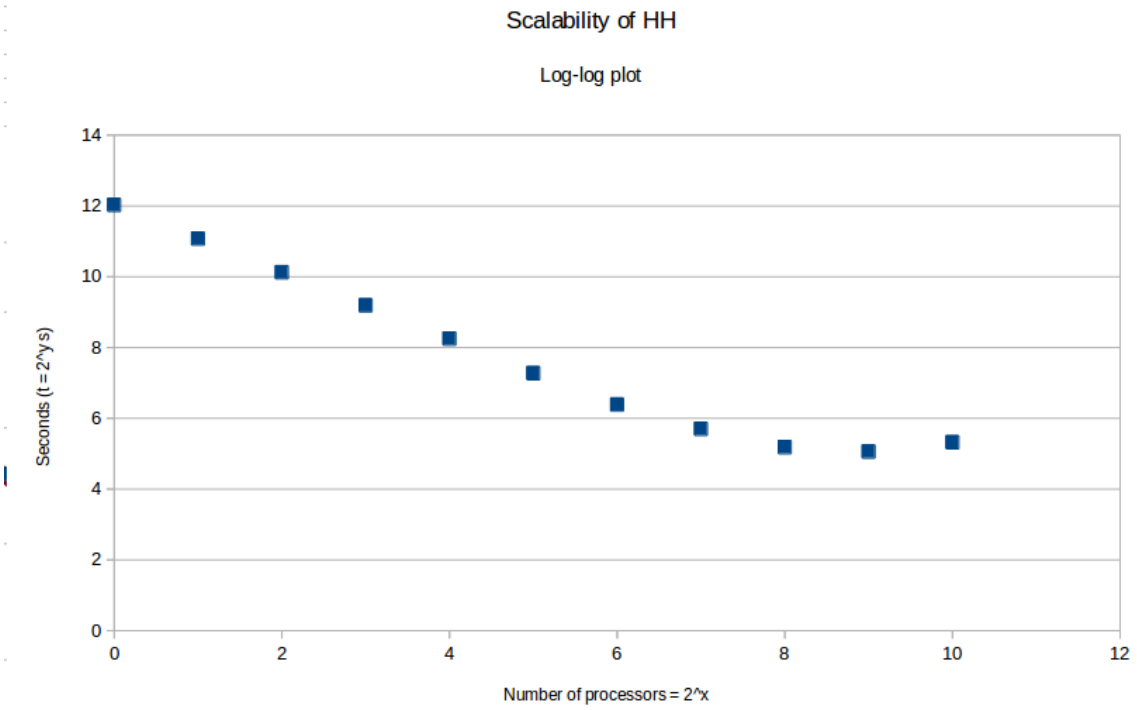


Figure 3: The scalability graph for 1024 elements. The x-axis gives the number of processors as 2^x; the y axis gives the run time as $t = 2^y$ s.

When the number of processors is less than the number of elements, the code displays high scalability, indicating a clear inverse relationship between time and processor number. However, this inverse relationship appears to break down at 256 processors, after which 512 processors has the smallest time while 1024 processors actually has a longer time.

This is likely due to the fact that the algorithm is not entirely parallel, due to the need for boundary communication. All boundary averages are calculated on a single root processor. In the future the code can be adapted to perform boundary sums only on the necessary elemental processors, which should improve the scalability for higher numbers of processors.

It should also be noted that implementing operator splitting should improve the overall efficiency of the code on all processors; the degree to which it improves remains to be seen.

0.14 Conclusions and Future Research

It is clear that the parallel code can successfully solve the above-given chemical transport problems, in both one and two dimensions. There are many more ways in which it can be tested, however. The code can be compared with the efficiency of Finite Volume codes in an equal-accuracy test, and it may be tested on a wider range of models (more chemicals, higher-order reactions, and higher dimensions). As the code is intended to solve general PDEs with the SEM, it will be necessary to also test the code on a wider variety of differential equations.

There are also ways in which the code can be further improved, which include: implementing implicit Euler methods and Strang splitting; paralellizing the boundary averages in the integration process; and allowing the user to define boundary conditions on the mesh and fields.

References

[1] Marlis Hochbruck and Alexander Ostermann. Time integration: splitting methods. Slideshow Presentation, 2005.

[2] Mark A. Taylor and AimÃ© Fournier. A compatible and conservative spectral element method on unstructured grids. *Journal of Computational Physics*, 229(17):5879 – 5895, 2010.