EffPPL

A Probabilistic Programming Library using Effect Handlers in Multicore OCaml

Arnhav Datar

Guided by: KC Sivaramakrishnan and Tom Kelly

Agenda

- Objective
- Effect Handlers
- Probabilistic Programming
- EffPPL, Inference Algorithm
- Algorithmic Differentiation
- Application and Results
- Evaluation
- Q&A

25min

5 min

Objective - EffPPL

With the recent rise of Probabilistic Programming Languages(PPLs) in various domains, we proposed to build a library as follows.

- Concurrent
- Probabilistic Programming Library
- in Multicore OCaml
- made using Effect Handlers.


MIT News
ON CAMPUS AND AROUND THE WORLD

Graphics in reverse

Probabilistic programming does in 50 lines of code what used to take thousands.

Larry Hardesty | MIT News Office
April 13, 2015

Deep probabilistic programming (DPP) combines three fields: Bayesian statistics and machine learning, deep learning, and probabilistic programming. Many applications in trading and risk management involve uncertainty quantification such as portfolio optimization and risk estimation. DPP provides a general purpose Bayesian framework for fitting high dimensional models to large datasets to provide a richer set of statistical properties and more explanatory power.

Another benefit is that PPLs reduce the amount of manually written code for a particular inference problem, facilitating the task and minimizing the risk of inadvertently introducing errors, biases or inaccuracies. Our verification experiments suggest that the light-weight PPL implementations of ClaDS1 and ClaDS2 provide more accurate computation of likelihoods than the thousands of lines of code developed originally for these models.

Objective

With the recent rise of Probabilistic Programming Languages(PPLs) in various domains, we present EffPPL. EffPPL is a

- Shallowly-Embedded
- Probabilistic Programming Library
- in Multicore OCaml
- made using Effect Handlers.

Effect Handlers

- Effect handlers are a mechanism for modular programming with user-defined effects.

- Similar to defining new exception values, the user can define their own effects. These are handled with an effect handler.

- But unlike exceptions, effect handlers permit the control to go back to where the effect was performed, but also save the context necessary for going back in a data structure.

- Multicore OCaml incorporates effect handlers as a way of supporting concurrency primitives.

Effect handler : Example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

Effect handler : Example

effect declaration

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

Effect handler : Example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

effect declaration

computation

Effect handler : Example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

effect declaration

computation

handler

Effect handler : Example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

effect declaration

suspends current computation

computation

handler

Effect handler : Example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

effect declaration

suspends current computation

computation

delimited continuation

handler

Effect handler : Example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

effect declaration

suspends current computation

computation

delimited continuation

handler

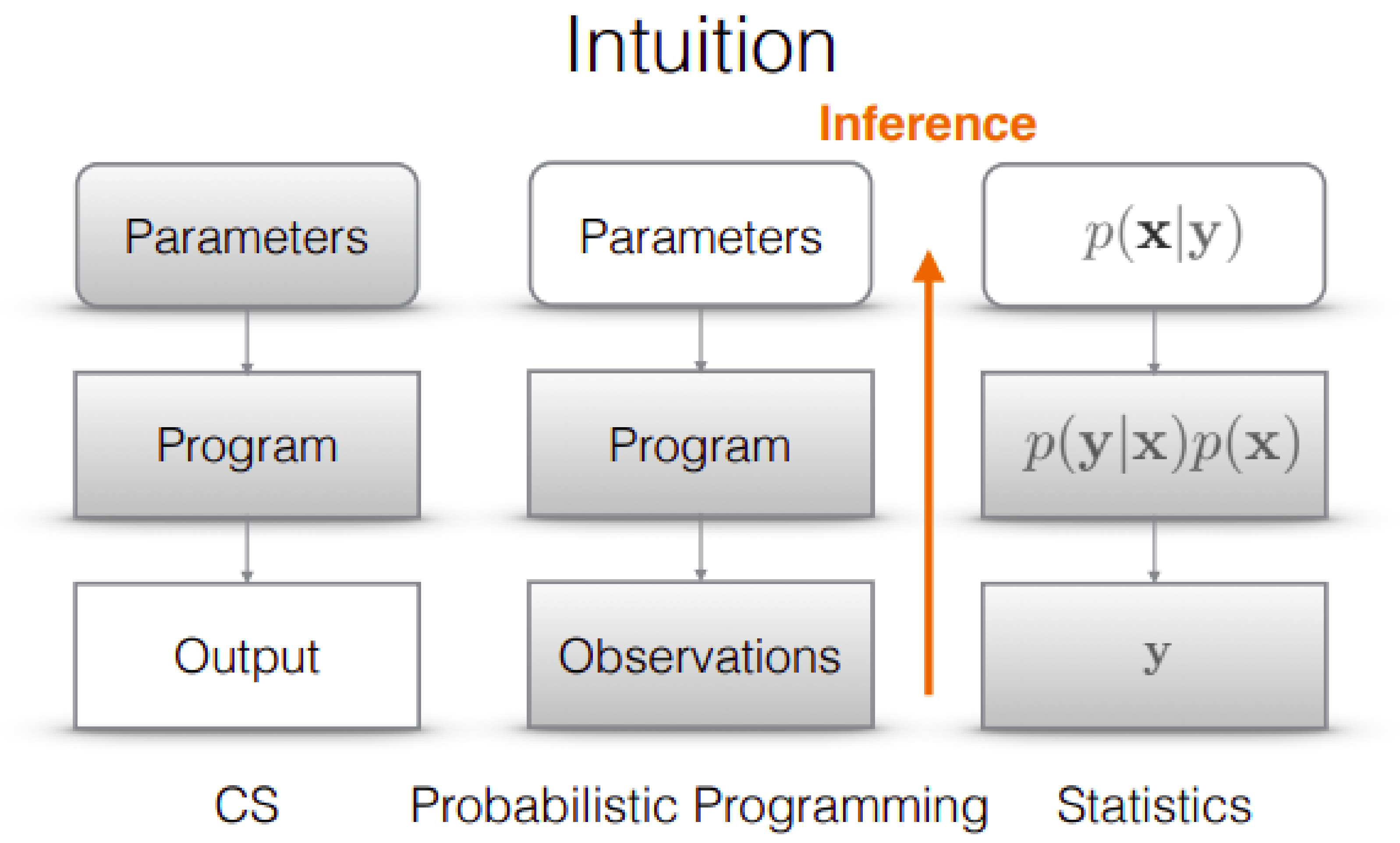
resume suspended computation

Probabilistic Programming

- Probabilistic Programming involves the construction of inference problems and the development of corresponding evaluators, that computationally characterize the denoted conditional distribution.

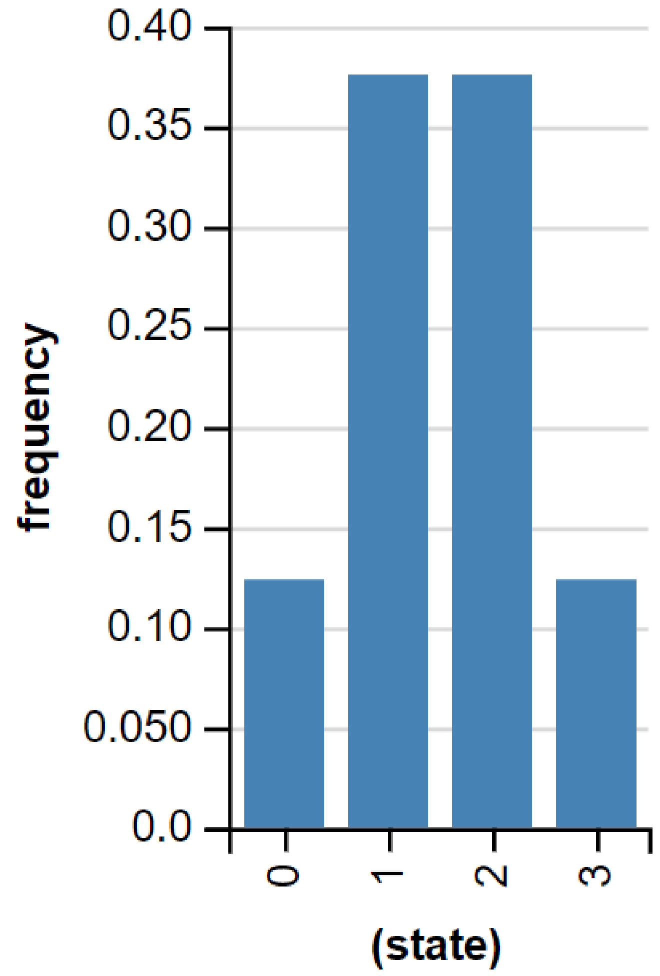
- Probabilistic programming languages(PPLs) have been used for performing approximate Bayesian inference in complex generative models. PPLs allow us to create these models using simpler models and probability distributions.

Probabilistic Programming



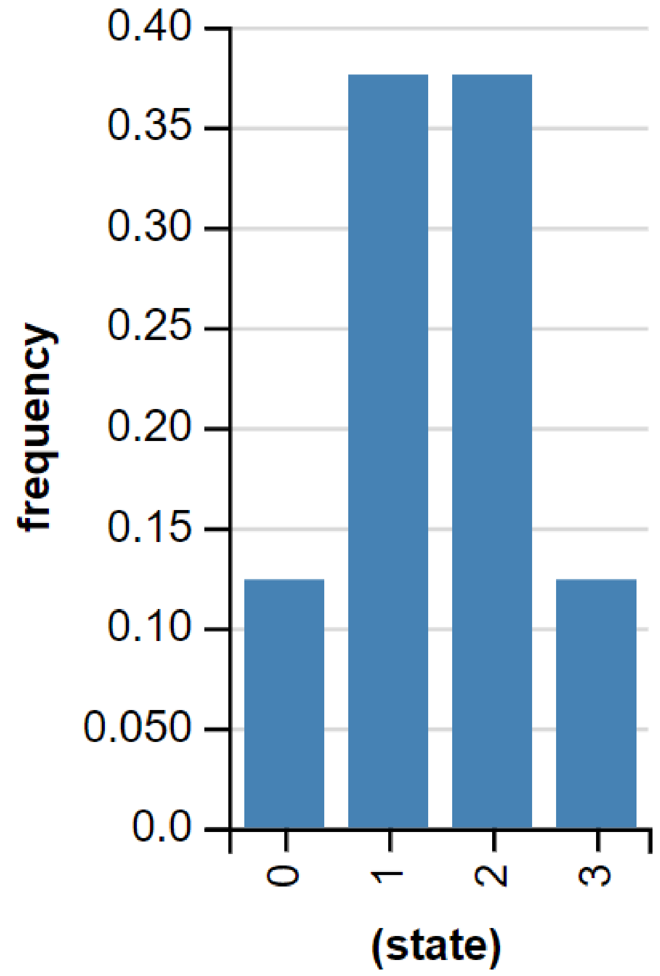
A WebPPL Example

```
var binomial = function() {
  var a = sample(Bernoulli({ p: 0.5 }))
  var b = sample(Bernoulli({ p: 0.5 }))
  var c = sample(Bernoulli({ p: 0.5 }))
  return a + b + c
}

var binomialDist = Infer({ model: binomial })

viz(binomialDist)
```



A WebPPL Example

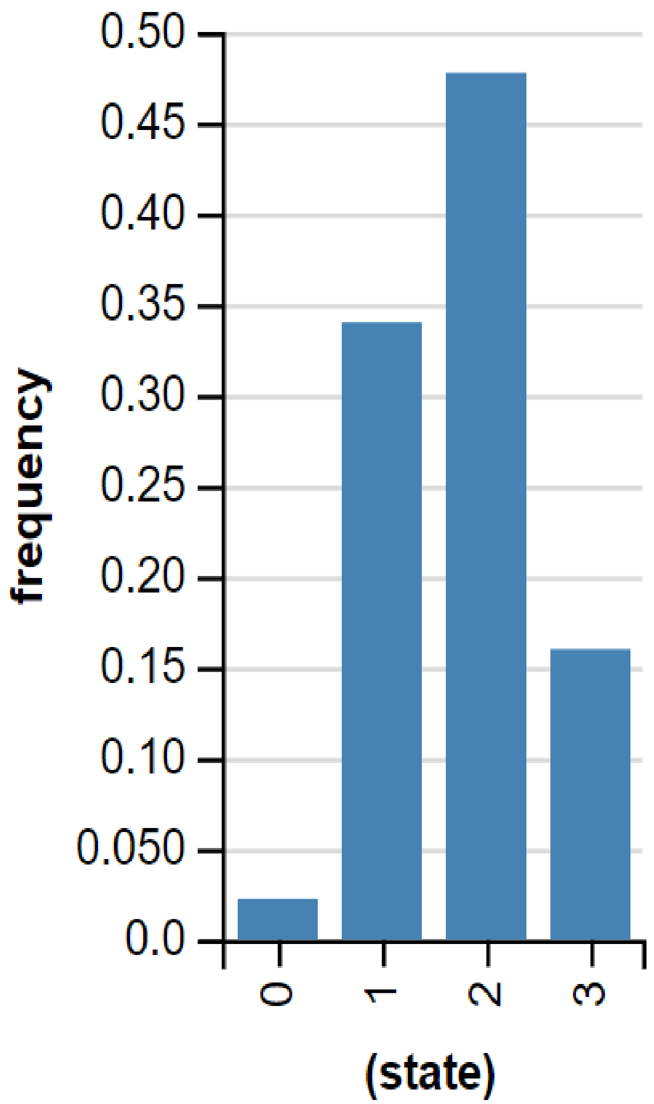
```
var binomial = function() {
  var a = sample(Bernoulli({ p: 0.5 }))
  var b = sample(Bernoulli({ p: 0.5 }))
  var c = sample(Bernoulli({ p: 0.5 }))
  return a + b + c
}

var binomialDist = Infer({ model: binomial })

viz(binomialDist)
```



H H T

A WebPPL Example

```
var funnyBinomial = function(){
  var a = sample(Bernoulli({ p: 0.5 }))
  var b = sample(Bernoulli({ p: 0.5 }))
  var c = sample(Bernoulli({ p: 0.5 }))
  factor( (a || b) ? 0 : -2)
  return a + b + c}

var funnyBinomialDist = Infer({ model: funnyBinomial })

viz(funnyBinomialDist)
```



Language Design

```
let sum_of_floats () =
  let x1 = 2. in
  let x2 = 3. in
  x1 +. x2
```

Sum of two floats in OCaml

EffPPL →

```
let sum_of_normals () =
  let* x1 = normal 0. 1. in
  let* x2 = normal 0. 1. in
  x1 +. x2
```

Sum of two standard normals

For the user to have an intuitive and easily understandable interface to create the models the language tries to mirror OCaml with the only difference being using let* as opposed to let.

The language is also shallowly embedded so all other OCaml functions and primitives such as for, if etc. work within the EffPPL functions.

Hamiltonian Monte-Carlo

Algorithm 1: Hamiltonian Monte Carlo

Given $\theta^0, L, \epsilon, \mathcal{L}, M$:

for $m = 1$ *to* M **do**

  Sample $r^0 \sim \mathcal{N}(0, I)$

  $\theta^m \leftarrow \theta^{m-1}$

  $\theta' \leftarrow \theta^{m-1}$

  $r' \leftarrow r^0$

  for $i = 1$ *to* L **do**

    $(\theta', r') \leftarrow Leapfrog(\theta', r', \epsilon)$

  end

  With probability $\min\{1, \frac{\exp(\mathcal{L}(\theta') - 0.5 \cdot r' \cdot r')}{\exp(\mathcal{L}(\theta^{m-1}) - 0.5 \cdot r^0 \cdot r^0)}\}$ set $\theta^m \leftarrow \theta'$ and $r^m \leftarrow -r'$

end

function $Leapfrog(\theta', r', \epsilon)$:

  $r' \leftarrow r' + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta)$

  $\theta' \leftarrow \theta' + (\epsilon) r'$

  $r' \leftarrow r' + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta')$

return (θ', r')

Hamiltonian Monte-Carlo

Algorithm 1: Hamiltonian Monte Carlo

Given $\boxed{\theta^0, L, \epsilon, \mathcal{L}, M:}$ → Initial sample, iterations, step size, model, epochs

for $m = 1$ *to* M **do**

    Sample $r^0 \sim \mathcal{N}(0, I)$

    $\theta^m \leftarrow \theta^{m-1}$

    $\theta' \leftarrow \theta^{m-1}$

    $r' \leftarrow r^0$

    for $i = 1$ *to* L **do**

        $(\theta', r') \leftarrow Leapfrog(\theta', r', \epsilon)$

    end

    With probability $\min\{1, \frac{\exp(\mathcal{L}(\theta') - 0.5 \cdot r' \cdot r')}{\exp(\mathcal{L}(\theta^{m-1}) - 0.5 \cdot r^0 \cdot r^0)}\}$ set $\theta^m \leftarrow \theta'$ and $r^m \leftarrow -r'$

end

function $Leapfrog(\theta', r', \epsilon)$:

    $r' \leftarrow r' + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta)$

    $\theta' \leftarrow \theta' + (\epsilon)r'$

    $r' \leftarrow r' + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta')$

return (θ', r')

Hamiltonian Monte-Carlo

Algorithm 1: Hamiltonian Monte Carlo

Given $\theta^0, L, \epsilon, \mathcal{L}, M$:

Initial sample, iterations, step size, model, epochs

for $m = 1$ *to* M **do**

Sample $r^0 \sim \mathcal{N}(0, I)$

$\theta^m \leftarrow \theta^{m-1}$

$\theta' \leftarrow \theta^{m-1}$

$r' \leftarrow r^0$

Initialising candidate momentum and position

for $i = 1$ *to* L **do**

$(\theta', r') \leftarrow Leapfrog(\theta', r', \epsilon)$

end

With probability $\min\{1, \frac{\exp(\mathcal{L}(\theta') - 0.5 \cdot r' \cdot r')}{\exp(\mathcal{L}(\theta^{m-1}) - 0.5 \cdot r^0 \cdot r^0)}\}$ set $\theta^m \leftarrow \theta'$ and $r^m \leftarrow -r'$

end

function $Leapfrog(\theta', r', \epsilon)$:

$r' \leftarrow r' + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta)$

$\theta' \leftarrow \theta' + (\epsilon)r'$

$r' \leftarrow r' + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta')$

return (θ', r')

Hamiltonian Monte-Carlo

Algorithm 1: Hamiltonian Monte Carlo

Given $\boxed{\theta^0, L, \epsilon, \mathcal{L}, M:}$

Initial sample, iterations, step size, model, epochs

for $m = 1$ *to* M **do**

- Sample $r^0 \sim \mathcal{N}(0, I)$
- $\theta^m \leftarrow \theta^{m-1}$
- $\theta' \leftarrow \theta^{m-1}$
- $r' \leftarrow r^0$

Initialising candidate momentum and position

- **for** $i = 1$ *to* L **do**
  - $(\theta', r') \leftarrow Leapfrog(\theta', r', \epsilon)$
- **end**
- With probability $\min\{1, \frac{\exp(\mathcal{L}(\theta') - 0.5 \cdot r' \cdot r')}{\exp(\mathcal{L}(\theta^{m-1}) - 0.5 \cdot r^0 \cdot r^0)}\}$ set $\theta^m \leftarrow \theta'$ and $r^m \leftarrow -r'$

end

function $Leapfrog(\theta', r', \epsilon)$:

- $r' \leftarrow r' + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta)$
- $\theta' \leftarrow \theta' + (\epsilon)r'$
- $r' \leftarrow r' + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta')$

return (θ', r')

Numerical Integrator

Hamiltonian Monte-Carlo

Algorithm 1: Hamiltonian Monte Carlo

Given $\theta^0, L, \epsilon, \mathcal{L}, M$:

Initial sample, iterations, step size, model, epochs

for $m = 1$ *to* M **do**

  Sample $r^0 \sim \mathcal{N}(0, I)$

  $\theta^m \leftarrow \theta^{m-1}$

  $\theta' \leftarrow \theta^{m-1}$

  $r' \leftarrow r^0$

Initialising candidate momentum and position

  for $i = 1$ *to* L **do**

    $(\theta', r') \leftarrow Leapfrog(\theta', r', \epsilon)$

  end

  With probability $\min\{1, \frac{\exp(\mathcal{L}(\theta') - 0.5 \cdot r' \cdot r')}{\exp(\mathcal{L}(\theta^{m-1}) - 0.5 \cdot r^0 \cdot r^0)}\}$ set $\theta^m \leftarrow \theta'$ and $r^m \leftarrow -r'$

Metropolis Acceptance

end

function $Leapfrog(\theta', r', \epsilon)$:

  $r' \leftarrow r' + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta)$

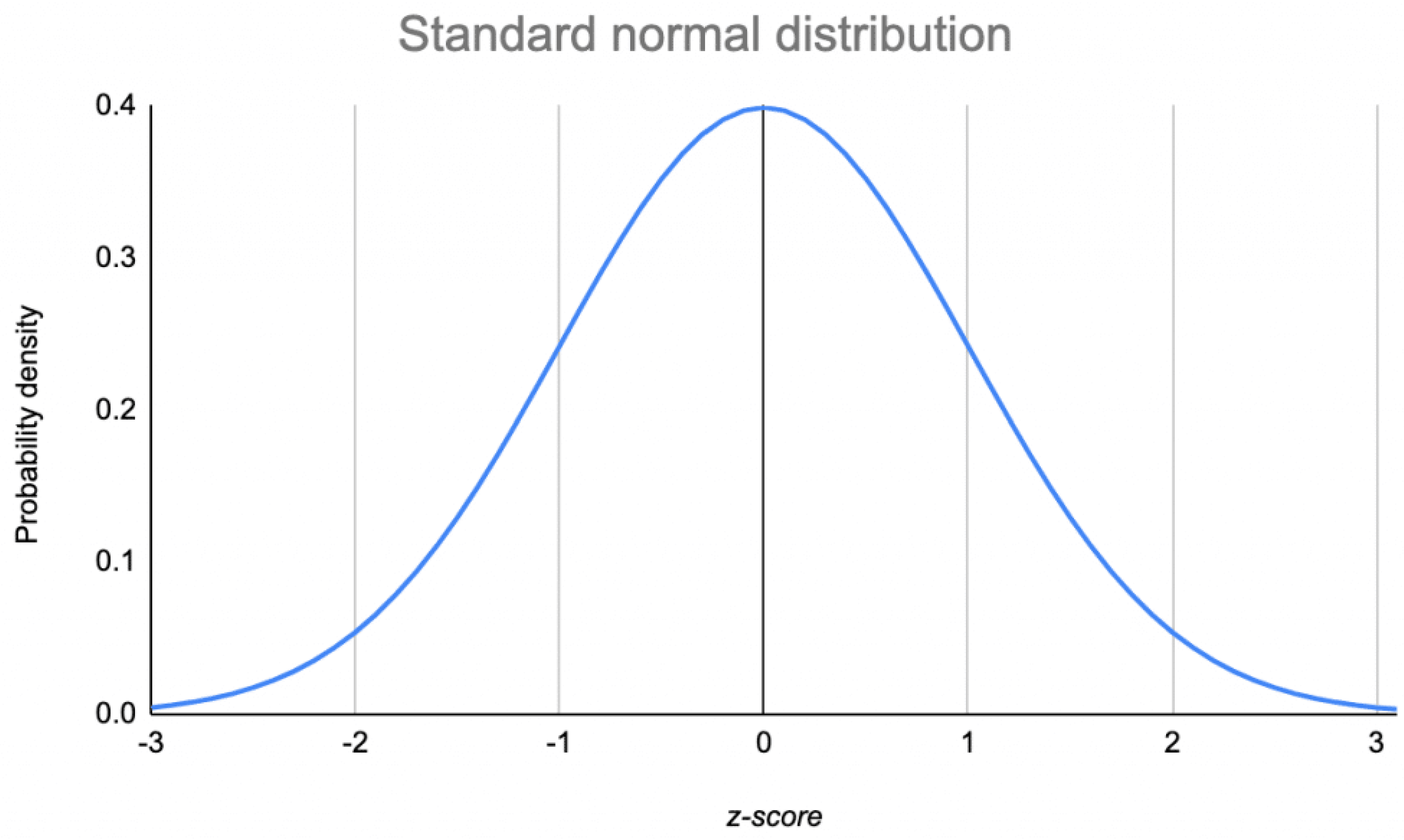
  $\theta' \leftarrow \theta' + (\epsilon) r'$

  $r' \leftarrow r' + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta')$

return (θ', r')

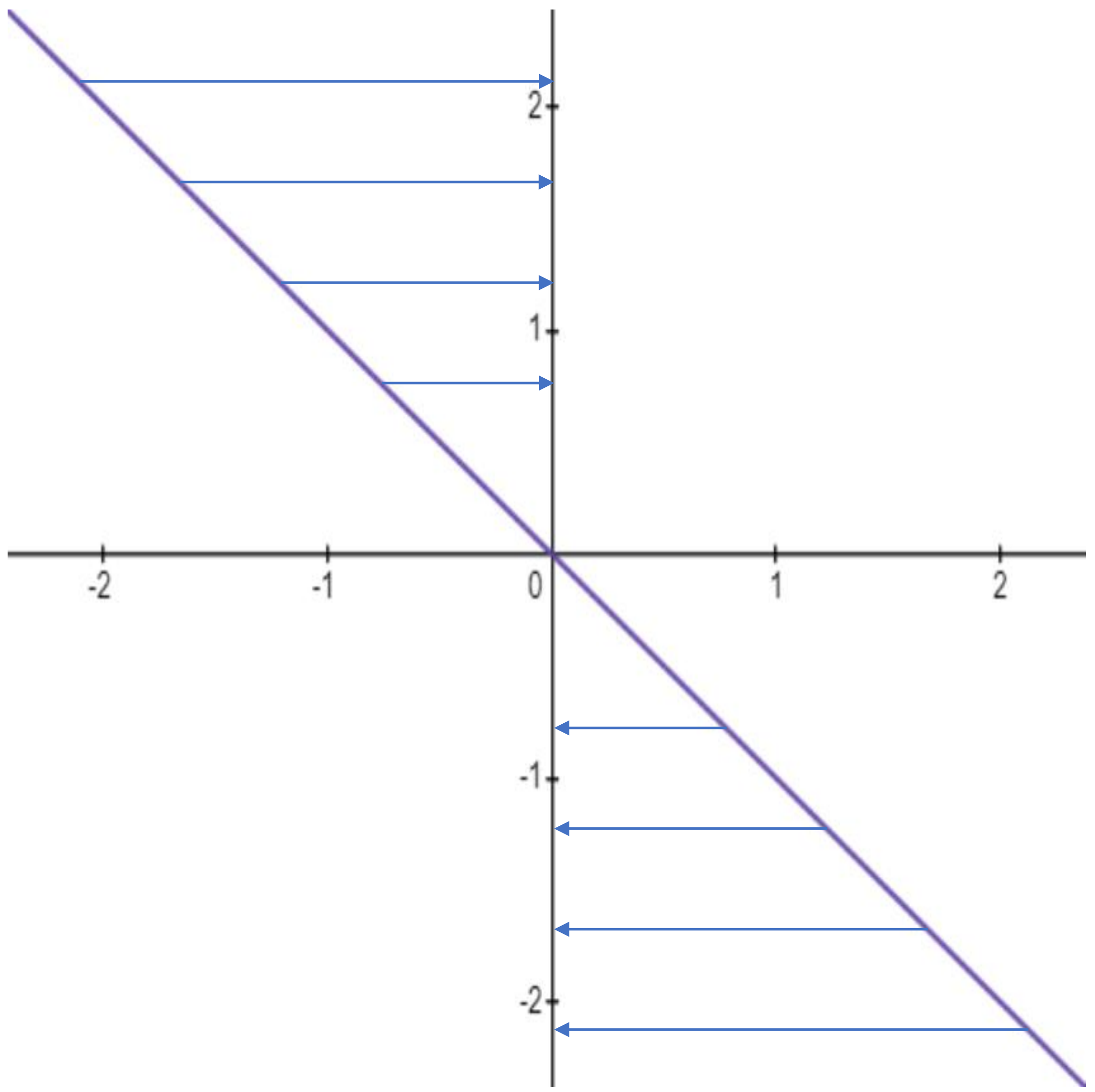
Numerical Integrator

Hamiltonian Monte-Carlo Example



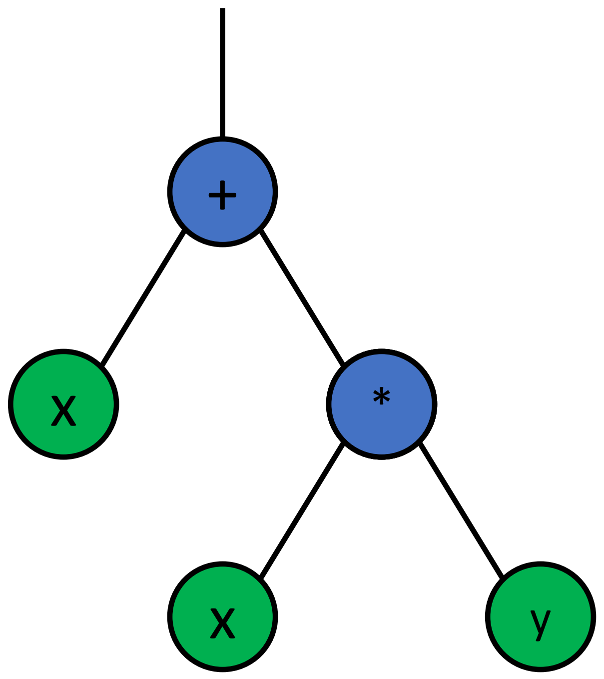
Hamiltonian Monte-Carlo Example

Derivative of LogPDF of standard normal distribution



Algorithmic Differentiation

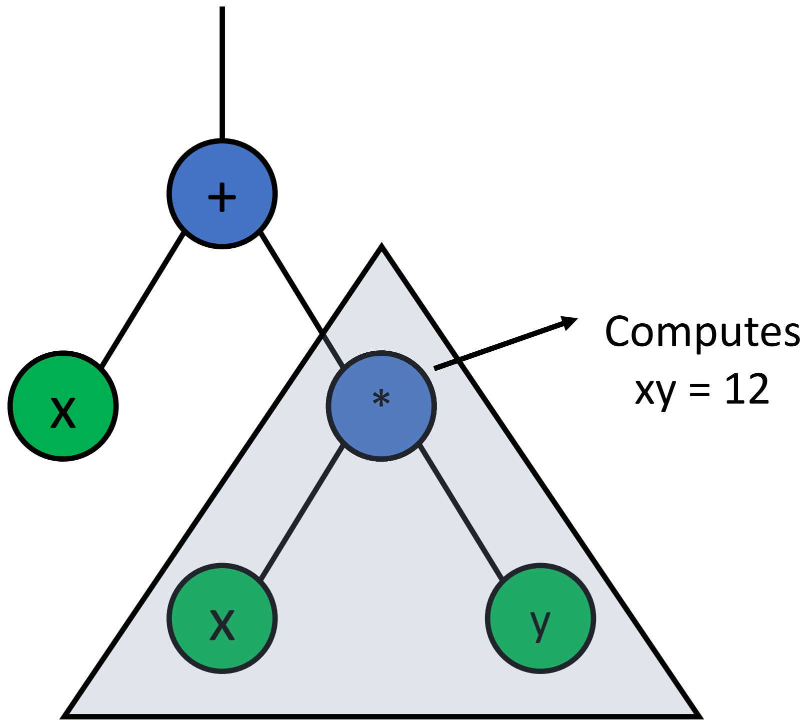
```
| r ->
  r.d <- 1.0;
  ls := modif_der !ls r.v r.d;
  (r)
| effect (Add(a,b)) k ->
  let t = {v = a.v +. b.v; d = 0.; m=1.} in
  ignore (continue k t);
  a.d <- a.d +. t.d;
  b.d <- b.d +. t.d;
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)

| effect (Mult(a,b)) k ->
  let t = {v = a.v *. b.v; d = 0.;  m=1.} in
  ignore (continue k t);
  a.d <- a.d +. (b.v *. t.d);
  b.d <- b.d +. (a.v *. t.d);
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)
```



z = x + xy
x = 3
y = 4
Task: To find partial derivatives of z with respect to x and y

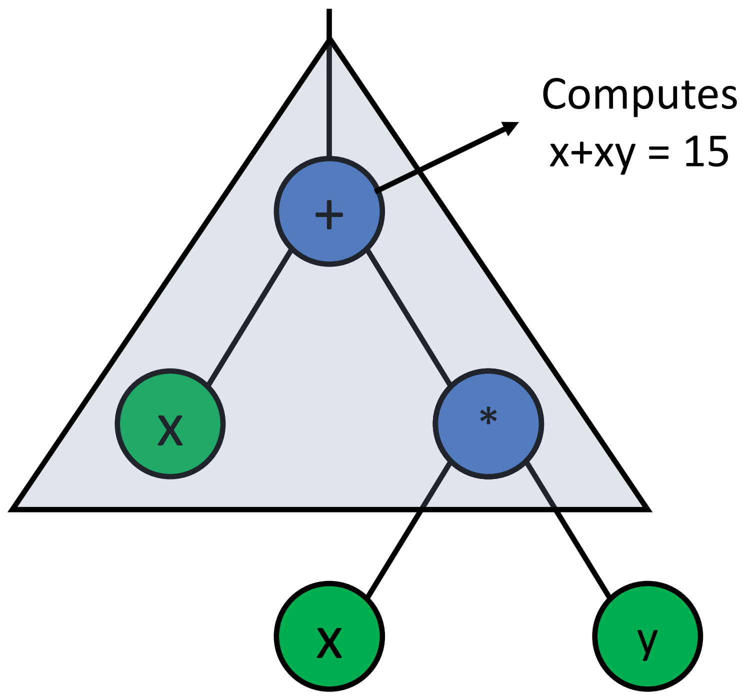
Algorithmic Differentiation

```
| r ->
  r.d <- 1.0;
  ls := modif_der !ls r.v r.d;
  (r)
| effect (Add(a,b)) k ->
  let t = {v = a.v +. b.v; d = 0.; m=1.} in
  ignore (continue k t);
  a.d <- a.d +. t.d;
  b.d <- b.d +. t.d;
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)

| effect (Mult(a,b)) k ->
  let t = {v = a.v *. b.v; d = 0.;  m=1.} in
  ignore (continue k t);
  a.d <- a.d +. (b.v *. t.d);
  b.d <- b.d +. (a.v *. t.d);
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)
```



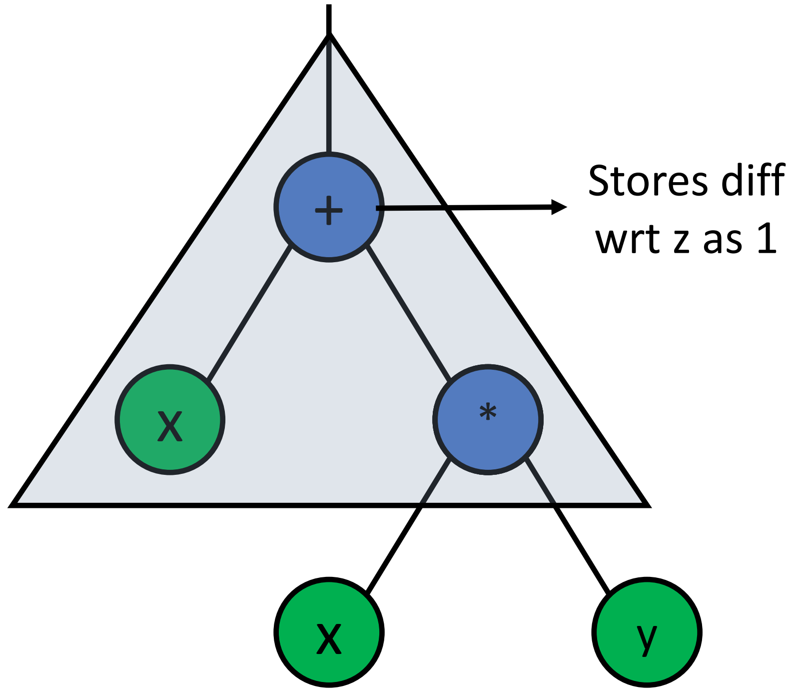
Algorithmic Differentiation

```
| r ->
  r.d <- 1.0;
  ls := modif_der !ls r.v r.d;
  (r)
| effect (Add(a,b)) k ->
  let t = {v = a.v +. b.v; d = 0.; m=1.} in
  ignore (continue k t);
  a.d <- a.d +. t.d;
  b.d <- b.d +. t.d;
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)

| effect (Mult(a,b)) k ->
  let t = {v = a.v *. b.v; d = 0.;  m=1.} in
  ignore (continue k t);
  a.d <- a.d +. (b.v *. t.d);
  b.d <- b.d +. (a.v *. t.d);
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)
```



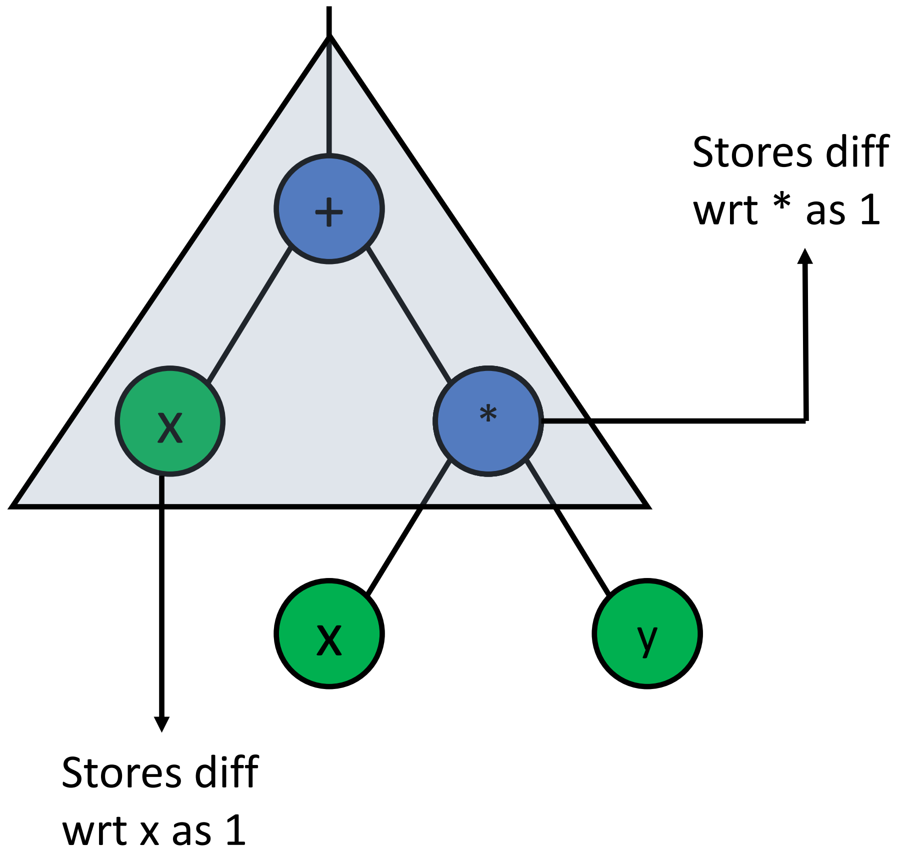
Algorithmic Differentiation

```
| r ->
  r.d <- 1.0;
  ls := modif_der !ls r.v r.d;
  (r)
| effect (Add(a,b)) k ->
  let t = {v = a.v +. b.v; d = 0.; m=1.} in
  ignore (continue k t);
  a.d <- a.d +. t.d;
  b.d <- b.d +. t.d;
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)

| effect (Mult(a,b)) k ->
  let t = {v = a.v *. b.v; d = 0.;  m=1.} in
  ignore (continue k t);
  a.d <- a.d +. (b.v *. t.d);
  b.d <- b.d +. (a.v *. t.d);
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)
```



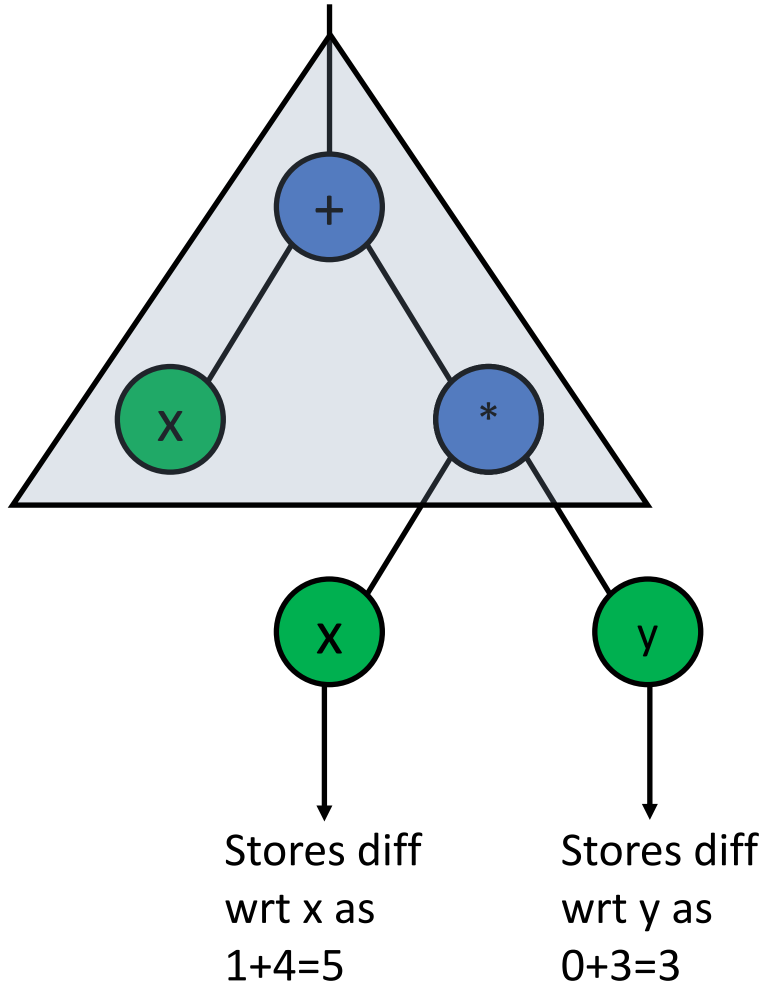
Algorithmic Differentiation

```
| r ->
  r.d <- 1.0;
  ls := modif_der !ls r.v r.d;
  (r)
| effect (Add(a,b)) k ->
  let t = {v = a.v +. b.v; d = 0.; m=1.} in
  ignore (continue k t);
  a.d <- a.d +. t.d;
  b.d <- b.d +. t.d;
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)

| effect (Mult(a,b)) k ->
  let t = {v = a.v *. b.v; d = 0.;  m=1.} in
  ignore (continue k t);
  a.d <- a.d +. (b.v *. t.d);
  b.d <- b.d +. (a.v *. t.d);
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)
```



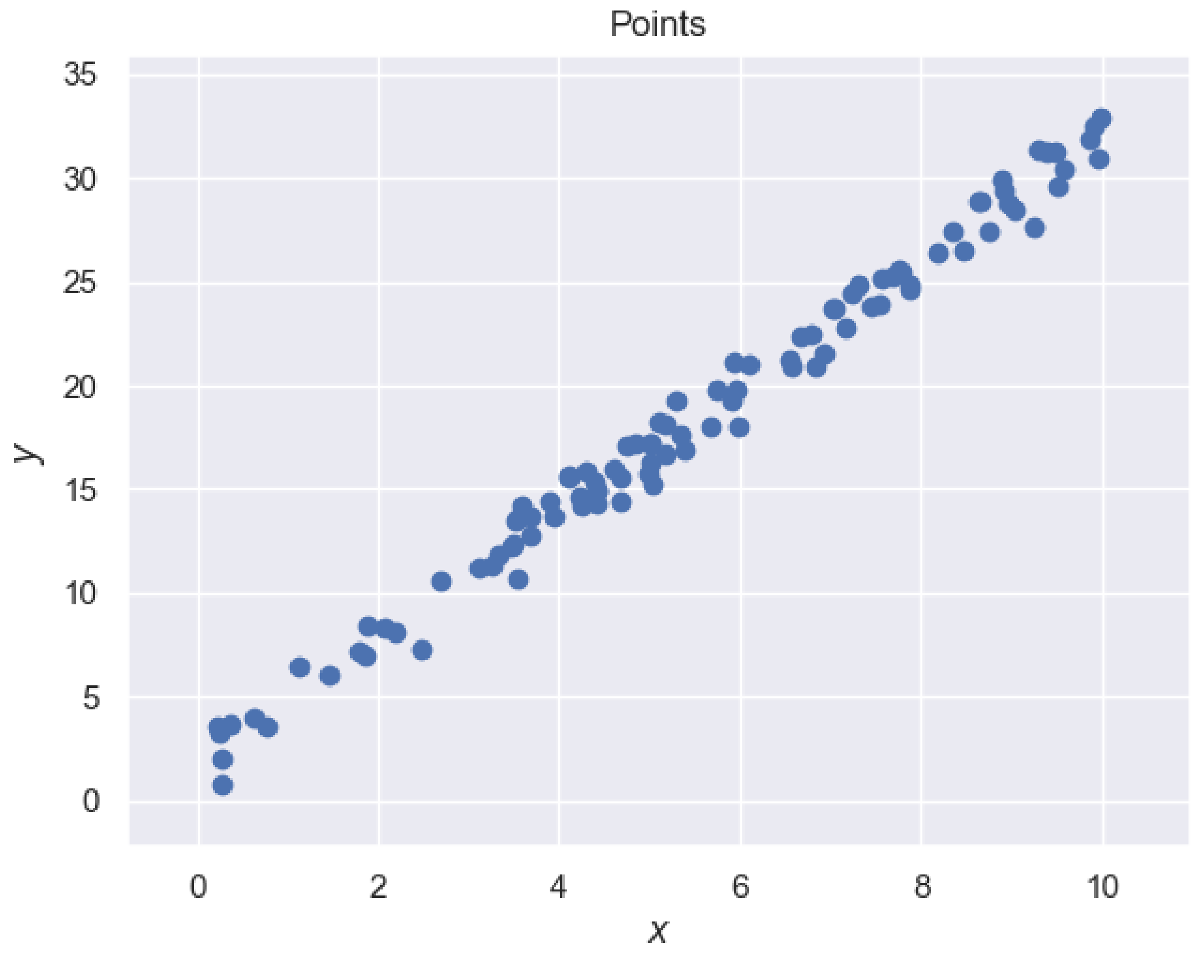
Algorithmic Differentiation

```
| r ->
  r.d <- 1.0;
  ls := modif_der !ls r.v r.d;
  (r)
| effect (Add(a,b)) k ->
  let t = {v = a.v +. b.v; d = 0.; m=1.} in
  ignore (continue k t);
  a.d <- a.d +. t.d;
  b.d <- b.d +. t.d;
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)

| effect (Mult(a,b)) k ->
  let t = {v = a.v *. b.v; d = 0.;  m=1.} in
  ignore (continue k t);
  a.d <- a.d +. (b.v *. t.d);
  b.d <- b.d +. (a.v *. t.d);
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)
```



Linear Regression problem

Given some values of an output y and some values of a dependent parameter x, assuming that they have a linear dependance find the best fitting line through the observed points



It is mathematically phrased as:

$$Y_n = normal\ (c + mX_n, \sigma)$$

Linear Regression example

```
let lin obs_points ax ay () =
  let* m = normal 2. 3. in
  let* c = normal 0. 10. in
  let* s = exp 5. in
  for i = 0 to (obs_points-1) do
    observe (mk ay.(i) -. m*.mk ax.(i) -. c)
    (logpdf Primitive.(normal 0. (get s)))
  done ;
```

Linear Regression example

```
let lin obs_points ax ay () =
  let* m = normal 2. 3. in
  let* c = normal 0. 10. in
  let* s = exp 5. in
  for i = 0 to (obs_points-1) do
    observe (mk ay.(i) -. m*.mk ax.(i) -. c)
    (logpdf Primitive.(normal 0. (get s)))
  done ;
```

Some priors that we may know

Linear Regression example

```
let lin obs_points ax ay () =
  let* m = normal 2. 3. in
  let* c = normal 0. 10. in
  let* s = exp 5. in
  for i = 0 to (obs_points-1) do
    observe (mk ay.(i) -. m*.mk ax.(i) -. c)
    (logpdf Primitive.(normal 0. (get s)))
  done ;
```

Some priors that we may know

Iterating over all observed (x_i, y_i)

Linear Regression example

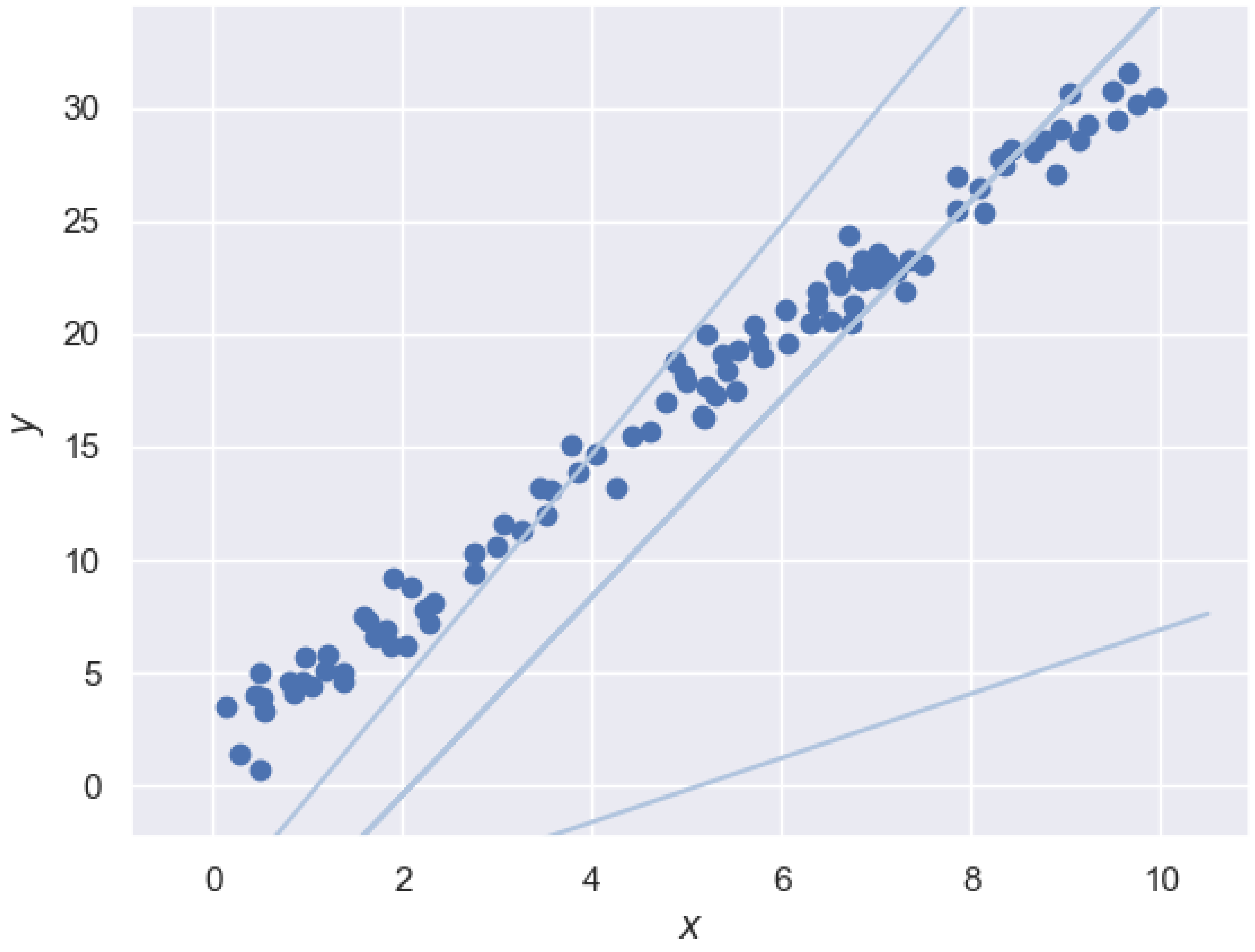
```
let lin obs_points ax ay () =
  let* m = normal 2. 3. in
  let* c = normal 0. 10. in
  let* s = exp 5. in
  for i = 0 to (obs_points-1) do
    observe (mk ay.(i) -. m*.mk ax.(i) -. c)
    (logpdf Primitive.(normal 0. (get s)))
  done ;
```

Some priors that we may know

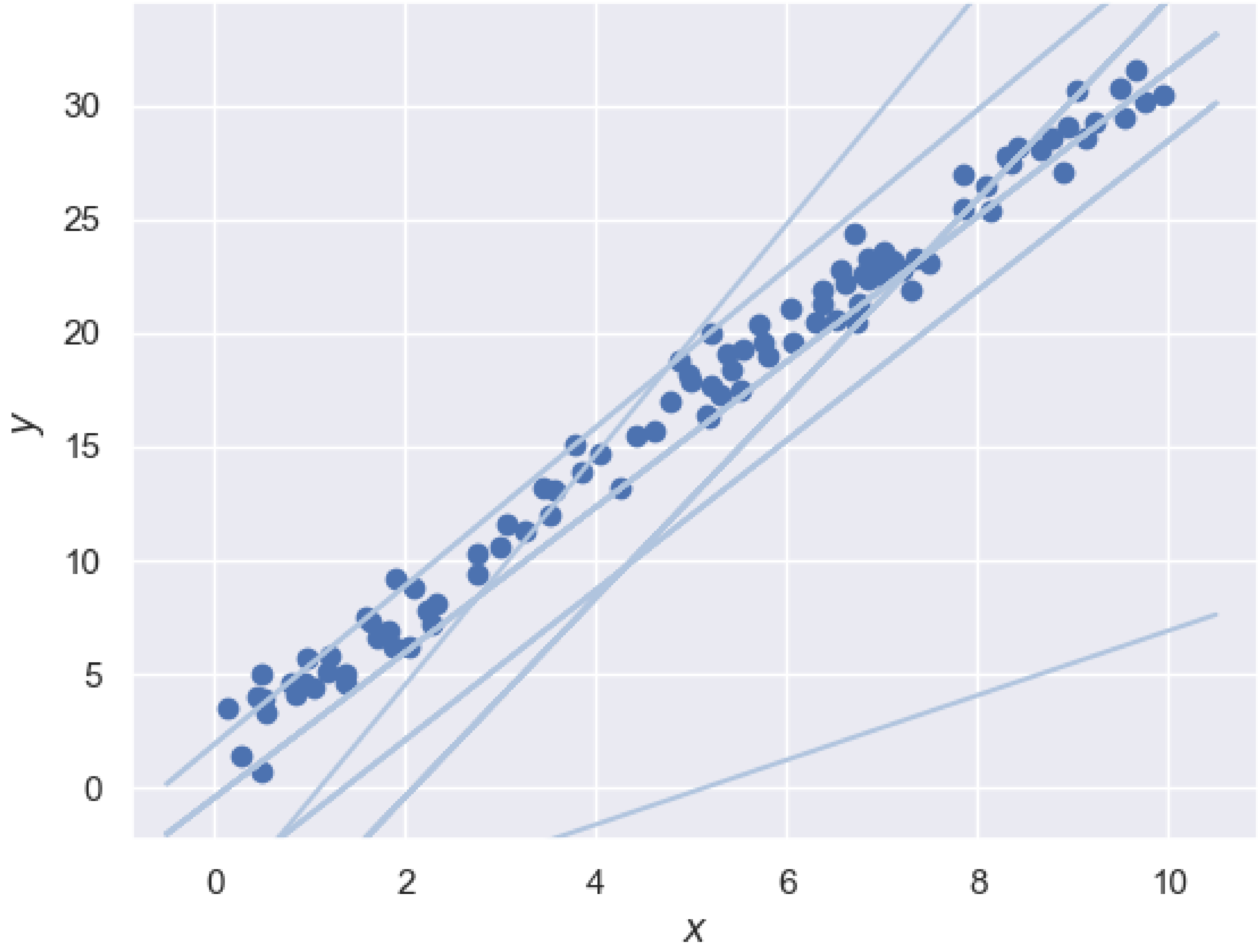
Iterating over all observed (x_i, y_i)

Observes that $y_i - mx_i - c$ is coming from a normal distribution with zero-mean

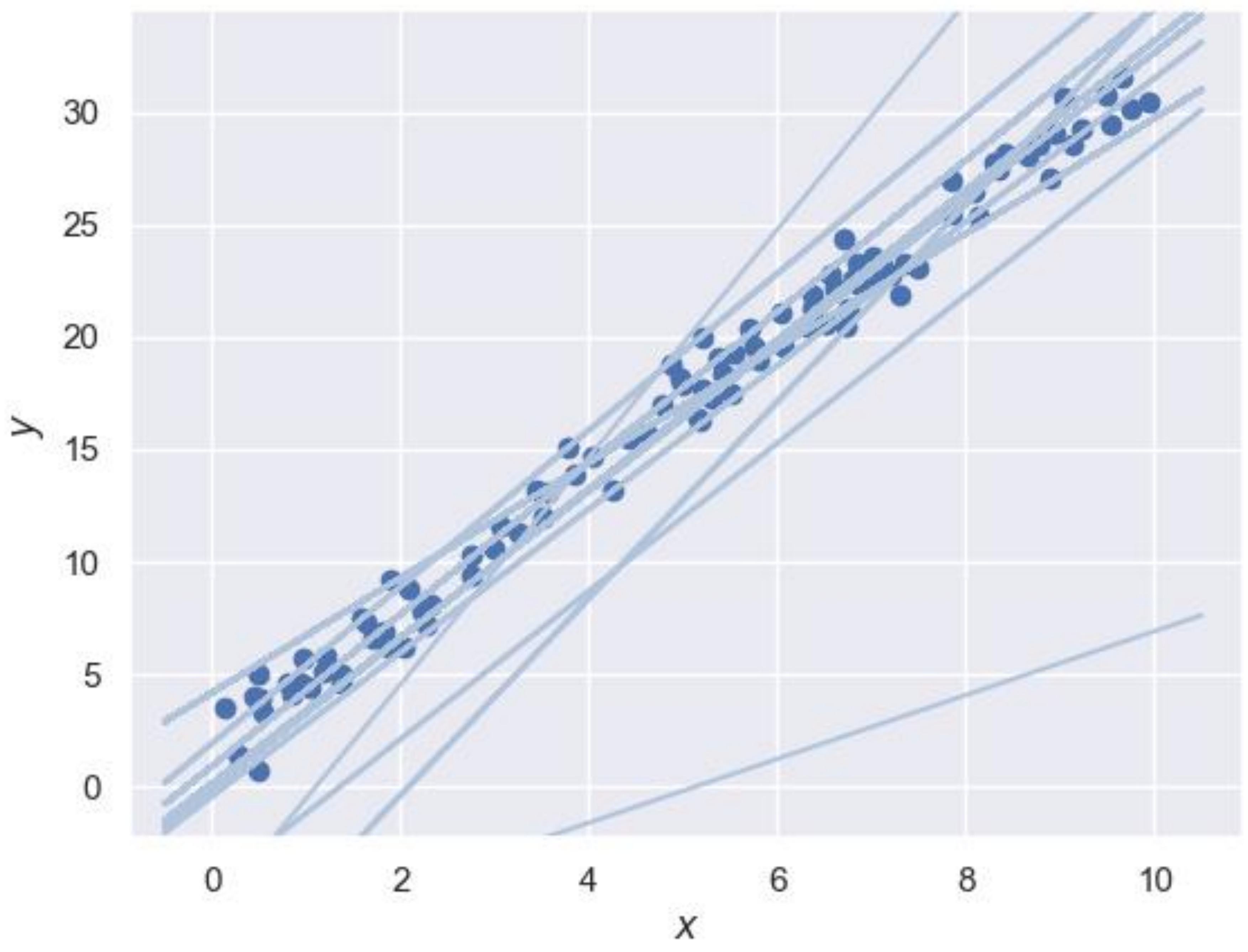
Linear Regression result



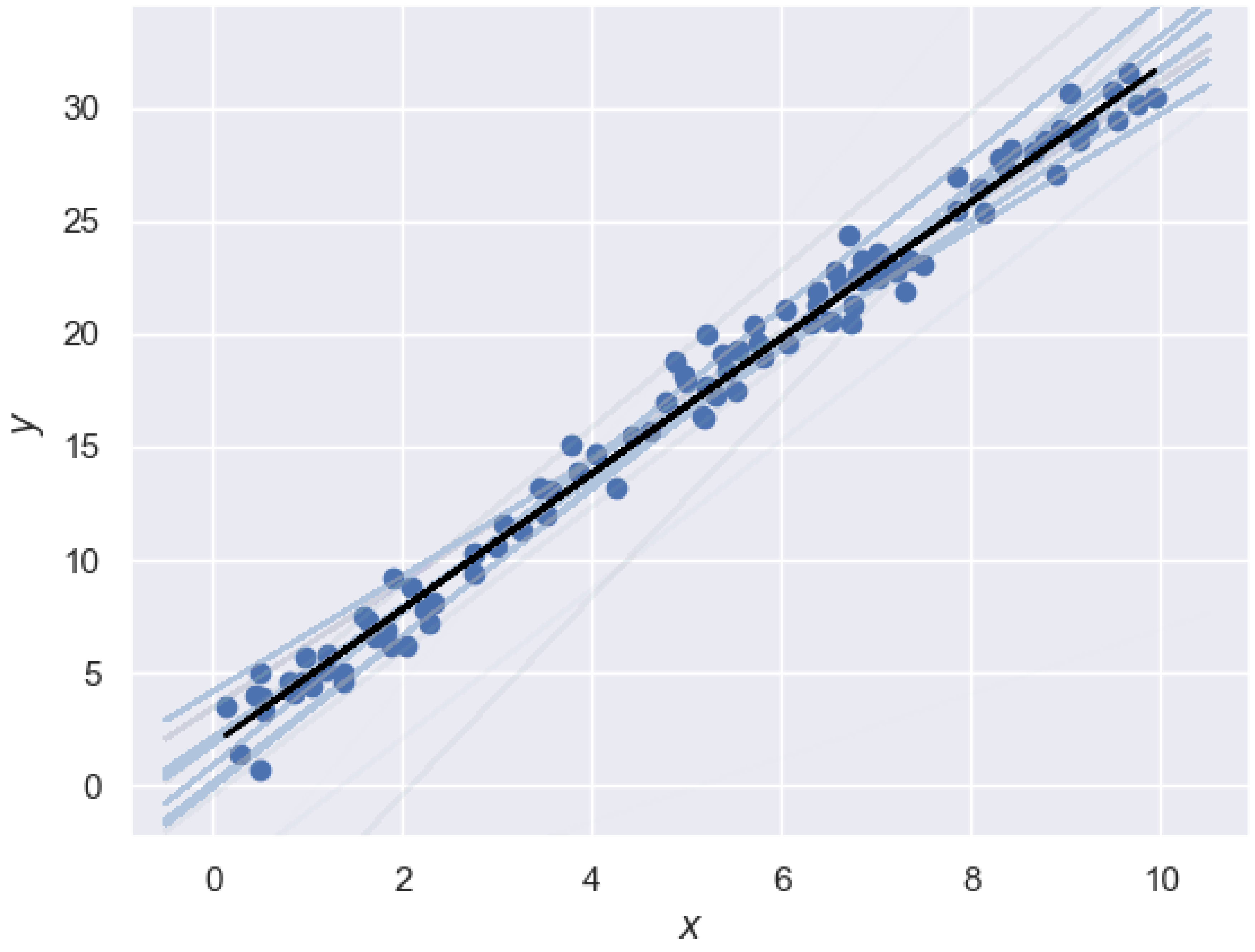
Linear Regression example



Linear Regression Result



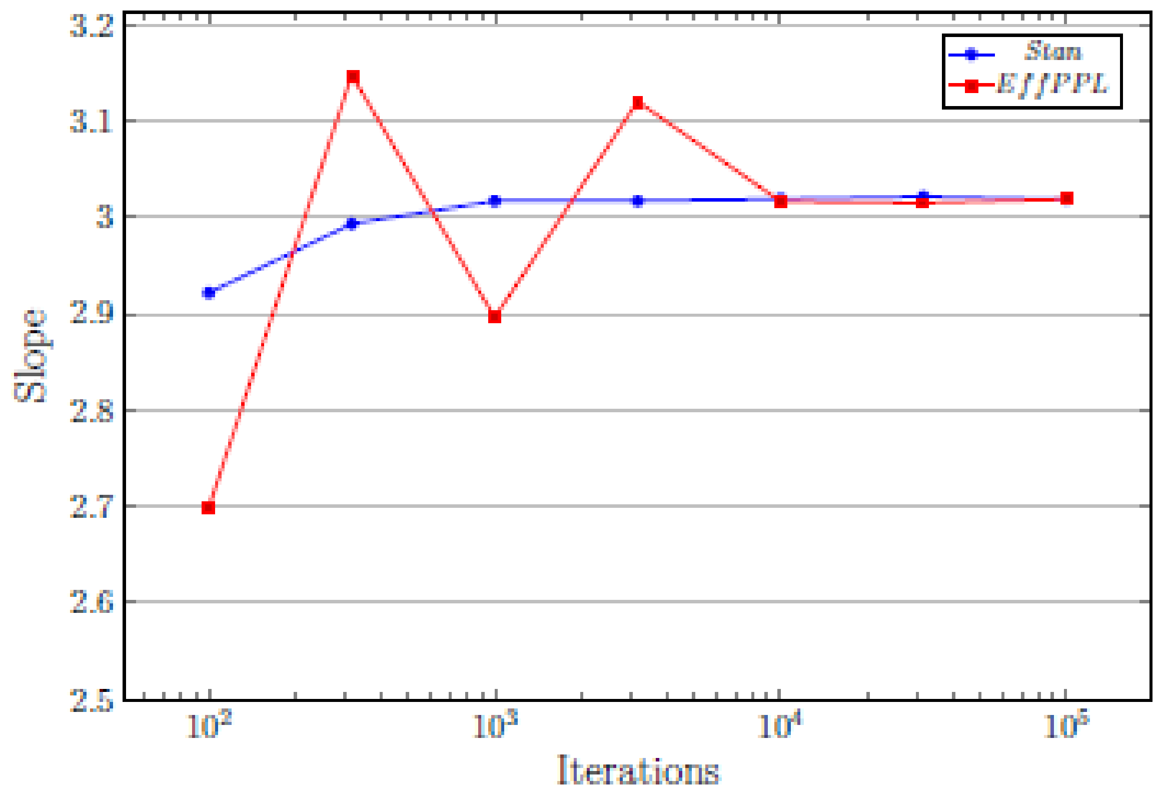
Linear Regression Result

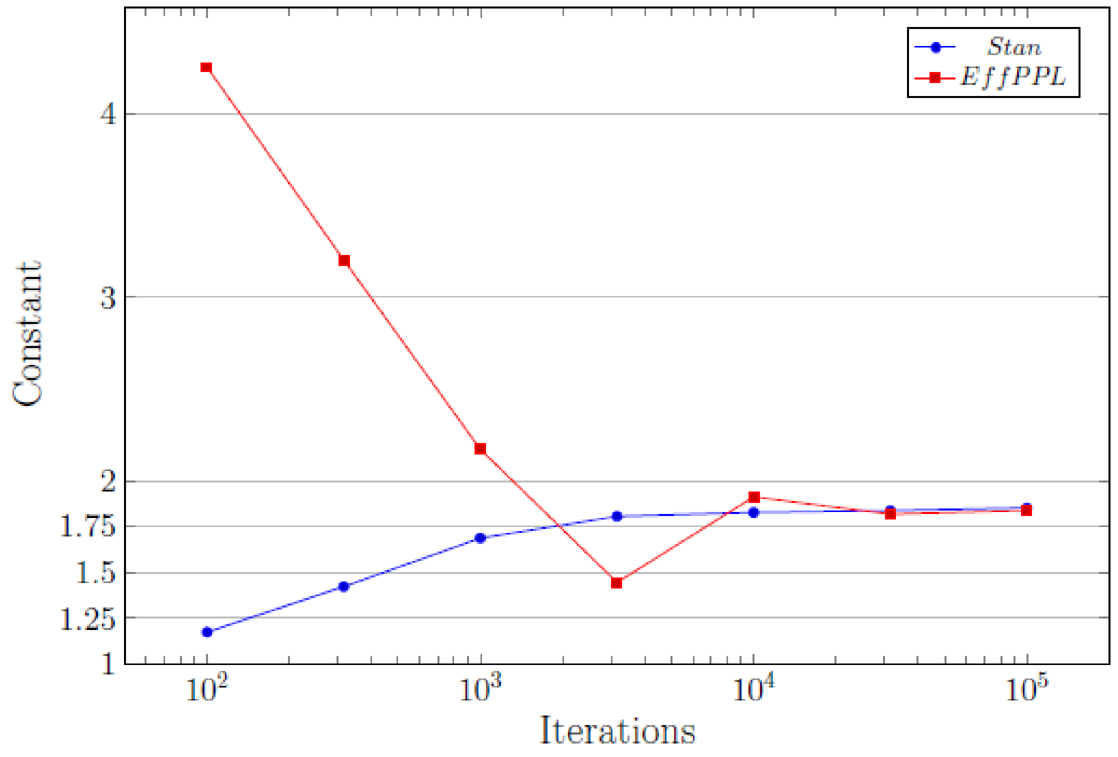


Comparison with Stan

For a linear regression model we compare how fast the means of the slopes converged in Stan versus EffPPL for the same data given to both models

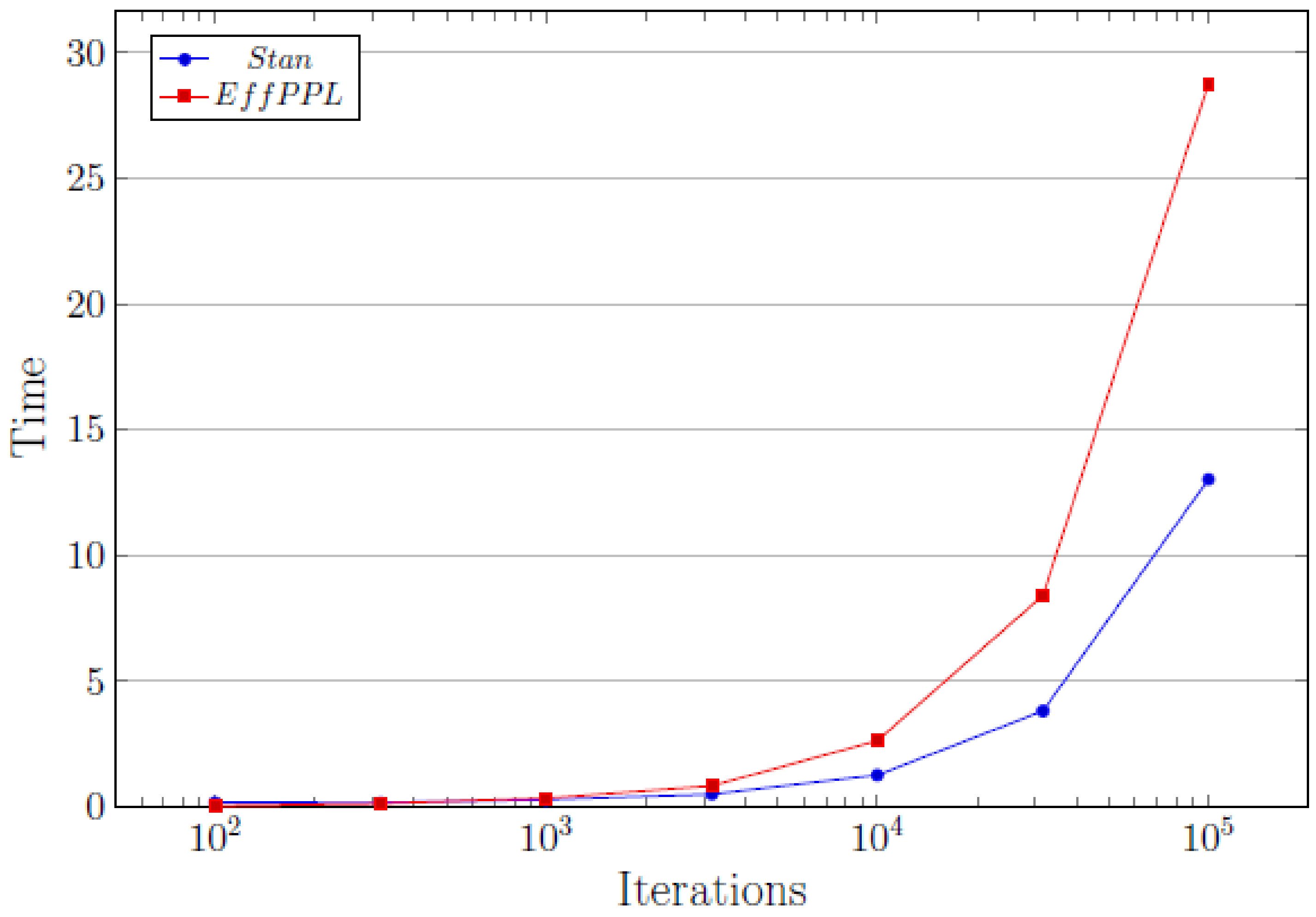
As can be seen Stan converges faster, as it achieved values very close to the mean in 10^3 iterations and $10^{3.5}$ iterations for the linear regression slopes and constants. While EffPPL took ten times more epochs to achieve a similar proximity to the means of slopes and constants.





Comparison with Stan

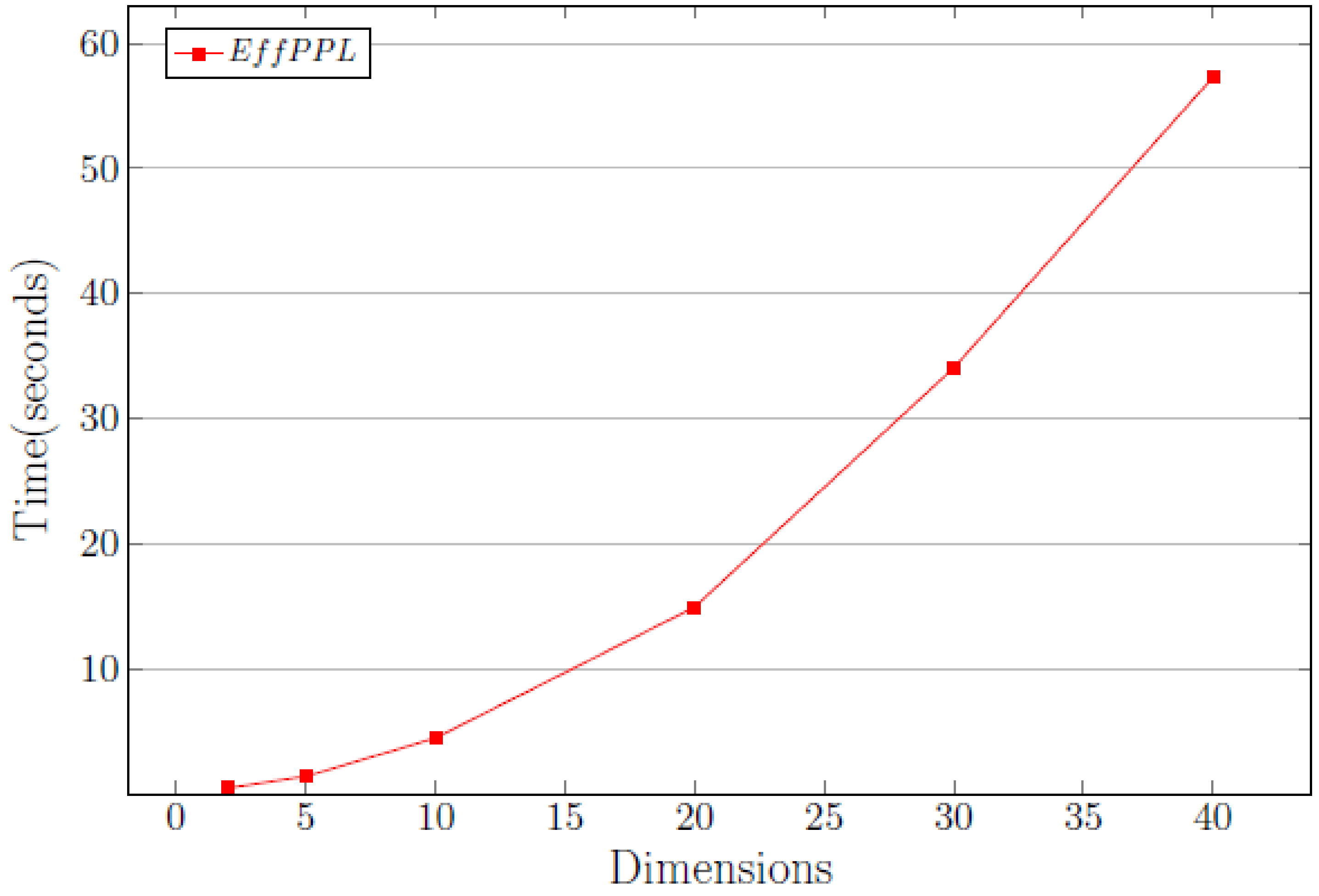
We now compare how close EffPPL was in comparison to Stan in terms of time taken by both libraries to perform a fixed number of epochs. We see that EffPPL performs well as it was able to be roughly 1.5-2X slower than Stan.



Dimensionality Evaluation

We used a linked list as the main data structure in our implementation. This leads to a complexity of $O(d^2)$, where d are the number of dimensions/parameters used in the model. Use of more advanced data structures can lead to a time complexity of $O(d)$.

Plot depicts EffPPL's performance against for varying dimensions. We don't compare against Stan here, because Stan was performing much better at $O(d)$.



Thank you.

Arnhav Datar
cs18b003@smail.iitm.ac.im
+91 9665561490