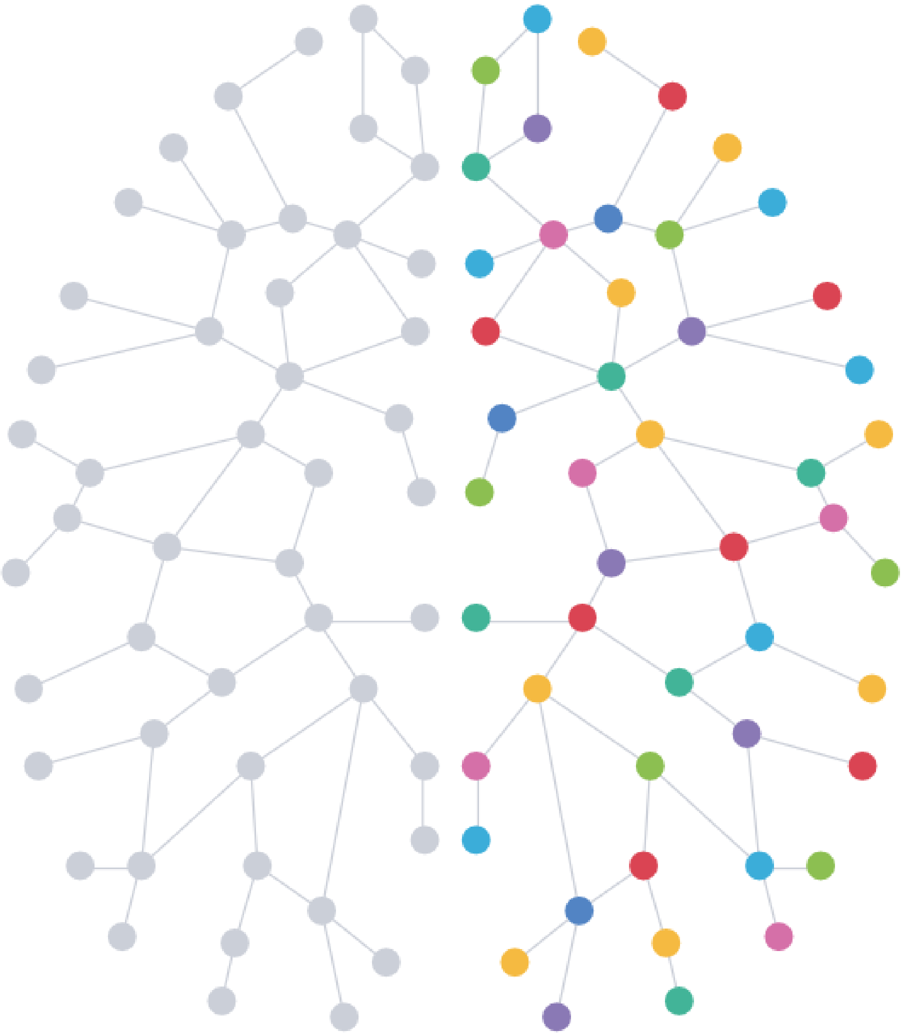


From automatic differentiation to message passing

Tom Minka

Microsoft Research

What I do

Algorithms for probabilistic inference

- Expectation Propagation
- Non-conjugate variational message passing
- A* sampling

infer.net

Probabilistic Programming

XBOX

TrueSkill

Office

Bing

Azure

Machine Learning Language

- A machine learning language should (among other things) simplify implementation of machine learning algorithms

Machine Learning Language

- A general-purpose machine learning language should (among other things) simplify implementation of all machine learning algorithms

Roadmap

1. Automatic Differentiation
2. AutoDiff lacks approximation
3. Message passing generalizes AutoDiff
4. Compiling to message passing

1. Automatic / algorithmic differentiation

Recommended reading

- “Evaluating derivatives” by Griewank and Walther (2008)

Programs are the new formulas

- Programs can specify mathematical functions more compactly than formulas
- Program is not a black box: undergoes analysis and transformation
- Numbers are assumed to have infinite precision

Multiply-all example

- As formulas:
- $f = \prod_i x_i$
- $df = \sum_i dx_i \prod_{j \neq i} x_j$

Multiply-all example

Input program

```
c[1] = x[1]
for i = 2 to n
   c[i] = c[i-1]*x[i]
f = c[n]
```

$$f = \prod_i x_i$$

Derivative program

```
dc[1] = dx[1]
for i = 2 to n
   dc[i] = dc[i-1]*x[i] + c[i-1]*dx[i]
df = dc[n]
```

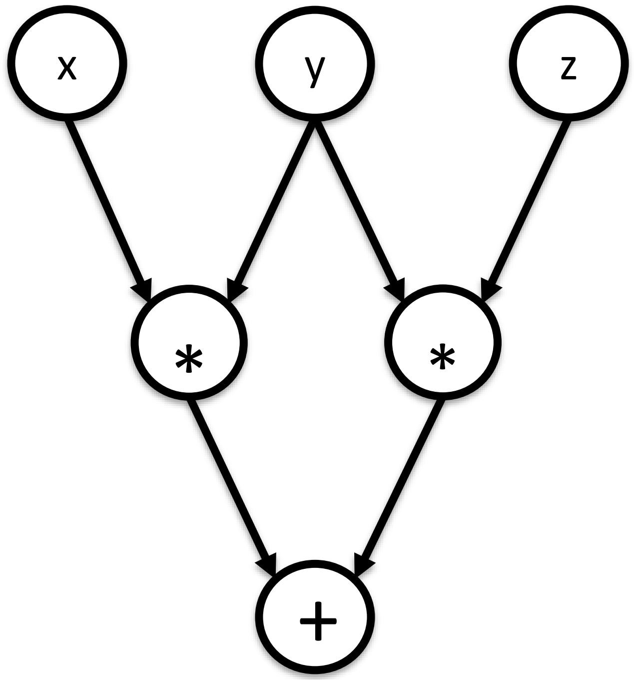
$$df = \sum_i dx_i \prod_{j \neq i} x_j$$

Phases of AD

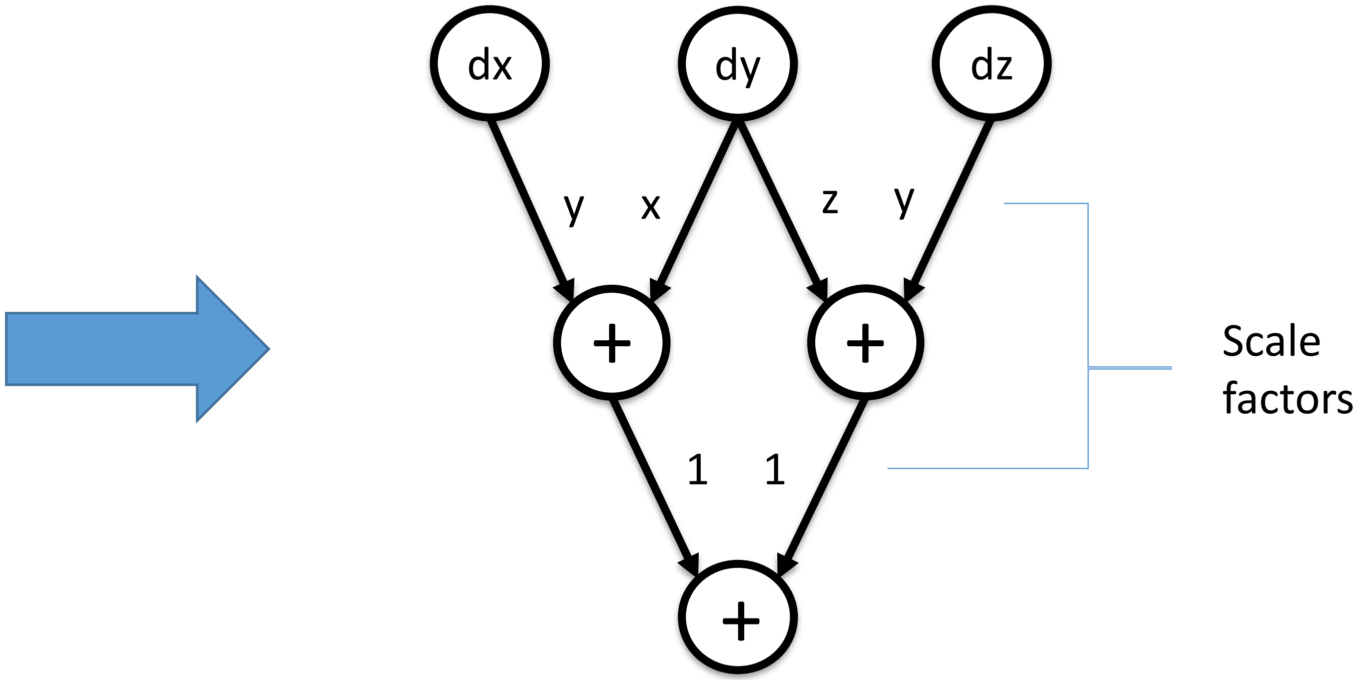
- Execution
  - Replace every operation with a linear one
- Accumulation
  - Collect linear coefficients

Execution phase

$x*y + y*z$

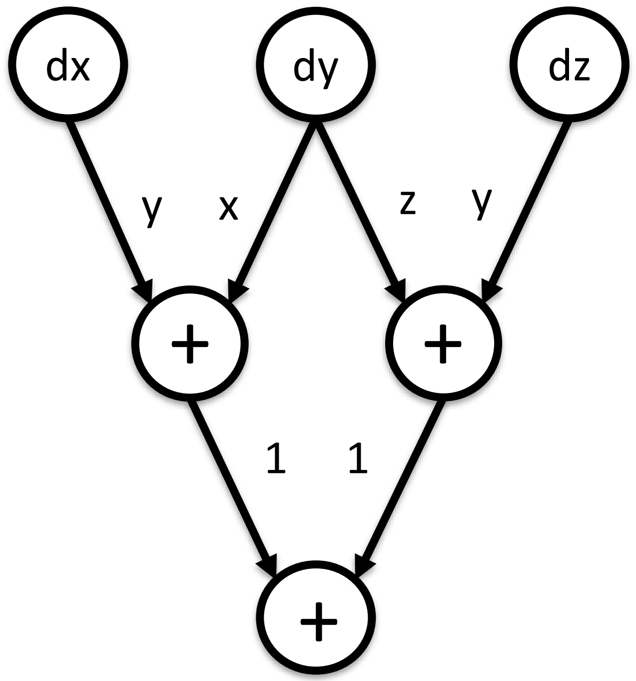


$dx*y + x*dy + dy*z + y*dz$



Accumulation phase

dx*y + x*dy + dy*z + y*dz (Forward)



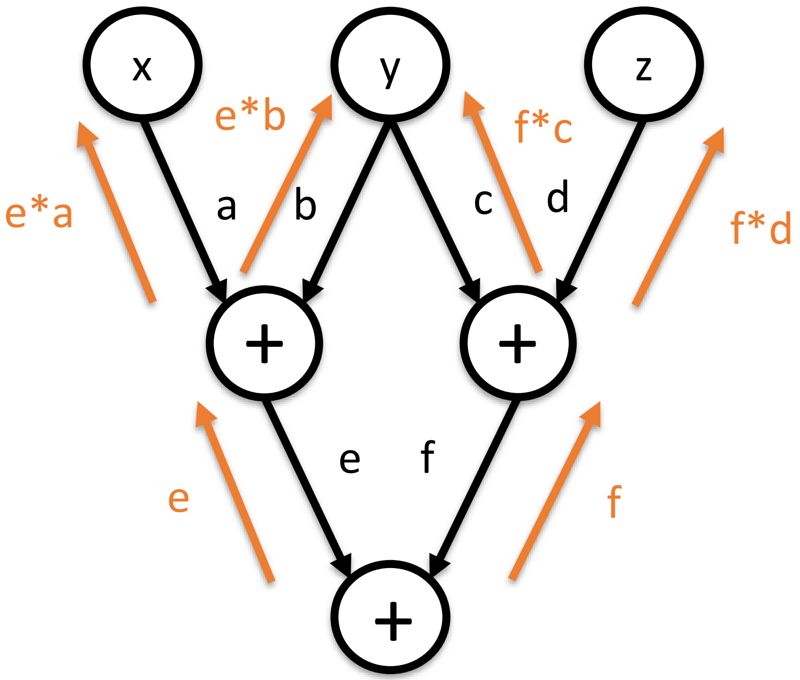
dx = 1*y

dy = 1*x + 1*z (Reverse)

dz = 1*y

Linear composition

e*(a*x + b*y) + f*(c*y + d*z)



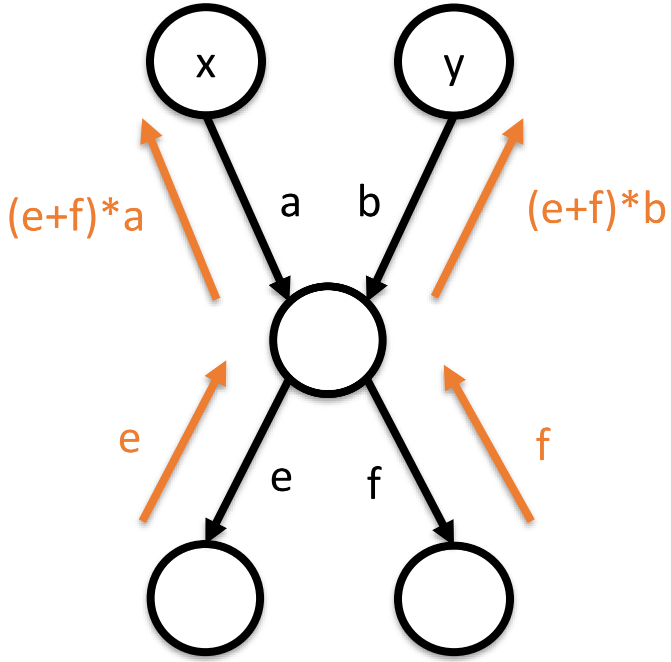
e*a*x +

(e*b + f*c)*y +

f*d*z

Dynamic programming

- Reverse accumulation is dynamic programming
- Backward message is sum over paths to output



Source-to-source translation

- Tracing approach builds a graph during execution phase, then accumulates it
- Source-to-source produces a gradient program matching structure of original

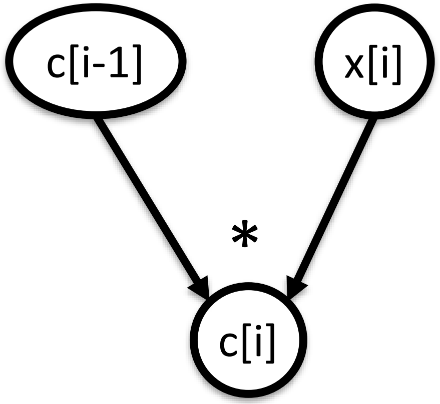
Multiply-all example

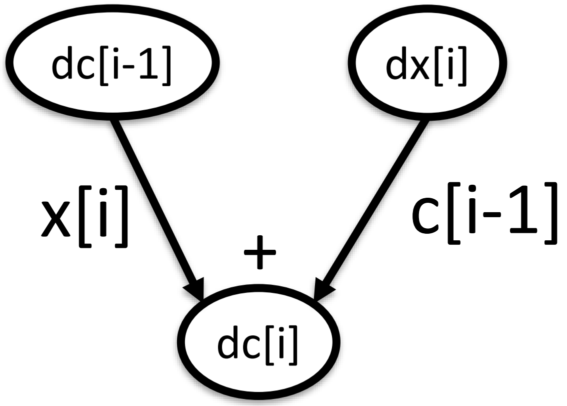
Input program

```
c[1] = x[1]
for i = 2 to n
   c[i] = c[i-1]*x[i]
return c[n]
```

Derivative program

```
dc[1] = dx[1]
for i = 2 to n
   dc[i] = dc[i-1]*x[i] + c[i-1]*dx[i]
return dc[n]
```

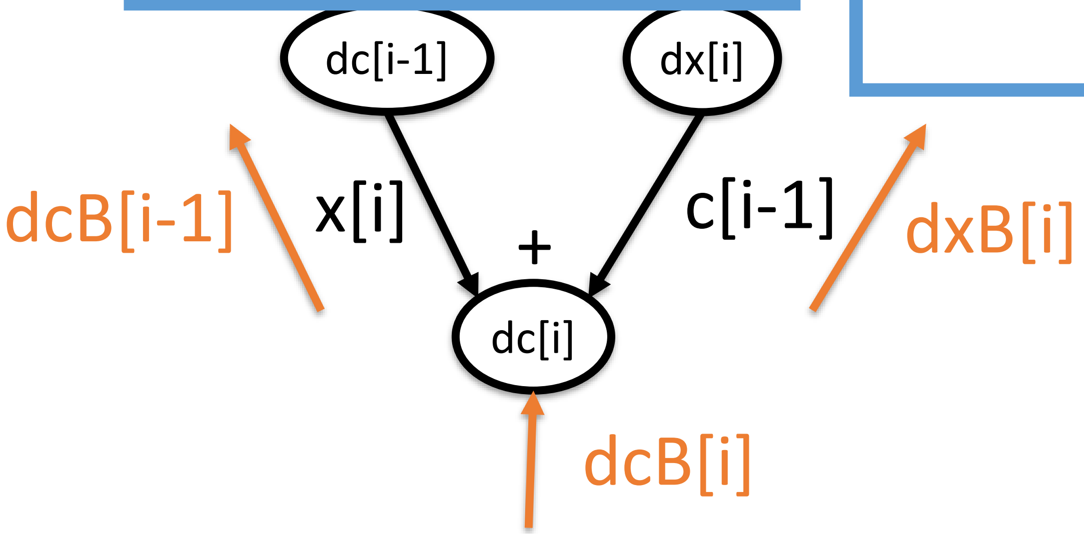




Multiply-all example

Derivative program

```
dc[1] = dx[1]
for i = 2 to n
   dc[i] = dc[i-1]*x[i] + c[i-1]*dx[i]
return dc[n]
```



Gradient program

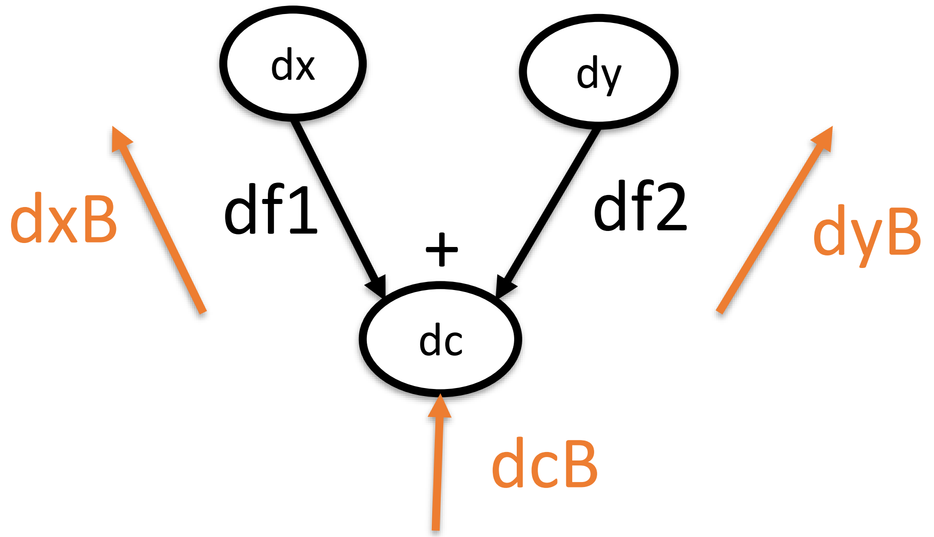
```
dcB[n] = 1
for i = n downto 2
   dcB[i-1] = dcB[i]*x[i]
   dxB[i] = dcB[i]*c[i-1]
dxB[1] = dcB[1]
return dxB
```

General case

c = f(x,y)

dc = df1(x,y) * dx + df2(x,y) * dy

dxB = dcB * df1(x,y)
dyB = dcB * df2(x,y)



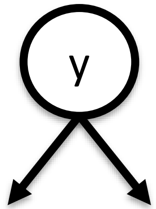
Fanout

- If a variable is read multiple times, we need to add its backward messages
- Non-incremental approach: transform program so that each variable is defined and used at most once on every execution path

Fanout example

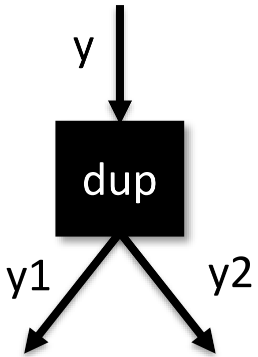
Input program

a = x * y
b = y * z
c = a + b



Edge program

(y1,y2) = dup(y)
a = x * y1
b = y2 * z
c = a + b



Gradient program

aB = cB
bB = cB
y2B = bB * z
y1B = aB * x
yB = y1B + y2B
…

Summary of AutoDiff

	AD	Message passing
Programs not formulas	Yes	Yes
Graph structure / sparsity	Yes	Yes
Source-to-source	Yes	Yes
Only one execution path	Yes	Not always
Single forward-backward sweep	Yes	Not always
Exact	Yes	Not always

2. AutoDiff lacks approximation

Approximate gradients for big models

- Mini-batching
- User changes input program to be approximate, then computes exact gradient

$$\nabla \sum_{i=1}^{n} f_i(\theta) \approx$$

$$\nabla \frac{n}{m} \sum_{s \sim (1:n)} f_s(\theta) =$$

$$\frac{n}{m} \sum_{s \sim (1:n)} \nabla f_s(\theta) \qquad \text{(AutoDiff)}$$

Black-box variational inference

1. Approximate the marginal log-likelihood with a lower bound

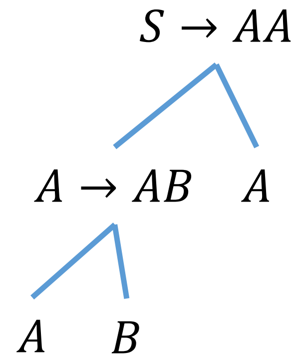
$$\int p(x, D)dx \geq -KL(q \,||\, p)$$

2. Approximate the lower bound by importance sampling

3. Compute exact gradient of approximation

AutoDiff in Tractable Models

- AutoDiff can mechanically derive reverse summation algorithms for tractable models
  - Markov chains, Bayesian networks (Darwiche, 2003)
  - Generative grammars, Parse trees (Eisner, 2016)



- Posterior expectations are derivatives of marginal log-likelihood, which can be computed exactly
  - User must provide forward summation algorithm

Approximation in Tractable Models

- Approximation is useful in tractable models
  - Sparse forward-backward (Pal et al, 2006)
  - Beam parsing (Goodman, 1997)
- Cannot be obtained through AutoDiff of an approximate model
- Neither can Viterbi

MLL should facilitate approximations

- Expectations
- Fixed-point iteration
    - Optimization
    - Root finding
- Should all be natively supported

3. Message-passing generalizes autodiff

Message-passing

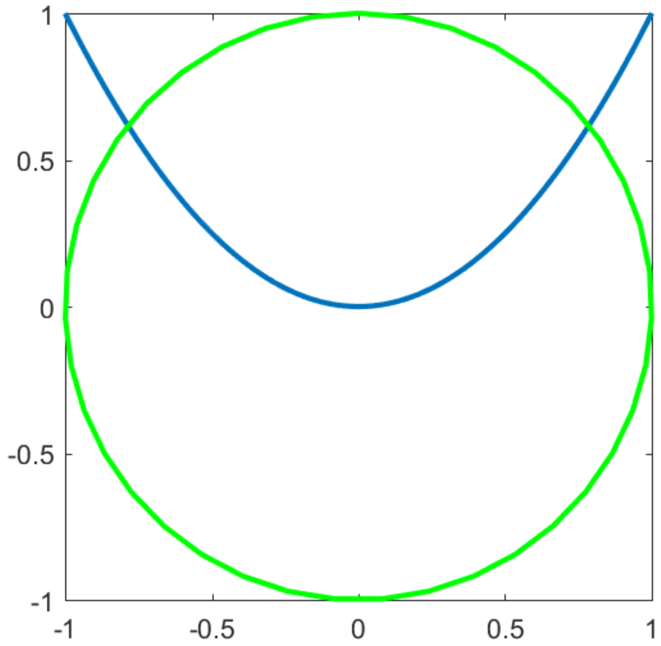
- Approximate reasoning about exponential state space of a program, along all execution paths
- Propagates state summaries in both directions
- Forward can depend on backward and vice versa
- Iterate to convergence

Interval constraint propagation

- What is largest and smallest value each variable could have?
- Each operation in program is interpreted as a constraint between inputs and output
- Propagates information forward and backward until convergence

Circle-parabola example

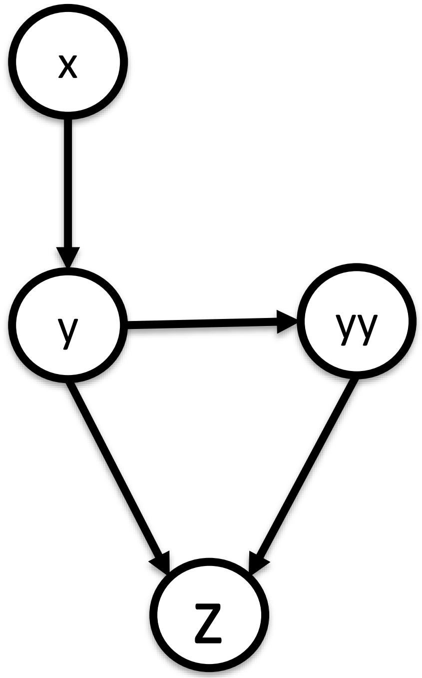
Find (x, y) that satisfies $x^2 + y^2 = 1$ and $y = x^2$



Circle-parabola program

Input program

```
y = x^2
yy = y^2
z = y + yy
assert(z == 1)
```



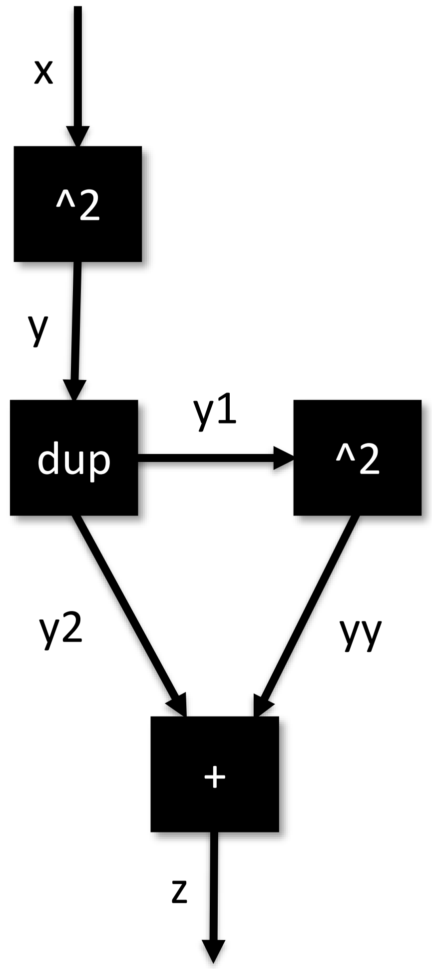
Interval propagation program

Input program

```
y = x^2

yy = y^2
z = y + yy
assert(z == 1)
```

Edge program

```
y = x^2
(y1,y2) = dup(y)
yy = y1^2
z = y2 + yy
assert(z == 1)
```



Interval propagation program

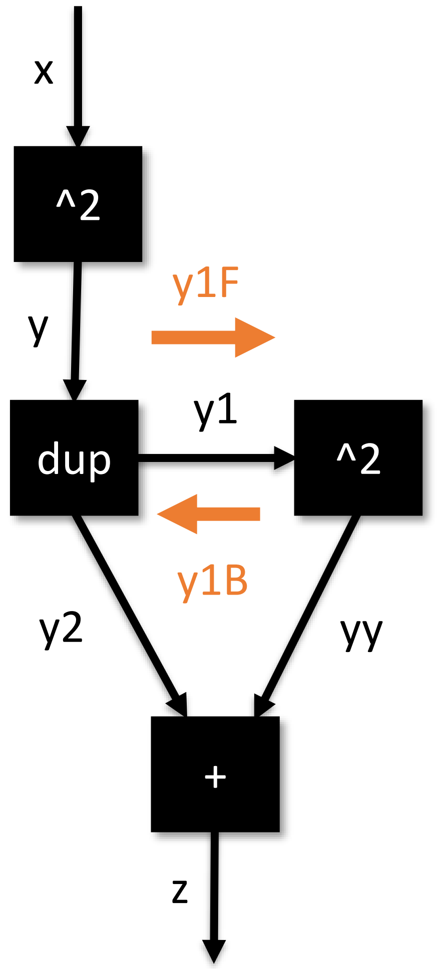
Edge program

```
y = x^2
(y1,y2) = dup(y)

yy = y1^2

z = y2 + yy

assert(z == 1)
```

Message program

Until convergence:

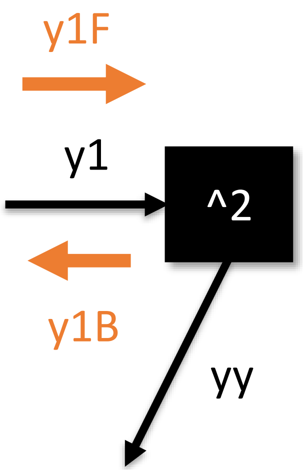
```
yF = xF^2
y1F = yF ∩ y2B
y2F = yF ∩ y1B
yyF = y1F^2
y1B = sqrt(y1F, yyB)
y2B = zB – yyF
yyB = zB – y2F
zB = [1,1]
```



Running ^2 backwards

$yy = y1\text{^}2 \quad \Rightarrow \quad y1B = sqrt(y1F, yyB)$

$\qquad\qquad = project[\ y1F \cap sqrt(yyB)\ ]$



$yyB = [2, 4]$

$sqrt(yyB) = [-2, -1] \cup [1, 2]$

$y1F = [0, 10]$

$y1F \cap sqrt(yyB) = [] \cup [1, 2]$

$project[\ y1F \cap sqrt(yyB)\ ] = [1, 2]$

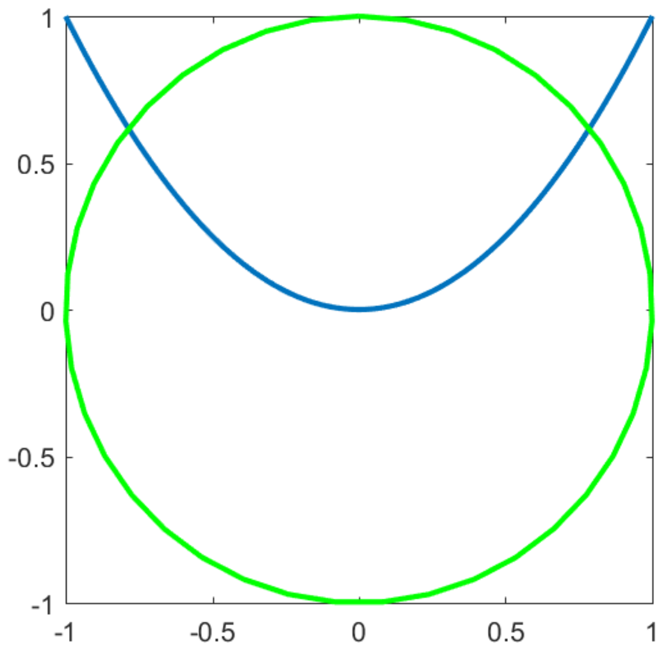
$y1F \cap project[\ sqrt(yyB)\ ] = [0, 2]$

Results

- If all intervals start $(-\infty, \infty)$ then $x \to (-1,1)$ (overestimate)

- Apply subdivision

- Starting at $x = (0.1,1)$ gives $x \to (0.786, 0.786)$



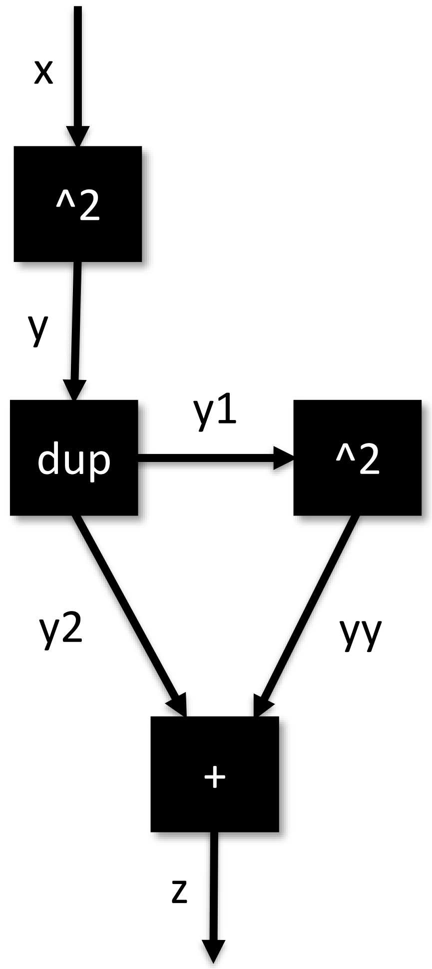
Interval propagation program

Until convergence:
  yF = xF^2
  xB = sqrt(xF, yB)
  yB = y1B ∩ y2B
  y1F = yF ∩ y2B
  y2F = yF ∩ y1B
  …
  zB = [1,1]

yF = xF^2
zB = [1,1]

Until convergence:
    (perform updates)

yB = y1B ∩ y2B
xB = sqrt(xF, yB)



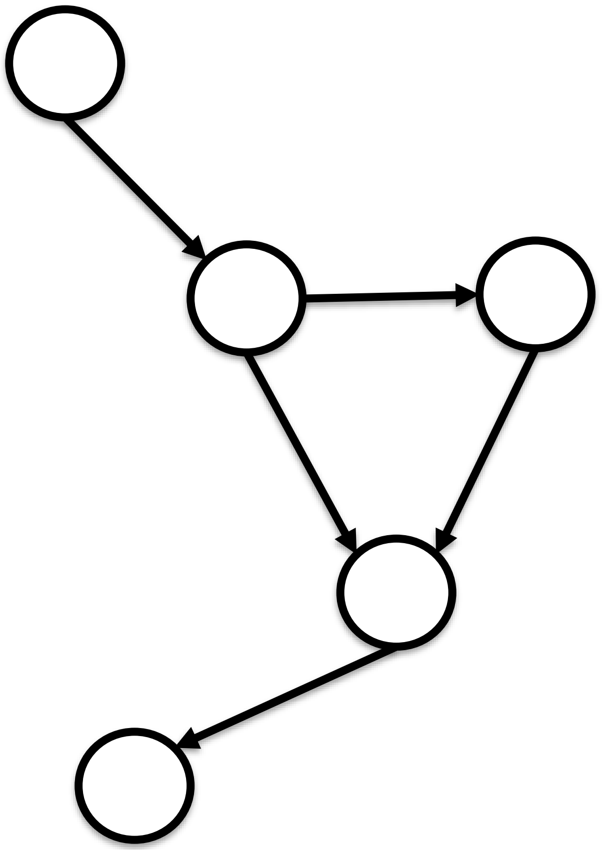
Typical message-passing program

1. Pass messages into the loopy core

2. Iterate

3. Pass messages out of the loopy core

Analogous to Stan's "transformed data" and "generated quantities"



Simplifications of message-passing

- Message dependencies dictate execution
- If forward messages do not depend on backward, becomes non-iterative
- If forward messages only include single state, only one control path is explored
- AutoDiff has both properties

Other message-passing algorithms

Probabilistic Programming

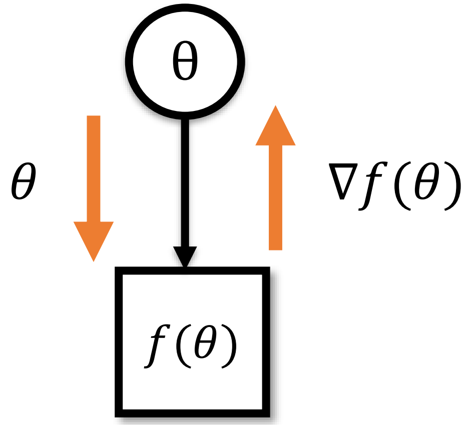
- Probabilistic programs are the new Bayesian networks
- Using a program to specify a probabilistic model
- Program is not a black box: undergoes analysis and transformation to help inference

Loopy belief propagation

- Loopy belief propagation has same structure as interval propagation, but using distributions
  - Gives forward and backward summations for tractable models
- Expectation propagation adds projection steps
  - Approximate expectations for intractable models
  - Parameter estimation in non-conjugate models

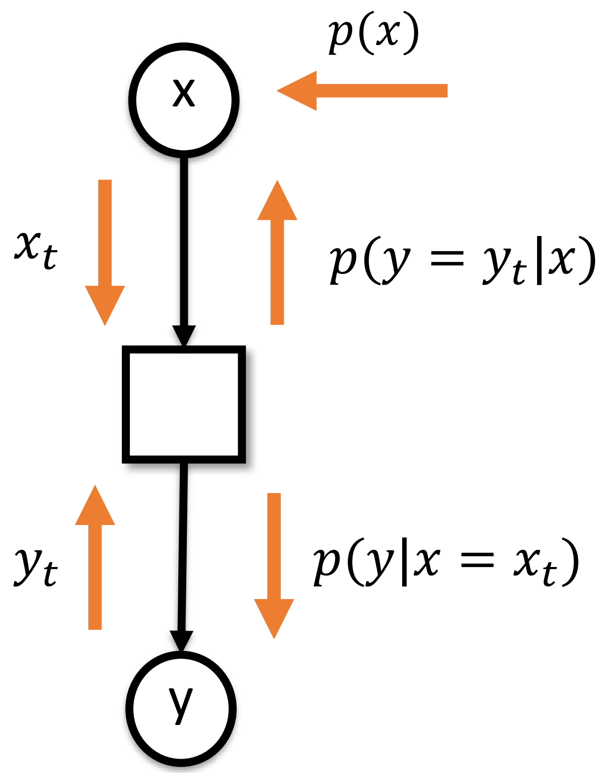
Gradient descent

- Parameters send current value out, receive gradients in, take a step
  - Gradients fall out of EP equations
- Part of the same iteration loop



Gibbs sampling

- Variables send current value out, receive conditional distributions in
- Collapsed variables send/receive distributions as in BP
  - No need to collapse in the model



Thanks!

Model-based machine learning book: http://mbmlbook.com/
Infer.NET is open source: http://dotnet.github.io/infer