```
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,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
```

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

```
###########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
```

```
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
```

```
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
```

```
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
```

```
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)
```

```
###########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)
```

```
##################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)
```

```
###################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)
```

```
##################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

```