Computação com Precisão Finita

Paulo J. S. Silva

12 de setembro de 2017

1 Introdução

O computador é uma máquina finita, feita a partir de um número finito de objetos e capaz de armazenar e manipular um número finito de dados. Fica então a dúvida: como ele pode armazenar ou fazer contas com números que não admitem representação finita, como os números irracionais? Como se pode calcular o $\sin(\pi)$ se o computador não pode armazenar o π, pelo menos não completamente? Isso não pode ser feito exatamente.

O que se pode fazer então é armazenar uma aproximação de π, uma aproximação muito boa, que será usada no lugar do número verdadeiro. Para a grande maioria das situações isso é bom o suficiente. O objetivo dessa primeira parte do curso é discutir um pouco sobre como o computador guarda os números, que tipo de garantia podemos esperar na qualidade das aproximações feitas e que problemas podem surgir devido a essas aproximações. Isso é especialmente importante quando lembramos que podemos executar milhões, ou até bilhões de operações em sequência, cada uma com um pequeno erro. Esses erros se acumulam? Eles se cancelam?

Primeiros vejamos o que é π para o computador:

```
In [1]: Float64()
```

```
Out[1]: 3.141592653589793
```

Como você pode vê o computador armazena uma boa aproximação do número, acima são mostrados os primeiros 16 dígitos significativos e estão todos corretos. Veremos abaixo porque.

Obs: Note que o código em Julia aceita caracteres UTF-8 que contém todos os símbolos do alfabeto grego, além dos símbolos do nosso alfabeto latino e muitos outros. Isso permite que definamos uma variável usando uma letra grega como nome, o que pode ser útil em código matemático. Para obter isso em uma célula de código experimente digitar "\pi"ou "\alpha"seguido de um TAB. Note também que o Julia já tem o símbolo de π pré-definido para ser uma representação do número π. Na verdade, a Julia associa esse símbolo à versão irracional e calcula a aproximação desejada na hora que ela for necessária. Acima, por exemplo, pedimos uma aproximação do tipo `Float64` que represenda o tipo de precisão dupla, o padrão do computador de hoje. Se você quiser ver π com mais casas de precisão experimente trocar o `Float64` acima por `BigFloat` na célula de código acima e mande executá-la.

2 Erros

Acima falamos que o computador armazena aproximações dos números (pelo menos no caso destes não admitirem representação finita). Ao se fazer uma aproximação comentemos um pequeno *erro*. Vamos definir formalmente esse conceito.

Definição: Seja $\hat{x}$ um valor que desejamos representar (ou calcular) e x uma aproximação de $\hat{x}$. O *erro absoluto de x com respeito a $\hat{x}$* é

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

Já o *erro relativo* é

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

Observe que para definir o erro relativo precisamos que $\hat{x} \neq 0$.

O erro (absoluto ou relativo) mede o quão distante x está do valor que ele quer representar. O erro absoluto é exatamente o módulo da diferença. Já o erro relativo tenta avaliar a proporção do erro com respeito ao valor real. Isso porque errar uma unidade ao se tentar aproximar o número 2 é algo muito grosseiro, mas errar uma unidade ao se tentar calcular o número 2.000.000.000 é bem mais aceitável. Nesse sentido, em muitos casos vamos estar mais interessados no erro relativo do que no erro absoluto. Porém há situações em que o erro absoluto é útil e informativo também.

Um problema com a definição acima é que para se calcular o erro devemos saber qual é o número desejado $\hat{x}$ e em muitos casos não o conhecemos: estamos justamente tentando calculá-lo. Porém não se preocupe muito com isso, em geral podemos estimar o erro apresentando limitantes para o seu valor. Você verá isso durante o curso.

Um outro tipo de erro que pode ser útil é o erro adimensional definido por

$$E_{adm}(\hat{x}) = \frac{|x - \hat{x}|}{L},$$

em que L é uma constante que representa de alguma forma valores típicos esperados. Por exemplo, se desejamos representar a distância entre duas cidades, podemos desejar verificar a precisão comparando com 1km. Já para o cálculo da distância entre os núcleos de átomos já é natural pensar em 1 Angstron (10^{-10}m). Fazendo-se a divisão por essa ordem de grandeza típica podemos então obter uma estimativa razoável da qualidade do erro.

Vejamos agora dois exemplos. Considere que obtivemos $x = 0{,}9273$ para aproximar $\hat{x} = 1$. Quais os erros associados? Temos

$$E_{abs} = |0{,}9273 - 1{,}0| = 0{,}0727, \qquad E_{rel} = \frac{|0{,}9273 - 1{,}0|}{|1{,}0|} = 0{,}0723.$$

Nesse caso, como o valor desejado tem módulo 1, os erros absoluto e relativo coincidem.

Já para $x = 0{,}9273$ para aproximar 0,9 teríamos

$$E_{abs} = |0{,}9273 - 0{,}9| = 0{,}0273, \qquad E_{rel} = \frac{0{,}0273}{0{,}9} = 0{,}0303333333\ldots.$$

Aqui o erro relativo é maior que o absoluto, dando mais peso ao erro porque o número que desejávamos aproximar tem módulo menor do que 1.

2.1 Origem dos erros

Mas de onde podem vir os erros? Podemos destacar pelo menos 4 fontes naturais de erros que enfrentamos no dia-a-dia:

1. Erro de aquisição ou medida: ocorre quando precisamos medir ou estimar algo. Essa é a situação que vocês encontram no laboratório de Física, por exemplo.

2. Erro de representação: imagine que você quer usar um número decimal com um grupo finito de dígitos para representar o fração $\frac{1}{3}$. Como ela é uma dízima periódica isso é impossível de ser feito e o erro será grande ou pequeno de acordo com o número de casas armazenadas.

3. Erro associados a cálculos com precisão finita. Esse erro aparece quando queremos realizar uma operação sobre números já representados e o resultado não pode ser representando. Por exemplo, queremos dividir 1 por 3. Note que muitas vezes desejamos realizar vários, mesmo milhões ou bilhões de cálculos em sequencia, e cada um deles tem o potencial de gerar erros. Como já disse esse é o principal tipo de erro que iremos estudar.

4. Erro associados a algoritmos que aproximam soluções (métodos iterativos). Infelizmente não há fórmula finita para o cálculo exato das soluções de muitos problemas matemáticos. O caso mais clássico é o cômputo de raízes de polinômio de grau maior ou igual a 5. Nesse caso lançamos mão de métodos iterativos que tentam aproximar a solução desejada através de um processo potencialmente infinito. Veremos exemplos disso no curso. Mas não temos como esperar tempo infinito para que o processo atinja o valor desejado. Nesse caso paramos a execução do programa uma vez que uma aproximação aceitável do valor desejado tenha sido obtida e guardamos essa aproximação.

3 Representação de números no computador

Para armazenar números no computador adotou-se um sistema que busca diminuir espaços vazios entre os números representados de forma relativa. Esse sistema é conhecido como *representação de ponto flutuante*. A ideia é guardar uma quantidade fixa de *dígitos significativos* (ignorando possíveis zeros à esquerda que não dizem nada) e um outro número dizendo onde está a vírgula, ou o ponto em inglês e daí o nome, *ponto flutuante*. Mais precisamente um sistema de ponto flutuante é caracterizado basicamente por três quantidades:

1. Uma *base* b. No computador essa base é tipicamente 2 (base binária). Mas nos nossos exemplos em sala iremos usar a base 10 que é mais usual para nós, humanos.

2. A quantidade de números (dígitos na base) armazenados. Os dígitos armazenados são conhecidos como *mantissa* e denotada por m. Para evitar duplicidade de representação é importante definir exatamente a forma da mantissa. Uma escolha comum é considerar que a mantissa é um número que tem o primeiro dígito nulo, depois a vírgula seguida de pelo menos um dígito não nulo. Ou seja a mantissa deve ser um número cujo módulo pertence a $[0{,}1, 1)$.

3. A quantidade mínima e máxima de um inteiro, chamado de *expoente*, que é usado para dizer onde está a vírgula, denotado por e.

Para deixar isso mais claro vamos definir um sistema simples em base decimal e ver que tipo de números podem ser representados.

1. Base 10.

2. A mantissa guarda 4 dígitos.

3. O menor expoente é -99 e o maior 99.

Imagine que queremos representar o número 0,034. Seguindo as regras e escolha descritas acima esse número será representado pela mantissa 0,3400 (notem que o número é menor estrito que 1 e maior ou igual a 0,1) e expoente -1. Ou seja representamos

$$0{,}034 = 0{,}3400 \cdot 10^{-1}.$$

Agora é um bom momento para discutir a regra do formato da mantissa. A regra busca a forçar o primeiro dígito da significativo (não nulo contando a partir da esquerda) a ficar logo depois da vírgula. Isso para garantir que todo número tem representação única. Se não houvesse essa regra poderíamos representar o mesmo 0,034 por

$$0{,}034 = 0{,}0340 \cdot 10^{0} = 0{,}0034 \cdot 10^{1}.$$

A unicidade da representação evita dúvidas e desperdício com múltiplas representações para o mesmo número.

E como seria representado o nosso amigo π? Vamos relembrar o seu valor.

```
In [25]: BigFloat()
```

```
Out[25]: 3.141592653589793238462643383279502884197169399375105820974944592307816406286198
```

A melhor representação que podemos obter é

$$\pi \approx 0{,}3146 \cdot 10^1.$$

Note que em particular o menor número representável em módulo no nosso sistema é $0{,}1000 \cdot 10^{-99}$ e o maior $0{,}9999 \cdot 10^{99}$.

Agora qual é o sistema de ponto flutuante adotado no computador? Quase todas as máquinas modernas implementam o padrão IEEE 754. Ele define dois tipos básicos de números. Números de precisão simples (o float de C), ocupam 32 bits divididos entre 1 bit para o sinal, 8 bits para o expoente e 23 para mantissa. Já a precisão dupla (o double de C) usa um bit para o sinal, 11 para o expoente e 52 para a mantissa totalizando 64 bits.

Em base decimal isso nos dá um número com aproximadamente 15 casas decimais na mantissa e expoente indo de -1022 a 1023. Quem quiser mais informações sobre o padrão IEEE 754 pode consultar esse texto.

Um fato interessante em sistemas de ponto flutuante é que há buracos entre os números representáveis, já que existe um número finito deles. Em particular depois do número 1 (que é representável usando mantissa 0,1 e expoente 1) há um primeiro próximo número representável. O que ocorre se tentarmos somar ao 1 um número tão pequeno que a soma resultante esteja mais perto do 1 do que desse próximo número? Vamos querer que a resposta seja o próprio 1, já que esse é o número representável mais próximo da resposta correta. Ou seja, se u é pequeno vamos querer que o computador devolva como resultado da operação

$$1 + u$$

o próprio 1! Vamos normalmente denotar os resultados calculados pelo computador através do operador fl. Usando essa notação vemos que para u pequeno

$$\mathrm{fl}(1 + u) = 1.$$

Vamos chamar de *unidade de arredondamento*, ou *epsilon da máquina*, denotado por ϵ_{mac}, o menor número para o qual ainda resulta que $\mathrm{fl}(1 + \epsilon_{mac}) > 1$. Isso é, basicamente, a metade da distância entre o 1 e o próximo número representável. Esse número nos dá uma ideia de quantas casas de precisão o nosso sistema tem. Em particular no caso do padrão IEEE 754 temos as unidades de arredondamento:

1. Precisão simples: $\epsilon_{mac} \approx 1{,}19209 \cdot 10^{-7}$.
2. Precisão dupla: $\epsilon_{mac} \approx 2{,}22045 \cdot 10^{-16}$.

O padrão IEEE 754 além de definir esses dois sistemas de ponto flutuante obriga ainda que as operações aritméticas básicas sejam realizadas de modo a garantir que o valor obtido ao final é a melhor representação possível do valor exato. Isso é, dados dois números representáveis x_1 e x_2 o sistema IEEE 754 exige que o computador implemente a sua versão da soma, que vamos representar por $\oplus$, de modo que $x_1 \oplus x_2$ seja o número representável mais próximo de $x_1 + x_2$. Em particular, isso garante que o erro

$$|(x_1 \oplus x_2) - (x_1 + x_2)| \leq \epsilon_{mac}|x_1 + x_2|$$

Ou seja, o erro relativo ao se fazer a operação de soma como implementada seguindo ao padrão IEEE 754 é no máximo ϵ_{mac}. Isso não vale apenas para a operação de soma, vale para todas as operações aritméticas fundamentais que são soma, subtração, multiplicação, divisão e cálculo da raiz quadrada.

3.1 Erros de cancelamento

Quando ficamos sabendo da propriedade descrita acima, isto é que o computador implementando o padrão IEEE 754 é capaz de garantir a execução das operações básicas com erro relativo máximo proporcional ao epsilon da máquina, ficamos com a impressão que essas operações não são capazes de gerar muitas dificuldades numéricas. Afinal de contas, para números de precisão dupla, isso dá impressão que os valores calculados estarão corretos pelo menos até a décima quinta casa. Parece mais do que o suficiente. Porém há um caso, que muitas vezes ignoramos em uma primeira leitura, que pode trazer muitos problemas. O

fenômeno é conhecido como *erro de cancelamento*. Vamos ver primeiro um exemplo em que ele ocorre e depois discutir o que ocorreu.

Considere que queremos calcular $49213 + 31{,}72849244 = 0{,}728$ em um computador com sistema decimal e cinco casas na mantissa. Note que, como todos os números da conta original têm cinco casas, parece que não estamos pedindo nada demais. A primeira operação executada obtém

$$\mathrm{fl}(49213 + 31{,}728) = \mathrm{fl}(49244{,}728) = 49245.$$

Note o resultado final armazenado é tão bom como prometido. Ele tem cinco casas corretas. De fato, o erro relativo é

$$\frac{|49245 - 49244{,}728|}{|49244{,}728|} \approx 5{,}523 \cdot 10^{-6},$$

que é próximo ao epsilon da máquina.

Agora fazemos a operação final, **usando o resultado já calculado**,

$$\mathrm{fl}(49245 - 49244) = \mathrm{fl}(1) = 1.$$

Veja que esse resultado tem quase nenhuma relação com o valor exato que é 0,728. Ele apenas acerta a ordem de grandeza. Mas **não tem nenhum dígito correto**, muito menos os cinco dígitos significativos esperados.

O que ocorreu? Como explicar um resultado tão ruim? O problema está na operação de subtração entre dois números muito parecidos presente na segunda operação. Quando dois números muito parecidos são subtraídos, os dígitos mais significativos "somem"e o resultado final fica limitado no número de dígitos significativos se que pode armazenar. Não há o que fazer. Esse é o caso da subtração $49245 - 49244$. Os quatro dígitos mais importantes são iguais, então apenas o último dígito carrega alguma informação gerando o 1. Os outros quatro dígitos, apesar de corretos, são zeros à esquerda que nem escrevemos pois não servem para nada. Isso estaria perfeito se a conta que gostaríamos de fazer fosse exatamente essa. Mas o 49245 é apenas uma aproximação, pois foi obtido de outra operação. Para obter uma resposta com mais dígitos significativos na subtração seria necessário relembrar justamente o que foi esquecido ao se aproximar o valor exato da primeira conta, que era 49244,728, por 49245. Mas não há como voltar atrás, a informação do 0,728 já foi esquecida e não pode ser recuperada. Já os dígitos mais significativos se cancelam. Daí vem o nome desse fenômeno: *erro de cancelamento*.

Agora se usarmos esse número para novos cálculos não podemos garantir muita precisão no resultado final. Isso porque um dos números envolvidos tem apenas a ordem de grandeza correta e essa precisão muito baixa vai se propagar, destruindo a precisão de novos resultados calculados com base nessa aproximação grosseira.

Desse modo, ao calcularmos valores no computador devemos prestar bastante atenção quando fazemos subtrações entre números potencialmente parecidos (ou soma de números de módulo parecido mas sinais distintos). Muitos problemas numéricos ocorrem quando contas aparentemente inocentes geram números similares que têm que ser subtraídos.

Vamos ver agora vários exemplos de possíveis erros de cancelamento e discutir algumas estratégias para evitá-los.

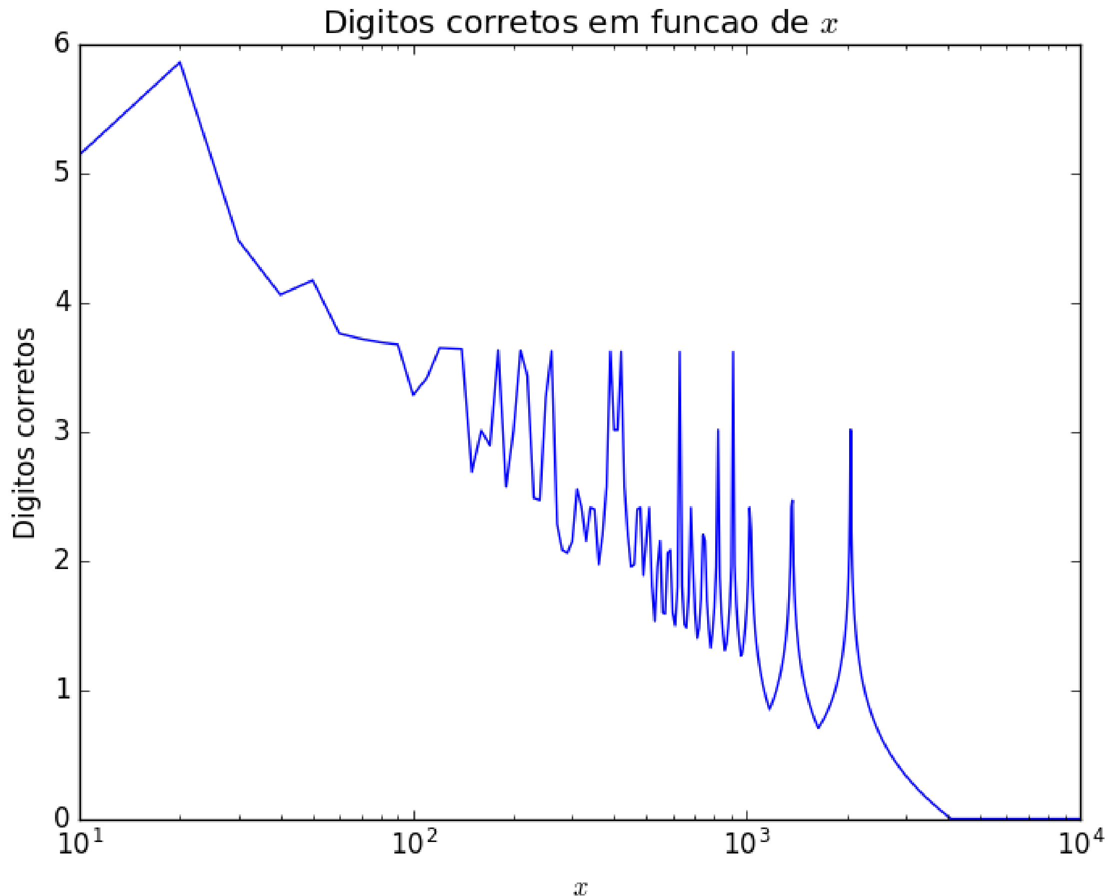
3.1.1 Exemplos de erros de cancelamento

Considere a seguinte expressão $\sqrt{x^2+1} - x$. Quando ela irá gerar erros de cancelamento? Se você pensar um pouco, à medida que x vai para ∞ o valor $x^2 + 1$ fica relativamente mais perecido com o x^2. O 1 se torna irrelevante perante o x^2 que é muito grande. Assim a raiz quadrada desse valor deve ficar muito próxima de $|x|$. Quando fomos subtrair essa raiz quadrada x, que é positivo, teremos erro de cancelamento.

Podemos então prever que $\sqrt{x^2+1} - x$ deve gerar erros de cancelamento para x grande. Para ver isso vamos aproximar o erro relativo comparando números calculados com precisão simples com números calculados com precisão dupla.

```
In [53]: # Edite abaixo para fazer testes com outras expressões se desejar.
```

```julia
# Expressão que se deseja estudar
function expr(x)
    return sqrt(x.^2 + 1) - x
end

# Intervalo de teste [a, b]
a, b = 1.0e+1, 1.0e+4

################## Possivelmente você não quer editar a partir daqui.

# Se prepara para usar rotinas que plotam gráficos.
using PyPlot

# Erro relativo
function erro_rel(x, xh)
    return abs(x - xh) ./ abs(xh)
end

# Calcula valores das expressões com precisão simples e dupla para x crescente
x = linspace(a, b, 1000)
expr_dupla = expr(x)
x_simples = map(Float32, x)
expr_simples = expr(x_simples)

# Apresenta o gráfico dos erros relativos.
# Lembre que quando o erro é 1 não há mais nenhum dígito significativo.
log_errorel = log10(erro_rel(expr_simples, expr_dupla))
semilogx(x, -log_errorel)

# Para deixar o gráfico bonito
title("Digitos corretos em funcao de \$x\$")
ylabel("Digitos corretos")
xlabel("\$x\$")
```




```
Out[53]: PyObject <matplotlib.text.Text object at 0x7f3e593e0d90>
```

Como você pode ver, a precisão começa razoável. Há mais de 5 casas significativas. O número de casas significativas cai rapidamente chegando a 0 antes de $x = 10^4$.

Será que é possível evitar esse erro? Será que é possível re-escrever a expressão de modo a evitar o problema para x grande? A resposta é sim, veja:

$$(\sqrt{x^2 + 1} - x)(\sqrt{x^2 + 1} + x) = x^2 + 1 - x^2 = 1.$$

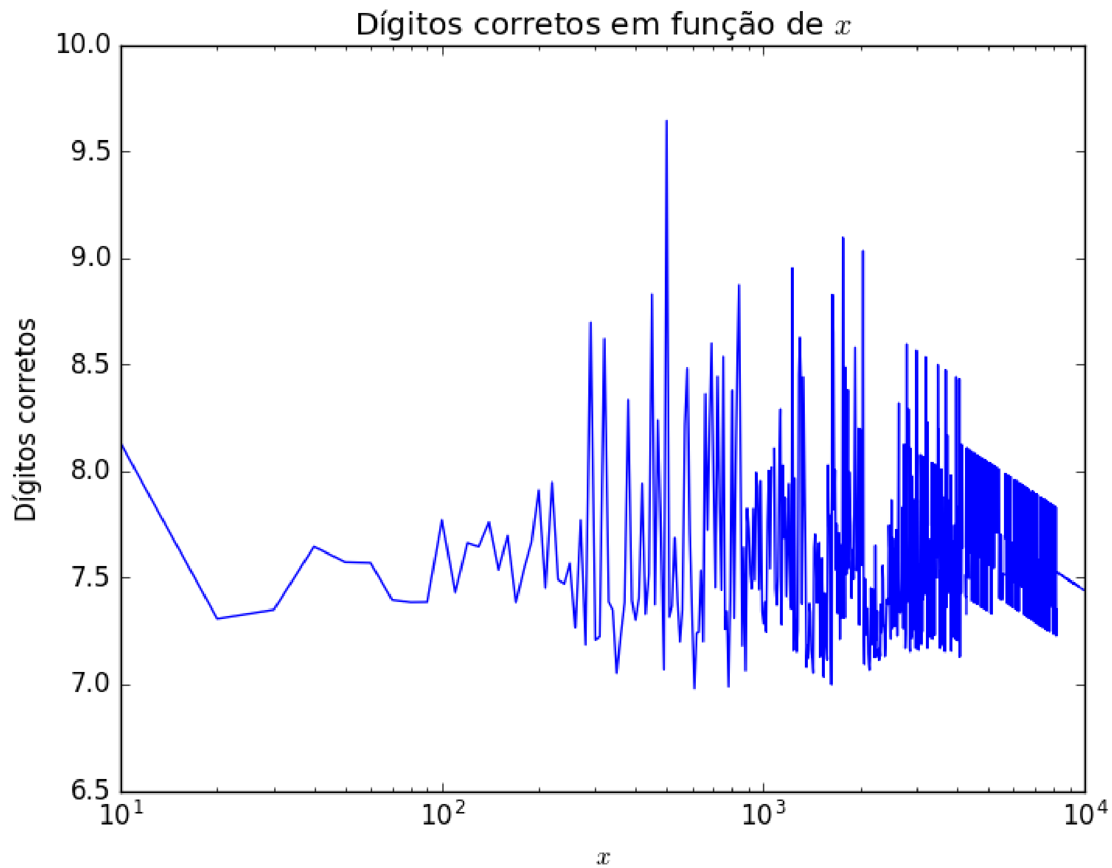
Ou seja,

$$\sqrt{x^2 + 1} - x = \frac{1}{\sqrt{x^2 + 1} + x}.$$

Essa última expressão não tem erros de cancelamento quando x é grande, já que não ocorre subtração de valores próximos. Note o que ocorre ao usarmos essa expressão para o cômputo da fórmula.

```
In [54]: # Versão alternativa que evita erros de cancelamento. Edite-a se quiser fazer testes.
         function expr_alt(x)
             return 1 ./ (sqrt(x.^2 + 1) + x)
         end

         ################### Possivelmente você não quer editar a partir daqui.
```

```
# Calcula a expressão pela fórmula alternativa
expr_simples_alt = expr_alt(x_simples)

# Apresenta o gráfico dos erros, note que quando o erro é 1 não há mais nenhum dígito signific
log_errorel = log10(erro_rel(expr_simples_alt, expr_dupla))
semilogx(x, -log_errorel)

# Para deixar o gráfico bonito
title("Dígitos corretos em função de \$x\$")
ylabel("Dígitos corretos")
xlabel("\$x\$")
```




```
Out[54]: PyObject <matplotlib.text.Text object at 0x7f3e592612d0>
```

Veja como o erro relativo se mantém na ordem do epsilon da máquina para a precisão simples, ou seja 10^{-8}.

Os exemplos abaixo também apresentam erros de cancelamento para alguns valores de x. Identifique esses valores e apresente uma fórmula alternativa que evita o problema. Pode ser interessante aproveitar o código acima e estudar o erro para *ver* se a sua solução está correta.

1. $\sqrt{1+x} - 1$.
2. $\log x - \log y$.
3. $(1 - \cos x)/\sin x$.

3.1.2 Um exemplo mais sofisticado

Um exemplo mais sofisticado aparece quando resolvemos equações do segundo grau. Nesse caso sabemos que as raízes desejadas podem ser obtidas através da fórmula de Báskara. Se queremos as raízes de $ax^2 + bx + c = 0$, calculamos

$$\Delta = b^2 - 4ac, \qquad x = \frac{-b \pm \sqrt{\Delta}}{2a}.$$

E a implementação direta dessa formula é dada abaixo.

In [55]:
```
function raizes(a, b, c)
     = b^2 - 4*a*c
    if  < 0
        printlm("Delta negativo!")
    end
    return (-b + sqrt())/(2*a), (-b - sqrt())/(2*a)
end

# (x - 1.5)(x - 10) = x^2 - 11.5x + 15
raizes(1, -11.5, 15)
```

Out[55]:
```
(10.0,1.5)
```

Problema resolvido. Parece que não há mais nada para fazer.

Mas se pensarmos um pouco é possível antecipar algumas situações em que a formula de Báskara pode sofrer de erros de cancelamento. Ela ainda é simples o suficiente para permitir alguma análise direta.

Observemos inicialmente que há duas somas, uma para achar o delta seguida de outra para achar as raízes. Infelizmente não se conhece uma forma de evitar o possível erro de cancelamento que pode surgir na fórmula do delta. Ele está associado a delta próximo de zero, ou seja $4ac$ negativo e com valor próximo a b^2. Vamos ver o que podemos fazer com a fórmula das raízes,

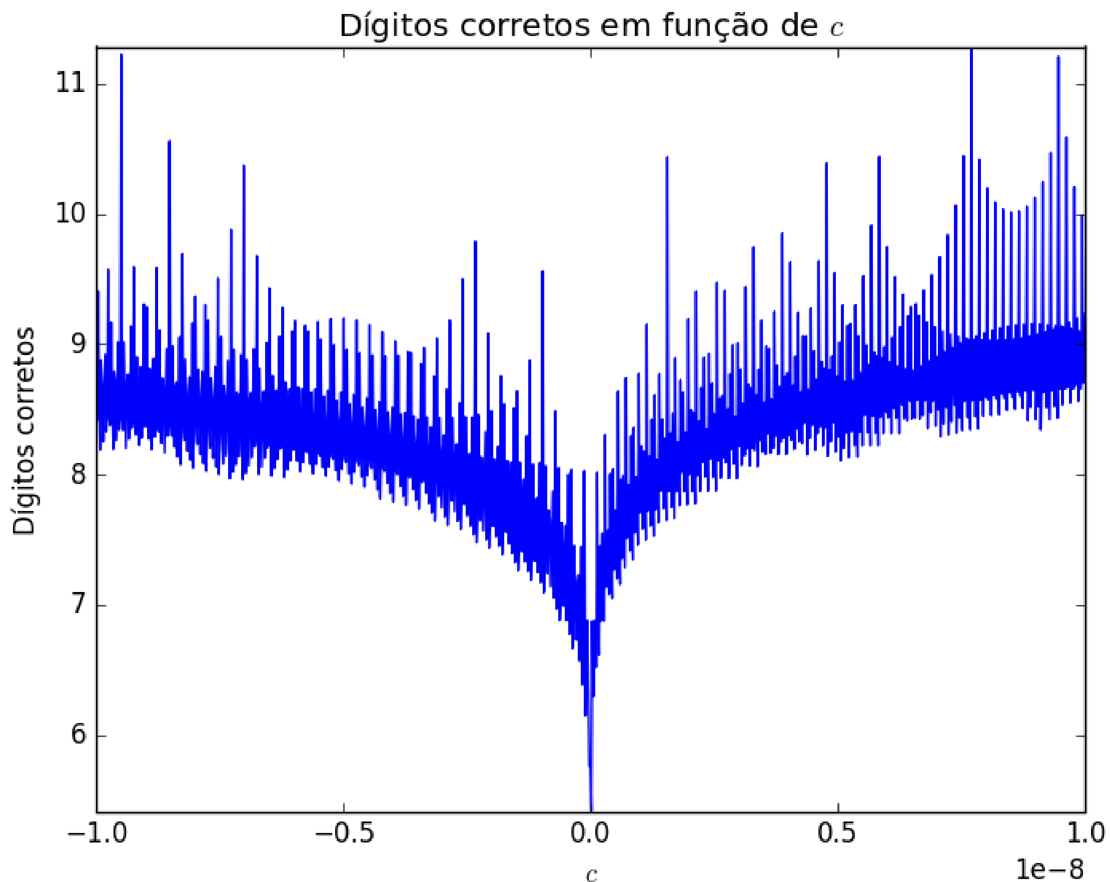
$$x = \frac{-b \pm \sqrt{\Delta}}{2a}.$$

Nela o valor de $-b$ será somando com valores positivos e negativos, ou seja necessariamente em um dos casos não há erro de cancelamento, pois os sinais serão iguais. Já quando $-b$ é positivo um possível erro de cancelamento ocorre quando calculamos $-b - \sqrt{\Delta}$. Caso $-b$ seja negativo a dificuldade pode ocorrer quando computamos $-b + \sqrt{\Delta}$. Além disso o cancelamento ocorre quando o $-b$ e $\sqrt{\Delta}$ tem módulos muito próximos.

Vamos analisar com cuidado um caso particular. Inicialmente, vamos fixar $a = 1$, isso sempre pode ser feito dividindo a equação original por a. Vamos também supor que $b = -1$, assim $-b = 1$. Nesse caso a fórmula da raiz associada ao à situação de cancelamento é $1 - \sqrt{1 - 4c}$, que terá problemas para c pequeno. Vamos ver se isso de fato ocorre. Para isso vamos usar o zero calculado com números `BigFloat` para comparação. O Julia permite que criemos números com qualquer precisão pré-definida usando esse tipo. O padrão é usar 256 bits de precisão, o que dá quatro vezes a precisão dupla que estamos mais acostumados. Se você quiser ainda mais bits de precisão basta usar a função `set_bigfloat_precision`. A desvantagem desse tipo de número é que as operações tornam-se muito mais lentas do que operações feitas com números do padrão IEEE 754, já que o `BigFloat` tem implementação feita por software.

In [56]:
```
# Usa BigFloat para obter solução de altíssima qualidade e depois converte para Float64.
function raizes_big(a, b, c)
    a, b, c = BigFloat(a), BigFloat(b), BigFloat(c)
    r1, r2 = raizes(a, b, c)
    return float(r1), float(r2)
end
```

```
# Coeficientes que definem o polinômio
a = 1.0
b = -1.0
pequeno = 1.0e-8
npontos = 1000
cs = linspace(-pequeno, pequeno, npontos)

# Calcula as raízes de polinomios e guarda os resultados para comparar.
raizes_double = Float64[]   # Raízes calculadas usando precisão double
raizes_bigfloat = Float64[] # Raízes calculadas usando BigFloat
for c in cs
    push!(raizes_double, minimum(raizes(a, b, c)))
    push!(raizes_bigfloat, minimum(raizes_big(a, b, c)))
end

# Apresenta o gráfico de -log_10 do erro relativo.
# Ou seja, do número de casas decimais corretas.
log_errorel = log10(erro_rel(raizes_double, raizes_bigfloat))
plot(cs, -log_errorel)

# Para deixar o gráfico bonito
axis("tight")
title("Dígitos corretos em função de \$c\$")
ylabel("Dígitos corretos")
xlabel("\$c\$")
```



```
Out[56]: PyObject <matplotlib.text.Text object at 0x7f3e58f7fa50>
```

Uma bela figura mostrando que a precisão cai com c próximo de zero, chegando a ter no mínimo quase 5 casas corretas apenas.

A pergunta importante é: como evitar isso? De fato se quiséssemos calcular a raiz maior, próximo de 1, não teríamos problema. Veja isso mudando o sinal da comparação para escolha da raiz no programa acima (troque `minimum` por `maximum`). A ideia agora é usar a raiz boa para estimar a outra. Como fazer isso? Lembremos que

$$x^2 + bx + c = (x - r_1)(x - r_2) = x^2 - (r_1 + r_2)x + r_1 r_2,$$

em que r_1 e r_2 denotam as raízes. Portanto se conhecemos uma raiz, digamos r_1, podemos calcular a outra pela expressão

$$r_2 = \frac{c}{r_1}$$

que não envolve nenhuma soma ou subtração, logo não há erro de cancelamento.

Vamos usar esse fato em uma versão alternativa para o cálculo de raízes.

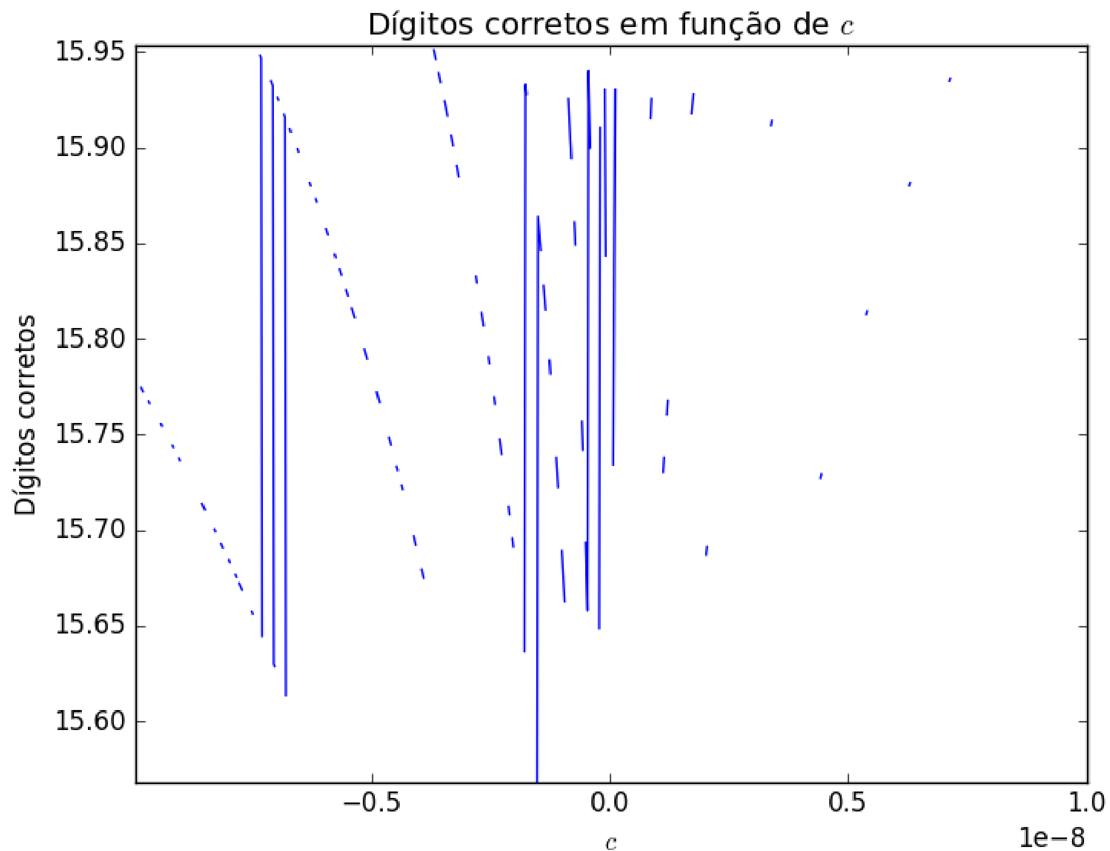
```
In [60]: function raizes(a, b, c)
             b /= a
             c /= a
             a = 1

              = b^2 - 4*c
             if  < 0
                 printlm("Delta negativo!")
             end

             if -b > 0
                 r1 = (-b + sqrt())/2
             else
                 r1 = (-b - sqrt())/2
             end
             r2 = c/r1
             return r1, r2
         end
```

```
Out[60]: raizes (generic function with 1 method)
```

Repetindo o teste acima.

```
In [61]: # Recalcula as aproximações em precisão dupla com fórmula sem erro
         # de cancelamento.
         raizes_double = Float64[]
         for c in cs
             push!(raizes_double, minimum(raizes(a, b, c)))
         end

         # Apresenta o gráfico de -log_10 do erro relativo.
         log_errorel = log10(erro_rel(raizes_double, raizes_bigfloat))
         plot(cs, -log_errorel)

         # Para deixar o gráfico bonito
         axis("tight")
```

```
title("Dígitos corretos em função de \$c\$")
ylabel("Dígitos corretos")
xlabel("\$c\$")
```




```
Out[61]: PyObject <matplotlib.text.Text object at 0x7f3e58cd05d0>
```

Note como a precisão se mantém constante, entre 15 e 16 casas decimais, que é tudo o que pode se esperar de cálculos em precisão dupla. O problema, pelo menos nesse caso foi completamente sanado.

3.2 Misturando números de ordem diferente

Outra situação em que ocorre a perda de dígitos significativos em operações de soma/subtração é quando combinamos números com ordens de grandeza diferentes. Um caso radical disso é quando tentamos somar a um número outro valor de módulo menor que o épsilon da máquina vezes o módulo do número. Nesse caso, não importa o quão complicado seja o número menor, o o resultado vai simplesmente repetir o de maior módulo. Isso vem diretamente da forma de representação de números de ponto flutuante e da definição do épsilon da máquina. Veja:

```
In [33]: # Pede pro Julia o valor pequeno como o eps_mac com relação a aproximação de pi.
         eps_pi = eps(Float64(pi))

         # Tenta somar metade desse valor com a aproximação pi (para garantir que nao
         # arredonda para cima) e compara com a aproximação inicial.
         Float64(pi) + 0.5*eps_pi == Float64(pi)
```

Out[33]: true

Se isso ocorrer uma única vez não há grande problema, a resposta obtida é uma ótima aproximação do valor real. Mas isso pode ser um problema se queremos somar um número grande a vários valores pequenos. Nesse caso os dígitos menos significativos dos números pequenos vão sendo esquecidos durante a soma com o grande a cada soma. Já se os números pequenos fossem somados juntos poderia ocorrer de eles todos combinados terem um valor mais representativo com relação ao valor maior.

Para deixar isso mais claro vamos mostrar um exemplo. Sabemos que a somatória

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}.$$

Podemos estar interessados em verificar isso experimentalmente no computador fazendo uma soma parcial, mas com grande número de termos. Isto é feito na forma mais natural pela rotina abaixo.

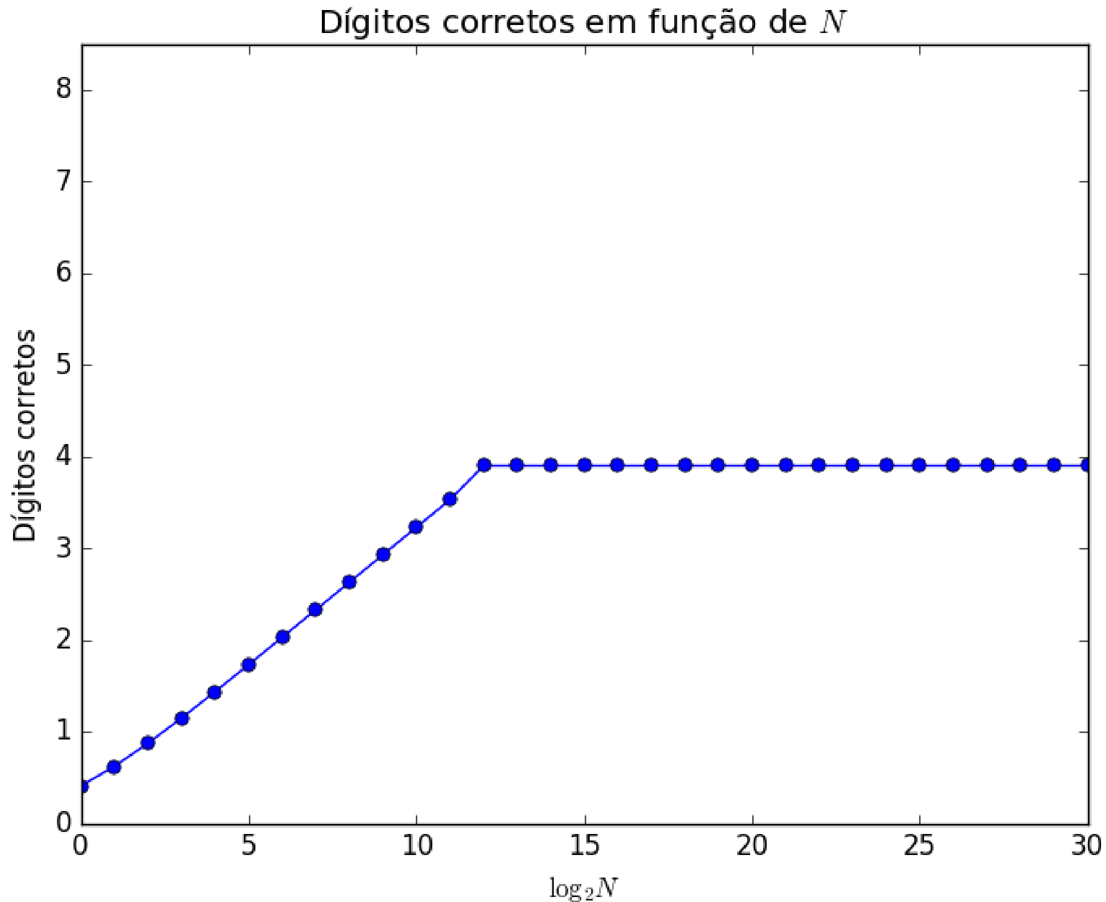
In [36]:
```
function soma_crescente(N)
    soma = 0.0f0
    for k = 1:N
        soma += 1.0f0/(k*k)
    end
    return soma
end
```

Out[36]: soma_crescente (generic function with 1 method)

Obs: Note que no código acima a variável `soma` é inicializada com `0.0f0` que é o 0 de precisão simples. Usamos precisão simples para ver os problemas mais facilmente. De uma maneira geral, para Julia, uma constante numérica do tipo `1.5` é um número de precisão dupla. Ou seja, para Julia, os números são em geral de precisão dupla. Para forçar a criação de números com precisão simples precisamos usar a função `Float32` ou deixar claro que a constante é desse tipo com a letra `f` em uma representação em notação científica do número. Se no lugar de `f` aparece o usual `e` a constante seria intepretada como um `Float64` ou seja um número de precisão dupla.

Podemos então comparar qual a precisão que conseguimos alcançar com essa rotina com um código simples.

In [58]:
```
# Calcula os erros relativos para valores de N como potencias de 2 de 1 a 2^30.
N = 1
erros = Float32[]
Ns = Int[]
for i = 0:30
    push!(Ns, N)
    push!(erros, erro_rel(soma_crescente(N), pi*pi/6))
    N *= 2
end

# Apresenta o gráfico de -log_10 do erro relativo.
log_errorel = log10(erros)
plot(log2(Ns), -log_errorel, "o-")

# Para deixar o gráfico bonito
axis([0, 30, 0, 8.5])
title("Dígitos corretos em função de \$N\$")
ylabel("Dígitos corretos")
xlabel("\$ \\log_2 N \$")
```



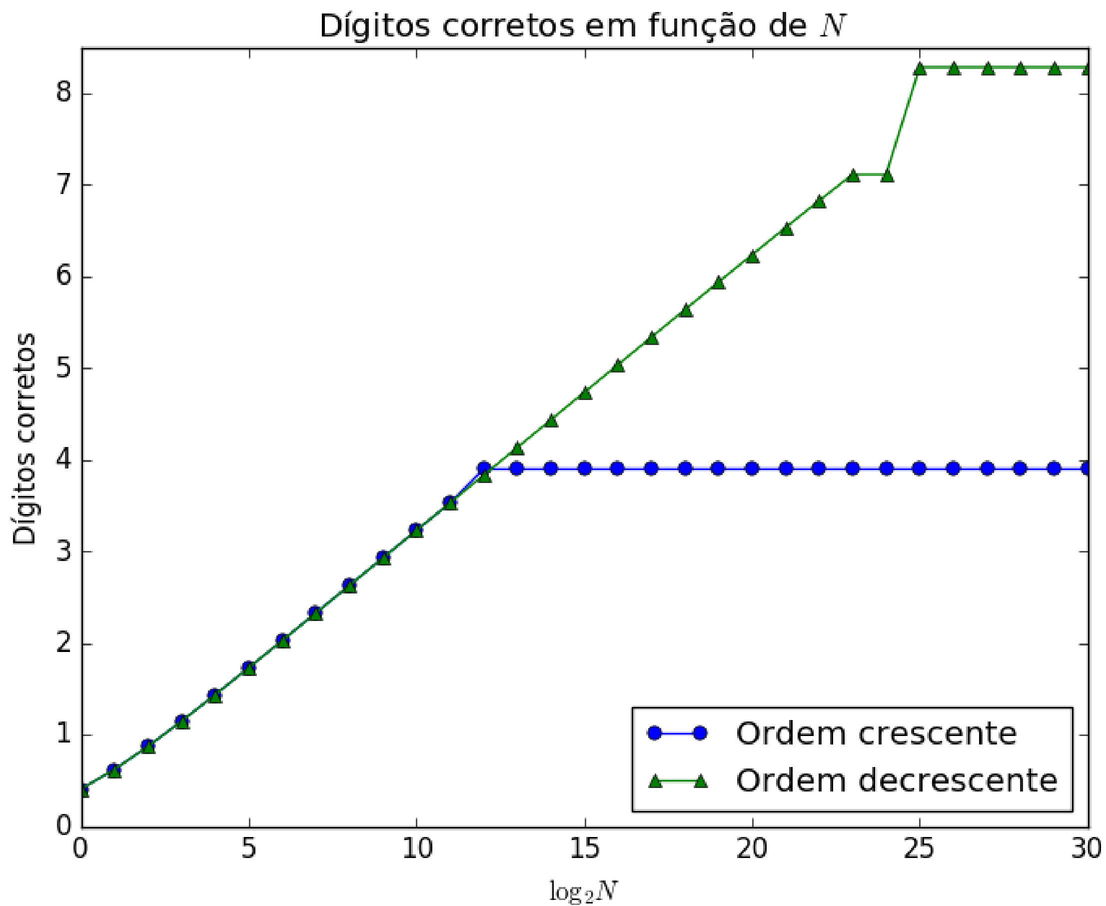

```
Out[58]: PyObject <matplotlib.text.Text object at 0x7f3e58e1e610>
```

Note que nessas somas, quando k é grande, então $1/k^2$ é muito pequeno em relação a parte inicial da soma já calculada, que iniciou em 1 e cresce. Assim, a partir de um certo ponto os valores $1/k^2$ não importam mais. Com isso você pode ver que a precisão atingida com números de precisão simples chega apenas a 4 casas, ao invés das 8 casas esperadas.

Vamos agora ver o que ocorre se fizermos a soma do menor número para o maior.

```
In [59]: # Apresenta o gráfico original de -log_10 do erro com soma cresence.
         PyPlot.plot(log2(Ns), -log_errorel, "o-", label="Ordem crescente")

         # Define a nova versão agora somando do menor para o maior.
         function soma_decrescente(N)
             soma = 0.0f0
             for k = N:-1:1
                 soma += 1.0f0/(k*k)
             end
             return soma
         end

         # Calcula os erros relativos para valores de N como potencias de 2 de 1 a 2^30.
         N = 1
```

```
erros_dec = Float32[]
for i = 0:30
    push!(erros_dec, erro_rel(soma_decrescente(N), pi*pi/6))
    N *= 2
end

log_errorel = log10(erros_dec)
plot(log2(Ns), -log_errorel, "^-", label="Ordem decrescente")

# Para deixar o gráfico bonito
legend(loc="lower right")
axis([0, 30, 0, 8.5])
title("Dígitos corretos em função de \$N\$")
ylabel("Dígitos corretos")
xlabel("\$ \\log_2 N \$")
```




```
Out[59]: PyObject <matplotlib.text.Text object at 0x7f3e58dd1590>
```

Observe que seguindo a ordem decrescente a precisão máxima para números de precisão simples, de 8 casas, é atingida.