ExaModels

Sungho Shin

August 15, 2023

Contents

Part I

Introduction

Chapter 1

Overview

Welcome to the documentation of ExaModels.jl

Note

This documentation is also available in PDF format: ExaModels.pdf.

Warning

This documentation page is currently under construction. Please help us improve ExaModels.jl and this documentation! ExaModels.jl is in the early stage of development, and you may encounter unintended behaviors or missing documentations. If you find anything is not working as intended or documentation is missing, please open issues or pull requests or start discussions.

1.1 What is ExaModels.jl?

ExaModels.jl is an algebraic modeling and automatic differentiation tool in Julia Language, specialized for SIMD abstraction of nonlinear programs. ExaModels.jl employs what we call SIMD abstraction for nonlinear programs (NLPs), which allows for the preservation of the parallelizable structure within the model equations, facilitating efficient automatic differentiation either on the single-thread CPUs, multi-threaded CPUs, as well as GPU accelerators. More details about SIMD abstraction can be found here.

1.2 When should I use ExaModels.jl?

ExaModels.jl shines when your model has

- nonlinear objective and constraints;
- a large number of variables and constraints;
- highly repetitive structure;
- sparse Hessian and Jacobian.

These features are often exhibited in optimization problems associated with first-principle physics-based models. Primary examples include optimal control problems formulated with direct subscription method [1] and network system optimization problems, such as optimal power flow [2] and gas network control/estimation problems.

1.3 New to Julia?

Welcome to Julia community! Below is the helpful link to start using Julia.

- juliaup: julia installer and version multiplexer.
- Julia tutorial in Julia's official documentation.

1.4 Supported Solvers

ExaModels can be used with any solver that can handle `NLPModel` data type, but several callbacks are not currently implemented, and cause some errors. Currently, it is tested with the following solvers:

- Ipopt (via NLPModelsIpopt.jl)
- MadNLP.jl

1.5 Documentation Structure

This documentation is structured in the following way.

- The remainder of this page highlights several key aspects of ExaModels.jl.
- The mathematical abstraction—SIMD abstraction of nonlinear programming—of ExaModels.jl is discussed in Mathematical Abstraction page.
- The step-by-step tutorial of using ExaModels.jl can be found in Tutorial page.
- This documentation does not intend to discuss the engineering behind the implementation of ExaModels.jl. Some high-level idea is discussed in a recent publication, but the full details of the engineering behind it will be discussed in the future publications.

1.6 Citing ExaModels.jl

If you use ExaModels.jl in your research, we would greatly appreciate your citing this preprint.

```
@misc{shin2023accelerating,
      title={Accelerating Optimal Power Flow with {GPU}s: {SIMD} Abstraction of Nonlinear Programs
     and Condensed-Space Interior-Point Methods},
      author={Sungho Shin and Fran{\c{c}}ois Pacaud and Mihai Anitescu},
      year={2023},
      eprint={2307.16830},
      archivePrefix={arXiv},
      primaryClass={math.OC}
}
```

1.7 Supporting ExaModels.jl

- Please report issues and feature requests via the GitHub issue tracker.
- Questions are welcome at GitHub discussion forum.

Chapter 2

Highlights

2.1 Key differences from other algebraic modeling tools

ExaModels.jl is different from other algebraic modeling tools, such as JuMP or AMPL, in the following ways:

- **Modeling Interface**: ExaModels.jl enforces users to specify the model equations always in the form of `Generator`. This allows ExaModels.jl to preserve the SIMD-compatible structure in the model equations.
- **Performance**: ExaModels.jl compiles (via Julia's compiler) derivative evaluation codes that are specific to each computation pattern, based on reverse-mode automatic differentiation. This makes the speed of derivative evaluation (even on the CPU) significantly faster than other existing tools.
- **Portability**: ExaModels.jl can evaluate derivatives on GPU accelerators. The code is currently only tested for NVIDIA GPUs, but GPU code is implemented mostly based on the portable programming paradigm, KernelAbstractions.jl. In the future, we are interested in supporting Intel, AMD, and Apple GPUs.

2.2 Performance

For the nonlinear optimization problems that are suitable for SIMD abstraction, ExaModels.jl greatly accelerate the performance of derivative evaluations. The following is a recent benchmark result. Remarkably, for the AC OPF problem for a 9241 bus system, derivative evalution using ExaModels.jl on GPUs can be up to 2 orders of magnitudes faster than JuMP or AMPL.

```
==============================================================
|                  Evaluation Wall Time (sec)                |
==============================================================
|      |           |            ExaModels (single)           |
| case | nvar ncon |   obj     con    grad     jac    hess   |
==============================================================
|  LV1 | 100   98  | 6.3e-06 7.6e-06 6.3e-06 1.2e-05 9.3e-05 |
|  LV2 |   1k 998  | 2.8e-05 3.0e-05 2.4e-05 5.5e-05 6.7e-04 |
|  LV3 |  10k  10k | 2.8e-04 2.9e-04 2.3e-04 5.4e-04 2.5e-03 |
|  QR1 | 659  459  | 8.9e-06 1.0e-05 6.2e-06 1.9e-05 3.1e-05 |
|  QR2 |   7k   5k | 5.7e-05 5.1e-05 4.1e-05 1.0e-04 2.1e-04 |
|  QR3 |  65k  45k | 5.9e-04 5.4e-04 4.6e-04 1.2e-03 6.4e-03 |
|  DC1 | 402  396  | 1.4e-06 4.9e-06 2.6e-06 6.4e-06 3.5e-05 |
|  DC2 |   3k   3k | 2.0e-06 2.6e-05 6.2e-06 5.4e-05 9.7e-05 |
|  DC3 |  34k  33k | 1.1e-05 2.4e-04 6.3e-05 5.3e-04 2.0e-03 |
|  PF1 |   1k   2k | 2.0e-06 3.5e-05 2.3e-06 3.7e-05 3.4e-04 |
```

```
|  PF2 |  11k  17k | 5.3e-06 3.1e-04 9.7e-06 3.2e-04 3.6e-03 |
|  PF3 |  86k 131k | 1.9e-05 2.8e-03 1.2e-04 2.7e-03 2.0e-02 |
==============================================================
|      |           |           ExaModels (multli)            |
| case | nvar ncon |   obj     con    grad     jac    hess   |
==============================================================
|  LV1 | 100   98  | 1.3e-05 1.5e-05 1.7e-05 1.6e-05 1.0e-04 |
|  LV2 |   1k 998  | 4.3e-05 4.1e-05 4.8e-05 7.5e-05 2.4e-04 |
|  LV3 |  10k  10k | 2.5e-04 2.2e-04 3.6e-04 5.1e-04 2.7e-03 |
|  QR1 | 659  459  | 2.2e-05 4.8e-05 2.4e-05 5.8e-05 8.4e-05 |
|  QR2 |   7k   5k | 2.5e-04 9.7e-05 2.6e-04 1.4e-04 6.5e-04 |
|  QR3 |  65k  45k | 5.0e-04 1.2e-03 7.7e-04 2.0e-03 6.9e-03 |
|  DC1 | 402  396  | 1.0e-05 3.7e-05 1.3e-05 3.9e-05 1.2e-04 |
|  DC2 |   3k   3k | 9.0e-06 1.8e-04 2.0e-05 2.0e-04 2.7e-04 |
|  DC3 |  34k  33k | 2.3e-05 4.3e-04 8.6e-05 8.2e-04 2.6e-03 |
|  PF1 |   1k   2k | 7.1e-06 7.8e-05 9.7e-06 8.7e-05 4.0e-04 |
|  PF2 |  11k  17k | 8.7e-06 1.5e-03 1.8e-05 1.4e-03 7.6e-03 |
|  PF3 |  86k 131k | 1.3e-04 1.9e-03 2.2e-04 2.3e-03 9.2e-03 |
==============================================================
|      |           |           ExaModels (gpu)               |
| case | nvar ncon |   obj     con    grad     jac    hess   |
==============================================================
|  LV1 | 100   98  | 8.3e-05 4.8e-05 8.9e-05 5.3e-05 1.1e-04 |
|  LV2 |   1k 998  | 5.1e-05 2.6e-05 5.4e-05 3.7e-05 6.6e-05 |
|  LV3 |  10k  10k | 6.4e-05 2.8e-05 5.7e-05 3.5e-05 6.8e-05 |
|  QR1 | 659  459  | 1.1e-04 2.2e-04 1.0e-04 2.3e-04 3.2e-04 |
|  QR2 |   7k   5k | 9.1e-05 1.5e-04 8.3e-05 1.6e-04 2.3e-04 |
|  QR3 |  65k  45k | 1.0e-04 1.7e-04 9.0e-05 1.8e-04 2.6e-04 |
|  DC1 | 402  396  | 8.3e-05 2.0e-04 7.9e-05 2.0e-04 2.6e-04 |
|  DC2 |   3k   3k | 6.6e-05 1.6e-04 6.7e-05 1.6e-04 2.1e-04 |
|  DC3 |  34k  33k | 7.3e-05 1.6e-04 7.0e-05 1.7e-04 2.5e-04 |
|  PF1 |   1k   2k | 5.7e-05 3.4e-04 5.2e-05 2.8e-04 3.5e-04 |
|  PF2 |  11k  17k | 5.6e-05 3.2e-04 5.6e-05 3.1e-04 3.1e-04 |
|  PF3 |  86k 131k | 9.9e-05 3.6e-04 5.0e-05 2.9e-04 3.8e-04 |
==============================================================
|      |           |                  JuMP                   |
| case | nvar ncon |   obj     con    grad     jac    hess   |
==============================================================
|  LV1 | 100   98  | 5.5e-06 2.8e-05 8.6e-06 3.1e-05 3.5e-04 |
|  LV2 |   1k 998  | 4.6e-05 2.9e-04 1.1e-04 4.9e-04 2.9e-03 |
|  LV3 |  10k  10k | 9.1e-04 4.8e-03 2.1e-03 7.0e-03 2.5e-02 |
|  QR1 | 659  459  | 9.2e-06 3.1e-05 8.3e-06 5.1e-05 1.0e-04 |
|  QR2 |   7k   5k | 1.2e-04 3.7e-04 1.1e-04 7.0e-04 3.8e-03 |
|  QR3 |  65k  45k | 2.1e-03 8.2e-03 2.1e-03 1.6e-02 4.0e-02 |
|  DC1 | 402  396  | 2.5e-06 2.0e-05 4.1e-06 3.9e-05 3.5e-04 |
|  DC2 |   3k   3k | 2.7e-05 2.9e-04 6.0e-05 6.0e-04 1.2e-03 |
|  DC3 |  34k  33k | 4.8e-04 7.7e-03 1.9e-03 1.5e-02 4.2e-02 |
|  PF1 |   1k   2k | 3.5e-06 1.1e-04 4.0e-06 2.3e-04 1.9e-03 |
|  PF2 |  11k  17k | 3.8e-05 2.1e-03 5.0e-05 4.4e-03 1.7e-02 |
|  PF3 |  86k 131k | 6.9e-04 3.5e-02 1.1e-03 7.2e-02 1.1e-01 |
==============================================================
|      |           |                  AMPL                   |
| case | nvar ncon |   obj     con    grad     jac    hess   |
==============================================================
|  LV1 | 100   98  | 1.2e-06 1.7e-06 9.1e-06 1.3e-05 2.1e-04 |
|  LV2 |   1k 998  | 1.5e-06 8.5e-06 2.6e-05 1.7e-04 1.6e-03 |
```

```
|  LV3 |  10k  10k | 8.9e-06 2.3e-04 3.1e-04 3.4e-03 2.2e-02 |
|  QR1 | 659  459  | 1.5e-06 3.4e-06 1.4e-06 2.7e-05 2.9e-04 |
|  QR2 |   7k   5k | 1.2e-05 4.5e-05 1.2e-05 3.5e-04 4.4e-03 |
|  QR3 |  65k  45k | 1.4e-04 1.1e-03 1.2e-04 7.9e-03 5.4e-02 |
|  DC1 | 402  396  | 5.3e-07 4.8e-06 1.2e-06 5.2e-06 3.5e-05 |
|  DC2 |   3k   3k | 8.6e-07 3.9e-05 5.5e-06 6.2e-05 2.6e-05 |
|  DC3 |  34k  33k | 4.8e-06 4.1e-04 2.5e-05 4.1e-04 3.3e-03 |
|  PF1 |   1k   2k | 9.5e-07 2.4e-04 2.8e-06 1.3e-04 1.6e-03 |
|  PF2 |  11k  17k | 1.1e-06 5.1e-03 7.7e-06 3.5e-03 2.6e-02 |
|  PF3 |  86k 131k | 3.5e-06 4.1e-02 5.4e-05 3.6e-02 2.3e-01 |
==============================================================
 * commit : 8a396718b7f7632d239e9edb18f6177fedf4e2a0
 * CPU    : Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz (nthreads = 20)
 * GPU    : Quadro GV100
```

Part II

Mathematical Abstraction

Chapter 3

SIMD Abstraction

In this page, we explain what SIMD abstraction of nonlinear program is, and why it can be beneficial for scalable optimization of large-scale optimization problems. More discussion can be found in our paper.

3.1 What is SIMD abstraction?

The mathematical statement of the problem formulation is as follows.

$$\begin{aligned}
\min_{x^\flat \leq x \leq x^\sharp} & \sum_{l\in[L]} \sum_{i\in[I_l]} f^{(l)}(x; p^{(l)}_i) \\
\text{s.t.} & \left[g^{(m)}(x; q_j)\right]_{j\in[J_m]} + \sum_{n\in[N_m]} \sum_{k\in[K_n]} h^{(n)}(x; s^{(n)}_k) = 0, \quad \forall m \in [M]
\end{aligned}$$

where $f^{(\ell)}(\cdot,\cdot)$, $g^{(m)}(\cdot,\cdot)$, and $h^{(n)}(\cdot,\cdot)$ are twice differentiable functions with respect to the first argument, whereas $\{\{p^{(k)}_i\}_{i\in[N_k]}\}_{k\in[K]}$, $\{\{q^{(k)}_i\}_{i\in[M_l]}\}_{m\in[M]}$, and $\{\{\{s^{(n)}_k\}_{k\in[K_n]}\}_{n\in[N_m]}\}_{m\in[M]}$ are problem data, which can either be discrete or continuous. It is also assumed that our functions $f^{(l)}(\cdot,\cdot)$, $g^{(m)}(\cdot,\cdot)$, and $h^{(n)}(\cdot,\cdot)$ can be expressed with computational graphs of moderate length.

3.2 Why SIMD abstraction?

Many physics-based models, such as AC OPF, have a highly repetitive structure. One of the manifestations of it is that the mathematical statement of the model is concise, even if the practical model may contain millions of variables and constraints. This is possible due to the use of repetition over a certain index and data sets. For example, it suffices to use 15 computational patterns to fully specify the AC OPF model. These patterns arise from (1) generation cost, (2) reference bus voltage angle constraint, (3-6) active and reactive power flow (from and to), (7) voltage angle difference constraint, (8-9) apparent power flow limits (from and to), (10-11) power balance equations, (12-13) generators' contributions to the power balance equations, and (14-15) in/out flows contributions to the power balance equations. However, such repetitive structure is not well exploited in the standard NLP modeling paradigms. In fact, without the SIMD abstraction, it is difficult for the AD package to detect the parallelizable structure within the model, as it will require the full inspection of the computational graph over all expressions. By preserving the repetitive structures in the model, the repetitive structure can be directly available in AD implementation.

Using the multiple dispatch feature of Julia, ExaModels.jl generates highly efficient derivative computation code, specifically compiled for each computational pattern in the model. These derivative evaluation codes can be run over the data in various GPU array formats, and implemented via array and kernel programming in

Julia Language. In turn, ExaModels.jl has the capability to efficiently evaluate first and second-order derivatives using GPU accelerators.

Part III

Tutorial

Chapter 4

Getting Started

ExaModels can create nonlinear prgogramming models and allows solving the created models using NLP solvers (in particular, those that are interfaced with `NLPModels`, such as NLPModelsIpopt and MadNLP. This documentation page will describe how to use `ExaModels` to model and solve nonlinear optimization problems.

We will first consider the following simple nonlinear program [3]:

$$
\begin{aligned}
\min_{\{x_i\}_{i=0}^N} & \sum_{i=2}^N 100(x_{i-1}^2 - x_i)^2 + (x_{i-1} - 1)^2 \\
\text{s.t.} & 3x_{i+1}^3 + 2x_{i+2} - 5 + \sin(x_{i+1} - x_{i+2}) \sin(x_{i+1} + x_{i+2}) + 4x_{i+1} - x_i e^{x_i - x_{i+1}} - 3 = 0
\end{aligned}
$$

We will follow the following Steps to create the model/solve this optimization problem.

- Step 0: import ExaModels.jl
- Step 1: create a `ExaCore` object, wherein we can progressively build an optimization model.
- Step 2: create optimization variables with `variable`, while attaching it to previously created `ExaCore`.
- Step 3 (interchangable with Step 3): create objective function with `objective`, while attaching it to previously created `ExaCore`.
- Step 4 (interchangable with Step 2): create constraints with `constraint`, while attaching it to previously created `ExaCore`.
- Step 5: create an `ExaModel` based on the `ExaCore`.

Now, let's jump right in. We import ExaModels via (Step 0):

```
using ExaModels
```

Now, all the functions that are necessary for creating model are imported to into `Main`.

An `ExaCore` object can be created simply by (Step 1):

```
c = ExaCore()
```

```
An ExaCore

  Float type: ...................... Float64
  Array type: ...................... Vector{Float64}
  Backend: ......................... Nothing

  number of objective patterns: .... 0
  number of constraint patterns: ... 0
```

This is where our optimziation model information will be progressively stored. This object is not yet an `NLPModel`, but it will essentially store all the necessary information.

Now, let's create the optimziation variables. From the problem definition, we can see that we will need N scalar variables. We will choose $N = 10$, and create the variable $x \in \mathbb{R}^N$ with the follwoing command:

```
N = 10
x = variable(c, N; start = (mod(i, 2) == 1 ? -1.2 : 1.0 for i = 1:N))
```

```
Variable

  x ∈ R^{10}
```

This creates the variable x, which we will be able to refer to when we create constraints/objective constraionts. Also, this modifies the information in the `ExaCore` object properly so that later an optimization model can be properly created with the necessary information. Observe that we have used the keyword argument `start` to specify the initial guess for the solution. The variable upper and lower bounds can be specified in a similar manner.

The objective can be set as follows:

```
objective(c, 100 * (x[i-1]^2 - x[i])^2 + (x[i-1] - 1)^2 for i = 2:N)
```

```
Objective

  min (...) + ∑_{p ∈ P} f(x,p)

  where |P| = 9
```

The constraints can be set as follows:

```
constraint(
    c,
    3x[i+1]^3 + 2 * x[i+2] - 5 + sin(x[i+1] - x[i+2])sin(x[i+1] + x[i+2]) + 4x[i+1] -
    x[i]exp(x[i] - x[i+1]) - 3 for i = 1:N-2
)
```

```
Constraint

  s.t. (...)
       ♭g ≤ [g(x,p)]_{p ∈ P} ≤ ♯g

  where |P| = 8
```

Finally, we are ready to create an `ExaModel` from the data we have collected in `ExaCore`. Since `ExaCore` includes all the necessary information, we can do this simply by:

```
m = ExaModel(c)
```

```
An ExaModel

  Problem name: Generic
   All variables: ████████████████████ 10      All constraints: ████████████████████ 8
            free: ████████████████████ 10                 free: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           lower: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                 lower: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           upper: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                 upper: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
         low/upp: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0               low/upp: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           fixed: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                 fixed: ████████████████████ 8
          infeas: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                infeas: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
            nnzh: (-36.36% sparsity)   75               linear: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
                                                     nonlinear: ████████████████████ 8
                                                          nnzj: ( 70.00% sparsity)   24
```

Now, we got an optimization model ready to be solved. This problem can be solved with for example, with the Ipopt solver, as follows.

```julia
using NLPModelsIpopt
result = ipopt(m)
```

```
"Execution stats: first-order stationary"
```

Here, `result` is an `AbstractExecutionStats`, which typically contains the solution information. We can check several information as follows.

```julia
println("Status: $(result.status)")
println("Number of iterations: $(result.iter)")
```

```
Status: first_order
Number of iterations: 6
```

The solution values for variable x can be inquired by:

```julia
sol = solution(result, x)
```

```
10-element view(::Vector{Float64}, 1:10) with eltype Float64:
 -0.9505563573613093
  0.9139008176388945
  0.9890905176644905
  0.9985592422681151
  0.9998087408802769
  0.9999745932450963
  0.9999966246997642
  0.9999995512524277
  0.999999944919307
  0.999999930070643
```

ExaModels provide several APIs similar to this:

- `solution` inquires the primal solution.
- `multiplier` inquires the dual solution.
- `multiplier_L` inquires the lower bound dual solution.
- `multiplier_U` inquires the upper bound dual solution.

This concludes a short tutorial on how to use ExaModels to model and solve optimization problems. Want to learn more? Take a look at the following examples, which provide further tutorial on how to use ExaModels.jl. Each of the examples are designed to instruct a few additional techniques.

- Example: Quadrotor: modeling multiple types of objective values and constraints.
- Example: Distillation Column: using two-dimensional index sets for variables.
- Example: Optimal Power Flow: handling complex data and using constraint augmentation.

This page was generated using Literate.jl.

Chapter 5

Performance Tips

5.1 Use a function to create a model

It is always better to use functions to create ExaModels. This in this way, the functions used for specifing objective/constraint functions are not recreated over all over, and thus, we can take advantage of the previously compiled model creation code. Let's consider the following example.

```
using ExaModels

t = @elapsed begin
    c = ExaCore()
    N = 10
    x = variable(c, N; start = (mod(i, 2) == 1 ? -1.2 : 1.0 for i = 1:N))
    objective(c, 100 * (x[i-1]^2 - x[i])^2 + (x[i-1] - 1)^2 for i = 2:N)
    constraint(
        c,
        3x[i+1]^3 + 2 * x[i+2] - 5 + sin(x[i+1] - x[i+2])sin(x[i+1] + x[i+2]) + 4x[i+1] -
            x[i]exp(x[i] - x[i+1]) - 3 for i = 1:N-2
                )
    m = ExaModel(c)
end

println("$t seconds elapsed")
```

```
0.116776726 seconds elapsed
```

Even at the second call,

```
t = @elapsed begin
    c = ExaCore()
    N = 10
    x = variable(c, N; start = (mod(i, 2) == 1 ? -1.2 : 1.0 for i = 1:N))
    objective(c, 100 * (x[i-1]^2 - x[i])^2 + (x[i-1] - 1)^2 for i = 2:N)
    constraint(
        c,
        3x[i+1]^3 + 2 * x[i+2] - 5 + sin(x[i+1] - x[i+2])sin(x[i+1] + x[i+2]) + 4x[i+1] -
            x[i]exp(x[i] - x[i+1]) - 3 for i = 1:N-2
                )
    m = ExaModel(c)
end
```

```
println("$t seconds elapsed")
```

```
0.117271018 seconds elapsed
```

the model creation time can be slightly reduced but the compilation time is still quite significant.

But instead, if you create a function, we can significantly reduce the model creation time.

```
function luksan_vlcek_model(N)
    c = ExaCore()
    x = variable(c, N; start = (mod(i, 2) == 1 ? -1.2 : 1.0 for i = 1:N))
    objective(c, 100 * (x[i-1]^2 - x[i])^2 + (x[i-1] - 1)^2 for i = 2:N)
    constraint(
        c,
        3x[i+1]^3 + 2 * x[i+2] - 5 + sin(x[i+1] - x[i+2])sin(x[i+1] + x[i+2]) + 4x[i+1] -
            x[i]exp(x[i] - x[i+1]) - 3 for i = 1:N-2
                )
    m = ExaModel(c)
end

t = @elapsed luksan_vlcek_model(N)
println("$t seconds elapsed")
```

```
0.201202923 seconds elapsed
```

```
t = @elapsed luksan_vlcek_model(N)
println("$t seconds elapsed")
```

```
7.052e-5 seconds elapsed
```

So, the model creation time can be essentially nothing. Thus, if you care about the model creation time, always make sure to write a function for creating the model, and do not directly create a model from the REPL.

5.2 Make sure your array's eltype is concrete

In order for ExaModels to run for loops over the array you provided without any overhead caused by type inference, the eltype of the data array should always be a concrete type. Furthermore, this is **required** if you want to run ExaModels on GPU accelerators.

Let's take an example.

```
using ExaModels

N = 1000

function luksan_vlcek_model_concrete(N)
    c = ExaCore()

    arr1 = Array(2:N)
    arr2 = Array(1:N-2)

    x = variable(c, N; start = (mod(i, 2) == 1 ? -1.2 : 1.0 for i = 1:N))
```

```
    objective(c, 100 * (x[i-1]^2 - x[i])^2 + (x[i-1] - 1)^2 for i = arr1)
    constraint(
        c,
        3x[i+1]^3 + 2 * x[i+2] - 5 + sin(x[i+1] - x[i+2])sin(x[i+1] + x[i+2]) + 4x[i+1] -
            x[i]exp(x[i] - x[i+1]) - 3 for i = arr2
                )
    m = ExaModel(c)
end

function luksan_vlcek_model_non_concrete(N)
    c = ExaCore()

    arr1 = Array{Any}(2:N)
    arr2 = Array{Any}(1:N-2)

    x = variable(c, N; start = (mod(i, 2) == 1 ? -1.2 : 1.0 for i = 1:N))
    objective(c, 100 * (x[i-1]^2 - x[i])^2 + (x[i-1] - 1)^2 for i = arr1)
    constraint(
        c,
        3x[i+1]^3 + 2 * x[i+2] - 5 + sin(x[i+1] - x[i+2])sin(x[i+1] + x[i+2]) + 4x[i+1] -
            x[i]exp(x[i] - x[i+1]) - 3 for i = arr2
                )
    m = ExaModel(c)
end
```

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

Here, observe that

```
isconcretetype(eltype(Array(2:N)))
```

```
true
```

```
isconcretetype(eltype(Array{Any}(2:N)))
```

```
false
```

As you can see, the first array type has concrete eltypes, whereas the second array type has non concrete eltypes. Due to this, the array stored in the model created by `luksan_vlcek_model_non_concrete` will have non-concrete eltypes.

Now let's compare the performance. We will use the following benchmark function here.

```
using NLPModels

function benchmark_callbacks(m; N = 100)
    nvar = m.meta.nvar
    ncon = m.meta.ncon
    nnzj = m.meta.nnzj
    nnzh = m.meta.nnzh

    x = copy(m.meta.x0)
    y = similar(m.meta.x0, ncon)
```

```
    c = similar(m.meta.x0, ncon)
    g = similar(m.meta.x0, nvar)
    jac = similar(m.meta.x0, nnzj)
    hess = similar(m.meta.x0, nnzh)
    jrows = similar(m.meta.x0, Int, nnzj)
    jcols = similar(m.meta.x0, Int, nnzj)
    hrows = similar(m.meta.x0, Int, nnzh)
    hcols = similar(m.meta.x0, Int, nnzh)

    GC.enable(false)

    NLPModels.obj(m, x) # to compile

    tobj = (1 / N) * @elapsed for t = 1:N
        NLPModels.obj(m, x)
    end

    NLPModels.cons!(m, x, c) # to compile
    tcon = (1 / N) * @elapsed for t = 1:N
        NLPModels.cons!(m, x, c)
    end

    NLPModels.grad!(m, x, g) # to compile
    tgrad = (1 / N) * @elapsed for t = 1:N
        NLPModels.grad!(m, x, g)
    end

    NLPModels.jac_coord!(m, x, jac) # to compile
    tjac = (1 / N) * @elapsed for t = 1:N
        NLPModels.jac_coord!(m, x, jac)
    end

    NLPModels.hess_coord!(m, x, y, hess) # to compile
    thess = (1 / N) * @elapsed for t = 1:N
        NLPModels.hess_coord!(m, x, y, hess)
    end

    NLPModels.jac_structure!(m, jrows, jcols) # to compile
    tjacs = (1 / N) * @elapsed for t = 1:N
        NLPModels.jac_structure!(m, jrows, jcols)
    end

    NLPModels.hess_structure!(m, hrows, hcols) # to compile
    thesss = (1 / N) * @elapsed for t = 1:N
        NLPModels.hess_structure!(m, hrows, hcols)
    end

    GC.enable(true)

    return (
        tobj = tobj,
        tcon = tcon,
        tgrad = tgrad,
        tjac = tjac,
        thess = thess,
        tjacs = tjacs,
```

```
        thesss = thesss,
    )
end
```

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

The performance comparison is here:

```
m1 = luksan_vlcek_model_concrete(N)
m2 = luksan_vlcek_model_non_concrete(N)

benchmark_callbacks(m1)
```

```
(tobj = 1.572299e-5, tcon = 2.6152030000000002e-5, tgrad = 1.868329e-5, tjac = 4.884695e-5, thess =
     0.00041794916000000004, tjacs = 1.68763e-5, thesss = 1.958946e-5)
```

```
benchmark_callbacks(m2)
```

```
(tobj = 0.00018287400999999998, tcon = 0.00025873918, tgrad = 0.00025151023000000004, tjac =
     0.00144025575, thess = 0.00345549689, tjacs = 0.0005031359, thesss = 0.00203735337)
```

As can be seen here, having concrete eltype dramatically improves the performance. This is because when all the data arrays' eltypes are concrete, the AD evaluations can be performed without any type inferernce, and this should be as fast as highly optimized C/C++/Fortran code.

When you're using GPU accelerators, the eltype of the array should always be concrete. In fact, non-concrete etlype will already cause an error when creating the array. For example,

```
using CUDA

try
    arr1 = CuArray(2:N)
catch e
    showerror(stdout,e)
end
```

```
999-element CUDA.CuArray{Int64, 1, CUDA.Mem.DeviceBuffer}:
    2
    3
    4
    5
    6
    7
    8
    9
   10
   11
   12
   13
   14
   15
   16
   17
```

```
 970
 971
 972
 973
 974
 975
 976
 977
 978
 979
 980
 981
 982
 983
 984
 985
 986
 987
 988
 989
 990
 991
 992
 993
 994
 995
 996
 997
 998
 999
1000
```

This page was generated using Literate.jl.

Chapter 6

Accelerations

One of the key features of ExaModels.jl is being able to evaluate derivatives either on multi-threaded CPUs or GPU accelerators. Currently, GPU acceleration is only tested for NVIDIA GPUs. If you'd like to use multi-threaded CPU acceleration, start julia with

```
$ julia -t 4 # using 4 threads
```

Also, if you're using NVIDIA GPUs, make sure to have installed appropriate drivers.

Let's say that our CPU code is as follows.

```julia
function luksan_vlcek_obj(x,i)
    return 100*(x[i-1]^2-x[i])^2+(x[i-1]-1)^2
end

function luksan_vlcek_con(x,i)
    return 3x[i+1]^3+2*x[i+2]-5+sin(x[i+1]-x[i+2])sin(x[i+1]+x[i+2])+4x[i+1]-x[i]exp(x[i]-x[i+1])-3
end

function luksan_vlcek_x0(i)
    return mod(i,2)==1 ? -1.2 : 1.0
end

function luksan_vlcek_model(N)

    c = ExaCore()
    x = variable(
        c, N;
        start = (luksan_vlcek_x0(i) for i=1:N)
    )
    constraint(
        c,
        luksan_vlcek_con(x,i)
        for i in 1:N-2)
    objective(
        c,
        luksan_vlcek_obj(x,i)
        for i in 2:N)

    return ExaModel(c)
end
```

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

Now we simply modify this by

```
function luksan_vlcek_model(N, backend = nothing)

    c = ExaCore(backend) # specify the backend
    x = variable(
        c, N;
        start = (luksan_vlcek_x0(i) for i=1:N)
    )
    constraint(
        c,
        luksan_vlcek_con(x,i)
        for i in 1:N-2)
    objective(
        c,
        luksan_vlcek_obj(x,i)
        for i in 2:N)

    return ExaModel(c)
end
```

```
luksan_vlcek_model (generic function with 2 methods)
```

The acceleration can be done simply by specifying the backend. In particular, for multi-threaded CPUs,

```
using ExaModels, NLPModelsIpopt, KernelAbstractions

m = luksan_vlcek_model(10, CPU())
ipopt(m)
```

```
"Execution stats: first-order stationary"
```

For NVIDIA GPUs, we can use CUDABackend. However, currently, there are not many optimization solvers that are capable of solving problems on GPUs. The only option right now is using a development branch in MadNLP.jl. To use this, first install

```
import Pkg; Pkg.add("MadNLP"; rev="sparse_condensed_2")
```

Then, we can run:

```
using CUDA, MadNLP, MadNLPGPU

m = luksan_vlcek_model(10, CUDABackend())
madnlp(m)
```

In the case we have arrays for the data, what we need to do is to simply convert the array types to the corresponding device array types. In particular,

```
function cuda_luksan_vlcek_model(N)
    c = ExaCore(CUDABackend())
    d1 = CuArray(1:N-2)
    d2 = CuArray(2:N)
    d3 = CuArray([luksan_vlcek_x0(i) for i=1:N])

    x = variable(
        c, N;
        start = d3
    )
    constraint(
        c,
        luksan_vlcek_con(x,i)
        for i in d1
    )
    objective(
        c,
        luksan_vlcek_obj(x,i)
        for i in d2
    )

    return ExaModel(c)
end
```

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

```
m = cuda_luksan_vlcek_model(10)
madnlp(m)
```

This page was generated using Literate.jl.

Chapter 7

Developing Extensions

ExaModels.jl's API only uses simple julia funcitons, and thus, implementing the extensions is straightforward. Below, we suggest a good practice for implementing an extension package.

Let's say that we want to implement an extension package for the example problem in Getting Started. An extension package may look like:

```
Root├──
Project.toml├──
 src│
   └── LuksanVlcekModels.jl└──
 test
    └── runtest.jl
```

Each of the files containing

```
# Project.toml

name = "LuksanVlcekModels"
uuid = "0c5951a0-f777-487f-ad29-fac2b9a21bf1"
authors = ["Sungho Shin <sshin@anl.gov>"]
version = "0.1.0"

[deps]
ExaModels = "1037b233-b668-4ce9-9b63-f9f681f55dd2"

[extras]
NLPModelsIpopt = "f4238b75-b362-5c4c-b852-0801c9a21d71"
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"

[targets]
test = ["Test", "NLPModelsIpopt"]
```

```julia
# src/LuksanVlcekModels.jl

module LuksanVlcekModels

import ExaModels

function luksan_vlcek_obj(x,i)
    return 100*(x[i-1]^2-x[i])^2+(x[i-1]-1)^2
end
```

```
function luksan_vlcek_con(x,i)
    return 3x[i+1]^3+2*x[i+2]-5+sin(x[i+1]-x[i+2])sin(x[i+1]+x[i+2])+4x[i+1]-x[i]exp(x[i]-x[i+1])-3
end

function luksan_vlcek_x0(i)
    return mod(i,2)==1 ? -1.2 : 1.0
end

function luksan_vlcek_model(N; backend = nothing)

    c = ExaModels.ExaCore(backend)
    x = ExaModels.variable(
        c, N;
        start = (luksan_vlcek_x0(i) for i=1:N)
    )
    ExaModels.constraint(
        c,
        luksan_vlcek_con(x,i)
        for i in 1:N-2)
    ExaModels.objective(c, luksan_vlcek_obj(x,i) for i in 2:N)

    return ExaModels.ExaModel(c) # returns the model
end

export luksan_vlcek_model

end # module LuksanVlcekModels
```

```
# test/runtest.jl

using Test, LuksanVlcekModels, NLPModelsIpopt

@testset "LuksanVlcekModelsTest" begin
    m = luksan_vlcek_model(10)
    result = ipopt(m)

    @test result.status == :first_order
    @test result.solution ≈ [
        -0.9505563573613093
        0.9139008176388945
        0.9890905176644905
        0.9985592422681151
        0.9998087408802769
        0.9999745932450963
        0.9999966246997642
        0.9999995512524277
        0.999999944919307
        0.999999930070643
    ]
    @test result.multipliers ≈ [
        4.1358568305002255
        -1.876494903703342
        -0.06556333356358675
        -0.021931863018312875
        -0.0019537261317119302
```

```
        -0.00032910445671233547
        -3.8788212776372465e-5
        -7.376592164341867e-6
    ]
end
```

Chapter 8

Example: Quadrotor

```
function quadrotor_model(N = 3; backend = nothing)

    n = 9
    p = 4
    nd= 9
    d(i,j,N) = (j==1 ? 1*sin(2*pi/N*i) : 0.) + (j==3 ? 2*sin(4*pi/N*i) : 0.) + (j==5 ? 2*i/N : 0.)
    dt = .01
    R = fill(1/10,4)
    Q = [1,0,1,0,1,0,1,1,1]
    Qf= [1,0,1,0,1,0,1,1,1]/dt

    x0s  = [(i,0.) for i=1:n]
    itr0 = [(i,j,R[j]) for (i,j) in Base.product(1:N,1:p)]
    itr1 = [(i,j,Q[j],d(i,j,N)) for (i,j) in Base.product(1:N,1:n)]
    itr2 = [(j,Qf[j],d(N+1,j,N)) for j in 1:n]

    c = ExaCore(backend)

    x= variable(c,1:N+1,1:n)
    u= variable(c,1:N,1:p);

    constraint(c, x[1,i]-x0 for (i,x0) in x0s)
    constraint(c, -x[i+1,1] + x[i,1] + (x[i,2])*dt for i=1:N)
    constraint(c, -x[i+1,2] + x[i,2] +
    ↪  (u[i,1]*cos(x[i,7])*sin(x[i,8])*cos(x[i,9])+u[i,1]*sin(x[i,7])*sin(x[i,9]))*dt for i=1:N)
    constraint(c, -x[i+1,3] + x[i,3] + (x[i,4])*dt for i=1:N)
    constraint(c, -x[i+1,4] + x[i,4] +
    ↪  (u[i,1]*cos(x[i,7])*sin(x[i,8])*sin(x[i,9])-u[i,1]*sin(x[i,7])*cos(x[i,9]))*dt for i=1:N)
    constraint(c, -x[i+1,5] + x[i,5] + (x[i,6])*dt for i=1:N)
    constraint(c, -x[i+1,6] + x[i,6] + (u[i,1]*cos(x[i,7])*cos(x[i,8])-9.8)*dt for i=1:N)
    constraint(c, -x[i+1,7] + x[i,7] +
    ↪  (u[i,2]*cos(x[i,7])/cos(x[i,8])+u[i,3]*sin(x[i,7])/cos(x[i,8]))*dt for i=1:N)
    constraint(c, -x[i+1,8] + x[i,8] + (-u[i,2]*sin(x[i,7])+u[i,3]*cos(x[i,7]))*dt for i=1:N)
    constraint(c, -x[i+1,9] + x[i,9] +
    ↪  (u[i,2]*cos(x[i,7])*tan(x[i,8])+u[i,3]*sin(x[i,7])*tan(x[i,8])+u[i,4])*dt for i=1:N)

    objective(c, .5*R*(u[i,j]^2) for (i,j,R) in itr0)
    objective(c, .5*Q*(x[i,j]-d)^2 for (i,j,Q,d) in itr1)
    objective(c, .5*Qf*(x[N+1,j]-d)^2 for (j,Qf,d) in itr2)
```

```
    m = ExaModel(c)

end
```

```
quadrotor_model (generic function with 2 methods)
```

```
using ExaModels, NLPModelsIpopt

m = quadrotor_model(100)
result = ipopt(m)
```

```
"Execution stats: first-order stationary"
```

This page was generated using Literate.jl.

Chapter 9

Example: Distillation Column

```
function distillation_column_model(T = 3; backend = nothing)

    NT = 30
    FT = 17
    Ac = 0.5
    At = 0.25
    Ar = 1.0
    D = 0.2
    F = 0.4
    ybar = 0.8958
    ubar = 2.0
    alpha = 1.6
    dt = 10 / T
    xAf = 0.5
    xA0s = ExaModels.convert_array([(i, 0.5) for i = 0:NT+1], backend)

    itr0 = ExaModels.convert_array(collect(Iterators.product(1:T, 1:FT-1)), backend)
    itr1 = ExaModels.convert_array(collect(Iterators.product(1:T, FT+1:NT)), backend)
    itr2 = ExaModels.convert_array(collect(Iterators.product(0:T, 0:NT+1)), backend)

    c = ExaCore(backend)

    xA = variable(c, 0:T, 0:NT+1; start = 0.5)
    yA = variable(c, 0:T, 0:NT+1; start = 0.5)
    u = variable(c, 0:T; start = 1.0)
    V = variable(c, 0:T; start = 1.0)
    L2 = variable(c, 0:T; start = 1.0)

    objective(c, (yA[t, 1] - ybar)^2 for t = 0:T)
    objective(c, (u[t] - ubar)^2 for t = 0:T)

    constraint(c, xA[0, i] - xA0 for (i, xA0) in xA0s)
    constraint(
        c,
        (xA[t, 0] - xA[t-1, 0]) / dt - (1 / Ac) * (yA[t, 1] - xA[t, 0]) for t = 1:T
    )
    constraint(
        c,
        (xA[t, i] - xA[t-1, i]) / dt -
        (1 / At) * (u[t] * D * (yA[t, i-1] - xA[t, i]) - V[t] * (yA[t, i] - yA[t, i+1])) for
```

```
        (t, i) in itr0
    )
    constraint(
        c,
        (xA[t, FT] - xA[t-1, FT]) / dt -
        (1 / At) * (
            F * xAf + u[t] * D * xA[t, FT-1] - L2[t] * xA[t, FT] -
            V[t] * (yA[t, FT] - yA[t, FT+1])
        ) for t = 1:T
    )
    constraint(
        c,
        (xA[t, i] - xA[t-1, i]) / dt -
        (1 / At) * (L2[t] * (yA[t, i-1] - xA[t, i]) - V[t] * (yA[t, i] - yA[t, i+1])) for
        (t, i) in itr1
    )
    constraint(
        c,
        (xA[t, NT+1] - xA[t-1, NT+1]) / dt -
        (1 / Ar) * (L2[t] * xA[t, NT] - (F - D) * xA[t, NT+1] - V[t] * yA[t, NT+1]) for
        t = 1:T
    )
    constraint(c, V[t] - u[t] * D - D for t = 0:T)
    constraint(c, L2[t] - u[t] * D - F for t = 0:T)
    constraint(
        c,
        yA[t, i] * (1 - xA[t, i]) - alpha * xA[t, i] * (1 - yA[t, i]) for (t, i) in itr2
    )

    return ExaModel(c)
end
```

```
distillation_column_model (generic function with 2 methods)
```

```
using ExaModels, NLPModelsIpopt

m = distillation_column_model(10)
ipopt(m)
```

```
"Execution stats: first-order stationary"
```

This page was generated using Literate.jl.

Chapter 10

Example: Optimal Power Flow

```
function parse_ac_power_data(filename)
    data = PowerModels.parse_file(filename)
    PowerModels.standardize_cost_terms!(data, order=2)
    PowerModels.calc_thermal_limits!(data)
    ref = PowerModels.build_ref(data)[:it][:pm][:nw][0]

    arcdict = Dict(
        a=>k
        for (k,a) in enumerate(ref[:arcs]))
    busdict = Dict(
        k=>i
        for (i,(k,v)) in enumerate(ref[:bus]))
    gendict = Dict(
        k=>i
        for (i,(k,v)) in enumerate(ref[:gen]))
    branchdict = Dict(
        k=>i
        for (i,(k,v)) in enumerate(ref[:branch]))

    return (
        bus = [
            begin
                bus_loads = [ref[:load][l] for l in ref[:bus_loads][k]]
                bus_shunts = [ref[:shunt][s] for s in ref[:bus_shunts][k]]
                pd = sum(load["pd"] for load in bus_loads; init = 0.)
                gs = sum(shunt["gs"] for shunt in bus_shunts; init = 0.)
                qd = sum(load["qd"] for load in bus_loads; init = 0.)
                bs = sum(shunt["bs"] for shunt in bus_shunts; init = 0.)
                (
                    i = busdict[k],
                    pd = pd, gs = gs, qd = qd, bs = bs
                )
            end
            for (k,v) in ref[:bus]],
        gen = [
            (i = gendict[k],
             cost1 = v["cost"][1], cost2 = v["cost"][2], cost3 = v["cost"][3], bus =
             ↪  busdict[v["gen_bus"]])
            for (k,v) in ref[:gen]],
        arc =[
```

```
            (i=k, rate_a = ref[:branch][l]["rate_a"], bus = busdict[i])
            for (k,(l,i,j)) in enumerate(ref[:arcs])],
        branch = [
            begin
                f_idx = arcdict[i, branch["f_bus"], branch["t_bus"]]
                t_idx = arcdict[i, branch["t_bus"], branch["f_bus"]]
                g, b = PowerModels.calc_branch_y(branch)
                tr, ti = PowerModels.calc_branch_t(branch)
                ttm = tr^2 + ti^2
                g_fr = branch["g_fr"]
                b_fr = branch["b_fr"]
                g_to = branch["g_to"]
                b_to = branch["b_to"]
                c1 = (-g*tr-b*ti)/ttm
                c2 = (-b*tr+g*ti)/ttm
                c3 = (-g*tr+b*ti)/ttm
                c4 = (-b*tr-g*ti)/ttm
                c5 = (g+g_fr)/ttm
                c6 = (b+b_fr)/ttm
                c7 = (g+g_to)
                c8 = (b+b_to)
                (
                    i = branchdict[i],
                    j = 1,
                    f_idx = f_idx,
                    t_idx = t_idx,
                    f_bus = busdict[branch["f_bus"]],
                    t_bus = busdict[branch["t_bus"]],
                    c1 = c1,
                    c2 = c2,
                    c3 = c3,
                    c4 = c4,
                    c5 = c5,
                    c6 = c6,
                    c7 = c7,
                    c8 = c8,
                    rate_a_sq = branch["rate_a"]^2,
                )
            end
            for (i,branch) in ref[:branch]],
        ref_buses = [busdict[i] for (i,k) in ref[:ref_buses]],
        vmax = [
            v["vmax"] for (k,v) in ref[:bus]],
        vmin = [
            v["vmin"] for (k,v) in ref[:bus]],
        pmax = [
            v["pmax"] for (k,v) in ref[:gen]],
        pmin = [
            v["pmin"] for (k,v) in ref[:gen]],
        qmax = [
            v["qmax"] for (k,v) in ref[:gen]],
        qmin = [
            v["qmin"] for (k,v) in ref[:gen]],
        rate_a =[
            ref[:branch][l]["rate_a"]
            for (k,(l,i,j)) in enumerate(ref[:arcs])],
```

```
        angmax = [b["angmax"] for (i,b) in ref[:branch]],
        angmin = [b["angmin"] for (i,b) in ref[:branch]],
    )
end

convert_data(data::N, backend) where {names, N <: NamedTuple{names}} =
↪  NamedTuple{names}(ExaModels.convert_array(d,backend) for d in data)

parse_ac_power_data(filename, backend) = convert_data(parse_ac_power_data(filename), backend)

function ac_power_model(
    filename;
    backend = nothing,
    T = Float64
    )

    data = parse_ac_power_data(filename, backend)

    w = ExaCore(T, backend)

    va = variable(
        w, length(data.bus);
    )

    vm = variable(
        w,
        length(data.bus);
        start = fill!(similar(data.bus,Float64),1.),
        lvar = data.vmin,
        uvar = data.vmax
    )
    pg = variable(
        w,
        length(data.gen);
        lvar = data.pmin,
        uvar = data.pmax
    )

    qg = variable(
        w,
        length(data.gen);
        lvar = data.qmin,
        uvar = data.qmax
    )

    p = variable(
        w,
        length(data.arc);
        lvar = -data.rate_a,
        uvar = data.rate_a
    )

    q = variable(
        w,
        length(data.arc);
        lvar = -data.rate_a,
```

```
        uvar = data.rate_a
    )

    o = objective(
        w,
        g.cost1 * pg[g.i]^2 + g.cost2 * pg[g.i] + g.cost3
        for g in data.gen)

    c1 = constraint(
        w,
        va[i] for i in data.ref_buses)

    c2 = constraint(
        w,
        p[b.f_idx]
        - b.c5*vm[b.f_bus]^2
        - b.c3*(vm[b.f_bus]*vm[b.t_bus]*cos(va[b.f_bus]-va[b.t_bus]))
        - b.c4*(vm[b.f_bus]*vm[b.t_bus]*sin(va[b.f_bus]-va[b.t_bus]))
        for b in data.branch)

    c3 = constraint(
        w,
        q[b.f_idx]
        + b.c6*vm[b.f_bus]^2
        + b.c4*(vm[b.f_bus]*vm[b.t_bus]*cos(va[b.f_bus]-va[b.t_bus]))
        - b.c3*(vm[b.f_bus]*vm[b.t_bus]*sin(va[b.f_bus]-va[b.t_bus]))
        for b in data.branch)

    c4 = constraint(
        w,
        p[b.t_idx]
        - b.c7*vm[b.t_bus]^2
        - b.c1*(vm[b.t_bus]*vm[b.f_bus]*cos(va[b.t_bus]-va[b.f_bus]))
        - b.c2*(vm[b.t_bus]*vm[b.f_bus]*sin(va[b.t_bus]-va[b.f_bus]))
        for b in data.branch)

    c5 = constraint(
        w,
        q[b.t_idx]
        + b.c8*vm[b.t_bus]^2
        + b.c2*(vm[b.t_bus]*vm[b.f_bus]*cos(va[b.t_bus]-va[b.f_bus]))
        - b.c1*(vm[b.t_bus]*vm[b.f_bus]*sin(va[b.t_bus]-va[b.f_bus]))
        for b in data.branch)

    c6 = constraint(
        w,
        va[b.f_bus] - va[b.t_bus] for b in data.branch;
            lcon = data.angmin,
            ucon = data.angmax
            )
    c7 = constraint(
        w,
        p[b.f_idx]^2 + q[b.f_idx]^2 - b.rate_a_sq for b in data.branch;
            lcon = fill!(similar(data.branch,Float64,length(data.branch)),-Inf)
            )
    c8 = constraint(
```

```
        w,
        p[b.t_idx]^2 + q[b.t_idx]^2 - b.rate_a_sq for b in data.branch;
            lcon = fill!(similar(data.branch,Float64,length(data.branch)),-Inf)
            )

    c9 = constraint(
        w,
        + b.pd
        + b.gs * vm[b.i]^2
        for b in data.bus)

    c10 = constraint(
        w,
        + b.qd
        - b.bs * vm[b.i]^2
        for b in data.bus)

    c11 = constraint!(
        w,
        c9,
        a.bus => p[a.i]
        for a in data.arc)
    c12 = constraint!(
        w,
        c10,
        a.bus => q[a.i]
        for a in data.arc)

    c13 = constraint!(
        w,
        c9,
        g.bus =>-pg[g.i]
        for g in data.gen)
    c14 = constraint!(
        w,
        c10,
        g.bus =>-qg[g.i]
        for g in data.gen)

    return ExaModel(w)

end
```

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

We first download the case file.

```
using Downloads

case = tempname() * ".m"

Downloads.download(
    "https://raw.githubusercontent.com/power-grid-lib/pglib-
↪   opf/dc6be4b2f85ca0e776952ec22cbd4c22396ea5a3/pglib_opf_case3_lmbd.m",
    case
)
```

```
"/tmp/jl_qpi5CNgfAz.m"
```

Then, we can model/sovle the problem.

```
using PowerModels, ExaModels, NLPModelsIpopt

m = ac_power_model(case)
ipopt(m)
```

```
"Execution stats: first-order stationary"
```

This page was generated using Literate.jl.

Part IV

API Manual

Chapter 11

ExaModels

`ExaModels.ExaModels` – Module.

```
ExaModels
```

An algebraic modeling and automatic differentiation tool in Julia Language, specialized for SIMD abstraction of nonlinear programs.

For more information, please visit https://github.com/sshin23/ExaModels.jl

source

`ExaModels.ExaModel` – Method.

```
ExaModel(core)
```

Returns an `ExaModel` object, which can be solved by nonlinear optimization solvers within `JuliaSmoothOptimizer` ecosystem, such as `NLPModelsIpopt` or `MadNLP`.

Example

```
julia> using ExaModels

julia> c = ExaCore();                      # create an ExaCore object

julia> x = variable(c, 1:10);              # create variables

julia> objective(c, x[i]^2 for i in 1:10); # set objective function

julia> m = ExaModel(c)                     # creat an ExaModel object
An ExaModel

  Problem name: Generic
   All variables: ████████████████████ 10      All constraints: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
            free: ████████████████████ 10                 free: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           lower: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                 lower: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           upper: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                 upper: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
         low/upp: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0               low/upp: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           fixed: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                 fixed: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
          infeas: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                infeas: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
            nnzh: ( 81.82% sparsity)   10                linear: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
                                                      nonlinear: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
                                                           nnzj: (------% sparsity)
```

```
julia> using NLPModelsIpopt

julia> result = ipopt(m; print_level=0)    # solve the problem
"Execution stats: first-order stationary"
```

source

ExaModels.constraint – Method.

```
constraint(core, generator; start = 0, lcon = 0,  ucon = 0)
```

Adds constraints specified by a generator to core, and returns an Constraint object.

Keyword Arguments

- start: The initial guess of the solution. Can either be Number, AbstractArray, or Generator.
- lcon : The constraint lower bound. Can either be Number, AbstractArray, or Generator.
- ucon : The constraint upper bound. Can either be Number, AbstractArray, or Generator.

Example

```
julia> using ExaModels

julia> c = ExaCore();

julia> x = variable(c, 10);

julia> constraint(c, x[i] + x[i+1] for i=1:9; lcon = -1, ucon = (1+i for i=1:9))
Constraint

  s.t. (...)
       g♭ ≤ [g(x,p)]_{p ∈ P} ≤ g♯

  where |P| = 9
```

source

ExaModels.multipliers – Method.

```
multipliers(result, y)
```

Returns the multipliers for constraints y associated with result, obtained by solving the model.

Example

```
julia> using ExaModels, NLPModelsIpopt

julia> c = ExaCore();

julia> x = variable(c, 1:10, lvar = -1, uvar = 1);

julia> objective(c, (x[i]-2)^2 for i in 1:10);

julia> y = constraint(c, x[i] + x[i+1] for i=1:9; lcon = -1, ucon = (1+i for i=1:9));
```

```
julia> m = ExaModel(c);

julia> result = ipopt(m; print_level=0);

julia> val = multipliers(result, y);


julia> val[1] ≈ 0.81933930
true
```

source

ExaModels.multipliers_L – Method.

```
multipliers_L(result, x)
```

Returns the multipliers_L for variable x associated with `result`, obtained by solving the model.

Example

```
julia> using ExaModels, NLPModelsIpopt

julia> c = ExaCore();

julia> x = variable(c, 1:10, lvar = -1, uvar = 1);

julia> objective(c, (x[i]-2)^2 for i in 1:10);

julia> m = ExaModel(c);

julia> result = ipopt(m; print_level=0);

julia> val = multipliers_L(result, x);

julia> isapprox(val, fill(0, 10), atol=sqrt(eps(Float64)), rtol=Inf)
true
```

source

ExaModels.multipliers_U – Method.

```
multipliers_U(result, x)
```

Returns the multipliers_U for variable x associated with `result`, obtained by solving the model.

Example

```
julia> using ExaModels, NLPModelsIpopt

julia> c = ExaCore();

julia> x = variable(c, 1:10, lvar = -1, uvar = 1);

julia> objective(c, (x[i]-2)^2 for i in 1:10);

julia> m = ExaModel(c);
```

```
julia> result = ipopt(m; print_level=0);

julia> val = multipliers_U(result, x);

julia> isapprox(val, fill(2, 10), atol=sqrt(eps(Float64)), rtol=Inf)
true
```

source

`ExaModels.objective` – Method.

```
objective(core::ExaCore, generator)
```

Adds objective terms specified by a `generator` to `core`, and returns an `Objective` object.

Example

```
julia> using ExaModels

julia> c = ExaCore();

julia> x = variable(c, 10);

julia> objective(c, x[i]^2 for i=1:10)
Objective

  min (...) + ∑_{p ∈ P} f(x,p)

  where |P| = 10
```

source

`ExaModels.solution` – Method.

```
solution(result, x)
```

Returns the solution for variable x associated with `result`, obtained by solving the model.

Example

```
julia> using ExaModels, NLPModelsIpopt

julia> c = ExaCore();

julia> x = variable(c, 1:10, lvar = -1, uvar = 1);

julia> objective(c, (x[i]-2)^2 for i in 1:10);

julia> m = ExaModel(c);

julia> result = ipopt(m; print_level=0);

julia> val = solution(result, x);

julia> isapprox(val, fill(1, 10), atol=sqrt(eps(Float64)), rtol=Inf)
true
```

source

ExaModels.variable – Method.

```
variable(core, dims...; start = 0, lvar = -Inf, uvar = Inf)
```

Adds variables with dimensions specified by dims to core, and returns Variable object. dims can be either Integer or UnitRange.

Keyword Arguments

- start: The initial guess of the solution. Can either be Number, AbstractArray, or Generator.
- lvar : The variable lower bound. Can either be Number, AbstractArray, or Generator.
- uvar : The variable upper bound. Can either be Number, AbstractArray, or Generator.

Example

```
julia> using ExaModels

julia> c = ExaCore();

julia> x = variable(c, 10; start = (sin(i) for i=1:10))
Variable

  x ∈ R^{10}

julia> y = variable(c, 2:10, 3:5; lvar = zeros(9,3), uvar = ones(9,3))
Variable

  x ∈ R^{9 × 3}
```

source

Part V

References

Chapter 12

References