数值计算实验报告

苏博南 202000460020 MAY. 2022, 山东大学, 青岛, 中国, https://www.subonan.com

1 第一章

1.1 实验 1

- **实验要求**：求方程 $x^2 + (\alpha + \beta)x + 10^9 = 0$ 的根，其中 $\alpha = -10^9, \beta = -1$。讨论如何设计计算格式才能有效地减少误差，提高计算精度。
- **实验过程**：观察到原方程为一元二次方程，故可以通过求根公式得到

$$x = \frac{-(\alpha + \beta) \pm \sqrt{(\alpha + \beta)^2 - 4 \times 10^9}}{2}$$

  然后将 $\alpha = -10^9, \beta = -1$ 代入即可。因为计算机浮点数存储遵循 IEEE754 标准，32 位机器的尾数域有 23 位已经足够精确。或可使用 4 位截断法也可以得到结果。
- **实验结果**：解得方程的根为 $x_1 = 10^9, x_2 = 1$。带入原方程发现误差为 0，因此解为精确解。

```
ghci> a=0-(10^9)
ghci> b=0-1
ghci> (-(a+b)+sqrt((a+b)^2-4*10^9)) / 2
1.0e9
ghci> (-(a+b)-sqrt((a+b)^2-4*10^9)) / 2
1.0
```

1.2 实验 2

- **实验要求**：以计算 x^{31} 为例，讨论如何设计计算格式才能减少计算次数。
- **算法**：可以采用快速幂算法。即需要计算 x^n 时，可以先计算 $y = x^{\frac{n}{2}}$。然后计算 $x = y^2$ 即可。故

$$x^{31} = x \cdot (x^{15})^2, \quad x^{15} = x \cdot (x^7)^2, \quad x^7 = x \cdot (x^3)^2, \quad x^3 = x \cdot x \cdot x$$

  总共需要 8 次乘法运算。一般看来，计算 x^n 时利用快速幂仅需要 $O(log_2(n))$ 次乘法运算。
- **程序代码**：

```haskell
-- Quick power method to compute n^m
-- E.g: ghci> quick_power 2 10
quick_power :: Integer -> Integer -> Integer
quick_power n m
    | m == 0 = 1
    | otherwise = if (rem m 2 == 1) then n * tmp * tmp
                        else tmp * tmp
        where tmp = quick_power n (div m 2)
```

- **算例及运行结果**：

```
ghci> quick_power 3 31
617673396283947
```

```
ghci> quick_power 2 31
2147483648
ghci> quick_power 1 31
1
```

- **结果分析**：结果和复杂度均正确。

2 第二章

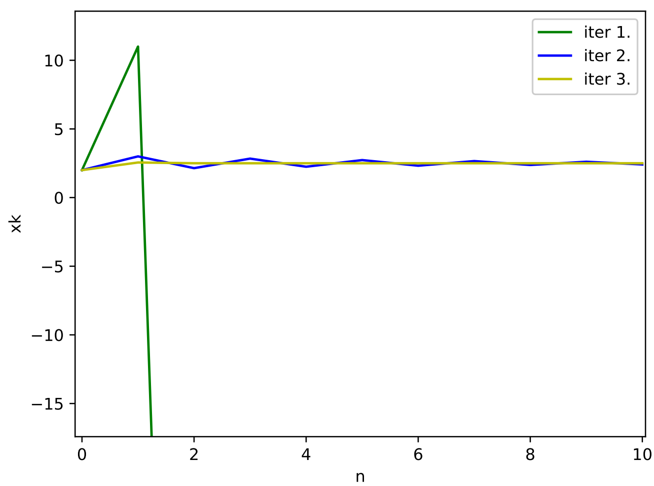
2.1 实验 1

- **实验要求**：求方程 $2x^2+x-15=0$ 的正根 ($x^*=2.5$) 近似值，分别利用如下三种格式编程计算
  1. $x_{k+1}=15-x_k^2$，取初始值 $x_0=2$。
  2. $x_{k+1}=\frac{15}{2x_k+1}$，取初始值 $x_0=2$。
  3. $x_{k+1}=x_k-\frac{2x_k^2+x_k-15}{4x_k+1}$，取初始值 $x_0=2$。
- **算法**：迭代算法，迭代方程在要求中已给出。
- **程序代码**：

```
iter :: Double->(Double->Double)->Integer->[Double]
iter x0 f n
    | n == 0 = [x0]
    | otherwise = tmp ++ [f (last tmp)]
    where
        tmp = iter x0 f (n - 1)
```

- **算例及运行结果**：

```
ghci> iter 2 (\x -> 15 - x^2) 5
[2.0,11.0,-106.0,-11221.0,-1.25910826e8,
-1.585353610400226e16]
ghci> iter 2 (\x -> 15 / (2 * x + 1)) 10
[2.0,3.0,2.142857142857143,2.8378378378378377,
2.2469635627530367,2.7302873986735445,2.3217748374586518,
2.6579016512723084,2.374994799783671,
2.6087003707152228,2.412583750641257]
ghci> iter 2 (\x -> x-(2*x*x + x - 15) / (4*x + 1)) 10
[2.0,2.5555555555555554,2.5005500550055006,
2.5000000550000006,2.5000000000000004,
2.5,2.5,2.5,2.5,2.5,2.5]
```



- **结果分析**：使用第一种迭代并无法计算出结果，且最后迭代将发散。使用第二种、第三种迭代方法都可以最终收敛至 x^* 附近。根据不动点迭代法 $g(x) = x - \phi(x) f(x)$，第二种迭代方法对应 $\phi(x) = \frac{1}{2x+1}$，此时 $g(x^*) \neq 0$，第三种迭代方法对应 $\phi(x) = f'(x)$，即为牛顿法。第二种、第三种方法分别以一阶 (线性)，二阶速率收敛，故第三种方法收敛速度更快。

2.2 实验 2

- **实验要求**：证明方程 $2 - 3 * x - sin(x) = 0$ 在 $(0, 1)$ 内有一个实根，使用二分法求误差不大于 0.0005 的根，及其需要的迭代次数。
- **算法代码**：

```
-- Binary Search
-- E.g: ghci> bisection (\x->(cos(x)-x)) 0 1
bisection :: (Double->Double)->Double->Double->Double
bisection f a b
    | (b - a > 0.0005) = if ((f mid) * (f a) < 0) then
      (bisection f a mid) else (bisection f mid b)
    | otherwise = a
    where mid = (a + b) / 2
```

- **算例及运行结果**：

```
ghci> bisection (\x->2-3*x-sin(x)) 0 1
0.5048828125
```

- **结果分析**：共进行了 11 次迭代运算，因为 $\frac{1}{2^{11}} = 0.000483 < 0.0005$. 得到了一个相对精确的结果，带入原方程发现结果接近成立。

2.3 实验 3

- **实验要求**：利用牛顿法求方程 $\frac{1}{2}+\frac{1}{4}x^2-xsinx-\frac{1}{2}cos2x=0$，分别取 $x_0=\frac{\pi}{2},5\pi,10\pi$，使精度不超过 10¯5。比较初值对实验结果的影响。
- **算法**：不断进行迭代 $x_{k+1}=x_k-\frac{f(x)}{f'(x)}$。
- **程序代码**：

```
-- Newton's Method
-- E.g: ghci> ntIter (\x->(cos(x)-x)) (\x->(-sin(x)-1))
          0.1 10
ntIter :: (Double->Double)->(Double->Double)->
          Double->Int->Double
ntIter f f' x n
    | n > 0 = ntIter f f' (x - (f x) / (f' x)) (n - 1)
    | n == 0 = x
```

- **算例及运行结果**：

```
ghci> f = \x -> 0.5+0.25*x*x-x*sin(x)-0.5*cos(2*x)
ghci> f' = \x -> 0.5*x-sin(x)-x*cos(x)+sin(2*x)
ghci> pi = 3.1415926
ghci> ntIter f f' (pi/2) 16
1.8954913430043157
ghci> ntIter f f' (5*pi) 20
1.895491635221433
ghci> ntIter f f' (10*pi) 100
11904.122472293395
```

- **结果分析**：当初值更靠近精确解时，达到同样的误差所需要的牛顿迭代次数更少。此外，当初始值选的偏离精确值太远时会导致迭代不收敛，即得不到解。

2.4 实验 4

- **实验要求**：已知 $f(x)=2x-e^x$ 在 $(0,1)$ 之间有一个实根，试分别用二分法，牛顿法，割线法，错位法，设计相应的计算格式，并编程求解。
- **算法代码**：

```
-- Secant Method
-- E.g: ghci> seIter (\x->(cos(x)-x)) 0.1 0 10
seIter :: (Double->Double)->Double->Double->Int->Double
seIter f p1 p0 n
    | delta < 1e-10 = p1
    | n > 0 = seIter f (p1 - (f p1) /
        (((f p1) - (f p0)) / (p1 - p0))) p1 (n - 1)
    | n == 0 = p1
    where delta = if (p1 > p0) then p1 - p0 else p0 - p1

-- False Position Method
-- E.g: ghci> falsePIter (\x->(cos(x)-x)) 0.1 0 10
```

```
falsePIter :: (Double->Double)->Double->Double->Int->
              Double
falsePIter f p1 p0 n
     | delta < 1e-10 = p1
     | n == 0 = p1
     | otherwise = falsePIter f p3 p2 (n - 1)
     where
         p2 = seIter f p1 p0 1
         a = p2 - (f p2) / (((f p2) - (f p1)) / (p2 - p1))
         b = p2 - (f p2) / (((f p2) - (f p0)) / (p2 - p0))
         p3 = if ((f p2) * (f p1) < 0) then a else b
         delta = if (p1 > p2) then p1 - p2 else p2 - p1
```

- **算例及运行结果**：

```
ghci> f = \x->5*x-exp(x)
ghci> bisection f 0 1
0.2591705322265625
ghci> ntIter f (\x->5-exp(x)) 0 20
0.2591711018190737
ghci> seIter f 0.1 0 20
0.25917110181907377
ghci> falsePIter f 0.1 0 20
0.2591711018190737
```

- **结果分析**：四种方法均在很少的计算次数内得到了很精确的解的近似值。

3 第三章

3.1 实验 1

- **实验要求**：以 $y = sin(x)$ 为例，在 $[0, \pi]$ 区间内生成 11 个、21 个数据点，设计算法或程序，用以下两个边界条件，分别计算其样条差值，并作图比较，分析其差异性。
  - 自然边界
  - 固支边界
  - 周期边界
  - 强制第一个子区间和第二个子区间样条多项式的三阶导数相同，倒数第二个子区间和最后一个子区间的样条多项式的三阶导数相同
- **算法代码**：

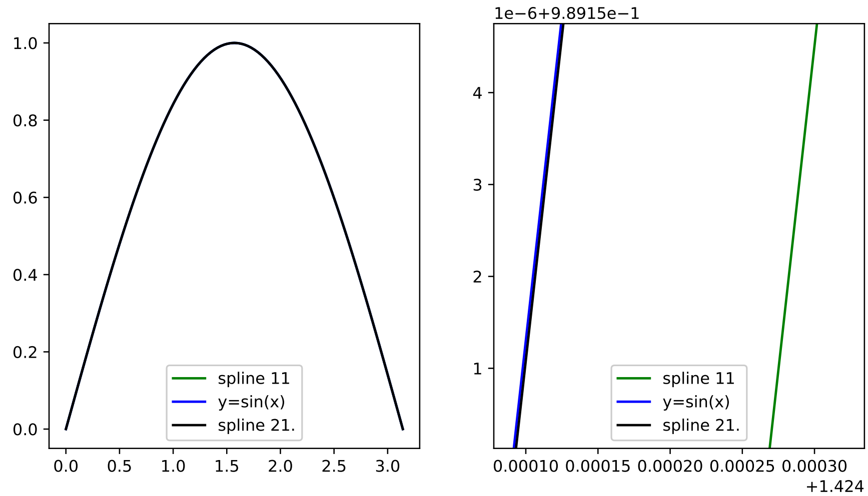
```
-- Natural spline interpolation
-- E.g: ghci> nsInter [1, 2, 3] [4, 5, 6] 2.5
generate_luz :: Int->[Double]->[Double]->[Double]->
                [[Double]]
generate_luz n xs h alpha
     | n == 0 = [[1], [0], [0]]
     | n == (length xs)-1 = [lst_l++[1], lst_u, lst_z
         ++[0]]
```

```
    | otherwise = [lst_l++[li], lst_u++[ui], lst_z++[zi]]
    where
        lst = generate_luz (n-1) xs h alpha
        lst_l = lst !! 0
        lst_u = lst !! 1
        lst_z = lst !! 2
        li = 2*((xs!!(n+1))-(xs!!(n-1)))-
              (h!!(n-1))*(lst_u!!(n-1))
        ui = (h!!n)/li
        zi = ((alpha!!n)-(h!!(n-1))*(lst_z!!(n-1)))/li

generate_abcd :: Int->[Double]->[Double]->[Double]->
                  [Double]->[[Double]]
generate_abcd n z u h ys
    | n == (length ys)-1 = [[ys!!n], [], [0], []]
    | otherwise = [[aj]++lst_a, [bj]++lst_b,
                   [cj]++lst_c, [dj]++lst_d]
    where
        lst = generate_abcd (n + 1) z u h ys
        lst_a = lst !! 0
        lst_b = lst !! 1
        lst_c = lst !! 2
        lst_d = lst !! 3
        cc = [0 | i <- [0..n]] ++ lst_c
        cj = (z !! n) - (u !! n) * (cc !! (n + 1))
        bj = ((ys !! (n + 1)) - (ys !! n)) / (h !! n) -
              (h !! n) * ((cc !! (n + 1)) + 2 * cj) / 3
        dj = ((cc !! (n + 1)) - cj) / (3 * (h !! n))
        aj = ys !! n

find_interval :: [Double]->Double->Int->Maybe Int
find_interval xs x n
    | n == (length xs) - 1 = Nothing
    | otherwise = if (x1 <= x && x <= x2) then (Just n)
                  else (find_interval xs x (n + 1))
    where
        x1 = xs !! n
        x2 = xs !! (n + 1)

elim :: Maybe Int -> Int
elim Nothing = 0
elim (Just a) = a

elim_d :: Maybe Double -> Double
elim_d Nothing = 0
elim_d (Just a) = a

nsInter :: [Double]->[Double]->(Double ->Maybe Double)
```

```haskell
nsInter xs ys x
    | i == Nothing = Nothing
    | otherwise = Just ((a !! (elim i)) +
                        (b!!(elim i)) * (x-(xs!!(elim i)))+
                        (c!!(elim i)) * (x-(xs!!(elim i)))^2+
                        (d!!(elim i)) * (x-(xs!!(elim i)))^3)
    where
        n = length xs - 1
        h = [(xs !! (i+1))-(xs !! i) | i <- [0..(n-1)]]
        alpha = [0] ++ [3/(h!!i)*((ys!!(i+1))-(ys!!i))-
                    3/(h!!(i-1))*((ys!!i)-(ys!!(i-1)))
                     | i <- [1..(n-1)]]
        tmp = generate_luz n xs h alpha
        l = tmp !! 0
        u = tmp !! 1
        z = tmp !! 2
        tmp_abcd = generate_abcd 0 z u h ys
        a = tmp_abcd !! 0
        b = tmp_abcd !! 1
        c = tmp_abcd !! 2
        d = tmp_abcd !! 3
        i = find_interval xs x 0
```

- **算例及运行结果**:

```haskell
ghci> xs = [pi / 10 * i | i <- [0..10]]
ghci> ys = map sin xs
ghci> f = nsInter xs ys
ghci> map elim_d (map f [1.1, 2.1, 3.1])
[0.891184200453932,0.8631914976064085,
4.157848292205905 4e-2]
```



- **结果分析**：会发现在自然边界条件下，当数据点为 11 个时，放大 1000 倍后样条函数就会和原函数有较大差别，然而当数据点为 21 个时，样条函数与原函数几乎重合。原因很很显然，当数据点更多，区间更小时，拟合的样条函数应该更为精确。

3.1.2 固支边界.

- **算法代码**：

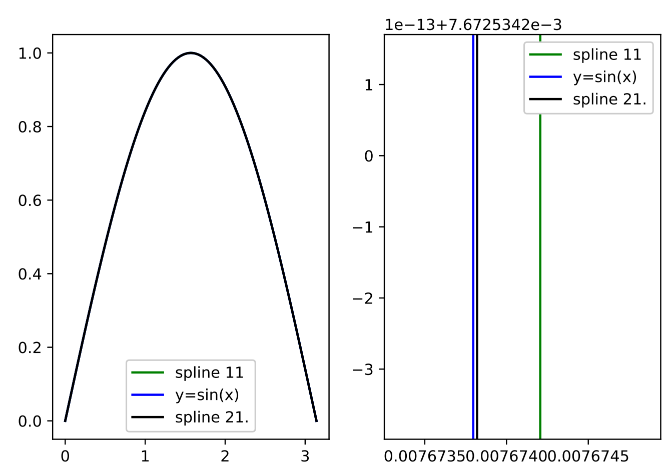
```haskell
-- Clamped spline interpolation
-- E.g: ghci> csInter [1, 2, 3] [4, 5, 6] 1 1 2.5
generate_luz' :: Int->[Double]-> [Double] ->
                 [Double] -> [[Double]]
generate_luz' i xs h alpha
    | i == 0 = [[2 * (h !! 0)], [0.5],
               [(alpha !! 0) / (2 * (h !! 0))]]
    | i == n = [lst_l ++ [(h !! (n - 1))*
                (2 - (lst_u !! (n - 1)))],
                lst_u,
               lst_z ++ [((alpha !! n) - (h !! (n - 1))*
                (lst_z !! (n - 1)))/
               ((h !! (n - 1))*
               (2 - (lst_u !! (n - 1))))]]
    | otherwise = [lst_l ++ [li], lst_u ++ [ui],
               lst_z ++ [zi]]
      where
        n = (length xs) - 1
        lst = generate_luz (n - 1) xs h alpha
        lst_l = lst !! 0
        lst_u = lst !! 1
        lst_z = lst !! 2
        li = 2 * ((xs !! (i + 1)) -
```

```haskell
                        (xs !! (i - 1))) -
                        (h !! (i - 1)) *
                        (lst_u !! (i - 1))
            ui = (h !! i) / li
            zi = ((alpha !! i) -
                        (h !! (i - 1)) *
                        (lst_z !! (i - 1))) / li

generate_abcd' :: Int -> [Double] -> [Double] ->
                  [Double] -> [Double] -> [[Double]]
generate_abcd' n z u h ys
    | n == (length ys) - 1 = [[ys !! n], [], [z !! n], []]
    | otherwise = [[aj] ++ lst_a, [bj] ++ lst_b,
                    [cj] ++ lst_c, [dj] ++ lst_d]
    where
        lst = generate_abcd (n + 1) z u h ys
        lst_a = lst !! 0
        lst_b = lst !! 1
        lst_c = lst !! 2
        lst_d = lst !! 3
        cc = [0 | i <- [0..n]] ++ lst_c
        cj = (z !! n) - (u !! n) * (cc !! (n + 1))
        bj = ((ys !! (n + 1)) - (ys !! n)) / (h !! n) -
                (h !! n) * ((cc !! (n + 1)) + 2 * cj) / 3
        dj = ((cc !! (n + 1)) - cj) / (3 * (h !! n))
        aj = ys !! n

csInter :: [Double]->[Double]->Double->
           Double->(Double->Maybe Double)
csInter xs ys fpo fpn x
    | i == Nothing = Nothing
    | otherwise = Just (a!!(elim i))+
                       (b!!(elim i))*(x-(xs!!(elim i)))
                       +
                       (c!!(elim i))*(x-(xs!!(elim i)))^2
                       +
                       (d!!(elim i))*(x-(xs!!(elim i)))^3
    where
        n = length xs - 1
        h = [(xs !! (i + 1)) - (xs !! i) | i<-[0..(n-1)]]
        alpha = [3*((ys!!1)-(ys!!0))/(h!!0)-3*fpo]++
                [3/(h!!i)*((ys!!(i + 1))-(ys!!i))-
                3/(h !! (i - 1))*((ys !! i)-(ys!!(i-1)))
                | i <- [1..(n-1)]] ++
                [3*fpn-3*((ys!!n)-(ys!!(n-1)))
```

```
                        / (h !! (n - 1))]
            tmp = generate_luz' n xs h alpha
            l = tmp !! 0
            u = tmp !! 1
            z = tmp !! 2
            tmp_abcd = generate_abcd' 0 z u h ys
            a = tmp_abcd !! 0
            b = tmp_abcd !! 1
            c = tmp_abcd !! 2
            d = tmp_abcd !! 3
            i = find_interval xs x 0
```

- **算例及运行结果**:

```
ghci> xs = [pi/10*i | i <- [0..10]]
ghci> ys = map sin xs
ghci> f = csInter xs ys 1 (-1)
ghci> map elim_d (map f [1.1, 2.1, 3.1])
[0.891184200453932,0.8631914976064085,
4.157848292205905 4e-2]
```



- **结果分析**: 在固支边界条件要求下，和自然边界其实得到了相似的结果。由于 $[0,\pi]$ 区间很小，取 11 个或 21 个数据点已经足够拟合曲线 $y = sin(x)$。但在放大后还有些许差别，显然取 21 个数据点的样条函数要更为优秀。此外，自然边界比固支边界拟合效果好。可以发现同样的差距自然边界要到 10^4 左右规模才能看出差别，但固定边界条件只需要到 7×10^-3。

3.1.3 周期边界.

- **算法**：首先我们有条件：

$$
\begin{aligned}
&S_j(x_j) = y_j, j = 0, 1, ..., n-1 \\
&S_j(x_{j+1}) = S_{j+1}(x_{j+1}), j = 0, 1, ..., n-2 \\
&S_j'(x_{j+1}) = S_{j+1}'(x_{j+1}), j = 0, 1, ..., n-2 \\
&S_j''(x_{j+1}) = S_{j+1}''(x_{j+1}), j = 0, 1, ..., n-2 \\
&S_0(x_0) = S_{n-1}(x_n) \\
&S_0'(x_0) = S_{n-1}'(x_n) \\
&S_0''(x_0) = S_{n-1}''(x_n)
\end{aligned}
\tag{1}
$$

可以发现一共有 $n + 3 * (n-1) + 3 = 4n$ 个条件。而我们待定的系数是 $a_j, b_j, c_j, d_j, j = 0, .., n-1$ 正好也有 $4n$ 个。我们可以设 $S_j(x) = a_j + b_j(x-x_j) + c_j(x-x_j)^2 + d_j(x-x_j)^3, j = 0, ..., n-1$，然后就是代入求解了。

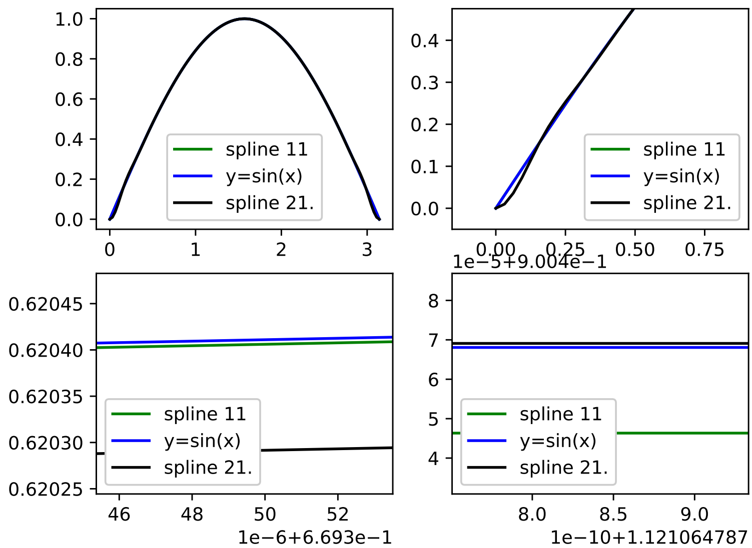
- **程序代码**：

```
psInter_generate_abcd :: [Double]->[Double]->[[Double]]
psInter_generate_abcd x y = solve_LES_GE eq b 0
    where
        n = (length x) - 1
        eq1 = [ [0 | i <- [0..(j*4-1)]] ++ [1] ++ [0 | i <- [(j*4+1)..(4*n-1)]]
            | j <- [0..(n-1)]]
        eq2 = [ [0 | i <- [0..(j*4-1)]] ++
                      [1, ((x !! (j + 1)) - (x !! j)), ((x !! (j + 1)) - (x !! j)) ^ 2, ((x !! (j + 1)) - (x !! j)) ^ 3, -1]
                      ++ [0 | i <- [(j*4+5)..(4*n-1)]]
            | j <- [0..(n-2)]]
        eq3 = [ [0 | i <- [0..(j*4-1)]] ++
                      [0, 1, 2 * ((x !! (j + 1)) - (x !! j)), 3 * ((x !! (j + 1)) - (x !! j)) ^ 2, 0, -1]
                      ++ [0 | i <- [(j*4+6)..(4*n-1)]]
            | j <- [0..(n-2)]]
        eq4 = [ [0 | i <- [0..(j*4-1)]] ++
                      [0, 0, 2, 6 * ((x !! (j + 1)) - (x !! j)), 0, 0, -2]
                         ++ [0 | i <- [(j*4+7)..(4*n-1)]]
            | j <- [0..(n-2)]]
        eq5 = [[0 | i <- [0..(4*(n-1)-1)]] ++
                  [1, ((x !! n) - (x !! (n - 1))), ((x !! n) - (x !! (n - 1))) ^ 2, ((x !! n) - (x !! (n - 1))) ^ 3]]
        eq6 = [[0,-1,0,0] ++ [0 | i <- [4..(4*(n-1)-1)]] ++
                  [0, 1, 2 * ((x !! n) - (x !! (n - 1))), 3 * ((x !! n) - (x !! (n - 1))) ^ 2]]
```

```
        eq7 = [[0,0,-2,0] ++ [0 | i <- [4..(4*(n-1)-1)]] ++
                [0, 0, 2, 6 * ((x !! n) - (x !! (n - 1)))]]
        eq = eq1 ++ eq2 ++ eq3 ++ eq4 ++ eq5 ++ eq6 ++ eq7
        b1 = [[(y !! j)] | j <- [0..(n-1)]]
        b2 = [[0] | j <- [0..(n-2)]]
        b3 = [[0] | j <- [0..(n-2)]]
        b4 = [[0] | j <- [0..(n-2)]]
        b5 = [[y !! 0]]
        b6 = [[0]]
        b7 = [[0]]
        b = b1 ++ b2 ++ b3 ++ b4 ++ b5 ++ b6 ++ b7

-- Periodic spline interpolation (ys[0] == ys[n] is necessary!)
-- E.g: ghci> psInter [1,2,3] [1,2,1] 2.5
psInter :: [Double]->[Double]->(Double->Maybe Double)
psInter xs ys x
    | i == Nothing = Nothing
    | otherwise = Just (aj + bj * (x - xj) + cj * (x - xj) ^ 2 + dj * (x - xj) ^ 3)
    where
        tmp = psInter_generate_abcd xs ys
        i = find_interval xs x 0
        j = elim i
        xj = xs !! j
        aj = tmp !! (4 * j) !! 0
        bj = tmp !! (4 * j + 1) !! 0
        cj = tmp !! (4 * j + 2) !! 0
        dj = tmp !! (4 * j + 3) !! 0
```

- **算例及运行结果**:

```
ghci> psInter [1,2,3] [1,2,1] 2.5
Just 1.5
ghci> xs = [pi/10*i | i <- [0..10]]
ghci> ys = map sin xs
ghci> f = psInter xs ys
```



- **结果分析**：可以看到样条函数也非常好地拟合了原函数。在边界处当取的数据点为 21 个时出现了明显的误差，但取的数据点只有 11 个时反而拟合的很好。在函数中部 21 个数据点表现更为出色。

3.1.4 强制边界.

- **算法**：首先我们有约束条件：

$$
\begin{aligned}
&S_j(x_j) = y_j, j = 0, 1, ..., n-1 \\
&S_j(x_{j+1}) = S_{j+1}(x_{j+1}), j = 0, 1, ..., n-2 \\
&S_j'(x_{j+1}) = S_{j+1}'(x_{j+1}), j = 0, 1, ..., n-2 \\
&S_j''(x_{j+1}) = S_{j+1}''(x_{j+1}), j = 0, 1, ..., n-2 \\
&y_n = S_{n-1}(x_n) \\
&S_0'''(x_1) = S_1'''(x_1) \\
&S_{n-2}'''(x_{n-1}) = S_{n-1}'''(x_{n-1})
\end{aligned}
\tag{2}
$$

我们设样条函数 $S_j(x) = a_j + b_j(x-x_j) + c_j(x-x_j)^2 + d_j(x-x_j)^3, j = 0, ..., n-1$ 仍然是 $4n$ 个待定系数，而约束条件也仍然是 $4n$ 个。因此和之前没什么区别直接列方程求解就好了。

- **程序代码**：

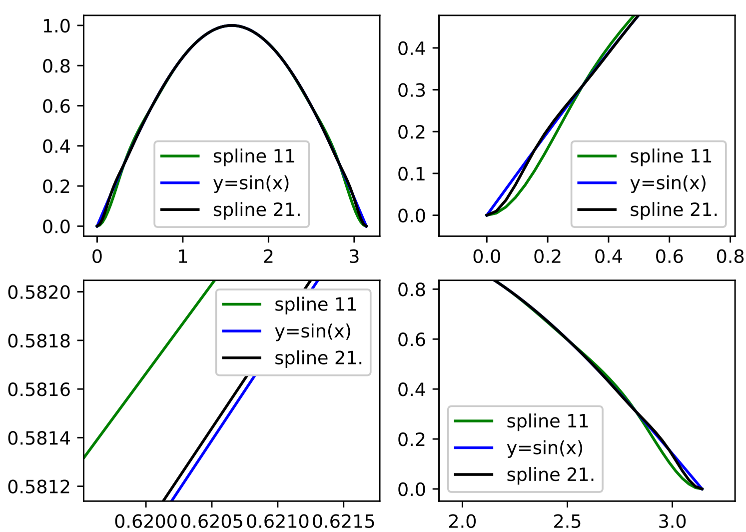
```
fsInter_generate_abcd :: [Double]->[Double]->[[Double]]
fsInter_generate_abcd x y = solve_LES_GE eq b 0
    where
        n = (length x) - 1
        eq1 = [ [0 | i <- [0..(j*4-1)]] ++ [1] ++ [0 | i <- [(j*4+1)..(4*n-1)]]
                | j <- [0..(n-1)]]
        eq2 = [ [0 | i <- [0..(j*4-1)]] ++
```

```
                        [1, ((x !! (j + 1)) - (x !! j)), ((x !! (j + 1)) - (x !! j)) ^ 2, ((x !! (j + 1)) - (x !! j)) ^ 3, -1]
                        ++ [0 | i <- [(j*4+5)..(4*n-1)]]
            | j <- [0..(n-2)]]
        eq3 = [ [0 | i <- [0..(j*4-1)]] ++
                        [0, 1, 2 * ((x !! (j + 1)) - (x !! j)), 3 * ((x !! (j + 1)) - (x !! j)) ^ 2, 0, -1]
                        ++ [0 | i <- [(j*4+6)..(4*n-1)]]
            | j <- [0..(n-2)]]
        eq4 = [ [0 | i <- [0..(j*4-1)]] ++
                        [0, 0, 2, 6 * ((x !! (j + 1)) - (x !! j)), 0, 0, -2]
                          ++ [0 | i <- [(j*4+7)..(4*n-1)]]
            | j <- [0..(n-2)]]
        eq5 = [[0 | i <- [0..(4*(n-1)-1)]] ++
                    [1, ((x !! n) - (x !! (n - 1))), ((x !! n) - (x !! (n - 1))) ^ 2, ((x !! n) - (x !! (n - 1))) ^ 3]]
        eq6 = [[0,0,0,1,0,0,0,-1] ++ [0 | i <- [8..(4*n-1)]]]
        eq7 = [[0 | i <- [8..(4*n-1)]] ++ [0,0,0,1,0,0,0,-1]]
        eq = eq1 ++ eq2 ++ eq3 ++ eq4 ++ eq5 ++ eq6 ++ eq7
        b1 = [[(y !! j)] | j <- [0..(n-1)]]
        b2 = [[0] | j <- [0..(n-2)]]
        b3 = [[0] | j <- [0..(n-2)]]
        b4 = [[0] | j <- [0..(n-2)]]
        b5 = [[y !! n]]
        b6 = [[0]]
        b7 = [[0]]
        b = b1 ++ b2 ++ b3 ++ b4 ++ b5 ++ b6 ++ b7

-- Forced spline interpolation
-- E.g: ghci> fsInter [1,2,3] [1,2,1] 2.5
fsInter :: [Double]->[Double]->(Double->Maybe Double)
fsInter xs ys x
    | i == Nothing = Nothing
    | otherwise = Just (aj + bj * (x - xj) + cj * (x - xj) ^ 2 + dj * (x - xj) ^ 3)
    where
        tmp = psInter_generate_abcd xs ys
        i = find_interval xs x 0
        j = elim i
        xj = xs !! j
        aj = tmp !! (4 * j) !! 0
```

```
            bj = tmp !! (4 * j + 1) !! 0
            cj = tmp !! (4 * j + 2) !! 0
            dj = tmp !! (4 * j + 3) !! 0
```

- **算例及运行结果**:

```
ghci> xs = [pi/10*i | i <- [0..10]]
ghci> ys = map sin xs
ghci> f = psInter xs ys
ghci> f 1.3
Just 0.9633978511257916
```



- **结果分析**: 同样两种取数据点的方式都很好地拟合了原函数。但由于奇怪的边界条件和取的数据点较少，在端点处还是出现了些许误差。也可以观察到取点数为 21 时要更贴近原函数。

4 第四章

4.1 实验 1-3

- **实验要求**: 自行编制复合梯形公式、Simpson 公式的计算程序。
- **算法**: 梯形公式为 $\int_a^b f(x)dx \approx \frac{b-a}{2}(f(a)+f(b))$，Simpson 公式为 $\int_a^b \approx \frac{b-a}{6}(f(a)+4f(\frac{a+b}{2})+f(b))$。复合情况则是分段应用梯形、Simpson 公式。
- **程序代码**:

```
-- Trapezoidal rule for numerical integration
-- E.g: ghci> trIntegrate (\x->x) 1 2
trIntegrate :: (Double->Double)->Double->Double->Double
trIntegrate f a b = (b - a) / 2 * ((f a) + (f b))
```

```

-- Simpson's rule for numerical integration
-- E.g: ghci> trIntegrate (\x->x) 1 2
simIntegrate :: (Double->Double)->Double->Double->Double
simIntegrate f a b = (b - a) / 6 * ((f a) +
                     4 * (f ((a + b) / 2)) + (f b))

-- Composite rule for numerical integration
-- E.g: ghci> csIntegrate (\x->x) simIntegrate
--                [i * 0.1 | i <- [0..10]]
-- E.g: ghci> csIntegrate (\x->x) trIntegrate
--                [i * 0.1 | i <- [0..10]]
csIntegrate :: (Double->Double)->((Double->Double)->
                Double->Double->Double)->[Double]->Double
csIntegrate f sim xs
    | (length xs) == 2 = ans
    | otherwise = ans + (csIntegrate f sim (tail xs))
    where
        x1 = head xs
        x2 = head (tail xs)
        ans = sim f x1 x2
```

- **算例及运行结果**:

```
ghci> csIntegrate (\x->1/sqrt(2*pi)*exp(-x*x/2))
    trIntegrate [i*0.01|i<-[0..100]]
0.3413427296391172
ghci> csIntegrate (\x->1/sqrt(2*pi)*exp(-x*x/2))
    simIntegrate [i*0.01|i<-[0..100]]
0.3413447460702231
```

- **结果分析**: 该积分结果应为 $\frac{1}{2}erf(\frac{1}{\sqrt{2}}) \approx 0.3413447460685434$ 这是个非基本函数，根本就没有精确解一说。然后发现 Simpson 公式求解出来更精确些。如果要计算精度为 10^{-4}，实验后我发现 Simpson 公式只需要取 $h = 0.5, n = 2$ 即可，而梯形公式需要 $h = 0.04, n = 25$。

```
ghci> csIntegrate (\x->1/sqrt(2*pi)*exp(-x*x/2))
    simIntegrate [i*0.5|i<-[0..2]]
0.34135548785664915
ghci> csIntegrate (\x->1/sqrt(2*pi)*exp(-x*x/2))
    trIntegrate [i*0.04|i<-[0..25]]
0.34131248158439637
```

4.2 实验 4

- **实验要求**：分别利用复合梯形，Simpson 公式计算定积分：

$$I(f) = \int_1^6 (2 + sin(2\sqrt{x}))dx$$

取 $h = 0.5, 0.25, 0.125$，列表给出两种格式的近似计算结果。
- **算法及代码**：见实验 1-3。
- **算例及运行结果**：

```
ghci> csIntegrate (\x->2+sin(2*sqrt(x))) trIntegrate
      [0.5*i|i<-[2..12]]
8.193854565172531
ghci> csIntegrate (\x->2+sin(2*sqrt(x))) trIntegrate
      [0.25*i|i<-[4..24]]
8.186049263770313
ghci> csIntegrate (\x->2+sin(2*sqrt(x))) trIntegrate
      [0.125*i|i<-[8..48]]
8.184120191790313
ghci> csIntegrate (\x->2+sin(2*sqrt(x))) simIntegrate
      [0.5*i|i<-[2..12]]
8.18344749663624
ghci> csIntegrate (\x->2+sin(2*sqrt(x))) simIntegrate
      [0.25*i|i<-[4..24]]
8.18347716779698
ghci> csIntegrate (\x->2+sin(2*sqrt(x))) simIntegrate
      [0.125*i|i<-[8..48]]
8.183479079161389
```

Table 1. 运行结果

$h =$	0.5	0.25	0.125
梯形公式	8.193854565172531	8.186049263770313	8.184120191790313
Simpson 公式	8.18344749663624	8.18347716779698	8.183479079161389

- **结果分析**：可以发现，在多数情况下，Simpson 公式会给出比梯形公式更精确的结果。

5 第五章

5.1 实验 1

- **实验要求**：求 $y' = 1 + y^2, y(0) = 0$ 的数值解，分别用欧拉显格式，梯形预估修正格式，4 阶龙格库塔格式，并与解析解比较这三种格式的收敛性。
- **算法及程序代码**：

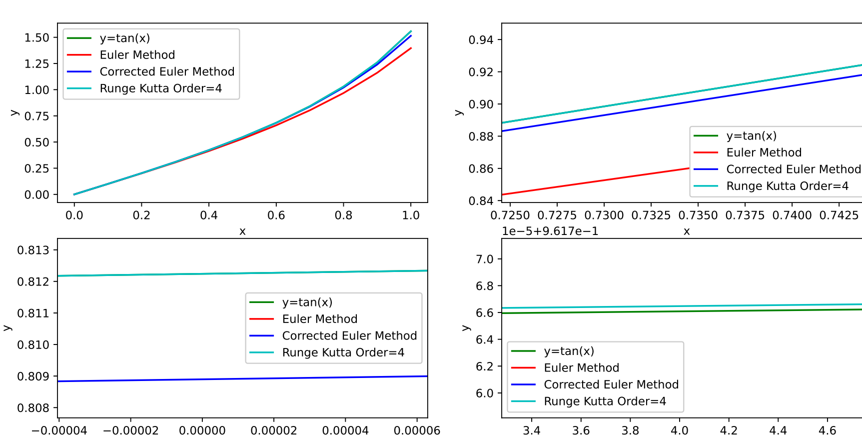
```
-- Euler's Method for ODE
```

```haskell
-- E.g: ghci> eulerMethod (\t y->y) 3 [0.01*i | i <- [0..3]]
-- ans: y=3*exp(t)
-- E.g: ghci> map (\x->3*exp(x)) [0.01*i | i <- [0..3]]
-- the result is close
eulerMethod :: (Double->Double->Double)->Double->[Double]->[Double]
eulerMethod f alpha xs
    | n == 1 = [alpha]
    | otherwise = pre ++ [ytj + h * (f tj ytj)]
    where
        n = length xs
        tj1 = xs !! (n - 1)
        tj = xs !! (n - 2)
        h = tj1 - tj
        ytj = last pre
        pre = eulerMethod f alpha [(xs !! i) | i <- [0..(n-2)]]

-- Corrected Euler's Method
-- E.g: ghci> ceulerMethod (\t y->y) 3 [0.01*i | i <- [0..3]]
-- the result of this method is more close to "map (\x->3*exp(x)) [0.01 * i | i <- [0..3]]" then Euler's Method
ceulerMethod :: (Double->Double->Double)->Double->[Double]->[Double]
ceulerMethod f alpha xs
  | n == 1 = [alpha]
  | otherwise = pre ++ [(h / 2) * ((f tj1 ytj1) + (f tj ytj)) + ytj]
  where
      n = length xs
      tj1 = xs !! (n - 1)
      tj = xs !! (n - 2)
      h = tj1 - tj
      ytj = last pre
      pre = ceulerMethod f alpha [(xs !! i) | i <- [0..(n-2)]]
      ytj1 = last pred
      pred = eulerMethod f alpha xs

-- Runge-Kutta Method of Order 4
-- E.g: ghci> rk4Method (\t y->y) 3 [0.01*i | i <- [0..3]]
-- the result of this method is more close to "map (\x->3*exp(x)) [0.01 * i | i <- [0..3]]" than other methods above
```

```haskell
rk4Method :: (Double->Double->Double)->Double->[Double]->[Double]
rk4Method f alpha xs
  | n == 1 = [alpha]
  | otherwise = pre ++ [ytj + (k1 + 2 * k2 + 2 * k3 + k4) / 6]
  where
    n = length xs
    tj1 = xs !! (n - 1)
    tj = xs !! (n - 2)
    h = tj1 - tj
    ytj = last pre
    pre = rk4Method f alpha [(xs !! i) | i <- [0..(n-2)]]
    k1 = h * (f tj ytj)
    k2 = h * (f (tj + h / 2) (ytj + k1 / 2))
    k3 = h * (f (tj + h / 2) (ytj + k2 / 2))
    k4 = h * (f tj1 (ytj + k3))
```

- **算例及运行结果**:

```haskell
ghci> eulerMethod (\t y->1+y^2) 0 [0.5*i | i <- [0..2]]
[0.0,0.5,1.125]
ghci> ceulerMethod (\t y->1+y^2) 0 [0.5*i | i <- [0..2]]
[0.0,0.5625,1.4580078125]
ghci> rk4Method (\t y->1+y^2) 0 [0.5*i | i <- [0..2]]
[0.0,0.5460530134538809,1.5546121041796463]
ghci> map tan [0.5*i | i <- [0..2]]
[0.0,0.5463024898437905,1.5574077246549023]
```



- **结果分析**：可以发现，修正梯形公式比显示欧拉格式更快速地收敛，而 Runge-Kutta 四阶方法则效果最好。只用了 10 个数据点就可以和解析解的误差只差了 10^4 左右。

5.2 实验 2

- **实验要求**：用 Runge-Kutta 4 阶方法求解描述振荡器的经典 van der Pol 微分方程：

$$\begin{cases} \frac{d^2y}{dt^2} - \mu(1-y^2)\frac{dy}{dt} + y = 0 \\ y(0) = 1, y'(0) = 0 \end{cases}$$

分别取 $\mu = 0.01, 0.1, 1$，作图比较计算结果。

- **算法**：先将原方程转化为：

$$\frac{d(y')}{dt} = \mu(1 - (\int_0^t y'(x)dx + 1)^2)y' - (\int_0^t y'(x)dx + 1)$$

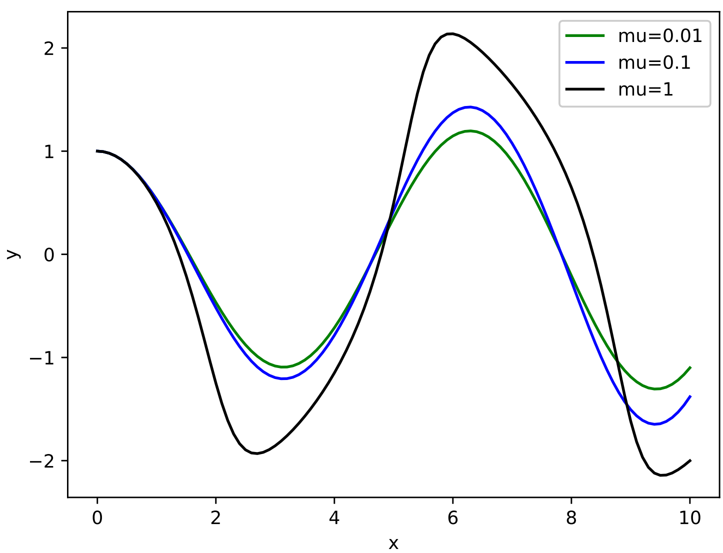
用 Runge-Kutta 4 阶方法求解出 $y'(t)$，然后可以继续求解 $y(t)$。

- **程序代码**：

```
elim_f :: (Double-> Maybe Double) -> (Double->Double)
elim_f f x = elim_d (f x)

integrate :: [Double]->[Double]->Double
integrate xs ys
     | length xs == 1 = 0
     | otherwise =  csIntegrate f simIntegrate xs
     where f = elim_f (nsInter xs ys)

rk4Method_vdP :: Double->[Double]->[Double]
rk4Method_vdP alpha xs
  | n == 1 = [alpha]
  | otherwise = pre ++ [ytj + (k1 + 2 * k2 + 2 * k3 + k4) / 6]
  where
      n = length xs
      tj1 = xs !! (n - 1)
      tj = xs !! (n - 2)
      h = tj1 - tj
      ytj = last pre
      pre = rk4Method_vdP alpha [(xs !! i) | i <- [0..(n-2)]]
      y = (integrate [(xs !! i) | i <- [0..(n-2)]] pre) + 1
      f = \t y' -> 1 * (1 - y ^ 2) * y' - y
      k1 = h * (f tj ytj)
      k2 = h * (f (tj + h / 2) (ytj + k1 / 2))
      k3 = h * (f (tj + h / 2) (ytj + k2 / 2))
      k4 = h * (f tj1 (ytj + k3))
```

- **算例及运行结果**：

```
ghci> xs = [0.1*i | i<- [0..100]]
ghci> ys = rk4Method_vdP 0 xs
ghci> y' = nsInter xs ys
ghci> y1 = elim_f y'
ghci> rk4Method (\t y-> (y1 t)) 1 [0.1*i| i <-[0..100]]
[1.0,0.9949986288868914,..., -2.000797986841788]
```



- **结果分析**：很显然，μ 越小时，函数震荡幅度越小。此外，由于我是在 $[0, 10]$ 这个区间上以间隔为 0.1 取了 100 个数据点，可能有点间隔太大，再加上 matplot 可能的圆滑描图操作看起来转折没有那么明显。但 $\mu = 1$ 情况还是能看出振荡器的函数图像的。

5.3 实验 3

- **实验要求**：试用 Adams Fourth-Order Predictor-Corrector 格式，求解如下常微分初值问题：

$$\begin{cases} \frac{dy}{dt} = \frac{t-y}{2} & 0 \le t \le 3 \\ y(0) = 1 \end{cases}$$

的数值解。分别取 $h = 1, 0.5, 0.25, 0.125$。

- **算法及程序代码**：

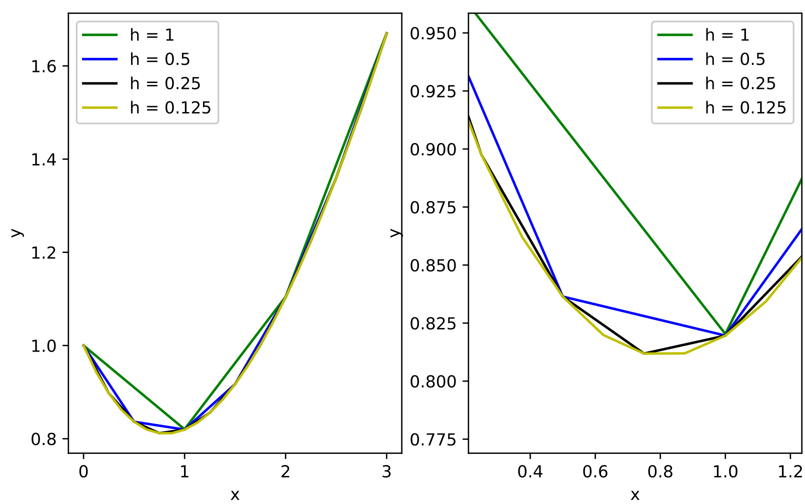
```
-- Adams Fourth-Order Predictor-Corrector
-- E.g: ghci> a4pcMethod (\t y->y) 3 [0.01*i | i <-
    [0..10]]
a4Method :: (Double->Double->Double)->Double->[Double]->[
    Double]
a4Method f alpha xs
```

```haskell
  | n == 4 = rk4Method f alpha xs
  | otherwise = pre ++ [ytj + (h / 24) * (55 * (f tj ytj) - 59 * (f tj1 ytj1) + 37 * (f tj2 ytj2) - 9 * (f tj3 ytj3))]
  where
      n = length xs
      pre = a4Method f alpha [(xs !! i) | i <- [0..(n-2)]]
      ntj = xs !! (n - 1)
      tj  = xs !! (n - 2)
      tj1 = xs !! (n - 3)
      tj2 = xs !! (n - 4)
      tj3 = xs !! (n - 5)
      h = ntj - tj
      ytj = pre !! (n - 2)
      ytj1 = pre !! (n - 3)
      ytj2 = pre !! (n - 4)
      ytj3 = pre !! (n - 5)


a4pcMethod :: (Double->Double->Double)->Double->[Double]->[Double]
a4pcMethod f alpha xs
  | n == 4 = rk4Method f alpha xs
  | otherwise = pre ++ [ytj + (h / 24) * (9 * (f ntj nytj) + 19 * (f tj ytj) - 5 * (f tj1 ytj1) + (f tj2 ytj2))]
  where
      n = length xs
      pred = a4Method f alpha xs
      pre = a4pcMethod f alpha [(xs !! i) | i <- [0..(n-2)]]
      ntj = xs !! (n - 1)
      tj  = xs !! (n - 2)
      tj1 = xs !! (n - 3)
      tj2 = xs !! (n - 4)
      h = ntj - tj
      nytj = pred !! (n - 1)
      ytj = pre !! (n - 2)
      ytj1 = pre !! (n - 3)
      ytj2 = pre !! (n - 4)
```

- **算例及运行结果**:

```
ghci> a4pcMethod (\t y->((t-y)/2)) 1 [i| i<- [0..3]]
[1.0,0.8203125,1.1045125325520833,1.6701859898037381]
ghci> a4pcMethod (\t y->((t-y)/2)) 1 [0.5*i| i<- [0..6]]
[1.0,0.83642578125,...,1.669132504283760 6]
```

```
ghci> a4pcMethod (\t y->((t-y)/2)) 1 [0.25*i| i<-
    [0..12]]
[1.0,0.897491455078125,...,1.6693638298881879]
ghci> a4pcMethod (\t y->((t-y)/2)) 1 [0.125*i| i<-
    [0..24]]
[1.0,0.9432392120361328,...,1.6693884660703506]
```



- **结果分析**：没什么好分析的，结果正确。

6 第六、七章

6.1 实验 1

- **实验要求**：使用 LU 分解、Jacobi、Gauss-Seidel 三种方法求解线性方程组：

$$\begin{cases} 4x - y + z = 7 \\ 4x - 8y + z = -21 \\ -2x + y + 5z = 15 \end{cases}$$

- **算法及程序代码**：

```
elim_LU :: Maybe ([[Double]], [[Double]]) -> ([[Double]],
    [[Double]])
elim_LU (Just (a, b)) = (a, b)
elim_LU Nothing = ([], [])

generate_ith_row_column :: [[Double]] -> Int -> Maybe ([[
    Double]], [[Double]])
generate_ith_row_column a i
    | i == 0 = if (a11 == 0) then Nothing else
```

```haskell
                        Just ([[a11] ++ [((a !! 0) !! j) | j <- [1..(n-1)]]], [[1] ++ [((a !! j) !! 0) / a11| j <- [1..(n-1)]]])
        | tmp == Nothing = Nothing
        | lu_ii == 0 = Nothing
        | otherwise = Just (u ++ [ui], l ++ [li])
        where
            n = length a
            a11 = ((a !! 0) !! 0)
            tmp = generate_ith_row_column a (i - 1)
            u = fst (elim_LU tmp)
            l = snd (elim_LU tmp)
            aii = ((a !! i) !! i)
            lu_ii = aii - (sum [((l !! k) !! i) * ((u !! k) !! i) | k <- [0..(i - 1)]])
            lii = 1
            uii = lu_ii
            ui = [0 | j <- [0..(i-1)]] ++ [uii] ++ [(((a !! i) !! j) - (sum [((u !! k) !! j) * ((l !! k) !! i) | k <- [0..(i-1)]])) / lii | j <- [(i + 1)..(n-1)]]
            li = [0 | j <- [0..(i-1)]] ++ [lii] ++ [(((a !! j) !! i) - (sum [((u !! k) !! i) * ((l !! k) !! j) | k <- [0..(i-1)]])) / uii | j <- [(i + 1)..(n-1)]]

-- LU Factorization
-- E.g: ghci> tmp = lu_fact [[4, -1, 1], [4, -8, 1], [-2, 1, 5]]
--      ghci> mult (fst tmp) (snd tmp)
lu_fact :: [[Double]]->([[Double]], [[Double]])
lu_fact a = (matrix_trans (snd tmp), fst tmp)
    where
        tmp = elim_LU (generate_ith_row_column a ((length a) - 1))

solve_l :: Int->[[Double]]->[[Double]]->[[Double]]
solve_l i l y
    | i == 0 = [[((y !! 0) !! 0) / ((l !! 0) !! 0)]]
    | otherwise = pre ++ [[(((y !! i) !! 0) - s) / ((l !! i) !! i)]]
    where
        pre = solve_l (i - 1) l y
        s = sum [((l !! i) !! j) * ((pre !! j) !! 0) | j <- [0..(i-1)]]

solve_u :: Int->[[Double]]->[[Double]]->[[Double]]
solve_u i u y
```

```haskell
    | i == n - 1 = [[((y !! i) !! 0) / ((u !! i) !! i)]]
    | otherwise = [[(((y !! i) !! 0) - s) / ((u !! i) !! i)]] ++ pre
    where
        n = length u
        pre = solve_u (i + 1) u y
        npre = [[0] | j <- [0..i]] ++ pre
        s = sum [((u !! i) !! j) * ((npre !! j) !! 0) | j <- [(i+1)..(n-1)]]

-- Solve linear equation systems by LU factorization
-- E.g: ghci> solve_LES_LU [[4,-1,1],[4,-8,1],[-2,1,5]] [[7],[-21],[15]]
solve_LES_LU :: [[Double]]->[[Double]]->[[Double]]
solve_LES_LU a y =
    solve_u 0 u x
    where
        n = length a
        lu = lu_fact a
        l = fst lu
        u = snd lu
        x = solve_l (n - 1) l y

-- Jacobi iteration to solve linear equation system
-- E.g: ghci> solve_LES_Jacobi [[4,-1,1],[4,-8,1],[-2,1,5]] [[7],[-21],[15]] [[0],[0], [0]] 40
solve_LES_Jacobi :: [[Double]]->[[Double]]->[[Double]]->Int->[[Double]]
solve_LES_Jacobi a y x nn
    | nn == 0 = x
    | otherwise = matrix_plus (matrix_mult t (solve_LES_Jacobi a y x (nn - 1))) b
    where
        n = length a
        dr = [ [0 | j <- [0..(i-1)]] ++ [1 / ((a !! i) !! i)] ++ [0 | j <- [(i+1)..(n-1)]]
                | i <- [0..(n-1)]]
        lu = [ [-((a !! i) !! j) | j <- [0..(i-1)]] ++ [0] ++ [-((a !! i) !! j) | j <- [(i+1)..(n-1)]]
                | i <- [0..(n-1)]]
        t = matrix_mult dr lu
        b = matrix_mult dr y

inverse_l :: [[Double]]->[[Double]]
inverse_l l =
```

```
    matrix_trans [  (matrix_trans (solve_l (n-1) l ([[0] | j <- [0..(i-1)]] ++ [[1]] ++ [[0] | j <- [(i+1)..(n-1)]]))) !! 0
        | i <- [0..(n-1)]
    ]
    where
        n = length l

-- Gauss-Seidel iteration to solve linear equation system
-- E.g: ghci> solve_LES_GS a y [[0],[0],[0]] 20
solve_LES_GS :: [[Double]]->[[Double]]->[[Double]]->Int->[[Double]]
solve_LES_GS a y x nn
    | nn == 0 = x
    | otherwise = matrix_plus (matrix_mult t (solve_LES_GS a y x (nn - 1))) b
    where
        n = length a
        dl = [  [(a !! i) !! j | j <- [0..i]]
                | i <- [0..(n-1)]]
        u = [   [0 | j <- [0..i]] ++ [-((a !! i) !! j) | j <- [(i+1)..(n-1)]]
                | i <- [0..(n-1)]]
        dlr = inverse_l dl
        t = matrix_mult dlr u
        b = matrix_mult dlr y
```

- **算例及运行结果**:

```
ghci> a = [[4,-1,1],[4,-8,1],[-2,1,5]]
ghci> b = [[7],[-21],[15]]
ghci> solve_LES_LU a b
[[2.0],[4.0],[3.0]]
ghci> solve_LES_Jacobi a b [[0],[0],[0]] 40
[[2.0],[4.0],[3.0]]
ghci> solve_LES_GS a b [[0],[0],[0]] 20
[[2.0],[4.0],[3.0]]
```

- **结果分析**: Gauss-Seidel 方法只需要 20 次迭代就可以收敛至解，而 Jacobi 方法要更慢一点。反代回方程发现解正确。考虑一下两种方法收敛性的证明:

Proof. 记 $\mathbf{x}^{(i)}$ 为迭代 $\mathbf{x}^{(k+1)} = \mathbf{T}\mathbf{x}^{(k)} + \mathbf{y}$ 了 i 次后得到的近似解，记 $\varepsilon^{(i)} = \mathbf{x}^{(i)} - \mathbf{x}$，其中 $\mathbf{x}$ 是 $\mathbf{Ax} = \mathbf{b}$ 的精确解，显然有 $\varepsilon^{(i+1)} = \mathbf{T}\varepsilon^{(i)}$。
下面我们可以证明，如果 $\mathbf{T}$ 的所有特征值 λ_i 都满足 $|\lambda_i| < 1$，而且它的特征向量张成了一个 n 维空间，则原迭代收敛。

记 $\mathbf{T}\mathbf{v}_i = \lambda_i \mathbf{v}_i$，$\varepsilon^{(0)} = \sum_{i=1}^{n} a_i \mathbf{v}_i$。则有：

$$\varepsilon^{(t)} = \sum_{i=1}^{n} a_i \lambda_i^t \mathbf{v}_i \Rightarrow \lim_{t\to\infty} \varepsilon^{(t)} = \lim_{t\to\infty} \sum_{i=1}^{n} a_i \lambda_i^t \mathbf{v}_i$$

故当 $|\lambda_i| < 1$ 时，有 $\lim_{t\to\infty} \varepsilon^{(t)} = 0$，即原迭代收敛。而且从这里可以看出收敛的速度也取决于 $\mathbf{T}$ 的特征值的绝对值大小。

□

7 第八章

7.1 实验 1

- **实验要求**：已知下表观测数据，求一个二次多项式拟合这组数据，试写出其最小二乘模型，并给出其正则方程组和解。

Table 2. **观测数据**

x	-2	-1	0	1	2
$f(x)$	0	1	2	1	0

- **算法**：在这个问题中，有 $m = 5, n = 2$。可以列出正则方程组 $\mathbf{R}^T\mathbf{R}\mathbf{a} = \mathbf{R}^T\mathbf{y}$，其中：

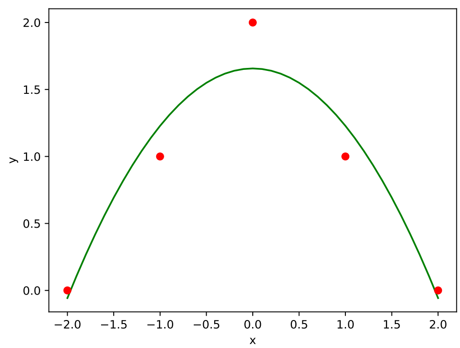
$$\mathbf{R} = \begin{pmatrix} x_1^0 & x_1^1 & \dots & x_1^{n-1} & x_2^n \\ x_2^0 & x_2^1 & \dots & x_2^{n-1} & x_2^n \\ \dots & \dots & \dots & \dots & \dots \\ x_m^0 & x_m^1 & \dots & x_m^{n-1} & x_m^n \end{pmatrix}, \mathbf{a} = \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_n \end{pmatrix}, \mathbf{y} = \begin{pmatrix} y_0 \\ y_1 \\ \dots \\ y_m \end{pmatrix} \tag{3}$$

- **程序代码**：

```haskell
-- Least square normal equations
-- E.g: ghci> equation_LSE [-2,-1,0,1,2] [0,1,2,1,0] 2
equation_LSE :: [Double]->[Double]->Int->[Double]
equation_LSE xs ys n = [((x !! i) !! 0) | i <- [0..n]]
    where
        m = length xs
        r = [   [(xs !! i) ^ j | j <- [0..n]]
              | i <- [0..(m-1)]]
        y = [[ys !! i] | i <- [0..(m-1)]]
        a = matrix_mult (matrix_trans r) r
        b = matrix_mult (matrix_trans r) y
        x = solve_LES_LU a b
```

- **算例及运行结果**：

```
ghci> equation_LSE [-2,-1,0,1,2] [0,1,2,1,0] 2
[1.657142857142857,0.0,-0.42857142857142855]
```



- **结果分析**：拟合的二次函数挺像的。

7.2 实验 2

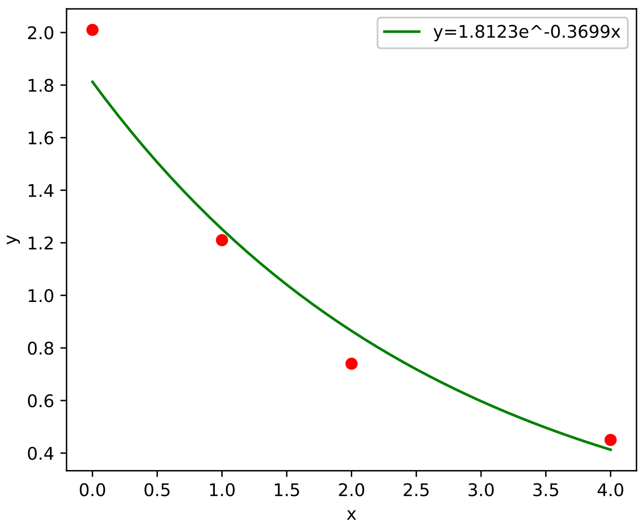
- **实验要求**：研究发现单原子波函数的基本形式为 $y = ae^{-bx}$，试根据实验室测试数据 (下表) 确定参数 a, b。

Table 3. **观测数据**

x	0	1	2	4
y	2.010	1.210	0.740	0.450

- **算法**：可以将 $y = ae^{-bx}$ 转化为 $lny = lna - bx$，然后用线性方程拟合的办法去做。
- **算例及运行结果**：

```
ghci> x = [0,1,2,4]
ghci> y = [2.010,1.210,0.740,0.450]
ghci> y1 = map log y
ghci> tmp = equation_LSE x y1 1
ghci> a = exp(tmp !! 0)
ghci> b = -(tmp !! 1)
ghci> a
1.8123230905565 6
ghci> b
0.3698993854876202
```



- **结果分析**：拟合的指数函数挺像的。

8 第九章

8.1 实验 1

- **实验要求**：已知矩阵

$$\mathbf{A}=\begin{pmatrix}4&-1&1\\-1&3&-2\\1&-2&3\end{pmatrix}$$

是一个对称矩阵，且其特征值为 $\lambda_1=6,\lambda_2=3,\lambda_3=1$。分别用幂法、对称幂法、反幂法求其最大特征值和特征向量。

- **算法代码**：

```
norm_inf :: [[Double]]->Int->Double
norm_inf v i
  | i == n = 0
  | otherwise = if ((abs ((v !! i) !! 0)) > nxt) then (
      abs ((v !! i) !! 0)) else nxt
  where
      n = length v
      nxt = norm_inf v (i + 1)

norm_2 :: [[Double]]->Double
norm_2 v = sqrt(((matrix_mult (matrix_trans v) v) !! 0)
    !! 0)

find_least_elem :: [[Double]]->Double->Int->Int
find_least_elem v d i
  | i == n = -1
```

```haskell
  | (abs ((v !! i) !! 0)) == d = i
  | otherwise = find_least_elem v d (i + 1)
  where
      n = length v

-- Power method to find the biggest eigenvalue of matrix A
-- E.g: ghci> ghci> power_method [[4,-1,1],[-1,3,-2],[1,-2,3]] [[1],[1],[1]] 40
power_method :: [[Double]]->[[Double]]->Int->(Double, [[Double]])
power_method a x n
  | n == 0 = (mu, x)
  | mu == 0 = (0, x)
  | otherwise = power_method a (matrix_dot y (1 / mu)) (n - 1)
  where
      norm = norm_inf x 0
      p = find_least_elem x norm 0
      y = matrix_mult a x
      mu = (y !! p) !! 0

-- Symmetric power method to find the biggest eigenvalue of A
-- E.g: ghci> symmetric_power_method [[4,-1,1],[-1,3,-2],[1,-2,3]] [[1],[0],[0]] 10
symmetric_power_method :: [[Double]]->[[Double]]->Int->(Double, [[Double]])
symmetric_power_method a x n
  | n == 0 = (mu, x)
  | norm == 0 = (0, x)
  | otherwise = symmetric_power_method a (matrix_dot y (1 / norm)) (n - 1)
  where
      y = matrix_mult a x
      mu = ((matrix_mult (matrix_trans x) y) !! 0) !! 0
      norm = norm_2 y

inverse_power_method_iter :: [[Double]]->[[Double]]->Double->Int->(Double, [[Double]])
inverse_power_method_iter a x q m
  | m == 0 = (mu, x)
  | otherwise  = inverse_power_method_iter a (matrix_dot y (1/(y !! p !! 0))) q (m - 1)
  where
      n = length x
      idn = [ [0 | j <- [0..(i-1)]] ++ [1] ++ [0 | j <- [(i+1)..(n-1)]]
```

```
                | i <- [0..(n-1)]]
        t = matrix_minus a (matrix_dot idn q)
        y = solve_LES_LU t x
        norm = norm_inf y 0
        p = find_least_elem y norm 0
        mu = y !! p !! 0

-- Inverse power method to find the eigenvalue of matrix a
-- E.g: ghci> inverse_power_method [[4,-1,1],[-1,3,-2],[1,-2,3]] [[1],[1],[1]] 10
inverse_power_method :: [[Double]]->[[Double]]->Int->(Double, [[Double]])
inverse_power_method a x n
  = (1 / (fst tmp) + q, snd tmp)
  where
        q = (matrix_dot (matrix_mult (matrix_mult (matrix_trans x) a) x) (1 / ((matrix_mult (matrix_trans x) x) !! 0 !! 0))) !! 0 !! 0
        norm = norm_inf x 0
        tmp = inverse_power_method_iter a (matrix_dot x (1 / norm)) q n
```

- **算例及运行结果**:

```
ghci> a = [[4,-1,1],[-1,3,-2],[1,-2,3]]
ghci> x = [[1],[1],[1]]
ghci> power_method a x 20
(5.999942779650155,[[0.9999999999999999],
[-0.999997138982507],[0.9999971389825084]])
ghci> symmetric_power_method a x 20
(5.999999999994542,[[0.5773513703973472],
[-0.5773497185849769],[0.5773497185849779]])
ghci> inverse_power_method a x 20
(1.0000000000001705,[[0.5000000000003411],
[1.0],[0.5000000000003411]])
```

- **结果分析**: 可以发现，幂法和对称幂法都找到了最大的特征值 $\lambda = 6$，然后对应的特征向量都形如 $\alpha(1,-1,1)^T$。在很少的迭代次数就得到了相对精确的解。然后反幂法本来就不是求解最大特征值的，他通过转化求 $\mathbf{A}$ 的特征值为求 $(\mathbf{A}-q\mathbf{I})^{-1}$ 的特征值，因此求出来的特征值应该为离 q 最近的特征值。($\frac{1}{\lambda-q}$ 最大的 λ)。而书上选取的 q 取决于输入向量，在输入向量为 $(1,1,1)^T$ 时，迭代结果为最小的特征值，即 $\lambda = 1$。