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. Cusack[1], Isabel L. Jones[1], Jeroen Minderman[1], Erlend B. Nilsen[2], Rocío A. Pozo[1], O. Sarobidy Rakotonarivo[1], Bram Van Moorter[2], and Nils Bunnefeld[1]

[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] 1081
##
## $observation_results
## [1] 1020.408
##
## $manager_results
##          resource_type scaring culling castration feeding help_offspring
## policy_1             1      NA      41         NA      NA             NA
##
## $user_results
##         resource_type scaring culling castration feeding help_offspring
## Manager             1      NA       0         NA      NA             NA
## user_1              1      NA      24         NA      NA             NA
## user_2              1      NA      24         NA      NA             NA
## user_3              1      NA      24         NA      NA             NA
## user_4              1      NA      24         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 ca 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] 181.4059
##
## $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
```

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);
```

```
## $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 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);
```

```
## $resource_results
## [1] 1500
##
## $observation_results
## [1] 1564.626
##
## $manager_results
## [1] 564.6259
##
## $user_results
## [1] 621.0884
```

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);
```

```
## $resource_results
## [1] 1500
##
## $observation_results
## [1] 1350
##
## $manager_results
##          resource_type scaring culling castration feeding help_offspring
## policy_1             1      NA      58         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
```

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_scare  <- FALSE;
sim_old   <- gmse_apply(scaring = to_scare, 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_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);
```

```
##       Time Pop_size   Pop_est Scare_cost Cull_cost Scare_count Cull_count
##  [1,]    1     1139 1179.1383         NA        10          NA        600
##  [2,]    2      623  748.2993         NA       110          NA         54
##  [3,]    3      636  952.3810         NA       110          NA         54
##  [4,]    4      700  634.9206         NA       110          NA         54
##  [5,]    5      813  997.7324         NA       110          NA         54
##  [6,]    6      882 1133.7868         NA        10          NA        600
##  [7,]    7      344  362.8118         NA       110          NA         54
##  [8,]    8      356  385.4875         NA       110          NA         54
##  [9,]    9      355  249.4331         NA       110          NA         54
## [10,]   10      356  317.4603         NA       110          NA         54
## [11,]   11      379  362.8118         NA       110          NA         54
## [12,]   12      374  476.1905         NA       110          NA         54
## [13,]   13      378  453.5147         NA       110          NA         54
## [14,]   14      384  453.5147         NA       110          NA         54
## [15,]   15      396  249.4331         NA       110          NA         54
## [16,]   16      391  340.1361         NA       110          NA         54
## [17,]   17      383  362.8118         NA       110          NA         54
## [18,]   18      399  272.1088         NA       110          NA         54
## [19,]   19      411  362.8118         NA       110          NA         54
## [20,]   20      421  430.8390         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     1147 1201.8141         NA        10          NA        600
##  [2,]    2      629  702.9478         NA       110          NA         54
##  [3,]    3      656  725.6236         NA       110          NA         54
##  [4,]    4      710  861.6780         NA       110          NA         54
##  [5,]    5      859 1315.1927         NA        10          NA        600
##  [6,]    6      282  362.8118         NA       110          NA         54
##  [7,]    7      277  340.1361         NA       110          NA         54
##  [8,]    8      254  317.4603         NA       110          NA         54
##  [9,]    9      232  158.7302         NA       110          NA         54
## [10,]   10      217  158.7302         NA       110          NA         54
## [11,]   11      199  136.0544         10       110         600          0
## [12,]   12      235  385.4875         10       110         600          0
## [13,]   13      283  317.4603         10       110         600          0
## [14,]   14      336  476.1905         10       110         600          0
## [15,]   15      401  340.1361         10       110         600          0
## [16,]   16      475  521.5420         10       110         600          0
## [17,]   17      576  476.1905         10       110         600          0
## [18,]   18      697  634.9206         10       110         600          0
## [19,]   19      834  952.3810         10       110         600          0
## [20,]   20     1013 1269.8413         58        10           0        600
```

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     1137  975.0567       110         24         500
##  [2,]    2     1284 1111.1111        10        360         600
##  [3,]    3     1034  793.6508       110         36         700
##  [4,]    4     1210 1201.8141        10        480         800
##  [5,]    5      949  770.9751       110         48         900
##  [6,]    6     1053  839.0023       110         54        1000
##  [7,]    7     1186  997.7324       110         60        1100
##  [8,]    8     1345 1224.4898        10        720        1200
##  [9,]    9      744  793.6508       110         66        1300
## [10,]   10      803  634.9206       110         72        1400
## [11,]   11      879  952.3810       110         78        1500
## [12,]   12      948 1043.0839        20        480        1600
## [13,]   13      547  521.5420       110         90        1700
## [14,]   14      549  498.8662       110         96        1800
## [15,]   15      531  476.1905       110        102        1900
## [16,]   16      505  702.9478       110        108        2000
## [17,]   17      483  521.5420       110        114        2100
## [18,]   18      444  476.1905       110        120        2200
## [19,]   19      392  453.5147       110        120        2300
## [20,]   20      326  294.7846       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 willl 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.