April, 2024

Google Summer of Code, 2024 : The PEcAn Project

By

Abhinav Pandey

Project Title : Optimize PEcAn for freestanding use of single packages

Project Mentors : Chris Black (@infotroph), Shashank Singh (@moki1202)

Organization : The PEcAn Project

Project Duration : 350 hours

Table of Content :

Project Overview

The objective of this project is to enhance the PEcAn Project by optimizing its modules for standalone use. Despite PEcAn's robust framework and its interconnected modules, there's a growing need to make these modules independently operable. This shift is essential to simplify module usage, testing, and development, making the system more accessible and efficient for users and contributors.
The focus is on optimizing our modules for standalone use, enhancing their individual operability within PEcAn's interconnected framework with our top priority being to 're-loosen these couplings by revisiting the design and interface of PEcAn packages .'

Contributor Introduction

Personal Background

Name : Abhinav Pandey
Major : Metallurgical Engineering (Bachelor's)
University : Indian Institute of Technology (IIT), Varanasi
Current Year of Study : Sophomore
Github Handle : Sweetdevil144
Linkedin Handle : abhinav-pandey-441504252/
Slack Username : Abhinav Pandey

Email : abhinavpandey1230@gmail.com
Phone : +91 70681 92135
Timezone : Indian Standard Time (IST : UTC + 5:30)

Hello. I'm Abhinav Pandey, a Sophomore at IIT Varanasi pursuing my Bachelor's degree in Metallurgical Engineering. I am a Software Engineer who loves to program multiple solutions for day to day problems. I have a keen interest in Environmental Science and am passionate about leveraging programming as a tool to enhance scientific research. My focus lies in utilization of Programming Languages to handle and analyze large-scale data sets in environmental studies. I also like Systems Programming (as a hobby) and Test Driven Development.
I am deeply fascinated by the role of open source technologies in driving technological advancement, as evidenced by everything from **Linux** to **The PEcAn Project** and how

it has helped in the utilization of technology for an ordinary citizen. This enthusiasm is what motivates my desire to contribute to The PEcAn Project.

Purpose of Proposal

The purpose of this proposal is to outline my planned contribution to The PEcAn Project through the Google Summer of Code program. It aims to present a comprehensive plan that includes the project's objectives, my approach, methodology, and a detailed timeline. This proposal is designed to demonstrate my understanding of the project's requirements, my technical skills, and my commitment to contributing meaningfully to the open-source community. Through this initiative, I aspire to enhance the functionalities and capabilities of the PEcAn Project, while simultaneously honing my skills and contributing to the broader field of environmental science and technology.

Previous Contributions to The PEcAn Project

I've been an active member of The PEcAn Project since November, 2023, that is, for the past 5 months. I've been actively engaged in the community over both Slack and Github and readily engaged in discussion about our current and upcoming plans for The PEcAn Project. I have **9+ Pull Requests merged/ready to be merged** and **10+ Issues Opened** to date and **many more to come** in upcoming days within our `PEcAn` repositories. In the course of my engagement with the organization, I had the opportunity to interact with several mentors and project leads, gaining deeper insights and a better understanding of the PEcAn Project.

Additionally, I have been collaborating with Chris (@infotroph) on the `pecan-status-board and the main PEcAn` repository, focusing on enhancing its functionalities, resolving pre-existing bugs, and setting the groundwork for this project's future tasks and objectives.

Links to my Pull Requests

Update cos_solar_zenith_angle function	Documentation Updates

Update book.yml	Update Icon Usage in Dashboard
Add missing dependencies	Configure config/config.R
Update server URLs in api	Update Card Styles
Update contributors.js	ADD ISSUE and PR Templates
Update r-lib version to v2	

Links to my Opened Issues

Packages out of Date	Uneven grid distribution in People Page
Feature : Issue and Pull Request Templates	Bug : HTTP Request Timeout
Error Loading Test Results	rpecanapi: ping() and get.status() Functions return Error
Outdated Documentations	CRAN Compliance Issue: Broken URLs
id variables not found in data: Workflow_id	Docker Image not compatible with arm64
Feature : Add Linter Packages	

Project Details

The PEcAn Ecosystem

The PEcAn testing setup is a blend of three key repositories, each playing its own unique role in our broader ecosystem. By emphasizing our focus and our goals over areas where these three interact will be critical for Optimization of our `packages`:

1. The PEcAn Project : This works as the brain of the operation. It's where most of our core code and models live. This repository is central to everything we do in PEcAn, containing the main algorithms and data models that drive our environmental forecasting and analysis. It serves as the home to our `R packages`. The primary challenge here is to maintain the interconnected nature of these functions while exploring areas where independence can be enhanced without compromising the overall system integrity.

2. rpecanapi : It connects with the PEcAn API, acting like a bridge that allows different parts of our system to talk to each other smoothly. It's essential for integrating various components of our ecosystem, ensuring they work together seamlessly. While essential for system integration, exploring ways to reduce over-dependence on this component for certain internal operations could streamline the overall process.

3. The pecan-status-board : The frontend dashboard of our project . It doesn't run tests by itself or has any major relevance to this project, but this is where we see the results of our integration tests displayed after it obtains results from `Github Actions Workflows'` results. This repository gives us a clear, visual representation of how our tests are doing, letting us monitor everything and diagnose issues quickly.

Our aim is to understand where the coupling between these different `components` is essential for the system's functionality, where it can be loosened and where it can't so as to make it more efficient and user-friendly for users.

Short-Term tasks

At the onset of our Project `Optimize PEcAn for freestanding use of single packages`, it's crucial to lay a strong foundation that will support the more complex tasks we aim to accomplish in the later stages. The Pre-GSoC tasks will be the tasks I will be doing before the GSoC Phase. These tasks will be designed to ensure that our core components are fully operational and optimized. These tasks are not just preliminary steps; they are the bedrock upon which our Short-Term as well as Long-Term objectives will be built and realized.

Objectives:

Following are the objectives of my Pre-GSoC Tasks :

1- Codebase familiarization

Achieving a deep understanding of the PEcAn project's codebase is a fundamental task for the successful optimization of its modules. Codebase familiarity not only involves grasping the current functionalities but also understanding the underlying architectural decisions and design patterns. A number of pre-existing features may help us find all dependencies of a particular package.
One interesting script for doing so is the `scripts/dependencies.R` which generates a dependency graph for a list of R packages. It uses the `igraph` package to create the graph and provides options to save the graph as a PNG. The function works by recursively scanning the dependencies of each package in the packages vector. It uses the `packageDescription` function to get the dependencies listed in the package's DESCRIPTION file. It then adds these dependencies to a data frame that stores the graph data and then plots the data using the `plot` feature of R. Another interesting feature I found within the codebase was the `scripts/generate_dependencies.R` file which is being used for generating a list of package dependencies within the PEcAn project.
In a similar way, a deep and thorough examination of the Codebase would greatly help us in defining pre-built features and analyzing the entire `ecosystem`.
This script uses the `purrr` and `dplyr` packages to extract the dependencies from each package's `DESCRIPTION` file, and saves the results to a `CSV` file.

2- Aging of Code

The PEcAn ecosystem stands as one of the largest ecosystems in fields of data modeling and forecasting with one of the most extensive and active online communities in this field. Despite its significant presence, the project faces challenges due to a somewhat static approach in its development cycle.

This stagnation has resulted in the PEcAn codebase not fully aligning with contemporary coding methodologies, practices, and concepts. *A crucial aspect of our project will involve a meticulous examination of outdated elements in these three repositories* ***so as to refine the sections of code which are related to this Project .***
The aging of the code has led to various complications, particularly noticeable when working on our models and conducting integration testing. A noticeable number of test

failures can be attributed directly to the lack of timely updates and modernization of our codebase.

This includes changes to the code across our `PEcAn modules` to make sure things move smoothly on the right track by executing the following steps:

Update Dependencies within our `module` functions : As a simple example source, let's take a look within our /modules/assim.batch/R/get.da.data.R file. A number of improvements can be made for such files as follow which may ensure removal of any bogus 3rd party dependencies which may not be required now, but have gone unnoticed and may need amendment now:

1. Use of `paste` function :
In R, it's often more readable to use the paste0 function or the glue package for string concatenation. For example, `paste(ameriflux.dir, year, "L2.csv", sep = "_")` could be rewritten as `paste0(ameriflux.dir, "_", year, "_L2.csv")`.

2. Filtering data :
As we proceed with our project, it's crucial to recognize that readability can be subjective and highly dependent on context. As we dive deeper into refining the `package` code, the focus will be on creating a balance between clarity, conciseness, and compliance with the standards of R package development. For example : Consider this section of code

```
observed <- observed[observed$DTIME > 60 & observed$DTIME < 305, ]
```

This could be more readable with the `dplyr` package's filter function as follow:

```
observed <- dplyr::filter(observed, DTIME > 60, DTIME < 305)
```

On closer inspection we find that the 'improved' definition does not significantly reduce complexity. This shows how tricky code reading is, especially within the constraints of R package development. This example is a reminder that

improving readability is not always an easy task; It requires careful consideration and sometimes the changes can be subtle or even minimal. In some cases, this might lead to adopting new coding practices like using `dplyr::filter`, while in others, retaining the original syntax may be more appropriate.
Ultimately, the focus will be on making the codebase more accessible and easier to work with, without compromising on efficiency or the specific requirements of R package code. This careful, context-sensitive approach to improving readability will ensure that the PEcAn project remains maintainable and user-friendly over a long period of time.

3. Error Handling :

The `stopifnot` function is used to check that all absolute values of `observed$FC` are less than or equal to 500. If this is not the case, an error is thrown, but the error message is not very informative. Using an if statement with a more informative error message would help ease our *debugging* process and save time.

This is how our code would look after applying the above changes:

```
if (!file.exists(samples.file)) {
PEcAn.logger::logger.error(glue::glue("{samples.file} not found, this file is
required by the get.da.data function"))
}

pfts <- names(ensemble.samples)
pfts <- pfts[pfts != "env"]

# OBSERVED
observed <- lapply(years, function(year) {
PEcAn.uncertainty::read.ameriflux.L2(paste0(ameriflux.dir, "_", year,
"_L2.csv"), year)
})
observed <- do.call(rbind, observed)
observed$yeardoytime <- observed$time

# filter out winter observations March 1 and November 1 are chosen based on
Yoshi's methods
observed <- dplyr::filter(observed, DTIME > 60, DTIME < 305)
if (!all(abs(observed$FC) <= 500, na.rm = TRUE)) {
stop("All absolute values of observed$FC must be less than or equal to 500")
```

```
}

# ENSEMBLE
ensemble.run.ids <- PEcAn.utils::get.run.id("ENS",
PEcAn.utils::left.pad.zeros(1:ensemble.size))
ensemble.x <- do.call(cbind, ensemble.samples[pfts])[1:ensemble.size, ]

# SENSITIVITY ANALYSIS
p.rng <- do.call(rbind, lapply(pfts, function(pft) {
# REST OF CODE CONTINUES TO REMAIN SAME
```

In a similar manner, a number of changes can be made to other important packages like `data.land`, `allometry` or `data.atmosphere` which would help deal with our Dependency tracking Issues even better.

Moreover, there's an opportunity to enhance PEcAn's unique approach to logging errors and messages. The PEcAn project extensively uses the PEcAn.logger package, which predates R's current condition-signaling system. As suggested by Chris, it's worth exploring the integration of the logger package with R's built-in condition-signaling mechanism. This would allow for the continued use of standard R functions like `stop()` and `warning()`, while still maintaining PEcAn's logging behavior.

For example, we could write `stop("Informative error message")` instead of `PEcAn.logger::logger.severe("Informative error message")`. This would keep the code in line with R's standards, while also utilizing PEcAn's logging system, offering a familiar experience to PEcAn users.

This approach to error handling and messaging achieves several goals:

1. It upholds R's standard practices, thereby keeping the codebase approachable for R programmers.

2. It enhances the clarity and utility of error messages where necessary, without unnecessary rewrites of functional code.

3. It leverages PEcAn's existing logging system in a way that aligns with R's evolving features.

3- Dependency and Data Flow Analysis

Dependency Analysis is important in comprehending the complex web of relationships between different segments of our ecosystem. Effective dependency analysis aids in crucial tasks including debugging, code refactoring, and optimizing overall performance, thereby streamlining our development lifecycle. Dependency analysis primarily revolves around identifying the packages upon which a given package relies.

This can be done using built-in R functions like `packageDescription`, which returns a list of packages listed in the `Depends`, `Imports`, and `Suggests` fields of a package's `DESCRIPTION` file.

Here's a simple function that uses `packageDescription` to get the dependencies of a package:

```r
get_dependencies <- function(package) {
desc <- packageDescription(package)
c(desc$Depends, desc$Imports, desc$Suggests)
}

# Example usage:
get_dependencies("PEcAn.allometry")
```

This function returns a vector of package names that the specified package depends on. However, it doesn't provide any information about the nature of these dependencies or how they are used in the package. Yet, we can obtain key information of what dependencies are being utilized by a package. This would help us in examination of all dependencies within our `modules`.

In our efforts to understand the various dependencies in the `PEcAn` project we have a valuable resource – the `pecan_package_dependencies.csv` file, which is generated during the `docker build` process. This file contains dependencies in the first list for each package

An example of how we can use this file to better identify dependencies for a particular package, such as PEcAn.allometry:

```r
# Loading the CSV file
dependencies <- read.csv("path/to/pecan_package_dependencies.csv")

# Filtering our dependencies for a specific package
specific_deps <- dependencies[dependencies$needed_by_dir == "PEcAn.allometry",
]

# Extracting the list of dependencies
package_dependencies <- specific_deps$package
```

This approach provides a simple and time-saving way to identify the dependencies of `PEcAn.allometry` (or any other specificity). It is important to note that although this method quickly tells us about direct dependencies, it does not go into how these dependencies are implemented in the package and requires an internal analysis love more, possibly including code analysis and data flow analysis. Understanding the nature of these dependencies and how they are implemented in a package is critical to our goal of optimizing the independent functionality of each module and minimizing unnecessary interactions.

In the context of the PEcAn project, Data flow analysis is particularly important due to the complexity and interconnectedness of the system. Understanding how data flows through the system can help us identify potential bottlenecks, redundancies, or areas of the code that could benefit from refactoring or optimization.

For instance, consider a snippet of code from the `model2netcdf.ED2.R` script in the PEcAn project:

```r
# create lat/long nc variables
lat <- ncdf4::ncdim_def("lat", "degrees_north",
vals = as.numeric(sitelat),
longname = "station_latitude")
lon <- ncdf4::ncdim_def("lon", "degrees_east",
vals = as.numeric(sitelon),
longname = "station_longitude")
```

```
# ----- put values to nc_var list
nc_var <- list()
for (i in seq_along(out_list)) {
rflag <- ed_res_flag[i]
#fcnx is either put_T_values() or put_E_values()
fcnx <- paste0("put_", gsub("-", "", rflag), "_values")
put_out <- do.call(fcnx, list(yr = y, nc_var = nc_var, var_list =
out_list[[rflag]],
lat = lat, lon = lon, start_date = start_date_real,
end_date = end_date_real))
nc_var <- put_out$nc_var
}
```

By performing a data flow analysis on this code, we can understand how the `lat`, `lon`, and `nc_var` variables are used and transformed. This can help us optimize the code, for example, by identifying unnecessary computations or potential points of failure within similar code sections.

As part of my proposal, I plan to perform a detailed data flow analysis on key parts of the PEcAn system. This will involve identifying the key data structures used in the system, understanding how they are passed between functions and modules, and how they are transformed. This analysis will be crucial when I start to refactor the code to use more standard data formats and remove unneeded dependencies. This part of the project has a special relevance to following information relayed by the project "*some packages expect inputs and outputs in data structures that are only generated by other PEcAn packages but might be more easily provided in standard well-known formats.*" and will be the key feature for identifying such sections and doing so. This `may` turn out to be a really `LONG` and `ENCOMPASSING` part of our Project but may prove to be worthwhile in the long run!

Long-Term tasks

Our short term goal is to refine the functionality of each module within the PEcAn project. This involves a thorough review and restructuring of the modules to ensure that each one operates efficiently as a standalone unit. Key to achieving this will be the identification and modification of exported functions.

For the sake of comfort, I will be utilizing the example of packages like `PEcAn.allometry`, `PEcAn.data.atmosphere` and `PEcAn.assim.batch` for visualization and simulation purposes.

1- Identification of Dependencies in `modules`

For a more detailed analysis, we can use the `pkgnet` package. `pkgnet` provides tools to analyze package dependencies, measure the complexity of functions, and visualize these characteristics in a clear and understandable way using Network Plots for the same.

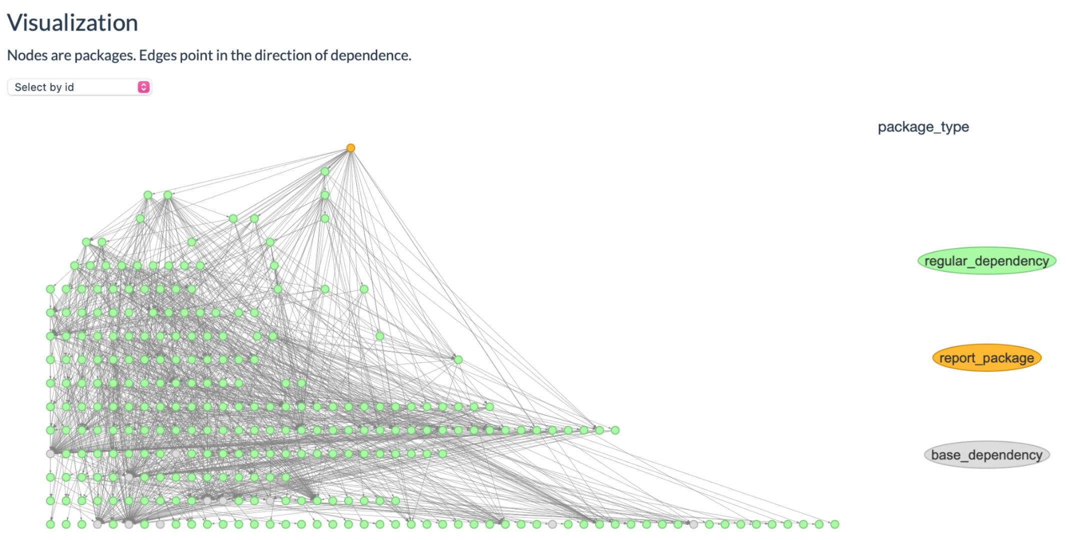
Here's an example of how I might use `pkgnet` to create a report for a package:

```
# Load the pkgnet package
library(pkgnet)

# Create a package report for the same
report <- CreatePackageReport(pkg_name = "PEcAn.data.land")
```

This will create a network plot showing the dependencies of the `PEcAn.allometry` package. Each node in the plot represents a package, and each edge represents a dependency. This can help us identify unneeded or exported dependencies that could be removed or implemented within `the single package`, to simplify and optimize our packages.

A number of questions arise when we want Identification of dependencies within our `packages`. *How can I visualize the dependency flow in my R package using a directed graph?*
One thing that I haven't portrayed is the powerful feature of `pkgnet` to make Network Plots of our dependency chart. One such network chart for our `PEcAn.data.land` is given as below :



This section analyzes the recursive package dependencies of **`PEcAn.data.land`**.

In the plot below, you'll see the following colors:

- gray: **`PEcAn.data.land`**
- orange: R packages available in every R session by default
- green: third-party R packages

We can see `HUNDREDS` of external dependencies, if not more than that. A detailed look into `module imports` is as follows :

```
Imports > coda, curl, datapack, dplyr, dplR, fs, future, furrr, httr,
lubridate, magrittr, mvtnorm, ncdf4 (>= 1.15), neonUtilities, neonstore,
swfscMisc, PEcAn.benchmark, PEcAn.data.atmosphere, PEcAn.DB, PEcAn.logger,
PEcAn.remote, PEcAn.utils, PEcAn.visualization, purrr, rjags, rlang, sf, sirt,
sp, stringr, terra, tidyr, tidyselect, traits, XML (>= 3.98-1.4)
```

We can see a high rate of dependence of `data.land` on `data.atmosphere` and other PEcAN modules. Some of them may be `necessary` for us to process and present data, but not all. Numerous instances of functional exports of `data.atmosphere` may be present within the `data.land` module. This is the main problem that we will need to scrounge and tackle at the start of this Project.

This complexity is not entirely avoidable given `data.land`'s role as an aggregator of data from various sources, many of which have their unique processing tools residing in external packages, such as `amerifluxr` and `neonUtilities`.

Two critical questions come to our mind when posed with such problems :

(1) "Is it better to use this functionality from the external package or to implement it ourselves inside data.land?" [I predict we'll usually decide it's better to use the external package.

(2) "Does every user of data.land need to install this dependency, or only people who want to use one specific dataset?" [I predict we'll find a lot of places where the dependency is only needed for a couple of functions, which sometimes means it can be moved from Imports to Suggests."

- *Original comments from Chris*

Necessity vs. Self-implementation: For each external functionality `data.land` relies on, we must assess whether it is more beneficial to utilize the existing external package or to internalize the functionality within `data.land`. More often than not, the answer may lean towards using the external package for efficiency and to avoid redundancy. For example, consider this snippet:

```
# Using amerifluxr package to fetch AmeriFlux data
library(amerifluxr)
ameriflux_data <- get_ameriflux(site_id = "AMERIFLUX_SITE_ID")
```

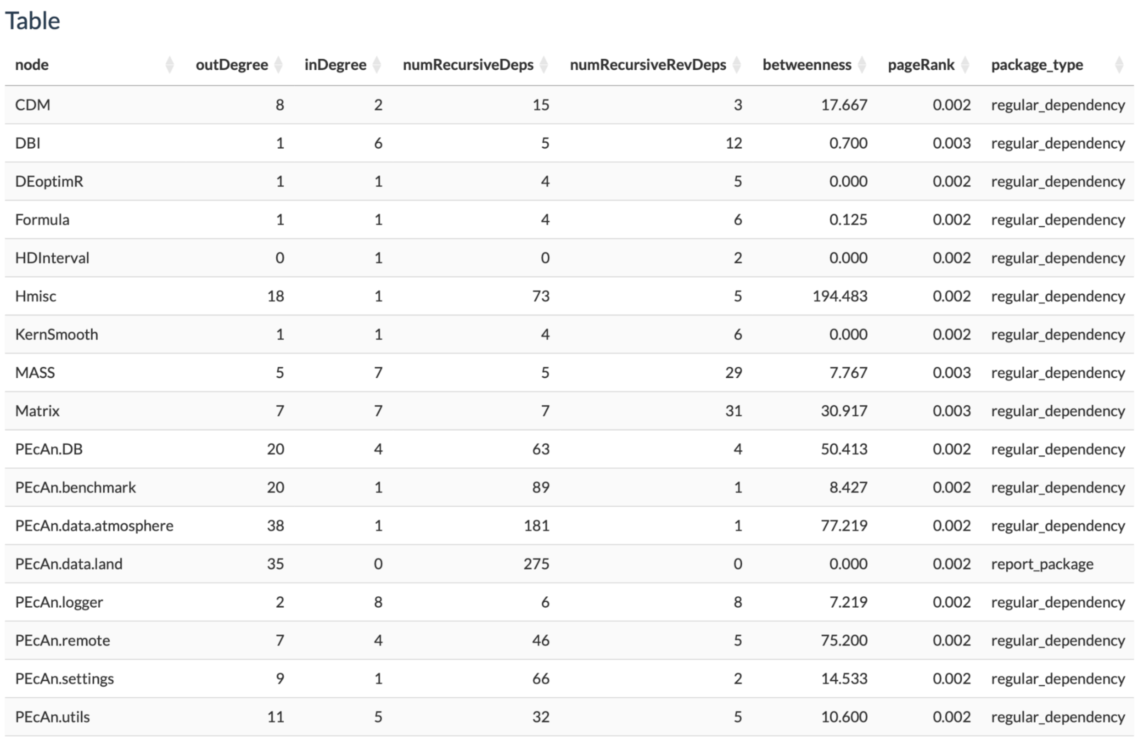
In cases like this, employing `amerifluxr` may be more advantageous than replicating its functionality within `data.land` given the vast complexity of `amerifluxr`.

Scope of Dependency Usage: We need to determine whether every `data.land` user requires all its dependencies or if some dependencies are specific to particular datasets or functions. In scenarios where a dependency is limited to a few functions, it might be more appropriate to classify it under Suggests rather than Imports. This distinction helps streamline the installation process for users who may not need the full suite of functionalities. For instance:

```
# Function utilizing a specific dependency
fetch_neon_data <- function(site_id) {
if (!requireNamespace("neonUtilities", quietly = TRUE)) {
stop("neonUtilities package is required for fetching NEON data")
}
neonUtilities::loadByProduct(dpID = "DP1.00001.001", site = site_id)
}
```

Here, `neonUtilities` is essential only for users who need to use `NEON` data, suggesting its potential movement from `Imports to Suggests.`

Given below is a table demonstrating the dependency of various packages `to and from` our `data.land` package :

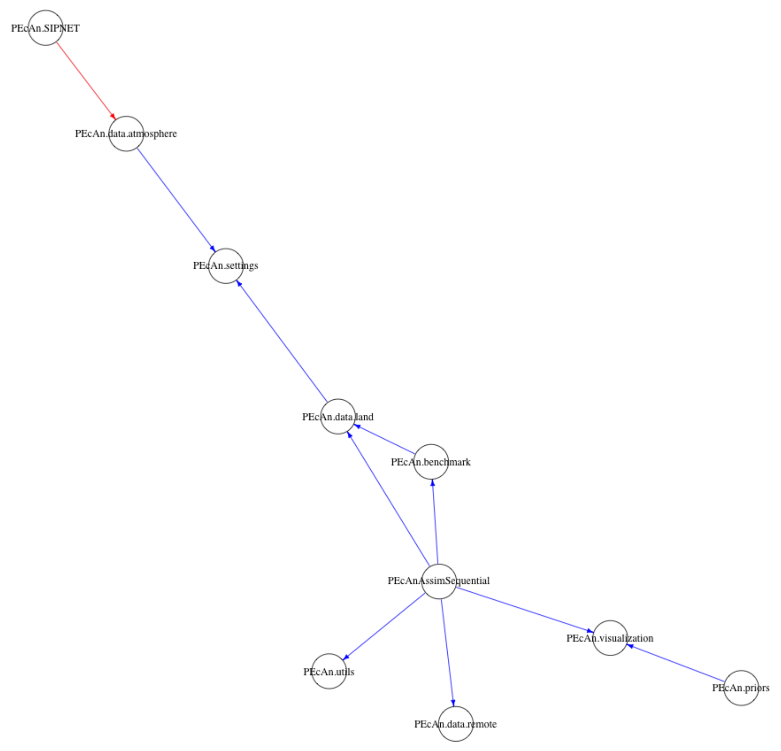


Table

node	outDegree	inDegree	numRecursiveDeps	numRecursiveRevDeps	betweenness	pageRank	package_type
CDM	8	2	15	3	17.667	0.002	regular_dependency
DBI	1	6	5	12	0.700	0.003	regular_dependency
DEoptimR	1	1	4	5	0.000	0.002	regular_dependency
Formula	1	1	4	6	0.125	0.002	regular_dependency
HDInterval	0	1	0	2	0.000	0.002	regular_dependency
Hmisc	18	1	73	5	194.483	0.002	regular_dependency
KernSmooth	1	1	4	6	0.000	0.002	regular_dependency
MASS	5	7	5	29	7.767	0.003	regular_dependency
Matrix	7	7	7	31	30.917	0.003	regular_dependency
PEcAn.DB	20	4	63	4	50.413	0.002	regular_dependency
PEcAn.benchmark	20	1	89	1	8.427	0.002	regular_dependency
PEcAn.data.atmosphere	38	1	181	1	77.219	0.002	regular_dependency
PEcAn.data.land	35	0	275	0	0.000	0.002	report_package
PEcAn.logger	2	8	6	8	7.219	0.002	regular_dependency
PEcAn.remote	7	4	46	5	75.200	0.002	regular_dependency
PEcAn.settings	9	1	66	2	14.533	0.002	regular_dependency
PEcAn.utils	11	5	32	5	10.600	0.002	regular_dependency

2- Creating Network Plots for all packages

Once we have analyzed which `packages` have what importance and what features they utilize from another package, we are ready to draw a network plot of dependence for each `model` and `module`. Such a dependency graph has been given below which was generated via `scripts/dependencies.R` file.



In a similar manner we may determine which `models` are dependent upon which `packages` along with knowing which `packages` are dependent on which `subsequent packages`?

3- Develop methods to track Dependencies for packages

Within the scope of optimizing the PEcAn project, the second key initiative is the functionality review and streamlining of its modules. This step is critical in enhancing the efficiency of the project. By functionality review, I always mean tracking and untangling all unnecessary functional imports from one package to another by either localizing the functionality or removing it.

To find out which functions from a specific package (e.g., `emulator`) are being used in another package (e.g., `data.land`), we would need to scan the source code of the `emulator` package for function calls to the `data.atmosphere` package.

Approach - 1 :

While R doesn't offer a built-in tool for this intricate task, there are resources within PEcAn's ecosystem that could be harnessed. For example, our `shiny/Pecan.depend` was a tool initially designed for visualizing our dependencies. However, it has been neglected for a long period of time during which some of its internal dependencies have been removed from `CRAN`. However, it is still fixable if we take an alternative approach to such removed `packages` via alternative implemented packages and if not, it still is a valuable resource for us to scrounge and look around for similar approaches.

Approach - 2 :

Alternatively, we may create a custom solution using the `tools` package, which provides tools for parsing and analyzing R code.

Here's a simplified example of how we might do this:

```
# file : scripts/findUtilization.R
library(tools) # For parsing R code

# Paths to the directories and NAMESPACE file
atmosphere_path <- "modules/data.atmosphere/R"
function_path <- "modules/emulator/R"
namespace_path <- "modules/emulator/NAMESPACE"

# Function to extract exported function names from NAMESPACE
extract_exported_functions <- function(namespace_path) {
lines <- readLines(namespace_path, warn = FALSE)
exported <- grep("export\\(", lines, value = TRUE)
exported_funcs <- gsub("export\\((.*)\\)", "\\1", exported)
return(exported_funcs)
}

# Function to extract function names from R files
extract_function_names <- function(path, exported_funcs) {
files <- list.files(path, pattern = "\\.R$", full.names = TRUE)
function_names <- c()
for (file in files) {
code <- readLines(file, warn = FALSE)
funcs <- getParseData(parse(text = code), includeText = TRUE)
```

```r
if (!is.null(funcs)) {
funcs <- funcs[funcs$token == "SYMBOL_FUNCTION_CALL", ]
function_names <- c(function_names, unique(funcs$text))
}
}
# Filter only exported functions
function_names <- intersect(function_names, exported_funcs)
return(unique(function_names))
}

# Extract exported functions from NAMESPACE
exported_data_functions <- extract_exported_functions(namespace_path)

# Extract function names from modules/emulator/R
data_functions <- extract_function_names(function_path,
exported_data_functions)

# Function to find usage of the functions in atmosphere files
find_function_usage <- function(atmosphere_path, data_functions) {
files <- list.files(atmosphere_path, pattern = "\\.R$", full.names = TRUE)
usage_list <- list()
for (file in files) {
code <- readLines(file, warn = FALSE)
for (data_function in data_functions) {
if (grepl(paste0("\\b", data_function, "\\b"), paste(code, collapse = "\n"))) {
usage_list[[data_function]] <- c(usage_list[[data_function]], basename(file))
}
}
}
return(usage_list)
}

# Find and list the usage
usage <- find_function_usage(atmosphere_path, data_functions)

# Print the result
print(usage)
```

The result is a list where each element is a vector of R files in the `data.atmosphere` package that call a particular function from the `emulator` package. Each element is named after the `emulator` function. A sample output of our results are such as below :

```
# Find and list the usage
> usage <- find_function_usage(atmosphere_path, data_functions)
>
> # Print the result
> print(usage)
$p
[1] "extract_FIA.R"               "fia2ED.R"
[3] "remote.copy.update.R"        "soilgrids_soc_extraction.R"

$distance
[1] "extract_NEON_veg.R"
```

In such a similar manner, we may proceed to create another script that utilizes this function to loop throughout our packages and determine **ALL** such instances where a function from one `package` may be exported to another `package`.

NOTE : *The Present script I designed may not be perfect and may also contain improper results but it will surely be refactored and results obtained by the start of our GSoC period.*

Approach - 3 :

We can also use the `grep` command in Linux to search for occurrences of `PEcAn.data.atmosphere::` across all files in the PEcAn project, excluding the files in the `/modules/data.atmosphere` folder. Here's the command:

```
> cd pecan
> grep -rl --exclude-dir=/modules/data.atmosphere "PEcAn.data.atmosphere::" ./
# we can similarly search for occurrences of various other packages within our local system and proceed to further refract it.
```

In this command:

- `grep` is the command used for searching text.
- `-r` tells grep to search recursively through directories.
- `-l` will list files with matches.
- `--exclude-dir=/modules/data.atmosphere` excludes the `/modules/data.atmosphere` directory from the search.
- `"PEcAn.data.atmosphere::"` is the text pattern to search for.
- `./` is the root directory in which we are searching, that is, `pecan`.

Further improvements can be made to this command for `package specific searchings` along with `package specific directories` further in our way.

<u>How This Manual Method Can Help and Its Limitations ?</u>

Using the `grep` command as described is a straightforward way to find where functions from `PEcAn.data.atmosphere` are being used in other parts of the PEcAn project. It can quickly give you an overview of the interdependencies within the project, highlighting which modules or files rely on `PEcAn.data.atmosphere`. In a similar way, we can further extend this automation by creating a bash script for this command which dynamically searches for different dependencies of all `packages`, except themselves, by looping through a list to do so.

However, this method is quite manual and repetitive, especially in a large project like PEcAn with numerous files. It requires examining each file individually to understand the context of the usage, which can be time-consuming. Moreover, `grep` does not provide information on how the functions are used or the nature of their dependencies. For a deeper analysis, more sophisticated tools or scripts would be needed to parse the code and extract more detailed information about these dependencies.

4- Refactoring code and Simplifying Packages

Now that we've identified the dependency of which `function` within which `package` on which `function` within which `package`, we're all ready to proceed with code refraction for the same.

<u>Objectives of Refactoring</u> :

a. Eliminating Unnecessary Dependencies: Our analysis of dependencies will guide us in pinpointing and eliminating non-essential cross-module dependencies. This process will involve assessing whether a function from one module, currently used in another, is critical to its operation or can be replaced with a local implementation.

b. Localizing Functions Where Possible: For dependencies deemed non-critical, the goal will be to localize functions within their respective modules. This could involve rewriting certain functions or implementing similar functionality directly within a module, thereby reducing reliance on external modules.

c. Simplifying Module Interfaces: A key aspect of this refactoring process will be the simplification of module interfaces. We'll focus on making them more intuitive and easier to use independently, which will also make the modules more accessible to new users and contributors. *A key tool for simplifying the interfaces will be to unexport functions that are only/primarily useful inside the package.*

d. Standardizing Data Formats: To further foster module independence, we'll work towards standardizing data formats across different modules. *This ensures that data passed between modules (where necessary) adheres to a common format, simplifying data handling and processing.*

For the sake of simplicity, we will try to take the example of our `data.land` and `data.atmosphere` as an example. Currently, if we grep the above results or run our script, we notice that the `PEcAn.data.land` has 4 such instances of dependencies of `PEcAn.data.atmosphere` as follow :

- Line 53 of `modules/data.land/R/ic_process.R`

```
#grab site lat and lon info
latlon <- PEcAn.data.atmosphere::db.site.lat.lon(site$id, con = con)
```

- Line 32 of `modules/data.land/R/soil_process.R`

```
# get site info
latlon <- PEcAn.data.atmosphere::db.site.lat.lon(site$id, con = con)
```

We have two similar occurrences of the function `db.site.lat.lon` within `data.land` imported from `data.atmosphere`. The `db.site.lat.lon function` fetches latitude and longitude for a given site ID. Similar information is frequently required in `data.land` as shown above. A parallel function could be developed within data.land. Hence, to remove such common dependencies is create a local local `db.site.lat.lon` within our `data.land` package as follow :

```
# file /modules/data.land/db.site.lat.lon.R :
function (site.id, con)
{
site <- PEcAn.DB::db.query(paste("SELECT id, ST_X(ST_CENTROID(geometry)) AS
lon, ST_Y(ST_CENTROID(geometry)) AS lat FROM sites WHERE id =",
site.id), con)
if (nrow(site) == 0) {
PEcAn.logger::logger.error("Site not found")
return(NULL)
}
if (!(is.na(site$lat)) && !(is.na(site$lat))) {
return(list(lat = site$lat, lon = site$lon))
}
else {
PEcAn.logger::logger.severe("We should not be here!")
}
}
```

Hence, Instead of a user's need to download the whole of `data.atmosphere` package while utilizing sole components of `data.land`, such dependencies are removed. This function could directly interact with the database to fetch site coordinates, thus eliminating the need to depend on `data.atmosphere` for this information.

db.site.lat.lon is a thin wrapper around a call to PEcAn.DB::db.query, making this a two-layer dependency! One possibility is that the function should live in PEcAn.DB and both data.atmosphere and data.land should call it from there.

- *Original Comments from Chris*

Therefore, a much better approach would be to bridge the dependency of `PEcAn.DB` with both our `data.atmosphere` and `data.land` packages via a `helper function`.

In a similar manner, the below issues too can be tackled by using specific defined methods above.

- Line 58 of `modules/data.land/R/write_ic.R`

```r
# Cohort2Pool -------------------------------------------------------------
# read in registration xml for pool specific information
register.xml <- system.file(paste0("register.", model$type, ".xml"), package =
paste0("PEcAn.", model$type))
if(file.exists(register.xml)){
register <- PEcAn.data.atmosphere::read.register(register.xml, con = NULL)
}else{
PEcAn.logger::logger.warn("No model register file found")
}
```

If these above calls are essential, we should explore ways to integrate similar functionalities within `data.land` or simplify their usage. If they are not critical, we might consider removing these dependencies altogether.

- Line 352 of `modules/data.land/inst/LoadPalEONsites.R`

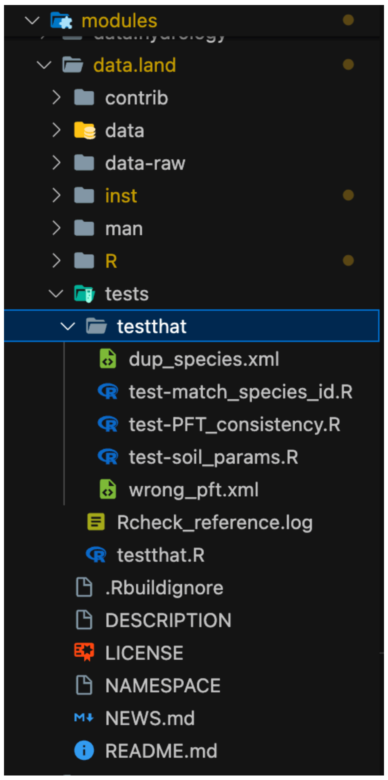
```r
for(j in seq_len(nrow(pecan.sgs))){
print(c(i,j))
## local folder
local.dir <- paste0(local.prefix,site.info$str_ns[j],"/")
if(!file.exists(local.dir) | length(dir(local.dir))==0) next
PEcAn.data.atmosphere::split_wind(local.dir,in.prefix,start_date,end_date)
}
```

We may create an internal `data.land/split_wind.R` for this case too as the following file.

5- Writing Tests for demonstration of Use of Packages

To effectively test the newly created local `db.site.lat.lon` function in the `PEcAn.data.land` package, we will employ the `testthat` package in R. This approach ensures that the function behaves as expected and maintains system integrity after refactoring. The testing process will be verifying both the functionality and the smooth functionality of the function. Below is a demonstration of how I will develop these tests:

We have a predefined test structure as follow for our current `package`, that is `data.land`



```
modules
  data.land
    contrib
    data
    data-raw
    inst
    man
    R
    tests
      testthat
        dup_species.xml
        test-match_species_id.R
        test-PFT_consistency.R
        test-soil_params.R
        wrong_pft.xml
      Rcheck_reference.log
      testthat.R
    .Rbuildignore
    DESCRIPTION
    LICENSE
    NAMESPACE
    NEWS.md
    README.md
```

For our `db.site.lat.lon` function, I'll write several test cases covering different scenarios:

1. Valid Input: Test the function with a valid `site.id` that exists in the database and ensure it returns the correct latitude and longitude.

2. Invalid Site ID: Pass an invalid` site.id` and assert that the function handles it gracefully, possibly returning NULL and logging an error.

3. Database Connection Issues: Simulate a scenario where the database connection fails or is unavailable to ensure the function handles such exceptions

appropriately.

I will create the test file `test-db.site.lat.lon.R` with the following structure:

```r
# Load necessary libraries
library(testthat)
library(PEcAn.data.land)

# Set the context for the tests
context("Testing split_wind function")

# Test that split_wind function handles valid inputs correctly
test_that("Handles valid inputs correctly", {
# Define valid inputs
in.path <- "valid/path"
in.prefix <- "valid_prefix"
start_date <- as.Date("2000-01-01")
end_date <- as.Date("2000-12-31")
overwrite <- FALSE
verbose <- FALSE
# Call the function with valid inputs
result <- split_wind(in.path, in.prefix, start_date, end_date, overwrite=FALSE,
verbose=FALSE)
# Check that the result is as expected
# The exact checks will depend on what the function is supposed to return
# Here are some examples:
expect_is(result, "list")
expect_true(all(c(local.dir,in.prefix,start_date,end_date) %in% names(result)))
})

# Test that split_wind function handles invalid inputs gracefully
test_that("Handles invalid inputs gracefully", {
# Define invalid inputs
in.path <- "invalid/path"
in.prefix <- "invalid_prefix"
start_date <- as.Date("2000-01-01")
end_date <- as.Date("2000-12-31")
overwrite <- FALSE
verbose <- FALSE
# Call the function with invalid inputs
result <- split_wind(in.path, in.prefix, start_date, end_date, overwrite, verbose)
# Check that the result is as expected
# The exact checks will depend on what the function is supposed to return when
given invalid inputs
# Here's an example for the same:
expect_null(result)
})
```

6- Functionality Review and streamlining

In the context of the `PEcAn.data.land` package, we have reviewed its dependencies and usage across different modules. The package is primarily used for handling and processing land data in the PEcAn project.

Understanding Current Functionality

Module-wise Analysis: Begin by cataloging the primary functions and features of each module within PEcAn, such as `PEcAn.data.land`, `PEcAn.allometry`, etc. This includes understanding their roles, inputs, outputs, and interactions with other modules.

Performance Metrics Collection: Utilize profiling tools (e.g., `profvis` in our case) to gather performance data for critical functions. This will help in identifying bottlenecks or inefficient code segments.

This is yet an optional opportunity still in consideration given the time complexity that only the Optimization part is going to cover. However, if time still persists we may tackle it during our GSoC Time Period, else we'll cover it as a part of my Post-GSoC Commitments.

Implementing Improvements

Refactoring Code: Based on the analysis, refactor code for improved performance and readability. Include comments and documentation for clarity.

User Interface Enhancements: If any module includes user-facing elements, ensure the interface is intuitive and user-friendly.

Detailed Brief of Tasks :

Below is a detailed brief/wrap-up of all our tasks as we discussed within this Proposal :

1. Aging of Code Analysis and Modernization: Conduct a comprehensive review of the current PEcAn codebase to identify outdated coding practices and areas for modernization. This includes updating legacy code to align with contemporary standards and best practices.

2. In-Depth Codebase Familiarization: Immerse deeply into the structure and functionalities of the PEcAn project. Understand the intricate details of each module, including their interactions and roles within the larger ecosystem.

3. Dependency and Data Flow Mapping: Analyze the dependencies among PEcAn modules and their data flow processes. Create a detailed map of these interdependencies to understand how data is exchanged and processed across the system.

4. Reviewing and Streamlining Module Functionalities: Scrutinize each module's functionalities for efficiency and relevance. Streamline processes where possible to enhance performance and user experience.

5. Revising Exported Functions: Evaluate the exported functions from each module, identifying those not core to the module's functionality. Consider removing or internalizing these functions to simplify the modules and reduce dependency complexities.

6. Creating Dependency Network Visualizations: Develop network plots for all PEcAn packages, illustrating their interdependencies. This visual representation will aid in identifying key areas for reducing complexity and improving module independence.

7. Refactoring for Reduced Module Dependencies: Engage in methodical refactoring to minimize unnecessary dependencies. Focus on simplifying module interfaces and standardizing data formats for inputs and outputs to facilitate independent module functionality.

8. Developing Tests and Examples for Independent Use: Design and implement tests and examples that demonstrate the autonomous use of individual modules. These will serve both as validation of independence and as instructional resources for users.

9. Tracking and Management of Essential Dependencies: Create methodologies to effectively track and manage critical dependencies that are intrinsic to the modules. This includes monitoring how these dependencies evolve over time, ensuring stability and consistency across different versions of the packages.

Expected Outcomes:

These tasks aim to transform the PEcAn project into a more modular and accessible framework. By enhancing the independence of its components, the project will become more user-friendly, encouraging broader use and contributions. The refactoring and streamlining efforts will also lead to a more maintainable and scalable codebase, facilitating future expansions and developments.

Relevance & Benefits to The PEcAn Ecosystem

Here is what we will be expecting to be the End results of this project as a whole, in collaboration with other mentors and Contributors :

- Refined Independence of Modules: Our chief aim is to disentangle the complex web of dependencies, as visible within our Network Plots, within PEcAn modules. By doing so, each module will not only operate more efficiently in isolation but also contribute to a streamlined, more coherent overall system. This refinement will ensure that individual components are optimized for standalone use, offering more versatility in their application.

- Enhanced Usability and Accessibility: By simplifying the interaction with individual modules, we're set to lower the barriers to entry. This translates to a user-friendly experience, where new users and developers can engage with the system without the overhead of understanding its entire architecture.

- Streamlined Development and Maintenance: In this initiative, every module becomes a more manageable and focused piece of the larger PEcAn puzzle. This approach promises a more efficient development and maintenance process, where updates, enhancements, and bug fixes can be implemented swiftly and with much higher precision.

- Scalability and Adaptation to New Challenges: By equipping each module to stand on its own, we make the PEcAn system adapt more fluidly to new scientific challenges and technological advancements.

Tech Stacks

1. Programming Languages : The project will extensively utilize the `R` programming environment, as specified and utilized by The `PEcAn` project. Essential skills include strong proficiency in R, a deep understanding of environmental data modeling, and expertise in software testing. Additionally, tools and languages like `Git` and `Bash scripting` will also be useful for effective collaboration.

2. Packages Required : I will be utilizing a number of packages throughout testing our Ecosystem. For Code Coverage, I will be utilizing the `covr` to identify the parts which lack testing, ensuring comprehensive code coverage. I am planning to use `testthat` package for testing our code. To analyze and optimize the performance of R code, profiler packages such as `profvis` will be useful. Packages like `pkgnet` for analyzing the network of dependencies within PEcAn modules. We also utilize `DiagrammeR` or `VisNetwork` for creating dependency network visualizations of our Packages after Identification of Dependencies. I will also utilize the `tools` package, which provides tools for parsing and analyzing R code. I will also decide on which other `packages` to utilize after a thorough discussion with the mentors.

Packages to be tackled first

Following packages have some of the highest package requirements as external dependencies, which can be reverified using `pkgnet` graphing OR simply viewed over here.

1. `data.atmosphere`
2. `data.land`
3. `assim.batch`
4. `assim.sequential`

5. `data.remote`

Project Timeline

April 2 - April 30 : Pre GSoC Period

- Week 1-2: Get familiar with `PEcAn's` code, set up your workspace. Intensive study of PEcAn's codebase before actually combating the issue. Develop an initial list of dependencies and a preliminary project improvement plan.

- Week 3-4: Plan out the project – decide which parts of PEcAn we have to focus on improving. Finalize key dependencies within the PEcAn modules; start drafting plans for tackling identified areas.

Milestones :

- Complete thorough study and understanding of the PEcAn codebase.
- Enhance initial list of dependencies and draft a preliminary improvement plan for the project.

May 1 - May 26: Community Bonding Period

- **Week 1-2 (May 1 - May 14):**

■ Build relationships with the PEcAn team. Set up biweekly meetings with mentors to discuss potential `packages` which will require our highest priorities in terms of identification and refactoring our dependencies. Current packages with the highest requirements are here

■ Finalize the project plan with advice from mentors. Get started with Key-Identification of Dependencies within our `modules` and `base` packages of highest priorities.

Milestones :

- ■ Implement Package Identification within our `base` and `modules`.
- ■ Interact with mentors to finalize `packages` of highest priority in our list.
- ■ Commute with mentors on a regular basis via biweekly meetings.

- **Week 3-4 (May 15 - May 26):**

■ Start with basic refraction of `modules` and `base` packages while at the same time keeping a check if the system breaks or not. Conduct biweekly meetings through the tenure of the Project. Keep informing the mentors about the progress of our Project and all major decisions. Explore complications within our CI systems, specifically those pertaining to monitor our dependencies. Have a thorough discussion with mentors about future goals and aspirations for such tasks.

■ Start exploring finalized PEcAn's `packages` to see how they're connected. Perform a detailed data flow analysis on key parts of the PEcAn system.

Milestones :

- Lay foundations for Dependency Tracking Mechanisms and test them with Refactoring key identified `packages` as a demonstration of successful implementation.
- Explore PEcAn Packages and respective testing schemes.

May 27 - June 30: Phase 1 - Identifying and Refactoring codebase

- **Week 1-4 (May 27 - June 9):**

■ Develop methods to track dependencies between `packages` that cannot be eliminated, including how these change between package versions.

■ Perform in-depth dependency analysis of selected modules; initiate code refactoring for simpler dependencies. Start improving the functionalities of chosen PEcAn `packages` with a goal to make them more independent and less tangled with each other.

Milestones :

- Develop methods for tracking inter-package dependencies.
- Test changes to ensure system functionality is maintained.

- **Week 3-4 (June 10 - June 30):**
  - Keep improving `modules` and `base` packages and start simplifying their connections.
  - Update any documentation to reflect changes.
  - Finish Refraction and Simplification of at least 2 `packages` before our Mid-Term Evaluation to demonstrate improvement of our ecosystem.

Milestones :

- Continue module improvements and connection simplifications.
- Update documentation to reflect the changes.

July 1 - July 8: Midterm Evaluation

- **Week 5 (July 1 - July 8):** I plan to conduct and prepare for the following while preparation of Midterm Evaluation begins.
  - Compile all the work done in Phase 1.
  - Prepare a detailed report highlighting achievements, challenges, and learnings.
  - Submit the midterm evaluation report after a thorough review with mentors.
  - Gather feedback from mentors and community for improvement.

July 9 - August 10: Phase 2 - Testing Phase

- **Weeks 1-2 (July 9 - July 22):**

○ Make sure the modules work well together after our changes. This Phase will be marked out by testing our newly modified `packages` with in-depth examples that would help demonstrate freestanding usage of `packages`. Develop and execute comprehensive test cases for each refactored module using `testthat`.

Milestones :

- Ensure cohesive module functionality post-changes.
- Develop and execute detailed test cases for each refactored module to cross-verify successful updation of our `modules` via Reducing, Improving and Changing Dependencies.
- Wrap Up Refraction and Improvement of Interdependencies of our High Priority `modules`.

- **Week 3-5 (July 23 - August 10):**

○ Finalize our improvements. Finish enhancement of our `packages`. Our work should be finished to an extent that we may call this project a success if we are able to perform the following tasks :

  - Reduce numerous direct and indirect dependencies for `[package]`.
  - Replace Outdated Dependencies (which were previously unnoticed) by Newest Dependencies for better results.
  - Minimizing calls from one `PEcAn` Package to another, either by localizing function, simplifying data interchanges and reducing calls to other external `packages`.
  - Once these `packages` have been refracted, simplified and tested thoroughly

○ Wrap up any last-minute testing and documentation updates. If all of the work for our pre-decided `packages` has been completed, we will then proceed with those `base` packages and `models` which have comparatively lower priorities (to be decided after thorough discussion with mentors).

Milestones :

- Finalize package improvements and complete the enhancement process.
- Wrap Up Testing and Documentation Phase.
- If time allows, engage with improvements of our lower priority `modules` for enhancement.

August 11 - August 26: Phase 3 - Finalizing and Documentation

- **Week 1 (August 11 - August 20):**
  - Put the finishing touches on your work.
  - Ensure comprehensive testing and clear documentation of all changes and new procedures; prepare for project presentation.
- **Final Week (August 21 - 26):**
  - Regular collaboration with the PEcAn team for further advancements.
  - Ongoing improvements based on community feedback.
  - Prepare and submit a detailed end-of-project report, reflecting on experiences and learnings.

Milestones :

- Collaborate with the PEcAn team for ongoing improvements.
- Prepare and submit a detailed end-of-project report.

September 3 - November 4: Post GSoC commitments

- Continued Contributions and Enhancements within the Project :
  - Engage in post-GSoC activities, including refining and enhancing our `packages` based on feedback by mentors.
  - Continue active participation in the PEcAn community and contribution to ongoing efforts and completion of remaining lower priority packages (if any) .
  - Regularly sync with mentors and the community for further collaboration opportunities.
  -
- Final Submission and Closure :
  - Prepare a final summary report of post-GSoC contributions.
  - Submit a reflective report on the overall GSoC experience and learnings.

    ○ Discuss potential future involvement and contributions to the PEcAn project.

Long-Term Commitment and Continuous Improvement

Beyond the GSoC period, my commitment to the PEcAn Project remains the same. My ongoing contributions will focus on Optimizing all `packages` within our `modules` and `base` directories. Continued test creations which will test our changes in repositories actively. I have been a part of The `PEcAn Project` for over five months now and plan to do so for a long term duration.

WHY ME? Most importantly, WHY PEcAn ?

My Journey with PEcAn

My involvement with the PEcAn project extends beyond just a professional capacity; it represents a crucial part of my journey in the world of open-source development and environmental research. Through my previous work with PEcAn, I've not only contributed to various aspects of the project but also deeply immersed myself in its ethos and objectives.

a) Previous Contributions: Detailing my past engagements with PEcAn, I've tackled several challenges and contributed to different modules. Each contribution has been a learning experience, helping me understand the nuances of the project more profoundly. From simple documentation updates to codebase changes, every Pull Request I opened helped me understand PEcAn way better than before.

b) Familiarity with the Codebase: Having spent considerable time navigating and working with the PEcAn codebase, I've developed an intimate understanding of its structure and functionality. This familiarity allows me to quickly identify issues and efficiently work towards resolutions.

Synergy with Mentors

One of the key aspects of my journey with PEcAn has been the continual interaction with mentors. These communications have not only sharpened my technical skills but also provided me with valuable insights into effective project management and collaboration within a diverse team. I have been communicating freely, clearly and concisely with Chris (@infotroph), Siddhant (@moki) as well as other mentors across the Project including Istem (@istemfer). Chris has been a truly wonderful guy. With clear communication of technical aspects while at the same time providing clarity on information every time I faced confusion and overwhelmed, which helped me gain confidence during the complete processing.

Alignment with Personal and Professional Goals

a) A Perfect Match: Choosing PEcAn was not coincidental; it was a deliberate decision driven by how closely the project aligns with my end goals. Working on environmental issues using data models similar to those in PEcAn resonates with my goals to contribute to sustainable environmental practices through technology and Programming that I've always dreamed about.

b) Future Aspirations: My long-term goal is to leverage my skills in software testing and data analysis to make significant contributions to environmental research. PEcAn serves as the ideal platform for this, allowing me to integrate my passion for technology with environmental stewardship.

Contribution to the Community

a) Adding Value: Beyond personal growth, my focus is on adding substantial value to the PEcAn community. Whether it's through developing new features, enhancing existing functionalities, or improving documentation, my aim is to contribute in ways that benefit the project and its users. PEcAn is completely maintained by community Volunteers and being a major contributor brings a sense of pride and integrity with it.

b) Community Engagement: Engaging with the community has given me insights into the needs and challenges faced by users. This perspective is crucial in shaping my contributions to be more user-centric and impactful.

Software Testing and Statistical Analysis

a) Testing Expertise: My keen interest in software testing, especially in a complex and dynamic environment like PEcAn, allows me to contribute significantly to maintaining and enhancing the quality and reliability of the software during the `testing` phase of our works.

b) Statistical Analysis with R: The use of R in statistical analysis and Data Modeling is the foundational block of the PEcAn project, and it aligns perfectly with my skill set. My proficiency in R not only enables me to contribute effectively but also to innovate and explore new methodologies within the project framework.

The PEcAn project offers a unique platform where I can apply my technical expertise, collaborate with like-minded individuals, and contribute to a field that is close to my heart – environmental sustainability. My journey with PEcAn is not just about what I have contributed so far, but about the potential of what we can achieve together in the future. My commitment to the project is unwavering, and I am excited about the opportunity to continue making meaningful contributions to this vibrant community.