MARGO JuMP arrays test

```
import ClimateMARGO
```

```
using ClimateMARGO.Models
```

```
using ClimateMARGO.Optimization
```

```
using ClimateMARGO.Diagnostics
```

```
model_parameters =
  ClimateModelParameters("default", Domain(5.0, 2020.0, 2020.0, 2200.0), Economic
```

```
time = 2020.0:5.0:2200.0
```

```
time = let
    d = model_parameters.domain
    d.initial_year:d.dt:d.final_year
end
```

```
Enter cell code...
```

```
max_slope_M = 0.02
```

```
max_slope_M = .02
```

Simple forward model function

To keep things simple, we wrap MARGO's forward model in a number of functions with:

- input: `Vector{Real}`
- output: `Real` or `Vector{Real}`

```
temperatures_controlled (generic function with 1 method)
```

```
function temperatures_controlled(M::Vector{<:Real})::Vector{<:Real}
    model = ClimateModel(model_parameters)

    model.controls.mitigate = M
    T(model; M=true, R=true, G=true)
end
```

```
sample_M =
 Float64[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0
```

```
 Float64[1.15472, 1.2157, 1.28262, 1.35512, 1.43286, 1.51547, 1.6026, 1.69388,
```

```
temperatures_controlled(sample_M)
```

```
final_temperature_controlled (generic function with 1 method)
```

```
function final_temperature_controlled(M::Vector{<:Real})::Real
    temperatures_controlled(M)[end]
end
```

```
3.3993935573899714
```

```
final_temperature_controlled(sample_M)
```

```
control_costs (generic function with 1 method)
```

```
function control_costs(M::Array{<:Real,1})::Real
    model = ClimateModel(model_parameters)

    model.controls.mitigate = M
    costs = cost(model; M=true, discounting=true)

    sum(costs .* model.domain.dt)
end
```

```
39.31668762833109
```

```
control_costs(sample_M)
```

Let's optimize!

```
using JuMP
```

```
import Ipopt
```

```
setup_opt_model (generic function with 1 method)
```

```
setup_opt_model() = Model(optimizer_with_attributes(Ipopt.Optimizer,
    "acceptable_tol" => 1.e-8, "max_iter" => Int64(1e8),
    "acceptable_constr_viol_tol" => 1.e-3, "constr_viol_tol" => 1.e-4,
    "print_frequency_iter" => 50,  "print_timing_statistics" => "no",
    "print_level" => 0,
))
```

Wrapping functions

We can have "vectors" in JuMP, but really, they are a list of scalar variables, with handy notation. It is not a vector in the sense of `Array` .

You cannot call a JuMP-registered Julia function with a JuMP vector, but you can call a function that takes a long list of arguments. So if we want to register a function that takes an array as argument, we have to write a wrapper function. **This trick is described in the JuMP docs**

```
temperatures_controlled_jump (generic function with 1 method)
```

```
temperatures_controlled_jump(M...) = temperatures_controlled(collect(M))
```

```
final_temperature_controlled_jump (generic function with 1 method)
```

```
final_temperature_controlled_jump(M...) =
final_temperature_controlled(collect(M))
```

```
control_costs_jump (generic function with 1 method)
```

```
control_costs_jump(M...) = control_costs(collect(M))
```

```
T_max = 2.5
```

```
T_max = 2.5
```

```
begin
    model_optimizer = setup_opt_model()

    local m = model_optimizer
    local N = length(time)

    M = @variable(model_optimizer, 0.0 <= M[1:N] <= 1.0)


    # Register our wrapper functions
    ###

    register(m,
        :final_temperature_controlled_jump,
        N,
        final_temperature_controlled_jump,
        autodiff=true
    )
    register(m,
        :control_costs_jump,
        N,
        control_costs_jump,
```

```julia
            autodiff=true
        )
        # register(m,
        #   :temperatures_controlled_jump,
        #   N,
        #   temperatures_controlled_jump,
        #   autodiff=true
        # )


        # Temperature constraint
        ###

        temp_constraints = @NLconstraint(m,
            final_temperature_controlled_jump(M...) <= T_max)



        # Slope constraint
        ###

        max_difference_M = max_slope_M * step(time)

        dM = @variable(m,
            -max_difference_M <= dM[1:N-1] <= max_difference_M
        )
        diff_con = @constraint(m, diff_con[i = 1:N-1],
            dM[i] == (M[i+1] - M[i])
        )

        # Initial value constraint
        ###

        init_con = @constraint(m, init_con,
            M[1] == 0.0
        )


        # Objective
        ###

        min_objective = @NLobjective(
            m, Min,
            control_costs_jump(M...)
        )

        model_optimizer
    end;
```

Run the optimization

```
model_optimized =

\text{$$ \begin{alignat*}{1}\min\quad & control_costs_jump(M_{1}, M_{2}, M_{3}, M
```

```
model_optimized = let
    optimize!(model_optimizer)
    model_optimizer
end
```

```
LOCALLY_SOLVED::TerminationStatusCode = 4
```

```
termination_status(model_optimized)
```

```
70.82786507385671
```

```
objective_value(model_optimized)
```

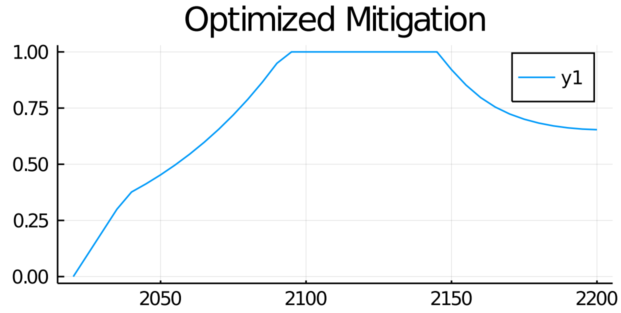
```
M_optimized =
 Float64[4.93536e-45,  0.1,  0.2,  0.3,  0.376304,  0.412795,  0.452837,  0.496775,  0.
```

```
M_optimized = let
    model_optimized
    value.(M)
end
```

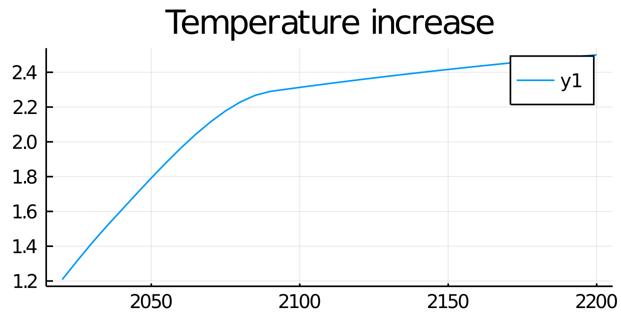
Result

```
using Plots
```

```
using PlutoUI
```



```
plot(time, M_optimized,
    title="Optimized Mitigation",
    dpi=300, size=(400,200))
```



```
plot(time, temperatures_controlled(M_optimized),
    title="Temperature increase",
    dpi=300, size=(400,200))
```

Conclusion

I was able to run some MARGO functions directly inside JuMP:

- The total control costs
- The final temperature

These are both functions that take the `M` array as input, and return a scalar. I had to make one modification to ClimateMARGO.jl: the type of the control vectors changed from `Vector{Float64}` to `Vector{<:Real}`. This is necessary because JuMP uses forward mode automatic diff: it runs your function with dual numbers instead of floats. See the diff **here** (don't merge this yet).

Using these two I was able to: *minimize* control costs *subject to* `temp[2200] <= T_max` (i.e. overshoot allowed).

I was not able to write the global temperature constraint, without calculating the entire temperature series once for each variable M. To my knowledge, it is not possible have this NLconstraint:

```
f(my_vector...) <= my_scalar
```

because you can only give scalar equations & constraints to JuMP. If you write a 'vector constraint' in JuMP, it is really just a pointwise scalar constraint, and this is not the case with our 'black box' Vector->Vector function.

4 vectors instead of 1

The unwrapping trick can also be used to take the M, R, G, A arrays as inputs:

```
small_N = 2
```

```
small_N = 2
```

```
f (generic function with 1 method)
```

```
f(M, R, G, A) = M .+ R .+ G .+ A
```

```
f_wrapped (generic function with 1 method)
```

```
function f_wrapped(MRGA...)
    M = collect(MRGA[              1 : 1 * small_N])
    R = collect(MRGA[    small_N + 1 : 2 * small_N])
    G = collect(MRGA[2 * small_N + 1 : 3 * small_N])
    A = collect(MRGA[3 * small_N + 1 : end])
    f(M, R, G, A)
end
```

```
Int64[10, 0]
```

```
f_wrapped(1, 0, 2, 0, 3, 0, 4, 0)
```

```
Int64[10, 0]
```

```
let
    # in jump it would look a bit like:
    M = [1, 0]
    R = [2, 0]
    G = [3, 0]
    A = [4, 0]
    f_wrapped(M..., R..., G..., A...)
end
```