

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] 1070
##
## $observation_results
## [1] 1405.896
##
## $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     67         NA      NA              NA
## user_2                1      NA     73         NA      NA              NA
## user_3                1      NA     72         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
```

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] 158.7302
##
## $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
##
```

```

112 ## $user_results
113 ## [1] 385

```

114 Note that the `manager_results` and `user_results` are ambiguous here, and can be interpreted as desired –
 115 e.g., as total allowable catch and catches made, or as something like costs of catching set by the manager and
 116 effort to catching made by the user. Hence while manger output is set in terms of costs of performing each
 117 action, and user output is set in terms of action attempts, this need not be the case when using `gmse_apply`
 118 (though it should be recognised when using default GMSE manager and user functions). GMSE default
 119 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)

```

```

120 ## $resource_results
121 ## [1] 1500
122 ##
123 ## $observation_results
124 ## [1] 1428.571
125 ##
126 ## $manager_results
127 ## [1] 428.5714
128 ##
129 ## $user_results
130 ## [1] 471.4286

```

131 It is possible to, for example, specify a simple resource and observation model, but then take advantage of
 132 the genetic algorithm to predict policy decisions and user actions. This can be done by using the default
 133 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)

```

```

134 ## $resource_results
135 ## [1] 1500
136 ##
137 ## $observation_results
138 ## [1] 1350
139 ##
140 ## $manager_results
141 ##      resource_type scaring culling castration feeding help_offspring
142 ## policy_1          1      NA      10          NA      NA          NA
143 ##
144 ## $user_results
145 ##      resource_type scaring culling castration feeding help_offspring
146 ## Manager           1      NA      0          NA      NA          NA
147 ## user_1             1      NA      75          NA      NA          NA
148 ## user_2             1      NA      73          NA      NA          NA
149 ## user_3             1      NA      68          NA      NA          NA
150 ## user_4             1      NA      68          NA      NA          NA
151 ##      tend_crops kill_crops
152 ## Manager        NA      NA
153 ## user_1          NA      NA
154 ## user_2          NA      NA
155 ## user_3          NA      NA
156 ## 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	
165	##	[1,]	1	1134	816.3265	NA	110	NA	54
166	##	[2,]	2	1208	1224.4898	NA	10	NA	425
167	##	[3,]	3	898	1133.7868	NA	10	NA	416
168	##	[4,]	4	554	748.2993	NA	110	NA	54
169	##	[5,]	5	653	929.7052	NA	110	NA	54
170	##	[6,]	6	707	725.6236	NA	110	NA	54
171	##	[7,]	7	782	748.2993	NA	110	NA	54
172	##	[8,]	8	860	1179.1383	NA	10	NA	431
173	##	[9,]	9	522	544.2177	NA	110	NA	54
174	##	[10,]	10	563	476.1905	NA	110	NA	54
175	##	[11,]	11	641	748.2993	NA	110	NA	54
176	##	[12,]	12	739	725.6236	NA	110	NA	54
177	##	[13,]	13	829	929.7052	NA	110	NA	54
178	##	[14,]	14	941	748.2993	NA	110	NA	54
179	##	[15,]	15	1070	929.7052	NA	110	NA	54
180	##	[16,]	16	1260	1383.2200	NA	10	NA	430
181	##	[17,]	17	994	1247.1655	NA	10	NA	431
182	##	[18,]	18	694	770.9751	NA	110	NA	54
183	##	[19,]	19	772	907.0295	NA	110	NA	54
184	##	[20,]	20	853	702.9478	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	
192	##	[1,]	1	1089	1156.4626	NA	10	NA	427
193	##	[2,]	2	728	975.0567	NA	110	NA	54
194	##	[3,]	3	768	748.2993	NA	110	NA	54
195	##	[4,]	4	829	839.0023	NA	110	NA	54
196	##	[5,]	5	996	1088.4354	NA	15	NA	370
197	##	[6,]	6	717	884.3537	NA	110	NA	54
198	##	[7,]	7	786	839.0023	NA	110	NA	54
199	##	[8,]	8	875	1043.0839	NA	30	NA	198
200	##	[9,]	9	809	793.6508	NA	110	NA	54
201	##	[10,]	10	941	1088.4354	NA	15	NA	366
202	##	[11,]	11	683	702.9478	10	110	193	37
203	##	[12,]	12	753	839.0023	10	110	160	40
204	##	[13,]	13	861	861.6780	10	110	210	35
205	##	[14,]	14	1043	1088.4354	49	15	32	287
206	##	[15,]	15	919	1179.1383	56	10	47	299
207	##	[16,]	16	711	997.7324	10	110	204	36
208	##	[17,]	17	827	1133.7868	54	10	55	293
209	##	[18,]	18	636	1224.4898	66	10	43	289
210	##	[19,]	19	401	430.8390	10	110	199	36
211	##	[20,]	20	446	589.5692	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
219	##						
220	##	[1,]	1	1180	1179.1383	10	300
221	##	[2,]	2	990	929.7052	110	30
222	##	[3,]	3	1108	907.0295	110	36
223	##	[4,]	4	1238	1156.4626	10	391
224	##	[5,]	5	1127	1133.7868	10	397
225	##	[6,]	6	884	634.9206	110	54
226	##	[7,]	7	1003	907.0295	110	60
227	##	[8,]	8	1151	1201.8141	10	471
228	##	[9,]	9	813	725.6236	110	66
229	##	[10,]	10	897	839.0023	110	72
230	##	[11,]	11	987	1111.1111	12	511
231	##	[12,]	12	572	453.5147	110	84
232	##	[13,]	13	577	816.3265	110	90
233	##	[14,]	14	576	453.5147	110	96
234	##	[15,]	15	573	589.5692	110	102
235	##	[16,]	16	570	476.1905	110	107
236	##	[17,]	17	556	839.0023	110	114
237	##	[18,]	18	511	453.5147	110	120
238	##	[19,]	19	460	748.2993	110	120
239	##	[20,]	20	410	385.4875	110	126

240 There is an important note to make about changing arguments to `gmse_apply` when `old_list` is being used:
 241 The function `gmse_apply` is trying to avoid a crash, so `gmse_apply` will accomodate parameter changes
 242 by rebuilding data structures if necessary. For example, if the number of stakeholders is changed (and by
 243 including an argument `stakeholders` to `gmse_apply`, it is assumed that stakeholders are changing even they
 244 are not), then a new array of agents will need to be built. If landscape dimensions are changed (or just
 245 include the argument `land_dim_1` or `land_dim_2`), then a new landscape willll be built. For most simulation
 246 purposes, this will not introduce any undesirable effect on simulation results, but it should be noted and
 247 understood when developing models.