Use of the gmse_apply function

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

A. Bradley Duthie13, Jeremy J. Cusack1, Isabel L. Jones1, Jeroen Minderman1,
Erlend B. Nilsen2, Rocío A. Pozo1, O. Sarobidy Rakotonarivo1,
Bram Van Moorter2, and Nils Bunnefeld1

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] 1113
##
## $observation_results
## [1] 1269.841
##
## $manager_results
## resource_type scaring culling castration feeding help_offspring
## policy_1 1 NA 57 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
```
Note that in the case above we have the total abundance of resources returned \((\text{sim}_1\$\text{resource\_results})\), the estimate of resource abundance from the observation function \((\text{sim}_1\$\text{observation\_results})\), the costs the manager sets for the only available action of culling \((\text{sim}_1\$\text{manager\_results})\), and the number of culls attempted by each user \((\text{sim}_1\$\text{user\_results})\). 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 \text{res\_mod} instead.

```r
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 \text{gmse\_apply} to use it instead of the predefined GMSE resource model.

```r
sim_2 <- gmse\_apply(res\_mod = alt_res, X = 100, rate = 0.3);
```

The \text{gmse\_apply} function will find the parameters it needs to run the \text{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 \text{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 \text{gmse\_apply}, then an error will be returned. Results for the above \text{sim}_2 are returned below.

```r
print(sim_2);
```

## $\text{resource\_results}$
## [1] 128

## $\text{observation\_results}$
## [1] 90.70295

## $\text{manager\_results}$

### policy_1  resource_type scaring culling castration feeding help\_offspring
## policy_1  1 NA 61 NA NA NA

## $\text{user\_results}$

### Manager  resource_type scaring culling castration feeding help\_offspring
## Manager  1 NA 0 NA NA NA

### user_1  resource_type scaring culling castration feeding help\_offspring
## user_1  1 NA 16 NA NA NA

### user_2  resource_type scaring culling castration feeding help\_offspring
## user_2  1 NA 16 NA NA NA

### user_3  resource_type scaring culling castration feeding help\_offspring
## user_3  1 NA 16 NA NA NA

### user_4  resource_type scaring culling castration feeding help\_offspring
## user_4  1 NA 16 NA NA NA

## tend\_crops kill\_crops
## Manager NA NA

## user_1 NA NA

## user_2 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 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] 1269.841
##
## $manager_results
## [1] 269.8413
##
## $user_results
## [1] 296.8254

resource_type scaring culling castration feeding help_offspring
policy_1  1   NA  65   NA   NA   NA

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
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);
## 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);
```

```
## Time Pop_size Pop_est Scare_cost Cull_cost Scare_count Cull_count
## [1,] 1 1062 1269.8413 NA 25 NA 240
## [2,] 2 918 861.6780 NA 110 NA 54
## [3,] 3 997 1179.1383 NA 39 NA 150
## [4,] 4 997 1111.1111 NA 62 NA 96
## [5,] 5 1193 1088.4354 NA 79 NA 72
## [6,] 6 1335 1360.5442 NA 19 NA 312
## [7,] 7 1201 1043.0839 NA 110 NA 54
## [8,] 8 1349 1201.8141 NA 35 NA 168
## [9,] 9 1450 1632.6531 NA 11 NA 540
## [10,] 10 1100 1133.7868 NA 52 NA 114
## [11,] 11 1174 997.7324 11 109 0 54
## [12,] 12 1334 1156.4626 43 47 0 126
## [13,] 13 1481 1859.4104 84 10 0 600
## [14,] 14 1070 1065.7596 12 107 5 54
## [15,] 15 1248 997.7324 10 110 3 54
## [16,] 16 1437 1201.8141 45 35 0 168
## [17,] 17 1562 1337.8685 71 20 0 300
## [18,] 18 1555 1383.2200 88 18 0 330
## [19,] 19 1370 1269.8413 NA 26 NA 228
## [20,] 20 1382 1337.8685 NA 21 NA 282
```

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.
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>1189</td>
<td>952.3810</td>
<td>109</td>
<td>24</td>
<td>500</td>
</tr>
<tr>
<td>2</td>
<td>1283</td>
<td>1247.1655</td>
<td>12</td>
<td>300</td>
<td>600</td>
</tr>
<tr>
<td>3</td>
<td>1098</td>
<td>1247.1655</td>
<td>17</td>
<td>246</td>
<td>700</td>
</tr>
<tr>
<td>4</td>
<td>995</td>
<td>1020.4082</td>
<td>110</td>
<td>42</td>
<td>800</td>
</tr>
<tr>
<td>5</td>
<td>1224</td>
<td>1337.8685</td>
<td>16</td>
<td>336</td>
<td>900</td>
</tr>
<tr>
<td>6</td>
<td>1070</td>
<td>1043.0839</td>
<td>110</td>
<td>54</td>
<td>1000</td>
</tr>
<tr>
<td>7</td>
<td>1199</td>
<td>1201.8141</td>
<td>35</td>
<td>186</td>
<td>1100</td>
</tr>
<tr>
<td>8</td>
<td>1196</td>
<td>1179.1383</td>
<td>43</td>
<td>162</td>
<td>1200</td>
</tr>
<tr>
<td>9</td>
<td>1240</td>
<td>1043.0839</td>
<td>110</td>
<td>66</td>
<td>1300</td>
</tr>
<tr>
<td>10</td>
<td>1413</td>
<td>1292.5170</td>
<td>29</td>
<td>288</td>
<td>1400</td>
</tr>
<tr>
<td>11</td>
<td>1377</td>
<td>839.0023</td>
<td>109</td>
<td>78</td>
<td>1500</td>
</tr>
<tr>
<td>12</td>
<td>1532</td>
<td>1179.1383</td>
<td>56</td>
<td>168</td>
<td>1600</td>
</tr>
<tr>
<td>13</td>
<td>1652</td>
<td>1882.0862</td>
<td>13</td>
<td>780</td>
<td>1700</td>
</tr>
<tr>
<td>14</td>
<td>1065</td>
<td>1224.4898</td>
<td>54</td>
<td>198</td>
<td>1800</td>
</tr>
<tr>
<td>15</td>
<td>1017</td>
<td>770.9751</td>
<td>110</td>
<td>102</td>
<td>1900</td>
</tr>
<tr>
<td>16</td>
<td>1117</td>
<td>1133.7868</td>
<td>99</td>
<td>120</td>
<td>2000</td>
</tr>
<tr>
<td>17</td>
<td>1196</td>
<td>1269.8413</td>
<td>52</td>
<td>240</td>
<td>2100</td>
</tr>
<tr>
<td>18</td>
<td>1159</td>
<td>907.0295</td>
<td>110</td>
<td>120</td>
<td>2200</td>
</tr>
<tr>
<td>19</td>
<td>1228</td>
<td>1156.4626</td>
<td>100</td>
<td>138</td>
<td>2300</td>
</tr>
<tr>
<td>20</td>
<td>1304</td>
<td>1541.9501</td>
<td>30</td>
<td>480</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] 1428.571
##
## $manager_results
## resource_type scaring culling castration feeding help_offspring
## policy_1 1 NA 67 NA NA NA
##
## $user_results
## resource_type scaring culling castration feeding help_offspring
## Manager 1 NA 0 NA NA NA
## user_1 1 NA 14 NA NA NA
## user_2 1 NA 14 NA NA NA
## user_3 1 NA 14 NA NA NA
## user_4 1 NA 14 NA NA NA
## tend_crops kill_crops
## Manager NA NA
## user_1 NA NA
## user_2 NA NA
## user_3 NA NA
## user_4 NA NA
```
Note that in the above output, the resource abundance has increased and is now 1500. But if we look at `sim_4$X`, the value is still 1000.

```r
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.

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

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

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

```r
## Time Pop_size Pop_est Cull_cost Cull_count
## [1,] 1 1500 1269.841 17 232
## [2,] 2 1875 2176.871 10 400
## [3,] 3 1992 1700.680 10 400
## [4,] 4 1999 1927.438 10 400
## [5,] 5 1999 2244.898 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.

```r
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,] 0.75 0.72 0.65 0.85 0.75 0.80 0.75 0.75 
# [2,] 0.75 0.75 0.75 0.75 0.75 0.85 0.80 0.80 
# [3,] 0.75 0.75 0.75 0.75 0.75 0.75 0.75 0.75 
# [4,] 0.75 0.75 0.75 0.75 0.75 0.75 0.75 0.75 
```

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

```r
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,] 0.75 0.72 0.65 0.85 0.75 0.80 0.75 0.75 
# [2,] 0.75 0.75 0.75 0.75 0.75 0.85 0.80 0.80 
# [3,] 0.75 0.75 0.75 0.75 0.75 0.75 0.75 0.75 
# [4,] 0.75 0.75 0.75 0.75 0.75 0.75 0.75 0.75 
```

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.

```r
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,] 0.75 0.72 0.65 0.85 0.75 0.80 0.75 0.75 
# [2,] 0.75 0.75 0.75 0.75 0.75 0.85 0.80 0.80 
# [3,] 0.75 0.75 0.75 0.75 0.75 0.75 0.75 0.75 
# [4,] 0.75 0.75 0.75 0.75 0.75 0.75 0.75 0.75 
```
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.