Laboratorio di Fisica Computazionale: Dinamica molecolare

Zhou Zheng

5 dicembre 2020

1 Moduli

Si definiscono tre moduli all'esterno del programma principale:

- `parametri`: contiene i parametri geometrici del problema.
- `list_mod`: contiene la struttura dati delle cellette della scatola e gestisce tutta la struttura a lista.
- `physics`: contiene i risultati fisici della simulazione e i vari sottoprogrammi.

Nel programma principale `DinamicaMolecolare` è presente una matrice $6 \times$ `nAtoms` che contiene le posizioni e le velocità:

```
! Le componenti 1:3 sono le componenti delle posizioni posizioni, le componenti 4:6 sono le
    componenti della velocita'
DOUBLE PRECISION :: atoms(1:6,1:nAtoms)
```

In questo modo ciascun atomo ha un indice associato ad esso. Questo sarà utilizzato nel modulo delle liste per la gestione delle cellette.

1.1 Parametri

Di seguito si mostra il contenuto del modulo `parametri`.

```
MODULE parametri

   ! Numero di atomi
  INTEGER, PARAMETER          :: nAtoms = 1000
  ! Dimensione della scatola
  DOUBLE PRECISION, PARAMETER :: boxLength = 100
  ! Raggio della forza, due atomi non interagiscono se la distanza e' maggiore di cutOff
  DOUBLE PRECISION, PARAMETER :: cutOff = 2
  ! Numero di suddivisioni in cellette della scatola grande: la scatola viene divisa in m*m*m
      celle
  INTEGER, PARAMETER          :: m = 50

END MODULE parametri
```

1.2 Gestione delle liste

La scatola è suddivisa in celletta, ciascuna celletta contiene un certo numero di atomi. Questa struttura dati è stata modellata attraverso un oggetto $m \times m \times m$ dimensionale, ciascun elemento di questo oggetto contiene un puntatore ad una lista, la lista contiene gli atomi presenti nella celletta.
Di seguito si mostra l'intestazione del modulo `liste`.

```
MODULE list_mod

    USE parametri

    ! Struttura che definisce un nodo
    TYPE listAtomsIndexType
        ! Indice dell'atomo
        INTEGER          :: atomIndex
        ! Puntatore che punta al nodo precedente
        TYPE(listAtomsIndexType), POINTER :: previous => NULL()
        ! Puntatore che punta al nodo successivo
        TYPE(listAtomsIndexType), POINTER :: next => NULL()
    END TYPE listAtomsIndexType

    ! Struttura che definisce un elemento dell'oggetto m*m*m
    TYPE listAtomsIndexTypePtr
        ! Puntatore alla testa della lista
        TYPE(listAtomsIndexType), POINTER :: head => NULL()
        ! Numero di elementi nella lista
        INTEGER     :: counter = 0
    END TYPE listAtomsIndexTypePtr

    ! Oggetto m*m*m dove ogni elemento e' associato ad una celletta
    TYPE(listAtomsIndexTypePtr) :: atomsInCellPtr(1:m,1:m,1:m)

    ! Matrice che contiene le coordinate delle celle di ogni atomo
    INTEGER :: atomsInCellCoordinates(1:3,1:nAtoms)

CONTAINS

   SUBROUTINE listAddHead(atomIndex,ix,iy,iz)

   SUBROUTINE listRmElement(atomIndex,ix,iy,iz)

   SUBROUTINE listCountAndCollect(interactAtomsIndexFilling,interactAtomsIndex,ix,iy,iz)

   SUBROUTINE fillInteractingArray(interactAtomsIndexFilling,interactAtomsIndex,ix,iy,iz)

   SUBROUTINE modCell(x,y,z)

END MODULE list_mod
```

Ciascuna celletta della scatola è associata ad una terna (x, y, z) che identifica la celletta stessa. Se chiamiamo (non presente nel codice) $smallBoxLength = boxLength/m =$ lunghezza di una celletta, le coordinate (i, j, k) corrisponderanno alla celletta con:

$$x \in [(i-1) \cdot smallBoxLength, i \cdot smallBoxLength]$$
$$y \in [(j-1) \cdot smallBoxLength, j \cdot smallBoxLength]$$
$$z \in [(k-1) \cdot smallBoxLength, k \cdot smallBoxLength]$$

Sono presenti due strutture dati derivate:

- `listAtomsIndexType` è il tipo che definisce il nodo di una cella. Esso contiene:
  - l'indice dell'atomo.

  - un puntatore che punta al nodo precedente (punta a `NULL()` se il nodo è il primo elemento della lista).
  - un puntatore che punta al nodo successivo (punta a `NULL()` se il nodo è l'ultimo elemento della lista).

- `listAtomsIndexTypePtr` è il tipo che punta alla testa di una lista e memorizza il numero di elementi in essa (punta a `NULL()` se la lista è vuota, ovvero se non ci sono atomi nella cella).

A far uso di queste due strutture dati è l'oggetto `atomsInCellPtr`, ovvero l'oggetto $m \times m \times m$ dimensionale in questione che indica quali atomi sono presenti in una data cella, in particolare è definito nel seguente modo: `TYPE(listAtomsIndexTypePtr) :: atomsInCellPtr(1:m,1:m,1:m)`. Infine `atomsInCellCoordinates` è una matrice $3 \times$ nAtomi che memorizza in quale cella si trova attualmente un atomo, la sua funzione è di evitare di dover scorrere ogni volta la `atomsInCellPtr` per verificare se un atomo si trova nella celletta. `atomsInCellCoordinates` viene aggiornata ogni volta che vengono effettuate operazioni sulla lista. Nelle prossime sezioni sono presenti le funzioni presenti nel modulo.

1.2.1 listAddHead(atomIndex,ix,iy,iz)

Aggiunge un nodo in testa: prende come ingresso l'indice dell'atomo `atomIndex` e le coordinate `(ix,iy,iz)` della celletta in cui l'atomo deve essere inserito. Dopo l'inserimento, si aggiorna il contatore del numero di nodi nella lista, ovvero si aggiorna `atomsInCellPtr(ix,iy,iz)%counter`. Successivamente si aggiorna anche `atomsInCellCoordinates`.

```
SUBROUTINE listAddHead(atomIndex,ix,iy,iz)

    IMPLICIT NONE

    INTEGER, INTENT(IN) :: atomIndex, ix, iy, iz
    TYPE(listAtomsIndexType), POINTER :: oldHead => NULL()
    TYPE(listAtomsIndexType), POINTER :: newHead => NULL()

    newHead => listAtomsIndex(atomIndex)
    oldHead => atomsInCellPtr(ix,iy,iz)%head

    ! Se esiste una lista, riarrangia gli indirizzi
    IF (ASSOCIATED(oldHead)) THEN
        newHead%next => oldHead
        oldHead%previous => newHead
        atomsInCellPtr(ix,iy,iz)%head => newHead

    ! Altrimenti inizializza gli indirizzi e poi inserisci
    ELSE
        NULLIFY(newHead%next)
        NULLIFY(newHead%previous)
   END IF

   atomsInCellPtr(ix,iy,iz)%head => newHead

    ! Aggiorna il numero di elementi nella lista
    atomsInCellPtr(ix,iy,iz)%counter = atomsInCellPtr(ix,iy,iz)%counter + 1
    atomsInCellCoordinates(1,atomIndex) = ix
    atomsInCellCoordinates(2,atomIndex) = iy
    atomsInCellCoordinates(3,atomIndex) = iz

END SUBROUTINE listAddHead
```

1.2.2 listRmElement(atomIndex,ix,iy,iz)

Rimuove un nodo: prende come ingresso l'indice dell'atomo `atomIndex` e le coordinate `(ix,iy,iz)` della celletta in cui l'atomo si trova. La rimozione del nodo avviene collegando il nodo successivo al nodo precedente.

- Se il nodo da rimuovere è in testa, il puntatore al nodo precedente del nodo successivo punta a `NULL()`
- Se il nodo da rimuovere è in coda, il puntatore al nodo successivo del nodo precedente punta a `NULL()`

Dopo la rimozione, si aggiorna `atomsInCellPtr(ix,iy,iz)%counter`. Successivamente si aggiorna anche `atomsInCellCoordinates` ponendo nella cella `(0,0,0)` l'atomo, ovvero si rimuove l'atomo dalla scatola (N.B. le coordinate dell'atomo non vengono toccate, quindi solo la gestione delle liste legge come l'atomo fuori dalla scatola, il programma principale legge l'atomo come ancora all'interno della scatola).

```
SUBROUTINE listRmElement(atomIndex,ix,iy,iz)

    IMPLICIT NONE

    INTEGER, INTENT(IN) :: atomIndex,ix,iy,iz

    TYPE(listAtomsIndexType), POINTER :: previous => NULL()
    TYPE(listAtomsIndexType), POINTER :: actual => NULL()
    TYPE(listAtomsIndexType), POINTER :: next => NULL()

   actual => listAtomsIndex(atomIndex)
   previous => actual%previous
   next => actual%next

   IF (ASSOCIATED(previous)) THEN
      IF (ASSOCIATED(next)) THEN
         previous%next => actual%next
         next%previous => actual%previous
      ELSE
         NULLIFY(previous%next)
      END IF
   ! Se il nodo e' il primo elemento
   ELSE
      ! Se esiste una lista
      IF (ASSOCIATED(next)) THEN
         NULLIFY(next%previous)
         atomsInCellPtr(ix,iy,iz)%head => actual%next
      ! Se e' l'unico elemento
      ELSE
         NULLIFY(atomsInCellPtr(ix,iy,iz)%head)
      END IF
   END IF

   ! Aggiorna il contatore
   atomsInCellPtr(ix,iy,iz)%counter = atomsInCellPtr(ix,iy,iz)%counter - 1
   atomsInCellCoordinates(1,atomIndex) = 0
   atomsInCellCoordinates(2,atomIndex) = 0
   atomsInCellCoordinates(3,atomIndex) = 0

END SUBROUTINE listRmElement
```

1.2.3 listCountAndCollect(interactAtomsIndexFilling,interactAtomsIndex,ix,iy,iz)

Il sottoprogramma conta il numero di atomi nella cella `(ix,iy,iz)` e riempie un array `interactAtomsIndex` che ne contiene gli indici a partire da `interactAtomsIndexFilling`. Il sottoprogramma successivamente incrementa `interactAtomsIndexFilling` del numero di atomi nella celletta `(ix,iy,iz)`, ovvero se per esempio `interactAtomsIndexFilling==5` prima dell'esecuzione del sottoprogramma e nella lista in `(ix,iy,iz)` ci sono 4 nodi, il sottoprogramma restituisce `interactAtomsIndexFilling=5+4=9` e `interactAtomsIndex` viene riempito dalle caselle dal 5 al 9.

```
SUBROUTINE listCountAndCollect(interactAtomsIndexFilling,interactAtomsIndex,ix,iy,iz)

   IMPLICIT NONE

    INTEGER, INTENT(INOUT) :: interactAtomsIndexFilling
    INTEGER, INTENT(OUT) :: interactAtomsIndex(1:nAtoms)
    INTEGER, INTENT(IN)  :: ix,iy,iz

    TYPE(listAtomsIndexType), POINTER :: ptr => NULL()

    ptr => atomsInCellPtr(ix,iy,iz)%head
    ! Esce dal DO WHILE quando ha finito di scorrere gli elementi della lista
    DO WHILE (ASSOCIATED(ptr))
        interactAtomsIndexFilling = interactAtomsIndexFilling + 1
        interactAtomsIndex(interactAtomsIndexFilling) = ptr%atomIndex
        ptr => ptr%next
    END DO

END SUBROUTINE listCountAndCollect
```

1.2.4 fillInteractingArray(interactAtomsIndexFilling,interactAtomsIndex,ix,iy,iz)

Questo sottoprogramma decide quali caselle adiacenti alla casella `(ix,iy,iz)` valutare, in questo modo si dimezzano le operazioni necessarie per scorrere tutte le liste.

```
SUBROUTINE fillInteractingArray(interactAtomsIndexFilling,interactAtomsIndex,ix,iy,iz)

   USE parametri

   IMPLICIT NONE

   INTEGER, INTENT(INOUT) :: interactAtomsIndexFilling
   INTEGER, INTENT(OUT) :: interactAtomsIndex(1:nAtoms)
   INTEGER, INTENT(IN)   :: ix,iy,iz

   INTEGER            :: tx,ty,tz

   INTEGER :: i

   ! Pagina 32 appunti di Fisica Computazionale

   ! Colleziona atomi nelle celle vicine (Sono essere 13 blocchi di codice quasi identici)

   ! Blocco 1
   tx = ix
   ty = iy + 1
```

```
   tz = iz
   CALL modCell(tx,ty,tz)
   CALL listCountAndCollect(interactAtomsIndexFilling,interactAtomsIndex,tx,ty,tz)

   ! Altri 12 blocchi dello stesso codice con solo tx, ty e tz che cambiano

END SUBROUTINE fillInteractingArray
```

1.2.5 modCell(x,y,z)

Si riportano gli indici `(x,y,z)` nel cubo di periodicità `m`.

```
SUBROUTINE modCell(x,y,z)

    USE parametri

   IMPLICIT NONE

   INTEGER, INTENT(INOUT) :: x,y,z

   IF (x>m) THEN
      x=x-m
   ELSE IF (x<1) THEN
      x=x+m
   END IF
   IF (y>m) THEN
      y=y-m
   ELSE IF (y<1) THEN
      y=y+m
   END IF
   IF (z>m) THEN
      z=z-m
   ELSE IF (z<1) THEN
      z=z+m
   END IF
END SUBROUTINE
```

1.3 La fisica del programma

La parte che riguarda la fisica della simulazione si trova nel modulo `physics`, in particolare di seguito si mostra l'intestazione del programma.

```
MODULE physics

    USE parametri
    USE list_mod

    DOUBLE PRECISION :: kineticEnergy
    DOUBLE PRECISION :: potentialEnergy
    DOUBLE PRECISION :: pressure = -1
    DOUBLE PRECISION :: temperature

    ! callingCalcRefresh serve per capire se e' stata chiamata calcRefresh
```

```
    INTEGER, PRIVATE :: callingCalcRefresh = 0
    ! computingForce serve per aggiornare pressure in modo piu' efficiente
    INTEGER, PRIVATE :: computingForce = 0
    ! Viene aggiornato durante il calcolo delle forze, la pressione finale pressure iene calcolata
        da qui
    DOUBLE PRECISION, PRIVATE :: tempPressure = 0

    DOUBLE PRECISION, PRIVATE :: force(1:3,1:nAtoms)

CONTAINS

   SUBROUTINE calcDistance(distance,direction,position1,position2)

   SUBROUTINE calcForceTwoParticles(force,position1,position2)

   SUBROUTINE calcForce(x)

   SUBROUTINE calcKineticEnergy(atoms)

   SUBROUTINE calcTemperature(atoms)

   SUBROUTINE calcPressure(atoms)

   SUBROUTINE calcRefresh(atoms)

   SUBROUTINE dynamic(xPrime,x,t)

END MODULE physics
```

Le variabili `kineticEnergy`, `potentialEnergy`, `pressure` e `temperature` sono le quantità di interesse fisico attualmente presenti nel programma. I sottoprogrammi che li calcolano sono implementati in modo da leggere le flags `callingCalcRefresh` e `computingForce` e ottimizzare l'utilizzo delle risorse. La variabile `tempPressure` serve per memorizzare dati per il calcolo della pressione e viene calcolata nel sottoprogramma `calcForceTwoParticles`. Energia cinetica, pressione e temperatura possono essere calcolate indipendentemente da `calcKineticEnergy`, `calcTemperature` e `calcPressure`, mentre il calcolo dell'energia potenziale avviene tramite `calcForce`.

1.3.1 dynamic(xPrime,x,t)

Il sottoprogramma prende in ingresso il tempo `t` (aggiunto per completezza per via del sottoprogramma che utilizza il metodo Runge Kutta), la posizione e la velocità delle particelle `x`, in particolare `x(1:3,:)` contiene la posizione delle particelle e `x(4:6,:)` contiene le velocità e restituisce `xPrime`. Il sottoprogramma imposta la flag `computingForce=1` dato che in questo modo la pressione viene calcolata in `calcPressure` utilizzando la variabile `tempPressure` e non a partire da zero (vedi la sezione 1.3.3).

```
SUBROUTINE dynamic(xPrime,x,t)

   IMPLICIT NONE
     DOUBLE PRECISION, INTENT(OUT) :: xPrime(1:6,1:nAtoms)
   DOUBLE PRECISION, INTENT(IN) :: x(1:6,1:nAtoms)
   DOUBlE PRECISION, INTENT(IN) :: t

   INTEGER :: i

   computingForce = 1
```

```
   potentialEnergy = 0
   CALL calcForce(x(:,:))
   CALL calcRefresh(x(:,:))
   computingForce = 0

   DO i=1,3
      xPrime(i,:) = x(i+3,:)
      xPrime(i+3,:) = force(i,:)
   END DO

END SUBROUTINE dynamic
```

1.3.2 calcRefresh(atoms)

Il sottoprogramma aggiorna le variabili fisiche e viene chiamato ogni volta che viene effettuato uno step di integrazione. Si imposta la flag `callingCalcRefresh=1` per non dover calcolare le stesse quantità fisiche più volte:

- `calcKineticEnergy` calcola l'energia cinetica degli atomi del sistema.
- `calcTemperature` calcola la temperatura del sistema; se `callingCalcRefresh=1` si utilizza l'energia cinetica del sistema precedentemente calcolata (anziché doverla ricalcolare).
- `calcPressure` calcola la pressione del sistema; se `callingCalcRefresh=1` si utilizza la temperatura del sistema precedentemente calcolata (anziché doverla ricalcolare).

```
SUBROUTINE calcRefresh(atoms)

   USE parametri

   IMPLICIT NONE

   DOUBLE PRECISION, INTENT(IN) :: atoms(1:6,1:nAtoms)

   callingCalcRefresh = 1
   CALL calcKineticEnergy(atoms(:,:))
   CALL calcTemperature(atoms(:,:))
   CALL calcPressure(atoms(:,:))
   callingCalcRefresh = 0

END SUBROUTINE calcRefresh
```

1.3.3 calcPressure(atoms)

Il sottoprogramma calcola la pressione del sistema e ne memorizza il valore nella variabile globale `pressure`:

- Se viene chiamato da `dynamic`, allora non si calcola la forza dato che `tempPressure` è già aggiornato.
- Se viene chiamato da `calcRefresh`, allora non calcola la temperatura dato che è stata appena calcolata.

```
SUBROUTINE calcPressure(atoms)
```

```
   IMPLICIT NONE

   DOUBLE PRECISION, INTENT(IN) :: atoms(1:6,1:nAtoms)

   IF (callingCalcRefresh == 0) THEN
      CALL calcTemperature(atoms(:,:))
   END IF

   IF (computingForce == 0) THEN
      CALL calcForce(atoms(:,:))
   END IF

   pressure = (2*temperature+tempPressure)/(3*boxLength**3)
   tempPressure = 0

END SUBROUTINE calcPressure
```

1.3.4 calcTemperature(atoms)

Il sottoprogramma calcola la temperatura del sistema e ne memorizza il valore nella variabile globale `temperature`. Se viene chiamato da `calcRefresh`, allora non calcola l'energia cinetica dato che è stata appena calcolata.

```
SUBROUTINE calcTemperature(atoms)

   IMPLICIT NONE

   DOUBLE PRECISION, INTENT(IN) :: atoms(1:6,1:nAtoms)

   DOUBLE PRECISION :: E_kin=0, temp

   INTEGER :: i,j

   IF (callingCalcRefresh==0) THEN
      E_kin = 0
      DO i=1,nAtoms
         temp = 0
         DO j = 4,6
            temp = temp + atoms(j,i)**2
         END DO
         E_kin = E_kin + temp
      END DO
   ELSE
      E_kin = kineticEnergy
   END IF

   temperature = E_kin/(3*nAtoms)

END SUBROUTINE calcTemperature
```

1.3.5 calcKineticEnergy(atoms)

Calcola l'energia cinetica del sistema e memorizza il valore nella variabile globale `kineticEnergy`. Il sottoprogramma prende come ingresso `atoms`, pur utilizzando solo le componenti della velocità.

```
SUBROUTINE calcKineticEnergy(atoms)

   IMPLICIT NONE

   DOUBLE PRECISION, INTENT(IN) :: atoms(1:6,1:nAtoms)
   DOUBLE PRECISION :: temp

   INTEGER :: i,j

   kineticEnergy = 0
   DO i=1,nAtoms
      temp = 0
      DO j = 4,6
         temp = temp + atoms(j,i)**2
      END DO
      kineticEnergy = kineticEnergy + temp
   END DO

END SUBROUTINE
```

1.3.6 calcForce(x)

Il sottoprogramma calcola le forze che agiscono su ciascuna particella e prende in ingresso le posizioni `x` degli atomi; le forze esercitate sono memorizzate sulla variabile globale `force`. Il sistema è una scatola cubica divisa in $m \times m \times m$ cellette: le variabili `(tx,ty,tz)` scorrono queste celle.

1. Quando si è su una cella`(tx,ty,tz)`, si calcola il numero di atomi presenti in essa e si salva il valore nella variabile `interactAtomsIndexFillingThisCell` e si pongono gli indici degli atomi nell'array `interactAtomsIndex`; quindi l'array `interactAtomsIndex` funge da pila ed è non nullo fino a `interactAtomsIndexFillingThisCell`. Se non ci sono atomi nella cella (pila vuota), si passa alla prossima cella e si ignorano le altre istruzioni nel ciclo (`CYCLE` in Fortran equivale a `continue` negli altri linguaggi).

2. Se invece ci sono atomi nella cella, si collezionano gli indici degli atomi in 13 celle[1] adiacenti sempre sopra `interactAtomsIndex`, stavolta è la variabile `interactAtomsIndexFillingAllCells` che indica quanti elementi ci sono in tutte le celle. Quindi, ricapitolando:

   `interactAtomsIndexFillingAllCells` = `interactAtomsIndexFillingThisCell` +
   + `numero di atomi in 13 celle adiacenti`

   `interactAtomsIndexFillingThisCell` = numero di atomi nella cella `(tx,ty,tz)`

   `interactAtomsIndex` = array che contiene gli indici degli atomi da valutare

   I primi `interactAtomsIndexFillingThisCell` indici in `interactAtomsIndex` sono quindi gli indici degli atomi nella cella `(tx,ty,tz)`.

3. Si fanno successivamente interagire gli atomi:

[1] Le 13 celle sono state scelte in modo da dimezzare il numero di operazioni da effettuare, questo è realizzato con `fillInteractingArray` nella sezione 1.2.4.

- Gli atomi nella cella `(tx,ty,tz)` non vanno valutati due volte, quindi bisogna porre il vincolo `i<j` (realizzato con il "ciclo `FOR`").
- Non è invece necessaria alcuna attenzione per atomi che si trovano in celle diverse da `(tx,ty,tz)` dato che non c'è il rischio di doppia valutazione (questo perché sono state scelte in modo accurato le 13 celle adiacenti).

4. Dopo aver scelto quali particelle far interagire e come, si calcola la forza fra due particelle attraverso il sottoprogramma `calcForceTwoParticles`.

Le forze calcolate vengono memorizzate nella variabile globale `force`. `tempVec` è un array che contiene la forza tra le due particelle.

```
SUBROUTINE calcForce(x)

   IMPLICIT NONE

   DOUBLE PRECISION, INTENT(IN) :: x(1:6,1:nAtoms)
   DOUBLE PRECISION :: tempVec(1:3)

   INTEGER :: i, j, atomI, atomJ, tx,ty,tz

   ! Array che contiene l'indici degli atomi che interagiscono
   INTEGER :: interactAtomsIndex(1:nAtoms)
   ! Intero che indica quanto e' pieno interactAtomsIndex
   INTEGER :: interactAtomsIndexFillingThisCell
   INTEGER :: interactAtomsIndexFillingAllCells

   ! Calcola la forza: decide quale forze vanno valutate e valuta quelle
   force = 0
   potentialEnergy = 0
   DO tx=1,m
      DO ty=1,m
         DO tz=1,m

            interactAtomsIndex = 0
            interactAtomsIndexFillingThisCell = 0
            interactAtomsIndexFillingAllCells = 0

            ! Colleziona atomi nella stessa cella (Blocco 0)
            CALL listCountAndCollect(interactAtomsIndexFillingThisCell,interactAtomsIndex,tx,ty,tz)

            ! Se in questa cella non ci sono atomi, skippa
            IF (interactAtomsIndexFillingThisCell == 0) THEN
               CYCLE
            END IF

            ! Colleziona gli atomi di tutte le celle
            interactAtomsIndexFillingAllCells = interactAtomsIndexFillingThisCell
            CALL fillInteractingArray(interactAtomsIndexFillingAllCells,
                interactAtomsIndex(:),tx,ty,tz)

            ! Fai interagire tra di loro gli atomi nella stessa cella
            DO i=1,interactAtomsIndexFillingThisCell,+1
               DO j=i+1,interactAtomsIndexFillingThisCell,+1

                  atomI = interactAtomsIndex(i)
```

```
            atomJ = interactAtomsIndex(j)

            CALL calcForceTwoParticles(tempVec(:), x(1:3,atomI), x(1:3,atomJ))
            force(:,atomI) = force(:,atomI) + tempVec
            force(:,atomJ) = force(:,atomJ) - tempVec

          END DO
        END DO

          ! Fai interagire tra di loro gli atomi della cella (tx,ty,tz) con gli atomi delle
              celle adiacenti
        DO i=1,interactAtomsIndexFillingThisCell,+1
          DO j=interactAtomsIndexFillingThisCell+1,interactAtomsIndexFillingAllCells,+1

            atomI = interactAtomsIndex(i)
            atomJ = interactAtomsIndex(j)

            CALL calcForceTwoParticles(tempVec(:), x(1:3,atomI), x(1:3,atomJ))
            force(:,atomI) = force(:,atomI) + tempVec
            force(:,atomJ) = force(:,atomJ) - tempVec

          END DO
        END DO
      END DO
    END DO
  END DO

END SUBROUTINE calcForce
```

1.3.7 calcForceTwoParticles(force,position1,position2)

Il sottoprogramma calcola e restituisce la forza `force` tra due particelle `position1` e `position2`. La forze è data dal potenziale di Lennard Jones ed è l'attrazione che la particella 2 esercita sulla particella 1; quindi, se la forza fosse attrattiva, il vettore punterebbe dalla particella 1 alla particella 2. Infine il sottoprogramma calcola anche il contributo all'energia potenziale data da questa interazione.

```
  SUBROUTINE calcForceTwoParticles(force,position1,position2)

  IMPLICIT NONE

  DOUBLE PRECISION, INTENT(IN) :: position1(1:3)
  DOUBLE PRECISION, INTENT(IN) :: position2(1:3)
  DOUBLE PRECISION, INTENT(OUT) :: force(1:3)

  ! distance e' la distanza, direction e' il versore della forza
  DOUBLE PRECISION :: distance, direction(1:3)

  INTEGER :: i

  CALL calcDistance(distance,direction(:),position1(:),position2(:))

  IF (distance>cutOff) THEN
    force = 0
  ELSE
```

```
      force = 4 * &
         ( ((1/distance)**12 * 12) &
         - ((1/distance)**6 * 6) ) &
         *direction/(distance**2)
   END IF

   ! Questa parte serve per il calcolo della pressione
   DO i=1,3
      tempPressure = tempPressure + force(i)*direction(i)
   END DO

   ! Questa parte aggiorna l'energia potenziale
   potentialEnergy = potentialEnergy + 4 * &
        (( (1/distance)**12 * 12) &
        - ((1/distance)**6 * 6) )

END SUBROUTINE
```

1.3.8 calcDistance(distance,direction,position1,position2)

Il sottoprogramma calcola la distanza tra due particelle 1 e 2, tenendo conto del cubo di periodicità `boxLength`. Prende in ingresso la posizione della particella 1 `position1` e la posizione della particella 2 `position2`, mentre restituisce la distanza `distance` e la direzione (non normalizzata) `direction` dalla particella 1 alla particella 2.

```
SUBROUTINE calcDistance(distance,direction,position1,position2)

   IMPLICIT NONE

   DOUBLE PRECISION, INTENT(IN) :: position1(1:3)
   DOUBLE PRECISION, INTENT(IN) :: position2(1:3)

   DOUBLE PRECISION, INTENT(OUT) :: distance
   ! Direction non e' normalizzato
   DOUBLE PRECISION, INTENT(OUT) :: direction(1:3)

   INTEGER :: halfBox = boxLength/2

   ! n assume valori che indicano se il secondo atomo va spostato e dove (condizioni periodiche
   ! a destra o a sinistra
   INTEGER    :: n(1:3)

   ! Contatori
   INTEGER    :: i


   ! Se la differenza delle coordinate tra le due particelle su una direzione e' maggiore halfBox,
       sposta
   ! la particella 2 a seconda della posizione della particella 1 rispetto alla particella (lo
       spostamento
   ! avviene impostando un valore idoneo sulla variabile n)

   ! Verifica che il secondo atomo sia nel cubo
   DO i=1,3
```

```
      ! La distanza e' minore di halfBox
      IF (ABS(position1(i)-position2(i))<halfBox) THEN
         n(i) = 0

      ! La distanza e' maggiore di halfBox, sposta gli atomi
      ELSE
         ! La particella 2 va spostata a sinistra dato che si trova a destra
         IF (position2(i)>position1(i)) THEN
            n(i) = -1

         ! La particella 2 va spostata a destra dato che si trova a sinistra
         ELSE IF (position2(i)<position1(i)) THEN
            n(i) = +1
         END IF
      END IF

   END DO

   ! Calcola la distanza
   distance = SQRT((position2(1)+n(1)*boxLength-position1(1))**2 + &
              (position2(2)+n(1)*boxLength-position1(2))**2 + &
              (position2(3)+n(1)*boxLength-position1(3))**2)
   direction = (position1-position2+n*boxLength)

END SUBROUTINE
```

---

2 Programma principale

Il programma principale è composto da un ciclo `DO WHILE` che effettua una chiamata a Runge Kutta e riporta gli atomi nel cubo di periodicità `boxLenght` attraverso il sottoprogramma `periodicConditions`. Si definisce un tempo iniziale `t_in`, un tempo finale `t_fin` e uno step di integrazione `step`.

```
PROGRAM DinamicaMolecolare

   USE parametri
   USE list_mod

   USE physics

   IMPLICIT NONE

   DOUBLE PRECISION :: atoms(1:6,1:nAtoms)

   INTEGER :: k=0

   DOUBLE PRECISION, PARAMETER :: t_in = 0, t_fin = 100
   DOUBLE PRECISION :: step = 1e-3
   DOUBLE PRECISION :: t

   CALL initStructure(atoms(:,:))

   ! Simulazione del sistema
```

```
   !OPEN(1,file="Dinamica.dat")

   t = t_in

   DO WHILE (t<t_fin)
      k=k+1

      ! Se si e' all'ultimo step di integrazione
      IF (t+step>t_fin) THEN
         step = t_fin-t
      END IF

      CALL RungeKutta(dynamic,atoms(:,:),t,step)
      t=t+step
      CALL periodicConditions(atoms(1:3,:))

      PRINT*, k,t,temperature, pressure

   END DO

   !CLOSE(1)

END PROGRAM DinamicaMolecolare
```

3 Sottoprogrammi vari

3.1 initStructure(atoms)

Questo sottoprogramma serve per inizializzare le strutture dati. Prende come ingresso la matrice `atoms` da inizializzare ed effettua le seguenti operazioni:

1. Posiziona gli atomi nella scatola in posizioni random con velocità random.
2. Successivamente compila l'oggetto $m \times m \times m$ dimensionale che contiene le liste con gli indici degli atomi, ovvero compila `listAtomsIndex`.

```
SUBROUTINE initStructure(atoms)

   USE parametri
   USE list_mod

   IMPLICIT NONE

   DOUBLE PRECISION, INTENT(INOUT) :: atoms(1:6,1:nAtoms)

   DOUBLE PRECISION :: normalizedPositions(1:3,1:nAtoms)

   INTEGER :: seed=6

   INTEGER :: i, j, ix, iy, iz

   CALL SRAND(seed)
```

```
   ! Genera la parte della posizione e della velocita'
   DO i=1,nAtoms
      DO j=1,3
         atoms(j,i) = RAND()*boxLength
      END DO
      DO j=4,6
         atoms(j,i) = RAND()
      END DO
   END DO

   !atoms(:,1) = (/2., 0., 0., -1., 0., 0./)
   !atoms(:,2) = (/98.9, 0., 0., 1., 0., 0./)

   ! Normalizza le posizioni su m, in questo modo normalizedPosition ha gia' l'indice incorporato
   normalizedPositions = atoms(1:3,:)/boxLength
   normalizedPositions = normalizedPositions*m

   DO i = 1,nAtoms

      ! Inizializza listAtomsIndex
      listAtomsIndex(i)%atomIndex = i

      ix = CEILING(normalizedPositions(1,i))
      iy = CEILING(normalizedPositions(2,i))
      iz = CEILING(normalizedPositions(3,i))

      IF (ix==0) THEN
         ix = 1
      END IF
      IF (iy==0) THEN
         iy = 1
      END IF
      IF (iz==0) THEN
         iz = 1
      END IF

      ! Inserisci di testa
      CALL listAddHead(i,ix,iy,iz)

   END DO

END SUBROUTINE initStructure
```

3.2 periodicConditions(positions)

Il sottoprogramma riporta le particelle nel cubo di periodicità `boxLength`, come ingresso prende la posizione degli atomi nella scatola. È possibile avere:

- `FLOOR(positions(j,i)/boxLength)==0`, pertanto l'atomo è ancora nella scatola e non viene fatta nessuna operazione.
- `FLOOR(positions(j,i)/boxLength)<0`, l'atomo si trova fuori dalla scatola "a sinistra", quindi lo si riporta a destra

- `FLOOR(positions(j,i)/boxLength)>0`, l'atomo si trova fuori dalla scatola "a destra", quindi lo si riporta a sinistra.

In questo caso non viene fatta nessuna

```
SUBROUTINE periodicConditions(positions)

   USE parametri

   IMPLICIT NONE

   DOUBLE PRECISION, INTENT(INOUT) :: positions(1:3,1:nAtoms)

   INTEGER         :: i, j, n

   DO i=1,nAtoms
      DO j=1,3
         n = FLOOR(positions(j,i)/boxLength)
         positions(j,i) = positions(j,i) - n*boxLength
      END DO
   END DO

   CALL orderAtomsInCell(positions(:,:))

END SUBROUTINE periodicConditions
```

3.3 orderAtomsInCell(atoms)

Il sottoprogramma prende come ingresso la posizione degli atomi e verifica che siano ancora nelle celle del passo di integrazione precedente, se hanno cambiato celletta, vengono rimossi dalla lista corrispondente e vengono aggiunti in testa alla nuova lista.

```
SUBROUTINE orderAtomsInCell(atoms)

   USE parametri
   USE list_mod

   IMPLICIT NONE

   DOUBLE PRECISION, INTENT(IN) :: atoms(1:3,1:nAtoms)

   TYPE(listAtomsIndexType), POINTER :: previous
   TYPE(listAtomsIndexType), POINTER :: actual
   TYPE(listAtomsIndexType), POINTER :: next

   DOUBLE PRECISION :: normalizedPositions(1:3,1:nAtoms)

   INTEGER         :: newX,newY,newZ,oldX,oldY,oldZ
   INTEGER         :: atomIndex

   ! Normalizza le posizioni su m, in questo modo normalizedPosition ha gia' l'indice incorporato
   normalizedPositions = atoms(1:3,:)/boxLength
   normalizedPositions = normalizedPositions*m
```

```
   DO atomIndex = 1,nAtoms

      newX = CEILING(normalizedPositions(1,atomIndex))
      newY = CEILING(normalizedPositions(2,atomIndex))
      newZ = CEILING(normalizedPositions(3,atomIndex))

      IF (newX==0) THEN
         newX = 1
      END IF
      IF (newY==0) THEN
         newY = 1
      END IF
      IF (newZ==0) THEN
         newZ = 1
      END IF

      oldX = atomsInCellCoordinates(1,atomIndex)
      oldY = atomsInCellCoordinates(2,atomIndex)
      oldZ = atomsInCellCoordinates(3,atomIndex)

      ! Controlla se l'atomo e' ancora nella celletta
      ! Se e' ancora nella celletta, passa al prossimo atomo
      IF ((newX == oldX) .AND. (newY == oldY) .AND. (newZ == oldZ)) THEN
         CYCLE
      END IF

      ! Se l'atomo ha cambiato celletta, esegui

      ! Rimuovi l'atomo dalla lista della cella
      CALL listRmElement(atomIndex,oldX,oldY,oldZ)

      ! Aggiungi in testa
      CALL listAddHead(atomIndex,newX,newY,newZ)

   END DO

END SUBROUTINE orderAtomsInCell
```

3.4 RungeKutta(func,x,t,step)

Algoritmo Runge Kutta.

```
SUBROUTINE RungeKutta(func,x,t,step)

   USE parametri


   IMPLICIT NONE

   DOUBLE PRECISION, INTENT(INOUT) :: x(1:6,1:nAtoms)
   DOUBLE PRECISION, INTENT(IN) :: t, step

   DOUBLE PRECISION :: c(0:3)=(/ 0., 0.5, 0.5, 1. /) ! Nodi
```

```
  DOUBLE PRECISION :: b(0:3)=(/ 1./6, 1./3, 1./3, 1./6 /) ! Pesi
  DOUBLE PRECISION :: k(0:4,1:6,1:nAtoms) ! k(0)=0 e serve per ottimizzare l'algoritmo

  DOUBLE PRECISION :: x_t(1:6,1:nAtoms)
  DOUBLE PRECISION :: tt

  INTEGER          :: j

  EXTERNAL      :: func

  ! Valuta i coefficienti
  k = 0
  DO j=1,4
     x_t = x + k(j-1,:,:)*step*c(j-1)
     tt = t + step*c(j-1)
     CALL func(k(j,:,:),x_t(:,:),tt)

  END DO

  DO j=1,4
     x = x + k(j,:,:)*b(j-1)*step
  END DO

END SUBROUTINE RungeKutta
```