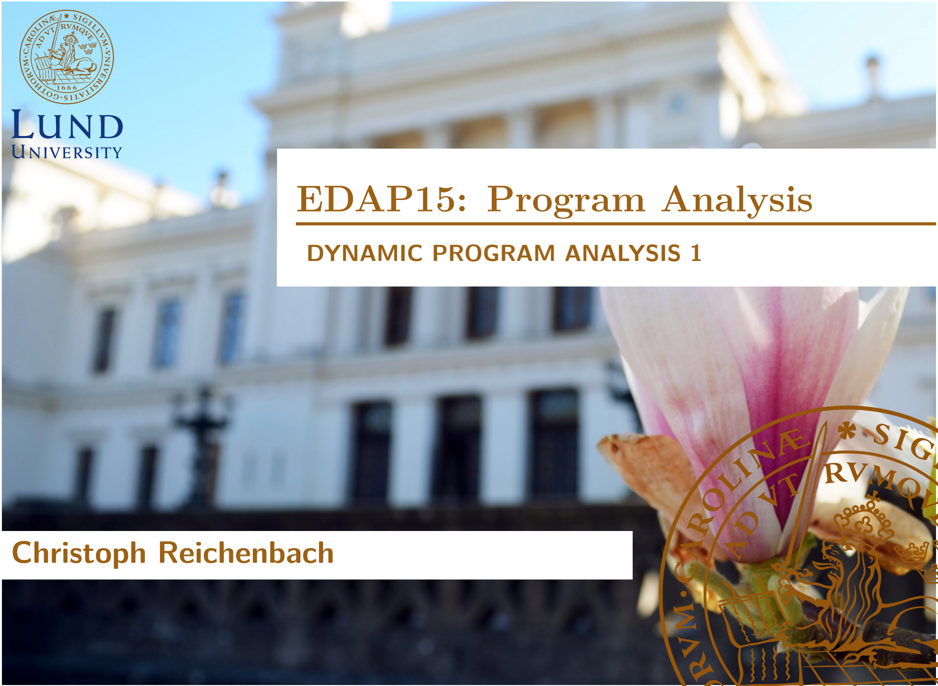


LUND UNIVERSITY

EDAP15: Program Analysis

DYNAMIC PROGRAM ANALYSIS 1

Christoph Reichenbach

Welcome back!

- Moodle notification mails seem back online

Questions?

Analysing Realistic Programs

Challenges:

- **Semantics**:
  - Language semantics may be imprecisely defined (e.g., custom or domain-specific languages)
  - Certain **language features** intrinsically hard to analyse
- Non-Semantic Properties:
  - Property of interest may not be part of semantics
  - Examples: execution time, energy usage

Reflection

Java

```
Class<?> cl = Class.forName(string);
Object obj = cl.getConstructor().newInstance();
System.out.println(obj.toString());
```

- Instantiates object by string name
- Similar features to call method by name
- **Challenge**:
  - `obj` may have *any* type ⇒ imprecision
  - Sound call graph construction very conservative
- **Approaches**
  - Dataflow: what strings flow into `string`?
    - Common: code draws from finite set or uses string prefix/suffix (e.g., ("com.x.plugins." + ...))
    - `Class.forName`: class only from some point in package hierarchy
  - Dynamic analysis

Dynamic Loading

C

```
handle = dlopen("module.so", RTLD_LAZY);
op = (int (*)(int)) dlsym(handle, "my_fn");
```

- Dynamic library and class loading:
  - Add new code to program that was not visible at analysis time
- **Challenge**:
  - Can't analyse what we can't see
- **Approaches**:
  - Conservative approximation
    - Tricky: External code may modify *all that it can reach*
  - With dynamic support and static annotation:
  - Allow only loading of signed/trusted code
    - signature must guarantee properties we care about
    - annotation provides properties to static analysis
  - *Proof-carrying code*
    - Code comes with proof that we can check at run-time

Native Code

Java

```
class A {
  public native Object op(Object arg);
}
```

- High-level language invokes code written in low-level language
  - Usually C or C++
  - May use nontrivial interface to talk to high-level language
- **Challenge**:
  - High-level language analyses don't understand low-level language
- **Approaches**:
  - Conservative approximation
    - Tricky: External code may modify *anything*
  - Manually model known native operations (e.g., Doop)
  - Multi-language analysis (e.g., Graal)

‘eval’ and dynamic code generation

Python

```
eval(raw_input())
```

- Execute a string as if it were part of the program
- **Challenge**:
  - Cannot predict contents of string in general
- **Approaches**:
  - Conservative approximation
    - Tricky: code may modify *anything*
  - Dynamically re-run static analysis
  - Special-case handling (cf. reflection)

Summary

- Static program analysis faces significant challenges:
  - **Decidability** requires lack of precision or soundness for most of the interesting analyses
  - **Reflection** allows calling methods / creating objects given by arbitrary string
  - **Dynamic module loading** allows running code that the analysis couldn't inspect ahead of time
  - **Native code** allows running code written in a different language
  - **Dynamic code generation** and `eval` allow building arbitrary programs and executing them
  - No universal solution
  - Can try to 'outlaw' or restrict problematic features, depending on goal of analysis
  - Can combine with dynamic analyses

Soundiness

- Can't analyse language feature?

$\Rightarrow$ We get $\top$ if we want soundness

$\Rightarrow$ Potentially many false positives

$\Rightarrow$ Tool may be useless

  - Google SWE practice: Bug checkers with $> 5\%$ false positives disabled automatically
- Soundness may not be *useful*
- Alternative proposal from research community: **Soundiness**
  - *Be explicit* about unsupported language features
  - Example: "Sound unless the code uses features X, Y, Z"

Soundiness: "capture all dynamic behaviour *within reason*"

B. Livshits, M. Sridharan, Y. Smaragdakis et al.: "In defense of Soundiness: A Manifesto", Communications of the ACM, 2015

Lecture Overview

Foundations

- 01 Foundations
- 02 Constructing Program Analyses in JastAdd

Static Analysis

Properties

- 03 04 Types
- 05 06 07 Data Flow
- 08 09 Memory

Control Flow

- 05 Intraprocedural
- 10 Interprocedural
- 11 Indirect

Dynamic Analysis

- 12 **Instrumentation**
- 13 Analysis

14 Review

Static Analysis: Limitations

Static program analysis challenges:

- Semantics:
  - *hard* to be sound / precise
- **Non-semantic properties**:
  - Underspecified in language specification
  - May be machine/implementation-dependent
  - Examples:
    - Resource usage
    - Execution time
    - Latency
    - Throughput

    . . .

Dynamic Analysis can help with both

More Difficulties for Static Analysis

- Does a certain piece of code actually get executed?
- How long does it take to execute this piece of code?
- How important is this piece of code in practice?
- How well does this code collaborate with hardware devices?
  - Harddisks?
  - Networking devices?
  - *Caches* that speed up memory access?
  - *Branch predictors* that speed up conditional jumps?
  - The *ALU(s)* that perform arithmetic in the CPU?
  - The *TLB* that helps look up memory?

  ...

Impossible to predict for all practical situations

Static vs. Dynamic Program Analyses

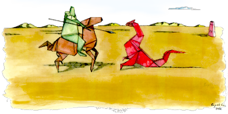
	Static Analysis	Dynamic Analysis
Examines	Program structure	Program execution
Input	Independent	Dependent
Hardware/OS	Independent	Dependent (for some properties)
Perspective	Sees anything that *could* happen	Sees that which *does* happen
False Negatives	*Avoidable*	Need all possible inputs
False Positives	Unavoidable	*Avoidable*

jastadd

Soot

Valgrind

daikon



Summary

- Static analysis has key limitations:
  - *Information missing from code* (cf. *Soundiness*)
  - *Dependency on hardware details* (e.g. *Execution Time*))
- This limits:
  - Optimisation: *which optimisations are worthwhile?*
  - Bug search: *which potential bugs are 'real'?*
- Can use *dynamic analysis* to examine run-time behaviour

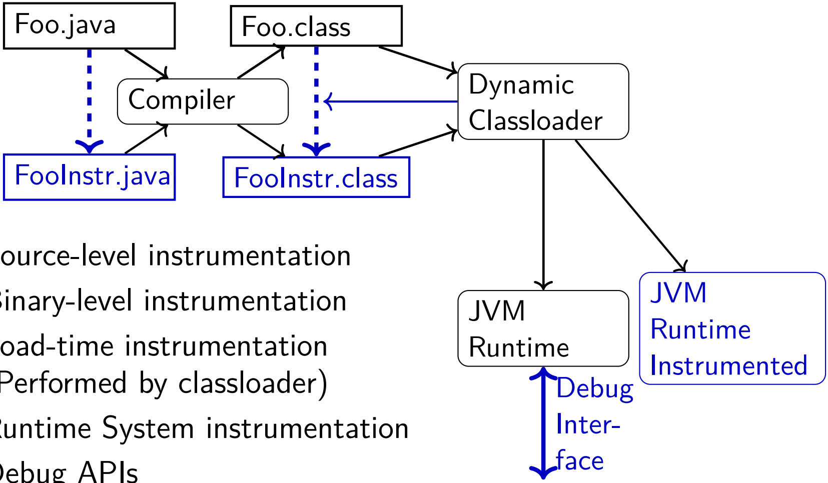
Probes

- **Probes**: devices for measuring property of interest
  - Software probe: code artefact
  - Hardware probe: physical device
- CPU, OS kernel etc. come with probes preinstalled
  - Generally need to be flipped on
- Want to probe custom location / property:
  - **Instrumentation**: insert new probes

Gathering Dynamic Data

- **Instrumentation and Software Probes**
- Simulation
- Hardware Probes

Gathering Dynamic Data: Java



- Source-level instrumentation
- Binary-level instrumentation
- Load-time instrumentation (Performed by classloader)
- Runtime System instrumentation
- Debug APIs

Comparison of Approaches

- **Source-level instrumentation**:
  - \+ Flexible
  - − Must handle syntactic issues, name capture, . . .
  - − Only applicable if we have all source code
- **Binary-level instrumentation**:
  - \+ Flexible
  - − Must handle binary encoding issues
  - − Only applicable if we know what binary code is used
- **Load-time instrumentation**:
  - \+ Flexible
  - \+ Can handle even unknown code
  - − Requires run-time support, may clash with custom loaders
- **Runtime system instrumentation**:
  - \+ Flexible
  - \+ Can see everything (gc, JIT, . . .)
  - − Labour-intensive and error-prone
  - − Becomes obsolete quickly as runtime evolves
- **Debug APIs**:
  - \+ Typically easy to use and efficient
  - − Limited capabilities

Instrumentation Tools

	C/C++ (Linux)	**Java**
Source-Level	C preprocessor, DMCE	ExtendJ
Binary Level	`pin`, `llvm`	`soot`, `asm`, `bcel`, AspectJ, ExtendJ
Load-time	?	Classloader, AspectJ
Debug APIs	`strace`	JVMTI

- Low-level data gathering:
  - Command line: `perf`
  - Time: `clock_gettime()` / `System.nanoTime()`
  - Process statistics: `getrusage()`
  - Hardware performance counters: PAPI

Practical Challenges in Instrumentation

- *Measuring*:
  - Need access to relevant data (e.g., Java: source code can't access JIT internal)
  - May need to insert **software probes** (measuring device)
- *Representing (optional)*:
  - Store data in memory until it can be emitted (optional)
  - May use memory, execution time, *perturb measurements*
- *Emitting*:
  - Write measurements out for further processing
  - May use memory, execution time, *perturb measurements*

Summary

- Different **instrumentation strategies**:
  - Instrument **source code** or **binaries**
  - Instrument **statically** or **dynamically**
  - Instrument **input program** or **runtime system**
- Challenges when handling analysis:
  - **In-memory representation of measurements** (for compression or speed)
  - **Emitting measurements**

Unit Tests

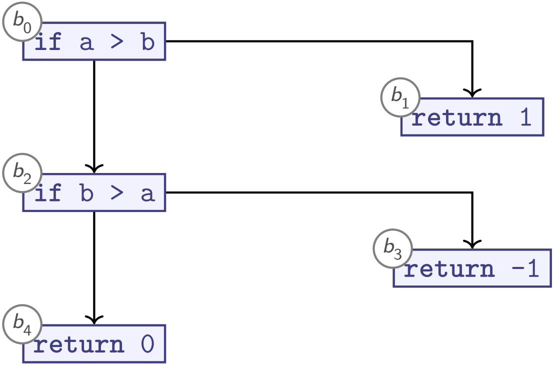
Teal

```
fun cmp(a, b) = {
  if a > b {
    return 1;
  }
  if a < b {
    return -1;
  }
  return 0;
}

fun test() = {
  assert cmp(1, 2) == -1;
  assert cmp(2, 1) == 1;
}
```

Unit tests are a simple form of dynamic program analysis

Unit Test Quality



b_0 `if a > b`

b_1 `return 1`

b_2 `if b > a`

b_3 `return -1`

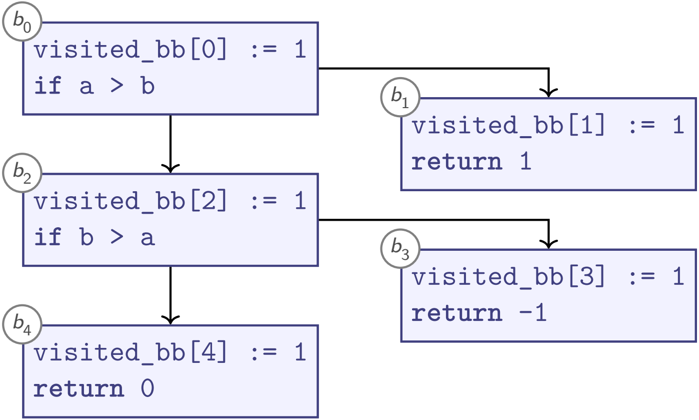
b_4 `return 0`

Teal

```
fun test() = {
  assert cmp(1, 2) == -1;
  assert cmp(2, 1) == 1;
}
```

Have I tested all behaviours?

Test Coverage



b_0

```
visited_bb[0] := 1
if a > b
```

b_1

```
visited_bb[1] := 1
return 1
```

b_2

```
visited_bb[2] := 1
if b > a
```

b_3

```
visited_bb[3] := 1
return -1
```

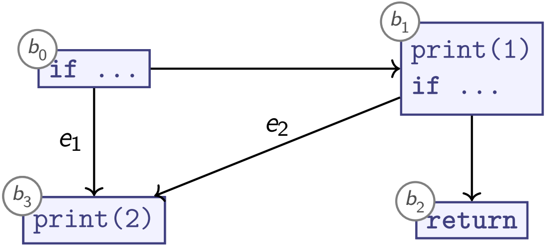
b_4

```
visited_bb[4] := 1
return 0
```

- Test coverage = fraction of `visited_bb` elements updated

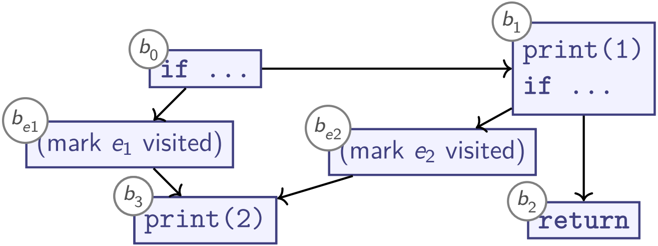
Test Coverage Properties

- **Statement Coverage**: % of executed CFG nodes
  or "Basic Blocks" of contiguous non-branching operations
  - Mark nodes/blocks as visited while testing
- **Edge Coverage**: % of taken CFG edges
  - Challenge: distinguish *edges* e_1 from e_2?



Test Coverage Properties

- **Statement Coverage**: % of executed CFG nodes
  or "Basic Blocks" of contiguous non-branching operations
  - Mark nodes/blocks as visited while testing
- **Edge Coverage**: % of taken CFG edges
  - Challenge: distinguish *edges* e_1 from e_2?



- **Path Coverage**: % of CFG paths
  - Must limit iterations
  - Must restart tracking block coverage on every method entry

Summary

- **Unit Tests** are a simple form of dynamic program analysis
  - Minimal tooling needed
  - Custom checks
  - Limited to what underlying language can express directly
- **Test Coverage** tells us how much of our code gets analysed by at least one unit test
- Implement by setting markers on relevant CFG nodes / blocks
  - Source-level: e.g. via DMCE (C/C++)
  - Binary-level: e.g. via JaCoCo/JCov (Java)
- Different criteria, such as:
  - **Statement Coverage**
  - **Edge Coverage**: may require helper CFG nodes
  - **Path Coverage**: paths through CFG (usually excluding loops)

General Data Collection

- *Probes*: How we measure
- *Events*: When we measure
- *Characteristics*: What we measure
- *Measurements*: Individual observations
- *Samples*: Collections of measurements

Events

- Subroutine call
- Subroutine return
- Memory access (read or write or either)
- System call
- Page fault

...

Characteristics

- *Value*: What is the type / numeric value / ...?
- *Counts*: How often does this event happen?
- *Wallclock times*: How long does one event take to finish, end-to-end?

Derived properties:

- *Frequencies*: How often does this happen
  - Per run
  - Per time interval
  - Per occurrence of another event
- *Relative execution times*: How long does this take
  - As fraction of the total run-time
  - As fraction of some surrounding event

Perturbation

Example challenge: can we use total counts to decide *whether* to optimise some function `f`?

- On each method entry: get current time
- On each method exit: get time again, update aggregate
- Reading timer takes: ~ 80 cycles
- Short `f` calls may be much faster than 160 cycles
  - `fun f(x) = x + 1 // ca.  0.25 cycles`
  - `fun f(x) = x     // ca.  0 cycles`
- Also: measurement needs CPU registers
  $\Rightarrow$ may require registers
  $\Rightarrow$ may slow down code further

1 GHz CPU: 1 cycle $= 10^{-9}s$ (1 nanosecond / ns)

Measurements perturb our results, slow down execution

Sampling

Alternative to full counts: *Sampling*

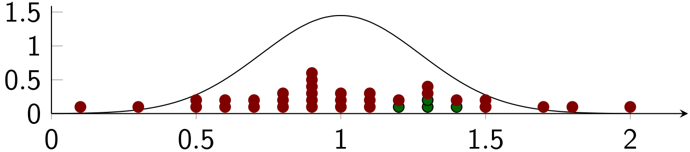
- Periodically interrupt program and measure
- Problem: how to pick the right period?

1. System events (e.g., GC trigger or ‘safepoint’)
   System events may bias results
2. Timer events: periodic intervals
    - May also bias results for periodic applications
    - Randomised intervals can avoid bias
    - Short intervals: perturbation, slowdown
    - Long intervals: imprecision

Samples and Measurements

Samples are *collections of measurements*

- Bigger samples:
  - Typically give more precise answers
  - May take longer to collect
- Challenge: representative sampling



Carefully choose what and how to sample

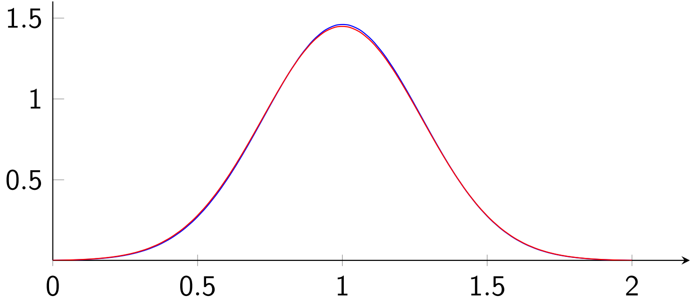
Summary

- We measure **Characteristics** of **Events**
- **Sample**: set of **Measurements** (of characteristics of events)
- Measurements often cause **perturbation**:
    - Measuring disturbs characteristics
    - Not relevant for all measurements
    - Measuring time: more relevant the smaller our time intervals get
- Can measure by:
    - **Counting**: observe every event
        - Gets all events
        - Maximum measurement perturbation
    - **Sampling**: periodically measure
        - Misses some events
        - Reduces perturbation

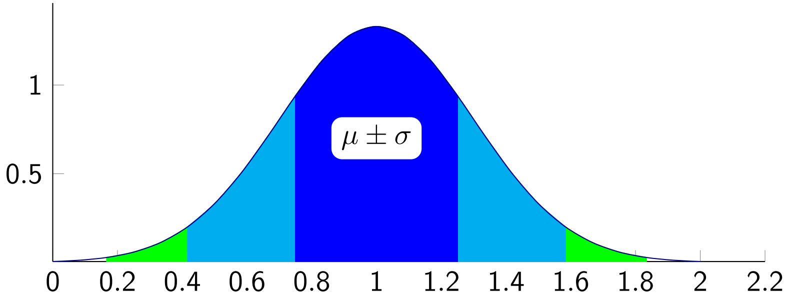
Presenting Measurements

	P1	**P2**
Mean μ	1,001	0,999
Standard Deviation σ	0,273	0,275

Assuming normal distribution:



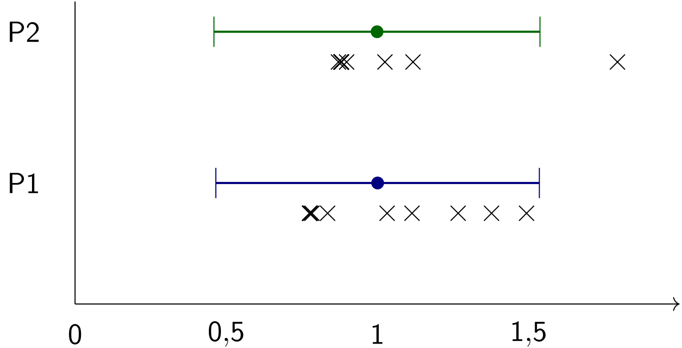
Standard Deviation, Assuming Normal Distribution



Deviation	**Chance of actual μ being in interval**
σ	68,27%
1,96σ	95,00%
2σ	95,45%
2,58σ	99,00%
3σ	99,73%

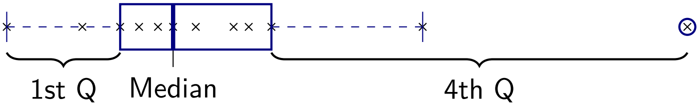
How Well Does Normal Distribution Fit?

Representation with error bars (95% confidence interval):



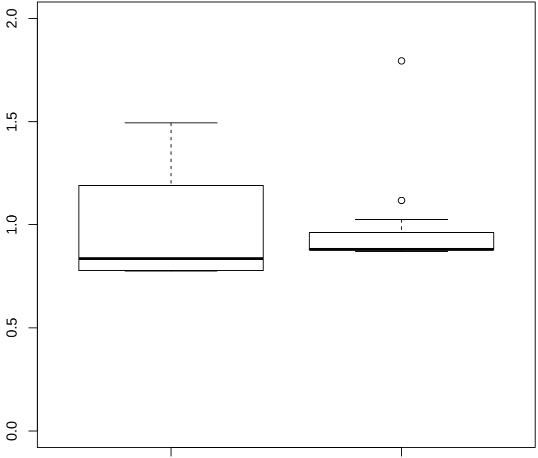
Mean + Std.Dev. are misleading if measurements don't observe normal distribution!

Box Plots

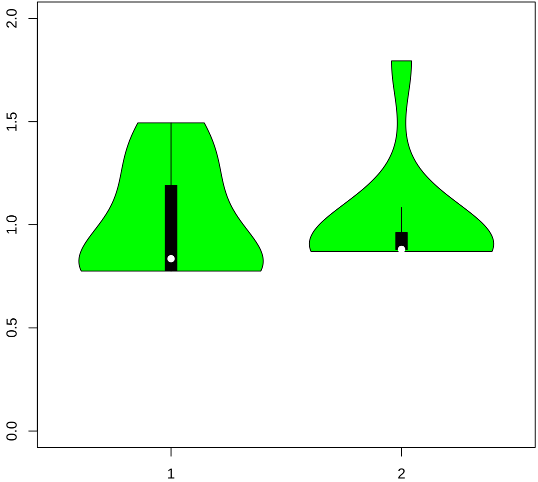


- Split data into 4 *Quartiles*:
  - Upper Quartile (1st Q): Largest 25% of measurements
  - Lower Quartile (4th Q): Smallest 25% of measurements
  - Median: measured value, middle of sorted list of measurements
- Box: Between 1st/4th quartile boundaries
  Box width = inter-quartile range (*IQR*)
- 1st Q whisker shows largest measured value $\leq 1{,}5 \times$ IQR (from box)
- 4th Q whister analogously
- Remaining *outliers* are marked

Box plot: example



Violin Plots



Summary

- We don't usually know our statistical distribution
- There exist statistical methods to work precisely with confidence intervals, given certain assumptions about the distribution (not covered here)
- Visualising without statistical analysis:
  - **Box Plot**
    - Splits data into **quartiles**
    - Highlights points of interest
    - No assumption about distribution
  - **Violin Plot**
    - Includes Box Plot data
    - Tries to approximate probability distribution function visually
    - Can help to identify actual distribution

Outlook

`http://cs.lth.se/EDAP15`