c code to calculate the average of one hundred floats with reduced precision

8:07 π.μ.

Claude responded: Key reduced-precision choices:

c

```c
#include <stdio.h>

int main(void) {
    float values[100];

    // Fill with sample data
    for (int i = 0; i < 100; i++)
        values[i] = (float)(i + 1) * 1.1f;

    // Sum using float (reduced precision vs double)
    float sum = 0.0f;
    for (int i = 0; i < 100; i++)
        sum += values[i];

    float average = sum / 100.0f;

    printf("Average: %.2f\n", average);  // %.2f = reduced decimal precision
    return 0;
}
```

Key reduced-precision choices:

- `float` instead of `double` — 32-bit single precision (~7 decimal digits) vs 64-bit double (~15 digits)
- `%.2f` format — prints only 2 decimal places
- `100.0f` divisor — keeps the division in `float` arithmetic (dropping the `f` would silently upcast to `double`)

If you want even less precision, you could accumulate into an `int` (truncating decimals entirely) or use `%.0f` to round the output to a whole number.

Claude finished the response

You said: c code to compress 100 floats with reduced precision

c code to compress 100 floats with reduced precision
8:10 π.μ.

Claude responded: Three compression strategies are shown, each trading precision for size:

c

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

// Compress float to N mantissa bits (reduced precision)
// IEEE 754: 1 sign + 8 exponent + 23 mantissa bits
uint32_t compress_float(float f, int mantissa_bits) {
    uint32_t bits;
    memcpy(&bits, &f, sizeof(bits));

    uint32_t mask = ~((1u << (23 - mantissa_bits)) - 1);
    bits &= mask;  // Zero out low-order mantissa bits

    return bits;
}

// Pack floats to 16-bit half-precision (1+5+10 bits)
uint16_t float_to_half(float f) {
    uint32_t bits;
    memcpy(&bits, &f, sizeof(bits));

    uint16_t sign     = (bits >> 31) & 0x1;
    int32_t  exp      = ((bits >> 23) & 0xFF) - 127 + 15;
    uint32_t mantissa = (bits >> 13) & 0x3FF;  // top 10 mantissa bits

    if (exp <= 0)  return sign << 15;           // underflow -> ±0
    if (exp >= 31) return (sign << 15) | 0x7C00; // overflow  -> ±inf

    return (sign << 15) | ((uint16_t)exp << 10) | (uint16_t)mantissa;
}

float half_to_float(uint16_t h) {
    uint32_t sign     = (h >> 15) & 0x1;
    uint32_t exp      = (h >> 10) & 0x1F;
    uint32_t mantissa =  h        & 0x3FF;

    uint32_t f_exp      = (exp == 0) ? 0 : (exp - 15 + 127);
    uint32_t f_mantissa = mantissa << 13;
    uint32_t bits       = (sign << 31) | (f_exp << 23) | f_mantissa;

    float out;
    memcpy(&out, &bits, sizeof(out));
    return out;
}

int main(void) {
    // --- Generate 100 sample floats ---
    float original[100];
    for (int i = 0; i < 100; i++)
        original[i] = (float)i * 0.1f + 0.01f * (float)(i % 7);
```

```
    // ── Method 1: Truncate mantissa bits (lossless exponent/sign) ──────────
    printf("=== Method 1: Truncated mantissa (10 bits kept) ===\n");
    int keep_bits = 10;  // keep top 10 of 23 mantissa bits
    uint32_t compressed32[100];
    for (int i = 0; i < 100; i++)
        compressed32[i] = compress_float(original[i], keep_bits);

    printf("Orig       -> Compressed (first 5)\n");
    for (int i = 0; i < 5; i++) {
        float restored;
        memcpy(&restored, &compressed32[i], sizeof(restored));
        printf("  %10.6f -> %10.6f  (err %.2e)\n",
               original[i], restored, original[i] - restored);
    }
    // Storage: still 4 bytes/float, but bottom 13 bits are zero -> compresses
    // better with gzip/zstd downstream. Or pack the 19 significant bits tightly.
    printf("Raw storage: 100 x 4 = 400 bytes (but entropy-compressible)\n\n");

    // ── Method 2: Half-precision (16-bit) ──────────────────────────────────
    printf("=== Method 2: Half-precision float16 (16 bits) ===\n");
    uint16_t compressed16[100];
    for (int i = 0; i < 100; i++)
        compressed16[i] = float_to_half(original[i]);

    printf("Orig       -> Restored  (first 5)\n");
    for (int i = 0; i < 5; i++) {
        float restored = half_to_float(compressed16[i]);
        printf("  %10.6f -> %10.6f  (err %.2e)\n",
               original[i], restored, original[i] - restored);
    }
    printf("Storage: 100 x 2 = 200 bytes (50%% vs float32)\n\n");

    // ── Method 3: Custom N-bit quantisation (e.g. 8-bit fixed-point) ────────
    printf("=== Method 3: 8-bit uniform quantisation ===\n");
    float min_val = original[0], max_val = original[0];
    for (int i = 1; i < 100; i++) {
        if (original[i] < min_val) min_val = original[i];
        if (original[i] > max_val) max_val = original[i];
    }
    float range = max_val - min_val;

    uint8_t compressed8[100];
    for (int i = 0; i < 100; i++)
        compressed8[i] = (uint8_t)(((original[i] - min_val) / range) * 255.0f + 0.5f);

    printf("Orig       -> Restored  (first 5)\n");
    for (int i = 0; i < 5; i++) {
        float restored = min_val + (compressed8[i] / 255.0f) * range;
        printf("  %10.6f -> %10.6f  (err %.2e)\n",
               original[i], restored, original[i] - restored);
    }
    printf("Storage: 100 x 1 = 100 bytes + 2 floats metadata (75%% saving)\n");

    return 0;
}
```

Three compression strategies are shown, each trading precision for size:

Method 1 — Mantissa truncation (still `float32`)

Zero out the low-order mantissa bits in-place. The floats stay 32-bit, but with many zero bits they compress extremely well with a downstream codec (zstd, zlib). Choose `keep_bits` (1–23):

`keep_bits`	Relative error	Effective bits
23	0 (lossless)	32
10	~0.1%	~19
7	~1%	~16

Method 2 — Half-precision `float16` (50% size)

Packs each float into 16 bits (1 sign + 5 exp + 10 mantissa). **Best general-purpose choice** — ~3 decimal digits of precision, handles the full float range (with clamping), and is the format used by GPUs/ML hardware.

Method 3 — 8-bit uniform quantisation (75% size)

Maps the entire array onto `[0, 255]` using the observed min/max. Only 1 byte per float + 2 floats of metadata. Good when:

- the value range is **known and bounded**
- ~2 decimal digits of precision is enough (e.g. sensor readings, normalised weights)

Compile & run:

bash

```
gcc -O2 -o compress compress.c && ./compress
```