

# Documentation for `cellmove.c`

by John C. Dallon

Brigham Young University

Provo, UT 84602

July 14, 2006

The following is documentation for the code `cellmove.c`. The code simulates the motion of an aggregate of cells in three dimensions. The forces on each cell include the viscous drag forces due to other cells, viscous force due to the substrate and fluid, active forces generated by a cell moving and passive forces due to the visco-elastic properties of a cell. These forces are calculated and the cells, assumed to be ellipsoids, move accordingly. For more details see Palsson and Othmer, "A Model for Individual and Collective Cell movement in *Dictyostelium discoideum*", PNAS (2000) 97, pg 10448-10453 and Dallon and Othmer "How Cellular movement determines the collective force generated by the *Dictyostelium discoideum* slug" in Journal of Theoretical Biology (2006) 231, pg 203-222. The code was written by Eirikur Palsson in 1999 and was last modified by John Dallon in 2006.

The code is a mixture of C and FORTRAN routines which allows cells to move in three dimensions. The cells are assumed to be ellipsoidal in shape, although for some calculations the shape is assumed to be a rectangular box. Each cell is modeled as a Maxwell element<sup>1</sup> in parallel with a nonlinear spring (which does not easily compress). They adhere to one another and to the substrate. In order to move, the cells require a motion stimulus and an orientation stimulus.

To identify neighboring cells more efficiently there is a cell location matrix (CLM) that locates cells in a coarse spatial grid. Searches for neighboring cells can be restricted to cells in nearby coarse grids squares.

The code requires the Fortran routine `dlode`, the Fortran routine `dgesv` (a Lapack routine), and the C++ libraries for SparseLib++ (Roldan Pozo, Karin Remington, and Andrew Lumsdaine) (SparseLib++ requires MV++ (Rolando Pozo)), and the C++ function in IML++ (Jack Dongarra, Andrew Lumsdaine, Roldan Pozo and, Karin Remington) called `gmres`.

## List of Algorithms

1	Structure of <code>Main</code> . . . . .	3
2	Structure of <code>initialization</code> . . . . .	4

---

<sup>1</sup>see Biomechanics Mechanical Properties of Living Tissues by Y.C. Fung

3	Structure of <code>move_cells</code> . . . . .	6
4	Structure of <code>move_cellsrk</code> . . . . .	9
5	Structure of <code>runge</code> . . . . .	11
6	Structure of <code>clm_update_all</code> . . . . .	12
7	Structure of <code>orient_once</code> . . . . .	13
8	Structure of <code>find_neighbour</code> . . . . .	14
9	Structure of <code>orient</code> . . . . .	16
10	structure of <code>all_force</code> . . . . .	18
11	Structure of <code>surface</code> . . . . .	20
12	Structure of <code>surfaceb</code> . . . . .	20
13	Structure of <code>aforce</code> . . . . .	21
14	Structure of <code>bforce</code> . . . . .	22
15	Structure of <code>active_force</code> . . . . .	23
16	Structure of <code>active_bforce</code> . . . . .	23
17	Structure of <code>bound_force</code> . . . . .	24
18	Structure of <code>compress</code> . . . . .	25
19	Structure of <code>move1cell</code> . . . . .	27
20	Structure of <code>find_stretchingfluidode</code> . . . . .	28
21	Structure of <code>fcell</code> . . . . .	28
22	Structure of <code>find_stretchingforce</code> . . . . .	29
23	Structure of <code>write_to_file</code> . . . . .	29

## 1 Main

This is the main C function which runs the code.

---

**Algorithm 1** Structure of Main

---

**initialization** (see section 2).  
**orient\_once** (see section 6.2).  
Memory is allocated for three arrays: **a\_genr**, **ia\_genr**, and **b**.  
**for**  $m = 1$  to  $nsteps$  **do** {the main time loop}  
  **if**  $m \bmod mout = 0$  **then**  
    Freeze a random subset of cells.  
  **end if**  
  **for**  $i = 1$  to  $ncells$  **do** {loop through the cells}  
    **determine\_motion\_stimulus** (see section 6.16).  
    Set  $gamax$  (a cell variable used as the motion stimulus) to be  $camp$ .  
    **orient** (see section 6.6).  
    Print warning statements if the cell sizes are extreme.  
    Set the neighbor distance to be the largest radius of the cell.  
  **end for**  
  **for**  $i = 1$  to  $ncells$  **do** {loop through the cells}  
    **find\_neighbour** (see section 6.4).  
  **end for**  
  **if**  $m \leq 3$  **then**  
    **move\_cellsrk** (see section 4). Use a Runge Kutta scheme to start up.  
  **else**  
    **move\_cells** (see section 3). Use Adams Bashforth and Moulton 4 order predictor corrector.  
  **end if**  
  **for**  $i = 1$  to  $ncells$  **do** {loop through the cells}  
    Update the cell locations.  
  **end for**  
  **if**  $(m - 1) \bmod j = 0$  **then**  
    **write\_to\_file** (see section 6.11). Write output for cell locations and shape.  
    **write\_to\_file\_wave** (see section 6.13). Write output for the motion indicator.  
  **end if**  
  Write some special outputs.  
  **clm\_update\_all** (see section 6.1). Update the cell location matrix.  
  Advance the time.  
  **if**  $m \bmod mout$  **then**  
    Write the time to standard output.  
  **end if**  
**end for**  
Free memory and close files.

---

## 2 initialization

This is a C function which initializes various parameters and variables for the C code.

---

**Algorithm 2** Structure of initialization

---

```
srand48(55). Set the seed for the random number generator.  
read_ifile (see section 6.12). Read data from the file ifile.dat.  
Set and nondimensionalize parameters.  
Remove existing files which will be used to store data for the new run.  
Open files for storing data.  
Set and nondimensionalize more parameters.  
for  $m = 1$  to  $ncells$  do {Loop through the cells to set initial cell values}  
    Set various individual cell parameters.  
end for  
for  $mm = 1$  to  $iz$  do {Loop through the  $z$  direction}  
    for  $n = 1$  to  $ix$  do {Loop through the  $x$  direction}  
        for  $kk = 1$  to  $iy$  do {Loop through the  $y$  direction}  
            Set the cell location.  
        end for  
    end for  
end for  
clm_update_all. Set the cell location matrix.
```

---

### 3 move\_cells

This C function moves the cells. It uses an Adams Bashforth and Moulton 4 order predictor corrector to solve  $\mathbf{M}\dot{\mathbf{x}} = \mathbf{b}$ .

---

**Algorithm 3** Structure of `move_cells`

---

**Begin by predicting with an Adams Bashforth 4th order method.**

Set the vectors `b`, `a_genr` (the vector used to store the values of the matrix `M`), and `ia_genr` the vector which identifies the position of the value in the matrix. `M` is a sparse matrix and thus stored in a packed manner in these two arrays.

**for**  $i = 1$  to  $ncells$  **do** {Loop through the cells}

Determine if the cell is in contact with the substrate. The substrate is set at the  $z = 1$  plane.

Set the change in the radii of the cell to be zero.

Determine if the cell is in contact with a top plate.

`compress` (see section 6.8). Determine how much the cell axes should be compress due to the physical presences of other cells and the turgidity of the cells.

`nr_contact` = 0, initializing the number of cells in contact with the current cell.

**end for**

**for**  $i = 1$  to  $ncells$  **do** {Loop through the cells}

`all_force` (see section 6.7). Determine all the forces acting on a cell.

**end for**

Set `totalnonzero`, the total number of nonzero elements in the sparse matrix `M` to be 0.

**for**  $i = 1$  to  $ncells$  **do** {Loop through the cells}

`move1cell` (see section 6.9). Set the force vector `b`.

Update `totalnonzero`.

`find_stretchingfluidode` (see section 6.9.1). Determine how the cell changes shape.

**end for**

*Solve the system  $\mathbf{M} \cdot \mathbf{x} = \mathbf{b}$  using a sparse iterative solver from IML++.*

Initialize some parameters.

Allocate memory.

Pack the arrays in the method necessary for the solver.

`CompRow_Mat_double` (see SparseLib++). Create the sparse matrix.

`VECTOR_double`. Create the initial guess for the vector.

`MATRIX_double`. Create a work matrix.

`DiagPreconditioner_double`. Create the diagonal preconditioner.

`GMRES` (see IML++). Solve the system with iterative solver.

---

---

Structure of `move_cells` cont.

Print warnings if something goes wrong.

Store the solution in `b`.

**for**  $i = 1$  to  $ncells$  **do** {Loop through the cells}

Store the new solution in `bvold`( $\cdot$ , 3).

$$\frac{\mathbf{x}(\cdot) - \mathbf{xold}(\cdot)}{dt} = \frac{1}{24} (55\mathbf{bvold}(\cdot, 3) - 59\mathbf{bvold}(\cdot, 2) + 37\mathbf{bvold}(\cdot, 1) - 9\mathbf{bvold}(\cdot, 0))$$

(Adams Bashforth 4th order).

Store several variables for use in the next iteration.

**end for**

Free memory.

**Do the correction using an Adams Moulton 4th order method.**

Set the vectors `b`, `a_genr` (the vector used to store the values of the matrix `M`), and `ia_genr` the vector which identifies the position of the value in the matrix. `M` is a sparse matrix and thus stored in a packed manner in these two arrays.

**for**  $i = 1$  to  $ncells$  **do** {Loop through the cells}

Determine if the cell is in contact with the substrate. The substrate is set at the  $z = 1$  plane.

Set the change in the radii of the cell to be zero.

Determine if the cell is in contact with a top plate.

`compress` (see section 6.8). Determine how much the cell axes should be compress due to the physical presences of other cells and the turgidity of the cells.

`nr_contact` = 0, initialize the number of cells in contact with the current cell.

**end for**

**for**  $i = 1$  to  $ncells$  **do** {Loop through the cells}

`all_force` (see section 6.7). Determine all the forces acting on a cell.

**end for**

Set `totalnonzero`, the total number of nonzero elements in the sparse matrix `M` to be 0.

**for**  $i = 1$  to  $ncells$  **do** {Loop through the cells}

`move1cell` (see section 6.9). Set the force vector `b`.

Update `totalnonzero`.

`find_stretchingfluidode` (see section 6.9.1). Determine how the cell changes shape.

**end for**

---

---

Structure of `move_cells` cont.

*Solve the system  $\mathbf{M}\cdot\mathbf{x} = \mathbf{b}$  using a sparse iterative solver from a package `IML++`.*

Initialize some parameters.

Allocate memory.

Pack the arrays in the method necessary for the solver.

`CompRow_Mat_double` (see `SparseLib++`). Create the sparse matrix.

`VECTOR_double`. Create the initial guess for the vector.

`MATRIX_double`. Create a work matrix.

`DiagPreconditioner_double`. Create the diagonal preconditioner.

`GMRES` (see `IML++`). Solve the system with iterative solver.

Print warnings if something goes wrong.

Store the solution in `b`.

**for**  $i = 1$  to  $ncells$  **do** {Loop through the cells}

$$\frac{\mathbf{x}(\cdot) - \mathbf{xold}(\cdot)}{dt} = \frac{1}{24} (9\mathbf{b}(\cdot) + 19\mathbf{bvold}(\cdot, 3) - 5\mathbf{bvold}(\cdot, 2) + \mathbf{bvold}(\cdot, 1))$$

(Adams Bashforth 4th order).

**end for**

Free memory.

---

## 4 move\_cellsrk

This C function moves the cells for the first three iterations using a Runge Kutta method until there is enough starting data for the predictor corrector.

---

**Algorithm 4** Structure of `move_cellsrk`

---

Set the time for the calculation of **K1**.

`runge` (see section 5). Find **K1**.

Store **b** in **K1**.

$\mathbf{x} = \mathbf{xold} + .5dt\mathbf{K1}$

Store several variables for recovery later.

Update the time step for the intermediate Runge Kutta steps.

`runge` (see section 5). Find **K2**.

Reset the time step for the next calculation.

Store **b** in **K2**.

$\mathbf{x} = \mathbf{xold} + .5dt\mathbf{K2}$

Update the time for the intermediate Runge Kutta steps.

`runge` (see section 5). Find **K3**.

Reset the time step for the next calculation.

Store **b** in **K3**.

$\mathbf{x} = \mathbf{xold} + .5dt\mathbf{K3}$

Update the time for the intermediate Runge Kutta steps.

`runge` (see section 5). Find **K4**.

Reset the time step for the next calculation.

Store **b** in **K4**.

$$\mathbf{b} = \frac{1}{6} (\mathbf{K1} + 2\mathbf{K2} + 2\mathbf{K3} + \mathbf{K4})$$

$d\mathbf{x} = dt\mathbf{b}$

Store some variables for later use.

---

## 5 `runge`

This C function defines the **K** vectors used in the Runge Kutta method.

---

**Algorithm 5** Structure of `runge`

---

Set the vectors `b`, `a_genr` (the vector used to store the values of the matrix `M`), and `ia_genr` the vector which identifies the position of the value in the matrix. `M` is a sparse matrix and thus stored in a packed manner in these two arrays.

**for** `i = 1` to `ncells` **do** {Loop through the cells}

Determine if the cell is in contact with the substrate. The substrate is set at the  $z = 1$  plane.

Set the change in the radii of the cell to be zero.

Determine if the cell is in contact with a top plate.

`nr_contact = 0`, initialize the number of cells in contact with the current cell.

`compress` (see section 6.8). Determine how much the cell axes should be compress due to the physical presences of other cells and the turgidity of the cells.

**end for**

**for** `i = 1` to `ncells` **do** {Loop through the cells}

`all_force` (see section 6.7). Determine all the forces acting on a cell.

**end for**

Set `totalnonzero`, the total number of nonzero elements in the sparse matrix `M` to be 0.

**for** `i = 1` to `ncells` **do** {Loop through the cells}

`move1cell` (see section 6.9). Set the force vector `b`.

Update `totalnonzero`.

`find_stretchingfluidode` (see section 6.9.1). Determine how the cell changes shape.

**end for**

*Solve the system  $M\dot{\mathbf{x}} = \mathbf{b}$  using a sparse iterative solver from package `IML++`.*

Initialize some parameters.

Allocate memory.

Pack the arrays in the method necessary for the solver.

`CompRow_Mat_double` (see `SparseLib++`). Create the sparse matrix.

`VECTOR_double`. Create the initial guess for the vector.

`MATRIX_double`. Create a work matrix.

`DiagPreconditioner_double`. Create the diagonal preconditioner.

`GMRES` (see `IML++`). Solve the system with iterative solver.

Print warnings if something goes wrong.

Store the solution in `b`.

Free memory.

---

## 6 Other functions

### 6.1 `clm_update_all`

This C function sets the cell location matrix by looping through all cells. In order to make identify neighboring cells more efficiently there is a cell location matrix (*clm*) which identifies which cells are in a coarse spatial grid. Searches for neighboring cells can be restricted to cells in nearby coarse grid squares.

The cell location matrix is a spatial grid and each grid variable *clm* points to a cell which is located in the south west grid box. If more than one cell is in the box the other cells are stored in *pt.next\_cell*.

---

**Algorithm 6** Structure of `clm_update_all`

---

Set *pt1* to null.

Clear the three dimensional matrix *clm* by setting it to null.

**randlist** (see section 6.3). Randomizes the integer array *randlisti* with entries from 1 to *ncell*. This will allow us to loop through the cells in a random order.

**for**  $l = 1$  to *ncells* **do** {Loop through the cells in a random order via the index *randlisti*}

    Identify the cell as *pt*.

    Find the location of *pt* on the *clm* grid.

    Print some warnings if *pt* leaves the *clm* grid.

    {In following **IF THEN** statement *pt1* acts as a tmp variable which is reset in the **IF** statement. It tests to see if the *clm* point is null or pointing to another cell. If it is point to another cell the other cell is placed in *pt.next\_cell* so all the cells at the *clm* location are linked together until *pt.next\_cell* equals null.}

**if** *pt1* equals *clm* at *pt*'s position **then** {Until *pt1* is reset, this tests to see if *clm* is null}

        Reset *pt1* to *clm*.

        Set *clm* to point to *pt* and set *pt.next\_cell* (the pointer associated with the cell identifying another cell at this *clm* location) to *pt1*.

**else**

        Reset *clm* to point to the current cell *pt* and have *pt.next\_cell* point to null.

**end if**

**end for**

---

## 6.2 orient\_once

Given a nonzero vector  $\mathbf{b}$  this C function randomly sets  $\mathbf{a}$  and finds  $\mathbf{b}$  and  $\mathbf{c}$  so that all three vectors are orthonormal. They are the axes of the ellipsoid. (It assumes that  $\mathbf{b}$  is not parallel to the randomly set  $\mathbf{a}$ .)

---

**Algorithm 7** Structure `orient_once`

---

```
for  $i = 1$  to  $ncells$  do {Loop through all the cells}
  Set  $\mathbf{a} = (a_1, a_2, a_3)$  where  $a_j$  are randomly chosen from the interval
   $[-0.5, 0.5]$ . { $\mathbf{a}$  is the axis of the ellipsoid which determines orientation.}
  Normalize  $\mathbf{a}$ 
  Set  $\mathbf{b}$  orthonormal to  $\mathbf{a}$ , {use Gram-Schmidt}
  Calculate  $\mathbf{c}$  so that  $\mathbf{a}$ ,  $\mathbf{b}$  and  $\mathbf{c}$  are orthonormal
  Set motion control ( $gamax$ ) to zero.
end for
```

---

## 6.3 randlist

This C function returns an array `randlisti` which is full of integers ranging from 1 to `ncells` in a random order.

It loops through the cells and puts a random number in a temporary array and initializes an array of indices. Then it calls `hpsort` which puts the temporary array in ascending order and keeps track of the original order with the array of indices. Thus the index array is randomized.

## 6.4 find\_neighbour

This C function finds the neighboring cells. Its argument is a particular cell  $C_c$ .

---

**Algorithm 8** Structure of `find_neighbour`

---

Find  $C_c$ 's location on  $clm$  matrix by finding the indices for the (upper north east)  $clm$  point.

Print warnings if the cell has left the  $clm$  grid.

**for** the 27 adjacent  $clm$  grid points **do**

$po1 = clm$

**if**  $po1$  is not null **then** {Test to see if there are cells in the adjacent  $clm$  grid squares}

        Increment the number of neighbors  $nr\_neigh$

        Put the nearby cell in the neighbour array.

        Calculate the distance  $d$  between the cell  $C_c$  and the neighbor  $po1$

**if**  $C_c = po1$  **then** {The neighbor cell is the current cell}

            Decrease  $nr\_neigh$

**else if**  $d > 3dist$  **then** {Test if the cell is too far away.}

            Decrease  $nr\_neigh$ {The cells are too far apart}

**else**

            Set  $po1$  to be  $po1.next\_cell$

**end if**

**end if**

**end for**

---

## 6.5 campdir

This C function takes a cell as an argument and determines the direction of the orientation stimulus. The user needs to define this function. The orientation stimulus should be a unit vector stored in *plt* → *dir*. An example function is given which sets the orientation stimulus to (0, 0, 1).

## 6.6 orient

This C function orients the cells. The user needs to define the orientation stimulus which is in the cell vector *dir*. It takes a cell *plt* as an argument. If the cells are being stimulated to move by the motion stimulus they move for a period of *rtim* in one direction. If the cells are not being stimulated to move they are stationary (if modified they can randomly move).

---

**Algorithm 9** Structure of orient

---

```
if  $plt.rtim \leq 0$  then {Cell is responsive to change directions}
  if  $motioncycle \neq motioncycleold$  then {Check if cell is changing from
  random motion to directed motion or vica versa}
    campdir (see section 6.5). Get the direction from the orientation
    input.
     $randomforce = (.4 * drand48() + .8)$ . Give a slight random variation
    to the force of each different cell.
     $tmporient = vdot(a, dir)$ .
    if  $|1 - tmporient| < .0001$  then {Check to see if the new cell direction
    is the same as the old one.}
       $\kappa = 0$ .
    else if  $|1 + tmporient| < .0001$  then {Check to see if the new cell
    direction is opposite the old one.}
       $\mathbf{a} = -\mathbf{a}$ . Change the cell direction.
       $\kappa = 0$ .
    else {Directions are different we must rotate the cell.}
      Find the vector  $\mathbf{v} = \mathbf{a} \times \mathbf{dir}$ .
      Normalize  $\mathbf{v}$ .
       $tmpangle = arccos(tmporient)$ . Calculate the angle between  $\mathbf{a}$  and
       $\mathbf{dir}$ .
       $tmpangle = tmpangle * dt / turning\_time$ . Find the angle of rota-
      tion for each time step.
      Set some useful values
      Determine the rotation matrix.
       $\kappa = 1$ .
    end if
     $rtim = turning\_time$ . Reset the turning time.
  end if
  if  $rtim > 0$  and  $\kappa = 1$  then {Check if the cell is changing direction due
  to stimulus}
    Rotate the vector  $\mathbf{a}$ .
  end if
end if
 $rtim = rtim - dt$ . Update the recovery clock.
```

---

## 6.7 all\_force

This C function determines all the forces acting on a cell, it has as an argument a cell *plt*.

---

**Algorithm 10** structure of `all_force`

---

`find_stretchingforce` (see section 6.10).

Set entries in the location vector for the sparse matrix **M**.

**for**  $l = 1$  to  $nr\_neigh$  **do** {Loop through neighbor cells}

Find **r**, the vector between the center of  $p1t$ , the current cell, and the neighbor cell  $C_n$ .

Find the components of **r** in terms of the axes of the current cell.

Find some important distances.  $d1$  is the distance from the current cell center to its edge in the direction of **r**.  $d2$  is the distance from the center of  $C_n$  to its edge in the direction of **r**.  $d$  is the distance between the edge of the current cell and the edge of  $C_n$  in the direction of **r**. The value  $dadj = ||\mathbf{r}|| - 1.2d1 - 1.2d2$  is the distance between the edge of the buffers around each cell of the two cells.

**if**  $dadj < -\min(1.2d1, 1.2d2)$  **then** {Check to see if cell edge plus a buffer has passed the center of the other cell}

Reset  $dadj$  to slightly larger than  $-\min(1.2d1, 1.2d2)$  so the buffer does not pass through the cell center.

**end if**

**if**  $dadj < 0$  **then** {Check to see if the two cells are in physical contact (this includes the buffer region)}

Determine which half of the axis the forces will be acting on.

Make adhesion stronger for different cell type.

$surfa = \mathbf{surface}$  (see section 6.7.1). Determine how much surface contact there is.

Set the coefficient for the matrix **M**.

Fill the matrix **M**.

Determine which cell is softer and let the overlap distance be in that cell.

$force1 = \mathbf{aforce}$  (see section 6.7.3) in the direction of vector **a**.

Determine the force on the cell due to the adhesion of the other cell or compression of other cell.

$force1 = \mathbf{aforce}$  (see section 6.7.3) in the direction of vector **b**.

Determine the force on the cell due to the adhesion of the other cell or compression of other cell.

$force1 = \mathbf{aforce}$  (see section 6.7.3) in the direction of vector **c**.

Determine the force on the cell due to the adhesion of the other cell or compression of other cell.

Add the forces to the neighboring cell.

Determine which neighboring cell is close to the direction which the cell wants to move so active force can be applied to that cell. Set that cell to be  $po2$ .

**end if**

**end for**

---

---

Structure of `all_force` cont.

Store index for the diagonal entry of the matrix.  
Add a viscous term for the surrounding fluid.  
Add viscous terms if cell is close to substrate  $z = 1$  or the top plate.  
**if** `po2` is not null **then** {There is a cell to grab onto for moving}  
    Find the vector  $\mathbf{r}$  between the current cell center and `po2`'s center.  
    Find the components of  $\mathbf{r}$  in terms of the axes of the current cell.  
    Find the components of  $\mathbf{r}$  in terms of the axes of cell `po2`.  
    Determine if the force is going to act on the front or back of the cells.  
     $force = active\_force$  (see section 6.7.5). Find the active force the cell exerts.  
    **if** The cell is in contact with the substrate **then**  
        Apply the force only to the  $z$  direction of the cell.  
         $force1 = bound\_force$  (see section 6.7.7). Apply the rest of the force to the boundary.  
         $force1 = bound\_force$  (see section 6.7.7). Apply the rest of the force to the boundary.  
        **if** The cell is in contact with the top plate **then**  
             $force1 = bound\_force2$  (see section 6.7.8). Apply the force from the top plate.  
        **end if**  
    **else if** The cell is in contact with the top plate **then**  
        Apply the force only to the  $z$  direction of the cell.  
         $force1 = bound\_force2$  (see section 6.7.8). Apply the rest of the force from the top plate.  
    **else** {The cell is not in contact with the substrate or the top plate}  
        Apply the force to the cell.  
    **end if**  
    **else** {There is no cell to grab onto for moving}  
        **if** The cell is in contact with the substrate **then**  
             $force = active\_force$  (see section 6.7.5). Find the active force the cell exerts.  
             $force1 = bound\_force$  (see section 6.7.7). Apply the force to the boundary (only in the  $x$  and  $y$  directions).  
        **end if**  
        **if** The cell is in contact with the top plate **then**  
             $force = active\_force$  (see section 6.7.5). Find the active force the cell exerts.  
             $force1 = bound\_force2$  (see section 6.7.8). Apply the force to the boundary (only in the  $x$  and  $y$  directions).  
        **end if**  
    **end if**  
**end if**

### 6.7.1 surface

This C function determines how much surface area of a cell is in contact with other cells. It takes three numbers as arguments  $d$ ,  $d1$  and  $d2$ . The first  $d$  is the distance between two cells, the second  $d1$  is the distance from the center of one cell to its surface and the third  $d2$  is the distance from the center of the other cell to its surface. It returns a number  $fad$  which is a measure of how much surface area the two cells have in contact with each other.

---

**Algorithm 11** Structure of **surface**

---

*surface* = 0. The cells are not in contact.

$h = d1 + d + d2$

**if**  $d < 0$  **then** {The cells are in contact}

$surface = \pi(2h^2d1^2 + 2h^2d2^2 + 2d1^2d2^2 - h^4 - d1^4 - d2^4)/(4h^2)$ . This is the area of the region bounded by the circle of intersection of two spheres of radius  $d1$  and  $d2$ .

**end if**

---

### 6.7.2 surfaceb

This C function determines how much surface area of a cell is in contact with the boundary. Its first argument  $d$  is the distance from the buffer to the boundary. The second argument is  $d1$  the distance from the center of the cell to the edge of the buffer.

---

**Algorithm 12** Structure of **surfaceb**

---

*surface* = 0. The cells are not in contact.

**if**  $d < 0$  **then** {The cell and the substrate are in contact}

**if**  $d1 + d < 0$  **then** {The substrate is beyond the cell center}

$d = -d1$ . A fix to make the calculations work.

**end if**

$surface = \pi(d1^2 - (d1 + d)^2)$ . The area of the circle formed when a plane intersects a sphere normal to the radius.

**end if**

---

### 6.7.3 aforce

A C function which returns the adhesive forces acting on a cell. It takes as arguments  $d$  the distance from the two cells' membranes including a buffer

region in the direction of one of the axis adjusted due to pressure of a cell (negative means overlap), cell adhesion a measure of how the cells stick to each other,  $d_1$  same as  $d$  only not including the buffer region,  $d_2$  same as  $d$  with out the buffer and measuring from a distance inside the membrane (a negative buffer region),  $pforce$  the cell's pressure,  $surfa$  the surface contact, and  $cvect$  the component of  $\mathbf{r}$  in the direction of one of the axis of the cell. It returns the value  $fo$ .

---

**Algorithm 13** Structure of `aforce`

---

Set adhesions scale factor  $atmp$ .

**if**  $d < 0$  and  $d_1 > 0$  **then** {The cells overlap in the buffer region but not the surface of the ellipsoids}

$fo = atmp \sin(.5\pi d_1 / (d_1 - d))$ . A positive force pulling the cell together.

**end if**

**if**  $d_1 < 0$  and  $d_2 > 0$  **then** {The cell's surface overlap but not too far into the interior}

$fo = pforce |\sin(.5\pi d_1 / (d_1 - d_2))|$ .

**if**  $fo > atmp$  **then** {If the force is greater than adhesion break the bonds}

$fo = 0$

**end if**

**end if**

**if**  $d_1 < 0$  and  $d_2 < 0$  **then** {The cell's overlap far into the interior (in the negative buffer zone)}

$fo = pforce$

**if**  $fo > atmp$  **then** {If the force is greater than adhesion break the bonds}

$fo = 0$

**end if**

**end if**

---

#### 6.7.4 bforce

A C function which returns the forces acting on a cell due to the boundary. It takes as arguments  $d$  the distance from the cell membrane to the substrate including a buffer region around the membrane (negative means overlap), cell adhesion a measure of how the cells stick to each other,  $d_1$  same as  $d$  only not including the buffer region,  $d_2$  same as  $d$  with out the buffer and measuring from a distance inside the membrane (a negative buffer region),  $pforce$  the cell's pressure,  $surfa$  the surface contact, and  $cvect$  the component of the axis in the  $z$  direction. It returns the value  $fo$ .

---

**Algorithm 14** Structure of **bforce**

---

Set adhesions scale factor  $atmp$ .

**if**  $d < 0$  and  $d_1 > 0$  **then** {The cells overlap in the buffer region but not the surface of the ellipsoids}

$fo = atmp \sin(.5\pi d_1 / (d_1 - d))$ . A positive force pulling the cell toward the substrate.

**end if**

**if**  $d_1 < 0$  and  $d_2 > 0$  **then** {The cell's surface overlap but not too far into the interior}

$fo = pforce |\sin(.5\pi d_1 / (d_1 - d_2))|$ .

**if**  $fo > atmp$  **then** {If the force is greater than adhesion break the bonds}

$fo = 0$

**end if**

**end if**

**if**  $d_1 < 0$  and  $d_2 < 0$  **then** {The cell's overlap far into the interior (in the negative buffer zone)}

$fo = pforce$

**if**  $fo > atmp$  **then** {If the force is greater than adhesion break the bonds}

$fo = 0$

**end if**

**end if**

---

### 6.7.5 active\_force

This C function determines the active force of a cell when pulling on another cell. It takes a cell *p1t* as an argument. It returns the magnitude of the force.

---

**Algorithm 15** Structure of `active_force`

---

Set the motion indicator to 1 to indicate active motion.  
**if**  $gmax > threshold$  **then** {cell is signaled to actively move}  
    Ramp the force up to a maximum force in 20 time steps.  
**else** {If the cell is not stimulated to move}  
    Ramp the force down in 20 time steps to a minimum or zero.  
    Set the motion indicator to 0 to indicate not actively moving.  
**end if**  
Randomly vary the force.

---

### 6.7.6 active\_bforce

A C function which determines the forces due to the boundary. It takes as an argument a cell *p1t*. It sets the active forces due to contact with the surface. `active_force` must be called before this routine.

---

**Algorithm 16** Structure of `active_bforce`

---

Use the force set in `active_force`.  
Randomly vary the force.

---

### 6.7.7 bound\_force

A function which determines the forces due to the boundary. The passive forces due to contact with the surface and the active forces due to contact with the surface is calculated. (Adhesion and a correction are done in `move1cell`).

---

**Algorithm 17** Structure of `bound_force`

---

Define several values including:  $d1$  the distance from the center of the ellipsoid to the boundary in the  $z$  direction,

$surfa = \text{surfaceb}$  (see section 6.7.2). Define area in contact with the boundary.

Find the force due to the surface on each axis and add it to the front or back of the cell. The force is found using the function `bforce` (see section 6.7.4).

Find the forward force on the cell due to the active force on the substrate via the function `active_bforce` (see section 6.7.6).

---

### 6.7.8 bound\_force2

A function which determines the forces due to the top boundary. Same structure as `bound_force`.

### 6.8 compress

This routine determines how much pressure each cell would have if compressed by neighbors.

---

**Algorithm 18** Structure of compress

---

**for**  $l = 1$  to  $nr\_neigh$  **do** {Loop through neighbor cells}

Find  $\mathbf{r}$ , the vector between the center of  $p1t$ , the current cell, and the neighbor cell  $C_n$ .

Find the components of  $\mathbf{r}$  in terms of the axes of the current cell.

Find some important distances.  $d1$  is the distance from the current cell center to its edge in the direction of  $\mathbf{r}$ .  $d2$  is the distance from the center of  $C_n$  to its edge in the direction of  $\mathbf{r}$ .  $d$  is the distance between the edge of the current cell and the edge of  $C_n$  in the direction of  $\mathbf{r}$ . The value  $dadj = \|\mathbf{r}\| - 1.2d1 - 1.2d2$  is the distance between the edges of the buffers around each cell.

**if**  $d < 0$  **then** {Check to see if cell edge has passed the edge of the other cell (they are compressing each other).}

Determine which half of the axis the forces will be acting on.

Determine which cell is softer and let the overlap distance be in that cell.

Find the maximum overlap for all the neighboring cells.

**end if**

Include the compression due to the substrate.

Include the compression due to the top plate.

**end for**

Add the maximum compression for the front and the back of each axis and store in cell variables.

---

## 6.9 move1cell

This C function moves each cell. It takes as an argument a cell  $p1t$ .

---

**Algorithm 19** Structure of `move1cell`

---

Calculate the total force in the  $x$ ,  $y$  and  $z$  directions.

**if** cell touches top or bottom boundary **then**

**if** the force in the  $z$  direction is less than a set `actb_bound` force and is positive **then**

Define some values.  $dd$  is the distance from the buffer to the substrate.  $d$  is the distance from the ellipsoid boundary to the substrate.  $ddd$  is the distance from the negative buffer to the substrate.

**if**  $dd < 0$  **then** {The surface is above the buffer to the ellipsoid surface}

$$force = -\sin(.5\pi d/(d - dd))$$

**end if**

**if**  $d < 0$  and  $ddd > 0$  **then** {The ellipsoid surface is below the substrate but not too much}

$$force = |\sin(.5\pi d/(d - ddd))|$$

**end if**

**if**  $d < 0$  and  $ddd < 0$  **then** {The ellipsoid surface is below the substrate too much}

$$force = 0 .$$

$$fz = 0.$$

**end if**

Add the modified force to the front or back of the cell.

Recalculate the force totals.

**end if**

**end if**

Determine if the cell is compressed or stretched.

If the diagonal elements of  $\mathbf{M}$  are too small make them 1 and set the right hand side to zero. (Makes solving the system easier.)

Pack the forces in the right side vector  $\mathbf{b}$ .

Finish packing the matrix  $\mathbf{M}$ .

Make some adjustments to make it easier for the solver.

Reset the pressure values.

---

### 6.9.1 find\_stretchingfluidode

This C function determines how the cell shape is altered. This function determines the new lengths of the principle axes of the ellipsoid assuming nonlinear spring in parallel with a Maxwell element (a linear spring in series with a dashpot) to make something like a Kelvin model<sup>2</sup>. The nonlinear spring is more resistant to compression when the cell is deformed too much (this is to account for the nucleus). It also forces the cell to maintain constant volume.  $F_e$  is the external forces,  $k_1$  is the spring constant for the linear spring in the Maxwell element,  $\mu$  is the dashpot resistivity. This function takes a cell as an argument.

---

**Algorithm 20** Structure of find\_stretchingfluidode

---

Define several quantities.

Get variables ready to run `dlsode` a Fortran ode solver.

Call `dlsode` (a freely distributed Fortran code used to solve systems of differential equations). To solve the system of equations describe in Dallon and Othmer (2004) pg 209 equation 3.

Check to make sure the solver was successful.

Change the size of the cell.

In order to get the correct pressure for the cell solve the system of equations with the information from the ODE solver using `dgesv` (see LAPACK).

---

### 6.9.2 fcell

This function is required by `dlsode` to define the system of differential equations.

---

**Algorithm 21** Structure of fcell

---

Define the change in the length of the axes.

Find the nonlinear spring force using `f2sp` (see section 6.14).

Define the system.

Use `dgesv` to solve the system.

---

---

<sup>2</sup>see Biomechanics Mechanical Properties of Living Tissue by Y.C. Fung

### 6.10 find\_stretchingforce

This C function finds the pushing or pulling force due to the rheology of the cells. The force is describe in Dallon and Othmer (2004) pg. 219. This routine solves the equations A.20-A.22.

---

**Algorithm 22** Structure of find\_stretchingforce

---

Define important values.  
Find the old change in the radii.  
Find the force due to the nonlinear spring using `f2sp` (see section 6.14).  
Find the new change in the radii.  
Find the force due to the nonlinear spring using `f2sp` (see section 6.14).  
Solve for the pressure.

---

### 6.11 write\_to\_file

This C function writes the cell size and location to a file for graphing.

---

**Algorithm 23** Structure of write\_to\_file

---

Write the time.  
**for**  $i = 1$  to  $ncells$  **do** {Loop through the cells}  
    Write the coordinates of the cell center.  
    Write the **a** axis.  
    Write the **b** axis.  
    Write the **c** axis.  
    Write the lengths of the axes including the buffer region.  
**end for**

---

### 6.12 read\_ifile

This C function reads parameters from the file ifile.dat

### 6.13 write\_to\_file\_wave

This C function writes the motion indicator information for graphing

### 6.14 f2sp

This C function defines the nonlinear spring.

### **6.15 pf2sp**

This C function defines the derivative of the nonlinear spring.

### **6.16 sr determine\_motion\_stimulus**

This function is used for the motion stimulus and should be altered by the user.