LiNGAM and OT-LiNGAM on a simulated structural causal model

Observational data and the estimated graph, in R and Python

Every step is done twice, first in **R**, then in **Python**, and the two halves are self-contained. In Python, DirectLiNGAM is the `lingam` package and OT-LiNGAM the `otlingam` package. In R, no package implements DirectLiNGAM (`pcalg::lingam` is the older ICA-based LiNGAM), so the algorithm is written out in a few functions, following the `lingam` package step by step. OT-LiNGAM exists only in Python: it appears in the Python half alone.

1 The structural causal model

Consider the SCM of the lecture, now with **uniform** noises $U_Z, U_X, U_Y \overset{iid}{\sim} \mathcal{U}(0,1)$:

$$\begin{aligned} Z &:= U_Z, \\ X &:= 2Z + U_X, \\ Y &:= 3X + Z + U_Y. \end{aligned}$$

Its graph is $Z \to X \to Y$ with the extra edge $Z \to Y$: Z confounds the effect of X on Y. The noises are non-Gaussian (a uniform variable has excess kurtosis -1.2), which is exactly the assumption under which LiNGAM identifies the causal order from observational data alone. With Gaussian noises the same code would return one member of the Markov equivalence class, here the complete graph on three nodes, with no way to orient it.

R

```r
set.seed(2026)
n <- 2000
BLUE <- "#41597a"; ORANGE <- "#d96125"; GREY <- "#8a8a8a"

simulate <- function(n, do_x = NULL) {
  # Sample the SCM; if do_x is given, replace the equation of X by X := do_x
  UZ <- runif(n); UX <- runif(n); UY <- runif(n)
  Z <- UZ
  X <- if (is.null(do_x)) 2 * Z + UX else rep(do_x, n)
  Y <- 3 * X + Z + UY
  data.frame(Z = Z, X = X, Y = Y)
}
```

```r
obs <- simulate(n)
summary(obs)
```

```
       Z                   X                Y          
 Min.   :0.0008128   Min.   :0.0485   Min.   : 0.4018  
 1st Qu.:0.2472163   1st Qu.:0.9941   1st Qu.: 3.7453  
 Median :0.4996129   Median :1.4953   Median : 5.4740  
 Mean   :0.4979181   Mean   :1.4929   Mean   : 5.4726  
 3rd Qu.:0.7390258   3rd Qu.:1.9988   3rd Qu.: 7.2031  
 Max.   :0.9996828   Max.   :2.9278   Max.   :10.5170  
```

```r
kurt <- function(u) mean((u - mean(u))^4) / mean((u - mean(u))^2)^2 - 3
round(replicate(3, kurt(runif(1e5))), 3)   # excess kurtosis of the noises
```

```
[1] -1.205 -1.205 -1.200
```

```r
par(mfrow = c(1, 3), mar = c(4, 4, 1, 1))
for (pr in list(c("Z", "X"), c("X", "Y"), c("Z", "Y")))
  plot(obs[[pr[1]]], obs[[pr[2]]], pch = 16, cex = 0.3,
       col = adjustcolor(GREY, 0.5), xlab = pr[1], ylab = pr[2])
```

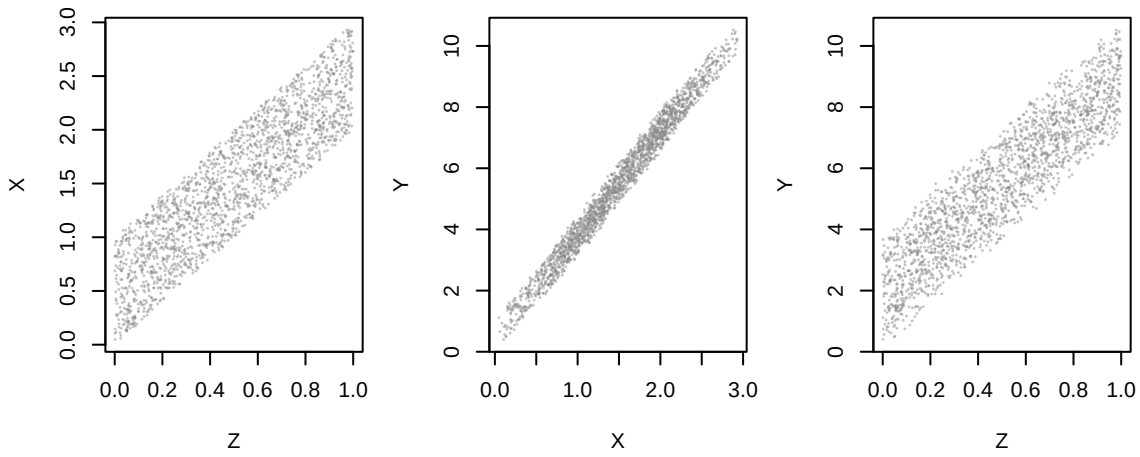

Figure 1: R sample. Pairwise scatter plots of the observational sample. The supports have sharp edges, a signature of the uniform noises: a Gaussian sample would give elliptical clouds.

Python

```python
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
import networkx as nx
from scipy import stats
import lingam
from otlingam import ExhaustiveOTLiNGAM
from otlingam.utils import disorder

BLUE, ORANGE, GREY = "#41597a", "#d96125", "#8a8a8a"
rng = np.random.default_rng(2026)
n = 2000

def simulate(n, rng, do_x=None):
    """Sample the SCM; if do_x is given, replace the equation of X by X := do_x."""
    UZ, UX, UY = rng.uniform(size=(3, n))
    Z = UZ
    X = 2 * Z + UX if do_x is None else np.full(n, float(do_x))
    Y = 3 * X + Z + UY
    return pd.DataFrame({"Z": Z, "X": X, "Y": Y})

obs = simulate(n, rng)
print(obs.describe().round(3).to_string())
```

```
              Z         X         Y
count  2000.000  2000.000  2000.000
mean      0.508     1.512     5.561
std       0.288     0.647     2.214
min       0.001     0.027     0.318
25%       0.259     1.006     3.797
50%       0.511     1.511     5.549
75%       0.755     2.046     7.345
max       1.000     2.971    10.898
```

```
print("excess kurtosis of the noises:",
      np.round(stats.kurtosis(rng.uniform(size=(3, 100_000)), axis=1), 3))
```

```
excess kurtosis of the noises: [-1.202 -1.196 -1.207]
```

```
fig, axes = plt.subplots(1, 3)
for ax, (a, b) in zip(axes, [("Z", "X"), ("X", "Y"), ("Z", "Y")]):
    ax.scatter(obs[a], obs[b], s=1, color=GREY, alpha=0.5)
    ax.set_xlabel(a); ax.set_ylabel(b)
plt.tight_layout(); plt.show()
```

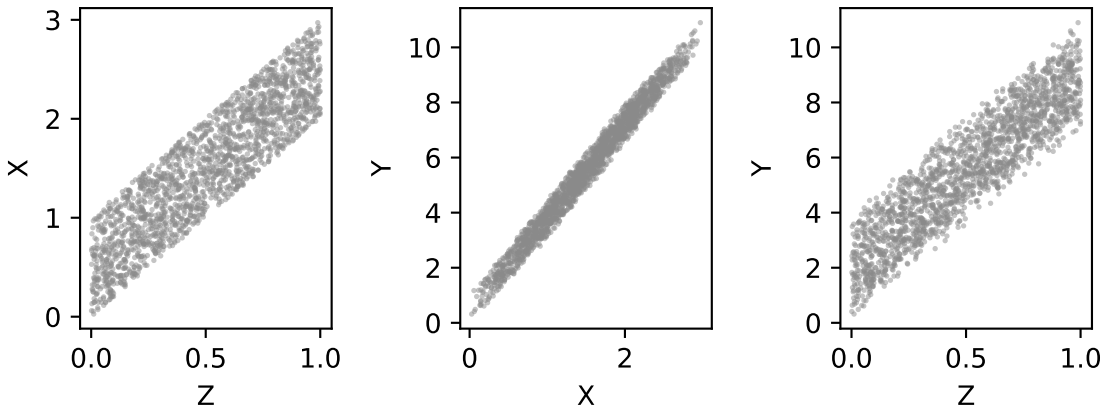

Figure 2: Python sample. The same pairwise scatter plots.

2 Estimating the SCM with LiNGAM and OT-LiNGAM

Both estimators receive only the observational sample. To make the problem non-trivial, the columns are given in the order (Y, Z, X), so the estimators cannot read the causal order off the data matrix. In the adjacency matrix B, entry $B[j, k]$ is the coefficient of variable k in the equation of variable j.

- **DirectLiNGAM** (Shimizu et al., 2011) builds the causal order one variable at a time. For every pair of remaining variables (x_i, x_j), both standardised, it compares the two regression directions with the pairwise likelihood ratio of Hyvärinen & Smith (2013),

  $$\Delta_{ij} = [H(x_j) + H(r_{i|j})] - [H(x_i) + H(r_{j|i})],$$

  where $r_{i|j}$ is the standardised residual of x_i regressed on x_j and H is a maximum-entropy approximation of the differential entropy. $\Delta_{ij} < 0$ is evidence against x_i being a cause of x_j. The variable with the smallest penalty $\sum_j \min(0, \Delta_{ij})^2$ is taken as the most exogenous one and regressed out of all the others. This is the default measure of the `lingam` package (`measure="pwling"`).
- **OT-LiNGAM** (Laplante, Ambroise & Humbert) uses the Wasserstein distance to the Gaussian as a measure of non-Gaussianity; `ExhaustiveOTLiNGAM` searches all causal orders, which is cheap for three variables.

Once the order is found, both packages estimate the coefficients the same way: each variable is regressed on its predecessors by an adaptive lasso with a BIC choice of the penalty, which selects the parents, then by ordinary least squares on the selected parents. Python uses scikit-learn's `LassoLarsIC`, the R code below uses `glmnet` with the same BIC.

R

```r
library(glmnet)

pop_sd <- function(x) sqrt(mean((x - mean(x))^2))      # numpy's np.std
standardise <- function(x) (x - mean(x)) / pop_sd(x)
residual <- function(xi, xj)                             # xi regressed on xj
  xi - mean((xi - mean(xi)) * (xj - mean(xj))) / pop_sd(xj)^2 * xj
```

```r
entropy <- function(u)                      # maximum-entropy approximation
  (1 + log(2 * pi)) / 2 - 79.047 * (mean(log(cosh(u))) - 0.37457)^2 -
    7.4129 * mean(u * exp(-u^2 / 2))^2

diff_mutual_info <- function(xi, xj, ri_j, rj_i)
  (entropy(xj) + entropy(ri_j / pop_sd(ri_j))) -
    (entropy(xi) + entropy(rj_i / pop_sd(rj_i)))

search_causal_order <- function(X, U) {
  if (length(U) == 1) return(U)
  penalty <- sapply(U, function(i) {
    sum(sapply(setdiff(U, i), function(j) {
      xi <- standardise(X[, i]); xj <- standardise(X[, j])
      min(0, diff_mutual_info(xi, xj, residual(xi, xj), residual(xj, xi)))^2
    }))
  })
  U[which.min(penalty)]
}

bic_ols <- function(y, Z) {                 # BIC of a no-intercept linear fit
  rss <- if (ncol(Z) == 0) sum(y^2) else sum(lm.fit(Z, y)$residuals^2)
  length(y) * log(rss / length(y)) + log(length(y)) * ncol(Z)
}

adaptive_lasso <- function(X, predictors, target) {
  Xs <- apply(X, 2, standardise)
  y <- Xs[, target]; Z <- Xs[, predictors, drop = FALSE]
  w <- abs(lm.fit(Z, y)$coefficients)       # adaptive weights
  if (length(predictors) == 1) {            # path: empty model or full model
    keep <- bic_ols(y, Z) < bic_ols(y, Z[, 0, drop = FALSE])
  } else {
    Zw <- sweep(Z, 2, w, "*")
    # explicit grid: glmnet's own path stops early and can miss late entries
    lambda_max <- max(abs(crossprod(Zw, y))) / length(y)
    path <- glmnet(Zw, y, intercept = FALSE, standardize = FALSE,
                   lambda = lambda_max * 10^seq(0, -8, length.out = 200))
    rss <- colSums((y - predict(path, Zw))^2)
    bic <- length(y) * log(rss / length(y)) + log(length(y)) * path$df
    keep <- as.vector(path$beta[, which.min(bic)] != 0)
  }
  coef <- numeric(length(predictors))
  if (any(keep)) {                          # least squares on the original scale
    fit <- lm.fit(cbind(1, X[, predictors[keep], drop = FALSE]), X[, target])
    coef[keep] <- fit$coefficients[-1]
  }
  coef
}
```

```r
direct_lingam <- function(X) {
  p <- ncol(X); U <- seq_len(p); K <- integer(0); Xr <- X
  for (step in seq_len(p)) {
    m <- search_causal_order(Xr, U)
    for (i in setdiff(U, m)) Xr[, i] <- residual(Xr[, i], Xr[, m])
    K <- c(K, m); U <- setdiff(U, m)
  }
  B <- matrix(0, p, p, dimnames = list(colnames(X), colnames(X)))
  for (s in seq_len(p)[-1])
    B[K[s], K[seq_len(s - 1)]] <- adaptive_lasso(X, K[seq_len(s - 1)], K[s])
  list(causal_order = K, B = B)
}
```

```r
names3 <- c("Y", "Z", "X")
D <- as.matrix(obs[, names3])

B_true <- matrix(0, 3, 3, dimnames = list(names3, names3))
B_true["X", "Z"] <- 2; B_true["Y", "X"] <- 3; B_true["Y", "Z"] <- 1

fit <- direct_lingam(D)
cat("causal order:", paste(names3[fit$causal_order], collapse = " < "), "\n")
```

```
causal order: Z < X < Y
```

```r
order3 <- c("Z", "X", "Y")
round(B_true[order3, order3], 3)
```

```
  Z X Y
Z 0 0 0
X 2 0 0
Y 1 3 0
```

```r
round(fit$B[order3, order3], 3)
```

```
      Z     X Y
Z 0.000 0.000 0
X 2.001 0.000 0
Y 1.043 2.969 0
```

Python

```python
names = ["Y", "Z", "X"]
D = obs[names].to_numpy()

B_true = pd.DataFrame(0.0, index=names, columns=names)
B_true.loc["X", "Z"], B_true.loc["Y", "X"], B_true.loc["Y", "Z"] = 2, 3, 1
```

```python
models = {"DirectLiNGAM": lingam.DirectLiNGAM().fit(D),
          "OT-LiNGAM": ExhaustiveOTLiNGAM().fit(D)}

B_hat = {}
for label, m in models.items():
    B_hat[label] = pd.DataFrame(m.adjacency_matrix_, index=names, columns=names)
    order = " < ".join(names[i] for i in m.causal_order_)
    print(f"{label:13s} causal order: {order}   "
          f"disorder: {disorder(B_true.to_numpy(), m.causal_order_)}")
```

```
DirectLiNGAM  causal order: Z < X < Y   disorder: 0
OT-LiNGAM     causal order: Z < X < Y   disorder: 0
```

```python
order3 = ["Z", "X", "Y"]
for label, B in [("true", B_true), *B_hat.items()]:
    print(f"\nB ({label}), rows = effects, columns = causes")
    print(B.loc[order3, order3].round(3).to_string())
```

```
B (true), rows = effects, columns = causes
     Z    X    Y
Z  0.0  0.0  0.0
X  2.0  0.0  0.0
Y  1.0  3.0  0.0

B (DirectLiNGAM), rows = effects, columns = causes
       Z      X    Y
Z  0.000  0.000  0.0
X  2.019  0.000  0.0
Y  0.966  2.999  0.0

B (OT-LiNGAM), rows = effects, columns = causes
       Z      X    Y
Z  0.000  0.000  0.0
X  2.019  0.000  0.0
Y  0.966  2.999  0.0
```

All three fits recover the causal order $Z < X < Y$ (disorder 0) and the three coefficients within a few hundredths. In Python, DirectLiNGAM and OT-LiNGAM return identical matrices: they share the coefficient step and differ only in how they search for the order. The R and Python estimates differ slightly because the two samples differ (R and numpy have different random generators).

3 The estimated graphs

R

```r
library(igraph)
pos <- rbind(Z = c(0, 0), X = c(1, 0.9), Y = c(2, 0))

draw <- function(B, title, color) {
  # B[j, k] != 0 is the edge k -> j
  E <- which(abs(B) >= 0.05, arr.ind = TRUE)
  g <- graph_from_edgelist(cbind(colnames(B)[E[, 2]], rownames(B)[E[, 1]]))
  # edge labels at the midpoints, pushed away from the centre of the triangle
  ends <- ends(g, E(g))
  mid <- (pos[ends[, 1], , drop = FALSE] + pos[ends[, 2], , drop = FALSE]) / 2
  away <- sweep(mid, 2, colMeans(pos))
  lab <- mid + 0.15 * away / sqrt(rowSums(away^2))
  plot(g, layout = pos[V(g)$name, ], rescale = FALSE,
       edge.label.x = lab[, 1], edge.label.y = lab[, 2],
       xlim = c(-0.3, 2.3), ylim = c(-0.3, 1.2), asp = 0,
       vertex.color = "white", vertex.frame.color = color, vertex.size = 40,
       vertex.label.color = "black", vertex.label.cex = 1.2,
       edge.color = color, edge.width = 2, edge.arrow.size = 0.6,
       edge.label = sprintf("%.2f", B[E]), edge.label.color = color,
       edge.label.cex = 0.9, main = title)
}

par(mfrow = c(1, 2), mar = c(0, 0, 1.5, 0))
draw(B_true, "True SCM", GREY)
draw(fit$B, "DirectLiNGAM", BLUE)
```

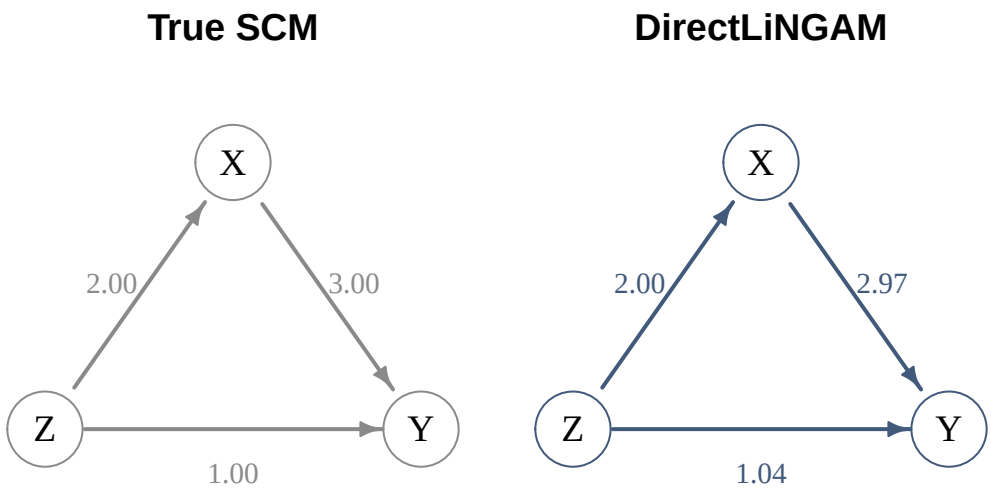

Figure 3: R sample. True graph and graph estimated by DirectLiNGAM (R code). Edge labels are the coefficients; edges with $|B[j,k]| < 0.05$ are not drawn.

Python

```python
pos = {"Z": (0, 0), "X": (1, 0.9), "Y": (2, 0)}

def draw(ax, B, title, color):
```

```
    G = nx.DiGraph()
    G.add_nodes_from(pos)
    for j in names:
        for k in names:
            if abs(B.loc[j, k]) >= 0.05:
                G.add_edge(k, j, w=B.loc[j, k])
    nx.draw_networkx_nodes(G, pos, ax=ax, node_color="white", edgecolors=color,
                           linewidths=2, node_size=900)
    nx.draw_networkx_labels(G, pos, ax=ax, font_size=13)
    nx.draw_networkx_edges(G, pos, ax=ax, edge_color=color, width=2, arrowsize=18,
                           node_size=900)
    nx.draw_networkx_edge_labels(G, pos, ax=ax, font_size=10, font_color=color,
                                 edge_labels={(k, j): f"{w:.2f}"
                                              for k, j, w in G.edges(data="w")})
    ax.set_title(title, fontsize=11); ax.set_xlim(-0.4, 2.4); ax.set_ylim(-0.4, 1.3)
    ax.axis("off")

fig, axes = plt.subplots(1, 3)
draw(axes[0], B_true, "True SCM", GREY)
draw(axes[1], B_hat["DirectLiNGAM"], "DirectLiNGAM", BLUE)
draw(axes[2], B_hat["OT-LiNGAM"], "OT-LiNGAM", ORANGE)
plt.tight_layout(); plt.show()
```

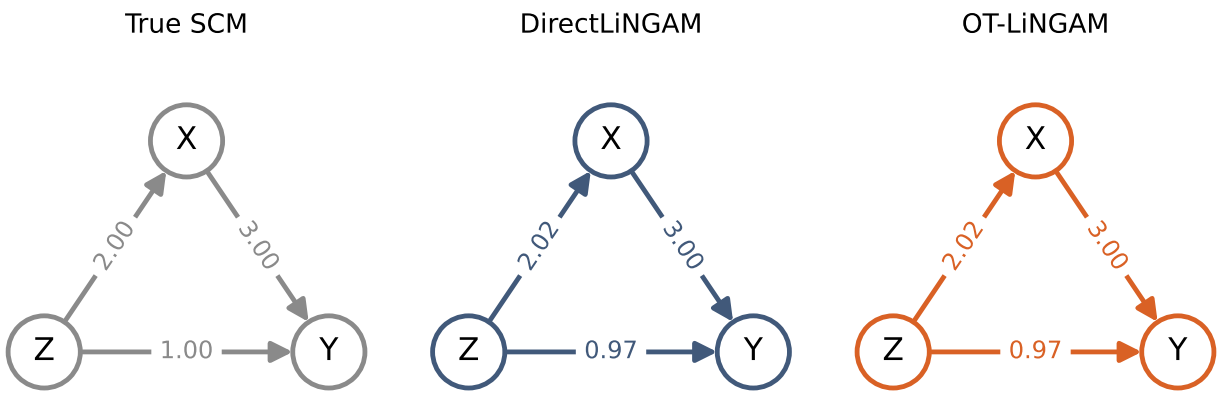

Figure 4: Python sample. True graph and graphs estimated by DirectLiNGAM and OT-LiNGAM. Edge labels are the coefficients; edges with $|B[j,k]| < 0.05$ are not drawn.