```julia
using GSL
```

```julia
using MultipleScattering
```

```julia
using Plots
```

```julia
using PyPlot
```

PyPlotBackend()

```julia
pyplot()
```

```julia
using LinearAlgebra
```

```julia
using DelimitedFiles
```

```julia
using Base.Threads
```

```julia
hk(n::Integer,x::Float64)=sf_bessel_jl(n,x)+im*sf_bessel_yl(n,x);
```

ymn (generic function with 1 method)

```julia
function ymn(n::Integer,m::Integer,θ::Float64,ϕ::Float64)   #spherical harmonics
if m>=0
    return sf_legendre_sphPlm(n,m,cos(θ))*exp(im*m*ϕ);
elseif isodd(-m)
    return -sf_legendre_sphPlm(n,-m,cos(θ))*exp(im*m*ϕ);
else
    return sf_legendre_sphPlm(n,-m,cos(θ))*exp(im*m*ϕ);
end
end
```

Selection deleted

```
buildModelProto (generic function with 1 method)
```

```julia
#########################build simulation model###############################
function buildModelProto(dimensionTmp::Integer, ρbTmp::Float64, cbTmp::Float64, ρpTmp::Float64, cpTmp::Float64, ωbTmp::Float64, pNoTmp::Integer, RsTmp::Float64, parPos::Matrix{Float64})
parPosTmp=deepcopy(parPos);
#####################set incident waves#####################################
incAmpTmp = 3000.0; #amplitude of incident beam
incDirTmp1 = [0.0, 0.0, 1.0];   #set incident direction 1
incDirTmp2 = [0.0, 0.0, -1.0];  #set incident direction 2
incPosTmp1 = [0.0, 0.0, -1];    #original position of incident wave 1
incPosTmp2 = [0.0, 0.0, 1]; #original position of incident wave 2
bgMediumTmp = Acoustic(dimensionTmp; ρ = ρbTmp, c = cbTmp); #build background acoustic model
waveTmp = plane_source(bgMediumTmp; amplitude = incAmpTmp, direction = incDirTmp1, position = incPosTmp1)+plane_source(bgMediumTmp; amplitude = -incAmpTmp, direction = incDirTmp2, position = incPosTmp2); #build incident plane wave
#########################incident wave done#######################################
########################set particles############################################
parMediumTmp = Acoustic(dimensionTmp; ρ = ρpTmp, c = cpTmp);    #build the acoustic model in particles
particlesTmp=Array{Particle{dimensionTmp, Acoustic{Float64, dimensionTmp}, Sphere{Float64, dimensionTmp}}}(undef, 0);  #define a null array to store particles model
#build particle set
for iTmp in 1:pNoTmp
    parShapeTmp=Sphere(parPosTmp[iTmp,:],RsTmp);
    particlesTmp=push!(particlesTmp,Particle(parMediumTmp,parShapeTmp));
end
#########################particles done########################################
simModelTmp=FrequencySimulation(particlesTmp,waveTmp);#build simulation model
return simModelTmp
end
```

```
getCoefProto (generic function with 1 method)
```

```julia
#######################get expansion coefficients########################
function getCoefProto(ωbTmp::Float64, modelTmp::FrequencySimulation, coefOrderTmp::Integer)
simModelTmp=modelTmp;
coefDataTmp=basis_coefficients(simModelTmp,ωbTmp,basis_order=coefOrderTmp);#store the expansion coefficients
return coefDataTmp
end
```

Selection deleted

```
pProto (generic function with 1 method)
```

```julia
##########define function to calculate pressure in position [x,y,z]###########
function pProto(x::Float64, y::Float64, z::Float64, ωbTmp::Float64, modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
simModelTmp=modelTmp;
pNoTmp=length(simModelTmp.particles);
dimensionTmp=typeof(simModelTmp.source.medium).parameters[2];
cbTmp=Float64(simModelTmp.source.medium.c);     #make soundspeed a real number
coefOrderTmp=Int(sqrt(length(coefDataTmp[:,1])))-1;
parPosTmp=Matrix{Float64}(undef,pNoTmp,dimensionTmp);
k=ωbTmp/cbTmp;
for iTmp in 1:pNoTmp
    for jTmp in 1:dimensionTmp
        parPosTmp[iTmp,jTmp]=simModelTmp.particles[iTmp].shape.origin[jTmp];
    end
end
pField=0.0+0.0*im;
r=Array{Float64}(undef,pNoTmp);
θ=Array{Float64}(undef,pNoTmp);
ϕ=Array{Float64}(undef,pNoTmp);
for iTmp in 1:pNoTmp
    parPosTmp2=deepcopy(parPosTmp[iTmp,:]);
    xx=x-parPosTmp2[1];
    yy=y-parPosTmp2[2];
    zz=z-parPosTmp2[3];
    r[iTmp]=sqrt(xx*xx+yy*yy+zz*zz);
    θ[iTmp]=acos(zz/max(r[iTmp],0.000000001));
    if yy==0&&xx>0
        ϕ[iTmp]=0.0;
    elseif yy==0&&xx<0
        ϕ[iTmp]=π;
    else
        ϕ[iTmp]=(1-sign(yy))*π+sign(yy)*acos(xx/max(sqrt(xx*xx+yy*yy),0.00000001));
    end
    for nTmp in 0:coefOrderTmp
        for mTmp in -nTmp:nTmp
            pField+=coefDataTmp[nTmp*nTmp+nTmp+mTmp+1,iTmp]*hk(nTmp,k*r[iTmp])*ymn(nTmp,mTmp,θ[iTmp],ϕ[iTmp]);
        end
    end
end
return pField+simModelTmp.source.field([x,y,z],ωbTmp);
end
```

Selection deleted

```
vProto (generic function with 1 method)
```

```
###########Calculate velocity field by using five-point stencil#############
function vProto(x::Float64, y::Float64, z::Float64, ωbTmp::Float64,
modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
simModelTmp=modelTmp;
dimensionTmp=typeof(simModelTmp.source.medium).parameters[2];
δh=0.00005;
ρbTmp=simModelTmp.source.medium.ρ;
vField=Array{ComplexF64}(undef,dimensionTmp);
pX2=pProto(x+2*δh,y,z,ωbTmp,simModelTmp,coefDataTmp);
pX1=pProto(x+δh,y,z,ωbTmp,simModelTmp,coefDataTmp);
pXN2=pProto(x-2*δh,y,z,ωbTmp,simModelTmp,coefDataTmp);  #N denotes negative
pXN1=pProto(x-δh,y,z,ωbTmp,simModelTmp,coefDataTmp);
pY2=pProto(x,y+2*δh,z,ωbTmp,simModelTmp,coefDataTmp);
pY1=pProto(x,y+δh,z,ωbTmp,simModelTmp,coefDataTmp);
pYN2=pProto(x,y-2*δh,z,ωbTmp,simModelTmp,coefDataTmp);
pYN1=pProto(x,y-δh,z,ωbTmp,simModelTmp,coefDataTmp);
pZ2=pProto(x,y,z+2*δh,ωbTmp,simModelTmp,coefDataTmp);
pZ1=pProto(x,y,z+δh,ωbTmp,simModelTmp,coefDataTmp);
pZN2=pProto(x,y,z-2*δh,ωbTmp,simModelTmp,coefDataTmp);
pZN1=pProto(x,y,z-δh,ωbTmp,simModelTmp,coefDataTmp);
vField[1]=-im/ρbTmp/ωbTmp*(-pX2+8*pX1-8*pXN1+pXN2)/12/δh;
vField[2]=-im/ρbTmp/ωbTmp*(-pY2+8*pY1-8*pYN1+pYN2)/12/δh;
vField[3]=-im/ρbTmp/ωbTmp*(-pZ2+8*pZ1-8*pZN1+pZN2)/12/δh;
return vField
end
```

Selection deleted

T (generic function with 1 method)

```
###########calculate time average of stress tensor################
function T(x::Float64, y::Float64, z::Float64, ωbTmp::Float64,
modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
simModelTmp=modelTmp;
dimensionTmp=typeof(simModelTmp.source.medium).parameters[2];
TDataTmp=Matrix{Float64}(undef,dimensionTmp,dimensionTmp);  #matrix to store stress
tensor in the form:
##############################
######### Txx Txy Txz ########
######### Tyx Tyy Tyz ########
######### Tzx Tzy Tzz ########
##############################
ρbTmp=simModelTmp.source.medium.ρ;
cbTmp=simModelTmp.source.medium.c;
vTmp=vProto(x,y,z,ωbTmp,simModelTmp,coefDataTmp)
vxTmp=vTmp[1];
vyTmp=vTmp[2];
vzTmp=vTmp[3];
cjvxTmp=conj(vxTmp);
cjvyTmp=conj(vyTmp);
cjvzTmp=conj(vzTmp);
pTmp=pProto(x,y,z,ωbTmp,simModelTmp,coefDataTmp);
cjpTmp=conj(pTmp);
vSqu=real(vxTmp*cjvxTmp+vyTmp*cjvyTmp+vzTmp*cjvzTmp);
pSquCoeff=real(pTmp*cjpTmp/2/ρbTmp/cbTmp/cbTmp);
TDataTmp[1,1]=0.5*(ρbTmp*(real(vxTmp*cjvxTmp)-0.5*vSqu)+pSquCoeff);
TDataTmp[1,2]=0.5*ρbTmp*real(vxTmp*cjvyTmp);
TDataTmp[1,3]=0.5*ρbTmp*real(vxTmp*cjvzTmp);
TDataTmp[2,1]=0.5*ρbTmp*real(vyTmp*cjvxTmp);
TDataTmp[2,2]=0.5*(ρbTmp*(real(vyTmp*cjvyTmp)-0.5*vSqu)+pSquCoeff);
TDataTmp[2,3]=0.5*ρbTmp*real(vyTmp*cjvzTmp);
TDataTmp[3,1]=0.5*ρbTmp*real(vzTmp*cjvxTmp);
TDataTmp[3,2]=0.5*ρbTmp*real(vzTmp*cjvyTmp);
TDataTmp[3,3]=0.5*(ρbTmp*(real(vzTmp*cjvzTmp)-0.5*vSqu)+pSquCoeff);
return TDataTmp
end
```

Selection deleted

fDens (generic function with 1 method)

```
##################calculate force density#########################
function fDens(θ::Float64,ϕ::Float64,parID::Integer,ωbTmp::Float64,
modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
simModelTmp=modelTmp;
dimensionTmp=typeof(simModelTmp.source.medium).parameters[2];
fDensDataTmp=Array{Float64}(undef,dimensionTmp);    #[fDensx,fDensy,fDensz]
R=simModelTmp.particles[parID].shape.radius+0.00015;
x0=simModelTmp.particles[parID].shape.origin[1];#x0,y0,z0 denote particle's position
y0=simModelTmp.particles[parID].shape.origin[2];
z0=simModelTmp.particles[parID].shape.origin[3];
sθ=sin(θ);cθ=cos(θ);sϕ=sin(ϕ);cϕ=cos(ϕ);
xTmp=x0+R*sθ*cϕ;
yTmp=y0+R*sθ*sϕ;
zTmp=z0+R*cθ;
TData=T(xTmp,yTmp,zTmp,ωbTmp,simModelTmp,coefDataTmp);
fDensDataTmp[1]=(TData[1,1]*sθ*cϕ+TData[1,2]*sθ*sϕ+TData[1,3]*cθ)*R*R*sθ;
fDensDataTmp[2]=(TData[2,1]*sθ*cϕ+TData[2,2]*sθ*sϕ+TData[2,3]*cθ)*R*R*sθ;
fDensDataTmp[3]=(TData[3,1]*sθ*cϕ+TData[3,2]*sθ*sϕ+TData[3,3]*cθ)*R*R*sθ;
return fDensDataTmp
end
```

forcePart1 (generic function with 1 method)

```
###########function to execute part of the parallel computation##############
function
forcePart1(parID::Integer,n1::Integer,n2::Integer,lowLim1::Float64,upLim1::Float64,lo
wLim2::Float64,upLim2::Float64,ωbTmp::Float64, modelTmp::FrequencySimulation,
coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
fDataTmp=[0.0,0.0,0.0]############[fx,fy,fz]#######################
xDataStoreTmp=zeros(n1,n2);
yDataStoreTmp=zeros(n1,n2);
zDataStoreTmp=zeros(n1,n2);
step1=(upLim1-lowLim1)/n1;
step2=(upLim2-lowLim2)/n2;
for i in 1:n1
    θTmp=lowLim1+(i-1)*step1;
    for j in 1:n2
        ϕTmp=lowLim2+(j-1)*step2;
        fDensData=fDens(θTmp,ϕTmp,parID,ωbTmp,modelTmp,coefDataTmp);
        xDataStoreTmp[i,j]=fDensData[1];
        yDataStoreTmp[i,j]=fDensData[2];
        zDataStoreTmp[i,j]=fDensData[3];
```

Selection deleted

```
    end
end
fDataTmp[1]=-sum(xDataStoreTmp)*step1*step2;
fDataTmp[2]=-sum(yDataStoreTmp)*step1*step2;
fDataTmp[3]=-sum(zDataStoreTmp)*step1*step2;
return fDataTmp
end
```

```
forcePart2 (generic function with 1 method)
```

```julia
##########function to execute part of the parallel computation##############
function forcePart2(parID::Integer,n1::Integer,n2::Integer,lowLim1::Float64,upLim1::Float64,lowLim2::Float64,upLim2::Float64,ωbTmp::Float64, modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
fDataTmp=[0.0,0.0,0.0]###########[fx,fy,fz]######################
xDataStoreTmp=zeros(n1,n2);
yDataStoreTmp=zeros(n1,n2);
zDataStoreTmp=zeros(n1,n2);
step1=(upLim1-lowLim1)/n1;
step2=(upLim2-lowLim2)/n2;
for i in 1:n1
    θTmp=lowLim1+(i-1)*step1;
    for j in 1:n2
        ϕTmp=lowLim2+(j-1)*step2;
        fDensData=fDens(θTmp,ϕTmp,parID,ωbTmp,modelTmp,coefDataTmp);
        xDataStoreTmp[i,j]=fDensData[1];
        yDataStoreTmp[i,j]=fDensData[2];
        zDataStoreTmp[i,j]=fDensData[3];
    end
end
fDataTmp[1]=-sum(xDataStoreTmp)*step1*step2;
fDataTmp[2]=-sum(yDataStoreTmp)*step1*step2;
fDataTmp[3]=-sum(zDataStoreTmp)*step1*step2;
return fDataTmp
end
```

Selection deleted

forcePart3 (generic function with 1 method)

```
##########function to execute part of the parallel computation##############
function
forcePart3(parID::Integer,n1::Integer,n2::Integer,lowLim1::Float64,upLim1::Float64,lowLim2::Float64,upLim2::Float64,ωbTmp::Float64, modelTmp::FrequencySimulation,
coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
fDataTmp=[0.0,0.0,0.0]###########[fx,fy,fz]######################
xDataStoreTmp=zeros(n1,n2);
yDataStoreTmp=zeros(n1,n2);
zDataStoreTmp=zeros(n1,n2);
step1=(upLim1-lowLim1)/n1;
step2=(upLim2-lowLim2)/n2;
for i in 1:n1
    θTmp=lowLim1+(i-1)*step1;
    for j in 1:n2
        ϕTmp=lowLim2+(j-1)*step2;
        fDensData=fDens(θTmp,ϕTmp,parID,ωbTmp,modelTmp,coefDataTmp);
        xDataStoreTmp[i,j]=fDensData[1];
        yDataStoreTmp[i,j]=fDensData[2];
        zDataStoreTmp[i,j]=fDensData[3];
    end
end
fDataTmp[1]=-sum(xDataStoreTmp)*step1*step2;
fDataTmp[2]=-sum(yDataStoreTmp)*step1*step2;
fDataTmp[3]=-sum(zDataStoreTmp)*step1*step2;
return fDataTmp
end
```

Selection deleted

forcePart4 (generic function with 1 method)

```
##########function to execute part of the parallel computation##############
function
forcePart4(parID::Integer,n1::Integer,n2::Integer,lowLim1::Float64,upLim1::Float64,lowLim2::Float64,upLim2::Float64,ωbTmp::Float64, modelTmp::FrequencySimulation,
coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
fDataTmp=[0.0,0.0,0.0]###########[fx,fy,fz]######################
xDataStoreTmp=zeros(n1,n2);
yDataStoreTmp=zeros(n1,n2);
zDataStoreTmp=zeros(n1,n2);
step1=(upLim1-lowLim1)/n1;
step2=(upLim2-lowLim2)/n2;
for i in 1:n1
    θTmp=lowLim1+(i-1)*step1;
    for j in 1:n2
        ϕTmp=lowLim2+(j-1)*step2;
        fDensData=fDens(θTmp,ϕTmp,parID,ωbTmp,modelTmp,coefDataTmp);
        xDataStoreTmp[i,j]=fDensData[1];
        yDataStoreTmp[i,j]=fDensData[2];
        zDataStoreTmp[i,j]=fDensData[3];
    end
end
fDataTmp[1]=-sum(xDataStoreTmp)*step1*step2;
fDataTmp[2]=-sum(yDataStoreTmp)*step1*step2;
fDataTmp[3]=-sum(zDataStoreTmp)*step1*step2;
return fDataTmp
end
```

Selection deleted

forcePart5 (generic function with 1 method)

```
##########function to execute part of the parallel computation##############
function
forcePart5(parID::Integer,n1::Integer,n2::Integer,lowLim1::Float64,upLim1::Float64,lowLim2::Float64,upLim2::Float64,ωbTmp::Float64, modelTmp::FrequencySimulation,
coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
fDataTmp=[0.0,0.0,0.0]###########[fx,fy,fz]######################
xDataStoreTmp=zeros(n1,n2);
yDataStoreTmp=zeros(n1,n2);
zDataStoreTmp=zeros(n1,n2);
step1=(upLim1-lowLim1)/n1;
step2=(upLim2-lowLim2)/n2;
for i in 1:n1
    θTmp=lowLim1+(i-1)*step1;
    for j in 1:n2
        ϕTmp=lowLim2+(j-1)*step2;
        fDensData=fDens(θTmp,ϕTmp,parID,ωbTmp,modelTmp,coefDataTmp);
        xDataStoreTmp[i,j]=fDensData[1];
        yDataStoreTmp[i,j]=fDensData[2];
        zDataStoreTmp[i,j]=fDensData[3];
    end
end
fDataTmp[1]=-sum(xDataStoreTmp)*step1*step2;
fDataTmp[2]=-sum(yDataStoreTmp)*step1*step2;
fDataTmp[3]=-sum(zDataStoreTmp)*step1*step2;
return fDataTmp
end
```

Selection deleted

forcePart6 (generic function with 1 method)

```
##########function to execute part of the parallel computation##############
function
forcePart6(parID::Integer,n1::Integer,n2::Integer,lowLim1::Float64,upLim1::Float64,lowLim2::Float64,upLim2::Float64,ωbTmp::Float64, modelTmp::FrequencySimulation,
coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
fDataTmp=[0.0,0.0,0.0]###########[fx,fy,fz]######################
xDataStoreTmp=zeros(n1,n2);
yDataStoreTmp=zeros(n1,n2);
zDataStoreTmp=zeros(n1,n2);
step1=(upLim1-lowLim1)/n1;
step2=(upLim2-lowLim2)/n2;
for i in 1:n1
    θTmp=lowLim1+(i-1)*step1;
    for j in 1:n2
        ϕTmp=lowLim2+(j-1)*step2;
        fDensData=fDens(θTmp,ϕTmp,parID,ωbTmp,modelTmp,coefDataTmp);
        xDataStoreTmp[i,j]=fDensData[1];
        yDataStoreTmp[i,j]=fDensData[2];
        zDataStoreTmp[i,j]=fDensData[3];
    end
end
fDataTmp[1]=-sum(xDataStoreTmp)*step1*step2;
fDataTmp[2]=-sum(yDataStoreTmp)*step1*step2;
fDataTmp[3]=-sum(zDataStoreTmp)*step1*step2;
return fDataTmp
end
```

forcePart7 (generic function with 1 method)

```
##########function to execute part of the parallel computation##############
function
forcePart7(parID::Integer,n1::Integer,n2::Integer,lowLim1::Float64,upLim1::Float64,lowLim2::Float64,upLim2::Float64,ωbTmp::Float64, modelTmp::FrequencySimulation,
coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
fDataTmp=[0.0,0.0,0.0]###########[fx,fy,fz]######################
xDataStoreTmp=zeros(n1,n2);
yDataStoreTmp=zeros(n1,n2);
zDataStoreTmp=zeros(n1,n2);
step1=(upLim1-lowLim1)/n1;
step2=(upLim2-lowLim2)/n2;
for i in 1:n1
    θTmp=lowLim1+(i-1)*step1;
    for j in 1:n2
        ϕTmp=lowLim2+(j-1)*step2;
        fDensData=fDens(θTmp,ϕTmp,parID,ωbTmp,modelTmp,coefDataTmp);
        xDataStoreTmp[i,j]=fDensData[1];
        yDataStoreTmp[i,j]=fDensData[2];
        zDataStoreTmp[i,j]=fDensData[3];
    end
end
fDataTmp[1]=-sum(xDataStoreTmp)*step1*step2;
fDataTmp[2]=-sum(yDataStoreTmp)*step1*step2;
fDataTmp[3]=-sum(zDataStoreTmp)*step1*step2;
return fDataTmp
end
```

```
forcePart8 (generic function with 1 method)
```

```julia
##########function to execute part of the parallel computation#############
function forcePart8(parID::Integer,n1::Integer,n2::Integer,lowLim1::Float64,upLim1::Float64,lowLim2::Float64,upLim2::Float64,ωbTmp::Float64, modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
fDataTmp=[0.0,0.0,0.0]###########[fx,fy,fz]######################
xDataStoreTmp=zeros(n1,n2);
yDataStoreTmp=zeros(n1,n2);
zDataStoreTmp=zeros(n1,n2);
step1=(upLim1-lowLim1)/n1;
step2=(upLim2-lowLim2)/n2;
for i in 1:n1
    θTmp=lowLim1+(i-1)*step1;
    for j in 1:n2
        ϕTmp=lowLim2+(j-1)*step2;
        fDensData=fDens(θTmp,ϕTmp,parID,ωbTmp,modelTmp,coefDataTmp);
        xDataStoreTmp[i,j]=fDensData[1];
        yDataStoreTmp[i,j]=fDensData[2];
        zDataStoreTmp[i,j]=fDensData[3];
    end
end
fDataTmp[1]=-sum(xDataStoreTmp)*step1*step2;
fDataTmp[2]=-sum(yDataStoreTmp)*step1*step2;
fDataTmp[3]=-sum(zDataStoreTmp)*step1*step2;
return fDataTmp
end
```

```
forceT1 (generic function with 1 method)
```

```
#################single-threaded routine################################
###############numerical quadrature for acoustic force################
############n1,n2 denote sample numbers in polar angle θ and azimuthal angle ϕ coordinate respectively##############
function forceT1(parID::Integer,n1::Integer,n2::Integer,ωbTmp::Float64, modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
fDataTmp=[0.0,0.0,0.0]###########[fx,fy,fz]######################
xDataStoreTmp=zeros(n1,n2);
yDataStoreTmp=zeros(n1,n2);
zDataStoreTmp=zeros(n1,n2);
step1=π/n1;
step2=2*π/n2;
for i in 1:n1
    θTmp=(i-1)*step1;
    for j in 1:n2
        ϕTmp=(j-1)*step2;
        fDensData=fDens(θTmp,ϕTmp,parID,ωbTmp,modelTmp,coefDataTmp);
        xDataStoreTmp[i,j]=fDensData[1];
        yDataStoreTmp[i,j]=fDensData[2];
        zDataStoreTmp[i,j]=fDensData[3];
    end
end
fDataTmp[1]=-sum(xDataStoreTmp)*step1*step2;
fDataTmp[2]=-sum(yDataStoreTmp)*step1*step2;
fDataTmp[3]=-sum(zDataStoreTmp)*step1*step2;
return fDataTmp
end
```

```
forceT2 (generic function with 1 method)
```

```julia
#################2-threaded routine################################
###############numerical quadrature for acoustic force###############
############n1,n2 denote sample numbers in polar angle θ and azimuthal angle ϕ coordinate respectively##############
function forceT2(parID::Integer,n1::Integer,n2::Integer,ωbTmp::Float64, modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
threadNo=2;
threadNo1=1;
threadNo2=2;
n1Tmp=Int(n1/threadNo1);
n2Tmp=Int(n2/threadNo2);
lowLim1=Array{Float64}(undef,threadNo1);
upLim1=Array{Float64}(undef,threadNo1);
lowLim2=Array{Float64}(undef,threadNo2);
upLim2=Array{Float64}(undef,threadNo2);
for i in 1:threadNo1
    lowLim1[i]=(i-1)*π/threadNo1;
    upLim1[i]=lowLim1[i]+π/threadNo1;
end
for i in 1:threadNo2
    lowLim2[i]=(i-1)*2π/threadNo2;
    upLim2[i]=lowLim2[i]+2π/threadNo2;
end
result1=Threads.@spawn forcePart1(parID,n1Tmp,n2Tmp,lowLim1[1],upLim1[1],lowLim2[1],upLim2[1],ωbTmp, modelTmp, coefData);
result2=Threads.@spawn forcePart2(parID,n1Tmp,n2Tmp,lowLim1[1],upLim1[1],lowLim2[2],upLim2[2],ωbTmp, modelTmp, coefData);
return fetch(result1) + fetch(result2)
end
```

forceT4 (generic function with 1 method)

```
##################4-threaded routine#################################
################numerical quadrature for acoustic force################
#############n1,n2 denote sample numbers in polar angle θ and azimuthal angle ϕ coordinate respectively###############
function forceT4(parID::Integer,n1::Integer,n2::Integer,ωbTmp::Float64, modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
threadNo=4;
threadNo1=2;
threadNo2=2;
n1Tmp=Int(n1/threadNo1);
n2Tmp=Int(n2/threadNo2);
lowLim1=Array{Float64}(undef,threadNo1);
upLim1=Array{Float64}(undef,threadNo1);
lowLim2=Array{Float64}(undef,threadNo2);
upLim2=Array{Float64}(undef,threadNo2);
for i in 1:threadNo1
    lowLim1[i]=(i-1)*π/threadNo1;
    upLim1[i]=lowLim1[i]+π/threadNo1;
end
for i in 1:threadNo2
    lowLim2[i]=(i-1)*2π/threadNo2;
    upLim2[i]=lowLim2[i]+2π/threadNo2;
end
result1=Threads.@spawn forcePart1(parID,n1Tmp,n2Tmp,lowLim1[1],upLim1[1],lowLim2[1],upLim2[1],ωbTmp, modelTmp, coefData);
result2=Threads.@spawn forcePart2(parID,n1Tmp,n2Tmp,lowLim1[1],upLim1[1],lowLim2[2],upLim2[2],ωbTmp, modelTmp, coefData);
result3=Threads.@spawn forcePart3(parID,n1Tmp,n2Tmp,lowLim1[2],upLim1[2],lowLim2[1],upLim2[1],ωbTmp, modelTmp, coefData);
result4=Threads.@spawn forcePart4(parID,n1Tmp,n2Tmp,lowLim1[2],upLim1[2],lowLim2[2],upLim2[2],ωbTmp, modelTmp, coefData);
return fetch(result1) + fetch(result2) + fetch(result3) + fetch(result4)
end
```

forceT6 (generic function with 1 method)

```julia
##################6-threaded routine##################################
################numerical quadrature for acoustic force################
############n1,n2 denote sample numbers in polar angle θ and azimuthal angle ϕ coordinate respectively##############
function forceT6(parID::Integer,n1::Integer,n2::Integer,ωbTmp::Float64, modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
threadNo=4;
threadNo1=2;
threadNo2=3;
n1Tmp=Int(n1/threadNo1);
n2Tmp=Int(n2/threadNo2);
lowLim1=Array{Float64}(undef,threadNo1);
upLim1=Array{Float64}(undef,threadNo1);
lowLim2=Array{Float64}(undef,threadNo2);
upLim2=Array{Float64}(undef,threadNo2);
for i in 1:threadNo1
    lowLim1[i]=(i-1)*π/threadNo1;
    upLim1[i]=lowLim1[i]+π/threadNo1;
end
for i in 1:threadNo2
    lowLim2[i]=(i-1)*2π/threadNo2;
    upLim2[i]=lowLim2[i]+2π/threadNo2;
end
result1=Threads.@spawn forcePart1(parID,n1Tmp,n2Tmp,lowLim1[1],upLim1[1],lowLim2[1],upLim2[1],ωbTmp, modelTmp, coefData);
result2=Threads.@spawn forcePart2(parID,n1Tmp,n2Tmp,lowLim1[1],upLim1[1],lowLim2[2],upLim2[2],ωbTmp, modelTmp, coefData);
result3=Threads.@spawn forcePart3(parID,n1Tmp,n2Tmp,lowLim1[1],upLim1[1],lowLim2[3],upLim2[3],ωbTmp, modelTmp, coefData);
result4=Threads.@spawn forcePart4(parID,n1Tmp,n2Tmp,lowLim1[2],upLim1[2],lowLim2[1],upLim2[1],ωbTmp, modelTmp, coefData);
result5=Threads.@spawn forcePart5(parID,n1Tmp,n2Tmp,lowLim1[2],upLim1[2],lowLim2[2],upLim2[2],ωbTmp, modelTmp, coefData);
result6=Threads.@spawn forcePart6(parID,n1Tmp,n2Tmp,lowLim1[2],upLim1[2],lowLim2[3],upLim2[3],ωbTmp, modelTmp, coefData);
return fetch(result1) + fetch(result2) + fetch(result3) + fetch(result4)+ fetch(result5) + fetch(result6)
end
```

forceT8 (generic function with 1 method)

```julia
##################8-threaded routine#################################
################numerical quadrature for acoustic force################
############n1,n2 denote sample numbers in polar angle θ and azimuthal angle ϕ coordinate respectively##############
function forceT8(parID::Integer,n1::Integer,n2::Integer,ωbTmp::Float64, modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
threadNo=4;
threadNo1=2;
threadNo2=4;
n1Tmp=Int(n1/threadNo1);
n2Tmp=Int(n2/threadNo2);
lowLim1=Array{Float64}(undef,threadNo1);
upLim1=Array{Float64}(undef,threadNo1);
lowLim2=Array{Float64}(undef,threadNo2);
upLim2=Array{Float64}(undef,threadNo2);
for i in 1:threadNo1
    lowLim1[i]=(i-1)*π/threadNo1;
    upLim1[i]=lowLim1[i]+π/threadNo1;
end
for i in 1:threadNo2
    lowLim2[i]=(i-1)*2π/threadNo2;
    upLim2[i]=lowLim2[i]+2π/threadNo2;
end
result1=Threads.@spawn forcePart1(parID,n1Tmp,n2Tmp,lowLim1[1],upLim1[1],lowLim2[1],upLim2[1],ωbTmp, modelTmp, coefData);
result2=Threads.@spawn forcePart2(parID,n1Tmp,n2Tmp,lowLim1[1],upLim1[1],lowLim2[2],upLim2[2],ωbTmp, modelTmp, coefData);
result3=Threads.@spawn forcePart3(parID,n1Tmp,n2Tmp,lowLim1[1],upLim1[1],lowLim2[3],upLim2[3],ωbTmp, modelTmp, coefData);
result4=Threads.@spawn forcePart4(parID,n1Tmp,n2Tmp,lowLim1[1],upLim1[1],lowLim2[4],upLim2[4],ωbTmp, modelTmp, coefData);
result5=Threads.@spawn forcePart5(parID,n1Tmp,n2Tmp,lowLim1[2],upLim1[2],lowLim2[1],upLim2[1],ωbTmp, modelTmp, coefData);
result6=Threads.@spawn forcePart6(parID,n1Tmp,n2Tmp,lowLim1[2],upLim1[2],lowLim2[2],upLim2[2],ωbTmp, modelTmp, coefData);
result7=Threads.@spawn forcePart7(parID,n1Tmp,n2Tmp,lowLim1[2],upLim1[2],lowLim2[3],upLim2[3],ωbTmp, modelTmp, coefData);
result8=Threads.@spawn forcePart8(parID,n1Tmp,n2Tmp,lowLim1[2],upLim1[2],lowLim2[4],upLim2[4],ωbTmp, modelTmp, coefData);
return fetch(result1) + fetch(result2) + fetch(result3) + fetch(result4)+ fetch(result5) + fetch(result6)+ fetch(result7) + fetch(result8)
end
```

force (generic function with 1 method)

```
function force(parID::Integer,n1::Integer,n2::Integer,ωbTmp::Float64,
modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
num_threads = Threads.nthreads();
if num_threads==1
    println("Single-threaded computation")
    return forceT1(parID,n1,n2,ωbTmp, modelTmp, coefDataTmp)
elseif 2<=num_threads&&num_threads<4
    println("2-threaded parallel computation")
    return forceT2(parID,n1,n2,ωbTmp, modelTmp, coefDataTmp)
elseif 4<=num_threads&&num_threads<6
    println("4-threaded parallel computation")
    return forceT4(parID,n1,n2,ωbTmp, modelTmp, coefDataTmp)
elseif 6<=num_threads&&num_threads<8
    println("6-threaded parallel computation")
    return forceT6(parID,n1,n2,ωbTmp, modelTmp, coefDataTmp)
else
    println("8-threaded parallel computation")
    return forceT8(parID,n1,n2,ωbTmp, modelTmp, coefDataTmp)
end
end

```

allForce (generic function with 1 method)

```
##################calculate all particles forces##################
function allForce(n1::Integer,n2::Integer,ωbTmp::Float64,
modelTmp::FrequencySimulation, coefData::Matrix{ComplexF64})
coefDataTmp=deepcopy(coefData);
simModelTmp=modelTmp;
pNoTmp=length(simModelTmp.particles);
dimensionTmp=typeof(simModelTmp.source.medium).parameters[2];
allForceTmp=Array{Float64}(undef,0)
for iTmp in 1:pNoTmp
    forceTmp=force(iTmp,n1,n2,ωbTmp, modelTmp, coefDataTmp);
    allForceTmp=push!(allForceTmp,forceTmp[1],forceTmp[2],forceTmp[3]);
end
return allForceTmp
end
```

forcePackLow (generic function with 1 method)

```
function forcePackLow(RsTmp::Float64,parPos::Matrix{Float64})
parPosTmp=deepcopy(parPos);
freqIn=40000;
ω=2.0*π*freqIn;
dimension=3;
ρb=1.225;
ρp=29.0;
cb=343.0;
cp=900.0;
pNo=length(parPosTmp[:,1]);
coefOrder=6;
modelTmp=buildModelProto(dimension, ρb, cb, ρp, cp, ω, pNo, RsTmp, parPosTmp)
coefData=getCoefProto(ω, modelTmp, coefOrder);
forceTmp=allForce(24,48,ω, modelTmp, coefData);
return forceTmp
end
```

forcePackMiddle (generic function with 1 method)

```
function forcePackMiddle(RsTmp::Float64,parPos::Matrix{Float64})
parPosTmp=deepcopy(parPos);
freqIn=40000;
ω=2.0*π*freqIn;
dimension=3;
ρb=1.225;
ρp=29.0;
cb=343.0;
cp=900.0;
pNo=length(parPosTmp[:,1]);
coefOrder=8;
modelTmp=buildModelProto(dimension, ρb, cb, ρp, cp, ω, pNo, RsTmp, parPosTmp)
coefData=getCoefProto(ω, modelTmp, coefOrder);
forceTmp=allForce(30,60,ω, modelTmp, coefData);
return forceTmp
end
```

forcePackHigh (generic function with 1 method)

```
function forcePackHigh(RsTmp::Float64,parPos::Matrix{Float64})
parPosTmp=deepcopy(parPos);
freqIn=40000;
ω=2.0*π*freqIn;
dimension=3;
ρb=1.225;
ρp=29.0;
cb=343.0;
cp=900.0;
pNo=length(parPosTmp[:,1]);
coefOrder=10;
modelTmp=buildModelProto(dimension, ρb, cb, ρp, cp, ω, pNo, RsTmp, parPosTmp)
coefData=getCoefProto(ω, modelTmp, coefOrder);
forceTmp=allForce(36,72,ω, modelTmp, coefData);
return forceTmp
end
```