Use of the gmse_apply function

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

A. Bradley Duthie¹, Jeremy J. Cusack¹, Isabel L. Jones¹, Jeroen Minderman¹, Erlend B. Nilsen², Rocío A. Pozo¹, O. Sarobidy Rakotonarivo¹, Bram Van Moorter², and Nils Bunnefeld¹


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] 1084
##
## $observation_results
## [1] 793.6508
##
## $manager_results
## resource_type scaring culling castration feeding help_offspring
## policy_1 1 NA 68 NA NA NA

## $user_results
## resource_type scaring culling castration feeding help_offspring
## Manager 1 NA 0 NA NA NA
## user_1 1 NA 14 NA NA NA
## user_2 1 NA 14 NA NA NA
## user_3 1 NA 14 NA NA NA
## user_4 1 NA 14 NA NA NA
##
## tend_crops kill_crops
## Manager NA NA
## user_1 NA NA
## user_2 NA NA
## user_3 NA NA
## user_4 NA NA
Note that in the case above we have the total abundance of resources returned (\texttt{sim\_1$resource\_results}), the estimate of resource abundance from the observation function (\texttt{sim\_1$observation\_results}), the costs the manager sets for the only available action of culling (\texttt{sim\_1$manager\_results}), and the number of culls attempted by each user (\texttt{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 \texttt{res\_mod} instead.

\begin{verbatim}
alt_res <- function(X, K = 2000, rate = 1){
  X_1 <- X + rate*X*(1 - X/K);
  return(X_1);
}
\end{verbatim}

The above function takes in a population size of \texttt{X} and returns a value \texttt{X\_1} based on the population intrinsic growth rate \texttt{rate} and carrying capacity \texttt{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 time steps. The function can be substituted into \texttt{gmse\_apply} to use it instead of the predefined GMSE resource model.

\begin{verbatim}
sim\_2 <- gmse\_apply(res\_mod = alt_res, X = 100, rate = 0.3);
\end{verbatim}

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

\begin{verbatim}
print(sim\_2);
\end{verbatim}

\begin{verbatim}
## $resource\_results
## [1] 128
##
## $observation\_results
## [1] 90.70295
##
## $manager\_results
## resource\_type scaring culling castration feeding help\_offspring
## policy\_1 1 NA 56 NA NA NA
##
## $user\_results
## resource\_type scaring culling castration feeding help\_offspring
## Manager 1 NA 0 NA NA NA
## user\_1 1 NA 17 NA NA NA
## user\_2 1 NA 17 NA NA NA
## user\_3 1 NA 17 NA NA NA
## user\_4 1 NA 17 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
\end{verbatim}
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 identites, 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.

```r
# 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.

```r
gmse_apply(res_mod = alt_res, obs_mod = alt_obs,
            man_mod = alt_man, use_mod = alt_usr, X = 1000);
```

```r
## $resource_results
## [1] 1500

## $observation_results
## [1] 1350

## $manager_results
## [1] 350
```
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 manager 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 sub-models can be added in at any point.

```r
gmse_apply(res_mod = alt_res, obs_mod = observation, 
           man_mod = alt_man, use_mod = alt_usr, X = 1000);
```

It is possible to, e.g., specify a simple resource and observation model, but then take advantage of the genetic algorithm to predict policy decisions and user actions (see SI5 for a fisheries example). This can be done by using the default GMSE manager and user functions (written below explicitly, though this is not necessary).

```r
gmse_apply(res_mod = alt_res, obs_mod = alt_obs, 
           man_mod = manager, use_mod = user, X = 1000);
```
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.

```r
# Set initial conditions
scaring <- FALSE;
initial_pop <- 1000;

# Set up initial state and output
sim_old <- gmse_apply(scaring = scaring, 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 = scaring, get_res = "Full",
                        old_list = sim_old);
  sim_sum_1[,time_step] <- 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;
}

# Show results
print(sim_sum_1);
```

```r
# Time Pop_size Pop_est Scare_cost Cull_cost Scare_count Cull_count
# [1,] 1 1171 1133.7868 NA 10 NA 600
# [2,] 2 637 385.4875 NA 110 NA 54
# [3,] 3 680 589.5692 NA 110 NA 54
# [4,] 4 718 498.8662 NA 110 NA 54
# [5,] 5 857 975.0567 NA 110 NA 54
# [6,] 6 950 770.9751 NA 110 NA 54
# [7,] 7 1068 952.3810 NA 110 NA 54
# [8,] 8 1225 793.6508 NA 110 NA 54
# [9,] 9 1374 1292.5170 NA 10 NA 600
# [10,] 10 951 997.7324 NA 110 NA 54
# [11,] 11 1066 1315.1927 NA 10 NA 600
# [12,] 12 564 702.9478 NA 110 NA 54
# [13,] 13 612 566.8934 NA 110 NA 54
# [14,] 14 640 702.9478 NA 110 NA 54
# [15,] 15 718 657.5964 NA 110 NA 54
# [16,] 16 787 861.6780 NA 110 NA 54
# [17,] 17 885 929.7052 NA 110 NA 54
# [18,] 18 998 816.3265 NA 110 NA 54
# [19,] 19 1130 861.6780 NA 110 NA 54
# [20,] 20 1308 1247.1655 NA 10 NA 600
```

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.

```r
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);
```

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Pop_size</th>
<th>Pop_est</th>
<th>Scare_cost</th>
<th>Cull_cost</th>
<th>Scare_count</th>
<th>Cull_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1150</td>
<td>1224.498</td>
<td>NA</td>
<td>10</td>
<td>NA</td>
<td>600</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>628</td>
<td>680.2721</td>
<td>NA</td>
<td>110</td>
<td>NA</td>
<td>54</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>663</td>
<td>385.4875</td>
<td>NA</td>
<td>110</td>
<td>NA</td>
<td>54</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>729</td>
<td>589.5692</td>
<td>NA</td>
<td>110</td>
<td>NA</td>
<td>54</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>874</td>
<td>907.0295</td>
<td>NA</td>
<td>110</td>
<td>NA</td>
<td>54</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>984</td>
<td>1292.5170</td>
<td>NA</td>
<td>10</td>
<td>NA</td>
<td>600</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>447</td>
<td>430.8390</td>
<td>NA</td>
<td>110</td>
<td>NA</td>
<td>54</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>469</td>
<td>521.5420</td>
<td>NA</td>
<td>110</td>
<td>NA</td>
<td>54</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>513</td>
<td>770.9751</td>
<td>NA</td>
<td>110</td>
<td>NA</td>
<td>54</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>531</td>
<td>521.5420</td>
<td>NA</td>
<td>110</td>
<td>NA</td>
<td>54</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>573</td>
<td>385.4875</td>
<td>10</td>
<td>110</td>
<td>600</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>694</td>
<td>498.8662</td>
<td>10</td>
<td>110</td>
<td>600</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>824</td>
<td>861.6780</td>
<td>10</td>
<td>110</td>
<td>600</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>988</td>
<td>861.6780</td>
<td>10</td>
<td>110</td>
<td>600</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>1213</td>
<td>1201.8141</td>
<td>68</td>
<td>10</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>742</td>
<td>589.5692</td>
<td>10</td>
<td>110</td>
<td>600</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>901</td>
<td>952.3810</td>
<td>10</td>
<td>110</td>
<td>600</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>1079</td>
<td>1383.2200</td>
<td>55</td>
<td>10</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>568</td>
<td>498.8662</td>
<td>10</td>
<td>110</td>
<td>600</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>689</td>
<td>430.8390</td>
<td>10</td>
<td>110</td>
<td>600</td>
<td>0</td>
</tr>
</tbody>
</table>

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;
}
print(sim_sum_3);

## Time Pop_size Pop_est Cull_cost Cull_count User_budget
## [1,] 1 1209 1541.9501 10 300 500
## [2,] 2 1075 1224.4898 10 360 600
## [3,] 3 825 1088.4354 10 420 700
## [4,] 4 482 521.5420 110 42 800
## [5,] 5 571 702.9478 110 48 900
## [6,] 6 609 612.2449 110 54 1000
## [7,] 7 641 544.2177 110 60 1100
## [8,] 8 664 793.6508 110 60 1200
## [9,] 9 737 612.2449 110 66 1300
## [10,] 10 767 725.6236 110 72 1400
## [11,] 11 816 839.0023 110 78 1500
## [12,] 12 845 498.8662 110 84 1600
## [13,] 13 911 748.2993 110 90 1700
## [14,] 14 986 861.6780 110 96 1800
## [15,] 15 1066 1020.4082 52 216 1900
## [16,] 16 1040 1043.0839 26 456 2000
## [17,] 17 708 657.5964 110 114 2100
## [18,] 18 710 839.0023 110 120 2200
## [19,] 19 725 725.6236 110 120 2300
## [20,] 20 717 793.6508 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 such as 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.
Special considerations for looping with custom sub-models

There are some special considerations that need to be made when using custom sub-models and the `old_list` argument within a loop as above. These considerations boil down to two key points.

1. Custom sub-models *always* need to read in explicitly as an argument in `gmse_apply` (i.e., they will not be remembered by `old_list`).
2. Custom sub-model arguments also *always* need to be updated *outside* of `gmse_apply` before output is used as an argument in `old_list` (i.e., `gmse_apply` cannot know what custom function argument needs to be updated, so this needs to be done manually).

An example below illustrates the above points more clearly. Assume that the custom resource sub-model defined above needs to be integrated with the default observation, manager, and user sub-models using `gmse_apply`.

```r
alt_res <- function(X, K = 2000, rate = 1){
  X_1 <- X + rate*X*(1 - X/K);
  return(X_1);
}
```

The sub-model can be integrated once using `gmse_apply` as demonstrated above, but in the full `gmse_apply` output, the argument `X` will not change from its initial value (because sub-model functions can take any number of arbitrary arguments, `gmse_apply` has no way of knowing that `X` is meant to be the resource number and not some other parameter).

```r
sim_4 <- gmse_apply(res_mod = alt_res, X = 1000, get_res = "Full");
print(sim_4$basic_output);
## $resource_results
## [1] 1500
## $observation_results
## [1] 1519.274
## $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
```

Note that in the above output, the resource abundance has increased and is now `sim_4$basic_output$resource_results`. But if we look at `sim_4$X`, the value is still 1000.

```r
print(sim_4$X);
```
To loop through multiple time steps with the custom function `alt_res`, it is therefore necessary to update `sim4$X` with the updated value from either `sim4$resource_vector` or `sim4$basic_output$resource_results` (the two values should be identical). The loop below shows a simple example.

```r
init_abun <- 1000;
sim_old <- gmse_apply(get_res = "Full", res_mod = alt_res, X = init_abun);
for(time_step in 1:20){
  sim_new <- gmse_apply(res_mod = alt_res, get_res = "Full",
                        old_list = sim_old);
  sim_old <- sim_new;
  sim_old$X <- sim_new$resource_vector;
}
```

Note again that the custom sub-model is read into to `gmse_apply` as an argument within the loop (`res_mod = alt_res`), and the output of `sim_new` is used to update the custom argument `X` in `alt_res` (`sim_old$X <- sim_new$resource_vector`). The population quickly increases to near carrying capacity, which can be summarised by using the same table structure explained above.

```r
init_abun <- 1000;
sim_old <- gmse_apply(get_res = "Full", res_mod = alt_res, X = init_abun);
sim_sum_4 <- matrix(data = NA, nrow = 5, ncol = 5);
for(time_step in 1:5){
  sim_new <- gmse_apply(res_mod = alt_res, get_res = "Full",
                        old_list = sim_old);
  sim_sum_4[time_step, 1] <- time_step;
  sim_sum_4[time_step, 2] <- sim_new$basic_output$resource_results[1];
  sim_sum_4[time_step, 3] <- sim_new$basic_output$observation_results[1];
  sim_sum_4[time_step, 4] <- sim_new$basic_output$manager_results[3];
  sim_sum_4[time_step, 5] <- sum(sim_new$basic_output$user_results[,3]);
  sim_old <- sim_new;
  sim_old$X <- sim_new$resource_vector;
}
colnames(sim_sum_4) <- c("Time", "Pop_size", "Pop_est", "Cull_cost",
                          "Cull_count");
print(sim_sum_4);
```

```r
## Time Pop_size Pop_est Cull_cost Cull_count
## [1,] 1 1500 1473.923 10 400
## [2,] 2 1875 1836.735 10 400
## [3,] 3 1992 1882.086 10 400
## [4,] 4 1999 2222.222 10 400
## [5,] 5 1999 2086.168 10 400
```

This is the recommended way to loop custom functions in `gmse_apply`. Note that elements of `old_list` will over-ride custom arguments to `gmse_apply` so specifying custom arguments that are already present in `old_list` will not work.