实验四报告

数据结构

- 用顺序表来完成任意同维度向量的计算
- 使用顺序表和链表来完成任意一元多项式的计算
- 分别使用顺序表和链表完成表达式的存储
- 使用栈完成表达式运算
- 使用数组完成矩阵的运算

程序模块

使用简单的交互界面判断用户需要进行的操作.

- 当选定操作之后,将调用不同函数进行数据的输入、处理以及输出返回处理好的表达式或者处理好的矩阵。
- 向量调用函数(`vectorCalculation`)、一元多项式调用函数(`polyCalculation`)、表达式调用函数(`Expr`)以及矩阵调用函数(`mat`).
- 将处理好的数据以及选择的操作,传入总的计算函数(`calculator`)或者储存函数(`definition`)(比如 `DEF` 表达式),根据操作分发给不同的计算函数,并选择输出结果或者迭代进入下一轮(比如 `DEF` 的情况).

另外,为方便操作,在每个含参表达式计算过程中可以当场更换变量并计算结果.

下面我将介绍一下程序各个功能的实现

- 向量的计算
  1. 先创建顺序表用来存放向量.
  2. 通过 `initList` 将以字符串形式输入的向量转化成顺序表中的储存形式.
  3. 之后调用 `cos` 函数计算两个向量的余弦值并输出，或调用 `valCal` 函数计算两个向量的和或差,并调用 `vecDisplay` 输出结果.
- 任意一元多项式的计算
  1. 根据用户的不同要求，选择使用顺序表或链表来解决一元多项式计算问题.
  2. 如果选择顺序表，则先通过重载函数 `initList` 将以字符串形式输入的一元多项式转化成顺序表中的储存形式；如果选择链表，同样调用重载函数 `initList` 将以字符串形式输入的一元多项式转化成链表中的储存形式.

3. 之后调用重载函数 `polyCal` 进行两个一元多项式之间的计算，调用重载函数 `grad` 进行一元多项式的求导运算。在编写程序时，一元多项式的加法和减法都在函数 `polyCal` 中完成，而将一元多项式乘法写成单独的一个函数 `polyMulti` 并使用优先队列完成计算，最后调用 `polyDisplay` 输出结果.

- 含参表达式求值
  1. 通过 `findLocalVar` 函数搜索得到函数中的参数,并将其名称以及位置依次存储在一个 `vector<pair<> >` 当中.
  2. 然后进入循环,要求用户输入每个参数的值,并对表达式中的每个参数用数值进行替换.
  3. 之后调用 `calExpr` 函数计算处理之后的表达式,输出结果.
- 不含参表达式求值
  1. 此时 `findLocalVar` 函数返回 `false` ,因此直接调用 `calExpr` ,输出结果.
- 函数相关的表达式求值
  1. 定义一个全局变量 `func` ,类型为 `pair<>` 存储所有DEF出来的函数,其中 `first` 存函数名, `second` 存函数体.
  2. 若传入字符串的开头识别为 `DEF`
     1. 对 `expr` 第四位之后的子串进行分割为函数名和函数体,按照1格式先存入函数名,然后让函数体进入函数 `findFunc` .
     2. 在这个函数当中,将搜索函数体中是否含有已存储的函数,并将函数名替换为带括号的函数体.
     3. 将处理后的函数体存入 `func` ,流程结束.
  3. 若传入字符串开头识别为 `RUN`
     1. 将四位之后的子串分割为函数名 `function` 和传入参数的值 `param` (比如f(5)中,5为传入参数的值).
     2. 在 `func` 中进行遍历,找到函数名 `function` ,并取出函数体,和 `param` 一起传入含参表达式求值的函数,即使用功能1.
     3. 得到结果,输出结果.
- 矩阵计算

  使用一个矩阵类 `Matrix` 完成所有操作.
  1. `matrix` 函数中会得到按照要求输入的矩阵,形式为一个字符串,使用 `processInput` 函数将之传入一个 `Matrix` 对象中
  2. 根据要求操作不同,将会需要用户输入一个矩阵 `mat` ,或是两个矩阵 `mat1,mat2` .全部存储完成后将会进行各自的操作
  3. 下面介绍一下矩阵求逆的过程
     1. 将 A 矩阵分解为 L 下三角矩阵和 U 上三角矩阵。
        1. L对角线填充为1
        2. $U_{0,j} = A_{0,j}(j = 0, ..., m - 1)$
        3. $U_{0,j} = A_{0,j}(j = 0, ..., m - 1)$
        4. U是按行迭代计算，L是按列迭代计算，UL交错计算，且U先L一步

           for k = 1 to m-1:

{

$$U_{k,j} = \frac{A_{k,j} - \sum_{t=0}^{k-1}(L_{i,t}U_{t,j})}{L_{k,k}} = A_{k,j} - \sum_{t=0}^{k-1}(L_{k,t}U_{t,j}), (j = k, ..., m-1)$$

$$L_{i,k} = \frac{A_{i,k} - \sum_{t=0}^{k-1}(L_{i,t}U_{t,k})}{U_{k,k}}, (i = k, ..., m-1)$$

}

2. 分别对 `L` 和 `U` 求逆，得到 `Linv` 和 `Uinv` .
   1. 列顺序行顺序
      for j = 0 to m-1, i = j to m-1
      {

$$L_{inv(i,j)} = \begin{cases} L_{i,j}^{-1} & ,i = j \\ 0 & ,i < j \\ -L_{inv(j,j)} \sum_{k=j}^{i-1}(L_{i,k}L_{inv(k,j)}) & ,i > j \end{cases}$$

      }
   2. 列顺序行倒序
      for j = 0 to m-1, i = j to 0
      {

$$U_{inv(i,j)} = \begin{cases} U_{i,j}^{-1} & ,i = j \\ 0 & ,i > j \\ -\frac{1}{U_{i,i}} \sum_{k=i+1}^{j}(U_{i,k}U_{inv(k,j)}) & ,i < j \end{cases}$$

      }
3. $Ainv = Linv * Uinv$

复杂度分析

核心的计算函数包括:

1. 向量计算(`vecCal`)
   > 时间复杂度为$O(N)$ 空间复杂度为$O(N)$

2. 一元多项式加减法(`polyCal`)
    1. 顺序表储存
    > 时间复杂度为$O(M+N)$ 空间复杂度为$O(M+N)$

    2. 链表储存
    > 时间复杂度为$O(M+N)$ 空间复杂度为$O(M+N)$

3. 一元多项式乘法(`polyMulti`)
    1. 顺序表储存
    > 时间复杂度为$O(MN)$ 空间复杂度为$O(M+N)$

    2. 链表储存
    > 时间复杂度为$O(MN)$ 空间复杂度为$O(M+N)$

4. 表达式求值(`calExpr`)
    > 时间复杂度为$O(N)$ 空间复杂度为$O(N)$

5. 矩阵求行列式(`getDet`)
    > 对于任意矩阵，交换其任意两行，其行列式变为相反数；对于任意矩阵，其中任意一行加上任意另一行的倍数，其行列式不变——对矩阵变换，并记录变换对行列式产生的影响，就可以得到一个上三角矩阵，并 $O(N)$ 地得知其行列式，从而推算出原矩阵的行列式。
    > 时间复杂度为$T(N)=O(N^3+N)=O(N^3)$ 空间复杂度为$O(1)$

6. 矩阵求逆(`inv_LU`)
    > 时间复杂度为$O(\frac{N^3}{3})$ 空间复杂度为$O(N^2)$

思考题

1. 优点：交互简单,功能强大,支持向量计算、一元多项式计算、表达式求值以及矩阵计算；可以调用定义过的函数；代码模块比较清楚,便于添加功能.缺点：输入自由度低，需要根据要求进行输入；算法时间复杂度和空间复杂度较高.
2. 表达式计算库
    1. `Blitz++`
       参考网站：Blitz++
       Blitz++ 是一个高效率的数值计算函数库，它的设计目的是希望建立一套既具像C++ 一样方便，同时又比Fortran速度更快的数值计算环境。通常，用C++所写出的数值程序， 比 Fortran慢20%左右，因此Blitz++正是要改掉这个缺点。方法是利用C++的template 技术，程序执行甚至可以比Fortran更快。
    2. `Eigen`
       参考网站:Eigen

Eigen是一个高层次的C++库,有效支持 得到的线性代数，矩阵和矢量运算，数值分析及其相关的算法。

比方说计算矩阵的逆

```
// 求特征值及特征向量
// 实对称矩阵可以保证对角化成功
Eigen::Matrix3d matrix_33 = Eigen::Matrix3d::Random();
Eigen::SelfAdjointEigenSolver<Eigen::Matrix3d> eigen_solver(matrix_33.transpose()*matrix_33)
cout << "特征值eigenvalues = \n" << eigen_solver.eigenvalues() << endl;
cout << "特征向量eigenvectors = \n" << eigen_solver.eigenvectors() << endl;
//求矩阵的逆,采用QR分解的方法较快
Eigen::Matrix< double, 50, 50 > matrix_NN = Eigen::MatrixXd::Random(50, 50);   //初始化为随机
Eigen::Matrix< double, 50, 1> v_Nd = Eigen::MatrixXd::Random(50, 1);
x = matrix_NN.colPivHouseholderQr().solve(v_Nd);
cout << "矩阵的逆inverse=" << x;
```

3. 加速矩阵计算
    - 可以使用 `Strassen` 算法加速矩阵乘法($N > 300$有明显效果)
    - 使用GPU可以做到高效并行计算矩阵.

      CPU是专为顺序串行处理而优化的几个核心组成。而GPU则由数以千计的更小、更高效的核心组成，这些核心专门为同时处理多任务而设计，可高效地处理并行任务。也就是，CPU虽然每个核心自身能力极强，处理任务上非常强悍，无奈他核心少，在并行计算上表现不佳；反观GPU，虽然他的每个核心的计算能力不算强，但他胜在核心非常多，可以同时处理多个计算任务，在并行计算的支持上做得很好。
    - CUDA就是N卡的运算平台,我们可以编写代码在CUDA上进行运算