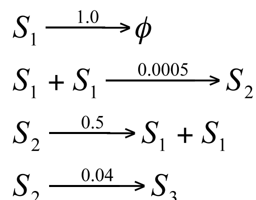
Examples of SSC simulations

1. Dimer Decay

Consider the following reaction scheme:



$$S_1 \xrightarrow{1.0} \phi$$
$$S_1 + S_1 \xrightarrow{0.0005} S_2$$
$$S_2 \xrightarrow{0.5} S_1 + S_1$$
$$S_2 \xrightarrow{0.04} S_3$$

We can encode it in SSC in the following manner (create text file dimer-decay.rxn and paste this text into it):

```
rxn x:S1 at 1.0 -> destroy x
rxn x:S1 y:S1 at 0.0005 -> destroy x; destroy y; new S2
rxn x:S2 at 0.5 -> destroy x; new S1; new S1
rxn x:S2 at 0.04 -> destroy x; new S3

new S1 at 100000
record S1
record S2
record S3
```

In order to compile the code (assuming that SSC is installed, see installation manual), simply run,

```
[bla@blabla]$ ssc dimer-decay.rxn
```

You should get the following message:

```
reading: dimer-decay.rxn...
expanding reactions...
expansion complete after 9 steps: 3 compounds and 4 reactions
simulator executable: dimer-decay
```

This tells you that executable simulator `dimer-decay` has been created and you can now run the simulation. But before actually running the simulation we want to learn about the network that SSC has actually created based on our dimer-decay.rxn input. In order to do so, we can use option `--dump-epxpanded=FILE` :

```
[bla@blabla]$ ssc –dump-expanded=network dimer-decay.rxn
```

After executing this command you will find that text-file network has been created with the following content:

```
Compounds:
  S1
  S2
  S3
Reactions:
  S1 at 1.0 -->
  S1 S1 at 5.0e-4 --> S2
  S2 at 0.5 --> S1 S1
  S2 at 4.0e-2 --> S3
Records:
```

```
 record S1
 record S2
 record S3
```
where all reactions are mentioned explicitly. In this simple example, we see that number of reactions is actually the same as the number of rxn lines in the initial code. This is because there is no combinatorial expansion in this chemical network (for the network with combinatorial expansion see next example)

We now have compiled executable file dimer-decay and checked that chemical network is read properly. In order to run the simulation we have to execute it with the appropriate flags. List of possible flags can be obtained by running executable with help flag –h:

```
[bla@blabla]$ ./dimer-decay -h
Options:
 -c FILE    read variable values from FILE
 -e T       end simulation after time T
 -s T       report trajectory mean and std. dev from time T until the end
 -d T       report amount probability distribution from time T until the end
 -t T       sample trajectory at interval T (0 = at every step)
            (default: report at the end of simulation only)
 -r N       initialize random number generator with seed n
 -R         report random seed used
 -v         show version information
 -h         show this help message
```

In the above example rate constants values are hardwired into the simulator code produced by SSC and can not be changed without recompiling the simulator file dimer-decay. Since all the parameters have been provided there is no need to use –c flag (see below for the axample with –c flag).

- Simplest way to run the simulator is

```
[bla@blabla]$ ./dimer-decay –e 1
```

which means that simulation should be stopped at 1 second and last values will be reported. The following lines will appear on the screen:

```
time    # S1    # S2    # S3
1.000018       8308    41829   1724
```

- In order to dump the output from screen to file you can use >> :

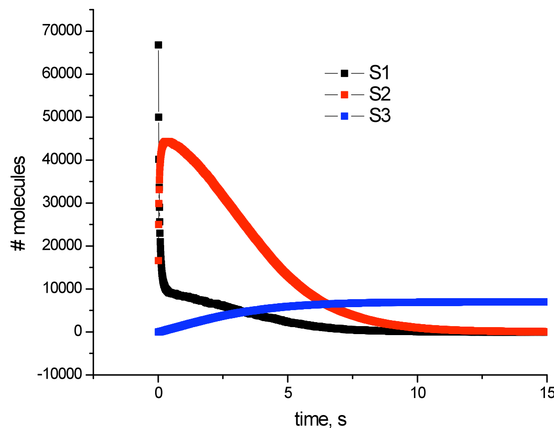
```
[bla@blabla]$ ./dimer-decay –e 1 >> res
```

In this case file res will be created containing same two lines

- In order to output concentrations of species

```
[bla@blabla]$ ./dimer-decay –e 10 –t 0.01 >> res
```

after plotting the content of file res with your favorite plotting program you see:



- If you need to determine average value and standard deviation during particular period of time, you can use option –s TIME.

```
[bla@blabla]$ ./dimer-decay –e 1 –s 0.01
time    mean S1 std. dev. S1    mean S2 std. dev. S2    mean S3 std. dev. S3
1.000000        11693.344569   7465.839577     42517.984423   3299.143436    813.002673        494.718499
```

In this example the averaging is done from 0.01 second to 1 second.

- SSC carries out stochastic simulations. Therefore, every time you run simulation the results you obtain are different. It is important sometime to have control over this randomness, which is achieved with options –r NUMBER and –R that allow manipulations with random numbers. Option –R prints out the seed of random number generator used for particular trajectory:

```
[bla@blabla]$ ./dimer-decay –e 1 –s 0.01 –R
Random seed: 1504910080
time    mean S1 std. dev. S1    mean S2 std. dev. S2    mean S3 std. dev. S3
1.000032        11669.866422   7573.517657     42478.163637   3337.886209    840.521778        496.966612
```

Running this simulation again with exactly same seed will reproduce results of simulations exactly:

```
[bla@blabla]$ ./dimer-decay –e 1 –s 0.01 –r 1504910080
time    mean S1 std. dev. S1    mean S2 std. dev. S2    mean S3 std. dev. S3
1.000032        11669.866422   7573.517657     42478.163637   3337.886209    840.521778        496.966612
```

While using different seed, obviously, gives slightly different result:

```
[bla@blabla]$ ./dimer-decay –e 1 –s 0.01 –r 1
time    mean S1 std. dev. S1    mean S2 std. dev. S2    mean S3 std. dev. S3
1.000007        11725.893253   7481.325239     42455.385369   3296.012889    838.716479        488.348794
```

- It is often important to know probability distribution of molecules at during some period of time. In order to learn that use option –d TIME which gives probability distribution of seeing x molecules of reported compound in the period between TIME and the end of simulation.

```
[bla@blabla]$ ./dimer-decay –e 1 –d 0.01 >>probdistr
```

Where probdistr contains 4 columns first containing number of molecules and the others containing the probabilities to observe that number of molecules S1,S2 or S3.

- It is not always convenient to have rate constant values hardwired into simulation code. Normally, one would like to perform some kind of parameter sensitivity

study for a given chemical network. In SSC rate constants can be defined as variables and then provided in separate easily changeable text-file.

Let's consider particular example. Change original .rxn file in the following way:

```
rxn x:S1 at k1 -> destroy x
rxn x:S1 y:S1 at k2 -> destroy x; destroy y; new S2
rxn x:S2 at k3 -> destroy x; new S1; new S1
rxn x:S2 at k4 -> destroy x; new S3

new S1 at initial
record S1
record S2
record S3
```

This file can still be normally compiled into the executable simulator. But in order to actually run the simulator, we need supplementary file, let's say param.dat, that would contain

```
k1 = 1.0
k2 = 0.0005
k3 = 0.5
k4 = 0.04
initial = 100000
```

Then, we can execute the simulation and use all aforementioned flags, for example:

```
[bla@blabla]$ ./dimer-decay –e 1 –c param.dat
time    # S1    # S2    # S3
1.000028        8035    38578   1532
```

2. EGFR receptors dimerization during the signaling

As an example of a more complicated problem we can turn to the simple model of EGFR receptor signaling. We want to illustrate the problem of combinatorial expansion in the simple way with the help of this model. Initial steps of EGFR signaling require binding of EGF to the EGF-receptor, which can dimerize with the other EGF-receptor provided that both are attached to EGF ligand.

Create file egfr.rxn with the following text in it:

```
new EGF at 1200000
new EGFR(Y1068="U") at 180000

-- # Ligand-receptor binding
rxn x:EGFR(l#,r#) y:EGF(r#) at 1.667e-6 -> x.l # y.r
-- # Ligand-receptor unbinding
rxn EGFR(l#1,r#) EGF(r#1) at 0.06 -> break 1

-- # Receptor-aggregation
rxn x:EGFR(l#1,r#) y:EGFR(l#2,r#) EGF(r#1) EGF(r#2) at 1.667e-6 -> x.r # y.r
-- # Receptor-disintegration
rxn EGFR(l#2,r#1) y:EGFR(l#3,r#1) EGF(r#2) EGF(r#3) at 0.06 -> break 1

-- # Transphosphorylation of EGFR by RTK
rxn x:EGFR(r#1,Y1068="U") EGFR(r#1) at 0.5 -> x.Y1068 = "P"

-- # Dephosphorylation
rxn x:EGFR(Y1068="P", y#) at 4.505 -> x.Y1068="U"

record EGFR(Y1068="P")
```

Now,executing

```
[bla@blabla]$ ssc egfr.rxn
reading: egfr.rxn...
expanding reactions...
expansion complete after 45 steps: 8 compounds and 16 reactions
simulator executable: egfr
```

Note that even though we have 6 rxn-lines specifying possible chemical reactions, actual number of chemical reactions in the system is 16. By running SSC with option - -dump-expanded :

```
[bla@blabla]$ ssc –dump-expanded=network egfr.rxn
```

we find the complete list of compounds and reactions in file network. For example, list of compounds is:

```
Compounds:
 EGF
 EGFR(Y1068="U")
 EGFR(Y1068="U", l#1) EGF(r#1)
 EGFR(Y1068="U", r#1, l#2) EGFR(Y1068="U", r#1, l#3) EGF(r#3) EGF(r#2)
 EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="U", r#2, l#3) EGF(r#3)
```

```
  EGFR(Y1068="P", l#1) EGF(r#1)
  EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="P", r#2, l#3) EGF(r#3)
  EGFR(Y1068="P")
Reactions:
  EGF EGFR(Y1068="U") at 1.6669999999999999e-6 --> EGFR(Y1068="U", l#1) EGF(r#1)
  EGFR(Y1068="U", l#1) EGF(r#1) at 6.0e-2 --> EGF EGFR(Y1068="U")
  EGFR(Y1068="U", l#1) EGF(r#1)
  EGFR(Y1068="U", l#1) EGF(r#1) at 1.6669999999999999e-6 --> EGFR(Y1068="U", r#1, l#2) EGFR(Y1068="U", r#1,
l#3) EGF(r#3) EGF(r#2)
  EGFR(Y1068="U", r#1, l#2) EGFR(Y1068="U", r#1, l#3) EGF(r#3) EGF(r#2) at 6.0e-2 (* 2) --> EGFR(Y1068="U", l#1)
EGF(r#1) EGFR(Y1068="U", l#1) EGF(r#1)
  EGFR(Y1068="U", r#1, l#2) EGFR(Y1068="U", r#1, l#3) EGF(r#3) EGF(r#2) at 0.5 (* 2) --> EGFR(Y1068="P", l#1,
r#2) EGF(r#1) EGFR(Y1068="U", r#2, l#3) EGF(r#3)
  EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="U", r#2, l#3) EGF(r#3) at 6.0e-2 (* 2) --> EGFR(Y1068="U", l#1)
EGF(r#1) EGFR(Y1068="P", l#1) EGF(r#1)
  EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="U", r#2, l#3) EGF(r#3) at 0.5 --> EGFR(Y1068="P", l#1, r#2)
EGF(r#1) EGFR(Y1068="P", r#2, l#3) EGF(r#3)
  EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="U", r#2, l#3) EGF(r#3) at 4.505 -->  EGFR(Y1068="U", r#1, l#2)
EGFR(Y1068="U", r#1, l#3) EGF(r#3)  EGF(r#2)
  EGFR(Y1068="P", l#1) EGF(r#1) at 6.0e-2 --> EGF EGFR(Y1068="P")
  EGFR(Y1068="U", l#1) EGF(r#1)
  EGFR(Y1068="P", l#1) EGF(r#1) at 1.6669999999999999e-6 (* 2) --> EGFR(Y1068="P", l#1, r#2) EGF(r#1)
EGFR(Y1068="U", r#2, l#3)  EGF(r#3)
  EGFR(Y1068="P", l#1) EGF(r#1)
  EGFR(Y1068="P", l#1) EGF(r#1) at 1.6669999999999999e-6 -->EGFR(Y1068="P", l#1, r#2) EGF(r#1)
EGFR(Y1068="P", r#2, l#3)   EGF(r#3)
  EGFR(Y1068="P", l#1) EGF(r#1) at 4.505 -->  EGFR(Y1068="U", l#1) EGF(r#1)
  EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="P", r#2, l#3) EGF(r#3) at 6.0e-2 (* 2) --> EGFR(Y1068="P", l#1)
EGF(r#1) EGFR(Y1068="P", l#1) EGF(r#1)
  EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="P", r#2, l#3) EGF(r#3) at 4.505 (* 2) --> EGFR(Y1068="P", l#1,
r#2) EGF(r#1) EGFR(Y1068="U", r#2, l#3)  EGF(r#3)
  EGF EGFR(Y1068="P") at 1.6669999999999999e-6 --> EGFR(Y1068="P", l#1) EGF(r#1)
  EGFR(Y1068="P") at 4.505 --> EGFR(Y1068="U")
Records:
  record
    EGFR(Y1068="P"),
    2 * EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="P", r#2, l#3) EGF(r#3),
    EGFR(Y1068="P", l#1) EGF(r#1), or
    EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="U", r#2, l#3) EGF(r#3)
```

Because of the templating in the SSC code one line of SSC code corresponds to several reactions, for example:

```
rxn x:EGFR(Y1068="P") at 4.505 -> x.Y1068="U"
```

stands for the following reactions

```
  EGFR(Y1068="P") at 4.505 --> EGFR(Y1068="U")
  EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="P", r#2, l#3) EGF(r#3) at 4.505 (* 2) --> EGFR(Y1068="P", l#1,
r#2) EGF(r#1) EGFR(Y1068="U", r#2, l#3) EGF(r#3)
  EGFR(Y1068="P", l#1) EGF(r#1) at 4.505 -->  EGFR(Y1068="U", l#1) EGF(r#1)
  EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="U", r#2, l#3) EGF(r#3) at 4.505 --> EGFR(Y1068="U", r#1, l#2)
EGFR(Y1068="U", r#1, l#3) EGF(r#3) EGF(r#2)
```

The reason is that when specification is not explicitly mentioned in rxn line, therefore the reaction is possible for any specification. So, in this example

```
rxn x:EGFR(Y1068="P") at 4.505 -> x.Y1068="U"
```

binding site **r** and **l** are not explicitly mentioned, therefore both dimerized and non-dimerized EGFR can get dephosphorylated.

Similarly, since we are interested in total amount of phosphorylated EGFR, whether they are a part of bigger complex or individual proteins, it is convenient to use line

```
record EGFR(Y1068="P")
```

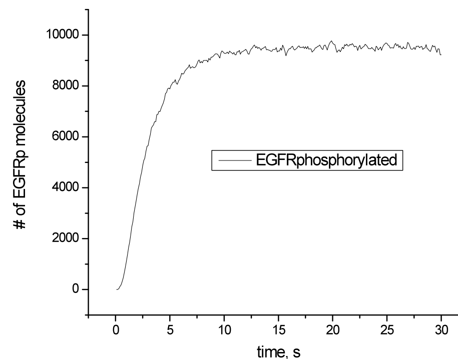
since absence of binding sites specification allows to compute EGFRp in all of the following situations:

```
EGFR(Y1068="P"),
2 * EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="P", r#2, l#3) EGF(r#3),
EGFR(Y1068="P", l#1) EGF(r#1), or
EGFR(Y1068="P", l#1, r#2) EGF(r#1) EGFR(Y1068="U", r#2, l#3) EGF(r#3)
```

Next, we run the simulation:

```
[bla@blabla]$ ./egfr –e 30 –t 0.1 >>dat
```

This creates file dat which contains simulation trajectory (number of EGFRp vs time). After plotting we get:



We see immediately that number of EGFRp molecules reaches steady state after ~15 s. We can obtain more precise value of the average amount of EGFRp by running:

```
[bla@blabla]$ ./egfr –e 300 –s 15
time    mean EGFR(Y1068="P")    std. dev. EGFR(Y1068="P")
300.000011    9487.382022    93.114016
```

Similarly, we can obtain probability distribution of EGFRp in steady state by:

```
[bla@blabla]$ ./egfr –e 300 –d 15 >>distr
```

After plotting the content of the file distr we have: