Volume of the Off-center Spherical Pyramidal Trunk

Richard J. Mathar*
Sterrewacht Leiden, P.O. Box 9513, 2300 RA Leiden, The Netherlands

(Dated: February 6, 2012)

The volume inside intersecting spheres may be computed by a standard method which computes a surface integral over all visible sections of the spheres. If the visible sections are divided in simple zonal sections, the individual contribution by each zone follows from basic analysis. We implement this within a semi-numerical program which marks the zones individually as visible or invisible.

I. SCOPE

The computation of the volume of intersecting spheres appears in some applications of molecular chemistry [1–7], or while hashing the volume inside complicated geometries by inflating spheres until they hit the internal walls of some boundary. The task is well defined once the centers $\mathbf{c}_s$ and radii ρ_s of all participating spheres are provided.

We implement the well-known approach which converts the integral over all points that are in at least one of the spheres to a surface integral with the aid of the divergence theorem. The volume is one third of the integral over all visible surfaces; the integral kernel is the dot product of the vector to the point on the surface by the normal at that point of the sphere pointing away from the sphere. (Here, visible surface elements are those that on the surface of one sphere and not in the interior of any other sphere. This includes surfaces inside cages and in that respect differs from *accessible* surfaces defined in molecular chemistry.)

II. TESSELATION

A subdivision of the surface of each sphere s in patches of approximately equidistributed areas is devised as follows. The polar angle θ is split into N_θ intervals of length $\Delta\theta$ centered at θ_i:

$$\theta_i = (i+\frac{1}{2})\Delta\theta; \quad 0 \le i < N_\theta \tag{1}$$

$$\Delta\theta = \frac{\pi}{N_\theta}; \quad N_\theta \ge 1. \tag{2}$$

The quality of the division increases if N_θ becomes larger.

The interval $[\theta_i - \Delta\theta/2, \theta_i + \Delta\theta/2]$ of polar angles defines a band around the sphere. The area of this region on the sphere surface is estimated by Guldin's rule by rotating the line segment of length $\rho\Delta\theta$ around the spherical axis such that the center of the line travels a distance $2\pi\rho\sin\theta_i$ with a full rotation. This product of these two values is

$$A_i = 2\pi\rho^2\Delta\theta\sin\theta_i. \tag{3}$$

(Note that is not the exact value of the associated surface because we have replaced the mean center of the line segment in Guldin's rule by the geometric center of mass.)

The azimuthal angles are chosen such that their separation along the equator equals again $\Delta\theta$, dividing the azimuth 2π into $2N_\theta$ intervals there. Dividing A_i by $2N_\theta$ at $\theta_i = \pi/2$ (at the equator) and using (2) yields that the average tile area is designed to be approximately $(\pi\rho/N_\theta)^2$. Lifting this to general latitudes is done by dividing A_i through this value and obtaining

$$N_{\phi,i} = 2N_\theta \sin\theta_i \tag{4}$$

(rounded up or down) subdivisions along the azimuths for approximately equal areas of the tiles:

$$\Delta\phi_i = \frac{2\pi}{N_{\phi,i}}; \tag{5}$$

$$\phi_j = (j+\frac{1}{2})\Delta\phi_i; \quad 0 \le j < N_{\phi,i}. \tag{6}$$

This defines one convenient subdivision of the sphere surface into spherical quadrangles centered at polar coordinates (θ_i, ϕ_j) with edge lengths $\Delta\theta$ and $\Delta\phi_i$. The computation below does not require such a semi-regular pattern and is valid for more general tilings, provided that their individual areas can be written as square intervals with vertices at $\theta_i \pm \Delta\theta/2$ and $\phi_j \pm \Delta\phi_i/2$ in the polar coordinate system.

A tiling of this kind is illustrated in Figure 1.

III. TILE INTEGRALS

The description in the introductory section states that the volume inside the intersecting spheres is the sum over all spheres s and for each sphere a sum over visible tiles with centers at θ_i and ϕ_j:

$$V = \sum_s \sum_{ij} V_s. \tag{7}$$

The vector to the surface of the tile is $(\mathbf{c}_s + \rho_s\hat{\mathbf{r}}_s)$ where $\hat{\mathbf{r}}_s$ is a unit vector starting at the sphere center, so the integral representation is

$$V_s = \frac{1}{3}\iint_{\text{vis}} (\mathbf{c}_s + \rho_s\hat{\mathbf{r}}_s)\cdot\hat{\mathbf{r}}_s d^2s \tag{8}$$

$$= \frac{1}{3}\iint_{\text{vis}} \mathbf{c}_s\cdot\hat{\mathbf{r}}_s d^2s + \frac{\rho_s}{3}\iint_{\text{vis}} d^2s. \tag{9}$$

*URL: http://www.strw.leidenuniv.nl/~mathar

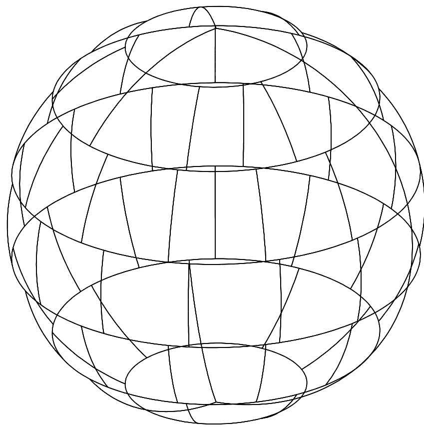

FIG. 1: Example of a tessellation of the sphere surface with spherical quadrangles as proposed in section II: $N_\theta = 7$ with rounding to the next integer in (4).

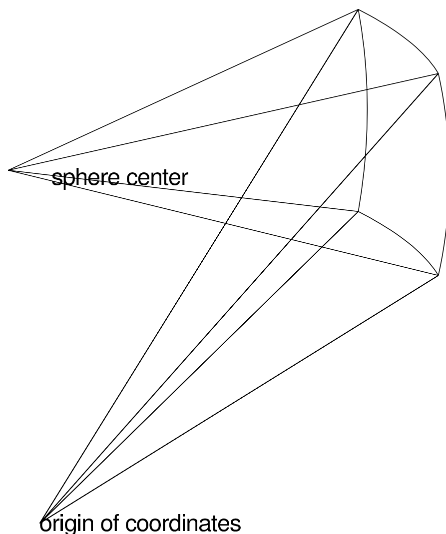

FIG. 2: Equation (9) computes the signed volume within the trunk with four planar "pyramidal" faces meeting at the center of coordinates and one spherical tile of an off-axis sphere defining the fifth surface.

Inverse interpretation with the divergence theorem demonstrates that V_s is the volume inside the five-faced surface shown in Figure 2, four straight faces that intersect at the origin, and the fifth the piece of sphere surface that is inside the tile. The volume is *signed* and becomes negative if the sphere normal points towards the origin.

Recognizing the surface element in polar coordinates

$$d^2s = \rho^2 \sin\theta d\theta d\phi, \tag{10}$$

the second integral on the right hand side of (9) and its Taylor expansion are

$$\begin{aligned} \frac{\rho_s}{3}\iint_{vis} d^2s &= \frac{\rho_s^3}{3}\int_{\theta_i-\Delta\theta/2}^{\theta_i+\Delta\theta/2}\sin\theta d\theta\int_{\phi_j-\Delta\phi_i/2}^{\phi_j+\Delta\phi_i/2}d\phi \quad (11)\\ &= \frac{\rho_s^3}{3}\Delta\phi_i\left[\cos(\theta_i-\frac{\Delta\theta}{2})-\cos(\theta_i+\frac{\Delta\theta}{2})\right]\\ &= \frac{2\rho_s^3}{3}\Delta\phi_i\sin(\theta_i)\sin(\Delta\theta/2) \quad (12)\end{aligned}$$

$$= \frac{\rho_s^3}{3}\Delta\phi_i\Delta\theta_i\sin\theta_i[1-\frac{(\Delta\theta_i)^2}{24}+\frac{(\Delta\theta_i)^4}{1920}-\frac{(\Delta\theta_i)^6}{322560}+\cdots]$$

The first integral in (9) is rendered by introducing the spherical coordinates of the center $\mathbf{c}_s$

$$\mathbf{c}_s = c_s\begin{pmatrix}\cos\phi_s\sin\theta_s\\ \sin\phi_s\sin\theta_s\\ \cos\theta_s\end{pmatrix} = \begin{pmatrix}c_x\\ c_y\\ c_z\end{pmatrix}, \tag{13}$$

$$\begin{aligned}\iint \mathbf{c}_i\cdot\hat{\mathbf{r}}d^2s = \rho_s^2\int_{\theta_i-\Delta\theta_i/2}^{\theta_i-\Delta\theta_i/2}\sin\theta d\theta\int_{\phi_j-\Delta\phi_i/2}^{\phi_j-\Delta\phi_i/2}d\phi\\ \times[c_x\cos\phi\sin\theta+c_y\sin\phi\sin\theta+c_z\cos\theta] \quad (14)\end{aligned}$$

$$= \rho_s^2[c_xI_x+c_yI_y+c_zI_z], \tag{15}$$

where the three components of I are elementary trigonometric integrals found to be

$$\begin{aligned}I_x &= \frac{1}{2}[\sin(\phi_i+\Delta\phi_i/2)-\sin(\phi_i-\Delta\phi_i/2)]\\ &\times[\frac{1}{2}\sin(2\theta_i-\Delta\theta)-\frac{1}{2}\sin(2\theta_i+\Delta\theta)+\Delta\theta], \quad (16)\end{aligned}$$

$$\begin{aligned}I_y &= \sin(\phi_i)\sin(\Delta\phi_i/2)\\ &\times[\frac{1}{2}\sin(2\theta_i-\Delta\theta)-\frac{1}{2}\sin(2\theta_i+\Delta\theta)+\Delta\theta] \quad (17)\end{aligned}$$

and

$$I_z = \frac{\Delta\phi_i}{2}\sin(2\theta_i)\sin(\Delta\theta). \tag{18}$$

In practise, the orientation of the polar axis of the coordinate system is an essentially free parameter if one is integrating over all surfaces anyway; this could be used to align the axis with the direction of $\mathbf{c}_s$, such that $\theta_s = 0$ and only the value of I_z is needed.

IV. C++ IMPLEMENTATION

These formulas are the core of an implementation which is listed in the appendix in a `makefile`, C++ header files `*.h` and functions `*.cxx`, and two examples of input files `*.txt`.

Once compiled, the main function `SphereSurf` is called with one argument, which is the name of a file which contains four blank-separated floating point numbers per line, Cartesian coordinates c_x, c_y and c_z and radius ρ_s for a sphere, one line per sphere.

The only nontrivial aspect is the function `SphereVect::volume` which works as follows: in an outer loop over the polar angles with index `i` we generate an inner loop over the azimuth angles with index `j`. The products of trigonometric functions are stored in the variables `IxyTheta`, `IzTheta` and `I`. In the innermost loop over all spheres `s`, we determine if the tile center is inside some other sphere. If it is not, the two contributions from (12) and (15) are added to the result. This decision on whether a point on one sphere is an interior point of any other sphere lets the computational expense grow proportional to the square of the number of spheres.

(The order of the loops could be permuted. The decision here was based on the assumption that early and therefore low-frequent computations of the common trigonometric functions may speed up the computation.)

The implementation is a finite element method in the sense that (i) the center of each sphere is used as an indicator of visibility representing the *entire* tile, so the exact boundaries of the sphere intersections are not sensed and (ii) the integration of the function within each tile is exact.

Appendix A: makefile

```
CXXFLAGS = -O
SphereSurf: SphereSurf.cxx Point3D.o Sphere.o SphereVect.o
	$(CXX) $(CXXFLAGS) -o $@ SphereSurf.cxx Point3D.o Sphere.o SphereVect.o

Point3D.o : Point3D.h Point3D.cxx
	$(CXX) $(CXXFLAGS) -c -o $@ Point3D.cxx

Sphere.o : Sphere.h Sphere.cxx
	$(CXX) $(CXXFLAGS) -c -o $@ Sphere.cxx

SphereVect.o : SphereVect.h SphereVect.cxx
	$(CXX) $(CXXFLAGS) -c -o $@ SphereVect.cxx

# two known test cases
tst: SphereSurf
	SphereSurf SphereSurf2.txt
	SphereSurf SphereSurf3.txt
```

Appendix B: SphereSurf.cxx

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>
#include "SphereVect.h"

using namespace std;

/** Main invocation of the program.
* Takes one argumet (file name with sphere definitions)
* and enters a loop with refined tesselations from 5 up to 150 polar angles.
*/
int main(int argc, char *argv[])
{
	if ( argc != 2)
	{
		cerr << "usage: " << argv[0] << " file-name-with-spheres " << endl;
```

```
                return 1 ;
        }

        /* Read the sphere coordinates from the file.
        */
        SphereVect sps(argv[1]) ;

        /* Provide an exact answer if there are only two spheres
        */
        if ( sps.spheres.size() == 2)
        {
                double v = sps.volume(sps.spheres[0],sps.spheres[1]) ;
                cout << "# " << setprecision(14)<<  v << " exact result with two spheres " << endl ;
        }
        else
                cout << "# " << sps.spheres.size() << " spheres "<<  endl ;

        cout << "# " << setprecision(14) << sps.volumeIsol() << " non-intersecting" << endl ;
        cout << "# " << setprecision(14) << sps.volume2Clust() << " in 2-clusters" << endl ;

        for(int ntess = 4 ; ntess < 150 ; ntess *= 2)
        {
                /* tabulate the convergence, number of tiles and volume estimation
                */
                double v = sps.volume(ntess) ;
                cout << ntess << " " << setprecision(14) << v << endl ;
        }
        return 0 ;
} /* main */
```

Appendix C: Point3D.h

```
#ifndef POINT3D_H
#define POINT3D_H
class Point3D {
public:
        /** Three cartesian coordinates
        */
        double coo[3] ;

        Point3D(double x, double y, double z) ;

        Point3D add(const Point3D & oth) const ;

        double distance(const Point3D & oth) const ;
protected:
private:
} ;

Point3D operator* (const Point3D & left, double f) ;
#endif /* POINT3D_H */
```

Appendix D: Point3D.cxx

```
#include <cmath>
#include "Point3D.h"
```

```
/** ctor with three cartesian coordinates
* @param x X coordinate.
* @param y Y coordinate.
* @param z Z coordinate.
*/
Point3D::Point3D(double x, double y, double z)
{
        coo[0] = x ;
        coo[1] = y ;
        coo[2] = z ;
}

/** Add (translate by) another vector
* @param oth The amount of translation.
* @return The sum of the two points considered as vectors.
*/
Point3D Point3D::add(const Point3D & oth) const
{
        return Point3D(coo[0]+oth.coo[0], coo[1]+oth.coo[1], coo[2]+oth.coo[2]) ;
}

/** Distance to another point.
* @param oth The other point.
* @return The Euclidean distance between this and oth.
*/
double Point3D::distance(const Point3D & oth) const
{
        return hypot(hypot(coo[0]-oth.coo[0], coo[1]-oth.coo[1]), coo[2]-oth.coo[2]) ;
}

/** Multiply (stretch) by a factor
* @param left The point to the left of the multiplication sign.
* @param f The factor to the right.
* @return A point that has all three Cartesian components multiplied by f.
*/
Point3D operator* (const Point3D & left, double f)
{
        return Point3D(left.coo[0]*f, left.coo[1]*f, left.coo[2]*f) ;
}
```

Appendix E: Sphere.h

```
#ifndef SPHERE_H
#define SPHERE_H
#include "Point3D.h"
class Sphere : public Point3D {
public:
        /** Radius
        */
        double radius ;

        Sphere(double x, double y, double z, double r) ;

        bool intersects(const Sphere & oth) const ;

        bool contains(const Point3D & pt) const ;
```

```
        double volume() const ;

        double volumeInters(const Sphere &oth) const ;
protected:
private:
} ;
#endif /* SPHERE_H */
```

Appendix F: Sphere.cxx

```
#include <cmath>
#include "Sphere.h"

/** ctor with three cartesian coordinates
* @param x X center coordinate.
* @param y Y center coordinate.
* @param z Z center coordinate.
* @param r Radius.
*/
Sphere::Sphere(double x, double y, double z, double r) : Point3D(x,y,z), radius(r)
{
}

/** True if the sphere intersects the other sphere.
* @param oth The sphere which may collide/intersect with this one.
* @return True if the distance between the two sphere centers is less than the sum
*  of the radii.
*/
bool Sphere::intersects(const Sphere &oth) const
{
        return ( distance(oth) < radius + oth.radius ) ;
}

/** True if the point is inside the sphere.
* @param pt The point to be tested.
* @return True if the distance between point and sphere center is less than the sphere radius.
*/
bool Sphere::contains(const Point3D &pt) const
{
        return ( distance(pt) <= radius ) ;
}

/** Volume of the sphere.
* @return 4/3 times pi times radius cubed.
*/
double Sphere::volume() const
{
        return 4.*M_PI* pow(radius,3.)/3. ;
}

/** Volume of the intersection between two spheres.
* @return The volume defined by common points inside the two polar caps.
* The result ranges from 0 (no intersection) up to the smaller of the two volumes
* (maximum overlap).
*/
double Sphere::volumeInters( const Sphere &oth) const
```

```
{
        if ( intersects(oth) )
        {
                /* distance between the two centers
                */
                double d = distance(oth) ;
                /* hypothenuse section along the line that connects the two centers
                * Plumb line from the line of intersection orthogonal to the axis between both centers.
                */
                double h = ( radius*radius -oth.radius*oth.radius+d*d)/(2.*d) ;
                if ( h <= -radius)
                        /* this sphere completely immersed in oth
                        */
                        return volume() ;
                else if ( h >= radius)
                        return 0. ;
                else
                        return -M_PI*(3.*pow(radius-oth.radius,2.)-2*d*(radius+oth.radius)-d*d)
                                *pow(radius+oth.radius-d,2.)/(12.*d) ;
        }
        else
                return 0. ;
}
```

Appendix G: SphereVect.h

```
#ifndef SPHEREVECT_H
#define SPHEREVECT_H
#include <vector>
#include <string>
#include "Sphere.h"

using namespace std;
class SphereVect {
public:
        /** a list of all spheres involved
        */
        vector<Sphere> spheres ;

        SphereVect(string fname) ;

        double volume(int tileN) const ;

        static double volume(const Sphere & sp1, const Sphere & sp2) ;

        double volumeIsol() const ;

        double volume2Clust() const ;
protected:
private:
} ;
#endif /* SPHEREVECT_H */
```

Appendix H: SphereVect.cxx

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <cmath>
#include "SphereVect.h"

using namespace std;

/** ctor with three coordinates read from a file
* @param fname The file which contains x, y, z and radius information.
*/
SphereVect::SphereVect(string fname)
{
        /* open file
        */
        ifstream inf(fname.c_str()) ;

        /* read line by line
        */
        for( ; ! inf.eof() ;)
        {
                string strinp ;
                getline(inf,strinp) ;
                /* skip comment lines (those that contain a numeral sign)
                */
                string::size_type pos = strinp.find("#") ;
                if ( pos != string::npos )
                        continue ;

                /* empty line, no input either
                */
                if ( strinp.size() == 0 )
                        continue ;

                /* read four floating point values and interpret them as x, y, z and radius
                */
                istringstream thisl(strinp) ;
                double xyzr[4] ;
                thisl >> xyzr[0] >> xyzr[1] >> xyzr[2] >> xyzr[3] ;
                Sphere s(xyzr[0],xyzr[1],xyzr[2],xyzr[3]) ;

                /* attach this sphere to the collection
                */
                spheres.push_back(s) ;
        }
}

/** Determine the combined volume by defining semi-regular tiles
* @param tileN The number of subdivisions of the polar angles.
* @return The estimate of the volume defined with the finite element approach
*  of surface integration.
*/
double SphereVect::volume(int tileN) const
{
        /* volume accumulated over tiles and spheres
        */
```

```
double v=0. ;

/* stride in polar angles.
*/
double dtheta = M_PI/tileN ;

/* loop over polar angles
*/
for(int i =0 ; i < tileN ; i++)
{
        /* center polar angle in tile.
        */
        double thetai = (i+0.5)*dtheta ;

        /* number of azimuths in that zone
        */
        int Nphi = 2*tileN*sin(thetai) ;

        /* stride of azimuth in that zone
        */
        double dphi = 2.*M_PI/Nphi ;

        /* main part of the integral related to center at origin.
        */
        double sinThInt = 2.*dphi*sin(thetai)*sin(0.5*dtheta) ;

        /* factor in the integrals Ix and Iy
        * double IxyTheta = dtheta + 0.5*( sin(2.*thetai-dtheta)-sin(2.*thetai+dtheta) ) ;
        * Replaced by a 7th order taylor expansion to avoid loss of precision.
        */
        double cos2theta = cos(2.*thetai) ;
        double IxyTheta = dtheta *(1.+cos2theta*(-1.+dtheta*dtheta*(1./6.+dtheta*dtheta
                              *(-1./120.+dtheta*dtheta/5040.)))) ;

        /* factor in the integral Iz
        */
        double IzTheta = sin(2.*thetai)*sin(dtheta) ;

        /* the three auxiliary integrals
        * and z-component, which does not depend on phi itself
        */
        double I[3] ;
        I[2] = 0.5*dphi*IzTheta ;

        /* loop over azimuths
        */
        for( int j=0 ; j < Nphi ; j++)
        {
                /* the center azimuth in the tile
                */
                double phi = (j+0.5)*dphi ;

                /* I[0] = 0.5*(sin(phi+0.5*dphi)-sin(phi-0.5*dphi))*IxyTheta ;
                * is replaced by its Taylor series, assuming dphi may become small.
                */
                I[0] = IxyTheta*cos(phi)*dphi*(1./2.+dphi*dphi*(-1./48.+dphi*dphi/3840.)) ;
                I[1] = sin(phi)*sin(dphi*0.5)*IxyTheta ;
```

```
                        /* construct unit vector radially outwards into the direction
                        * of the center of the tile
                        */
                        Point3D nunit(cos(phi)*sin(thetai), sin(phi)*sin(thetai), cos(thetai)) ;

                        /* loop over visible surface of spheres
                        */
                        for(int s=0 ; s < spheres.size() ; s++)
                        {
                                /* construct center point in tile
                                */
                                Point3D tilept = spheres[s].add(nunit * spheres[s].radius) ;

                                /* check whether this is inside any of the other spheres
                                */
                                bool isvisible = true ;
                                for(int sprime=0 ; sprime < spheres.size() && isvisible; sprime++)
                                {
                                        if ( sprime == s)
                                                continue ;
                                        if ( spheres[sprime].contains(tilept) )
                                                isvisible = false ;
                                }

                                /* if visible, add the contribution from this tile
                                */
                                if ( isvisible)
                                {
                                        /* the part from the term that does not depend on the center
                                        * coordinate
                                        */
                                        v += pow(spheres[s].radius,3.) * sinThInt ;
                                        /* the part from the term that is a dot product of the center
                                        * coordinate with three integrals.
                                        */
                                        v += pow(spheres[s].radius,2.) * (spheres[s].coo[0]* I[0]
                                                + spheres[s].coo[1]*I[1] + spheres[s].coo[2]*I[2]) ;
                                }
                        }
                }
        }
        /* factor 1/3 from the divergence theorem
        */
        return v/3.;
} /* SphereVect::volume */

/** Compute the volume within a pair of spheres
* @param sp1 The first sphere.
* @param sp2 The second sphere, possibly intersecting the first one.
* @return The sum of the volume of the individual spheres minus the
*  sum of the intersection.
*/
double SphereVect::volume(const Sphere &sp1, const Sphere &sp2)
{
        double v = sp1.volume() + sp2.volume() ;
        return v - sp1.volumeInters(sp2) ;
}
```

```
/** Determine the total volume of the non-intersecting spheres.
* This is the term that appears as the uncorrelated V(1) if the
* volume of the intersecting spheres is expanded as V(1)-V(2)+V(3)-V(4)+...
* by the inclusion-exclusion principle.
* @return The sum of the individual volumes of all spheres.
*/
double SphereVect::volumeIsol() const
{
        double v=0. ;
        for(int s=0 ; s < spheres.size() ; s++)
                v += spheres[s].volume() ;
        return v;
}

/** Determine the volume in the intersections of all pairs of spheres.
* This is the term that appears as the correlation V(2) if the
* volume of the intersecting spheres is expanded as V(1)-V(2)+V(3)-V(4)+...
* by the inclusion-exclusion principle.
* @return The sum of the individual volumes of all sphers.
*/
double SphereVect::volume2Clust() const
{
        double v=0. ;
        /* scan all pairs (each pair counted once)
        */
        for(int s=0 ; s < spheres.size()-1 ; s++)
                for(int sprime=s+1 ; sprime < spheres.size() ; sprime++)
                        v += spheres[s].volumeInters(spheres[sprime]) ;
        return v;
}
```

Appendix I: SphereSurf2.txt

```
# test file SphereSurf with two spheres
0 0 0 1.
2. 0. 0. 2.5
```

Appendix J: SphereSurf3.txt

```
# test file SphereSurf with three spheres and 0.5736 volume of the triple intersection
# as in Gibson and Sherage, JCP 91 (1987) 4121.
# The exact volume is therefore 150.7964 -7.00319 +0.5736 = 144.36685
0 0 0 1.
2. 0. 0. 2.
-0.75 2.90473 0. 3.
```

---

[1] M. L. Connolly, J. Am. Chem. Soc. **107**, 1118 (1985).
[2] R. Pavani and G. Ranghino, Comput. Chem. **6**, 133 (1982).
[3] T. J. Richmond, J. Mol. Biol. **178**, 63 (1984).
[4] J. A. Grant and B. T. Pickup, J. Phys. Chem. **99**, 3503 (1995).
[5] J. Buša, J. Džurina, E. Hayryan, S. Hayryan, C.-K. Hu, J. Plavka, I. Pokomý, J. Skřivánek, and M.-C. Wu, Comp. Phys. Commun. **165**, 59 (2005).
[6] K. D. Gibson and H. A. Scheraga, J. Phys. Chem. **91**, 4121 (1987).

[7] M. Petitjean, Int. J. Quant. Chem. **15**, 507 (1994).