Tetration and nth-term iterative operators

Benjamin Rose
brose@stevens.edu

Stevens Institute of Technology
Professor Pavel Dubovskiy
MA – 134
Discrete Mathematics
5 April, 2008

[Introduction]

Pre-collegiate teaching dictates that there are a finite, simple set of mathematical operators. These operators are simple to understand and perform on any number or set of numbers. However, more complex numerical operators exist. In fact, an infinite number of operators exist. In order, they are known as: addition, multiplication, exponentiation, tetration, and so on. Each operator is a recursive call of the operator below it. These terms iterate themselves into infinity, each operator *n* being the repetition of the *n-1* term.

These operators cause number growth at a gargantuan rate. It is extremely difficult to write programs to represent these operators, given how quickly f(x) increases where f(x) = ${}^{4}x$ (the tetration of x). The results of such operations are very interesting number patterns, which generate unique fractals and graph patterns.

This document provides a simple C++ hyper-operator function and an arbitrary-length integer class in the hopes that further calculations can be performed on more powerful computers in the future. It illustrates the basics of hyper-operators, and demonstrates the beauty of hyper and non-fundamental operators. It also makes conjectures about operators that are proven to exist through inductive reasoning.

[A brief history of tetration]

The word “tetration” was coined by English mathematician Reuben Louis Goodstein from tetra- (four) and iteration. Tetration is the fourth operator in the set of n^{th}-term iterators, hence the use of the word “four”. Tetration was originally used as a way to express very large numbers, similar to scientific notation. Over the years, mathematicians began to realize the significance of tetration as more than just a symbolic notation. Tetration was the first step towards realizing there were an infinite amount of operations. A pattern became evident – each known operator is a recursive call of the previous operator. For example:

Addition as an operation is simply adding 1 to the original value *b* times.

$$a + b = a + \underbrace{1 + 1 + \cdots + 1}_{b}$$

Multiplication is repeated addition.

$$a \times b = \underbrace{a + a + \cdots + a}_{b}$$

Exponentiation is repeated multiplication.

$$a^b = \underbrace{a \times a \times \cdots \times a}_{b}$$

Tetration is repeated exponentiation.

$${}^{b}a = \underbrace{a^{a^{\cdot^{\cdot^{a}}}}}_{b}$$

And so on into infinitum. This recursion principle of operators became more widely known with the dawn of the computing age. Computers can do two things with a set of numbers – they can add them, and they can compare them. All other operations performed on a computer eventually boil down to these two actions. As such, in order to perform basic calculations, the operations would come down to simple addition. A simple example of how this works at the low level can be seen in appendix A.

[Notation of tetration]

Every operator has a symbol that distinguishes it as a set of rules that the user must follow in order to achieve a correct and universally-standard answer. These symbols are commonly known: addition is +, multiplication can be either *, ·, ×, or denoted with parenthesis, and exponentiation can be expressed as a^b or with a carrot (^). Tetration, when first conceived, used the inverse notation as exponentiation, that is, it was denoted as ${}^{b}a$. It was not long before mathematicians realized it was foolish to create a separate symbol for each and every operator, since there are an infinite number. Several different but useful notations arose. Each notation brought to the table it's own strengths and weaknesses in representing operators into infinity. There are several notations that allow for infinite expansion.

One such notation is known as Conway chained-arrow notation. This notation uses right-arrows to distinguish the base from the operand and the iteration-level. Consider the example $a \rightarrow b \rightarrow 2$, where a is the base, b is the operand, and 2 is the level of iteration. A level of 2 implies tetration, the fourth absolute iterative level operator. Expansion into further terms is done simply by changing the last number.

Another such notation is called iterated exponential notation. It is written in the form $\exp_a^b(1)$, and can be expanded for infinite terms by changing the value in the parenthesis. This form is important, because it most easily accommodates variables and abstract ideas as a term for iterative operations. As such, this form will be used later in this document to discuss a the ∞^{th} operator. It is also extremely important to notice that when the constant 1 is provided to the function as the iteration of the operator in this form, it is implied that the function will return the tetration (4th iteration) of the values.

The final widely-used form is known as Knuth's up-arrow notation. This is the simplest notation, in the form $a \uparrow\uparrow b$, which provides tetration. This form can also be represented in ASCII code using a double-carrot (^^). While not usually used to express any iterations except tetration, the form can accommodate more by adding more arrows. For example, $a \uparrow\uparrow\uparrow b$ can be used to represent the 5th iterative operation. Clearly, this form can be cumbersome to use, especially beyond the 6th or 7th term, where it becomes too impractical to count the arrows, however, it is quite easy to use, and has somewhat become the de-facto standard to represent tetration. As such, this form will be used throughout this document when referring to tetration.

[Implementing code for tetration]

The basic goal presented in this document is to measure how the values between certain nth-term operators grow and distance themselves from other operators. In order to perform such calculations, a computer with specially-written software is needed. To perform such calculations manually would take an extremely long time, and would be prone to mistakes that could throw off every subsequent calculation. As such, a C++ program was written that can make these large calculations effortlessly. However, the standard C++ libraries do not support data-types that can extend to calculate such enormous values at such high precision. Because of this, a C++ header file, ALInteger, was written and implemented, representing an Arbitrary-Length Integer. This data type can handle any size number, and at great precision. The subsequent calculations have all been performed using this software, which have then been checked for accuracy.

[Measuring the time complexity of the hyper-operator code]

The hyper-operator, which is declared as part of the C++ Math namespace, uses the principle of recursion to make it's calculations. When a hyper-operator other than 1 is called, the program enters a for-loop. This for-loop executes *n* times, each time calling the hyper function *b* times. The full extent of the time complexity is unknown. Time was spent among professors at Stevens Institute of Technology trying to analyze the time-complexity of the hyper-function, but due to the combination of the recursion, the for-loop, and the ever-increasing and ever-dependent *r* variable, no simple function could be constructed to model the number of physical additions the computer would have to perform.

[Some Sample Values from tetration]

	1	**2**	**3**	**4**	**5**
1	1	1	1	1	1
2	2	2	16	65536	
3	3	27	7625597484987	SYSTEM OVERFLOW	SYSTEM OVERFLOW
4	4	256	13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649006084096	SYSTEM OVERFLOW	SYSTEM OVERFLOW

Entries marked with "SYSTEM OVERFLOW" had segmentation faults due to the capacity of the hardware the computations were performed on. It is interesting to note the huge growth in the numbers between cells. Even run on machines with several gigabytes of RAM, the calculations caused an overflow all of the memory the kernel had allocated to the program. Given a cluster testbed, more advanced calculations could be performed.

[Building tetration fractals]

Tetration and nth-term operators provide very interesting fractals when they are constructed as a member of a function. Figure 1 is a fractal of tetration showing the points in the complex plane that are periodic under iteration. The red kidney shaped area has a period of one and is the area of convergence of the tetration function. The small yellow circle in the depression to the left of the red area is period two and contains zero. The large green area is period three. This fractal is similar to the Mandelbrot fractal, the main difference being that it is built upon an exponential map instead of a quadratic map. The fractal has a domain from -4.5 to 3.5 of the real axis and a range from $-3i$ to $3i$ on the imaginary axis.

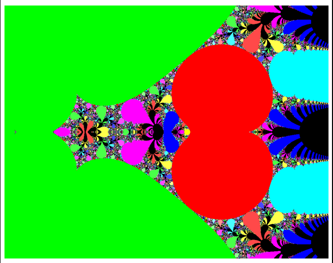
Figure 1: Tetration by Period

Figure two is the same function, but is instead built showing the points in the complex plane that escape to infinity under recursion. It is constructed using the same axises as figure 1. It is also colored differently than figure 1, but it is quite easy to see where patterns exist between figures 1 and 2. The large area on the left, and the kidney-shaped sections exist in both figures. It is also easy to see the right-most striped sections in both fractals. It is clear, by way of these pictures, that hyper-operators follow a definite set rules regarding numbers, and, although those numbers are large, they do relate to one-another on a grand scale.

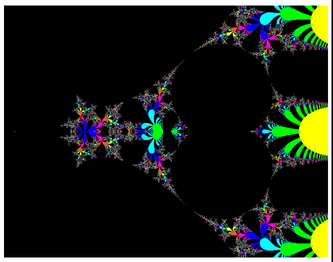
Figure 2: Tetration by escape

[Constructing hyper-operators in an infinite number-system]

Hyper-operators do not necessarily have to be constructed in a finite system. Normally, a magnitude is specified, such as in the case *a*+*b*, where we would supply both *a* and *b*, the base and the number of 1's to add to the base. In the event of providing infinity as *b*, we yield: $a + \infty = \infty$, a simple to understand equality. Now consider tetration to infinite heights. Basically, it is the principle of taking a base, and raising it to itself an infinite number of times. It would be expected that this would yield either ∞ or -∞ in all cases, however this is erroneous. Tetration to infinite heights, while yielding ∞ or -∞ in most cases, can yield any number. Consider the following case, which is a truncated version of the tetration of $\sqrt{2}$ to infinite heights:

$$\sqrt{2}^{\sqrt{2}^{\sqrt{2}^{\sqrt{2}^{\sqrt{2}^{1.41}}}}} = \sqrt{2}^{\sqrt{2}^{\sqrt{2}^{\sqrt{2}^{1.63}}}} = \sqrt{2}^{\sqrt{2}^{\sqrt{2}^{1.76}}} = \sqrt{2}^{\sqrt{2}^{1.84}} = \sqrt{2}^{1.89} = 1.93$$

It is clear that this equation is approaching a value. This value, oddly enough, is 2, the base value inside the square-root. To express this equation in proper notation for tetration, it can be written that $\lim_{x\to\infty} \exp^x_{\sqrt{2}}(1) = 2$. This is not a rare occurrence, see *figure 3* for a pictorial representation of several other values subjected to infinite tetration. Tetration to infinite heights yields many other results, as already proven in *figure 2*, where the escape set was built for the tetration operation. In these such cases, we see a classic case of what Euler referred to as "mathematical beauty from disorder". While Euler was speaking of his identity relating operations and fundamental numbers, the concept remains. In this example, we have an irrational number, $\sqrt{2}$, and an imaginary conception of an operator, tetration to infinite heights. Out of two chaotic concepts, a sane and perfectly rational value is produced. This is a shining example of the hidden beauty in mathematics, and a clear case of patterns inherent in the fabric of the universe itself.

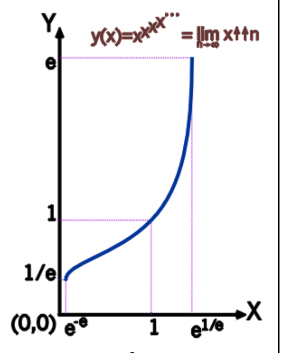

Figure 3: Infinite Tetration

It is clear that tetration to infinite heights yields real and sane values, but there is another aspect of nth-term operators that can be extended beyond the finite realm. Before, we investigated how hyper-operators are recursive, that is, that each nth-term is simply a repeated call of the (n-1)th-term. Since each term, *n*, utilizes the term (*n-1*), for *n* ≥ 2, the converse must be true. Mathematical Inductive reasoning can be used to prove there are an infinite number of operators. Since it is well known that addition exists, and this is the first hyper-operator, the basis step is completed and valid. Every operator *n* has an operator *(n+1)* that uses recursion of *n* to yield results. This is proven by the simple fact that multiplication exists as repeated addition, exponentiation as repeated multiplication, and tetration as repeated exponentiation. For each *n : n* ≥ 2, there exists both a valid *n+1* term, and *n-1* term. Therefore, there are an infinite amount of mathematical operators, each a recursive call of the lower hyper-operation.

Following this logic, there is, therefore, an imaginary ∞^{th} operator. This operator is very difficult to conceive. It is an operator so absolutely massive that any input to the function instantly yields infinity, including even the operation's identity value. While this operator is thoroughly useless from a mathematical point of view, it is an interesting exception to all mathematical laws that is exciting to consider.

[Conclusion]

Tetration is one of the relatively unexplored realms of modern mathematics. It involves writing specialized software, running calculations on massive machines, and drawing advanced inferences from relatively basic concepts of mathematics and computer science. Please refer to the appendices of this document for some basic code to aid in the further pursuit of hyper-operators. This document barely scratches the surface of such a massive field of mathematics. It submits for further review the time complexity of performing such calculations on a computer system and the relationships between obscure and irrational numbers and ideas.

[Appendix A]

Code for hyper operators, including an arbitrary-length integer header file.

Math.h

```
#ifndef MATH_H
#define MATH_H
#include <iostream>
typedef ALInteger u_long;
typedef unsigned char u_char;
using namespace Math ;
{
	u_long hyper(u_long a, u_char n, u_long b)
	{
		if (n == 0)
		{return a;}
		u_long r = a;
		for (u_long i=1; i<b; ++i)
		{r = hyper(a, n-1, r);}
		return r;
	}
}
#endif
```

ALInteger.cpp

```
#include "ALInteger.h"
const ALInteger ALInteger::ONE = ALInteger(1);
const ALInteger ALInteger::ZERO = ALInteger(0);
```

ALInteger.h

```
#ifndef	ALINTEGER_H
#define ALINTEGER_H
#include <iostream>
#define SIG_BYTES 1024
typedef unsigned long long u_long;
typedef unsigned char u_char;
inline u_char setBitOnByte(u_char c, u_char b, bool v)
{
	return v? (c | ((u_char)(1) << b)): (c & (255 - ((u_char)(1) << b)));
}
inline bool getBitOnByte(u_char c, u_char b)
{
	return c == (c | ((u_char)(1) << b));
}
class ALInteger
{
	private:
		u_char x[SIG_BYTES];
	public:
		static const ALInteger ONE, ZERO;
		ALInteger()
		{
			for (int i=0; i<SIG_BYTES; ++i)
			{x[i] = 0;}
		}
		ALInteger(u_long a)
		{
			for (int i=0; i<SIG_BYTES; ++i)
			{x[i] = 0;}
			for (int i=0; i<sizeof(u_long); ++i)
			{
				for (int j=0; j<8; ++j)
				{
					int bit = i*8 + j;
					u_long temp = (u_long)(1) << bit;
					if ((a | temp) == a)
					{x[i] |= 1 << j;}
				}
		}
	}
	ALInteger(const ALInteger &a)
	{
		for (int i=0; i<SIG_BYTES; ++i)
		{x[i] = a.x[i];}
	}
	~ALInteger() {}
	char operator [] (int i) const
	{return x[i];}
	ALInteger &operator = (const ALInteger &al)
	{
		for (int i=0; i<SIG_BYTES; ++i)
		{x[i] = al.x[i];}
		return *this;
	}
	void setBit(u_char byte, u_char bit, bool v)
	{x[byte] = setBitOnByte(x[byte], bit, v);}
	void setBit(u_char b, bool v)
	{setBit(b/8, b%8, v);}
	bool getBit(u_char byte, u_char bit) const
	{return getBitOnByte(x[byte], bit);}
	bool getBit(int b) const
	{return getBit(b/8, b%8);}
	ALInteger operator ~() const
	{
		ALInteger r;
		for (int i=0; i<SIG_BYTES; ++i)
		{r.x[i] = ~x[i];}
		return r;
	}
	bool operator == (const ALInteger &al) const
	{
		for (int i=0; i<SIG_BYTES; ++i)
		{
			if (x[i] != al.x[i])
			{return false;}
		}
		return true;
	}
	bool operator != (const ALInteger &al) const
	{
		for (int i=0; i<SIG_BYTES; ++i)
		{
			if (x[i] != al.x[i])
			{return true;}
		}
		return false;
	}
	bool operator < (const ALInteger &al) const
	{
		for (int i=SIG_BYTES-1; i>=0; --i)
		{
			if (x[i] != 0 || al.x[i] != 0)
			{return x[i] < al.x[i];}
		}
		return false;
	}
	bool operator <= (const ALInteger &al) const
	{
		return (*this) < al || (*this) == al;
	}
	bool operator > (const ALInteger &al) const
	{
		for (int i=SIG_BYTES-1; i>=0; --i)
		{
			if (x[i] != 0 || al.x[i] != 0)
			{return x[i] > al.x[i];}
		}
		return false;
	}
	bool operator >= (const ALInteger &al) const
	{return (*this) > al || (*this) == al;}
```

```
        ALInteger operator << (int b) const
        {
                ALInteger r = 0;
                for (int i=SIG_BYTES-1; i>=0; --i)
                {
                        for (int j=7; j>=0; --j)
                        {
                                int bit = i*8 + j;
                                bool v = false;
                                if (bit >= b)
                                {v = getBit(bit - b);}
                                r.setBit(bit, v);
                        }
                }
                return r;
        }
        ALInteger operator + (const ALInteger &al) const
        {
                ALInteger r;
                bool carry = false;
                for (int i=0; i<SIG_BYTES; ++i)
                {
                        for (int j=0; j<8; ++j)
                        {
                                u_char t = 1 << j;
                                bool a = (x[i] | t) == x[i];
                                bool b = (al.x[i] | t) == al.x[i];
                                if (a && b && carry)
                                {
                                        r.x[i] |= t;
                                        carry = true;
                                }
                                else if ((a && carry) || (b && carry) || (a && b))
                                {carry = true;}
                                else if (carry || a || b)
                                {
                                        r.x[i] |= t;
                                        carry = false;
                                }
                                else
                                {carry = false;}
                        }
                }
                return r;
        }
        ALInteger &operator += (const ALInteger &al)
        {return ((*this) = ((*this) + al));}
        ALInteger &operator ++ ()
        {return ((*this) = (*this) + ONE);}
        ALInteger operator - (const ALInteger &a) const
        {
                ALInteger r = (*this) + (~a) + ONE;
                r.x[SIG_BYTES-1] &= 127;
                return r;
        }
        ALInteger &operator -= (const ALInteger &al)
        {return ((*this) = (*this) - al);}
        ALInteger &operator -- ()
        {return ((*this) = (*this) - ONE);}
        ALInteger operator * (const ALInteger &a) const
        {
                ALInteger r = ZERO;
                for (ALInteger i=ZERO; i<a; ++i)
                {r += (*this);}
                return r;
        }
        ALInteger &operator *= (const ALInteger &a)
        {return ((*this) = (*this) * a);}
        ALInteger operator / (const ALInteger &d) const
        {
                ALInteger t = (*this);
                ALInteger r = ZERO;
                for (; t >= d; r=r+ONE, t-=d);
```

```
                                        return r;
                                }
                                ALInteger &operator /= (const ALInteger &a)
                                {return ((*this) = (*this) / a);}
                                ALInteger operator % (const ALInteger &a) const
                                {return (*this) - a*((*this) / a);}
                                ALInteger &operator %= (const ALInteger &a)
                                {return ((*this) = (*this) % a);}
                                template <class Type>
                                operator Type() const
                                {
                                        Type r = 0;
                                        for (int i=0; i<sizeof(Type); ++i)
                                        {
                                                for (int j=0; j<8; ++j)
                                                {
                                                        if ((x[i] | ((u_char)(1) << j)) == x[i])
                                                        {r |= (Type)(1) << (i*8 + j);}
                                                }
                                        }
                                        return r;
                                }
};
inline std::ostream &operator << (std::ostream &os, const ALInteger &al)
{
        ALInteger a = al;
        char s[2467];
        int i = 0;
  do
  {
                                static const ALInteger TEN = 10;
    s[i++] = (char)(a % TEN) + '0';
                                a /= TEN;
  }
  while (a != ALInteger::ZERO);
  --i;
  for (;i >= 0; --i)
  {os << s[i];}
  return os;
}
#endif
```

[References]

- tetration.org, by Daniel Geisler.
  - http://www.tetration.org/
- Extending hyper4 and Knuth's up-arrow notation to the reals, by I.N. Galidakis.
  - http://ioannis.virtualcomposer2000.com/math/papers/Extensions.pdf
- Extension of the hyper4 function to reals, by Robert Munafo.
  - http://home.earthlink.net/~mrob/pub/math/ln-notes1.html#real-hyper4
- Power Tower, by Galidakis, Ioannis and Weisstein, Eric W. [Mathematica].
  - http://mathworld.wolfram.com/PowerTower.html
- Tetration, by the wikipedia community.
  - http://en.wikipedia.org/wiki/Tetration
- Some Critical Points of the Hyperpower Function, by Joseph MacDonell.
  - http://www.faculty.fairfield.edu/jmac/ther/tower.htm
- The home of tetration, by Andrew Robbins.
  - http://tetration.itgo.com/