# Assignment 2

> 컴퓨터소프트웨어학부 2024017201 최선웅
> 12034 수치해석

```
#include <stdio.h>
#include <math.h>
#include <windows.h>

#include "nr_wrap.h"
#include "muller.h"

static long func_calls, df_calls;
static long total_iters[6], total_fcalls[6], total_dfcalls[6];
static double total_time[6];
static const char *names[6] = {
  "Bisection", "LinearInterp", "Secant",
  "Newton", "NewtonSafe", "Muller"
};

float f_wrapper(float x) {
  func_calls++;
  return bessj0(x);
}
void fdf_wrapper(float x, float *f, float *df) {
  func_calls++;
  df_calls++;
  *f = bessj0(x);
  *df = -bessj1(x);
}

enum { I_BISEC=0, I_FLSP, I_SEC, I_NEW, I_SAFE, I_MULL };

static LARGE_INTEGER qpc_freq;

void timer_init(void) {
  QueryPerformanceFrequency(&qpc_freq);
```

```
}

double timer_ms(void) {
  LARGE_INTEGER now;
  QueryPerformanceCounter(&now);
  return (double)now.QuadPart * 1000.0 / (double)qpc_freq.QuadPart;
}

int main(void) {
  timer_init();

  const int NX = 1000;
  const float x1 = 1.0f, x2 = 10.0f, tol = 1e-6f;
  float xb1[NX], xb2[NX];
  int nb = 0;

  zbrak(f_wrapper, x1, x2, NX, xb1, xb2, &nb);
  printf("Found %d roots in [%.1f, %.1f]\n\n", nb, x1, x2);

  for (int i = 0; i < 6; i++) {
    total_iters[i] = total_fcalls[i] = total_dfcalls[i] = 0;
    total_time[i] = 0.0;
  }

  for (int k = 1; k <= nb; k++) {
    float a = xb1[k], b = xb2[k];
    printf("Root %d: bracket [%.6f, %.6f]\n", k, a, b);

    for (int m = 0; m < 6; m++) {
      double t0 = timer_ms();
      long before_f = func_calls, before_df = df_calls;
      float root;
      long iters = 0;

      switch (m) {
        case I_BISEC:
          root = rtbis(f_wrapper, a, b, tol);
          break;
```

```
      case I_FLSP:
        root = rtflsp(f_wrapper, a, b, tol);
        break;
      case I_SEC:
        root = rtsec(f_wrapper, a, b, tol);
        break;
      case I_NEW:
        root = rtnewt(fdf_wrapper, a, b, tol);
        break;
      case I_SAFE:
        root = rtsafe(fdf_wrapper, a, b, tol);
        break;
      case I_MULL:
        root = muller(f_wrapper, a, b, tol);
        iters = get_muller_iterations();
        break;
    }

    double t1 = timer_ms();
    long fc = func_calls - before_f;
    long dfc = df_calls - before_df;
    double tms = t1 - t0;

    if (m != I_MULL) iters = fc;

    total_iters[m]   += iters;
    total_fcalls[m]  += fc;
    total_dfcalls[m] += dfc;
    total_time[m]    += tms;

    printf("  %-12s root=%.8f iters=%4ld f-calls=%4ld df-calls=%4ld time
=%.3fms\n",
           names[m], root, iters, fc, dfc, tms);
  }
  printf("\n");
 }

 printf("Summary (averaged over %d roots)\n", nb);
```

```
  printf("Method        AvgIters  AvgFcalls  AvgDf  AvgTime(ms)\n");
  printf("------        --------  ---------  -----  ----------\n");
  for (int m = 0; m < 6; m++) {
    printf("%-14s %8.1f %9.1f %6.1f %12.3f\n",
           names[m],
           (double)total_iters[m]   / nb,
           (double)total_fcalls[m]  / nb,
           (double)total_dfcalls[m] / nb,
           total_time[m]            / nb
    );
  }
  return 0;
}
```

이 `main.c` 코드는 Numerical Recipes의 실수형 루틴들을 사용해 Bessel 함수 J0(x)의 근을 여러 구간에서 찾고, 6가지 방법(이분법, 선형 보간, 할선법, 뉴턴-랩슨법, 브래킷 뉴턴, 밀러)의 성능을 고해상도 타이머로 측정, 비교하는 코드이다.

## 동작 개요

- `zbrak` 로 구간에서 J0(x)의 부호 변화를 탐지해 브래킷들을 수집하고, 각 브래킷에 대해 6가지 근 찾기 알고리즘을 실행한다.
- 각 방법에 대해 반복 횟수(내장 카운트 없는 루틴은 함수 호출로 근사), 함수/도함수 호출 수, 경과 시간을 측정하고, 루트별 결과를 출력한다.

## 타이머와 충돌 해결

- Window의 고해상도 QPC(Query Performance Counter)로 보다 더 정밀한 경과 시간을 얻는다.
- NR 헤더의 fmin, select가 시스템 헤더와 충돌하는 문제를 피하기 위해 래퍼 해더(nr_wrap.h)에서 이름을 일시 치환한 뒤 nr.h를 포함하는 방식으로 해결했다.

## 출력값

- 각 루트에 대해 `root, iters, f-calls, df-calls, time` 을 한 줄로 보고하고, 마지막에 루트들에 대한 평균을 표로 요약한다.

- 이를 바탕으로 수렴 속도와 효율(평균 반복, 평균 호출, 평균 시간)을 비교하여 방법들의 상대적 성능을 논의한다.

## 출력값의 분석

```
Found 3 roots in [1.0, 10.0]

Root 1: bracket [2.403996, 2.412997]
  Bisection    root=2.40482569 iters=  16 f-calls=  16 df-calls=   0 time=0.00
1ms
  LinearInterp root=2.40482569 iters=   5 f-calls=   5 df-calls=   0 time=0.0
01ms
  Secant       root=2.40482569 iters=   5 f-calls=   5 df-calls=   0 time=0.00
0ms
  Newton       root=2.40482569 iters=   3 f-calls=   3 df-calls=   3 time=0.00
0ms
  NewtonSafe   root=2.40482569 iters=   5 f-calls=   5 df-calls=   5 time=0.
000ms
  Muller       root=2.40482545 iters=   2 f-calls=   6 df-calls=   0 time=0.001
ms

Root 2: bracket [5.517978, 5.526978]
  Bisection    root=5.52007771 iters=  16 f-calls=  16 df-calls=   0 time=0.001
ms
  LinearInterp root=5.52007818 iters=   5 f-calls=   5 df-calls=   0 time=0.00
0ms
  Secant       root=5.52007818 iters=   5 f-calls=   5 df-calls=   0 time=0.000
ms
  Newton       root=5.52007818 iters=   2 f-calls=   2 df-calls=   2 time=0.00
0ms
  NewtonSafe   root=5.52007818 iters=   4 f-calls=   4 df-calls=   4 time=0.0
01ms
  Muller       root=5.52007818 iters=   2 f-calls=   6 df-calls=   0 time=0.001
ms

Root 3: bracket [8.649918, 8.658917]
  Bisection    root=8.65372849 iters=  16 f-calls=  16 df-calls=   0 time=0.00
```

```
3ms
  LinearInterp root=8.65372753 iters=   5 f-calls=   5 df-calls=   0 time=0.00
1ms
  Secant       root=8.65372753 iters=   5 f-calls=   5 df-calls=   0 time=0.001
ms
  Newton       root=8.65372753 iters=   2 f-calls=   2 df-calls=   2 time=0.001
ms
  NewtonSafe   root=8.65372753 iters=   4 f-calls=   4 df-calls=   4 time=0.
001ms
  Muller       root=8.65372753 iters=   2 f-calls=   6 df-calls=   0 time=0.001
ms

Summary (averaged over 3 roots)
Method        AvgIters  AvgFcalls  AvgDf  AvgTime(ms)
------        --------- ---------- ----- -----------
Bisection         16.0      16.0    0.0       0.001
LinearInterp       5.0       5.0    0.0       0.001
Secant             5.0       5.0    0.0       0.001
Newton             2.3       2.3    2.3       0.001
NewtonSafe         4.3       4.3    4.3       0.001
Muller             2.0       6.0    0.0       0.001

[Extra] Solve f(x)=e^{-x}-x in [0.0,1.0]
[Extra] Bracket [0.560000,0.570000] → root=0.56714332  f(root)=-5.96e-0
8  f-calls=4 df-calls=4 time=0.000ms
```

## 근사 정확도

- 세 방법 모두 근 값이 2.4048257, 5.5200782, 8.653728 근방으로 일치해 동일한 허용오차 내에 수렴했다는 점을 확인할 수 있다. 특히 같은 브래킷에서 시작했음에도 최종 값이 소수점 7~8자리에서 일치해 알고리즘 간 정확도 차이는 미미하다.

## 반복 호출 지표

- 이분법: 각 루트마다 iters = 16, f-calls = 16 으로 가장 많은 반복을 사용했다. 선형 수렴 특성상 수렴 속도가 가장 느리다.

- 선형 보간(rtflsp), 할선법(rtsec): 각 루트마다 iter $\approx$ 5, f-calls $\approx$ 5로 이분법 대비 크게 감수한다. 수렴 차수가 1보다 큰 초선형으로 알려져 있어 상대적으로 빠르다.
- 뉴턴-랩슨법(rtnewt): Root1에서 iter = 3, Root2에서 iter = 2, 평균 약 2.3회로 가장 적다. 도함수 호출(df-calls)이 동일횟수로 발생하지만, 2차 수렴으로 반복 수 자체가 최저다.
- 브래킷 뉴턴(rtsafe): iters가 4~5회 수준으로 뉴턴보다 다소 크지만, 브래킷 보장이 있어 안정성과 수렴성을 동시에 확보한다.
- 밀러(muller): iter = 2, f-calls = 6으로 반복 횟수는 매우 적고, 한 반복에서 함수 값을 3점에서 평가하므로, f-calls가 반복 횟수보다 크다. 도함수 불필요 장점과 높은 수렴 차수(이론적으로 약 1.84) 덕에 빠르게 수렴한다.

### 시간 지표

- 보고된 time 값이 0.000 ~ 0.001ms로 매우 작다. 고해상도 타이머를 사요했지만, 각 루틴의 실행 자체가 짧아 측정 잡음이 존재한다. 상대 순위는 반복/호출 지표와 일치하며, 뉴턴 $\geq$ 밀러 $>$ 할선법 $\geq$ 선형보간 > 브래킷 뉴턴 $>$ 이분법 이다.

### rtsafe를 이용한 $f(x) = e^{-x} - x$ 근 찾기

- `exp_f_wrapper` 와 `zbrak` 을 사용하여 $[0.0, 1.0]$ 구간에서 함숫값의 부호가 바뀌는 소구간을 찾는다.
- `rtsafe` 로 근 계산
  - 찾은 첫 번째 소구간에 대해 `rtsafe` 를 호출해 방정식 $e^{-x} - x = 0$의 근을 구한다.
  - `exp_f_anf_df` 는 함수값과 도함수를 동시에 계산하므로, `rtsafe` 내부에서 뉴턴 단계와 이분 단계 모두를 활용해 안전하고 빠르게 수렴한다.
- 결론: $e^{-x} - x = 0$의 해 $x \approx 0.56714332$를 `rsafe` 로 계산한 결과가 이어진다.