Travel time calculation in regular 3D grid in local and regional scale using fast marching method

Marcin POLKOWSKI
Institute of Geophysics, Faculty of Physics, University of Warsaw, Poland (marcin@marcinpolkowski.com)

Abstract

Local and regional 3D seismic velocity models of crust and sediments are very important for numerous technics like mantle and core tomography, localization of local and regional events and others. Most of those techniques require calculation of wave travel time through the 3D model. This can be achieved using multiple approaches from simple ray tracing to advanced full waveform calculation. In this study simple and efficient implementation of fast marching method is presented. This method provides more information than ray tracing and is much less complicated than methods like full waveform being the perfect compromise. Presented code is written in C++, well commented and is easy to modify for different types of studies. Additionally performance is widely discussed including possibilities of multithreading and massive parallelism like GPU. Source code will be published in 2016 as it is part of the PhD thesis. National Science Centre Poland provided financial support for this work via NCN grant DEC-2011/02/A/ST10/00284.

The theory

All calculations in my implementation are done according to Sethian & Popovici 1999, Chopp 2001 and Popovici & Sethian 2002.

The eikonal equation states:

$$(\nabla T)^2 = \frac{1}{c^2}$$

where T is traveltime and c is wave velocity.

The equation above can be rewritten as:

$$\begin{bmatrix} \max\left(D_a^{-x}T, -D_b^{+x}T, 0\right)^2 + \\ \max\left(D_c^{-y}T, -D_d^{+y}T, 0\right)^2 + \\ \max\left(D_e^{-z}T, -D_f^{+z}T, 0\right)^2 \end{bmatrix}_{ijk}^{1/2} = s_{i,j,k}$$

where s is slowness and D are proper coefficients depending on coordinate system chosen:

$$D_1^{-x}T_{i,j,k} = \frac{T_{i,j,k} - T_{i-1,j,k}}{\delta x}$$

$$D_2^{-x}T_{i,j,k} = \frac{3T_{i,j,k} - 4T_{i-1,j,k} + T_{i-2,j,k}}{2\delta x}$$

$$D_1^{+x}T_{i,j,k} = \frac{T_{i,j,k} - T_{i-1,j,k}}{\delta r}$$

$$D_2^{+x}T_{i,j,k} = \frac{3T_{i,j,k} - 4T_{i-1,j,k} + T_{i-2,j,k}}{2\delta r}$$

$$D_1^{-\theta}T_{i,j,k} = \frac{T_{i,j,k} - T_{i,j-1,k}}{r\delta\theta}$$

$$D_2^{-\theta}T_{i,j,k} = \frac{3T_{i,j,k} - 4T_{i,j-1,k} + T_{i,j-2,k}}{2r\delta\theta}$$

$$D_1^{+\phi}T_{i,j,k} = \frac{T_{i,j,k} - T_{i,j,k-1}}{r\cos\theta\delta\phi}$$

$$D_2^{+\phi}T_{i,j,k} = \frac{3T_{i,j,k} - 4T_{i,j,k-1} + T_{i,j,k-2}}{2r\cos\theta\delta\phi}$$

Solution to this equation is easy to calculate because for each max() in equation we get one or none element depending on data availability (see step by step example). For example, if we would like to calculate traveltime in grid cell knowing time in previous cell in x and y direction:

$$\begin{bmatrix} \max\left(D_a^{-x}T, -D_b^{+x}T, 0\right)^2 + \\ \max\left(D_c^{-y}T, -D_d^{+y}T, 0\right)^2 + \\ \max\left(D_e^{-z}T, -D_f^{+z}T, 0\right)^2 \end{bmatrix}_{ijk}^{1/2} = s_{i,j,k}$$

$$\left(\frac{T_{i,j,k} - T_{i-1,j,k}}{\delta x}\right)^2 + \left(\frac{T_{i,j,k} - T_{i,j-1,k}}{\delta y}\right)^2 = \frac{1}{v^2}$$

$$T_{i,j,k}^2\left(\frac{1}{\delta x^2} + \frac{1}{\delta y^2}\right) + T_{i,j,k}\left(\frac{-2T_{i-1,j,k}}{\delta x^2} + \frac{-2T_{i,j-1,k}}{\delta y^2}\right) + \left(\frac{T_{i-1,j,k}^2}{\delta x^2} + \frac{T_{i,j-1,k}^2}{\delta y^2} - \frac{1}{v^2}\right) = 0$$

We get simple quadratic equation with easy solution:

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}.$$

As final solution we take:

$$T_{i,j,k} = \max(x_1, x_2)$$

This equation is solved at least once for every grid cell in the model.

AGU FALL MEETING
San Francisco | 14–18 December 2015
S33C: New Developments in Open Source and Academic Seismic Software Posters

Traveltime calculation step by step

This schematic shows how traveltime is being propagated through the model. Each grid cell is assigned in to one of three states: accepted (green), narrow band (yellow) and far (white). At the beginning one grid cell is accepted – the source with time 0.0s. Every time grid cell is accepted all close neighbors get their travel time calculated (6 neighbors in 3D, 4 in 2D) and marked as narrow band. Grid cell might have its time recalculated if it has been previously marked as narrow band. After all neighbors have been calculated the cell with minimal time is accepted and process continues.

The simplest source code example

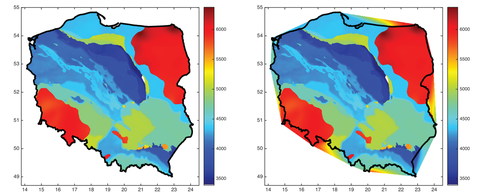
```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <deque>
#include <vector>
#include <algorithm>
#include <ctime>
#include <queue>

#define pi 3.14159265358979323846

int size_x = 536;
int size_y = 631;
int size_z = 6261;

int sub2ind(int y, int x, int z)
{
        return y + (x)*size_y + (z)*size_x*size_y;
}

void ind2sub(int n, int dy, int dx, int dz)
{
        y = (n - 1) % size_y + 1;
        m = (n - y)/size_y +1;

        x = (m - 1) % size_x + 1;
        m = (m - x)/size_x +1;

        z = (m - 1) % size_z + 0;
        z--;
}

struct NB
{
        int index;
        float time;

        NB(int a, float b) : index(a), time(b) {}
};

struct CompareTime
{
        inline bool operator() (const NB& n1, const NB& n2)
        {
                return(n1.time > n2.time);
        }
};

struct Neighbour
{
        double a;
        double b;
        double c;

        Neighbour(double a, double b, double c) : a(a), b(b), c(c) {}
};

double deg2rad(double deg)
{
        return deg * (pi / 180);
}

void ReadData(double *lat, double *lon, double *H, float *MODELVFP, unsigned char *MODEL)
{
        FILE *fptr;
        const clock_t begin_time = clock();

        std::cout << "Reading H\\t\\t";
        fptr = fopen("H.bin", "rb");
        fread(H, size_z*sizeof(double),1,fptr);
        std::cout << "done\n";

        std::cout << "Reading lat\\t\\t";
        fptr = fopen("lat.bin", "rb");
        fread(lat, size_y*sizeof(double),1,fptr);
        std::cout << "done\n";

        std::cout << "Reading lon\\t\\t";
        fptr = fopen("lon.bin", "rb");
        fread(lon, size_x*sizeof(double),1,fptr);
        std::cout << "done\n";

        std::cout << "Reading MODEL\\t\\";
        fptr = fopen("MODEL.bin", "rb");
        fread(MODEL, size_x*size_y*size_z*sizeof(unsigned char),1,fptr);
        std::cout << "done\n";

        std::cout << "Reading MODELVFP\\";
        fptr = fopen("MODELVFP.bin", "rb");
        fread(MODELVFP, size_x*size_y*size_z*sizeof(float),1,fptr);
        std::cout << "done\n";

        std::cout << "Reading completated in " << float( clock () - begin_time ) /
        ↪  CLOCKS_PER_SEC << "\n";

}

Neighbour GetNeighbours(int y, int x, int z, int direction, float *time, bool *accepted, double
↪ *lat, double *H)
{
        int diry = 0;
        int dirx = 0;
        int dirz = 0;


        double dy = deg2rad(0.01);
        double dx = deg2rad(0.02);
        double dz = 10.0;
        double r = 6371000 + H[z];

        double element = 0;

        if(direction == 1)
        {
                diry = 1;
                element = r*dy;
        }
        else if (direction == 2)
        {
                dirx = 1;
                element = r*cos(deg2rad(lat[y]))*dx;
        }
        else if ( direction == 3)
        {
                dirz = 1;
                element = dz;
        }

        double up = 0;
        if(accepted[sub2ind(y-diry,x-dirx,z-dirz)] == 1 &&
        ↪  accepted[sub2ind(y-diry,x-dirx,z-dirz)] == 1)
        {
                if(time[sub2ind(y-diry,x-dirx,z-dirz)] <= time[sub2ind(y-diry,x-dirx,z-dirz)])
                {
                        up = time[sub2ind(y-diry,x-dirx,z-dirz)];
                }
                else
                {
                        up = time[sub2ind(y-diry,x-dirx,z-dirz)];
                }
        }
        else if(accepted[sub2ind(y-diry,x-dirx,z-dirz)] == 1 &&
        ↪  accepted[sub2ind(y-diry,x-dirx,z-dirz)] == 0)
        {
                up = time[sub2ind(y-diry,x-dirx,z-dirz)];
        }
        else if(accepted[sub2ind(y-diry,x-dirx,z-dirz)] == 0 &&
        ↪  accepted[sub2ind(y-diry,x-dirx,z-dirz)] == 1)
        {
                up = time[sub2ind(y-diry,x-dirx,z-dirz)];
        }

        double a = 1 / pow(element,2);
        double b = (-2*up) / pow(element,2);
        double c = pow(up,2) / pow(element,2);

        if(up > 0)
        {
                return Neighbour(a,b,c);
        }
        else
        {
                return Neighbour(0,0,0);
        }

}
```

This listing presents the simplest implementation of traveltime calculation using fast marching method. This 300-line code is fully functional and very efficient. It's limited to only first approximation finite difference coefficient and has all parameters hard coded. Full code has about 1000 lines and will be published soon.

```
void calculate(int y, int x, int z, double *lat, double *lon, double *H, float *time, bool
↪  *accepted, float *MODELVFP, std::priority_queue<NB, std::vector<NB>, CompareTime>
↪  /narrow_band)
{
        double dy, dx, dz;
        double r;
        double u = NAN;
        double u1, u2;

        if(accepted[sub2ind(y, x, z)] == 0 && MODELVFP[sub2ind(y,x,z)] > 0)
        {
                if(y > 0 && y < size_y && x > 0 && x < size_x && z > 0 && z < size_z )
                {
                        dy = deg2rad(0.01);
                        dx = deg2rad(0.02);
                        dz = 10.0;
                        r = 6371000 + H[z];

                        Neighbour ny = GetNeighbours(y, x, z, 1, time, accepted, lat, H);
                        Neighbour nx = GetNeighbours(y, x, z, 2, time, accepted, lat, H);
                        Neighbour nz = GetNeighbours(y, x, z, 3, time, accepted, lat, H);


                        double a = ny.a + nx.a + nz.a;
                        double b = ny.b + nx.b + nz.b;
                        double c = ny.c + nx.c + nz.c - 1/pow(MODELVFP[sub2ind(y,x,z)],2);

                        double delta = sqrt(pow(b,2) - 4*a*c);
                        u1 = (-b - delta) / (2*a);
                        u2 = (-b + delta) / (2*a);
                        u = std::max(u1,u2);

                        if (time[sub2ind(y,x,z)] > u || time[sub2ind(y,x,z)] == 0)
                        {
                                time[sub2ind(y,x,z)] = u;
                                narrow_band.push(NB(sub2ind(y,x,z),u));
                        }
                }
        }
}

int main(int argc, const char * argv[])
{
        double *H = (double*) malloc(size_z*sizeof(double));
        double *lon = (double*) malloc(size_x*sizeof(double));
        double *lat = (double*) malloc(size_y*sizeof(double));
        unsigned char *MODEL = (unsigned char*) malloc(size_x*size_y*size_z*sizeof(unsigned
        ↪  char));
        float *MODELVFP = (float*) malloc(size_x*size_y*size_z*sizeof(float));
        bool *accepted = (bool*) malloc(size_x*size_y*size_z*sizeof(bool));
        float *time = (float*) malloc(size_x*size_y*size_z*sizeof(float));
        std::priority_queue<NB, std::vector<NB>, CompareTime> narrow_band;

        int y, x, z;

        ReadData(lat, lon, H, MODELVFP, MODEL);

        int total_to_calculate = 0;
        for(int i = 0; i < size_y*size_x*size_z; i++)
        {
                if(MODELVFP[i] > 0)
                {
                        total_to_calculate++;
                }
        }

        // Source location
        z = 245                 - 1;
        y = 539                 - 1;
        x = 193                 - 1;

        time[sub2ind(y, x, z)] = 10e-10;
        accepted[sub2ind(y, x, z)] = 1;

        clock_t begin_time = clock();
        clock_t part_time = clock();
        int number_off_accepted = 0;
        float last_time = 0;
        double goal = total_to_calculate;
        double last_compute_time = 0;
        while(number_off_accepted < goal - 1)
        {
                if(((float( clock () - part_time ) /  CLOCKS_PER_SEC) > 1) ||
                ↪  ((number_off_accepted/goal) > 0.9999))
                {
                        last_compute_time =float( clock () - part_time ) /  CLOCKS_PER_SEC;
                        double time_to_go = 0;
                        if(last_compute_time > 0)
                        {
                                time_to_go = ((goal - number_off_accepted) /100000)
                                ↪  *last_compute_time;
                        }
                        printf("completated: %3.0f (%3.2f%%)\tpart time: %3.5f\ttotal time:
                        ↪  %09.5f\testimated time to go: %06.2f(%8b) %08d\n", last_time,
                        ↪  ((double)number_off_accepted/(double)goal)*100.0,
                        ↪  last_compute_time, float( clock () - begin_time ) / CLOCKS_PER_SEC,
                        ↪  time_to_go, int(narrow_band.size()));
                        part_time = clock();
                }
                calculate(y + 1, x, z, lat, lon, H, time, accepted, MODELVFP, narrow_band);
                calculate(y - 1, x, z, lat, lon, H, time, accepted, MODELVFP, narrow_band);
                calculate(y, x + 1, z, lat, lon, H, time, accepted, MODELVFP, narrow_band);
                calculate(y, x - 1, z, lat, lon, H, time, accepted, MODELVFP, narrow_band);
                calculate(y, x, z + 1, lat, lon, H, time, accepted, MODELVFP, narrow_band);
                calculate(y, x, z - 1, lat, lon, H, time, accepted, MODELVFP, narrow_band);

                if(narrow_band.size() == 1)
                {
                        break;
                }


                while(time[narrow_band.top().index] != narrow_band.top().time ||
                ↪  accepted[narrow_band.top().index]==1)
                {
                        narrow_band.pop();
                }

                accepted[narrow_band.top().index] = 1;
                number_off_accepted++;
                time[narrow_band.top().index] = narrow_band.top().time;
                last_time = narrow_band.top().time;
                ind2sub(narrow_band.top().index, y, x, z);
                narrow_band.pop();
        }

        for(int i = 1; i < size_x*size_y*size_z; i++)
        {
                if(accepted[i] == 0)
                {
                        time[i] = 0;
                }
        }


        FILE *fptr;
        fptr = fopen("/Volumes/RAM/time_5000.bin", "wb");
        fwrite(time, size_x*size_y*size_z*sizeof(float),1,fptr);

        free(H);
        free(lon);
        free(lat);
        free(MODEL);
        free(MODELVFP);
        free(accepted);
        free(time);


        return 0;
}
```

Our 3D model

This implementation was prepared to work with 3D P-wave velocity model of Poland (Grad et. al 2015). The model covers area of Poland from topography to 60 km deep. Total grid size is 631 x 536 x 6261 (2.117.570.376) cells. In 1.390.642.955 P-wave velocity is provided. To allow simpulation model has to be converted to convex shape.



Code efficiency

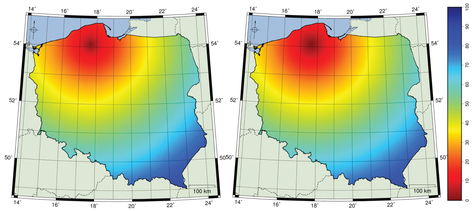
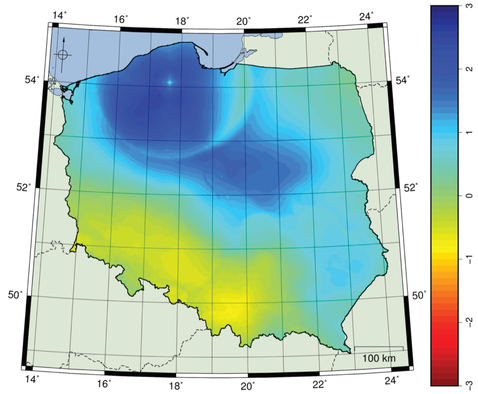
Single simulation provides traveltime to all grid cells in the model for given source location. Source can be located in one or more grid cells with proper time. For example, an interface can be assumed as source for reflected wave.

Single simulation requires about 20GB of operating memory and takes 2 hours to complete. By average about 200.000 grid cells are calculated every second on modern workstation.

This result can still be improved depending on expected result. Unfortunately, due to algorithm construction it is not possible to migrate to GPU.

Example result

The code was used to calculate traveltime from source located on surface in northern Poland (location of one of our temporary broadband stations). Travel time to all grid cells was calculated and time at surface is presented on left map below. Right map shows traveltime from the same source calculated using 1D IASP reference model.



Both map look very similar (this tells that the code is working), however their mathematical difference show influence of 3D structures: