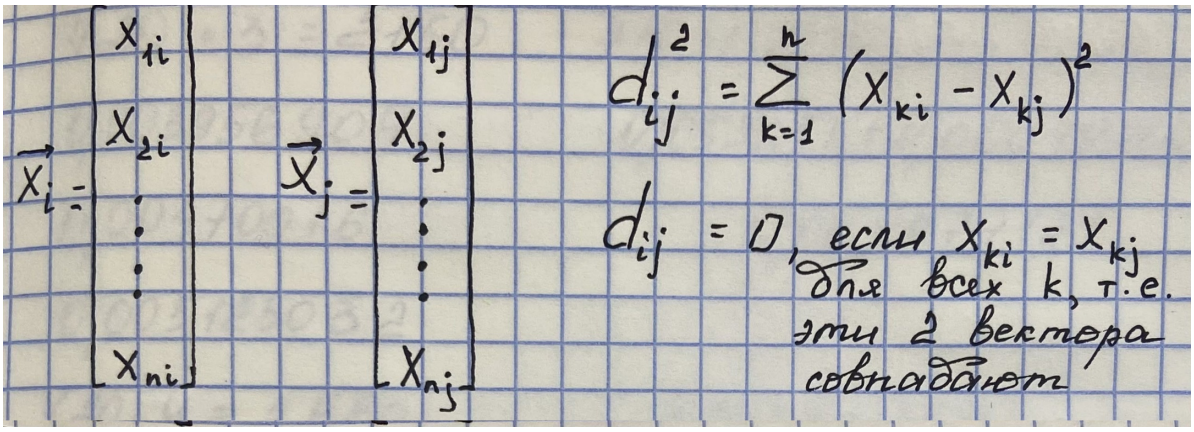
Sammon

Мы имеем в *n-мерном* пространстве набор точек векторов $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N$, где N – штук. Имеем на плоскости набор точек вектора заданные $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_N$.

Для каждой пары точек i и j в исходном *n-мерном* пространстве мы можем посчитать расстояние между ними, по формуле

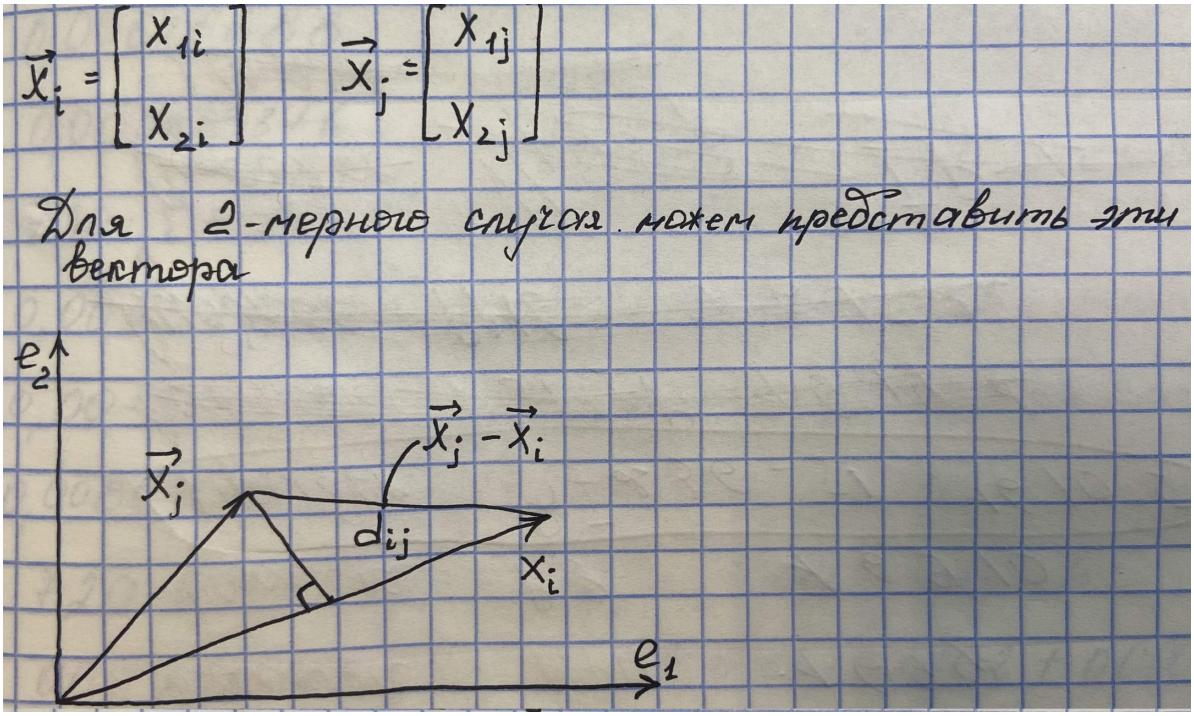
$$d_{ij} = \|\vec{x}_i - \vec{x}_j\| = \sqrt{\sum_{k=1}^{n} (x_{ik} - x_{jk})^2}$$



Тоже самое можем делать в двумерном пространстве (расстояние между y).

*–плоскость

$$d_{ij}^{*} = \|\vec{y}_i - \vec{y}_j\| = \sqrt{\sum_{k=1}^{2} (y_{ik} - y_{jk})^2}$$



В чем простая по смыслу, но трудоемкая в вычислительном плане, идея Сэммона?

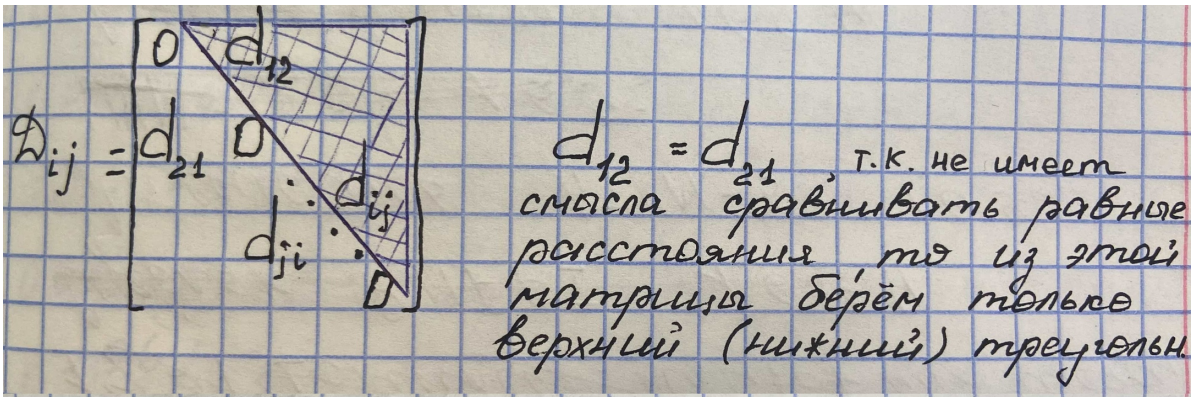
В том, чтобы расположить точки на плоскости так, чтобы взаимные расстояния * между i и j точками максимально соответствовали расстояниям в исходном *n-мерном* пространстве. Т.е. те точки, которые в исходном *n-мерном* пространстве далеко друг от друга, должны быть и на плоскости далеко друг от друга и наоборот.

Критерием этого, чтобы кто далеко в *n-мерном* был и на плоскости далеко или наоборот, является следующее.

$$\xi=(\sum_{i<j}^{N} d_{ij})^{-1}\sum_{i<j}^{N}(d_{ij}-d_{ij}^{*})^2/d_{ij}$$

Обратим внимание на разность расстояния в *n-мерном* и расстояния на плоскости взятую в квадрате.

Т.к. матрица расстояний симметричная, т.е.



Для определенности берем нижний треугольник и это обусловлено суммированием, где *i<j*, т.о. задается нижний треугольник.

Т.е. мы должны оптимизировать квадраты разностей между расстояниями в исходном *n-мерном* пространстве и на плоскости.

Что делает /d_{ij}? Он делает разности относительными, если в исходном n-мерном пространстве d_{ij} большое, то разность может быть побольше, они и так далеко друг от дружки и эта разность может быть побольше, а там, где они маленькие, то чувствительность к разности становится сильнее.

$(\sum_{i<j}^{N} d_{ij})^{-1}$ – это некий нормировочный множитель , когда мы все то хозяйство доделили на сумму всех d_{ij}. Вся любовь.

И для минимизации этого критерия, нужно минимизировать его, т.е. с d_{ij} мы ничего не можем делать, а с d_{ij}* мы можем двигать в разные стороны, т.е. если какие-то ***i j*** точки оказались близко между собой, а в исходном пространстве они далеко, мы их просто раздвинем и наоборот.

Откуда берутся $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_N$? Один из простых вариантов – мы на плоскость N точек высыпаем случайно. Например, на ватмане рисуем систему координат y1 y2 и высыпаем N зерен на плоскость. Далее нумеруем эти зерна в произвольном порядке каждое и после, посчитав критерий ξ, начать раздвигать зернышки, которые упали рядом, а в *n-мерном* пространстве они далеко друг от друга или сдвигать те, которые на плоскости упали далеко, а в n-мерном пространстве они близко.

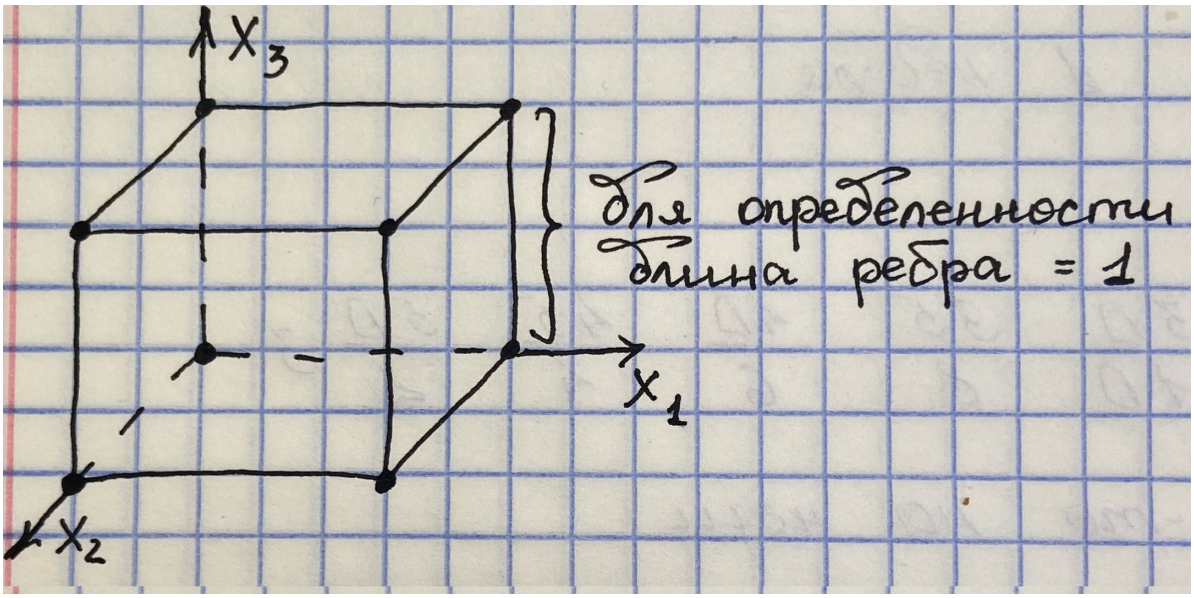
Реализация на APL:

```
[0] z←{y}sam x;e;i;j;k;dx;dy;m;a;c;l
[1] a←0.35  ⍝ [.3,.4] Sammon told
```

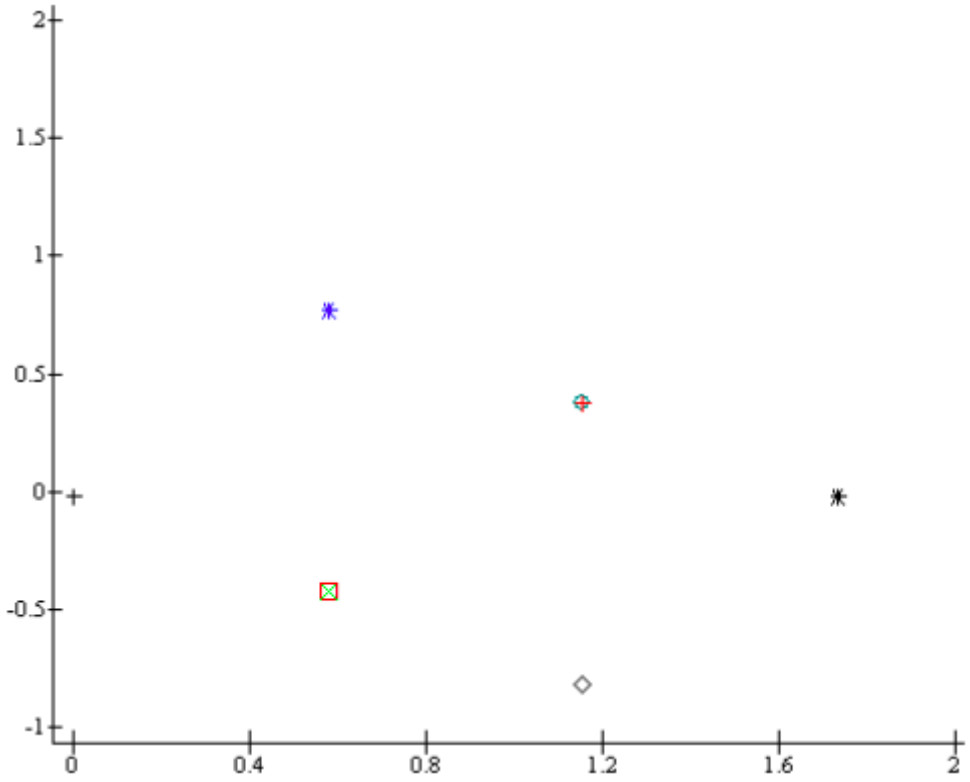
```
[2] ⍎(0=⎕NC'y')/'y←⍀mds.Orloci x'
[3] dx←dist x
[4] m←,m°.<m←⍳↑⍴dx
[5] dy←dist y
[6] e←⍬
[7] L:e,←(÷+/m/,dx)×+/(m/,(dx-dy)*2)÷m/,dx
[8] :For i :In ⍳↑⍴y
[9] l←i≠⍳↑⍴y
[10] c←(l⌿(y[i;]-[2]y)×[1]dx[i;]-dy[i;])÷[1]l/dx[i;]×dy[i;]
[11] y[i;]←y[i;]+(2×a÷+/m/,dx)×+⌿c
[12] :EndFor
[13] dy←dist y
[14] →(1000>⍴e)/L
[15] z←e y
```

Правый аргумент от *sam* – вектор векторов n-мерных, а левый аргумент от *sam* в фигурных скобках означает опциональность, т.е. он может быть, а может не быть. Если он есть, то откуда-то мы на плоскости эти N точек поместили. А если мы не задаем левый аргумент, то в качестве левого приближения используются проекции на оси Орлочи. Орлочи нам как-то скорректирует, а дальше начинаем оптимизировать, далее идет цикл, т.к. при градиентной оптимизации без цикла никак.

Рассмотрим пример в трехмерном пространстве:



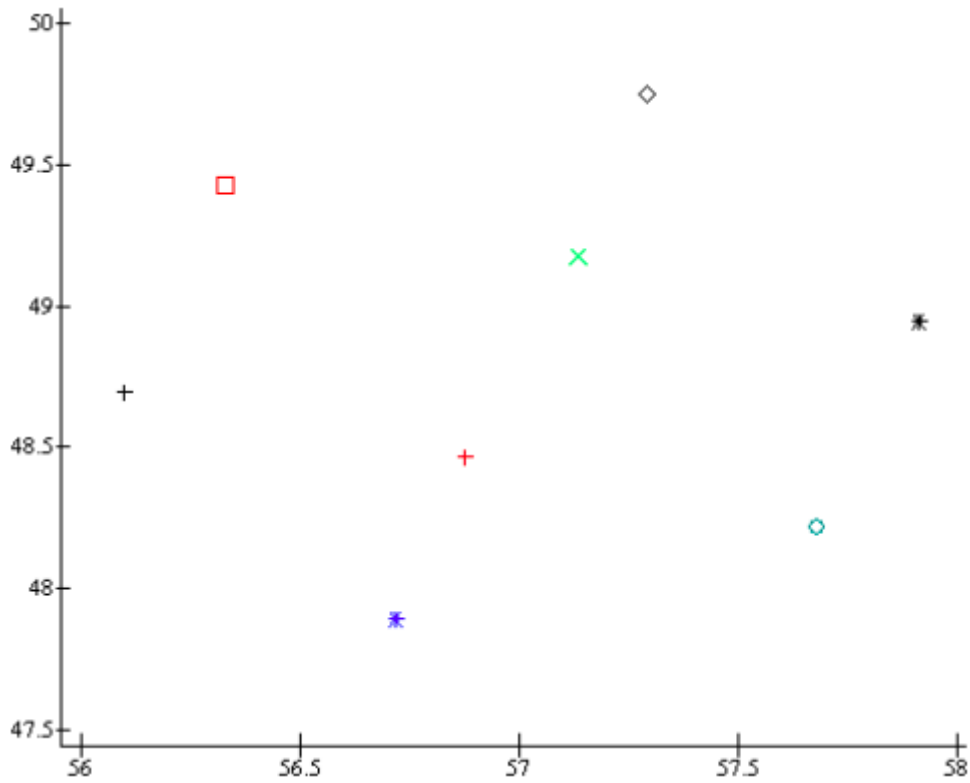

```
      ⍴xy                         ⍝вершины этих кубиков
8 3
      xy                          ⍝матрица из координат вершин
0 0 0
1 0 0
0 1 0
0 0 1
0 1 1
1 0 1
1 1 0
1 1 1
      (⍳8) plotc ⊂[2]Orloci xy    ⍝спроектируем ее с помощью Орлочи
```



Специально делаем каждую точку своим маркером и цветом, если бы мы так не сделали, то на проекции Орлочи мы бы увидели 5 точек вместо 8-ми. А так видим, что при линейном проектировании товарища Орлочи некоторые точки совпали, это фигово, в том плане, что расстояние между любой парой вершин не меньше 1, а тут вышло, что между какими-то вершинами расстояние 0.

Теперь зовем товарища Сэммона для проектирования.

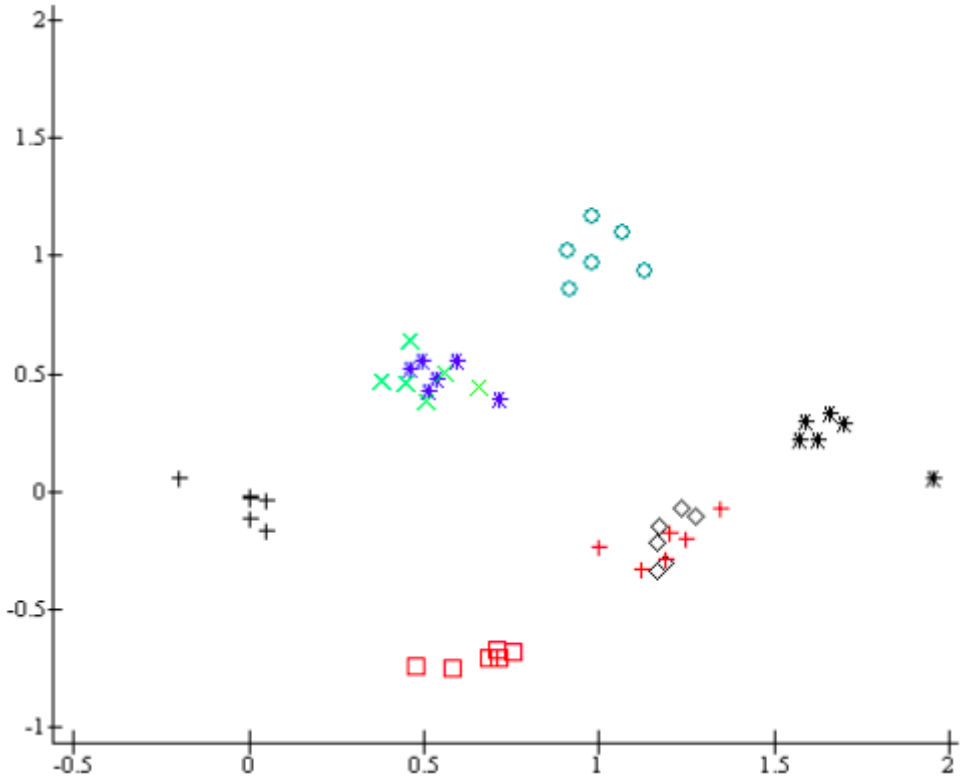
```
z←sam xy
(⍳8) plotc ⊂[1]2⊃z
```



Результат – все наши 8 точек на месте, так выглядит кубик при проектировании на плоскость.

Представим, что в каждой вершине кубика несколько точек соответствующие одному из 8-ми классов:

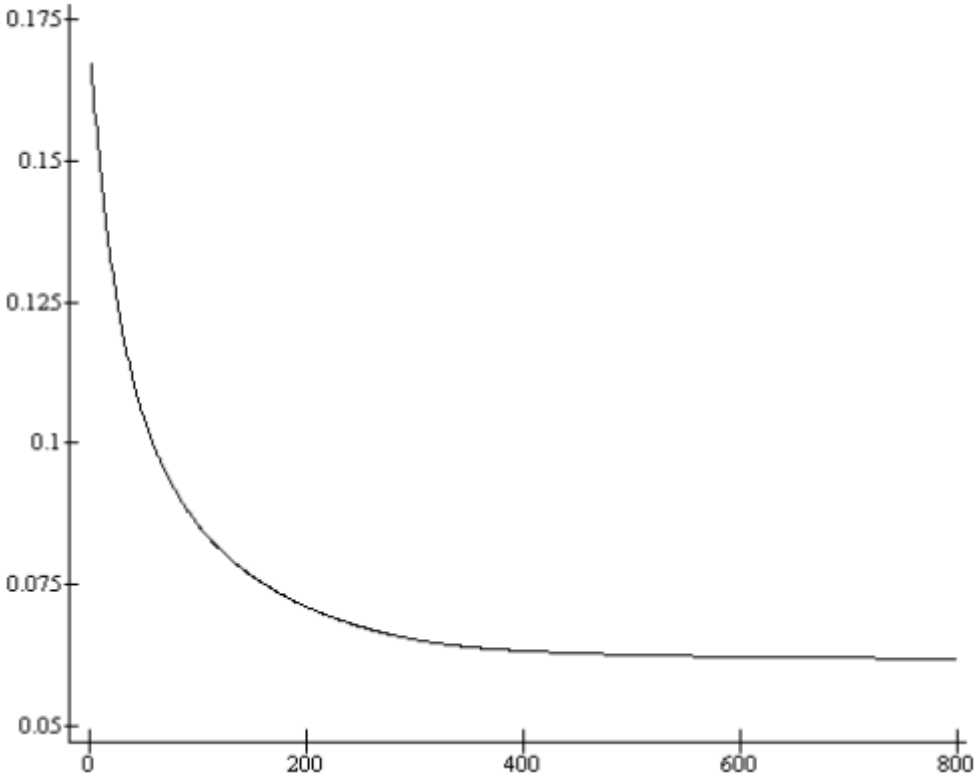
```
      xy1←⊂[2]xy
      xy2←xy1{⍺,[1]⍺+[2]⍵}¨⊂[2 3]8 5 3⍴0 0.1 stat.rndn×/8 5 3
      ⍴xy2←,[1 2]⊃xy2
48 3
      (∊6⍴¨⍳8) plotc ⊂[2]mds.Orloci xy2
```



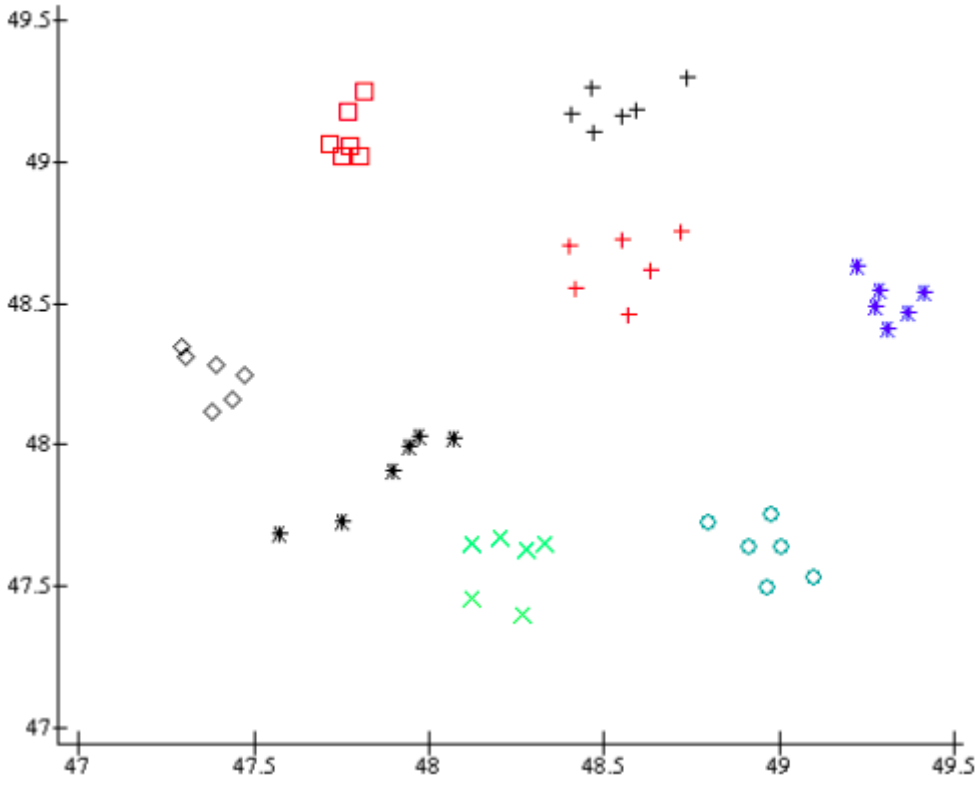
Вот у нас 8 болезней и больные болезнью номер 1, попали в одну вершину кубика, больные болезнью 2 в другую и т.д. И мы с помощью Орлочи это спроектировали на плоскость. Но можем различить только 4 болезни из 8-ми, потому что как и с точками у нас одна болезнь наложилась на другую и мы не можем их различить. Это очень плохо как для медицинской, так и технической диагностики.

Эти же данные спроектируем на оси Сэммона

```
      z2←(?48 2⍴100)sam xy2
      plot 200↑↓z2    ⍝ как меняется критерий ξ при оптимизации
```



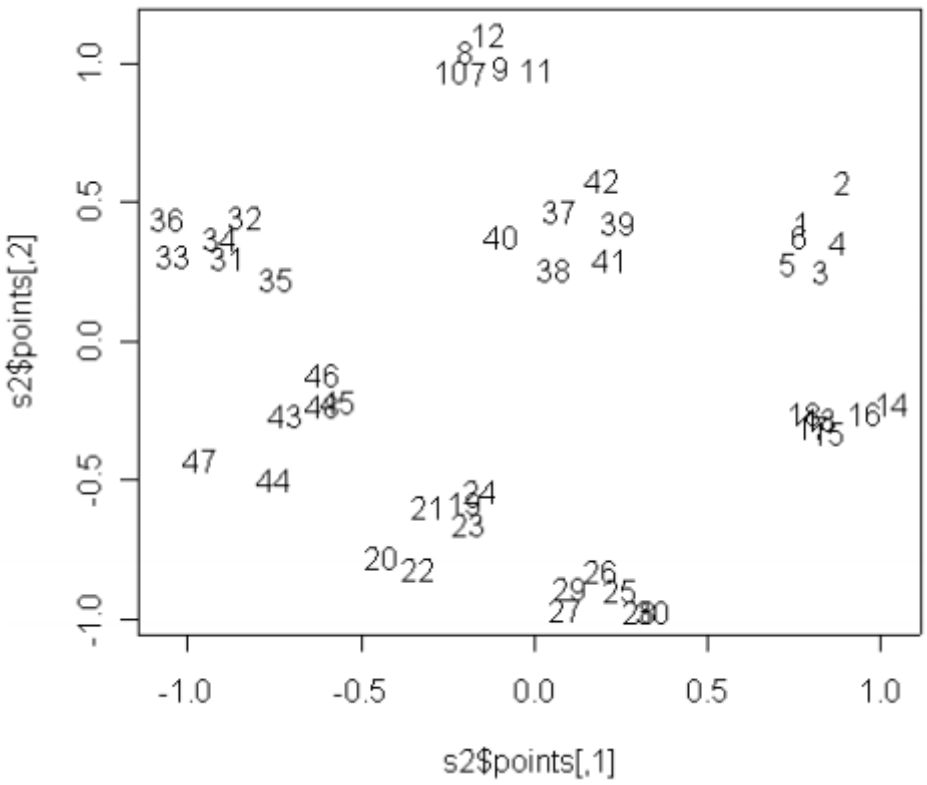

```
(∈6ρ¨ι8) plotc ⊂[1]2⊃z2
```



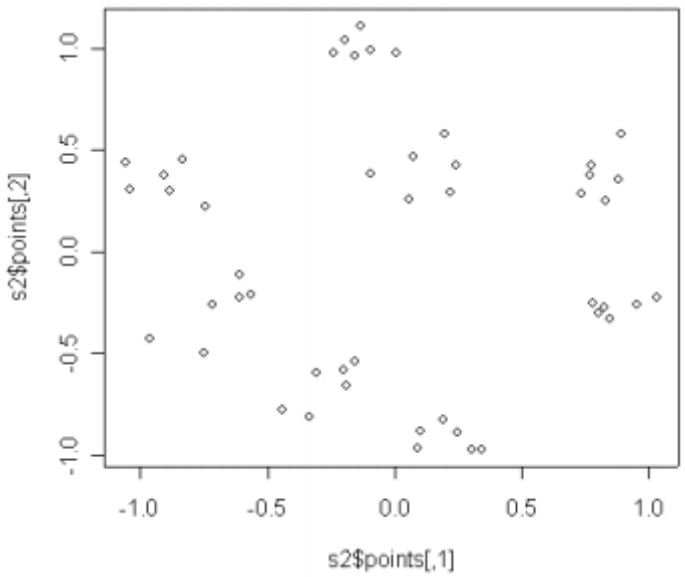
Видим, что все 8 разных классов друг от дружки отстоят далеко и, используя метод распознавания (эталонов – среднее по всем точкам принадлежащим данному классу), можем распознать центры для каждого класса болезней. Метод эталонов заключается в том, что мы можем поставленную точку в пространстве определить к какому-то из классов, посчитав расстояние от неизвестной точки до центра эталона и там где оно меньше, значит к тому классу мы точку и определим.

Реализация Сэммона на R:

```
> xy2<-read.table("d:/xy2.txt")
> xy2<-as.matrix(xy2)
> s2<-sammon(xy2)
Error in sammon(xy2) : Distances must be result of dist or a square matrix
> s2<-sammon(dist(xy2))
Initial stress : 0.08900
stress after 10 iters: 0.05915, magic = 0.500
stress after 20 iters: 0.05881, magic = 0.500
stress after 30 iters: 0.05871, magic = 0.500
stress after 40 iters: 0.05864, magic = 0.500
> plot(s2$points,type="n")
> text(s2$points,labels=as.character(1:nrow(xy2)))
>
```



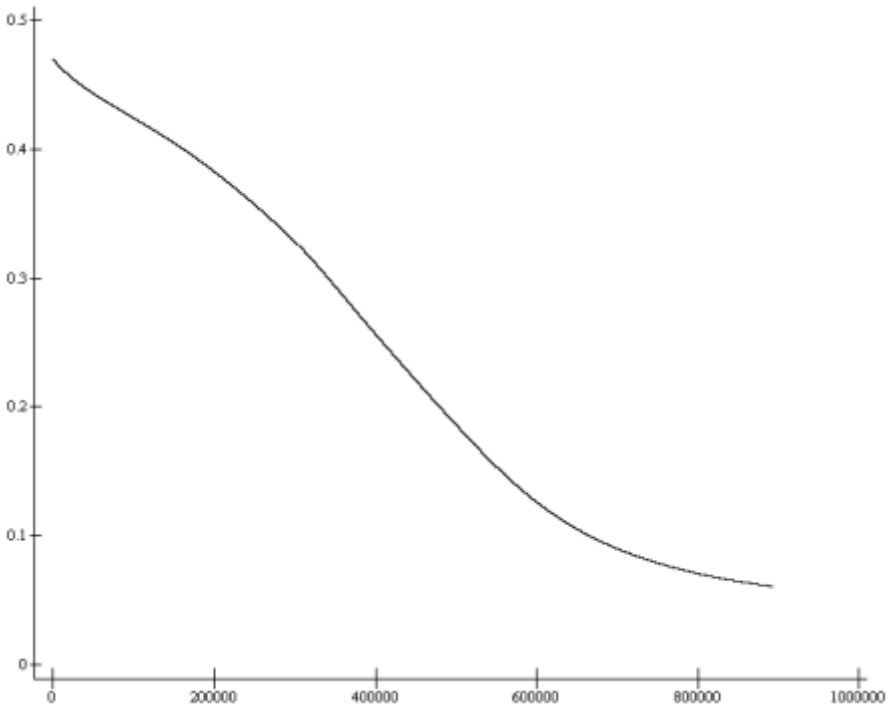

```
> plot(s2$points,type="p")
```



```
      ⍴s                  ⍝ взяли в качестве начального приближения какой-то
пятимерный массив
47 5
      z←(?47 2⍴100)sam s   ⍝ запустили его с Сэммоном, взяв в качестве
приближения случайные 47 точек на плоскости
```

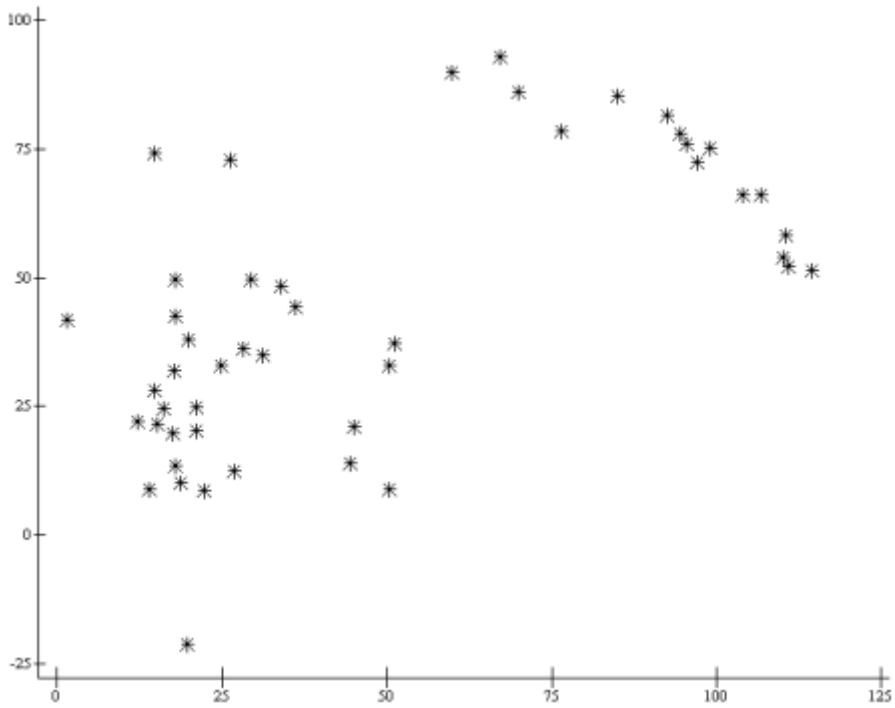
Запустили на ночь, прервали утром. Прервался на 9-ой строчке.

```
      z←(2⊃z)sam s
      sam[9]
      ⍴e        ⍝видим, что за ночь 894678 итераций прошли за ночь
894678
      ee←e
      ⍴y
47 2
      yy←y
      plot ee   ⍝строим график изменений в процессе ночного бдения
```



Видим как выглядят наши проекции:

```
     0 plot ⊂[1]yy
```



Это, конечно, безобразие. Дело в том, что APL – язык интерпретатор.

Это означает, что мы должны какую-то строчку интерпретировать, т.е. в компилируемых языках мы должны вот эту программу, которая должна "3+4" считать, компилировать и она мухой выполнится.

В APL мы должны интерпретировать и естественно, то как мы интерпретировали эту строчку и поняли, что **«+»** это суммирование, у нас уже включится для плюса написанная на языке низкого уровня откомпилированная процедура, которая его мухой выполнит, но на интерпретацию уйдет время (нас это не напрягает).

```
      3+4
7
      +/(⍳1000)*2
333833500
```

Посмотрим со стороны Basic. Результат у него будет:

```
     )ed basic
     ∇ r←basic n;i
     [1]    r←0
         [2]    :For i :In ⍳n
             [3]        r←r+i*2  ⍝ interpret this line n times
                 [4]    :EndFor
                       ∇
                             basic 1000
333833500
```

Трагизм ф-ции Basic, что в [3] будет интерпретироваться не 1 раз, а 1000 раз, следовательно ф-ции Basic в разы медленнее, чем **«+/»**.

R является интерпретирующим. Он быстрый, потому что. если в APL мы Сэммона пишем по строчкам и цикл, внутри цикла интерпретировались все строчки каждый раз, именно поэтому он всю ночь телепался.

Поэтому в R, когда зовем *dist(xy**2)*, то эта ф-ция написана не на R, а, например, на C, из-за чего в R это является вызовом. Мы никогда не увидим, что там внутри Сэммона делается или, что внутри dist делается. Это откомпилированный код, который вместе с R поступает на компьютер при установке. Они этому коду передают только аргумента, а он быстро-быстро откомпилированный крутится, поэтому нам кажется, что R быстрее, но быстрота заключается в том, что он вызывает каждый раз функцию.

В APL реализована связь с R:

```
      )copy rconnect
      r←⎕new R   ⍝создаем новый объект
      r.init     ⍝инициализируем
RConnect initialized
```

Внутри R даны функции:

```
      'x' r.p ⍳9      ⍝помещает в R какой-то объект из APL, левый аргумент – под
каким именем я его буду помещать, а правый аргумент – что за объект
      +r.g 'x'        ⍝взять из R какой-то объект
1 2 3 4 5 6 7 8 9
      +r.x 'sum(x)'   ⍝в качестве аргумента берет вызов какой-то R-ой функции
каким-то аргументом. Например, X – которые засланы в R просуммируем там же
45
      +/⍳9            ⍝проверили не наврал ли R
45
```

```
⍝ generate 3 clusters data for sammon prejections
      x1←?30 50⍴0
      x2←2+?30 50⍴0
      x3←¯2+?30 50⍴0
      ⍴xx←x1⍪x2⍪x3
90 50
      r.x'library(MASS)'
      r.x'ss<-sammon(dist(ω))'xx
      r.x'ss<-plot(ss$points)'
      pp←r.g'ss$points'
      pp
[R matrix: 90x2 : dimnames]
      p1←r.x'p1<-pp[,1]'
      p2←r.x'p2<-pp[,2]'
      ]load plt
#.plt
      plt.plot p1 p2
      r.x'ss<-plot(ss$points)'
      'x2'r.p x2
      pp←r.x 'ss$points'
      'xx' r.p xx
      r.x'ss<-sammon(dist(xx))'
      r.x 'plot(ss$points)'
      ⎕←pp←r.x 'ss$points'
[R matrix: 90x2 : dimnames]
       ⎕←r.x 'summary(ss$points)'
 [R table - 6 rows]
        V1                    V2
  Min.   :-16.994313    Min.   :-3.5412
  1st Qu.:-13.543887    1st Qu.:-1.1105
```

```
  Median :  0.006133    Median :-0.1200
  Mean   :  0.000000    Mean   : 0.0000
  3rd Qu.: 13.483473    3rd Qu.: 0.9875
  Max.   : 16.769582    Max.   : 6.5274
       pp←r.x 'ss$points'
      ⍴pp.Value ⍝ what I forgot it was get Value from a complex object pp
90 2
      +⌿pp.Value
¯2.675637489E¯14 ¯4.218847494E¯15
      ⌈⌿pp.Value
16.76958233 6.527426142
      ⌊⌿pp.Value
¯16.99431306 ¯3.54116532
      plt.plot pp.Value
```