ExaModels.jl

Sungho Shin

August 9, 2023

Contents

Part I

Introduction

Chapter 1

Introduction

Welcome to the documentation of ExaModels.jl

Warning

This documentation page is under construction.

Chapter 2

What is ExaModels?

ExaModels.jl implements SIMD abstraction of nonlinear programs and the automatic differentiation of its functions. ExaModels.jl expresses the functions in the form of iterables over statically typed data. This allows highly efficient derivative computations based on reverse-mode automatic differentiation.

Chapter 3

Bug reports and support

Please report issues and feature requests via the Github issue tracker.

Part II

Quick Start

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. We now use `ExaModels` to model the following nonlinear program:

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

We model the problem with:

```
using ExaModels
```

We set

```
N = 10000
```

```
10000
```

First, we create a `ExaModels.Core`.

```
c = ExaCore()
```

```
An ExaCore

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

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

The variables can be created as follows:

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

```
Variable

  x ∈ R^{10000}
```

The objective can be set as follows:

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

```
Objective

  min (...) + ∑_{p ∈ P} f(x,p)

  where |P| = 9999
```

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 in 1:N-2)
```

```
Constraint

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

  where |P| = 9998
```

Finally, we create an NLPModel.

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

```
An ExaModel

  Problem name: Generic
   All variables: ████████████████████ 10000  All constraints: ████████████████████ 9998
            free: ████████████████████ 10000             free: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           lower: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                lower: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           upper: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                upper: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
         low/upp: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0              low/upp: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           fixed: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                fixed: ████████████████████ 9998
          infeas: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0               infeas: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
            nnzh: ( 99.82% sparsity)   89985           linear: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
                                                    nonlinear: ████████████████████ 9998
                                                         nnzj: ( 99.97% sparsity)   29994
```

To solve the problem with `Ipopt`,

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

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

The solution `sol` contains the field `sol.solution` holding the optimized parameters.

This page was generated using Literate.jl.

Part III

API Manual

Chapter 5

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.ExaCore` – Type.

ExaCore([array_type::Type, backend])

Returns an intermediate data object `ExaCore`, which later can be used for creating `ExaModel`

Example

```
julia> using ExaModels

julia> c = ExaCore()
An ExaCore

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

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

julia> c = ExaCore(Float32)
An ExaCore

  Float type: ...................... Float32
  Array type: ...................... Vector{Float32}
  Backend: ......................... Nothing

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

julia> using CUDA
```

```
julia> c = ExaCore(Float32, CUDABackend())
An ExaCore

  Float type: ...................... Float32
  Array type: ...................... CUDA.CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}
  Backend: ......................... CUDA.CUDAKernels.CUDABackend

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

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(y, result)
```

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(y, result);

julia> val[1] ≈ 0.81933930
true
```

source

`ExaModels.multipliers_L` – Method.

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

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(x, result);

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

source

ExaModels.multipliers_U – Method.

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

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(x, result);

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(x, result)
```

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(x, result);

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