PROJECT EUCLID

This project measures the time it takes to calculate the distance between any two point(rows) in a n-dimensional matrix in R and Python, and extending with C and C++.

R

In R, we compare times with the base R function utils::dist . Here are the five functions used

1. dist: This is the base R function. However it is technically a C function except that the output is formatted for readability
2. eucdist_R: This function uses vectorization in R to calculate distance between any two n-dimensional vectors
3. dotprod: This function uses dot product to calculate distance between any two n-dimensional vectors. It uses pure R
4. eucdist_C: This fucntion uses a C wrapper in R to calculate distances for a given matrix
5. fastdist2: This uses a C++ function loaded into R using Rcpp package

Note that 2 and 3 takes vectors as input. They can be called on every combinations of rows in an n-dimensianal matrix using the euc_func function below. The ouputs can be formatted as a "dist" object using the function mefa::vec2dist for easy viewing. To calculate the distances for a matrix, euc_func takes two arguments: the matrix and the function to be used.

In [1]:
```
%load_ext rpy2.ipython
```

In [2]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [3]:
```
%%R

# This function uses one of the above method to calculate distance between
euc_func <- function(dist_func, M){
  # takes a distance calculating function as a string (for two points)
  # function dist_func and a matrix M
  # returns the distance between rows of matrix M
  f <- match.fun(dist_func) # as a function
  n <- nrow(M)
  l <- n*(n-1)/2
  last_element <- f(M[n,], M[n-1, ])
  distance <- numeric(l)
  distance[l] <- last_element
  k <- 1 # for indexing the distance vector
  for(i in 1:(n-1)){

    A = M[i, ]
    # removing the column containing A
    for(j in (i + 1):n){
      #print(j)
      B <- M[j, ]
      distance[k] <- f(A, B)
      k <- k + 1

    }

  }
#   return( mefa::vec2dist(distance, n)  ) # vect2dist formats the output
    return( distance )

}

```

The code for the dot product method(3) and vectorization method

Dot Product

In [5]:
```
%%R
# (1) Dot Product
dot_prod_dist <- function(A, B){
  # function for computing euclidean distances using dot-product
  # Arguements : A and B are vectors of the same length
  # Outputs: The Euclidean distance between A and B


  # checking if the vectors are in the same dimension
  #if(length(A) != length(B)) stop('Invalid input(s)')

  u <- A - B
  return(sqrt(sum(u*u)))
}

```

Vectorization

In [6]:
```
%%R
#(2) using vectorization in R
e_dist <- function(A, B){
  # calculating the distance between vectors A and B
  #if(length(A) != length(B)) stop('Invalid Input(s)')

  return( sqrt(sum((A - B)^2)) )

}
```

The C and C++ function uses the same loop with the calculation done using a loop

The C Function

The C function is in a file called eucdist.c. After compiling, the .so is loaded using dyn.load and then used in an R wrapper below

In [3]:
```
%%bash
cat C C++/eucdist.c
```

```
#include <math.h>
#include <R.h>
#include <Rinternals.h>
#include <stdio.h>
/* Euclidean distance */
/*q=.C("eucdist",as.integer(c(1,2,3,4,5,6,7,8)),as.integer(4),as.integer(2),a
s.double(vector("double",6))*/
void eucdist(double *x, int *m, int *n, double *d)
{
  /* Arguement:
    1.  x is a matrix of dimension n by m
    2.  m is the number of rows
    3.  n is the number of coloums
    4.  d is the pointer for output */
  /*
   d = sqrt(sum((XI-XJ).^2,2));            % Euclidean
   */
  int i,j,k; /* **pointer; /* Indexers */
  int local_m, local_n;
  local_m = *m, local_n = *n;
  double theSum; /* size_t is an unsigned integer of size 16 bits */
  /*
  XI for indexing rows
  XJ for indexing columns
  XI0 unknown for now
  */
  int  XI, XJ, XI0, index; /* pointers as row indexers*/
  // d = malloc( local_m*(local_m - 1)/2);
  // XI0 =  (double *) x; /* we are not tuching x but using its memory address
as XI */
  // x = (double) x;
  index = 0;

  for (i=0; i<local_m-1; ++i) { /
    XI = i*local_n; //indexing the start of the i the row
    XI0 = XI;
    // Rprintf("XI is %d\n", XI);
    for (j=i+1; j<local_m; ++j) {
      XJ = j*local_n; // indexing the start of the start jth row
      // Rprintf("XJ is %d\n", XJ);
      // XI = XI0; /* Index? */
      theSum = 0.0;
      for (k=0;k<local_n;k++,++XI,++XJ){
        theSum += pow((x[XI]- x[XJ]), 2.0);
      }
      XI = XI0;
      d[index++] = sqrt(theSum);
    }
  }
}
```

In [7]:
```
%%R
eucdist_C <- function(M){
#     nrows = as.integer(nrow(M))
    # M is a matrix
    out <- .C("eucdist",
        x = as.vector(t(M), "double"),
        m = nrows,
        n = as.integer(ncol(M)),
        d = as.double(vector("double", nrow(M)*(nrow(M)-1)/2))
    )
    #return(vec2dist(as.vector(out$d), nrows))
    return(out$d)
}

```

The C++ Function

In [8]:
```
%%bash
cd /home/medfad/Desktop/WD/2019/Abdul/Project_Euclid
cat C C++/fastdist2.cpp
```

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector fastdist2 (const NumericMatrix & x){
  unsigned int outrows = x.nrow(), i = 0, j = 0,k=-1;
  Rcpp::NumericVector out(outrows*(outrows-1)/2); /* empty matrix */

  for (i = 0; i <= outrows-2; i++){
    for (j = i + 1; j <= outrows-1 ; j ++){
      k++;
      out(k) = sqrt(sum(pow(x.row(i)-x.row(j), 2.0)));
    }
  }
  // out(outrows*(outrows-1)/2) = sqrt(sum(pow(x.row(outrows)-x.row(outrows-
1), 2.0)));
  return out;
}
```

The function is sourced using Rcpp::sourceCpp

In [9]:
```
 %%R
Rcpp::sourceCpp("C C++/fastdist2.cpp")
```

Testing for Consistency

A matrix of 1000 rows is used to check that the functions produced the same output with a seed of 1 for consistency. All of the function used are declared/sourced from in the R script functions.R

In [10]:
```
%%R
set.seed(1)
M <- matrix(rnorm(2000, 10, 10), ncol = 2)
head(M)
```

```
          [,1]      [,2]
[1,]  3.735462 21.349651
[2,] 11.836433 21.119318
[3,]  1.643714  1.292224
[4,] 25.952808 12.107316
[5,] 13.295078 10.693956
[6,]  1.795316 -6.626489
```

In [11]:
```
%%bash
R CMD SHLIB C C++/eucdist.c
```

```
make: Nothing to be done for 'all'.
```

In [12]:
```
%%R

dyn.load("C C++/eucdist.so")
```

In [13]:
```
%%R
source("/home/medfad/Desktop/WD/2019/Abdul/Project Euclid/R code/functions.
```

In [14]:
```
%%R
d1 <- dotprod(M)
d2 <- eucdist_C(M)
d3 <- eucdist_R(M)
d4 <- as.vector(dist(M)) # this output is format as an upper triangular mat
d5 <- fastdist2(M)
```

In [15]:
```
%%R
#compairing the first ten elements of each output
# however this is done for smaller inputs and they give exactly the same ou
cat("d1[1:10]: ", d1[1:10], "\n\n", "d2[1:10]: " ,
    d2[1:10], "\n\n", "d3[1:10]: ", d3[1:10], "\n\n", "d4[1:10]: ", d4[1:10
```

```
d1[1:10]:  8.104245 20.1662 24.06307 14.31538 28.04333 11.60083 33.38971 26.67
949 3.489988 27.16704

 d2[1:10]:  8.104245 20.1662 24.06307 14.31538 28.04333 11.60083 33.38971 26.6
7949 3.489988 27.16704

 d3[1:10]:  8.104245 20.1662 24.06307 14.31538 28.04333 11.60083 33.38971 26.6
7949 3.489988 27.16704

 d4[1:10]:  8.104245 20.1662 24.06307 14.31538 28.04333 11.60083 33.38971 26.6
7949 3.489988 27.16704

 d5[1:10]:  8.104245 20.1662 24.06307 14.31538 28.04333 11.60083 33.38971 26.6
7949 3.489988 27.16704
```

Timing The Functions

The timing package rbenchmarks is used. It calculates the average time for a number of replication of the same function. Here, we used 25 replications for small inputs and 10 above 1000 rows. The output for each function's timing is a dataframe with two columns: test(name of the function) and the time elapsed. A columns for relative timings is also added. The relative coloumn is calculated with respect to the base function "dist". Since the smaller the time the better, we divided each function's time by base R function's time. Example if base R took 0.04 seconds and one of the functions took 0.01 seconds, then that function is 0.04/0.01 (= 4) times faster than base R. The code for timing is in euclid_timer.R . It should be noted that this timings are done on linux(ubuntu 19.04 Disco Dingo) on 4 GB. So it might be different for PCs/Operatiing systems with different specifications but the relative times should be nearly the same.

In [15]:
```
%%bash
cat R_code/euclid_timer.R
```

```
source("R_code/functions.R")


set.seed(1)
require('rbenchmark')
sink("small_inputs.txt", append = TRUE)
# the test is done with a 2 coloumn matrix( vectors in 2 dim  ensions )
nrows = c(5, 10, 25, 50, 75, 100, 200, 250, 500)
# nrows = c(10000)

funcs <- c("dist", "dotprod", "eucdist_C", "eucdist_R", "fastdist2")
l <- length(funcs)
Srelative_times <- data.frame(row.names = funcs)
col_index = 1 # for indexing the columns of the relative time dataframe
cat("This records the average times for 25 executions of each function\n")
for(i in nrows){
    cat(i, "by", 2, "\n")
    M <- matrix(rnorm(i*2, 10, 10), ncol = 2)
    #print(M)
    timer <- benchmark(
                dotprod(M),
                eucdist_R(M),
                dist(M),
                fastdist2(M),
                eucdist_C(M),
                columns = c("test", "elapsed"),
                replications = 25
        )

    t_base = timer$elapsed[1] # time for dist(M), the base R function

    Srelative_now <- numeric(l) # vector to contain the relative time for each
iteration
    for(rows in 1:l){
        relative_now[rows] <- round(t_base/timer$elapsed[rows], 4)
    }
    Srelative_times[col_index] <- relative_now
    col_index <- col_index + 1
    timer$relative <- relative_now
    print(timer)
    cat("\n", paste(rep("*", 100), collapse = ""), "\n")
}
names(Srelative_times) <- nrows
saveRDS(Srelative_times, file = "Srelative_times.rds")
sink()



sink("large_inputs.txt", append = TRUE)
# the test is done with a 2 coloumn matrix( vectors in 2 dim  ensions )
# nrows = c(1000, 2500, 3000, 5000, 10000)

nrows = c(2500, 3000, 5000, 10000)


funcs <- c("dist", "dotprod", "eucdist_C", "eucdist_R", "fastdist2")
l <- length(funcs)
Lrelative_times <- data.frame(row.names = funcs)
col_index = 1 # for indexing the columns of the relative time dataframe
cat("This records the average times for 10 executions of each function\n")
for(i in nrows){
    cat(i, "by", 2, "\n")
    M <- matrix(rnorm(i*2, 10, 10), ncol = 2)
    #print(M)
    timer <- benchmark(
                dotprod(M),
```

Times for Small Inputs

In [3]:
```
%%bash
cat small_inputs.txt
```

```
This records the average times for 25 executions of each function
5 by 2
          test elapsed relative
3      dist(M)   0.001   1.0000
1   dotprod(M)   0.003   0.3333
5 eucdist_C(M)   0.001   1.0000
2 eucdist_R(M)   0.002   0.5000
4 fastdist2(M)   0.001   1.0000


********************************************************************************
**********************
10 by 2
          test elapsed relative
3      dist(M)   0.001   1.0000
1   dotprod(M)   0.007   0.1429
5 eucdist_C(M)   0.002   0.5000
2 eucdist_R(M)   0.006   0.1667
4 fastdist2(M)   0.000      Inf


********************************************************************************
**********************
25 by 2
          test elapsed relative
3      dist(M)   0.001   1.0000
1   dotprod(M)   0.031   0.0323
5 eucdist_C(M)   0.001   1.0000
2 eucdist_R(M)   0.034   0.0294
4 fastdist2(M)   0.002   0.5000


********************************************************************************
**********************
50 by 2
          test elapsed relative
3      dist(M)   0.003   1.0000
1   dotprod(M)   0.115   0.0261
5 eucdist_C(M)   0.001   3.0000
2 eucdist_R(M)   0.110   0.0273
4 fastdist2(M)   0.006   0.5000


********************************************************************************
**********************
75 by 2
          test elapsed relative
3      dist(M)   0.003   1.0000
1   dotprod(M)   0.249   0.0120
5 eucdist_C(M)   0.002   1.5000
2 eucdist_R(M)   0.235   0.0128
4 fastdist2(M)   0.011   0.2727


********************************************************************************
**********************
100 by 2
          test elapsed relative
3      dist(M)   0.005   1.0000
1   dotprod(M)   0.448   0.0112
5 eucdist_C(M)   0.002   2.5000
2 eucdist_R(M)   0.444   0.0113
4 fastdist2(M)   0.018   0.2778


********************************************************************************
```

Times for Large Inputs

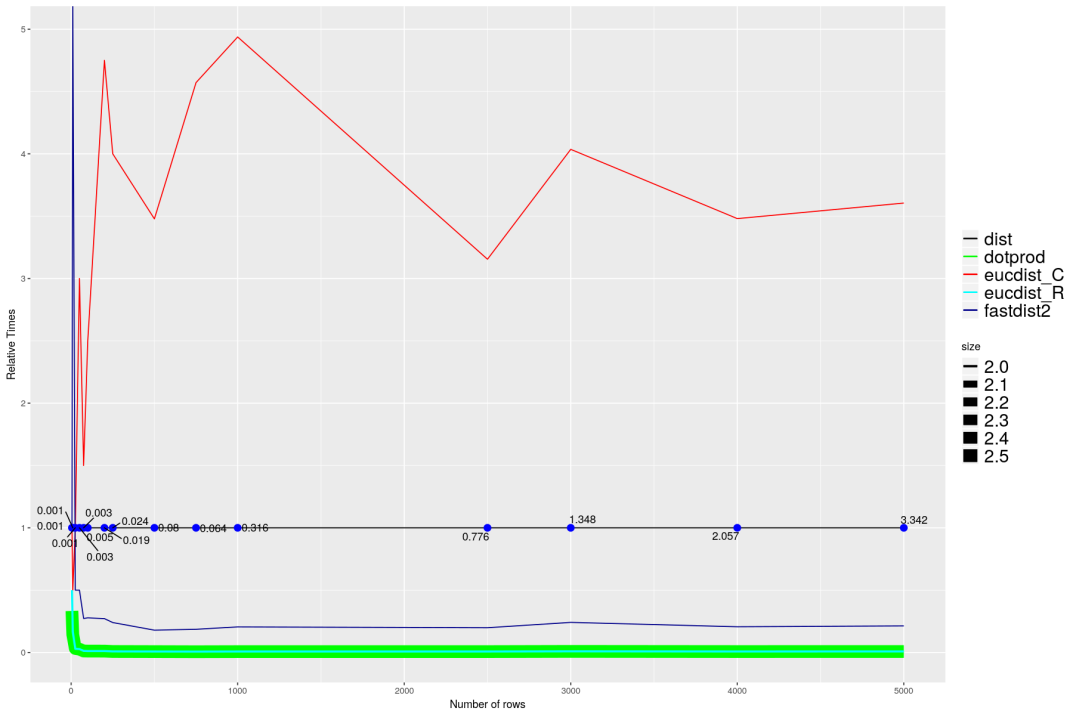
In [4]:
```
%%bash
cat large_inputs.txt
```

```
This records the average times for 10 executions of each function
2500 by 2
          test elapsed relative
3      dist(M)   0.776   1.0000
1   dotprod(M) 111.565   0.0070
5 eucdist_C(M)   0.246   3.1545
2 eucdist_R(M) 102.172   0.0076
4 fastdist2(M)   3.910   0.1985


********************************************************************************
**********
3000 by 2
          test elapsed relative
3      dist(M)   1.348   1.0000
1   dotprod(M) 155.299   0.0087
5 eucdist_C(M)   0.334   4.0359
2 eucdist_R(M) 147.574   0.0091
4 fastdist2(M)   5.587   0.2413


********************************************************************************
**********
5000 by 2
          test elapsed relative
3      dist(M)   3.342   1.0000
1   dotprod(M) 432.544   0.0077
5 eucdist_C(M)   0.927   3.6052
2 eucdist_R(M) 408.946   0.0082
4 fastdist2(M)  15.679   0.2132


********************************************************************************
**********
4000 by 2
          test elapsed relative
3      dist(M)   2.057   1.0000
1   dotprod(M) 278.817   0.0074
5 eucdist_C(M)   0.591   3.4805
2 eucdist_R(M) 265.138   0.0078
4 fastdist2(M)   9.977   0.2062


********************************************************************************
**********
```

The relative times are merged in a dataframe and save in an Rdata file relative_times.rds, and the plotting is done in ggplot2.

In [7]:
```
require(ggplot2)
relative_times <- as.data.frame(t(readRDS("relative_times.rds")))
time_base = as.character(readRDS("time_base.rds")) # the base R times)
n = as.numeric(row.names(relative_times))

g = ggplot(data = relative_times, aes(x = n))
g1 = g + geom_line(aes(y = dotprod, color = "dotprod", size=2.5))+ geom_lir
g2 = g1 + geom_line(aes(y = dist, color = "dist")) + geom_line(aes(y = eucc
g3 <- g2 + geom_line(aes(y = fastdist2, color = "fastdist2"))
g4 <- g3 + scale_color_manual(NULL,
            values=c(dotprod = "green", eucdist_R = "cyan", dist= "black",
            fastdist2 = "darkblue", eucdist_C = "red")) +
            theme(legend.text = element_text(size=18)) + labs(y ="Relative
```

In [8]:
```
# %%R
library(repr); library(ggrepel)

options(repr.plot.width=15, repr.plot.height = 10)
g4 + annotate("point", x = n, y = 1, color = "blue", size=3) + geom_text_re
```

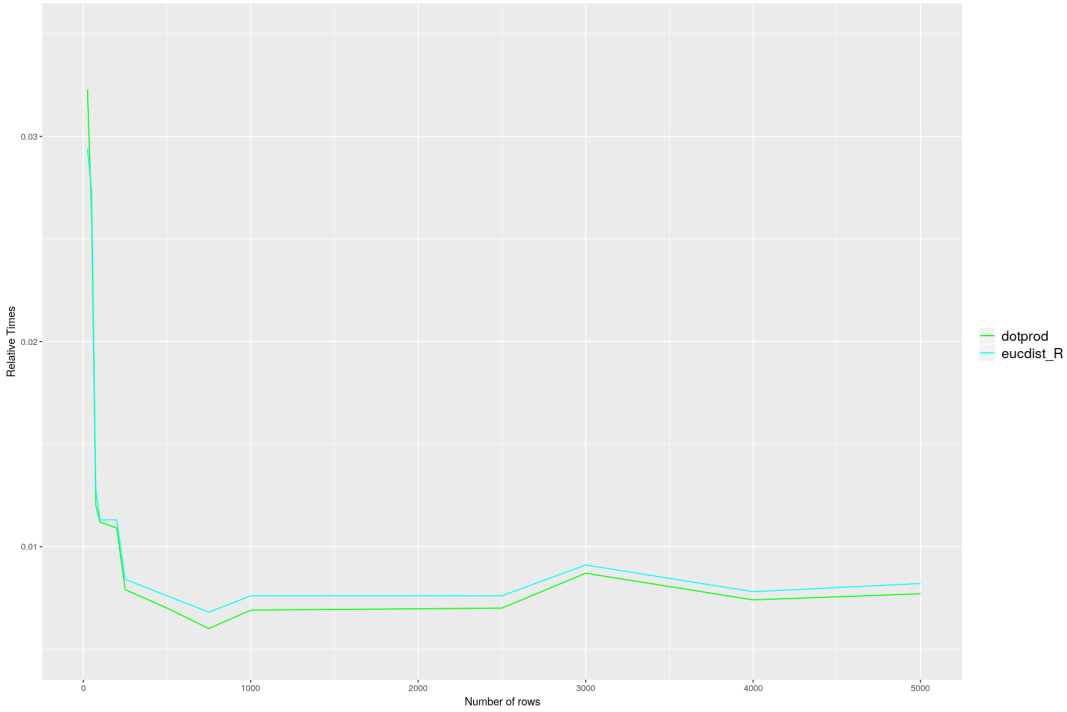


Note that the lines for eucdist_R and dotprod are superimposed on each other. Here is a closer look at their comparison

In [21]:
```
# %%R
g5 = g + geom_line(aes(y = dotprod, color = "dotprod")) +
    geom_line(aes(y = eucdist_R, color = "eucdist_R")) +  scale_color_manua
    values=c(dotprod = "green", eucdist_R = "cyan")) +
    theme(legend.text = element_text(size=14)) + ylim(0.005, 0.035) + labs(
```

In [22]:
```
1 g5
```

```
Warning message:
“Removed 2 rows containing missing values (geom_path).”Warning message:
“Removed 2 rows containing missing values (geom_path).”
```



In [2]:
```
# %%R
relative_times <- as.data.frame(t(readRDS("relative_times.rds")))
relative_times
```

A data.frame: 15 × 5

	dist	dotprod	eucdist_C	eucdist_R	fastdist2
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
5	1	0.3333	1.0000	0.5000	1.0000
10	1	0.1429	0.5000	0.1667	Inf
25	1	0.0323	1.0000	0.0294	0.5000
50	1	0.0261	3.0000	0.0273	0.5000
75	1	0.0120	1.5000	0.0128	0.2727
100	1	0.0112	2.5000	0.0113	0.2778
200	1	0.0109	4.7500	0.0113	0.2714
250	1	0.0079	4.0000	0.0084	0.2400
500	1	0.0070	3.4783	0.0076	0.1790
1000	1	0.0069	4.9375	0.0076	0.2051
2500	1	0.0070	3.1545	0.0076	0.1985
3000	1	0.0087	4.0359	0.0091	0.2413
5000	1	0.0077	3.6052	0.0082	0.2132
4000	1	0.0074	3.4805	0.0078	0.2062
750	1	0.0060	4.5714	0.0068	0.1860

Conclusion

From the plots it is evident that C is performing a lot better than the base R function. Even after formatting the output for the C wrapper using mefa::vec2dist it still performs at least 2x faster. C++ does well on small inputs compared to base R but for number of rows more than 25 it becomes slower untill it settles at minimum of about 0.3x as fast(about 3x slower). The function written in pure R (dotprod and eucdist_R) were the slowest throughout. This might be because all the other functions are using C/C++.

In [2]:
```
%%bash
cat out_compareAf.txt
```

```
This records the average times for 50 executions of each function
5 by 2
            test elapsed relative
3        dist(M)   0.001   1.0000
1     dotprod(M)   0.005   0.2000
5   eucdist_C(M)   0.015   0.0667
2   eucdist_R(M)   0.005   0.2000
4 fastdist2_R(M)   0.007   0.1429


********************************************************************************
**********
10 by 2
            test elapsed relative
3        dist(M)   0.001   1.0000
1     dotprod(M)   0.006   0.1667
5   eucdist_C(M)   0.002   0.5000
2   eucdist_R(M)   0.006   0.1667
4 fastdist2_R(M)   0.001   1.0000


********************************************************************************
**********

25 by 2
            test elapsed relative
3        dist(M)   0.001   1.0000
1     dotprod(M)   0.020   0.0500
5   eucdist_C(M)   0.001   1.0000
2   eucdist_R(M)   0.018   0.0556
4 fastdist2_R(M)   0.001   1.0000



********************************************************************************
**********
50 by 2
            test elapsed relative
3        dist(M)   0.002   1.0000
1     dotprod(M)   0.128   0.0156
5   eucdist_C(M)   0.001   2.0000
2   eucdist_R(M)   0.113   0.0177
4 fastdist2_R(M)   0.005   0.4000


********************************************************************************
**********
75 by 2
            test elapsed relative
3        dist(M)   0.003    1.000
1     dotprod(M)   0.249    0.012
5   eucdist_C(M)   0.002    1.500
2   eucdist_R(M)   0.251    0.012
4 fastdist2_R(M)   0.012    0.250


********************************************************************************
**********
100 by 2
            test elapsed relative
3        dist(M)   0.004   1.0000
1     dotprod(M)   0.445   0.0090
5   eucdist_C(M)   0.002   2.0000
2   eucdist_R(M)   0.415   0.0096
4 fastdist2_R(M)   0.018   0.2222
```

In [40]:
```
dist # the .Call bit is using R's C API to call a C function
```

```
function (x, method = "euclidean", diag = FALSE, upper = FALSE,
    p = 2)
{
    if (!is.na(pmatch(method, "euclidian")))
        method <- "euclidean"
    METHODS <- c("euclidean", "maximum", "manhattan", "canberra",
        "binary", "minkowski")
    method <- pmatch(method, METHODS)
    if (is.na(method))
        stop("invalid distance method")
    if (method == -1)
        stop("ambiguous distance method")
    x <- as.matrix(x)
    N <- nrow(x)
    attrs <- if (method == 6L)
        list(Size = N, Labels = dimnames(x)[[1L]], Diag = diag,
            Upper = upper, method = METHODS[method], p = p, call = match.call
(),
            class = "dist")
    else list(Size = N, Labels = dimnames(x)[[1L]], Diag = diag,
        Upper = upper, method = METHODS[method], call = match.call(),
        class = "dist")
    .Call(C_Cdist, x, method, attrs, p)
}
```

Python

In python, almost the same thing is done except that the C++ function was using R's data types and could not be loaded in python without significant changes. The functions wriiten are

1. dot_prod_dist: using dot product
2. e_dist: using vectorization
3. dist: written in cython(C with python) Since there is no function in python that calculate distances, all comparison were made with respect to the e_dist function. Few tweaks were made to the eucdist.c used in R to make it work with cython.

In [7]:
```
from os import chdir
chdir("CyPy")
```

In [8]:
```
%%bash
# cd CyPy
cat eucdist.c
```

```
#include <math.h>
#include <stdio.h>
/* Euclidean distance */
/*q=.C("eucdist",as.integer(c(1,2,3,4,5,6,7,8)),as.integer(4),as.integer(2),a
s.double(vector("double",6))*/
int euc_dist(double *x, int m, int n, double *d)
{
  /* Arguement:
    1.  x is a matrix of dimension n by m
    2.  m is the number of rows
    3.  n is the number of coloums
    4.  d is the pointer for output */
  /*
   d = sqrt(sum((XI-XJ).^2,2));            % Euclidean
   */
  int i,j,k; /* **pointer; /* Indexers */
  double theSum; /* size_t is an unsigned integer of size 16 bits */
  /*
  XI for indexing rows
  XJ for indexing columns
  XI0 unknown for now
  */
  int  XI, XJ, XI0, index; /* pointers as row indexers*/
  index = 0;

  for (i=0; i<m-1; ++i) { /* Iterating through the rows of the matrix */
    // XI0 =  XI; /* taking the memory address of the array (Refer to line 29)
*/
    XI = i*n; /* Move along memory by n ( the first coloumn */
    XI0 = XI;
    // Rprintf("XI is %d\n", XI);
    for (j=i+1; j<m; ++j) { /* iterating through the rows from the i_th row*/
      // XI = x + i*(*n); /* Change to XI happpens here after using it on line
28*/
      XJ = j*n;
      // Rprintf("XJ is %d\n", XJ);
      // XI = XI0; /* Index? */
      theSum = 0.0;
      for (k=0;k<n;k++,++XI,++XJ){
        theSum += pow((x[XI]- x[XJ]), 2.0);
        // Rprintf("x[XI] is %lf and x[XJ] is %lf\n", x[XI], x[XJ]);
        // Rprintf("The sum is %lf\n", theSum);
        // Rprintf("The sum is %d\n", theSum);
      }
      XI = XI0;
      d[index++] = sqrt(theSum);
      // Rprintf("d is %lf\n", d[index]);
      // XI = XI0; /* Index? */
    }
  }
  return 1;
}
```

In [9]:
```
%%bash
cat ceucdist.pyx
```

```
import numpy as np
cimport numpy as np
from copy import deepcopy # this is used to avoid the function from changing t
he input

cdef extern from "eucdist.h":
    bint euc_dist(double* input, int m, int n, double* output)


def dist(input):
    "Input is an nd array and the function returns the distance between any tw
o rows"
    # this helps to prevent changing the nature of the input
    retain_global_input = deepcopy(input)
    m, n = retain_global_input.shape
    length = m*n
    retain_global_input.shape = length
    output = np.empty(int(m*(m-1)/2), dtype=np.float64)
    status = euc_dist(<double*> np.PyArray_DATA(retain_global_input),
                      m, n, <double*> np.PyArray_DATA(output))

    assert status == 1, "There is a problem with the compilation/linking"
    return output
```

The C function is buid as a module using setup.py and imported into python

In [11]:
```
%%bash
cat setup.py
```

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy


setup(
name = "Something2",
# Not the name of the module
cmdclass = {"build_ext":build_ext},
# magic
ext_modules = [ Extension("mymodule2",
# The name of the module
["ceucdist.pyx"],
libraries=["eucdist"], include_dirs=[numpy.get_include()]) ]
)

# gcc -c eucdist.c -o eucdist.o
# ar cr libeucdist.a eucdist.o
# CFLAGS="-I." LDFLAGS="-L." python3 setup.py build_ext -i
```

In [46]:
```
# To compile the C code just run the following on the command line
# gcc -c eucdist.c -o eucdist.o
# ar cr libeucdist.a eucdist.o
# CFLAGS="-I." LDFLAGS="-L." python3 setup.py build ext -i
```

The Functions Written In Pure Python

In [13]:
```
%%bash
cat functions.py
```

```
import numpy as np
from math import sqrt


def dist_func(A, B,  func):
        ''' apply a distance calculating function two list'''
        return func(A, B)


# using vectorization
def e_dist(A, B):
        assert len(A) == len(B), "Invalid input(s)"
        return sqrt(sum([(A[i] - B[i])**2 for i in range(len(A))]))

#using dot product
def dot_prod(u, v):
        assert len(u) == len(v), "invalid input(s)"
        return sum(np.array(u)*np.array(v))


def dot_prod_dist(u, v):
        assert len(u) == len(v), "invalid input(s)"
        A = np.array(u) - np.array(v)
        return sqrt(dot_prod(A, A))



def the_loop(M, func):
        # M is a numpy nd array
        m = M.shape[0]
        distance = np.zeros(int(m*(m-1)//2))
        k = 0
        for i in range(m-1):
                A = M[i, ]
                for j in range(i+1, m):
                        distance[k] = dist_func(A, M[j, ], func)
                        k += 1
        return distance
```

Note that the same idea used in R. The_loop function takes a matrix(nd array) and a distance calculating function and apply to any two unique row combinatins.

Testing for Consistency

In [14]:
```
import numpy as np
import os
# os.chdir('/home/medfad/Desktop/WD/Abdul/2019/Project_Euclid/CyPy')
from random import normalvariate, seed
from math import sqrt
from mymodule2 import dist
import sys
import timeit
from functions import *
```

In [15]:
```
seed(1)
m, n = 1000, 2
M = np.empty([m,n])

for i in range(m):
    for j in range(n):
        M[i, j] = normalvariate(10, 10)
```

In [16]:
```
1 M[:11, ]
```

Out[16]:
```
array([[16.07455858,  9.85774554],
       [22.30907229, 20.15481167],
       [ 6.63543072, 22.17481809],
       [ 1.58685673,  7.88658975],
       [ 4.04653337,  1.07250122],
       [23.05670242, 12.99854821],
       [ 0.25372232, 23.15622659],
       [22.95613455, 27.17303476],
       [ 1.8207828 , 10.22054965],
       [-0.54504182, 18.71124194],
       [-2.43269821, 20.70067292]])
```

In [18]:
```
1 d1 = dist(M)
2 d2 = the_loop(M, e_dist)
3 d3 = the_loop(M, dot_prod_dist)
```

In [19]:
```
1 print("d1[:11]: ", d1[:11], "\n\n", "d2[:11]: ", d2[:11], "\n\n", "d3[:11]:
```

```
d1[:11]:  [12.03738892 15.51797057 14.62118189 14.8947611   7.65604167 20.6675
7019
 18.63264148 14.25839229 18.83070672 21.44965332  9.93768337]

 d2[:11]:  [12.03738892 15.51797057 14.62118189 14.8947611   7.65604167 20.667
57019
 18.63264148 14.25839229 18.83070672 21.44965332  9.93768337]

 d3[:11]:  [12.03738892 15.51797057 14.62118189 14.8947611   7.65604167 20.667
57019
 18.63264148 14.25839229 18.83070672 21.44965332  9.93768337]
```

Compairing Times

The times were compared for few rows and here is the output.

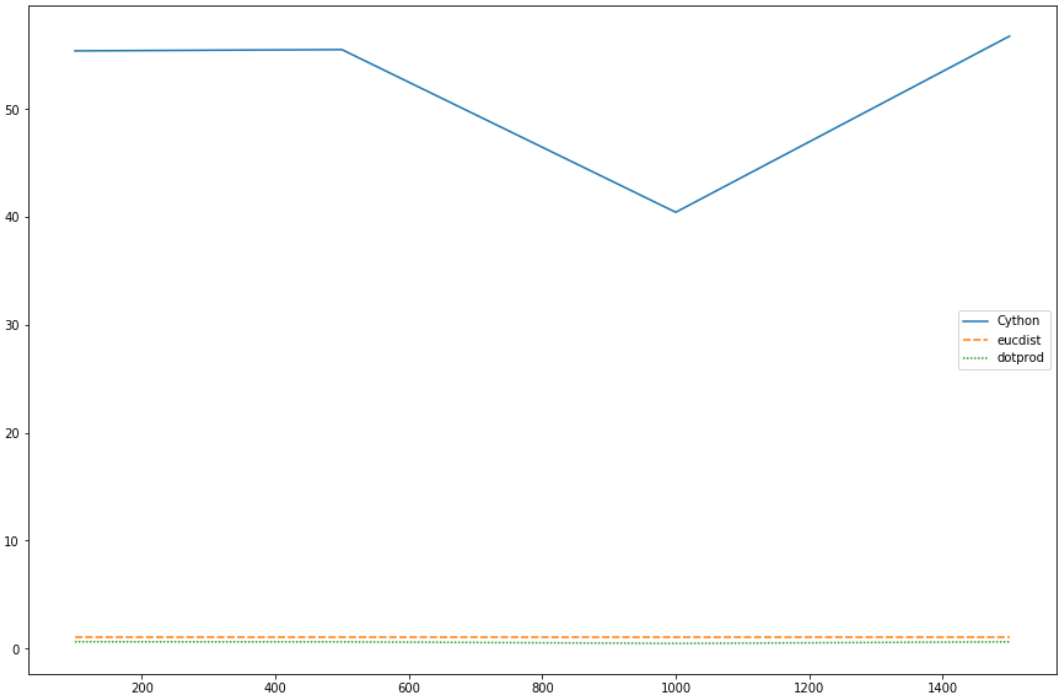
In [43]:
```
%%bash
cat py_compare_times.txt
```

```
The times are calculated by taking the average time of 100 replications of the
same function


********************************************************************************
**

100 by 2

python function without dot product
        time    relative
        3.045   1

using C
        time    relative
        0.055   55.364

using dot product
        time    relative
        4.872   0.625



********************************************************************************
**

500 by 2

python function without dot product
        time    relative
        75.231  1

using C
        time    relative
        1.356   55.48

using dot product
        time    relative
        122.563 0.614



********************************************************************************
**

1000 by 2

python function without dot product
        time    relative
        315.346 1

using C
        time    relative
        7.803   40.413

using dot product
        time    relative
        673.697 0.468



********************************************************************************
**

1500 by 2
```

Here we can see the trend repeats except that the C/Python function is performing significantly faster. Upto about 55x faster

In [20]:
```
import pandas as pd
import numpy as np

time_rows = pd.DataFrame(columns = ["Cython", "eucdist", "dotprod"],
                         index = [100, 500, 1000, 1500], dtype=np.float64)
time_rows['Cython'] = [55.364, 55.48, 40.413, 56.729]
time_rows['eucdist'] = np.ones(4)
time_rows["dotprod"] = [0.625, 0.614, 0.468, 0.617]
base_time = [3.045, 75.231, 315.346, 982.838]
```

In [29]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
plt.figure(figsize=(15, 10))
sns.lineplot(data=time_rows)
plt.show()
```



Unsuprisingly Cython is doing much better. The trend does seemed to settle at a value as it does in R but this might be evident for more inputs. Overall R is performing better in almost all of the function they have in common. Here is look at the time the base python function eucdist takes

In [27]:
```
base_time # times for [100, 500, 1000, 1500]
```

Out[27]:
```
[3.045, 75.231, 315.346, 982.838]
```