Numbers and Functions Using Java

An Object-oriented Approach

A crazy boy

Wonderful World

CONTENTS

CHAPTER 1

NUMBERS IN JAVA

Contents

In this chapter, we overview the floating number representation of real numbers and their arithmetic operations in Java. A good understanding how they work in Java is essential for numerical analysis using Java. We are not going into great details except hightlighting the key features, because there are quite a few good references that gives more rigorous and broader treatment on this subject, such as Higham's book [Higham:1996:AS] and Goldberg's article [Goldberg:WEC]. They are very informative, e.g., why a 2-based representation is better than a 16-based representation, or why a guard bit helps improving the accuracy.

1.1 Number Representation in Java

Java uses IEEE 754 standard to represent real numbers. *double* in Java is stored in the following binary format:

sign	exponent	fraction(significand, or mantissa)
1 bit	11 bits	52 bits
s	e	f

Table 1.1: Special Representations

exponent	significand	values
$e = 0$	$f = 0$	± 0
$e = 0$	$f \neq 0$	denormalized numbers $\pm 0.f \times 2^{-1022}$
$e = 2027$	$f = 0$	$\pm$infinity
$e = 2027$	$f \neq 0$	NaN

The sign field s is 0 for positive and 1 for negative.

The exponent field e is interpreted as unsiged integers, 0 to $2^{11} - 1 = 2047$, with 0 and 2047 reserved for special purposes. To represent negative integers power, we use a bias 1023 to shift the range to between -1022 to 1023.

The significand field f is interpreted as follows. Normally the representation of a number is not unique, e.g., 1.1101 can be expressed as 1.1101, or 0.11101×2, or 0.011101×2^2, just like we do in the decimal cases, such as 3.1415926 can be expressed in many ways, such as 3.1415926, 0.31415926×10, 0.031415926×10^2, etc. Among these different representations, there is only one representation such that the leading digit is nonzero and there is only one digit before the period. So this is the representation we are going to use. If the leading bit is nonzero(i.e., 1 in binary case), the representation is called *normalized* (Otherwise, it is *denormalized*, we will discuss this more later) and the number is evaluated as the following

$$(-1)^s \times 1.f \times 2^e$$

Since we know the leading digit already, we don't need to store it and thus gain an extra bit for the fraction. The leading bit is called the implied bit. **Precision** is defined as the number of bits in the significand(including the implied bit), so Java double precision is 53 bits.

The table 1.1 shows the corner cases for e and f, the second case shows that if $e = 0$, then the leading digit is zero, otherwise if $e \neq 0$, then the leading digit is 1. So by using the exponent, we can figure out the leading implied digit. Java Double and Float classes have methods to convert the decimal numbers to hex decimal string so that we could inspect the bits in the storage, for example

```
1 System.out.println(Double.toHexString(0.9));
```

The result is $0x1.cccccccccccdp-1$, in the hexdecimal format.

The denormalized numbers are used for gracefully flushing to zero. They fill in the gap between zero and smallest normalized positive number. However, when we run into this range, we are losing accuracy, for example

```
1 x = 2.999999999999999E-308;
2 y = x / 30000 * 30000;
3 y = 2.999999999995396E-308
```

x is initialized to very close to the smallest positive normalized number such that $x/30000$ is a denormalized number. y should be the same as x, but apparently not.

Table 1.2: Range of the double

sign	+, -
exponent	1 ... 2046, with bias 1023
significand	[1, 1.9999999999999998]
largest positive	1.79769313486231570e+308
	0x1.fffffffffffp1023
	Double.MAX_VALUE
smallest positive	4.94065645841246544e-324
	0x0.0000000000001p-1022
	Double.MIN_VALUE
smallest positive normalized	2.2250738585072014E-308
	0x1.0p-1022
largest denormalized	2.225073858507201E-308
	0x0.fffffffffffp-1022

The range of the real numbers we can represent are listed in Table 1.2. In java, it is a compiler error when specifying a number larger than the Double.MAX_VALUE, e.g., 1.0e+310, or smaller than Double.MIN_VALUE. JDK documentation for the toHexString(double) method in the Double class has more information.

There is a *C* routine, machar, written by Cody, that can explore some of the characteristics of a floating number model.

1.2 Errors

Since the real numbers are continuous while the binary representation is discrete, we can not express every real number in the binary representation, even if they are well in the range. For example,

1.666666666666666666

is stored as

1.6666666666666667

The difference is called the **rounding error**. IEEE 754 standard has 4 rounding mode, but Java is using the default, round to nearest only.

When a given number is represented in Java, the last bit of the significand can be inaccurate due to rounding(assuming the number is within the range, not underflow or overflow), this one-bit difference is magnified by the exponent. In other word, there is more gap toward the infinity, for example

```
1 Math.ulp(2.0d) = 4.440892098500626E-16
2 Math.ulp(20000000.0d) = 3.725290298461914E-9
```

The **ulp**, or unit in the last place, is defined as this one-bit difference, the gap distance between two adjacent floating representations. If $x = 1.d_1 d_2 ... d_p \times 2^e$, i.e., the precision is p bits, then

$$\text{ulp}(x) = 2^{-p} \times 2^e = 2^{e-p}$$

It follows that

$$\frac{\text{ulp}(x)}{|x|} = \frac{2^{e-p}}{2^e \times 1.f} \leq \frac{2^{e-p}}{2^e} = 2^{-p}$$

This is the difference ignoring the exponent.

The rounding error of the floating representation is bounded by $\frac{1}{2}ulp(x)$ if we choose the closest represenation, i.e.,

$$|\hat{x} - x| \leq \frac{1}{2}ulp(x)$$

Another way to state this is

$$\frac{|\hat{x} - x|}{|x|} \leq \frac{1}{2}\frac{ulp(x)}{|x|} \leq \frac{1}{2}2^{-p}$$

The last constant depends on only p, which is only related to the floating number model we are using. We define this constant, called **machine epsilion**, as follows

$$\epsilon = 2^{-p}$$

In java, $p = 52$ and thus $\epsilon = 2^{-52} = 2.220446049250313E-16$. On a side note, ϵ is the one bit change in the significand, and thus $\epsilon = \text{Math.ulp}(1.0d)$. Now the above becomes

$$\frac{|\hat{x} - x|}{|x|} \leq \frac{1}{2}\epsilon$$

Let's define two more terms. **Absolute error** is defined as

$$E_{abs}(\hat{x}) = |x - \hat{x}|$$

Relative error is defined as

$$E_{rel}(\hat{x}) = \frac{|x - \hat{x}|}{|x|}$$

Then the above can be rewritten as

$$E_{abs}(\hat{x}) \leq \frac{1}{2}ulp(x)$$

and

$$E_{rel}(\hat{x}) \leq \frac{1}{2}\epsilon$$

The relative formula can be rewritten as

$$\hat{x} = x(1 \pm E_{rel}(\hat{x}))$$

The relative error is more "uniform" than the absolute error, e.g.,

$$|1000 - 999.99| < 0.01 \text{ and } |0.02 - 0.01| < 0.01$$

have the same absolute error, however, the difference is very small in the first case and very big in the second case because the exponent field plays a role. Relative errors count on the difference in the significant.

Another way to express the relative error is the **significant digits**. This term is very useful when we discuss the cancellation errors. Given a number in the scientific format,

$$x = 3.205400 \times 10^7$$

the number of significant digits is 7. Now if we have

$$x = 1.23456789 \text{ and } y = 1.23455555$$

then

$$x - y = 0.00001234 = 1.234 \times 10^{-5}$$

this means that we are losing sigificant digits from 9 to 4 due to substraction.

Sometimes, we need to round to certain digits, e.g., we want to round to 2 decimal places when dealing with US dollars so that we deal with up to 1 cent. In Java, we could do the following

```
1 Math.round(x * 100) / 100.0d
```

here the rounding policy is to round to the closest, we could use other rounding policies, such as ceiling or floor (Do not forget to deal with corner cases, like NaN, the above code is just for illustration purpose).

1.3 Comparison

In general, it is not safe to check whether two floating numbers are equal due to rounding. Even if it is the same number, double and float types are not the same, e.g., $1.0d/3 \neq 1.0f/3$. The alternative is to check whether these two numbers are close enough. However, being close enough is not transitive as in the case of being equal, i.e., if $|x-y| < eps$ and $|y-z| < eps$, for a given bound eps, we can not deduct $|x-z| < eps$. To extend this further, it is not safe to use double as part of the hash related keys either. In case of we have to use double in a hash key, define a fix number for precision, such as 10^{-12}, for the safe range. But this will limit the usage and should be well documented.

There are a few special cases. First Java Double class has two static values POSITIVE_INFINITY and NEGATIVE_INFINITY. Though they are used for flaging overflows, they do follow conventional math when comparing.

Another corner case in Java is +0.0 and −0.0, they are equal but have different signs. One pitfall is the following implementation of absolute value in math:

```
1 public double abs(double d)
2 {
3     return d >= 0.0 ? d : -d;
4 }
```

Since -0.0 = +0.0, abs(-0.0) returns -0.0, the correct answer is 0.0. Similarly, the expression return d<=0.0 ? -d : d is also not right because for 0.0 it returns -0.0. The sign is important in some context. The correct way is

```
1 public double abs(double d)
2 {
3     if (d == 0.0) return 0.0;
4     else return d > 0.0 ? d : -d;
5 }
```

Another one is NaN, which means nondeterministic, for example, when zero divided by zero, the result is not able to be determined. When $x \to 0$, $\frac{x}{x^2} \to \infty$ but $\frac{x^2}{x} \to 0$. So depends on how "fast" the denominator and numerator go to zero, we end up different results. So NaN is not comparable at all, e.g., $5.0 > NaN$ and $5.0 \leq NaN$ are both false and thus we lose exclusiveness. The following code doesn't catch the NaN case:

```
1 if (rate < 0.0)
2     throw new IllegalArgumentException("rate is negative");
3 interestPayment = principal * rate;
```

If we want to catch the NaN case, we should either add the explicit check for NaN or reverse the logic

```
1 if (rate >= 0.0)
2         interestPayment = principal * rate;
3 else
4    throw new IllegalArgumentException("rate is negative");
```

1.4 Arithmetic Operations

Java arithmetic operations follow IEEE 754 standard, if we denote the arithmetic operations $+, -, \times, \div$ by $\maltese$, then

$$fl(x \maltese y) = (x \maltese y)(1+\delta), \ |\delta| \leq \mu$$

where $fl(x)$ is the floating representation of x in the previous section and μ is the unit roundoff, $2^{-53} \approx 1.1102230246251565E-16$, which is $\frac{1}{2}\epsilon$, half of the machine epsilon.

The common problems with the arithmetic operations are divided by zero, underflow and overflow. Java throws an ArithmeticException when dividing by zero. It flags the overflow by setting the result to

Double.POSITIVE_INFINITY or Double.NEGATIVE_INFINITY

however it does not stop the process. So users have to check this value for overflow. Java does not flag underflow at all.

Another problem is losing accuracy. The first possible way is to operate on numbers that are too far apart, for example, $10^{10}+10^{-10}$. The second way is the subtraction cancellation. When two very close numbers are subtracted from each other, significant digits are lost and consequently errors are magnified by the exponent. For example, $1-0.999=0.001=1.0\times10^{3}$ If the last digit of 0.999 is not accurate, then the result is not accurate at all. The third way is the error scaling, either multiply by a relatively very large number or divide by a relatively very small number, the consequence is that the error is magnified. The forth way is error accumulation, when we have a series of calculations and each step depends on the result from previous step, for example, when calculating sums or averages.

1.5 Functions

In the above section, when we analyze errors, we assume x and y are accurate and then calculate the upper bound of the error. This is called forward error. However, most of the time, x and y are also approximate values, i.e., we really have $\hat{x}$ and $\hat{y}$. Instead of computing $x+y$, we are actually computing $\hat{x}+\hat{y}$. Now the question is how far we are from real $x+y$.

Let us consider this situation in a more general context. Let $f(x)$ be a function of x(so arithmetic operations are a special case), if x is perturbed by a small Δx, the eror analysis can be carried out via Taylor expansion

$$\Delta y=f(x+\Delta x)-f(x)=f'(x)\Delta x+O\Big((\Delta x)^2\Big)$$

The relative error is

$$\frac{\Delta y}{y}=\frac{f'(x)}{f(x)}\Delta x+O\Big((\Delta x)^2\Big)=\Big(\frac{xf'(x)}{f(x)}\Big)\frac{\Delta x}{x}+O\Big((\Delta x)^2\Big)$$

The scaler

$$\kappa_f(x)=cond_f(x)=\Big|\frac{xf'(x)}{f(x)}\Big|$$

is defined as the **condition number** of the function $f(x)$. If $f(x)=0$, we define

$$\kappa_f(x)=cond_f(x)=|f'(x)|$$

It measures the sensitive of f at the point x, i.e., a small perturbation of x causes how much of the change of the function value. If $\kappa_f(x)$ is small enough so that any small perturbation in x results small changes in y, then we call f *numerically stable* at f. The perception about the $\kappa_f(x)$ is that the number of correct digits in the calculated function value is roughly

$$p-log_{10}\Big(conf_f(x)\Big)$$

where $p=-log_{10}\Big|\frac{\Delta x}{x}\Big|$, about 16 for double and 9 for float. A special case is when x_0 is a roote of $f(x)$, i.e., $f(x_0)=0$ and x_0 and $f'(x_0)$ are nonzero, then near x_0, $\kappa_f(x)$ is

very large and tends to ∞ as x goes to x_0. So this small neighbourhood of x_0 is very sensitive to small perturbation.

In general, when we implement a function $f(x)$ with an algorithm $\hat{f}(x)$, we usually describe the accuracy of the algorithm as

$$\left|f(x) - \hat{f}(x)\right| < eps$$

for some small eps, such as 10^{-16}. However, this small absolute measure does not necessarily mean high accuracy. Take again the log gamma function as an example. The C99 implementation(and others) written by Sun Micro has very good accuracy except the small neighbourhoods around 1 and 2. In these two regions, C99 suffers the cancellation error first, then the error gets magnified further so that the function value gets shift by huge number of ulps.

When implementing functions, we have to adjust the required error bound criteria depending on the range of the function values. When the function value is large(> 1), the absolute error is more rigorous. When the function value is near zero, the relative error is more rigorous. However, most of the time, it is acceptable to use the following

$$\left|f(x) - \hat{f}(x)\right| < \text{Math.ulp}(\hat{f}(x))$$

This means that we should try to find the right "bucket" for the function value. Some of the elementary functions are implemented with more strict restriction:

$$\left|f(x) - \hat{f}(x)\right| < \frac{1}{2}\text{Math.ulp}(\hat{f}(x))$$

See the references for more information and reasons for doing this.

Two powerful tools to approximate functions are rational approximation and continuous fraction.Another approach to get high accuracy results is the arbitrary/-multi precision approach. It is slow but can be as accurate as pre-specified.

When we implement functions, we care two things: accurate and performant.

1.6 Bibliography

[1] Higham, N.J., *Accuracy and Stability of Numerical Algorithms*, SIAM, 1996

[2] Goldberg, David, *What Every Computer Scientist Should Know About Floating-point Arithmetic*

[3] Sun Micro, *Numerical Computation Guide*, http://docs.sun.com/source/806-3568/

CHAPTER 2

ERROR FUNCTION

Contents

In this chapter we are going to discuss several functions related to error function, namely, the error function itself, the complementary error function, and the inverse of the errof function.

2.1 Error Function

The error function is defined as the following integral

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

where x belongs to $(-\infty, \infty)$. The scaler in front of the integral is to normalize the function between -1 and 1. Here are some basic facts:

1. the range is between -1 and 1.
2. $\operatorname{erf}(+\infty) = 1$ and $\operatorname{erf}(-\infty) = -1$.
3. $\operatorname{erf}(x)$ is strictly monotonely increasing.
4. $\operatorname{erf}(x)$ is an odd function.
5. the derivative is $\frac{2}{\sqrt{\pi}} e^{-x^2}$

6. the antiderivative is $x\,\mathrm{erf}(x) + \frac{1}{\sqrt{\pi}} e^{-x^2}$

The graph of the error function is below.

Figure 2.1: Error Function, from wiki

The implementation is a Java translation of c code from fdlibm53 written by Sun. There are a few ways to implement the error function, such as numerical integration, taylor expansion. But Sun's implementation is accurate and fast. The accuracy is 1ulp, the performance is ?ms per call on average.

2.2 Complementary Error Function

The complementary error function is defined as the following integral

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{+\infty} e^{-t^2}\, dt$$

over the interval $(-\infty, \infty)$. $erfc(x)$ is the same integral as the error function, except the integral range now is over (x, ∞) rather than $(0, x)$. It is easy to show

$$erf(x) + erfc(x) = 1$$

Here are more basic facts:

1. the range is between 0 and 2.
2. $erf(+\infty) = 0$ and $erf(-\infty) = 2$.
3. $erf(x)$ is strictly monotonely decreasing.

The graph of the error function is below.

One way to implement $erfc(x)$ is to use $1 - erf(x)$. However, when x is large and $erf(x)$ is near 1, this suffers from cancellation error. One way to fix this is to use BigDecimal when x is large.

Fortunately, Sun's fdlibm53 has an implmentation for the complementary error function too. The accuracy is 1ulp, the performance is ?ms.

2.3 Error Function Inverse

Since the error function is strictly increasing on the entire real axis, the inverse function exists in $(-1, 1)$. The implementation is a translation of c code from cephes. performance?

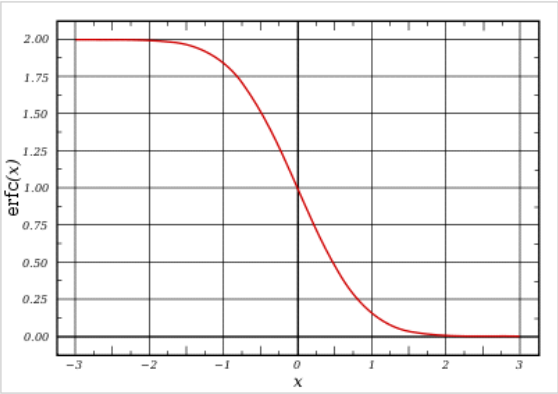

Figure 2.2: Complementary Error Function, from wiki

CHAPTER 3

GAMMA FUNCTION

Contents

This chapter discusses the gamma function, the log gamma function and the incomplete Gamma function.

3.1 Gamma Function

The Gamma function is defined as the following integral

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$

on $(-\infty, \infty)$ except 0 and negative integers. It is an extension of the factorial function for positive integers. Here are some basic facts:

1. For positive integer n, $\Gamma(n) = (n-1)!$ with $0! = 1$.
2. For any real x in the domain, $\Gamma(x) = x\Gamma(x-1)$.
3. For any real x in the domain,

$$\Gamma(1-x)\Gamma(x) = \frac{\pi}{sin(\pi x)}$$

This is called Euler's reflection and can be used to compute gamma function values for negative x if we know the values for positive x.

4. $\Gamma(x)$ is asymptotic to $\frac{1}{x}$ when $x \to 0$, and it asymptotic to $\left(\frac{x}{e}\right)^x \sqrt{\frac{2\pi}{x}}$ when $x \to \infty$

The graph of the gamma function is below.

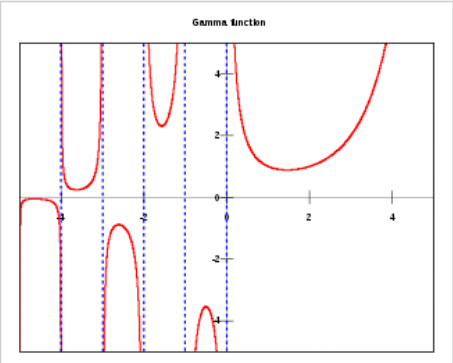

Figure 3.1: Gamma Function, from wiki

When x > 171.624, the gamma function value is too large to fit into a double and thus causes overflow. One way to get around this is to use log gamma function in the next section.

We tested several implementations:

- log gamma based implementation
- The cephes lib implmentation
- The Cody implementation
- Lanzcos 15-term implementation
- The implementation based on the reciprocal gamma function
- The implementation based on Pugh's Thesis

Accuracy and Performance.

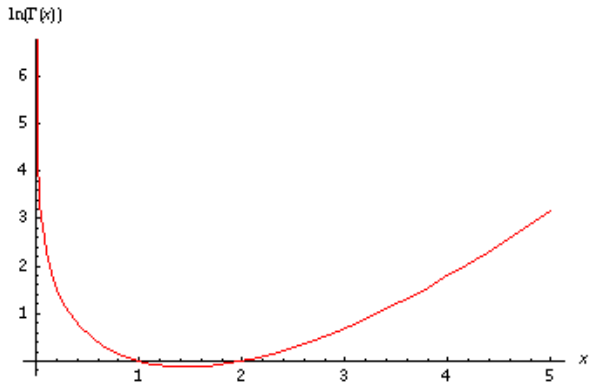

Figure 3.2: Log Gamma Function, from MathWorld

3.2 Log Gamma Function

The log gamma function is the log of the gamma function defined on the positive real axis. The graph of the gamma function is below.

The implementation is translated from fdlibm53. However, since loggamma is zero when x = 1 and 2. The original implementation suffers from cancellation near these two points. One way to fix it is to use BigDecimal, though it suffers performance hit.

Accuracy and Performance.

3.3 Incomplete Gamma Function

The gamma function is definted as a definite integral over a constant interval $(0,\infty)$. By changing the interval, we can define the incomplete gamma function. The upper incomplete gamma function is defined as the following integral

$$\Gamma(a,x) = \int_x^\infty t^{a-1}e^{-t}dt$$

The lower incomplete gamma function is defined as

$$\gamma(a,x) = \int_0^x t^{a-1}e^{-t}dt$$

and thus

$$\Gamma(a,x) + \gamma(a,x) = \Gamma(a)$$

Now scale $\Gamma(a,x)$ and $\gamma(a,x)$ by $\Gamma(a)$, we yield the normalized definitions:

$$P(a,x) = \frac{\gamma(a,x)}{\Gamma(a)} = \frac{1}{\Gamma(a)}\int_0^x t^{a-1}e^{-t}dt$$

and

$$Q(a,x) = \frac{\Gamma(a,x)}{\Gamma(a)} = \frac{1}{\Gamma(a)} \int_x^\infty t^{a-1} e^{-t} dt$$

These functions are defined in the domain $a > 0$ and $x >= 0$. Here are some basic facts for $P(a,x)$:

1. the range of the function is $[0,1)$.
2. the function changes most rapidly around $x = a$.
3. the function is strictly increasing for x when a is fixed.
4. if $a > b$, then $P(a,x) < P(b,x)$ for all x.
5. $P(0,x) = 1 + \mathrm{Ei}(-x)$???
6. if $a \to \infty$, $P(a,x) \to 0$ for all x.???

The graph for $P(a,x)$ is showing some of the above properties.

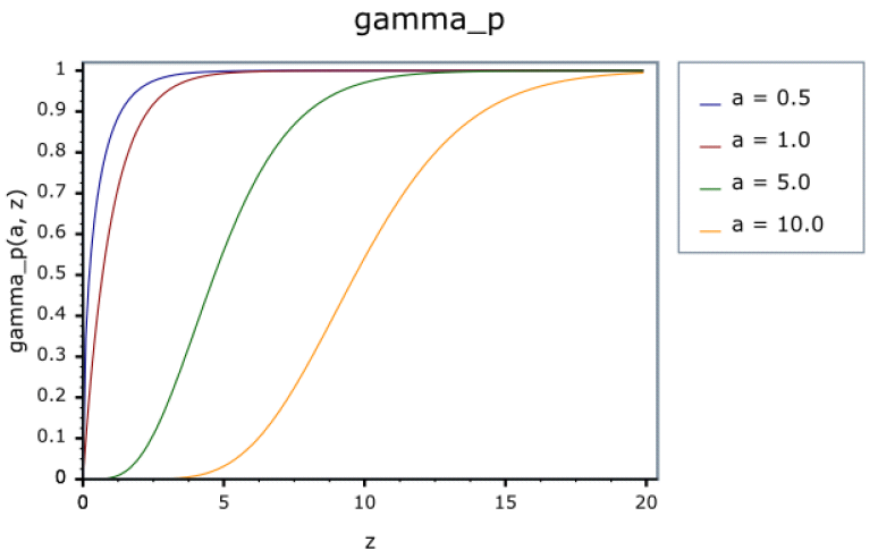

Figure 3.3: Normalized Lower incomplete Gamma Function, from Boost

Similarly, here are some basica facts for $Q(a,x)$:

1. the range of the function is $[0,1)$.
2. the function changes most rapidly around $x = a$.

3. the function is strictly decreasing for x when a is fixed.
4. if $a > b$, then $Q(a,x) > Q(b,x)$ for all x.
5. $Q(0,x) = -\mathrm{Ei}(-x)$???
6. if $a \to \infty$, $P(a,x) \to 0$ for all x.???

The graph for $Q(a,x)$ shows some of the above properties.

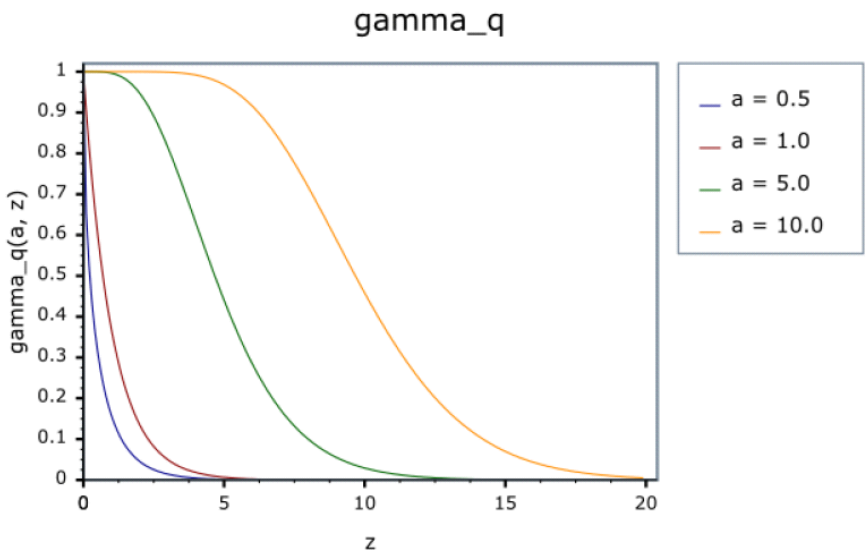

Figure 3.4: Normalized Upper incomplete Gamma Function, from Boost

3.4 Reference

Java implementations:

- JLapack: http://www.netlib.org/lapack/
- Colt implementation:
- Jsci implementation:
- ojAlgo
- Jama
- Jampack

- UJMP: http://www.ujmp.org/

Other implementations:

- Aglib: for C++, .net
- newmat: a good C++ implementation
- BLAS: http://www.netlib.org/blas/
- BOOST: http://www.boost.org/
- PLapack C implementation: http://www.cs.utexas.edu/ plapack/

BLAS

3.5 Bibliography

[1] Carpenter, R.H.S., *M*ovements of the Eyes, 2nd Edition, Pion Publishing, 1988.

[2] Franklin, G.F., Powel, J.D., Workman, M.L., *D*igital Control of Dynamic Systems, Second Edition, Addison-Wesley, 1990.

[3] Oh, P.Y., Allen, P.K., "Design a Partitioned Visual Feedback Controller," *I*EEE Int Conf Robotics & Automation, Leuven, Belgium, pp. 1360-1365 5/98

[4] Oh, P.Y., Allen, P.K., "Visual Servoing by Partitioning Degrees of Freedom," *I*EEE Trans on Robotics & Automation, V17, N1, pp. 1-17, 2/01

CHAPTER 4

BETA FUNCTION

Contents

In this chapter, we discuss the Beta function related functions, namely, Beta Fucntion and incomplete Beta function.

4.1 Beta Function

The Beta function is definted in terms of the Gamma function:

$$B(x,y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)} = \int_0^1 t^{x-1}(1-t)^{y-1}dt$$

for $x > 0$ and $y > 0$. Here are some basic facts about the beta function:

1. Beta function is symmetric, $B(x,y) = B(y,x)$.
2. For any real x in the domain, $\Gamma(x) = x\Gamma(x-1)$.
3. For any real x in the domain,

$$\Gamma(1-x)\Gamma(x) = \frac{\pi}{sin(\pi x)}$$

   This is called Euler's reflection and can be used to compute gamma function values for negative x if we know the values for positive x.

The graph for the beta function is below.

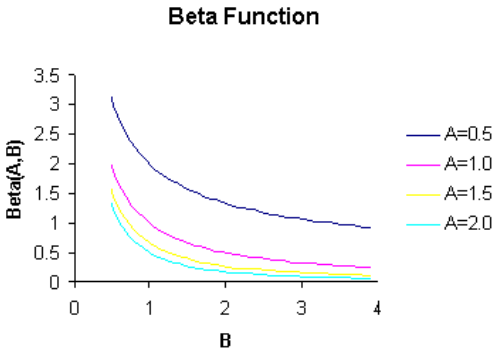

Figure 4.1: Incomplete Beta Function, from Boost

4.2 Incomplete Beta Function

The incomplete Beta function is defined as

$$B(x;a,b) = \int_0^x t^{a-1}(1-t)^{b-1}\,dt$$

for $a > 0$, $b > 0$, and $0 < x < 1$. The regularized incomplete Beta function is defined as

$$I_x(a,b) = \frac{B(x;a,b)}{B(a,b)}$$

similar to the case of incomplete gamma function. Similarly, we could define the complementary regulaized incomplete beta function. Here are some basica facts about the incomplete beta function:

The graph is below

4.3 Bibliography

[1] Demmel, J.W., *Applied Numerical Linear Algebra*, SIAM, 1997.

[2] Golub, G.H., Loan, C.F., *Matrix Computations*, 3rd Edition, The Johns Hopkins University Press, 1996.

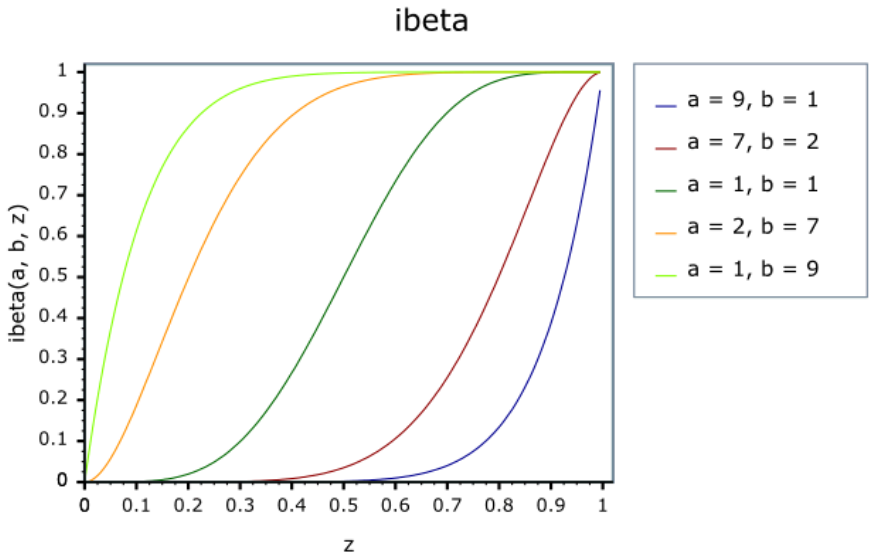

Figure 4.2: Incomplete Beta Function, from Boost