

# GMSE: an R package for generalised management strategy evaluation

## Supporting Information 2

A. Bradley Duthie<sup>1,3</sup>, Jeremy J. Cusack<sup>1</sup>, Isabel L. Jones<sup>1</sup>, Jeroen Minderman<sup>1</sup>, Erlend B. Nilsen<sup>2</sup>, Rocío A. Pozo<sup>1</sup>, O. Sarobidy Rakotonarivo<sup>1</sup>, Bram Van Moorter<sup>2</sup>, and Nils Bunnefeld<sup>1</sup>

[1] *Biological and Environmental Sciences, University of Stirling, Stirling, UK* [2] *Norwegian Institute for Nature Research, Trondheim, Norway* [3] *alexander.duthie@stir.ac.uk*

## 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] 1122
##
## $observation_results
## [1] 1179.138
##
## $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      69         NA      NA             NA
## user_2             1      NA      71         NA      NA             NA
## user_3             1      NA      69         NA      NA             NA
## user_4             1      NA      75         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). 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 identities, 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] 1791.383
##
## $manager_results
## [1] 791.3832
##
## $user_results
## [1] 870.5215
```

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     76         NA      NA             NA
## user_2             1      NA     73         NA      NA             NA
## user_3             1      NA     68         NA      NA             NA
## user_4             1      NA     73         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. Note that custom functions sent to `gmse_apply` still need to be specified (`res_mod`, `obs_mod`, `man_mod`, and `use_mod`). An example of using `get_res` and `old_list` in tandem to loop `gmse_apply` is shown below.

```
to_score <- FALSE;
sim_old  <- gmse_apply(scaring = to_score, 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_score, 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	1173	1224.4898	NA	10	NA	438
## [2,]	2	862	1065.7596	NA	19	NA	312
## [3,]	3	619	521.5420	NA	110	NA	54
## [4,]	4	693	725.6236	NA	110	NA	54
## [5,]	5	832	770.9751	NA	110	NA	54
## [6,]	6	939	839.0023	NA	110	NA	54
## [7,]	7	1072	748.2993	NA	110	NA	54
## [8,]	8	1217	1201.8141	NA	10	NA	429
## [9,]	9	949	1179.1383	NA	10	NA	420
## [10,]	10	637	498.8662	NA	110	NA	54
## [11,]	11	714	476.1905	NA	110	NA	54
## [12,]	12	804	816.3265	NA	110	NA	54
## [13,]	13	892	748.2993	NA	110	NA	54
## [14,]	14	1012	1020.4082	NA	64	NA	90
## [15,]	15	1139	1043.0839	NA	30	NA	198
## [16,]	16	1130	816.3265	NA	110	NA	54
## [17,]	17	1282	929.7052	NA	110	NA	54
## [18,]	18	1460	1292.5170	NA	10	NA	417
## [19,]	19	1301	1292.5170	NA	10	NA	425
## [20,]	20	1065	1315.1927	NA	10	NA	424

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	707	997.7324	NA	110	NA	54
##	[2,]	2	753	997.7324	NA	110	NA	54
##	[3,]	3	810	884.3537	NA	110	NA	54
##	[4,]	4	895	1179.1383	NA	10	NA	432
##	[5,]	5	602	816.3265	NA	110	NA	54
##	[6,]	6	655	748.2993	NA	110	NA	54
##	[7,]	7	713	748.2993	NA	110	NA	54
##	[8,]	8	790	861.6780	NA	110	NA	54
##	[9,]	9	877	997.7324	NA	110	NA	54
##	[10,]	10	970	1020.4082	NA	64	NA	90
##	[11,]	11	1069	1337.8685	47	10	59	305
##	[12,]	12	891	544.2177	10	110	193	37
##	[13,]	13	1035	793.6508	10	110	193	37
##	[14,]	14	1179	1587.3016	71	10	40	288
##	[15,]	15	1104	1043.0839	42	30	0	198
##	[16,]	16	1079	1292.5170	72	10	37	294
##	[17,]	17	942	634.9206	10	110	182	38
##	[18,]	18	1085	1292.5170	57	10	48	301
##	[19,]	19	920	1292.5170	64	10	44	283
##	[20,]	20	766	861.6780	10	110	171	39

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	903	952.3810	110	24	500
##	[2,]	2	1003	929.7052	110	30	600
##	[3,]	3	1095	952.3810	110	36	700
##	[4,]	4	1268	1519.2744	10	385	800
##	[5,]	5	1128	1360.5442	10	410	900
##	[6,]	6	845	680.2721	110	54	1000
##	[7,]	7	961	544.2177	110	60	1100
##	[8,]	8	1063	884.3537	110	60	1200
##	[9,]	9	1193	997.7324	110	66	1300
##	[10,]	10	1355	1383.2200	10	522	1400
##	[11,]	11	981	907.0295	110	78	1500
##	[12,]	12	1085	1224.4898	10	570	1600
##	[13,]	13	620	521.5420	110	90	1700
##	[14,]	14	621	521.5420	110	96	1800
##	[15,]	15	627	385.4875	110	102	1900
##	[16,]	16	641	476.1905	110	108	2000
##	[17,]	17	652	249.4331	110	114	2100
##	[18,]	18	650	408.1633	110	120	2200
##	[19,]	19	645	634.9206	110	120	2300
##	[20,]	20	619	612.2449	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 will 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.