Agenda

- Functions as return values
- Example: Linear predictor
- Example: the gradient operator
- Example: surface

Optional Recommended Reading: Chapter 3 of Chambers
Merely Useful Reading: Chapter 3 of the textbook
Functions are objects, just like everything else in R
Last time, saw how to pass functions in to other functions as arguments
This time, will look at how to create functions as return values
Mathematically

Something which takes a function in and gives a function back is an **operator**

Differentiation: the operator $d/dx$ takes $f$ and gives a new function

Gradient: the operator $\nabla$ takes $f$ and gives a new function

similarly $\nabla \cdot$, $\nabla \times$, ...

Indefinite integration: $\int_{-\infty}^{x} f(u) du$ takes $f$ and gives a new function
Functions are objects
The easiest way to create such an object is by calling `function`
The resulting function object has a body it executes, arguments, and
an environment it looks names up in
An admittedly trivial example

```r
make.noneuclidean <- function(ratio.to.diameter=pi) {
  circumference <- function(d) { return(ratio.to.diameter*d) }
  return(circumference)
}
```

Define `make.noneuclidean` but don’t run it yet

```r
> circumference(10)
Error: could not find function "circumference"
> kings.i <- make.noneuclidean(3)
> kings.i(10)
[1] 30
> formals(kings.i)
$d
> body(kings.i)
{
  return(ratio.to.diameter * d)
}
> environment(kings.i)
<environment: 0xe43d64>
> circumference(10)
Error: could not find function "circumference"
```
A Less Trivial Example

Create a linear predictor, based on sample values of two variables

```r
make.linear.predictor <- function(x,y) {
  linear.fit <- lm(y~x)
  predictor <- function(x) {
    return(predict(object=linear.fit,newdata=data.frame(x=x)))
  }
  return(predictor)
}
```

The predictor function persists and works, even when the data we used to create it is gone
Example:

```r
> independent.variable <- runif(10)
> dependent.variable <- 7 + 3*independent.variable
> my.predictor <- make.linear.predictor(x=independent.variable,
+   y=dependent.variable)
> rm(independent.variable,dependent.variable)
> independent.variable
Error: object 'independent.variable' not found
> dependent.variable
Error: object 'dependent.variable' not found
> my.predictor(5)
1
22
> my.predictor(runif(10))
  1   2   3   4   5   6   7   8
9   10
7.912262 8.125505
```
Last time wrote, `gradient` function to find $\nabla f$ at a point $x$

```r
nabla <- function(f,...) {
  g <- function(x,...) { gradient(f=f,x=x,...) }
  return(g)
}
```

Re-using the `mse.for.optimization` function from last time

```r
> mse.gradient <- nabla(mse.for.optimization)
> mse.gradient(c(6611,0.15),deriv.steps=c(1,1e-6))
[1] 1.646082e+05 1.428795e+10
> gradient(mse.for.optimization,c(6611,0.15),c(1,1e-6))
[1] 1.646082e+05 1.428795e+10
> gradient(mse.for.optimization,c(6611,0.15),c(1,1e-6),Y=2*gmp$pcgmp)
[1] -2.908638e+05 -2.486987e+10
> mse.gradient(c(6611,0.15),deriv.steps=c(1,1e-6),Y=2*gmp$pcgmp)
[1] -2.908638e+05 -2.486987e+10
```
Actually, as I said, the simple first-differences method is not so hot, so use the \texttt{grad} function from \texttt{numDeriv}

\begin{verbatim}
del <- function(f,...) {
  require(numDeriv)
  g <- function(x,...) { grad(func=f,x=x,...) }
  return(g)
}
\end{verbatim}

How would you check this?
Example: \textit{surface}

Last time: \textit{curve}, which takes an expression and, as a side-effect, plots a 1-D curve by sweeping over $x$

Suppose we want something like that but sweeping over two variables

Built-in plotting function \texttt{contour}:

\begin{verbatim}
contour(x, y, z, [[other stuff]])
\end{verbatim}

$x$ and $y$ are vectors of coordinates, $z$ is a matrix of the corresponding shape

(see \texttt{help(contour)} for graphical options)

Strategy: \textit{surface} should make $x$ and $y$ sequences, evaluate the expression at each combination to get $z$, and then call \texttt{contour}
First attempt at surface

Only works with vector-to-number functions:

```r
surface.0 <- function(f, from.x=0, to.x=1, from.y=0, to.y=1, n.x=101, n.y=101, ...) {
  x.seq <- seq(from=from.x, to=to.x, length.out=n.x)
  y.seq <- seq(from=from.y, to=to.y, length.out=n.y)
  plot.grid <- expand.grid(x=x.seq, y=y.seq)
  z.values <- apply(plot.grid, 1, f)
  z.matrix <- matrix(z.values, nrow=n.x)
  contour(x=x.seq, y=y.seq, z=z.matrix, ...)
  invisible(list(x=x.seq, y=y.seq, z=z.matrix))
}
```
surface.0(function(p){return(sum(p^3))}, from.x=-1, from.y=-1)
curve doesn’t require us to write a function every time — what’s its trick?
Expressions are just another class of R object, so they can be created and manipulated
One manipulation is evaluation

eval(expr, envir)

evaluates the expression expr in the environment envir, which can be a data frame or even just a list
When we type something like $x^2 + y^2$ as an argument to curve, R tries to evaluate it prematurely
substitute returns the unevaluated expression
curve uses first substitute(expr) and then eval(expr, envir), having made the right envir
Second attempt at *surface*

```r
surface.1 <- function(expr, from.x=0, to.x=1, from.y=0, to.y=1, n.x=101, n.y=101, ...) {
  x.seq <- seq(from=from.x, to=to.x, length.out=n.x)
  y.seq <- seq(from=from.y, to=to.y, length.out=n.y)
  plot.grid <- expand.grid(x=x.seq, y=y.seq)
  unevaluated.expression <- substitute(expr)
  z.values <- eval(unevaluated.expression, envir=plot.grid)
  z.matrix <- matrix(z.values, nrow=n.x)
  contour(x=x.seq, y=y.seq, z=z.matrix, ...)
  invisible(list(x=x.seq, y=y.seq, z=z.matrix))
}
```
surface.l(abs(x^3)+abs(y^3),from.x=-1,from.y=-1)
Evaluating a function at every combination of two arguments is a really common task. There is a function to do it for us: `outer` (seen in lecture 3)

```r
surface.2 <- function(expr, from.x=0, to.x=1, from.y=0, to.y=1, n.x=101, n.y=101, ...) {
  x.seq <- seq(from=from.x, to=to.x, length.out=n.x)
  y.seq <- seq(from=from.y, to=to.y, length.out=n.y)
  unevaluated.expression <- substitute(expr)
  z <- function(x,y) {
    return(eval(unevaluated.expression, envir=list(x=x, y=y)))
  }
  z.values <- outer(X=x.seq, Y=y.seq, FUN=z)
  z.matrix <- matrix(z.values, nrow=n.x)
  contour(x=x.seq, y=y.seq, z=z.matrix, ...)
  invisible(list(x=x.seq, y=y.seq, z=z.matrix))
}
```

could also include the function as part of the returned list
surface.2(x^4-y^4, from.x=-1, from.y=-1)
Summary

- Functions can be return values, just like any other object
- Variables other than the arguments to the function are fixed by the environment of creation
- Manipulation of expressions gives us ways of flexibly creating functions

Next week: The split/apply/combine trick