Using the Wormulator from the command line

Brian Ross

August 4, 2011

This document describes the command-line '`wormulator`' program, which computes end-to-end statistics for polymers. It's basically a souped-up version of the web-based wormulator, available at

`http://www.wiggins.wi.mit.edu/wormulator/`

The command-line `wormulator` application and the '.zoo' accessory files should all be in the same directory. To use the program:

- Open a command prompt window. On a PC this is the DOS prompt; on a Mac, use the Terminal application (inside Applications → Utilities).
- Navigate to the `wormulator` directory. Use the `cd` command to change directory. On a Macintosh the directory name looks something like "`/Users/me/Desktop/wormulator_mac/`". On a DOS machine it might read "`C:\WINDOWS\Desktop\wormulator_pc>`".
- Type "`wormulator`" to run the program. On the Macintosh/UNIX box you may have to type "`./wormulator`" (the dot tells the computer to look in the current directory).
- Perform calculations, save results, etc. by:

  1. entering commands into the command prompt

     ```
     >> 2 + 3

     5
     ```

  2. or, alternatively, putting the commands in a text file in the `wormulator` directory, and running that file from the command prompt:

     ```
     >> run("twoplusthree.txt")

     5
     ```

- To exit, enter the command '`exit`'.

1 What this program calculates

This first section duplicates the online explanation page, at:

`http://www.wiggins.wi.mit.edu/wormulator/help_what.html`

This `wormulator` tool calculates end-to-end statistics for polymers such as DNA. In particular, it calcuates the likelihood of finding the two ends of the polymer a given distance apart, and/or pointing in such-and-such direction relative to each other. The default is to model DNA using is the so-called 'wormlike chain' polymer model, where the polymer bends according to Hooke's Law. The Monte Carlo method can work with any number of other polymer models as well, and it includes a more realistic sequence-dependent DNA model due to Olson et. al[2].

The way to think about these distributions is to pretend that we are at one end of the polymer that points with a specified orientation, and then ask: what are the chances of finding the other end at a given location, and/or pointing at such-and-such other given orientation? In other words, what we calculate is the following probability density:

$$P\left(\mathbf{R}, \mathbf{u_f}, \mathbf{n_f} | \mathbf{u_0}, \mathbf{n_0}, L\right)$$

Here the displacement of the second end relative to the first is represented by $\mathbf{R}$, the tangent direction that a given end points is $\mathbf{u}$, and the 'normal vector' which determines the twist angle is $\mathbf{n}$. L is the length of the polymer. The situation is shown in Figure 1.

The units of the above distribution are given by the three quantities to the left of the vertical bar: it is a probability per volume in $\mathbf{R}$, per solid angle in final tangent $\mathbf{u_f}$, per twist angle T. The quantities to the right do not affect the units since they are the initial conditions at the starting end of the polymer.

Other reduced distributions can be obtained by summing or averaging over the quantities in the full distribution. For example, if we sum the distribution over all end-to-end displacements, we get just the probability of finding given relative tangents and twists, ignoring any dependence on $\mathbf{R}$. This distribution would be written

$$P\left(\mathbf{u_f}, T | \mathbf{u_0}, L\right)$$

Likewise, we can sum over (ignore) the tangents and/or twists.

Each output from `wormulator` is some distribution $P(...)$ evaluated only at a *single point* (i.e. at one value of $\mathbf{R}$, $\mathbf{u_f}$, and T). Built-in commands allow one to perform these measurements for various parameters or over many points of the distribution, and then print or save the results in tables that can be plotted in some other program.

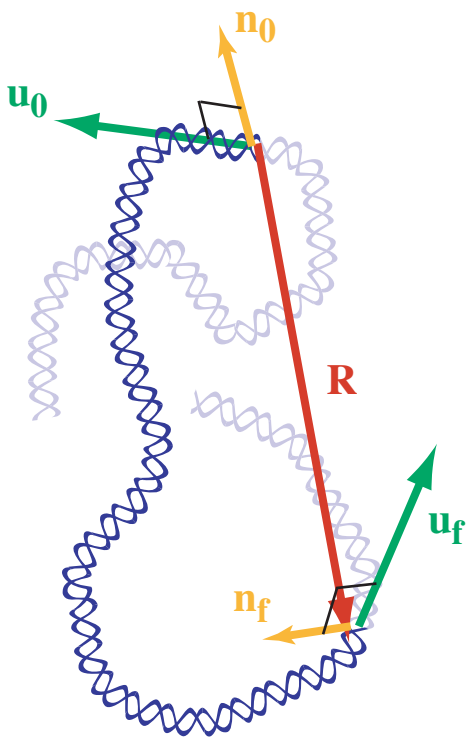

Figure 1: Above: one particular conformation of a DNA segment with the positions and orientations of its two ends marked as vectors. The calculated distribution would represent the probability of *any* conformation having the ends positioned as shown. The displacement is given by the red vector $\mathbf{R}$. The orientation is specified by green tangent vectors $\mathbf{u}$ and yellow normal vectors $\mathbf{n}$. The program can also accept a 'relative twist': if $\mathbf{u_f}$ and $\mathbf{n_f}$ are rotated together so that $\mathbf{u_f}$ is parallel to $\mathbf{u_0}$, then the relative twist is defined as the the angle between the $\mathbf{n_0}$ and $\mathbf{n_f}$.

2 Analytic calculations

2.1 Gaussian chain

When the total length of a polymer is much greater than its (bending) persistence length, the probability distribution in $\mathbf{R}$ (i.e. the likelihood of a given displacement of the two ends) approaches a distribution that is Gaussian in $R = |\mathbf{R}|$ and uniform in direction/orientation. It's trivial, but `wormulator` nevertheless provides the `Gauss` command for evaluating this distribution.

```
> Gauss(5, 3)

3.42551e-05
```

gives $p(R = 3; L = 5)$. The units are arbitrary, so long as they are consistent between L, l_p, $\mathbf{R}$, etc.

The persistence length is an optional parameter, set to 1 by default. If we want to change the persistence length, then we write a semicolon and set to a different value using an equals sign:

```
> Gauss(5, 3; persistence_length = 2.5)

1.94802e-05
```

In general, the required parameters to Wormulator commands always have to be given in the same order: in this case chain length, then end separation. Optional parameters, which come after a semicolon, are named explicitly when they are changed, so they can be set in any order. Both required and optional parameters are separated by commas.

There are three other optional parameters that control the particular distribution we are measuring. To ignore the relative twists of the two ends, we set the optional `sum_twist` to the value `true`. We can ignore tangents (and, automatically, twists as well) by setting `sum_tangent` to true. Finally, the way the end-to-end displacement is factored into the distribution depends on the `sum_R` parameter, which can be set to one of three values:

- 0: probability $P(\mathbf{R}, ...)$ of finding a given relative *vector displacement* of the two ends, whose magnitude we specified. Units are $1/\mathrm{L}^3$.
- 1: probability $P(R, ...)$ of finding the two ends a given *distance* apart. This is equivalent to integrating over a spherical shell. Units are $1/\mathrm{L}$.
- 2: probability $P(...)$ with no conditions on $\mathbf{R}$. This is equivalent to integrating over all space. Answer has no units.

For example, we can obtain the tangent-only distribution by the following:

```
> Gauss(2, 3; sum_twist = true, sum_R = 2)

0.0795775
```

2.2 Eigenfunction method

The eigenfunction method [3, 4] applies strictly to wormlike chain polymers. It computes the end-to-end distribution as an infinite sum of terms which can be reasonably truncated for long ($L \gtrsim l_p$) chains. The lowest-order $l = 0$ term is just the Gaussian approximation, while subsequent $l > 0$ terms successively refine the answer. The number of terms in the sum increases very quickly with l, so whereas truncating l at 0 involves only a single term, setting l_{max} at 1 or 2 necessitates summing 12 or 57 terms respectively; the runtime increases accordingly.

In addition to the sum over l-values, the eigenfunction computation involves a further integration from Fourier 'K'-space to real space:

$$\begin{aligned} p() &= \int_0^\infty dK \int_0^\pi d\theta \int_0^{2\pi} d\phi \cdot K^2 \sin\theta \cdot ... \\ &\approx \sum_0^{K_{max}} \Delta K \sum_0^\pi \Delta\theta \sum_0^{2\pi} \Delta\phi \cdot K^2 \sin\theta \cdot ... \end{aligned}$$

The tolerances of the integration are controlled by four parameters: K_{max}, ΔK, $\Delta\theta$, $\Delta\phi$. The general strategy is to increase l_{max} and K_{max}, and decrease ΔK, $\Delta\theta$ and $\Delta\phi$, until the answer stops changing. These tolerances must be set much more stringently for short chains than long ones.

The Wormulator decouples two steps of the eigenfunction calculation, in order to speed up mapping a distribution over many points. In the first 'initialization' step one passes l_{max}, ΔK and K_{max} in that order. In step 2, after the initialization, the distribution can be evaluated at different chain lengths L and over different $\mathbf{R}$, $\mathbf{u}_0$, $\mathbf{u}_f$, etc. L is the only required parameter for the second step; all other parameters are optional and so come after the semicolon. Eigenfunction calculations work through the `EF` object.

The example below performs two calculations using the same initialization. The first calculation returns a probability density for cyclization that ignores twist, because by default `R = { 0, 0, 0 }`, `uf = u0` and `nf = n0`. The second calculation uses a different set of boundary conditions, but this time it includes twist, because `sum_twist` defaults to `false`. In both cases $L = 5$.

```
> EF.Init(3, .1, 30)

> EF.P(5; sum_twist = true)

0.00059331

> EF.P(5; u0 = uf = { 0, 0, 1 }, n0 = { 1, 0, 0 }, nf = { 0, 1, 0 })

6.13346e-05
```

In these last calculations, the relative twist between the two ends was implied by the pair of normal vectors `n0` and `nf`. Since the answer only depends on the relative twist angle ψ between the ends, it is sometimes easier to just specify ψ directly. We could have done the same as calculation 2 above using `SetTwist()`:

```
> EF.P(5; u0 = uf = { 0, 0, 1 }, SetTwist(pi/2) )

6.13346e-05
```

In the general case, where `u0` and `uf` differ by some rotation $\mathbf{R}$, `SetTwist()` obtains `nf` by first rotating `n0` by $\mathbf{R}$, and then further rotating by ψ along $\mathbf{u}_f$.

The name "`SetTwist()`" is unfortunately somewhat misleading, in that it only specifies `n0` and `nf` rather than imposing the relative twist angle directly. The result is therefore a sum of the contributions of chains having relative twist $\psi + N \cdot 2\pi$ where N is any integer. There is no way of selecting, e.g., one linking number in the case of cyclization.

Other optional arguments to `EF.P()` following the semicolon let the user change the material parameters: the (bending) persistence length, the twist persistence length, and the unstressed twist angle. The default values of these are for DNA, in units of the (bending) persistence length. All angles are in radians.

The final set of optional parameters instruct `EF.P()` to calculate various reduced distributions, by summing over the ignored degrees of freedom. In particular, the `sum_R` argument takes three values: 0 means no summation, 1 sums over a spherical shell (i.e. integrates over θ and ϕ but not R), and 2 integrates over all space. If `sum_R` is set to 2 then we can skip the initialization step.

Any of the following conditions make the eigenfunction calculation run much faster: `sum_R` is set to 2, `sum_tangent` is true, `sum_twist` is true, or a cyclization probability is being computed. Cyclization occurs when the relative displacement is precisely zero, the initial and final tangents are equal, and the relative twist is zero (i.e. the initial and final normal vectors are equal).

Summary of eigenfunction and Gaussian chain commands

Gauss(polymer_length, end_separation ; optional parameters**)**

Calculates $p()$ based on the Gaussian chain model.

optional parameters	default value	description
`persistence_length`	1	bending persistence length of polymer
`sum_R`	0	whether/how to sum distribution over space
`sum_tangent`	false	whether to sum distribution over relative tangents
`sum_twist`	false	whether to sum distribution over relative twists

EF.Init(l_max, K_step, K_max**)**

Initializes the eigenfunction calculator using given tolerances. No optional parameters.

EF.P(polymer_length ; optional parameters**)**

Calculates $p()$ using the eigenfunction method. This must follow an `EF.Init()` command, except when `sum_R = 2`.

optional parameters	default value	description
`persistence_length`	1	bending persistence length of polymer
`twist_persistence_length`	2.08	twist persistence length of polymer
`natural_twist`	98	twist rate of polymer when relaxed
`R`	{ 0, 0, 0 }	relative displacement of two ends: $\mathbf{R}_2 - \mathbf{R}_1$
`u0`	{ 0, 0, 1 }	tangent vector at first end
`uf`	{ 0, 0, 1 }	tangent vector at second end
`n0`	{ 1, 0, 0 }	normal vector at first end
`nf`	{ 1, 0, 0 }	normal vector at second end
`sum_R`	0	whether/how to sum distribution over space
`sum_tangent`	false	whether to sum distribution over relative tangents
`sum_twist`	false	whether to sum distribution over relative twists
`theta_step_num`	100	number of integration steps in θ
`phi_step_num`	100	number of integration steps in ϕ

EF.SaveRoots(file_to_write**)**

EF.LoadRoots(file_to_read**)**

Save/load an eigenfunction initialization (the result of `EF.Init()`). No optional parameters.

3 Numerical calculations

3.1 Monte Carlo

The Monte Carlo method randomly generates polymer chain configurations composed of discrete straight segments, and then samples the statistics of their endpoints. The trajectory of each segment is determined by three translations (shift, slide, rise) and three rotations (bend, azimuthal angle of bending, twist). The difference between polymer models is in the energy functions they assign these degrees of freedom. For generality, these energy functions are stored in discrete interpolation tables. Making a Monte Carlo measurement is therefore a two- or three-step process: 1) initialize with a particular DNA model; 2) generate the polymer chains and/or 3) sample statistics from the endpoints of these chains. This way we can reuse the time-consuming initialization and chain-generation steps over many measurements.

All numerical calculations are done through the `MC` object:

```
> MC.InitWormlike( ; segment_length = .1 )

> MC.P(10, 1e4; at(r, { 0, 0, 1 } ), dr = 0.1)

{ 1.19366, 0.196139, 50 }
```

The three numbers returned by `MC.P()` were the estimated p-value, the error, and the number of 'hits' (samples that satisfied the boundary conditions). (These numbers will change somewhat over different runs because Monte Carlo is stochastic.)

3.1.1 Initialization

There is a very general way to initialize the numerical calculator with a DNA model, but it is easier to use one of the built-in DNA models if that is what we are interested in. The first built-in model is the **wormlike chain model**, which uses the `MC.InitWormlike()` command we demonstrated above. All parameters are optional, so if we do pass any parameters we first need a semicolon. The basic parameters are: `persistence_length` and `twist_persistence_length` for bending and twist respectively; `unstressed_twist` which gives the zero-strain twist in radians/length; and `segment_length` which determines the length of the discrete segments. If we want to initialize a bending-only polymer we can set `do_twist = false`.

Two further parameters control the approximation in the interpolation tables: `dist_evals` gives the number of interpolating points in both bending and twist, and `sigmas` gives the number of standard deviations where each Gaussian ($e^{-l_p\theta^2/2}$, $e^{-l_t\psi^2/2}$) is truncated. For finer control, one can give the discretization of the bend/twist tables individually using the `BendRange` and `TwistRange` parameters: each contains the minimum and maximum range values and the number of sampled points. For example, using the defaults for `persistence_length` and `segment_length`, both methods below create the same bending distributions.

```
> MC.InitWormlike( ; sigmas = 4, dist_evals = 50 )

> MC.InitWormlike( ; BendRange = { 0, 4/10^.5, 50 } )
```

To model a 2-dimensional wormlike chain polymer, we use the `init_2D` parameter:

```
> MC.InitWormlike( ; init_2D = true )
```

This fixes the azimuthal angle ϕ at 0 and extends the range of θ to negative bending angles.

We can use a more realistic **DNA model** using the base-step parameters of Olson et. al.[2] by calling the `MC.InitBP()` initialization. Here the units of length are Angstroms, and each segment corresponds to a single base step. The only optional parameters are `sigmas` and `pdf_samples`, having the same meaning as in `MC.InitWormlike()`.

```
> MC.InitBP( ; sigmas = 3, pdf_samples = 200 )
```

`MC.InitBP()` initializes both an average base-pair-step model, as well as a sequence-dependent model, and we will have the option of choosing sequences when we evolve the polymers.

The **general-purpose** `MC.Init()` **command** is used to build a numerical polymer model that is neither wormlike-chain nor our built-in DNA model. Using this, we need to give an energy function for each degree of freedom that differs from the default initialization. The degrees of freedom are `shift`, `slide` and `rise` for translations between segments along the normal, binormal and tangent axes; and `bend` (θ), `azimuth` (azimuthal angle of bending ϕ) and `twist` (ψ) for rotations. Inside the parentheses, the syntax is: the number of energy functions for each degree of freedom (if we have a sequence-dependent model; otherwise we leave it blank); semicolon; the various energy functions; semicolon; optional parameters, including `DistRange[]` which sets the extremes (minimum, maximum) and number of samples for each interpolation table. Here is an example that creates an extensible wormlike chain model with a cubic contribution to the bending energy. (The `&` just continues a broken line.)

```
> MC.Init( ; { bend; return 3*bend^2 + 0.1*bend^3 }, &
             { rise; return 40*(rise-1)^2 } ; &
             DistRange[bend] = { 0, pi/2, 100 }, &
             DistRange[rise] = { 0.5, 1.5, 100 } )
```

Degrees of freedom that weren't mentioned—`shift`, `slide`, `azimuth` and `twist`—took default values. In our case `azimuth` was a uniform distribution over $[0, 2\pi]$ and the others were fixed at zero. Passing no distributions at all generates an inextensible, unbendable polymer model. The ranges and discretization of the distribution also take default values unless they are changed. If we're curious about these we could type, e.g.:

```
> MC.DistRange[slide]

{ { 0 }, { 0 }, 1 }
```

A shortcut for setting the rise distribution of inextensible polymers is to use `SegmentLength()`. The two commands below both make identical polymer models with inextensible segments of length of 3.

```
> MC.Init( ; ; DistRange[rise] = { 3, 3, 1 } )

> MC.Init( ; ; SegmentLength(3) )
```

Coupled degrees of freedom have to be put into the same interpolation table. For example, if we're modeling a bending polymer that is extensible in both rise and shift, the r^2 term coming from use of the spherical basis couples the two translations, and we would write something like:

```
> MC.Init( ; { bend; return 0.5*bend^2 }, &
             { shift, rise; return 0.1*(shift^2 + (rise-.4)^2) } )
```

Multi-dimensional tables require much more memory and take much longer to initialize than one-dimensional ones, because the number of entries in the table is the product of the number of samples along each dimension.

Sequence-dependent models require that we 1) tell `Init()` the number of sequences before the semicolon, and 2) enter a separate function for each sequence, for each distribution. For example, an extension-only polymer with stiff, flexible and loose links might look like:

```
> MC.Init(3; { rise; return 10*(rise-1)^2; return .5*rise^2; return 0 }; &
             DistRange[rise] = { { 0, -4, 2 }, { 2, 4, 3 }, 100 })
```

Notice how each sequence table can have different minimum and maximum values, which is why the minimum and maximum ranges turned from single numbers to lists grouped by braces.

The interpolation tables absorb a **volume element** from the spherical basis, which for a wormlike chain is $r^2|\sin\theta|$. In most cases `MC.Init()` knows what these factors should be, but for some strange initializations it may not be able to guess properly. (To check them, type `MC.Init.vol_els`.) If need be, we can specify the volume elements ourselves, by setting the option `default_vol_el = false` and giving the appropriate factor after the (last) energy function. In the example below, we discover that the Wormulator doesn't understand that the bend axis is always aligned with the `slide` axis when there's no azimuthal freedom, so we pass a unit volume element manually instead.

```
> MC.Init( ; { slide; return slide^2 }, { bend; return bend^2 }; &
             DistRange[rise] = DistRange[azimuth] = { 0, 0, 1 })

> MC.Init.vol_els

{ 1, (slide^2)^.5, 1, 1, 1, 1 }


> MC.Init( ; { slide; return slide^2; return 1 }, &
             { bend; return bend^2; return 1 }; &
             DistRange[rise] = DistRange[azimuth] = { 0, 0, 1 }, &
             default_vol_el = false)

> MC.Init.vol_els

{ 1, 1, 1, 1, 1, 1 }
```

Degrees of freedom that are coupled by the volume factor must appear in the same distribution.

As a debugging tool, we can save our polymer's energy functions into text files that can be plotted in another program. The command to do so is `MC.ExportDists()`. If we have a sequence-dependent model we would look at, for example, the third energy function of each distribution by writing `MC.ExportDists(3)`. Each output file contains a row for every coordinate in the distribution followed by the energy at that point: for example the columns of `shift_twist_dist.txt` would

contain shift, twist, energy in that order. Remember that these energy functions will have absorbed various $-\log v_i$ terms coming from the volume element $\prod v_i$.

3.1.2 Calculating p-values

To calculate a p-value we use `MC.P()` after initializing a model. The syntax inside of the parentheses is: 1) the number of segments of each polymer to generate, then 2) the number of samples; semicolon; any conditions on the chain endpoints/midpoints using `at()` along with any optional parameters. Some of the optional parameters—`r0`, `n0`, `b0` and `u0`—give the initial state of the chain; others (`dr` and `dangle`) give the volume of the target space in units of length and radians. The result will be the number of hits divided by the target volume. Increasing the target volume reduces counting error but averages the distribution over a larger region. A typical calculation of the probability density for evolving from $(1, 0, 0)$ to the origin might be:

```
> MC.InitWormlike()

> MC.P(20, 1e5; at(r, { 0, 0, 0 }), r0 = { 1, 0, 0 }, dr = .2)

{ 0.0214859, 0.00130076, 72 }
```

The way to read the output is: $p \approx 0.0235748 \pm 0.00191079$, as estimated from 79 samples.

There are two ways to measure the probability density.

1. The one-shot method is to generate chains, storing in memory the endpoints of only those chains that satisfy the given critera within tolerances. If relatively few chains are expected to pass selection, then we can use the `max_hits` option to limit the memory overhead. In the example below we only allocate storage for 1000 endpoints even though we conduct 10^5 runs. The calculation ends after either 10^5 chains have been tested, or the number passing selection hits 1000. In our case we can see that the former condition stopped the calculation, at which point the storage space shrank from 1000 to 16 (the number of hits).

   ```
   > MC.P(20, 1e5; at(r, { 0, 0, 0 }), dr = .2, max_hits = 1e3)

   { 0.00447623, 0.00124836, 15 }
   ```

2. The alternative, 2-step method is to first store the endpoints of *all* chains with *no* selection criteria, then to search that long list for those that satisfy the endpoint criteria using a second call to `MC.P()`. The second `MC.P()` shouldn't regenerate the chains, and we communicate this to the Wormulator by omitting everything before the semicolon. Decoupling chain-generation from endpoint-sampling makes repeated measurements of the same distribution much faster, because chain generation is much the slower step and we only have to do that once.

   ```
   > MC.P(20, 1e5)


   > MC.P( ; at(r, { 0, 0, 0 }), dr = .2)

   { 0.00447623, 0.00124836, 15 }
   ```

Using method (2), we can also calculate the mean of R^{2n} and $(\mathbf{R} \cdot \mathbf{u_0})^n$ from a set of chains. The parameter to either command is the power n; the default is $n = 1$. This next example calculates $\mathbf{R} \cdot \mathbf{u_0}$ and R^4.

```
> MC.InitWormlike(), MC.P(20, 1e5)


> MC.R_dot_u0()

{ 0.894948, 0.00116446 }


> MC.R_2N(2)

{ 6.27563, 0.00693568 }
```

The **at() function** lets us set position constraints by writing a '`r`' (as in "`at(r, {...})`"), or orientation constraints on the tangent by writing '`u`' or '`z`', constraints on the normal by `n` or `x` and binormal constraints by `b` or `y`. Ignorable degrees of freedom in `r`, denoted by a '`*`', are useful for setting one- or two-dimensional `r`-constraints:

```
> MC.P(10, 1e5; at(r, { *, 0, * }))
```

We can also put constraints on the midpoints. The command below selects chains having a tangent along $\hat{z}$ at the midpoint of the chain (end of segment 5), and a tangent along $-\hat{z}$ at the end (of segment 10 by default).

```
> MC.P(10, 1e5; at(u, { 0, 0, 1 }; segment = 5), at(u, { 0, 0, -1 }))
```

We need to have the Wormulator **store midpoints** when we generate the chains, if we want to set criteria on those midpoints of a chain using method #2, or if we want to look their R^2 or $\mathbf{R} \cdot \mathbf{u_0}$ statistics. Borrowing our last example, the way to do this is as follows:

```
> MC.P( { 5, 10 }, 1e5)

> MC.P( ; at(u, { 0, 0, 1 }; segment = 5), at(u, { 0, 0, -1 }))
```

At the extreme, we can store the endpoint of every single segment by the shorthand:

```
MC.P( { 10, * }, 1e5)
```

which is equivalent to the more laborious:

```
MC.P( { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }, 1e5)
```

Memory usage goes up accordingly.

If it is more convenient to specify a relative twist angle between the two ends rather than their normal vectors, use `SetTwist()` (discussed in the eigenfunction section). There must be a final tangent constraint for this function to work. Example:

```
> MC.P(10, 1e5; at(u, { 0, 0, 1 }), SetTwist(pi/3))
```

which sets `nf = { 1/2, 3^.5/2, 0 }`. If we are computing cyclization with some arbitrary twist, then we can check the linking number of the last sample.

```
> MC.P(10, 1; at(r, { 0, 0, 0 }), at(u, { 0, 0, 1 }), SetTwist(pi/3))

{ 0, nan, 0 }

> MC.LinkingNumber()

-16
```

Sequence-dependent models use a list of numbers that determine which set of energy functions is used by each segment. The default sequence is `{ 1, 1, 1, ... }`. To change this we can use the optional `sequence()` command, with the number sequence either specified directly or retrieved from a file using `Load()`.

```
MC.P( 5, 1e5; sequence("1 2 3 2 3") )

MC.P( 5, 1e5; sequence(Load("mysequence.txt")) )
```

Using the `InitBP()` initialization, it is more convenient to give a base-pair sequence. We can do this by using `bp_sequence()` instead of `sequence()`, passing the base pairs either directly or as a file as before. Since each *dinucleotide* step corresponds to a segment, the number of bases should be one more than the length of the chain.

```
MC.P( 5, 1e5; bp_sequence("acgtag") )
```

One final optional parameter to `MC.P()` that bears mentioning is `poly_translate_mode`, which determines how the translations of the polymer co-evolve with the rotations. This parameter can take one of three integer values: -1, 0, or 1. Setting it to `poly_translate_mode = -1` causes each segment to translate (shift, slide, rise) along its initial orientation vector; setting it to '1' translates from the final orientation. Setting `poly_translate_mode = 0` translates from the 'midstep vector' (see [1]).

3.2 High-energy methods

The numerical component of the Wormulator has several ways of trying to calculate high-energy p-values, by sampling or direct integration. The pure-sampling high-energy method involves sampling the chains from a different distribution than the actual, then correcting by post-weighting. The integration methods make a quadratic approximation to the energy function (after absorbing the volume element). A hybrid method uses the quadratic basis as a sampling-bias function. We'll take each of these in turn.

3.2.1 Biased sampling

The straightforward biased-sampling method involves two initialization calls, giving: 1) the polymer model function using the true energy function, and 2) an imaginary energy function whose Boltzmann distribution is representative of chains that hit their target. Step 1 is just the standard

initialization procedure, using `InitWormlike()`, `InitBP()` or `Init()`. Step 2 involves using either `InitWormlike()` or `Init()` as before, but adding `biased_distribution = true` in the optional parameters section. As an example we look at a highly-extended 1-D extensible chain.

```
> MC.Init( ; { rise; return 0.5*rise^2 } ; &
             DistRange[rise] = { -3, 7, 1000 }, &
             DistRange[azimuth] = { 0, 0, 1 } )

> MC.Init( ; { rise; return 0.5*(rise-2)^2 } ; &
             DistRange[rise] = { -2, 6, 100 }, &
             DistRange[azimuth] = { 0, 0, 1 }, &
             biased_distribution = true )

> MC.P( 5, 1e5; at(r, { *, *, 10 }), dr = .01 )

{ 8.38005e-06, 5.15665e-07, 367 }
```

We need to be careful that the range of the first initialization encompasses the range of the second initialization; otherwise we are likely to get a warning about sampling above/below some of the distributions. The first initialization also needs to cover the normal range of the unconstrained polymer, in order to integrate the partition function accurately. The resolution (third number in `DistRange[]`) needs to be reasonably high for the first, though not second, initialization.

The biased sampling method tends to work only when the sampling distributions are close to the actual distributions of constrained polymers. Otherwise, there tends to be a disparity in the post-weights and the answer depends heavily on only one or a few samples. We can output the constrained distribution, as estimated from the weighted distribution of hits, into the file `hits.txt` by typing:

```
> MC.SaveHitTrajectories(rise)
```

To improve the sampling of bending chains, we can set the optional parameters `fixed_bend = true` (for 2D chains) or `fixed_azimuth = true` (3D chains) to have the respective angles sampled against a fixed coordinate axis rather than the local unit triad.

3.2.2 Direct integration

The first step in integrating the constrained distribution directly is to initialize the model as usual:

```
> MC.Init( ; { rise; return 0.5*rise^2 } ; &
             DistRange[rise] = { -3, 7, 1000 }, &
             DistRange[azimuth] = { 0, 0, 1 } )
```

The second step is to find an approximate minimum-energy configuration using `Propagate()`; if we start close enough then `MC.P()` will be able to find the true energy minimum on its own. Inside the parenthesis we specify the number of segments of the chain, then the shift-slide-rise-bend-azimuth-twist trajectory function after a semicolon.

```
> MC.Propagate( 5; return { 0, 0, 2, 0, 0, 0 } )
```

Finally we make the measurement as usual, except that we set `perturbative = true` in `MC.P()`.

```
> MC.P( 5, 0; at(r, { *, *, 10 }), dr = .01, perturbative = true )

{ 8.15422e-06, 8.1543e-06 }
```

We see that the integration method gives us two answers. The first result enforces soft constraints by quadratic potentials. The second enforces the constraints exactly by the method of Zhang and Crothers[5]. The usefulness of the first method is that it provides the eigenmodes as a basis for sampling, as described in the next section.

The integration methods only use as many components of the angular constraints as there are constrained degrees of freedom. For example, if we have `at(u, { 0, 0, 1 })` then the scalar constraints are $u_x = 0$ and $u_y = 0$ but ignores the redundant $u_z = 1$; if we have `at(u, { 0, 0, 1 })` followed by `at(n, { 1, 0, 0 })` then the scalar constraints are $u_x = u_y = n_z = 0$. The program is usually able to pick the best components for the analysis, but in certain cases it may pick wrong and complain about redundant or missing constraints or not being able to find Lagrange multipliers. In such a case we can force a different choice by writing some components within braces to signal that are ignorable insofar as the integrations go. For numerical accuracy we should pick small-magnitude components; never a ± 1 component (see [5]). In the following example of an bending-only 2D chain the final orientation is specified only by a single angle, so we need to tell the Wormulator that there is only one degree of freedom to constrain. This must be $u_x = 0$ because a 2D chain with the default initial orientation cannot move in $\hat{y}$.

```
> MC.InitWormlike( ; init_2D = true, do_twist = false, segment_length = .01)

> MC.Propagate( 20 ; return { 0, 0, 0.01, pi/10, 0, 0 } )

> MC.P(20, 0; perturbative = true, at(u, { 0, { 0 }, { 1 } }))

{ 7.74798e-44, 1.19298e-43 }
```

If we just want to obtain the minimum-energy state of the polymer without integrating the probability density, we can use the command `MC.MinE()`:

```
> MC.MinE( ; at(r, { *, *, 10 }) )

> SaveTable("minE_conformation.txt", MC.Trajectory)
```

3.2.3 Hybrid: quadratic biased sampling

The methods of direct integration approximate the energy functions as Gaussians in the coordinates; they provide no way to improve on that approximation if it is inaccurate. But if those Gaussians are used as a sampling basis, then Monte Carlo with enough runs should be able to get a better result. Adding a sampling component to the method of the last section is straightforward: just give a number of runs greater than zero (2nd parameter of `MC.P()`).

```
> MC.Init( ; { rise; return 0.5*rise^2 } ; &
             DistRange[rise] = { -3, 7, 1000 }, &
             DistRange[azimuth] = { 0, 0, 1 } )

> MC.Propagate( 5; return { 0, 0, 2, 0, 0, 0 } )

> MC.P( 5, 1e5; at(r, { *, *, 10 }), dr = .01, perturbative = true )

{ { 8.13374e-06, 2.23502e-08, 68052 }, 8.15422e-06, 8.1543e-06 }
```

The first three numbers are the result, error and number of hits from Monte Carlo. The last two numbers are the exact integrated results (see last subsection).

For bendable chains, the hybrid-sampling method works poorly when the constraint potentials (determined by `dr` and `dangle`) are much stiffer than the energetic potentials, because higher orders of the least-stiff eigenmodes start coupling to the constraints, leading to very discrepant weighting factors. By this logic there is a tradeoff between sampling noise and averaging error over the sample volume. If this is a problem we can try to adjust the amplitudes of the eigenmodes without changing the sampling volume by using the optional `amplitude` parameter, as in:

```
> MC.P( 5, 1e5; at(r, { *, *, 10 } ; amplitude = 2), &
        dr = .01, perturbative = true )

{ { 8.12318e-06, 5.72334e-08, 35484 }, 2.93427e-06, 2.93439e-06 }
```

With a doubled amplitude in the constraint-breaking eigenmode, roughly half as many samples hit their target, so statistical noise increased. In this simple example discrepant weights are not a problem, but if they were the weighting noise would probably have been reduced.

Summary of Monte Carlo commands

MC.Init([# sequences] ; f_1, f_2, ... ; optional parameters**)**

Initializes a Monte Carlo object with the specified distribution functions. Each distribution f_i has the form: { x_1, x_2 ; return $E_1(x)$; return $E_2(x)$; ... [; return $v(x)$] }, where x_i is a degree of freedom (shift, slide, etc.) and $E(x)$ and $v(x)$ are energy/volume element functions.

options	default value	description
`DistRange[shift]`	{ 0, 0, 1 } / { -1, 1, 100 }	ranges of interpolation tables over dofs that were not / were given ({min, max, samples }
`DistRange[slide]`	{ 0, 0, 1 } / { -1, 1, 100 }	
`DistRange[rise]`	{ 1, 1, 1 } / { 0, 2, 100 }	
`DistRange[bend]`	{ 0, 0, 1 } / { 0, pi, 100 }	
`DistRange[azimuth]`	{ -pi, pi, 2 } / { -pi, pi, 100 }	
`DistRange[twist]`	{ 0, 0, 1 } / { -pi, pi, 100 }	
`biased_distribution`	`false`	`true` if setting bias distributions
`default_vol_el`	`true`	`false` if user provides a $v(x)$ function
`invert_CDFs`	`false`	pre-invert PDF tables for faster sampling (inaccurate w/ high-E methods)

MC.InitWormlike(; optional parameters)

A simpler alternative to `Init()` if we are using the wormlike chain model.

options	default value	description
`persistence_length`	1	bending persistence length of polymer
`twist_persistence_length`	2.08	twist persistence length of polymer
`unstressed_twist`	98	twist rate of polymer when relaxed
`segment_length`	1	length of each segment
`dist_evals`	100	discretization of distributions
`sigmas`	5	distribution cutoff
`BendRange[shift]`	{ $\mp\sigma\sqrt{l_s/l_p}$, 100 }	range of bending table
`TwistRange[slide]`	{ $\mp\sigma\sqrt{l_s/l_t}$, 100 }	range of twist table
`init_2D`	`false`	use a 2-D WLC model
`do_twist`	`true`	allow twist
`biased_distribution`	`false`	`true` if setting a bias function

MC.InitBP(; optional parameters)

Initializes a sequence-dependent DNA model.

`dist_evals`	100	discretization of distributions
`sigmas`	5	distribution cutoff

MC.Propagate(# of segments ; $x(i)$; optional parameters)

Propagates the segments of a chain using the specified trajectory function. $x(i)$ should be of the form `return [expr]`, where `expr` can involve the segment number written as '`segment`'.

options	default value	description
`r0`	{ 0, 0, 0 }	initial position
`n0`	{ 1, 0, 0 }	initial normal vector
`b0`	{ 0, 1, 0 }	initial binormal vector
`u0`	{ 0, 0, 1 }	initial tangent vector

MC.P([# of segments, # samples] ; at(..), at(..), ..., optional parameters)
MC.MinE(; at(..), at(..))

Finds the minimum energy configuration, and in the case of `MC.P()` then generates a series of chains using the current initialization and/or saves the mid/endpoints of a selection of these. Each constraint requires: `at(type, { a, b, c } ; options)` where the type is one of (r/x/y/z/n/b/u).

options	default value	description
`r0`	{ 0, 0, 0 }	initial position
`n0`	{ 1, 0, 0 }	initial normal vector
`b0`	{ 0, 1, 0 }	initial binormal vector
`u0`	{ 0, 0, 1 }	initial tangent vector
`max_hits`	# samples	endpoint storage size/calc. cutoff
`dr`	1.	max. $d\mathbf{R}$ for position constraint
`dangle`	pi/6	max. $d\theta$ for orientation constraint
`sample_all`	`false`	store all conformations
`perturbative`	`false`	quadratic approximation
`fixed_bend`	`false`	measure bend angles on fixed frame
`fixed_azimuth`	`false`	measure azimuthal angles on fixed frame
`poly_translate_mode`	0	segment translation axes
`minE_method`	4	energy minimization algorithm (1-4)
`max_iterations`	1000	# of iterations before giving up on E-minimization
`grad_convergence_limit`	10^{-6}	max gradient for E-minimization
`max_C`	10^{-6}	max constraint violation for E-minimization
`segment -- at()`	# segments	segment to constrain endpoint
`amplitude -- at()`	1	constraint amplitude for quadratic-bias sampling

MC.R_2N(moment ; optional params)
MC.R_dot_u0(moment ; optional params)

Measure the even moment $\langle R^{2n} \rangle$, or the mean dot product $\langle (\mathbf{R} \cdot \mathbf{u0})^n \rangle$, of the sampled mid/endpoints.

options	default value	description
`segment`	# segments	segment to measure

4 Miscellaneous

4.1 Generating tables

To map a distribution over one or more parameters, we just iterate the commands described above while systematically changing those parameters. There is an automated tool to do this called `MakeTable()`. The input to `MakeTable()` is: the filename for the output, followed by a list of: each parameter that we want to vary in double-quotes, along with their minimum and maximum values and optionally their spacing; semicolon, then the commands that generate output. If we also want screen output as each result comes in, then we put a second semicolon and invoke `writeout = true`. Example:

```
> MakeTable( "test_lK.txt", { "l", 2, 4 }, { "Kmax", 20, 40, 10 } ; &
             EF.Init( l, 0.1, Kmax ), &
             EF.P( 5 ; sum_twist = true ) ; &
             writeout = true )

2          20         5.90488e-4
2          30         5.8079e-4
2          40         5.83384e-4
3          20         6.13102e-4
3          30         5.93310e-4
3          40         5.86193e-4
4          20         6.01703e-4
4          30         5.92093e-4
4          40         5.94614e-4
```

The first column is the `l`-value, the second column is `Kmax` and the final column is the output of the `EF.P()` function. Since for `l` we only gave minimum and maximum values, the spacing was assumed to be one.

There is no restriction on what our parameters are called, so long as they are alphanumeric (beginning with a letter) and avoid reserved keywords like `print`. Also, they can go anywhere, including the initialization commands, and we can have as many outputs as we like.

```
> MakeTable( "test_lp.txt", { "lp", 2, 4, 0.5 } ; &
             Gauss( 5, 0 ; persistence_length = lp, sum_tangent = true ), &
             MC.InitWormlike( ; segment_length = 0.1, persistence_length = lp ), &
             MC.P(50, 1e5 ; at(r, { 0, 0, 0 }), dr = 0.4) ; &
             writeout = true )

2          3.68864e-3  8.20642e-4  1.51521e-4  22
2.5        2.63938e-3  2.98415e-4  9.51016e-5  8
3          2.00784e-3  7.46039e-5  4.56853e-5  2
3.5        1.59334e-3  3.73019e-5  3.73019e-5  1
4          1.30413e-3  0           0           0
```

The columns are: 1) `l`; 2) the output from `Gauss()`; 3-5) the output from `MC.P()` (answer, error, number of hits). Clearly the Gaussian chain model becomes invalid as $l_p \to L$.

4.2 Mathematical functions

The commands we've described all require parameters that are numbers. We can either calculate these numbers ourselves and plug them in directly, or else have the Wormulator calculate them. To show this, we'll write the same `at()` function in three different ways:

```
at(u, { 0.5, 0.866025, 0 })
at(u, { 1/2, 3^.5/2, 0 })
at(u, { cos(pi/3), sin(pi/3), 0 })
```

If we choose to let the Wormulator evaluate the numbers, we have the standard palette of basic mathematical operations, functions and constants to choose from. The standard operators are `+`, `-`, `*`, `/`, `^` (exponentiation). `pi` and `e` are predefined; the exponential function is written `e^...`. The built-in functions are: `log()` (base e); the trigonometric functions: `sin()`, `cos()`, `tan()`, along with their inverses: `asin()`, `acos()`, `atan()`; `abs()` (absolute value); `round_down()`, `round_up()` (to nearest integer); `random()` (uniform on [0, 1]); `min()`, `max()`. The keyword `ans` is a shortcut for whatever was last printed.

4.3 Troubleshooting

Since the Monte Carlo calculation is rather complex, there are a few internal checks we can do to make sure we're doing what we think. These are split between internal tables we can look at, and functions that export files with useful information.

After the initialization step we can look at the distributions by entering `ExportDists()` and `ExportBiasedDists()`. As described earlier, these create a set of files, named according to the initialization, containing the coordinates of each interpolation point along with its p-value. Note: after a `MC.P()` call in which `perturbative = true`, running `ExportBiasedDists()` outputs the energy functions, i.e. the $-log()$ values of the (true) probability distributions.

There are two variables that store the state of the current chain after running `MC.P()` or `MC.Propagate()`. In the case of `MC.P()` this will be the *last* chain generated, regardless of whether it hit its target or not.

- `MC.Trajectory` contains 6 columns: shift-slide-rise-bend-azimuth-twist of each segment in the chain.
- `MC.Segments[*].r` contains 3 columns: the x-y-z positions of the endpoints of the segments. There are $N + 1$ segment endpoints for a chain having N segments.
- `MC.Segments[*].n` stores the normal vector of each segment endpoint in x-y-z coordinates.
- `MC.Segments[*].b` stores the binormal vectors.
- `MC.Segments[*].n` stores the tangent vectors.

To print any of these tables to the screen, we use `mprint()`, as in:

```
mprint(MC.Segments[*].r)
```

Alternatively, we can save a given table to a file by typing, for example:

```
SaveTable("trajectory.txt", MC.Trajectory)
```

(The name of the file is our choice.) One further list that is useful after a `MC.P()` command is `MC.P_weights`, which contains the log-probability weight for each chain that hit its target.

Finally, there are three commands that save data related to the last `MC.P()` call.

- `MC.SaveHitSegments()` stores the endpoints, or stored midpoints (passing an optional segment number), of the hits resulting from a given run. One use for this is to look at the distribution of *all* generated chains, by running this command after a `MC.P()` call that had `sample_all = true` set.
- `MC.SaveHitTrajectories(table #)` stores the weighted distributions (shift, slide, etc.) averaged over all segments of the chains that hit their target. This should be 1) close to the biased sampling distrbiution if one is being used, and 2) should not be noisy (otherwise the `MC.P_weights` distribution is dominated by a few samples). An optional second parameter specifies the set of distributions if it is a sequence-dependent model: e.g. `MC.SaveHitTrajectories(shift)`.
- `MC.PropMode(mode #)` pertains to the perturbative high-energy methods. It propagates using the minimum-energy trajectory plus the given eigenmode with some amplitude (which can be adjusted using an optional second parameter). Saving the output of two of these calls allows us to visualize the mode in a plotting program. Use only after `MC.P()` with *zero* samples.

```
> MC.PropMode(1, 0)

> SaveTable("min_E.txt", MC.Segments[*].r)

> MC.PropMode(1)

> SaveTable("with_mode_0.txt", MC.Segments[*].r)
```

4.4 Estimating computation time and memory

Some calculations can really blow up in terms of time and memory requirements, so it's helpful to know in advance how intensive a calculation will be. We can get a very rough estimate using the `Comps()` function with the command(s) to test in parentheses after a semicolon:

```
> Comps( ; MC.InitBP(), &
           MC.P(100, 1e5) )

{ 1.08552e+07, 36.1719 }
```

The first number that `Comps()` returns is the memory usage in bytes; the second number is the time in seconds that the operation is expected to take. Of the numerical routines, only `MC.Init()/InitWormlike()/InitBP()` and `MC.P()` (when new chains are being generated) contribute to this estimator, since the other processes are comparatively very fast and use little memory. The program can only give a minimum time estimate for the perturbative method when it doesn't start in a minimum-energy conformation.

The expected time depends on the machine one is using, so before we try this feature on a new machine, we should calibrate the estimator by running `Comps.Calibrate()` on any command in

parentheses that takes more than a few seconds or so to run (in order to get an accurate estimate of the machine's speed).

```
> Comps.Calibrate( ; MC.InitBP(), MC.P(100, 1e5) )
```

We can also measure the true time taken by some operation using `Time()`. This is basically a stopwatch function: it actually runs the operations in question and just reports after the fact how long they took.

```
> Time( ; MC.InitBP(), MC.P(100, 1e5) )

{ 45.0466, 45 }
```

Both times are in seconds: the first time is based on clock ticks, the second by looking at the computer's time-of-day record. The first method is more accurate, but the second method is included because the author has encountered some machines where the first method gives wildly inaccurate results.

4.5 Random numbers

Monte Carlo sampling produces chains with random trajectories, which makes the exact results of a sampling hard to reproduce. Even if the program were run twice and exactly the same commands were entered in the same order, the Monte Carlo outcomes would be different. The random sequence generator, however, isn't really random: it generates each new number by performing some complicated operation on the last number in the sequence. Therefore, it is possible to reproduce Monte Carlo results by setting the 'seed' (the prior number in the random sequence) to some original value, and entering the same sequence of commands in the same order subsequent to having that value. We can do this in the Wormulator in the following way:

```
> MC.InitWormlike()

> InitRandom(100)

> MC.P(100, 1e4; at(r, { 0, 0, 0 }))

{ 0.0027693, 0.000227736, 116 }

> MC.P(100, 1e4; at(r, { 0, 0, 0 }))

{ 0.00288866, 0.000197586, 121 }

> InitRandom(100)

> MC.P(100, 1e4; at(r, { 0, 0, 0 }))

{ 0.0027693, 0.000227736, 116 }
```

In case we want to reproduce a result where we didn't set the random seed, we need to type "`random_seed`"—which tells us the number in the sequence when the program was first run—then reinitialize the sequence generator using that number and then repeat all of our calculations. For consistency all example runs above were done following `InitRandom(0)`.

References

[1] M. El Hassan and C. Calladine. The assessment of the geometry of dinucleotide steps in double-helical DNA; a new local calculation scheme. *Journal of molecular biology*, 251(5):648–664, 1995.

[2] W. Olson, A. Gorin, X. Lu, L. Hock, and V. Zhurkin. Dna sequence-dependent deformability deduced from protein–dna crystal complexes. *Proceedings of the National Academy of Sciences of the United States of America*, 95(19):11163, 1998.

[3] A. J. Spakowitz. Wormlike chain statistics with twist and fixed ends. *EPL (Europhysics Letters)*, 73(5):684–690, 2006.

[4] A. J. Spakowitz and Z.-G. Wang. End-to-end distance vector distribution with fixed end orientations for the wormlike chain model. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 72(4):041802, 2005.

[5] Y. Zhang and D. Crothers. Statistical mechanics of sequence-dependent circular DNA and its application for DNA cyclization. *Biophysical Journal*, 84(1):136–153, 2003.