Runge-Kutta Methods

陳光悅

假設我們要解 $y'(x)+y(x)=0$ 的微分方程。
移項過後可得 $y'(x)=-y(x)$。
令 $f(y(x))=-y(x)=y'(x)$，則 $f(y(x))$ 可視為 y 在 x 點時的斜率。

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

若我們已知 $x=0$ 時 $y=1$。則可根據 $x=0$ 時的斜率（-1），推出下一個點的位置（$x=0.1$ 時 $y=1+(0.1*-1)=0.9$）。
上述方法即為 1 階的 Runge-Kutta Method。
公式如下：

$$y_{i+1}=y_i+h*k_1$$

其中，

$$k_1=f(x_i,y_i)$$

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 y 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 $x=0$ 時 $y=1$，推得 $x=0$ 時的斜率為 -1，以此斜率推出下一個點，得到$x=0.1$ 時 $y=1+(0.1*-1)=0.9$，再根據此點得出一個新的斜率（-0.9）。將這兩個斜率取平均值 $\frac{-1+(-0.9)}{2}=-0.95$。此時再退回原點（$x=0$），以斜率 -0.95 推出下一個點的位置（$x=0.1$ 時 $y=1+(0.1*-0.95)=0.905$）。
公式如下：

$$y_{i+1}=y_i+\frac{1}{2}*h*(k_1+k_2)$$

其中，

$$k_1=f(x_i,y_i)$$
$$k_2=f(x_i+h,y_i+h*k_1)$$

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 y 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 $x=0$ 時 $y=1$，推得 $x=0$ 時的斜率為 -1，以此斜率向下一點走半步，得到$x=0.05$ 時 $y=1+(0.05*-1)=0.95$，再根據此點得出一

個新的斜率（-0.95）。此時再退回原點（$x = 0$），以斜率 -0.95 依然只是向下一點走半步，得到$x = 0.05$ 時 $y = 1 + (0.05 * -0.95) = 0.9525$，此時會再得到一個新的斜率 -0.9525。再退回原點（$x = 0$），以斜率 -0.9525 推出下一個點的位置（$x = 0.1$ 時 $y = 1 + (0.1 * -0.9525) = 0.90475$），又會再得到一個新的斜率$-0.90475$。將四個斜率取平均 $\frac{-1+2*(-0.95)+2*(-0.9525)+(-0.90475)}{6} = -0.951625$（中點的斜率有更大的權值）。此時再退回原點（$x = 0$），以斜率 -0.951625 推出下一個點的位置（$x = 0.1$ 時 $y = 1 + (0.1 * -0.951625) = 0.9048375$）。

公式如下：

$$y_{i+1} = y_i + \frac{1}{6} * h * (k_1 + 2k_2 + 2k_3 + k_4)$$

其中，

$$k_1 = f(x_i, y_i)$$

$$k_2 = f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}h * k_1\right)$$

$$k_3 = f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}h * k_2\right)$$

$$k_4 = f(x_i + h, y_i + h * k_3)$$

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。

```
0.000
  RK1:   1.0000000000000000
  RK2:   1.0000000000000000
  RK4:   1.0000000000000000
e^(-x):  1.0000000000000000

0.050
  RK1:   0.9512056281970315
  RK2:   0.9512294324335736
  RK4:   0.9512294245007142
e^(-x):  0.9512294245007140

0.100
  RK1:   0.9047921471137096
  RK2:   0.9048374331278987
  RK4:   0.9048374180359603
e^(-x):  0.9048374180359595

0.150
  RK1:   0.8606433826830369
  RK2:   0.8607079979589024
  RK4:   0.8607079764250593
e^(-x):  0.8607079764250578

0.200
  RK1:   0.8186488294786360
  RK2:   0.8187307803894840
  RK4:   0.8187307530779840
e^(-x):  0.8187307530779818

0.250
  RK1:   0.7787033741169904
  RK2:   0.7788008155457855
  RK4:   0.7788007830714071
e^(-x):  0.7788007830714049

0.300
  RK1:   0.7407070321560997
  RK2:   0.7408182577504218
  RK4:   0.7408182206817205
e^(-x):  0.7408182206817179

0.350
  RK1:   0.7045646978320010
  RK2:   0.7046881308563623
  RK4:   0.7046880897187164
e^(-x):  0.7046880897187134
```

```
0.400
  RK1:   0.6701859060067403
  RK2:   0.6703200907571729
  RK4:   0.6703200460356426
e^(-x):  0.6703200460356393

0.450
  RK1:   0.6374846057319379
  RK2:   0.6376281994797670
  RK4:   0.6376281516217768
e^(-x):  0.6376281516217733

0.500
  RK1:   0.6063789448611849
  RK2:   0.6065307102947802
  RK4:   0.6065306597126368
e^(-x):  0.6065306597126334

0.550
  RK1:   0.5767910651721363
  RK2:   0.5769498633072362
  RK4:   0.5769498103804905
e^(-x):  0.5769498103804867

0.600
  RK1:   0.5486469074854966
  RK2:   0.5488116910163700
  RK4:   0.5488116360940306
e^(-x):  0.5488116360940264

0.650
  RK1:   0.5218760262931004
  RK2:   0.5220458333584115
  RK4:   0.5220457767610207
e^(-x):  0.5220457767610160

0.700
  RK1:   0.4964114134310991
  RK2:   0.4965853617698334
  RK4:   0.4965853037914141
e^(-x):  0.4965853037914095

0.750
  RK1:   0.4721893303569046
  RK2:   0.4723666118311392
  RK4:   0.4723665527410192
e^(-x):  0.4723665527410147
```

```
0.800
  RK1:  0.4491491486100750
  RK2:  0.4493290240727046
  RK4:  0.4493289641172258
e^(-x): 0.4493289641172216

0.850
  RK1:  0.4272331980578082
  RK2:  0.4274149925446102
  RK4:  0.4274149319487307
e^(-x): 0.4274149319487267

0.900
  RK1:  0.4063866225452041
  RK2:  0.4065697207718098
  RK4:  0.4065696597406032
e^(-x): 0.4065696597405991

0.950
  RK1:  0.3865572425889807
  RK2:  0.3867410847344451
  RK4:  0.3867410234545051
e^(-x): 0.3867410234545012

1.000
  RK1:  0.3676954247709637
  RK2:  0.3678795025306910
  RK4:  0.3678794411714463
e^(-x): 0.3678794411714423
```