In [1]:
```
#load "Angara.Charting.Paket.fsx"
#load "Angara.Charting.fsx"
```

In [2]:
```
Paket.Package ["FSharp.Data"]
```

Out[2]:
```
<null>
```

In [3]:
```
Paket.Package ["Angara.Table"]
Paket.Package ["Angara.Statistics"]
#r "packages/Angara.Table/lib/net452/Angara.Table.dll"
#r "packages/Angara.Statistics/lib/net452/Angara.Statistics.dll"
#r "packages/System.Collections.Immutable/lib/portable-net45+win8+wp8+wpa81/System.Collections.Immutable.dll"

open Angara.Data
open Angara.Charting
open Angara.Statistics
open System.Linq
```

In [4]:
```
type Stat = {
    min: float
    lb95: float
    lb68: float
    median: float
    ub68: float
    ub95: float
    max: float
    sd: float
}
let stat series =
    let summ = series |> summary
    let qsumm = series |> qsummary
    { min = summ.min; lb95 = qsumm.lb95; lb68 = qsumm.lb68; median = qsumm.median;
      ub68 = qsumm.lb68; ub95 = qsumm.ub95; max = summ.max; sd = sqrt(summ.variance) }

let colors = ["blue";"red";"green";"lightblue";"orange";"yellow";"grey";"magenta"]
```

Single VM vs Multiple VMs

Multiple VMs:

In [5]:
```
let wctime = [446.3462;438.8182;442.2185;417.7349;444.6729;436.9418;432.9028;402.9171;413.887;440.8813]
stat wctime
```

Out[5]:
```
{min = 402.9171;
 lb95 = 402.9171;
 lb68 = 413.7407347;
 median = 437.88;
 ub68 = 413.7407347;
 ub95 = 446.3462;
 max = 446.3462;
 sd = 14.90006123;}
```

In [6]:
```
let x,y = kde2 512 300.0 600.0 wctime
let plot_pdf =
    Plot.line(LineX.Values x, LineY.Values y, displayName = "Multiple VMs",
              titles = Titles.line("Duration (sec)", "Probability"))
```

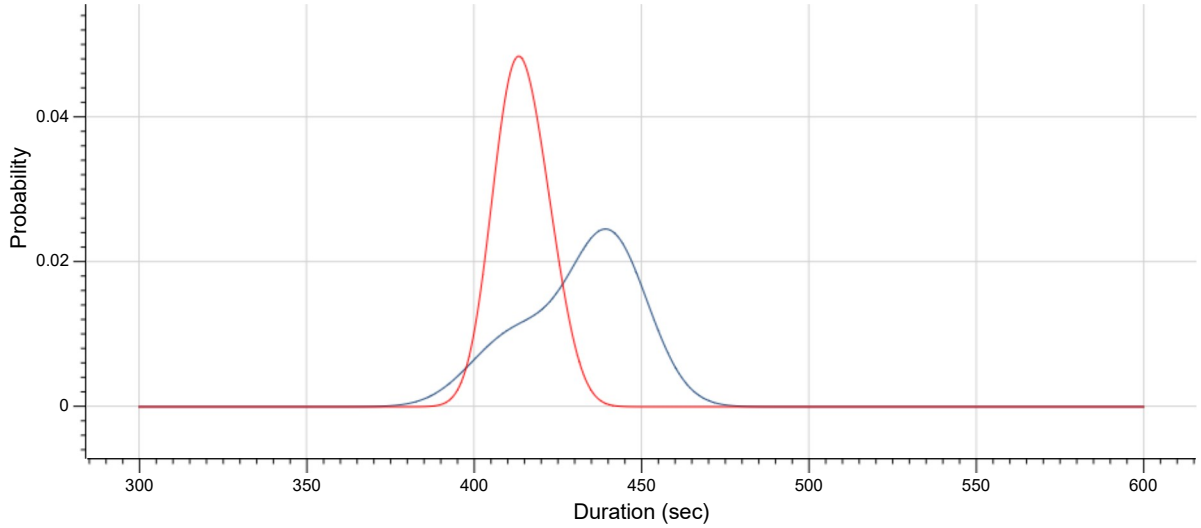
Single VM:

In [7]:
```
let wctime_same = [421.4155;408.283;409.7987;422.9174;415.5584;411.1633;413.4903;409.9151;418.3712;410.9465]

stat wctime_same
```

Out[7]:
```
{min = 408.283;
 lb95 = 408.283;
 lb68 = 409.7784907;
 median = 412.3268;
 ub68 = 409.7784907;
 ub95 = 422.9174;
 max = 422.9174;
 sd = 5.167761907;}
```

In [8]:
```
let x,y = kde2 512 300.0 600.0 wctime_same
let plot_pdf_same =
    Plot.line(LineX.Values x, LineY.Values y, stroke = "red", displayName = "Single VM",
        titles = Titles.line("Duration (sec)", "Probability"))
[ plot_pdf; plot_pdf_same ] |> Chart.ofList
```

Out[8]:



In [9]:
```
// This function returns a float array of the given table column
let col (colName:string) (t:Table) : float[] = t.[colName].Rows.AsReal.ToArray()

let rename (name:string) (c:Column) = Column.Create(name, c.Rows, c.Height)

let loadTable strContent =
    let reader = new System.IO.StringReader(strContent)
    Table.Load(reader,
               { Angara.Data.DelimitedFile.ReadSettings.Default
                   with Delimiter = Angara.Data.DelimitedFile.Delimiter.Tab })

let loadTableC strContent =
    let reader = new System.IO.StringReader(strContent)
    Table.Load(reader,
               { Angara.Data.DelimitedFile.ReadSettings.Default
                   with Delimiter = Angara.Data.DelimitedFile.Delimiter.Comma })

let schema (table:Table) = table
                           |> Seq.iteri (fun colIdx col ->
                               printfn "%d: %s of type %s" colIdx col.Name
                                           (match col.Rows with
                                           | RealColumn _    -> "float"
                                           | IntColumn _     -> "int"
                                           | StringColumn _  -> "string"
                                           | DateColumn _    -> "DateTime"
                                           | BooleanColumn _ -> "bool"))
```

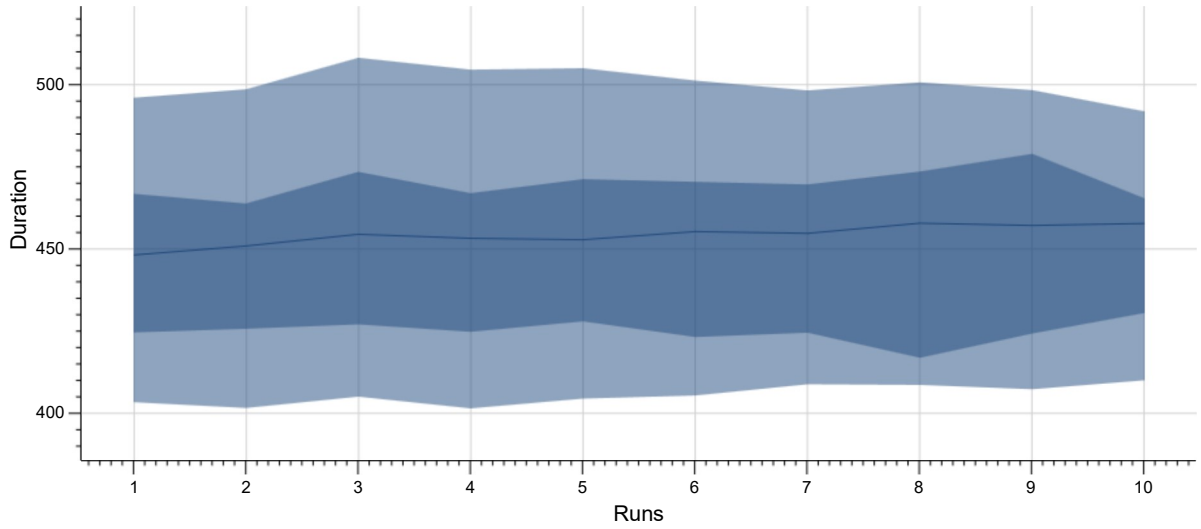
In [10]:
```
let transpose (table:Table) =
    table
    |> Seq.map(fun col -> col.Rows.AsReal.ToArray())
    |> Seq.toArray
    |> MatrixTable.OfMatrix

let takeRows n (table:Table) =
    table
    |> Seq.map(fun col -> Column.Create(col.Name, col.Rows.AsReal.ToArray() |> Array.take n))
    |> Table.OfColumns

let drawPdfColumnsRangeColored table vmin vmax (color: int -> string) =
    table
    |> Seq.map(fun (col:Column) ->
        let name = col.Name
        let vals = col.Rows.AsReal.ToArray()
        let x,y = kde2 512 vmin vmax vals
        name, x, y)
    |> Seq.mapi (fun i (name, x,y) ->
        Plot.line(LineX.Values x, LineY.Values y, stroke = color i,
            displayName = name,
            titles = Titles.line("Duration (sec)", "Probability")))
    |> Seq.toList
    |> Chart.ofList

let defaultColoring i = colors.[i%colors.Length]

let drawPdfColumnsRange table vmin vmax = drawPdfColumnsRangeColored table vmin vmax defaultColoring

let drawPdfColumns table = drawPdfColumnsRange table 300.0 600.0

let toQuantiles (qsums: qsummaryType[]) =
        { median =  qsums |> Array.map(fun q -> q.median)
          lower68 = qsums |> Array.map(fun q -> q.lb68)
          upper68 = qsums |> Array.map(fun q -> q.ub68)
          lower95 = qsums |> Array.map(fun q -> q.lb95)
          upper95 = qsums |> Array.map(fun q -> q.ub95) }

let drawRowsUncertainties table =
    let m = table |> transpose
    let unc = m.Columns |> Seq.map(fun runRes -> qsummary (runRes.ToArray())) |> Seq.toArray
    let x = Array.init m.Count (fun i -> float(i+1))
    let duration = toQuantiles unc
    [ Plot.line(LineX.Values x, LineY.UncertainValues duration,
        titles = Titles.line("Runs", "Duration"),
        displayName = "Duration Distribution") ]
    |> Chart.ofList
```

Multiple runs on multiple machines

In [11]:
```
let s = "Run #  VM0     VM1     VM2     VM3     VM4     VM5     VM6     VM7     VM8     VM9     AVERAGE STD
0       435.4531        403.5313        495.9219        462.125 466.375 448.2031        453.3906        424.9844
        448.1875        443.7344        448.1906        23.54834
1       455.1094        401.7813        498.5   457.1719        463.3594        441.5938        450.9531        4
26.1406 450.9063        431.7031        447.7219        24.203
2       457.6094        405.2031        508.0469        461.5   472.9688        443.1094        453.1719        4
27.4375 455.7656        442.2344        452.7047        25.84747
3       456.1406        401.6406        504.4531        466.4531        463.4844        441.5469        455.5938
        425.2031        450.9063        450.9063        451.6328        25.46885
4       456.6719        404.6094        504.8906        464.5938        470.7656        446.9688        454.6563
        428.375 451     450.8906        453.3422        24.80039
5       456.8594        405.5469        501.1406        463.0625        469.9844        442.6406        453.7188
        423.5469        452.7656        457.9375        452.7203        24.43953
6       456.7188        408.9688        498.1406        460.9531        469.2188        447.4375        456.1563
        424.8281        453.3594        451.4375        452.7219        22.70002
7       458.75  408.7656        500.6094        461.3438        473.1875        444.1875        456.9219        4
17.125  452.9375        459.4688        453.3297        24.808
8       456.5781        407.4531        498.2344        462.3438        478.6563        446.5781        457.7188
        424.6094        455.8125        461.0625        454.9047        24.05242
9       462.6406        410.1875        491.7813        460.5469        465.0313        450.1094        456.2969
        430.8906        459.2188        456.0781        454.2781        20.42402
"
let tableF = s |> loadTable |> Seq.filter(fun c -> c.Name.StartsWith "VM") |> Table.OfColumns
```

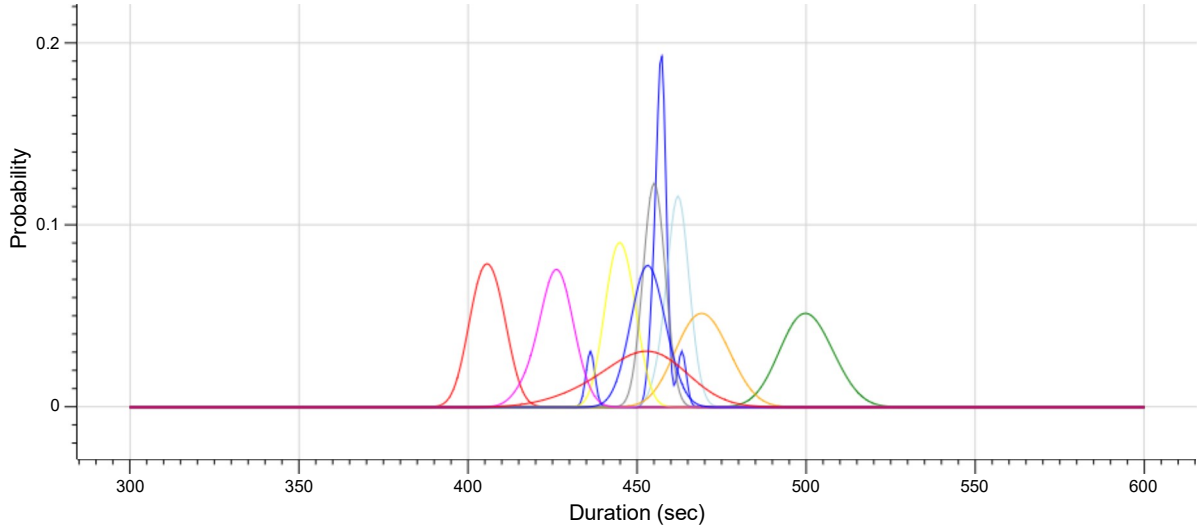
In [12]:
```
drawRowsUncertainties tableF
```

Out[12]:



In [13]:
```
drawPdfColumns tableF
```

Out[13]:



1. Each of the VMs shows quite stable result; standard deviation is about 5 sec.
2. Different VMs have different stable results (location of peaks); it varies between 400 and 500 sec.

Then:

1. How to make the results across different machines consistent?
2. What is the reason of different performance of machines?

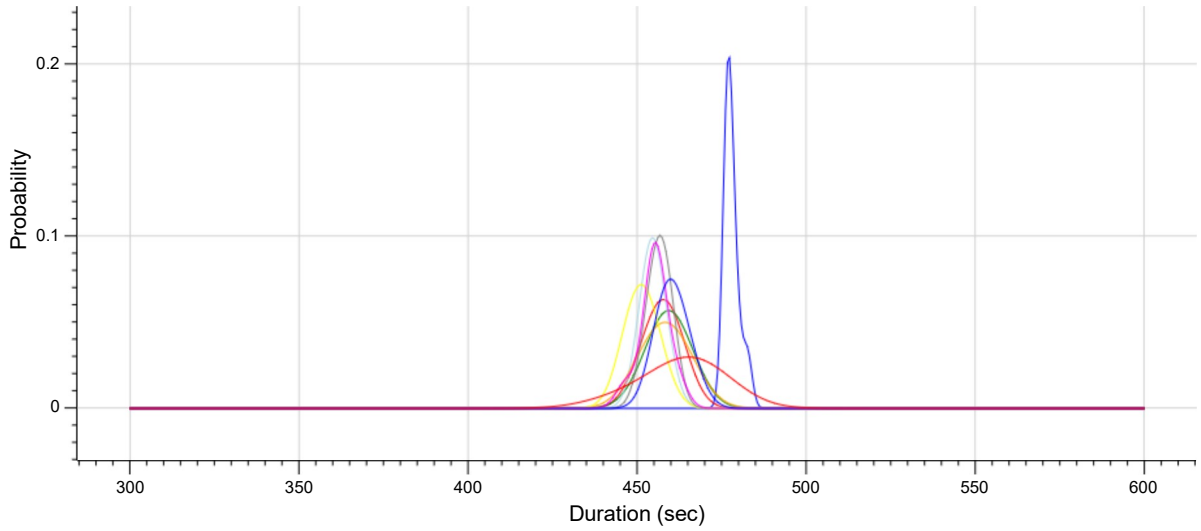
Normalization

Now, if we consider 454.8578 to be 'standard' time, use the first measurement on each VM to compute 'performance coefficient' (standard time / measurement) and multiply other measurement by that coefficient, we'll get

In [14]:
```
let s = "VM0     VM1     VM2     VM3     VM4     VM5     VM6     VM7     VM8     VM9
475.39  452.8852        457.2224        449.9826        451.9166        448.1503        452.4124        456.0953
        457.617 442.525
478.0014        456.7423        465.9788        454.2426        461.2887        449.6884        454.6383        4
57.4834 462.5487        453.3202
476.4672        452.7267        462.6826        459.1179        452.0386        448.1027        457.0681        4
55.0919 457.617 462.2095
477.0221        456.0731        463.0839        457.2877        459.14  453.6051        456.1275        458.4868
        457.7122        462.1934
477.218 457.1298        459.6444        455.7806        458.378 449.2127        455.187 453.3193        459.5041
        469.417
477.0711        460.9869        456.8928        453.7044        457.6313        454.0808        457.6324        4
54.6906 460.1066        462.754
479.1928        460.758 459.1572        454.0888        461.5021        450.7826        458.4005        446.446 4
59.6785 470.9866
476.9242        459.2785        456.9788        455.0731        466.8358        453.2087        459.1999        4
54.4564 462.5963        472.6203
483.2568        462.3607        451.06  453.3045        453.5472        456.7924        457.7735        461.1792
        466.0532        467.511
"
let tableFCoeff = s |> loadTable
```

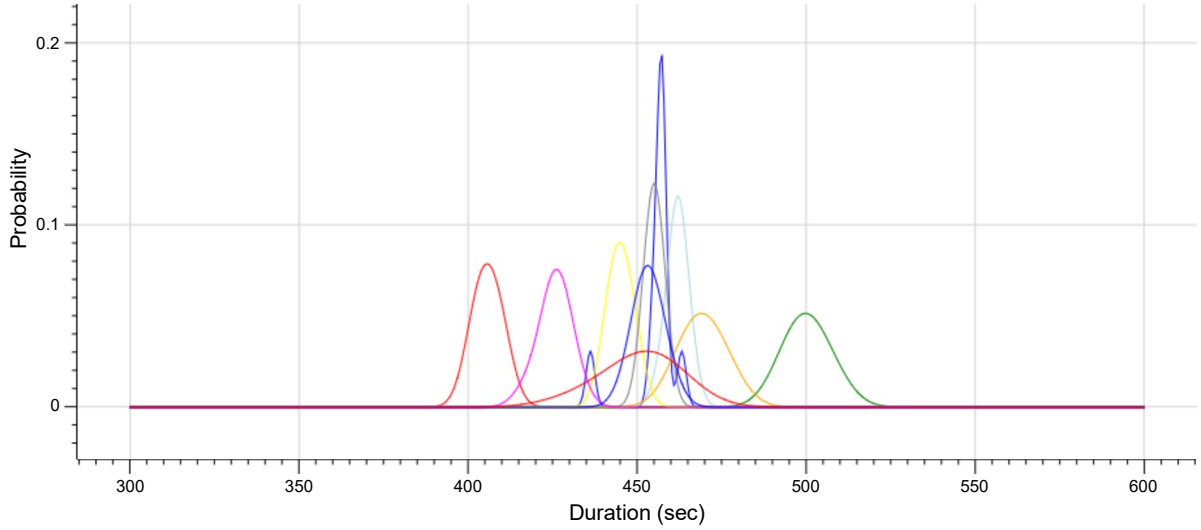
In [15]:
```
drawPdfColumns tableFCoeff
```

Out[15]:



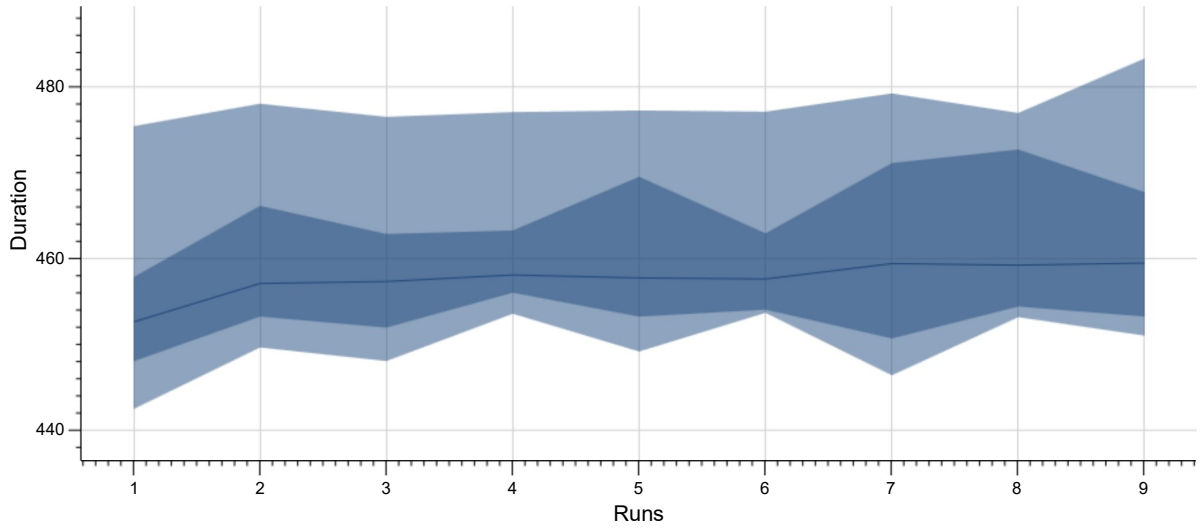
In [16]:
```
drawPdfColumns tableF
```

Out[16]:



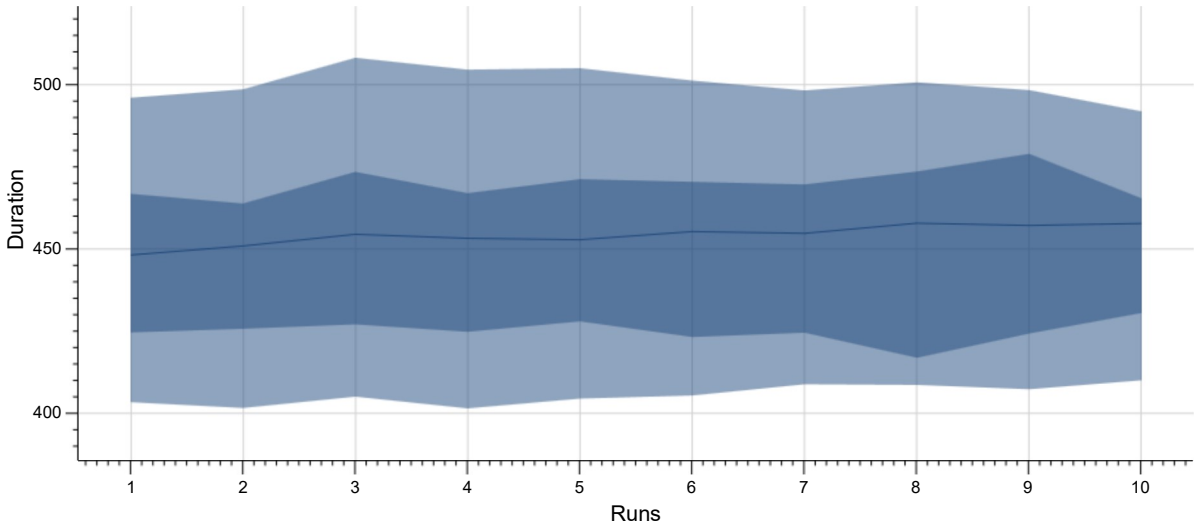
In [17]:
```
drawRowsUncertainties tableFCoeff
```

Out[17]:



In [18]:
```
drawRowsUncertainties tableF
```

Out[18]:

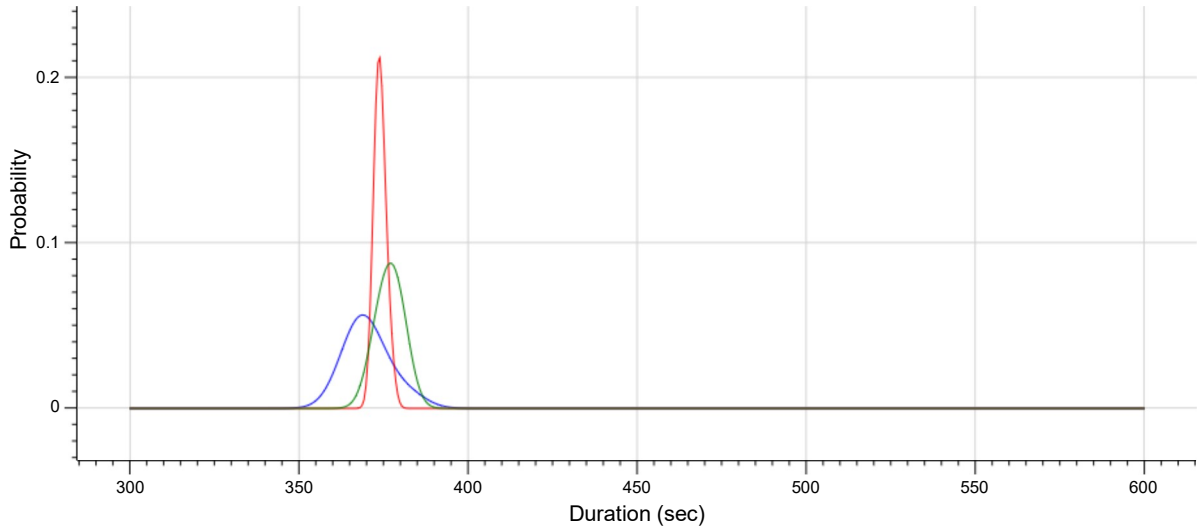


Effect of sharing same physical machine

In [19]:
```
let share_1 = "Run #    Result code     Return value    CPUTime WCTime  Memory
0       OK      0       374.46875       374.4983537     448053248
1       OK      0       373.328125      373.3428245     462352384
2       OK      0       372.171875      372.1924532     456261632
3       OK      0       372.453125      372.4779358     456318976
4       OK      0       373.375 373.3988532     454131712
5       OK      0       374.25  374.2604102     462417920
6       OK      0       374.796875      374.8168565     456237056
7       OK      0       377.890625      377.9116777     454197248
8       OK      0       373.453125      373.4841731     462446592
9       OK      0       372.265625      372.2689823     454057984
"
let share_2 = "Run #    Result code     Return value    CPUTime WCTime  Memory
0       OK      0       379.171875      383.9745942     448053248
1       OK      0       376.59375       382.5917482     462352384
2       OK      0       374.75  379.3451419     456261632
3       OK      0       372.609375      378.4968077     456318976
4       OK      0       376.25  381.1666909     454131712
5       OK      0       378.109375      382.640683      462417920
6       OK      0       378.0625        382.2941934     454144000
7       OK      0       381.109375      384.8785986     462446592
8       OK      0       378.03125       381.9148036     456298496
9       OK      0       372.3125        376.4337885     454025216
"
let excl = "Run #       Result code     Return value    CPUTime WCTime  Memory
0       OK      0       368.421875      374.9458616     232374272
1       OK      0       369.203125      375.735785      232259584
2       OK      0       370.140625      377.652724      232243200
3       OK      0       374.34375       378.1208825     230219776
4       OK      0       368.15625       373.7346988     230203392
5       OK      0       367.046875      371.729211      232296448
6       OK      0       362.859375      365.9769345     230060032
7       OK      0       376.5   380.5797761     232366080
8       OK      0       369.125 372.6886823     230232064
9       OK      0       383.25  389.6150868     230236160
"

let takeAndRename oldName newName table =
    table
    |> Seq.choose(fun (c:Column) -> if c.Name = oldName then Some(c |> rename newName) else None)
    |> Seq.exactlyOne

let cpu_share1 = loadTable share_1 |> takeAndRename "CPUTime" "Shared 1"
let cpu_share2 = loadTable share_2 |> takeAndRename "CPUTime" "Shared 2"
let cpu_Excl = loadTable excl |> takeAndRename "CPUTime" "Exclusive"
let table_F = [ cpu_Excl; cpu_share1; cpu_share2 ] |> Table.OfColumns
```

In [20]:
```
drawPdfColumns table_F
```

Out[20]:



Running experiment one per physical machine and then two simultaneously on different cores, shows that stuff going on physical cores other than those we run experiment on **does not affect the result** any more than random factors within the machine.

In [21]:
```
let task1 = @"run # \ VM #,0,1,2,3,4,5,6,7,8,9
0,0.953125,0.921875,0.921875,1.03125,1.171875,0.828125,0.9375,0.9375,1.078125,0.9375
1,0.90625,0.921875,0.890625,0.9375,0.984375,0.828125,0.921875,0.921875,0.96875,0.9375
2,0.90625,0.90625,0.90625,0.9375,1.046875,0.890625,0.921875,0.90625,1,0.90625
3,0.890625,0.9375,0.875,0.9375,1.03125,0.890625,0.921875,0.96875,0.984375,0.90625
4,0.890625,0.921875,0.84375,0.96875,1.03125,0.84375,0.9375,0.984375,1,0.9375
5,0.859375,0.9375,0.84375,0.953125,1.015625,0.890625,0.9375,0.9375,1,0.9375
6,0.828125,0.9375,0.859375,0.96875,1.109375,0.875,0.921875,1,0.984375,0.921875
7,0.875,0.9375,0.859375,0.96875,1.09375,0.8125,0.9375,0.90625,1,0.90625
8,0.96875,0.90625,0.828125,0.9375,1.03125,0.828125,0.921875,0.90625,1.015625,0.9375
9,0.9375,0.9375,0.875,0.9375,1.03125,0.890625,0.9375,0.953125,1.03125,0.9375"
let task2 = @"run # \ VM #,0,1,2,3,4,5,6,7,8,9
0,1.140625,1.1875,1.125,1.203125,1.296875,1.140625,1.203125,1.1875,1.25,1.171875
1,1.109375,1.171875,1.09375,1.28125,1.28125,1.046875,1.203125,1.203125,1.203125,1.171875
2,1.15625,1.203125,1.078125,1.203125,1.296875,1.125,1.203125,1.28125,1.296875,1.15625
3,1.140625,1.15625,1.125,1.1875,1.265625,1.15625,1.1875,1.21875,1.234375,1.203125
4,1.1875,1.15625,1.09375,1.1875,1.390625,1.171875,1.203125,1.265625,1.234375,1.1875
5,1.171875,1.15625,1.109375,1.1875,1.296875,1.234375,1.1875,1.171875,1.21875,1.171875
6,1.1875,1.1875,1.125,1.296875,1.296875,1.140625,1.171875,1.171875,1.234375,1.125
7,1.15625,1.171875,1.109375,1.125,1.28125,1.1875,1.21875,1.1875,1.234375,1.140625
8,1.21875,1.15625,1.125,1.140625,1.4375,1.09375,1.1875,1.28125,1.265625,1.171875
9,1.109375,1.15625,1.125,1.15625,1.25,1.171875,1.1875,1.234375,1.21875,1.140625"
let task3 = @"run # \ VM #,0,1,2,3,4,5,6,7,8,9
0,533.375,558.8125,541.640625,588.9375,624.796875,545.234375,566.734375,580.078125,603.640625,559.015625
1,536.359375,505.3125,565.53125,585.140625,628.78125,546.609375,576.328125,575.625,592.15625,564.09375
2,567.21875,502.84375,553.9375,575.484375,632.53125,541.140625,575.25,576.03125,605.640625,565.03125
3,577.578125,501.46875,530.265625,585.71875,633.140625,541.03125,572.34375,580.34375,606.515625,552.3125
4,577.671875,497.703125,536.75,581.953125,627.671875,546.796875,569.1875,572.015625,599.296875,557.96875
5,576.890625,499.046875,562.15625,587.375,616.109375,550.90625,570.984375,577.375,593.890625,558.5
6,567.625,503.1875,577.28125,587.828125,620.46875,536.453125,569.5,571.4375,592.46875,555.21875
7,585.0625,496.28125,576.859375,590.546875,628.203125,540.09375,567.484375,572.453125,588.828125,553.9375
8,588.640625,558.125,548.203125,589.15625,628.484375,535.15625,569.265625,576.140625,600.1875,562.296875
9,586.875,560.171875,557.015625,595.96875,621.3125,540.234375,567.6875,570.578125,591.90625,560.109375"
let task4 = @"run # \ VM #,0,1,2,3,4,5,6,7,8,9
0,95.359375,92.421875,93.140625,94.5,103.078125,88.28125,91.703125,93.75,95.75,91.484375
1,91.484375,91.15625,92.296875,95.203125,103.953125,85.453125,91.734375,92.265625,95.875,90.65625
2,88.96875,92.203125,91,96.359375,102.421875,86.09375,91.8125,94.140625,96.171875,91.09375
3,95.515625,93,89.296875,97.859375,101.890625,86.59375,92.984375,93.46875,96.4375,89.234375
4,95.40625,91.5,89,97.578125,101.234375,86.265625,92.609375,93.625,97.8125,90.921875
5,95.3125,92.0625,90.875,95.921875,102.890625,87.9375,92.65625,93.546875,96.65625,91.859375
6,95.34375,92.421875,91.328125,96.75,101.6875,89.359375,92.84375,93.765625,97.1875,90.515625
7,95.734375,92.03125,92.890625,94.9375,102.21875,87.734375,92.796875,93.984375,97.609375,92.328125
8,94.046875,92.21875,94.5,96.8125,102.4375,87.9375,92.625,92.125,96.734375,91.96875
9,95.25,92.453125,94,95.671875,102.015625,86.5625,93.4375,92.1875,96.53125,92.890625"
let task5 = @"run # \ VM #,0,1,2,3,4,5,6,7,8,9
0,446.390625,411.859375,436.640625,450.1875,466.421875,405.84375,430.984375,430.375,449.0625,427.65625
1,452.15625,379.15625,430.9375,449.546875,467.84375,418.71875,429.453125,433.671875,448.921875,426.046875
2,448.21875,371.671875,409.421875,448.421875,471,413.71875,425.171875,428.71875,442.125,422.1875
3,428.671875,375.234375,405.96875,448.671875,490.03125,417.71875,424.8125,432.90625,443.953125,425.5625
4,434.078125,376.421875,429.765625,446.140625,481.6875,418.234375,431.578125,427.78125,457.625,421.96875
5,429.90625,374.75,409.578125,435.96875,479,415.734375,428.09375,428.65625,449.5625,427.75
6,431.9375,372.234375,401.0625,437.796875,483.734375,409.828125,429.75,432.28125,454.84375,425.625
7,435.203125,375.96875,416.140625,434.875,482.46875,410.03125,427.75,435.015625,455.15625,424.984375
8,430.328125,371,434.609375,436.796875,481.421875,409.375,425.40625,430.46875,449.71875,421.59375
9,432.265625,377.484375,430.5625,437.296875,475.78125,407.859375,428.9375,430.25,448.453125,422.0625"

let tableT1 = task1 |> loadTableC |> Seq.filter(fun c -> not (c.Name.StartsWith "run")) |> Table.OfColumns
let tableT2 = task2 |> loadTableC |> Seq.filter(fun c -> not (c.Name.StartsWith "run")) |> Table.OfColumns
let tableT3 = task3 |> loadTableC |> Seq.filter(fun c -> not (c.Name.StartsWith "run")) |> Table.OfColumns
let tableT4 = task4 |> loadTableC |> Seq.filter(fun c -> not (c.Name.StartsWith "run")) |> Table.OfColumns
let tableT5 = task5 |> loadTableC |> Seq.filter(fun c -> not (c.Name.StartsWith "run")) |> Table.OfColumns
```

In [22]:
```
let allTables = [| tableT1; tableT2; tableT3; tableT4; tableT5 |]
let uncertaintyR data = let s = stat data in s.sd / s.median
let uncertaintyP data = (uncertaintyR data) * 100.0
allTables |> Array.map (fun t -> t |> Array.ofSeq |> Array.map (fun c -> uncertaintyP c.Rows.AsReal) |> Array.red
uce max)
```

Out[22]:
```
[|5.309039404; 4.649950274; 5.614944324; 2.35291249; 3.189734557|]
```

In [23]:
```
//let averages3T1 = tableT1 |> Seq.map (fun c -> c.Rows.AsReal |> Seq.mapi (fun i v -> (v + c.Rows.AsReal.[(i + 1
) % c.Height] + c.Rows.AsReal.[(i + 2) % c.Height]) / 3.0) |> Array.ofSeq) |> Array.ofSeq
//let coefs1 = averages3T1 |> Array.map (fun a -> a.[0])
let coefs1 = tableT1 |> Seq.map (fun c -> 0.9375 / (qsummary c.Rows.AsReal).median) |> Array.ofSeq
coefs1
```

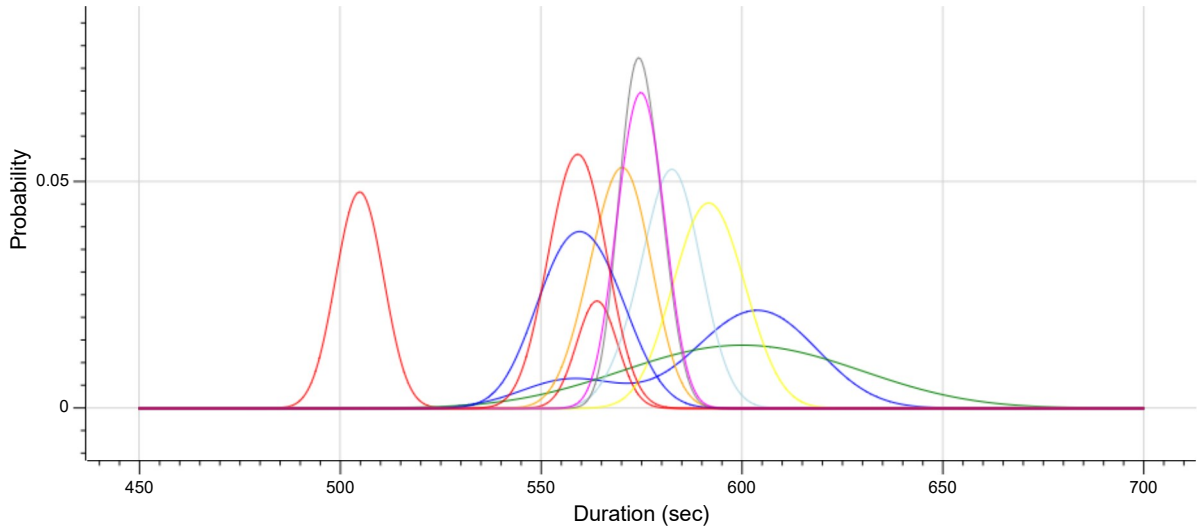
Out[23]:
```
[|1.043478261; 1.008403361; 1.081081081; 0.9917355372; 0.9090909091; 1.090909091;
  1.008403361; 1.0; 0.9375; 1.0|]
```

In [24]:
```
let tableT3a = tableT3 |> Seq.mapi (fun i c -> c.Rows.AsReal |> Seq.map (fun v -> v * coefs1.[i]) |> Array.ofSeq
) |> Array.ofSeq
let tT3a = tableT3 |> Seq.mapi (fun i c -> Column.Create (c.Name, c.Rows.AsReal |> Seq.map (fun v -> v * coefs1.[
i]) )) |> List.ofSeq |> Table.OfColumns
tT3a
```

Out[24]:
```
seq
  [0[10]: Array is not evaluated yet; 1[10]: Array is not evaluated yet;
   2[10]: Array is not evaluated yet; 3[10]: Array is not evaluated yet; ...]
```

In [25]:
```
drawPdfColumnsRange tT3a 450.0 700.0
```

Out[25]:



Analysis of relative standard deviation of medians among multiple runs of a test

Here we run a small problem for 2000 times on each of 10 VMs of our cluster. Then, we combine runs within each machine into equal sized groups (each run belongs to only one group), compute median value for each group and then compute relative standard deviation of those median value. This is done separately for each VM. So, for every group size (ranging from 1 to 200) we obtain 10 relative standard deviations (one per VM). We visualize them as a band plot.

In [26]:
```
let tableSmalls = Table.Load ("smalls2k.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = A
ngara.Data.DelimitedFile.Delimiter.Tab })
//tableSmalls
let smalls = [0..9] |> List.map (fun i -> Column.Create (i.ToString(), (tableSmalls |> Table.Filter ["Task #"] (f
un (x: float) -> x = float i)).["CPUTime"].Rows.AsReal)) |> Table.OfColumns
smalls |> Array.ofSeq |> Array.map (fun c -> uncertaintyP c.Rows.AsReal)
```

Out[26]:
```
[|4.401407015; 2.677162547; 5.327866042; 4.377607359; 4.769894385; 2.934385546;
  3.529012758; 4.642639781; 3.785091269; 6.074913539|]
```

In [27]:
```
let blur k (data: System.Collections.Generic.IList<float>) =
    let c = data.Count
    [| 0 .. c - 1 |] |> Array.map (fun i -> ([| i .. i + k - 1 |] |> Array.map (fun j -> data.[j % c]) |> qsummar
y).median)

let blurT k (table: Table) = table |> Seq.mapi (fun i c -> Column.Create (c.Name, blur k c.Rows.AsReal)) |> List.
ofSeq |> Table.OfColumns
```

In [28]:
```
let smallsB = blurT 10 smalls
smallsB |> Array.ofSeq |> Array.map (fun c -> uncertaintyP c.Rows.AsReal)
```

Out[28]:
```
[|1.558699626; 1.432006232; 2.249972518; 1.214148978; 2.357909631; 1.304095235;
  1.318277488; 1.639993701; 1.372957626; 2.332889617|]
```

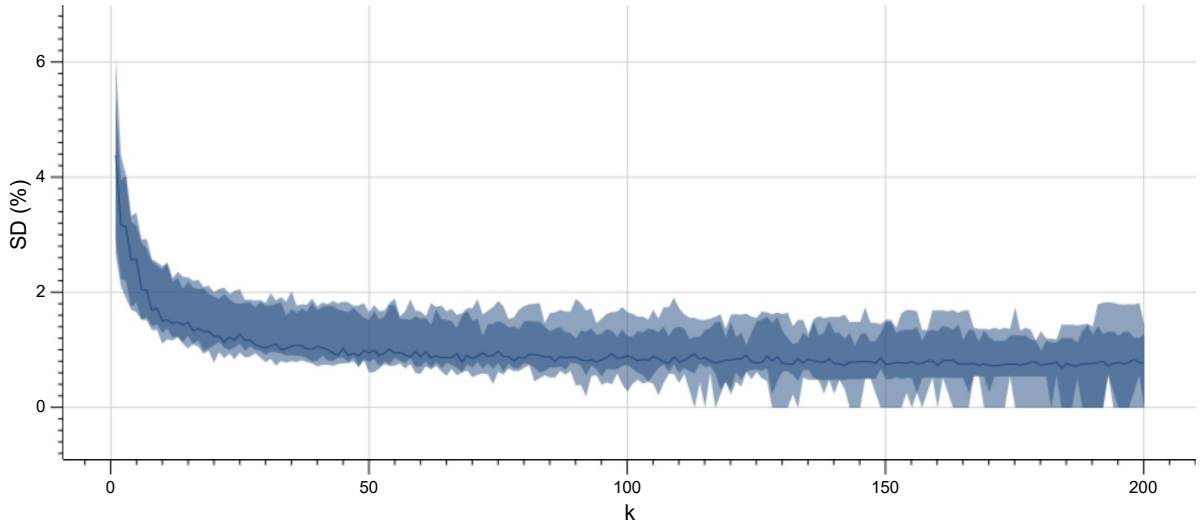
In [29]:
```
let blurGroups k (data: System.Collections.Generic.IList<float>) =
    let c = data.Count
    [| 0 .. c / k - 1 |] |> Array.map (fun i -> ([| i * k .. i * k + k - 1 |] |> Array.map (fun j -> data.[j % c]
) |> qsummary).median)

let blurGroupsT k (table: Table) = table |> Seq.mapi (fun i c -> Column.Create (c.Name, blurGroups k c.Rows.AsRea
l)) |> List.ofSeq |> Table.OfColumns
```

In [30]:
```
let smallsBG = blurGroupsT 50 smalls
smallsBG |> Array.ofSeq |> Array.map (fun c -> uncertaintyP c.Rows.AsReal)
```

Out[30]:
```
[|0.9653347394; 1.065755067; 1.426853362; 0.7969931839; 1.658535029;
  0.6171766131; 0.8996990601; 0.9435641951; 0.9162086768; 1.515441649|]
```

In [31]:
```
let smallsBGsd k = blurGroupsT k smalls |> Seq.map (fun c -> uncertaintyP c.Rows.AsReal) |> qsummary
//let smallsBGarr = [|1..200|] |> Array.map (fun i -> smallsBGsd i)

//let smallsBGquants = toQuantiles smallsBGarr
[Plot.line(LineX.Values ([|1..200|] |> Array.map float),
           LineY.UncertainValues ([|1..200|] |> Array.map (fun i -> smallsBGsd i) |> toQuantiles),
           titles = Titles.line("k", "SD (%)")) ] |> Chart.ofList
```

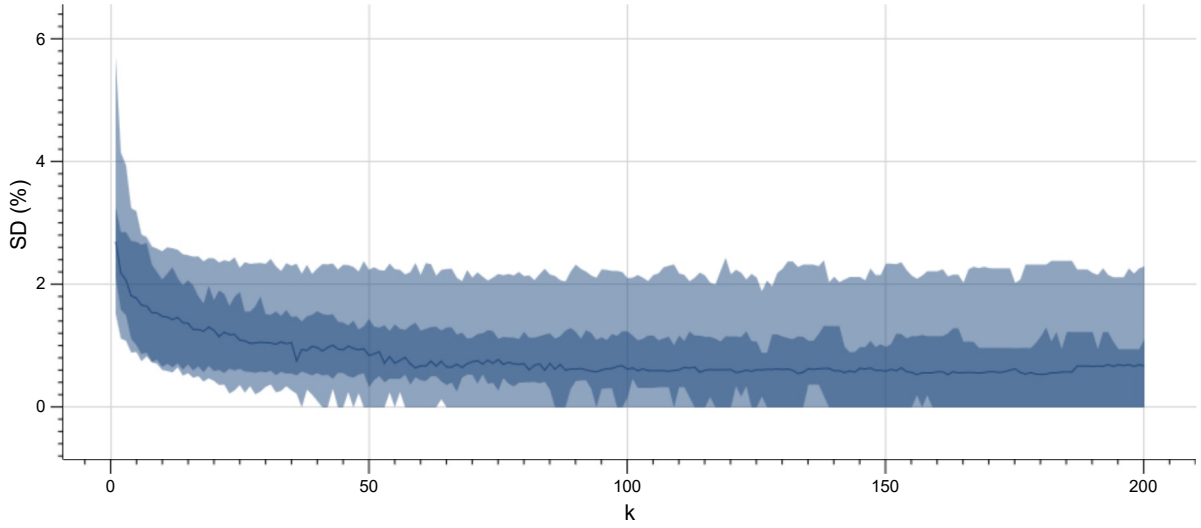
Out[31]:



Dependence between relative standard deviation of median values of the groups of test runs and the size of the group. Plot's uncertainty (band) shows range of relative SD obtained on different VMs.

In [32]:
```
let tableSmalls2 = Table.Load ("smalls2k2.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter =
 Angara.Data.DelimitedFile.Delimiter.Tab })
//tableSmalls
let smalls2 = [0..9] |> List.map (fun i -> Column.Create (i.ToString(), (tableSmalls2 |> Table.Filter ["Task #"]
(fun (x: float) -> x = float i)).["CPUTime"].Rows.AsReal)) |> Table.OfColumns
let smalls2BGsd k = blurGroupsT k smalls2 |> Seq.map (fun c -> uncertaintyP c.Rows.AsReal) |> qsummary
//let smallsBGarr = [|1..200|] |> Array.map (fun i -> smallsBGsd i)

//let smallsBGquants = toQuantiles smallsBGarr
[Plot.line(LineX.Values ([|1..200|] |> Array.map float),
           LineY.UncertainValues ([|1..200|] |> Array.map (fun i -> smalls2BGsd i) |> toQuantiles),
           titles = Titles.line("k", "SD (%)")) ] |> Chart.ofList
```

Out[32]:



Same but for a different test task.

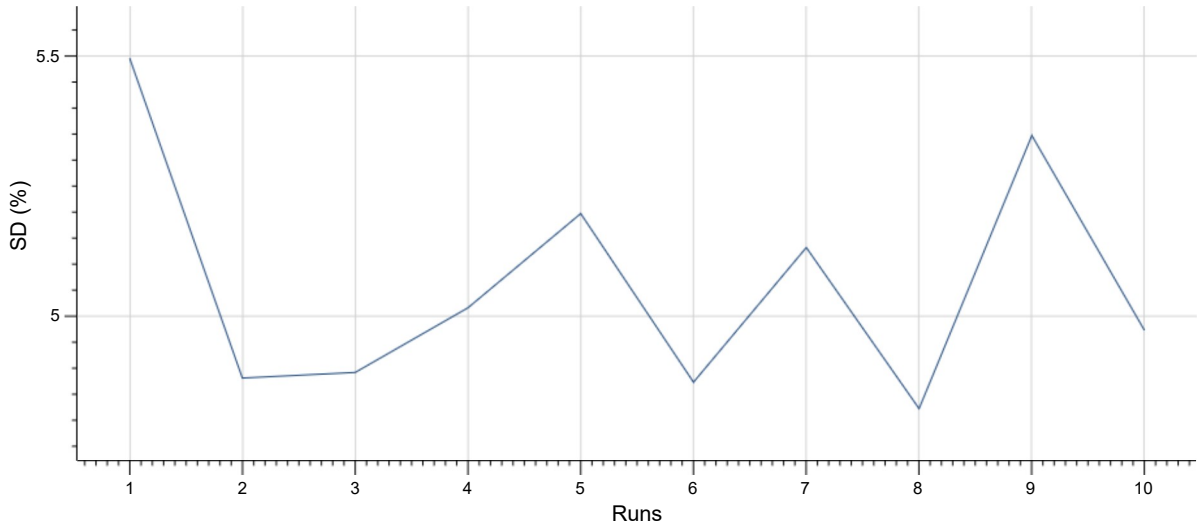
In [33]:
```
let drawRowsUncertaintiesP table =
    let m = table |> transpose
    let unc = m |> Seq.map (fun runRes -> uncertaintyP (runRes.Rows.AsReal)) |> Seq.toArray
    let x = Array.init m.Count (fun i -> float(i+1))
    //let duration = toQuantiles unc
    [ Plot.line(LineX.Values x, LineY.Values unc,
        titles = Titles.line("Runs", "SD (%)"),
        displayName = "SD / Median (%)") ]
    |> Chart.ofList
```

In [34]:
```
let tableExp1 = Table.Load ("exp1.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara.
Data.DelimitedFile.Delimiter.Tab })
//tableSmalls
let smallTask = [0..9] |> List.map (fun i -> Column.Create (i.ToString(), (tableExp1 |> Table.Filter ["Task #"; "
File #"] (fun (x: float) (y: float) -> x = float i && y = 0.0)).["CPUTime"].Rows.AsReal)) |> Table.OfColumns
let largeTask = [0..9] |> List.map (fun i -> Column.Create (i.ToString(), (tableExp1 |> Table.Filter ["Task #"; "
File #"] (fun (x: float) (y: float) -> x = float i && y = 1.0)).["CPUTime"].Rows.AsReal)) |> Table.OfColumns

let etalonicTime = 0.9375

let coefByN n = blurGroupsT n smallTask |> Seq.map (fun c -> etalonicTime / c.Rows.AsReal.[0]) |> Array.ofSeq
let coefs = coefByN 30

let adjustedLargeTask = largeTask |> Seq.mapi (fun i c -> Column.Create (c.Name, c.Rows.AsReal |> Seq.map (fun v
-> v * coefs.[i]) )) |> List.ofSeq |> Table.OfColumns

adjustedLargeTask.[9].Rows.AsReal |> Array.ofSeq
```

Out[34]:
```
[|95.19886364; 96.35795455; 96.06818182; 95.30113636; 94.53409091; 95.25;
  95.76136364; 96.13636364; 95.53977273; 96.46022727|]
```

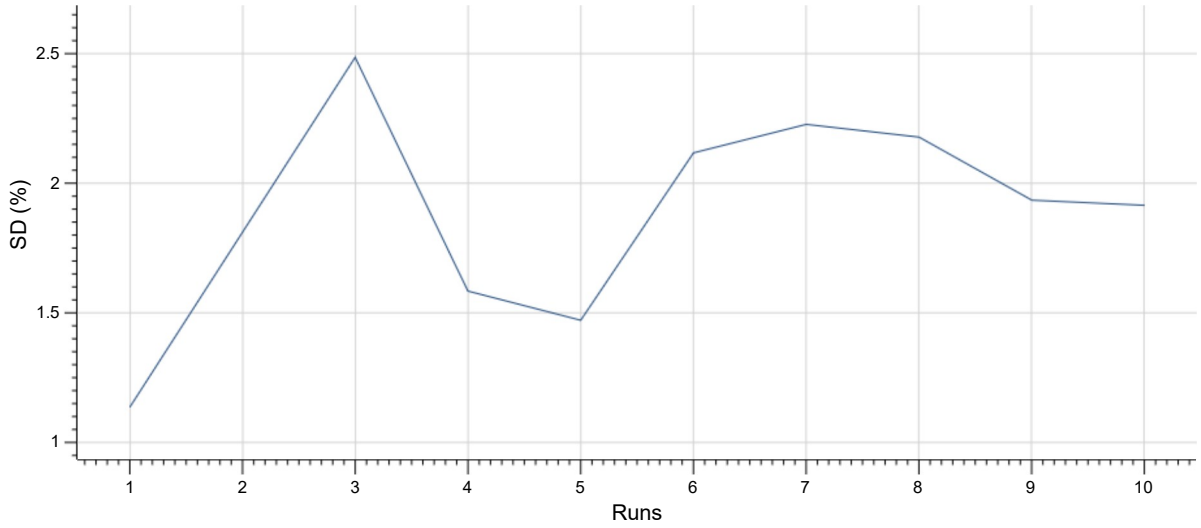
In [35]:
```
drawRowsUncertaintiesP largeTask
```

Out[35]:



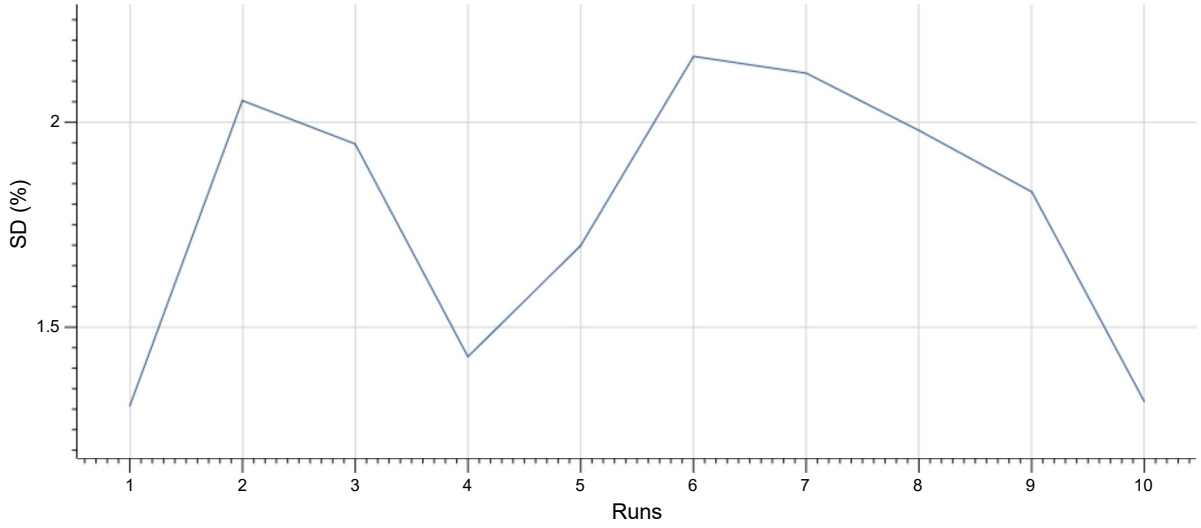
In [36]:
```
drawRowsUncertaintiesP adjustedLargeTask
```

Out[36]:



In [37]:
```
let largeTaskB = blurT 2 largeTask
let adjustedLargeTaskB = largeTaskB |> Seq.mapi (fun i c -> Column.Create (c.Name, c.Rows.AsReal |> Seq.map (fun
v -> v * coefs.[i]) )) |> List.ofSeq |> Table.OfColumns
drawRowsUncertaintiesP adjustedLargeTaskB
```

Out[37]:

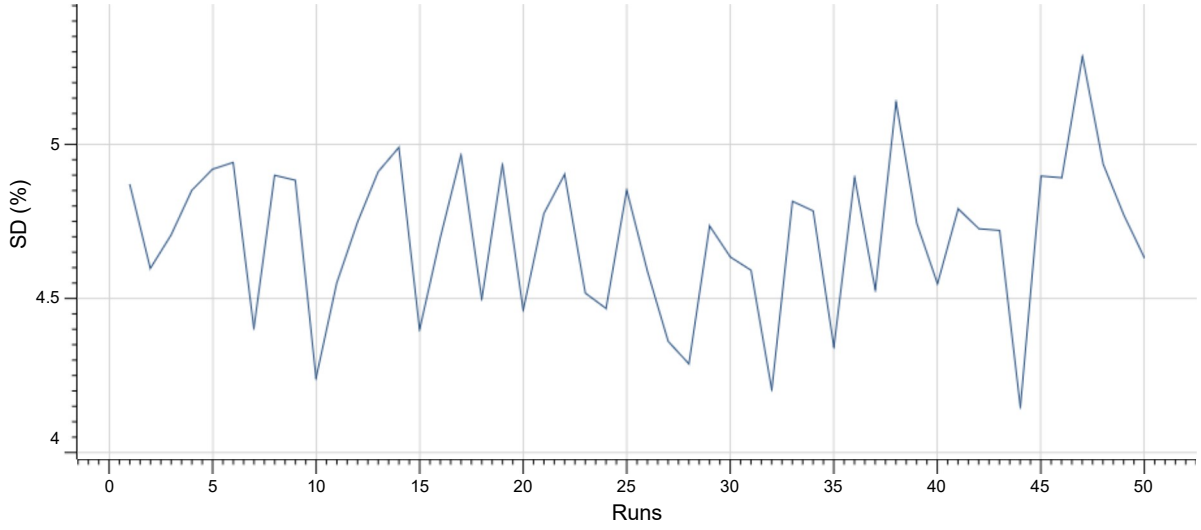


Computing Performance Coefficient of a VM using median result of 30 test task runs

In [38]:
```
let tableExp2 = Table.Load ("exp2.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara.
Data.DelimitedFile.Delimiter.Tab })
//tableSmalls
let smallTask2 = [0..9] |> List.map (fun i -> Column.Create (i.ToString(), (tableExp2 |> Table.Filter ["Task #";
"File #"] (fun (x: float) (y: float) -> x = float i && y = 0.0)).["CPUTime"].Rows.AsReal)) |> Table.OfColumns
let largeTask2 = [0..9] |> List.map (fun i -> Column.Create (i.ToString(), (tableExp2 |> Table.Filter ["Task #";
"File #"] (fun (x: float) (y: float) -> x = float i && y = 1.0)).["CPUTime"].Rows.AsReal)) |> Table.OfColumns

let etalonicTime2 = 0.9375

let coefByN2 n = blurGroupsT n smallTask2 |> Seq.map (fun c -> etalonicTime2 / c.Rows.AsReal.[0]) |> Array.ofSeq
let coefs2 = coefByN2 30

let adjustedLargeTask2 = largeTask2 |> Seq.mapi (fun i c -> Column.Create (c.Name, c.Rows.AsReal |> Seq.map (fun
v -> v * coefs2.[i]) )) |> List.ofSeq |> Table.OfColumns
```

In [39]:
```
drawRowsUncertaintiesP largeTask2
```

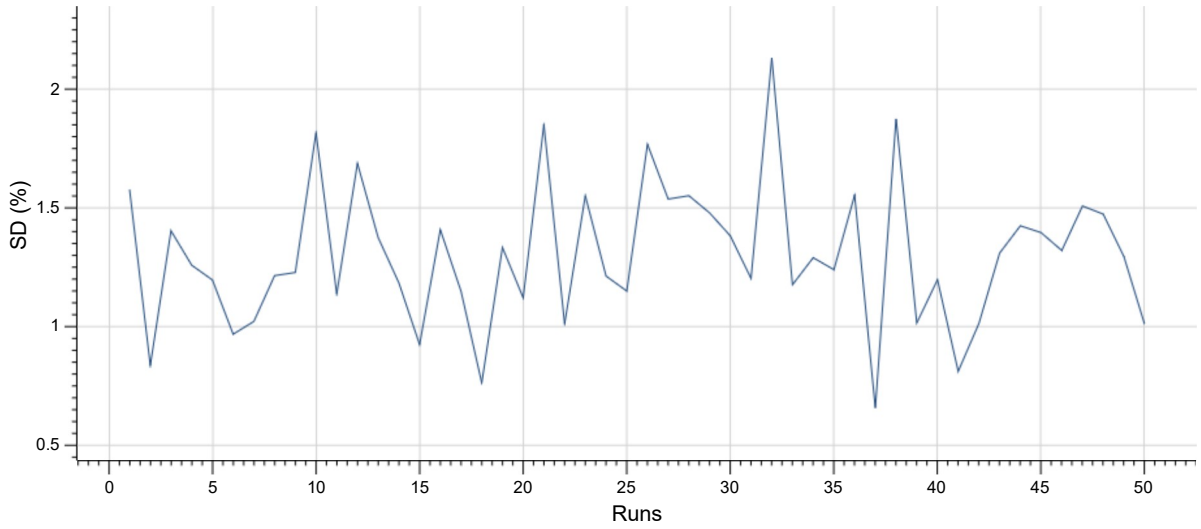
Out[39]:



Relative uncertainty between different VMs.

In [40]:
```
drawRowsUncertaintiesP adjustedLargeTask2
```

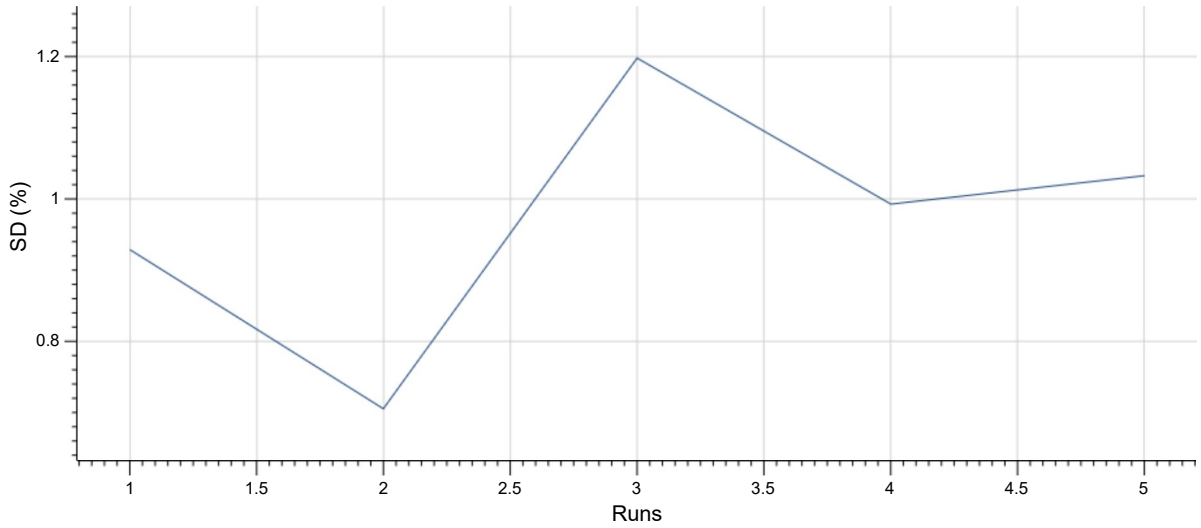
Out[40]:



Relative uncertainty between different VMs after application of performance coefficient. About 2-3 times lower then it used to be.

In [41]:
```
let largeTaskB2 = blurGroupsT 10 largeTask2
let adjustedLargeTaskB2 = largeTaskB2 |> Seq.mapi (fun i c -> Column.Create (c.Name, c.Rows.AsReal |> Seq.map (fu
n v -> v * coefs2.[i]) )) |> List.ofSeq |> Table.OfColumns
drawRowsUncertaintiesP adjustedLargeTaskB2
```

Out[41]:

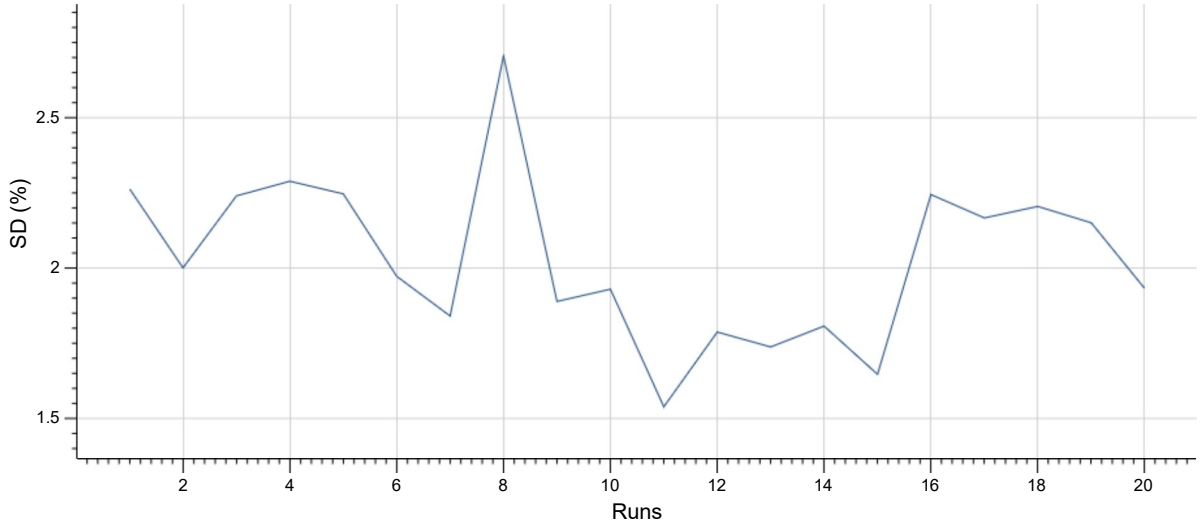


Using Performance Coefficient to Compare Results Obtained on VMs of Different Types (A2 and D2_V2)

In [42]:
```
let tableExp3 = Table.Load ("exp3a.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara
.Data.DelimitedFile.Delimiter.Tab })
//tableSmalls
let smallTask3 = [0..9] |> List.map (fun i -> Column.Create (i.ToString(), (tableExp3 |> Table.Filter ["Task #";
"File #"] (fun (x: float) (y: float) -> x = float i && y = 0.0)).["CPUTime"].Rows.AsReal)) |> Table.OfColumns
let largeTask3 = [0..9] |> List.map (fun i -> Column.Create (i.ToString(), (tableExp3 |> Table.Filter ["Task #";
"File #"] (fun (x: float) (y: float) -> x = float i && y = 1.0)).["CPUTime"].Rows.AsReal)) |> Table.OfColumns

let etalonicTime3 = 0.9375

let coefByN3 n = blurGroupsT n smallTask3 |> Seq.map (fun c -> etalonicTime3 / c.Rows.AsReal.[0]) |> Array.ofSeq
let coefs3 = coefByN3 30

let adjustedLargeTask3 = largeTask3 |> Seq.mapi (fun i c -> Column.Create (c.Name, c.Rows.AsReal |> Seq.map (fun
v -> v * coefs3.[i]) )) |> List.ofSeq |> Table.OfColumns
```

```
In [43]: let shortenedAdjLarge2 = takeRows 20 adjustedLargeTask2

         let diffMachinesLargeTask = shortenedAdjLarge2 |> Seq.append adjustedLargeTask3 |> Table.OfColumns

         drawRowsUncertaintiesP diffMachinesLargeTask
```

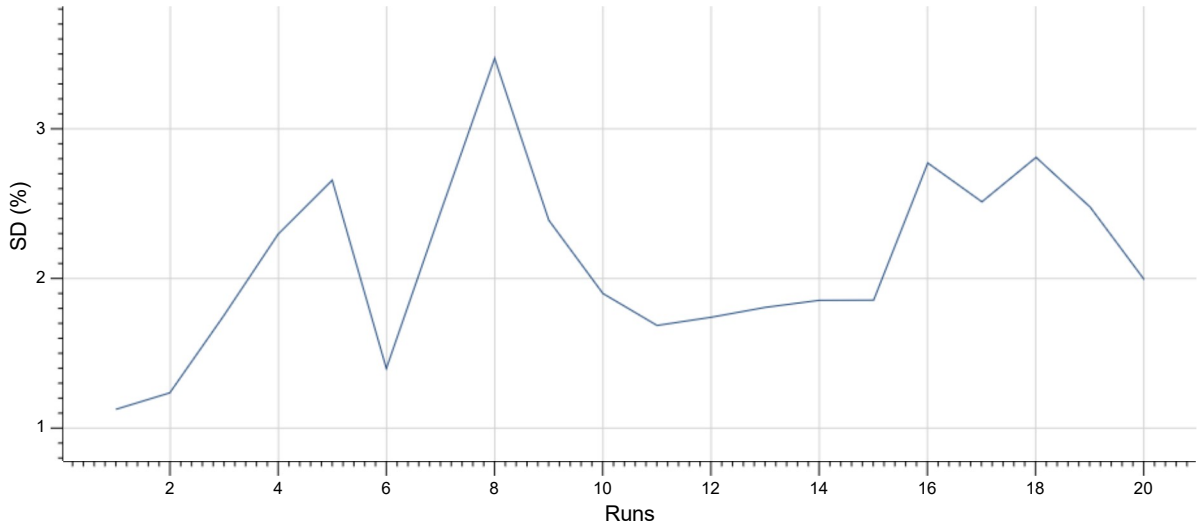
Out[43]:



Relative SD for 10 d2_v2 and 10 a2 machines (performance coefficient applied).

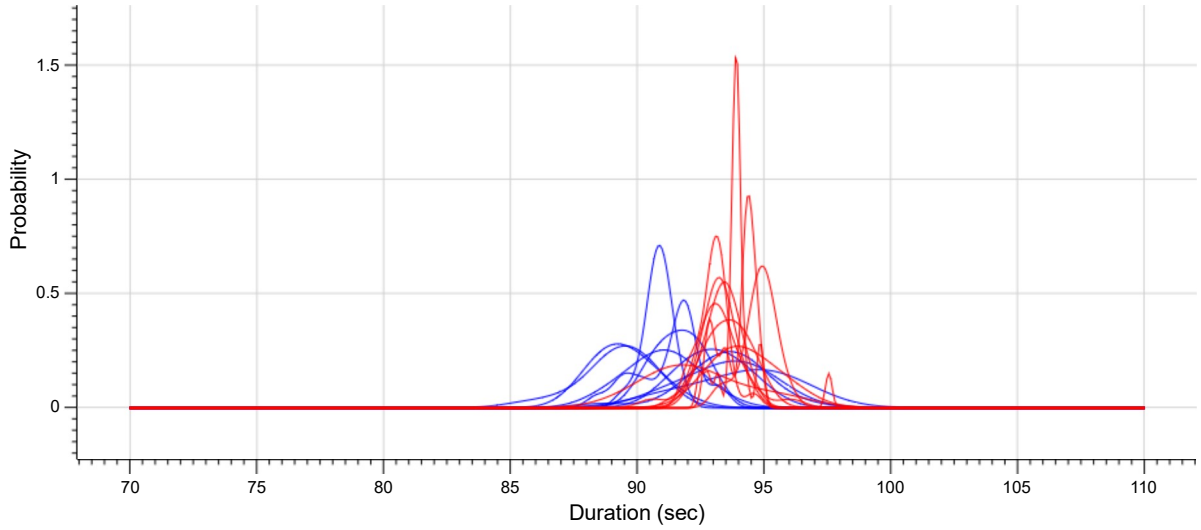
```
In [44]: drawRowsUncertaintiesP adjustedLargeTask3
```

Out[44]:




```
In [45]: drawPdfColumnsRangeColored diffMachinesLargeTask 70.0 110.0 (fun i -> if i < 10 then "blue" else "red")
```

Out[45]:

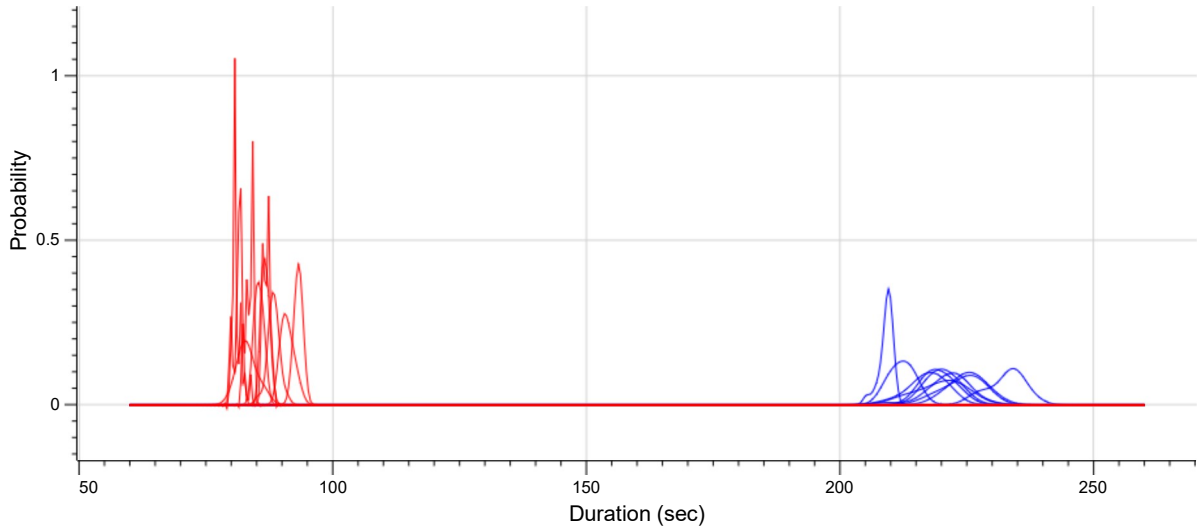


PDFs for results obtained on different VMs adjucted by performance coefficient (red - d2_v2, blue - a2).

In [46]:
```
let shortenedLarge2 = takeRows 20 largeTask2

let diffMachinesLargeTaskNA = shortenedLarge2 |> Seq.append largeTask3 |> Table.OfColumns
```

In [47]:
```
drawPdfColumnsRangeColored diffMachinesLargeTaskNA 60.0 260.0 (fun i -> if i < 10 then "blue" else "red")
```

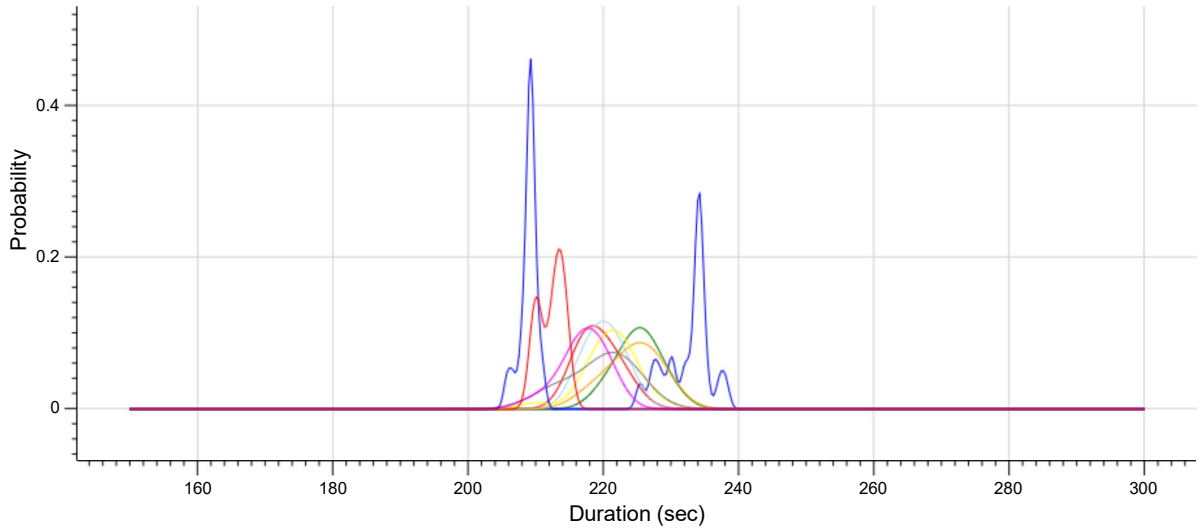
Out[47]:



PDFs for results on different VMs without adjustment.

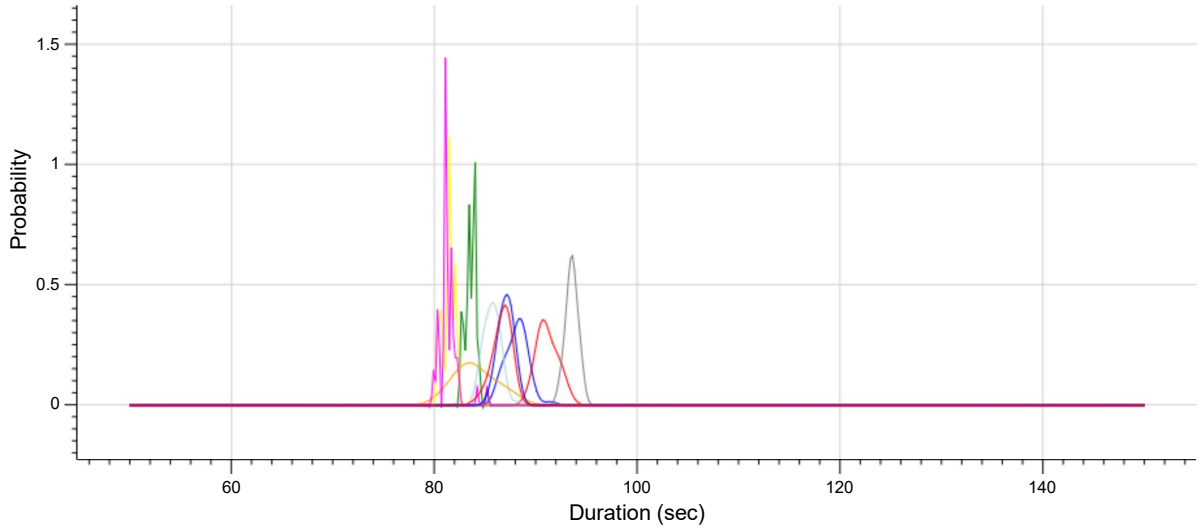
In [48]:
```
drawPdfColumnsRange largeTask3 150.0 300.0
```

Out[48]:



In [49]:
```
drawPdfColumnsRange largeTask2 50.0 150.0
```

Out[49]:



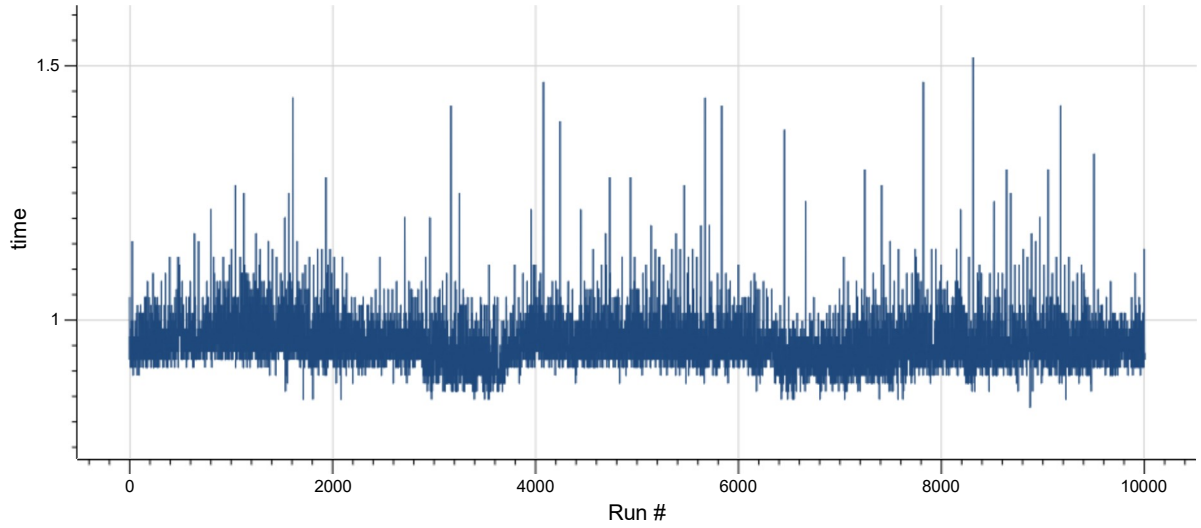
Impact of Long Uptime on Performance of a VM

Here we research, how performance will vary on a VM during a long (~10 hours) period of time. We need this to determine, how often (if at all) should we re-evaluate performance coefficient of a VM.

In [50]:
```
let tablelr = Table.Load ("longrun.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara
.Data.DelimitedFile.Delimiter.Tab })
//tableSmalls
let smallTasklr = [ Column.Create ("0", (tablelr |> Table.Filter ["Task #"; "File #"] (fun (x: float) (y: float)
-> x = 0.0 && y = 0.0)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns
let largeTasklr = [ Column.Create ("0", (tablelr |> Table.Filter ["Task #"; "File #"] (fun (x: float) (y: float)
-> x = 0.0 && y = 1.0)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns
```

In [51]:
```
[Plot.line(LineX.Values ([|1..smallTasklr.[0].Rows.AsReal.Length|] |> Array.map float),
           LineY.Values (smallTasklr.[0].Rows.AsReal.ToArray()),
           titles = Titles.line("Run #", "time")) ] |> Chart.ofList
```

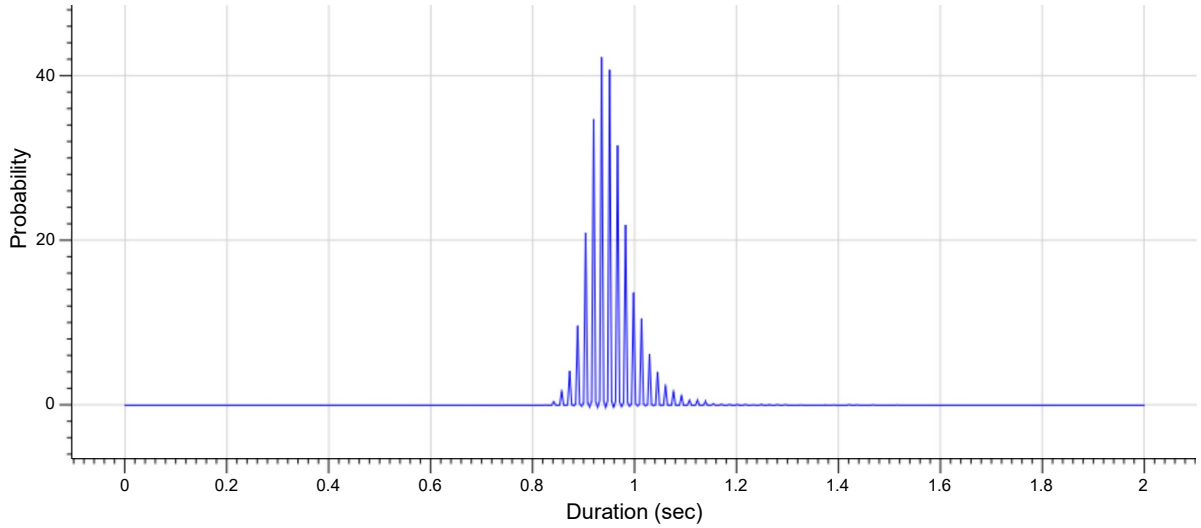
Out[51]:



Running times for a small task on a ~3 hour time interval.

In [52]:
```
drawPdfColumnsRange smallTasklr 0.0 2.0
```

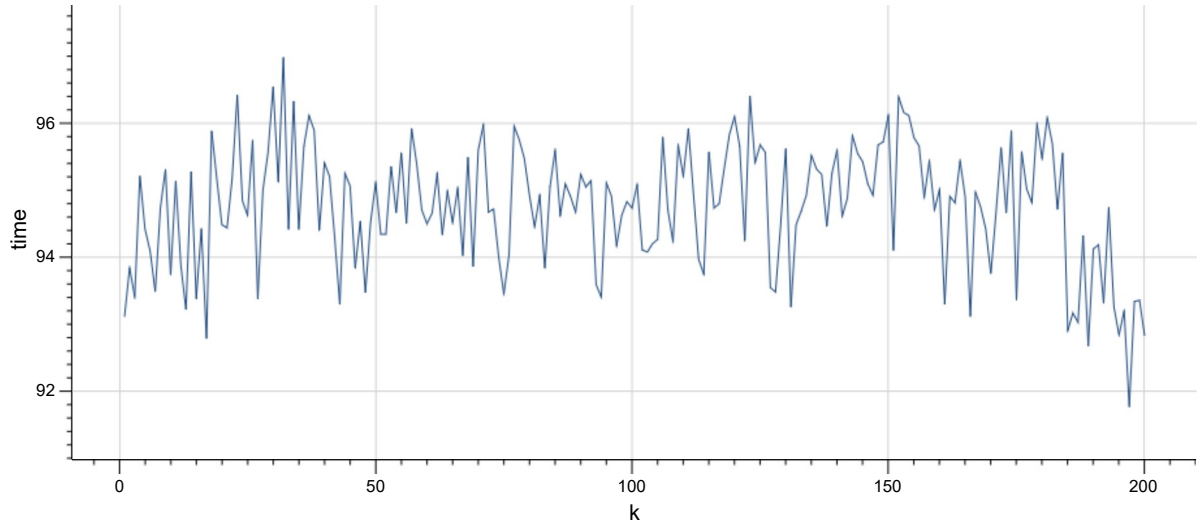
Out[52]:



PDF for 10000 runs of a small task.

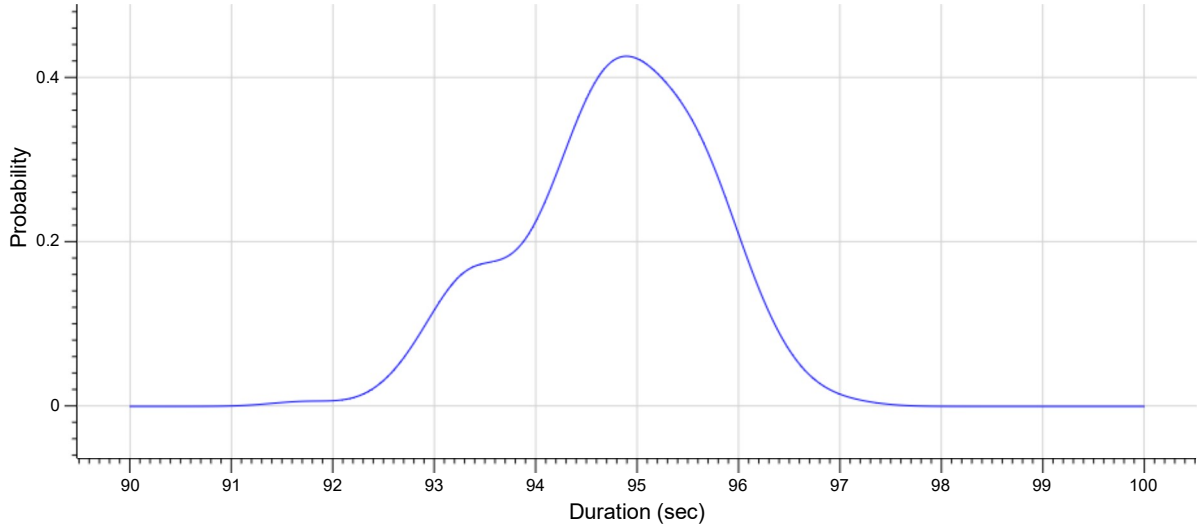
```
In [53]: [Plot.line(LineX.Values ([|1..largeTasklr.[0].Rows.AsReal.Length|] |> Array.map float),
                    LineY.Values (largeTasklr.[0].Rows.AsReal.ToArray()),
                    titles = Titles.line("k", "time")) ] |> Chart.ofList
```

Out[53]:




```
In [54]: drawPdfColumnsRange largeTasklr 90.0 100.0
```

Out[54]:



Same for a large (~90 sec.) task.

```
In [55]: stat (smallTasklr.[0].Rows.AsReal)
```

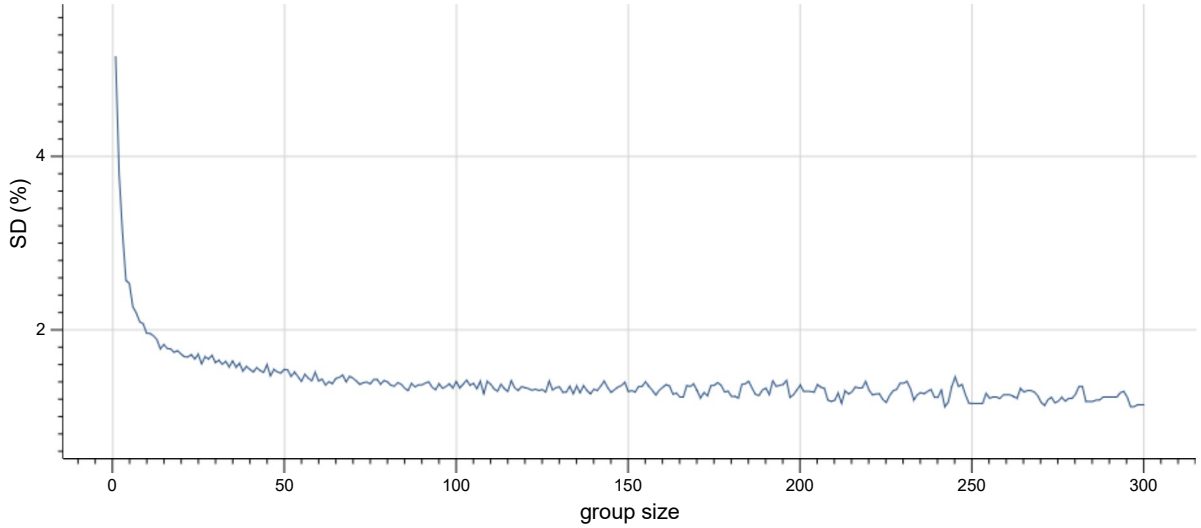
```
Out[55]: {min = 0.828125;
          lb95 = 0.875;
          lb68 = 0.921875;
          median = 0.953125;
          ub68 = 0.921875;
          ub95 = 1.0625;
          max = 1.515625;
          sd = 0.04908684442;}
```

```
In [56]: stat (largeTasklr.[0].Rows.AsReal)
```

```
Out[56]: {min = 91.765625;
          lb95 = 92.86510417;
          lb68 = 93.75;
          median = 94.8359375;
          ub68 = 93.75;
          ub95 = 96.36927083;
          max = 96.984375;
          sd = 0.9126280627;}
```

In [57]:
```
let groupedSmallLr k = blurGroupsT k smallTasklr

[Plot.line(LineX.Values ([|1..300|] |> Array.map float),
                LineY.Values [| for i in 1..300 -> uncertaintyP ((groupedSmallLr i).[0].Rows.AsReal) |],
                titles = Titles.line("group size", "SD (%)")) ] |> Chart.ofList
```

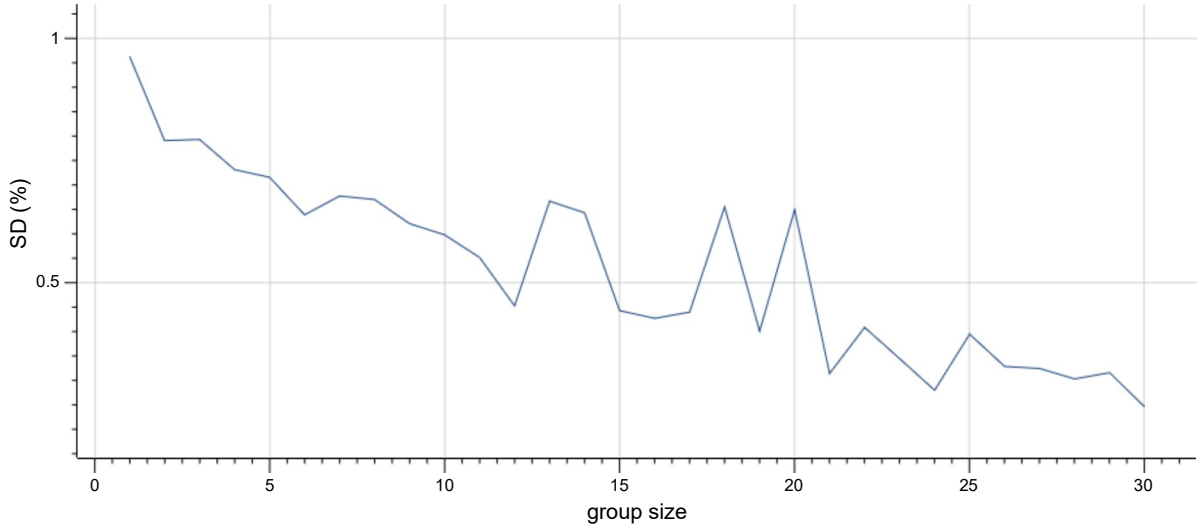
Out[57]:



Relative SD for median value of a group of runs depending on group size (see above for explanations). Small task. It appears, running a small task 20-50 times and taking median value significantly reduces uncertainty.

In [58]:
```
let groupedLargeLr k = blurGroupsT k largeTasklr

[Plot.line(LineX.Values ([|1..30|] |> Array.map float),
                LineY.Values [| for i in 1..30 -> uncertaintyP ((groupedLargeLr i).[0].Rows.AsReal) |],
                titles = Titles.line("group size", "SD (%)")) ] |> Chart.ofList
```

Out[58]:

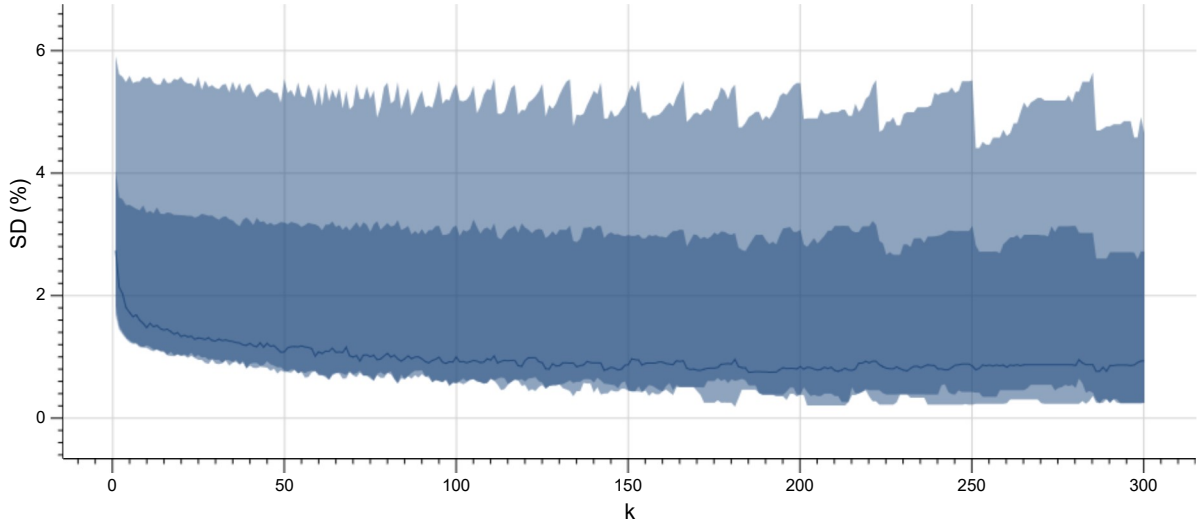


Same for large task. Relative SD is below 1% from the beginning, which is much better, than that for a small task.

In [59]:
```
let tablelrA2 = Table.Load ("longrunA2.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = An
gara.Data.DelimitedFile.Delimiter.Tab })

let smallTasklrA2 = [ for i in 0..9 -> Column.Create (i.ToString(), (tablelrA2 |> Table.Filter ["Task #"; "File #
"] (fun (x: float) (y: float) -> x = float i && y = 0.0)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns
let largeTasklrA2 = [ for i in 0..9 -> Column.Create (i.ToString(), (tablelrA2 |> Table.Filter ["Task #"; "File #
"] (fun (x: float) (y: float) -> x = float i && y = 1.0)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns
```

In [60]:
```
let groupedSmallLrA2 k = blurGroupsT k smallTasklrA2 |> Seq.map (fun c -> uncertaintyP c.Rows.AsReal) |> qsummary

[Plot.line(LineX.Values ([|1..300|] |> Array.map float),
           LineY.UncertainValues ([|1..300|] |> Array.map (fun i -> groupedSmallLrA2 i) |> toQuantiles),
           titles = Titles.line("k", "SD (%)")) ] |> Chart.ofList
```

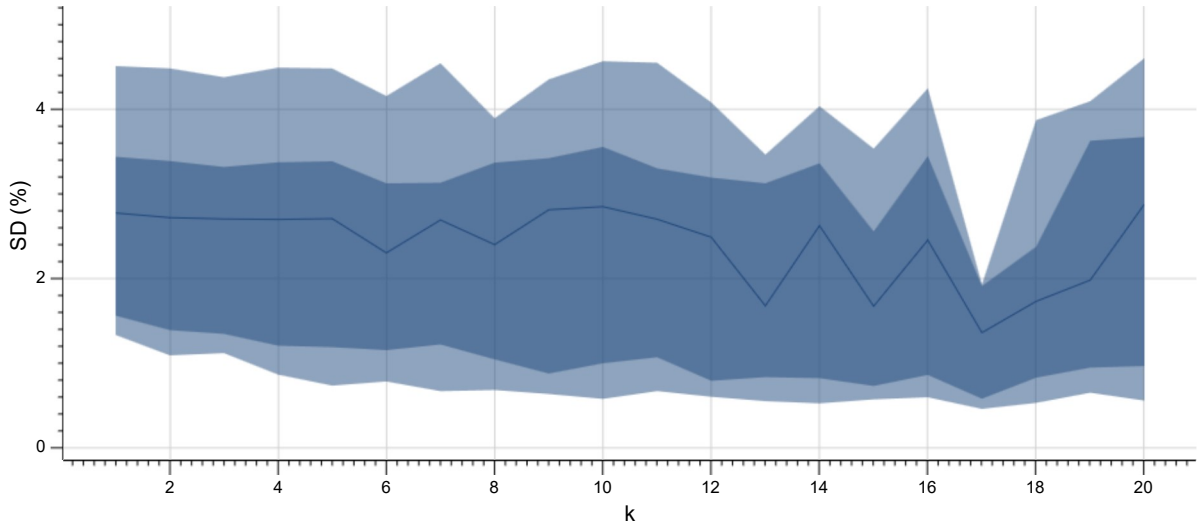
Out[60]:



Dependence between relative standard deviation of median values of the groups of test runs and the size of the group. A2 VMs. Plot's uncertainty (band) shows range of relative SD obtained on different VMs. It appears, A2 machines produce quite unstable results.

In [61]:
```
let groupedLargeLrA2 k = blurGroupsT k largeTasklrA2 |> Seq.map (fun c -> uncertaintyP c.Rows.AsReal) |> qsummary

[Plot.line(LineX.Values ([|1..20|] |> Array.map float),
           LineY.UncertainValues ([|1..20|] |> Array.map (fun i -> groupedLargeLrA2 i) |> toQuantiles),
           titles = Titles.line("k", "SD (%)")) ] |> Chart.ofList
```

Out[61]:



Same for a large task. (A2)

Comparison of Run Statistics (z3 -st) Obtained by Running Same Tests on Different VMs

In [62]:
```
let tableExpSt = Table.Load ("expst.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angar
a.Data.DelimitedFile.Delimiter.Tab })

let smallTaskSt = tableExpSt |> Table.Filter ["File #"] (fun (x: float)  -> x = 0.0)
let largeTaskSt = tableExpSt |> Table.Filter ["File #"] (fun (x: float)  -> x = 1.0)
```

In [63]:
```
schema smallTaskSt
```

Out[63]: <null>

In [64]:
```
[| for i in 4..36 -> (smallTaskSt.[i].Name, (smallTaskSt.[i].Rows.AsReal |> Seq.distinct |> Seq.length) = 1) |]
```

Out[64]:
```
[|("Return value", true); ("CPUTime", false); ("WCTime", false);
  ("Memory", false); ("binary_propagations", true); ("conflicts", true);
  ("decisions", true); ("del_clause", true);
  ("dyn-subsumption-resolution", true); ("elim-blocked-clauses", true);
  ("elim-bool-vars", true); ("elim-clauses", true); ("elim-literals", true);
  ("eliminated-applications", true); ("eliminated-vars", true);
  ("gc-clause", true); ("max-memory", true); ("memory", true);
  ("minimized-lits", true); ("mk-binary-clause", true); ("mk-bool-var", true);
  ("mk-clause", true); ("mk-ternary-clause", true); ("num-allocs", true);
  ("probing-assigned", true); ("propagations", true); ("restarts", true);
  ("rlimit-count", true); ("subsumed", true); ("subsumption-resolution", true);
  ("ternary-propagations", true); ("time", false); ("total-time", false)|]
```

In [65]:
```
[| for i in 4..36 -> (largeTaskSt.[i].Name, (smallTaskSt.[i].Rows.AsReal |> Seq.distinct |> Seq.length) = 1) |]
```

Out[65]:
```
[|("Return value", true); ("CPUTime", false); ("WCTime", false);
  ("Memory", false); ("binary_propagations", true); ("conflicts", true);
  ("decisions", true); ("del_clause", true);
  ("dyn-subsumption-resolution", true); ("elim-blocked-clauses", true);
  ("elim-bool-vars", true); ("elim-clauses", true); ("elim-literals", true);
  ("eliminated-applications", true); ("eliminated-vars", true);
  ("gc-clause", true); ("max-memory", true); ("memory", true);
  ("minimized-lits", true); ("mk-binary-clause", true); ("mk-bool-var", true);
  ("mk-clause", true); ("mk-ternary-clause", true); ("num-allocs", true);
  ("probing-assigned", true); ("propagations", true); ("restarts", true);
  ("rlimit-count", true); ("subsumed", true); ("subsumption-resolution", true);
  ("ternary-propagations", true); ("time", false); ("total-time", false)|]
```

Tests appear to be deterministic in everything, except running time, which is expected.

Minimizing Uncertainty of Performance Coeffitient

Here we research relative standard deviation of median results winthin equal-sized group of runs depending on size of the group and typical running time of the test. The goal is to find a test, median result of a group of which will produce the least relative standard deviation within a reasonable time frame.

To put it simplier, we want a test that we will run for, say, 3 minutes, take the median result, and that result shoud have as less uncertainty as possible.

In [66]:
```
let tableLots = Table.Load ("lots.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara.
Data.DelimitedFile.Delimiter.Tab })

let lotsByFileFull = [| for i in 0..4 -> tableLots |> Table.Filter ["File #"] (fun (x:float) -> x = float i) |]

let lotsByFile = [| for i in 0..4 -> [for j in 0..9 -> Column.Create (j.ToString(), (lotsByFileFull.[i] |> Table.
Filter ["Task #"] (fun (x: float) -> x = float j)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns |]
```

In [67]:
```
[| for i in 0..4 -> lotsByFileFull.[i].["CPUTime"].Rows.AsReal |> qsummary |]
```

Out[67]:
```
[|{min = 1.953125;
   lb95 = 2.03125;
   lb68 = 2.1875;
   median = 2.296875;
   ub68 = 2.46875;
   ub95 = 2.671875;
   max = 3.03125;}; {min = 5.046875;
                     lb95 = 5.1875;
                     lb68 = 5.578125;
                     median = 5.84375;
                     ub68 = 6.21875;
                     ub95 = 6.671875;
                     max = 7.171875;}; {min = 268.15625;
                                        lb95 = 273.7397135;
                                        lb68 = 293.484375;
                                        median = 307.265625;
                                        ub68 = 324.2535417;
                                        ub95 = 332.96875;
                                        max = 347.890625;}; {min = 56.59375;
                                                             lb95 = 57.734375;
                                                             lb68 = 62.30291667;
                                                             median = 64.609375;
                                                             ub68 = 69.0;
                                                             ub95 = 72.125;
                                                             max = 75.109375;};
  {min = 12.40625;
   lb95 = 12.67721354;
   lb68 = 13.640625;
   median = 14.1875;
   ub68 = 15.265625;
   ub95 = 16.4375;
   max = 17.234375;}|]
```

We wanted tests with average running time of 2, 5, 10, 20, and 30 seconds, but benchmarks that were completed in such times during last experient on the cluster suddenly produced different times in batch. Obviously, something was different, e.g. z3 parameters; also use of a single core was enforced in batch, while in cluster z3 had access to multiple cores.

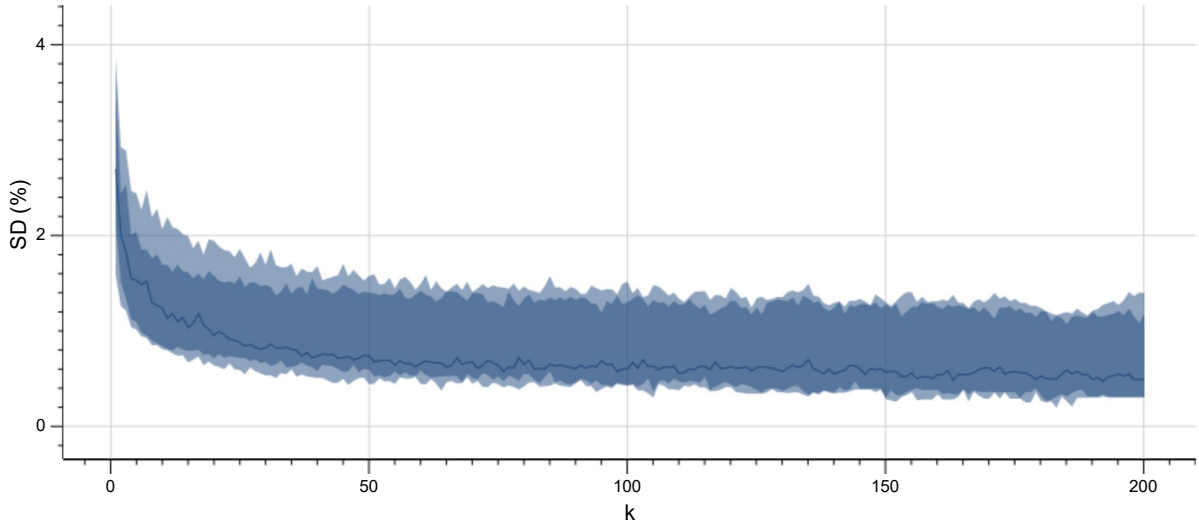
Anyway, we've got 3 good sets of data: 1st with average time of 2.2 sec; 2nd - 5.8 sec; 5th - 15 sec.

In [68]:
```
let lotsByFileBGsd i k = blurGroupsT k lotsByFile.[i] |> Seq.map (fun c -> uncertaintyP c.Rows.AsReal) |> qsummar
y
```

In [69]:
```
let plotLotssd i k = [Plot.line(LineX.Values ([| for j in 1..k -> float j|]),
                        LineY.UncertainValues ([|for j in 1..k -> lotsByFileBGsd i j |] |> toQuantiles),
                        titles = Titles.line("k", "SD (%)")) ] |> Chart.ofList
```

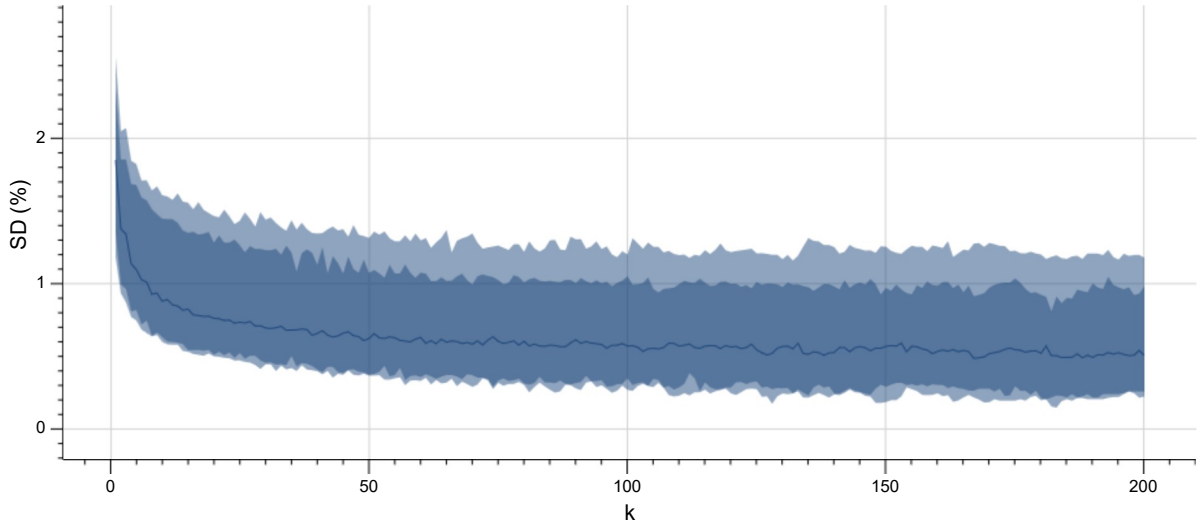
In [70]:
```
plotLotssd 0 200
```

Out[70]:



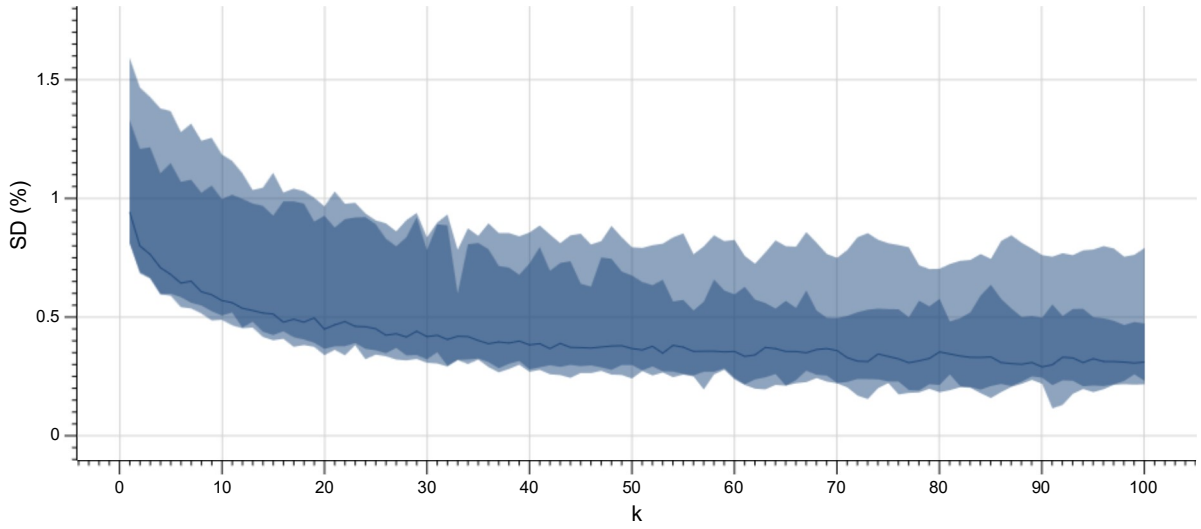
In [71]:
```
plotLotssd 1 200
```

Out[71]:



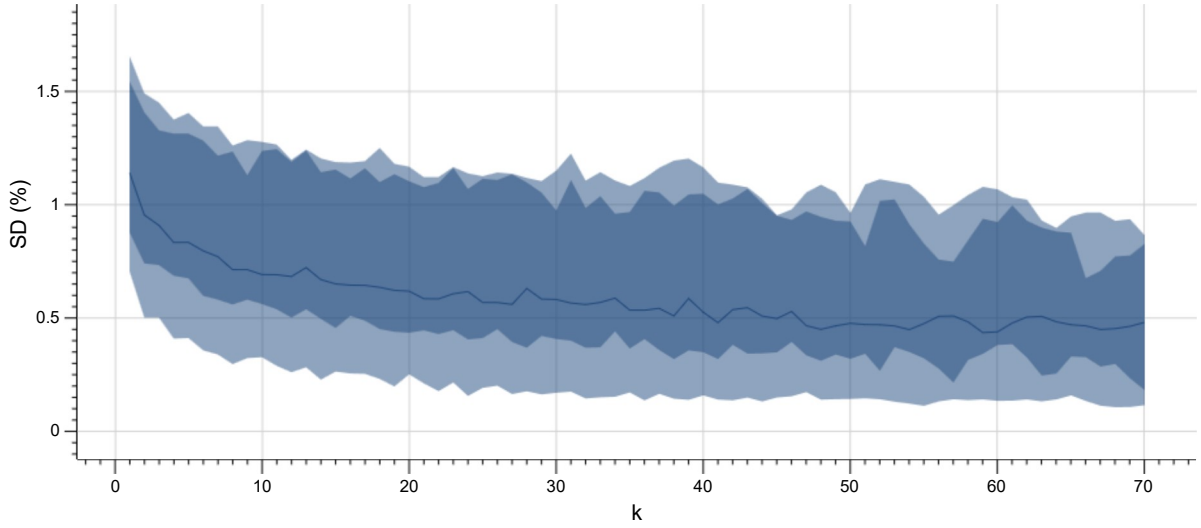
In [72]:
```
plotLotssd 2 100
```

Out[72]:



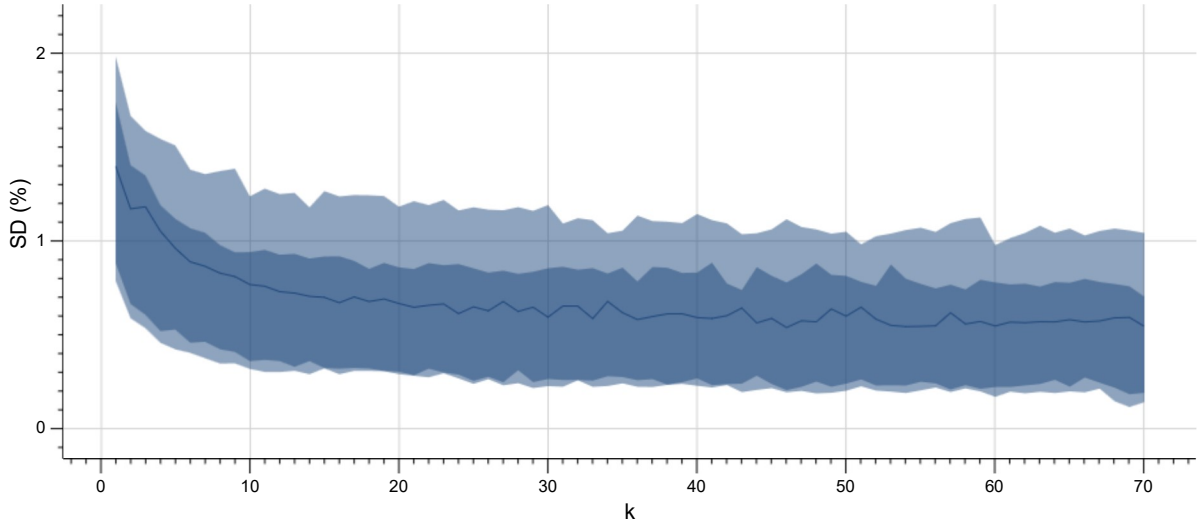
In [73]:
```
plotLotssd 3 70
```

Out[73]:



In [74]:
```
plotLotssd 4 70
```

Out[74]:



So,

- 50+ runs of 2.3 sec problem give us 0.7-1.6% of uncertainty within 130 sec
- 50+ runs of 5 sec problem give us 0.4-1.3% of uncertainty within 250 sec
- 10+ runs of 15 sec problem give us 0.3-1.3% of uncertainty within 150 sec

Among the listed, the last one appears most pleasant. We need to experiment more, obviously. Also, it must be mentioned, that longer (1 min and 5 min) problems do not produce any better results within 10 minutes time frame. In fact, to reduce uncertainty to values less that 1% we need to run these experiments for more than an hour.

Averaging Results for Short-Running Experiments to Decrease Uncertainty

As we've seen above, shot-running experiments have a rather high (compared to the same of long-running ones) uncertainty, which can be decreased by repeating the experiment multiple times and picking the median value.

Here we're going to try to do that with the data obtained on previous experiment: we'll compute performance coeffitient based on 10 runs of 15 sec task, then we'll apply them to median results of groups of 2.3 sec experiment runs, and we'll watch, how will it affect the uncertainty of end result.

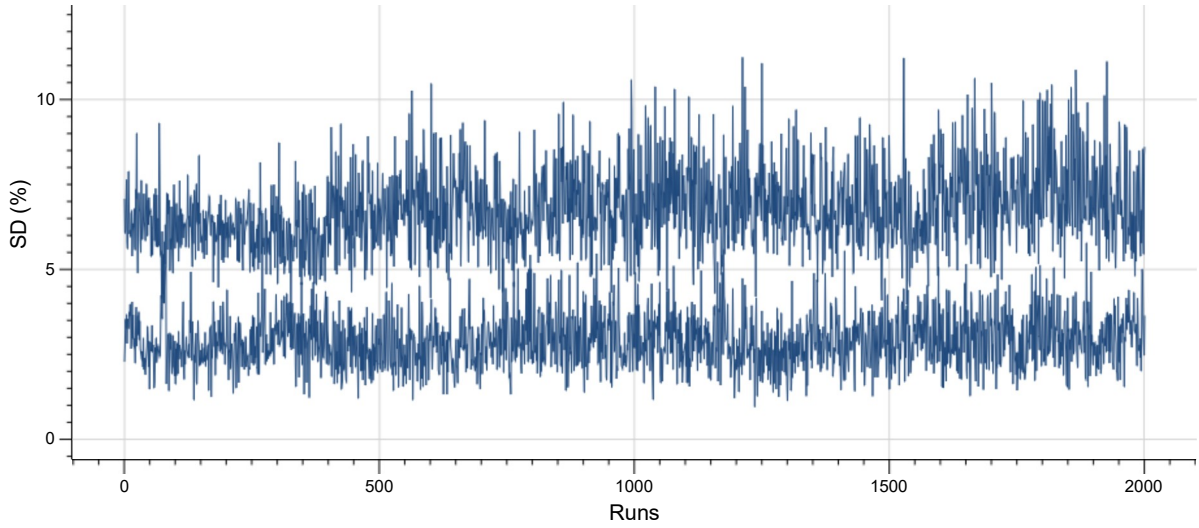
In [75]:
```
//computing coefficients
let typicalTime = 14.1875
let perfCoefs = blurGroupsT 10 (lotsByFile.[4]) |> Seq.map (fun c -> typicalTime / c.Rows.AsReal.[0]) |> Array.of
Seq
perfCoefs
```

Out[75]:
```
[|0.9222955815; 1.044879171; 0.8590350047; 1.03652968; 1.006651885; 1.002207506;
  0.9848156182; 1.115479115; 1.003314917; 0.9956140351|]
```

In [76]:
```
let times2 = lotsByFile.[0]
let times2grouped k = blurGroupsT k times2
let times2groupedAdj k = times2grouped k |> Seq.mapi (fun i c -> Column.Create (c.Name, c.Rows.AsReal |> Seq.map
(fun v -> v * perfCoefs.[i]) )) |> List.ofSeq |> Table.OfColumns
```

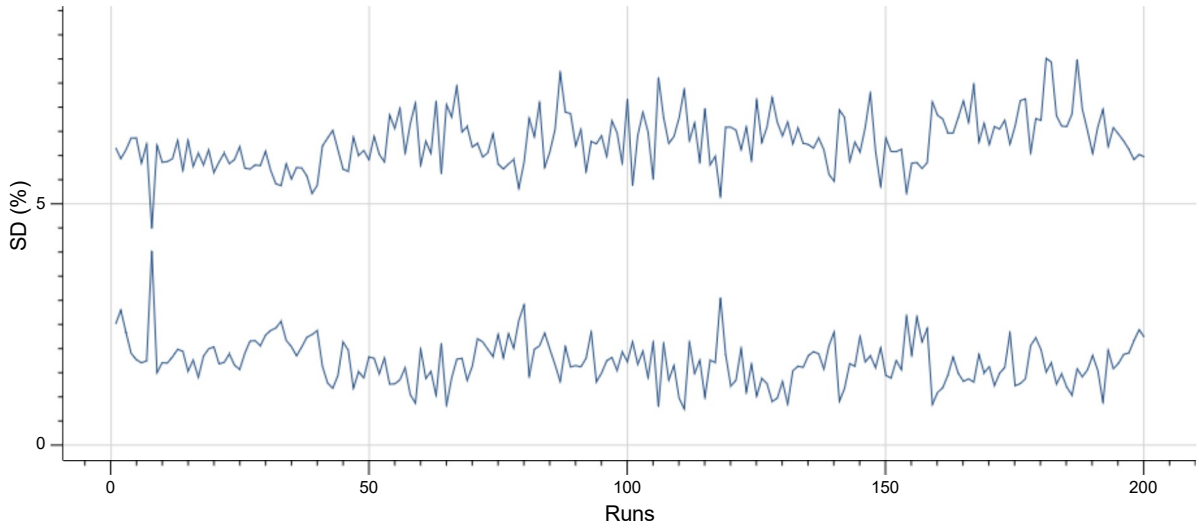
In [77]:
```
[ (drawRowsUncertaintiesP <| times2groupedAdj 1).Plots.[0]; (drawRowsUncertaintiesP <| times2grouped 1).Plots.[0]
 ] |> Chart.ofList
```

Out[77]:



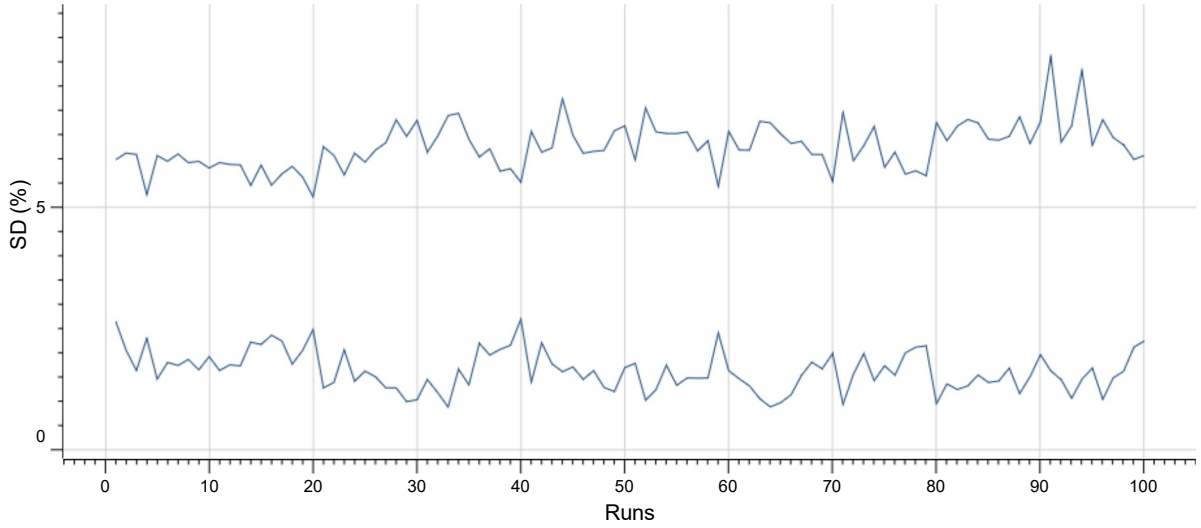
In [78]:
```
[ (drawRowsUncertaintiesP <| times2groupedAdj 10).Plots.[0]; (drawRowsUncertaintiesP <| times2grouped 10).Plots.[
0] ] |> Chart.ofList
```

Out[78]:



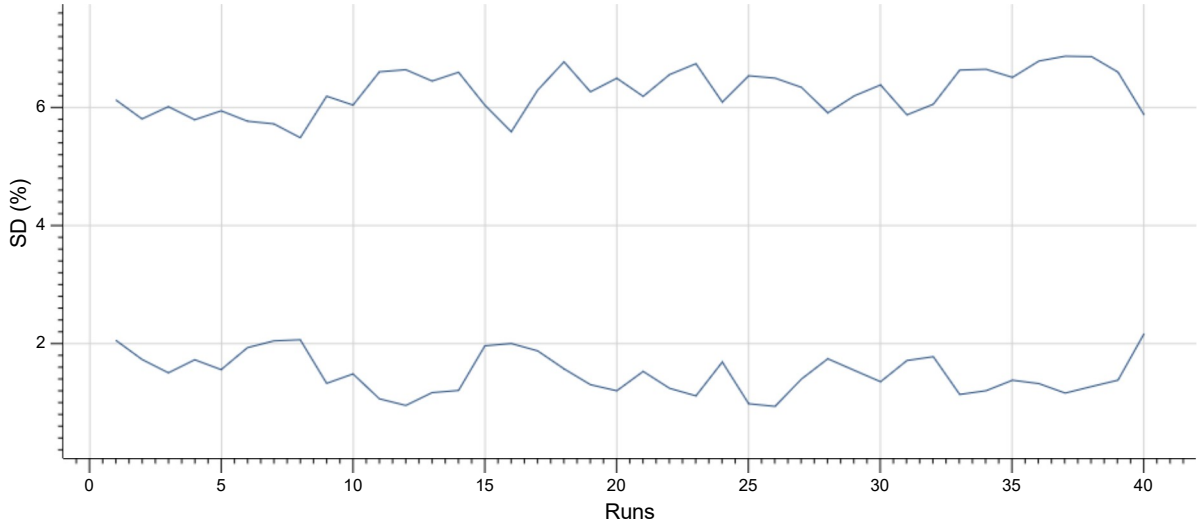
In [79]:
```
[ (drawRowsUncertaintiesP <| times2groupedAdj 20).Plots.[0]; (drawRowsUncertaintiesP <| times2grouped 20).Plots.[
0] ] |> Chart.ofList
```

Out[79]:



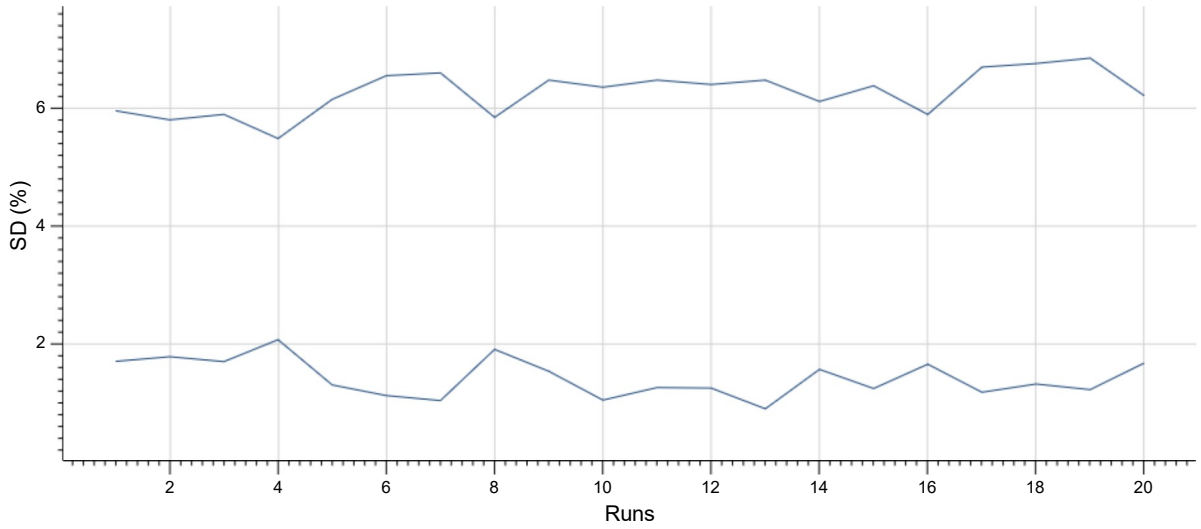
In [80]:
```
[ (drawRowsUncertaintiesP <| times2groupedAdj 50).Plots.[0]; (drawRowsUncertaintiesP <| times2grouped 50).Plots.[
0] ] |> Chart.ofList
```

Out[80]:



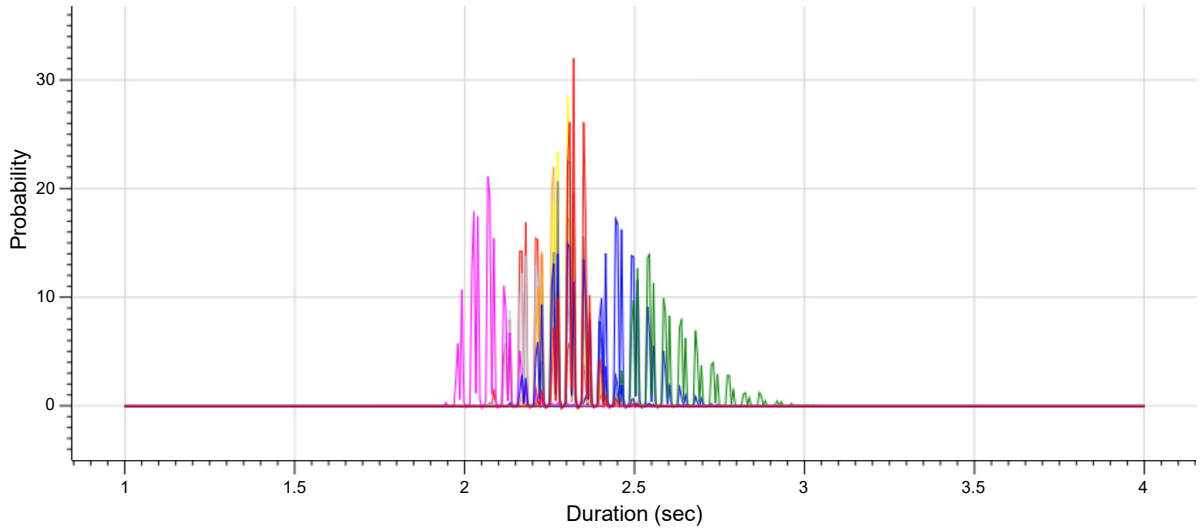
In [81]:
```
[ (drawRowsUncertaintiesP <| times2groupedAdj 100).Plots.[0]; (drawRowsUncertaintiesP <| times2grouped 100).Plots
.[0] ] |> Chart.ofList
```

Out[81]:



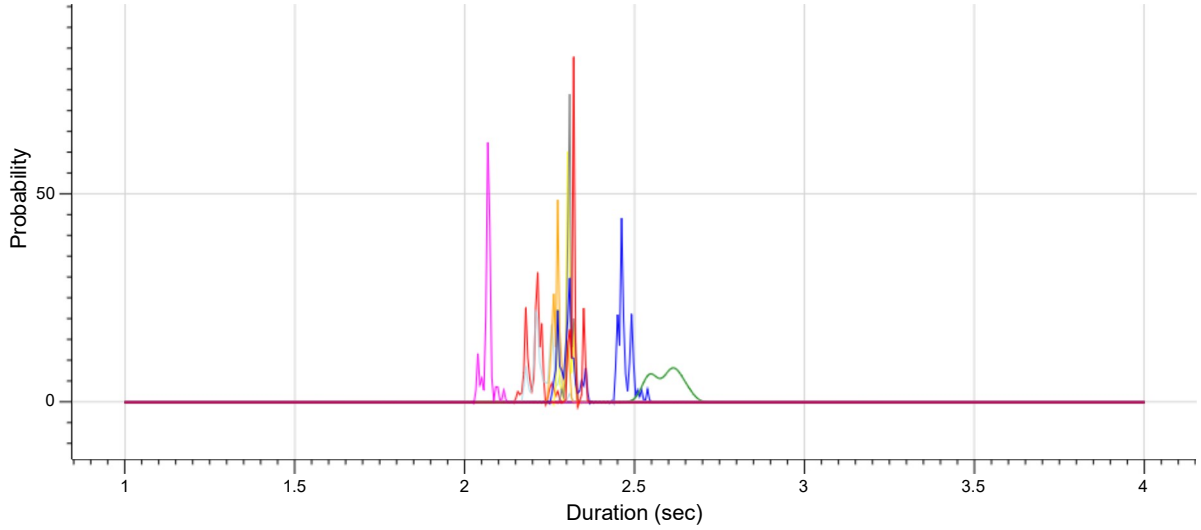
In [82]:
```
drawPdfColumnsRange (times2grouped 1) 1.0 4.0
```

Out[82]:



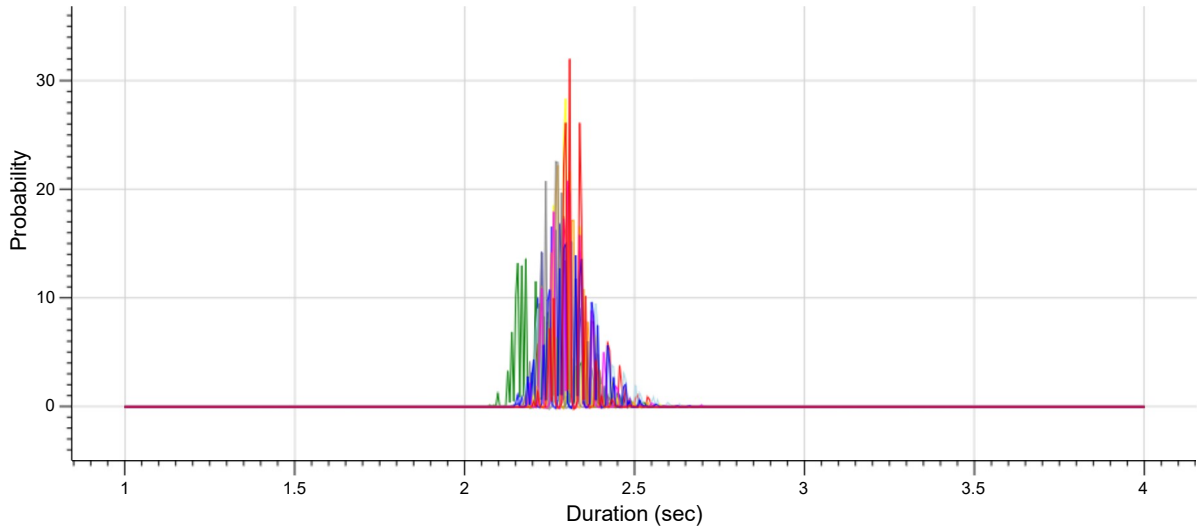
In [83]:
```
drawPdfColumnsRange (times2grouped 50) 1.0 4.0
```

Out[83]:



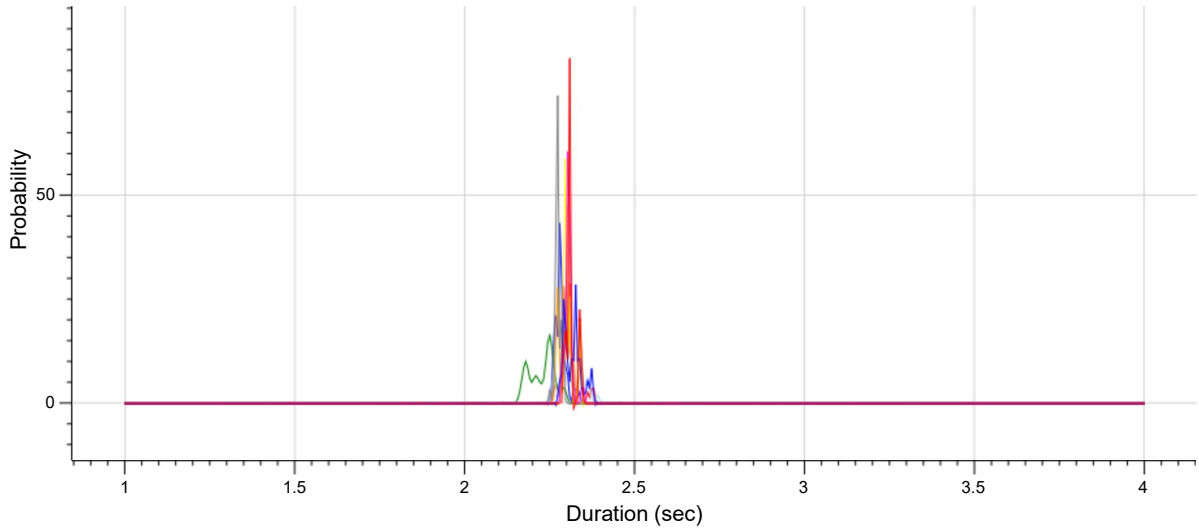
In [84]:
```
drawPdfColumnsRange (times2groupedAdj 1) 1.0 4.0
```

Out[84]:



In [85]:
```
drawPdfColumnsRange (times2groupedAdj 50) 1.0 4.0
```

Out[85]:



Finding the Best Test Problem

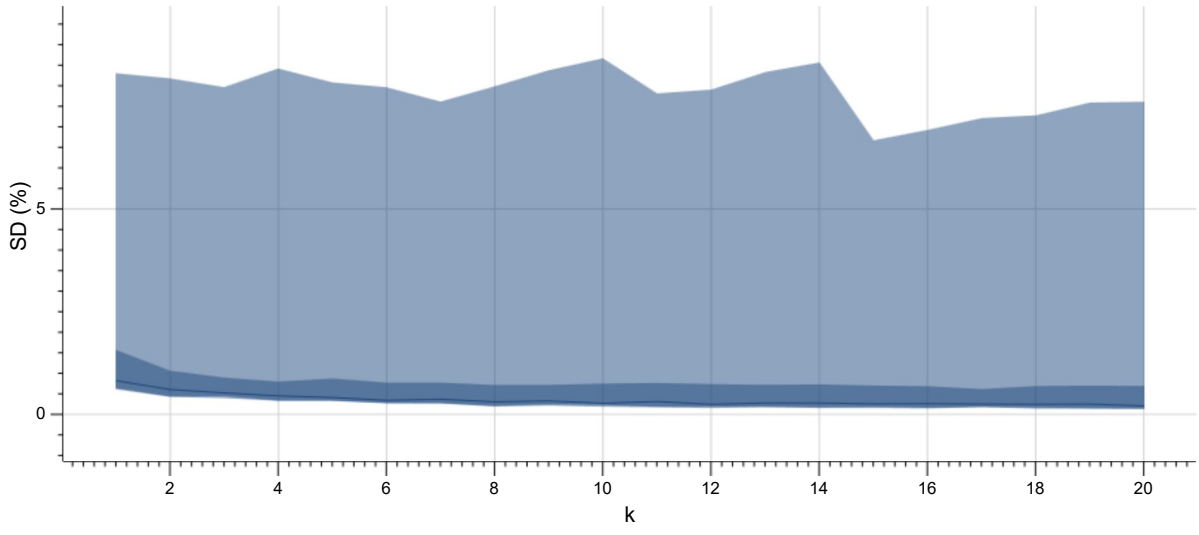
In [86]:
```
let table15s = Table.Load ("15s.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara.Da
ta.DelimitedFile.Delimiter.Tab })

let t15sByFileFull = [| for i in 0..7 -> table15s |> Table.Filter ["File #"] (fun (x:float) -> x = float i) |]

let t15sByFile = [| for i in 0..7 -> [for j in 0..9 -> Column.Create (j.ToString(), (t15sByFileFull.[i] |> Table.
Filter ["Task #"] (fun (x: float) -> x = float j)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns |]
```

In [87]:
```
let tableBGsd table k = blurGroupsT k table |> Seq.map (fun c -> uncertaintyP c.Rows.AsReal) |> qsummary

let plotsd table k = [Plot.line(LineX.Values ([| for j in 1..k -> float j|]),
                        LineY.UncertainValues ([|for j in 1..k -> tableBGsd table j |] |> toQuantiles),
                        titles = Titles.line("k", "SD (%)")) ] |> Chart.ofList
```

In [88]:
```
let table35s = Table.Load ("35s.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara.Da
ta.DelimitedFile.Delimiter.Tab })

let t35s = [for j in 0..9 -> Column.Create (j.ToString(), (table35s |> Table.Filter ["Task #"] (fun (x: float) ->
 x = float j)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns
```

In [89]:
```
plotsd t35s 20
```

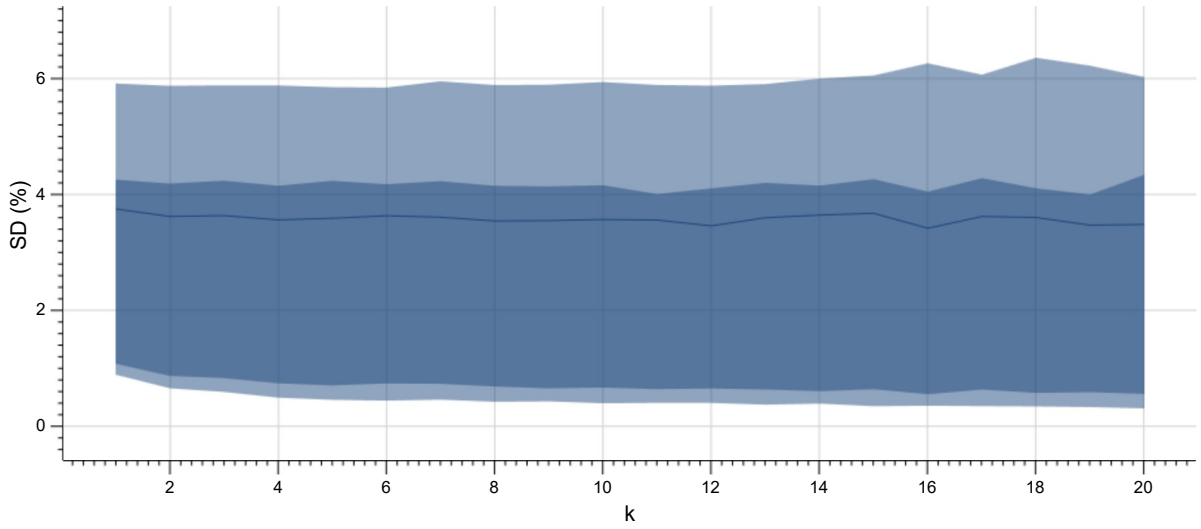
Out[89]:



In [90]:
```
let table25s = Table.Load ("25s.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara.Da
ta.DelimitedFile.Delimiter.Tab })

let t25s = [for j in 0..9 -> Column.Create (j.ToString(), (table25s |> Table.Filter ["Task #"] (fun (x: float) ->
 x = float j)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns
```

In [91]:
```
plotsd t25s 20
```

Out[91]:



In [92]:
```
let table5s = Table.Load ("5s.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara.Data
.DelimitedFile.Delimiter.Tab })

let t5sByFileFull = [| for i in 0..12 -> table5s |> Table.Filter ["File #"] (fun (x:float) -> x = float i) |]

let t5sByFile = [| for i in 0..12 -> [for j in 0..9 -> Column.Create (j.ToString(), (t5sByFileFull.[i] |> Table.F
ilter ["Task #"] (fun (x: float) -> x = float j)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns |]
```

In [93]:
```
let table10s = Table.Load ("10s.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara.Da
ta.DelimitedFile.Delimiter.Tab })

let t10sByFileFull = [| for i in 0..5 -> table10s |> Table.Filter ["File #"] (fun (x:float) -> x = float i) |]

let t10sByFile = [| for i in 0..5 -> [for j in 0..9 -> Column.Create (j.ToString(), (t10sByFileFull.[i] |> Table.
Filter ["Task #"] (fun (x: float) -> x = float j)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns |]
```

In [94]:
```
let table7s = Table.Load ("7s.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara.Data
.DelimitedFile.Delimiter.Tab })

let t7sByFileFull = [| for i in 0..19 -> table7s |> Table.Filter ["File #"] (fun (x:float) -> x = float i) |]

let t7sByFile = [| for i in 0..19 -> [for j in 0..9 -> Column.Create (j.ToString(), (t7sByFileFull.[i] |> Table.F
ilter ["Task #"] (fun (x: float) -> x = float j)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns |]
```

In [95]:
```
let table12s = Table.Load ("12s.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara.Da
ta.DelimitedFile.Delimiter.Tab })

let t12sByFileFull = [| for i in 0..7 -> table12s |> Table.Filter ["File #"] (fun (x:float) -> x = float i) |]

let t12sByFile = [| for i in 0..7 -> [for j in 0..4 -> Column.Create (j.ToString(), (t12sByFileFull.[i] |> Table.
Filter ["Task #"] (fun (x: float) -> x = float j)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns |]
```

In [96]:
```
let table20s = Table.Load ("20s.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara.Da
ta.DelimitedFile.Delimiter.Tab })

let t20sByFileFull = [| for i in 0..4 -> table20s |> Table.Filter ["File #"] (fun (x:float) -> x = float i) |]

let t20sByFile = [| for i in 0..4 -> [for j in 0..4 -> Column.Create (j.ToString(), (t5sByFileFull.[i] |> Table.F
ilter ["Task #"] (fun (x: float) -> x = float j)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns |]
```

In [97]:
```
(tableBGsd t7sByFile.[3] 60).ub95
```

Out[97]:
```
1.542811847
```

In [98]:
```
let ub95 (tables: Table array) (groupSize: int) = tables |> Array.map (fun t -> (tableBGsd t groupSize).ub95)

let minInd arr = arr |> Array.fold (fun (acc, ind, i) elem -> if elem < acc then (elem, i+1, i+1) else (acc, ind,
 i+1)) (arr.[0] + 1.0, -1, -1)

let minub95Ind tables groupSize = ub95 tables groupSize |> minInd
```

In [99]:
```
minub95Ind t5sByFile 60
```

Out[99]:
```
(0.5778548224, 1, 12)
```

In [100]:
```
minub95Ind t7sByFile 60
```

Out[100]:
```
(0.6824209025, 17, 19)
```

In [101]:
```
minub95Ind t10sByFile 60
```

Out[101]:
```
(0.8074975797, 3, 5)
```

In [102]:
```
minub95Ind t12sByFile 45
```

Out[102]: (0.9086729278, 0, 7)

In [103]:
```
minub95Ind t15sByFile 40
```

Out[103]: (0.3834899836, 7, 7)

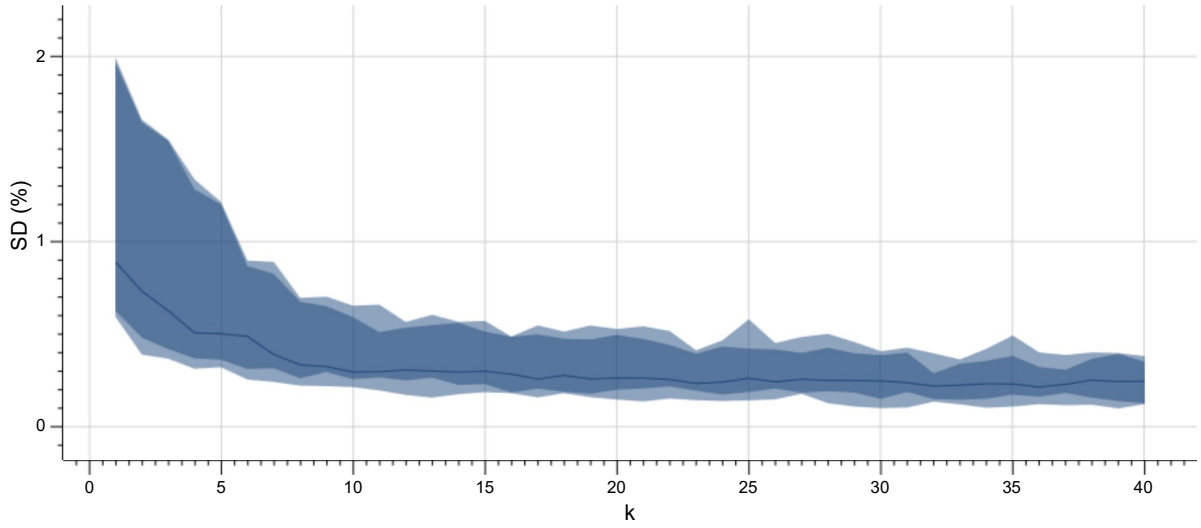
In [104]:
```
minub95Ind t20sByFile 30
```

Out[104]: (0.6922145329, 1, 4)

File, appearing to be the best choice is bench_7536.smt2 (7th file in 15s set)

In [105]:
```
plotsd t15sByFile.[7] 40
```

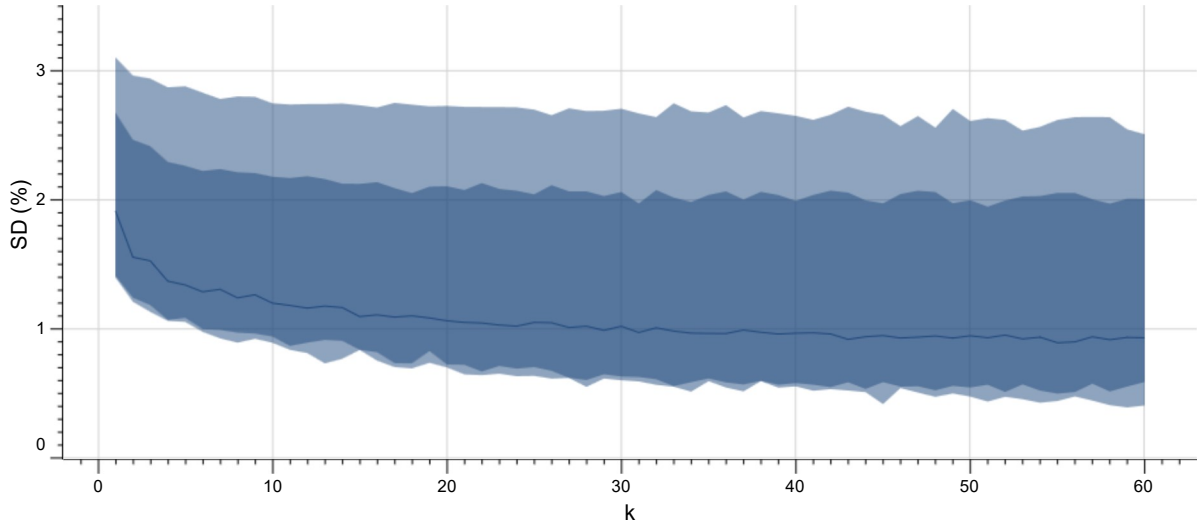
Out[105]:



In [106]:
```
let tableBest = Table.Load ("best.tsv", { Angara.Data.DelimitedFile.ReadSettings.Default with Delimiter = Angara
.Data.DelimitedFile.Delimiter.Tab })

let tBest = [for j in 0..9 -> Column.Create (j.ToString(), (tableBest |> Table.Filter ["Task #"] (fun (x: float)
 -> x = float j)).["CPUTime"].Rows.AsReal) ] |> Table.OfColumns
```

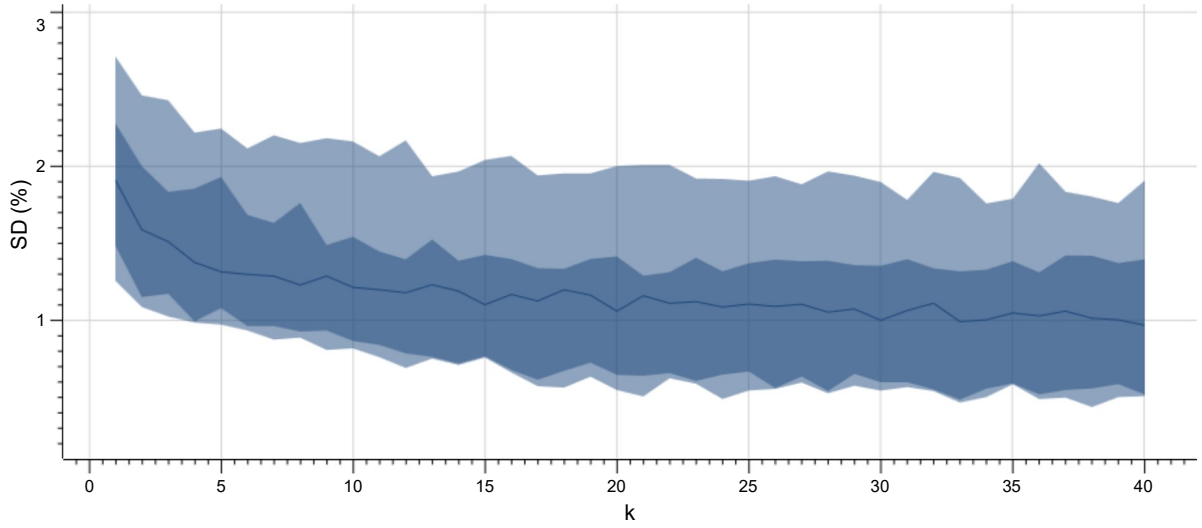
In [107]:
```
plotsd tBest 60
```

Out[107]:



In [108]:
```
let best_cut1 = takeRows 400 tBest
plotsd best_cut1 40
```

Out[108]:



In [109]:
```
qsummary tableBest.["CPUTime"].Rows.AsReal
```

Out[109]:
```
{min = 14.140625;
 lb95 = 14.640625;
 lb68 = 15.5625;
 median = 16.34375;
 ub68 = 17.03125;
 ub95 = 17.921875;
 max = 18.828125;}
```

In [ ]: