**שיטות מעורבות למציאת שורשים**

ניסיון ליצור שיטת שמשלבות שיטות בסיסיות לתוך מעין אלגוריתם הנהנה מהיתרונות של המרכיבים אינו עניין פשוט, מסתבר.

השיטות שיסקרו כאן הם שיטות שמסתמכות על שיטת החצייה על מנת להתקרב לפתרון ושיטות אחרות כאשר הקרבה הזו מושגת. למעשה האלגוריתמים הללו הם שאפתנים יותר. הן מתייחסות לשיטת החצייה כמוצא אחרון ומנסות להשתמש בשיטות מבטיחות יותר כאשר אלו מספקות קירוב טוב יותר משיטת החצייה, גם כאשר עדיין אין קירבה לפתרון. השיטות הקושי העיקרי הגדרת הקריטריון כיצד לבחור את בשיטה של החישוב הנוכחי שתבטיח ביצועים יותר טובים מאשר שיטת החצייה. למען האמת, גם השיטות שיסקרו לא יקיימו את היומרה הזו בכל מקרה, אבל הם בדרך כלל יקיימו אותו.

 מאחר והשיטות מסתמכות על שיטת החצייה, הן דורשות שיספקו להן אינטרוול ראשוני שבו פונקצית המטרה מחליפה סימן, והן אינן יכולות לפתור משוואות שרק נושקות לציר ה-X, בניגוד לשיטת ניוטון למשל.

**השיטה של דקר dekker**

 השיטה של דקר משלבת את שיטת החצייה עם שיטת המיתר.

הרעיון הוא כזה:

 יהי a ו-b קצוות נוכחיים של הקטע שבהם f(a)f(b) < 0.

לשם הפשטות נניח ש-f(b) > 0 ו-f(a) < 0 וכי |f(b)| < |f(a)| כלומר ש-b הוא ניחוש יותר טוב לפתרון (אם הפונקציה היא מונוטונית, למשל). במקרה כזה נקודת החיתוך s של המיתר את ציר ה-X קרוב יותר ל-b מאשר אמצע הקטע. אם f(s) קרוב מספיק לאפס סיימנו. אם f(s)< 0 אזי הקטע החדש יהיה (a,b) = (s,b). אחרת הקטע החדש יהיה (a,b) = (a,s). לדעתי במקרה זה האחרון אפשר גם לבחור את (a,b) = (a,m) בתור הקטע החדש. המקרים האחרים הם סימטריים.

התוכנית הבאה היא כנראה מימוש של שיטת דקר:

long double dekker( long double (\*fun)(long double), long double a,

long double b, long double eps)

{

 long double x, f, s, m, bk, bk1, funa, funb, funbk, funbk1;

 int flag;

 funa = (\*fun)(a);

 funb = (\*fun)(b);

 do {

 if ( fabsl(funa) < fabsl(funb))

 {

 bk1 = b;

 funbk1 = funb;

 bk = a;

 funbk = funa;

 flag = 1;

 }// if

 else

 {

 bk1 = a;

 funbk1 = funa;

 bk = b;

 funbk = funb;

 flag = 2;

 }// else

 m = (a+b)/2.0;

 s = bk - ((bk - bk1)/((\*fun)(bk) -(\*fun)(bk1)))\* (\*fun)(bk);

 if (flag == 1)

 if( (s >= a) && (s <=m))

 x = s;

 else

 x = m;

 else

 if( (s <= b) && (s >=m))

 x = s;

 else

 x = m;

 f = (\*fun)(x);

 if (fabsl(f) < eps)

 return x;

 if ( funa\*f < 0.0)

 {

 b = x;

 funb = f;

 }// if

 else

 {

 a = x;

 funa = f;

 }// else

 } while( fabs(b-a) > eps);

 return x;

} /\* dekker \*/

השיטה של דקר תעבוד היטב על פונקציות מונוטוניות או טבעיות אבל השיטה של דקר לא תמיד תהיה שיפור של שיטת החצייה. אפשר לתפור פונקציות קפריזיות כאלה שתמיד יתבצע חישוב נקודה חדש לפי שיטת המיתר והנקודות המחושבות (s) יהיו במרחקים מזעריים זה מזה וההתכנסות תהיה איטית יותר משיטת החצייה.

**השיטה של ברנט brent**

השיטה של ברנט היא שיטה שבה הוכנסו מספר שינויים לשיטה של דקר.

ברנט בחר שני תנאים נוספים שהוא דורש בכל איטרציה על מנת להעדיף את חישוב המבוסס על אינטרפולציה, כאשר אי השוויון המדויק תלוי בשאלה איך חושב הנקודה האחרונה. אינני נכנס לפרטים כי כלל לא ברור איך הוא הגיע אליהם.

התובנה מאחורי התנאים הללו הם שהנקודות החדשות צריכות להיות במרחק לפחות δ נבחר זה מזה (שלא יהיו קרובות מדי) ושהנקודות החדשות הם בתהליך של התכנסות.

בנוסף לכך, השיטה של ברנט משתמשת בשיטת ההיפרבולה ההפוכה כאשר הדבר אפשרי כלומר כאשר ה-f(xk) של שלושת הנקודות האחרונות הן שונות.

אפילו כל זה השיטה של ברנט עשויה להתכנס לאט יותר משיטת החצייה. ברנט הצליח להוכיח רק שאם שיטת החצייה תתכנס על מקרה נתון ב-N צעדים, השיטה שלו תתכנס ב-2N צעדים. יחד עם זאת ברור שכמעט תמיד השיטה תתכנס מהר יותר קרוב לפתרון.

להן כנראה מימוש של השיטה של ברנט:

long double brent(long double (\*fun)(long double),

long double a, long double b, long double eps)

{

 long double fa, fb, fc, fs, c, c0, c1, c2,temp, mtflag, d, s;

 int i, mflag;

 c = a;

 d = c;

 fa = (\*fun)(a);

 fb = (\*fun)(b);

 fc = (\*fun)(c);

 if ( fa\*fb >= 0)

 return 0.0;

 if ( fabsl(fa) < fabsl(fb))

 {

 swap(&a, &b);

 swap(&fa, &fb);

 } // if

 mflag = 1;

 while ( (fabsl(fb) > eps) && ( fabsl(b-a) > eps))

 {

 if ( (fa != fc) && fb != fc)

 {

 c0 = a\*fb\*fc/((fa-fb)\*(fa-fc));

 c1 = b\*fa\*fc/((fb-fa)\*(fb-fc));

 c2 = c\*fa\*fb/((fc-fa)\*(fc-fb));

 s = c0 + c1 + c2;

 } // if

 else

 s = b - fb\*(b-a)/(fb - fa);

 if ( ( s < (3\*(a+b)/4) || s > b) || ( (mflag == 1) &&

 fabsl(s-b) >= (fabsl(b-c)/2) ) ||

 ( (mflag == 0) && fabsl(s-b) >= (fabsl(c-d)/2) ) )

 {

 s = (a+b)/2;

 mflag = 1;

 } //if

 else

 mflag = 0;

 fs = (\*fun)(s);

 d = c;

 c = b;

 fc = fb;

 if ( (fa\*fs)< 0)

 b = s;

 else

 a = s;

 if ( fabsl(fa) < fabsl(fb))

 {

 swap(&a, &b);

 swap(&fa, &fb);

 } // if

 } // while

return b;

} /\*brent \*/