```
/*

Copyright 1995-2006 Roeland Merks, Nick Savill

This file is part of Tissue Simulation Toolkit.

Tissue Simulation Toolkit is free software; you can
redistribute
it and/or modify it under the terms of the GNU General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later
version.

Tissue Simulation Toolkit is distributed in the hope that it
will
be useful, but WITHOUT ANY WARRANTY; without even the implied
warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
See the GNU General Public License for more details.

You should have received a copy of the GNU General Public
License
along with Tissue Simulation Toolkit; if not, write to the
Free
Software Foundation, Inc., 51 Franklin St, Fifth Floor,
Boston, MA
02110-1301 USA

*/


/* CA.cpp: implementation of Glazier & Graner's Cellular Potts
Model */

// This code derives from a Cellular Potts implementation
written around 1995
// by Nick Savill

#include <stdio.h>
#include <math.h>
#include <cstdlib>
#include <cstring>
#include "sticky.h"
#include "random.h"
#include "ca.h"
#include "parameter.h"
#include "dish.h"
#include "sqr.h"
#include "crash.h"
#include "hull.h"
#include "random.h"

#define ZYGFILE(Z) <Z.xpm>
#define XPM(Z) Z ## _xpm
#define ZYGXPM(Z) XPM(Z)

/* define default zygote */
```

```
/* NOTE: ZYGOTE is normally defined in Makefile!!!!!! */
#ifndef ZYGOTE
#define ZYGOTE init
#include "init.xpm"
#else
#include ZYGFILE(ZYGOTE)
#endif

/* STATIC DATA MEMBER INITIALISATION */
double copyprob[BOLTZMANN];


const int CellularPotts::nx[25] = {0, 0, 1, 0,-1, 1, 1,-1,-1,
0, 2, 0, -2, 1, 2, 2, 1,-1,-2,-2,-1, 0, 2, 0,-2 };
const int CellularPotts::ny[25] = {0,-1, 0, 1, 0,-1, 1,
1,-1,-2, 0, 2,  0,-2,-1, 1, 2, 2, 1,-1,-2,-2, 0, 2, 0 };

const int CellularPotts::nbh_level[5] = { 0, 4, 8, 20, 24 };
int CellularPotts::shuffleindex[9]={0,1,2,3,4,5,6,7,8};

extern Parameter par;


/** PRIVATE **/

using namespace std;
void CellularPotts::BaseInitialisation(vector<Cell> *cells) {
  CopyProb(par.T);
  cell=cells;
  if (par.neighbours>=1 && par.neighbours<=4)
    n_nb=nbh_level[par.neighbours];
  else
    throw "Panic in CellularPotts: parameter neighbours
invalid (choose [1-4]).";

}

CellularPotts::CellularPotts(vector<Cell> *cells,
                      const int sx, const int sy) {

  sigma=0;
  frozen=false;
  thetime=0;
  zygote_area=0;


  BaseInitialisation(cells);
  sizex=sx;
  sizey=sy;

  AllocateSigma(sx,sy);


  // fill borders with special border state
  for (int x=0;x<sizex;x++) {
    sigma[x][0]=-1;
    sigma[x][sizey-1]=-1;
```

```
  }
  for (int y=0;y<sizey;y++) {
    sigma[0][y]=-1;
    sigma[sizex-1][y]=-1;
  }

  if (par.neighbours>=1 && par.neighbours<=4)
    n_nb=nbh_level[par.neighbours];
  else
    throw "Panic in CellularPotts: parameter neighbours
invalid (choose [1-4])";
}

CellularPotts::CellularPotts(void) {

  sigma=0;
  sizex=0; sizey=0;
  frozen=false;
  thetime=0;
  zygote_area=0;

  CopyProb(par.T);

  // fill borders with special border state
  for (int x=0;x<sizex;x++) {
    sigma[x][0]=-1;
    sigma[x][sizey-1]=-1;
  }
  for (int y=0;y<sizey;y++) {
    sigma[0][y]=-1;
    sigma[sizex-1][y]=-1;
  }
  if (par.neighbours>=1 && par.neighbours<=4)
    n_nb=nbh_level[par.neighbours];
  else
    throw "Panic in CellularPotts: parameter neighbours
invalid (choose [1-4])";
}

// destructor (virtual)
CellularPotts::~CellularPotts(void) {
  if (sigma) {
    free(sigma[0]);
    free(sigma);
    sigma=0;
  }
}

void CellularPotts::AllocateSigma(int sx, int sy) {

  sizex=sx; sizey=sy;

  sigma=(int **)malloc(sizex*sizeof(int *));
  if (sigma==NULL)
    MemoryWarning();

  sigma[0]=(int *)malloc(sizex*sizey*sizeof(int));
```

```
  if (sigma[0]==NULL)
    MemoryWarning();


  {for (int i=1;i<sizex;i++)
    sigma[i]=sigma[i-1]+sizey;}

  /* Clear CA plane */
   {for (int i=0;i<sizex*sizey;i++)
     sigma[0][i]=0; }

}

void CellularPotts::IndexShuffle() {

  int i;
  int temp;
  int index1,index2;

  for (i=0;i<9;i++) {

    index1=RandomNumber(8);
    index2=RandomNumber(8);

    temp=shuffleindex[index1];
    shuffleindex[index1]=shuffleindex[index2];
    shuffleindex[index2]=temp;

  }
}


double sat(double x) {

  return x/(par.saturation*x+1.);
  //return x;

}

int CellularPotts::DeltaH(int x,int y, int xp, int yp, PDE
*PDEfield)
{
  int DH = 0;
  int i, sxy, sxyp;
  int neighsite;

  /* Compute energydifference *IF* the copying were to occur
*/
  sxy = sigma[x][y];
  sxyp = sigma[xp][yp];

  /* DH due to cell adhesion */
  for (i=1;i<=n_nb;i++) {
    int xp2,yp2;
    xp2=x+nx[i]; yp2=y+ny[i];
    if (par.periodic_boundaries) {
```

```
      // since we are asynchronic, we cannot just copy the
borders once
      // every MCS

      if (xp2<=0)
      xp2=sizex-2+xp2;
      if (yp2<=0)
      yp2=sizey-2+yp2;
      if (xp2>=sizex-1)
      xp2=xp2-sizex+2;
      if (yp2>=sizey-1)
      yp2=yp2-sizey+2;

      neighsite=sigma[xp2][yp2];


    } else {

      if (xp2<=0 || yp2<=0
        || xp2>=sizex-1 || yp2>=sizey-1)
      neighsite=-1;
      else
      neighsite=sigma[xp2][yp2];

    }

    if (neighsite==-1) { // border
      DH += (sxyp==0?0:par.border_energy)-
      (sxy==0?0:par.border_energy);
    } else {
      DH += (*cell)[sxyp].EnergyDifference((*cell)[neighsite])
      - (*cell)[sxy].EnergyDifference((*cell)[neighsite]);
    }
  }


  // lambda is determined by chemical 0

  //cerr << "[" << lambda << "]";
  if ( sxyp == MEDIUM ) {
    DH += (int)(par.lambda *  (1. - 2. *
                        (double) ( (*cell)[sxy].Area() -
(*cell)[sxy].TargetArea()) ));
  }
  else if ( sxy == MEDIUM ) {
    DH += (int)((par.lambda * (1. + 2. *
                        (double) ( (*cell)[sxyp].Area() -
(*cell)[sxyp].TargetArea()) )));
  }
  else
    DH += (int)((par.lambda * (2.+  2.  * (double)
                        (  (*cell)[sxyp].Area() - (*cell)
[sxyp].TargetArea()
                        - (*cell)[sxy].Area() + (*cell)
[sxy].TargetArea() )) ));
/*Chemotaxis*/
  if (PDEfield && (par.vecadherinknockout || (sxyp==0 || sxy==
```

```
0))) {
    int DDH1 = 0;
   if (!( par.extensiononly && sxyp==0)) {
      int DDH=(int)(par.chemotaxis*(sat(PDEfield->
Sigma(0,x,y))-sat(PDEfield->Sigma(0,xp,yp))));
      if (PDEfield->Sigma(1,xp,yp)>1e-5) { // Maybe a new
parameter can be set here to see the effects of changing this.
       int DDH1= 5000; //can also be changed into
par.saturation in order to see the effects of changing this
variable.
        }
      DH += DDH1;
      DH-=DDH;
    }
  }


  const double lambda2=par.lambda2;

  /* Length constraint */
  // sp is expanding cell, s is retracting cell


  if ( sxyp == MEDIUM ) {
    DH -= (int)(lambda2*( DSQR((*cell)[sxy].Length()-(*cell)
[sxy].TargetLength())
                  - DSQR((*cell)
[sxy].GetNewLengthIfXYWereRemoved(x,y) -
                       (*cell)[sxy].TargetLength()) ));

  }
  else if ( sxy == MEDIUM ) {
    DH -= (int)(lambda2*(DSQR((*cell)[sxyp].Length()-(*cell)
[sxyp].TargetLength())
                 -DSQR((*cell)
[sxyp].GetNewLengthIfXYWereAdded(x,y)-(*cell)
[sxyp].TargetLength())));


  }
  else {
    DH -= (int)(lambda2*( (DSQR((*cell)[sxyp].Length()-(*cell)
[sxyp].TargetLength())
                -DSQR((*cell)
[sxyp].GetNewLengthIfXYWereAdded(x,y)-(*cell)
[sxyp].TargetLength())) +
               ( DSQR((*cell)[sxy].Length()-(*cell)
[sxy].TargetLength())
                 - DSQR((*cell)
[sxy].GetNewLengthIfXYWereRemoved(x,y) -
                      (*cell)[sxy].TargetLength()) ) ));
  }

  return DH;
}
```

```
bool CellularPotts::Probability(int DH)
{
  if ( DH > BOLTZMANN-1 )
    return false;
  else if ( DH < 0 || RANDOM() < copyprob[DH] )
    return true;
   return false;
}

void CellularPotts::ConvertSpin(int x,int y,int xp,int yp)
{
  int tmpcell;
  if ( (tmpcell=sigma[x][y]) ) { // if tmpcell is not MEDIUM
    (*cell)[tmpcell].DecrementArea();
    (*cell)[tmpcell].RemoveSiteFromMoments(x,y);

    if (!(*cell)[tmpcell].Area()) {
      (*cell)[tmpcell].Apoptose();
      cerr << "Cell " << tmpcell << " apoptosed\n";
    }
  }

  if ( (tmpcell=sigma[xp][yp]) ) {// if tmpcell is not MEDIUM
    (*cell)[tmpcell].IncrementArea();
    (*cell)[tmpcell].AddSiteToMoments(x,y);

  }
  sigma[x][y] = sigma[xp][yp];


}


/** PUBLIC **/
int CellularPotts::CopyvProb(int DH,  double stiff) {

  double dd;
  int s;
  s=(int)stiff;
  if (DH<=-s) return 2;

  // if DH becomes extremely large, calculate probability on-
the-fly
  if (DH+s > BOLTZMANN-1)
    dd=exp( -( (double)(DH+s)/par.T ));
  else
    dd=copyprob[DH+s];

  if (RANDOM()<dd) return 1; else return 0;
}

void CellularPotts::CopyProb(double T) {
  int i;
  for ( i = 0; i < BOLTZMANN; i++ )
    copyprob[i] = exp( -( (double)(i)/T ) );
}
```

```
void CellularPotts::FreezeAmoebae(void)
{
  if (frozen)
    frozen=FALSE;
  else
    frozen=TRUE;
}

#include <fstream>
//! Monte Carlo Step. Returns summed energy change
int CellularPotts::AmoebaeMove(PDE *PDEfield)
{

  int loop,p;
  //int updated=0;
  thetime++;
  int SumDH=0;

  if (frozen)
    return 0;

  loop=(sizex-2)*(sizey-2);

  for (int i=0;i<loop;i++) {

    // take a random site
    int xy = (int)(RANDOM()*(sizex-2)*(sizey-2));
    int x = xy%(sizex-2)+1;
    int y = xy/(sizex-2)+1;

    // take a random neighbour
    int xyp=(int)(n_nb*RANDOM()+1);
    int xp = nx[xyp]+x;
    int yp = ny[xyp]+y;

    int k=sigma[x][y];

    int kp;
    if (par.periodic_boundaries) {

      // since we are asynchronic, we cannot just copy the
borders once
      // every MCS


      if (xp<=0)
      xp=sizex-2+xp;
      if (yp<=0)
      yp=sizey-2+yp;
      if (xp>=sizex-1)
      xp=xp-sizex+2;
      if (yp>=sizey-1)
      yp=yp-sizey+2;

      kp=sigma[xp][yp];
```

```
    } else {

      if (xp<=0 || yp<=0
        || xp>=sizex-1 || yp>=sizey-1)
      kp=-1;
      else
      kp=sigma[xp][yp];

    }


    // test for border state (relevant only if we do not use
    // periodic boundaries)
    if (kp!=-1) {
      // Don't even think of copying the special border state
into you!


      if ( k  != kp ) {

      /* Try to copy if sites do not belong to the same cell */

      // connectivity dissipation:
      int H_diss=0;
      if (!ConnectivityPreservedP(x,y)) H_diss=par.conn_diss;


      int D_H=DeltaH(x,y,xp,yp,PDEfield);

      if ((p=CopyvProb(D_H,H_diss))>0) {
        ConvertSpin ( x,y,xp,yp );
        SumDH+=D_H;
      }

       }
    }
  }

  return SumDH;

}

/** A simple method to plot all sigma's in window
    without the black lines */
void CellularPotts::PlotSigma(Graphics *g, int mag) {

  for (int x=0;x<sizex;x++)
    for (int y=0;y<sizey;y++) {
      for (int xm=0;xm<mag;xm++)
      for (int ym=0;ym<mag;ym++)
      g->Point( sigma[x][y], mag*x+xm, mag*y+ym);
  }

}

int **CellularPotts::SearchNandPlot(Graphics *g, bool
```

```
get_neighbours)
{
  int i, j,q;
  int **neighbours=0;


  /* Allocate neighbour matrix */
  if (get_neighbours) {
    neighbours=(int **)malloc((cell->size()+1)*sizeof(int *));
    if (neighbours==NULL)
      MemoryWarning();

    neighbours[0]=(int *)malloc((cell->size()+1)*(cell->
size()+1)*sizeof(int));
    if (neighbours[0]==NULL)
      MemoryWarning();

    for (i=1;i<(int)cell->size()+1;i++)
      neighbours[i]=neighbours[i-1]+(cell->size()+1);

    /* Clear this matrix */
    for (i=0;i<((int)cell->size()+1)*((int)cell->size()+
1);i++)
      neighbours[0][i]=EMPTY;
  }

  for ( i = 0; i < sizex-1; i++ )
    for ( j = 0; j < sizey-1; j++ ) {


      int colour;
      if (sigma[i][j]<=0) {
      colour=0;
      } else {
      colour = (*cell)[sigma[i][j]].Colour();
      //colour = sigma[i][j];
      }

      if (g && sigma[i][j]>0)  /* if draw */
        g->Point( colour, 2*i, 2*j);

      if ( sigma[i][j] != sigma[i+1][j] )  /* if cellborder */
/* etc. etc. */
      {
        if (g)
          g->Point( 1, 2*i+1, 2*j );
        if (get_neighbours) {
          if (sigma[i][j]>0) {
            for (q=0;q<(int)cell->size();q++)
            if (neighbours[sigma[i][j]][q]==EMPTY) {
              neighbours[sigma[i][j]][q]=sigma[i+1][j];
              break;
            }
            else
              if (neighbours[sigma[i][j]][q]==sigma[i+1][j])
                break;
          }
```

```
          if (sigma[i+1][j]>0) {
            for (q=0;q<(int)cell->size();q++)
            if (neighbours[sigma[i+1][j]][q]==EMPTY) {
              neighbours[sigma[i+1][j]][q]=sigma[i][j];
              break;
            }
            else
              if (neighbours[sigma[i+1][j]][q]==sigma[i][j])
                break;
          }
        }
      }
      else
        if (g && sigma[i][j]>0)
          g->Point( colour, 2*i+1, 2*j );


      if ( sigma[i][j] != sigma[i][j+1] ) {

        if (g)
        g->Point( 1, 2*i, 2*j+1 );

      if (get_neighbours) {
        if (sigma[i][j]>0) {
          for (q=0;q<(int)cell->size();q++)
            if (neighbours[sigma[i][j]][q]==EMPTY) {
            neighbours[sigma[i][j]][q]=sigma[i][j+1];
            break;
            }
            else
            if (neighbours[sigma[i][j]][q]==sigma[i][j+1])
              break;
        }

        if (sigma[i][j+1]>0) {

          for (q=0;q<(int)cell->size();q++)
            if (neighbours[sigma[i][j+1]][q]==EMPTY) {
            neighbours[sigma[i][j+1]][q]=sigma[i][j];
            break;
            }
            else
            if (neighbours[sigma[i][j+1]][q]==sigma[i][j])
              break;
        }
      }
      }
      else
        if (g && sigma[i][j]>0)
          g->Point( colour, 2*i, 2*j+1 );

      /* Cells that touch eachother's corners are NO
neighbours */

      if (sigma[i][j]!=sigma[i+1][j+1]
        || sigma[i+1][j]!=sigma[i][j+1] ) {
        if (g)
```

```
          g->Point( 1, 2*i+1, 2*j+1 );
      }
      else
        if (g && sigma[i][j]>0)
          g->Point( colour, 2*i+1, 2*j+1 );
    }

  if (get_neighbours)
    return neighbours;
  else
    return 0;

}


void CellularPotts::ReadZygotePicture(void) {



  int pix,cells,i,j,c,p,checkx,checky;
  char **pixelmap;
  char pixel[3];

  sscanf(ZYGXPM(ZYGOTE)[0],"%d %d %d %d",&checkx,&checky,
&cells,&pix);

  if ((checkx>sizex)||(checky>sizey)) {
    std::cerr <<  "ReadZygote: The included xpm picture is
smaller than the grid!\n";
    std::cerr << "\n Please adjust either the grid size or the
picture size.\n";
    std::cerr << sizex << "," << sizey << "," << checkx << ","
<< checky << "\n";
    exit(1);
  }

  pixelmap=(char **)malloc(cells*sizeof(char *));
  if (pixelmap==NULL) MemoryWarning();

  pixelmap[0]=(char *)malloc(cells*3*sizeof(char));
  if (pixelmap[0]==NULL) MemoryWarning();

  for(i=1;i<cells;i++)
    pixelmap[i]=pixelmap[i-1]+3;

  for (i=0;i<cells;i++) {
    for (j=0;j<pix;j++)
      pixelmap[i][j]=ZYGXPM(ZYGOTE)[i+1][j];
    pixelmap[i][pix]='\0';
  }

  for (i=0;i<sizex*sizey;i++) sigma[0][i]=0;
  fprintf(stderr,"[%d %d]\n",checkx,checky);

  int offs_x, offs_y;
  offs_x=(sizex-checkx)/2;
  offs_y=(sizey-checky)/2;
```

```
  for (i=0;i<checkx;i++)
    for (j=0;j<checky;j++) {
      for (p=0;p<pix;p++)
        pixel[p]=ZYGXPM(ZYGOTE)[cells+1+j][i*pix+p];

      pixel[pix]='\0';

      for (c=0;c<cells;c++) {
      if (!(strcmp(pixelmap[c],pixel))) {
        if ( (sigma[offs_x+i][offs_y+j]=c) ) {

          // if c is _NOT_ medium (then c=0)
          // assign pixel values from "sigmamax"
          sigma[offs_x+i][offs_y+j]+=(Cell::MaxSigma()-1);
        }
      }

      }
    }

  free(pixelmap[0]);
  free(pixelmap);
}


void CellularPotts::ConstructInitCells (Dish &beast) {

  // Get the maximum cell ID (mostly equal to the cell number)
  int loop=sizex*sizey;
  int cells=0;
  for (int i=0;i<loop;i++) {
    if (cells<sigma[0][i]) cells=sigma[0][i];
  }

  cerr << "[ cells = " << cells << "]\n";

  // construct enough cells for the zygote.  "cells", contains
the
  // number of colours (excluding background).
  {
    for (int i=0; i<cells; i++) {
      cell->push_back(Cell(beast));
    }
  }

  // Set the area and target area of the cell
  // makes use of the pointer to the Cell pointer of Dish
  // which is a member of CellularPotts
  MeasureCellSizes();

  // set zygote_area to mean cell area.
  int mean_area=0;
  for (vector<Cell>::iterator c=cell->begin();c!=cell->
end();c++) {
    mean_area+=c->Area();
  }
```

```
  if (cells!=0)
    mean_area/=cells;

  zygote_area=mean_area;

  cout << "mean_area = " << mean_area << "\n";
  // set all cell areas to the mean area
  {
    for (vector<Cell>::iterator c=cell->begin();c!=cell->
end();c++) {
      if (par.target_area) {
     c->SetTargetArea(par.target_area);
      } else     {
     c->SetTargetArea(mean_area);
      }
    }
  }
}

void CellularPotts::MeasureCellSizes(void) {

  // Clean areas of all cells, including medium
  for (vector<Cell>::iterator c=cell->begin();c!=cell->
end();c++) {
    c->SetTargetArea(0);
    c->area = 0;
  }

  // calculate the area of the cells
  for (int x=1;x<sizex-1;x++) {
    for (int y=1;y<sizey-1;y++) {
      if (sigma[x][y]) {
     (*cell)[sigma[x][y]].IncrementTargetArea();
     (*cell)[sigma[x][y]].IncrementArea();
     (*cell)[sigma[x][y]].AddSiteToMoments(x,y);

      }
    }
  }

  // set the actual area to the target area
  {
  for (vector<Cell>::iterator c=cell->begin();c!=cell->
end();c++) {
    c->SetAreaToTarget();

  }
  }
}

void CellularPotts::MeasureCellSize(Cell &c) {

  c.CleanMoments();

  // calculate the area of the cell
  for (int x=1;x<sizex-1;x++) {
    for (int y=1;y<sizey-1;y++) {
```

```
      if (sigma[x][y] == c.sigma) {
      (*cell)[sigma[x][y]].IncrementTargetArea();
      (*cell)[sigma[x][y]].IncrementArea();
      (*cell)[sigma[x][y]].AddSiteToMoments(x,y);

      }
    }
  }

//   // set the actual area to the target area
//   {
//   for (vector<Cell>::iterator c=cell->begin();c!=cell->
end();c++) {
//     c->SetAreaToTarget();

//   }

}

Dir *CellularPotts::FindCellDirections(void) const
{

  double *sumx=0,*sumy=0;
  double *sumxx=0,*sumxy=0,*sumyy=0;
  double *n=0;

  double xmean=0,ymean=0,sxx=0,sxy=0,syy=0;
  double D,lb1=0,lb2=0;

  Dir *celldir;

  /* Allocation of sufficient memory space */
  if( (sumx= (double *)malloc(cell->
size()*sizeof(double)))==NULL)
    MemoryWarning();
  else
    if( (sumy= (double *)malloc(cell->
size()*sizeof(double)))==NULL)
      MemoryWarning();
    else
      if ((sumxx=(double *)malloc(cell->
size()*sizeof(double)))==NULL)
     MemoryWarning();
      else
     if((sumxy=(double *)malloc(cell->
size()*sizeof(double)))==NULL)
       MemoryWarning();
     else
       if((sumyy=(double *)malloc(cell->
size()*sizeof(double)))==NULL)
         MemoryWarning();
       else
         if((n=(double *)malloc(cell->
size()*sizeof(double)))==NULL)
           MemoryWarning();
```

```
  if ( !(celldir=new Dir[cell->size()]) )
    MemoryWarning();


  /* Initialization of the variables */

  for (int i=0;i<(int)cell->size();i++) {

    sumx[i]=0.;
    sumy[i]=0.;
    sumxx[i]=0.;
    sumxy[i]=0.;
    sumyy[i]=0.;
    n[i]=0L;

  }


  /* Find sumx, sumy, sumxx and sumxy for all cells */

  for (int x=0;x<sizex;x++)
    for (int y=0;y<sizey;y++)
      if (sigma[x][y]>0) {
      sumx[0]+=(double)x;
      sumy[0]+=(double)y;
      sumxx[0]+=(double)x*x;
      sumxy[0]+=(double)x*y;
      sumyy[0]+=(double)y*y;

      n[0]++;

      sumx[sigma[x][y]]+=(double)x;
      sumy[sigma[x][y]]+=(double)y;

      sumxx[sigma[x][y]]+=(double)x*x;
      sumxy[sigma[x][y]]+=(double)x*y;
      sumyy[sigma[x][y]]+=(double)y*y;

      n[sigma[x][y]]++;

       }

  /* Compute the principal axes for all cells */

  {
    for (int i=0;i<(int)cell->size();i++) {

       if (n[i]>10) {

      xmean=((double)sumx[i])/((double)n[i]);
      ymean=((double)sumy[i])/((double)n[i]);

      sxx=(double)(sumxx[i])-((double)
(sumx[i]*sumx[i]))/(double)n[i];
      sxx=sxx/(double)(n[i]-1);

      sxy=(double)(sumxy[i])-((double)
```

```
(sumx[i]*sumy[i]))/(double)n[i];
      sxy=sxy/(double)(n[i]-1);

      syy=(double)(sumyy[i])-((double)
(sumy[i]*sumy[i]))/(double)n[i];
      syy=syy/(double)(n[i]-1);

      D=sqrt( (sxx+syy)*(sxx+syy)-4.*(sxx*syy-sxy*sxy) );
      lb1=(sxx+syy+D)/2.;lb2=(sxx+syy-D)/2.;
      celldir[i].lb1=lb1; celldir[i].lb2=lb2;
       }
       if (sxy==0.0)
      celldir[i].bb1=1.;
       else
      celldir[i].bb1=sxy/(lb1-syy);

       if (fabs(celldir[i].bb1)<.00001) {
      if (celldir[i].bb1>0.)
        celldir[i].bb1=.00001;
      else
        celldir[i].bb1=-.00001;
       }

       celldir[i].aa1=ymean-xmean*celldir[i].bb1;
       celldir[i].bb2= (-1.)/celldir[i].bb1;

       celldir[i].aa2=ymean-celldir[i].bb2*xmean;
    }

  }

  /* bevrijd gealloceerd geheugen */
  free(sumx);
  free(sumy);
  free(sumxx);
  free(sumxy);
  free(sumyy);
  free(n);

  return celldir;

}

void CellularPotts::ShowDirections(Graphics &g, const Dir
*celldir) const
{
  int i;

  if (cell->size()>1)
    for (i=1;i<(int)cell->size();i++)
      g.Line(0,(int)(2*celldir[i].aa1),sizex*2,(int)
((celldir[i].aa1+celldir[i].bb1*sizey)*2),2);

}

void CellularPotts::DivideCells(vector<bool> which_cells)
{
```

```
  // for the cell directions
  Dir *celldir=0;

  /* Allocate space for divisionflags */
  int *divflags=(int *)malloc((cell->size()*2+5)*sizeof(int));

  /* Clear divisionflags */
  for (int i=0;i<(int)(cell->size()*2+5);i++)
    divflags[i]=0;


  if ( !(which_cells.size()==0 || which_cells.size()>=cell->
size()) ) {
    throw "In CellularPotts::DivideCells, Too few elements in
vector<int> which_cells.";
  }

  /* division */
  {for (int i=0;i<sizex;i++)
      for (int j=0;j<sizey;j++)
     if (sigma[i][j]>0) // i.e. not medium and not border
state (-1)
       {


          // Pointer to mother. Warning: Renew pointer after a
new
          // cell is added (push_back). Then, the array *cell
is relocated and
          // the pointer will be lost...

          Cell *motherp=&((*cell)[sigma[i][j]]);
          Cell *daughterp;

          /* Divide if NOT medium and if DIV bit set or
divide_always is set */
          // if which_cells is given, divide only if the cell
          // is marked in which_cells.
          if  ( !which_cells.size() || which_cells[motherp->
sigma] )    {

            if (!(divflags[ motherp->Sigma() ]) ) {

            // add daughter cell, copying states of mother
            daughterp=new Cell(*(motherp->owner));
            daughterp->CellBirth(*motherp);
            cell->push_back(*daughterp);

            // renew pointer to mother
            motherp=&((*cell)[sigma[i][j]]);

            divflags[ motherp->Sigma() ]=daughterp->Sigma();
            delete daughterp;

            // array may be relocated after "push_back"
```

```
            // renew daughter pointers
            daughterp=&(cell->back());

            /* administration on the onset of mitosis */

            /* Ancestry is taken care of in copy constructor of
Cell
               see cell.hh: Cell(const Cell &src, bool
newcellP=false) : Cytoplasm(src) {} */

            /* inherit  polarity of mother */
            // All that needs to be copied is copied in the copy
constructor
            // of Cell and in the default copy constr. of its
base class Cytoplasm
            // note: also the celltype is inherited



             } else {
            daughterp=&((*cell)[ divflags[motherp->Sigma()] ]);
             }


             /* Now the actual division takes place */

             /* If celldirections where not yet computed: do it
now */
             if (!celldir)
            celldir=FindCellDirections();

             /* if site is below the minor axis of the cell:
sigma of new cell */
             if (j>((int)(celldir[motherp->sigma].aa2+
                     celldir[motherp->sigma].bb2*(double)i))) {

            motherp->DecrementArea();
            motherp->DecrementTargetArea();
            motherp->RemoveSiteFromMoments(i,j);
            sigma[i][j]=daughterp->Sigma();
            daughterp->AddSiteToMoments(i,j);
            daughterp->IncrementArea();
            daughterp->IncrementTargetArea();

             }


          }

        }
  }
  if (celldir)
    delete[] (celldir);

  if (divflags)
    free(divflags);
}
```

```
/**! Fill the plane with initial cells
 \return actual amount of cells (some are not draw due to
overlap) */
int CellularPotts::ThrowInCells(int n,int cellsize) {

  //  int gapx=(sizex-nx*cellsize)/(nx+1);
  //int gapy=(sizey-ny*cellsize)/(ny+1);

  int cellnum=1;

  for (int i=0;i<n;i++) {

    // draw a circle at x0, y0
    int x0=RandomNumber(sizex);
    int y0=RandomNumber(sizey);

    bool overlap=false;

    // check overlap
    for (int x=0;x<cellsize;x++)
      for (int y=0;y<cellsize;y++)
      if ( (
            ( (x-cellsize/2)*(x-cellsize/2)+(y-cellsize/2)*(y-
cellsize/2) )<
            ( (cellsize/2)*(cellsize/2))) &&
          ( x0+x<sizex && y0+y<sizey ) )
        if (sigma[x0+x][y0+y]) {
          overlap=true;
          break;
        }

    if (!overlap) {
      for (int x=0;x<cellsize;x++)
      for (int y=0;y<cellsize;y++)
        if ( (
            ( (x-cellsize/2)*(x-cellsize/2)+(y-cellsize/2)*(y-
cellsize/2) )<
            ( (cellsize/2)*(cellsize/2))) &&
             ( x0+x<sizex && y0+y<sizey ) )
          sigma[x0+x][y0+y]=cellnum;

      cellnum++;
    }
  }
  cerr << "[ cellnum = " << cellnum << "]";

  // repair borders
  // fill borders with special border state
  for (int x=0;x<sizex-1;x++) {
    sigma[x][0]=-1;
    sigma[x][sizey-1]=-1;
  }
  for (int y=0;y<sizey-1;y++) {
    sigma[0][y]=-1;
    sigma[sizex-1][y]=-1;
```

```
  }

  {for (int x=1;x<sizex-2;x++) {
      sigma[x][1]=0;
      sigma[x][sizey-2]=0;
    }}
  {for (int y=1;y<sizey-2;y++) {
      sigma[1][y]=0;
      sigma[sizex-2][y]=0;
    }}
  return cellnum;
}


int CellularPotts::GrowInCells(int n_cells, int cell_size,
double subfield) {


  int sx = (int)((sizex-2)/subfield);
  int sy = (int)((sizey-2)/subfield);

  int offset_x = (sizex-2-sx)/2;
  int offset_y = (sizey-2-sy)/2;

  if (n_cells==1) {
    return GrowInCells(1, cell_size, sizex/2, sizey/2, 0, 0);
  } else {
    return GrowInCells(n_cells, cell_size, sx, sy, offset_x,
offset_y);
  }
}

int CellularPotts::GrowInCells(int n_cells, int cell_size, int
sx, int sy, int offset_x, int offset_y) {

  // make initial cells using Eden Growth

  int **new_sigma=(int **)malloc(sizex*sizeof(int *));
  if (new_sigma==NULL)
    MemoryWarning();

  new_sigma[0]=(int *)malloc(sizex*sizey*sizeof(int));
  if (new_sigma[0]==NULL)
    MemoryWarning();

  for (int i=1;i<sizex;i++)
    new_sigma[i]=new_sigma[i-1]+sizey;

  /* Clear CA plane */
  { for (int i=0;i<sizex*sizey;i++)
     new_sigma[0][i]=0;
  }


  // scatter initial points, or place a cell in the middle
  // if only one cell is desired
  int cellnum=cell->size()-1;
```

```
  if (n_cells>1) {



    { for (int i=0;i<n_cells;i++) {

      sigma[RandomNumber(sx)+offset_x]
[RandomNumber(sy)+offset_y]=++cellnum;

    }}
  } else {
    sigma[sx][sy]=++cellnum;

  }

  // Do Eden growth for a number of time steps
  {for (int i=0;i<cell_size;i++) {
    for (int x=1;x<sizex-1;x++)
      for (int y=1;y<sizey-1;y++) {

     if (sigma[x][y]==0) {
       // take a random neighbour
       int xyp=(int)(8*RANDOM()+1);
       int xp = nx[xyp]+x;
       int yp = ny[xyp]+y;
       int kp;
       //  NB removing this border test yields interesting
effects :-)
       // You get a ragged border, which you may like!
       if ((kp=sigma[xp][yp])!=-1)
         if (kp>(cellnum-n_cells))
           new_sigma[x][y]=kp;
         else
           new_sigma[x][y]=0;
       else
         new_sigma[x][y]=0;

     } else {
       new_sigma[x][y]=sigma[x][y];
     }
      }

    // copy sigma to new_sigma, but do not touch the border!
       {  for (int x=1;x<sizex-1;x++) {
      for (int y=1;y<sizey-1;y++) {
     sigma[x][y]=new_sigma[x][y];
      }
    }
  }}}
  free(new_sigma[0]);
  free(new_sigma);

  return cellnum;
}
```

```
// Predicate returns true when connectivity is locally
preserved
// if the value of the central site would be changed
bool CellularPotts::ConnectivityPreservedP(int x, int y) {

  // Use local nx and ny in a cyclic order (starts at upper
left corner)
  // first site is repeated, for easier looping
  const int cyc_nx[10] = {-1, -1, 0, 1, 1, 1, 0, -1, -1, -1 };
  const int cyc_ny[10] = {0, -1,-1,-1, 0, 1, 1,  1,  0, -1 };

  int sxy=sigma[x][y]; // the central site
  if (sxy==0) return true;

  int n_borders=0; // to count the amount of sites in state
sxy bordering a site !=sxy

  static int stack[8]; // stack to count number of different
surrounding cells
  int stackp=-1;
  bool one_of_neighbors_medium=false;

  for (int i=1;i<=8;i++) {

    int s_nb=sigma[x+cyc_nx[i]][y+cyc_ny[i]];
    int s_next_nb=sigma[x+cyc_nx[i+1]][y+cyc_ny[i+1]];

    if ((s_nb==sxy || s_next_nb==sxy) && (s_nb!=s_next_nb)) {

      // check whether s_nb is adjacent to non-identical site,
      // count it
      n_borders++;
    }
    int j;
    bool on_stack_p=false;

    // we need the next heuristic to prevent stalling at
    // cell-cell borders
    // do not enforce constraint at two cell interface(no
medium)
    if (s_nb) {
      for (j=stackp;j>=0;j--) {
      if (s_nb==stack[j]) {
        on_stack_p=true;
        break;
      }
      }
      if (!on_stack_p) {
      if (stackp>6) {
        cerr << "Stack overflow, stackp=" << stackp << "\n";
      }
      stack[++stackp]=s_nb;
      }
    } else {
      one_of_neighbors_medium=true;
    }
  }
```

```
  // number of different neighbours is stackp+1;
  if (n_borders>2 && ( (stackp+1)>2 ||
one_of_neighbors_medium) ) {
    return false;
  }
  else
    return true;

}


double CellularPotts::CellDensity(void) const {

  // return the density of cells
  int sum=0;
  for (int i=0;i<sizex*sizey;i++) {
    if (sigma[0][i]) {
      sum++;
    }
  }
  return (double)sum/(double)(sizex*sizey);

}

double CellularPotts::MeanCellArea(void) const {

  int sum_area=0, n=0;
  double sum_length=0.;
  vector<Cell>::iterator c=cell->begin(); ++c;

  for (;
      c!=cell->end();
      c++) {

    sum_area+=c->Area();
    sum_length+=c->Length();
    n++;
  }

  cerr << "Mean cell length is " << sum_length/((double)n)
<< endl;
  return (double)sum_area/(double)n;
}

void CellularPotts::ResetTargetLengths(void)  {
   vector<Cell>::iterator c=cell->begin(); ++c;

   for (;
        c!=cell->end();
        c++) {

     c->SetTargetLength(par.target_length);

}

}
```

```
void CellularPotts::SetRandomTypes(void) {

  // each cell gets a random type 1..maxtau

  vector<Cell>::iterator c=cell->begin(); ++c;

  for (;
       c!=cell->end();
       c++) {

    int celltype = RandomNumber(Cell::maxtau);
    c->setTau(celltype);

  }

}

void CellularPotts::GrowAndDivideCells(int growth_rate) {

  vector<Cell>::iterator c=cell->begin(); ++c;
  vector<bool> which_cells(cell->size());


  for (;
       c!=cell->end();
       c++) {
    double c_used = 0;
    c_used = c->chem[2];
     if (RANDOM()<c_used){
      c->SetTargetArea(c->TargetArea()+growth_rate);
     } else ;
      if (c->Area()>par.target_area) {
         which_cells[c->Sigma()]=true;
      } else {
         which_cells[c->Sigma()]=false;
       }
     }

  DivideCells(which_cells);

}

double CellularPotts::DrawConvexHull(Graphics *g, int color) {

  // Draw the convex hull of the cells
  // using Andrew's Monotone Chain Algorithm (see hull.cpp)

  // Step 1. Prepare data for 2D hull code

  // count number of points to determine size of array
  int np=0;
  for (int x=1;x<sizex-1;x++)
    for (int y=1;y<sizey-1;y++) {
      if (sigma[x][y]) {
     np++;
      }
```

```
    }

  Point *p=new Point[np];

  int pc=0;
  for (int x=1;x<sizex-1;x++)
    for (int y=1;y<sizey-1;y++) {
      if (sigma[x][y]) {
     p[pc++]=Point(x,y);
      }
    }

  // Step 2: call 2D Hull code
  Point *hull=new Point[np];
  int nph=chainHull_2D(p,np,hull);

  // Step 3: draw it
  for (int i=0;i<nph-1;i++) {
    g->Line(2*hull[i].x,2*hull[i].y,2*hull[i+1].x,2*hull[i+
1].y, color);
  }


  // Step 4: calculate area of convex hull
  double hull_area=0.;
  for (int i=0;i<nph-1;i++) {
    hull_area+=hull[i].x*hull[i+1].y-hull[i+1].x*hull[i].y;
  }
  hull_area/=2.;

  //cerr << "Area = " << hull_area << "\n";

  delete[] p;
  delete[] hull;

  return hull_area;

}
int CellularPotts::Circumference(void) {
// Calculate the circumference by counting all the pixels at
the border of cells
    int circum = 0;
    for (int x=1;x<sizex-1;x++)
        for (int y=1;y<sizey-1;y++) {

            if (Sigma(x,y)) {
                if (!(Sigma(x+1,y))) {
                circum++;
                } else if (!(Sigma(x-1,y))) {
                circum++;
                } else if (!(Sigma(x,y+1))) {
                circum++;
                } else if (!(Sigma(x,y-1))) {
                circum++;
                } else if (!(Sigma(x+1,y+1))) {
                circum++;
                } else if (!(Sigma(x-1,y-1))) {
```

```
                circum++;
                } else if (!(Sigma(x-1,y+1))) {
                circum++;
                } else if (!(Sigma(x+1,y-1))) {
                circum++;
                } else {}
             }
            }

   return circum;
   }

double CellularPotts::Compactness(double *res_compactness,
double *res_area, double *res_cell_area) {

  // Calculate compactness using the convex hull of the cells
  // We use Andrew's Monotone Chain Algorithm (see hull.cpp)

  // Step 1. Prepare data for 2D hull code

  // count number of points to determine size of array
  int np=0;
  for (int x=1;x<sizex-1;x++)
    for (int y=1;y<sizey-1;y++) {
      if (sigma[x][y]) {
     np++;
      }
    }

  Point *p=new Point[np];

  int pc=0;
  for (int x=1;x<sizex-1;x++)
    for (int y=1;y<sizey-1;y++) {
      if (sigma[x][y]) {
     p[pc++]=Point(x,y);
      }
    }

  // Step 2: call 2D Hull code
  Point *hull=new Point[np];
  int nph=chainHull_2D(p,np,hull);

  //// Step 3: draw it
  //for (int i=0;i<nph-1;i++) {
  //  g->Line(2*hull[i].x,2*hull[i].y,2*hull[i+1].x,2*hull[i+
1].y, color);
  //}


  // Step 3: calculate area of convex hull
  double hull_area=0.;
  for (int i=0;i<nph-1;i++) {
    hull_area+=hull[i].x*hull[i+1].y-hull[i+1].x*hull[i].y;
  }
  hull_area/=2.;
```

```
  // Step 4: calculate total cell area
  double cell_area=0;

  vector<Cell>::const_iterator c;

  for ( (c=cell->begin(),c++);
       c!=cell->end();
       c++) {
    cell_area+=c->Area();
  }

  delete[] p;
  delete[] hull;


  // put intermediate results into optional pointers
  if (res_compactness) {
    *res_compactness = cell_area/hull_area;
  }
  if (res_area) {
    *res_area = hull_area;
  }
  if (res_cell_area) {
    *res_cell_area = cell_area;
  }

  // return compactness

  return cell_area/hull_area;

}
```