Probabilistic & Approximate Computing

Sasa Misailovic

UIUC

Operational Semantics *(deterministic)*

Simulates how the program executes on an abstract state-machine

X	Y
3	0

PC
32

```
31:   …
32:   y = x + 2
33:   …
```

X	Y
3	5

PC
33

Two flavors:

- Small-step operational semantics

  read from location pointed to by x, do addition, store to location pointed to by y

- Big-step operational semantics

  x + 2 evaluates to the value 5; store this value to the location pointed to by y

Operational Semantics *(deterministic)*

Execution Stages:

- Map the program and inputs to the ***initial configuration***

```
init:     (x, 3)
01:  def program(x){…}
```

X	Y
3	0

PC
01

- ***Execute the steps*** that represent the instructions of the abstract machine

```
32:  y = x + 2
```

X	Y
3	5

PC
33

- Map the ***final configuration*** (if it exists) to the output

```
33:  return y
```

X	Y
3	5

PC
33

Simple Deterministic Language

$$
\begin{array}{lcl}
x & \in & \text{Vars} \\
\mathcal{T} & ::= & \texttt{bool} \\
\textbf{uop} & ::= & \texttt{not} \\
\textbf{bop} & ::= & \texttt{and} \mid \texttt{or} \\
\\
\mathcal{D} & ::= & \mid \mathcal{T}\, x_1, x_2, \ldots, x_n \\
\\
\mathcal{E} & ::= & \\
 & & x \\
 & & \mid c \\
 & & \mid \mathcal{E}_1 \ \textbf{bop} \ \mathcal{E}_2 \\
 & & \mid \textbf{uop} \ \mathcal{E}_1
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{S} & ::= & \\
 & & x := \mathcal{E} \\
 & & \mid \mathcal{S}_1; \mathcal{S}_2 \\
 & & \mid \texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2 \\
 & & \mid \texttt{while } \mathcal{E} \texttt{ do } \mathcal{S}_1 \\
\\
\mathcal{P} & ::= & \mathcal{D}\ \mathcal{S}
\end{array}
$$

Subset of Language from Claret et al. Bayesian inference using data flow analysis (FSE'13)

Operational Semantics

Tuple: $S = (\mathcal{C}, \rightarrow, \mathcal{C}_{final}, \mathcal{I}, \mathcal{O})$

- $\mathcal{C}$: Set of configurations (e.g.,$\mathcal{C} = Stmt \times Stack \times Mem$)
- $\rightarrow$: Transition relation, which defines possible transitions between the configurations
  - Deterministic if for each start configuration $c \in \mathcal{C}$ there exists a single result configuration $c' \in \mathcal{C}$
  - Nondeterministic if there can be multiple $c \in \mathcal{C}$
- $\mathcal{C}_{final}$: Set of final configurations ($\mathcal{C}_{final} \subseteq \mathcal{C}$) in which the program successfully ends
- $\mathcal{I}$ maps program source and input to initial configuration $\mathcal{C}_0$
- $\mathcal{O}$ maps final configuration $\mathcal{C}_{final}$ to the output

Operational Semantics *(small step)*

Configuration: $c \in \mathcal{C} ::= Stmt \times \Sigma$

$\sigma \in \Sigma ::= (x_1 \to v_1, \ldots x_n \to v_n)$

$Stmt$ – next statement or no statement to run $(\cdot)$

x_i – variables, v_i – values

Final Configuration:

- Has the form `(skip,` σ`) = (skip,`$(x_1 \to v_1, \ldots x_n \to v_n)$`)`
- Specifies configurations in which program terminated normally
- But, execution can get stuck (there is a statement for which there is no transition)
- Or, execution may never terminate (the execution loops infinitely)

Operational Semantics *(small step)*

For expressions: $\rightarrow_b \in Expr \times \Sigma \rightarrow Expr$

e1, e2 – expressions
v1, v2 – values

$(x, \sigma) \rightarrow_b \sigma(x)$ — *Reading a variable*

$(uop\ v1, \sigma) \rightarrow_b v2$ — *Unary operation on constants (v2 is the result of unary operation)*

$$\frac{(e1, \sigma) \rightarrow_b e1'}{(uop\ e1, \sigma) \rightarrow_b uop\ e1'}$$ — *Unary operation on subexpressions*

$(v1\ bop\ v2, \sigma) \rightarrow_b v3$ — *Binary operation on constants (v3 is the result of binary operation)*

$$\frac{(e1, \sigma) \rightarrow_b e1'}{(e1\ bop\ e2, \sigma) \rightarrow_b e1'\ bop\ e2}$$ — *Binary operation on subexpressions*

$$\frac{(e2, \sigma) \rightarrow_b e2'}{(v1\ bop\ e2, \sigma) \rightarrow_b v1\ bop\ e2'}$$ — *Binary operation on subexpressions*

Operational Semantics *(small step)*

For statements: $\rightarrow \in \mathcal{C} \rightarrow \mathcal{C}$

e1, e2 – expressions
v1, v2 – values

$(\texttt{x = v}, \sigma) \rightarrow (\texttt{skip}, \sigma[x \leftarrow v])$ — *Assigning a constant*

$(\texttt{x = e}, \sigma) \rightarrow (\texttt{x = e}', \sigma)$ — *Assigning an expression*
where $(\texttt{e}, \sigma) \rightarrow_b \texttt{e'}$

$(\texttt{skip; s1}, \sigma) \rightarrow (\texttt{s1}, \sigma)$ — *Sequence rule (1)*

$(\texttt{s1; s2}, \sigma) \rightarrow (\texttt{s1'; s2}, \sigma')$ — *Sequence rule (2)*
where $(\texttt{s1}, \sigma) \rightarrow (\texttt{s1'}, \sigma')$

Operational Semantics *(small step)*

For statements: $\rightarrow \in \mathcal{C} \rightarrow \mathcal{C}$

e1, e2 – expressions
v1, v2 – values

$(\texttt{if true then s1 else s2}, \sigma) \rightarrow (\texttt{s1}, \sigma)$ ***Conditional (then)***

$(\texttt{if false then s1 else s2}, \sigma) \rightarrow (\texttt{s2}, \sigma)$ ***Conditional (else)***

$$\frac{(\texttt{e1}, \sigma) \rightarrow_b \texttt{e1'}}{(\texttt{if e then s1 else s2}, \sigma) \rightarrow (\texttt{if e' then s1 else s2}, \sigma)}$$ ***Conditional (expr)***

$(\texttt{while e do s}, \sigma) \rightarrow (\texttt{if e then \{s1; while e do s\} else skip}, \sigma)$ ***While loop***

Simple Probabilistic Language

$$
\begin{array}{lcl}
r & \in & \mathbb{R} \\
x & \in & \text{Vars} \\
\mathcal{T} & ::= & \texttt{bool} \\
\textbf{uop} & ::= & \texttt{not} \\
\textbf{bop} & ::= & \texttt{and} \mid \texttt{or} \\
\\
\mathcal{D} & ::= & \mid \mathcal{T}\, x_1, x_2, \ldots, x_n \\
\\
\mathcal{E} & ::= & \\
 & & x \\
 & & \mid c \\
 & & \mid \mathcal{E}_1 \ \textbf{bop} \ \mathcal{E}_2 \\
 & & \mid \textbf{uop} \ \mathcal{E}_1
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{S} & ::= & \\
 & & x := \mathcal{E} \\
 & & \mid x := \texttt{Bernoulli}(r) \\
 & & \mid \texttt{observe}\ (\mathcal{E}) \\
 & & \mid \texttt{skip} \\
 & & \mid \mathcal{S}_1; \mathcal{S}_2 \\
 & & \mid \texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2 \\
 & & \mid \texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}_1 \\
\\
\mathcal{P} & ::= & \mathcal{D}\ \mathcal{S}
\end{array}
$$

Claret et al. Bayesian inference using data flow analysis (FSE'13)

Probabilistic State

Deterministic

State: $\sigma_0 \in \Sigma_0 ::= (x_1 \to v_1, \ldots x_n \to v_n)$

x_i −variables v_i − values

Probabilistic

State: $\sigma \in \Sigma = \Sigma_0$

Expressions: $A \in \mathcal{E} \times \Sigma \to \text{Bool}$

Statements: $T \subseteq (S \times \Sigma) \times (\Sigma \times [0,1])$

(we use notation $(\cdot,\cdot) \xrightarrow{\cdot} \cdot$)

Probabilistic Assignment

Statements: $T \subseteq (S \times \Sigma) \times (\Sigma \times [0,1])$

$(\texttt{x = v}, \sigma) \rightarrow (\texttt{skip}, \sigma[x \leftarrow v])$ ***Assigning a constant***

$(\texttt{x = e}, \sigma) \rightarrow (\texttt{x = e}', \sigma)$ ***Assigning an expression***
where $(\texttt{e}, \sigma) \rightarrow_b \texttt{e}'$

$$(x = \texttt{Bern}(p_B), \sigma) \xrightarrow{p_B} (\texttt{skip}, \sigma[x \leftarrow \text{True}])$$
$$(x = \texttt{Bern}(p_B), \sigma) \xrightarrow{1-p_B} (\texttt{skip}, \sigma[x \leftarrow \text{False}])$$

Probabilistic Control Flow

$$(\texttt{skip; s1},\ \sigma) \xrightarrow{1} (\texttt{s1}, \sigma)$$

Sequence rule (1)

$$\frac{(\texttt{s1},\ \sigma) \xrightarrow{p_1} (\texttt{s1'},\ \sigma')}{(\texttt{s1; s2},\ \sigma) \xrightarrow{p_1} (\texttt{s1'; s2},\ \sigma')}$$

Sequence rule (2)

Recall:

$$(x = \texttt{Bern}(p_B), \sigma) \xrightarrow{p_B} (\texttt{skip}, \sigma[x \leftarrow \text{True}])$$

$$(x = \texttt{Bern}(p_B), \sigma) \xrightarrow{1-p_B} (\texttt{skip}, \sigma[x \leftarrow \text{False}])$$

Example: Bernoulli Program

(

, 1.0)

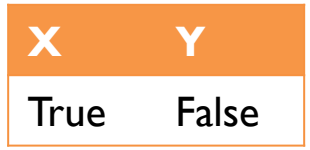
```
X := Bernoulli(0.7);
```

(

, 0.7), (

, 0.3)

```
Y := not X;
```

(

, 0.7), (

, 0.3)

Example: Bernoulli Program

(

, 1.0)

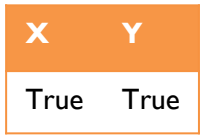
```
X := Bernoulli(0.7);
```

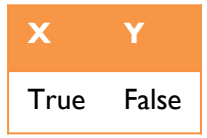
(

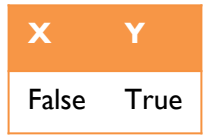
, 0.7), (

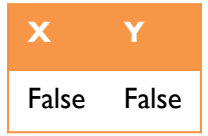
, 0.3)

```
Y := Bernoulli(0.7);
```

(

, 0.49), (

, 0.21), (

, 0.21), (

, 0.09)

Bring together: Probability of a Trace

(Finite) trace of executing a statement/program ($c \in \mathcal{C}$):

$$\theta = c_1 \xrightarrow{p_1} c_2 \xrightarrow{p_2} c_3 \xrightarrow{p_3} \ldots \xrightarrow{p_n} c_{n+1}$$

- $c_1 = (S, \sigma_{init})$ is an initial configuration
- $c_{n+1} = (\texttt{skip}, \sigma_{final})$ is the final configuration, assuming that S terminates
- The execution took specific transitions from c_1 to c_{n+1}

Probability of the trace: $\Pr(\theta) = p_1 \cdot p_2 \cdot \ldots \cdot p_n$

Example: Bernoulli Program

(

, 1.0)

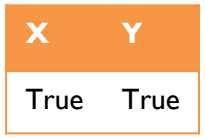
```
X := Bernoulli(0.7);
```

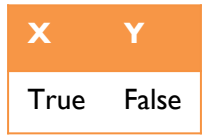
(

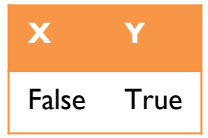
, 0.7), (

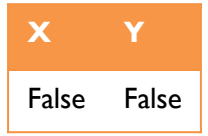
, 0.3)

```
Y := Bernoulli(0.7);
```

(

, 0.49), (

, 0.21), (

, 0.21), (

, 0.09)

Observations

$$(\text{condition True}, \sigma) \xrightarrow{1} (\text{skip}, \sigma)$$

$$(\text{condition False}, \sigma) \xrightarrow{1} \emptyset$$

(doesn't exist: no such transition*)

**alternatively: can go to special end state*

Normalization: If the probabilities $p_1 \ldots p_k$ do not sum up to 1, rescale them so that they do:

1. Compute sum: $Z = p_1 + p_2 + \cdots + p_k$
2. Rescale: $p_1' = \frac{p_1}{Z}, p_2' = \frac{p_1}{Z}, \ldots p_k' = \frac{p_k}{Z}$

Do only at the end of the program – although still expensive

Example: Bernoulli Program

(X Y -- -- , 1.0)

```
X := Bernoulli(0.7);
Y := Bernoulli(0.7);
```

(X Y True True , 0.49), (X Y True False , 0.21), (X Y False True , 0.21), (X Y False False , 0.09)

```
condition ( X == True );
return Y;
```

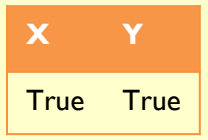
(X Y True True , 0.49/0.7), (X Y True False , 0.21/0.7),

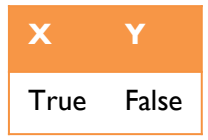
Example: Bernoulli Program

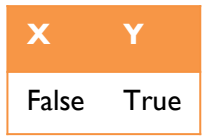
(

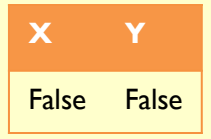
, 1.0)

```
X := Bernoulli(0.7);
Y := Bernoulli(0.7);
```

(

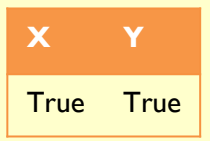
, 0.49), (

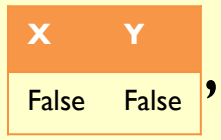
, 0.21), (

, 0.21), (

, 0.09)

```
condition ( X == Y );
return Y;
```

(

, 0.49/0.58), (

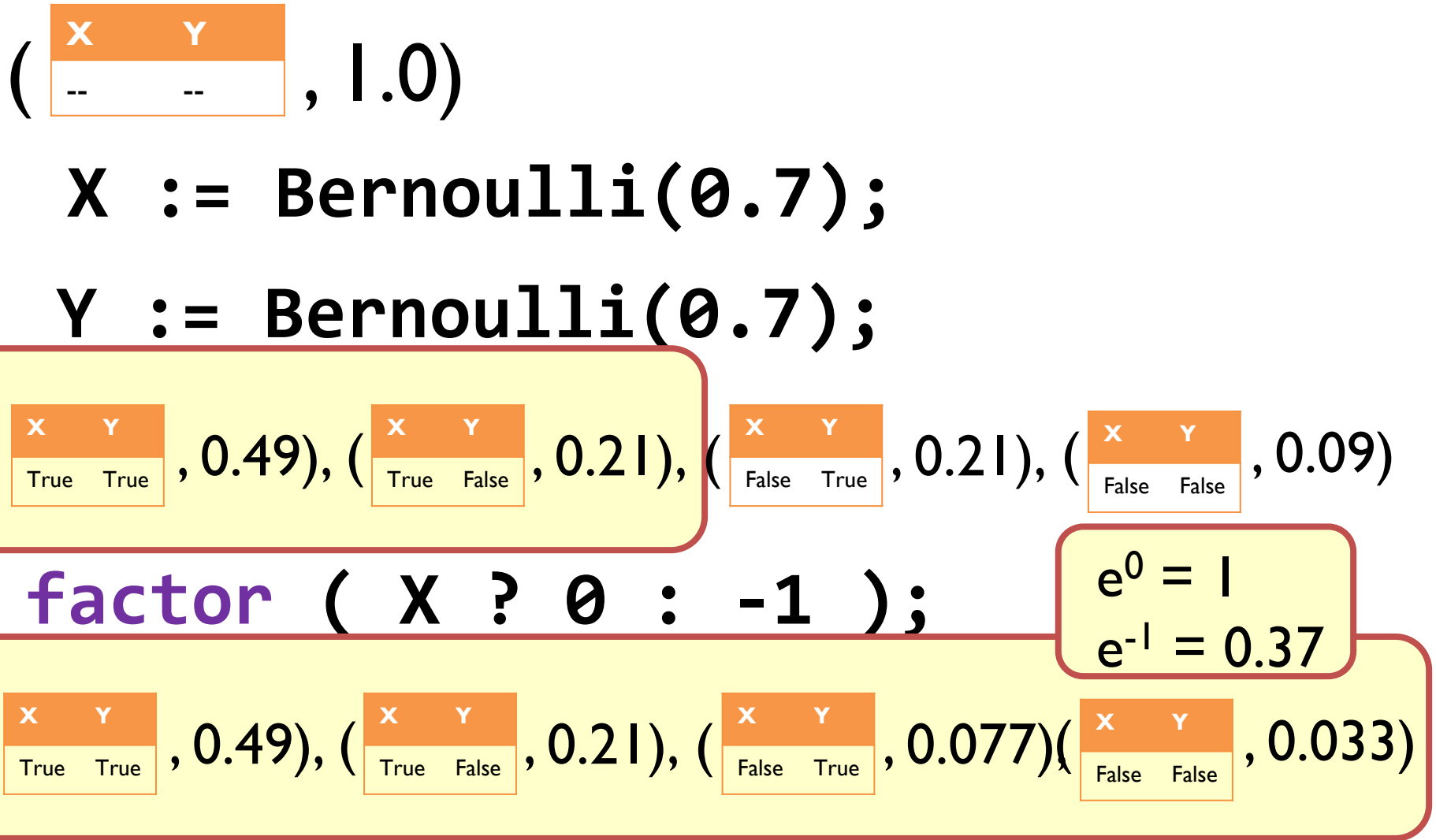
, 0.09/0.58)

Observations

$$(\text{factor value}, \sigma) \xrightarrow{value} (\text{skip}, \sigma)$$

Normalization: still necessary

Example: Bernoulli Program



Example: Bernoulli Program

factor => condition

(X Y -- -- , 1.0)

```
X := Bernoulli(0.7);
Y := Bernoulli(0.7);
```

(X Y True True , 0.49), (X Y True False , 0.21), (X Y False True , 0.21), (X Y False False , 0.09)

```
factor ( X ? 0 : -∞ );
```

$e^{0} = 1$

$e^{-\infty} = 0$

(X Y True True , 0.49), (X Y True False , 0.21), (X Y False True , 0), (X Y False False , 0)

Denotational Semantics

Syntactic Domain: describes the syntax of the language (grammar): elements are e.g., nodes in the abstract syntax tree (AST)

Semantic Domain: mathematical entities and operations on them

- For instance, sets of numbers, sets of tuples
- For probabilistic programs expectations are easy to envision

Meaning Function: Translates elements of the syntactic domain to the elements and operations in the semantic domain

- Compositionality: the meaning of the AST is composite of the meaning of its nodes

How Do We Get Randomness?

Linear Congruental Generator

```
int rseed = 0;

inline void srand(int x)
    rseed = x;
}
```

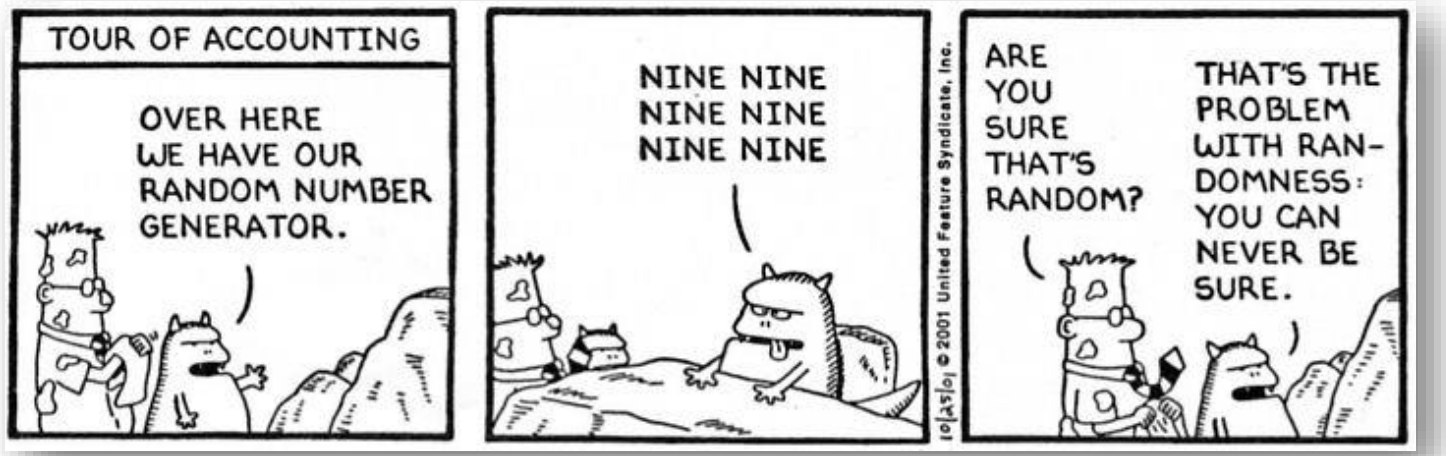
```
#define RAND_MAX ((1U << 31) - 1)

inline int rand() {
   return rseed =
          (rseed * 1103515245 + 12345) & RAND_MAX;
}
```

https://rosettacode.org/wiki/Linear_congruential_generator

Still Better Than…

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

© xkcd



Better Pseudorandom Generators

Mersenne Twister

- Large cycle (up to $2^{19937}-1$)
- Fast, but requires lots of (cache) memory
- Default choice for the languages from this century

Xorshift

- Moderate cycle (from $2^{64} - 1$ to $2^{1024} - 1$)
- Very fast, uses only bitshifts and xor operators
- May not pass all tests for uniformity, but good for simulation

Simulation vs. cryptography (e.g., Yarrow/Fortuna)

True Randomness?

Hardware generators

- Based on thermal noise (or other natural phenomena)
- Main use is cryptographic, speed is less of a concern
- E.g., Intel IvyBridge-EP microarchitecture uses hardware RNG (see RDRAND instruction)

True random sequences from the Internet

- E.g., https://www.random.org/ gets numbers from atmospheric noise

Tests for pseudorandom number generators

- DieHard (Marsaglia)
- TestU01 (L'Ecuyer and Simard)

PROBABILISTIC 4 APPROXIMATE

CASE 1: Sum Computation

- Original sum computation

```
s = 0;
for (i = 0; i < n; i++)    s = s + f(i);
```

- Perforated, extrapolated sum computation

```
s = 0;
for (i = 0; i < n; i += 2)    s = s + f(i);
  s = s * 2;
```

Step 1: Represent Result Difference

- Original sum computation

```
s = 0;
for (i = 0; i < n; i++)    s = s + f(i);
```

- Perforated, extrapolated sum computation

```
s = 0;
for (i = 0; i < n; i += 2)    s = s + f(i);
  s = s * 2;
```

- Perforation noise: $D = s_{original} - s_{perforated}$

Step 2: Probabilistic Modeling

- Original sum computation

```
s = 0;
for (i = 0; i < n; i++)    s = s + f(i);
```

- Perforated, extrapolated sum computation

```
s = 0;
for (i = 0; i < n; i += 2)    s = s + f(i);
  s = s * 2;
```

- Perforation noise: $D = s_{original} - s_{perforated}$

Step 2: Probabilistic Modeling

- Original sum computation

```
s = 0;
for (i = 0; i < n; i++)    s = s + X_i;
```

- Perforated, extrapolated sum comp

```
s = 0;
for (i = 0; i < n; i += 2)    s = s + X_i;
  s = s * 2;
```

- Perforation noise: $D = s_{original} - s_{perforated}$

Analysis: Input/Output Relation

Perforation noise:

$$D = s_{original} - s_{perforated}$$

Analysis: Input/Output Relation

Perforation noise:

$$D = s_{original} - s_{perforated}$$

$$= X_0 + X_1 + X_2 + X_3 + X_4 + X_5 + X_6 + X_7 + \ldots$$

$$- 2 \cdot (X_0 + X_2 + X_4 + X_6 + \ldots)$$

Analysis: Input/Output Relation

Perforation noise*:

$$D = s_{original} - s_{perforated}$$

$$= \quad X_0 + X_1 + X_2 + X_3 + X_4 + X_5 + X_6 + X_7 + \ldots$$

$$- \; X_0 - X_0 - X_2 - X_2 - X_4 - X_4 - X_6 - X_6 - \ldots$$

* Assuming for simplicity that the number of elements is even

Analysis: Input/Output Relation

Perforation noise*:

$$D = s_{original} - s_{perforated}$$

$$= \quad X_0 + X_1 + X_2 + X_3 + X_4 + X_5 + X_6 + X_7 + \ldots$$

$$- \; X_0 - X_0 - X_2 - X_2 - X_4 - X_4 - X_6 - X_6 - \ldots$$

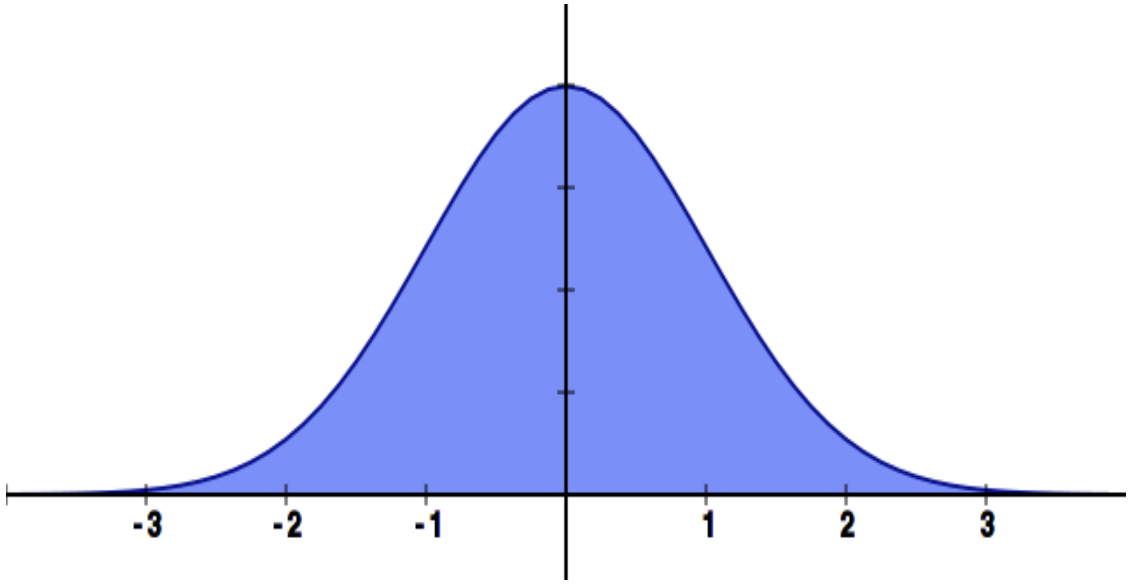
$$= \sum_{0 \le i < \frac{n}{2}} (X_{2i+1} - X_{2i})$$

* Assuming for simplicity that the number of elements is even

Analysis Results

Perforation noise:

$$D = \varphi(X_0, X_2, \ldots, X_{n-1})$$



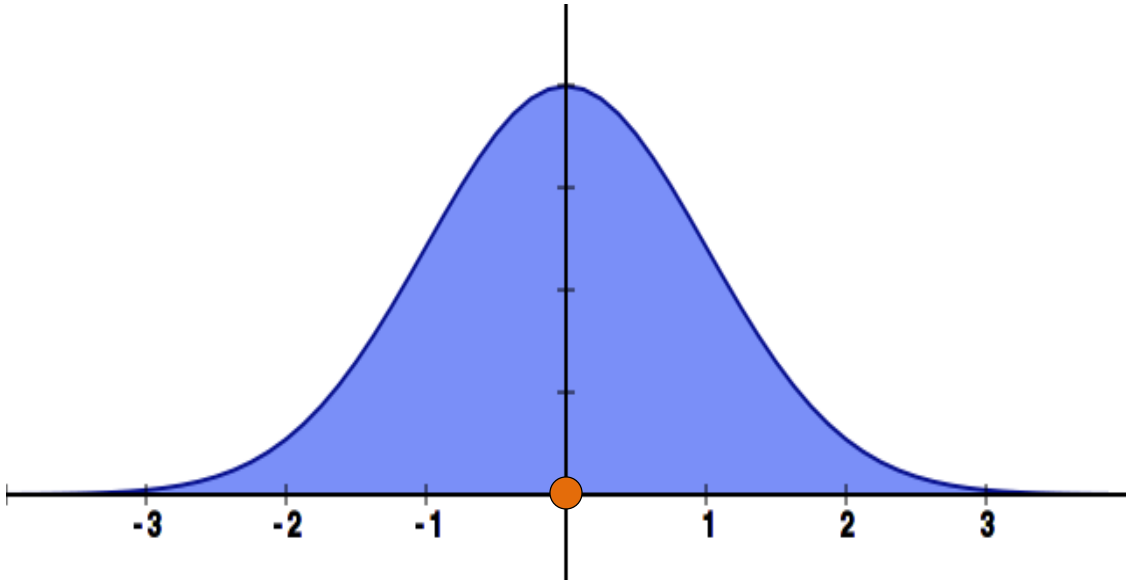
Analysis Results

Perforation noise:

$$D = \varphi(X_0, X_2, \ldots, X_{n-1})$$

Location: Mean

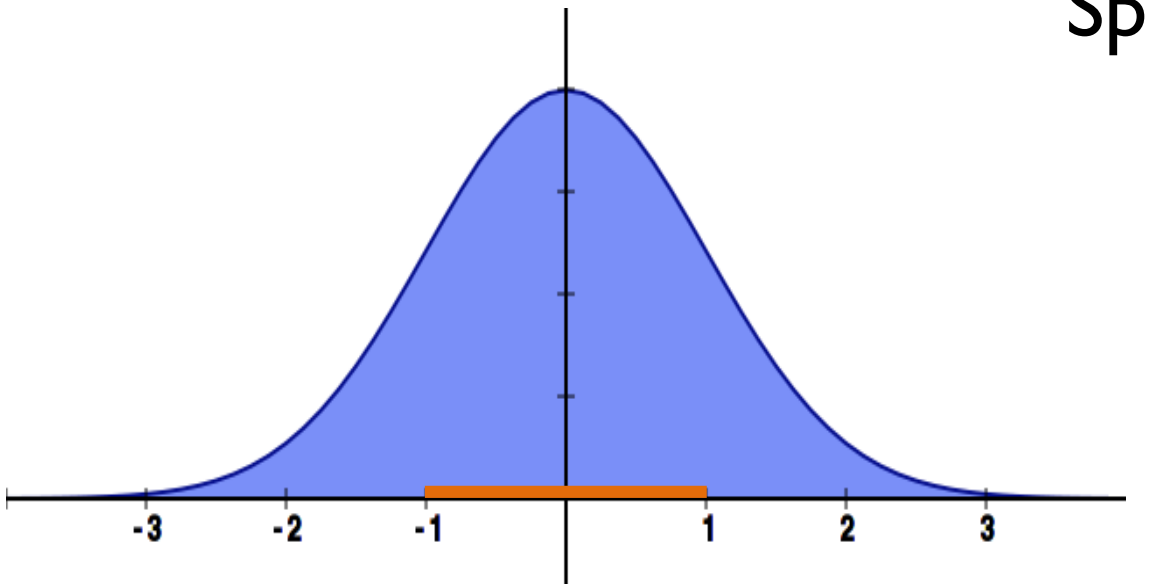
$$E(D) = \mu$$



Analysis Results

Perforation noise:

$$D = \varphi(X_0, X_2, \ldots, X_{n-1})$$



Location: Mean

$$E(D) = \mu$$

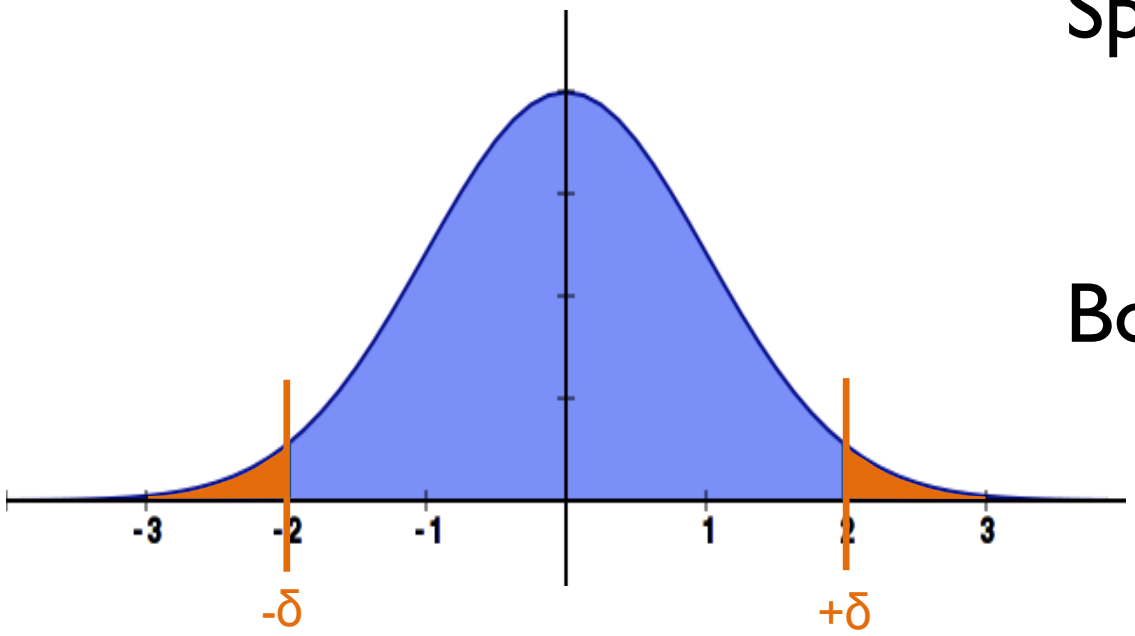
Spread: Variance

$$Var(D) = \sigma^2$$

Analysis Results

Perforation noise:

$$D = \varphi(X_0, X_2, \ldots, X_{n-1})$$



Location: Mean

$$E(D) = \mu$$

Spread: Variance

$$Var(D) = \sigma^2$$

Bound: Distribution tail

$$Pr[|D| > \delta] < \varepsilon$$