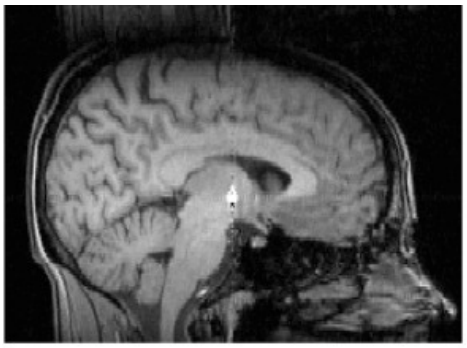
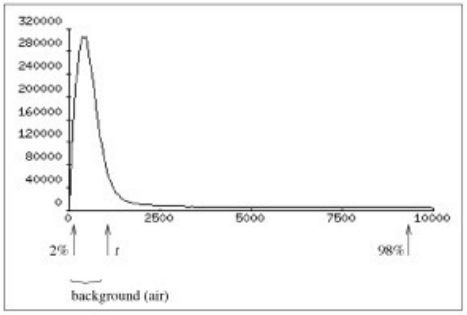
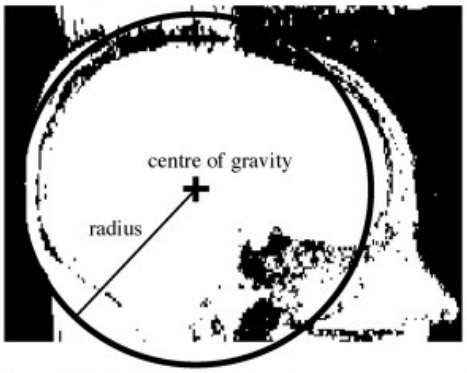
Analysis of the “Fast Robust Automated Brain Extraction” Algorithm

▼ Overview of the brain extraction method

The intensity histogram is processed to find “robust” lower and upper intensity values for the image, and a rough brain/non-brain threshold. The centre-of-gravity of the head image is found, along with the rough size of the head in the image. A triangular tessellation of a sphere's surface is initialised inside the brain and allowed to slowly deform, one vertex at a time, following forces that keep the surface well-spaced and smooth, while attempting to move toward the brain's edge. If a suitably clean solution is not arrived at then this process is re-run with a higher smoothness constraint. Finally, if required, the outer surface of the skull is estimated. A graphical overview is shown below.







input image

histogram-based threshold estimation

binarisation using t
to find centre of gravity and
radius of equivalent volume sphere

initialisation of tesselated surface

iterate updates to surface

final tesselated surface

self-intersecting?

yes

no

stop

In the "**Algorithm Breakdown**" section, we'll analyse each step from the BET processing flowchart above. To avoid repetition, we'll use the following aliases instead of the full names:

- **step 0** - input image
- **step 1** - histogram-based threshold estimation
- **step 2** - binarisation using t to find centre of gravity and radius of equivalent volume sphere
- **step 3** - initialisation of tesselated surface
- **step 4** - iterate updates to surface
- **step 5** - final tesselated surface
- **step 6** - check for self-intersecting

▼ Algorithm breakdown

▼ Imports and using directives

▼ Code with description of custom importation

```
#include <iostream>
#include <string>
#include <fstream>
#include <stdio.h>
#include <cmath>
#include <algorithm>

#include "utils/options.h" // a custom header file for h
#include "newimage/newimageall.h" // a header file from
#include "meshclass/meshclass.h" // a custom header file


using namespace std;
using namespace NEWIMAGE; // contains classes and functi
using namespace Utilities; // contains utility functions
using namespace mesh; // contains classes and functions
```

> lines 72-87

▼ Establishing a structure for managing command-line options

General description

bet and bet2

By default bet just calls the bet2 program for simple brain extraction. However, it also allows various additional behaviour. The basic usage is:

bet <input> <output> [options]

Main bet2 options:

- **o** generate brain surface outline overlaid onto original image
- **m** generate binary brain mask
- **s** generate rough skull image (not as clean as what betsurf generates)
- **n** don't generate the default brain image output
- **f** <f> fractional intensity threshold (0→1); default=0.5; smaller values give larger brain outline estimates
- **g** <g> vertical gradient in fractional intensity threshold (-1→1); default=0; positive values give larger brain outline at bottom, smaller at top
- **r** <r> head radius (mm not voxels); initial surface sphere is set to half of this
- **c** < x y z> centre-of-gravity (voxels not mm) of initial mesh surface.
- **t** apply thresholding to segmented brain image and mask
- **e** generates brain surface as mesh in .vtk format.

🌐 Source: https://fsl.fmrib.ox.ac.uk/fsl/fslwiki/BET/UserGuide#betgui

▼ Code Correspondence

```
string title="BET (Brain Extraction Tool) v2.1 - FMRIB A
string examples="bet2 <input_fileroot> <output_fileroot>

Option<bool> verbose(string("-v,--verbose"), false,
             string("switch on diagnostic messages"),
             false, no_argument);
Option<bool> generate_mesh(string("-e,--mesh"), false,
             string("generates brain surface as mesh in
             false, no_argument);
Option<bool> help(string("-h,--help"), false,
             string("displays this help, then exits"),
             false, no_argument);
Option<bool> outline(string("-o,--outline"), false,
             string("generate brain surface outline over
             false, no_argument);
Option<bool> skull(string("-s,--skull"), false,
           string("generate approximate skull image"),
           false, no_argument);
Option<bool> mask(string("-m,--mask"), false,
          string("generate binary brain mask"),
          false, no_argument);
Option<bool> no_output(string("-n,--nooutput"), false,
               string("don't generate segmented brain im
               false, no_argument);
Option<bool> apply_thresholding(string("-t,--threshold")
                string("-apply thresholding to segmented
                false, no_argument);
Option<float> fractional_threshold(string("-f"), 0.5,
                    string("~<f>\t\tfractional intensity

Option<float> gradient_threshold(string("-g"), 0.0,
                 string("~<g>\t\tvertical gradient in fr
```

```
Option<float> smootharg(string("-w,--smooth"), 1.0,
                    string("~<r>\tsmoothness factor; de

Option<float> radiusarg(string("-r,--radius"), 0.0,
             string("~<r>\thead radius (mm not voxels);

Option<float> centerarg(string("-c"), 0.0,
                 string("~<x y z>\tcentre-of-gravity (vo
```

> lines 106-148

▼ Estimation of basic image and brain parameters, a.k. the essence of step 1, step 2 and step 3

The first processing that is carried out is the estimation of a few simple image parameters, to be used at various stages in subsequent analysis.

1. **First, the robust image intensity minimum and maximum are found.** Robust means the effective intensity extrema, calculated ignoring small numbers of voxels that have widely different values from the rest of the image. These are calculated by looking at the intensity histogram, and ignoring long low tails at each end. Thus, the intensity “minimum”, referred to as ***t*2** is the intensity below which lies 2% of the cumulative histogram. Similarly, ***t*98** is found. It is often important for the latter threshold to be calculated robustly, as it is quite common for brain images to contain a few high intensity “outlier” voxels; for example, the DC spike from image reconstruction, or arteries, which often appear much brighter than the rest of the image.
    a. **Finally, a roughly chosen threshold *t* is calculated that attempts to distinguish between brain matter and background** (because bone appears dark in most MR images, “background” is taken to include bone). This *t* is simply set to lie 10% of the way between *t*2 and *t*98.
2. **The brain/background threshold *t* is used to roughly estimate the position of the centre of gravity (COG) of the brain/head in the image.**

For all voxels with intensity greater than *t*, their intensity ("mass") is used in a standard weighted sum of positions. Intensity values are upper limited at *t*98, so that extremely bright voxels do not skew the position of the COG.

3. **Next, the mean "radius" of the brain/head in the image is estimated.** There is no distinction made here between estimating the radius of the brain and the head, this estimate is very rough, and simply used to get an idea of the size of the brain in the image; it is used for initializing the brain surface model. All voxels with intensity greater than *t* are counted, and a radius is found, taking into account voxel volume, assuming a spherical brain.
4. **Finally, the median intensity of all points within a sphere of the estimated radius and centered on the estimated COG is found (*t*m).**

P.S. The simple image parameters listed above, particularly t2, t, cog, and radius, are crucial for calculating the update movement vector u. This vector will dictate the growth direction of the tessellated surface, which in turn defines the targeted brain mass.

▼ Code Correspondence (Including a Step-by-Step Description)

▼ 1. + 1.a Evaluation of intensity minimum/**t2** and maximum/**t98** and of a roughly chosen threshold **t**

▼ 1.1. Declaration of **t2, t98,** and **t** along with **cog**, **radius** and **tm**, which will be referenced later in sections 2, 3 and 4 respectively.

```
struct bet_parameters
{
  double min, max, t98, t2, t, tm, radius;
  Pt cog;
};
```

▼ 1.2. Initialization of **t2**, **t98** with *robustmin()*, *robustmax()* return values and calculation of **t**

```
bet_parameters adjust_initial_mesh(const volume<fl
{
  bet_parameters bp;
  double xdim = image.xdim();
  double ydim = image.ydim();
  double zdim = image.zdim();
  double t2, t98, t;

  //computing t2 && t98
  //  cout<<"computing robust min && max begins"<<

  bp.min = image.min();
  bp.max = image.max();

  t2 = image.robustmin();
  t98 = image.robustmax();
  //t2=32.;
  //t98=16121.;

  //  cout<<"computing robust min && max ends"<<en

  t = t2 + .1*(t98 - t2);
  bp.t98 = t98;
  bp.t2 = t2;
  bp.t = t;
  //  cout<<"t2 "<<t2<<" t98 "<<t98<<" t "<<t<<end

  //... sections 2, 3 and 4
```

▼ 1.3. Definition of *robustmin()*, *robustmax()*

```
template <class T, class S, class R>
  void find_thresholds(const S& vol, T& minval, T&
  {
    // STEVE SMITH'S CODE (in ancient times) - ada
```

```
int HISTOGRAM_BINS=1000;
ColumnVector hist(HISTOGRAM_BINS);
int MAX_PASSES=10;
int top_bin=0, bottom_bin=0, count, pass=1,
  lowest_bin=0, highest_bin=HISTOGRAM_BINS-1;
int64_t validsize;
//vector<int64_t> coordinates;
//vector<T> extrema(calculateExtrema(vol,coordin

 T thresh98=0, thresh2=0, min, max;
if (use_mask) { min=vol.min(mask), max=vol.max(m
else { min=vol.min();  max=vol.max(); }

if (hist.Nrows()!=HISTOGRAM_BINS) { hist.ReSize(

while ( (pass==1) ||
    ( (double) (thresh98 - thresh2) < (((double)
  // find histogram and thresholds
  {
    if (pass>1) // redo histogram with new min a
  {
    // increase range slightly from the 2-98% ra
    bottom_bin=Max(bottom_bin-1,0);
    top_bin=Min(top_bin+1,HISTOGRAM_BINS-1);

    // now set new min and max on the basis of t
    T tmpmin = (T)( min + ((double)bottom_bin/(d
    max = (T)( min + ((double)(top_bin+1)/(doubl
    min=tmpmin;
  }

    if (pass==MAX_PASSES || min==max)  // give u
  {
    if (use_mask) { min=vol.min(mask); max=vol.m
    else { min=vol.min();  max=vol.max(); }
  }
```

```
  if (use_mask) validsize = imageHistogram(vol
  else validsize = imageHistogram(vol,HISTOGRA

  if (validsize<1)
{
      minval=thresh2=min;
  maxval=thresh98=max;
  return;
}

  if (pass==MAX_PASSES)  /* ... _but_ ignore e
{
  validsize-= MISCMATHS::round(hist(lowest_bin
    MISCMATHS::round(hist(highest_bin+1));
  lowest_bin++;
  highest_bin--;
}

  if (validsize<0) /* ie zero range */
{
  thresh2=thresh98=min;
  break;
}

  double fA = (max-min)/(double)(HISTOGRAM_BIN

  for(count=0, bottom_bin=lowest_bin; count<va
count+=MISCMATHS::round(hist(bottom_bin + 1));
  bottom_bin--;
  thresh2 =  min + (T)((double)bottom_bin*fA);

  for(count=0, top_bin=highest_bin; count<vali
count+=MISCMATHS::round(hist(top_bin + 1));
  top_bin++;
  thresh98 = min + (T)((double)(top_bin+1)*fA)
```

```
      if (pass==MAX_PASSES) break;
      pass++;
    }
    minval=thresh2; //corresponds to robustmin ret
    maxval=thresh98; //corresponds to robustmax re
```

▼ 2. Estimation of the center of gravity (cog) of the brain/head in the image

```
//... sections 1 and 1.a

//  cout<<"computing center && radius begins"<<endl;

  //finds the COG
  Pt center(0, 0, 0);
  double counter = 0;
  if (xpara == 0. & ypara==0. & zpara==0.)
    {
      double tmp = t - t2;
      for (int k=0; k<image.zsize(); k++)
    for (int j=0; j<image.ysize(); j++)
      for (int i=0; i<image.xsize(); i++)
        {
          double c = image(i, j, k ) - t2;
          if (c > tmp)
        {
          c = min(c, t98 - t2);
          counter+=c;
          center +=  Pt(c*i*xdim, c*j*ydim, c*k*zdim)
        }
        }
      center=Pt(center.X/counter, center.Y/counter, c
      //cout<<counter<<endl;
```

```
      //  cout<<"cog "<<center.X<<" "<<center.Y<<" "<
    }
  else center=Pt(xpara*xdim, ypara*ydim, zpara*zdim);

  bp.cog = center;

  m.translation(center.X, center.Y, center.Z);
```

▼ 3. Estimation of the mean radius of the brain/head in the image

```
if (rad == 0.)
    {
      double radius=0;
      counter=0;
      double scale=xdim*ydim*zdim;
      for (int k=0; k<image.zsize(); k++)
    for (int j=0; j<image.ysize(); j++)
      for (int i=0; i<image.xsize(); i++)
        {
          double c = image(i, j, k);
          if (c > t)
        {
          counter+=1;
        }
        }
      radius = pow (.75 * counter*scale/M_PI, 1.0/3.0
      //      cout<<radius<<endl;
      bp.radius = radius;
    }
  else (bp.radius = rad);

  m.rescale (bp.radius/2, center);
```

▼ 4. Calculation of the median intensity **tm** of all points within a sphere of the estimated radius (indicates a rough brain/non-brain threshold)

```
//computing tm
  //  cout<<"computing tm begins"<<endl;
  vector<double> vm;
  for (int k=0; k<image.zsize(); k++)
    for (int j=0; j<image.ysize(); j++)
      for (int i=0; i<image.xsize(); i++)
    {
      double d = image.value(i, j, k);
      Pt p(i*xdim, j*ydim, k*zdim);
      if (d > t2 && d < t98 && ((p - center)|(p - cen
        vm.push_back(d);
    }

  int med = (int) floor(vm.size()/2.);
  //  cout<<"before sort"<<endl;
  nth_element(vm.begin(), vm.begin() + med - 1, vm.en
  //partial_sort(vm.begin(), vm.begin() + med + 1, vm
  //double tm = vm[med];
  double tm=(*max_element(vm.begin(), vm.begin() + me
  //  cout<<"tm "<<tm<<endl;
  bp.tm = tm;
  //  cout<<"computing tm ends"<<endl;

  return bp;

} //end of adjust_initial_mesh func.
```

> lines 94-98 and line 302-407 → *adjust_initial_mesh* function from *bet2.cpp*

▼ Surface model and initialisation, a.k. step 0-6, a.k. main program

General description

1. **The brain surface is modeled by a surface tessellation using connected triangles.** The initial model is a tessellated sphere, generated by starting with an icosahedron and iteratively subdividing each triangle into four smaller triangles, while adjusting each vertex's distance from the centre to form as spherical a surface as possible. This is a common tessellation of the sphere. Each vertex has five or six neighbors, according to its position relative to the original icosahedron.
2. The **spherical tessellated surface** is initially centered on the COG, with its **radius set to half of the estimated brain/head radius**, i.e., intentionally small. **Allowing the surface to grow to the optimal estimate gives better results in general than setting the initial size to be equal to (or larger than) the estimated brain size (see Fig. 1)**. An example final surface mesh can be seen in Figure 2.

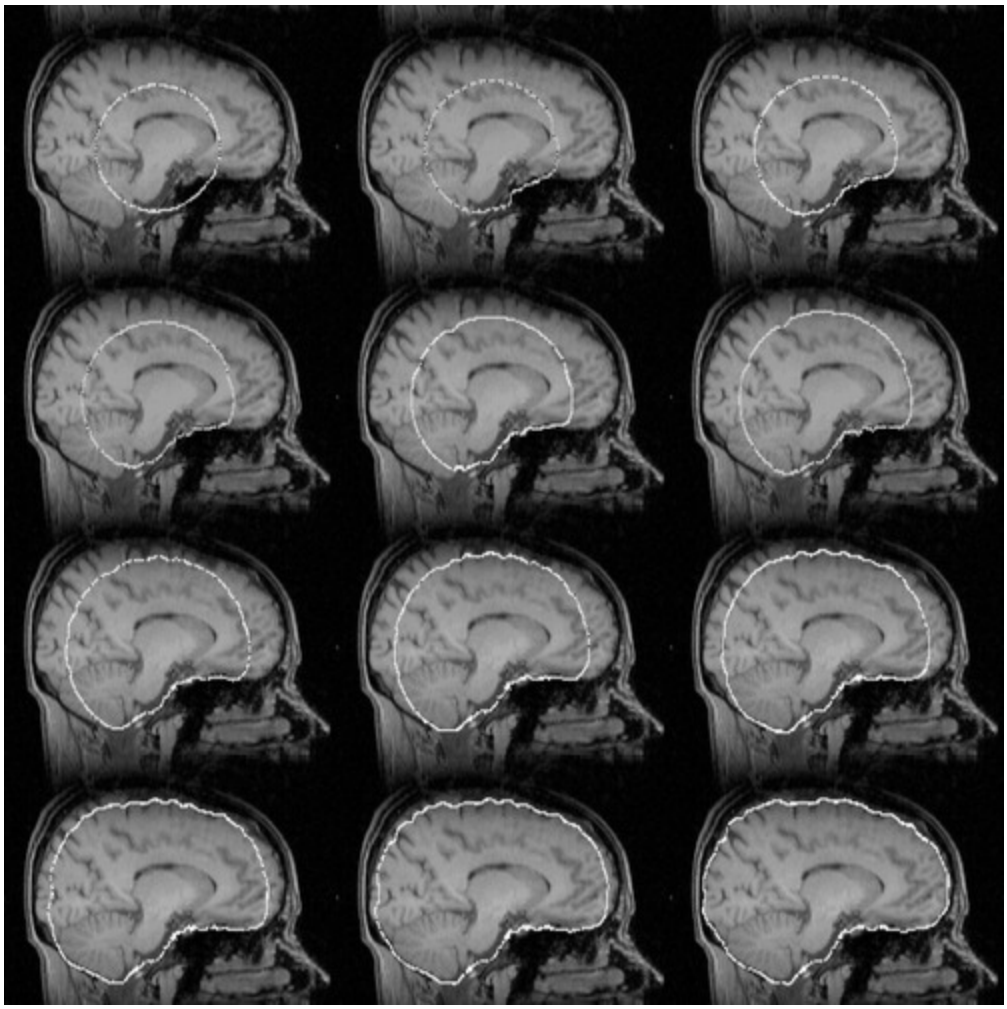


Figure 1. Example of surface model development as the main loop iterates. The dark points within the model outline are vertices.

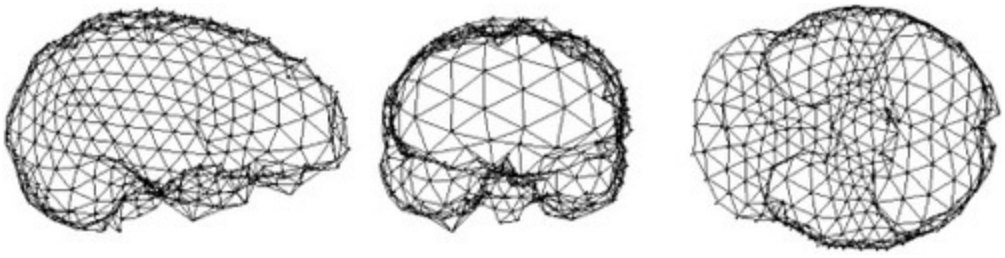


Figure 2. Three views of a typical surface mesh, shown for clarity with reduced mesh density.

The **vertex positions are in real (floating point) space**, i.e., not constrained to the voxel grid points. A major reason for this is that making **incremental**

(small) adjustments to vertex positions would not be possible otherwise. Another obvious advantage is that the image does not need to be pre-processed to be made up of cubic voxels.

- **Remark:**

  Sections 1 and 2 provide a general overview of the main program defined in the main function of the bet2.cpp file. In the following section, "**Step of Computation**", we will examine the essential surface modelling step, which involves calculating an **update movement vector u**. This process is contained within the step_of_computation function of bet2.cpp. **Grasping this step is vital in fully understanding the algorithm.**

P.S. In the Code Correspondence section, all steps (step 0-6) will be outlined, described (see the comments in the code) and attached to the corresponding code blocks in the *main* function from bet2.cpp.

Additionally, I highly recommend reading the "Step of Computation" section first before proceeding to review the "Code Correspondence" part.

▼ Code Correspondence (Including a Step-by-Step Description)

▼ **Step 0:** 2. Input image, or read in the image + 1. command-line options parsing

```
int main(int argc, char *argv[]) {

  //1. Parsing options
  OptionParser options(title, examples);
  options.add(outline);
  options.add(mask);
  options.add(skull);
  options.add(no_output);
  options.add(fractional_threshold);
  options.add(gradient_threshold);
```

```
  options.add(radiusarg);
  options.add(smootharg);

  options.add(centerarg);
  options.add(apply_thresholding);
  options.add(generate_mesh);
  options.add(verbose);
  options.add(help);

  if (argc < 3) {
    if (argc == 1) {options.usage(); exit(EXIT_FAILUR
    if (argc>1)
      {
    string s = argv[1];
    if (s.find("-h")!=string::npos | s.find("--help")
      {options.usage(); exit (0);}
      }
    cerr<<"error: not enough arguments, use bet -h fo
    exit (-1);
  }

  vector<string> strarg;
  for (int i=0; i<argc; i++)
    strarg.push_back(argv[i]);

  string inputname=strarg[1];
  string outputname=strarg[2];

  if (inputname.find("-")==0 || outputname.find("-")=
    {cerr<<"error : two first arguments should be inp

  /*
  int c=0;
  for (int i=0; i<argc; i++)
    if (i!=1 & i!=2)
      {
```

```
  strcpy(argv[c], strarg[i].c_str());
  c++;
    }

argc -=2;
*/

try {
  options.parse_command_line(argc, argv, 2);  // sk
}
catch(X_OptionError& e) {
  options.usage();
  cerr << endl << e.what() << endl;
  exit(EXIT_FAILURE);
}
catch(std::exception &e) {
  cerr << e.what() << endl;
}

if (help.value()) {options.usage(); return 0;};

string out = outputname;
if (out.find(".hdr")!=string::npos) out.erase(out.f
if (out.find(".img")!=string::npos) out.erase(out.f
string in = inputname;
if (in.find(".hdr")!=string::npos) in.erase(in.find
if (in.find(".img")!=string::npos) in.erase(in.find
if (out == "default__default") {out=in+"_brain";}


//set a memory hanlder that displays an error messa
set_new_handler(noMoreMemory);


//the real program
```

```
  //2. Read in the image
  volume<float> testvol;
  if (read_volume(testvol,in.c_str())<0)  return -1;
```

▼ **Step 1, 2, 3:** 2. Evaluation of t2, t98, t, tm, cog (Center Of Gravity), radius and the following 1. initialisation of tesselated surface

```
double xarg = 0, yarg = 0, zarg = 0;
  if (centerarg.set()) //if the cog is preset
    {
      /*
      if (centerarg.value().size()!=3)
    {
      cerr<<"three parameters expected for center opt
      cerr<<"please check there is no space after com
      exit (-1);
    }
      else
      */
    {
      xarg = (double) centerarg.value(0);
      yarg = (double) centerarg.value(1);
      zarg = (double) centerarg.value(2);
      ColumnVector v(4);
      v << xarg << yarg << zarg << 1.0;
      v = testvol.niftivox2newimagevox_mat() * v;
      xarg = v(1);  yarg = v(2);  zarg = v(3);
    }
    }

  const double bet_main_parameter = pow(fractional_th

  float originalX(testvol.xdim()),originalY(testvol.y
  // 2D kludge (worked for bet, but not here in bet2,
```

```
if (testvol.xsize()*testvol.xdim()<20) testvol.setx
if (testvol.ysize()*testvol.ydim()<20) testvol.sety
if (testvol.zsize()*testvol.zdim()<20) testvol.setz

//declaration of the mesh, wich will represent a te
Mesh m;

/*
   initialisation of the tesselated surface,
   generated by starting with an icosahedron and it
   triangle into four smaller triangles, while adju
   from the centre to form as spherical a surface a
*/
make_mesh_from_icosa(5, m);

//adjusting the tessellated surface according to th
bet_parameters bp = adjust_initial_mesh(testvol, m,
//P.S. t, t2, t98 are also evaluated in adjust_init

//just a prints out the priviouse calculation if wa
if (verbose.value())
  {
    cout<<"min "<<bp.min<<" thresh2 "<<bp.t2<<" thr
    cout<<"c-of-g "<<bp.cog.X<<" "<<bp.cog.Y<<" "<<
    cout<<"radius "<<bp.radius<<" mm"<<endl;
    cout<<"median within-brain intensity "<<bp.tm<<
  }
```

▼ **Step 4, 5:** Update the surface iteratively until it reaches the edges of the brain mass.

```
Mesh moriginal=m; //will be used in verifying self-i

 //preparetion for the step of computation
 const double rmin=3.33 * smootharg.value();
```

```
const double rmax=10 * smootharg.value();
const double E = (1/rmin + 1/rmax)/2.;
const double F = 6./(1/rmin - 1/rmax);
const int nb_iter = 1000;
const double self_intersection_threshold = 4000;
double l = 0;

/*
In the following code snippet:
Each vertex in the tessellated surface is updated b
that vertex should move to, to improve the surface.
incrementing each vertex position by the value of t
which will be evaluated for each vertex in the foll

To find an optimal solution, each individual moveme
(typically 1,000) iterations of each complete incre
*/
for (int i=0; i<nb_iter; i++)
  {
    step_of_computation(testvol, m, bet_main_parame
  }
```

▼ **Step 6:** Verify for self-intersection and resolve any identified issues.

```
  //Verify for self-intersection
  double tmp = m.self_intersection(moriginal);
if (verbose.value() && !generate_mesh.value())
  cout<<"self-intersection total "<<tmp<<" (thresho

bool self_intersection;
if (!generate_mesh.value()) self_intersection = (tm
else (self_intersection = m.real_self_intersection(
int pass = 0;

if (verbose.value() && generate_mesh.value() && sel
```

```
    cout<<"the mesh is self-intersecting "<<endl;


  //self-intersection treatment
    /*
    Treatment description
    If the surface is found to self-intersect, the al
    much higher smoothness constraint (applied to con
    it is not necessary for the convex parts) for the
    the smoothness weighting then linearly drops down
    over the remaining iterations. This results in pr
    in almost all cases.
    */

  while (self_intersection)
    {
      if (self_intersection && verbose.value()) {cout
      m = moriginal;
      l = 0;
      pass++;
      for (int i=0; i<nb_iter; i++)
    {
      double incfactor = pow (10.0,(double) pass + 1)
      if (i > .75 * (double)nb_iter)
        incfactor = 4.*(1. - i/(double)nb_iter) * (in
      //P.S. infactore corresponds to increase_smooth
      step_of_computation(testvol, m, bet_main_parame
    }
      double tmp = m.self_intersection(moriginal);

      self_intersection = (tmp > self_intersection_th
      if (!generate_mesh.value()) self_intersection =
      else (self_intersection = m.real_self_intersect

      if (verbose.value() && !generate_mesh.value())
    cout<<"self-intersection total "<<tmp<<" (thresho
```

```
    if (verbose.value() && generate_mesh.value() &&
  cout<<"the mesh is self-intersecting"<<endl;

    if (pass==10) // give up
  self_intersection=0;
  }
```

▼ **Display** the result considering command-line options

```
//display
  volume<short> brainmask = make_mask_from_mesh(testv

  if (apply_thresholding.value())
    {
      int xsize = testvol.xsize();
      int ysize = testvol.ysize();
      int zsize = testvol.zsize();
      for (int k=0; k<zsize; k++)
    for (int j=0; j<ysize; j++)
      for (int i=0; i<xsize; i++)
        if (testvol.value(i, j, k) < bp.t) brainmask.
    }

  if (!(no_output.value()))
    {
      int xsize = testvol.xsize();
      int ysize = testvol.ysize();
      int zsize = testvol.zsize();
      volume<float> output = testvol;
      for (int k=0; k<zsize; k++)
    for (int j=0; j<ysize; j++)
      for (int i=0; i<xsize; i++)
        output.value(i, j, k) = (1-brainmask.value(i,
      output.setdims(originalX,originalY,originalZ);
```

```
      if (save_volume(output,out.c_str())<0)  return
    }

  if (mask.value())
    {
      string maskstr = out+"_mask";
      brainmask.setDisplayMaximumMinimum(1,0);
      brainmask.setdims(originalX,originalY,originalZ
      if (save_volume((short)1-brainmask, maskstr.c_s
    }

  if (outline.value())
    {
      string outlinestr = out+"_overlay";
      volume<float> outline = draw_mesh_bis(testvol,
      outline.setdims(originalX,originalY,originalZ);
      if (save_volume(outline, outlinestr.c_str())<0)
    }

  if (generate_mesh.value())
    {
      string meshstr = out+"_mesh.vtk";
      m.save(meshstr.c_str(),3);
    }

  if (skull.value())
    {
      string skullstr = out+"_skull";
      volume<float> skull = find_skull(testvol, m, bp

      volume<short> bskull;
      copyconvert(skull,bskull);
      bskull.setdims(originalX,originalY,originalZ);
      if (save_volume(bskull, skullstr.c_str())<0)  r
    }
```

```
  return 0;

} //end of the main function
```

> lines 654 - 908 (main part) → *main* function from *bet2.cpp* and lines 149 - 230 (secondary functions)

▼ Step of Computation, a.k. Evaluation of the update movement vector u, a.k. the essence of step 4 and step 5

▼ Main iterated loop

Each vertex in the tessellated surface is updated by estimating where best that vertex should move to, to improve the surface. To find an optimal solution, each individual movement is small, with many (typically 1,000) iterations of each complete incremental surface update. In this context, “small movement” means small relative to the mean distance between neighboring vertices. Thus for each vertex, a small update movement vector **u** is calculated, using the following steps.

P.S.

```
//loop from step_of_computation function
for (vector<Mpoint*>::iterator i = m._points.begin(); i!

//main iterated loop defined in the main function
for (int i=0; i<nb_iter; i++)
    {
      step_of_computation(testvol, m, bet_main_parameter
    }
```

The loops from the *step_of_computation* function, paired with the nb_iter variable from the main function, iterate over all existing vertices of the mesh as many times as specified by the nb_iter value, as described in the previous section.

▼ 1. Local surface normal

First, the local unit vector surface normal n̂ is found. Each consecutive pair of [central vertex]–[neighbor A], [central vertex] – [neighbor B] vectors is taken and used to form the vector product (Fig. 1). The vector sum of these vectors is scaled to unit length to create **n̂**. By initially taking the sum of normal vectors *before* rescaling to unity, the sum is made relatively robust; the smaller a particular [central vertex]–[neighbor A]–[neighbor B] triangle is, the more poorly conditioned is the estimate of normal direction, but this normal will contribute less toward the sum of normals.

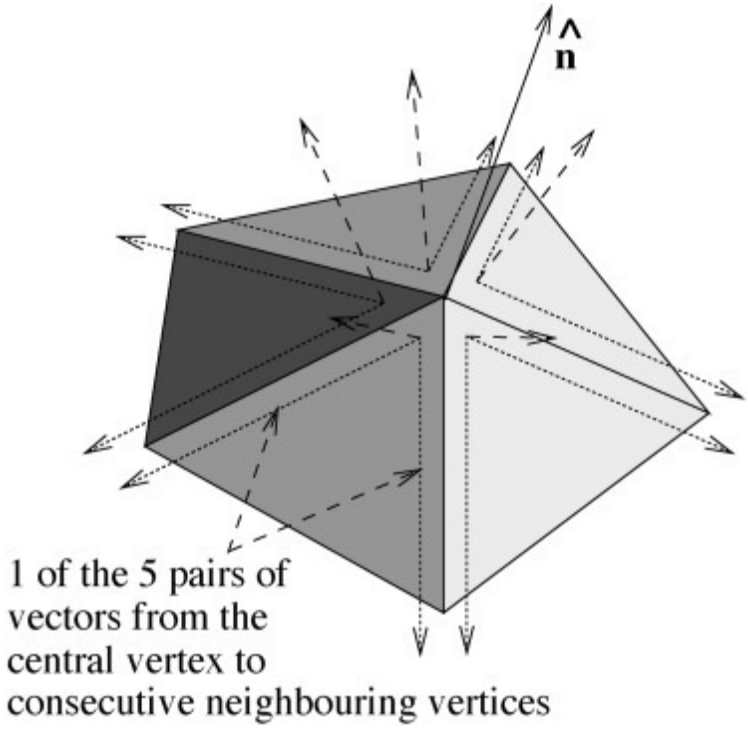


Figure 1

P.S. **n̂** is crucial for further calculations, which **will determine the evaluation of the update movement vector u.**

▼ 2. Mean position of neighbors and difference vector s

2.1. The next step is the calculation of the mean position of all vertices neighboring the vertex in question. This is used to find a difference vector s (dv variable in code), the vector that takes the current vertex to

the mean position of its neighbors. **If this vector were minimized for all vertices (by positional updates), the surface would be forced to be smooth and all vertices would be equally spaced. Also, due to the fact that the surface is closed, the surface would gradually shrink.**

2.2. Next, s is decomposed into orthogonal components, normal and tangential to the local surface;

$$\mathbf{s}_n = (\mathbf{s}.\hat{\mathbf{n}})\hat{\mathbf{n}}$$

and

$$\mathbf{s}_t = \mathbf{s} - \mathbf{s}_n.$$

For the 2D case, see Figure 2 (the extension to 3D is conceptually trivial). It is these two orthogonal vectors that form the basis for the three components of the vertex's movement vector **u**; **these components (s, st, sn) will be combined, with relative weightings, to create an update vector u for every vertex in the surface. The three components of u are described in the following sections.**

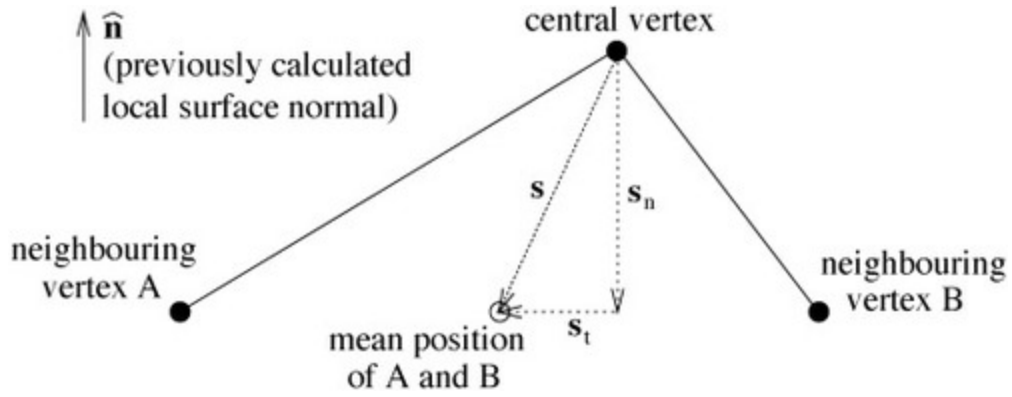


Figure 2. Decomposing the "perfect smoothness" vector **s** into components normal and tangential to the local surface.

▼ 3. Update component I: within-surface vertex spacing

The simplest component of the update movement vector **u** is **u1**, the component that is tangential to the local surface. **Its sole role is to keep**

all vertices in the surface equally spaced, moving them only within the surface. Thus, u1 is directly derived from st. To give simple stability to the update algorithm, **u1** is not set equal to **st**, but to **st**/2; the current vertex is always tending toward the position of perfect within-surface spacing (as are all others).

▼ 4. Update component 2: surface smoothness control

The remaining two components of **u** act parallel to the local surface normal. **4.1.The first, u2, is derived directly from sn, and acts to move the current vertex into line with its neighbors, thus increasing the smoothness of the surface**. A simple rule here would be to take a constant fraction of **sn**, in a manner equivalent to that of the previous component **u2**:

$$\mathbf{u}_2 = \mathrm{f}_2 \mathbf{s}_n$$

where *f*2 is the fractional update constant. Most other methods of surface modelling have taken this approach. A great improvement can be made, however, using a nonlinear function of **sn**. **The primary aim is to smooth high curvature in the surface model more heavily than low curvature.** The reason for this is that although high curvature is undesirable in the brain surface model, forcing surface smoothing to an extent that gives stable and good results (in removing high curvature) weights too heavily against successful following of the low curvature parts of the surface. To keep the surface model sufficiently smooth for the overall algorithm to proceed stably, the surface is forced to be over-smooth, causing the underestimation of curvature at certain parts, i.e., the "cutting of corners." **It has been found that this problem is not overcome by allowing *f*2 to vary during the series of iterations (this a natural improvement on a constant update fraction). 4.2.Instead, a nonlinear function is used, starting by finding the local radius of curvature, *r*:**

$$r = \frac{l^2}{2|\mathbf{s}_n|},$$

where *l* is the mean distance from vertex to neighboring vertex across the whole surface. 4.3.Now, a sigmoid function of r is applied, to find the update fraction:

$$f_2 = (1 + \tanh(F * (1/r - E)))/2,$$

4.4.where *E* and *F* control the scale and offset of the sigmoid. These are derived from a minimum and maximum radius of curvature; below the minimum *r*, heavy smoothing takes place (i.e., the surface deformation remains stable and highly curved features are smoothed), whereas above the maximum *r*, little surface smoothing is carried out (i.e., "long slow" curves are not over-smoothed). **The empirically optimized values for *rmin* and *rmax* are suited for typical geometries found in the human brain.** Consideration of the *tanh* function suggests:

$$E = (1/r_{min} + 1/r_{max})/2,$$

and

$$F = 6/(1/r_{min} - 1/r_{max}).$$

The resulting smoothness term gives much better results than a constant update fraction, both in ability to accurately model brain surface and in developmental stability during the many iterations.

▼ 5. Update component 3: brain surface selection term

The final update component, **u3**, is also parallel to **sn**, and **is the term that actually interacts with the image, attempting to force the surface model to fit to the real brain surface.**

5.1. First, along a line pointing inward from the current vertex, minimum and maximum intensities are found:

$$I_{min} = MAX(t_2, MIN\,(t_m, I(0), I(1), \ldots, I(d_1))),$$

$$I_{max} = MIN(t_m, MAX\,(t, I(0), I(1), \ldots, I(d_2))),$$

where *d*1 determines how far into the brain the minimum intensity is searched for, and d2 determines how far into the brain the maximum intensity is searched for. Typically, *d*1 = 20 *mm* and *d*2 = *d*1/2 (this ratio is empirically optimized, and reflects the relatively larger spatial reliability of the search for maximum intensity compared with the minimum). ***tm*, *t*2, and *t* are used to limit the effect of very dark or very bright voxels, and *t* is included in the maximum intensity search to limit the effect of very bright voxels.**

5.2. Now, *Imax* is used to create *tI*, a locally appropriate intensity threshold that distinguishes between brain and background:

$$t_1 = (I_{max} - t_2) * b_t + t_2.$$

It lies at a predetermined fraction of the distance between the global robust low-intensity threshold ***t*2** and the local maximum intensity ***I*max**, determined by the fractional constant ***b*t**. This preset constant is the main parameter that BET can take as input. **5.2.1. The default value of 0.5 has been found to give excellent results for most input images.** For certain image intensity distributions it can be varied (in the range 0–1) to give optimal results. The necessity for this is rare, and for an MR sequence that requires changing *bt*, one value normally works for all other images taken with the same sequence (the only other input parameter, and one that needs changing from the default even less often than *bt*, for example, if there is very strong vertical intensity inhomogeneity, causes *bt* to vary linearly with *Z* in the image, causing

"tighter" brain estimation at the top of the brain, and "looser" estimation at the bottom, or vice versa).

5.3. **The update "fraction" is then given by:**

$$f_3 = \frac{2(I_{min} - t_1)}{I_{max} - t_2},$$

with the factor of 2 causing a range in values of *f*3 of roughly −1 to 1. **If *Imin* is lower than local threshold *tl*, *f*3 is negative, causing the surface to move inward at the current point (in other words breaks, if the edge of the brain surface was reached). If it is higher, then the surface moves outward.**

5.4. **The full update term is 0.05 *f*3 *l* $\hat{sn}$.** The update fraction is multiplied by a relative weighting constant, 0.05, and the mean inter-vertex distance, *l*. The weighting constant sets the balance between the smoothness term and the intensity-based term, it is found empirically, but because all terms in BET are invariant to changes in image voxel size, image contrast, mesh density, etc., this constant is not a "worked-once" heuristic, it is always appropriate.

P.S. In fact, the term was slightly altered (see the corresponding code).

▼ 6. Final update equation

The total update equation, for each vertex, is

$$\mathbf{u} = 0.5\mathbf{s}_t + f_2\mathbf{s}_n + 0.05 f_3 l \hat{\mathbf{s}}_n.$$

▼ Code Correspondence (Including a Step-by-Step Description)

▼ 1. Evaluation of local unit vector surface normal $\hat{\mathbf{n}}$

▼ 1.1 Initialising $\hat{\mathbf{n}}$ with a return value of *local_normal* func.

```
double step_of_computation(const volume<float> & i
  double xdim = image.xdim();
```

```
  double ydim = image.ydim();
  double zdim = image.zdim();
  double dscale = Min(Min(Min(xdim,ydim),zdim),1.0
  //cout << xdim << " " << ydim << " " << zdim <<

  if (iteration_number==50 || iteration_number%100
    {
      l = 0;
      int counter = 0;
      for (vector<Mpoint*>::iterator i = m._points
    {
      counter++;
      l += (*i)->medium_distance_of_neighbours();
    }
      l/=counter;
    }

  for (vector<Mpoint*>::iterator i = m._points.beg
    {
      Vec sn, st, u1, u2, u3, u;
      double f2, f3=0;

      Vec n = (*i)->local_normal();
```

▼ 1.1 Definition of *local_normal* func. and normalisation

```
const Vec Mpoint::local_normal() const
{
  Vec v(0, 0, 0);
  for (list<Triangle*>::const_iterator i = _triang
    {
      v+=(*i)->normal();
    }
  v.normalize();
  return v;
```

```
}

//normalisation: point.h from meshclass, line 130
void normalize(){
    double n = norm();
    if (n!=0){
    X/=n;
    Y/=n;
    Z/=n;}
  }
```

So, as you can see, a vector product described in section 1 will be defined in a *normal()* function.

▼ 1.2 Vector product

```
const Vec Triangle::normal() const{
  Vec result = (_vertice[2]->get_coord() - _vertic
  return result;
}
```

▼ 2. Calculate the difference vector s, and derive st and **sn**, which are crucial for the evaluation of **u**.

▼ 2.1 Calculation of **s**

▼ 2.1.1. Initialising s (or dv) with a return value of the *difference_vector()* func.

```
Vec dv = (*i)->difference_vector();
```

▼ 2.1.2. definition of *difference_vector()* func.

```
const Vec Mpoint::difference_vector() const
{
  return medium_neighbours() - _coord;
}
```

▼ 2.1.3. Evaluation of medium neighbours coords

```
const Pt Mpoint::medium_neighbours() const
{
  Pt resul(0, 0, 0);
  int counter=_neighbours.size();
  for (list<Mpoint*>::const_iterator i = _neigh
    {
      resul+=(*i)->_coord;
    }
  resul=Pt(resul.X/counter, resul.Y/counter, re
  return resul;
}
```

▼ 2.2 Decomposition of **s** into **sn** and **st**

```
double tmp = dv|n;
sn = n * tmp;
st = dv - sn;
```

▼ 3. Calculation of **u1**

```
u1 = st*.5;
```

▼ 4. Calculation of **u2**

▼ 4.1. Term for **u2** in code

```
u2 = f2 * sn;
```

▼ 4.2. Evaluation of a local radius of curvature, **r**

```
//evaluation of the mean distance from vertex to n
//bet.cpp, line 418
if (iteration_number==50 || iteration_number%100 =
    {
```

```
      l = 0;
      int counter = 0;
      for (vector<Mpoint*>::iterator i = m._points
    {
      counter++;
      l += (*i)->medium_distance_of_neighbours();
    }
      l/=counter;
    }

//evaluation of the local radius of curvature.
//bet2.cpp, line 444
double rinv = (2 * fabs(sn|n))/(l*l);
```

▼ 4.3. Evaluation of the update fraction **f2**

```
double tmp = dv|n; //bet2.cpp, line 438

//bet2.cpp, line 446
 f2 = (1+tanh(F*(rinv - E)))*0.5;
      if (pass > 0) //in case of self-intersaction
    if (tmp > 0) {
      f2*=increase_smoothing;
      f2 = Min(f2, 1.);
    }
```

▼ 4.4. Evaluation of **E** and **F**

```
  const double rmin=3.33 * smootharg.value();
  const double rmax=10 * smootharg.value();
  const double E = (1/rmin + 1/rmax)/2.;
  const double F = 6./(1/rmin - 1/rmax);
```

P.S. The 'smootharg' will be provided from the command-line:

```
Option<float> smootharg(string("-w,--smooth"), 1.0
                    string("~<r>\tsmoothness fact
```

▼ 5. Calculation of **u3**

▼ 5.1. Evaluation of the minimum and maximum intensities **Imin** and **Imax**

```
  double Imin = tm;
    double Imax = t;

    Pt p = (*i)->get_coord() + (-1)*n;
    double iv = p.X/xdim + .5, jv = p.Y/ydim +.5,
    if (image.in_bounds((int)iv,(int) jv,(int) kv)
      {
        double im=image.value((int)iv,(int)jv,(int
        Imin = Min(Imin, im);
        Imax = Max(Imax,im);

        double nxv=n.X/xdim, nyv=n.Y/ydim, nzv=n.Z
        int i2=(int)(iv-(d1-1)*nxv), j2 =(int) (jv
        nxv*=dscale; nyv*=dscale; nzv*=dscale;
        if (image.in_bounds(i2, j2, k2))
          {
        im=image.value(i2,j2,k2);
        Imin = Min(Imin, im);

        for (double gi=2.0; gi<d1; gi+=dscale)
          {
            //cout << gi << " " << endl;
            // the following is a quick calc of Pt
            iv-=nxv; jv-=nyv; kv-=nzv;
            im = image.value((int) (iv), (int) (jv
            Imin = Min(Imin, im);
```

```
            if (gi<d2)
              Imax = Max(Imax,im);
          }

        Imin = Max (t2, Imin);
        Imax = Min (tm, Imax);
```

▼ 5.2. Evaluation of the locally appropriate intensity threshold **tl**

```
//bet2.cpp, line 457
double local_t = bet_main_parameter; //given in ar
    if (local_th != 0.0) //when image intensity di
      {
           //5.2.1.
        local_t = Min(1., Max(0., bet_main_paramet
      }


//... sections 1, 2, 3, 4, 5.1

//bet2.cpp, line 497
const double tl = (Imax - t2) * local_t + t2;
```

▼ 5.3. Evaluation of the update fraction **f3**

```
if (Imax - t2 > 0)
          f3=2*(Imin - tl)/(Imax - t2);
        else f3=(Imin - tl)*2;
```

▼ 5.4. Evaluation of **u3**

```
f3 *= (normal_max_update_fraction * lambda_fit * l
```

```
        u3 = f3 * n;
```

▼ 6. Calculation of **u**

```
u = u1 + u2 + u3;

      //cout<<"l "<<l<<"u1 "<<((u1*n).norm())<<"u2 "<

      (*i)->_update_coord = (*i)->get_coord() + u;
    }

  m.update();

  return (0);
} //end of the step_of_computation function
```

> lines 411 - 524 → *step_of_computation* function from *bet2.cpp*

▼ Exterior skull surface estimation

A few applications require the estimation of the position of the skull in the image. A major example is in the measurement of brain atrophy [Smith et al., 2001]. Before brain change can be measured, two images of the brain, taken several months apart, have to be registered. Clearly this registration cannot allow rescaling, otherwise the overall atrophy will be underestimated. Because of possible changes in scanner geometry over time, however, it is necessary to hold the scale constant somehow. This can be achieved using the exterior skull surface, which is assumed to be relatively constant in size, as a scaling constraint in the registration.

In most MR images, the skull appears very dark. In T1-weighted images the internal surface of the skull is largely indistinguishable from the cerebrospinal fluid (CSF), which is also dark. Thus the exterior surface is found. This also can be difficult to identify, even for trained clinical experts, but the algorithm is largely successful in its aim.

For each voxel lying on the brain surface found by BET, a line perpendicular to the local surface, pointing outward, is searched for the exterior surface of the skull, according to the following algorithm:

- Search outward from the brain surface, a distance of 30 mm, recording the maximum intensity and its position, and the minimum intensity.
- If the maximum intensity is not higher than *t*, assume that the skull is not measurable at this point, as there is no bright signal (scalp) on the far side of the skull. In this case do not proceed with the search at this point in the image. This would normally be due to signal loss at an image extreme, for example, at the top of the head.
- Find the point at greatest distance from brain surface *d* that has low intensity, according to maximisation of the term $d/30 - I(d)/(t98 - t2)$. The first part weighs in favor of increased distance, the second part weighs in favor of low intensity. The search is continued only out to the previously found position of maximum intensity. The resulting point should be close to the exterior surface of the skull.
- Finally, search outward from the previous point until the first maximum in intensity gradient is found. This is the estimated position of the exterior surface of the skull. This final stage gives a more well-defined position for the surface, it does not depend on the weightings in the maximised term in the previous section, i.e., is more objective. For example, if the skull/scalp boundary is at all blurred, the final position will be less affected than the previous stage.

This method has been quite successful, even when fairly dark muscle lies between the skull and the brighter skin and fat. It is also mainly successful in ignoring the marrow within the bone, which sometimes is quite bright.

P.S. This part of the algorithm has not been extensively analysed since it is not utilised for the primary purpose of brain extraction. It serves solely as an optional feature.

▼ Code Correspondence

```
volume<float> find_skull (volume<float> & image, const M
{
```

```
  const double skull_search = 30;
  const double skull_start = -3;

  volume<float> result = image;
  result=0;

  volume<short> volmesh;
  copyconvert(image,volmesh);
  int xsize = volmesh.xsize();
  int ysize = volmesh.ysize();
  int zsize = volmesh.zsize();
  double xdim = volmesh.xdim();
  double ydim = volmesh.ydim();
  double zdim = volmesh.zdim();
  double scale = Min(xdim, Min(ydim, zdim));
  volmesh = 1;
  volmesh = draw_mesh(volmesh, m);

  image.setinterpolationmethod(trilinear);

  for (vector<Mpoint*>::const_iterator i = m._points.beg
    {
      double max_neighbour = 0;
      const Vec normal = (*i)->local_normal();
      const Vec n = Vec(normal.X/xdim, normal.Y/ydim, no

      for (list<Mpoint*>::const_iterator nei = (*i)->_ne
    max_neighbour = Max(((**i) - (**nei)).norm(), max_ne

      max_neighbour = ceil((max_neighbour)/2);

      const Pt mpoint((*i)->get_coord().X/xdim,(*i)->get
      for (int ck = (int)floor(mpoint.Z - max_neighbour/
    for (int cj = (int)floor(mpoint.Y - max_neighbour/yd
      for (int ci = (int)floor(mpoint.X - max_neighbour/
        {
```

```
            bool compute = false;
            const Pt point(ci, cj, ck);
            const Pt realpoint(ci*xdim, cj*ydim, ck*zdim);
            if (volmesh(ci, cj, ck) == 0)
          {
            double mindist = 10000;
            for (list<Mpoint*>::const_iterator nei = (*i)-
              mindist = Min(((realpoint) - (**nei)).norm()
            if (mindist >= ((realpoint) - (**i)).norm()) c
          }



            if (compute)
          {
            double maxval = t;
            double minval = image.interpolate(point.X, poi
            double d_max = 0;

            for (double d=0; d<skull_search; d+=scale*.5)
              {
                Pt current = point + d * n;
                double val = image.interpolate(current.X,
                if (val>maxval)
              {
                maxval=val;
                d_max=d;
              }

                if (val<minval)
              minval=val;
              }

            if (maxval > t)
              {
                double d_min=skull_start;
                double maxJ =-1000000;
```

```
                double lastJ=-2000000;
                for(double d=skull_start; d<d_max; d+=scal
              {
                Pt current = point + d * n;
                if (current.X >= 0 && current.Y >= 0 && cu
                  {
                    double tmpf = d/30 - image.interpolate
                    if (tmpf > maxJ)
                  {
                    maxJ=tmpf;
                    d_min = d;
                  }
                    lastJ=tmpf;
                  }
              }
                double maxgrad = 0;
                double d_skull;
                Pt current2 = point + d_min * n;
                if (current2.X >= 0 && current2.Y >= 0 &&
              {
                double val2 = image.interpolate(current2.X
                for(double d=d_min + scale; d<d_max; d+=0.
                  {
                    Pt current = point + d * n;
                    if (current.X >= 0 && current.Y >= 0 &
                  {
                    double val = image.interpolate(current
                    double grad = val - val2;
                    val2 = val;
                    if (grad > 0)
                      {
                        if (grad > maxgrad)
                      {
                        maxgrad=grad;
                        d_skull=d;
                      }
```

```
                        else d = d_max;
                    }
                }
                }
            }
              if (maxgrad > 0)
            {
              Pt current3 = point + d_skull * n;
              if (current3.X >= 0 && current3.Y >= 0 &&
                result ((int)current3.X, (int)current3.Y
            }
            }
        }
        }
    }
  return result;
}
```

> lines 525 - 653 → *find_skull* function from *bet2.cpp*

References

🌐 Source: https://fsl.fmrib.ox.ac.uk/fsl/fslwiki/BET/UserGuide#betgui

📄 Source: *S.M. Smith. Fast robust automated brain extraction. Human Brain Mapping, 17(3):143-155, November 2002.*