Representação de número de ponto flutuante

As representações de ponto flutuante variam de máquina para máquina, como sugeri. Felizmente, um deles é de longe o mais comum atualmente: o padrão IEEE-754. Esse padrão é predominante o suficiente para que valha a pena examiná-lo em profundidade; há boas chances de você conseguir usar essas informações em sua plataforma (procure ieee754.h).

Um float IEEE-754 (4 bytes) ou duplo (8 bytes) tem três componentes (há também um formato análogo de precisão estendida de 96 bits no IEEE-854): um bit de sinal informando se o número é positivo ou negativo, um expoente dando sua ordem de grandeza e uma mantissa especificando os dígitos reais do número. Usando flutuadores de precisão simples como exemplo, aqui está o layout dos bits:

```
seeeeeeemmmmmmmmmmmmmmmmmmmmmmm significado
31 0 bit #

s = bit de sinal, e = expoente, m = mantissa
```

O valor do número é a mantissa vezes 2^x, onde x é o expoente. Observe que estamos lidando com frações binárias, de modo que 0,1 (o bit de mantissa mais à esquerda) significa 1/2 (os valores posicionais à direita da vírgula decimal são 2^-1, 2^-2, etc., assim como nós tem 10^-1, 10^-2, etc. em decimal).

Observe ainda que há um problema potencial com o armazenamento de uma mantissa e de um expoente: 2x10^-1 = 0,2x10^0 = 0,02x10^1 e assim por diante. Isso corresponderia a muitos padrões de bits diferentes representando a mesma quantidade, o que seria um enorme desperdício (provavelmente também tornaria mais difícil e mais lento implementar operações matemáticas em hardware). Este problema é contornado interpretando toda a mantissa como estando à direita da vírgula decimal, com um "1" implícito sempre presente à esquerda da vírgula decimal. Vou me referir a isso como uma representação “1.m”. "Mas espere!" você chora. "E se eu não quiser um 1 aí?" Pense da seguinte forma: imagine escrever um número real em binário. A menos que seja zero, deve haver 1 em algum lugar. Mude seu ponto decimal para logo após o primeiro 1 e não se preocupe em armazenar esse 1, pois sabemos que está sempre implícito que ele esteja lá. Agora tudo que você precisa fazer é definir o expoente corretamente para reproduzir a quantidade original.

Mas e se o número *for* zero? As boas pessoas do comitê de padrões do IEEE resolvem isso fazendo do zero um caso especial: se todo bit for zero (sendo o bit de sinal irrelevante), então o número é considerado zero.

Oh céus. Pare um momento para pensar sobre a última frase. Agora parece que não temos como representar o humilde 1,0, que teria que ser 1,0x2^0 (um expoente de zero, vezes o implícito)! A saída para isso é que a interpretação dos bits do expoente também não é direta. O expoente de um ponto flutuante de precisão simples é codificado como "shift-127", o que significa que o expoente real é eeeeeeee menos 127. Felizmente, podemos obter um expoente zero armazenando 127 (0x7f). É claro que simplesmente mudar a amplitude do expoente não é uma panacéia; algo ainda precisa ceder em algum lugar. Em vez disso, cedemos no extremo inferior do espectro de magnitudes representáveis, que deveria ser 2^-127. Devido ao deslocamento 127, o expoente mais baixo possível é na verdade -126 (1 - 127). Parece-me sensato desistir do menor expoente em vez de desistir da capacidade de representar 1 ou zero!

Zero não é o único ponto flutuante de "caso especial". Existem também representações para o infinito positivo e negativo e para um valor que não é um número (NaN), para resultados que não fazem sentido (por exemplo, números não reais ou o resultado de uma operação como infinito vezes zero) . Como isso funciona? Um número é infinito se cada bit do expoente for definido (sim, perdemos outro), e é NaN se cada bit do expoente for definido mais quaisquer bits de mantissa forem definidos. O bit de sinal ainda distingue +/-inf e +/-NaN.

Para revisar, aqui estão alguns exemplos de representações de ponto flutuante:

```
0 0x00000000
1,0 0x3f800000
0,5
0x3f000000 3 0x40400000
+inf 0x7f800000
-inf 0xff800000
+NaN 0x7fc00000 ou 0x7ff00000
em geral: número = (sinal? -1:1) * 2^(expoen
```

Como programador, é importante conhecer certas características da sua representação FP. Eles estão listados abaixo, com valores de exemplo para números de ponto flutuante IEEE de precisão simples e dupla:

Propriedade	**Valor para float**	**Valor para o dobro**
Maior número representável	3.402823466e+38	1.7976931348623157e+308
Menor número sem perder precisão	1.175494351e-38	2.2250738585072014e-308
Menor número representável (*)	1.401298464e-45	5e-324
Pedaços de mantissa	23	52
Bits expoentes	8	11
Épsilon(**)	1.1929093e-7	2.220446049250313e-16

Observe que todos os números no texto deste artigo assumem flutuadores de precisão simples; duplos estão incluídos acima para fins de comparação e referência.

(*)
Só para tornar a vida interessante, aqui temos mais um caso especial. Acontece que se você definir os bits do expoente como zero, poderá representar números diferentes de zero definindo os bits da mantissa. Contanto que tenhamos um 1 inicial implícito, o menor número que podemos obter é claramente 2 ^ -126, portanto, para obter esses valores mais baixos, abrimos uma exceção. A interpretação “1.m” desaparece e a magnitude do número é determinada apenas pelas posições dos bits; se você deslocar a mantissa para a direita, o expoente aparente mudará (experimente!). Pode ajudar a esclarecer a questão apontar que 1,401298464e-45 = 2^(-126-23), em outras palavras, o menor expoente menos o número de bits de mantissa.

No entanto, como impliquei na tabela acima, ao usar esses números extrapequenos você sacrifica a precisão. Quando não há 1 implícito, todos os bits à esquerda do bit mais baixo são zeros à esquerda, o que não adiciona nenhuma informação a um número (como você sabe, você pode escrever zeros à esquerda de qualquer número o dia todo, se quiser) . Portanto, o menor número representável absoluto (1.401298464e-45, com apenas o bit mais baixo do conjunto de palavras FP) tem um mero bit de precisão terrível!

(**)
Epsilon é o menor x tal que 1+x > 1. É o valor posicional do bit menos significativo quando o expoente é zero (ou seja, armazenado como 0x7f).

III. Programação FP Eficaz

A programação numérica é uma área enorme; se você precisar desenvolver algoritmos numéricos sofisticados, este artigo não será suficiente. Perguntar como calcular com exatidão e precisão ideais é como perguntar como escrever o programa mais rápido ou perguntar como cada peça de software deve ser projetada – a resposta depende da aplicação e pode exigir um livro ou dois para ser comunicada. Aqui tentarei apenas cobrir o que acho que todo programador deveria saber.

Igualdade

Primeiro, vamos abordar a incômoda questão da igualdade: por que é tão difícil saber quando dois carros alegóricos são iguais? Em certo sentido, não é tão difícil; o operador == irá, de fato, informar se dois pontos flutuantes são exatamente iguais (isto é, correspondem bit por bit). Você concordará, entretanto, que geralmente faz pouco sentido comparar bits quando alguns deles podem estar incorretos de qualquer maneira, e essa é a situação que temos com a precisão limitada dos pontos flutuantes. Os resultados precisam ser arredondados para caber em uma palavra finita e, se a CPU e/ou software não arredondar conforme o esperado, seus testes de igualdade poderão falhar.

Pior ainda, muitas vezes não é a imprecisão inerente dos carros alegóricos que incomoda você, mas o fato de que muitas operações comumente feitas em carros alegóricos são imprecisas. Por exemplo, as funções trigonométricas padrão da biblioteca C (sin, cos, etc.) são implementadas como aproximações polinomiais. Pode ser demais esperar que cada bit do cosseno de pi/2 seja 0.

Portanto, a questão da igualdade levanta outra questão: "O que *você* quer dizer com igualdade?" Para a maioria das pessoas, igualdade significa “suficientemente próximo”. Com esse espírito, os programadores geralmente aprendem a testar a igualdade definindo uma pequena distância como “suficientemente próxima” e verificando se dois números estão tão próximos. É mais ou menos assim:

```
1  #define EPSILON 1.0e-7
2
3  #define flt_equals(a, b) (fabs((a)-(b))
```

As pessoas costumam chamar essa distância de EPSILON, embora não seja o épsilon da representação do FP. Usarei EPSILON (todas em maiúsculas) para me referir a essa constante e épsilon (minúsculas) para me referir ao épsilon real dos números FP.

Essa técnica às vezes funciona, por isso pegou e se tornou idiomática. Na realidade este método pode ser muito ruim, e você deve estar ciente se ele é apropriado para sua aplicação ou não. O problema é que não leva em consideração os expoentes dos dois números; assume que os expoentes estão próximos de zero. Como é isso? Isso ocorre porque a precisão de um float não é determinada pela magnitude ("Nesta CPU, os resultados estão sempre dentro de 1,0e-7 da resposta!"), mas pelo *número de bits corretos* . O EPSILON acima é uma tolerância; é uma declaração de quanta precisão você espera em seus resultados. E a precisão é medida em algarismos significativos, não em magnitude; não faz sentido falar em "1.0e-7 de precisão". Um exemplo rápido torna isso óbvio: digamos que temos os números 1,25e-20 e 2,25e-20. A diferença entre eles é de 1e-20, muito menor que o EPSILON, mas claramente não queremos dizer que sejam iguais. Se, no entanto, os números fossem 1,2500000e-20 e 1,2500001e-20, então poderíamos pretender chamá-los iguais.

A mensagem principal é que, ao definir o quão próximo é o suficiente, você precisa falar sobre quantos dígitos significativos deseja corresponder. Responder a esta pergunta pode exigir alguma experimentação; experimente seu algoritmo e veja quão próximos os resultados "iguais" podem chegar.

Transbordar

Vamos continuar. Devido à incômoda finitude dos computadores reais, o excesso numérico é uma das preocupações mais comuns dos programadores. Se você adicionar um ao maior número inteiro sem sinal possível, o número volta para zero. Irritantemente, você não pode dizer que esse número inteiro transbordou só de olhar para ele; parece igual a qualquer zero. A maioria das CPUs definirá um bit de sinalização sempre que uma operação estourar, e verificar esse bit é uma das poucas otimizações de linguagem assembly codificadas manualmente que não são obsoletas.

No entanto, uma das coisas realmente interessantes sobre os carros alegóricos é que, quando eles transbordam, você fica convenientemente com +/-inf. Essas quantidades tendem a se comportar conforme o esperado: +inf é maior que qualquer outro número, -inf é menor que qualquer outro número, inf+1 é igual a inf e assim por diante. Esta propriedade também torna os floats úteis para verificar o estouro na matemática de inteiros. Você pode fazer um cálculo em ponto flutuante e simplesmente comparar o resultado com algo como INT_MAX antes de retornar ao número inteiro.

Casting abre sua própria lata de minhocas. Você precisa ter cuidado, pois seu float pode não ter precisão suficiente para preservar um número inteiro inteiro. Um número inteiro de 32 bits pode representar qualquer número decimal de 9 dígitos, mas um número flutuante de 32 bits oferece apenas cerca de 7 dígitos de precisão. Portanto, se você tiver números inteiros grandes, fazer essa conversão irá derrotá-los. Felizmente, os duplos têm precisão suficiente para preservar um número inteiro inteiro de 32 bits (observe, novamente, a analogia entre a precisão do ponto flutuante e a

faixa dinâmica de números inteiros). Além disso, há alguma sobrecarga associada à conversão entre tipos numéricos, indo de float para int ou entre float e double.

Esteja você usando números inteiros ou não, às vezes o resultado é simplesmente grande demais e isso é tudo. No entanto, você deve tentar evitar transbordar resultados desnecessariamente. Freqüentemente, o resultado final de um cálculo é menor do que alguns dos valores intermediários envolvidos; mesmo que seu resultado final seja representável, você poderá transbordar durante uma etapa intermediária. Evite essa gafe numérica! O exemplo clássico (de "Receitas Numéricas em C") é calcular a magnitude de um número complexo. A implementação ingênua é:

```
magnitude dupla(duplo re, duplo im)
{
    return sqrt(re*re + im*im);
}
```

Digamos que ambos os componentes sejam 1e200. A magnitude é 1,4142135e200, bem dentro da faixa de um duplo. No entanto, elevar 1e200 ao quadrado resulta em 1e400, o que está fora do intervalo – você obtém o infinito, cuja raiz quadrada ainda é infinita. Aqui está uma maneira muito melhor de escrever esta função:

```
1  double magnitude(double re, double im)
2  {
3      double r;
4
5      re = fabs(re);
6      im = fabs(im);
7      if (re > im) {
8          r = im/re;
9          return re*sqrt(1.0+r*r);
10     }
11     if (im == 0.0)
12         return 0.0;
13     r = re/im;
14     return im*sqrt(1.0+r*r);
15 }
```

Tudo o que fizemos foi reorganizar a fórmula trazendo um re ou im fora da raiz quadrada. Qual deles destacamos depende de qual é maior; se elevarmos ao quadrado im/re quando im for maior, ainda corremos o risco de estouro. Se im for 1e200 e re for 1, claramente não queremos elevar im/re ao quadrado, mas elevar ao quadrado re/im está ok, pois é 1e-400, que é arredondado para zero – próximo o suficiente para obter a resposta correta. Observe a assimetria: grandes magnitudes podem fazer com que você se perca em +inf, mas pequenas magnitudes terminam como zero (não -inf), o que é uma boa aproximação.

Perda de significância

Finalmente, chegamos às questões de “obter a resposta certa”. A igualdade incerta é apenas a ponta do iceberg dos problemas causados pela exatidão e precisão limitadas. Ter seus decimais cortados em algum momento causa uma quantidade surpreendente de estragos na matemática. "Perda de significância" refere-se a uma classe de situações em que você acaba perdendo inadvertidamente a precisão (descartando informações) e potencialmente terminando com resultados ridiculamente ruins.

Como vimos, a representação 1.m evita desperdícios ao garantir que quase todos os carros alegóricos tenham precisão total. Mesmo que apenas a parte mais à direita da mantissa seja definida (assumindo um expoente comum), todos os zeros antes dela contam como algarismos significativos por causa do 1 *implícito* . caso contrário, os implícitos seriam cancelados, juntamente com quaisquer dígitos da mantissa correspondentes. Se os dois números diferissem apenas no último bit, nossa resposta teria precisão de apenas um bit! Ai!

Assim como evitamos o estouro na função de magnitude complexa, há essencialmente sempre uma maneira de reorganizar um cálculo para evitar a subtração de quantidades muito próximas (eu me cubro dizendo "essencialmente sempre", já que a matemática por trás disso está muito além do escopo deste artigo). Naturalmente não existe um método geral para fazer isso; meu conselho seria apenas examinar e dar uma olhada em todas as suas subtrações sempre que começar a obter resultados suspeitos. Um exemplo de técnica que pode funcionar seria transformar polinômios em funções de 1/x em vez de x (isso pode ajudar no cálculo da fórmula quadrática, por exemplo).

Um problema relacionado surge ao somar uma série de números. Se alguns termos da sua série estiverem em torno de um épsilon de outros termos, sua contribuição será efetivamente perdida se os termos maiores forem adicionados primeiro. Por exemplo, se começarmos com 1,0 (flutuação de precisão simples) e tentarmos adicionar 1e-8, o resultado será 1,0, pois 1e-8 é menor que épsilon. Neste caso o pequeno termo é engolido completamente. Em casos menos extremos (com termos de magnitude mais próxima), o termo menor será engolido parcialmente – você perderá precisão.

Se você tiver sorte e os pequenos termos de sua série não significarem muito, então esse problema não o afetará. No entanto, muitas vezes um grande número de pequenos termos pode contribuir significativamente para uma soma. Nestes casos, se você não tomar cuidado, você continuará perdendo precisão até ficar confuso. Às vezes, as pessoas literalmente classificam os termos de uma série do menor para o maior antes de somar, se esse problema for uma grande preocupação.

Uma regra prática

Uma enorme quantidade de informações está disponível descrevendo pegadinhas numéricas e suas soluções – muito mais do que todas, exceto o programador científico dedicado, deseja lidar. Para simplificar as coisas, a maneira como frequentemente pensamos sobre problemas de perda de precisão é que um flutuador gradualmente fica "corrompido" à medida que você realiza mais e mais operações nele. Pegue o cosseno de pi/2 mencionado anteriormente, 6.12303e-17. Por si só não é tão ruim, é bem próximo de zero. Mas se o nosso próximo passo fosse dividir por 1e-17, então ficaríamos com cerca de 6, o que está muito longe do zero que esperávamos.

Isso torna suspeitos algoritmos com muito "feedback" (considerando saídas anteriores como entradas). Muitas vezes você tem a opção de modificar alguma quantidade de forma incremental ou explícita; você poderia dizer "x += inc" em cada iteração de um loop ou poderia usar "x = n*inc". Abordagens incrementais tendem a ser mais rápidas e, neste caso simples, não é provável que haja um problema, mas para estabilidade numérica é preferível "atualizar" um valor definindo-o em termos de quantidades estáveis. Infelizmente, o feedback é uma técnica poderosa que pode fornecer soluções rápidas para muitos problemas importantes. Tudo o que posso dizer aqui é que você deve evitá-lo se for claramente desnecessário; quando precisar de um bom algoritmo para algo como resolver equações não lineares, você precisará procurar aconselhamento especializado.

Não se esqueça dos números inteiros

Por último, um lembrete para não esquecer o humilde número inteiro: a sua precisão pode ser uma ferramenta útil. Às vezes, um programa precisa acompanhar algum tipo de fração variável, talvez um fator de escala. Nesta situação você sabe que o número que está armazenando é racional, então você pode evitar todos os problemas da matemática de ponto flutuante armazenando-o como um numerador e denominador inteiro. Isto é particularmente fácil para frações unitárias; se você precisar se mover entre 1/2, 1/3, 1/4, etc., deverá armazenar claramente apenas o denominador e regenerar 1.0/denom sempre que precisar da fração como um ponto flutuante. O próximo artigo: Impressão limpa de números de ponto flutuante