Assignment 1

Calculating Constants

Otto Hermann
otto.hermann@ucdconnect.ie
16203034

ACM40290

University College Dublin
27th September 2017

Contents

1 Write and test four small MATLAB functions

For each of these functions and throughout the assignment, it was assumed we were to use single-precision floating point numbers; however, all arguments and results stemming from the use of single-precision floating point numbers are generalisable to double-precision floating point numbers (or other arbitrarily precise floating point numbers).

1.1 Calculate π using $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$

```
% a while based implementation of the requested function
function p = Pi1(n)
    current = 0; % the incremented input value to the function
  estimate = 0; % the placeholder value for p
  % increment current and iteratively approximate Pi until the limit is reached
    while current <= n
    estimate = estimate + the_sum(current);
    current = current + 1;
  end
    % return value, given to 32 digits as a single-precision floating point number
  p = vpa(single(estimate * 4));
end

% the sum component for each n
function s = the_sum(n)
  s1 = sum_element(n, 1);
  s2 = sum_element(n, 3);
  s = s1 - s2;
end

% atomic component of the denominator
function s = sum_element(n, increment)
  denom_1 = 4 * n;
  denom_2 = increment;
  denom = denom_1 + denom_2;
  s = 1 / denom;
end
```

1.2 Calculate π using $\frac{\pi}{2} = \frac{2 \cdot 2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdot 8}{1 \cdot 3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdot 9} \ldots$

```
% a while based implementation of the requested function
function p = Pi2(n)
  current = 1; % the incremented input value to the function
  estimate = 1; % the placeholder value for p
    % increment current and iteratively approximate Pi until the limit is reached
  while current <= n
    estimate = estimate * product_element(current);
    current = current + 1;
  end
    % return value, given to 32 digits as a single-precision floating point number
  p = vpa(single(estimate * 2));
end

function p = product_element(n)
  numerator = 2 * n;
  denominator_1 = numerator - 1;
  denominator_2 = numerator + 1;
  p1 = numerator / denominator_1;
```

```
  p2 = numerator / denominator_2;
  p = p1 * p2;
end
```

1.3 Calculate Euler's Number using $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$

```
% a while based implementation of the requested function
function value = EulerN(n)
  current = 0; % the incremented input value to the function
  estimate = 0; % the placeholder for value
    % increment current and iteratively approximate e until the limit is reached
  while current <= n
    estimate = estimate + sum_element(current);
    current = current + 1;
  end
    % return value, given to 32 digits as a single-precision floating point number
  value = vpa(single(estimate));
end

function s = sum_element(n)
  s = 1 / factorial(n);
end
```

1.4 Calculate Euler's Constant using $\gamma = \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n} - ln(n)$

```
% a while based implementation of the requested function
function value = EulerC(n)
  current = 1; % the incremented input value to the function
  estimate = 0; % the placeholder for value
    % increment current and iteratively approximate gamma until the limit is reached
  while current <= n
    estimate = estimate + sum_element(current);
    current = current + 1;
  end
    % return value, given to 32 digits as a single-precision floating point number
  value = vpa(single(estimate - log(n)));
end

function s = sum_element(n)
  s = 1 / n;
end
```

2 Test functions and compare to given values

2.1 Test each function for $n = 10^1, 10^3, 10^6, 10^9$

$Pi1(n)$
$Pi1(10^1) = 3.096161603927612304687500000000$
$Pi1(10^3) = 3.141093254089355468750000000000$
$Pi1(10^6) = 3.141592264175415039062500000000$
$Pi1(10^9) = 3.141592741012573242187500000000$

$Pi2(n)$
$Pi2(10^1) = 3.067703723907470703125000000000$
$Pi2(10^3) = 3.140807628631591796875000000000$
$Pi2(10^6) = 3.141591787338256835937500000000$

$Pi2(10^9) = 3.14159250259399414062500000000000$

$EulerN(n)$
$EulerN(10^1) = 2.71828174591064453125000000000000$
$EulerN(10^3) = 2.71828174591064453125000000000000$

$EulerC(n)$
$EulerC(10^1) = 0.6263831853866577148437 5$
$EulerC(10^3) = 0.5777155756950338944287 7$
$EulerC(10^6) = 0.57721614837646484375$
$EulerC(10^9) = 0.57721567153930664062 5$

2.1.1 What's going on with $EulerN(n)$?

You'll notice that this function was only tested on $n = 10^1, 10^3$ and that both return the same value. In fact, if the function is run for $n = 10^6, 10^9$, the same value will be returned. This occurs because of an **overflow**: the denominator in each sum-element of $EulerN(n)$ is a factorial, which is defined as $n! = n \cdot n-1 \cdot n-2 \cdot \ldots \cdot 2 \cdot 1$ and, when using Matlab, $factorial(n)$ for $n = 10^3, 10^6, 10^9$ returns Inf, because the number generated is too large to be represented by a single-precision floating point number. Below is a more detailed explanation.

- The largest single-precision floating point number in Matlab is 3.4028×10^{38}
- Notice that $10! = 3,628,800 = 3.6288 \times 10^6$
- and $log_{10}(3.6288 \times 10^6) = log_{10}(3.6288) + log_{10}(10^6) = log_{10}(3.6288) + 6$
- but $log(3.6288) = \frac{log_2(3.6288)}{log_2(10)}$
- hence, $1 < log_2(3.6288) < 2$ and $3 < log_2(10) < 4$
- So, $\frac{1}{4} < log_{10}(3.6288) < \frac{2}{3}$ and $log_{10}(10!) > 6.25$
- Hence, $10^x! > \prod_{j=1}^{x} 10^{6.25j}$
- but $6.25 + 12.50 + \ldots + x = 6.25(1 + 2 + \ldots + x) = 6.25$(x)(x+1)
- so $10^x! > 10^{6.25(x^2+x)}$ and $10^x! < 10^{6.67(x^2+x)}$
- $10^{12.50} < 10! < 10^{13.34} = 10 \times 10^{12.34} < 3.4028 \times 10^{38}$, so 10! does not lead to an overflow (a similar argument can be made to show why $1/n!$ does not lead to an **underflow** i.e. become smaller than 1.17549×10^{-38})
- but $10^3! > 10^{75} > 3.4028 \times 10^{38}$ and $10^{75} = 10^{6.25(3^2+3)}$
- $EulerN(10^6)$ and $EulerN(10^9)$ are significantly larger than $EulerN(10^3)$ and will also lead to an overflow. Given than an overflow occurs in the $factorial(n)$ component of $EulerN(n)$, it should not come as a surprise that $1/factorial(n)$ would lead to an underflow.

But why are the values of $EulerN(10^1)$ and $EulerN(10^3)$ the same? A first question might be why $EulerN(10^3)$ doesn't return NaN: presumably because Matlab, when generating my function, will return the most precise calculation it can generate i.e. instead of returning NaN, it will not use the overflow and underflow values in my function.

2.1.2 OK, this part isn't specific to $EulerN$

So why then are the values of the same? We can explain this with the concept of Machine Epsilon: ϵ_m, defined as the distance between 1.0 and the next highest floating point number. $\epsilon_m = b^{1-p}$, where b is the base, and p is precision. When using single-precision floating point numbers in Matlab, $b = 2$ and $p = 24$, so $\epsilon_m = 2^{1-24} = 2^{-23} \approx 1.1921 \times 10^{-7}$.

ϵ_m sets a limit on the relative error when generating a floating point number: in our case, given that $\epsilon_m \approx 1.1921 \times 10^{-7}$, we should expect for large n that $f(n)$ be accurate up to seven digits.

In all of the functions, as n goes from 10^1 to 10^3 to 10^6 to 10^9, we see less relative error (see below for outputs) but also very similar results for $n = 10^6$ and $n = 10^9$. This is because the values of $f(n)$ for $n = 10^6$ are very close to the the upper bound of accuracy, while $f(n)$ for $n = 10^9$ exceeds it. An intuitive appreciation of this is that as n grows sufficiently large and correspondingly changes the output of $f(n)$, rounding or discretisation errors will not be an issue up to the precision identified by ϵ_m (in our case seven digits).

For smaller values of n, the higher levels of relative error can be attributed to discretisation error: the formulas we've used as a basis for the functions hold as $n \to \infty$, so low values of n truncate the approximation and reduce it's accuracy. Rounding error will also feature in these calculations and explain most of the relative error for larger values of n: beyond the level of precision identified by ϵ_m, we should expect there to be rounding errors because a floating point is a finite rational approximation of a real number.

2.2 Comparison with Matlab and given values

2.2.1 $Pi1(n)$

$matlab_{pi} = 3.14159274101257324218750000000000$
$acm40290_{pi} = 3.14159265358979311599796346854419$

Relative Error of $Pi1(n)$ compared to $matlab_{pi}$
$Pi1(10^1)$ has relative error of 0.014461179672295099152279362897 39
$Pi1(10^3)$ has relative error of 0.000158991621255433202009044180 16
$Pi1(10^6)$ has relative error of 0.000000151781977320020189381466 48
$Pi1(10^9)$ has relative error of 0.000000000000000000000000000000 00

Relative Error of $Pi1(n)$ compared to $acm40290_{pi}$
$Pi1(10^1)$ has relative error of 0.014461152247178921115278171782 85

$Pi1(10^3)$ has relative error of 0.00015896379814450689593741117278
$Pi1(10^6)$ has relative error of 0.00000012395444637380848007524037
$Pi1(10^9)$ has relative error of 0.00000002782753516505920288182097

2.2.2 $Pi2(n)$

$matlab_{pi} = 3.14159274101257324218750000000000$
$acm40290_{pi} = 3.14159265358979311599796346854419$

Relative Error of $Pi2(n)$ compared to $matlab_{pi}$
$Pi2(10^1)$ has relative error of 0.023519603970464708631027406227 08
$Pi2(10^3)$ has relative error of 0.000249909025677341745108606119 17
$Pi2(10^6)$ has relative error of 0.000000303563954640040378762932 96
$Pi2(10^9)$ has relative error of 0.000000075890988715521245921991 07

Relative Error of $Pi2(n)$ compared to $acm40290_{pi}$
$Pi2(10^1)$ has relative error of 0.023519576797422114822211369755 68
$Pi2(10^3)$ has relative error of 0.000249881205096502686985242351 77
$Pi2(10^6)$ has relative error of 0.000000275736427912676163032301 70
$Pi2(10^9)$ has relative error of 0.000000048063455548863487365451 87

2.2.3 $EulerN(n)$

$matlab_e = 2.71828174591064453125000000000000$
$acm40290_e = 2.71828182845904509079559829842765$

Relative Error of $EulerN(n)$ compared to $matlab_e$
$EulerN(10^1)$ has relative error of 0.000000000000000000000000000000 00
$EulerN(10^3)$ has relative error of 0.000000000000000000000000000000 00

Relative Error of $EulerN(n)$ compared to $acm40290e$
$EulerN(10^1)$ has relative error of 0.000000030367859449320633302704 66
$EulerN(10^3)$ has relative error of 0.000000030367859449320633302704 66

2.2.4 $EulerC(n)$

$acm40290_c = 0.57721566490153286554942724251305$

Relative Error of $EulerC(n)$ compared to $acm40290c$
$EulerC(10^1)$ has relative error of 0.085180502669677693461380840744 82
$EulerC(10^3)$ has relative error of 0.000866072811080531579364105709 83
$EulerC(10^6)$ has relative error of 0.000000837598425285079883906291 80
$EulerC(10^9)$ has relative error of 0.000000011499642571521917489008 05

2.2.5 A brief explanation

The change in the relative errors of the approximations of $Pi, e, and \gamma$ as n grows large can all be explained by the reasoning elaborated in 2.1.2: as n grows large, ϵ_m is the limit of the relative error, which in our case of single-precision floating point numbers means a precision to roughly seven digits. Each of the relative errors of the approximations approaches this limit (and in one case the approximation used is equivalent to the approximation generated by Matlab).

3 Final Questions

3.1 Are the two approximations of π equal?

No, they are not equal. See above for details.

3.2 Is one method better?

$Pi1(n)$ has a lower relative error than $Pi2(n)$, which, ceteris paribus, suggests that $Pi1(n)$ is better than $Pi2(n)$. However, the sum components of $Pi1(n)$ are ill-conditioned as $cond(Pi1(n)) = 2$ as $n \to \infty$. The product components of $Pi2(n)$ are well-conditioned for large n, with $cond(Pi2(n)) = 1$ as $n \to \infty$. But because the initial product component of $Pi2(n)$ is $\frac{4}{3}$, $Pi2(n)$ is also ill-conditioned.

On the whole, it seems that $Pi1(n)$ is a better method than $Pi2(n)$ for approximating π because it produces a more accurate value of Pi more quickly, which at the very least saves time and the cost of computation.

3.3 How accurately can π be computed if n is pushed to a very large number?

The comments in 2.1.2 provide a fuller answer, but given that $\epsilon_m \approx 1.1921 \times 10^{-7}$ for single-precision floating point numbers and $acm40290_{pi} = 3.14159265358979323846264338327 95$, we should expect an accurate prediction up to seven digits i.e. $3.141592r_1r_2\ldots$; this is consistent when reviewing the output for our approximation of π for $n = 6, 9$ (and also obtains if run for $n = 7, 8$, which also approach the limit of relative error).

The remaining digits $r_1r_2\ldots$ would be affected by some combination of discretisation and rounding error, with the severity of each contribution depending on the magnitude of h in $\frac{Mh}{2} + \frac{\epsilon|f(x)|}{h} \geq |f'(x) - \tilde{f}_{diff}(x;h)|$, which is the measure of total error (rounding and truncation).