The GCF Computational Platform for DT Models: Version 0.11 Tutorial

Steffen Fürst (Global Climate Forum)

July 27, 2020

This text constitutes a tutorial for working with the GCF Computational Platform for DT Models, version 0, based on a simple example model. The basic ideas underlying the platform are explained in the document "A Computational Platform for DT Models" (Jaeger et al., June 28, 2020). A more detailed technical documentation will be available by mid July, 2020. Hence, the present document jumps right into the matter.

1 Prerequisites

This version of the GCF Platform is written in the R programming language[1], so a working R installation is required to do the tutorial. Precompiled binary distributions for Windows and Mac, as well as installation guidelines for Linux can be found under `https://cran.r-project.org/`. The tutorial describes how the platform can be used with the RStudio IDE[2]. Please be aware that RStudio does not come with R itself. The platform uses the `tidyverse` and `magrittr` libraries, but will install them automatically if they are not found in your R library path.

We tested the code with the following versions:

- R: 3.6 and 4.0
- `tidyverse`: 1.3
- `magrittr`: 1.5

In case you cannot execute the tutorial model without errors, please check your versions and update them accordingly. To update the packages, you can enter `install.packages(c("tidyverse", "magrittr"))` into the R console. If this does not help, please send a message to `steffen.fuerst@globalclimateforum.org`.

2 Model Example

To illustrate how one can work with the computational platform, in this tutorial we use a model with multiple buyers and sellers, where the buyers randomly choose a seller from a fixed subset of sellers at each step. This subset is different for each buyer.

2.1 Model Background

We have a simple but volatile market with n buyers, m sellers, and two goods called x and y. The two goods are joint products, so that each seller sells both commodities. Their units are chosen so that the production process yields the same quantity of each product. Buyers buy both commodities from a single seller, but at each step each buyer randomly selects a seller from a fixed subset of all

[1] Its aim is not computational performance, but an illustration of the underlying ideas.

[2] `https://rstudio.com/`

sellers. We are only interested in relative prices, so we set the price for good x to 1, while the price for good y is p.

At each step, each buyer has a fixed budget B for buying the commodities. The quantities are chosen using a Cobb-Douglas utility function:

$$\begin{aligned} \max_{x,y} \quad & u(x,y) = x^{\alpha} \cdot y^{1-\alpha} \\ \text{s.t.:} \quad & x + y \cdot p \leq B \end{aligned} \tag{1}$$

The solution of this optimization is:

$$\begin{aligned} x &= B \cdot \alpha \\ y &= \frac{B \cdot (1-\alpha)}{p} \end{aligned} \tag{2}$$

For illustrative reasons, we neglect production except for the assumption that x and y are joint products, so that the seller tries to find a price where $x = y$. In the case of a single buyer, the solution is $p = \frac{1-\alpha}{\alpha}$. However, in our case, the sellers encounter different buyers with different preferences α and a different budget B. So they start with a random price and adjust that price at each time step:

$$\begin{aligned} & p_t = \frac{d_y}{d_x} \cdot p_{t-1} \\ \text{where} \quad & d_x = \sum_{b \in \text{buyers}} x_b, \quad d_y = \sum_{b \in \text{buyers}} y_b \end{aligned} \tag{3}$$

It is easy to see that in the single buyer case, this will lead to $p = \frac{1-\alpha}{\alpha}$ after a single step.

2.2 Code

The code for this tutorial is available in the zip folder "DT_gcf_V0_Code" sent together with this text (from July 1 on the code will also be available at `https://globalclimateforum.org/GCF-Platform-for-DT-Models`.

You can look at the code by opening the `tutorial.R` file in RStudio (or any text editor). To view results – or to test your R installation – you can simply click on Source.

tutorial.R

Source on Save | Run | Source

```
1 if ("rstudioapi" %in% loadedNamespaces() && rstudioapi::isAvailable()) {
2     setwd(dirname(rstudioapi::getActiveDocumentContext()$path))
3 }
4 source("platform_v0.R")
```

After a while, you should see a plot in the `Plots` section of RStudio. This diagram shows the overall difference in demand for good x and good y. Unlike the one buyer/one seller example, there is always a surplus of one of the two goods, and there is no evidence of a convergence towards an equilibrium.

3 *Initialization*

We define two types of agents, `BUYER` and `SELLER` and two types of networks.

```
agentTypes <- c("BUYER", "SELLER")
edgeTypes <- c("KNOWN_SELLERS", "BOUGHT_FROM")
```

The first network, KNOWN_SELLERS, is fixed and describes the sellers known to each buyer. Since the direction of an edge determines the flow of information, and buyers needs the price information to calculate their demand, it is a network from sellers to buyers.

The second network BOUGHT_FROM will be constructed in the transition function called calcDemand. This network describes from which seller a buyer bought something, so its edges will go from the buyers to the sellers. Beside calcDemand which will be the implementation of equation (2), we will also implement a transition function with the name calcPrice, which implements equation (3).

We further define a list (a tibble in R, that is, the list elements may be of different types) of the model parameters: modelParams; other than the user-defined parameters – in this case the numbers of buyers and sellers and the number of sellers each buyer knows – it must contain the number of (virtual) processing elements numPEs to which the simulation will be distributed[3].

```
modelParams <- tibble(numBuyer = 50, numSeller = 5, knownSellers = 2, numPEs = 5)
```

To start a simulation, we must call the platform function[4]

```
initSimulation <- function(agentTypes,
                           edgeTypes,
                           modelParams,
                           stateInitFunc,
                           globalTypes,
                           globalInitFunc,
                           rasterTypes = NULL,
                           rasterInitFunc = NULL)
```

The next three function parameters for initSimulation are explained in the following; the last two, rasterTypes and rasterInitFunc are not relevant for this tutorial's model as it does not have a spatial dimension.

3.1 stateInitFunc

The stateInitFunc is a function that gets the modelParams as argument, and returns the inital model state, in form of a list of tables, with one table for each agent type and edge type. We want to initialize our buyers with values for α between 0 and 1, with values for B between 1 and 100 and the sellers initial prices p between 0.5 and 1.5. Also, we create the KNOWN_SELLERS network, by randomly selecting for each BUYER a number of sellers, where the number is given by the parameter modelParams$knownSellers.

So, our implementation of the stateInitFunc is the following:

```
initModel <- function(modelParams) {
  initState <- function(modelParams) {
    nb <- modelParams$numBuyer
    ns <- modelParams$numSeller
    list(
        "BUYER" = tibble(
            alpha = runif(nb),
            B = runif(nb, max = 100) %>% ceiling,
            x = 0,
            y = 0
        ),
        "SELLER" = tibble(p = runif(ns, min = 0.5, max = 1.5),
                          sum_y = 0)
    )
  }

  initNetwork <- function(modelParams, modelState) {
```

[3]In this version the distribution is a virtual one, the implementation uses a single CPU core for illustrating the principle. Also numPE must be smaller or equal then the smallest number of agents per type, so in our example numPE must be smaller or equal to min(numBuyer, numSeller).

[4]Platform functions are written in rubine red in this document.

```r
    sampleSellers <- function() {
      sample_n(modelState$SELLER, modelParams$knownSellers) %>% pull(ID)
    }

    list("KNOWN_SELLERS" =
           modelState$BUYER$ID %>%
           map(~ tibble(fromID = sampleSellers(),
                        toID = .x)) %>%
           bind_rows)
  }

  agentState <- initState(modelParams) %>% withStringID
  c(agentState, initNetwork(modelParams, agentState))
}
```

In our `initModel` function, we construct the agents without any ID, but we need those IDs for creating the edges. The `withStringID` function from the platform does this for us.

So we must first source the platform code,

```r
source("platform_v0.R")
```

and can then check the result of our `initModel` function:

```r
initModel(modelParams)
```

```
$BUYER
# A tibble: 50 x 5
   ID      alpha     B     x     y
   <chr>   <dbl> <dbl> <dbl> <dbl>
 1 BUYER1  0.189    90     0     0
 2 BUYER2  0.472    19     0     0
 3 BUYER3  0.228    10     0     0
 4 BUYER4  0.644    68     0     0
 5 BUYER5  0.703    74     0     0
# ... with 45 more rows

$SELLER
# A tibble: 5 x 3
  ID          p sum_y
  <chr>   <dbl> <dbl>
1 SELLER1 0.619     0
2 SELLER2 0.893     0
3 SELLER3 1.48      0
4 SELLER4 1.43      0
5 SELLER5 0.874     0

$KNOWN_SELLERS
# A tibble: 100 x 2
   fromID  toID
   <chr>   <chr>
 1 SELLER4 BUYER1
 2 SELLER1 BUYER1
 3 SELLER1 BUYER2
 4 SELLER2 BUYER2
 5 SELLER5 BUYER3
# ... with 95 more rows
```

As we can see, we build a list with three elements, two for our two agent types, and one for the KNOWN_SELLER network. We ignored the BOUGHT_FROM network in the initialization, as it will be constructed in the calcDemand transition functions, in which every buyer returns an edge to the seller from whom he bought the goods.

3.2 globalTypes and globalInitFunc

The initSimulation function further requires a globalInitFunc and the globalTypes list. Here "global" refers to the entities on the top layer in Figures 1 and 2 of the document "A Computational Platform for DT Models" (Jaeger et al., June 28, 2020), sent out along with this tutorial. For the example model, we want to know the trajectories of price and excess demand, so time series for the variables we want to observe have to be written in a simulation. Thus, in the globalInitFunc we create an empty tibble with the corresponding three columns, and name this tibble OBSERVER:

```
initObserver <- function(modelParams) {
    list("OBSERVER" =
             tibble(
                 p = numeric(),
                 sum_x = numeric(),
                 sum_y = numeric()
             ))
}
```

The modelParams argument from the initSimulation call will also be available as part of the global layer, in form of a tibble with the "type" MODELPARAMS. It is not necessary to take this into account for the globalTypes parameter, we list here only the types that are created in the globalInitFunc:

```
sim <- initSimulation(agentTypes = agentTypes,
                      edgeTypes = edgeTypes,
                      modelParams = modelParams,
                      stateInitFunc = initModel,
                      globalTypes = "OBSERVER",
                      globalInitFunc = initObserver)
```

4 *Transition functions*

The returned value from the initSimulation call is an R environment, which contains the complete state of our simulation. To modify the state using a transition function we call the platform function applyTransition(simulation, transitionFunction, networks, invariantTypes, distance), whereby only simulation and transitionFunction are mandatory parameters. A transition function has the form function(agent, type, modelState). This function is called for every agent in the explicit graph layer. The returned agents and edges are gathered by the platform and then concatenated to yield together with the invariant agents and edges the new state of the explicit graph layer. The function parameter agent contains the state of a single agent with the type type. The parameter modelState contains the part of the overall model state that is visible for the agent. In the current implementation all entitites of the Global Layer of Figures 1 and 2 in the document "A Computational Platform for DT Models" are visible. From the **Interagent Layer**, the states of all agents which are on a tail position of one of the specified networks are also included in modelState. As all nodes from the Spatial Layer in a Moore Neighborhood of the agent, provided that the corresponding raster is listed in the network parameter.

The state of nodes and edges of the **Interagent Layer** which have a type listed in invariantTypes are retained and cannot be changed by the agents in this transition. In the case that an agent returns an entity of an invariantTypes in the transition function, this entity will be discarded.

The following shows the visible model state for the agent with the ID BUYER1 and the associated network KNOWN_SELLERS:

```r
simCopy <- as.environment(as.list(sim, all.names=TRUE))
showVisibleState <- function(agent, type, modelState) {
  if (agent$ID == "BUYER1") {
    print(modelState)
  }
}
applyTransition(simCopy, showVisibleState, "KNOWN_SELLERS")
```

```
$BUYER
# A tibble: 0 x 6
# ... with 6 variables: PE <dbl>, ID <chr>, alpha <dbl>, B <dbl>, x <dbl>,
#   y <dbl>

$SELLER
# A tibble: 2 x 4
     PE ID          p sum_y
  <dbl> <chr>   <dbl> <dbl>
1     2 SELLER2 0.552     0
2     5 SELLER5 1.27      0

$KNOWN_SELLERS
# A tibble: 2 x 2
  fromID  toID
  <chr>   <chr>
1 SELLER2 BUYER1
2 SELLER5 BUYER1

$BOUGHT_FROM
NULL

$OBSERVER
# A tibble: 0 x 3
# ... with 3 variables: p <dbl>, sum_x <dbl>, sum_y <dbl>

$MODELPARAMS
# A tibble: 1 x 4
  numBuyer numSeller knownSellers numPEs
     <dbl>     <dbl>        <dbl>  <dbl>
1       50         5            2      5
```

4.1 `calcDemand`

The first transition function we implement calculates the demand for the goods x and y as shown in equation (2). The fact that the state is reconstructed, not modified, means that agents that do not apply this transition need to simply return their state, as will be seen for sellers below.

In the case that the agent is a `BUYER`, one of the available sellers is selected using the `sample_n` function. Then the agent updates its demand for the goods x and y and adds the new state to the `BUYER`s of the overall new model state.

The agent also adds an edge to the `BOUGHT_FROM` network, which is then used in the next transition function by the sellers to sum up the demand and calculate the new price.

The `KNOWN_SELLERS` network is constant, so we will add `KNOWN_SELLERS` to the list of `invariantTypes` when `applyTransition` is called. As the sellers do not change their state, `SELLER` will be also added to `invariantTypes`.

```
calcDemand <- function(agent, type, modelState) {
  if (type == "BUYER") {
    seller <- modelState$SELLER %>% sample_n(1)
    agent$x <- agent$B * agent$alpha
    agent$y <- agent$B * (1 - agent$alpha) / seller$p
    list("BUYER" = agent,
         "BOUGHT_FROM" = tibble(fromID = agent$ID, toID = seller$ID))
  }
}
```

4.2 calcPrice

In the `calcPrice` transition function, the sellers summarise all the goods x and y they sold. As in their model state there are only the buyers which have created a link in the `BOUGHT_FROM` network, they can do this by selecting the `x` and `y` columns from the `modelState$BUYER` tibble and then calculate the sum for those rows. Then the price is determined as shown in equation (3).

The network `BOUGHT_FROM` is only constructed temporarily for this transition function, so no agent returns an edge of this network and it will be also not added to the `invariantTypes`, so this parameter will be `c(BUYER,KNOWN_SELLERS)` to retain those agents and edges for the next iteration of the simulation.

```
calcPrice <- function(agent, type, modelState) {
  if (type == "SELLER") {
    quant <- modelState$BUYER %>% select(x, y) %>% summarise_all(sum)
    agent$sum_y <- quant$y
    agent$p <- if (quant$x > 0) { quant$y / quant$x * agent$p }
               else             { agent$p }
    list("SELLER" = agent)
  }
}
```

We could now run the simulation for e.g. ten iterations by a simple `for` loop:

```
for (i in 1:10) {
  applyTransition(sim,
                  transitionFunction = calcDemand,
                  network = "KNOWN_SELLERS",
                  invariantTypes = c("SELLER","KNOWN_SELLERS"))
  applyTransition(sim,
                  transitionFunction = calcPrice,
                  network = "BOUGHT_FROM",
                  invariantTypes = c("BUYER", "KNOWN_SELLERS"))
}
```

However, in this case we could only inspect the model state after the ten iterations, as the `sim` environment does not store the intermediary model states.

5 *Observer*

This is where the previously created empty `OBSERVER` tibble will be used: in each iteration, we add a row to this tibble.

The model state is distributed over different (currently virtual) processing entities. You can see this by calling `sim$getExplicitLayer()`, this will return a list with the length `numPEs` from our model parameters. We have e.g. on our first (virtual) processing entity the following part of the model state:

```
sim$getExplicitLayer()[1]
```

```
[[1]]
[[1]]$BUYER
# A tibble: 10 x 6
      PE ID      alpha     B     x      y
   <dbl> <chr>   <dbl> <dbl> <dbl>  <dbl>
 1     1 BUYER1  0.949    17 16.1   0.792
 2     1 BUYER6  0.362    58 21.0  55.2
 3     1 BUYER11 0.343    18  6.18 17.6
 4     1 BUYER16 0.591    12  7.09  5.77
 5     1 BUYER21 0.543    80 43.4  55.5
 6     1 BUYER26 0.184    81 14.9  60.4
 7     1 BUYER31 0.309    28  8.66 17.7
 8     1 BUYER36 0.620    19 11.8   6.60
 9     1 BUYER41 0.658    20 13.2   8.02
10     1 BUYER46 0.207    31  6.41 28.9

[[1]]$SELLER
# A tibble: 1 x 4
     PE ID          p sum_y
  <dbl> <chr>   <dbl> <dbl>
1     1 SELLER1 0.413  195.

[[1]]$KNOWN_SELLERS
# A tibble: 20 x 2
   fromID  toID
   <chr>   <chr>
 1 SELLER2 BUYER1
 2 SELLER5 BUYER1
 3 SELLER1 BUYER6
 4 SELLER2 BUYER6
 5 SELLER1 BUYER11
 6 SELLER5 BUYER11
 7 SELLER1 BUYER16
 8 SELLER4 BUYER16
 9 SELLER1 BUYER21
10 SELLER2 BUYER21
11 SELLER4 BUYER26
12 SELLER5 BUYER26
13 SELLER2 BUYER31
14 SELLER5 BUYER31
15 SELLER5 BUYER36
16 SELLER3 BUYER36
17 SELLER3 BUYER41
18 SELLER4 BUYER41
19 SELLER4 BUYER46
20 SELLER3 BUYER46

[[1]]$BOUGHT_FROM
# A tibble: 0 x 0
```

The framework does the parallelization mostly automatically. Until now we ignored this completely, except when adding the `numPEs` to our model parameters. However, updating the observer is now not as easy as in the non-distributed case. Therefore, there is the helper function `reduce` in

the environment returned from the `initSimulation` call. This function has the form `reduce(type, agentFunc, aggregationFunc)`. If we want, e.g., the minimal α of all buyers, we can call:

```
sim$reduce("BUYER", "alpha", min)
```

```
[1] 0.03635454
```

So `type` is the type of the agents for which we want to aggregate some information from the states. `agentFunc` is a function that gets the state of a single agent and returns a single value. Thereby, a single column name like the "alpha" is a shortcut for `function (agent){ agent$alpha }`. The `aggregationFunc` is a function that gets a list of the values return by the `agentFunc` as argument.

For our model, we want to calculate the average price. As sellers have almost certainly sold different amounts of good y, we cannot just call `reduce("SELLER", "p", mean)`, but need to calculate $\frac{\sum p*y}{\sum y}$:

```
## current is the current state of the "OBSERVER" node
## we add an new row to this, which has the demand and average price as columns
observeFunc <- function(current, sim) {
    current %>%
        bind_rows(
            tibble(
                sum_x = sim$reduce("BUYER", "x", sum),
                sum_y = sim$reduce("BUYER", "y", sum),
                p = sim$reduce("SELLER",
                               function(agent) { agent$p * agent$sum_y },
                               sum) / sim$reduce("SELLER", "sum_y", sum)
            ))
}
```

The thus defined `observeFunc` can then be used in the `updateGlobal` function:

```
sim <- updateGlobal(sim, "OBSERVER", observeFunc)
```

6 Simulation run

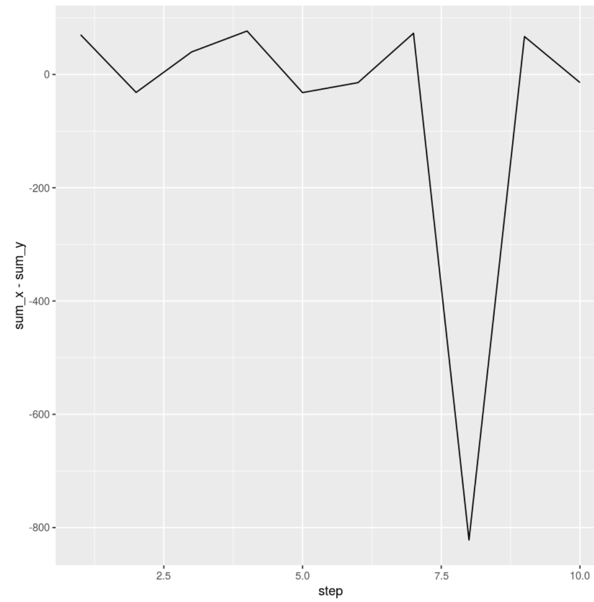
We can now run and observe simulations, e.g., accessing the OBSERVER table via `sim$global$OBSERVER` and creating a plot from this table, which shows the excess demand for good x. To conclude, we obtain a visualisation as shown below:

```
sim <- initSimulation(agentTypes = agentTypes,
                      edgeTypes = edgeTypes,
                      modelParams = modelParams,
                      stateInitFunc = initModel,
                      globalTypes = "OBSERVER",
                      globalInitFunc = initObserver)

for (i in 1:10) {
  applyTransition(sim,
                  transitionFunction = calcDemand,
                  network = "KNOWN_SELLERS",
                  invariantTypes = c("SELLER","KNOWN_SELLERS"))
  applyTransition(sim,
                  transitionFunction = calcPrice,
                  network = "BOUGHT_FROM",
                  invariantTypes = c("BUYER", "KNOWN_SELLERS"))
  updateGlobal(sim, "OBSERVER", observeFunc)
}
```

```r
plot <- sim$global$OBSERVER %>%
    rowid_to_column("step") %>%
    ggplot(aes(x = step, y = sum_x - sum_y)) + geom_line()

print(plot)
```



With this we end the tutorial.
Feel free to play and experiment with the code!