A Procedure to Estimate the Absorption Rate of Sound Propagating Through High Temperature Gas

K. Fujii and H.G. Hornung
Graduate Aeronautical Laboratories

California Institute of Technology
Pasadena, California 91125

GALCIT Report FM 2001.004

August 8, 2001

Abstract

A procedure to estimate the absorption rate of small disturbance acoustic waves propagating through high temperature gas due to relaxation is outlined. The procedure considers more than one relaxation mode, such as vibration modes and chemical reactions. It contains the known results for one relaxation mode as a special case. In the relaxation process, it is assumed that rotation and translation are in equilibrium and that vibration and chemistry may proceed at finite rates, while electronic excitation is assumed to be frozen in the equilibrium state. Some calculations are made as example cases with undisturbed base state in equilibrium for air, carbon dioxide and pure nitrogen in the temperature range from 1000K to 6000K. According to the results, carbon dioxide and air can have strong sound absorption due to vibrational relaxation at a frequency of around 1MHz and at a density ranging between $0.1\text{kg/m}^3 \sim 1.0\text{kg/m}^3$, which are typical conditions of boundary layer transition experiments in the T5 hypervelocity shock tunnel. On the other hand, nitrogen does not have a strong absorption effect even at a very high temperature of 5000K.

Nomenclature

Symbols

$\mathbf{A}$	Matrix for reciprocal of relaxation time constant, 1/s
a	speed of sound, m/s
a_e	equilibrium speed of sound, m/s
a_f	frozen speed of sound, m/s
C	Constant of integration in Equation 16
C_f	Coefficient in Equation 27
$\hat{C}_v$	specific heat at constant volume per mole, kJ/K mole
D	Absorption parameter, defined by Equation 16
e	energy per unit mass of gas, J/kg, or logarithmic constant
$\hat{e}$	energy per mole, J/mole
$\mathbf{f}$	Vector of spacial distribution function of disturbance, defined by Equation 12
f	frequency of disturbance, Hz, or arbitrary function of N_j in Equation 29
g	Spacial distribution function for potential ψ, defined in Equation 17
g_e	Degeneracy of electronic excitation of partition function, Q
g_v	Degeneracy of vibrational excitation of partition function, Q
h	Enthalpy per unit mass of gas, J/kg, or Planck constant, $= 6.6256 \times 10^{-34}$, Js
$\mathbf{I}$	Unit matrix
i	Imaginary unit, or index
K	Equilibrium constant
k	Boltzmann constant, $= 1.38054 \times 10^{-23}$ J/K
k_f	Forward reaction rate coefficient, Equation 26
$\overline{M}$	Averaged molecular weight, kg/mole
M_w	Molecular weight, kg/mole
N	Mole concentration per unit mass of mixture, mole/kg
N_A	Avogadro's number, $= 6.02252 \times 10^{23}$, 1/mole
n_r	Number of reactions
n_{r_e}	Number of linearly independent reactions
n_s	Number of species
p	Pressure, Pa
p'	Disturbance pressure, Pa
Q	Partition function
$\mathbf{q}$	Vector of nonequilibrium variables measured per unit mass , defined by Equation 21
$\mathbf{R}$	Matrix defined by Equation 23
$\mathbf{R}'$	Matrix defined by Equation 22
R	Universal gas constant, $= 8.31433$ J/K mole
T	Temperature, K
t	Time, s
u	Velocity, m/s
V	Volume, m^3
X	Square of the ratio of the frozen speed of sound and equilibrium speed of sound
x	Coordinate
$\boldsymbol{\alpha}$	Vector of degrees of reactions per unit mass
η	Coefficient in Equation 27
Θ	Characteristic temperature (rotational, vibrational, or electronic)
Θ_f	Characteristic temperature in forward reaction rate equation (27)

λ	Wave length of disturbance
μ_{sr}	Equivalent molecular weight of species s and r, defined by, $= \frac{Mw_s Mw_r}{Mw_s + Mw_r}$
ν_{ij}	Stoichiometric coefficient of jth species of the ith reaction, defined by, $\nu_{ij} = \nu''_{ij} - \nu'_{ij}$
ν'_{ij}	Stoichiometric coefficient of jth species on the reactants side of the ith reaction
ν''_{ij}	Stoichiometric coefficient of jth species on the products side of the ith reaction
ρ	Density, kg/m^3
σ	Symmetry factor
τ	Relaxation time, s
ψ	Potential function defined by Equation 8
ω	Angular frequency of disturbance wave, 1/s

Subscripts

o	Undisturbed value
el	electronic contribution
$imag.$	Imaginary part
$real$	Real part
rot	Rotational contribution
tr	Transitional contribution
v, or vib	Vibrational contribution

Superscripts

$*$	Undisturbed equilibrium condition
$\prime$	Disturbance value

1 Introduction

It is known that the propagation of acoustic waves can be affected significantly by relaxation of thermal vibrational excitation and/or chemical reactions in gases at high temperature. Although a weak relaxation effect of thermal vibrational modes at room temperature can be interpreted as a bulk viscosity, more careful considerations are required when the effect is larger, as has been pointed out by Lighthill[1]. Since such absorption phenomena can affect boundary layer transition in hypersonic flow[2–4], estimation of absorption due to relaxation in actual gas composition is important. For a simple case where only one relaxation mode exists, such as in the case of vibrational relaxation, or in an ideal dissociating gas, there are good text books on this phenomenon[5,6]. Generally speaking, the speed of sound depends on its frequency in relaxing gas, *i.e.*, ranging from the frozen sound speed at high frequency to the equilibrium sound speed at low frequency. For relaxation due to dissociation and/or vibration, the frozen sound speed is larger than the equilibrium sound speed.

In such case, sound wave is always damped by relaxation. This damping rate is a function of its frequency, which has a peak around a frequency of reciprocal of relaxation time (Figure 1, reproduction from Clarke *et. al.*[5]). Maximum damping rate depends on a ratio between the two sound speeds, a_f/a_e. This results is for an ideal dissociating gas of sound speed ratio, a_f/a_e, of approximately 1.18. This figure shows that sound at right frequency can be attenuated to 60% in magnitude per wave length.

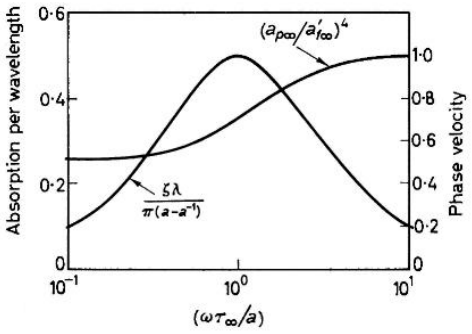


Figure 1: Absorption rate and speed of sound which propagate in ideal dissociating gas. $a_f/a_e \approx 1.183$. Reproduced from Clarke *et. al.*[5]

However, in high enthalpy flow, such as are produced in the T5 hypervelocity shock tunnel, relaxation of both thermal vibrational modes and chemical reactions can occur simultaneously. The sound speed ratio can change depending on the mole fraction of the species of interest, as well as temperature. Also, several chemical reactions can affect each other so that the situation may be more complicated than a simple sum of the individual damping effects taken separately. In this report, the damping rate will be estimated in such situations where each species may have more than one vibrationalmode with different relaxation times, and where several reactions each with a different relaxation time can take place. The procedure here is basically following the method described in Chap. VIII, "Introduction to physical gas dynamics", Vincenti &Kruger[6]. The difference from the text book is that more than two relaxation effects are considered here. In the relaxation process, it is assumed that rotation and translation are in equilibrium, and that vibration and chemistry may proceed at finite rates, while electronic excitation is assumed to be frozen in the equilibrium state.

2 Procedure

Estimation of sound attenuation rate for equilibrium gas at rest

Let **q** be a vector of non-equilibrium variables measured per unit mass (for example, these could

include e_v for vibrational non-equilibrium and/or $\boldsymbol{\alpha}$ for ideal dissociating gas). When the disturbance quantities are denoted with $'$, the governing equations for a small disturbance propagating through gas in equilibrium can be written as,

$$\frac{\partial \rho'}{\partial t} + \rho_o \frac{\partial u_i'}{\partial x_i} = 0, \tag{1}$$

$$\rho_o \frac{\partial u_i}{\partial t} + \frac{\partial p'}{\partial x_i} = 0, \tag{2}$$

$$\rho_o \frac{\partial h'}{\partial t} - \frac{\partial p'}{\partial t} = 0, \tag{3}$$

$$h' = h_{p_o} p' + h_{\rho_o} \rho' + \frac{\partial h}{\partial \mathbf{q}} \mathbf{q}', \tag{4}$$

$$\frac{\partial \mathbf{q}'}{\partial t} = \mathbf{A}(\mathbf{q}^{*\prime} - \mathbf{q}'), \tag{5}$$

$$\mathbf{q}^{*\prime} = \mathbf{q}^*{}_{p_o} p' + \mathbf{q}^*{}_{\rho_o} \rho', \tag{6}$$

where $*$ denotes equilibrium value under the given pressure and density. And $\mathbf{A}$ is a matrix related with the inverse of relaxation time, defined as,

$$\mathbf{A} \equiv -\frac{\partial}{\partial \mathbf{q}} \left(\frac{\partial \mathbf{q}}{\partial t} \right) \qquad \left(\sim \frac{1}{\tau} \right). \tag{7}$$

We restrict the discussion to one-dimensional sound propagation. In order to eliminate p' and u', introduce a function ψ from Equation 2 as follows:

$$p' = -\rho_o \psi_t, \qquad u' = \psi_x. \tag{8}$$

Thus, Equation 1 becomes

$$\frac{\partial \rho'}{\partial t} = -\rho_o \psi_{xx}.$$

Substituting into Equation 4, we can get

$$-\psi_{tt} = -\rho_o h_{p_o} \psi_{tt} - \rho_o h_{\rho_o} \psi_{xx} + \frac{\partial h}{\partial \mathbf{q}} \frac{\partial \mathbf{q}'}{\partial t}. \tag{9}$$

Differentiating Equation 5 with respect to time, t, and substituting Equation 6 to eliminate virtual fluctuation of $\mathbf{q}^{*\prime}$ in the equilibrium state, we get

$$\frac{\partial}{\partial t} \left(\frac{\partial \mathbf{q}'}{\partial t} \right) = \mathbf{A} \left\{ -\rho_o \mathbf{q}^*{}_{p_o} \psi_{tt} - \rho_o \mathbf{q}^*{}_{\rho_o} \psi_{xx} - \frac{\partial \mathbf{q}'}{\partial t} \right\}, \tag{10}$$

$$\frac{1 - \rho_o h_{p_o}}{\rho_o h_{\rho_o}} \psi_{tt} - \psi_{xx} + \frac{1}{\rho h_{\rho_o}} \frac{\partial h}{\partial \mathbf{q}} \frac{\partial \mathbf{q}'}{\partial t} = 0. \tag{11}$$

Assuming the frequencies for the fluctuation vector $\mathbf{q}'$ and ψ to be identical, *i.e.*,

$$\frac{\partial \mathbf{q}'}{\partial t} = \mathbf{f}(x) e^{i\omega t} \tag{12}$$

$$\psi = g(x) e^{i\omega t}. \tag{13}$$

Then the equations become

$$\frac{1-\rho_o h_{p_o}}{\rho_o h_{\rho_o}}\omega^2 g - g'' + \frac{1}{\rho h_{\rho_o}}\frac{\partial h}{\partial \mathbf{q}}\mathbf{f} = 0 \tag{14}$$

$$(i\omega\mathbf{I}+\mathbf{A})\,\mathbf{f} = \mathbf{A}\left(\rho_o\omega^2 g\mathbf{q}^*_{p_o} - \rho_o g''\mathbf{q}^*_{\rho_o}\right). \tag{15}$$

Solving the above equations, the function g can be written in the form

$$g = Ce^{Dx} \tag{16}$$

or,

$$\psi = ge^{i\omega t} = Ce^{-\left(\frac{2\pi D_{real}}{D_{imag.}}\right)\frac{x}{\lambda}}e^{i\omega\left(t+\frac{D_{imag.}}{\omega}x\right)}, \tag{17}$$

where

$$D = \pm\sqrt{\frac{-\omega^2\left(\frac{1}{a^2_{f_o}} - \frac{1}{h_{\rho_o}}\left(\frac{\partial h}{\partial \mathbf{q}}\right)(i\omega\mathbf{I}+\mathbf{A})^{-1}\mathbf{A}\mathbf{q}^*_{p_o}\right)}{1+\frac{1}{h_{\rho_o}}\left(\frac{\partial h}{\partial \mathbf{q}}\right)(i\omega\mathbf{I}+\mathbf{A})^{-1}\mathbf{A}\mathbf{q}^*_{\rho_o}}}, \tag{18}$$

$$a^2_{f_o} = \frac{h_{\rho_o}}{\frac{1}{\rho_o} - h_{p_o}},$$

$$\lambda = \frac{a}{f} = -\frac{2\pi(\omega/D_{imag.})}{\omega}.$$

Both for waves propagating in the positive direction ($-\frac{\omega}{D_{imag.}} > 0$) and in the negative direction ($-\frac{\omega}{D_{imag.}} < 0$), positive $\frac{D_{real}}{D_{imag.}}$ means damping. To confirm that this result contains the case of a single relaxation mode as a special case, consider the matrix $\mathbf{A}$ to be reduced to $\frac{1}{\tau}$, where τ is the relaxation time. Introducing another relaxation time, τ^+,

$$\tau^+ = \frac{\tau\,(h_p - 1/\rho)}{h_p + h_q q^*_p - 1/\rho},$$

the absorption parameter, D, can be reduced to

$$\begin{aligned} D^2 &= -\frac{\omega^2(i\omega\tau^+ + 1)}{i\omega\tau^+ a_f^2 + a_e^2} \\ &= -\frac{\omega^2}{a_e^2}\left[\frac{1+X(\omega\tau^+)^2}{1+X^2(\omega\tau^+)^2} + i\frac{(1-X)\omega\tau^+}{1+X^2(\omega\tau^+)^2}\right], \end{aligned} \tag{19}$$

where a_e is the equilibrium speed of sound, which can be written as

$$a_e^2 = -\frac{h_\rho + h_q q^*_\rho}{h_p + h_q q^*_p - 1/\rho},$$

and X is defined as the square of the ratio of the frozen to the equilibrium speed of sound,

$$X \equiv (a_f/a_e)^2\,.$$

Equation 19 is identical to well-known result for the ideal dissociating gas case[5,6]. It should be noted that the sign of $\frac{D_{real}}{D_{imag.}}$, which determines whether relaxation acts as damping or amplifying

force, depends on the sign of $1 - X$, *i.e.*, $a_e - a_f$, as can be seen from the sign of the imaginary part of D^2 shown in Equation 19:

$$\frac{D_{real}}{D_{imag.}} \quad \begin{cases} > 0 & (a_f > a_e) : \text{damping} \\ < 0 & (a_f < a_e) : \text{amplifing} \end{cases}.$$

Definition of variables and description of the system

The linearly independent set of chemical reactions can be represented as

$$\overbrace{\sum_{j=1}^{n_s} \nu'_{ij} X_j}^{reactants} \rightleftharpoons \overbrace{\sum_{j=1}^{n_s} \nu''_{ij} X_j}^{products},$$

$$\nu_{ij} \equiv \nu''_{ij} - \nu'_{ij}.$$

Defining a vector $\boldsymbol{\alpha}$, which represents the degree of advancement of the reactions per unit mass (mole/kg), each species concentration, N_j, moles per unit mass of mixture, can be written by $\boldsymbol{\alpha}$ as

$$N_j = \sum_{i=1}^{n_{r_e}} \alpha_i \nu_{ij} + N_{0j}. \tag{20}$$

The vector of non-equilibrium variables, $\mathbf{q}$, can be defined as a combination of $\boldsymbol{\alpha}$ and the vibrational energies per unit mass for the molecular species:

$$\mathbf{q} \equiv \begin{pmatrix} \boldsymbol{\alpha} \\ e_v(1) \\ \vdots \\ e_v(n_s) \end{pmatrix}. \tag{21}$$

Since it is necessary to take account of several reaction paths to evaluate the non-equilibrium rate processes properly, a matrix, $\mathbf{R}'$, which represents the relation between the set of linearly independent reactions (n_{r_e}) and the remaining reactions($n_r > n_{r_e}$) can be introduced:

$$\boldsymbol{\nu}^* = \begin{pmatrix} \mathbf{I} \\ \mathbf{R}' \end{pmatrix} \boldsymbol{\nu}. \tag{22}$$

where $\boldsymbol{\nu^*}$ is a $n_r \times n_s$ coefficient matrix for all reactions, and $\mathbf{R}'$ is $n_r - n_{r_e} \times n_{r_e}$ matrix. Then, letting $\boldsymbol{\alpha'}$ express the degrees of advancement of reactions $\boldsymbol{\nu^*}$,

$$\boldsymbol{\alpha} = \begin{pmatrix} \mathbf{I} & {}^{\mathbf{t}}\mathbf{R}' \end{pmatrix} \boldsymbol{\alpha}' \equiv \mathbf{R}\boldsymbol{\alpha}'. \tag{23}$$

Calculation of derivative values

The pressure and density dependence of the $\mathbf{q}$ in equilibrium can be calculated as follows: In the equilibrium state,

$$K_i(T) = \rho^{\sum_j \nu_{ij}} \prod_j N_j^{*\nu_{ij}}, \tag{24}$$

must be satisfied. Using the following form of the equation of state,

$$\frac{dT}{T} = \frac{1}{p} dp - \frac{1}{\rho} d\rho - \overline{M} \sum_i (\sum_j \nu_{ij}) d\alpha_i, \tag{25}$$

and differentiating the law of mass action (Equation 24) with respect to T, the following equation can be derived

$$\mathbf{A}_p dp + \mathbf{A}_\rho d\rho - \mathbf{A}_\alpha d\boldsymbol{\alpha}^* = 0,$$

where

$$\mathbf{A}_{pi} \equiv \frac{T}{K_i}\frac{dK_i}{dT}\left(\frac{1}{p}\right)$$

$$\mathbf{A}_{\rho i} \equiv \left[\frac{T}{K_i}\frac{dK_i}{dT}\left(\frac{1}{\rho}\right) + \frac{\sum_j \nu_{ij}}{\rho}\right]$$

$$\mathbf{A}_{\alpha ij} \equiv \left[\frac{T}{K_i}\frac{dK_i}{dT}\overline{M}\sum_k \nu_{jk} + \sum_k \frac{\nu_{ik}\nu_{jk}}{N_k}\right].$$

Then, the equilibrium change of the vector $\boldsymbol{\alpha}$ corresponding to given changes in pressure and density can be calculated as follows,

$$d\boldsymbol{\alpha}^* = \mathbf{A}_\alpha^{-1}\mathbf{A}_p dp + \mathbf{A}_\alpha^{-1}\mathbf{A}_\rho d\rho,$$

$$\left(\frac{\partial \boldsymbol{\alpha}^*}{\partial p}\right) = \mathbf{A}_\alpha^{-1}\mathbf{A}_p,$$

$$\left(\frac{\partial \boldsymbol{\alpha}^*}{\partial \rho}\right) = \mathbf{A}_\alpha^{-1}\mathbf{A}_\rho.$$

Similarly, the change in vibrational energy, assuming a harmonic oscillator model is

$$de^*_{v_i} = \frac{de^*_{v_i}}{dT}dT = e^*_{v_i}\frac{\left(\frac{\Theta_{v_i}}{T^2}\right)e^{\frac{\Theta_{v_i}}{T}}}{e^{\frac{\Theta_{v_i}}{T}}-1}\left\{\left(\frac{T}{p} + \frac{\partial T}{\partial \boldsymbol{\alpha}}\frac{\partial \boldsymbol{\alpha}^*}{\partial p}\right)dp + \left(-\frac{T}{\rho} + \frac{\partial T}{\partial \boldsymbol{\alpha}}\frac{\partial \boldsymbol{\alpha}^*}{\partial \rho}\right)d\rho\right\},$$

since

$$dT = \frac{T}{p}dp + \frac{T}{\rho}d\rho + \frac{\partial T}{\partial \boldsymbol{\alpha}}d\boldsymbol{\alpha},$$

where, from Equation 25,

$$\left(\frac{\partial T}{\partial \alpha_i}\right) = -\overline{M}T\sum_k \nu_{ik}.$$

In Equation 18, $\frac{\partial h}{\partial \mathbf{q}}$ as well as h_{ρ_o} and h_{p_o} need to be known. They are calculated under the assumption mentioned before, as,

$$\frac{\partial h}{\partial \rho} = -\sum_{s=1}^{n_s} N_s \frac{T}{\rho}\hat{C}_{v,t+r} - \frac{p}{\rho^2},$$

$$\frac{\partial h}{\partial p} = \sum_{s=1}^{n_s} N_s \frac{T}{p}\hat{C}_{v,t+r} + \frac{1}{\rho},$$

$$\frac{\partial h}{\partial e_{v_i}} = M_i N_i,$$

$$\frac{\partial h}{\partial \alpha_i} = \sum_{j=1}^{n_s}\left[\nu_{ij}\left(\hat{e}_j + \hat{e}_{0j}\right) - N_j\overline{M}T\sum_j \nu_{ij}\hat{C}_{v,t+r}\right].$$

The first two equations indicate our assumption of "thermal equilibrium in translation and rotation" and "frozen electronic excitation" in the disturbance wave.

Relaxation time

The relaxation time of chemical reaction is related to reaction rate, $\frac{d\boldsymbol{\alpha}'}{dt}$, each of which can be estimated by

$$\rho\frac{d\alpha_i'}{dt} = k_{f_i}\left[\prod_s (\rho N_s)^{\nu'_{i,s}} - \frac{1}{K_i}\prod_s (\rho N_s)^{\nu''_{i,s}}\right], \tag{26}$$

where, k_{f_j} is the forward reaction rate coefficient of the form

$$k_{f_j} = C_{f_j} T^{\eta_{f_j}} e^{\frac{\Theta_{f_j}}{T}}. \tag{27}$$

The reaction rates for the linearly independent reactions are then derived using Equation 23, *i.e.*,

$$\frac{d\boldsymbol{\alpha}}{dt} = \begin{pmatrix}\mathbf{I} & {}^{\mathbf{t}}\mathbf{R}'\end{pmatrix}\frac{d\boldsymbol{\alpha}'}{dt} = \mathbf{R}\frac{d\boldsymbol{\alpha}'}{dt}. \tag{28}$$

In order to differentiate an arbitrary function (say, f) of species concentration, N_j, with respect to α_i, we can use the following relation from Equation 20. The reaction rates for the linearly independent reactions are then derived using

$$\begin{aligned}\frac{\partial f}{\partial \alpha_i} &= \sum_j \frac{\partial f}{\partial N_j}\frac{\partial N_j}{\partial \alpha_i} \\ &= \sum_j \nu_{ij}\frac{\partial f}{\partial N_j}.\end{aligned} \tag{29}$$

Differentiate Equation 28 with respect to α_j, using Equation 29,

$$\frac{\partial}{\partial \alpha_j}\left(\frac{d\alpha_i}{dt}\right) = \sum_k R_{ik}\frac{k_{f_k}}{\rho}\left[\sum_l\left(\frac{\nu'_{kl}\nu_{jl}}{N_l}\right)\prod_s (\rho N_s)^{\nu'_{ks}} - \frac{1}{K_k}\sum_l\left(\frac{\nu''_{kl}\nu_{jl}}{N_l}\right)\prod_s (\rho N_s)^{\nu''_{ks}}\right].$$

When the undisturbed condition is in equilibrium, the above equation can be reduced to

$$\frac{\partial}{\partial \alpha_j}\left(\frac{d\alpha_i}{dt}\right) = -\sum_k R_{ik}\frac{k_{f_k}}{\rho}\left[\sum_l\left(\frac{\nu_{kl}\nu_{jl}}{N_l}\right)\right]\prod_s (\rho N_s)^{\nu'_{ks}}.$$

In the above equation, changes in k_{f_j} and K_k in accordance with temperature changes resulting from $\delta\alpha_i$, *i.e.*, $(\partial K_k/\partial T)(\partial T/\partial \alpha_i)$, are ignored.

The vibarational energy relaxation for species s due to collisions with species r is determined from the expression given by Millikan and White[7],

$$\ln A_1 \tau_{sr} p = A_2 \mu_{sr}^{0.5}\Theta_{v_s}^{4/3}\left(T^{-1/3} - A_3\mu_{sr}^{1/4}\right),$$

where μ_{sr} is the equivalent molecular weight between the two species. Constants, A_1, A_2 and A_3 are,

$A_1 = 9.8625\times10^2$ Pa^{-1}s^{-1} $\qquad$ $A_2 = 0.0367$ kg$^{-1/2}$mol$^{1/2}$K$^{-5/3}$ $\qquad$ $A_3 = 0.08435$ K$^{1/3}$mol$^{1/4}$kg$^{-1/4}$.

The relaxation time of species s is determined by taking the number-weighted average of τ_{sr}:

$$\tau_s = \frac{\sum_r N_r}{\sum_r \frac{N_r}{\tau_{sr}}}. \tag{30}$$

For CO_2, however, since it has four vibrational modes that relax at the same rate according to Camac's experiments[8], the following formula proposed by him is used here, instead:

$$\ln A_4 \tau_{CO_2} p = A_5 T^{-1/3},$$

where constants A_4 and A_5 are

$$A_4 = 4.8488 \times 10^2 \text{ Pa}^{-1}\text{s}^{-1} \qquad A_5 = 36.5 \text{ K}^{1/3}.$$

Strictly speaking, the vibrational relaxation time might be a function of concentration. However, it is assumed that this effect of the disturbance wave is small. (Vibrational relaxation time is assumed constant through the disturbance wave at a value evaluated with the undisturbed equilibrium concentration.) Then the matrix $\mathbf{A}$ in equation 7 becomes

$$\mathbf{A}_{ij} = \frac{\partial}{\partial \alpha_j}\left(\frac{d\alpha_i}{dt}\right) \quad i,j < n_{r_e},$$

for chemical relaxation, and

$$\mathbf{A}_{ij} = \begin{cases} 0 & i \neq j \\ \frac{1}{\tau_s} & i = j \end{cases}.$$

for vibrational relaxation.

3 Example cases

This procedure for the absorption rate calculation can be easily applied to various gases, which may have several reactions and vibrational modes. Here, results of some example cases, which are typical boundary layer reference conditions of T5 hypervelocity shock tunnel experiments[2–4].

Estimation of the equilibrium state

The equilibrium condition, including energy, $e_{tr}, e_{rot}, e_{vib}, e_{el}$, pressure and partition functions for each species and for each excitation mode, $Q_{tr}, Q_{rot}, Q_{vib}, Q_{el}$ are required for the sound absorption calculation as an initial undisturbed state. The equilibrium constant of the i-th reaction is determined from the relation below by definition,

$$K_i = \prod \left[\left(\frac{Q_j}{N_A V}\right) e^{-\frac{\epsilon_i}{kT}}\right]^{\nu_{ij}},$$

where, Q_j is the partition function of the j-th species which, for a harmonic oscillator model, can be written as

$$\begin{aligned} \frac{Q_j}{N_A V} &= \frac{Q_{j_{tr}}}{N_A V} Q_{j_{rot}} Q_{j_{vib}} Q_{j_{el}} \\ &= (2\pi M_w T)^{3/2} \left(\frac{k^{3/2}}{N_A^{5/2} h^3}\right) \frac{1}{\sigma} \left(\frac{T}{\Theta_{rot}}\right) \prod \left[\left(1 - e^{-\frac{\Theta_{v_i}}{T}}\right)^{g_{v_i}}\right] \sum g_{e_i} e^{-\frac{\Theta_{e_i}}{T}}. \end{aligned}$$

By substituting these into Equation 24, the mole fraction of each species can be obtained. For nonlinear polyatomic molecules, the rotational partition function should be replaced by

$$\frac{\sqrt{\pi}}{\sigma}\left(\frac{T^3}{\Theta_{rot,A}\Theta_{rot,B}\Theta_{rot,C}}\right)^{1/2}.$$

The energy of each excitation mode is calculated directly from partition functions:

$$e_i = RT^2\frac{\partial \log Q_i}{\partial T}.$$

Results of some examples with the gases air, carbon dioxide and pure nitrogen are calculated from property data of species (shown in Table 2) and reactions (shown in Table 3) and are compared with a STANJAN[9] calculation, which is based on JANAF table[10], in Figure 2,3. STANJAN results are indicated by the symbols with circle at each 1000K, while results from the harmonic oscillator model are shown with lines and symbols at each 500K. They show good agreement throughout the temperature range, indicating validity of the harmonic oscillator model at the conditions calculated here.

Air

Using the data shown in Tables 2,3, sound absorption rates per wavelength, as defined by $\frac{2\pi D_{real}}{D_{imag.}}$, are calculated and are shown in Figure 4 as functions of disturbance frequency. Since there are a lot of chemical reactions and vibrational modes in the air case, the resultant curves appear quite complicated. However, for sufficiently low temperature, there should be only one relaxation process, *i.e.*, vibration of oxygen molecule. Actually, we can see a weak absorption peak at a temperature below 2000K which corresponds to this mode. Disturbance frequencies lower than the peak absorption frequency behave, in general, like equilibrium disturbances, while much higher frequencies are frozen. As temperature increases, generally speaking, the relaxation time decreases. This means that the peak absorption frequency increases with increasing temperature, as can be seen in the figures. At sufficiently high density, and as the temperature is increased, another and stronger absorption peak, whose frequency is lower, appears and then merges with the first peak. It corresponds to relaxation due to vibration of the nitrogen molecule and to dissociation of oxygen.

For boundary layer transition experiments in the T5 hypervelocity shock tunnel, the frequency range of interest is typically around 1MHz at a density of approximately 0.1kg/m^3. In this region, it can be seen that both temperature and density increase cause stronger relaxation effects. At sufficiently high density, considerable absorption can be expected at a temperature of 4000K$\sim$5000K.

At T=2000K, and ρ=1.0kg/m^3, where only vibrational relaxation of oxygen takes place, the minimum value of the ratio of sound speeds a/a_f, is approximately 0.98, so that a strong relaxation effect cannot be expected as is seen in the sound absorption results. Higher temperature cases this ratio reaches values as low as approximately 0.9. This corresponds to vibration of nitrogen and dissociation of oxygen as noted above.

Carbon Dioxide

Estimations of the absorption rate in carbon dioxide are shown in Figure 5. Since all vibrational relaxation times are assumed identical here, the absorption curves have two humps at most, one is for absorption due to this vibration and the other for that due to dissociation of the carbon dioxide molecule. The higher-frequency absorption peaks, which corresponds to vibration, decrease with increasing temperature. This is because the mole fraction of carbon dioxide molecule decreases with increasing temperature due to dissociation. This effect can be seen also in the plots of sound

speed ratio. The first plateau part increases from approximately 0.9 to 0.98 as temperature increase from 1000K to 5000K.

Due to the rich vibrational modes of carbon dioxide molecule and their relatively low characteristic vibrational temperature, absorption at a frequency of arrond 1MHz is quite strong even at a temperature below 3000K. Also, the sound speed ratio, a/a_f, due to vibration is as low as 0.9, which leads to high sound absorption rate even at relatively low temperature.

Nitrogen
Results for pure nitrogen are shown in Figure 6. Since there are only two relaxation modes, they look quite simple. Due to higher characteristic vibrational temperature and to higher dissociation energy than those of oxygen or carbon dioxide, relaxation gives much lower absorption rate and lower peak frequency than the above two examples. In order to obtain a large absorption effect at a frequency of approximately 1MHz, the temperature has to be even higher than 5000K for a density of 0.1kg/m^3.

4 Conclusion

A procedure to calculate absorption rates for sound propagating through high temperature gas is extended to conditions where more than two relaxation mechanisms may exist. It contains the known results for one relaxation mode as a special case.

Using the procedure, absorption rates in the gases air, carbon dioxide and pure nitrogen are calculated under several conditions with equilibrium species concentration. According to the results, carbon dioxide and air can have strong sound absorption due to vibrational relaxation at a frequency of around 1MHz and at a density ranging between $0.1\text{kg/m}^3 \sim 1.0\text{kg/m}^3$, which are typical conditions of boundary layer transition experiments in the T5 hypervelocity shock tunnel. On the other hand, nitrogen does not have a strong absorption effect at condition even at a very high temperature of 5000K.

References

[1] M.J. Lighthill. Viscosity effects in sound waves of finite amplitude. In G.K. Batchelor and R.M. Davies, editors, *Surveys in Mechanics*, pages 250–351. Cambridge University Press, 1956.

[2] P. Germain. *The Boundary Layer on a Sharp Cone in High-Enthalpy Flow.* PhD thesis, California Institute of Technology, 1994.

[3] P. Adam. *Enthalpy Effects on Hypervelocity Boundary Layers.* PhD thesis, California Institute of Technology, 1997.

[4] K. Fujii and H.G. Hornung. An experiment of high enthalpy effect on attachment line transition. AIAA Paper 2001-2779, 2001. (31st AIAA Fluid Dynamics Conference and Exhibit, June 11-14, Anaheim, CA, USA).

[5] J.F. Clarke and M. McChesney. *The Dynamics of Real Gases.* Butterworths, 1964.

[6] W.G. Vincenti and C.H. Kruger. *Introduction to Physical Gas Dynamics.* Krieger Publishing Company, 1965.

[7] R.C. Millikan and D.R. White. Systematics of vibrational relaxation. *The Journal of Chemical Physics*, 39(12):3209–3213, 1963.

[8] M. Camac. CO_2 relaxation processes in shock waves. In J.G. Hall, editor, *Fundamental Phenomena in Hypersonic Flow*, pages 195–218. Cornell University Press, 1966.

[9] W.C. Reynolds. The element potential method of chemical equilibrium analysis: Implementation in the interactive program STANJAN. Technical report, Department of Mechanical Engineering, Stanford University, 1986.

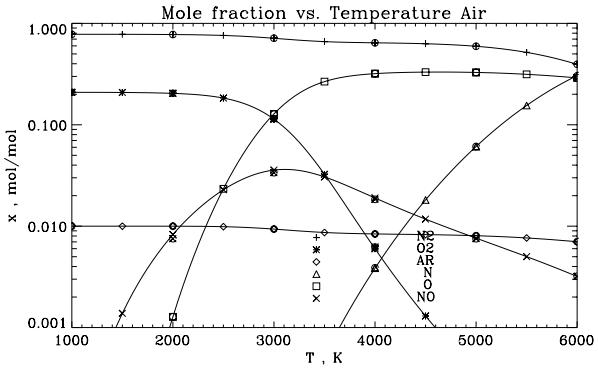
[10] Malcolm W. Chase. NIST-JANAF thermochemical tables. *Journal of Physical and Chemical Reference Data, Monograph*, (9), 1998.

Table 2: Properties used in the calculations

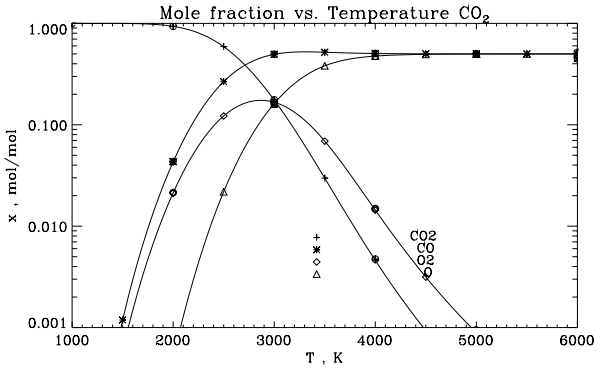
species	M_w g/mol	$\sigma\Theta_{rot}$ K	h_f^o J/mol	g_v	Θ_v K	g_{e_o}	g_{e_i}	ϵ_e J/mol
N_2	28.016	5.79	0.0	1	3353.2	1	3	6.015×10^5
							6	7.136×10^5
							1	7.342×10^5
O_2	32.000	4.16	0.0	1	2239.0	3	2	9.225×10^4
							1	1.579×10^5
							3	4.320×10^5
							3	5.960×10^5
Ar	39.944	–	0.0	–	–	1	5	1.115×10^6
							3	1.122×10^6
N	14.008	–	4.713×10^5	–	–	4	6	2.301×10^5
							4	2.308×10^5
							6	3.452×10^5
							12	9.971×10^5
O	16.000	–	2.468×10^5	–	–	5	3	1.903×10^3
							1	2.717×10^3
							5	1.899×10^5
							1	4.044×10^5
							5	8.829×10^5
NO	30.008	2.45	8.990×10^4	1	2699.2	4	2	5.262×10^5
							4	5.496×10^5
C	12.011	–	7.116×10^5	–	–	1	3	1.962×10^2
							5	5.204×10^2
							5	1.219×10^5
							1	2.590×10^5
							5	4.036×10^5
CO_2	44.011	1.13	-3.933×10^5	2	960.1	1		
				1	1992.5			
				1	3380.2			
CO	28.011	2.78	-1.139×10^5	1	3082.0	1	6	5.824×10^5
							3	6.687×10^5
							6	7.453×10^5
							2	7.785×10^5
H_2	2.016	175.09	0.0	1	6100.0	1	1	1.097×10^6
							2	1.196×10^6
							3	1.352×10^6
H	1.008	–	2.110×10^5	–	–	1	3	9.839×10^5
							4	1.166×10^6
							5	1.230×10^6

Table 3: Chemical rate coefficients

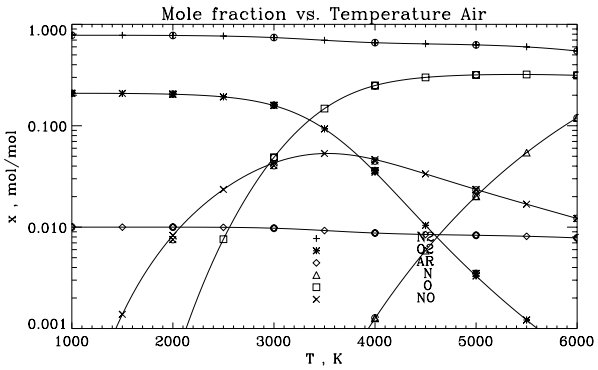
Third body	C_f m^3/mole·s	η_f	Θ_f K
pure N_2 case			
	$N_2 \rightleftharpoons 2N$		
N_2	2.30×10^{23}	-3.5	113261.
N	8.50×10^{19}	-2.5	113261.
Air case			
	$N_2 \rightleftharpoons 2N$		
N_2	2.30×10^{23}	-3.5	113261.
N	8.50×10^{19}	-2.5	113261.
the others	9.90×10^{14}	-1.5	113260.
	$O_2 \rightleftharpoons 2O$		
O_2	3.60×10^{15}	-1.5	59390.
O	2.10×10^{12}	-0.5	59390.
the others	1.20×10^{15}	-1.5	59390.
	$NO \rightleftharpoons N + O$		
all species	5.20×10^{15}	-1.5	75500.
	$O_2 + N \rightleftharpoons O + NO$		
none	1.00×10^{6}	0.5	3625.
	$N_2 + O \rightleftharpoons N + NO$		
none	5.00×10^{7}	0.0	38020.
	$N_2 + O_2 \rightleftharpoons 2NO$		
none	9.10×10^{18}	-2.5	65010.
CO_2 case			
	$CO_2 \rightleftharpoons CO + O$		
all species	2.88×10^{5}	0.5	37655.
	$2CO \rightleftharpoons CO_2 + C$		
none	2.33×10^{3}	0.5	65694.
	$CO + O_2 \rightleftharpoons CO_2 + O$		
none	1.60×10^{7}	0.0	20640.
	$CO \rightleftharpoons C + O$		
CO	1.76×10^{24}	-3.52	128751.
O	1.29×10^{25}	-3.52	128751.
the others	8.79×10^{23}	-3.52	128751.
	$O_2 \rightleftharpoons 2O$		
O_2	2.75×10^{13}	-1.0	59754.
O	2.10×10^{12}	-0.5	59382.
the others	2.55×10^{12}	-1.0	59754.
	$CO + O \rightleftharpoons O_2 + C$		
none	2.73×10^{5}	0.5	69520.



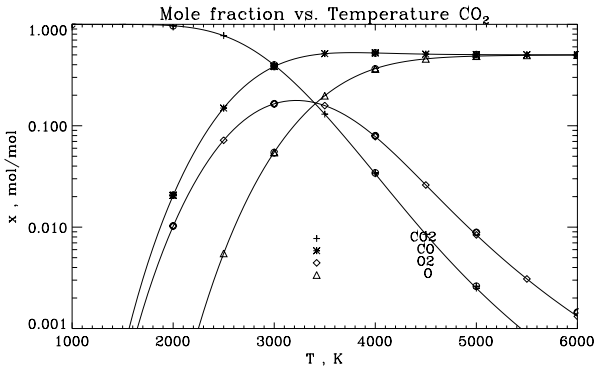
Air, $\rho = 0.01\text{kg/m}^3$



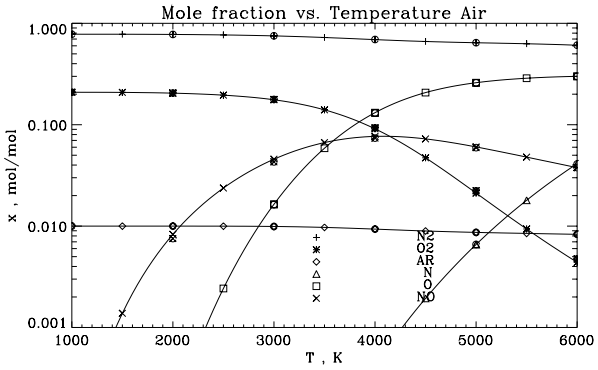
CO_2, $\rho = 0.01\text{kg/m}^3$



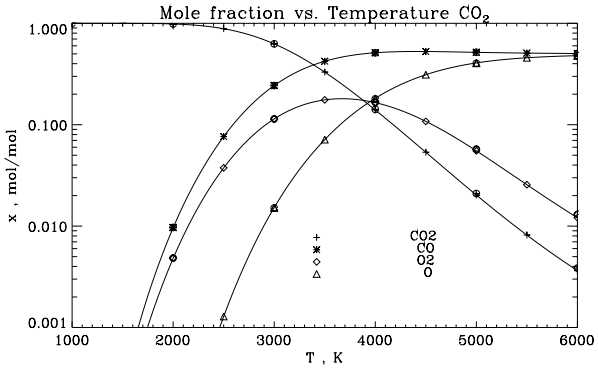
Air, $\rho = 0.10\text{kg/m}^3$



CO_2, $\rho = 0.10\text{kg/m}^3$

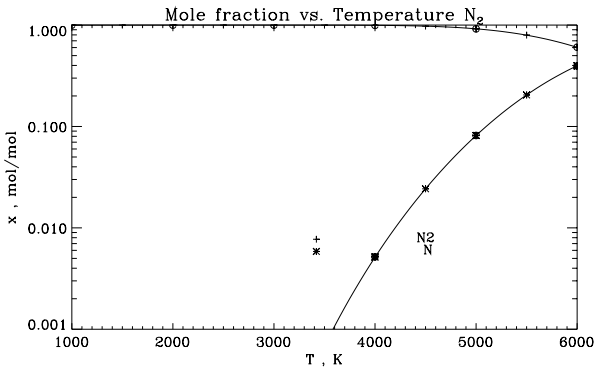


Air, $\rho = 1.0\text{kg/m}^3$

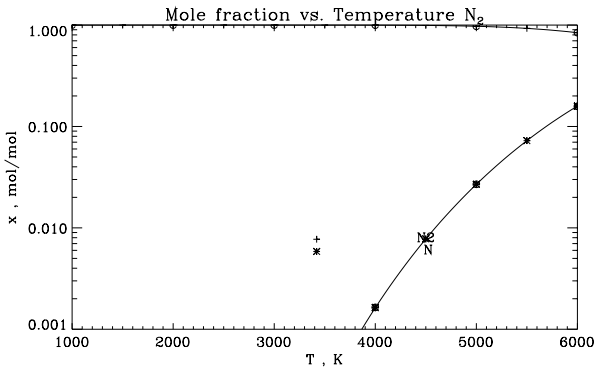


CO_2, $\rho = 1.0\text{kg/m}^3$

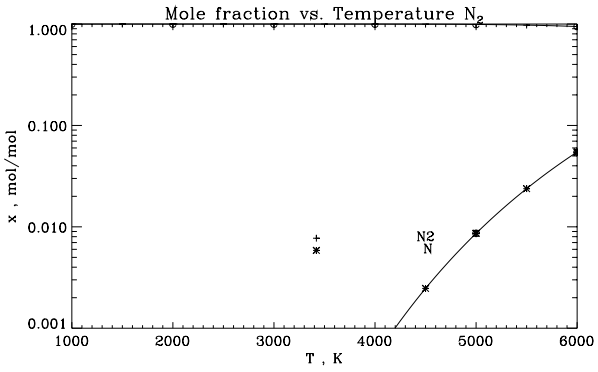
Figure 2: Comparisons of estimated mole concentrations with STANJAN results , symbols with circle, at constant density in equilibrium state



N_2, $\rho = 0.01\text{kg/m}^3$

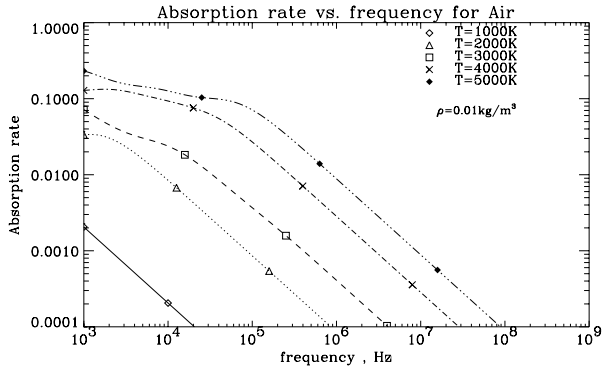


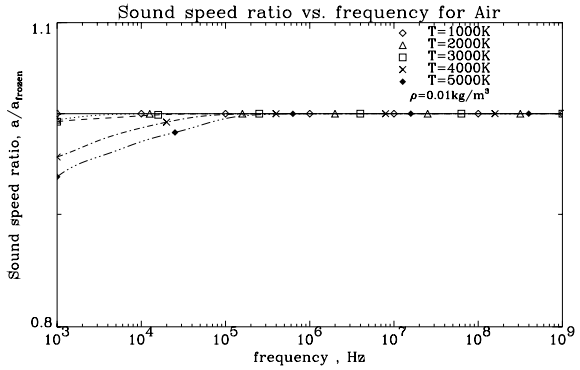
N_2, $\rho = 0.10\text{kg/m}^3$



N_2, $\rho = 1.0\text{kg/m}^3$

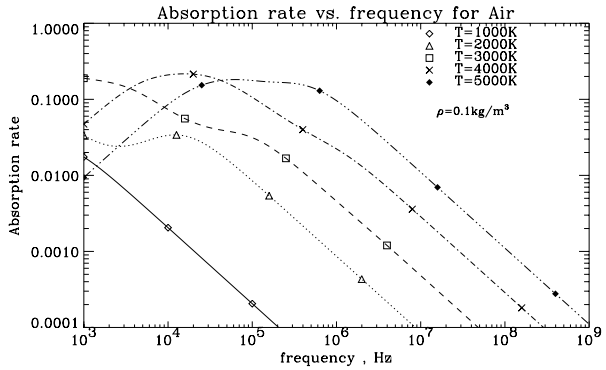
Figure 3: Comparisons of estimated mole concentrations with STANJAN results , symbols with circle, at constant density in equilibrium state

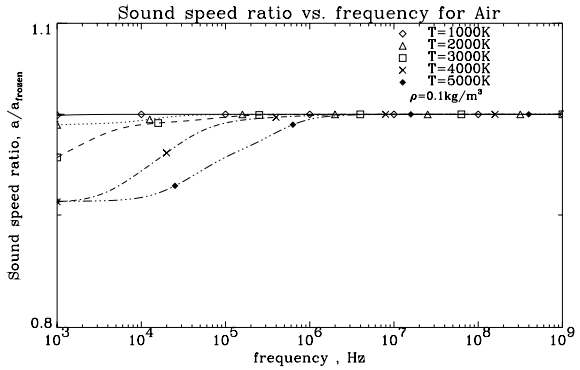




Sound absorption rate per wave length, $\frac{2\pi D_{real}}{D_{imag.}}$ Ratio of sound speeds, a/a_f

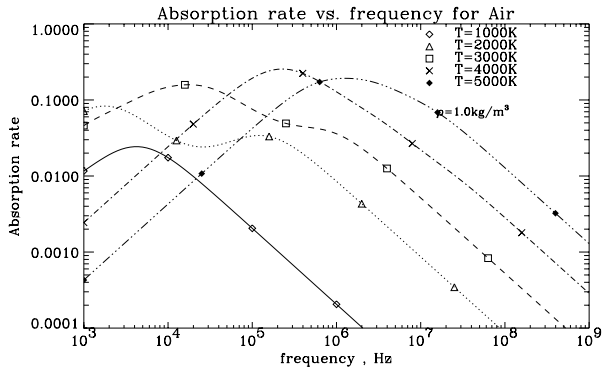
$\rho = 0.01\text{kg/m}^3$, Air

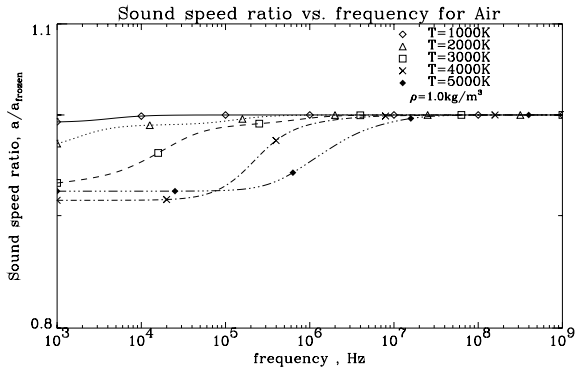




Sound absorption rate per wave length, $\frac{2\pi D_{real}}{D_{imag.}}$ Ratio of sound speeds, a/a_f

$\rho = 0.1\text{kg/m}^3$, Air

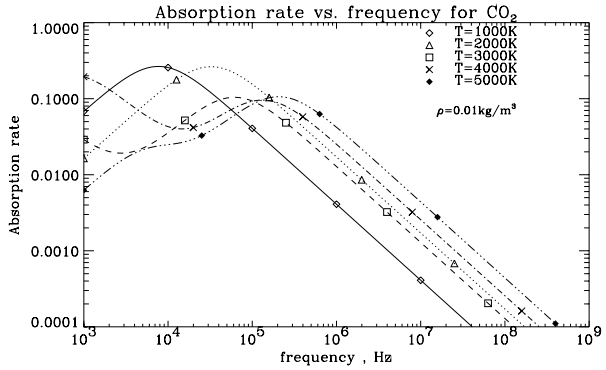


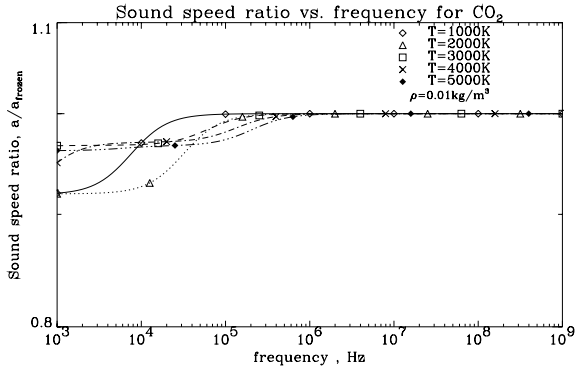


Sound absorption rate per wave length, $\frac{2\pi D_{real}}{D_{imag.}}$ Ratio of sound speeds, a/a_f

$\rho = 1.0\text{kg/m}^3$, Air

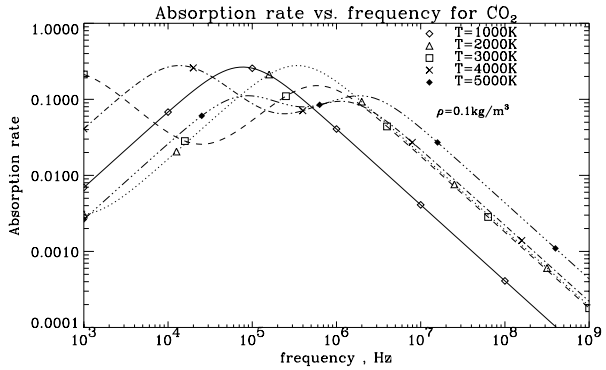
Figure 4: Estimated sound absorption rate and sound speeds ratio at constant density versus frequency for air.

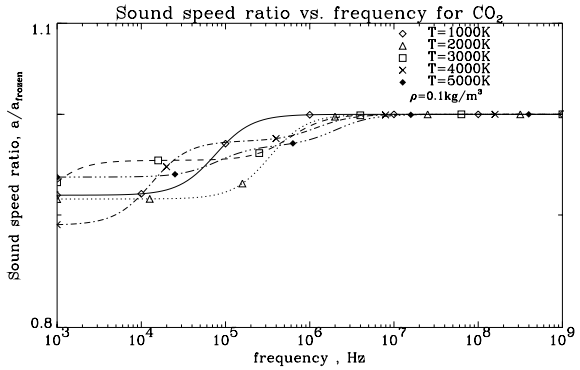




Sound absorption rate per wave length, $\frac{2\pi D_{real}}{D_{imag.}}$ Ratio of sound speeds, a/a_f

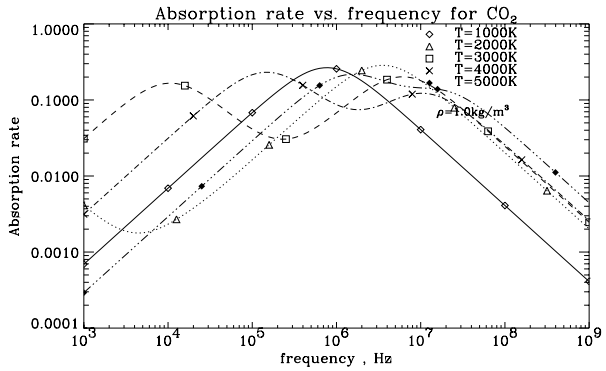
$\rho = 0.01\text{kg/m}^3$, Carbon dioxide

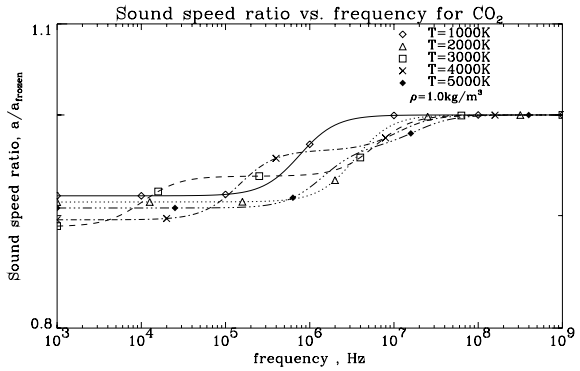




Sound absorption rate per wave length, $\frac{2\pi D_{real}}{D_{imag.}}$ Ratio of sound speeds, a/a_f

$\rho = 0.1\text{kg/m}^3$, Carbon dioxide

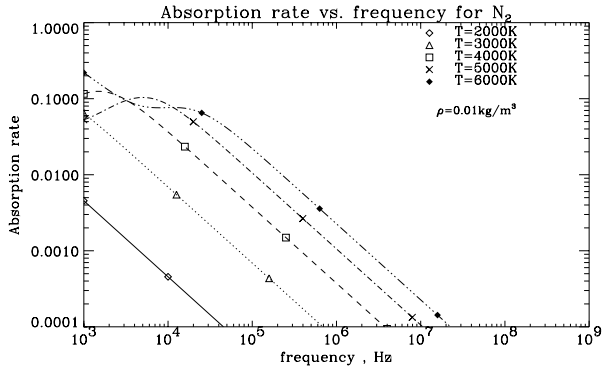


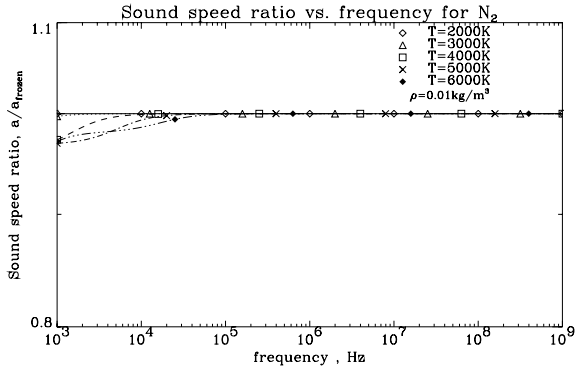


Sound absorption rate per wave length, $\frac{2\pi D_{real}}{D_{imag.}}$ Ratio of sound speeds, a/a_f

$\rho = 1.0\text{kg/m}^3$, Carbon dioxide

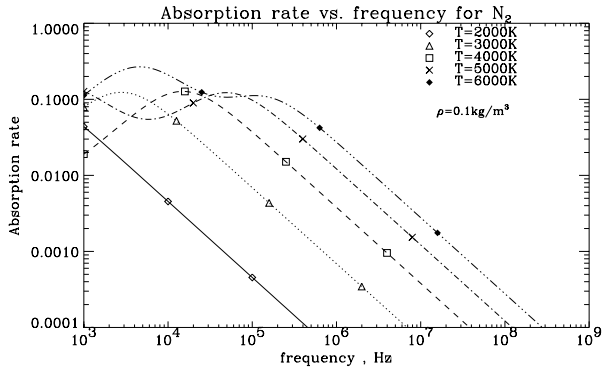
Figure 5: Estimated sound absorption rate and sound speeds ratio at constant density versus frequency for carbon dioxide.

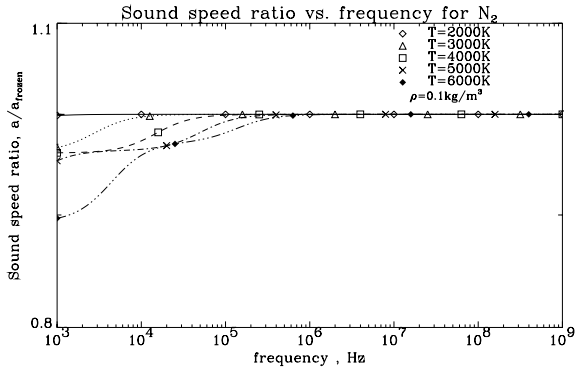




Sound absorption rate per wave length, $\frac{2\pi D_{real}}{D_{imag.}}$ Ratio of sound speeds, a/a_f

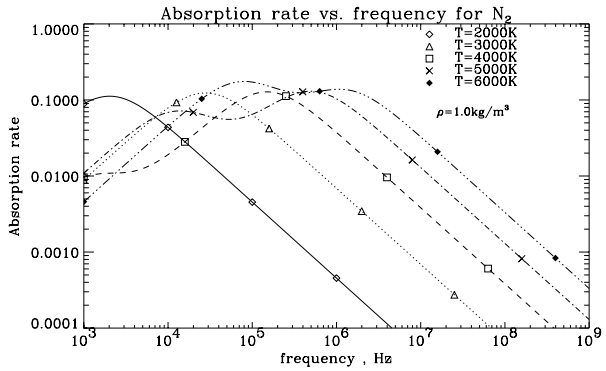
$\rho = 0.01\text{kg/m}^3$, Nitrogen

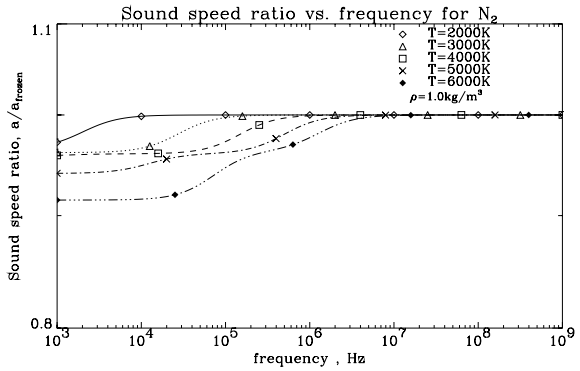




Sound absorption rate per wave length, $\frac{2\pi D_{real}}{D_{imag.}}$ Ratio of sound speeds, a/a_f

$\rho = 0.1\text{kg/m}^3$, Nitrogen





Sound absorption rate per wave length, $\frac{2\pi D_{real}}{D_{imag.}}$ Ratio of sound speeds, a/a_f

$\rho = 1.0\text{kg/m}^3$, Nitrogen

Figure 6: Estimated sound absorption rate and sound speeds ratio at constant density versus frequency for nitrogen.

Appendix

Source programs and an example input file for calculating sound absorption are presented here. "sprg.cpp" is a main program to output sound absorption rate per wave length and sound speed ratio as functions of frequency. "SRP_datd.cpp" is the library program for the procedure described in this memrandum. "SRP_datd.h" is the header file for "SRP_datd.cpp". "mathfj3.cpp" contains mathematical functions including inverse of matrix and complex operations. Only header file for it appears here.

1.Main program : sprg.cpp

```
//      program sprg.cpp (Sound Propagation in Relaxing Gases)
/* This program is to calculate the effect of relaxing gas effect on the
sound propagation at a rest equilibrium condition.
To compile, type "g++ -o sprg sprg.cpp SRP_dat.cpp ../mathfj3.cpp"
at "/home/keisuke/programs/Processing" directory.
*/
#include <stdio.h>
#include <string.h>
#include <iostream>
#include "SRP_dat.h"

void main() {
        int      ifs;
        float    w,af0,ae0,factaui,dwe,freq0;
        vector<float>    x0;
        int      i,j,k,stauv,iwend;
        RP_dat   rp;
        FP_dat   f1;
        string   fnm;
        char     Path[200],fnmo0[200],fnmo[200];
        FILE     *fpo;
        printf("select gas propertoes file..\n");
        printf("0:CO2(low T)\n1:CO2\n2:N2\n3:Air\n");
        scanf("%d",&ifs);
        strcpy(Path,"/home/keisuke/data/");
        if (ifs==0) {
                fnm     = "/home/keisuke/data/co2lowT.dat";
                stauv   = 1; // 1 for CO2, 0 for the others
        } else if (ifs==1) {
                fnm     = "/home/keisuke/data/co2.dat";
                stauv   = 1; // 1 for CO2, 0 for the others
        } else if (ifs==2) {
                fnm     = "/home/keisuke/data/n2.dat";
                stauv   = 0; // 1 for CO2, 0 for the others
        } else if (ifs==3) {
                fnm     = "/home/keisuke/data/air.dat";
                stauv   = 0; // 1 for CO2, 0 for the others
        }
        rp.readReact(fnm,f1.sp);
        printf("# of species = %d\n# of reactions = %d\n"
                ,f1.sp.size(),rp.nr);

        printf("Temperature..(K) ? ");  scanf("%f",&f1.T);
        printf("Density..(kg/m3) ? ");  scanf("%f",&f1.rhot);

        f1.De0  = 0.;
        f1.SetInit(rp);
        x0.resize(f1.ns);
        for (i=0; i<f1.ns; i++) x0[i]   = 0.;
        if (ifs<3)  {
                x0[0]   = 1.;
        } else if (ifs==3) {
                x0[0]   = 0.781;
                x0[1]   = 0.209;
                x0[2]   = 1.-x0[0]-x0[1];
        }
        f1.NO   = f1.InitNO(1,x0);
        f1.SetEq(rp);
        printf("Mw0=%f\n",f1.Mw0);
        for (i=0; i<f1.ns; i++)
                printf("%s : C[%d]=%f\n",f1.sp[i].HP,i,f1.C[i]);
        for (i=0; i<f1.ns; i++)
                printf("%s : Q[%d]=%e, lQ[%d]=%e, dlQ[%d]=%e\n",f1.sp[i].HP
                        ,i,f1.sp[i].Q,i,f1.sp[i].lQ
                        ,i,f1.sp[i].dlQ);
        for (i=0; i<rp.nr; i++)
                printf("Keq[%d]=%f, lKeq[%d]=%f, dlKeq[%d]=%f\n"
                        ,i,rp.Keq[i],i,rp.lKeq[i],i,rp.dlKeq[i]);
        f1.CalPropfrmQPRT(2,rp);
        rp.Rate(f1.T,f1.rhot,f1.N);
        for (i=0; i<rp.nr; i++) printf("kf[%d]=%e\n",i,rp.kf[i]);
        f1.CalDqEq(rp,stauv);
//      af0     = sqrt(f1.afz2);
//      ae0     = sqrt( (f1.dhdr +f1.dhdq*f1.dqdrE)
//                      /(1./f1.rhot -f1.dhdp -f1.dhdq*f1.dqdpE) );
//      printf("ae0/af0=%e, ae0=%e, af0=%e\n",ae0/af0,ae0,af0);
        printf("dhdr=%e, rest num.=%e\n1./r=%e, -dhdp=%e, rest den.=%e\n"
                ,f1.dhdr,f1.dhdq*f1.dqdrE
                ,1./f1.rhot,-f1.dhdp,-1.*f1.dhdq*f1.dqdpE);
        for (i=0; i<f1.nq; i++)
                printf("q[%d]=%e, dh/dq=%e, dq/dp=%e, dq/drho=%e\n"
                        ,i,f1.q[i],f1.dhdq[i],f1.dqdpE[i],f1.dqdrE[i]);
        printf("\n");
        for (i=0; i<rp.nre; i++)
                for (j=0; j<rp.nre; j++)
                        printf("dqqa[%d][%d]=%e\n",i,j,f1.dqqa[i][j]);
        for (i=0; i<f1.nv; i++)
                printf("dqqv[%d]=%e\n",i,1./f1.tauv[i]);
        factaui = 1 +(f1.dhdq*f1.dqdrE)/f1.dhdr;
        printf("fac-tau-i=%f\n",factaui);
        for (i=0; i<f1.nq; i++) {
                for (j=0; j<f1.nq; j++)
                        printf("%10.2e",f1.atau[i][j]);
                printf("\n");
        }

//      printf("factaui = %e, denomi = %e, af0=%e, "
//                      ,1. +(1./f1.dhdr)*(f1.dhdq*f1.dqdrE)
//                      ,1./f1.afz2 -(1./f1.dhdr)*(f1.dhdq*f1.dqdpE)
//                      ,af0 );
//      printf("a[0]/af0=%e\n"
//                      ,sqrt( ( 1. +(1./f1.dhdr)*(f1.dhdq*f1.dqdrE) )
//                      /( 1./f1.afz2 -(1./f1.dhdr)*(f1.dhdq*f1.dqdpE) ) )
//                      /af0 );
//      printf("\n");

        printf("frequency to start at..(Hz) ");
        scanf("%f",&freq0);
        printf("Total number of calculation points.. ");
        scanf("%d",&iwend);
        printf("# of points per decade.. ");
        scanf("%f",&dwe);       dwe = 1./dwe;
        printf("Output filename.. ");
        scanf("%s",fnmo0);
        sprintf(fnmo,"%s%s",Path,fnmo0);
        fpo     = fopen(fnmo,"w");
        printf("%13s%14s%14s\n","freq.","abs. per lamda","a/af0");
        fprintf(fpo,"%13s%14s%14s\n","freq","abslamda","aaf0");
        for (int iw=0; iw<iwend; iw++) {
                w       = freq0*pow(10.,dwe*iw)*(2.*pi_rC);
                f1.sprg(rp,w);
                printf("%13.6e%14.6e%14.6e\n"
                        ,w/(2.*pi_rC),f1.Atperl,f1.RtoAf);
                fprintf(fpo,"%13.6e%14.6e%14.6e\n"
                        ,w/(2.*pi_rC),f1.Atperl,f1.RtoAf);
        }

        fclose(fpo);
}
```

2.Header file : SRP_datd.h

```
//program SRP_datd.h
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <algorithm>
#include <vector>
#include <string>
#include "/home/keisuke/programs/mathfj3.h"
// using std::string;
// using std::vector;
const   double  R_Univ   = 8.31433;     // = k_Boltz x N_Avog
const   double  k_Boltz  = 1.38054e-23;
const   double  N_Avog   = 6.02252e23;
const   double  h_Planck = 6.6256e-34;
const   double  khN      = 6.26545e5;
        // khN actually means (k_Boltz^1.5)*(h_Plank^-3)*(N_Avog^-2.5)
const   double  eVtoJ    = 1.602e-19;
const   double  pi_rC    = 3.141592654;

class    SP_dat {       // class for species properties
        public:
        double           T,p,rhs,s,Rgas,thr // entropy is for per mole.
                        ,e,et,er,ee,es  // es : formation energy, per mole
                        ,de             // de = de/dT
                        ,em             // em : energy per unit mass
                        ,Cvt,Cvr,Mw,Avis,Bvis,Cvis,mu
                        ,Q,lQ,dlQ,d2lQ;
        int              nvmd,nelv,nofa;        // # of vib. and elect. mods
        vector<double>   thv,the,ev,evm; // characteristic temperatures
        vector<int>      vdg,edg;       // degeneracies
        char*            HP;
        double   evmEq(double); //!calculate equilibrium vibrational energies
                                // from temperature.
//      double   Tv();          //!calculate vibrational temperature using vib.
                                // energy in the member
        SP_dat&  CalEqProp(double); //!calculate energies in each modes and
                // entropy using partition functions. temperature as the input.
                // " calPartFunc" will be called in this procedure.
//      private:
        SP_dat&  CalPartFunc(double);
        double           QT,QR,QE,lQT,lQR,lQE
                        ,dlQT,dlQR,dlQE
                        ,d2lQT,d2lQR,d2lQE;
        vector<double>   QV,lQV,dlQV,d2lQV;
        // !! This QT is not actually the partition function, but that
        //      multiplied by rho/(Mw*n*N_Avog)
        // !! Consequently, for the calculation of entropy etc., log(QT)
        //      should be replaced by lQT -ln(rho / Mw), where Mw is molecular
        //      weight of the species and rho is density of the species. Though,
        //      rho(i)/Mw(i)=rho(mix)*n(i), where n(i) is mole per unit mass of
        //      mixture
};

class    RP_dat {       // class for reactions properties
        public:
        int              nr,nre;        // nr:# of reactions
        vector<int>      s3;            // s3:# of subreactions,  dim=nr
        vector<double>   Keq,kf,lKeq,dlKeq;             // dim=nr
        vector<vector<int> >     n0,n1,n2,n3; // n3:species' index of 3rd body
        vector<vector<double> >  Cfj,nfj,thfj; // React. rate param.s for subr.s
        vector<vector<double> >  na;

        RP_dat&  readGasProp(vector<SP_dat>&);
                //!read gas properties from the file of "/home/keisuke/programs/
                // Processing/gasesEqProp.dat" for each species.

        RP_dat&  readReact(string&,vector<SP_dat>&);
                //!read reactions informations. It will call the above procedure
                // of "readGasProp".

        RP_dat&  CalKeq(double,const vector<SP_dat>&);
                //!calculate eqilibrium constants from partion function of each
                // species. It is neccesary to call "sp.CalPartFunc" procedure
                // before this is called.

        RP_dat&  Rate(double,double,const vector<double> &);
                // input (Temperature, density, N:mole/kg)
                //!calculate rate coefficients for each reactions. mole
                // concentrations per unit mass, N, is supposed.
};

class    FP_dat {       // class for flow or mixture properties
        public:
        int              ns,nq,nv; // nq=nv+nre ns:# of species, nv:of v.mode
        vector<SP_dat>   sp;    // dim=ns
        vector<int>              idxv1,idxv2; // vib. index 1:species, 2:mode
        vector<double>           C,x,N,NO;      // dim=rp.ns
        vector<double>           q,dhdq,dqdpE,dqdrE,tauv;       // dim=nr+ns
        vector<vector<double> >  atau,dqqa;     // dim(atau)=(nre+nv)*(nre+nv)
        double           p,rhot,T,u,M,s,h,e,de,De0,mu,gam,Cpfz
                        ,T0,P0,h0,Mw0,Rmix,afz2,dhdp,dhdr
                        ,Atperl // attenuation per wave length
                        ,RtoAf; // Ratio to frozen speed of sound

        FP_dat&  SetInit(const RP_dat&); //!set dimensions of arrays
                // "RP_dat.readReact" is supposed to be called in advance.
                // De0 is the energy to adjust the zero point. It should be
                // determined before "CalPropfrmPRQ" is called.

        FP_dat&  CalPropfrmQPRT(int, const RP_dat&);
                // calculates N,C,x,Mw0,Rmix,T,energies,dhdp,dhdr
                // and frozen sound of speed. This procedure is assuming
                // equilibrium conditioin except vibrational thermal modes.
                // This means translational and rotational temperatures are in
                // equilibrium and electronic temperatures is frozen.

        FP_dat&  CalVISC(); //!calculate viscousities from temperature
                // and mole fractions(x).

        FP_dat&  SetEq(RP_dat&); //!input:Temperature & density of mixture
                // This procedure uses Temperature and density as input and
                // calculates N (moles per unit mass of mixture), mole
                // fractions, mass fractions, density of each species, pressure
                // and molecular weight of mixture. q-vector is also calculated.

        FP_dat&  CalDqEq (const RP_dat&, int); // using T,rhot,p,reaction
                // parameters, N and species informations, calculates matrices
                // and vectors of derivatives, time constants. integer stauv is
                // the sign for the tau formula. stauv=1 for CO2 gas and others
                // for the others.
                // RP_dat::Rate(T,rhot,N) is supposed to be called in advance.

        FP_dat&  CxfrmN(); // using N and density of mixture,
                // calcilates Mw0,C,x and densities of each species.

        vector<double>   InitN0(int, const vector<double>&);
                // calculates N, consentrations in moles
                // per unit mass of mixture
                // is=0 : from mass fractions, C
                // is=1 : from mole fractions, x

        FP_dat& FP_dat::sprg(const RP_dat&, double);
                // This is for estimating the effect of relaxing gas on the
                // sound propergation at a rest equilibrium condition.
                // It calculates Atperl and RtoAf. 2nd parameter is radial
                // frequency in unit of rad/s.

        private:
        FP_dat&  Nevm(const RP_dat&); // from q-vector, calculates N and
                // vibrational energies of each species per unit mass.
};
```

3.Library program : SRP_datd.cpp

```
 //program SRP_datd.cpp
#include "SRP_datd.h"

double   SP_dat::evmEq(double T) {
        double   fevm;
        fevm    = 0.;
        for (int i=0; i<nvmd; i++)
                fevm += vdg[i]*(R_Univ/Mw)*thv[i]/(exp(thv[i]/T)-1);
        return fevm;
}
/*
double   SP_dat::Tv() {
        double   Tvs,ev1,dev;
        Tvs     = T;
        do {
                ev1     = evmEq(Tvs);
                dev     = 0.;
                for (int i=0; i<nvmd; i++)
                        dev += vdg[i]*(R_Univ/Mw)*exp(thv[i]/Tvs)
                                *pow( thv[i]/( Tvs*(exp(thv[i]/Tvs)-1) ),2 );
                Tvs     -= (ev1-evm)/dev;
        } while ( fabs(1.-ev1/evm)>1.e-4 );
        return Tvs;
}
*/
SP_dat&  SP_dat::CalPartFunc(double T) {
        double   QeN,dQeN,d2QeN;
        int      i;
        QT      = khN*pow( 2.*pi_rC*Mw*T,1.5);
        lQT     = log(QT);
        dlQT    = 1.5/T;        d2lQT   = -1.5/(T*T);
        QV.resize(nvmd);        lQV.resize(nvmd);
        dlQV.resize(nvmd);      d2lQV.resize(nvmd);
        if ( nofa>1 ) {
                if ( nofa==2 ) {
                        QR      = T/thr;
                        lQR     = log(QR);
                        dlQR    = 1./T;         d2lQR   = -1./(T*T);
                } else {
                        QR      = pow(T/thr,1.5);
                        lQR     = 1.5*log(T/thr);
                        dlQR    = 1.5/T;        d2lQR   = -1.5/(T*T);
                }
                for (int j=0; j<nvmd; j++) {
                        lQV[j]  = -vdg[j]*log(1.-exp(-thv[j]/T));
                        dlQV[j] =  vdg[j]*(thv[j]/(T*T))/(exp(thv[j]/T)-1.);
                        d2lQV[j] = (dlQV[j]/T)
                                 *(-2. +(thv[j]/T)/(1.-exp(-thv[j]/T)));
                        QV[j]   =  exp(lQV[j]);
                }
        } else {
                QR      = 1.;   lQR     = 0.;   dlQR    = 0.;   d2lQR   = 0.;
                for (int j=0; j<nvmd; j++) {
                        lQV[j]  = 0.;
                        dlQV[j] = 0.;
                        d2lQV[j]= 0.;
                        QV[j]   = 1.;
                }
        }

        QE      = edg[0];       dQeN    = 0.;   d2QeN   = 0.;
        for (int j=1; j<nelv; j++) {
                QeN     = edg[j]*exp(-the[j]/T);
                QE      += QeN;
                dQeN    += QeN*the[j];
                d2QeN   += QeN*the[j]*(-2.*T +the[j]);
        }
        lQE     = log(QE);
        if (fabs(QE)>1.e-10) {
                dlQE    = ( dQeN/QE )*( 1./(T*T) );
                d2lQE   = -dlQE*dlQE +(d2QeN/(T*T*T*T))/QE;
        } else {
                dlQE    = 0.;
                d2lQE   = 0.;
        }

        Q       = QT*QR*QE;     for (i=0; i<nvmd; i++) Q *= QV[i];
        lQ      = lQT +lQR +lQE;  for (i=0; i<nvmd; i++) lQ += lQV[i];
        dlQ     = dlQT +dlQR +dlQE;  for (i=0; i<nvmd; i++) dlQ += dlQV[i];
        d2lQ    = d2lQT +d2lQR +d2lQE;
        for (i=0; i<nvmd; i++) d2lQ += d2lQV[i];
        return *this;
}

SP_dat&  SP_dat::CalEqProp(double T) {
        CalPartFunc(T);
        s       = R_Univ*(lQ +1. +T*dlQ -log(rhs/Mw));
                // remenber, symbol "Q" here is NOT a partition function, Q,
                // but Q / N_Avog x V.
        e       = R_Univ*T*T*dlQ;       de = R_Univ*T*(2.*dlQ +T*d2lQ);
        et      = R_Univ*T*T*dlQT;
        er      = R_Univ*T*T*dlQR;
        ee      = R_Univ*T*T*dlQE;
        for (int i=0; i<nvmd; i++) {
                ev[i]   = R_Univ*T*T*dlQV[i];
                evm[i]  = ev[i]/Mw;
        }
        em      = e/Mw;
        return *this;
}

RP_dat&  RP_dat::readGasProp(vector<SP_dat>& sp) {
        vector<double>   At,Bt,Ct,mwt,thrt,shjap;
        vector<vector<double> > thvt,Elj;
        vector<int>      nvmdt,nelvt,nofat;
        vector<vector<int> >     vdgt,edgt;
        int      i,j,k,n,ns;
        char     fn[200],dchr;
        vector<char*>   HPt;
        FILE     *fp;
        ns      = sp.size();
        strcpy(fn,"/home/keisuke/programs/Processing/gasesEqProp.dat");
        fp      = fopen(fn,"r");
        fscanf(fp,"%d",&n);
        for (i=0; i<2; i++) do { fscanf(fp,"%c",&dchr); } while (dchr!='\n');
        mwt.resize(n);  At.resize(n);   Bt.resize(n);   Ct.resize(n);
        HPt.resize(n);  thrt.resize(n); vdgt.resize(n); thvt.resize(n);
        Elj.resize(n);  nvmdt.resize(n);        edgt.resize(n);
        nofat.resize(n);        nelvt.resize(n);        shjap.resize(n);
        for (i=0; i<n; i++) {
                HPt[i]  = (char*)malloc(sizeof(char)*5);
                fscanf(fp,"%s%lf%d%le%le%lf%lf%lf"
                        ,HPt[i],&mwt[i],&nofat[i],&thrt[i],&shjap[i]
                        ,&At[i],&Bt[i],&Ct[i]);
                do {    fscanf(fp,"%c",&dchr); } while (dchr!='\n');
                fscanf(fp,"%d",&nvmdt[i]);
                vdgt[i].resize(nvmdt[i]);       thvt[i].resize(nvmdt[i]);
                for (k=0; k<nvmdt[i]; k++)
                        fscanf(fp,"%d %lf",&vdgt[i][k],&thvt[i][k]);
                do {    fscanf(fp,"%c",&dchr); } while (dchr!='\n');
                fscanf(fp,"%d",&nelvt[i]);
                edgt[i].resize(nelvt[i]);       Elj[i].resize(nelvt[i]);
                for (k=0; k<nelvt[i]; k++)
                        fscanf(fp,"%d %lf",&edgt[i][k],&Elj[i][k]);
                do {    fscanf(fp,"%c",&dchr); } while (dchr!='\n');
        }
        fclose(fp);
        for (i=0; i<ns; i++) {
                for (j=0; j<n; j++)
                        if ( strcmp(sp[i].HP,HPt[j])==0 ) {
                                sp[i].Mw        = mwt[j]/1.e3;
                                sp[i].nofa      = nofat[j];
                                sp[i].thr       = thrt[j];
                                sp[i].es        = shjap[j];
                                sp[i].Avis      = At[j];
                                sp[i].Bvis      = Bt[j];
                                sp[i].Cvis      = Ct[j];
                                sp[i].nvmd      = nvmdt[j];
                                sp[i].vdg.resize(sp[i].nvmd);
                                sp[i].thv.resize(sp[i].nvmd);
                                sp[i].ev.resize(sp[i].nvmd);
                                sp[i].evm.resize(sp[i].nvmd);
                                for (k=0; k<sp[i].nvmd; k++) {
                                        sp[i].vdg[k] = vdgt[j][k];
                                        sp[i].thv[k] = thvt[j][k];
                                }
                                sp[i].nelv      = nelvt[j];
                                sp[i].the.resize(sp[i].nelv);
                                sp[i].edg.resize(sp[i].nelv);
                                for (k=0; k<sp[i].nelv; k++) {
                                        sp[i].edg[k] = edgt[j][k];
                                        sp[i].the[k] = Elj[j][k]/R_Univ;
                                }
                                sp[i].Rgas      = R_Univ/sp[i].Mw;
                                sp[i].Cvt       = 1.5*R_Univ;
                                if (sp[i].nofa>1)
                                        sp[i].Cvr       = R_Univ;
                                else    sp[i].Cvr       = 0.;
                        }
        }
        return *this;
}

RP_dat& RP_dat::readReact(string& fns, vector<SP_dat>& sp) {
        int      i,j,ns;
        const    char*   fnm    = fns.c_str();
        FILE     *fp;
        char     dchar;
        fp      = fopen(fnm,"r");
        fscanf(fp,"%d %d",&ns,&nr);
        sp.resize(ns);
        Keq.resize(nr);         lKeq.resize(nr);        dlKeq.resize(nr);
        kf.resize(nr);          s3.resize(nr);
        n0.resize(nr);  n1.resize(nr);  n2.resize(nr);  n3.resize(nr);
        Cfj.resize(nr); nfj.resize(nr); thfj.resize(nr);

        for (i=0; i<ns; i++) {
                sp[i].HP = (char*)malloc(sizeof(char)*4);
                fscanf(fp,"%s",sp[i].HP);
        }
        do { fscanf(fp,"%c",&dchar); } while (dchar!='\n');
        readGasProp(sp);
        for (i=0; i<nr; i++) {
                n0[i].resize(ns);       n1[i].resize(ns);
                n2[i].resize(ns);
                for (j=0; j<ns; j++)    fscanf(fp,"%d",&n1[i][j]);
                for (j=0; j<ns; j++) {
                        fscanf(fp,"%d",&n2[i][j]);
                        n0[i][j] = n2[i][j] -n1[i][j];
                }
                fscanf(fp,"%d",&s3[i]);
                n3[i].resize(s3[i]);    Cfj[i].resize(s3[i]);
                nfj[i].resize(s3[i]);   thfj[i].resize(s3[i]);
                if (s3[i]>1)    for (j=0; j<s3[i]-1; j++)
                        fscanf(fp,"%d%lf%lf%lf",&n3[i][j],&Cfj[i][j]
                                        ,&nfj[i][j],&thfj[i][j]);
```

```
                j       = s3[i]-1;
                fscanf(fp,"%*d%lf%lf%lf",&Cfj[i][j],&nfj[i][j]
                                                ,&thfj[i][j]);
                n3[i][j]        = -1;
        }
        if (nr>1) {
                fscanf(fp,"%d",&nre);
                na.resize(nre);
                for (i=0; i<nre; i++)
                        na[i].resize(nr);
                for (i=0; i<nr; i++)
                        for (j=0; j<nre; j++)
                                fscanf(fp,"%lf",&na[j][i]);
        } else {
                nre     = 1;
                na.resize(1);
                na[0].resize(1);
                na[0][0] = 1.;
        }
        fclose(fp);
        return *this;
}

RP_dat&  RP_dat::CalKeq(double T, const vector<SP_dat> &sp) {
        int      i,j,ns;
        ns      = sp.size();
        for (i=0; i<nr; i++) {
                Keq[i]  = 1.;
                lKeq[i] = 0.;
                dlKeq[i]        = 0;
                for (j=0; j<ns; j++) {
                        Keq[i]  *= pow( sp[j].Q*exp(-sp[j].es/(R_Univ*T))
                                                                ,n0[i][j]);
                        lKeq[i] += n0[i][j]*( sp[j].lQ
                                        -sp[j].es/(R_Univ*T) );
                        dlKeq[i]+= n0[i][j]*( sp[j].dlQ
                                                +(sp[j].es/(R_Univ*T*T)) );
        }       }
        return *this;
}

RP_dat&  RP_dat::Rate(double T, double rhot, const vector<double> &N) {
        double   sN,sN1;
        int      i,j,ns;
        ns      = N.size();
        sN      = 0.;
        for (i=0; i<ns; i++)    sN += N[i];
        for (i=0; i<nr; i++)
                if (s3[i]>1) {
                        kf[i] = 0.;     sN1 = 0.;
                        for (j=0; j<s3[i]-1; j++) {
                                kf[i] += Cfj[i][j]*pow(T,nfj[i][j])
                                        *exp(-thfj[i][j]/T)
                                        *rhot*N[n3[i][j]];
                                sN1     += N[n3[i][j]];
                        }
                        j = s3[i]-1;
                        kf[i]   += rhot*(sN-sN1)*Cfj[i][j]*pow(T,nfj[i][j])
                                        *exp(-thfj[i][j]/T);
                } else {
                        j       = 0;
                        kf[i]   = Cfj[i][j]*pow(T,nfj[i][j])
                                        *exp(-thfj[i][j]/T);
                }
        return *this;
}

FP_dat&  FP_dat::SetInit(const RP_dat& rp) {
        int i,j,inv;
        ns      = sp.size();
        nv      = 0;
        for (i=0; i<ns; i++)
                for (j=0; j<sp[i].nvmd; j++)
                        nv++;
        idxv1.resize(nv);       idxv2.resize(nv);       inv     = 0;
        for (i=0; i<ns; i++)
                for (j=0; j<sp[i].nvmd; j++) {
                        idxv1[inv] = i;
                        idxv2[inv] = j;
                        inv++;
                }
        nq      = rp.nre +nv;
        N.resize(ns);   N0.resize(ns);  C.resize(ns);   x.resize(ns);
        q.resize(nq);           dhdq.resize(nq);
        dqdpE.resize(nq);       dqdrE.resize(nq);
        atau.resize(nq);        for (i=0; i<nq; i++) atau[i].resize(nq);
        return *this;
}

FP_dat&  FP_dat::CalPropfrmQPRT(int isp, const RP_dat& rp) {
        int      i,j;
        Nevm(rp);       // N and evm are from q-vector.
        MW0     = 0.;
        for (i=0; i<ns; i++)    MW0 += N[i];
        MW0     = 1./MW0;
        Rmix    = R_Univ/MW0;
        if (isp==0)             p = (rhot*Rmix)*T;
        else if (isp==1)        rhot = p/(Rmix*T);
        else                    T = p/(rhot*Rmix);
        for (i=0; i<ns; i++) {
                sp[i].T = T;
                sp[i].p = p;
                sp[i].CalEqProp(T);     // !Here, evm is overwrited to
                                        // equilibrium values.
        }
        Nevm(rp);       // Replace evm again. Usually it is not needed.
        e       = De0;  de = 0.;
        for (i=0; i<ns; i++) {
                C[i]    = sp[i].Mw*N[i];
                x[i]    = MW0*N[i];
                sp[i].rhs       = rhot*C[i];
                for (j=0; j<sp[i].nvmd; j++)
                        sp[i].ev[j]     = sp[i].evm[j] *sp[i].Mw;
                sp[i].e = sp[i].et  +sp[i].er +sp[i].ee;
                for (j=0; j<sp[i].nvmd; j++)
                        sp[i].e += sp[i].ev[j];
                sp[i].em        = sp[i].e  /sp[i].Mw;
                e       += N[i]*( sp[i].e + sp[i].es );
                de      += N[i]*sp[i].de;
        }
        h       = e +p/rhot;
        Cpfz    = de +Rmix;
        dhdr    = -p/(rhot*rhot);
        dhdp    = 1./rhot;
        for (i=0; i<ns; i++) {
                dhdr    -= N[i]*(sp[i].Cvt+sp[i].Cvr)*T/rhot;
                dhdp    += N[i]*(sp[i].Cvt+sp[i].Cvr)*T/p;
        }
        afz2    = dhdr/(1./rhot -dhdp);
        gam     = 1. +Rmix/de;
        return *this;
}

FP_dat& FP_dat::CalVISC() {
        double   phi;
        int              k;
        for (k=0; k<ns; k++)
                sp[k].mu = 0.1*exp( (sp[k].Avis*log(T)
                                                +sp[k].Bvis)*log(T)
                                        +sp[k].Cvis );
        mu = 0.;
        for (k=0; k<ns; k++) {
                phi = 0.;
                for (int l=0; l<ns; l++) {
                        phi += x[l]*pow( 1.+sqrt(sp[k].mu/sp[l].mu)
                                        *pow(sp[l].Mw/sp[k].Mw,0.25),2.0)
                                        /sqrt( 8.*(1.+sp[k].Mw/sp[l].Mw));
                }
                mu += x[k]*sp[k].mu/phi;
        }
        return *this;
}

FP_dat&  FP_dat::SetEq(RP_dat& rp) {
        vector<vector<double> > df,idf;
        vector<double>  f,a,da;
        vector<double>  Nt;
        double   det;
        double   fda,dis0,dif0;
        int      k,ni,nj,nre,mi,nofi;
        nre     = rp.nre;
        df.resize(nre); for (ni=0; ni<nre; ni++) df[ni].resize(nre);
        f.resize(nre);  da.resize(nre); a.resize(nre);
        Nt.resize(ns);
        for (ni=0; ni<ns; ni++)
                sp[ni].CalEqProp(T);
        rp.CalKeq(T,sp);
        for (ni=0; ni<nre; ni++) {
                a[ni]   = 0.;
                da[ni]  = 0.;
        }
        nofi = 0;
        do {
                N       = N0;
                for (k=0; k<ns; k++)
                        for (ni=0; ni<nre; ni++)
                                N[k] += rp.n0[ni][k]*a[ni];
//              for (k=0; k<ns; k++)
//                      printf("N[%d]=%e, ",k,N[k]);
//              printf("\n");
                for (ni=0; ni<nre; ni++) {
                        for (nj=0; nj<nre; nj++) {
                          df[ni][nj]    = 0.;
                            for (k=0; k<ns; k++)
                              if (rp.n0[ni][k]*rp.n0[nj][k]!=0)
                                df[ni][nj] += rp.n0[ni][k]
                                                *rp.n0[nj][k]/N[k];
                        }
                        f[ni]   = 0.;
                        for (k=0; k<ns; k++)
                                f[ni] += rp.n0[ni][k]*log(rhot*N[k]);
                        f[ni]   -= rp.lKeq[ni];
                }
                InvMtx(df,idf,det);
//              printf("det=%e\n\n",det);
                da      = -1.*(idf*f);
                fda     = 0.5;
                Nt      = N0;
                for (k=0; k<ns; k++)    for (ni=0; ni<nre; ni++)
                        Nt[k] += rp.n0[ni][k]*(a+fda*da)[ni];
                while ( Minelement(Nt,&mi)<=0. ) {
                        fda     *= 0.5;
                        Nt      = N0;
                        for (k=0; k<ns; k++)    for (ni=0; ni<nre; ni++)
                                Nt[k] += rp.n0[ni][k]*(a+fda*da)[ni];
                }
                dif0    = vabs(da);
                a       = a +fda*da;
                dis0    = 0.;
                for (ni=0; ni<nre; ni++)
                        dis0    += pow( f[ni]/rp.lKeq[ni],2 );
                dis0    = sqrt(dis0);
//              for (int i=0; i<ns; i++) printf("%f ",Nt[i]);
//              printf("dis0=%e, dif0=%e, fda=%e\n",dis0,dif0,fda);
```

```
//              scanf("%*c");
                nofi++;
//      } while ( ( dis0>1.e-6 )&&(dif0>1.e-6) );
        } while ( ( dis0>1.e-6 )&&( nofi<500 ) );
        if (dis0>1.e-6) printf("Disagreement (ratio) is %f\n",dis0);
//      } while ( dif0>1.e-8 );
        for (ni=0; ni<rp.nre; ni++)
                q[ni]   = 0.;
        for (ni=0; ni<nre; ni++)
                q[ni]   = a[ni];
        for (ni=0; ni<nv; ni++)
                q[nre+ni]       = sp[idxv1[ni]].evm[idxv2[ni]];
        N = Nt;
        CxfrmN();
        p       = rhot*(R_Univ/Mw0)*T;
        return *this;
}

FP_dat&  FP_dat::CalDqEq (const RP_dat& rp, int stauv) {
        int             i,j,s,nr,nre;
        double   det,devT,redM,sNtvs,tauvs,thvT,ata,bta;
        vector<vector<double> > taa,aa,iaa,snn0n,dqqan;
        vector<double>   ap,ar,sn0,dap,dar,dTa,devp,devr,pin1;
        atau.resize(nq);
        for (i=0; i<nq; i++) {
                atau[i].resize(nq);
                for (j=0; j<nq; j++) atau[i][j] = 0.;
        }
        nr = rp.nr;             nre = rp.nre;
        sn0.resize(nr);         snn0n.resize(nr);
        dqqan.resize(nr);       pin1.resize(nr);        taa.resize(nre);
        ap.resize(nre);         ar.resize(nre);         aa.resize(nre);
        dap.resize(nre);        dar.resize(nre);        dqqa.resize(nre);
        for (i=0; i<nr; i++) {
                snn0n[i].resize(nr);
                dqqan[i].resize(nr);
        }
        for (i=0; i<nre; i++) {
                dqqa[i].resize(nre);
                aa[i].resize(nre);
                taa[i].resize(ns);
        }
        devp.resize(nv);        devr.resize(nv);        tauv.resize(nv);
        for (i=0; i<nr; i++) {
                sn0[i]  = 0.;   pin1[i] = 1.;
                for (s=0; s<ns; s++) {
                        sn0[i]  += rp.n0[i][s];
                        pin1[i] *= pow(rhot*N[s],rp.n1[i][s]);
                }
                for (j=0; j<nr; j++) {
                        snn0n[i][j]     = 0.;
                        for (s=0; s<ns; s++)
                                if ( rp.n0[i][s]*rp.n0[j][s]!=0 )
                                        snn0n[i][j] += rp.n0[j][s]
                                                        *rp.n0[i][s]/N[s];
                }
        }
        for (i=0; i<nre; i++) {
                ap[i]   = T*rp.dlKeq[i]/p;
                ar[i]   = -(T*rp.dlKeq[i] +sn0[i])/rhot;
                for (j=0; j<ns; j++) {
                        if (rp.n0[i][j]==0)
                                taa[i][j] = T*rp.dlKeq[i]*Mw0;
                        else
                                taa[i][j] = T*rp.dlKeq[i]*Mw0
                                                +rp.n0[i][j]/N[j];
                }
        }
        for (i=0; i<nre; i++) for (j=0; j<nre; j++) {
                aa[i][j] = 0.;
                for (int k=0; k<ns; k++) aa[i][j] += taa[i][k]*rp.n0[j][k];
        }
        InvMtx(aa,iaa,det);

        dap     = iaa*ap;
        dar     = iaa*ar;
        for (i=0; i<nre; i++)
                for (j=0; j<nre; j++) {
                        dqqa[i][j] = 0.;
                        for ( int k=0; k<nr; k++)
                                dqqa[i][j] += rp.na[i][k]*(rp.kf[k]/rhot)
                                                *snn0n[k][j]*pin1[k];
                }
        for (i=0; i<nre; i++) {
                dqdpE[i]        = dap[i];
                dqdrE[i]        = dar[i];
                dhdq[i]         = 0.;
                for (j=0; j<ns; j++)
                        dhdq[i] += rp.n0[i][j]*( sp[j].e +sp[j].es )
                                  -N[j]*sn0[i]*(sp[j].Cvt+sp[j].Cvr)*Mw0*T;
                for (j=0; j<nre; j++)
                        atau[j][i]      = dqqa[j][i];
        }
        for (i=0; i<nv; i++) {
                thvT    = sp[idxv1[i]].thv[idxv2[i]]/T;
                devT    = q[nre+i]*thvT/(1-exp(-thvT));
                devp[i] = 1./p;         devr[i] = -1./rhot;
                for (j=0; j<nre; j++) {
                        devp[i] -= Mw0*sn0[j]*dap[j];
                        devr[i] -= Mw0*sn0[j]*dar[j];
                }
                devp[i] *= devT;        devr[i] *= devT;
                dqdpE[nre+i]    = devp[i];
                dqdrE[nre+i]    = devr[i];
                dhdq[nre+i]     = sp[idxv1[i]].Mw*N[idxv1[i]];
                sNtvs   = 0.;
                for (j=0; j<ns; j++) {
                        if (stauv==1) {
                                ata  = 36.5;
                                bta  = -0.0193;
                        } else {
                                redM = sp[idxv1[i]].Mw*sp[j].Mw
                                         /( sp[idxv1[i]].Mw+sp[j].Mw);
                                ata  = 0.0367*sqrt(redM)
                                        *pow(sp[idxv1[i]].thv[idxv2[i]],4./3.);
                                bta  = 0.08435*pow( redM,0.25 );
                        }
                        tauvs   = (1./(p*9.86923e-6))
                                        *exp(ata*(pow(T,-1./3.)-bta) -18.42 );
                        sNtvs   += N[j]/tauvs;
                }
                tauv[i] = 1./(Mw0*sNtvs);
                atau[nre+i][nre+i]      = 1./tauv[i];
//              printf("tau-v[i]=%13.4e\n",tauv[i]);
        }
        return *this;
}

FP_dat&  FP_dat::CxfrmN() {
        int      i;
        double   iMw0;
        iMw0    = 0.;
        for (i=0; i<ns; i++)
                iMw0    += N[i];
        Mw0             = 1./iMw0;
        for (i=0; i<ns; i++) {
                C[i]    = sp[i].Mw*N[i];
                x[i]    = Mw0*N[i];
                sp[i].rhs       = rhot*C[i];
        }
        return *this;
}

FP_dat&  FP_dat::Nevm(const RP_dat& rp) {
        int      i,j,nr;
        nr      = q.size() -nv;
        for (j=0; j<ns; j++) {
                N[j]    = N0[j];
                for (i=0; i<nr; i++)
                        N[j]    += q[i]*rp.n0[i][j];
        }
        for (i=0; i<nv; i++)
                sp[idxv1[i]].evm[idxv2[i]]      = q[nr+i];
        return *this;
}

vector<double>   FP_dat::InitN0(int is, const vector<double>& C0i) {
        vector<double>  N0t,C0;
        double   minN = 1.e-10;
        double   sc,npM;
        int      i;
        if (C0i.size()<ns)
                printf("INPUt C0 size is not approproate!\n");
        N0t.resize(ns);         C0.resize(C0i.size());
        sc      = 0.;
        for (i=0; i<min(ns,(int)C0i.size()); i++)       sc += C0i[i];
        for (i=0; i<min(ns,(int)C0i.size()); i++)       C0[i] = C0i[i]/sc;
        if (C0i.size()<ns)
                for (i=0; i<ns; i++)    C0[i] = 1.e-30;
        if (is==0) {
                npM = 0.;
                for (i=0; i<ns-1; i++) {
                        N0t[i]  = C0[i]/sp[i].Mw;
                        if (minN>N0t[i]) N0t[i] = minN;
                        npM += N0t[i];
                }
                Mw0= 1./npM;
        } else {
                Mw0     = 0;
                for (i=0; i<ns; i++) Mw0 += C0[i]*sp[i].Mw;
                for (i=0; i<ns; i++) {
                        N0t[i]  = C0[i]/Mw0;
                        if (minN>N0t[i]) N0t[i] = minN;
                }
        }
        if (minN>N0t[ns-1]) N0t[ns-1] = minN;
        return N0t;
}

FP_dat& FP_dat::sprg(const RP_dat &rp, double w) {
        double   af0;
        vector<vector<complex<double> > > iwIAa,iwIAai,iwIAi,B
                                        ,dhdqB,Mnum,Mden;
        complex<double> Num1,Den1,D,cdet;
        int      i,j;
        af0     = sqrt(afz2);
        iwIAa.resize(rp.nre);
        for (i=0; i<rp.nre; i++) {
                iwIAa[i].resize(rp.nre);
                for (j=0; j<rp.nre; j++)
                        if (i==j)
                                iwIAa[i][j] = c_i*w +dqqa[i][j];
                        else
                                iwIAa[i][j] = dqqa[i][j];
        }
        InvMtx(iwIAa,iwIAai,cdet);
//      printf("det= %e +( %e )i, ",cdet.real(),cdet.imag());
        iwIAi.resize(nq);
        for (i=0; i<nq; i++) {
                iwIAi[i].resize(nq);
                for (j=0; j<nq; j++) {
                        iwIAi[i][j]     = 0.;
                        if ( (i<rp.nre)&&(j<rp.nre) )
                                iwIAi[i][j] = iwIAai[i][j];
```

```
                else if (i==j)
                        iwIAi[i][j] = 1./(c_i*w
                                +1./tauv[i-rp.nre]);
        }
}

B       = iwIAi*(((complex<double>)1.)*atau);

dhdqB   = (cd_1*Tmat(dhdq))*B;

        Mnum    = dhdqB*(cd_1*VtoM(dqdpE));
        Mden    = dhdqB*(cd_1*VtoM(dqdrE));
        Num1    = -w*w*( 1./afz2 -(1./dhdr)*Mnum[0][0] );
        Den1    = 1. +(1./dhdr)*Mden[0][0];
        D       = sqrt( Num1/Den1 );
        Atperl  = 2.*pi_rC*D.real()/D.imag();
        RtoAf   = w/(af0*D.imag());
        return *this;
}
```

4.Header file : mathfj3.h

```
//      mathfj3.h
#include <stdio.h>
#include <iostream>
#include <numeric>
#include <vector>
#include <complex>
// using std::abs;
// using std::vector;
// using std::complex;
const   complex<float>  c_i = complex<float>(0,1);
const   complex<float>  c_1 = complex<float>(1,0);
const   complex<double> cd_i = complex<double>(0,1);
const   complex<double> cd_1 = complex<double>(1,0);

        void     InvMtx(const vector<vector<double> >&, vector<vector<double> >&
                        , double&);
        void     ludcmp(vector<vector<double> >&, vector<int>&, double *);
        void     lubksb(vector<vector<double> >&, vector<int>&, vector<double>&);
        void     InvMtx(const vector<vector<float> >&, vector<vector<float> >&
                        , float&);
        void     ludcmp(vector<vector<float> >&, vector<int>&, float *);
        void     lubksb(vector<vector<float> >&, vector<int>&, vector<float>&);
        void     InvMtx(const vector<vector<complex<float> > > &
                , vector<vector<complex<float> > > &, complex<float>&);
        void     ludcmp(vector<vector<complex<float> > > &
                , vector<int>&, float *);
        void     lubksb(vector<vector<complex<float> > > &
                , vector<int>&, vector<complex<float> > &);
        void     InvMtx(const vector<vector<complex<double> > > &
                , vector<vector<complex<double> > > &,complex<double>&);
        void     ludcmp(vector<vector<complex<double> > > &
                , vector<int>&, double *);
        void     lubksb(vector<vector<complex<double> > > &
                , vector<int>&, vector<complex<double> > &);
        vector<float>           operator+ (const vector<float> &
                                ,const vector<float> &);
        vector<double>          operator+ (const vector<double> &
                                ,const vector<double> &);
        vector< complex<float> > operator+ (const vector< complex<float> > &
                                ,const vector< complex<float> > &);
        vector< complex<double> > operator+ (const vector< complex<double> > &
                                ,const vector< complex<double> > &);
        vector<float>           operator- (const vector<float> &
                                ,const vector<float> &);
        vector<double>          operator- (const vector<double> &
                                ,const vector<double> &);
        vector< complex<float> > operator- (const vector< complex<float> > &
                                ,const vector< complex<float> > &);
        vector< complex<double> > operator- (const vector< complex<double> > &
                                ,const vector< complex<double> > &);
        vector<float>           operator* (float,const vector<float> &);
        vector<double>          operator* (double,const vector<double> &);
        vector< complex<float> > operator* (complex<float>
                                ,const vector< complex<float> > &);
        vector< complex<double> > operator* (complex<double>
                                ,const vector< complex<double> > &);
        float                   operator* (const vector<float> &
                                ,const vector<float> &);
        double                  operator* (const vector<double> &
                                ,const vector<double> &);
        complex<float>          operator* (const vector<complex<float> > &
                                ,const vector<complex<float> > &);
        complex<double>         operator* (const vector<complex<double> > &
                                ,const vector<complex<double> > &);
        float                   vabs(const vector<float> &);
        double                  vabs(const vector<double> &);
        float                   vabs(const vector<complex<float> > &);
        complex<float>          conjug(const complex<float>&);
        vector<complex<float> > conjug(const vector<complex<float> >& );
        vector<complex<double> > conjug(const vector<complex<double> >& );
        vector<vector<float> >  operator+ (const vector<vector<float> > &
                                ,const vector<vector<float> > &);
        vector<vector<double> > operator+ (const vector<vector<double> > &
                                ,const vector<vector<double> > &);
        vector<vector<float> >  operator- (const vector<vector<float> > &
                                ,const vector<vector<float> > &);
        vector<vector<double> > operator- (const vector<vector<double> > &
                                ,const vector<vector<double> > &);
        vector<float>           operator* (const vector<vector<float> > &
                                ,const vector<float> &);
        vector<double>          operator* (const vector<vector<double> > &
                                ,const vector<double> &);
        vector<float>           operator* (const vector<vector<int> > &
                                ,const vector<float> &);
        vector<double>          operator* (const vector<vector<int> > &
                                ,const vector<double> &);
        vector<vector<float> >  operator* (const vector<vector<float> > &
                                ,const vector<vector<float> > &);
        vector<vector<double> > operator* (const vector<vector<double> > &
                                ,const vector<vector<double> > &);
        vector<vector<float> >  InvM(const vector<vector<float> >&);
        vector<vector<double> > InvM(const vector<vector<double> >&);
        vector<vector<float> >  Tmat(const vector<vector<float> >&);
        vector<vector<double> > Tmat(const vector<vector<double> >&);
        vector<vector<int> >    Tmat(const vector<vector<int> >&);
        vector<vector<float> >  Tmat(const vector<float>&);
        vector<vector<double> > Tmat(const vector<double>&);
        vector<float>           MtoV(const vector<vector<float> >&, int);
        vector<double>          MtoV(const vector<vector<double> >&, int);
        vector<vector<float> >  VtoM(const vector<float> &);
        vector<vector<double> > VtoM(const vector<double> &);
        float   Minelement(const vector<float> &, int*);
        double  Minelement(const vector<double> &, int*);
        vector<vector<complex<float> > >
                operator+ (const vector<vector<complex<float> > > &
                                ,const vector<vector<complex<float> > > &);
        vector<vector<complex<float> > >
                operator- (const vector<vector<complex<float> > > &
                                ,const vector<vector<complex<float> > > &);
        vector<complex<float> >
                operator* (const vector<vector<complex<float> > > &
                                ,const vector<vector<complex<float> > > &);
        vector<vector<complex<float> > >
                operator* (const vector<vector<complex<float> > > &
                                ,const vector<vector<complex<float> > > &);
        vector<vector<complex<float> > >
                operator* (complex<float>
                                ,const vector<vector<complex<float> > > &);
        vector<vector<complex<float> > >
                operator* (complex<float>,const vector<vector<float> > &);
        vector<vector<complex<double> > >
                operator* (complex<double>
                                ,const vector<vector<complex<double> > > &);
        vector<vector<complex<double> > >
                operator* (complex<double>,const vector<vector<double> > &);
        vector<vector<complex<double> > >
                operator* (const vector<vector<complex<double> > > &
                                ,const vector<vector<complex<double> > > &);
        vector<vector<complex<double> > >
                operator* (const vector<vector<complex<float> > > &
                                ,const vector<vector<complex<double> > > &);
        vector<vector<complex<double> > >
                operator* (const vector<vector<complex<double> > > &
                                ,const vector<vector<complex<float> > > &);
        vector<complex<float> >
                operator* (complex<float>,const vector<float> &);
        vector<vector<complex<float> > >
                InvM(const vector<vector<complex<float> > >&);
        vector<vector<complex<float> > >
                Hmat(const vector<vector<complex<float> > >&);
        vector<vector<complex<float> > >
                Hmat(const vector<complex<float> >&);
        vector<complex<float> >
                MtoV(const vector<vector<complex<float> > >&, int);
        vector<vector<complex<float> > >
                VtoM(const vector<complex<float> > &);
        void spline(const vector<double> &, const vector<double> &, double
                , double, vector<double> &, vector<double> &);
        void splint(const vector<double> &, const vector<double> &
                , const vector<double> &, double, double*, double*);
```

5.Input file for equilibrium properties : gasesEqProp.dat

```
      24
name     Mol.Weight   # SxTHERT   SHAJP(J/mol)          A_visc          B_visc          C_visc
N2       28.01600000  2    5.79   0.000000E+00       0.0268142       0.3177838     -11.3155513
                      1       1      3353.2400
                      4       1   0.000000E+00 3   6.014654E+05 6   7.136084E+05 1   7.342244E+05
O2       32.00000000  2    4.16   0.000000E+00        0.044929       -0.0826158      -9.2019475
                      1       1      2238.9700
                      5       3   0.000000E+00 2   9.224688E+04 1   1.579169E+05 3   4.319868E+05 3   5.960445E+05
AR       39.94400000  1    0.00   0.000000E+00        0.029568        0.244716      -10.638224
                      0       1         0.0000
                      3       1   0.000000E+00 5   1.114761E+06 3   1.122024E+06
N        14.00800000  1    0.00   4.713043E+05        0.0115572       0.6031679     -12.4327495
                      0       1         0.0000
                      5       4   0.000000E+00 6   2.301212E+05 4   2.307533E+05 6   3.451566E+05 12  9.971471E+05
O        16.00000000  1    0.00   2.467723E+05        0.0203144       0.4294404     -11.6031403
                      0       1         0.0000
                      6       5   0.000000E+00 3   1.903039E+03 1   2.716630E+03 5   1.899104E+05 1   4.044304E+05 5   8.828567E+05
NO       30.00800000  2    2.45   8.990272E+04        0.0436378       -0.0335511       -9.576743
                      1       1      2699.1800
                      3       4   0.000000E+00 2   5.261802E+05 4   5.495506E+05
C        12.01100000  1    0.00   7.115781E+05       -0.0083285        0.770324        -12.7378
                      0       1         0.0000
                      6       1   0.000000E+00 3   1.961851E+02 5   5.203692E+02 5   1.219422E+05 1   2.589691E+05 5   4.035575E+05
CO2      44.01100000  2    1.13  -3.933329E+05       -0.01952739        1.047818        -14.32212
                      3       2       960.1000 1      1992.5200 1      3380.1500
                      1       1   0.000000E+00
CO       28.01100000  2    2.78  -1.138559E+05       -0.01952739        1.013295        -13.97873
                      1       1      3082.0000
                      5       1   0.000000E+00 6   5.824246E+05 3   6.687160E+05 6   7.452570E+05 2   7.784577E+05
H2        2.01600000  2  175.09   0.000000E+00       -8.3346e-3        0.7815380       -13.5351000
                      1       1      6100.0000
                      4       1   0.000000E+00 1   1.096736E+06 2   1.196150E+06 3   1.351651E+06
H        1.00800000   1    0.00   2.118807E+05       -8.3912e-3        0.7743270       -13.6653000
                      0       1         0.0000
                      4       1   0.000000E+00 3   9.839277E+05 4   1.166148E+06 5   1.229918E+06
CH       13.01894000  2   20.80   5.906830E+05                0               0                0
                      1       1      4116.9353
                      6       2   0.000000E+00 2   2.141299E+02 4   5.383153E+04 4   2.769333E+05 2   3.104165E+05 2   3.806607E+05
CH2      14.02688000  3   26.42   3.859200E+05                0               0                0
                      3       1      4250.1817 1      1519.3607 1      4493.3367
                      1       3   0.000000E+00
CH2a     14.02688000  3   18.32   3.859200E+05                0               0                0
                      3       1      4353.7742 1      2011.4264 1      4422.8361
                      2       0   0.000000E+00 1   3.110266E+04
CH3      15.03482000  4   24.66   1.490310E+05                0               0                0
                      4       1      4319.2433 1       834.4974 2      4581.1028 2      1989.8446
                      2       2   0.000000E+00 2   5.527302E+05
CH4      16.04276000  5   27.13  -6.691100E+04                0               0                0
                      4       1      4196.2269 2      2207.1017 3      4343.2711 3      1879.0579
                      1       1   0.000000E+00
C2H      25.02994000  2   2.122   4.739000E+05                0               0                0
                      3       1      2762.4741 2       920.8247 1      4632.8992
                      2       2   0.000000E+00 4   1.076631E+05
C2H2     26.03788000  2   3.386   2.357550E+05                0               0                0
                      5       1      4854.0410 1      2839.8809 1      4721.9602 2       879.9631 2      1049.3085
                      1       1   0.000000E+00
C2H4     28.05376000  6   3.937   6.098600E+04                0               0                0
                     12       1      4354.3497 1      4463.8416 1      4468.1579 1      2335.0100 1      1758.1996 1      1188.4394 1      1931.1420
                              1      1365.8420 1      4300.1074 1      1471.8807 1      1356.7776 1      2076.8913
                      1       1   0.000000E+00
NO+      30.00745153  2    2.88   9.850914E+05                0               0                0
                      1       1      3372.9500
                      5       1   0.000000E+00 6   4.823612E+05 3   7.073796E+05 6   8.737522E+05 2   8.773437E+05
CO+      28.01045153  2    2.84   1.238976E+06                0               0                0
                      1       1      3082.0000
                      3       2   0.000000E+00 4   2.480197E+05 2   5.488001E+05
O+       15.99945153  1    0.00   1.561248E+06        0.0203144       0.4294404      -11.6031403
                      0       1         0.0000
                      4       4   0.000000E+00 6   3.208276E+05 4   3.210788E+05 6   4.843202E+05
C+       12.01045153  1    0.54   1.799277E+06       -0.0083285        0.770324        -12.7378
                      0       1         0.0000
                      5       2   0.000000E+00 4   7.656001E+02 2   5.143899E+05 4   5.146486E+05 6   5.149942E+05
E-        0.00054847  1    0.00   0.000000E+00                0               0                0
                      0       1         0.0000
                      1       2   0.000000E+00
```

6.Input file for reaction properties : air.dat

```
6       6
N2      O2      AR      N       O       NO      N2      O2      AR      N       O       NO
1       0       0       0       0       0       0       0       0       2       0       0
        3
        0       2.30e+23        -3.5    113260.
        3       8.50e+19        -2.5    113260.
        -1      9.90e+14        -1.5    113260.
0       1       0       0       0       0       0       0       0       0       2       0
        3
        1       3.60e+15        -1.5     59390.
        4       2.10e+12        -0.5     59390.
        -1      1.20e+15        -1.5     59390.
0       0       0       0       0       1       0       0       0       1       1       0
        2
        5       5.20e+15        -1.5     75500.
        -1      5.20e+15        -1.5     75500.
0       1       0       1       0       0       0       0       0       0       1       1
        1
        -1      1.00e+6          0.5      3625.
1       0       0       0       1       0       0       0       0       1       0       1
        1
        -1      5.00e+7          0.0     38020.
1       1       0       0       0       0       0       0       0       0       0       2
        1
        -1      9.10e+18        -2.5     65010.
3
1.      0.      0.
0.      1.      0.
0.      0.      1.
0.      1.      -1.
1.      0.      -1.
1.      1.      -2.
```