Matriarch User's Guide

Written by:

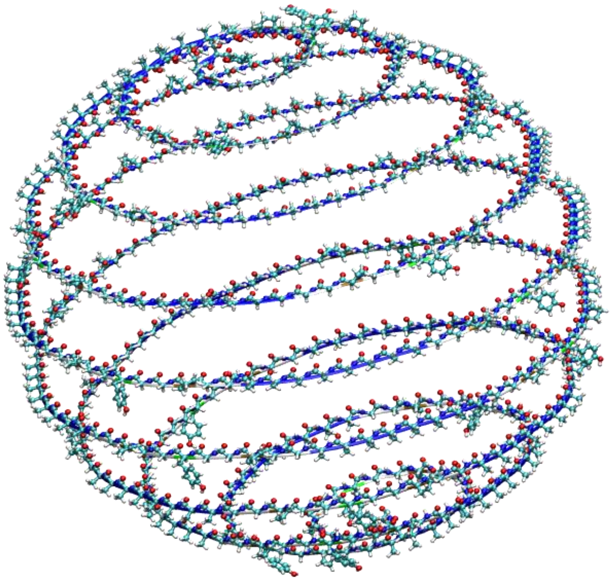
David I. Spivak, Tristan Giesa, Ravi Jagadeesan, and Markus J. Buehler

Programmed by:

Ravi Jagadeesan
Laboratory for Atomistic and Molecular Mechanics,
Department of Civil and Environmental Engineering,
Massachusetts Institute of Technology,
77 Massachusetts Avenue,
Cambridge, Massachusetts 02139, USA

If you use this program, please cite the journal article:

Giesa, T; Jagadeesan, R; Spivak, DI; and Buehler, MJ. (2014). “A Python Library for Materials Architecture.” *In submission.*



Software purpose

Matriarch is a high-level language of materials architecture, implemented in an open-source Python library. This language is based on the mathematical field of category theory, as discussed in the journal article, *A Python Library for Materials Architecture*. Matriarch creates material architectures for protein structures and can output them as atomic configurations, in the form of PDB (protein data bank) files. Using Matriarch, an engineer can substitute building blocks and vary building instructions to create and study new materials.

1 Table of contents

2 Download and install

You can download the Matriarch program library, free of charge, at the following website: http://web.mit.edu/matriarch/.

The Matriarch distribution contains three folders and two files:

- aminoAcids (folder)
- examples (folder)
- matriarch (folder)
- parseAminoAcids.py
- setup.py

Open a terminal and navigate to the directory in which you downloaded the Matriarch package. Run the command `python setup.py install`, as an administrator (e.g., using `sudo python setup.py install` on Linux and Mac, and with a command prompt as administrator on Windows). This requires the Python package distutils, which is usually preinstalled in your Python distribution.

Open a Python console (restart it if it is already running). You can now import the library using *either* of the following commands:

```
import matriarch
```

or

```
from matriarch import *
```

3 First things first

3.1 Basic concepts of Matriarch

The word *matriarch* is a portmanteau of “materials architecture”. Matriarch is a Python library for creating material architectures, i.e., building blocks that have been formed out of simpler building blocks, using a building instruction.

A *building block* is a construct that contains various sorts of information about its architecture, such as atomic coordinates and bonds. More details can be found in Section 4.2.1. The pre-

installed building blocks in Matriarch are the 20 standard amino acids, plus hydroxyproline. The user can add custom basic building blocks, as explained in Section 4.2.5.
A *building instruction* is a function with various arguments (existing building blocks) and parameters (real numbers, etc.), which creates a new building block from previously constructed ones. Thus a material architecture can be thought of as a tree, whose leaf nodes are basic building blocks, and whose non-leaf nodes are building instructions. The final product is a material building block, which can be exported as a PDB file. Building instructions are described in Section 4.3.

The whole framework described here is formalized using the mathematics of *operads*, which can model the assembly of hierarchical structures (see ref. [1] and [2]). Expert-level details about the mathematical framework for Matriarch can be found in the Mathematics Supplement.

3.2 Organization of the Matriarch package

The file *matriarch.py* contains the implementation of the core building instructions. Classes in matriarch.py are organized in accordance with the mathematical definition of material building blocks, as described in Section 4.2.1. The file *PDB_operations.py* contains the classes and functions that can read and write PDB files. These may be of independent use, for example to rotate polypeptides in PDB files and to reformat incorrectly structured PDB files (e.g., to add "TER" at the end of each polypeptide strand).

The file *vector.py* contains a lambda-calculus style implementation of basic linear algebra and rigid motions of Euclidean space to provide simple notation for *PDB_operations.py* and *matriarch.py*.

The *aminoAcids* folder contains the PDB files of the 20 standard amino acids, as well as hydroxyproline. The *Examples* folder contains several Python files that use Matriarch to output various material architectures, again as PDB files. These can be visualized, for example using VMD (visual molecular dynamics software).

4 Operation of the Matriarch program

In this section you will learn the basics of creating custom material architectures using Matriarch, and outputting them as PDB files. Simple example scripts will be written in blue boxes, each of which can be run directly in Python, once the library is installed as described in Section 2. All of these blue-box examples can be found in the Examples folder of the Matriarch installation.

4.1 Your first Matriarch program

Begin by launching Python. Inside the Python terminal, enter the following:

```
import matriarch
mySeq = 'AAAPPY'
myChain = matriarch.chain(mySeq)
myLongChain = matriarch.attachSeries(myChain,5)
myLongChain.fileOut('MyLongChain.pdb')
```

In this Matriarch program, we first create a string representing a sequence of six amino acids in letter codes: alanine, alanine, alanine, proline, proline, tyrosine. The command *chain* converts this list into a building block, called myChain. Using the building instruction *attachSeries,* this building block is attached to itself, end-to-end, five times to create a new building block, myLongChain. Finally, the building block MyLongChain is output as MyLongChain.pdb, a file in the working directory.

The result can be viewed using a simple text editor or a molecular visualization program, such as VMD. Opening MyLongChain in VMD will show a straight polypeptide chain with 30 residues.

4.2 Building blocks

The Matriarch program can be used to create highly complex proteins, each of which will be called a building block. The notion of building block is therefore itself fairly complex. It involves a number of amino acids, information about bonding, and information about the 3-dimensional shape. The next section makes this precise, but can be safely skipped on a first reading.

4.2.1 What is a building block?

Roughly speaking, a building block is a set of oriented rigid bodies that are bonded together, organized in space, and equipped with left and right interfaces that allow them to connect to other building blocks. A more precise description is given below. In the case of proteins, we treat amino acids as oriented rigid bodies, with orientation given by its amine (N) and carboxyl (C) terminals. Because many proteins have a persistence length of at least 4 Å, which is approximately the contour length of a single amino acid, the approximation that amino acids are rigid is reasonable.

For concreteness, we will speak of each oriented rigid body as an amino acid with one of its terminals (carboxyl or amine) chosen as its left end. We fix a set *AA* of *amino acid types*, each element of which is denoted by its 3-letter code, such as *ala* or *tyr*. Each element of AA has a contour length $cLen: AA \rightarrow \mathbb{R}$, calculated from its PDB file. There is also a fixed set $PTT = \{C, N\}$ with two elements, called *primitive terminal types*. Define $ORB = AA \times PPT$, and call it the set of *oriented rigid bodies* (or *ORBs*).

The second projection of an ORB is called the *left primitive terminal* function $LTerm: ORB \rightarrow PTT$; it returns the orientation of the ORB. Consider the *left-right swap* function $k: \{C, N\} \rightarrow \{C, N\}$ such that $k(C) = N$ and $k(N) = C$. Define $RTerm: ORB \rightarrow PTT$, called the *right primitive terminal function*, by $RTerm(x) = k\big(LTerm(x)\big)$. We will say two ORBs (x, y) are *bondable* if $RTerm(x) = k(LTerm(y))$, which is the same as saying the ORBs have the same left primitive terminal, $LTerm(x) = LTerm(y)$.

We introduce the concept of a *building block axis*. The building block axis is a pair of numbers (l, θ), where $l \in \mathbb{R}_{>0}$ and $0 \leq \theta < 2\pi$, signifying a portion of the nonnegative *z*-axis $[0, l]$, together with an angle θ, called the *clutch angle*, which indicates a direction in the *xy*-plane. Now we are ready for the formal definition of a building block.

A *building block* consists of:

1. a finite ordered set S, called the *enumerator* of the building block.
2. a *bond structure* on S, which consists of:
    a. a function $sig: S \rightarrow ORB$, called the *signature*. Together, the pair (S, sig) is called the *ORB sequence*.
    b. a set B, elements of which are called *bonds*, and a function $epts: B \rightarrow S \times S$, called the *endpoints* function. We put the further restriction that for every $b \in B$, its endpoints have bondable signatures (as defined above).
    c. an *interface structure*: two ordered subsets $L, R \subseteq S$, called the *left* and *right terminals* of the building block. The sets L and R are not required to be disjoint.
3. an *axis structure* on S, which consists of:
    a. a building block axis (l, θ) of some finite length $l \geq 0$ and clutch angle $\theta \in [0,2\pi)$,
    b. three functions $p, N_1, N_2: S \rightarrow \mathbb{R}^3$, such that N_1 and N_2 are orthogonal unit vectors. We call p the *point in space* of the amino acid, N_1 the *backbone direction*, and N_2 the *side-chain direction*.
    c. a function $\pi: S \rightarrow [0, l]$, called the *axis projection*.

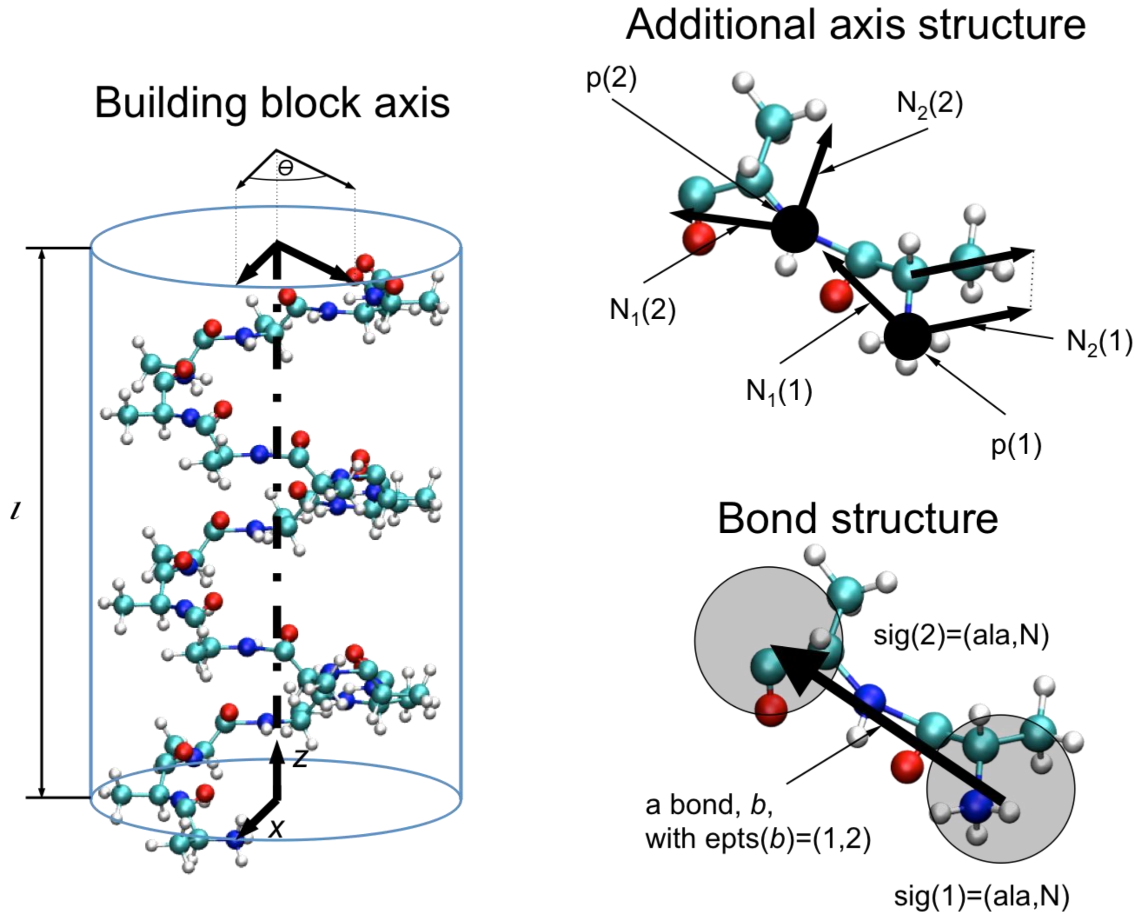


Figure 1

In Figure 1, the bond structure and axis structure of a helix with an ORB sequence consisting of 20 alanines, are shown for the following Matriarch code.

```
import matriarch
mySeqA = 'AAAAAAAAAAAAAAAAAAAA'
myChainA = matriarch.chain(mySeqA)
myHelix = matriarch.helix(myChainA, 4, 8, 'L')
myHelix.fileOut('myHelix.pdb')
```

The result is shown to the left in Figure 1, under the heading “Building block axis”. The helix winds around the *z*-axis. The amine group of the first amino acid in the sequence is placed on the *x*-axis, and the clutch angle θ points at the amine group of the hypothetical “next” amino acid. With these orientations, it will be possible to attach another helix *nextHelix* to *myHelix*, using the *attach* command. This command will automatically align the two by moving *nextHelix* with a rigid motion g, such that $\theta(myHelix) = g(1,0,0)$.

More of the axis structure is shown in Figure 1 under the heading “Additional axis structure”. This includes the points in space $p(i)$, the backbone direction $N_1(i)$, and the side chain directions $N_2(i)$ of the ORBs, for $i \in \{1,2,\ldots,20\}$. The axis projections $\pi(i)$ of the ORBs onto the line segment $[0, l]$ are not shown. The vector $N_1(i)$ is a unit vector pointing from the nitrogen atom of amino acid i to the nitrogen atom of amino acid $i + 1$. The backbone direction for the last amino acid in the sequence, in this case $N_1(20)$, is approximated from the position of the last carbon. For each i, the vector $N_2(i)$ points from the backbone to the side chain direction.

The bond structure of a building block is a network of ORBs and bonds, and a sample of it is shown in Figure 1 under the heading “Bond structure”. Each edge in the network is a bond $b \in B$, and its two endpoints $epts(b) \in S \times S$ indicate the ORBs that are being bonded. Each number $i \in \{1,2,\ldots,20\}$ has a signature $sig(i) = (aa_i, ptt_i)$, which is an ORB. Recall that an ORB consists of an amino acid name, such as Alanine, and a primitive terminal type (C or N). The primitive terminal type determines the orientation of the amino acid; for example if (Alanine, N) were replaced by (Alanine, C) the amino acid would be turned around. If a bond b has endpoints $epts(b) = (s_1, s_2)$, then the primitive terminal types of ORBs, $O_1 = sig(s_1) = (aa_1, ptt_1)$ and $O_2 = sig(s_2) = (aa_2, ptt_2)$, must be bondable, i.e., $Lterm(O_1) = ptt_1 = ptt_2 = Lterm(O_2)$.

4.2.2 Standard protein building blocks

Every oriented rigid body $(aa, ptt) \in ORB$ can be converted to a building block, which we call a *standard building block*, as follows.

1. **Enumerator**: S={1}
2. **Bond structure**: $sig(1) = (aa, ptt)$. There is one ORB in the sequence, and its signature is (aa, ptt). $B = \emptyset$ (no bonds), $L = R = \{1\}$. The left and right terminals are equal because the building block consists only of one amino acid, to which other building blocks can be attached on either side.
3. **Axis structure***:* $l = cLen(aa)$. The clutch angle is $\theta = 0$, i.e. pointing along the *x*-axis. The point in space of the amino acid is the amine end: $p(1) = (0,0,0)$ if $ptt = N$ and $p(1) = (0,0, cLen(aa))$ if $ptt = C$. The backbone direction is $N_1(1) = (0,0,1)$, the side-chain direction is chosen to be $N_2(1) = (1,0,0)$, and the axis projection is given by $\pi(1) = 0$.

Other than hypdroxyproline, all of the amino acid structures (see Table 1) were taken from the following website: http://wbiomed.curtin.edu.au/biochem/tutorials/pdb/. They were then processed through VMD's AutoPSF function. These include the standard 20, as well as hydroxyproline:

Table 1. Standard amino acids and their letter codes.

Amino acid (aa)	Standard abbreviation	Matriarch letter code
Alanine	ala	A
Arginine	arg	R
Asparagine	asn	N
Aspartic acid	asp	D
Cysteine	cys	C
Glutamine	gln	Q
Glutamic acid	glu	E
Glycine	gly	G
Histidine	his	H
Hydroxyproline	hyp	Z
Isoleucine	ile	I
Leucine	leu	L
Lysine	lys	K
Methionine	met	M
Phenylalanine	phe	F
Proline	pro	P
Serine	ser	S
Threonine	thr	T
Tryptophan	trp	W
Tyrosine	tyr	Y
Valine	val	V

Note that hydroxyproline should not be placed at an end of a polypeptide strand because the PDB file for hyp (version 1.0) does not have correctly placed hydrogen atoms at the carboxyl terminal.

4.2.3 Chains of standard building blocks

For any sequence of amino acids, say seq='AAAPPY', the command *chain(seq, ptt)* will create a building block. Here, ppt is an optional parameter, either 'N' or 'C', that allows the user to specify the left terminal of the ORBs. The default is 'N', amine.

For example, the standard building block, as discussed in Section 4.2.2, for the amino acid Alanine, is formed by the command *chain*('*A*'). The longer sequence seq is converted to the following building block by *chain('AAAPPY', 'N')*:

1. **Enumerator:** $S = \{1,2,\dots,6\}$.
2. **Bond structure:** $sig(1) = (ala, N), sig(2) = (ala, N), \dots, sig(6) = (tyr, N)$. Bonds $B = \{12, 23, 34, 45, 56\}$, with endpoints $epts(12) = (1,2)$, etc. The left terminal is $L = \{1\}$, and the right terminal is $R = \{6\}$.
3. **Axis Structure:** Length $l = \sum_{i=1}^{6} cLen\big(sig(i)\big)$, the clutch angle is $\theta = 0$, i.e. pointing along the *x*-axis. The point in space for each $j \in S$ is $p(j) = \left(0,0, \sum_{i=1}^{j-1} cLen(i)\right)$, the amino backbone direction is constant $N_1(j) = (0,0,1)$, the side-chain direction is constant $N_2(j) = (1,0,0)$, and the axis projection is given by $\pi(j) = \sum_{i=1}^{j-1} cLen(i)$.

Note that the function *chain* will automatically remove charges from intermediate amino acids, but leave the terminal (first and last) amino acids charged.

4.2.4 Examining a building block's structure

For a building block Γ, the command *print(Γ)* will output its building block structure, as defined in Section 4.2.1. (Note that this does not override Python's usual print command.) Here is an example.

```
import matriarch
mySeq = 'AAAPPY'
myChain = matriarch.chain(mySeq)
print(myChain)
myChain.fileOut('myChain.pdb')
```

```
>>> print(myChain)
BOND STRUCTURE
Type of amino acid 1: (ala,C)
Type of amino acid 2: (ala,C)
Type of amino acid 3: (ala,C)
Type of amino acid 4: (pro,C)
Type of amino acid 5: (pro,C)
Type of amino acid 6: (tyr,C)
Bonds: [[1, 2], [2, 3], [3, 4], [4, 5], [5, 6]]
Left end: [1]
Right end: [6]

AXIS STRUCTURE
Axis -- Length: 22.0512400171; Clutch angle: 0.0
Amino acid 1 -- Point: [0, 0, 0]; N1: [0, 0, 1]; N2: [1, 0, 0]; Projection: 0
Amino acid 2 -- Point: [0.0, 0.0, 3.694812428370368]; N1: [0.0, 0.0, 1.0]; N2: [1.0, 0.0, 0.0]; Projection: 3.69481242837
Amino acid 3 -- Point: [0.0, 0.0, 7.389624856740736]; N1: [0.0, 0.0, 1.0]; N2: [1.0, 0.0, 0.0]; Projection: 7.38962485674
Amino acid 4 -- Point: [0.0, 0.0, 11.084437285111104]; N1: [0.0, 0.0, 1.0]; N2: [1.0, 0.0, 0.0]; Projection: 11.0844372851
Amino acid 5 -- Point: [0.0, 0.0, 14.7330884733543]; N1: [0.0, 0.0, 1.0]; N2: [1.0, 0.0, 0.0]; Projection: 14.7330884734
Amino acid 6 -- Point: [0.0, 0.0, 18.381739661597496]; N1: [0.0, 0.0, 1.0]; N2: [1.0, 0.0, 0.0]; Projection: 18.3817396616
```

4.2.5 Adding custom amino acids

You can add custom amino acids to the existing library, the standard 20 plus hydroxyproline, using PDB files. The command is *userDefinedAminoAcid(filePath, symbol, abbrv)*, where *abbrv* is an optional string parameter.

For example, suppose you want to add Pyrrolysine to Matriarch. The steps for doing so are as follows:

1. Assign an abbreviation, such as *abbrv*='pyr'. This will be used for Matriarch's print command.
2. Assign it a symbol, which must be a character; for example choose *symbol*='&'.
3. Create a PDB file, and supply the path to Matriarch; for example *filePath*='C:/Users/matriarchUser/Desktop/pyrrolysine.pdb'. This file must satisfy the following constraints.
    a. The first line should be Nitrogen. In particular there should be no REMARKs or crystal sizes.
    b. The second, third, and fourth lines must be the three hydrogens in the amine group.
    c. The naming scheme should be the same as used in PDB files of the standard building blocks, (use N, HT1, HT2, HT3, C, CA, CB, OT1, and OT2). Matriarch will automatically adjust the orientation of the backbone and the side chain to the needs of the program.
4. Use the following command to import your custom amino acid

    ```
    matriarch.userDefinedAminoAcid('C:/Users/matriarchUser/Desktop/
    pyrrolysine.pdb', '&', 'pyr')
    ```

    This command has to be run only once per session, e.g., immediately after importing Matriarch.

Once you have added the custom amino acid, you can use *chain* to create a building block; see Section 4.2.3.

4.3 Building instructions

A building instruction always returns a building block. There are Matriarch commands that perform other useful functions; they are discussed in Section 4.4.

The simplest building instructions are the ones that take one building block as an argument, such as those that move it in $\mathbb{R}^3$. For example, one can apply the *moveORBs* command to a building block to move its atoms in space. These instructions are purely based in the geometry of $\mathbb{R}^3$.

Given a building block, one can *twist* it into a spiral, or more generally into an arbitrary shape, without deforming the individual rigid pieces that comprise the building block. For example, one can twist the amino acid sequence for a strand of tropocollagen into a spiral, and then twist the result into a helix to yield a strand of tropocollagen. *Padding* a building block by adding empty space at its right terminal is another instruction belonging to the same class. *Reversing* a building block mirrors the building block and puts its left terminal at the origin.

The instructions *attach*, *overlay*, and *space* are used to combine building blocks; each takes two building blocks as arguments. The *attach* instruction is straightforward; it bonds two building blocks together. One can also *overlay* blocks, placing them in the same space and combining their positive (resp. negative) terminals into a single composite positive (resp. negative) terminal. For example, one can *overlay* three helices at 120 degree angles offset from one another, and *twist* the result to obtain tropocollagen. *Spacing* two building blocks together places them next to each other with any given distance in between, without bonding the blocks to one another, and forgets any positive and negative terminals in the middle.

This section begins with a table that shows all the basic building instructions, including their arguments and their parameters. The effect of each building instruction on the bond structure and the axis structure of the building blocks is then made precise in the subsections that follow.

Table 2. Basic building instructions. Throughout, Γ represents a building block; all other variables are parameters. For fully formal specifications of these instructions, see the Matriarch Mathematics Supplement.

Instruction	Function
$moveOrbs(\Gamma, g)$	Moves the atoms in building block Γ by rigid motion g. See Section 4.3.1.
$shiftOrbs(\Gamma, s)$	Shifts the atoms of Γ by s units in the z-direction. See Section 4.3.1.
$rotateOrbs(\Gamma, \psi)$	Rotates the atoms of Γ by ψ (in degrees) around the z-axis. See Section 4.3.1.
$twist(\Gamma, W)$	Twists Γ by the axis twister W. See Section 4.3.2.
$helix(\Gamma, rad, pitch, handed)$	Twists Γ into a helix with radius rad, pitch length $pitch$, both in Å. The argument $handed$ must be 'L' or 'R', to choose the handed-ness of the helix. See Section 4.3.2.
$pad(\Gamma, s)$	Pads Γ with blank space of length s at its right terminal. See Section 4.3.3.
$attach(\Gamma_1, \Gamma_2)$	Attaches Γ_1 and Γ_2 in series. See Section 4.3.4.
$attachSeries(\Gamma, n)$	Attaches n copies of Γ in series. See Section 4.3.4.
$reverseOrbs(\Gamma)$	Reverses the direction of Γ. See Section 4.3.5.
$overlay(\Gamma_1, \Gamma_2)$	Overlays Γ_1 and Γ_2 in the same space. See Section 4.3.6.
$makeArray(\Gamma, n_x, n_y, s_x, s_y, alt)$	Places $n_x n_y$ copies of Γ into an $n_x \times n_y$ array, spaced at a distance s_x and s_y respectively, either antiparallel ($alt = True$) or parallel ($alt = False$). See Section 4.3.6.
$space(\Gamma_1, \Gamma_2, s)$	Spaces Γ_1 and Γ_2, with a distance s in between. See Section 4.3.7.
$spaceSeries(\Gamma, n, s)$	Places n copies of Γ in series, spaced at a distance s. See Section 4.3.7.

4.3.1 Move ORBs

The command $moveOrbs(\Gamma, g)$ moves the atoms of building block Γ in space by the transformation (rigid motion) g. For any building block Γ and transformation g, the ORB and bond structure of the output building block $moveOrbs(\Gamma, g)$ are the same as those of Γ. Its axis structure is obtained by applying g to the points in space, backbone directions, and side chain directions of Γ.

In Python, the form of g should be *[F,F']*, where *F* is a rigid motion of Euclidean space and *F'* is the derivative of *F*. Both *F* and *F'* should be written as functions.

For example, we could use $g = (F, F')$ where $F(x, y, z) = (z + 25, y, -x)$.

$$F' = \begin{pmatrix} \frac{\partial F_x}{\partial x} & \frac{\partial F_x}{\partial y} & \frac{\partial F_x}{\partial z} \\ \frac{\partial F_y}{\partial x} & \frac{\partial F_y}{\partial y} & \frac{\partial F_y}{\partial z} \\ \frac{\partial F_z}{\partial x} & \frac{\partial F_z}{\partial y} & \frac{\partial F_z}{\partial z} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} z \\ y \\ -x \end{pmatrix}.$$

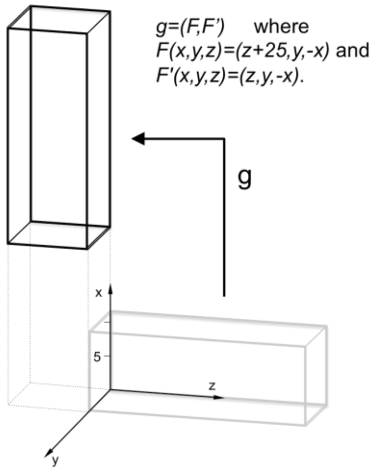


Figure 2

Note that when attaching building blocks, it is not necessary to use *moveOrbs* because the program automatically aligns building blocks correctly before attaching them.

```
import matriarch
mySeq = 'AAAPPY'
myChain = matriarch.chain(mySeq)

def T(x):
    return[-x[2], x[1],x[0]-25]

def TPrime(x):
    return[-x[2],x[1],x[0]]

g = [T,TPrime]
movedChain = matriarch.moveOrbs(myChain,g)
movedChain.fileOut(movedChain.pdb')
```

Shift ORBs: The command *shiftOrbs(Γ, s)* shifts the atoms of building block *Γ* by *s* units in the *z*-direction. The argument *s* can be negative. The bond structure of *shiftOrbs(Γ, s)* is the same as the bond structure of *Γ*.

Rotate ORBs: The command *rotateOrbs(Γ, ψ)* rotates the ORBs of Γ by ψ (in degrees) around the z-axis. The axis (including the clutch angle) does not change, nor does the axis projection. The bond structure of *rotateOrbs(Γ, ψ)* is the same as the bond structure of Γ. The point in space, the backbone direction, and the side-chain direction are rotated by the angle ψ about the z-axis. This instruction is useful to rotate building blocks before overlaying them, for example in the building instruction for tropocollagen.

4.3.2 Twist

The command *Γ' =twist(Γ,W)* is a two-part function. First, it deforms the building block Γ in space, by laying it along a chosen curve (which will be called W_{map}). Second, it defines a new axis for the output building block *Γ'* (which will be accomplished using the *total clutch* function Θ). The *twist* command changes the axis structure, but not the bond structure, of its input. Twist is the most complicated of the building instructions.

The *axis twister* is a pair $W \ = (W_{map}, \Theta)$. Here the *total clutch,* Θ: $[0, \infty) \to [0,2\pi)$ assigns an angle to each point on the positive z-axis. We will explain its function later.
$W_{map} = (f_{map}, f'_{map}, d_{map})$ is a *curved axis*, which includes three functions $f_{map}, f'_{map}, d_{map}$: $[0, \infty) \to \mathbb{R}^3$, called the *mapping curve,* the *mapping tangent,* and the *mapping direction,* respectively. Here f_{map} is the curve on which the axis of the original building block will be laid, f'_{map} is the unit tangent vector to f_{map}, and d_{map} is a unit vector orthogonal to f'_{map}.

Again, the positive z-axis of Γ will be laid on f_{map}. The points in space of the new building block will be obtained by applying the rigid motion T_t, which moves $(0,0,t)$ to $f_{map}(t)$, and rotates the positive z-axis in line with $f'_{map}(t)$. It also rotates the positive x-axis in line with $d_{map}(t)$, as shown in Figure 3. This is necessary to determine the locations of the amino acids in the twisted structure: for each ORB $s \in S$, we will apply the rigid motion $T_{\pi(s)}$ to each of $p(s), N_1(s)$, and $N_2(s)$.

A new length and clutch angle for the twisted building block needs to be computed. There is a slight difference between the math and the code here, because it can be useful to allow the user to create non-legal building blocks, for example with negative length. Let p_z: $\mathbb{R}^3 \to \mathbb{R}$ denote projection onto the z-axis. The new length and angle are given by $l' = p_z\left(f_{map}(l)\right)$, and let $\theta' = \Theta(l')$. Because the final length is not known ahead of time, the function $\Theta(z_0)$ specifies the expected direction of the clutch angle at all points $z_0 \in \mathbb{R}_{>0}$; the evaluation at l' gives the actual value for the clutch angle.

Note that there are two different kinds of axis-like structures that appear in Matriarch. The most important is the building block axis, (l, θ), which is always positioned on the positive z-axis; this is defined in Section 4.2.1. The other axis-like structure is only used when twisting. It is the curved axis $(f_{map}, f'_{map}, d_{map})$, which determines the family of rigid motions T_t.

If f_{map} is not parameterized by arc length, Matriarch will stretch or compress bonds when twisting; this is rarely desirable. To avoid this, there are useful functions, *buildAxisTwister* and *smoothedPiecewiseLinear*, described in Sections 4.4.2 and 4.4.3, respectively. These create an axis twister W for you, including the arc length reparameterization, either from an arbitrarily parameterized curve, or from a sequence of points.

We now give some example code and the corresponding figure, to explain the above remarks.

```
import matriarch
mySeq = 'AAAPPY'
myChain = matriarch.chain(mySeq)
def parabola(t):
     return [-t*t, 0, 10*t]

Rout = matriarch.Ray([18,0,0],[0,0,0])
W = matriarch.buildAxisTwister(parabola, Rout)
twistedBlock = matriarch.twist(myChain, W)
matriarch.fileOut(twistedBlock,'twistedBlock.pdb')
```

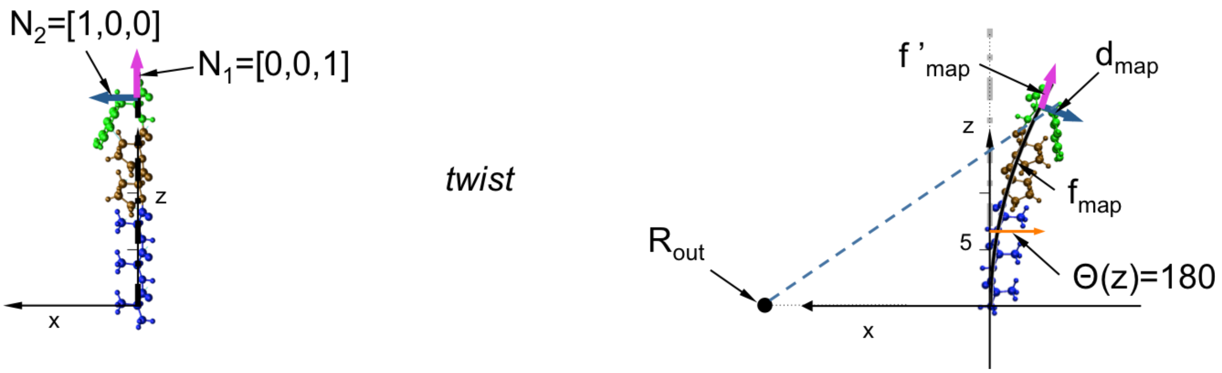


Figure 3

The purpose of Figure 3 is to explain the *twist* command, in a simple case. Consider the left side of Figure 3. We begin with a chain called *myChain* (whose axis is, as always, aligned with the z-axis. The pink vector indicates the backbone directions $N_1(s) = [0,0,1]$. The blue vector indicates the side chain direction $N_2(s) = [1,0,0]$.

The twisting process is performed by using a certain family of transformations $T_{\pi(s)}$, one to each ORB s. The transformation $T_{\pi(s)}$ is a complicated function, and does not need to be directly specified by the user. Instead, it is internally determined by the parameters passed to the *twist* command, W, which in our case is provided by *buildAxisTwister* (see Section 4.4.2). This transformation includes both a translation of the z-axis onto f_{map} and a rotation that is determined by d_{map}. The *buildAxisTwister* command defines $f_{map}$by reparameterizing of g_{map}, which is a user input. In our case, $g_{map}$is given by the "parabola" function. Furthermore, the *buildAxisTwister* command defines $d_{map}(t)$, for any $t \in [0, t_{max}]$, to be the unique unit vector orthogonal to $f'_{map}(t)$, coplanar with the line from R_{out} to $f_{map}(t)$, and pointing away from R_{out} (forming an angle of more than $90°$). In our case R_{out} is specified as [18,0,0], which makes d_{map} orthogonal to f'_{map}, pointing in the negative x-direction. Using f_{map}, d_{map}, and $f'{}_{map}$, we obtain the final point in space, amino backbone direction, and side chain direction of each amino acid s. In the specific example of Figure 3, the rigid motion $T_{\pi(s)}$ sends the pink vector [0,0,1] to $f'_{map}\big(\pi(s)\big)$ and the blue vector [1,0,0] to $d_{map}\big(\pi(s)\big)$.

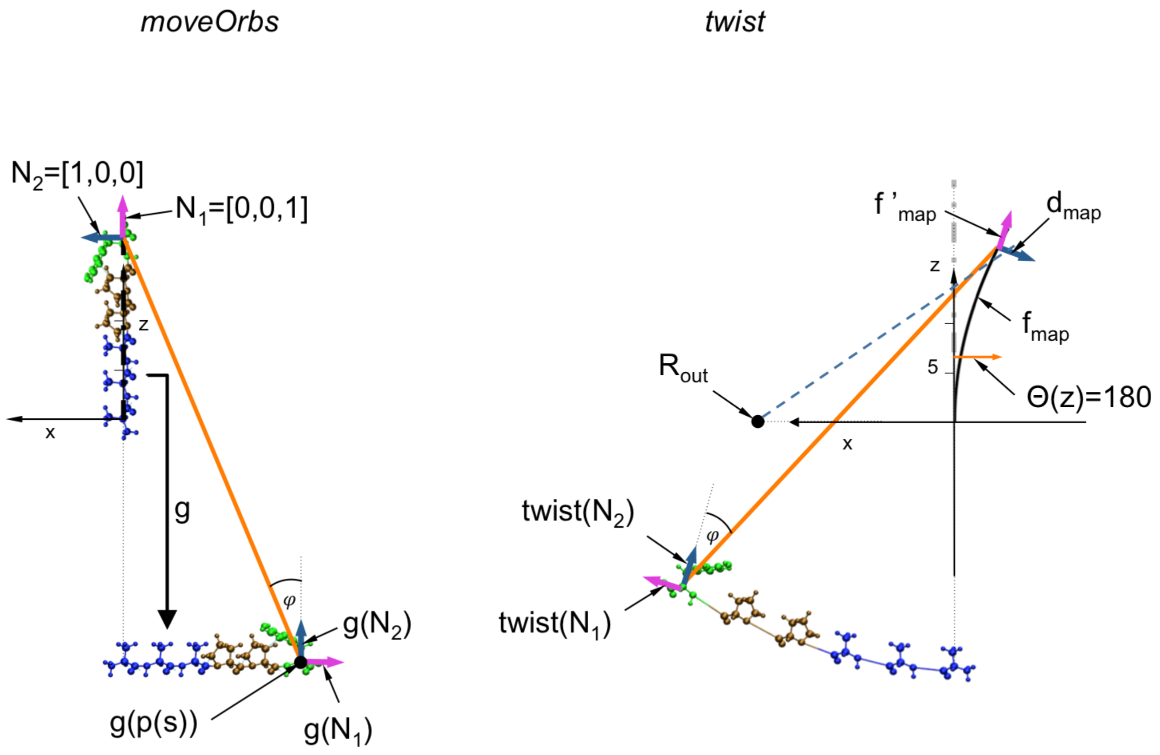
It remains to provide the clutch direction of the new building block, which will always be $\Theta(l')$, where l' is the new length discussed above. Recall that the total clutch, $\Theta\colon [0, \infty) \to [0{,}2\pi)$, is a component of the axis twister W, input to *twist*. The command *buildAxisTwister* generally computes the function $\Theta(z_0)$ for you. However, you can specify the optional parameter $\Theta_{spec} \in [0{,}2\pi)$, in which case $\Theta(z_0) = \Theta_{spec}$ for all $z_0 \in (0, \infty)$, so the clutch angle will be $\theta = \Theta_{spec}$. In the example of Figure 3, Θ_{spec} is not supplied, so the total clutch Θ is determined by the algorithm discussed in Section 4.4.2. The precise definition of Θ is given in Section 4.4.2, but roughly $\Theta(l')$ is the angle, with respect to the x-axis, of the last ORB (projected onto the xy-plane).

```
import matriarch
mySeq = 'AAAPPY'
myChain = matriarch.chain(mySeq)
def F(x):
     return [-x[2], x[1], x[0]-25]

def FPrime(x):
     return [-x[2], x[1], x[0]]

g = [F, FPrime]
myChain_newAxis = matriarch.moveOrbs(myChain, g)
def parabola(t):
     return [-t*t, 0, 10*t]

Rout = matriarch.Ray([18,0,0],[0,0,0])
W = matriarch.buildAxisTwister(parabola,Rout)
warpedBlock = matriarch.twist(myChain_newAxis, W)
matriarch.fileOut(warpedBlock,'warpedBlock.pdb')
```


moveOrbs

twist

Figure 4

The purpose of Figure 4 is to explain the inner workings of the *twist* command, by performing some irregular transformations at the outset. Beginning with a chain called *myChain* (whose axis is, as always, aligned with the *z*-axis), we apply *moveOrbs(myChain,g)* to rotate and move the atoms “downwards”. Consider the orange line in the left-hand side of Figure 4. Given an amino acid $s \in S$, the orange line indicates the connection between the old and new points in space $p(s) = \big(0,0,\pi(s)\big)$ and $g\big(p(s)\big)$. The pink vectors indicate the relationship between the old and new backbone directions $N_1(s) = [0,0,1]$ and $g(N_1(s))$. The blue vectors indicate the relationship between the old and new side chain directions $N_2(s) = [1,0,0]$ and $g(N_2(s))$. Note that, in general, we will not have that $p(s) = \big(0,0,\pi(s)\big)$ nor $N_1(s) = [0,0,1]$ nor $N_2(s) = [1,0,0]$; they are true in this case because *myChain* is a chain. The transformation g has moved the ORBs without changing the axis. You can think of the orange line, pink vector, and blue vector as together forming a rigid connector, which will be used by twist to determine the final position and side-chain direction of the amino acid *s*.

Specifically, *twist* applies $T_{\pi(s)}$ to the line segment connecting the ORBs to their projection onto the *z*-axis, and on this line segment it acts like a rigid motion. In the case of Figure 4, that line segment had length 0, so we spoke as though we were applying $T_{\pi(s)}$ to each ORB, *s* itself. In this case, the twisting process is performed by applying $T_{\pi(s)}$ as a family of rigid motions on the orange lines (one for each ORB), and the associated pink and blue vectors. This gives the final point in space, amino backbone direction, and side chain direction of each amino acid s. As a family of rigid motions, $T_{\pi(s)}$ sends the pink vector [0,0,1] to $f'_{map}\big(\pi(s)\big)$ and the blue vector [1,0,0] to $d_{map}\big(\pi(s)\big)$.

There is a strong similarity between Figure 4 and 5, deriving from the fact that they use the same axis twister W. Thus the values, as well as the derivation, of the mapping direction d_{map} and the total clutch Θ are identical in both cases.

Note that because we began by moving the ORBs, the bonds in the backbone between the amino acids become stretched non-physically. If the axis remains on the backbone (as in Figure 4), neither stretching nor compressing will occur. In ordinary cases, even if the axis is not directly on the backbone, the stretching effect should not be noticeable.

The example shown in the Figure 5 is rather exotic, but it illustrates the twisting mechanism. More common would be a helix (or triple helix). We will also show how to build the spherical spiral, seen on the front page, in Section 4.5.3.

Helix

Helix(x, rad, pitch, handed) is a special case of twist, for which the axis twister *W* has been built into Matriarch. The argument *handed* is optional, and defaults to right-handed, ‘R’. Define

$sign$ to be 1 if *handed* is 'R' and -1 if *handed* is 'L' and define $scale = \sqrt{rad^2 + 4\pi^2\,pitch^2}$. Then

$$f_{map}(t) = \left(rad * \cos\left(\frac{sign * t}{scale}\right), rad * \sin\left(\frac{sign * t}{scale}\right), \frac{pitch * t}{2\pi\, scale}\right),$$

$$d_{map}(t) = \left(\cos\left(\frac{sign * t}{scale}\right), \sin\left(\frac{sign * t}{scale}\right)\right),$$

$$\Theta(t) = \frac{sign * t * 2\pi}{pitch}.$$

The code that produces Figure 1 is below.

```
import matriarch
mySeqA = 'AAAAAAAAAAAAAAAAAAAA'
myChainA = matriarch.chain(mySeqA)
myHelix = matriarch.helix(myChainA, 4, 8, 'L')
myHelix.fileout('myHelix.pdb')
```

Expanding the *helix* command, the code above is equivalent to the following:

```
import matriarch
from math import *
mySeqA = 'AAAAAAAAAAAAAAAAAAAA'
myChainA = matriarch.chain(mySeqA)
def parameterizedHelix(t):
    return [4*cos(2*pi*t),-4*sin(2*pi*t),8*t]

W = matriarch.buildAxisTwister(parameterizedHelix)
myHelix = matriarch.twist(myChainA, W)
matriarch.fileOut(myHelix, 'helixEg.pdb')
```

4.3.3 Pad

The command *pad(Γ,s)* adds *s* Angstrom of blank space to the right terminal of building block *Γ*. The bond structure of *Γ'=pad(Γ,s)* is the same as the bond structure of *Γ*. The axis structure of *pad(Γ,s)* is obtained by increasing the length by *s*, i.e., $l' = l + s$. Everything else is the same for *Γ* and *Γ'*.

```
from matriarch import *
from math import *
mySeqA = 'AAAAAA'
myChainA = chain(mySeqA)
myPadded = pad(myChainA, 3)
fileOut(myPadded, 'myPadded.pdb')
```

4.3.4 Attach

The command *attach*(Γ_1, Γ_2) attaches two bondable building blocks together to create a composite building block. The *attach* command moves the left terminal of Γ_2 next to the right terminal of Γ_1. It then forms the appropriate bonds between the ORBs in the right terminal of Γ_1 and the ORBs in the left terminal of Γ_2. In order for the command *attach*(Γ_1, Γ_2) to work, the ORBs of Γ_1 at its right terminal must be bondable (see Section 4.2.1) to the ORBs of Γ_2 at its left terminal: there is a one-to-one correspondence $\phi: R_1 \xrightarrow{\sim} L_2$.

The new building block $\Gamma' = attach(\Gamma_1, \Gamma_2)$ is defined as follows. Its ORB sequence is given by

$$S' = S_1 \cup S_2 \quad \text{and} \quad sig'(s) = \begin{cases} sig_1(s), \text{if } s \in S_1 \\ sig_2(s), \ \text{if } s \in S_2 \end{cases}$$

Its bond structure is given as follows. The set of bonds is $B' = B_1 \cup B_2 \cup B_{12}$, where we put $B_{12} = R_1$; this set has the right number of new bonds to be created. The endpoints function $epts': B' \to S' \times S'$ is given on $b \in B'$ by

$$epts'(b) = \begin{cases} epts_1(b), \text{if } b \in B_1 \\ epts_2(b), \text{if } b \in B_2 \\ \big(b, \phi(b)\big), \text{if } b \in B_{12} \end{cases}$$

The interface structure is given by $L' = L_1$, $R' = R_2$.

The axis structure on Γ' is defined as follows. The building block axis is straightforward: $l' = l_1 + l_2$ and $\theta' = \theta_2$. The points in space is given piecewise:

$$p'(s) = \begin{cases} p_1(s), & s \in S_1 \\ p_2(s) + l_1, & s \in S_2 \end{cases}$$

The amino acid direction and side-chain direction are given by union. Finally, the axis projection is the union of the two axis projections, after composing them with the interval inclusions $[0, l_1] \subseteq [0, l']$ and $[0, l_2] \cong [l_1, l'] \subseteq [0, l']$.

For an example of this, see Figure 5 below.

Attach series

The command *attachSeries(Γ, n)* applies the *attach* command consecutively, *n* times in the *z*-direction, on the same building block Γ. This will only work if the left and right terminals of Γ are bondable.

4.3.5 Reverse Orbs

The command *reverseOrbs(Γ)* does what one would expect: it reverses the building block. This command is useful in a few contexts. For example, if you want to connect two chains, coming from two different functions, the orientation of one chain may need to be *reversed*; for example, see Figure 5 below. The *reverse* command is also useful in the construction of antiparallel polypeptides. It is used in *makeArray* in combination with *overlay*.

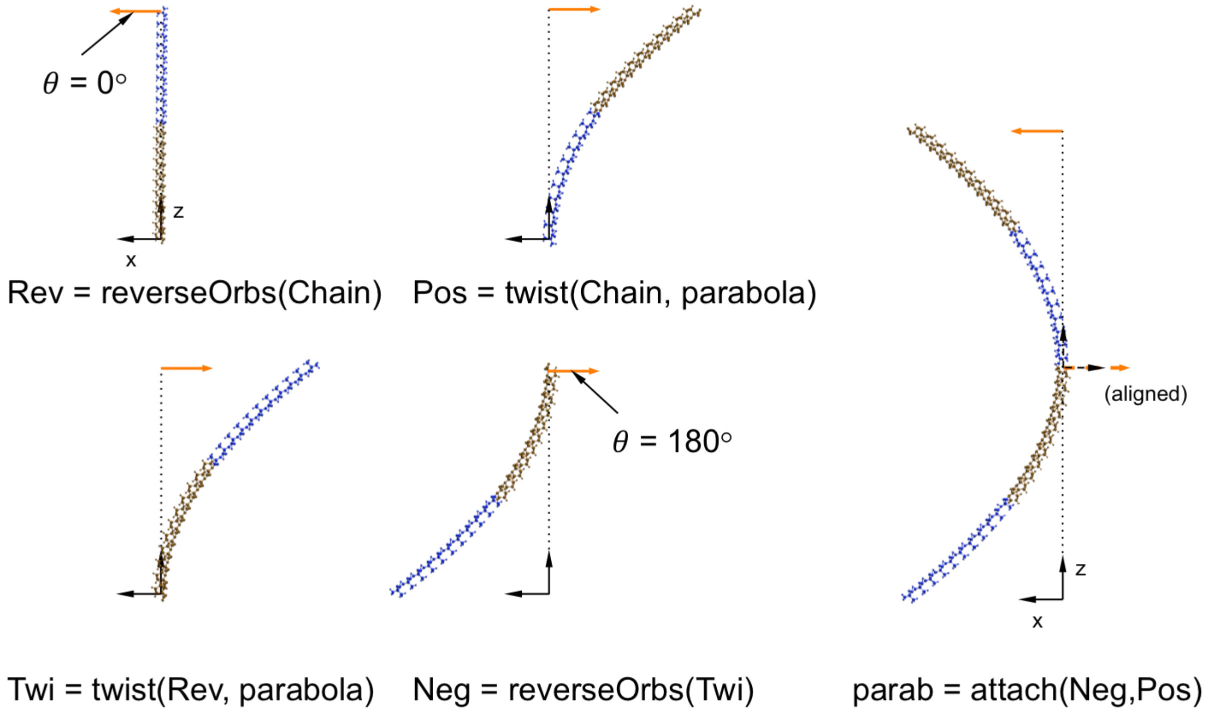
The full description of *Γ'=reverseOrbs (Γ)* is as follows. Let S' be S, but with the order reversed, i.e., $i \leq j$ in S' if and only if $i \geq j$ in S. The signature is given by $sig'(s) = k_{ORB} \circ sig(s)$, where $k_{ORB} = (\mathrm{id}_{AA}, k): ORB \to ORB$ is the left-right swap, i.e., it switches C and N. See Section 4.2.1. It has the same set of bonds, $B' = B$, but the endpoints are swapped: for $b \in B$ if $epts(b) = (s, t)$ then $epts'(b) = (t, s)$. The left and right terminals are also swapped: $L' = R$ and $R' = L$. The length and clutch angles of the axis are unchanged: $l' = l$ and $\theta' = \theta$. Let $T: \mathbb{R}^3 \to \mathbb{R}^3$ be the rigid motion of rotation about the line $\left(x, 0, \frac{l}{2}\right)$ followed by rotation by angle θ, counterclockwise about the positive z-axis. This transformation swaps the left and right terminals $(0,0,0)$ and $(0,0,l)$, and swaps the positive x-axis with the clutch angle of $(\cos\theta, \sin\theta, 0)$. The new point in space p', amino backbone direction N_1', and side-chain directions N_2' are given by composing p, N_1, N_2 with T, respectively. The axis projection is reversed: $\pi'(s) = l - \pi(s)$.

The intuitive idea behind *reverseOrbs(Γ)* is that it rotates the building block Γ about a certain line L lying in the plane $z = l/2$. The angle in the *xy*-plane, between L and the *x*-axis, is $\theta/2$, where θ is the clutch angle of Γ.

```
from matriarch import *

def parabola(t):
    return [-t*t,0,10*t]

mySeq = 'AAAAAAAAAAAPPPPPPPPPPP'
myChain = chain(mySeq)
Rout = Ray([-1,0,0], [0,0,0])
W = buildAxisTwister(parabola,Rout)
pos = twist(myChain,W)
rev = reverseOrbs(myChain)
twi = twist(rev,W)
neg = reverseOrbs(twi)
t = attach(neg,pos)
fileOut(t, 'parab.pdb')
fileOut(pos, 'parab-pos.pdb')
fileOut(rev, 'parab-rev.pdb')
fileOut(twi, 'parab-twi.pdb')
fileOut(neg, 'parab-neg.pdb')
```



Rev = reverseOrbs(Chain)

Pos = twist(Chain, parabola)

Twi = twist(Rev, parabola)

Neg = reverseOrbs(Twi)

parab = attach(Neg,Pos)

Figure 5

4.3.6 Overlay

The command *overlay(Γ_1, Γ_2)* puts the building blocks Γ_1 and Γ_2 together by leaving each ORB, in both Γ_1 and Γ_2 in the same position and adding no new bonds, as we explain below.

Let $\Gamma' = overlay(\Gamma_1, \Gamma_2)$. The enumerator of Γ' is the ordered union $S' = S_1 \cup S_2$, so if $S_1 = \{1,2, \ldots m\}$ and $S_2 = \{1,2, \ldots, n\}$, then $S' = \{1,2, \ldots, m+n\}$. The bond structure is given by the union; e.g., $sig' = sig_1 \cup sig_2$, and same for bonds, endpoints, and terminals. The axis structure is given as follows. The length of the new building block is given by the maximum, $l' = \max(l_1, l_2)$; the clutch angle is given by $\theta' = \theta_1$. The point in space, backbone direction, and side-chain direction are given by union. Finally, the axis projection is the union of the two axis projections, after composing them with the interval inclusions $[0, l_1] \subseteq [0, l']$ and $[0, l_2] \subseteq [0, l']$.

Collagen has two amino acid sequences, each coiled into a left-handed helix with radius 1.5 Å and pitch 9.5238 Å. These will be twisted together into a triple helix as follows. Both helices are twisted into the shape of another left-handed helix with radius 4 Å and pitch 85.5 Å. These two are offset from one another by rotating (by $2\pi/3$ and $4\pi/3$) and shifting (by 2.8 Å and 5.6 Å), and then they are overlaid with the first helix to produce the final collagen output. See Figure 6.

```
from matriarch import *
import math

def collagen(seq1, seq2):
    a1 = chain(seq1)
    a2 = chain(seq2)
    hel1 = helix(a1,1.5,9.5238,'L')
    hel2 = helix(a2,1.5,9.5238,'L')
    helhel1 = helix(hel1,4,85.5,'L')
    helhel2 = helix(hel2,4,85.5,'L')
    helhel1rot = shiftOrbs(rotateOrbs(helhel1,2*math.pi/3),2.8)
    helhel2rot = shiftOrbs(rotateOrbs(helhel2,4*math.pi/3),5.6)
    homodimer = overlay(helhel1,helhel1rot)
    output = overlay(homodimer,helhel2rot)
    return output

seq1 = 'GFZGPKGTAGEZGKAGERGVZGPZGAVGPAGKDGEAGAQGAZGPAGPAGERGEQGPA'
seq2 = 'GFZGPKGPSGDZGKZGEKGHPGLAGARGAZGPDGNNGAQGPZGPQGVQGGKGEQGPA'
collgn = collagen(seq1,seq2)
fileOut(collgn,'collgn.pdb')
```

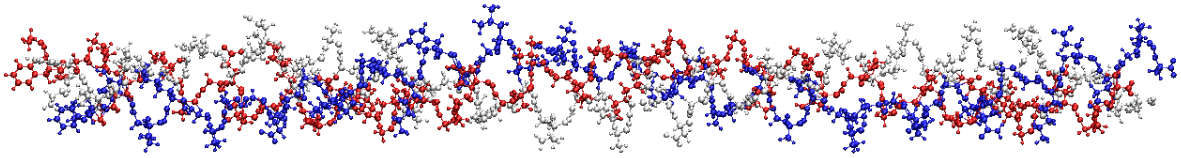


Figure 6

Make Array

The command *makeArray(Γ, nx, ny, sx, sy, alt)* copies Γ as an $n_x \times n_y$ stack (n_x copies in the *x*-direction and n_y copies in the *y*-direction), spaced apart by distance s_x in the *x*-direction and s_y in the *y*-direction. Each copy of building block in the resulting array is parallel to Γ, if the Boolean variable *alt* is set to *alt=False*, and the copies alternate if *alt=True*. Said another way, the *(i,j)*th copy is parallel if *i+j* is even, and is antiparallel if *i+j* is odd.

The *makeArray* command is a composition of previously-defined commands: *moveOrbs*, *reverse*, and *overlay*, as described above. For example, the *(i,j)*-entry will involve the command $moveOrbs(\Gamma, [F, F'])$ where $F = \left[x + i\ s_x, y + j\ s_y, z\right]$, and $F' = [x, y, z]$.

See Figure 7 below.

4.3.7 Space

The command *space(*Γ_1*,* Γ_2*, s)* leaves Γ_1 where it is and moves the left terminal of Γ_2 to a distance of d from the right terminal of Γ_1, as we explain below.

Let $\Gamma' = space(\Gamma_1, \Gamma_2, s)$. The enumerator is the ordered union $S' = S_1 \cup S_2$, so if $S_1 = \{1,2, \ldots m\}$ and $S_2 = \{1,2, \ldots, n\}$, then $S' = \{1,2, \ldots, m+n\}$. Most of the bond structure is given by the union; e.g., $sig' = sig_1 \cup sig_2$, $B' = B_1 \cup B_2$, $epts' = epts_1 \cup epts_2$. However, the terminals are a bit different: $L' = L_1$ and $R' = R_2$. The axis structure is as follows. The building block axis is straightforward: $l' = l_1 + l_2 + s$ and $\theta' = \theta_2$. The points in space is given piecewise:

$$p'(a) = \begin{cases} p_1(a), & a \in S_1 \\ p_2(a) + l_1 + s, & a \in S_2 \end{cases}$$

The amino acid direction and side-chain direction are given by union. Finally, the axis projection is the union of the two axis projections, after composing them with the interval inclusions

$[0, l_1] \subseteq [0, l']$ and $[0, l_2] \cong [l_1 + s, l'] \subseteq [0, l']$.

```
from matriarch import *

mySeqG = 'GGGGGGGGG'
mySeqA = 'AAA'
myChainG = chain(mySeqG)
myChainA = chain(mySeqA)
spacedBlock = space(myChainG, myChainA, 10)
array = makeArray(spacedBlock, 10, 15, 4, 6, True)
fileOut(array, 'arraySpaced.pdb')
```

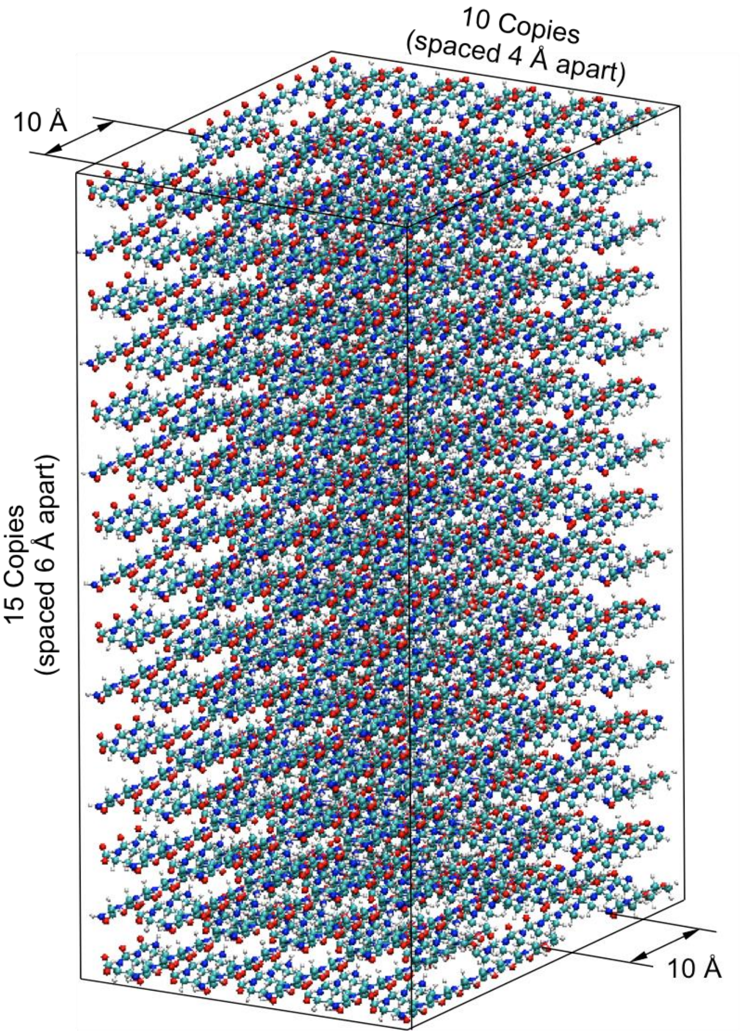


Figure 7

Space series

The command *spaceSeries(Γ, n, s)* applies the *space* command consecutively, *n* times in the *z*-direction, on the same building block *Γ*.

4.4 Other Matriarch functions

Table 3. Other useful functions. Throughout, *Γ* represents a building block; all other variables are parameters.

Instruction	Function
$fileout(\Gamma, loc)$	Saves the building block Γ to a PDB file at location *loc*. See Section 4.4.1.
$chain(seq)$	Returns a building block constructed by attaching the sequence of amino acids given by *seq*. See Section 4.2.3.
$buildAxisTwister($ $g_{map}, R_{out}, \Theta_{spec}, t_{max})$	Creates an axis twister by reparameterizing the mapping curve *gmap* according to arclength to avoid stretching or compressing bonds. *Rout*, *Thetaspec*, and *tmax* are optional parameters, described in Section 4.4.2.
$smoothedPieceWiseLinear($ $p_{List}, R_{out}, \Theta_{spec},$ $smthFactor, n_{pts})$	Creates an axis twister from a sequence of points p_{List}, by making a piecewise linear curve and then smoothing it. There are several optional parameters: a ray R_{out}, a specified constant total clutch Θ_{spec}, a smoothing factor between 0 and 1, and n_{pts} which controls the precision. See Section 4.4.3.
$userDefinedAminoAcid($ $symbol, threeLetterCode)$	You can add custom amino acids to the existing library, the standard 20 plus hydroxyproline, using PDB files. See Section 4.2.5.
$length(\Gamma)$	Returns the length l of the axis of the building block □. See Section 4.2.1.
$print(\Gamma)$	Displays the building block structure (enumerator, bonds, axis) of Γ. See Section 4.2.4.

4.4.1 File out

The command *fileout(Γ, loc)* saves the building block Γ to a PDB file at location *loc*.

4.4.2 Build axis twister

Recall that an axis twister, $W \ = \left(\left(f_{map}, f'_{map}, d_{map}\right), \Theta\right)$ includes a mapping curve f_{map} as one of its parameters. If f_{map} is not parameterized by arc length, Matriarch will stretch or compress bonds when twisting; this is rarely desirable. The usual constructor for axis twisters *W,* which automatically parameterizes f_{map} by arc length, is the command *buildAxisTwister(gmap,Rout,Thetaspec,tmax)*. The function g_{map}: $[0, t_{max}] \rightarrow \mathbb{R}^3$ is automatically reparameterized by arc length and smoothed to make the mapping curve f_{map}.

The following may seem complicated, but it is necessary for building complex twisted structures in 3D. For sufficiently simple 2D structures, you do not need to specify or understand R_{out} and t_{max}; they are optional parameters and can be just left as default values. There are predefined twisted structures, such as helices, that can be used without understanding the following details. The implementation of the *helix* command is shown below.

For the command *buildAxisTwister(gmap,Rout,thetaSpec,tmax)*, the ray $R_{out} = (R_{out,0}, R_{out,tng})$ is used to compute the mapping direction d_{map}, as explained below. The ray R_{out} is defined by a point and a tangent vector. By default, the length t_{max} is set to 1,000 and $R_{out} = (R_{out,0}, R_{out,tng})$ defaults to the positive z-axis, $R_{out,0} = (0,0,0), R_{out,tng} = (0,0,1)$. It is also ok for R_{out} to be a point (i.e., with $R_{out,tng} = (0,0,0)$), as in the default case. The *buildAxisTwister* command defines $d_{map}(t)$, for any $t \in [0, t_{max}]$, to be the unique unit vector orthogonal to $f'_{map}(t)$, coplanar with the line from R_{out} to $f_{map}(t)$, and pointing away from R_{out} (forming an angle of more than 90°).

If the optional parameter Θ_{spec} is set to a non-default value, it must be set to a constant in $[0,2\pi)$. In this case the total clutch is set to the constant function $\Theta(z_0) = \Theta_{spec}$. This will eventually be the new clutch angle, $\theta' = \Theta(l') = \Theta_{spec}$.

If the default value $\Theta_{spec} = [\,]$ is used, then Matriarch will compute the total clutch Θ from f_{map} as follows. First, for each $z_0 \in \mathbb{R}_{>0}$, we define a real number $S(z_0)$ as follows

$$S(z_0) = \max\left(\{0\} \cup f_{map}^{-1}\left(p_z^{-1}\left((-\infty, z_0]\right)\right)\right)$$

Here p_z is the projection of $\mathbb{R}^3$ onto the z-axis. Then the total clutch is given by

$$\Theta(z_0) = \psi\left(f_{map}\big(S(z_0)\big)\right)$$

Here ψ is a function from (the complement of the z-axis in) $\mathbb{R}^3$ to the interval $[0,2\pi)$. It takes a point (x_0, y_0, z_0) to the angle of $[x_0, y_0] \neq [0,0]$ with the x-axis. The idea is that for each height z_0, the set of ORBs of height at most z_0 is computed, and the angle of the latest one (i.e., of largest t) is taken as the total clutch $\Theta(z_0)$ at this height.

All the optional parameters are specified in the code for the spherical spiral cover art; see Section 4.5.3.

4.4.3 Smoothed Piecewise Linear

It is sometimes inconvenient to provide g_{map} in functional form. Like *buildAxisTwister*, the command *smoothedPiecewiseLinear(pList, Rout, thetaSpec, smthFactor, nPts)* produces axis twisters *W*, but instead of a function g_{map}, the user provides a list of points $p_{List} = [q_1, q_2, \dots, q_n]$, where each $q_i \in \mathbb{R}^3$. The command will first create a piecewise linear function from the sequence of points, and then smooth it using an optional parameter that is by default set to 1/3.

Again, there are several optional parameters. The ray R_{out} (see Section 4.3.2) has default value $([0,0,0], [0,0,1])$. The specified clutch angle Θ_{spec} is by default [], which causes the total clutch to be calculated, as discussed above (see Section 4.4.2). The smoothing factor is by default 1/3; it should be a number between 0 and 1. When smthFactor=0, the resulting curve will be an unsmoothed piecewise linear curve. In general, Matriarch fits a circle around the corners of the piecewise linear curve. The radii of these circles depend on the smoothing factor and the distances between consecutive points in p_{List}. Finally, the number of points n_{Pts} only makes sense if Θ_{spec}=[] is the default value, in which case it controls the precision with which Θ_{spec} is calculated. The default is $n_{Pts} = 10000$.

4.5 Three more sample scripts

The following source code contains three building instructions, one for tropocollagen, one for a triangular helix, and one for the spherical spiral (cover art).

4.5.1 Tropocollagen

```python
from matriarch import *
from math import *

def collagen(seq1, seq2):
    a1 = chain(seq1)
    a2 = chain(seq2)
    hel1 = helixBuilder(a1,1.5,9.5238,'L')
    hel2 = helixBuilder(a2,1.5,9.5238,'L')
    helhel1 = helixBuilder(hel1,4,85.5,'L')
    helhel2 = helixBuilder(hel2,4,85.5,'L')
    helhel1rot = shiftOrbs(rotateOrbs(helhel1,2*pi/3),2.8)
    helhel2rot = shiftOrbs(rotateOrbs(helhel2,4*pi/3),5.6)
    homodimer = overlay(helhel1,helhel1rot)
    output = overlay(homodimer,helhel2rot)
    return output

def helixBuilder(myBB,rad,pitch,handed):
    scale = sqrt(rad*rad + pitch*pitch/(4*pi*pi))
    if handed=='R':
        sign=1
    elif handed=='L':
        sign=-1
    else:
        print handed,' should be L or R.'
    def parameterizedHelix(t):
        sc = sign/scale
        return [rad*cos(sc*t),-rad*sin(sc*t),pitch*t/(2*pi*scale)]
    W = buildAxisTwister(parameterizedHelix)
    return twist(myBB, W)

seq1 = 'GFZGPKGTAGEZGKAGERGVZGPZGAVGPAGKDGEAGAQGAZGPAGPAGERGEQGPA'
seq2 = 'GFZGPKGPSGDZGKZGEKGHPGLAGARGAZGPDGNNGAQGPZGPQGVQGGKGEQGPA'
collgn = collagen(seq1,seq2)
fileOut(collgn,'collgn.pdb')
```

The building instruction collagen works by making three coiled coils and overlaying them, to form a tropocollagen molecule. To do so, it first creates two linear chains, for seq1 and seq2. Then it twists the chains into left-handed helices with radius 1.5 Å and pitch 9.5238 Å. It then twists the results into a coil of coils, i.e., a left-handed helix with radius 4 Å and pitch 85.5 Å. These two building blocks are rotated and shifted, so that they can be overlaid with the original. See Figure 8.

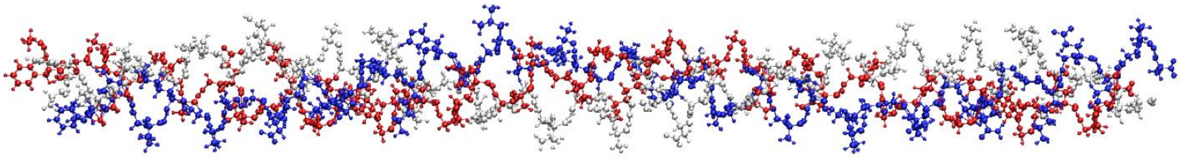


Figure 8

4.5.2 Triangular helix

```
from matriarch import *
from math import *

def triangleTwister(side, pitch, length, smoothingFactor):
    iterations = int(length/(3*side)) + 2
    const = sqrt(3)/6
    def loop(Z0):
        return [[side*2*const,0,Z0],[-
side*const,side/2.0,Z0+pitch/3.0],[-side*const,-
side/2.0,Z0+pitch*2/3.0]]
    PList = []
    for n in range(0,iterations):
        PList.extend(loop(pitch*n))
    Rout = Ray([0,0,0],[0,0,1])
    provideTheta = []
    return smoothedPieceWiseLinear(PList, Rout, provideTheta,
smoothingFactor)

aminoLength=3.4

actualSeq ='TNVIIEGNVTLGHRVKIGTGCVIKNSVIGDDCEISP'
Mult=3
Side=9*aminoLength
Pitch=7
SmthFact = 0.33
totalLen=Mult*aminoLength*len(actualSeq)
myChain = attachSeries(chain(actualSeq),Mult)
myTriangle = twist(myChain, triangleTwister(Side, Pitch, totalLen,
SmthFact))

fileOut(myTriangle,'triangleHelix.pdb')
```

The triangular helix can be found in nature [3]. We build it with Matriarch using the twist command, applied to an axis twister called triangleTwister, which encodes a parameterized triangular helix. The input parameters for this axis twister are the side-length (here, that of nine amino acids), the pitch (here 7 Å), a length (here, that of 3*36 amino acids), and a smoothing factor (here the default value of .33). See Figure 9 below for two views of the output.

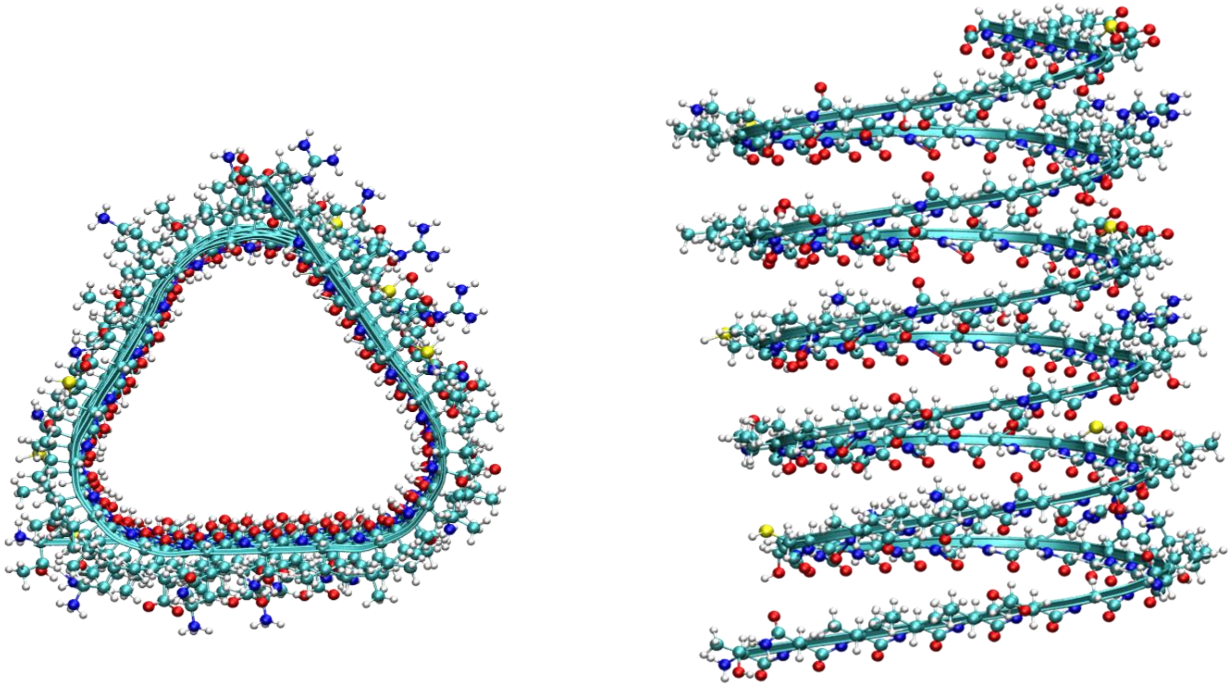


Figure 9

4.5.3 The spherical spiral (cover art)

The spherical spiral does not occur in nature—it is a construct to show the capability of our code.

```
from matriarch import *
from math import *

seq1 = 'AAAAGGPGGYGGPGGAAAA'
a = chain(seq1)
ser = attachSeries(a,25)

def SphSprl(k,Rout):
    def curve(t):
        return [k*sin(t)*cos(20*t), k*sin(t)*sin(20*t), k*cos(t)]
    tmax=pi
    Thetaspec=0
    return buildAxisTwister(curve,Rout,Thetaspec,tmax)

Rout = Ray([0,0,0], [0,0,0])
k1 = 1        #first try
W1 = SphSprl(k1,Rout)

contourLength = length(ser)
lengthOfCurveWithKEquals1 = W1[0].length
knew = contourLength / lengthOfCurveWithKEquals1

SphSprlTwister = SphSprl(knew + 0.001,Rout)
output = twist(ser, SphSprlTwister)

fileOut(output,'sphericalSpiral.pdb')
```

The SphSprl function parameterizes a spherical spiral. To avoid stretching or compressing bonds, the length of the spiral needs to be equal to that of the amino acid chain. This is what the scaling parameter k_{new} does for us. We compute k_{new} as follows. We first create the spherical spiral without scaling (k=1) and measure its length. The factor k_{new} is obtained by dividing the amino acid chain length by the measured length of the spherical spiral.

4.6 Technical remarks and trouble shooting

4.6.1 Stability of structures in MD

It is possible to use Matriarch to create configurations that will not run stably in a molecular dynamics simulator. This can happen either when atoms are too close, or when bonds are stretched non-physically. In either case, infinite forces may occur.

In most cases, a good work-around is to first run energy minimization on the configuration and then to set up a short NVT equilibration without holonomic constraints (LINCS or SHAKE) using a small time step. Then the production run using constraints should be stable.

4.6.2 Terminal charges in Matriarch

The ends of each polypeptide chain are left charged by Matriarch.

Note that hydroxyproline should not be placed at an end of a polypeptide strand because the PDB file for hyp (version 1.0) does not have correctly placed hydrogen atoms at the carboxyl terminal.

4.6.3 Optional parameters in Python

Several Matriarch commands contain optional parameters. The default value of each is given in the respective function descriptions above.

If a command has multiple optional parameters, and you want to specify for example the third, then the first and second also need to be specified. This is a constraint of the Python programming language. In other words, you can specify the earlier parameters without specifying the later ones, but you cannot specify the later parameters unless you also specify the earlier ones (which can be set to the default values if desired).

4.6.4 Comments on the buildAxisTwister command

It is recommended to use twist with buildAxisTwister. It is recommended that, when using buildAxisTwister and twist, the function f_{map} should be monotonically increasing in z, and $p_z\left(f_{map}(0)\right) = 0$, meaning that the z-component of $f_{map}(0)$ should be zero. The *buildAxisTwister* command will only work if the line from R_{out} to $f_{map}(t)$ is not tangent to f_{map}. If you get an error message, try specifying the total clutch to be a constant at the angle you think it should be at.

5 References

1. Giesa, T., et al., *A Python Library for Materials Architecture.* in submission, 2015.
2. Spivak, D.I., *The operad of wiring diagrams: Formalizing a graphical language for databases, recursion, and plug-and-play circuits.* http://arxiv.org/abs/1305.0297, 2013.
3. Solar, M. and M.J. Buehler, *Comparative analysis of nanomechanics of protein filaments under lateral loading.* Nanoscale, 2012. **4**(4): p. 1177-1183.