AD for CV, AC, ML, AI, and CL

Jeffrey Mark Siskind

Purdue University
School of Electrical and Computer Engineering

Friday 23 November 2012

Andrei Barbu, Daniel Barrett, Tommy Chang, Pranay Gupta, Seongwoon Ko, Aaron Michaux, Zachary Burchill, Siddharth Narayanaswamy, Haonan Yu

This work was supported, in part, by NSF grant CCF-0438806, by the Naval Research Laboratory under Contract Number N00173-10-1-G023, by the Army Research Laboratory accomplished under Cooperative Agreement Number W911NF-10-2-0060, and by computational resources provided by Information Technology at Purdue through its Rosen Center for Advanced Computing. Any views, opinions, findings, conclusions, or recommendations contained or expressed in this document or material are those of the author(s) and do not necessarily reflect or represent the views or official policies, either expressed or implied, of NSF, the Naval Research Laboratory, the Office of Naval Research, the Army Research Laboratory, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation herein.

Outline

Outline

Automatic Differentiation (AD)

Wengert (1964), Speelpenning (1980)

Automatic Differentiation (AD)

Wengert (1964), Speelpenning (1980)

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x}$$

Automatic Differentiation (AD)

Wengert (1964), Speelpenning (1980)

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x}$$

```
(derivative f)
```

Automatic Differentiation (AD)

Wengert (1964), Speelpenning (1980)

$$\frac{\mathtt{d}f(x)}{\mathtt{d}x}$$

```
(derivative f)
```

$$\frac{\partial f(\mathbf{x})}{\partial x_i}$$

Automatic Differentiation (AD)

Wengert (1964), Speelpenning (1980)

$$\frac{\mathtt{d}f(x)}{\mathtt{d}x}$$

```
(derivative f)
```

$$\frac{\partial f(\mathbf{x})}{\partial x_i}$$

```
(derivative (lambda (x) (f x y)))
```

Automatic Differentiation (AD)

Wengert (1964), Speelpenning (1980)

$\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ `(derivative f)`

$\frac{\partial f(\mathbf{x})}{\partial x_i}$ `(derivative (lambda (x) (f x y)))`

$\nabla_{\mathbf{x}} f(\mathbf{x})$

Automatic Differentiation (AD)

Wengert (1964), Speelpenning (1980)

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x}$$

```
(derivative f)
```

$$\frac{\partial f(\mathbf{x})}{\partial x_i}$$

```
(derivative (lambda (x) (f x y)))
```

$$\nabla_{\mathbf{x}} f(\mathbf{x})$$

```
(list (derivative (lambda (x) (f x y)))
      (derivative (lambda (y) (f x y))))
```

Automatic Differentiation (AD)

Wengert (1964), Speelpenning (1980)

$\frac{\mathrm{d}f(x)}{\mathrm{d}x}$	`(derivative f)`
$\frac{\partial f(\mathbf{x})}{\partial x_i}$	`(derivative (lambda (x) (f x y)))`
$\nabla_{\mathbf{x}} f(\mathbf{x})$	`(list (derivative (lambda (x) (f x y)))` `      (derivative (lambda (y) (f x y))))`
$\operatorname*{argmin}_{\mathbf{x}} f(\mathbf{x})$	

Automatic Differentiation (AD)

Wengert (1964), Speelpenning (1980)

$\dfrac{\mathtt{d}f(x)}{\mathtt{d}x}$	`(derivative f)`
$\dfrac{\partial f(\mathbf{x})}{\partial x_i}$	`(derivative (lambda (x) (f x y)))`
$\nabla_{\mathbf{x}} f(\mathbf{x})$	`(list (derivative (lambda (x) (f x y)))` `      (derivative (lambda (y) (f x y))))`
$\operatorname{argmin}_{\mathbf{x}} f(\mathbf{x})$	`(argmin f x0)`

Automatic Differentiation (AD)

Wengert (1964), Speelpenning (1980)

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x}$$

```
(derivative f)
```

$$\frac{\partial f(\mathbf{x})}{\partial x_i}$$

```
(derivative (lambda (x) (f x y)))
```

$$\nabla_{\mathbf{x}} f(\mathbf{x})$$

```
(list (derivative (lambda (x) (f x y)))
      (derivative (lambda (y) (f x y))))
```

$$\operatorname*{argmin}_{\mathbf{x}} f(\mathbf{x})$$

```
(argmin f x0)
```

$$\operatorname*{argmin}_{\mathbf{x}} f(\mathbf{x}, \mathbf{p})$$

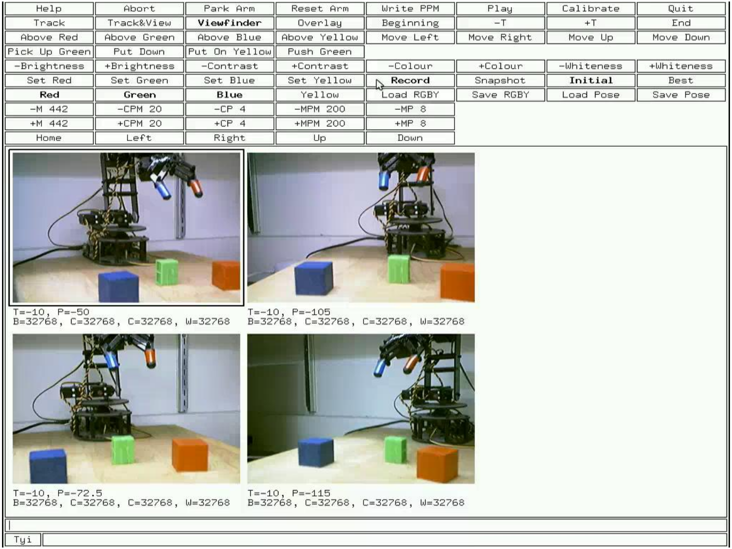
Automatic Differentiation (AD)

Wengert (1964), Speelpenning (1980)

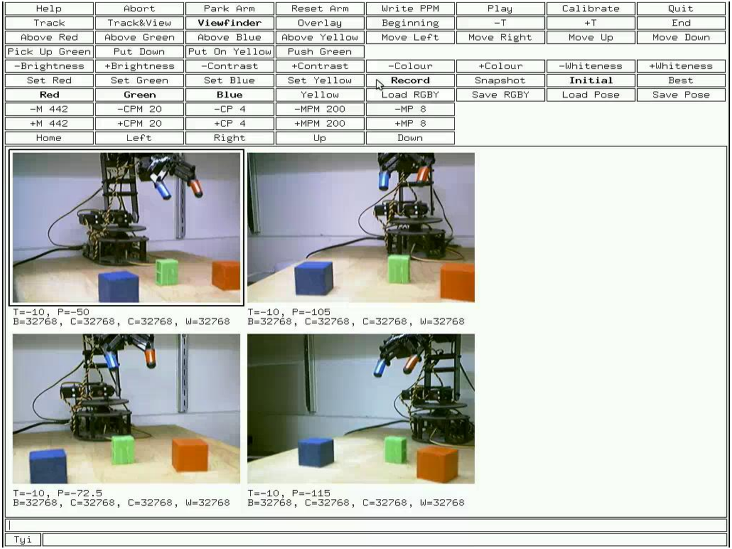
$\frac{\mathrm{d}f(x)}{\mathrm{d}x}$	`(derivative f)`
$\frac{\partial f(\mathbf{x})}{\partial x_i}$	`(derivative (lambda (x) (f x y)))`
$\nabla_{\mathbf{x}} f(\mathbf{x})$	`(list (derivative (lambda (x) (f x y))) (derivative (lambda (y) (f x y))))`
$\operatorname{argmin}_{\mathbf{x}} f(\mathbf{x})$	`(argmin f x0)`
$\operatorname{argmin}_{\mathbf{x}} f(\mathbf{x}, \mathbf{p})$	`(argmin (lambda (x) (f x p)) x0)`

Outline

Demo of Inverting Motor Programs



Demo of Inverting Motor Programs



How It Works

How It Works

z world coordinates of executor's end effector

How It Works

z	world coordinates of executor's end effector
a	arm configuration

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er})$, $c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er})$, $x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}),\ c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}),\ x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program
	inverse kinematics

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program
	inverse kinematics
	closed-loop visual servoing

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program
	inverse kinematics
	closed-loop visual servoing

x_o

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}),\ c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}),\ x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program
	inverse kinematics
	closed-loop visual servoing

$$p^{-1}(x_o, c_o)$$

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program
	inverse kinematics
	closed-loop visual servoing

$$p(p^{-1}(x_o, c_o), c_e)$$

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program
	inverse kinematics
	closed-loop visual servoing

$$m(p(p^{-1}(x_o, c_o), c_e))$$

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er})$, $c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er})$, $x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program
	inverse kinematics
	closed-loop visual servoing

$$f(m(p(p^{-1}(x_o, c_o), c_e)), b)$$

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er})$, $c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er})$, $x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program
	inverse kinematics
	closed-loop visual servoing
$\hat{x}_o = p(f(m(p(p^{-1}(x_o, c_o), c_e)), b), c_o)$	imagination capacity

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}),\ c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}),\ x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program
	inverse kinematics
	closed-loop visual servoing
$\hat{x}_o = p(f(m(p(p^{-1}(x_o, c_o), c_e)), b), c_o)$	imagination capacity
$\mathrm{argmin}_m \|x_o - \mathbf{lsh}(\hat{x}_o)\|$	classification

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}), c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}), x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program
	inverse kinematics
	closed-loop visual servoing
$\hat{x}_o = p(f(m(p(p^{-1}(x_o, c_o), c_e)), b), c_o)$	imagination capacity
$\operatorname{argmin}_m \|x_o - \mathbf{lsh}(\hat{x}_o)\|$	classification

How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}),\ c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}),\ x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program
	inverse kinematics
	closed-loop visual servoing
$\hat{x}_o = p(f(m(p(p^{-1}(x_o, c_o), c_e)), b), c_o)$	imagination capacity
$\mathrm{argmin}_m \lVert x_o - \mathbf{lsh}(\hat{x}_o) \rVert$	classification
$\mathrm{argmin}_m\ \min_{b, c_o, c_e} \lVert x_o - \mathbf{lsh}(\hat{x}_o) \rVert$	pose estimation by gradient descent

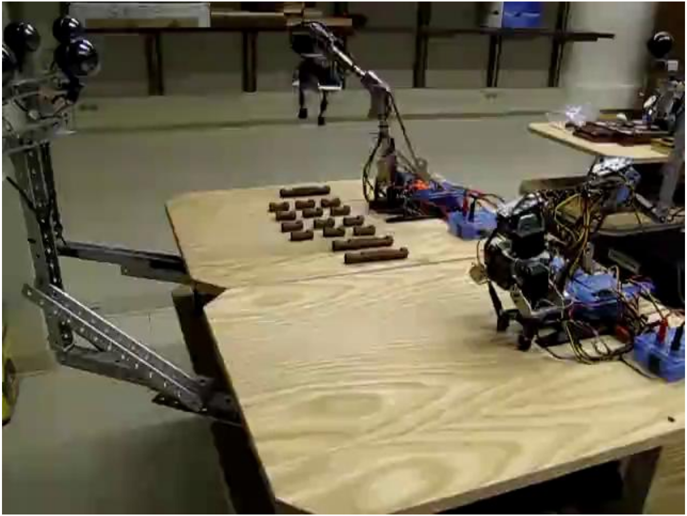
How It Works

z	world coordinates of executor's end effector
a	arm configuration
b	robot pose
$c_e = (c_{el}, c_{er}),\ c_o = (c_{ol}, c_{or})$	camera poses
$x_e = (x_{el}, x_{er}),\ x_o = (x_{ol}, x_{or})$	image coordinates of executor's end effector
$z = f(a, b)$	forward kinematics
$x = p(z, c)$	forward optics
$z = p^{-1}(x_l, x_r, c_l, c_r)$	inverse optics
	binocular vision
	not stereo
	robustness in the face of occlusion
	visual servoing along all axes
$a = m(x_e)$	motor program
	inverse kinematics
	closed-loop visual servoing
$\hat{x}_o = p(f(m(p(p^{-1}(x_o, c_o), c_e)), b), c_o)$	imagination capacity
$\operatorname{argmin}_m \lVert x_o - \mathbf{lsh}(\hat{x}_o) \rVert$	classification
$\operatorname{argmin}_m\ \min_{b,c_o,c_e} \lVert x_o - \mathbf{lsh}(\hat{x}_o) \rVert$	pose estimation by gradient descent
$\nabla_{b,c_o,c_e} \lVert x_o - \mathbf{lsh}(\hat{x}_o) \rVert$	calculated using AD

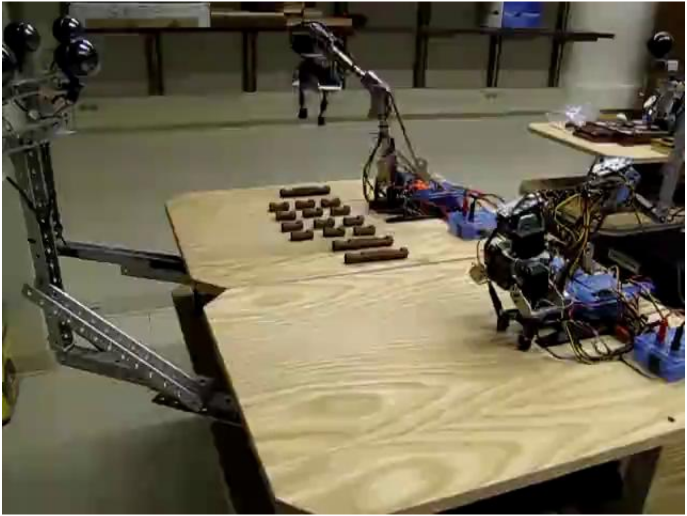
Robot Collaboration

Robot Collaboration

Robot Collaboration



Robot Collaboration



How It Works

Forward kinematics

$$\mathbf{x} = f(\theta, \mathbf{p})$$

How It Works

Forward kinematics

$$\mathbf{x} = f(\theta, \mathbf{p})$$

Estimating the parameters of the kinematic chain

$$\mathbf{p}^* = \operatorname*{argmin}_{\mathbf{p}} \sum_i \|\mathbf{x}_i - f(\theta_i, \mathbf{p})\|^2$$

How It Works

Forward kinematics

$$\mathbf{x} = f(\theta, \mathbf{p})$$

Estimating the parameters of the kinematic chain

$$\mathbf{p}^* = \operatorname*{argmin}_{\mathbf{p}} \sum_i \|\mathbf{x}_i - f(\theta_i, \mathbf{p})\|^2$$

Inverse kinematics

$$\begin{aligned} \theta &= f^{-1}(\mathbf{x}, \mathbf{p}) \\ &= \operatorname*{argmin}_{\theta} \|\mathbf{x} - f(\theta, \mathbf{p})\|^2 \end{aligned}$$

Outline

$$\begin{array}{cc|ccccc}
 & & & & B & & \\
 & & b_1 & \dots & b_j & \dots & b_n \\
\hline
 & a_1 & & & & & \\
 & \vdots & & \ddots & \vdots & & \\
A & a_i & & \dots & \textsc{Payoff}(a_i, b_j) & \dots & \\
 & \vdots & & & \vdots & \ddots & \\
 & a_m & & & & &
\end{array}$$

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ.

Game Theory

$$
\begin{array}{cc|ccccc}
 & & & & B & & \\
 & & b_1 & \ldots & b_j & \ldots & b_n \\
\hline
 & a_1 & & & & & \\
 & \vdots & & \ddots & \vdots & & \\
A & a_i & & \ldots & \textsc{Payoff}(a_i, b_j) & \ldots & \\
 & \vdots & & & \vdots & \ddots & \\
 & a_m & & & & &
\end{array}
$$

$$\max_{a \in A} \min_{b \in B} \textsc{Payoff}(a, b)$$

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ.

$$\begin{array}{cc|ccc} & & & \mathbb{R}^n & \\ & & \ldots & \mathbf{b} & \ldots \\ \hline & \vdots & \ddots & \vdots & \\ \mathbb{R}^m & \mathbf{a} & \ldots & \textsc{Payoff}(\mathbf{a},\mathbf{b}) & \ldots \\ & \vdots & & \vdots & \ddots \end{array}$$

$$\max_{\mathbf{a}\in\mathbb{R}^m}\min_{\mathbf{b}\in\mathbb{R}^n}\textsc{Payoff}(\mathbf{a},\mathbf{b})$$

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ.

```
      subroutine deriv2(f, x, y, yprime)
      external f
        adf(tangent(x) = 1.0)
        y = f(x)
        end adf(yprime = tangent(y))
      end

      function root(f, x0, n)
      x = x0
      do 1669 i = 1, n
      call deriv2(f, x, y, yprime)
 1669 x = x-y/yprime
      root = x
      end

      function deriv1(f, x)
      external f
        adf(tangent(x) = 1.0)
        y = f(x)
        end adf(deriv1 = tangent(y))
      end
```

Continuous-Strategy Two-Person Nonzero-Sum Game

VLAD/STALINGRAD

```
(define (deriv2 f x) (j* f (times 0 f) x 1))

(define (root f x n)
 (if (zero? n)
     x
     (let (((cons y yprime) (deriv2 f x)))
      (root f (- x (/ y yprime)) (- n 1)))))

(define (deriv1 f x)
 (let (((cons y y-tangent) (j* f (times 0 f) x 1))) y-tangent))
```

Continuous-Strategy Two-Person Nonzero-Sum Game

Farfel/Farfallen

```
function argmax(f, x0, n)
  function fprime(x)
  fprime = deriv1(f, x)
  end
argmax = root(fprime, x0, n)
end

subroutine eqlbrm(biga, bigb, astar, bstar, n)
external biga, bigb
  function f(astar)
    function g(a)
      function h(b)
      h = bigb(astar, b)
      end
    bstar = argmax(h, bstar, n)
    g = biga(a, bstar)
    end
  f = argmax(g, astar, n)-astar
  end
astar = root(f, astar, n)
end
```

Continuous-Strategy Two-Person Nonzero-Sum Game


```
(define (argmax f x0 n)
 (define (fprime x) (deriv1 f x))
 (root fprime x0 n))

(define (eqlbrm biga bigb astar bstar n)
 (define (f astar)
  (define (g a)
   (define (h b) (bigb astar b))
   (biga a (argmax h bstar n)))
  (- (argmax g astar n) astar))
 (let ((astar (root f astar n)))
  (define (h b) (bigb astar b))
  (let ((bstar (argmax h bstar n)))
   (cons astar bstar))))
```

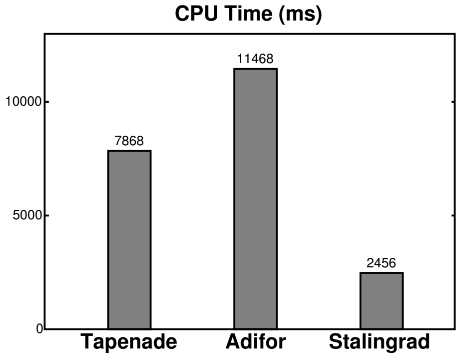
```
function gmbiga(a, b)
price = 20-0.1*a-0.1*b
costs = a*(10-0.05*a)
gmbiga = a*price-costs
end

function gmbigb(a, b)
price = 20-0.1*b-0.0999*a
costs = b*(10.005-0.05*b)
gmbigb = b*price-costs
end

program main
read *, astar
read *, bstar
read *, n
call eqlbrm(gmbiga, gmbigb, astar, bstar, n)
print *, astar, bstar
end
```

Continuous-Strategy Two-Person Nonzero-Sum Game

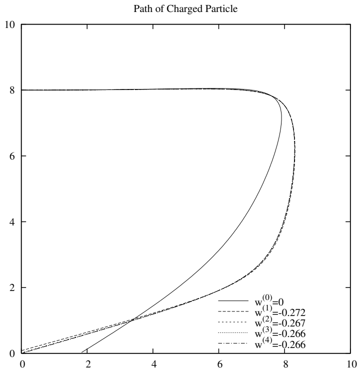
VLAD/STALINGRAD

```
(define (gmbiga a b)
 (let ((price (- (- 20 (* 0.1 a)) (* 0.1 b)))
       (costs (* a (- 10 (* 0.05 a)))))
  (- (* a price) costs)))

(define (gmbigb a b)
 (let ((price (- (- 20 (* 0.1 b)) (* 0.0999 a)))
       (costs (* b (- 10.005 (* 0.05 b)))))
  (- (* b price) costs)))

(let (((cons astar bstar) (eqlbrm gmbiga gmbigb 0 0 (real 500))))
 (cons (write-real astar) (write-real bstar)))
```

Performance Comparison



Cathode Ray Tubes



$$
\begin{aligned}
\text{potential: } p(\mathbf{x};w) &= \|\mathbf{x}-(10,10-w)\|^{-1}+\|\mathbf{x}-(10,0)\|^{-1}\\
\ddot{\mathbf{x}}(t) &= -\nabla_{\mathbf{x}}\, p(\mathbf{x})|_{\mathbf{x}=\mathbf{x}(t)}\\
\dot{\mathbf{x}}(t+\Delta t) &= \dot{\mathbf{x}}(t)+\Delta t\,\ddot{\mathbf{x}}(t)\\
\mathbf{x}(t+\Delta t) &= \mathbf{x}(t)+\Delta t\,\dot{\mathbf{x}}(t)\\
\text{When: } x_1(t+\Delta t) &\leq 0\\
\text{let: } \Delta t_f &= -x_1(t)/\dot{x}_1(t)\\
t_f &= t+\Delta t_f\\
\mathbf{x}(t_f) &= \mathbf{x}(t)+\Delta t_f\,\dot{\mathbf{x}}(t)\\
\text{Error: } E(w) &= x_0(t_f)^2\\
\text{Find: } & \operatorname*{argmin}_w E(w)
\end{aligned}
$$

Sprague, C. S. and George, R. H. (1939). *Cathode Ray Deflecting Electrode*. US Patent 2,161,437.

George, R. H. (1940). *Cathode Ray Tube*. US Patent 2,222,942.

Performance Comparison

		particle				saddle			
		FF	FR	RF	RR	FF	FR	RF	RR
VLAD	Stalingrad	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Fortran	ADIFOR	1.52	🟦	🟦	🟦	2.07	🟦	🟦	🟦
	Tapenade	3.40	🟥	🟩	🟥	2.56	🟥	🟩	🟥
C++	FADBAD++	65.69	🟩	🟩	🟩	22.44	🟩	🟩	🟩
ML	MLton	53.89	88.88	16.08	28.06	40.39	51.21	1.86	2.67
	OCaml	160.50	340.35	147.91	263.66	107.71	156.33	6.75	13.51
	SML/NJ	106.21	182.45	105.04	185.15	84.38	106.01	3.55	6.31
Haskell	GHC	165.22	🟥	🟥	🟥	121.18	🟥	🟥	🟥
Scheme	Bigloo	505.90	761.40	104.81	228.56	423.69	440.25	15.77	24.59
	Chicken	1120.37	2026.31	425.60	1872.85	889.58	1144.65	35.73	68.94
	Gambit	444.13	752.63	138.34	256.30	362.65	420.48	14.08	23.87
	Ikarus	192.07	312.28	61.79	114.87	158.88	205.97	6.75	11.40
	Larceny	726.59	1108.18	144.55	270.14	571.81	613.65	19.14	29.77
	MIT Scheme	1472.26	2500.00	309.66	591.36	1243.26	1428.57	51.36	79.10
	MzC	2073.26	3434.64	340.30	655.83	2436.26	1996.40	72.45	150.02
	MzScheme	2344.70	4076.16	409.95	843.68	2000.89	2332.43	80.78	134.00
	Scheme->C	391.42	605.26	109.77	198.43	324.95	328.84	12.74	18.28
	SCMUTILS	3321.20	🟦	🟦	🟦	2800.71	🟦	🟦	🟦
	Stalin	208.10	366.08	51.84	91.86	166.96	212.93	7.68	11.40

- 🟩 not implemented but could implement
- 🟦 not implemented in existing tool
- 🟥 can't implement

Outline

Gradient-Based Optimization

```
(define (e i n)
 (if (zero? n)
     '()
     (cons (if (zero? i) 1.0 0.0)
           (e (- i 1) (- n 1)))))
```

Gradient-Based Optimization

```
(define (e i n)
 (if (zero? n)
     '()
     (cons (if (zero? i) 1.0 0.0)
           (e (- i 1) (- n 1)))))

(define ((gradient f) x)
 (let ((n (length x)))
  (map (lambda (i) (tangent ((j* f) (bundle x (e i n)))))
       (iota n))))
```

Gradient-Based Optimization

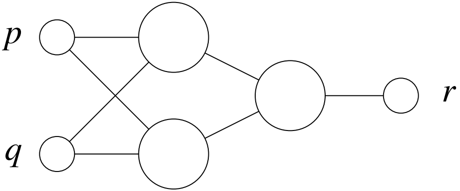
```
(define (e i n)
 (if (zero? n)
     '()
     (cons (if (zero? i) 1.0 0.0)
           (e (- i 1) (- n 1)))))

(define ((gradient f) x)
 (let ((n (length x)))
  (map (lambda (i) (tangent ((j* f) (bundle x (e i n)))))
       (iota n))))

(define (gradient-ascent f x0 n eta)
 (if (zero? n)
     (list x0 (f x0) ((gradient f) x0))
     (gradient-ascent f
                      (zip (lambda (xi gi) (+ xi (* eta gi)))
                           x0
                           ((gradient f) x0))
                      (- n 1)
                      eta)))
```

Gradient-Based Optimization

```
(define ((gradient f) x) (cdr ((cdr ((*j f) (*j x))) 1.0)))
```

```
(define (gradient-ascent f x0 n eta)
 (if (zero? n)
     (list x0 (f x0) ((gradient f) x0))
     (gradient-ascent f
                      (zip (lambda (xi gi) (+ xi (* eta gi)))
                           x0
                           ((gradient f) x0))
                      (- n 1)
                      eta)))
```

Neural Networks



p	q	r
0	0	0
0	1	1
1	0	1
1	1	0

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). *Learning representations by back-propagating errors*. Nature, **323**:533–6.

Neural Networks in VLAD

```
(define ((sum-activities activities) bias ws)
 ((fold + bias) (zip * ws activities)))
```

Neural Networks in VLAD

```
(define ((sum-activities activities) bias ws)
 ((fold + bias) (zip * ws activities)))

(define (sum-layer activities ws-layer)
 (map (sum-activities activities) ws-layer))
```

Neural Networks in VLAD

```
(define ((sum-activities activities) bias ws)
 ((fold + bias) (zip * ws activities)))

(define (sum-layer activities ws-layer)
 (map (sum-activities activities) ws-layer))

(define (sigmoid x) (/ 1 (+ (exp (- 0 x)) 1)))
```

Neural Networks in VLAD

```
(define ((sum-activities activities) bias ws)
 ((fold + bias) (zip * ws activities)))

(define (sum-layer activities ws-layer)
 (map (sum-activities activities) ws-layer))

(define (sigmoid x) (/ 1 (+ (exp (- 0 x)) 1)))

(define ((forward-pass ws-layers) in)
 (if (null? ws-layers)
     in
     ((forward-pass (cdr ws-layers))
      (map sigmoid (sum-layer in (car ws-layers))))))
```

Neural Networks in VLAD

```
(define ((sum-activities activities) bias ws)
 ((fold + bias) (zip * ws activities)))

(define (sum-layer activities ws-layer)
 (map (sum-activities activities) ws-layer))

(define (sigmoid x) (/ 1 (+ (exp (- 0 x)) 1)))

(define ((forward-pass ws-layers) in)
 (if (null? ws-layers)
     in
     ((forward-pass (cdr ws-layers))
      (map sigmoid (sum-layer in (car ws-layers))))))

(define ((error-on-dataset dataset) ws-layers)
 ((fold + 0)
  (map (lambda ((list in target))
        (* 0.5 (magnitude-squared (v- ((forward-pass ws-layers) in) target))))
       dataset)))
```

Neural Networks in VLAD

```
(define ((sum-activities activities) bias ws)
 ((fold + bias) (zip * ws activities)))

(define (sum-layer activities ws-layer)
 (map (sum-activities activities) ws-layer))

(define (sigmoid x) (/ 1 (+ (exp (- 0 x)) 1)))

(define ((forward-pass ws-layers) in)
 (if (null? ws-layers)
     in
     ((forward-pass (cdr ws-layers))
      (map sigmoid (sum-layer in (car ws-layers))))))

(define ((error-on-dataset dataset) ws-layers)
 ((fold + 0)
  (map (lambda ((list in target))
        (* 0.5 (magnitude-squared (v- ((forward-pass ws-layers) in) target))))
       dataset)))

(gradient-descent (error-on-dataset '(((0 0) (0))
                                      ((0 1) (1))
                                      ((1 0) (1))
                                      ((1 1) (0))))
                  '(((0 -0.284227 1.16054) (0 0.617194 1.30467))
                    ((0 -0.084395 0.648461)))
                  1000.0
                  0.3)
```

Performance Comparison

		backprop		
		Fs	Fv	R
VLAD	STALINGRAD	1.00	🟩	1.00
FORTRAN	ADIFOR	11.84	2.68	🟦
	TAPENADE	11.35	4.33	6.24
C	ADIC	16.33	3.93	🟦
C++	ADOL-C	12.34	3.89	35.53
	CPPAD	42.15	🟦	23.69
	FADBAD++	98.96	33.15	53.03
ML	MLTON	73.94	🟩	37.94
	OCAML	157.75	🟩	149.14
	SML/NJ	142.71	🟩	94.97
HASKELL	GHC	🟩	🟩	🟥
SCHEME	BIGLOO	577.45	🟩	306.60
	CHICKEN	1391.75	🟩	971.91
	GAMBIT	545.20	🟩	341.73
	IKARUS	216.42	🟩	147.49
	LARCENY	955.98	🟩	486.64
	MIT SCHEME	1900.04	🟩	1141.22
	MZC	2439.93	🟩	1571.52
	MZSCHEME	3477.86	🟩	1866.28
	SCHEME->C	484.24	🟩	233.75
	SCMUTILS	4544.48	🟦	🟦
	STALIN	832.68	🟩	367.84

- 🟩 not implemented but could implement
- 🟦 not implemented in existing tool
- 🟥 can't implement

Outline

Probabilistic Lambda Calculus

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

Probabilistic Lambda Calculus

$P = \textbf{if}\ x_0\ \textbf{then}\ 0\ \textbf{else}\ \textbf{if}\ x_1\ \textbf{then}\ 1\ \textbf{else}\ 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\qquad$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$
$\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\qquad$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

Probabilistic Lambda Calculus

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\quad$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$
$\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\quad$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

$$\Pr(\mathcal{E}(P) = 0|p_0, p_1) = p_0$$
$$\Pr(\mathcal{E}(P) = 1|p_0, p_1) = (1 - p_0)p_1$$
$$\Pr(\mathcal{E}(P) = 2|p_0, p_1) = (1 - p_0)(1 - p_1)$$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

Probabilistic Lambda Calculus

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\qquad \Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$
$\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\qquad \Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

$\Pr(\mathcal{E}(P) = 0|p_0, p_1) = p_0$
$\Pr(\mathcal{E}(P) = 1|p_0, p_1) = (1 - p_0)p_1$
$\Pr(\mathcal{E}(P) = 2|p_0, p_1) = (1 - p_0)(1 - p_1)$

$$\prod_{v \in \{0,1,2,2\}} \Pr(\mathcal{E}(P) = v|p_0, p_1) = p_0(1 - p_0)^3 p_1 (1 - p_1)^2$$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

Probabilistic Lambda Calculus

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\quad$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$

$\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\quad$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

$$\Pr(\mathcal{E}(P) = 0|p_0, p_1) = p_0$$
$$\Pr(\mathcal{E}(P) = 1|p_0, p_1) = (1 - p_0)p_1$$
$$\Pr(\mathcal{E}(P) = 2|p_0, p_1) = (1 - p_0)(1 - p_1)$$

$$\prod_{v \in \{0,1,2,2\}} \Pr(\mathcal{E}(P) = v|p_0, p_1) = p_0(1 - p_0)^3 p_1(1 - p_1)^2$$

$$\operatorname*{argmax}_{p_0, p_1} \prod_{v \in \{0,1,2,2\}} \Pr(\mathcal{E}(P) = v|p_0, p_1) = \left\langle \frac{1}{4}, \frac{1}{3} \right\rangle$$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

Probabilistic Prolog

```
p(0).
p(X):-q(X).
q(1).
q(2).
```

Probabilistic Prolog

$\Pr(\texttt{p(0).}) = p_0$
$\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$
$\Pr(\texttt{q(1).}) = p_1$
$\Pr(\texttt{q(2).}) = 1 - p_1$

Probabilistic Prolog

$\Pr(\texttt{p(0).}) = p_0$
$\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$
$\Pr(\texttt{q(1).}) = p_1$
$\Pr(\texttt{q(2).}) = 1 - p_1$

$\Pr(\texttt{?-p(0).}) = p_0$
$\Pr(\texttt{?-p(1).}) = (1 - p_0)p_1$
$\Pr(\texttt{?-p(2).}) = (1 - p_0)(1 - p_1)$

Probabilistic Prolog

$$\Pr(\texttt{p(0).}) = p_0$$
$$\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$$
$$\Pr(\texttt{q(1).}) = p_1$$
$$\Pr(\texttt{q(2).}) = 1 - p_1$$

$$\Pr(\texttt{?-p(0).}) = p_0$$
$$\Pr(\texttt{?-p(1).}) = (1 - p_0)p_1$$
$$\Pr(\texttt{?-p(2).}) = (1 - p_0)(1 - p_1)$$

$$\prod_{q \in \{\texttt{p(0)},\texttt{p(1)},\texttt{p(2)},\texttt{p(2)}\}} \Pr(\texttt{?-}q\texttt{.}) = p_0(1 - p_0)^3 p_1(1 - p_1)^2$$

Probabilistic Prolog

$$\Pr(\texttt{p(0).}) = p_0$$
$$\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$$
$$\Pr(\texttt{q(1).}) = p_1$$
$$\Pr(\texttt{q(2).}) = 1 - p_1$$

$$\Pr(\texttt{?-p(0).}) = p_0$$
$$\Pr(\texttt{?-p(1).}) = (1 - p_0)p_1$$
$$\Pr(\texttt{?-p(2).}) = (1 - p_0)(1 - p_1)$$

$$\prod_{q \in \{\texttt{p(0)},\texttt{p(1)},\texttt{p(2)},\texttt{p(2)}\}} \Pr(\texttt{?-}q\texttt{.}) = p_0(1 - p_0)^3 p_1(1 - p_1)^2$$

$$\operatorname*{argmax}_{p_0,p_1} \prod_{q \in \{\texttt{p(0)},\texttt{p(1)},\texttt{p(2)},\texttt{p(2)}\}} \Pr(\texttt{?-}q\texttt{.}) = \left\langle \frac{1}{4}, \frac{1}{3} \right\rangle$$

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment))))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
        (map-tagged-distribution
         (lambda (value) (value tagged-distribution))
         (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...)))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...)))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x₀ then 0 else if x₁ then 1 else 2
                   (list Pr(x₀ ↦ true) = p₀   Pr(x₀ ↦ false) = 1 − p₀
                         Pr(x₁ ↦ true) = p₁   Pr(x₁ ↦ false) = 1 − p₁
                         ...)))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...)))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...)))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
        (evaluate **if** x_0 **then** 0 **else if** x_1 **then** 1 **else** 2
                (list $\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$
                     $\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$
                     ...)))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x₀ then 0 else if x₁ then 1 else 2
                   (list Pr(x₀ ↦ true) = p₀   Pr(x₀ ↦ false) = 1 − p₀
                         Pr(x₁ ↦ true) = p₁   Pr(x₁ ↦ false) = 1 − p₁
                         ...)))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses))))))
   clauses)))
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 - p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 - p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 - p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 - p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 − p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 − p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 - p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 - p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 - p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 - p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 - p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 - p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 - p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 - p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Generated Code

```
static void f2679(double a_f2679_0,double a_f2679_1,double a_f2679_2,double a_f2679_3){
  int t272381=((a_f2679_2==0.)?0:1);
  double t272406;
  double t272405;
  double t272404;
  double t272403;
  double t272402;
  if((t272381==0)){
    double t272480=(1.-a_f2679_0);
    double t272572=(1.-a_f2679_1);
    double t273043=(a_f2679_0+0.);
    double t274185=(t272480*a_f2679_1);
    double t274426=(t274185+0.);
    double t275653=(t272480*t272572);
    double t275894=(t275653+0.);
    double t277121=(t272480*t272572);
    double t277362=(t277121+0.);
    double t277431=(t277362*1.);
    double t277436=(t275894*t277431);
    double t277441=(t274426*t277436);
    double t277446=(t273043*t277441);
    ...
    double t1777107=(t1774696+t1715394);
    double t1777194=(0.-t1745420);
    double t1778533=(t1777194+t1419700);
    t272406=a_f2679_0;
    t272405=a_f2679_1;
    t272404=t277446;
    t272403=t1778533;
    t272402=t1777107;}
  else {...}
  r_f2679_0=t272406;
  r_f2679_1=t272405;
  r_f2679_2=t272404;
  r_f2679_3=t272403;
  r_f2679_4=t272402;}
```

Performance Comparison

		probabilistic-lambda-calculus F	probabilistic-lambda-calculus R	probabilistic-prolog F	probabilistic-prolog R
VLAD	STALINGRAD	1.00	1.00	1.00	1.00
ML	MLTON	106.45	124.95	789.41	483.47
	OCAML	215.73	538.68	1207.13	1534.61
	SML/NJ	197.75	272.45	2448.02	1471.94
HASKELL	GHC	■	■	■	■
SCHEME	BIGLOO	832.92	1048.11	14422.16	8286.06
	CHICKEN	2305.98	3283.00	66948.70	37792.84
	GAMBIT	879.88	1153.86	24316.03	13649.81
	IKARUS	437.46	531.10	8242.92	4845.86
	LARCENY	1651.01	1673.22	25589.62	14833.53
	MIT SCHEME	3491.10	4130.19	85819.57	48335.38
	MZC	5289.17	5929.14	154206.95	83480.27
	MZSCHEME	6235.78	7134.71	166129.12	91630.70
	SCHEME->C	682.15	794.31	10530.66	5980.27
	SCMUTILS	6456.99	■	80100.23	■
	STALIN	1240.73	1137.41	22511.79	10986.43

- ■ not implemented but could implement, including FORTRAN, C, and C++
- ■ not implemented in existing tool
- ■ can't implement

Outline

Maximum-Likelihood Methods

Probabilistic Programming

(flip p)

Maximum-Likelihood Methods

Probabilistic Programming

`(flip` p`)`

`(probability` e`)`

(`flip` p)

(`probability` e)

(`argmax` f x_0)

Reduced Gradient

Wolfe (1962, 1967)

$$\operatorname*{argmax}_{\mathbf{x}} f(\mathbf{x})$$

Reduced Gradient

Wolfe (1962, 1967)

$$\operatorname*{argmax}_{\mathbf{x}} f(\mathbf{x})$$

$$\operatorname*{argmax}_{\substack{\mathbf{x}\\ \mathbf{x}\geq 0\\ \mathbf{A}\mathbf{x}=\mathbf{b}}} f(\mathbf{x})$$

Reduced Gradient

Wolfe (1962, 1967)

$$\operatorname*{argmax}_{\mathbf{x}} f(\mathbf{x})$$

$$\operatorname*{argmax}_{\substack{\mathbf{x}\\ \mathbf{x}\geq 0\\ \mathbf{A}\mathbf{x}=\mathbf{b}}} f(\mathbf{x})$$

$$\operatorname*{argmax}_{\substack{\mathbf{x}\\ \mathbf{x}\geq 0\\ \sum \mathbf{x}=1}} f(\mathbf{x})$$

Reduced Gradient

Wolfe (1962, 1967)

$$\operatorname*{argmax}_{\mathbf{x}} f(\mathbf{x})$$

$$\operatorname*{argmax}_{\substack{\mathbf{x} \\ \mathbf{x}\geq 0 \\ \mathbf{A}\mathbf{x}=\mathbf{b}}} f(\mathbf{x})$$

$$\operatorname*{argmax}_{\substack{\mathbf{x} \\ \mathbf{x}\geq 0 \\ \sum \mathbf{x}=1}} f(\mathbf{x})$$

```
(argmax f x0)
```

A Unified Computational Framework

Comprehensive Stochastic Cognitive Models

- cost functions over observable and hidden variables:

A Unified Computational Framework

Comprehensive Stochastic Cognitive Models

- cost functions over observable and hidden variables:
  images, sensor data, robot control parameters, representations of game and assembly states, game rules, strategies, policies, assembly plans, utterances, parse trees, word, phrase, and utterance meanings

A Unified Computational Framework

Comprehensive Stochastic Cognitive Models

- cost functions over observable and hidden variables:
  images, sensor data, robot control parameters, representations of game and assembly states, game rules, strategies, policies, assembly plans, utterances, parse trees, word, phrase, and utterance meanings
- terms encode the relationships between different variables:

A Unified Computational Framework

Comprehensive Stochastic Cognitive Models

- cost functions over observable and hidden variables:
  images, sensor data, robot control parameters, representations of game and assembly states, game rules, strategies, policies, assembly plans, utterances, parse trees, word, phrase, and utterance meanings
- terms encode the relationships between different variables:
  compositional semantics, visual perception, motor control

A Unified Computational Framework

Comprehensive Stochastic Cognitive Models

- cost functions over observable and hidden variables:
  images, sensor data, robot control parameters, representations of game and assembly states, game rules, strategies, policies, assembly plans, utterances, parse trees, word, phrase, and utterance meanings
- terms encode the relationships between different variables:
  compositional semantics, visual perception, motor control
- taking different variables to be dependent vs. independent and deciding which to marginalize and which to optimize (i.e., perform maximum-likelihood estimation), allows the same cost function and stochastic model to be used for a variety of different purposes

- By taking images as input, marginalizing over game states, and optimizing over rule representations, we can learn game rules from observed game play.

A Unified Computational Framework

- By taking images as input, marginalizing over game states, and optimizing over rule representations, we can learn game rules from observed game play.
- By taking images, game rules, and word meanings as input, marginalizing over game states and parse trees, and optimizing over utterance likelihood, we can produce natural-language descriptions of game play and learned game rules.

A Unified Computational Framework

Comprehensive Stochastic Cognitive Models

- By taking images as input, marginalizing over game states, and optimizing over rule representations, we can learn game rules from observed game play.
- By taking images, game rules, and word meanings as input, marginalizing over game states and parse trees, and optimizing over utterance likelihood, we can produce natural-language descriptions of game play and learned game rules.
- By taking utterances and word meanings as input, marginalizing over game rules, and optimizing over robotic control sequences we can teach a robot game rules or command robotic game play with linguistic input.

```
(define (lexicon game-state)
 (let ((things
        (append
         (map-n (lambda (position) (list 'position position)) 9)
         (map-n (lambda (position)
                 (list 'position-state
                       position
                       (list-ref game-state position)))
                9))))
  (list
   (cons 'the ⟨meaning for the⟩)
   (cons 'x ⟨meaning for x⟩)
   (cons 'is-on ⟨meaning for is on⟩)
   (cons 'center ⟨meaning for center⟩))))
```

A Unified Computational Framework

Comprehensive Stochastic Cognitive Models

```
(define (draw distribution)
 (let loop ((p 1)
            (pairs
             (remove-if
              (lambda (pair) (zero? (cdr pair)))
             distribution)))
  (if (or (null? (rest pairs))
          (flip (/ (cdr (first pairs)) p)))
      (car (first pairs))
      (loop (- p (cdr (first pairs)))
            (rest pairs)))))
```

```
(define (position-state-draw distribution)
 (draw (map cons
            '(empty x o)
            distribution)))
```

A Unified Computational Framework

Comprehensive Stochastic Cognitive Models

```
(define (word-draw distribution)
 (draw (map cons
            '(the x is-on center)
            distribution)))
```

```
(define (interpret words game-state)
 (if (= (length words) 1)
     (cdr (assq (first words) (lexicon game-state)))
     (let* ((i (+ (draw
                   (map
                    cons
                    (enumerate (- (length words) 1))
                    (uniform (- (length words) 1))))
                  1))
            (left (interpret
                   (sublist words 0 i)
                   game-state))
            (right (interpret
                    (sublist words i (length words))
                    game-state)))
      (if (flip 0.5) (left right) (right left)))))
```

```
(argmax
 (lambda (distributions)
  (probability
   (interpret
    (map word-draw distributions)
    '(empty empty empty
      empty x     empty
      empty empty empty))))
 (map-n (lambda (i) (uniform 4)) 5))
```

```
(argmax
 (lambda (distributions)
  (probability
   (interpret
    (map word-draw distributions)
    '(empty empty empty
      empty x     empty
      empty empty empty))))
 (map-n (lambda (i) (uniform 4)) 5))

 #(#(0 41 0 59)              ;x/center
   #(100 0 0 0)              ;the
   #(0 0 100 0)              ;is-on
   #(100 0 0 0)              ;the
   #(0 41 0 59))             ;x/center
```

```
(argmax
 (lambda (distributions)
  (probability
   (interpret
    '(the x is-on the center)
    (map position-state-draw distributions))))
 (map-n (lambda (i) (uniform 3)) 9))
```

```
(argmax
 (lambda (distributions)
  (probability
   (interpret
    '(the x is-on the center)
    (map position-state-draw distributions))))
 (map-n (lambda (i) (uniform 3)) 9))
```

```
#(#(67 0 33) #(67 0 33) #(67 0 33)   ;empty/o empty/o empty/o
  #(67 0 33) #(0 100 0) #(67 0 33)   ;empty/o x       empty/o
  #(67 0 33) #(67 0 33) #(67 0 33))  ;empty/o empty/o empty/o
```