AOSS 555 Final Project: Modeling Seismic Wave Propagation with Radial Basis Functions

Trever Hines

April 28, 2015

Introduction

In recent years seismologists have become interested in using observable seismic waveforms to image the elastic properties of the Earth's interior. We use the term 'seismic waveform' to refer the the displacement history that can be observed on a seismogram over the course of hundreds of seconds following an earthquake. As with any inverse problem, imaging the elastic properties of the earth using seismic waveforms requires one to be able to solve the forward problem, that is, to compute the displacements that would be predicted for a given realization of the Earth's interior. Researchers have used various numerical methods to solve this forward problem including finite difference methods, finite element methods, and spectral element methods (see Fichtner (2011) for a review of numerical methods in seismology). In this paper We will explore the potential for using radial basis functions (RBFs) to solve for the time dependent displacements throughout the Earth following an earthquake.

One reason why RBFs are appealing is because they offer freedom in choosing the collocation points. In global seismology, one would ideally have a high clustering of collocation points at the core-mantle boundary, which is an interfaces where the elastic properties suddenly and drastically change. Additionally, the crust and the core have relatively low velocities and would require a higher density of nodes in accordance with the CFL criteria. RBFs seem to be appropriate for handling such heterogeneity in the Earth.

Another great advantage to RBFs is that they can be used to solve a problem on an arbitrarily complicated domain. Although the domain in global seismology is typically taken to be a sphere or a disk, more complicated geometries can arise in regional scale seismology problems where topography may need to be considered.

Problem Formulation

We can apply Newton's second law of motion to an infinitesimal parcel within a continuum to obtain displacements, $\boldsymbol{u}$, at any point $\boldsymbol{x}$ as

$$\rho \frac{\partial^2 u_i}{\partial t^2} = \frac{\partial \tau_{ij}}{\partial x_j} + f_i, \tag{1}$$

where $\boldsymbol{\tau}$ and $\boldsymbol{f}$ are the stress tensor and body force per unit volume acting on point $\boldsymbol{x}$, respectively, and ρ is the density at point $\boldsymbol{x}$. Note that We have adopted Einstein notation where summations are implied by repeated indices. We can express eq. (1) in terms of $\boldsymbol{u}$ by using the definition of infinitesimal strain,

$$\varepsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_j}{\partial x_i} + \frac{\partial u_i}{\partial x_j}\right), \tag{2}$$

and Hook's law in an isotropic medium,

$$\tau_{ij} = \lambda \delta_{ij} \varepsilon_{kk} + 2\mu \varepsilon_{ij}, \tag{3}$$

where λ and μ are Lamé parameters which we generally assume to be heterogeneous in the Earth. The differential equation being solved is then

$$\begin{aligned} \rho \frac{\partial^2 u_i}{\partial t^2} &= \frac{\partial}{\partial x_j}\left(\lambda \delta_{ij} \frac{\partial u_k}{\partial x_k} + \mu\left(\frac{\partial u_j}{\partial x_i} + \frac{\partial u_i}{\partial x_j}\right)\right) + f_i \\ &= \frac{\partial}{\partial x_i}\left(\lambda \frac{\partial u_k}{\partial x_k}\right) + \frac{\partial}{\partial x_j}\left(\mu\left(\frac{\partial u_j}{\partial x_i} + \frac{\partial u_i}{\partial x_j}\right)\right) + f_i \\ &= \frac{\partial \lambda}{\partial x_i}\frac{\partial u_k}{\partial x_k} + \lambda \frac{\partial^2 u_k}{\partial x_i x_k} + \frac{\partial \mu}{\partial x_j}\frac{\partial u_j}{\partial x_i} + \frac{\partial \mu}{\partial x_j}\frac{\partial u_i}{\partial x_j} + \mu \frac{\partial^2 u_j}{\partial x_j \partial x_i} + \mu \frac{\partial^2 u_i}{\partial x_j \partial x_j} + f_i. \end{aligned} \tag{4}$$

We impose free-surface boundary conditions, which means that

$$\boldsymbol{\tau} \cdot \hat{\boldsymbol{n}} = 0 \tag{5}$$

for all $\boldsymbol{x}$ on the surface of the Earth, which have surface normal vectors $\hat{\boldsymbol{n}}$. We can write the boundary conditions on $\boldsymbol{u}$ more explicitly as

$$\hat{n}_i \lambda \frac{\partial u_k}{\partial x_k} + \hat{n}_j \mu \left(\frac{\partial u_j}{\partial x_i} + \frac{\partial u_i}{\partial x_j}\right) = 0. \tag{6}$$

Although the equation that we are trying to solve looks daunting, if we consider a homogeneous medium, neglect body forces, and assume that displacement are in one direction and propagate in one direction, then we can see that eq. (4) reduces to the one dimensional wave equation. In particular, if displacements are in the direction of wave propagation then eq. (4) becomes

$$\frac{\partial^2 u}{\partial t^2} = \frac{\lambda + 2\mu}{\rho}\frac{\partial^2 u}{\partial x^2}, \tag{7}$$

where waves propagate at the P wave velocity, $\sqrt{(\lambda + 2\mu)/\rho}$. If displacements are perpendicular to the direction of wave propagation then eq. (4) reduces to

$$\frac{\partial^2 u}{\partial t^2} = \frac{\mu}{\rho}\frac{\partial^2 u}{\partial x^2}, \tag{8}$$

where waves propagate at the S wave velocity, $\sqrt{\mu/\rho}$.

When solving eq. (4), we assume the Earth is initially stationary and that all displacements are caused by an earthquake. Earthquakes are a sudden dislocation in the displacement field along a fault plane and so it seems reasonable that earthquakes should be imposed in this model by setting appropriate displacement boundary conditions on a prescribed fault plane. However, imposing a dislocation in the displacement field would undoubtedly cause numerical troubles. Fortunately, we can instead represent an earthquake as a collection of slightly offset point force couples, $\boldsymbol{F}^{(i,j)}$ (Aki & Richards, 2002). All nine unit force couples needed to represent a dislocation in three dimensional space are shown in figure 1. Approximating a dislocation as point force couples is reasonable if one is concerned with displacements far away from the source, i.e. seismic waves on a global scale. The finite length of a fault rupture must be considered if one is interested in near field displacements resulting from an earthquake. Even so, seismologists routinely represent earthquakes in terms of point force couples using what is known as a moment tensor, $\boldsymbol{M}$. The components of the moment tensor are

$$M_{ij} = |\boldsymbol{F}^{(i,j)}|d, \tag{9}$$

where d is the offset between the force couples, which would ideally be infinitesimally small. To give the moment tensor more tangible meaning, we can consider an isotropic compression source, such as a subsurface explosion, where the force couples needed to describe the source are $\boldsymbol{F}^{(\mathbf{1,1})}$, $\boldsymbol{F}^{(\mathbf{2,2})}$, and $\boldsymbol{F}^{(\mathbf{3,3})}$, where each couple has the same magnitude, M_o/d. The moment tensor that describes the source is then

$$\boldsymbol{M} = M_o\boldsymbol{I}. \tag{10}$$

For our numerical simulation, we imposed an earthquake with moment tensor

$$\boldsymbol{M} = \begin{vmatrix} 0 & 10^{20} & 0 \\ 10^{20} & 0 & 0 \\ 0 & 0 & 0 \end{vmatrix} Nm, \tag{11}$$

which is equivalent to a Mw7.3 earthquake on a fault that is aligned with either the first or second coordinate axis.

We can represent the body force density for each force couple (without implied summations) as

$$\boldsymbol{f}^{(i,j)} = \frac{M_{ij}\hat{\boldsymbol{e}}_i}{d}\left(\delta\left(\boldsymbol{x} - \frac{d\hat{\boldsymbol{e}}_j}{2}\right) - \delta\left(\boldsymbol{x} + \frac{d\hat{\boldsymbol{e}}_j}{2}\right)\right), \tag{12}$$

where $\hat{\mathbf{e}}_i$ denotes a unit basis vector in direction i and δ is a three dimensional delta function, which can be taken to be a three dimensional Gaussian function with an infinitely small width. We can then represent the forcing term in 4 as

$$\boldsymbol{f} = \sum_{ij}\frac{M_{ij}\hat{\boldsymbol{e}}_i}{d}\left(\delta\left(\boldsymbol{x} - \frac{d\hat{\boldsymbol{e}}_j}{2}\right) - \delta\left(\boldsymbol{x} + \frac{d\hat{\boldsymbol{e}}_j}{2}\right)\right). \tag{13}$$

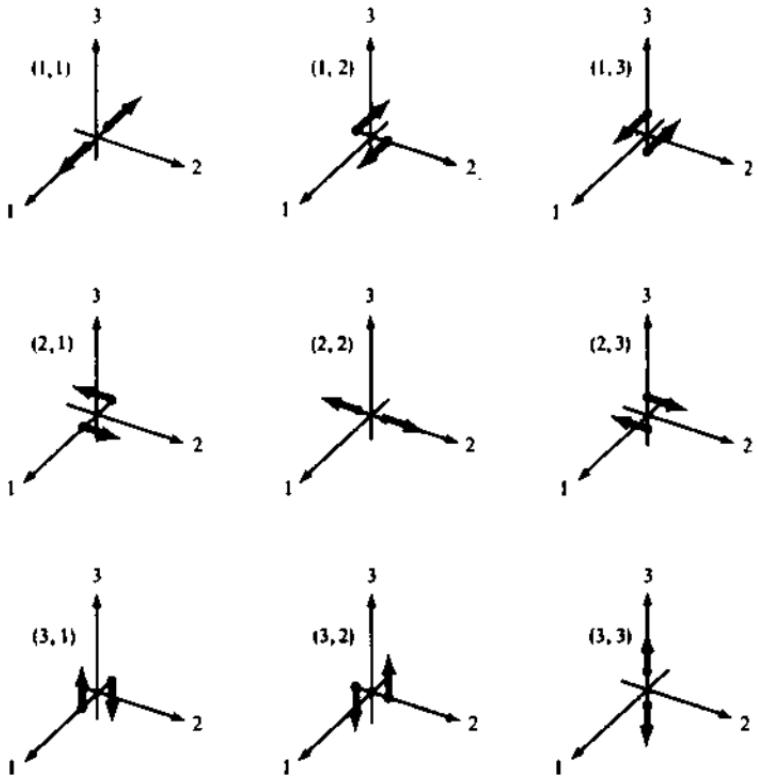

Figure 1: Force couples, $\boldsymbol{F}^{(i,j)}$, that can be used to represent any type of dislocation on a fault (taken from Aki & Richards (2002))

Although we circumvented to troubles of imposing a dislocation in the displacement field, we now face the difficulty of numerically representing a delta function. Indeed, the accuracy of our solution will come down to our ability to accurately represent a point force, which requires a very high density of collocation points. Using a desktop computer with 16 GB of RAM, we are only able to accurately represent a delta function as a 100 km wide Gaussian function, which is certainly not infinitesimally small compared to the Earth but it is the best we can do.

For the solutions presented in this paper, we assume that all energy is instantaneously released during an earthquake and $\boldsymbol{f}$ has no time dependence. This assumption may not be reasonable because large earthquakes could last for several minutes, which is on the timescale of seismic wave propagation.

Spectral Discretization

We will assume, for computational tractability but without loss of generality, that the domain is a two dimensional cross-section of the Earth and displacements are only in the plane of

that cross-section. After expanding the summation notation, eq. (4) can then be written as

$$\begin{vmatrix} \partial^2 u_1/\partial t^2 \\ \\ \partial^2 u_2/\partial t^2 \end{vmatrix} = \begin{vmatrix} \partial_1\left((\lambda+2\mu)\partial_1 u_1\right) + \partial_2\left(\mu\partial_2 u_1\right) + \partial_1\left(\lambda\partial_2 u_2\right) + \partial_2\left(\mu\partial_1 u_2\right) \\ \\ \partial_1\left(\mu\partial_2 u_1\right) + \partial_2\left(\lambda\partial_1 u_1\right) + \partial_2\left((\lambda+2\mu)\partial_2 u_2\right) + \partial_1\left(\mu\partial_1 u_2\right) \end{vmatrix} + \begin{vmatrix} f_1 \\ \\ f_2 \end{vmatrix}, \tag{14}$$

where we use ∂_i to denote a derivative with respect to x_i. Likewise, the boundary conditions can be written as

$$\begin{vmatrix} 0 \\ \\ 0 \end{vmatrix} = \begin{vmatrix} \hat{n}_1(\lambda+2\mu)\partial_1 u_1 + \hat{n}_2\mu\partial_2 u_1 + \hat{n}_1\lambda\partial_2 u_2 + \hat{n}_2\mu\partial_1 u_2 \\ \\ \hat{n}_1\mu\partial_2 u_1 + \hat{n}_2\lambda\partial_1 u_1 + \hat{n}_2(\lambda+2\mu)\partial_2 u_2 + \hat{n}_1\mu\partial_1 u_2 \end{vmatrix}. \tag{15}$$

We chose to solve eq. (4) in the spatial dimension using multiquadratic radial basis functions, which are defined as

$$\phi(\boldsymbol{x};\boldsymbol{x_0}) = \sqrt{1+(\epsilon|\boldsymbol{x}-\boldsymbol{x}_0|)^2}. \tag{16}$$

In the above equation, ϵ is the shape parameter and $\boldsymbol{x}_0$ is the center of the RBF. We chose to use multiquadratic radial basis functions because interpolating a function using $\phi(\boldsymbol{x};\boldsymbol{x_0})$ has the desirable trait of being equivalent to piece-wise linear interpolation for $\epsilon \to \infty$ and polynomial interpolation for $\epsilon \to 0$ at least for one-dimensional problems (Driscoll & Fornberg, 2002). This suggests that a wide range of values for ϵ should produce reasonably accurate results because either end-member is a perfectly valid means of interpolation. This is in contrast to using RBFs that decay to zero as $|\boldsymbol{x}-\boldsymbol{x_o}| \to \infty$, where the interpolant for $\epsilon \to \infty$ is essentially useless. We follow the common practice of making ϵ for each RBF proportional to the distance to the nearest RBF center. We did not rigorously search for an optimal proportionality constant; however, we note that we tend to get unstable or obviously inaccurate solutions for proportionality constants greater than 2 and less than 0.4. In particular, small values of epsilon tend to produce instabilities at sharp material interface or at the surface. large values of epsilon create large scale, unrealistic undulations. For the solution presented in this paper, we used a proportionality constant of 1, that is, we set ϵ equal to the distance to the nearest neighbor.

We approximate the solution for displacements as

$$u_1 \approx \sum_{j=1}^{N} \alpha_j(t)\phi_j(\boldsymbol{x};\boldsymbol{x_j}) \tag{17}$$

and

$$u_2 \approx \sum_{j=1}^{N} \beta_j(t)\phi_j(\boldsymbol{x};\boldsymbol{x_j}), \tag{18}$$

which can also be written as

$$\begin{vmatrix} u_1 \\ u_2 \end{vmatrix} \approx \boldsymbol{G}\begin{vmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\beta} \end{vmatrix} \tag{19}$$

where $\boldsymbol{G}$ is given by

$$\boldsymbol{G} = \begin{vmatrix} \boldsymbol{\phi} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{\phi} \end{vmatrix} \tag{20}$$

and we are no longer explicitly writing the time and space dependence. We can then express acceleration in terms of the spectral coefficients, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, as

$$\begin{vmatrix} \partial^2 u_1/\partial t^2 \\ \partial^2 u_2/\partial t^2 \end{vmatrix} = \boldsymbol{H} \begin{vmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\beta} \end{vmatrix} + \begin{vmatrix} f_1 \\ f_2 \end{vmatrix}, \tag{21}$$

where

$$\boldsymbol{H} = \begin{vmatrix} \partial_1\left((\lambda+2\mu)\partial_1\boldsymbol{\phi}\right) + \partial_2\left(\mu\partial_2\boldsymbol{\phi}\right) & \partial_1\left(\lambda\partial_2\boldsymbol{\phi}\right) + \partial_2\left(\mu\partial_1\boldsymbol{\phi}\right) \\ \partial_1\left(\mu\partial_2\boldsymbol{\phi}\right) + \partial_2\left(\lambda\partial_1\boldsymbol{\phi}\right) & \partial_2\left((\lambda+2\mu)\partial_2\boldsymbol{\phi}\right) + \partial_1\left(\mu\partial_1\boldsymbol{\phi}\right) \end{vmatrix}. \tag{22}$$

The boundary conditions can also be expressed in terms of the spectral coefficients as

$$\begin{vmatrix} 0 \\ 0 \end{vmatrix} = \boldsymbol{B} \begin{vmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\beta} \end{vmatrix}, \tag{23}$$

where

$$\boldsymbol{B} = \begin{vmatrix} \hat{n}_1(\lambda+2\mu)\partial_1\boldsymbol{\phi} + \hat{n}_2\mu\partial_2\boldsymbol{\phi} & \hat{n}_1\lambda\partial_2\boldsymbol{\phi} + \hat{n}_2\mu\partial_1\boldsymbol{\phi} \\ \hat{n}_1\mu\partial_2\boldsymbol{\phi} + \hat{n}_2\lambda\partial_1\boldsymbol{\phi} & \hat{n}_2(\lambda+2\mu)\partial_2\boldsymbol{\phi} + \hat{n}_1\mu\partial_1\boldsymbol{\phi} \end{vmatrix}. \tag{24}$$

At this point, all the information in $\boldsymbol{G}$, $\boldsymbol{H}$, $\boldsymbol{B}$, and $\boldsymbol{f}$ are given by the problem description or are user defined variables. Finding the spectral coefficients for $\boldsymbol{u}$ is then straightforward. We perform the time integration with a leapfrog method and our procedure for solving eq. (4) is as follows:

1. Form the matrix $\boldsymbol{J}$, which consists of $\boldsymbol{B}$ evaluated at the boundary collocation points and $\boldsymbol{G}$ evaluated at the interior collocation points. We use the collocation points as the RBF centers.
2. Form the vector $\boldsymbol{r}^0$, which consists of zeros for the boundary collocation points and $\boldsymbol{u}^0$ at the interior collocation points
3. Solve $\boldsymbol{r}^0 = \boldsymbol{J}[\boldsymbol{\alpha}^0\ \boldsymbol{\beta}^0]^T$ for $\boldsymbol{\alpha}^0$ and $\boldsymbol{\beta}^0$
4. evaluate the initial acceleration, $\boldsymbol{a}^0$, using eq. (21)
5. begin time stepping
   (a) evaluate $\boldsymbol{u}^t = \boldsymbol{u}^{t-1} + \boldsymbol{v}^{t-1}\Delta t + \boldsymbol{a}^{t-1}\Delta t^2$
   (b) Form $\boldsymbol{r}^t$
   (c) Solve $\boldsymbol{r}^t = \boldsymbol{J}[\boldsymbol{\alpha}^t\ \boldsymbol{\beta}^t]^T$ for $\boldsymbol{\alpha}^t$ and $\boldsymbol{\beta}^t$
   (d) evaluate $\boldsymbol{a}^t$ from eq. (21)
   (e) evaluate $\boldsymbol{v}^t = \boldsymbol{v}^{t-1} + 1/2(\boldsymbol{a}^{t-1} + \boldsymbol{a}^t)\Delta t$

There are, however, a few additional details that need to be addressed. First, we must deal with the material properties, μ, λ, and ρ. We use the material properties from the Preliminary Reference Earth Model (PREM) (Dziewonski & Anderson, 1981). PREM was inferred using observed P and S wave travel times in addition to other geophysical data. We then have a means of validating our numerical solution because the travel times predicted

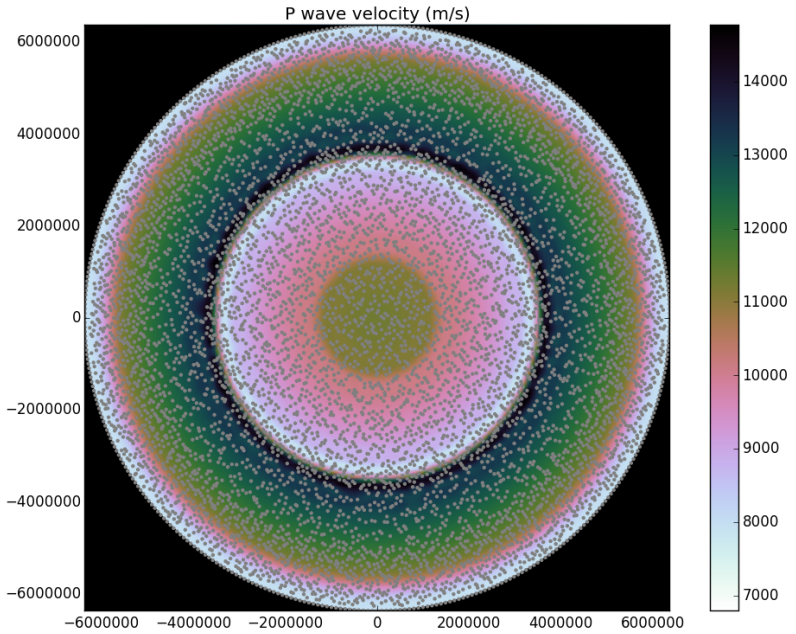

Figure 2: P wave velocities ($\sqrt{(\lambda + 2\mu)/\rho}$) from PREM. Gray dots indicate collocation points used in solving eq. (4)

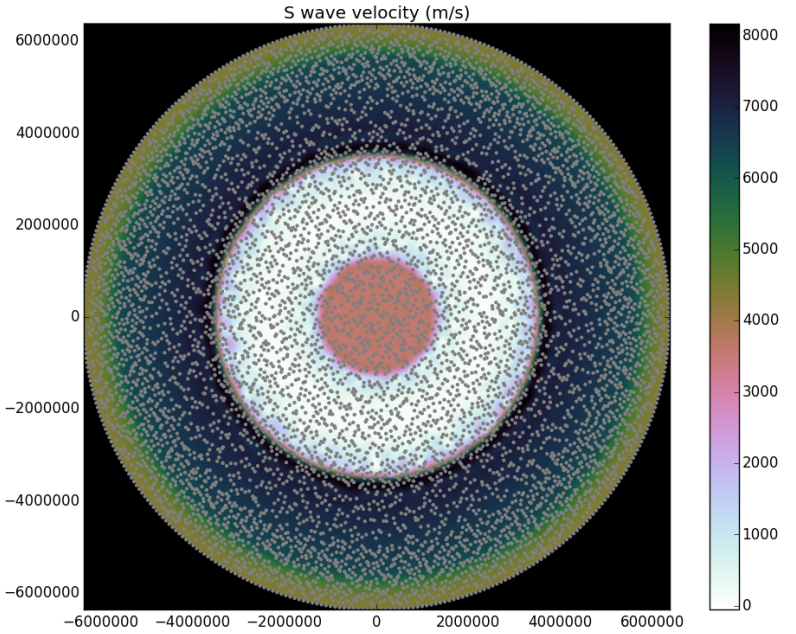

Figure 3: S wave velocities ($\sqrt{\mu/\rho}$) from PREM

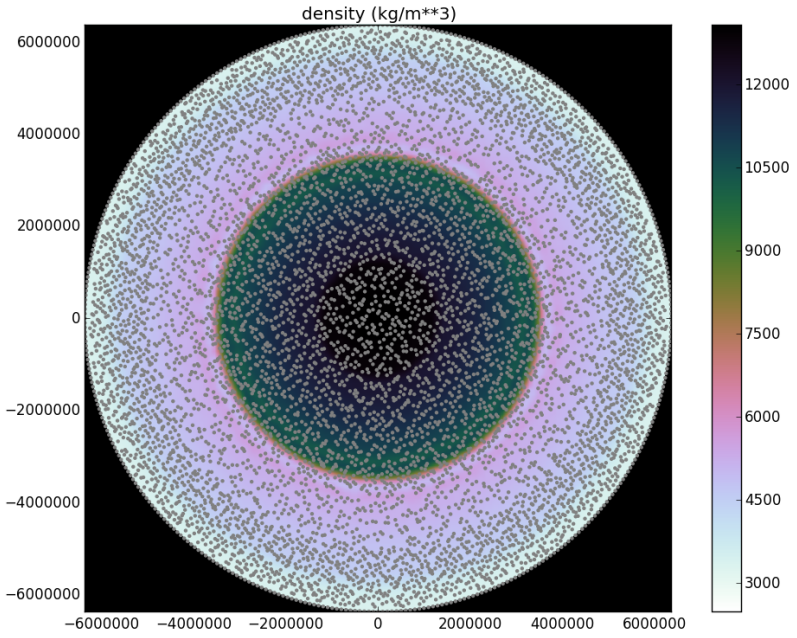

Figure 4: Densities from PREM

by our numerical simulation should be consistent with observable travel times. The material properties for PREM are given at discrete depths, and so we perform an interpolation using multiquadratic RBFs to obtain PREM as a continuous and analytically differentiable function. Our interpolants for PREM are shown in figure 2, 3, and 4.

We also must address how the collocation points are chosen. PREM has a major material discontinuity at the core-mantle boundary and we would ideally prefer to have a higher density of nodes at that interface. Additionally, the lower mantle has higher P and S wave velocities relative to the upper mantle, crust, and core. We would then prefer a lower density of collocation points in the lower mantle in order to keep our solution stable during time stepping. Although we seek regional scale variations in node density, we still want our collocation points to have low discrepancy in any given region of the earth. We have developed a simple method to produce a distribution of collocation points that satisfy our specifications. Let $\psi : \boldsymbol{x} \to [0, 1]$ be a normalized function describing the desired density of collocation points at position $\boldsymbol{x}$. Let H_k^1 be the k^{th} element in a one-dimensional Halton sequence. Let H_k^2 be the k^{th} element of a two-dimensional Halton sequence which has been scaled so that H_k^2 can lie anywhere in the problem domain, D. The set of N collocation points, $\boldsymbol{C}$, with the desired density can then be found with algorithm 1.

In our experience, we found that it is necessary to have a small buffer between the boundary collocation points and interior collocation points. This is essential for a stable solution. We take the width of the buffer to be the smallest distance between boundary nodes. The reader may be able to notice this slight buffer in either Figures 2, 3 or 4.

Algorithm 1 Collocation points

$\boldsymbol{C} = \emptyset$
$k = 0$
while (length of $\boldsymbol{C}$) $< N$ **do**
  if $(\phi(H_k^2) > H_k^1) \& (H_k^2 \in D)$ **then**
    append H_k^2 to $\boldsymbol{C}$
  end if
  $k = k + 1$
end while

Results

Our numerical solution to eq. (4) is shown in figures 5-14. The most prominent feature in our solution are the Raleigh waves which travel along the surface and have decaying amplitude with depth. One can identify the P waves as wave fronts which have displacements parallel to the propagation direction. These are the faint blue, fast traveling waves. The S waves can be identified as having displacements perpendicular to the propagation direction.

It is worth noting that the solution has symmetry about the vertical axis. This symmetry is no surprise and we have attempted to solve eq. (4) on only half the domain. However, we have found that solutions using RBFs are incredibly sensitive to corners in the domain and our solution quickly went unstable after a few time steps. Although we are being inefficient by solving for displacements on a full disk, we are now able to check that our solution is accurate based on the symmetry of our results. With this means of validation, it is apparent that our solution can not possibly be accurate beyond about 800 seconds when substantial asymmetric structures appear in the solution. This is unfortunate because seismic waves can be observed for thousands of seconds after an earthquake. We note that the solution presented here required about $8GB$ of memory and took 30 hours to run on two cores of a desktop computer (using a parallelized matrix solver). A higher density of collocation points is clearly necessary to accurately model seismic wave propagation but we do not have the computational resources for such a calculation. In fact, existing spectral element software for solving seismic wave propagation on a similar two-dimensional domain is intended for use on state of the art super computers (e.g. Nissen-Meyer et al., 2007), so we are not too discouraged by the poor accuracy we obtained using a standard desktop computer.

Conclusion

After our best effort, we were unable to satisfactorily model seismic wave propagation using radial basis functions. This is not due to any fault of the RBF method, but rather due to our computational limitations. This is not to say that RBF methods have no faults. In our experience the RBF method seems quite precarious. A poorly chosen shape parameter or a single collocation point that is too close to the boundary would cause the entire solution to go unstable. Additionally, a corner in the problem domain can lead to an unstable solution, even though the RBF method is touted as being able to handle arbitrary geometries. Further exploration is clearly needed to make our model for seismic wave propagation more robust.

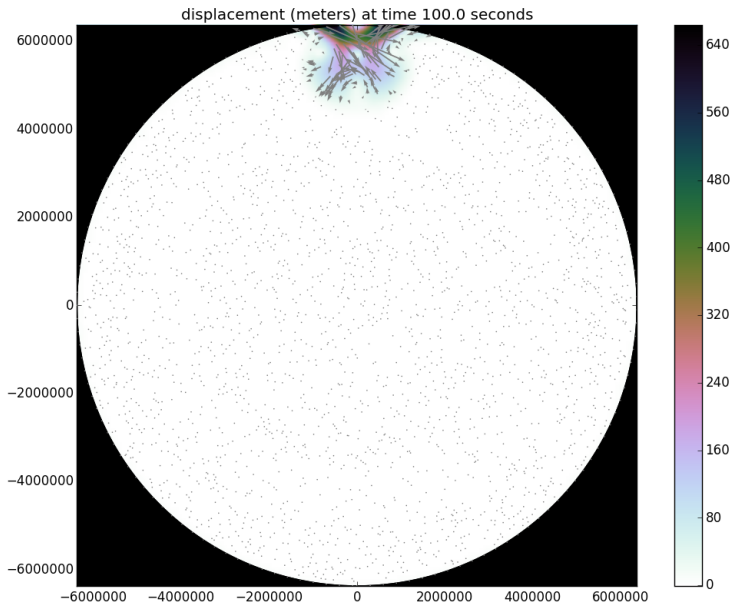

Figure 5: Solution for displacements at 100 seconds after an earthquake. Color indicates the magnitude of displacement and vectors indicate direction

Additional notes

All the software we have written to produce the results shown in this paper are included below. The software (with eventual bug fixes) can also be found at https://github.com/treverhines/SeisRBF. The packages used in our software can be found in the Anaconda Python distribution from Continuum Analytics.

References

Aki, K. & Richards, P., 2002. Quantitative Seismology, second edition, *University Science Books*

Driscoll, T. & Fornberg, B., 2002. Interpolation in the limit of increasingly fiat radial basis functions, *Computers and Mathematics with Applications*, 43, 413-422.

Dziewonski, A. M. & Anderson, D.L., 1981. Preliminary reference Earth model. *Phys Earth Planet Inter*, 25, 297-356. doi:10.1016/0031-9201(81)90046-7.

Fichtner, A., 2011. Full Seismic Waveform Modelling and Inversion, *Springer*

Nissen-Meyer, T., Fournier, A., Dahlen, F. A., 2007. A 2-D spectral-element method for computing spherical-earth seismograms–I. Moment-tensor source, Geophys. J. Int., Vol. 168, issue 3, pp. 1067-1093

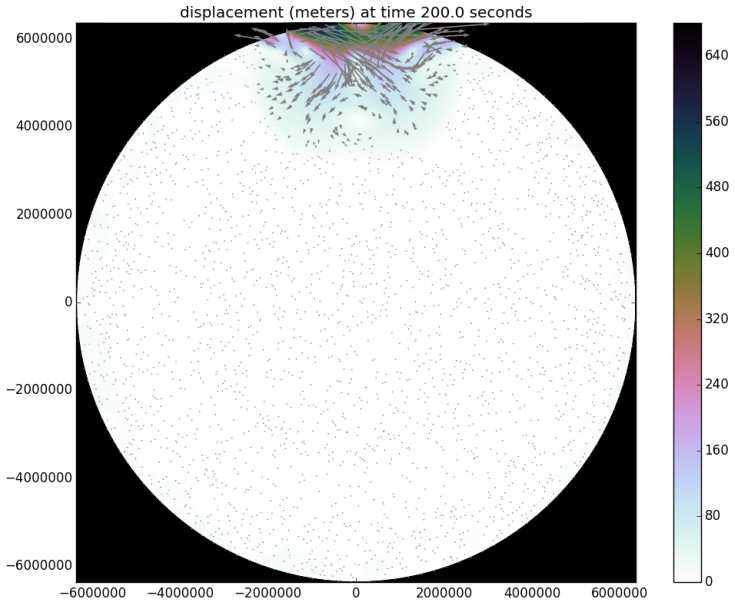

Figure 6: Solution for displacements at 200 seconds after an earthquake

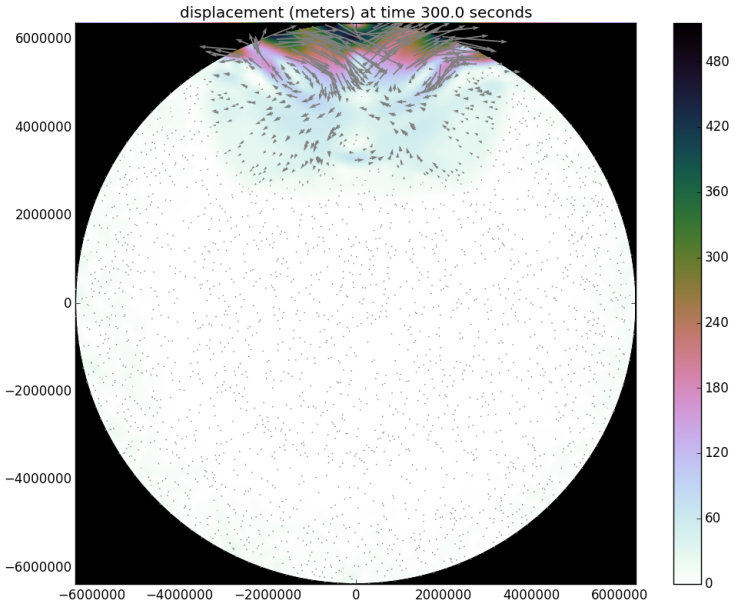

Figure 7: Solution for displacements at 300 seconds after an earthquake

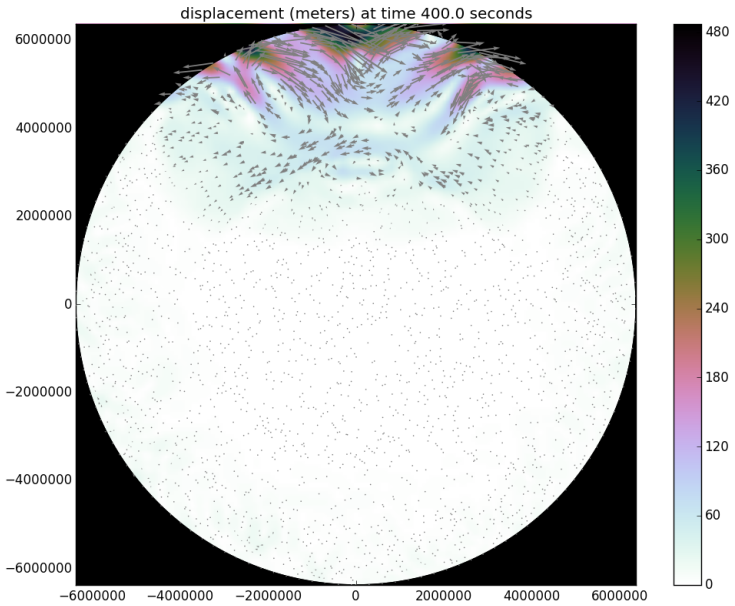

Figure 8: Solution for displacements at 400 seconds after an earthquake

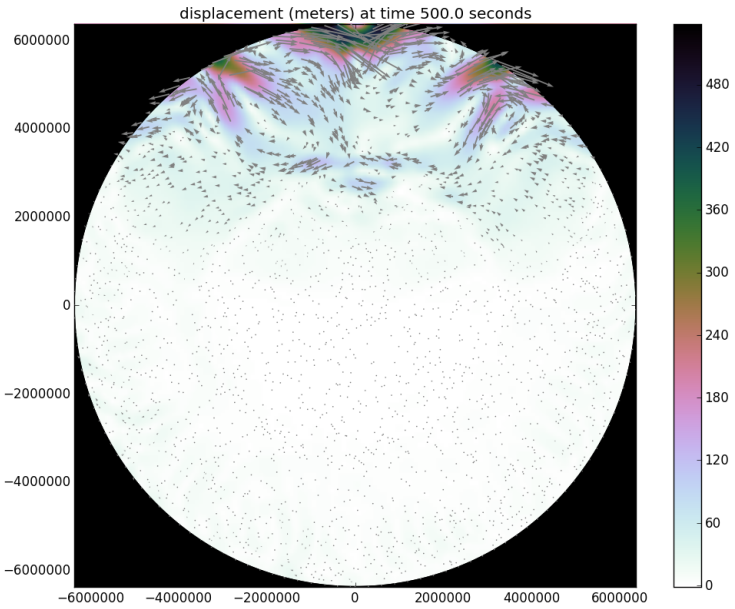

Figure 9: Solution for displacements at 500 seconds after an earthquake

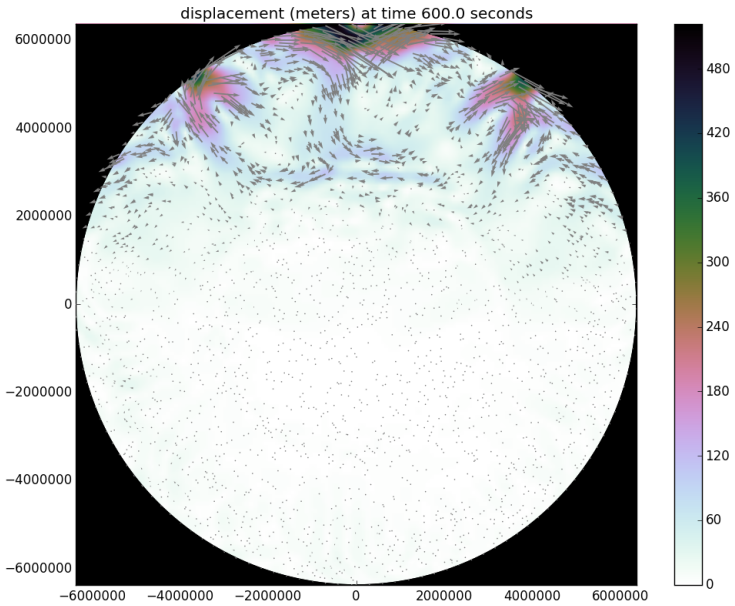

Figure 10: Solution for displacements at 600 seconds after an earthquake

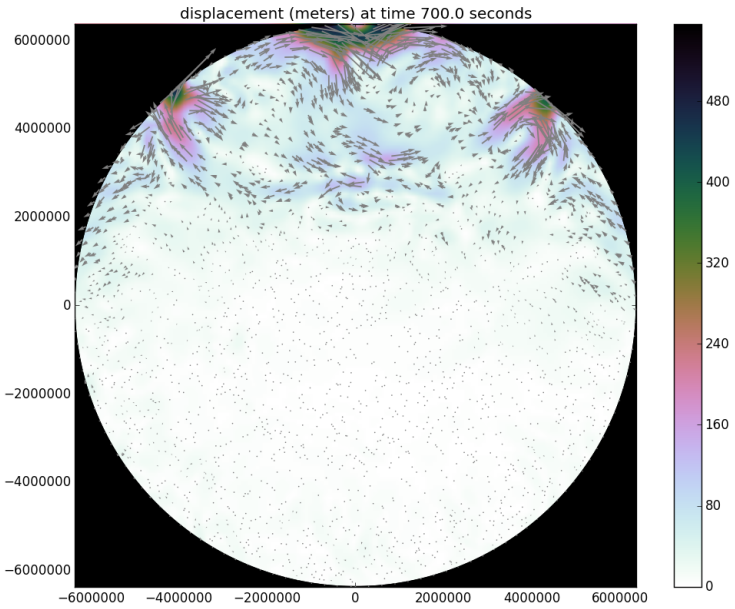

Figure 11: Solution for displacements at 700 seconds after an earthquake

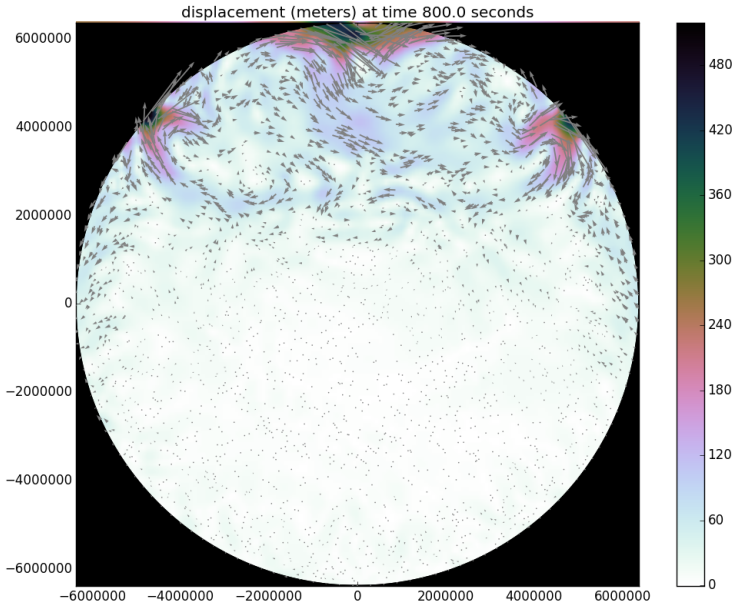

Figure 12: Solution for displacements at 800 seconds after an earthquake

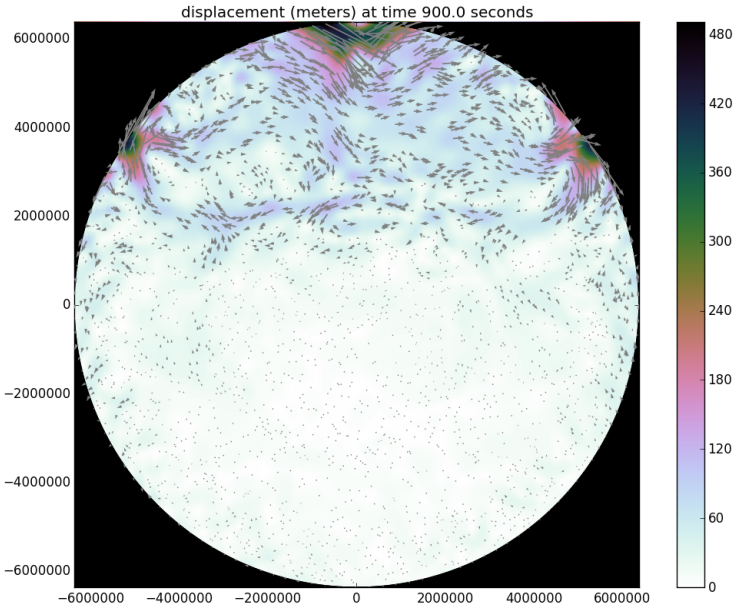

Figure 13: Solution for displacements at 900 seconds after an earthquake

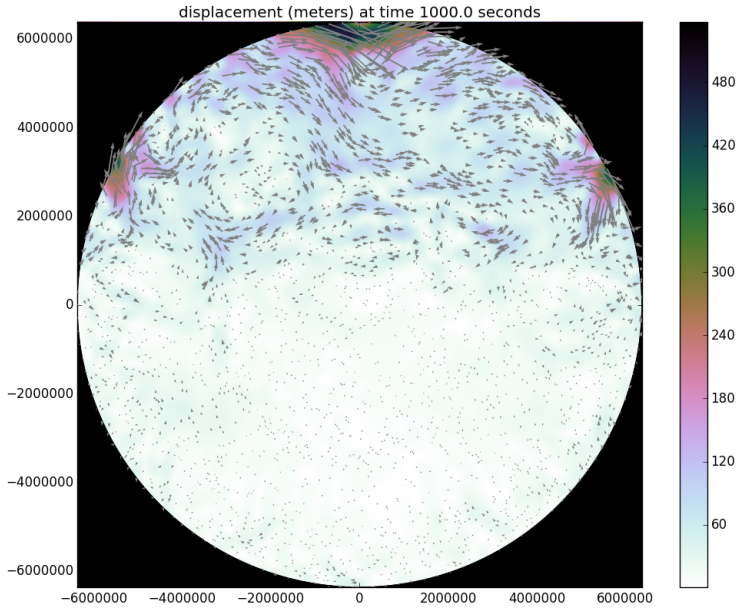

Figure 14: Solution for displacements at 1000 seconds after an earthquake

Listing 1: SeisRBF.py

```
#!/usr/bin/env python
#
# AOSS 555
# Final Project
# Trever Hines
#
# Description
# -----------
#   This program solves the equation of motion for two-dimensional
#   displacement in a two dimensional elastic domain.  This is
#   intended to simulate seismic waves on a global scale

# Dependencies
# ------------
#   Most of the modules used in this program are in the Anaconda
#   Python distribution from Continuum Analytics.  This he program
#   also requires the following files to be exist in your Python path
#
#     usrfcn.py: file containing user defined functions that are
#       specific to the problem being solved
#
#     usrparams.json: This file contains a dictionary of user
#       specified parameters. The parameters can also be specified
#       via the command line when this program is run. for example
#       "SeisRBF.py --nodes 1000" will run this program using 1000
#       nodes
#
```

```
#     radial.py: Contains the radial basis function definitions
#
#     halton.py: Used to create a low discrepancy node distribution
#

import sys
import os
import numpy as np
from radial import mq
from radial import RBFInterpolant
from halton import Halton
from usrfcn import initial_conditions
from usrfcn import source_time_function
from usrfcn import boundary
from usrfcn import node_density
from shapely.geometry import Polygon
from shapely.geometry import Point
from shapely.geometry import LineString
import pickle
import h5py
import argparse
import json
RBF = mq # multiquadtratic radial basis function
SOLVER = np.linalg.solve # this is pythons default solver

def delta(x,x_o,eps):
    '''
    Two-dimensional Gaussian function with center at x_o, standard
    deviation equal to eps, and evaluated at x
    '''
    z = ((x[:,0]-x_o[0])/eps)**2 + ((x[:,1]-x_o[1])/eps)**2
    c = 1.0/(2*np.pi*eps**2)
    return c*np.exp(-z/2)


def force_couple(x,x_o,i,j,d):
    '''
    returns a spatial distribution of forces resulting from component
    (i,j) of the moment tensor.

    Parameters
    ----------
      x: output locations
      x_o: center of the force couple
      i,j: force couple components
      d: distance between the couples

    Returns
    -------
      force density at positions x
    '''
    force_out = np.zeros((len(x),2))
    x_1 = 1.0*x_o
    x_1[j] += d/2.0
```

```
    x_2 = 1.0*x_o
    x_2[j] -= d/2.0
    force_out[:,i] +=  delta(x,x_1,d/4.0)
    force_out[:,i] += -delta(x,x_2,d/4.0)
    return force_out


def source(x,x_o,t,M,d):
    '''
    Returns the force density at position x and time t
    '''
    out = M[0]*force_couple(x,x_o,0,0,d)/d
    out += M[1]*force_couple(x,x_o,1,1,d)/d
    out += M[2]*force_couple(x,x_o,0,1,d)/d
    out += M[2]*force_couple(x,x_o,1,0,d)/d
    return out*source_time_function(t)


def leapfrog(F,u,v,a,dt,F_args=None,F_kwargs=None):
    '''
    Leapfrog time stepping algorithm which solves the next time step in
    d^u/dt^2 = F(u)

    Parameters
    ----------
      F: Funtion which returns both the acceleration at the current time
        step and the spectral coefficients of u. This function takes u
        as the first argument and then F_args and F_kwargs
      u: displacement at the current time step
      v: velocity at the current time step
      a: acceleration at the current time step
      dt: time step size
      F_args: (optional) tuple of dditional arguments to F
      F_kwargs: (optional) dictionary of additional key word arguments
        to F

    Returns
    -------
      tuple of u_new,v_new,a_new, and alpha_new.  These are the
      displacements, velocities, accelerations, and spectral
      coefficients for the new time step

    '''
    if F_args is None:
      F_args = ()
    if F_kwargs is None:
      F_kwargs = {}

    u_new = u + v*dt + 0.5*a*dt**2
    a_new,alpha_new = F(u_new,*F_args,**F_kwargs)
    v_new = v + 0.5*(a + a_new)*dt
    return u_new,v_new,a_new,alpha_new


```

```
def acceleration_matrix(x,c,eps,lam_itp,mu_itp,rho_itp):
    '''
    forms a matrix which returns accelerations at points x when
    multiplied by the spectral coefficients. This is H in the main text

    Paramters
    _________
        x: (N,2) array of points
        c: (M,2) array of RBF centers
        eps: (M,) array of shape parameters
        lam_itp: RBFInterpolant for lambda
        mu_itp: RBFInterpolant for mu
        rho_itp: RBFInterpolant for rho

    Returns
    _______
        H: (N,2,M,2) array. Organized such that a_ij = H_ijkl*alpha_kl,
        where a_ij is the acceleration at point i in direction j, while
        alpha_kl is the spectral coefficient k for displacement in
        direction l

    '''
    N = len(x)
    M = len(c)
    H = np.zeros((N,2,M,2))
    H[:,0,:,0] = ((lam_itp(x,diff=(1,0)) +
                   2*mu_itp(x,diff=(1,0)))*RBF(x,c,eps,diff=(1,0)) +
                  (lam_itp(x,diff=(0,0)) +
                   2*mu_itp(x,diff=(0,0)))*RBF(x,c,eps,diff=(2,0)) +
                  mu_itp(x,diff=(0,1))*RBF(x,c,eps,diff=(0,1)) +
                  mu_itp(x,diff=(0,0))*RBF(x,c,eps,diff=(0,2)))/rho_itp(x)

    H[:,0,:,1] = (lam_itp(x,diff=(1,0))*RBF(x,c,eps,diff=(0,1)) +
                  lam_itp(x,diff=(0,0))*RBF(x,c,eps,diff=(1,1)) +
                  mu_itp(x,diff=(0,1))*RBF(x,c,eps,diff=(1,0)) +
                  mu_itp(x,diff=(0,0))*RBF(x,c,eps,diff=(1,1)))/rho_itp(x)

    H[:,1,:,0] = (mu_itp(x,diff=(1,0))*RBF(x,c,eps,diff=(0,1)) +
                  mu_itp(x,diff=(0,0))*RBF(x,c,eps,diff=(1,1)) +
                  lam_itp(x,diff=(0,1))*RBF(x,c,eps,diff=(1,0)) +
                  lam_itp(x,diff=(0,0))*RBF(x,c,eps,diff=(1,1)))/rho_itp(x)

    H[:,1,:,1] = ((lam_itp(x,diff=(0,1)) +
                   2*mu_itp(x,diff=(0,1)))*RBF(x,c,eps,diff=(0,1)) +
                  (lam_itp(x,diff=(0,0)) +
                   2*mu_itp(x,diff=(0,0)))*RBF(x,c,eps,diff=(0,2)) +
                  mu_itp(x,diff=(1,0))*RBF(x,c,eps,diff=(1,0)) +
                  mu_itp(x,diff=(0,0))*RBF(x,c,eps,diff=(2,0)))/rho_itp(x)

    return H


def traction_matrix(x,n,c,eps,lam_itp,mu_itp):
    '''
```

```
    forms a matrix which returns traction force at points x and normal n
    when multiplied by the spectral coefficients. This is B in the main
    text.

    Paramters
    ---------
    x: (N,2) array of locations
    n: (N,2) array of normal directions
    c: (M,2) array of RBF centers
    eps: (M,) array of shape parameters
    lam_itp: RBFInterpolant for lambda
    mu_itp: RBFInterpolant for mu

    Returns
    -------
      B: shape (N,2,M,2) array. Organized such that
        tau_ij = B_ijkl*alpha_kl, where tau_ij is the traction force in
        direction j at point i with normal direction i, while alpha_kl is
        the spectral coefficient k for displacement in direction l

    '''
    N = len(x)
    M = len(c)
    B = np.zeros((N,2,M,2))
    B[:,0,:,0] = (n[:,[0]]*(lam_itp(x) +
                    2*mu_itp(x))*RBF(x,c,eps,diff=(1,0)) +
                    n[:,[1]]*mu_itp(x)*RBF(x,c,eps,diff=(0,1)))

    B[:,0,:,1] = (n[:,[0]]*lam_itp(x)*RBF(x,c,eps,diff=(0,1)) +
                    n[:,[1]]*mu_itp(x)*RBF(x,c,eps,diff=(1,0)))

    B[:,1,:,0] = (n[:,[0]]*mu_itp(x)*RBF(x,c,eps,diff=(0,1)) +
                    n[:,[1]]*lam_itp(x)*RBF(x,c,eps,diff=(1,0)))

    B[:,1,:,1] = (n[:,[1]]*(lam_itp(x) +
                    2*mu_itp(x))*RBF(x,c,eps,diff=(0,1)) +
                    n[:,[0]]*mu_itp(x)*RBF(x,c,eps,diff=(1,0)))

    return B


def interpolation_matrix(x,c,eps):
    '''
    forms a matrix which returns displacement at points x when
    multiplied by the spectral coefficients.  This is G in the main text

    Paramters
    ---------
      x_ij: shape (N,2) array of locations
      c_ij: shape (M,2) array of RBF centers
      eps_i: shape (M,) array of shape parameters

    Returns
    -------
```

```
    G_ijkl: shape (N,2,M,2) array. Organized such that
      u_ij = G_ijkl*alpha_kl, where u_ij is the displacement in
      direction j at point i, while alpha_kl is the spectral coefficient
      k for displacement in direction l

  '''
  N = len(x)
  M = len(c)
  G = np.zeros((N,2,M,2))
  G[:,0,:,0] = RBF(x,c,eps)
  G[:,1,:,1] = RBF(x,c,eps)

  return G


def nearest(x):
  '''
  Returns the distance to nearest point for each point in x. It also
  returns the index of the nearest point.  This is used to determine
  the shape parameter for each radial basis function.
  '''
  tol = 1e-4
  x = np.asarray(x)
  if len(np.shape(x)) == 1:
    x = x[:,None]

  N = len(x)
  A = (x[None] - x[:,None])**2
  A = np.sqrt(np.sum(A,2))
  A[range(N),range(N)] = np.max(A)
  nearest_dist = np.min(A,1)
  nearest_idx = np.argmin(A,1)
  if any(nearest_dist < tol):

  return nearest_dist,nearest_idx


def pick_nodes(H,D,R,N):
  '''
  This is algorithm 1 in the main text

  Parameters
  ----------
    H: 3 dimensional Halton sequence (first two dimensions are used
      for testing points in the domain and the third dimension is
      used for accepting/rejecting)
    D: Polygon instance which defines the domain boundary
    R: Density function which takes a coordinate and returns the
      desired node density at that point
    N: number of nodes

  Returns
  -------
    out: (N,2) array of nodes within the domain and distributed with
```

```
        the desired density
    '''
    minval = np.min(D.exterior.xy)
    maxval = np.max(D.exterior.xy)

    out = []
    while len(out) < N:
        Hk3 = H(1)[0]
        Hk2 = Hk3[[0,1]]
        Hk1 = Hk3[2]
        # scale Hk2 so that it encompasses the domain
        Hk2 *= (maxval-minval)
        Hk2 += minval
        if (R(Hk2) > Hk1) & D.contains(Point(Hk2)):
            out += [Hk2]

    out = np.array(out)
    return out


def F(u,J,H,f,BCidx,BCval):
    '''
    computes acceleration from u.  This is done by first finding the
    spectral coefficients from u and then using the spectral
    coefficients to evaluate the derivative. The matrix J is the
    interpolation matrix except that rows BCidx have been swapped out
    with rows that will enforce the boundary conditions.  The
    corresponding rows in u get replaced with BCval in this function
    '''
    # set boundary conditions
    u[BCidx,:] = BCval

    # Reshape the arrays into something tractable
    N,D1,M,D2 = np.shape(J)
    u_shape = (N,D1)
    assert np.shape(u) == u_shape
    alpha_shape = (M,D2)
    J = np.reshape(J,(N,D1,M*D2))
    J = np.einsum('ijm->mij',J)
    J = np.reshape(J,(M*D2,N*D1))
    J = np.einsum('mn->nm',J)
    u = np.reshape(u,N*D1)


    # solve for alpha
    alpha = SOLVER(J,u)

    # reshape alpha into a 2D array
    alpha = np.reshape(alpha,alpha_shape)

    # solve for acceleration
    a = np.einsum('ijkl,kl',H,alpha)

    # add acceleration due to forcing term
```

```
    a = a + f

    return a,alpha


def initial_save(name,
                 nodes,
                 surface_idx,
                 interior_idx,
                 eps,
                 mu,
                 lam,
                 rho,
                 time_steps):
    '''
    Saves the progress of the simulation
    '''
    f = h5py.File('output/%s/%s.h5' % (name,name),'w')
    f['name'] = name
    f['rbf'] = RBF.__name__
    f['nodes'] = np.asarray(nodes)
    f['surface_index'] = np.asarray(surface_idx)
    f['interior_index'] = np.asarray(interior_idx)
    f['epsilon'] = np.asarray(eps)
    f['mu'] = np.asarray(mu)
    f['lambda'] = np.asarray(lam)
    f['rho'] = np.asarray(rho)
    f['time'] = np.asarray(time_steps)
    f['alpha'] = np.zeros((len(time_steps),)+np.shape(nodes))
    f.close()
    return


def update_save(name,time_indices,alpha):
    '''
    Saves the progress of the simulation
    '''
    f = h5py.File('output/%s/%s.h5' % (name,name),'a')
    alpha = np.asarray(alpha)
    f['alpha'][time_indices,:,:] = alpha
    f.close()
    return


def main(args):
    # setup logger
    name = args.name
    os.makedirs('output/%s' % name)

    # Define Topology
    # ---------------
    # Keep track of the total number of nodes
    node_count = 0
    total_nodes = np.zeros((0,2))
```

```
407 # initiate a Halton sequence
    H = Halton(dim=3)
409
    # define boundary nodes
411 p = np.linspace(0,1,args.boundary_nodes+1)[:-1]
    surface_nodes = boundary(p)
413 surface_idx = range(node_count,len(surface_nodes)+node_count)
    total_nodes = np.vstack((total_nodes,surface_nodes))
415 node_count += len(surface_nodes)

417 # find the normal vectors for each surface node, this is used for
    # applying the boundary conditions
419 dp = 1e-6
    surface_nodes_plus_dp = boundary(p+dp)
421 surface_tangents = surface_nodes_plus_dp - surface_nodes
    surface_normals = np.zeros((args.boundary_nodes,2))
423 surface_normals[:,0] = surface_tangents[:,1]
    surface_normals[:,1] = -surface_tangents[:,0]
425 normal_lengths = np.sqrt(np.sum(surface_normals**2,1))
    surface_normals = surface_normals/normal_lengths[:,None]
427 # Define polygon based on boundary nodes
    D = Polygon(surface_nodes)
429
    # Add a buffer so that the polygon is slightly smaller than what is
431 # defined by the boundary nodes. This is needed for a stable
    # solution.  The buffer is the minimum of the distances between
433 # adjacent surface nodes
    nearest_surface_node = nearest(surface_nodes)[0]
435 for p,n in zip(surface_nodes,nearest_surface_node):
      pbuff = Point(p).buffer(n)
437   D = D.difference(pbuff)

439 assert D.is_valid

441 # Find interior nodes that are within the domain and adhere to the
    # user specified node density
443 interior_nodes = pick_nodes(H,D,node_density,args.nodes-node_count)
    interior_idx = range(node_count,len(interior_nodes)+node_count)
445 total_nodes = np.vstack((total_nodes,interior_nodes))
    node_count += len(interior_nodes)
447
    # Material Properties
449 # ---------------------
    # load P-wave and S-wave velocities from PREM table
451 prem_table = np.loadtxt(args.material_file,skiprows=1)
    depth = prem_table[:,0]*1000 # m
453 P_vel = prem_table[:,1] # m/s
    S_vel = prem_table[:,2] # m/s
455 rho = prem_table[:,3] # kg/m**3

457 # lame parameters
    mu = (S_vel**2*rho) # kg/m*s**2
459 lam = (P_vel**2*rho - 2*mu) # kg/m*s**2
```

```
461  # create interpolants. The interpolants are 1D with respect to
     # depth
463  eps = args.epsilon/nearest(depth)[0]
     lam_itp = RBFInterpolant(depth,eps,value=lam)
465  mu_itp = RBFInterpolant(depth,eps,value=mu)
     rho_itp = RBFInterpolant(depth,eps,value=rho)
467
     # Convert the 1D interpolants to 2D cartesian interpolants
469  r = np.sqrt(np.sum(total_nodes**2,1))
     eps = args.epsilon/nearest(total_nodes)[0]
471  lam_itp = RBFInterpolant(total_nodes,eps,value=lam_itp(r))
     mu_itp = RBFInterpolant(total_nodes,eps,value=mu_itp(r))
473  rho_itp = RBFInterpolant(total_nodes,eps,value=rho_itp(r))

475  # Build Problem Matrices
     # ----------------------
477  # calculate shape parameter
     eps = args.epsilon/nearest(total_nodes)[0]
479
     # Form interpolation, acceleration, and boundary condition matrix
481  G = interpolation_matrix(total_nodes,total_nodes,eps)
     A = acceleration_matrix(total_nodes,total_nodes,eps,
483                          lam_itp,mu_itp,rho_itp)
     B = traction_matrix(surface_nodes,surface_normals,total_nodes,eps,
485                      lam_itp,mu_itp)/1e8
     G[surface_idx,:,:,:] = B
487
     # Print Run Information
489  # ---------------------
     min_dist = np.min(nearest(total_nodes)[0])
491  max_speed = np.max(np.concatenate((P_vel,S_vel)))

493  # Time Stepping
     # -------------
495  time_steps = np.arange(0.0,args.max_time,args.time_step)
     total_steps = len(time_steps)
497
     initial_save(name,
499               total_nodes,
                  surface_idx,
501               interior_idx,
                  eps,
503               mu_itp(total_nodes),
                  lam_itp(total_nodes),
505               rho_itp(total_nodes),
                  time_steps)
507
     last_save = 0
509  alpha_list = []
     for itr,t in enumerate(time_steps):
511    if itr == 0:
         # compute initial velocity and displacement
513      u,v = initial_conditions(total_nodes)
```

```
      # compute acceleration due to source term
      f = source(total_nodes,
                 np.asarray(args.epicenter),
                 t,
                 np.asarray(args.moment_tensor),
                 args.couple_distance)
      f = f/rho_itp(total_nodes)
      # compute acceleration and the spectral coefficients
      a,alpha = F(u,G,A,f,surface_idx,0.0)
      alpha_list += [alpha]
      u = np.einsum('ijkl,kl',G,alpha)

    else:
      # compute acceleration due to source term
      f = source(total_nodes,
                 np.asarray(args.epicenter),
                 t,
                 np.asarray(args.moment_tensor),
                 args.couple_distance)
      f = f/rho_itp(total_nodes)
      # compute next step from leapfrom iteration
      u,v,a,alpha = leapfrog(F,u,v,a,args.time_step,
                             F_args=(G,A,f,surface_idx,0.0))
      alpha_list += [alpha]

    if ((itr+1)%args.save_interval==0) | ((itr+1) == len(time_steps)):
      update_save(name,range(last_save,itr+1),alpha_list)
      last_save = itr+1
      alpha_list = []


if __name__ == '__main__':
  # set up command line argument parser
  parser = argparse.ArgumentParser(
           description='''Computes two-dimensional displacements
             resulting from seismic wave propagation''')

  parser.add_argument('--name',type=str,help='''run name''')

  parser.add_argument('--nodes',type=int,metavar='int',
                      help='''total number of collocation points which
                        are the centers of each RBF''')

  parser.add_argument('--boundary_nodes',type=int,metavar='int',
                      help='''number of boundary nodes''')

  parser.add_argument('--rbf',type=str,metavar='str',
                      help='''type of RBF, either mq, ga, or iq''')

  parser.add_argument('--max_time',type=float,metavar='float',
                      help='''model run time in seconds''')

  parser.add_argument('--time_step',type=float,metavar='float',
                      help='''size of each time step in seconds''')
```

```
569   parser.add_argument('--save_interval',type=float,metavar='float',
                          help='''number of time steps between each
571                         save''')

573   parser.add_argument('--epsilon',type=float,metavar='float',
                          help='''Uniformly scales all shape parameters
575                         after they have normalized by the nearest
                            adjacent node. So epsilon=2 would make each
577                         RBFs shape parameter equal to twice the
                            distance to the nearest node''')
579
      parser.add_argument('--epicenter',nargs=2,type=float,
581                       metavar='float',
                          help='''Two coordinates in km indicating the
583                         location of the earthquake epicenter''')

585   parser.add_argument('--couple_distance',nargs=2,type=float,
                          metavar='float',
587                       help='''distance in km between force couples.
                            Ideally this would be infinitesimally small;
589                         however, this should be on order of the node
                            spacing''')
591
      parser.add_argument('--moment_tensor',nargs=3,type=float,
593                       metavar='float',
                          help='''Three unique components of the moment
595                         tensor in N m.  These need to be given in the
                            order (M11, M22, M12)''')
597
      parser.add_argument('--material_file',type=str,metavar='str',
599                       help='''Three unique components of the moment
                            tensor in N m.  These need to be given in the
601                         order (M11, M22, M12)''')

603   if os.path.exists('usrparams.json'):
        argfile = open('usrparams.json','r')
605     default_args = json.load(argfile)
      else:
607     default_args = {}

609   default_args['name'] = 'default_name'
      parser.set_defaults(**default_args)
611   args = parser.parse_args()

613   main(args)
```

Listing 2: radial.py

```
1 #!/usr/bin/env python
  #
3 # This module defines the gaussian, multiquadratic, and inverse
  # quadratic radial basis functions.  This module makes use of the
5 # class, RBF, which takes a symbolic expression of an RBF and converts
```

```
# it and its derivatives into a numerical function. So you can
# evaluate any arbitrary derivative of an RBF even though the
# derivatives are not explicitly written anywhere in this module.
#
# For example, consider the Gaussian RBF which is saved as 'ga'.
# If we want to evaluate the first derivative of ga with respect to
# the second spatial dimension then the command would be.
#
# >> ga(x,centers,eps,diff=(0,1)).
#
# See the help documentation for ga for more information about its
# arguments

from __future__ import division
import sympy
import numpy as np

_R = sympy.symbols('R')
_EPS = sympy.symbols('EPS')

class RBF(object):
  def __init__(self,expr):
    '''
    Parameters
    ----------
      expr: Symbolic expression of the RBF with respect to _R and _EPS
    '''
    assert expr.has(_R), (
      'RBF expression does not contain _R')

    assert expr.has(_EPS), (
      'RBF expression does not contain _EPS')
    self.R_expr = expr
    self.diff_dict = {}

  def __call__(self,x,c,eps=None,diff=None):
    '''
    evaluates M radial basis functions (RBFs) with arbitary dimension
    at N points.

    Parameters
    ----------
      x: ((N,) or (N,D) array) locations to evaluate the RBF

      centers: ((M,) or (M,D) array) centers of each RBF

      eps: ((M,) array, default=np.ones(M)) Scale parameter for each RBF

      diff: ((D,) tuple, default=(0,)*dim) a tuple whos length is
        equal to the number of spatial dimensions.  Each value in the
        tuple must be an integer indicating the order of the
        derivative in that spatial dimension.  For example, if the the
        spatial dimensions of the problem are 3 then diff=(2,0,1)
        would compute the second derivative in the first dimension and
```

```
        the first derivative in the third dimension.
61
    Returns
63  -------
      out: (N,M) array for each M RBF evaluated at the N points
65
    Note
67  ----
      the derivatives are computed symbolically in Sympy and then
69    lambdified to evaluate the expression with the provided values.
      The lambdified functions are cached in the scope of the radial
71    module and will be recalled if a value for diff is used more
      than once in the Python session.
73
    '''
75  x = np.asarray(x)
    c = np.asarray(c)
77  xshape = np.shape(x)
    cshape = np.shape(c)
79  assert (len(xshape) == 1) | (len(xshape) == 2), (
      'x must be a 1-D or 2-D array')
81  assert (len(cshape) == 1) | (len(cshape) == 2), (
      'c must be a 1-D or 2-D array')
83
    if len(xshape) == 1:
85    x = x[:,None,None]

87  if len(cshape) == 1:
      c = c[None,:,None]
89
    if len(xshape) == 2:
91    x = x[:,None,:]

93  if len(cshape) == 2:
      c = c[None,:,:]
95
    N = np.shape(x)[0]
97  M = np.shape(c)[1]
    assert np.shape(x)[2] == np.shape(c)[2], (
99    'if x and c are 2-D arrays then their second dimensions must '
      'have the same length')
101 dim = np.shape(x)[2]
    if eps is None:
103   eps = np.ones(M)

105 eps = np.asarray(eps)
    assert len(np.shape(eps)) == 1, (
107   'eps must be a 1D array')

109 assert len(eps) == M, (
      'length of eps must be equal to the number of centers')
111
    x = np.einsum('ijk->kij',x)
113 c = np.einsum('ijk->kij',c)
```

```
115     if diff is None:
          diff = (0,)*dim
117
        while len(diff) < dim:
119       diff += (0,)
121     assert len(diff) == dim, (
          'cannot specify derivatives for dimensions that are higher than '
123       'the dimensions of x and center')
125     if diff not in self.diff_dict:
          self._make_function(diff)
127
        args = (tuple(x)+tuple(c)+(eps,))
129     return self.diff_dict[diff](*args)
131   def _make_function(self,diff):
        dim = len(diff)
133     c_sym = sympy.symbols('c:%s' % dim)
        x_sym = sympy.symbols('x:%s' % dim)
135     r = sympy.sqrt(sum((x_sym[i]-c_sym[i])**2 for i in range(dim)))
        expr = self.R_expr.subs(_R,r)
137     for direction,order in enumerate(diff):
          if order == 0:
139         continue
          expr = expr.diff(*(x_sym[direction],)*order)
141
        self.diff_dict[diff] = sympy.lambdify(x_sym+c_sym+(_EPS,),expr,'numpy')
143
class RBFInterpolant(object):
145   '''
      A callable RBF interpolant
147   '''
      def __init__(self,
149                x,
                   eps,
151                value=None,
                   alpha=None,
153                rbf=None):
        '''
155     Initiates the RBF interpolant

157     Parameters
        ----------
159       x: ((N,) or (N,D) array) x coordinate of the interpolation
            points which also make up the centers of the RBFs
161
          eps: ((N,) array) shape parameters for each RBF
163
          value: ((N,) or (N,R) array) Values at the x coordinates. If this
165         is not provided then alpha must be given

167       alpha: ((N,) or (N,R) array) Coefficients for each RBFs. If this
```

```
      is not provided then value must be given

      rbf: type of rbf to use. either mq, ga, or iq
    '''
    if rbf is None:
      rbf = mq

    assert (value is not None) != (alpha is not None), (
      'either val or alpha must be given')

    x = np.asarray(x)
    eps = np.asarray(eps)
    x_shape = np.shape(x)
    N = x_shape[0]
    assert len(x_shape) <= 2
    assert np.shape(eps) == (N,)

    if len(x_shape) == 1:
      x = x[:,None]

    if alpha is not None:
      alpha = np.asarray(alpha)
      alpha_shape = np.shape(alpha)
      assert len(alpha_shape) <= 2
      assert alpha_shape[0] == N
      if len(alpha_shape) == 1:
        alpha = alpha[:,None]

      alpha_shape = np.shape(alpha)
      R = alpha_shape[1]

    if value is not None:
      value = np.asarray(value)
      value_shape = np.shape(value)
      assert len(value_shape) <= 2
      assert value_shape[0] == N
      if len(value_shape) == 1:
        value = value[:,None]

      value_shape = np.shape(value)
      R = value_shape[1]

    if alpha is None:
      alpha = np.zeros((N,R))
      G = rbf(x,x,eps)
      for r in range(R):
        alpha[:,r] = np.linalg.solve(G,value[:,r])

    self.x = x
    self.eps = eps
    self.alpha = alpha
    self.R = R
    self.rbf = rbf
```

```
223   def __call__(self,xitp,diff=None):
        '''
225     Returns the interpolant evaluated at xitp

227     Parameters
        ----------
229       xitp: ((N,) or (N,D) array) points where the interpolant is to
            be evaluated
231
          diff: ((D,) tuple, default=(0,)*dim) a tuple whos length is
233         equal to the number of spatial dimensions. Each value in the
            tuple must be an integer indicating the order of the
235         derivative in that spatial dimension. For example, if the the
            spatial dimensions of the problem are 3 then diff=(2,0,1)
237         would compute the second derivative in the first dimension and
            the first derivative in the third dimension.
239
        '''
241     out = np.zeros((len(xitp),self.R))
        for r in range(self.R):
243       out[:,r] = np.sum(self.rbf(xitp,
                                     self.x,
245                                  self.eps,
                                     diff=diff)*self.alpha[:,r],1)
247
        return out
249

251 _FUNCTION_DOC = '''
      evaluates M radial basis functions (RBFs) with arbitary dimension at
253   N points.

255   Parameters
      ----------
257     x: ((N,) or (N,D) array) locations to evaluate the RBF

259     centers: ((M,) or (M,D) array) centers of each RBF

261     eps: ((M,) array, default=np.ones(M)) Scale parameter for each RBF

263     diff: ((D,) tuple, default=(0,)*dim) a tuple whos length is equal
          to the number of spatial dimensions. Each value in the tuple
265       must be an integer indicating the order of the derivative in
          that spatial dimension. For example, if the the spatial
267       dimensions of the problem are 3 then diff=(2,0,1) would compute
          the second derivative in the first dimension and the first
269       derivative in the third dimension.

271   Returns
      -------
273     out: (N,M) array for each M RBF evaluated at the N points

275   Note
```

```
    -----
    the derivatives are computed symbolically in Sympy and
    then lambdified to evaluate the expression with the provided values.
    The lambdified functions are cached in the scope of the radial
    module and will be recalled if a value for diff is used more than
    once in the Python session.'''

_IQ = RBF(1/(1+(_EPS*_R)**2))
def iq(*args,**kwargs):
    '''
    Inverse Quadratic
    '''
    return _IQ(*args,**kwargs)

iq.__doc__ += _FUNCTION_DOC

_GA = RBF(sympy.exp(-(_EPS*_R)**2))
def ga(*args,**kwargs):
    '''
    Gaussian
    '''
    return _GA(*args,**kwargs)

ga.__doc__ += _FUNCTION_DOC

_MQ = RBF(sympy.sqrt(1 + (_EPS*_R)**2))
def mq(*args,**kwargs):
    '''
    multiquadratic
    '''
    return _MQ(*args,**kwargs)

mq.__doc__ += _FUNCTION_DOC
```

Listing 3: halton.py

```
#!/usr/bin/env python
from __future__ import division
import numpy as np

class Prime(object):
    '''
    enumerates over N prime numbers
    '''
    def __init__(self,N):
        self.N = N
        self.n = 0

    def __iter__(self):
        return self

    def next(self):
        if self.n == self.N:
            raise StopIteration
```

```
    elif self.n == 0:
      self.primes = [2]
      self.n += 1

    else:
      test = self.primes[-1] + 1
      while True:
        if not any(test/p == test//p for p in self.primes):
          self.primes += [test]
          self.n += 1
          break

        test += 1

    out = self.primes[-1]
    return out

class Halton(object):
  '''
  A class which produces a Halton sequence when called and remembers
  the state of the sequence so that repeated calls produce the next
  items in the sequence.

  '''
  def __init__(self,dim=1,start=0,skip=1):
    '''
    Parameters
    ----------
      dim: (default 1) dimensions of the Halton sequence

      start: (default 0) Index to start at in the Halton sequence

      skip: (default 1) Indices to skip between successive
        output values

    '''
    self.count = start
    self.skip = skip
    self.dim = dim

  def __call__(self,N):
    '''
    Parameters
    ----------
      N: (integer) Number of elements of the Halton sequence to return

    Returns
    -------
      (N,dim) array of elements from the Halton sequence
    '''
    out = halton(N,self.dim,self.count,self.skip)
    self.count += N*self.skip
    return out
```

```
74    def qunif(self,low=0.0,high=1.0,N=50):
          '''
76        Returns the halton sequence with values scaled to be between low
          and hi
78        '''
          return self(N)*(high-low) + low
80
82  def halton(N,dim=1,start=0,skip=1):
      '''
84    returns a halton sequence using the algorithm from
      http://en.wikipedia.org/wiki/Halton_sequence
86    '''
      out = np.zeros((N,dim))
88    for d,base in enumerate(Prime(dim)):
        i = start + 1 + np.arange(0,skip*N,skip)
90      f = np.ones(N)
        while any(i > 0):
92        f = f/base
          out[:,d] += f*(i%base)
94        i //= base

96    return out
```

Listing 4: usrfcn.py

```
1 #!/usr/bin/env python
  #
3 # User defined functions used by SeisRBF.py. The user should modify
  # these functions based on their problem specifications
5
  import numpy as np
7
  def initial_conditions(x):
9   '''
    Initial displacement an velocity field.
11
    Parameters
13  ----------
      x: array of points where the initial displacments and velocity
15       field are to be output.  This is an (N,D) array where N is
         the number of points and D is the number of spatial dimensions
17
    Returns
19  -------
      u: displacement field
21    v: velocity field
    '''
23  u = np.zeros(np.shape(x)) # returns 0 for initial disp
    v = np.zeros(np.shape(x)) # returns 0 for initial vel
25  return u,v

27
```

```
def source_time_function(t):
    '''
    Describes the time evolution of seismic moment

    Parameters
    ----------
      t: scalar value of time

    Returns
    -------
      The proportion of the seismic moment that has been released at
      the given time.  This value should be between 0 and 1
    '''
    out = 1.0 # all seismic moment is instantly released
    return out


def boundary(t):
    '''
    parameterized boundary function

    Parameters
    ----------
      t: array with values between 0 and 1

    Returns
    -------
      The coordinates of the boundary that correspond with the given
      values of t.  The coordinates should make an full loop as t goes
      from 0 to 1

    '''
    p = 2*np.pi*t
    out = np.zeros((len(t),2))
    out[:,0] =  6371000*np.sin(p) # The boundary is a circle with radius
    out[:,1] = -6371000*np.cos(p) # equal to that of the earth in meters

    return out

def node_density(x):
    '''
    returns a value between 0 and 1 indicating the desired density
    of nodes at the given points. This function corresponds to psi
    in my paper.

    Parameters
    ----------
      x: Array of points where node density is to be given. This is an
         N by D array, where N is the number of points and D is the
         number of spatial dimensions

    Returns
    -------
      length N array of node densities between 0 and 1
```

```
83  '''
    layer_depths = [[np.inf,5800],
85                  [5800,5000],
                    [5000,3700],
87                  [3700,3300],
                    [3300,1500],
89                  [1500,1100],
                    [1100,0]]
91
    layer_depths = 1000*np.array(layer_depths) # meters
93
    layer_density = [1.0,
95                   1.0,
                     0.5,
97                   1.0,
                     0.7,
99                   0.7,
                     0.7]
101
    R = np.sqrt(x[0]**2 + x[1]**2)
103 #R = np.sqrt(np.sum(x**2,1))
    #out = np.zeros(len(R))
105 for depths,density in zip(layer_depths,layer_density):
      if (R<=depths[0]) & (R>depths[1]):
107     return density

109 #out[(R<=depths[0])&(R>depths[1])] = density
    #return out
```