SimGen— a generalised multi-scale modelling and simulation program

William R. Taylor

[†]Division of Mathematical Biology,
MRC National Institute for Medical Research,
The Ridgeway, Mill Hill, London NW7 1AA, U.K.

December 2, 2014

Part I

Overview and Model Design

1 `main`

The program **`SimGen`** simulates the motion and interaction of a collection of objects. The objects can be a sphere, tube or ellipsoid and can be bonded or linked together. Each object can contain other objects (called children) and the motion of a parent object applies to all its offspring. The objects have no mass or momentum and any movement is applied as a fixed-size step which is made irrespective of whether the move creates a violation of any constraint, such as bond-length or leads to an undesired steric overlap between objects.

In the same way that objects move irrespective of any constraint, so too, bond lengths and steric constraints are independently enforced by applying a fixed length movement to restore the bond length or separate an overlap. Precedence in the application of these, often conflicting corrections, is random as each algorithm executes as an asynchronous process (a thread), all operating on the same set of positional coordinates.

The process generating motion is the **`shaker`**, bonds and link lengths are fixed by the **`linker`** and steric clashes by the **`bumper`**. In addition there is a routine that keeps children within the boundaries of their parent (the **`keeper`**) and one that helps maintain local geometry (**`tinker`**) and the configuration is visualised (using openGL) by the **`viewer`**.

The structure of the objects is set-up through a command script that is interpreted by the **`models`** routine and the parameters that control their behaviour is read by the **`params`** routine. Both these routines have many built-in settings that allow the constraints commonly found in protein and nucleic acids (such as their secondary structures) to be automatically generated without having to specify every detail.

`SimGen` is currently configured to accept a hierarchy of objects in up to 10 levels (not all of which need be used). The top level is referred to as the ”world” and the lowest as ”atomic” (not because the objects necessarily represent atoms but only because they have no further divisions). Objects interact directly (through bumps, bonds and links) only with other objects on the same level. Besides following the geometric transformations of their parents, interaction between levels is confined to keeping families (parent and children) together. As mentioned above, this can be enforced using the **`keeper`** but the default action is for the parent to track the centroid position of its children and for the children to shift together so their centroid tracks the position of their parent. This mutual co-location is implemented by the **`center`** (”er” not ”re”) routine.

In the input stream (read by **`models`**) each group of objects is introduced with the line `GROUP 0 N`, where `N` is the number of children in the group. For example:

```
GROUP 0 1
    GROUP 0 2
        GROUP 0 3
            ATOM...
            ATOM...
            ATOM...
        GROUP 0 4
            ATOM...
            ATOM...
            ATOM...
            ATOM...
```

specifies a world consisting of a single object that contains one child that itself has two children with three and four children each at the atomic level. The atoms at the lowest level are specified by the `ATOM` command, which conveniently, has a format that is identical to that used by the Protein structure DataBank (PDB), which is the source of most of the structures being simulated.

The above example does not specify the nature of the objects in each group (although those specified by `ATOM` commands are, by default, spheres). Objects on the same level within a group (such as the `ATOM`s or their parents in the above example) are assumed to have the same properties, and to prevent repeated specification of these, they are defined in a parameter file (read by **`params`**). This file consists of a set of rows and columns with each column specifying a set of parameters for each level.

After a model has been set-up and executed, after an initial period of equilibration (during which conflicting constraints will attempt to resolve themselves), the structure will remain still or jiggle around, depending on the degree of motion assigned to the objects on each level. Behaviour of a more purposeful nature can be introduced through the **`driver`** routine which implements a set of user-specified behaviours, such as moving components of the model to execute some action. Irrespective of the instructions issued from this routine, all the remaining routines will continue to act independently

Each routine will be elaborated below, beginning with the set-up routines (**`models`** and **`params`**) followed by the main routines: **`shaker`**, **`linker`**, **`bumper`** and **`keeper`** that implement their specifications. Finally, the minor routines: **`tinker`**, **`sorter`**, **`center`** and **`looker`** will be described, along with some specialised routines under the title **`fixers`**.

2 `params`

2.1 Model parameter specification

The **`params`** routine reads the file specified by the first argument on the **`SimGen`** command line (usually ”something.run”), or by default, ”test.run” if no name is given. The main purpose of the **`params`** routine is to read a file of parameters (specified by the `PARAMS` command described below) and set-up the generic behaviours for each level in the model hierarchy.

Each property is specified on one line as a series of integers corresponding to successively deeper levels. There can be up to 10 levels and unused fields are left as trailing zeros which can be followed by an optional comment. Presently, there are also ten different properties specified by the parameter file which will be described below. (With their parameter name italicized in the sub-section title).

Firstly however, the routine expects a declaration of the molecule type, which can be any of: `PROT`, `RNA`, `DNA`, `CHEM` or `CELL`. Any other text will probably behave in a protein-like way. This can be followed by a secondary command that is used by the **`viewer`** to specify a colour scheme for rendering which will be described under the section on that routine.

Except for models specified as `CHEM`, it is assumed that the lowest (most detailed) level of the molecular representation will be a backbone ’virtual’ chain, without any atomic detail.

2.1.1 `Line 1`: *type* of object

As described above, there can be three types of object, a: sphere, tube and ellipsoid that are specified by the numbers 1,2,3, respectively. There can also be a virtual object that can have all the properties of a sphere but is not rendered except as a tiny sphere. Ellipsoids (with axis lengths, A, B, C) can only be oblate ($A < B = C$) or prolate ($A > B = C$). These shapes are generally, referred to as spheroids but the term ellipsoid will be retained below to maintain a distinction from the sphere-object (which persists even when $A = B = C$).

If a negative number is specified, then the object is rendered as a wire mesh but otherwise behaves identically.

2.1.2 `Line 2`: ***size*** of object

The second line specifies the size of the object. For spheres and tubes, this is the diameter while for ellipsoids it is the diameter at the centrt across the axis of symmetry (B or C, in the above specification). The values specified in the parameter file are used at a tenth their size inside **`SimGen`**.

With a specified known type of molecule (protein of nucleic acid), standard values will be adopted and maintained for bond lengths and internal secondary structure links (more below). This means that higher levels (eg: protein domains) are free to move within their specified constraints. While this is usually the normal desired behaviour, it is sometimes required that the local starting geometry of the configuration should be preserved. This is implemented by the **`tinker`** and a negative prefix on the `size` value will signal that it should be active during the simulation for that level of structure.

2.1.3 `Line 3`: ***bump*** size of object

Although it might be assumed that objects will bump when their surfaces make contact, there are situations where it is useful to have a different bump size, which can be specified on this line.

2.1.4 `Line 4`: number of ***links***

Any object can form links to any other object (on the same level) but to avoid allocating unused space to objects that never link, the maximum number of links can be specified on this line.

2.1.5 `Line 5`: number of ***bonds***

The number of bonds an object can make is specified in an identical manner but has some implications for polymers (protein, nucleic acid) that are expected to form chains.

Specifying a non-zero number of bonds for a polymer automatically results in the creation of bonds between adjacent monomers within their group. So in the `GROUP` structure declared in the previous section, if bonds were specified at the atomic level, then there would be two chains of 3 and 4 monomers. If bonds were allocated to the next level up (nominally the secondary structure level), then a chain of two secondary structures would be created. However, for consistency, the program will also create the bond joining their two atomic level chains (giving a single chain of 7 monomers at the ’atomic’ level). If the atomic level has no bonds, then each secondary structure in the chain will contain a group of free-floating atoms.

If the number of bonds specified is negative, this is taken as a flag to create a circular chain. So in the current example, a negative number on the atomic level will create two circular chains of 3 and 4 monomers. If the secondary structure level is also negative, and the atomic level positive, there will be a ’ring’ of 2 secondary structures containing a ring of 7 monomers and if both values are negative, the ’ring’ of 2 will contain a triangular and a square ring with no bond between them.

In some higher level polymer domain configurations and chemical structures, each object may form multiple bonds and the value of the bond number is equivalent to the maximum valance of the object.

2.1.6 `Line 6`: ***move*** step size

The step size applied by the **`shaker`** is specified on this line. As with all motion, it is applied from the top level downwards with every movement at the higher level also propagated to all offspring below.

If the specified value of *size* (S) is positive, the motion is a translation in a random direction with a magnitude of $S/100$, followed by a rotation in a random direction with a angular displacement of $S/100$ radians.

If the value is negative, then just the translation is applied.

2.1.7 `Line 7:` ***keep*** **within parent boundaries**

If a child strays outside its parent's boundary (as defined by its *size*), then a shift (or kick) is applied to redirect it inwards with a magnitude specified on this line. Note that the value appears in the position of the level of the parent but is applied to the level below).

If the parent is a sphere, then 'inwards' is towards its centre and for an ellipsoid, 'inwards' is towards the nearest focus (simpler than calculating surface normals). For a tube, the default action is to shift the child towards the surface whether inside or outside the tube, in a direction perpendicular to the tube axis.

If the value of the parameter is negative, then these default actions are reversed and children within spheres and ellipsoids will be directed towards the parental surface while children will be free to wander inside the tube.

Some of these behaviours can be modified for individual objects by values specified in the model description read in the **`models`** routine described below.

2.1.8 `Line 8:` ***bond*** **length**

The bond lengths specified on this line are the default ideal values which every bond associated with that level will be refined towards. As not all objects are spheres, the bond length is not a simple centre—centre distance. For spheres, it is the distance between the surfaces (so the centre—centre distance = $bond + size$). For tubes, it is the desired separation between the ends of two tubes (axis end—end distance) and for ellipsoids it is the equivalent pole—pole distance between the ends of their axes of symmetry. (The ends of the A axis, in the example above.).

A negative value of the bond length evokes a behaviour that prevents the default joining of chains of children when their parents are in a chain. This is useful when constructing a fiber formed from unbonded subunits (such as actin).

2.1.9 `Line 9`: ***hard*** bump repulsion

The way in which objects bump is simple only at the atomic level where a kick is applied to both atoms in a direction away from their common centre with a magnitude of $M/100$, where M is the specified value. This repulsion is applied when the distance between the atom centres is less than the value of *bump* and is independent of the degree of penetration or any other constraint.

2.1.10 `Line 10`: ***soft*** bump repulsion

Objects at a higher level do not behave in this way but can 'happily' interpenetrate, providing none of their children are clashing. The degree to which they are separated is a function of the number of inter-family bumps on a non-linear scale between the value of the *hard* repulsion specified on the previous line and the $soft$ value specified on this line. Specifically, the kick size, k, is: $k = g * S + (1 - g) * H$, where S and H are the *hard* and $soft$ values and $g = \exp(-m^2)$, for m child collisions.

As the value of g ranges from 1 (no bumping children) to 0 (many bumping children) the degree of parental repulsion will scale from $soft$ to *hard*. This means that if $soft$ has a non-zero value, then there will be a 'soft' repulsion before any of the children in the two families clash. This creates a 'jelly' like behaviour to the collision of high-level objects.

If the value of $soft$ is negative, then the behaviour of the objects reverts to hard repulsion.

2.2 An example input file

```
PROT sec
    0,    0,    3,    2,    1,    0,    0,    0,    0,    0     // type
 9999, -120,  -50,   -8,    2,    0,    0,    0,    0,    0     // size
    0,  120,   50,    8,    8,    0,    0,    0,    0,    0     // bump
    0,    0,    0,    6,    4,    0,    0,    0,    0,    0     // nlinks
    0,    0,    0,    1,    1,    0,    0,    0,    0,    0     // nbonds
    0,    0,    1,    1,    1,    0,    0,    0,    0,    0     // move
    0,    2,    5,   10,    0,    0,    0,    0,    0,    0     // keep
    0,    0,    0,    0,    4,    0,    0,    0,    0,    0     // bond
    0,   10,   10,   10,   10,    0,    0,    0,    0,    0     // hard bump
    0,    1,    2,    5,    0,    0,    0,    0,    0,    0     // soft bump
```

In this example, a protein model is set-up with 4 levels (not including the world). From bottom up, the levels corresponds to: α-carbon-chain, secondary structure elements, domains and individual chains (subunits).

The first column describes the world simply as big (9999). Other values can be assigned to the world and, except that it will not move or be rendered, it is a normal object. A useful behaviour is to give it a smaller size and assign a value to *keep* to stop objects wandering too far away.

The atomic level (fifth column) specifies each atom is a small sphere ($size = 2$) and is in a linear chain ($nbonds > 0$). So with a relatively long bond between atoms ($bond = 4$), the chain will will appear as a ”ball-and-stick” model (with an atom—atom distance of 6). It has a bump size that is larger than this ($bump = 8$) but as bonded atoms do not bump (more detail in the `bumper` section), this will first apply to neighbours-but-one along the chain, preventing the virtual bond angle dropping much below 90°. (Since $6 * 6 + 6 * 6 \approx 8 * 8$).

Each atom has the capacity to make 4 links, two of which will be automatically used in local secondary structure formation). The atoms will have a slight motion and a reasonably strong (*hard*) bump repulsion.

The secondary structure level is a chain ($nbonds = 1$) of tubes ($type = 2$) with equal *size* and *bump* values that just enclose their atomic configurations.

Note that the *size* values at this level and above are negative, indicating that there is no generic bond length and the local geometry will be preserved by the `tinker`. If the *size* value were positive, the **`linker`** would enforce the generic bond length, which is set at zero, and would link the tubes without any spacer. So not a disaster, but causing a marked shift away from the original structure once the simulation is started. Quite a few links are allocated (6) and as none are automatically assigned at this level (?), there is scope to tie-down the structure further. A value of *keep* is specified to hold the atoms to the tube surface (except in loop regions where this is not enforced). Both the *hard* and *soft* bump settings are now used so the secondary structures have a bit of a jelly-like constitution.

At the next higher (domain) level, the objects are ellipsoids (of softer jelly) and are still in a chain. Parents are also a bit more relaxed about keeping their children close. These trends continue to the next (subunit) level which is no longer chained and is rendered as a virtual sphere (*type* = 0).

2.3 Global value specification

Although the main purpose of **`params`** is to parameterise the model behaviour, it also sets-up some general behaviours most of which are used in 'debugging' the model specification script described in the next section.

2.3.1 `NORUN`

Compile and run the model but do not execute the **`driver`** routine.

2.3.2 `NOMOVE`

Compile and run the model but do not execute the **`shaker`**, **`keeper`** or **`bumper`** routines. (Or the **`driver`** routine). This command is very useful for checking (and as the **`viewer`** is active, seeing) if the model is OK before it has the chance to do anything (like explode).

2.3.3 `NOVIEW`

The **`viewer`** is not executed. Essential for use in batch mode, say on a computer cluster.

2.3.4 `HIDDEN`

Freezes execution of objects that are out of view, either outside the field-of-view or beyond the back-plan or behind the point-of-view. Distant objects also become solid and their contents are frozen.

2.3.5 `TINKER <update>`

Activates the regularisation routine `tinker()`, described under the section on minor routines.

2.3.6 `SCALE <in> <out>`

Sets two scale factors, the first is applied to all coordinates read in by **`models`** and the second to coordinates written out (usually from **`looker`**). Default values are used for protein and nucleic acid for input but when writing very large structures in PDB format, a small scale factor is often needed to get the values to fit the format.

The line: `SCALE 0.158 0.633 # scalein = .6/3.8, scaleout = 3.8/6`, scales a PDB input to have a CA—CA bond length of 0.6 and an output length of 0.38 (tenth size).

2.3.7 `SHRINK <value>`

This scale factor is applied on every simulation cycle to the highest level (1, not counting the world). A factor of $1- < value >$ is applied to the positions of all level-1 objects causing them to fall towards the centre of the world. Its main use has been in testing collisions as it saves time waiting for wandering objects to collide.

2.3.8 `PARAM <filename>`

This is the key command that specifies the name of the file containing all the model parameters described in the first section (which is usually called ”something.model”). Up to five models can be introduced by separate `PARAM` lines and are assigned numbers starting with zero upwards, in order of their occurrence.

The `PARAM` command(s) must follow all the above commands and be followed by the `END` command.

2.3.9 `END`

Marks the end of the parameter input stage and the start of the model description.

3 `models`

3.1 Overview of the object hierarchy

The **`models`** routine continues reading the main input file (”test.run”) after **`params`** has completed. It reads-in a hierarchy of groups of objects (as outlined in the `main` section). Besides the basic hierarchy sketched previously, each group declaration can be preceded by a series of commands (referred to below as a ”Move set”) that specify geometric operations that will be applied to the group and all its sub-groups. Correspondingly, at the end of each group, another set of instructions (called the ”Bond set”) can be included to specify bonds and links or modify those that have been created automatically. Although the end of each group is usually identified automatically (after all its children have been read or by the appearance of a command that implies the end), it is convenient for the moment to use the `TER` command which explicitly marks the end of a group.

Using `TRANS` to represent a Move-set and `REBOND` to represent a Bond-set, then the model outlined in the opening section can be elaborated as it might appear in the file ”test.run”. (Indentation is not required but can help clarity).

```
    PARAM some.model
END
    TRANS set-1
    GROUP 0 1
        GROUP 0 2
            TRANS set-2
            GROUP 0 3
                ATOM...
                ATOM...
                ATOM...
            TER
            GROUP 0 4
                ATOM...
                ATOM...
                ATOM...
```

```
                    ATOM...
                REBOND set-A
                TER
           TER
      REBOND set-B
      TER
END
```

The transforms specified in the first Move-set (`TRANS set-1`) will apply rotations and translations to everything that is created whereas the second set will apply just to the first group of 3 'atoms'. The first Bond-set (`REBOND set-A`) could specify new bonds within the second group of 4 atoms while set-B could make links between and within the two groups of atoms.

Representing a Group (`GROUP`) declaration as `G`, a Move-set (`TRANS`) declaration as `M`, an atomic level (`ATOM`) declaration as `A`, a Bond-set (`REBOND`) declaration as `B` and a group termination (`TER`) as `T`; the model in the example above could be written: `MGGMGAAATGAAAABTTBT`. Using this representation, each group-structure has the form `MGABT`, or writing the optional commands in lower-case: `mGAbt`. If `*` now designates any number of atoms or any number of valid group-structures, then the input stream expected by **`models`** is: `mG*bt`. Expanding each '`*`' recursively within parentheses and bracketing groups on the same level, the above example can be seen to be a valid stream which will be parsed as: `MG( G( [MG( AAA )T][G( AAAA )BT] )T )BT`.

In a structure that includes many repeated substructures, it would be tedious to specify the group structure for each occurrence. To avoid this, a group structure can be contained within a file and just the filename specified (using the `INPUT` command, of which more below) instead of a full group specification. As different instances of each substructure may require a different Move-set, these can precede the `INPUT` command but the end of an input file is treated as an End-of-Group signal, so group definitions cannot span files. `INPUT` files can, of course, contain `INPUT` commands.

3.2 Move-set commands

These are described in the order in which they are applied to the completed group. Characters within brackets indicate options and if separated by a bar '|', are optional strings. Strings within angle brackets represent numbers.

3.2.1 `TRANS <x> <y> <z>`

A translation of the group by a displacement vector {x,y,z}.

3.2.2 `SPIN[XYZ] <theta>`

A rotation of the group by an angle of `theta` degrees about the specified axis (X,Y,Z). For example: `SPINY 90` rotates the group by 90° around the Y-axis.

3.2.3 `SPINS <tx> <ty> <tz>`

Is a more general `SPIN` command, rotating about the X, then Y, then Z axes by the three specified angles: `tx`, `ty`, `tz`.

3.2.4 `TWIST <x><y><z> <p><q><r> <twist>`

Rotate about the line defined from {x,y,z} to {p,q,r} by and angle `twist`.

3.2.5 `HELIX <x><y><z> <turns>`

Move the group by `x,y`, rotate by `<turns>` degrees about the Z axis then move `z` along Z. (N.B., the commands `TWIST` and `TRANS` can be combined to generate a helix along any axis.)

3.3 `GROUP <sort> <kids> [<x><y><z> <p><q><r>]`

The full syntax of the `GROUP` command allows not only the the specification of the number of expected children (`kids`) but also the `sort` of group that is wanted.

This is an integer number that acts on a non-spherical object type (tube or ellipsoid) to modify its length. This is not something that can be specified in the parameter file as each object will typically have a different length depending on the number of atoms it contains.

The axis of these objects can also be specified as two additional end-point coordinates ({x,y,z} and {p,q,r}).

3.3.1 Tubes

If the molecule type is protein and the group is at the secondary structure level (one up from atomic) then the `sort` value is used to specify secondary structure sort as: 0 = loop, 1 = an α-helix and 2 = a β-strand. Each have a particular tube length/thickness ratio determined by the helical rise/residue along the secondary structure and the expected bulk for a given loop length. For each sort (0,1,2) the values used are: 0.5, 1.5, 3.0. If the end-points are unspecified, then the axis is estimated from the terminal residues. Even when the axis is specified, it provides only the direction and is set to pass through the centre of the object.

If the molecule type is nucleic acid, the length is set in a similar manner using the known rise/basepair.

Tubes that are not associated with a known secondary structure can have their length specified by the value of `sort`, either as a multiple of their thickness (up to 10 times) or if the value is negative, by the distance between the end-points (which must then be provided).

3.3.2 Ellipsoids

As there are no secondary structures automatically associated with the ellipsoid shape, the length of the ellipsoid axis is set only by the value of `sort` but whereas the value for a tube was a simple ratio (1..10), the value for an ellipsoid runs from 1..21 with values over 10 creating oblate shapes and under 10, prolate shape

(10=spherical). The scaling is not linear but follows the progression: $n/10$ up to 10 and $10/n$ over 10 where $n = abs(10 - sort)$.

Any negative value of `sort` again causes the length of the specified axis to be used.

3.3.3 Nucleic acids

For both RNA and DNA, the `GROUP` command can adopt a two aliases, `DOUBL` and `SINGL`, to deal with differences between double and single stranded segments. The latter is in fact identical to `GROUP` and was included just for neatness but the `DOUBL` command uses the value of `sort` to label a strand (Watson) that requires a complementary strand (Crick) to occur later with the negated value of `sort`.

The default construction for nucleic acids treats a base-pair as a secondary structure so when a `DOUBL` command is encountered, rather than allocating the specified number of `kids` as individual atoms, the equivalent number of two-atom groups (basepairs) are created and the following `ATOM` records are used to fill the first child in each basepair. When the complementary strand is encountered, the the second child is then filled (running backwards). Clearly, complementary strands must be equal in length and the first atom in Watson will be basepaired with the last atom in Crick.

3.4 Bond-set commands

3.4.1 `REBOND` and `REJOIN` commands

The commands that create and modify a chain take the form: `RE[BOND|JOIN|TERM] mode <from> <to>`. For example `REBOND pdbid 22 66` sets a bond between the atoms with residue numbers 22 and 66. But what if the residue numbers in the PDB file (which became `ATOM` records) are inconsistent or missing or, as is likely, residues 22 and 66 are already bonded?

So as not to rely on PDB specified residue numbers **`SimGen`** maintains three internal atom counts as well as a separate count of all objects (called their Unique IDentifier, or UID). The `allatom` count starts with the first and ends with the last atom. The `teratom` count is set to zero with the start of each group and between these is the `endatom` count that is re-zeroed automatically by any Bond-set command or explicitly by an `INPUT` command with a zero appended (`INPUT0`).

These numbering schemes are selected by the `mode` string which can be set as: `group`, `local`, `atomid`, `unique` and `pdbid`, referring respectively to the `teratom`, `endatom` and `allatom` counts, the UID and the PDB number. The first two can obviously only be used withing a level of nesting (scope) where their values have not been re-zeroed but the `atomid` and `unique` counts are universal. The uniqueness of the PDB numbering depends both on the original source and how often it is reread using the `INPUT` command, so must be used cautiously.

Only the `unique` values can be used to rewire objects higher than the atomic level but this must be done carefully as internal consistency checks refer only to the atomic level chain (see below) so any higher level rewiring may well get wiped-out.

The `REJOIN` command has an identical syntax and operation but creates a virtual bond that is not refined or rendered. (In other words it makes a gap without terminating the chain).

Both commands overwrite whatever existing bond may have been there so after a series of reconnections, there may even be no free termini. To avoid this, the `RETERM` command can be used to specify new start and final positions of the chain using the `from` and `to` fields as start and end. Again, care must be taken as the inclusion of a `RETERM` command triggers the execution of the internal consistency check described below. Otherwise it is assumed that the rewiring operations are harmless.

3.4.2 Rewiring consistency

Even when applied properly, the above commands have the potential to alter the chain connectivity of the polymer, leaving the higher levels out-of-step with the order at the atomic level. To restore consistency, once all the groups have been completed, a subroutine traces the new chain path at the atomic level and renumbers the higher levels in a sequential order. This process is only executed if there is one or more `RETERM` commands with each new terminus taken as a start point. So if the chain topology has been altered but the termini remain the same, a `RETERM` command re-specifying the original termini must be included.

At the domain level (atomic-2) and above, the atomic level chain can make multiple entries and exits from the same object leading to branched topologies. To accommodate branched topologies, the original bond allocation must be equal to the maximum valance encountered on each level. The recalculated numbering of the higher level objects then follows the order in which they are encountered by a path (a double linked-list) that traces a path following the outside of the topology tree. (See the actin example below for clarification).

3.4.3 `BONDS` and `LINKS` commands

The `BOND` and `LINK` commands operate irrespective of whether there is any polymer chain and each follow the same syntax to read a list of simple bond/link assignments, as: `[LINKS|BONDS] mode filename`, where `mode` is the numbering scheme to be used (as described above) and the `filename` contains a list of lines of four numbers specifying: `<donate> <accept> <type> <link>`. The value of `type` sets the bond thickness on rendering or how far a link can extend without breaking and the `link` value is used to determine which ends to join between elongated objects. (More below).

Unlike the `RE[BOND|JOIN]` commands, `BONDS` and `LINKS` will not overwrite an existing bond/link but will fill the first free slot in the bond/link list of the donor object. In the context of a chain, they behave as cross-links and if the `BONDS` are specified after any `RETERM` command, they will not alter the chain topology.

3.4.4 `SHEET` and `BETA` commands

This pair of commands are a specialised version of the `LINKS` command, described above, and are used to create cross-links between strands in a β-sheet. As would be expected they can be used only when the molecule type is a protein.

The command `SHEET [mode]`, alerts **`models`** to expect a series of links, but rather than open a new filename, these are just read from the current stream with each link specified by the command `BETA <donate> <accept> <strength>`. Generally, these commands are kept within the group that contains the sheet although links between groups can be defined (I think).

The string `mode` specifies the numbering scheme as described above and if omitted the default numbering is the `group` atom number.

4 Example Application (actin)

In this section, the construction of a model of an actin molecule is described starting with a single actin.

4.1 Monomer construction

actin.run

```
PARAM actin.model
END
GROUP 0 1
INPUT actin.dat
END
```

The parameter file `actin.model` specifies a 4-level hierarchy which is almost identical to the example given previously.

actin.model

```
PROT sse
   0,    3,    3,    2,    1,    0,    0,    0,    0,    0,   // type
3000, -150,  -70,   -8,    2,    0,    0,    0,    0,    0,   // size
   0,  150,   70,    8,    8,    0,    0,    0,    0,    0,   // bump
   0,    0,    0,    6,    4,    0,    0,    0,    0,    0,   // links
   0,    0,    3,    1,    1,    0,    0,    0,    0,    0,   // bonds
   0,    1,   -1,   -1,    1,    0,    0,    0,    0,    0,   // move
   0,    1,    1,    1,    0,    0,    0,    0,    0,    0,   // keep
   0,    0,    0,    0,    4,    0,    0,    0,    0,    0,   // bond
   0,   10,   10,   10,   10,    0,    0,    0,    0,    0,   // hard
   0,    1,    2,    5,    0,    0,    0,    0,    0,    0,   // soft
```

The `INPUT` file `actin.dat` reads each of actin's four domains in separate files into an oblate ellipsoid and reconnects the chain through the domains using `REBOND` commands.

actin.dat

```
GROUP 5 4    -5.0  1.0  0.0    5.0  0.0  0.0
INPUT actin.dom1.dat
INPUT actin.dom2.dat
INPUT actin.dom3.dat
INPUT actin.dom4.dat
REBOND atomid  32 127
REBOND atomid 172  33
REBOND atomid  90 173
REBOND atomid 216 281
REBOND atomid 372 217
REBOND atomid 280  91
RETERM atomid   1 126
```

Each of the four domains are also contained within an ellipsoid and the top part of the first two show how the ellipsoids are tailored to each domain shape. The division of the ATOM records into secondary structure groups was generated automatically, along with the axis end-points by a program that is described in an appendix.

actin.dom1.dat

```
GROUP 10 19    0  1  0    0 -1  0
GROUP 0 6
ATOM      1  CA  GLY A   1     -23.868 -17.022  -6.339  1.33  1.00
ATOM      2  CA  GLY A   2     -21.762 -20.183  -6.100  1.33  1.00
ATOM      3  CA  GLY A   3     -18.845 -17.924  -6.394  1.33  1.00
ATOM      4  CA  GLY A   4     -20.602 -14.455  -6.961  1.33  1.00
ATOM      5  CA  GLY A   5     -20.699 -14.221  -3.138  1.33  1.00
ATOM      6  CA  GLY A   6     -17.825 -16.538  -2.703  1.33  1.00
GROUP 2 5   -14.79 -14.10 -3.15   -4.36 -8.33 -0.83
ATOM      7  CA  GLY A   7     -14.841 -14.717  -4.363  1.33  2.00
ATOM      8  CA  GLY A   8     -12.300 -13.449  -1.827  1.33  2.00
ATOM      9  CA  GLY A   9      -9.949 -10.497  -2.418  1.33  2.00
ATOM     10  CA  GLY A  10      -6.767 -10.364  -0.491  1.33  2.00
ATOM     11  CA  GLY A  11      -4.370  -7.485  -0.937  1.33  2.00
GROUP 0 6
ATOM     12  CA  GLY A  12      -1.017  -8.244   0.718  1.33  1.00
ATOM     13  CA  GLY A  13       0.995  -5.192   1.721  1.33  1.00
:
120 more lines
```

```
:
ATOM    119  CA  GLY A 119      -0.359 -27.580 -16.028  1.33  1.00
ATOM    120  CA  GLY A 120       1.825 -26.970 -12.925  1.33  1.00
GROUP 1 6    1.77 -24.25 -13.35    0.40 -17.76 -15.53
ATOM    121  CA  GLY A 121       2.324 -23.445 -11.325  1.33  3.00
ATOM    122  CA  GLY A 122       3.335 -22.259 -14.760  1.33  3.00
ATOM    123  CA  GLY A 123      -0.391 -22.075 -15.685  1.33  3.00
ATOM    124  CA  GLY A 124      -1.234 -18.947 -13.632  1.33  3.00
ATOM    125  CA  GLY A 125       2.182 -17.434 -14.080  1.33  3.00
ATOM    126  CA  GLY A 126       1.159 -16.277 -17.640  1.33  3.00
SHEET
BETA   9 20 7
BETA   20 9 7
BETA   9 58 6
BETA   8 57 6
BETA   58 9 6
:
40 more lines
:
BETA   19 29 1
BETA   17 30 1
BETA   17 12 1
BETA   16 32 1
BETA   12 17 1
```

The first domain is contained in a sphere (`sort = 10`) but because it is an ellipsoidal object it has axis end-points defined. Since the value of `sort` is positive, these end-points only specify the direction of the axis and not its length.

The second domain has a negative `sort` value so the length of the specified axis sets its size. As the absolute value of `sort` is over 10, the object will be a prolate ellipsoid with a diameter scaled to preserve the axial ratio associated with `sort` = 15.

actin.dom2.dat

```
GROUP -15 8      0 0 -25   0 -10 25
GROUP 2 5    1.42 -7.71  8.19    6.31 -14.04 18.22
ATOM      1  CA  GLY A   1       1.363  -7.636   8.264  1.00  2.00
ATOM      2  CA  GLY A   2       3.361  -8.978  11.218  1.00  2.00
ATOM      3  CA  GLY A   3       3.992 -12.395  12.769  1.00  2.00
ATOM      4  CA  GLY A   4       5.160 -12.308  16.329  1.00  2.00
ATOM      5  CA  GLY A   5       7.149 -15.046  17.922  1.00  2.00
GROUP 0 13
ATOM      6  CA  GLY A   6       7.323 -15.011  21.823  1.00  1.00
ATOM      7  CA  GLY A   7      10.702 -14.236  23.284  1.00  1.00
ATOM      8  CA  GLY A   8       9.826 -16.218  26.547  1.00  1.00
:
many more lines
:
```

The full path of the α-carbon-chain through the four domains is: 1-2-1-3-4-3-1, which corresponds to a branched (Y-shaped) topology with domain 1 being visited three times. To accommodate this, the value of *nbonds* = 3 in the parameter file for the domain level. Strictly, all the others should be 2, but the program allocates a minimum of 2 anyway if a chain is requested. In a linear chain these are referred to by the 'short-hand' pointers `sis` for the previous object in the chain and `bro` for the following. Thus a chain of 4 objects (A–D) appears as:

```
        bro-->B     bro-->C     bro-->D     bro-->x
         A - - - - - B - - - - - C - - - - - D
    x<--sis     A<--sis     B<--sis     C<--sis
```

With `x` being an end-of-chain marker (actually the parent cell). If the chain is circular then `D.bro->A` and `A.sis->D`. In terms of the underlying structure that holds the bonding data, this is equivalent to:

```
       bond1-->B   bond1-->C   bond1-->D   bond1-->(x)
       next1 = 1   next1 = 1   next1 = 1   next1 = -1
         A - - - - - B - - - - - C - - - - - D
  -1 = next0   0 = next0   0 = next0   0 = next0
(x)<--bond0   A<--bond0   B<--bond0   C<--bond0
```

The seemingly redundant `next` values tell where to find the next link. However, in a branched topology they have a less trivial job as can be seen when the

`C` object is made into a side branch:

```
                    bond1-->(x)
                    next1 = -1
                      C
                2 = next0
                B<--bond0
                      |
                    bond2-->D
                    next2 = 0
                      B
      bond1-->B     bond1-->C     bond1-->(x)
      next1 = 1     next1 = 0     next1 = -1
        A - - - - - - B - - - - - - D
 -1 = next0     0 = next0     0 = next0
(x)<--bond0     A<--bond0     B<--bond0
```

So following the path of links from `A1`:

```
A1-->B1-->C0-->B2-->D0-->B0-->A0-->(x)
```

In this representation, the connection through the actin domains is:

```
A1-->B0-->A2-->C1-->D0-->C0-->A0-->(x)
```

4.2 Filament construction

Actin polymerises into a helical filament (with a left-handed twist) which consists of repeated actin molecules related by a rotation of -166° and 27.5Å translation [?]. The subunit used here is taken from the structure of the filament (PDB code = `1M8Q`)and as this has a fiber axis along Z, the simple `HELIX` command (described in the previous section) can be used to regenerate the filament. However, as the internal coordinates of the molecule will have been centred on the origin, these require the first two values of the `HELIX` command to shift the molecule (in X and Y) off the axis, then the standard helical parameters quoted above can be applied. The file `actin13.run` generates 13 subunits which is the half-repeat length of the filament. For longer filaments, clearly it would be worthwhile writing a script to generate the run-file.

actin13.run

```
NOMOVE
HIDDEN
PARAM actin.model
END
GROUP 0 13
HELIX -16  -5    0.0     0
INPUT actin.dat
HELIX -16  -5  -27.5  -166
INPUT actin.dat
HELIX -16  -5  -55.0  -332
INPUT actin.dat
HELIX -16  -5  -82.5  -498
INPUT actin.dat
HELIX -16  -5 -110.0  -664
INPUT actin.dat
HELIX -16  -5 -137.5  -830
INPUT actin.dat
HELIX -16  -5 -165.0  -996
INPUT actin.dat
HELIX -16  -5 -192.5 -1162
INPUT actin.dat
HELIX -16  -5 -220.0 -1328
INPUT actin.dat
HELIX -16  -5 -247.5 -1494
INPUT actin.dat
HELIX -16  -5 -275.0 -1660
INPUT actin.dat
HELIX -16  -5 -302.5 -1826
INPUT actin.dat
HELIX -16  -5 -330.0 -1992
INPUT actin.dat
END
```

Part II

Implementation and Simulation

5 `shaker`

The shaker is a simple routine that traverses the tree of objects and gives each one a random displacement as specified by the *move* value in the parameter file.

The tree is completely traversed firstly to implement translations which are accumulated on the decent of the tree and applied directly to the coordinates of the object (rather than use a `cell.move()` call which would needlessly traverse the subtree from that object). Translations are applied as 1/100 times the value specified in the parameters.

Secondly, the tree is re-traversed to implement rotations. As there is no simple way to accumulate random rotations about different centres (?) these are individually applied to each sub-tree using the `cell.spin()` function. Rotations are applied in radians as 1/100 times the value specified in the parameters. This is an arbitrary balance between translation and rotation which could be improved to better reflect the inertial properties of the object.

As stated in the description of model construction, a negative value of *move* causes the rotational component to be skipped. This is desirable for objects with limited freedom such as those in a chain or otherwise cross-linked.

6 `bumper`

The **`bumper`** routine has the task of identifying objects that have approached closer than permitted and repelling them by a fixed-size kick specified in the parameter file. It must overcome two difficulties: firstly, for large numbers of objects, it is too computationally expensive to compare all-against-all and, secondly, the objects do not all have simple shapes so the surface-surface distance between all combinations of object shapes must be accommodated.

6.1 Avoiding $N \times N$

The **`bumper`** routine uses the hierarchic structure of the data to avoid a N^2 order calculation between all pairs of objects. Beginning at the highest level (1) all objects at that level are compared pairwise but only if two objects at this level are in collision, are their children then considered. In computer-graphics terms, the high-level objects are the ”bounding-boxes” for their children.

However, before the parents are repelled, a calculation is made of how many collisions there are between their children. These two distinct calculations of collisions within a family and collisions between families are performed by `bumpin()` and `bumpex()`, respectively.

6.1.1 `bumpin()`

If the number of children in a family is less than 20 (`SWITCH`), then `bumpin()` uses a simple pairwise algorithm (in `getBumps()`) to provide a list of objects that are potentially in collision. Over `SWITCH` children, then an approximate algorithm is used that is based on the partially sorted X,Y,Z lists maintained by `sorter` (more below). This selection is based on the largest dimension of the object: the maximum axis length for an ellipsoid or the larger of the length and diameter of a tube. The list is sorted by degree of violation so `bumpin()` will deal with the worst cases first.

`bumpin()` firstly checks the true separation of the two objects (`a,b`) using `touch()` which returns a negative distance if the objects inter-penetrate. (Details will be provided in the next section). If `a` and `b` are not atoms, then a count (`m`) is made of how many collisions occur between their children using `bumpex()` (of which, more below). If both *hard* and *soft* collision parameters have been specified, then the degree of repulsion is calculated using `m` as described earlier (and explicitly in code below) with the result being used by `part2cells()` to push the objects symmetrically back towards a distance (`d`) where they are no longer in collision.

```
int Cell::bumpin () {
// the children of the current cell <this> are checked for intra-family bumps
	:
	kidlev = level+1;
	:
	for (i=0; i<in; i++) { Cell *a = list[i].a, *b = list[i].b;
		bump = touch(a,b);
		if (bump > -NOISE) continue;
		if (kidlev<depth) m = bumpex(a,b); else m = 0;  // bumping a+b children parted in bumpex()
		if (exempt(a,b)) continue;	// exempt parents (exempt atoms skipped in getBumpin())
		// the pair (a,b) are bumping so repel more with more bumping children (m)
		//	unless weight=0 then just use unmodified <soft> value
		if (weight) { // Gaussian switch from soft to hard with increasing <m>
			d = (float)m; d = exp(-d*d);
			boot = d*soft + (1.0-d)*hard;
			boot *= kick;
		} else { boot = kick; }
		d = (a->xyz|b->xyz)-bump*over; // clash = -ve bump
		part2cells(a,b,d,-boot); // -kick = repel only
		:
	}
}
```

Both `bumpin()` and `getBumps()` employ a filter encoded in `exempt()` that is `TRUE` if the two objects are exempt from collisions, for example, if they are bonded or linked. Note that `exempt()` only checks at the atomic level in `getBumps()` for some reason. In `bumpin()`, however, `exempt()` is only called after `bumpex()` as the children (and their offspring) of two exempt objects might well be making unwanted collisions.

6.1.2 `bumpex()`

`bumpex()` evaluates each pair of children between their two colliding parents and returns the number of collisions. As it does so, it also takes stems to rectify the situation by separating the clashing kids. As it is known to which parent each child belongs, they are given a nudge back towards their parent before being separated. The strength of these kicks depend on both the *hard* and *soft* parameter values: the nudge back home is always *soft* while the separation is *hard* at the atomic level and *soft* for higher levels. (NB this is multiplied by strenght=0.1 but not sure it should be).

```
int bumpex ( Cell *a, Cell *b ) {
// the children of the cell <a> and <b> are checked for inter-family bumps
float   strength = 0.1;
        :
        kidlev = level+1;
        if (kidlev==Data::depth) kick = hard; else kick = soft;
        axis = b->xyz - a->xyz;
        axis.setVec(soft);        // soft length vector from a to b (NB has to be set at atom level)
        :
        DO(i,a->kids) { Cell* ai = a->child[i];
                DO(j,b->kids) { Cell* bj = b->child[j];
                        if (exempt(ai,bj)) continue;
                        bump = touch(ai,bj);
                        if (bump > -NOISE) continue;
                        d = (ai->xyz|bj->xyz)-bump*over; // d = target gap (NB clash has -ve bump)
                        moveCell(ai,axis,-1);        // nudge ai towards a
                        moveCell(bj,axis, 1);        // nudge bj towards b
                        part2cells(ai,bj,d,-kick*strength); // -kick = repel only
                        :
                        n++;
                }
        }
        return n;
}
```

The application of the `bumpin()`, `bumpex()` pair is not recursive: ie: `bumpex()` does not call `bumpin()` on colliding children. However, once `bumpin()` has completed at one level, it continues to traverse the hierarchic tree of objects.

6.2 Calculating contact

`SimGen` employes three object types, giving six possible types of encounter which are dealt with by the `touch()` routine.

```
float touch ( Cell *a, Cell *b )
{ // closest approach between two object surfaces
  // +ve = separation, -ve = penetration depth
       :
       rab = ra + rb;  // average of bump radii
       tab = ta * tb;  // product of object types
       if (tab>1) s = Seg(b->endN,b->endC);
       switch (tab) {  // smallest type cell first
               case 0: // virtual spheres don't bump?
                       return (pa|pb)-rab; // 9999.9;
               case 1: // spheres = centre distance
                       return (pa|pb)-rab;
```

```
          case 2: // sphere+tube (closest approach to line segment or ends)
                  if (pa.vec_in_seg(s)) return pa.vec_to_line(s)-rab;
                  return fmin(pa|s.A,pa|s.B) - rab;
          case 3: // sphere+ellipsoid (in keeper.cpp)
                  return vec_to_egg(pa,s,db) - ra;
          case 4: // tubes = closest approach of 2 line segments
                  return seg_to_seg(Seg(a->endN,a->endC),Seg(b->endN,b->endC))-rab;
          case 6: // tube+ellipsoid (in bumper.cpp)
                  return tube_to_egg(a,b);
          case 9: // ellipsoid (in bumper.cpp)
                  return egg_to_egg(a,b);
     }
}
```

6.2.1 Sphere/Sphere

Is the simplest case, with surface contact made at the average distance of their bumping diameters. (Or the sum of the corresponding radii: $R_{ab} = R_a + R_b$, for two objects a and b).

6.2.2 Sphere/Tube

For a sphere and a tube, the closest approach is the shortest sphere-centre to tube axis line-segment, less their joint radii (R_{ab}). If a perpendicular construction from the sphere centre to the axis line lies within the tube end-points then this, less R_{ab}, is the closest approach of their surfaces. Otherwise, it is simply the shorter tube-end to sphere distance (less R_{ab}).

6.2.3 Sphere/Ellipsoid

The distance of a point from the surface of a general (scalene) ellipsoid in not trivial, however, as **`SimGen`** only deals with spheroids, a simple construction based on the foci of their ellipse-of-rotation can be used to decide if a point is inside or outside the ellipsoid. A path from one focus to any point on the ellipse and back to the other focus, has a constant length. (A property often exploited to draw an ellipse with a fixed length of string). The distance of the foci from the centre can be solved from the lengths of the axes, A and B, as: $c = \sqrt{(a^2 - b^2)}$, where a and b are the semi-axis lengths. (See construction below). So if the summed distance from any point to the foci is longer than the ’string’, it is outside and, if less, it is inside.

```
                             * external point
                     ..-+-../
                .    d /|  /    .           d = focus pole distance
              .       /b| /       .         b = minor axis length/2
             :       /  |/c        :        c = foucs cent distance
        endN |------x---+---x------| endC   x = focii on major axis
                       cent                 a = major axis length/2
length of focus1-surface-focus2 path:
at major axis = c+a+(a-c) = 2a
at minor axis = 2d, dd = bb+cc
since the paths are equal: d = a;
so cc = dd-bb = aa-bb
and  c = sqrt(aa-bb)
```

The sum of the foci distances less the ’string’ length is zero on the surface but elsewhere is not the true distance to the surface. However, when scaled by 1.4, this value is a good approximation to the true distance to the surface for both prolate and oblate ellipsoids and is the value returned by the routine `inEgg()`, which encodes this algorithm. In its most minimal form, the algorithm only needs to calculate two distances but as the foci positions are not stored, `inEgg()` does a bit more.

In the range $0 \ldots R_{ab}$ a more complicated but accurate routine `vec_to_egg()` is used to return the true value of the distance between the surfaces.

6.2.4 Tube/Tube

If a mutual perpendicular line (the contact normal) is constructed between the axes of a pair of tubes and if the ends of this lie between the end-points of the tubes, then this is the closest approach. Otherwise one of the four end-end distances will be shortest. The shortest distance, less R_{ab} is the distance between the surfaces.

6.2.5 Tube/Ellipsoid

The distance of a tube to an ellipsoid is found using the same algorithm described above for a point (sphere) and ellipsoid (`inEgg()`) by iteratively bisecting the line between the tube end-points.

Starting with the three end—mid—end points along the axis ($p1, p2, p3$), then if any corresponding value ($d1, d2, d3$) returned by `inEgg()` is negative, there is a clash. Otherwise, the point associated with the largest value can be excluded and the calculation repeated with the remaining two points and their midpoint. The algorithm converges rapidly and is stopped when the points get too close. At the end, the true distance to the ellipsoid surface is returned using the more complicated `vec_to_egg()` routine.

```
float tube_to_egg ( Cell *a, Cell *b ) {
// returns an approximation to the closest approach of a tube <a> to an ellipsoid <b> surface
// NB the value returned by inEgg() is not a surface distance but is zero on the surface
// NB assumes radially symmeric ellipsoid
        :
        p1 = a->endN; p2 = a->xyz; p3 = a->endC;
        LOOP { float d;
                d1 = inEgg(p1,cb,sizeb);
                d2 = inEgg(p2,cb,sizeb);
                d3 = inEgg(p3,cb,sizeb);
                if (d1<0 || d2<0 || d3<0) return -999.9; // flag bump;
                d = p1|p3;
                if (d < 0.01) { // pretty close
                        d = vec_to_egg(p2,cb,sizeb);
                        return d - sizea*0.5;
                }
                if (d1+d2 < d2+d3) {
                        p3 = p2; p2 = p1 & p2;
                } else {
                        p1 = p2; p2 = p3 & p2;
                }
        }
}
```

6.2.6 Ellipsoid/Ellipsoid

As would be expected, the best is kept to last. Surprisingly, there is no analytic solution for the contact normal between two ellipsoid surfaces as the expression for this is quartic and requires a numerical solution. A solution probably could be found for the more symmetric case of two spheroids but not by me. Instead, an iterative algorithm is used to find the contact normal using an algorithm that is an extension of that used for the simpler tube/ellipsoid problem.

Rather than iteratively bisect a line, as was done on the tube axis, extending the approach to a surface leads to the iterative trisection of a triangle — or rather two, as there are two ellipsoids to consider. If the triangles are trisected using a mid-point/vertex construction, the sub-triangles become progressively elongated. To avoid this, an internal triangle was constructed from the mid-points of each edge. (So strictly, each triangle is quad-sected).

A starting set of triangles was obtained from the end-points of the axes, giving 8 triangles per ellipsoid and a starting pair was selected which had the shortest mid/mid point distance. In all the iterations, the mid point is not simply the mean of the vertices but is the point where the extension of a line from the centre through this point cuts the ellipsoid surface. The utility routine `sholl()` that calculates this is given below and is called by the wrapper routine `shell()` that identifies the ellipsoid type as there is different routine (`shall()`) that deals with scalene ellipsoids.

```
Vec sholl ( Vec line, Vec cent, float A, float B, Vec axis ) {
// returns the point on the ellipsoid (<axes> = A>B=C at 0) surface cut by a <line> from the <cent>re
/*
in the plane of the major axis (A) and the <line> with components a,b to A,
the point where the line cuts the surface has corresponding components g,h.
Now    gg/AA + hh/BB = 1
and    g/a = h/b
so     gg = AA(1-hh/BB) = aa.hh/bb
       AA - hh.AA/BB = hh.aa/bb
       AA = hh.aa/bb + hh.AA/BB
       hh = AA/(aa/bb+AA/BB)
*/
Vec    surf;
float  AA=A*A, BB=B*B, aa,bb,b, d,ff,gg,hh;
       line -= cent;                            // shift line to origin
       b = line.vec_to_line(axis);              // perpendicular dist from line to axis
       bb = b*b;
       ff = line.sqr();
       aa = ff-bb;
       hh = AA/(aa/bb+AA/BB);
       gg = aa*hh/bb;
       d = sqrt((gg+hh)/ff);
       surf = line*d;                           // extend <line> to ellipsoid surface
       return cent+surf;                        // added back to centre
}
```

As the selection of the two starting quadrants is based on a rough estimate, all 64 distances are ranked and the top three combinations taken as separate starting pairs. The routine then iterates down to the best pair of (sub-)$_n$triangles in each (max $n = 3$) and takes the solution with the shortest separation. As a safe-guard, a number of points (currently 10) are scattered randomly around these mid-points and the closest pair selected.

This solution is then checked using `vec_to_egg()` to find the distance from each point to the opposing ellipsoid. These should be identical if the true contact normal has been found and any discrepancy greater than 50% is reported. This may seem generous but errors arise, not because of a poor solution, but because the ellipsoids may have moved during the calculation as `vec_to_egg()` uses the current axis end-points. With minimal motion, the error is typically less than 0.001%.

7 Bonds and links

The maintenance of bond and link lengths is very similar and the two routines, **`bonder`** and **`linker`**, that implement this task will be considered together. Both recursively traverse the object tree looking for things to fix.

7.1 `bonder`

All of the decisions about what should be bonded are determined at the model construction stage and except for the `CHEM` mode (small molecules, which will be considered later), the bond assignments remain unaltered throughout a simulation.

7.1.1 Bond lengths

The **`bonder`** simply checks if an object has any assigned bonds and if so, uses the utility `part2cells()` to push them towards their assigned bond length. The value for the ideal bond length depends on whether the *size* parameter was given as a positive or negative value. If positive, the value of the *bond* parameter is used (at 1/100 the specified value) and if negative, the starting value of the bond length is used. More specifically, this is the value of the `x` and `z` components of the vector `prox` which is part of the object class (`Cell`) data and can be varied during the simulation. The `prox.x` is the length to the previous object in the chain and `prox.z` the length to the next.

For both generic and specific bond lengths, a random choice is made over which side to refine as, doing both in one pass may mean that they simply cancel each other.

7.1.2 Nucleic acid exceptions

Exceptions need to be made when bonding tubes in nucleic acids, which occur both at the secondary structure level (depth-1) as basepairs and the domain level (depth-2) as segments of double helix.

For basepairs, if these are part of a double helix, their mid-point (object centre) is refined against their generic bond-length. Outside a base-pair, say in a loop region, the 'secondary structure' can contain multiple nucleotides and no bond length is used.

At the domain level, double-stranded DNA segments will always be bonded end-to-end at a specific distance that allows the helix to run continuously from one segment to the next. On the other hand, when the segment is an RNA stem-loop, the chain can enter and exit the same end of the tube or, with an insertion, even through the side. No bonds appear to be enforced (need to fix or check this!).

7.1.3 Cross-links

If the number of bond slots allocated by *nbonds* in the parameter file is more than one, then a check is made for cross-link bonds that will have been recorded in `bond[2]` and above.

7.2 `linker`

7.2.1 Breaking links

The **`linker`** follows the same basic outline as the **`bonder`** but with the main difference that links can be made and broken during the simulation. The dynamic creation of links is not a built-in feature of **`SimGen`** and must be provided through the user-supplied **`driver`** routine. However, if a link becomes over stretched, it is destroyed in **`linker`**. The default length of a link is the *bump* diameter (specified in the parameter file) and the degree to which this can be exceeded is set by the fourth number read on the `LINK` command (I think). The value is held in the `next` value of the `Bond` class which is unused otherwise for links. The default extension is 5%, beyond which the link breaks. (All this hasn't been checked for ages).

7.2.2 Preset link lengths

When links are automatically created for standard secondary structures, they get their lengths from the routine `setLinks()` that converts the code number held in the `type` field to the required value. Links are set not only between the H-bonded connections in the α-helix, i—i+3 and i—i+4, but also between the i-1—i+1 separation along a β-strand.

Links are also set between adjacent β-strands and between packed α-helix–α-helix pairs and packed α-helix–β-strand pairs. These use one of six discrete values between 1.7 . . . 3.2Å.

7.2.3 Link refinement

Between spheres, links are refined using the `part2cells()` utility but for extended secondary structures this is undesirable as it will tend to align their midpoints. Instead, the ideal distance is refined along the line of their closest approach.

Links involving loop 'secondary structures' are refined only when their separation strays beyond ±50% of their bond length.

8 **`keeper`**

The **`keeper`** keeps the children of the current object inside (or on) its surface. There are only the three basic shapes to consider and the treatment of these uses much the same subroutines that were described in **`bumper`**.

As described in the section on the parameter input, if the value of *keep* is positive, then children are confined within spheres and ellipsoids but are confined to the surface of a tube and these roles are reversed when *keep* is negative.

8.1 Spheres

The `packBall()` routine used by **`keeper`** to keep children inside a sphere provides a simple template of the other two routines (`packTube()` and `packEgg()` described below). The code is pretty self-explanatory but contains two aspects the need a comment.

```
int packBall ( Cell* cell, Cell* child, float strict )
{
       :
       radius -= model->sizes[child->level]*0.5;       // keep totally inside
       if (push < -NOISE) shell = 1;                   // confine to shell (+/-margin)
                else      shell = 0;                   // confine inside
       :
       shift = cell->xyz - child->xyz;                 // shift from child to zero
       d = shift.len();                                // distance from child to parent
       if (shell) {
               if (d>radius*margin && d<radius*margout) return 0; // in the margin zone
               if (d > radius) push = -push;           // outside the sphere (so pull)
       } else {
               if (d < radius*margout) return 0;       // within the sphere
       }
       shift *= push;
       child->move(shift);
       return 1;
}
```

As implemented, the code subtracts the child radius from that of the parent (first line of above code), so for spheres, the full body of the child will be kept inside the parent. However for elongated objects, only the radius normal to the axis of symmetry will be used so tubes and prolate ellipsoids can stick-out whereas oblate ellipsoids will be slightly over-confined.

To save a little computation time, a margin is maintained about the surface, within which no action is taken. This is set at ±10% (`margin`=0.9, `margout`=1.1).

Although it appears that this will save only a few arithmetic operations, it should be remembered that the `move()` utility is recursive and will apply the shift to the full underlying sub-tree of objects. The margin also prevents rapid small in/out fluctuations of children that lie close to the surface.

8.2 Tubes

Tubes have a cylindrical body and hemi-spherical end-caps. The default action of **`SimGen`** is to constrain children to lie on the surface of the cylinder and the caps. If the normal from the child intersects the axis line between the end-points then the child is shifted along the normal. If outside the axial line-segment, then it is shifted in the same way as described for a sphere (above), taking the nearest end-point as a centre. If the value of the *keep* parameter is negative, then children that lie inside the tube are left unmoved. The same margin zone described above for spheres is also implemented.

An exception is made for the children of protein loop regions (which also have a 'secondary structure' tube object associated with them but they are much less constrained and are only held inside the tube with 1/10 the weight of the equivalent α-helix and β-strand. In addition, the end-cap constraints are not enforced.

An exception is also made for double-stranded nucleic acid segments where the tube enclosing the double helix is a domain level object (depth-2). It is therefore quite undesirable to have the base-pair 'secondary-structure' tubes confined to this surface but rather their children (the pair of phosphates) should be on the surface. To implement this, a wrapper routine is (`packBase()`) is used to call `packTube()` with a skipped generation as: `packTube(grandparent,child,...)` instead of the normal `packTube(parent,child,...)`. `packBase()` also refines the P—P distance across the basepair and sets the axis end-points of their tube to track the phosphates.

8.3 Ellipsoids

For ellipsoids, the **`keeper`** follows the template for the sphere but uses the utility `inEgg()`, which was described above, to decide who is out and who is inside. It will be recalled that `inEgg()` uses only the distances to the two foci of the ellipse-of-rotation to do this.

Unlike the tube end-caps, where the nearest centre was used to set the shift direction, instead, a weighted combination of both focus distances are used. The weights are taken as the distance to the other focus: so if the child lies closer to focus-1, it will have a larger distance to focus-2 which is used as a weight to give a bigger contribution to the direction of focus-1, and *vice versa*.

9 Minor routines

9.1 `tinker`

Steric exclusion combined with the range of linkage described above can generate a relatively stable structure. However, given the background of "thermal" noise generated by **`shaker`**, any less constrained parts of the structure will be free to diverge from their starting configuration under the given distance constraints. Typically, this involves twisting and shearing that can generate large motions with little violation of the specified distances although, in principle, distances cannot constrain chirality. A general mechanism, based only on local angles and distances was provided to reduce these distortions and was applied equally to all levels that form a chain. The routine that implements this fixing is **`tinker`**.

In a chain segment of five units (designated: b2,b1,c0,a1,a2), six distances were recorded from the starting configuration in the upper half of the matrix of pairwise distances excluding adjacent units. Three angles were also recored as b1-c0-a1 and the torsion angles around b1-c0 and c0-a1. The innermost three distances are held the vector `prox` with the others in `dist` and the three angles in `geom` all of which form part of the data associated with every object (in the `Cell` class). The distances are firstly refined in the routine `fixDist()` followed by (mainly) the angles in `fixGeom()`

The starting values for the three vectors come from the starting model but, unlike fixed bond lengths, these local values can be continually updated as the simulation progresses. Distances can be regularised with little disruption, however, refining torsion angles can sometimes lead to an error propagation with dramatic effects. To limit the potential for this the torsion angles were dialled-up exactly to generate new positions for b2 (b2') and a2 (a2'). These were used to form a basis-set of unit length vectors along: $x = a2' - b2', y = c0 - (a2' + b2')/2$, with z mutually orthogonal. Starting from the centroid of the five points, the coefficients of an equivalent basis-set defined on the original positions were applied to the new basis-set to generate the new coordinate positions. The result is a compromise between angle and position that remains stable over repeated application. The core code for `fixGeom()` is given below and should be self-explanatory.

```
float fixgeom ( Cell *cell, float fix )
:
// quality checks
d = vdif(b1->xyz,a1->xyz);
dlo = c0->prox.y*(1.0-good);
dhi = c0->prox.y*(1.0+good);
if (d<dlo || d>dhi) return 99.9; // proximal distance too poor
d = vdif(b2->xyz,a2->xyz);
dlo = c0->dist.y*(1.0-good);
dhi = c0->dist.y*(1.0+good);
if (d<dlo || d>dhi) return 99.9; // distal distance too poor
d = angle(b2->xyz,c0->xyz,a2->xyz)*180/PI;
if (d < 30.0 || d > 150.0) return 0.0; // base triangle too thin
t1 = torsion(b2->xyz,b1->xyz,c0->xyz,a1->xyz);
if (t1>999.0) { Pt(Bad t1 in) Pi(cell->uid) NL
Pv(b2->xyz) NL Pv(b1->xyz) NL Pv(c0->xyz) NL Pv(a1->xyz) NL exit(1);
}
th = angle(b1->xyz, c0->xyz, a1->xyz);
t2 = torsion(a2->xyz,a1->xyz,c0->xyz,b1->xyz);
if (t2>999.0) { Pt(Bad t2 in) Pi(cell->uid) NL
Pv(a2->xyz) NL Pv(a1->xyz) NL Pv(c0->xyz) NL Pv(b1->xyz) NL exit(1);
}
dt1old = angdif(t1,tau1);
dthold = angdif(th,theta);
dt2old = angdif(t2,tau2);
if (dt1old+dthold+dt2old > 3.0) return 99.9; // too far to fix
// configuration close enough to fix
xyz[0] = b2->xyz; xyz[1] = b1->xyz;
xyz[2] = c0->xyz;
xyz[3] = a1->xyz; xyz[4] = a2->xyz;
// dial-up correct torsion angles
t1 = torsion(xyz[3],xyz[2],xyz[1],xyz[0]);
xyz[0].get_rot(xyz[2],xyz[1],tau1-t1);
t2 = torsion(xyz[1],xyz[2],xyz[3],xyz[4]);
xyz[4].get_rot(xyz[2],xyz[3],tau2-t2);
t1 = torsion(xyz[0], xyz[1], xyz[2], xyz[3]);
th = angle(xyz[1], xyz[2], xyz[3]);
t2 = torsion(xyz[1], xyz[2], xyz[3], xyz[4]);
dt1new = angdif(t1,tau1);
dthnew = angdif(th,theta);
dt2new = angdif(t2,tau2);
// fit positions xyz[0,2,4] to b2,c0,a2
x = xyz[4]- xyz[0];
mid = xyz[4] & xyz[0];
y = xyz[2] - mid;
z = x^y;
mat = Mat(x.norm(), y.norm(), z.norm()); // new unit basis vectors in mat
wat = mat.get_inv();
mid = (xyz[0] + xyz[2] + xyz[4])/3.0; // new CoG
for (i=0; i<5; i++) {  // get xyz from centre in terms of basis set coeficients (abc)
xyz[i] -= mid;
abc[i] = wat * xyz[i];
}
x = a2->xyz - b2->xyz;
mid = a2->xyz & b2->xyz;
y = c0->xyz - mid;
z = x^y;
x.setVec(); y.setVec(); z.setVec(); // old basis vectors
mid = (b2->xyz + c0->xyz + a2->xyz)/3.0; // old CoG
for (i=0; i<5; i++) { // reconstruct new positions with old basis vectors
xyz[i] = mid; // start at centre and sum basis vector components
xyz[i] += x * abc[i].x;
xyz[i] += y * abc[i].y;
xyz[i] += z * abc[i].z;
}

t1 = torsion(xyz[0], xyz[1], xyz[2], xyz[3]);
th = angle(xyz[1], xyz[2], xyz[3]);
t2 = torsion(xyz[1], xyz[2], xyz[3], xyz[4]);
dt1new = angdif(t1,tau1);
```

```
dthnew = angdif(th,theta);
dt2new = angdif(t2,tau2);
if (dt1new>dt1old || dthnew>dthold || dt2new>dt2old) return 0.0;
// shift towards new positions (by fix) if all angles better
shift = (xyz[0]-b2->xyz)*fix; moveCell(b2,shift);
shift = (xyz[1]-b1->xyz)*fix; moveCell(b1,shift);
shift = (xyz[2]-c0->xyz)*fix; moveCell(c0,shift);
shift = (xyz[3]-a1->xyz)*fix; moveCell(a1,shift);
shift = (xyz[4]-a2->xyz)*fix; moveCell(a2,shift);
t1 = torsion(b2->xyz,b1->xyz,c0->xyz,a1->xyz);
th = angle(b1->xyz, c0->xyz, a1->xyz);
t2 = torsion(a2->xyz,a1->xyz,c0->xyz,b1->xyz);
t1 = angdif(t1,tau1);
th = angdif(th,theta);
t2 = angdif(t2,tau2);
tt = t1 + th + t2;
return tt;
}
```

Although only local information is used, its application over all levels leads to a global effect which is almost sufficient by itself to recapitulate a full structure [1]. As the procedure was designed to correct defects caused by the addition of random motion (by **`shaker`**), it was not implemented as an independent parallel process but included in the thread along with **`shaker`** and applied after the coordinates had been displaced, so keeping a balance between disruption and correction.

`tinker` can be applied in two different modes depending on the setting of the global parameter `TINKER [update]`. In static mode, the starting distances and angles remain unchanged throughout the simulation, while in dynamic mode they are updated with a frequency set by `update`. With a value of N, **`tinker`** is called on every N^{th} cycle of the simulation. An update value less than zero (or no value) is taken to specify the static mode. If the `TINKER` command is not given, then **`tinker`** is skipped.

The overall effect of the **`tinker`** procedure is to provide a buffering effect against random motion and is similar to giving inertia to the structure but still allowing movement under a persistent "force". In the current implementation, this shift is by 1% once in roughly every 100 activations of **`tinker`**.

[1]Applied to the chemotaxis protein `3chy` (130 residues), if no helices get trapped on the wrong side of the sheet and with a few links between the terminal helices, the recapitulated structure can come as close as 5Å RMSD (over all CA atoms).

9.2 fixers

The routines collected here share the common feature that they are designed to maintain specialised geometries, usually associated with both protein and nucleic acid secondary structures. Many of them use the vector of distances contained in `cent`, the three components of which hold the distance of the child to it parent's centre and both end-points.

The **fixers** are all bundled into the same thread.

9.2.1 fixSSEaxis

For protein secondary structures (α and β), rather than calculate the inertial axes which is not only slow but also inaccurate for short α-helices, an axis was defined using the mid-points of positions $i-1$ and $i+1$. The axis direction is built-up by the addition of displacements from one mid-point to the next, as:

```
:
axis.zero();
was = cell->child[2]->xyz & cell->child[0]->xyz;
for (i=2; i<n-1; i++) { // +/- 1 line is close to axis for alpha and beta
        now = cell->child[i-1]->xyz & cell->child[i+1]->xyz;
        axis += now - was;
        was = now;
        in++;
}
:
```

The resulting axis is taken as the direction along which to place the new axis end-points at a distance calculated by what is expected from the number of residues in the SSE and their known rise per residue (with an approximation for loops) as:

```
:
// reset the end-points relative to the centre
if (sort==0) len = loopAXIS*size*sqrt(fn+1.0);
if (sort==1) len = alphAXIS*fn*bondCA/3.8;
if (sort==2) len = betaAXIS*fn*bondCA/3.8;
axis.setVec(len*0.5);
x = cell->xyz - axis;
shift = x - cell->endN;
cell->endN += shift*weight;
y = cell->xyz + axis;
shift = y - cell->endC;
cell->endC += shift*weight;
:
```

The values of `loopAXIS`, `alphAXIS` and `betaAXIS` are: 0.5, 1,5 and 3.0, respectively.

The positions of the children are then refined relative to the new end-points using the values in the vector `cent`. As these may be inconsistent, a consensus shift is calculated, weighted by the distance from each of the three points.

```
:
DO(i,n) { Cell *kidi = cell->child[i]; Vec dif; float rms1, rms2;
        dif.x = kidi->xyz|cell->endN;
        dif.y = kidi->xyz|cell->xyz;
        dif.z = kidi->xyz|cell->endC;
        dif -= kidi->cent; // difference between have and want (cent)
        rms1 = sqrt(dif*dif);
        shift.zero();
        shift += (cell->endN - kidi->xyz)*dif.x;
        shift += (cell->xyz  - kidi->xyz)*dif.y;
        shift += (cell->endC - kidi->xyz)*dif.x;
        shift *= 0.0;//weight/3;
        now = kidi->xyz + shift;
        dif.x = now|cell->endN;
        dif.y = now|cell->xyz;
        dif.z = now|cell->endC;
        dif -= kidi->cent;
        rms2 = sqrt(dif*dif);
        if (rms1-rms2 > NOISE) { // apply the shift
                kidi->xyz = now;
        }
}
:
```

The improvement in the new position is monitored as a difference in RMSD, but this should not be necessary.

9.2.2 `fixRNAaxis`

9.2.3 `fixRNAstem`

9.2.4 `fixDOMaxis`

9.2.5 `fixSheet`

9.3 `center`

The **`center`** routine was introduced previously as the process that couples the position of a parent to its children and except for **`keeper`** (which only acts on individual children) is the only routine to cross between levels of the hierarchy.

The code is simple enough to show in its entirety but there are a few aspects that will be discussed below.

```
void center ()
{
Cell    *world = Cell::world;
        DO(i,world->kids) { Cell *child = world->child[i];
                if (child->empty) continue;
                child->group();
        }
}

void Cell::group () {
        this->shifter( 0.5 );
        for (int i=0; i<kids; i++) child[i]->group();
}

void Cell::shifter ( const float couple )
{ // shift child centroid to cell centre by couple and cell to centroid by 1-couple
Vec     shift, s;
float   wkids = 0.0;
        if (kids==0) return;
        shift = xyz - this->getWcent();         // shift for children towards cell centre
        s = shift * couple;                     // scale by coupling constant
        DO(i,kids) moveCell(child[i],s,1);      // move children (and all their offspring) home
        s = shift * (1.0 - couple);             // scale by 1-couple
        xyz -= s; endN -= s; endC -= s;         // shift shell (parent and poles) towards children
}
```

The centroid of the children is calculated by the routine `getWcent()` which returns the centroid with each child weighted by the number of children it contains, plus 1. All the kids are then shifted along the line between the centroid and the centre of the parent by a factor `couple`, which has a built-in value of 0.5. The parent in turn is shifted towards the children by the complementary factor 1-`couple`.

9.4 `sorter`

For all objects above the atomic level, the **`viewer`** uses transparent rendering. Whereas normal solid objects are automatically sorted into and drawn from a Z-buffer, for some reason this is not done for transparent objects. As the view changes, three sorted buffers in X,Y and Z are maintained and the quadrant that is closest to the viewer is used to set the ranking. As **`SimGen`** is not striving for perfect rendering, these buffers are resorted only intermittently using a single pass of a bubble sort and in a complex field of objects, the occasional mis-layering of the objects is seldom noticed.

Irrespective of graphical rendering, the buffers also have uses in finding pairs of neighbouring objects.