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] 1149
##
## $observation_results
## [1] 1224.49
##
## $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 (`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] 136.0544
##
## $manager_results
##          resource_type scaring culling castration feeding help_offspring
## policy_1             1      NA      48         NA      NA             NA
##
## $user_results
##         resource_type scaring culling castration feeding help_offspring
## Manager             1      NA       0         NA      NA             NA
## user_1              1      NA      20         NA      NA             NA
## user_2              1      NA      20         NA      NA             NA
## user_3              1      NA      20         NA      NA             NA
## user_4              1      NA      20         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.

```
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] 1632.653
##
## $manager_results
## [1] 632.6531
##
## $user_results
## [1] 695.9184
```

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).

```
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      63         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
```

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     1083 1043.084         NA       110          NA         54
##  [2,]    2     1161 1065.760         NA       108          NA         54
##  [3,]    3     1277 1541.950         NA        13          NA        456
##  [4,]    4      973 1156.463         NA        45          NA        132
##  [5,]    5     1089 1043.084         NA       110          NA         54
##  [6,]    6     1213 1269.841         NA        27          NA        222
##  [7,]    7     1182 1065.760         NA       106          NA         54
##  [8,]    8     1373 1337.868         NA        20          NA        300
##  [9,]    9     1284 1179.138         NA        39          NA        150
## [10,]   10     1380 1360.544         NA        19          NA        312
## [11,]   11     1302 1156.463         NA        45          NA        132
## [12,]   12     1404 1201.814         NA        35          NA        168
## [13,]   13     1481 1292.517         NA        24          NA        246
## [14,]   14     1466 1292.517         NA        24          NA        246
## [15,]   15     1483 1451.247         NA        15          NA        396
## [16,]   16     1299 1201.814         NA        35          NA        168
## [17,]   17     1347 1315.193         NA        22          NA        270
```

```
## [18,]   18      1297 1587.302         NA         12            NA         498
## [19,]   19       975 1247.166         NA         28            NA         210
## [20,]   20       904 1156.463         NA         44            NA         132
```

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     1141 1179.1383         NA        37          NA        162
##  [2,]    2     1112 1201.8141         NA        35          NA        168
##  [3,]    3     1125  975.0567         NA       110          NA         54
##  [4,]    4     1249 1269.8413         NA        26          NA        228
##  [5,]    5     1349 1247.1655         NA        28          NA        210
##  [6,]    6     1370 1383.2200         NA        18          NA        330
##  [7,]    7     1263 1201.8141         NA        35          NA        168
##  [8,]    8     1309 1292.5170         NA        24          NA        246
##  [9,]    9     1303 1179.1383         NA        39          NA        150
## [10,]   10     1426 1428.5714         NA        16          NA        372
## [11,]   11     1278 1088.4354         13        79           6         72
## [12,]   12     1487 1632.6531         53        10           0        600
## [13,]   13     1060  884.3537         10       110           1         54
## [14,]   14     1206  793.6508         10       110           0         54
## [15,]   15     1388 1224.4898         47        31           0        192
## [16,]   16     1453 1088.4354         31        79           0         72
## [17,]   17     1672 1156.4626         37        43           0        138
## [18,]   18     1815 1405.8957         98        17           0        348
```

```
## [19,]   19      1812 1655.3288         104         11            0         540
## [20,]   20      1566 1360.5442          94         19            0         312
```

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     1188  975.0567       110         24         500
##  [2,]    2     1373 1768.7075        10        360         600
##  [3,]    3     1176  839.0023       110         36         700
##  [4,]    4     1344 1224.4898        21        228         800
##  [5,]    5     1487 1269.8413        21        252         900
##  [6,]    6     1463 1360.5442        17        348        1000
##  [7,]    7     1296 1315.1927        22        300        1100
##  [8,]    8     1196 1269.8413        29        246        1200
##  [9,]    9     1136 1564.6259        15        516        1300
## [10,]   10      737  770.9751       110         72        1400
## [11,]   11      774  634.9206       110         78        1500
## [12,]   12      824  770.9751       110         84        1600
## [13,]   13      907  793.6508       110         90        1700
## [14,]   14      980  793.6508       110         96        1800
## [15,]   15     1046  975.0567       110        102        1900
## [16,]   16     1123 1111.1111       110        108        2000
## [17,]   17     1250 1065.7596       110        114        2100
## [18,]   18     1402 1315.1927        47        276        2200
## [19,]   19     1361 1700.6803        22        624        2300
## [20,]   20      892 1088.4354       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.

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`.

```
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).

```
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] 1655.329
##
## $manager_results
##           resource_type scaring culling castration feeding help_offspring
## policy_1              1      NA      59         NA      NA             NA
##
## $user_results
##          resource_type scaring culling castration feeding help_offspring
## Manager              1      NA       0         NA      NA             NA
## user_1               1      NA      16         NA      NA             NA
## user_2               1      NA      16         NA      NA             NA
## user_3               1      NA      16         NA      NA             NA
## user_4               1      NA      16         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.

```
print(sim_4$X);
```

```
## [1] 1000
```

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.

```
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.

```
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 1292.517        16        248
## [2,]    2     1875 1950.113        10        400
## [3,]    3     1992 1791.383        10        400
## [4,]    4     1999 2063.492        10        400
## [5,]    5     1999 2335.601        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**.

Replenishing crops after consumption

Unlike with the gmse function, gmse_apply does not automatically assume that crop production should be replenished after a single time step. The second layer of the landscape holds crop production on a cell. This will be depleted if resources consume crops on the landscape.

```
sim_consume <- gmse_apply(land_dim_1 = 8, land_dim_2 = 8,
                          res_consume = 0.02, get_res = "Full");
print(round(sim_consume$LAND[,,2], digits = 2));
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,] 0.72 0.75 0.75 0.83 0.74 0.67 0.71 0.71
## [2,] 0.80 0.77 0.83 0.71 0.67 0.71 0.74 0.67
## [3,] 0.64 0.70 0.67 0.72 0.89 0.78 0.68 0.85
## [4,] 0.75 0.63 0.72 0.80 0.63 0.68 0.68 0.83
## [5,] 0.77 0.72 0.68 0.72 0.71 0.71 0.58 0.72
## [6,] 0.74 0.82 0.78 0.82 0.71 0.77 0.74 0.78
## [7,] 0.78 0.71 0.59 0.75 0.60 0.68 0.75 0.82
## [8,] 0.78 0.70 0.74 0.72 0.74 0.80 0.77 0.68
```

If we run this for five time steps using a loop in gmse_apply, then resources will continue to deplete crops on the landscape.

```
sim_old     <- gmse_apply(land_dim_1 = 8, land_dim_2 = 8,
                          res_consume = 0.02, get_res = "Full");
for(time_step in 1:5){
    sim_new                 <- gmse_apply(get_res = "Full", old_list = sim_old);
    sim_old                 <- sim_new;
}
print(round(sim_old$LAND[,,2], digits = 2));
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,] 0.19 0.12 0.15 0.15 0.15 0.12 0.11 0.14
## [2,] 0.16 0.12 0.16 0.17 0.14 0.14 0.15 0.12
## [3,] 0.21 0.16 0.14 0.12 0.21 0.17 0.17 0.12
## [4,] 0.17 0.13 0.12 0.10 0.14 0.14 0.16 0.10
## [5,] 0.18 0.13 0.18 0.12 0.16 0.14 0.09 0.14
## [6,] 0.15 0.16 0.18 0.18 0.14 0.17 0.17 0.19
## [7,] 0.12 0.15 0.15 0.17 0.19 0.12 0.12 0.10
## [8,] 0.10 0.13 0.21 0.17 0.12 0.19 0.17 0.14
```

Notice that the amount of crops on each cell has decreased substantially after five time steps. To replenish the crops after every repl time step, we can use the following code.

```
sim_old     <- gmse_apply(land_dim_1 = 8, land_dim_2 = 8,
                          res_consume = 0.02, get_res = "Full");
repl        <- 1;
for(time_step in 1:5){
    sim_new                 <- gmse_apply(get_res = "Full", old_list = sim_old);
    if(time_step %% repl == 0){ # If remainder of time_step / repl is zero
        sim_new$LAND[,,2] <- 1;
    }
    sim_old                 <- sim_new;
}
print(round(sim_old$LAND[,,2], digits = 2));
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
```

```
## [1,]    1    1    1    1    1    1    1    1
## [2,]    1    1    1    1    1    1    1    1
## [3,]    1    1    1    1    1    1    1    1
## [4,]    1    1    1    1    1    1    1    1
## [5,]    1    1    1    1    1    1    1    1
## [6,]    1    1    1    1    1    1    1    1
## [7,]    1    1    1    1    1    1    1    1
## [8,]    1    1    1    1    1    1    1    1
```

Now with the crop on the landscape replenishing every time step, each new time step starts with the landscape crop values set to 1. This is likely to be critical if simulating resources that must consume a certain amount to survive (`consume_surv > 0`) or reproduce (`consume_repr > 0`), and `gmse_apply` thereby provides some flexibility in terms of how frequently (and how much) landscape values change from one time step to the next.