Introduction to computational programming

Chapter 1 Exercise

Functions and Algorithms

Solutions

David M. Rosenberg
University of Chicago
Committee on Neurobiology

Version control information:
Last changed date: 2009-10-12 19:11:18 -0500 (Mon, 12 Oct 2009)
Last changes revision:106
Version: Revision 106
Last changed by: David M. Rosenberg

October 17, 2009

1. Write a function that takes as input a polynomial[1] and returns the polynomial representing the first derivative of the input polynomial. You will be provided two functions, `parsePolynomial()` and `deparsePolynomial` to help you in this task. The function `parsePolynomial()` takes as input a polynomial in x and returns a vector of exponents and a vector of coefficients. The function `deparsePolynomial()` takes as input a vector of exponents and a vector of coefficients and returns a polynomial in x.

```
> .calcFirstDerivative <- function(coefs, expons) {
+   new_expons <- expons - 1;
+   new_expons <- new_expons[new_expons != -1]
+   new_coefs <- coefs * expons;
+   new_coefs <- new_coefs[new_expons != -1];
+   new_coefs <- new_coefs[1:length(new_expons)]
+   result <- list(coefs=new_coefs, expons=new_expons);
+ }
```

[1] Helper functions are included at the end of the exercise.

```
> deparsePolynomial <- function(coefficients, exponents) {
+   if (length(exponents) != length(coefficients)) {
+     stop('Exponent vector and coefficient vector must be of
    equal length.\n')
+   }
+   out_string <- '';
+   for (ii in seq(along=exponents)) {
+     out_string <- paste(out_string, ' + ', as.character(
    coefficients[ii]),
+                           'x^', as.character(exponents[ii]),
    sep='');
+   }
+   out_string <- gsub('\\+ -', '- ', out_string);
+   out_string <- gsub('1x', 'x', out_string);
+   out_string <- gsub(' x\\^0', ' 1', out_string);
+   out_string <- gsub('x\\^0', '', out_string);
+   out_string <- gsub('^ [\\+|-] ', '', out_string);
+   out_string <- gsub('x\\^1', 'x', out_string);
+   return(out_string);
+ }
```

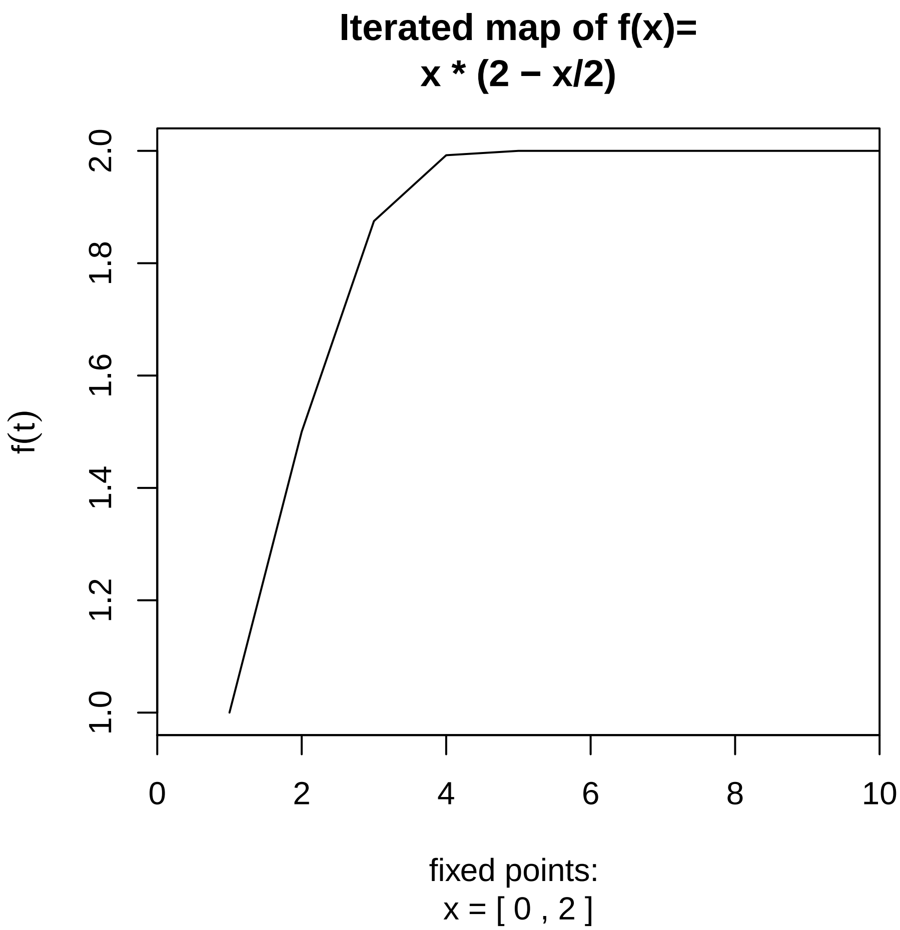
```
> parsePolynomial <- function(inputExpression) {
+   numberTerms <- nchar(gsub("[^x]", "", inputExpression));
+   coefficients <- numeric(length=(numberTerms)+1);
+   exponents <- numeric((length=numberTerms)+1);
+   splitTerms <- strsplit(gsub('-', '+ -', inputExpression), '
    \\+')[[1]]
+   for (ii in seq(along=splitTerms)) {
+     term <- gsub(' ', '', splitTerms[ii]);
+     if(gsub("[^x]", "", term) == '') {
+       term <- paste(term, 'x^0', sep='');
+     }
+     if (gsub("([\\+|-]*)(.*)x.*", "\\2", term) == '') {
+       term <- gsub('x', '1x', term);
+     }
+     if (gsub(".*x", "", term) == '') {
+       term <- paste(term, '^1', sep='');
+     }
+     coef <- eval(as.integer(gsub('x.*', '', term)));
+     coefficients[ii] <- coef;
+     exponent <- eval(as.integer(gsub('.*\\^(.*)', '\\1', term
    )));
+     exponents[ii] <- exponent;
+   }
+   o <- -(order(exponents)) + length(exponents) + 1;
+   exponents <- exponents[o];
+   coefficients <- coefficients[o];
+   result <- list(exponents=exponents, coefficients=
    coefficients);
+   return(result);
+ }
```

```
> calcPolyDeriv <- function(input_exp) {
+   parsed <- parsePolynomial(input_exp);
+   converted <- .calcFirstDerivative(parsed$coefficients,
    parsed$exponents);
+   result <- deparsePolynomial(converted$coefs, converted$
    expons);
+   return(result);
+ }
> calcPolyDeriv('3x^3 + 2x + 1');

 [1] "9x^2 + 2"

> calcPolyDeriv('5x^4 - 3x^3 + x^2 - 1');

 [1] "20x^3 - 9x^2 + 2x"
```

2. Write a *functor* which takes as an argument a function which is a logistic difference equation. Your functor should find the fixed points of the difference equation and plot its solution as an iterated map.

```
> source('http://rosenbergdm.uchicago.edu/maxima_utilities.R');
> plotImap <- function(f) {
+   f_zeros <- mSolve(paste(deparse(body(f)), " = 0", sep=''));
+   max_iter <- 10;
+   init_value <- f_zeros[1] + diff(f_zeros) / 4;
+   x_array <- numeric(length=max_iter);
+   x_array[1] <- init_value;
+   for (ii in 2:max_iter) {
+     x_array[ii] <- do.call(f, list(x_array[ii-1]));
+   }
+   fixed_points <- mSolve(paste(deparse(body(f)),
+     " = x", sep=''));
+   plot(1:max_iter, x_array, main=paste('Iterated map of f(x)
    =\n',
+         deparse( body(f) ), sep=''), type='l', xaxs='i',
+         xlim=c(0, max_iter), xlab='', ylab=expression(f(t)),
+         sub=paste(c("fixed points: \nx = [", fixed_points[1],
    ",",
+         fixed_points[2], "]"), collapse=" ") );
+ }
> f <- function(x)
+   x * (2 - x / 2)
```

```
> plotImap(f)
```



3. **This exercise was not assigned. The solution is included here for you to read if you like**. *Quicksort.* Recall exercise 2 from the previous section. There you sorted a list of integers using a method called a *bubble sort*. A more efficient method for sorting is the *quicksort*, given below in pseudocode.

```
function qsort (int array x[n])
    if n == 0 do
        return
    else if n == 1 do
        return x
    else
        int pivot
        int array head
        int array tail
        pivot ←  x[0]
        head ← [k] in x with k < pivot
        head ← qsort (head)
        tail ← [k] in x with k >= pivot
        tail ← qsort (tail)
        x ← join(head, pivot, tail)
    return x
```

Pseudocode listing 1 – Quicksort pseudocode.

```
> options(width=60);     # To make it fit on the page
> qSort <- function (x) {
+   if (length(x) < 2) {
+     return(x);
+   } else {
+     pivot <- x[1];
+     x <- x[-1];
+     head <- qSort(x[x < pivot]);
+     tail <- qSort(x[x >= pivot]);
+     return(c(head, pivot, tail));
+   }
+ }
> input_vector <- rnorm(n=20) * 100;
> sorted_vector <- qSort(input_vector);
> input_vector

 [1]    8.330124   23.579529   -2.182591 -250.919532
 [5]   42.035448 -150.132709  -60.302339  214.852173
 [9] -138.916182   45.045513  -47.803216   49.986170
[13]  136.510902  -55.176539   21.170415 -148.596700
[17]  -53.379635   33.240570  -34.297054  109.632461

> sorted_vector

 [1] -250.919532 -150.132709 -148.596700 -138.916182
 [5]  -60.302339  -55.176539  -53.379635  -47.803216
 [9]  -34.297054   -2.182591    8.330124   21.170415
[13]   23.579529   33.240570   42.035448   45.045513
[17]   49.986170  109.632461  136.510902  214.852173
```

4. **This exercise was not assigned. The solution is included here for you to read if you like.** *Efficiency.* Under ideal conditions, *quicksorting* a list of n integers involves calling `qsort` approximately $n \log n$ times. Under the worst conditions, `qsort` is called approximately n^2 times. What do these

conditions look like?

> Although it may seem counterintuitive, the *quicksort* is most efficient when its input is "disorganized" and is least efficient when operating on an already sorted array. The command `system.time()`, which measures how long a computation takes, is used in the following code snippet to demonstrate this difference in efficiency. The `options(expression=...)` command is necessary to keep R from crashing in the "worst case" scenario.

```
> options(expressions=500000);   # Can cause stack overflow
> inVec <- rnorm(n=1000);
> inVec2 <- qSort(inVec);
> system.time(qSort(inVec));

   user  system elapsed
  0.009   0.000   0.010

> system.time(qSort(inVec2));    # Might crash on some systems

   user  system elapsed
  0.207   0.016   0.284
```

5. Using the functor from the previous exercise, plot iterated logistic of the following models and analytically find the fixed points of the models. State whether the solution found computationally approaches the analytically calculated fixed points.

   (a) $f(x) = 2x(1-x)$

   (b) $f(x) = 4x(1-x)$

```
> options(width=78)
> layout(matrix(c(1,2), nrow=1));                  # Side-by-side plots
> plotImap(function(x) 2 * x * (1 - x));           # implicit function definition
> plotImap(function(x) 4 * x * (1 - x));
```