LPJmL Runner

Jannes Breier

2023-01-20

LPJmL Runner is a lpjmlkit module of functions that have the goal to simplify the execution of simulations with LPJmL and further to execute complex, nested and multiple simulation sequences fast and less error prone without having a big (bash) script overhead.

Setup

Please make sure to have set the working environment for LPJmL correctly if you are not working on the PIK cluster (with Slurm Workload Manager). On the PIK cluster please load the lpjml module (below) or add it to your `".profile"`.

```
# load lpjml module to use LPJmL internal functions
module load lpjml
```

Overview

The LPJmL Runner generally requires 3 to 4 working steps: Define a modified parameter table (1), create the corresponding configuration files (2), check if the these are valid for LPJmL (3 - optional) and run or submit LPJmL with each configurations (4).

1. Define a table of modified configuration parameters

Define what LPJmL parameters/settings (here all referred to as parameters) to be changed. They can be changed directly in the corresponding `"js"` file or in a `tibble` table (see example). For a single simulation this is a matter of personal routine, but when it comes to multiple runs where these parameters differ from each other, they have to be specified in a `tibble`.
`?write_config` *for more information.*

```
my_params <- tibble(
  sim_name = c("scenario1", "scenario2"),
  random_seed = c(42, 404),
  param.k_temp = c(NA, 0.03),
  new_phenology = c(TRUE, FALSE)
)
```

2. Create corresponding Configuration files

Now the central function is `write_config`, create and write LPJmL Configuration (Config) file(s) `"config_*.json"` from a table (`tibble`) with the parameters of a base `"lpjml.js"` file to be changed. `?write_config` *for more information.*

```
config_details <- write_config(my_params, model_path, output_path)
```

3. Check validity of Configurations

Check whether your Config(s) are valid for LPJmL by passing the returned `tibble` to `check_lpjml`. It won't raise an error (dependencies might not be satisfied yet) but will print/return the information of `lpjcheck`.

```
lpjml_check(config_details, model_path, output_path)
```

4. Run or submit LPJmL

Run LPJmL for each Configuration locally via `run_lpjml` or submit as a batch job to SLURM (PIK Cluster) via `submit_lpjml`. `run_lpjml` can also be utilized within slurm jobs to execute multiple single cell runs. `?submit_lpjml` *or* `?run_lpjml` *for more information.*

```
# run interactively
run_details <- run_lpjml(config_details, model_path, output_path)

# OR submit to Slurm
submit_details <- submit_lpjml(config_details, model_path, output_path)
```

miscellaneous

More helpful functions that come with LPJmL Runner are:

- `read_config` to read a `"config_*.json"` file as a nested R list object
- use the R internal `View` function for a tree visualization of a `"config_*.json"` file
- `make_lpjml` function for compiling LPJmL.

Usage

```
library(lpjmlkit)
# why tibble? -> https://r4ds.had.co.nz/tibbles.html
library(tibble)

model_path <- "./LPJmL_internal"
output_path <- "./my_runs"
```

Single cell simulations

Single cell (or short number of multiple cells) simulations can be executed locally or on a login node. This mode is especially useful when it comes to testing or comparing local data.

Example *Potential natural vegetation and land-use run*

```
# create parameter tibble
params <- tibble(
  sim_name = c("spinup", "lu", "pnv"),
  landuse = c("no", "yes", "no"),
  # only for demonstration
  nspinup = c(1000, NA, NA),
  reservoir = c(FALSE, TRUE, FALSE),
  startgrid = c(27410, 27410, 27410),
  river_routing = c(FALSE, FALSE, FALSE),
  wateruse = c("no", "yes", "no"),
  const_deposition = c(FALSE, FALSE, TRUE),
  # run parameter: dependency sets the restart paths to the corresponding
  #   restart_filename and calculates the execution order
  dependency = c(
    NA, "spinup", "spinup"
  )
)

# write config files
config_details <- write_config(
  params = params, # pass the defined parameter tibble
  model_path = model_path,
  output_path = output_path,
  js_filename = "lpjml.js" # (default) the base js file
)

# read and view config
config_lu <- read_config(
  filename = paste0(output_path,"/configurations/config_lu.json")
)
View(config_lu)

# check config & LPJmL
check_config(
  x = config_details, # can be filename (vector) or tibble
  model_path = model_path,
  output_path = output_path
)

# execute runs sequentially
run_details <- run_lpjml(
  config_details,
  model_path = model_path,
  output_path = output_path)
```

Example *Old vs. new phenology and old land-use vs. input toolbox*

```r
# create parameter tibble
params <- tibble(
  sim_name = c("spinup_oldphen",
               "spinup_newphen",
               "oldphen",
               "old_lu",
               "lu_toolbox"),
  # object oriented like syntax to access nested json elements
  input.landuse.name = c(
    NA,
    NA,
    NA,
    NA,
    "input_toolbox_30arcmin/cftfrac_1500-2017_64bands_f2o.clm"
  ),
  nspinup = c(1000, 1000, NA, NA, NA),
  new_phenology = c(FALSE, TRUE, FALSE, TRUE, TRUE),
  startgrid = c(27410, 27410, 27410, 27410, 27410),
  river_routing = c(FALSE, FALSE, FALSE, FALSE, FALSE),
  dependency = c(NA, NA, "spinup_oldphen", "spinup_newphen", "spinup_newphen")
)

# write config files
config_details <- write_config(params, model_path, output_path)

# check config & LPJmL
check_config(config_details, model_path, output_path)

# execute runs sequentially
run_details <- run_lpjml(config_details, model_path, output_path)
```

Global simulations on the PIK cluster

Global simulations are simulations on all available cells with a coherent water cycle. It requires more computational ressources which is why they have to be run at dedicated compute nodes, at PIK Cluster only accessible via SLURM Job scheduler. Therefore LPJmL has to be "submitted".

Example ***Compare old vs new land use (lpjml input toolbox)***

```
# create parameter tibble
params <- tibble(
  sim_name = c("spinup",
               "old_lu",
               "lu_toolbox"),
  input.landuse.name = c(
    NA,
    NA,
    "input_toolbox_30arcmin/cftfrac_1500-2017_64bands_f2o.clm"
  ),
  dependency = c(NA, "spinup", "spinup"),
  # slurm option wtime: analogous to sbatch -wtime defines slurm option
  #   individually per config, overwrites submit_lpjml argument
  #   (same for sclass, ntasks, blocking)
  wtime = c("15:00:00", "3:00:00", "3:00:00")
)

# write config files
config_details <- write_config(
  params = params,
  model_path = model_path,
  output_path = output_path,
  output_list = c("vegc", "soilc", "cftfrac", "pft_harvestc", "irrig"),
  output_list_timestep = c("annual","annual", "annual", "annual", "monthly"),
  # output_list_timestep = "annual",
  output_format = "clm"
)

# check config & LPJmL
check_config(config_details, model_path, output_path)

# submit runs to slurm
run_details <- submit_lpjml(
  x = config_details,
  model_path = model_path,
  output_path = output_path,
  group = "open")
```

Notes & tips

1. You can save the generated config tibble by applying `saveRDS` to it to reuse for a rerun or resubmission next time ...

```
saveRDS(config_details,
        paste0(output_path, "/configurations/config_details.rds"))

# next time ...
config_details <- readRDS(paste0(output_path,
                                 "/configurations/config_details.rds"))
```

2. Also if you want do not want to submit all runs you can ...

```
# use a subset for the rows - in this example you may only want to resubmit the
#   transient runs
run_details <- submit_lpjml(
  x = config_details[2:3, ],
  model_path = model_path,
  output_path = output_path,
  group = "open")
```

3. *a bit dirty though* If you want to reuse an old spinup simulation, you can copy the file or create a symlink of the file to `"<output_path>/restart/<spinup_sim_name>/restart.lpj"`. Make sure the file/symlink is named `"restart.lpj"`