# Visualising Protein Surfaces

*Guillaume Valadon*

*June 14, 2002*

## Abstract

Molecular modelling softwares helps humans to easily see molecular structures using three-dimensional graphics. The aim of this project is to add new display mode to an existing software to visualize physics properties and internal molecular representations of this software.

# Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date:

# Acknowledgements

I would like to thank my supervisor Dr Dave Ricthie for all his devotion helping me on this project, and for giving me a lot of his time debugging it. His knowledge in biochemistry and computing helped me understand what I was doing.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Project motivation

Complex molecular structures such as proteins or DNA are not easy to visualize by humans. Molecular modelling softwares display these molecules using 3D graphics. They usually provide a lot of differents ways to represent these structures using spheres for each atoms or three-dimensional surfaces defined by van der Waals forces. *Hex* which was written by Dave Ritchie[1] can compute and display such surfaces.

Adding more surfaces colouring mode to that existing software will be useful to easily visualize molecules physics properties or display internal *hex* molecules representation.

## 1.2 Projet objectives

As *hex* is written using C, Motif and openGL under Unix, I have to use this language and those libraries for my project.

I will have to work with an existing code, it is not so vulgar because *hex* is a big software, sources are nearly three Mega Bytes. One of the main difficulties is to learn enough about *hex* structures and logic in order to put my modifications in the good place. The other difficulty is to provide a code as simple as possible to keep the same computation speed.

My work for this project is to implement three new display mode :

1. colouring actual surface by atoms colors
2. colouring actual surface by electrostatics forces
3. displaying and colouring internal hex surfaces used in docking computations

The first mode will permit to easily find chemicals groups and structures on molecules surface. If a *hex* user is looking for a particular amino acid group in the molecule, he can quickly find it by looking for atoms colors around the surface. With the cysteine molecule, one can identify that the side chain contains a sulfur atom, it is the yellow part on figure 1.1.

---

[1] `dritchie@csd.abdn.ac.uk`

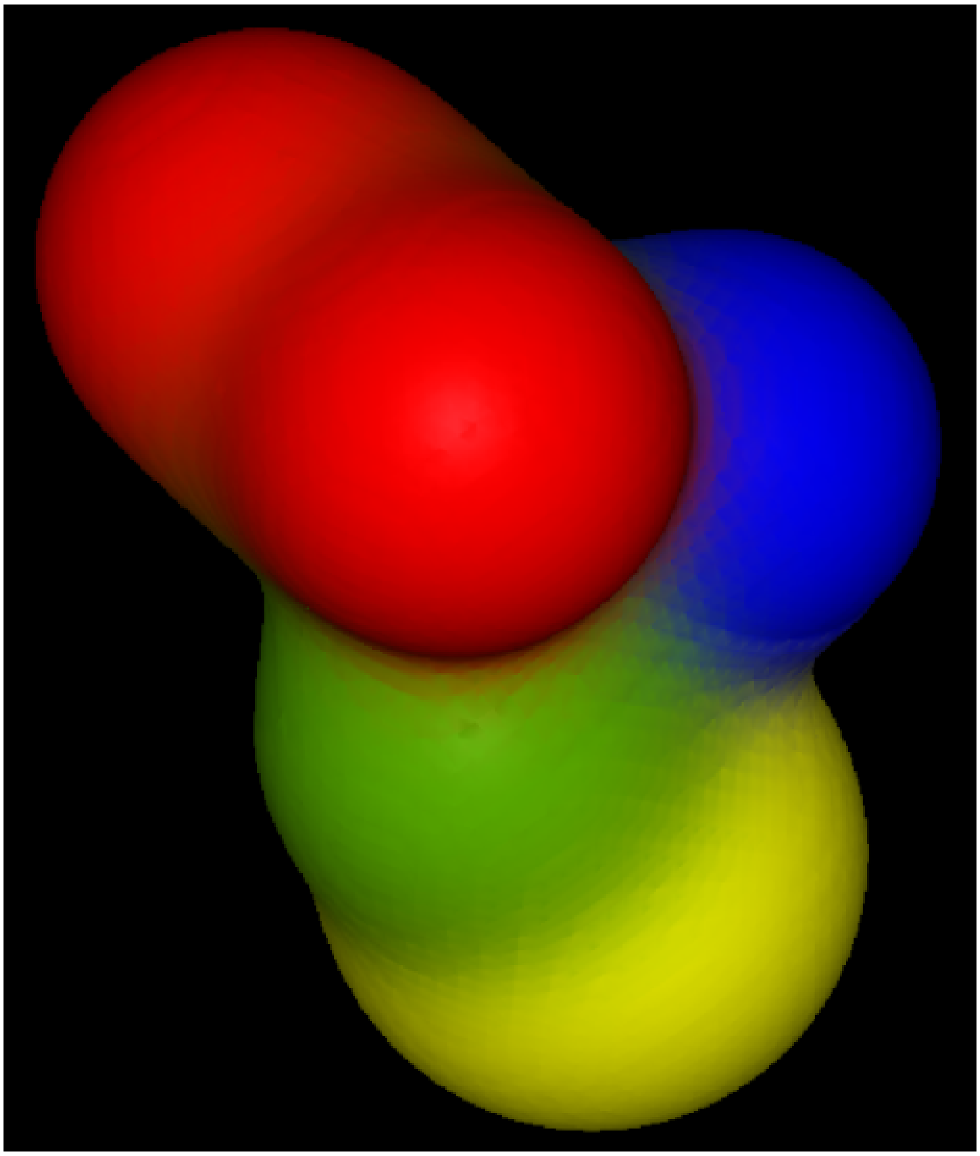


Figure 1.1: Cysteine molecule: atoms colors.

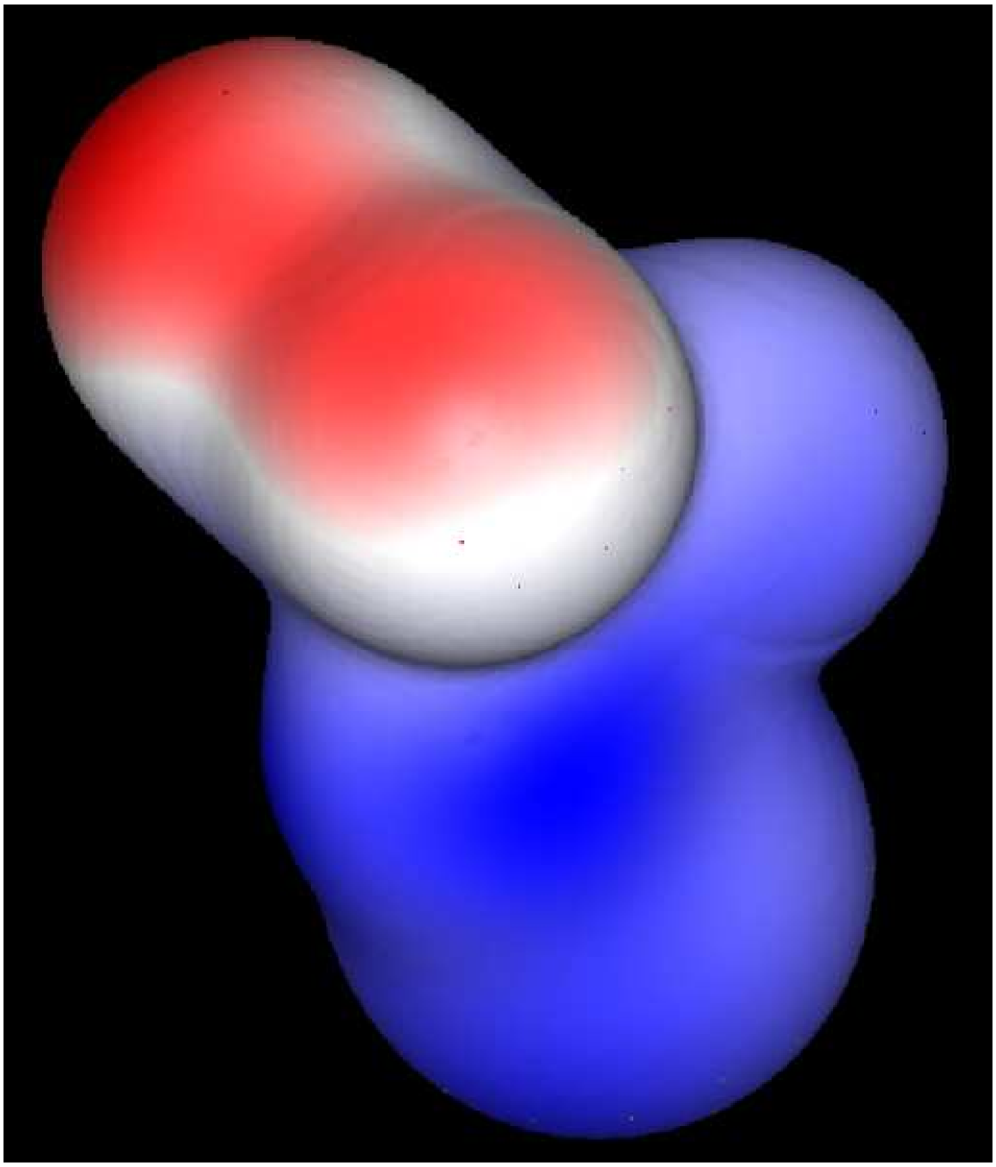


Figure 1.2: Cysteine molecule: electrostatic densities.

The second colouring mode will draw molecule's electrostatic potential or density on the surface. It will be easier to see wich parts of one molecule are more likely to interact with others. On figure 1.2, red stands for negative values and blue for positives ones.

The last mode will provide an helpfull view of internal surfaces representation. It will be very useful to visually verify if the surface approximation made by *hex* is correct by comparing it with the classic van der Waals surface.

## 1.3 Report structure

This report is organized as follow :

In the background chapter page 11, I describe the basic knowledge the reader must have to understand the implementation of colouring modes in *hex* . I first explain physics involved in colouring mode and then how *hex* is organized and works to display molecules in 3D.

In the implementation chapter page 20, I explain in details how and why I modify *hex* to achieve the implementation of colouring modes.

In the results chapter page 31, the reader could see screenshots from the modified and the actual version of *hex* in order to compare them. I also provide computation times of my code and explain why they are good.

Finally in the conclusion page 44, I will explain what I learnt and how that project is useful for me.

The code listing is in appendix **??** and is a commented output off the diff unix command. As I added the new code in a lot of different places providing that kind of listing is better because the original *hex* code does not appear.

## Chapter 2

# Background

In that chapter, I will describe what the reader must know to understand the implementation of this project. First, I will provide some useful explanation about forces in molecules. Then I will show how *hex* works and which functions I will use in the implementation chapter page 20.

## 2.1 Biophysics

Forces that provides stability and specificity to three-dimensionnal structures of molecules are mostly interactions which are individually weak but collectively strong. In that section, I will briefly describe two of them: van der Waals and electrostatics interactions.

### 2.1.1 Van der Waals interaction

All atoms and molecules attract each other; even wihtout charges. The size of an atom is described by the van der waals radius and is specific to each type of atoms, see table 2.1. The interaction increase as two non-bonded atoms are closer together. The maximum interaction is reached when the distance between them is equal to the sum of theirs van der Waals radii. If the atoms are brought still closer together there is strong van der Waals repulsion.

| atom | van der Waals radius (in nm) |
|---|---|
| hydrogen | 0.1 |
| oxygen | 0.14 |
| nytrogen | 0.15 |
| carbon | 0.17 |
| sulfur | 0.18 |

Figure 2.1: Van der Waals radii.

In molecular modelisation softwares, the van der waals radius is often used to draw molecules in a hard sphere mode, see figure 2.2. Each atom in the molecule will be represent as a sphere which radius is proportional to its van der waals radius.

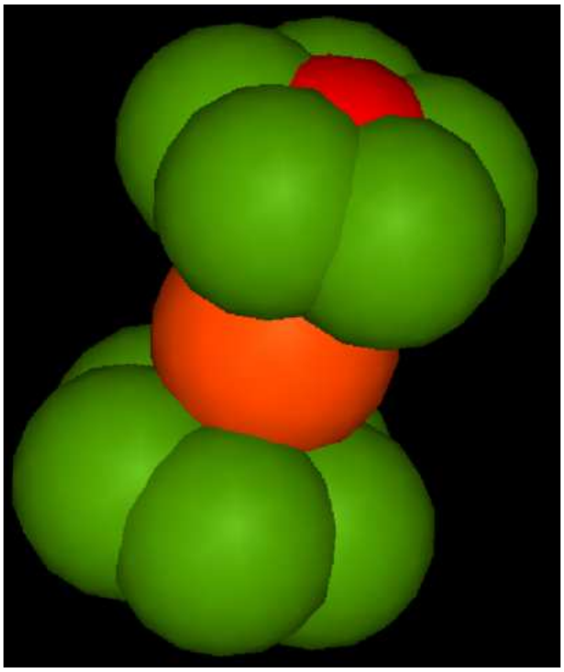


Figure 2.2: Ferrocene molecule: hard sphere mode.

### 2.1.2 Electrostatics interaction

An electrostatic interaction depends on the electric charges on atoms. The energy of an electrostatic interaction is given by the Coulomb's law:

$$\mathtt{E} = \mathcal{K} * \frac{q_A * q_B}{D_{AB}} \tag{2.1}$$

Where $\mathcal{K}$ is a constant, $q_A$ and $q_B$ the charges of the two atoms A and B, and $D_{AB}$ the distance between them. See figure 2.3 for a graphical representation of this values.

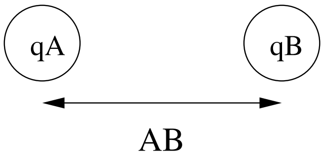



Figure 2.3: Coulomb's law between two charges A and B.

With that law, it is impossible to represent interactions between more than two atoms, to deal with that problem, the energy will now be compute using density ($\rho(\underline{x})$) and potential ($\Phi(\underline{x})$) functions:

$$\mathtt{E} = \mathcal{K} * \int \rho(\underline{x}) * \Phi(\underline{x}) * dV \tag{2.2}$$

The figure 2.4 shows what is $\rho(\underline{x})$. It represents densities of all charges in a group of atoms. As both $\rho$ and $\Phi$ can determine a specific value at one point of space, they will be used to implement the electrosatic colouring mode.

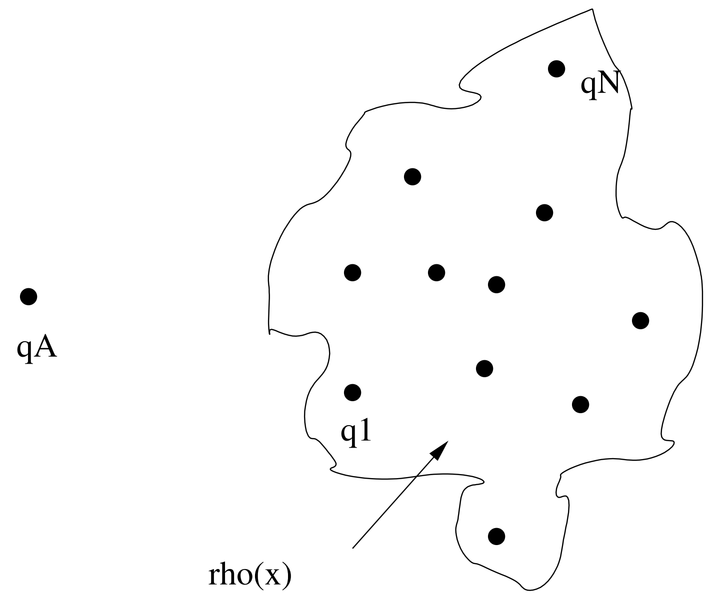



Figure 2.4: Representation of atom density.

## 2.2 Hex

In order to modify *hex* and start working on it, I first had to be familiar with the existing source code. In that section, I will briefly describe how it is organized and the process of displaying a molecule surface.

### 2.2.1 Source code

The source code is arranged in sub-folders in the **src3** folder (see figure 2.5), each one containing files regrouped by functionnalities.

The **all** folder regroups all .c and .h files, and provides an easy way to edit and search all files in the same place. I mainly worked with the **mol** folder that contains the code for molecules surfaces computation and display, and with the **gui** folder which contains all components of *hex* Graphic User Interface (G.U.I).

I had to stay coherent with the current coding style in order to keep my modifications as invisible as possible. To achieve that constraint, I tried to respect these points :

1. little use of *extern* variables
2. alignement of code to improve its reading, see figure 2.6
3. spaces between word and brackets in control flow statements but not in functions, see figure 2.7
4. shift witdh of three spaces, see figures 2.6 and 2.7

The file **hex.h** in the **include** folder contains prototypes for *hex* functions as well as definitions of *hex* data types. They are used to internally represent and compute molecules data, and to

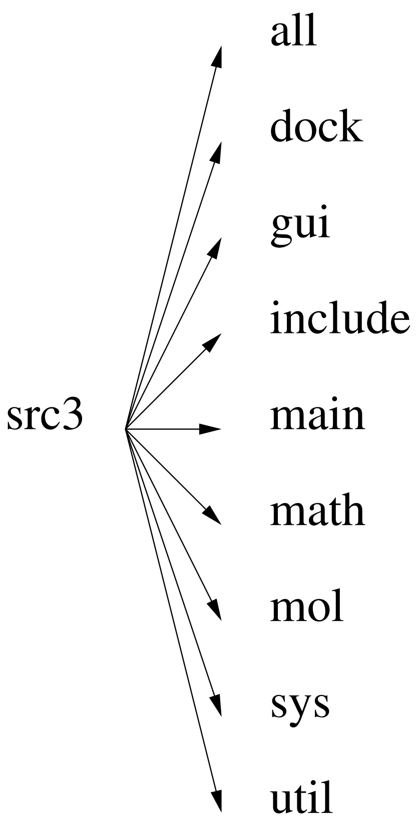



Figure 2.5: *Hex* sources tree

```
...k1 = molecule->msurf.triangles[3*i+0];
...k2 = molecule->msurf.triangles[3*i+1];
...k3 = molecule->msurf.triangles[3*i+2];

...set_red  (&ret, (int) colV.x);
...set_green(&ret, (int) colV.y);
...set_blue (&ret, (int) colV.z);
```

Figure 2.6: *Hex* coding style: alignement.

trigger a new computation when needed.

### 2.2.2 Common functions and jacket routines

*Hex* uses a lot of functions to save typing and to provide nice functionnalities to the programmer.

As 3D graphics deals with geometry in space, there is a lot of geometric functions defined in **geom/hex_geom.c**. It's very easy to do calculations as determining the angle between three points with *a_ptptpt()* or finding the normal to a plane with *v_n_pl()*, see figure 2.8

There is also jacket routines for memory management, to allocate and free shared and virtual memory. They work the same way as usuals libc functions *malloc()* and *free()* but provides nice features as the possibility to fill the allocated memory with zeros (*hex_vm_get0()*) and to prevent

```
...switch (molecule->mesh.colour)
...{
..o...case 0:
..o..o...pp = pp_pt(pt_fpt(molecule->msurf.vertices[k1]));
..o..o...set_red(&colour, (int) (255 * (0.2 + pp.theta*0.5/PI)));
...}
```

Figure 2.7: *Hex* coding style: use of spaces.

```
Angle angle;
Point p1, p2, p3;
p1.x = 1; p1.y = 0; p1.z = 0;
p2.x = 0; p2.y = 0; p2.z = 0;
p3.x = 0; p3.y = 0; p3.z = 1;

angle = a_ptptpt(p1, p2, p3);
```

Figure 2.8: Jackets routines: a_ptptpt().

freeing a NULL pointer (*hex_vm_wipe()*). They also provide memory usage statistics and nice error messages.

In the same way, there is also bindings for openGL calls. They are useful in order to directly use *hex* data representation, see figure 2.9.

```
FVector normal;
normal.x = 0;
normal.y = 0;
normal.z = 1;

hex_glNormalF(normal);
```

Figure 2.9: OpenGL and *hex* data types.

Here *hex_glNormalF()* will call the GL function *glNormal3d()* and will cast floats into doubles.

### 2.2.3 Display process

Here, I will describe all the functions *hex* uses to find a molecule surface using a contouring algorihtm and to display it using openGL.

The first step is to determine the surface with the *hex_surface()* function. It calls *hex_sample_grid()* to build a grid containing gaussian atoms densities values calculating with he Grant and Gallardo method, see (Grant *et al.*, 1996). For each atoms in the molecule, this function will evaluate a density at each vertex of the grid an atom can reach, see figure 2.11. That density function decrease as the distance from the atom center increase and is described by the following generic formula, where $\beta$ and $\alpha$ are constants and $R_{vdw}$ is the van der Waals radius of that atom.

(See figure 2.10 for a graphical representation of that function).

$$\rho_{atom}(d) = \beta * \mathrm{e}^{-\frac{\alpha}{R_{vdw}^2} * d^2} \tag{2.3}$$

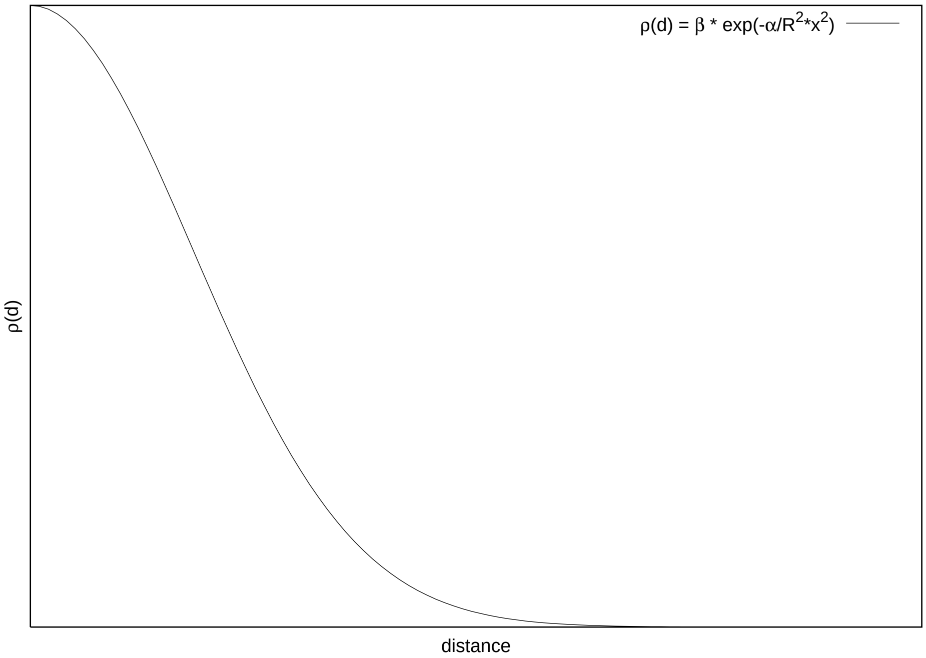



Figure 2.10: Graphical representation of density function using gnuplot

Once the density grid is built, the *hex_contour()* function is called. It will produce a list of vertices and triangles representing the molecule surface using the algorithm describe in the Guézziec and Hummel paper, see (Guéziec and Hummel, 1995) and a contouring value dynamically modified by user parameters. Usually its value is 0.25899, that the value returned by $\rho(d)$ when $d$ equals $R_{vdw}$. The main ideas of that algorithm is to divide a cube into tetrahedras see figure 2.12, find the tetrahedra that contains the contouring level see figure 2.13, and finally use an interpolation along edges and diagonals to find the contouring level and the triangle in the molecule surface, see figure 2.14.

The density value along one edge of the cube is given by a linear interpolation between two intensity values:

$$\rho(d) = \rho_A + \frac{\|AD\|}{\|AB\|} * (\rho_B - \rho_A) \tag{2.4}$$

Whereas that along a diagonal edge (A, D), the interpolation is quadratic and use all intensity values (A, B, C, D) on a face:

$$\rho(d) = (\rho_A + \rho_D - \rho_B - \rho_C) * d^2 + (-2 * \rho_A + \rho_b + \rho_c) * d + \rho_A \tag{2.5}$$

The molecule's surface is now determined and the final step is to display it using openGL. That is done in *hex_draw_surface()* using the list of triangles found by *hex_contour()* and the simple algorithm:

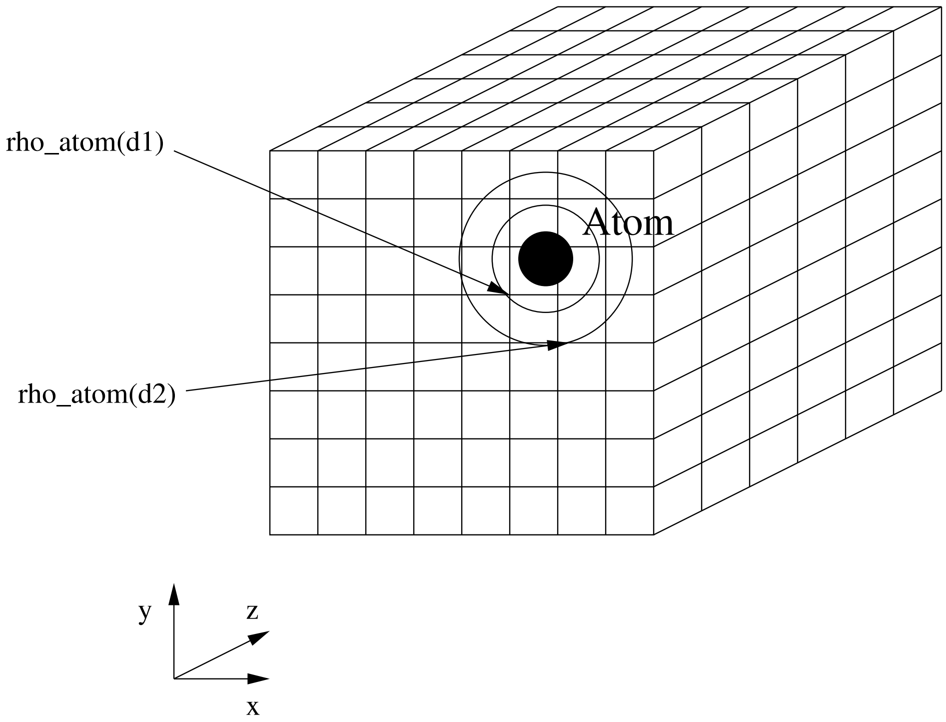
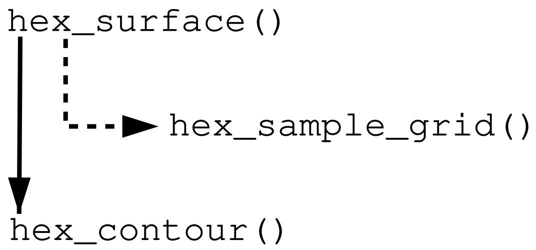



Figure 2.11: Atom's contribution to the density grid.

**for all** triangles in the list **do**
    draw_the_triangle()
**end for**

The displaying process can thus be described by the following diagram :




In the implementation chapter page 20, I will use that diagram to easily show the reader where I have modified *hex* to add colouring modes.

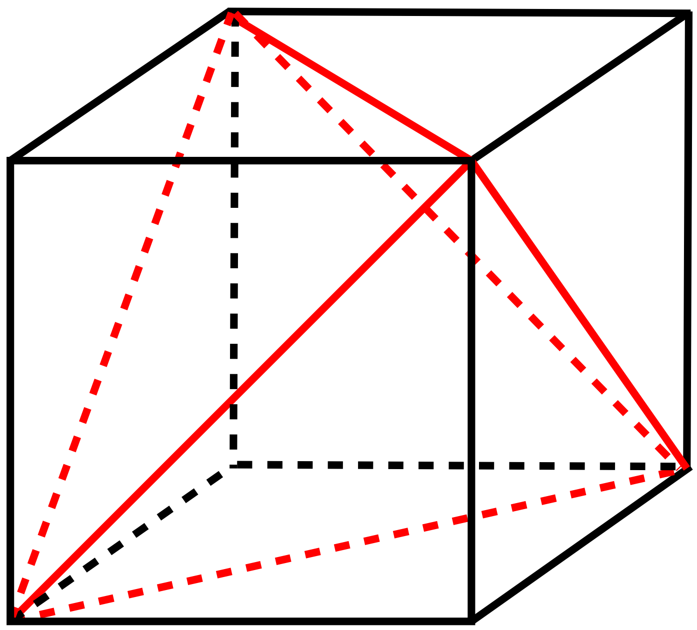


Figure 2.12: Dividing a cube into five tetrahedras.

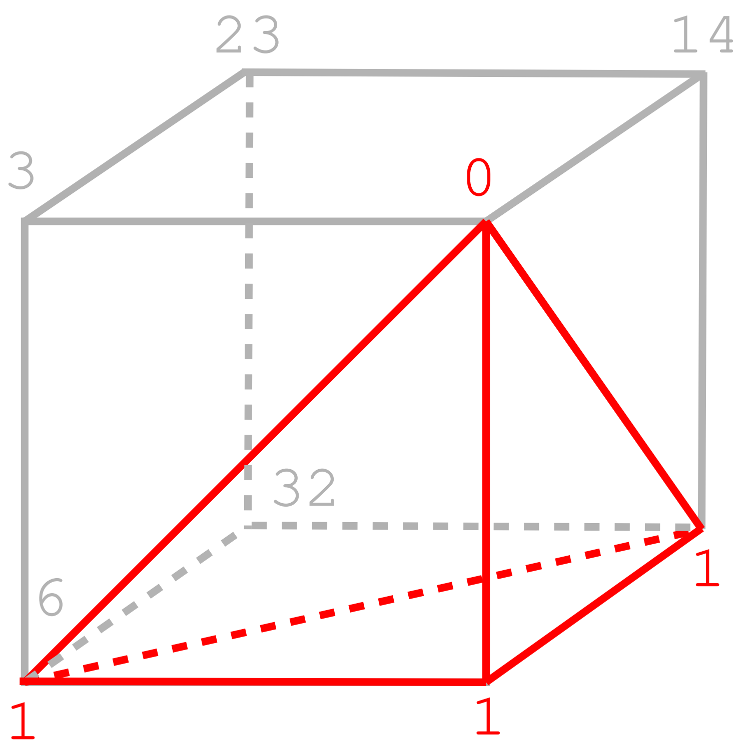



Figure 2.13: Selecting a tetrahedra for a contouring level of 0.25.

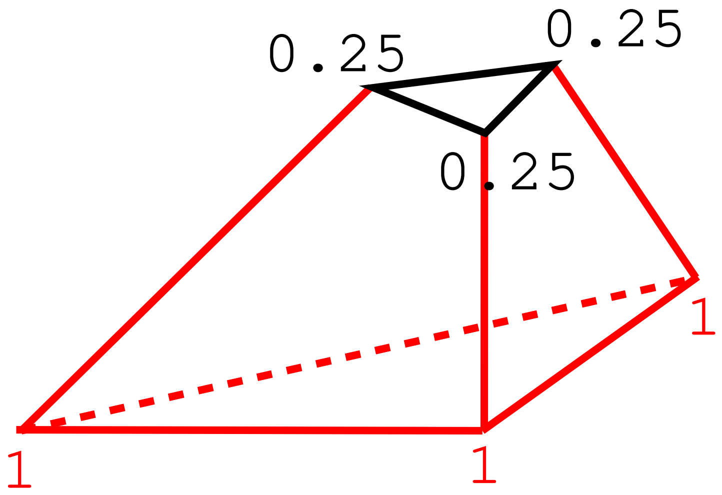



Figure 2.14: Surface's triangle for a contouring level of 0.25.

## Chapter 3

# Implementation

In that chapter, I describe how I modify *hex* to implement the three colouring modes.

## 3.1 Colouring surface by atoms color

The objective of that new colouring mode is to define a nicer default mode for surfaces display using atoms colors. I choosed to use the same process *hex* already used to build molecules surfaces adding it modifications to achieve this goal.

In order to colour the surface with openGL calls in *hex_draw_surface()* each vertex that defines the molecule's surface must have a color. As theses vertices are determined in *hex_contour()* using the density grid computed by *hex_sample_grid()* is an easy way to implement that colouring mode is to build another grid that will contain colors instead of density values, see figure 3.1.

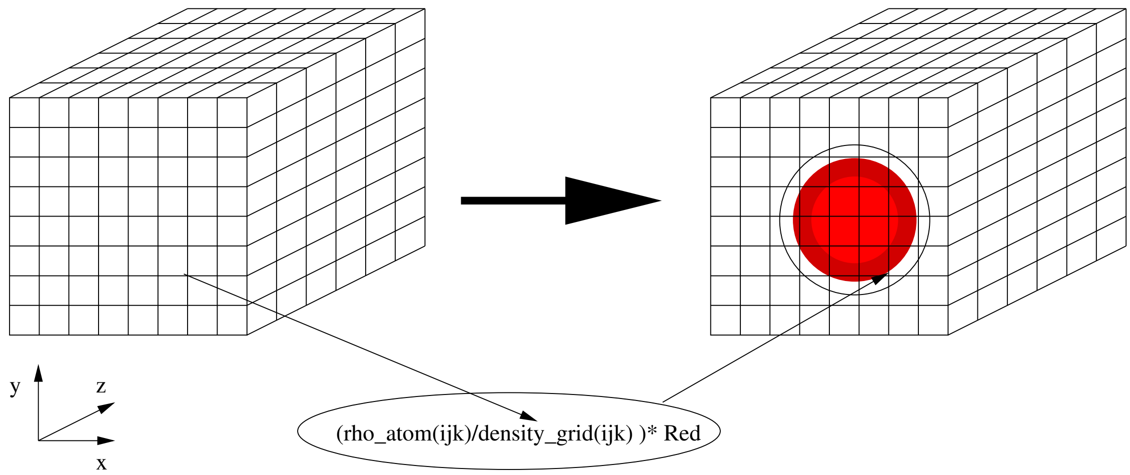



Figure 3.1: Using the density grid to build the colour grid.

Using only atoms colours to fill the grid will produce a naive visual effect as in figure 3.2. Colours are too bright, and the transition between two colours is too quick and not smooth. To suppress these effects, I choosed to use densities values in the classic grid to compute colours. Each atom will now contribute to each vertex it can *reach*, with its own colour multiplied by its density value at this point. This product is then divided by the total density at this vertex. The final vertex' colour is thus given by this mathematical formula:

$$\mathcal{C}_{ijk} = \sum_{0}^{N} \frac{\rho_N(ijk) * \mathcal{C}_N}{\rho_{Total}(ijk)} \tag{3.1}$$

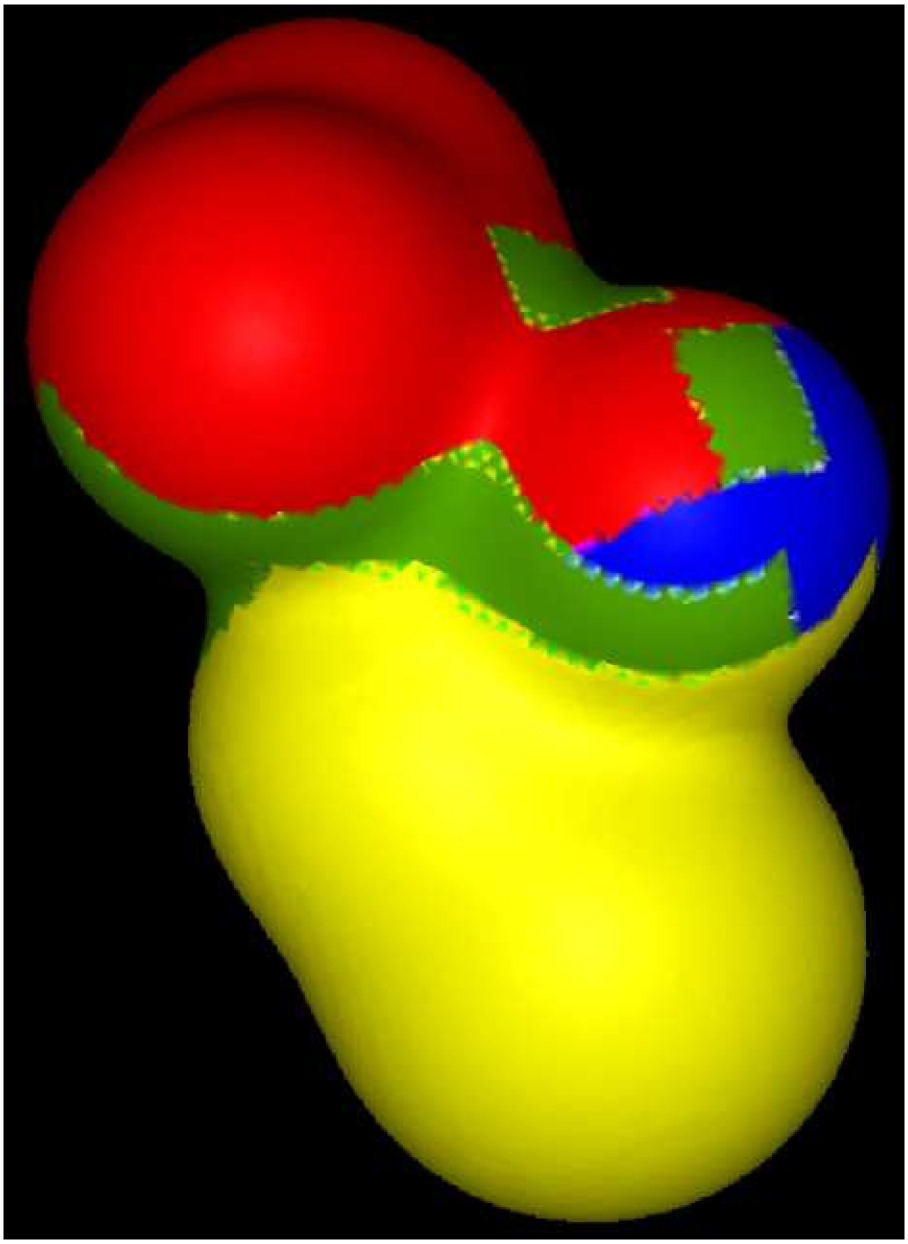


Figure 3.2: Cysteine molecule: naive colours.

It's important to understand that $N$ is the number of atoms that can contribute to the total density value at vertex $ijk$ and there is no need to evaluate the density function $\rho_N(ijk)$ if the atom can't *reach* that point in the grid. A better approach is to first find all vertices an atom can contribute to and then do the computation. Thus, the algorithm to build the colour grid is:

1: **for all** atoms in the molecule **do**
2: $\mathtt{L}_{ijk}$:= [list all vertices the selected atom can reach]

**for all** ijk in $\mathtt{L}_{ijk}$ **do**
density := $\rho_{atom}(ijk)$ {Evaluate the density function at that vertex}
$\mathcal{C}_{ijk} = density * \mathcal{C}_{atom}$ {Contribute to the final colour}
**end for**
**end for**

These loops are already used in *hex_sample_grid()* so I just added my code to that function to minimize the computation cost of the colouring surface. As you have maybe noticed, I do not divide by the total density in that algorithm. Remember that the colours grid is provided to *hex_contour()* to find colours of vertices that represents the surface. As the contouring algorithm deals with the densities grid, the divison will only occured when a vertex is selected to be in the molecule surface, i.e there is no useless divison.

Once *hex_contour()* found which vertices belong to the molecule's surface with the tetrahedral decomposition algorithm (see sub-section 15), it is possible to know between which two vertices `A` and `B` the final vertex `F` is. As we can easily access `A` and `B` colours in the colours grid, a simple linear interpolation can find the real colour of `F`.

$$\mathcal{C}_F = \mathcal{C}_A + \frac{\|AF\|}{\|AB\|} * (\mathcal{C}_B - \mathcal{C}_A) \tag{3.2}$$

In computer graphics, colours are often stored in RGB (Red Green Blue) components. Storing separately colours components for each vertex in the surface will use four times the actual memory. It's ok for small molecules but for bigger ones memory usage can be a real problem. I tried to get rid of that using the fact that range of RGB values is 0 to 255 and that nowadays most computers stores integers on at least 32 bits. As 8 bits are enough to store a number out of 255, it is possible to store colours components in the same integer using bits manipulation, see 3.4. The blue component is stored in bits from 0 to 7, the green one in bits from 8 to 15, and the red one in bits from 16 to 24. The memory usage will only be twice bigger as the actual one using one integers array for colour storage. Figure 3.3 shows the represensation of that storage in memory, and how theses components make a colour, whereas 3.5 show a possible usage of that idea.

The diagram in figure 3.6 easily shows where and why in modified *hex* displaying process to implement the atoms colours mode.

## 3.2 Colouring surface with electrostatics forces

The aim of that colouring mode is to use molecule's electrostatic potentials and densities to assign colours to the surface. The main work in that part was to build a nice motif widget to easily change the range of electrostatics values on the surface.

Electrostatic forces computation are already done in *hex* but are not visible. Dave Ritchie provides me a file called **hex_fnval.c** that regroups useful functions to achieve theses computations. My first job on that part was to identify which functions I should use.

Each of these computations relies on two functions *hex_setup_electro()* and *hex_electro_fn()*

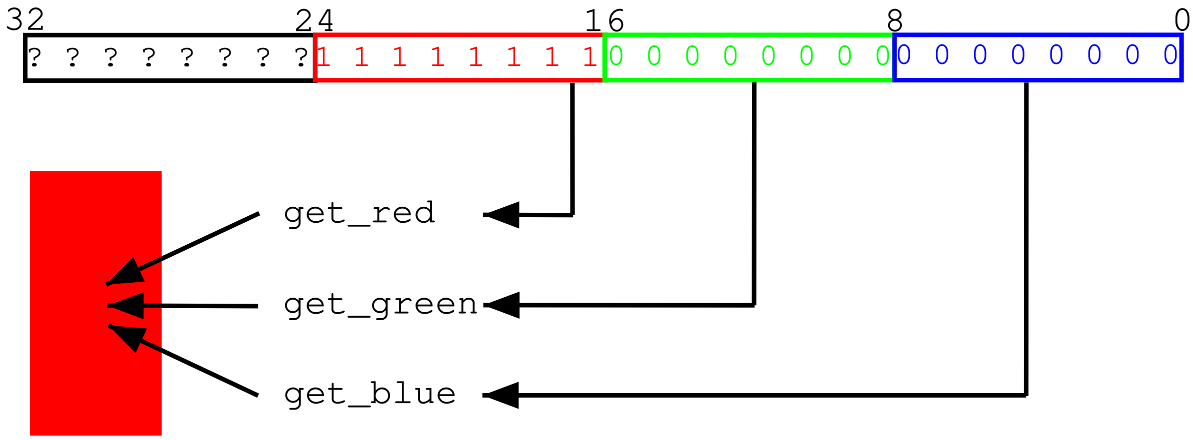



Figure 3.3: Storing RGB components in one integer : memory representation.

```
void set_blue (int *rgb, int blue)
{
   *rgb = (*rgb & 0xFFFF00) | (blue & 0xFF);
}
```

Figure 3.4: Storing RGB components in one integer : code sample.

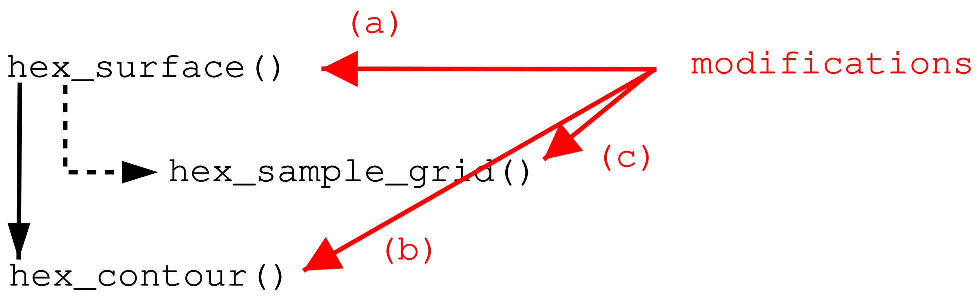
```
int colour;

set_red  (&colour, 255);
set_green(&colour,   0);
set_blue (&colour,   0);

printf("Red:%i - Green: %i - Blue %i\n", get_red  (colour),
                                         get_green(colour),
                                         get_blue (colour));
```

Figure 3.5: Storing RGB components in one integer : usage.

that must be called in that order. The first one sets electrostatic coefficients used by the second one which does all the job. As the evaluation of theses functions takes a long time, I can not evaluated electrostatic forces for each vertex in the grid because it is a waste of time.

In fact I discovered that the most efficient way is to only evaluate useful vertices, i.e those determined by the *hex_contour()* function. Another way to improve the computation time is to evaluate electrostatic potentials and densities together because there are nearly the same. It takes only five seconds to perfom both in a single function call whereas it takes nearly nine to perform one after another one a dual-Pentiurm III at 800Mhz, see table 3.7. This also improve useability on slower computers because the user don't have to wait when switching from density to potential and vice versa. I have done few modifications in the **hex_fnval.c** file to perform that task and to have nice jacket routines that are more easy to use than the basic functions.

First I choosed to display electrostatic forces with threes colours: red for negatives values, white for zero and blue for positives. The display was nice but I thought it will be better to let




**(a)** determine atoms colours

**(b)** build a colour grid

**(c)** find colours of surface's vertices

Figure 3.6: Modifications diagram: atoms colors.

| computations | times (in s) |
|---|---|
| density | 4.40 |
| potential | 4.40 |
| both | 5.04 |

Figure 3.7: Computation times of electrostatic forces.

the user choose the *zero* and values he wants to see himself. Therefore, I coded the motif widget you can see in figure 3.8, setting the minimum and maximum to zero produce the same as my first attempt.

The left scale widget permits to select the minimum value for the *zero*, the right one does the same for the maximum. The colours gradient is done using low level X11 functions and it is redrawn automatically when a scale widget is moved. As theses selected values are used to colour the surface, they must be correct, that's why the widget will only return $minimum < maximum$.

Putting all theses informations together, it is now possible to colour the surface with electrostatics potentials and densities using values computed by *hex_setup_electro()* and *hex_electro_fn()*, and minimum and maximum values for *zero* with this algorithm:

**for all** electrostatic values corresponding to surface's vertices **do**
    $\mathcal{C}_{vertex} := WHITE$
    **if** $value < minimum$ **then**
        $\mathcal{C}_{vertex} := RED$
    **end if**
    **if** $value > maximum$ **then**
        $\mathcal{C}_{vertex} := BLUE$
    **end if**
**end for**

The final part of that mode was to modify the *hex* 3D surface control panel in order to let the

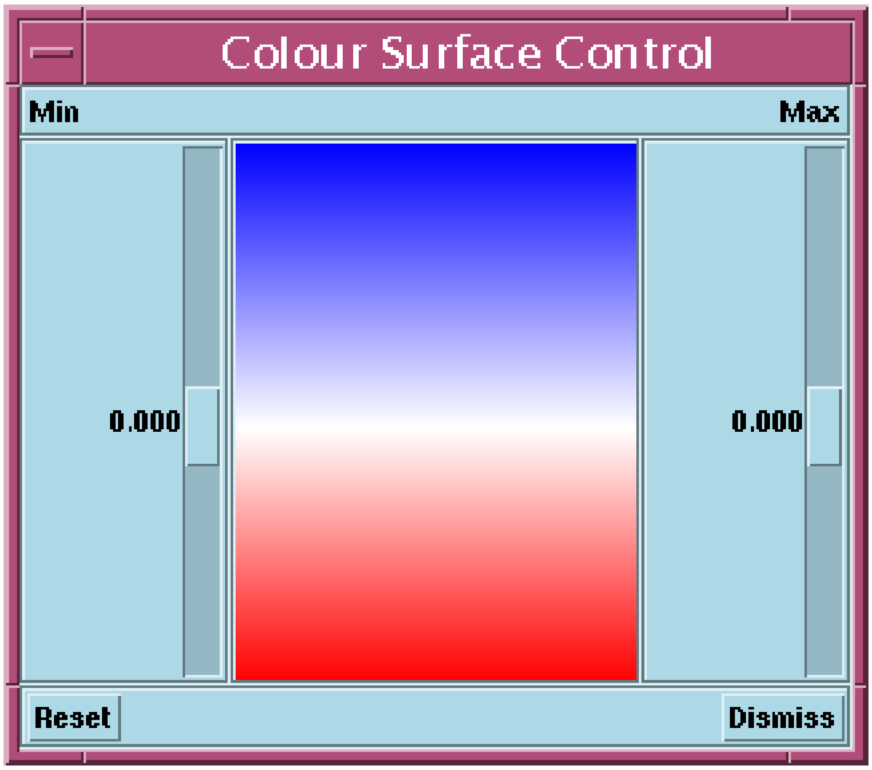



Figure 3.8: Screenshot of the motif gradient widget.

user choose between the four colouring mode: the old mode, the atoms mode, and the electrostatic modes, see 3.12. I had to change differents things in many places to achieve that goal but the code that better shows all modifications is the following:

```
switch (molecule->mesh.colour)
{
   case 0:
      pp = pp_pt(pt_fpt(molecule->msurf.vertices[k1]));
      set_red(&colour, (int) (255 * (0.2 + pp.theta*0.5/PI)));
      set_green(&colour, (int) (255 * 0.4));
      set_blue(&colour, (int) (255 * (0.9 -
      pp.theta*0.5/PI)));
      break;

   case 1:
      colour = molecule->msurf.vcolours[k1];
      break;

   case 2:
      colour = molecule->msurf.potential.vcolours[k1];
      break;

   case 3:
      colour = molecule->msurf.density.vcolours[k1];
```

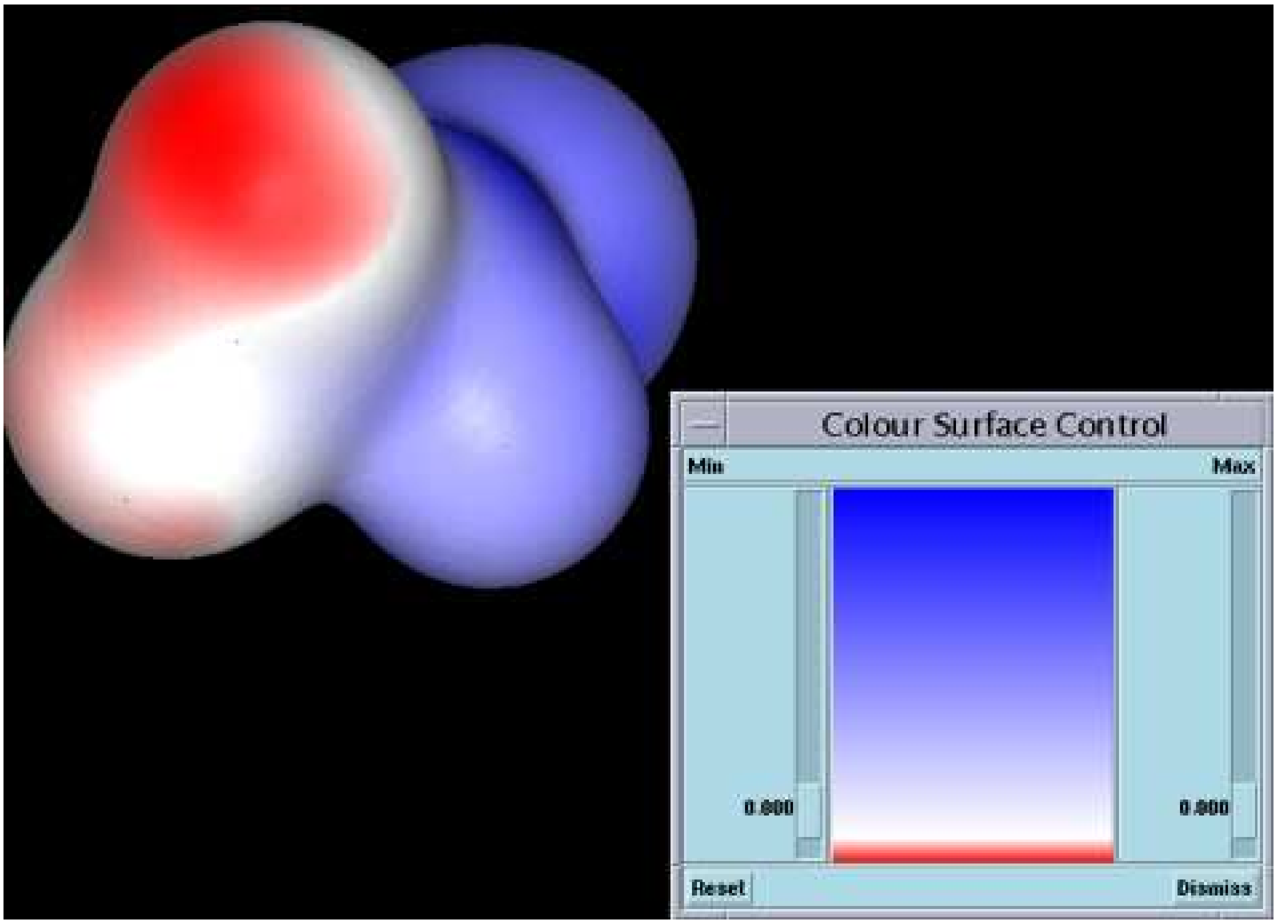



Figure 3.9: Gradient widget controls the surface's colours (a).

```
        break;
    }
```

That is an extract from *hex_surface()* , and used before *hex* assigns a colour to a vertex with openGL calls. That piece of code clearly shows that I had to modify *hex* structures to store the selected colouring mode (`molecule->mesh.colour`), and computed colours (`molecule->msurf.vcolours`,`molecule->msurf.potential.vcolours`, `molecule->msurf.density.vcolours`).

Modifications of the *hex* source only take place in the *hex_draw_surface()* function. If the electrostatic mode is required, the computation is done, and colours are determined using the *zero* of the gradient widget.

## 3.3 Steric mode

In that section, the objective is not to add a new colouring mode but to implement the visualisation of *hex* surfaces using for docking computations.

As for the electrostatic colouring mode, I had to work with the `hex_fnval()` file and be familiar with the new set of steric functions.

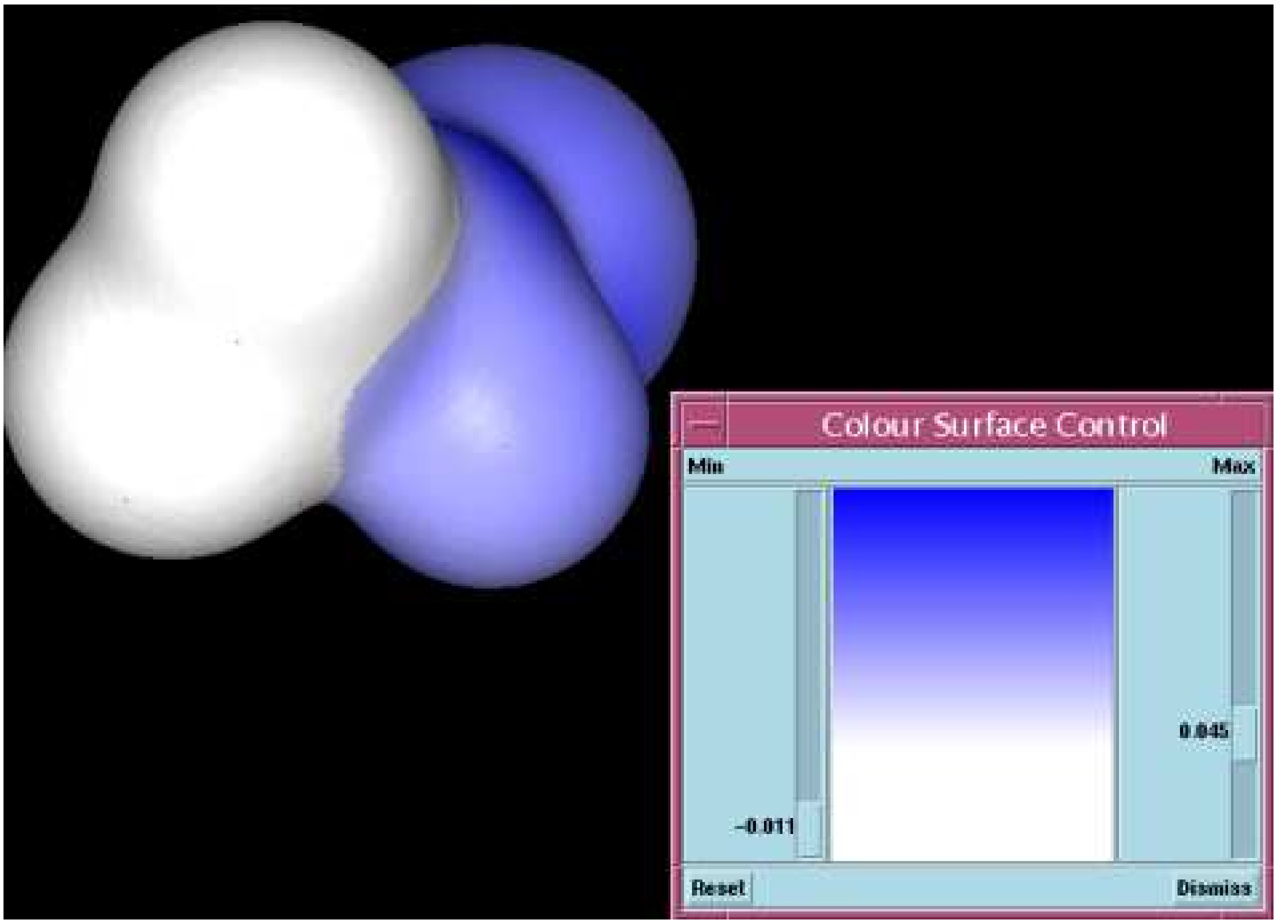



Figure 3.10: Gradient widget controls the surface's colours (b).

Two new surfaces will be produced, one called tau ($\tau$) and the other one called sigma ($\sigma$). The first one will represent a new van der Waals surface, i.e a surface that fits to the hard sphere molecular representation, and is called *interior surface*. The second one will represent a molecule surface extended by a probe radius.

On figure 3.13, $\tau$ is the solid line and $\sigma$ the dashed line. These surfaces are used in docking to determine if two molecules are complementary. By moving molecules positions from (a) to (b), it is possible to find the overlaping of two pairs of surfaces. The more surfaces overlaps, the more molecules are complementary.

Theses surfaces are computed and displayed with a new rendering process using special steric functions. That process is the following: *hex_setup_steric()*, *hex_steric_fn()*, and *hex_contour()*. The first function will build coefficients for the second one using *hex_contour()* , the second one will determine a steric grid and the last one will find vertices of the steric surface.

The main difficulty of the implementation of that new chain of functions calls is that I have done a lot of modifications in *hex_surface()* and *hex_contour()* to implement the atoms colouring

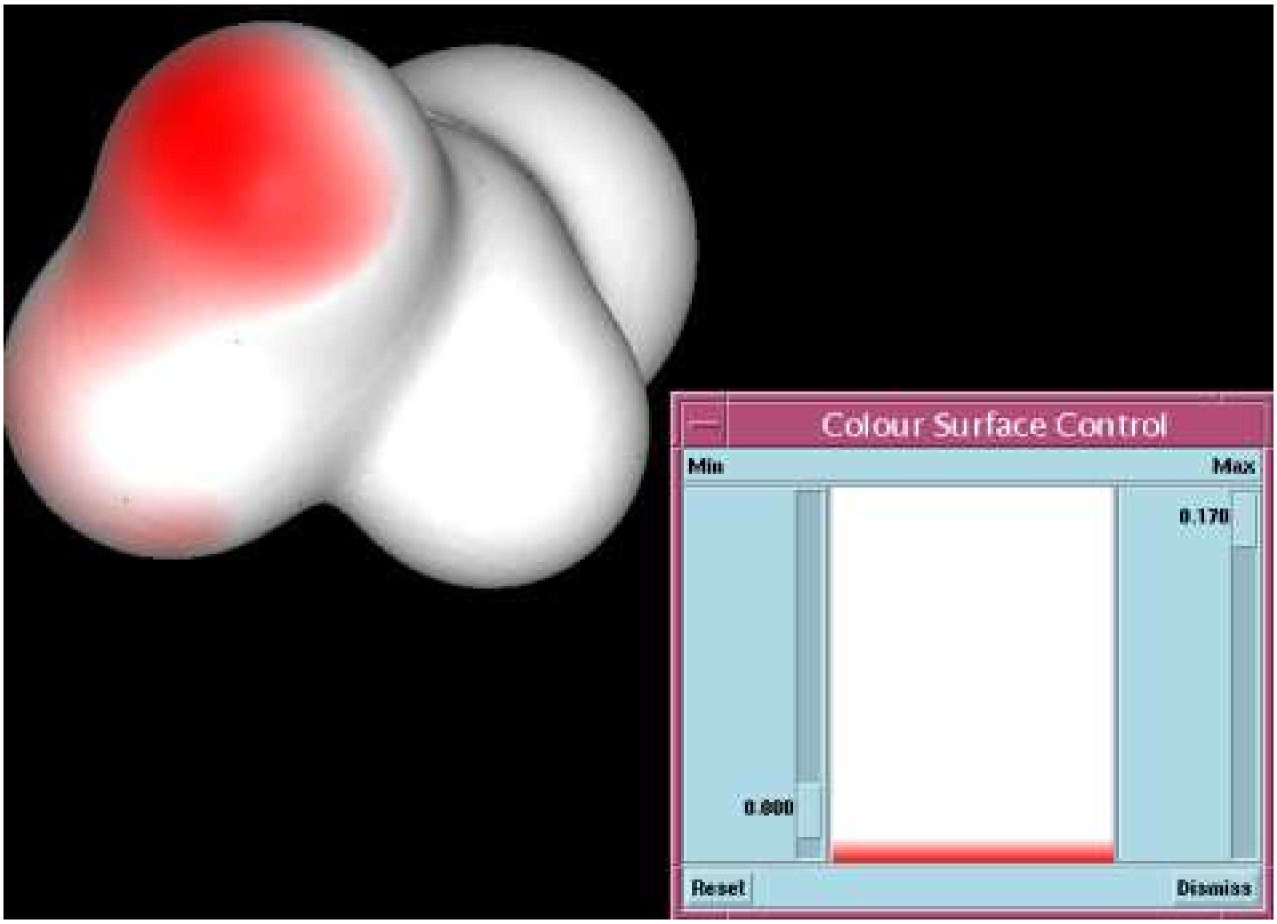



Figure 3.11: Gradient widget controls the surface's colours (c).

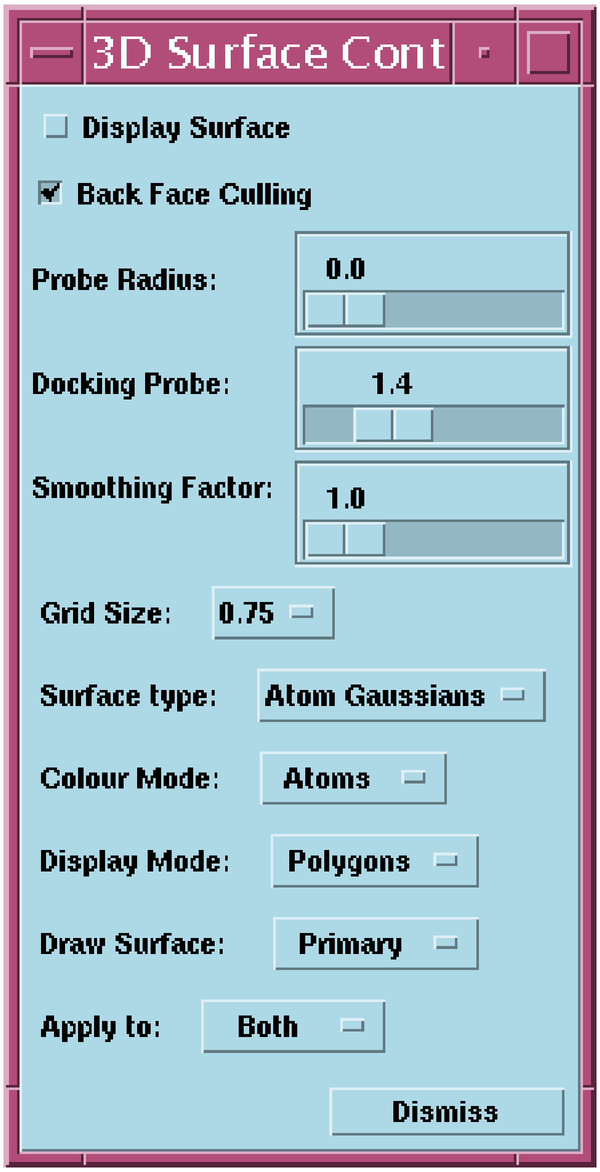
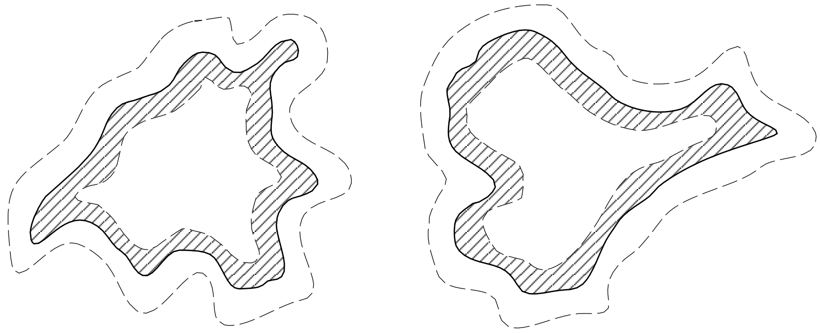
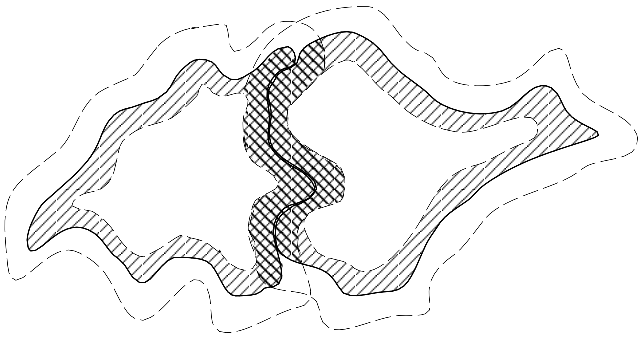



Figure 3.12: New 3D surface control.

mode. Theses versions are not compatible with the steric mode because the informatiom about atoms positions is lost in the steric process. Thus, my first job was to modify the basic rendering process to be able to disable the atoms colouring mode. I choosed to add a flag to functions arguments and build jackets routines to keep the compatibility with current functions, see figure 3.14 for an example with *hex_surface()* . Once it is possible to easily use differents versions of these functions, the implementation of the steric rendering mode is quick because it is only a sequence of fucntions calls.



(a)



(b)

Figure 3.13: Illustration of surfaces overlaping.

```
void hex_surface(Molecule *molecule, int model_no, double probe)
{
   /* Call the hex_surface function in the classic mode */
   hex_surface_main(molecule, model_no, probe, 1);
}

void hex_surface_steric(Molecule *molecule, int model_no, double probe)
{
   /* Call the hex_surface function for the steric computation */
   hex_surface_main(molecule, model_no, probe, 0);
}


void hex_surface_main(Molecule *molecule, int model_no, double probe, int flag)
{
[..]
  if (flag == 1) /* Colour mode is enable */
      cols = (int *)   hex_vm_get(n_atoms*1*sizeof(int));
[..]
   if (flag == 1) /* Colour mode is enable */
      hex_vm_wipe(cols);
[..]
}
```

Figure 3.14: New versions of *hex_surface()* .

## Chapter 4

# Results

The aim of that chapter is to provide an analysis of the new *hex* modes with high quality molecules pictures and comparisons with expected results.

## 4.1 Atoms colours mode

Tables 4.1 and 4.2 provide useful time comparison between the old colouring mode and the new one with atoms colours. For each of theses functions my modifications added a computation time that is constant for small and big molecules. This time is not very different from the one of the classic mode and did not modify the useability of *hex* because the user did not wait longer.

| atom's name | old colour | atoms colour | difference |
|---|---|---|---|
| insuline (346 atoms) | 0.12 | 0.14 | +16% |
| piece of dna (486 atoms) | 0.08 | 0.09 | +13% |
| big molecule (2616 atoms) | 0.88 | 1.02 | +14% |

Figure 4.1: Table of execution times for *hex_surface()* .

| atom's name | old colour | atoms colour | difference |
|---|---|---|---|
| insuline (346 atoms) | 0.38 | 0.46 | +21% |
| piece of dna (486 atoms) | 0.45 | 0.53 | +18% |
| big molecule (2616 atoms) | 3.01 | 3.49 | +16% |

Figure 4.2: Table of execution times for *hex_contour()* .

On pictures 4.3 and 4.4, there are the two first molecules of theses tables.

## 4.2 Electrostatics forces mode

In that section, I will not provide times comparisons because it is not relevant as the the electrostatic forces computation takes a long time.

The picture in figure 4.5 represents the density surface for the insuline molecule, whereas figure 4.6 shows the potential surface. The values range for the first one is from from -0.079990 mV (red) to 0.092619 mV, for the second one it is from -0.052585 e/$\AA^3$ (red) to 0.099926 e/$\AA^3$ (blue).

The last figure (4.7) shows two pictures of the erf1 molecule side by side. In the left, that is the density surface computed by *hex* and one the right that is the same surface but computed by the software *delphi* and published in (Bertram *et al.*, 2000). Comparing these two figures is interesting because it is now easy to see that the computation done by *hex* is quite similar to other softwares. For example on the right side of each surfaces, there is the same regrouping of negative values.

## 4.3 Steric mode

Here, I will provide pictures of molecules with the default colour mode as there is no colour mode for steric surfaces. The objective of this section is too easily visualize similarities between the gaussian surface (figure 4.8) and the new steric surface (figure 4.9).

I provide pictures for $\tau$ and $\sigma$ with differents values for N, a coefficient that determines the accuracy of the steric surface in order to show that even with a *small* coefficient as in figure 4.10, a lot of details are still captured by the steric functions.

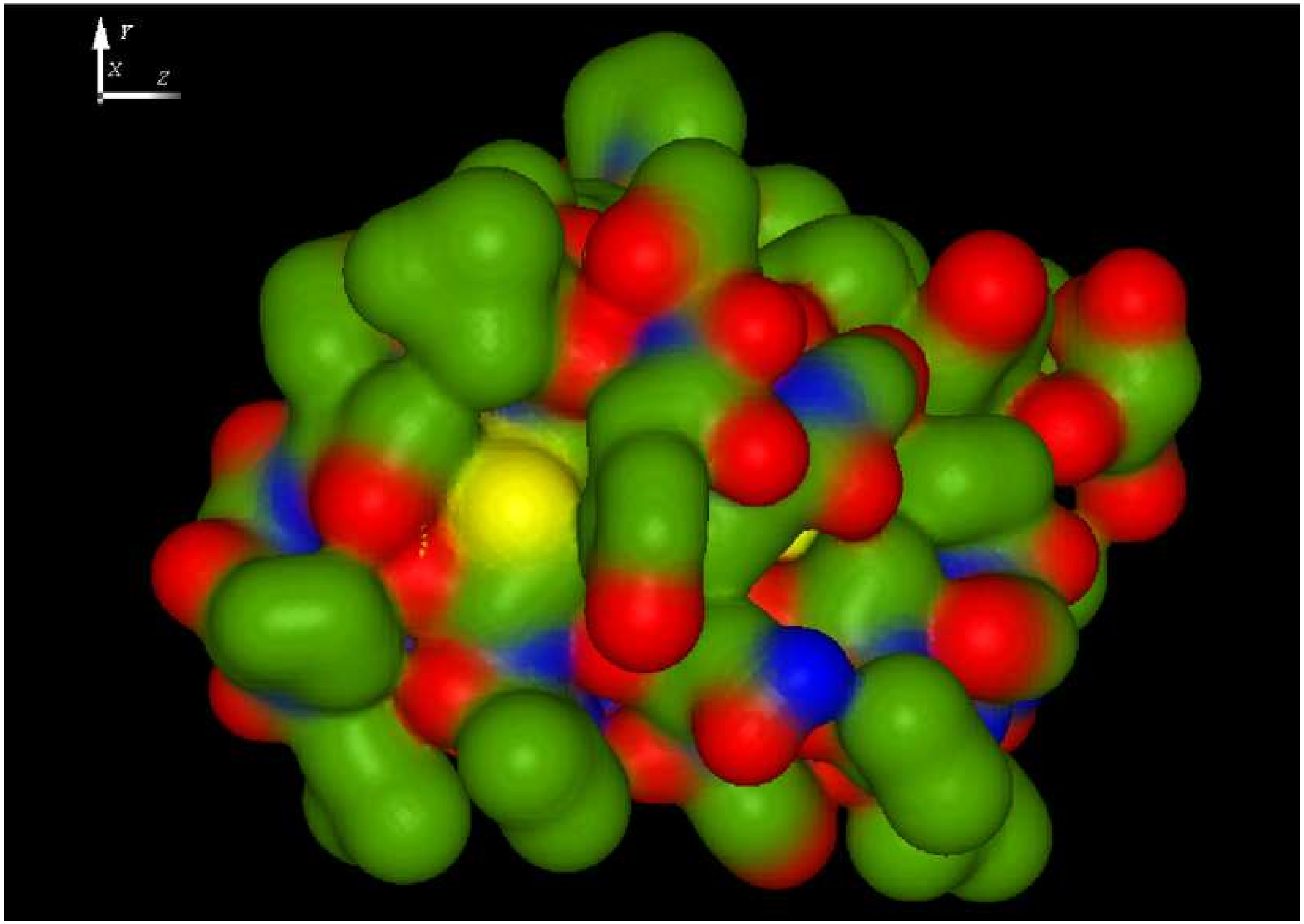


Figure 4.3: Atoms surface of insuline.

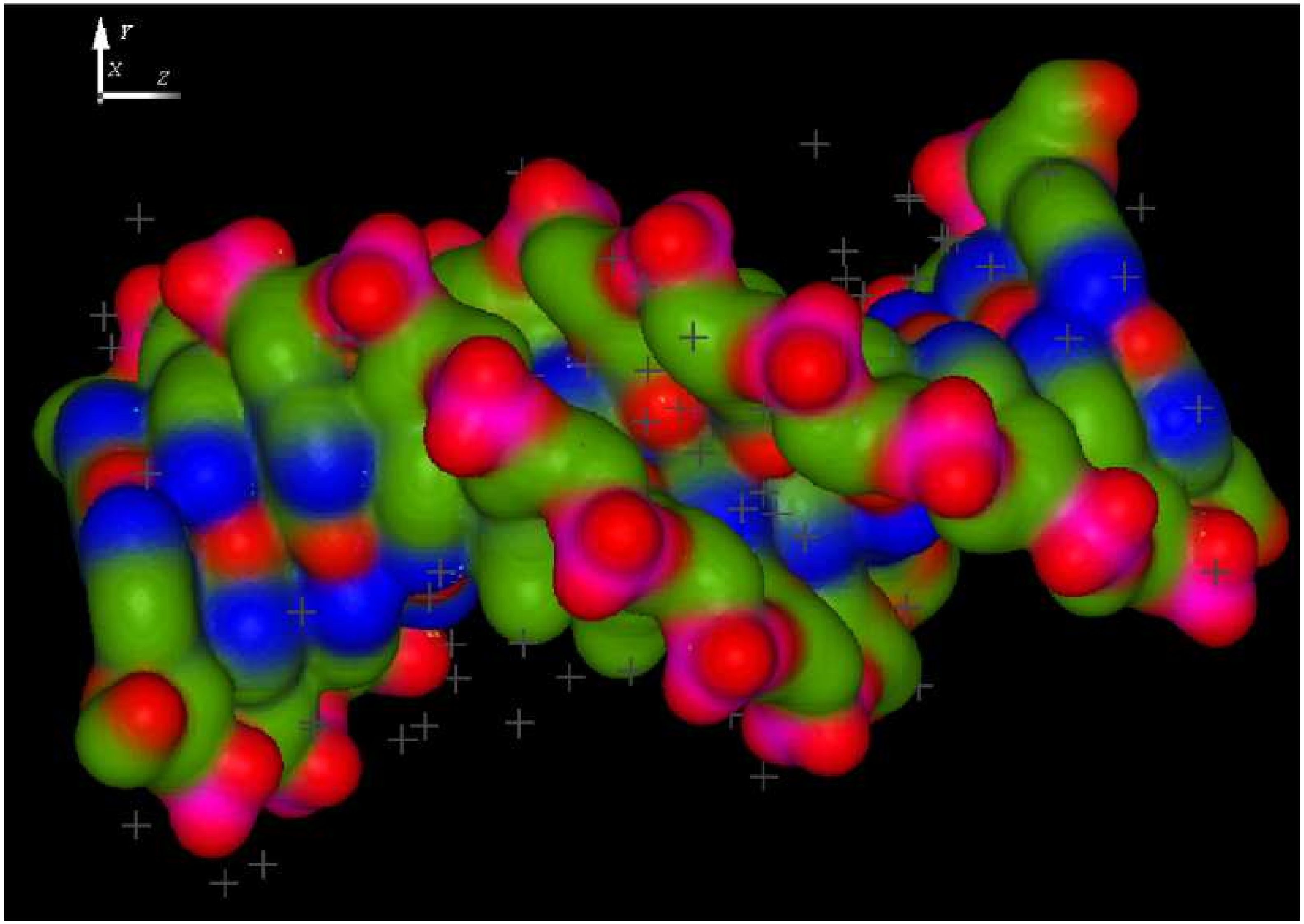



Figure 4.4: Atoms surface of a piece of DNA.

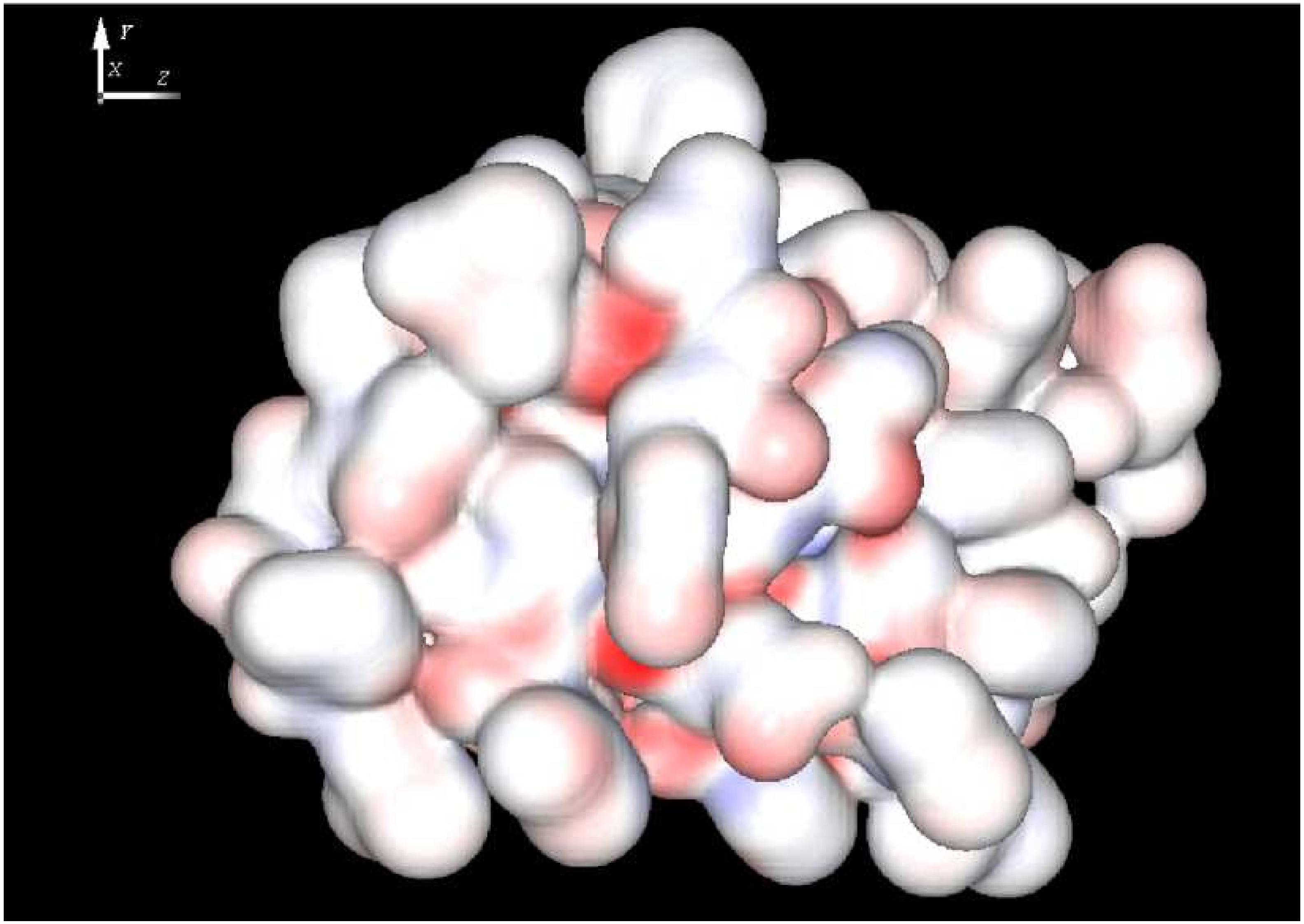



Figure 4.5: Density surface of insuline.

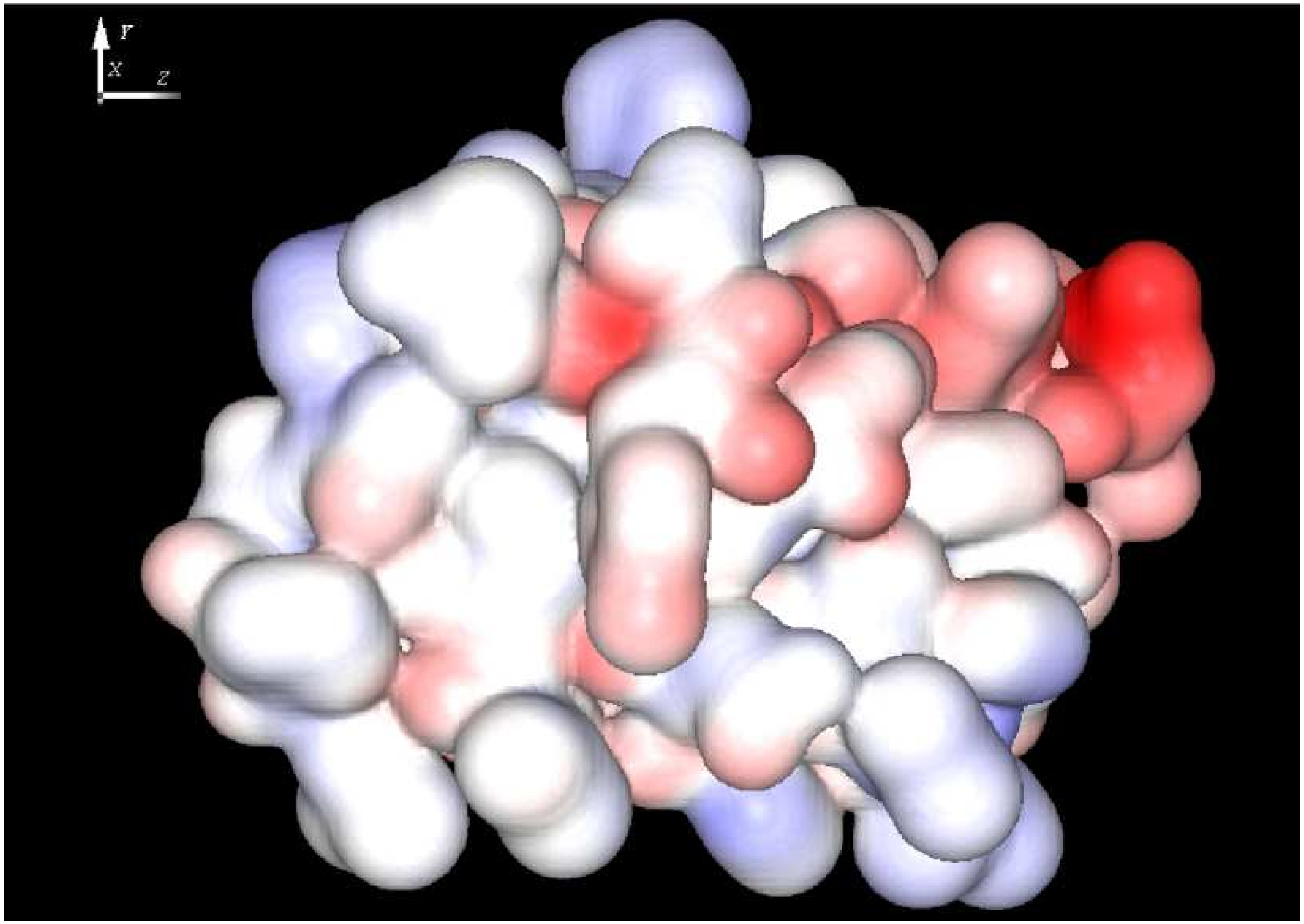



Figure 4.6: Potential surface of insuline.

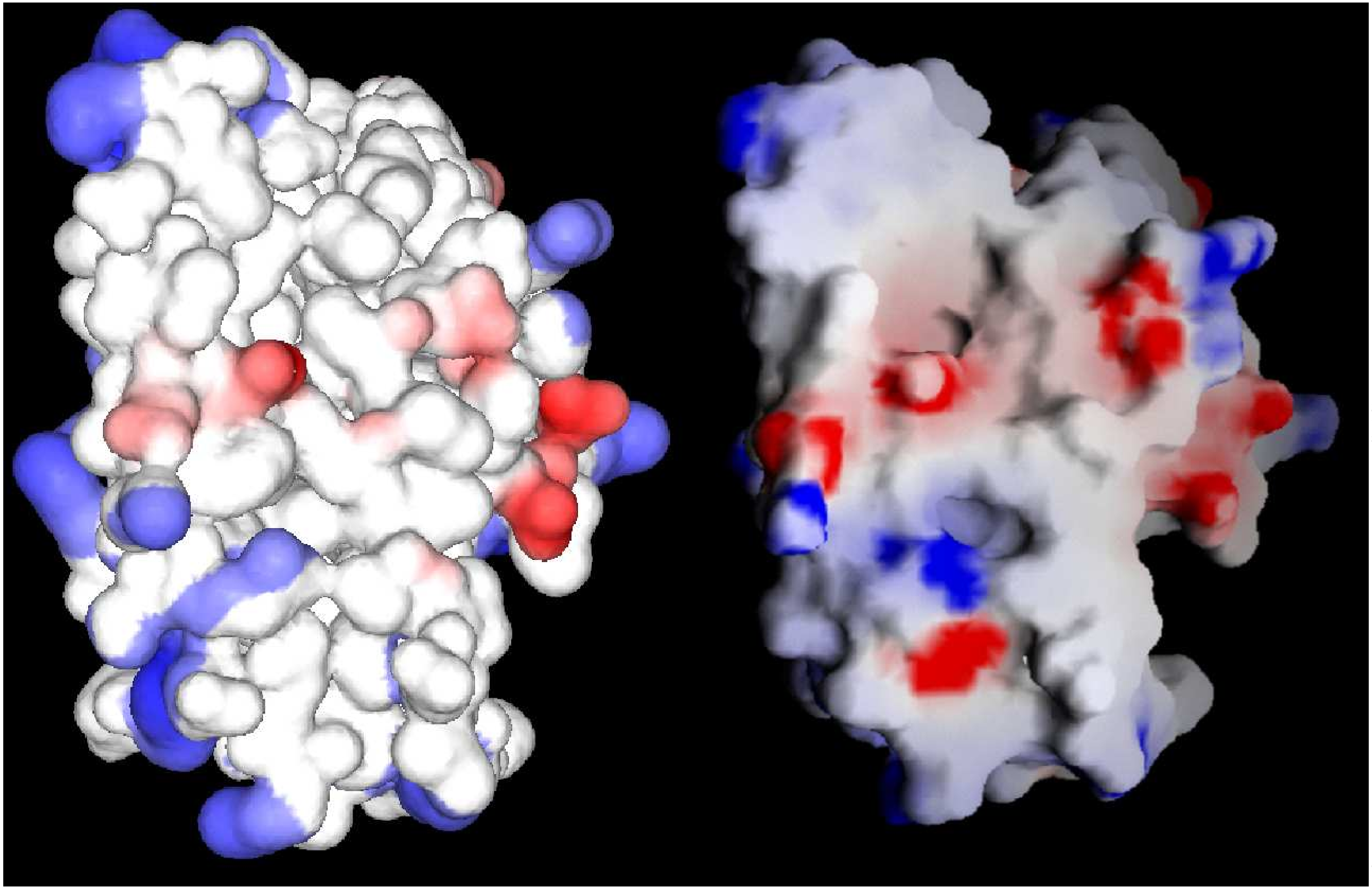


Figure 4.7: Electrostatic surfaces of the erf1 molecule.

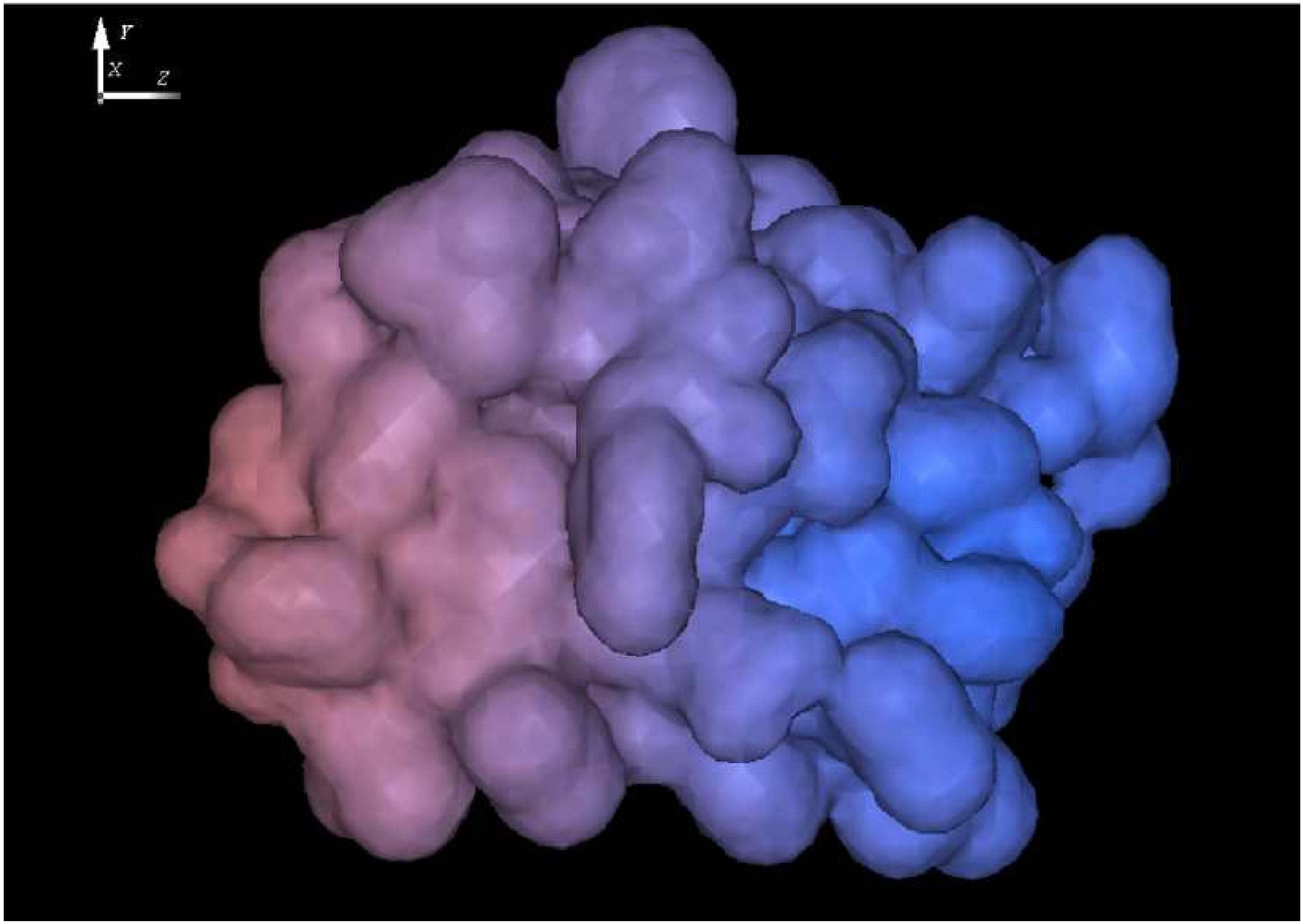



Figure 4.8: Gaussian surface of insuline.

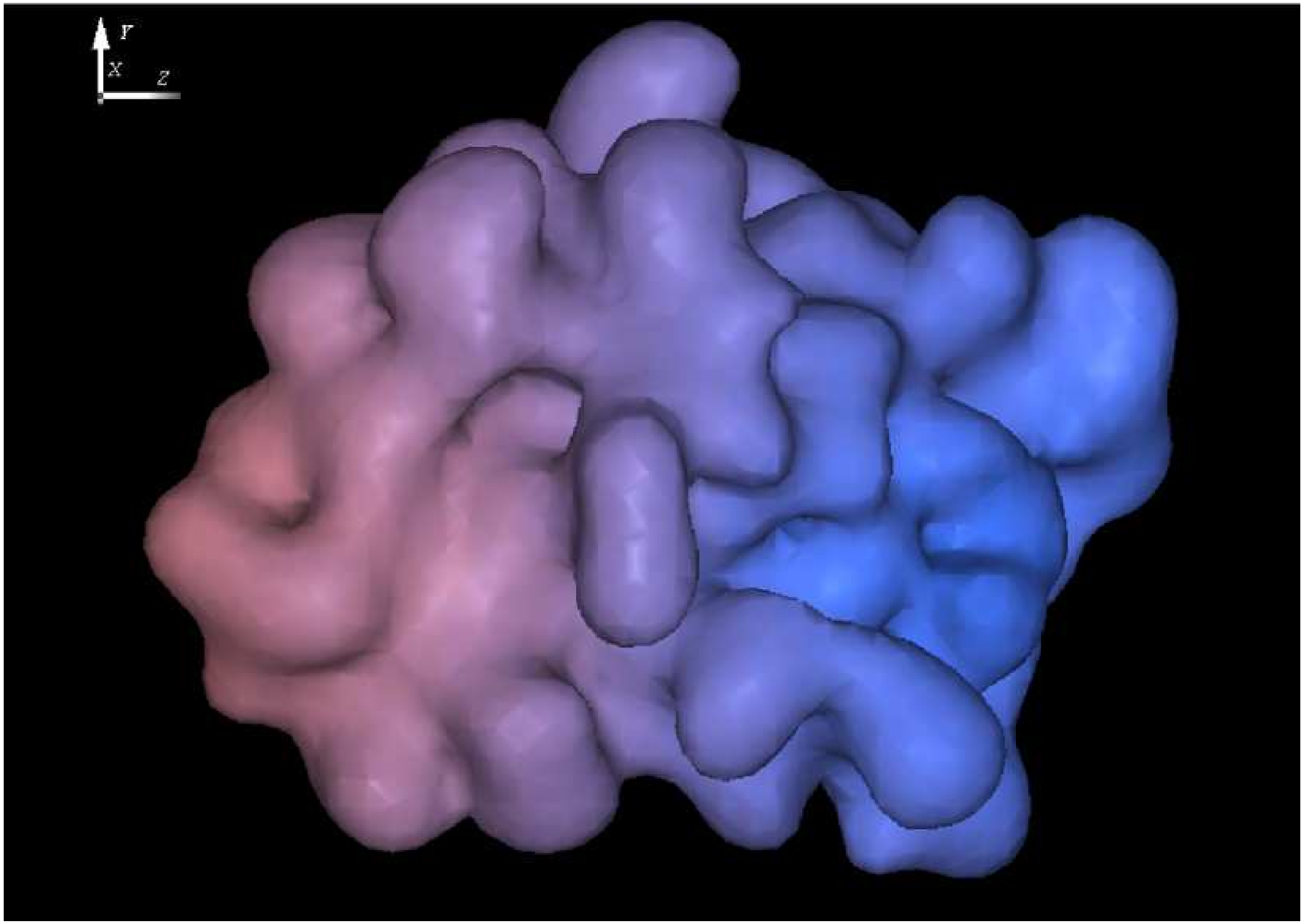



Figure 4.9: $\tau$ surface of insuline with N=25.

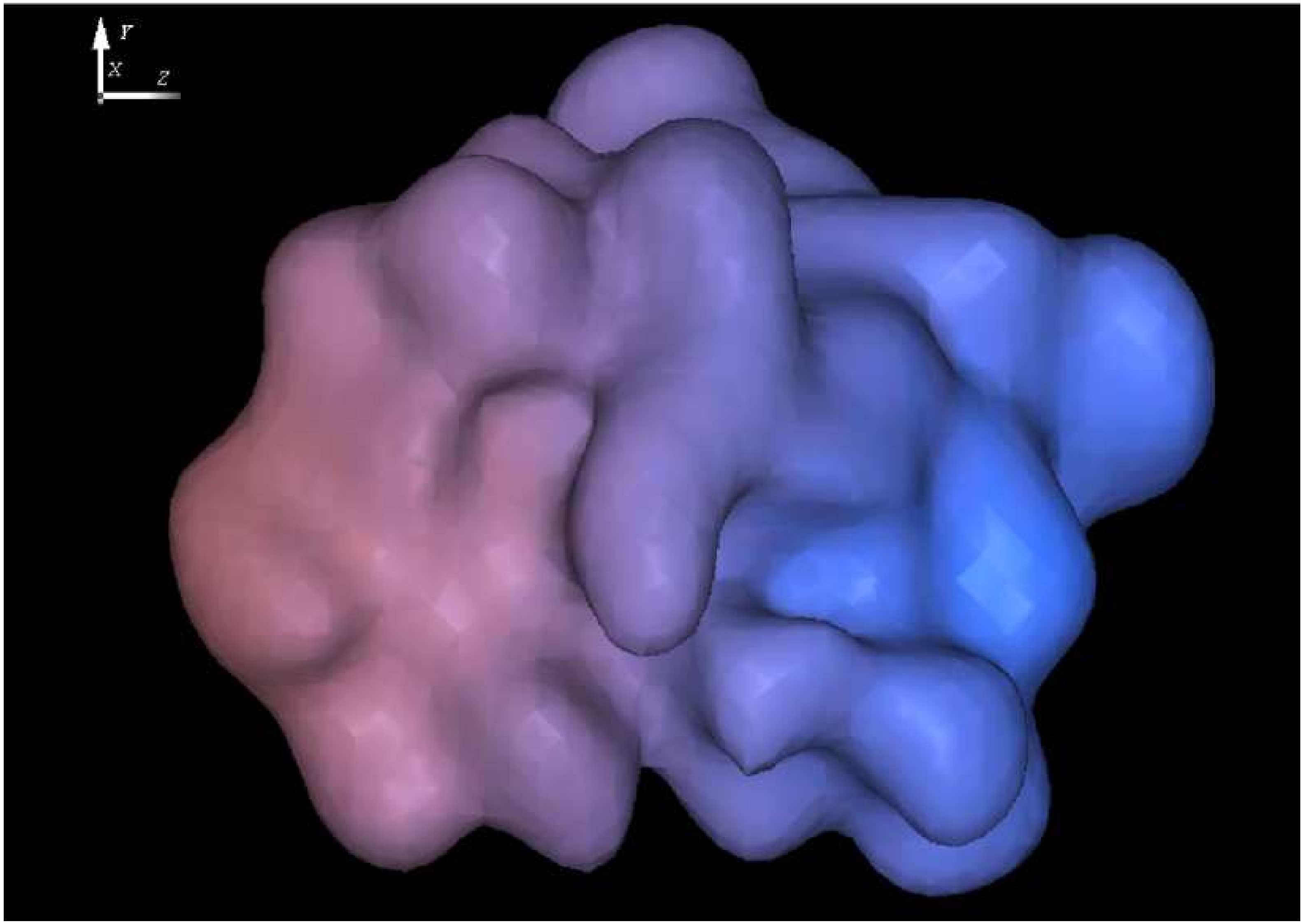



Figure 4.10: $\tau$ surface of insuline with N=16.

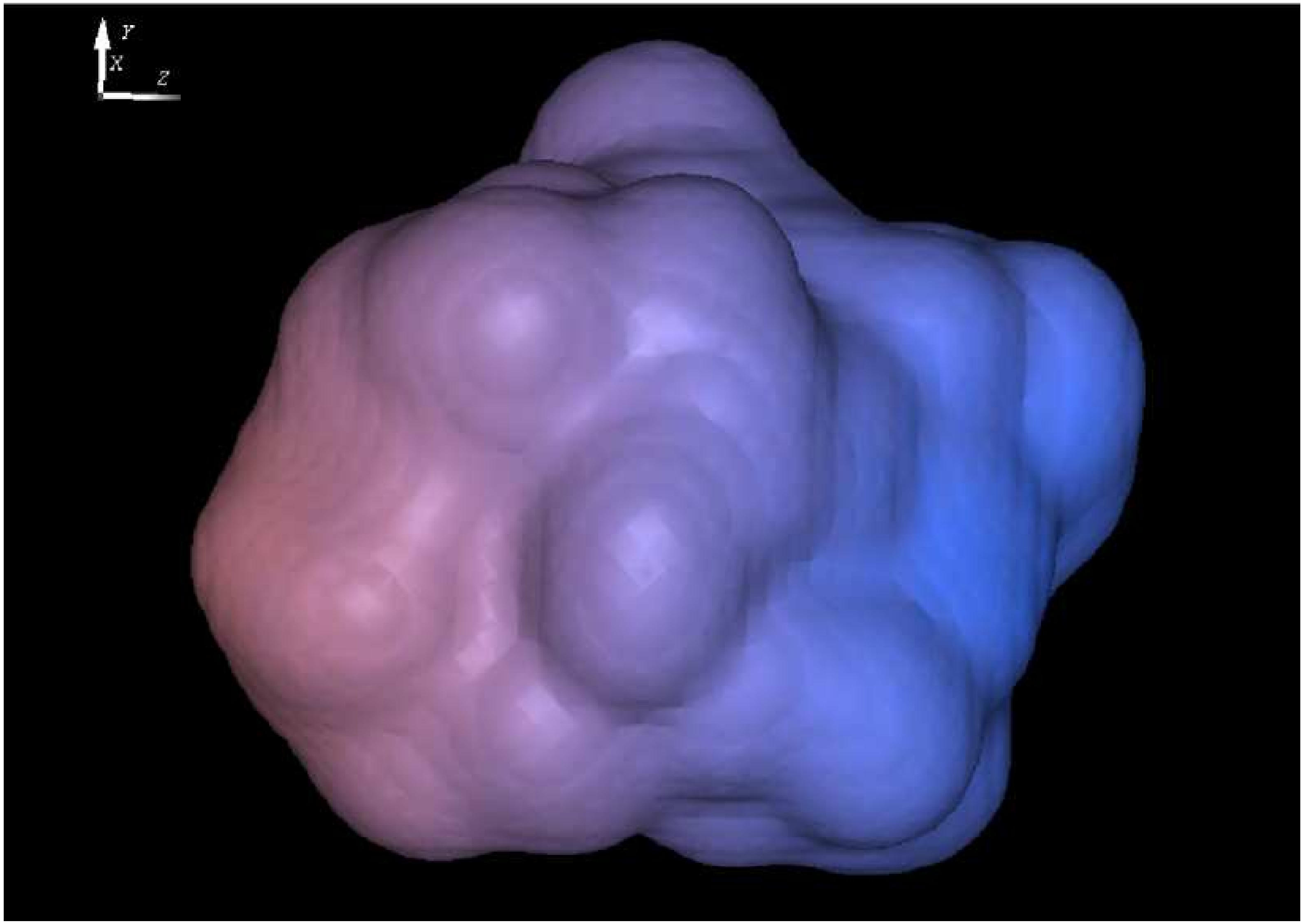



Figure 4.11: Gaussian surface of insuline with extended probe radius.

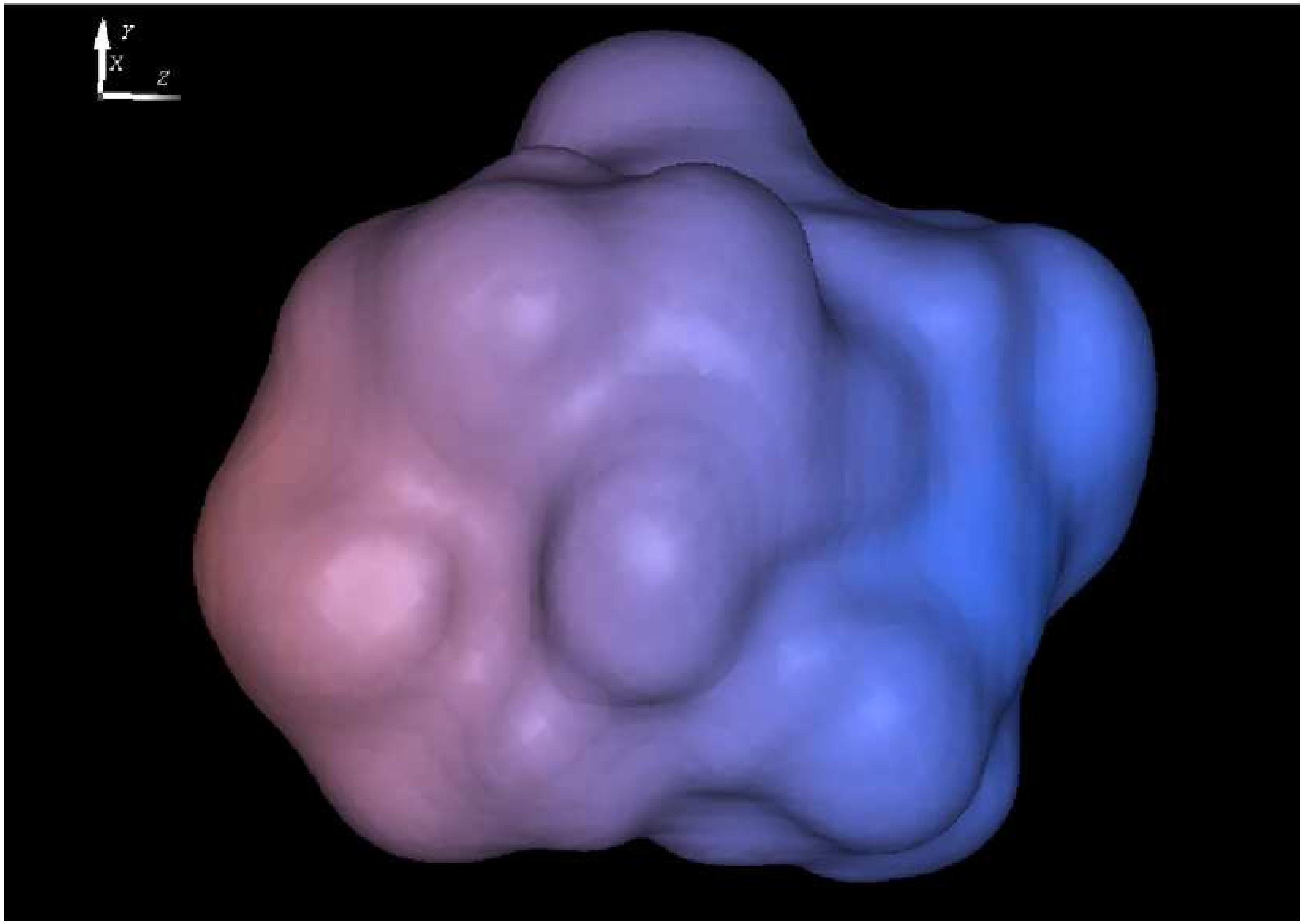



Figure 4.12: $\sigma$ surface of insuline with extended probe radius with N=25.

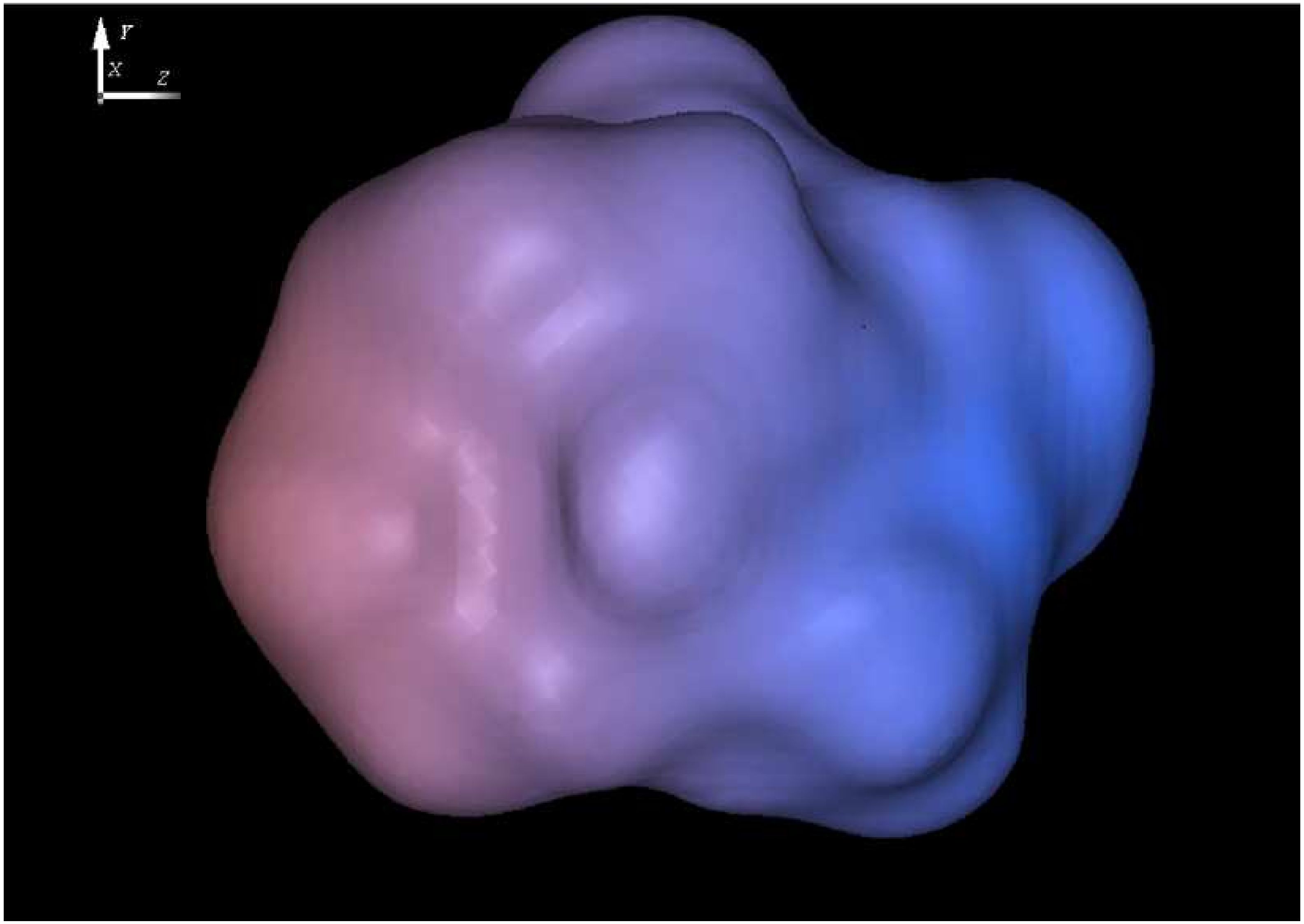



Figure 4.13: $\sigma$ surface of insuline with extender probe radius with N=16.

# Chapter 5

# Conclusion

## 5.1 Personal achievement

Working on that project learnt me many things. First of all it was my first experience in bioinformatics and in an area not related with systems and networks. It was a good experience to apply my computing knowledge to a new subject.

Modifying *hex* obliged me to learn some new things in the programming area. I had to learn the motif library in order to modify *hex* control panels and to add the gradient widget for the electrostatics mode. I also had to deal with a lot of memory and the *gdb* debugger was very useful to correct bugs in my modifications, and I am now more confident with its usage.

When writing this report, I had to produce nice figures to illustrate the text and I discover nice tools as *xfig* which help me to quickly produce nice pictures as the grid page 17. I knew LaTeXbefore but it was not enough to produce this report and the code listing because I did not know how to put nice figures and how to change the orientation of the page.

Finally, I think that this project will be very useful because I would like to follow my studies in the bioinformatics area. Working on *hex* will help me to go on that way because the study of three-dimensional structure of biomolecules will certainly be an important part of modern research. It also showed me that I add to learn more in biophysics if I later want to work in a team compound of people from the biology area.

## 5.2 What can be still be done

The modifications I made to *hex* are mostly good but I think it will be better to try to optimize computation time of the atoms colours mode. Avoiding useless computations, especially with the steric mode, will certainly be the most important thing to do because the code did not include a good way to determine if a computation as previously been done.

Some things in *hex* can be done to improve the re-useability of functions. Actually, it mostly use too much generic data structures and sometimes prevents a programmer to reuse functions. For example, a function that deals with computation on a surface will take a molecule structure as parameter instead of a surface. This sometimes leads to very difficult modifications of the existing code in order to reuse it.

The cube decomposition algorithm produces small triangles on the surface, reducing theses triangles will decrease computation times for the electrostatic colours mode and the displaying speed.

I think that future developpement for *hex* should be on the optimisation of actual algorithm and of the memory usage. For example, it is may be possible to build the density grid and to do the contouring in the same time.

## Appendix A

# Bibliography

G. Bertram, H. Bell, D. W. Ritchie, G. Fullerton, and I. Stansfield. Terminating eukaryotic translation: domain 1 of release factor eRF1 functions in stop codon recognition. *RNA*, 6(9):1236–1247, 2000.

J. A. Grant, M. A. Gallardo, and B. T. Pickup. A fast method of molecular shape comparison: A simple application of a Gaussian description of molecular shape. *J. Comp. Chem.*, 17(14):1653–1666, 1996.

A. Guéziec and R. Hummel. Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Trans. Vis. Comp. Graph.*, 1(4):328–342, 1995.