**Runge-Kutta Methods**

陳光悦

假設我們要解 的微分方程。

移項過後可得 。

令 ，則 可視為 在 點時的斜率。

Runge-Kutta即利用 在不同點的斜率，推算出最接近函式 的原始樣貌。

若我們已知 時 。則可根據 時的斜率（），推出下一個點的位置（ 時 ）。

上述方法即為1階的Runge-Kutta Method。

公式如下：

其中，

2階的Runge-Kutta Method，則是比1階的Runge-Kutta Method還多了一個參考點來校正斜率，此後才去推函式 的原始樣貌。其準確度會比1階的Runge-Kutta Method還要精準。

即根據已知條件 時 ，推得 時的斜率為 ，以此斜率推出下一個點，得到 時 ，再根據此點得出一個新的斜率（）。將這兩個斜率取平均值 。此時再退回原點（），以斜率 推出下一個點的位置（ 時 ）。

公式如下：

其中，

4階的Runge-Kutta Method，則是比2階的Runge-Kutta Method還多了兩個參考點來校正斜率（即用4個點來推得最終斜率），此後才去推函式 的原始樣貌。其準確度會比2階的Runge-Kutta Method還要精準。

即根據已知條件 時 ，推得 時的斜率為 ，以此斜率向下一點走半步，得到 時 ，再根據此點得出一個新的斜率（）。此時再退回原點（），以斜率 依然只是向下一點走半步，得到 時 ，此時會再得到一個新的斜率 。再退回原點（），以斜率 推出下一個點的位置（ 時 ），又會再得到一個新的斜率。將四個斜率取平均 （中點的斜率有更大的權值）。此時再退回原點（），以斜率 推出下一個點的位置（ 時 ）。

公式如下：

其中，

實踐程式碼如下：

#include <stdio.h>

#include <math.h>

int main(){

int i, num;

double h, k\_1, k\_2, k\_3, k\_4;

double x[1001], y\_1[1001], y\_2[1001], y\_4[1001];

num = 1000;

h = 1.0/num;

y\_1[0] = 1.0;

y\_2[0] = 1.0;

y\_4[0] = 1.0;

for(i=0; i<=num; i++) x[i] = i \* h;

for(i=0; i<=num; i++){

//1st order

k\_1 = -(y\_1[i]);

y\_1[i+1] = y\_1[i] + h\*k\_1;

//2nd order

k\_1 = -(y\_2[i]);

k\_2 = -(y\_2[i] + h\*k\_1);

y\_2[i+1] = y\_2[i] + (h/2)\*(k\_1+k\_2);

//3rd order

k\_1 = -(y\_4[i]);

k\_2 = -(y\_4[i] + (h/2)\*k\_1);

k\_3 = -(y\_4[i] + (h/2)\*k\_2);

k\_4 = -(y\_4[i] + h\*k\_3);

y\_4[i+1] = y\_4[i] + (h/6)\*(k\_1 + 2\*k\_2 + 2\*k\_3 + k\_4);

}

for(i=0; i<=num; i+=50) printf("%.3lf\n RK1:\t%.16lf\n RK2:\t%.16lf\n RK4:\t%.16lf\ne^(-x):\t%.16lf\n\n", x[i], y\_1[i], y\_2[i], y\_4[i], exp(-x[i]));

return 0;

}

以每50點印出一次結果（如下），可以看出1階的Runge-Kutta Method可精準至小數點後3位；2階的Runge-Kutta Method可精準至小數點後6位；而4階的Runge-Kutta Method則可精準至小數點後14位。

 

