Report: Floating-Point Arithmetic and Error Analysis

- ***AUTHORS:***

  MENGQIAN XU (21306077), EDOARDO CARRA' (21400562)

- ***Teacher :***

  Stef Graillat

Practical No. 1: Increasing the Accuracy

This report details the implementations and accuracy comparison of algorithms involved in polynomial evaluation and summation. The following sections outline the algorithms implemented using MATLAB.

1. Compensated Horner's Scheme

Let $p(x) = \sum_{i=0}^{n} a_i x^i$ be a polynomial with floating-point coefficients. The following implementations focus on compensated Horner's scheme and associated error-free transformations.

1.1 Classic Horner's Scheme

The classic Horner's method evaluates the polynomial using a recursive approach.

```
function res = classicHorner(a, x)
    n = length(a);
    res = a(n);
    for i = n-1:-1:1
        res = res * x + a(i);
    end
end
```

This function can be tested against MATLAB's `polyval` function for validation:

```
p = [ 2 3 5 6];
x = 0.56;
```

```
classic_result = classicHorner(p, x);
polyval_result = polyval(p, x);
disp(['Classic Horner: ', num2str(classic_result)]);
disp(['Polyval: ', num2str(polyval_result)]);
```

1.2 Error-Free Transformations (EFT)

To increase precision, error-free transformations (EFT) for summation and product of floating-point numbers are implemented.

Algorithm 1.1: FastTwoSum

```
function [x, y] = FastTwoSum(a, b)
    x = a + b;
    y = (a - x) + b;
end
```

Algorithm 1.2: TwoSum

```
function [x, y] = TwoSum(a, b)
    x = a + b;
    z = x - a;
    y = (a - (x - z)) + (b - z);
end
```

Test

```
a = 1.2345;
b = 0.5678;
[x, y] = FastTwoSum(a, b);
[x_ef, y_ef] = TwoSum(a, b);

disp('FastTwoSum：');
disp(['x = ', num2str(x)]);
disp(['y = ', num2str(y)]);
disp('TwoSum ：');
disp(['x = ', num2str(x_ef)]);
disp(['y = ', num2str(y_ef)]);
```

```
normal_sum = a + b;
disp("The result of an ordinary addition：");
disp(['sum = ', num2str(normal_sum)]);
```

The steps of two algorithms seem to be the same, but the prerequisites and applicable situations are different:

- The first algorithm can perform simpler calculations in the y-processing step because it assumes specific conditions ($|a| \geq |b|$).
- The second method is more general and handles rounding errors in all cases to ensure the accuracy of the calculation results.

Algorithm 1.3: Split

The Split algorithm splits a floating-point number into a high-precision and a low-precision part.

```
function [x,y] = Split(a)
    s = 27;
    factor = 2^s+1; % s = 27
    c = factor*a;
    x = c-(c-a);
    y = a-x;
end
```

Algorithm 1.4: TwoProduct

```
function [x, y] = TwoProduct(a, b)
    x = a * b;
    [a1, a2] = Split(a);
    [b1, b2] = Split(b);
    y = a2 * b2 - (((x - a1 * b1) - a2 * b1) - a1 * b2);
end
```

1.3 Compensated Horner's Scheme

The compensated Horner algorithm improves the accuracy of polynomial evaluation by using an error-free transform (EFT) at each step to compensate for errors in floating-point operations.

```
function res = CompensatedHorner(a, x)
    n = length(a);
    s = a(n);
    r = 0; % r represents the error
    for i = n-1:-1:1
            % Calculate the product and error of the curren
t item
        [p, pi] = TwoProduct(s, x);
        [s, sigma] = TwoSum(p, a(i));
        r = r * x + (pi + sigma);
    end
    res = s + r;
end
```

1.4 Symbolic Horner Scheme

```
function res = exactHorner(a, x)
    syms xs;
    poly = poly2sym(a);
    res = subs(poly, xs, x);
end
```

1.5 Condition Number

The condition number for a polynomial evaluation is defined as:

$$\mathrm{cond}(p, x) = \frac{\sum_{i=0}^{n} |a_i||x|^i}{|p(x)|}$$

The condition number is used to measure the sensitivity of a problem to input errors. The larger the condition number, the more unstable the problem.

```
function cond_num = condp(a, x)
    n = length(a) - 1;
    abs_sum = sum(abs(a) .* abs(x).^[0:n]);
    p_val = polyval(a, x);
    cond_num = abs_sum / abs(p_val);
end
```

Task 1.6: Testing the Polynomial $p_n(x) = (x-1)^n$ with $x = \mathrm{fl}(1.333)$

```
% Function to test Classic and Compensated Horner schemes
% This function computes and plots the direct relative errors
% and condition numbers for the polynomials p_n(x) = (x - 1)^

function testHornerSchemes()
  x = 1.333;
  n_values = 3:42;
  classic_errors = zeros(length(n_values), 1);
  compensated_errors = zeros(length(n_values), 1);
  condition_numbers = zeros(length(n_values), 1);
  for idx = 1:length(n_values)
    n = n_values(idx);
    a = poly(ones(1, n));
    true_value = polyval(a,
    classic_result = classicHorner(a, x);
    compensated_result = CompensatedHorner(a, x);
    classic_errors(idx) = abs((classic_result - true_value)
                                              / true_value)
    compensated_errors(idx) = abs((compensated_result-true_va
                                              / true_value)
    condition_numbers(idx) = condp(a, x);
  end
  figure;
  loglog(condition_numbers, classic_errors, '-o',
                'DisplayName', 'Classic Horner Scheme');
  hold on;
  loglog(condition_numbers, compensated_errors, '-x',
                'DisplayName', 'Compensated Horner Scheme');
  xlabel('Conditionnement');
  ylabel('Erreur directe relative');
  legend show;
  title('Conditionnement vs Erreur directe relative');
  grid on;
end
```

2. Accurate Summation Algorithms

This section compares different summation algorithms in terms of accuracy.

2.1 Classic Recursive Summation

```
function res = Sum(p)
    sigma = 0;
    for i = 1:length(p)
        sigma = sigma + p(i);
    end
    res = sigma;
end
```

2.2 Kahan's Summation Algorithm

The Kahan summation algorithm compensates for floating-point errors by tracking a compensation term that adjusts for the small errors introduced at each step of the summation.

```
function res = SCompSum(p)
    sigma = 0;
    e = 0;
    for i = 1:length(p)
        y = p(i) + e;
        [sigma, e] = FastTwoSum(sigma, y);
    end
    res = sigma;
end
```

2.3 Priest's Doubly Compensated Summation

The Priest algorithm achieves double accumulation compensation by sorting the elements by absolute value and using the `FastTwoSum` function to handle rounding errors. On each accumulation, a series of calculations are performed to minimize rounding errors.

```
function res = DCompSum(p)
    [~, I] = sort(abs(p), 'descend');
    p = p(I);
```

```
    s = 0;
    c = 0;
    for i = 1:length(p)
        [y, u] = FastTwoSum(c, p(i));
        [t, v] = FastTwoSum(s, y);
        z = u + v;
        [s, c] = FastTwoSum(t, z);
    end

    res = s;
end
```

2.4 Rump's Compensated Summation

The Rump algorithm implements compensation accumulation through iterative calculations and uses the `TwoSum` function at each iteration to handle rounding errors. The algorithm mainly aims to reduce rounding errors caused by floating-point operations.

```
function res = CompSum(p)
    pi = p(1); sigma = 0;
    for i = 2:length(p)
        [pi, qi] = TwoSum(pi, p(i));
        sigma = sigma + qi;
    end
    res = pi + sigma;
end
```

Accuracy Study

Studying the accuracy of different summation algorithms when the condition number of the summation changes is to explore the performance of these algorithms when dealing with summation tasks that are sensitive to numerical errors.

Research contents and methods:

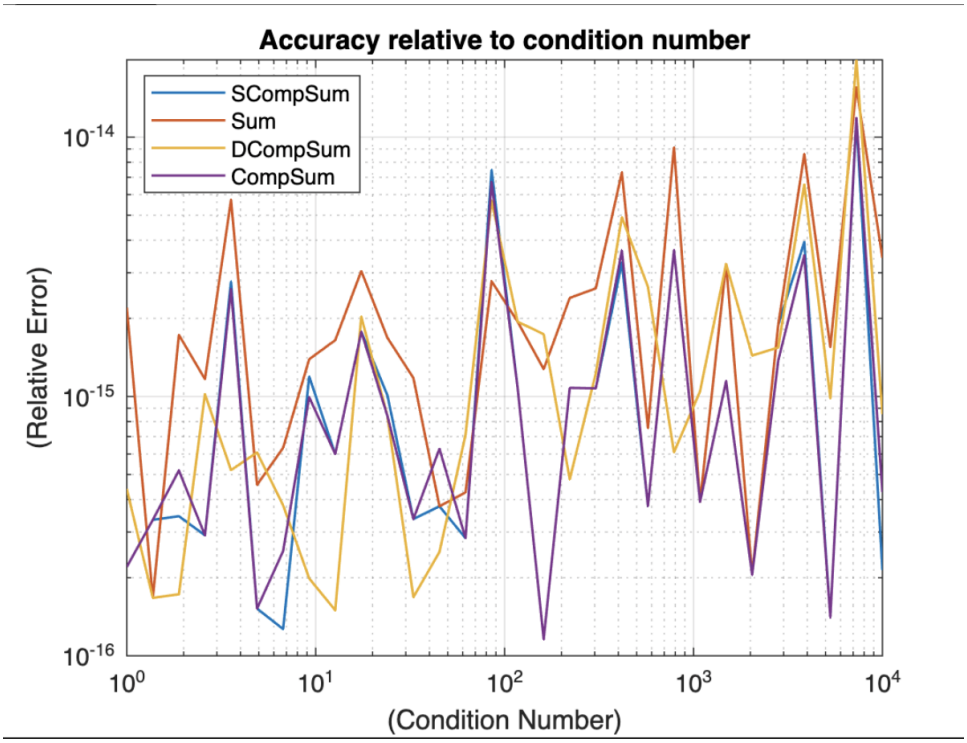
1. Generate a test data set: According to different condition numbers $\mathrm{cond}(p)$, generate a test data set p to be summed. Data sets with different condition numbers can be obtained by adjusting the numerical range.
2. Calculate the true sum: Use `sum` calculation methods to calculate the true sum of the test data set as a reference value.
3. Choose different summation algorithms.
4. Evaluate the accuracy of the summation algorithm: For each summation algorithm, calculate its relative error with respect to the true summation:

$$\text{Relative error} = \frac{|\text{Result of the algorithm} - \text{True sum}|}{|\text{True sum}|}$$

```
format longE;
num_datasets = 10;
function studyAlgorithmAccuracy(num_datasets)
  condition_numbers = logspace(0, 4, num_datasets);
  relative_errors = zeros(num_datasets, 4);
  n = 1000;
  for i = 1:num_datasets
    p = randn(n, 1) * condition_numbers(i);
    vrai_sum = sum(p);


    fast = Sum(p);  %  Sum
    kahan = SCompSum(p);  % SCompSum
    priest = DCompSum(p);  % DCompSum
    rump = CompSum(p);  % CompSum

    relative_errors(i, 1) = abs(double((fast - vrai_sum)
                                                    /vrai_sum
    relative_errors(i, 2) = abs(double((kahan - vrai_sum)
                                                    /vrai_sum
    relative_errors(i, 3) = abs(double((priest - vrai_sum)
                                                    / vrai_su
    relative_errors(i, 4) = abs(double((rump - vrai_sum)
                                                    /vrai_sum

    end
```

```
    figure;
    loglog(condition_number relative_errors,'LineWidth', 1);
    xlabel('(Condition Number)');
    ylabel('(Relative Error)');
    legend('Sum','SCompSum', 'DCompSum', 'CompSum',
                'Location', 'Best');
    title('Accuracy relative to condition number');
    grid on;
end
```

The result:



The smaller the relative error, the higher the accuracy of the algorithm.

- **Classic Summation**: Lowest precision, highly susceptible to rounding errors, only suitable when precision is not critical.
- **Kahan Summation**: Offers moderate precision, but its accuracy decreases as the condition number increases.
- **Priest's Summation**: An effective algorithm with consistently high precision across all condition numbers.

- **Rump’s Summation**: Equivalent to Priest's algorithm.