Use of the gmse_apply function

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

A. Bradley Duthie\textsuperscript{1,3}, Jeremy J. Cusack\textsuperscript{1}, Isabel L. Jones\textsuperscript{1}, Jeroen Minderman\textsuperscript{1}, Erlend B. Nilsen\textsuperscript{2}, Rocío A. Pozo\textsuperscript{1}, O. Sarobidy Rakotonarivo\textsuperscript{1}, Bram Van Moorter\textsuperscript{2}, and Nils Bunnefeld\textsuperscript{1}

\textsuperscript{1} Biological and Environmental Sciences, University of Stirling, Stirling, UK \textsuperscript{2} Norwegian Institute for Nature Research, Trondheim, Norway \textsuperscript{3} alexander.duthie@stir.ac.uk

Extended introduction to the GMSE apply function (\texttt{gmse_apply})

The \texttt{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 \texttt{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 \texttt{gmse_apply}, and how it might be used to simulate myriad management scenarios \textit{in silico}.

A simple run of \texttt{gmse_apply()} returns one time step of GMSE using predefined sub-models and default parameter values.

\begin{verbatim}
 sim_1 <- gmse_apply();
\end{verbatim}

For \texttt{sim_1}, the default ‘basic’ results are returned as below, which summarise key values for all sub-models.

\begin{verbatim}
print(sim_1);
\end{verbatim}

\begin{verbatim}
## $resource_results
## [1] 1113
##
## $observation_results
## [1] 1111.111
##
## $manager_results
## resource_type scaring culling castration feeding help_offspring
## policy_1 1 NA 70 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
\end{verbatim}
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 \(X\) 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 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., \(K = 2000\)) or values specified directly into \texttt{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 \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}
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.

```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
```
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 Fisheries example integrating FLR 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);
```

```r
# $resource_results
# [1] 1500
#
# $observation_results
# [1] 1473.923
#
# $manager_results
# [1] 473.9229
#
# $user_results
# [1] 521.3152
```
Running GMSE simulations by looping \texttt{gmse_apply}

Instead of using the \texttt{gmse} function, multiple simulations of GMSE can be run by calling \texttt{gmse_apply} through a loop, reassigning outputs where necessary for the next generation. This is best accomplished using the argument \texttt{old_list}, which allows previous full results from \texttt{gmse_apply} to be reinserted into the \texttt{gmse_apply} function. The argument \texttt{old_list} is NULL by default, but can instead take the output of a previous full list return of \texttt{gmse_apply}. This \texttt{old_list} produced when \texttt{get_res = Full} includes all data structures and parameter values necessary for a unique simulation of GMSE. Note that custom functions sent to \texttt{gmse_apply} still need to be specified (\texttt{res_mod}, \texttt{obs_mod}, \texttt{man_mod}, and \texttt{use_mod}). An example of using \texttt{get_res} and \texttt{old_list} in tandem to loop \texttt{gmse_apply} is shown below.

```r
to_scare <- FALSE;
sim_old <- \texttt{gmse_apply}(scaring = to_scare, get_res = "Full", stakeholders = 6);
sim_sum_1 <- \texttt{matrix}(data = NA, nrow = 20, ncol = 7);
for(time_step in 1:20){
  sim_new <- \texttt{gmse_apply}(scaring = to_scare, 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);
```

```r
## Time Pop_size Pop_est Scare_cost Cull_cost Scare_count Cull_count
## [1,] 1 1091 952.3810 NA 110 NA 54
## [2,] 2 1184 1247.1655 NA 29 NA 204
## [3,] 3 1175 1020.4082 NA 110 NA 54
## [4,] 4 1313 1405.8957 NA 17 NA 348
## [5,] 5 1252 1156.4626 NA 44 NA 132
## [6,] 6 1355 1247.1655 NA 28 NA 210
## [7,] 7 1338 1133.7868 NA 52 NA 114
## [8,] 8 1458 1405.8957 NA 17 NA 348
## [9,] 9 1304 1043.0839 NA 110 NA 54
## [10,] 10 1491 1337.8685 NA 20 NA 300
## [11,] 11 1425 1383.2200 NA 18 NA 330
## [12,] 12 1340 997.7324 NA 109 NA 54
## [13,] 13 1537 1564.6259 NA 12 NA 498
## [14,] 14 1241 1428.5714 NA 16 NA 372
## [15,] 15 1049 952.3810 NA 110 NA 54
## [16,] 16 1197 1088.4354 NA 80 NA 72
## [17,] 17 1379 1519.2744 NA 13 NA 456
```
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);
```

```r
## Time Pop_size Pop_est Scare_cost Cull_cost Scare_count Cull_count
## [1,] 1 1088 1043.0839 NA 110 NA 54
## [2,] 2 1160 1111.1111 NA 64 NA 90
## [3,] 3 1214 1360.5442 NA 19 NA 312
## [4,] 4 1061 1179.1383 NA 39 NA 150
## [5,] 5 1187 1201.8141 NA 34 NA 174
## [6,] 6 1167 1473.9229 NA 15 NA 396
## [7,] 7 897 997.7324 NA 110 NA 54
## [8,] 8 1026 997.7324 NA 110 NA 54
## [9,] 9 1180 1519.2744 NA 14 NA 426
## [10,] 10 899 1043.0839 NA 110 NA 54
## [11,] 11 1027 975.0567 10 110 0 54
## [12,] 12 1165 1088.4354 19 79 2 72
## [13,] 13 1318 1405.8957 51 17 0 348
## [14,] 14 1170 1179.1383 49 39 0 150
## [15,] 15 1231 1179.1383 65 38 0 156
## [16,] 16 1324 1473.9229 88 15 0 396
## [17,] 17 1097 861.6780 10 110 1 54
## [18,] 18 1251 1337.8685 68 21 0 282
```
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.

```r
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);
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Pop_size</th>
<th>Pop_est</th>
<th>Cull_cost</th>
<th>Cull_count</th>
<th>User_budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1172</td>
<td>1269.8413</td>
<td>13</td>
<td>228</td>
<td>500</td>
</tr>
<tr>
<td>2</td>
<td>1076</td>
<td>1201.8141</td>
<td>17</td>
<td>210</td>
<td>600</td>
</tr>
<tr>
<td>3</td>
<td>1011</td>
<td>929.7052</td>
<td>110</td>
<td>36</td>
<td>700</td>
</tr>
<tr>
<td>4</td>
<td>1145</td>
<td>952.3810</td>
<td>110</td>
<td>42</td>
<td>800</td>
</tr>
<tr>
<td>5</td>
<td>1430</td>
<td>1451.2472</td>
<td>12</td>
<td>450</td>
<td>900</td>
</tr>
<tr>
<td>6</td>
<td>1167</td>
<td>929.7052</td>
<td>110</td>
<td>54</td>
<td>1000</td>
</tr>
<tr>
<td>7</td>
<td>1324</td>
<td>1247.1655</td>
<td>29</td>
<td>222</td>
<td>1100</td>
</tr>
<tr>
<td>8</td>
<td>1327</td>
<td>1405.8957</td>
<td>19</td>
<td>378</td>
<td>1200</td>
</tr>
<tr>
<td>9</td>
<td>1113</td>
<td>1020.4082</td>
<td>110</td>
<td>66</td>
<td>1300</td>
</tr>
<tr>
<td>10</td>
<td>1242</td>
<td>1111.1111</td>
<td>78</td>
<td>102</td>
<td>1400</td>
</tr>
<tr>
<td>11</td>
<td>1364</td>
<td>1088.4354</td>
<td>107</td>
<td>84</td>
<td>1500</td>
</tr>
<tr>
<td>12</td>
<td>1556</td>
<td>1859.4104</td>
<td>12</td>
<td>798</td>
<td>1600</td>
</tr>
<tr>
<td>13</td>
<td>897</td>
<td>997.7324</td>
<td>110</td>
<td>90</td>
<td>1700</td>
</tr>
<tr>
<td>14</td>
<td>958</td>
<td>793.6508</td>
<td>110</td>
<td>96</td>
<td>1800</td>
</tr>
<tr>
<td>15</td>
<td>1034</td>
<td>997.7324</td>
<td>110</td>
<td>102</td>
<td>1900</td>
</tr>
<tr>
<td>16</td>
<td>1146</td>
<td>1496.5986</td>
<td>27</td>
<td>444</td>
<td>2000</td>
</tr>
<tr>
<td>17</td>
<td>843</td>
<td>816.3265</td>
<td>110</td>
<td>114</td>
<td>2100</td>
</tr>
<tr>
<td>18</td>
<td>869</td>
<td>929.7052</td>
<td>110</td>
<td>120</td>
<td>2200</td>
</tr>
<tr>
<td>19</td>
<td>896</td>
<td>884.3537</td>
<td>110</td>
<td>120</td>
<td>2300</td>
</tr>
<tr>
<td>20</td>
<td>960</td>
<td>770.9751</td>
<td>110</td>
<td>126</td>
<td>2400</td>
</tr>
</tbody>
</table>

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 accommodate 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 be 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);
```

```r
## $resource_results
## [1] 1500
##
## $observation_results
## [1] 1496.599
##
## $manager_results
## resource_type scaring culling castration feeding help_offspring
## policy_1 1 NA 64 NA NA NA
##
## $user_results
## resource_type scaring culling castration feeding help_offspring
## Manager 1 NA 0 NA NA NA
## user_1 1 NA 15 NA NA NA
## user_2 1 NA 15 NA NA NA
## user_3 1 NA 15 NA NA NA
## user_4 1 NA 15 NA NA NA
## tend_crops kill_crops
## Manager NA NA
## user_1 NA NA
## user_2 NA NA
## user_3 NA NA
## user_4 NA NA
```
Note that in the above output, the resource abundance has increased and is now sim_4$basic_output$resource_results. But if we look at sim_4$X, the value is still 1000.

\texttt{print(sim_4$X);}

\#
[1]
1000

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

\begin{verbatim}
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;
}
\end{verbatim}

Note again that the custom sub-model is read into \texttt{gmse_apply} as an argument within the loop (\texttt{res_mod = alt_res}), and the output of \texttt{sim_new} is used to update the custom argument \texttt{X} in \texttt{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.

\begin{verbatim}
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);
##
# Time  Pop_size  Pop_est  Cull_cost  Cull_count
## [1,] 1 1500 1655.329 10 400
## [2,] 2 1875 1678.005 10 400
## [3,] 3 1992 1768.707 10 400
## [4,] 4 1999 2154.195 10 400
## [5,] 5 1999 1882.086 10 400
\end{verbatim}

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