GMSE: an R package for generalised management strategy evaluation

Supporting Information 2

A. Bradley Duthie, Jeremy J. Cusack, Isabel L. Jones, Erlend B. Nilsen, Rocío Pozo, O. Sarobidy Rakotonarivo, Bram Van Moorter, and Nils Bunnefeld

2017-11-10

# Extended introduction to the GMSE apply function (gmse\_apply)

The gmse\_apply function is a flexible function that allows for user-defined sub-functions calling resource, observation, manager, and user models. Where such models are not specified, predefined GMSE submodels 'resource', 'observation', 'manager', and 'user' are run by default. Any type of sub-model (e.g., numerical, individual-based) is permitted as long as the input and output are appropriately specified. Only one time step is simulated per call to gmse\_apply, so the function must be looped for simulation over time. Where model parameters are needed but not specified, defaults from GMSE are used. In this Supporting Information, we demonstrate some uses of gmse\_apply, and how it might be used to simulate myriad management scenarios *in silico*.

A simple run of gmse\_apply() returns one generation of GMSE using predefined submodels and default parameter values.

sim\_1 <- gmse\_apply();

For sim\_1, the default 'basic' results are returned as below, which summarise key values for all submodels.

print(sim\_1);

## $resource\_results  
## [1] 1076  
##   
## $observation\_results  
## [1] 997.7324  
##   
## $manager\_results  
## resource\_type scaring culling castration feeding help\_offspring  
## policy\_1 1 NA 110 NA NA NA  
##   
## $user\_results  
## resource\_type scaring culling castration feeding help\_offspring  
## Manager 1 NA 0 NA NA NA  
## user\_1 1 NA 9 NA NA NA  
## user\_2 1 NA 9 NA NA NA  
## user\_3 1 NA 9 NA NA NA  
## user\_4 1 NA 9 NA NA NA  
## tend\_crops kill\_crops  
## Manager NA NA  
## user\_1 NA NA  
## user\_2 NA NA  
## user\_3 NA NA  
## user\_4 NA NA

Note that in the case above we have the total abundance of resources returned (sim\_1$resource\_results), the estimate of resource abundance from the observation function (sim\_1$observation\_results, the costs the manager sets for the only available action of culling (sim\_1$manager\_results), and the number of culls attempted by each user (sim\_1$user\_results). By default, only one resource type is used, but custom subfunctions could potentially allow for models with multiple resource types. Any custom subfunctions can replace GMSE predefined functions, provided that they have appropriately defined inputs and outputs (see [GMSE documentation](https://cran.r-project.org/web/packages/GMSE/GMSE.pdf)). For example, we can define a very simple logistic growth function to send to res\_mod instead.

alt\_res <- function(X, K = 2000, rate = 1){  
 X\_1 <- X + rate\*X\*(1 - X/K);  
 return(X\_1);  
}

The above function takes in a population size of X and returns a value X\_1 based on the population intrinsic growth rate rate and carrying capacity K. Iterating the logistic growth model by itself under default parameter values with a starting population of 100 will cause the population to increase to carrying capacity in ca seven generations. The function can be substituted into gmse\_apply to use it instead of the predefined GMSE resource model.

sim\_2 <- gmse\_apply(res\_mod = alt\_res, X = 100, rate = 0.3);

The gmse\_apply function will find the parameters it needs to run the alt\_res function in place of the default resource function, either by running the default function values (e.g., K = 2000) or values specified directly into gmse\_apply (e.g., X = 100 and rate = 0.3). If an argument to a custom function is required but not provided either as a default or specified in gmse\_apply, then an error will be returned. Results for the above sim\_2 are returned below.

print(sim\_2);

## $resource\_results  
## [1] 128  
##   
## $observation\_results  
## [1] 113.3787  
##   
## $manager\_results  
## resource\_type scaring culling castration feeding help\_offspring  
## policy\_1 1 NA 110 NA NA NA  
##   
## $user\_results  
## resource\_type scaring culling castration feeding help\_offspring  
## Manager 1 NA 0 NA NA NA  
## user\_1 1 NA 9 NA NA NA  
## user\_2 1 NA 9 NA NA NA  
## user\_3 1 NA 9 NA NA NA  
## user\_4 1 NA 9 NA NA NA  
## tend\_crops kill\_crops  
## Manager NA NA  
## user\_1 NA NA  
## user\_2 NA NA  
## user\_3 NA NA  
## user\_4 NA NA

## How gmse\_apply integrates across submodels

To integrate across different types of submodels, gmse\_apply translates between vectors and arrays between each submodel. For example, because the default GMSE observation model requires a resource array with particular requirements for column identites, when a resource model subfunction returns a vector, or a list with a named element 'resource\_vector', this vector is translated into an array that can be used by the observation model. Specifically, each element of the vector identifies the abundance of a resource type (and hence will usually be just a single value denoting abundance of the only focal population). If this is all the information provided, then a 'resource\_array' will be made with default GMSE parameter values with an identical number of rows to the abundance value (floored if the value is a non-integer; non-default values can also be put into this transformation from vector to array if they are specified in gmse\_apply, e.g., through an argument such as lambda = 0.8). Similarly, a resource\_array is also translated into a vector after the default individual-based resource model is run, should the observation model require simple abundances instead of an array. The same is true of observation\_vector and observation\_array objects returned by observation models, of manager\_vector and manager\_array (i.e., COST in the gmse function) objects returned by manager models, and of user\_vector and user\_array (i.e., ACTION in the gmse function) objects returned by user models. At each step, a translation between the two is made, with necessary adjustments that can be tweaked through arguments to gmse\_apply when needed. Alternative observation, manager, and user, submodels, for example, are defined below; note that each requires a vector from the preceding model.

# Alternative observation submodel  
alt\_obs <- function(resource\_vector){   
 X\_obs <- resource\_vector - 0.1 \* resource\_vector;  
 return(X\_obs);  
}  
  
# Alternative manager submodel  
alt\_man <- function(observation\_vector){  
 policy <- observation\_vector - 1000;  
 if(policy < 0){  
 policy <- 0;  
 }  
 return(policy);  
}  
  
# Alternative user submodel  
alt\_usr <- function(manager\_vector){  
 harvest <- manager\_vector + manager\_vector \* 0.1;  
 return(harvest);  
}

All of these submodels are completely deterministic, so when run with the same parameter combinations, they produce replicable outputs.

gmse\_apply(res\_mod = alt\_res, obs\_mod = alt\_obs,   
 man\_mod = alt\_man, use\_mod = alt\_usr, X = 1000);

## $resource\_results  
## [1] 1500  
##   
## $observation\_results  
## [1] 1350  
##   
## $manager\_results  
## [1] 350  
##   
## $user\_results  
## [1] 385

Note that the manager\_results and user\_results are ambiguous here, and can be interpreted as desired -- e.g., as total allowable catch and catches made, or as something like costs of catching set by the manager and effort to catching made by the user. Hence while manger output is set in terms of costs of performing each action, and user output is set in terms of action attempts, this need not be the case when using gmse\_apply (though it should be recognised when using default GMSE manager and user functions). GMSE default submodels can be added in at any point.

gmse\_apply(res\_mod = alt\_res, obs\_mod = observation,   
 man\_mod = alt\_man, use\_mod = alt\_usr, X = 1000)

## $resource\_results  
## [1] 1500  
##   
## $observation\_results  
## [1] 1814.059  
##   
## $manager\_results  
## [1] 814.059  
##   
## $user\_results  
## [1] 895.4649

It is possible to, for example, specify a simple resource and observation model, but then take advantage of the genetic algorithm to predict policy decisions and user actions. This can be done by using the default GMSE manager and user functions (written below explicitly, though this is not necessary).

gmse\_apply(res\_mod = alt\_res, obs\_mod = alt\_obs,   
 man\_mod = manager, use\_mod = user, X = 1000)

## $resource\_results  
## [1] 1500  
##   
## $observation\_results  
## [1] 1350  
##   
## $manager\_results  
## resource\_type scaring culling castration feeding help\_offspring  
## policy\_1 1 NA 10 NA NA NA  
##   
## $user\_results  
## resource\_type scaring culling castration feeding help\_offspring  
## Manager 1 NA 0 NA NA NA  
## user\_1 1 NA 72 NA NA NA  
## user\_2 1 NA 70 NA NA NA  
## user\_3 1 NA 70 NA NA NA  
## user\_4 1 NA 70 NA NA NA  
## tend\_crops kill\_crops  
## Manager NA NA  
## user\_1 NA NA  
## user\_2 NA NA  
## user\_3 NA NA  
## user\_4 NA NA

## Running GMSE simulations by looping gmse\_apply

Instead of using the gmse function, multiple simulations of GMSE can be run by calling gmse\_apply through a loop, reassigning outputs where necessary for the next generation. This is best accomplished using the argument old\_list, which allows previous full results from gmse\_apply to be reinserted into the gmse\_apply function. The argument old\_list is NULL by default, but can instead take the output of a previous full list return of gmse\_apply. This old\_list produced when get\_res = Full includes all data structures and parameter values necessary for a unique simulation of GMSE. An exampe of using get\_res and old\_list in tandem to loop gmse\_apply is shown below.

to\_scare <- FALSE;  
sim\_old <- gmse\_apply(scaring = to\_scare, get\_res = "Full", stakeholders = 6);  
sim\_sum\_1 <- matrix(data = NA, nrow = 20, ncol = 7);  
for(time\_step in 1:20){  
 sim\_new <- gmse\_apply(scaring = to\_scare, get\_res = "Full",   
 old\_list = sim\_old);  
 sim\_sum\_1[time\_step, 1] <- time\_step;  
 sim\_sum\_1[time\_step, 2] <- sim\_new$basic\_output$resource\_results[1];  
 sim\_sum\_1[time\_step, 3] <- sim\_new$basic\_output$observation\_results[1];  
 sim\_sum\_1[time\_step, 4] <- sim\_new$basic\_output$manager\_results[2];  
 sim\_sum\_1[time\_step, 5] <- sim\_new$basic\_output$manager\_results[3];  
 sim\_sum\_1[time\_step, 6] <- sum(sim\_new$basic\_output$user\_results[,2]);   
 sim\_sum\_1[time\_step, 7] <- sum(sim\_new$basic\_output$user\_results[,3]);   
 sim\_old <- sim\_new;  
}  
colnames(sim\_sum\_1) <- c("Time", "Pop\_size", "Pop\_est", "Scare\_cost",   
 "Cull\_cost", "Scare\_count", "Cull\_count");  
print(sim\_sum\_1);

## Time Pop\_size Pop\_est Scare\_cost Cull\_cost Scare\_count Cull\_count  
## [1,] 1 784 680.2721 NA 110 NA 54  
## [2,] 2 828 907.0295 NA 110 NA 54  
## [3,] 3 873 997.7324 NA 110 NA 54  
## [4,] 4 945 975.0567 NA 110 NA 54  
## [5,] 5 1134 1315.1927 NA 10 NA 423  
## [6,] 6 804 680.2721 NA 110 NA 54  
## [7,] 7 890 702.9478 NA 110 NA 54  
## [8,] 8 980 884.3537 NA 110 NA 54  
## [9,] 9 1090 1292.5170 NA 10 NA 426  
## [10,] 10 802 861.6780 NA 110 NA 54  
## [11,] 11 902 1179.1383 NA 10 NA 426  
## [12,] 12 585 680.2721 NA 110 NA 54  
## [13,] 13 616 521.5420 NA 110 NA 54  
## [14,] 14 681 770.9751 NA 110 NA 54  
## [15,] 15 739 793.6508 NA 110 NA 54  
## [16,] 16 823 748.2993 NA 110 NA 54  
## [17,] 17 896 725.6236 NA 110 NA 54  
## [18,] 18 998 929.7052 NA 110 NA 54  
## [19,] 19 1133 1383.2200 NA 10 NA 433  
## [20,] 20 851 793.6508 NA 110 NA 54

Note that one element of the full list gmse\_apply output is the 'basic\_output' itself, which is produced by default when get\_res = "basic". This is what is being used to store the output of sim\_new into sim\_sum\_1. Next, we show how the flexibility of gmse\_apply can be used to dynamically redefine simulation conditions.

# Changing simulation conditions using gmse\_apply

We can take advantage of gmse\_apply to dynamically change parameter values mid-loop. For example, below shows the same code used in the previous example, but with a policy of scaring introduced on time step 10.

to\_scare <- FALSE;  
sim\_old <- gmse\_apply(scaring = to\_scare, get\_res = "Full", stakeholders = 6);  
sim\_sum\_2 <- matrix(data = NA, nrow = 20, ncol = 7);  
for(time\_step in 1:20){  
 sim\_new <- gmse\_apply(scaring = to\_scare, get\_res = "Full",   
 old\_list = sim\_old);  
 sim\_sum\_2[time\_step, 1] <- time\_step;  
 sim\_sum\_2[time\_step, 2] <- sim\_new$basic\_output$resource\_results[1];  
 sim\_sum\_2[time\_step, 3] <- sim\_new$basic\_output$observation\_results[1];  
 sim\_sum\_2[time\_step, 4] <- sim\_new$basic\_output$manager\_results[2];  
 sim\_sum\_2[time\_step, 5] <- sim\_new$basic\_output$manager\_results[3];  
 sim\_sum\_2[time\_step, 6] <- sum(sim\_new$basic\_output$user\_results[,2]);   
 sim\_sum\_2[time\_step, 7] <- sum(sim\_new$basic\_output$user\_results[,3]);   
 sim\_old <- sim\_new;  
 if(time\_step == 10){  
 to\_scare <- TRUE;  
 }  
}  
colnames(sim\_sum\_2) <- c("Time", "Pop\_size", "Pop\_est", "Scare\_cost",   
 "Cull\_cost", "Scare\_count", "Cull\_count");  
print(sim\_sum\_2);

## Time Pop\_size Pop\_est Scare\_cost Cull\_cost Scare\_count Cull\_count  
## [1,] 1 792 861.6780 NA 110 NA 54  
## [2,] 2 831 952.3810 NA 110 NA 54  
## [3,] 3 893 997.7324 NA 110 NA 54  
## [4,] 4 969 929.7052 NA 110 NA 54  
## [5,] 5 1168 1269.8413 NA 10 NA 423  
## [6,] 6 866 748.2993 NA 110 NA 54  
## [7,] 7 933 861.6780 NA 110 NA 54  
## [8,] 8 1048 861.6780 NA 110 NA 54  
## [9,] 9 1198 1383.2200 NA 10 NA 428  
## [10,] 10 903 1020.4082 NA 64 NA 90  
## [11,] 11 981 453.5147 10 110 204 36  
## [12,] 12 1116 1111.1111 59 12 37 294  
## [13,] 13 1008 770.9751 10 110 193 37  
## [14,] 14 1139 907.0295 10 110 171 39  
## [15,] 15 1284 1337.8685 62 10 45 292  
## [16,] 16 1165 1292.5170 65 10 43 303  
## [17,] 17 1005 1247.1655 42 10 66 305  
## [18,] 18 860 1224.4898 68 10 41 307  
## [19,] 19 663 861.6780 10 110 204 36  
## [20,] 20 761 1020.4082 22 64 123 51

Hence, in addition to the previously explained benefits of the flexible gmse\_apply function, one particularly useful feature is that we can use it to study change in policy availability -- in the above case, what happens when scaring is suddenly introduced as a possible policy option. Similar things can be done, for example, to see how manager or user power changes over time. In the example below, users' budgets increase by 100 every time step, with the manager's budget remaining the same. The consequence of this increasing user budget is higher rates of culling and decreased population size.

ub <- 500;  
sim\_old <- gmse\_apply(get\_res = "Full", stakeholders = 6, user\_budget = ub);  
sim\_sum\_3 <- matrix(data = NA, nrow = 20, ncol = 6);  
for(time\_step in 1:20){  
 sim\_new <- gmse\_apply(get\_res = "Full", old\_list = sim\_old,  
 user\_budget = ub);  
 sim\_sum\_3[time\_step, 1] <- time\_step;  
 sim\_sum\_3[time\_step, 2] <- sim\_new$basic\_output$resource\_results[1];  
 sim\_sum\_3[time\_step, 3] <- sim\_new$basic\_output$observation\_results[1];  
 sim\_sum\_3[time\_step, 4] <- sim\_new$basic\_output$manager\_results[3];  
 sim\_sum\_3[time\_step, 5] <- sum(sim\_new$basic\_output$user\_results[,3]);  
 sim\_sum\_3[time\_step, 6] <- ub;  
 sim\_old <- sim\_new;  
 ub <- ub + 100;  
}  
colnames(sim\_sum\_3) <- c("Time", "Pop\_size", "Pop\_est", "Cull\_cost", "Cull\_count",  
 "User\_budget");  
print(sim\_sum\_3);

## Time Pop\_size Pop\_est Cull\_cost Cull\_count User\_budget  
## [1,] 1 1060 907.0295 110 24 500  
## [2,] 2 1218 1292.5170 10 329 600  
## [3,] 3 1013 1269.8413 10 356 700  
## [4,] 4 786 566.8934 110 42 800  
## [5,] 5 982 1292.5170 10 394 900  
## [6,] 6 707 1020.4082 64 90 1000  
## [7,] 7 733 929.7052 110 60 1100  
## [8,] 8 809 816.3265 110 60 1200  
## [9,] 9 890 861.6780 110 66 1300  
## [10,] 10 996 884.3537 110 72 1400  
## [11,] 11 1070 1111.1111 12 507 1500  
## [12,] 12 679 566.8934 110 84 1600  
## [13,] 13 702 680.2721 110 90 1700  
## [14,] 14 710 861.6780 110 96 1800  
## [15,] 15 729 816.3265 110 102 1900  
## [16,] 16 737 770.9751 110 108 2000  
## [17,] 17 782 839.0023 110 114 2100  
## [18,] 18 797 816.3265 110 120 2200  
## [19,] 19 839 589.5692 110 120 2300  
## [20,] 20 868 793.6508 110 126 2400

There is an important note to make about changing arguments to gmse\_apply when old\_list is being used: The function gmse\_apply is trying to avoid a crash, so gmse\_apply will accomodate parameter changes by rebuilding data structures if necessary. For example, if the number of stakeholders is changed (and by including an argument stakeholders to gmse\_apply, it is assumed that stakeholders are changing even they are not), then a new array of agents will need to be built. If landscape dimensions are changed (or just include the argument land\_dim\_1 or land\_dim\_2), then a new landscape willl be built. For most simulation purposes, this will not introduce any undesirable effect on simulation results, but it should be noted and understood when developing models.