PROJECT 10 - Non-floating-point Fractional Exponentiation Approach

Completed on 16-Nov-2016 (24 days)

Introduction >

Overview

This project explains the calculation methodology which I used to develop NumberParser's `Math2.PowDecimal`/`Math2.SqrtDecimal`, methods extending the native `double` (floating-point) implementations (i.e., `Pow`/`Sqrt` in `System.Math`) to `decimal` (not floating-point). It is eminently focused on fractional aspects because this is where the floating-point peculiarities are more influential, although there is also an integer exponentiation section.

Microsoft hasn't published the `Pow`/`Sqrt` source codes, but they are most likely very similar to equivalent functionalities in other languages or in mathematical libraries. While pre-analysing this whole situation, I took a look at the C library Cephes, which is an interesting reference for anyone looking for insights into all this.

Note that the floating-point peculiarities are precisely meant to efficiently deal with computationally-expensive calculations, what is certainly the case with exponentiation algorithms. Additionally, most of these algorithms are also relying on further assumptions and simplifications, at least for certain scenarios.

In summary, the goal of most of the existing (floating-point-based) exponentiation implementations is delivering reasonably accurate results as quickly as possible. On the other hand, the approach described here is mostly focused on delivering as accurate as possible results reasonably quickly.

.NET Decimal Type

As per the corresponding MSDN reference for C#, `decimal` is a 128-bit data type whose precision (e.g., `0.00000000000000000000000001m`) and range (e.g., `7e28m`) are around 28 digits. Contrarily to what happens with the other .NET decimal types (i.e., `double` and `float`), it isn't floating point.

The `decimal` type is defined by its high precision and that's why this has been an accuracy-oriented development since the very beginning. As explained in the corresponding section, the immediate result of such an intention is that `PowDecimal`/`SqrtDecimal` take into account most of the significant digits of the `decimal` type (note that the reasons for only "most of", the first 25 decimal digits, are also explained in the aforementioned other section).

The relatively big `decimal` range is also an important issue for the integer exponentiation aspects of this implementation. Note that this range is notably beyond the one of the biggest signed integer type (i.e., `long`, around ±9.2*10^18) and, consequently, no integer type might be used. The main consequence of such a limitation is the impossibility of relying on bitwise operations.

Algorithm >

Integer Exponentiation

The integer exponentiation method (i.e., the one being called when using `Math2.PowDecimal` with an integer exponent) is `PowIntegerPositive`. It implements an exponentiation by squaring approach and relies on the `decimal` type, either directly or indirectly through `Number`. As indicated by its name and equivalently to what happens with the fractional exponentiation, this method only expects positive inputs.

In calculations involving non-integer exponents, `PowIntegerPositive` is called at least twice: when determining the root of the exponent denominator and when raising that result to its numerator. In that first scenario, various calls are likely to occur because the root finding algorithm, despite converging quite quickly (custom-improved Newton-Raphson), usually needs various iterations.

Contrarily to what happens in the fractional part, the current version of the integer exponentiation algorithm might even compete in speed with the native version (i.e., `System.Math.Pow` with integer exponents). In any case, the following two issues would need to be taken into account:

- Just the fact of being part of the .NET Framework itself (`Pow`) or of a library created in a .NET language (`PowDecimal`) represents an unmeasurable uncertainty which affects the reliability of the measurements. Note that the open-source essence isn't helpful here as far as the `Pow`/`Sqrt` source codes haven't been made public. So in principle (i.e., without Microsoft's help), there is no way around this issue.
- `Number` is the most efficient NumberX, but it performs notably worse than a native type like `decimal`. For NumberParser, `Number` is undoubtedly the best option; for performance-measuring purposes, it would be better to replace these variables with native ones.

Fractional Exponentiation

The fractional exponentiation calculations are performed by `PowFractionalPositive`, which deals with all the exponentiation scenarios involving non-integer positive exponents. Its algorithm is a `Number`-based implementation of the expression *x^m/n = root(x, n)^m* and that's why is formed by three main parts:

- Fraction determination. The exponents are always `decimal` variables which need to be converted to fractions via `GetFraction`.

Note that the aforementioned method doesn't look for the most simplified alternative, but for a 10-based one (e.g., 0.4 outputting 4/10 rather than 2/5). The reason for relying on what seems a less efficient approach is explained in the next point. Additionally, this method avoids the denominator to grow beyond `1e25m`, what implies that only the first 25 decimal digits of the exponent will be considered. As explained in the corresponding (code) comment, this limitation is meant to avoid numbers beyond the `decimal` precision in the subsequent Newton-Raphson calculations. As far as this issue shouldn't be seen as a problem (the 26th decimal position of the exponent being relevant?!), I have no plans to fix it.

- Calculation of the *n* root. This is the most important part of the described implementation and of the current project. Here is where the non-floating-point high precision expectations become more problematic; the point where the generic implementations fail; what justifies the last part of this project.
  The next section, together with the aforementioned last part, explains all this code in detail. Nevertheless, its main ideas can already be summarised in the following points:
  - `GetNRoot` implements the Newton-Raphson method.
  - This method is highly dependent upon the initial guess (i.e., `fx[1]` right at the start).
  - The approach determining the initial guess works well with multiple-of-10 roots.
- Raising the calculated root to *m* by calling `PowIntegerPositive` (already described in the previous section).

Root Finding

As explained in the previous section, the n-root calculation is the most relevant part of the described fractional exponentiation algorithm. `GetNRoot` takes care of this through a `Number`-adapted optimised version of the Newton-Raphson method dealing with *f(x) = x^n - value*.

The last three sections of this project describe this algorithm in detail.

- The next section explains this approach and its implementation in `GetNRoot`.
- The last two sections (Exponential Proportionality and Method Improvement) analyse the Math2_Private_New_PowSqrt_RootIni.cs contents; a file which includes all the code in charge of determining the most adequate initial guess. Without this part, the Newton-Raphson method wouldn't be able to deliver what is required because of failing (i.e., getting stuck in an infinite-loop-like situation) under virtually any scenario involving a relevant number of digits.

Additionally, note that `GetNRoot` isn't meant to calculate any root for any number. It can only deal with certain inputs.

- Its algorithm assumes that `value` will always be a positive number. The whole approach determining the most adequate initial guess also comes from this assumption.
- Its heavy dependence upon the corresponding initial guess restricts the possible values of `n` to 2 and numbers which are divisible by 10. Note that this is just a consequence of the current implementation, not an absolute limitation.

Newton-Raphson Method >

Method Overview

The Newton-Raphson method is an iterative methodology for finding the roots of real functions, defined by *x_i+1 = x_i - f(x_i) / f'(x_i)* and starting from an initial guess *x0*. In the current implementation (i.e., `GetNRoot`), the goal is solving the function *f(x) = x^n - value*, what yields:

f(x) = x^n - value
*f'(x) = n*x^(n-1)*
*x_i+1 = ((n - 1) * x_i + value / x_i^(n-1)) / n*

`GetNRoot` includes a `Number`-based version of the aforementioned equation inside a loop exited when the difference between the *x_i+1* and *x_i* values is smaller or equal than the target accuracy of `1e-28m`. Note that this is assumed to be the smallest positive value with which the most precise type (i.e., `decimal`) can reliably deal.

This approach has only one relevant limitation: the initial guess *x0* has to be similar enough to the final result. A bad initial guess would provoke (practically speaking) infinite loops in quite a few scenarios. The methodology with which I came up to address this issue is undoubtedly the most important part of the current implementation. The following two sections (Exponential Proportionality and Method Improvement) include detailed explanations about it, but some points should already be clear:

- The calculations in `GetNRoot` are always started from an acceptably good initial guess (i.e., a right solution for the target accuracy will always be found quickly).
- Under very specific conditions (and, presumably, only when being called from `Math2.SqrtDecimal`), the calculations might have to be exited before reaching convergence in order to avoid an infinite loop. The accuracy in these cases (`5e-28m`) is very similar to the usual one.

Exponential Proportionality

The first thing to highlight is that I came up with the ideas in this and the next section completely by my own; I saw certain patterns while performing some tests meant to improve the fractional exponentiation algorithm. I didn't do any research on this front and am not aware about any theory on these lines.

With "exponential proportionality", I refer to the common trends underlying all the results generated by the power of certain real number (e.g., 2^3.2, 5.5^3.2 and 100000^3.2 being somehow related). Logically, these trends will never have a linear behaviour and, technically speaking, these values aren't proportional; on the other hand, "proportionality" seems to intuitively provide a very clear picture about this behaviour.

Validating the aforementioned ideas with NumberParser is quite straightforward. For instance, consider the following C# code:

```
decimal exponent = 3m;
NumberD res = Math2.ApplyPolynomialFit
(
   Math2.GetPolynomialFit
   (
      new NumberD[] { 3m, 4m, 6m, 7m },
      new NumberD[]
      {
         Math2.PowDecimal(3m, exponent), Math2.PowDecimal(4m, exponent),
         Math2.PowDecimal(6m, exponent), Math2.PowDecimal(7m, exponent)
      }
   ),
   5m
);
Number res2 = Math2.PowDecimal(5m, exponent);
NumberD diff = Math2.Abs(res - (NumberD)res2);
```

This code calculates certain power (3) of a given value (5) by analysing (second degree polynomial fit) the way in which the surrounding values (3, 4, 6 and 7) behave. This specific calculation is very accurate (i.e., `diff` is virtually zero), what isn't the case in quite a few other scenarios. In fact, the aforementioned implementation only works acceptably well with small values and exponents. Nevertheless, the restricted applicability of this implementation is exclusively provoked by the simplistic trend-finding methodology and doesn't affect the validity of the proposed ideas.

In any case and even by assuming that reliable trends could easily be found for any possible scenario, the resulting outputs would be unacceptably inaccurate. Bear in mind that the fastest exponentiation approach systematically delivering errors (0.1% or 0.0001%) wouldn't be acceptable; much less here, where accuracy is the top priority. But there is a situation which can be benefitted from these not-too-accurate results: the important determination of the initial guess in the Newton-Raphson method.

That initial guess expects a good enough estimate for the n-root calculation (i.e., the inverse of power, which also shows the described behaviour) of any positive number. In principle, these requirements seem quite far away from the aforementioned ideal conditions, but what if the number of potential n values could be highly reduced? In that case, wouldn't it be possible to create a limited number of trends to account for all the input scenarios? The answers to these and similar questions can be found in the next section.

Method Improvement

The intended fractional exponentiation approach is expected to account for most of the significant digits of the `decimal` type. Thus, the corresponding algorithm has to be able to deal with many scenarios involving fractions formed by big numbers (e.g., 0.23456789885 defined by the fraction 23456789885/10^11), what implies that the n-root calculating approach (i.e., Newton-Raphson) has to be able to deal with a huge range of n values (i.e., many integers within the 1-10^28 range).

The complexity of the aforementioned fraction can be reduced by analysing its constituent elements separately: the numerators are used in integer exponentiation (quick and reliable implementation easily dealing with any number) and the denominators in Newton-Raphson (reliability conditioned by the initial guess). Additionally, the ideas in the previous section seem to indicate that somehow reducing the number of n values/denominators (or inter-relating them) could be helpful to find good initial guesses. That's why it soon became clear that the best option was creating multiple-of-10-based fractions.

Even under the aforementioned conditions, the number of different input scenarios might still be too big (i.e., at least, 28 different trends); but the multiple-of-10 reliance proved to also be helpful on this front, via noticeable similarities across different n values (e.g., 10, 1000 or 10^10). I was able to come up with relatively simple approaches delivering acceptably good guesses for all the possible n values.

Math2_Private_New_PowSqrt_RootIni.cs includes all the code calculating the initial guesses for `GetNRoot`. These algorithms describe the patterns which I saw after analysing a reasonably big number of different input conditions; for example, after writing to a file the roots for 10, 10^2, 10^3, etc. with n = 10, 100, 1000, etc., a simple visual inspection was enough to extract worthy conclusions. All this information is referred by one of the following two main methods:

- `GetBase10IniGuess` deals with all the cases where n is a multiple of 10, what includes all the n-root calculations performed by the fractional exponentiation algorithm (i.e., using `Math2.PowDecimal` with non-integer exponents). By bearing in mind that it is the first version of an innovative approach, this code delivers what is expected (i.e., good enough initial guesses for any possible scenario). Although all this part does seem improvable, I have no short-term plans to optimise it. On the other hand, I will submit a CoreFX issue suggesting to add `decimal` overloads to the `System.Math` methods `Pow`/`Sqrt` (note that a previous similar issue was reviewed and rejected, although not completely); if the .NET team decides to go ahead with that new suggestion, I would certainly work on optimising the current implementation.
- `GetBase2IniGuess` deals with the specific scenario where n equals 2 (i.e., when using `Math2.SqrtDecimal`).

In summary, the described implementation allows the Newton-Raphson method to deal with virtually any input scenario (i.e., any value within the `Number` range and any exponent within the `decimal` range, by bearing in mind the aforementioned 25-first-decimal-digits limitation), what implies that a valid result for the target accuracy (i.e., `1e-28m` or, under very specific conditions, `5e-28m`) will always be found.