Bivariate Probability Distributions

Abby Spurdle

September 11, 2018

Provides alternatives to persp() for plotting bivariate functions, including both step and continuous functions. Also, provides convenience functions for constructing and plotting bivariate probability distributions. Currently, only normal distributions are supported but other probability distributions are likely to be added in the near future.

Introduction

This package provides alternatives to persp() to support bivariate functions and it also provides convenience functions for constructing and plotting bivariate distributions.

The alternatives to persp() can plot both step and continuous functions.

Currently, there are convenience functions for constructing and plotting normal bivariate probability density functions and cumulative distribution functions. Other distributions are likely to be added in the near future. The functions for constructing probability distributions take means, variances and covariance for x and y and return functions which can be evaluated for x and y. The functions for plotting distributions take a function object and can plot either a contour plot or a 3d plot.

Loading The mvtnorm and bivariate Packages

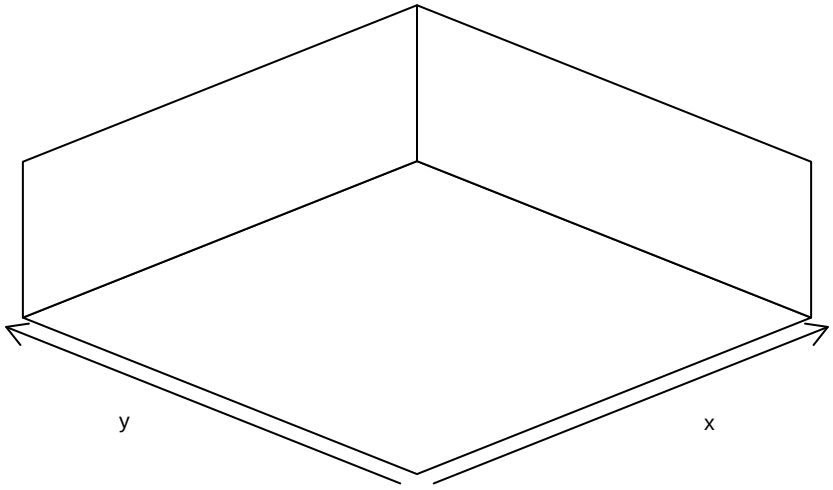
First we need to load the mvtnorm and bivariate packages.

```
> library (mvtnorm)
> library (bivariate)
```

Coordinate System

This package uses a nonstandard coordinate system. It's designed to make it simpler to view cumulative distribution functions.

```
> plot3d.empty ()
```



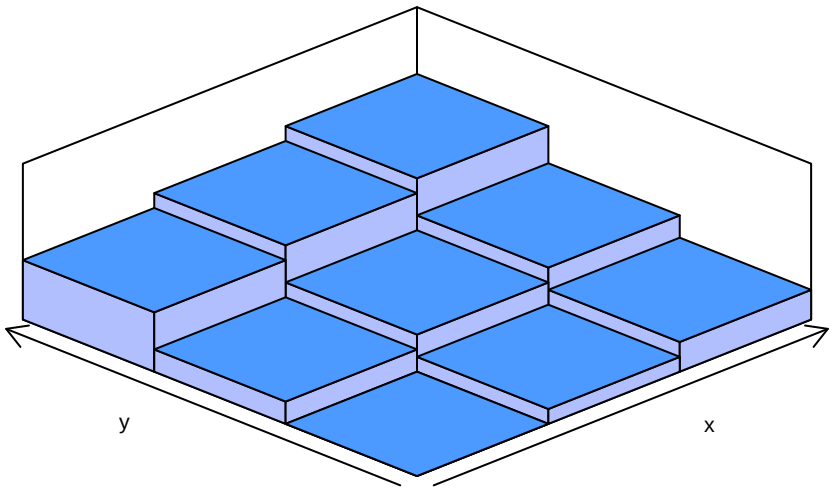
Note the directions of x and y.

Step Functions

We can create 3d plots of bivariate step functions using the plot3d.step.regular() or plot3d.step() functions.

Let's create a simple example that's not really a step function. Note that I'm not trying to explain how to create a bivariate step function rather I'm trying to explain how to create a 3d plot of a bivariate step function.

```
> x = y = 1:4
> f = function (x, y) 2 * x + y ^ 2
> z = outer (x, y, f)
> plot3d.step.regular (z)
```



We can use the plot3d.step() function if our x and y values aren't regularly spaced. Our

first two arguments are vectors of sorted x and y values.

Refer to the help pages for more information.

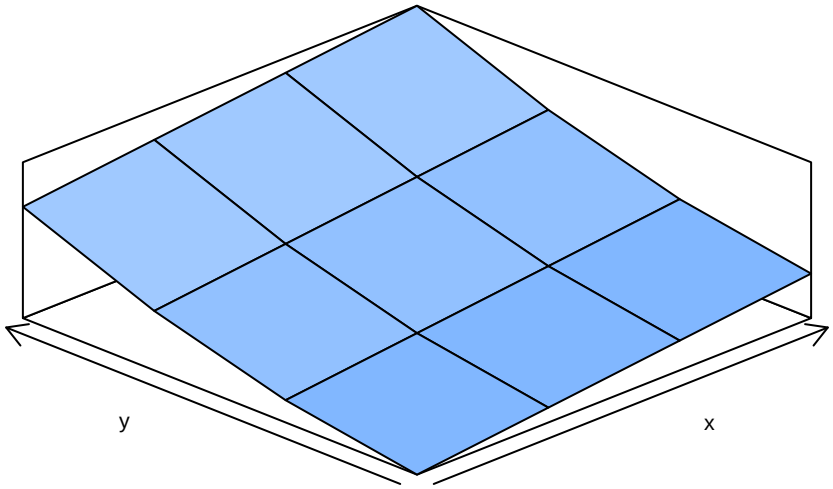
```
> ?plot3d.step
```

Continuous Functions

We can create 3d plots of bivariate continuous functions using the plot3d.continuous.regular() or plot3d.continuous() functions.

Let's use a similar example to the previous section.

```
> x = y = 1:4
> f = function (x, y) 2 * x + y ^ 2
> z = outer (x, y, f)
> plot3d.continuous.regular (z)
```



We can use the plot3d.continuous() function if our x and y values aren't regularly spaced. Our first two arguments are vectors of sorted x and y values.

Refer to the help pages for more information.

```
> ?plot3d.continuous
```

Normal Bivariate Probability Density Functions

We can construct a normal bivariate probability density function using the nbpdf() function. It takes five arguments, the mean of x, the mean of y, the variance of x, the variance of y and the covariance of x and y.

```
> f = nbpdf (0, 0, 1, 1, 0)
```

Alternatively we could use the standard deviations and correlation, using something like:

```
> #f = nbpdf (mean.x, mean,y, sd.x ^ 2, sd.y ^ 2, sd.x * sd.y * cor.xy)
```

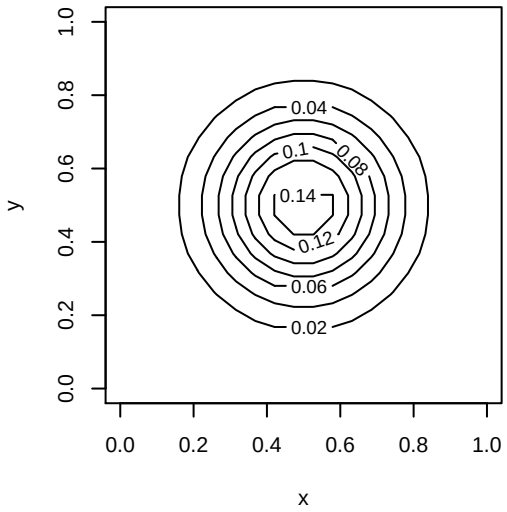
We can print our object out if we want to.

```
> f
```

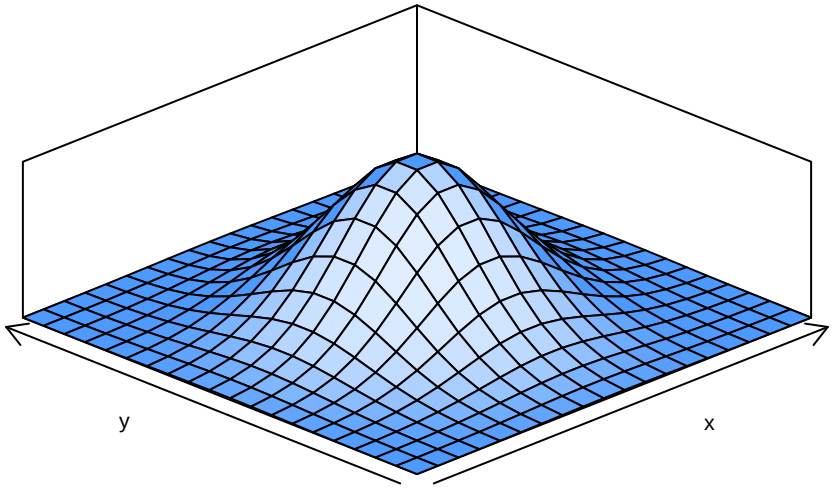
```
function (x, y)
.nbpdf.eval(x, y)
attr(,"class")
[1] "nbpdf"
attr(,"vector.means")
[1] 0 0
attr(,"matrix.variances")
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Once we have created our object we can plot it.

```
> plot (f)
```




```
> plot (f, TRUE)
```



Note that we can change the plot's xlab, ylab, xlim, ylim and colours. Refer to the help page for more information.

```
> ?nbpdf
```

The object f is a function so we can evaluate it.

```
> f (0, 0)
```

```
[1] 0.1591549
```

Note that f's arguments can be vectors.

Normal Bivariate Cumulative Distribution Functions

We can construct a normal bivariate cumulative distribution function using the nbcdf() function. It takes the same arguments as nbpdf().

```
> F = nbcdf (0, 0, 1, 1, 0)
```

Alternatively we could use the standard deviations and correlation, using something like:

```
> #F = nbcdf (mean.x, mean,y, sd.x ^ 2, sd.y ^ 2, sd.x * sd.y * cor.xy)
```

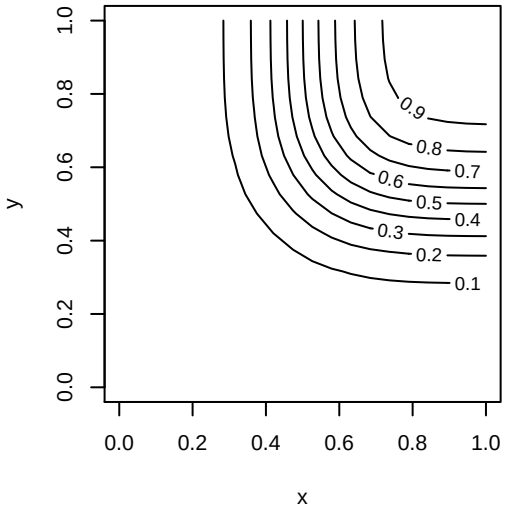
We can print our object out if we want to.

```
> F
```

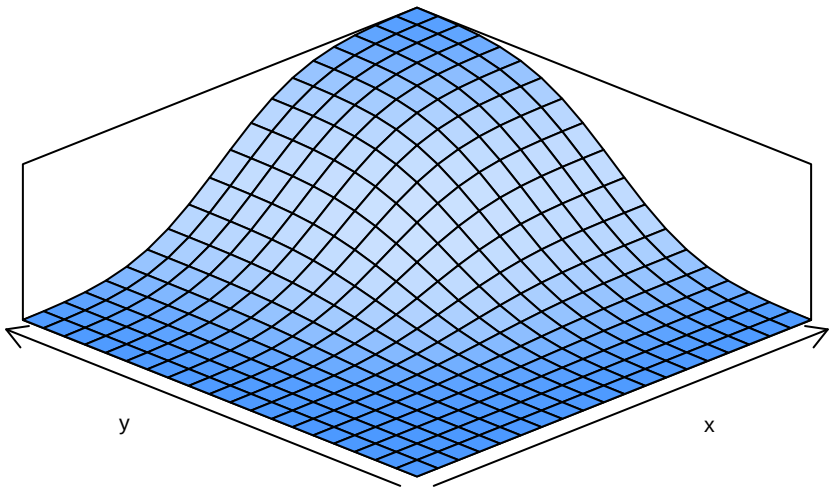
```
function (x, y)
.nbcdf.eval(x, y)
attr(,"class")
[1] "nbcdf"
attr(,"vector.means")
[1] 0 0
attr(,"matrix.variances")
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Once we have created our object we can plot it.

```
> plot (F)
```




```
> plot (F, TRUE)
```



Note that we can change the plot's xlab, ylab, xlim, ylim and colours. Refer to the help page for more information.

```
> ?nbcdf
```

The object F is a function so we can evaluate it.

```
> F (0, 0)

[1] 0.25
```

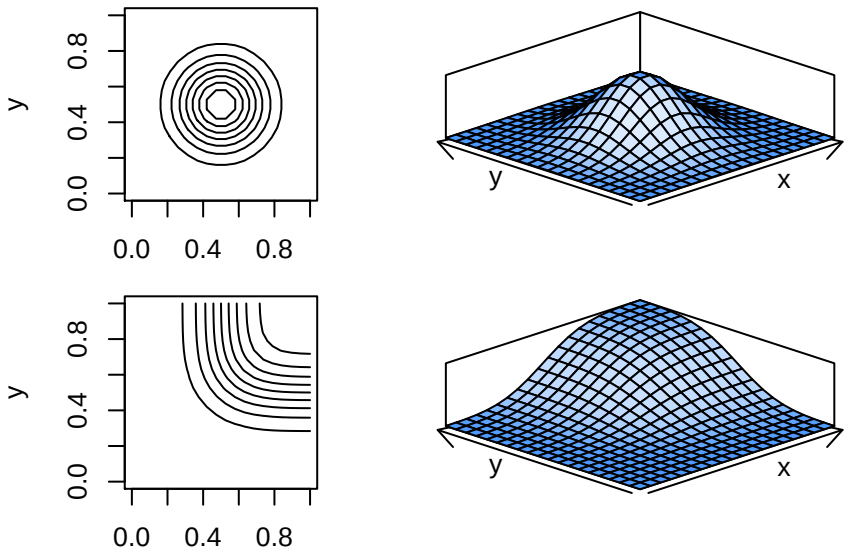
Note that F's arguments can be vectors.

Comparing Different Distributions

We can compare different distributions using different covariance parameters.

First, let's consider the bivariate distributions from the previous sections with zero correlation/covariance.

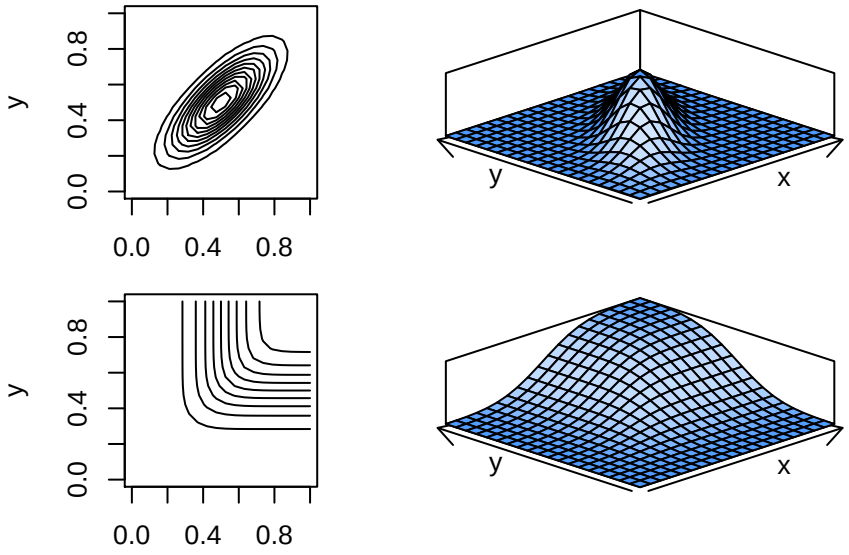
```
> plot (f, all=TRUE)
```



Note that this requires the probability density function rather than the cumulative distribution function.

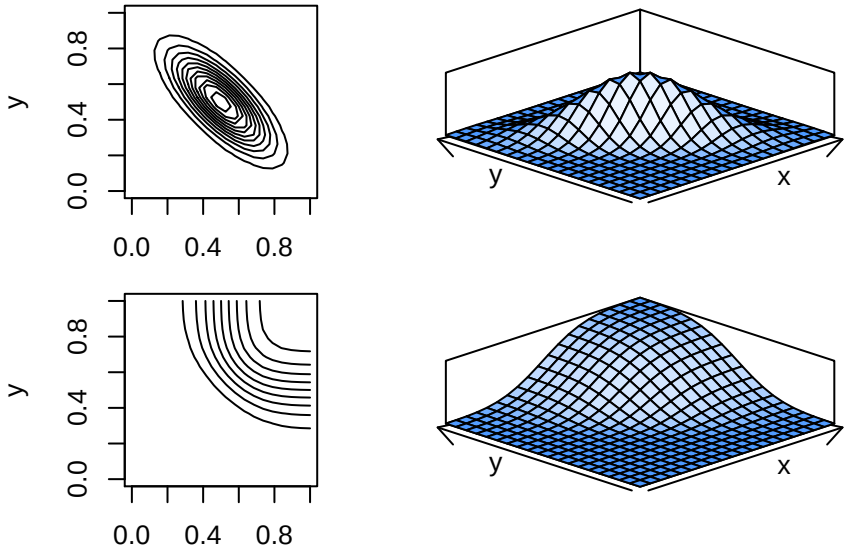
Second, let's consider bivariate distributions with positive correlation/covariance.

```
> f2 = nbpdf (0, 0, 1, 1, 0.75)
> plot (f2, all=TRUE)
```



Third, let's consider bivariate distributions with negative correlation/covariance.

```
> f3 = nbpdf (0, 0, 1, 1, -0.75)
> plot (f3, all=TRUE)
```



The differences between the probability density functions are reasonably obvious. Note that the cumulative distribution function with positive correlation is more angular.

Using Bivariate Cumulative Distributions to Compute Probabilities

Using a univariate distribution we can compute the probability that X is between two values using something like:

$$\mathbb{P}(x_1 < X < x_2) = F(x_2) - F(x_1)$$

Using a bivariate distribution we can compute the probability the X and Y are between two pairs of values using something like:

$$\mathbb{P}(x_1 < X < x_2, y1 < Y < y2) = F(x_1, y_1) + F(x_2, x_2) - F(x_1, y_2) - F(x_2, y_1)$$

We can compute this using something like:

```
> x1 = y1 = -1
> x2 = y2 = 1
> compute.bivariate.probability = function (F, x1, x2, y1, y2)
    F (x1, y1) + F (x2, y2) - F (x1, y2) - F (x2, y1)
> compute.bivariate.probability (F, x1, x2, y1, y2)

[1] 0.4660649
```

Our functions are vectorized so we could do something like:

```
> x1 = y1 = -1:-3
> x2 = y2 = 1:3
> p = compute.bivariate.probability (F, x1, x2, y1, y2)
> cbind (x1, x2, y1, y2, p)

     x1 x2 y1 y2         p
[1,] -1  1 -1  1 0.4660649
[2,] -2  2 -2  2 0.9110697
[3,] -3  3 -3  3 0.9946077
```