GMSE: an R package for generalised management strategy evaluation

Supporting Information 2

A. Bradley Duthie[13], 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 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] 1110
##
## $observation_results
## [1] 1111.111
##
## $manager_results
##          resource_type scaring culling castration feeding help_offspring
## policy_1             1      NA     444         NA      NA             NA
##
## $user_results
##         resource_type scaring culling castration feeding help_offspring
## Manager             1      NA       0         NA      NA             NA
## user_1              1      NA      22         NA      NA             NA
## user_2              1      NA      22         NA      NA             NA
## user_3              1      NA      22         NA      NA             NA
## user_4              1      NA      22         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] 136.0544
##
## $manager_results
##          resource_type scaring culling castration feeding help_offspring
## policy_1             1      NA     454         NA      NA             NA
##
## $user_results
##         resource_type scaring culling castration feeding help_offspring
## Manager             1      NA       0         NA      NA             NA
## user_1              1      NA      21         NA      NA             NA
## user_2              1      NA      21         NA      NA             NA
## user_3              1      NA      21         NA      NA             NA
## user_4              1      NA      21         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 identites, 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
##
```

```
## $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 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)
```

```
## $resource_results
## [1] 1500
##
## $observation_results
## [1] 1383.22
##
## $manager_results
## [1] 383.22
##
## $user_results
## [1] 421.542
```

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

```
## $resource_results
## [1] 1500
##
## $observation_results
## [1] 1350
##
## $manager_results
##          resource_type scaring culling castration feeding help_offspring
## policy_1             1      NA     462         NA      NA             NA
##
## $user_results
##         resource_type scaring culling castration feeding help_offspring
## Manager             1      NA       0         NA      NA             NA
## user_1              1      NA      21         NA      NA             NA
## user_2              1      NA      21         NA      NA             NA
## user_3              1      NA      21         NA      NA             NA
## user_4              1      NA      21         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. An exampe 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     1069  839.0023         NA       451          NA        126
##  [2,]    2     1083  929.7052         NA       453          NA        126
##  [3,]    3     1138 1043.0839         NA       452          NA        126
##  [4,]    4     1212 1269.8413         NA       469          NA        120
##  [5,]    5     1413 1564.6259         NA       454          NA        126
##  [6,]    6     1538 1587.3016         NA       466          NA        123
##  [7,]    7     1719 1678.0045         NA       447          NA        126
##  [8,]    8     1884 1655.3288         NA       449          NA        126
##  [9,]    9     2103 1473.9229         NA       454          NA        126
## [10,]   10     2295 2199.5465         NA       454          NA        126
## [11,]   11     2367 2290.2494         NA       461          NA        126
## [12,]   12     2345 2063.4921         NA       453          NA        126
## [13,]   13     2313 2358.2766         NA       442          NA        131
## [14,]   14     2431 2267.5737         NA       478          NA        120
## [15,]   15     2440 2471.6553         NA       469          NA        121
## [16,]   16     2396 2131.5193         NA       445          NA        130
## [17,]   17     2412 2448.9796         NA       451          NA        126
## [18,]   18     2415 2267.5737         NA       455          NA        126
## [19,]   19     2446 1995.4649         NA       443          NA        132
## [20,]   20     2425 2380.9524         NA       446          NA        129
```

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     1076  907.0295         NA       469          NA        121
##  [2,]    2     1090  793.6508         NA       450          NA        126
##  [3,]    3     1094  975.0567         NA       453          NA        126
##  [4,]    4     1147 1224.4898         NA       441          NA        132
##  [5,]    5     1323  929.7052         NA       457          NA        126
##  [6,]    6     1407 1383.2200         NA       448          NA        126
##  [7,]    7     1514 1609.9773         NA       460          NA        126
##  [8,]    8     1662 1791.3832         NA       443          NA        131
##  [9,]    9     1892 1768.7075         NA       450          NA        126
## [10,]   10     2090 2018.1406         NA       449          NA        126
## [11,]   11     2294 2018.1406        330       359         111         63
## [12,]   12     2397 2290.2494        342       337          77         97
## [13,]   13     2424 2698.4127        342       344          87         85
## [14,]   14     2495 2562.3583        348       336          77         96
## [15,]   15     2476 2290.2494        337       341          94         80
## [16,]   16     2422 2040.8163        346       355          96         72
## [17,]   17     2470 2199.5465        344       338          74        100
## [18,]   18     2518 2630.3855        346       327          78         96
## [19,]   19     2500 2607.7098        343       333          75         99
## [20,]   20     2481 2176.8707        348       338          74         95
```

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     1226 1133.787       455          6         500
##  [2,]    2     1373 1315.193       451          6         600
##  [3,]    3     1580 1632.653       462          6         700
##  [4,]    4     1849 1496.599       455          6         800
##  [5,]    5     2348 2131.519       461          6         900
##  [6,]    6     2473 2108.844       459         12        1000
##  [7,]    7     2500 2335.601       449         12        1100
##  [8,]    8     2575 2380.952       439         12        1200
##  [9,]    9     2634 2970.522       457         12        1300
## [10,]   10     2664 2947.846       464         18        1400
## [11,]   11     2670 2312.925       461         18        1500
## [12,]   12     2619 2131.519       451         18        1600
## [13,]   13     2654 2517.007       472         18        1700
## [14,]   14     2640 2267.574       476         18        1800
## [15,]   15     2619 2494.331       450         24        1900
## [16,]   16     2595 2222.222       457         24        2000
## [17,]   17     2520 2698.413       443         24        2100
## [18,]   18     2626 2335.601       436         28        2200
## [19,]   19     2592 3061.224       441         30        2300
## [20,]   20     2626 2879.819       448         30        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 `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.