关于实数精确计算的一个可行中值定理

张静雯

2011年 7月 15日

摘要

本论文在C++语言的基础上使用„iRRAM库“来编译和实现三分法这一算法，以便达到实数精确计算这一目标。论文中涉及了实数，实数序列和实数函数可计算性的基础理论知识，并且运用中值定理和三分法证明函数零点值的可计算性。最后对于这一算法的实现程序加以详细的解释，并同时对„iRRAM库“中所用到的主要函数进行了简单的介绍。

目录

1 前言

在本论文中编写了用三分法精确计算实数函数零点值的程序，该程序使用了特里尔大学诺伯特·米勒教授(Prof. Norbert Müller an der Universität Trier)编写的C++-库 iRRAM。该库模拟了一个Real-RAM 机，用迭代的方法计算出任意给定精度的实数值。

本论文分为三部分。首先分析实数，实数序列和实数函数的可计算性，并证明了三分法可用来计算(在限定条件下)实数函数的零点值。然后简单介绍并解释C++-库iRRAM的主要函数，他们被用于实现三分法这一算法。最后解释了所编程序的重要部分。

本论文也备有德文版本，标题为„ Ein effektiver Zwischenwertsatz als Fallstudie für exakte reelle Arithmetik“。

2 基本原理

可计算性分析是数值分析，也就是具备任意精度的近似数字计算的理论基础。在一般的分析中，人们只研究实数，函数以及在此基础上构建的结构。而可计算性分析则研究实数结构在数字计算机上何时及如何被计算。

可计算性分析这个概念和图灵机紧密相连，图灵机将在下节进行介绍。

2.1 图灵机

„图灵机是英国科学家阿兰·图灵(Alan Turing)在1936年发展出来的模型，用来构建一个可计算函数的类。它是理论计算机科学的基础。“ [8]

图灵机是一个简单的计算机，用来对函数计算的算法进行形式上的定义。 按照 „邱奇-图灵论题(Church-Turing-These)“：一切直觉上能计算的函数都可用图灵机计算，反之亦然 [1]。这个论题是无法证明的，因为 „可计算函数“是无法在形式上加以定义的。为了证明函数的不可计算性，人们就假定此论题为真。除了图灵以外其他人也没有发现比图灵机更有效的计算模型。

2.2 实数的可计算性

柯西序列(Cauchy-Folge)是计算实数及其序列，幂级数和函数的基本帮助工具。从有理数序列的可计算性出发，可以很容易得定义实数和实数序列的可计算性 [7]：

定义 2.2.1. 一个实数x被称为可计算的，如果存在一个可计算的有理数序列(ε_m)，并且对于另一个可计算的有理数序列(q_m)，$\forall m \in \mathbb{N}$，满足

$$|x - q_m| \leq \varepsilon_m \xrightarrow{m\to\infty} 0$$

与其等价的还有一个定义：

定义 2.2.2. 一个实数x被称为可计算的，如果对于一个可计算的有理数序列(q_n)，$\forall n \in \mathbb{N}$，满足

$$|x - q_n| \leq 2^{-n}$$

2.3 实数序列的可计算性

现在已证明了实数的可计算性。由于计算机的存储量是有限的，实数虽可计算，但不能储存。因此实数序列将被用来近似一个实数。这就引出了实数序列是否可计算的问题:

定义 2.3.1. 一个实数序列(x_m)被称为可计算的，如果对于一个可计算的有理数双序列$(q_{m,n})$，$\forall m,n\in\mathbb{N}$，满足

$$|x_m-q_{m,n}|\leq 2^{-n}$$

为了用实数序列对实数进行近似计算，有以下定义：

定义 2.3.2. 一个可计算的实数序列(x_m)被称为是有效收敛的，如果存在可计算有理数序列(ε_m)，满足

$$|x_m-x|\leq\varepsilon_m\xrightarrow{m\to\infty}0$$

这就引出以下定理：

定理 2.3.3. 如果一个可计算的实数序列(x_m)有效收敛，则$x:=\lim_{m\to\infty}x_m$ 可计算。

证明. (x_m)是一个可计算的实数序列，则对可计算的有理数序列$(q_{m,n})$满足：

$$|x_m-q_{m,n}|\leq 2^{-n}$$

由于(x_m)是有效收敛的，则存在可计算有理数序列 (ε_m)，且

$$|x_m-x|\leq\varepsilon_m\xrightarrow{m\to\infty}0$$

现在假设$q_m:=q_{m,m}\subseteq\mathbb{Q}$是可计算序列，则

$$|x-q_m|=|x-x_m+x_m-q_m|\leq\underbrace{|x-x_m|}_{\varepsilon_m\xrightarrow{m\to\infty}0}+\underbrace{|x_m-q_{m,m}|}_{2^{-m}\xrightarrow{m\to\infty}0}\xrightarrow{m\to\infty}0$$

因此按照定义2.2.1 $x:=\lim_{m\to\infty}x_m$ 是可计算的。 □

由此推出，一个可计算的实数序列有效收敛是其极限值可计算的充分条件。

2.4 实数函数的可计算性

从实数序列的可计算性可推出实数函数的可计算性。首先来观察一下幂级数并证明它是可计算的：

定理 2.4.1. 假设$P(x):=\sum_{n=0}^{\infty}a_nx^n$是一个幂级数，$x$是一个可计算的实数，且$a_n$是一个可计算的有理数系数序列，并且在它的收敛半径

$$R:=\frac{1}{\lim_{n\to\infty}\sup\sqrt[n]{|a_n|}}$$

里有效收敛，则$P(x)$是可计算的。

在两个实数 a 和 b，且 $a < b$ 之间总是存在至少一个有理数 r，且满足 $a < r < b$。所以在两个实数之间也存在无数个有理数。那么也可以说，有理数„稠密“地存在在实数之间。

证明.

$$|x|, R \in \mathbb{R}, |x| < R \Rightarrow \exists q, r \in \mathbb{Q},\ |x| < q < r < R$$

柯西－阿达马(Cauchy-Hadamard): $\exists M \in \mathbb{N}, |a_n| < Mr^{-n}, \forall n \in \mathbb{N}$

$$\underbrace{\left|\sum_{k=0}^{\infty} a_k x^k\right.}_{P(x)=:x} - \underbrace{\left.\sum_{k=0}^{m} a_k x^k\right|}_{x_m} = \left|\sum_{k=m+1}^{\infty} a_k x^k\right| \leq \sum_{k=m+1}^{\infty} |a_k||x|^k$$

$$\overset{\text{Cauchy-Hadamard}}{<} \sum_{k=m+1}^{\infty} Mr^{-k}|x|^k = M\sum_{k=m+1}^{\infty}\left(\frac{|x|}{r}\right)^k = M\left(\frac{|x|}{r}\right)^{m+1}\underbrace{\sum_{k=0}^{\infty}\left(\frac{|x|}{r}\right)^k}_{\text{几何级数}}$$

$$= M\left(\frac{|x|}{r}\right)^{m+1}\frac{1}{1-\frac{|x|}{r}} \overset{|x|<q<r}{\leq} \underbrace{M\left(\frac{q}{r}\right)^{m+1}\frac{1}{1-\frac{q}{r}}}_{\in\mathbb{Q}} =: \varepsilon_m \xrightarrow{m\to\infty} 0\text{可计算}$$

并且 x_m 是可计算的 [6]
$\Rightarrow x_m$ 根据定义 2.3.2 有效收敛。
因此 $P(x)$ 由定理 2.3.3 推出是可计算的。

□

现在下列函数的可计算性就可以很容易地理解:

2.4.1 指数函数

$$e^x = \sum_{n=0}^{\infty}\frac{1}{n!}x^n$$

指数函数的系数序列 $a_n = \frac{1}{n!} \in \mathbb{Q}$ 是可计算的，其收敛半径 $R = \infty$，因此指数函数 e^x 按照定理 2.4.1 $\forall x \in \mathbb{R}$ 是可计算的。

2.4.2 三角函数

首先观察一下正弦函数：

$$\sin x = \sum_{n=0}^{\infty}\frac{(-1)^n}{(2n+1)!}x^{2n+1}$$

它的系数序列

$$a_n = \begin{cases} -\frac{1}{m!}, & m = 2n+1 \\ 0, & m = 2n \end{cases}$$

是可计算的，其收敛半径 $R=\infty$， 因此按照定理 2.4.1 正弦函数 $\sin x$, $\forall x \in \mathbb{R}$ 同样是可计算的。

类似于正弦函数，得出余弦函数

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$

也是可计算的。

2.4.3 对数函数

对数函数同样是可计算的：

$$\ln(1+x) = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{x^n}{n}$$

其中系数序列

$$a_n = -\frac{(-1)^n}{n} \in \mathbb{Q}, \forall n > 0$$

是可计算的，其收敛半径为：

$$R = \frac{1}{\lim_{n\to\infty} \sup \sqrt[n]{\frac{1}{n}}} = 1$$

$$r = 1$$

$$|x| < 1$$

$$I := (0,2)$$

按照定理 2.4.1 对数函数 $\ln(1+x), \forall |x| < 1$ 是可计算的。

对上述例子中的函数，它们的可计算性得到了证明。实际上几乎所有的数学分析函数都是可计算的，也就是说：存在计算方法，能在有限步骤内得出函数值 [2]。

2.5 中值定理

目前为止，给出了可计算的实数x，则函数$f(x)$是可计算的。那么在什么条件下，满足 $f(x)=0$ 的实数 x 可计算呢？

中值定理保证了这种实数的存在性：

定理 2.5.1. 假设$f:[a,b]\to\mathbb{R}$是连续的实数函数，且实数u满足$f(a)\le u\le f(b)$或$f(b)\le u\le f(a)$， 则存在$x\in[a,b]$使得 $f(x)=u$。

零点定理: 假设实数函数$f(x)$在闭区间[0,1]上连续，且满足$f(0)*f(1)<0$，则必存在$x\in(0,1)$使$f(x)=0$成立。 零点定理是中值定理的一个特例，是由伯纳德·博尔扎诺（Bernard Bolzano）于1817年首先证明的 [9]。

2.6 三分法

现在假设一个给定的，可计算的连续函数$f:[0,1]\to\mathbb{R}$，且$f(0)*f(1)<0$ 证明能精确计算出一个零点值 $x\in[0,1]$，$f(x)=0$。我们先设定函数 f 有，且只有一个零点：

定理 2.6.1. 假设$f:[0,1]\to\mathbb{R}$是可计算的连续函数，满足$f(0)*f(1)<0$，且只有一个零点，则这个零点可计算的。

证明. [2, 第 459 页] 在传统的二分法中，函数值将与 0 比较，但是这种比较在有限的时间内是不可靠的。因此将使用二分法的一个变种即三分法，它要比二分法稳定。 首先设 $a_0:=0$，$d_0:=1$，并假设 $a_i,d_i\in[0,1]$，$d_i-a_i=(2/3)^i$ 且 $f(a_i)*f(d_i)<0$ (当 $i=0$时，此为真)。然后同时计算函数值 $f(a_i)*f(a_i+\frac{2}{3}(d_i-a_i))$ 和 $f(a_i+\frac{1}{3}(d_i-a_i))*f(d_i)$，当其中一值为负时，将进行下一个循环的计算。三分法保证了其中必定有一值为负。当第一个值小于 0 时，则设 $a_{i+1}:=a_i$, $d_{i+1}:=a_i+\frac{2}{3}(d_i-a_i)$; 当第二个值小于 0 时，则设 $a_{i+1}:=a_i+\frac{1}{3}(d_i-a_i)$, $d_{i+1}:=d_i$。这样就产生了一个区间 $[a_i,d_i]$ 的序列，其长度为 $(\frac{2}{3})^i$，并收敛于函数 f 的零点。因此由定理 2.3.3 得出此零点值是可计算的。 □

对于在闭区间 $[0,1]$ 上含有多个零点的函数，它的其中一些零点值是可计算的，但是三分法并不适用于这种情况。

3 iRRAM (迭代 Real-RAM)

iRRAM 是一个对实数进行精确计算的 C++ 库，它是在 Real-RAM 这一概念上构建的。Real-RAM即一个可以进行实数计算的图灵机，iRRAM 库模拟了这样的一个 Real-RAM 机，它的精度随着迭代而不断提高，达到任意给定的精度。 [4]

在这一章中将简要介绍 iRRAM 的基本结构和重要的函数，它们将用于编译和实现实数的精确计算。

3.1 iRRAM 简介

用 iRRAM精确计算实数值，要用到`REAL`类。作为例子，我们用Heron算法计算 $\sqrt{x}$ (表 1 [5])来说明。

```
1  // Function to approximate the square root for basis 2
2  REAL wurzel_approx(long p, REAL x)
3  {
4   // Define two real numbers a and b
5   REAL a = 1, b = x;
6   // Iterate the algorithm until the result has the needed precision
7   do {a = (a + b) / 2; b = x / a; } while( !bound(a - b, p) );
8   // Return the result
9   return a;
10 }
11
12 // The limit of the real series is the "real" square root with basis 2
13 REAL wurzel(const REAL& x) {return limit(wurzel_approx,x);}
```

表 1: iRRAM-程序：运用 Heron算法计算$\sqrt{x}$

在第5行，变量 a 和 b 被初始化，都定义为 REAL 类。REAL 类会在 3.2 节中作进一步介绍。Heron算法在第 7 行实现，其中使用 iRRAM 的函数 bound，来检验近似值的精度。如果精度不够，则将继续进行迭代。函数 bound 会在 3.3 节中介绍。

函数 wurzel_approx 得出一组实数序列，它会在第 13 行中被函数 limit 调用，以计算出极限值(即$\sqrt{x}$)。最后由函数 wurzel 返回这一结果。

函数 limit 在 iRRAM 中定义为：
REAL limit (REAL a(long, const REAL&), const REAL& x); 它要求二个参数：实数函数 a(long, const REAL&)，其本身又带两个参数，以及一个要开根号的实数 x。返回结果$\sqrt{x}$就是区间序列的极限值。

由表 1 所示，只需实例化 REAL 类，就可精确计算出实数值。iRRAM 确保计算出任意给定精度的近似值，使用者无须关心其实现过程。

3.2 REAL类(REAL Class)

REAL 类是 iRRAM 库的核心，它有很多构造函数，在求零点值的程序里用到了以下构造函数：

```
1 // Returns real number 0
2 REAL ();
3 // Returns real number i
4 REAL (const int i);
5 // Returns real number y
6 REAL (const REAL& y);
```

表 2: REAL类的构造函数

REAL 重定义了操作符$+, -, *, /$ 因此在大多数情况下，可以象平时一样进行算术运算。iRRAM 保证操作符运算的精确度。

3.3 多值函数

iRRAM 要求所有函数是连续的 [4, 第15页]，但有很多函数是不连续的，因此它们不能直接被用于 iRRAM。这些函数至少要在给定区间内连续，或者就如除法中 0 被从定义域里去掉。

在表 1 的程序 wurzel_approx 及求零点值的程序 trisection_approx (程序第 109 行) 中， 函数 bound(x,k) 被调用来作为近似中断的条件。它在 iRRAM 中是这样定义：bool bound(const REAL& x, const long k)，用它可以检验实数 x 是否„足够小“。

$$\texttt{bound(x,k)} = \begin{cases} T, & |x| < 2^k \\ \left\{ \begin{matrix} T \\ F \end{matrix} \right\}, & 2^{k-2} \le |x| \le 2^k \\ F, & |x| > 2^{k-2} \end{cases}$$

如上所示， 函数 bound(x,k) 的返回值为布尔值的真或假 (T 或 F)。并可以看到，当 $2^{k-2} \le |x_0| \le 2^k$ 时，该函数可以有两个值即„多值函数“。到底该函数返回 T 还是 F，是不确定的。由于返回值的选择被打包在函数中，每次的迭代结果也就不可预见。

因为函数 bound(x,k) 是个多值函数，所以函数 wurzel_approx 也是个多值函数：

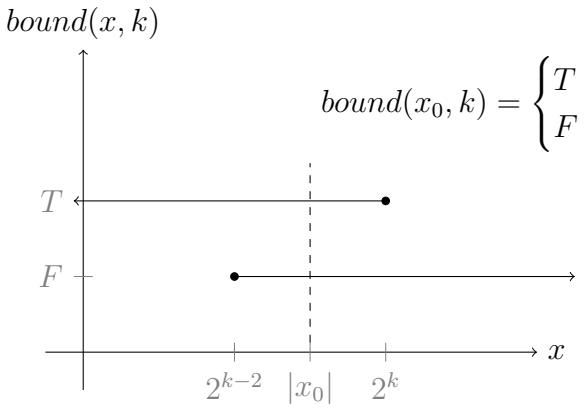

图 1: 多值函数 $bound(x,k)$

- 当 $|a-b| > 2^p$ 时，肯定会继续迭代 (因为 `!bound(a-b,p)==TRUE`);
- 当 $|a-b| < 2^{p-2}$ 时，肯定停止迭代 (因为 `!bound(a-b,p)==FALSE`);
- 当 $|a-b| \in [2^{p-2}, 2^p]$ 时，函数 `wurzel_approx` 具有多个值，迭代循环可在不同的深度上中止 (因为 `!bound(a-b,p)==TRUE` 或 `FALSE`)

3.4 惰性布尔 LAZY_BOOLEAN

由于 iRRAM 用区间近似来计算实数，因此当比较两个实数值的大小时，可能出现不同情况，如图 2 所示：

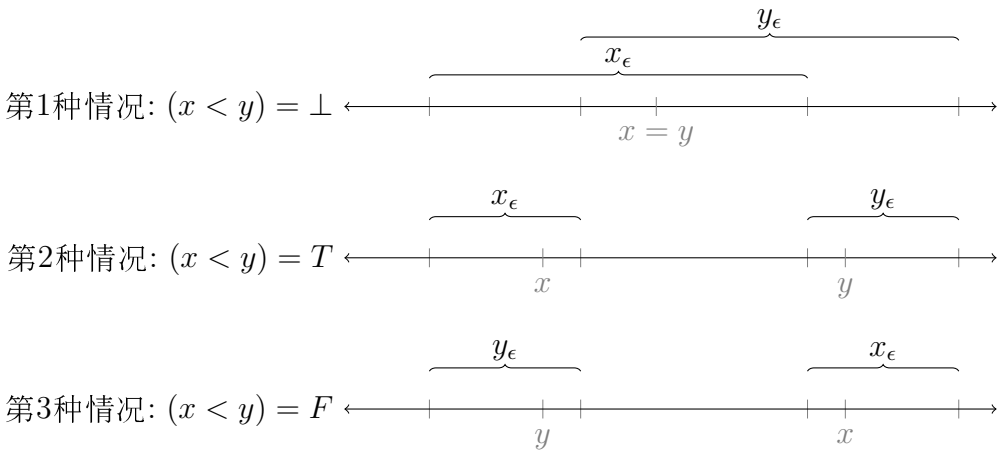

图 2: 两个实数大小比较的3种情况，x_ϵ, y_ϵ 描述了 iRRAM 所使用的近似区间。

只要区间 x_ϵ und y_ϵ 不相交(如图 2 中的第 2，3 种情况)，那么就能比较 x 和 y 。对于第1种情况 $x = y$ ，两个区间 x_ϵ 和 y_ϵ 有重叠，无论如何迭代缩小区间 x_ϵ 和 y_ϵ ，都会有重叠。在这种情况下试图比较数的大小，将导致死循环。

由于无法决定结果到底为真还是假，iRRAM 提供了一个值 ⊥，使计算能够继续。也因此产生了惰性布尔 LAZY_BOOLEAN 这一数据类型，它有三个值分别为：T, F 和 ⊥。逻辑操作符 &&, || 和 ! 也同时被扩展，使之能接受这三个值。逻辑操作符的真值表如图 3 所示：

x||y	T	F	⊥
T	T	T	T
F	T	F	⊥
⊥	T	⊥	⊥

x&&y	T	F	⊥
T	T	F	⊥
F	F	F	F
⊥	⊥	F	⊥

!x	
T	F
F	T
⊥	⊥

图 3: 逻辑操作符 &&, || 和 ! 的惰性布尔真值表

对惰性布尔这一数据类型有两种处理方法：其一是把值 T 和 F 转化为布尔(bool)值，但值 ⊥ 则会直接导致死循环。 [3, Vgl.]

$$\texttt{(x<y)} = \begin{cases} T, & x < y \\ F, & x > y \\ \bot, & x = y \end{cases}$$

x	bool(x)
T	T
F	F
⊥	undefined

3.5 函数 choose

其二用函数 choose 来对惰性布尔这一数据类型加以处理：

```
int choose ( const LAZY_BOOLEAN& x1= false,
             const LAZY_BOOLEAN& x2= false,
             const LAZY_BOOLEAN& x3= false,
             ... ) ;
```

表 3: 函数 choose 定义

函数 choose 最多可有 6 个参数 $x1, x2, x3...$，并按以下规则确定函数值：如果函数 choose 至少有一个参数值为 T，就返回那个参数的指数，在该情况下允许其它参数值为 F 和 ⊥；

如果所有参数值为 F，就返回 0；

当参数值只有 F 和 ⊥，就会进入无限循环 (使用函数 choose 时要避免这一情况的产生)；

当有多个参数值同时为 T 时，其中任意一个的指数将被返回，但是无法确定是哪一个。

3.6 仿函数 FUNCTION

仿函数是一个对象，但在使用时，可象函数一样被调用。为了更好的定义数字操作符，我们使用了FUNCTION 这个仿函数，它要求参数类型和返回值类型作为模板参数。
FUNCTION<PARAM,RESULT>()

4 用 iRRAM 库实现三分法并精确计算零点值

我所编的程序要在给定的可计算的实数函数 $f:[a,b]\to\mathbb{R}$ 且满足 $f(a)*f(b)<0$ 的条件下能够精确计算零点值。如果给出在闭区间上的零点个数，该程序能够计算出每个零点的精确值。

4.1 程序解释

函数 `null_points (FUNCTION<REAL,REAL> f, int n, REAL l, REAL r)`，的第一个参数是一个仿函数 f，即给定的可计算的实数函数，我们就是要求此函数的零点; 第二个参数 n 给出了函数零点值的个数; 最后的两个参数分别是区间的左右边界 l 和 r。

函数 `null_points` 把闭区间 $[l,r]$ 分成 n 个小的闭区间，并且在每个小的闭区间上满足 $f(l_i)*f(r_i)<0$(函数值一正一负)，且$i=0,...,n-1$ 。这就相当于每个小的闭区间里有，且只有一个零点，也就能用三分法精确计算零点值。

```
214   // vzw: used to count the intervals with different sign of function values at boundary
215   int vzw=0;
216   if (vl!=vr) vzw++;
217
218   // number of null points which are not yet found
219   int numNullPoints = n;
220   while (vzw<n)
221   {
222     intvl z=q.front();
223     q.pop();
224
225     // compute m
226     REAL m = Find_Non_null_point(f, numNullPoints, z.l, z.r);
227     bool vm=(f(m)>0);
228     q.push(intvl(z.l,m,z.vl,vm));
229     q.push(intvl(m,z.r,vm,z.vr));
230     // count the change of signs ...
231     if (z.vl == z.vr && z.vl != vm)
232     {
233       vzw+=2;
234       numNullPoints -= 2;
235     }
236   }
```

Docs/nullstellensuche.cc

变量 `vzw` (第 215 行) 记录正负符号变换的次数, 并在 (第 220 行) 作为循环的中止判断条件。

在每一次循环时，都从等待队列 q (先进先出FIFO)中取出最上面的那个区间 $[z.l,z.r]$，然后调用函数 `Find_Non_null_point` 找出在此区间中任意一个不是零点的数 m (即 $f(m)!=0$)。现在把区间在点 m 处一分为二：$[z.l,m]$ 和 $[m,z.r]$，这两个新区间被存入等待队列 q 中，它们又将被继续划分。每次区间划分后都会检验是否有正负符号在新区间上变换 (第 231 行)，如果有变换，变量 `vzw` 就会自动加 2 。

如此当找到 n 个这样的区间后，函数 `trisection` (第 245 行) 就能被调用，因为每个区间里正好只有一个零点。如同表1，函数`trisection` 调用 `limit` 来计算由函数 `trisection_approx` 描述的序列的极限值。

4.2 函数 find_non_null_point

该函数用来找出给定函数 iFunc 在区间 $[left, right]$ 上任意一个不是零点的数 m 。它需要 4 个参数分别为：函数 iFunc，在区间上的零点个数 iNumberOfNullPoints，和区间的左右边界 left 及 right。该函数将区间 $[left, right]$ 进行等分，得到点 $m_1, ..., m_{iNumberOfNullPoints+1}$，再计算它们函数值的绝对值 $|f(m_1)|$, ..., $|f(m_{iNumberOfNullPoints+1})|$，然后存入数组 func_values)中。

```
167 // Find an arbitrary x in (left, right), whose y value is not 0.
168 // Function values at n+1 positions (x) are calculated. At least one of them are not null.
169 // The absolute values of n+1 function values are saved in an array. Use function choose
        to check if they are
170 // greater than zero (not undefined). To do this, loop the array in the reverse order, use
        the k-th element
171 // and the 0 to k-1 elements, which are or-ed, as the two parameters of the function
        choose.
172 REAL Find_Non_null_point(FUNC iFunc, int iNumberOfNullPoints, REAL left, REAL right)
173 {
174   REAL diff = right - left;
175   std::vector<REAL> func_values;
176   std::vector<REAL> x_values;
177   for(int i=1; i<=iNumberOfNullPoints+1; ++i)
178   {
179     REAL x = left + i * diff / (iNumberOfNullPoints + 2);
180     REAL abs_value = abs(iFunc(x));
181     func_values.push_back(abs_value);
182     x_values.push_back(x);
183   }
184   std::vector<LAZY_BOOLEAN> tests(func_values.size());
185   for(unsigned int i=0; i < func_values.size(); ++i)
186   {
187     tests[i] = func_values[i] > 0;
188   }
189
190   std::vector<LAZY_BOOLEAN> new_tests(func_values.size());
191   new_tests[0] = tests[0];
192   for(unsigned int i=1; i<new_tests.size(); ++i)
193     new_tests[i] = new_tests[i-1] || tests[i];
194   for(unsigned int i=tests.size()-1; i>0; --i)
195   {
196    if (choose(new_tests[i-1], tests[i])==2)
197    return x_values[i];
198   }
199   return x_values[0];
200 }
```

Docs/nullstellensuche.cc

接下来检验所有绝对值是否大于 0，其结果作为 LAZY_BOOLEAN 数据类型被存入数组 tests 中。因为函数 choose 最多只能有 6 个参数，而零点 $iNumberOfNullPoints$ 的个数很可能远远多于 6 个，所以在从 192 行到 198 行用到了两次循环，及两个数组来找出不是零点的数。因为函数有 n 个零点，而我们在区间上找了 $n+1$ 个点，所以保证了至少有一个点不是零点，即 $\exists i.\ |f(m_i)| > 0 = T$，函数 choose 不会进入无限循环。

4.3 函数 trisection_approx

该函数要求两个 „对“ (pairs) 作为参数。„对“ 的第一个元素是精度 p 和函数 $func$，第二个元素是区间的左右边界 a 和 d (111 - 114行)。

第 117 行函数 `bound` 检验是否已经达到所需精度，或者继续循环。每个循环步骤如同定理 2.6.1 的证明所描述的一样。由于在给定的区间中至少有一个正负号变换，因此第 130 行的函数 `choose` 必有一个参数值为 T ，可以用它选择在哪个区间继续迭代。当精度达到时，该函数返回零点的近似值。

```
107 // in order to use the iRRAM limit operator for FUNCTION<int,REAL>
108 // we formulate with a "trisection_parameter" as argument
109 REAL trisection_approx(const trisection_parameter& p_f_l_r)
110 {
111   int  p           = p_f_l_r.first.first;
112   FUNCTION<REAL,REAL> func = p_f_l_r.first.second;
113   REAL a           = p_f_l_r.second.first;
114   REAL d           = p_f_l_r.second.second;
115
116   REAL result = d;
117   while( !bound(d-a,p-1) )
118   {
119     REAL new_a = calc_a(a, d);
120     REAL new_d = calc_d(a, d);
121
122     REAL f_left_left = func(a);
123     REAL f_left_right = func(new_d);
124     REAL f_right_left = func(new_a);
125     REAL f_right_right = func(d);
126
127     REAL f_left = f_left_left * f_left_right;
128     REAL f_right = f_right_left * f_right_right;
129
130     int test=choose(f_left<0,f_right<0);
131     if(test==1) // f_left<0: [a, new_d] is the next interval to be concerned
132       {
133         d = new_d;
134         result = d;
135       }
136     else        // f_right<0: [new_a, d] is the next interval to be concerned
137       {
138         a = new_a;
139         result = a;
140       }
141   }
142   return result;
143 }
```

Docs/nullstellensuche.cc

该函数编程的难点在于使用嵌入的„对“ 来解决 iRRAM 要求的函数只能有一个参数所带来的限制。

4.4 程序运行结果

程序代码开头有 12 个给定的测试函数。运行程序时，先通过输入数字1到12，来选择测试函数，再给出所需精度（小数点后位数），然后程序能正确返回任意给定精度的零点值。但是有以下几个例外：

- 第三个测试函数 `sin3x` 不能计算零点值，是因为函数的左区间函数值为 0 （即$f(a) = 0$），不符合条件$f(a) * f(b) < 0$ 。

- 第五个测试函数 `exp_flat` ，因为它在给定区间里几乎就和 x 轴重叠，在用 iRRAM 库求它的实数值时，所需存储空间成指数增加，我们的计算机没有足够存储空间，所以也不能返回结果。

- 程序中的函数 `Find_Non_null_points` 运行较慢，特别当一个函数的零点个数很多时更是如此。这是因为所用算法比较复杂，在分隔区间时必须使用 REAL 类。所以如表格1 所示，第十个测试函数 `x20` 在给定精度 25 位的情况下，就需要$306.95s$ ，当精度要求提高时，计算机没有足够存储空间，所以也不能返回结果(对于第十一，十二个测试函数情况相似)。

在目前的算法中，区间上靠右的点被优先使用（循环是从后面开始）。这意味着所分区间不是平均分布的，区间都挤靠在右边。这也许对少数某些函数是好的，但对大多数函数来说，这是很低效的。我们现在将此算法进行适当修改，使其首先采用中间的点。当区间相对平均分布时，则能更快地找到n个要找的正确的区间。

二分区间的改进算法如下：

```
177   // begin new algorithm
178   std::queue<intvl> q;
179   intvl first(left, right, true, true);
180   q.push(first);
181   REAL l = left;
182   REAL r = right;
183   while(x_values.size() < iNumberOfNullPoints+2)
184   {
185     intvl item = q.front();
186     q.pop();
187     // the interval is always cut in half
188     REAL x = (item.l + item.r) / 2;
189     REAL y = abs(iFunc(x));
190     func_values.push_back(y);
191     x_values.push_back(x);
192
193     q.push(intvl(item.l, x, true, true));
194     q.push(intvl(x, item.r, true, true));
195   }
196   // end new algorithm
```

Docs/nullstellensuche–final.cc

在第188行中显示，区间一直在中点处被一分为二，这个改动大大提高了程序的运行时间，如表格 2 所示：对于零点个数较少的测试函数（第一到第九）运行时间基本没有变动; 但是对于零点个数多的测试函数，变化是显而易见的，第十，十一，和第十二个测试函数都给出了精确的零点值。

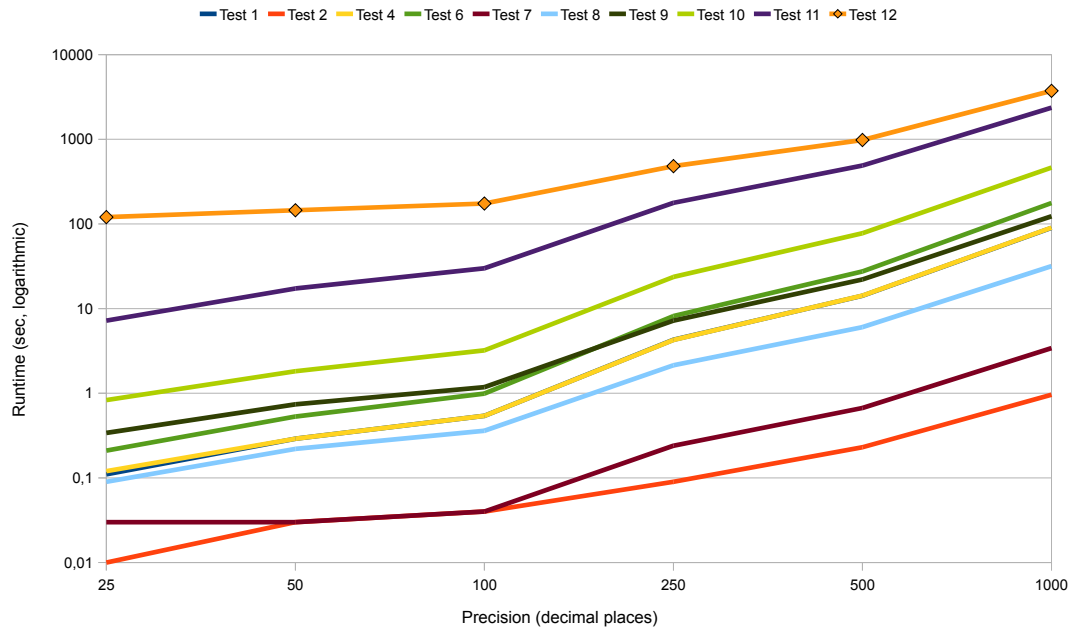

图 4: 优化算法的运行时间

Literatur

[1] 维基百科. 邱奇-图灵论题 — 维基百科, 自由的百科全书, 2011. [Online; accessed 14-June-2011].

[2] V. Brattka, P. Hertling, and K. Weihrauch. A tutorial on computable analysis. *New Computational Paradigms*, pages 425–491, 2008.

[3] N. Mueller. *Exact Real Arithmetic*.

[4] N. Mueller. The irram: Exact arithmetic in c++. *Computability and Complexity in Analysis*, pages 222–252, 2001.

[5] N. T. Mueller. The irram: Exact arithmetic in c++, 2010.

[6] M. Neeb. Beispiele berechenbarer reeller zahlen und abgeschlossenheitseigenschaften, 2010. [Online; Stand 11. Mar 2010].

[7] K. Weihrauch. *Computable analysis: an introduction*. Springer Verlag, 2000.

[8] Wikipedia. Turingmaschine — wikipedia, die freie enzyklopaedie, 2011. [Online; Stand 11. Juni 2011].

[9] Wikipedia. Zwischenwertsatz — wikipedia, die freie enzyklopaedie, 2011. [Online; Stand 14. Juni 2011].

A 程序运行时间表

测试函数	零点个数	精度(小数点后位数)					
		25	50	100	250	500	1000
TF 1	1	0.11(s)	0.29	0.53	4.25	14.1	89.82
TF 2	1	0.01	0.03	0.04	0.09	0.23	0.96
TF 4	1	0.12	0.29	0.54	4.26	14.25	90.32
TF 6	2	0.21	0.53	0.99	8.16	27.51	176.85
TF 7	2	0.03	0.03	0.04	0.24	0.67	3.41
TF 8	5	0.09	0.22	0.36	2.14	6.04	31.73
TF 9	10	0.68	1.1	1.62	8.29	24.33	127.62
TF 10	20	306.95	–	–	–	–	–
TF 11	40	–	–	–	–	–	–
TF 12	40	–	–	–	–	–	–

表格 1: 原程序的运行时间， (–) 表示1000s内没有返回运行结果。

测试函数	零点个数	精度(小数点后位数)					
		25	50	100	250	500	1000
TF 1	1	0.11(s)	0.29	0.53	4.25	14.1	89.82
TF 2	1	0.01	0.03	0.04	0.09	0.23	0.96
TF 4	1	0.12	0.29	0.54	4.26	14.25	90.32
TF 6	2	0.21	0.53	0.99	8.16	27.51	176.85
TF 7	2	0.03	0.03	0.04	0.24	0.67	3.41
TF 8	5	0.09	0.22	0.36	2.14	6.04	31.73
TF 9	10	0.34	0.74	1.18	7.21	22.09	123.12
TF 10	20	0.83	1.82	3.21	23.63	77.67	463.07
TF 11	40	7.21	17.28	29.95	177.44	490.07	2364.05
TF 12	40	120.32	145.08	174.11	481.21	980.46	3738.51

表格 2: 优化算法的运行时间，**TF 1**到**TF 8**的运行时间没有变动，如表格1。

B 程序代码

```
#include "iRRAM.h"
#include <queue>
#include <vector>
#include <utility>
#include <time.h>
using namespace iRRAM;

//******************  Test functions  **********************************/
// Test 1: cos3x has exact one null point in interval [0, 1], and x = pi/6.
REAL cos3x(const REAL& iX)
{
  return cos(3 * iX);
}
// Test 2: line has one null point at 1/3.
REAL line(const REAL& iX)
{
  return REAL(3) * iX - REAL(1);
}
// Test 3: sin3x has one null point on boundary. Error message is shown.
REAL sin3x(const REAL& iX)
{
  return sin(3 * iX);
}
// Test 4: flat is a ver flat function and has the same null point as cons3x.
REAL flat(const REAL& iX)
{
  return cos(3 * iX)/1000000000;
}
// Test 5: exp_flat is an exponential (!) sloping flat function with the null point at
    1/3.
REAL exp_flat(const REAL& iX)
{
  return (iX-REAL(1)/3)*exp(-1/(iX-REAL(1)/3)/(iX-REAL(1)/3));
}
// Test 6: sinus has 2 null points: 0 and pi
REAL sinx(const REAL& iX)
{
  return sin(iX);
}
// Test 7: parable has 2 null points: -1 and 1
REAL parable(const REAL& iX)
{
  return iX* iX-1;
}
// Test 8: x5 has 5 null points: 1/3,1/6,1/7,1/9,1/11
REAL x5(const REAL& iX)
{
  return (iX-REAL(1)/3)*(iX-REAL(1)/6)*(iX-REAL(1)/7)*(iX-REAL(1)/9)*(iX-REAL(1)/11);
}
// Test 9: x10 has 10 null points: 1/3,1/6,1/7,1/9,1/11,6,7,8,9,10
REAL x10(const REAL& iX)
{
  return (iX-REAL(1)/3)*(iX-REAL(1)/6)*(iX-REAL(1)/7)*(iX-REAL(1)/9)*(iX-REAL(1)/11)*(iX
      -6)*(iX-7)*(iX-8)*(iX-9)*(iX-10);
}
// Test 10: x20 has 20 null points: -10,-9,-8,...,-2,-1,1,2,...,8,9,10
REAL x20(const REAL& iX)
{
  REAL Z=1;
  for (int i=-20;i<= -1;i++) Z=Z*(iX - REAL(i) );
  for (int i=  1;i<= 20;i++) Z=Z*(iX - REAL(i) );
  return Z;
```

```
}
// Test 11: x40 has 40 null points: -1, -1/2, -1/3, ..., -1/20, 1/20, 1/19, ..., 1
REAL x40(const REAL& iX)
{
  REAL Z=1;
  for (int i=-20;i<= -1;i++) Z=Z*(iX - 1/REAL(i) );
  for (int i=  1;i<= 20;i++) Z=Z*(iX - 1/REAL(i) );
  return Z;
}
// Test 12: x40a has 40 null points: -1, -1/2^3, -1/3^3, ..., -1/20^3, 1/20^3, 1/19^3,
    ...1/2^3, 1
REAL x40a(const REAL& iX)
{
  REAL Z=1;
  for (int i=-20;i<= -1;i++) Z=Z*(iX - 1/REAL(i)/REAL(i)/REAL(i));
  for (int i=  1;i<= 20;i++) Z=Z*(iX - 1/REAL(i)/REAL(i)/REAL(i) );
  return Z;
}
/*****************************************************************/

// new left boundary is calculated.
REAL calc_a(REAL old_a, REAL old_d)
{
  return old_a + (old_d - old_a) / 3;
}

// new right boundary is calculated.
REAL calc_d(REAL old_a, REAL old_d)
{
  return old_a + 2 * (old_d - old_a) / 3;
}

// we need four parameters to do the trisection:
// 1) the intended precision
// 2) the function under consideration
// 3) a left border
// 4) a right border

// first we define a corresponding type.
// the standard libaries only allow for pairs, so we have to nest the pairing
typedef FUNCTION<REAL,REAL> FUNC;

typedef std::pair< std::pair<int, FUNCTION<REAL,REAL> >, std::pair<REAL,REAL> >
    trisection_parameter;

// the following routine returns an approximation to one of the roots
// of the function under consideration

// in order to use the iRRAM limit operator for FUNCTION<int,REAL>
// we formulate with a "trisection_parameter" as argument
REAL trisection_approx(const trisection_parameter& p_f_l_r)
{
  int  p           = p_f_l_r.first.first;
  FUNCTION<REAL,REAL> func = p_f_l_r.first.second;
  REAL a           = p_f_l_r.second.first;
  REAL d           = p_f_l_r.second.second;

  REAL result = d;
  while( !bound(d-a,p-1) )
  {
    REAL new_a = calc_a(a, d);
    REAL new_d = calc_d(a, d);

    REAL f_left_left = func(a);
    REAL f_left_right = func(new_d);
```

```
124     REAL f_right_left = func(new_a);
125     REAL f_right_right = func(d);
126 
127     REAL f_left = f_left_left * f_left_right;
128     REAL f_right = f_right_left * f_right_right;
129 
130     int test=choose(f_left<0,f_right<0);
131     if(test==1) // f_left<0: [a, new_d] is the next interval to be concerned
132       {
133         d = new_d;
134         result = d;
135       }
136     else        // f_right<0: [new_a, d] is the next interval to be concerned
137       {
138         a = new_a;
139         result = a;
140       }
141   }
142   return result;
143 }
144 
145 // now we add the limit process and do the necessary conversions between the types
146 REAL trisection(FUNCTION<REAL,REAL> f, const REAL& left,  const REAL& right){
147   FUNCTION<int,REAL> fp=bind_second(
148         bind_second(
149           FUNCTION<trisection_parameter,REAL>(trisection_approx),
150           std::pair<REAL,REAL>(left,right)),
151         f);
152   return limit(fp);
153 }
154 
155 // Interval structure to save the boundaries and signs of function values on the
      boundaries
156 struct intvl
157 {
158   REAL l;
159   REAL r;
160   bool vl;
161   bool vr;
162 
163   intvl(REAL il, REAL ir, bool ivl, bool ivr) : l(il), r(ir), vl(ivl), vr(ivr)
164   {}
165 } ;
166 
167 // Find an arbitrary x in (left, right), whose y value is not 0.
168 // Function values at n+1 positions (x) are calculated. At least one of them are not null.
169 // The absolute values of n+1 function values are saved in an array. Use function choose
      to check if they are
170 // greater than zero (not undefined). To do this, loop the array in the reverse order, use
       the k-th element
171 // and the 0 to k-1 elements, which are or-ed, as the two parameters of the function
      choose.
172 REAL Find_Non_null_point(FUNC iFunc, int iNumberOfNullPoints, REAL left, REAL right)
173 {
174   REAL diff = right - left;
175   std::vector<REAL> func_values;
176   std::vector<REAL> x_values;
177   for(int i=1; i<=iNumberOfNullPoints+1; ++i)
178   {
179     REAL x = left + i * diff / (iNumberOfNullPoints + 2);
180     REAL abs_value = abs(iFunc(x));
181     func_values.push_back(abs_value);
182     x_values.push_back(x);
183   }
184   std::vector<LAZY_BOOLEAN> tests(func_values.size());
```

```
185   for(unsigned int i=0; i < func_values.size(); ++i)
186   {
187     tests[i] = func_values[i] > 0;
188   }
189
190   std::vector<LAZY_BOOLEAN> new_tests(func_values.size());
191   new_tests[0] = tests[0];
192   for(unsigned int i=1; i<new_tests.size(); ++i)
193     new_tests[i] = new_tests[i-1] || tests[i];
194   for(unsigned int i=tests.size()-1; i>0; --i)
195   {
196    if (choose(new_tests[i-1], tests[i])==2)
197    return x_values[i];
198   }
199   return x_values[0];
200 }
201
202 // Find the n null points of the function in the interval (l, r).
203 // We search n sub intervals, which have only one null point.
204 // Then we use the algorithm 'trisection' to calculate the x value.
205 void null_points(FUNCTION<REAL,REAL> f, int n, REAL l, REAL r, int prec)
206 {
207   // use queue to store the interval information
208   using std::queue;
209   queue<intvl> q;
210   bool vl=f(l)>0,vr=f(r)>0;
211   // first item in queue: the initial interval
212   q.push(intvl(l,r,vl,vr));
213
214   // vzw: used to count the intervals with different sign of function values at boundary
215   int vzw=0;
216   if (vl!=vr) vzw++;
217
218   // number of null points which are not yet found
219   int numNullPoints = n;
220   while (vzw<n)
221   {
222     intvl z=q.front();
223     q.pop();
224
225     // compute m
226     REAL m = Find_Non_null_point(f, numNullPoints, z.l, z.r);
227     bool vm=(f(m)>0);
228     q.push(intvl(z.l,m,z.vl,vm));
229     q.push(intvl(m,z.r,vm,z.vr));
230     // count the change of signs ...
231     if (z.vl == z.vr && z.vl != vm)
232     {
233       vzw+=2;
234       numNullPoints -= 2;
235     }
236   }
237   // use trisection to calculate null points
238   int deci_places = prec;
239   while(q.empty()==false)
240   {
241     intvl item = q.front();
242     q.pop();
243     if(item.vl == item.vr)
244       continue;
245     REAL result = trisection( f, item.l, item.r);
246     cout << "result: ";
247     cout << setRwidth(deci_places+8);
248     cout << result << "\n" ;
249   }
```

```
250 }
251
252 void compute(){
253   int test, prec;
254   clock_t start, end;
255   cout <<
256   "Function selection: 1#cos(3x); 2#line; 3#sin(3x); 4#flach; 5#exp-flach;\n"
257   <<  " 6#sin(x); 7#parable(x); 8#x5; 9#x10; 10#x20; 11#x40; 12#x40a\n";
258   cin >> test;
259   cout << "Praezision: \n";
260   cin >> prec;
261   FUNCTION<REAL,REAL> f;
262   unsigned int numOfNullPoints;
263   REAL left_boundary, right_boundary;
264   switch (test) {
265       case  1: f= FUNCTION<REAL,REAL>(cos3x)    ; numOfNullPoints=1;  left_boundary=0;
              right_boundary=1;   break;
266       case  2: f= FUNCTION<REAL,REAL>(line)     ; numOfNullPoints=1;  left_boundary=0;
              right_boundary=1;   break;
267       case  3: f= FUNCTION<REAL,REAL>(sin3x)    ; numOfNullPoints=1;  left_boundary=0;
              right_boundary=1;   break;
268       case  4: f= FUNCTION<REAL,REAL>(flat)     ; numOfNullPoints=1;  left_boundary=0;
              right_boundary=1;   break;
269       case  5: f= FUNCTION<REAL,REAL>(exp_flat) ; numOfNullPoints=1;  left_boundary=0;
              right_boundary=1;   break;
270       case  6: f= FUNCTION<REAL,REAL>(sin)      ; numOfNullPoints=2;  left_boundary=-1;
              right_boundary=4;   break;
271       case  7: f= FUNCTION<REAL,REAL>(parable)  ; numOfNullPoints=2;  left_boundary=-3;
              right_boundary=5;   break;
272       case  8: f= FUNCTION<REAL,REAL>(x5)       ; numOfNullPoints=5;  left_boundary=-0.9;
              right_boundary=5.1; break;
273       case  9: f= FUNCTION<REAL,REAL>(x10)      ; numOfNullPoints=10; left_boundary=-2;
              right_boundary=50;  break;
274       case 10: f= FUNCTION<REAL,REAL>(x20)      ; numOfNullPoints=20; left_boundary=-20;
              right_boundary=20;  break;
275       case 11: f= FUNCTION<REAL,REAL>(x40)      ; numOfNullPoints=40; left_boundary=-2;
              right_boundary=2;   break;
276       case 12: f= FUNCTION<REAL,REAL>(x40a)     ; numOfNullPoints=40; left_boundary=-2;
              right_boundary=2;   break;
277       default: cout << "Wrong input!\n"; return;
278     }
279     start = clock();
280     null_points(f, numOfNullPoints, left_boundary, right_boundary, prec);
281     end = clock();
282     cout << "Zeit (s):" <<(float)(end - start)/(float)(CLOCKS_PER_SEC) << "\n\n\n";
283 }
```

Docs/nullstellensuche.cc