I. SUPPLEMENTARY MATERIAL

A. C/Fortran implementation of the SANN scheme

Hereby, we propose the C (SANN.c) and Fortran (SANN.f) implementation of the SANN scheme.

B. SANN.c

```
/*
* @file    sann.c
* @author van Meel, Filion, Valeriani, Frenkel
* @date    November 2011
* @brief  Sample implementation of the SANN algorithm in C
*/



/*
* @struct NbData
* @brief Defines an {index,distance} pair for use as neighbour data
*/
struct NbData
{
        int    id; // Id of neighbour particle
        double distance; // Distance to neighbour particle
};

/*
* @brief Compares the distance of two neighbours for use in 'qsort'
* @param nb1 Pointer to first neighbour's NbData
* @param nb2 Pointer to second neighbour's NbData
```

```
* @retval Returns negative if less, positive otherwise
*/
int nbLess( const void *nb1, const void *nb2 )
{
// cast 'const void' pointer to 'const NbData' pointer for
//both neighbours
        NbData *pnb1 = (const NbData*) nb1;
        NbData *pnb2 = (const NbData*) nb2;

// If first distance is smaller than second distance return
//negative value
        if ( pnb1->distance < pnb2->distance ) return -1;

// Forget about 'equal' when using floating point numbers...
// Return positive value for 'greater than'
        return 1;
}

/*
* @brief Computes the SANN set of nearest neighbours for a given particle
* @param id Id of particle who's neighbours are to be computed
* @param neighbours NbData array to receive neighbour {id,distance} pairs
* @param radius Pointer to double to receive SANN radius
* @retval Number of neighbours computed by SANN, (-1) on error
*/
int computeSANN ( int id, NbData *neighbours, double *radius )
{
        double distanceSum; // sum of neighbour distances
        int count; // number of all potential neighbours available
        int i; // a loop variable
```

```
//Step 1:
//    Get number and {id,distance} pairs of all potential neighbours.
//    In this example we use a Verlet neighbour list with a
//    large-enough cutoff distance for this task. SANN then chooses
//    its neighbours from this set.
        count = computeVerletNeighbors( id, neighbours );

// If there are not enough neighbours available, report an error
        if (count < 3) return -1;

// Step 2:
//    Sort neighbours according to their distance in increasing order.
//    In this example we use a 'quicksort' algorithm for this task,
//    which exists in the standard C library stdlib.h
        qsort( neighbours, count, sizeof( NbData ), nbLess );

// Step 3 / 4:
//    Start with 3 neighbours (it's the minimum number of
//    neighbours possible)
        distanceSum = 0;
        for (i=0; i<3; ++i)
        {
// Add neighbour distance to sum
                distanceSum += nbData[i].distance;
        }
// Set SANN radius to distanceSum / (i - 2)
        *radius = distanceSum;

// Step 4 / 5:
//    Iteratively include further neighbours until finished, which is if
//    the SANN radius is smaller than the distance to the next neighbour
```

```
        while ((i < count) && (radius > neighbours[i].distance))
        {
// Add neighbour distance to sum
                distanceSum += neighbours[i].distance;
// Compute new SANN radius
                *radius = distanceSum / (i - 2.0);
// increase the SANN number of neighbours
                ++i;
        }

// If there were not enough neighbours for the algorithm to converge,
// report an error
        if (i == count) return -1;

// Step 6:
//    Return the number of SANN neighbours.
//    Note: the SANN radius has already been stored in the
//pointer 'radius',
//    which was provided as parameter to the function
        return i;
}

// end-of-file
```

C. SANN.f

```
      subroutine SANN

!     Fortran implementation of the SANN algorithm
!
!     van Meel, Filion, Valeriani and Frenkel November (2011)
```

```
!       declare all variable used in the subroutine
        implicit none

!       npart = total number of particles in the system
        integer npart
!       m = tentative number of neighbours
        integer i,j,k,m
!       countneighbors = number of neighbours of particle i
        integer countneighbors(1000)
!       neighbor = list of neighbours of particles i
        integer neighbor(1000,100)
!       sortneighbor = sorted neighbours
        integer sortneighbor(1000,100)
!       selectedneighbors = list of selected neighbours
        integer selectedneighbors(1000,100)
!       Nb = final number of neighbours of particle i
        integer Nb(1000)
!       edge of the simulation box
        double precision box
!       distance = list of distances between each
!       neighbour of particle i and particle i
        double precision  distance(1000,100)
!       distancesorted = sorted distances
        double precision distancesorted(1000,100)
!       R(m) as in Eq.3 in the manuscript
        double precision rm,rm1
!       x,y,z component of every particle i
        double precision x(1000),y(1000),z(1000)
```

```
!       distance between particle i and particle j
        double precision dr
!       cutoff distance to identify all potential neighbours
        double precision rcutoff



!       Step 1:
!       first we identify the particles within a cutoff radius rcutoff
        do i=1,npart
!       loop over all particles different from i
           do j =1,npart
              if (j.ne.i)then
!       compute x,y,z component of the distance between particle i and j
                 dx = x(j) - x(i)
                 dy = y(j) - y(i)
                 dz = z(j) - z(i)
!       applying periodic boundary conditions
                 dx=dx-nint(dx/box)*box
                 dy=dy-nint(dy/box)*box
                 dz=dz-nint(dz/box)*box
!       compute distance dr  between particle i and j
                 dr =  sqrt(dx*dx+dy*dy+dz*dz)
!       identify neighbours that are within a cutoff (rcutoff)
                 if(dr.lt.rcutoff)then
!       j is a neighbour of i
                    countneighbors(i) = countneighbors(i) + 1
!       build a list of neighbours
                    neighbor(i,countneighbors(i))= j
!       create a list with the distance between i and j
                    distance(i,countneighbors(i))=dr
                 endif
```

```
            endif
         enddo
      enddo

!     Step 2:
!     for every particle i sort all (countneighbors)
!     neighbours (neighbor) according to their
!     distances (distance) and create  a new list of
!     particle i's (sortneighbor)
!     and a new sorted list of distances (distancesorted)
      do i=1,npart
         call sort(i,countneighbors,distance,neighbor,
     &         sortneighbor,distancesorted)
      enddo

      do i=1,npart
!     Step 3:
!     start with 3 neighbours
         m = 3
!     Step 4:
!     compute R(m) as in Eq.3
         rm = 0
         do k=1,m
            rm = rm + distancesorted(i,k)
         enddo
         rm = rm/(m-2)
!     compute r(m+1)
         do j = 1,countneighbors(i)
            rm1 = 0
            do k=1,m
               rm1 = rm1 + distancesorted(i,k)
```

```
          enddo
          rm1 = rm1/(m-2)
!      Step 5:
!      if rm > rm1
          if(rm.ge.rm1)then
            rm = rm1
!      increase m
            m = m+1
          else
!      Step 6:
!      if rm < rm1, m is the final number of neighbours
            exit
          endif
        enddo
!      the final number of neighbours is m = Nb(i)
!      and the neighbours are  selectedneighbors
        Nb(i) = m
        do j=1,Nb(i)
          selectedneighbors(i,j) = sortneighbor(i,j)
        enddo
      enddo



      return
      end

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```