Calculus Homework 6

介紹

本次作業，我將使用以下三種自然常數的定義，分別設計出三個計算自然常數的程式。

1. $\ln e = \int_1^e \frac{1}{t} dt = 1$
2. $e = \lim_{\delta \to 0} (1 + \delta)^{\frac{1}{\delta}} = \lim_{n \to \infty} (1 + \frac{1}{n})^n$
3. $e = \sum_{n=0}^{\infty} \frac{1}{n!}$

此外，我會以三個程式「達到特定準度所需的運行次數」當作運行效率的參考，比較使用各個定義之間的效率差異。

程式設計

我使用 C++ 來撰寫本次作業所需的所有程式。此章節我將進一步探討程式的實作細節。

完整程式碼：
https://gist.github.com/nekogravitycat/4cd05d772bb6919104f221c7c563a292

程式架構

本次作業，我將三個定義的運算放在單一的 `main.cpp` 檔案裡。執行時，三個定義算出的答案將會列印在程式執行的終端視窗。

程式碼

```
#define ull unsigned long long
#define ld long double

typedef struct Result {
  ld e;
  ull iterations;
} Result;

void method_a(ld epsilon, Result *result) { /*...*/ }
void method_b(ld epsilon, Result *result) { /*...*/ }
void method_c(ld epsilon, Result *result) { /*...*/ }

int main() {
```

```
  const ld epsilon = 1e-10;

  Result result_a;
  method_a(epsilon, &result_a);

  Result result_b;
  method_b(epsilon, &result_b);

  Result result_c;
  method_c(epsilon, &result_c);

  // Print epsilon, and the results (e and iterations) of each method.
}
```

- 為求精準，整數皆使用 `unsigned long long` 儲存、浮點數皆使用 `long double` 儲存。為了使程式碼精簡，前者定義為 `ull`、後者定義為 `ld`
- 定義結構 `Result` 來儲存每個函式的 e 估計值、運算次數
- 定義一個變數 `epsilon` 為計算要求的精度，此變數將是演算法停止的判斷條件（須達到此精度才停止運算）
- 定義三個函式，分別實作三種定義的運算
  1. `method_a()` ：$\int_1^e \frac{1}{t} dt = 1$
  2. `method_b()` ：$e = \lim_{n \to \infty}(1 + \frac{1}{n})^n$
  3. `method_c()` ：$e = \sum_{n=0}^{\infty} \frac{1}{n!}$

演算法一：$\int_1^e \frac{1}{t} dt = 1$

令函數 $f(x) = \frac{1}{x}$，則定積分 $\int_1^e f(t)dt$ 的值為 1。

實作原理

使用黎曼和計算 1 到 x 的定積分。當黎曼和 < 1 時，則增加 x 的值；當黎曼和 ≥ 1 時，即終止積分，此時的 x 值將近似於 e。

精準度要求

要用黎曼和計算定積分 $\int_1^b f(t)dt$，過程中會將 1 到 b 之間切成 n 等份，其中每等份的寬度為 $\frac{1+b}{n}$。但是在此題中，我們並不知道 b 的值，導致無計算出 1 到 b 間每等份的寬度。於是，在此我們直接令每等份的寬度為 $\frac{1}{n}$。

若想使 b 更加接近 e，只要增加 n 的值，使得每一等份的寬度減少（相當於 1 與 e 之間切成更多等份），便可得到更加精確的結果。

如果計算出的黎曼和與 1 的差小於精度要求 `epsilon`，則認為此時的 b 足夠接近 e；反之則使 n 增加 1，並重新計算一次黎曼和。

運行次數計算

定義整數變數 `iterations` 來紀錄求 e 所需的運算次數。

程式碼

```
void method_a(ld epsilon, Result *result) {
  ld riemann = 2, x = 1;
  ull iterations = 0;
  for (ull n = 2; riemann - 1.0 > epsilon; n++) {
    riemann = 0;
    for (ull i = 0; riemann < 1; i++) {
      x = 1 + (1.0/n)*i;
      riemann += (1.0/x) * (1.0/n);
      iterations++;
    }
  }
  result->e = x;
  result->iterations = iterations;
}
```

備註：此段程式碼在 `epsilon` $< 10^{-7}$ 時可能要運行較長時間，請耐心等候。

演算法二：$e = \lim_{n\to\infty}(1 + \frac{1}{n})^n$

令函數 $f(x) = (1 + \frac{1}{x})^x$，當 x 趨近於無限時，$f(x)$ 會趨近於 e。

實作原理

使用足夠大的 x，讓 $f(x)$ 的值足夠接近 e。

精準度要求

因為當 $x > 0$ 時，$f(x)$ 為嚴格遞增函數，故 $f(x+1) > f(x)$ 恆成立。

當 x 愈大，$f(x+1) - f(x)$ 愈小。當 x 足夠大時，$f(x+1) - f(x)$ 將會小於 `espilon`，則認為此時的 $f(x+1)$ 足夠接近 e。

運行次數計算

定義整數變數 `iterations` 來紀錄求 e 所需的運算次數。

值得注意的是，為了計算 $(1 + \frac{1}{x})^x$，需要用到 `<cmath>` 中的 `pow()` 函式（`pow((1+1.0/n), n)`）。但 `pow()` 函式裡的運算次數沒辦法被計入 `iterations` 變數

中，於是我在迴圈內實作了快速冪（Binary Exponentiation）以用來計算 $(1+\frac{1}{x})^x$，並將 `iterations` 的記數放入快速冪的迴圈內。

程式碼

```
void method_b(ld epsilon, Result *result) {
  ld limit = 1, limit_prevoius = 0;
  ull iterations = 0;
  for (ull n = 1; limit - limit_prevoius > epsilon; n++) {
    limit_prevoius = limit;
    // limit = pow((1+1.0/n), n);
    // Binary Exponentiation: a^b
    limit = 1;
    ld a = 1 + 1.0/n;
    ull b = n;
    while (b != 0) {
      if (b & 1) { limit *= a; }
      a *= a;
      b = b >> 1;
      iterations++;
    }
  }
  result->e = limit;
  result->iterations = iterations;
}
```

演算法三：$e=\sum_{n=0}^{\infty}\frac{1}{n!}$

令函數 $f(x)=\sum_{n=0}^{x}\frac{1}{n!}$，當 x 趨近於無限時，$f(x)$ 會趨近於 e。

實作原理

使用足夠大的 x，讓 $f(x)$ 的值足夠接近 e。

精準度要求

因為當 $x>0$ 時，$f(x)$ 為嚴格遞增函數，故 $f(x+1)>f(x)$ 恆成立。

當 x 愈大，$f(x+1)-f(x)$ 愈小。當 x 足夠大時，$f(x+1)-f(x)$ 將會小於 `espilon`，則認為此時的 $f(x+1)$ 足夠接近 e。

運行次數計算

函式裡的 `n` 是用來計算 $f(x)$ 式子中 $n!$ 的整數變數。每運算一次 $f(x)$，`n` 值都會加一。故以變數 `n` 來紀錄求 e 所需的運算次數。

程式碼

```
void method_c(ld epsilon, Result *result) {
  ld sum = 1, sum_previous = 0;
  ull n = 1;
  for (ull denominator = 1; sum - sum_previous > epsilon; n++) {
    sum_previous = sum;
    denominator *= n;
    sum += 1.0 / denominator;
  }
  result->e = sum;
  result->iterations = n - 1;
}
```

運行結果與效率比較

透過修改 `epsilon` 參數、運行程式並記錄不同精度要求下，各個演算法所得出的 e 值、及各個算法所需的計算次數。

運行結果

1. 設 `epsilon` 為 10^{-6} 時，三種演算法的運行結果

	Calculated e	Iterations
Method A	2.7164898746	924,412
Method B	2.7171171001	10,790
Method C	2.7182818011	10

2. 設 `epsilon` 為 10^{-8} 時，三種演算法的運行結果

	Calculated e	Iterations
Method A	2.7180507210	55,595,539
Method B	2.7181652532	146,843
Method C	2.7182818283	12

3. 設 `epsilon` 為 10^{-10} 時，三種演算法的運行結果

	Calculated e	Iterations
Method A	2.7182133117	632,559,464
Method B	2.7182701692	1,850,653
Method C	2.7182818285	14

效率比較

從上面三組運行結果可以看出，在相同精度要求下，各演算法的效率為：

Method C >> Method B > Method A

而且隨著精度要求提升，此差距將會愈來愈顯著。

※比較依據為：演算法達到精度要求 `epsilon` 所需的運算次數。運算次數愈大，花費時間愈久，則效率愈差。

結論

使用定義 $e = \sum_{n=0}^{\infty} \frac{1}{n!}$ 來計算 e 是三個定義之中，最有效率的方法。