Code-Base Documentation

Optimization Research Group

June 28, 2024

1 Introduction

This report includes three sections: Firstly, we will go over how to use this code base to create and optimize Multi Period Optimal Power Flow systems with additional components for Uncertainty, Linearization, and Local Search Optimization strategies. Second, we will explain the design in detail and how different components work together. Thirdly, we will showcase how development can continue, and how new optimization strategies can be added to the code-base. Lastly, we will go over some key points about the design Philosophy of this project.

2 Getting Started

In these following sections we will describe how to get started with using our code.

2.1 Before we Begin

To run the scripts, you need to have the following Julia packages installed:

- JuMP
- PowerModels
- Ipopt
- Gurobi

The packages can be installed with Julia's package manager:

```
using Pkg
Pkg.add("JuMP")
Pkg.add("PowerModels")
Pkg.add("Ipopt")
```

2.2 What to Include in your Workspace

To be able to run the code we need to use the packages we just installed, we need to include the **MPOPF.jl** file, and we need to explicitly say we want to use our defined **MPOPF** module. (Note: The period indicates that it is a local Module). This can be done with the following three lines of code.

```
using JuMP, PowerModels, Ipopt, Gurobi
include("MPOPF.jl")
using .MPOPF
```

2.3 Basic Example

Here is a simple example to showcase how a model can be created and optimized:

```
using PowerModels, JuMP, Ipopt, Gurobi
include("MPOPF.jl")
using .MPOPF

# We define the file path of the case we want to solve
file_path = "./Cases/case14.m"

# To create a DC model we need to first define a DC factory
# It is done with the following function
# Takes in two parameters, the fille path for the case we want
    to solve
# and the optimizer we want to use, Ipopt or Gurobi
dc_factory = DCMPOPFModelFactory(file_path, Ipopt.Optimizer)

# After creating our factory we pass it to our create model
    function
my_dc_model = create_model(dc_factory)

# Once we have our model we just optimize
# This will print the Minimum Cost
optimize_model(my_dc_model)

# If we want to make an AC model instead simply create it with
    an AC Factory
ac_factory = ACMPOPFModelFactory(file_path, Ipopt.Optimizer)
my_ac_model = create_model(ac_factory)
optimize_model(my_ac_model)
```

2.4 Multi-Period Example

To create a model with multiple periods we just specify the number of periods, the factors for the loads (multiplied to the current load to create different demand for the next period), and the ramping cost. They are specified in the `create_model` function.

```
using PowerModels, JuMP, Ipopt, Gurobi
include("MPOPF.jl")
using .MPOPF

# We define the file path of the case we want to solve
file_path = "./Cases/case14.m"

# Our DC factory
dc_factory = DCMPOPFModelFactory(file_path, Ipopt.Optimizer)

# Create the model as before but now with multiperiod variables
    specified
# Time Periods = 3
# One factor per time period
# Ramping Cost = 7
my_dc_model = create_model(dc_factory, 3, [1.0, 0.98, 1.03], 7)

# Once we have our model we just optimize
# This will print the Minimum Cost
optimize_model(my_dc_model)
```

3 How it works

3.1 Factory Structs

The Factories are used as parameters so that Julia's multiple dispatch feature runs the correct functions depending on the factory given.

We have two, `ACMPOPFModelFactory` and `DCMPOPFModelFactory` which are subtypes of the abstract type `AbstractMPOPFModelFactory`. This abstract type is what the implementation functions expect but since AC and DC are subtypes they will work. This makes it possible to create functions with the same names that preform different operations depending on the factory provided.

Here is the code for our Factory Structs:

```
# Abstract type as a base so that we can use this type as a
    parameter in fucntions
abstract type AbstractMPOPFModelFactory end

# This struct "inherits" from PowerFlowModelFactory
mutable struct ACMPOPFModelFactory <: AbstractMPOPFModelFactory
    file_path::String
    optimizer::Type

    function ACMPOPFModelFactory(file_path::String, optimizer::
    Type)
        return new(file_path, optimizer)
    end
end

# This struct "inherits" from PowerFlowModelFactory
mutable struct DCMPOPFModelFactory <: AbstractMPOPFModelFactory
    file_path::String
    optimizer::Type

    function DCMPOPFModelFactory(file_path::String, optimizer::
    Type)
        return new(file_path, optimizer)
    end
end
```

3.2 MPOPF Model Structs

The MPOPF Model objects are what the **`create_model`** function returns. They have all the information specific to MPOPF.

Similarly to our factory structs we currently have two concrete structs of MPOPF, **`MPOPFModel`** and **`MPOPFModelUncertainty`** which are subtypes of the abstract type **`AbstractMPOPFModel`**.

This is useful when we want both MPOPF models to be passed in a function interchangeably (That is the case for the **`optimize_model`** function).

`MPOPFModel` has the following as variables:

- **`Jump Model`**,
- **`Data read from file`**
- **`time periods`**
- **`Load Factors`**
- **`Ramping Cost`**

`MPOPFModelUncertainty` has the same variables except that it holds one more variable **`scenarios`** which is only relevant for Uncertainty.

Here is the code for out MPOPF Model Structs:

```
# Abstract type as a base so that we can use this type as a parameter in fucntions
abstract type AbstractMPOPFModel end

# The actual PowerFlowModel struct that "inherits" forrm AbstractPowerFlowModel
mutable struct MPOPFModel <: AbstractMPOPFModel
    model::JuMP.Model
    data::Dict
    time_periods::Int64
    factors::Vector{Float64}
    ramping_cost::Int64

    function MPOPFModel(model::JuMP.Model, data::Dict, time_periods::Int64=1, factors::Vector{Float64}=[1.0], ramping_cost::Int64=0)
        return new(model, data, time_periods, factors, ramping_cost)
    end
end

# Similar PowerFlowModel object but with an additional scenrios variable for uncertainty
mutable struct MPOPFModelUncertainty <: AbstractMPOPFModel
    model::JuMP.Model
    data::Dict
    scenarios::Dict
    time_periods::Int64
    factors::Vector{Float64}
    ramping_cost::Int64

    function MPOPFModelUncertainty(model::JuMP.Model, data::Dict, scenarios::Dict, time_periods::Int64=1, factors::Vector{Float64}=[1.0], ramping_cost::Int64=0)
        return new(model, data, scenarios, time_periods, factors, ramping_cost)
    end
end
```

3.3 Create Model Functions

At the moment we have two `create_model` functions. The first returns an `MPOPFModel` object and the second returns an `MPOPFModelUncertainty` object. The system knows which one to call depending on whether the scenarios variable was given.

Both share the same logic so whatever I explain about the first can be extrapolated to the second.

The `create_model` function takes in a factory of type `AbstractMPOPFModelFactory` as the first parameter. This means that both `ACMPOPFModelFactory`, `ACMPOPFModelFactory`, or any other Factory that inherits from the Abstract one is accepted.

The following three parameters:
`time_periods`, `factors`, and `ramping_cost`
are only relevant for multi-period so they are optional. (If not provided, the system will assume one period).

For AC and DC models, the steps of creating a model are the same. We first define the model variables, then we define the model objective function, and lastly we set the model constraints.

However, inside of these functions different things happen depending if we want AC, DC, or any other form of defining. This is why the factory is passed as a parameter inside the `set_model_variables!`, `set_model_objective_function!`, and `set_model_constraints!` functions.

Thanks to Julia's multiple dispatch feature, the correct function will be called depending on the type of the factory. Therefore, the correct variables, objective function, and constraints will be added without having to create massive if statements that check what model we want.

Here is our create model function:

```
function create_model(factory::AbstractMPOPFModelFactory,
    time_periods::Int64=1, factors::Vector{Float64}=[1.0],
    ramping_cost::Int64=0)::MPOPFModel
     data = PowerModels.parse_file(factory.file_path)
     PowerModels.standardize_cost_terms!(data, order=2)
     PowerModels.calc_thermal_limits!(data)

     model = JuMP.Model(factory.optimizer)

     power_flow_model = MPOPFModel(model, data, time_periods,
    factors, ramping_cost)

     set_model_variables!(power_flow_model, factory)
     set_model_objective_function!(power_flow_model, factory)
     set_model_constraints!(power_flow_model, factory)

     return power_flow_model
end
```

Here is an example of the AC and DC **set_model_variables!** functions. Take note of the factory type accepted.

AC:

```
function set_model_variables!(power_flow_model::
    AbstractMPOPFModel, factory::ACMPOPFModelFactory)
# Here would be the code for AC model
# Take note of the second parameter which accepts type
    ACMPOPFModelFactory
end
```

DC:

```
function set_model_variables!(power_flow_model::
    AbstractMPOPFModel, factory::DCMPOPFModelFactory)
# Here would be the code for DC model
# Take note of the second parameter which accepts type
    DCMPOPFModelFactory
end
```

The factory is not used for any computation, it is just there to let the system know which function should be called in what situation.

4 Future Development

To further improve the project and add more functionality to the system, there are two good things that can be done.

4.1 Similar Procedure as AC or DC

If the functionality that we want to add follows the same procedure for creating a model that AC or DC follow then we can follow these steps:

1. Create a new `MPOPFModelFactory` that "inherits" from `AbstractMPOPFModelFactory`. At the simplest case this can be identical to AC or DC factories with the name changed.
2. Create your new model functionality by implementing these three functions: `set_model_variables!`, `set_model_objective_function!`, and `set_model_constraints!`. Note that the factory passed to these functions should be your newly created factory.
3. That's it, now you can create a model with your new implementation with the `create_model` function and your factory passed to it.

4.2 Different Procedure as AC or DC

If the new functionality that we want to add does not follow the same steps then a little more work needs to be done.

Let's take uncertainty for example. Uncertainty should work for both AC and DC, it needs a new variable to handle scenarios and it modifies current constraints instead of adding on to them. Here are the steps I took to create it. Similar process can be taken for something new.

1. Since we need a new variable I created a new struct `MPOPFModelUncertainty` which is identical to `MPOPFModel` but with a new variable `scenarios`. It is also a subtype of `AbstractMPOPFModel`.
2. I then created a new `create_model` function that accepts this new variable `scenarios` as a parameter and returns a model of type `MPOPFModelUncertainty`. (The system will know which create model function to call depending on if the `scenarios` variable is provided).
3. I implemented the process for uncertainty inside this new `create_model` function.

5 Design Philosophy

The design of this project is grounded in several key principles aimed at ensuring flexibility, modularity, and ease of use. These principles guide the structure and development of the codebase, making it robust and adaptable to future development.

5.1 Modularity

The codebase is designed with modularity in mind. By defining abstract types and leveraging Julia's multiple dispatch feature, we allow for the seamless addition of new model types and functionalities. Each component, whether it be AC, DC, or uncertainty models, can be developed and maintained independently. This modular approach ensures that changes in one part of the system do not inadvertently impact others.

5.2 Separation of Concerns

The design follows the principle of separation of concerns, where each part of the system has a distinct responsibility. For instance, the factories are responsible for creating models, while the models themselves encapsulate the specific optimization logic. This separation helps in isolating and addressing issues, testing components independently, and ensuring that each part of the system can evolve without causing disruptions.

5.3 Reusability

Reusability is emphasized through the use of common abstract interfaces and the implementation of general functions that can operate on any subtype. For example, the `optimize_model` function can be used with any model that conforms to the `AbstractMPOPFModel` interface.

6 System Design Diagram

— Design Diagram Goes Here —