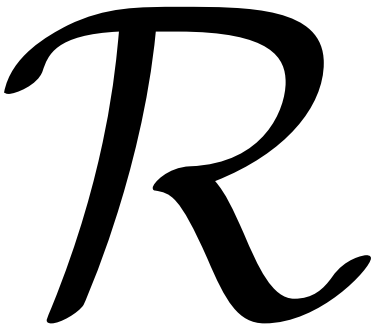
PRZEMYSŁAW BIECEK

Przewodnik po pakiecie



Wydanie drugie, rozszerzone

Oficyna Wydawnicza GiS

Wrocław, 2011

Wnioski, skargi i zażalenia należy kierować do
Przemysław Biecek
e-mail: przemyslaw.biecek@gmail.com
www: http://www.biecek.pl
Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski

Recenzent wydania pierwszego
Dr hab. Jan Mielniczuk
Instytut Podstaw Informatyki PAN

Redakcja i korekta
Karolina Biecek

Projekt okładki i skład
Przemysław Biecek
Skład komputerowy książki wykonano w systemie LaTeX.

Książka ta została przygotowana, aby ułatwić poznawanie i codzienna pracę z programem R. Przyda się ona tym wszystkim, którzy w pracy lub w szkole zajmują się analizą danych. Książka może być wykorzystana, jako pomoc w nauce programu R lub jako encyklopedyczna ściągawka przydatna w codziennej pracy z tym programem. W latach 2010 i 2011 używałem jej z powodzeniem jako główny podręcznik do przedmiotu „Programowanie w R".
Pod adresem http://www.biecek.pl/R/ *czytelnik znajdzie dodatkowe informacje, rozwiązania zadań umieszczonych w tej książce, oraz odnośniki do innych materiałów pomagających w nauce programu R.*

ISBN: 978-83-89020-98-7

Wydanie II rozszerzone, Wrocław 2011
Oficyna Wydawnicza GiS, s.c., http://www.gis.wroc.pl
Druk i oprawa: I-BiS Usługi Komputerowe Wydawnictwo s.c.

Spis treści

Przeczytaj zanim kupisz

Szanowny Czytelniku, trzymasz właśnie w ręku książkę od początku do końca poświęconą pakietowi R. Książka ta powstała po to, by zaprezentować szeroki wachlarz możliwości pakietu R i ułatwić poznanie jego prostych i zaawansowanych aspektów. W sposób systematyczny przedstawia język R, na licznych przykładach opisuje podstawowe funkcje, prezentuje przydatne biblioteki dostępne w tym środowisku, opisuje popularne procedury statystyczne oraz funkcje do tworzenia grafiki.

Pozycja ta zaczęła powstawać w roku 2006, jako materiały pomocnicze dla moich studentów dzielnie poznających tajniki statystyki i analizy danych. Została rozbudowana i uzupełniona, aby mogła z niej skorzystać szersza grupa odbiorców. Starałem się wybrać materiał tak, by tę książkę chciały przeczytać:

- osoby, które chcą poznać pakiet R od podstaw, słyszały że warto i szukają łagodnego wprowadzenia dla zupełnych laików,
- osoby korzystające już z R, znające podstawy i chcące swoją wiedzę usystematyzować, uzupełnić, rozszerzyć, pogłębić,
- osoby pracujące z R na co dzień (eksperci), szukające podręcznej ściągawki (trudno spamiętać np. nazwy wszystkich argumentów graficznych) lub też chcące upewnić się, że o R wiedzą już (prawie) wszystko.

Innymi słowy, mam nadzieję, że każdy znajdzie tu coś dla siebie.

Książka podzielona jest na cztery części. Pierwsza część, to skrótowe przedstawienie możliwości pakietu R. Rozpoczyna się od wprowadzenia dla zupełnych nowicjuszy, ale w miarę upływu stron przedstawiane są kolejne, coraz bardziej zaawansowane informacje o języku oraz pakiecie R. Ta część jest przygotowana z myślą o osobach początkujących i o osobach chcących swoją wiedzę o R uzupełnić. Nie jest zakładana jakiejkolwiek wstępna wiedza o pakiecie R. Zaczynamy od podstaw, ale jestem pewien, że również spore grono zaawansowanych użytkowników znajdzie tutaj coś nowego. Dlatego warto przejrzeć tę część bez względu na stopień zaawansowania.

Kolejne części mają charakter encyklopedyczny i można je czytać w dowolnej kolejności. Część druga „pazuRrry" przedstawia możliwości języka R, o których warto wiedzieć i z których warto korzystać, a które nie znalazły się w innych częściach.

Najsilniejszą stroną R jest potężne wsparcie dla szeroko pojętych analiz statystycznych. W części trzeciej pt. „Wybrane procedury statystyczne" przedstawiono listę funkcji statystycznych wykorzystywanych przy najpopularniejszych procedurach statystycznych wraz z informacją, jak z tych funkcji korzystać i jak interpretować ich wyniki. Pakiet R świetnie nadaje się do tworzenia dobrze wyglądających rysunków, dlatego część czwarta „gRrrafika" poświęcona jest mechanizmom R umożliwiającym tworzenie i modyfikacje dobrze wyglądających wykresów (zarówno prostych jak i bardzo wymyślnych), schematów, grafik itp. Część czwarta kończy się prezentacją funkcji i argumentów graficznych, dzięki którym użytkownik ma pełną kontrolę nad tym co, jak i gdzie jest rysowane.

Pakiet R rozwija się dynamicznie i nieustannie. Ma tak wiele możliwości, że nie sposób wszystkich opisać. Dołożyłem wszelkich starań, by ta pozycja była zrozumiała dla początkujących użytkowników i ciekawa dla użytkowników zaawansowanych. Będę zobowiązany czytelnikom za wszelkie uwagi i komentarze, które pozwolą uczynić tę pozycję czytelniejszą lub ciekawszą, zarówno te dotyczące zawartości jak i te dotyczące formy. Pod adresem `http://www.biecek.pl/R/R.pdf` znajdują się (w postaci elektronicznej) pierwsze 64 strony tej książki. Jest to, mam nadzieję, wystarczający fragment, by przekonać czytelnika, że warto bliżej zapoznać się z programem R. Ten fragment książki może być drukowany i kopiowany na użytek własny. Mam nadzieję, że pomoże on wielu osobom w pierwszym kontakcie z R, a także zachęci do nabycia całej książki w postaci drukowanej.

Książka ta mogła powstać wyłącznie dzięki mniejszej lub większej pomocy bardzo wielu osób, którym serdecznie dziękuję. Szczególnie gorąco dziękuję żonie Karolinie za jej wsparcie, wyrozumiałość, wytrwałość przy wielokrotnym czytaniu kolejnych wersji i moc cennych uwag. Wiele cennych wskazówek, sugestii, propozycji i uwag do kolejnych wersji otrzymałem od prof. dra hab. Jana Mielniczuka, który poświęcił bardzo wiele czasu korygując moje liczne pomyłki, serdecznie mu za to dziękuję. Za cenne uwagi merytoryczne chciałbym podziękować dr Janowi Ćwikowi i dr hab. Pawłowi Mackiewiczowi a również Grzegorzowi Hermanowiczowi i moim studentom, którzy czasem dzielili się uwagami, wątpliwościami oraz pomysłami na zmiany. Za pomoc przy wydawaniu tej książki chcę podziękować prof. dr hab. Jackowi Koronackiemu oraz wydawcom: dr. Marianowi Gewertowi i doc. dr. Zbigniewowi Skoczylasowi, bez których pomocy i zaangażowania książka ta nie powstałaby w postaci papierowej. Korzystając z okazji dziękuję moim wieloletnim współpracownikom dr inż. Adamowi Zagdańskiemu i dr inż. Arturowi Suchwałce za „zarażenie" mnie pakietem R i za wiele wspólnie realizowanych projektów wykonanych w R i nie tylko. Specjalne podziękowania składam również moim przełożonym: prof. dr hab. Teresie Ledwinie i prof. dr hab. Stanisławowi Cebratowi za pozostawienie mi swobody w wyborze zadań do realizacji.

To tyle tytułem wstępu. Życzę owocnej pracy z programem R.

Przemysław Biecek, Wrocław 2008

Przedmowa do wydania drugiego

Minęły już ponad dwa lata od pierwszego wydania „Przewodnika ...". W międzyczasie program R rozwija się w wykładniczym tempie i zapewnił sobie pozycję wyśmienitego narzędzia do analizy danych. Liczba pakietów dostępnych w repozytorium CRAN zwiększyła się w ciągu ostatnich dwóch lat z 700 do 2400. Przybyło wiele funkcjonalności, pewne uległy zmianie, pewne rzeczy można zrobić już znacznie łatwiej. Widząc ilość zmian zacząłem się zastanawiać nad aktualizacją tego podręcznika.

Kolejna zachęta do zmian przyszła od czytelników od których przez te dwa lata otrzymałem wiele listów. Część listów ograniczających się do stwierdzenia „dobra robota" część z sugestiami, co warto zmienić, co warto dodać, co lepiej usunąć. Były też listy z pogróżkami czy ultimatum, albo poprawię wskazany błąd na stronie x w linii y albo zostanę narażony na interwencję profesora Jana Miodka. Gorąco dziękuję za listy i uwagi! Miło jest wiedzieć, że ktoś poświęcił trochę czasu by podzielić się wrażeniami, przesłać kilka pomysłów, wskazać co mu się podoba, a co nie.

Część z propozycji rozszerzenia tej książki dotyczyło rozdziału dotyczącego statystyki i data mining. Zamiast jednak dodać kolejne sto stron do tej książki stwierdziłem, że właściwiej będzie napisać kolejną poświęconą wyłącznie analizie statystycznej danych w programie R z wykorzystaniem modeli liniowych i mieszanych. Prace nad tą pozycją trwają i powinny być zakończone przed końcem tego roku.

Cześć uwag dotyczyło wielkości czcionki. W pierwszym wydaniu użyłem rozmiaru czcionki 10 punktów, dzięki czemu książka była mniejsza i tym samym tańsza. Ponieważ jednak wiele osób narzekało na niewielkie literki, dlatego w tym wydaniu użyłem czcionki o rozmiarze 11 punktów co zwiększyło liczbę stron, ale poprawiło czytelność.

Poza rozmiarem czcionki wprowadziłem następujące zmiany:

- rozbudowałem rozdział poświęcony pisaniu wydajnych skryptów w programie R, rosnące rozmiary zbiorów danych wymuszają liczenie się z czasem obliczeń,
- dodałem rozdział poświęcony tworzeniu własnych pakietów,
- rozbudowałem rozdział poświęcony generowaniu zmiennych losowych,
- dodałem dwa podrozdziały opisujące funkcje graficzne i filozofię tworzenia wykresów w pakietach `lattice` i `ggplot2`,
- sprawdziłem czy w wersji R 2.12.1 (aktualnie najnowsza) działają wszystkie opisane w tej książce funkcje.

Usunąłem też różne drobne usterki, które zostały mi wskazane przez uważnych czytelników. Pewnie też wprowadziłem sporo nowych usterek, za wyśledzenie których będę wdzięczny uważnym czytelnikom.

Na koniec chciałbym szczególnie podziękować kilku osobom, które pośrednio lub bezpośrednio bardzo mi pomogły w pracy nad drugim wydaniem „Przewodnika ...". W pierwszej kolejności dziękuję żonie, Karolinie Biecek, bez której nieustannego wsparcia nie dałbym rady ani przygotować drugiego wydania ani zrealizować wielu innych projektów. Za wiele propozycji usprawnień, modyfikacji, korekt i rozszerzeń pierwszego wydania chciałbym podziękować przyjacielowi i współpracownikowi dr hab. Pawłowi Mackiewiczowi. Bardzo serdecznie chciałbym podziękować dr. Maciejowi Michalewiczowi i Justinowi Lindsleyowi, szefom nzLabs, oddziału badawczego firmy Netezza, za pomoc w finansowaniu konferencji WZUR (Warszawski/Wrocławski Zlot Użytkowników R), pomoc w finansowaniu mojego udziału w konferencjach use!R i umożliwienie mi realizacji ciekawych projektów rozwojowych dotyczących programu R. Dzięki tej współpracy łatwiej mi zrozumieć jakie są oczekiwania dużych firm informatycznych co do modułów analitycznych oraz jak w tych zastosowaniach odnajduje się program R. Gorąco chciałbym podziękować również najlepszym wydawcom pod słońcem, panom dr. Marianowi Gewertowi i doc. dr. Zbigniewowi Skoczylasowi, za pomoc przy edycji i korekcie tak pierwszego jak i drugiego wydania oraz olbrzymi kapitał zaufania bez którego ta książka nigdy by się nie ukazała.

Przemysław Biecek, Warszawa 2011

2.7 Budowanie własnych pakietów

W programie R pakiety służą do grupowania zbiorów danych lub funkcji. Jeżeli opracowaliśmy kilka użytecznych funkcji lub zbiorów danych i chcemy je udostępnić innym osobom, to najlepszym sposobem będzie złożenie ich w pakiet. Taki pakiet możemy wysłać naszym studentom, współpracownikom, klientom, udostępniając im opracowane funkcje. Coraz popularniejsze staje się też dołączanie pakietów do publikacji proponujących nową metodę analizy danych. Dołączając pakiet umożliwiamy łatwy dostęp do opracowanej przez nas metody. Stworzony pakiet możemy umieścić na naszej stronie www, wysłać zainteresowanym osobom mailem lub, jeżeli chcemy by był dostępny szerokiemu gronu, możemy go umieścić w publicznym repozytorium pakietów R, np. repozytorium CRAN lub Bioconductor. Nawet jeżeli opracowaliśmy zbiór funkcji tylko do naszego użytku wciąż wygodnie jest złożyć w pakiet. Korzystanie z pakietów pozwala na łatwe przenoszenie kodu, automatyczne testowanie opracowanych funkcji, łatwe zarządzanie dokumentacją kodu, przenaszalność funkcjonalności pomiędzy różnymi systemami operacyjnymi, komputerami lub wersjami programu R.

Podsumowując: warto tworzyć pakiety!

Aby zbudować własny pakiet musimy wykonać kilka czynności:

- zainstalować dodatkowe oprogramowanie niezbędne do budowy pakietów,
- przygotować odpowiednią strukturę plików i katalogów,
- przetestować i zbudować opracowany pakiet.

Poniżej omówimy każdy z tych kroków. Szczegółowo proces budowy pakietów jest opisany w dokumencie [42]. Jest on obszerniejszy niż informacje z tego rozdziału, zawiera np. opis jak dołączyć do pakietu kompilowane źródła w innych językach, jak dołączyć dodatkową dokumentację, itp. W tym rozdziale omówimy proces tworzenia prostego przykładowego pakietu.

2.7.1 Niezbędne oprogramowanie

Skrypty budujące pakiety R wymagają dodatkowych programów, które nie są zawarte w podstawowej instalacji programu R. Możemy każdy z tych dodatkowych programów zainstalować samodzielnie lub wykorzystać gotowe zestawy zawierające niezbędne oprogramowanie w odpowiedniej wersji. Poniżej opiszemy, gdzie znaleźć taki zestaw. Dalsze postępowanie zależy od tego jakiego systemu operacyjnego używamy.

2.7.1.1 Unix/Linux

W większości dystrybucji niezbędne programy są już zainstalowane.

2.7.1.2 Windows

Zestaw dodatkowego oprogramowania dla systemu Windows można pobrać ze strony `http://www.murdoch-sutherland.com/Rtools/`. Ten zestaw jest nazywany „Rtools" i jest przygotowywany przez Duncana Murdocha dla każdej wersji R począwszy od 1.9 (wcześniej zajmował się tym Brian Ripley). Dla wersji 2.12 ten zestaw zawiera MinGW, Perl i kilka innych dodatków. Po zainstalowaniu wersji „Rtools" odpowiedniej do wersji R na którą chcemy budować pakiety, musimy wykonać jeszcze trzy kroki opisane w `http://www.murdoch-sutherland.com/Rtools/Rtools.txt`.

- Zainstalować dystrybucję LaTeXa. Jest ona niezbędna, by budować dokumentację w formacie pdf. Jedną z popularniejszych dystrybucji dla systemu Windows jest MikTeX, zobacz `http://www.miktex.org`.
- Zainstalować wsparcie do generowania plików pomocy w formacie HTML dla systemu Windows. Odpowiedni program można znaleźć tutaj `http://msdn.microsoft.com/en-us/library/ms669985.aspx`. Jest on niezbędny do generowania dokumentacji w formacie `.chm`.
- Zainstalować instalator Inno (`http://www.innosetup.com`). Ten krok potrzebny jest tylko jeżeli chcemy też przebudowywać cały program R. Do budowania pakietów instalator Inno nie jest potrzebny.

Po zainstalowaniu tych narzędzi, należy dodać odpowiednie ścieżki do zmiennych systemowych. Przyjmując, że `Rtools` zainstalowaliśmy do katalogu `c:\Rtools`, możemy to zrobić poleceniem:
`PATH=c:\Rtools\bin;c:\Rtools\perl\bin;c:\Rtools\MinGW\bin`.

2.7.1.3 MacOS

Aby budować nowe pakiety w systemie MacOS X 10.4 (i wyżej) muszą być zainstalowane następujące programy:

- Xcode Developer Tools w wersji 3.1 lub nowszej,
- Kompilator GNU dla Fortrana,

oba można pobrać ze strony `http://r.research.att.com/tools/`.

W Internecie można znaleźć wiele wskazówek dotyczących instalacji i rozwijania programu R w systemach Windows i Linux. Dla użytkowników MacOSa takich przewodników jest mniej, ale wiele użytecznych informacji można znaleźć pod adresami `http://r.research.att.com/` oraz `http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html`.

2.7.2 Przygotowanie pakietu

Pakiety dla programu R mogą zawierać funkcje napisane w języku R, funkcje napisane w innych językach, np. C++, fortran itp, dokumentacje w formacie `Rd`, `pdf` lub `tex`, grafikę, dane i inne komponenty. Aby można było

z nich korzystać należy te komponenty odpowiednie opisać i umieścić je w odpowiednich katalogach.

Techniczny i szczegółowy opis możliwości i ograniczeń związanych z tworzeniem pakietów można znaleźć w dokumencie [42]. Poniżej przedstawimy przykład jak krok po kroku stworzyć, zbudować i zainstalować nowy pakiet. Na potrzebę przykładu stworzymy pakiet `PBImisc`, w którym umieścimy zbiór danych oraz funkcję rysującą histogram.

2.7.2.1 Przygotowanie funkcji i zbiorów danych

W pierwszej wersji pakietu umieścimy jedną funkcję i jeden zbiór danych. Poniższy skrypt definiuje nową funkcję `hist2()`, która rysuje histogram i dorysowuje do niego jądrowy estymator gęstości. Ten skrypt definiuje również wektor o nazwie `petlen` zawierający 150 liczb losowych.

```
# zbior danych, 150 liczb losowych
petlen <- c(rnorm(100), rnorm(50,4))
# funkcja hist2()
hist2 <- function(x, breakes="Sturges", bw="nrd0", name="") {
 hist(x, probability=T, breaks=breakes, main=name)
 dtmp <- density(x, bw=bw)
 lines(dtmp$x, dtmp$y,lwd=3, col="red3")
 rug(x, col="red")
}
```

2.7.2.2 Przygotowanie struktury katalogów

Dane w pakiecie rozmieszczone są w określonej strukturze katalogów. Zaawansowani użytkownicy niezbędną strukturę katalogów mogą stworzyć ręcznie. Tutaj jednak, na potrzeby tego przykładu, wykorzystamy do tego celu funkcję `package.skeleton(utils)`. Pierwszym argumentem tej funkcji jest lista obiektów, które chcemy umieścić w pakiecie. Mogą to być funkcje lub zbiory danych lub dowolne inne obiekty R. Drugi argument to nazwa pakietu, który chcemy utworzyć.

W poniższym przykładzie utworzymy pakiet `PBImisc` zawierający funkcję `hist2` i wektor `petlen`.

```
> package.skeleton(list=c("hist2","petlen"), name="PBImisc")
Creating directories ...
Creating DESCRIPTION ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in './PBImisc/Read-and-delete-me'.
```

Jako efekt uboczny działania funkcji `package.skeleton()`, w katalogu `R/R-2.11.1/bin/` został utworzony podkatalog `PBImisc` z automatycznie wygenerowaną minimalną strukturą podkatalogów. Ta minimalna struktura składa się z trzech podkatalogów:

- podkatalog `data` zawiera zbiory danych zapisane w binarnym formacie `rda`. Każdy plik odpowiada jednemu zbiorowi danych. Po zainstalowaniu i włączeniu pakietu `PBImisc` dane te automatycznie będą dostępne w przestrzeni nazw pakietu.

  Pliki binarne zapisane w formacie `rda` można odczytywać funkcją `load(base)`. Do formatu `rda` można zapisywać funkcją `save(base)`. Więcej informacji o tych funkcjach jest w rozdziale 1.6.4.

- podkatalog `R` zawiera definicje funkcji zapisane w postaci tekstowej. Podczas budowania pakietu wszystkie pliki z rozszerzeniem `.R` z podkatalog `R` będą wykonane inicjując przestrzeń nazw. W każdym pliku może być dowolna liczba definicji funkcji, jednak dla czytelności zalecane jest by jeden plik odpowiadał jednej funkcji.

- podkatalog `man` zawiera opisy zbiorów danych i funkcji zapisane w formacie `Rd`. Format `Rd` to tekstowy format opisu obiektu. W trakcie budowania pakietu będzie on automatycznie przekształcony do formatu HTML, TEX oraz `pdf`.

Wszystkie obiekty, które udostępniamy w pakiecie, powinny mieć przygotowane pliki pomocy. Plik `Rd` składa się z kilku sekcji. Nazwy sekcji różnią się w zależności od tego, czy opisywany jest zbiór danych czy funkcja.

Plik pomocy dla zbioru danych

Podstawowe sekcje dla plików pomocy opisujących zbiory danych to:

- sekcja `\name` określa nazwę danego pliku pomocy,
- każda sekcja `\alias` wskazuje alias tego pliku pomocy, wskazany plik pomocy zostanie otwarty, również po wpisaniu komendy `?_alias_`. Można umieścić więcej niż jedną sekcje alias, dlatego mówimy tutaj o opisie pliku pomocy, a nie opisie zbioru danych. Jeden plik pomocy może opisywać kilka zbiorów danych lub kilka tematów pomocy,
- sekcja `\title` zawiera tytuł opisu, najczęściej jest krótka i zawiera jedno lub dwa zdania,
- sekcja `\usage` pokazuje, jak wczytać dany zbiór danych,
- sekcja `\format` przedstawia strukturę zbioru danych,
- sekcja `\details` przedstawia szczegółowy opis zbioru danych, może być dowolnie długa,

- sekcja `\source` zawiera listę publikacji lub innych referencji, w których można znaleźć więcej informacji o zbiorze danych,
- sekcja `\examples` prezentuje przykłady użycia zbioru danych.

W naszym przykładzie plik `PBImisc/man/petlen.Rd` został wygenerowany automatycznie. Musimy samodzielnie wypełnić odpowiednie pola. Poniżej znajduje się przykładowa zawartość uzupełnionego pliku pomocy.

```
\name{petlen}
\alias{petlen}
\docType{data}
\title{
  Przykladowy zbior danych
}
\description{
  Przykladowy zbior danych zawierający 150 liczb losowych z
      dwumodalnego rozkladu
}
\usage{data(petlen)}
\format{
  Wektor 150 liczb
  num [1:150] 1.46 1.29 1.31 1.56 1.36 ...
}
\details{
  Wektor zostal wygenerowany poleceniem
  petlen <- c(rnorm(100), rnorm(50,4))
}
\source{
  Brak zrodel.
}
\examples{
  data(petlen)
  hist2(petlen)
}
\keyword{petlen}
```

Plik pomocy dla funkcji

Podstawowe sekcje dla plików pomocy opisujących funkcje to:

- sekcja `\name` opisuje nazwę danego pliku pomocy,
- każda sekcja `\alias` (można umieścić więcej niż jedną) wskazuje alias tego pliku pomocy, wskazany plik pomocy zostanie otwarty, również po wpisaniu komendy `?_alias_`,
- sekcja `\title` zawiera tytuł opisu, najczęściej jest to jedno lub dwa zdania,

- sekcja `\description` zawiera rozszerzony opis działania funkcji,
- sekcja `\usage` przedstawia sygnaturę funkcji wraz z listą argumentów i ich wartościami domyślnymi,
- sekcja `\arguments` zawiera opisy dla każdego z argumentów,
- sekcja `\details` przedstawia szczegółowy opis działania funkcji, może być dowolnie długa,
- sekcja `\value` opisuje jakiego typu obiekt jest zwracany przez funkcję, oraz jakie pola zawiera ten obiekt,
- sekcja `\seealso` zawiera odwołania do innych funkcji o podobnym działaniu,
- sekcja `\examples` prezentuje przykłady użycia danej funkcji.

W naszym przykładzie plik `PBImisc/man/hist2.Rd` został wygenerowany automatycznie, należy go uzupełnić. Poniżej znajduje się przykładowa treść tego pliku po uzupełnieniu.

```
\name{hist2}
\alias{hist2}
\title{
  Rozszerzony histogram
}
\description{
  Funkcja hist2() rysuje histogram i dorysowuje do niego jądrowy
      estymator gęstości, razem ze znacznikami punktów
      odpowiadających kolejnym obserwacjom.
}
\usage{
  hist2(x, breakes = "Sturges", bw = "nrd0", name = "")
}
\arguments{
  \item{x}{wektor liczb}
  \item{breakes}{liczba przedziałów, ten argument zostanie
      przekazany do funkcji hist()}
  \item{bw}{szerokość okna, ten argument będzie przekazany do
      funkcji density()}
  \item{name}{tytuł wykresu}
}
\details{
  Funkcja hist2() wywołuje kolejno funkcje: hist() do narysowania
      histogramu, density() do wyznaczenia   jądrowego estymatora
      gęstości i rug() do dorysowania znaczników obserwacji.
}
\value{Nie zwraca wyniku}
\seealso{
Funkcja rysujaca histogram \code{\link{hist}}
}
```

```
\examples{
  data(petlen)
  hist2(petlen)
}
\keyword{histogram}
```

Dodatkowo w podkatalogu `man` można umieścić plik pomocy dotyczący całego pakietu. W naszym przykładzie jest to plik `PBImisc-package.Rd`.

W katalogu opisującym pakiet muszą się znaleźć pliki tekstowe o nazwach `DESCRIPTION` i `NAMESPACE`. Plik `DESCRIPTION` utworzony będzie automatycznie przez funkcję `package.skeleton()`, plik `NAMESPACE` będziemy musieli dodać ręcznie. Poniżej krótko omówimy zawartość obu plików.

Opis pakietu, plik DESCRIPTION

Plik `DESCRIPTION` zawiera podstawowe informacje o pakiecie, jego wersji, autorze pakietu, zależnościach od innych pakietów. Jeżeli zdecydujemy się umieścić pakiet na serwerze CRAN, to informacje z pliku `DESCRIPTION` będą umieszczone na stronie www z której będzie można pobrać pakiet.

Przykładowa zawartość pliku `DESCRIPTION`.

```
Package: PBImisc
Type: Package
Title: Zbior pomocniczych funkcji dla PBI
Version: 1.0
Date: 2010-08-02
Author: Przemyslaw Biecek
Maintainer: PBI <przemyslaw.biecek@gmail.com>
Description: Prywatny pakiet z pomocniczymi funkcjami
License: GPL 2
LazyLoad: yes
```

Opis zawartości pakietu, plik NAMESPACE

Plik `NAMESPACE` zawiera informacje o tym, które funkcje mają zostać udostępnione dla użytkownika po włączeniu pakietu. Funkcje wymienione w tym pliku będą widoczne w głównej przestrzeni nazw. Pozostałe funkcje zdefiniowane w plikach z podkatalogu `R` będą funkcjami prywatnymi, dostępnymi jedynie w przestrzeni nazw pakietu. Z funkcji prywatnych korzystać mogą inne funkcje z tego samego pakietu.

Poniższa zawartość pliku `NAMESPACE` wskazuje, że jedyną udostępnianą funkcją ma być funkcja `hist2()`.

```
export(
     hist2
)
```

Jeżeli nie chcemy wymieniać z nazwy wszystkich funkcji to możemy użyć wyrażeń regularnych do wskazywania, które funkcje mają być udostępnione. Np. poniższy wpis w pliku `NAMESPACE` uczyni publicznymi wszystkie obiekty zaczynające swoje nazwy od przedrostka `hist`.

```
exportPattern("^hist*")
```

Jeżeli funkcja nie będzie publiczna, czyli nie zostanie wymieniona w pliku `NAMESPACE`, to wciąż można się do niej odwołać wykorzystując operator zasięgu `:::`. Np. `PBImisc:::hist2` odwoła się do funkcji `hist2` z pakietu `PBImisc`, bez względu na to, czy jest ona publiczna czy nie.

2.7.3 Weryfikacja, budowanie i instalacja pakietu

Po przygotowaniu pakietu następuje zazwyczaj sekwencja czynności, takich jak budowanie, testowanie i instalacja pakietu. Poniżej omówiony jest każdy z tych kroków.

Budowanie pakietu

Zakładamy, że wszystkie narzędzia opisane w rozdziale 2.7 zostały zainstalowane. Przyjmijmy też, że pakiet który chcemy zbudować nazywa się `PBImisc` i znajduje się w katalogu `R/bin`. Otwieramy konsolę i przechodzimy do katalogu `R/bin` w którym znajduje się plik `R.exe`. Jeżeli wywołamy polecenie `R.exe` bez dodatkowych argumentów, to uruchomiony zostanie interpreter R. Jeżeli jednak pierwszym parametrem polecenia `R` będzie `CMD`, to uruchomiony będzie jeden z predefiniowanych skryptów perlowych, znajdujących się w katalogu `R/bin`. Jednym z takich skryptów jest `build`, który służy do budowania pakietów.

W poniższym przykładzie uruchamiamy skrypt `build` oraz budujemy pakiet `PBImisc`.

```
C:\_Soft_\R\R-2.11.1\bin>R CMD build PBImisc

* checking for file 'PBImisc/DESCRIPTION' ... OK
* preparing 'PBImisc':
* checking DESCRIPTION meta-information ... OK
* removing junk files
* checking for LF line-endings in source and make files
* checking for empty or unneeded directories
* building 'PBImisc_1.0.tar.gz'
```

Wynikiem wykonania polecenia `R CMD build PBImisc` jest utworzenie pliku `PBImisc_1.0.tar.gz`, który zawiera źródła pakietu. Ten plik może teraz być użyty do instalacji pakietu.

Testowanie pakietu

Do testowania pakietu można wykorzystać skrypt `check`. Skrypt ten sprawdzi, czy podany pakiet można zainstalować, czy dokumentacja generuje się bez błędów, czy przykłady dla funkcji udało się wykonać bez błędów, czy nie ma kolizji oznaczeń z innymi pakietami itp.

Jest to bardzo użyteczna funkcjonalność. Jeżeli dla naszych funkcji przygotujemy zestawy przykładów, które mogą służyć jako testy, to skrypt `check` umożliwi automatyczne testowanie funkcji z naszego pakietu.

Wynik procesu testowania przykładowego pakietu `PBImisc` przedstawiony jest poniżej.

```
C:\_Soft_\R\R-2.11.1\bin>R CMD check PBImisc
* checking for working pdflatex ... OK
* using log directory 'C:/_Soft_/R/R-2.11.1/bin/PBImisc.Rcheck'
* using R version 2.11.1 (2009-12-14)
* using session charset: CP1250
* checking for file 'PBImisc/DESCRIPTION' ... OK
* checking extension type ... Package
* this is package 'PBImisc' version '1.0'
* checking package name space information ... OK
* checking package dependencies ... OK
* checking if this is a source package ... OK
* checking for .dll and .exe files ... OK
* checking whether package 'PBImisc' can be installed ... OK
* checking package directory ... OK
* checking for portable file names ... OK
* checking DESCRIPTION meta-information ... OK
* checking top-level files ... OK
* checking index information ... OK
* checking package subdirectories ... OK
* checking R files for non-ASCII characters ... OK
* checking R files for syntax errors ... OK
* checking whether the package can be loaded ... OK
* checking whether the package can be loaded with
      stated dependencies ... OK
* checking whether the name space can be loaded with
       stated dependencies ... OK
* checking for unstated dependencies in R code ... OK
* checking S3 generic/method consistency ... OK
* checking replacement functions ... OK
* checking foreign function calls ... OK
* checking R code for possible problems ... OK
* checking Rd files ... OK
* checking Rd metadata ... OK
* checking Rd cross-references ... OK
* checking for missing documentation entries ... OK
```

Angielskie słowo copula, używane w kontekście generowania wielowymiarowych zmiennych zależnych, najczęściej tłumaczone jest niepoprawnie na polskie słowo kopuła. Odpowiedniejszym tłumaczeniem jest polskie słowo pochodzenia łacińskiego kopula oznaczające łącznik w znaczeniu formy czasownika być łączącej podmiot i orzeczenie. Możemy też używać określenia funkcja łącząca. Warto dodać, że wikipedia dopuszcza nawet pisownię copula traktując to słowo jako już polskie, słownik PWN jest bardziej konserwatywny.

W programie R nie musimy do generacji zmiennych z rozkładu normalnego wielowymiarowego ręcznie wykonywać dekompozycji macierzy Σ, wystarczy użyć funkcji `rmvnorm(mvtnorm)`. Pozwala ona wskazać metodę faktoryzacji argumentem `method=c("eigen", "svd", "chol")`. Domyślnie wykorzystywana jest dekompozycja na wartości spektralne (na wypadek gdyby macierz Σ była niepełnego rzędu), ale jeżeli zależy nam na szybkości i mamy pewność że macierz Σ jest dodatnio określona to lepiej użyć pierwiastka Choleskiego.

3.2.3.3 Kopule/funkcje łączące

Powyżej opisaliśmy jak generować zależne zmienne losowe o łącznym rozkładzie normalnym. W wielu zastosowaniach przydatna jest możliwość generowania zmiennych o zadanej strukturze zależności i o dowolnych rozkładach brzegowych. Do tego celu można wykorzystać kopule, czyli funkcje łączące. Za tym narzędziem stoi bardzo ciekawa teoria, jak również wiele zastosowań, szczególnie w finansach i ubezpieczeniach. Poniżej przedstawimy kilka przykładów użycia kopuli do generowania zależnych zmiennych.

Kopula to wielowymiarowy rozkład określony na kostce $[0,1]^n$, taki, że rozkłady brzegowe dla każdej współrzędnej są jednostajne na odcinku $[0,1]$. Poniżej przez $C(u) : [0,1]^n \to [0,1]$ będziemy oznaczać dystrybuantę tego rozkładu. Poniższe twierdzenie wyjaśnia dlaczego kopule są ciekawe.

Twierdzenie Sklara

Dla zadanego rozkładu H określonego na p wymiarowej przestrzeni i odpowiadających mu rozkładów brzegowych $F_1, ..., F_p$ istnieje kopula C, taka że $C(F_1, ..., F_p) = H$. Oznacza to, że każdy rodzaj zależności można opisać pewną kopulą.

Rozważmy przypadek dwuwymiarowy. Dla zmiennej dwuwymiarowej o rozkładzie H o brzegowych rozkładach F_1, F_2 istnieje kopula C taka, że

$$H(x,y) = C\left(F_1(x), F_2(y)\right).$$

Co więcej, jeżeli brzegowe dystrybuanty są ciągłe, to C jest wyznaczona jednoznacznie.

Jeżeli chcemy generować obserwacje z rozkładu H, to wystarczy, że potrafimy generować obserwacje z kopuli C i znamy rozkłady F_1, F_2. Wszystkich możliwych kopuli jest nieskończenie wiele, ale w zastosowaniach najczęściej pojawiają się następujące klasy kopul.

- Kopula Gaussowska. W przypadku dwuwymiarowym kopuła Gaussowska wyrażona jest następującym wzorem

  $$C_\rho(u,v) = \Phi_\rho\left(\Phi^{-1}(u), \Phi^{-1}(v)\right),$$

  gdzie ρ jest parametrem. Ta kopula odpowiada zależności pomiędzy zmiennymi o łącznym rozkładzie normalnym z korelacją ρ. Podobnie definiuje się kopule Gausowskie dla wyższych wymiarów.

- Kopula Archimedesowska. Dla p wymiarów kopuła Archimedesowska wyraża się wzorem

$$C(x_1, x_2, ..., x_p) = \Psi^{-1}\left(\sum_{i=1}^{n} \Psi\left(F_i\left(x_i\right)\right)\right),$$

gdzie Ψ to tzw. funkcja generująca, spełniająca cztery warunki: (1) $\Psi(1) = 0$, (2) $\lim_{x\to 0} \Psi(x) = \infty$, (3) $\Psi'(x) < 0$ i (4) $\Psi''(x) > 0$.

Wybierając odpowiednie funkcje generujące, otrzymujemy następujące podklasy kopuli.

  - Kopula produktowa, w tym przypadku funkcja generująca zadana jest wzorem

$$\begin{aligned} \Psi(x) &= -\log(x), \\ H(x,y) &= F_1(x)F_2(y). \end{aligned}$$

  - Kopula Claytona, w tym przypadku funkcja generująca zadana jest wzorem

$$\begin{aligned} \Psi(x) &= x^{\theta} - 1, \\ H(x,y) &= \left(F_1(x)^{\theta} + F_2(y)^{\theta} - 1\right)^{1/\theta}. \end{aligned}$$

  - Kopula Gumbela, w tym przypadku funkcja generująca zadana jest wzorem

$$\Psi(x) = \left(-\log(x)\right)^{\alpha}.$$

  - Kopula Franka, w tym przypadku funkcja generująca zadana jest wzorem

$$\begin{aligned} \Psi(x) &= \log\frac{\exp(\alpha x) - 1}{\exp(\alpha) - 1}, \\ H(x,y) &= F_1(x)F_2(y). \end{aligned}$$

W pakiecie `copula` do operacji na kopulach służą funkcje:

- `dcopula(copula, u)`, wylicza gęstość kopuli `copula` w punkcie `u`,
- `pcopula(copula, u)`, wylicza dystrybuantę kopuli w punkcie `u`,
- `rcopula(copula, n)`, generuje `n` wartości z kopuli `copula`.

Pierwszym argumentem tych funkcji jest obiekt klasy `copula`, opisujący wielowymiarowy rozkład na kostce jednostkowej. Taki obiekt możemy zainicjować używając jednej z funkcji wymienionych w tabeli 3.3. Gęstości i przykładowe próby wylosowane z wybranych kopuli przedstawione są na rysunkach 3.17, 3.18, 3.19, 3.20 i 3.21.

Zobaczmy jak w programie R zainicjować obiekt typu `copula` i wygenerować wektor zmiennych o zadanej strukturze zależności.

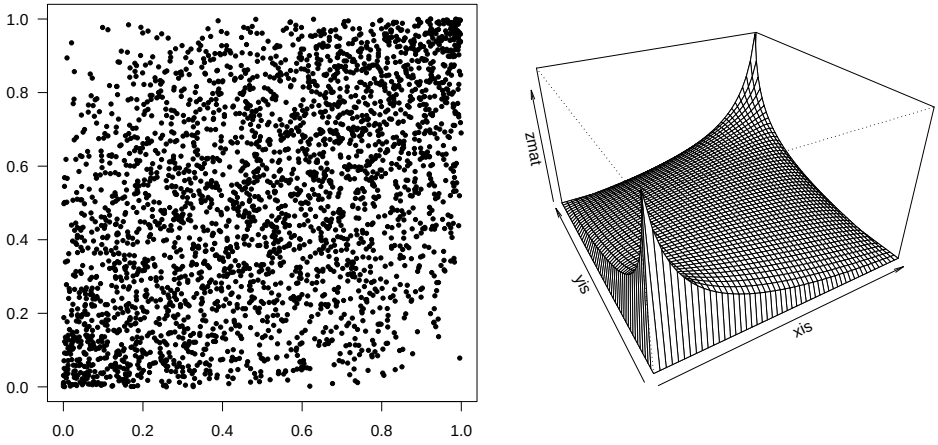
```
> library(copula)
> # tworzymy kopule Gaussowską dla korelacji 0.5
> (norm.cop <- normalCopula(0.5))
Normal copula family
Dimension:  2
Parameters:   rho.1  =  0.5
> # wewnątrz obiektu przechowywane są informacje o strukturze
> str(norm.cop)
Formal class 'normalCopula' [package "copula"] with 8 slots
  ..@ dispstr     : chr "ex"
  ..@ dimension   : num 2
  ..@ parameters  : num 0.5
  ..@ param.names : chr "rho.1"
  ..@ message     : chr "Normal copula family"
> # używając funkcji rcopula generujemy 1000 obserwacji
> x <- rcopula(norm.cop, 1000)
> head(x)
[1,] 0.3231129 0.09705419
[2,] 0.9842054 0.73810795
[3,] 0.1281278 0.16273136
```

Na poniższym przykładzie wygenerujemy 40 obserwacji z rozkładu, którego rozkładami brzegowymi są rozkłady wykładnicze o parametrze $\lambda = 2$, a struktura zależności opisana jest kopulą Claytona.

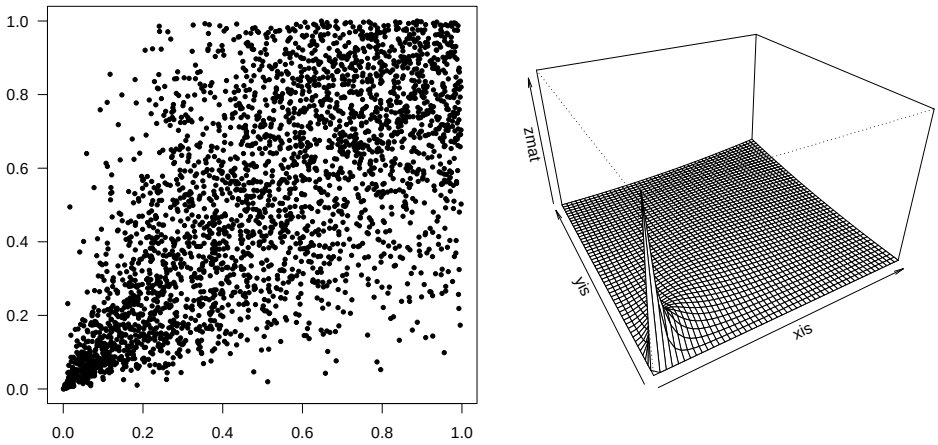
```
> N = 40
> # definiujemy kopulę Claytona
> clayton.cop <- claytonCopula(1, dim = 2)
> # generujemy 40 obserwacji z zadanej kopuli
> x = rcopula(clayton.cop, N)
> # nakładamy na wygenerowane obserwacje odwrotne dystrybuanty, by
    otrzymać zmienne o żadnych rozkładach brzegowych, poniżej
    nałożono odwrotną dla rozkładu wykładniczego z lambda=2
> y = cbind(qexp(x[,1],2),
+           qexp(x[,2],2))
> head(y)
          [,1]       [,2]
[1,] 0.14719249 0.09109257
[2,] 0.37359682 0.36651862
[3,] 0.57194136 2.17383084
[4,] 0.33073801 0.16533854
[5,] 0.44271815 1.64094375
[6,] 0.07162174 0.45268937
```

Kopule stosowane są w wielu zagadnieniach. Poniżej przedstawimy przykład badania mocy dwóch testów korelacji, gdy dwuwymiarowa zmienna ma brzegowe rozkłady wykładnicze i zależność opisaną przez kopulę Claytona.

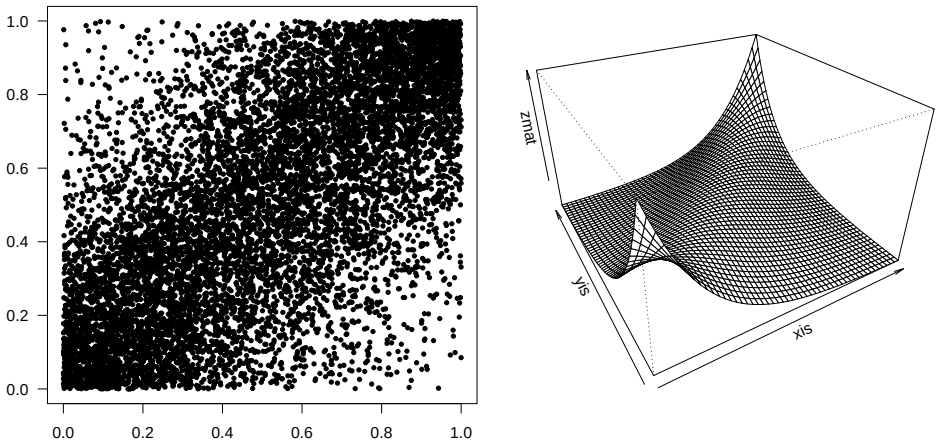
Porównajmy test dla współczynnika korelacji Pearsona i Spearmana. Pierwszy zakłada normalny rozkład obserwacji, zobaczymy jak niespełnienie tego założenia wpływa na moc.



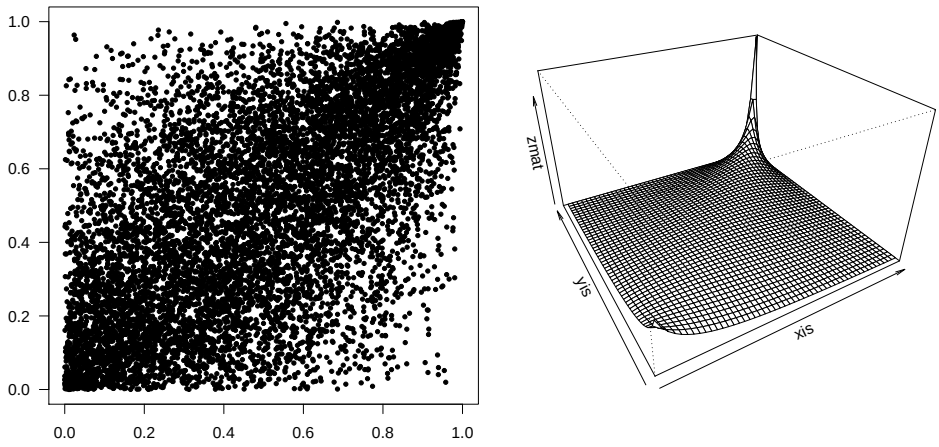
Rysunek 3.17: Kopula Gaussowska $\rho = 0.5$. Przykładowa 200 elementowa próba przedstawiona jest po lewej stronie, po prawej stronie przedstawiana jest gęstość.



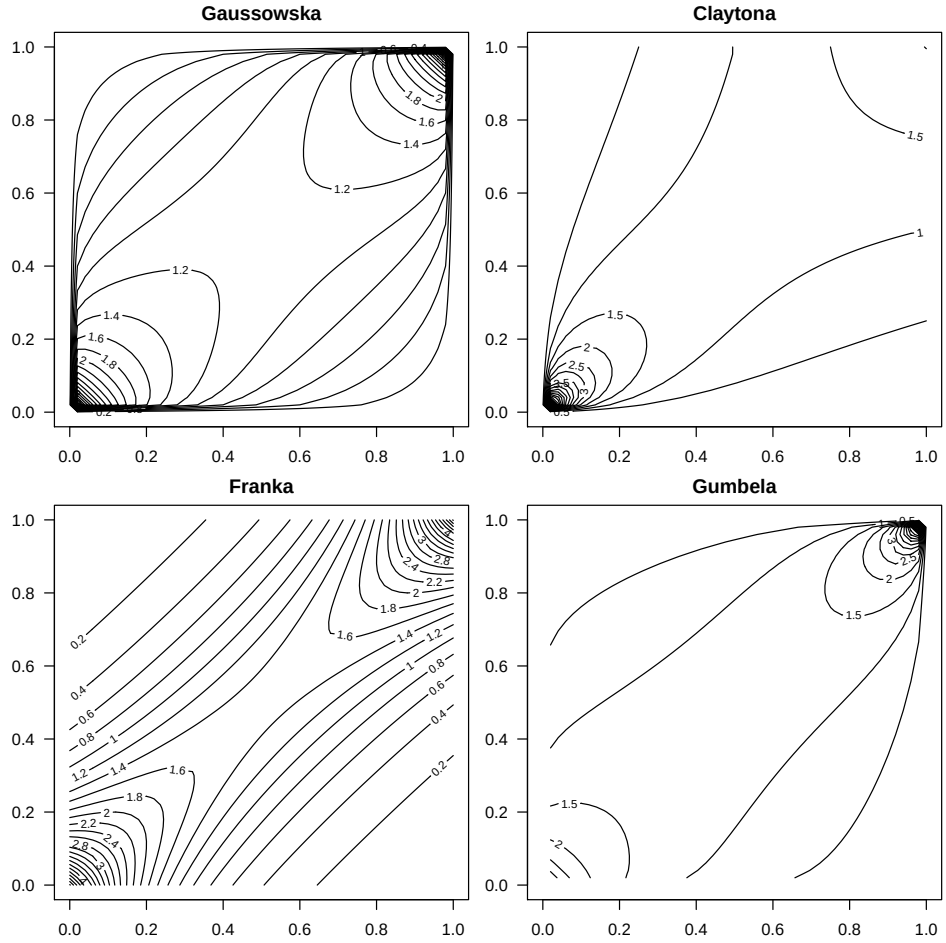
Rysunek 3.18: Kopula Claytona z parametrem 2.



Rysunek 3.19: Kopula Franka z parametrem 5.



Rysunek 3.20: Kopula Gumbela z parametrem 2. Struktura zależności jest niesymetryczna ale inna niż w przypadku kopuli Claytona.



Rysunek 3.21: Wykresy konturowe dla gęstości różnych kopuli.

Tabela 3.3: Funkcje tworzące kopule poszczególnych klas.

`normalCopula(param, dim=2, dispstr="ex")`	Funkcja tworząca kopule Gaussowską o parametrze `param`. Argument `dim` określa wymiar kopuli, argument `dispstr` określa typ zależności dla macierzy kowariancji, wartość `"ex"` oznacza macierz kowariancji permutowalną (ang. *exchangeable*), `"ar1"` macierz kowariancji typu AR(1), a więc na kolejnych diagonalach znajdują się wartości `param^k`, `"un"` dowolną macierz kowariancji (ang. *unstructured*).
`archmCopula(family, param, dim=2, ...)`	Funkcja pozwalająca na tworzenie kopuli z zadanej klasy.
`claytonCopula(param,dim)`	Funkcja tworząca kopule Claytona.
`frankCopula(param,dim)`	Funkcja tworząca kopule Franka.
`gumbelCopula(param,dim)`	Funkcja tworząca kopule Gumbela.

```
> # definiujemy kopule
>  N = 40
>  clayton.cop <- claytonCopula(1, dim = 2)
>  pv = matrix(0, 2, 1000)
>  rownames(pv) = c("Pearson", "Spearman")

> # moc testu ocenimy na podstawie 1000 powtórzeń
>  for (i in 1:1000) {
+       x = rcopula(clayton.cop, N)
+       y = cbind(qexp(x[,1],2), qexp(x[,2],2))
+       # aplikujemy test Spearmana i Pearsona
+       pv[1, i]=cor.test(y[,1],y[,2],method="pearson")$p.value
+       pv[2, i]=cor.test(y[,1],y[,2],method="spearman")$p.value
+  }
> # Policzymy moc testów na poziomie istotności 0.05
>  rowMeans(pv<0.05)
 Pearson Spearman
   0.479    0.864
> # test Spearmana ma tutaj prawie dwukrotnie wyższą moc!
```

W zagadnieniach praktycznych, często nie wiemy z jakiej rodziny wybrać rozkłady brzegowe. W takich sytuacjach dobrym pomysłem jest użycie empirycznej dystrybuanty. Funkcję odwrotną do dystrybuanty empirycznej (która nie jest monotoniczna, ale to tzw. szczegół techniczny) jest funkcja `quantile(stats)`.

3.2.4 Estymacja parametrów rozkładu

Do oceny parametrów metodą największej funkcji wiarogodności w określonej rodzinie rozkładów służy funkcja `fitdistr(MASS)`. Estymowane mogą być parametry dla szerokiej klasy rozkładów, praktycznie wszystkich wymienionych w tabeli 3.2. Pierwszym argumentem funkcji `fitdistr()` jest

`yaxp`	Odpowiednik argumentu `xaxp` dla osi y.
`yaxs`	Odpowiednik argumentu `xaxs` dla osi y.
`yaxt`	Odpowiednik argumentu `xaxt` dla osi y.
`ylog`*	Odpowiednik argumentu `xlog` dla osi y.
`ylim`	Odpowiednik argumentu `xlim` dla osi y.

4.3 Pakiet lattice

W tym rozdziale przedstawimy wprowadzenie do pakietu `lattice`. Z uwagi na ograniczenia objętościowe nie możemy przedstawić wszystkich funkcji, parametrów i ustawień z tą samą dokładnością jak dla pakietu `graphics`. To wprowadzenie przygotowaliśmy by uważny Czytelnik potrafił świadomie korzystać z podstawowych funkcji pakietu `lattice` oraz by zdobyta wiedza umożliwiła łatwe poznanie nowych funkcji pakietu `lattice`. Więcej szczegółowych informacji dotyczących pakietu `lattice` znaleźć można w książce [38] oraz [33].

Pakiet `latiice` nadaje się świetnie do wizualizacji zależności regresyjnych pomiędzy zmiennymi w całej populacji i podpopulajach. Pakiet ten jest często używany do wizualizacji danych w analizie modeli liniowych. W porównaniu z pakietem `graphics` zaletą pakietu `lattice` jest jednolity i wygodny w użyciu interfejs. Ponieważ pakiety `graphics` i `lattice` bazują na różnych systemach graficznych nie można łatwo łączyć funkcji z obu systemów.

4.3.1 Wprowadzenie

W pakiecie `lattice` wygląd wykresu zależy głównie od trzech elementów.

- Użytej funkcji graficznej, określającej rodzaj/szablon wykresu. Inne funkcje służą do rysowania histogramu, wykresu rozrzutu, wykresu pudełkowego, czy innego szablonu prezentacji danych. W pakiecie `lattice` dostępnych jest piętnaście funkcji graficznych odpowiadających różnym szablonom wykresów. Kompletna lista szablonów/funkcji graficznych przedstawiona jest w tabeli 4.5. W kolejnych sekcjach przedstawimy przykłady dla wybranych.

- Formuły, opisującej które zmienne i w jakiej roli mają wystąpić na wykresie. Formuła jest pierwszym argumentem każdej z funkcji graficznych w pakiecie `lattice`. Formuła składa się z trzech części, lewej strony formuły, prawej strony formuły i warunkowania. Na wykresie przedstawiane są zmienne występujące w formule. Zmienne te powinny albo być widoczne w lokalnej przestrzeni nazw albo odpowiadać nazwom kolumn ramki danych wskazanej przez argument `data`. Więcej o semantyce formuł znaleźć można w sekcji 4.3.3.

- Typy zmiennych, określające charakter zmiennych. W zależności od tego czy zmienna jest zmienną ilościową, czy czynnikową, czy reprezentuje czas inaczej będzie przedstawiona na wykresie.

Przyjrzyjmy się przykładowemu wywołaniu funkcji `xyplot()` z pakietu `lattice`.

```
# w pakiecie PBImisc znajdują się wykorzystywane tutaj dane
library(PBImisc)
# wykres rozrzutu tworzy się z użyciem funkcji xyplot()
xyplot(MDRD12~MDRD7|discrepancy.DR, kidney)
```

Rodzaj wykresu określony jest przez nazwę funkcji. W tym przykładzie wywołana jest funkcja `xyplot()`, która tworzy wykresy rozrzutu. Pierwszym argumentem jest formuła, wskazująca, która zmienna ma być przedstawiona na osi X, a która na osi Y. W tym wykresie na osi X jest zmienna MDRD7 a na osi Y jest zmienna MDRD12, zmienną warunkującą jest `discrepancy.DR`. Typ zmiennej zaważy na sposobie jej przedstawienia. W tym przykładzie obie zmienne są ilościowe zostaną przedstawione za pomocą klasycznego wykresu rozrzutu. Ponieważ wykres ten jest warunkowany zmienną czynnikową o trzech poziomach, zatem narysowane będą trzy panele, każdy panel przedstawiać będzie zależności dla pacjentów o innej liczbie niezgodności w antygenach DR.

Dla funkcji z pakietu `lattice` możemy wskazać dodatkowe argumenty i tym samym modyfikować parametry graficzne wykresu, takie jak szerokość linii, typy punktów, opisy osi itp. Większość z tych argumentów ma te same nazwy co w pakiecie `graphics` (patrz też rozdział 4.2). Przykładowo w poniższym wykresie zmieniamy opisy osi X i Y, określamy kształt i kolor punktów.

```
xyplot(MDRD12~MDRD7|discrepancy.DR,kidney,type=c("p","smooth","r"),
     col="grey", pch=16, ylab="MDRD 30d", xlab="MDRD 7d")
```

W powyższym przykładzie argument `type` określa co ma być narysowane na wykresie. W powyższym przykładzie wartość `c("p","smooth","r")` wskazuje, że do wykresu rozrzutu poza punktami należy dorysować krzywą trendu liniowego oraz krzywą ruchomej średniej. Lista wartości, które może przyjmować argument `type` znajduje się w tabeli 4.4.

4.3.2 Szablony wykresów

W pakiecie `lattice` do tworzenia wykresu można wykorzystać piętnaście różnych funkcji. Każda z nich odpowiada innemu typowi wykresu. Każda zakłada różne liczby zmiennych lub typy zmiennych wyznaczając dla nich inną reprezentację graficzną, stąd też nazwa szablon. W tabeli 4.5 znajduje się lista funkcji wraz z krótkim opisem. W kolejnych sekcjach przedstawimy przykłady dla wybranych typów wykresów.

4.3.3 Formuła

W pakiecie `lattice` we wszystkich funkcjach tworzących wykresy rolę zmiennych określa się za pomocą formuły, czyli obiektu klasy **formula**. Pakiet `lattice` potrafi interpretować formuły o składni

```
L ~ R | C,
```

gdzie fragmenty formuły `L`, `R` i `C` zawierać może dowolną liczbę zmiennych rozdzielonych symbolami `+` i `*`. W przypadku pakietu `lattice` symbole `+` i `*` mają to samo znaczenie i mogą być stosowane zamiennie.

Przykładowa formuła wygląda następująco.

```
x + y ~ u + v | a + b
```

gdzie

- `x`, `y` - zmienne, które będą przedstawione na osi Y wykresu. Niektóre rodzaje wykresów wymagają podania przynajmniej jednej zmiennej po lewej stronie formuły, czyli po lewej stronie symbolu `~`.
- `u`, `v` - zmienne, które będą przedstawione na osi X wykresu. Większość wykresów wymaga podania przynajmniej jednej zmiennej po prawej stronie formuły.
- `a`, `b` - zmienne warunkujące. Warunkować można po jednej, dwóch lub większej liczbie zmiennych. Symbol `|` jest niepotrzebny jeżeli nie podaje się żadnych zmiennych warunkujących.

4.3.4 Mechanizm warunkowania

Użyteczną funkcjonalnością pakietu `lattice` jest możliwość warunkowania każdego wykresu. Warunkować można jedną lub większą liczbą zmiennych. Wyznaczą one podział całego zbioru danych na, niekoniecznie rozłączne, podzbiory. Każdy podzbiór będzie narysowany na innym panelu, ale na każdym panelu będzie użyty ten sam typ wykresu, panele będą współdzielić niektóre cechy jak np. zakresy osi itp. Dzięki temu, możemy porównać zależności pomiędzy zmiennymi dla podzbiorów obserwacji.

Zmienną warunkującą może być zarówno zmienna jakościowa jak i ilościowa. Jeżeli zmienną warunkującą jest zmienna jakościowa, to dla każdego poziomu tej zmiennej zostanie narysowany osobny panel. Jeżeli zmienną warunkującą jest zmienna ilościowa, to zostanie ona podzielona na przedziały i każdy panel odpowiadać będzie jednemu przedziałowi. Przedziały te nie muszą być rozłączne. Przedziały mogą być wyznaczone automatycznie lub w sposób kontrolowany, ale można również użyć funkcji `equal.count()` lub `shingle()` do kontrolowania liczby przedziałów lub wielkości zachodzenia przedziałów.

W poniższym przykładzie przedstawimy dwa wykresy. Pierwszy jest warunkowany zmienną jakościową o trzech poziomach, drugi jest warunkowany zmienną ilościową, która została podzielona na cztery przedziały. Rysunki 4.48 i 4.48 przedstawiają graficzny wynik poniższych poleceń.

```
histogram(~MDRD12 | therapy, data=kidney)
histogram(~MDRD12 | equal.count(kidney$donor.age,4), data=kidney)
```

4.3.5 Mechanizm grupowania

Mechanizm warunkowania pozwala na przedstawienie zależności dla różnych podpopulacji na oddzielnych panelach. Grupowanie pozwala na przedstawienie wykresów dla różnych podpopulacji na tym samym panelu. Zmienną grupującą czyli wyznaczającą podział na grupy określa się poprzez podanie argumentu `group`.

Poniżej przedstawiamy przykładowe wywołanie funkcji rysującej wykres na którym trzema różnymi kolorami przedstawiono jądrowy estymator gęstości zmiennej MDRD12 dla trzech różnych terapii. Wynikowy wykres znajduje się na rysunku 4.50.

```
densityplot(~MDRD12, group=therapy, data=kidney, plot.points=F)
```

4.3.6 Obiekt klasy trellis

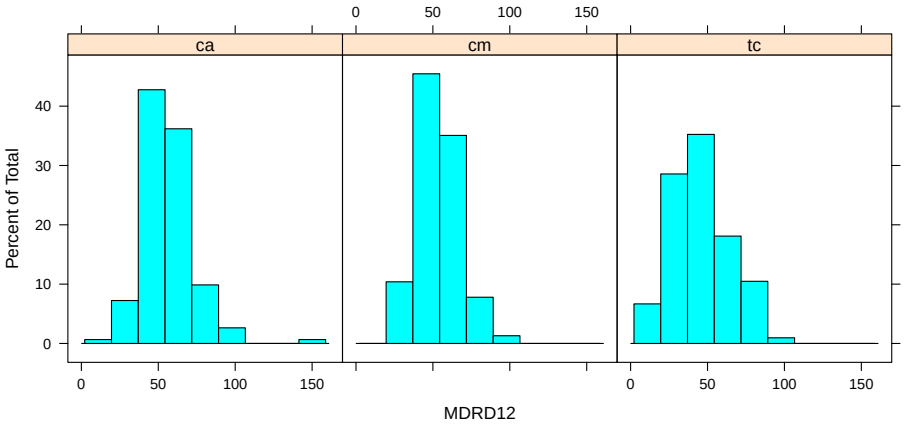
W odróżnieniu od pakietu `graphics`, funkcje graficzne z pakietu `lattice`, wymienione w tabeli 4.5 niczego nie rysują. Każda z tych funkcji tworzy opis wykresu i jako wynik zwraca obiekt klasy `trellis`. Obiekt ten określa jak wykres ma wyglądać.

Do narysowania wykresu, reprezentowanego przez obiekt klasy `trellis`, służą generyczne funkcje `plot()` i `print()`. Obie funkcje mają identyczne działanie i można ich używać zamiennie.

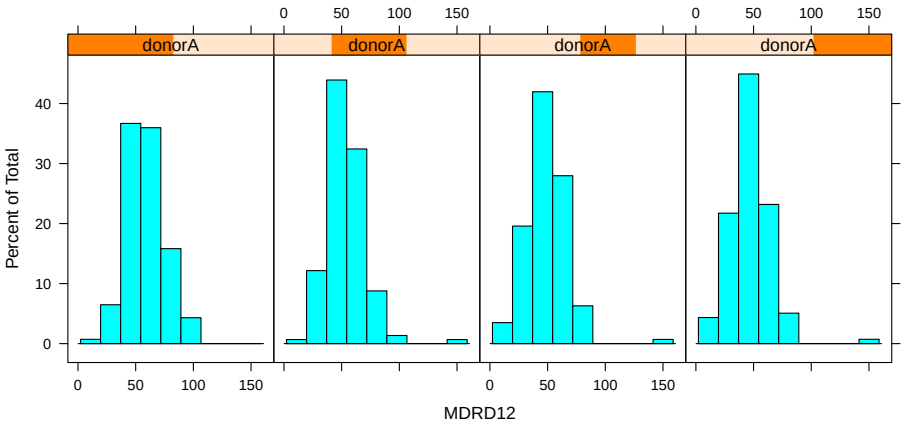
Dlaczego więc poniższa komenda powoduje powstanie wykresu?

```
xyplot(MDRD12~MDRD7|discrepancy.DR, kidney)
```

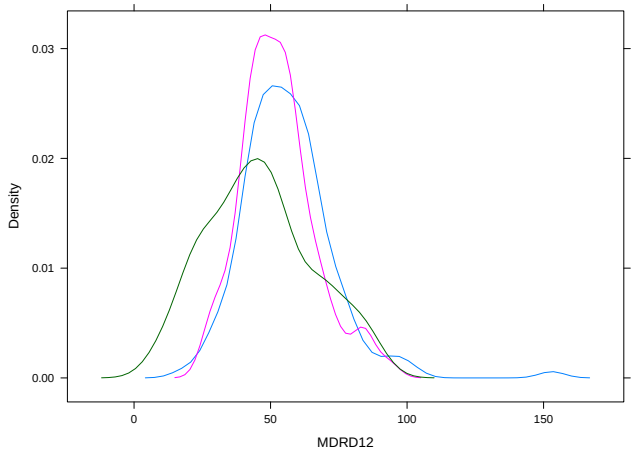
Wynikiem funkcji `xyplot()` jest obiekt klasy `trellis`. Ponieważ wynik ten nie został przypisany do żadnej zmiennej, więc wywoływana jest automatycznie generyczna funkcja `print()`. Funkcja `print()` jest przeciążona dla klasy `trellis` i jako efekt uboczny rysuje wykres na aktywnym urządzeniu graficznym. Funkcja `plot()` ma identyczne działanie, z ta tylko różnicą, że nie jest automatycznie wywoływana. Więcej informacji o funkcjach generycznych w programie R znaleźć można w rozdziale 1.6.2.3.



Rysunek 4.48: Warunkowanie zmienną jakościową o trzech poziomach.



Rysunek 4.49: Warunkowanie zmienną ilościową podzieloną na cztery przedziały.



Rysunek 4.50: Wykres dla trzech grup, podpopulacje przedstawiane są na wspólnym panelu.

```
>  # ta komenda nie narysuje niczego na ekranie
>  wykres <- xyplot(MDRD12~MDRD7|discrepancy.DR, kidney)
>  # jakiej klasy jest wynik
>  class(wykres)
[1] "trellis"
>  # tutaj funkcja print jest wywołana w sposób niejawny
>  wykres
>  # tutaj funkcja print jest wywołana w sposób jawny
>  print(wykres)
>  plot(wykres)
```

Jeżeli zapomnimy, że funkcje graficzne same nie rysują wykresu, a jedynie przygotowują obiekt z opisem wykresu, to możemy znaleźć się w sytuacji gdy mała modyfikacja kodu spowoduje, że wykresy nagle przestaną się wyświetlać. Wystarczy, że nie wywoła się niejawnie funkcja `print()`. Typowe przykłady, w który funkcja `print()` nie będzie wywołana automagicznie to:

- wynik działania funkcji graficznej z pakietu `lattice` jest przypisany od zmiennej z użyciem operatora przypisania,
- funkcja graficzna z pakietu `lattice` jest wywołana wewnątrz bloku kodu otoczonego nawiasami `{` i `}`, np. w pętli lub wewnątrz funkcji,
- skrypt R jest wywołany z użyciem funkcji `source()`,
- wynikowy obiekt został oznaczony jako niewidoczny, np. poprzez użycie funkcji `invisible()`.

Jeżeli chcemy być pewni, że funkcja graficzna narysuje wykres, powinniśmy jawnie wywołać funkcję `print()` lub `plot()`.

Operator [,] oraz funkcja t()

Nie tylko funkcja `print()` i `plot()` jest przeciążona dla klasy `trellis`. Inna przeciążoną funkcją jest `t()` oraz operator indeksowania `[,]`.

Jeżeli w formule opisującej wykres użyliśmy zmiennej warunkującej to otrzymamy wykres z kilkoma panelami. Panele te są rozmieszczone na planie macierzy, o określonej liczbie kolumn i wierszy. Kolejność paneli jest określona przez zmienne warunkujące. Domyślnie jeżeli warunkujemy jedną zmienną, to kolejne wartości tej zmiennej odpowiadają panelom rysowanym kolejno od lewej do prawej. Jeżeli warunkujemy dwoma zmiennymi to wiersze odpowiadają poziomem jednej zmiennej, a kolumny poziomom drugiej zmiennej itp.

Generyczna funkcja `dim()` dla obiektu typu `trellis` jako wynik przekazuje wektor z liczbami wartości zmiennych warunkujących. Generyczna funkcja `t()` dla obiektu typu `trellis` zmienia kolejność rysowania paneli transponując wiersze z kolumnami, a więc zamieniając kolejność zmiennych warunkujących. Generyczny operator `[,]` dla obiektu typu `trellis` pozwala na wybranie tylko wskazanych wierszy lub kolumn z siatki paneli. Używając tego operatora możemy rysować tylko wybrane panele.

Na poniższym przykładzie przedstawimy przykład użycia wymienionych powyżej funkcji.

```
> wykres <- xyplot(MDRD7~MDRD12|diabetes+therapy,kidney,pch=19)
> dim(wykres)
[1] 2 3
> dim(wykres[,1])
[1] 2 1
> # rysujemy tylko cztery wybrane panele z tej siatki
> plot(wykres[1:2,c(1,3)])
> # rysujemy panele w innej kolejności
> plot(t(wykres))
```

Funkcja update()

Funkcje graficzne w pakiecie `lattice` tworzą obiekt, który opisuje wykres. Obiekt ten można następnie narysować np. z użyciem funkcji `plot()`. Jeżeli po narysowaniu wykresu, będziemy chcieli zmienić jakiś jego element, to możemy na nowo uruchomić funkcję tworzącą wykres lub użyć funkcji `update()`, która uaktualnia obiekt `trellis`.

Używanie funkcji `update()` pozwala w wielu sytuacjach na zwiększenie czytelności kodu poprzez skracanie wywoływanych poleceń. Ułatwia też wykonywanie podobnych wykresów na różnych zbiorach danych, podmieniać możemy nie tylko parametry graficzne, ale też zmienne lub całe dane.

Poniżej przedstawiamy kilka przykładów użycia funkcji `update()`. Wynik działania przedstawiony jest na rysunku 4.51.

```
# tworzymy wykres rozrzutu
macierz <- xyplot( MDRD12~receiver.age|diabetes, data=kidney)
# rysujemy powyżej zdefiniowany wykres
plot(macierz)
# zmieniamy kształt punktów
update(macierz,pch=19)
# dodajemy krzywe regresji
update(macierz,type=c("p","smooth","r"))
# rysujemy wykres z innymi parametrami dotyczącego położenia paneli
update(macierz,layout=c(0,9),between=list(x=c(0,0.5),y=0.5))
```

4.3.7 Legenda

W pakiecie `lattice`, aby dodać do wykresu legendę, należy określić jakąkolwiek jej właściwość za pomocą argumentu `key`. Argument ten można przekazać każdej funkcji z pakietu `lattice()`. Aby zdefiniować obiekt opisujący legendę można użyć funkcji `draw.key()`. Za pomocą tej funkcji możemy narysować dowolną legendę w dowolnym miejscu i o dowolnej zawartości.

Legendę można dodać wskazując argument `auto.key=T`. W tym przypadku zawartość legendy zostanie zbudowana automatycznie przez funkcję tworzącą wykres. Mamy możliwość modyfikacji poszczególnych właściwości tej legendy. Zamiast wartości `TRUE` należy podać listę określającą właściwości, które chcemy zmodyfikować.

Poniżej przedstawiamy wywołanie wykresu z automatycznie utworzoną legendą i z legendą przesuniętą do prawego brzegu. Kompletną listę właściwości obiektu legendy znaleźć można w książce [38]. Wykresy utworzone przez poniższe polecenia znajdują się na rysunku 4.52.

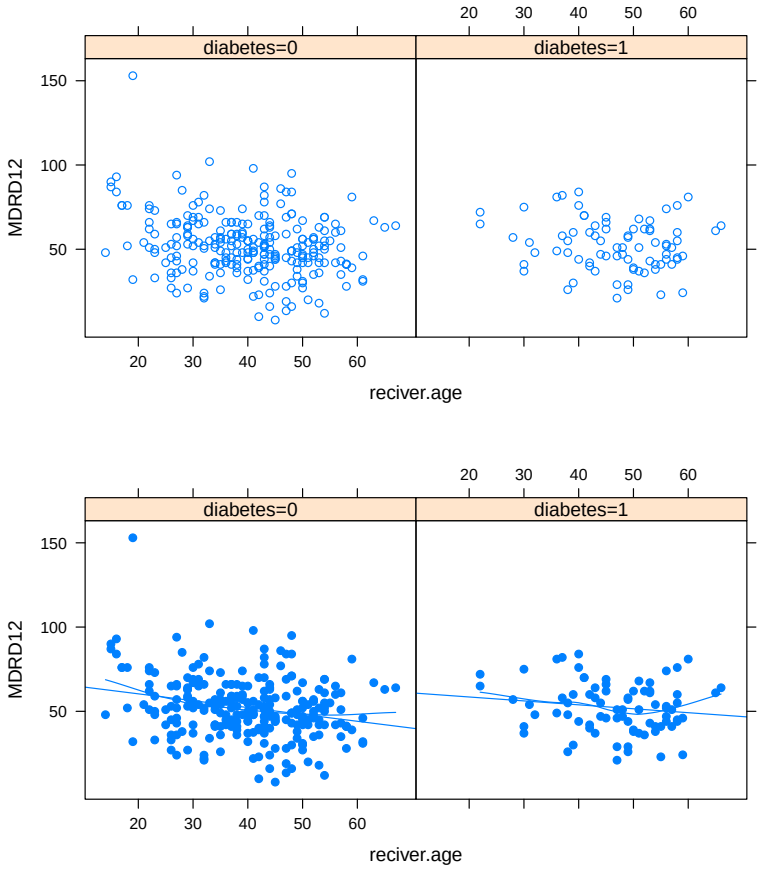
```
densityplot(~MDRD12, group=therapy, auto.key=T)
densityplot(~MDRD12, group=therapy, auto.key=list(space="right",
    columns=1))
```

4.3.8 Pozycjonowanie wykresu, wiele wykresów na rysunku

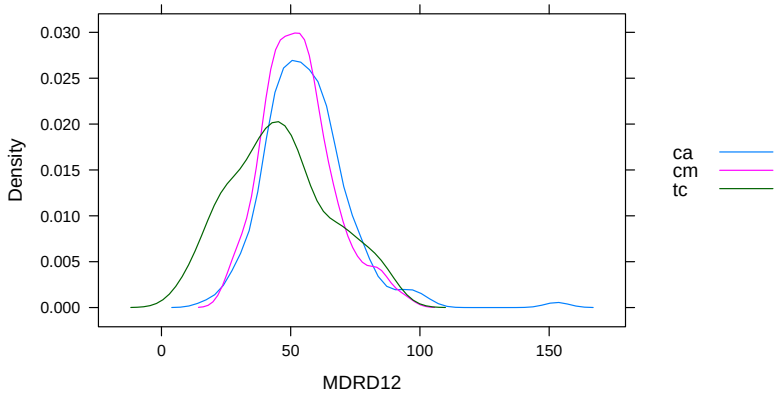
W pakiecie `lattice` wszystkie wykresy są rysowane poprzez wywołanie funkcji `print()` lub `plot()`. Domyślnie wykres rysowany jest w taki sposób, by wypełnił cały obszar dostępny do rysowania. Aby narysować więcej niż jeden wykres można wykorzystać argumenty `split` lub `position` w funkcji `plot()` lub `print()`.

Jeżeli planujemy rysować wykresy na planie siatki to wygodnie jest użyć argumentu `split`. Wywołując funkcję `plot()`, należy za wartość tego argumentu podać wektor czteroelementowy `c(x,y,mx,my)`. Po takim wywołaniu, obszar do rysowania podzielony będzie na siatkę o `mx` wierszach i `my` kolumnach, a wykres zostanie wrysowany w komórkę o współrzędnych `x,y` tej siatki. Poniższy przykład ilustruje zastosowanie argumentu `split`. Wynik wywołania tego kodu przedstawiony jest na rysunku 4.53. Argument `newpage=F` wyłącza inicjacje urządzenia graficznego i zapobiega nadpisaniu już narysowanych elementów graficznych.

```
wykres <- xyplot(MDRD7~MDRD12,kidney, pch=19)
# lewy
plot(wykres, split=c(1,1,2,1))
# prawy gorny
plot(wykres, split=c(2,1,2,2), newpage=FALSE)
# prawy dolny
plot(wykres, split=c(3,2,4,2), newpage=FALSE)
plot(wykres, split=c(4,2,4,2), newpage=FALSE)
```



Rysunek 4.51: Przykładowe użycie funkcji update(). Aktualizowany był argument type i pch



Rysunek 4.52: Argumentem auto.key zmieniamy miejsce, w którym jest rysowana legenda.

Jeżeli chcemy narysować wykresy w dowolnym miejscu, niekoniecznie na regularnej siatce, to wygodniej będzie użyć argumentu `position`. Wywołując funkcję `plot()` należy za wartość tego argumentu podać wektor czteroelementowy `c(x1,y1,x2,y2)` określający współrzędne prostokąta, w którym narysowany ma być wykres. Współrzędne powinny być podane w układzie współrzędnych, w których lewy dolny róg okna graficznego ma współrzędne `0,0` a prawy górny ma współrzędne `1,1`.

Poniższy przykład ilustruje zastosowanie parametru `position`. Wynik wywołania tego kodu przedstawiony jest na rysunku 4.54.

```
wykres <- xyplot(MDRD7~MDRD12,kidney, pch=19)
# trzy wykresy, pozycje zostały tak wybrane aby wykresy się na
    siebie nakładały
plot(wykres, position=c(0,0,.8,.8))
# argument newpage=FALSE wyłącza rysowanie nowego wykresu
plot(wykres, position=c(0.35,0.35,.9,.9), newpage=FALSE)
plot(wykres, position=c(0.7,0.7,1,1), newpage=FALSE)
```

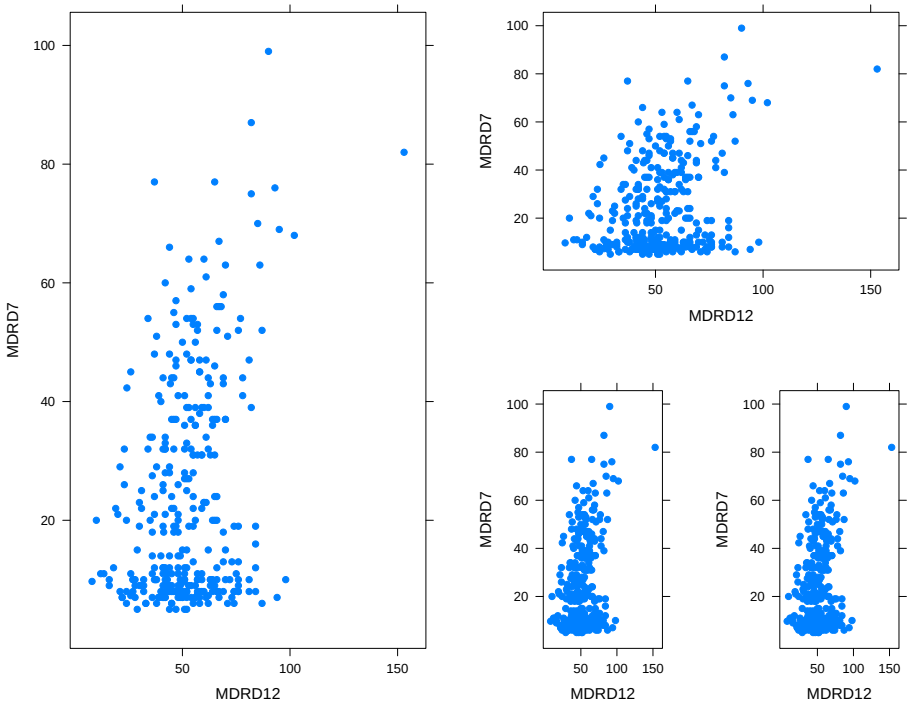
4.3.9 Proporcje jednostek na osiach i reguła 45 stopni

W pakiecie `lattice` możemy kontrolować względne proporcje długości odcinków na osiach OX i OY.

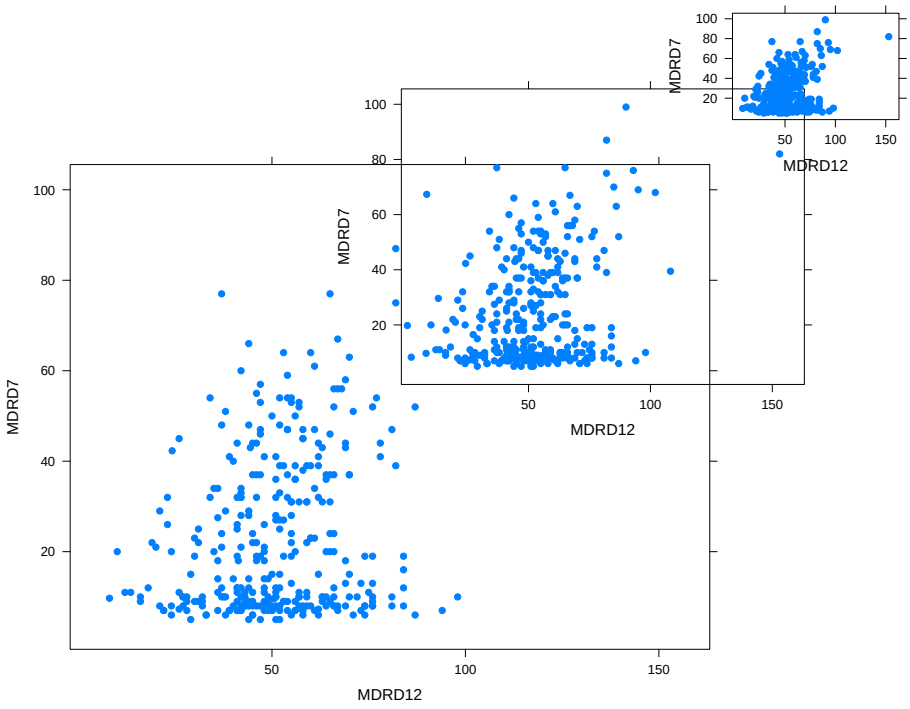
Jeżeli obie osie wykresu przedstawiają wartości w tej samej jednostce, np długość w metrach, wiek w latach, zysk w PLN itp, to w pewnych sytuacjach chcemy, by na wykresie były zachowane proporcje i by odległość 1cm na osi OX odpowiadała różnicy tej samej liczby jednostek co odległość 1cm na osi OY. Dodatkowo te proporcje nie powinny zależeć od tego czy wykres rysowany jest w oknie „graphics" programu R czy do pliku graficznego. Do kontroli proporcji na osiach wykresu służy argument `aspect`. Przypisać można mu jedną z trzech wartości:

- `"fill"` (domyślna), wykres wypełni panel w sposób maksymalny, proporcje na osiach zależą od wymiarów panelu a pośrednio od wymiarów okna graficznego lub pliku w którym wykres jest rysowany.
- `"iso"`, wykres narysowany będzie z zachowaniem względnych proporcji jednostek na obu osiach.
- `"xy"`, wykres narysowany będzie z zachowaniem reguły 45 stopni. Reguła ta dotyczy czytelnego przedstawiania przebiegów czasowych. Przyjęto stosować regułę kciuka, zgodnie z którą najczytelniejsza reprezentacja szeregu czasowego to taka, w której średni kąt nachylenia kolejnych odcinków wynosi 45 stopni.

Aby zaprezentować regułę 45 stopni wykorzystamy zbiór danych `sunspot.year`, zawierający szereg czasowy o długości 289 obserwacji opisujących roczną liczbę plam na słońcu.



Rysunek 4.53: Przykład użycia argumentu split.



Rysunek 4.54: Przykład użycia argumentu position.

Poniżej przedstawiamy przykład narysowania wykresu z zachowaniem reguły 45 stopni. Wykres odpowiadający temu wywołaniu znajduje się na rysunku 4.55. Na tym rysunku łatwo zauważyć, że przy każdym lokalnym maksimum liczby plam na słońcu, przyrost funkcji jest większy niż spadek liczby plam, co oznacza, że liczba plam jeżeli rośnie to gwałtownie a jeżeli spada to powoli. Taką zależność znacznie trudniej zauważyć, jeżeli nie zachowa się reguły 45 stopni.

Pilnego czytelnika zachęcamy do eksperymentów!

```
form = sunspot.year~1:length(sunspot.year)
xyplot(form, type="l", aspect="xy", xlab="", subset=1:140)
xyplot(form, type="l", aspect="xy", xlab="", subset=141:280)
# porównanie aspect="iso" i aspect="fill"
xyplot(MDRD12~MDRD7, kidney, pch=19, aspect="iso")
xyplot(MDRD12~MDRD7, kidney, pch=19, aspect="fill")
```

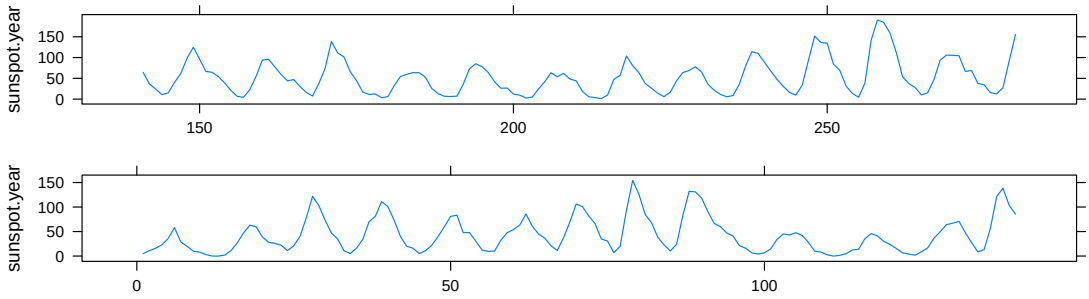
4.3.10 Modyfikacja parametrów graficznych

W pakiecie `lattice` ustawienia takie jak grubość linii, typ linii, kolory itp można modyfikować na dwa sposoby. Lokalnie, poprzez podanie odpowiednich argumentów do funkcji graficznej rysującej wykres, lub globalnie, modyfikując globalne ustawienia rysowania.

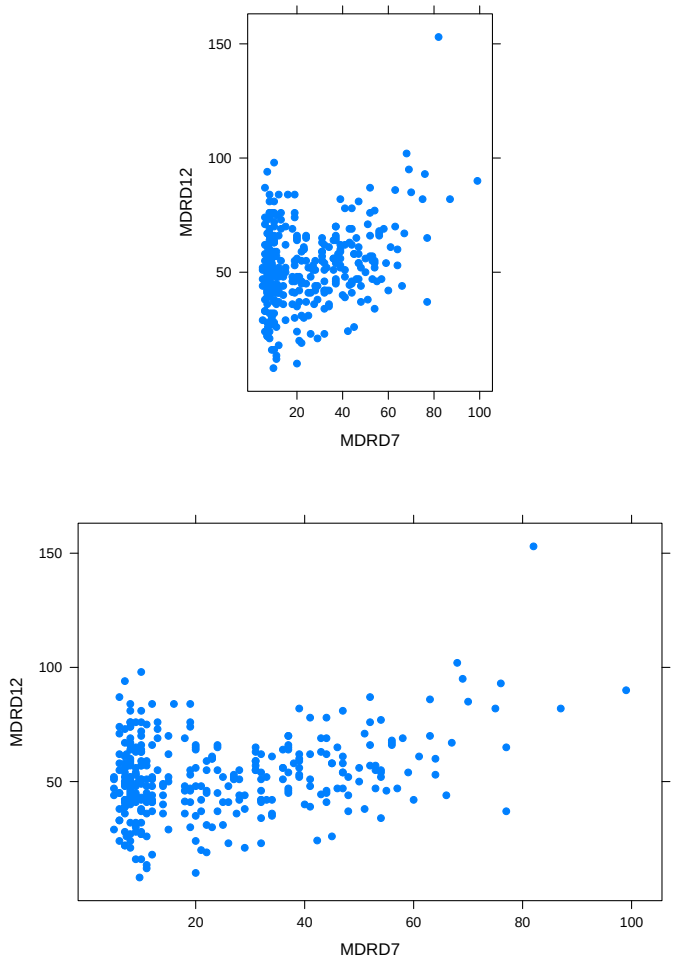
Do modyfikacji globalnych ustawień służą funkcje `trellis.par.get()` i `trellis.par.set()`. Jak wyglądają elementy graficzne przy aktualnych ustawieniach można sprawdzić wywołując funkcję `show.settings()`. Przykład wywołania tej funkcji znajduje się na rysunku 4.58. Dodatkowo z tego wykresu możemy odczytać, jak nazywają się poszczególne modyfikowalne parametry graficzne. Przykładowo w trzecim wierszu i pierwszej kolumnie przedstawione są właściwości parametrów `plot.symbol` i `plot.line`.

W poniższym przykładzie sprawdzamy, jakie są wartości parametru `plot.line` i zmieniamy domyślną grubość linii na trzy punkty.

```
> # aktualne ustawienia właściwości linii
> trellis.par.get("plot.line")
$alpha
[1] 1
$col
[1] "#0080ff"
$lty
[1] 1
$lwd
[1] 1
> # zmieniamy tylko grubość linii na 3 punkty
> trellis.par.set(plot.line=list(lwd=3))
```



Rysunek 4.55: Liczba plam na słońcu z zachowaniem reguły 45 stopni. Przy takich proporcjach na wykresie, wyraźnie widać że prawe zbocza podwyższonych aktywności słońca są łagodniejsze niż lewe zbocza. Nie widać tego jeżeli wykres jest zbyt ściśnięty lub zbyt rozszerzony.



Rysunek 4.56: Przykładowe użycie argumentu aspect. Na górze zachowane są proporcje pomiędzy wartościami na osi OX i OY (aspect="iso") na dole nie (aspect="fill").

4.3.11 Panele

W pakiecie `lattice` wskazanie zmiennej lub zmiennych warunkujących powoduje wyświetlanie wykresów na kolejnych panelach. Domyślnie każdy wykres ma przedefiniowaną funkcję, która jest wykorzystana do narysowania panelu. Mamy też możliwość modyfikacji zachowania funkcji rysującej kolejne panele. Poniżej przedstawimy kilka przykładów.

Uwolnione osie

Domyślnie na wszystkich panelach osie mają te same zakresy. To rozwiązanie pozwala na łatwe porównywanie wartości pomiędzy panelami. Są jednak sytuacje, gdy chcemy uwolnić osie na różnych panelach by zwiększyć czytelność. Możemy to zrobić modyfikując argument `scales`, który wskazuje na listę opisującą zachowanie każdej z osi.

Do każdej z osi możemy przypisać jedną z wartości:

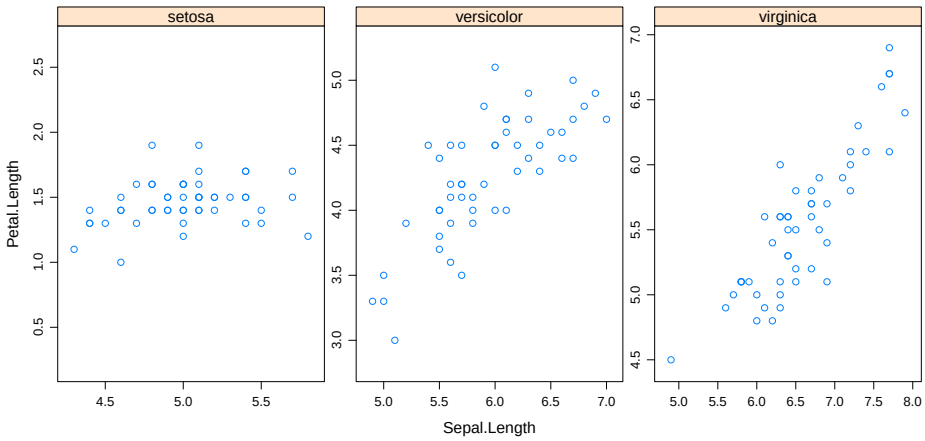
- `same`, ta opcja powoduje, że wszystkie panele będą miały ten sam zakres wartości na wskazanej osi,
- `free`, ta opcja powoduje, że dla każdego z paneli zakres wskazanej osi będzie ustalony niezależnie,
- `sliced`, ta opcja powoduje, że każdy z paneli może mieć inne zakresy zmienności ale przy zachowaniu proporcji. Jeżeli na jednym panelu oś rozciąga się na `k` jednostek, to na pozostałych panelach oś również będzie rozciągała się przez `k` jednostek, osie na poszczególnych panelach mogą zaczynać się w różnych punktach.

Powyższą wartość możemy ustawić niezależnie dla każdej z osi. Na poniższym przykładzie wywołujemy funkcję `xyplot()`, zaznaczając, że panele mogą mieć różne osie x i y, ale na osi y powinny być zachowane proporcje we wszystkich panelach. Wynik działania tego polecenia przedstawiony jest na rysunku 4.57.

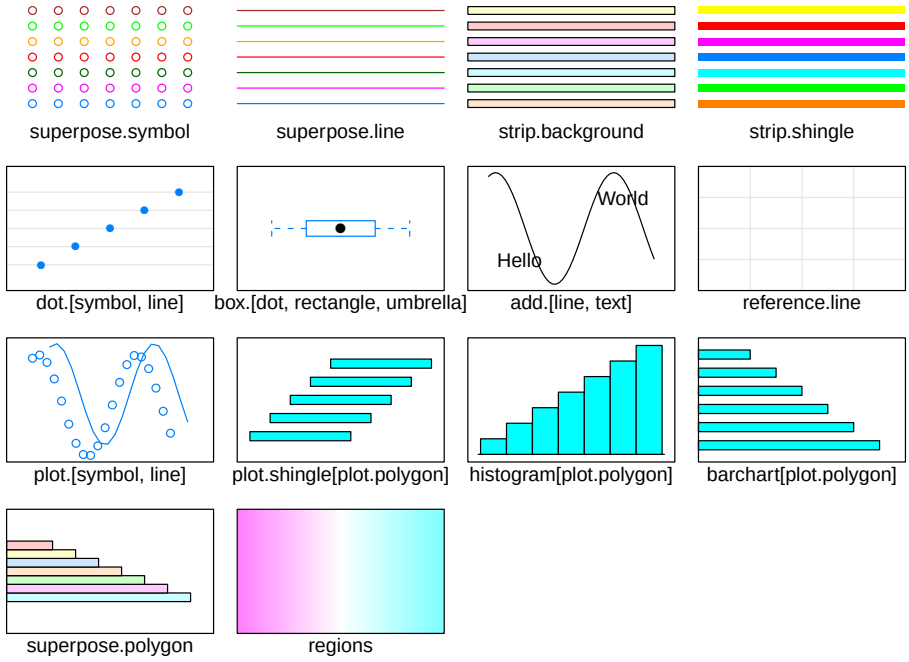
```
xyplot(Petal.Length~Sepal.Length|Species, iris,
       scales=list(x="free", y="sliced"))
```

Modyfikacja funkcji rysującej panele

W tabeli 4.5 przedstawiliśmy funkcje do rysowania wykresów. Każda z tych funkcji przygotowuje obiekt klasy `trellis` opisujący wykres, który ma być narysowany. Jedną z właściwości tego obiektu jest właściwość `panel`, wskazująca na funkcję, która ma narysować zawartość każdego z paneli. I tak funkcja `xyplot()` do rysowania zawartości paneli ustawia domyślnie funkcję `panel.xyplot()`, funkcja `barchart()` do rysowania zawartości paneli ustawia domyślnie funkcję `panel.barchart()` itd.



Rysunek 4.57: Przykład użycia argumentu scales. Na osiach różnych paneli zachowane są proporcje, zakresy wartości są jednak różne.



Rysunek 4.58: Przykładowe wywołanie funkcji show.settings().

Tę domyślną funkcję możemy zmodyfikować lub zastąpić inną. Przedstawmy tę możliwość na poniższym przykładzie. Zacznijmy od narysowania wykresu paskowego z użyciem funkcji `barchart()`. Wynik tej funkcji posiada właściwość `panel` domyślnie równą `panel.barchart`.

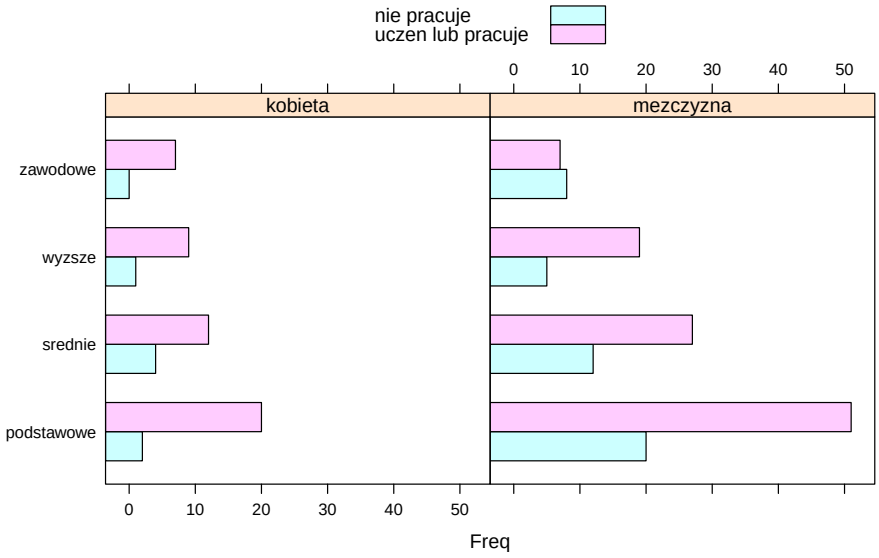
```
> tmp = as.data.frame(table(daneSoc$wyksztalcenie, daneSoc$plec,
    daneSoc$praca ))
> wykres = barchart(wyksztalcenie~Freq|plec, groups=praca,
       auto.key=TRUE, data=tmp)
> wykres$panel
[1] "panel.barchart"
```

Zdefiniujmy teraz własną funkcję `nasz.panel()`, która wykorzysta funkcję `panel.grid()` do dorysowania pionowych linii oraz zmodyfikuje zachowanie funkcji `panel.barchart` poprzez wyłączenie rysowania obrysu pasków. Wewnątrz funkcji `nasz.panel()` możemy też użyć dowolnych innych funkcji graficznych, modyfikując tym samym wygląd każdego z paneli.

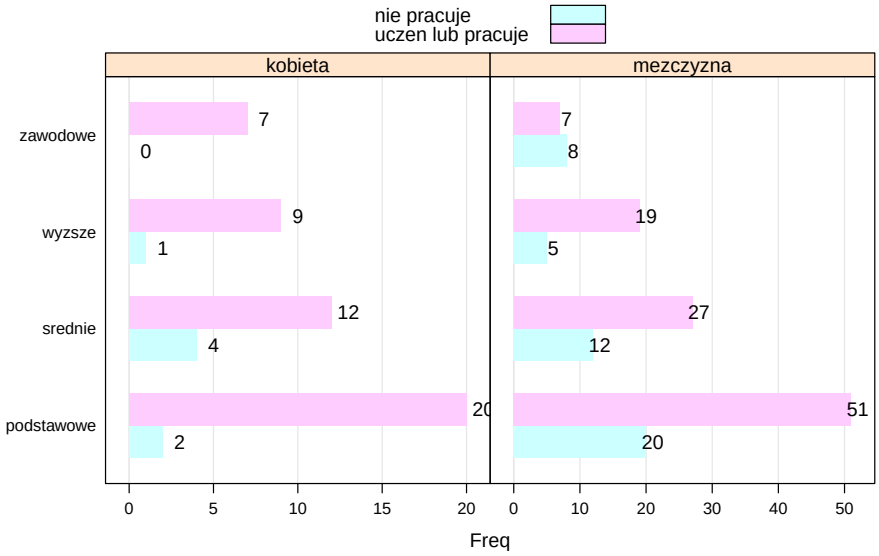
```
> # tworzymy własną funkcję rysującą panel, w tym przypadku
    dorysowujemy linie pomocnicze, dodajemy liczebności oraz
    usuwamy obramówkę pasków
>  nasz.panel <- function(..., border) {
+     panel.grid(h=0, v=-1)
+     panel.barchart(..., border="transparent")
+      panel.text(list(...)$x+1, as.numeric(list(...)$y) -0.5 +
+           as.numeric(list(...)$groups[list(...)$subscripts])/3,
+           as.numeric(list(...)$x))
+     }
>  update(wykres, panel=nasz.panel, scales=list(x="free"),
         origin=0)
```

Wyniki działania powyższych dwóch uruchomień znajdują się na rysunkach 4.59 i 4.60. W poniższym przykładzie modyfikujemy funkcję rysującą panel w taki sposób, by na każdym z paneli zaznaczała wszystkie obserwacje szarymi kropkami, a obserwacje należące do określonej grupy uwydatnia dodatkowo niebieskimi plusami.

```
xyplot(cisnienie.skurczowe~cisnienie.rozkurczowe|plec, daneSoc,
  panel = function(x,y,...) {
     lpoints(cisnienie.rozkurczowe,cisnienie.skurczowe,
         pch=19, col='grey', cex=0.5)
     panel.xyplot(x,y,pch='+', cex=2)
  }
)
```



Rysunek 4.59: Panele narysowane przez funkcję domyślną panel.barchart().



Rysunek 4.60: Panele narysowane przez naszą funkcję nasz.panel().

4.3.12 Wybrane funkcje graficzne z pakietu lattice

4.3.12.1 Wykres rozrzutu, funkcja: xyplot(lattice), splom(lattice)

Funkcja `xyplot()` służy do rysowania wykresów rozrzutu. W formule po lewej stronie wskazujemy zmienną lub zmienne, których wartości mają odpowiadać współrzędnym OY, prawa strona wskazuje zmienną, której wartości odpowiadać będą współrzędnym OX.

Funkcja `splom()` służy do rysowania macierzy wykresów rozrzutu. Pierwszym argumentem tej funkcji nie jest formuła, ale ramka danych zawierająca zmienne ilościowe. Funkcja `splom()` tworzy siatkę wykresów rozrzutu, dla każdej pary kolumn ze wskazanej ramki danych.

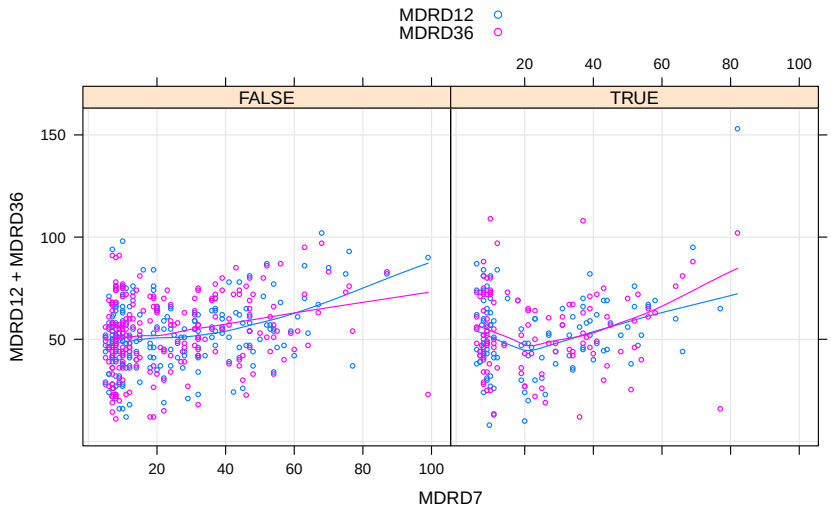
Obie funkcje umożliwiają określenie argumentu `type`, który określa w jaki sposób oznaczane mają być kolejne punkty. Domyślnie każda obserwacja zaznaczana jest na wykresie przez punkt, odpowiada to ustawieniu `type='p'`. Inne możliwe wartości wymienione są w tabeli 4.4. Za wartość argumentu `type` można podać kilka wartości, w tym przypadku na wykresie narysowane będą wszystkie wskazane sposoby prezentacji punktów.

W przykładzie poniżej `type=c("p","smooth","g")` oznacza, że na wykresie rozrzutu obserwacje będą oznaczane punktami (`type="p"`), do wykresu dorysowane będą pomocnicze linie siatki (`type="g"`), oraz krzywa ruchomej średniej (`type="smooth"`). Poniżej przedstawiamy przykład użycia funkcji `xyplot()` i `splom()`. Wykresy odpowiadające tym wywołaniom przedstawione są na rysunkach 4.61 i 4.62.

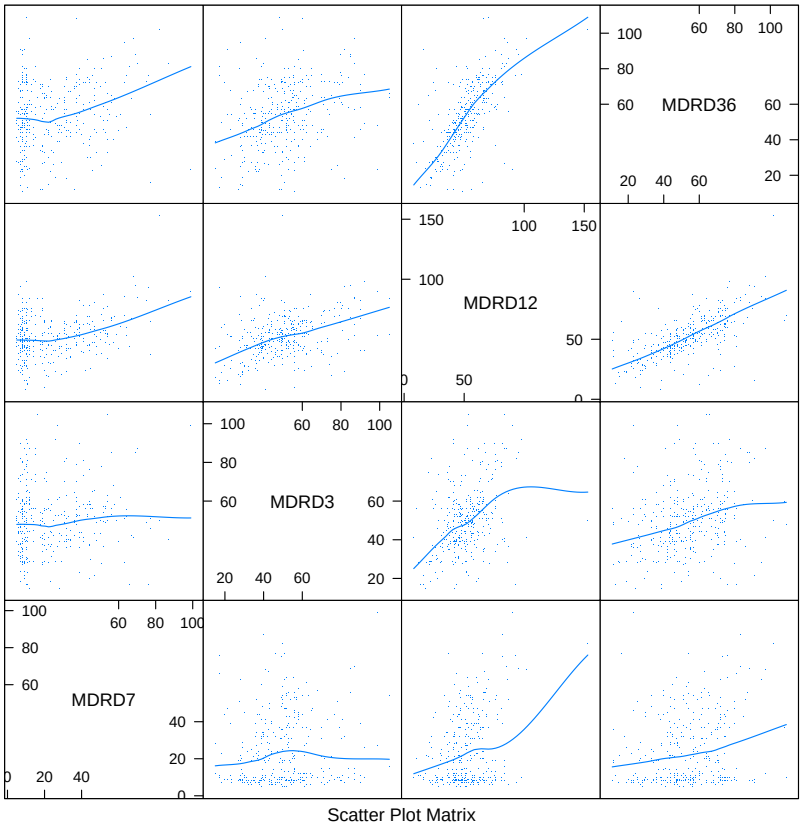
```
> xyplot(MDRD12 + MDRD36 ~ MDRD7 | discrepancy.DR==0, kidney,
       type=c("p","smooth","g"), cex=0.5, auto.key=T)
> splom(kidney[,c(9,11,13,15)], type=c("smooth","p"), pch='.')
```

Tabela 4.4: Wartości, które przyjmować może argument type.

Wartość	Znaczenie
`p`	Rysowane są punkty o współrzędnych odpowiadającym wartościom wskazanych zmiennych.
`l`	Zamiast punktów rysowana jest linia łącząca ww. punkty.
`b`	Rysowane są i punkty i linia.
`o`	Rysowane są i punkty i linia.
`s`	Punkty łączone są krzywą schodkową.
`h`	Rysowana jest pionowa linia od początku układu współrzędnych do współrzędnej punktu.
`a`	Dla każdej grupy wyznaczane są średnie wartości punktów w tych grupach, następnie te średnie wartości łączone są krzywą.
`r`	Do wykresu dodawana jest krzywa regresji.
`smooth`	Do wykresu dodawana jest krzywa lokalne wygładzenia.
`g`	Do wykresu dodawane są pomocnicze linie siatki.



Rysunek 4.61: Przykładowe wywołanie funkcji xyplot() z argumentem type=c("p", "smooth", "r").



Rysunek 4.62: Przykładowe wywołanie funkcji splom(), dodatkowe argumenty type=c("p", "smooth") i pch=".".

4.3.12.2 Wykres pudełkowy i paskowy, funkcja: stripplot(lattice), bwplot(lattice)

Funkcja `xyplot()` przedstawia dwie lub więcej zmiennych ilościowych. Jeżeli chcemy pokazać jak zmienia się pewna zmienna ilościowa dla różnych poziomów zmiennej jakościowej, możemy do tego celu wykorzystać funkcję `stripplot()`. Funkcja `stripplot()` w zasadzie rysuje wykres rozrzutu, zamieniając wartości zmiennej jakościowej na odpowiadający im numer poziomu. Ustawiając argument `jitter.data=TRUE` dodatkowo otrzymujemy efekt zaburzenia współrzędnych szumem, przez co punkty odpowiadające różnym obserwacjom będą miały mniejszą szansę zachodzić na siebie. W sytuacji, gdy dużo obserwacji ma podobne wartości dodanie półprzezroczystości pomaga zwiększyć czytelność. Funkcja `bwplot()` odpowiada za przygotowanie wykresu typu ramka wąsy, czyli wykresu pudełkowego. Poniżej przedstawiamy przykłady wywołania obu funkcji, wykresy utworzone z użyciem tych poleceń przedstawione są na rysunkach 4.63 i 4.65.

```
# liczbę niezgodności w antygenach AB dzielimy na trzy przedziały
discrepancy = equal.count(kidney$discrepancy.AB, number=3)
# wykres pudełkowy
bwplot(therapy~MDRD12|discrepancy, data=kidney, varwidth=T)
# wykres paskowy
stripplot(factor(discrepancy.AB)~MDRD7, kidney, jitter.data=T,
    alpha=0.5)
```

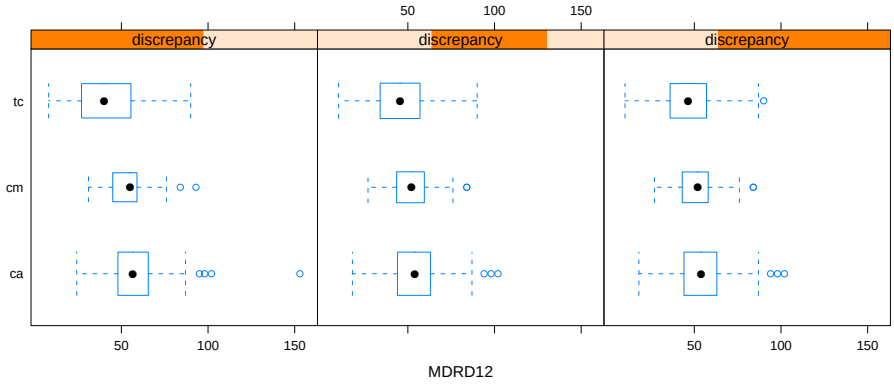
4.3.12.3 Wykres kropkowy i paskowy, funkcja: dotplot(lattice), barchart(lattice)

Do graficznej prezentacji tablic kontyngencji można wykorzystać funkcję `dotplot()`. Domyślnie ta funkcja zaznacza punktami na poziomych osiach wartości ze wskazanego zbioru danych. Dane mogą być rysowane na różnych panelach, po wyłączeniu grupowania `groups=F`, lub na jednym panelu po włączeniu grupowania `groups=T`.

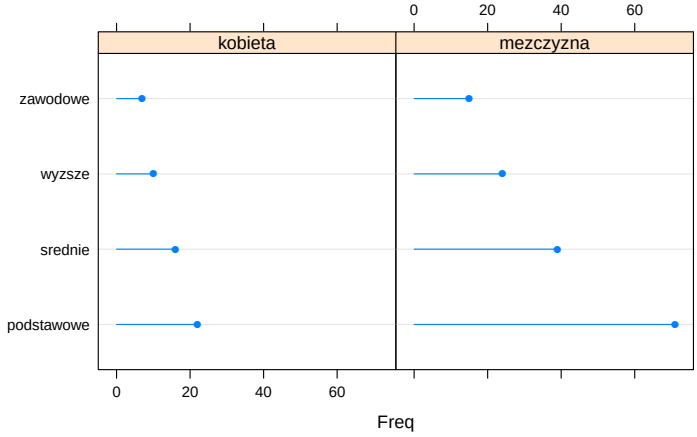
Argument `origin` pozwala na wskazanie, która wartość ma być uznana za początek układu współrzędnych. Jeżeli ta wartość zostanie określona, to skala na osi będzie tak dobrana, by również zawierała wskazany początek układu współrzędnych. Przykładowo na rysunku 4.64 wykres został tak zmodyfikowany by oś OX zawierała wartość 0.

Dla wykresu kropkowego możemy określić argument `type`. Możliwe wartości dla tego argumentu przedstawione są w tabeli 4.4.

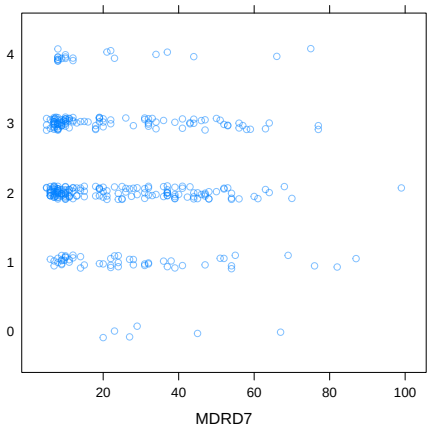
Tabele kontyngencji często przedstawia się również z użyciem wykresu paskowego, który można przygotować z użyciem funkcji `barchart()`. Poniżej przykład wywołania. Odpowiadający mu wykres przedstawialiśmy na rysunku 4.59.



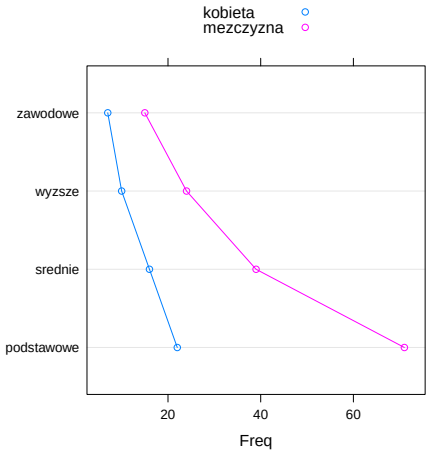
Rysunek 4.63: Przykładowe wywołanie funkcji bwplot().



Rysunek 4.64: Przykładowe wywołanie funkcji dotplot().



Rysunek 4.65: Przykładowe wywołanie funkcji stripplot().



Rysunek 4.66: Przykładowe wywołanie funkcji dotplot().

```
wPlec = table(daneSoc$wyksztalcenie, daneSoc$plec)
dotplot(wPlec, groups=F, origin=0, type=c("p","h"))
dotplot(wPlec, type=c("o"),auto.key=list(space="right"))

attach(daneSoc)
# argumentem data musi być ramka danych
tmp = as.data.frame(table(wyksztalcenie, plec, praca ))
barchart(wyksztalcenie~Freq|plec, groups=praca, auto.key=T,
   data=tmp)
```

4.3.12.4 Wykres profili, funkcja: parallel(lattice)

Funkcja `parellel()` przedstawia profil zmienności wielu cech jednocześnie.

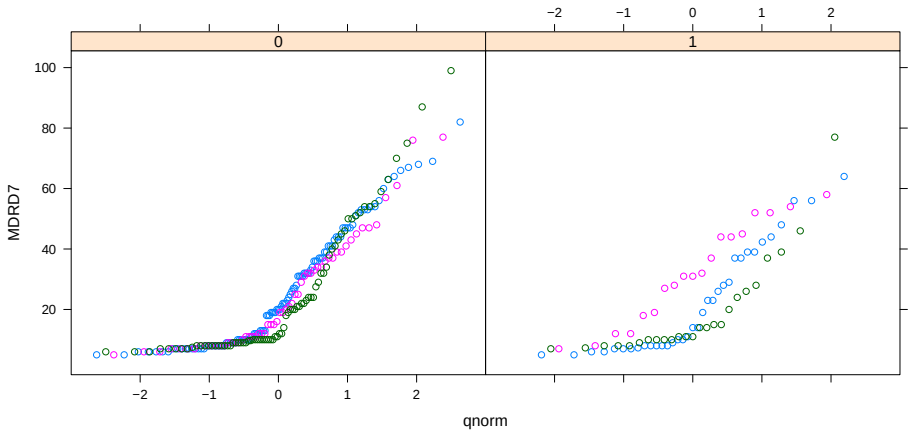
```
parallel(~kidney[,c(9:16)], groups=MDRD7<30, kidney, alpha=0.2,
     horizontal.axis = FALSE, scales = list(x = list(rot = 90)))
```

4.3.12.5 Wykres dystrybuanty empirycznej i gęstości, funkcje: histogram(lattice), density(lattice), qq(lattice), qqmath(lattice)

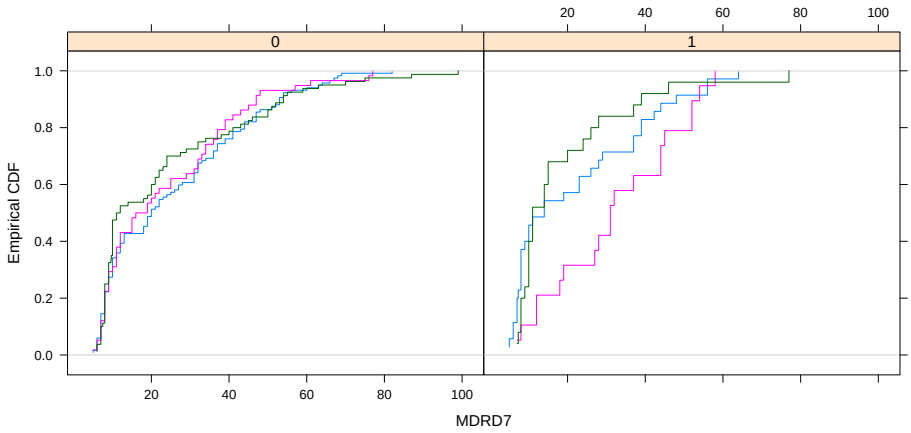
Do graficznego opisu zmiennej ilościowej wykorzystuje się ocenę gęstości lub dystrybuanty. Typowym estymatorem dystrybuanty jest dystrybuanta empiryczna, a popularnymi estymatorami gęstości jest histogram oraz jądrowy estymator gęstości. Każda z tych statystyk ma swoich zwolenników. Zaletą dystrybuanty empirycznej jest to, że nie wymaga podania żadnego dodatkowego parametru, zależy więc wyłącznie od danych. Zaletą histogramu jest jego popularność i fakt, że praktycznie każdy potrafi go poprawnie odczytać. Wadą jądrowego estymatora gęstości jest konieczność podania jądra oraz szerokości okna, które istotnie wpływają na wynikową gęstość. Estymator jądrowy charakteryzuje się niedobrym zachowaniem dla punktów skrajnych z próby. Szersza dyskusja na temat tych statystyk przedstawiona jest w rozdziale 3.1.2.

Do przygotowania wykresu histogramu w pakiecie `lattice` służy funkcja `histogram(lattice)`. Po prawej stronie formuły należy podać jaka zmienna ma zostać przedstawiona na wykresie. Ten wykres nie pozwala na wykorzystanie mechanizmu grupowania.

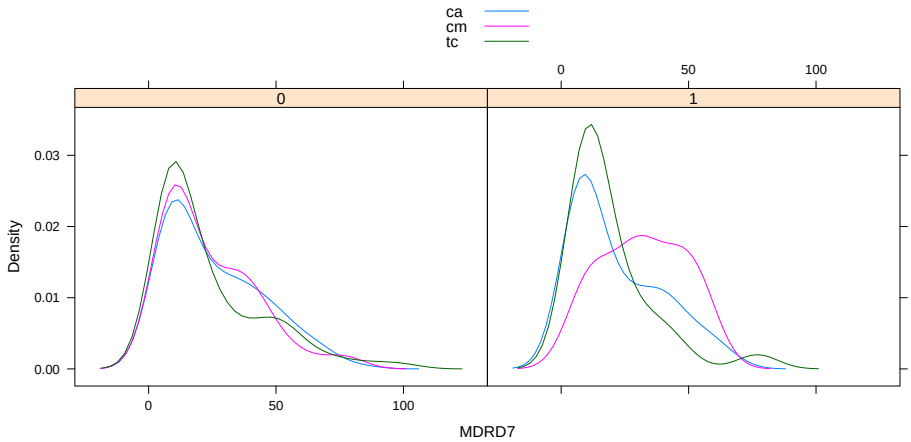
Do przygotowania wykresu gęstości jądrowej w pakiecie `lattice` służy funkcja `densityplot(lattice)`. Po prawej stronie formuły należy podać jaka zmienna ma zostać przedstawiona na wykresie. Parametry jądra można określać argumentami `bw` i `kernel`. Argument `plot.points` określa, czy na osi OX mają być zaznaczone współrzędne obserwacji, na podstawie których oceniono gęstość.



Rysunek 4.67: Przykładowe wywołanie funkcji qqmath().



Rysunek 4.68: Przykładowe wywołanie funkcji ecdfplot().



Rysunek 4.69: Przykładowe wywołanie funkcji densityplot().

Do przygotowania wykresu dystrybuanty empirycznej służy funkcja `ecdfplot(latticeExtra)`, która znajduje się w pakiecie `latticeExtra`. Po prawej stronie formuły należy podać, jaka zmienna ma zostać przedstawiona na wykresie.

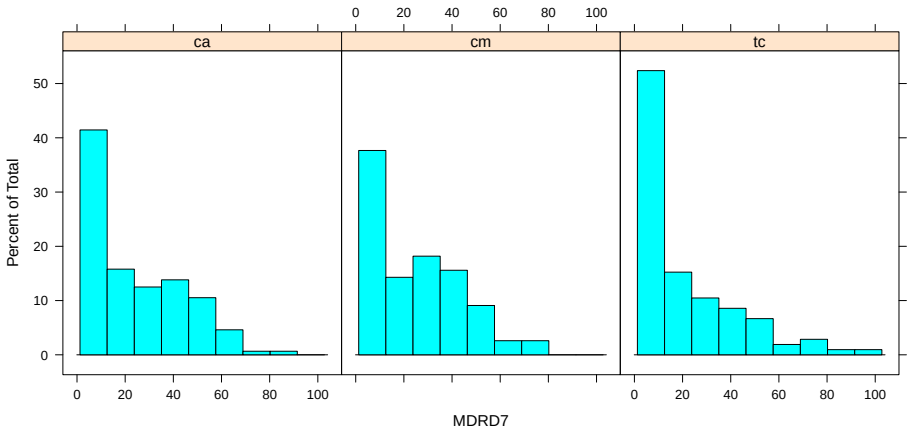
Funkcje `qq(lattice)` i `qqmath(lattice)` służą do konstrukcji wykresów kwantylowych. Funkcja `qqmath()` przygotowuje wykres przedstawiający zgodność z rozkładem zadanym przez argument `distribution`, domyślnie z rozkładem normalnym. Funkcja `qq()` przygotowuje wykres kwantylowy przedstawiający zgodność rozkładu zmiennej w dwóch grupach. Dla tej funkcji po prawej stronie formuły powinna być podana zmienna ilościowa, a po lewej stronie zmienna z dwoma poziomami, wyznaczająca podział na grupy.

Poniżej umieszczone są przykłady wywołania kolejnych funkcji, wyniki graficzne przedstawione są na wykresach 4.70, 4.71, 4.67, 4.69 i 4.68.

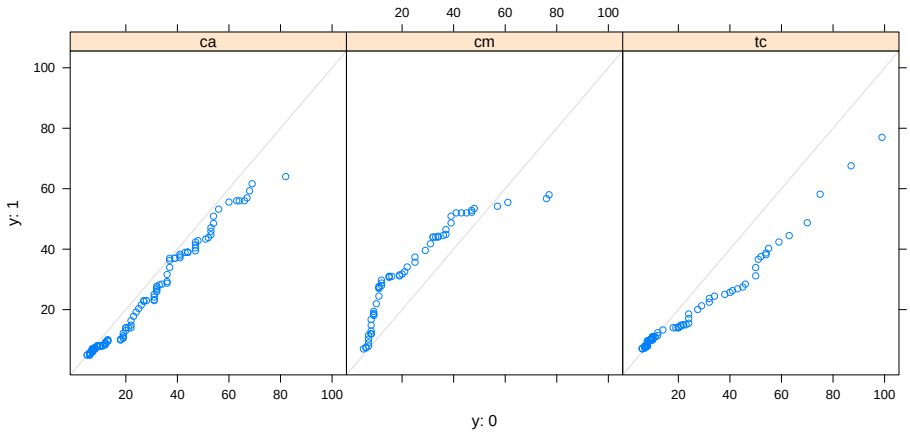
```
# histogram
histogram(~MDRD7|therapy, data=kidney)

# wykres kwantylowy badający zgodność z rozkładem normalnym
qq(diabetes~MDRD7|therapy,  distribution = qnorm, data=kidney)

# wykres kwantylowy cechy MDRD7 w grupach osób chorych na cukrzycę
    i zdrowych w podgrupach osób leczonych różnymi terapiami
qqmath(~MDRD7|factor(diabetes), groups=therapy, data=kidney)

# jądrowy estymator gęstości dla MDRD7 dla grup i warunkowania
densityplot(~MDRD7|factor(diabetes), groups=therapy, data=kidney,
    bw=8, plot.points="rug", auto.key=T)

# dystrybuanta empiryczna dla MDRD7 w podziale jak wyżej
library(latticeExtra)
ecdfplot(~MDRD7|factor(diabetes), groups=therapy, data=kidney)
```

4.3.12.6 Wykresy trójwymiarowe, funkcje: cloud(lattice), levelplot(lattice), contourplot(lattice), wireframe(lattice)

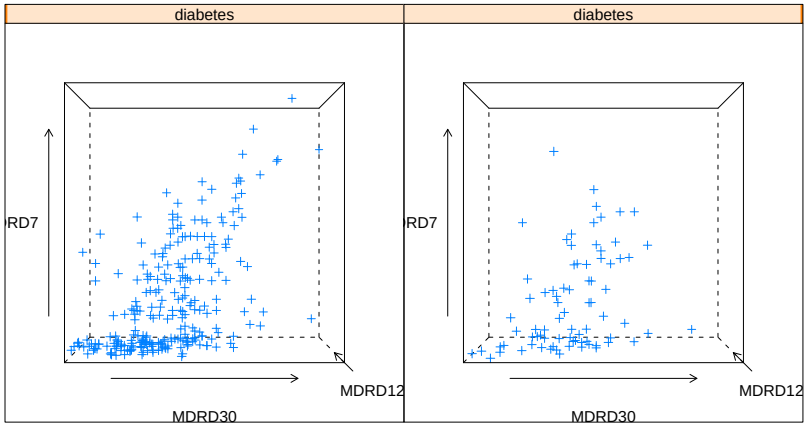
Wykorzystywanie złudzenia trzeciego wymiaru na dwuwymiarowym rysunku rzadko jest dobrym pomysłem, gdy chcemy czytelnie zaprezentować pewną informację. Nasz mózg zupełnie inaczej interpretuje głębokość niż szerokość czy wysokość. Jeżeli jednak chcemy przedstawić trzecią zmienną jako kolejny wymiar, to w pakiecie `lattice` do tego celu służą cztery funkcje `cloud(lattice)`, `levelplot(lattice)`, `contourplot(lattice)` i `wireframe(lattice)`.



Rysunek 4.70: Przykładowe wywołanie funkcji histogram().



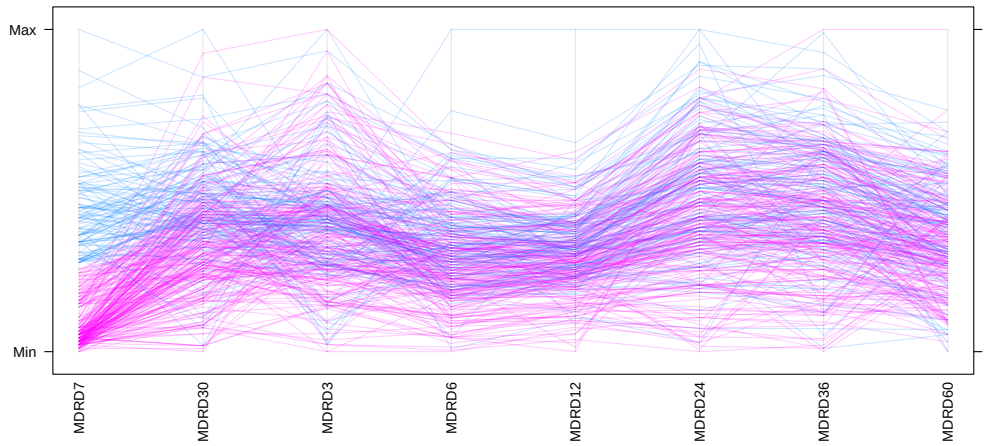
Rysunek 4.71: Przykładowe wywołanie funkcji qq().



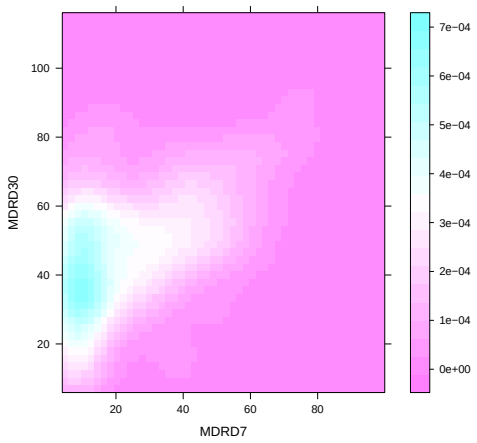
Rysunek 4.72: Przykładowe wywołanie funkcji cloud().

Tabela 4.5: Lista funkcji graficznych w pakiecie lattice.

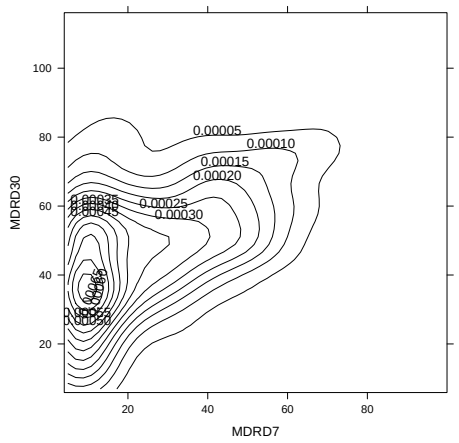
	Dla jednej zmiennej
`bwplot()`	Wykres pudełko-wąsy, tzw. boxplot, oraz wykresy pochodne, np. wiolinowy. Formuła powinna wskazywać dwie zmienne, jedną ilościową i jedną jakościową. Przykładowy wykres jest przedstawiony na rysunku 4.63.
`barchart()`	Wykres słupkowy.
`dotplot()`	Wykres kropkowy. Formuła powinna wskazywać dwie zmienne, jedną ilościową i jedną jakościową. Przykładowe wykresy są przedstawione na rysunkach 4.64 i 4.66.
`densityplot()`	Jądrowy estymator gęstości. Formuła powinna wskazywać jedną zmienną po prawej stronie. Przykładowy wykres jest na rysunku 4.69.
`histogram()`	Histogram. W przypadku tego wykresu nie można korzystać z mechanizmu grupowania. Formuła powinna wskazywać jedną zmienną po prawej stronie. Przykładowy wykres jest przedstawiony na rysunku 4.70.
`qqmath()`	Wykres kwantylowy dla jednej próby do badania zgodności z wybranym rozkładem. Formuła powinna wskazywać jedną zmienną po prawej stronie. Przykładowy wykres jest przedstawiony na rysunku 4.67.
`ecdfplot()`	Wykres dystrybuanty empirycznej. Formuła powinna wskazywać jedną zmienną po prawej stronie. Przykładowy wykres jest przedstawiony na rysunku 4.68.
`stripplot()`	Wykres paskowy. Przykładowy wykres jest na rysunku 4.65.
	Dla pary zmiennych
`xyplot()`	Wykres rozrzutu. Formuła powinna wskazywać dwie zmienne, obie ilościowe. Przykładowy wykres jest przedstawiony na rysunku 4.61.
`qq()`	Wykres kwantylowy dla dwóch prób. Formuła powinna wskazywać jedną zmienną po lewej i jedną po prawej stronie. Przykładowy wykres jest przedstawiony na rysunku 4.71.
	Dla trzech zmiennych
`contourplot()`	Wykres konturowy. Formuła powinna wskazywać trzy zmienne odpowiadające współrzędnym x, y i z trójwymiarowej siatki. Przykładowy wykres jest przedstawiony na rysunku 4.75.
`levelplot()`	Wykres typu mapa ciepła. Formuła powinna wskazywać trzy zmienne odpowiadające współrzędnym x, y i z trójwymiarowej siatki. Przykładowy wykres jest przedstawiony na rysunku 4.74.
`cloud()`	Trójwymiarowy wykres rozrzutu. Formuła powinna wskazywać trzy zmienne ilościowe. Przykładowy wykres jest na rysunku 4.72.
`wireframe()`	Powierzchnia 2D. Formuła powinna wskazywać trzy zmienne odpowiadające współrzędnym x, y i z trójwymiarowej siatki. Przykładowy wykres jest przedstawiony na rysunku 4.76.
	Dla wielu zmiennych
`parallel()`	Wykres przebiegu. Formuła po prawej stronie powinna wskazywać listę zmiennych ilościowej. Przykładowy wykres jest na rysunku 4.73.
`splom()`	Macierz wykresów rozrzutu. Formuła po prawej stronie powinna wskazywać listę zmiennych. Przykładowy wykres jest na rysunku 4.62.



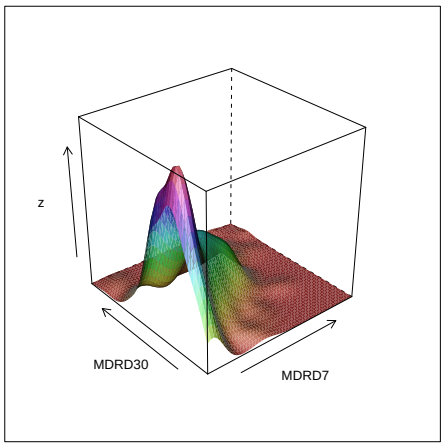
Rysunek 4.73: Przykładowe wywołanie funkcji parallel().



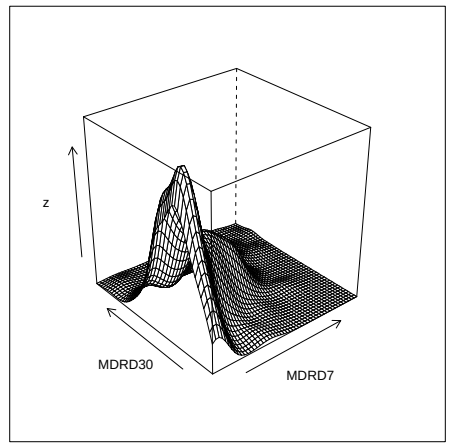
Rysunek 4.74: Przykładowe wywołanie funkcji levelplot().



Rysunek 4.75: Przykładowe wywołanie funkcji contourplot().



Rysunek 4.76: Przykładowe wywołanie funkcji wireframe().



Rysunek 4.77: Przykładowe wywołanie funkcji wireframe().

Funkcja `cloud()`, to trójwymiarowa wersja wykresu rozrzutu znanego z funkcji `xyplot()`. Po lewej stronie formuły wskazuje się zmienną, która ma zostać przedstawiona na osi Z, po prawej stronie wskazuje się zmienne, które mają być zaznaczone na osiach X i Y. Pozostałe argumenty tych funkcji pozwalają na włączanie i wyłączanie perspektywy, ustawianie położenia obserwatora itp. Więcej szczegółów można znaleźć w pomocy dla funkcji `panel.cloud()`. W przykładzie poniżej argument `screen` jest użyty do określenia położenia obserwatora.

Funkcje `levelplot()` i `contourplot()` służą do rysowania wykresów konturowych. Jako argument wejściowy należy podać trzy zmienne opisujące powierzchnię określoną na regularnej kracie. W formule po lewej stronie formuły podaje się zmienną określającej wysokości, po prawej stronie podaje się dwie zmienne określające współrzędne na równomiernej siatce. Obie funkcje można łatwo wykorzystać do przedstawiania dwuwymiarowego estymatora gęstości. Poniżej na przykładzie wyznaczany jest jądrowy dwuwymiarowy estymator gęstości z użyciem funkcji `kde2d(MASS)`.

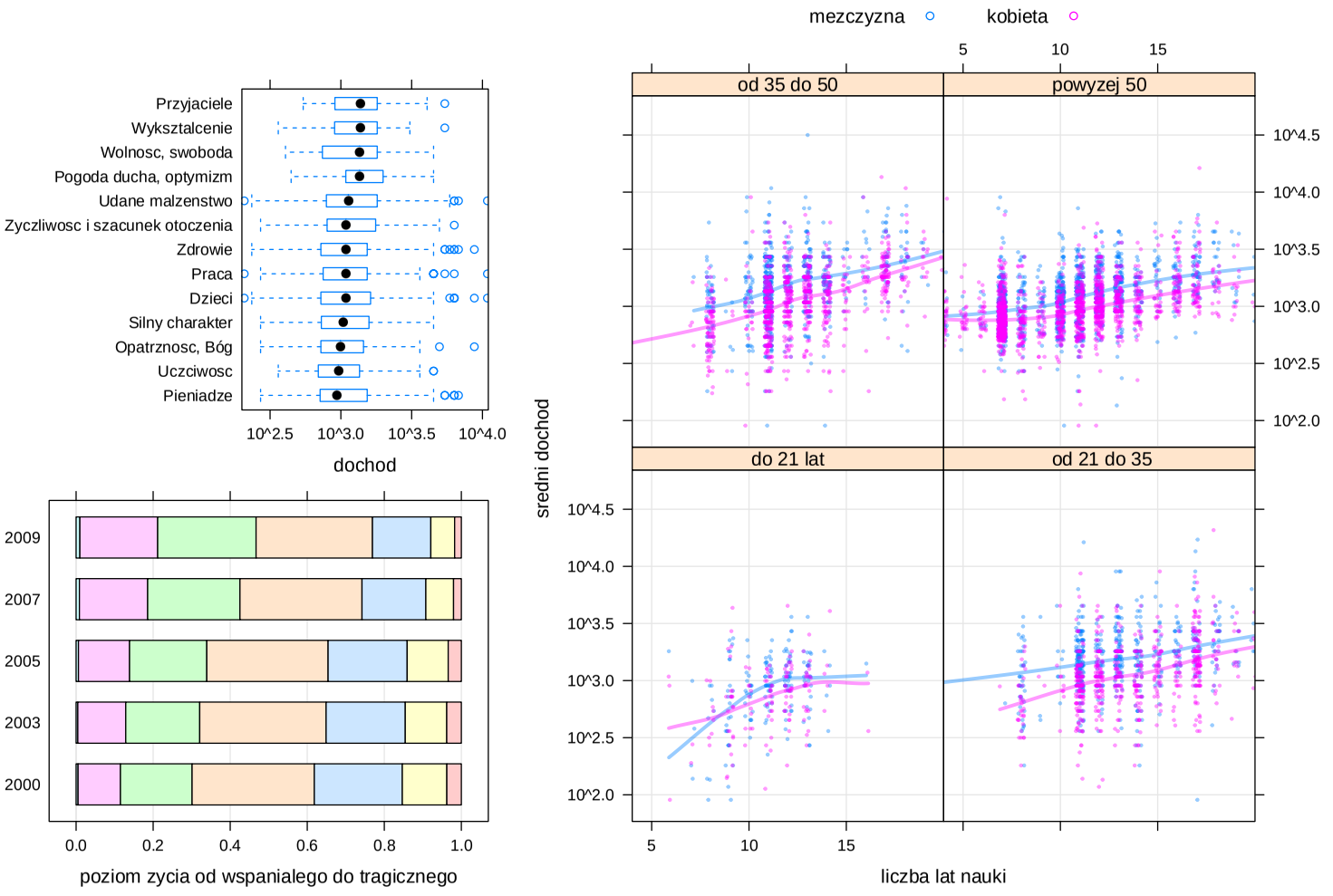
Funkcja `wireframe()` służy do rysowania powierzchni. Powierzchnia może być pokolorowana lub nie w zależności od wartości argumentu `shade`. Powierzchnię należy opisać za pomocą trzech zmiennych, z których dwie opisują regularną kratę, a trzecia wysokości na tej kracie (patrz opis dla funkcji `contourplot()` powyżej).

Poniżej umieszczone są przykłady wywołania opisanych funkcji. Wyniki tych wywołań przedstawione są na wykresach 4.72, 4.74, 4.75, 4.76 i 4.77.

```
library(MASS)
siatka = kde2d(kidney$MDRD7, kidney$MDRD30, n=50)
siatka = data.frame(expand.grid(MDRD7=siatka$x, MDRD30=siatka$y),
                    z=c(siatka$z))
# poniższe funkcje prezentują na różne sposoby powierzchnię opisaną
     przez wysokości punktów na regularnej kracie
levelplot(z~MDRD7*MDRD30, siatka, cuts=20, colorkey=T,region=T)
contourplot(z~MDRD7*MDRD30, siatka, cuts=20)
wireframe(z~MDRD7*MDRD30, siatka, cuts=20, shade=TRUE)
wireframe(z~MDRD7*MDRD30, siatka, cuts=20, shade=FALSE)
# funkcja cloud() prezentuje trójwymiarowy wykres rozrzutu
cloud(MDRD7~MDRD30+MDRD12|diabetes, data=kidney,
   screen = list(z = 0, x = -90))
```

4.3.13 Przykład użycia

Poniżej przedstawiamy przykład użycia pakietu `lattice` do danych z projektu diagnoza społeczna. Przykład ten ma pokazać jak zrobić kompletną wizualizację danych. Dane z projektu diagnoza społeczna są publicznie dostępne na stronie `http://www.diagnoza.com/`. W ramach dużego panelowego badania co mniej więcej dwa lata ankietowanych jest kilka tysięcy osób. Na bazie uzyskanych odpowiedzi ocenia się trendy w jakości życia Polaków.



Rysunek 4.78: Przykład użycia pakietu lattice, dane z projektu diagnoza społeczna. Pierwszy wykres przedstawia średni dochód ankietowanych w zależności od wartości, które są dla nich istotne. Najniższy średni dochód mają osoby dla których ważne są pieniądze lub uczciwość, najwyższy średni dochód miały osoby dla których ważni są przyjaciele, wykształcenie lub wolność. Kolejny wykres pokazuje, że zadowolenie z życia zwiększa się na przestrzeni kolejnych lat. Ostatni wykres przedstawia zależność dochodów od płci i liczby lat nauki. Liczba lat nauki i dochody korelują ze sobą dodatnio. Kobiety zarabiają mniej od mężczyzn, nawet jeżeli mają tyle samo lat nauki na koncie. Najwięcej zarabiają osoby w wieku 35-50 lat a najmniej osoby do 21 lat.

To bardzo ciekawy zbiór danych, zachęcam do ściągnięcia danych i eksperymentowania ze zmiennymi.

Używając tego zbioru danych przedstawimy przykład tworzenia grafiki z pakietem `lattice`.

```
# informacja co jest ważne dla ankietowanych osób i jak ta zmienna
    koreluje z wartością dochodu na głowę. Ankietowany mógł
    zaznaczyć odpowiedzi, wyniki w kolumnach ap2_a, ..., ap2_c.
coWazne = factor(c(as.character(dane$ap2_a), as.character(dane$ap2_
    b), as.character(dane$ap2_c)))
# ramka danych o trzech kolumnach, dochód w kolejnych trzech
    badaniach
dochod = c(dane$edoch_ind_5, dane$edoch_ind_5, dane$edoch_ind_5)

# zmieniamy kolejność poziomów w zmiennej coWazne, tak by ta
    kolejność odpowiadała kolejności średnich dochodów
coWazne2 = reorder(coWazne,dochod,median,na.rm=T)

# w tabeli tab będziemy przechowywali o liczebność osób
    deklarujących określony poziom jakości życia jako funkcję roku
    badania. Jest pięć lat w których przeprowadzano badanie i 7
    poziomów zmiennej opisującej jakość życia
tab = matrix(0,5,7)
# ta i kolejne linie wyznaczają liczebności poszczególnych poziomów
tab[1,] = table(factor(dane[,"ap46"]))
tab[2,] = table(factor(dane[,"bp38"]))
tab[3,] = table(factor(dane[,"cp38"]))
tab[4,] = table(factor(dane[,"dp38"]))
tab[5,] = table(factor(dane[,"ep34"]))
colnames(tab) = levels(factor(dane[,"bp38"]))
rownames(tab) = c("2000", "2003","2005","2007","2009")

# zależność dochodu od liczby lat nauki, w rozbiciu na wiek i na
    płeć. Zmienną dochód standardowo logarytmujemy z uwagi na
    asymetryczny charakter
wiek = cut(dane$wiek2007, c(0,21,35,50,100), c("do 21 lat", " od 21
     do 35", "od 35 do 50", "powyzej 50"))
wykres1 = xyplot(edoch_ind_5~jitter(lata_nauki_2007)|wiek, group=
    PLEC, data=dane, scale=list(y=list(log=10)), type=c("p","smooth
    ","g"), pch=19, cex=0.3, alpha=0.4, lwd=3, xlim=c(4,20), xlab="
    liczba lat nauki", ylab="sredni dochod", auto.key=list(space="
    top",columns=2))

# przygotowujemy wykres pudełkowy, modyfikujemy funkcję rysującą
    panel dodając siatkę pionowych linii
wykres2 = barchart( prop.table(tab, margin=1), panel = function
    (...) {
    panel.grid(h=0,v=-1)
    panel.barchart(...)}, xlab="poziom zycia od wspaniałego do
        tragicznego")

# wykres pokazujący zależność pomiędzy dochodem tym co się w życiu
    liczy dla ankietowanego a dochodem, wynik jest zapisywany do
    obiektu, więc nic nie jest tutaj rysowane do ekranu
```

```
wykres3 = bwplot(coWazne2~dochod, data=dane, scales=list(x=list  (
    log=10)), xlim=c(200,11000))

# używając argumentu position rozmieszczamy wyprodukowane wykres na
     całej przestrzeni obrazka
# prawy wykres zajmuje 60% powierzchni rysunku
plot(wykres1, position=c(0.4,0,1,1))
# pozostałe dwa wykresy umieszczamy po lewej stronie
plot(wykres2, position=c(0,0,0.4,0.5), newpage=FALSE)
plot(wykres3, position=c(0,0.45,0.4,0.95), newpage=FALSE)
```

4.4 Pakiet ggplot2

W tym rozdziale przedstawimy wprowadzenie do pakietu `ggplot2`. Pakiet `ggplot2` pozwala na obiektowy sposób tworzenia eleganckiej grafiki. Proces projektowania wykresu jest bardzo elastyczny i dobrze zaprojektowany. Wynikowa grafika jest bardzo czytelna i wygląda dobrze nawet przy domyślnych ustawieniach. W porównaniu z pakietem `lattice` tworzenie wykresu jest troch bardziej złożone, możliwości są jednak większe. Ponieważ pakiet `gplot2` i `graphics` bazują na różnych systemach graficznych, nie można łatwo łączyć funkcji z obu pakietów. Pakiet `ggplot2` podobnie jak i `lattice` wykorzystuje system graficzny implementowany przez pakiet `grid`.

Po lekturze tego rozdziału czytelnik będzie zaznajomiony z logiką tworzenia wykresów z użyciem pakietu `ggplot2`, będzie też potrafił samodzielnie przygotować i zmodyfikować wykres. Podobnie jednak jak w przypadku pakietu `lattice`, nie ma tutaj miejsca na kompletne przedstawienie wszystkich parametrów, opcji i możliwości pakietu `ggplot2`. Szczegółowe informacje zainteresowany czytelnik znajdzie w książce [40] lub na stronie internetowej [27].

W pakiecie `ggplot2`, podobnie jak w pakiecie `lattice`, tworzymy obiekt, który mapuje interesujące nas zmienne na właściwości wykresu, a następnie rysujemy ten obiekt używając przeciążonych funkcji `print()` lub `plot()`. Pakiety te różnią się filozofią tworzenia grafiki. W pakiecie `lattice` mamy dostępnych piętnaście funkcji, każda odpowiadająca pewnemu szablonowi przedstawiania danych. Tworząc wykres musimy wybrać szablon, a następnie możemy go modyfikować dodając lub zmieniając jego elementy. W pakiecie `ggplot2` używana jest inna logika tworzenia wykresów. Do tworzenia wykresów wykorzystuje się głównie funkcję `ggplot()`, za pomocą której określa się jakie znaczenie mają poszczególne zmienne i jak mają być zaprezentowane. To wystarczy by wygenerować wykres. Nie ma szablonów wykresów, które mają z góry narzuconą semantykę. Dowolna zmienna może zostać mapowana na dowolną właściwość. Funkcja `plot()` rysująca wykres wykona opisane mapowanie i narysuje wykres bez względu na to czy ma on sens czy nie. Taki sposób tworzenia wykresów jest na początku trudniejszy do opanowania, ale daje nieporównywalnie większe możliwości.

Będziemy używać tu określenia „mapowanie" które najlepiej oddaje sytuacje w której tworzona jest tzw. mapa, czyli lista par skojarzonych właściwości, w tym przypadku właściwość wykresu jest kojarzona ze zmienną ze zbioru danych.