Debugging Floating-Point Math in Racket

Neil Toronto

RacketCon 2013

Racket Floating-Point Support

- Fast (JIT-ed) and compliant (IEEE 754 and C99)

Racket Floating-Point Support

- Fast (JIT-ed) and compliant (IEEE 754 and C99)
- More flonum (i.e. 64-bit float) functions:
  - **`math/special-functions`**: gamma, beta, psi, zeta, erf, etc.
  - **`math/distributions`**: Gamma, Normal, etc.

Racket Floating-Point Support

- Fast (JIT-ed) and compliant (IEEE 754 and C99)
- More flonum (i.e. 64-bit float) functions:
  - `math/special-functions`: gamma, beta, psi, zeta, erf, etc.
  - `math/distributions`: Gamma, Normal, etc.
- Other floating-point modules:
  - `racket/extflonum`: basic 80-bit operations
  - `math/bigfloat`: arbitrary-precision floats

Racket Floating-Point Support

- Fast (JIT-ed) and compliant (IEEE 754 and C99)
- More flonum (i.e. 64-bit float) functions:
  - **`math/special-functions`**: gamma, beta, psi, zeta, erf, etc.
  - **`math/distributions`**: Gamma, Normal, etc.
- Other floating-point modules:
  - **`racket/extflonum`**: basic 80-bit operations
  - **`math/bigfloat`**: arbitrary-precision floats
- **`math/flonum`**: a bunch of weird things like **`fl`**, **`flnext`**, **`+max.0`**, **`flonum->ordinal`**, **`fllog1p`**, **`flsqrt1pm1`**, **`flcospix`**

You Could Have Invented Floating-Point

Need to represent $\pm n \times 10^m$ or $\pm n \times 2^m$...

You Could Have Invented Floating-Point

Need to represent $\pm n \times 10^m$ or $\pm n \times 2^m$...

```
(struct: float ([sign : (U -1 1)]
                [sig : Natural]
                [exp : Integer]))
```

You Could Have Invented Floating-Point

Need to represent $\pm n \times 10^m$ or $\pm n \times 2^m$...

```
(struct: float ([sign : (U -1 1)]
                [sig : Natural]
                [exp : Integer]))

(: float->real (float -> Real))
(define (float->real x)
  (match-define (float s n m) x)
  (* s n (expt 2 m)))
```

You Could Have Invented Floating-Point

Need to represent $\pm n \times 10^m$ or $\pm n \times 2^m$...

```
(struct: float ([sign : (U -1 1)]
                [sig : Natural]
                [exp : Integer]))

(: float->real (float -> Real))
(define (float->real x)
  (match-define (float s n m) x)
  (* s n (expt 2 m)))

> (float->real (float -1 10 0))
-10
```

You Could Have Invented Floating-Point

Need to represent $\pm n \times 10^m$ or $\pm n \times 2^m$...

```
(struct: float ([sign : (U -1 1)]
                [sig : Natural]
                [exp : Integer]))

(: float->real (float -> Real))
(define (float->real x)
  (match-define (float s n m) x)
  (* s n (expt 2 m)))

> (float->real (float -1 10 0))
-10

> (float->real (float -1 10 3))
-80
```

You Could Have Invented Floating-Point Multiplication

```
(struct: float ([sign : (U -1 1)]
                [sig : Natural]
                [exp : Integer]))
```

You Could Have Invented Floating-Point Multiplication

```
(struct: float ([sign : (U -1 1)]
                [sig : Natural]
                [exp : Integer]))
```

$$s_1 \times n_1 \times 2^{m_1} \times s_2 \times n_2 \times 2^{m_2}$$

$$= (s_1 \times s_2) \times (n_1 \times n_2) \times 2^{m_1 + m_2}$$

You Could Have Invented Floating-Point Multiplication

```
(struct: float ([sign : (U -1 1)]
                [sig : Natural]
                [exp : Integer]))
```

$$s_1 \times n_1 \times 2^{m_1} \times s_2 \times n_2 \times 2^{m_2}$$
$$= (s_1 \times s_2) \times (n_1 \times n_2) \times 2^{m_1+m_2}$$

```
(: float* (float float -> float))
(define (float* x1 x2)
  (match-define (float s1 n1 m1) x1)
  (match-define (float s2 n2 m2) x2)
  (float (* s1 s2) (* n1 n2) (+ m1 m2)))
```

You Could Have Invented Floating-Point Multiplication

```
(struct: float ([sign : (U -1 1)]
                [sig : Natural]
                [exp : Integer]))
```

$$s_1 \times n_1 \times 2^{m_1} \times s_2 \times n_2 \times 2^{m_2}$$

$$= (s_1 \times s_2) \times (n_1 \times n_2) \times 2^{m_1+m_2}$$

```
(: float* (float float -> float))
(define (float* x1 x2)
  (match-define (float s1 n1 m1) x1)
  (match-define (float s2 n2 m2) x2)
  (float (* s1 s2) (* n1 n2) (+ m1 m2)))

> (float->real (float* (float -1 10 0)
                       (float -1 10 3)))
800
```

Finite Approximation

- Actual flonum fields are fixed-size, requiring
  - Rounding least significant bit after operations
  - Representations for overflow (i.e. `+inf.0` and `-inf.0`) and underflow (i.e. `+0.0` and `-0.0`)

Finite Approximation

- Actual flonum fields are fixed-size, requiring
  - Rounding least significant bit after operations
  - Representations for overflow (i.e. `+inf.0` and `-inf.0`) and underflow (i.e. `+0.0` and `-0.0`)
- Consequence: a natural well-order over flonums

```
> (flnext 0.0) ; from math/flonum
4.9406564584125e-324
```

Finite Approximation

- Actual flonum fields are fixed-size, requiring
  - Rounding least significant bit after operations
  - Representations for overflow (i.e. `+inf.0` and `-inf.0`) and underflow (i.e. `+0.0` and `-0.0`)
- Consequence: a natural well-order over flonums

```
> (flnext 0.0) ; from math/flonum
4.9406564584125e-324

> (list (flonum->ordinal 0.0)
        (flonum->ordinal +max.0))
'(0 9218868437227405311)
```

Finite Approximation

- Actual flonum fields are fixed-size, requiring
  - Rounding least significant bit after operations
  - Representations for overflow (i.e. `+inf.0` and `-inf.0`) and underflow (i.e. `+0.0` and `-0.0`)
- Consequence: a natural well-order over flonums

```
> (flnext 0.0) ; from math/flonum
4.9406564584125e-324

> (list (flonum->ordinal 0.0)
        (flonum->ordinal +max.0))
'(0 9218868437227405311)

> (flonums-between 0.0 1.0)
4607182418800017408 ; 4.6 *quintillion*
```

Finite Approximation

- Actual flonum fields are fixed-size, requiring
  - Rounding least significant bit after operations
  - Representations for overflow (i.e. `+inf.0` and `-inf.0`) and underflow (i.e. `+0.0` and `-0.0`)
- Consequence: a natural well-order over flonums

```
> (flnext 0.0) ; from math/flonum
4.9406564584125e-324

> (list (flonum->ordinal 0.0)
        (flonum->ordinal +max.0))
'(0 9218868437227405311)

> (flonums-between 0.0 1.0)
4607182418800017408 ; 4.6 *quintillion*
```

- Consequence: most flonum functions aren't exact

Correctness Means Minimizing Error

- A flonum's **unit in last place (ulp)** is the distance between it and the next flonum

```
> (flulp #i355/113) ; from math/flonum
4.440892098500626e-16
```

Correctness Means Minimizing Error

- A flonum's **unit in last place (ulp)** is the distance between it and the next flonum

```
> (flulp #i355/113) ; from math/flonum
4.440892098500626e-16
```

- Error is distance from the true value, in ulps

```
> #i355/113 pi
3.141592920353983
3.141592653589793
```

Correctness Means Minimizing Error

- A flonum’s **unit in last place (ulp)** is the distance between it and the next flonum

```
> (flulp #i355/113) ; from math/flonum
4.440892098500626e-16
```

- Error is distance from the true value, in ulps

```
> #i355/113 pi
3.1415929203539825
3.141592653589793
```

```
> (flulp-error #i355/113 pi)
600699552.0 ; 600.7 million ulps
```

Correctness Means Minimizing Error

- A flonum's **unit in last place (ulp)** is the distance between it and the next flonum

```
> (flulp #i355/113) ; from math/flonum
4.440892098500626e-16
```

- Error is distance from the true value, in ulps

```
> #i355/113 pi
3.1415929203539825
3.141592653589793
```

```
> (flulp-error #i355/113 pi)
600699552.0 ; 600.7 million ulps
```

- A flonum function is **correctly rounded*** when its outputs' maximum error is no more than 0.5 ulps

Correctness Means Minimizing Error

- A flonum's **unit in last place (ulp)** is the distance between it and the next flonum

```
> (flulp #i355/113) ; from math/flonum
4.440892098500626e-16
```

- Error is distance from the true value, in ulps

```
> #i355/113 pi
3.141592920353982
3.141592653589793

> (flulp-error #i355/113 pi)
600699552.0 ; 600.7 million ulps
```

- A flonum function is **correctly rounded*** when its outputs' maximum error is no more than 0.5 ulps

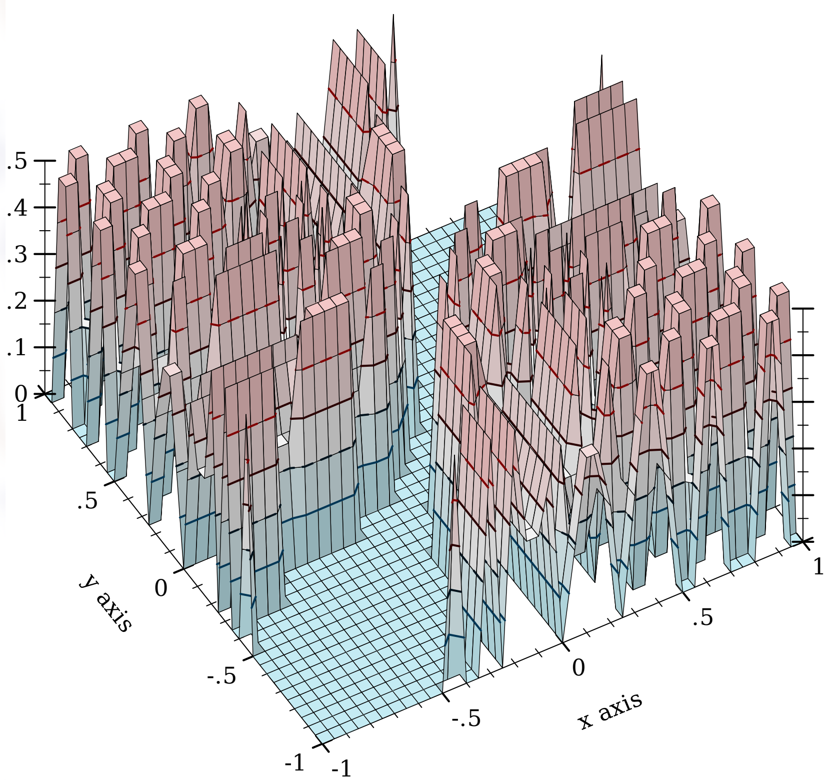
* assuming inputs are exact; i.e. no guarantees are given for arguments with error

Correctness Example: Subtraction

```
> (plot3d (contour-intervals3d
           (λ (x y) (let ([x  (fl x)] [y  (fl y)])
                      (define z* (- (inexact->exact x)
                                    (inexact->exact y)))
                      (flulp-error (fl- x y) z*)))
           -1 1 -1 1))
```

Correctness Example: Subtraction

```
> (plot3d (contour-intervals3d
            (λ (x y) (let ([x  (fl x)] [y  (fl y)])
                       (define z* (- (inexact->exact x)
                                     (inexact->exact y)))
                       (flulp-error (fl- x y) z*)))
            -1 1 -1 1))
```

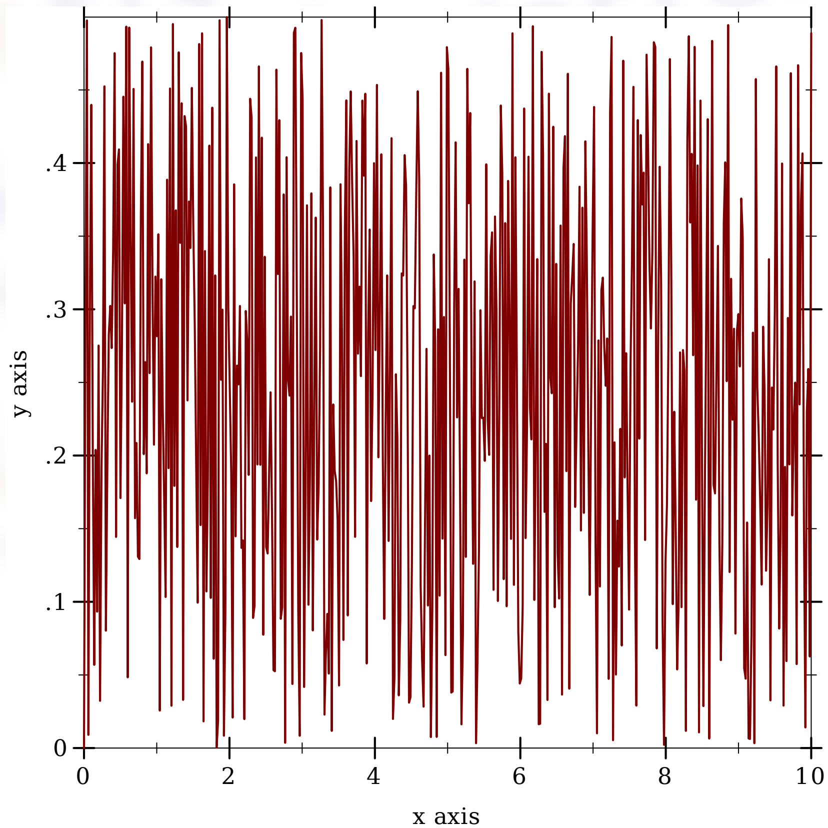


Correctness Example: Logarithm

```
> (require math/bigfloat) ; default sig. size: 128 bits
> (plot (function
         (λ (x) (let ([x  (fl x)])
                  (define z* (bigfloat->real (bflog (bf x))))
                  (flulp-error (fllog x) z*)))
         0 10))
```

Correctness Example: Logarithm

```
> (require math/bigfloat) ; default sig. size: 128 bits
> (plot (function
         (λ (x) (let ([x  (fl x)])
                  (define z* (bigfloat->real (bflog (bf x))))
                  (flulp-error (fllog x) z*)))
         0 10))
```

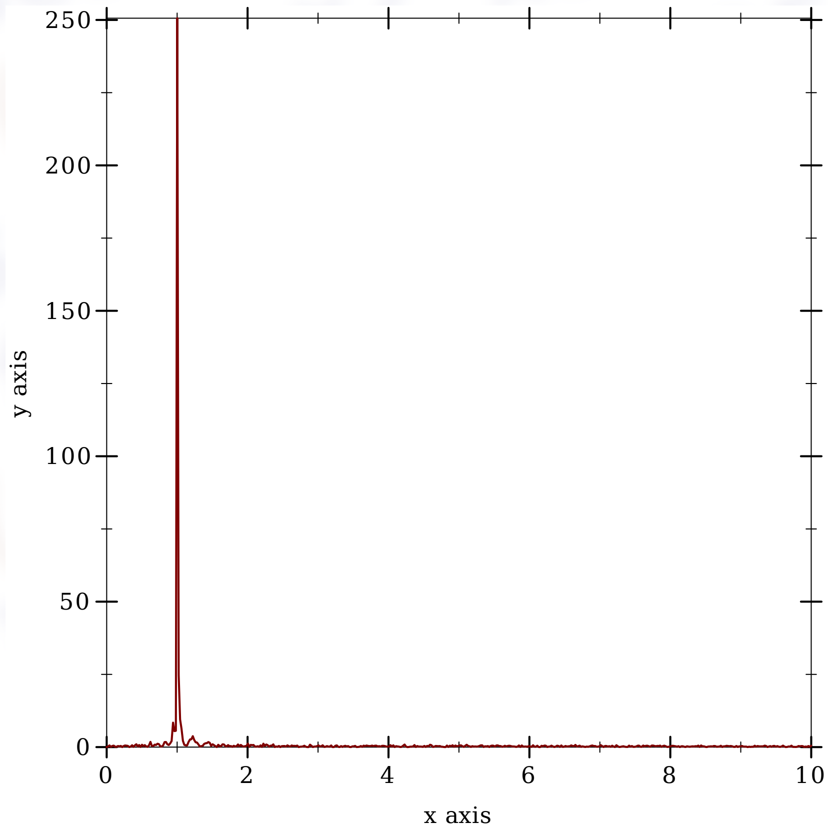


Correctness is Noncompositional

```
> (plot (function
         (λ (x)
           (define z* (bigfloat->real (bflog (bf x))))
           (flulp-error (fllog (fl x)) z*)))
         0 10)
```

Correctness is Noncompositional

```
> (plot (function
         (λ (x)
           (define z* (bigfloat->real (bflog (bf x))))
           (flulp-error (fllog (fl x)) z*))
         0 10))
```



Debugging: Geometric Inverse CDF

Implement $f(p,u) = \log(u)/\log(1-p)$ for $p, u \in [0,1]$

Debugging: Geometric Inverse CDF

Implement $f(p, u) = \log(u) / \log(1 - p)$ for $p, u \in [0, 1]$

- First stab:

```
(define (geom p u)
  (fl/ (fllog u) (fllog (fl- 1.0 p))))
```

Debugging: Geometric Inverse CDF

Implement $f(p, u) = \log(u)/\log(1-p)$ for $p, u \in [0, 1]$

- First stab:

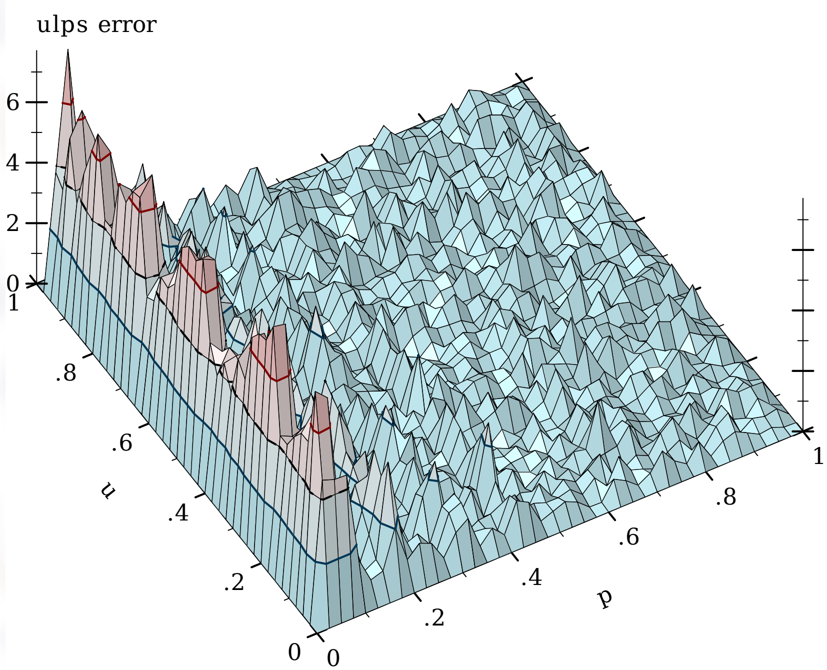
```
(define (geom p u)
  (fl/ (fllog u) (fllog (fl- 1.0 p))))
```

- Reference implementation:

```
(define (geom* p u)
  (let ([p  (bf p)] [u  (bf u)])
    (bigfloat->real
     (bf/ (bflog u) (bflog (bf- 1.bf p))))))
```

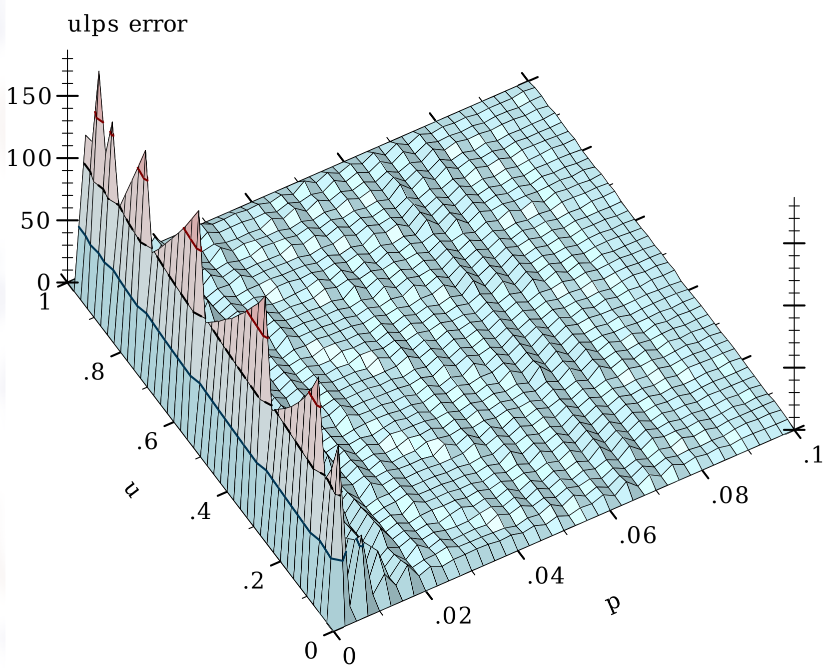
Debugging: Geometric Inverse CDF

- Error plot for `geom` for `p <= 1`:



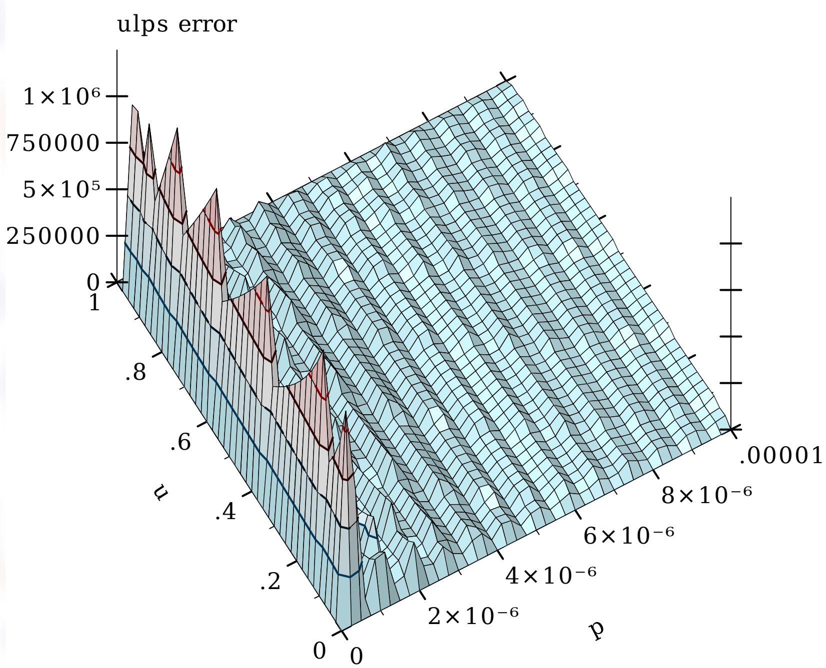
Debugging: Geometric Inverse CDF

- Error plot for `geom` for `p <= 0.1`:



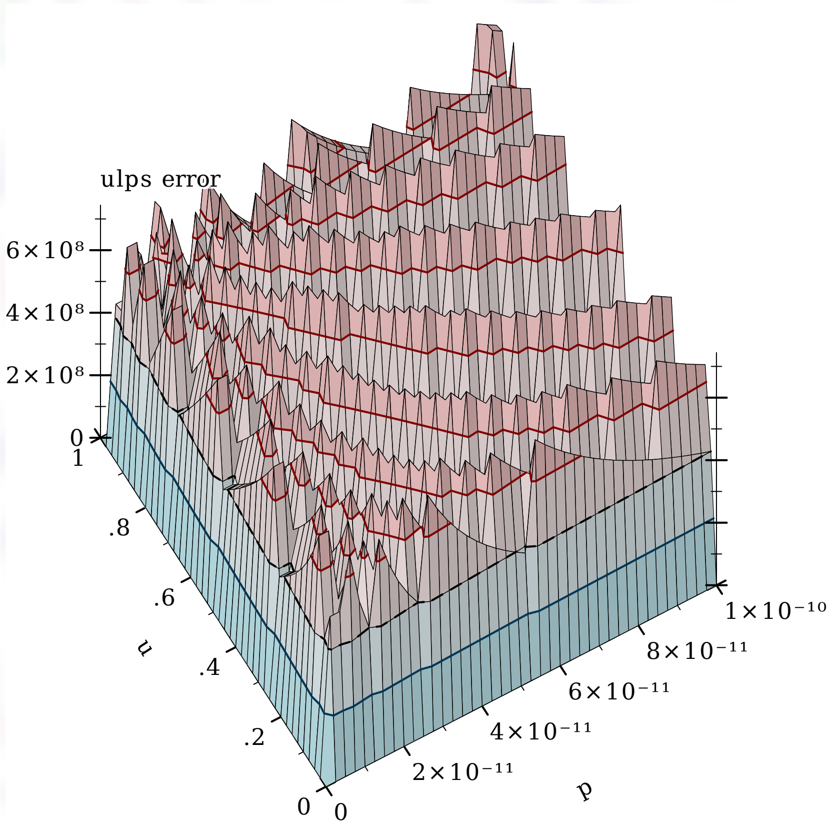
Debugging: Geometric Inverse CDF

- Error plot for `geom` for `p <= 1e-05`:



Debugging: Geometric Inverse CDF

- Error plot for `geom` for `p <= 1e-10`:



Argh!

Q. Is this normal???

A. Yes. Most straightforward flonum function implementations have low error most places and high error (often unbounded) in others.

Argh!

Q. Is this normal???

A. Yes. Most straightforward flonum function implementations have low error most places and high error (often unbounded) in others.

The good news:
You can usually fix them using just flonum ops.

Argh!

Q. Is this normal???

A. Yes. Most straightforward flonum function implementations have low error most places and high error (often unbounded) in others.

The good news:
You can usually fix them using just flonum ops.

Q. How do I fix them???

A. Most functions—not implementations, but functions themselves—have **ill-conditioned** places where they turn low input error into high output error. Avoid these badlands.

The Floating-Point Priest Says...

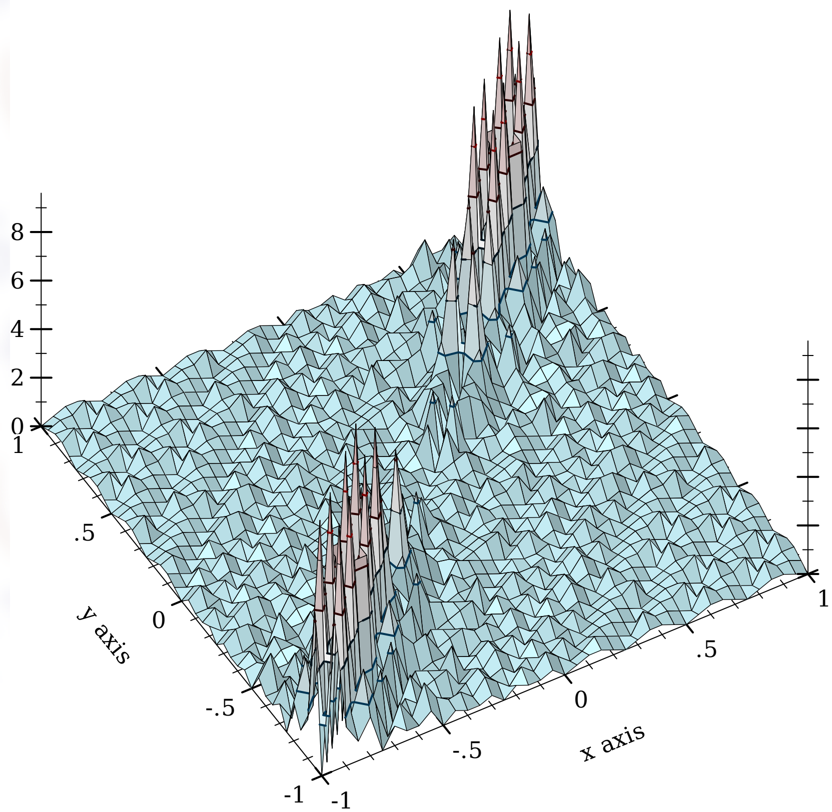
"The condition number of a function is the absolute value of the ratio of its derivative and its value, multiplied by the blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah."

The Badlands: Subtraction

```
> (plot3d (contour-intervals3d
           (λ (x y)
             (define z* (- x y))
             (flulp-error (fl- (fl x) (fl y)) z*))
           -1 1 -1 1))
```

The Badlands: Subtraction

```
> (plot3d (contour-intervals3d
           (λ (x y)
             (define z* (- x y))
             (flulp-error (fl- (fl x) (fl y)) z*))
           -1 1 -1 1))
```

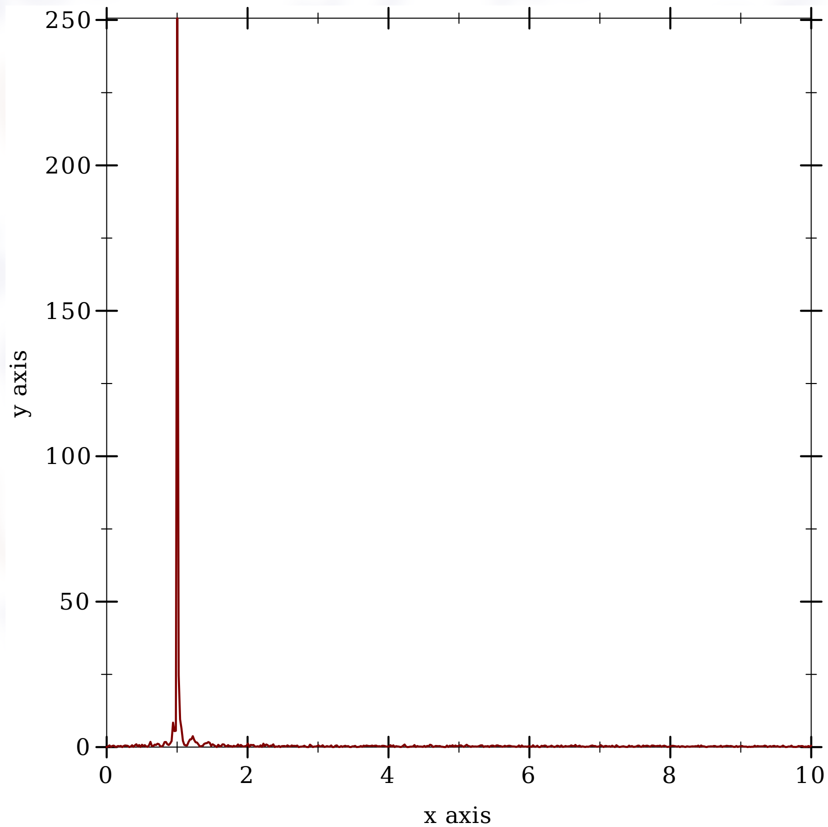


The Badlands: Logarithm

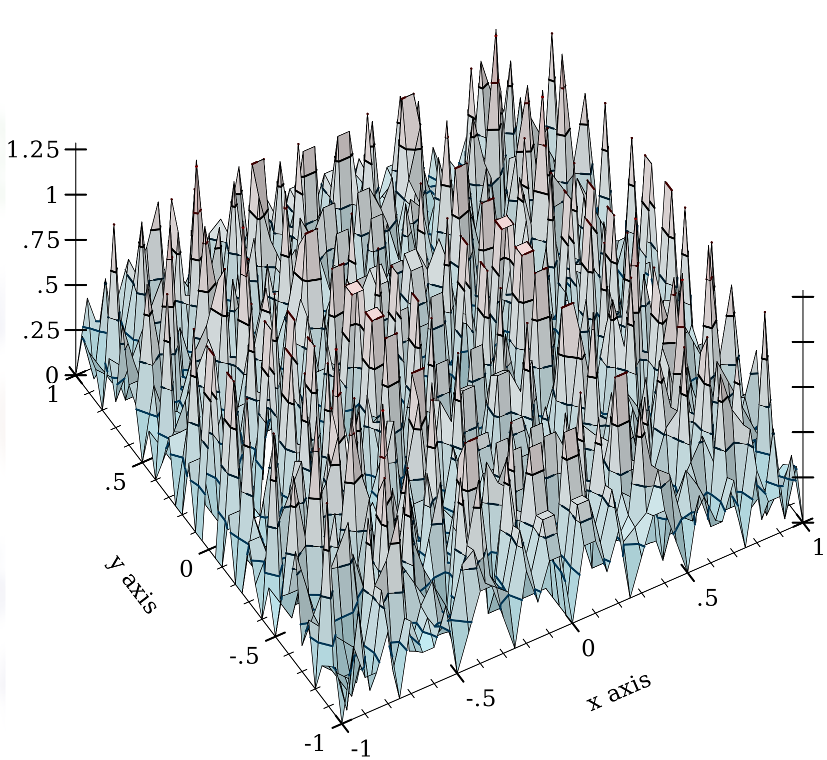
```
> (plot (function
         (λ (x)
           (define z* (bigfloat->real (bflog (bf x))))
           (flulp-error (fllog (fl x)) z*))
         0 10))
```

The Badlands: Logarithm

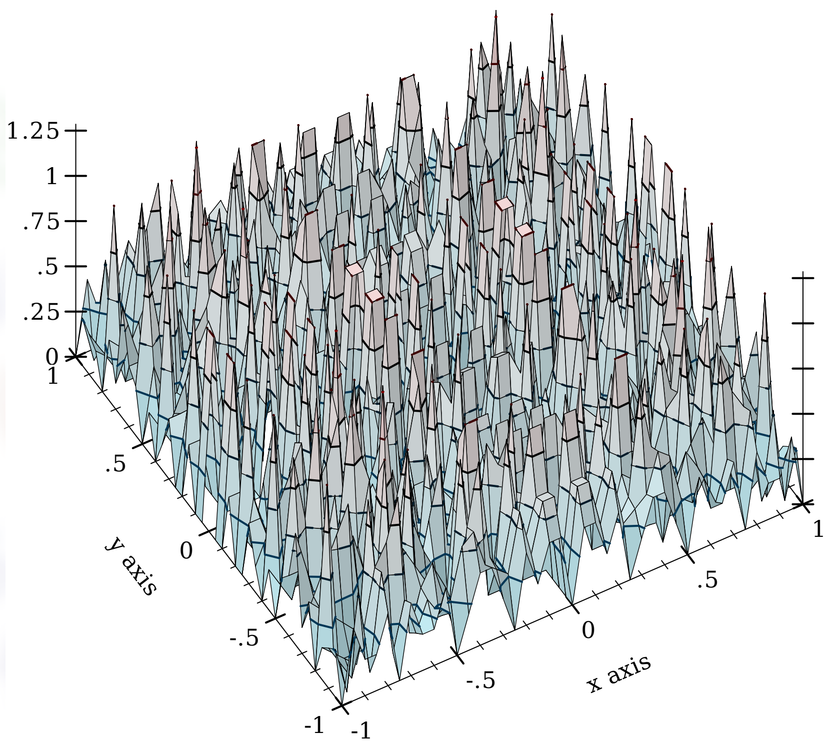
```
> (plot (function
         (λ (x)
           (define z* (bigfloat->real (bflog (bf x))))
           (flulp-error (fllog (fl x)) z*))
         0 10))
```



The Badlands: Division

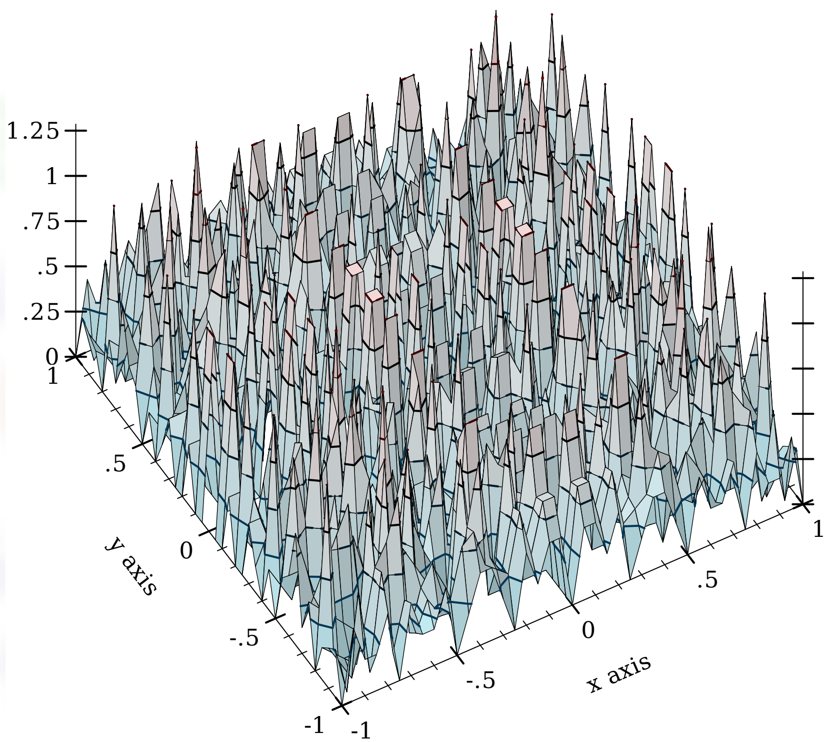


The Badlands: Division



- **No badlands:** except for flonum rounding error, division error doesn't depend on inputs

The Badlands: Division



- **No badlands:** except for flonum rounding error, division error doesn't depend on inputs
- Multiplication error is the same way

Informal Error Analysis

- Recursively reason about the body of **geom**:

```
(define (geom p u)
  (fl/ (fllog u) (fllog (fl- 1.0 p))))
```

Informal Error Analysis

- Recursively reason about the body of `geom`:

```
(define (geom p u)
  (fl/ (fllog u) (fllog (fl- 1.0 p))))
```

- Can't do anything about `fl/` except make sure its arguments are as accurate as possible

Informal Error Analysis

- Recursively reason about the body of `geom`:

```
(define (geom p u)
  (fl/ (fllog u) (fllog (fl- 1.0 p))))
```

- Can't do anything about `fl/` except make sure its arguments are as accurate as possible
- If `u` is exact, `(fllog u)` has <= 0.5 ulps error

Informal Error Analysis

- Recursively reason about the body of `geom`:

```
(define (geom p u)
  (fl/ (fllog u) (fllog (fl- 1.0 p))))
```

- Can't do anything about `fl/` except make sure its arguments are as accurate as possible
- If `u` is exact, `(fllog u)` has <= 0.5 ulps error
- If `p` is exact, `(fl- 1.0 p)` has <= 0.5 ulps error

Informal Error Analysis

- Recursively reason about the body of `geom`:

```
(define (geom p u)
  (fl/ (fllog u) (fllog (fl- 1.0 p))))
```

- Can't do anything about `fl/` except make sure its arguments are as accurate as possible
- If `u` is exact, `(fllog u)` has <= 0.5 ulps error
- If `p` is exact, `(fl- 1.0 p)` has <= 0.5 ulps error
- If `p` is exact and near `0.0`...

Informal Error Analysis

- Recursively reason about the body of `geom`:

```
(define (geom p u)
  (fl/ (fllog u) (fllog (fl- 1.0 p))))
```

- Can't do anything about `fl/` except make sure its arguments are as accurate as possible
- If `u` is exact, `(fllog u)` has <= 0.5 ulps error
- If `p` is exact, `(fl- 1.0 p)` has <= 0.5 ulps error
- If `p` is exact and near `0.0`...
  - Then `(fl- 1.0 p)` is inexact and near `1.0`...

Informal Error Analysis

- Recursively reason about the body of `geom`:

```
(define (geom p u)
  (fl/ (fllog u) (fllog (fl- 1.0 p))))
```

- Can't do anything about `fl/` except make sure its arguments are as accurate as possible
- If `u` is exact, `(fllog u)` has <= 0.5 ulps error
- If `p` is exact, `(fl- 1.0 p)` has <= 0.5 ulps error
- If `p` is exact and near `0.0`...
  - Then `(fl- 1.0 p)` is inexact and near `1.0`...
  - So `(fllog (fl- 1.0 p))` may have **high error**

Informal Error Analysis

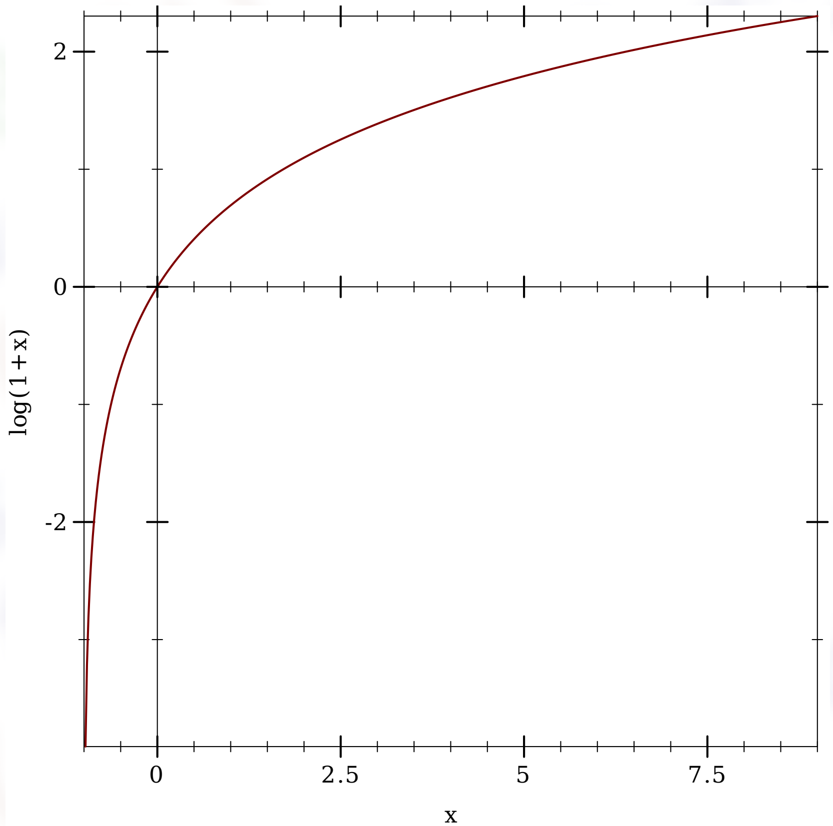
- Recursively reason about the body of `geom`:

```
(define (geom p u)
  (fl/ (fllog u) (fllog (fl- 1.0 p))))
```

- Can't do anything about `fl/` except make sure its arguments are as accurate as possible
- If `u` is exact, `(fllog u)` has <= 0.5 ulps error
- If `p` is exact, `(fl- 1.0 p)` has <= 0.5 ulps error
- If `p` is exact and near `0.0`...
  - Then `(fl- 1.0 p)` is inexact and near `1.0`...
  - So `(fllog (fl- 1.0 p))` may have **high error**
- Let's check `math/flonum` for another incantation...

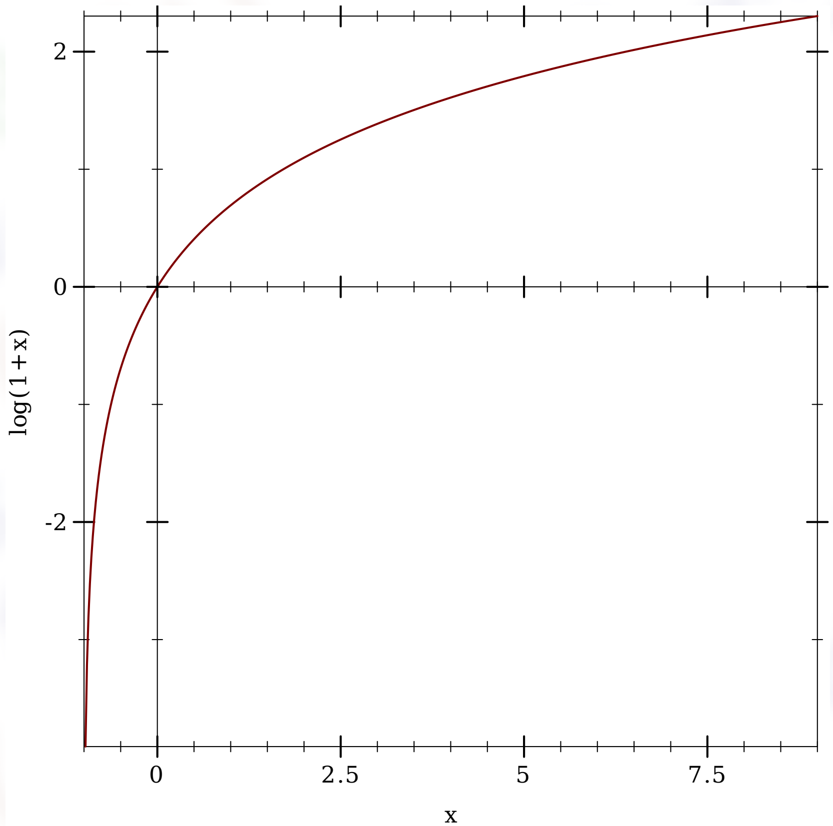
log(1+x)

- Looks interesting: **`fllog1p`**



log(1+x)

- Looks interesting: **`fllog1p`**



- We can use it almost directly—mathematically,

$$\log(1-p) = \log(1+(-p)) = \text{log1p}(-p)$$

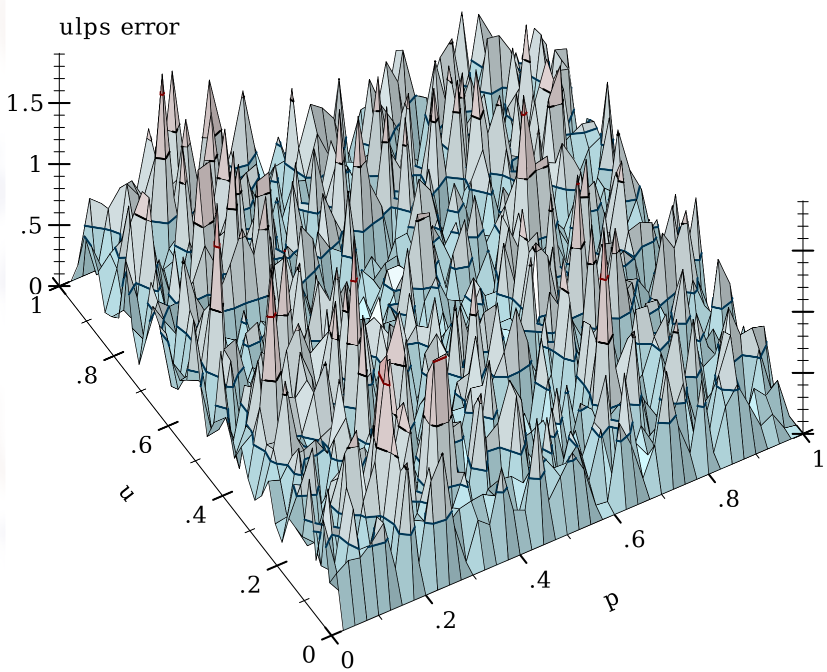
Debugging: Geometric Inverse CDF (Second Stab)

```
(define (geom p u)
  (fl/ (fllog u) (fllog1p (- p))))
```

Debugging: Geometric Inverse CDF (Second Stab)

```
(define (geom p u)
  (fl/ (fllog u) (fllog1p (- p))))
```

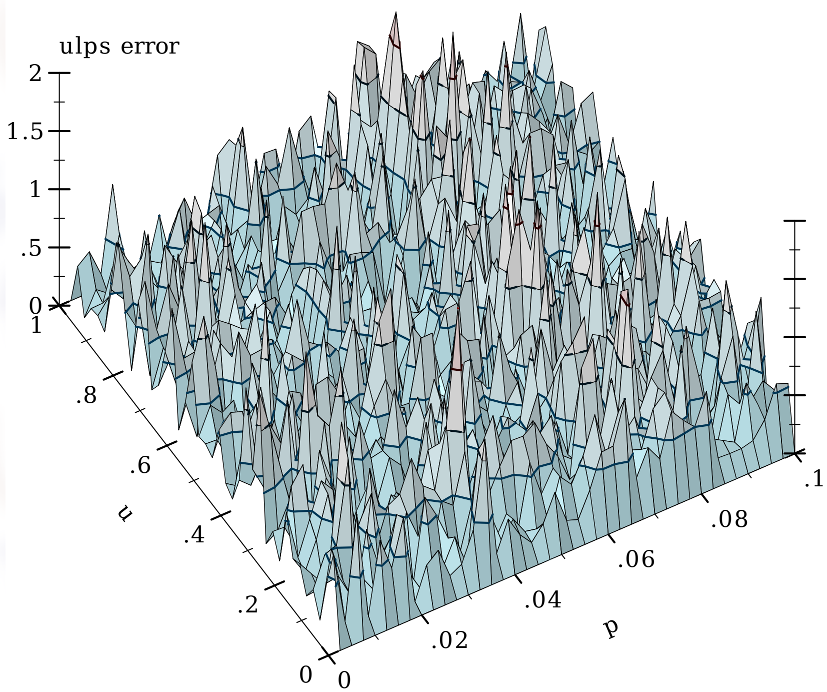
- Error plot for `geom` for `p <= 1`:



Debugging: Geometric Inverse CDF (Second Stab)

```
(define (geom p u)
  (fl/ (fllog u) (fllog1p (- p))))
```

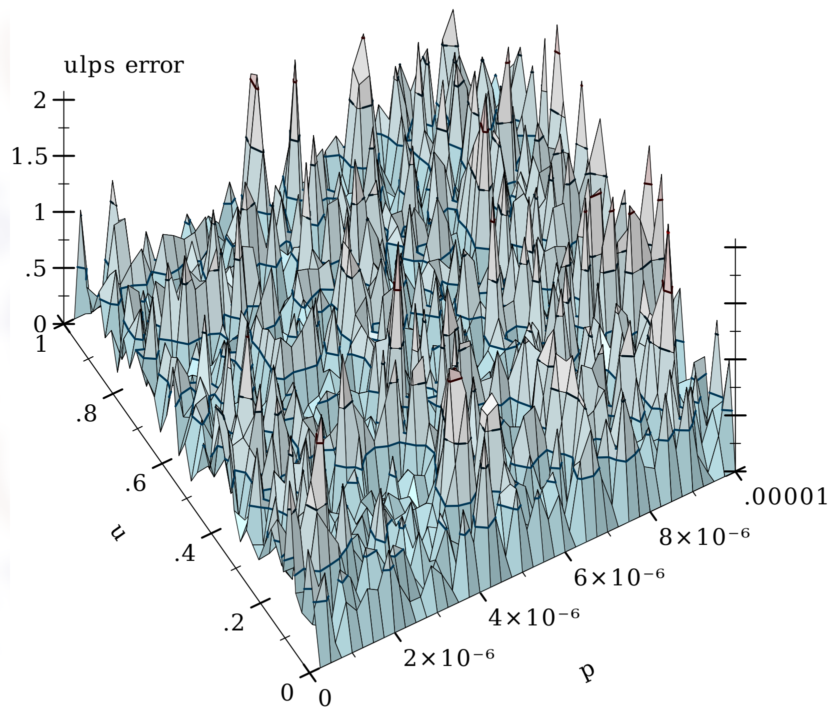
- Error plot for `geom` for `p <= 0.1`:



Debugging: Geometric Inverse CDF (Second Stab)

```
(define (geom p u)
  (fl/ (fllog u) (fllog1p (- p))))
```

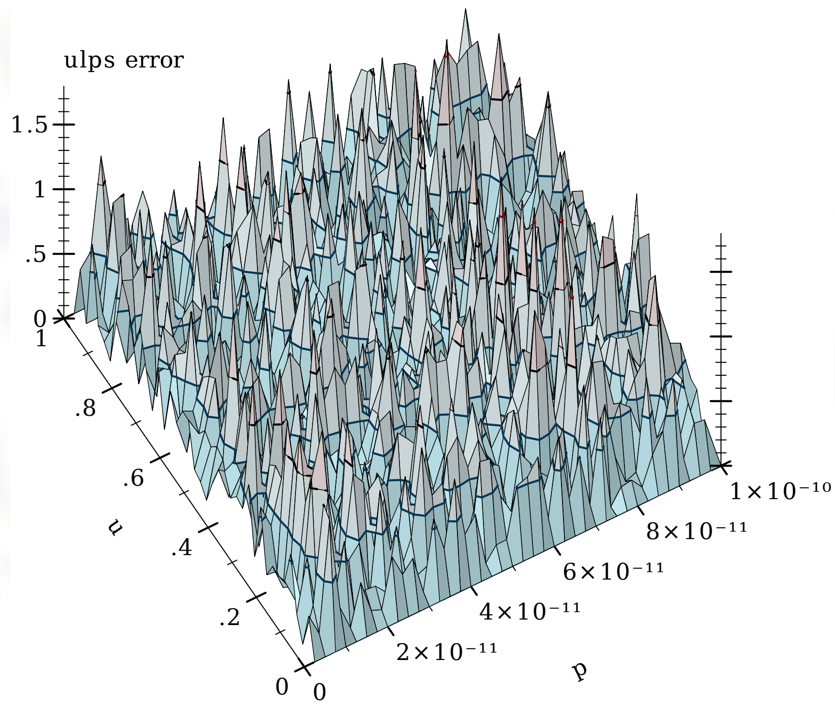
- Error plot for `geom` for `p <= 1e-05`:



Debugging: Geometric Inverse CDF (Second Stab)

```
(define (geom p u)
  (fl/ (fllog u) (fllog1p (- p))))
```

- Error plot for `geom` for `p <= 1e-10`:



Argh It Is Not Perfect!

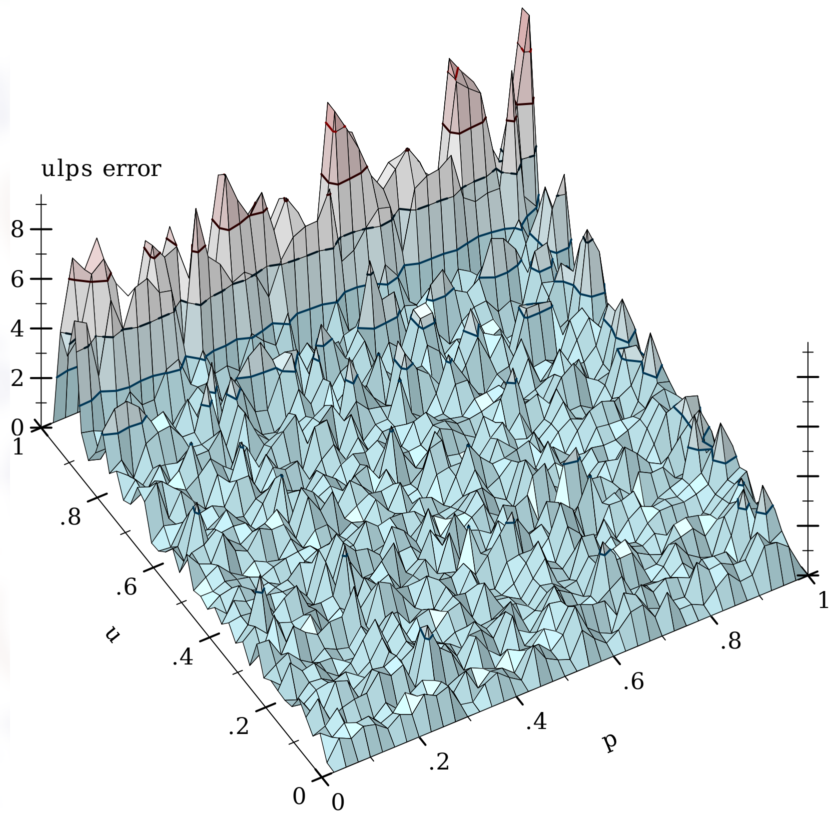
- But < 3 ulps error is very accurate

Argh It Is Not Perfect!

- But < 3 ulps error is very accurate
- Does `geom` have badlands?

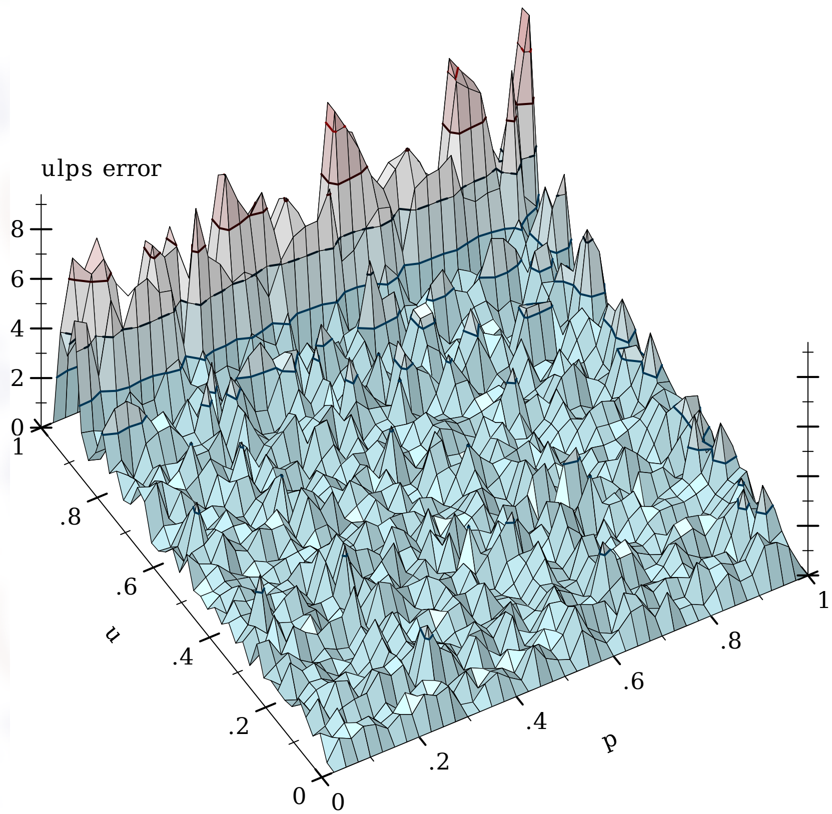
Argh It Is Not Perfect!

- But < 3 ulps error is very accurate
- Does **`geom`** have badlands?



Argh It Is Not Perfect!

- But < 3 ulps error is very accurate
- Does **geom** have badlands?



- This is a property of the **function**, so we can't do anything about it

Debugging Summary

- Make direct and reference implementations

Debugging Summary

- Make direct and reference implementations
- Repeat:
  - Plot error, identify high-error regions
  - Find badlands inputs, replace computations

Debugging Summary

- Make direct and reference implementations
- Repeat:
  - Plot error, identify high-error regions
  - Find badlands inputs, replace computations
- Avoid:
  - Subtracting nearby values
  - Taking logs of values near `1.0`
  - Other badlands (most zero crossings, exponential growth)

Debugging Summary

- Make direct and reference implementations
- Repeat:
  - Plot error, identify high-error regions
  - Find badlands inputs, replace computations
- Avoid:
  - Subtracting nearby values
  - Taking logs of values near `1.0`
  - Other badlands (most zero crossings, exponential growth)
- Move multiplication and division outward

What If I Need Moar Bits?

- `racket/extflonum`: 80-bit extended flonums
  - Requires `(extflonum-available?) = #t`
  - 64-bit significand, 15-bit exponent

```
(extfl->exact (extflexp (real->extfl 1/7)))
```

What If I Need Moar Bits?

- `racket/extflonum`: 80-bit extended flonums
  - Requires `(extflonum-available?) = #t`
  - 64-bit significand, 15-bit exponent

```
(extfl->exact (extflexp (real->extfl 1/7)))
```

- **double-double** flonums: sum of two nonoverlapping flonums represent a number
  - Requires correctly rounded arithmetic
  - ~105-bit significand, 11-bit exponent

```
(let*-values ([(x2 x1)  (fl2 1/7)]
              [(y2 y1)  (fl2exp x2 x1)])
  (fl2->real y2 y1))
```

What If I Need Moar Bits?

- `math/bigfloat`: arbitrary precision floats
  - Requires MPFR (Racket ships with libraries)
  - Any size significand, 32-bit exponent

What If I Need Moar Bits?

- `math/bigfloat`: arbitrary precision floats
  - Requires MPFR (Racket ships with libraries)
  - Any size significand, 32-bit exponent
- Exact rationals

What If I Need Moar Bits?

- `math/bigfloat`: arbitrary precision floats
  - Requires MPFR (Racket ships with libraries)
  - Any size significand, 32-bit exponent
- Exact rationals
- **AN IMPORTANT DATA POINT:**

What If I Need Moar Bits?

- `math/bigfloat`: arbitrary precision floats
  - Requires MPFR (Racket ships with libraries)
  - Any size significand, 32-bit exponent
- Exact rationals
- **AN IMPORTANT DATA POINT:**

It takes 1074-bit bigfloats to fix `geom`

using `(bflog (bf- 1.bf p))`

What If I Need Moar Bits?

- `math/bigfloat`: arbitrary precision floats
  - Requires MPFR (Racket ships with libraries)
  - Any size significand, 32-bit exponent
- Exact rationals
- **AN IMPORTANT DATA POINT:**

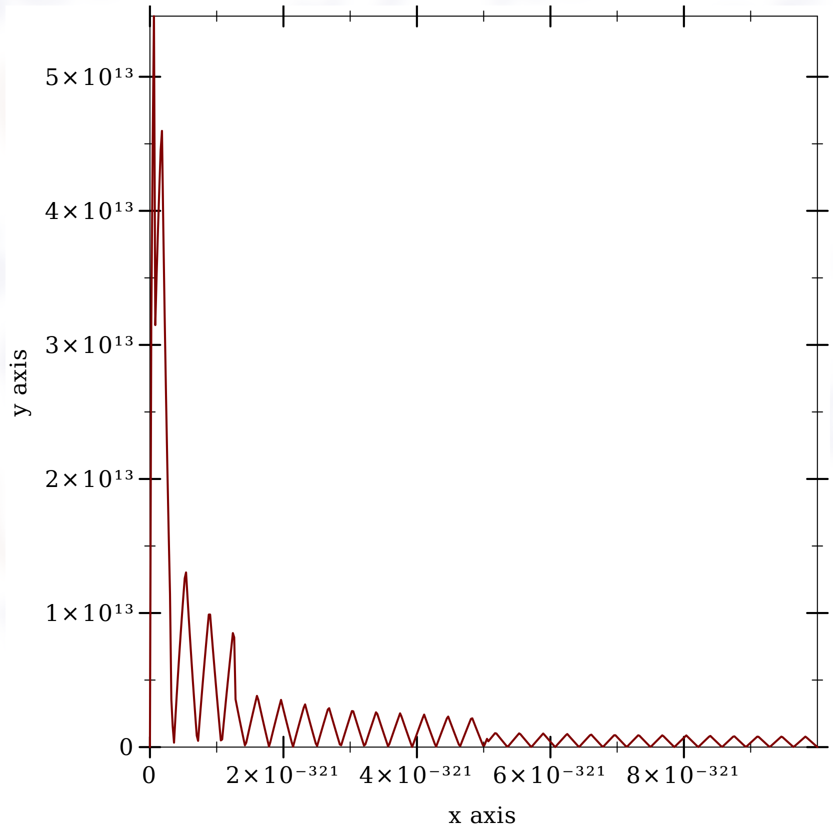
It takes 1074-bit bigfloats to fix `geom`

using `(bflog (bf- 1.bf p))`

- Conclusion: “moar bits” is not a general solution

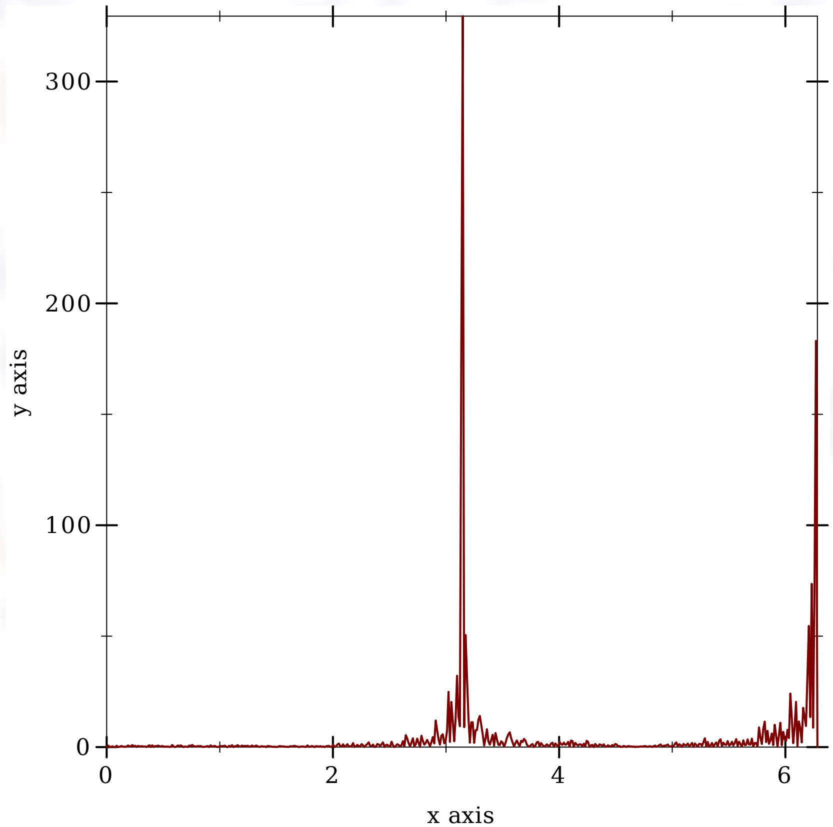
The Badlands: Square Oot

```
> (plot (function
          (λ (x)
            (define z* (bigfloat->real (bfsqrt (bf x))))
            (flulp-error (flsqrt (fl x)) z*))
          0 1e-320))
```



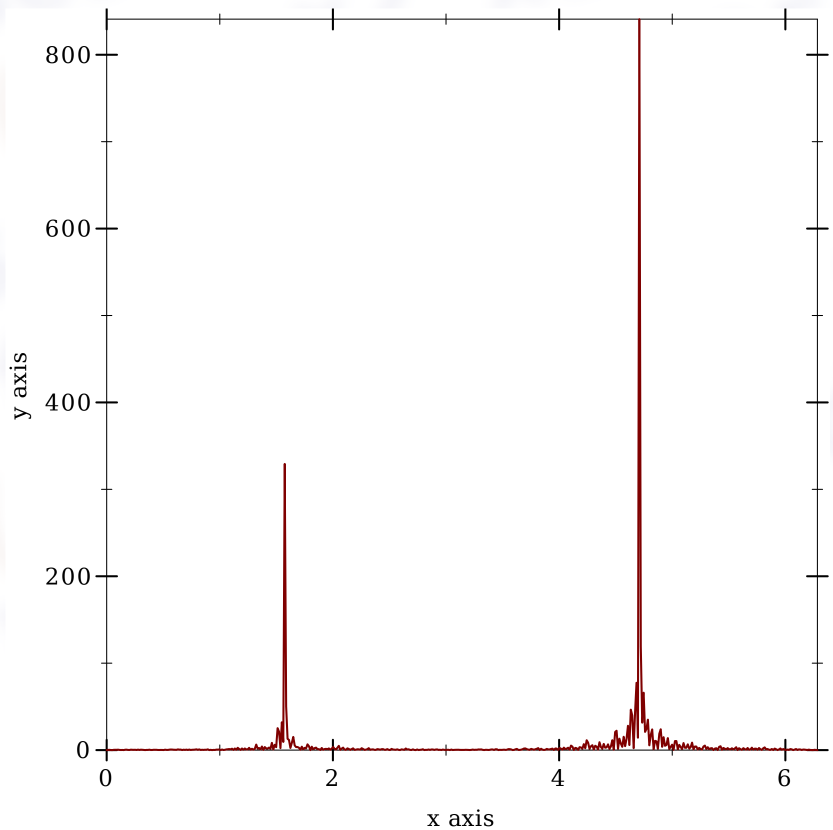
The Badlands: Sine

```
> (plot (function
          (λ (x)
            (define z* (bigfloat->real (bfsin (bf x))))
            (flulp-error (flsin (fl x)) z*))
          0 (* 2 pi)))
```



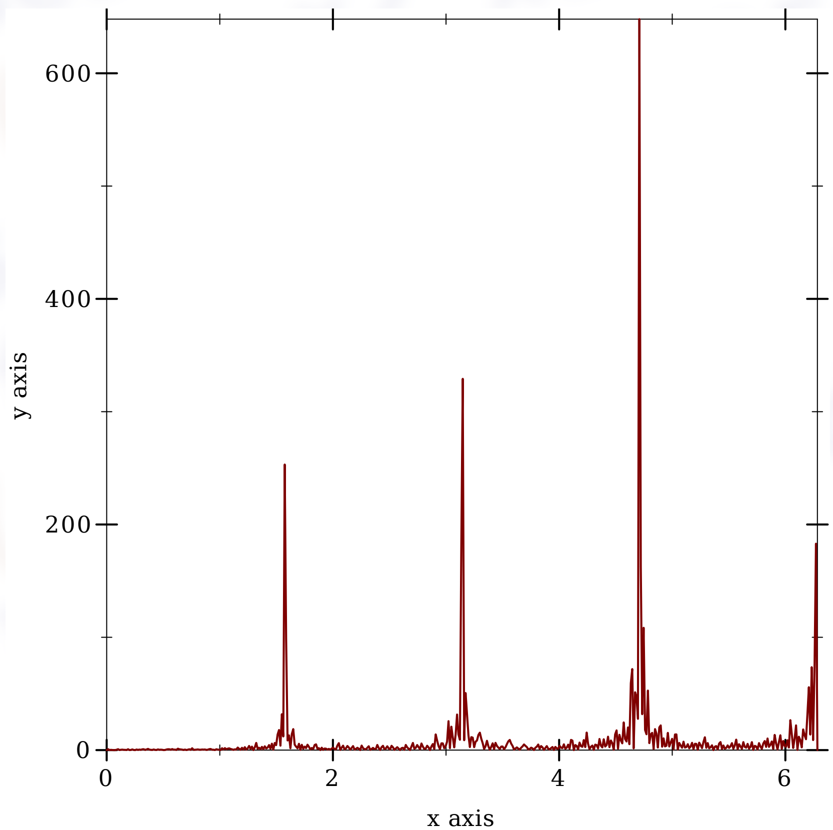
The Badlands: Cosine

```
> (plot (function
         (λ (x)
           (define z* (bigfloat->real (bfcos (bf x))))
           (flulp-error (flcos (fl x)) z*))
         0 (* 2 pi)))
```



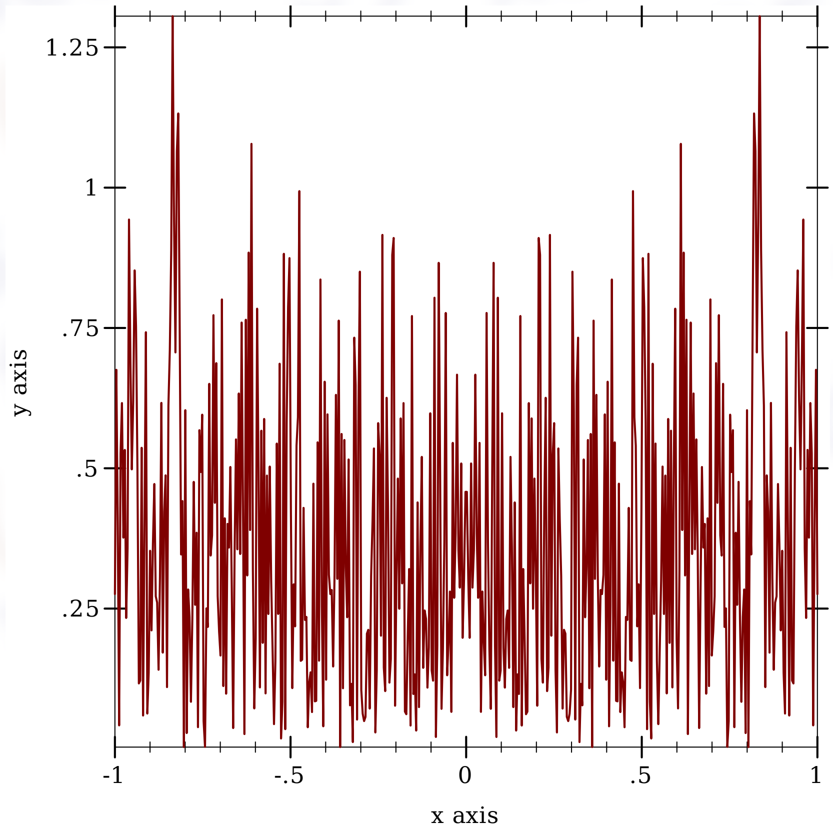
The Badlands: Tangent

```
> (plot (function
         (λ (x)
           (define z* (bigfloat->real (bftan (bf x))))
           (flulp-error (fltan (fl x)) z*))
         0 (* 2 pi)))
```



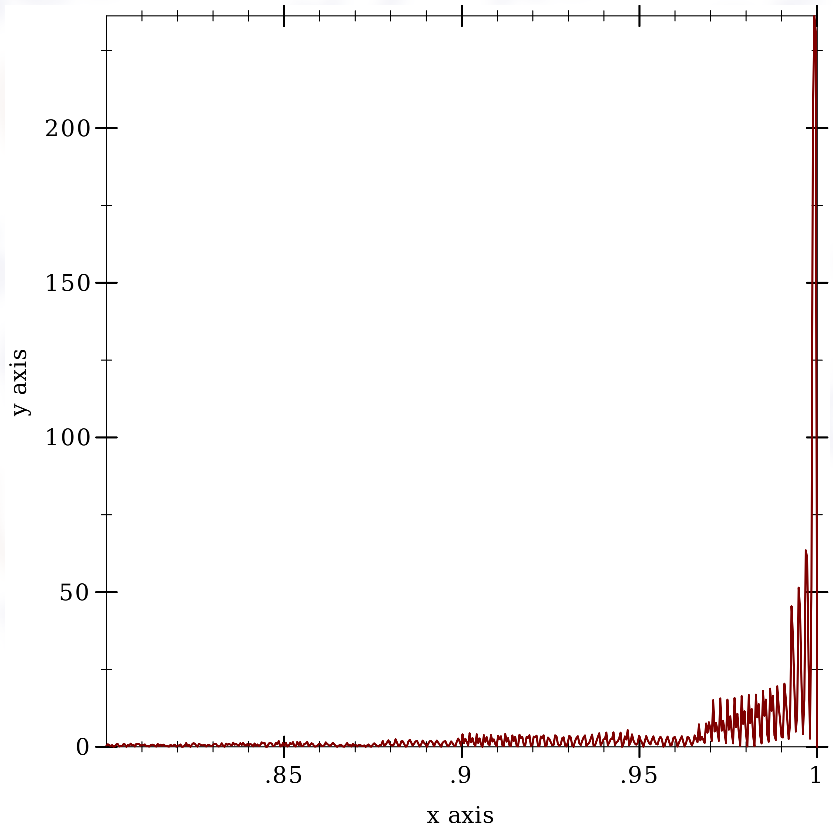
The Badlands: Arcsine

```
> (plot (function
         (λ (x)
           (define z* (bigfloat->real (bfasin (bf x))))
           (flulp-error (flasin (fl x)) z*))
         -1 1))
```



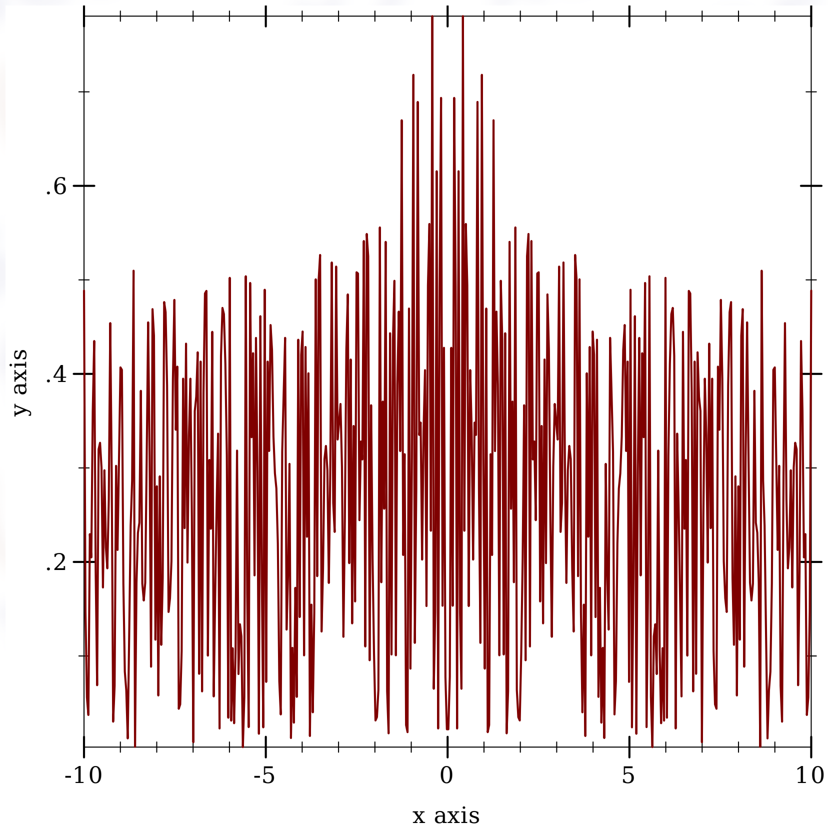
The Badlands: Arccosine

```
> (plot (function
          (λ (x)
            (define z* (bigfloat->real (bfacos (bf x))))
            (flulp-error (flacos (fl x)) z*))
          0.8 1))
```



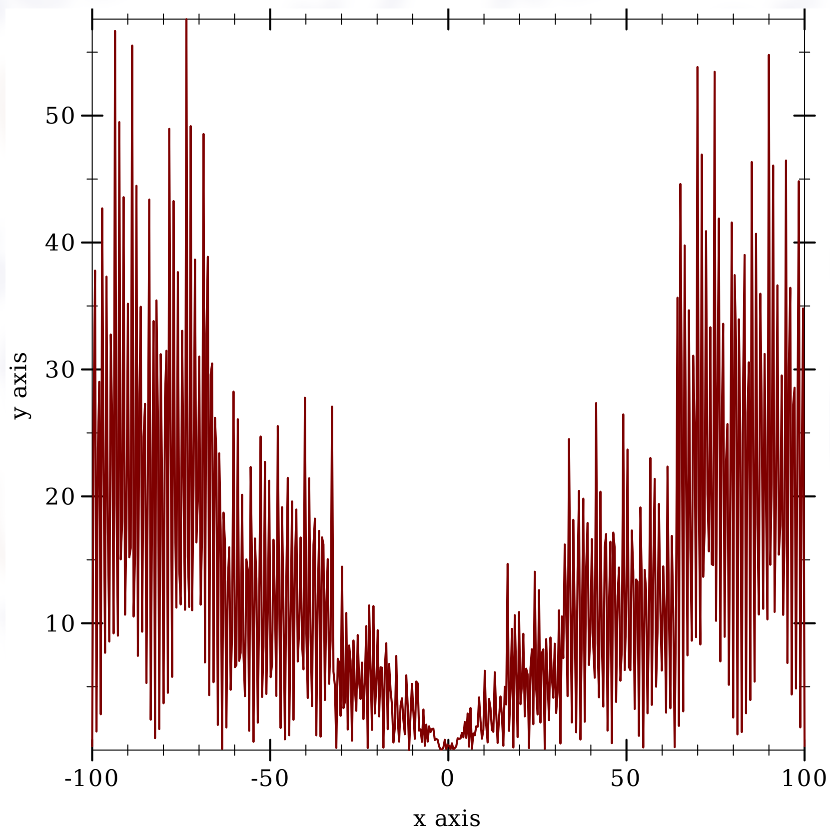
The Badlands: Arctangent

```
> (plot (function
         (λ (x)
           (define z* (bigfloat->real (bfatan (bf x))))
           (flulp-error (flatan (fl x)) z*))
         -10 10))
```



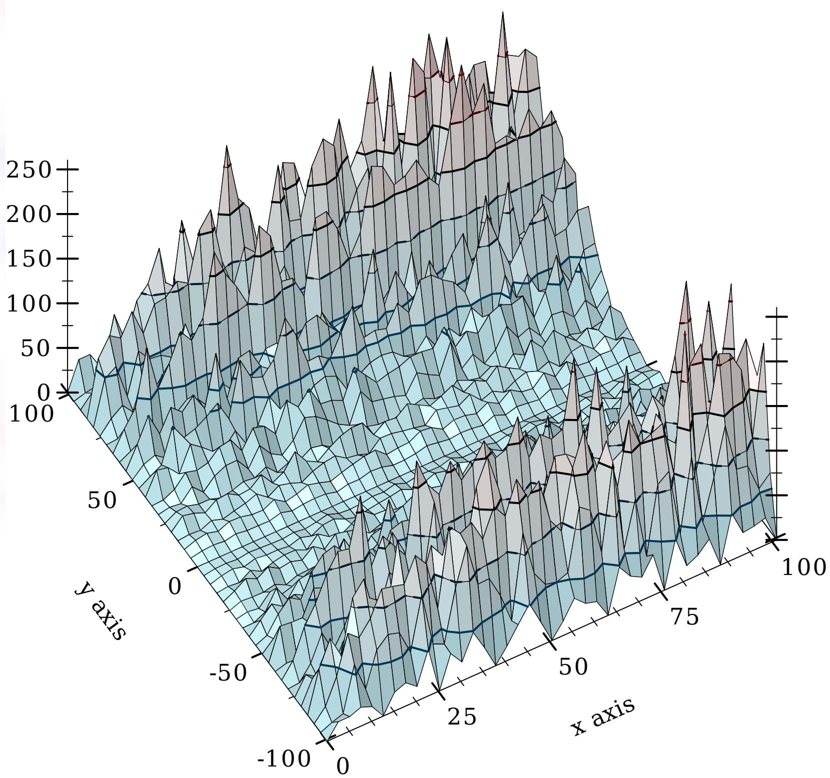
The Badlands: Exponential

```
> (plot (function
         (λ (x)
           (define z* (bigfloat->real (bfexp (bf x))))
           (flulp-error (flexp (fl x)) z*))
         -100 100))
```



The Badlands: Exponential With Base

```
> (plot3d (contour-intervals3d
            (λ (x y)
              (define z* (bigfloat->real
                          (bfexpt (bf x) (bf y))))
              (flulp-error (flexpt (fl x) (fl y)) z*))
            0 101 -101 101))
```



Condition Number

```
(: condition ((Flonum -> Flonum)
              (Flonum -> Flonum)
              -> (Flonum -> Flonum)))
(define ((condition f df) x)
  (abs (/ (* x (df x)) (f x))))


(: condition2d ((Flonum Flonum -> Flonum)
                (Flonum Flonum -> (Values Flonum Flonum))
                -> (Flonum Flonum -> Flonum)))
(define ((condition2d f df) x y)
  (define-values (dx dy) (df x y))
  (define z (f x y))
  (max (abs (/ (* x dx) z))
       (abs (/ (* y dy) z))))
```