

Use of the gmse_apply function

GMSE: an R package for generalised management strategy evaluation (Supporting Information 2)

A. Bradley Duthie^{1,3}, Jeremy J. Cusack¹, Isabel L. Jones¹, Jeroen Minderman¹, Erlend B. Nilsen², Rocío A. Pozo¹, O. Sarobidy Rakotonarivo¹, Bram Van Moorter², and Nils Bunnefeld¹

[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 sub-models ‘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. Here 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 time step of GMSE using predefined sub-models 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 sub-models.

```
print(sim_1);
```

```
## $resource_results
## [1] 1093
##
## $observation_results
## [1] 952.381
##
## $manager_results
##           resource_type scaring culling castration feeding help_offspring
## policy_1             1      NA      64          NA      NA          NA
##
## $user_results
##           resource_type scaring culling castration feeding help_offspring
## Manager             1      NA      0          NA      NA          NA
## user_1               1      NA     15          NA      NA          NA
## user_2               1      NA     15          NA      NA          NA
## user_3               1      NA     15          NA      NA          NA
## user_4               1      NA     15          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 sub-functions could potentially allow for models with multiple resource types. Any custom sub-functions 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 seven time steps. 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      67         NA      NA             NA
##
## $user_results
##           resource_type scaring culling castration feeding help_offspring
## Manager             1      NA      0         NA      NA             NA
## user_1              1      NA     14         NA      NA             NA
## user_2              1      NA     14         NA      NA             NA
## user_3              1      NA     14         NA      NA             NA
## user_4              1      NA     14         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 sub-models

To integrate across different types of sub-models, `gmse_apply` translates between vectors and arrays between each sub-model. For example, because the default GMSE observation model requires a resource array with particular requirements for column identities, when a resource model sub-function 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 a custom 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, sub-models, for example, are defined below; note that each requires a vector from the preceding model.

```
# Alternative observation sub-model
alt_obs <- function(resource_vector){
  X_obs <- resource_vector - 0.1 * resource_vector;
  return(X_obs);
}

# Alternative manager sub-model
alt_man <- function(observation_vector){
  policy <- observation_vector - 1000;
  if(policy < 0){
    policy <- 0;
  }
  return(policy);
}

# Alternative user sub-model
alt_usr <- function(manager_vector){
  harvest <- manager_vector + manager_vector * 0.1;
  return(harvest);
}
```

All of these sub-models 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
```

```

113 ##
114 ## $user_results
115 ## [1] 385

```

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

```

```

122 ## $resource_results
123 ## [1] 1500
124 ##
125 ## $observation_results
126 ## [1] 1201.814
127 ##
128 ## $manager_results
129 ## [1] 201.8141
130 ##
131 ## $user_results
132 ## [1] 221.9955

```

133 It is possible to, e.g., specify a simple resource and observation model, but then take advantage of the genetic
 134 algorithm to predict policy decisions and user actions (see SI5 for a fisheries example). This can be done by
 135 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);

```

```

136 ## $resource_results
137 ## [1] 1500
138 ##
139 ## $observation_results
140 ## [1] 1350
141 ##
142 ## $manager_results
143 ##      resource_type scaring culling castration feeding help_offspring
144 ## policy_1          1      NA      74          NA      NA          NA
145 ##
146 ## $user_results
147 ##      resource_type scaring culling castration feeding help_offspring
148 ## Manager           1      NA      0          NA      NA          NA
149 ## user_1             1      NA     13          NA      NA          NA
150 ## user_2             1      NA     13          NA      NA          NA
151 ## user_3             1      NA     13          NA      NA          NA
152 ## user_4             1      NA     13          NA      NA          NA
153 ##      tend_crops kill_crops
154 ## Manager        NA      NA
155 ## user_1          NA      NA
156 ## user_2          NA      NA
157 ## user_3          NA      NA
158 ## 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	
168	##	[1,]	1	1115	907.0295	NA	105	NA	54
169	##	[2,]	2	1240	1020.4082	NA	10	NA	454
170	##	[3,]	3	904	793.6508	NA	110	NA	54
171	##	[4,]	4	1017	816.3265	NA	105	NA	54
172	##	[5,]	5	1258	1405.8957	NA	10	NA	456
173	##	[6,]	6	965	1269.8413	NA	10	NA	451
174	##	[7,]	7	619	839.0023	NA	110	NA	54
175	##	[8,]	8	659	680.2721	NA	110	NA	54
176	##	[9,]	9	754	748.2993	NA	110	NA	54
177	##	[10,]	10	838	952.3810	NA	108	NA	54
178	##	[11,]	11	995	1020.4082	NA	10	NA	462
179	##	[12,]	12	625	793.6508	NA	110	NA	54
180	##	[13,]	13	677	566.8934	NA	108	NA	54
181	##	[14,]	14	765	657.5964	NA	109	NA	54
182	##	[15,]	15	856	748.2993	NA	106	NA	54
183	##	[16,]	16	969	770.9751	NA	109	NA	54
184	##	[17,]	17	1097	748.2993	NA	110	NA	54
185	##	[18,]	18	1258	1133.7868	NA	10	NA	462
186	##	[19,]	19	959	1111.1111	NA	10	NA	465
187	##	[20,]	20	593	612.2449	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	1113	975.0567	NA	107	NA	54
##	[2,]	2	1154	1337.8685	NA	10	NA	461
##	[3,]	3	815	952.3810	NA	109	NA	54
##	[4,]	4	880	634.9206	NA	102	NA	54
##	[5,]	5	1061	793.6508	NA	105	NA	54
##	[6,]	6	1162	1133.7868	NA	10	NA	462
##	[7,]	7	824	884.3537	NA	109	NA	54
##	[8,]	8	952	634.9206	NA	110	NA	54
##	[9,]	9	1036	1088.4354	NA	10	NA	459
##	[10,]	10	683	521.5420	NA	110	NA	54
##	[11,]	11	763	476.1905	10	105	244	31
##	[12,]	12	838	975.0567	17	101	256	15
##	[13,]	13	972	975.0567	16	95	259	18
##	[14,]	14	1135	1156.4626	64	10	49	280
##	[15,]	15	1035	975.0567	10	108	284	28
##	[16,]	16	1213	1156.4626	64	10	47	285
##	[17,]	17	1117	952.3810	10	110	299	27
##	[18,]	18	1356	1043.0839	62	10	48	289
##	[19,]	19	1279	861.6780	10	110	319	24
##	[20,]	20	1499	1315.1927	59	10	52	283

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
222	##	[1,]	1	1219	1065.7596	10	298
223	##	[2,]	2	1044	1088.4354	10	339
224	##	[3,]	3	806	1337.8685	10	376
225	##	[4,]	4	504	521.5420	110	42
226	##	[5,]	5	591	521.5420	105	48
227	##	[6,]	6	617	770.9751	101	54
228	##	[7,]	7	656	589.5692	107	60
229	##	[8,]	8	698	702.9478	105	66
230	##	[9,]	9	749	657.5964	107	72
231	##	[10,]	10	831	907.0295	109	72
232	##	[11,]	11	936	793.6508	104	84
233	##	[12,]	12	1026	861.6780	105	90
234	##	[13,]	13	1112	1043.0839	10	634
235	##	[14,]	14	583	566.8934	110	96
236	##	[15,]	15	585	521.5420	104	108
237	##	[16,]	16	584	725.6236	109	108
238	##	[17,]	17	562	702.9478	110	113
239	##	[18,]	18	561	589.5692	110	120
240	##	[19,]	19	514	476.1905	104	132
241	##	[20,]	20	453	226.7574	104	138
242	##						

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