Purpose of this exercise: Eikonal solver design

Solving the non-linear Eikonal equation in a 2D Cartesian medium [xmin,xmax]x[zmin,zmax] by the fast sweeping method which perform a Gauss-Seidel iteration (find on the web how this iterative method for solving non-linear system F(x)=0) along alternating directions called sweeps

$$\left[\frac{\partial T(x,z)}{\partial x}\right]^2 + \left[\frac{\partial T(x,z)}{\partial z}\right]^2 = \frac{1}{c^2(x,z)} = u^2(x,z)$$

with fixed values at specific points on boundaries (essentially the punctual source in our case). There is a need of global natural ordering because the new solution at one point will depend on the new solution at the previous point in the ordering and not on the previous solution. This is an efficient technique when the numerical information is transported along directions.

Schematic sketch of the fast sweeping method. The discretization of partial derivatives will be an upwind scheme and the ordering will be natural on a Cartesian grid.

Therefore, the fast sweeping methods can be summarized as follows

1. Initialization:

   Define a regular Cartesian grid (nx, nz) such that the derivatives are accurately described by a finite difference expression. Impose the exact value of $T(i,j)$ at grid points on or near boundaries, for which values are fixed during iterations; assign arbitrary high values at all other points (in our case, for a point source)

2. Gauss-Seidel iterations: sweep the whole mesh following alternating orderings:

   At each point of the grid (nx, nz), update $T^k(i,j)$ with an upwind scheme $Upwind(T^{k-1}(:,:))$, often called the local solver because only local values will be used leading to a sparse stencil. One iteration corresponds to sweep all grid points once in four directions (x positive, x negative, z positive, z negative).

3. Termination:

Stop the iterations if two successive iterations have small changes, i.e. $|T^k(i,j) - T^{k-1}(i,j)| < \delta$ from a threshold $\delta > 0$.

Useful approach when many rays should be traced in 2D or 3D media: a uniform sampling of values is expected on the grid, thanks to the Eulerian approach.

References: Han et al, 2017. Calculating qP-wave traveltimes in 2D TTI media by high-order fast sweeping methods with a numerical quartic equation solver, Geophys. J. Int., 210, 1560-1569.

Luo, S. and J. Qian, 2012. Fast sweeping methods for factored anisotropic eikonal equations: multiplicative and additive factors, J. Sci. Comput., 53, 360-382.

Basically, two ingredients have to be designed: the sweeping strategy and the local solver. First, let us consider the sweeping strategy and then the local solver.

Sweeping strategy (should sample all directions of propagation)

```
!~~~~~~~~~~~~~~>>>>>>>>>>>>the loop of "Fast Sweeping Method (FSM)"
Max_fsm=Far_time                    ! high value for starting comparison between solutions
it_fsm=0
!============= LOOP ITERATION (until convergence or after a given number of iterations
 DO WHILE (Max_fsm >= fsm_stp .and. it_fsm < Max_iter)   ! always two criteria for exit -
   it_fsm=it_fsm + 1                ! increment the iteration count
   write(*,*) 'iter ',it_fsm,Max_iter,Max_fsm
   TT_old_s=TT_s                    ! save the previous solution (high values at the beginning)
   !~~~~~~~~~~~~~~~~~~~~~~~~>>>>>>>>>>>> Sweep 1
   DO i2=1,n2                        ! along z the outer direction
     DO i1=1,n1                      ! along x the inner direction
       CALL local_solver(neig_ind,i1,i2,n1,n2,xh1_s,xh2_s,TT_s,vel_p_s,Taprx_s)
       TT_s(i1,i2)=DMIN1(TT_s(i1,i2),Taprx_s)  ! always keep the smallest between new/old values
     ENDDO
   END DO
  !~~~~~~~~~~~~~~~~~~~~~~~~>>>>>>>>>>>> Sweep 2
   DO i2=1,n2
     DO i1=n1,1,-1
       CALL local_solver(neig_ind,i1,i2,n1,n2,xh1_s,xh2_s,TT_s,vel_p_s,Taprx_s)
       TT_s(i1,i2)=DMIN1(TT_s(i1,i2),Taprx_s)
     END DO
   END DO
   !~~~~~~~~~~~~~~~~~~~~~~~>>>>>>>>>>>> Sweep 3
   DO i2=n2,1,-1
     DO i1=n1,1,-1
       CALL local_solver(neig_ind,i1,i2,n1,n2,xh1_s,xh2_s,TT_s,vel_p_s,Taprx_s)
       TT_s(i1,i2)=DMIN1(TT_s(i1,i2),Taprx_s)
     END DO
   END DO
   !~~~~~~~~~~~~~~~~~~~~~~~>>>>>>>>>>>> Sweep 4
   DO i2=n2,1,-1
     DO i1=1,n1
       CALL local_solver(neig_ind,i1,i2,n1,n2,xh1_s,xh2_s,TT_s,vel_p_s,Taprx_s)
       TT_s(i1,i2)=DMIN1(TT_s(i1,i2),Taprx_s)
!======================== ##################################    convergence ...
         Max_fsm = DMAX1(Max_fsm, dabs(dble(TT_old_s(i1,i2)-TT_s(i1,i2))))
         !======================== #####" to be included in the last sweep for testing
convergence
     END DO
    ENDDO
   !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>>>>>>>>>>>>>>>>> End of FSM loop
 END DO                              ! back to the while loop Max_fsm small: convergence
```

Local solver (should be efficient but still more complex than those for wave equation)

How to estimate the travel time at a given point C from neighboring points: a stencil should be designed.

We shall consider the following configuration with eight triangles (figure 1). For each triangle, we shall consider points A and B and possible already known values of travel times at these points.

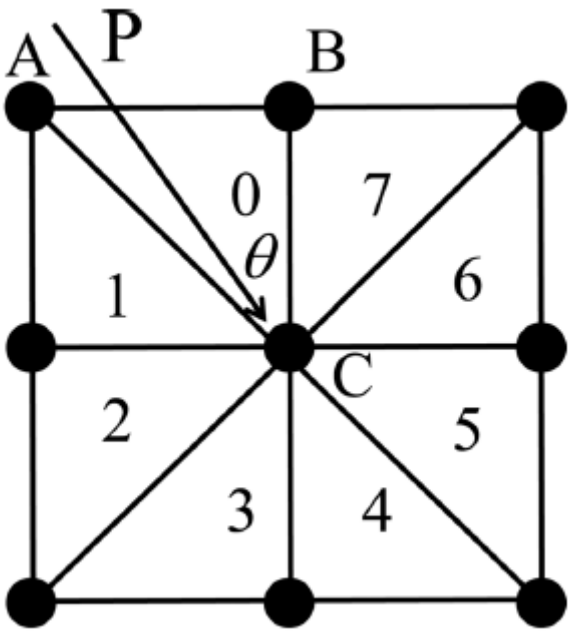

Figure 1: Eight-triangles stencil with first-order finite-difference approximations. P is the direction of the expected propagation arriving at the point C.

We shall take the smallest value of all travel times computed for the eight triangles. This will be the FD stencil we shall consider: it is a MIN operator.

Therefore, we have to consider a triangle and build the algorithm for estimating the expected travel time T_c at the point C.

Let us consider the slowness vector $\overrightarrow{p^c}$ at the point C. We have the following relation

$$\overrightarrow{AC}.\overrightarrow{p^c} = (x_C - x_A).p_x^c + (z_C - z_A).p_z^c$$

$$\overrightarrow{BC}.\overrightarrow{p^c} = (x_C - x_B).p_x^c + (z_C - z_B).p_z^c$$

We assume the finite difference approximation $T_A + \overrightarrow{AC}.\overrightarrow{p^c} \approx T_C$ and $T_B + \overrightarrow{BC}.\overrightarrow{p^c} \approx T_C$. We can write the matrix formulation

$$\begin{bmatrix} \overrightarrow{AC}.\overrightarrow{p^c} \\ \overrightarrow{BC}.\overrightarrow{p^c} \end{bmatrix} = \begin{bmatrix} x_C - x_A & z_C - z_A \\ x_C - x_B & z_C - z_B \end{bmatrix} \begin{bmatrix} p_x^c \\ p_z^c \end{bmatrix}$$

or the following approximation

$$\begin{bmatrix} T_C - T_A \\ T_C - T_B \end{bmatrix} = \begin{bmatrix} x_C - x_A & z_C - z_A \\ x_C - x_B & z_C - z_B \end{bmatrix} \begin{bmatrix} p_x^c \\ p_z^c \end{bmatrix}$$

$$\begin{bmatrix} T_C - T_A \\ T_C - T_B \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} p_x^c \\ p_z^c \end{bmatrix}$$

We can invert the matrix

$$\begin{bmatrix} p_x^c \\ p_z^c \end{bmatrix} = \begin{bmatrix} a' & b' \\ c' & d' \end{bmatrix} \begin{bmatrix} T_C - T_A \\ T_C - T_B \end{bmatrix} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \begin{bmatrix} T_C - T_A \\ T_C - T_B \end{bmatrix}$$

We can express the slowness vector with the unknown travel time T_C with

$$\begin{bmatrix} p_x^c \\ p_z^c \end{bmatrix} = \begin{bmatrix} MT_C + N \\ PT_C + Q \end{bmatrix}$$

with $M = a' + b'; N = -a'T_A - b'T_B; P = c' + d'; N = -c'T_A - d'T_B$. The modulus of the slowness vector should be equal to the inverse of the velocity or the slowness. We find the quadratic equation for T_C

$$(M^2 + P^2)T_C^2 + 2(MN + PQ)T_C + (N^2 + Q^2 - u^2(x_C, z_C)) = 0$$

which should be solved. We have to select the smallest real root (if there is any), such that the slowness vector is inside the cone ACB . If not, we select the smallest value between solutions assuming that the slowness is parallel to $\overrightarrow{AC}$ or $\overrightarrow{BC}$ (this related linear relation guarantees a solution which is known as the viscous solution).

Algorithm inside a triangle

```
!=================================================================

  ! triangle A-C-B
  ! A (xa,za) B(xb,zb)  C(xc,zc)
  ! one of elementary stencil
  !======================== linear expansion of (px,pz) wrt Tc
!=================================================================
  mat_a=xc-xa; mat_b=zc-za
  mat_c=xc-xb; mat_d=zc-zb
!     matrice composing the slowness contribution
!    (                              )
!    ( (xc-xa)       (zc-za)        )
!    ( ((xc-xb)      (zc-zb)        )
!    (                              )
!
!======================= get the inverse of the matrix 2x2
det_inv=1.0d0/(mat_a*mat_d-mat_b*mat_c)!
tampon=mat_a                                !permutation
mat_a=mat_d*det_inv                         !   1    (d   -b)   (a'  b')
mat_d=tampon*det_inv                        ! ----   (      ) = (      )
mat_c=-mat_c*det_inv                        !(ad-bc) (-c   a)   (c'  d')
mat_d=-mat_d*det_inv                        !

! get expression of Tc     px=M Tc + N
!                          pz=P Tc + Q
!                      to be found
!
!   a'(Tc-Ta) + b'(Tc-Tb) = M Tc + N
!   c'(Tc-Ta) + a'(Tc-Tb) = P Tc + Q
!
!   M = a'+b'     N = -a'*Ta -b'*Tb   (1st order)
!   P = c'+d'     Q = -c'*Ta -d'*Tb
!
```

```
!@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ first-order stencil
     mat_m=mat_a+mat_b
     mat_n=-mat_a*Time_A-mat_b*Time_B
     mat_p=mat_d+mat_c
     mat_q=-mat_c*Time_A-mat_d*Time_B
!========================== quadratic equation
!
!(M*M+P*P) Tc*Tc + 2 (MN+PQ) Tc +N*N+Q*Q-u*u=0  eikonal
!
! transformed into
!
!  Tc*Tc + q1 Tc + q0 = 0
!
!==========================
  up=1.0d00/vpc          ! phase slowness
  up2=up*up              ! square of the phase slowness
  racine(:,:)=0.0D0      ! set roots equal to zero (real,imag)

!============ quadratic
  q1=2.0*(mat_m*mat_n + mat_p*mat_q)/(mat_m*mat_m + mat_p*mat_p)
  q0=(mat_n*mat_n + mat_q*mat_q-up2)/(mat_m*mat_m + mat_p*mat_p)
  call quadraticRoots (q1, q0, nReal, racine)  ! TOMS 954 algorithm
  n_d=2
!========================== scan solution if any
  Time_C_OK=1.d0/eps   ! set to high values
  iopt=0
  do i=1,n_d           ! only looking at real roots
!============== consider only real roots but not those equal to zero
  if(racine(i,2) < eps  .and. racine(i,1) > eps ) then
    Time_C=racine(i,1)   ! maybe the solution
!========================== check the causality
!================= compute the ray velocity (group velocity) tangent
!================= to the ray (not normal to waveform which defines
!================= the phase velocity)
    px_c=mat_m*Time_C+mat_n
    pz_c=mat_p*Time_C+mat_q
    pinv=1.0d0/dsqrt(px_c*px_c+pz_c*pz_c)
    xnx_c=px_c*pinv ! get the unit vector
    xnz_c=pz_c*pinv ! get the unit vector
    dispersion=up*pinv - 1.0d0   ! dispersion p=slowness
    if(dabs(dispersion) < 0.000001) then ! P phase slowness
!========================== check the causality
!========================== compute the ray velocity
!============= to the ray (not normal to waveform // to slowness p)
!==================================
!   the point defined by x=xc+tc*pxc should belong to the line (AB)
!                          z=zc+tc*pzc
!equation of line (AB)    (x-xa)*(zb-za)-(z-za)*(xb-xa)=0 and deduce
! tc [(zb-za)*pxc-(xb-xa)*pzc]=(zc-za)*(xb-xa)-(xc-xa)*(zb-za)
!==================================
 tc=((xb-xa)*zc-(zb-za)*xc+zb*xa-xb*za)/((zb-za)*px_c-(xb-xa)*pz_c)
! parameter on the line (c,p_c)
      if(tc < 0.0d0) then ! pointing in the right direction toward C
!  should increase and, therefore, we have to look backward from C
!                            toward A and B
        if(dabs(xa-xb) < eps) then
```

```
          tau=(zc-za+tc*pz_c)/(zb-za) !(zb-za) should be # from zero
        else
          tau=(xc-xa+tc*px_c)/(xb-xa)
        endif
!==================== inside the causality cone (between A and B)
        if(tau <= 1.0d00 .and. tau >=0.d00) then
          Time_C_OK=dmin1(Time_C,Time_C_OK) ! always the smallest
          iopt=1
        endif
      endif ! correct direction
    endif ! end of the slowness surface P
  endif ! end over real solutions
enddo ! end of the loop over the four solutions
!================================
! get the final value which should exist in any case
!================================
if(iopt > 0) then
  Time_C=1.0d12
  do i=1,iopt
    Time_C=dmin1(Time_C,Time_C_OK) ! take the causal smallest value
  enddo
  return
else
!================ should get the viscous solution (two options)
!============= from A   @@@@@@@@@@@@@@@@@@@@@@@@@@
  xnx=(xc-xa)
  xnz=(zc-za)
  d_ac=dsqrt(xnx*xnx+xnz*xnz)
  Time_AC=Time_A+d_ac*up ! time as if coming from A
!============= from B   @@@@@@@@@@@@@@@@@@@@@@@@@@
  xnx=(xc-xb)
  xnz=(zc-zb)
  d_bc=dsqrt(xnx*xnx+xnz*xnz)
  Time_BC=Time_B+d_bc*up   ! time as if coming from B
!============= set the minimum time so we always get one solution
  Time_C=dmin1(Time_AC,Time_BC)
endif  ! end of the search for causal or viscous solution
```

End of the algorithm inside the triangle.

The local solver will scan the eight triangles and will get the smallest travel time, which will provide the Cartesian stencil based on eight points to estimate the value on the ninth one.

Do it yourself:

Inside the SRC_ISOTROPE, you will find

Subroutines to perform these algorithms

stencil_triangle.f90: the search of the solution inside a triangle

local_solver.f90: the Cartesian FD stencil

eikonal_solver.f90: the sweeping strategy over the grid

Additional subroutines

Polynomial2RootSolvers.f90: the TOMS 954 algorithm for quadratic equation.

neighbors.f90: the definition of the local stencil

read_inputs.f90: the reading of parameters

inside the file “inputs”, here are values

```
n1, n2             ! "number of grid points on z and x: n1 n2"
xh1_s,xh2_s        ! "grid spacing "
max_iter           ! "number of maximum iterations "
x1_src, x2_src     ! "source position in meter in direction 1 , 2"
homo_1500.bin.h50 ! “velocity file in binary format”
ioptio,vhomo       ! “option for homo. vel.(0) and possible value”
```

inside the velocity file “homo_1500.bin.h50” (for example), one should find the n1*n2 binary values of the velocity

EIKONAL.f90: the program for computing the travel time map for one source.

Compiling:

Type following commands

Type “cd SRC_ISOTROPIC” (to go into this source directory)

Type “make clean” (to clean the directory in case)

Type “make” (assuming that you have the gfortran compiler available on your computer and the utility make)

It will put binaries into a directory ../BIN

(if you have not the utility “make”, please type “sh compile.sh”, but a good advice will come with the installation of this utility).

Type “cd ..” (to go back to the project directory)

Running:

Copy the directory RUN_ISOTROPIC_TEMPLATE into RUN_ISOTROPIC

(type the command “cp –r RUN_ISOTROPIC_TEMPLATE RUN_ISOTROPIC”)

Move into the new directory

(type the command “cd RUN_ISOTROPIC”)

Launch by ‘sh run_homo.sh’ which will compute solution on a 501 x 501 grid: see what you get as new files, especially those with an extension jpg obtained by using gnuplot into the shell file plot_gnu_homo.sh. Arguments are needed (-h option will provide an hint)

Launch by ‘sh run_bp.sh’ which will compute solution on a 126 x 626 grid: see what you get following the same workflow as the previous one using now plot_gnu_bp.sh.

Analysis:

Please point out what you find … meaning, interesting, unexpected …