Technical Report OUCS-2016-03

How to compute a mean?

Author:
Richard O'Keefe

Department of Computer Science, University of Otago, New Zealand

How to compute a mean?

Richard A. O'Keefe
Computer Science, Otāgo

August 2016

Abstract

Computing the mean of a sequence of numbers is easy; computing means of other kinds of measurements, and for other kinds of container, is harder. This note presents some algorithms and benchmarks.

1 Introduction

Computing an arithmetic mean seems so simple:

$$\bar{x} = n^{-1} \sum_{i=1}^{n} x_i$$

Of course, if the x_i are floating-point numbers we run into the usual "floating-point addition is not associative" problems, but mostly we do not expect any problems. The following C code is typical:

```
double mean(double const *x, int n) {
    double s = 0.0;
    for (int i = 0; i < n; i++) s += x[i];
    return s/n;
}
```

However, on closer inspection, the problem has several parameters which require different approaches. In this working note, I examine some of these parameters in the context of developing a Smalltalk library.

The parameters select between

- linear or circular statistics,
- absolute, ratio, or interval scale measurements,
- levels of numerical accuracy, and
- off-line or on-line algorithm.

2 Linear or circular statistics?

For ordinary linear measurements, like the difference between two prices, we have

$$\begin{array}{rcl} \text{mean} \quad \bar{x} & = & n^{-1}\sum_{i=1}^{n} x_i \\ \text{sample variance} \quad V_x & = & (n-1)^{-1}\sum_{i=1}^{n}(x_i-\bar{x})^2 \\ \text{sample standard deviation} \quad s_x & = & \sqrt{V_x} \end{array}$$

Aside from the fact that the mean is undefined unless $n \geq 1$ and the variance and standard deviation is undefined unless $n \geq 2$, there are no special difficulties here. If the x_i are rational numbers, the mean and variance can be calculated exactly. If they are floating-point numbers, they have the problems common to sums, which we consider later.

But what if the measurements are things like angles, such as compass directions, time of day, day of week, or day of year? These things are not even ordered: Wednesday follows Saturday, but Saturday also follows Wednesday.[1] Such data are quite common, yet even commercial statistics packages may fail to handle them. (There is a third-party "CircStat2009" toolbox for Matlab. For R there are packages "CircStats", "Circular", and "Directional".) This leads to absurdities like a date distribution with many observations in January and December and no observations in February to November being reported as having a mean at the end of June. Some references:

- *Topics in Circular Statistics*, Jammalamadaka, S. Rao, and A. SenGupta, 2001, World Scientific Press, Singapore.
- *Statistical analysis of spherical data*, N. I. Fisher, T. Lewis, and B. .J. J. Embleton, 1987, Cambridge University Press.
- *Statistical Analysis of Circular Data*, N. I. Fisher, 1993, Cambridge University Press.
- *Circular Statistics in R*, A. Pewsey, M. Neuhauser, and G. D. Ruxton, 2014, Oxford University Press.
- *Mean of circular quantities*, Wikipedia.

Simple circular statistics are

$$\begin{array}{rcl} \text{mean} \quad \bar{\theta} & = & \text{arctan2}(S, C) \\ \text{sample variance} \quad V_\theta & = & 1-\overline{R} \\ \text{sample standard deviation} \quad s_\theta & = & \sqrt{-2\ln \overline{R}} \end{array}$$

where

$$C = \sum_{i=1}^{n} \cos\theta_i$$

[1] A colleague reminded me of a sign "No parking before midnight", which entails no parking at any time.

$$
\begin{aligned}
S &= \sum_{i=1}^{n} \sin\theta_i \\
R &= \sqrt{C^2 + S^2} \\
\overline{R} &= R/n
\end{aligned}
$$

The intuition is straightforward: represent a circular (or spherical) datum as a unit vector in 2D (or 3D) space, compute the mean of the vectors, take the direction of the mean as the mean direction, and take the length of the vector as an indication of how scattered the data are (1: the data all point the same way, 0: the data point every which way). The mean is a direction, the variance and standard deviation are not.

The point here is that the calculations are *different*. We can for example compute the mean day-of-year from a collection of Dates like this:

```
require: 'date-angles.st'

angles := dates collect: [:each | dayOfYearRadians].
meanAngle := angles circularMean.
meanDay := (meanAngle radiansToDegrees * (365/360) + 1) rounded.
```

Some directional data is "axial". Think of it as reporting a *diameter* of a circle or sphere instead of a *radius*. The direction of a road is axial. The 2D axial mean is computed by doubling the angles, computing the circular mean, and halving the result.

Strictly speaking, we'd want at least linear, circular, 2D axial, spherical (points on the surface of the Earth, say), and 3D axial sets of methods.

3 Smalltalk

Smalltalk is a class-based object-oriented programming language with no compile-time types. A ⊕ beside a class name means it is peculiar to my Smalltalk system, called ASTC.

3.1 Numbers

Here is part of the class hierarchy relating to numbers. The "abstractSubclass:" and "valueSubclass:" forms are extensions to ANSI Smalltalk: an abstract subclass is one which may have no direct instances, and a value subclass is one which may not be mutated by public methods, and in particular may not be created in an uninitialised state.

```
Object abstractSubclass: #Magnitude
  comment: "things that can be compared"
  methods:
    < other
      self subclassResponsibility.
    > other ...
    >= other ...
    <= other ...
    between: lower and: upper
      ↑self <= other and: [self >= lower]
    max: other ...
    min: other ...

Magnitude abstractSubclass: #MagnitudeWithAddition ⊕
  comment: "Types $T$ such that there is a type $D$ for which
          $T + D \to T$, $T - D \to T$, $T < T \to$ Boolean."

MagnitudeWithAddition valueSubclass: #DateAndTime
  comment: "The ANSI standard timestamp class; $D$ = Duration."

MagnitudeWithAddition valueSubclass: #Date
  comment: "The Smalltalk-80 date class; $D$ = Integer"

MagnitudeWithAddition valueSubclass: #IntervalScaleQuantity ⊕
  comment: "Physical measurements without a zero; $D$ = Quantity."

MagnitudeWithAddition abstractSubclass: #QuasiArithmetic ⊕
  comment: "Roughly speaking, measurements on a ratio scale.
          $D + D \to D$, $D - D \to D$, $D * N \to D$, $D/D \to N$."

QuasiArithmetic valueSubclass: #Duration
  comment: "The ANSI standard amount-of-time class."

QuasiArithmetic valueSubclass: #Money ⊕
  comment: "Represents an amount of money in a specified currency."

QuasiArithmetic valueSubclass: #Quantity ⊕
  comment: "A physical measurement on a ratio scale."

QuasiArithmetic abstractSubclass: #Number
  comment: "Real and rational numbers."

Number abstractSubclass: #AbstractRationalNumber ⊕
  comment: "exact integers and rationals."

AbstractRationalNumber abstractSubclass: #Integer
  comment: "integers, immediate or boxed."

AbstractRationalNumber valueSubclass:: #Fraction
```

```
  comment: "$n/d$ where $d > 1$."

AbstractRationalNumber valueSubclass: #ScaledDecimal
  comment: "$n \times 10^s$ where $n$, $s$ are integers."

Number abstractSubclass: #AbstractFloatingPointNumber ⊕
  comment: "floating-point numbers in general."

AbstractFloatingPointNumber valueSubclass: #FloatE
  comment: "IEEE single precision."

AbstractFloatingPointNumber valueSubclass: #FloatD
  comment: "IEEE double precision."

AbstractFloatingPointNumber valueSubclass: #FloatQ
  comment: "IEEE double extended precision."

Number valueSubclass: #DualNumber ⊕
  comment: "$x + y\epsilon$ where $\epsilon^2 = 0$."

Number valueSubclass: #QuadraticSurd ⊕
  comment: "$(a + b\sqrt{k})/c$ where $a, b, c, k$ are integers,
        $k > 0$ is square-free, $c > 0$, and gcd$(a, b, c) = 1$."

Object valueSubclass: #Complex ⊕
  comment: "$x + iy$ where $i^2 = -1$."

Object valueSubclass: #Quaternion ⊕
  comment: "$x + iy + jz + kw$ where $i^2 = j^2 = k^2 = -1$.
```

3.2 Containers

Here's part of the class hierarchy for containers.

```
Object abstractSubclass: #Enumerable ⊕
  comment: "containers that can be traversed once"
  methods:
    do: aBlock
      "for each element of the container, invoke aBlock passing that element as its argument."
      self subclassResponsibility.
    inject: initial into: binaryBlock
      "Combine the elements, e.g., for sums."
      |accumulator|
      accumulator ← initial.
      self do: [:each |
        accumulator ← binaryBlock value: accumulator value: each].
      ↑accumulator
    detectSum: f
      ↑self inject: 0 into: [:s :each | s+(f value: each)]
    size
      "Answer the number of elements."
      ↑self inject: 0 into: [:n :each | n+1]
    sum
      ↑self inject: 0 into: [:s :each | s+each]

Enumerable abstractSubclass: #Collection
  comment: "containers that know their size and can be traversed multiple times"
  methods:
    detectMean: f
      ↑(self detectSum: f)/self size
    mean
      ↑self sum/self size
    size
      "This should be an O(1) operation."
      self subclassResponsibility.
```

The distinction between Enumerable and Collection is an important one. Smalltalk has a large number of "enumeration" methods that can be used to iterate over or summarise collections. Enumerable provides those methods that only require a single traversal; after using such a method you should not in general use the enumerable object again. Here is a typical example. We're going to print the sum of the numbers in the standard input stream.

```
s ← PluggableInputStream
        atEndBlock: [StdIn skipSeparators]
        nextBlock: [Number readFrom: StdIn]. "stream of numbers"
Transcript print: (EnumerableWrapper on: s) sum; cr.
```

In contrast, Collection is used for collections that can be traversed multiple times, such as arrays, strings, and internally generated collections.

4 Mean in a single traversal

With the definitions above, we cannot compute the mean of the stream of numbers in standard input, because that requires traversing the stream twice. We can fix that by changing the definition of #mean.

```
Enumerable
  methods:
    mean
    |s n|
    n ← 0.
    s ← 0.
    self do: [:each |
      n ← n + 1.
      s ← s + each].
    ↑s / n
  detectMean: aBlock
    |s n|
    n ← 0.
    s ← 0.
    self do: [:each |
      n ← n + 1.
      s ← s + (aBlock value: each)].
    ↑s / n
```

Here we have fused the two #inject:into: loops into one. We could add that as a new higher-order method:

```
Enumeration
  methods:
    inject: a0 into: aBlock and: b0 into: bBlock finally: wrapup
      |a b|
      a ← a0.
      b ← b0.
      self do: [:each | |t|
        t ← aBlock value: a value: b value: each.
        b ← bBlock value: a value: b value: each.
        a ← t].
      ↑wrapup value: a value: b
    mean
      ↑self inject: 0 into: [:s :n :each | s+each]
            and: 0 into: [:s :n :each | n + 1]
            finally: [:s :n | s / n]
```

Whichever definition we put into Enumerable, we have now solved the problem of being able to compute the mean of a stream. But we have also created a performance issue: if we use this definition with a stored collection, we are recomputing the #size, thus taking extra time.

This is where Object Orientation comes in handy. We can have *two* definitions of #mean: one in Enumeration which (re)computes the size, and one in Collection which uses the stored size.

However, ASTC's library includes #mean, #geometricMean, #harmonicMean, #rootMeanSquare, and #powerMean:, plus #trimmedMean, #trimmedMean:, #winsorisedMean, and #winsorisedMean:, which is more methods than I'd prefer to duplicate this way. So later in this note we shall look at the actual performance cost.

5 It's not just about Numbers

ANSI Smalltalk includes Duration and DateAndTime objects. Other Smalltalks have things like Money or Length or Quantity. One way of thinking about computing arithmetic means is to think in terms of types and operations. The types we need are

M Measurements, like temperatures on the Celsius or Fahrenheit scales, or timestamps measured relative to some epoch.

D Differences of measurements.

N Numbers, such as rationals or floating-point.

P Positive integers.

What are the operations we need on these types?

- $M + D \to M$
- $M - M \to D$
- $D + D \to D$
- $D/P \to D$

Examples include

M	D
Number	Number
Point	Point[2]
Point3D	Point3D
Money	Money
Duration	Duration
DateAndTime	Duration

In general, D classes correspond to statistical variables measured on a "ratio scale" and M classes correspond to statistical variables measured on an "interval scale". Directional and axial data as described above fall outside the traditional "nominal scale", "ordinal scale", "interval scale", "ratio scale" classification. Circular means having been considered above, Angle and Time are not considered here.

What happens if we try to compute the mean of a collection of sums of money? We run into the problem that the very first sum, 0+$1.24, say, is not defined.

Here's the simplest solution I can think of. The fiddling at the end of each method is to ensure that we get a division by zero error for an empty collection.

```
Enumeration
  methods:
    mean
```

```
      |s n|
      n ← 0.
      s ← nil.
      self do: [:each |
        n ← n + 1.
        s ← s ifNil: [each] ifNotNil: [each + s]].
      ↑(s ifNil: [0] ifNotNil: [s])/ n
```

```
Collection
  methods:
    mean
      |s|
      s ← nil.
      self do: [:each |
        s ← s ifNil: [each] ifNotNil: [each + s]].
      ↑(s ifNil: [0] ifNotNil: [s]) / self size
```

This avoids the initial each+0 at the price of an extra conditional branch on every iteration. Can we do better?

All the D classes I've had occasion to use have several more operations. One of them is

- $D - D \to D$

There is a popular but nonstandard operation #anyOne that returns an arbitrary element of a Collection. We can put these two observations together to get

```
Collection
  methods:
    mean
      |s|
      s ← self anyOne. "take any element"
      s ← s - s. "convert it to the right kind of zero."
      self do: [:each | s ← each + s].
      ↑s / self size
```

The #anyOne method is not available in Enumerable because that would require visiting an element twice.

6 Mean DateAndTime

We still have a problem. Measurements on an interval scale (like DateAndTime, or temperature on the Celsius or Fahrenheit scale, or height above mean sea level) cannot be added, but it is still meaningful to compute means of them.

If you choose to represent interval scale measurements just as numbers, instead of an appropriate data type, then you lose compile-time or run-time sanity checking. For example,

```
aTemperature + heightAboveMSL reciprocal
```

will be allowed. In compensation, the numeric mean will give you the right answer. So the problem here is to get means *and* sanity checking. Note that the harmonic or geometric mean of an interval scale measurement still doesn't make sense; it is only the arithmetic mean that fortuitously works.

The trick is that we can subtract any such measurement from all of the others, compute the mean, and add what we subtracted back. This relies on laws such as

- $(m_1 - m_2) + d = (m_1 + d) - m_2 = m_1 - (m_2 - d)$

```
Enumeration
  methods:
    mean
      |m s n|
      n ← 0.
      s ← nil.
      self do: [:each |
        n ← n + 1.
        s ifNil: [
          m ← each.
          s ← m - m "zero in D"]
         ifNotNil: [
          s ← (each - m) + s]].
      ↑m + (s / n)

Collection
  methods:
    mean
      |m s|
      m ← self anyOne.
      s ← m - m "zero in D".
      self do: [:each | s ← (each - m) + s].
      ↑m + (s / self size)
```

7 Handling runs

Smalltalk systems have a wide range of collection classes, which Java has only recently caught up with. Traditionally these include Bag and RunArray. A RunArray is a sequence stored as blocks of equal values, *e.g.,* 5 6 5 5 6 6 6 5 7 7 5 5 5 would be stored as (1,5), (1,6), (2,5), (3,6), (1,5), (2,7), (3,5) or as something from which that can be derived. (ASTC uses a representation that allows $O(\log n)$ indexing.) In order to permit efficient iteration of such collections and over "ordinary" collections, ASTC includes the following method:

```
Enumerable
  methods:
    runsDo: aBlock
      "Report the elements of the receiver to aBlock in runs.
      All the elements of a run have the same value, as defined by
      the receiver's notion of equality, typically #=.
      The arguments of aBlock are this common value and the length
      of the run. Runs need not be maximally long.
      For Sets, Bags, and AbstractSequences, they will be
      maximally long. The order in which the runs are reported is
      not in general defined, but for AbstractSequences the
      elements are reported in their natural order.
      This makes bag construction and traversing more efficient."
      self do: [:each | aBlock value: each value: 1].
```

With the aid of this method, we can compute the mean of a collection of Numbers or Durations *etc.* thus:

```
Enumerable
  methods:
    mean
      |s n|
      n ← 0.
      s ← nil.
      self runsDo: [:each :count |
        n ← n + count.
        s ← s ifNil: [each × count]
          ifNotNil: [each × count + s]].
      ↑s / n

Collection
  methods:
    mean
      |s|
      s ← self anyOne. "take any element"
      s ← s - s. "convert it to the right kind of zero."
      self runsDo: [:each :count |
        s ← each × count + s].
      ↑s / self size
```

This pays off for Bags and RunArrays, but for ordinary collections, like Arrays, just adds overhead.

8 Other Kinds of Average

There is a family of "power" means:

$$\mu_p = (n^{-1} \sum_{i=1}^{n} x_i^p)^{1/p}$$

Examples of this include the #harmonicMean ($p = -1$), the #rootMeanSquare ($p = 2$), and as limits, the #max ($p \to \infty$) and the #geometricMean ($p \to 0$).

Like the #median, #max depends only on ordering, so need not detain us here.

Power means are straightforward for numbers (N). For measurements on a ratio scale (D), we have to do this:

```
Enumerable
  methods:
    powerMean: p
      |s n scale|
      n ← 0.
      self do: [:each |
        n ← n+1.
        scale ifNil: [scale := each. s := 1] ifNotNil: [
          s ← (each / scale raisedTo: p) + s]].
      ↑(s / n raisedTo: p reciprocal) × scale
```

This is analogous to the subtract-then-add-back technique for computing means of interval scale measurements.

I am not aware of any way to define power means for measurements on an interval scale, such as DateAndTime. Except for #max, which works, I'm also unaware of any need for them.

9 A Digression on Durations

The next section presents benchmark results for several ways of calculating the mean of an array of Integers, of 64-bit FloatDs, and of Durations.

Straight away we run into a problem. What we want to know is the effect of the different algorithms. But the three Smalltalk systems I measured represent Durations differently and implement their operations differently. You cannot measure a mean method fairly if the underlying arithmetic is poorly implemented.

An important issue is the range of unboxed integers.

My system, ASTC, uses a scheme where the bottom 2 bits of a word are 00 for a SmallInteger, 10 for nil, Booleans, and Characters, and 01 for all other objects. This gives immediate numbers a range of -536,870,912 to +536,870,911 in a 32-bit environment.

Squeak and Pharo use a scheme where only 1 bit is taken from a word, giving a range of -1,073,741,824 to +1,073,741,823 in a 32-bit environment. Although

VisualWorks and Squeak are both derived versions of the original Smalltalk-80, the range of immediate numbers in VW matches ASTC, not Squeak and Pharo.

Integer results outside these ranges can be computed in any Smalltalk, they are just more expensive to calculate, to calculate with, and to store.

How big a range do we need for Durations? A number that's easy to remember is that a year is approximately 31 megaseconds, so a century is about 32 bits. (Which is why UNIX's `time_t` has historically been 32 bits, and why the 68-year gap between the UNIX epoch and the year 2038 gives rise to the "2038 bug".) To interoperate with data bases, we'd like a range of ±10000 years around 2000 AD, which is about 2^{39} seconds, but only 2^{23} days. If we want to measure times to higher precision than seconds, we're going to need 10, 20, or 30 bits more for millisecond, microsecond, or nanosecond precision. If we record Durations as single numbers, we'll need 49, 59, or 69 bits respectively.

Millisecond counts could be held as IEEE doubles (boxed in all three systems). Microsecond counts could be held in immediate integers in a 64-bit-only Smalltalk. Nanosecond counts would need arbitrary-precision integers, referred to using the Lisp term "bignums" henceforth.s.

Here is part of ASTC's definition:

```
QuasiArithmetic valueSubclass: #Duration
  instanceConstantNames: 'days milliseconds'
  methods for: 'checking'
    invariant
      (days isKindOf: Integer) and: [
      (milliseconds isMemberOf: SmallInteger) and: [
      (days >= 0 and: [
        milliseconds between: 0 and: 86399999]) or: [
      (days <= 0 and: [
        milliseconds between: -86399999 and: 0])]]]
  methods for: 'accessing'
    days
      ↑days
    millisecondPart
      ↑milliseconds
  methods for: 'comparing'
    = other
      ↑ other class == self class and: [
        other days = days and: [
        other millisecondPart = milliseconds]]
    < other
      ↑other days = days
        ifTrue: [milliseconds < other millisecondPart]
        ifFalse: [days < other days]
  methods for: 'arithmetic'
    pvtD: dd m: mm
      "This completes an addition or subtraction."
      |d m|
      d ← (mm quo: 86400000) + dd.
      m ← mm rem: 86400000.
      (0 < d and: [m < 0])
        ifTrue: [d ← d - 1. m ← m + 86400000].
      (d < 0 and: [m > 0])
        ifTrue: [d ← d + 1. m ← m - 86400000].
      ↑self
      days ← d.
      milliseconds ← m.
    + other
      ↑(other isKindOf: DateAndTime)
        ifTrue: [other + self]
        ifFalse: [self pvtClone pvtD: days + other days
                                 m: milliseconds + other millisecondPart]
    - other
        ↑self pvtClone pvtD: days - other days
                                 m: milliseconds - other millisecondPart
    * scale
      "may involve bignum or rational arithmetic"
    / scale
      "may involve bignum or rational arithmetic"
```

The aim here was to do as much as possible with immediate integers. For Durations less in magnitude than about $1\frac{1}{2}$ million years, this aim was met. Dividing two Durations is exact, meaning you can get non-integral rational results.

VisualAge Smalltalk uses a similar representation, and functionally similar code. It makes the split between seconds and microseconds rather than days and milliseconds. The important thing about splitting a Duration into two parts is that the less significant part should have its value bounded by a value U such that $2U - 1$ fits in a SmallInteger. VAST and ASTC both satisfy this.

Here is the corresponding code in Pharo, using ASTC syntax and edited lightly.

```
Magnitude subclass: #Duration
  instanceVariableNames: 'nanos seconds'
  methods for: 'checking'
    invariant "not actually in Pharo"
      ↑ (nanos isKindOf: Integer) and: [
        (seconds isKindOf: Integer) and: [
        (nanos between: -999999999 and: 999999999) and: [
        nanos sign * seconds sign >= 0]]]
  class methods for: 'instance creation'
    nanoSeconds: ns
      |s|
      s ← ns quo: NanosInSecond.
      ↑self basicNew seconds: s nanoSeconds: ns-(q*NanosInSecond)
  methods for: 'private'
    seconds: s nanoSeconds: ns
      seconds ← s.
      nanos ← ns rounded.
      [nanos < 0 and: [0 < seconds]] whileTrue: [
        seconds ← seconds + 1.
        nanos ← nanos + NanosInSecond].
      [0 < nanos and: [seconds < 0]] whileTrue: [
        seconds ← seconds - 1.
        nanos ← nanos - NanosInSecond].
  methods for: 'accessing'
    asNanoSeconds
      "breaks the style rules for capitalisation"
      ↑seconds * NanosInSecond + nanos
  methods for: 'comparing'
    = other
      ↑ self == other or: [
        self species = other species and: [
        self asNanoSeconds = other asNanoSeconds]]
    < other
      ↑self asNanoSeconds < other asNanoSeconds
  methods for: 'arithmetic'
    + other
      ↑self class nanoSeconds:
        self asNanoSeconds + other asNanoSeconds
    - other
      ↑self class nanoSeconds:
        self asNanoSeconds - other asNanoSeconds
    * other
      ↑self class nanoSeconds:
        (self asNanoSeconds * other) rounded
    / other
      ↑other isNumber
        ifTrue: [self class nanoSeconds:
          (self asNanoSeconds / other) rounded]
        ifFalse: [self asNanoSeconds / other asNanoSeconds]
```

This holds Durations to nanosecond precision. A nanosecond count will certainly fit in a Pharo SmallInteger, but if a Duration is more than ± 34 years the second count will be a bignum. Comparing, adding, or subtracting two Durations will involve creating (and then discarding) two bignums and doing bignum quotient and remainder. It appears that efficiency of Duration operations was not a priority in this design.

We can improve Pharo's efficiency here by adding three private methods and changing four others.

```
methods for: 'private'
  privateSeconds
    ↑seconds
  privateNanos
    ↑nanos
  privateSeconds: s nanos: ns
    seconds ← s.
    nanos ← ns.
methods for: 'comparing'
  = other
    ↑ other class == Duration and: [
      other privateSeconds = seconds and: [
      other privateNanos = nanos]]
  < other
    ↑seconds = other privateSeconds
      ifTrue: [nanos < other privateNanos]
      ifFalse: [seconds < other privateSeconds]
methods for: 'arithmetic'
  + other
    |s ns|
    s ← seconds + other privateSeconds.
    n ← nanos + other privateNanos.
    (n between: -999999999 and: 999999999) ifFalse: [
      n < 0 ifTrue: [n ← n + NanosInSecond. s ← s - 1]
        ifFalse: [n ← n - NanosInSecond. s ← s + 1]].
    s sign * n sign < 0 ifTrue: [
      s < 0 ifTrue: [n ← n - NanosInSecond. s ← s + 1]
        ifFalse: [n ← n + NanosInSecond. s ← s - 1]].
    ↑Duration basicNew privateSeconds: s nanos: n
  - other
    |s ns|
    s ← seconds - other privateSeconds.
    n ← nanos - other privateNanos.
    (n between: -999999999 and: 999999999) ifFalse: [
      n < 0 ifTrue: [n ← n + NanosInSecond. s ← s - 1]
        ifFalse: [n ← n - NanosInSecond. s ← s + 1]].
    s sign * n sign < 0 ifTrue: [
      s < 0 ifTrue: [n ← n - NanosInSecond. s ← s + 1]
        ifFalse: [n ← n + NanosInSecond. s ← s - 1]].
    ↑Duration basicNew privateSeconds: s nanos: n
```

This is not as simple, nor as obviously correct, as the existing Pharo code, but like the ASTC code, it avoids bignum arithmetic. The difference is astonishing. The three lines are the original times, the improved times, and the ratios.

2486	2478	3588	3564	2536	2544	3653	3630	6142	6181	4217
139	110	424	436	2536	2544	2723	2716	293	297	1583
17.9	22.5	8.5	8.2	1.0	1.0	1.3	1.3	21.0	20.8	2.7

There's a block where the ratio is close to 1. That's where there are many multiplications. Despite the improvement to #+ and #-, the benchmarks are still dominated by bignum arithmetic. The reason is simple: pushing up against the limits of the representation means that Durations are *stored* compactly, but nearly half of adds or subtracts still produce one bignum intermediate.

What about the premier commercial system, VisualWorks?

```
Magnitude subclass: #Duration
  instanceVariableNames: 'period scale'
  methods for: 'checking'
    invariant "not actually in VW"
      ↑ (period isKindOf: Number) and: [
        (scale isKindOf: Number) and: [
        0 < scale]]
  methods for: 'accessing'
    period
      ↑period
    scale ↑scale
  methods for: 'comparing'
    = other
      "You would not believe me if I told you the truth."
      ↑ (other isKindOf: Duration) and: [
      other scale = scale
        ifTrue: [period = other period]
        ifFalse: [(period * other scale) = (other period * scale)]
    < other
      "You would not believe me if I told you the truth."
      ↑other scale = scale
        ifTrue: [period < other period]
        ifFalse: [(period * other scale) < (other period * scale)]
  methods for: 'arithmetic'
    + other
      "Simplified."
      (other isKindOf: Duration) ifTrue: [
        |s|
        s ← self asSeconds asRational + other asSeconds asRational.
        ↑self class period: s numerator scale: s denominator].
      "other standard (and nonstandard) cases."
    - other
      ↑self + other negated
    negated
      ↑self class period: period negated scale: scale
```

The main idea here is that period * scale is the amount of time in seconds, and that if you only need low precision (like seconds), you should not have to pay for high precision (like nanoseconds).

VisualWorks 8.1 Personal Use Licence does not include scaling operations on Durations, although these operations are required by the ANSI Smalltalk standard. This deviation is explained in the class comment. I do not regard the argument given as having any validity, certainly not as justifying a situation where standard-conforming code cannot be executed at all. I show here perfectly satisfactory implementations. The Duration class in VW8.1 also fails to respond to the ANSI standard #seconds: message, requiring #fromSeconds: instead. Three methods were added:

```
Duration
    class methods for: 'instance creation'
        seconds: s
            ↑self period: s scale: Seconds
    methods for: 'arithmetic'
        * other
            ↑Duration period: period * other scale: scale
        / other
            ↑(other isKindOf: Duration)
                ifTrue: [self period * scale / other asSeconds]
                ifFalse: [Duration period: period / other scale: scale]
```

The *Runs measurements for VisualWorks are therefore either unfair to VW (because its Duration is clunky) or to the other systems (because the native speed here is "infinitely slow").

But what about adding and subtracting? One thing leaps to the eye: many such operations will involve Durations with the same scale. So it's worth investigating

```
methods for: 'arithmetic'
  + other
    (other isKindOf: Duration) ifTrue: [
      scale = other scale ifTrue: [
        ↑Duration period: period + other period scale: scale].
      everything else as before
  - other
    (other isKindOf: Duration) ifTrue: [
      scale = other scale ifTrue: [
        ↑Duration period: period - other period scale: scale].
      everything else as before
```

This is a simple and obvious change. What does it do to the measured times?

314	308	428	431	373	370	500	512	1193	1186	1622
226	225	283	291	287	284	346	357	484	499	401
1.4	1.4	1.5	1.5	1.3	1.3	1.4	1.4	2.5	2.4	4.0

This is not as dramatic as the improvement for Pharo, but it is enough that any kind of benchmarking would have been *wholly misleading* without examining Duration.

Another issue with VisualWorks is that in Squeak and Pharo, and in *astc* (which copied them), "α ifNil: [β] ifNotNil: [γ]" is a primitive control structure which is at least as efficient as "α isNil ifTrue: [β] ifNotNil: [γ]", but in VisualWorks it is emulated. Converting from ifNil: to ifTrue: doubled the speed of most of the methods that use it. The times for VW in the next section include this change.

GNU Smalltalk was not benchmarked because the version that works for me does not compile to native code, so the comparison would be misleading. (It does not define the ANSI DateAndTime class, but does have a DateTime class, which it strangely claims to be ANSI. It does have Duration. In a feat of surpassing strangeness, it derives the linear measure Duration from the circular measure Time, with the oddity that Duration midnight is defined.

Smalltalk/X was not benchmarked because it does not have the ANSI DateAndTime and Duration classes, although it has not entirely dissimilar Timestamp and TimeDuration classes. Like GNU Smalltalk, it derives a linear measure (TimeDuration) from a circular one (Time) and so uses a representation suitable for periods up to 24 hours for periods up to thousands of years.

10 Measurements

The goal of this mini-project was to decide what definitions to provide for #sum, #detectSum:, #mean, and #detectMean:.

The following measurements were made on an early 2011 15-inch MacBook Pro with a 2 GHz intel Core i7 processor and 8 GiB of 1333 MHz DDR3 memory running Mac OS X 10.11.5. The Smalltalk systems used were Pharo 5.0, VisualWorks Personal Use Licence 8.1, and ASTC built and using Apple LLVM version 7.3.0, which came with Xcode 7.3. The test data were Arrays of 999,999 Integers, 999,999 FloatDs, and 999,999 Durations. All times are in milliseconds. The last digit must be regarded as noise.

	Integer			Float			Duration		
Variant	Pharo	VW	Astc	Pharo	VW	Astc	Pharo	VW	Astc
Basic	57	89	28	20	40	26	N/A	N/A	N/A
BasicNoSize	62	92	30	21	41	28	N/A	N/A	N/A
NoZero	58	117	29	20	42	27	139	226	58
NoZeroNoSize	61	124	31	23	43	29	110	225	59
Shift	291	117	31	24	65	53	424	283	120
ShiftNoSize	295	121	33	28	69	55	436	291	121
Runs	310	127	43	26	68	56	N/A	N/A	N/A
RunsNoSize	302	129	44	24	70	58	N/A	N/A	N/A
RunsNoZero	316	131	43	21	71	55	2536	287	86
RunsNoZeroNoSize	317	134	44	22	74	57	2544	284	87
RunsShift	323	139	44	25	105	81	2723	346	150
RunsShiftNoSize	312	137	46	26	99	81	2716	357	151
Comp	101	170	71	35	117	106	N/A	N/A	N/A
CompNoSize	103	173	72	36	124	105	N/A	N/A	N/A
CompNoZero	108	171	71	36	120	104	293	484	249
CompNoZeroNoSize	109	174	72	38	122	108	297	499	249
Summary	147	67	102	35	174	103	N/A	N/A	N/A
SummaryNoZero	152	72	104	40	188	104	1583	401	1366

The differences between the first four methods are within noise level, so it makes sense to pick the most general implementation in that group.

The *Shift and *Runs implementations are clearly slower than the others. The only advantage of the *Shift methods is that they can compute the mean of a collection of DateAndTimes, while the other methods cannot. The price of not having this is that if you want to compute the mean of a collection of DateAndTimes you have to write

```
n ← DateAndTime now.
m ← aCollection detectMean: [:each | each - n].
m ← m + n.
```

which does not seem onerous, given the rarity of this usage.

The code that was originally written for ASTC was the meanRuns variant, but it has never been sent to a Bag or RunArray except for testing purposes.

Therefore, the #meanNoZeroNoSize variant has been chosen.

11 Arithmetic and Accuracy

Integers, Fractions, and ScaledDecimals are exact. Durations and Times recorded in milliseconds are exact. Angles recorded in degrees and fractional degrees are exact. Money and Quantity objects are exact if the underlying numbers are exact. QuadraticSurds are exact. DualNumbers, Complex numbers, and Quaternions are exact if their component are, similarly Points.

The sum of a compatible collection of exact items is exact; it is only when we divide by the number of items that inexactness is forced. We could of course divide by n instead of $nasFloat$, but since the standard deviation requires square roots, it seemed good to make the mean like the standard deviation.

The important thing about exact addition is that it is order-independent (associative). This is not true of floating-point addition.

```
((1 to: 1000) detectSum: [:each | 1/each])/1000
⇒  53362913282294785045591045624042980409652472280384
   26009710134924845626888949710175750609790198503569
   14090887315504680983784421721178850094643023443265
   66022502100278425632852081405544941210442510142672
   77029477471270891796396777961045322469242686646888
   82815820719848971051107968732493191555293970175089
   31564519976085734473014183284011724412280649074307
   70373668317005580029365923508858936023528585280816
   0759574737836655413175508131522517 / 7128865274665
   09305316638415571427292066835886188589304045200199
   11543240875811114994764441519138715869117178170195
   75256512980264067621009251465871004305131072686268
   14320019660997486274593718834370501543445252373974
   52989631456749821282369562328237940110688092623177
   08861979540791247754558049326475737829923352751796
   73524804246363805113703433121478174685087845348567
   80218880753732499219956720569320290993908916874876
   72697950931603520000000
(as above) asFloatD
⇒  0.0074854708605503d0
((1 to: 1000) detectSum: [:each | 1.0e0/each])/1000
⇒  0.00748548e0
((1000 to: 1 by: -1) detectSum: [:each |1.0e0/each])/1000
⇒  0.00748547e0
```

from which we see that for this problem, adding from small to big gave a more accurate answer than adding from big to small. This is well known, and while it's not the optimal strategy in every case, it is a good one.

William Kahan, to whom we owe IEEE 754 arithmetic, came up with a way of doing sums more accurately. Kahan's compensated summation algorithm is

```
compensatedSum: aBlock
  |r e t y|
  r ← e ← 0
  self do: [:each |
    t ← r.
    y ← (aBlock value: each) + e.
    r ← y + t.
    e ← (t - r) + y].
  ↑r
```

The reference here is

- *Further remarks on reducing truncation errors*, W. Kahan, 1965, CACM vol. 8, no. 1

Using #compensatedSum:, the answer is 0.00748547e0 whether we sum up or down. In double precision we get 0.0074854708605503d0 summing up or down. This is indeed more accurate than the#detectSum: method.

In effect, this method doubles the working precision within the loop, taking four floating-point add/subtract operations instead of one. We see from the table above that the "compensated" algorithms take 2 to 4 times as long as the simple ones, depending on the costs of adding the observations. We also see that the differences between systems are comparable to the differences between algorithms, so that this is a price we might be willing to pay if we know little about the numeric properties of the data.

Compensated summation can obviously be applied to the calculation of linear variances and circular means as well.

12 On-line *vs* off-line

All of the algorithms presented above work through all the data before reporting a summary, in the manner of an off-line algorithm. Computing the variance and/or standard deviation requires two passes over the data. Computing trimmed or Winsorised means or standard deviation even requires sorting the data. (More precisely, that requires the top and bottom k elements of the data to be found, for some k. That can be done by taking $2k$ elements, sorting them, then running two heaps and pushing the remaining elements through the heaps at $O((n-2k).(1+\lg k))$ work, for a grand total of $O(n.\lg k)$ instead of $O(n.\lg n)$. That can be done with $O(2k)$ workspace.)

Another approach, following Simula 67, is to have a Summary class into which we can push elements, that we can ask for the current mean and so on at any time. Java 1.8 includes a DoubleSummaryStatistics class; ASTC's Summary is older and more capable.

Algorithms for incrementally and stably updating the mean and higher moments can be found in

- *Algorithm AS 52: Calculation of power sums of deviations about the mean*, C. C. Spicer, Applied Statistics, 21, 1972, pp 226–227.

The algorithm for updating the mean found therein can be found in

- *Updating Mean and Variance Estimates: An Improved Method*, D. H. D. West (University of Dublin), Communications of the ACM, 22.9, September 1979, pp 532–535.

If n is the number of observations seen so far, and m is the current estimate of the mean, the response to a new observation x is

$n \leftarrow n + 1$
$m \leftarrow (x - m)/n + m$

This is widely used. For example, *IBM SPSS 20 Algorithms* states that the DESCRIPTIVES command uses Spicer's provisional means algorithm.

There is a problem with this technique. Floating-point arithmetic being what it is, you can easily find a strictly ascending sequence of numbers where the final value of m is arbitrarily far from the true value. Here is an example in C:

```
int main(void) {
    float m = 2.0f;
    float x;
    for (int n = 2; n < 0xFFFFFFF; n++) {
        x = 2.0f + (n-1)*FLT_EPSILON;
        m = (x - m) / n + m;
    }
    printf("m = %g mean = %g x = %g\n", m, (1+x)/2, x);
    return 0;
}
```

The output is

```
m = 2 mean = 18 x = 34
```

The author has never seen this defect mentioned before and would be delighted to use a better method but does not know of one.

Objects that you can push data into have a name in C++ are called output iterators. Smalltalk has an analogue, (output) streams. Here's part of ASTC's class hierarchy:

```
Object abstractSubclass: #Stream
  comment: "covers input and output streams"
  methods:
    close
      "close stream and release resources"
    critical: aBlock
      "lock receiver, perform aBlock, unlock receiver.
        Use if a stream is shared by threads, rare."
    species
      "what would #contents be"

Stream abstractSubclass: #BasicOutputStream ⊕
  comment: "general output"
  methods:
    flush
      "force buffered output if any to destination"
    nextPut: anObject
      "accept/process one item"
      self subclassResponsibility.
    next: count put: anObject
      1 to: count do: [:i | self nextPut: anObject].
    nextPutAll: aCollection
    aCollection do: [:each | self nextPut: each].

BasicOutputStream abstractSubclass: #OutputStream
  comment: "character output streams"
```

ASTC now has four interfaces for computing means, not counting "robust" versions.

works on	linear means	circular means
Enumerable	`mean.st`	`circmean.st`
BasicOutputStream	`summary.st`	`circsum.st`

So now we have two ways to compute the mean of an array of numbers:

```
mean := theArray mean.

mean := (Summary new) nextPutAll: theArray; mean.
```

Which is faster? Which is more accurate?

Since Summary is intended to provide a range of statistics, not just the mean, it clearly does more work. For the purposes of benchmarking, a stripped down version of Summary was created. The Summary and SummaryNoZero entries in the table above benchmark a Number-only method and a method that works on Duration, Money, and so on as well, just like the distinction between Basic and NoZero. Using the provisional means algorithm is indeed a lot slower.

A test array was used to probe the accuracy of the various techniques. To make the problem hard, it is a large array of increasing single-precision floats, a situation where simple summing does not do well.[3]

```
fs := (1 to: 999999) collect: [:k | k asFloatE ln].
```

value	error	technique
12.815517	N/A	computed in 128-bit IEEE arithmetic
12.8917	+0.076183	basic algorithm
12.81552	+0.000003	Kahan's compensated sum
12.75901	-0.056507	provisional means

As expected, Kahan's compensated sum algorithm worked well. The provisional means algorithm was closer to the true value than the basic algorithm. The same pattern was observed with double precision numbers.

When we consider exact numbers, the provisional means algorithm runs into a problem. Let's just take 2,3,4 as our data.

m_0 = 0.
m_1 = (2-0)/1 + 0 = 2.
m_2 = (3-2)/2 + 2 = (3/2) + 2 = (5/2).
m_3 == (4-(5/2))/3 + (5/2) = (3/2)/3 + (5/2) = (1/2)+(5/2) = 3.

[3]Its mean is of course ln(999999!)/999999. You probably don't want to read about how to compute factorials efficiently and which Smalltalks don't.

It gets the exact answer, at the price of doing a lot of *rational* arithmetic along the way, whereas the basic algorithm just adds up integers until the very last step. To avoid this, the benchmark results above kept n as a floating-point number.

One possibility would be a hybrid algorithm using provisional means for inexact numbers and sums for exact numbers. At this point Smalltalk really starts to suffer from not having a Haskell-level type system.

My recommendation is to use whichever gets the job done, but to remain aware of the issues. Computing means is not as simple as it looks.