Experimental Evaluation of Programming Systems

Tomas Kalibera
Jan Vitek

Practices (not?) to follow

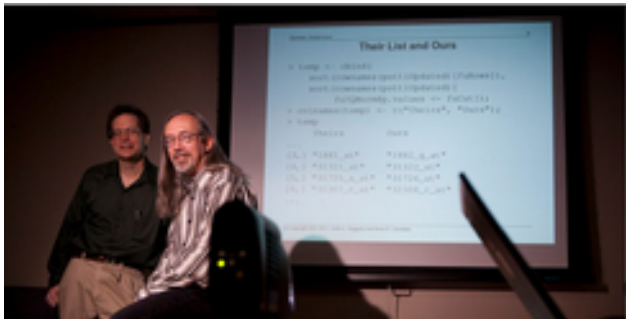
In 2006, Potti and Nevins at Duke reported they could predict lung cancer using expression arrays, and started a company

By 2010, 3 major papers were retracted, the company was gone, Potti resigned, and a major investigation was underway

By 2016, 10 papers were retracted and Office of Research Integrity (ORI) found that Potti engaged in research misconduct

Due to a combination of bad science ranging from fraud, unsound methods, to off-by-one errors in Excel spreadsheets

Uncovered by a repetition study conducted by statisticians Baggerly and Coombes with access to raw data and 2,000 hours of effort



More information on the case available at https://en.wikipedia.org/wiki/Anil_Potti

The GPGPU speedups story

A number of sources claimed 10x-1000x speedups of GP-GPU over CPU computation for several kernels (2008-2009)

The sources included 2 papers at ACM/IEEE Conference on Supercomputing, 1 conference paper and 1 journal paper

Lee et al found in 2010 (ISCA'10 paper) that when optimizing the software for both CPU and GPU, the speedup is about 2.5x.

The binary search trees story

Gary Knott (1975) in his thesis found empirically on random inserts/deletions that BSTs do not deteriorate with asymmetric deletions.

Eppinger (1983) studied larger trees and found this is not true.

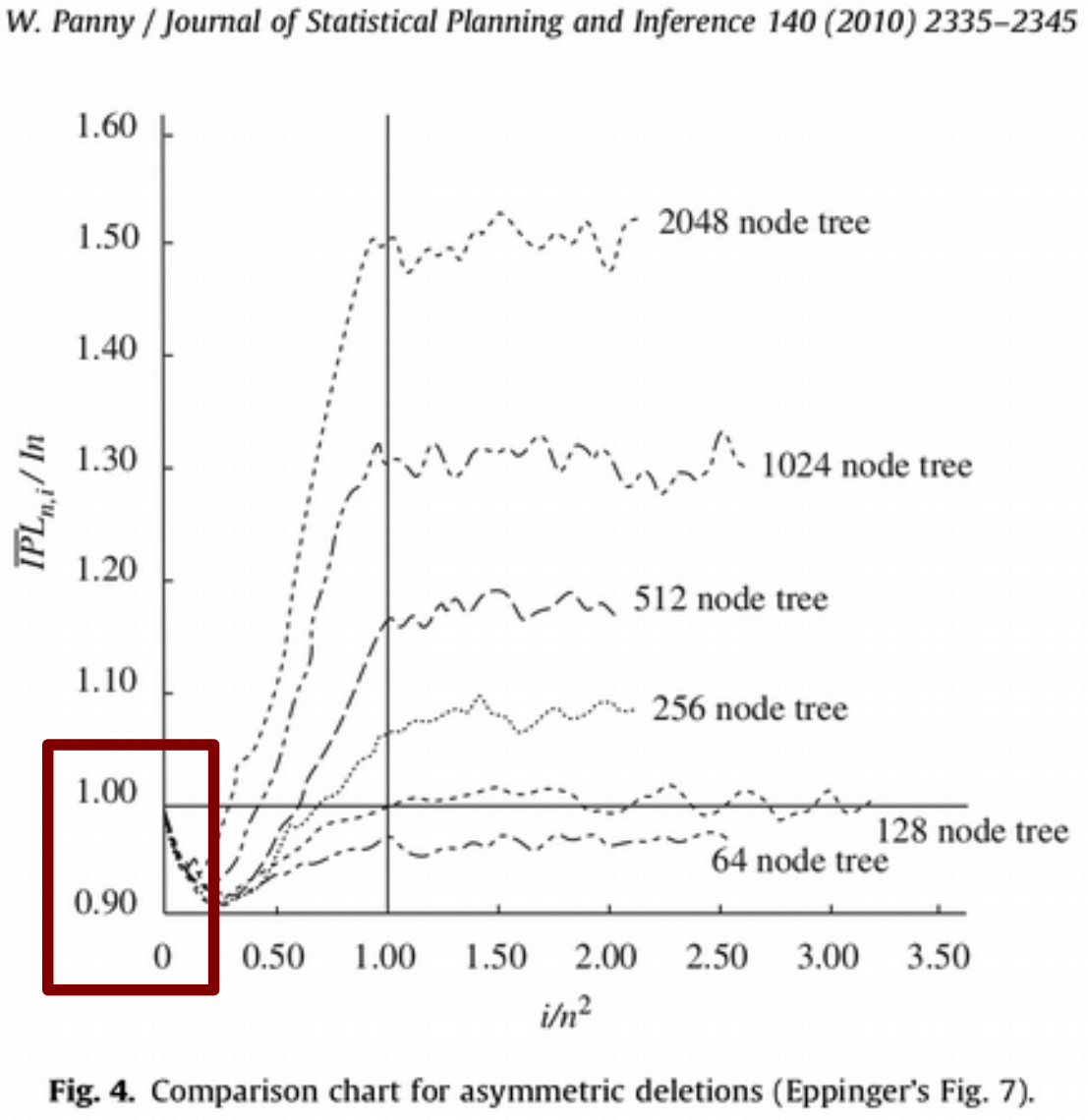


Fig. 4. Comparison chart for asymmetric deletions (Eppinger's Fig. 7).

Goals and quantities of experiments

Know the goals

- Ends-based
  - Performance characteristics of a single system
  - Comparison of two or more systems
- Explanatory
  - Find hints, evidence to explain observed behavior
- Scope
  - How general answer we’re looking for?

Usual goals in PL/Systems

- Ends-based
  - Improvement over the best performing system

    (in execution time, parallel speedup, power consumption, code size, pause time, reaction time)
  - Measured on application benchmarks, kernels
- Explanatory
  - Explain the improvements/overheads (cache misses, cache size, TLB, time spent in GC, memory utilization)
  - Sometimes using directed micro-benchmarks
- Scope
  - Pick one or two common platforms & OS
  - Common benchmark suites

Kinds of performance quantities

- Responsiveness
  - Time (response time, latency)
  - *Time between arrival of packet to the gateway and its successful delivery to destination*
- Productivity
  - Rate (throughput, speed, network bandwidth)
  - *Number of transactions processed per second by application server*
- Utilization
  - Percentage of time a particular resource is at least at given load level
  - *Percentage of time the CPU is not running the idle task*
- Stalls
  - Cache-misses, page-faults, pipeline

Prevailing metrics in PL/Systems are based on execution time

- Ratio of times – measure of optimizations
  - Improvement in execution time
  - Speed-up, parallel speed-up, performance overhead
- Absolute time
  - Time of a system to boot and start accepting input
  - Time of a garbage collection cycle
  - Pause time in concurrent garbage collector
  - Time to call a function

Execution time and statistics

Factors impacting execution time

- Fixed effects
  - Algorithm/code/optimization – what we work on
  - Input (benchmark programs)
  - CPU, OS, libraries, compiler optimizations, location in virtual memory
  - Report (reduce scope) or **randomize**
- Random effects
  - Location in physical memory, system load, scheduling, context switches, hw interrupts, randomized algorithms
  - **Model, summarize using statistics**

Run DaCapo fop benchmark (Java) repeatedly, record execution times

```
for I in `seq 1 100` ; do
    java -jar dacapo-9.12-bach.jar fop
done > fop.out 2>&1
```

Read the times into R, into vector “x” (in seconds)

```
out <- readLines("fop.out")
rlines <-  grep("==== DaCapo .* in [0-9]+ msec.*", out, val=T)
timesms <-  as.numeric(gsub(".* in ([0-9]+) msec.*","\\1",rlines))
x <- timesms / 1000
```

Show first 10 times

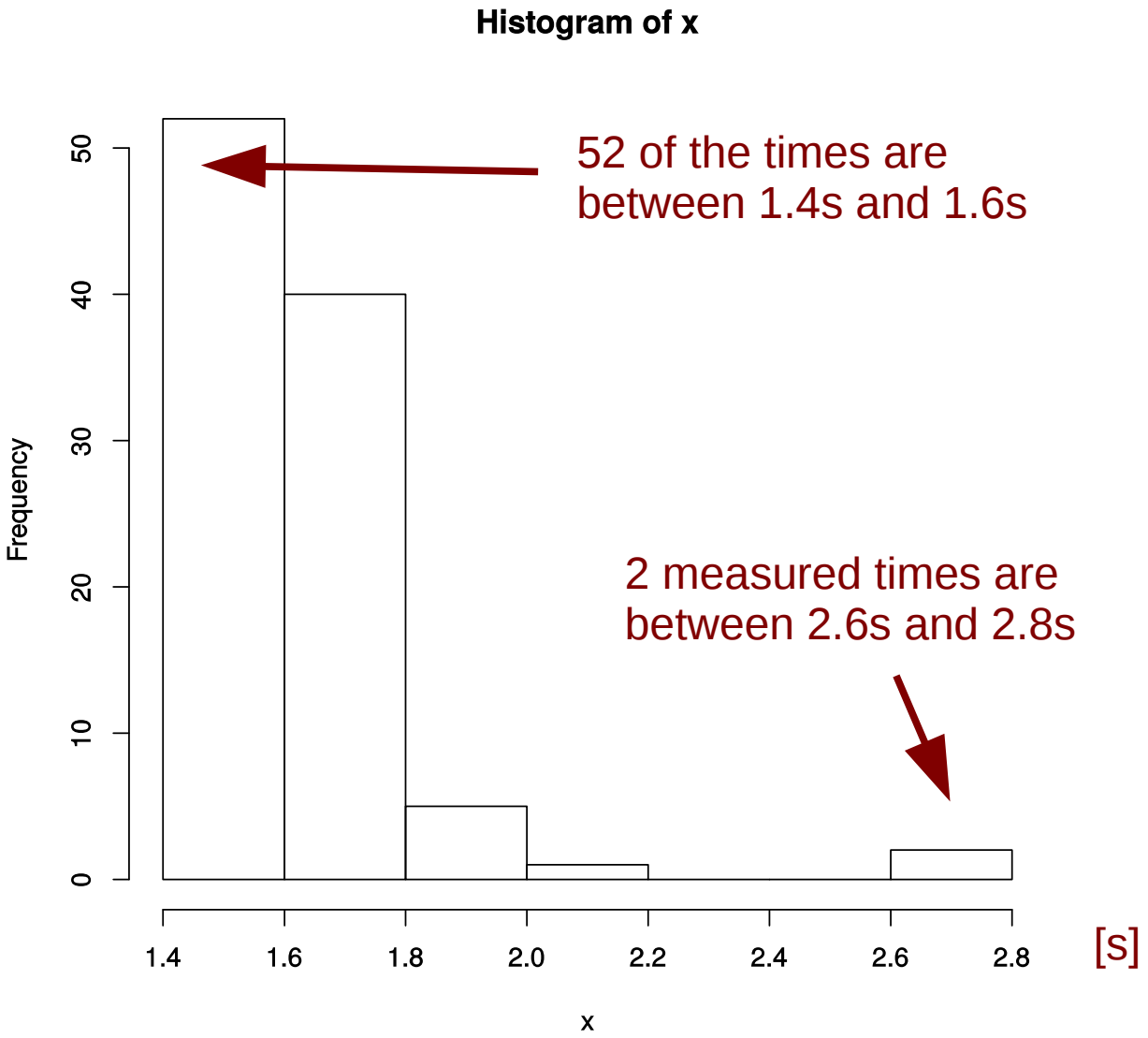
```
> x[1:10]
[1] 2.750 1.785 1.627 1.672 1.667 1.584 1.557 1.730 1.505 1.464
```

We **assume** times are repetitions of the same process, we have the same expectations about x[1] as about x[2].

We **assume** times are (statistically) independent – the fact that x[1] is 1.785 does not give us a clue what x[2] will be.

Show a histogram

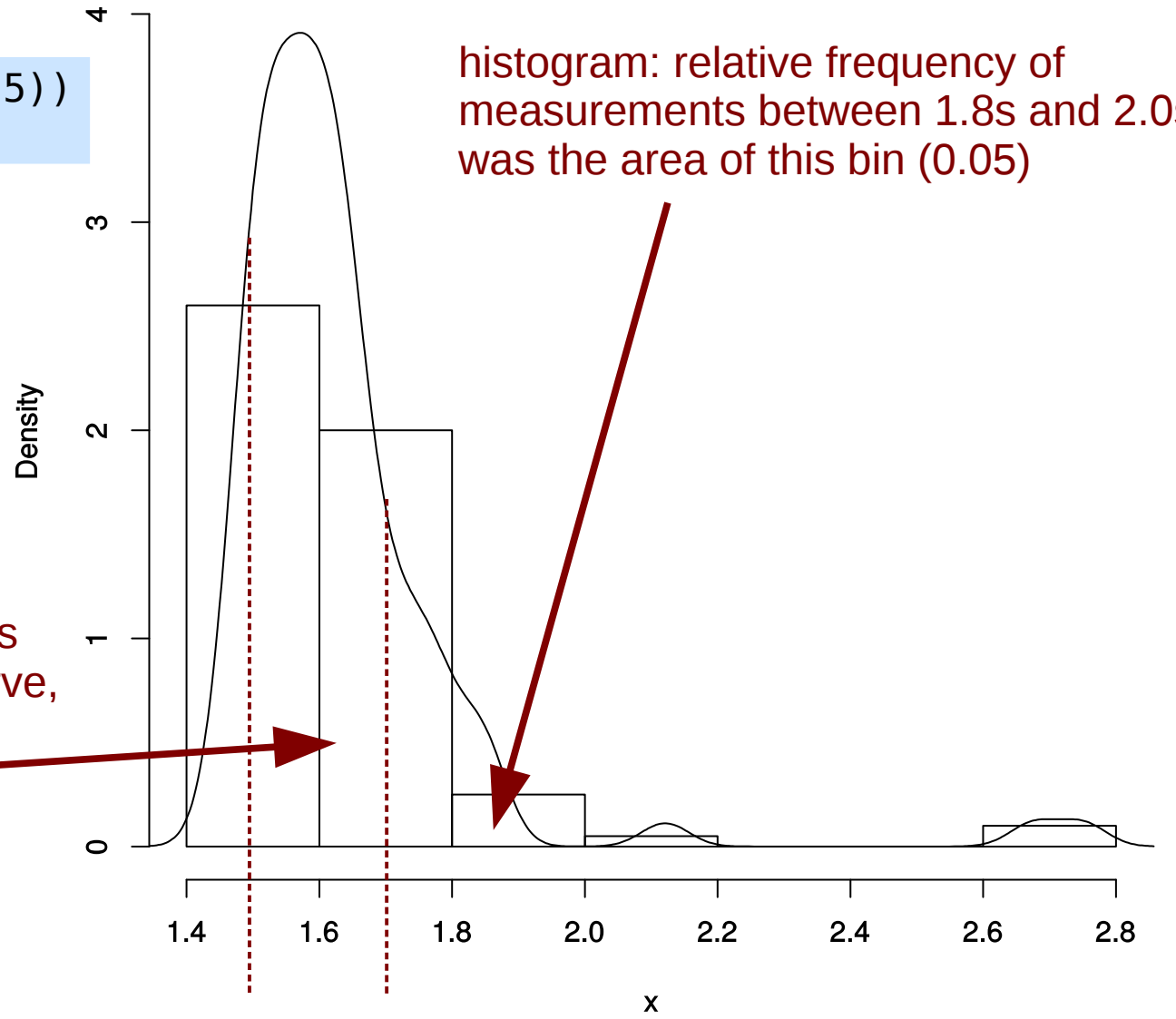
```
hist(x, freq=T)
```



We assume all times come from the same underlying process. With increasing number of iterations, the shape of the histogram should stabilize.

Histogram with **estimate** of probability density function

```
hist(x, prob=T, ylim=c(0,5))
lines(density(x))
```



histogram: relative frequency of measurements between 1.8s and 2.0s was the area of this bin (0.05)

density: probability of measured time to be between 1.5s and 1.7s is the area below the density curve, between 1.5 and 1.7.

Under our assumptions, the **true** density function would fully describe the execution times of the benchmark.

Probability density

For continuous random variable **X** (e.g. time of “fop”) and its **density function** **f**, the probability that a value of **X** will be between **a** and **b** is the area below the density curve, between **a** and **b**

$$P[a \le X \le b] = \int_a^b f(t)\,dt$$

$$f(t) \ge 0 \qquad \int_{-\infty}^{+\infty} f(t)\,dt = 1 \qquad P[X \le b] = \int_{-\infty}^{b} f(t)\,dt$$

The **distribution function** **F**, for a given value **x**, gives the probability that the random variable will take on a value up to **x**

$$F(x) = P[X \le x] = \int_{-\infty}^{x} f(t)\,dt$$

$$0 \le F(x) \le 1 \qquad P[a < X \le b] = F(b) - F(a) = \int_a^b f(t)\,dt$$

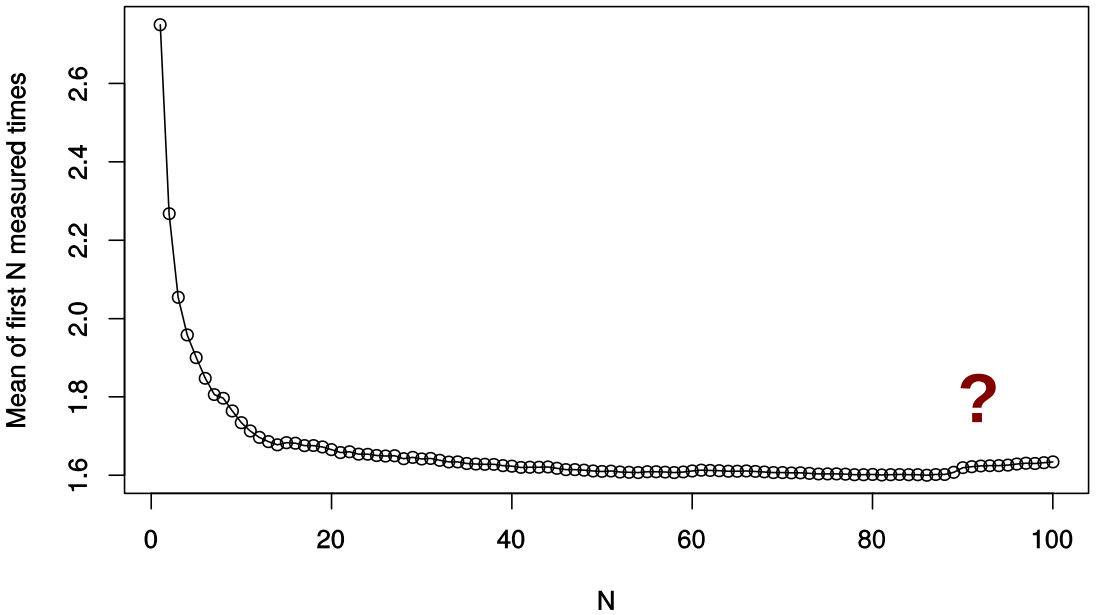
Estimating the expectation

We could report just a histogram as a result of a benchmark, which is an estimate of the density function. But it is often too detailed. We hence also estimate properties of the distribution.

Expectation E(X) is a measure of central tendency. It represents the "center of gravity" of a random variable and the sample arithmetic mean converges to it.

$$E(X) = \int_{-\infty}^{\infty} t f(t)\, dt \qquad \overline{X_n} = \frac{1}{n}\sum_{i=1}^{n} X_i \qquad \overline{X_n} \rightarrow E(X)$$

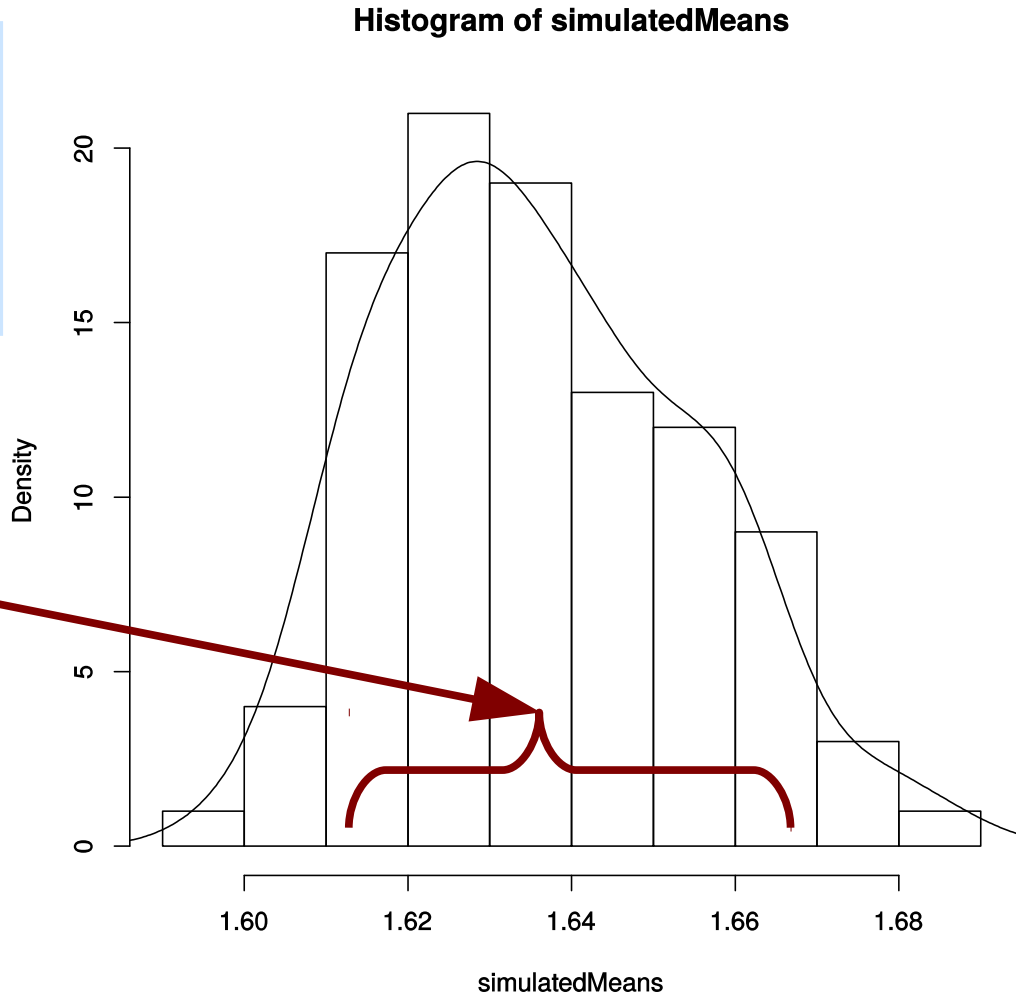
```
> mean(x)
[1] 1.63418
> plot(
    sapply(
      1:length(x),
      function(i) mean(x[1:i])
    ),
    type="o",
    ylab="Mean of first N",
    xlab="N"
  )
```



How good estimate do we have?

Given our assumptions that the order of measurements is irrelevant, lets simulate more experiments – re-sample the measured values 100 times (with replacement), take the mean, and plot a histogram of these means (**bootstrap histogram**)

```
simulatedMeans <- sapply(1:100,
  function(i)
    mean(sample(x, replace=T))
)
hist(simulatedMeans, prob=T)
lines(density(simulatedMeans))
```



By the 6 bins from 1.61 to 1.67, we cover 91% of the are of the area of the histogram.

Intuitively, with 91% confidence, the true expectation should be between 1.61s and 1.67s.

Confidence interval

Lets find bounds (interval) that covers the true mean with a fixed confidence, say 95%.

That is, find **a** and **b** such that $P[a<Y\leq b]=F(b)-F(a)=0.95$

One way is to skip 2.5% of the values at the left and 2.5% of the values at the right. For that we need **a** and **b** so that

$$F(b)=0.975 \qquad F(a)=0.025$$

The quantile function helps, it is the inverse of distribution function **F**. The sample quantile funciton would take the 25th and 975th value out of 1000 sorted values.

```
> quantile(simulatedMeans, probs=c(0.975, 0.025))
   97.5%     2.5%
1.675548 1.607960
```

With 95% confidence, the true mean lies between 1.61s and 1.68s.
This is is percentile boostrap confidence interval.

Bootstrap confidence interval

R can compute the bootstrap confidence intervals for us

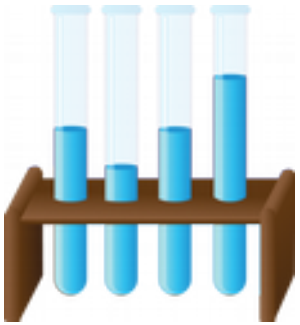
Lets do more simulations

```
> library(boot)
> b <- boot(x, function(d, sel) mean(d[sel]), R=10000)
> boot.ci(b, conf=0.95, type="perc")
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 10000 bootstrap replicates

CALL :
boot.ci(boot.out = b, conf = 0.95, type = "perc")

Intervals :
Level     Percentile
95%   ( 1.601,  1.674 )
Calculations and Intervals on Original Scale
```

With 95% confidence, the true mean lies between 1.601s and 1.674s. We may be unlucky sometimes, but if we evaluate experiments like this, in 95% we should have covered the true mean (IF THE ASSUMPTIONS WE MADE HOLD).



Exercise

R tool+language for data analysis, visualization, statistical computing

http://www.r-project.org

- **Linux**
  Ubuntu, Debian: r-base
  openSuse: R-base
  Redhat/Fedora: R
- **OS/X**
  binaries at https://cloud.r-project.org/bin/macosx/
  install Xquartz
- **Windows**
  binaries at https://cloud.r-project.org/bin/windows/base/

Spectralnorm: mean + conf. interval

Run spectralnorm benchmark repeatedly (R), record execution times
(you can skip this and read the results from snorm.out)

```
for N in `seq 1 100` ; do
    time R --slave < spectralnorm.r
done >snorm.out 2>&1
```

1. Read the file into R (use “real” time)
2. Plot a histogram
3. Calculate the sample mean
4. Calculate a percentile bootstrap confidence interval for the mean
5. Calculate the sample median
6. Calculate the bootstrap confidence interval for the median

Checking independence in data

Spotting dependence in data

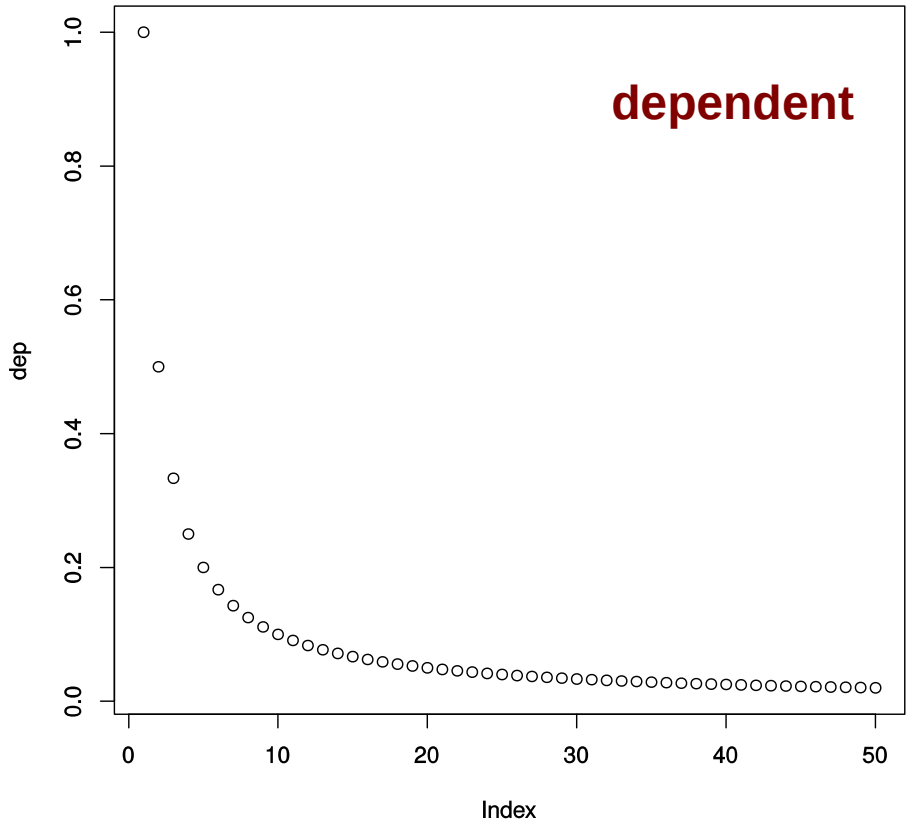
Generate 50 dependent data points into “dep” and 50 independent into “ind”

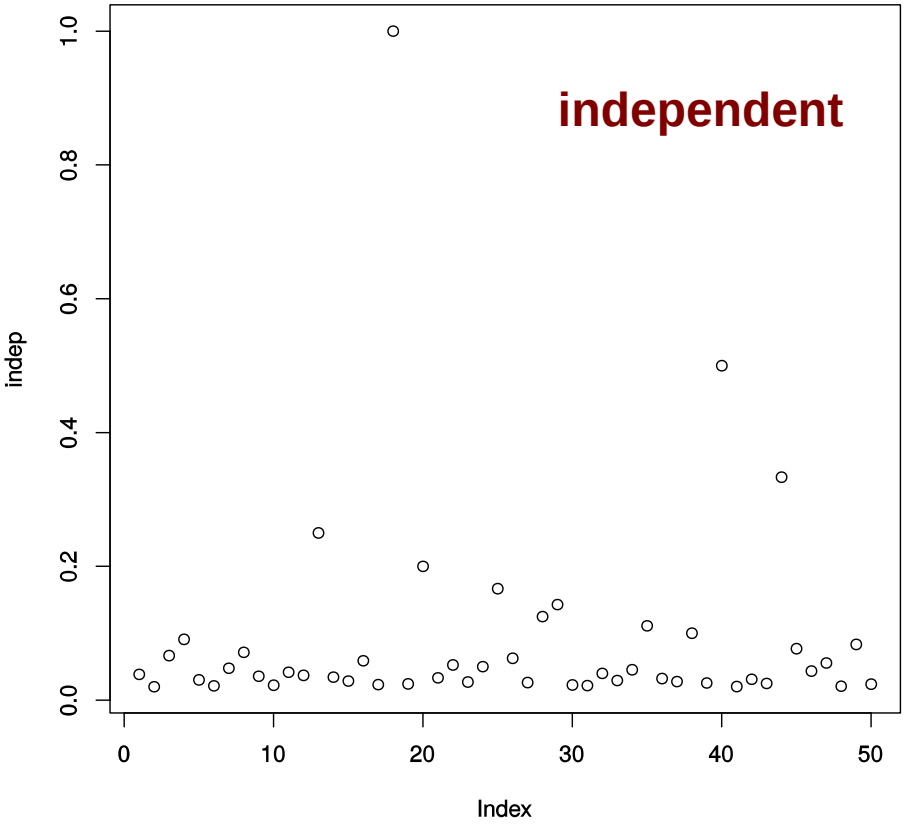
```
dep <- sapply(1:50,function(x) 1/x)
indep <- sample(dep, replace=F)
```

← Randomly reordered

Run-sequence plot

```
plot(dep)
```

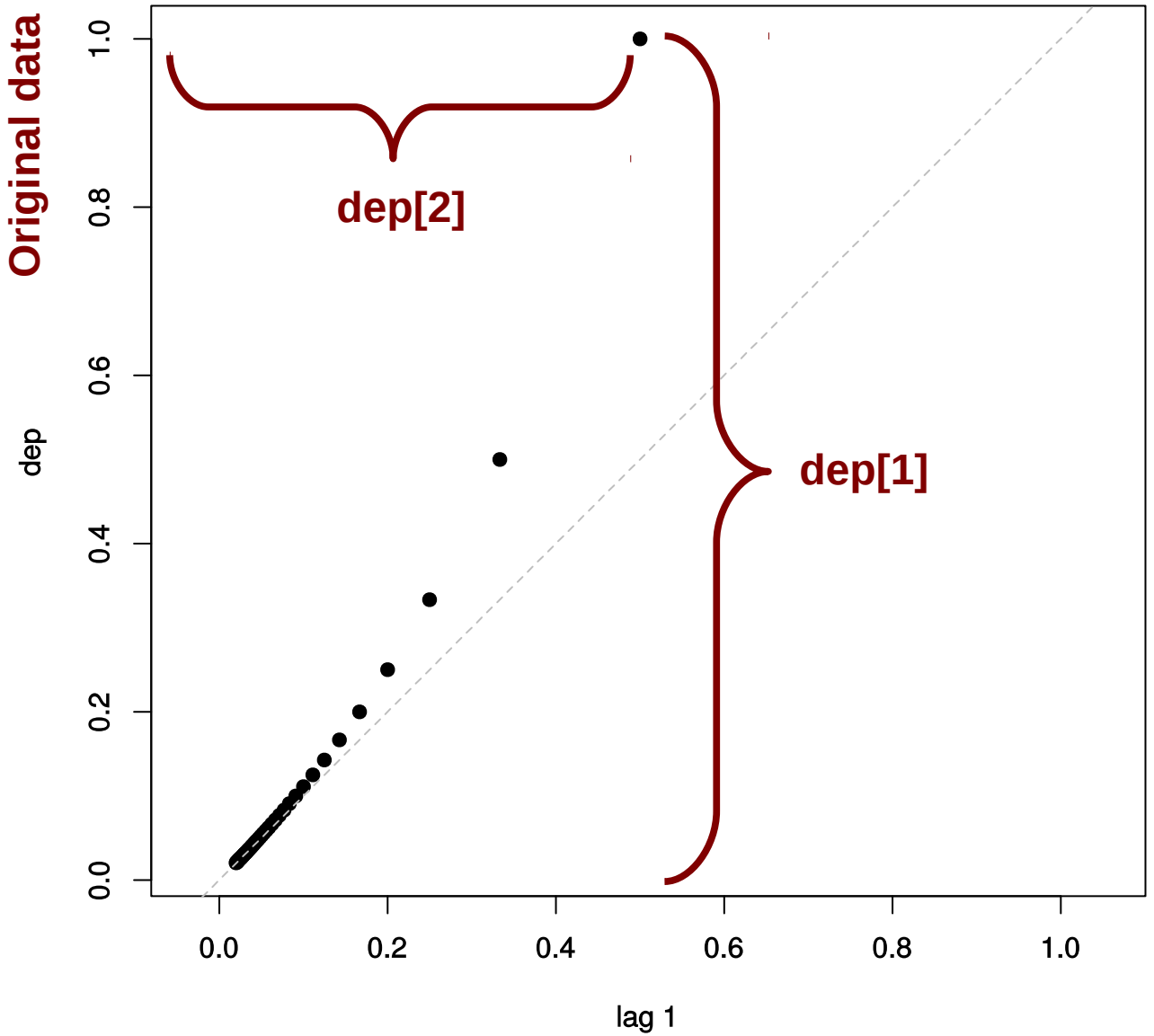




Spotting dependence: Lag plot

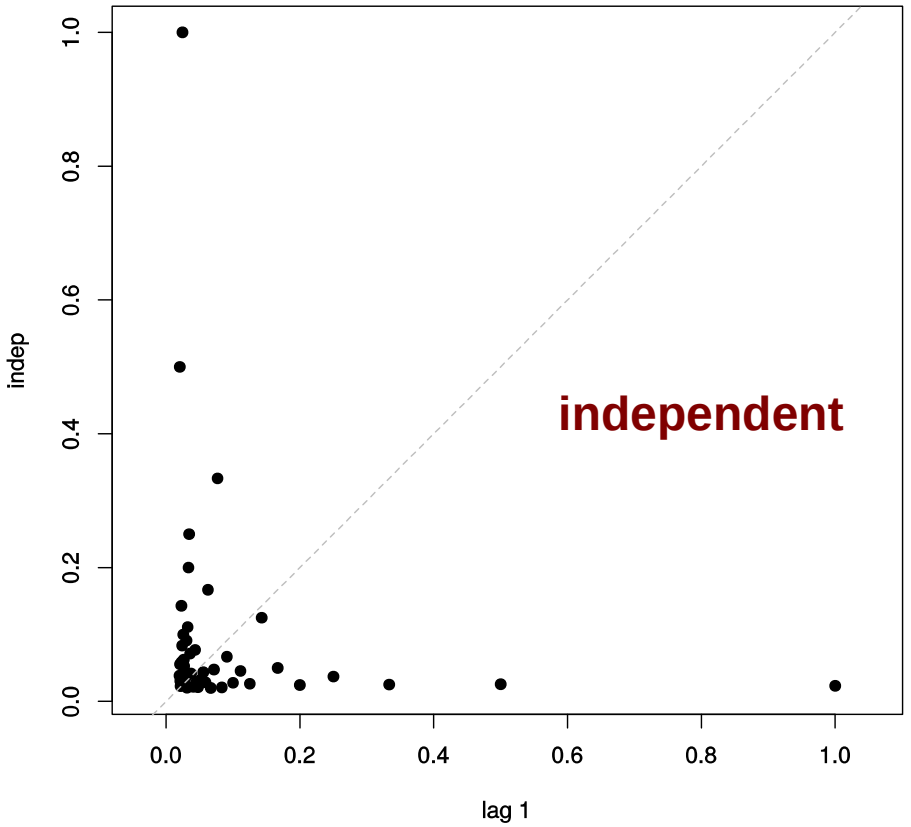
Lag plot

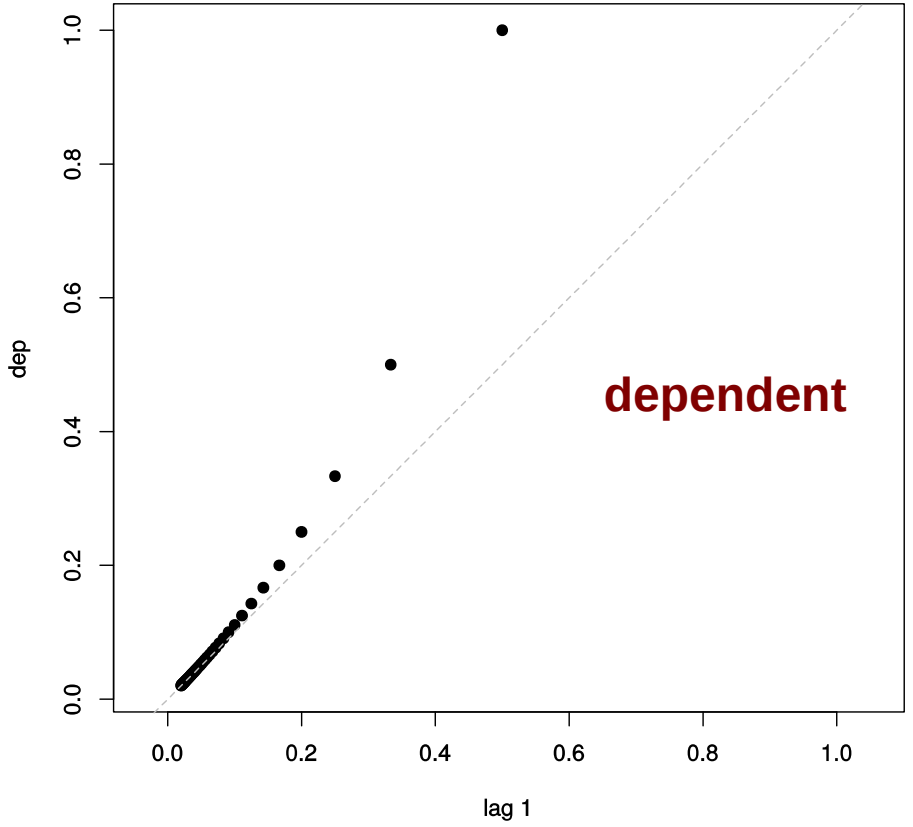
```
lag.plot(dep,
  labels=F,
  do.lines=F,
  pch=19
)
```



Spotting dependence: Lag plot

Compare for original and randomly reordered data



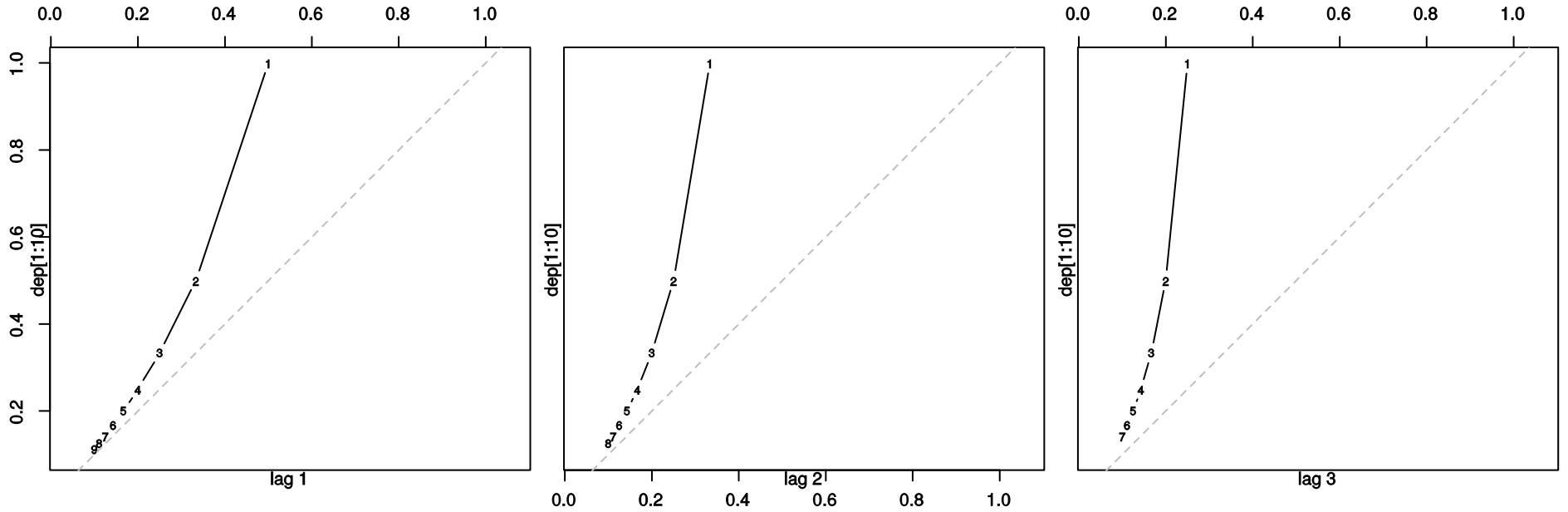


Independent data should be mostly symmetrical over the dotted line (y=x). There should be no observable patterns. Independent data have lag-plot similar to randomly reordered data.

Spotting dependence: Lag plot

Lag plots with different lags

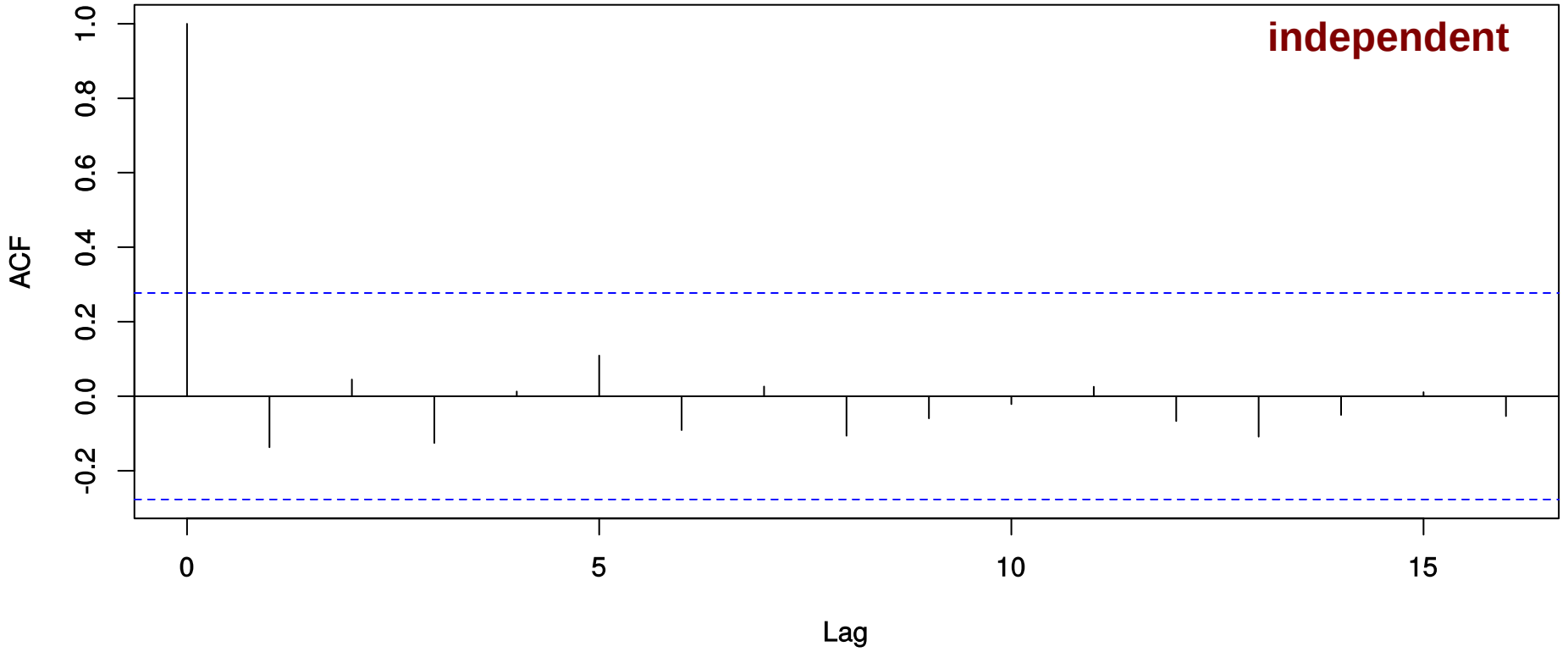
```
lag.plot(dep[1:10], lags=3)
```



Dependencies can be easier to spot with fewer data points, indexed and connected by lines.

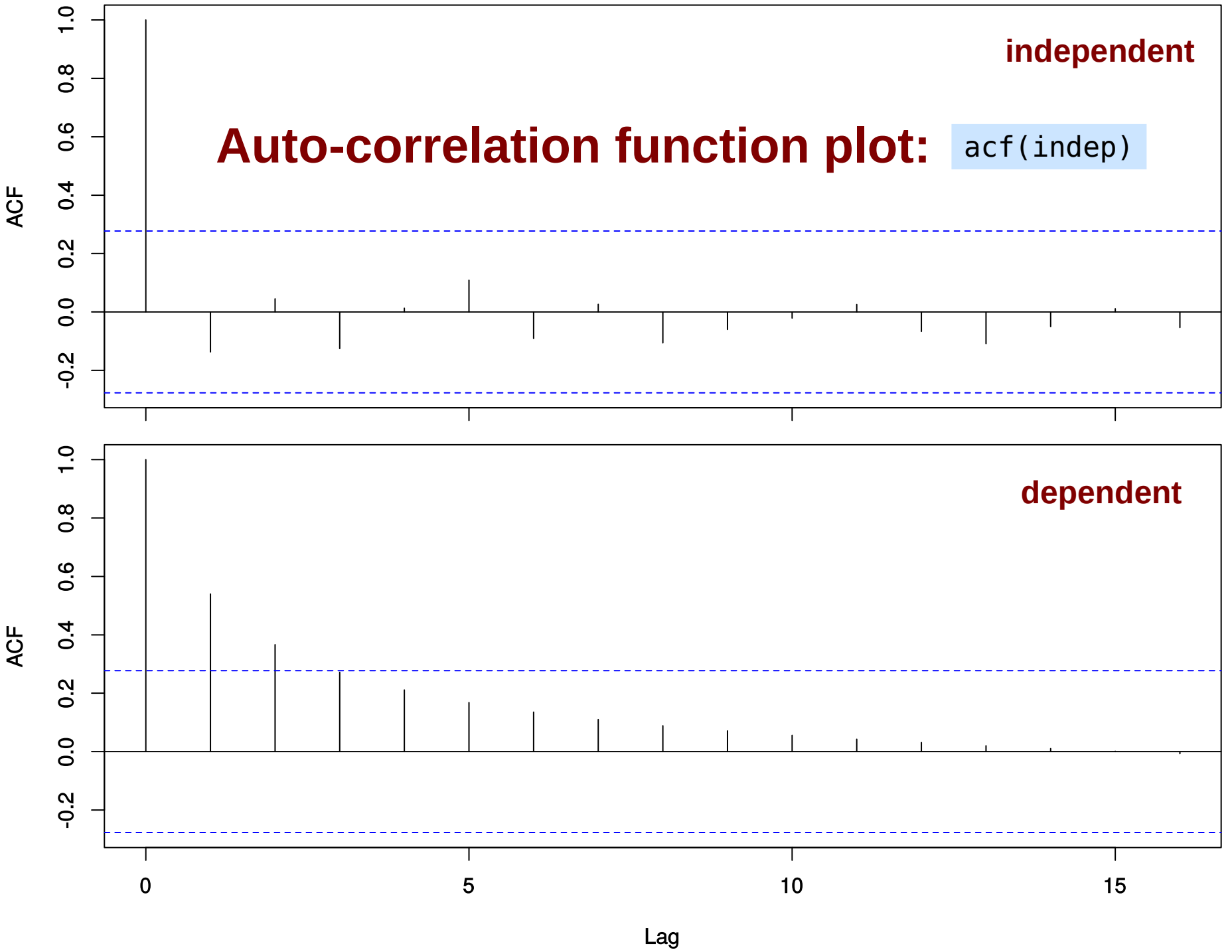
Spotting dependence: ACF plot

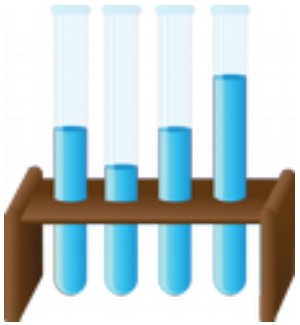
Auto-correlation function plot: `acf(indep)`



Correlation is a measure of linear dependency.

Blue lines mark autocorrelation of white noise.


Auto-correlation function plot:
acf(indep)
independent
dependent
ACF
ACF
1.0
0.8
0.6
0.4
0.2
0.0
-0.2
0
5
10
15
Lag



Exercise

FFT: checking independence

Do this for two platforms, Pentium 4 and Itanium (fft_p4.out and ia64.out). Read data into R and convert timer ticks to time
(P4 ran at 2194.055Mhz and Itanium at 800.179008Mhz).

```
p4 <- read.table("fft_p4.out", header=T)
p4$time <- p4$tsc/(2194.055*1e6)
```

1. Generate run-sequence plots at different scales (y-scale, different subsets of samples), for original and re-sampled data, looking for statistical dependence
2. The same using lag-plots, acf plot
3. If data is dependent, try to learn as much as possible about the dependencies
4. If you find the data to be independent, calculate bootstrap confidence interval for the mean

Experiment design - repetitions

FFT: re-running the benchmark

Now we have 100 runs of the benchmark (before we only looked at 1 run),
from each run we have 2048 measurements.

List of 100 vectors (100 runs of the benchmark)

File names are like fft_ia64/run1.out

```
runs <- lapply(1:100, function(n) {
  d <- read.table(
    paste("fft_ia64/run", n, ".out", sep=""),
    header=T
  )
  d[[1]]
})
ia64 <- do.call(c, runs)
ia64 <- ia64/(800.179008*1e6)
```

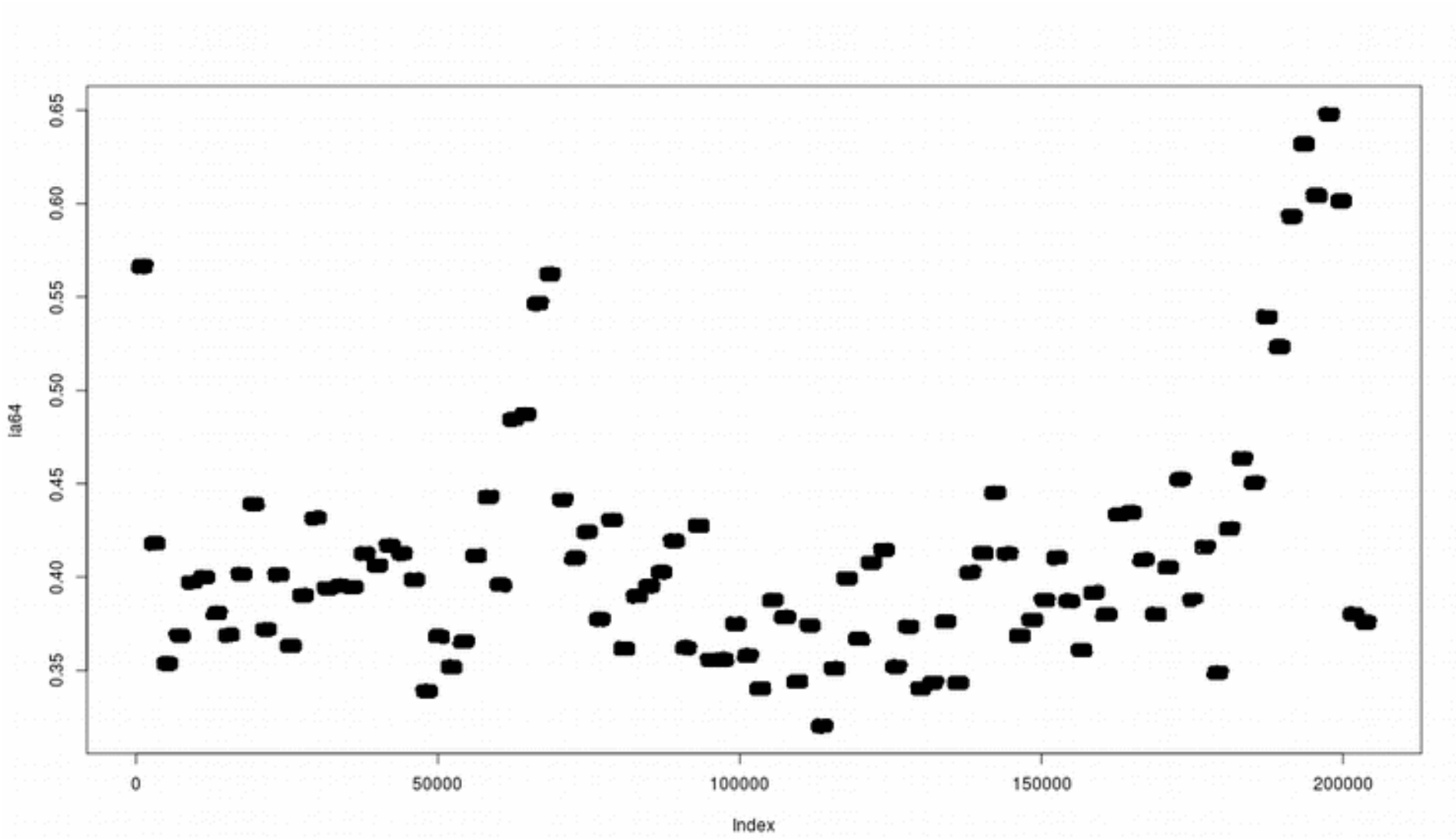
Vector of 2048 measurements
(first column of result data frame)

Vector of 100 * 2048 measurements

FFT: re-running the benchmark

Lets explore the sequence of measurements from different runs.

```
plot(ia64)
```

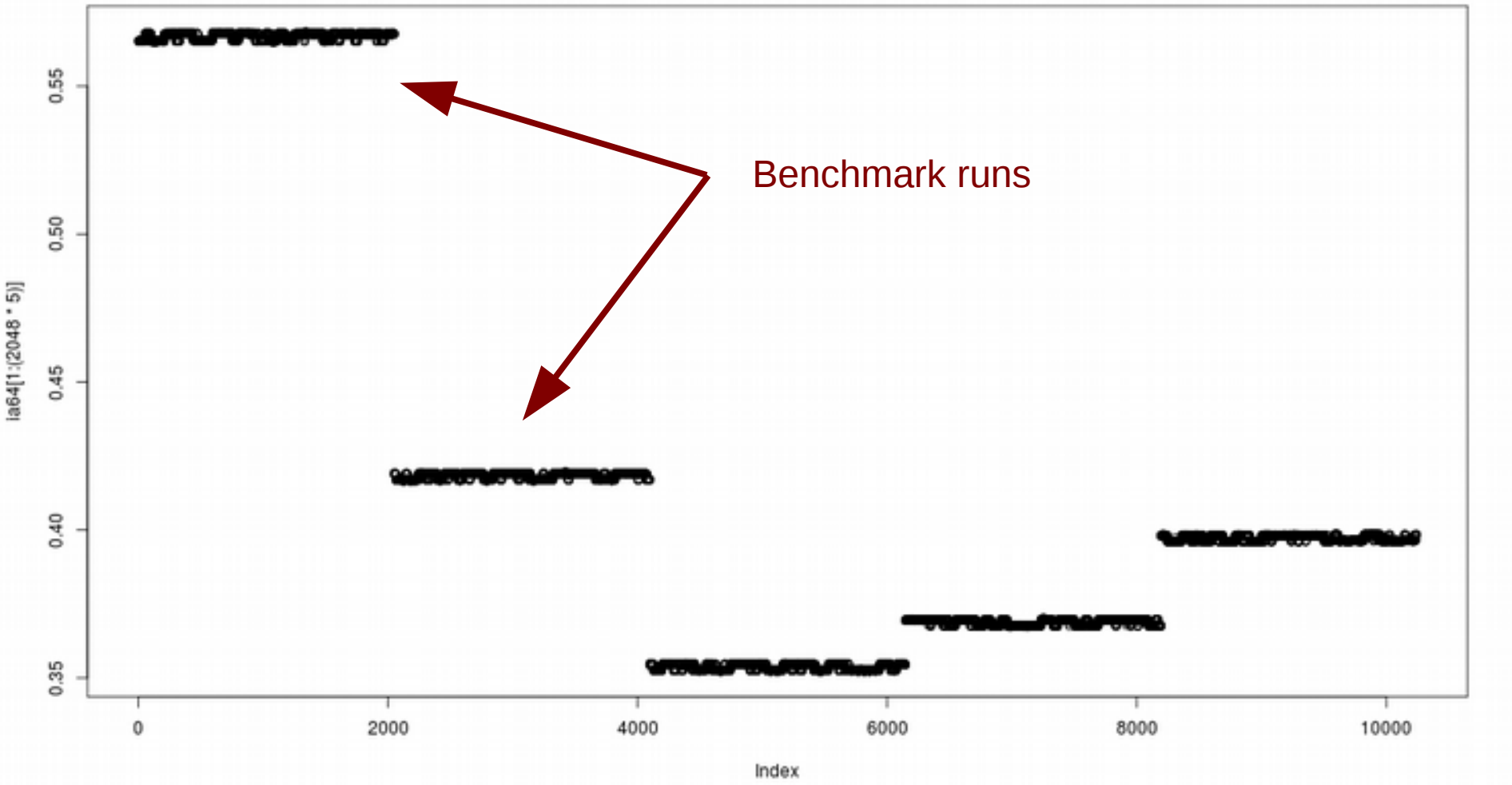


FFT: re-running the benchmark

Lets explore the sequence of measurements from different runs.

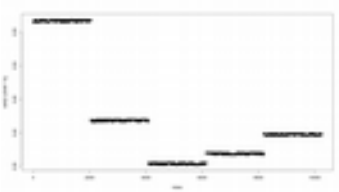
```
plot(ia64[1:(2048*5)])
```

(a simple variant of DEX scatter plot)



Non-determinism in execution (that does not appear in iterations)

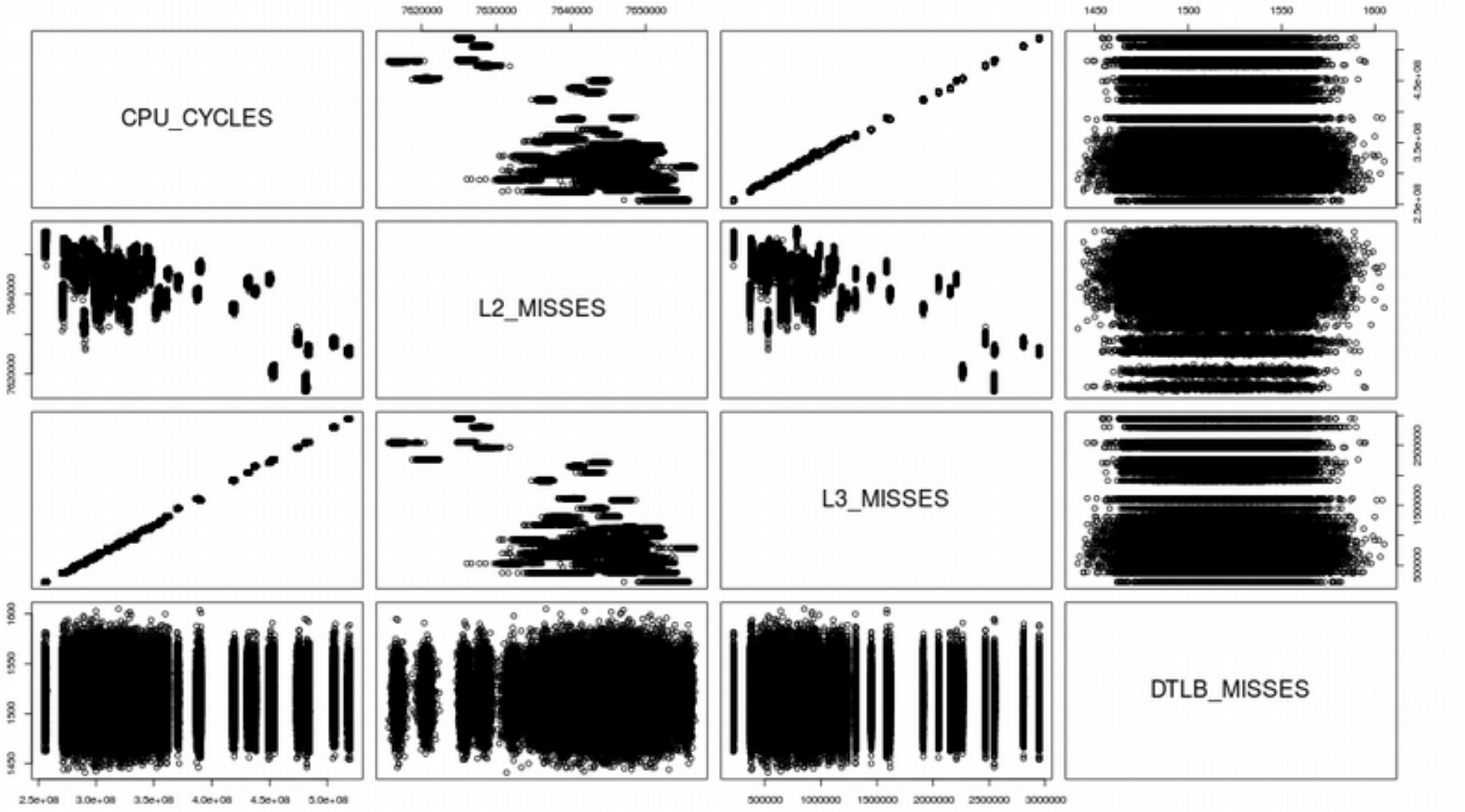
- Different executions of a benchmark have different performance
  - Plus with FFT, the difference is much bigger than between iterations in the same run
- Uncontrolled fixed effect
- Must re-run executions to avoid **bias**
  - And given the big impact of “execution”, no need to repeat iterations within execution



What is the cause of this non-determinism?

```
runs <- lapply(1:100, function(n)
  read.table(paste("fft_ia64/run", n, ".out", sep=""), header=T)
)
ia64 <- do.call(rbind, runs)
plot(ia64)
```

Joining data frames from multiple runs

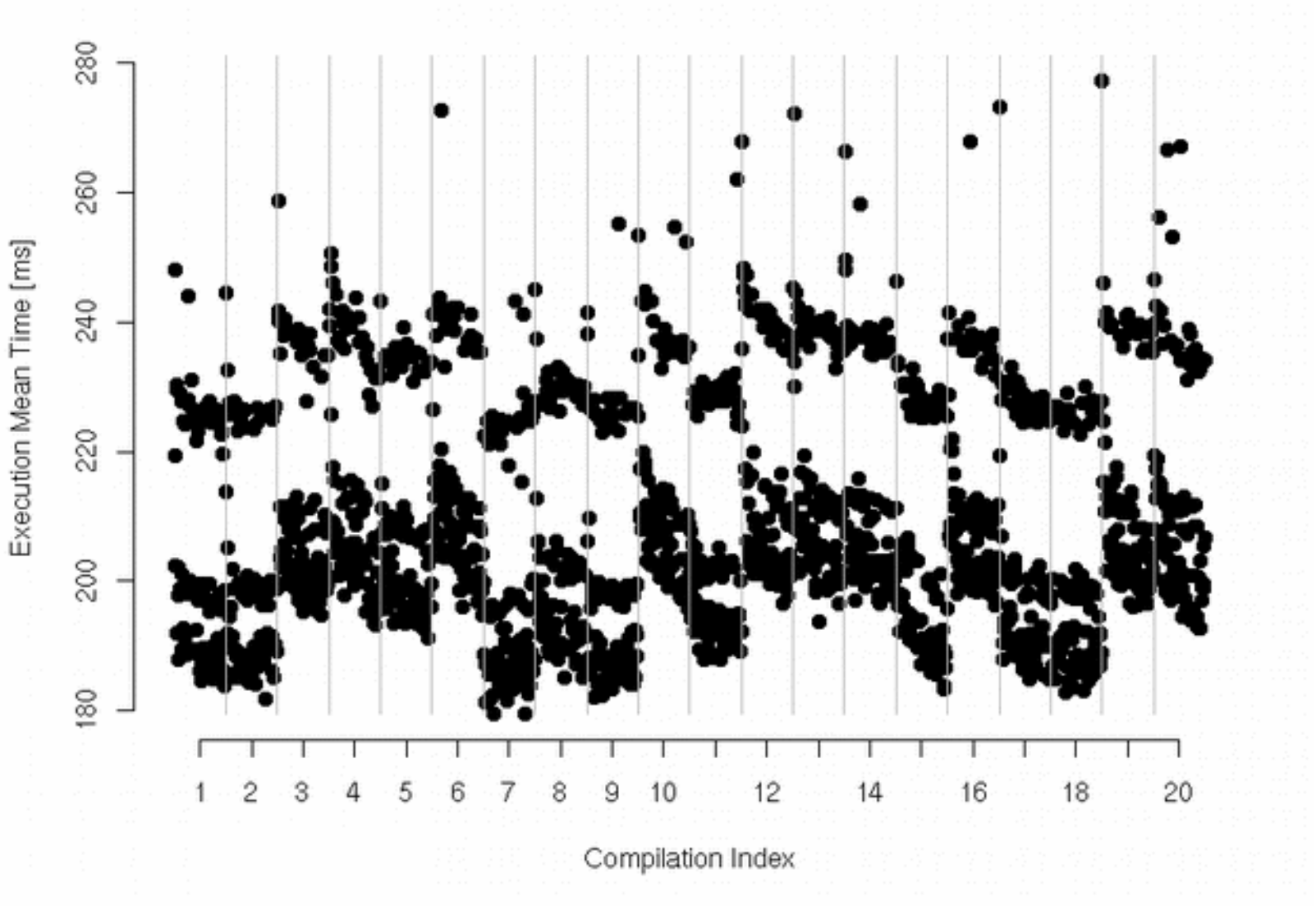


Non-determinism in compilation (that does not appear in executions)

- On some systems, linking order impacts performance (e.g. SPECCPU, training size)
  - Controlled fixed effects
  - Should be randomized – and then need to repeat compilation
- On some systems, build is non-deterministic
  - e.g. C++ compiler implementing anonymous namespaces
- Need to repeat compilation...

NOTE: naming of identifiers has also been reported to impact performance; code layout by function/data order does too..

Non-determinism in compilation in Mono

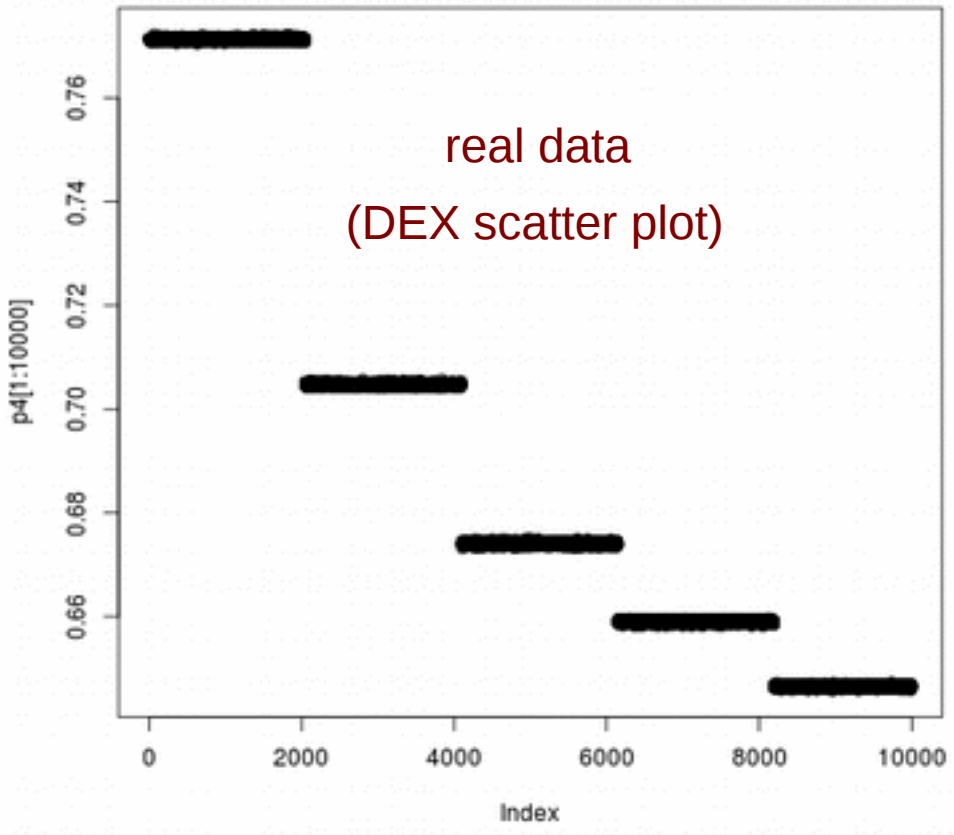


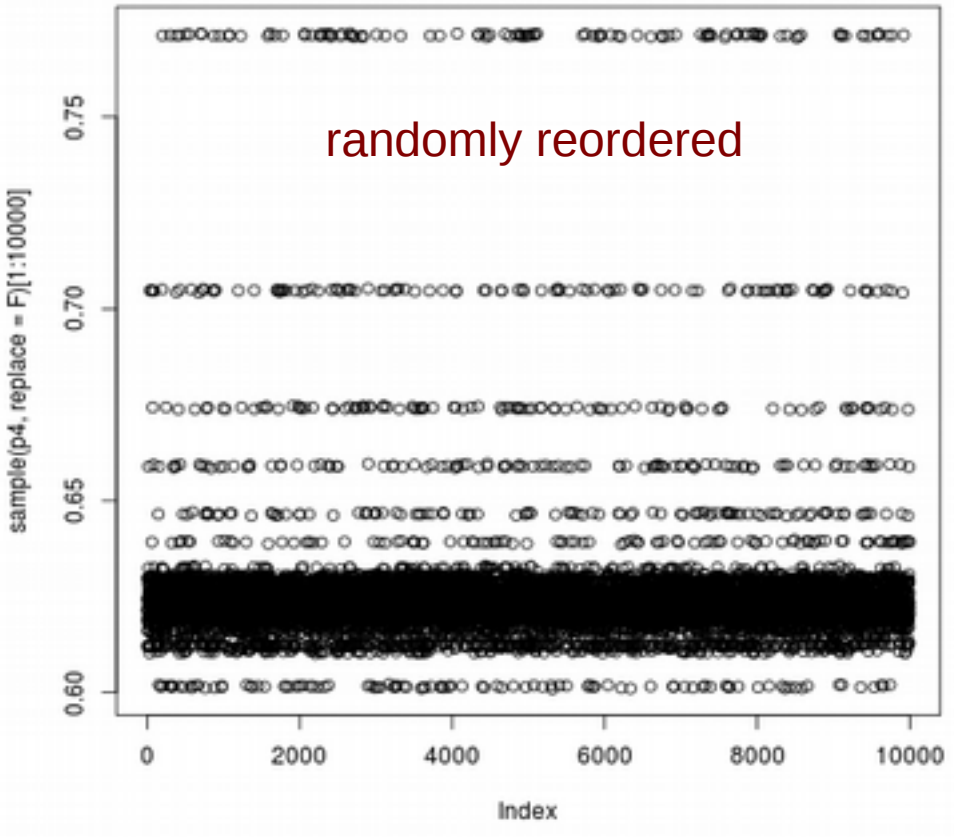
Highest level for repetition

- Find the highest “level” of non-determinism in the given system/benchmark
  - Identify important uncontrolled fixed effects and randomize them (like linking order, code layout)
  - Check if building is deterministic, binaries have different performance
- If it is cheap enough, repeat at a higher level anyway
  - If execution is cheap, repeat whole executions always
- If in doubt, repeating at a higher level is never wrong

Spotting non-determinism at particular experiment level

```
runs4 <- lapply(1:100, function(n) {
  d <- read.table(paste("fft_p4/run", n, ".out", sep=""), header=T)
  d[[1]]
})
p4 <- do.call(c, runs4)
p4 <- p4/(2194.055*1e6)
plot(p4[1:10000])
plot(sample(p4, replace=F)[1:10000])
```





Estimating the variance

If the graphical comparison is inconclusive, we can also compare variances (variance estimates) for iterations in the same execution and iterations in different executions.

Variance Var(X) is a measure of dispersion/spread of random variable, “mean square distance from the expectation”

$$Var(X) = E((X - EX)^2) = \int_{-\infty}^{\infty} t^2 f(t)\,dt - \left(\int_{-\infty}^{\infty} t f(t)\,dt\right)^2$$

$$S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X_n})^2 \rightarrow Var(X)$$

Standard deviation sd(X) is square root of the variance (so in units of variable **X**). **Coefficient of variation** (**COV**) is standard deviation divided by the mean (aka variation, relative variation, sometimes given as percentage)

```
var(x)
sd(x)
sd(x)/mean(x)
```

Identifying non-determinism through comparison of variations

```
> iters <- read.table("fft_p4/run1.out",header=T)$tsc
> sd(iters)/mean(iters)
[1] 0.0003345541
```

Variation in 1st run

```
> runs <- sapply(1:100, function(n) {
    read.table(paste("fft_p4/run", n, ".out", sep=""), header=T)$tsc[1]
  })
> sd(runs)/mean(runs)
[1] 0.03017004
```

Variation of 1st iterations from different runs

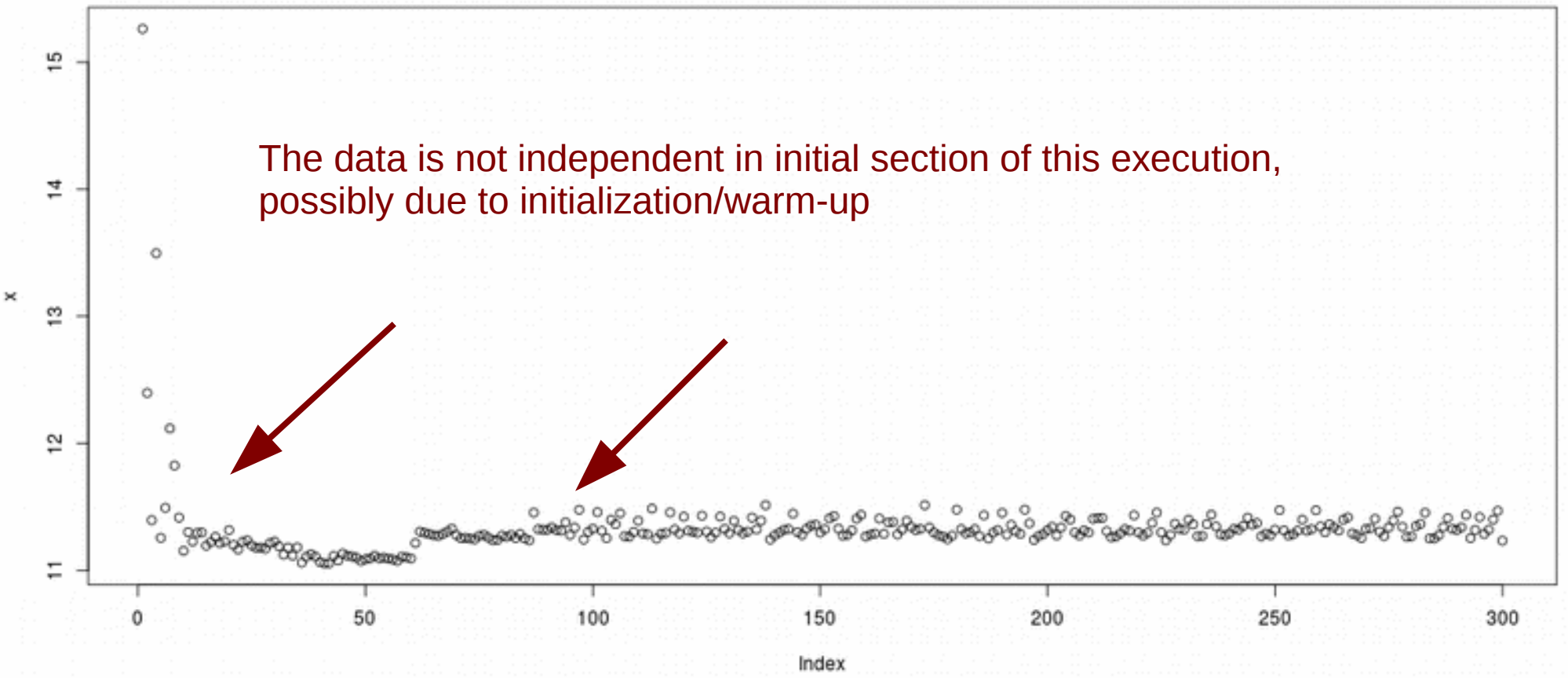
In many cases the difference in variations is not this obvious.

A sophisticated method for comparing variations is ANOVA (analysis of variance), but it has its own assumptions, and only focuses on variance.

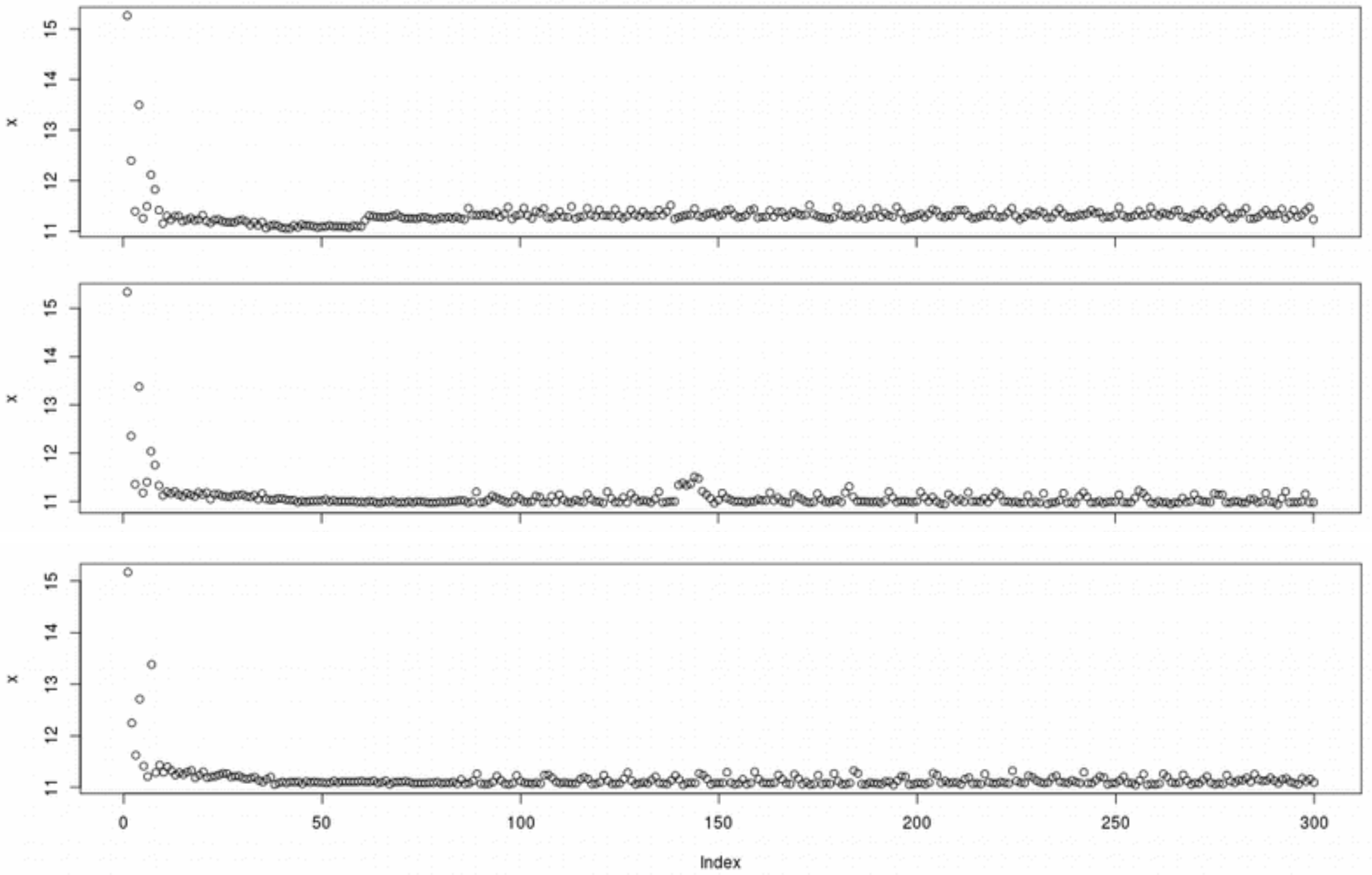
Experiment design - warmup

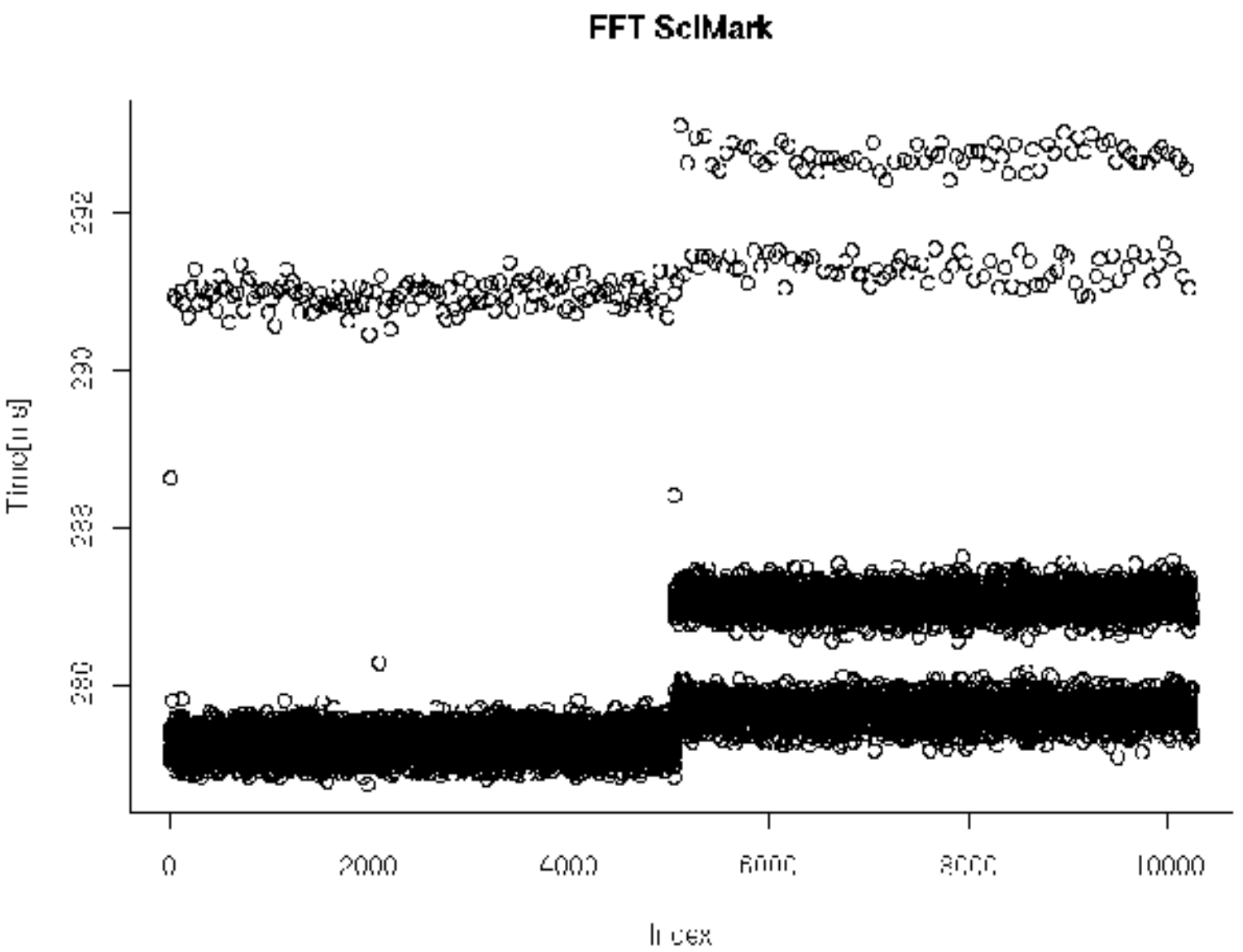
Read chart6 times into R, into vector “x” (in seconds)

```
readDacapo <- function(fn) {
  out <- readLines(fn)
  rlines <-  grep("==== DaCapo .* in [0-9]+ msec.*", out, val=T)
  timesms <-  as.numeric(gsub(".* in ([0-9]+) msec.*","\\1",rlines))
  timesms / 1000
}
x <- readDacapo("dacapo/chart6/chart6_1_1.out")
plot(x)
```



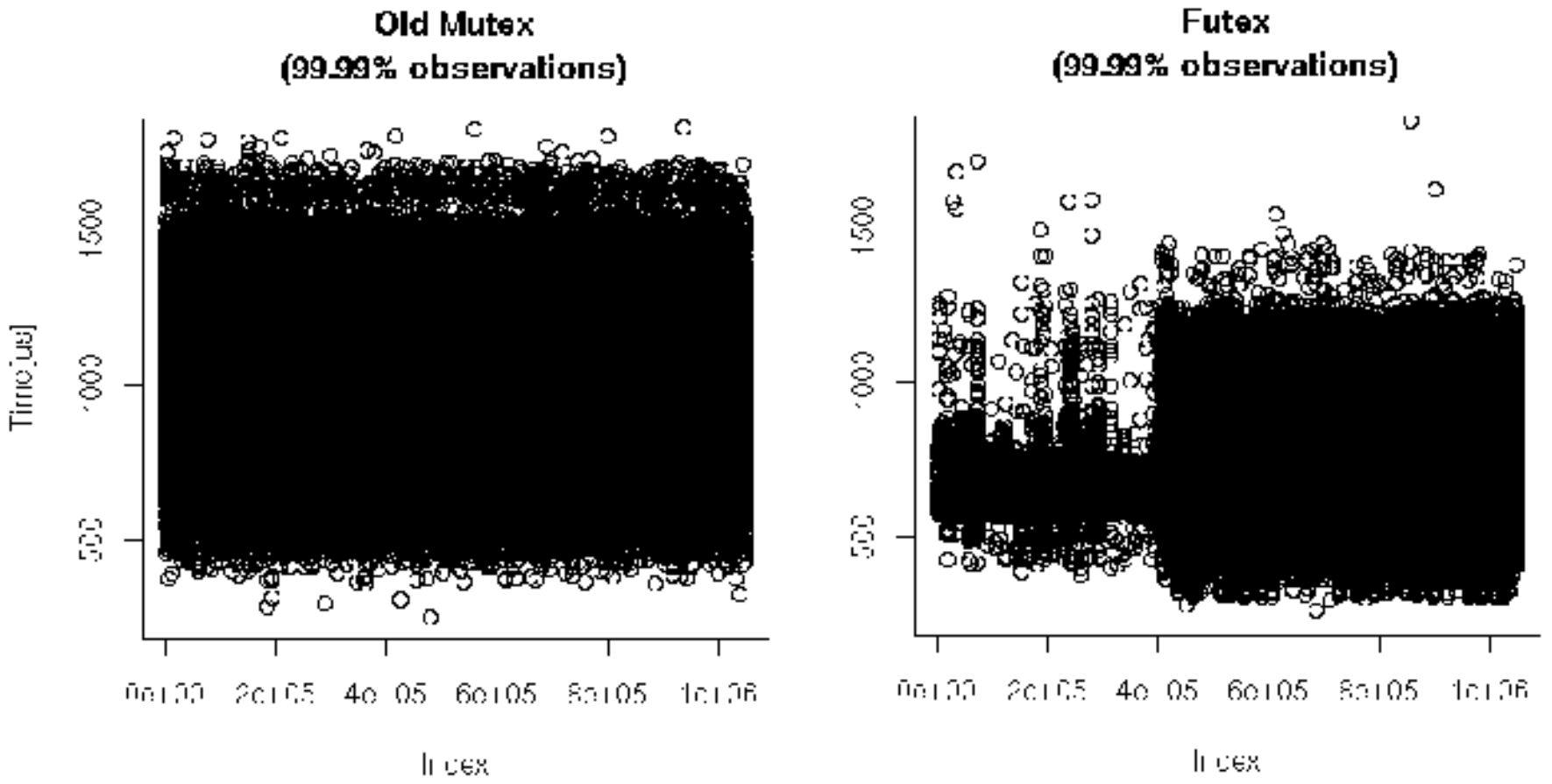
3 different runs of chart6 – the initialization seems repeatable.
It is not just a random fluctuation during an execution.





FFT-SciMark benchmark, Linux/Mono, C#, Pentium 4

Sometimes results seem stable, but they still change much later during execution.



Ping benchmark, Linux, TAO/C++, Dual Pentium 3 SMP

Sometimes results seem stable, but they still change much later during execution.

Dealing with warm-up

- Identify #iterations affected by initialization
  - Using run-sequence plots of different scales
  - Validating on several runs
  - Only obvious initialization, after which results are stable/similar
- Identify #iterations to independent state
  - Using acf, lag.plot, run-sequence plots
  - **Only independent data can be used for summarization using confidence interval**
- Neither stability nor independence is always reached
- If not, can only use 1 number from run for summarization

Implications for experiment design

- If iterations do not reach independence, cannot repeat iterations
  - Can only do repetitions higher, e.g. at execution level
- If iterations reach independence too late, it may not pay off repeating them
  - Can repeat only e.g. at execution level and take a fixed iteration possibly not yet from independent state

Automated detection doesn't work

	Initialized	Independent	Harness	Georges
bloat	2	4	8	∞
chart	3	∞	4	1
eclipse	5	7	7	4
fop	10	180	7	8
hsqldb	6	6	8	15
jython	3	∞	5	2
luindex	13	∞	4	8
lusearch	10	85	7	8
pmd	7	∞	4	1
xalan	6	13	15	139

(DaCapo 2006, OpenJDK, Dual Xeon, Ubuntu – platform dependent)

Ratio of execution times

Comparing two systems

T_{new}	Time on the new system (usually ours). Lower is better.	**58s**
T_{old}	Time on the old/baseline system.	**16s**

$\frac{T_{new}}{T_{old}}$	**0.28 (28%)**	**Ratio of execution times**
$1-\frac{T_{new}}{T_{old}}$	**0.72 (72%)**	**Percentage improvement in execution time**
$\frac{T_{old}}{T_{new}}$	**3.63 (363%, 3.63x)**	**Speedup**
$\frac{T_{old}}{T_{new}}-1$	**2.63 (263%)**	**“Percentage improvement in speed”**

Estimating speedup

The time on the old system is given by random variable $\mathbf{X_{old}}$, on new system by $\mathbf{X_{new}}$. We are estimating

$$\frac{E(X_{old})}{E(X_{new})} \text{ using } \frac{\overline{X_{old}}}{\overline{X_{new}}}$$

```
ast <- as.matrix(read.table("ast.out", row.names=1))
bc <- as.matrix(read.table("bc.out", row.names=1))
pidata <- data.frame(new=bc["pidigits",], old=ast["pidigits",])
f <- function(d, sel) mean(d[sel,"old"])/mean(d[sel,"new"])
b <- boot(pidata, f, R=10000)
ci <- boot.ci(b, conf=0.95, type="perc")
> ci$perc
     conf
[1,] 0.95 250.03 9750.98 1.684363 1.698577
> ci
...
Intervals :
Level     Percentile
95%   ( 1.684,  1.699 )
```

With 95% confidence, the BC is between 1.68x and 1.70x faster than AST when running the pidigits benchmark.

Plots for multiple benchmarks

Compute over benchmarks

Execution time means for Shooutout benchmarks running on BC.

```
> bc <- as.matrix(read.table("bc.out", row.names=1))
> m <- apply(bc, 1, mean)
> m["pidigits"]
pidigits
 62.8277
```

Sum time for all iterations of all runs executed by each DaCapo benchmark

```
> benchTotal <- function(b) {
  files <- list.files(paste("dacapo", b, sep="/"), pattern="*.out")
  sum(sapply(files, function(f) sum(
    readDacapo(paste("dacapo", b, f, sep="/"))
  )))
 }
> st <-sapply(list.files("dacapo"), benchTotal)
> sum(st)/3600
[1] 139.9371
```

Barplot for multiple benchmarks

Barplot for the summary time for all iterations of the benchmark

Using base graphics package

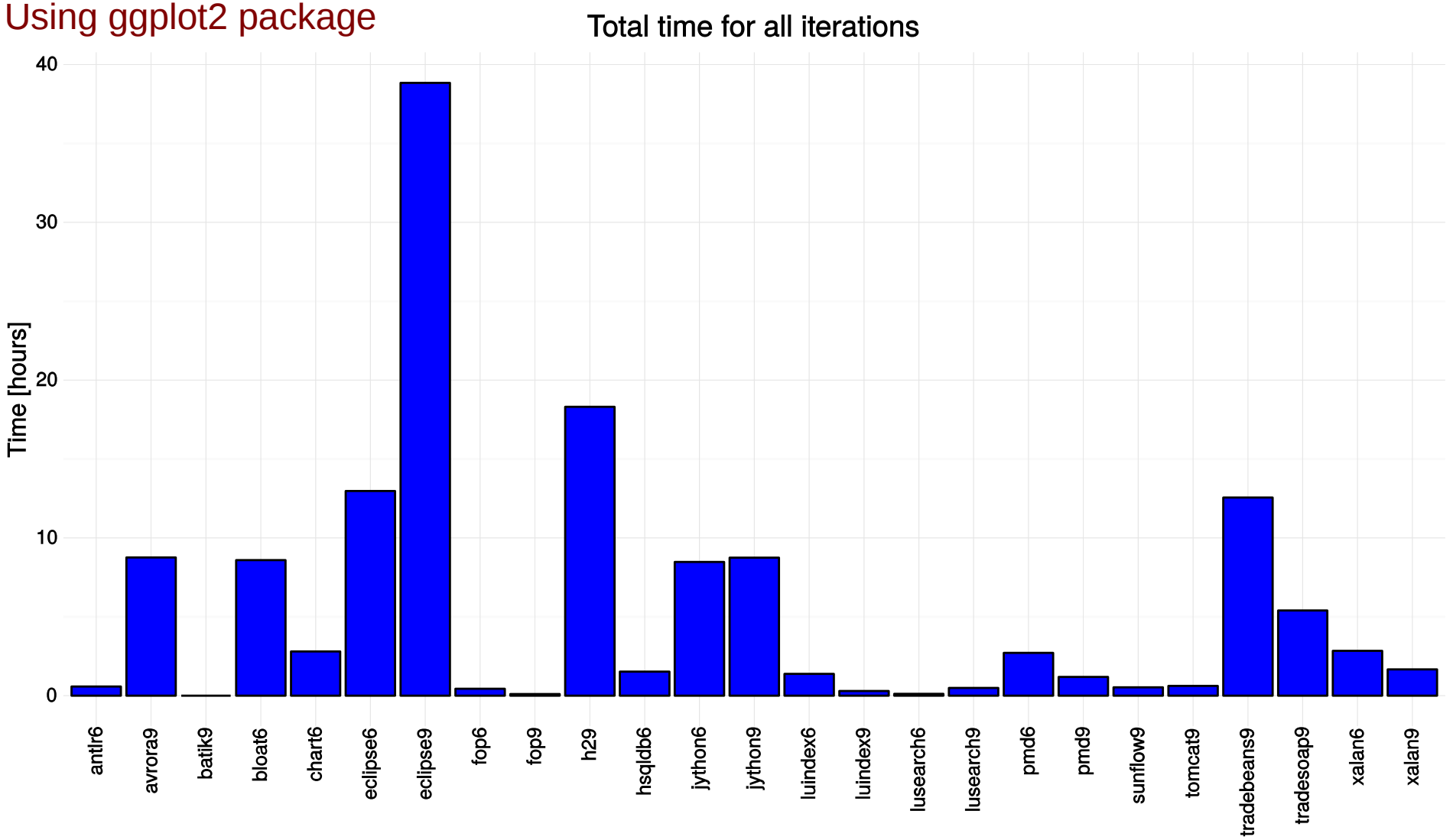
```
par(mar=c(6, 4, 4, 2) + 0.1, las=2)
barplot(t(st/3600), col="blue",
  ylab="Time [hours]", main = "Time for all iterations")
```

Using ggplot2 package

```
df <- cbind(benchmark=names(st), data.frame(time=st/3600))
library(ggplot2)
theme_set(theme_minimal())
theme_update(axis.text.x=element_text(angle=90, hjust=1))
ggplot(df, aes(x=benchmark, y=time)) +
  geom_bar(position=position_dodge(),stat="identity,
           colour=”black”, fill=”blue") +
  xlab("") +
  ylab("Time [hours]") +
  ggtitle("Total time for all iterations")
```

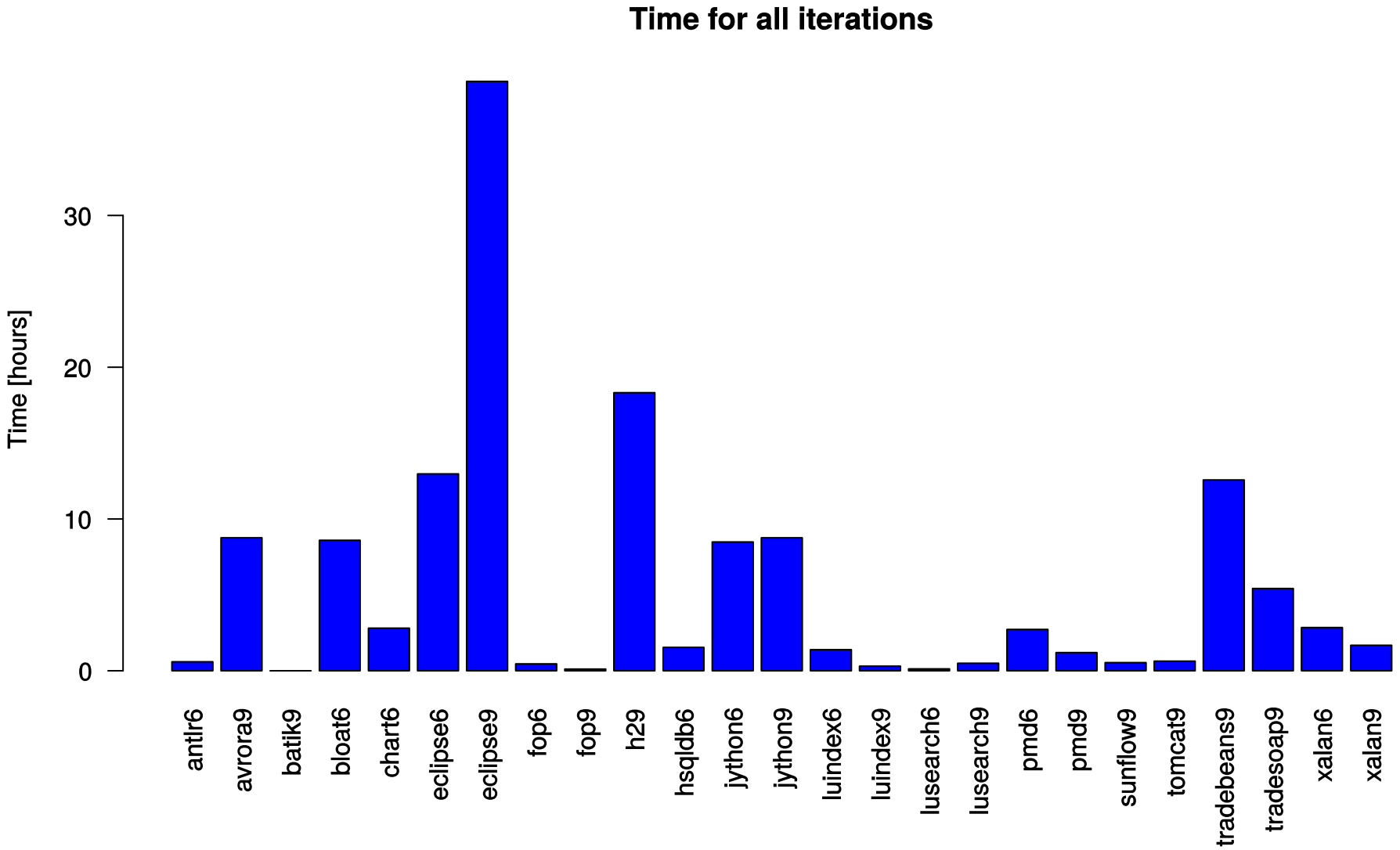
Barplot for multiple benchmarks

Using ggplot2 package



Barplot for multiple benchmarks

Using base graphics package



Barplot with error bars

Barplot for the speedup of BC over AST, with confidence intervals for the ratio of means.

```
cfratio <- function(old, new) { # old/new - vectors
  df <- data.frame(old=old, new=new)
  f <- function(d, sel) mean(d[sel,"old"])/mean(d[sel, "new"])
  b <- boot(df, f, R=10000)
  ci <- boot.ci(b, conf=0.95, type="perc")
  ci$perc[4:5] # low, hi
}

cfratios <- function(old, new) { # old/new - matrices benchmarks x times
  m <- sapply(rownames(old), function(b) {
    o <- old[b,]
    n <- new[b,]
    c(cfratio(o,n), mean(o)/mean(n))
  })
  rownames(m) <- c("low", "hi", "meansRatio")
  t(m) # matrix benchmarks x c("low", "hi", "meansRatio")
}
```

Barplot with error bars

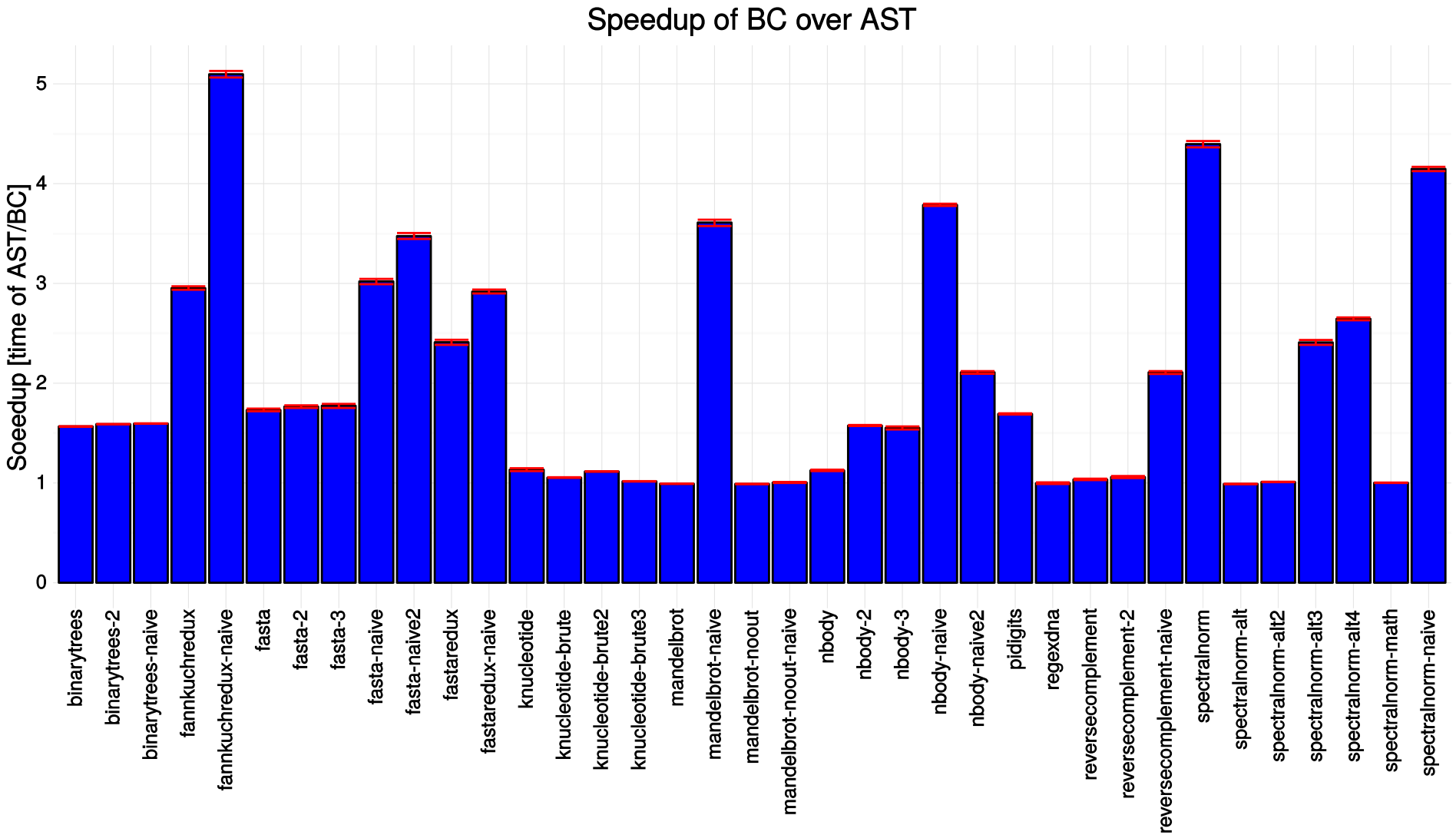
Barplot for the speedup of BC over AST, with confidence intervals for the ratio of means.

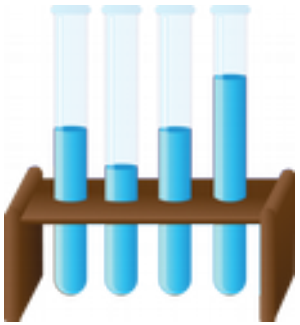
```
plotErrbars <- function(df) {
  library(ggplot2)

  theme_set(theme_minimal())
  theme_update(
    axis.text.x = element_text(angle=90, hjust=1)
  )
  ggplot(df, aes(x=row.names(df), y=meansRatio)) +
    geom_bar(position=position_dodge(), stat="identity",
      colour="black", fill="blue") +
    geom_errorbar(aes(ymin=low, ymax=hi), position=position_dodge(),
      colour="red") +
    xlab("") +
    ylab("Speedup [time of AST/BC]") +
    ggtitle("Speedup of BC over AST")
}

m <- cfratios(ast, bc)
plotErrbars(as.data.frame(m))
```

Barplot with error bars





Exercise

```
help(layout)
```

Use “layout” to stack graphs in a plot when convenient.

1. Identify the number of iterations needed to reach stable and independent state (at least) for these DaCapo9 benchmarks: luindex, jython, tomcat, sunflow
2. Calculate relative variations for R benchmarks run using the AST interpreter
3. Calculate bootstrap confidence intervals for the medians of times for R benchmarks using the BC interpreter
4. Plot these confidence intervals and the medians in a barplot