

# 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>, 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] 1120
##
## $observation_results
## [1] 1156.463
##
## $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      75         NA      NA             NA
## user_2            1      NA      77         NA      NA             NA
## user_3            1      NA      67         NA      NA             NA
## user_4            1      NA      74         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] 136.0544
##
## $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] 1496.599
##
## $manager_results
## [1] 496.5986
##
## $user_results
## [1] 546.2585
```

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     69         NA      NA             NA
## user_2             1      NA     70         NA      NA             NA
## user_3             1      NA     73         NA      NA             NA
## user_4             1      NA     71         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 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	768	884.3537	NA	110	NA	54
##	[2,]	2	818	929.7052	NA	110	NA	54
##	[3,]	3	875	793.6508	NA	110	NA	54
##	[4,]	4	950	1020.4082	NA	64	NA	90
##	[5,]	5	1104	952.3810	NA	110	NA	54
##	[6,]	6	1259	1201.8141	NA	10	NA	415
##	[7,]	7	1006	725.6236	NA	110	NA	54
##	[8,]	8	1117	975.0567	NA	110	NA	54
##	[9,]	9	1269	884.3537	NA	110	NA	54
##	[10,]	10	1447	1065.7596	NA	20	NA	300
##	[11,]	11	1379	1201.8141	NA	10	NA	439
##	[12,]	12	1127	997.7324	NA	110	NA	54
##	[13,]	13	1257	1020.4082	NA	64	NA	90
##	[14,]	14	1364	997.7324	NA	110	NA	54
##	[15,]	15	1536	1315.1927	NA	10	NA	434
##	[16,]	16	1319	1519.2744	NA	10	NA	432
##	[17,]	17	1045	1179.1383	NA	10	NA	424
##	[18,]	18	735	680.2721	NA	110	NA	54
##	[19,]	19	831	702.9478	NA	110	NA	54
##	[20,]	20	918	952.3810	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	782	952.3810	NA	110	NA	54
##	[2,]	2	813	748.2993	NA	110	NA	54
##	[3,]	3	886	975.0567	NA	110	NA	54
##	[4,]	4	990	1043.0839	NA	30	NA	198
##	[5,]	5	1028	975.0567	NA	110	NA	54
##	[6,]	6	1152	1473.9229	NA	10	NA	431
##	[7,]	7	875	975.0567	NA	110	NA	54
##	[8,]	8	953	997.7324	NA	110	NA	54
##	[9,]	9	1071	1065.7596	NA	20	NA	298
##	[10,]	10	948	884.3537	NA	110	NA	54
##	[11,]	11	1125	1405.8957	57	10	51	290
##	[12,]	12	1018	1133.7868	46	10	61	303
##	[13,]	13	842	725.6236	10	110	180	38
##	[14,]	14	978	1020.4082	29	64	105	45
##	[15,]	15	1128	1156.4626	67	10	40	300
##	[16,]	16	1008	1065.7596	71	20	3	273
##	[17,]	17	864	748.2993	10	110	215	35
##	[18,]	18	994	793.6508	10	110	182	38
##	[19,]	19	1112	1065.7596	64	20	2	282
##	[20,]	20	1000	839.0023	10	110	204	36

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	920	975.0567	110	24	500
## [2,]	2	1043	839.0023	110	30	600
## [3,]	3	1192	952.3810	110	36	700
## [4,]	4	1346	929.7052	110	42	800
## [5,]	5	1714	1224.4898	10	406	900
## [6,]	6	1564	1655.3288	10	421	1000
## [7,]	7	1337	1383.2200	10	445	1100
## [8,]	8	1072	1133.7868	10	470	1200
## [9,]	9	718	589.5692	110	66	1300
## [10,]	10	762	634.9206	110	72	1400
## [11,]	11	841	748.2993	110	78	1500
## [12,]	12	938	702.9478	110	84	1600
## [13,]	13	1023	1156.4626	10	582	1700
## [14,]	14	525	544.2177	110	96	1800
## [15,]	15	505	476.1905	110	102	1900
## [16,]	16	479	385.4875	110	108	2000
## [17,]	17	438	453.5147	110	114	2100
## [18,]	18	394	362.8118	110	120	2200
## [19,]	19	330	340.1361	110	120	2300
## [20,]	20	261	476.1905	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 willll 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.