Pseudospectral-element modelling of elastic waves in Matlab®

Matt McDonald, Michael Lamoureux and Gary Margrave

ABSTRACT

Numerical modelling of elastic waves is an integral part of many procedures in seismic imaging. As such it is important to have a fast and efficient algorithm that can properly model the underlying physics of elastic waves propagating in the subsurface of the earth. Because the inherent layering of the subsurface, an appropriate numerical method must take into account the level of continuity present in the underlying assumptions of the physical model. Failure to do so can result in reflected waves with incorrect phase and amplitudes. A convenient place to explore the properties of numerical methods is in the Matlab® computing environment. In this paper we build a Pseudospectral-element method for the elastic wave equation in two spatial dimensions with second-order absorbing boundary conditions using the *sparse* data structure in Matlab® with explicit time-stepping.

MATLAB®

Matlab® is a contraction of “Matrix Laboratory". The fundamental data structure in Matlab is the Matrix. As such, it is very convenient for rapid prototyping of new algorithms and is much more readable than lower-level languages such as C or Fortan. There is however a downside to Matlab® ’s convenience. Because it is an *interpreted* language, meaning that each line of code is interpreted "just-in-time", or, line by line as the program runs, the number one complaint is that Matlab® is slow. It is possible to mitigate the slow-down inherent to Matlab® , however, by replacing the dreaded *nested-for-loop* with vectorization and by using the built-in pre-compiled functions already available.

For example, suppose that there’s no such thing as the FFT and we want to numerically compute the first 1001 Fourier cosine coefficients of the function $exp(-10x^2)$ over the interval $[-1, 1]$. This can be done using the Legendre-Gauss-Lobatto (LGL) nodes and weights (more on those later) as described in Code 1.

```
1 [x w] = Legendre_Gauss_Lobatto_nodes_and_weights(1000);
2
3 f = exp(-10*x*x);
4
5 for i = 1:101
6     for j=1:1001
7         a(i) = a(i) + cos((2*i-1)*pi*x(j)/2)*f(j)*w(j);
8     end
9 end
```

Code 1: Computing Fourier coefficients with a double-for loop.

The computation doesn’t really take that long, about 0.39 seconds on a really small computer with a 1.6 GHz Intel Atom processor to compute the double for-loop. However,

in Code 2 we perform the exact same computation with zero for-loops.

```
1 [x w] = LGLNodesAndWeights(1000);
2
3 [K,X] = meshgrid(1:1001,x);
4
5 F = exp(-10*X.*X);
6
7 a = w'*(F.*cos((2*K-1)*pi.*X/2));
```

Code 2: Vectorized computing of Fourier coefficients.

The last line of which takes about 0.29 seconds. Granted this is not a very dramatic example of the computational speedup capable with vectorization but, in some of the computations done later on, the speedup can be in terms of *hours*.

PSEUDOSPECTRAL METHODS IN MATLAB®

There's a fair bit of discrepancy in the use of the term *pseudospectral*, but they generally fall into the category of solutions to partial differential equations, where it is understood to mean solutions derived from assuming an eigenfunction expansion of the form

$$u(\mathbf{x}) = \sum_{n=0}^{\infty} a_n \phi_n(\mathbf{x}). \tag{1}$$

For time-dependent problems the coefficients a_n are generally assumed to be functions of t. When the functions $\phi_n(\mathbf{x})$ in 1 form an orthogonal basis the expansion is known as a *generalized-Fourier-series*. Perhaps the most well-known form of the pseudospectral method in geophysical wave propagation stems from the choice of the the standard Fourier basis for the ϕ_n, and, in practice, this is probably the best place to start when seeking pseudospectral solutions due to the availability of the fast-Fourier-transform for computing numerical derivatives and convolutional sums, but is not the approach taken here. Another family of methods comes from choosing the ϕ_n's to be from a basis of orthogonal polynomials such as *Chebyshev* or *Legendre* polynomials. These are defined as the eigenfunctions of singular Sturm-Liouville differential equations but may be derived by many methods. All of the methods mentioned so-far are termed *modal* methods where the unknowns are the coefficients of the expansion and, thus, require transformations to and from sampled values of the functions we are interested in approximating and the appropriate coefficients.

The method used here is a so-called *nodal* method based on interpolation formulas that make use of the Lagrange polynomials

$$l_j(x) = \prod_{i \neq j} \frac{x - x_i}{x_j - x_i}$$

for a set of nodes $\{x_i\}_{i=0}^{N}$. In these methods the unknowns are the actual sampled values of the function and so no transformation is needed. The choice of nodes from which to sample

our functions is important however. Choosing the nodes to be equally spaced can result in Runge's phenomenon and may prevent the method from converging. Choosing the nodes to be the zeros of $(1-x^2)P_n'$, where P_n is the n^{th} Chebyshev or Legendre polynomial, fixes this problem. These points are known as the Chebyshev or Legendre Gauss-Lobatto nodes, respectively.

The $N+1$ Chebyshev-Gauss-Lobatto (CGL) nodes can be computed analytically in Matlab® using Code 3.

```
function x = CGLNodes(N)
x = -cos(pi*(0:N)/N);
```

Code 3: Matlab function for computing the Chebyshev-Gauss-Lobatto nodes.

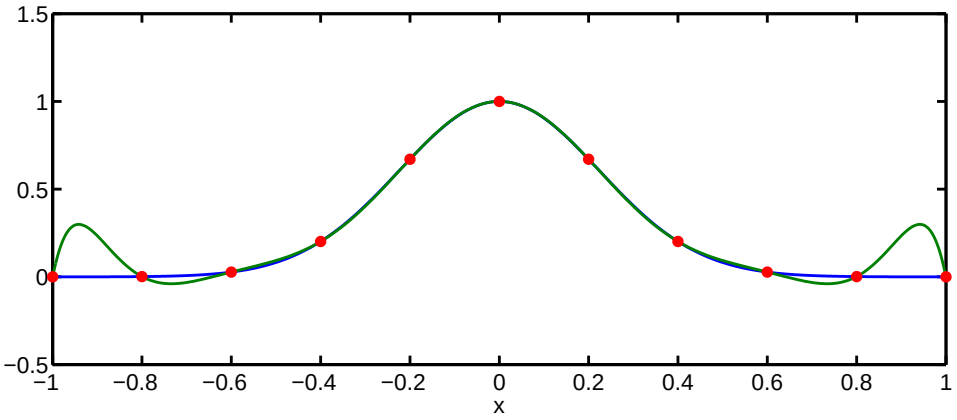
The Legendre-Gauss-Lobatto (LGL) nodes, on the other hand, have no closed form and so must be computed by a numerical root-finding method. Code 4 computes the LGL nodes by a Newton method using the asymptotic relation in (2) as a starting point. At the same time, it computes the Legendre polynomials using the recursion relation

$$\begin{cases} P_0(x) = 1, \\ P_1(x) = x, \\ (n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x). \end{cases}$$

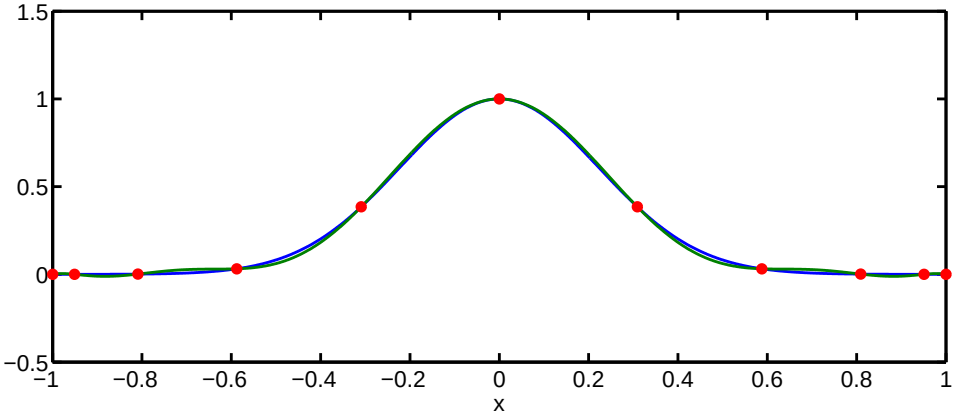
```
function x = LGLNodes(N)

x = -cos(((0:N/2)+.25)*pi/N - 3./(8*N*pi*((0:N/2)+.25)));
xold = 0;

V = zeros(N+1,length(x));

while max(abs(x-xold))>eps
     V(1,:) = 1;
     V(2,:) = x;

     for n = 3:N+1
          V(n,:) =((2*n-3)*x.*V(n-1,:)-(n-2)*V(n-2,:))/(n-1);
     end
     xold = x;
     x=xold-( x.*V(N+1,:)-V(N,:) )./( (N+1)*V(N+1,:) );
end

x = [x,-x(ceil(N/2):-1:1)];
```

Code 4: Matlab function for computing the Legendre-Gauss-Lobatto nodes.

Figure 1 shows the result of interpolating a compactly-supported function using different choices of nodes and the presence of Runge's phenomenon.



(a) Runge's Phenomenon.



(b) No Runge's Phenomenon.

FIG. 1: Interpolation of the function $exp(-10x^2)$ using equispaced nodes (a) vs. Chebyshev-Gauss-Lobatto nodes (b).

Associated with a set of Gauss-Lobatto nodes $\{x_i\}_{i=0}^{N}$ is a set of numerical integration weights $\{w_i\}_{i=0}^{N}$. The weights are computed analytically in both the Chebyshev and Legendre cases. For the CGL nodes the weights are such that

$$\int_{-1}^{1} \frac{f(x)}{\sqrt{1-x^2}} dx = \frac{\pi}{N} \left\{ \sum_{i=1}^{N-1} f(x_i) w_i + \frac{f(x_1)w_1 + f(x_N)w_N}{2} \right\}$$

is exact when f is a polynomial of degree less then or equal to $2N-1$. For a general function f the weights would then be $(\sqrt{1-x^2})\pi/N$, where $\sqrt{1-x^2}$ is the weight function associated with the Sturm-Liouville equation for the Chebyshev functions. The integration weights are computed in Code 5.

For the Legendre polynomials the Sturm-Liouville weight function is equal to 1 and so the quadrature formula is

$$\int_{-1}^{1} f(x) dx = \sum_{i=0}^{N} f(x_i) w_i.$$

```
function x = CGLNodesAndWeights(N)
x = CGLNodes;
w = sqrt(1-x.^2)*pi/N;
```

Code 5: Matlab function for computing the Chebyshev-Gauss-Lobatto nodes and weights.

This is, again, exact for polynomials of degree less than or equal to $2N-1$. The integration weights are computed from the values of the N^{th} Legendre polynomial evaluated at the LGL nodes in Code 6.

```
function [x w] = LGLNodesAndWeights(N)

x = LGLNodes(N);

V = zeros(N+1,length(x));
V(1,:) = 1;
V(2,:) = x;

for n = 3:N+1
  V(n,:) =((2*n-3)*x.*V(n-1,:)-(n-2)*V(n-2,:))/(n-1);
end

w = 2./(N*(N+1)*V(N+1,:).^2);
```

Code 6: Matlab function for computing the Legendre-Gauss-Lobatto nodes and weights.

Now that we have defined a numerical integration technique it is only natural to focus now on numerical differentiation. Consider the interpolary expansion

$$u(x) = \sum_{n=0}^{\infty} a_n \phi_n(x) \quad \forall x_i, \quad i = 0, ..., N,$$

We can write this in matrix form

$$\begin{pmatrix} u(x_0) \\ u(x_1) \\ \vdots \\ u(x_N) \end{pmatrix} = \begin{pmatrix} \phi_0(x_0) & \cdots & \phi_N(x_0) \\ \phi_0(x_1) & \cdots & \phi_N(x_1) \\ \vdots & \ddots & \vdots \\ \phi_0(x_N) & \cdots & \phi_N(x_N) \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_N \end{pmatrix}$$

so then

$$\begin{pmatrix} a_0 \\ \vdots \\ a_N \end{pmatrix} = \begin{pmatrix} \phi_0(x_0) & \cdots & \phi_N(x_0) \\ \phi_0(x_1) & \cdots & \phi_N(x_1) \\ \vdots & \ddots & \vdots \\ \phi_0(x_N) & \cdots & \phi_N(x_N) \end{pmatrix}^{-1} \begin{pmatrix} u(x_0) \\ \vdots \\ u(x_N) \end{pmatrix}$$

Then matrix equation for the nodal values of the derivative is then

$$\begin{pmatrix} u'(x_0) \\ \vdots \\ u'(x_N) \end{pmatrix} = \begin{pmatrix} \phi'_0(x_0) & \cdots & \phi'_N(x_0) \\ \phi'_0(x_1) & \cdots & \phi'_N(x_1) \\ \vdots & \ddots & \vdots \\ \phi'_0(x_N) & \cdots & \phi'_N(x_N) \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_N \end{pmatrix}$$

$$= \begin{pmatrix} \phi'_0(x_0) & \cdots & \phi'_N(x_0) \\ \phi'_0(x_1) & \cdots & \phi'_N(x_1) \\ \vdots & \ddots & \vdots \\ \phi'_0(x_N) & \cdots & \phi'_N(x_N) \end{pmatrix} \begin{pmatrix} \phi_0(x_0) & \cdots & \phi_N(x_0) \\ \phi_0(x_1) & \cdots & \phi_N(x_1) \\ \vdots & \ddots & \vdots \\ \phi_0(x_N) & \cdots & \phi_N(x_N) \end{pmatrix}^{-1} \begin{pmatrix} u(x_0) \\ \vdots \\ u(x_N) \end{pmatrix}$$

Choosing a basis $\{\phi_n\}_{n=0}^N$ (such as the Legendre or Chebyshev polynomials) and set of points $\{x_n\}_{n=0}^N$ (such as the LGL or CGL nodes) fully defines the pseudospectral differentiation matrix

$$D = \begin{pmatrix} \phi'_0(x_0) & \phi'_1(x_0) & \cdots & \phi'_N(x_0) \\ \phi'_0(x_1) & \phi'_1(x_1) & \cdots & \phi'_N(x_1) \\ \vdots & \ddots & \ddots & \vdots \\ \phi'_0(x_N) & \phi'_1(x_N) & \cdots & \phi'_N(x_N) \end{pmatrix} \begin{pmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_N(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_N(x_1) \\ \vdots & \ddots & \ddots & \vdots \\ \phi_0(x_N) & \phi_1(x_N) & \cdots & \phi_N(x_N) \end{pmatrix}^{-1}$$

We compute this matrix for the Legendre polynomials using two functions in Matlab® in Code 7 and 8.

```
function [V Vx] = legVVx(x)

if size(x,2) == 1; x = x.'; end

N = length(x);
V = zeros(N);
Vx = zeros(N);
Vxx = zeros(N);

V(:,1) = 1;
V(:,2) = x;
for n = 3:N
    V(:,n) =((2*n-3)*x'.*V(:,n-1)-(n-2)*V(:,n-2))/(n-1);
end

Vx(:,1) = 0;
Vx(:,2) = 1;

for n = 2:N-1;
    Vx(:,n+1) = (2*n-1)*V(:,n) + Vx(:,n-1);
end
```

Code 7: Compute the matrices of nodal values of the Legendre polynomials and their first derivatives.

```
function Dx = legDMat(x)

if size(x,2) == 1; x = x.'; end

[V Vx] = legVVx(x);
Dx = Vx/V;
```

Code 8: Compute the pseudospectral differentiation matrix.

As a final note it should be pointed out that while it is possible to define a higher-order differentiation matrices $D^{(n)}$ that compute the n^{th} nodal derivative by taking powers of the first derivative matrix, this is generally a bad idea for a large number of nodes. Instead the matrices should be computed either using the same procedure we used to derive the first-order differentiation matrix or via special recursion formulas specific to the choice of basis functions (6).

The 2D versions of the pseudospectral differentiation matrices and integration weights are obtained by defining their 1D counter-parts along each dimension and then taking Kronecker tensor products. This can be done in Matlab® in several ways. For the integration weights, assume we have two column vectors `w_x` and `w_z` containing the integration weights in the x and z directions, respectively, associated with the vectors `x = ... LGLNodes(Nx)` and `z = LGLNodes(Nz)` of dimension `Nx` and `Nz`. Then the 2D integration weights can be computed as `W = w_z*w_x.'` giving an `Nz`-by-`Nx` matrix containing the 2D integration weights. Then if `U` is the column-major-storage of the matrix of nodal values of a function $u(x, z)$ we can perform integration by taking the dot product with the column-major-storage-vector version of `W`, computed in Matlab® as `W(:)`.

Computing the differentiation matrices is a little different. Suppose again that we are working with the vector `U` and wish to compute the matrices `Dx` and `Dz` that discretely compute ∂_x and ∂_z, respectively. We first compute the 1D differentiation matrices `Dx1D` and `Dz1d`. The matrix `Dx` can be computed as `Dx=kron(Dx1d,eye(Nz)` and `Dz=kron(eye(Nx),Dz1d)`.

WEAK FORM OF THE ELASTIC WAVE EQUATION

To define our method we first need to derive the *weak* form of the elastic wave equation. Consider the *strong* formulation of the elastic wave equation for an arbitrary isotropic heterogeneous medium $\Omega \in \mathbb{R}^d, d = 1, 2, 3$, with boundary $\partial\Omega = \Gamma$.

$$\begin{cases} \rho\ddot{u}_i = \partial_j \sigma_{ij}(\mathbf{u}) + f_i, & \mathbf{x} \in \Omega, t \geq 0 \\ \mathbf{u}(\mathbf{x}, t = 0) = \mathbf{u}_0(\mathbf{x}), & \mathbf{x} \in \Omega \\ \dot{\mathbf{u}}(\mathbf{x}, t = 0) = \mathbf{u}_1(\mathbf{x}), & \mathbf{x} \in \Omega \end{cases} \tag{2}$$

The stresses are

$$\sigma_{ij}(\mathbf{u}) = \lambda(\nabla \cdot \mathbf{u})\delta_{ij} + 2\mu\varepsilon_{ij}(\mathbf{u})$$

where ∂_j denotes differentiation with respect to the j^{th} element x_j and

$$\varepsilon_{ij}(\mathbf{u}) = \frac{1}{2}\left(\partial_i u_j + \partial_j u_i\right).$$

Summation over repeated indices, as per Einstein notation, is assumed unless otherwise noted. The parameters λ, μ and ρ are the elastic constants of the medium and all may be bounded, spatially dependent, functions. $f_i(\mathbf{x}, t)$ is the i^{th} component of the body force applied to the medium.

We obtain the *weak* form by the *Galerkin*. Multiplying both sides of 2 by an arbitrary test function $v \equiv v(\mathbf{x})$ and integrating over the entire space yields

$$\int_\Omega \rho \ddot{u}_i v d\Omega = \int_\Omega \partial_j \sigma_{ij}(\mathbf{u}) v d\Omega + \int_\Omega f_i v d\Omega. \tag{3}$$

Expanding the first term on the right hand side and applying Green's theorem gives us the relationship

$$\int_\Omega \partial_j \sigma_{ij}(\mathbf{u}) v d\Omega = \oint_\Gamma \sigma_{ij}(\mathbf{u}) v \hat{n}_j d\Gamma - \int_\Omega \sigma_{ij}(\mathbf{u}) \partial_j v d\Omega.$$

where $\hat{n}_j$ denotes the j^{th} component of the outward-pointing normal vector. Substituting this into 3 yields the i^{th} component of displacement of the weak form of 2

$$\int_\Omega \rho \ddot{u}_i v d\Omega + \int_\Omega \sigma_{ij}(\mathbf{u}) \partial_j v d\Omega = \int_\Omega f_i v d\Omega + \oint_\Gamma \sigma_{ij}(\mathbf{u}) v \hat{n}_j d\Gamma. \tag{4}$$

This is the form for which we will derive the numerical method. The boundary terms and is what allows us to *talk* to the boundary; incorporating absorbing and free-surface boundary conditions. The most appropriate absorbing boundary conditions for our purposes are those for which the time and space derivatives appear independently. We omit the details of the derivations here for the sake of brevity as even the two-dimensional case involves 16 boundary integrals, 8 of which account for the absorbing boundary conditions (all of which factor into a single operator). We note, however, that there are several different choices available and refer the reader to (5), (4) and (3) for the construction and implementation of several higher-order methods that fit naturally into variational schemes.

PSEUDOSPECTRAL-ELEMENTS

As with any method involving domain-decomposition we decompose Ω into a union of smaller subdomains,

$$\Omega = \bigcup_{k=1}^{M} \Omega_k.$$

On each element we then define a tensor-product grid of LGL nodes and make the definition that the edges of each element share the associated nodes, as seen in figure 3.

For the elastic wave equation in 2D we are solving for two components of displacement $u_1(x, z, t)$ and $u_2(x, z, t)$ which are the horizontal and vertical displacement of the medium. In domain decomposition methods we then define these to be combinations of the contributions over each subdomain.

$$u_i(x, z, t) = \sum_{k=1}^{M} u_i^k(x, z, t)$$

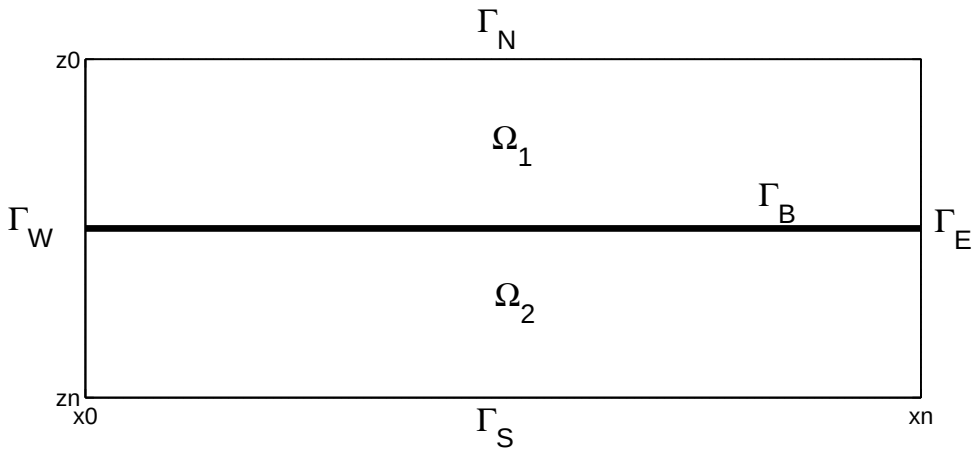


FIG. 2: Two subdomains and their shared boundaries over the entire domain.

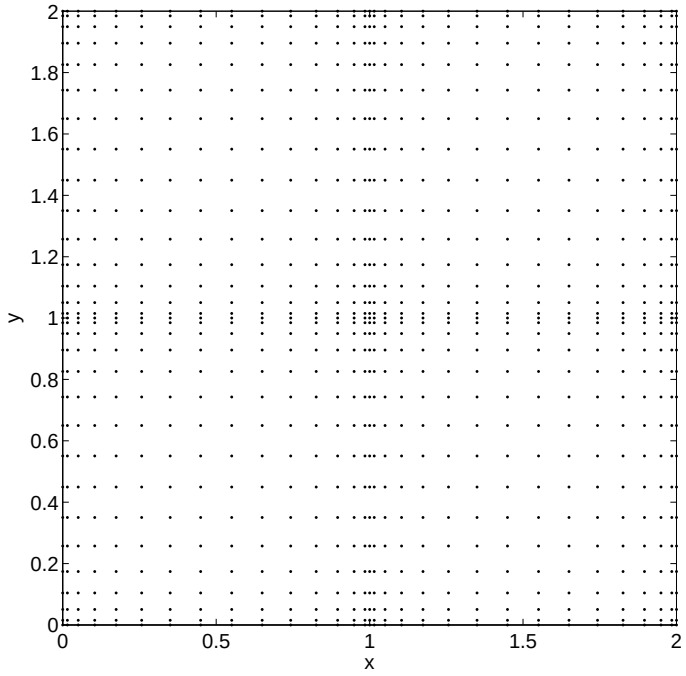


FIG. 3: 2D Legendre-Gauss-Lobatto SEM nodes distributed over 4 subdomains.

and split up the weak form 4 over the subdomains. This results in

$$\int_{\Omega_k} \rho \ddot{u}_i v d\Omega_k + \int_{\Omega_k} \sigma_{ij}(\mathbf{u}) \partial_j v d\Omega_k = \int_{\Omega_k} f_i v d\Omega_k + \oint_{\Gamma_k} \sigma_{ij}(\mathbf{u}) v \hat{n}_j d\Gamma_k. \tag{5}$$

where Γ_k is the boundary of the k^{th} subdomain Ω_k. To enforce proper interface conditions we require the displacements to be continuous across the boundaries of each element and that the stresses across the interface vanish, known as a free-surface condition. Thus, the

boundary integral vanishes everywhere except at the boundaries where we enforce the absorbing boundary conditions. The continuity of displacement is represented by defining the basis functions associated with the edge nodes to be the piece-wise continuous functions constructed by equating the basis functions from each element. Several examples of these functions are seen in figure 4.

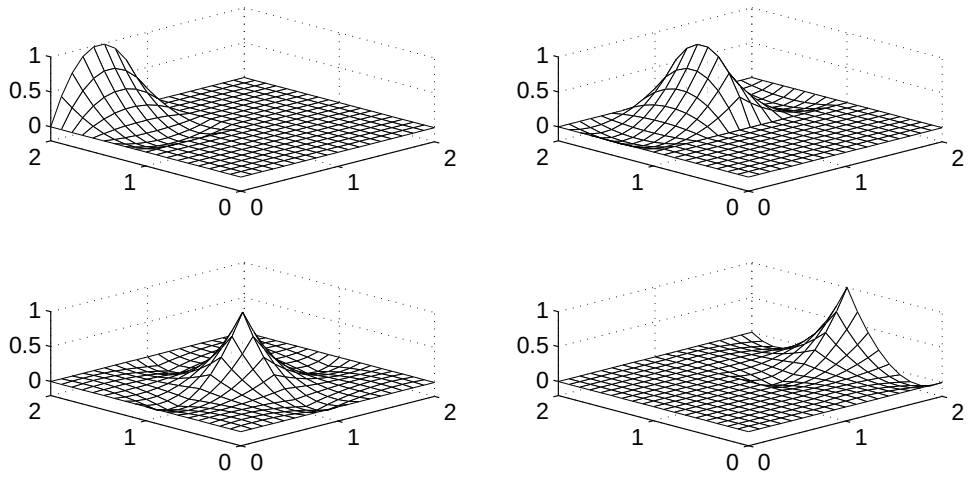


FIG. 4: 2D SEM basis functions defined on 4 elements.

Interior to each domain, equation 5 discretized using pseduospectral differentiation matrices and integration weights by writing

$$u_i^k(x,t) = \sum_{i=1}^{n} u_i^k(\mathbf{x}_i,t) l_i(\mathbf{x},z).$$

Substituting this into 5 and choosing the functions v to be equal to $l_j(x,z)$ produces the system of equations for the vector of nodal values $\mathbf{u}_i^k(t)$ in the k^{th} element

$$M^k \ddot{\mathbf{u}}_i^k(t) + A_i^k \dot{\mathbf{u}}_i^k(t) + \sum_j K_{ij}^k \mathbf{u}_j^k(t) = M^k \mathbf{f}_i^k(t)$$

The element mass matrix M^k is a diagonal matrix with the integration weights along the main diagonal and the structure of the element damping matrices A_i^k depends on the absorbing boundary conditions but is generally diagonal and only non-zero along the main diagonal at the positions corresponding to the indices of nodes along outer boundaries. The element stiffness matrix K_{ij}^k are the discrete representation of the integro-differential operator in the i^{th} equation 5 acting on the nodal values of the j^{th} component of the displacement. The Global mass, damping and stiffness matrices are assembled by transforming their respective indices into the global indices and summing over the connected nodes. This is done using the so called *connectivity* matrix wherein the i^{th} column contains the global node numbers of the i^{th} element. This is easier to portray in an example. In figure 5 we show 4 elements defined on $[-1,1] \times [-1,1]$ numbered column-major. If we number the

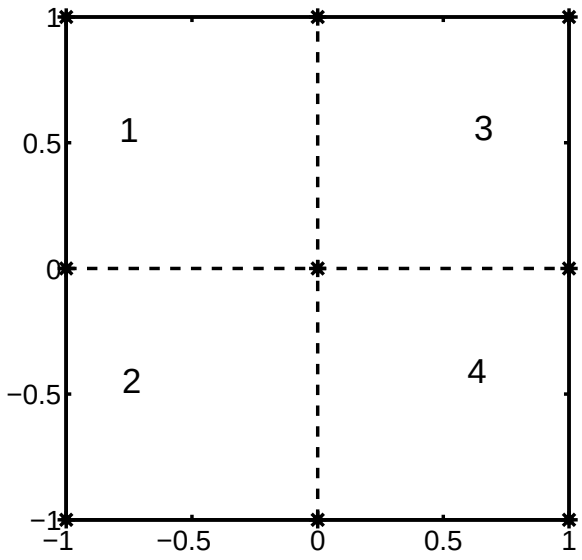


FIG. 5: 4 elements with 4 nodes each for a total of 9 global nodes.

global nodes column-major as well, the connectivity matrix is defined as

$$C = \begin{pmatrix} 1 & 2 & 4 & 5 \\ 2 & 3 & 5 & 6 \\ 4 & 5 & 7 & 8 \\ 5 & 6 & 8 & 9 \end{pmatrix}$$

Using the connectivity matrix we could assemble the global stiffness matrix via Code 9. This, however, requires 3 for-loops and, for large numbers of nodes and/or elements, takes extremely long. It also assembles a fully-populated matrix that would be a drastic waste of memory.

```
1 for i=1:Nx*Nz
2     Dxi = Dx*2/dXk(i); Dzi = Dz*2/dZk(i); Mi = M*(dXk(i)*dZk(i))/4;
3     Ki = Dxi.'*Vp(i)*Vp(i)*Mi*Dxi + Dzi.'*Vp(i)*Vp(i)*Mi*Dzi;
4     for j=1:Np*Np
5         for k=1:Np*Np
6             K(C(k,i),C(j,i)) = K(C(k,i),C(j,i)) + Ki(j,k);
7         end
8     end
9 end
```

Code 9: Assemble the global stiffness matrix *very* slowly.

A much better way of assembly is to define 3 vectors containing the row and column indices and element entries of the matrix and then call `sparse` to define the global matrix. This is done for a single block of the larger block stiffness matrix in code 10. In practice it is again, much faster to assemble all the blocks at the same time, but the code is much longer and less readable.

```
for i=1:Nx*Nz
    Dxi = Dx*2/dXk(i); Dzi = Dz*2/dZk(i);
    Mi = M*(dXk(i)*dZk(i))/4;
    Ki = (Dxi.'*Vp(i)*Vp(i)*Mi*Dxi + Dzi.'*Vp(i)*Vp(i)*Mi*Dzi);

    for j=1:Np*Np
        idx = ((j-1)*Np^2+1:j*Np^2) + (i-1)*Np*Np*Np*Np;
        I(idx) = C(j,i)*ones(Np^2,1); % row positions
        J(idx) = C(:,i);% col positions
        X(idx) = Ki(:,j); % entries
    end
end

dim = (Nz*(Np-1)+1)*(Nx*(Np-1)+1);
K = sparse(I,J,X,dim,dim);
```

Code 10: Assemble the global stiffness matrix using `sparse`.

The vectors `dXk` and `dZk` are the width and height of the elements and are used to map the differentiation matrices and integration weights from local coordinates to global coordinates.

Once the global system is assembled it can be written in block matrix form

$$\begin{pmatrix} M & 0 \\ 0 & M \end{pmatrix} \frac{\partial^2}{\partial t^2} \begin{pmatrix} \mathbf{u}_1^k \\ \mathbf{u}_2^k \end{pmatrix}(t) + \begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix} \frac{\partial}{\partial t} \begin{pmatrix} \mathbf{u}_1^k \\ \mathbf{u}_2^k \end{pmatrix}(t)$$
$$+ \begin{pmatrix} K_{11} & K_{12} \\ K_{21} & A_{22} \end{pmatrix} \begin{pmatrix} \mathbf{u}_1^k \\ \mathbf{u}_2^k \end{pmatrix}(t) = \begin{pmatrix} \mathbf{f}_1^k \\ \mathbf{f}_2^k \end{pmatrix}(t)$$

which we will write simply as

$$M\ddot{\mathbf{u}} + A\dot{\mathbf{u}} + K\mathbf{u} = \mathbf{F}. \tag{6}$$

The sparsity pattern of stiffness matrix K is shown in figure 6.

TIME-STEPPING

To deal with the time-dependent system a numerical procedure must be implemented that is capable of handling the first and second order derivatives in equation (6). A second order in time scheme can be constructed by replacing the derivatives with second-order central difference approximations

$$\ddot{\mathbf{u}}(t_j) = \frac{\mathbf{u}(t_{j+1}) - 2\mathbf{u}(t_j) + \mathbf{u}(t_{j-1})}{\Delta t^2} + \mathcal{O}(\Delta t^2),$$

$$\dot{\mathbf{u}}(t_j) = \frac{\mathbf{u}(t_{j+1}) - \mathbf{u}(t_{j-1})}{2\Delta t} + \mathcal{O}(\Delta t^2).$$

After dropping the error term, (6) then becomes

$$M\left(\frac{\mathbf{u}(t_{j+1}) - 2\mathbf{u}(t_j) + \mathbf{u}(t_{j-1})}{\Delta t^2}\right) + A\left(\frac{\mathbf{u}(t_{j+1}) - \mathbf{u}(t_{j-1})}{2\Delta t}\right) + K\mathbf{u}(t_j) = M\mathbf{F}(t_j)$$

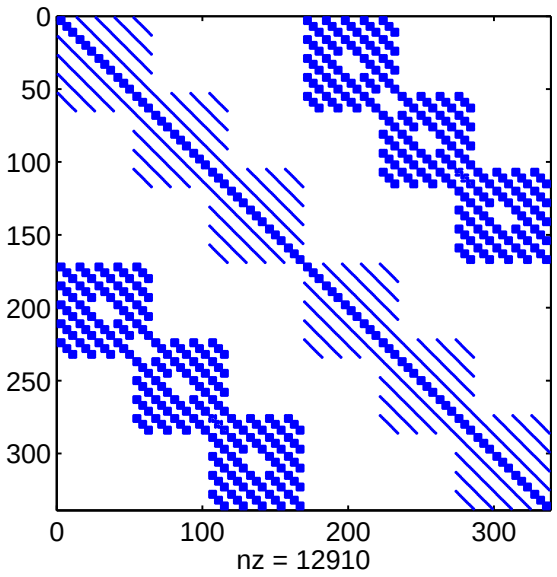


FIG. 6: Sparsity patterns of the stiffness matrix.

or in terms of the t_j's

$$\left[M + \frac{\Delta t}{2}A\right]\mathbf{u}(t_{j+1}) + \left[\Delta t^2 K - 2M\right]\mathbf{u}(t_j) + \left[M - \frac{\Delta t}{2}A\right]\mathbf{u}(t_{j-1}) = \Delta t^2 M\mathbf{F}(t_j)$$

The matrix $\left[M + \frac{\Delta t}{2}A\right]$ must now be inverted in order to step forward in time. Since both the matrices M and A are diagonal this is trivial. The method is implemented in Code 11 where the solution is returned sampled at SR ms.

```
function [U t] = CFD_SR(M,A,K,U1,U2,tn,dt,fx,ft,SR)

Np = length(fx);

P = (M+.5*dt*A)\(2*M-dt*dt*K);
Q = (M+.5*dt*A)\(.5*dt*A-M);
Fx = dt*dt*((M+.5*dt*A)\M)*fx;

numskip = ceil(SR/dt);
numkept = ceil(tn/(numskip*dt));

t = 0:dt:(numskip*numkept*dt);

Ft = ft(t);

U = zeros(Np,numkept+1);

for k=1:numkept
    for j=1:numskip
        U3 = P*U2 + Q*U1 + Ft(j+(k-1)*numskip)*Fx;
        U1 = U2;
        U2 = U3;
```

```
    end
    U(:,k+1) = U3;
end

t = 0:(numskip*dt):(numskip*numkept*dt);
```

Code 11: Matlab function for time-stepping mixed order ODE systems by central finite-differences.

Another way to time-step the problem would be to re-write it as a first order system by making the substitution $\mathbf{v} = \dot{\mathbf{u}}$. Then (6) can be rewritten as

$$\begin{bmatrix} \dot{\mathbf{u}} \\ \dot{\mathbf{v}} \end{bmatrix} + \begin{bmatrix} 0 & I \\ M^{-1}K & M^{-1}A \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{F} \end{bmatrix}$$

and solved by an appropriate method for first-order systems. In Code 12 we define a function that time steps this equation by the 4^{th}-order low-storage explicit-Runge-Kutta scheme (1) and returns the solution sampled at SRms.

```

function [U t] = LSERK_SR(K,U0,tn,dt,fx,ft,SR)

Np = length(U0);
numskip = ceil(SR/dt);
numkept = ceil(tn/(numskip*dt));

t = 0:dt:(numskip*numkept*dt);

U = zeros(Np,numkept+1);
U(:,1) = U0;

Pk = zeros(Np,1);
Kk = zeros(Np,1);

[a b c] = LSERKcoefs;

for i=1:numkept
    for j=1:numskip
        for k=1:5
            Kk = a(k)*Kk + dt*( K*Pk + ...
                fx*ft(t(j+(i-1)*numskip)+c(k)*dt) );
            Pk = Pk + b(k)*Kk;
        end
    end
    U(:,i+1) = Pk;
end

t = 0:(numskip*dt):(numskip*numkept*dt);
```

Code 12: Matlab function for 1st order ode systems by LSERK.

The LSERK method is desirable over standard Runge-Kutta methods in that it only requires a single extra level of storage, while a standard RK45 scheme requires an extra 5. The trade-off, however, is an extra level of computation.

EXAMPLE

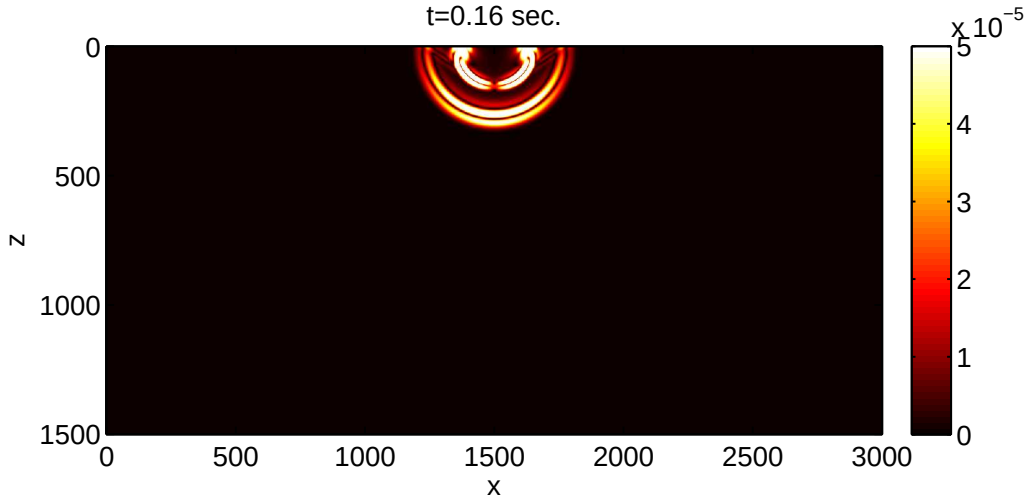
To test the method we consider a forcing term of the form

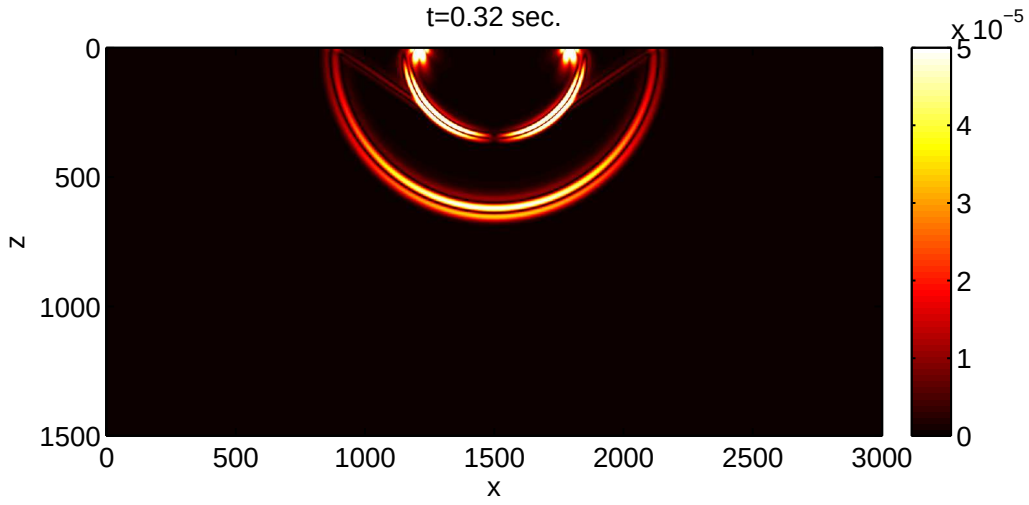
$$\mathbf{F}(\mathbf{x},t) = \delta(\mathbf{x}-\mathbf{x}_0)f(t)$$

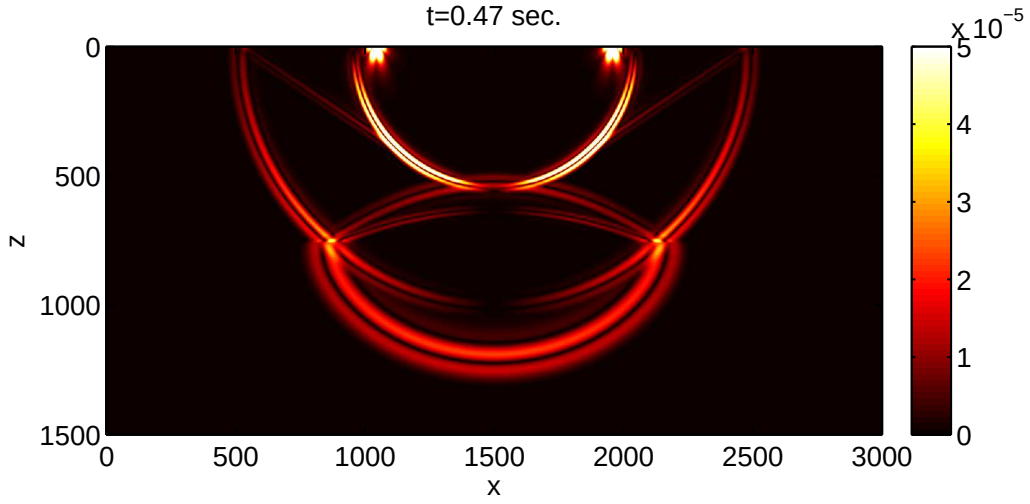
where the time-component is a Ricker wavelet

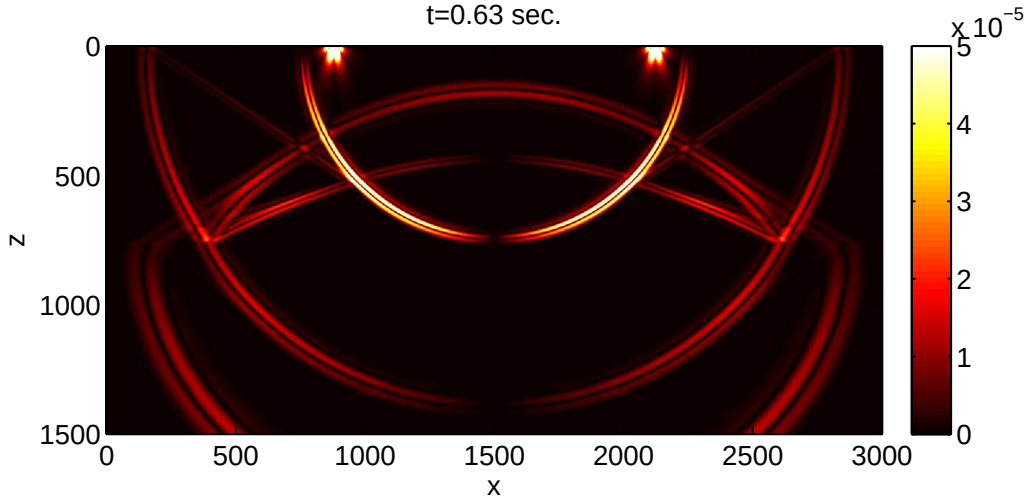
$$f(t) = \frac{2}{\sqrt{3\sigma}\pi^{\frac{1}{4}}}\left(1-\frac{t^2}{\sigma^2}\right)e^{\frac{-t^2}{2\sigma^2}}.$$

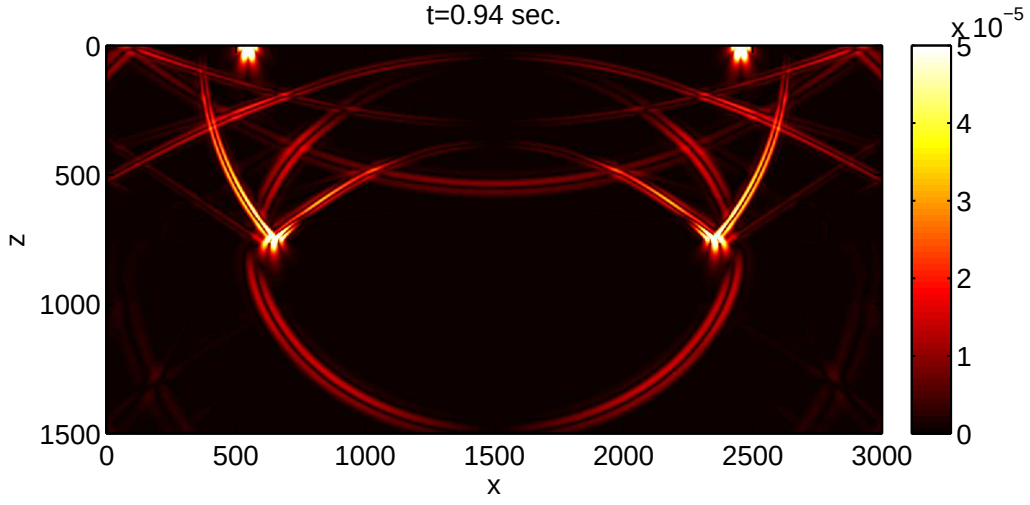
The model is a simple 2-layer medium with V_p,V_s, and ρ constant in each layer. The propagating wavefield is shown in the last 3 figures. At first we can see the pressure, shear and surface waves originating from the source term. Then, as the wave propagates through the interface each wave is converted into more pressure and shear waves until, finally, as the waves reach the side and bottom boundaries, they are absorbed.

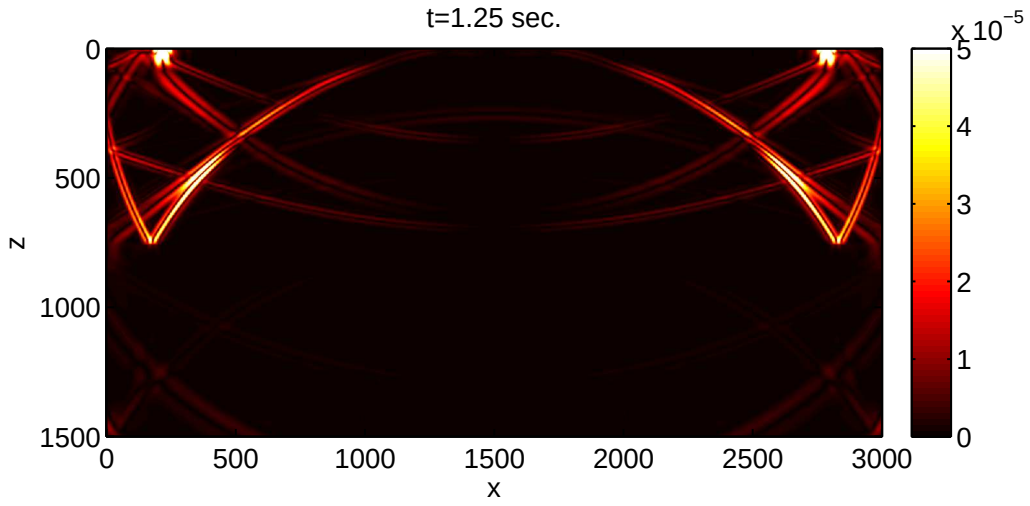





t=0.47 sec.
x 10^-5
z
x
0
500
1000
1500
2000
2500
3000
1
2
3
4
5


t=0.63 sec.
x 10^-5
z
x
0
500
1000
1500
2000
2500
3000
1
2
3
4
5


t=0.94 sec.
x 10^-5
z
x
0
500
1000
1500
2000
2500
3000
1
2
3
4
5



ACKNOWLEDGMENTS

We gratefully acknowledge the continued support of *mprime* through the POTSI research project and its industrial collaborators, the support of NSERC through the CREWES consortium and its industrial sponsors, and support of the Pacific Institute for the Mathematical Sciences.

REFERENCES

[1] David and Ketcheson, *Runge kutta methods with minimum storage implementations*, Journal of Computational Physics **229** (2010), no. 5, 1763 – 1773.

[2] Seymour V. Parter, *On the Legendre Gauss Lobatto points and weights*, J. Sci. Comput. **14** (1999), 347–355.

[3] A. Quarteroni, A. Tagliani, and E. Zampieri, *Generalized galerkin approximations of elastic waves with absorbing boundary conditions*, Computer Methods in Applied Mechanics and Engineering **163** (1998), no. 1-4, 323 – 341.

[4] J. Sochacki, *Absorbing boundary conditions for the elastic wave equations*, Applied Mathematics and Computation **28** (1988), no. 1, 1 – 14.

[5] R. Stacey, *Improved transparent boundary formations for the elastic-wave equation*, Bulletin of the Seismological Society of America **78** (1988), 2089–2097.

[6] J. A. Weideman and S. C. Reddy, *A matlab differentiation matrix suite*, ACM Trans. Math. Softw. **26** (2000), 465–519.