How to use dbinom, dpois, dnbinom2, dgamma, rbinom, rpois, rnbinom2, and rgamma in TMB

This is a basic demonstration of how to use the binomial, Poisson, negative binomial, and gamma distributions for fitting and simulating in TMB. In this example, we only look at the version of the negative binomial that is parameterized by it's mean and variance in TMB because it is more commonly useful in ecology, `dnbinom2` and `rnbinom2`. For each distribution, we simulate observations in R, fit a model in TMB, simulate data in TMB from the fitted model, and then plot the observations with the simulations from the fitted model for comparison.

Simulate data in R

```r
n=1000
set.seed(1)
B=rbinom(n, size=10, prob=0.3)
P=rpois(n, lambda=10)
NB=rnbinom(n, mu=10, size=10)#variance = 20
G=rgamma(n, shape=9, scale=0.5)
```

Fit a model in TMB

The code below is saved in a file named `dists4.cpp` because it deals with 4 distributions. Lines 5 to 8 read in the observed data. Lines 10 to 27 initialize parameters and transform them to the appropriate scales. Fitting on the log scale (e.g. `log_lambda`) constrains the parameter to be positive. Fitting on the logit scale (e.g. `logit_prob`) constrains the parameter to be between 0 and 1. Lines 29 to 33 calculate the negative log-likelihood of the four sets of observations.

Because they are enclosed by `SIMULATE{}`, lines 35 to 48 describe what to do when in simulation mode. In TMB, the functions `rbinom`, `rpois`, `rnbinom`, and `rgamma` return objects of the same dimension as their arguments. In this simple case, our arguments are all scalers, but this is not usually the case. Therefore, to simulate the right number of values for each distribution, we use a for loop on lines 37 to 43. Then we return the simulated values on lines 44 to 47. These are returned as a list, as you will see by the end of this section.

```cpp
1  #include <TMB.hpp>
2  template<class Type>
3  Type objective_function<Type>::operator() ()
4  {
5    DATA_VECTOR(B);
6    DATA_VECTOR(P);
7    DATA_VECTOR(NB);
8    DATA_VECTOR(G);
9
10   PARAMETER(logit_prob); //for binomial
11   Type prob=invlogit(logit_prob);
12   ADREPORT(prob);
13   PARAMETER(log_lambda); // for Poisson
14   Type lambda=exp(log_lambda);
15   ADREPORT(lambda);
16   PARAMETER(log_mu); // for negative binomial
```

```
  Type mu=exp(log_mu);
  ADREPORT(mu);
  PARAMETER(log_var); // for negative binomial
  Type var=exp(log_var);
  ADREPORT(var);
  PARAMETER(log_shape); // for gamma
  Type shape=exp(log_shape);
  ADREPORT(shape);
  PARAMETER(log_scale); // for gamma
  Type scale=exp(log_scale);
  ADREPORT(scale);

  Type nll=0;
  nll -= sum(dbinom(B, Type(10), prob, true));
  nll -= sum(dpois(P, lambda, true));
  nll -= sum(dnbinom2(NB, mu, var, true));
  nll -= sum(dgamma(G, shape, scale, true));

  SIMULATE
  {
    for(int i=0; i<B.size(); i++)
    {
      B(i) = rbinom(Type(10), prob);
      P(i) = rpois(lambda);
      NB(i) = rnbinom2(mu, var);
      G(i) = rgamma(shape, scale);
    }
    REPORT(B);
    REPORT(P);
    REPORT(NB);
    REPORT(G);
  }
  return nll;
}

```

We can compile and fit the model in the standard way.

```
library(TMB)
compile("dists4.cpp")
```

```
## Note: Using Makevars in /Users/molliebrooks1/.R/Makevars
```

```
## [1] 0
```

```
dyn.load(dynlib("dists4"))
obj=MakeADFun(data=list(B=B, P=P, NB=NB, G=G),
            parameters=list(
              logit_prob=0,
              log_lambda=log(5),
              log_mu=log(5),
              log_var=log(30),#var must be greater than mu
              log_shape=log(5),
              log_scale=log(0.3)
            ),
            DLL="dists4",
            silent=TRUE)
opt = nlminb(obj $par, obj $fn, obj $gr)
sdr=sdreport(obj)
summary(sdr, "report")
```

```
##          Estimate  Std. Error
## prob    0.3016998 0.004589957
## lambda  9.9349801 0.099674362
## mu     10.2169962 0.143902092
## var    20.7078343 1.040833668
## shape   8.7453822 0.383874657
## scale   0.5144679 0.023242744
```

These estimated parameters are not too far from the values used to simulate the observations above. Then to get simulated values from our fitted model, we use the `simulate()` function which is now part of `obj`.

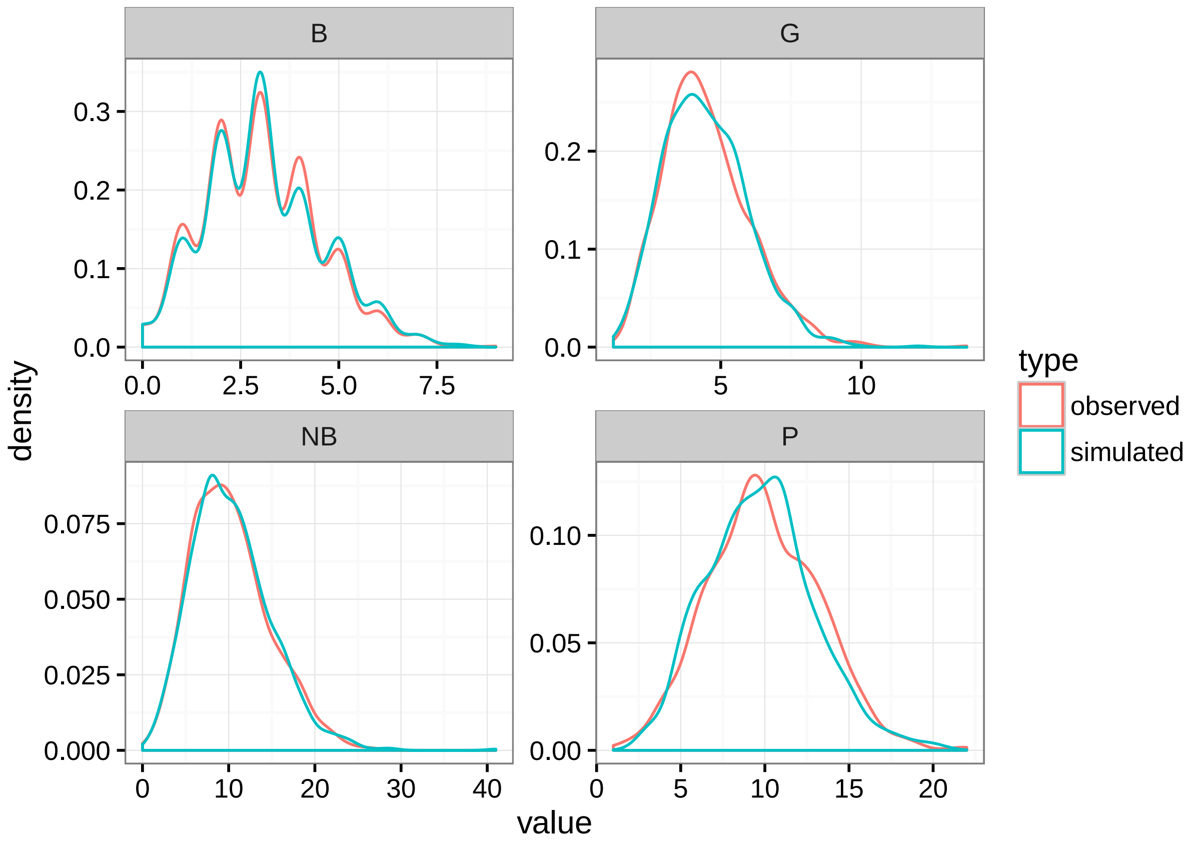
```
sim=obj$simulate()
str(sim)
```

```
## List of 4
##  $ B : num [1:1000] 2 4 5 4 2 3 3 4 6 3 ...
##  $ NB: num [1:1000] 8 10 13 8 7 3 13 3 6 15 ...
##  $ G : num [1:1000] 3.34 3.41 2.66 4.74 3.97 ...
##  $ P : num [1:1000] 11 11 15 13 12 10 9 9 10 11 ...
```

You can see that it returns a list with one element for each object that we reported in lines 44 to 47 of the C++ code above.

Compare simulated values from the fitted model to the observed data

Check that the observed and simulated values have similar distributions.


B
G
NB
P
density
value
type
observed
simulated
0.0
0.1
0.2
0.3
2.5
5.0
7.5
5
10
0.000
0.025
0.050
0.075
0
20
30
40
0.00
0.05
0.10
15