ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIN-Dipartimento di ingegneria industriale

Ingegneria Energetica LM-30

Calcolo delle proprietà termodinamiche e di trasporto per plasmi composti da miscele binarie.

CANDIDATO:
Alberto Vagnoni

RELATORE:
Prof. Emanuele Ghedini

CORRELATORI:
Dott. Giorgio La Civita
Dott.ssa Ilaria Maria Paponetti

Anno Accademico [2022-2023]

Sessione [III]

Introduzione

I plasmi innescati a pressione atmosferica o ad alta pressione sono un esempio delle tecnologie emergenti che, per il basso impatto ambientale, possono sostituire convenzionali trattamenti chimici in diversi settori della ricerca ed industriali. Tra le prime si ricordano le tecnologie di sintesi di nanoparticelle, e i trattamenti superficiali, tra le seconde il taglio al plasma.
Il plasma, cosiddetto quarto stato della materia, è quel gas contenente ioni, atomi e radicali gassosi. Nella fattispecie, il lavoro di tesi è consistito nel calcolo delle proprietà termodinamiche e di trasporto di miscele binarie di gas in diverse condizioni di temperatura, a pressione atmosferica; influenzando queste direttamente le performance dei sistemi plasma o, ad esempio, il flusso di calore attraverso sistemi ad ablazione per il rientro atmosferico.
È implicita quindi la traslazione dello studio da una miscela di gas a quello di un plasma, con l'innalzamento della temperatura. È nell'ottica dell'efficientamento, della scalabilità e dell'usabilità di codici di calcolo di tali proprietà, costruiti su modelli fisici di riferimento nell'ambito, che si collocano il fulcro e l'obiettivo principale del lavoro di tesi. Grazie ad una tipologia di programmazione orientata agli oggetti, *OOP* (*ObjectOrientedProgramming*), sono state favorite alcune funzioni chiave come la prototipizzazione, la rifattorizzazione e la manutenzione del codice, poiché i singoli oggetti possono essere modificati senza influire su altri aspetti dell'applicazione.
Tale tecnica ha permesso di realizzare una struttura più flessibile, adatta ad ospitare dati relativi a qualsivoglia tipologia di miscela di gas binaria, in condizioni di equilibrio o non, cambiando semplicemente dei dati input.
La stesura di questa tesi segue la successione dei due step di lavoro: il primo teorico, il secondo applicativo. Allo stesso modo si è scelto quindi di suddividere in due macro parti la descrizione del lavoro svolto.

La prima parte prevede una spiegazione dei modelli fisici di riferimento, delle tecniche di approssimazione, degli algoritmi di riferimento su cui il codice di calcolo è stato sviluppato.
Si fornisce qui una panoramica completa, quindi, della teoria cinetica dei gas e dell'equazione del trasporto di Boltzmann, per la comprensione della teoria che sta alla base dei calcoli delle proprietà termodinamiche e di trasporto. Tra le proprietà si citano la composizione, la conducibilità elettrica, la conducibilità termica, la viscosità, i calori specifici, etc...

Si esplora la teoria cinetica dei gas, l'equazione del trasporto di Boltzmann, e l'approccio di Chapman-Enskog per risolverla. Per ultimi, i metodi di Devoto,che ne utilizzano gli sviluppi per il calcolo numerico delle proprietà di trasporto.
In particolare, si utilizzano l'algoritmo di Godin per il calcolo della composizione della miscela, le formule di Devoto per gli sviluppi di ordine superiore (fino al 4^o) della teoria di Chapman-Enskog per il calcolo delle proprietà di trasporto. Alla conducibilità termica verrà inoltre sommato il suo contributo reattivo, rilevante alle alte temperature.

La seconda parte della tesi descrive invece il processo di re-ingegnerizzazione di un codice di calcolo preesistente per renderlo più efficiente, organizzando già in *incipit* la struttura che avrebbe dovuto avere, sulla base del codice che era stato implementato *ad hoc* per l'applicazione studiata. Quest'organizzazione consiste essenzialmente nella creazione di strutture dati gerarchiche e combina i dati insieme alle operazioni che li manipolano. Tale paradigma è utile soprattutto nella strutturazione di programmi di grandi dimensioni. Se correttamente usato, può rendere un programma maggiormente comprensibile e le sue parti facilmente riutilizzabili in altri contesti. I tre aspetti della programmazione *OO* sono incapsulazione, ereditarietà e polimorfismo che consistono rispettivamente nella separazione della cosiddetta interfaccia di una classe dalla corrispondente implementazione, nel permettere di definire delle classi a partire da altre già definite, e che il client si serva di oggetti di classi diverse, ma dotati di una stessa interfaccia comune. Le classi definiscono un raggruppamento dei tipi di dato e permettono la creazione degli oggetti secondo le caratteristiche definite nella classe stessa. La classe è caratterizzata da attributi, ossia variabili e/o costanti che definiscono le proprietà degli oggetti instanziabili invocando la classe, e da metodi, ossia procedure che operano sugli attributi.
Il codice originale era in linguaggio C e consentiva solo il calcolo delle proprietà di trasporto di una miscela di Argon e Idrogeno. Il nuovo codice, ad oggetti, ha permesso di eseguire il primo calcolo delle proprietà termodinamiche e di trasporto di una miscela aria, semplicemente cambiando negli input le specie da considerare, una volta scelte, e seguendo la routine di esecuzione che verrà descritta nel capitolo dedicato all'attività di re-factoring.
L'obiettivo della tesi è stato dunque lo sviluppo di un programma che possa eseguire queste simulazioni in modo efficiente e accurato, con il minimo numero di azioni e modifiche richieste all'user.
Tale lavoro è stato validato comparando i risultati con quelli ottenuti con il codice *ad hoc* realizzato dal professor E.Ghedini e del dottor P.Sanibondi nel 2009, e poi con quelli in letteratura ad opera di D'Angola, Colonna, Gorse e Capitelli, nel novembre del 2007.

Indice

I I gas nella meccanica statistica 5

1 Proprietà termodinamiche e funzione di distribuzione 5
1.1 Densità e moto medio 6
1.2 Pressione e tensore delle pressioni 9
1.3 Temperatura 11
1.4 Calori specifici 14
1.5 Estensione delle proprietà alle miscele gassose 17

2 Dinamica dei gas nel modello cinetico 18
2.1 L'equazione di Maxwell-Botzman 18
2.2 Le derivazioni dell'equazione di Boltzmann: le equazioni dell'idrodinamica 20
2.2.1 Equazione di continuità 21
2.2.2 Equazione di conservazione della quantità di moto 21
2.2.3 Conservazione dell'energia 22
2.3 Gli urti binari 24
2.4 L'approccio di Chapman-Enskog 26

3 Modello Numerico 28
3.1 Viscosità,conducibilità termica ed elettrica 28
3.2 Miscele proposte e calcolo della composizione 30
3.3 Termodinamica 32
3.4 Collision Integrals calcolati 32

II Codice di calcolo e Validazione 35

4 Re-factoring del codice 35
4.1 Caratteristiche del codice in linguaggio C 35
4.2 Caratteristiche del codice in linguaggio C++ 37

5 Verifica e Validazione 42
5.1 Confronto per una miscela ArH_2 42
5.2 Confronto per l'Aria 45

6 Conclusioni 48

7 Bibliografia 49

Parte I
I gas nella meccanica statistica

1 Proprietà termodinamiche e funzione di distribuzione

L'idea alla base del modello cinetico di un gas è che, data una qualunque proprietà estensiva di un gas questa è funzione solo delle velocità delle particelle costituenti:

$$\phi = \phi(\mathbf{c})$$
$$\frac{\partial \phi(\mathbf{c})}{\partial \mathbf{c}} = \sum_{i=0}^{N} \frac{\partial \phi_i}{\partial c_i} \quad i = 1..N$$

Immaginando di estendere lo spazio tridimensionale cartesiano considerando anche le componenti tridimensionali della velocità, si ottiene uno spazio esadimensionale. È in questo spazio che lo stato cinematico di una particella viene rappresentato da un punto. Gli stati cinematici di tutte le molecole di un gas possono dunque essere rappresentati all'interno di questo spazio dalla funzione di distribuzione:

$$f(\mathbf{c}, \mathbf{r}, t)$$

Quest'ultima descrive le velocità di ogni particella mediante una distribuzione, di fatto, probabilistica. Perciò, il numero di particelle con una certa velocità sarà:

$$n = \int f(\mathbf{c}, \mathbf{r}, t)\ d\mathbf{c} \tag{1.1}$$

Nell'ultima espressione si può notare come l'integrazione della medesima funzione di distribuzione nell'elemento di volume infinitesimo dello spazio delle velocità **dc** riduca n ad essere solo funzione della posizione e del tempo $n = n(\mathbf{r}, t)$. Per questo la funzione di distribuzione prende il nome di **funzione di distribuzione delle velocità**.
Perchè la funzione di distribuzione delle velocità abbia un senso fisico è necessario che sia sempre positiva e che tenda a zero per valori della velocità che tendono all'infinito.

1.1 Densità e moto medio

Per la definizione 1.1,data la massa di una particella di gas, m, la densità si definisce:

$$\rho = nm \tag{1.2}$$

Per proseguire ci si pone in un sistema di riferimento solidale al gas in movimento, di modo che il vettore velocità di una particella possa essere scomposto

$$\mathbf{c} = \mathbf{c_0} + \mathbf{C} \tag{1.3}$$

$\mathbf{c}_0$ rappresenta la velocità macroscopica del gas ovvero la velocità delle molecole come manifestazione di insieme, per definizione la media nel tempo dt tra t,t+dt.
C è la componente ortogonale a $\mathbf{c}_0$ e rappresenta di quanto la velocità della signola particella si discosti da quella di insieme, ed è detta dunque **velocità peculiare**. Il requisito richiesto è che la sua media nel tempo sia nulla, come verrà dimostrato di seguito.
Nel nuovo sistema di riferimento un elemento di volume **dc** diventa un elemento di volume sulle velocità peculiari. Dato $\mathbf{c_0}$ costante e comune a tutte le molecole le espressioni precedenti diventano

$$f\left(\mathbf{c_0} + \mathbf{C}, \mathbf{r}, t\right)$$

$$n = \int f\left(\mathbf{c_0} + \mathbf{C}, \mathbf{r}, t\right)\ d\mathbf{C}$$

$$\phi = \phi(\mathbf{c_0} + \mathbf{C})$$

$$\frac{\partial \phi(\mathbf{C})}{\partial \mathbf{C}} = \sum_{i=0}^{N} \frac{\partial \phi_i}{\partial C_i} \qquad i = 1..N$$

Sia:

$$\sum \phi = n\overline{\phi} d\mathbf{r}$$

la media nel tempo dt della somma dei valori di ϕ per le n**dr** molecole contenute nel volume **dr** allora $\overline{\phi}$ rappresenta il valore medio di ϕ per le molecole nelle vicinanze del punto **r**.

Per mezzo della 1.1 l'espressione precedente può essere espressa in termini della funzione di distribuzione delle velocità

$$\sum \phi = d\mathbf{r} \int \phi f \mathbf{dc} \tag{1.4}$$

dividendo per l'elemento di volume **dr** e cambiando sistema di riferimento dalle precedenti otteniamo

$$n\overline{\phi} = \int \phi f d\mathbf{c} = \int \phi f d\mathbf{C} \tag{1.5}$$

La 1.5 è un'espressione molto importante: descrive che una qualunque proprietà macroscopica funzione di **c**, ad un dato intervallo di tempo, è l'integrale della sua controparte microscopica, composta con la funzione di distribuzione delle velocità delle particelle, integrata in tutto lo spazio delle velocità.
Di fatto, come definito in 1.4, la somma degli n-esimi microstati di ogni singola molecola del sistema in esame. In particolare, considerando $\phi = \mathbf{c}$ ossia velocità delle particelle, essendo per definizione $\overline{\mathbf{c}} = c_0$ troviamo

$$\begin{aligned} n\overline{\mathbf{c}} &= \int \mathbf{c} f \; d\mathbf{c} \\ &= \int (\mathbf{c_o} + \mathbf{C}) f \; dC \\ &= n\overline{\mathbf{c_0}} + n\overline{\mathbf{C}} \\ &= n\mathbf{c_0} \\ \overline{\mathbf{C}} = 0, &\quad \overline{U} = \overline{V} = \overline{W} = 0 \end{aligned}$$

Si consideri l'elemento di volume in figura 1. Se dv è il volume di cilindro, allora le molecole con velocità tra **C** e **C+dC** che attraversano la superficie dS in un tempo infinitesimo dt, piccolo a tal punto che non ci siano collisioni, sono $f\mathbf{dCdr}$. Dalla figura 1 , $dv = dScos(\theta)Cdt$ il flusso delle molecole è dunque

$$f(\mathbf{C})\, d\mathbf{C} \cdot C cos(\theta)\, dt\, dS$$
$$C_n f(\mathbf{C})\, d\mathbf{C}\, dt\, dS$$

con C_n la normale a dS della velocità peculiare.
Integrando sulle velocità si ottiene il flusso netto di molecole che attraversano

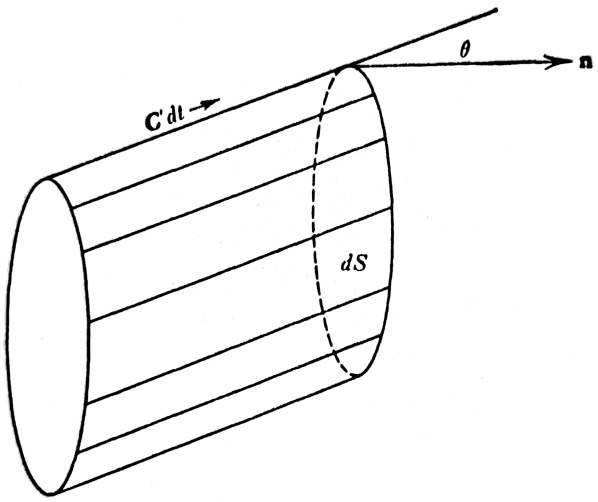


Figura 1: Fig.1

dS in dt.

$$dSdt \int C_n f(\mathbf{C}) d\mathbf{C} = n dS dt \overline{C_n} \tag{1.6}$$

Se $\phi(\mathbf{C})$ è una qualsiasi proprietà scalare per ognuna delle molecole con velocità $\mathbf{C}d\mathbf{C}$ che attraversa l'elemento di volume $dS\mathbf{C}dt$, allora il flusso elementare netto della proprietà trasportata dalle particelle è:

$$\phi(\mathbf{C}) \cdot C_n f(\mathbf{C}) d\mathbf{C} dt dS \tag{1.7}$$

Analogamente, il flusso totale è:

$$dSdt \int C_n \phi(\mathbf{C}) f(\mathbf{C}) d\mathbf{C} = dSdtn\overline{C_n \phi(\mathbf{C})} \tag{1.8}$$

L'eq 1.8 è un caso particolare dell'eq.1.7 ovvero quello con $\phi(\mathbf{C}) = 1$. In realtà dalla eq.1.8 è possibile esprimere una qualunque proprietà macroscopica del gas considerando questa come il risultato dei singoli microstati delle molecole costituenti il gas.
Confrontando le due equazioni si vede che la 1.7 non è altro che l'estensione della formula 1.5 al flusso della stessa proprietà.
La relazione $\phi(\mathbf{C}) = 1$ rappresenta dunque la legge di conservazione della materia mentre sostituendo $\phi(\mathbf{C}) = m\mathbf{c}$ e $\phi(\mathbf{C}) = \frac{1}{2}mc^2$ otteniamo delle espressioni per il flusso della quantità di moto e il flusso dell'energia.
Il rateo di flusso di ϕ si ottiene dividendo 1.8 per $dSdt$. Considerando inoltre che $C_n = \mathbf{C} \cdot \hat{n}$ si ottiene:

$$n\overline{\mathbf{C}\phi(\mathbf{C})} \tag{1.9}$$

L'equazione 1.9 è un'importante espressione detta *flusso vettore* ed esprime il flusso di una quantità molecolare trasportata dalle particelle nella sua forma macroscopica mediata nel tempo.

1.2 Pressione e tensore delle pressioni

Per un gas nel modello cinetico, la pressione è il risultato della quantità di moto trasferita dalle molecole alla superficie di un contenitore attraverso gli urti. Dalla formula 1.8, scegliendo $\phi = m\mathbf{c}$ si ottiene che la quantità di moto "trasferita" delle molecole incidenti dS è

$$dSdt \int_{+} C_n m\mathbf{c} f(\mathbf{C}) dC$$

mentre quella "sottratta" dalle molecole che all'istante dt "rimbalzano" da dS è:

$$dSdt \int_{-} (-C_n) m\mathbf{c} f(\mathbf{C}) dC$$

In condizioni di equilibrio, in assenza di lavoro compiuto dalle molecole sulle pareti del contenitore (non si verifica alcuna espansione), la quantità di moto trasferita e sottratta devono essere uguali e possiamo scrivere:

$$\begin{aligned}
&dSdt \int_{+} C_n m\mathbf{c} f(\mathbf{C}) dC = dSdt \int_{-} (-C_n) m\mathbf{c} f(\mathbf{C}) dC \\
&\Rightarrow dSdt \int C_n m\mathbf{c} f(\mathbf{C}) dC = \\
&= dSdt \int_{+} C_n m\mathbf{c} f(\mathbf{C}) dC - dSdt \int_{-} (-C_n) m\mathbf{c} f(\mathbf{C}) dC \\
&= dSdt \; nm\overline{C_n \mathbf{c}}
\end{aligned}$$

il cui rateo si trova dividendo per $dSdt$. Si definisce di conseguenza il vettore pressione normale:

$$\mathbf{p_n} = nm\overline{C_n \mathbf{c}} \tag{1.10}$$

considerando che il numero di molecole ($\phi(\mathbf{C}) = 1$) che incidono con la superficie all'equilibrio devono uguagliare quelle che rimbalzano da essa si ottiene:

$$
\begin{aligned}
& dSdt \int_{+} C_n f(\mathbf{C}) dC \\
& = dSdt \int_{-} (-C_n) f(\mathbf{C}) dC \\
& \Longrightarrow dSdt \int C_n f(\mathbf{C}) dC \\
& = dSdtn\overline{C_n} = 0 \\
& \Longleftrightarrow \overline{C_n} = 0
\end{aligned}
$$

Quando urtano la parete, le molecole non avranno componente normale, ma il gas conserva una velocità peculiare rispetto alla parete infatti:

$$
\begin{aligned}
\overline{C_n \mathbf{c}} &= \overline{C_n(\mathbf{c_0} + \mathbf{C})} \\
&= \overline{C_n}\mathbf{c_0} + \overline{C_n \mathbf{C}} \\
&= \overline{C_n \mathbf{C}} \neq 0
\end{aligned}
$$

che giustifica l'eq.(1.10) per il vettore pressione normale.
La 1.10 può anche essere definita in termini di un tensore

$$
\begin{aligned}
p_n &= nm\overline{C_n \mathbf{c}} \\
&= \rho \overline{C_n \mathbf{C}} \\
&= \rho \overline{(\hat{n} \cdot \mathbf{C})\mathbf{C}} \\
&= \hat{n} \cdot \rho \overline{\mathbf{C}\mathbf{C}} \\
&= \hat{n} \cdot \underline{\underline{p}}
\end{aligned}
$$

$$
\underline{\underline{p}} = \rho \begin{pmatrix} UU & UV & UW \\ VU & VV & VW \\ WU & WV & WW \end{pmatrix} = \begin{pmatrix} p_{xx} & p_{xy} & p_{xz} \\ p_{yx} & p_{yy} & p_{yz} \\ p_{zx} & p_{zy} & p_{zz} \end{pmatrix} \tag{1.11}
$$

Definendo la pressione media idrostatica:

$$
P = \frac{1}{3} \underline{\underline{p}} : \underline{\underline{I}} \tag{1.12}
$$

Con $\underline{\underline{\mathrm{I}}}$ matrice identità, se $\underline{\underline{\mathrm{p}}}$ è diagonale, ovvero non esistono sforzi tangenziali sulle pareti, allora:

$$P = p_{xx} = p_{yy} = pzz \tag{1.13}$$

Il sistema si dice idrostatico $\underline{\underline{p}} = P\underline{\underline{I}}$ e $\mathbf{p_n} = P\hat{n}$. Si noti che la definizione proposta 1.12 è in accordo con l'assunzione che la pressione non sia mai nulla o negativa.

1.3 Temperatura

Considerando ora la quantità $\phi(\mathbf{C}) = \frac{1}{2}mc^2$, energia molecolare, allora la quantità di energia delle molecole costituenti un gas contenuta in un volume $d\mathbf{r}$, ricordando 1.6, è :

$$\int \frac{1}{2}mc^2 f d\mathbf{c} d\mathbf{r} = \frac{1}{2}mc^2 n d\mathbf{r}$$

Tendendo inoltre conto della trasformazione 1.3, si può scrivere l'energia del gas in termini della velocità peculiare:

$$n\mathbf{dr}\frac{1}{2}m(\mathbf{c_0^2} + 2\mathbf{c_0} \cdot \overline{\mathbf{C}} + \overline{\mathbf{C}}^2) \tag{1.14}$$

$$\frac{1}{2}\rho\mathbf{dr}\mathbf{c_0^2} + \frac{1}{2}\rho\mathbf{dr}\overline{\mathbf{C}}^2 \tag{1.15}$$

Il primo termine della 1.15 rappresenta l'energia cinetica **visibile** della massa in movimento, il secondo termine invece l'energia **invisibile** dovuta ad una moltitudine di moti microscopici che un generico osservatore non può percepire. Dalla formula 1.12 è possibile ottenere i valori della velocità tipica $\mathbf{C}$ per un dato gas misurandone la pressione. Valori tipici, che dipendono dalla massa della particella, sono per l'idrogeno 1839 m/s e per l'azoto molecolare 493 m/s. Queste stime sono confermate da esperimenti sulla misurazione della velocità di efflusso di un gas da una bombola nel vuoto cui viene applicato un piccolo foro.

In un gas a temperature e pressioni ordinarie c'è dunque una ben maggiore quantità di energia microscopica, in termini di velocità peculiare delle molecole, rispetto alla macroscopica.

Il termine al secondo membro di 1.15, diviso per d$\mathbf{r}$ rappresenta l'energia interna o densità di calore interna al gas come conseguenza degli urti delle particelle e

del trasferimento per tutto il volume del gas della quantità $\frac{1}{2}m\mathbf{c}^2$.
Questa ipotesi è esatta se si considera un sistema di particelle costituito da sfere rigide elastiche con la sola energia di traslazione, prive dunque di quella di rotazione o di quella chimica.
Nell'ambito della teoria cinetica dei gas viene definita la **Temperatura** come:

$$\overline{E} = \frac{1}{2}m\overline{\mathbf{C}^2} = \frac{3}{2}k_B T \tag{1.16}$$

con k_B uguale per tutti i gas e introdotta da Ludwig Boltzmann.
La definizione di temperatura per una popolazione di particelle di cui è possibile identificare massa e valore medio quadratico della velocità peculiare $\mathbf{C}$, vale sia se il gas è in stato uniforme che non.
Il calore viene definito inoltre come un flusso della densità di calore $\overline{E}$ e, applicando il vettore 1.9, si ottiene il vettore $\mathbf{q}$ flusso di calore:

$$\mathbf{q} = n\overline{E\mathbf{C}} \tag{1.17}$$

È in questo momento che la teoria cinetica incontra la termodinamica classica. Per un gas in equilibrio, la pressione idrostatica dall'eq1.12 (d'ora in poi solo p), applicando la definizione 1.16 diventa:

$$p = \frac{1}{3}nm\overline{\mathbf{CC}} = \frac{1}{3}\rho\overline{\mathbf{C}}^2 \tag{1.18}$$

ed applicando la definizione di temperatura 1.16 :

$$p = k_B n T \tag{1.19}$$

Questa relazione esprime la proporzionalità con la temperatura analogamente alla legge dei gas perfetti.

- Legge di Charles:
  Per una data massa di gas a pressione costante il volume è direttamente proporzionale alla temperatura assoluta.

- Legge di Boyle:
  Per una data massa di gas a temperatura costante il prodotto tra la pressione e il volume è costante.

- Ipotesi di Avogadro:
  Uguali volumi di gas differenti alle stesse condizioni di temperatura e pressione contengono lo stesso numero di molecole.

Si sostituisca $n = \frac{M}{mV}$ dove M e V massa e volume macroscopici, m massa molecolare, per M di riferimento si scelga un grammo mole di sostanza definito come:

$$M = W = \frac{12m}{m_c} \tag{1.20}$$

dove m_c è la massa di un atomo di carbonio. Un grammo mole di sostanza così definito contiene al suo interno lo stesso numero di molecole, per una qualsiasi sostanza, pari al numero di Avogadro $6.022 \cdot 10^{23}$. Dalla 1.19 segue:

$$p = k_B \frac{M}{mV} T$$

$$pV = k_B \frac{W}{m} T$$

$$pV = k_B \frac{12}{m_c} T$$

definita la costante dei gas per mole di sostanza come:

$$R = \frac{12 k_B}{m_c} = k_B \frac{W}{m} \tag{1.21}$$

si ottiene la legge dei gas perfetti.

$$pV = RT$$

Nel modello in esame la proporzionalità tra il prodotto pressione-volume e temperatura devia da $pV \propto T$ per basse temperature vicine ai fenomeni di condensazione e per alte pressioni a causa della mancata considerazione nel derivare la 1.12 di termini come la grandezza finita delle molecole e i campi di forza a larga distanza intermolecolare.
Questo importante risultato lega la teoria cinetica alla termodinamica antecedente in un ben ampio range di temperature e pressioni attraverso l'adozione della costante k_B.
R come definita in 1.21 ha lo stesso valore per ogni gas, presi $k_B = 1.380649 \cdot 10^{-23}\ J/K$ e $m_c = 1.994473\ g$ si ottiene $R = 8.314462\ J/Kmol$.

1.4 Calori specifici

Si consideri una massa di gas a riposo in un volume chiuso e mantenuto costante, per incrementare la sua temperatura da T a T+∂T deve essere fornita una certa quantità di calore proporzionale all'incremento di temperatura richiesto. Si chiama allora c_v la costante di questa proporzionalità.
Poichè il volume è mantenuto costante, il gas non svolge alcun lavoro sulle pareti del recipiente e la quantità di calore fornita incrementa la sola componente densità di calore dell'equazione 1.15. Il gas subisce dunque un incremento di temperatura specifico:

$$c_v \partial T = \frac{\partial \overline{E}}{m}$$

$$c_v = \frac{1}{m}\Big(\frac{d\overline{E}}{dT}\Big)_v$$

Il calore specifico ottenuto è un differenziale esatto perchè l'energia, come è stata definita in 1.16, dipende solo dalla temperatura. Derivando la definizione di temperatura si ottiene:

$$\frac{d\overline{E}}{dT} = \frac{3}{2}k_B \tag{1.22}$$

$$c_v = \frac{3k_B}{2m} \tag{1.23}$$

Il termine 1.23 è chiamato calore specifico a volume costante. Un procedimento analogo può essere svolto per il calore specifico a pressione costante dall'equazione di stato 1.19, considerando il lavoro svolto dal gas in espansione $p\partial V$ per un aumento di temperatura ∂T:

$$p\partial V = \frac{k_B}{m}\partial T$$

$$c_p \partial T = p\partial V + \frac{\partial \overline{E}}{m}$$

$$= \frac{1}{m}\Big(k_B \partial T + \partial \overline{E}\Big)$$

$$\Longrightarrow c_p = \frac{k_B}{m} + \frac{1}{m}\Big(\frac{d\overline{E}}{dT}\Big)_p$$

Si può eliminare l'indicazione di mantenere costante la pressione nel fare la derivata dell'energia poichè quest'ultima dipende solo dalla temperatura, attraverso la 1.22 si ottiene:

$$c_p = \frac{k_B}{m} + c_v \tag{1.24}$$

moltiplicando per W (1.20),peso molecolare del gas in esame, e ricordando 1.21 i calori specifici per mole di sostanza sono:

$$C_v = \frac{3k_B W}{2m} \tag{1.25}$$

$$C_p = \frac{W k_B}{m} + W c_v = R + C_v \tag{1.26}$$

Definendo il valore

$$\gamma = \frac{c_p}{c_v} = \frac{C_p}{C_v}$$

si noti che

$$\gamma = \frac{\frac{k_B}{m}}{\frac{3k_B}{2m}} \frac{c_v}{c_v}$$

$$\gamma = \frac{2}{3} + 1 \simeq 1.66..$$

L'ultima formula è vera per tutti quei gas con la sola energia traslazionale, senza altre forme di energia interna come quella chimica e di rotazione.
Definito N il numero di gradi di libertà per una molecola tali che N=3 per una molecola che possiede solo energia traslazionale, in generale, sono valide le seguenti approssimazioni:

$$c_v = \frac{k_b N}{2m}$$

$$c_p = \frac{k_B}{m}\left(1 + \frac{N}{2}\right)$$

$$\gamma = 1 + \frac{2}{N}$$

In seguito sono forniti alcuni valori sperimentali per gamma.
Si noti come per gas nobili sia esatto il modello di sfere rigide elastiche con sola energia traslazionale, a differenza delle molecole biatomiche per cui N = 5

(all'energia traslazionale se ne aggiungono due terzi nascosti dietro moti rotazionali e vibrazionali). Se ne deduce che, per molecole più complesse, i gradi di libertà si discostano ancora di più N ≥ 5.
Noto γ è possibile calcolare la velocità del suono

$$a = \sqrt{\frac{\gamma RT}{1 - \frac{P}{R}\left(\frac{\partial R}{\partial P}\right)_T}}$$

gas	γ	N
Ar	1.669	$\simeq 3$
Ne	1.668	$\simeq 3$
H_2	1.408	$\simeq 5$
O_2	1.398	$\simeq 5$
CO_2	1.303	≥ 5
NH_3	1.318	≥ 5

Tabella 1: Valori tipici di γ per diverse sostanze

1.5 Estensione delle proprietà alle miscele gassose

Tutte le formule precedenti sono estendibili per una miscela gassosa di n gas costituenti, considerando che la k-esima popolazione ha la propria funzione di distribuzione delle velocità:

$$f_k\left(\mathbf{c_k}, \mathbf{r}, t\right)$$

$$n_k = \int f_k(\mathbf{c_k}, \mathbf{r}, t) d\mathbf{c_k}, \quad n = \sum_k n_k$$

$$\rho_k = n_k m_k, \quad \rho = \sum_k n_k m_k$$

$$n\overline{\phi_k} = \int f_k \phi_k d\mathbf{c}_k, \quad n\overline{\phi} = \sum_k n_k \overline{\phi_k}$$

$$\rho C_0 = \sum_k \int f_k m_k c_k \mathbf{dc_k} = \sum_k \rho_k \overline{\mathbf{c_k}}$$

$$c_k = c_0 + C_k, \quad \overline{C_k} = 0$$

$$\frac{1}{2}\overline{mC}^2 = \frac{1}{n}\sum_k \int f_k 0.5 m_k C_k^2 d\mathbf{c}_k = \frac{3}{2} k_B T$$

$$\underline{\underline{p}} = \sum_k \underline{\underline{p_k}} = \sum_k n_k \overline{m_k \mathbf{C_k} \mathbf{C_k}}$$

2 Dinamica dei gas nel modello cinetico

2.1 L'equazione di Maxwell-Botzman

Con riferimento ad alcune definizioni del capitolo uno, la funzione di distribuzione di velocità descrive lo stato dinamico di una molecola con tre coordinate di posizione, tre coordinate di velocità, ed una di tempo.
Una molecola appare attraverso la f come punto in uno spazio di sei dimensioni e disegna una traiettoria nello stesso spazio in un tempo $t,\ t + dt$. Tale spazio viene comunemente chiamato *spazio delle fasi*: come un elemento di volume in uno spazio tridimensionale ha tre coordinate, un elemento di volume dello spazio delle fasi ha sei coordinate (più il tempo).
Per definizione, integrando la funzione di distribuzione delle molecole nell'intero spazio delle velocità otteniamo il numero di molecole che occupano il volume $\vec{r}, \vec{r} + d\vec{r}$, e, la media sulle velocità di una qualunque proprietà molecolare $\phi = \phi(\vec{c})$ che è funzione delle velocità delle n molecole:

$$f\left(\vec{c}, \vec{r}, t\right) \tag{2.1}$$

$$n = \int f\left(\vec{c}, \vec{r}, t\right)\ d\vec{c} \tag{2.2}$$

$$n\overline{\phi} = \int \phi f d\vec{c} = \int \phi f d\vec{C} \tag{2.3}$$

$$\tag{2.4}$$

Se è vero che una qualunque proprietà molecolare può essere espressa in termini della funzione di distribuzione, allora la variazione nel tempo della stessa si può esprimere in funzione della variazione nel tempo della funzione di distribuzione. L'intuizione di Boltzmann fu che le proprietà macroscopiche di un gas sono il risultato del trasporto di determinate proprietà molecolari che avviene tra le molecole nel momento in cui si urtano. Il numero di molecole nell'elemento di volume dello spazio delle fasi in movimento $\vec{r} + \vec{c}d\vec{r}dt$ con velocità nell'intervallo $\vec{c} + \vec{F}d\vec{c}dt$ dove $\vec{F}$ è la forza esterna applicata alle molecole, per definizione di 2.1 è allora :

$$f(\vec{c} + \vec{F}dt, \vec{r} + \vec{c}dt, t)\vec{dc}\vec{dr}dt$$

Alla stessa maniera la variazione del numero di molecole nell'elemento di volume

dovuto agli scontri tra le stesse(∂_e)(escono ed entrano dall' elemento di volume dello spazio delle fasi) è:

$$\frac{\partial_e f}{\partial t} d\vec{c} d\vec{r} dt = \left[f(\vec{c} + \vec{F}dt, \vec{r} + \vec{c}dt, t + dt - f(\vec{c}, \vec{r}, t) \right]$$

dividendo per l'elemento di volume dello spazio delle fasi e il tempo si ottiene

$$\frac{\partial_e f}{\partial t} = \frac{\partial f}{\partial t} + \vec{c} \cdot \frac{\partial f}{\partial \vec{r}} + \vec{F} \cdot \frac{\partial f}{\partial \vec{c}} \tag{2.5}$$

Tale relazione è detta *Equazione differenziale di Maxwell-Boltzmann.*
Questa fondamentale equazione, derivata solo da considerazioni geometriche esa-dimensionali, descrive l'andamento nel tempo della funzione di distribuzione f per *"perturbazioni"* dovute agli scontri tra le molecole.
Il primo termine rappresenta il naturale evolversi della funzione di distribuzione nel tempo se per ragioni non dinamiche le molecole subiscono delle trasformazioni che alterano il loro stato di moto. Il secondo termine rappresenta le molecole che, a causa dell'urto con altre molecole, escono dall'elemento di volume tridimensionale $\vec{dr}$. L'ultimo termine rappresenta invece le molecole che, a causa di una accelerazione esterna (forza esterna), escono dall'elemento di volume (anche esso tridimensionale) $\vec{dc}$.
Spostiamo ora il sistema di riferimento in quello 1.3 della velocità peculiare delle molecole

$$\begin{aligned}
\vec{c} &= \vec{C} + \vec{c_0} \\
\frac{\partial f}{\partial t} &= \frac{\partial f}{\partial t} - \frac{\partial \vec{c_0}}{\partial t} \cdot \frac{\partial f}{\partial \vec{C}} \\
\frac{\partial f}{\partial \vec{r}} &= \frac{\partial f}{\partial \vec{r}} - \frac{\partial \vec{c_0}}{\partial \vec{r}} \cdot \frac{\partial f}{\partial \vec{C}} \\
\frac{\partial f}{\partial \vec{c}} &= \frac{\partial f}{\partial \vec{C}}
\end{aligned}$$

$$\Rightarrow \frac{\partial_e f}{\partial t} = \frac{\partial f}{\partial t} - \frac{\partial \vec{c_0}}{\partial t} \cdot \frac{\partial f}{\partial \vec{C}} + (\vec{c_0} + \vec{C}) \cdot \left[\frac{\partial f}{\partial \vec{r}} - \frac{\partial}{\partial \vec{r}} \vec{c_0} \cdot \frac{\partial f}{\partial \vec{C}} \right] + \vec{F} \cdot \frac{\partial f}{\partial \vec{C}}$$

Definito l'operatore

$$\frac{\mathcal{D}}{\mathcal{D}t} \equiv \frac{\partial}{\partial t} + \vec{c_0} \cdot \frac{\partial}{\partial \vec{r}} \tag{2.6}$$

è possibile ricondurre la precedente a

$$\frac{\partial_e f}{\partial t} = \frac{\mathcal{D} f}{\mathcal{D} t} + \vec{C} \cdot \frac{\partial f}{\partial \vec{r}} + \left(\vec{F} - \frac{\mathcal{D} \vec{c_0}}{\mathcal{D} t} \right) \cdot \frac{\partial f}{\partial \vec{C}} - \frac{\partial f}{\partial \vec{C}} \vec{C} : \frac{\partial \vec{c_0}}{\partial \vec{r}} \tag{2.7}$$

2.2 Le derivazioni dell'equazione di Boltzmann: le equazioni dell'idrodinamica

Per quanto trattato nel capitolo 1 la conoscenza della f è sufficiente a descrivere le proprietà del gas, a patto che queste siano funzione delle velocità delle molecole. Se l'ipotesi di Boltzmann è vera, allora l'equazione 2.7 è sufficiente per descriverne l'evolversi nel tempo. Con l'adozione dell'operatore 2.6 si entra di fatto nella descrizione dello stato dinamico di un fluido.
L'operatore 2.6 chiamato derivata Euleriana o sostanziale esprime un'operazione di derivazione nel tempo di una variabile, eseguita su di un volume in moto solidale con il moto del fluido, ed è usato anche nella definizione delle equazioni di Navier-Stokes per la fluidodinamica .
Dunque, se la 2.4 descrive la media di una determinata grandezza molecolare, allora la sua variazione nel tempo dovuta agli urti tra le particelle, ricorrendo alla 2.5, è data da:

$$\Delta\overline{\phi} = \frac{1}{n}\int \phi \frac{\partial_e f}{\partial t} d\vec{C} \tag{2.8}$$

Sostituendo la 2.7 nella 2.8 e considerando

$$\frac{\mathcal{D}}{\mathcal{D}t}\int \phi f d\vec{C} = \int \phi \frac{\mathcal{D}f}{\mathcal{D}t} d\vec{C} + \int f \frac{\mathcal{D}\phi}{\mathcal{D}t} d\vec{C}$$

dopo alcuni passaggi algebrici [18] si ottiene un espressione per $\Delta\overline{\phi}$ che non dipende più dalla funzione di distribuzione $f(\vec{c}, \vec{r}, t)$ ma dalla stessa proprietà macroscopica $\overline{\phi}$

$$\begin{aligned}\Delta\overline{\phi} = \frac{\mathcal{D}\overline{\phi}}{\mathcal{D}t} + \overline{\phi}\cdot\frac{\partial}{\partial\vec{r}}\cdot\vec{c_0} + \frac{\partial}{\partial\vec{r}}\cdot\overline{\phi\vec{C}} - \overline{\frac{\mathcal{D}\phi}{\mathcal{D}t}} + \overline{\vec{C}\cdot\frac{\partial\phi}{\partial\vec{r}}} + \\ + \left(\vec{F} - \frac{\mathcal{D}\vec{c}_0}{\mathcal{D}t}\right)\cdot\overline{\frac{\partial\phi}{\partial\vec{C}}} - \overline{\frac{\partial\phi}{\partial\vec{C}}\vec{C}} : \frac{\partial\vec{c}_0}{\partial\vec{r}}\end{aligned} \tag{2.9}$$

che vale per la quantità $\overline{\phi}$ che sia essa vettoriale o scalare.
La precedente è una generalizzazione dell'equazione di Maxwell-Boltzmann per la variazione di ϕ dovuta agli incontri tra le particelle formulata da Enskog, a cui è dovuto l'inserimento della velocità peculiare.
Sostituendo opportuni valori a ϕ nell'equazione precedente si riottengono le equazioni di Navier-Stokes per un sistema gassoso come viene definito nel capitolo 1.

2.2.1 Equazione di continuità

Moltiplicando ambo i membri di 2.8 per n otteniamo

$$n\Delta\overline{\phi} = \int \phi \frac{\partial_e f}{\partial t} d\vec{C} \tag{2.10}$$

Per l'equazione di conservazione della massa poniamo $\phi = 1$ e dall'analisi svolta nel capitolo 1 otteniamo

$$\overline{\phi} = 1$$
$$\overline{\phi\vec{C}} = \overline{\vec{C}} = 0$$
$$\frac{\mathcal{D}\overline{\phi}}{\mathcal{D}t} = 0$$
$$\frac{\partial\phi}{\partial\vec{r}} = 0$$

Se ipotizziamo che il sistema sia chiuso il numero di molecole deve conservarsi quindi

$$\Delta\phi = 0$$

la 2.10 applicandovi la 2.9 diventa:

$$\frac{\mathcal{D}n}{\mathcal{D}t} + n\frac{\partial}{\partial\vec{r}} \cdot \vec{c_0} = 0 \tag{2.11}$$

Questa è detta equazione di continuità: il numero totale di molecole ed equivalentemente la massa macroscopica del sistema si conservano.

2.2.2 Equazione di conservazione della quantità di moto

Scegliendo $\phi = m\vec{C}$ quantità di moto e ricordando le definizioni(1.2) di densità $\rho = nm$ e del tensore delle pressioni (1.11) $\underline{\underline{p}}$ si sostituiscono in 2.9 i termini:

$$\overline{\phi} = 0 \tag{2.12}$$
$$n\overline{\phi\vec{C}} = \rho\overline{\vec{C}\vec{C}} = \underline{\underline{p}} \tag{2.13}$$
$$\frac{\partial\phi}{\partial\vec{r}} = 0 \tag{2.14}$$

$$\frac{\mathcal{D}f}{\mathcal{D}t} = 0 \tag{2.15}$$

$$\frac{\partial \phi}{\partial \vec{C}} = (m, m, m) \tag{2.16}$$

$$\overline{\frac{\partial \phi}{\partial \vec{C}} \vec{C}} = \hat{m}\overline{\vec{C}} = 0 \tag{2.17}$$

Grazie alle 2.12-2.17 la 2.10 diventa:

$$\frac{\partial}{\partial \vec{r}} \cdot \underline{\underline{p}} - \rho\left(\vec{F} - \frac{\mathcal{D}\vec{c}_0}{\mathcal{D}t}\right) = 0 \tag{2.18}$$

Esattamente identica all'equazione della quantità di moto per il modello del continuo di un fluido.
Si noti che la differenza sussiste solo nell'interpretazione del fenomeno della pressione: qui la pressione non è la resistenza di una cella di elemento fluido a subire una deformazione, ma il risultato del trasporto di quantità di moto delle molecole costituenti un determinato volume per via degli urti tra le stesse.

2.2.3 Conservazione dell'energia

In questo caso, si sostituisce la quantità 1.16 energia cinetica delle molecole espressa in termini di velocità peculiare seguendo lo stesso metodo usato per le due equazioni di conservazione precedenti

$$\phi = \overline{E}$$
$$n\overline{\phi \vec{C}} = n\overline{E\vec{C}} = \vec{q}$$
$$\frac{\partial \phi}{\partial \vec{r}} = 0$$
$$\frac{\mathcal{D}f}{\mathcal{D}t} = 0$$
$$\frac{\partial \phi}{\partial \vec{C}} = m\vec{C}$$
$$\overline{\frac{\partial \phi}{\partial \vec{C}}} = 0$$
$$n\overline{\frac{\partial \phi}{\partial \vec{C}} \vec{C}} = \rho\overline{CC} = \underline{\underline{p}}$$

e infine per

$$\Delta \phi = 0$$

la 2.9 diventa

$$\frac{D(nE)}{Dt} + nE\frac{\partial}{\partial \vec{r}} \cdot \vec{c}_0 + \frac{\partial}{\partial \vec{r}} \cdot \vec{q} + \underline{\underline{p}} : \frac{\partial c_0}{\partial \vec{r}}$$

applicando la definizione di temperatura 1.16 e la

$$\frac{dE}{dT} = \frac{1}{2} N k_B$$

otteniamo

$$\frac{DT}{Dt} = -\frac{2}{N k_B n}\left(\underline{\underline{p}} : \frac{\partial \vec{c}_0}{\partial \vec{r}} + \frac{\partial}{\partial \vec{r}} \cdot \vec{q}\right) \tag{2.19}$$

La 2.19 o la sua forma precedente viene detta solitamente equazione dell'energia termica del gas. N sono i gradi di libertà del gas in esame illustrati nel capitolo 1, k_B la costante di Boltzmann q il flusso di calore p la pressione.

2.3 Gli urti binari

Nei capitoli precedenti si sono introdotte le equazioni di stato per un gas nelle ipotesi particolari del modello cinetico ed è stata ricavata l'equazione di Maxwell-Boltzmann che regola l'evoluzione nel tempo della funzione di distribuzione, attraverso considerazioni geometriche in uno spazio di sei dimensioni. In seguito, l'equazione di Maxwell-Boltzmann insieme al particolare sistema di riferimento della velocità peculiare delle molecole è stata derivata per ricavare e equazioni simili a quelle di Navier-Stokes dell'idrodinamica.
Boltzmann, ebbe poi l'idea che tra tutti gli urti tra le particelle, il numero di urti binari, cioè tra sole due particelle, fosse il meccanismo predominante di trasporto di quantità di moto ed energia all'interno di un gas a pressioni moderate.
Al di sopra delle suddette pressioni, ad elevate densità, la quantità di molecole

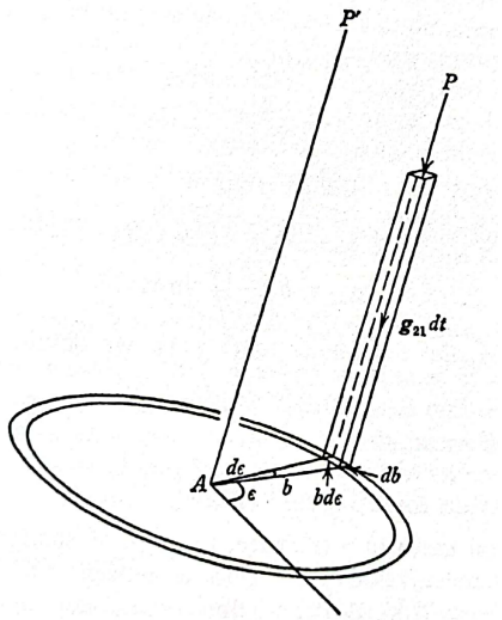


Figura 2: Geometria di un urto binario

per unità di volume troppo elevata fa sì che diventi importante anche il numero degli urti terziari, cioè tra tre molecole simultaneamente.
Fortunatamente, la teoria approssima in maniera soddisfacente un range di pressioni che è piuttosto ampio per l'esperianza comune.
Si veda come esempio la figura 2 per descrivere la geometria di un urto binario tra molecole di "due set", ossia due molecole che differiscono anche solo per la diversa $f(\vec{c}, \vec{r}, t)$ nello spazio delle fasi. Ci si pone in un particolare sistema di riferimento in movimento solidale con le molecole del primo set. Sia $\vec{g}$ la velocità relativa tra le molecole che stanno avvicinandosi nell'urto, e siano f e f_1 rispettivamente le funzioni di distribuzione delle molecole del primo set e del secondo set che urtano le prime in movimento. Queste ultime descrivono prima dell'urto, ognuna, un volume infinitesimo $bdbd\epsilon d\vec{r}gdt$; il volume totale per le

molecole del primo tipo sarà dunque per definizione di 2.1

$$dv = fbdbd\epsilon gdtd\vec{r}d\vec{c}$$

. Scegliendo un dt sufficientemente piccolo, compreso per esempio tra il tempo di un urto e il libero cammino medio delle molecole, si può supporre che nel volume in esame non avvenga più di un urto per ogni molecola.
Allora nel volume in esame il numero di incontri tra le molecole dei due set è

$$ff_1g\alpha(b,\chi)d\hat{e}d\vec{c}d\vec{c_1}d\vec{r}dt \tag{2.20}$$

dove $d\hat{e}$ rappresenta una "direzione di volo" per le molecole espressa in angoli polari

$$d\hat{e} = sin\chi d\chi d\epsilon \tag{2.21}$$

$$\alpha = b\Big|\frac{\partial b}{\partial \chi}\Big|/sin\chi \tag{2.22}$$

ed $\alpha = \alpha(b,\chi)$ una funzione scalare positiva del parametro d'urto e dell'angolo polare in questione.
La variazione del numero di molecole del primo set per incontri con le molecole del secondo set è dato da un bilancio del tipo:
-Urti diretti:
molecole che da altri set entrano nel primo set per effetto di urti;
-Urti inversi:
molecole nel primo set che per effetto di urti vengono sbalzate fuori dal primo set.[1]

$$\begin{aligned}\frac{\partial_e f}{\partial t}d\vec{c} &= \Gamma_+ - \Gamma_- \\ &= d\vec{c}d\vec{r}dt\iint f'f_1'g\alpha d\hat{e}d\vec{c}_1 - d\vec{c}d\vec{r}dt\iint ff_1g\alpha d\hat{e}d\vec{c}_1 \\ &= d\vec{c}_1d\vec{r}dt\iint (f'f_1' - ff_1)g\alpha d\hat{e}d\vec{c}_1\end{aligned}$$

dividendo per $d\vec{c}, d\vec{r}, dt$ si ottiene *l'equazione di Boltzmann.*

$$\frac{\partial_e f}{\partial t} = \iint (f'f_1' - ff_1)g\alpha d\hat{e}d\vec{c}_1 \tag{2.23}$$

[1] l'incontro binario modifica le velocità delle particelle, di conseguenza la funzione di distribuzione, esse ad esempio non apparterranno più alle molecole con velocità nel range c, c+dc dopo l'incontro ma ad uno differente c',c'+dc'

2.4 L'approccio di Chapman-Enskog

Uguagliando 2.5 e 2.23 possiamo scrivere

$$\iint (f'f_1' - ff_1) g\alpha d\hat{e} d\vec{c}_1 = \frac{\partial f}{\partial t} + \vec{c} \cdot \frac{\partial f}{\partial \vec{r}} + \vec{F} \cdot \frac{\partial f}{\partial \vec{c}}$$

$$\Big(\frac{\partial f}{\partial t} + \vec{c} \cdot \frac{\partial f}{\partial \vec{r}} + \vec{F} \cdot \frac{\partial f}{\partial \vec{c}}\Big) - \iint (f'f_1' - ff_1) g\alpha d\hat{e} d\vec{c}_1 = 0$$

$$\mathscr{D}f + \mathscr{J}(f, f_1) = \xi(f) = 0$$

Enskog espande la funzione di distribuzione in una serie infinita tale che ogni termine aggiunto approssimi "meglio" la f, e anche la soluzione del problema $\xi(f) = 0$ potrà essere scritta come sommatoria di una serie di soluzioni.

$$f = f^0 + f^1 + f^2 + ... + f^n$$
$$\xi^0(f^0) = 0$$
$$\xi^1(f^0, f^1) = 0$$
$$.$$
$$.$$
$$.$$
$$\xi^n(f^0, f^1, ..., f^n) = 0$$

$$\xi(f) = \xi(f^0) + \xi^1(f^0, f^1) + ... + \xi^n(f^0, f^1, ..., f^n) = 0 \qquad (2.24)$$

È possibile dimostrare attraverso il teorema H di Boltzmann che la soluzione di prima approssimazione $\xi^0(f^0) = 0$ descrive il comportamento del gas nel suo *"Uniform steady state"* e che la sua f^0 prende la forma di una (non a caso) Maxwelliana

$$f = n\Big(\frac{m}{2\pi k_B T}\Big)^{3/2} e^{-\frac{mC^2}{2k_B T}}$$

La soluzione degli altri termini dell'espansione invece non è così banale e passa attraverso la definizione di una funzione di perturbazione così che la f^1 ad esempio sia il risultato della composizione tra la f^0 e la funzione di perturbazione.

$$f^1 = f^0 \phi^1$$

In generale, la soluzione del sistema 2.24 permette di conoscere le funzioni di distribuzione ad un livello di approssimazione scelto capace di descrivere anche lo stadio non uniforme del gas.

L'espansione in *polinomi di Sonine* definiti come

$$S_m^{(n)}(x) = \sum_{p=0}^{n} (-x)^p (m+n)_{n-p} / p!(n-p)!$$

del 2.24 permette di ottenere delle espressioni esplicite della 2.23 per le interazioni in funzione della velocità relativa e della sezione d'urto α.

3 Modello Numerico

3.1 Viscosità,conducibilità termica ed elettrica

Le espressioni del capitolo precedente, comunemente chiamate *Bracket Integrals*, dipendono dal grado di approssimazione scelto per la soluzione del sistema 2.24. Proseguire da qui per ricavare tali espressioni risulterebbe tedioso e totalmente al di fuori dagli scopi di questo elaborato, fortunatamente si trovano le espressioni necessarie in letteratura. [1]
Basti sapere che suddetti *Bracket Integrals* possono essere ridotti ad un sistema lineare sui *Collision Integrals*, che verranno chiariti in seguito.
In breve, i coefficienti di tale sistema lineare si possono scrivere in forma matriciale come

$$q_{ij}^{mp} = \begin{pmatrix} q_{11}^{00} & \dots & q_{1\nu}^{00} & q_{11}^{01} & \dots & q_{1\nu}^{0p} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ q_{\nu 1}^{00} & \dots & q_{\nu\nu}^{00} & q_{\nu 1}^{01} & \dots & q_{\nu\nu}^{0p} \\ \\ q_{11}^{10} & \dots & q_{1\nu}^{10} & q_{11}^{11} & \dots & q_{1\nu}^{1p} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ q_{\nu 1}^{m0} & \dots & q_{\nu\nu}^{m0} & q_{\nu 1}^{m1} & \dots & q_{\nu\nu}^{mp} \end{pmatrix}$$

Dove il generico elemento $q_{\nu\nu}^{mp}$ come anticipato al paragrafo precedente è funzione dei soli *collision integrals* a loro volta funzioni della sola temperatura.

$$q_{\nu\nu}^{mp} = q_{\nu\nu}^{mp}(\Omega_{\nu\nu}^{mp}(T)) \tag{3.1}$$

Una volta ottenuta questa matrice, viscosità, conducibilità elettrica e termica sono calcolabili in termini di determinanti di sottomatrici della stessa [2] [**?**]

$$[\eta]_2 = -\frac{5}{2}\frac{(2\pi k_B T)^{0.5}}{|\hat{q_{\nu\nu}^{mp}}|}\begin{vmatrix} \hat{q}_{ij}^{00} & \hat{q}_{ij}^{01} & n_i m_i^{0.5} \\ \hat{q}_{ij}^{10} & \hat{q}_{ij}^{11} & 0 \\ n_i & 0 & 0 \end{vmatrix} \tag{3.2}$$

[2] Per ottenerle dal sistema discretizzato di Chapman-Enskog si introduce la *collision frequency*, il *libero cammino medio* e il *modello di diffusione* per cui si rimanda alla bibliografia

con n_i come al solito densità molecolare dell'i-esima specie e m_i massa molecolare

$$[\lambda_h]_4 = -\frac{75k_B}{8}\frac{(2\pi k_B T)^{0.5}}{|q_{\nu\nu}^{mp}|}\begin{vmatrix} q_{ij}^{00} & q_{ij}^{01} & q_{ij}^{02} & q_{ij}^{03} & 0 \\ q_{ij}^{10} & q_{ij}^{11} & q_{ij}^{12} & q_{ij}^{13} & n_i \\ q_{ij}^{20} & q_{ij}^{21} & q_{ij}^{22} & q_{ij}^{23} & 0 \\ q_{ij}^{30} & q_{ij}^{31} & q_{ij}^{32} & q_{ij}^{33} & 0 \\ 0 & n_i/m_i^{0.5} & 0 & 0 & 0 \end{vmatrix} \tag{3.3}$$

dove il subscript h indica i *pesanti*, mentre, per la conducibilità termica dovuta agli elettroni liberi[3,4] [6][5]

$$[\lambda_e]_4 = \frac{75n_e^2 k_B(2\pi k_B T/m_e)^{0.5}}{8|q_s^{mp}|}\begin{vmatrix} q_s^{22} & q_s^{23} \\ q_s^{32} & q_s^{33} \end{vmatrix} \tag{3.4}$$

Il coefficiente di diffusione degli elettroni analogamente è

$$[D_{ee}]_4 = \frac{3n_e\rho}{2n_{tot}m_e}\frac{(2\pi k_B T/m_e)^{0.5}}{|q_s^{mp}|}\begin{vmatrix} q_s^{11} & q_s^{12} & q_s^{13} \\ q_s^{21} & q_s^{22} & q_s^{23} \\ q_s^{31} & q_s^{32} & q_s^{33} \end{vmatrix} \tag{3.5}$$

e grazie ad esso la conducibilità elettrica

$$\sigma = \frac{e^2 n_e n_{tot} m_e}{\rho k_B T} D_{ee} \tag{3.6}$$

[3]Le formule per la conducibilità elettrica degli elettroni e la formula semplificata per la conducibilità elettrica si ottengono dalla teoria di Devoto semplificata [2] semplificata considerando che la massa dell'elettrone è almeno e^{-4} minore di quella dei pesanti

[4]Alle alte temperature diventa importante il contributo della conducibilità termica reattiva, calcolata in questo elaborato ma per le cui formule e sviluppi si rimanda alla bibliografia

3.2 Miscele proposte e calcolo della composizione

Per proseguire è necessario ora indicare brevemente le miscele proposte nell'elaborato e il calcolo della loro composizione.

- Ar-H2 7 specie:
  $Ar, Ar^{+}, Ar^{+2}, H, H^{+}, H_2, e^{-}$
- Aria 17 specie:
  $N, N^{+}, N^{+2}, N^{+3}, N^{+4}, N_2, N_2^{+}, NO,$
  $NO^{+}, O, O^{+}, O^{+2}, O^{+3}, O^{+4}, O_2, O_2^{+}, e^{-}$

Per calcolare la composizione viene utilizzato l'algoritmo di Godin Trepanièr che sfrutta una costruzione matriciale con *matrice di composizione* $C \in Mat^{NxM}$ e di conservazione $A \in Mat^{MxN}$ con M ed N rispettivamente numero di elementi così come appaiono nella tavola periodica più l'elettrone e numero di specie presenti.
Le matrici di composizione per le miscele proposte sono dunque:

	Ar	H	$Carica$
Ar	1	0	0
Ar^{+}	1	0	1
Ar^{+2}	1	0	2
H	0	1	0
H^{+}	0	1	1
H_2	0	2	0
$e-$	0	0	-1

	N	O	$Carica$
N	1	0	0
$N+$	1	0	1
N^{+2}	1	0	2
N^{+3}	1	0	3
N^{+4}	1	0	4
N_2	2	0	0
N_2^{+}	2	0	1
NO	1	1	0
NO^{+}	1	1	1
O	0	1	0
$O+$	0	1	1
O^{+2}	0	1	2
O^{+3}	0	1	3
O^{+4}	0	1	4
O_2	0	2	0
O_2^{+}	0	2	1
$e-$	0	0	-1

(3.7)

Le matrici di conservazione A definiscono un sistema non lineare con incognite le densità di particelle delle varie specie $An = A_0$. I suoi elementi [7] si ottengono dall'applicazione di tre relazioni di conservazione, rispettivamente, dei nuclei, delle cariche e la legge di stato dei gas ideali del modello cinetico (1.19) alle matrici di composizione C.

$$\varepsilon_j \sum_{i=1}^{N} n_i C_{i,k} = \varepsilon_k \sum_{i=1}^{N} n_i C_{i,k} \quad j,k \begin{cases} \in \{1, M\} \\ \neq charge \end{cases} \quad and\ j \neq k$$

$$\sum_{i=1}^{N} n_i Z_i = 0$$

$$p = \sum_{i=1}^{N} n_i k_B T$$

Tipo di conservazione	$A_{i,j}$	A_i^0	righe in A
Stechiometria	$\varepsilon_j C_{j,k} - \varepsilon_k C_{i,j}$	0	$M-2$
Neutralità	$C_{i,charge}$	0	1
Legge di stato	$k_B T$	p	1

Tabella 2: Forma della matrice di conservazione

Si propongono le matrici di conservazione delle due miscele come esempio per la sola temperatura di $300K^o$, calcolata partendo dalla percentuale di O_2 del 21% nel caso dell'aria e del 25% di Idrogeno per la miscela Argon-Idrogeno.

$$A_{air}^T = \begin{pmatrix} 5.41e^{-27} & 0 & 4.14e^{-21} \\ 5.41e^{-27} & 1 & 4.14e^{-21} \\ 5.41e^{-27} & 2 & 4.14e^{-21} \\ 5.41e^{-27} & 3 & 4.14e^{-21} \\ 5.41e^{-27} & 4 & 4.14e^{-21} \\ 1.08e^{-26} & 0 & 4.14e^{-21} \\ 1.08e^{-26} & 1 & 4.14e^{-21} \\ 1.16e^{-26} & 0 & 4.14e^{-21} \\ 1.16e^{-26} & 1 & 4.14e^{-21} \\ -2.03e^{-26} & 0 & 4.14e^{-21} \\ -2.03e^{-26} & 1 & 4.14e^{-21} \\ -2.03e^{-26} & 2 & 4.14e^{-21} \\ -2.03e^{-26} & 3 & 4.14e^{-21} \\ -2.03e^{-26} & 4 & 4.14e^{-21} \\ -4.07e^{-26} & 0 & 4.14e^{-21} \\ -4.07e^{-26} & 1 & 4.14e^{-21} \\ 2.12e^{-31} & -1 & 4.14e^{-21} \end{pmatrix} \quad A_{ArH_2}^T = \begin{pmatrix} 1.08e^{-27} & 0 & 4.14e^{-21} \\ 1.08e^{-27} & 1 & 4.14e^{-21} \\ 1.08e^{-27} & 2 & 4.14e^{-21} \\ -1.64e^{-27} & 0 & 4.14e^{-21} \\ -1.64e^{-27} & 1 & 4.14e^{-21} \\ -3.2e^{-27} & 0 & 4.14e^{-21} \\ 1.49e^{-32} & -1 & 4.14e^{-21} \end{pmatrix}$$

Un algoritmo si occupa di selezionare tra le densità n delle basi per il problema con le densità maggiori tra le specie, di moodo che il sistema sia ben condizionato.
Il sistema non lineare viene risolto dunque per le sole densità della base e i valori per le densità delle specie non-base vengono riaggiornate mediante l'utilizzo della *Mass Action Law* [7]

$$\prod_{i}^{N} n_i^{\nu_i} = \prod_{i}^{N} Q_i^{\nu_i} \tag{3.8}$$

dove ν_i sono i coefficienti di reazione tra i costituenti della miscela e Q_i le funzioni di partizione totali delle specie.
Le funzioni di partizione sono state calcolate in due modi: tramite fitting polinomiale con la temperatura, quando disponibili i coefficienti da lavori precedenti e interpolate linearmente dalle tabelle del Capitelli Chemistry Handbook. [10]

3.3 Termodinamica

Una volta note le composizioni è possibile calcolare l'energia interna e da essa le proprietà termodinamiche con le formule

$$e = \left(\frac{3}{2}\frac{k_B T}{\rho}\sum_{i=1}^{N} n_i\right) + \left(\frac{1}{\rho}\sum_{i=1}^{N} n_i \varepsilon_{0i} - \frac{k_B T}{8\pi\rho r_d^3}\right) + \left(\frac{k_B T}{\rho}\sum_{i=1}^{N} n_i \frac{\partial ln\zeta_i^{int}}{\partial T}\right) \tag{3.9}$$

$$c_p = \left(\frac{\partial e}{\partial T}\right)_P + R\left[1 + \frac{T}{R}\left(\frac{\partial R}{\partial T}\right)_P\right] \tag{3.10}$$

$$c_v = c_p - R\frac{\left[1 + \frac{T}{R}\left(\frac{\partial R}{\partial T}\right)_P\right]^2}{\left[1 - \frac{P}{R}\left(\frac{\partial R}{\partial P}\right)_T\right]} \tag{3.11}$$

$$\gamma = \frac{c_p}{c_v} \tag{3.12}$$

3.4 Collision Integrals calcolati

I *collision integrals* per costruire la matrice 3.1 si ottengono dalla 2.23 con le 2.21 e 2.22 in termini dell'espansione di Sonine. [18]
Alcuni dei collision integrals per le miscele simulate in questo elaborato erano disponibili da lavori precedenti, i mancanti sono stati ottenuti mediante il meto-

do di Colonna, [4] con $\gamma^2 = E/k_B T$ energia ridotta, r distanza tra le particelle, b parametro d'impatto

$$\Omega_{ij}^{mp}(T) = \frac{4(m+1)}{(p+1)![2m+1-(-1)^p]} \int_0^\infty \gamma^{2p+3} Q_{ij}^m(E) e^{-\gamma^2} d\gamma \quad (3.13)$$

$$Q_{ij}^m(b, E) = 2\pi \int_0^\infty b(1 - cos^m\Theta) db \quad (3.14)$$

$$\Theta(b, E) = \pi - 2b \int_{r_m}^\infty \Big(1 - \frac{b^2}{r^2} - \frac{\phi(r)}{E}\Big) \frac{dr}{r^2} \quad (3.15)$$

Θ angolo di deflessione, Q_{ij}^m sezione d'urto di trasporto del m-esimo momento tra le molecole. Per la computazione di questo elaborato nella 3.15 è stato adottato il potenziale di interazione di Capitelli [8] che descrive adeguatamente il comportamento tra le molecole sia a grande che a breve distanza.

$$\phi(r) = \varepsilon_0 \left[\frac{m}{n(x) - m} \left(\frac{1}{x}\right)^{n(x)} - \frac{n(x)}{n(x) - m} \left(\frac{1}{x}\right)^m \right] \quad (3.16)$$

per le interazioni neutro-neutro e neutro-ione con $m = 6$ e $m = 4$ rispettivamente nei due casi, $x = r/r_e$ $n(x) = \beta + 4x^2$, beta calcolabile mediante formule empiriche per cui si rimanda a .
I parametri per 3.16 sono stati presi dalla letteratura [11]
quando disponibili o calcolati mediante formule empiriche dai valori di polarizzabilità espressa in $Angstrom^3$ [9].
Per le interazioni ione-ione viene utilizzato il potenziale schermato di Coulomb, disponibile dai lavori precedenti per diverse combinazioni di carica degli ioni.
Per l'interazione NO-e invece sono state integrate direttamente le differential cross section riportate nel *Plasma exchange Project* **??** mediante la 3.14 in funzione della differential cross section $\sigma(\chi, E)$ definito $\chi = \pi - \Theta$

$$Q_{ij}^m(b, E) = \int_0^\pi \sigma(\chi, E)(1 - cos^m\chi) sin\chi d\chi$$

Una miscela generica possiede

$$\sum_{i=1}^{N} (N - i)$$

interazioni binarie.
Per la miscela ArH2 a sette specie i ventisette integrali collisionali necessari erano già tutti disponibili dal lavoro precedente mentre per la miscela d'Aria a diciassette specie sono stati calcolati in un certo numero.

	N	N+	N+2	N+3	N+4	N2	N2+	NO	NO+	O	O+	O+2	O+3	O+4	O2	O2+	e-
N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
N+		18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
N+2			34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
N+3				49	50	51	52	53	54	55	56	57	58	59	60	61	62
N+4					63	64	65	66	67	68	69	70	71	72	73	74	75
N2						76	77	78	79	80	81	82	83	84	85	86	87
N2+							88	89	90	91	92	93	94	95	96	97	98
NO								99	100	101	102	103	104	105	106	107	108
NO+									109	110	111	112	113	114	115	116	117
O										118	119	120	121	122	123	124	125
O+											126	127	128	129	130	131	132
O+2												133	134	135	136	137	138
O+3													139	140	141	142	143
O+4														144	145	146	147
O2															148	149	150
O2+																151	152
e-																	153

Figura 3: Integrali collisionali per l'aria nel modello in oggetto. In Blu, le collisioni Coulombiane, in Arancio gli integrali collisionali disponibili dai lavori precedenti in giallo quelli calcolati con il metodo descritto.

Parte II
Codice di calcolo e Validazione

4 Re-factoring del codice

Oggetto del tirocinio in preparazione a questo elaborato è stata la re-ingegnerizzazione di un codice di calcolo sviluppato dal Professore E. Ghedini e dal Dottor P. Sanibondi.

L'obiettivo del re-factoring è stato quello di rendere il programma più $user-friendly$, soddisfando due requisiti principali:

- Dati privati accessibili e modificabili *solo* mediante una serie di metodi pubblici.

- Rendere il codice scalabile ad altre miscele, con il minimo numero di modifiche agli statement del corpo del programma.

4.1 Caratteristiche del codice in linguaggio C

Il codice *ArH2transport.c* implementa gli algoritmi per il calcolo della composizione, delle proprietà termodinamiche e di trasporto per la miscela ArH2 a sette specie vista nel capitolo precedente.
Il codice di calcolo sovracitato ha funzionamento seriale, ogni variabile del problema viene inizializzata al momento della computazione necessaria.
La gestione delle quantità vettoriali avviene attraverso delle variabili di tipo *puntatore*.
Un puntatore o *pointer* è una variabile che contiene un indirizzo di memoria del computer, è possibile inizializzare un puntatore con il comando

double p[N];*

sintassi frequente nel programma dato.
Con *p[N]* per l'esattezza si sta creando una variabile pointer che conterrà N indirizzi di memoria, in questo modo, è possibile conservarvi dentro N dati ed è possibile accedervi mediante un iteratore *i=0...N*.
Analogamente, le quantità matriciali vengono gestite con una struttura a doppio puntatore di tipo

*double** p = matrix_alloc(N,M);*

dove N ed M sono rispettivamente il numero di righe e di colonne della matrice.
matrix_alloc(int N, int M) è una funzione di tipo *void* implementata che genera *pointer di pointer*, nella pratica, però, p resta una "struttura unidimensionale" come nell'istruzione per l'inizializzazione del puntatore precedente.
In questo caso, però, i *p[0],p[1],...p[N]* indirizzi di memoria conterranno *p[0],p[1],...p[M]* indirizzi di memoria in cui contenere dunque NxM dati.
Per accedere al dato si usa dunque una sintassi del tipo:

$$p[i][j]$$

```
/*---MATRICE DI COMPOSIZIONE---*/
           /*Ar             H           e-      */
/* e-   */ C[0][0] = 0; C[0][1] = 0; C[0][2] = -1;
/* Ar   */ C[1][0] = 1; C[1][1] = 0; C[1][2] = 0;
/* Ar+  */ C[2][0] = 1; C[2][1] = 0; C[2][2] = 1;
/* Ar+2 */ C[3][0] = 1; C[3][1] = 0; C[3][2] = 2;
/* H    */ C[4][0] = 0; C[4][1] = 1; C[4][2] = 0;
/* H+   */ C[5][0] = 0; C[5][1] = 1; C[5][2] = 1;
/* H2   */ C[6][0] = 0; C[6][1] = 2; C[6][2] = 0;
    double mass[] = {
/* e-   */    9.110e-31,
/* Ar   */    6.6358e-26,
/* Ar+  */    6.6358e-26,
/* Ar+2 */    6.6358e-26,
/* H    */    1.6737e-27,
/* H+   */    1.6737e-27,
/* H2   */    3.3211e-27};
    double e_ion[] = {
/* e-   */0.000000e+00,
/* Ar   */0.000000e+00,
/* Ar+  */2.524968e-18,
/* Ar+2 */6.951732e-18,
/* H    */0.000000e+00,
/* H+   */2.178500e-18,
/* H2   */-7.174875e-19};
```

Figura 4: Porzione del codice in esame

Le funzioni chiamate nel codice sono in gran parte di tipo *void* come quella appena descritta;
in particolare, vengono usate funzioni di tipo *void* per gli algoritmi complessi.
Con questa scelta è stato possibile creare facilmente delle funzioni con cui gestire ordinatamente le quantità vettoriali e matriciali del problema, gestendo quest'ultimo con una struttura piramidale di chiamate di funzioni.

Un esempio esplicativo:

void transport(double Te, double Th, double n, double* x){*
do some calculation...
}

la funzione riceve in input due variabili di tipo double che rappresentano la temperatura elettronica e dei pesanti, e due puntatori, il primo *n* contenente

i dati, già computati, sulla composizione all'attuale temperatura ed x che rappresenta un po' un falso input.
Infatti, la funzione *transport* calcola di fatto le proprietà di trasporto con le formule del capitolo precedente e solo poi le inserisce all'interno del "vettore" x, questo giustifica l'adozione delle funzioni *void*.

Il codice in esame funziona però come un lungo listato, in particolare sono presenti per la gestione dati delle strutture del tipo riportate in figura 4.
È chiaro che l'utilizzo dello stesso per lo studio di una miscela diversa da quella implementata sia altamente tedioso e aperto all'errore.

Questo tipo di architettura, per quanto non generica nè facilmente adattabile all'esecuzione di simulazioni diverse resta la migliore in assoluto in termini di prestazioni.
Il codice in questione computa in poco più di un secondo composizione, termodinamica e proprietà di trasporto di una miscela ArH2 a sette specie per cinque configurazioni di miscela.

- 0.01%H2
- 25%H2
- 50%H2
- 75%H2
- 99.99%H2

La gestione dati degli integrali collisionali viene effettuata caricando i file con nome ad esempio

"Q4th_ari_ar.txt"

per l'interazione tra Argon ionizzato una volta e Argon neutro.
Il caricamento nel programma avviene specificandone ogni nome come un vettore (pointer) di tipo *char* all'interno di una apposita funzione di lettura.

4.2 Caratteristiche del codice in linguaggio C++

Per la soddisfa dei requisiti sovracitati viene fortemente in aiuto la possibilità del linguaggio C++ rispetto al nativo C di sfruttare i paradigmi di programmazione OOP - Object Oriented Programming.
La ridefinizione delle funzioni in classi e oggetti andava eseguita di pari passo con la generalizzazione degli algoritmi implementati a miscele di diverso tipo.

Ridefinire una struttura dati *vettore* o *matrice* avrebbe significato re-inventare tutte le funzioni già implementate per la nuova struttura dati, motivo per cui si è scelto di conservare la struttura a puntatore e doppio puntatore.

La funzione *matrix_alloc(N,M)* è già semi-dinamica, in quanto le dimensioni dell'array vengono definite al momento della chiamata della funzione stessa, per le variabili *vettoriali* invece (pointer singolo) si è scelta un'allocazione più dinamica, tipica del C++, con la possibilità di separare la dichiarazione della variabile pointer dalla definizione delle sue dimensioni mediante l'operatore

$$double^* \;\; p = new \;\; double\;[N].$$

Altrove, dove la struttura pointer è risultata insufficiente, si è fatto invece uso della libreria *vector*, ad esempio, per la raccolta e l'ordinamento delle specie e per la definizione delle path di dati di ingresso e uscita dal programma.
Solo con questa accortezza è stata possibile la separazione totale dei file di implementazione header ".h" e C++ ".cpp", ed è stata possibile la definizione ordinata delle classi con attributi privati come da specifica.

La gestione dati degli integrali collisionali viene effettuata mediante definizione tramite stringa della path al file e del nome del file, reso capace di autogenerarsi in fase di lettura grazie all'adozione per le specie di una nomenclatura pseudo IUPAC.
Il nuovo file tipo degli integrali collisionali avrà dunque nome, per riprendere l'esempio del paragrafo precedente,

"Q4th_Ar+_Ar.txt"

Il programma sviluppato genera dunque automaticamente una lista ordinata di file .txt da caricare sfruttando il metodo della classe delle specie *Species.get_formula()* che restituisce una stringa con la sigla (pseudo) IUPAC della specie considerata. Sono state istanziate dunque sei classi principali:

- Species:
  Catalogo delle specie disponibili nel programma.
  Contiene le caratteristiche della specie atomica quali massa, carica, energia di ionizzazione oltre a variabili booleane utili alla definizione e allo svolgimento del problema.
  È possibile selezionare il metodo di computazione delle funzioni di partizione seguendo: il fitting polinomiale con i coefficienti definiti per ogni specie a patto che questi vengano inclusi in catalogo, oppure, interpolando linearmente delle tabelle da file nominate e caricate in modo analogo ai file per gli integrali collisionali (es: "PF_NO+.txt").

- GasMixture:
  Ricevuto in ingresso un vettore di specie le riordina in ordine alfabetico e le punta con un oggetto puntatore locale SPECS, per cui è stato possibile implementare un metodo che permetta la navigazione con iteratore delle specie nel vettore.
  Da qui, è possibile dunque accedere anche ai metodi delle specie.
  Contiene i dati riguardanti il gas sia in ingresso (in veste di finti dati dell'oggetto GasMixture, appartenenti in realtà agli oggetti Species) che in uscita, accessibili mediante una serie di metodi pubblici.
  Implementa il metodo

  write_results()

  che si occupa della scrittura dei risultati della simulazione in maniera ordinata su file .csv

- numerical_tools:
  Di fatto un archivio delle funzioni già implementate nel codice C ma generalizzate il più possibile rispetto la versione precedente.
  È resa classe padre delle seguenti di modo che ne fossero accessibili ed utilizzabili le funzioni membro, catalogate come metodi della classe.

- composition_c:
  Implementa l'algoritmo di Godin per il calcolo della composizione con il solo oggetto GasMixture in ingresso.
  Sovrascrive la composizione in GasMixture _n $[\#/m^3]$.
  Il metodo composition_solve(&gas) è pubblico per garantirne la chiamata alle classi Thermodynamics e transport.

- Thermodynamics:
  Mediante computazione a step delle composizioni per $T \pm dT$ e $p \pm dp$ calcola le derivate parziali in forma di differenze finite centrate per il calcolo delle proprietà termodinamiche definite al capitolo 3.
  Sovrascrive i risultati in GasMixture _TD.

- transport:
  Come il precedente fa uso dei metodi di composition_c per la computazione di derivate alle differenze finite centrate per il calcolo delle proprietà di trasporto.
  Inoltre, gestisce il flusso dati per gli integrali collisionali in ingresso.
  Sovrascrive i risultati in GasMixture _x.

Con l'adozione di questa struttura è stato possibile instaurare una routine di compilazione ed esecuzione del programma per diverse miscele piuttosto semplice rispetto alla versione precedente:

1. Assicurarsi che gli integrali collisionali richiesti siano nella giusta cartella di destinazione.

2. Rinominare cartella e file di output.

3. Definizione delle specie che si vogliono simulare.

4. Definizione delle basi del problema di Godin e dei dati di input di mole fraction con gli appositi metodi.

5. Includere le specie sopradefinite nel vettore di specie precedente l'inizializzazione dell'oggetto GasMixture.

6. Inizializzare l'oggetto GasMixture con il vettore appena descritto e le condizioni iniziali di Temperatura, pressione e il parametro di non equilibrio. ($\theta_{LTE} = 1 \;\; \theta_{NLTE} \neq 1$)

Alla figura 5 è possibile apprezzare la struttura del nuovo codice, di molto semplificata, con input, output e dati da database. Per approfondimenti,
è possibile visionare la documentzione del programma generata con doxygen all'appendice di questo elaborato.

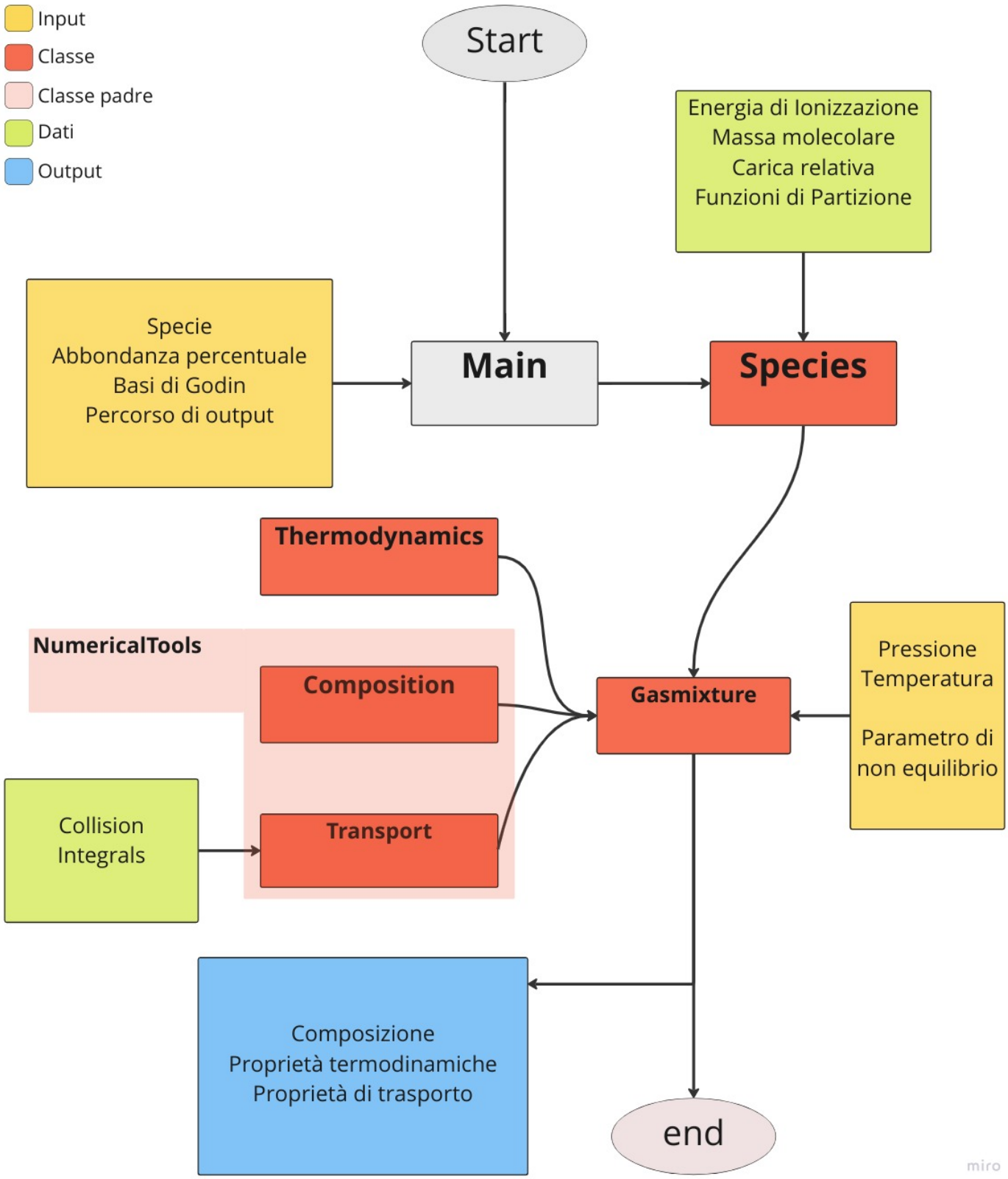


Figura 5: Schema di funzionamento del programma sviluppato

5 Verifica e Validazione

Al fine di validare il codice sviluppato allo stato attuale di questo elaborato sono state svolte ed analizzate le due simulazioni proposte alla sezione 3.2. Entrambe si riferiscono ad un range di temperature compreso tra 300 K^o e 30000 K^o, a pressione atmosferica ed in condizioni di equilibrio

$$\theta = \frac{T_h}{T_e} = 1$$

Per la prima miscela sono state eseguite le stesse cinque configurazioni di miscela del codice originale in linguaggio C.
Per la seconda sono state adottate le composizioni standard dell'atmosfera terrestre, trascurando l'1% di *altri gas* oltre Azoto e Ossigeno.

$$x_{N_2} = 0.79 \qquad x_{O_2} = 0.21$$

5.1 Confronto per una miscela ArH_2

In figura 6 si possono apprezzare le composizioni di sei dei sette costituenti la miscela in esame, per una particolare configurazione di miscela.
Si noti che le linee blu e rosse dei lavori precedente ed attuale rispettivamente si sovrappongono pressochè indistintamente.

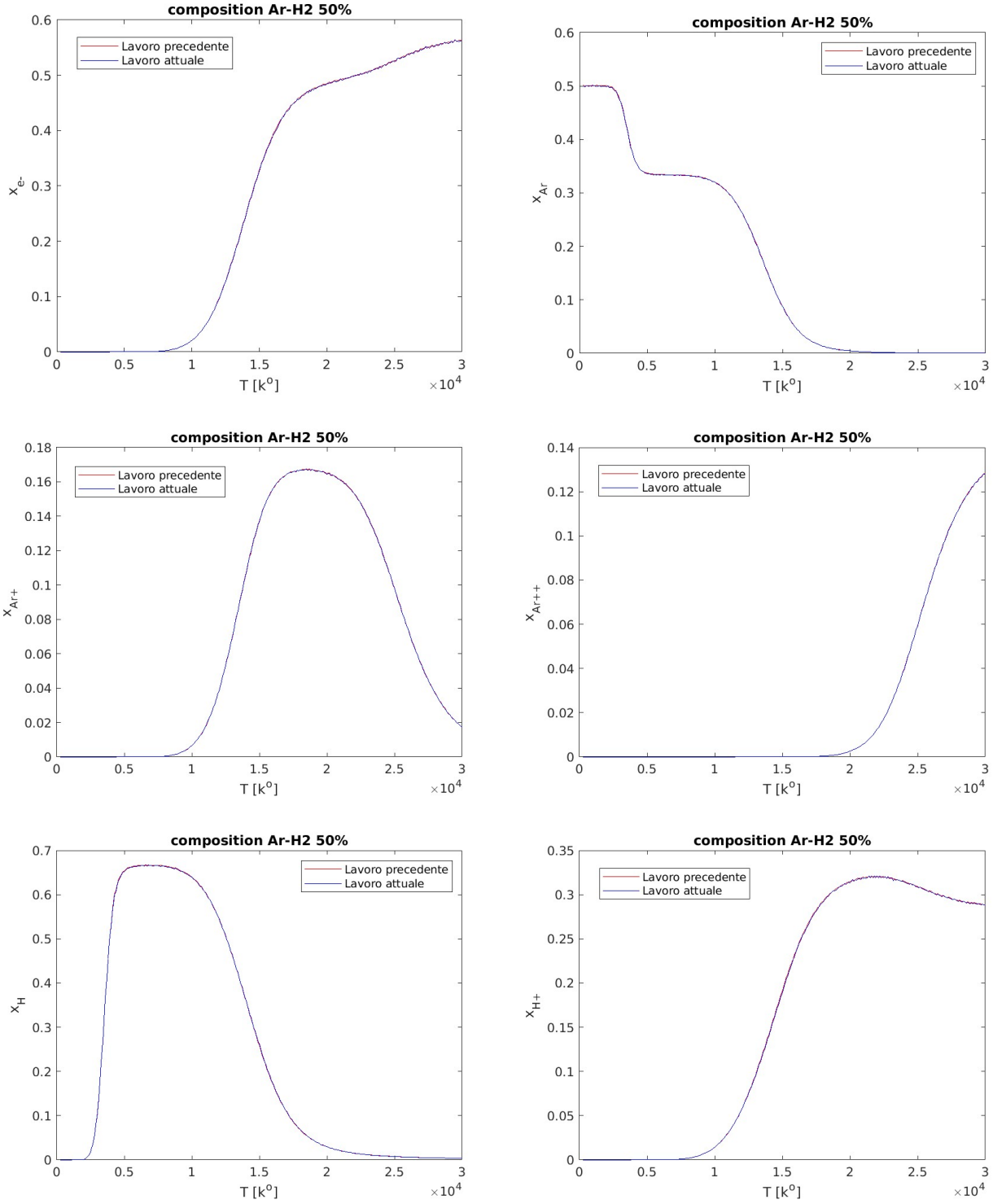
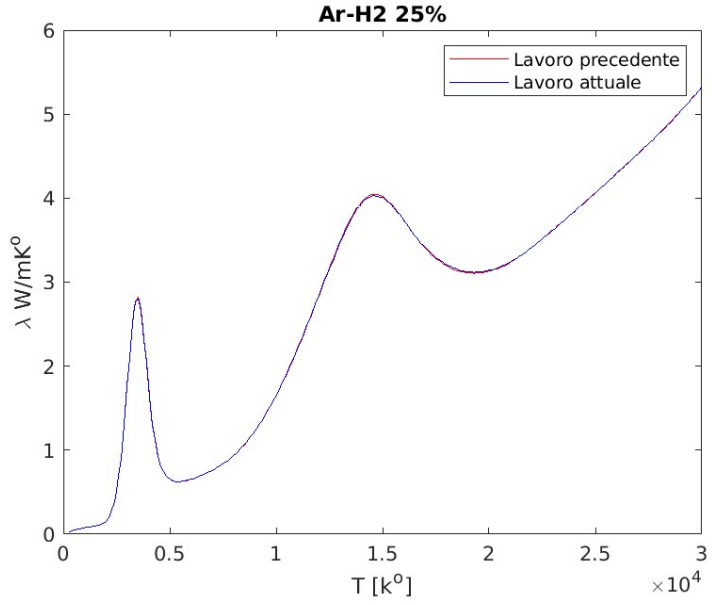
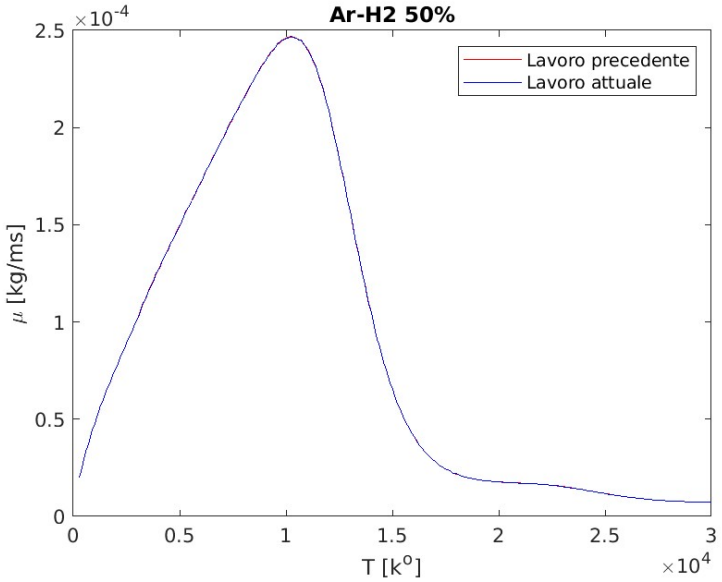


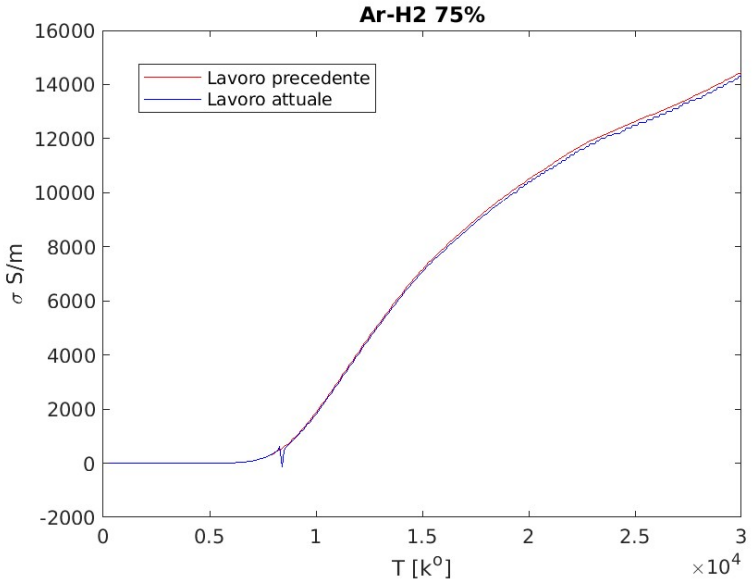
Figura 6: Composizione ArH_2 $x_{H_2} = 50\%$



(a) Conducibilità termica



(b) Viscosità



(c) Conducibilità elettrica

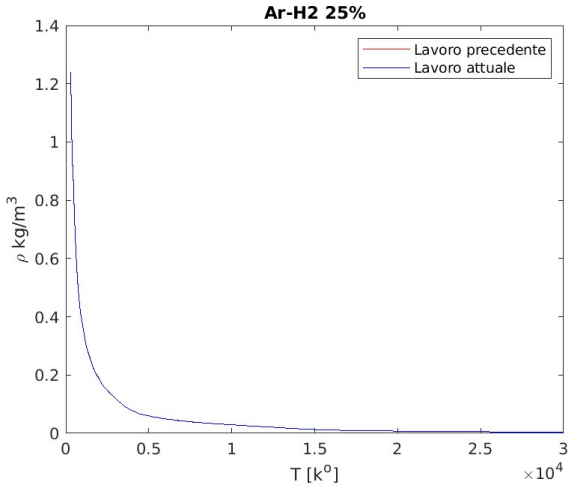
Figura 7: Proprietà di trasporto ArH_2

Si riportano in figura 7 tre diverse proprietà di trasporto per 3 diverse configurazioni di miscela, la conducibilità elettrica per $x_{H_2} = 25\%$, la viscosità per $x_{H_2} = 50\%$ e la conducibilità elettrica per $x_{H_2} = 75\%$, ognuna espressa in unità SI.

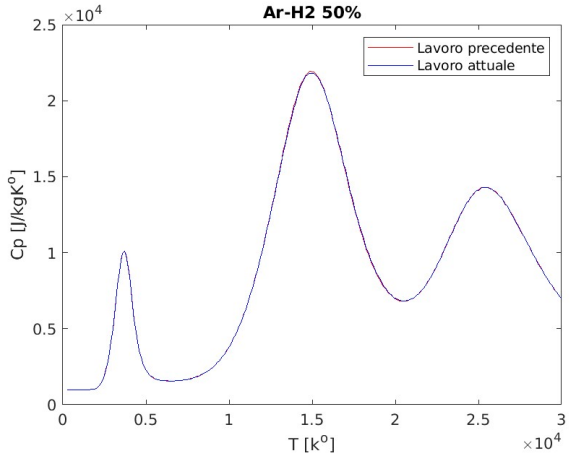
Anche qui le divergenze col lavoro precedente sono quasi nulle, c'è un fenomeno di inversione per alcuni punti di temperatura intorno gli 8000K° per quanto riguarda la conducibilità elettrica.

Anche la simulazione seguente soffrirà della stessa inversione, probabilmente si tratta di un errore di implementazione sistematico. All'attuale stato del codice questo fenomeno non è stato ancora risolto, interessa comunque una porzione molto piccola del range di temperature studiato.

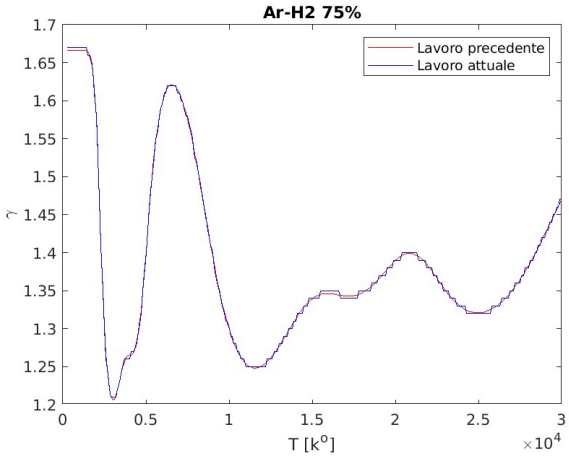
In figura 8 sono presenti invece tre diverse proprietà termodinamiche per le stesse configurazioni di miscela.



(a) Densità



(b) Calore specifico a pressione costante

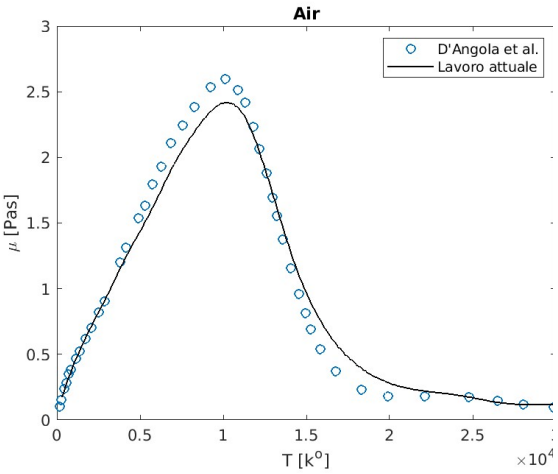


(c) Cp/Cv

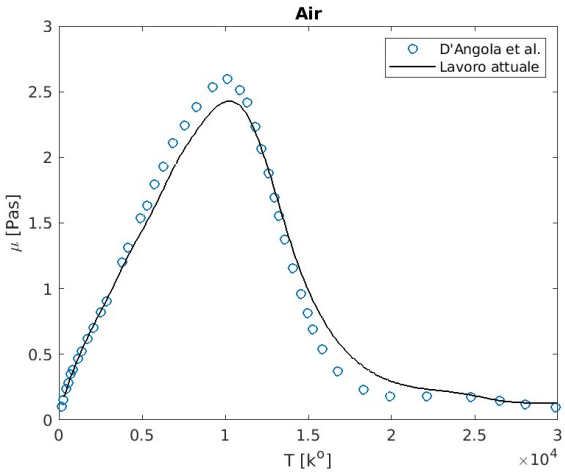
Figura 8: Proprietà termodinamiche ArH_2

5.2 Confronto per l'Aria

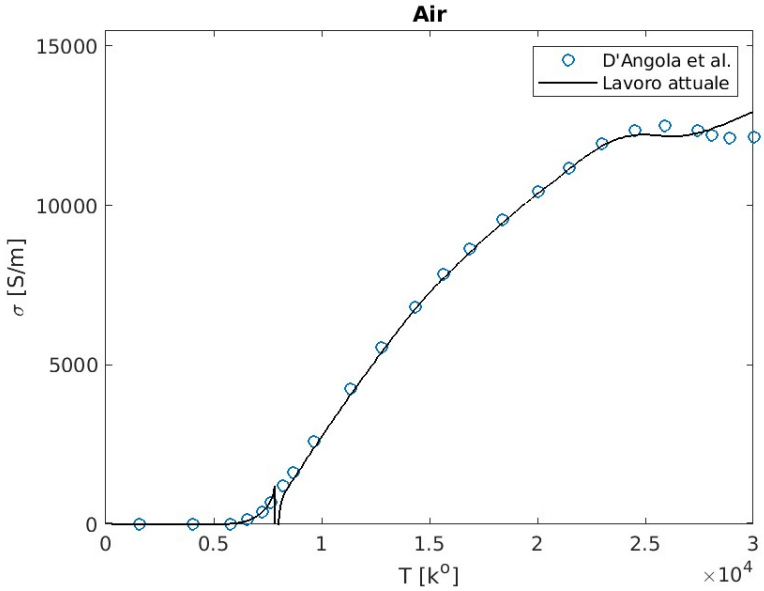
Di seguito, in figura 9 verranno proposti i grafici per le proprietà di trasporto per la miscela aria a diciassette specie in confronto con la simulazione a diciannove specie di D'Angola et.al.
La miscela dello studio di confronto presenta in più rispetto alla computazione oggetto di questo elaborato le specie O^- ed O_2^-.
I dati dello studio di D'Angola et al. sono stati estratti manualmente dai grafici [12] mediante la snap *Engauge digitizer* la loro accuratezza è quindi dipendente *anche* dall'accuratezza con cui è stata eseguita la digitalizzazione.
Alla figura 10 è presente invece la composizione calcolata con l'algoritmo dello studio oggetto di questo elaborato.



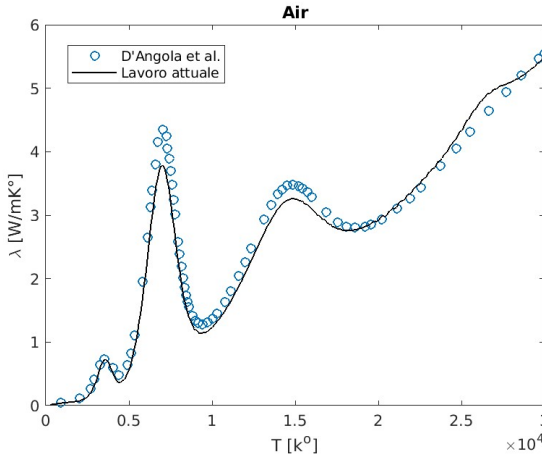
(a) Viscosità 1^oordine



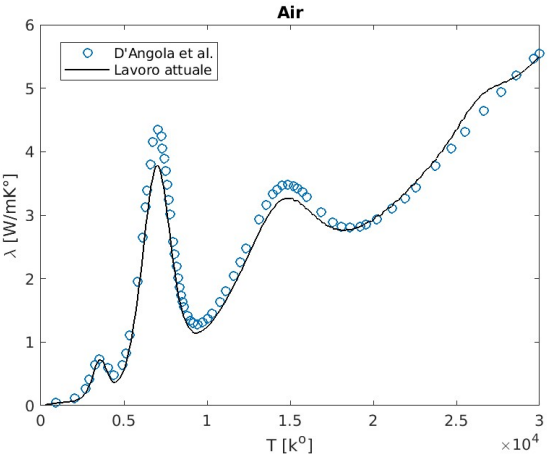
(b) Viscosità 2^oordine



(c) Conducibilità elettrica 4^o ordine



(d) Conducibilità termica λ_h 2^o ordine λ_e 4^o



(e) Conducibilità termica λ_h 4^o ordine λ_e 4^o

Figura 9: Proprietà di trasporto *Aria*

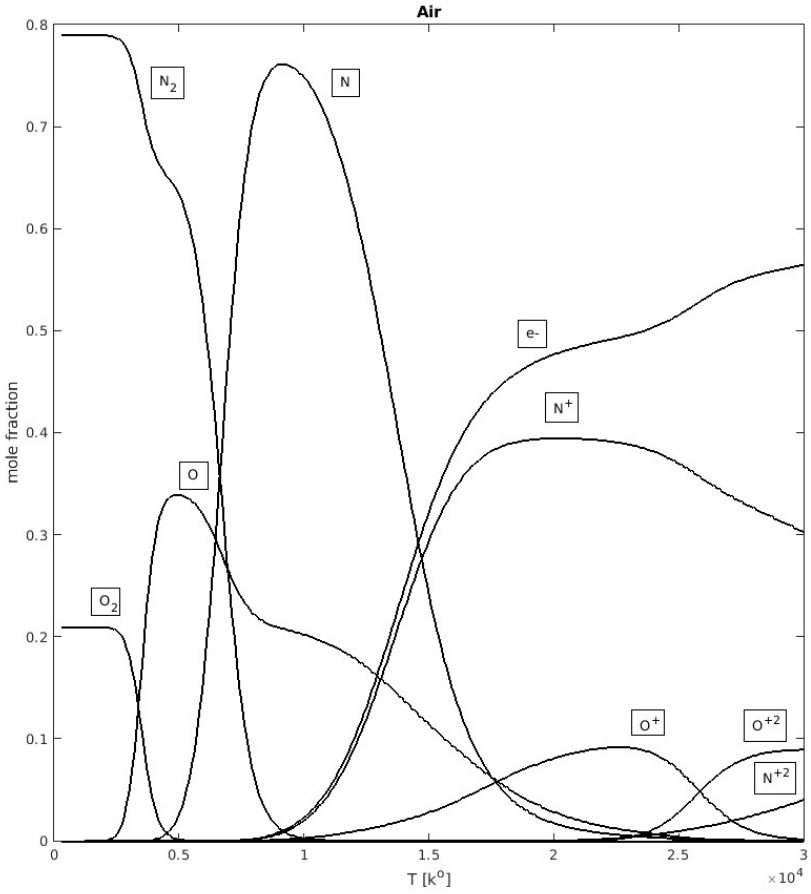


Figura 10: Composizione *Aria*

Si noti come solo una parte delle specie simulate si trovano in misura apprezzabile nel range di temperature studiato.
L'effetto delle loro interazioni tuttavia rientra in 3.3 e contribuisce quindi alla conducibilità termica calcolata.
In figura 9 si nota come gli ordini di approssimazione dei coefficienti di 3.3 sembrino influire poco o nulla sulla magnitudine di conducibilità termica e viscosità.
Ad un'analisi postuma è stato verificato un aumento effettivo del valore della conducibilità termica del 18%, che sul totale non produce un effetto visibile.
Per quanto riguarda la conducibilità elettrica invece si può notare come intorno ai 29000 gradi l'andamento del presente lavoro si discosti da quello di D'Angola et.al.
Questa differenza è dovuta alla omissione nel modello adottato degli anioni O^- ed O_2^- , questi due sottraggono con la loro formazione elettroni liberi alla massa di plasma, riducendone di conseguenza la conducibilità come si nota per il valore di D'Angola, più basso.
Simile interpretazione la si può dare per le divergenze nella conducibilità termica.

6 Conclusioni

La re-ingegnerizzazione del codice di calcolo preesistente ha permesso la definizione delle simulazioni molto più agile e con meno margine di errore umano.
Alla luce dei risultati della prima simulazione in particolare è possibile affermare che la generalizzazione degli algoritmi del codice C in fase di re-ingegnerizzazione è andata a buon fine, salvo il fenomeno di inversione descritto per la conducibilità elettrica, probabilmente un bug di implementazione.
I risultati sono coerenti anche per la seconda simulazione, suggerendo che la computazione degli integrali collisionali descritta alla sezione 3.4 abbia prodotto, a sua volta, dei risultati coerenti. In questo caso, le divergenze riscontrate possono essere giustificate dalla differenza del modello adottato (diciassette specie) rispetto a quello utilizzato per il confronto (diciannove specie).

Il calcolo degli integrali collisionali di cui sopra rappresenta ancora uno scoglio alla fruibilità del codice sviluppato, poichè *conditio sine qua non* per la computazione delle proprietà di trasporto.
Tale computazione, perciò, verrà integrata in futuro all'interno dell'altra, ad esempio sfruttando la validità del potenziale di Capitelli 3.16 nel descrivere la "legge della forza" (*power law*) a cui sono soggette le particelle negli incontri binari.

Sviluppi immediatamente futuri prevedono la simulazione completa e il confronto per la validazione $forte$ del codice sviluppato.
Seguiranno poi simulazioni per miscele non binarie e più complesse, con alto numero di specie, anche implementando, se sarà necessario, le tecniche di parallelizzazione per processi parallelizzabili.

La misurazione sperimentale delle proprietà termodinamiche e di trasporto non è un'operazione semplice, a causa delle elevate temperature, della presenza di specie reattive, di forti campi elettrici per i plasmi ad induzione o delle condizioni dinamiche di un rientro atmosferico, che fanno insistere la condizione di plasma in uno strato sottilissimo a contatto con l'oggetto in caduta.
In sintesi, regimi critici per gli strumenti di misura.
Tali condizioni giustificano l'esistenza e l'importanza dello sviluppare codici di calcolo efficienti e robusti con cui svolgere questo tipo di simulazioni numeriche.

Il codice sviluppato, in particolare, in accordo con una corretta definizione del problema e con il calcolo degli integrali collisionali, è capace di fornire fino al quarto ordine di approssimazione delle proprietà di trasporto di una miscela binaria, spesso assenti in letteratura.
In generale, la più accurata caratterizzazione delle proprietà termodinamiche e di trasporto dei plasmi permette di migliorare ed ampliare i modelli in uso.

Un esempio di utilizzo sono le simulazioni CFD, cui i dati calcolati in questo elaborato possono rappresentare un valido input.

Ringraziamenti

Un ringraziamento al Professor Emanuele Ghedini, relatore di questo elaborato, che mi ha dato l'opportunità di cimentarmi in un problema molto complesso.
Due ringraziamenti speciali ai correlatori, il Dott. La Civita e la Dott.ssa Paponetti, benchè impegnatissimi nei rispettivi progetti mi hanno seguito, accudito, incoraggiato quando credevo di non riuscire, bacchettato nei momenti adeguati.
Senza l'aiuto informatico del Dott. La Civita, il codice sviluppato avrebbe avuto la funzionalità di un compitino di scuola superiore, e senza le correzioni della Dott.ssa Paponetti questo elaborato sarebbe stato un guazzabuglio informe di frasi e definizioni.
Nella mia prossima carriera lavorativa spero di arrivare alla loro età abile e gentile almeno la metà di loro.

7 Bibliografia

Riferimenti bibliografici

[1] R.S.Devoto. "Transport Properties of Ionized Monoatomic Gases", *The Physics of Fluids*, June 1966

[2] "R.S.Devoto. Simplified Expressions for the Transport Properties of Ionized Monoatomic Gases", *The Physics of Fluids*, October 1967

[3] V.Rat, P.André, J.Aubreton, M.F.Elchinger, P.Fauchais and A.Lefort "Transport properties in a two temperature plasma: Theory and application" 23 July 2001

[4] G.Colonna, A.Laricchiuta "General numerical algorithm for classical collision integral calculation" *Computer Physics Communications*

[5] V.Rat, P.André, J.Aubreton, M.F.Elchinger, P.Fauchais and A.Lefort "Two-Temperature Transport Coefficients in Argon–Hydrogen Plasmas—II: Inelastic Processes and Influence of Composition" March 12, 2002

[6] Brokaw, R.S. "Thermal conductivity of gas mixture in chemical equilibrium." *J. Chem. Phys.* 1960.

[7] D.Godin and J.Y.Trèpanier "A Robust and Efficient Method for the Computation of Equilibrium Composition in Gaseous Mixtures" *Plasma Chemistry and Plasma Processing*,vol.24,No.3,September 2004

[8] A.Laricchiuta, G.Colonna, D.Bruno, R.Celiberto, C.Gorse, F.Pirani, M.Capitelli "Classical trasport collision integrals for Lennard-Jones like phenomenological model potential",*Chemical Physics Letters*,7 August 2007

[9] A.Laricchiuta, D.Bruno, M.Capitelli, C.Catalfamo, R.Celiberto, G.Colonna, P.Diomede, D.Giordano, C.Gorse, S.Longo, D.Pagano, and F.Pirani "High temperature Mars atmosphere. Part I: transport cross sections"

[10] M.Capitelli, G.Colonna, A.D'Angola "Fundamental Aspects of Plasma Chemical Physics"*Springer Series on Atomic, Optical and Plasma Physics*

[11] D.Bruno, M.Capitelli, C.Catalfamo, R.Celiberto, G.Colonna, P.Diomede, D.Giordano, C.Gorse, A.Laricchiuta, S.Longo, D. Pagano, F.Pirani."Transport Properties of High-Temperature Mars-Atmosphere Components",*ESA Communication Production Office*, September 2008 ESTEC, Noordwijk, The Netherlands

[12] A.D'Angola, G.Colonna, C.Gorse, M.Capitelli. "Thermodynamic and transport properties in equilibrium air plasmas in a wide pressure and temperature range" *The European Physical Journal*, 23, November, 2007

[13] V.Colombo, E.Ghedini, P.Sanibondi. "Thermodynamic and transport properties in non-equilibrium argon, oxygen and nitrogen thermal plasmas"*Progress in Nuclear Energy*, 2008

[14] NIST Chemisrty WebBook `https://webbook.nist.gov/chemistry/`

[15] Plasma Data Exchange Project `https://us.lxcat.net/data/set_type.php`

[16] S.Chapman and T.G.Cowling "The Mathematical Theory of Non-Uniform Gases, an account of the kinetic theory of viscosity, thermal conduction and diffusion in gases" *Cambridge University Press* 1970

[17] J.O.Hirschfelder, C.F. Curtiss, R.B. Bird. "Molecular Theory of Gases and Liquids" *John Wiley & Sons* 1955

[18] S.G.Brush "Kinetic Theory vol.3 - The Chapman-Enskog solution of the transport equation for moderately dense gases" *Pergamon Press* 1972

Appendix

Devoto

AUTHOR
Version 0.7.0
Mon Mar 6 2023

README

Devoto-Godin refactoring.

Compilation routine on VS Code: Install all C++ and CMake extensions developed by Microsoft. Build, run and debug using the CMake tab. For further instructions, please take a look at VS Code CMake doc.

Hierarchical Index

Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

GasMixture 33
numerical_tools 49
composition_c 6
transport 84

Species 64
Thermodynamics 75
WallClock 115

Class Index

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

composition_c 6
GasMixture 33
numerical_tools 49
Species 64
Thermodynamics 75
transport 84
WallClock 115

File Index

File List

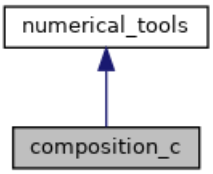
Here is a list of all files with brief descriptions:

main.cpp 119
build/CMakeFiles/3.22.1/CompilerIdC/CMakeCCompilerId.c .... **Error! Bookmark not defined.**
build/CMakeFiles/3.22.1/CompilerIdCXX/CMakeCXXCompilerId.cpp **Error! Bookmark not defined.**
build/CMakeFiles/FindOpenMP/OpenMPCheckVersion.c ..... **Error! Bookmark not defined.**
build/CMakeFiles/FindOpenMP/OpenMPCheckVersion.cpp ..... **Error! Bookmark not defined.**
build/CMakeFiles/FindOpenMP/OpenMPTryFlag.c ..... **Error! Bookmark not defined.**
build/CMakeFiles/FindOpenMP/OpenMPTryFlag.cpp ..... **Error! Bookmark not defined.**
CI_database/final_order.cpp ..... **Error! Bookmark not defined.**
gas/composition_c.cpp 117
gas/composition_c.h 117
gas/gasMixture.cpp 117
gas/gasMixture.h 118
Output_files/Aria_17sp_DANGOLA/dangola_valid.m ..... **Error! Bookmark not defined.**
species/species.cpp 123
species/species.h 124
thermodynamics/thermodynamics.cpp 125
thermodynamics/thermodynamics.h 126
tools/clock.cpp 127
tools/clock.h 127
tools/numerical_tools.cpp 128
tools/numerical_tools.h 129
transport/transport.cpp 130
transport/transport.h 130

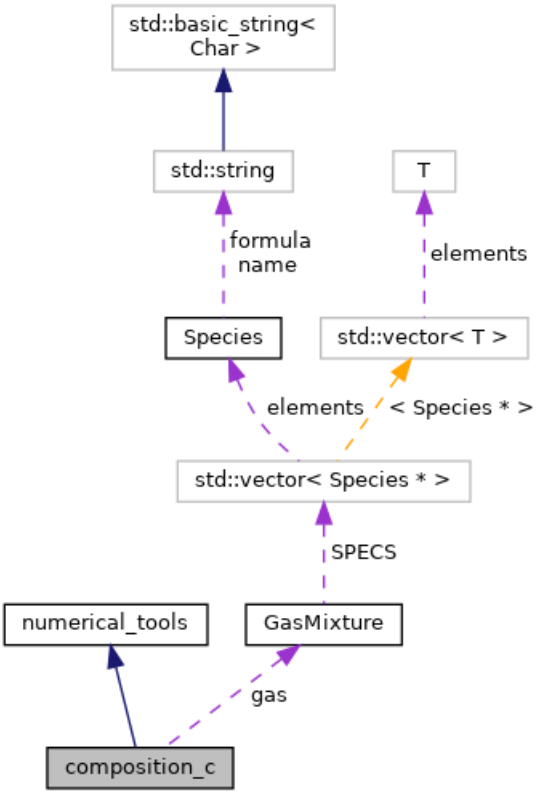
Class Documentation

composition_c Class Reference

`#include <composition_c.h>`
Inheritance diagram for composition_c:



Collaboration diagram for composition_c:



Public Member Functions

- void **create_composition_matrix** ()
  *compute composition matrix with **GasMixture** species*

- void **create_B_Bs_matrix** ()
  creates Base & not-base matrix

- void **create_conservation_system** ()
  creates non-linear system An=A0 to solve

- **composition_c** ()=default

Default constructor.

- **composition_c** (**GasMixture** *gas_i)
  Constructor initialize GODIN composition problem.

- void **composition_solve** ()
- void **solve_from_n0** (double *n0, double *n_step)
- void **update_system** ()
- void **trace** ()
- void **printmat** (double **MAT, int NN, int MM)
- bool **foo** (int **i**, int j, double **N**, double **M**)
- void **sort** (double *arr, int *perm, int n)
- void **matrix_prod** (double ****A**, double ****B**, double ****C**, int N1, int N2)
- double ** **matrix_alloc** (int **N**, int **M**)
  *initialize size of **A objects*

- int ** **matrix_alloc_int** (int n, int m)
  same as matrix_alloc but int variables

- void **matrix_free** (double **matrix, int n)
  clears a matrix

- void **lu_ssolve** (double *x, int *p, double *b0, double **m, int **N**)
  solve lu decomposition of a matrix

- void **lu_inv** (double **AINV, double ****A**, int **N**)
  use lu algorithm to compute the inverse matrix

- void **base_calc** (double *n, int ***b**, int ***bs**, double ****C**, int **N**, int **M**)
  track C rows to set basis & not-basis species based on n density vestor

- void **residual** (double *R, double **J, double *n, double ****A**, double ***A0**, double ****v**, int ***b**, int ***bs**, int **N**, int **M**)
  compute residuals as in GODIN eq.35

- void **lu_sistema** (double *x, double ****A**, double ***b**, int **N**)
  solve an LU system

- double **lu_det** (double ****A**, int **N**)
  *compute det(**A) with LU scomposition*

- double **max_double** (double *arr, int **N**)
  return maximum element of an array

- void **read_Q** (double ***Q, int N_NC, int N_C, int N_TEMP, std::vector< std::string > _filenames)
  reads cross sections data / DA GENERALIZZARE */*

- void **read_Q_coulomb** (double **Qc, std::string path)

reads Coullomb cross sections data / DA GENERALIZZARE */*

- void **il_calc** (int ***i**, int *l, int il, int N_specs)
- int **Qij_calc** (int il, int *Z, int N_specs)
- double **Q_gen** (double TT, double T, double theta, double *n, int *Z, int mp, int **i**, int j, int N_SPC)
  generates firsts collision integrals

- double **Q_coulomb** (double **Qc, double TT, double T, double theta, double *n, int *Z, int mp, int **i**, int j, int N_SPC)
  generates Coulomb collisions integral

- double **interp** (double *x, double *y, double xx, int N_xy)
  interpolator

- double **Diff_bin_rat** (double **Qt, double TT, double T, double theta, double *mass, double nn, int **i**, int j, int N_SPC)
  Binary diffusion as RAT et. al. 30Apr2002.

- double **Qmpil** (double **Qt, int m, int p, int **i**, int l, int N_SPC)
  extract Qt element for every couple interaction i l

- double **Fij_calc** (double **Dbin, double *n, double *mass, double rho, int **i**, int j, int N_SPC)
  Colombo,Ghedini,Sanibondi UNIBO 20/01/2009 eq.17.

- double **Fij_cofactor** (double **Fij, int **i**, int j, int N_SPC)
- int **delta** (int **i**, int j)
  Kronecker delta.

- double **delta_double** (int **i**, int j)
  Kronecker delta.

- double **qsimpmpij** (double **Qt, double *n, int N_SPC, int m, int p)
  simplified Devoto 08/08/1966 Appendix, assumed electron as last specie

- double **qmpij** (double **Qt, double *n, double *mass, int N_SPC, int m, int p, int **i**, int j)
  DEVOTO Appendix 06/1966.

- double **DiffT** (double **Qt, double T, double theta, double *n, double *mass, int N_SPC, int ii)
- double **Diff_amb** (double **D, double T, double theta, double *mass, double *n, int *Z, int ii, int jj, int N_SPC)
  ambipolar diffusion and thermal diffusion

- double **Diff_T_amb** (double **D, double *DT, double T, double theta, double *mass, double *n, int *Z, int ii, int N_SPC)
- double **thermal_cond_el** (double **Qt, double Te, double *n, double *mass, int N_SPC)
  electron thermal conductivity simplified DEVOTO eq.21

- double **thermal_cond_heavy** (double **Qt, double T, double *n, double *mass, int N_SPC)

heavy thermal conductivity 2ND order approx

- double **viscosity** (double **Qt, double T, double *n, double *mass, int N_SPC)
  funzione calcolo viscosita ///1Approx, to further (2nd available) modify M_ORDV

- double **qcapmpij** (double **Qt, double *n, double *mass, int N_SPC, int m, int p, int **i**, int j)
  elementi per calcolo bracket integrals da teoria DEVOTO per viscosità

- double **el_cond** (double **Qt, double T, double theta, double *n, double *mass, int N_SPC)
  DEVOTO simplified electrical conductivity eq.16.

- double **Qeh** (double **Qt, double T, double theta, double *mass, double *n, int N_SPC)
  thermal exchange electron-heavy

- double **Dx_AB_comp** (double **Da, double theta, double *mass, double *n, double *DxDxB, double ***b**, int pp, int N_SPC)
  MURPHY-RAT ordinary diffusion coefficients eq. 2.36.

- double **DYAB_comb** (double DxxAB, double *nf, double *nb, double dya, double *n, double rho, double *mass, int *Z, int p, int N_SPC)
  UNIBO 2008 diffusion coefficients.

- double **DT_AB_comp** (double **Da, double *DT, double Temp, double theta, double rho, double *mass, double *n, double ***b**, double *DxxDT, int p, int N_SPC)
- double **DtAB_comb_y** (double T, double dT, double DtAB, double Dxab, double *n_f, double *n_b, double rho, double *n, double *mass, int *_Z, int p, int N_SPC)

Public Attributes

- **GasMixture** * **gas**
- int **N**
  private pointer

- int **M**
  number of elements in the mixture

- int **L**
  N-M.

- double ** **C**
  composition matrix

- double ** **B**
  matrix of base elements

- double ** **Binv**
  B^-1.

- double ** **Bs**
  matrix of derived by chemical reactions

- double ** **A**
  linear system to solve

- double * **A0**
  RHS A.

- double ** **v**
  *v = Binv*B*

- int * **b**
  vector of base elements it takes the row of the i-th base in the composition matrix

- int * **bs**
  vector of not-base elements

Detailed Description

implementation of GODIN composition computational method 2004. A Robust and Efficient Method for the Computation of Equilibrium Composition in Gaseous Mixtures, D. Godin and J.Y. Trépanier

Definition at line 9 of file composition_c.h.

Constructor & Destructor Documentation

composition_c::composition_c () `[default]`

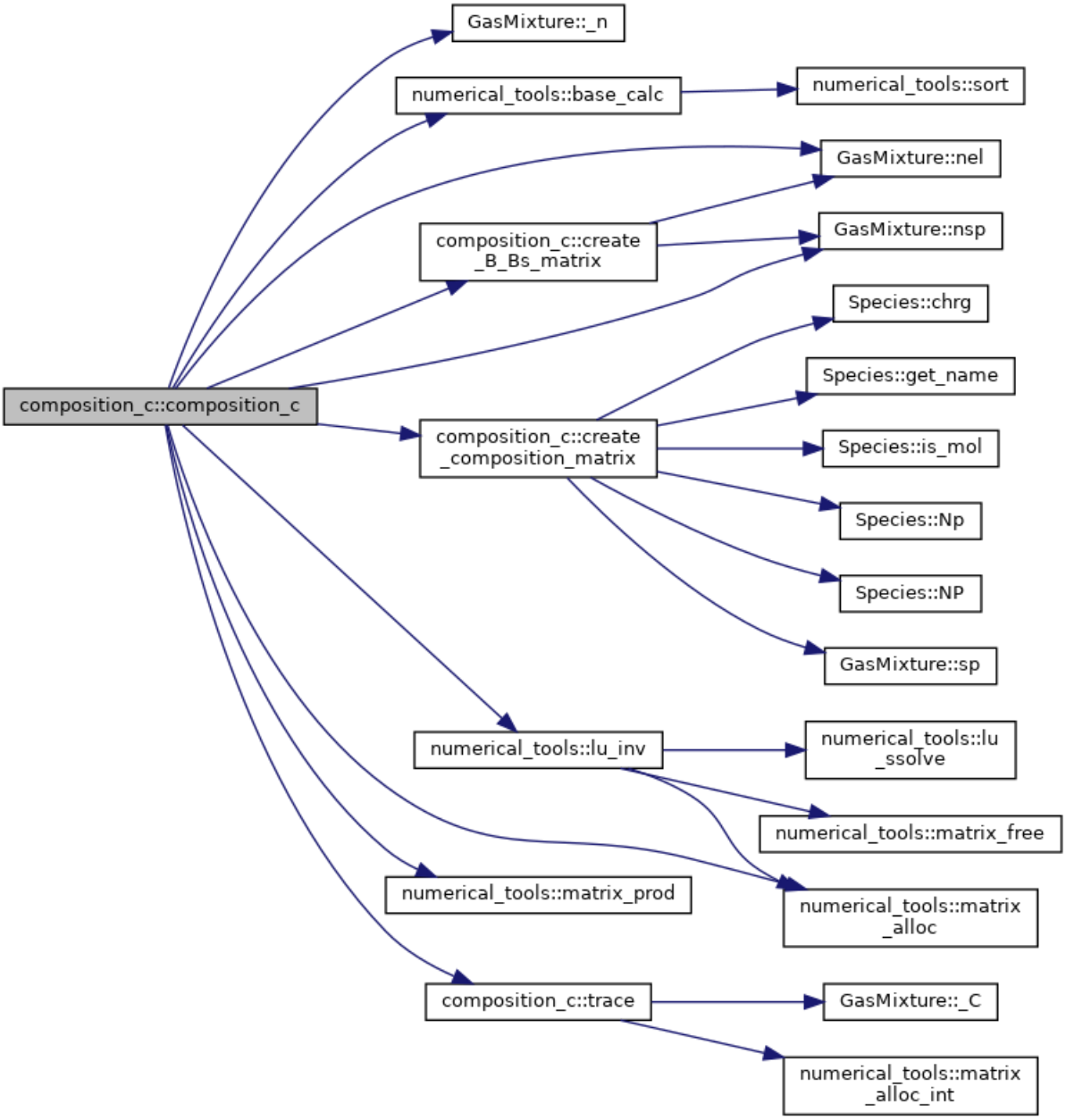
Default constructor.

composition_c::composition_c (GasMixture * *gas_i*)

Constructor initialize GODIN composition problem.

Definition at line 7 of file composition_c.cpp.

Here is the call graph for this function:



Member Function Documentation

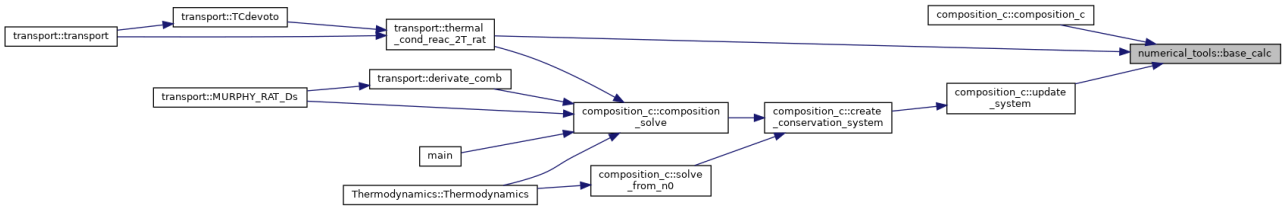
void numerical_tools::base_calc (double * *n*, int * *b*, int * *bs*, double ** *C*, int *N*, int *M*) `[inherited]`

track C rows to set basis & not-basis species based on n density vestor

Definition at line 230 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

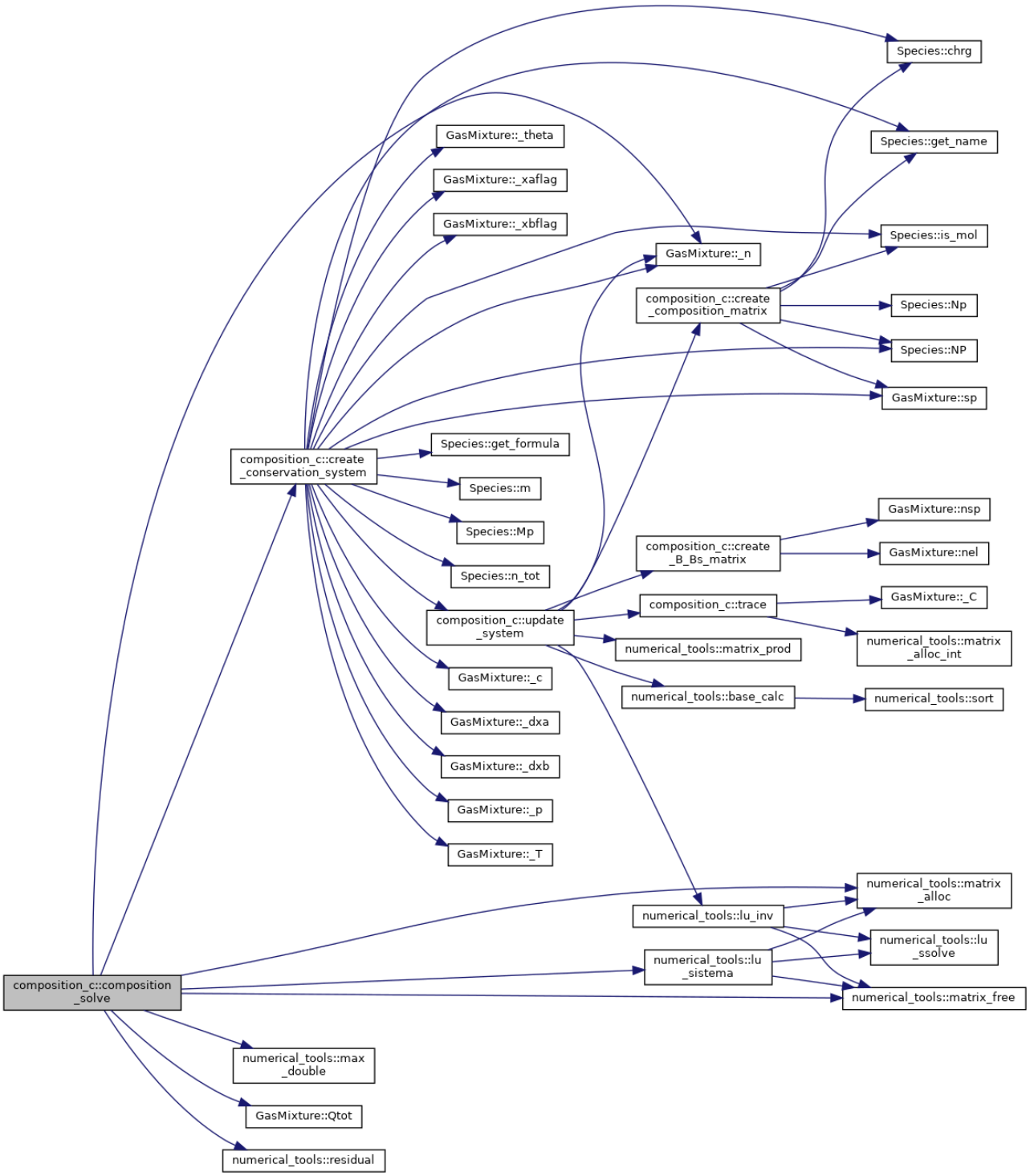


void composition_c::composition_solve ()

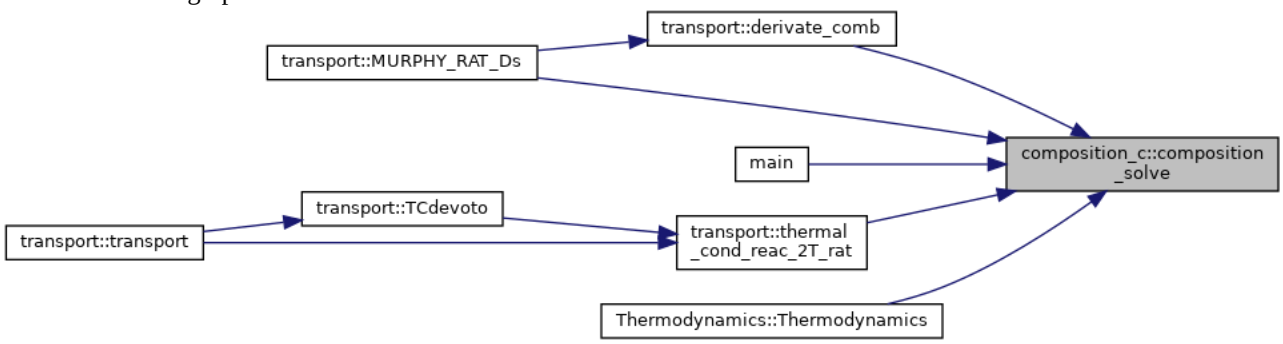
solve base linear system and not-base system, with Newton Method distinguishing base & not-base species as independent GODIN eq. 34 eq.35

Definition at line 308 of file composition_c.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

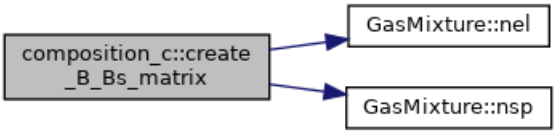


void composition_c::create_B_Bs_matrix ()

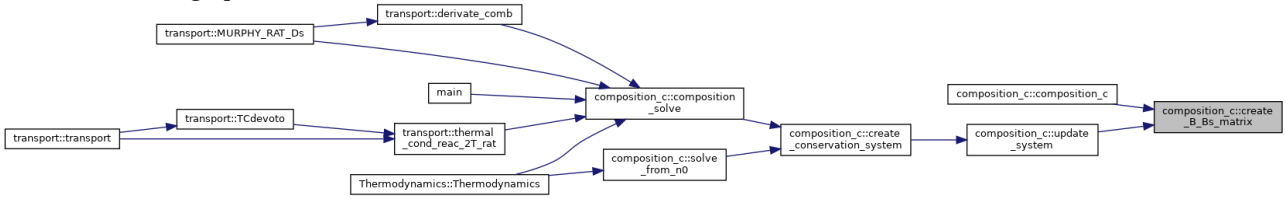
creates Base & not-base matrix

Definition at line 292 of file composition_c.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

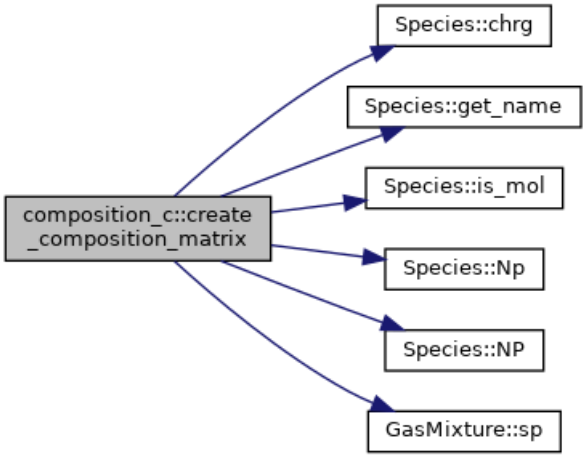


void composition_c::create_composition_matrix ()

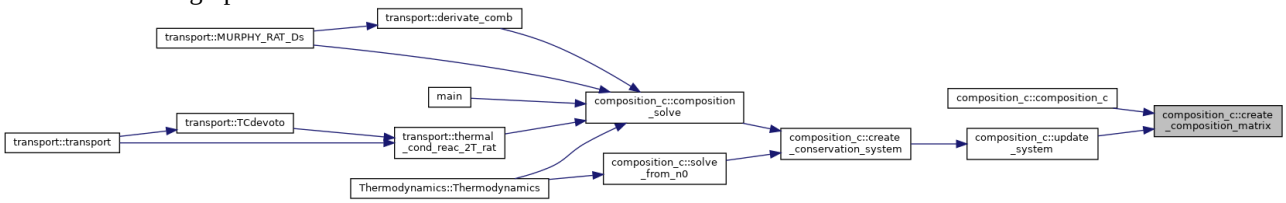
compute composition matrix with **GasMixture** species

Definition at line 74 of file composition_c.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



void composition_c::create_conservation_system ()

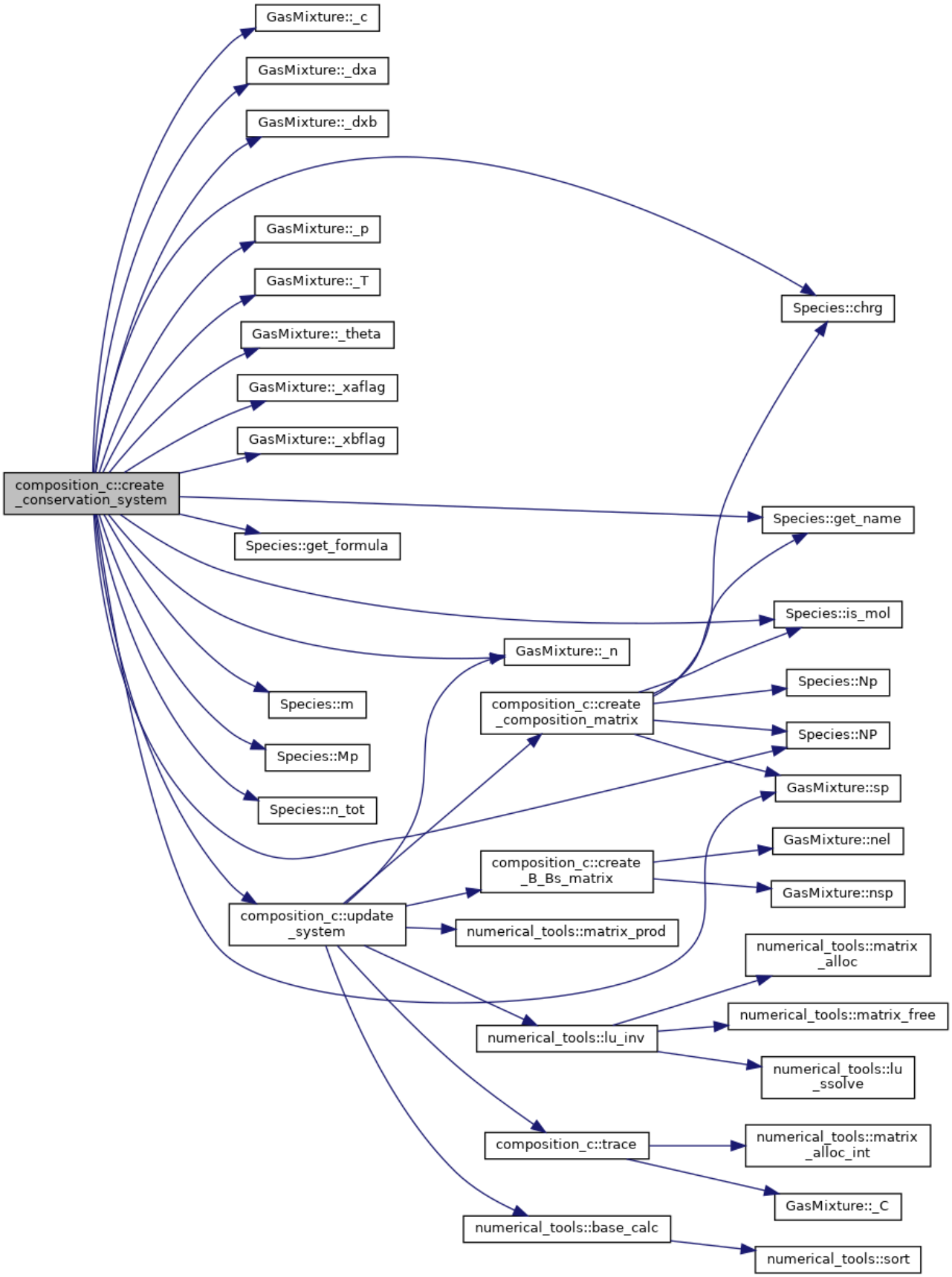
creates non-linear system An=A0 to solve

variante con xB SOLO MISCELE BINARIE

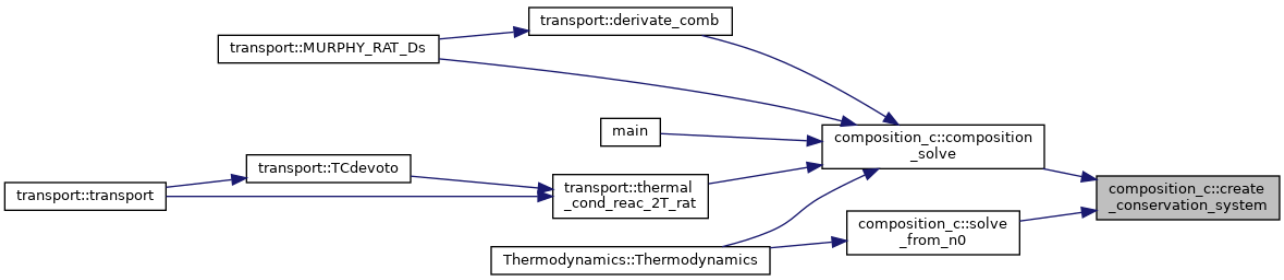
MODIFICARE MANAZZA per ora solo M=3 (elettrone incluso)

Definition at line 128 of file composition_c.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

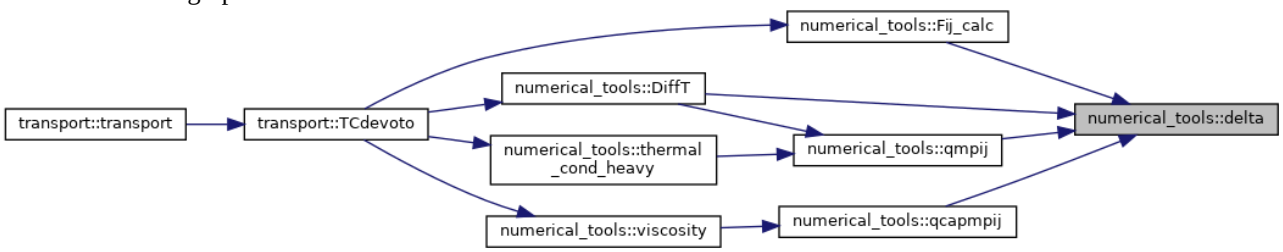


int numerical_tools::delta (int *i*, int *j*) `[inherited]`

Kronecker delta.

Definition at line 886 of file numerical_tools.cpp.

Here is the caller graph for this function:



double numerical_tools::delta_double (int *i*, int *j*) `[inherited]`

Kronecker delta.

Definition at line 892 of file numerical_tools.cpp.

double numerical_tools::Diff_amb (double ** *D*, double *T*, double *theta*, double * *mass*, double * *n*, int * *Z*, int *ii*, int *jj*, int *N_SPC*) `[inherited]`

ambipolar diffusion and thermal diffusion

Definition at line 1417 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::Diff_bin_rat (double ** *Qt*, double *TT*, double *T*, double *theta*, double * *mass*, double *nn*, int *i*, int *j*, int *N_SPC*) `[inherited]`

Binary diffusion as RAT et. al. 30Apr2002.

Definition at line 779 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

double numerical_tools::Diff_T_amb (double ** *D*, double * *DT*, double *T*, double *theta*, double * *mass*, double * *n*, int * *Z*, int *ii*, int *N_SPC*) `[inherited]`

Definition at line 1455 of file numerical_tools.cpp.

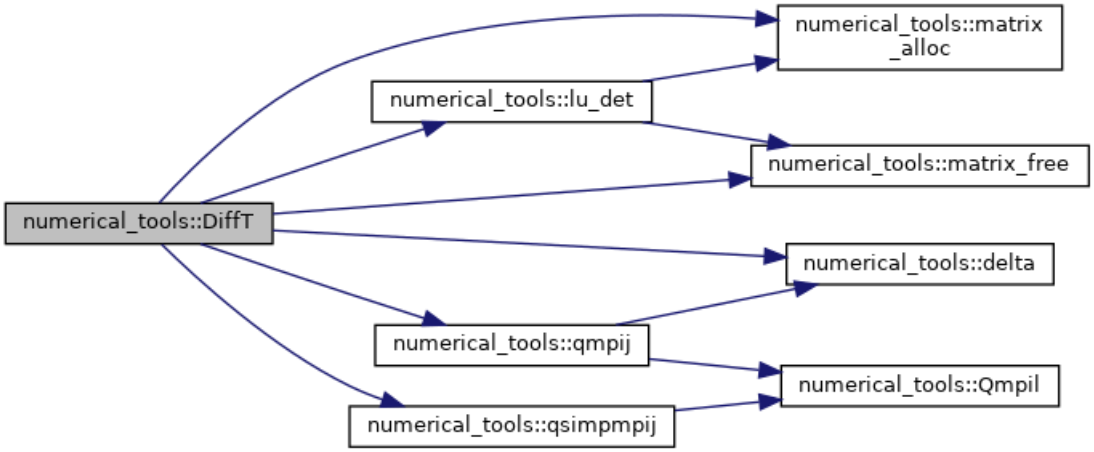
Here is the caller graph for this function:

double numerical_tools::DiffT (double ** *Qt*, double *T*, double *theta*, double * *n*, double * *mass*, int *N_SPC*, int *ii*) `[inherited]`

Temperature Diffusion computed with Devoto 4th approx for electrons and 2nd for heavy particles, modify N_ORD_H for further approx

Definition at line 1342 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

double numerical_tools::DT_AB_comp (double ** *Da*, double * *DT*, double *Temp*, double *theta*, double *rho*, double * *mass*, double * *n*, double * *b*, double * *DxxDT*, int *p*, int *N_SPC*) `[inherited]`

Definition at line 1886 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::DtAB_comb_y (double *T*, double *dT*, double *DtAB*, double *Dxab*, double * *n_f*, double * *n_b*, double *rho*, double * *n*, double * *mass*, int * *_Z*, int *p*, int *N_SPC*) `[inherited]`

Definition at line 1955 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::Dx_AB_comp (double ** *Da*, double *theta*, double * *mass*, double * *n*, double * *DxDxB*, double * *b*, int *pp*, int *N_SPC*) `[inherited]`

MURPHY-RAT ordinary diffusion coefficients eq. 2.36.

Definition at line 1746 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::DYAB_comb (double *DxxAB*, double * *nf*, double * *nb*, double *dya*, double * *n*, double *rho*, double * *mass*, int * *Z*, int *p*, int *N_SPC*) `[inherited]`

UNIBO 2008 diffusion coefficients.

Definition at line 1826 of file numerical_tools.cpp.

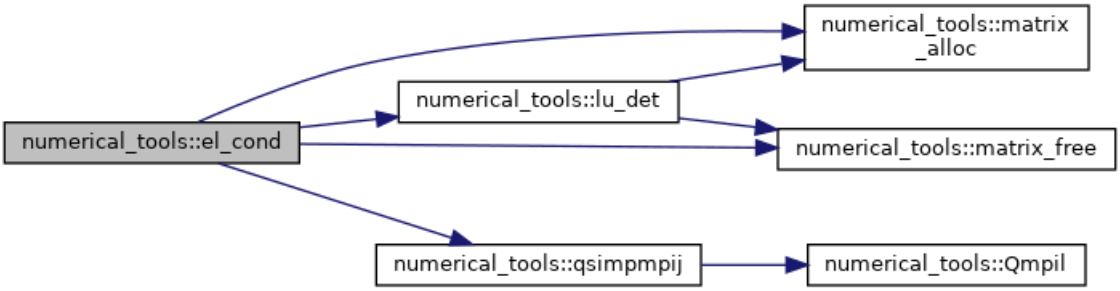
Here is the caller graph for this function:

double numerical_tools::el_cond (double ** *Qt*, double *T*, double *theta*, double * *n*, double * *mass*, int *N_SPC*) `[inherited]`

DEVOTO simplified electrical conductivity eq.16.

Definition at line 1688 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

double numerical_tools::Fij_calc (double ** *Dbin*, double * *n*, double * *mass*, double *rho*, int *i*, int *j*, int *N_SPC*) `[inherited]`

Colombo,Ghedini,Sanibondi UNIBO 20/01/2009 eq.17.

Definition at line 842 of file numerical_tools.cpp.

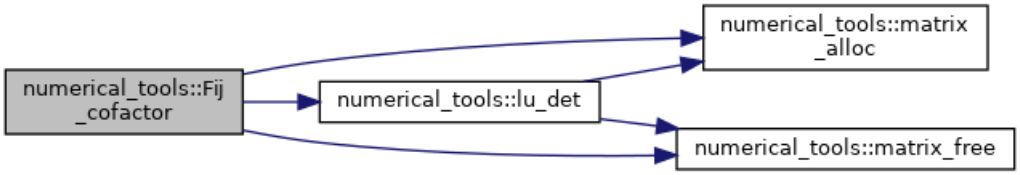
Here is the call graph for this function:

Here is the caller graph for this function:

double numerical_tools::Fij_cofactor (double ** *Fij*, int *i*, int *j*, int *N_SPC*) [inherited]

Definition at line 856 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

bool numerical_tools::foo (int *i*, int *j*, double *N*, double *M*) [inherited]

void numerical_tools::il_calc (int * *i*, int * *I*, int *il*, int *N_specs*) [inherited]

Associates species couple indexes to binary interaction progressively numerated (il)

Definition at line 502 of file numerical_tools.cpp.

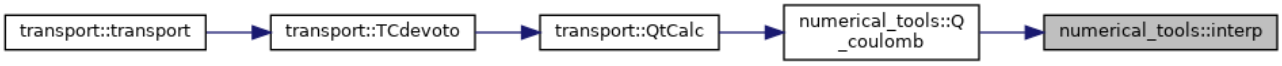
Here is the caller graph for this function:

double numerical_tools::interp (double * *x*, double * *y*, double *xx*, int *N_xy*) [inherited]

interpolator

Definition at line 742 of file numerical_tools.cpp.

Here is the caller graph for this function:

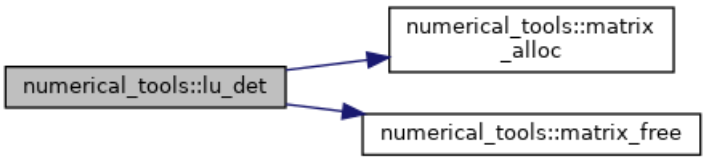


double numerical_tools::lu_det (double ** *A*, int *N*) [inherited]

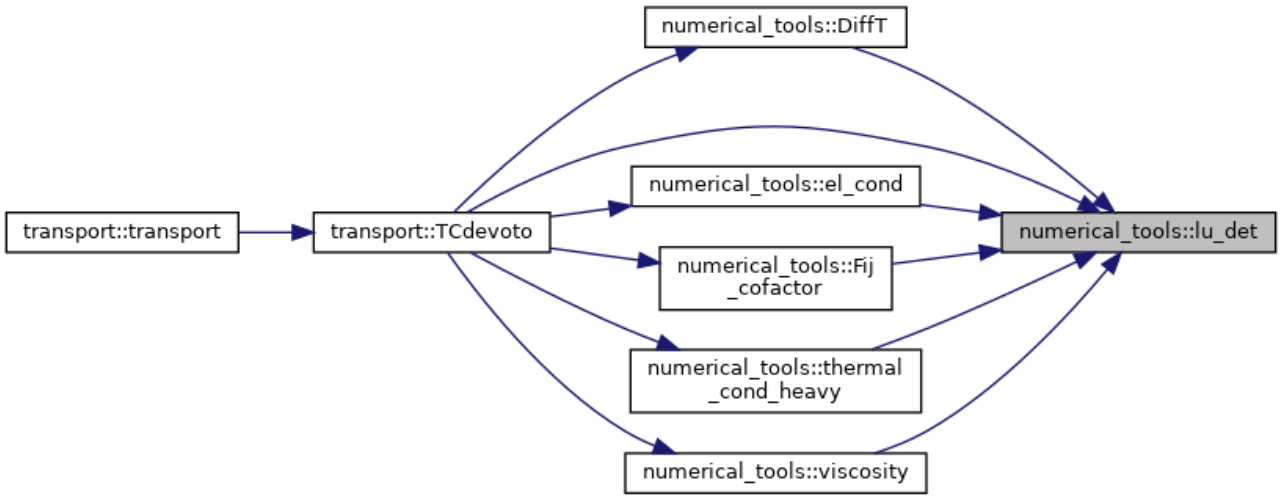
compute det(**A) with LU scomposition

Definition at line 372 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

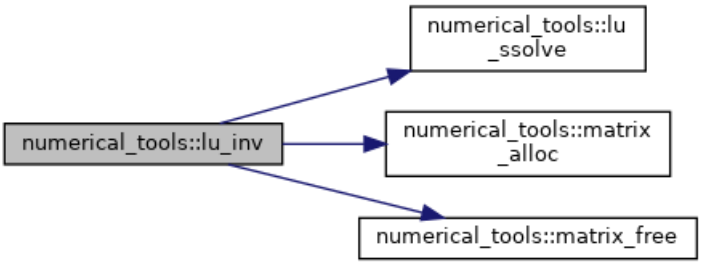


void numerical_tools::lu_inv (double ** *AINV*, double ** *A*, int *N*) `[inherited]`

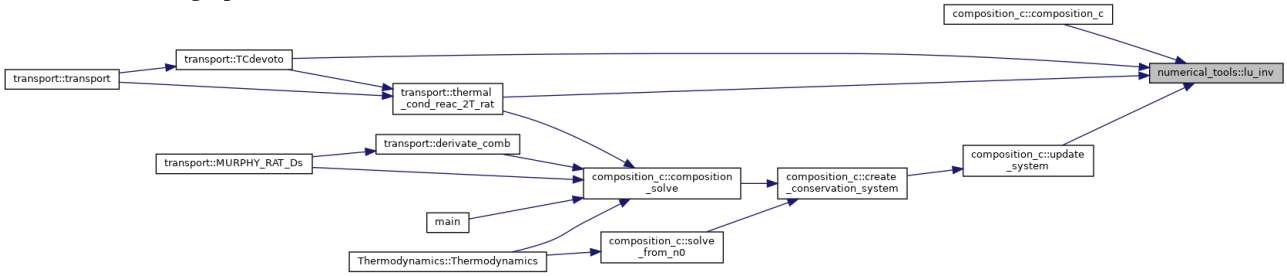
use lu algorithm to compute the inverse matrix

Definition at line 167 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

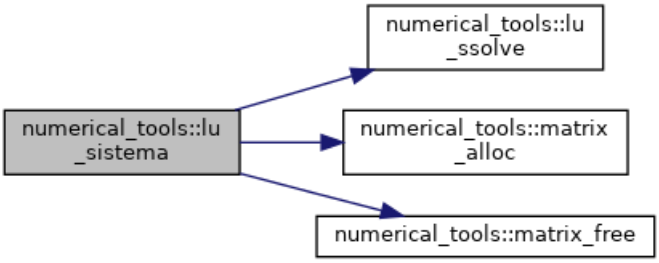


void numerical_tools::lu_sistema (double * *x*, double ** *A*, double * *b*, int *N*) `[inherited]`

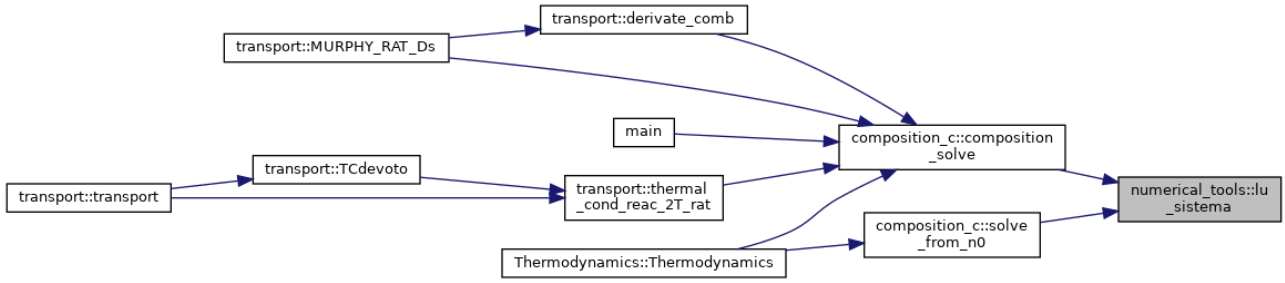
solve an LU system

Definition at line 321 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

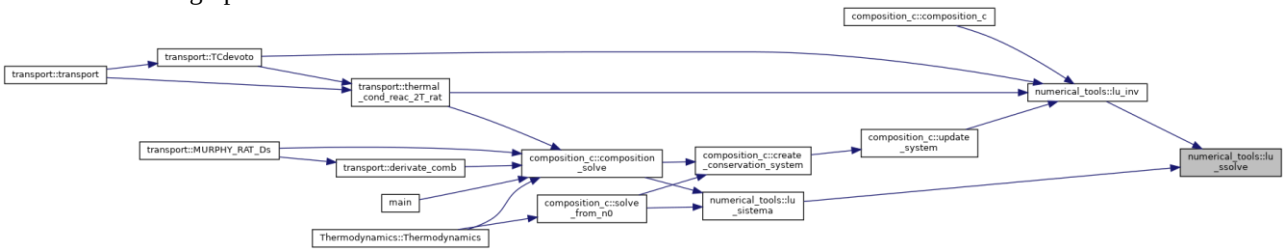


void numerical_tools::lu_ssolve (double * *x*, int * *p*, double * *b0*, double ** *m*, int *N*) `[inherited]`

solve lu decomposition of a matrix

Definition at line 122 of file numerical_tools.cpp.

Here is the caller graph for this function:

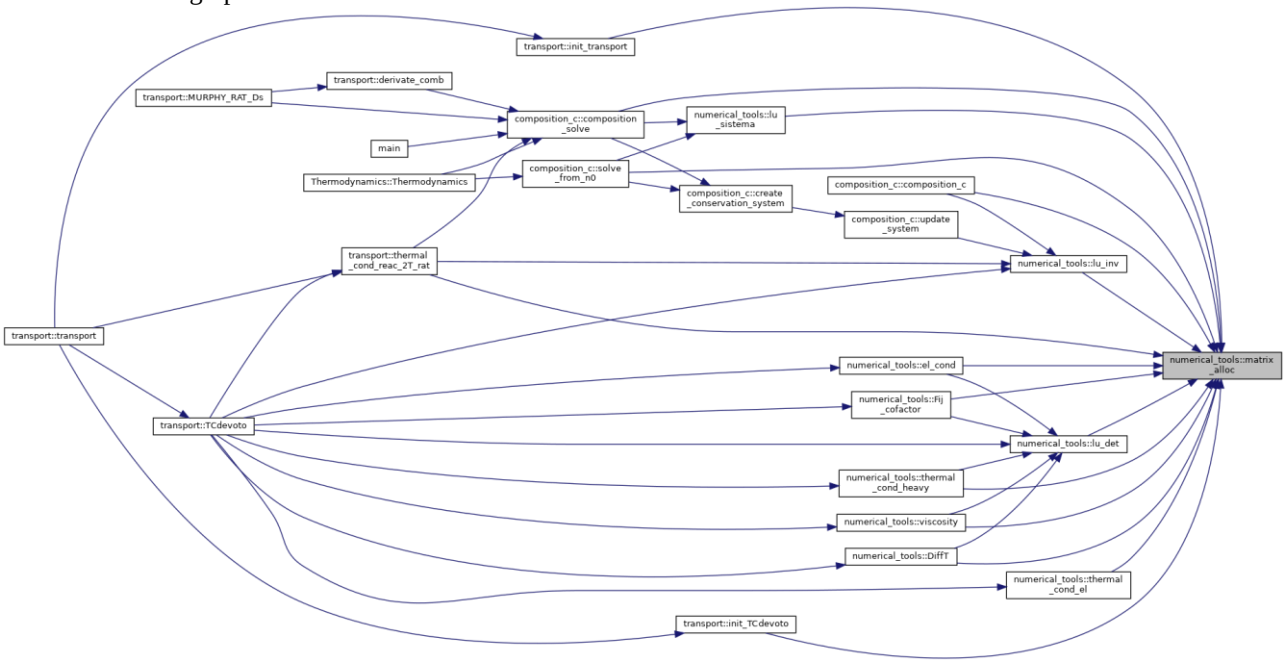


double ** numerical_tools::matrix_alloc (int *N*, int *M*) `[inherited]`

initialize size of **A objects

Definition at line 84 of file numerical_tools.cpp.

Here is the caller graph for this function:

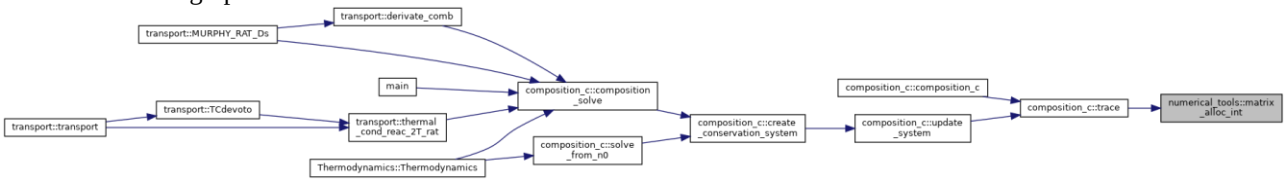


int ** numerical_tools::matrix_alloc_int (int *n*, int *m*) `[inherited]`

same as matrix_alloc but int variables

Definition at line 97 of file numerical_tools.cpp.

Here is the caller graph for this function:

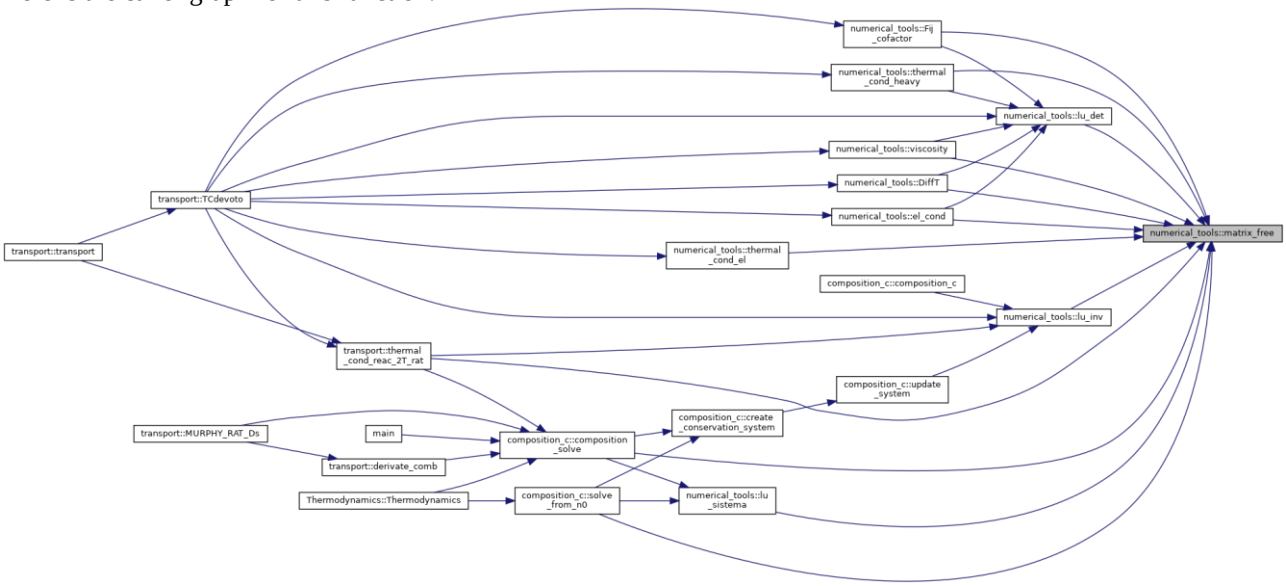


void numerical_tools::matrix_free (double ** *matrix*, int *n*) [inherited]

clears a matrix

Definition at line 111 of file numerical_tools.cpp.

Here is the caller graph for this function:

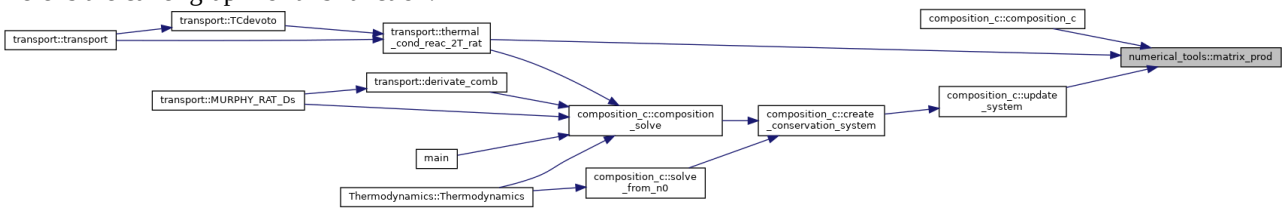


void numerical_tools::matrix_prod (double ** *A*, double ** *B*, double ** *C*, int *N1*, int *N2*) [inherited]

compute matrix product A = B*C

Definition at line 63 of file numerical_tools.cpp.

Here is the caller graph for this function:

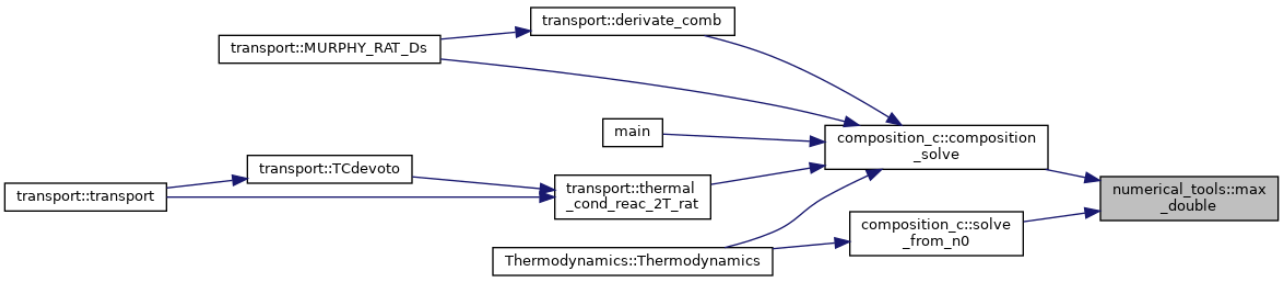


double numerical_tools::max_double (double * *arr*, int *N*) [inherited]

return maximum element of an array

Definition at line 432 of file numerical_tools.cpp.

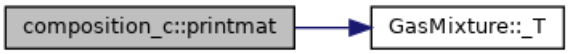
Here is the caller graph for this function:



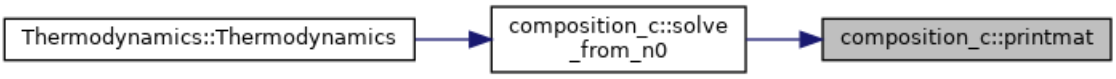
void composition_c::printmat (double ** *MAT*, int *NN*, int *MM*)

Definition at line 505 of file composition_c.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



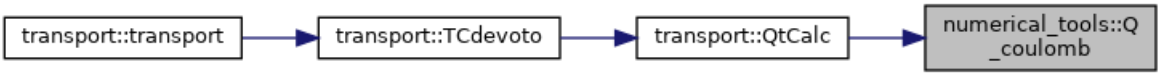
double numerical_tools::Q_coulomb (double ** *Qc*, double *TT*, double *T*, double *theta*, double * *n*, int * *Z*, int *mp*, int *i*, int *j*, int *N_SPC*) `[inherited]`

generates Coulomb collisions integral

Definition at line 683 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:



double numerical_tools::Q_gen (double *TT*, double *T*, double *theta*, double * *n*, int * *Z*, int *mp*, int *i*, int *j*, int *N_SPC*) `[inherited]`

generates firsts collision integrals

Definition at line 553 of file numerical_tools.cpp.

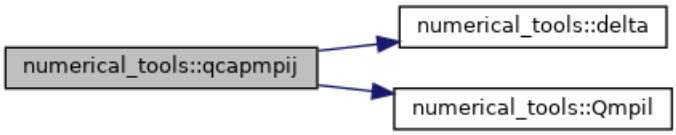
Here is the caller graph for this function:

double numerical_tools::qcapmpij (double ** *Qt*, double * *n*, double * *mass*, int *N_SPC*, int *m*, int *p*, int *i*, int *j*) `[inherited]`

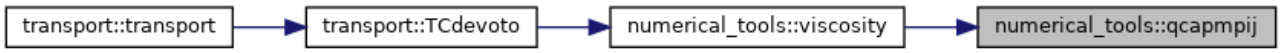
elementi per calcolo bracket integrals da teoria DEVOTO per viscosità

Definition at line 1580 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



double numerical_tools::Qeh (double ** *Qt*, double *T*, double *theta*, double * *mass*, double * *n*, int *N_SPC*) `[inherited]`

thermal exchange electron-heavy

Definition at line 1729 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

int numerical_tools::Qij_calc (int *il*, int * *Z*, int *N_specs*) `[inherited]`

associates to il the Qij coulomb interaction progressively numerated when Z[i]*Z[j]==0

Definition at line 522 of file numerical_tools.cpp.

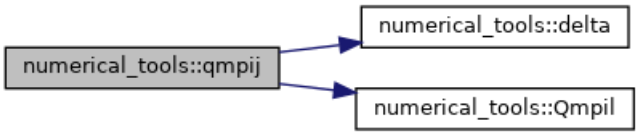
Here is the caller graph for this function:

double numerical_tools::qmpij (double ** *Qt*, double * *n*, double * *mass*, int *N_SPC*, int *m*, int *p*, int *i*, int *j*) `[inherited]`

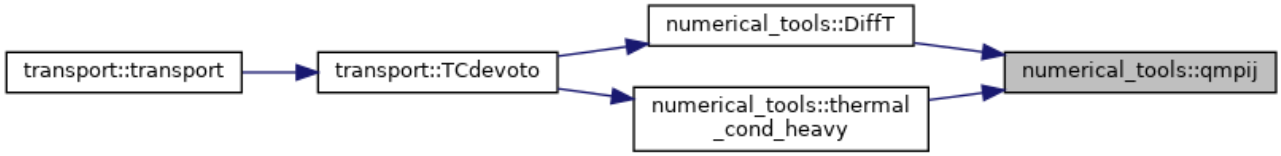
DEVOTO Appendix 06/1966.

Definition at line 1063 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

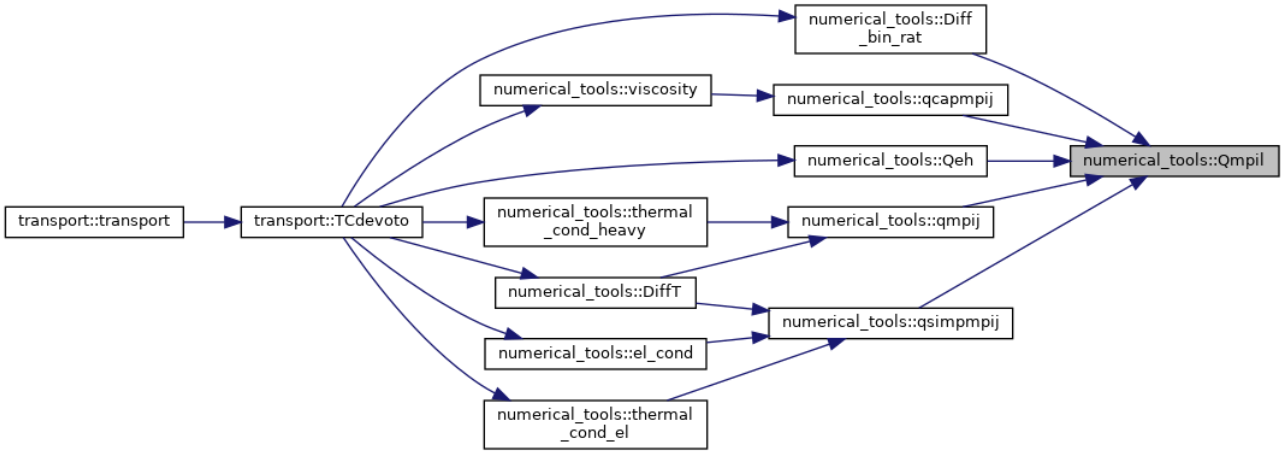


double numerical_tools::Qmpil (double ** *Qt*, int *m*, int *p*, int *i*, int *l*, int *N_SPC*) `[inherited]`

extract Qt element for every couple interaction i l

Definition at line 801 of file numerical_tools.cpp.

Here is the caller graph for this function:



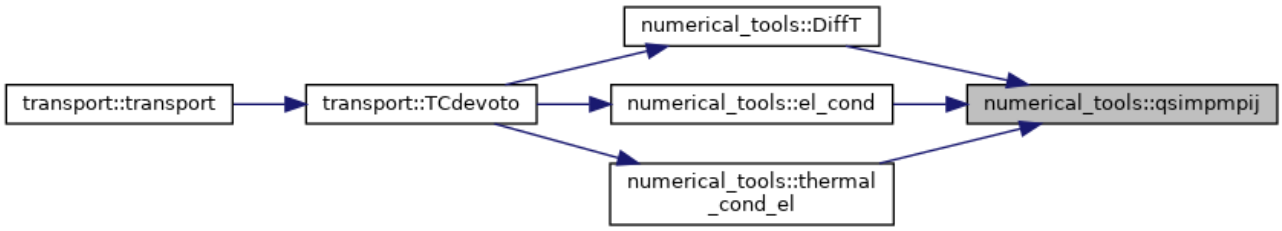
double numerical_tools::qsimpmpij (double ** *Qt*, double * *n*, int *N_SPC*, int *m*, int *p*) [inherited]

simplified Devoto 08/08/1966 Appendix, assumed electron as last specie

Definition at line 900 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

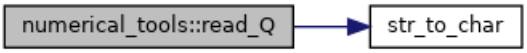


void numerical_tools::read_Q (double * *Q*, int *N_NC*, int *N_C*, int *N_TEMP*, std::vector< std::string > *_filenames*) [inherited]**

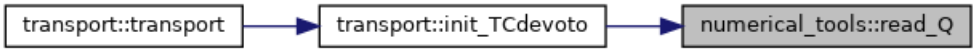
reads cross sections data /* DA GENERALIZZARE */

Definition at line 446 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

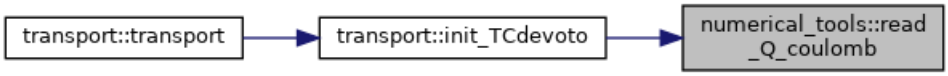


void numerical_tools::read_Q_coulomb (double ** *Qc*, std::string *path*) [inherited]

reads Coullomb cross sections data /* DA GENERALIZZARE */

Definition at line 480 of file numerical_tools.cpp.

Here is the caller graph for this function:

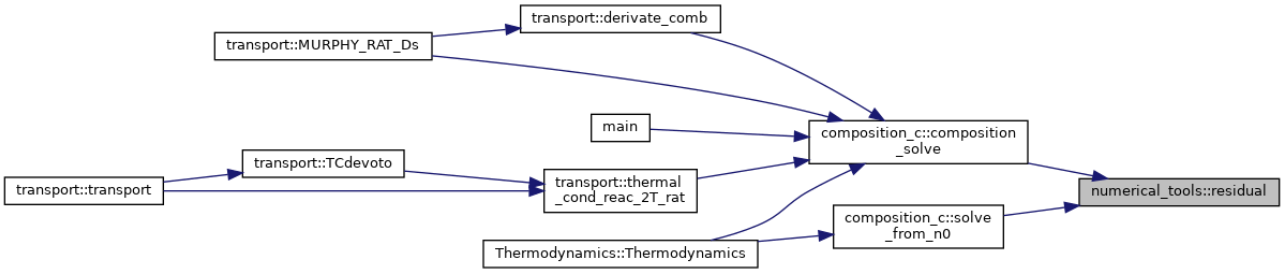


void numerical_tools::residual (double * *R*, double ** *J*, double * *n*, double ** *A*, double * *A0*, double ** *v*, int * *b*, int * *bs*, int *N*, int *M*) `[inherited]`

compute residuals as in GODIN eq.35

Definition at line 282 of file numerical_tools.cpp.

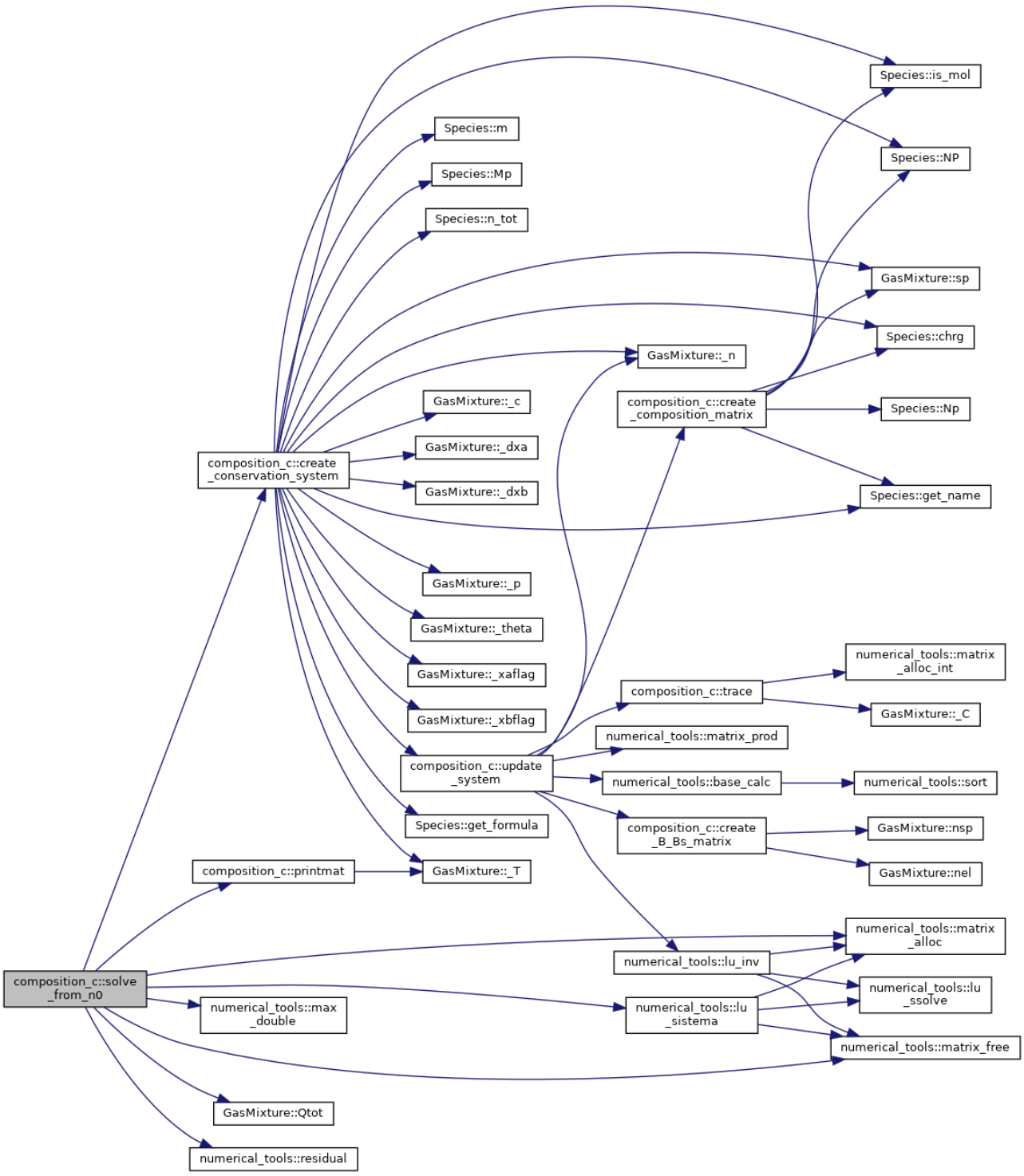
Here is the caller graph for this function:



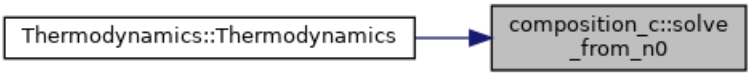
void composition_c::solve_from_n0 (double * *n0*, double * *n_step*)

Definition at line 391 of file composition_c.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

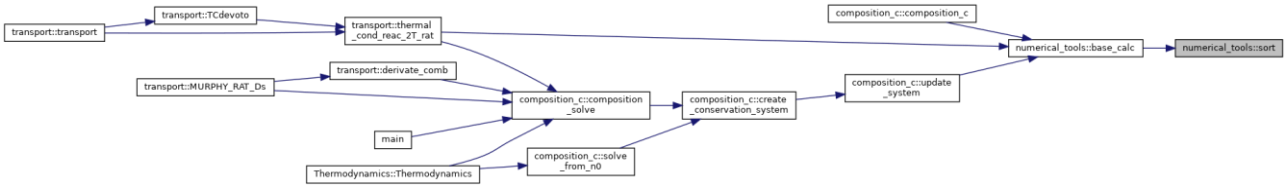


void numerical_tools::sort (double * *arr*, int * *perm*, int *n*) `[inherited]`

indices of arr elements in descent order initialize a vector whose component are

Definition at line 37 of file numerical_tools.cpp.

Here is the caller graph for this function:

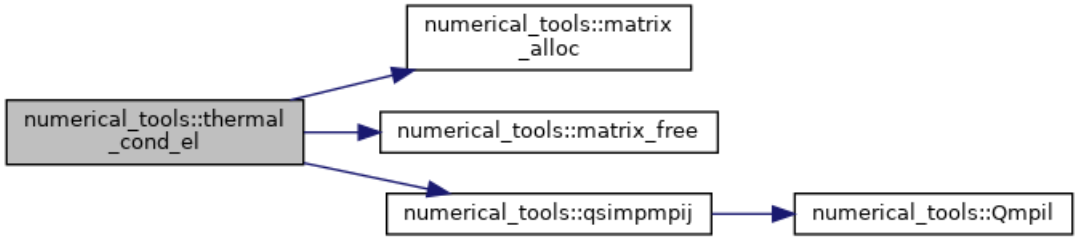


double numerical_tools::thermal_cond_el (double ** *Qt*, double *Te*, double * *n*, double * *mass*, int *N_SPC*) [inherited]

electron thermal conductivity simplified DEVOTO eq.21

Definition at line 1492 of file numerical_tools.cpp.

Here is the call graph for this function:



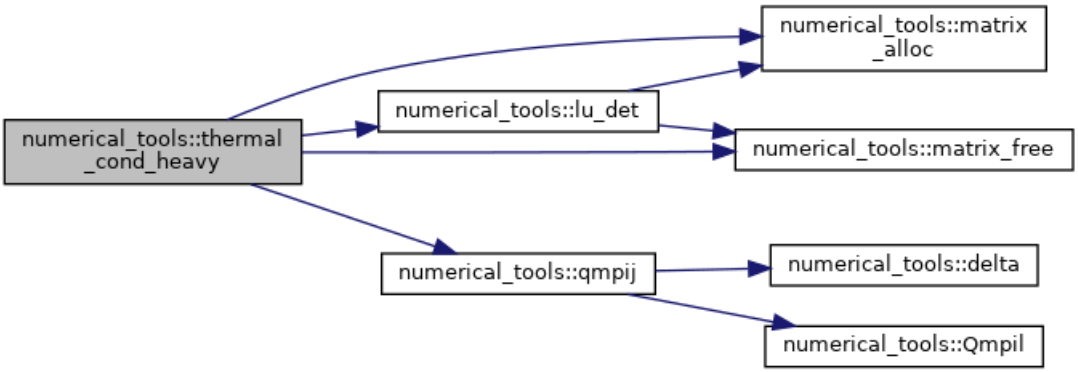
Here is the caller graph for this function:

double numerical_tools::thermal_cond_heavy (double ** *Qt*, double *T*, double * *n*, double * *mass*, int *N_SPC*) [inherited]

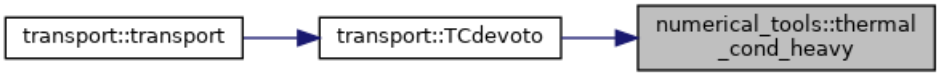
heavy thermal conductivity 2ND order approx

Definition at line 1519 of file numerical_tools.cpp.

Here is the call graph for this function:



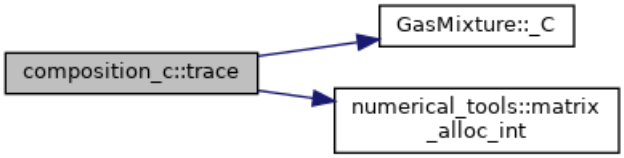
Here is the caller graph for this function:



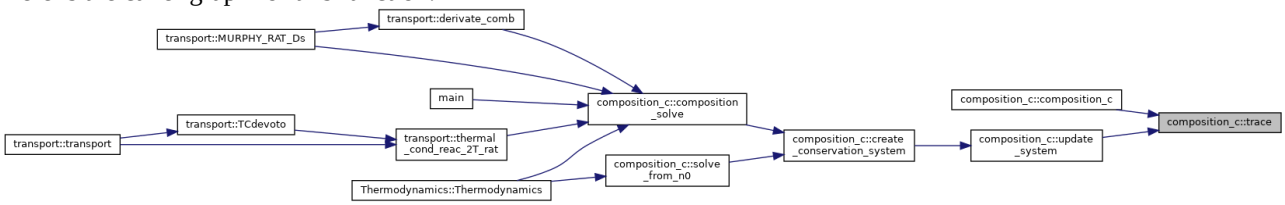
void composition_c::trace ()

Definition at line 493 of file composition_c.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

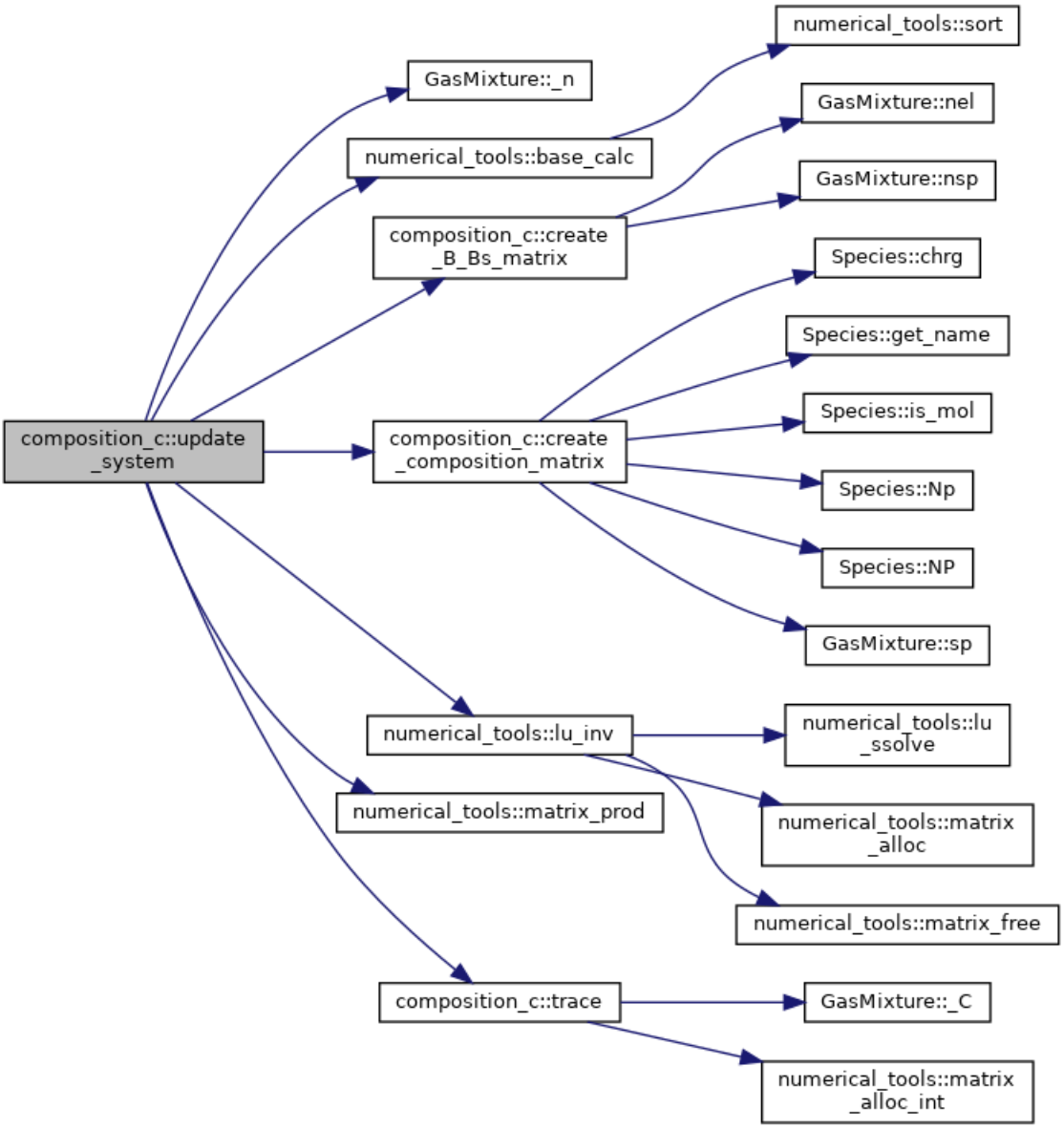


void composition_c::update_system ()

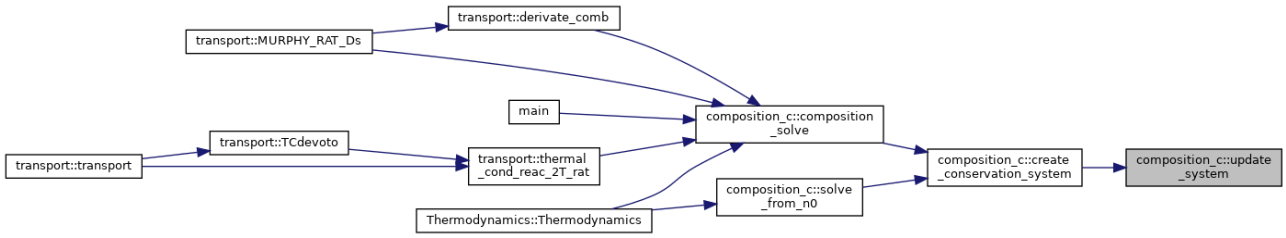
update GODIN problem matrices with new bases selected by n density vector if any

Definition at line 472 of file composition_c.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

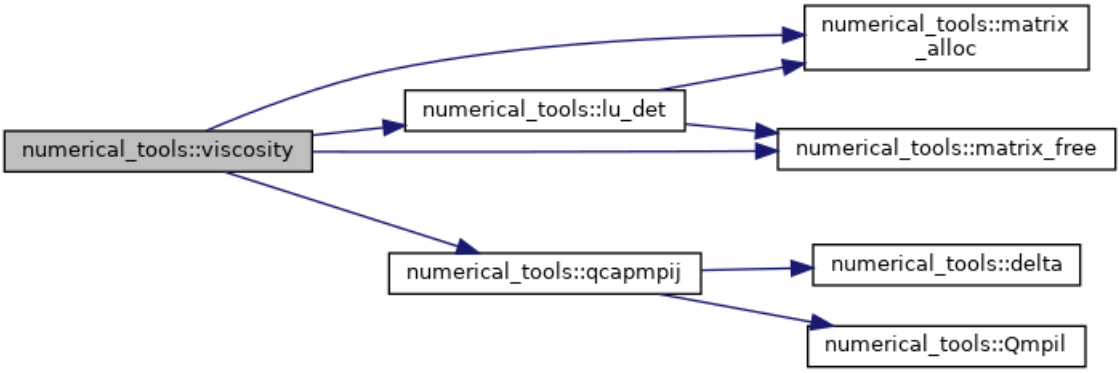


double numerical_tools::viscosity (double ** *Qt*, double *T*, double * *n*, double * *mass*, int *N_SPC*) `[inherited]`

funzione calcolo viscosita ///1Approx, to further (2nd available) modify M_ORDV

Definition at line 1645 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

Member Data Documentation

double** composition_c::A

linear system to solve

Definition at line 25 of file composition_c.h.

double* composition_c::A0

RHS A.

Definition at line 26 of file composition_c.h.

double** composition_c::B

matrix of base elements

Definition at line 22 of file composition_c.h.

int* composition_c::b

vector of base elements it takes the row of the i-th base in the composition matrix

Definition at line 28 of file composition_c.h.

double composition_c::Binv**

B^-1.
Definition at line 23 of file composition_c.h.

double composition_c::Bs**

matrix of derived by chemical reactions
Definition at line 24 of file composition_c.h.

int* composition_c::bs

vector of not-base elements
Definition at line 29 of file composition_c.h.

double composition_c::C**

composition matrix
Definition at line 21 of file composition_c.h.

GasMixture* composition_c::gas

Definition at line 12 of file composition_c.h.

int composition_c::L

N-M.
Definition at line 17 of file composition_c.h.

int composition_c::M

number of elements in the mixture
Definition at line 16 of file composition_c.h.

int composition_c::N

private pointer
number of species in the mixture
Definition at line 15 of file composition_c.h.

double composition_c::v**

v = Binv*B
Definition at line 27 of file composition_c.h.

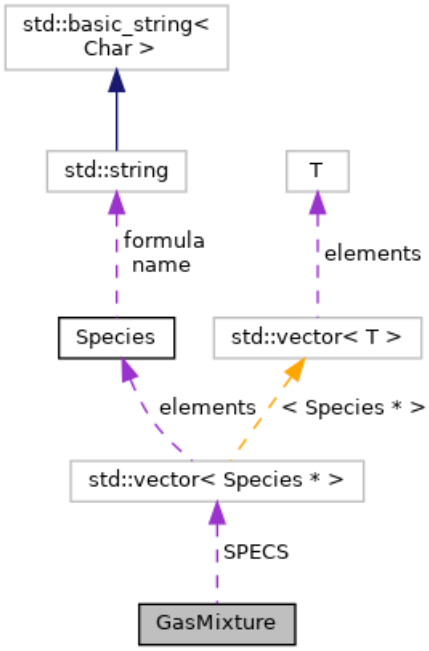
The documentation for this class was generated from the following files:

- gas/**composition_c.h**
- gas/**composition_c.cpp**

GasMixture Class Reference

`#include <gasMixture.h>`

Collaboration diagram for GasMixture:



Public Member Functions

- **Species sp** (int **x**)
  return the i-Species object

- void **_C** (int **_C_)
- int ** **__C** ()
- int **Cij** (int **i**, int j)
- double * **_c** ()
  return M-2 mole fraction relations

- void **set_ccflag** (bool TF)
- bool **_ccflag** ()
- void **set_xbflag** (bool TF, double dxb_step)
- double **_dxb** ()
- bool **_xbflag** ()
- void **set_xaflag** (bool TF, double dxa_step)
- double **_dxa** ()
- bool **_xaflag** ()
- void **_setlegend** (bool legenda)
- int **nsp** ()
  return number of species

- int **nel** ()
  return number of elements

- double **_p** ()

get pressure

- void **set_p** (double **x**)
  set pressure

- double **_T** ()
  get Temperature

- void **set_T** (double **x**)
  set Temperature

- double **_theta** ()
  get n.e. param

- void **set_theta** (double **x**)
  set n.e. param

- void **Qtot** ()
  compute Total partiction functions

- double **Qtot** (int **i**)
- **GasMixture** ()=default
  Default constructor.

- **GasMixture** (std::vector< **Species** > &species, double P, double T, double THETA)
  Constructor.

- **~GasMixture** ()=default
  Destructor.

- void **Update_Species** (std::vector< **Species** > &species)
  *update **Species** of the gas mixture*

- void **write_results** (std::string path)
  Write results to file.

- void **restore_n** ()
- double * **_n** ()
  return density as a pointer to n array

- double **_n** (int **i**)
  return i-th density

- double * **_td** ()
  return pointer to thermodynamics

- double **_td** (int **i**)

return i-th thermodynamics array

- double * **_x** ()
  return pointer to transport properties

- double **_x** (int **i**)
  return i-th transport property

- double **_rho** ()
  return gas_density, compute ***Thermodynamics(&gas)***

- double **_he** ()
  return electron specie enthalpy

- double **_hh** ()
  return heavy species enthalpy

- double **_ee** ()
  return electron internal energy

- double **_eh** ()
  return heavy internal energy

- double **_Cp** ()
  return pressure constant specific heat

- double **_Cv** ()
  return volume constant specific heat

- double **_gamma** ()
  return gamma=Cp/Cv

- double **_a** ()
  return speed of sound

Private Attributes

- std::vector< **Species** * > **SPECS**
  Vector of <Species>Objects.

- int **N**
  number of species

- int **M**
  number of elements

- double **pressure**
  pressure

- double **Temperature**
  Temperature.

- double **Theta**
  Non-equilibrium parameter.

- double * **Q**
  array of total partition functions

- double * **cstar**
  M-2 mole fraction relations.

- bool **ccflag** {false}
  Trigger cstar to be particle densities fraction.

- bool **xbflag** {false}
  Trigger mole fractions defined by MURPHY combined diffusion.

- bool **xaflag** {false}
- double **dxb** {0}
  dxb step activate composition calculations with set_xbflag(bool, dx)

- double **dxa** {0}
  dxa step activate composition calculations with set_xaflag(bool, dx)

- int ** **C_**
  composition matrix local data

- double * **n**
  output

- double * **td**
- double * **x**
- bool **legend** {true}

Detailed Description

Definition at line 21 of file gasMixture.h.

Constructor & Destructor Documentation

GasMixture::GasMixture () `[default]`

Default constructor.

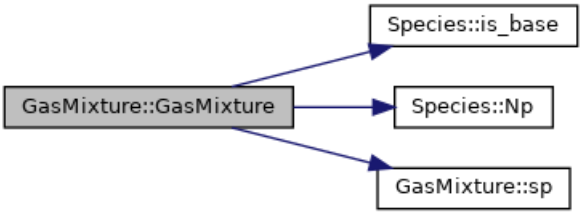
GasMixture::GasMixture (std::vector< Species > & *species*, double *P*, double *T*, double *THETA*)

Constructor.

WARNING: no composites mol as 1st mf_base

Definition at line 3 of file gasMixture.cpp.

Here is the call graph for this function:



GasMixture::~GasMixture () `[default]`

Destructor.

Member Function Documentation

int GasMixture::__C () `[inline]`**

get composition data needed to compute reactive conductivity

Definition at line 101 of file gasMixture.h.

double GasMixture::_a ()

return speed of sound

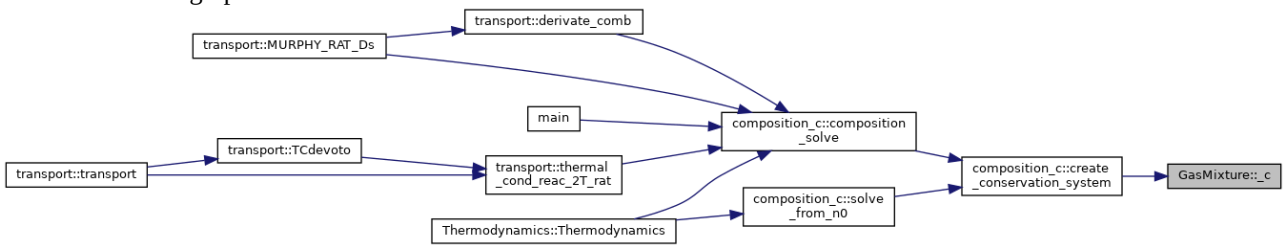
Definition at line 300 of file gasMixture.cpp.

double* GasMixture::_c () `[inline]`

return M-2 mole fraction relations

Definition at line 105 of file gasMixture.h.

Here is the caller graph for this function:

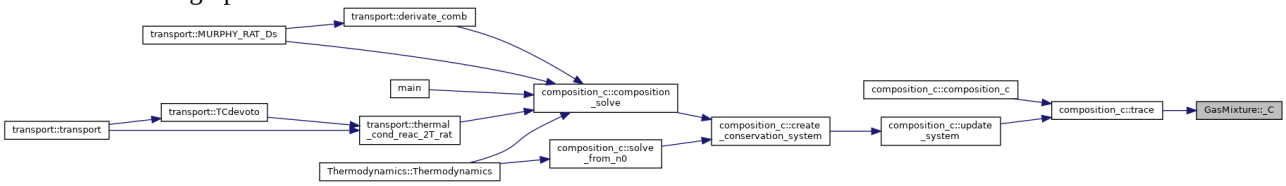


void GasMixture::_C (int ** _C_)[inline]

set composition data needed to compute reactive conductivity

Definition at line 98 of file gasMixture.h.

Here is the caller graph for this function:



bool GasMixture::_ccflag ()[inline]

Definition at line 107 of file gasMixture.h.

double GasMixture::_Cp ()

return pressure constant specific heat

Definition at line 304 of file gasMixture.cpp.

double GasMixture::_Cv ()

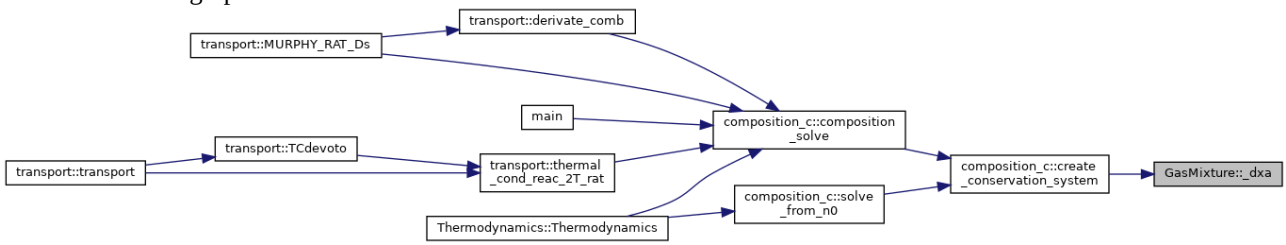
return volume constant specific heat

Definition at line 306 of file gasMixture.cpp.

double GasMixture::_dxa ()[inline]

Definition at line 112 of file gasMixture.h.

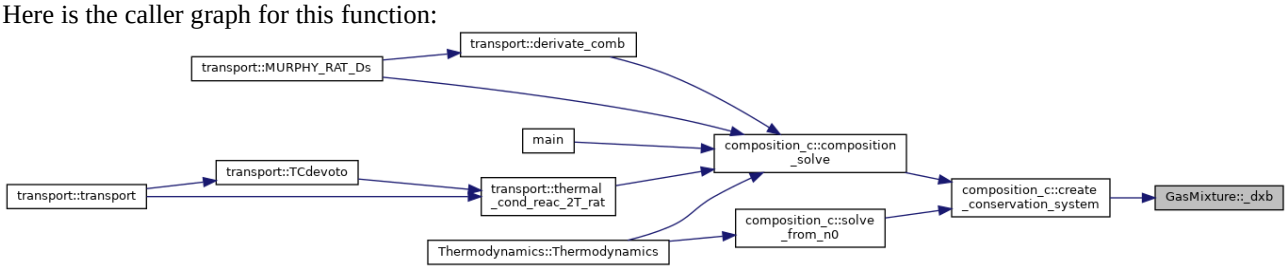
Here is the caller graph for this function:



double GasMixture::_dxb ()[inline]

Definition at line 109 of file gasMixture.h.

Here is the caller graph for this function:



double GasMixture::_ee ()

return electron internal energy

Definition at line 298 of file gasMixture.cpp.

double GasMixture::_eh ()

return heavy internal energy

Definition at line 302 of file gasMixture.cpp.

double GasMixture::_gamma ()

return gamma=Cp/Cv

Definition at line 308 of file gasMixture.cpp.

double GasMixture::_he ()

return electron specie enthalpy

Definition at line 294 of file gasMixture.cpp.

double GasMixture::_hh ()

return heavy species enthalpy

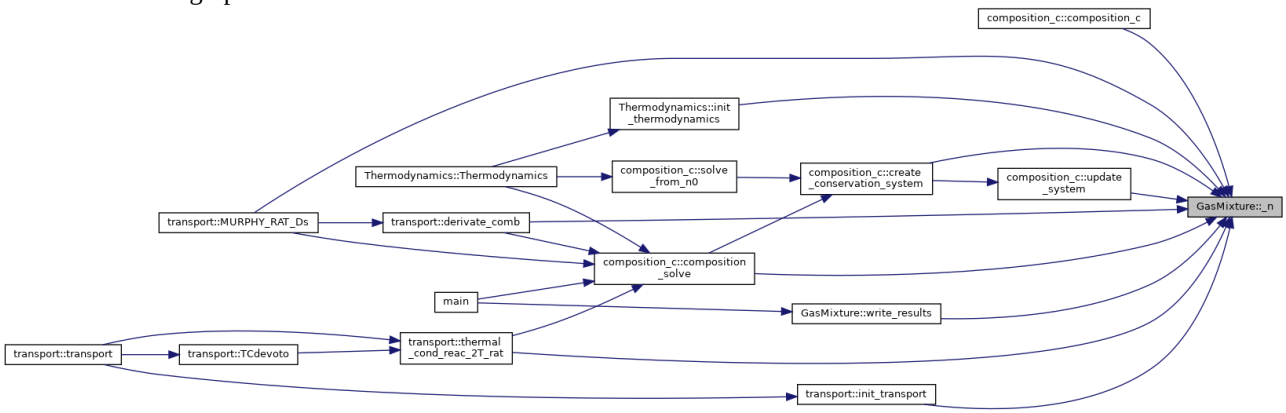
Definition at line 296 of file gasMixture.cpp.

double * GasMixture::_n ()

return density as a pointer to n array

Definition at line 277 of file gasMixture.cpp.

Here is the caller graph for this function:



double GasMixture::_n (int *i*)

return i-th density

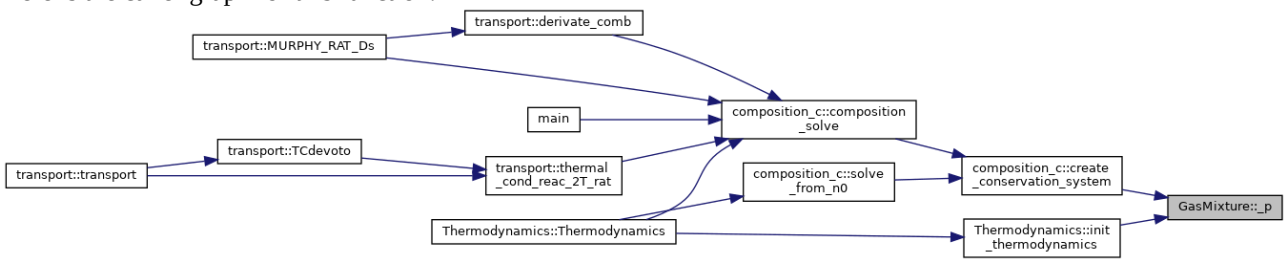
Definition at line 279 of file gasMixture.cpp.

double GasMixture::_p ()

get pressure

Definition at line 265 of file gasMixture.cpp.

Here is the caller graph for this function:



double GasMixture::_rho ()

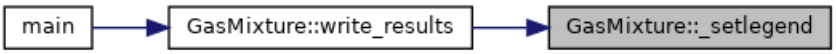
return gas_density, compute **Thermodynamics(&gas)**

Definition at line 292 of file gasMixture.cpp.

void GasMixture::_setlegend (bool *legenda*) `[inline]`

Definition at line 114 of file gasMixture.h.

Here is the caller graph for this function:

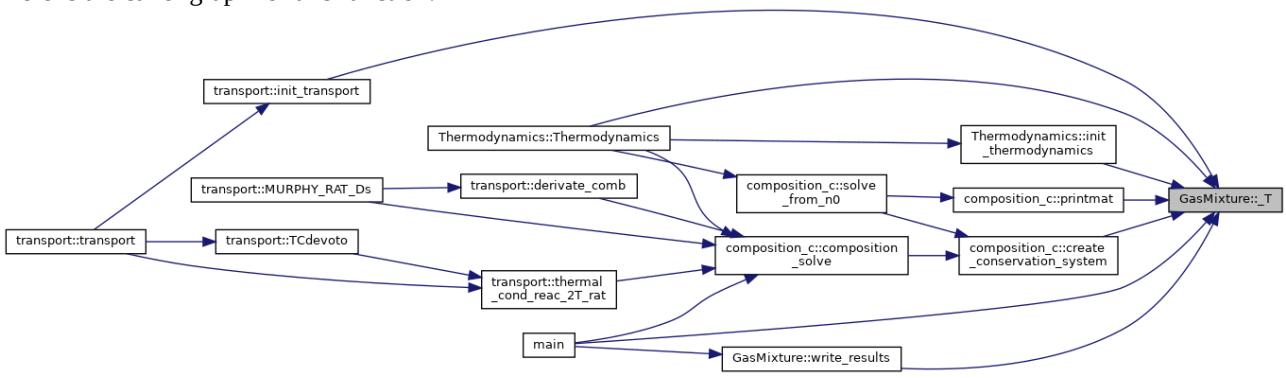


double GasMixture::_T ()

get Temperature

Definition at line 269 of file gasMixture.cpp.

Here is the caller graph for this function:

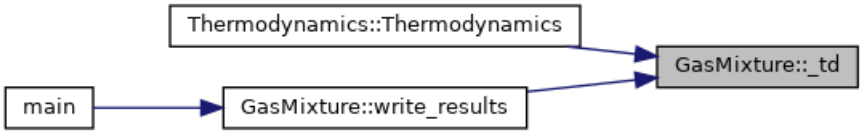


double * GasMixture::_td ()

return pointer to thermodynamics

Definition at line 282 of file gasMixture.cpp.

Here is the caller graph for this function:



double GasMixture::_td (int *i*)

return i-th thermodynamics array

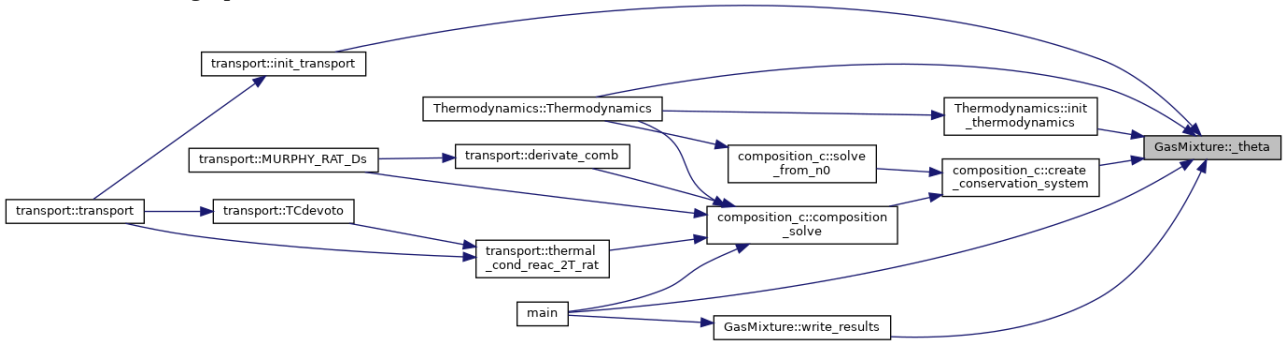
Definition at line 284 of file gasMixture.cpp.

double GasMixture::_theta ()

get n.e. param

Definition at line 273 of file gasMixture.cpp.

Here is the caller graph for this function:

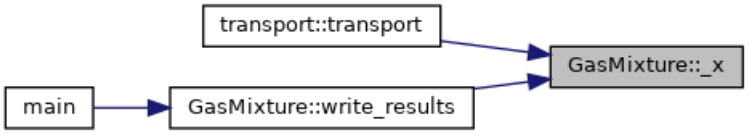


double * GasMixture::_x ()

return pointer to transport properties

Definition at line 286 of file gasMixture.cpp.

Here is the caller graph for this function:



double GasMixture::_x (int *i*)

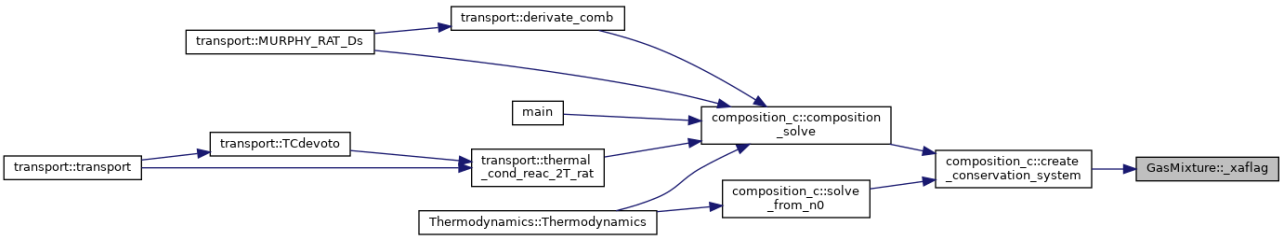
return i-th transport property

Definition at line 288 of file gasMixture.cpp.

bool GasMixture::_xaflag () `[inline]`

Definition at line 113 of file gasMixture.h.

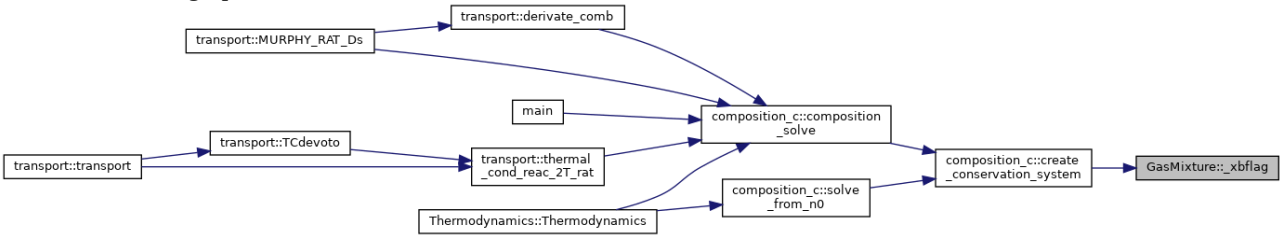
Here is the caller graph for this function:



bool GasMixture::_xbflag () `[inline]`

Definition at line 110 of file gasMixture.h.

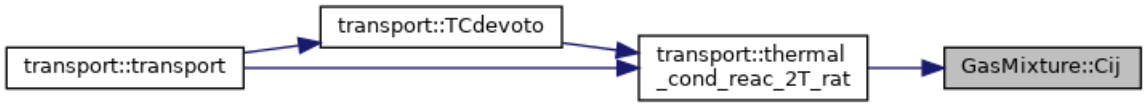
Here is the caller graph for this function:



int GasMixture::Cij (int *i*, int *j*) `[inline]`

Definition at line 102 of file gasMixture.h.

Here is the caller graph for this function:

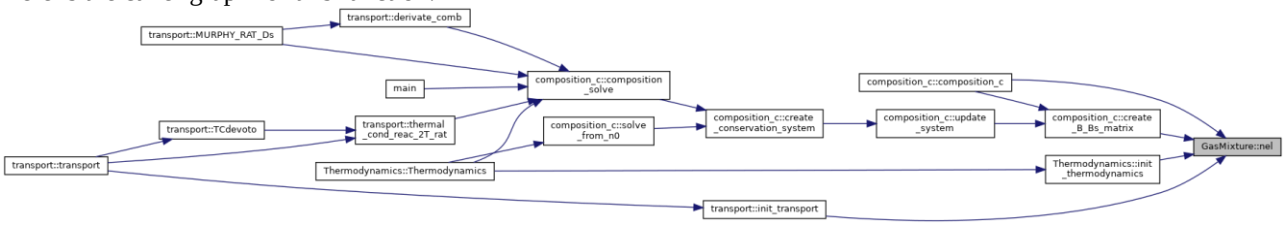


int GasMixture::nel ()

return number of elements

Definition at line 262 of file gasMixture.cpp.

Here is the caller graph for this function:

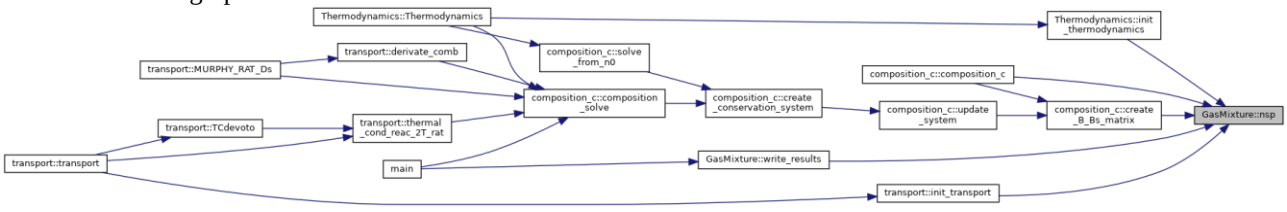


int GasMixture::nsp ()

return number of species

Definition at line 259 of file gasMixture.cpp.

Here is the caller graph for this function:

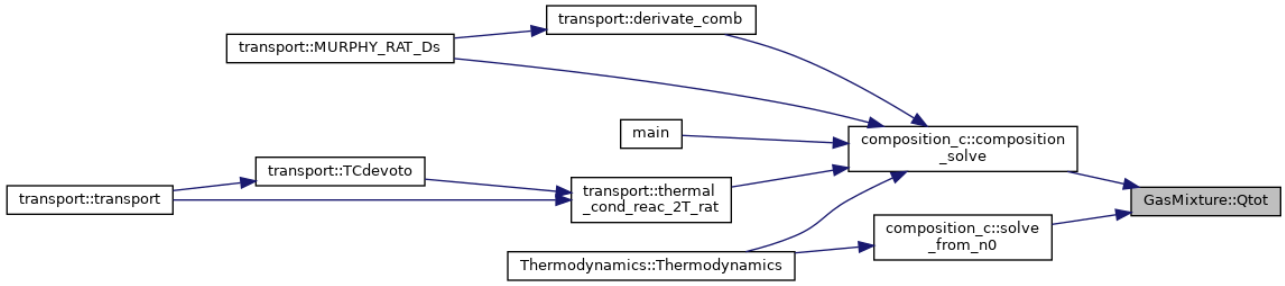


void GasMixture::Qtot ()

compute Total partiction functions

Definition at line 128 of file gasMixture.cpp.

Here is the caller graph for this function:



double GasMixture::Qtot (int *i*)

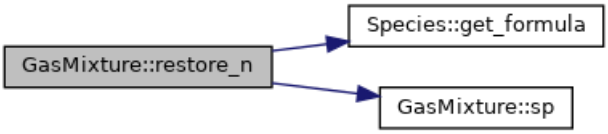
get total partiction functions to be computed with void **Qtot()**>

Definition at line 253 of file gasMixture.cpp.

void GasMixture::restore_n ()

Definition at line 355 of file gasMixture.cpp.

Here is the call graph for this function:

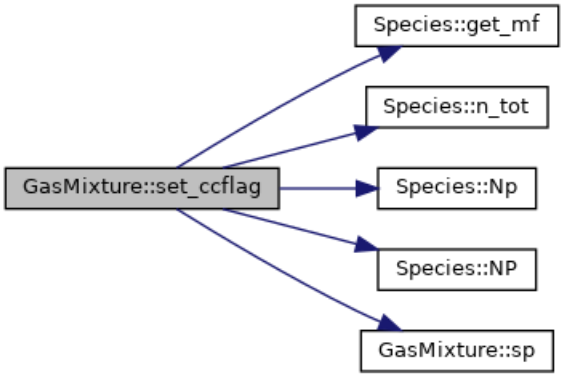


void GasMixture::set_ccflag (bool *TF*)

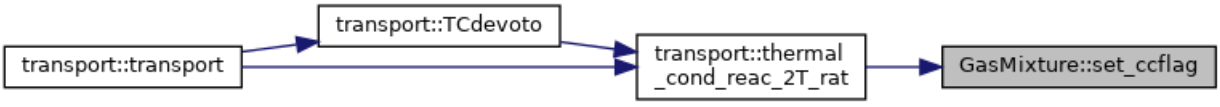
WARNING : Do not set composite moleculae as mf_base

Definition at line 156 of file gasMixture.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



void GasMixture::set_p (double *x*)

set pressure

Definition at line 266 of file gasMixture.cpp.

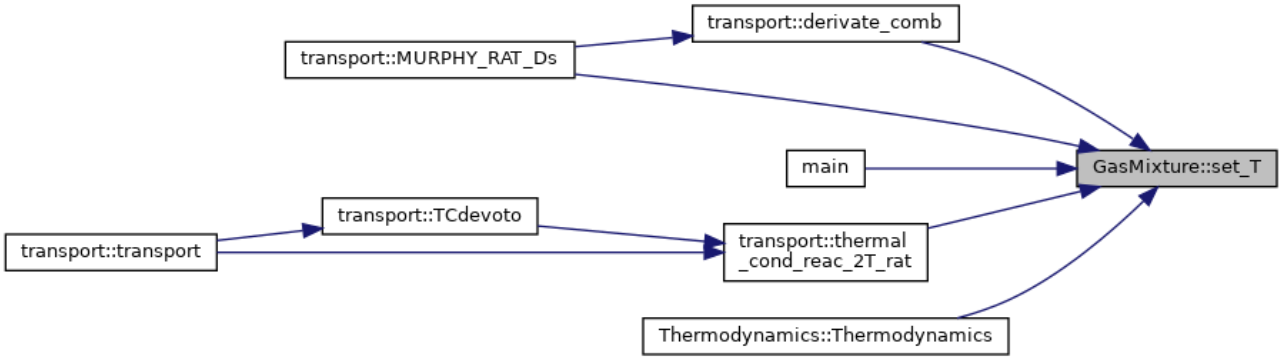
Here is the caller graph for this function:

void GasMixture::set_T (double *x*)

set Temperature

Definition at line 270 of file gasMixture.cpp.

Here is the caller graph for this function:



void GasMixture::set_theta (double *x*)

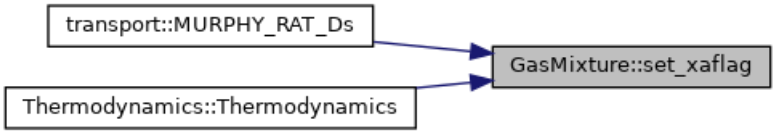
set n.e. param

Definition at line 274 of file gasMixture.cpp.

void GasMixture::set_xaflag (bool *TF*, double *dxa_step*) `[inline]`

Definition at line 111 of file gasMixture.h.

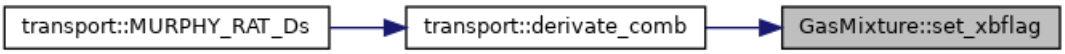
Here is the caller graph for this function:



void GasMixture::set_xbflag (bool *TF*, double *dxb_step*) `[inline]`

Definition at line 108 of file gasMixture.h.

Here is the caller graph for this function:

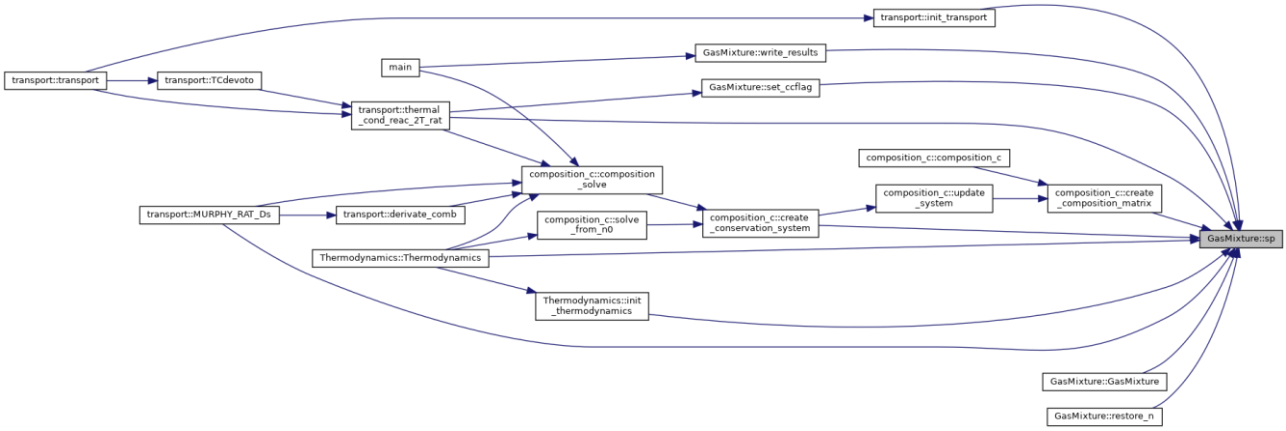


Species GasMixture::sp (int *x*)

return the i-Species object

Definition at line 256 of file gasMixture.cpp.

Here is the caller graph for this function:



void GasMixture::Update_Species (std::vector< Species > & *species*)

update **Species** of the gas mixture

Definition at line 92 of file gasMixture.cpp.

void GasMixture::write_results (std::string *path*)

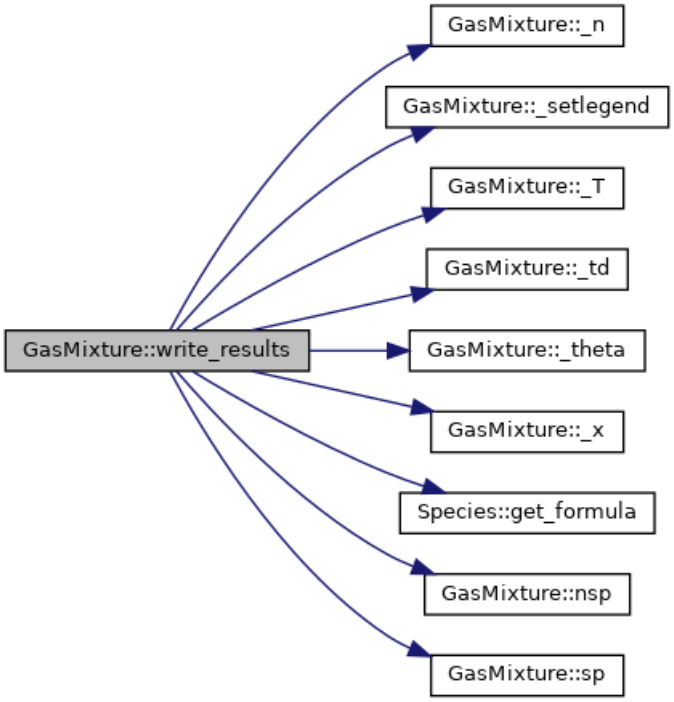
Write results to file.

write file legend

write data

Definition at line 311 of file gasMixture.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

Member Data Documentation

int** GasMixture::C_ [private]

composition matrix local data

Definition at line 47 of file gasMixture.h.

bool GasMixture::ccflag {false} [private]

Trigger cstar to be particle densities fraction.

Definition at line 41 of file gasMixture.h.

double* GasMixture::cstar [private]

M-2 mole fraction relations.

Definition at line 40 of file gasMixture.h.

double GasMixture::dxa {0} [private]

dxa step activate composition calculations with set_xaflag(bool, dx)

Definition at line 45 of file gasMixture.h.

double GasMixture::dxb {0} [private]

dxb step activate composition calculations with set_xbflag(bool, dx)

Definition at line 44 of file gasMixture.h.

bool GasMixture::legend {true} [private]

transport legend: x[0] "ke" electron thermal conducivity x[1] "kh" heavy thermal conducivity

x[2] "kre" electron reactive conductivity x[3] "krh" heavy reactive conductivity x[4] "mu" viscosity x[5] "sigma" electrical conductivity x[6] "Qeh" thermal exchange x[7] "DxAB" ordinary combined MURPHY diffusion coefficient

x[8] "DTAB" combined thermal diffusion coefficient x[9] "DYAB" diffusion coefficient

x[10] "DTABy" ordinary combined diffusion coefficient FLUENT

x[11] "DTBAy" thermal combined diffusion coefficient FLUENT

Definition at line 87 of file gasMixture.h.

int GasMixture::M [private]

number of elements

Definition at line 30 of file gasMixture.h.

int GasMixture::N `[private]`

number of species

Definition at line 28 of file gasMixture.h.

double* GasMixture::n `[private]`

output

particle density starting n0 by **GasMixture** constructor, computed with **composition_c(&gas)**.composition_solve()

Definition at line 53 of file gasMixture.h.

double GasMixture::pressure `[private]`

pressure

Definition at line 32 of file gasMixture.h.

double* GasMixture::Q `[private]`

array of total partition functions

Definition at line 38 of file gasMixture.h.

std::vector<Species*> GasMixture::SPECS `[private]`

Vector of <Species>Objects.

Definition at line 26 of file gasMixture.h.

double* GasMixture::td `[private]`

thermodynamics data computed with **Thermodynamics(&gas)**

Definition at line 57 of file gasMixture.h.

double GasMixture::Temperature `[private]`

Temperature.

Definition at line 34 of file gasMixture.h.

double GasMixture::Theta `[private]`

Non-equilibrium parameter.

Definition at line 36 of file gasMixture.h.

double* GasMixture::x `[private]`

thermodynamics legend: TD[0] "rho" density TD[1] "he" electron entalphy TD[2] "hh" heavy entalphy TD[3] "ee" electron internal energy TD[4] "eh" heavy internal energy TD[5] "Cp" constant pressure specific heat TD[6] "Cv" constant volume specific heat

TD[7] "gamma" Cp/Cv TD[8] "a" Speed of sound */ Transport data computed with **transport(&gas)**.TCdevoto()

Definition at line 72 of file gasMixture.h.

bool GasMixture::xaflag {false} `[private]`

Definition at line 43 of file gasMixture.h.

bool GasMixture::xbflag {false} `[private]`

Trigger mole fractions defined by MURPHY combined diffusion.

Definition at line 42 of file gasMixture.h.

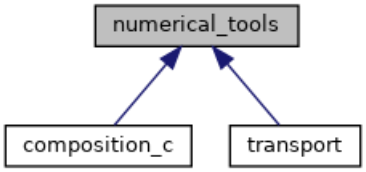
The documentation for this class was generated from the following files:

- gas/**gasMixture.h**
- gas/**gasMixture.cpp**

numerical_tools Class Reference

`#include <numerical_tools.h>`

Inheritance diagram for numerical_tools:



Public Member Functions

- bool **foo** (int **i**, int j, double N, double M)
- void **sort** (double *arr, int *perm, int n)
- void **matrix_prod** (double **A, double **B, double **C, int N1, int N2)
- double ** **matrix_alloc** (int N, int M)
  *initialize size of **A objects*

- int ** **matrix_alloc_int** (int n, int m)
  same as matrix_alloc but int variables

- void **matrix_free** (double **matrix, int n)
  clears a matrix

- void **lu_ssolve** (double *x, int *p, double *b0, double **m, int N)
  solve lu decomposition of a matrix

- void **lu_inv** (double **AINV, double **A, int N)
  use lu algorithm to compute the inverse matrix

- void **base_calc** (double *n, int *b, int *bs, double **C, int N, int M)
  track C rows to set basis & not-basis species based on n density vestor

- void **residual** (double *R, double **J, double *n, double **A, double *A0, double **v, int *b, int *bs, int N, int M)
  compute residuals as in GODIN eq.35

- void **lu_sistema** (double *x, double **A, double *b, int N)
  solve an LU system

- double **lu_det** (double **A, int N)
  *compute det(**A) with LU scomposition*

- double **max_double** (double *arr, int N)
  return maximum element of an array

- void **read_Q** (double ***Q, int N_NC, int N_C, int N_TEMP, std::vector< std::string > _filenames)

reads cross sections data / DA GENERALIZZARE */*

- void **read_Q_coulomb** (double **Qc, std::string path)
  reads Coullomb cross sections data / DA GENERALIZZARE */*

- void **il_calc** (int ***i**, int *l, int il, int N_specs)
- int **Qij_calc** (int il, int *Z, int N_specs)
- double **Q_gen** (double TT, double T, double theta, double *n, int *Z, int mp, int **i**, int j, int N_SPC)
  generates firsts collision integrals

- double **Q_coulomb** (double **Qc, double TT, double T, double theta, double *n, int *Z, int mp, int **i**, int j, int N_SPC)
  generates Coulomb collisions integral

- double **interp** (double *x, double *y, double xx, int N_xy)
  interpolator

- double **Diff_bin_rat** (double **Qt, double TT, double T, double theta, double *mass, double nn, int **i**, int j, int N_SPC)
  Binary diffusion as RAT et. al. 30Apr2002.

- double **Qmpil** (double **Qt, int m, int p, int **i**, int l, int N_SPC)
  extract Qt element for every couple interaction i l

- double **Fij_calc** (double **Dbin, double *n, double *mass, double rho, int **i**, int j, int N_SPC)
  Colombo,Ghedini,Sanibondi UNIBO 20/01/2009 eq.17.

- double **Fij_cofactor** (double **Fij, int **i**, int j, int N_SPC)
- int **delta** (int **i**, int j)
  Kronecker delta.

- double **delta_double** (int **i**, int j)
  Kronecker delta.

- double **qsimpmpij** (double **Qt, double *n, int N_SPC, int m, int p)
  simplified Devoto 08/08/1966 Appendix, assumed electron as last specie

- double **qmpij** (double **Qt, double *n, double *mass, int N_SPC, int m, int p, int **i**, int j)
  DEVOTO Appendix 06/1966.

- double **DiffT** (double **Qt, double T, double theta, double *n, double *mass, int N_SPC, int ii)
- double **Diff_amb** (double **D, double T, double theta, double *mass, double *n, int *Z, int ii, int jj, int N_SPC)
  ambipolar diffusion and thermal diffusion

- double **Diff_T_amb** (double **D, double *DT, double T, double theta, double *mass, double *n, int *Z, int ii, int N_SPC)
- double **thermal_cond_el** (double **Qt, double Te, double *n, double *mass, int N_SPC)

electron thermal conductivity simplified DEVOTO eq.21

- double **thermal_cond_heavy** (double **Qt, double T, double *n, double *mass, int N_SPC)
  heavy thermal conductivity 2ND order approx

- double **viscosity** (double **Qt, double T, double *n, double *mass, int N_SPC)
  funzione calcolo viscosita ///1Approx, to further (2nd available) modify M_ORDV

- double **qcapmpij** (double **Qt, double *n, double *mass, int N_SPC, int m, int p, int **i**, int j)
  elementi per calcolo bracket integrals da teoria DEVOTO per viscosità

- double **el_cond** (double **Qt, double T, double theta, double *n, double *mass, int N_SPC)
  DEVOTO simplified electrical conductivity eq.16.

- double **Qeh** (double **Qt, double T, double theta, double *mass, double *n, int N_SPC)
  thermal exchange electron-heavy

- double **Dx_AB_comp** (double **Da, double theta, double *mass, double *n, double *DxDxB, double *b, int pp, int N_SPC)
  MURPHY-RAT ordinary diffusion coefficients eq. 2.36.

- double **DYAB_comb** (double DxxAB, double *nf, double *nb, double dya, double *n, double rho, double *mass, int *Z, int p, int N_SPC)
  UNIBO 2008 diffusion coefficients.

- double **DT_AB_comp** (double **Da, double *DT, double Temp, double theta, double rho, double *mass, double *n, double *b, double *DxxDT, int p, int N_SPC)
- double **DtAB_comb_y** (double T, double dT, double DtAB, double Dxab, double *n_f, double *n_b, double rho, double *n, double *mass, int *_Z, int p, int N_SPC)

Detailed Description

Definition at line 12 of file numerical_tools.h.

Member Function Documentation

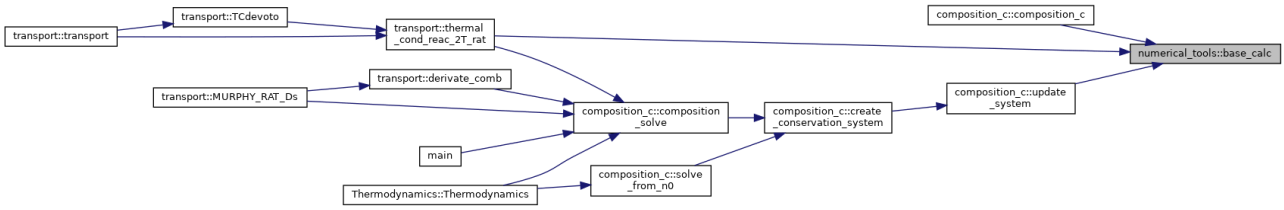
void numerical_tools::base_calc (double * *n*, int * *b*, int * *bs*, double ** *C*, int *N*, int *M*)

track C rows to set basis & not-basis species based on n density vestor

Definition at line 230 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

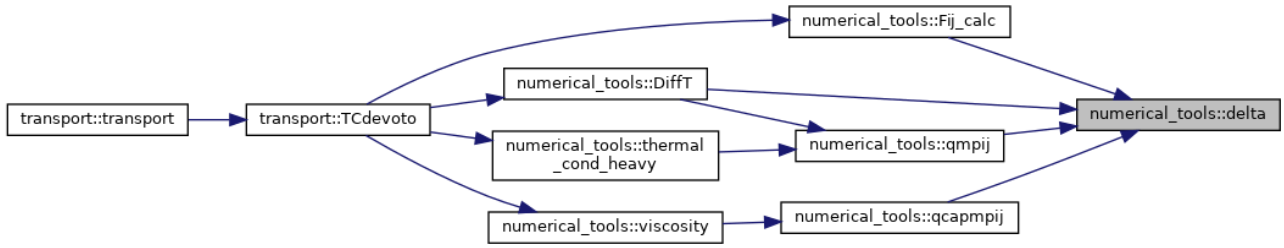


int numerical_tools::delta (int *i*, int *j*)

Kronecker delta.

Definition at line 886 of file numerical_tools.cpp.

Here is the caller graph for this function:



double numerical_tools::delta_double (int *i*, int *j*)

Kronecker delta.

Definition at line 892 of file numerical_tools.cpp.

double numerical_tools::Diff_amb (double ** *D*, double *T*, double *theta*, double * *mass*, double * *n*, int * *Z*, int *ii*, int *jj*, int *N_SPC*)

ambipolar diffusion and thermal diffusion

Definition at line 1417 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::Diff_bin_rat (double ** *Qt*, double *TT*, double *T*, double *theta*, double * *mass*, double *nn*, int *i*, int *j*, int *N_SPC*)

Binary diffusion as RAT et. al. 30Apr2002.

Definition at line 779 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

double numerical_tools::Diff_T_amb (double ** *D*, double * *DT*, double *T*, double *theta*, double * *mass*, double * *n*, int * *Z*, int *ii*, int *N_SPC*)

Definition at line 1455 of file numerical_tools.cpp.

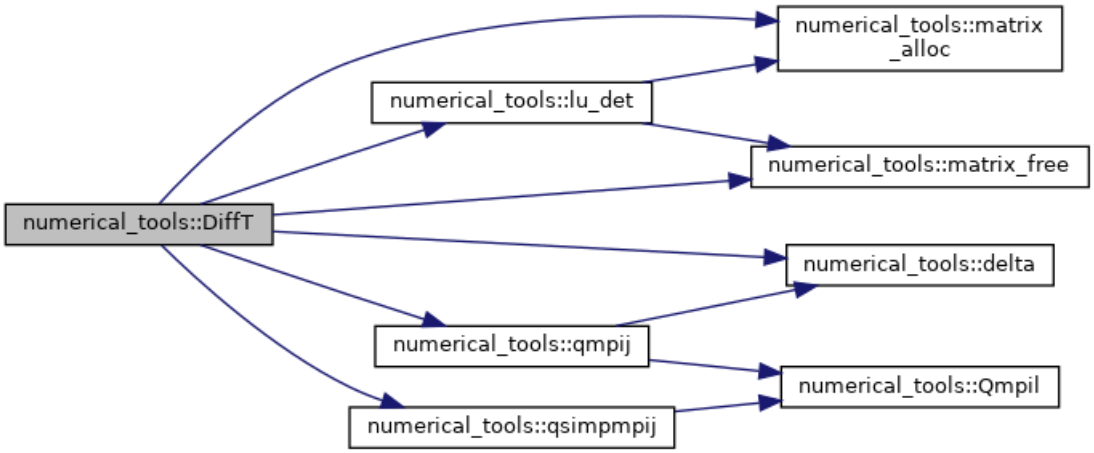
Here is the caller graph for this function:

double numerical_tools::DiffT (double ** *Qt*, double *T*, double *theta*, double * *n*, double * *mass*, int *N_SPC*, int *ii*)

Temperature Diffusion computed with Devoto 4th approx for electrons and 2nd for heavy particles, modify N_ORD_H for further approx

Definition at line 1342 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

double numerical_tools::DT_AB_comp (double ** *Da*, double * *DT*, double *Temp*, double *theta*, double *rho*, double * *mass*, double * *n*, double * *b*, double * *DxxDT*, int *p*, int *N_SPC*)

Definition at line 1886 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::DtAB_comb_y (double *T*, double *dT*, double *DtAB*, double *Dxab*, double * *n_f*, double * *n_b*, double *rho*, double * *n*, double * *mass*, int * *_Z*, int *p*, int *N_SPC*)

Definition at line 1955 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::Dx_AB_comp (double ** *Da*, double *theta*, double * *mass*, double * *n*, double * *DxDxB*, double * *b*, int *pp*, int *N_SPC*)

MURPHY-RAT ordinary diffusion coefficients eq. 2.36.

Definition at line 1746 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::DYAB_comb (double *DxxAB*, double * *nf*, double * *nb*, double *dya*, double * *n*, double *rho*, double * *mass*, int * *Z*, int *p*, int *N_SPC*)

UNIBO 2008 diffusion coefficients.

Definition at line 1826 of file numerical_tools.cpp.

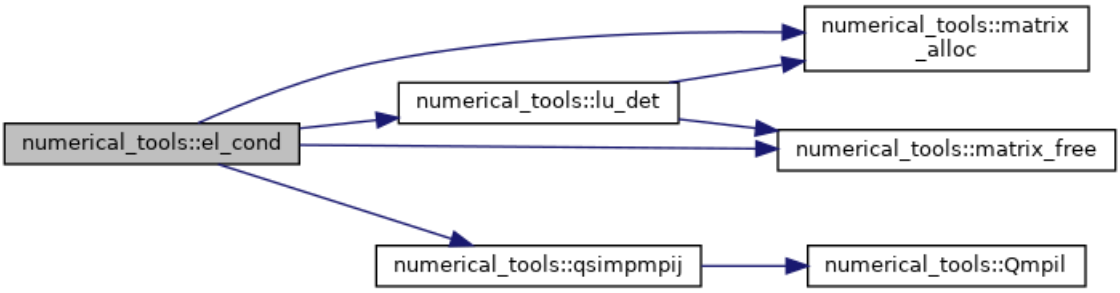
Here is the caller graph for this function:

double numerical_tools::el_cond (double ** *Qt*, double *T*, double *theta*, double * *n*, double * *mass*, int *N_SPC*)

DEVOTO simplified electrical conductivity eq.16.

Definition at line 1688 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

double numerical_tools::Fij_calc (double ** *Dbin*, double * *n*, double * *mass*, double *rho*, int *i*, int *j*, int *N_SPC*)

Colombo,Ghedini,Sanibondi UNIBO 20/01/2009 eq.17.

Definition at line 842 of file numerical_tools.cpp.

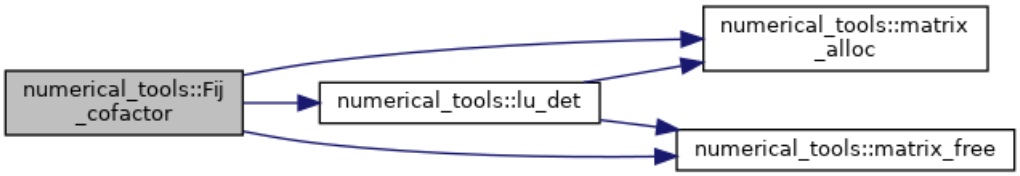
Here is the call graph for this function:

Here is the caller graph for this function:

double numerical_tools::Fij_cofactor (double ** *Fij*, int *i*, int *j*, int *N_SPC*)

Definition at line 856 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

bool numerical_tools::foo (int *i*, int *j*, double *N*, double *M*)

void numerical_tools::il_calc (int * *i*, int * *I*, int *il*, int *N_specs*)

Associates species couple indexes to binary interaction progressively numerated (il)

Definition at line 502 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::interp (double * *x*, double * *y*, double *xx*, int *N_xy*)

interpolator

Definition at line 742 of file numerical_tools.cpp.

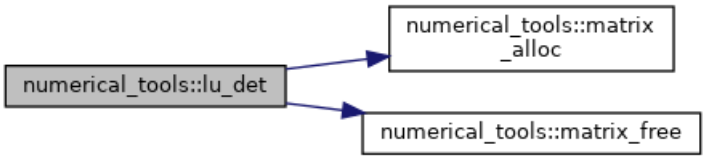
Here is the caller graph for this function:

double numerical_tools::lu_det (double ** *A*, int *N*)

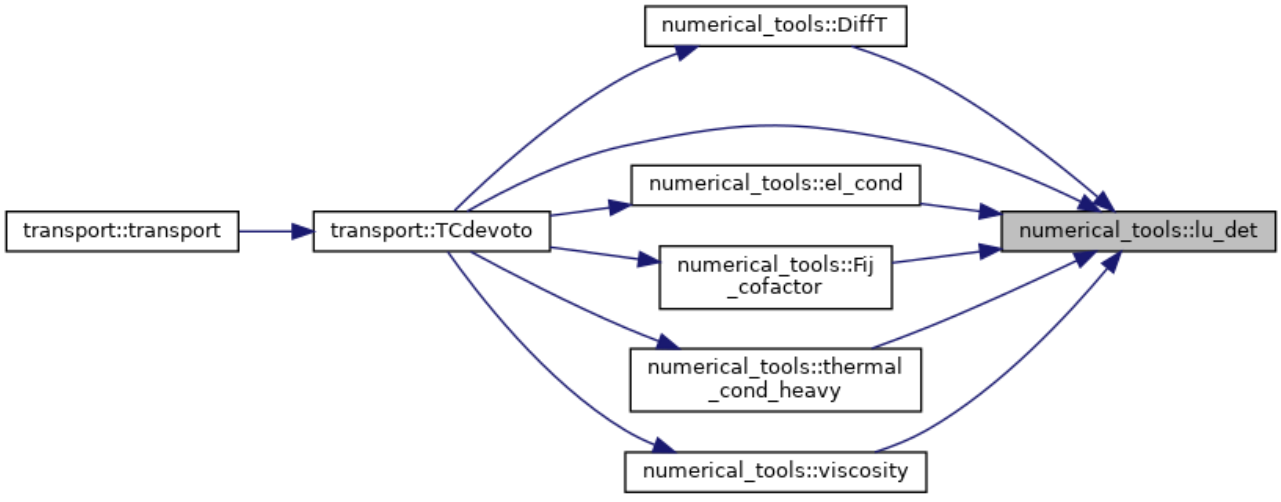
compute det(**A) with LU scomposition

Definition at line 372 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

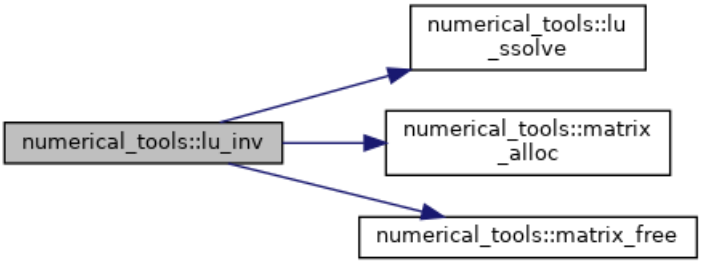


void numerical_tools::lu_inv (double ** *AINV*, double ** *A*, int *N*)

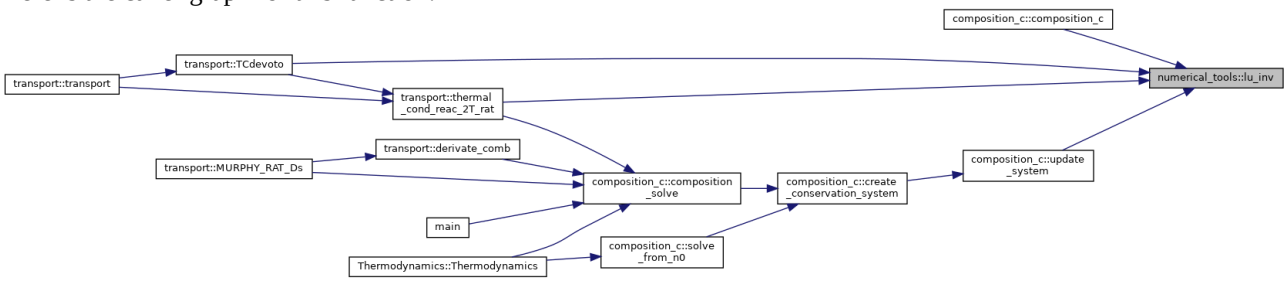
use lu algorithm to compute the inverse matrix

Definition at line 167 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

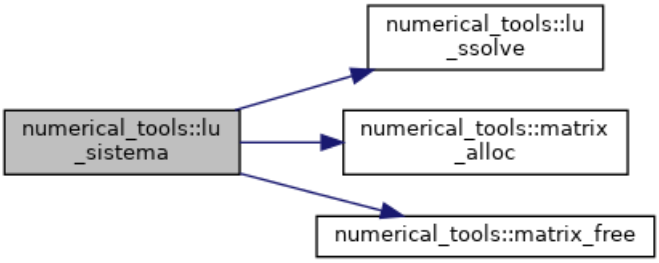


void numerical_tools::lu_sistema (double * *x*, double ** *A*, double * *b*, int *N*)

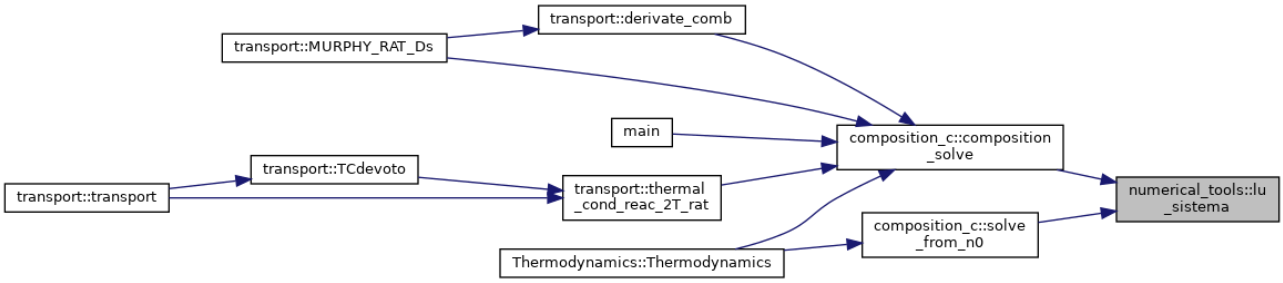
solve an LU system

Definition at line 321 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

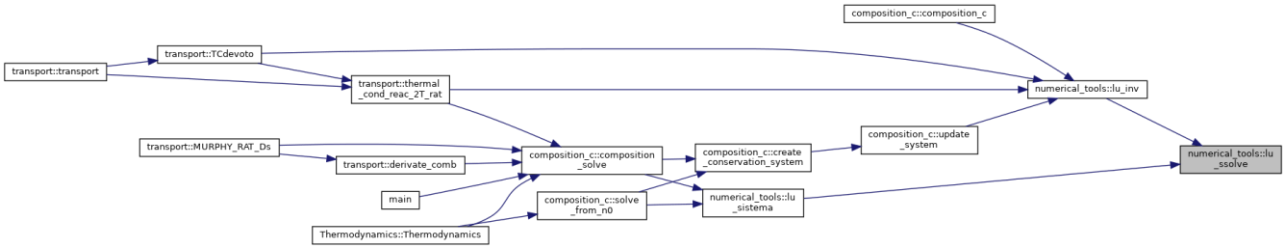


void numerical_tools::lu_ssolve (double * *x*, int * *p*, double * *b0*, double ** *m*, int *N*)

solve lu decomposition of a matrix

Definition at line 122 of file numerical_tools.cpp.

Here is the caller graph for this function:

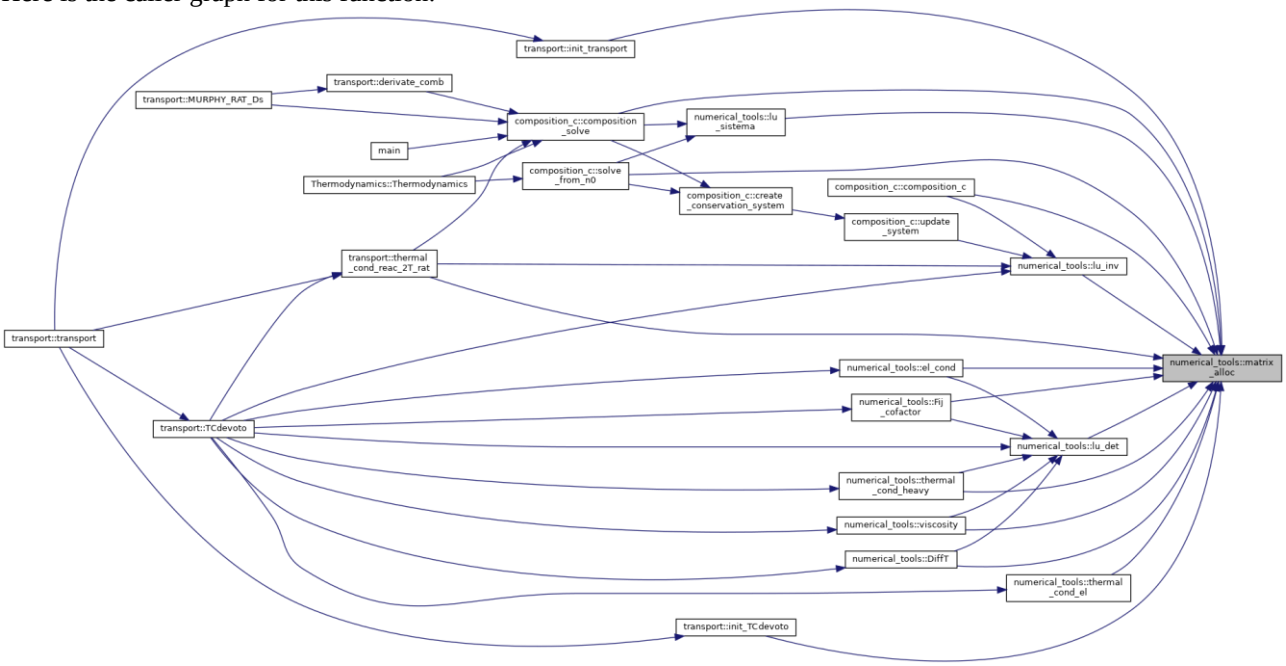


double ** numerical_tools::matrix_alloc (int *N*, int *M*)

initialize size of **A objects

Definition at line 84 of file numerical_tools.cpp.

Here is the caller graph for this function:

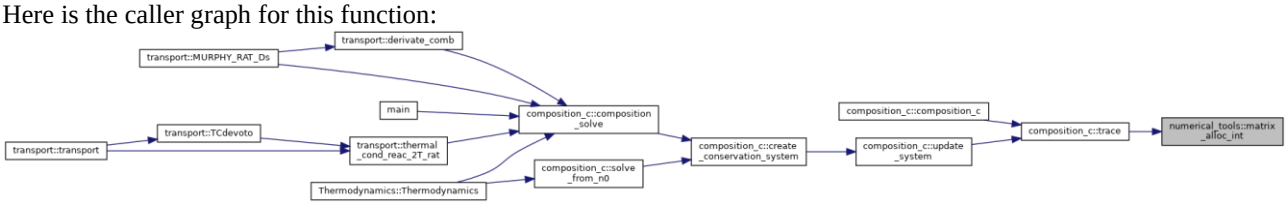


int ** numerical_tools::matrix_alloc_int (int *n*, int *m*)

same as matrix_alloc but int variables

Definition at line 97 of file numerical_tools.cpp.

Here is the caller graph for this function:

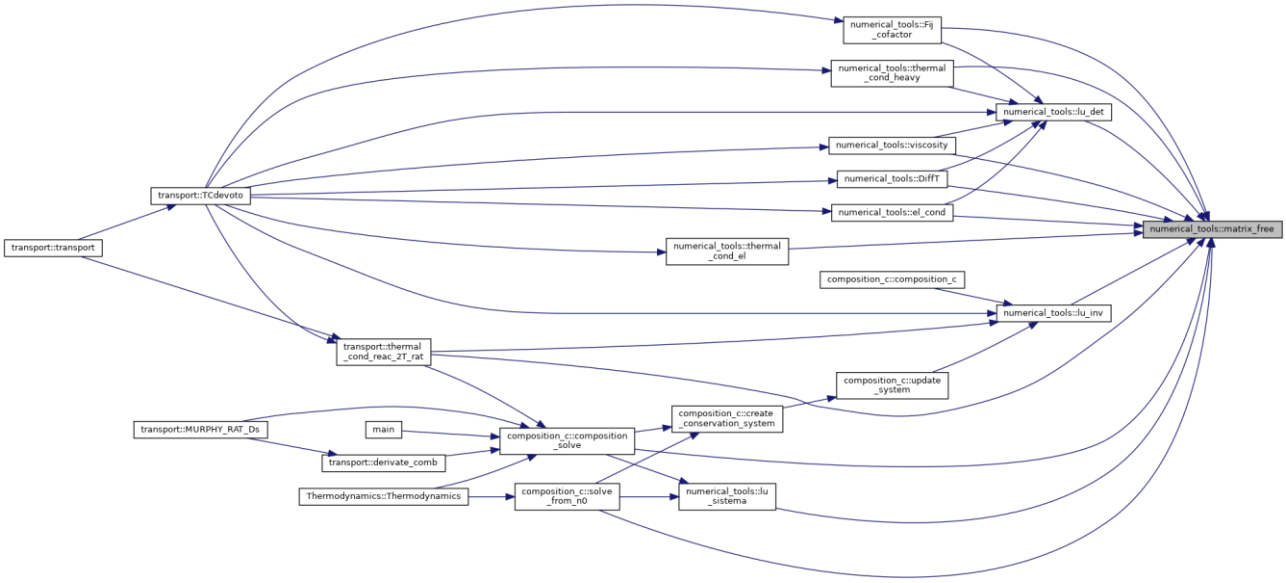


void numerical_tools::matrix_free (double ** *matrix*, int *n*)

clears a matrix

Definition at line 111 of file numerical_tools.cpp.

Here is the caller graph for this function:

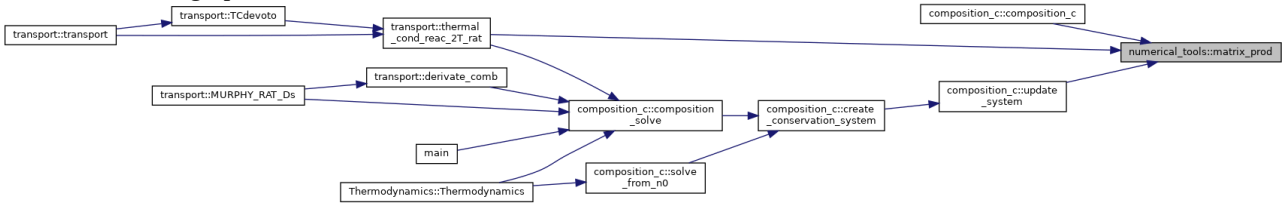


void numerical_tools::matrix_prod (double ** *A*, double ** *B*, double ** *C*, int *N1*, int *N2*)

compute matrix product A = B*C

Definition at line 63 of file numerical_tools.cpp.

Here is the caller graph for this function:

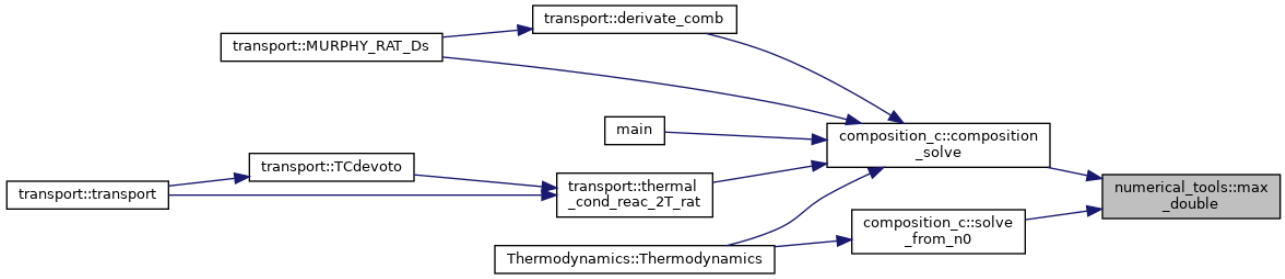


double numerical_tools::max_double (double * *arr*, int *N*)

return maximum element of an array

Definition at line 432 of file numerical_tools.cpp.

Here is the caller graph for this function:



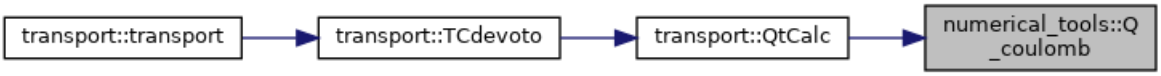
double numerical_tools::Q_coulomb (double ** *Qc*, double *TT*, double *T*, double *theta*, double * *n*, int * *Z*, int *mp*, int *i*, int *j*, int *N_SPC*)

generates Coulomb collisions integral

Definition at line 683 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:



double numerical_tools::Q_gen (double *TT*, double *T*, double *theta*, double * *n*, int * *Z*, int *mp*, int *i*, int *j*, int *N_SPC*)

generates firsts collision integrals

Definition at line 553 of file numerical_tools.cpp.

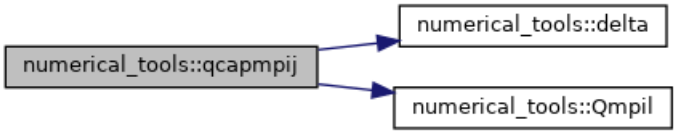
Here is the caller graph for this function:

double numerical_tools::qcapmpij (double ** *Qt*, double * *n*, double * *mass*, int *N_SPC*, int *m*, int *p*, int *i*, int *j*)

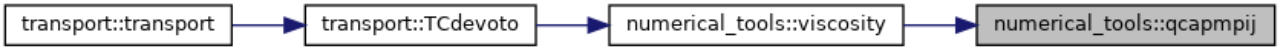
elementi per calcolo bracket integrals da teoria DEVOTO per viscosità

Definition at line 1580 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



double numerical_tools::Qeh (double ** *Qt*, double *T*, double *theta*, double * *mass*, double * *n*, int *N_SPC*)

thermal exchange electron-heavy

Definition at line 1729 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

int numerical_tools::Qij_calc (int *il*, int * *Z*, int *N_specs*)

associates to il the Qij coulomb interaction progressively numerated when Z[i]*Z[j]==0

Definition at line 522 of file numerical_tools.cpp.

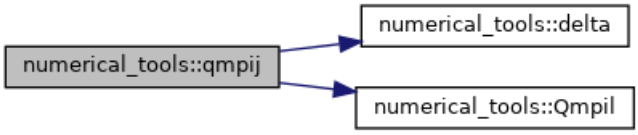
Here is the caller graph for this function:

double numerical_tools::qmpij (double ** *Qt*, double * *n*, double * *mass*, int *N_SPC*, int *m*, int *p*, int *i*, int *j*)

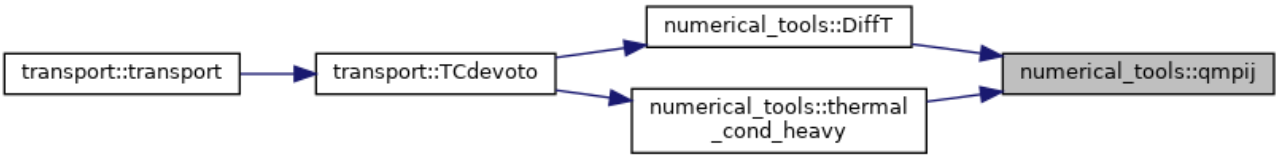
DEVOTO Appendix 06/1966.

Definition at line 1063 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

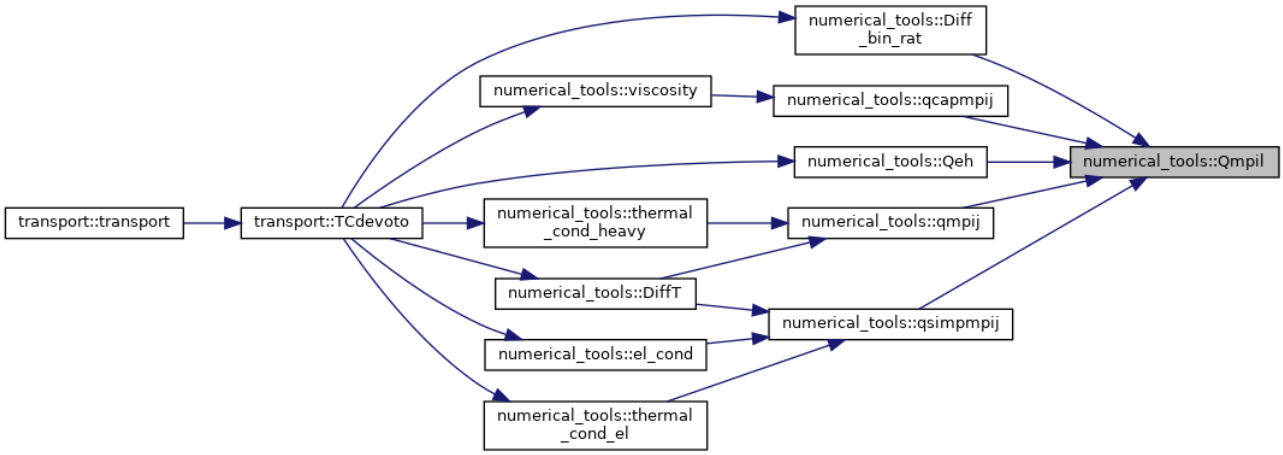


double numerical_tools::Qmpil (double ** *Qt*, int *m*, int *p*, int *i*, int *l*, int *N_SPC*)

extract Qt element for every couple interaction i l

Definition at line 801 of file numerical_tools.cpp.

Here is the caller graph for this function:



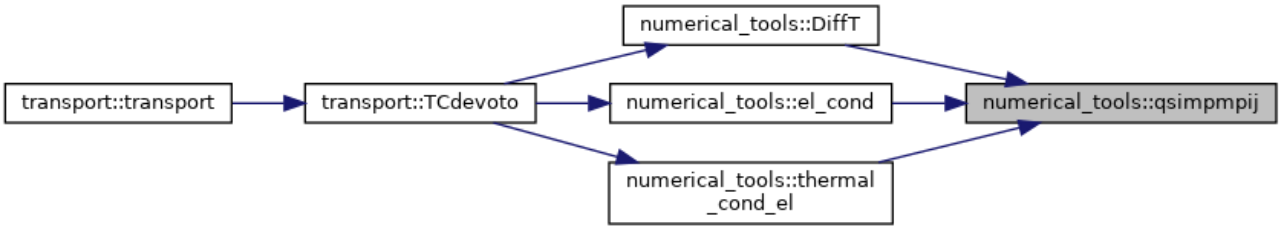
double numerical_tools::qsimpmpij (double ** *Qt*, double * *n*, int *N_SPC*, int *m*, int *p*)

simplified Devoto 08/08/1966 Appendix, assumed electron as last specie

Definition at line 900 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

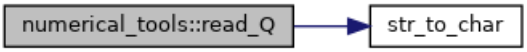


void numerical_tools::read_Q (double * *Q*, int *N_NC*, int *N_C*, int *N_TEMP*, std::vector< std::string > *_filenames*)**

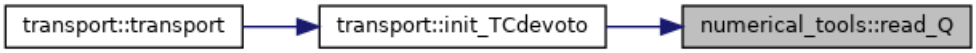
reads cross sections data /* DA GENERALIZZARE */

Definition at line 446 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

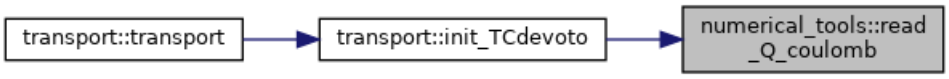


void numerical_tools::read_Q_coulomb (double ** *Qc*, std::string *path*)

reads Coullomb cross sections data /* DA GENERALIZZARE */

Definition at line 480 of file numerical_tools.cpp.

Here is the caller graph for this function:

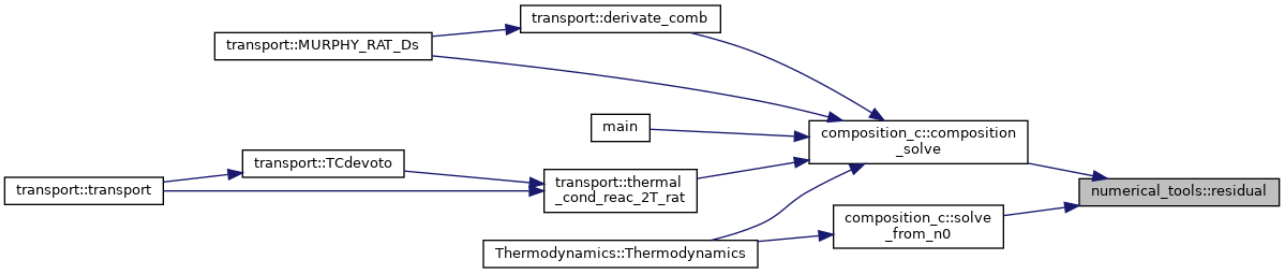


void numerical_tools::residual (double * *R*, double ** *J*, double * *n*, double ** *A*, double * *A0*, double ** *v*, int * *b*, int * *bs*, int *N*, int *M*)

compute residuals as in GODIN eq.35

Definition at line 282 of file numerical_tools.cpp.

Here is the caller graph for this function:

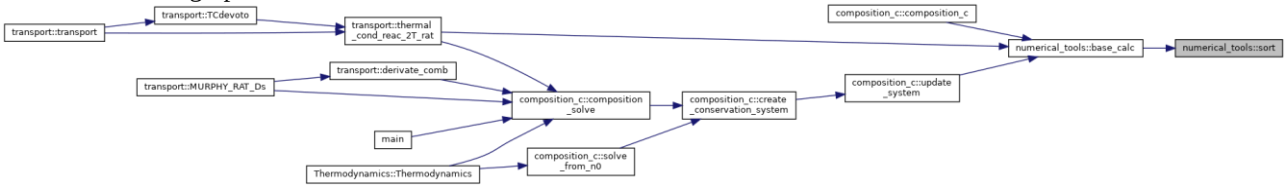


void numerical_tools::sort (double * *arr*, int * *perm*, int *n*)

indices of arr elements in descent order initialize a vector whose component are

Definition at line 37 of file numerical_tools.cpp.

Here is the caller graph for this function:

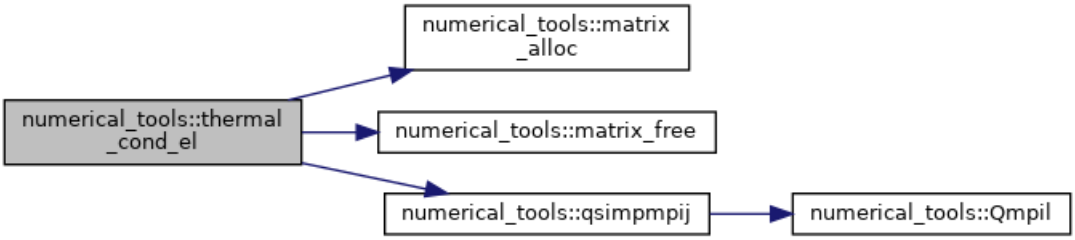


double numerical_tools::thermal_cond_el (double ** *Qt*, double *Te*, double * *n*, double * *mass*, int *N_SPC*)

electron thermal conductivity simplified DEVOTO eq.21

Definition at line 1492 of file numerical_tools.cpp.

Here is the call graph for this function:



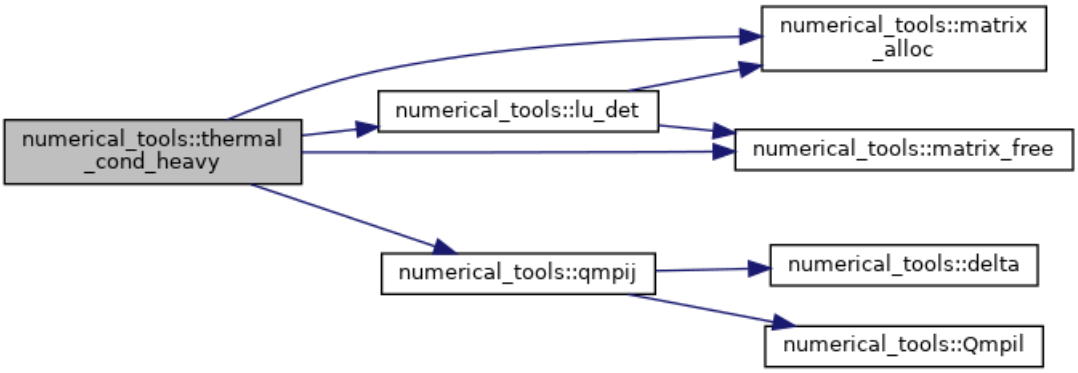
Here is the caller graph for this function:

double numerical_tools::thermal_cond_heavy (double ** *Qt*, double *T*, double * *n*, double * *mass*, int *N_SPC*)

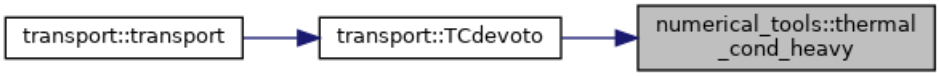
heavy thermal conductivity 2ND order approx

Definition at line 1519 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

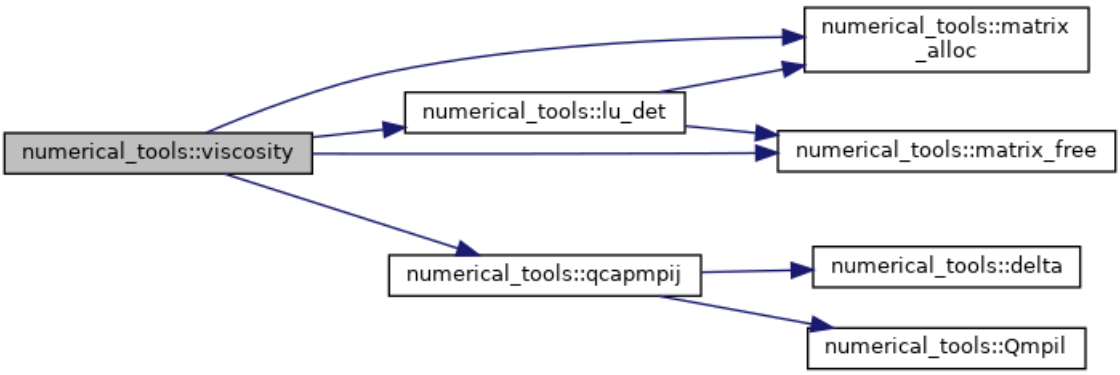


double numerical_tools::viscosity (double ** *Qt*, double *T*, double * *n*, double * *mass*, int *N_SPC*)

funzione calcolo viscosita ///1Approx, to further (2nd available) modify M_ORDV

Definition at line 1645 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

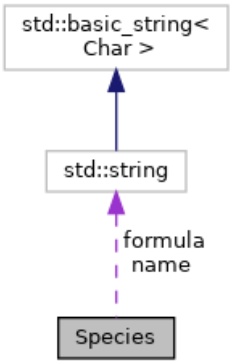
The documentation for this class was generated from the following files:

- tools/**numerical_tools.h**
- tools/**numerical_tools.cpp**

Species Class Reference

`#include <species.h>`

Collaboration diagram for Species:



Public Member Functions

- **Species** ()=default
  Default constructor.

- **Species** (std::string _formula)
  Constructor.

- std::string **get_formula** () const
  outer methods

- std::string **get_name** () const
- bool **is_mf** () const
  true if hold mf_data

- double **get_mf** () const
- void **set_mf** (double new_mf)
  set percentage of abundancy

- void **pf_table** (bool interp_bool)
  set parf interpolation method

- double **m** () const
  return mass of the specie

- double **Ei** () const
  return ionization energy of the specie

- double **Np** () const

return n of particles in molecula

- double **NP** (int **i**)
  return atoms in composite moleaculae

- int **n_tot** ()
  return total number of atoms in composite moleaculae

- double **Mp** (int **i**)
  return masses of costituent species in composite moleaculae

- int **get_np** () const
  return number of partiction coefficients

- double **get_p** (int **i**) const
  return i-partition coefficient

- bool **isElm** () const
  true if element as appear in periodic table

- int **chrg** () const
  return charge of the specie

- bool **do_base** (bool x)
  trigger the element as a base of Godin composition

- bool **is_base** () const
  return true if element set to base

- bool **is_mol** () const
- double **interp** (std::vector< double > x, std::vector< double > y, double xx)
  interpolator

- double **parf_poly** (double T)
  compute the internal partiction function given T by polynomial coefficients in "species.cpp"

- void **print_parf** ()
  print the polynomial computed partition function

- double **parf_interp** (double T)
  compute the internal partiction function given T by table interpolation in "species.cpp"

- double **parf** (double T)
  compute the internal partiction function given T

Private Attributes

- std::string **formula**
  IUPAC formula.

- std::string **name**
  name

- double **mass**
  mass

- double * **masses**
  mass of composites costituents

- double **e_ion**
  ionization energy

- int **n_part**
  number of particles in molecula

- int * **n_Part**
  number of particles in composites moleculae order it alphabetically to build composition matrix

- int **n_TOT**
  atoms in molecula

- int **charge**
  charge +-1, 2, 3 ecc es. O2- charge=-2

- int **np**
  number of coefficients of partiction function

- bool **interpolate** = false
  option for intepolating parf instead of poly fit

- double * **p**
  internal partiction function coefficients

- double **MF**
  percentage of abundancy

- bool **element** {false}
  boolean flag if element as appears in the periodic table

- bool **molecula** {false}
- bool **base** {false}
  if base for ground state GODIN problem

- bool **mf_data** {false}

Detailed Description

species class will contain a catalogue of chemical species i.e. elements, ions, molecule...
Definition at line 15 of file species.h.

Constructor & Destructor Documentation

Species::Species () `[default]`

Default constructor.

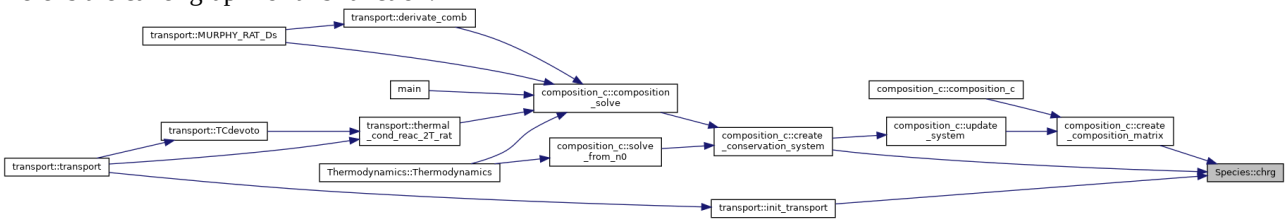
Species::Species (std::string *_formula*)

Constructor.
Definition at line 16 of file species.cpp.

Member Function Documentation

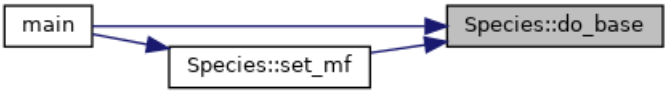
int Species::chrg () const `[inline]`

return charge of the specie
Definition at line 105 of file species.h.
Here is the caller graph for this function:



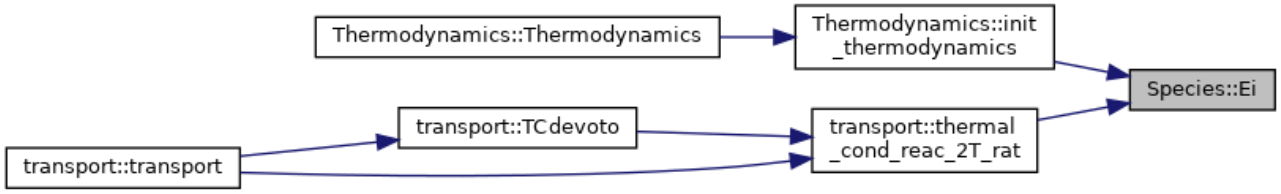
bool Species::do_base (bool *x*) `[inline]`

trigger the element as a base of Godin composition
Definition at line 108 of file species.h.
Here is the caller graph for this function:



double Species::Ei () const `[inline]`

return ionization energy of the specie
Definition at line 81 of file species.h.
Here is the caller graph for this function:

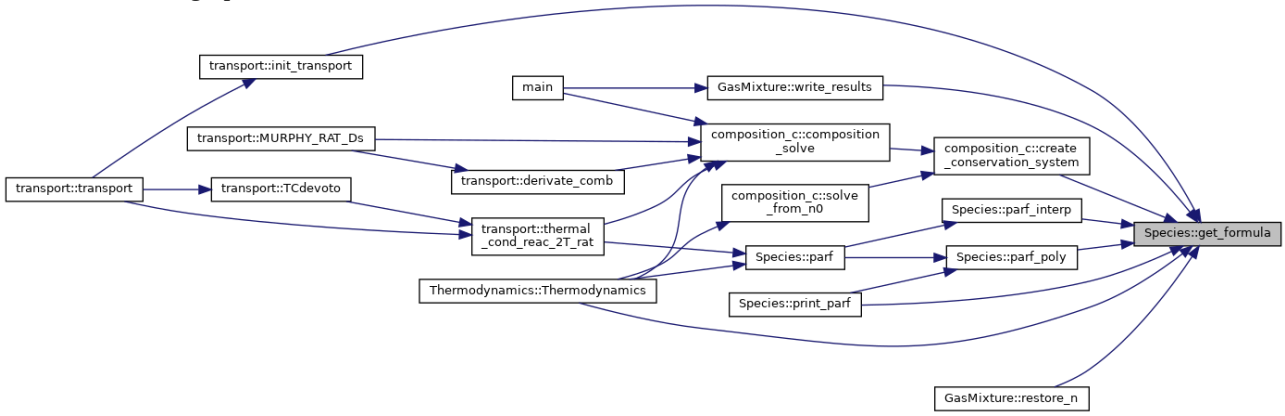


std::string Species::get_formula () const [inline]

outer methods

Definition at line 60 of file species.h.

Here is the caller graph for this function:

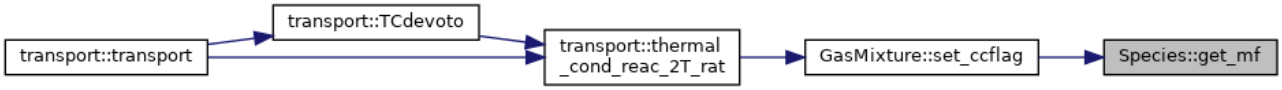


double Species::get_mf () const [inline]

return mf_data as percentage of abundancy mole fraction computed in **composition_c**

Definition at line 69 of file species.h.

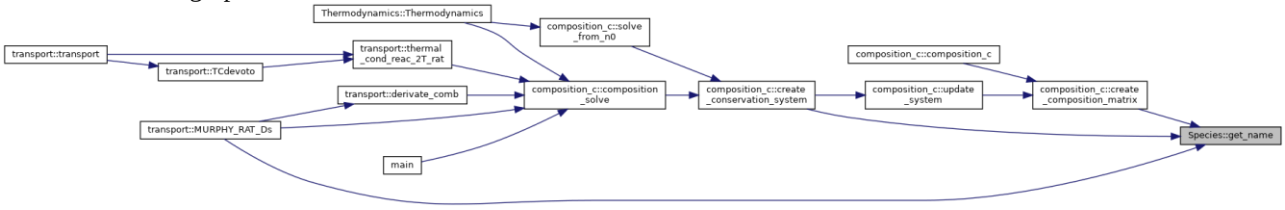
Here is the caller graph for this function:



std::string Species::get_name () const [inline]

Definition at line 62 of file species.h.

Here is the caller graph for this function:



int Species::get_np () const [inline]

return number of partiction coefficients

Definition at line 96 of file species.h.

double Species::get_p (int *i*) const [inline]

return i-partition coefficient

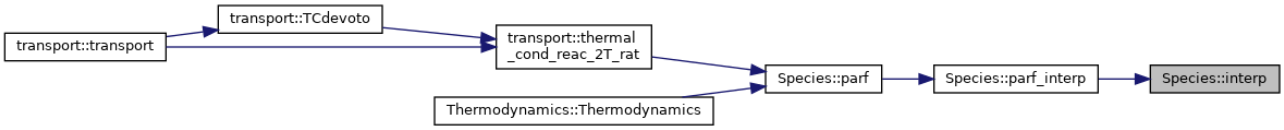
Definition at line 99 of file species.h.

double Species::interp (std::vector< double > *x*, std::vector< double > *y*, double *xx*)

interpolator

Definition at line 656 of file species.cpp.

Here is the caller graph for this function:

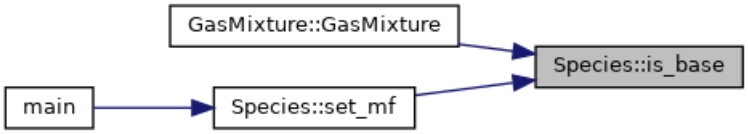


bool Species::is_base () const [inline]

return true if element set to base

Definition at line 111 of file species.h.

Here is the caller graph for this function:

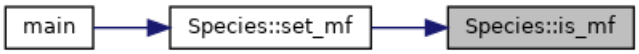


bool Species::is_mf () const [inline]

true if hold mf_data

Definition at line 65 of file species.h.

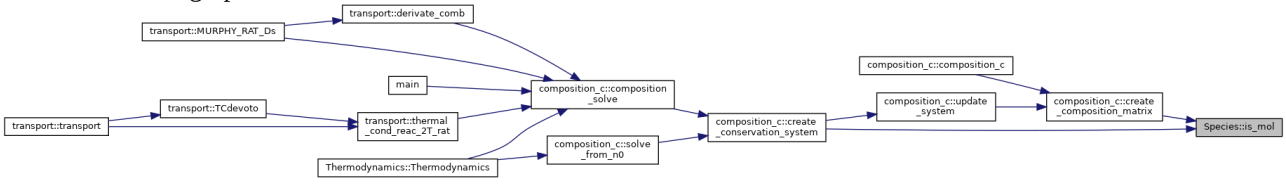
Here is the caller graph for this function:



bool Species::is_mol () const [inline]

Definition at line 113 of file species.h.

Here is the caller graph for this function:



bool Species::isElm () const [inline]

true if element as appear in periodic table

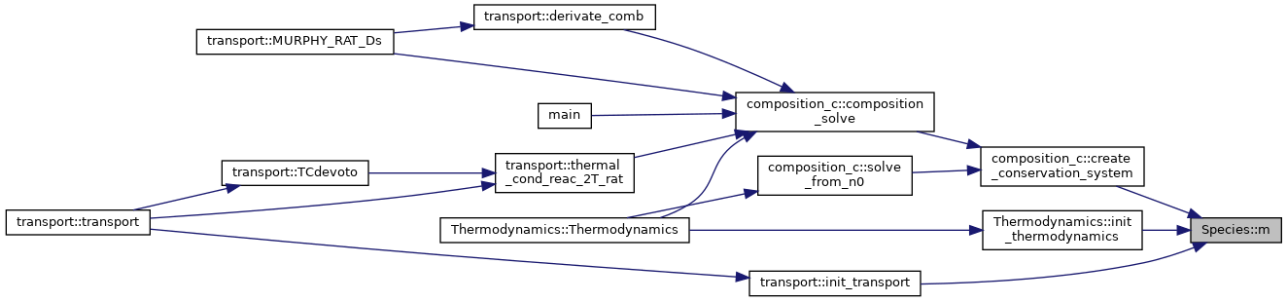
Definition at line 102 of file species.h.

double Species::m () const [inline]

return mass of the specie

Definition at line 78 of file species.h.

Here is the caller graph for this function:

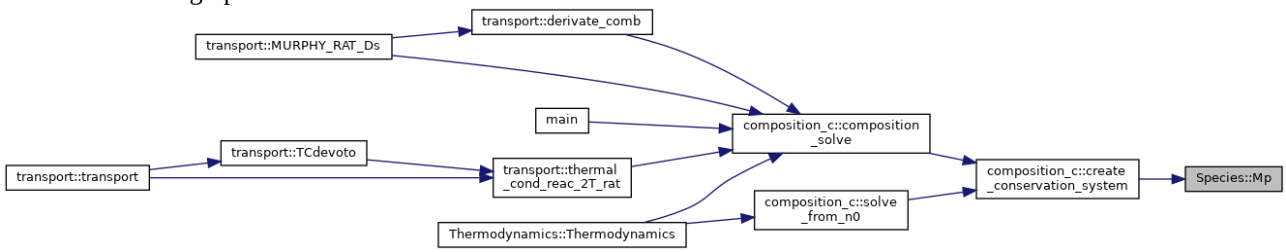


double Species::Mp (int *i*) [inline]

return masses of costituent species in composite moleaculae

Definition at line 93 of file species.h.

Here is the caller graph for this function:

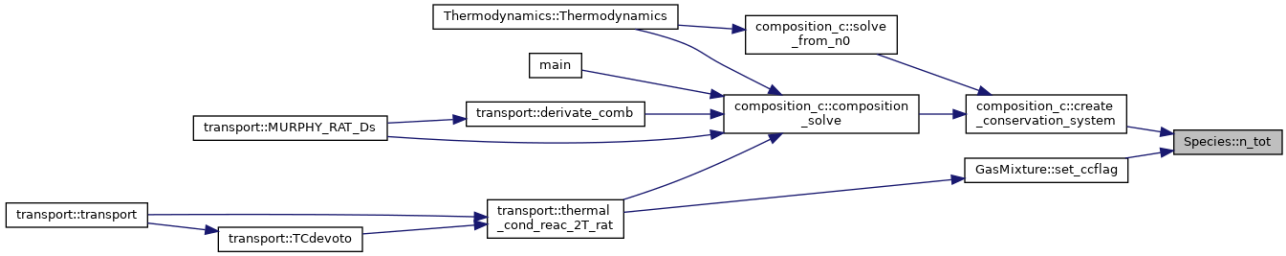


int Species::n_tot () [inline]

return total number of atoms in composite moleaculae

Definition at line 90 of file species.h.

Here is the caller graph for this function:

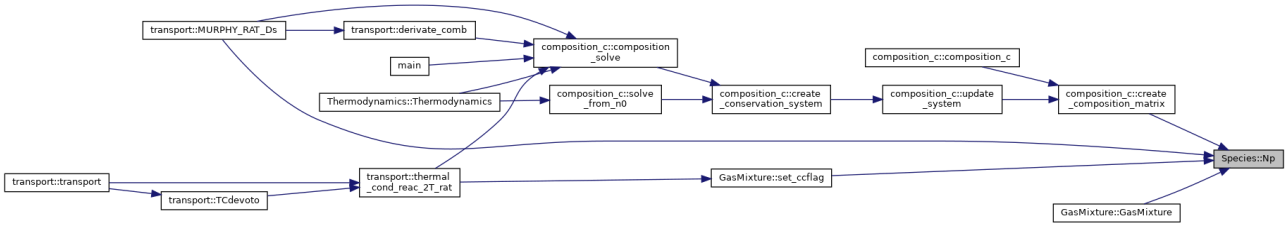


double Species::Np () const [inline]

return n of particles in molecula

Definition at line 84 of file species.h.

Here is the caller graph for this function:

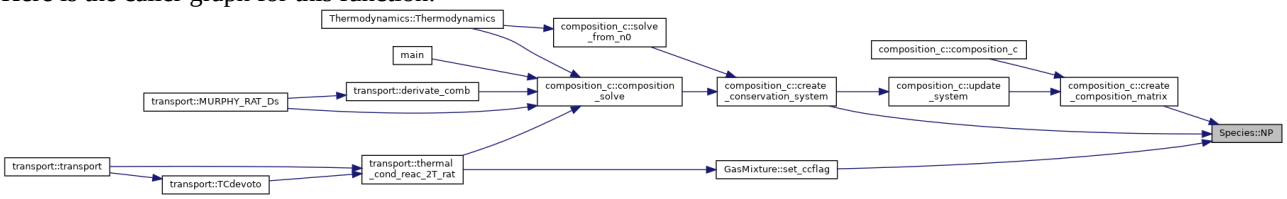


double Species::NP (int *i*) [inline]

return atoms in composite moleaculae

Definition at line 87 of file species.h.

Here is the caller graph for this function:

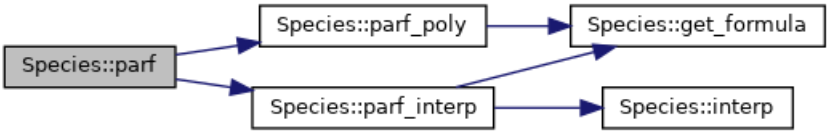


double Species::parf (double *T*)

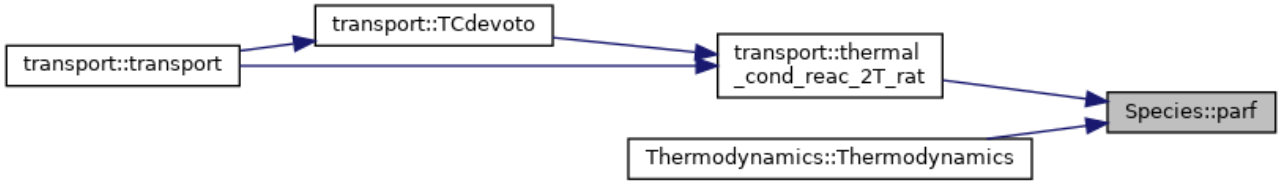
compute the internal partiction function given T

Definition at line 755 of file species.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

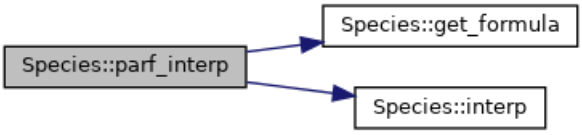


double Species::parf_interp (double *T*)

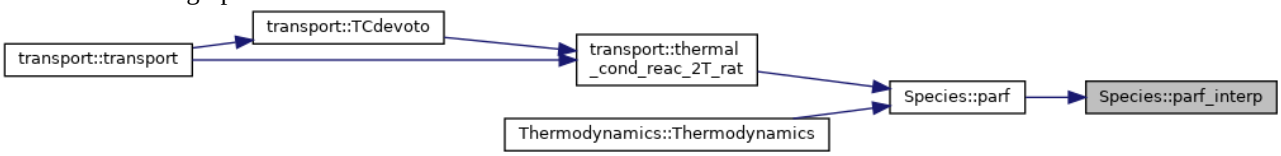
compute the internal partiction function given T by table interpolation in "species.cpp"

Definition at line 706 of file species.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



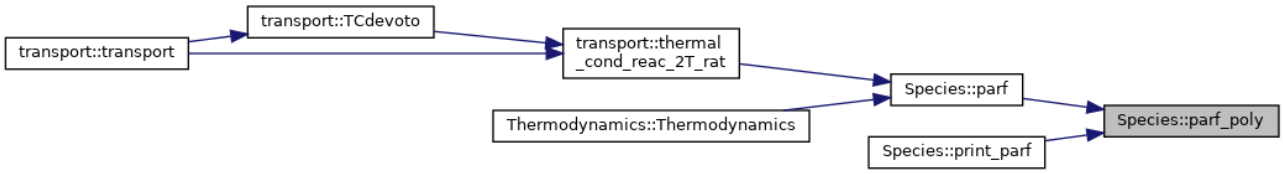
double Species::parf_poly (double *T*)

compute the internal partiction function given T by polynomial coefficients in "species.cpp"

Definition at line 694 of file species.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:



void Species::pf_table (bool *interp_bool*) [inline]

set parf interpolation method

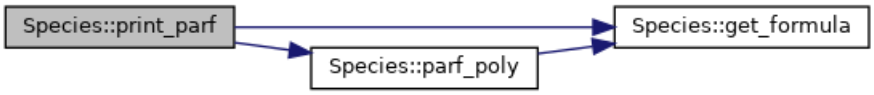
Definition at line 75 of file species.h.

void Species::print_parf ()

print the polynomial computed partition function

Definition at line 764 of file species.cpp.

Here is the call graph for this function:

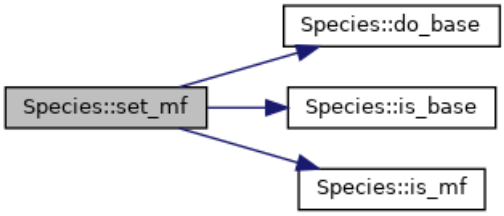


void Species::set_mf (double *new_mf*)

set percentage of abundancy

Definition at line 637 of file species.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

Member Data Documentation

bool Species::base {false} [private]

if base for ground state GODIN problem

Definition at line 44 of file species.h.

int Species::charge [private]

charge +-1, 2, 3 ecc es. O2- charge=-2

Definition at line 30 of file species.h.

double Species::e_ion [private]

ionization energy

Definition at line 25 of file species.h.

bool Species::element {false} [private]

boolean flag if element as appears in the periodic table

Definition at line 40 of file species.h.

std::string Species::formula [private]

IUPAC formula.

Definition at line 19 of file species.h.

bool Species::interpolate = false [private]

option for intepolating parf instead of poly fit

Definition at line 34 of file species.h.

double Species::mass [private]

mass

Definition at line 23 of file species.h.

double* Species::masses [private]

mass of composites costituents

Definition at line 24 of file species.h.

double Species::MF [private]

percentage of abundancy

Definition at line 37 of file species.h.

bool Species::mf_data {false} [private]

contains mf data default false, use mf constructor when inizialize, M-2 mole fraction expressions needed

Definition at line 49 of file species.h.

bool Species::molecula {false} [private]

Definition at line 41 of file species.h.

int Species::n_part [private]

number of particles in molecula
Definition at line 27 of file species.h.

int* Species::n_Part [private]

number of particles in composites moleculae order it alphabetically to build composition matrix
Definition at line 28 of file species.h.

int Species::n_TOT [private]

atoms in molecula
Definition at line 29 of file species.h.

std::string Species::name [private]

name
Definition at line 21 of file species.h.

int Species::np [private]

number of coefficients of partiction function
Definition at line 32 of file species.h.

double* Species::p [private]

internal partiction function coefficients
Definition at line 35 of file species.h.

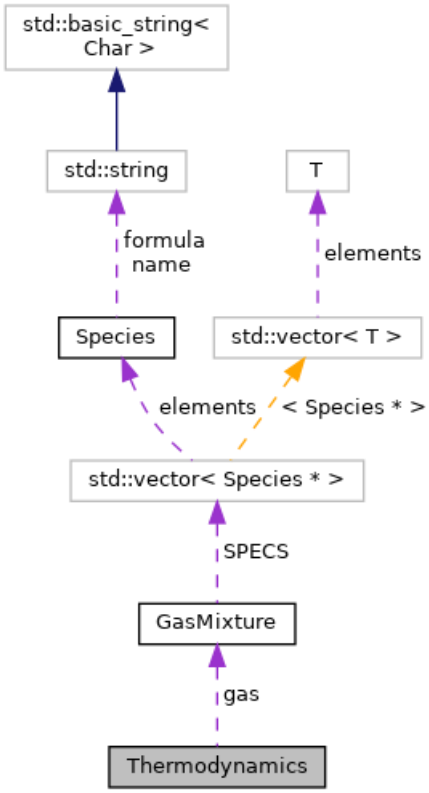
The documentation for this class was generated from the following files:

- species/**species.h**
- species/**species.cpp**

Thermodynamics Class Reference

`#include <thermodynamics.h>`

Collaboration diagram for Thermodynamics:



Public Member Functions

- **Thermodynamics** ()=default
- **Thermodynamics** (**GasMixture** *gas_i)

Private Member Functions

- void **init_thermodynamics** ()

Private Attributes

- **GasMixture** * **gas**
  local copies of gas object

- double * **n_local**
  local composition copy

- double **dT**
  Temperature step for derivatives.

- double **dp**
  pressure step for derivatives

- double **dRdp**
  differential

- double **dRdT**
  differential

- double **rho**
  density

- double **cip**
  constant pressure specific heat

- double **civ**
  constant volume specific heat

- double **he**
  electron hentalphy

- double **ee**
  electron internal energy

- double **hh**
  heavy hentalphy

- double **eh**
  heavy internal energy

- double **vs**
  speed of sound

- double **gamma**
  Cp/Cv.

- double **R**
  gas constant

- double * **nf**
- double **Tf**
- double **thetaf**
- double **rhof**
- double **hf**
- double **ef**
- double **Rf**
- double * **nb**
- double **Tb**
- double **thetab**
- double **rhob**
- double **hb**
- double **eb**

- double **Rb**
- double * **mass**
  gasmixture's species masses

- double * **e_ion**
  gasmixture's species ionization energy

- double **PRESS**
  gasmixture's pressure

- double **TEMP**
  gasmixture's Temperature

- double **THETA**
  gasmixture's non-equilibrium parameter

- int **N**
  gasmixture's number of species

- int **M**
  gasmixture's number of elements

Detailed Description

Definition at line 7 of file thermodynamics.h.

Constructor & Destructor Documentation

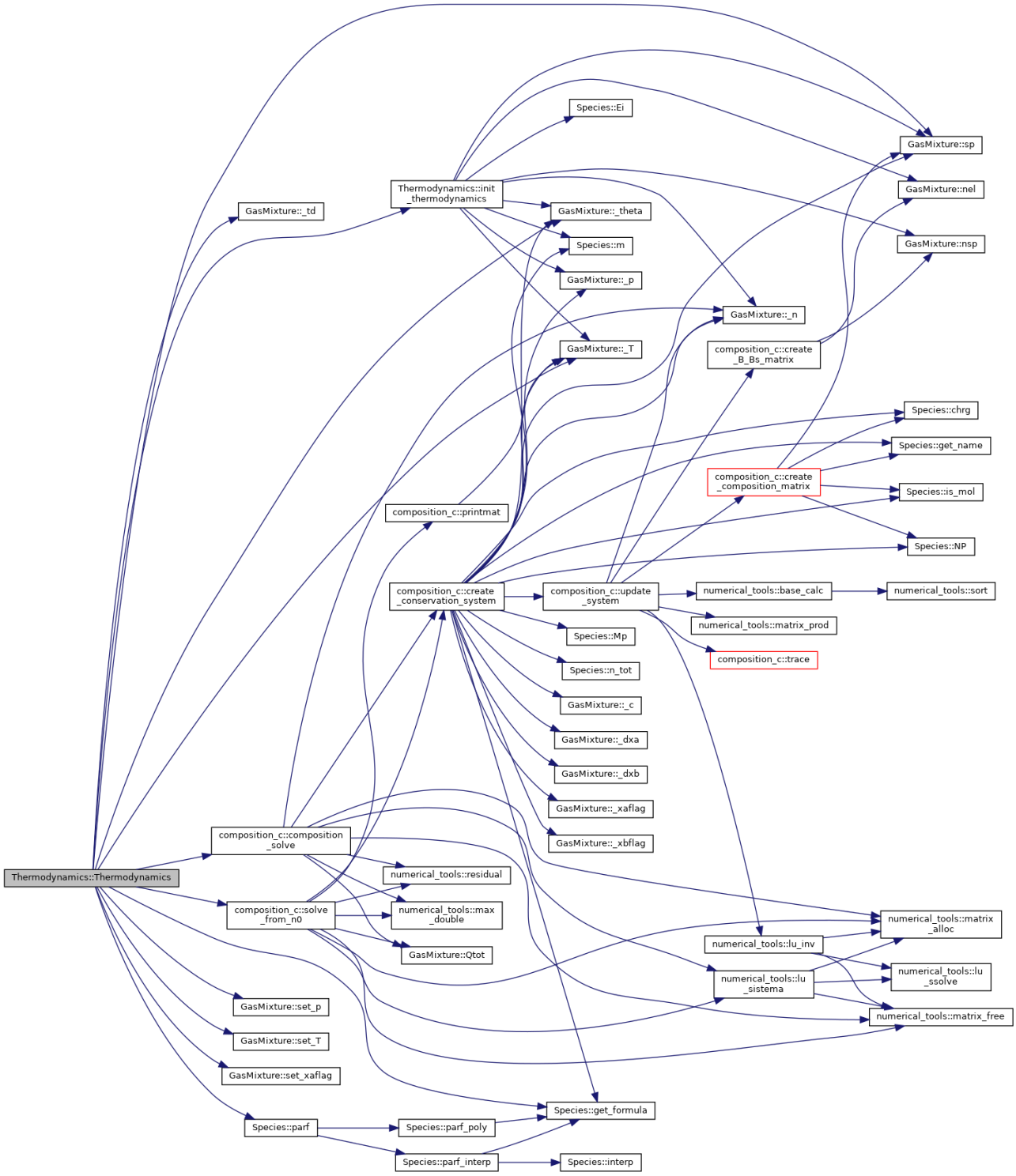
Thermodynamics::Thermodynamics () `[default]`

Thermodynamics::Thermodynamics (GasMixture * *gas_i*)

compute thermodynamic proprerties from GODIN composition this class will work only constructing the relative object

Definition at line 6 of file thermodynamics.cpp.

Here is the call graph for this function:

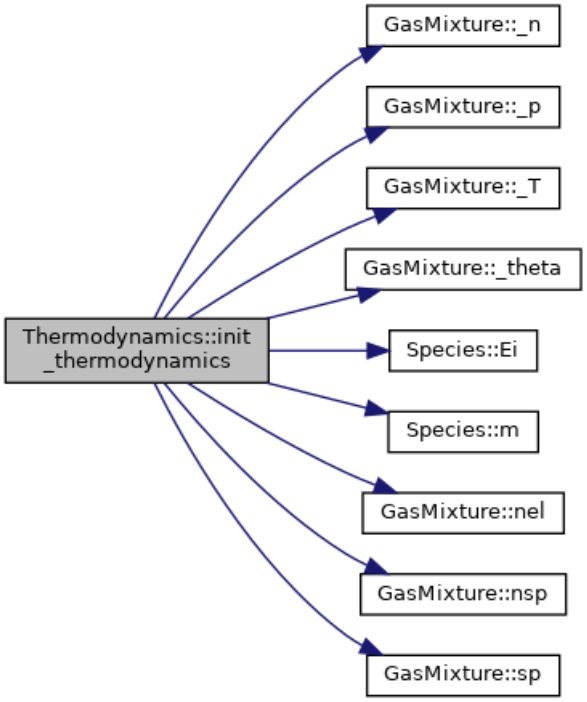


Member Function Documentation

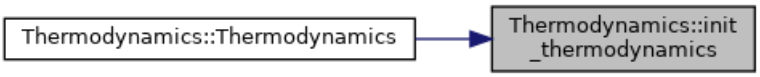
void Thermodynamics::init_thermodynamics () [private]

Definition at line 207 of file thermodynamics.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



Member Data Documentation

double Thermodynamics::cip [private]

constant pressure specific heat

Definition at line 22 of file thermodynamics.h.

double Thermodynamics::civ [private]

constant volume specific heat

Definition at line 23 of file thermodynamics.h.

double Thermodynamics::dp [private]

pressure step for derivatives

Definition at line 18 of file thermodynamics.h.

double Thermodynamics::dRdp [private]

differential

Definition at line 19 of file thermodynamics.h.

double Thermodynamics::dRdT [private]

differential

Definition at line 20 of file thermodynamics.h.

double Thermodynamics::dT [private]

Temperature step for derivatives.

Definition at line 17 of file thermodynamics.h.

double* Thermodynamics::e_ion [private]

gasmixture's species ionization energy

Definition at line 43 of file thermodynamics.h.

double Thermodynamics::eb [private]

Definition at line 35 of file thermodynamics.h.

double Thermodynamics::ee [private]

electron internal energy

Definition at line 25 of file thermodynamics.h.

double Thermodynamics::ef [private]

Definition at line 33 of file thermodynamics.h.

double Thermodynamics::eh [private]

heavy internal energy

Definition at line 27 of file thermodynamics.h.

double Thermodynamics::gamma [private]

Cp/Cv.

Definition at line 29 of file thermodynamics.h.

GasMixture* Thermodynamics::gas [private]

local copies of gas object

Definition at line 12 of file thermodynamics.h.

double Thermodynamics::hb [private]

Definition at line 35 of file thermodynamics.h.

double Thermodynamics::he `[private]`

electron hentalphy
Definition at line 24 of file thermodynamics.h.

double Thermodynamics::hf `[private]`

Definition at line 33 of file thermodynamics.h.

double Thermodynamics::hh `[private]`

heavy hentalphy
Definition at line 26 of file thermodynamics.h.

int Thermodynamics::M `[private]`

gasmixture's number of elements
Definition at line 48 of file thermodynamics.h.

double* Thermodynamics::mass `[private]`

gasmixture's species masses
Definition at line 42 of file thermodynamics.h.

int Thermodynamics::N `[private]`

gasmixture's number of species
Definition at line 47 of file thermodynamics.h.

double* Thermodynamics::n_local `[private]`

local composition copy
Definition at line 14 of file thermodynamics.h.

double* Thermodynamics::nb `[private]`

Definition at line 35 of file thermodynamics.h.

double* Thermodynamics::nf `[private]`

Definition at line 33 of file thermodynamics.h.

double Thermodynamics::PRESS `[private]`

gasmixture's pressure
Definition at line 44 of file thermodynamics.h.

double Thermodynamics::R `[private]`

gas constant
Definition at line 30 of file thermodynamics.h.

double Thermodynamics::Rb `[private]`

Definition at line 35 of file thermodynamics.h.

double Thermodynamics::Rf `[private]`

Definition at line 33 of file thermodynamics.h.

double Thermodynamics::rho `[private]`

density
Definition at line 21 of file thermodynamics.h.

double Thermodynamics::rhob `[private]`

Definition at line 35 of file thermodynamics.h.

double Thermodynamics::rhof `[private]`

Definition at line 33 of file thermodynamics.h.

double Thermodynamics::Tb `[private]`

Definition at line 35 of file thermodynamics.h.

double Thermodynamics::TEMP `[private]`

gasmixture's Temperature
Definition at line 45 of file thermodynamics.h.

double Thermodynamics::Tf `[private]`

Definition at line 33 of file thermodynamics.h.

double Thermodynamics::THETA `[private]`

gasmixture's non-equilibrium parameter
Definition at line 46 of file thermodynamics.h.

double Thermodynamics::thetab `[private]`

Definition at line 35 of file thermodynamics.h.

double Thermodynamics::thetaf [private]

Definition at line 33 of file thermodynamics.h.

double Thermodynamics::vs [private]

speed of sound

Definition at line 28 of file thermodynamics.h.

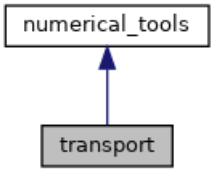
The documentation for this class was generated from the following files:

- thermodynamics/**thermodynamics.h**
- thermodynamics/**thermodynamics.cpp**

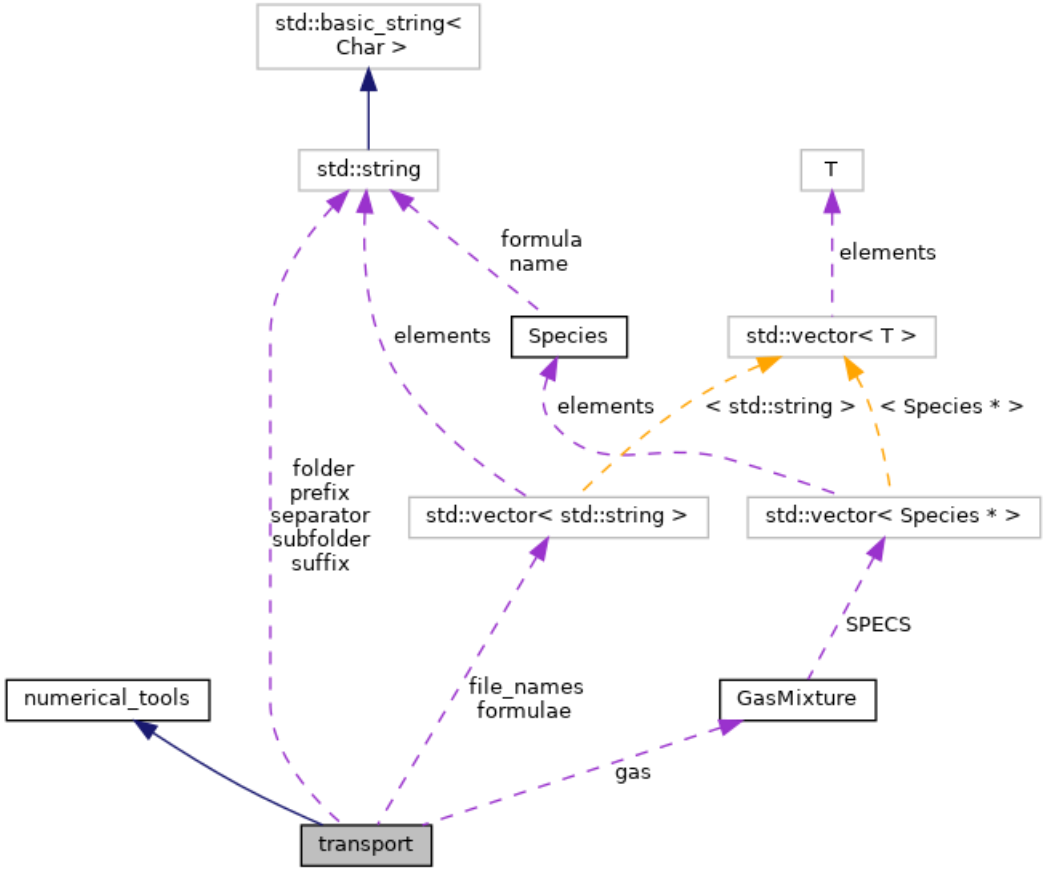
transport Class Reference

`#include <transport.h>`

Inheritance diagram for transport:



Collaboration diagram for transport:



Public Member Functions

- **transport** ()=default
- **transport** (**GasMixture** *gas_i)
- void **init_transport** ()
  initialize transport object data structure

- void **init_TCdevoto** ()
  Initialize Devoto by reading cross sections data.

- void **QtCalc** ()
  Compute Qt data structure with Q & Qc data DEVOTO eqns. 5-6.

- void **TCdevoto** ()
- void **thermal_cond_reac_2T_rat** ()
  reactive thermal conductivity

- void **derivate_comb** ()
  compute diffusion coeffiecients defined by MURPHY-RAT-combined diffusion

- void **MURPHY_RAT_Ds** ()
- int **get_Cint** ()
  gets number of Coulomb interaction

- int **get_NCint** ()
  gets number of non Coulomb interaction

- int **get_BINint** ()
  gets number of binary interactions

- bool **foo** (int **i**, int j, double **N**, double **M**)
- void **sort** (double *arr, int *perm, int n)
- void **matrix_prod** (double **A, double **B, double **C, int N1, int N2)
- double ** **matrix_alloc** (int **N**, int **M**)
  *initialize size of **A objects*

- int ** **matrix_alloc_int** (int n, int m)
  same as matrix_alloc but int variables

- void **matrix_free** (double **matrix, int n)
  clears a matrix

- void **lu_ssolve** (double *x, int *p, double *b0, double **m, int **N**)
  solve lu decomposition of a matrix

- void **lu_inv** (double **AINV, double **A, int **N**)
  use lu algorithm to compute the inverse matrix

- void **base_calc** (double *n, int *b, int *bs, double **C, int **N**, int **M**)
  track C rows to set basis & not-basis species based on n density vestor

- void **residual** (double *R, double **J, double *n, double **A, double *A0, double **v, int *b, int *bs, int **N**, int **M**)
  compute residuals as in GODIN eq.35

- void **lu_sistema** (double *x, double **A, double *b, int **N**)
  solve an LU system

- double **lu_det** (double **A, int **N**)
  *compute det(**A) with LU scomposition*

- double **max_double** (double *arr, int **N**)
  return maximum element of an array

- void **read_Q** (double ***Q**, int N_NC, int N_C, int **N_TEMP**, std::vector< std::string > _filenames)
  reads cross sections data / DA GENERALIZZARE */*

- void **read_Q_coulomb** (double **Qc, std::string path)
  reads Coullomb cross sections data / DA GENERALIZZARE */*

- void **il_calc** (int ***i**, int *l, int il, int N_specs)
- int **Qij_calc** (int il, int ***Z**, int N_specs)
- double **Q_gen** (double TT, double **T**, double theta, double *n, int ***Z**, int mp, int **i**, int j, int N_SPC)
  generates firsts collision integrals

- double **Q_coulomb** (double **Qc, double TT, double **T**, double theta, double *n, int ***Z**, int mp, int **i**, int j, int N_SPC)
  generates Coulomb collisions integral

- double **interp** (double *x, double *y, double xx, int N_xy)
  interpolator

- double **Diff_bin_rat** (double **Qt, double TT, double **T**, double theta, double *mass, double **nn**, int **i**, int j, int N_SPC)
  Binary diffusion as RAT et. al. 30Apr2002.

- double **Qmpil** (double **Qt, int m, int p, int **i**, int l, int N_SPC)
  extract Qt element for every couple interaction i l

- double **Fij_calc** (double **Dbin, double *n, double *mass, double rho, int **i**, int j, int N_SPC)
  Colombo,Ghedini,Sanibondi UNIBO 20/01/2009 eq.17.

- double **Fij_cofactor** (double **Fij, int **i**, int j, int N_SPC)
- int **delta** (int **i**, int j)
  Kronecker delta.

- double **delta_double** (int **i**, int j)
  Kronecker delta.

- double **qsimpmpij** (double **Qt, double *n, int N_SPC, int m, int p)
  simplified Devoto 08/08/1966 Appendix, assumed electron as last specie

- double **qmpij** (double **Qt, double *n, double *mass, int N_SPC, int m, int p, int **i**, int j)
  DEVOTO Appendix 06/1966.

- double **DiffT** (double **Qt, double **T**, double theta, double *n, double *mass, int N_SPC, int ii)
- double **Diff_amb** (double **D, double **T**, double theta, double *mass, double *n, int ***Z**, int ii, int jj, int N_SPC)

ambipolar diffusion and thermal diffusion

- double **Diff_T_amb** (double ****D**, double ***DT**, double **T**, double theta, double *mass, double *n, int ***Z**, int ii, int N_SPC)
- double **thermal_cond_el** (double ****Qt**, double Te, double *n, double *mass, int N_SPC)
  electron thermal conductivity simplified DEVOTO eq.21

- double **thermal_cond_heavy** (double ****Qt**, double **T**, double *n, double *mass, int N_SPC)
  heavy thermal conductivity 2ND order approx

- double **viscosity** (double ****Qt**, double **T**, double *n, double *mass, int N_SPC)
  funzione calcolo viscosita ///1Approx, to further (2nd available) modify M_ORDV

- double **qcapmpij** (double ****Qt**, double *n, double *mass, int N_SPC, int m, int p, int **i**, int j)
  elementi per calcolo bracket integrals da teoria DEVOTO per viscosità

- double **el_cond** (double ****Qt**, double **T**, double theta, double *n, double *mass, int N_SPC)
  DEVOTO simplified electrical conductivity eq.16.

- double **Qeh** (double ****Qt**, double **T**, double theta, double *mass, double *n, int N_SPC)
  thermal exchange electron-heavy

- double **Dx_AB_comp** (double **Da, double theta, double *mass, double *n, double *DxDxB, double *b, int pp, int N_SPC)
  MURPHY-RAT ordinary diffusion coefficients eq. 2.36.

- double **DYAB_comb** (double DxxAB, double *nf, double *nb, double dya, double *n, double rho, double *mass, int ***Z**, int p, int N_SPC)
  UNIBO 2008 diffusion coefficients.

- double **DT_AB_comp** (double **Da, double ***DT**, double Temp, double theta, double rho, double *mass, double *n, double *b, double *DxxDT, int p, int N_SPC)
- double **DtAB_comb_y** (double **T**, double dT, double DtAB, double Dxab, double *n_f, double *n_b, double rho, double *n, double *mass, int *_Z, int p, int N_SPC)

Private Attributes

- **GasMixture** * **gas**
  local gas pointer

- int **N**
- int **M**
- double **T**
- double **Theta**
- std::vector< std::string > **formulae**
- double *** **Q**
  cross section data

- double ** **Qc**
  Coulomb cross section data.

- double ** **Qt**
  Devoto matrix as in DEVOTO eq.7.

- double ** **Dbin**
  binary diffusion coefficients RAT et.al.30Apr2002

- double ** **Fij**
- double ** **Fij_co**
- double ** **D**
  Fij factors.

- double ** **D_amb**
  ambipolar diffusion coefficients

- double ** **E**
- double ** **Einv**
- double * **DT**
  matrix to compute thermal reactive conductivity

- double * **DTamb**
  Ambipolar Thermal diffusion.

- double * **TC**
  12 tranport properties

- double * **n_**
  local particle densities copies

- double * **m_**
  vector of species masses [g]

- int * **Z**
  vector of particles charges

- double **nn**
  total particle density

- double **rho_**
  density

- int **C_interaction**
  Coulomb collision.

- double **kB**
  to compute transport properties K-boltzmann is in fJ/K° units

- int **NC_interaction**

Non-Coulomb collisions.

- int **N_TEMP** = 448
  Temperature interpolation steps.

- int **N_BINARY**
  number of binary collisions

- int **N_4TH** = 4
  level of approximation desired

- double * **DxDxb**
- double * **DxDT**
  combined diffusion partial derivatives

- double **DxAB**
  2T combined diffusion coefficient due to gradients in concentration MURPHY-RAT 2.36

- double **DYAB**
  diffusion coefficient from UNIBO 08

- double **DTAB**
  combined thermal diffusion coefficient MURPHY-RAT

- double **DTABy**
  combined diffusion coefficient UNIBO 09 fluent

- std::vector< std::string > **file_names**
- std::filesystem::path **DevotoGodin_path**
- std::string **folder**
- std::string **subfolder** {"/CI_database/"}
- std::string **prefix** {"Q4th_"}
- std::string **separator** {"_"}
- std::string **suffix** {".txt"}

Detailed Description

transport class will compute transport properties starting from composition and thermodynamics of a gasmixture

Definition at line 14 of file transport.h.

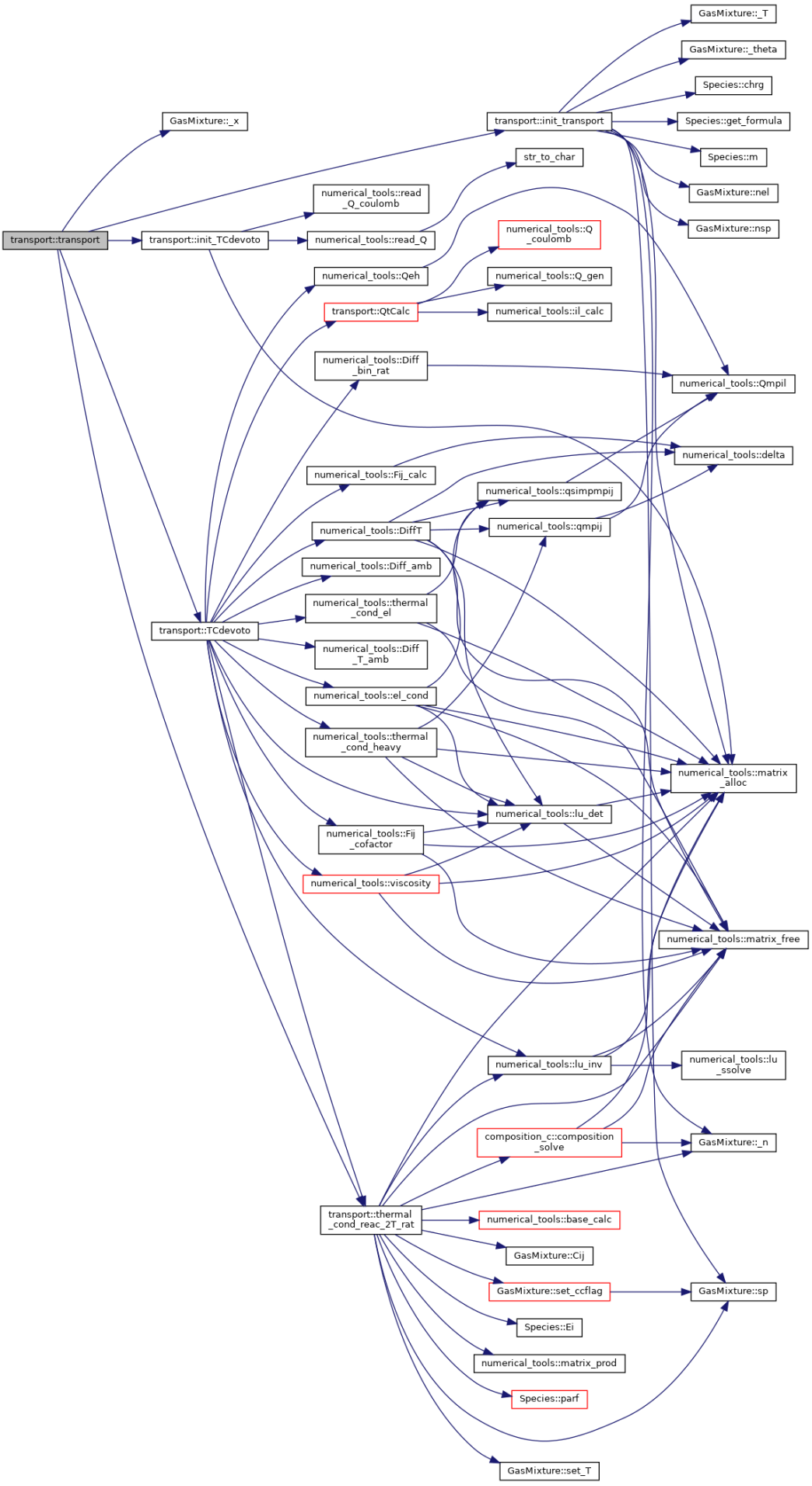
Constructor & Destructor Documentation

transport::transport () `[default]`

transport::transport (GasMixture * *gas_i*)

Definition at line 4 of file transport.cpp.

Here is the call graph for this function:


transport::transport
GasMixture::_x
transport::init_TCdevoto
numerical_tools::read_Q_coulomb
numerical_tools::read_Q
str_to_char
transport::init_transport
GasMixture::_T
GasMixture::_theta
Species::chrg
Species::get_formula
Species::m
GasMixture::nel
GasMixture::nsp
numerical_tools::Q_coulomb
numerical_tools::Qeh
numerical_tools::Q_gen
transport::QtCalc
numerical_tools::il_calc
numerical_tools::Diff_bin_rat
numerical_tools::Qmpil
numerical_tools::delta
numerical_tools::Fij_calc
numerical_tools::qsimpmpij
numerical_tools::DiffT
numerical_tools::qmpij
numerical_tools::Diff_amb
numerical_tools::thermal_cond_el
transport::TCdevoto
numerical_tools::Diff_T_amb
numerical_tools::el_cond
numerical_tools::thermal_cond_heavy
numerical_tools::matrix_alloc
numerical_tools::lu_det
numerical_tools::Fij_cofactor
numerical_tools::viscosity
numerical_tools::matrix_free
numerical_tools::lu_inv
numerical_tools::lu_ssolve
composition_c::composition_solve
GasMixture::_n
transport::thermal_cond_reac_2T_rat
numerical_tools::base_calc
GasMixture::Cij
GasMixture::set_ccflag
GasMixture::sp
Species::Ei
numerical_tools::matrix_prod
Species::parf
GasMixture::set_T

Member Function Documentation

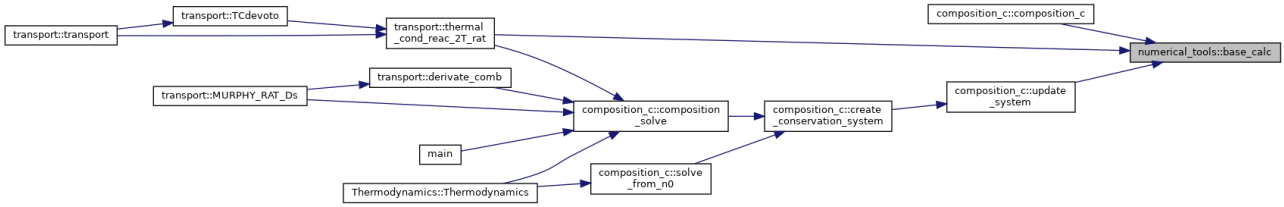
void numerical_tools::base_calc (double * *n*, int * *b*, int * *bs*, double ** *C*, int *N*, int *M*) [inherited]

track C rows to set basis & not-basis species based on n density vestor

Definition at line 230 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

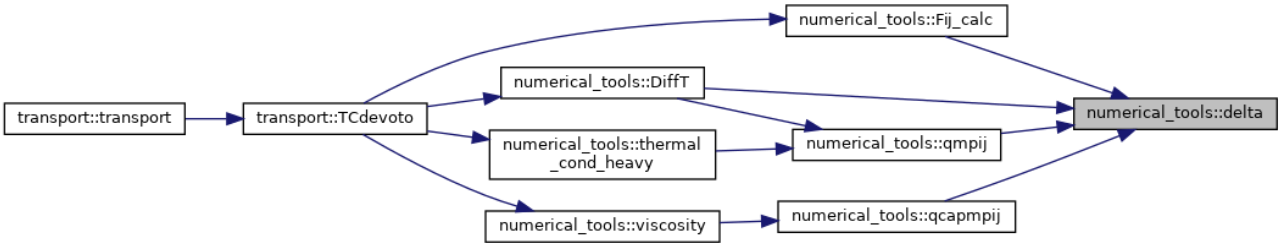


int numerical_tools::delta (int *i*, int *j*) [inherited]

Kronecker delta.

Definition at line 886 of file numerical_tools.cpp.

Here is the caller graph for this function:



double numerical_tools::delta_double (int *i*, int *j*) [inherited]

Kronecker delta.

Definition at line 892 of file numerical_tools.cpp.

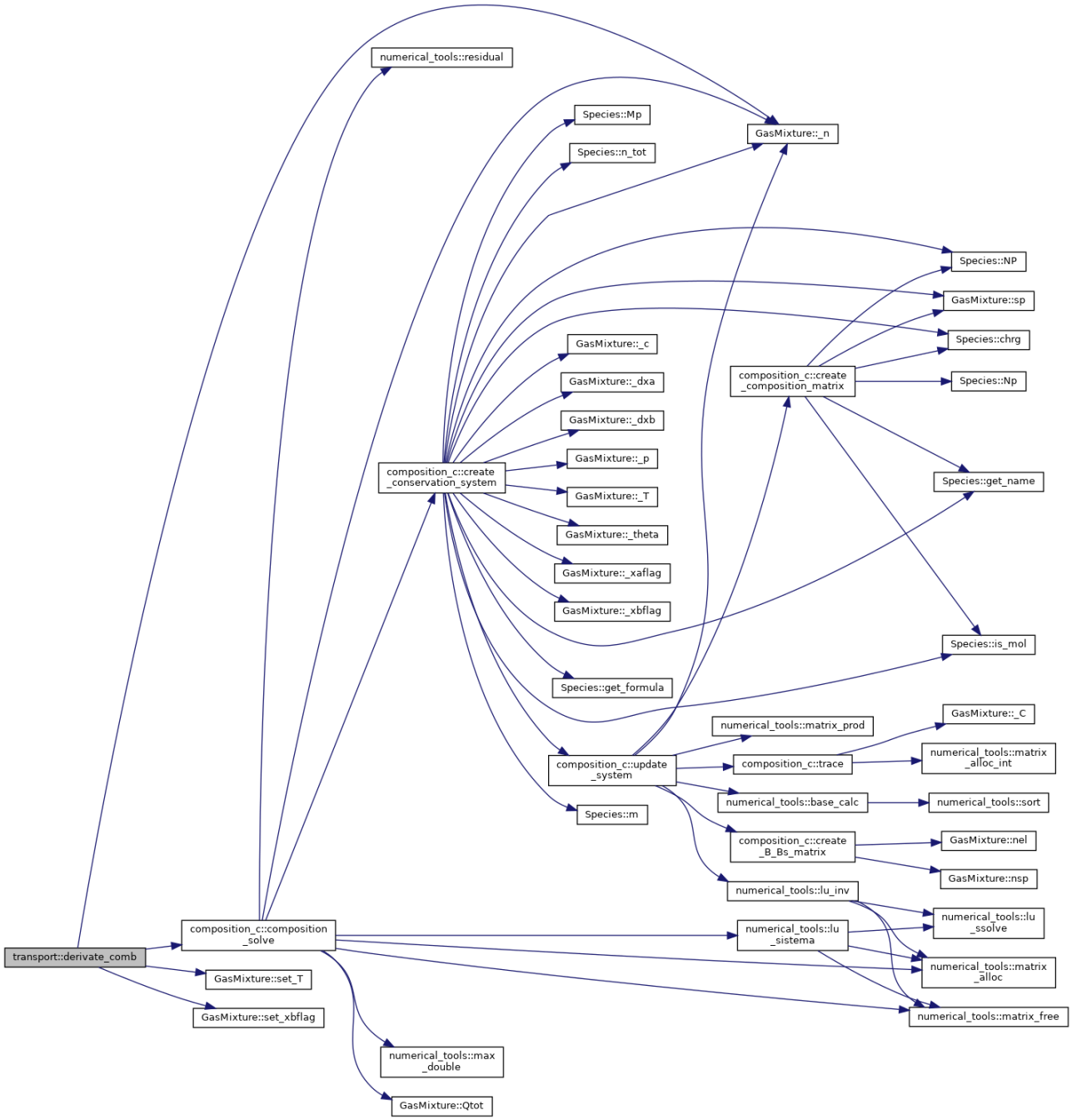
void transport::derivate_comb ()

compute diffusion coeffiecients defined by MURPHY-RAT-combined diffusion

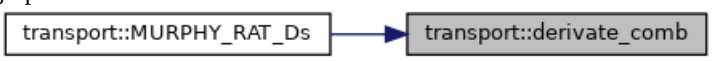
compute combined diffusion derivatives

Definition at line 559 of file transport.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



double numerical_tools::Diff_amb (double ** *D*, double *T*, double *theta*, double * *mass*, double * *n*, int * *Z*, int *ii*, int *jj*, int *N_SPC*) `[inherited]`

ambipolar diffusion and thermal diffusion

Definition at line 1417 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::Diff_bin_rat (double ** *Qt*, double *TT*, double *T*, double *theta*, double * *mass*, double *nn*, int *i*, int *j*, int *N_SPC*) `[inherited]`

Binary diffusion as RAT et. al. 30Apr2002.

Definition at line 779 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

double numerical_tools::Diff_T_amb (double ** *D*, double * *DT*, double *T*, double *theta*, double * *mass*, double * *n*, int * *Z*, int *ii*, int *N_SPC*) `[inherited]`

Definition at line 1455 of file numerical_tools.cpp.

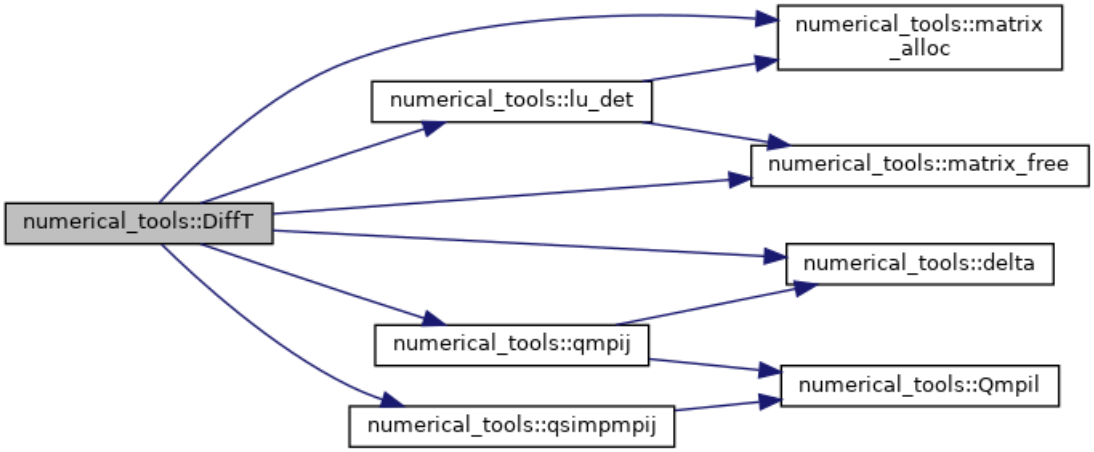
Here is the caller graph for this function:

double numerical_tools::DiffT (double ** *Qt*, double *T*, double *theta*, double * *n*, double * *mass*, int *N_SPC*, int *ii*) `[inherited]`

Temperature Diffusion computed with Devoto 4th approx for electrons and 2nd for heavy particles, modify N_ORD_H for further approx

Definition at line 1342 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

double numerical_tools::DT_AB_comp (double ** *Da*, double * *DT*, double *Temp*, double *theta*, double *rho*, double * *mass*, double * *n*, double * *b*, double * *DxxDT*, int *p*, int *N_SPC*) `[inherited]`

Definition at line 1886 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::DtAB_comb_y (double *T*, double *dT*, double *DtAB*, double *Dxab*, double * *n_f*, double * *n_b*, double *rho*, double * *n*, double * *mass*, int * *_Z*, int *p*, int *N_SPC*) `[inherited]`

Definition at line 1955 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::Dx_AB_comp (double ** *Da*, double *theta*, double * *mass*, double * *n*, double * *DxDxB*, double * *b*, int *pp*, int *N_SPC*) `[inherited]`

MURPHY-RAT ordinary diffusion coefficients eq. 2.36.

Definition at line 1746 of file numerical_tools.cpp.

Here is the caller graph for this function:

double numerical_tools::DYAB_comb (double *DxxAB*, double * *nf*, double * *nb*, double *dya*, double * *n*, double *rho*, double * *mass*, int * *Z*, int *p*, int *N_SPC*) `[inherited]`

UNIBO 2008 diffusion coefficients.

Definition at line 1826 of file numerical_tools.cpp.

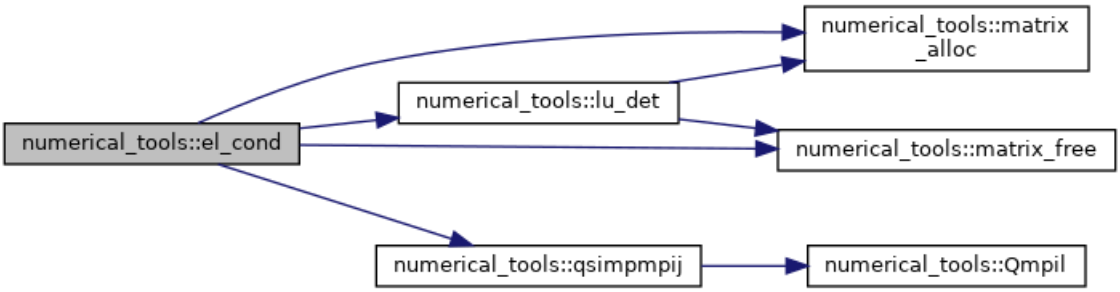
Here is the caller graph for this function:

double numerical_tools::el_cond (double ** *Qt*, double *T*, double *theta*, double * *n*, double * *mass*, int *N_SPC*) `[inherited]`

DEVOTO simplified electrical conductivity eq.16.

Definition at line 1688 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

double numerical_tools::Fij_calc (double ** *Dbin*, double * *n*, double * *mass*, double *rho*, int *i*, int *j*, int *N_SPC*) [inherited]

Colombo,Ghedini,Sanibondi UNIBO 20/01/2009 eq.17.

Definition at line 842 of file numerical_tools.cpp.

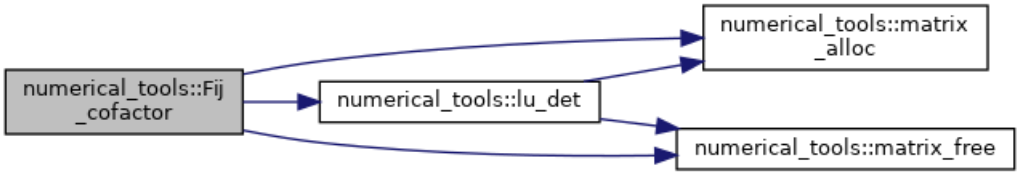
Here is the call graph for this function:

Here is the caller graph for this function:

double numerical_tools::Fij_cofactor (double ** *Fij*, int *i*, int *j*, int *N_SPC*) [inherited]

Definition at line 856 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

bool numerical_tools::foo (int *i*, int *j*, double *N*, double *M*) [inherited]

int transport::get_BINint ()

gets number of binary interactions

Definition at line 664 of file transport.cpp.

int transport::get_Cint ()

gets number of Coulomb interaction

Definition at line 658 of file transport.cpp.

int transport::get_NCint ()

gets number of non Coulomb interaction

Definition at line 661 of file transport.cpp.

void numerical_tools::il_calc (int * *i*, int * *I*, int *il*, int *N_specs*) [inherited]

Associates species couple indexes to binary interaction progressively numerated (il)

Definition at line 502 of file numerical_tools.cpp.

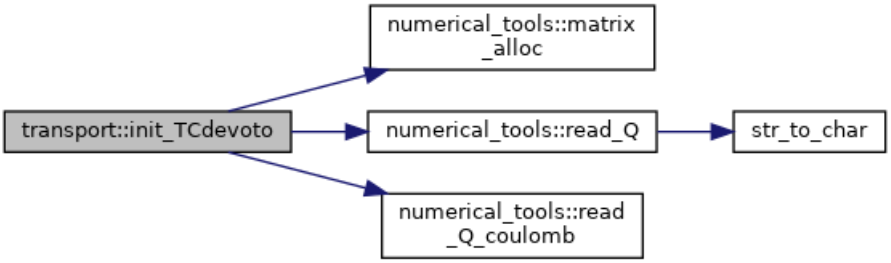
Here is the caller graph for this function:

void transport::init_TCdevoto ()

Initialize Devoto by reading cross sections data.

Definition at line 130 of file transport.cpp.

Here is the call graph for this function:



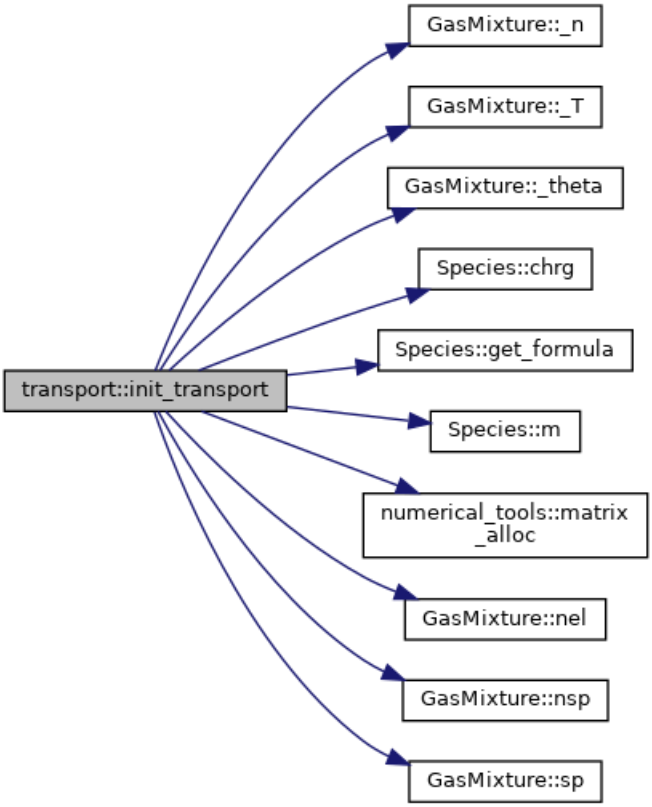
Here is the caller graph for this function:

void transport::init_transport ()

initialize transport object data structure

Definition at line 37 of file transport.cpp.

Here is the call graph for this function:



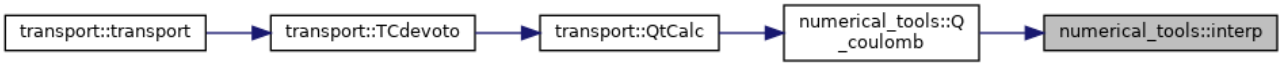
Here is the caller graph for this function:

double numerical_tools::interp (double * *x*, double * *y*, double *xx*, int *N_xy*) [inherited]

interpolator

Definition at line 742 of file numerical_tools.cpp.

Here is the caller graph for this function:

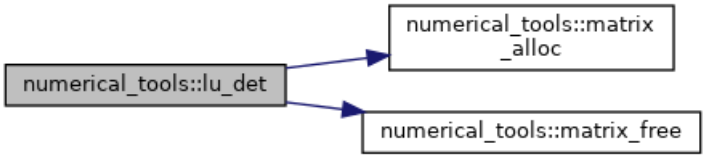


double numerical_tools::lu_det (double ** *A*, int *N*) `[inherited]`

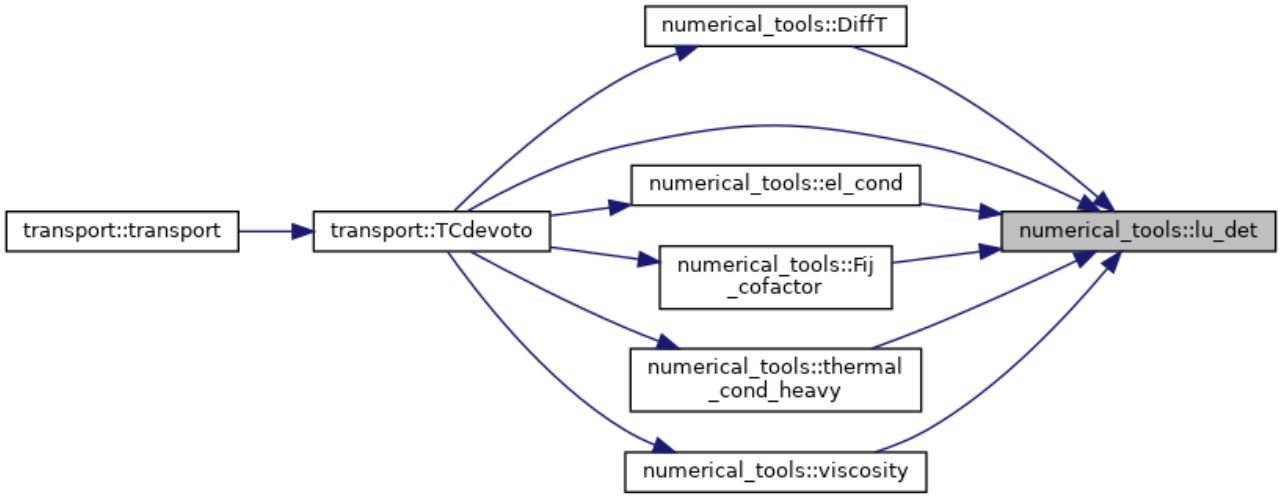
compute det(**A) with LU scomposition

Definition at line 372 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

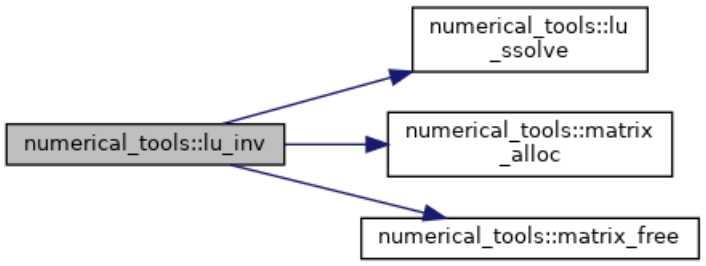


void numerical_tools::lu_inv (double ** *AINV*, double ** *A*, int *N*) `[inherited]`

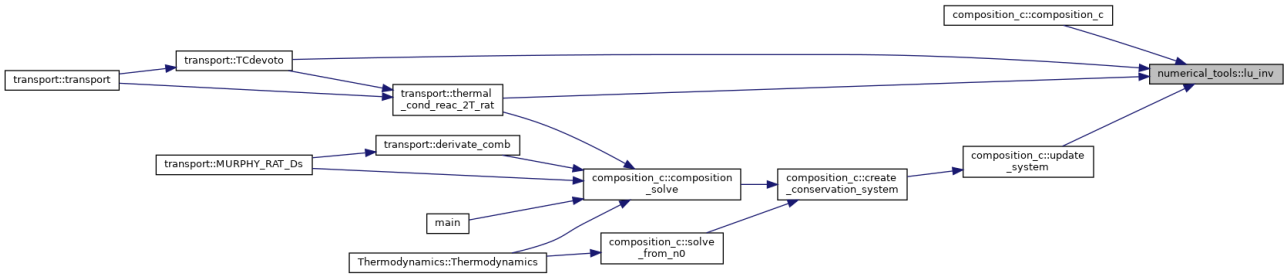
use lu algorithm to compute the inverse matrix

Definition at line 167 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

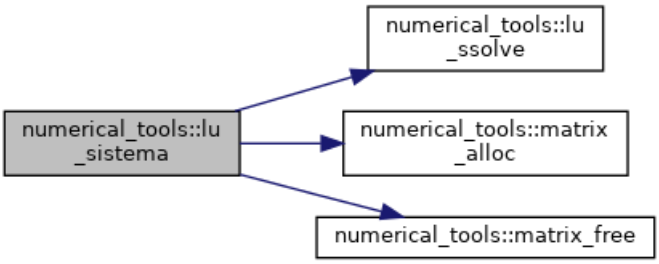


void numerical_tools::lu_sistema (double * *x*, double ** *A*, double * *b*, int *N*) `[inherited]`

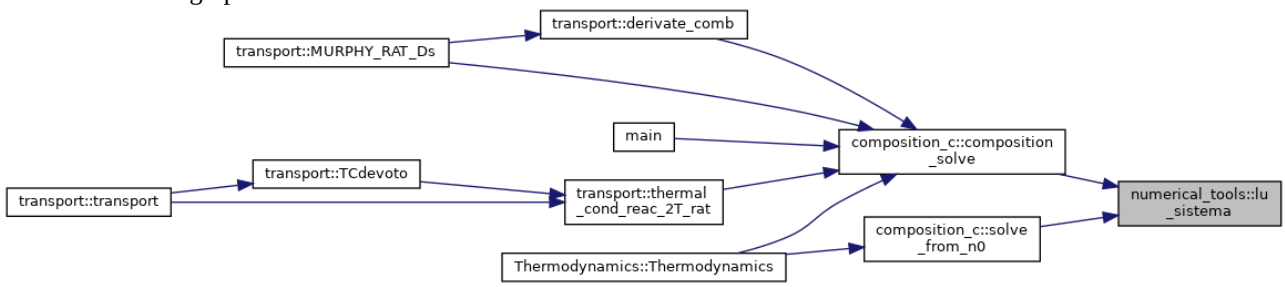
solve an LU system

Definition at line 321 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

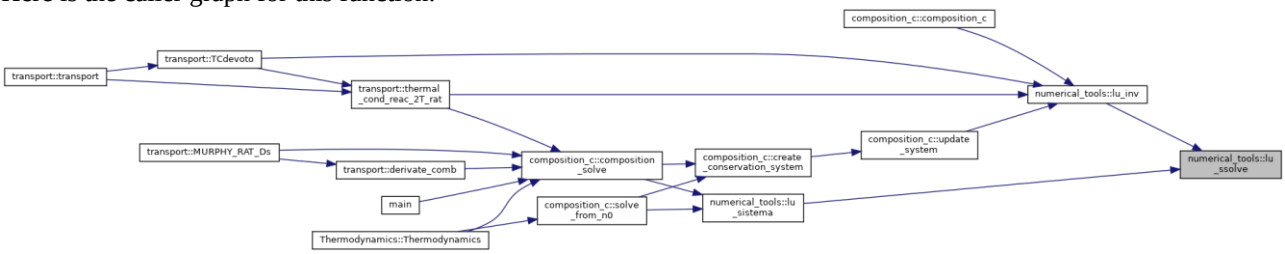


void numerical_tools::lu_ssolve (double * *x*, int * *p*, double * *b0*, double ** *m*, int *N*) `[inherited]`

solve lu decomposition of a matrix

Definition at line 122 of file numerical_tools.cpp.

Here is the caller graph for this function:

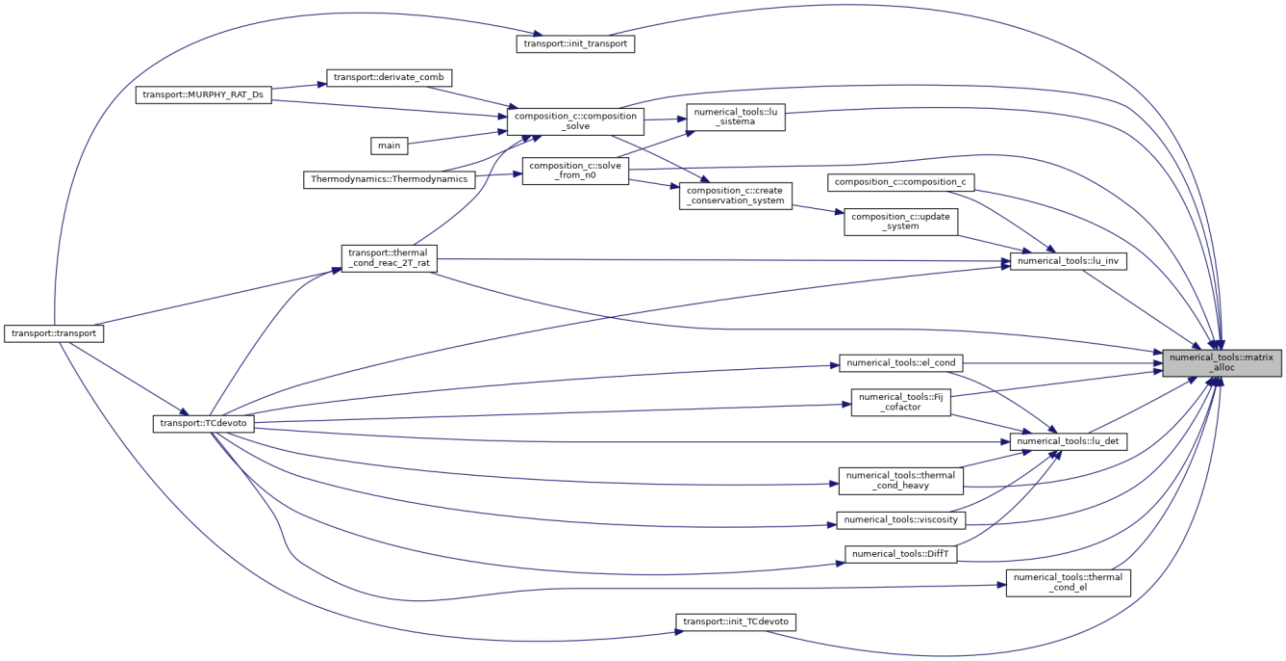


double ** numerical_tools::matrix_alloc (int *N*, int *M*) `[inherited]`

initialize size of **A objects

Definition at line 84 of file numerical_tools.cpp.

Here is the caller graph for this function:

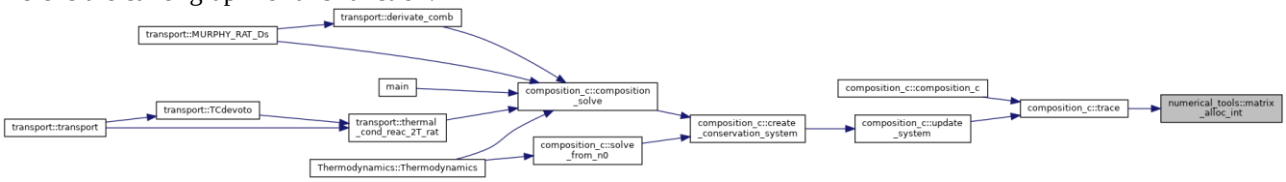


int ** numerical_tools::matrix_alloc_int (int *n*, int *m*) [inherited]

same as matrix_alloc but int variables

Definition at line 97 of file numerical_tools.cpp.

Here is the caller graph for this function:

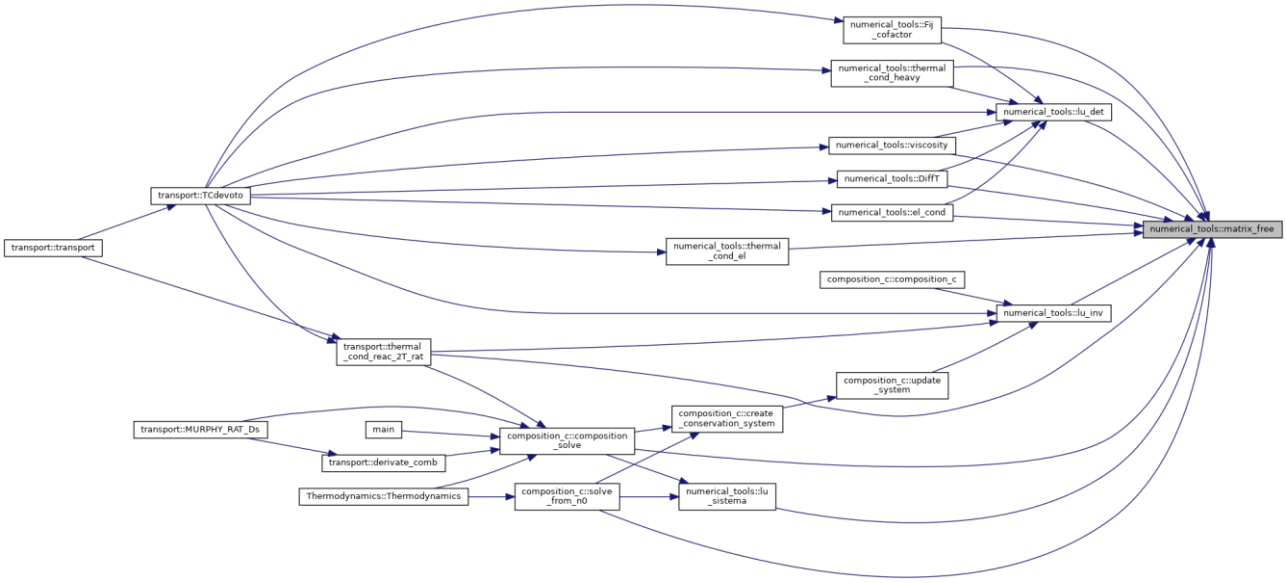


void numerical_tools::matrix_free (double ** *matrix*, int *n*) [inherited]

clears a matrix

Definition at line 111 of file numerical_tools.cpp.

Here is the caller graph for this function:

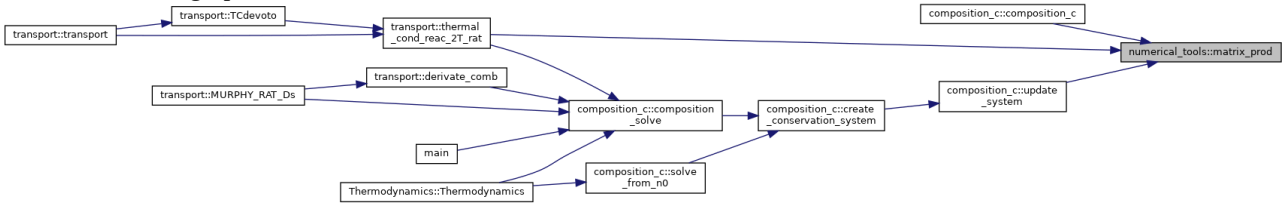


void numerical_tools::matrix_prod (double ** *A*, double ** *B*, double ** *C*, int *N1*, int *N2*) `[inherited]`

compute matrix product A = B*C

Definition at line 63 of file numerical_tools.cpp.

Here is the caller graph for this function:

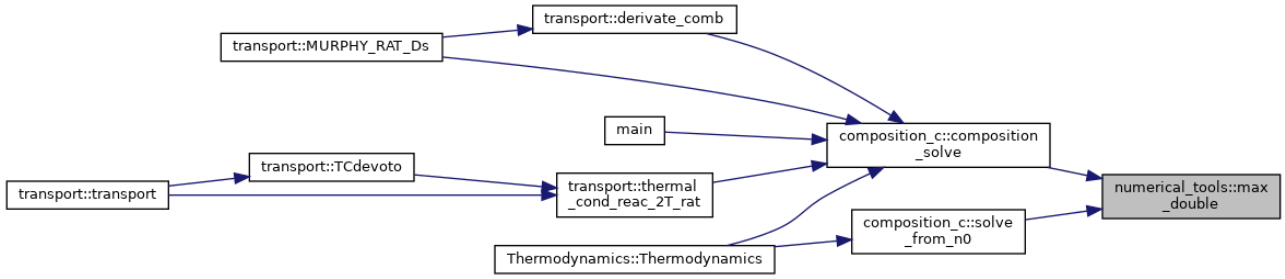


double numerical_tools::max_double (double * *arr*, int *N*) `[inherited]`

return maximum element of an array

Definition at line 432 of file numerical_tools.cpp.

Here is the caller graph for this function:

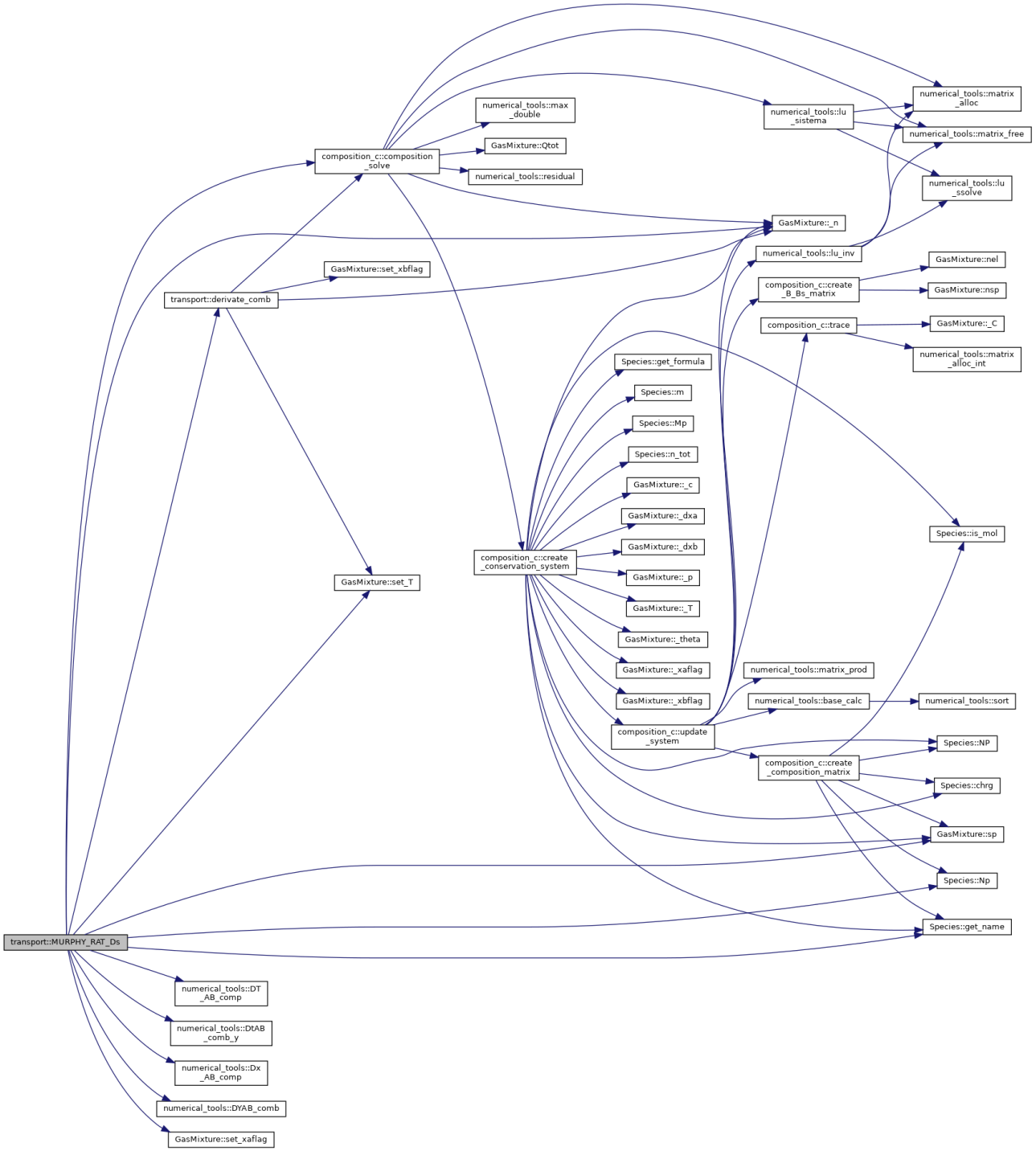


void transport::MURPHY_RAT_Ds ()

< combined thermal diffusion coefficient MURPHY-RAT

Definition at line 468 of file transport.cpp.

Here is the call graph for this function:

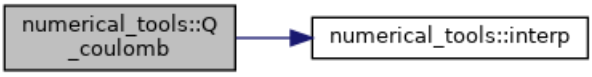


double numerical_tools::Q_coulomb (double ** *Qc*, double *TT*, double *T*, double *theta*, double * *n*, int * *Z*, int *mp*, int *i*, int *j*, int *N_SPC*) `[inherited]`

generates Coulomb collisions integral

Definition at line 683 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

double numerical_tools::Q_gen (double *TT*, double *T*, double *theta*, double * *n*, int * *Z*, int *mp*, int *i*, int *j*, int *N_SPC*) `[inherited]`

generates firsts collision integrals

Definition at line 553 of file numerical_tools.cpp.

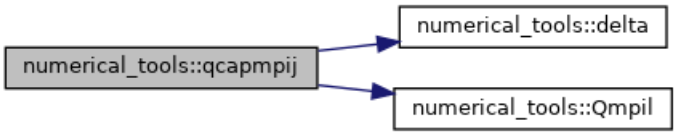
Here is the caller graph for this function:

double numerical_tools::qcapmpij (double ** *Qt*, double * *n*, double * *mass*, int *N_SPC*, int *m*, int *p*, int *i*, int *j*) `[inherited]`

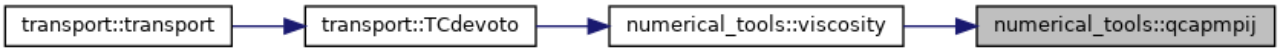
elementi per calcolo bracket integrals da teoria DEVOTO per viscosità

Definition at line 1580 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



double numerical_tools::Qeh (double ** *Qt*, double *T*, double *theta*, double * *mass*, double * *n*, int *N_SPC*) `[inherited]`

thermal exchange electron-heavy

Definition at line 1729 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

int numerical_tools::Qij_calc (int *il*, int * *Z*, int *N_specs*) `[inherited]`

associates to il the Qij coulomb interaction progressively numerated when Z[i]*Z[j]==0

Definition at line 522 of file numerical_tools.cpp.

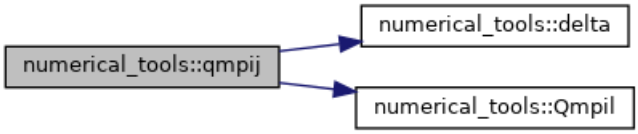
Here is the caller graph for this function:

double numerical_tools::qmpij (double ** *Qt*, double * *n*, double * *mass*, int *N_SPC*, int *m*, int *p*, int *i*, int *j*) `[inherited]`

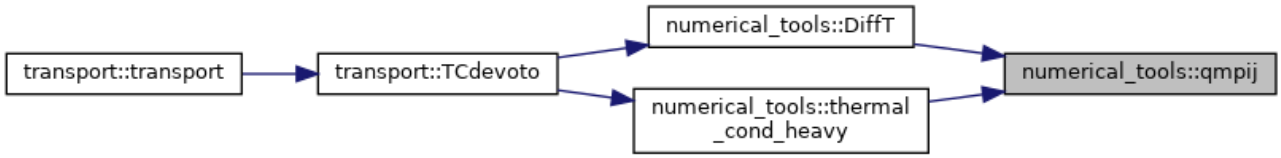
DEVOTO Appendix 06/1966.

Definition at line 1063 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

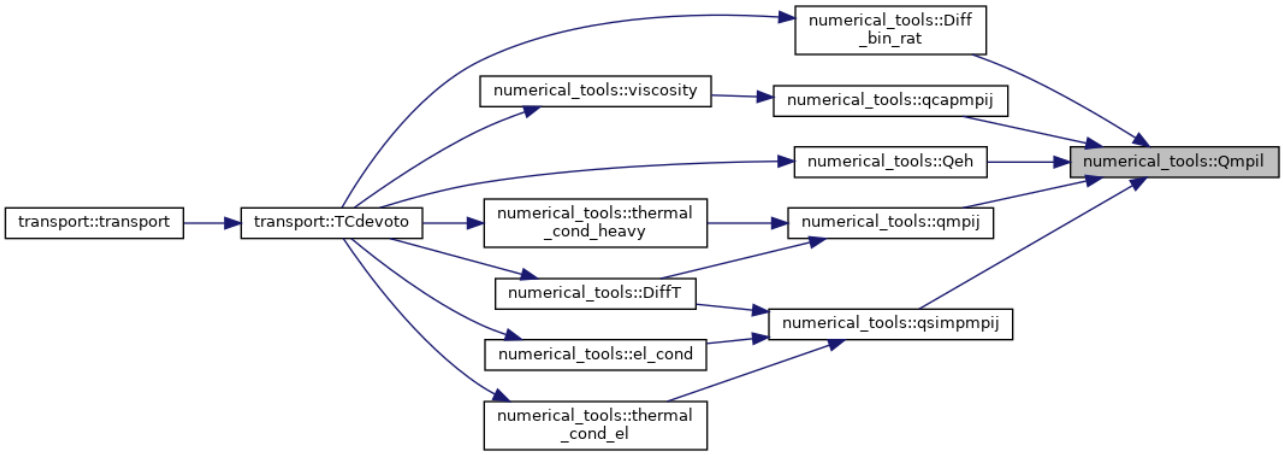


double numerical_tools::Qmpil (double ** *Qt*, int *m*, int *p*, int *i*, int *l*, int *N_SPC*) `[inherited]`

extract Qt element for every couple interaction i l

Definition at line 801 of file numerical_tools.cpp.

Here is the caller graph for this function:



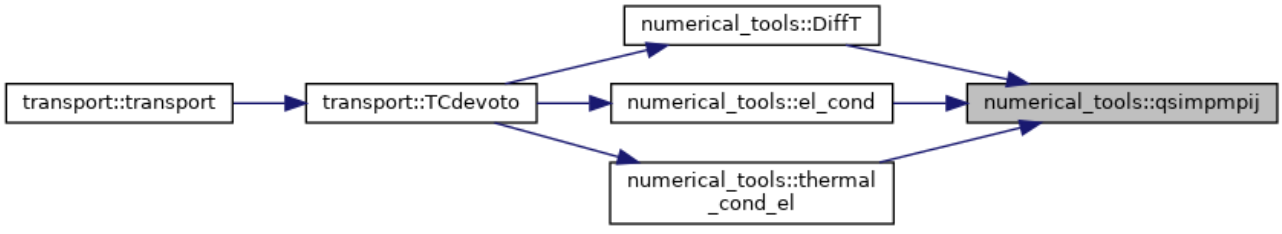
double numerical_tools::qsimpmpij (double ** *Qt*, double * *n*, int *N_SPC*, int *m*, int *p*) `[inherited]`

simplified Devoto 08/08/1966 Appendix, assumed electron as last specie

Definition at line 900 of file numerical_tools.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

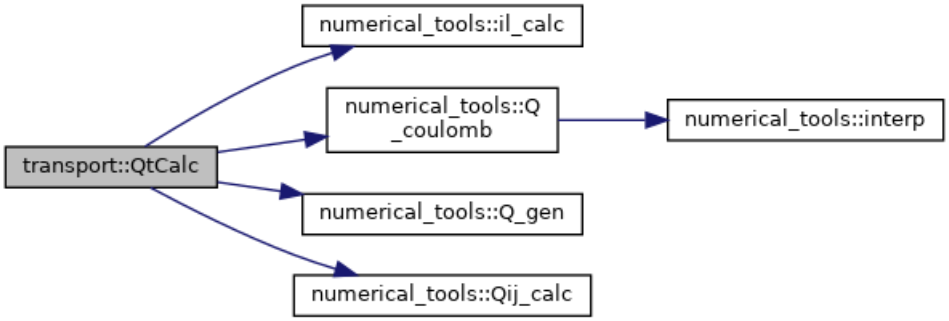


void transport::QtCalc ()

Compute Qt data structure with Q & Qc data DEVOTO eqns. 5-6.

Definition at line 147 of file transport.cpp.

Here is the call graph for this function:



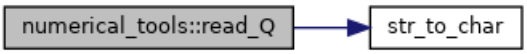
Here is the caller graph for this function:

void numerical_tools::read_Q (double * *Q*, int *N_NC*, int *N_C*, int *N_TEMP*, std::vector< std::string > *_filenames*) `[inherited]`**

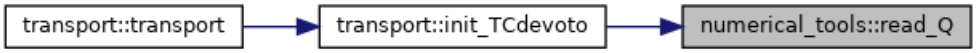
reads cross sections data /* DA GENERALIZZARE */

Definition at line 446 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

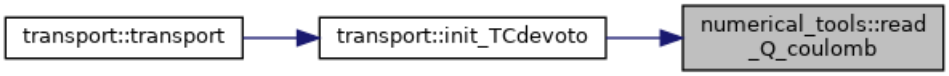


void numerical_tools::read_Q_coulomb (double ** *Qc*, std::string *path*) `[inherited]`

reads Coullomb cross sections data /* DA GENERALIZZARE */

Definition at line 480 of file numerical_tools.cpp.

Here is the caller graph for this function:

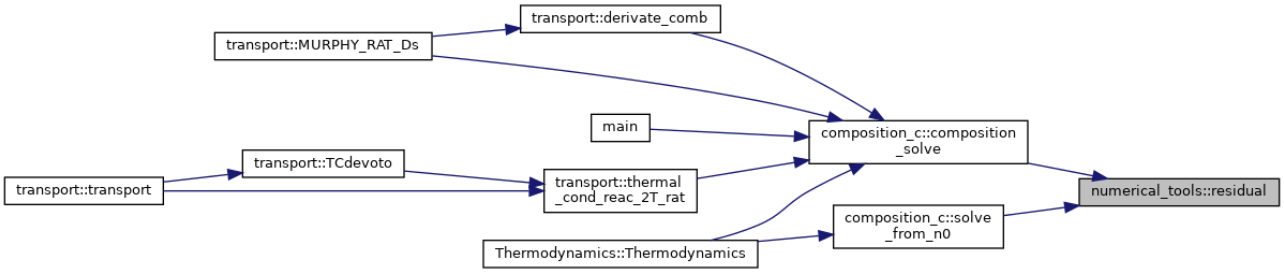


void numerical_tools::residual (double * *R*, double ** *J*, double * *n*, double ** *A*, double * *A0*, double ** *v*, int * *b*, int * *bs*, int *N*, int *M*) `[inherited]`

compute residuals as in GODIN eq.35

Definition at line 282 of file numerical_tools.cpp.

Here is the caller graph for this function:

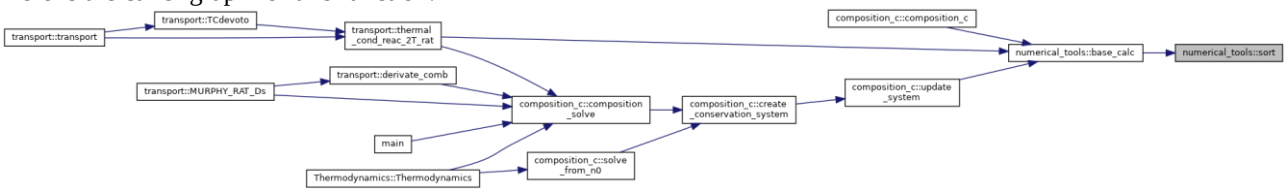


void numerical_tools::sort (double * *arr*, int * *perm*, int *n*) `[inherited]`

indices of arr elements in descent order initialize a vector whose component are

Definition at line 37 of file numerical_tools.cpp.

Here is the caller graph for this function:

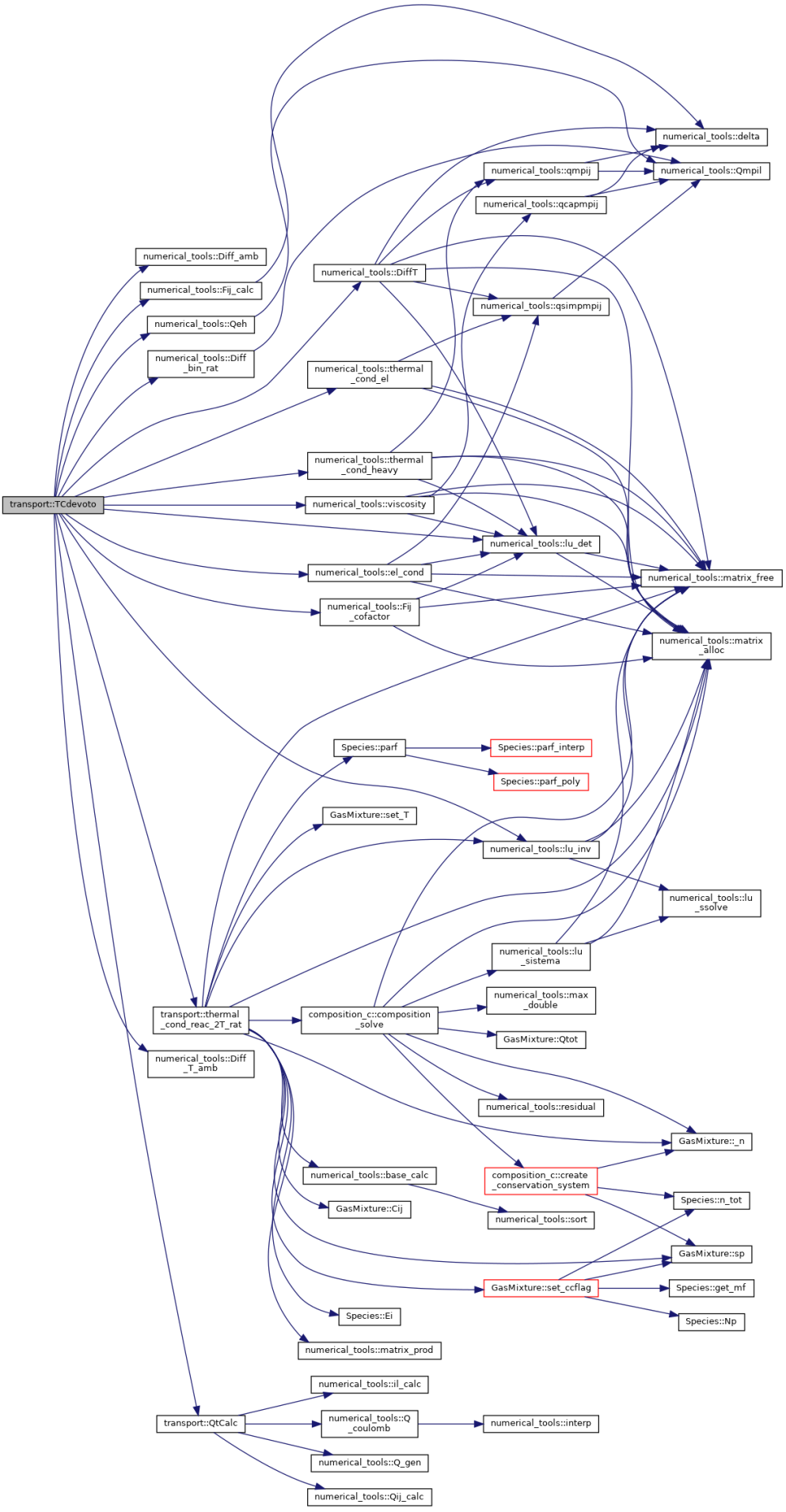


void transport::TCdevoto ()

compute tranport properties by Devoto algorithm described in Transport Properties of Ionized Monoatomic Gases R.S.Devoto June 1966 Simplified Expressions for the Transport Properties of Ionized Monoatomic Gases R.S. Devoto October 1967

Definition at line 197 of file transport.cpp.

Here is the call graph for this function:


transport::TCdevoto
numerical_tools::Diff_amb
numerical_tools::Fij_calc
numerical_tools::Qeh
numerical_tools::Diff_bin_rat
numerical_tools::delta
numerical_tools::qmpij
numerical_tools::Qmpil
numerical_tools::qcapmpij
numerical_tools::DiffT
numerical_tools::qsimpmpij
numerical_tools::thermal_cond_el
numerical_tools::thermal_cond_heavy
numerical_tools::viscosity
numerical_tools::lu_det
numerical_tools::el_cond
numerical_tools::matrix_free
numerical_tools::Fij_cofactor
numerical_tools::matrix_alloc
Species::parf
Species::parf_interp
Species::parf_poly
GasMixture::set_T
numerical_tools::lu_inv
numerical_tools::lu_ssolve
numerical_tools::lu_sistema
numerical_tools::max_double
transport::thermal_cond_reac_2T_rat
composition_c::composition_solve
GasMixture::Qtot
numerical_tools::Diff_T_amb
numerical_tools::residual
GasMixture::_n
numerical_tools::base_calc
composition_c::create_conservation_system
Species::n_tot
GasMixture::Cij
numerical_tools::sort
GasMixture::sp
GasMixture::set_ccflag
Species::get_mf
Species::Ei
Species::Np
numerical_tools::matrix_prod
numerical_tools::il_calc
transport::QtCalc
numerical_tools::Q_coulomb
numerical_tools::interp
numerical_tools::Q_gen
numerical_tools::Qij_calc

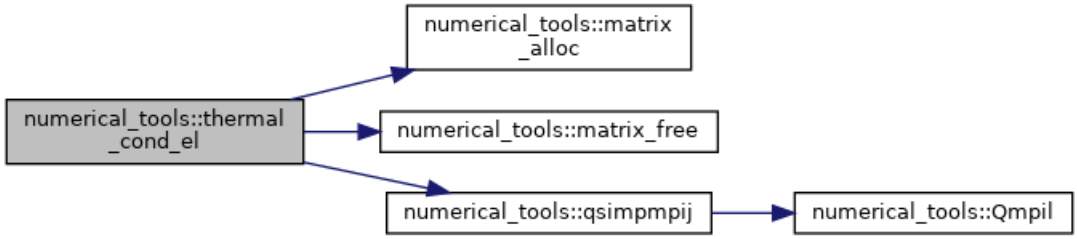
Here is the caller graph for this function:

double numerical_tools::thermal_cond_el (double ** *Qt*, double *Te*, double * *n*, double * *mass*, int *N_SPC*) `[inherited]`

electron thermal conductivity simplified DEVOTO eq.21

Definition at line 1492 of file numerical_tools.cpp.

Here is the call graph for this function:



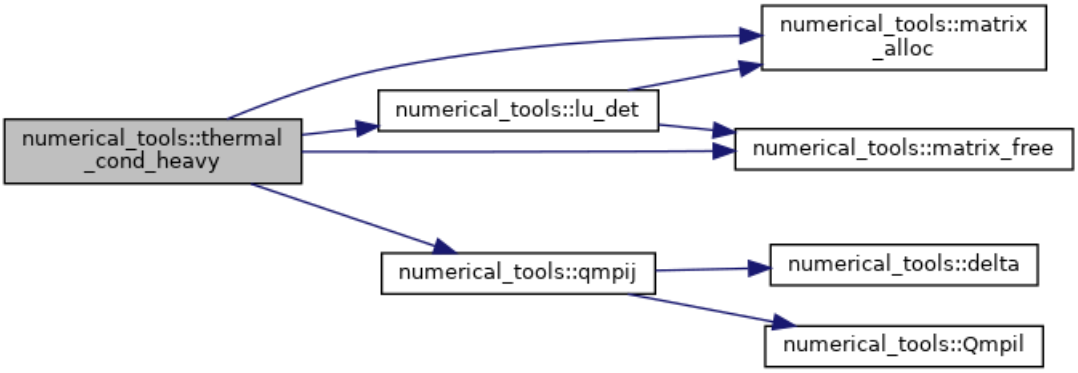
Here is the caller graph for this function:

double numerical_tools::thermal_cond_heavy (double ** *Qt*, double *T*, double * *n*, double * *mass*, int *N_SPC*) `[inherited]`

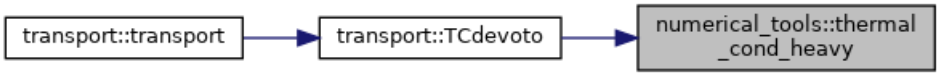
heavy thermal conductivity 2ND order approx

Definition at line 1519 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



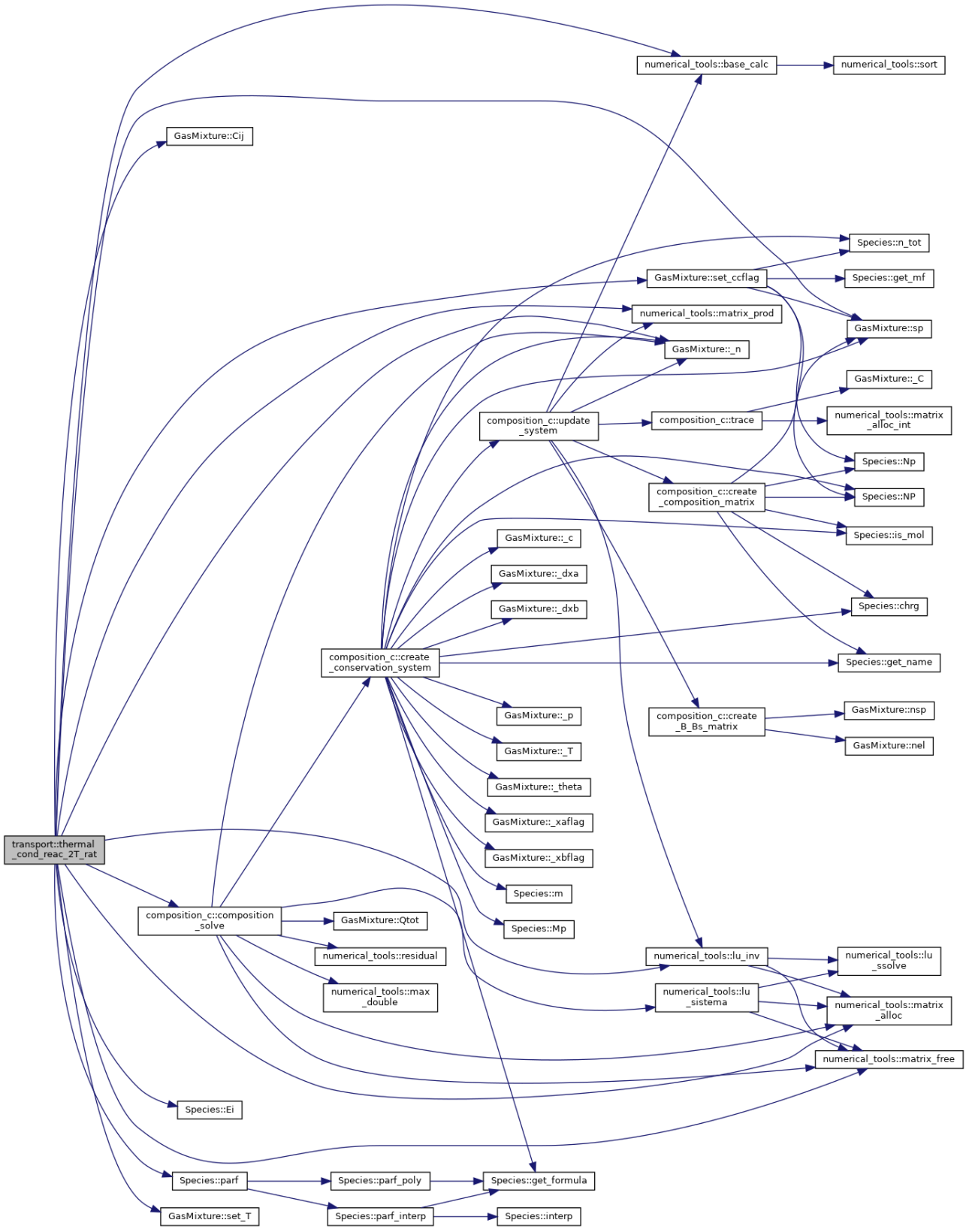
void transport::thermal_cond_reac_2T_rat ()

reactive thermal conductivity

finally reactive conductivity

Definition at line 290 of file transport.cpp.

Here is the call graph for this function:



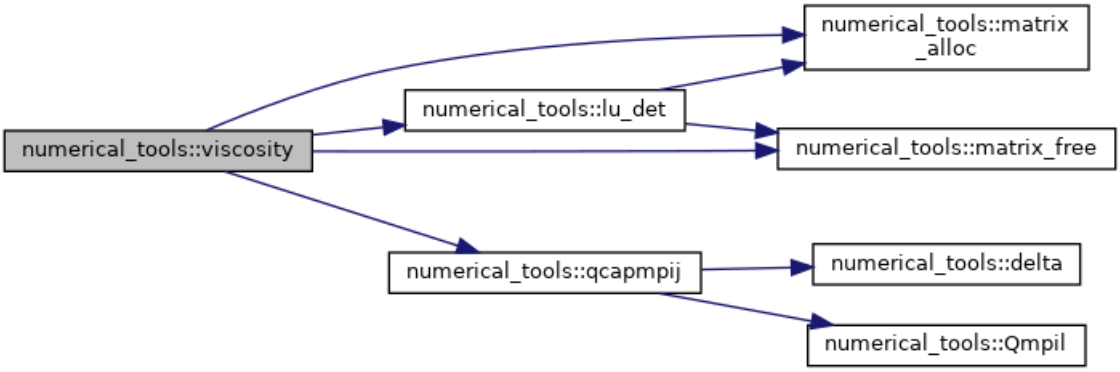
Here is the caller graph for this function:

double numerical_tools::viscosity (double ** *Qt*, double *T*, double * *n*, double * *mass*, int *N_SPC*) [inherited]

funzione calcolo viscosita ///1Approx, to further (2nd available) modify M_ORDV

Definition at line 1645 of file numerical_tools.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

Member Data Documentation

int transport::C_interaction [private]

Coulomb collision.

Definition at line 58 of file transport.h.

double transport::D [private]**

Fij factors.

diffusion coefficients

Definition at line 36 of file transport.h.

double transport::D_amb [private]**

ambipolar diffusion coefficients

Definition at line 38 of file transport.h.

double transport::Dbin [private]**

binary diffusion coefficients RAT et.al.30Apr2002

Definition at line 29 of file transport.h.

std::filesystem::path transport::DevotoGodin_path [private]

Definition at line 80 of file transport.h.

double* transport::DT [private]

matrix to compute thermal reactive conductivity
Thermal diffusion DEVOTO eq.9
Definition at line 42 of file transport.h.

double transport::DTAB [private]

combined thermal diffusion coefficient MURPHY-RAT
Definition at line 74 of file transport.h.

double transport::DTABy [private]

combined diffusion coefficient UNIBO 09 fluent
Definition at line 75 of file transport.h.

double* transport::DTamb [private]

Ambipolar Thermal diffusion.
Definition at line 44 of file transport.h.

double transport::DxAB [private]

2T combined diffusion coefficient due to gradients in concentration MURPHY-RAT 2.36
Definition at line 72 of file transport.h.

double * transport::DxDT [private]

combined diffusion partial derivatives
Definition at line 70 of file transport.h.

double* transport::DxDxb [private]

Definition at line 70 of file transport.h.

double transport::DYAB [private]

diffusion coefficient from UNIBO 08
Definition at line 73 of file transport.h.

double transport::E [private]**

Definition at line 40 of file transport.h.

double ** transport::Einv [private]

Definition at line 40 of file transport.h.

double transport::Fij** `[private]`

Definition at line 34 of file transport.h.

double ** transport::Fij_co `[private]`

Definition at line 34 of file transport.h.

std::vector<std::string> transport::file_names `[private]`

Definition at line 78 of file transport.h.

std::string transport::folder `[private]`

Definition at line 82 of file transport.h.

std::vector<std::string> transport::formulae `[private]`

Definition at line 21 of file transport.h.

GasMixture* transport::gas `[private]`

local gas pointer
Definition at line 18 of file transport.h.

double transport::kB `[private]`

to compute transport properties K-boltzmann is in fJ/K° units
Definition at line 60 of file transport.h.

int transport::M `[private]`

Definition at line 19 of file transport.h.

double* transport::m_ `[private]`

vector of species masses [g]
Definition at line 50 of file transport.h.

int transport::N `[private]`

Definition at line 19 of file transport.h.

double* transport::n_ `[private]`

local particle densities copies
Definition at line 48 of file transport.h.

int transport::N_4TH = 4 [private]

level of approximation desired
Definition at line 68 of file transport.h.

int transport::N_BINARY [private]

number of binary collisions
Definition at line 66 of file transport.h.

int transport::N_TEMP = 448 [private]

Temperature interpolation steps.
Definition at line 64 of file transport.h.

int transport::NC_interaction [private]

Non-Coulomb collisions.
Definition at line 62 of file transport.h.

double transport::nn [private]

total particle density
Definition at line 54 of file transport.h.

std::string transport::prefix {"Q4th_"} [private]

Definition at line 86 of file transport.h.

double* transport::Q [private]**

cross section data
Definition at line 23 of file transport.h.

double transport::Qc [private]**

Coulomb cross section data.
Definition at line 25 of file transport.h.

double transport::Qt [private]**

Devoto matrix as in DEVOTO eq.7.
Definition at line 27 of file transport.h.

double transport::rho_ [private]

density

Definition at line 56 of file transport.h.

std::string transport::separator {"_"} [private]

Definition at line 88 of file transport.h.

std::string transport::subfolder {"/CI_database/"} [private]

Definition at line 84 of file transport.h.

std::string transport::suffix {".txt"} [private]

Definition at line 90 of file transport.h.

double transport::T [private]

Definition at line 20 of file transport.h.

double* transport::TC [private]

12 tranport properties

Definition at line 46 of file transport.h.

double transport::Theta [private]

Definition at line 20 of file transport.h.

int* transport::Z [private]

vector of particles charges

Definition at line 52 of file transport.h.

The documentation for this class was generated from the following files:

- transport/**transport.h**
- transport/**transport.cpp**

WallClock Class Reference

`#include <clock.h>`

Public Member Functions

- **WallClock** ()
  set start time point

- void **start** ()
  set stop time point

- void **stop** ()
  evaluate interval between start and stop in seconds

- double **interval** ()

Private Attributes

- std::chrono::high_resolution_clock::time_point **t1**
- std::chrono::high_resolution_clock::time_point **t2**

Detailed Description

Definition at line 27 of file clock.h.

Constructor & Destructor Documentation

WallClock::WallClock ()

set start time point

Definition at line 25 of file clock.cpp.

Member Function Documentation

double WallClock::interval ()

Definition at line 37 of file clock.cpp.

Here is the caller graph for this function:

void WallClock::start ()

set stop time point

Definition at line 28 of file clock.cpp.

Here is the caller graph for this function:

void WallClock::stop ()

evaluate interval between start and stop in seconds

Definition at line 33 of file clock.cpp.

Here is the caller graph for this function:

Member Data Documentation

std::chrono::high_resolution_clock::time_point WallClock::t1 `[private]`

Definition at line 29 of file clock.h.

std::chrono::high_resolution_clock::time_point WallClock::t2 `[private]`

Definition at line 29 of file clock.h.

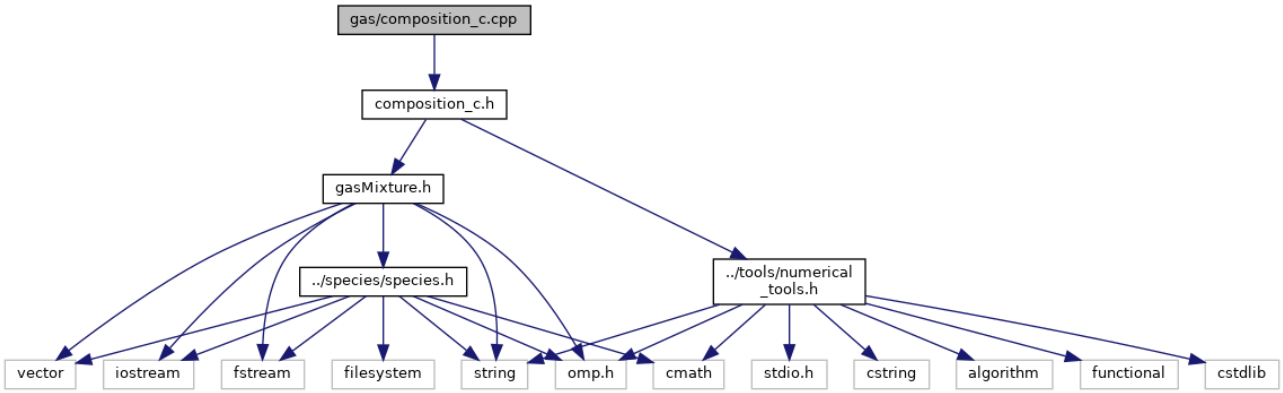
The documentation for this class was generated from the following files:

- tools/**clock.h**
- tools/**clock.cpp**

gas/composition_c.cpp File Reference

`#include "composition_c.h"`

Include dependency graph for composition_c.cpp:

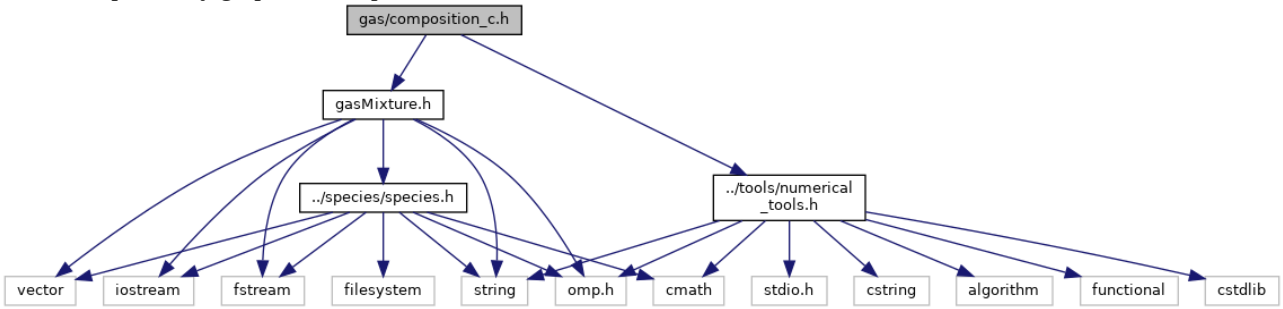


gas/composition_c.h File Reference

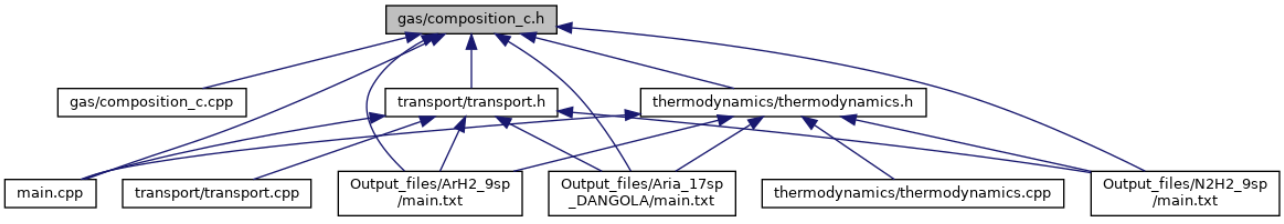
`#include "gasMixture.h"`

`#include "../tools/numerical_tools.h"`

Include dependency graph for composition_c.h:



This graph shows which files directly or indirectly include this file:



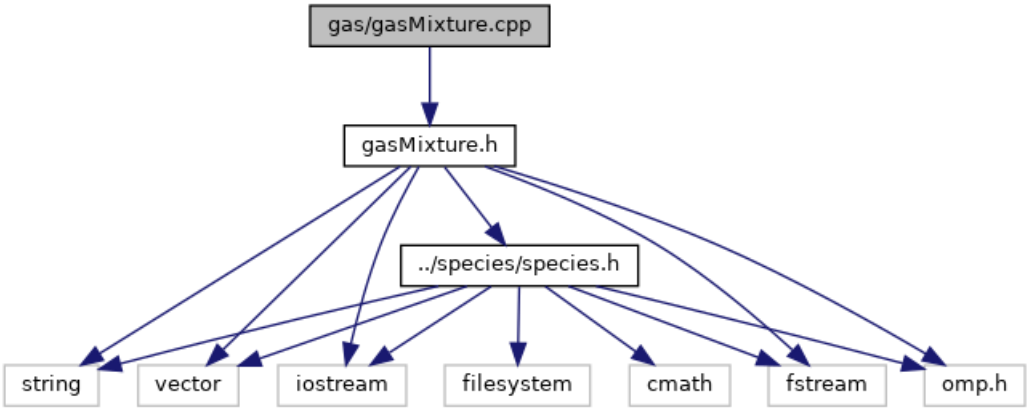
Classes

- class **composition_c**

gas/gasMixture.cpp File Reference

`#include "gasMixture.h"`

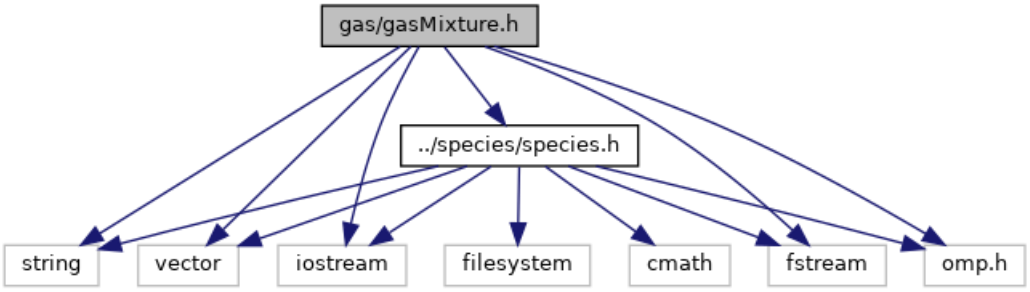
Include dependency graph for gasMixture.cpp:



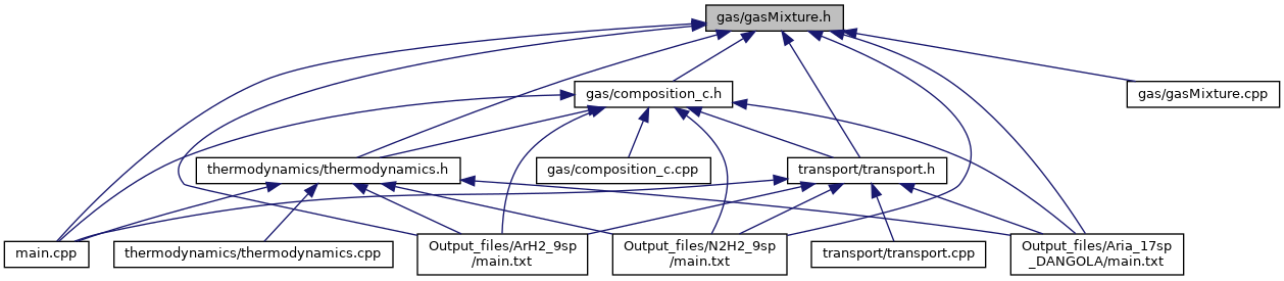
gas/gasMixture.h File Reference

```
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <omp.h>
#include "../species/species.h"
```

Include dependency graph for gasMixture.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **GasMixture**

Macros

- #define **KB** 1.380650524e-23
- #define **PI** 3.1415926535
- #define **HP** 6.62606896e-34

Macro Definition Documentation

#define HP 6.62606896e-34

Definition at line 19 of file gasMixture.h.

#define KB 1.380650524e-23

Gasmixture object will hold the gas parameters, list of species and output variables to be computed with classes basically the body of the simulation

Definition at line 17 of file gasMixture.h.

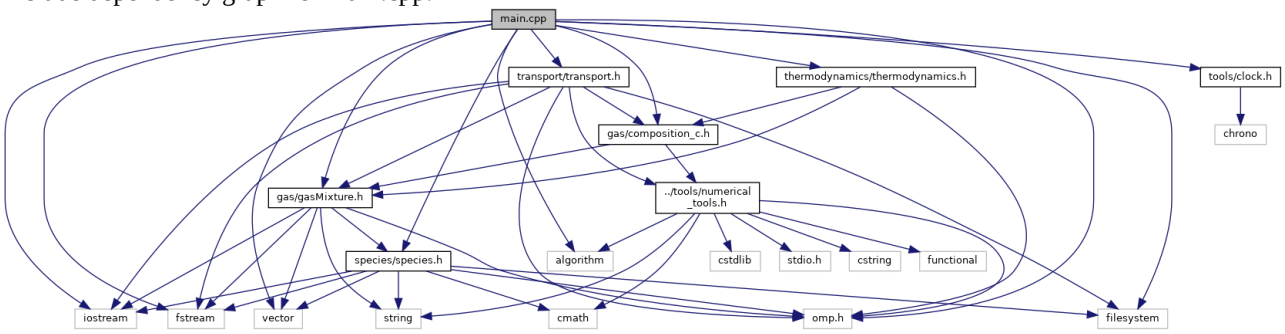
#define PI 3.1415926535

Definition at line 18 of file gasMixture.h.

main.cpp File Reference

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <fstream>
#include <filesystem>
#include <omp.h>
#include "species/species.h"
#include "gas/gasMixture.h"
#include "gas/composition_c.h"
#include "thermodynamics/thermodynamics.h"
#include "transport/transport.h"
#include "tools/clock.h"
```

Include dependency graph for main.cpp:



Functions

- std::string **output_path** ()
  initialize path to write data

- int **main** ()

Function Documentation

int main (void)

Declare species in the mixture:

Species species_name("formula", bool ifbase); **Species** species_name("formula", double set_mole_fraction);

set bases

WARNING:

>>bases have to be set as the gases of the mixture in the ground state

- electron

>>there must be M-2 mole_fraction relations where M are number elements electron included

M-2 abundancy relations âˆˆ [0 1]

output files

initialize output files

open file

set k-th percentage abundancy ALPHABETICALLY starting from 2nd specie (see GODIN eq.24 && **GasMixture.cpp**)

create a vector of species WARNING: to loop on concentration this has to be built after set_mf data inside conc loop

create **GasMixture** object */ type **GasMixture** name_of_the_object(vector<Species>,pressure,temperature,non-eq parameter)

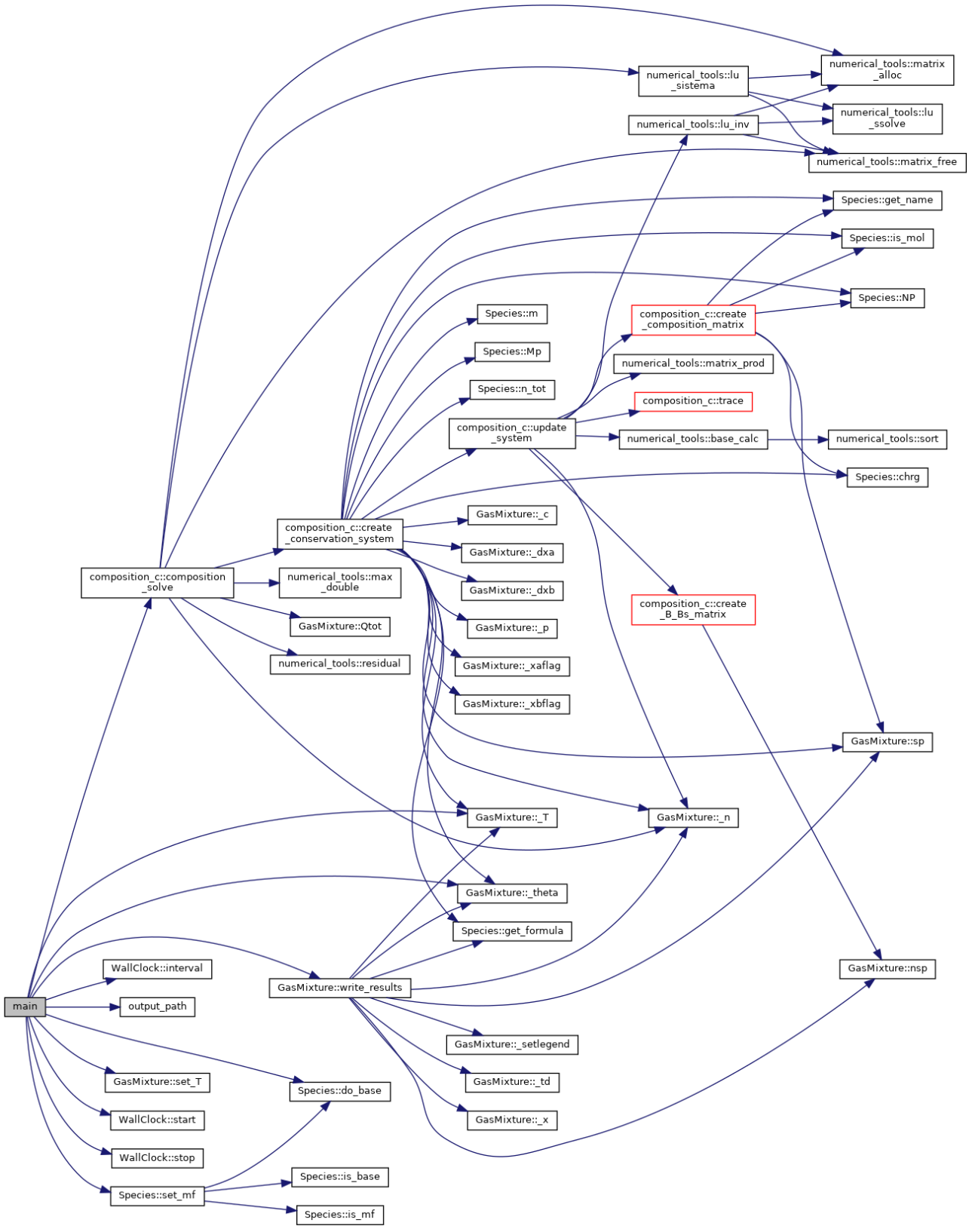
loop on T

compute data

write data see **gasmixture.cpp** to set results to be printed

Definition at line 33 of file main.cpp.

Here is the call graph for this function:



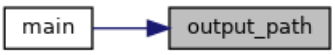
std::string output_path ()

initialize path to write data

!!change folder!!

Definition at line 18 of file main.cpp.

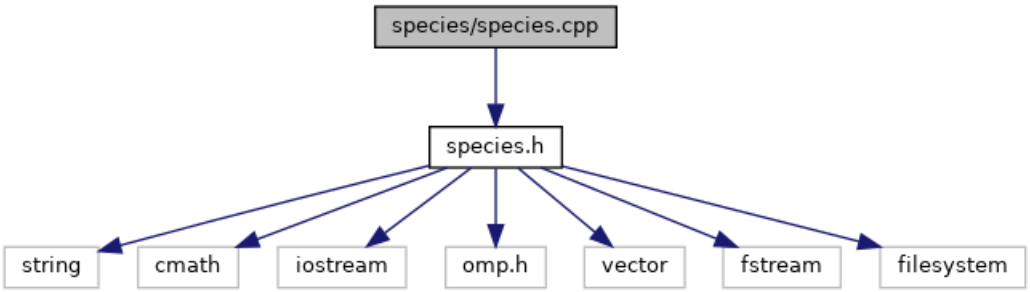
Here is the caller graph for this function:



species/species.cpp File Reference

```
#include "species.h"
```

Include dependency graph for species.cpp:



Functions

- double ** **matrix_alloc** (int N, int M)

Function Documentation

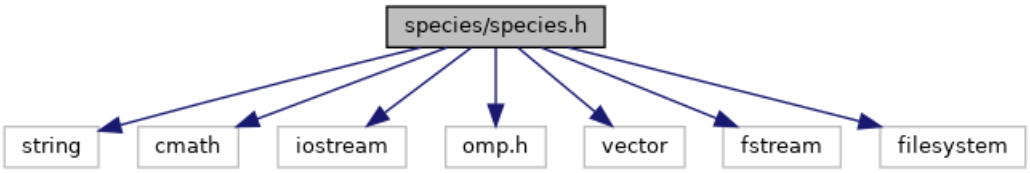
double** matrix_alloc (int *N*, int *M*)

Definition at line 644 of file species.cpp.

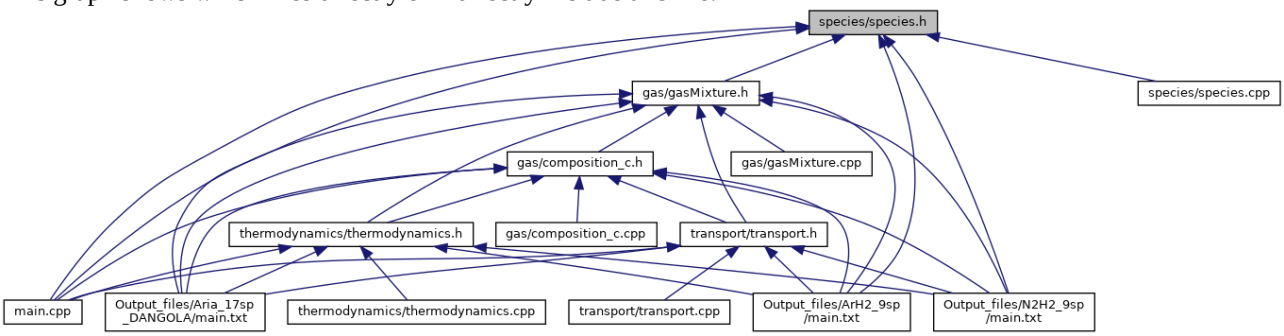
species/species.h File Reference

```
#include <string>
#include <cmath>
#include <iostream>
#include <omp.h>
#include <vector>
#include <fstream>
#include <filesystem>
```

Include dependency graph for species.h:



This graph shows which files directly or indirectly include this file:



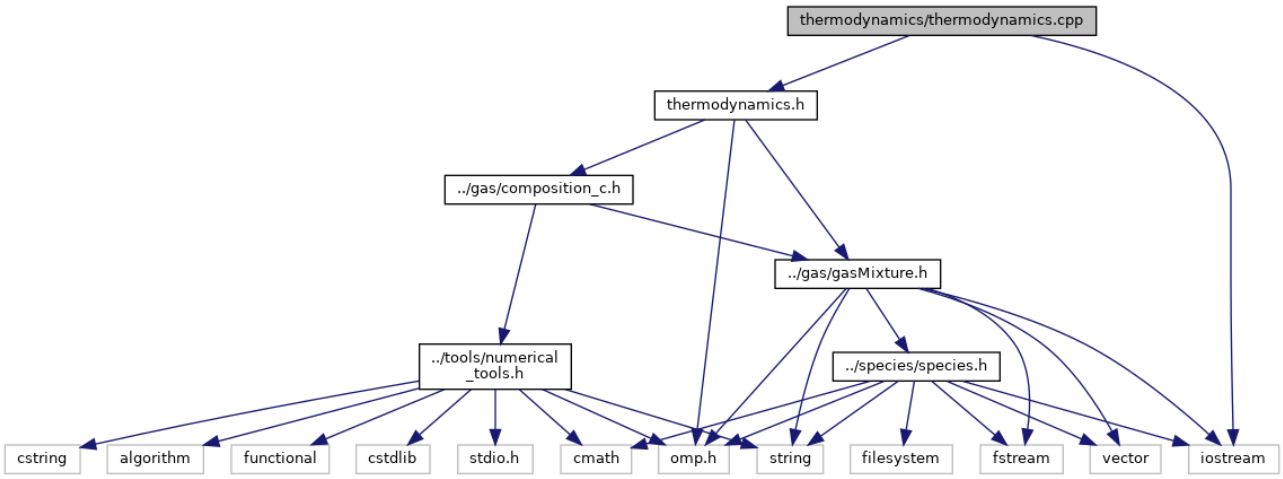
Classes

- class **Species**

thermodynamics/thermodynamics.cpp File Reference

```
#include "thermodynamics.h"
#include <iostream>
```

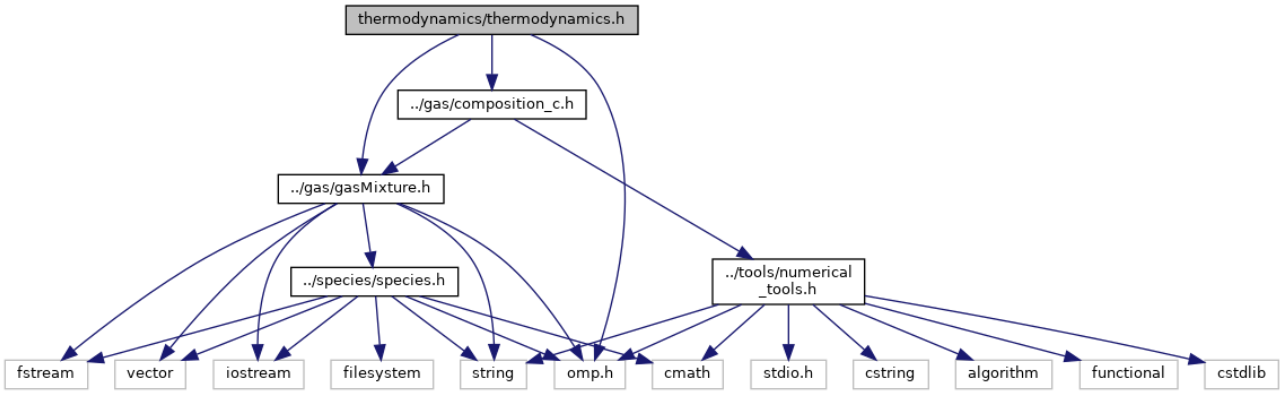
Include dependency graph for thermodynamics.cpp:



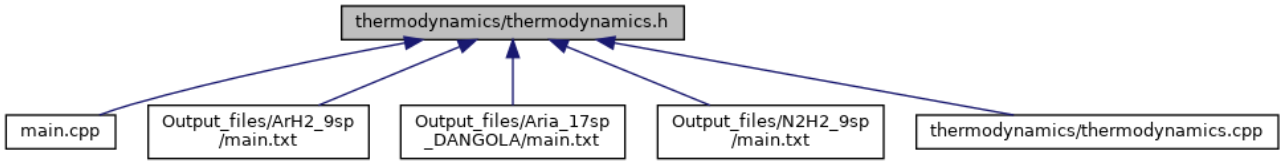
thermodynamics/thermodynamics.h File Reference

```
#include <omp.h>
#include "../gas/gasMixture.h"
#include "../gas/composition_c.h"
```

Include dependency graph for thermodynamics.h:



This graph shows which files directly or indirectly include this file:



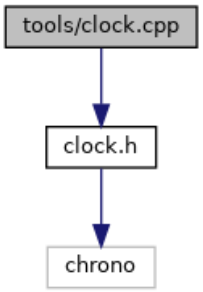
Classes

- class **Thermodynamics**

tools/clock.cpp File Reference

`#include "clock.h"`

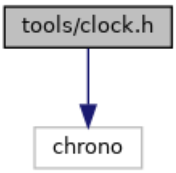
Include dependency graph for clock.cpp:



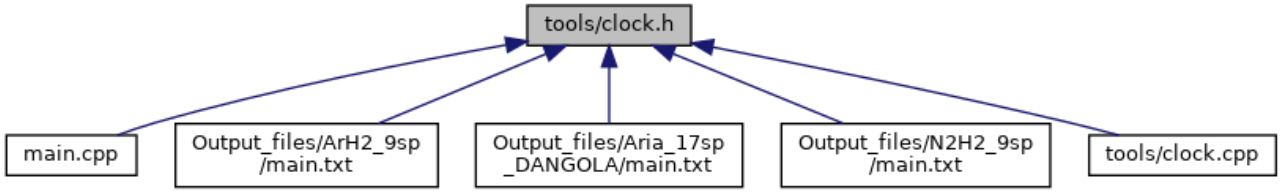
tools/clock.h File Reference

`#include <chrono>`

Include dependency graph for clock.h:



This graph shows which files directly or indirectly include this file:



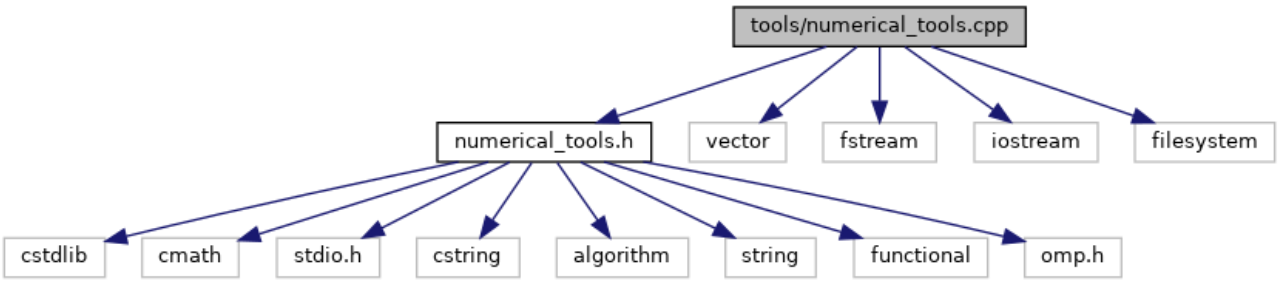
Classes

- class **WallClock**

tools/numerical_tools.cpp File Reference

```
#include "numerical_tools.h"
#include <vector>
#include <fstream>
#include <iostream>
#include <filesystem>
```

Include dependency graph for numerical_tools.cpp:



Macros

- #define **KB** 1.380650524e-23
- #define **kB** 1.380650524e-8
- #define **PI** 3.1415926535
- #define **HP** 6.62606896e-34
- #define **e0** 8.8541878176e-12
- #define **q_e** 1.60217646e-19

Functions

- char * **str_to_char** (std::string &s)

Variables

- std::vector< char * > **file_names_char**

Macro Definition Documentation

#define e0 8.8541878176e-12

Definition at line 20 of file numerical_tools.cpp.

#define HP 6.62606896e-34

Definition at line 17 of file numerical_tools.cpp.

#define KB 1.380650524e-23

Definition at line 7 of file numerical_tools.cpp.

#define kB 1.380650524e-8

Definition at line 11 of file numerical_tools.cpp.

#define PI 3.1415926535

Definition at line 14 of file numerical_tools.cpp.

#define q_e 1.60217646e-19

Definition at line 23 of file numerical_tools.cpp.

Function Documentation

char* str_to_char (std::string & *s*)

Definition at line 27 of file numerical_tools.cpp.

Here is the caller graph for this function:

Variable Documentation

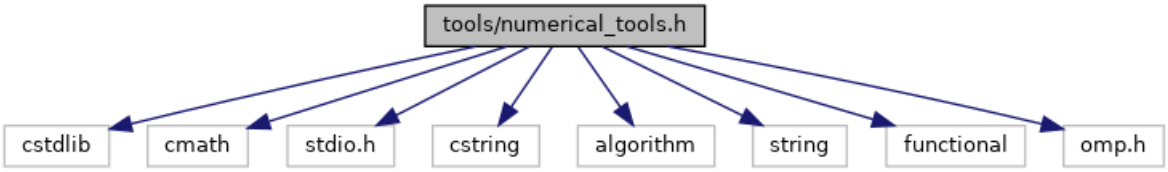
std::vector<char *> file_names_char

Definition at line 33 of file numerical_tools.cpp.

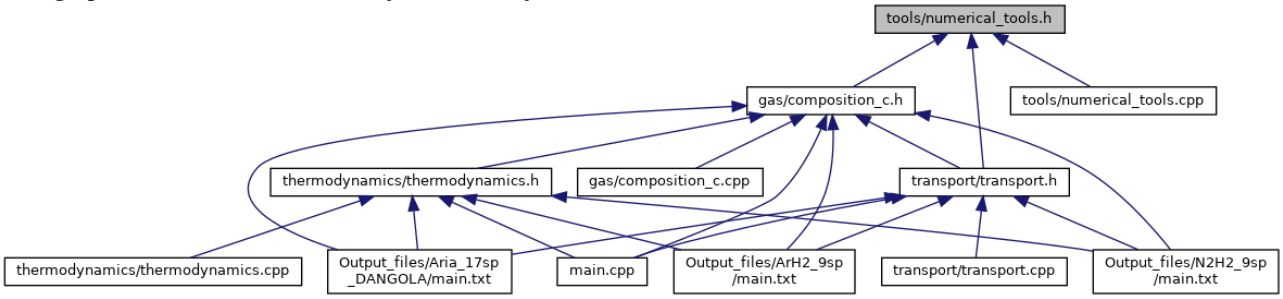
tools/numerical_tools.h File Reference

```
#include <cstdlib>
#include <cmath>
#include <stdio.h>
#include <cstring>
#include <algorithm>
#include <string>
#include <functional>
#include <omp.h>
```

Include dependency graph for numerical_tools.h:



This graph shows which files directly or indirectly include this file:



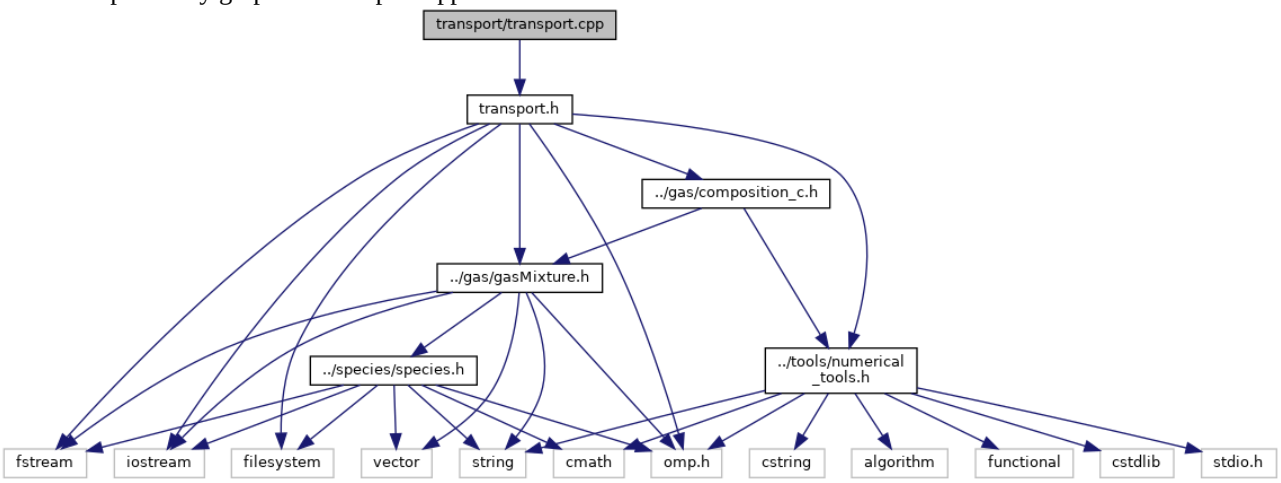
Classes

- class **numerical_tools**

transport/transport.cpp File Reference

```
#include "transport.h"
```

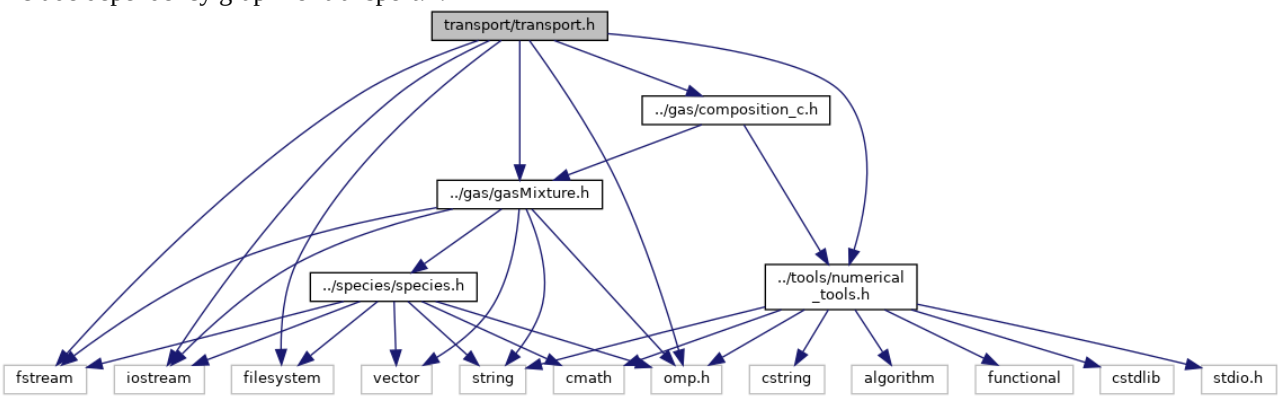
Include dependency graph for transport.cpp:



transport/transport.h File Reference

```
#include <iostream>
#include <fstream>
#include <filesystem>
#include <omp.h>
#include "../gas/gasMixture.h"
#include "../tools/numerical_tools.h"
#include "../gas/composition_c.h"
```

Include dependency graph for transport.h:



This graph shows which files directly or indirectly include this file: