演算法：計算函數的極值

last modified April 23, 2008

計算函數的極值 (最大值或最小值) 一直是數學的應用上常見的問題, 其應用範圍非常廣泛, 幾乎涵蓋理工、商學的所有領域, 在微積分的教材中也是典型的範例, 而統計學中最大概似值 (Maximum likelihood Estimate) 的計算也是一例。一般而言, 極值的計算還是從函數的導數開始, 求其一次導數爲零的根, 不過當函數本身的微分很困難或甚至函數本身都不可得時, 數值演算 (numerical methods) 的方法就顯得重要。本練習將只針對簡單的函數做練習, 並與函數圖形配合, 了解數值計算的方法、精神與一些細節。待熟練後, 複雜的函數也可以依樣畫葫蘆。

本章將學到關於程式設計

演算法的程式寫作與除錯、迴圈中斷的技巧與匿名函數的應用。

⟨本章關於 MATLAB 的指令與語法⟩

指令:min, polyder, roots, fminbnd, inline, linspace, num2str

1 背景介紹

假設函數爲

$$f(x) = x^4 - 8x^3 + 16x^2 - 2x + 8 \tag{1}$$

如圖1所示, 該函數包括一個區域 (local) 最大值、一個區域最小值及一個全域 (global) 最小值。計算函數之極值通常先求其一次導數的根, 相當於函數斜率爲零的位置, 也就是圖形中波峰或波谷的位置, 或說區域極值 (local extreme) 的位置。當某區域極值實爲所有極值之最時, 稱爲全域極值。極值分最大與最小, 可從其二次導數的正負區別之。

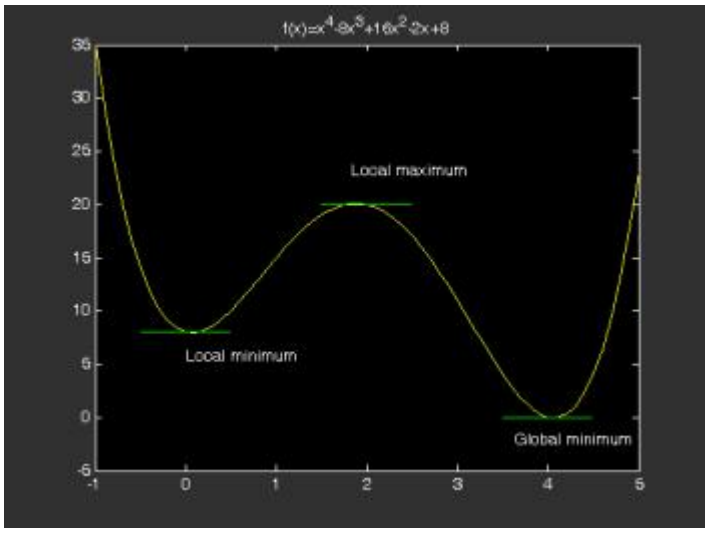

圖 1: 函數的極值

一個函數的極值如果可以透過紙筆演算得到答案, 即所謂的 analytical solution 或 closed-form solution, 自然不需以電腦的數值方法解決, 所得到的答案當然是完美的。但是當函數複雜或囿於數學演算能力之不足, 便需要藉助電腦以其快速的計算能力, 配合事先定義好的演算法則, 找到那些甚至連函數圖形都畫不出來的極值 (函數繪圖的極限是兩個變數。) 雖然通常必須犧牲完美精準的答案, 但在可接受的範圍內, 也算是不得已的良策。本單元利用上述簡單的多項式函數, 介紹利用演算法計算極值的一個典型。觀念很簡單, 但透過電腦強大的計算能力, 其威力無窮, 再複雜的問題也常能迎刃而解。

電腦畢竟不如人腦, 可以拐彎抹角的思考。利用電腦解決複雜問題時, 需要給予明確的步驟指示, 讓電腦按部就班的執行。這些步驟一般稱爲演算法 (algorithm), 定義電腦從何處開始、如何執行及何時結束。演算法也可以說是寫程式前的前置步驟, 程式的細節不在此表達, 只是點出精神與方向。以下舉出典型的計算區域最小值的演算步驟:

1. 找一個起始值 x_1, 並計算 $f(x_1)$
2. 在第 k 步 (已知 x_k), 計算 x_{k+1}, 使得 $f(x_{k+1}) < f(x_k)$, 其中$k = 1, 2, \cdots$
3. 重複步驟 2, 直到 $|f(x_{k+1}) - f(x_k)| < \epsilon$

其中 ϵ 爲一預設的、通常很小的誤差值 (譬如 10^{-5})。其值的大小需依函數的不同調整, 有時候也會使用相對的誤差值, 即標準化的值, 如

$$\frac{|f(x_{k+1}) - f(x_k)|}{|f(x_k)|} < \epsilon$$

上述的演算法的精神爲

從函數的一個已知的位置 (x_k) 去找下一個比它低的點 (x_{k+1}), 直到谷底。

看來簡單, 其實變化很大, 尤其在步驟 2 對於如何找尋『下一個低點,』更是可以衍生出許多的方法。演算法只是提綱挈領, 其細節則是變化萬千。簡單的說, 如何在步驟 2 的 x_k 位置, 找到下一個點 x_{k+1} 使得 $f(x_{k+1}) < f(x_k)$, 即

$$x_{k+1} = x_k + d_k, \qquad k = 1, 2, \cdots$$

d_k 代表從 x_k 移動的方向, 其『正負』與『大小』代表移動的『方向』與『距離』。關於 d_k 的選擇, 以下是一些建議:(請透過作業的引導瞭解其正當性)

- $d_k = -f'(x_k)$
- $d_k = -\frac{f'(x_k)}{f''(x_k)}$

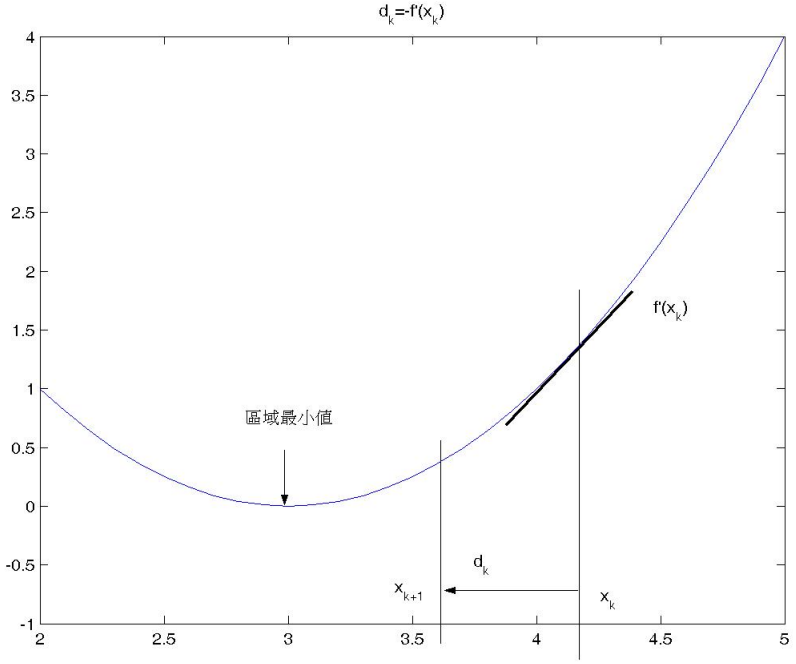

圖 2: steepest descent 的移動方向

$d_k = -f'(x_k)$ 選擇了朝著「負斜率」的方向, 從函數圖來看, 那是一個「下坡」的方向, 如圖2所示。這個選擇一般稱爲 steepest descent。當斜率爲 0 代表到達了山谷(最小值),x_k 便不再移動。第二個選擇則是加進了二次導數的影響, 稱爲牛頓法。在理論上, 其往最小值移動的速度比 steepest descent 快, 值得玩味的是, 二次導數的正負還會影響移動的方向。不管如何, 在運用上述演算法計算極值前, 先來看看還有什麼比較簡單的想法與作法, 再與 MATLAB 提供的方式 (指令) 比較, 最後回到上述演算法對於 d_k 的移動速度與方向的探討。

2 練習

範例1: 求極值的方法很多, 對於單純如式(1) 的單一變數函數,Direct Search(或稱 Grid Search) 是個直覺的可行方式。它是利用將變數在一定的範圍內分割成等距的格子點 (grids), 分別計算每個格子點對應的函數值, 最後再從中挑選最小者當作是函

數的最小值。試著利用這個方法估計函數 (1) 的最小值。

所有 X 軸上格子點的函數值計算, 類似之前的單元介紹函數圖的描點繪製法, 最後再以 MATLAB 指令 min 來搜尋函數的最小值, 作法如下:

$p = [1\ -8\ 16\ -2\ 8];$
$x = -1 : 0.00001 : 5;\ \%$ 切割變數空間成格子點
$y = polyval(p, x);\ \%$計算所有格子點的函數值
$[Y, I] = min(y);\ \%\ I$代表最小值所在的索引位置
$global_min_ds = x(I)\ \%$最小值爲 Y, 發生在 $x = x(I)$。

很明顯的, 最小值的準確度與格子的「粗細」相關, 當然間隔太細, 計算量變大, 但在電腦速度日益倍增的情況下, 這個方法也不錯。請注意第四行 min 的用法, 其中結果 Y 代表向量 y 裡面所有元素的最小值, 而 I 則代表最小值出現的位置。利用這個位置找到相對應的 x 值 (第五行)。

當要求的精準度 (precision) 很高或是函數本身的計算比較費時, 這種一網打盡的方式往往不能被接受。另一種可行的方式是鎖定「可能的區域,」不做無謂的計算。所謂「可能的區域」指最小值可能出現的區域。其中最簡單的方式是先將變數空間切割成若干等分, 各區域選一個點來計算函數值, 取其最小者代表該區域存在函數的最小值。接著便排除其他區域, 在選定的區域裡面再等分成若干區域, 依上述的方式再選定一個更小的區域, 如此類推, 直到需求的精準度爲止, 下列程式寫出這個概念。

$p = [1\ -8\ 16\ -2\ 8];$
$k = 4;\ \%$ 多層次 grid search 的層次
$a = -1; b = 5; n = 11;\ \%$ 變數空間設定爲 $[a\ b]$ 切割成 $n - 1$ 等分
$for\ i = 1 : k$
    $x = linspace(a, b, n);\ \%$格子點的位置
    $y = polyval(p, (x(1 : n - 1) + x(2 : n))/2);$
    $[Y, I] = min(y);$
    $a = x(I); b = x(I + 1);\ \%$決定下一個層次的變數空間
end
$global\_min\_ds = (a + b)/2$ 。

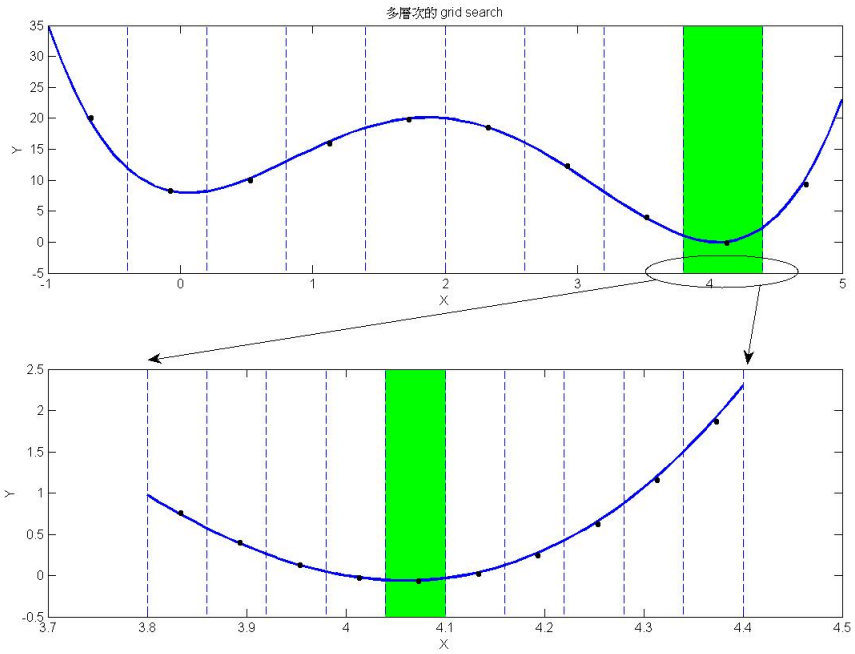

圖 3: 多層次grid search 的第一層與第二層

圖3展示了這種多層次 grid search 前兩層的概念。程式第 2 行的變數 k 代表層次, k 愈大表示精準度愈高。第 5 行的指令 linspace 在 a 與 b 的範圍內均等地產生 n 個點的向量 x。上述的程式將 $(-1, 5)$ 的範圍均等切分成 10 個等分, 每個區域取中間點計算函數值 (第 6 行)。第 7與 8 行取這些函數值最小者的區域做爲下一個 grid search 的範圍。如此逐步縮小 grid search 的範圍, 達到最少的函數計算, 並保有一定的精準程度。

這個做法雖然減輕不少計算的負擔, 但也有其風險, 使用時必須非常小心, 才能正確決定每次切割的區域數, 避免遺漏最小值的可能。

範例2: 求極值的方法很多, 可採計算一次導數爲零的方程式的根, 也就是前一個單元的演算法則。先將函數的一次導數寫出來, 再套入前一個單元的程式即可。這個方法適用於一次導數, 甚至二次導數 (求方程式的根也需要做一次導數) 已知的情況下。請利用指令roots 計算函數 (1) 的最小值。

作法如下

```
p = [1 −8 16 −2 8];                % f(x) 的係數
pp = polyder(p);                   % f'(x) 的係數
x = roots(pp);                     % f'(x)=0的根
[Y, I] = min(polyval(p, x));
global_min_roots = x(I)
```

其中 $polyder$ 可以從多項式的係數產生其一次導數的係數。利用多項式一次導數的係數配合 $roots$ 指令, 找出多項式導數爲零的位置 (x 值), 再利用 min 指令找出最小的函數值。

上述程式僅適用於多項式函數, 因指令 roots 可以找出所有的根。而計算一般函數根的指令 fzero 只能找出一個根, 必須配合適當的程式手段才能找到所有的根, 並從中找出函數值最小者。下列程式可供參考。

```
p = [1 −8 16 −2 8];                % f(x) 的係數
pp = polyder(p);                   % f'(x) 的係數
f = @(x)polyval(pp, x);                % 函數 f'(x) 定義
x = −1 : 5;                        % 選擇適當的變數空間
r = [ ];                           % 定義空值向量, 準備放置根值
for i = x
      r = [r  fzero(f, i)];   計算在 i 附近的根並逐步併疊至 r 向量
end
[Y, I] = min(polyval(p, r)); %計算所有根的函數值後, 求最小值
global_min_roots = r(I)
```

在進行新的嘗試前, 最好從簡單已知的問題開始。在將演算法以程式來表達前, 若先知道答案會對程式的除錯與判斷有幫助。待程式可以解決這個已知的簡單問題時, 再慢慢擴大到比較複雜且未知答案的問題上, 千萬別急著一開始便挑戰複雜度高的問題, 那無疑是找自己的麻煩, 對於程式寫作能力的提升也沒有幫助, 卻是一般初學者常犯的毛病, 切記!切記! 這個簡單的範例與程式提供一個好的開始。

範例3:MATLAB 也提供了 fminbnd 計算非線性函數的最小值, 試著計算函數 (1) 的最小值。

作法如下:

以下適合 MATLAB 7.x 版以上

$p = [1\ -8\ 16\ -2\ 8];$ % f(x) 的係數

$f = @(x)polyval(p, x);$ % 定義函數 f(x)

$global\_min\_fmin = fminbnd(f, -1, 5)$%在[−1, 5]的範圍內找最小值

以下適合 MATLAB 7.x 版以前

$p = [1\ -8\ 16\ -2\ 8];$

$f = inline('x\text{^}4 - 8 * x\text{^}3 + 16 * x\text{^}2 - 2 * x + 8');$

$global\_min\_fmin = fminbnd(f, -1, 5)$

未來在應用上盡量使用現成的指令, 以節省時間。不過現成指令畢竟受限使用的範圍, 不可能兼顧所有狀況 (函數), 必要還是得運用本單元介紹的演算方法。

範例4: 應用第 1 節的演算法, 寫程式計算函數 (1) 的一個區域最小值。先嘗試 $d_k = -f'(x_k)$ 的選擇, 看看是否朝著目標前進。成功了, 再試著採用牛頓法的方向。能成功的找到一個極值之後, 是否也能找到另一個? 初始值的決定是否影響所找到的極值, 都值得細細的觀察與一再的試驗。

程式的寫作宜從簡而繁, 由陋而精。在過程中, 必須經常測試, 最好列印出過程中的某個值或畫某個函數圖來檢視程式是否正確 (如圖 4)。然後才逐步修正往精簡路線調整。換句話說, 先求正確, 再求精美。[1]如此一來, 程式的執行出了狀況, 才知道是程式寫錯

[1]好程式具備幾項特點: 效率、易讀、精簡。其中「易讀」與「精簡」兩項時有衝突, 如何拿捏沒有一定的標準。本文作者偏向程式的易讀性, 認爲程式常有機會與他人分享, 只要效率不太差, 還是盡量寫清楚, 除了他人容易閱讀外, 也方便自己除錯。而精簡的程式碼常常是玩弄些巧妙的想法, 自娛罷了。

了、還是演算法搞錯了、或是觀念還不清楚。如果都不清楚、不能掌握, 就只能兩手一攤, 無力回天。

有時候我們看到一支寫好的複雜程式, 常驚嘆作者的功力。其實, 一支程式的完成需經歷反覆的修改, 一開始的模樣與最後的作品很難聯想在一起, 如同一幅油畫作品的創作, 從鉛筆草圖到顏色的塗塗抹抹, 一層又一層。所以建議程式寫作者下手不必太拘泥, 這不是中國的山水潑墨畫, 一提筆片刻間即完成, 旁人不解, 畫家說:「別看我作畫的時間短, 我可想了好幾年呢。」常見初學者望著螢幕久久不能敲下指令, 往往是想太多, 想要下手就是正確無誤。其實不然, 寫程式的過程常常是從中段寫起, 再補前補後穿插缺漏的指令或修正錯誤的地方。有經驗者可能從最難的那一段寫起, 只要沒把握的部分完成了, 其他的只是時間的問題。而初學者可以從最簡單的地方寫起, 先在空白的地方填起一行行指令, 建立信心, 再慢慢擴增指令「版圖。」

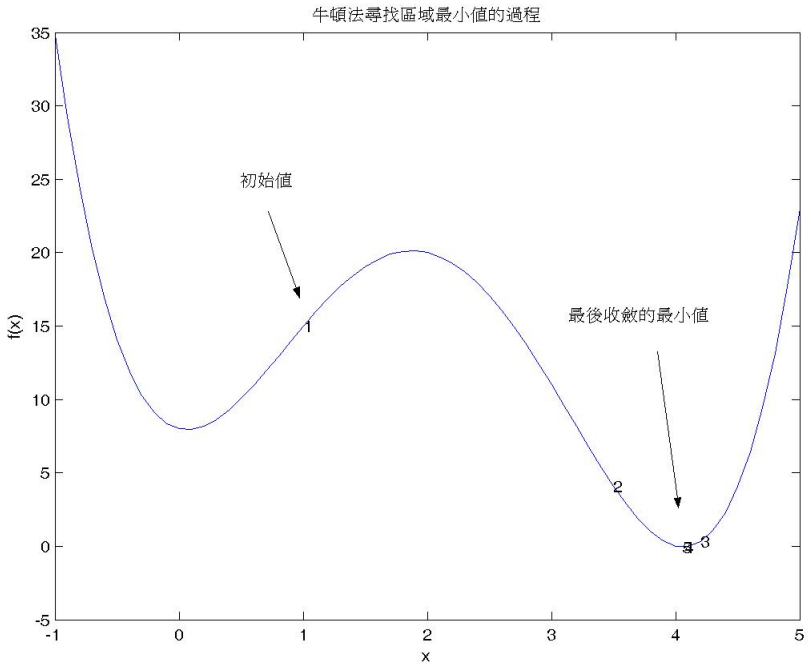

圖 4: 牛頓法尋找區域最小值的過程

在之前「簡單的演算法: 以計算方程式的根爲例」的單元中, 提到初學者程式寫作的技巧, 從模仿演算法的數學式開始。以下的程式碼可以作爲一個開始:

```
p=[1 -8  16  -2  8];
f=@(x) polyval(p,x);% 定義函數 f(x)
pp=polyder(p);
fp=@(x)polyval(p,x); % 定義函數 f'(x)
xk=4.5; % 初始值
dk=-fp(xk);% steepest descent 向量
xk1=xk+dk;
```

上述程式執行的結果常令初學者感到不知所措, 這個 steepest descent 的方向, 竟然得到一個很離譜的函數值 f(xk1), 比 f(xk) 大多了。難道是演算法錯了? 還是程式寫錯了? 初學者常因經驗不足便困在這裡。仔細觀察 dk 的值爲 -20.5, 負號所代表的方向是對的, 但是數值太大了, 導致落點的 xk1 的函數值比 xk 的函數值還高。程式沒錯, 但演算法不夠完整, 考慮不周詳。

當實際採用上述的 d_k 進行最小值的逼近時, 有時候無法收斂到區域的最小值, 反而是發散出去、或發生在兩點間震盪、或甚至收斂到區域的最大值去了。最常見的情況是, 移動方向對了, 可是步伐卻太大了。本該往低處移動的, 卻「一飛沖天」反而到高處去了。因此爲了調節 d_k, 保證往低處移, 在選擇下一個點時, 採下列的「管制措施」

$$x_{k+1} = x_k + \beta d_k$$

β 有時被稱爲「步伐調節器」。選擇適當的 β 值 (通常 $0 < \beta < 1$), 以確保 $f(x_{k+1}) < f(x_k)$, 甚至當 $\beta > 1$ 時還可以加速收斂的速度。

β 值對於步伐的調節常扮演重要的角色。程式是否收斂或收斂的快慢與 β 的選擇息息相關。關於 β 的選擇非常多樣, 最簡單是所謂的 $bi - section$ 法。也就是:

$$\beta = (\frac{1}{2})^{\alpha} \qquad \alpha = 0, 1, 2, \cdots$$

即一開始 $\beta = 1$ (即 $\alpha = 0$) 代表走一個 d_k 的距離, 如果發現太大步導致 $f(x_{k+1}) > f(x_k)$, 則改走 $1/2$ 個 d_k 的距離 (即 $\alpha = 1$), 再檢查一次, 如果還太大步, 則改走 $1/4$ 個 d_k 的距離 (即 $\alpha = 2$), 以此類推逐步減半縮回, 直到 $f(x_{k+1}) < f(x_k)$ 爲止。

β 的選擇讓每次都縮回原來的一半, 因此稱爲 $bi-section$ 法。另外, 也可以選擇一個奇妙的數字

$$\beta = (\frac{\sqrt{5}-1}{2})^{\alpha} \qquad \alpha = 0, 1, 2, \cdots$$

$(\sqrt{5}-1)/2$ 稱爲黃金比例 (Golden Ratio), 這個選擇也許沒太多道理, 既然縮回來可以按 $1/2$ 的比例, 爲什麼不能是神奇的黃金比例呢? 試試看, 它是否展現神奇的效果。

步伐調整器 β 的加入使得程式必須在原有的迴圈內再加上一個「內迴圈。」以下的程式片段可以做爲參考

```
N = 10;                                  %設定最多縮回來 (調整)10次
beta = 0.5.^(0 : N − 1);                 %設定 N 個縮回的比例,
while 1
    dk = fp(xk)                          %計算移動方式
    for j = 1 : N                        %內迴圈, 負責步伐的調整,
        xk1 = xk + beta(j) ∗ dk          %步伐調整: 逐次縮回
        if f(xk1) < f(xk), break, end    %步伐調整是否滿意,
    end                                  %如果滿足則跳出內迴圈。
    if 停止條件                           %stopping criterion: 是否已達最小值
        break;                           %滿足停止條件, 退出外迴圈
    else
        xk = xk1                         %接受新的點, 進行下一次遞迴
    end
end
```

上述程式中, 內迴圈的迴圈數N預設爲10, 表示最多退回10次, 對於某些函數而言並不恰當。適當的調整或乾脆放大一點都是實務上的作法。至於程式中的「停止條件」也是一大學問, 在下一節有進一步的說明。這裡可以先用下列的方式:

```
if abs((f(xk1)-f(xk))/f(xk)) < 10^(−5)
```

程式寫作過程必須隨時檢測寫下的指令是否正確, 這包括語法、邏輯與自己設定的目標。當程式可以正確執行, 但結果卻與預設的目標有出入時, 表示語法正確了, 但是或許是邏輯錯了, 或是觀念搞錯了, 有可能只是數字打錯了, 或數學推導錯了, 這時便開

始進入除錯 (Debug) 的階段, 這才是初學者最大的考驗, 過了這一關才稱得上「會寫程式。」以上述程式爲例, 程式執行不需一秒時間, 如果結果錯了該如何檢測呢?
寫作者可以利用程式技巧, 在程式過程中安排除錯的指令, 譬如印出某個變數的結果, 或是利用程式軟體提供的除錯工具。MATLAB的除錯工具列如圖5所示的 7 個工具按鈕。

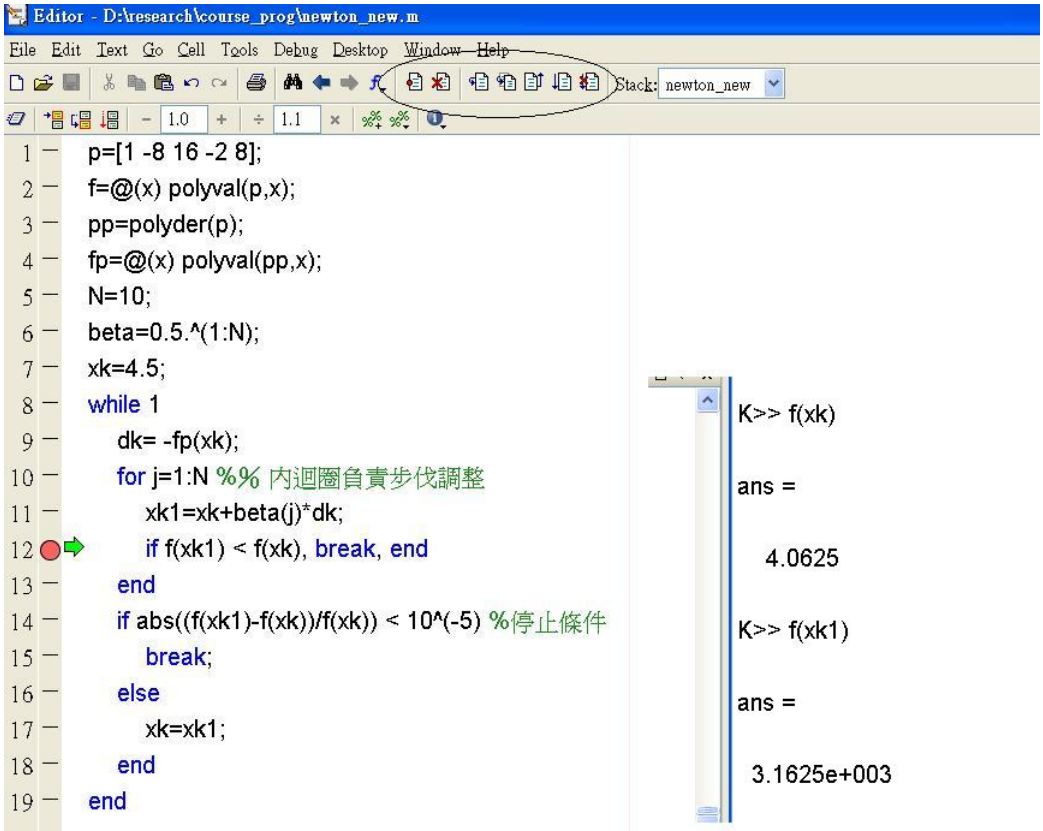

圖 5: MATLAB 編輯器的除錯工具列

除錯工具的精神在「暫停程式的執行, 方便觀察過程中的結果。」其方式是利用編輯器上的 7 個工具按鈕, 進行 (由左而右)

- 設定/取消停止點 (Set/clear breakpoint): 設定游標所在的那一行 (指令) 爲停止點, 程式將執行至前一行並停留在這一行。如圖5的紅色圈圈所示。另一個設定停止點的方式是利用滑鼠點選目前圈圈所在的位置即可。再點選一次即取消。當程式執行至停止點時, 會出現綠色指標, 此時命令視窗出現如圖右邊所示

的除錯模式「K>>」, 在這裡可以輸入任何指令來察看任何想知道的結果, 譬如圖5所示的察看函數值 f(xk) 及 f(xk1) 的大小。

- 取消所有停止點 (Clear breakpoints in all files): 用來重新調整停止點。
- 執行下一個指令 (Step): 每按一次會往下執行一個指令, 綠色指標會跟著移動。
- 進入下一個指令的內部 (Step in): 按下這個按鈕會進入下一個指令的內部。MATLAB 的每個指令本身就是一個副程式, 這個按鈕會進入這個副程式中。如果想學習 MATLAB 程式的精湛程式技巧, 這倒是個不錯方式。
- 退出指令內部 (Step out): 同前, 但退出副程式。
- 繼續執行程式直到下一個停止點 (Continue): 從綠色指標的位置執行至下一個停止點。
- 退出除錯模式 (Exit debug mode): 退出除錯模式。

除錯的技巧是門學問, 顯非在此能盡述。初學者多嘗試多觀摩, 應該可以從中獲得寫作程式的樂趣, 並且可以大膽的說: 只有寫不出來的程式, 但沒有寫不對的程式。

範例5: 利用匿名函數的方式繪製函數

$$f(x) = 2e^{-x}\sin(2\pi x) - 0.5 \qquad -1 < x < 1$$

作法如下:

```
x=-1:0.01:1;
f=@(x)2*exp(-x).*sin(2*pi*x)-0.5;
plot(x,f(x))
```

當自己寫程式利用各種演算法計算極值時, 經常需要計算函數值 (Objective Function)。如果該函數比較龐雜, 指令寫起來往往冗長, 又需要重複放在程式的不同地方。這對一個有經驗有訓練的程式員而言是痛苦不堪且看不下去的方式, 因此常會採用副

程式的方式來專門計算該函數值。但這又製造另一個麻煩, 就是多一個檔案要維護。這時候 MATLAB7.x 提出的匿名函數便顯得可愛許多。同樣是一條指令, 清爽乾淨, 且不需費心去建立及維護一個新檔案, 畢竟檔案管理也是程式員要去關心與在意的。

3 觀察

1. 一個函數的極值通常區分爲兩種: 區域性 (local) 及全域性 (global)。換句話說, 函數圖形上的『波峰』與『波谷』都是極值所在。一般的應用通常是找所有峰頂的最高或所有谷底的最低, 也就是全域極值。但目前對於全域極值計算的相關演算法通常無法擺脫區域極值。這是一個幾乎無解的題目, 目前大部分的研究都還是很依賴從該問題的本身尋求其他線索。單純從數學的角度來看, 對於函數本身較爲『崎嶇』的問題而言, 大概找不到萬無一失的解法。

2. 試試你寫好的程式, 從不同的起始點出發, 看看得到的答案是否不同。其實許多難解的問題最後都是依靠一個好的起始值才順利找到極值。起始值的選擇對某些函數而言並不重要, 但對某些函數卻很『敏感』。特別是想找到全域極值的問題, 一個好的起始值是成功的一半。通常起始值的選擇會從問題本身的特性去找尋。如何選擇好的起始值也是很重要的研究課題, 特別當演算法本身已經無能爲力時。

3. 不論是 steepest descent 或是牛頓法的移動方向, 都有其限制 (見作業), 並不能保證一定移往極值的方向。一些調整的手段是必要, 特別當函數比較特殊或是變數增多時, 更是常見。本單元並不討論這些問題, 有興趣者很容易可以找到相關的研究議題。

4. 演算法中經常要設定迴圈的停止條件; 譬如, 當找到「滿意」的最佳值時, 或「步伐」調整到「可以接受」的情況, 否則將按一定的法則繼續演算下去。在上述的練習中建議的條件爲
$$\frac{|f(x_{k+1}) - f(x_k)|}{|f(x_k)|} < \epsilon$$
是當函數 f 隨著 x 的改變已經呈現「穩定」的時候, 則可以假設已到達「谷底」的最佳值。但某些函數在接近「谷底」時, 呈現非常「平坦」的趨勢, 函數值變化

很微小, 看似已接近谷底, 其實還有一大段距離, 如圖6所示, 當 x 值從 4 移動到 6 時, f 改變還是非常小。若使用上述的停止條件, 容易停在「半路」誤以爲已到谷底。遇到這樣的情況, 停止條件可以作適當的調整, 譬如改以 x 值的變化做爲是否到達谷底的依據, 如

$$\frac{|x_{k+1} - x_k|}{|x_k|} < \epsilon$$

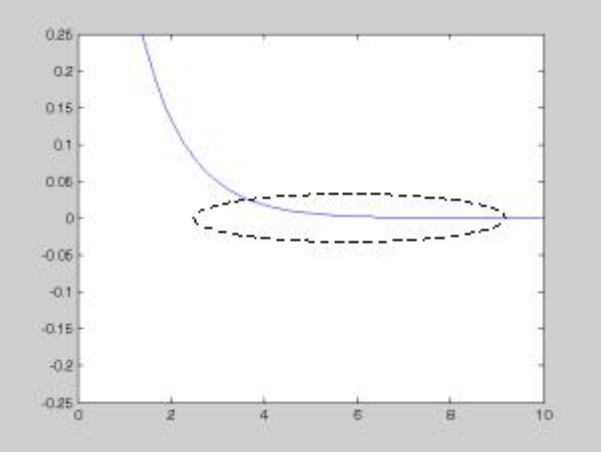

圖 6: 當函數的極值位於「平坦」區域時

4 作業

1. 將之前介紹的演算法更詳細的寫出來, 包括把 x_{k+1} 明確的表示出來, 並加入步伐調整器 β 的選擇及跳出迴圈的條件。

2. 寫一支程式求 $f(x) = x^4 - 8x^3 + 16x^2 - 2x + 8$ 的幾個區域極值。(注意起始值的選擇)。將過程的演進表達出來, 譬如將數字 $1, 2, 3, \cdots$ 寫在函數圖上 (如圖4利用 text 指令), 代表 $x_1, x_2, x_3, \cdots$ 的位置。執行中更可以利用 pause(秒數) 來觀察程式走勢。方式如下

```
text(xk,f(xk),num2str(i));
pause(1)
```

其中第三個參數利用 $num2str$ 指令, 將數字 i 轉爲語法可以接受的字串型態。

3. 自行找一個非多項式的函數, 利用 fzero, fminbnd 及本單元介紹的 Steepest Descent 等方法, 分別計算其最小值。

4. 關於 $d_k = -f'(x_k)$ 的選擇, 可以從下列的泰勒展開式得到

$$f(x_{k+1}) = f(x_k + d_k) = f(x_k) + f'(x_k)d_k + O(d_k^2)$$

假設 d_k 夠小, 足以忽略 d_k^2 以後的項目, 請證實 $d_k = -f'(x_k)$ 的選擇可以保證 $f(x_{k+1})$ 比 $f(x_k)$ 低 (小)。

5. 同上的泰勒展開式, 但多取一項 (忽略 d_k^3 以後的項目)

$$f(x_{k+1}) = f(x_k + d_k) = f(x_k) + f'(x_k)d_k + \frac{f''(x_k)d_k^2}{2!} + O(d_k^3)$$

證明

$$\min_{d_k} f(x_{k+1})$$

的解爲

$$d_k = -\frac{f'(x_k)}{f''(x_k)}$$

6. (optional) 限制式極值 (Constraint Optimization): 計算下列函數的極小值

$$f(x) = 2e^{-x}\sin(2\pi x) - 0.5 \qquad\qquad -1 < x < 1$$

本單元並未針對限制式極值做任何討論, 在沒有參考任何書籍前, 不妨自己試著想想看, 嘗試些直覺的方法, 最後再求諸關於限制式極值的做法。千萬別急著找參考資料, 這會阻斷自己思考的機會, 無法進一步提升解決問題的能力。

7. Maximum Likelihood Estimator: 計算函數的極值常見於最大概似估計 (MLE) 的問題, 這類問題的目標函數通常不「單純,」有別於本單元見到的函數, 因此有必要讓讀者熟悉不同函數的極值計算。問題如下所述:

假設有 N 個樣本 $\{x_1, x_2, \cdots, x_N\}$, 已知來自一指數分配 $exp(\lambda)$, 但參數 λ 未知。

(a) 請寫出參數 λ 的 MLE 估計問題, 即

$$\hat{\lambda}_{MLE} = arg \max_{\lambda} \log l(\mathbf{x}|\lambda)$$

其中 $l(\mathbf{x}|\lambda)$ 稱爲概似函數或已知資料的聯合機率密度函數, 請清楚的以機率密度函數 $f(x_i|\lambda)$ 寫出來。

(b) 上述的概似函數需要樣本才能計算其值, 在此以模擬的方式產生資料, 即先設定一個 λ 值 (譬如 , $\lambda = 2$) 藉此產生 N 個具 $exp(\lambda)$ 分配的亂數做爲樣本, 其中的樣本數也需要事先設定, 譬如 $N = 30$。

(c) 利用本單元介紹的幾種計算極值的方法, 進行 λ 的 MLE 估計。

(d) 觀察估計值與設定值間的差異是否與樣本數 N 有關, 試著改變 N, 紀錄 N 與相對的估計值並繪圖或製表。

(e) 觀察估計值與設定值間的差異是否與採用的演算法相關? 是否與設定的母體參數 λ 相關?