UNIVERSITY OF
COPENHAGEN

Object-Oriented Programming and Density Estimation

Computational Statistics

Johan Larsson

Department of Mathematical Sciences, University of Copenhagen

August 26, 2025

Today's Agenda

Object-Oriented Programming (OOP)

We learn what OOP is and how the S3 OOP system in R works.

Kernel Density Estimation

We learn how to create non-parametric density estimates using kernel density estimation

Object-Oriented Programming

What Is Object-Oriented Programming (OOP)?

A programming paradigm based on the concept of "objects", which contain data and methods.

Data
Contains attributes and properties (e.g. name, type, size, color, etc.)

Methods
Functions that operate on the data (e.g. print, plot, summarize, etc.).

Benefits of OOP
OOP allows us to bundle data and methods together.

It enables **inheritance**, which means that different classes can share functionality.

It also facilitates **polymorphism**, which means that the same method can behave differently for different classes.

In Base R

There are three OOP systems in base R:

- S3
- S4
- Reference Classes

Through Packages

- R6
- S7

This Course

In this course, we focus **entirely** on S3: a *very* informal OOP system.

Example: Integration

The function `integrate()` takes a function as argument and returns the value of numerically integrating the function. It is an example of a *functional*.

```
integral <- integrate(sin, 0, 1)
integral
```

```
0.4596977 with absolute error < 5.1e-15
```

The numerical value of the integral

$$\int_0^1 \sin(x)\mathrm{d}x$$

is helpfully printed—including an indication of the numerical error.

Return Values

In fact, `integrate()` returns a class object: a list with a *class label*.

```
str(integral)
```

```
List of 5
 $ value       : num 0.46
 $ abs.error   : num 5.1e-15
 $ subdivisions: int 1
 $ message     : chr "OK"
 $ call        : language integrate(f = sin, lower = 0, upper = 1)
 - attr(*, "class")= chr "integrate"
```

What is the Point of This Class Label?
It allows us to write functions that work differently depending on the class of the argument.

If `x` is an object of class `numeric`, then do A. If `x` is an object of class `integrate`, then do B.

The Return Value of `integrate()`

We can extract the class label with `class()`.

```
class(integral)
```

```
[1] "integrate"
```

The printed result of `integrate()` is not the same as the object itself. What you see is the result a **method**, `print.integrate()`.

Printing Objects of Class Integrate

```
stats:::print.integrate
```

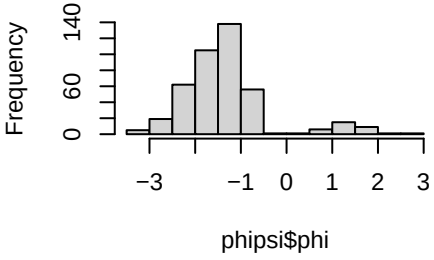
```
function (x, digits = getOption("digits"), ...)
{
    if (x$message == "OK")
        cat(format(x$value, digits = digits), " with absolute error < ",
            format(x$abs.error, digits = 2L), "\n", sep = "")
    else cat("failed with message ", sQuote(x$message), "\n",
        sep = "")
    invisible(x)
}
<bytecode: 0x323eb2c8>
<environment: namespace:stats>
```

(The `print.integrate()` function is not exported from the stats package. It is in the namespace of the **stats** package, and to access it directly we use `stats:::`.)

Histogram Objects

`hist()` also returns an object (of class `histogram`)

```
phi_hist <- hist(phipsi$phi, main = NULL)
```




```
class(phi_hist)
```

```
[1] "histogram"
```

Histogram Object Structure

```
str(phi_hist)
```

```
List of 6
 $ breaks  : num [1:14] -3.5 -3 -2.5 -2 -1.5 -1 -0.5 0 0.5 1 ...
 $ counts  : int [1:13] 5 19 62 105 138 56 1 1 6 15 ...
 $ density : num [1:13] 0.0239 0.0907 0.2959 0.5012 0.6587 ...
 $ mids    : num [1:13] -3.25 -2.75 -2.25 -1.75 -1.25 -0.75 -
0.25 0.25 0.75 1.25 ...
 $ xname   : chr "phipsi$phi"
 $ equidist: logi TRUE
 - attr(*, "class")= chr "histogram"
```

We Can Extract Elements

```
phi_hist[1:3]
```

```
$breaks
 [1] -3.5 -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3

$counts
 [1]   5  19  62 105 138  56   1   1   6  15   9   1   1

$density
 [1] 0.02386635 0.09069212 0.29594272 0.50119332 0.65871122 0.26730310
 [7] 0.00477327 0.00477327 0.02863962 0.07159905 0.04295943 0.00477327
[13] 0.00477327
```

Getting Help for Objects

You can find documentation for `plot()` using e.g.

```
?plot
```

However, this will be uninformative on how an object of class histogram is plotted. Try instead

```
?plot.histogram
```

This will give the documentation for the plot method for objects of class histogram.

S3 Overview

S3 classes are standard data structures (typically lists) with *class labels*.

It is an informal system. Objects are not formally defined, and there is no type checking.

To use S3, we implement a *generic* function via `UseMethod()`. E.g.

```
plot
```

```
function (x, y, ...)  UseMethod("plot")
```

Methods for specific classes are implemented as standard R functions with the naming convention `f.classname()` for a method for class `classname` of the function `f()`.

The system is widely used to write methods for the generic functions `print()`, `plot()` and `summary()`.

Constructing a New Class

Recall the function `count_zeros_vec()` that counts the number of zeros in a vector.

If we need this number many times it is beneficial to compute it once and then extract it whenever needed.

We first write a *constructor function* that returns a list with a class label.

```
count_object <- function(x) {
  structure(
    list(
      x = x,
      n = count_zeros_vec(x)
    ),
    class = "count_object"
  )
}
```

A Data Example

```
count_data <- count_object(rpois(10, 2))
count_data
```

```
$x
 [1] 0 2 2 2 4 2 0 1 2 2

$n
[1] 2

attr(,"class")
[1] "count_object"
```

The Generic Function

To activate looking up a method for a specific class, one needs to tell R that the function `count_zeros()` is a *generic function*.

```
count_zeros <- function(x) {
  UseMethod("count_zeros")
}
```

We can let the default method be the vectorized version.

```
count_zeros.default <- function(x) {
  count_zeros_vec(x)
}
```

Then we can implement a class-specific version for the class `count_object`.

```
count_zeros.count_object <- function(x) {
  x[["n"]]
}
```

A Print Method

And we can also implement a print method.

```
print.count_object <- function(x) {
  cat("Values:", x$x, "\nNumber of zeros:", x$n, "\n")
}
```

```
count_data # Invokes print.count_object()
```

```
Values: 0 2 2 2 4 2 0 1 2 2
Number of zeros: 2
```

```
count_zeros(count_data) # Invokes count_zeros.count_object()
```

```
[1] 2
```

Density Estimation

Density Estimation

Let f_0 denote the unknown desnity we want to estimate.

- If we fit a parameterize statistical model $(f_\theta)_\theta$ to data using the estimator $\hat{\theta}$, then $f_{\hat{\theta}}$ is an estimate of f_0.
- The histogram is a nonparameteric density estimator, $\hat{f}$.
- We are interested in nonparametric estimators because
  - we want to compare data with the parametric estimate,
  - we don't know a suitable parametric model, and
  - we want aid in visualization.

Density estimation relies on the approximation

$$P(X \in (x-h, x+h)) = \int_{x-h}^{x+h} f_0(z)\ dz \simeq f_0(x)2h.$$

Rearranging and using the LLN gives

$$\begin{aligned} f * 0(x) &\simeq \frac{1}{2h} P(X \in (x-h, x+h)) \\ &\simeq \frac{1}{2h}\frac{1}{n}\sum *i = 1^n 1 * (x-h, x+h)(x_i) \\ &= \frac{1}{2hn}\sum *i = 1^n 1\_(-h, h)(x - x_i) = \hat{f}\_h(x) \end{aligned}$$

We will consider *kernel estimators*

$$\hat{f}_h(x) = \frac{1}{hn} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right).$$

The *uniform* or *rectangular kernel* is

$$K(x) = \frac{1}{2} 1_{(-1,1)}(x),$$

which leads to the expression on the last slide.
The *Gaussian kernel* is

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}.$$

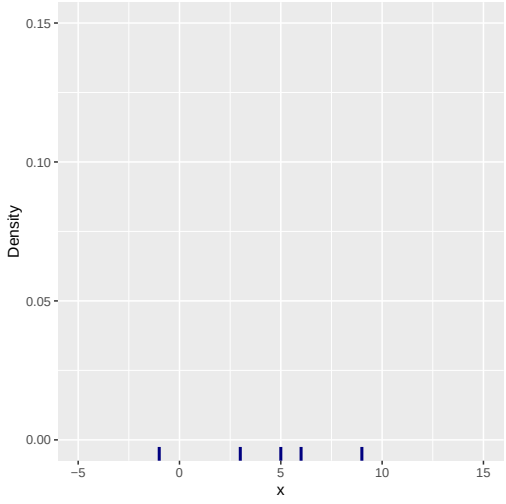
Implementation with the Gaussian Kernel

```
kern_dens <- function(x, h, m = 512) {
  rg <- range(x)
  xx <- seq(rg[1] - 3 * h, rg[2] + 3 * h, length.out = m)
  y <- numeric(m)

  for (i in seq_along(xx)) {
    for (j in seq_along(x)) {
      y[i] <- y[i] + exp(-(xx[i] - x[j])^2 / (2 * h^2))
    }
  }

  y <- y / (sqrt(2 * pi) * h * length(x))

  list(x = xx, y = y)
}
```

An Illustration

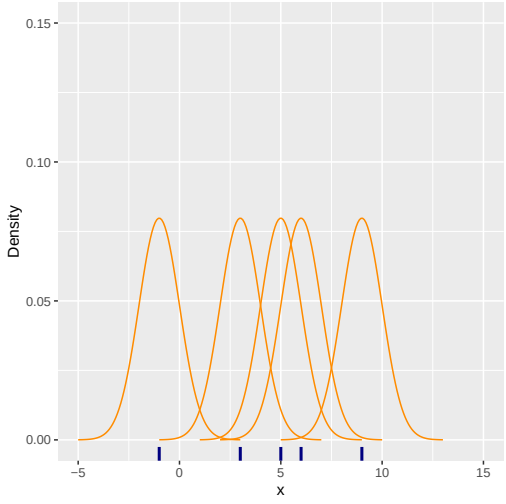
Let's say we have a data set

$$\boldsymbol{x} = (-1, 3, 5, 6, 9).$$



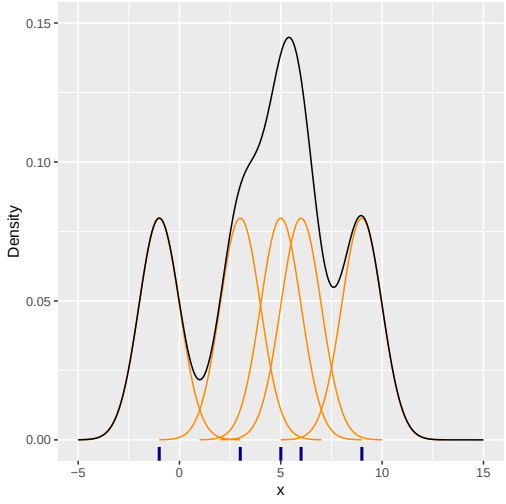
An Illustration: Gaussian Kernel

We add a Gaussian density kernel with bandwidth 1 for each point.



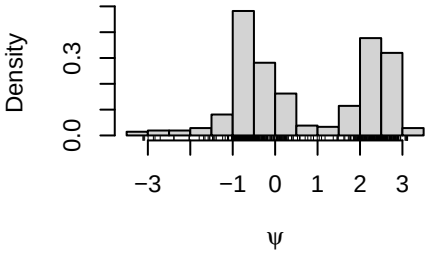
An Illustration: Gaussian Kernel

Finally, we average the kernels.



Angle Data

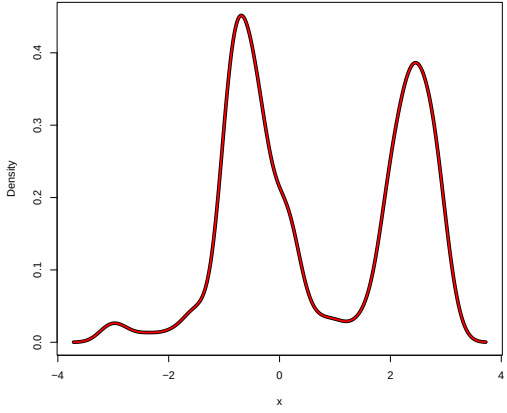
```
hist(phipsi$psi, prob = TRUE, xlab = expression(psi), main = NULL)
rug(phipsi$psi)
```



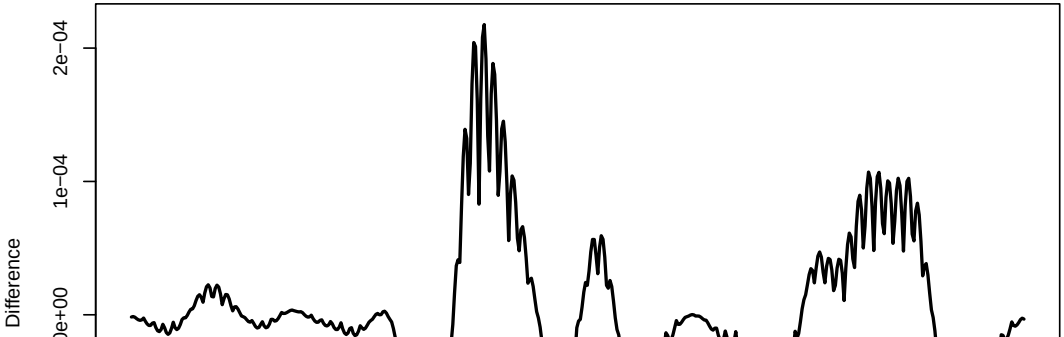
A First Test

```
f_hat <- kern_dens(phipsi$psi, 0.2)
f_hat_dens <- density(phipsi$psi, 0.2)
```

```
plot(
  f_hat,
  type = "l",
  lwd = 4,
  xlab = "x",
  ylab = "Density"
)

lines(
  f_hat_dens,
  col = "red",
  lwd = 2
)
```



```
plot(
  f_hat$x,
  f_hat$y - f_hat_dens$y,
  type = "l",
  lwd = 2,
  xlab = "x",
  ylab = "Difference"
)
```



```r
library(testthat)

test_that("Our density implementation corresponds to density()", {
  expect_equal(kern_dens(phipsi$psi, 0.2)$y, density(phipsi$psi, 0.2)$y)
})
```

```
-- Failure: Our density implementation corresponds to density() --------
kern_dens(phipsi$psi, 0.2)$y not equal to density(phipsi$psi, 0.2)$y.
511/512 mismatches (average diff: 4e-05)
[1] 0.000136 - 0.000138 == -1.69e-06
[2] 0.000170 - 0.000171 == -1.29e-06
[3] 0.000211 - 0.000213 == -1.81e-06
[4] 0.000261 - 0.000264 == -2.89e-06
[5] 0.000322 - 0.000325 == -3.76e-06
[6] 0.000394 - 0.000398 == -4.16e-06
[7] 0.000480 - 0.000484 == -3.79e-06
[8] 0.000581 - 0.000584 == -2.44e-06
[9] 0.000701 - 0.000706 == -4.84e-06
```

Tolerance

It is often necessary to allow for small differences in floating point numbers.

```
library(testthat)
test_that("Our density implementation corresponds to density()", {
  expect_equal(
    kern_dens(phipsi$psi, 0.2)$y,
    density(phipsi$psi, 0.2)$y,
    tolerance = 1e-3
  )
})
```

```
Test passed
```

Setting Tolerance Level

- You need to decide on the tolerance level on a case-by-case basis.
- Uses `all.equal()` internally (but depends on testthat edition!). Usually tests **relative** difference (but not always!)

For a parametric family we can use the MLE

$$\hat{\theta} = \text{arg max}_\theta \sum_{j=1}^{n} \log f_\theta(x_j).$$

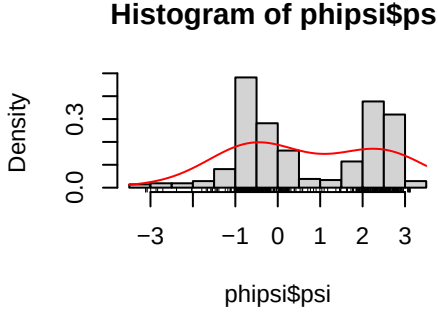
For nonparametric estimation we can still introduce the log-likelihood:

$$\ell(f) = \sum_{j=1}^{n} \log f(x_j)$$

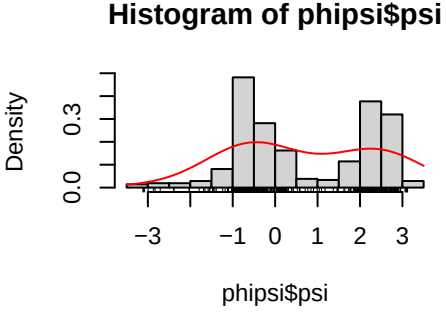
Let's see what happens for the Gaussian kernel density estimate

$$f(x) = f_h(x) = \frac{1}{nh\sqrt{2\pi}} \sum_{j=1}^{n} e^{-\frac{(x-x_j)^2}{2h^2}}.$$

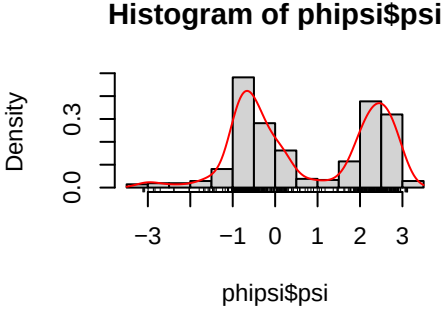
```r
f_h <- function(x, h) mean(dnorm(x, phipsi$psi, h))
f_h <- Vectorize(f_h)
hist(phipsi$psi, prob = TRUE)
rug(phipsi$psi)
curve(f_h(x, 1), add = TRUE, col = "red")
```



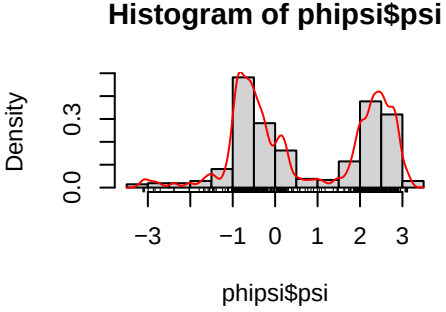
$h = 1$



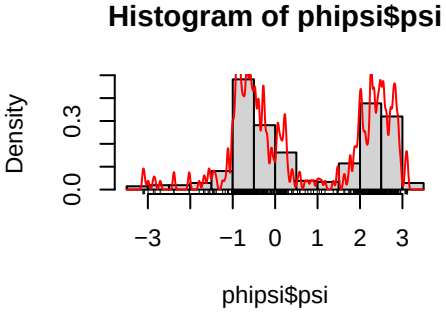
$h = 0.25$



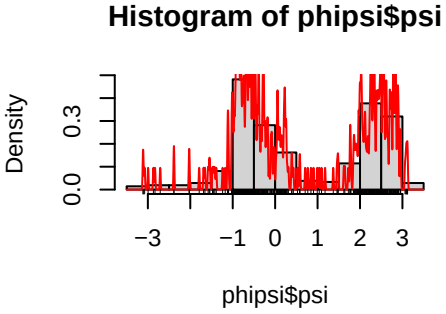
$h = 0.01$

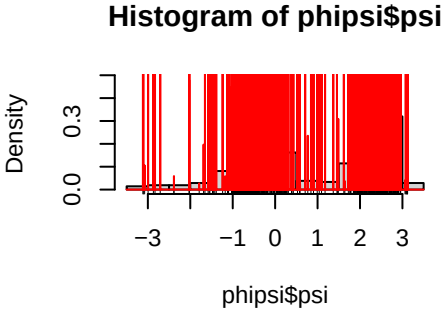


$h = 0.025$



$h = 0.01$




Histogram of phipsi$psi
Density
0.0
0.3
−3
−1
0
1
2
3
phipsi$psi

Log-Likelihood

If $x_i \neq x_j$ when $i \neq j$

$$\begin{aligned}\ell(f * h) &= \sum *i \log\left(1 + \sum \_j \neq i e^{-(x_i - x_j)^2/(2h^2)}\right) - n\log(nh\sqrt{2\pi}) \\ &\sim -n\log(nh\sqrt{2\pi})\end{aligned}$$

for $h \to 0$.

Hence, $\ell(f_h) \to \infty$ for $h \to 0$ and there is **no MLE** in the set of distributions with densities.

ISE, MISE, and MSE

Quality of $\hat{f}_h$ can be quantified by the *integrated squared error*,

$$\mathrm{ISE}(\hat{f}_h) = \int (\hat{f}_h(x) - f_0(x))^2 \, dx = ||\hat{f}_h - f_0||_2^2.$$

Quality of the estimation procedure can be quantified by the mean ISE,

$$\mathrm{MISE}(h) = \mathrm{E}(\mathrm{ISE}(\hat{f}_h)),$$

where the expectation integral is over the data.

$$\mathrm{MISE}(h) = \int \mathrm{MSE}_h(x) \, dx$$

where $\mathrm{MSE}_h(x) = \mathrm{var}(\hat{f}_h(x)) + \mathrm{bias}(\hat{f}_h(x))^2$.
Let's derive the case of the uniform kernel!

AMISE

If K is a square-integrable probability density with mean 0,

$$\mathrm{MISE}(h) = \mathrm{AMISE}(h) + o((nh)^{-1} + h^4)$$

where the *asymptotic mean integrated squared error* is

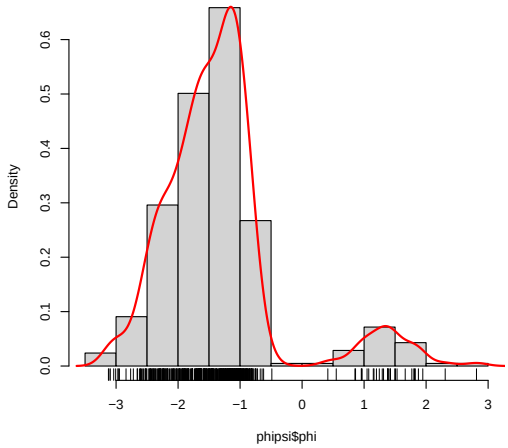
$$\mathrm{AMISE}(h) = \frac{\|K\|_2^2}{nh} + \frac{h^4 \sigma_K^4 \|f_0''\|_2^2}{4}$$

with $\sigma_K^2 = \int t^2 K(t)\, dt$.
Using various plug-in estimates of $\|f_0''\|_2^2$, AMISE can be used to estimate the *asymptotically optimal bandwidth* in a mean integrated squared error sense.

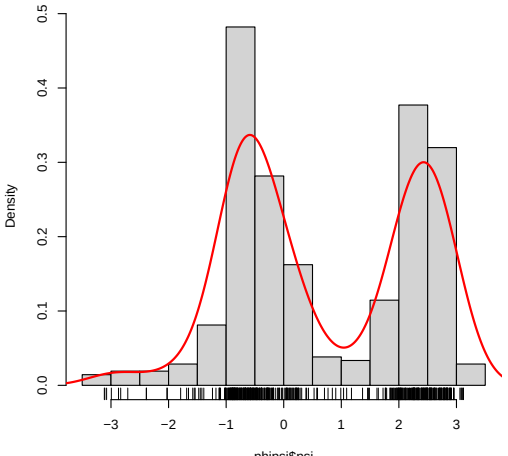
Amino Acid Angles (Silverman, the default)

$$h = 0.9 \min\left(\hat{\sigma}, \frac{\text{IQR}}{1.34}\right) n^{-1/5}$$

```
density(phipsi$phi)
```

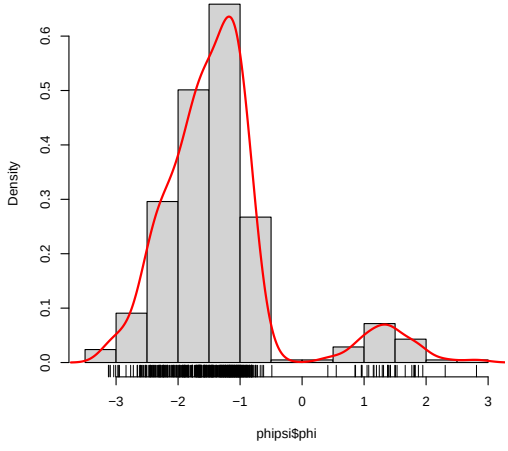



```
density(phipsi$psi)
```

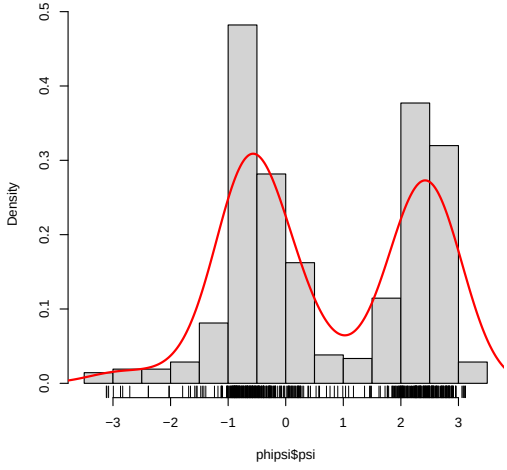


Amino Acid Angles (Scott, using 1.06)

```
density(phipsi$phi, bw = "nrd")
```

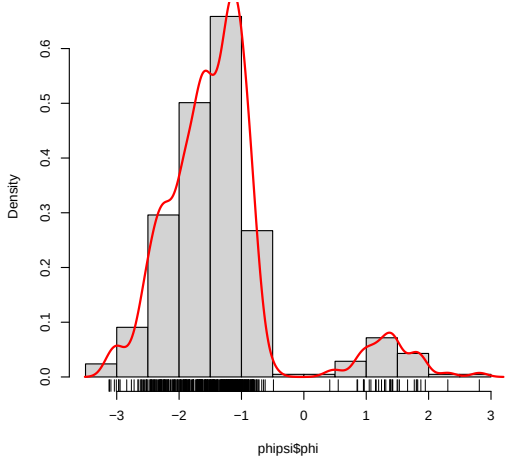



```
density(phipsi$psi, bw = "nrd")
```

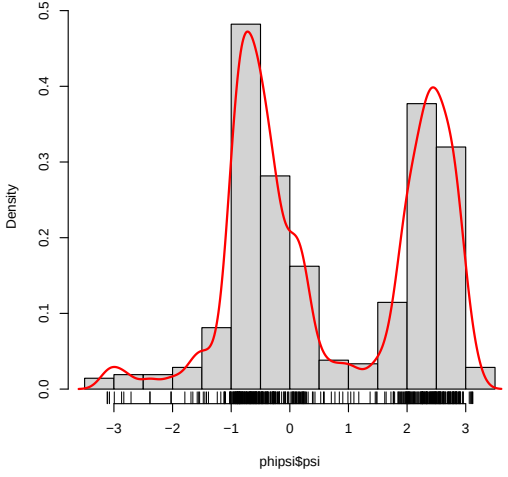


Amino Acid Angles (Sheather & Jones)

```
density(phipsi$phi, bw = "SJ")
```




```
density(phipsi$psi, bw = "SJ")
```



Exercises

Exercise 1

Exercise 1
Create a constructor function called `summarize_vector()` that that takes a numeric vector as input and returns an object of class `vector_summary` containing the mean, median, and standard deviation of the input vector.

Exercise 2
Write a `print()` method for the class `vector_summary` that prints the mean, median, and standard deviation in a neatly formatted way.

Exercise 2

Take the `kern_dens()` function we started out with and implement Silverman's rule of thumb for choosing the bandwidth. Test against the default behavior of `density()`.

```
kern_dens_silverman <- function(x, h = NULL, m = 512) {
  # If h is NULL, compute h using Silverman's rule

  rg <- range(x)
  xx <- seq(rg[1] - 3 * h, rg[2] + 3 * h, length.out = m)
  y <- numeric(m)

  for (i in seq_along(xx)) {
    for (j in seq_along(x)) {
      y[i] <- y[i] + exp(-(xx[i] - x[j])^2 / (2 * h^2))
    }
  }

  y <- y / (sqrt(2 * pi) * h * length(x))

  list(x = xx, y = y)
```

Exercise 2

Multiple Kernels

Let the user choose between the Gaussian and the uniform kernel.

Object

Return an S3 object from your density function and write a plot method for it, using either ggplot2 or base graphics.

Generic

Write a new generic called `my_density()` and provide different methods depending on whether you provide a vector or a matrix, storing density estimates for each column if it is a matrix.
Modify the `plot()` method to handle the new object.