

# MadDiff

Sungho Shin

June 4, 2022

# Contents

<b>Contents</b>	<b>ii</b>
<b>I Introduction</b>	<b>1</b>
1 Introduction	2
2 What is MadDiff?	3
3 Bug reports and support	4
<b>II Quick Start</b>	<b>5</b>
4 Getting Started	6
4.1 Automatic Differentiation . . . . .	6
4.2 Nonlinear Programming . . . . .	8
<b>III How it Works</b>	<b>13</b>
5 How it Works	14
<b>IV API Manual</b>	<b>15</b>
6 MadDiffCore	16
6.1 MadDiffCore . . . . .	16
7 MadDiffSpecialFunctions	21
7.1 MadDiffMOI . . . . .	21
8 MadDiffModels	22
8.1 MadDiffModels . . . . .	22
9 MadDiffMOI	32
9.1 MadDiffMOI . . . . .	32

## **Part I**

# **Introduction**

## Chapter 1

# Introduction

Welcome to the documentation of [MadDiff.jl](#)

### **Warning**

This documentation page is under construction.

### **Note**

This documentation is also available in [PDF format](#).

## **Chapter 2**

### **What is MadDiff?**

MadDiff.jl is a simple algebraic modeling/differentiation package. MadDiff.jl constructs first and second derivative functions off-line (i.e., prior to calling the optimization solver) by applying operator overloading-based automatic differentiation on functions. The exact derivative functions can be obtained as results.

## **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

### 4.1 Automatic Differentiation

MadDiff provides a flexible user-interface for evaluating first/second-order derivatives of nonlinear expressions. In the following example, using MadDiff, we will create a function, gradient, and Hessian evaluator of the following function:

$$f(x) = x_1^2 + e^{(x_2^{p_1})/2} + \log(x_2 x_3 + p_2),$$

where  $x$  is the variable vector, and  $p$  is the parameter vector.

We first import MadDiff.

```
| using MadDiff
```

First, we create a Source of Variable's.

```
| x = Variable()
```

```
| x
```

The `Base.getindex!` function is extended so that `x[i]` for any `i` creates an expression for  $x_i$ . For example,

```
| x[2]
```

```
| x[2]
```

We can do a similar thing for Parameter's.

```
| p = Parameter()  
| p[1]
```

```
| p[1]
```

Now, we create the nonlienar expression expression.



```
| expr = x[1]^2 + exp(x[2]^p[1])/2 + log(x[2]*x[3]+p[2])
```

```
| x[1]^2 + exp(x[2]^p[1])/2 + log(x[2]*x[3] + p[2])
```

The function evaluator of the above expression can be created by using `MadDiff.function_evaluator` as follows:

```
| f = Evaluator(expr)
```

```
| Evaluator:
| x[1]^2 + exp(x[2]^p[1])/2 + log(x[2]*x[3] + p[2])
```

Now for a given variable and parameter values, the function can be evaluated as follows.

```
| x0 = [0.,0.5,1.5]
| p0 = [2,0.5]
| f(x0,p0)
```

```
| 0.8651562596580804
```

The gradient evaluator can be created as follows:

```
| y0 = similar(x0)
| g = GradientEvaluator(expr)
| g(y0,x0,p0)
| y0
```

```
| 3-element Vector{Float64}:
|  0.0
|  1.8420127083438709
|  0.4
```

The Hessian evaluator can be created as follows:

```
| z0 = zeros(3,3)
| h = HessianEvaluator(expr)
| h(z0,x0,p0)
| z0
```

```
| 3×3 Matrix{Float64}:
|  2.0  0.0  0.0
|  0.0  0.486038  0.0
|  0.0  0.32  -0.16
```

Note that only lower-triangular entries are evaluated.

The evaluator can be constructed in a sparse format:

```
| sh = SparseHessianEvaluator(expr);
```

The sparse coordinates are:

```
| sh.sparsity
```

```
| 4-element Vector{Tuple{Int64, Int64}}:
| (1, 1)
| (2, 2)
| (3, 2)
| (3, 3)
```

The sparse Hessian can be evaluated as follows:

```
| z1 = zeros(length(sh.sparsity))
| sh(z1,x0,p0)
| z1
```

```
| 4-element Vector{Float64}:
| 2.0
| 0.4860381250316117
| 0.31999999999999995
| -0.16000000000000003
```

## 4.2 Nonlinear Programming

### Built-in API

MadDiff provides a built-in API for creating nonlinear programming models and allows solving the created models using NLP solvers (in particular, those that are interfaced with `NLPModels`, such as `NLPModelsIpopt` and `MadNLP`). We now use MadDiff's built-in API to model the following nonlinear program:

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

We model the problem with:

```
| N = 10000
```

```
| 10000
```

First, we create a `MadDiffModel`.

```
| m = MadDiffModel()
```

```
| MadDiffModel{Float64} (not instantiated).
```

The variables can be created as follows:

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

The objective can be set as follows:

```
objective(m, sum(100(x[i-1]^2-x[i])^2+(x[i-1]-1)^2 for i=2:N));
```

The constraints can be set as follows:

```
for i=1:N-2
    constraint(m,
        ↪ 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 ==
        ↪ 0);
end
```

The important last step is instantiating the model. This step must be taken before calling optimizers.

```
instantiate!(m)
```

```
MadDiffModel{Float64} (instantiated).
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: ( 99.96% sparsity) 19999    linear: ..... 0
                                       nonlinear: ██████████ 9998
                                       nnzj: ( 99.97% sparsity) 29994

Counters:
    obj: ..... 0          grad: ..... 0
    cons: ..... 0          cons_nln: ..... 0
    cons_lin: ..... 0      jac: ..... 0
    jcon: ..... 0          jprod: ..... 0
    jgrad: ..... 0         jtprod: ..... 0
    jac_lin: ..... 0       hess: ..... 0
    jac_nln: ..... 0       jhprod: ..... 0
    jprod_lin: ..... 0
    jprod_nln: ..... 0
    jtprod_lin: ..... 0
    jtprod_nln: ..... 0
    hprod: ..... 0
    jhess: ..... 0
```

To solve the problem with Ipopt,

```
using NLPModelsIpopt
ipopt(m);
```

```

*****
This program contains Ipopt, a library for large-scale nonlinear optimization.
Ipopt is released as open source code under the Eclipse Public License (EPL).
For more information visit https://github.com/coin-or/Ipopt
*****

This is Ipopt version 3.14.4, running with linear solver MUMPS 5.4.1.

Number of nonzeros in equality constraint Jacobian...: 29994
Number of nonzeros in inequality constraint Jacobian.: 0
Number of nonzeros in Lagrangian Hessian.....: 19999

Total number of variables.....: 10000
    variables with only lower bounds: 0
    variables with lower and upper bounds: 0
    variables with only upper bounds: 0
Total number of equality constraints.....: 9998
Total number of inequality constraints.....: 0
    inequality constraints with only lower bounds: 0
    inequality constraints with lower and upper bounds: 0
    inequality constraints with only upper bounds: 0

iter   objective    inf_pr  inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  0  2.5405160e+06  2.48e+01  2.73e+01  -1.0  0.00e+00   -  0.00e+00  0.00e+00  0
  1  1.3512419e+06  1.49e+01  8.27e+01  -1.0  2.20e+00   -  1.00e+00  1.00e+00f  1
  2  1.5156131e+05  4.28e+00  1.36e+02  -1.0  1.43e+00   -  1.00e+00  1.00e+00f  1
  3  6.6755024e+01  3.09e-01  2.18e+01  -1.0  5.63e-01   -  1.00e+00  1.00e+00f  1
  4  6.2338933e+00  1.73e-02  8.47e-01  -1.0  2.10e-01   -  1.00e+00  1.00e+00h  1
  5  6.2324586e+00  1.15e-05  8.16e-04  -1.7  3.35e-03   -  1.00e+00  1.00e+00h  1
  6  6.2324586e+00  8.36e-12  7.97e-10  -5.7  2.00e-06   -  1.00e+00  1.00e+00h  1

Number of Iterations....: 6

                                (scaled)                (unscaled)
Objective.....: 7.8692659500479645e-01  6.2324586324379885e+00
Dual infeasibility.....: 7.9743417331632266e-10  6.3156786526652763e-09
Constraint violation....: 8.3555384833289281e-12  8.3555384833289281e-12
Variable bound violation: 0.0000000000000000e+00  0.0000000000000000e+00
Complementarity.....: 0.0000000000000000e+00  0.0000000000000000e+00
Overall NLP error.....: 7.9743417331632266e-10  6.3156786526652763e-09

Number of objective function evaluations = 7
Number of objective gradient evaluations = 7
Number of equality constraint evaluations = 7
Number of inequality constraint evaluations = 0
Number of equality constraint Jacobian evaluations = 7
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations = 6
Total seconds in IPOPT = 1.332

EXIT: Optimal Solution Found.

```

### MadDiff as an AD backend of JuMP

MadDiff can be used as an automatic differentiation backend of JuMP. The problem above can be modeled in JuMP and solved with Ipopt along with MadDiff

```
using JuMP, Ipopt

m = JuMP.Model(Ipopt.Optimizer)

@variable(m, x[i=1:N], start=mod(i,2)==1 ? -1.2 : 1.)
@NLobjective(m, Min, sum(100(x[i-1]^2-x[i])^2+(x[i-1]-1)^2 for i=2:N))
@NLconstraint(m, [i=1:N-2],
  ↪ 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 == 0)

optimize!(m; differentiation_backend = MadDiffAD())
```

This is Ipopt version 3.14.4, running with linear solver MUMPS 5.4.1.

```
Number of nonzeros in equality constraint Jacobian...: 29994
Number of nonzeros in inequality constraint Jacobian.: 0
Number of nonzeros in Lagrangian Hessian.....: 19999
```

```
Total number of variables.....: 10000
      variables with only lower bounds: 0
      variables with lower and upper bounds: 0
      variables with only upper bounds: 0
Total number of equality constraints.....: 9998
Total number of inequality constraints.....: 0
      inequality constraints with only lower bounds: 0
      inequality constraints with lower and upper bounds: 0
      inequality constraints with only upper bounds: 0
```

iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
0	2.5405160e+06	2.48e+01	2.73e+01	-1.0	0.00e+00	-	0.00e+00	0.00e+00	0
1	1.3512419e+06	1.49e+01	8.27e+01	-1.0	2.20e+00	-	1.00e+00	1.00e+00f	1
2	1.5156131e+05	4.28e+00	1.36e+02	-1.0	1.43e+00	-	1.00e+00	1.00e+00f	1
3	6.6755024e+01	3.09e-01	2.18e+01	-1.0	5.63e-01	-	1.00e+00	1.00e+00f	1
4	6.2338933e+00	1.73e-02	8.47e-01	-1.0	2.10e-01	-	1.00e+00	1.00e+00h	1
5	6.2324586e+00	1.15e-05	8.16e-04	-1.7	3.35e-03	-	1.00e+00	1.00e+00h	1
6	6.2324586e+00	8.36e-12	7.97e-10	-5.7	2.00e-06	-	1.00e+00	1.00e+00h	1

Number of Iterations.....: 6

	(scaled)	(unscaled)
Objective.....:	7.8692659500479645e-01	6.2324586324379885e+00
Dual infeasibility.....:	7.9743417331632266e-10	6.3156786526652763e-09
Constraint violation.....:	8.3555384833289281e-12	8.3555384833289281e-12
Variable bound violation:	0.0000000000000000e+00	0.0000000000000000e+00
Complementarity.....:	0.0000000000000000e+00	0.0000000000000000e+00
Overall NLP error.....:	7.9743417331632266e-10	6.3156786526652763e-09

```
Number of objective function evaluations      = 7
Number of objective gradient evaluations      = 7
Number of equality constraint evaluations      = 7
Number of inequality constraint evaluations    = 0
```

```
Number of equality constraint Jacobian evaluations = 7
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations = 6
Total seconds in IPOPT = 1.482

EXIT: Optimal Solution Found.
```

---

This page was generated using [Literate.jl](#).

## **Part III**

# **How it Works**

## Chapter 5

### How it Works

---

This page was generated using [Literate.jl](#).



## **Part IV**

# **API Manual**

## Chapter 6

# MadDiffCore

### 6.1 MadDiffCore

[MadDiffCore.MadDiffCore](#) - Module.

| `MadDiffCore`

Core algorithm for MadDiff.

[source](#)

[MadDiffCore.AbstractExpression](#) - Type.

| `AbstractExpression{T <: AbstractFloat}`

Abstract type for expression, gradient, hessian, entry, and field evaluators.

[source](#)

[MadDiffCore.Entry](#) - Type.

| `Entry{T <: AbstractFloat}`

Abstract type for entry evaluators.

[source](#)

[MadDiffCore.Expression](#) - Type.

| `Expression{T <: AbstractFloat}`

Abstract type for expression evaluators.

[source](#)

[MadDiffCore.Expression1](#) - Type.

| `Expression1{T, F <: Function, E <: Expression{T}} <: Expression{T}`

Expression for univariate function

[source](#)

[MadDiffCore.Expression2](#) - Type.

```
| Expression2{T, F <: Function, E1, E2} <: Expression{T}
```

Expression for bivariate function

[source](#)

[MadDiffCore.ExpressionIfElse](#) - Type.

```
| ExpressionIfElse{T, E0 <: Expression{T}, E1, E2} <: Expression{T}
```

Expression for ifelse

[source](#)

[MadDiffCore.ExpressionSum](#) - Type.

```
| ExpressionSum{T, E <: Expression{T}, I} <: Expression{T}
```

Expression for a summation of Expressions

[source](#)

[MadDiffCore.Field](#) - Type.

```
| Field{T <: AbstractFloat}
```

Abstract type for field evaluators.

[source](#)

[MadDiffCore.Gradient](#) - Type.

```
| Gradient{T <: AbstractFloat}
```

Abstract type for gradient evaluators.

[source](#)

[MadDiffCore.Gradient](#) - Type.

```
| Gradient(e :: Expression{T}) where T
```

Returns the Gradient of an abstraction e.

[source](#)

[MadDiffCore.Gradient0](#) - Type.

```
| Gradient0{T} <: Gradient{T}
```

Gradient of Variable.

[source](#)

[MadDiffCore.Gradient1](#) - Type.

```
| Gradient1{T, F, D1 <: Gradient} <: Gradient{T}
```

Gradient of Expression1.

[source](#)

`MadDiffCore.Gradient2` - Type.

```
| Gradient2{T, F, D1 <: Gradient, D2 <: Gradient} <: Gradient{T}
```

Gradient of Expression2.

[source](#)

`MadDiffCore.Gradient2F1` - Type.

```
| Gradient2F1{T, F, D1 <: Gradient, R<: Real} <: Gradient{T}
```

Gradient of Expression2 whose first argument is <: Real.

[source](#)

`MadDiffCore.Gradient2F2` - Type.

```
| Gradient2F2{T, F, D1 <: Gradient, R<: Real} <: Gradient{T}
```

Gradient of Expression2 whose second argument is <: Real.

[source](#)

`MadDiffCore.GradientIfElse` - Type.

```
| GradientIfElse{T, G1, G2} <: Gradient{T}
```

Gradient of ExpressionIfElse

[source](#)

`MadDiffCore.GradientNull` - Type.

```
| GradientNull{T} <: Gradient{T}
```

Gradient of Parameter or Real.

[source](#)

`MadDiffCore.GradientSum` - Type.

```
| GradientSum{T, D <: Gradient{T}, I} <: Gradient{T}
```

Gradient of ExpressionSum.

[source](#)

`MadDiffCore.Hessian` - Type.

```
| Hessian{T <: AbstractFloat}
```

Abstract type for hessian evaluators.

[source](#)

`MadDiffCore.Hessian02` - Type.

```
| Hessian02{T, H11, H12, H21, H22} <: Hessian{T}
```

Hessian of ‘

[source](#)

`MadDiffCore.Hessian11` – Type.

```
| Hessian11{T,F,H1,H11} <: Hessian{T}
```

Hessian of Expression1

[source](#)

`MadDiffCore.Hessian11F1` – Type.

```
| Hessian11F1{T,F,H1,H11,R} <: Hessian{T}
```

Hessian of Expression2 whose first argument is <: Real.

[source](#)

`MadDiffCore.Hessian11F2` – Type.

```
| Hessian11F2{T,F,H1,H11,R} <: Hessian{T}
```

Hessian of Expression2 whose second argument is <: Real.

[source](#)

`MadDiffCore.HessianNull` – Type.

```
| HessianNull{T} <: Hessian{T} end
```

Hessian of linear expressions (e.g., Variable, Expression2{T, typeof(\*), Int64, Variable{T}} where T)

[source](#)

`MadDiffCore.Parameter` – Type.

```
| Parameter{T} <: Expression{T}
```

Expression for parameters.

[source](#)

`MadDiffCore.Parameter` – Method.

```
| Parameter(n::Int)
```

Returns a Parameter{Float64} whose index is n

### Example

```
| julia> e = Parameter(3)
p[3]
julia> non_caching_eval(e, [1.,2.,3.], [4.,5.,6.])
6.0
```

[source](#)

`MadDiffCore.Parameter` – Method.

```
| Parameter{T}(n::Int) where T
```

Returns a Parameter{T} whose index is n.

[source](#)

`MadDiffCore.Variable` – Type.

```
| Variable{T} <: Expression{T}
```

Expression for variables.

[source](#)

`MadDiffCore.Variable` – Method.

```
| Variable(n::Int)
```

Returns a Variable{Float64} whose index is n

### Example

```
julia> e = Variable(2)
x[2]
julia> non_caching_eval(e, [1.,2.,3.])
2.0
```

[source](#)

`MadDiffCore.Variable` – Method.

```
| Variable{T}(n::Int) where T
```

Returns a Variable{T} whose index is n.

[source](#)

## **Chapter 7**

# **MadDiffSpecialFunctions**

### **7.1 MadDiffMOI**

## Chapter 8

# MadDiffModels

### 8.1 MadDiffModels

`MadDiffModels.MadDiffModels` – Module.

```
| MadDiffModels
```

`MadDiffModels` is a submodule of `MadDiff`. `MadDiffModels` allows modeling nonlinear optimization problem of the following form:

```
| minimize:    f(x)
| subject to:  xl ≤ x ≤ xu
|              gl ≤ g(x) ≤ gu,
```

where:

- $x \in \mathbb{R}^n$  is the decision variable.
- $f : \mathbb{R}^n \rightarrow \mathbb{R}$  is the objective function
- $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$  is the constraint mapping.

The model is constructed as an `NLPModel` (see <https://github.com/JuliaSmoothOptimizers/NLPModels.jl>), and can be solved with various NLP solvers such as:

- `MadNLP` (<https://github.com/MadNLP/MadNLP.jl>)
- `Ipopt` (<https://github.com/JuliaSmoothOptimizers/NLPModelsIpopt.jl>)
- `Knitro` (<https://github.com/JuliaSmoothOptimizers/NLPModelsKnitro.jl>)

[source](#)

`MadDiffModels.Constraint` – Type.

```
| Constraint
```

A constraint index of `MadDiffModel`.

[source](#)

`MadDiffModels.MadDiffModel` – Type.

```
| MadDiffModel{T <: Real}
```



A mathematical model of a nonlineaer program.

[source](#)

`MadDiffModels.MadDiffModel` – Method.

```
| MadDiffModel()
```

Creates an empty `MadDiffModel{Float64}`.

**Example** `m = MadDiffModel(linear_solver = "ma27")`

[source](#)

`MadDiffModels.MadDiffModel` – Method.

```
| MadDiffModel{T}()
```

Creates an empty `MadDiffModel{T}`.

**Example** `m = MadDiffModel{Float32}()`

[source](#)

`MadDiffModels.ModelParameter` – Type.

```
| ModelParameter{T <: AbstractFloat} <: MadDiffCore.AbstractParameter{T}
```

A model parameter of `MadDiffModel`.

[source](#)

`MadDiffModels.ModelVariable` – Type.

```
| ModelVariable{T <: AbstractFloat} <: MadDiffCore.AbstractVariable{T}
```

A model variable of `MadDiffModel`.

[source](#)

`MadDiffModels.constraint` – Method.

```
| constraint(m::MadDiffModel, e::MadDiffCore.Expression; lb=0., ub=0.)
```

Adds a constraint to `MadDiffModel`. **Example** ““ `m = MadDiffModel()`

```
x = [variable(m) for i=1:3] constraint(m, x[1]^2 + 2*sin(x[2]) - exp(x[3]) >= 0) constraint(m, x[1]^4 +
x[2]^4 x[3]^4; lb = 0.1, ub = 1.)
```

[source](#)

`MadDiffModels.dual` – Method.

```
| dual(c::Constraint)
```

Retrun the dual of constraint `c`.

[source](#)

`MadDiffModels.instantiate!` – Method.

```
| instantiate!(m::MadDiffModel; sparse = true)
```

Instantiates the model `m`. The model must be instantiated before solving.

### Example

```
using MadDiff, NLPModelsIpopt

m = MadDiffModel()

x = [variable(m) for i=1:3]
objective(m, x[1]^2 + x[2]^2 + sin(x[3]))
constraint(m, 3x[2]^2 <= 1.)

instantiate!(m)
ipopt(m)
```

[source](#)

`MadDiffModels.lower_bound` – Method.

```
lower_bound(c::Constraint)
```

Retrun the lower bound of constraint `c`.

[source](#)

`MadDiffModels.lower_bound` – Method.

```
lower_bound(x::ModelVariable{T}) where T
```

Retrun the lower bound of variable `x`.

[source](#)

`MadDiffModels.objective` – Method.

```
objective(m::MadDiffModel, e::MadDiffCore.Expression)
```

Sets the objective function for `MadDiffModel`. Only minimization is supported. **Example**

```
m = MadDiffModel()

x = [variable(m) for i=1:3]
objective(m, x[1]^2 + x[2] + sin(x[3]))
```

[source](#)

`MadDiffModels.parameter` – Method.

```
parameter(m::MadDiffModel{T}, val)
```

Creates a parameter for `MadDiffModel` with value `val`. **Example**

```
m = MadDiffModel()

p = parameter(m, 0.5)
```

[source](#)

`MadDiffModels.set_lower_bound` – Method.

```
| set_lower_bound(c::Constraint, val)
```

Set the lower bound of constraint *c* to *val*.

[source](#)

`MadDiffModels.set_lower_bound` – Method.

```
| set_lower_bound(x::ModelVariable{T}, val) where T
```

Set the lower bound of variable '*x*' to *val*.

[source](#)

`MadDiffModels.set_upper_bound` – Method.

```
| set_upper_bound(c::Constraint, val)
```

Set the upper bound of constraint *c* to *val*.

[source](#)

`MadDiffModels.set_upper_bound` – Method.

```
| set_upper_bound(x::ModelVariable{T}, val) where T
```

Set the upper bound of variable '*x*' to *val*.

[source](#)

`MadDiffModels.setvalue` – Method.

```
| setvalue(p::ModelParameter{T}, val) where T
```

Set the value of parameter '*p*' to *val*.

[source](#)

`MadDiffModels.setvalue` – Method.

```
| setvalue(x::ModelVariable{T}, val) where T
```

Set the value of variable '*x*' to *val*.

[source](#)

`MadDiffModels.upper_bound` – Method.

```
| upper_bound(c::Constraint)
```

Retrun the upper bound of constraint *c*.

[source](#)

`MadDiffModels.upper_bound` – Method.

```
| upper_bound(x::ModelVariable{T}) where T
```

Retrun the upper bound of variable *x*.

[source](#)

`MadDiffModels.value` – Method.

```
| value(p::ModelParameter{T}) where T
```

Return the value of parameter p.

[source](#)

`MadDiffModels.value` – Method.

```
| value(x::ModelVariable{T}) where T
```

Return the value of variable x.

[source](#)

`MadDiffModels.variable` – Method.

```
| variable(m::MadDiffModel{T}; lb=-Inf, ub=Inf, start=0.)
```

Creates a variable for MadDiffModel.

#### Example

```
| m = MadDiffModel()
|x = variable(m; lb = -1, ub = 1, start = 0.5)
```

[source](#)

`NLPModels.cons!` – Method.

```
| NLPModels.cons!(m::MadDiffModel, x::AbstractVector, y::AbstractVector)
```

Evaluate the constraints of m at x and store the result in the vector y.

[source](#)

`NLPModels.get_ifix` – Function.

```
| get_ifix(m::MadDiffModel)
```

Return the value ifix from MadDiffModel.

[source](#)

`NLPModels.get_ifree` – Function.

```
| get_ifree(m::MadDiffModel)
```

Return the value ifree from MadDiffModel.

[source](#)

`NLPModels.get_iinf` – Function.

```
| get_iinf(m::MadDiffModel)
```

Return the value iinf from MadDiffModel.

[source](#)

`NLPModels.get_ilow` – Function.

```
| get_ilow(m::MadDiffModel)
```

Return the value `ilow` from `MadDiffModel`.

[source](#)

`NLPModels.get_irng` – Function.

```
| get_irng(m::MadDiffModel)
```

Return the value `irng` from `MadDiffModel`.

[source](#)

`NLPModels.get_islp` – Function.

```
| get_islp(m::MadDiffModel)
```

Return the value `islp` from `MadDiffModel`.

[source](#)

`NLPModels.get_iupp` – Function.

```
| get_iupp(m::MadDiffModel)
```

Return the value `iupp` from `MadDiffModel`.

[source](#)

`NLPModels.get_jfix` – Function.

```
| get_jfix(m::MadDiffModel)
```

Return the value `jfix` from `MadDiffModel`.

[source](#)

`NLPModels.get_jfree` – Function.

```
| get_jfree(m::MadDiffModel)
```

Return the value `jfree` from `MadDiffModel`.

[source](#)

`NLPModels.get_jinf` – Function.

```
| get_jinf(m::MadDiffModel)
```

Return the value `jinf` from `MadDiffModel`.

[source](#)

`NLPModels.get_jlow` – Function.

```
| get_jlow(m::MadDiffModel)
```

Return the value jlow from MadDiffModel.

[source](#)

[NLPMODELS.get\\_jrng](#) – Function.

```
| get_jrng(m::MadDiffModel)
```

Return the value jrng from MadDiffModel.

[source](#)

[NLPMODELS.get\\_jupp](#) – Function.

```
| get_jupp(m::MadDiffModel)
```

Return the value jupp from MadDiffModel.

[source](#)

[NLPMODELS.get\\_lcon](#) – Function.

```
| get_lcon(m::MadDiffModel)
```

Return the value lcon from MadDiffModel.

[source](#)

[NLPMODELS.get\\_lin](#) – Function.

```
| get_lin(m::MadDiffModel)
```

Return the value lin from MadDiffModel.

[source](#)

[NLPMODELS.get\\_lin\\_nnzj](#) – Function.

```
| get_lin_nnzj(m::MadDiffModel)
```

Return the value lin\_nnzj from MadDiffModel.

[source](#)

[NLPMODELS.get\\_lvar](#) – Function.

```
| get_lvar(m::MadDiffModel)
```

Return the value lvar from MadDiffModel.

[source](#)

[NLPMODELS.get\\_minimize](#) – Function.

```
| get_minimize(m::MadDiffModel)
```

Return the value minimize from MadDiffModel.

[source](#)

[NLPMODELS.get\\_name](#) – Function.

```
| get_name(m::MadDiffModel)
```

Return the value name from MadDiffModel.

[source](#)

[NLPMODELS.get\\_ncon](#) – Function.

```
| get_ncon(m::MadDiffModel)
```

Return the value ncon from MadDiffModel.

[source](#)

[NLPMODELS.get\\_nlin](#) – Function.

```
| get_nlin(m::MadDiffModel)
```

Return the value nlin from MadDiffModel.

[source](#)

[NLPMODELS.get\\_nln](#) – Function.

```
| get_nln(m::MadDiffModel)
```

Return the value nln from MadDiffModel.

[source](#)

[NLPMODELS.get\\_nln\\_nnzj](#) – Function.

```
| get_nln_nnzj(m::MadDiffModel)
```

Return the value nln\_nnzj from MadDiffModel.

[source](#)

[NLPMODELS.get\\_nlvb](#) – Function.

```
| get_nlvb(m::MadDiffModel)
```

Return the value nlvb from MadDiffModel.

[source](#)

[NLPMODELS.get\\_nlvc](#) – Function.

```
| get_nlvc(m::MadDiffModel)
```

Return the value nlvc from MadDiffModel.

[source](#)

[NLPMODELS.get\\_nlvo](#) – Function.

```
| get_nlvo(m::MadDiffModel)
```

Return the value nlvo from MadDiffModel.

[source](#)

`NLPModels.get_nnln` – Function.

```
| get_nnln(m::MadDiffModel)
```

Return the value nnln from MadDiffModel.

[source](#)

`NLPModels.get_nnzh` – Function.

```
| get_nnzh(m::MadDiffModel)
```

Return the value nnzh from MadDiffModel.

[source](#)

`NLPModels.get_nnzj` – Function.

```
| get_nnzj(m::MadDiffModel)
```

Return the value nnzj from MadDiffModel.

[source](#)

`NLPModels.get_nnzo` – Function.

```
| get_nnzo(m::MadDiffModel)
```

Return the value nnzo from MadDiffModel.

[source](#)

`NLPModels.get_nvar` – Function.

```
| get_nvar(m::MadDiffModel)
```

Return the value nvar from MadDiffModel.

[source](#)

`NLPModels.get_ucon` – Function.

```
| get_ucon(m::MadDiffModel)
```

Return the value ucon from MadDiffModel.

[source](#)

`NLPModels.get_uvar` – Function.

```
| get_uvar(m::MadDiffModel)
```

Return the value uvar from MadDiffModel.

[source](#)

`NLPModels.get_x0` – Function.

```
| get_x0(m::MadDiffModel)
```



Return the value x0 from MadDiffModel.

[source](#)

`NLPModels.get_y0` – Function.

```
| get_y0(m::MadDiffModel)
```

Return the value y0 from MadDiffModel.

[source](#)

`NLPModels.grad!` – Method.

```
| NLPModels.grad!(m::MadDiffModel, x::AbstractVector, y::AbstractVector)
```

Evaluate the gradient of m at x and store the result in the vector y.

[source](#)

`NLPModels.hess_coord!` – Method.

```
| NLPModels.hess_coord!(m::MadDiffModel, x::AbstractVector, lag::AbstractVector, z::AbstractVector;  
| ↪ obj_weight = 1.0)
```

Evaluate the Lagrangian Hessian of m at primal x, dual lag, and objective weight obj\_weight and store the result in the vector zin sparse coordinate format.

[source](#)

`NLPModels.hess_structure!` – Method.

```
| NLPModels.hess_structure!(m::MadDiffModel, I::AbstractVector{T}, J::AbstractVector{T})
```

Evaluate the structure of the Lagrangian Hessian and store the result in I and J in sparse coordinate format.

[source](#)

`NLPModels.jac_coord!` – Method.

```
| NLPModels.jac_coord!(m::MadDiffModel, x::AbstractVector, J::AbstractVector)
```

Evaluate the constraints Jacobian of m at x and store the result in the vector J in sparse coordinate format.

[source](#)

`NLPModels.jac_structure!` – Method.

```
| NLPModels.jac_structure!(m::MadDiffModel, I::AbstractVector{T}, J::AbstractVector{T})
```

Evaluate the structure of the constraints Jacobian and store the result in I and J in sparse coordinate format.

[source](#)

`NLPModels.obj` – Method.

```
| NLPModels.obj(m::MadDiffModel, x::AbstractVector)
```

Return the objective value of m at x.

[source](#)

## Chapter 9

# MadDiffMOI

### 9.1 MadDiffMOI

`MadDiffMOI.MadDiffMOI` – Module.

```
| MadDiffMOI
```

MadDiffMOI is a submodule of MadDiff. MadDiffMOI allows solving nonlinear optimization problems specified by MathOptInterface (<https://github.com/jump-dev/juMP.jl/tree/od/moi-nonlinear>).

[source](#)

`MadDiffCore.Expression` – Method.

```
Expression(ex::MOI.Nonlinear.Expression; subex = nothing)
```

Create a MadDiff.Expression from MOI.Expression.

[source](#)

`MadDiffCore.SparseNLPCore` – Method.

```
| MadDiffCore.SparseNLPCore(nlp_data::MOI.Nonlinear.Model)
```

Create MadDiffCore.SparseNLPCore from MOI.Nonlinear.Model.

[source](#)

`MadDiffMOI.MadDiffAD` – Type.

```
| MadDiffAD() <: MOI.Nonlinear.AbstractAutomaticDifferentiation
```

A differentiation backend for MathOptInterface based on MadDiff

[source](#)

`MadDiffMOI.MadDiffEvaluator` – Type.

```
| MadDiffEvaluator <: MOI.AbstractNLP evaluator
```

A type for callbacks for MathOptInterface's nonlinear model.

[source](#)

`MathOptInterface.NLPBlockData` – Method.

```
| MOI.NLPBlockData(evaluator::MadDiffEvaluator)
```

Create MOI.NLPBlockData from MadDiffEvaluator

[source](#)

[MathOptInterface.Nonlinear.Evaluator](#) – Method.

```
| MOI.Nonlinear.Evaluator(model::MOI.Nonlinear.Model, ::MadDiffAD, ::Vector{MOI.VariableIndex})
```

Create a MOI.Nonlinear.Evaluator from MOI.Nonlinear.Model using MadDiff's AD capability.

[source](#)

[MathOptInterface.eval\\_constraint](#) – Method.

```
| MOI.eval_constraint(evaluator::MadDiffEvaluator, g, x)
```

Evaluate the gradient of evaluator at x and store the result in the vector g.

[source](#)

[MathOptInterface.eval\\_constraint\\_jacobian](#) – Method.

```
| MOI.eval_constraint_jacobian(evaluator::MadDiffEvaluator, J, x)
```

Evaluate the constraints Jacobian of evaluator at x and store the result in the vector J in sparse coordinate format.

[source](#)

[MathOptInterface.eval\\_hessian\\_lagrangian](#) – Method.

```
| MOI.eval_hessian_lagrangian(evaluator::MadDiffEvaluator, H, x, σ, μ)
```

Evaluate the Lagrangian Hessian of evaluator at primal x, dual  $\mu$ , and objective weight  $\sigma$  and store the result in the vector H in sparse coordinate format.

[source](#)

[MathOptInterface.eval\\_objective](#) – Method.

```
| MOI.eval_objective(evaluator::MadDiffEvaluator, x)
```

Return the objective value of evaluator at x.

[source](#)

[MathOptInterface.eval\\_objective\\_gradient](#) – Method.

```
| MOI.eval_objective_gradient(evaluator::MadDiffEvaluator, g, x)
```

Evaluate the constraints of evaluator at x and store the result in the vector g.

[source](#)

[MathOptInterface.hessian\\_lagrangian\\_structure](#) – Method.

```
| MOI.hessian_lagrangian_structure(evaluator::MadDiffEvaluator)
```

Return the structure of the Lagrangian Hessian in `Vector{Tuple{Int,Int}}` format.

[source](#)

`MathOptInterface.jacobian_structure` - Method.

```
| MOI.jacobian_structure(evaluator::MadDiffEvaluator)
```

Return the structure of the constraints Jacobian in `Vector{Tuple{Int,Int}}` format.

[source](#)