probability Assignment

Please open each step by clicking on ▶ and see the explanations.

Current InverseCumulativeNormal

It converts a **probability (x)** between 0 and 1 into the corresponding **z-value** on the normal distribution.

That means:

Probability x	Output
0.5	0.0 (the center)
0.975	1.96 (right tail)
0.025	-1.96 (left tail)

This header file has 2 functions central_value_baseline (used when x is near the **middle** (like around 0.5)) and tail_value_baseline used when x is in the **tails** (very small or very large values, close to 0 or 1).

The class already works correctly, it gives correct z-values. But it uses a slow method called bisection, which tries many guesses until it finds the right answer. The job is to make it faster by replacing that slow guessing with a mathematical formula that approximates the result directly.

Run the program in the terminal (**Folder step 1**): Bisection
g++ main.cpp -o InvZ -O2 -std=c++17
./InvZ

Type	x	z
scalar	1e-12	-7.03448
scalar	1e-06	-4.75342
scalar	0.01	-2.32635
scalar	0.1	-1.28155
scalar	0.5	4.96308e-24
scalar	0.9	1.28155
scalar	0.99	2.32635
scalar	0.999999	4.75342
scalar	1	7.03448
vector	0.0001	-3.71902
vector	0.01	-2.32635
vector	0.1	-1.28155

Type	x	z
vector	0.3	-0.524401
vector	0.5	4.96308e-24
vector	0.7	0.524401
vector	0.9	1.28155
vector	0.99	2.32635
vector	0.9999	3.71902

Replacing and running bisection with rational approximation

The goal is to replace the slow "bisection" with a **fast formula** (called a rational approximation).
A rational approximation is word for a fraction of two polynomials. This formula can be made very accurate, and it's very fast.

The curve (polynomial) behaves differently in the middle and at the ends, so we use two formulas:

Region	Description	Example ⨯	What to use
Central region	Around 0.5	0.3–0.7	"Central formula"
Tails	Near 0 or 1	0.0001 or 0.9999	"Tail formula"

I should keep public API means; I am not allowed to change how the class looks from outside.

There are already well-known formulas for the inverse normal function. I can use Peter J. Acklam's approximation that is very common and accurate. It uses two sets of coefficients, one for the centre and one for the tails.

After I get a fast first result, I can make it a bit more accurate using **Halley's method**.
That's just one extra line of math that corrects the small error.

Run the program in the terminal (**Folder step 2**): Rational approximation
g++ main.cpp -o InvZ -O2 -std=c++17
./InvZ

Type	x	z
scalar	1e-12	-7.03448
scalar	1e-06	-4.75342
scalar	0.01	-2.32635
scalar	0.1	-1.28155
scalar	0.5	0
scalar	0.9	1.28155

Type	x	z
scalar	0.99	2.32635
scalar	0.999999	4.75342
scalar	1	7.03448
vector	0.0001	-3.71902
vector	0.01	-2.32635
vector	0.1	-1.28155
vector	0.3	-0.524401
vector	0.5	0
vector	0.7	0.524401
vector	0.9	1.28155
vector	0.99	2.32635
vector	0.9999	3.71902

Refinement & Vector calculation

After running the rational formula, refinement was done and these are results. If I want to write clearer:

First:
The function was tested with **quartile (quantile) values**, special points like 0.01, 0.1, 0.5, 0.9, 0.99, to check if it gives the correct results at important positions of the normal distribution.

Second:
Then it was tested with **other random or different x values**, to see if the function behaves correctly everywhere (from near 0 to near 1).

Third:
The same function was calculated in **two different ways**:

- **Scalar mode:** calculates one value at a time, like a small loop that goes one by one.
  Example:

  for x in xs:

  z = g(x)

- **Vector mode:** calculates **many values at once**, using array or SIMD operations.
  Example:

  z = g(xs)

This uses the CPU (or GPU) to do several computations together, it's much faster.

Scalar Tests:

x	Hex	z
1e-12	0x1.19799812dea11p-40	-7.034483818
1e-06	0x1.0c6f7a0b5ed8dp-20	-4.753424314
0.01	0x1.47ae147ae147bp-7	-2.326347874
0.1	0x1.999999999999ap-4	-1.281551564
0.5	0x1p-1	0
0.9	0x1.cccccccccccdp-1	1.281551564
0.99	0x1.fae147ae147aep-1	2.326347874
0.999999	0x1.ffffde7210be9p-1	4.753424314
1	0x1p+0	inf

Vector Tests:

x	z
0.0001	-3.719016482
0.01	-2.326347874
0.1	-1.281551564
0.3	-0.5244005133
0.5	0
0.7	0.5244005133
0.9	1.281551564
0.99	2.326347874
0.9999	3.719016482

Symmetry Checks (g(1−x) + g(x)):

x	g(x)	g(1−x)	sum
1e−06	-4.753424314	4.753424314	-5.813127757e−12
0.01	-2.326347874	2.326347874	0
0.1	-1.281551564	1.281551564	0
0.3	-0.5244005133	0.5244005133	-1.110223025e−16
0.45	-0.1256613469	0.1256613469	1.110223025e−16

Vector Path Timing:

Processed Elements	Total Time (ms)	Avg Time per Element (ms)
5,000,000	26.7618	5.35236e−06

https://colab.research.google.com/drive/1FNUDU4xAWwl9hPrl4z5wHRc6K29IYez6?usp=sharing

Last Step

I ran Python code to generate coefficients

python fit_export_icn.py > coeffs.txt

and replaced the coefficient in central and tail to my header file. I changed main.cpp to main_test.cpp and result is:

Round-trip (Phi(icn(x)) - x):

max abs error = 3.10862446895e-15

mean abs error = 2.63524573907e-17

99.9%ile abs error = 2.22044604925e-16

Max symmetry violation (g(1-x)+g(x)) sample = 0.000432693554034

Min positive step in g(x) (sampled grid) = 1.13871727514e-06

Max relative derivative error (sampled) = 106.965158996

Scalar throughput: 31.26329 ns/eval (icn)

Baseline bisection: 2418.758 ns/eval (small sample 2e5)

Speedup over baseline (rough) = 77.367353212x

Vector path: 33.6679 ms for 1000000 elems => 33.6679 ns/elem

Naive loop: 35.5616 ms => 35.5616 ns/elem

Vector speedup = 1.05624645434x

It needs to work on to get correct output.