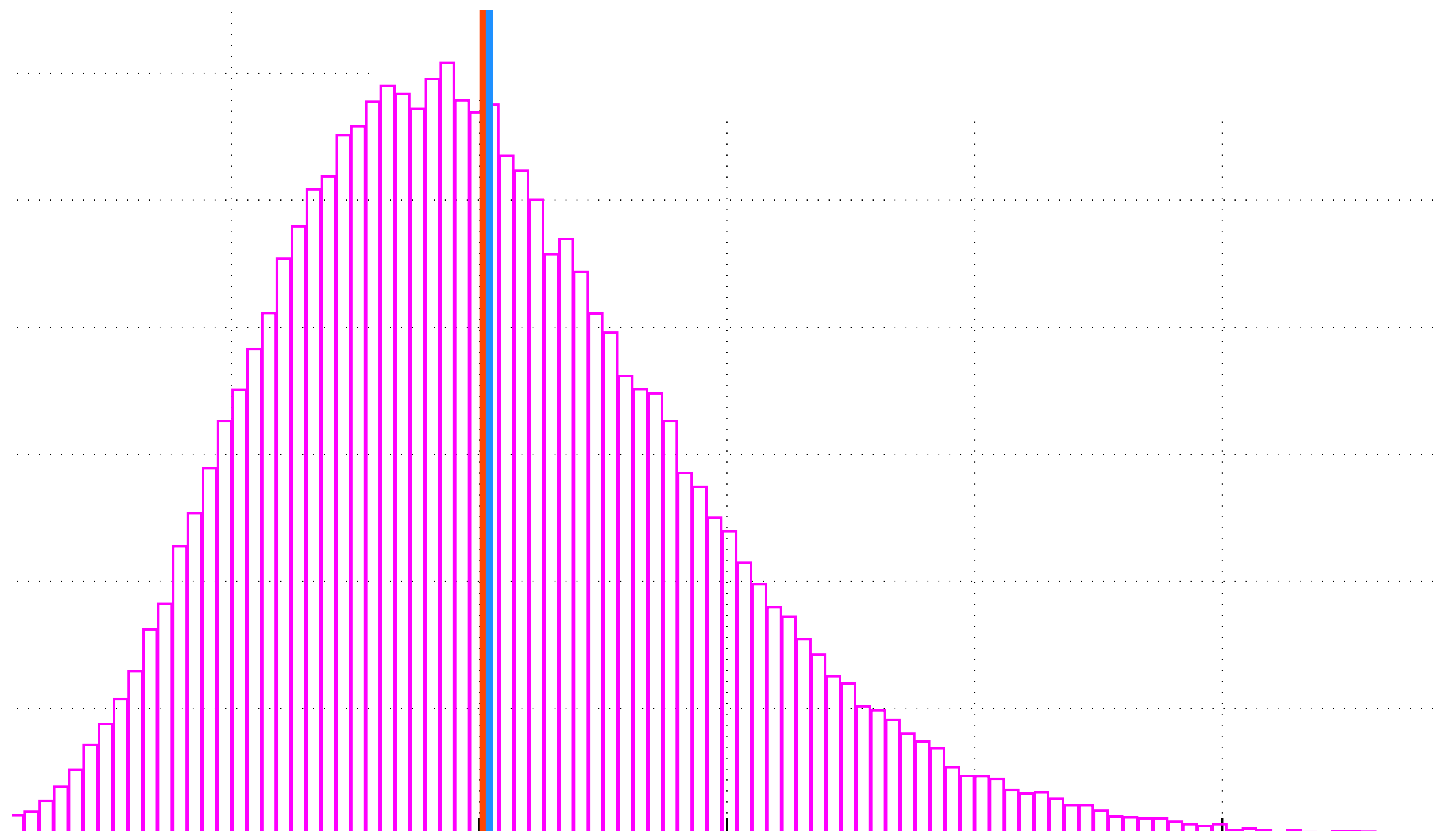
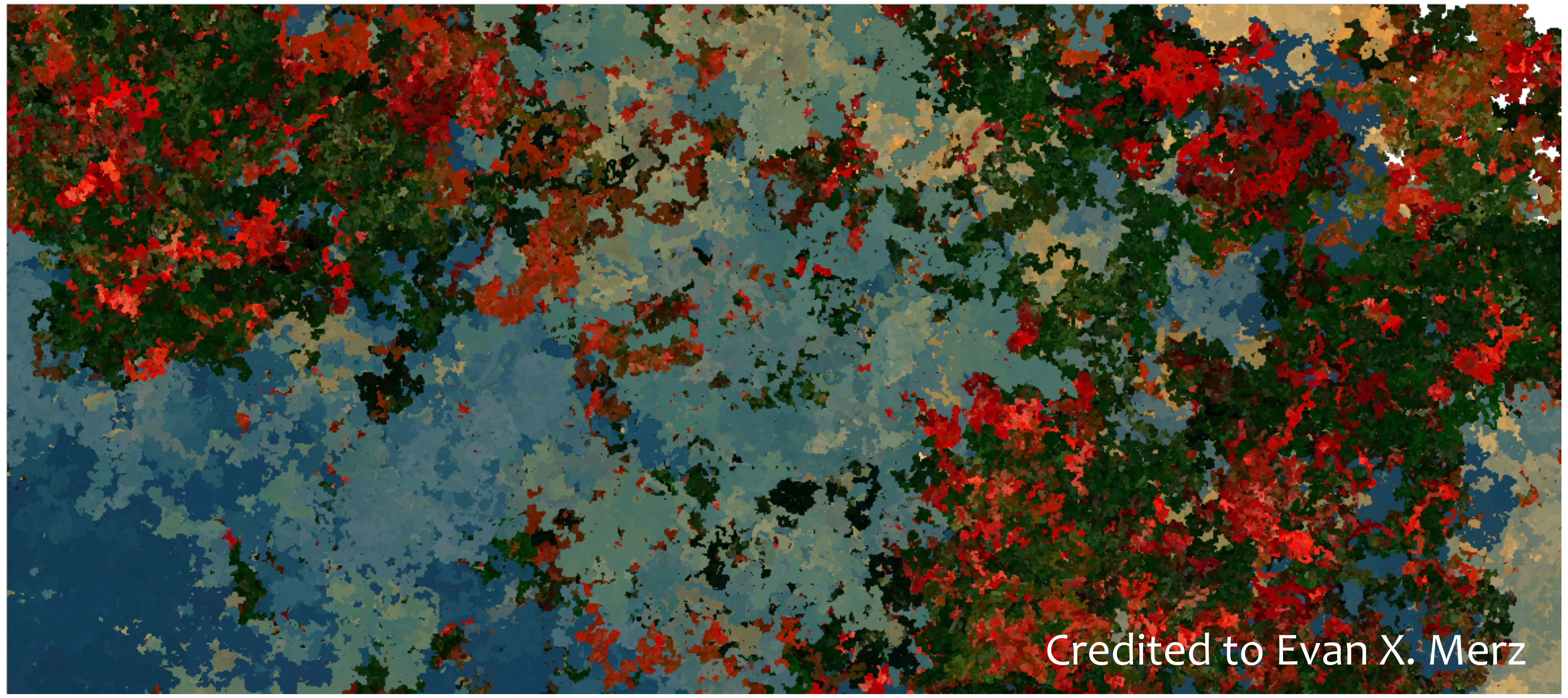


Histogram: 1D random walk



Drunk painting: 2D random walks

Bounded Expectations:
Resource Analysis for Probabilistic Programs

Van Chan Ngo   *Quentin Carbonneaux*   *Jan Hoffmann*

Carnegie Mellon University

Static resource analysis

Given: A program P

Question: What is the amount of resource as function of the inputs sizes that is required to execute P?

Static resource analysis

Given: A program P

Time, memory, or energy

Question: What is the amount of resource as function of the inputs sizes that is required to execute P?

Static resource analysis

Given: A program P

Time, memory, or energy

Question: What is the amount of resource as function of the inputs sizes that is required to execute P?

Goal: To help developers answer this question as an analysis of the programming language support

Static resource analysis

Given: A program P

Time, memory, or energy

Question: What is the amount of resource as function of the inputs sizes that is required to execute P?

Goal: To help developers answer this question as an analysis of the programming language support

Techniques
Recurrence Relations
Type Systems
Abstract Interpretation
Term Rewriting
Ranking Functions
Automatic Amortized Resource Analysis

Static resource analysis

Given: A program P

Time, memory, or energy

Question: What is the amount of resource as function of the inputs sizes that is required to execute P?

Goal: To help developers answer this question as an analysis of the programming language support

Worst-case resource usage

Techniques
Recurrence Relations
Type Systems
Abstract Interpretation
Term Rewriting
Ranking Functions
Automatic Amortized Resource Analysis

Probabilistic programs

- Are usual functional or imperative programs with two added constructs:
  - Sampling assignments to draw values at random from probability distributions, and
  - Probabilistic branchings to control program flow by observations

Probabilistic programs

- Are usual functional or imperative programs with two added constructs:
  - Sampling assignments to draw values at random from probability distributions, and
  - Probabilistic branchings to control program flow by observations

Hicks 2014

"The crux of probabilistic programming is to consider normal-looking programs as if they were probability distributions"

Probabilistic programs

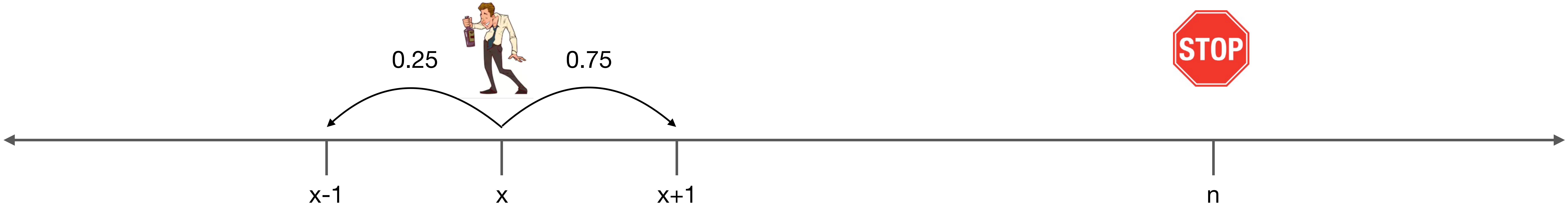
- Are usual functional or imperative programs with two added constructs:
  - Sampling assignments to draw values at random from probability distributions, and
  - Probabilistic branchings to control program flow by observations

Hicks 2014

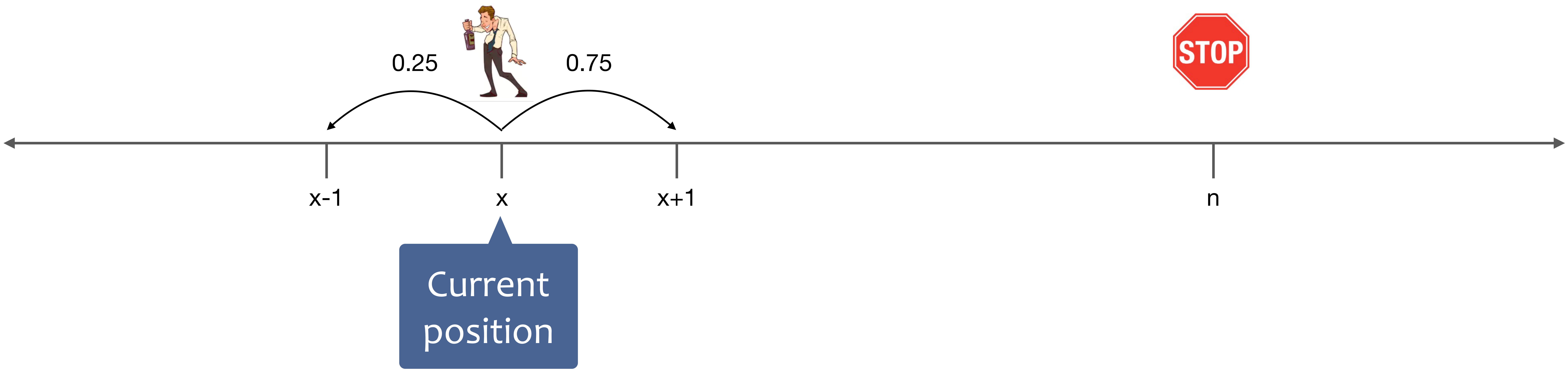
"The crux of probabilistic programming is to consider normal-looking programs as if they were probability distributions"

- Some probabilistic programming languages: *Probabilistic C*, *Church*, *PyMC3*, *Figaro*, *Edward*

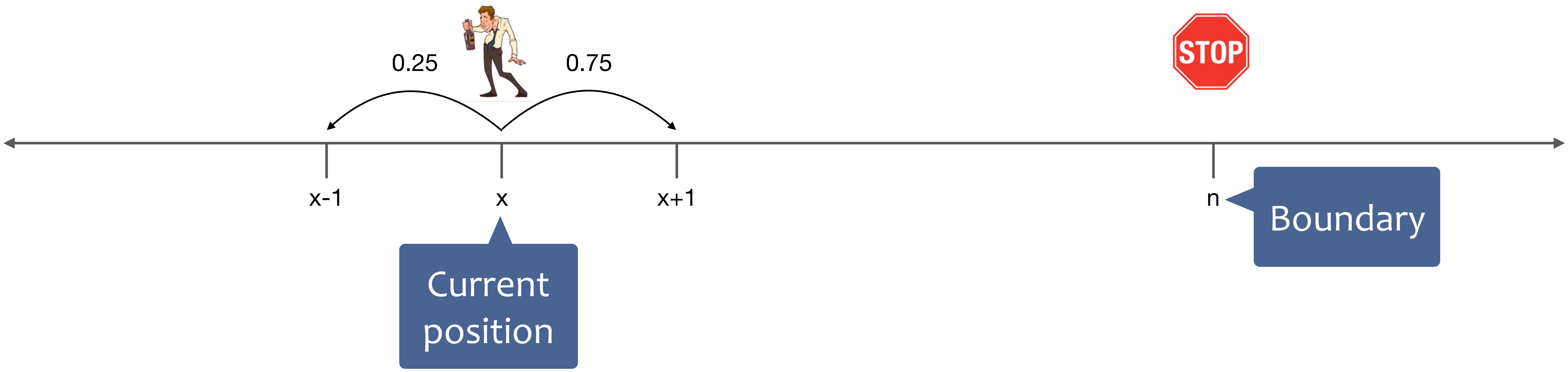
Example: Random walk



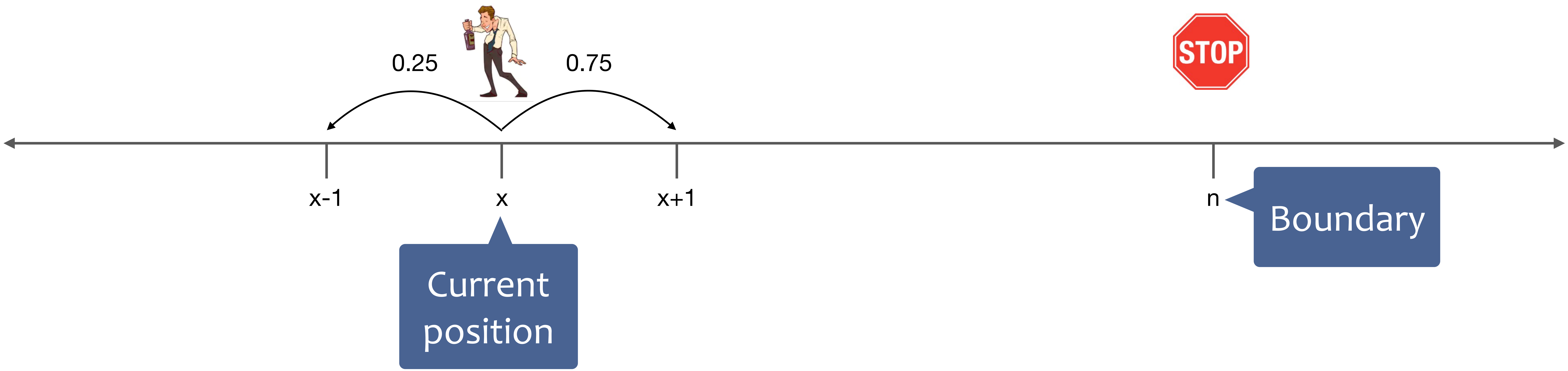
Example: Random walk



Example: Random walk



Example: Random walk




```
while x < n:
    prob(3,1):
        x = x + 1
    else:
        x = x - 1
    tick 1
```

Example: Random walk


```
while x < n:
  prob(3,1):
    x = x + 1
  else:
    x = x - 1
  tick 1
```

Cost = # iterations
= walking time

Example: Random walk

Current position

Boundary

- The worst-case cost is infinite
  - It does not make sense to reason.about

```
while x < n:
  prob(3,1):
    x = x + 1
  else:
    x = x - 1
  tick 1
```

Cost = # iterations
= walking time

Example: Random walk

Boundary

Current position

```
while x < n:
   prob(3,1):
      x = x + 1
   else:
      x = x - 1
   tick 1
```

Cost = # iterations
= walking time

- The worst-case cost is infinite
  - It does not make sense to reason.about
- It is better to reason about:
  - The expected value of the cost, or
  - The probability that the cost is bounded by a threshold

Example: Random walk

Boundary

Current position

- The worst-case cost is infinite
  - It does not make sense to re
- It is better to reason about:
  - The expected value of the cost, or
  - The probability that the cost is bounded by a threshold

This talk: automatic bounds on the expected cost

```
while x < n:
  prob(3,1):
    x = x + 1
  else:
    x = x - 1
  tick 1
```

Cost = # iterations
= walking time

Why expected resource usage?

Why expected resource usage?

There are many interesting applications:

- Predict the expected resource usage of sampling in probabilistic inference
- Reason about the average-case complexity of randomized algorithms, positive and almost-sure terminations

Why expected resource usage?

There are many interesting applications:

- Predict the expected resource usage of sampling in probabilistic inference

- Reason about the average-case complexity of randomized algorithms, positive and almost-sure terminations

It is a technical challenge problem:

- Manual analysis is often difficult or impossible even for simple programs (e.g., requires probability theory knowledge, mathematic reasoning, …)

- No techniques that automatically infer symbolic bounds on the expected cost

Approach: Expected potential method

Kozen ('81), McIver et al ('04), Kaminski et al ('16)

Weakest Pre-expectation Calculus
Strength and conceptual simplicity
Soundness w.r.t a simple operational semantics

Hofmann and Jost ('03)

Automatic Amortized Resource Analysis
Template-based bound inference
Efficiently reduced to LP solving

Expected Potential Method

Approach: Expected potential method

Kozen ('81), McIver et al ('04), Kaminski et al ('16)

Weakest Pre-expectation Calculus

- Strength and conceptual simplicity
- Soundness w.r.t a simple operational semantics

Hofmann and Jost ('03)

Automatic Amortized Resource Analysis

- Template-based bound inference
- Efficiently reduced to LP solving

Expected Potential Method

- Automatically infers bounds on expected resource usage
- Bounds are multivariate symbolic polynomials
- Enables compositional and effective reasoning

Expected potential method

- Associate potential functions to program points
  - Function from states to non-negative values
- Potential pays the expected resource consumption and the expected potential at the following point
- The initial potential is an upper bound on the expected resource usage

$$\Phi(state) \geq 0$$

$$\Phi(state) \geq \mathbb{E}(\text{cost}) + \mathbb{E}(\Phi'(next\_state))$$

Total expectation and linearity

$$\Phi(init\_state) \geq \mathbb{E}(\Sigma\text{cost})$$

Quantitative Hoare logic

$$\{\Phi\} \quad c \ \{\Phi'\}$$

Quantitative Hoare logic

Expected cost $\mathbb{E}(c)$

$$\{\Phi\} \quad c \ \{\Phi'\}$$

Quantitative Hoare logic

Expected cost $\mathbb{E}(c)$

Resource available before executing c

$$\{\Phi\} \quad c \; \{\Phi'\}$$

Quantitative Hoare logic

Resource available before executing c

Expected cost $\mathbb{E}(c)$

$$\{\Phi\} \quad c \ \{\Phi'\}$$

$\mathbb{E}(\Phi')$ is the expected resource available after executing c

Quantitative Hoare logic

Resource available before executing c

Expected cost $\mathbb{E}(c)$

$$\{\Phi\} \quad c \; \{\Phi'\}$$

$\mathbb{E}(\Phi')$ is the expected resource available after executing c

For all states σ , $\Phi(\sigma)$ is sufficient to pay for the expected cost of executing c and the expected resource available after the execution w.r.t the distribution over next states

Example rules

(Q:PIF)

$$\frac{Q = p \cdot Q_1 + (1-p) \cdot Q_2 \qquad \vdash \{\Gamma; Q_1\} c_1 \{\Gamma'; Q'\} \qquad \vdash \{\Gamma; Q_2\} c_2 \{\Gamma'; Q'\}}{\vdash \{\Gamma; Q\} c_1 \oplus_p c_2 \{\Gamma'; Q'\}}$$

(Q:SAMPLE)

$$\frac{\begin{array}{cc} \Gamma \models R \in [a, b] & Q = \sum_i p_i \cdot Q_i \\ \forall v_i \in [a, b]. [\![\mu_R : v_i]\!] = p_i & \forall v_i. \vdash \{\Gamma; Q_i\} x = e \text{ bop } v_i \{\Gamma'; Q'\} \end{array}}{\vdash \{\Gamma; Q\} x = e \text{ bop } R \{\Gamma'; Q'\}}$$

Example rules

(Q:PIF)

$$\frac{Q = p \cdot Q_1 + (1-p) \cdot Q_2 \qquad \vdash \{\Gamma; Q_1\} c_1 \{\Gamma'; Q'\} \qquad \vdash \{\Gamma; Q_2\} c_2 \{\Gamma'; Q'\}}{\vdash \{\Gamma; Q\} c_1 \oplus_p c_2 \{\Gamma'; Q'\}}$$

Logical assertions

(Q:SAMPLE)

$$\frac{\begin{array}{cc} \Gamma \models R \in [a, b] & Q = \sum_i p_i \cdot Q_i \\ \forall v_i \in [a, b]. [\![\mu_R : v_i]\!] = p_i & \forall v_i . \vdash \{\Gamma; Q_i\} x = e \text{ bop } v_i \{\Gamma'; Q'\} \end{array}}{\vdash \{\Gamma; Q\} x = e \text{ bop } R \{\Gamma'; Q'\}}$$

Example rules

Potential functions

(Q:PIf)

$$\frac{Q = p\cdot Q_1 + (1-p)\cdot Q_2 \qquad \vdash \{\Gamma; Q_1\} c_1 \{\Gamma'; Q'\} \qquad \vdash \{\Gamma; Q_2\} c_2 \{\Gamma'; Q'\}}{\vdash \{\Gamma; Q\} c_1 \oplus_p c_2 \{\Gamma'; Q'\}}$$

Logical assertions

(Q:Sample)

$$\frac{\begin{array}{cc} \Gamma \models R \in [a,b] & Q = \sum_i p_i \cdot Q_i \\ \forall v_i \in [a,b]. [\![\mu_R : v_i]\!] = p_i & \forall v_i.\ \vdash \{\Gamma; Q_i\} x = e \text{ bop } v_i \{\Gamma'; Q'\} \end{array}}{\vdash \{\Gamma; Q\} x = e \text{ bop } R \{\Gamma'; Q'\}}$$

Example rules

Encoded as linear constraints

Potential functions

(Q:PIf)

$$\frac{Q = p\cdot Q_1 + (1-p)\cdot Q_2 \qquad \vdash \{\Gamma; Q_1\} c_1 \{\Gamma'; Q'\} \qquad \vdash \{\Gamma; Q_2\} c_2 \{\Gamma'; Q'\}}{\vdash \{\Gamma; Q\} c_1 \oplus_p c_2 \{\Gamma'; Q'\}}$$

Logical assertions

(Q:Sample)

$$\frac{\begin{array}{cc} \Gamma \models R \in [a,b] & Q = \sum_i p_i \cdot Q_i \\ \forall v_i \in [a,b]. [\![\mu_R : v_i]\!] = p_i & \forall v_i . \vdash \{\Gamma; Q_i\} x = e \text{ bop } v_i \{\Gamma'; Q'\} \end{array}}{\vdash \{\Gamma; Q\} x = e \text{ bop } R \{\Gamma'; Q'\}}$$

Example rules

Encoded as linear constraints

Potential functions

(Q:PIF)

$$\frac{Q = p \cdot Q_1 + (1-p) \cdot Q_2 \qquad \vdash \{\Gamma; Q_1\} c_1 \{\Gamma'; Q'\} \qquad \vdash \{\Gamma; Q_2\} c_2 \{\Gamma'; Q'\}}{\vdash \{\Gamma; Q\} c_1 \oplus_p c_2 \{\Gamma'; Q'\}}$$

Logical assertions

(Q:SAMPLE)

$$\frac{\Gamma \models R \in [a,b] \qquad Q = \sum_i p_i \cdot Q_i \qquad \forall v_i \in [a,b].[\![\mu_R : v_i]\!] = p_i \qquad \forall v_i. \vdash \{\Gamma; Q_i\} x = e \text{ bop } v_i \{\Gamma'; Q'\}}{\vdash \{\Gamma; Q\} x = e \text{ bop } R \{\Gamma'; Q'\}}$$

Probability that the sample value is v_i

Example rules

Encoded as linear constraints

Potential functions

(Q:PIF)

$$\frac{Q = p{\cdot}Q_1 + (1-p){\cdot}Q_2 \qquad \vdash \{\Gamma; Q_1\}c_1\{\Gamma'; Q'\} \qquad \vdash \{\Gamma; Q_2\}c_2\{\Gamma'; Q'\}}{\vdash \{\Gamma; Q\}c_1 \oplus_p c_2\{\Gamma'; Q'\}}$$

Logical assertions

(Q:SAMPLE)

$$\frac{\begin{array}{cc}\Gamma \models R \in [a,b] & Q = \sum_i p_i{\cdot}Q_i \\ \forall v_i \in [a,b].[\![\mu_R : v_i]\!] = p_i & \forall v_i. \vdash \{\Gamma; Q_i\}x = e \text{ bop } v_i\{\Gamma'; Q'\}\end{array}}{\vdash \{\Gamma; Q\}x = e \text{ bop } R\{\Gamma'; Q'\}}$$

Probability that the sample value is v_i

Distribution with finite domain

Derivation: Random walk

```
while x < n:
    prob(3,1):
        x = x + 1
    else:
        x = x - 1
    tick 1
```

Derivation: Random walk

Bound on the expected cost: 2max(0,n-x) = 2|[x,n]|

```
{ . ; 2|[x,n]| }
while x < n:
   prob(3,1):
      x = x + 1
   else:
      x = x - 1
   tick 1
```

Derivation: Random walk

Bound on the expected cost: 2max(0,n-x) = 2|[x,n]|

```
{ . ; 2|[x,n]| }
while x < n:
  prob(3,1):
    x = x + 1
  else:
    x = x - 1
  tick 1
  { . ; 2|[x,n]| }
```

Derivation: Random walk

Bound on the expected cost: 2max(0,n-x) = 2|[x,n]|

```
{ . ; 2|[x,n]| }
while x < n:

  prob(3,1):

    x = x + 1

  else:

    x = x - 1
    { x<n; 2|[x,n]| + 1 }
  tick 1
  { . ; 2|[x,n]| }
```

Derivation: Random walk

Bound on the expected cost: 2max(0,n-x) = 2|[x,n]|

```
{ . ; 2|[x,n]| }
while x < n:

   prob(3,1):

      x = x + 1

   else:
      { x<n; 2|[x,n]| + 3 }
      x = x - 1
      { x<n; 2|[x,n]| + 1 }
   tick 1
   { . ; 2|[x,n]| }
```

Derivation: Random walk

Bound on the expected cost: 2max(0,n-x) = 2|[x,n]|

```
{ . ; 2|[x,n]| }
while x < n:

  prob(3,1):

    x = x + 1
    { x<n; 2|[x,n]| + 1 }
  else:
    { x<n; 2|[x,n]| + 3 }
    x = x - 1
    { x<n; 2|[x,n]| + 1 }
  tick 1
  { . ; 2|[x,n]| }
```

Derivation: Random walk

Bound on the expected cost: 2max(0,n-x) = 2|[x,n]|

```
{ . ; 2|[x,n]| }
while x < n:

    prob(3,1):
      { x<n; 2|[x,n]| - 1 }
      x = x + 1
      { x<n; 2|[x,n]| + 1 }
    else:
      { x<n; 2|[x,n]| + 3 }
      x = x - 1
      { x<n; 2|[x,n]| + 1 }
    tick 1
    { . ; 2|[x,n]| }
```

Derivation: Random walk

Bound on the expected cost: 2max(0,n-x) = 2|[x,n]|

```
{ . ; 2|[x,n]| }
while x < n:
  { x<n; 2|[x,n]| }
  prob(3,1):
    { x<n; 2|[x,n]| - 1 }
    x = x + 1
    { x<n; 2|[x,n]| + 1 }
  else:
    { x<n; 2|[x,n]| + 3 }
    x = x - 1
    { x<n; 2|[x,n]| + 1 }
  tick 1
  { . ; 2|[x,n]| }
```

Derivation: Random walk

Bound on the expected cost: 2max(0,n-x) = 2|[x,n]|

```
{ . ; 2|[x,n]| }
while x < n:
   { x<n; 2|[x,n]| }
   prob(3,1):
      { x<n; 2|[x,n]| - 1 }
      x = x + 1
      { x<n; 2|[x,n]| + 1 }
   else:
      { x<n; 2|[x,n]| + 3 }
      x = x - 1
      { x<n; 2|[x,n]| + 1 }
   tick 1
   { . ; 2|[x,n]| }
```

Weighted sum:
3/4* (2|[x,n]| - 1) + 1/4*(2|[x,n]| + 3)

Derivation: Random walk

Bound on the expected cost: 2max(0,n-x) = 2|[x,n]|

It is the exact expected cost

```
{ . ; 2|[x,n]| }
 while x < n:
   { x<n; 2|[x,n]| }
   prob(3,1):
     { x<n; 2|[x,n]| - 1 }
     x = x + 1
     { x<n; 2|[x,n]| + 1 }
   else:
     { x<n; 2|[x,n]| + 3 }
     x = x - 1
     { x<n; 2|[x,n]| + 1 }
   tick 1
   { . ; 2|[x,n]| }
```

Weighted sum:
3/4* (2|[x,n]| - 1) + 1/4*(2|[x,n]| + 3)

Automation

Automation

- Fix potential functions as linear combinations of monomials with unknown coefficients

$$\Phi := \sum_i k_i \cdot m_i$$

$$M := 1 \mid x \mid M_1 \cdot M_2 \mid \max(0,\Phi)$$

Automation

- Fix potential functions as linear combinations of monomials with unknown coefficients

$$\Phi := \sum_i k_i \cdot m_i$$

$$M := 1 \mid x \mid M_1 \cdot M_2 \mid \max(0, \Phi)$$

- Encode the relations between the potential functions at the current and next program points as linear constraints

Automation

- Fix potential functions as linear combinations of monomials with unknown coefficients

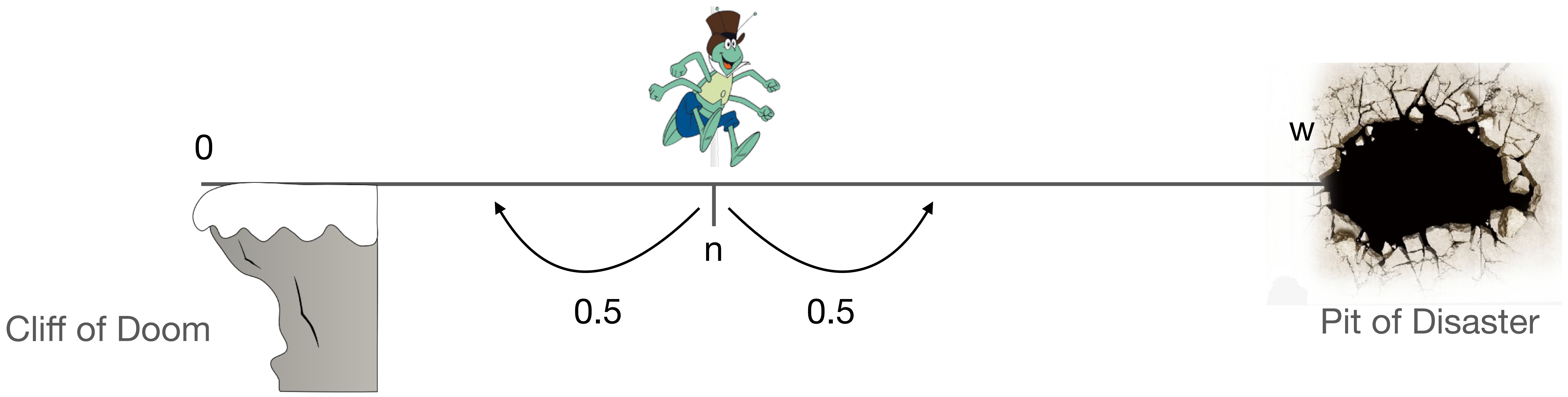
$$\Phi := \sum_i k_i \cdot m_i$$

$$M := 1 \mid x \mid M_1 \cdot M_2 \mid \max(0,\Phi)$$

- Encode the relations between the potential functions at the current and next program points as linear constraints

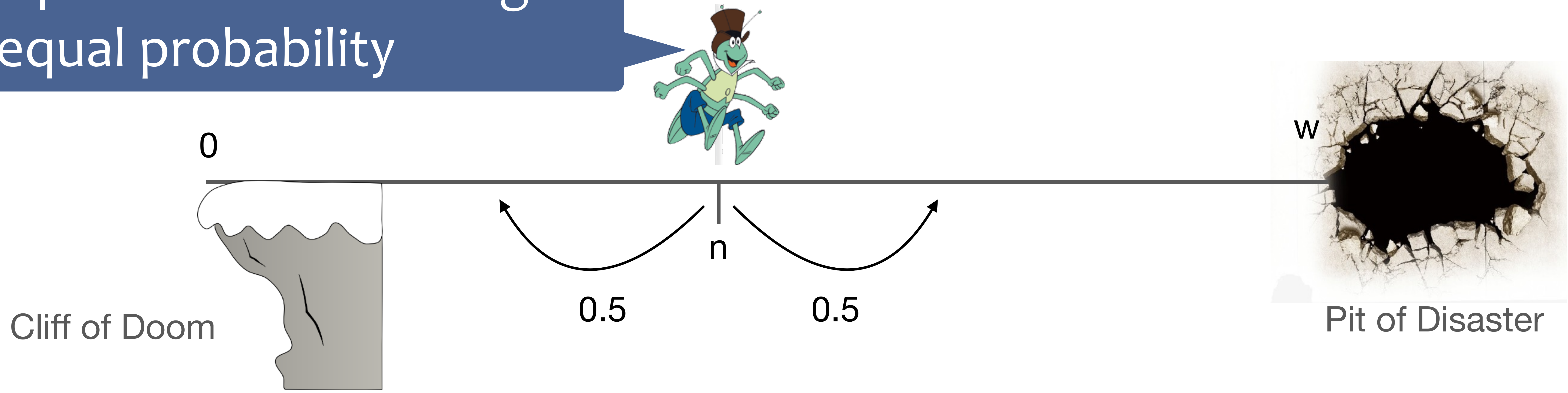
- Obtain the optimal solution by solving the generated constraints with an off-the-shelf LP solver

Example: Bug's life



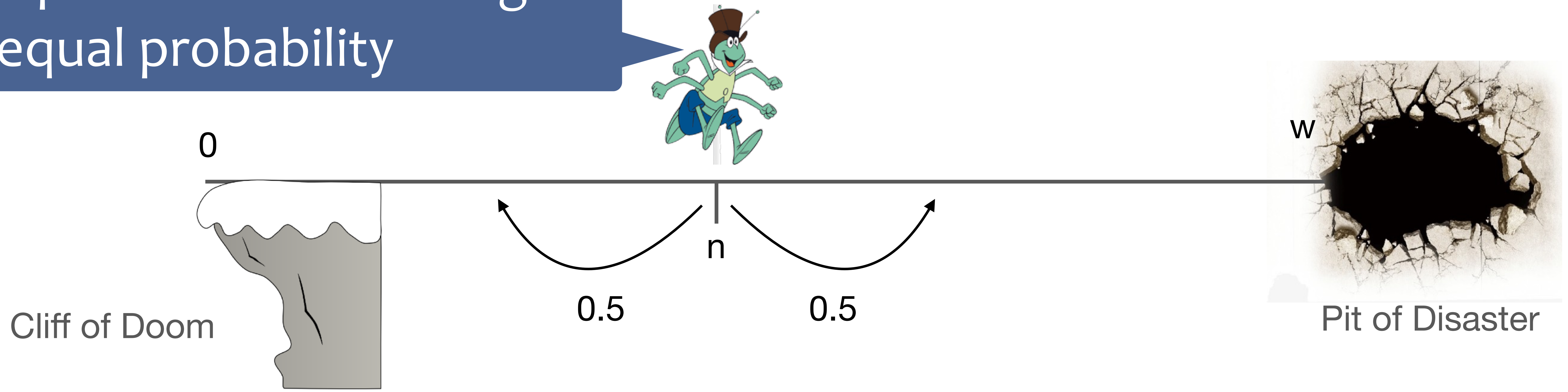
Example: Bug's life

Repeatedly hops 1 inch to left or right with equal probability



Example: Bug's life

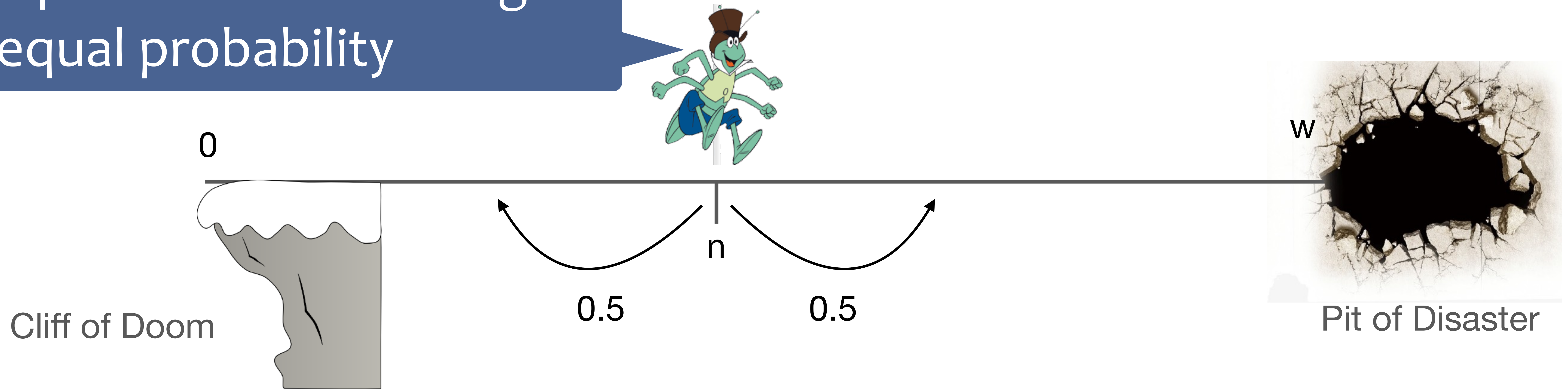
Repeatedly hops 1 inch to left or right with equal probability




```
while n > 0 && n < w:
  prob(1,1):
     n = n + 1
  else:
     n = n - 1
  tick 1
```

Example: Bug’s life

Repeatedly hops 1 inch to left or right with equal probability

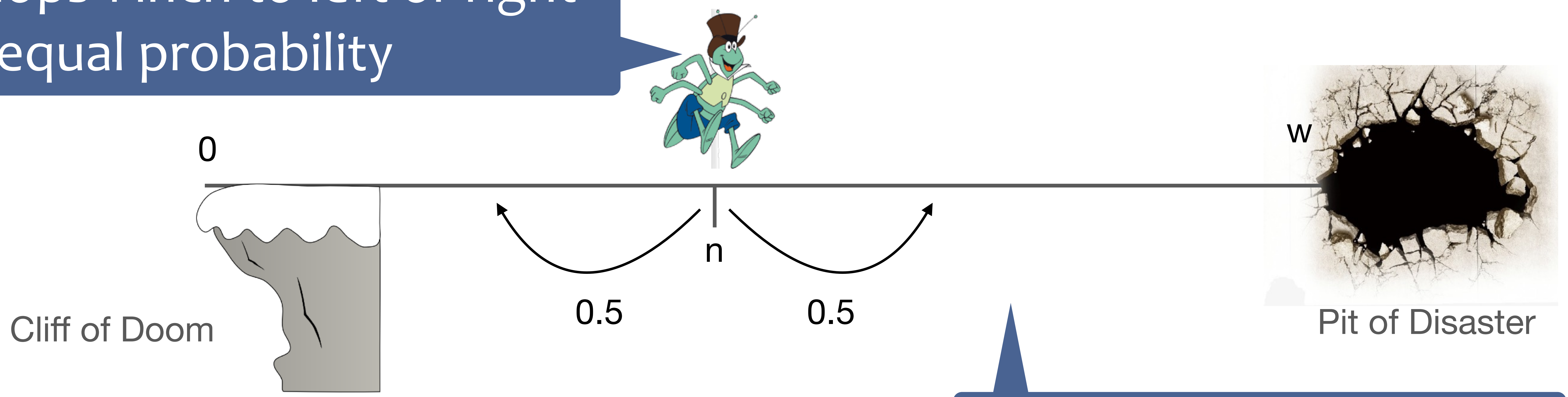



```
while n > 0 && n < w:
  prob(1,1):
     n = n + 1
  else:
     n = n - 1
  tick 1
```

Cost = # hops
= Bug’s life

Example: Bug's life

Repeatedly hops 1 inch to left or right with equal probability



The expected life:
|[0,n]| * |[n,w]|

```
while n > 0 && n < w:
  prob(1,1):
     n = n + 1
  else:
     n = n - 1
  tick 1
```

Cost = # hops
= Bug's life

Derivation: Bug’s life

```
while n > 0 && n < w:
  prob(1,1):
    n = n + 1
  else:
    n = n - 1
  tick 1
```

Derivation: Bug's life

inv(n,w) = |[0,n]|*|[n,w]|

```
{ . ; inv(n,w)}
while n > 0 && n < w:
    prob(1,1):
        n = n + 1
    else:
        n = n - 1
    tick 1
    { . ; inv(n,w)}
```

Derivation: Bug's life

inv(n,w) = |[0,n]|*|[n,w]|

```
{ . ; inv(n,w)}
while n > 0 && n < w:

   prob(1,1):

      n = n + 1

   else:

      n = n - 1
      { . ; inv(n,w) + 1}
   tick 1
   { . ; inv(n,w)}
```

Derivation: Bug's life

inv(n,w) = |[0,n]|*|[n,w]|

```
{ . ; inv(n,w)}
while n > 0 && n < w:

   prob(1,1):

      n = n + 1

   else:
      { 0<n<w; inv(n-1,w) + 1}
      n = n - 1
      { . ; inv(n,w) + 1}
   tick 1
   { . ; inv(n,w)}
```

Derivation: Bug's life

inv(n,w) = |[0,n]|*|[n,w]|

```
{ . ; inv(n,w)}
while n > 0 && n < w:

   prob(1,1):

      n = n + 1

   else:
      { 0<n<w; inv(n,w)-|[n,w]|+|[0,n]| }
      n = n - 1
      { . ; inv(n,w) + 1}
   tick 1
   { . ; inv(n,w)}
```

Derivation: Bug's life

inv(n,w) = |[0,n]|*|[n,w]|

```
{ . ; inv(n,w)}
while n > 0 && n < w:

  prob(1,1):

     n = n + 1
     { . ; inv(n,w) + 1}
  else:
     { 0<n<w; inv(n,w)-|[n,w]|+|[0,n]| }
     n = n - 1
     { . ; inv(n,w) + 1}
  tick 1
  { . ; inv(n,w)}
```

Derivation: Bug's life

inv(n,w) = |[0,n]|*|[n,w]|

```
{ . ; inv(n,w)}
while n > 0 && n < w:

   prob(1,1):
      { 0<n<w; inv(n+1,w) + 1 }
      n = n + 1
      { . ; inv(n,w) + 1}
   else:
      { 0<n<w; inv(n,w)-|[n,w]|+|[0,n]| }
      n = n - 1
      { . ; inv(n,w) + 1}
   tick 1
   { . ; inv(n,w)}
```

Derivation: Bug's life

$inv(n,w) = |[0,n]| * |[n,w]|$

```
{ . ; inv(n,w)}
while n > 0 && n < w:

    prob(1,1):
      { 0<n<w; inv(n,w)+|[n,w]|-|[0,n]| }
      n = n + 1
      { . ; inv(n,w) + 1}
    else:
      { 0<n<w; inv(n,w)-|[n,w]|+|[0,n]| }
      n = n - 1
      { . ; inv(n,w) + 1}
    tick 1
    { . ; inv(n,w)}
```

Derivation: Bug's life

inv(n,w) = |[0,n]|*|[n,w]|

```
{ . ; inv(n,w)}
while n > 0 && n < w:
    { 0<n<w; inv(n,w) }
  prob(1,1):
    { 0<n<w; inv(n,w)+|[n,w]|-|[0,n]| }
    n = n + 1
    { . ; inv(n,w) + 1}
  else:
    { 0<n<w; inv(n,w)-|[n,w]|+|[0,n]| }
    n = n - 1
    { . ; inv(n,w) + 1}
  tick 1
  { . ; inv(n,w)}
```

Weighted sum in which terms are canceled out

Implementation: Absynth

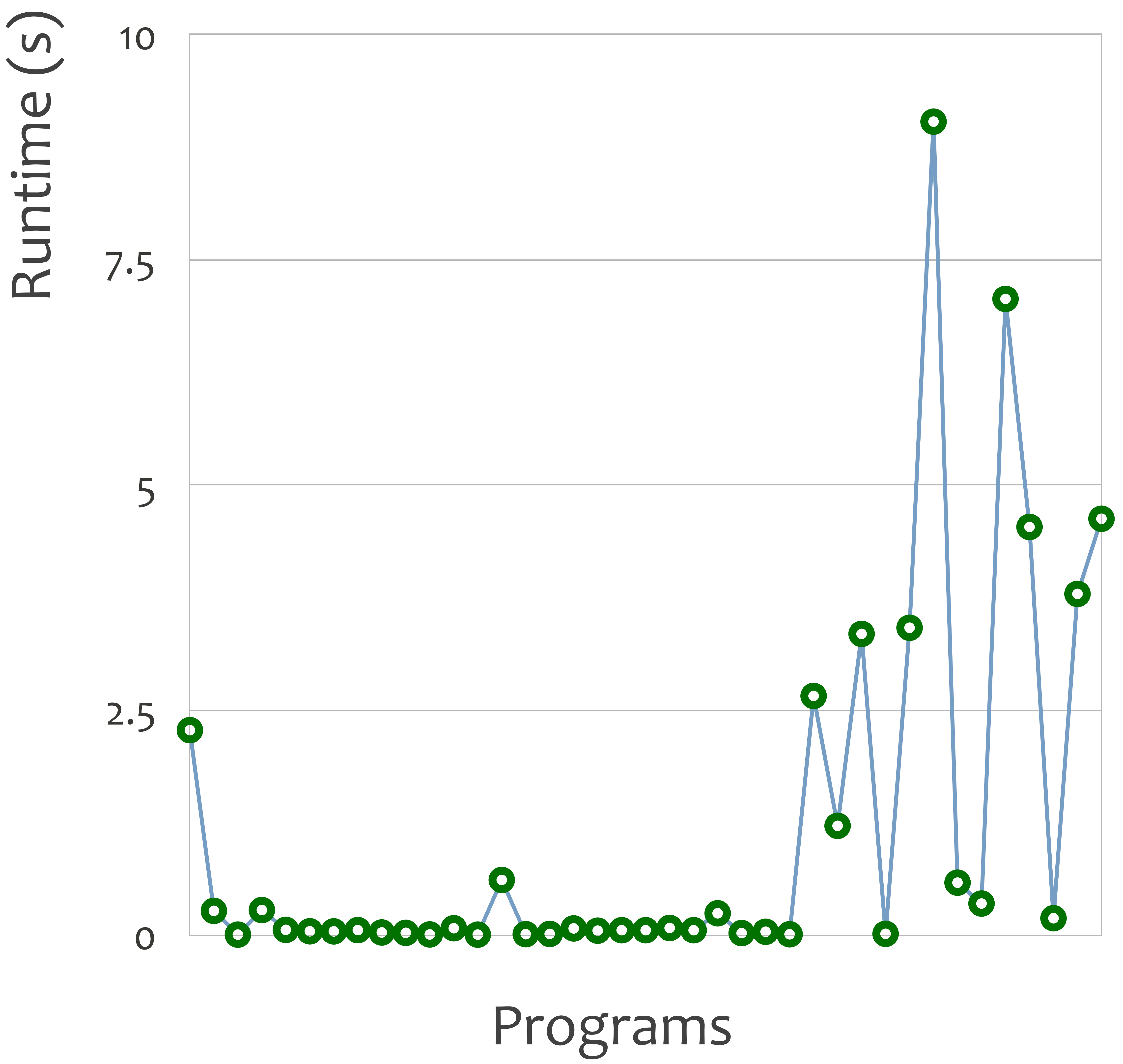
- Accepts imperative (integer) probabilistic programs

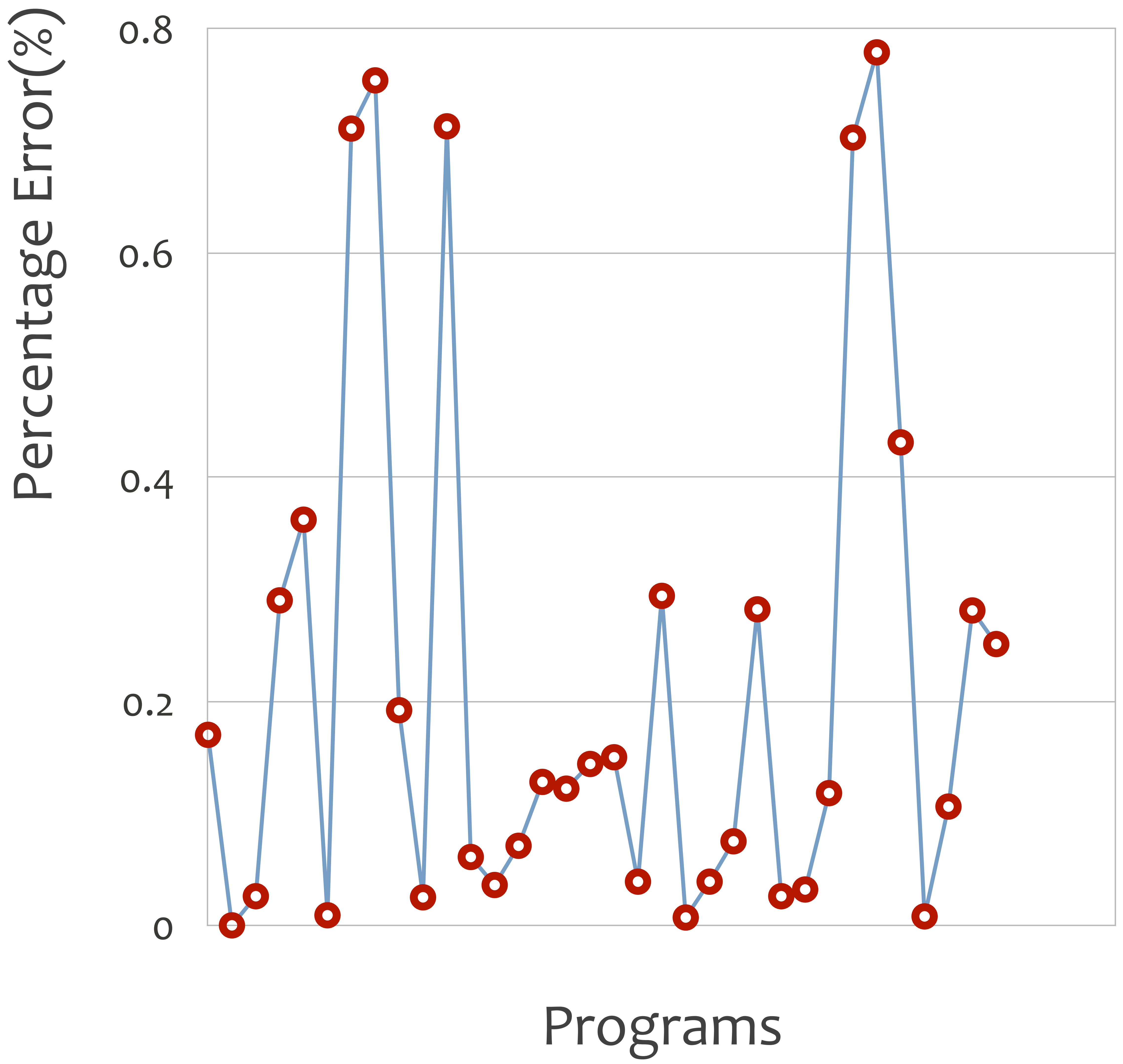
- Infers multivariate polynomial bounds on the expected resource consumption

- Automatically analyzes 40 challenging probabilistic programs and randomized algorithms with different looping patterns

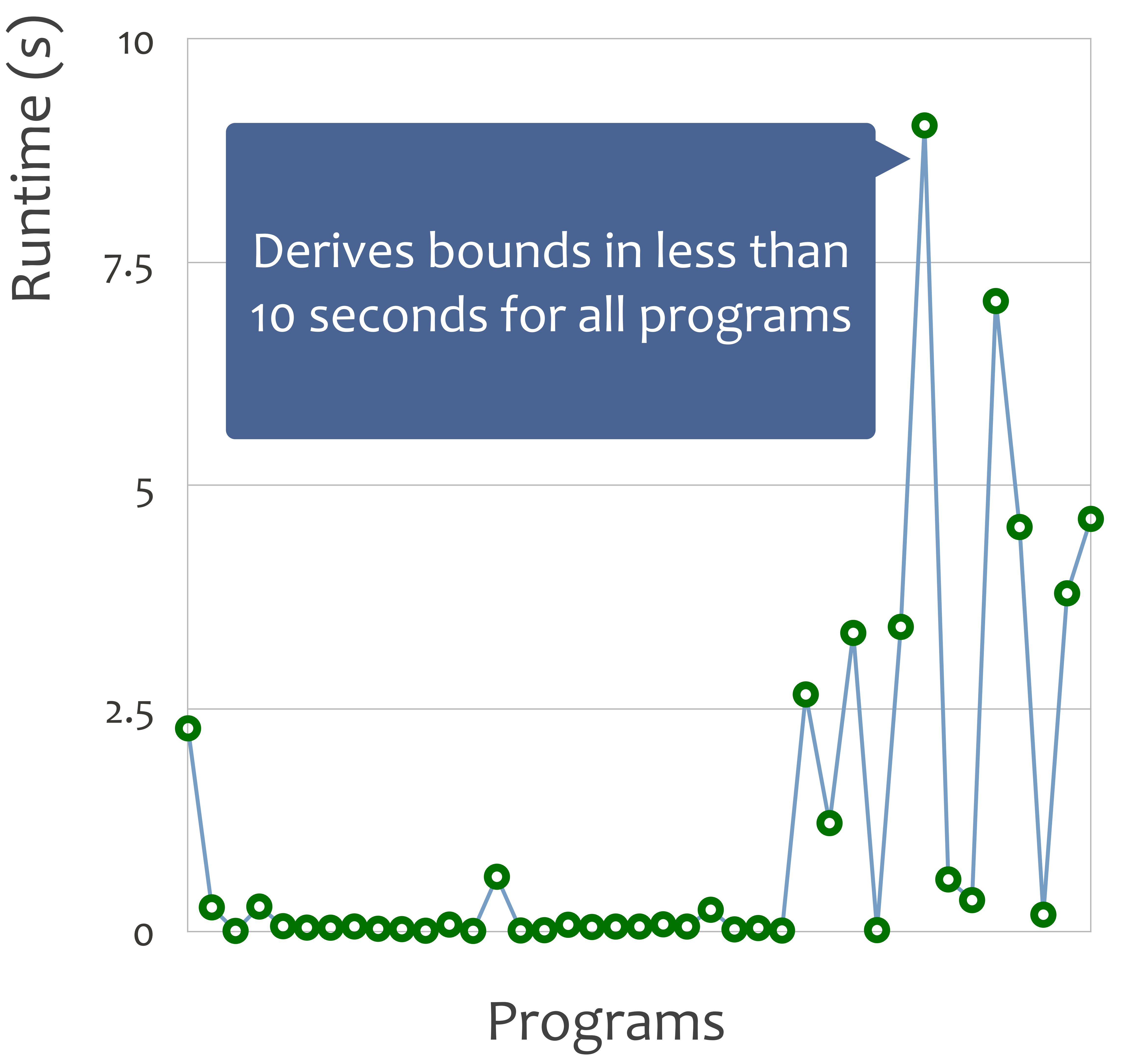
- Statically derived bounds are compared with simulation-based expectations to show that constant factors are very precise

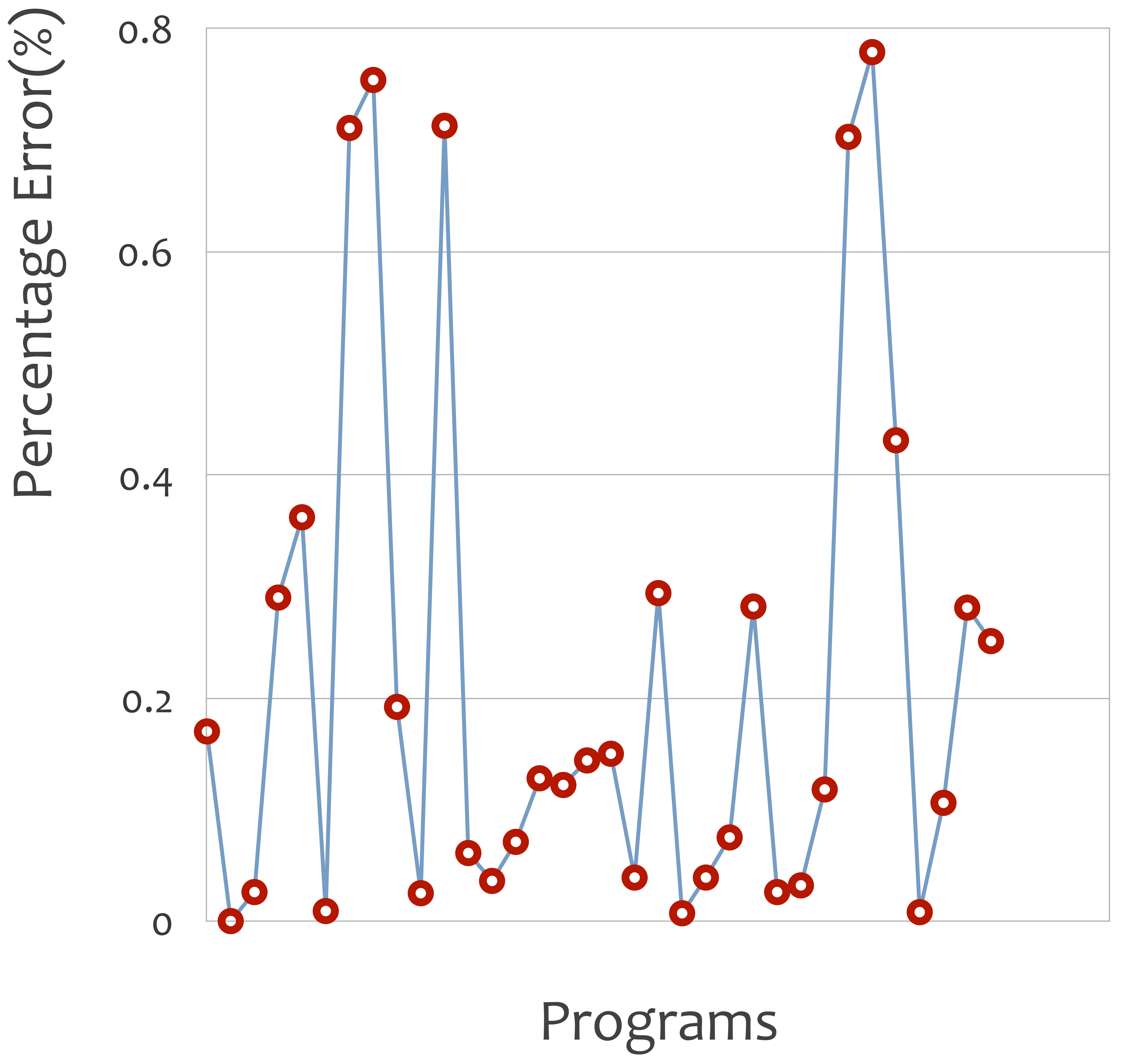
Experiments: Overview



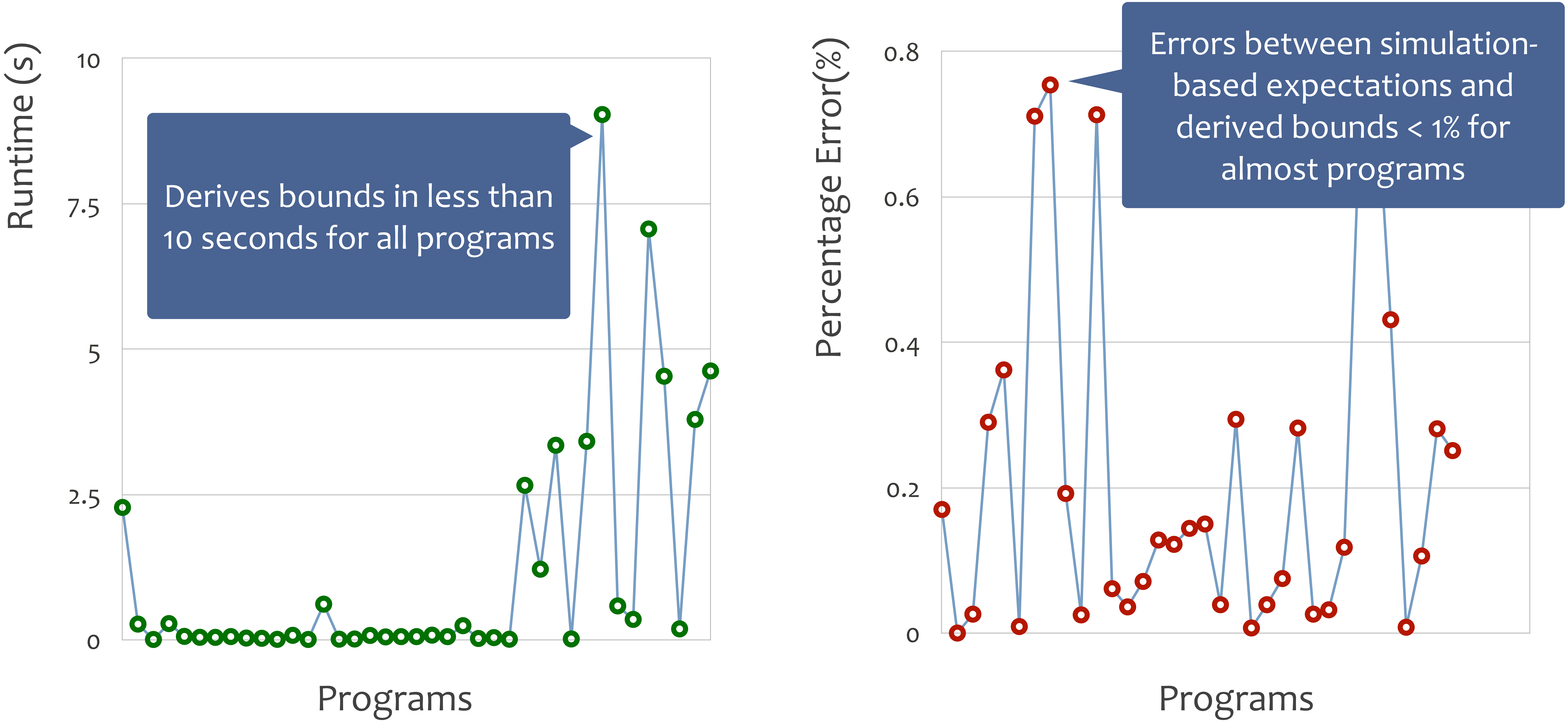


Experiments: Overview



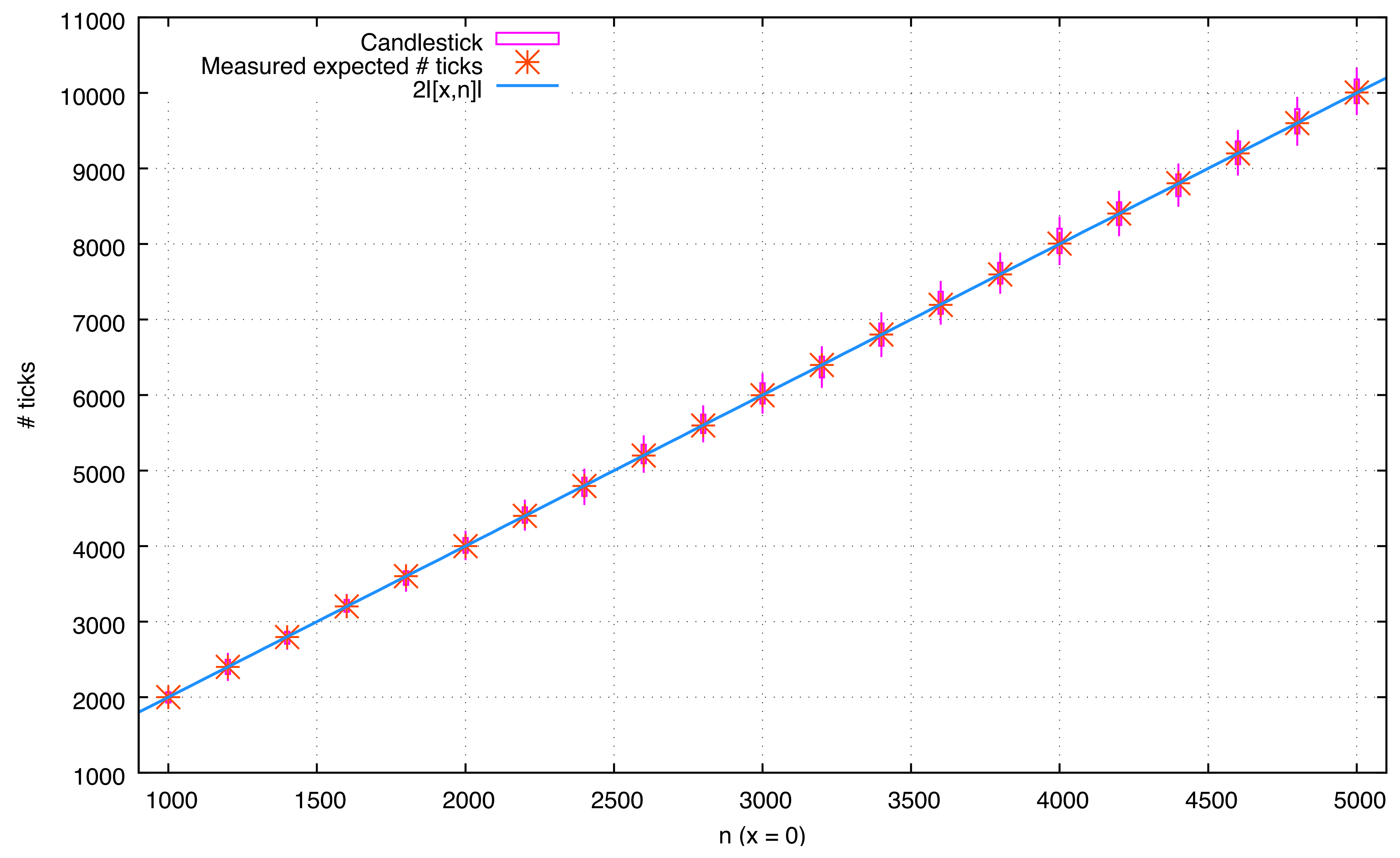


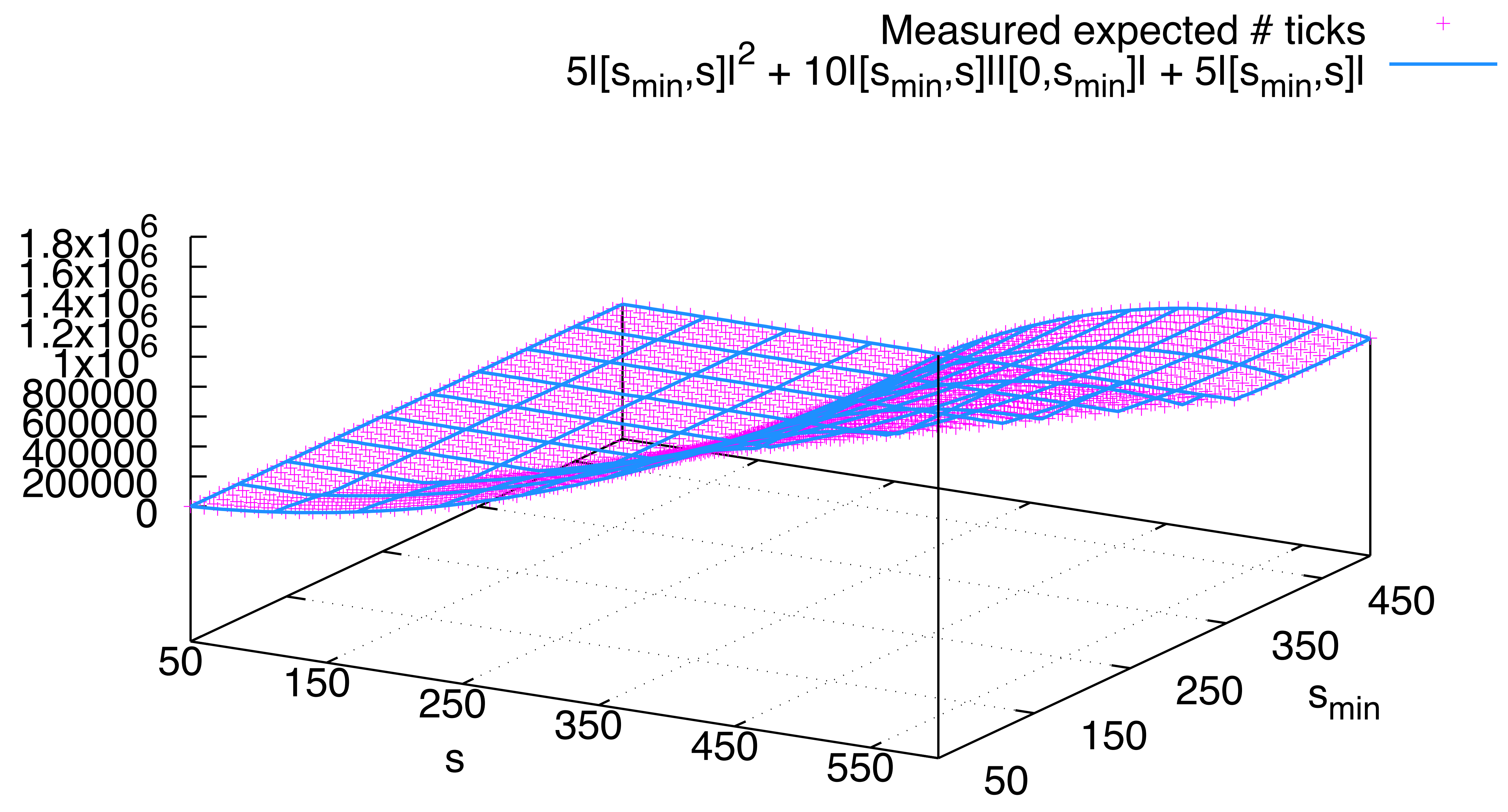
Experiments: Overview



Precise constant factors

- For example, figures show the constant factors in derived bounds for random walk and polynomial programs are very precise

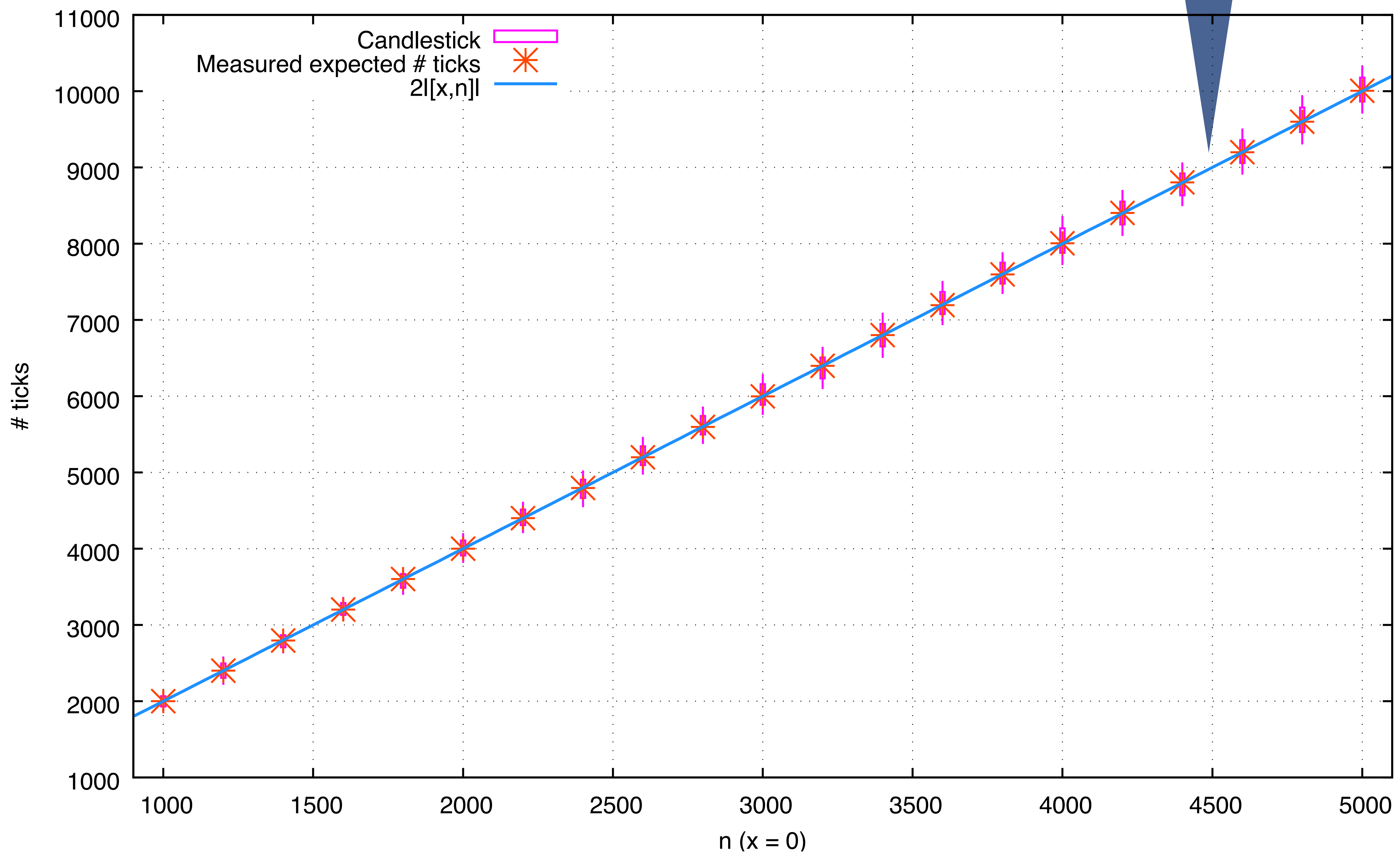


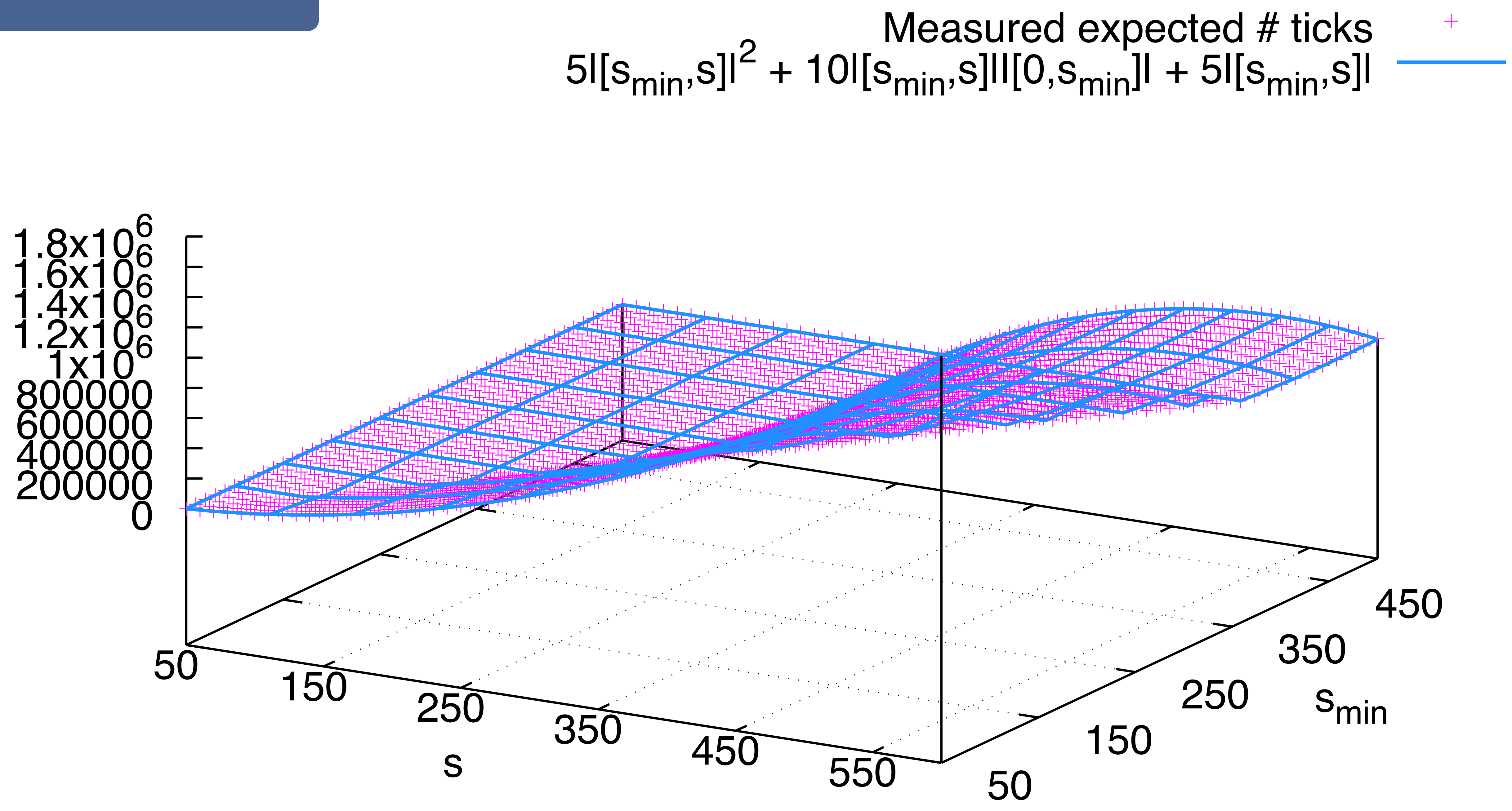


Precise constant factors

- For example, figures show the constant factors in derived bounds for random walk and polynomial programs are very precise

Blue lines are plotting of derived bounds

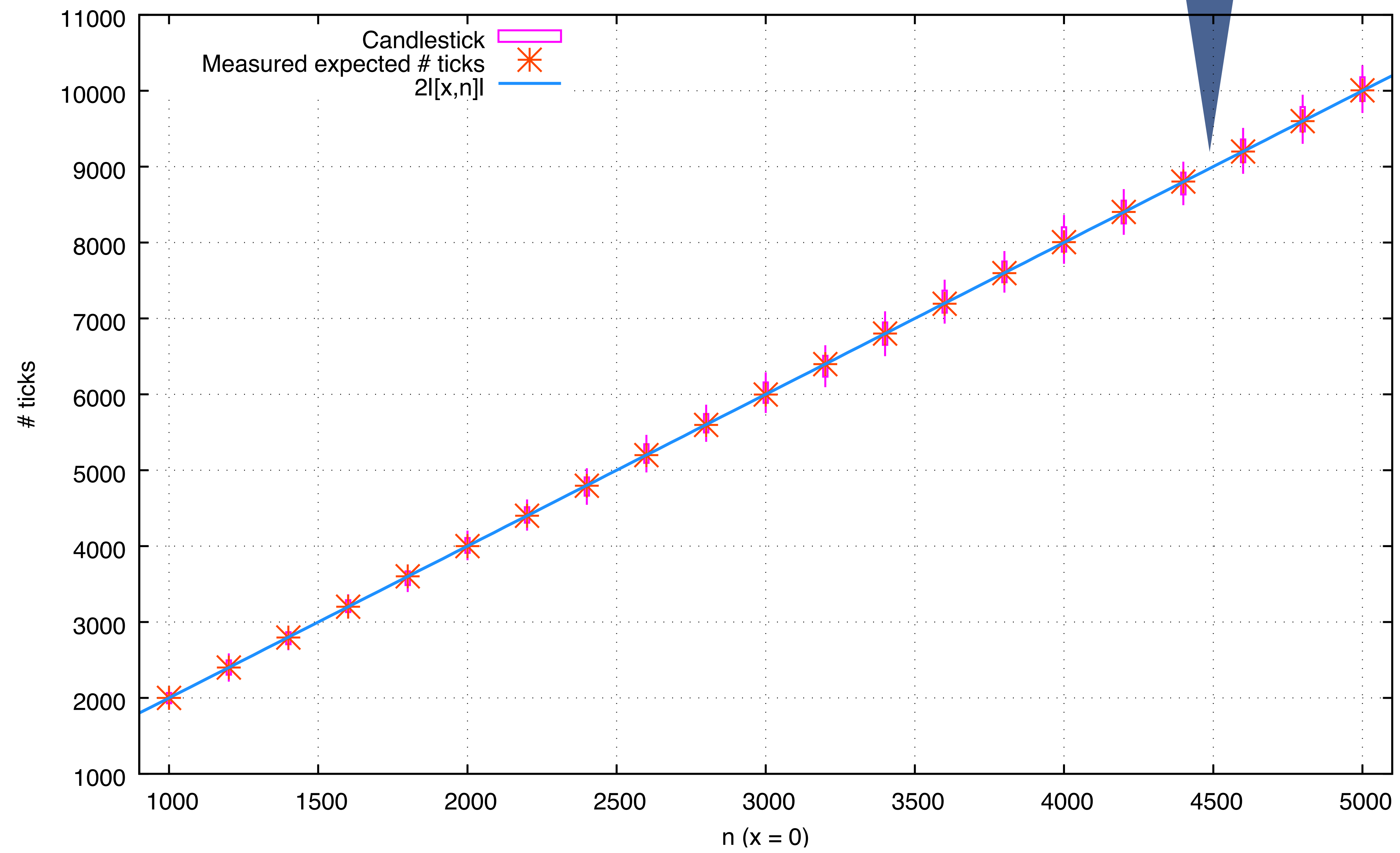




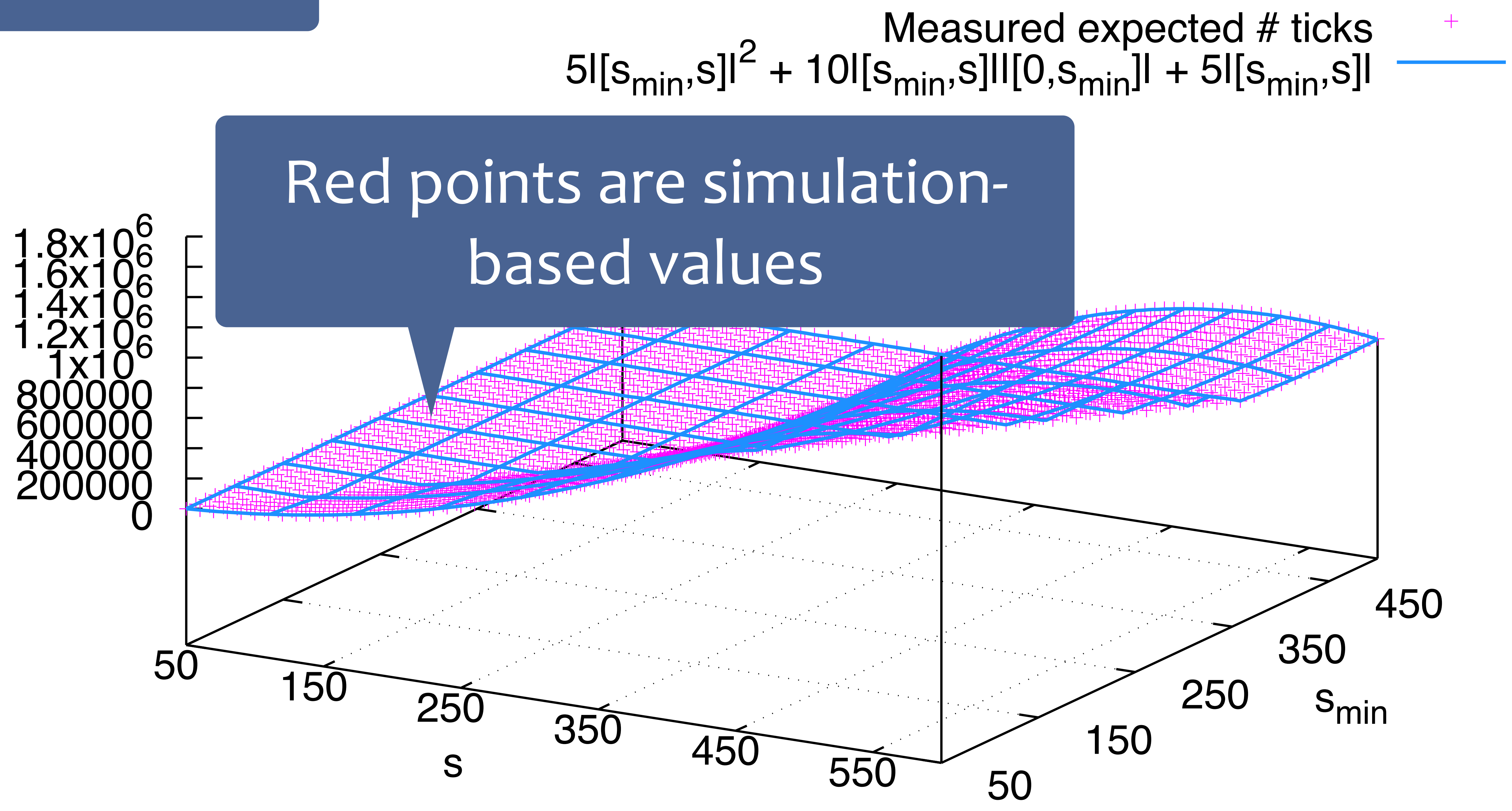
Precise constant factors

- For example, figures show the constant factors in derived bounds for random walk and polynomial programs are very precise

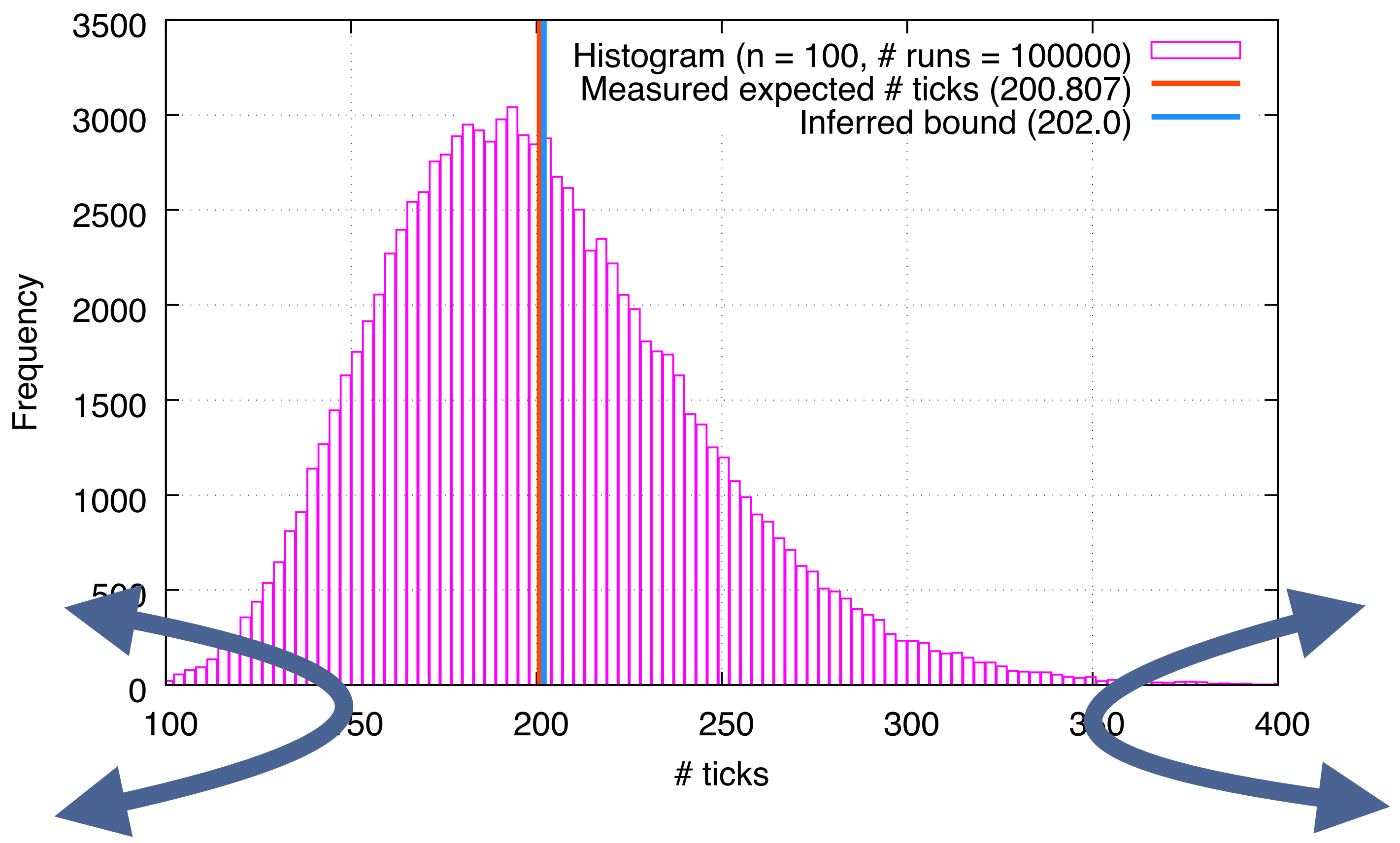
Blue lines are plotting of derived bounds



Red points are simulation-based values

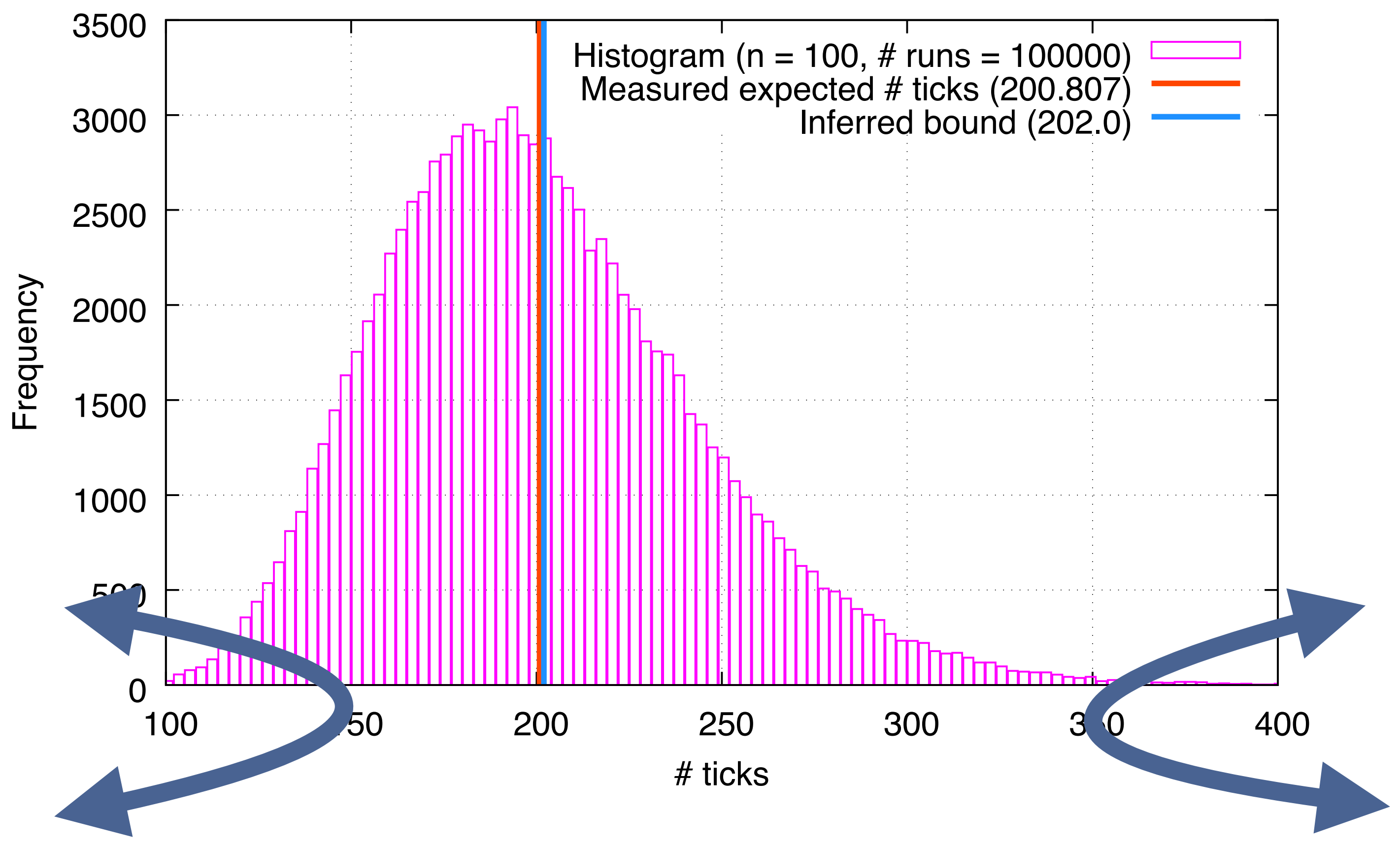


Application: Tail-bound analysis



- Can be reduced to expected resource analysis using concentration inequalities (e.g., Markov and Chebyshev's inequalities)
- Assert that resource usage is bounded with a high probability
- Thus, they are good for analyzing safety properties of programs

Application: Tail-bound analysis



- Can be reduced to expected resource analysis using concentration inequalities (e.g., Markov and Chebyshev's inequalities)
- Assert that resource usage is bounded with a high probability
- Thus, they are good for analyzing safety properties of programs

Random walk example:

$$\mathbb{P}(t \geq 10|[0,n]|)$$

$$\leq \frac{\mathbb{E}(t)}{10|[0,n]|} \leq \frac{2|[0,n]|}{10|[0,n]|} = 0.2$$

Summary

Summary

Contributions

Summary

Contributions

- First automatic analysis for deriving symbolic bounds on the expected resource usage
- Practical implementation for imperative (integer) probabilistic programs

Summary

Contributions

- First automatic analysis for deriving symbolic bounds on the expected resource usage
- Practical implementation for imperative (integer) probabilistic programs

Limitations

- Non-polynomial bounds
- Discrete distributions with finite domains

Summary

Contributions

- First automatic analysis for deriving symbolic bounds on the expected resource usage
- Practical implementation for imperative (integer) probabilistic programs

Future work

- Lower bounds on the expected resource usage
- Tail-bound analysis with Chebyshev’s inequality

Limitations

- Non-polynomial bounds
- Discrete distributions with finite domains