Automatic Differentiation (or Differentiable Programming)

Atılım Güneş Baydin

National University of Ireland Maynooth

Joint work with Barak Pearlmutter

Alan Turing Institute, February 5, 2016

Hamilton Institute

Maynooth University
National University of Ireland Maynooth

- A brief introduction to AD
- My ongoing work

Vision

Functional programming languages with

- deeply embedded,
- general-purpose

differentiation capability, i.e., **automatic differentiation** (AD) in a functional framework

Vision

Functional programming languages with

- deeply embedded,
- general-purpose

differentiation capability, i.e., **automatic differentiation** (AD) in a functional framework

We started calling this **differentiable programming**

Christopher Olah's blog post (September 3, 2015)
`http://colah.github.io/posts/2015-09-NN-Types-FP/`

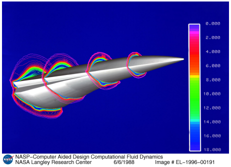
The AD field

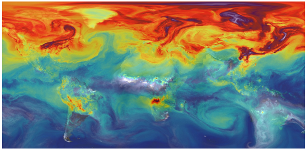
AD is an active research area

`http://www.autodiff.org/`

Traditional application domains of AD in industry and academia (Corliss et al., 2002; Griewank & Walther, 2008) include

- Computational fluid dynamics
- Atmospheric chemistry
- Engineering design optimization
- Computational finance





AD in probabilistic programming

(Wingate, Goodman, Stuhlmüller, Siskind. "Nonstandard interpretations of probabilistic programs for efficient inference." 2011)

- Hamiltonian Monte Carlo (Neal, 1994)
  `http://diffsharp.github.io/DiffSharp/examples-hamiltonianmontecarlo.html`
- No-U-Turn sampler (Hoffman & Gelman, 2011)
- Riemannian manifold HMC (Girolami & Calderhead, 2011)
- Optimization-based inference

Stan (Carpenter et al., 2015)
`http://mc-stan.org/`

What is AD?

Many machine learning frameworks (Theano, Torch, Tensorflow, CNTK) handle derivatives for you

- You build models by defining **computational graphs**
  $\rightarrow$ (constrained) symbolic language
  $\rightarrow$ highly limited control-flow (e.g., Theano's `scan`)
- The framework handles backpropagation
  $\rightarrow$ you don't have to code derivatives
  (unless adding new modules)
- Because derivatives are "automatic", some call it "autodiff" or "automatic differentiation"

What is AD?

Many machine learning frameworks (Theano, Torch, Tensorflow, CNTK) handle derivatives for you

- You build models by defining **computational graphs**
  $\rightarrow$ (constrained) symbolic language
  $\rightarrow$ highly limited control-flow (e.g., Theano's `scan`)
- The framework handles backpropagation
  $\rightarrow$ you don't have to code derivatives
  (unless adding new modules)
- Because derivatives are "automatic", some call it "autodiff" or "automatic differentiation"

This is NOT the traditional meaning of **automatic differentiation** (AD) (Griewank & Walther, 2008)

What is AD?

Many machine learning frameworks (Theano, Torch, Tensorflow, CNTK) handle derivatives for you

- You build models by defining **computational graphs**
  $\rightarrow$ (constrained) symbolic language
  $\rightarrow$ highly limited control-flow (e.g., Theano's `scan`)
- The framework handles backpropagation
  $\rightarrow$ you don't have to code derivatives
  (unless adding new modules)
- Because derivatives are "automatic", some call it "autodiff" or "automatic differentiation"

This is NOT the traditional meaning of **automatic differentiation** (AD) (Griewank & Walther, 2008)

Because "automatic" is a generic (and bad) term, **algorithmic differentiation** is a better name

What is AD?

- AD does not use symbolic graphs
- Gives numeric code that **computes the function AND its derivatives** at a given point

```
f(a, b):                  f'(a, a', b, b'):
   c = a * b        ⟶       (c, c') = (a*b, a'*b + a*b')
   d = sin c                 (d, d') = (sin c, c' * cos c)
   return d                  return (d, d')
```

- Derivatives propagated at the elementary operation level, as a side effect, at the same time when the function itself is computed
  $\rightarrow$ Prevents the "expression swell" of symbolic derivatives
- Full expressive capability of the host language
  $\rightarrow$ **Including conditionals, looping, branching**

Function evaluation traces

All **numeric evaluations** are sequences of elementary operations: a **“trace,”** also called a **“Wengert list”** (Wengert, 1964)

```
f(a, b):
   c = a * b
   if c > 0
     d = log c
   else
     d = sin c
   return d
```

Function evaluation traces

All **numeric evaluations** are sequences of elementary operations: a **“trace,”** also called a **“Wengert list”** (Wengert, 1964)

```
f(a, b):
   c = a * b
   if c > 0
     d = log c
   else
     d = sin c
   return d

f(2, 3)
```

Function evaluation traces

All **numeric evaluations** are sequences of elementary operations: a **"trace,"** also called a **"Wengert list"** (Wengert, 1964)

```
f(a, b):
   c = a * b
   if c > 0
     d = log c
   else
     d = sin c
   return d

f(2, 3)
```

```
a = 2

b = 3

c = a * b = 6

d = log c = 1.791

return 1.791
```

(primal)

Function evaluation traces

All **numeric evaluations** are sequences of elementary operations: a **“trace,”** also called a **“Wengert list”** (Wengert, 1964)

```
f(a, b):
   c = a * b
   if c > 0
     d = log c
   else
     d = sin c
   return d

f(2, 3)
```

```
a = 2

b = 3

c = a * b = 6

d = log c = 1.791

return 1.791
```

(primal)

```
a  = 2
a’ = 1
b  = 3
b’ = 0
c  = a * b = 6
c’ = a’ * b + a * b’ = 3
d  = log c = 1.791
d’ = c’ * (1 / c) = 0.5
return 1.791, 0.5
```

(tangent)

Function evaluation traces

All **numeric evaluations** are sequences of elementary operations: a **"trace,"** also called a **"Wengert list"** (Wengert, 1964)

```
f(a, b):
   c = a * b
   if c > 0
     d = log c
   else
     d = sin c
   return d

f(2, 3)
```

(primal)

```
a = 2

b = 3

c = a * b = 6

d = log c = 1.791

return 1.791
```

(tangent)

```
a  = 2
a' = 1
b  = 3
b' = 0
c  = a * b = 6
c' = a' * b + a * b' = 3
d  = log c = 1.791
d' = c' * (1 / c) = 0.5
return 1.791, 0.5
```

i.e., a Jacobian-vector product $\mathbf{J}_f\,(1,0)\big|_{(2,3)} = \frac{\partial}{\partial a} f(a,b)\big|_{(2,3)} = 0.5$

This is called the **forward (tangent) mode** of AD

Function evaluation traces

```
f(a, b):
   c = a * b
   if c > 0
     d = log c
   else
     d = sin c
   return d

f(2, 3)
```

Function evaluation traces

```
f(a, b):
   c = a * b
   if c > 0
     d = log c
   else
     d = sin c
   return d

f(2, 3)
```

```
a = 2
b = 3
c = a * b = 6
d = log c = 1.791
return 1.791
```

(primal)

Function evaluation traces

```
f(a, b):
   c = a * b
   if c > 0
     d = log c
   else
     d = sin c
   return d

f(2, 3)
```

```
a = 2
b = 3
c = a * b = 6
d = log c = 1.791
return 1.791
```

(primal)

```
a  = 2
b  = 3
c  = a * b = 6
d  = log c = 1.791
d’ = 1
c’ = d’ * (1 / c) = 0.166
b’ = c’ * a = 0.333
a’ = c’ * b = 0.5
return 1.791, 0.5, 0.333
```

(adjoint)

Function evaluation traces

```
f(a, b):
  c = a * b
  if c > 0
    d = log c
  else
    d = sin c
  return d

f(2, 3)
```

```
a = 2
b = 3
c = a * b = 6
d = log c = 1.791
return 1.791
```

(primal)

```
a  = 2
b  = 3
c  = a * b = 6
d  = log c = 1.791
d’ = 1
c’ = d’ * (1 / c) = 0.166
b’ = c’ * a = 0.333
a’ = c’ * b = 0.5
return 1.791, 0.5, 0.333
```

(adjoint)

i.e., a transposed Jacobian-vector product
$\mathbf{J}_f^T(1)\big|_{(2,3)} = \nabla f\big|_{(2,3)} = (0.5, 0.333)$

This is called the **reverse (adjoint) mode** of AD

Backpropagation is just a special case of the reverse mode: code a neural network objective computation, apply reverse AD

AD in a functional framework

AD has been around since the 1960s
(Wengert, 1964; Speelpenning, 1980; Griewank, 1989)

The foundations for AD in a functional framework
(Siskind & Pearlmutter, 2008; Pearlmutter & Siskind, 2008)

With research implementations

- R6RS-AD
  `https://github.com/qobi/R6RS-AD`
- Stalingrad
  `http://www.bcl.hamilton.ie/~qobi/stalingrad/`
- Alexey Radul's DVL
  `https://github.com/axch/dysvunctional-language`
- Recently, my DiffSharp library
  `http://diffsharp.github.io/DiffSharp/`

AD in a functional framework

"Generalized AD as a first-class function in an augmented λ-calculus" (Pearlmutter & Siskind, 2008)

Forward, reverse, and **any nested combination** thereof, instantiated according to usage scenario

Nested lambda expressions with free-variable references

$$\min\left(\lambda x\ .\ (f\ x) + \min\left(\lambda y\ .\ g\ x\ y\right)\right)$$

($\min$: gradient descent)

AD in a functional framework

"Generalized AD as a first-class function in an augmented λ-calculus" (Pearlmutter & Siskind, 2008)

Forward, reverse, and **any nested combination** thereof, instantiated according to usage scenario

Nested lambda expressions with free-variable references

$$\min\left(\lambda x \,.\, (f\ x) + \min\left(\lambda y \,.\, g\ x\ y\right)\right)$$

$$(\min\text{: gradient descent})$$

Must handle "perturbation confusion" (Manzyuk et al., 2012)

$$D\left(\lambda x \,.\, x \times \left(D\left(\lambda y \,.\, x + y\right) 1\right)\right) 1$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(x\left(\frac{\mathrm{d}}{\mathrm{d}y}x + y\right)\bigg|_{y=1}\right)\bigg|_{x=1} \stackrel{?}{=} 1$$

DiffSharp

`http://diffsharp.github.io/DiffSharp/`

- implemented in F#
- generalizes functional AD to high-performance linear algebra primitives
- arbitrary nesting of forward/reverse AD
- a comprehensive higher-order API
- gradients, Hessians, Jacobians, directional derivatives, matrix-free Hessian- and Jacobian-vector products
- F#'s “code quotations” (Syme, 2006) has great potential for deeply embedding transformation-based AD

DiffSharp

Higher-order differentiation API

	Op.	Value	Type signature	AD	Num.	Sym.
$f : \mathbb{R} \to \mathbb{R}$	`diff`	f'	$(\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}$	X, F	A	X
	`diff'`	(f, f')	$(\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to (\mathbb{R} \times \mathbb{R})$	X, F	A	X
	`diff2`	f''	$(\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}$	X, F	A	X
	`diff2'`	(f, f'')	$(\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to (\mathbb{R} \times \mathbb{R})$	X, F	A	X
	`diff2''`	(f, f', f'')	$(\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to (\mathbb{R} \times \mathbb{R} \times \mathbb{R})$	X, F	A	X
	`diffn`	$f^{(n)}$	$\mathbb{N} \to (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}$	X, F		X
	`diffn'`	$(f, f^{(n)})$	$\mathbb{N} \to (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to (\mathbb{R} \times \mathbb{R})$	X, F		X
$f : \mathbb{R}^n \to \mathbb{R}$	`grad`	∇f	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to \mathbb{R}^n$	X, R	A	X
	`grad'`	$(f, \nabla f)$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to (\mathbb{R} \times \mathbb{R}^n)$	X, R	A	X
	`gradv`	$\nabla f \cdot \mathbf{v}$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to \mathbb{R}^n \to \mathbb{R}$	X, F	A	
	`gradv'`	$(f, \nabla f \cdot \mathbf{v})$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to \mathbb{R}^n \to (\mathbb{R} \times \mathbb{R})$	X, F	A	
	`hessian`	$\mathbf{H}_f$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to \mathbb{R}^{n \times n}$	X, R-F	A	X
	`hessian'`	$(f, \mathbf{H}_f)$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to (\mathbb{R} \times \mathbb{R}^{n \times n})$	X, R-F	A	X
	`hessianv`	$\mathbf{H}_f \mathbf{v}$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to \mathbb{R}^n \to \mathbb{R}^n$	X, F-R	A	
	`hessianv'`	$(f, \mathbf{H}_f \mathbf{v})$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to \mathbb{R}^n \to (\mathbb{R} \times \mathbb{R}^n)$	X, F-R	A	
	`gradhessian`	$(\nabla f, \mathbf{H}_f)$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to (\mathbb{R}^n \times \mathbb{R}^{n \times n})$	X, R-F	A	X
	`gradhessian'`	$(f, \nabla f, \mathbf{H}_f)$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to (\mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^{n \times n})$	X, R-F	A	X
	`gradhessianv`	$(\nabla f \cdot \mathbf{v}, \mathbf{H}_f \mathbf{v})$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to \mathbb{R}^n \to (\mathbb{R} \times \mathbb{R}^n)$	X, F-R	A	
	`gradhessianv'`	$(f, \nabla f \cdot \mathbf{v}, \mathbf{H}_f \mathbf{v})$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to \mathbb{R}^n \to (\mathbb{R} \times \mathbb{R} \times \mathbb{R}^n)$	X, F-R	A	
	`laplacian`	$\mathrm{tr}(\mathbf{H}_f)$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to \mathbb{R}$	X, R-F	A	X
	`laplacian'`	$(f, \mathrm{tr}(\mathbf{H}_f))$	$(\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^n \to (\mathbb{R} \times \mathbb{R})$	X, R-F	A	X
$\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$	`jacobian`	$\mathbf{J}_\mathbf{f}$	$(\mathbb{R}^n \to \mathbb{R}^m) \to \mathbb{R}^n \to \mathbb{R}^{m \times n}$	X, F/R	A	X
	`jacobian'`	$(\mathbf{f}, \mathbf{J}_\mathbf{f})$	$(\mathbb{R}^n \to \mathbb{R}^m) \to \mathbb{R}^n \to (\mathbb{R}^m \times \mathbb{R}^{m \times n})$	X, F/R	A	X
	`jacobianv`	$\mathbf{J}_\mathbf{f} \mathbf{v}$	$(\mathbb{R}^n \to \mathbb{R}^m) \to \mathbb{R}^n \to \mathbb{R}^n \to \mathbb{R}^m$	X, F	A	
	`jacobianv'`	$(\mathbf{f}, \mathbf{J}_\mathbf{f} \mathbf{v})$	$(\mathbb{R}^n \to \mathbb{R}^m) \to \mathbb{R}^n \to \mathbb{R}^n \to (\mathbb{R}^m \times \mathbb{R}^m)$	X, F	A	
	`jacobianT`	$\mathbf{J}_\mathbf{f}^T$	$(\mathbb{R}^n \to \mathbb{R}^m) \to \mathbb{R}^n \to \mathbb{R}^{n \times m}$	X, F/R	A	X
	`jacobianT'`	$(\mathbf{f}, \mathbf{J}_\mathbf{f}^T)$	$(\mathbb{R}^n \to \mathbb{R}^m) \to \mathbb{R}^n \to (\mathbb{R}^m \times \mathbb{R}^{n \times m})$	X, F/R	A	X
	`jacobianTv`	$\mathbf{J}_\mathbf{f}^T \mathbf{v}$	$(\mathbb{R}^n \to \mathbb{R}^m) \to \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{R}^n$	X, R		
	`jacobianTv'`	$(\mathbf{f}, \mathbf{J}_\mathbf{f}^T \mathbf{v})$	$(\mathbb{R}^n \to \mathbb{R}^m) \to \mathbb{R}^n \to \mathbb{R}^m \to (\mathbb{R}^m \times \mathbb{R}^n)$	X, R		
	`jacobianTv''`	$(\mathbf{f}, \mathbf{J}_\mathbf{f}^T(\cdot))$	$(\mathbb{R}^n \to \mathbb{R}^m) \to \mathbb{R}^n \to (\mathbb{R}^m \times (\mathbb{R}^m \to \mathbb{R}^n))$	X, R		
	`curl`	$\nabla \times \mathbf{f}$	$(\mathbb{R}^3 \to \mathbb{R}^3) \to \mathbb{R}^3 \to \mathbb{R}^3$	X, F	A	X
	`curl'`	$(\mathbf{f}, \nabla \times \mathbf{f})$	$(\mathbb{R}^3 \to \mathbb{R}^3) \to \mathbb{R}^3 \to (\mathbb{R}^3 \times \mathbb{R}^3)$	X, F	A	X
	`div`	$\nabla \cdot \mathbf{f}$	$(\mathbb{R}^n \to \mathbb{R}^n) \to \mathbb{R}^n \to \mathbb{R}$	X, F	A	X
	`div'`	$(\mathbf{f}, \nabla \cdot \mathbf{f})$	$(\mathbb{R}^n \to \mathbb{R}^n) \to \mathbb{R}^n \to (\mathbb{R}^n \times \mathbb{R})$	X, F	A	X
	`curldiv`	$(\nabla \times \mathbf{f}, \nabla \cdot \mathbf{f})$	$(\mathbb{R}^3 \to \mathbb{R}^3) \to \mathbb{R}^3 \to (\mathbb{R}^3 \times \mathbb{R})$	X, F	A	X
	`curldiv'`	$(\mathbf{f}, \nabla \times \mathbf{f}, \nabla \cdot \mathbf{f})$	$(\mathbb{R}^3 \to \mathbb{R}^3) \to \mathbb{R}^3 \to (\mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R})$	X, F	A	X

DiffSharp

Matrix operations
`http://diffsharp.github.io/DiffSharp/api-overview.html`

High-performance OpenBLAS backend by default,
currently working on a CUDA-based GPU backend

Support for 64- and 32-bit floats (faster on many systems)

Benchmarking tool
`http://diffsharp.github.io/DiffSharp/benchmarks.html`

A growing collection of tutorials: gradient-based optimization algorithms, clustering, Hamiltonian Monte Carlo, neural networks, inverse kinematics

Hype

`http://hypelib.github.io/Hype/`

An experimental library for “compositional machine learning and **hype**rparameter optimization”, built on DiffSharp

A robust optimization core

- highly configurable functional modules
- SGD, conjugate gradient, Nesterov, AdaGrad, RMSProp, Newton’s method
- Use nested AD for gradient-based hyperparameter optimization (Maclaurin et al., 2015)

Hype

Extracts from Hype neural network code,
freely use F# and higher-order functions, don't think about gradients or backpropagation

https://github.com/hypelib/Hype/blob/master/src/Hype/Neural.fs

```
// Use mixed mode nested AD
open DiffSharp.AD.Float32

type FeedForward() =
    inherit Layer()
    // Feedforward layers executed as "fold", DM -> DM
    override n.Run(x:DM) = Array.fold Layer.run x layers

type GRU(inputs:int, memcells:int) =
    inherit Layer()
    // RNN many-to-many execution as "map", DM -> DM
    override l.Run (x:DM) =
        x |> DM.mapCols
            (fun x ->
                let z = sigmoid(l.Wxz * x + l.Whz * l.h + l.bz)
                let r = sigmoid(l.Wxr * x + l.Whr * l.h + l.br)
                let h' = tanh(l.Wxh * x + l.Whh * (l.h .* r))
                l.h <- (1.f - z) .* h' + z .* l.h
                l.h)
```

Hype

Derivatives are instantiated within the optimization code

```
type Method
    | CG -> // Conjugate gradient
        fun w f g p gradclip ->
            let v', g' = grad' f w // gradient
            let g' = gradclip g'
            let y = g' - g
            let b = (g' * y) / (p * y)
            let p' = -g' + b * p
            v', g', p'
    | NewtonCG -> // Newton conjugate gradient
        fun w f _ p gradclip ->
            let v', g' = grad' f w  // gradient
            let g' = gradclip g'
            let hv = hessianv f w p // Hessian-vector product
            let b = (g' * hv) / (p * hv)
            let p' = -g' + b * p
            v', g', p'
    | Newton -> // Newton's method
        fun w f _ _ gradclip ->
            let v', g', h' = gradhessian' f w // gradient, Hessian
            let g' = gradclip g'
            let p' = -DM.solveSymmetric h' g'
            v', g', p'
```

Hamiltonian Monte Carlo with DiffSharp

Try it on your system: `http://diffsharp.github.io/DiffSharp/examples-hamiltonianmontecarlo.html`

```
let leapFrog (u:DV->D) (k:DV->D) (d:D) steps (x0, p0) =
    let hd = d / 2.
    [1..steps]
    |> List.fold (fun (x, p) _ ->
        let p' = p - hd * grad u x
        let x' = x + d * grad k p'
        x', p' - hd * grad u x') (x0, p0)

let hmc n hdelta hsteps (x0:DV) (f:DV->D) =
    let u x = -log (f x) // potential energy
    let k p = (p * p) / D 2. // kinetic energy
    let hamilton x p = u x + k p
    let x = ref x0
    [|for i in 1..n do
        let p = DV.init x0.Length (fun _ -> rndn())
        let x', p' = leapFrog u k hdelta hsteps (!x, p)
        if rnd() < float (exp ((hamilton !x p) - (hamilton x' p'))) then x := x'
        yield !x|]
```

Thank You!

References

- Baydin AG, Pearlmutter BA, Radul AA, Siskind JM (Submitted) Automatic differentiation in machine learning: a survey [arXiv:1502.05767]
- Baydin AG, Pearlmutter BA, Siskind JM (Submitted) DiffSharp: automatic differentiation library [arXiv:1511.07727]
- Carpenter B, Hoffman MD, Brubaker M, Lee D, Li P, Betancourt M (2015) The Stan math library: reverse-mode automatic differentiation in C++. [arXiv:1509.07164]
- Griewank A, Walther A (2008) Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Society for Industrial and Applied Mathematics, Philadelphia [DOI 10.1137/1.9780898717761]
- Maclaurin D, David D, Adams RP (2015) Gradient-based Hyperparameter Optimization through Reversible Learning [arXiv:1502.03492]
- Manzyuk O, Pearlmutter BA, Radul AA, Rush DR, Siskind JM (2012) Confusion of tagged perturbations in forward automatic differentiation of higher-order functions [arXiv:1211.4892]
- Pearlmutter BA, Siskind JM (2008) Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. ACM TOPLAS 30(2):7 [DOI 10.1145/1330017.1330018]
- Siskind JM, Pearlmutter BA (2008) Nesting forward-mode AD in a functional framework. Higher Order and Symbolic Computation 21(4):361–76 [DOI 10.1007/s10990-008-9037-1]
- Syme D (2006) Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. 2006 Workshop on ML. ACM.
- Wengert R (1964) A simple automatic derivative evaluation program. Communications of the ACM 7:463–4