Large Numbers Library

Assets by Votrubec

Welcome

Hello and welcome.

Let me start off by saying thank you for your purchase, and I hope you find the asset fulfils your coding needs.

The most common implementation for large numbers is "clicker" or "idle" style games. You know the ones. They're fun, but they're also deceptively technical. Most developer's initial instinct is that "the computer just does it". Unfortunately, in this instance, the computer doesn't just do it[1]. Due to the fantastically huge nature of the numbers described, there's no computer representation for them. As such, this library is intended to overcome the "large number" hurdle.

[1] The C# struct BigIntegers can theoretically store values of any size, memory permitting, but we're still left without a number representation and naming system.

This library contains structures for Large Numbers, Scientific Notation numbers, Alphabetic Notation numbers and StringBuilder extension class with number converters producing almost no extra Garbage[2].

There is some "smoke and mirrors" used here. Firstly, if the numbers are so far different in scale (magnitude), the operations (e.g. add and subtract) aren't even performed. Secondly, similar to floating point numbers, the structures in this library contain two component parts to store the value. And just like floating point numbers, floating point errors occur[3]. While numbers of exact or close magnitudes won't produce floating point drift, working with values of vastly different magnitudes can introduce small drift errors. Reversing the operations can also lead to different end results. For example, adding a value to another a million times, then subtracting the value a million times could lead to minute differences in value. For the most part, the error is SO small it won't even be humanly noticeable and will not affect the game's outcome. I only mention this for the sake of complete transparency.

For each of the large number types, these operators have been overloaded:

+, -, /, *, <, <=, >, >=, ==, !=

Using these operators gives you the ability to use the large numbers without having to think of the underlying mechanics. In addition, the Equals(Object) and GetHashCode( ) methods were overridden. For Equality, I chose to use 3 decimal places and the magnitude to determine if two large numbers were equal. Just as with all floating point values, it's usually never a good idea to check for equality. Instead, comparing values is normally the better option.

I had intended to make the structures for large numbers immutable. There are some benefits and a slight speed bump in readonly structs. But the downside is that the structures then couldn't be serialised through the Unity Inspector. I didn't need a crystal ball to hear the howls of complaint that would have ensued if these values couldn't be modified and saved through the inspector. And to be honest, it does make it more handy to modify the values through the Inspector.

[2] At the time of this writing, the StringBuilder extensions produced NO extra Garbage other than the resulting string.
[3] See the "Precision and Accuracy in Floating-Point Calculations" here.

Installation

The Large Numbers Library requires you to set your “Api Compatibility Level” to either “.NET 4.x” or “.NET Standard 2.0” (these might be named slightly different in older Unity versions). You can find this under the Edit menu > Project Settings > Player section. This pertains mostly to older Unity versions, as the newer Unity versions do not allow the older Api options anymore.

A simple test of the Large Number Library can be done by opening the LargeNumebersTest scene. The scene contains a GameObject that has the LargeNumberTestBehaviour attached. When the game runs, the component will output, via Debug.Log calls to the Console, a series of results testing some of the library features. It will also update the TextMeshPro text items demonstrating compound operators working on both deserialised LargeNumber and ScientificNotation items. Because of the overloaded operators, the structures can be used in a similar fashion to standard value types.

Other than that, the library doesn’t use any Unity specific libraries, and can safely be used in any C# project outside of Unity. So, that’s a bonus of sorts… but why would you want to program for anything else but Unity?

Each of these following Number classes has a coefficient and magnitude. The `ToString()` method has been overridden to provide basic representations as well. But it's likely you as the developer will want to lay out the coefficient and magnitude values separately, based on your UI design. It's also worthwhile mentioning that although all the number classes use the same terminology, the "magnitude" in each of the classes isn't exactly the same every time.

Large Numbers

The Large Numbers structure is based on the John Horton Conway and Richard K. Guy system[4]. If you're familiar with the Conway-Guy system, you'll be aware that it starts with a Base of 1, which represents Millions. For simplicity, I have taken the liberty of adjusting the Base up by one. This now accounts for Thousands as well. The minor tweak vastly simplifies the maths involved and seamlessly allows for not only addition of large numbers, but subtraction, multiplication, division as well as the reciprocal values as well. So, with this Large Number structure, the possible values are ±999.999 NoveNonagintaNongentillion (999.999×10^{3000}) as well as ±999.999 NoveNonagintaNongentillionth ($999.999 \times 10^{-3000}$).

`GetLargeNumberName(int base)`
Get the Large Number Name by magnitude. The base parameter is synonymous with magnitude.

Scientific Notation Numbers

In addition to the Latin-based Large Numbers, the library also includes a standard Scientific Notation[5] representation of values. As with the above Large Numbers structure, the Scientific Notation structure allows for ridiculously large values; $\pm 9.999 \times 10^{2147483648}$ as well as $\pm 9.999 \times 10^{-2147483648}$. In technical terms, the magnitude is the Min and Max of a 32 bit signed integer.

[4] See the Wikipedia page for more information on Large Number Naming.
[5] See the Wikipedia page for more information on Scientific Notation.

Alphabetic Notation Numbers

An alphabetic notation numbering system is available that is similar to other popular idle games. This system counts uses K, M, B and T (thousands, millions, billions and trillions) before starting with lower case alphabetic characters. The first is ‘a’ through to ‘z’ after which a second character is added that looks like ‘aa’, then ‘ab’ etc. The minimum and maximum alphabetic magnitude is a strange value – ‘fxshrxs’- which is very loosely Max of a 32-bit signed integer converted to base26. So large extent range is in the vicinity of ±999 fxshrxs while the fraction extents are all the way down to ±999 fxshrxsth.

```
GetAlphabeticMagnitudeName(int magnitude)
```
Get the Alphabetic Magnitude Name by magnitude.

```
GetAlphabeticNotationFromString (string alphabeticNotationString, var
AlphabeticNotation alphabeticNotation)
```
Get an AlphabeticNotation item from a string representation. NOTE : the parsing of the numeric part use EN-US as the culture info.

```
GetMagnitudeFromAlphabeticName (string alphabeticName, out var magnitude)
```
Get the integer magnitude from an Alphabetic Name.

StringBuilderExtensions

I've extended StringBuilder with three extension methods;

```
ConvertAndAppend(int value)
```

This will convert any given integer, and append it to the current StringBuilder object.

```
ConvertAndAppend(double value, int decimalPlaces = 3, bool padDecimalWithZeros = true)
```

This method will convert a double value to a string representation and append it to the current StringBuilder object.

```
ConvertAndAppendTruncated(double value, int totalNumerals = 3, bool forcePaddedDecimal = false)
```

This method will create a string number, that only has the desired number of numerals, excluding the decimal or negative sign if present. forcePaddedDecimal, if set to true, will always include a decimal part, even if the given value didn't warrant one.

Each of these methods endeavour to produce as little garbage as possible. Because idle/clicker games tend to display lots of values all over the screen (the more numbers flashing on the screen, the more addicted I become!), the game runs the risk of producing a lot of string garbage. An unverified statement suggested that about 70% of garbage in .Net applications was string objects. With that in mind, care should be given when creating new string instances, hence why I included these StringBuilder extensions.

Versions

1.0 [27-11-2019]
Initial publication. Includes both Conway-Guy and Scientific Notation structures.

1.1 [02-01-2020]
[Updated] The demo scene code has been reformatted to be even clearer about what's being demonstrated. [Fix] A small code issue was identified that prevented LargeNumber to be properly serialsed. [Fix] A few spelling errors in the comments were fixed.

1.2 [14-04-2020]
[Fix] Multiplication and Division of ScientificNotation numbers by doubles was returning Conway & Guy LargeNumbers instead of ScientificNotation numbers. This has been fixed. Thanks jlwauchope.

1.3 [05-05-2020]
[NEW] Added the Alphabetic Notation numbers.

1.7 [24-07-2020]
[NEW] Two new methods for AlphabeticNotation have been added. GetMagnitudeFromAlphabeticName and GetAlphabeticNotationFromString. These allow string representations to be used in creating new AlphabeticNotation items. Useful for debugging and serialising to/from human readable.
[UPDATED] Removed some hard-coded Standard Names lengths through code. This allows the developer to more easily specify how many Standard Names to use (i.e. 'K', 'M', 'B', 'T').
[FIX] Triple (and higher) AlphabeticNotation names would drop the most significant magnitude when reaching the end of the alphabet array. This has been resolved.

1.8 [27-07-2020]
[FIX] Comparisons of numbers with vastly different magnitudes would result in an IndexOutOfRangeException. A bounds check has been implemented and this has been fixed across all number types.