Numerical analysis labs

Contents

Lab 2 (Lab 1 doesn't exist): Numeric root finding

In this lab we will find roots numerically by implementing the newton's- and secant- method. We will test the implementations on the following functions:

$$x^4 - 5x^3 + 9x + 3 = 0, x \in [4, 6]$$
$$2x^2 + 5 - e^x = 0, x \in [3, 4]$$

We implement these functions below:

```
f1 <- function(x) {
  x^4 - 5*x^3 + 9*x + 3
}

f2 <- function(x) {
  2*x^2 + 5 - exp(x)
}
```

Newton's method

Formula: $x_{n+1} = x_n - \frac{f(x)}{f'(x)}$

Implementation:

```
#Needs f to have x as argument-name
newton <- function(f, tol, x0, maxiter = 1000) {
  df <- Deriv(f)
  next_x <- x0

  while ((maxiter != 0) & (abs(f(next_x)) > tol))  {
    next_x <-  next_x - f(next_x)/df(next_x)
    maxiter = maxiter - 1
  }

  next_x
}

newt_f1 <- map_dbl(4:6, ~newton(f1, 0.01, .x, 1000) %>% f1) %>% round(6)
newt_f2 <- map_dbl(c(3,3.5,4), ~newton(f2, 0.01, .x, 1000) %>% f2) %>% round(6)

data.frame(f1 = newt_f1, f2 = newt_f2) %>% knitr::kable()
```

f1	f2
0.004066	-0.000354
0.004066	-0.000012
0.000009	-0.000041

Secant method

Formula: $x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$

Implementation:

```
secant <- function(f, tol, inits, maxiter = 1000) {
  x_prev <- inits[1]
  x_next <- inits[2]

  root <- x_next
  while ((abs(f(root)) > tol) & (maxiter != 0))  {
    root <-  x_next - (x_next-x_prev)/(f(x_next)-f(x_prev)) * f(x_next)
   # print(root)
    if(is.nan(root)) stop("Division by zero")
    x_prev <- x_next
    x_next <- root
    maxiter = maxiter - 1
  }

  root
}

#secant(f1, 0.01, c(3,4), 1000) %>% f1
```

Lab 3: Numeric integration

The integration methods we will are trapezoidal and Simpson's rule. Both will be tested using the:

- function $\int_0^2 \sqrt{x}dx$
- interval partitions (10, 100, 1000)

Trapezoidal rule

Approx between points: $\frac{f(a)+f(b)}{2} \times \Delta x$

```
trap_formula <- function(f, a, b, dx) {
  dx * (f(a)+f(b))/2
}
```

Simpson's rule

```
simp_formula <- function(f, a, b, dx) {
  dx/6 * (f(a) + 4 * f((a+b)/2) + f(b))
}
```

Integral comparison

```
num_int <- function(f, int, n, int_formula) {
  a <- int[1]
  b <- int[2]
  dx <- (b-a)/n
  rectangles <- n+1
  integral <- 0

  b <- a+dx
  for (i in 1:rectangles) {
    integral <- integral + int_formula(f, a, b, dx)
    a <- b
    b <- b + dx
  }
  integral
}
```

```
int_interval <- c(0,2)
n <- c(10,100,1000)

x_2 <- function(x) x^2

data.frame(Divisions   = n,
           Simpsons_error_percent    = (integrate(x_2, 0,2)$value -
                                          map_dbl(n, ~ num_int(x_2, int_interval, .x, simp_formula))) /
           Trapezoidal_error_percent = (integrate(x_2, 0,2)$value -
                                          map_dbl(n, ~ num_int(x_2, int_interval, .x, trap_formula))) /
           Simpson_VS_Trapeziodal = ((integrate(x_2, 0,2)$value -
                                          map_dbl(n, ~ num_int(x_2, int_interval, .x, simp_formula))) /
             (integrate(x_2, 0,2)$value -
                                          map_dbl(n, ~ num_int(x_2, int_interval, .x, trap_formula))) /
  round(7) %>% knitr::kable()
```

Divisions	Simpsons_error_percent	Trapezoidal_error_percent	Simpson_VS_Trapeziodal
10	-33.1000000	-33.6500000	0.55000
100	-3.0301000	-3.0351500	0.00505
1000	-0.3003001	-0.3003502	0.00005

Lab 4: Basic Gaussian Elimination

Skipped due to poor xp/work-ratio.

Lab 5: Gaussian Elimination with partial pivoting

Same as Lab 4

Lab 6: QR-decomposition

We will implement a function for QR-decomposition and test it on the following matrix:

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 2 \\ 0 & 0 & 3 \end{pmatrix}$$

Finding solution manually

We need a result to test our implementation with, hence we'll calculate the solution by hand first.

We start with finding Q using gram-schmidt:

1. $u_1 = v_1 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$
2. $u_2 = v_2 - poj_{u_1} v_2 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$
3. $u_3 = v_3 - poj_{u_1} v_3 - poj_{u_2} v_3 = \begin{pmatrix} 0 \\ 0 \\ 3 \end{pmatrix}$

We convert the vectors into unit-vectors:

$$\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \frac{1}{3} \begin{pmatrix} 0 \\ 0 \\ 3 \end{pmatrix}$$

The above make up the columns of Q, which now allows us to calculate R by observing that:

$$A = QR \implies Q^t A = R = \begin{pmatrix} 1 & 1 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 3 \end{pmatrix}$$

QR - implemenation

We implement gram- schmidt below:

```r
A_1 <- data.frame(v1 = c(0,1,0), v2 = c(1,1,0), v3 = c(1,2,3)) %>% as.matrix()

# Projects v2 on v1
project <- function(v1, v2) {
  sum(v1*v2) / sqrt(sum(v1*v1)) * v1
}

gram <- function(A) {
  n_vectors <- ncol(A)

  for (i in 2:n_vectors) {
    u_i <- A[,i] #A[,i] is v_i

    for (j in 1:(i - 1)) {
      u_i <- u_i - project(A[,j], u_i)
    }

    A[, i] = u_i / sqrt(sum((u_i*u_i)))
  }
  A
}

gram(A_1)
```

```
##      v1 v2 v3
## [1,]  0  1  0
## [2,]  1  0  0
## [3,]  0  0  1
```

Now that we can get Q, we'll make a function that gives us Q and R

```r
get_qr <- function(A) {
  Q <- gram(A)
  R <- t(Q) %*% A
  list(Q,R)
}

A_1 %>% get_qr()
```

```
## [[1]]
##      v1 v2 v3
## [1,]  0  1  0
## [2,]  1  0  0
## [3,]  0  0  1
##
## [[2]]
##    v1 v2 v3
## v1  1  1  2
## v2  0  1  1
## v3  0  0  3
```

Lab 7: Iterative equation-solving & Gradient Descent Method

This lab will be divided into two parts. In the first part we'll solve system of equations by implementing a subroutine for the Jacobi-method. In the second part we'll minimize a function using Gradient Descent.

The Jacobi-method

This method reminds of the fix-point method for solving x = f(x). It solves for x_i for row i to get $x_i = f(X_{-i})$. Given a guess for X, we can now iterate the way towards a solution.

Seeing the above as $Dx = b - (L+U)x$, we can derive f(x):

$$Ax = (D+L+U)x = b \implies Dx = b - (L+U)x \iff x = D^{-1}(b - (L+U)x), x_{n+1} = D^{-1}(b - (L+U)x_n)$$

Before we implement the algorithm, we define the following system to try the solution on:

$$\begin{aligned} 5x_1 - 2x_2 + 3x_3 &= -1 \\ -3x_1 + 9x_2 + x_3 &= 2 \\ 2x_1 - x_2 - 7x_3 &= 3 \end{aligned}$$

We implement the above as follows:

```r
A_2 <- data.frame(v1 = c(5,-3,2), v2 = c(-2,9,-1), v3 = c(3,1,-7)) %>% as.matrix()
b <- matrix(c(-1,2,3))
```

Then we implement and test Jacobi's-method as follows:

```r
jacobi <- function(A, b, iters = 1000) {
  #Extract upper-triangular part of A
  upper <- A
  upper[lower.tri(upper,diag=TRUE)] <- 0

  #Extract lower-triangular part of A
  lower <- A
  lower[upper.tri(lower,diag=TRUE)] <- 0

  #Finds inverse of diagonal
  inv_diag_mtx <- A %>%
    #extract diag of a as vector
    diag() %>%
    #gets inverse of diagonal elements
    ifelse(. == 0, ., 1/.) %>%
    #creates the matrix
    diag()

  x <- b*0
  for (i in 1:iters) {
    x <-   inv_diag_mtx %*% (b - (lower + upper)%*%x)

  }
  x
}

data.frame(jacobi = jacobi(A_2,b), calculator = solve(A_2,b)) %>% knitr::kable()
```

	jacobi	calculator
v1	0.1861199	0.1861199
v2	0.3312303	0.3312303
v3	-0.4227129	-0.4227129

Gradient descent

Poor work/xp ratio

Lab 8: Error analysis

Theoretical i.e. no code/work here

Lab 9: Basic interpolation and ODE

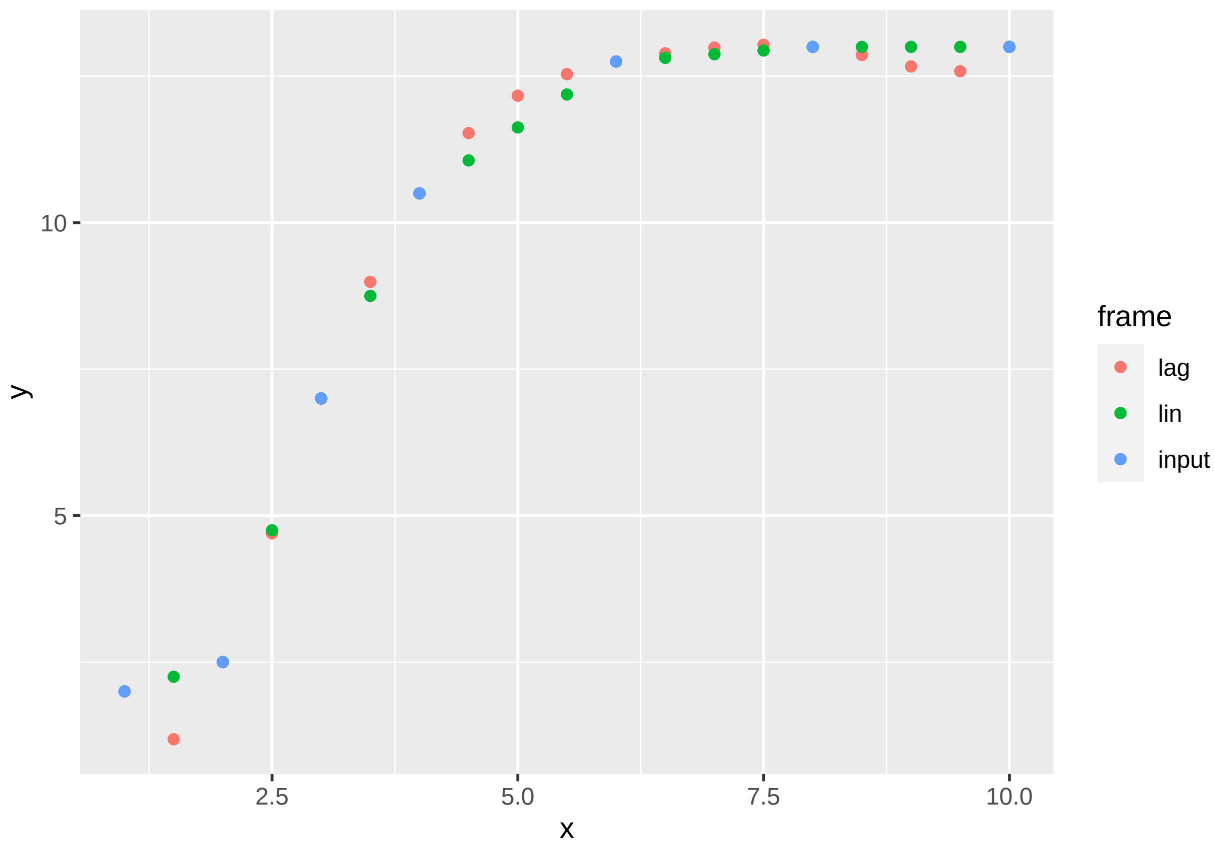
This lab will have one interpolation and one ODE section. In the interpolation section we'll implement linear and lagrange interpolation. In the ODE section we'll implement Euler's method.

Interpolation

```r
x <- c(1:4, 6, 8,10)
y <- c(2,2.5, 7, 10.5, 12.75, 13, 13)
to_pol <- list(x = x, y = y)

inter_line <- function(to_pol, inc) {
  x = to_pol$x
  y = to_pol$y

  get_line <- function(p1, p2) {
    m <- (p2[2]-p1[2])/(p2[1]-p1[1])
    function(x) p1[2] + m * (x - p1[1])
  }

  x_pol <- c(x[1])
  y_pol <- c(y[1])
  for (i in 1:(length(x)-1)) {
    p1 <- c(x[i], y[i])
    p2 <- c(x[i+1], y[i+1])
    line <- get_line(p1,p2)
    vals_for_line <- seq(p1[1], p2[1], inc)[-1]

    x_pol <- c(x_pol, vals_for_line)
    y_pol <- c(y_pol, line(vals_for_line))
  }

  y_pol
  list(x = x_pol, y = y_pol)
}

lin <- inter_line(to_pol, 0.5)
```

```r
#plot(seq(1,4,.5),inter_line(c(1,1), c(4,4), 0.5))
inter_lagrange <- function(to_pol, inc) {
  x <- to_pol$x
  y <- to_pol$y
  vals_to_f <- seq(x[1], tail(x,1), inc)

  #Helpfunction: Can create denom/num-erator for L_i
  f <- function(val, vector, index) {
    prod(val-vector[-index])
  }

  pol_f <- function(inter_request) {
    result <- 0

    for (i in 1:length(x)) {
      L_i <- f(inter_request, x, i)/f(x[i], x, i)

      result <- result + y[i] * L_i
    }
    result
  }

  x_int = vals_to_f
  y_int = map_dbl(vals_to_f, pol_f)
  list(x = x_int, y = y_int)
}

lag <- inter_lagrange(to_pol, 0.5)
```

```r
data.frame(x = lag$x,y = lag$y, frame = "lag") %>%
  bind_rows(data.frame(x = lin$x, y = lin$y, frame = "lin")) %>%
  bind_rows(data.frame(x, y, frame = "input")) %>%
  ggplot(aes(x = x, y = y, color = frame)) +
  geom_point()
```



Eulers method

Using Euler's method with step-sizes 0.1, 0.05 and 0.01 we will approximate the solution to the following IVP:

$$y' = y - xy(0) = \frac{1}{2}$$

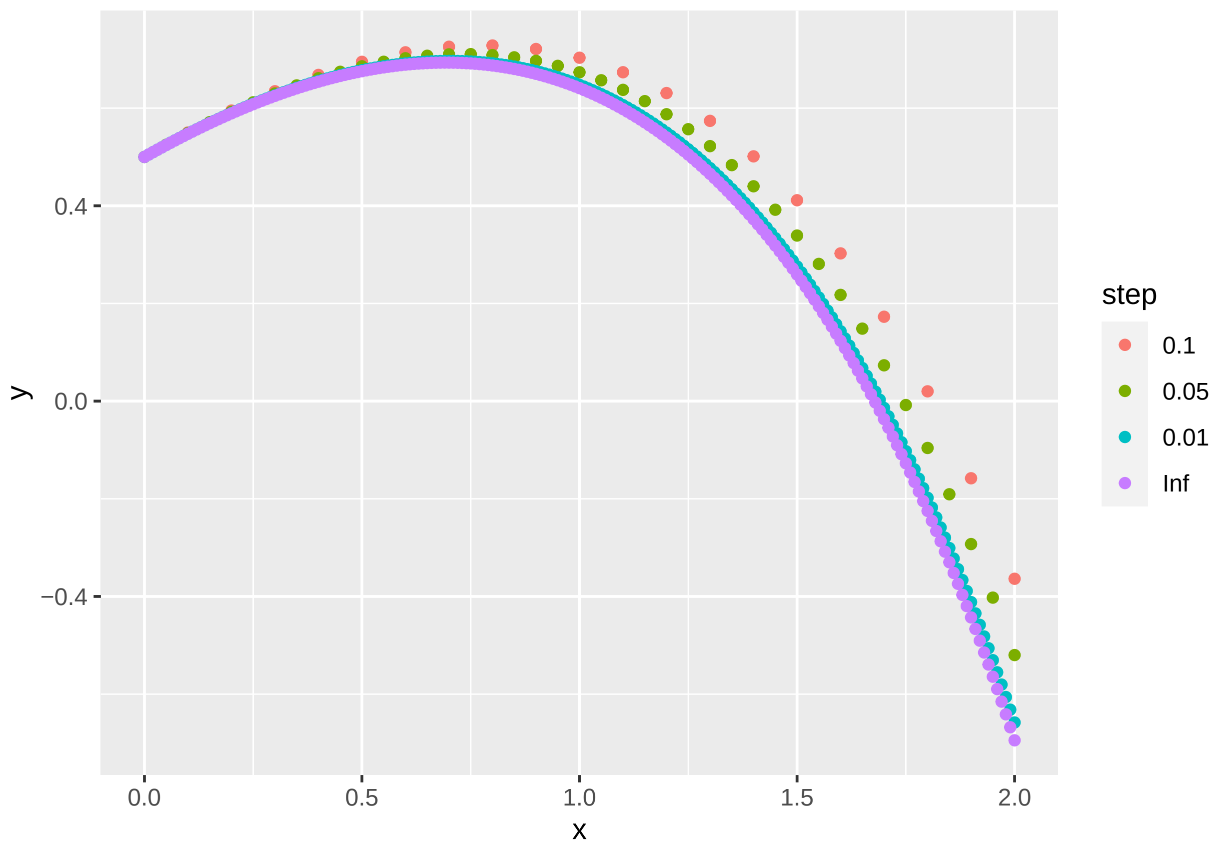
The results will be compared to the real solution $x + 1 - \frac{1}{2}e^x$.

First we implement the IVP:

```
dy <- function(x, y) {
  y-x
}

y_sol <- function(x) x + 1 - 1/2*exp(x)

y_0 <- 1/2
```

Next we implement the algorithm:

```
euler <- function(f, y_0, t_0, t_n, stepsize) {
  x <- c(t_0)
  y <- c(y_0)
  iters <- (t_n - t_0)/stepsize

  for (i in 1:iters) {
    y_next <- y[i] + stepsize * f(x[i], y[i])
    y <- c(y, y_next)
    x <- c(x, x[i] + stepsize)
  }

 list(x = x, y = y)
}

h_1 <- euler(dy, y_0, 0, 2, 0.1)
h_2 <- euler(dy, y_0, 0, 2, 0.05)
h_3 <- euler(dy, y_0, 0, 2, 0.01)
sol <- list(x = seq(0,2,0.01), y = y_sol(seq(0,2,0.01)))

data.frame(h_1, step = "0.1") %>%
  bind_rows(data.frame(h_2, step = "0.05")) %>%
  bind_rows(data.frame(h_3, step = "0.01")) %>%
  bind_rows(data.frame(sol, step = "Inf")) %>%
  ggplot(aes(x,y, color = step)) + geom_point()
```



Lab 10: Runge kutta and multistep methods

10.1

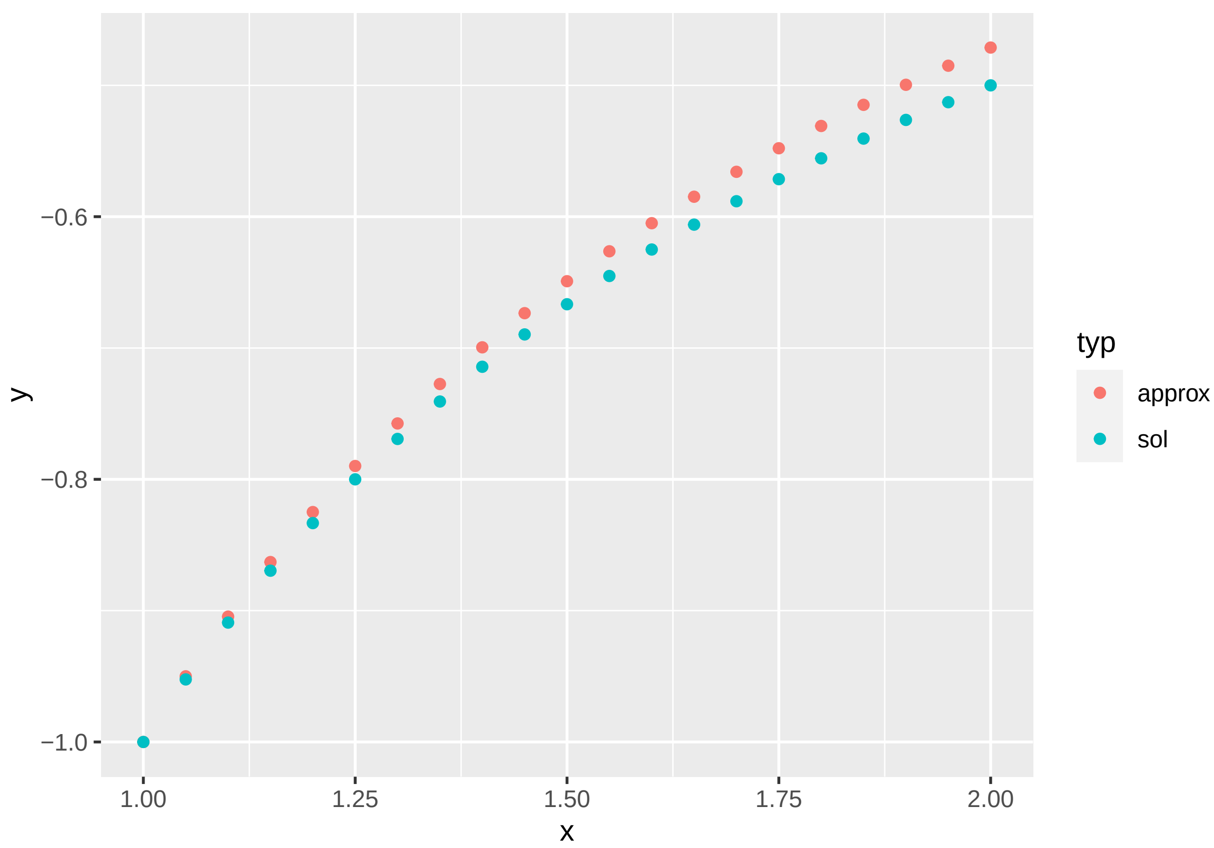
$$y' = \frac{1}{x^2} - \frac{y}{x} - y^2 1 \leq x \leq 2y(1) = -1, y = -1/x$$

We implement the above:

```
dy <- function(x, y) {
  1/x^2 - y/x - y^2
}

interval <- c(1, 2)

y1 <- -1

y <- function(x) {-1/x}
```

10.1.1: Using euler to approximate solution with h = 0.05.

```
approx <- euler(dy, y1, interval[1], interval[2], 0.05) %>%
  data.frame() %>%
  mutate(typ = "approx")
sol <- data.frame(x = seq(1,2,0.05), y = y(seq(1, 2, 0.05)), typ = "sol")
bind_rows(approx, sol) %>%
  ggplot(aes(x,y, color = typ)) + geom_point()
```



Did not do more due to similar problem again.

10.2 R4 to adams-bashforth/moulton

$$y' = y - x^2 y(0) = 1 \Delta x = 0.1, x \in [0, 3.3] y = 2 + 2x + x^2 - e^x$$

```
dy <- function(x, y) y - x^2
y0 <- 1
inter <- c(0, 3.3)
y <- function(x) 2 + 2*x + x^2 - exp(x)
```

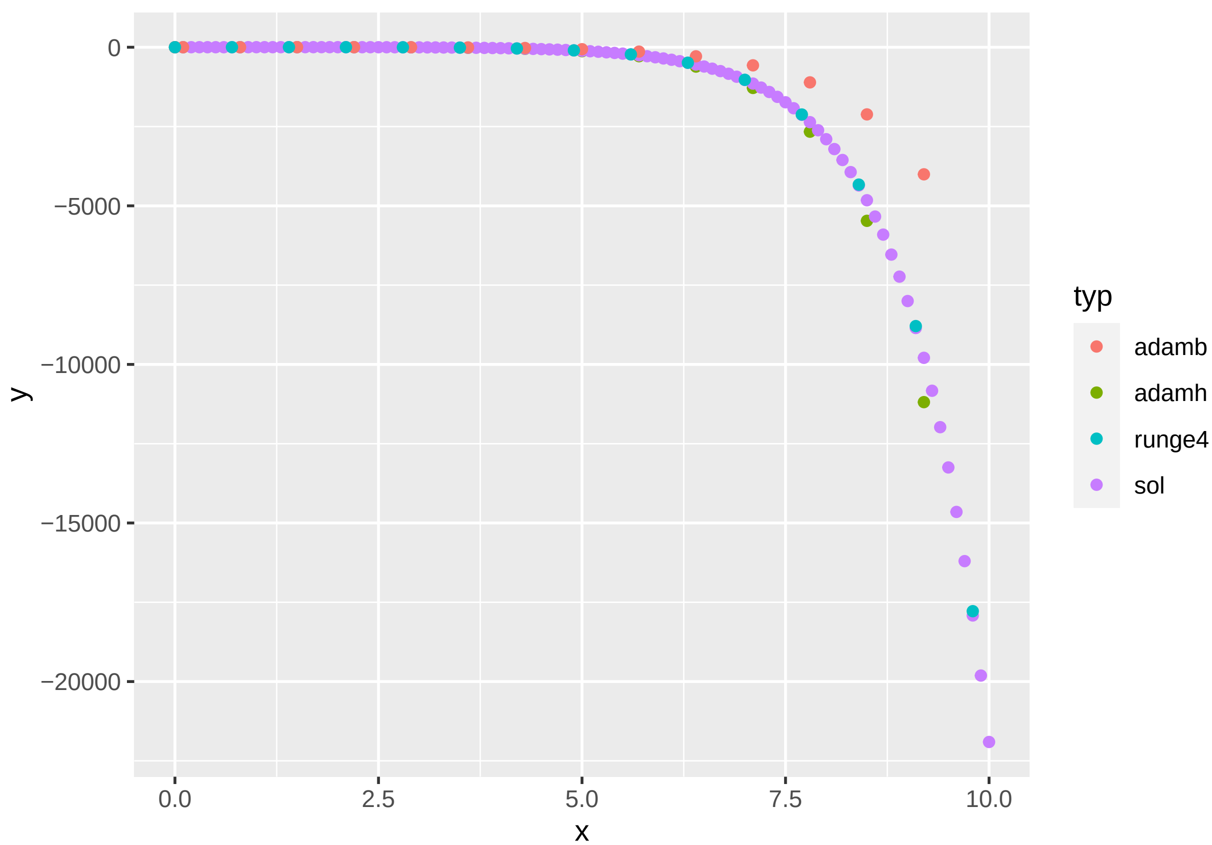
Adam basshfourth:

$$y_{i+1} = y_i + h[\frac{3f(x_i, y_i)}{2} - \frac{f(x_{i-1}, y_{i-1})}{2}]$$

```
adam_b <- function(f, x_inits, y_inits, x_n, stepsize) {
  x <- x_inits
  y <- y_inits
  iters <- (x_n - x[1])/stepsize
  for (i in 2:iters) {
    y_next <- y[i] + stepsize * (3/2*f(x[i], y[i]) - 1/2*f(x[i-1], y[i-1]))
    y <- c(y, y_next)
    x <- c(x, x[i] + stepsize)
  }

 list(x = x, y = y)
}

adam_h <- function(x_inits, y_inits, x_n, stepsize) {
  x <- x_inits
  y <- y_inits
  iters <- (x_n - x[1])/stepsize

  for (i in 2:iters) {
    x <- c(x, x[i] + stepsize)
    y_next <- 12/(12-5* stepsize) * (y[i] + stepsize/12 * (
      (-5 * x[i + 1]^2) +
        8*dy(x[i], y[i]) -
        dy(x[i-1], y[i-1])) )
    y <- c(y, y_next)
  }

 list(x = x, y = y)
}

r4 <- function(dy, x_inits, y_inits, x_n, stepsize) {
  x <- x_inits
  y <- y_inits
  fourth <- function(x, y, h, dy) {
    k1 <- dy(x, y)
    k2 <- dy(x + h/2, y + h/2 * k1)
    k3 <- dy(x + h/2, y + h/2 * k2)
    k4 <- dy(x + h, y + h * k3)
    1/6 * (k1 + 2 * k2 + 2 * k3 + k4)
  }
```

```
  iters <- (x_n - x[1])/stepsize
  for (i in 1:iters) {
    y_next <- y[i] + h * fourth(x[i], y[i], stepsize, dy)
    y <- c(y, y_next)
    x <- c(x, x[i] + stepsize)
  }

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

```
x_n <- 10
h <- 0.7
adamb <- adam_b(dy, c(0,0.1), c(y(0), y(0.1)), x_n, h) %>% data.frame() %>% mutate(typ = "adamb")
adamh <- adam_h(c(0,0.1), c(y(0), y(0.1)), x_n, h) %>% data.frame() %>% mutate(typ = "adamh")
runge4 <- r4(dy, 0, 1, x_n, h) %>% data.frame() %>% mutate(typ = "runge4")
sol <- data.frame(x = seq(0, x_n, 0.1), y = y(seq(0, x_n, 0.1)), typ = "sol")

bind_rows(adamh, sol) %>%
  bind_rows(adamb) %>%
  bind_rows(runge4) %>%
  ggplot(aes(x,y, color = typ)) + geom_point()
```



Lab 11: Boundary value problem (BVP)

In this lab we'll display the finitie difference and shooting method on the following BVP:

$$y''(x) = \frac{3}{2}y(x)x(0) = 4, x(1) = 1$$

```
ddy <- function(x, y, dy) 3/2 * y
x_init <- c(0, 1)
y_init <- c(4, 1)
```

11.1: Finite difference method

The idea here is to express y" using an approximation of y" and then create a system of equations using the boundary conditions.

The finite difference formula is:

$$\frac{y(x+h) - 2y(x) + y(x-h)}{h^2} = \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} = \frac{3}{2}y_i$$

To create a system of equations we rewrite the above to:

$$y_{i+1} - \frac{3h^2+4}{2}y_i + y_{i-1} = 0$$

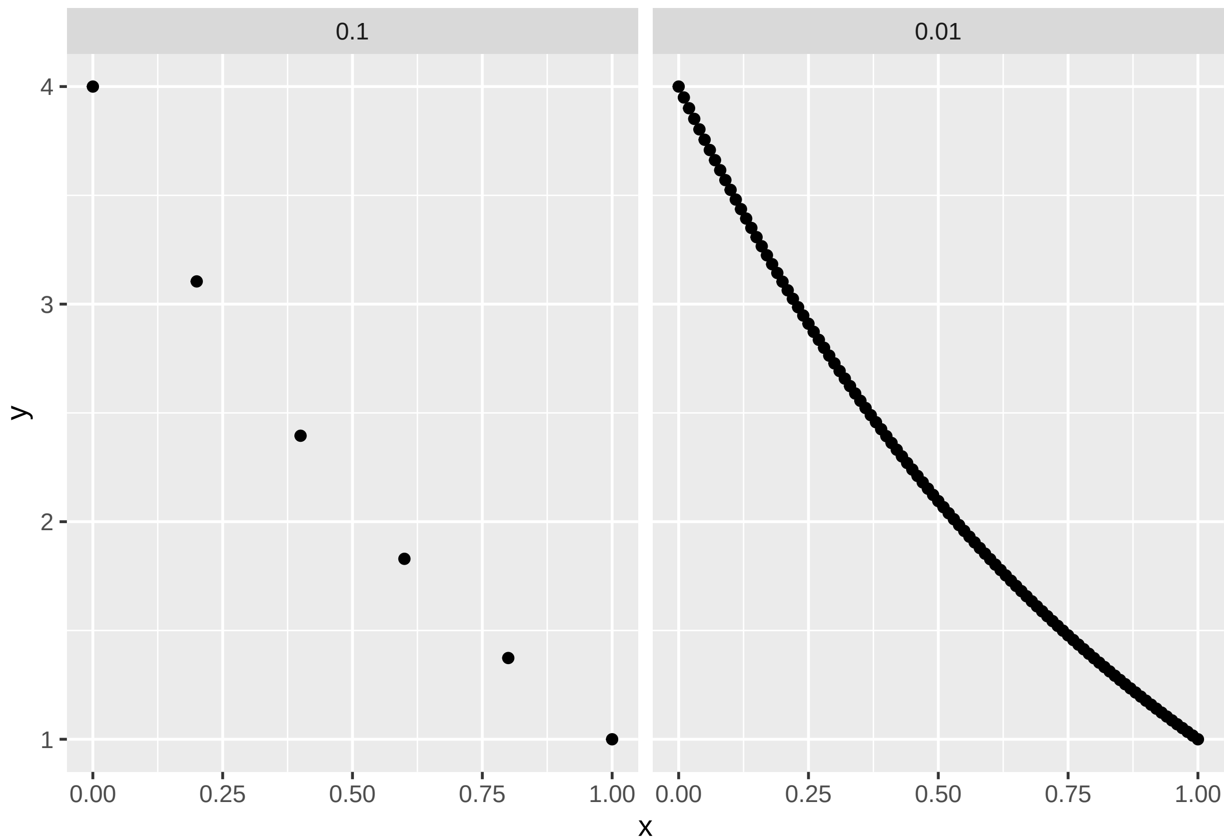
Using the boundary conditions we can now create the following system:

$$\begin{bmatrix} -\frac{3h+1}{2} & 1 & 0 & 0 & . & . & 0 & 0 \\ 1 & -\frac{3h+1}{2} & 1 & 0 & . & . & 0 & 0 \\ . & . & . & . & . & . & . & . \\ 0 & 0 & . & . & . & 0 & 1 & -\frac{3h+1}{2} \end{bmatrix} \times \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_n \end{bmatrix} = \begin{bmatrix} -4 \\ 0 \\ . \\ . \\ -1 \end{bmatrix}$$

```
gen_mtx <- function(a, b, h) {
  n <- (b-a)/h -1
  mtx <- matrix(rep(0, n^2), nrow = n, ncol = n)
  h_f <- function(h) -(3*h^2+4)/2
  mtx[1, 1:2] <- c(h_f(h), 1)
  mtx[n, (n-1):n] <- c(1, h_f(h))

  for (i in 2:(n - 1)) {
    mtx[i, c(i-1, i, i+1)] <- c(1, h_f(h), 1)
  }
  mtx
}
gen_mtx(0,1, 0.1)
```

```
##        [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]   [,9]
##  [1,] -2.015  1.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
##  [2,]  1.000 -2.015  1.000  0.000  0.000  0.000  0.000  0.000  0.000
##  [3,]  0.000  1.000 -2.015  1.000  0.000  0.000  0.000  0.000  0.000
##  [4,]  0.000  0.000  1.000 -2.015  1.000  0.000  0.000  0.000  0.000
##  [5,]  0.000  0.000  0.000  1.000 -2.015  1.000  0.000  0.000  0.000
##  [6,]  0.000  0.000  0.000  0.000  1.000 -2.015  1.000  0.000  0.000
##  [7,]  0.000  0.000  0.000  0.000  0.000  1.000 -2.015  1.000  0.000
##  [8,]  0.000  0.000  0.000  0.000  0.000  0.000  1.000 -2.015  1.000
##  [9,]  0.000  0.000  0.000  0.000  0.000  0.000  0.000  1.000 -2.015
```

```r
f_d <- function(x1, x2, h) {

  A <- gen_mtx(x1, x2, h)

  n <- nrow(A)
  b <- matrix(rep(0, n), nrow = n, ncol = 1)
  b[c(1,n),1] <-  c(-4, -1)

  list(x = seq(x1, x2, h), y = c(4, solve(A,b), 1))
}

data.frame(f_d(0,1,0.2), h = "0.1") %>%
  bind_rows(data.frame(f_d(0,1,0.01), h = "0.01")) %>%
  ggplot(aes(x,y)) + geom_point() + facet_wrap(~h)
```



11.2 Shooting method

This method is based on converting the BVP into a IVP by setting y' = z, giving z' = y". * Both assignments above result in IVPs where we solve for y and z respectively as a system. * To start the system we guess the initial value for $z(a) = s_i$. The system is a function of $s_i : f(s_i)$. * We now want the system to intersect the given/correct y(b) and can hence set up an equation $f(s_i) - y(b) = 0$. * We will use the secand method to solve that system.

```
shooting_f <- function(s, h, y_0 = 4, y_n = 1, x_0 = 0, x_n = 1, dz = ddy) {
  n <- (x_n - x_0)/h

  x <- c(x_0)
  y <- c(y_0)
  z <- c(s)
  for (i in 1:n) {
    x_next <- x[i] + h
    y_next <- y[i] + h * z[i]
    z_next <- z[i] + h * dz(x[i], y[i], dy(x[i], y[i])) #dy not used in this task, only there to be gen

    x <- c(x, x_next)
    y <- c(y, y_next)
    z <- c(z, z_next)
  }
  test_val <- y[n] - y_n
  list(val = test_val, x = x, y = y)
}
shooting_f(1, .1)
```

```
## $val
## [1] 6.303383
##
## $x
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
##
## $y
##  [1] 4.000000 4.100000 4.260000 4.481500 4.766900 5.119522 5.543648 6.044567
##  [9] 6.628641 7.303383 8.077555
```

```
# Here we chose secant, however, any equation solver may be used.
shooting_method <- function(f = shooting_f, h = 0.1, tol = 0.001, inits = c(0, 1), maxiter = 10) {
  z_prev <- inits[1]
  z_next <- inits[2]

  root <- z_next

  shoot_prev <- f(z_prev, h)
  shoot_next <- f(z_next, h)
  f_prev <- shoot_prev$val
  f_next <- shoot_next$val

  while ((abs(f_next) > tol) & (maxiter != 0))  {

    root <-  z_next - (z_next-z_prev)/(f_next-f_prev) * f_next
    if(is.nan(root)) stop("Division by zero")

    z_prev <- z_next
```

```
    z_next <- root

    shoot_prev <- f(z_prev, h)
    shoot_next <- f(z_next, h)
    f_prev <- shoot_prev$val
    f_next <- shoot_next$val

    maxiter = maxiter - 1
  }

  shoot_next
}

shooting_method() %>% plot()
```