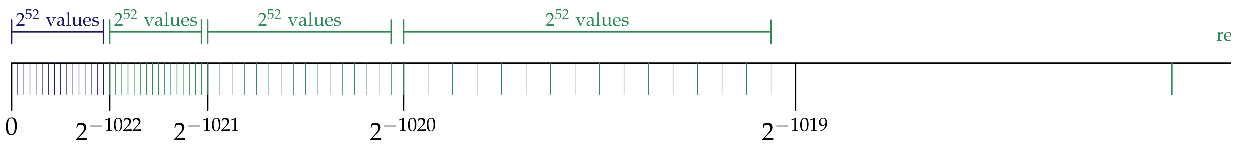
Machine Arithmetic and Numerical Error

16 September 2019

DATA 1010

Machine Arithmetic

Arbitrary real numbers can't be represented in a computer. Indeed, if we choose a fixed amount of memory like 64 bits, then only 2^{64} real numbers can be represented. The figure below shows the set of (nonnegative) numbers which are chosen for the `Float64` system, which is the most common system for real arithmetic in modern computers.



The numbers are much more densely packed around zero because we want to be able to represent numbers with a high precision *relative to the size of the number*, and this requires that gaps between representable numbers be smaller close to zero.

Problem 1

Show that $x - y = 0$ if and only if $x = y$ (where the subtraction is a `Float64` operation). Show that this is *not* the case in the variant of the `Float64` system which lacks subnormal numbers.

Problem 2

Select the numbers which are exactly representable as a `Float64`:

$$2^{1024}, \quad \frac{4}{3}, \quad -2^{-1074}, \quad \frac{3}{8}, \quad 0.0, \quad 1.5, \quad 0.8$$

Problem 3

In Julia, the function `nextfloat` returns the next largest representable value. Predict the values returned by the following lines of code, and then run them to confirm your predictions.

```
log2(nextfloat(15.0)-15.0)
log2(nextfloat(0.0))
log2(1.0 - prevfloat(1.0))
```

Problem 4

Investigate the rounding behavior when the result of a calculation is exactly between two representable values. Define `ϵ = 0.5^52` and check whether `1.0 + 0.5ϵ == 1.0`. Repeat with 1.5 in place of 0.5 (and appropriate changes made to the right-hand side). What does the rounding rule appear to be?

Problem 5

Between which two consecutive powers of 2 are the (Float64) representable numbers exactly the integers in that range?

Condition number

Problem 6

Consider a function $S : \mathbb{R} \to \mathbb{R}$. If the input changes from a to $a + \Delta a$ for some small value Δa, then the output changes to approximately $S(a) + \frac{\mathrm{d}}{\mathrm{d}a} S(a)\, \Delta a$. Calculate the ratio of the relative change in the output to the relative change in the input, and show that you get

$$\frac{a \frac{\mathrm{d}}{\mathrm{d}a} S(a)}{S(a)}.$$

Problem 7

The expression $\frac{a \frac{\mathrm{d}}{\mathrm{d}a} S(a)}{S(a)}$ is called the **condition number** of S. (We can also discuss the condition number of a *problem* which maps initial data a to the solution $S(a)$).

Show that the condition number of $a \mapsto a^n$ is constant, for any $n \in \mathbb{R}$.

Problem 8

Show that the condition number of the function $a \mapsto a - 1$ is very large for values of a near 1.

Well-conditioned problems and stable algorithms

If the condition number of a problem is very large, then small errors in the problem data lead to large changes in the result. A problem with large condition number is said to be **ill-conditioned**.

An algorithm used to solve a problem is **stable** if it is approximately as accurate as the condition number of the problem allows. In other words, an algorithm is *unstable* if the answers it produces have relative error many times larger than $\kappa\epsilon_{\text{mach}}$.

Problem 9

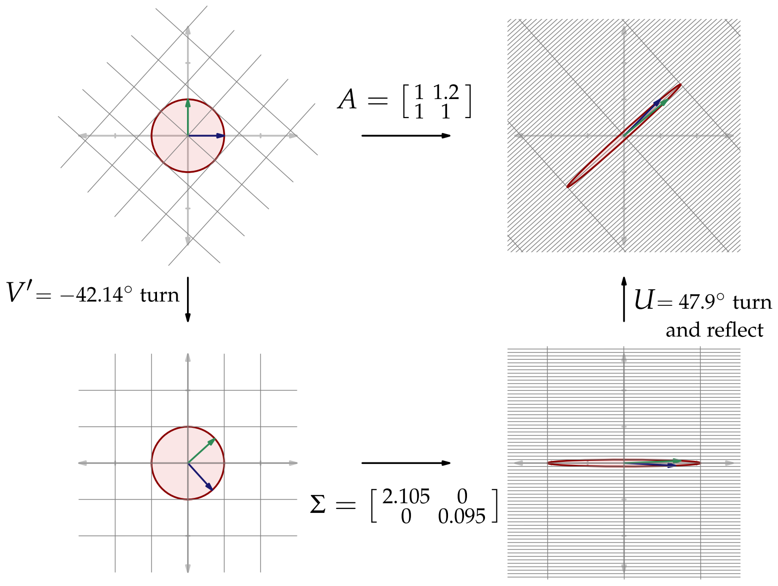
Find a stable algorithm for evaluating the function $f(x) = \sqrt{1+x} - 1$, and compare the stable algorithm to the order-of-operations algorithm implemented in the function f below.

```
[ ]   f(x) = sqrt(1+x) - 1
      # f_stable(x) =
      # f(1e-12), f_stable(1e-12)
```

Problem 10

Show that the condition number of a matrix A is equal to the ratio of its largest and smallest singular values.

Hint: consider two vectors on the domain side of the picture: $\mathbf{v}$ and $\mathbf{v} + \mathbf{e}$ (where $\mathbf{e}$ represents error). If we want the relative error to be magnified as much as possible under the transformation A, we want the error to be magnified as much as possible while the norm of $\mathbf{v}$ is shrunk as much as possible.



$A = \begin{bmatrix} 1 & 1.2 \\ 1 & 1 \end{bmatrix}$
$V' = -42.14^\circ$ turn
$U = 47.9^\circ$ turn and reflect
$\Sigma = \begin{bmatrix} 2.105 & 0 \\ 0 & 0.095 \end{bmatrix}$

Challenge Problem

Calculating inverse square roots is a very common task in graphics-intensive settings like video games. In the late 1990's, the following algorithm for approximating the inverse square root function appeared in the source code of the game *Quake III Arena*. The operator `>>` shifts the bits in the underlying representation over by 1 position, and `reinterpret` creates a new instance of the given type whose bits are the same as the bits of the given value.

```
function invsquareroot(x::Float32)
    y = 0x5f3759df - (reinterpret(Int32,x) >> 1)
    z = reinterpret(Float32,y)
    z * (1.5f0 - (0.5f0*x)*z*z)
end
```

If you are amazed by the appearance of the magic constant `0x5f3759df` (*Side Note*: this is syntax for an unsigned, 32-bit integer) so was the original author. You can see their code comments on the Wikipedia entry for Fast Inverse Square Root: https://en.wikipedia.org/wiki/Fast_inverse_square_root

Evaluate this function with a few input values and determine its relative error on each. Define another function which calculates the inverse square root in the obvious way (`1/sqrt(x)`) and check that the one above actually does run faster. As a double extra bonus, figure out why this code works.