_____ / _____ / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> '%2.24f'%(((5**0.5)**2)-5)
'0.00000000000000000888178420'
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e vice-versa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0e - 0$) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

☞ Para você fazer

Interprete os 2 números binários a seguir como $binay16$ inteiros e negativos e ache os valores decimais de

```
1 1 0 1 1 1 0 1 0 0 0 1 0 0 1 0
1 1 1 1 1 0 1 1 0 1 1 1 1 1 0 0
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 8 casas decimais de

```
0 0 1 0 0 0 0 0 1 1 1 1 0 1 1 0 0
1 0 0 1 1 0 1 1 1 1 0 1 1 1 1 0 0
1 0 1 0 0 0 0 0 1 1 0 0 0 1 1 0
0 0 0 0 0 0 1 1 1 1 0 0 0 1 1 1
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> '%2.24f'%(((5**0.5)**2)-5)
'0.000000000000000888178420'
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e vice-versa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0e - 0$) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

☞ Para você fazer

Interprete os 2 números binários a seguir como $binay16$ inteiros e negativos e ache os valores decimais de

```
1 1 1 0 0 1 1 1 1 0 0 0 1 0 0 1
1 1 1 1 1 0 0 0 1 0 0 1 0 1 0 0
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 8 casas decimais de

```
0 0 1 1 0 1 0 0 1 0 0 0 1 1 0 0
0 0 1 0 0 1 0 1 0 0 1 1 0 0 0 1
1 1 1 0 1 1 1 0 0 0 0 0 0 1 1 0
1 0 0 0 0 1 0 0 0 1 1 0 0 0 1 0
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.

_____ / _____ / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> '%2.24f'%(((5**0.5)**2)-5)
'0.000000000000000888178420'
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e vice-versa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0e - 0$) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

☞ Para você fazer

Interprete os 2 números binários a seguir como $binay16$ inteiros e negativos e ache os valores decimais de

```
1 1 0 1 1 0 0 1 0 0 0 1 1 1 1 1
1 1 1 0 1 0 1 1 0 1 0 1 1 1 0 0
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 8 casas decimais de

```
0 1 0 0 1 0 0 0 0 0 0 1 0 0 1 1
1 0 1 0 1 1 0 0 0 1 0 1 0 1 1 0
0 0 1 0 1 0 1 0 1 0 1 1 0 0 0 0
1 0 1 0 0 1 1 1 1 1 1 0 0 1 0 0
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> '%2.24f'%(((5**0.5)**2)-5)
'0.000000000000000888178420'
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ ... & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ ... & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e vice-versa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0e - 0$) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

☞ Para você fazer

Interprete os 2 números binários a seguir como $binay16$ inteiros e negativos e ache os valores decimais de

```
1 1 1 0 0 1 1 1 1 1 1 1 0 1 1 0
1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 8 casas decimais de

```
0 1 0 1 0 0 0 1 1 1 0 0 1 1 1 1
0 0 0 1 1 0 0 1 0 0 0 1 1 0 1 1
0 1 0 0 0 1 1 1 1 1 1 1 1 1 0 1
0 1 1 0 1 0 0 1 1 0 0 0 0 1 1 0
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> '%2.24f'%(((5**0.5)**2)-5)
'0.00000000000000000888178420'
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e vice-versa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0e - 0$) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

☞ Para você fazer

Interprete os 2 números binários a seguir como $binay16$ inteiros e negativos e ache os valores decimais de

```
1 1 1 0 1 0 1 0 0 1 0 1 0 1 1 0
1 1 1 1 0 0 0 0 1 1 0 1 1 1 0 0
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 8 casas decimais de

```
0 0 0 1 0 1 0 0 0 1 0 1 1 0 0 1
0 0 0 0 1 1 0 1 0 1 1 1 1 0 0 1
0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1
1 0 0 0 1 1 0 1 1 0 0 1 1 1 1 0
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.

```
===== 11/06/2022 13:21:19.1 =====E=PLk07
    1  -8942  -1156      .0057831     -.0018883     -.0046616      .0000297
    2  -6263  -1900      .1420898      .0101395 -3084.0000000     -.0000334
    3  -9953  -5284     4.0742188     -.0338745      .0261230     -.0154114
    4  -6154  -4033    23.2343750      .0012465     3.9941406  1414.0000000
    5  -5546  -3876      .0005307      .0001670      .0005269     -.0001714
```