bob v2.0 : A code to compute linear rheology of Branch-On-Branch polymers

May 12, 2007

Contents

Chapter 1

Overview

BOB calculates the linear rheological response of polymer melts of arbitrary architecture using extended *tube model*. This chapter gives a brief overview about how to install the code, how to get help and how to use the code.

1.1 Installation

The progrm should be easily imported to any operating system with a native c++ compiler. For any flavor of Unix, download the current version bob_2.0.tar.gz and use gunzip and untar to retrieve the directory structure. Type ./configure from the directory bob_2.0. If it does not find c++ compiler in the current path, it will prompt you to input the c++ compiler. Next it creates the binaries in bob_2.0/bin. It also creates paths required to access the binary files and man files and prompts you if you want to add them in your .cshrc. You can also compile the codes by going to the directory `bob_2.0/code/src/obj` and typing make. You may need to change the compiler name/path in the file compname in the same directory.

For windows platform, download the executable (it has been compiled via MinGW/gcc route and tested on windows-XP). You can also download the source files and compile under MinGW/MSYS (http://www.mingw.org) or Cygwin/X (http://x.cygwin.com) in the same fashion as for Unix machines. If you are willing to put in some effort in resolving the paths of the different components, you can also use any c++ compiler that you happen to have.

1.2 Usage

Assuming that bob_2.0/bin is in your $path, you are ready to use bob. **bob** starts a text based interface which asks you some questions to find out what it is supposed to do. **bob** $--$**help** gives you a brief help on usage and directs you to more detailed man pages. **bob** $--$**version** shows the version information.

If you are using the code repeatedly, the easier option will be running in batch mode. **bob -b** starts a batch mode where the input is read from a file instead of the terminal. Default input file is *inp.dat*. You can supply a different input file, ex. *myinp.dat* by typing **bob -b -i myinp.dat**. **bob** can create a number of simple and some not-so-simple polymer architectures. You can also supply a configuration file with information about the polymers you want to consider by typing **bob -b -c myconf.dat** - the default is *polyconf.dat*. Details about the formats for input file and configuration file appears later in this document.

1.3 Issues

- **bob** can not handle cyclic molecules or gels. If any of the polymers remain unrelaxed after $10^{20}\tau_e$ (which amounts to centuries for most polymers), the program is instructed to give up with an warning message.

- The prefactor for compound arm retraction probably need some consideration. This has to wait for data on monodisperse polymers with branch-on-branch architecture.

1.4 Credits

This program was developed with a funding from EPSRC.
Internally it uses a Mersenne Twister random number generator:
`http://www-personal.engin.umich.edu/~wagnerr/MersenneTwister.html`
Special thanks to Jorge Ramirez for reporting several bugs which have been corrected in this version.

1.5 Bug reports

If you discover any bugs please report to `chinmaydas@yahoo.com`. Also you can report any additions you make to the code and want it to be considered in future versions to the above mentioned address.

1.6 Reference

"Computational linear rheology of general branch-on-branch polymers", Chinmay Das, Nathanael Inkson, Daniel J. Read, Mark A. Kelmanson and T. C. B. McLeish; J. Rheol. **50**, 207-234 (2006).

1.7 Changes in version 2.0

- For comb polymers the input format for reading the number of arms has been corrected. The previous version generated linears instead of comb.
- Six arm star had a spurious zero length external branch in the previous version.
- The analysis of the generated polymers has been moved before the relaxation. Wrong placement of this function call often crashed Ver 1.0 (though before the crash, it should have generated the rheological data.).
- The previous version did not include the modification of arm retraction due to contribution from inner arms till the point relaxation reached the first branch-point.

  Some of these bugs were only on the released version (For JOR paper the polymers were generated with separate routines and some of the bugs got introduced while merging the different codes). The effect of modifying the retraction for the outermost arm between the first branch-point collapse and relaxing till that branch-point turns out to be insignificant for the results used in the JOR paper.
- New in-built polymer class `Comb_fxd` - this generates comb polymers with a fixed number of side-arms.
- New polymer type `UDF` - this is a dummy routine which the user can modify to take advantage of the code structure and generate new polymer classes. (details in sec. 3.4)

Chapter 2

Using bob

This chapter details the usage of bob including the formats for input and polymer configuration file.

2.1 Input

bob can be used interactively or in a batch mode (with the flag -b). For using in batch mode the input should be in a file *inp.dat* in the same directory from which you start the run. You can specify a different input filename (not longer than 70 characters) with the flag -i.

The structure of the input file is given here. For interactive mode, you require the same information - but you will be led through a number of questions and prompted to type in the information.

2.1.1 Memory

The first line of the input file has maximum number of polymers and maximum number of segments you want to consider. Linears are considered as made up of two segments. So, the maximum number of segments should be atleast twice the maximum number of polymers. These two numbers are used to allocate memory for representing the polymers. If you specify less than what you use, the program will behave unpredictably.

2.1.2 Relaxation parameters

The second line has a single double precision number : dynamic dilation exponent α. Different scaling arguments have been used to fix α as 1 or alternatively 4/3. The different parameters used in the model probably are not independent. In our calculations we used a value of 1.

A number of assumptions are required to model the relaxation process. The third entry is an integer : if it is 1, a predetermined choice for the assumptions will be made. Any other integer will require detailed inputs.

Here we are assuming that you wanted to provide detailed input. In that case, the next line has two double precision numbers and two integers : R_L, p^2, $pref_{mode}$ and $Rept_{scheme}$. Each of these are discussed below in some details.

R_L models when a star arm becomes disentangled. When the (dilated) length left to retract becomes smaller than R_L - the arm is considered to be disentangled and it relaxes completely. Similarly, if the length to reptate becomes smaller than $2R_L$, the linear reptates instantaneously. Traditionally R_L is considered to be zero.

When a side-arm collapses, the branch-point associated with the side-arm diffuses with hop-size pa - where p is a scalar number to be fixed from observation and a is the tube diameter. The input requires value of p^2. For $R_L \approx 0$, a value of 1/40 was found to model a diverse ensemble of polymer rheology data correctly.

For a compound arm retraction, the prefactor in the retraction time is not yet known. We handle this heuristically : If you set $pref_{mode} = 0$, the code uses the same prefactor as the outer-most arm. For $pref_{mode} = 1$ the same form of prefactor as a simple arm is used, but all instances of arm length is replaced by the effective arm length (for definition of this effective arm length look at JOR paper). $pref_{mode} = 2$ uses the full effective friction from collapsed side-arms. Possibly an interpolation from mode 0 to mode 2 is required to model deep retraction. For time being, $pref_{mode} = 1$ is found to give reasonable agreement for some comb molecules and metallocene-catalyzed branched polyethylene resins. This is the default choice.

For reptation, $Rept_{scheme} = 1$ and 2 uses reptation in thin and current tube respectively. Current tube performs badly and thin tube is preferred for most cases (the branch-point friction contribution is recast to the thin tube from a tube diameter associated with the branch-point withdrawal time scale). However, for extreme binary blends, thin tube reptation will fail. $Rept_{scheme} = 3$ and 4 tries to handle that. For $Rept_{scheme} = 3$, reptation is considered to happen in a tube diameter from the past where a linear molecule was able to reptate by a fixed amount. While, $Rept_{scheme} = 4$ considers reptation from time when a linear molecule was able to reptate by a fixed fraction of its length. For $Rept_{scheme} = 3$ or 4, the next entry is a double precision number $Rept_{amount}$ which tells what length (or fraction of chain length) the linears should be able to reptate at which time the tube diameter is stored and used in reptation. For $Rept_{scheme} = 3$, a value of unity for $Rept_{amount}$ has been used in literature. Probably more detailed analysis is wanted here. The default choice is reptation in thin tube.

We follow relaxation in time starting from a small value t_0 and taking snapshots at multiplicative time steps with step size multiplied by m at each step. The next entry are t_0 and m. Default values are $t_0 = 1e-4$ and $m = 1.002$ - these are small enough to not affect the relaxation due to discrete time-steps and large enough to produce results in reasonable run

time.

2.1.3 Material specific parameters

Next two lines in the input file requires material specific parameters. The first of these needs mass of a monomer M_0 (in atomic units), ex. M_0(PE)= 28.0; Number of monomers in an entanglement length N_e, ex. N_e(PE)$\approx$ 35.0 and mass-density of the polymer ρ in (Kg/m^3), ex. ρ(PE)$\approx$ 785.0.

The next line contains the entanglement time τ_e (in seconds) and the temperature (Kelvin).

2.1.4 Polymer configuration

The next line contains number of components to be considered. If number of component is 0, the program reads saved configuration from a file (unless instructed otherwise by using -c flag, from *polyconf.dat* from the run directory).

If you consider n component blends, there will be subsequent n blocks detailing the components. Each of them starts with a double precision entry : weight fraction of the current component. The next entry are two integers. The first is number of polymers of the current component. The second integer selects the polymer type : 0(linear), 1(star), 2(asymmetric star), 3(H), 4(Comb), 5(Cayley tree), 6(Metallocene-catalyzed PE), 7 (Comb with fixed number of arms), 8(UDF). Depending on your choice about the polymer, you will need to provide more information. The choice UDF is for user defined polymer which can be integrated in the program. For details about how to do that, look at the end of this document.

Linear

The first entry for linear block is type of arm you want. 0 considers strictly monodisperse arms. 1,2,3,4 draws the arms from Gaussian, Lognormal, living polymer ensemble and Flory distribution respectively. Living polymer ensemble is a Poisson distribution with the amount of monomer acting as a single unit being determined to reflect the experimentally determined polydispersity.

In batch mode, for any of the above choice you need to supply weight-averaged mass (M_W) for the molecules and polydispersity index (PDI). PDI is required but neglected for monodisperse case.

Star

Once again, the first entry is arm type. The second entry is M_W and PDI. Here M_W is the weight of a single arm and not the whole molecule. The

third entry is the number of arms. The built in polymer generation routine can create only 3, 4 or 6 arm stars.

Asymmetric star

Only three arm asymmetric stars are considered : two of the arms have the same length. First line has the arm type of the two similar arms. Second line contains M_W and PDI for those two arms. The next two lines contain the same information for the odd arm.

H

arm type, M_W and PDI for the side arms. Next the same informations for the cross bar.

Comb

Backbone armtype, backbone M_W and backbone PDI. Next side-arm armtype, M_W, PDI and average number of side arms. The program assumes that each monomer has the same probability of joining with a side arm. Thus a random interger is drawn from a Poisson distribution with the mean given by the supplied number of side arms. This many arms are attached to a backbone at random positions.

Cayley

Number of generations (0 signifies a star). For each generation, arm type, M_W and PDI

M-PE

M_W and number of branches per molecule b_m. Note that there is no information on arm type in this case. This is because arms are always sampled from Flory distribution. Also M_W refers to mass of a molecule and not segments.

Comb with fixed number of arms

Backbone armtype, backbone M_W and backbone PDI. Next side-arm armtype, M_W, PDI and integer number of side arms. The side-arms are connected at random positions on the backbone. Thus if the segments are mono-disperse, all the molecules will have the same mass. But the relaxation times will be different depending on the exact positioning of the side-arms.

UDF

The dummy code which is distributed with this program expects 9 lines each with armtype, arm M_W and PDI. Details about how to implement a more useful polymer type of your choice is provided in sec. 3.4. The input requirement will depend on the actual implementation.

2.1.5 Sample input files

Cayley tree of generation two

The lines are commented below in c++ style. In actual input file they should not be there.

```
100  5000  //maximum polymer=100, maximum segment = 5000
1.0        //Alpha = 1.0
0          //providing detailed choice
1e-5 0.025 1 1 //R_L, p^2, prefactor mode, thin tube reptation
1e-4 1.002  // t_0 and m
28.0 30.0 800.0   // M_0, N_e and mass density
1e-8 350     // tau_e and temperature
1           // single component
1.0         //occupies weight fraction unity
10  5       // actual number of polymers 10 of type 5(Cayley tree)
2           // generation 2
2 10000 1.01 // gen 0: inner star: lognormal, M_W = 10000, PDI=1.01
2 8000 1.02  // gen 1: lognormal, M_W=8000, PDI=1.02
0 5000 0.0   // gen 2: Monodisperse, M_W=5000, PDI is ignored
```

We can let the program use default values for some of the parameters:

```
100  5000
1.0
1          //Use default choice
28.0 30.0 800.0
1e-8 350
1
1.0
10  5
2
2 10000 1.01
2 8000 1.02
0 5000 0.0
```

We can store the polymer configuration in a file (default file *polyconf.dat*) and read from the file :

```
100  5000
1.0
1
28.0 30.0 800.0
1e-8 350
0         // read stored configuration
```

Cayley tree of generation two blended with 4 arm stars

Suppose we want to find out how Cayley tree polymers considered before behave when diluted with some 4 arm stars. For example let us consider 30% Cayley tree with 70% stars:

```
100  5000  //maximum polymer=100, maximum segment = 5000
1.0        //Alpha = 1.0
1          //use default choice
1e-8 350     // tau_e and temperature
2           // two components
0.30         //First component occupies 30% weight
10  5       // actual number of polymers 10 of type 5(Cayley tree)
2           // generation 2
2 10000 1.01 // gen 0
2 8000 1.02  // gen 1
0 5000 0.0   // gen 2
0.70        //second component 70%
15  1       // 15 polymers of type star
2           // stars arms from lognormal distribution
30000 1.01  // M_W and PDI of star arms
4           // number of arms
```

The number of polymers of different components need not have any relationship with the weight fraction. The generated polymers are properly rescaled to represent the supplied weight fraction in the relaxation process and calculation of the rheological response.

2.2 Output and files

So far we have talked about the input file and the polymer configuration file. These file names you can choose by using flags while running the code. The program also creates a file called *info.txt* which has the informations from different stages of the execution. It contains the choice of parameters used, the kind of polymers selected, a descriptive analysis about the topology of the polymers considered and few numbers like zero-shear viscosity.

If some of the molecules fail to relax, information about it is dumped in a file called *debg.dat*.

supertube.dat contains ascii data with time t, unrelaxed fraction ϕ, supertube fraction ϕ_{ST} and actual unrelaxed fraction ϕ_{true}.

gt.dat contains t and $G(t)$.

gtp.dat contains frequency ω, $G^{'}(\omega)$ and $G^{''}(\omega)$.

A file called *bobsv* is created which contain just enough information to compute $G^{'}(\omega)$ and $G^{''}(\omega)$ for the polymer ensemble considered with out going through the relaxation process again.

Without tinkering with the source code, you can not change these output filenames. Any file with same names in the execution directory will be overwritten.

2.3 Polymer configuration file

The first line of polymer configuration file should have a string naming the configuration. The string should be of length less than 10 characters.

The second line is a double precision number giving N_e. The segment lengths are stored in units of M_e - while M_0 is fixed, N_e is a fit parameter. Hence N_e appears here which enables one to get the actual segment lengths if one wishes to.

The third line is the number of polymers in the file.

Each polymer starts with number of segments n_s and then n_s lines of description of the segments. The segment descriptions are segments connected on the left and on the right, the segment mass (in units of M_e) and weight fraction of the segment in the whole ensemble.

For each polymer we number the segments from zero upwards. Each segment has four connectors two for connecting on the left (L_1 and L_2), two for the right(L_1 and L_2). The easiest way to input the values is by drawing the polymer on a paper, numbering them and noting the connection. If any of the connectors are not connected, they are *earthed* by putting a -1.

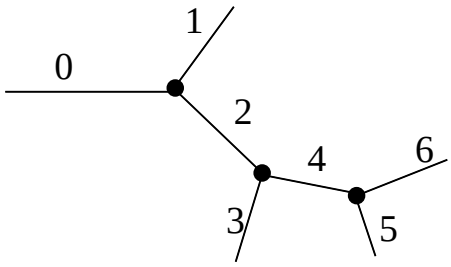
In what follows, we consider the mass of nth segment as `m_n` and the total mass of *all* the polymers considered is M (both in units of M_e).

First let us consider a linear molecule:

The configuration is represented as:

```
2 // number of segments
-1 -1  1 -1 l_0 m_0/M  // 0 is connected to 1 by R_1
 0 -1 -1 -1 l_1 m_1/M  // 1 is connected to 0 by L_1
```

Slightly more complicated molecule, a comb:




```
7 // number of segments
-1 -1  1  2  m_0 m_0/M   // 0 is connected to 1 (2) by R_1 (R_2)
 0  2 -1 -1  m_1 m_1/M   //1
 0  1  3  4  m_2 m_2/M   //2
-1 -1  2  4  m_3 m_3/M   //3
 2  3  5  6  m_4 m_4/M   //4
 4  6 -1 -1  m_5 m_5/M   //5
 4  5 -1 -1  m_6 m_6/M   //6
```

Since M is the summed mass of all the polymers, the sum of the 6^th column over all the polymers adds up to 1. For blends, you can multiply entries of this column corresponding to different components by different factors. This allows to work with number of molecules which need not have relationship with weight fraction and still gives the correct results. The reasoning for this complication is that you can represent a majority component with monodisperse architecture as a single molecule, while the minority component may have a large polydispersity, requiring large number of molecules to faithfully represent the molar mass distribution.

Chapter 3

Program structure

The working of **bob** can be subdivided in distinct groups. Firstly a set of files in subdirectory *parser* does exactly what implied - it parses the arguments passed from command line and decides about what to do. Next a set of routines in subdirectory *UI* finds out either interactively or from an input file about the actual polymers to consider. The files in *polygen* generates the polymers. Routines in directory *dyn* follows the relaxation over time and *calc* routines calculates the relaxation spectrum.

3.1 Datatype

subdirectories *include* and *arm_pool* :

Main types used:

```
class arm {
  public :
  int L1,L2,R1,R2,up,down;

  double arm_len, vol_fraction;
  bool relaxing, free_end, compound;

  int relax_end, nxt_relax;

/* The following are used only for free ends */
  int free_up, free_down, nxtbranch1, nxtbranch2;
  double z, dz, pot, gamma2, tau_K;

  double arm_len_eff, arm_len_end, deltazeff;
  double zeff_numer, zeff_denom;
```

```
  bool collapsed, ghost, prune, freeze_arm_len_eff;
  double tau_collapse, phi_collapse, extra_drag, pot_int;
  int next_friction;
};
```

and

```
class polymer{
  public :
   int first_end, first_free, num_branch;
   bool alive, linear_tag;
   double relaxed_frac, ghost_contrib;
   bool rept_set;
   double phi_rept;
};
```

These two classes are used as global variables:

```
arm * arm_pool; int first_avail_in_pool;
polymer * branched_poly;
```

The routines in *arm_pool* are used to initialize and handle the contents of arm_pool.

A polymer is represented as a collection of `arms` with specific connectivity. The integer tag `first_end` stores the index of one of the arms. The arms themselves form a doubly connected ring using the tags `up` and `down`. Each arm can be connected to atmost two other arms on each end - the addresses of these arms are stored through the tags `L1, L2,` and, `R1, R2`. `L1` and `L2` are at the same end of the current arm. Absence of a connection is represented through a special integer `-1`.

If one end of a given arm is not connected to any other arm, this starts with a true flag `free_end`. The `free_ends` also form a doubly connected ring and can be accessed through the tag `first_free` of the polymer. Arm retraction starts from the free ends; once through branch-collapse, an inner arm acquires the freedom to relax, the appropriate `free_end` becomes a `compound` arm with an effective arm length and an effective friction coefficient which takes care of the architecture (within certain approximations).

3.2 Parser

Find out if the mode is batch. If file names are supplied, open them. Else open default files. Main routine

```
int parser(int argc, char *argv[])
```

3.3 UI

The codes in this directory reads the input and uses function calls to appropriate polymer generation codes. Main routine:

```
void user_interface(void)
```

3.4 Polymer configuration

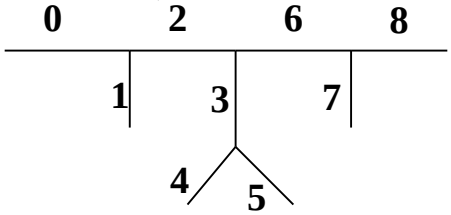
Input about type x is taken by *genx.cpp* and the actual connectivity and lengths are provided by *polygenx.cpp*. Try to keep the naming conventions for the routines out here.

3.4.1 Generating a new class of polymer

The model works with the assumption that we can create a faithful representation of the actual molecules in some coarse-grained sense. For extremely careful synthesis, the polymers can have simple structure like star or H polymers. For industrial polymers a modeling of the chemical reactions in the synthesis process itself is necessary. One example of such a reaction is metallocene catalyzed polyethylene which is implemented as one of the avilable polymer type. Even for model polymers, the different segments can have different molar mass distributions.

As a simple example we will design a polymer from scratch and embed it in the structure of the overall code. For simplicity the polymer has fixed topology, but the arms are polydisperse - possibly generated in separate reaction schemes, thus having different polydispersities. The files used for this user defined polymer (UDF) are *genUDF.cpp* and *polygenUDF.cpp*. These two files are included in the make file. So, if you make changes in these files to create a different polymer and recompile the code, the programme should handle the new polymer correctly.

For fixed connectivity, the simplest way is to draw the molecule on a piece of paper and decorate it with integers starting from zero. The molecule we are considering has 9 segments (this is in fact a generation 1 Cayley tree).



We assume that in the final product each of the molecules consist of exactly 9 segments connected as represented in the sketch. Thus, we need to read 9 different set of inputs, one for each arm; draw arm lenghs from appropriate distribution and connect them to create the polymers.

The code for UDF polymer is set to 8 and if either through interactive input or through input file, the function `get_poly_component()` is instructed to create UDF polymer, it calls `genUDF()`.

In *genUDF.cpp* the header line

```
#define UDF_segment_num 9
```

instructs the code to create space for 9 segments. If the run is in batch mode, the program reads in *arm_type, mass,* and *PDI* for each arm from the input file. Else it calls in a function

```
user_get_arm_type(&arm_type[i], &mass[i], &pdi[i])
```

to interactively find the input for each arm.

For actually generating the segments, we will use M_N while the input data are M_W. So, unless the segments are monodisperse, the M_W are divides by the polydispersity indices. Also at this stage we represent the polymer by number of monomers - so all the arm masses are divided by the monomer mass.

This polymer may be part of a blend. So, `genUDF(int ni, int nf)` gets called with two integers `ni` and `nf`. The calling routine will take care of supplying these integers correctly. `ni` is the first molecule of type UDF and there will be `(nf - ni)` molecules of this type.

The creation of the molecules are handled by `polygenUDF.cpp`, which retuns one polymer at a time. To construct a polymer, this routine calls `request_arm()` to get the address of the next arm. The length of the arm is set by calling `poly_get_arm(arm_type, mass, pdi)`, which returns a length from the distribution appropriate for the input variables (at this stage the arm length is divided by N_e to express the polymers in units of the entanglement molar mass).

The arms still remain unconnected. We use the `up` and `down` variables of the arms to create a ring (up of 0 is `UDF_segment_num-1`). Next the first of the arm is identified as `first_end` of `cur_poly`. Now for each arm `n0` we call `attach_arm(n0,n1,n2,n3,n4)`, where `n1, n2` are the arms connected to the left of `n0` and `n3, n4` are the arms connected to the right. (The left and right have no real meaning - the important part is that `n1` and `n2` are connected at the same side of `n0`.) When one end of the arm is free, we supply `-1` to signify the absence of connectivity.

A call to `poly_start(&cur_poly)` is necessary to set up different flags for the relaxation (which arms are allowed to relax from the beginning, if reptation is feasible to begin with $\cdots$) and the code returns the polymer.

You can change the codes in these two files to create different fixed architecture polymers, or with a little bit extra effort, polymers with polydispersity in the architecture too.

3.5 Relaxation

subdirectory *dyn*. Toplevel code *time_step.cpp*. It calls *retraction*, *extend_arm* and *reptation* in that order. For simple arms, the arm retraction inverts Kramers first passage time, interpolating with faster Rouse retraction at the chain ends through an interpolation scheme. For compound arms, where the friction is dominated by the branch points (with side arms fully relaxed at the current time), the problem is recast as a one dimensional Kramers problem through an effective arm length and friction of the free arm. For a longer description of the algorithm, see the JOR paper.

3.6 Calculation

Calcutate $G(t)$, $G^{'}(\omega)$, $G^{''}(\omega)$ using a discrete version of double reptation. The contribution from fast (forced) Rouse modes are calculated separately in both time and frequency space using analytical formula. The zero shear viscosity is calculated by extrapolating the frequency response to zero frequency through a least square fit from low enough frequncy results.