Наброски по 4 лабе по ЧМ

Кейс, когда $m \leq n$.

В таком случае исключаем из подсчёта те компоненты вектора x, для которых минимальны следующие значения:

$$\max_i \left(abs \left(\frac{\partial(F_i(x^k))}{\partial x_j} \right) \right)$$

Получается, для нахождения того, какие компоненты необходимо исключить, нужно посчитать для каждой переменной все производные функций, найти максимальное значение этих производных, отсортировать по возрастанию и убрать $(n - m)$ компонент.

Возможный вариант:

1. Создать структуру типа `std::vector<std::pair<int, double>>` размера n, где `key == int` - номер переменной x_j, а `value == double` - значение для суммы по модулю значений производных функций F_i в точке x_j;
2. Для каждой переменной x_j посчитать значения производных всех функций F_i, записать максимальное по модулю значение в отдельную переменную: double $a_j = \max\limits_{i=1,m} abs \left(\frac{\partial(F_i(x^k))}{\partial x_j} \right)$;
3. Добавить в вектор пары pair(j, a_j);
4. После добавления всех n пар чисел, отсортировать вектор при помощи функции `std::sort` по второму параметру лежащих внутри элементов;
5. Посмотреть индексы первых $(n - m)$ элементов, исключить их из расчётов вектора Δx (вместо недостающих элементов записать нули). Вариант исключения: создать `std::vector<bool>` в качестве маски элементов.

Кейс, когда $m \geq n$.

В таком случае исключаем из подсчёта те функции F_i, для которых минимальны следующие значения:

$$abs \left(F_i(x^k) \right)$$

Возможный вариант:

1. Создать `std::vector<std::pair<int, double>>` размера m, где `key == int` - номер функции F_i, а `value == double` - значение по модулю этой функции в точке x^k;
2. Отсортировать вектор при помощи функции `std::sort` по второму параметру лежащих внутри элементов;
3. Посмотреть индексы первых $(m - n)$ элементов, исключить их из расчётов вектора Δx. Вариант исключения: создать `std::vector<bool>` в качестве маски элементов.

Зависимости функций

Таким образом, все функции, занимающиеся вычислениями, должны будут принимать массив с маской элементов, и соответственно учитывать эту маску во время вычислений. Поскольку не может быть ситуаций, когда выполняются одновременно оба кейса (кроме ситуации $n = m$), то передавать необходимо лишь одну маску. Кейс $n = m$ должен игнорировать переданную в функцию маску, либо и вовсе её не получать (в таком случае лучше передавать вектор как указатель).

Относительно кейса $m \geq n$, для вычислений на каждой итерации необходимо найти вектор из значений $F_i(x^k)$, поэтому вполне себе возможно без дополнительных затрат его переиспользовать.

Относительно кейса $m \leq n$, в целом тоже можно переиспользовать вычисленную матрицу как матрицу Якоби, но здесь уже вопрос памяти (хранить лишнюю матрицу размера вплоть до 10x10, хотя в целом не так уж и много)

Хранение данных

Хранение матрицы Якоби

Матрицу Якоби точно придётся хранить в профильном формате, причём она ещё должна быть и симметричной. В этом случае довольно интересно будет как это всё реализовать, ибо есть 2 пути:

1. Подбирать все задачи так, чтобы можно было добиться симметричности матрицы (будет гемморой в случаях, когда $m \neq n$);

2. Менять формулы вычисления так, чтобы матрицы были симметричными (и тогда придётся вносить свою коррекцию в решение Δx).

Скорее всего, для данной задачи придётся делать именно второе, ибо в варианте 6 производные придётся считать численно, и всё равно не факт, что выйдет симметричная матрица

Примерный алгоритм симметризации:

$$Ax = F$$

$$A^T Ax = A^T F$$

В целом не сильно геморройно, вектор x будет получаться таким каким нужно, но лишние затраты на то, чтобы перемножить две матрицы и матрицу на вектор, лучше выбрать другой метод решения СЛАУ, например, LU в профильном формате.

Хранение функций

Функции можно хранить в следующих форматах:

1. Каждую функцию отдельно, например `F_1, F_2, F_3` и т.д. Тогда вызов функций будет выглядеть как:

```
    double a = F_1(x);
    double b = F_2(x);
    ...
```

Но в таком случае будет проблема с расширяемостью программы и невозможностью запихнуть всё в циклы;

2. Все функции можно объединить в массив следующего вида: `vector<function<double(const vector<double>&)>> F`, и тогда вызов функций будет выглядеть так:

```
    double a = F[0](x);
    double b = F[1](x);
    ...
```

А создание и инициализация массива функций будет выглядеть так:

```
    vector<function<double(const vector<double>&)>> F = {
        [](const vector<double>& x) {return 2*x[0] + 3*x[1]*x[1];},
        [](const vector<double>& x) {return 1.0/2.0*x[0] + 1*x[1];},
```

```
    };
```

Да, выглядит громоздко, зато позволит менять данные в программе, не переделывая при этом всю программу, можно будет запихнуть всё в циклы;

3. Просто создать общую функцию, в которой будет через кейс выбираться нужная функция:

```
    double F(int funcNum, const vector<double>&x) {
        switch (funcNum){
            case 0: return 2*x[0] + 3*x[1]*x[1];
            case 1: return 1.0/2.0*x[0] + 1*x[1];
            default: throw exception("Wrong function number");
        };
    }
```

В таком случае единственный недостаток - нельзя будет оперативно узнать количество функций, нужно вручную его вводить и менять. Зато выглядит не так громоздко, и вызов функций тоже будет выглядеть довольно просто:

```
    double a = F(0, x);
    double b = F(1, x);
    ...
```

Хранение производных

Производные функций можно хранить в нескольких вариантах:

1. Создать отдельно множество функций с производными, по типу `F_1_1`, `F_1_2`, `F_2_1` и т.д., но это породит чрезмерно много лишних функций;
2. Создать для каждой функции массив функций производных, например `dif_F_1`, `dif_F_2` и т.д., тогда будет гораздо меньше функций, но всё ещё нельзя будет полноценно пользоваться циклами и придётся каждый раз сильно менять прогу;
3. Создать общий массив массивов функций производных `dif_F`, тогда количество бесхозно болтающихся функций сократится, и так же можно будет пользоваться циклами, не переделывая при этом каждый раз программу

Создание и инициализация массива массивов функций будет выглядеть как-то так:

```
    vector<vector<function<double(const vector<double>&)>>> dif_F = {
        {
            [](const vector<double>& x) { return 2; }
            [](const vector<double>& x) { return 6*x[1]; }
        },
        ...
```

```
    }
```

Вызов такой функции будет выглядеть соответственно следующим образом:

```
    double a1 = dif_F[0][0](x);
    double a2 = dif_F[0][1](x);
    ...
```

4. Создать функцию `dif_F` с производными, тогда всё будет выглядеть не так страшно и красиво:

```
    double dif_F(int funcNum, int valNum, const vector<double>& x) {
        switch(funcNum){
            case 0: {
                switch (valNum){
                    case 0: return 2;
                    case 1: return 6*x[1];
                    default: throw exception("Wrong value number");
                };
            },

            case 1: {
                ...
            }

            default: throw exception("Wrong function number");
        };
    }
```

Вызов такой функции будет выглядеть соответственно следующим образом:

```
    double a1 = dif_F(0,0,x);
    double a2 = dif_F(0,1,x);
    ...
```

Генерация матрицы Якоби в профильном формате

Матрица Якоби: матрица прозводных вида:

$$A_{ij} = \frac{\partial F_i}{\partial x_j}$$

Для того, чтобы была возможность решать СЛАУ с матрицей Якоби, необходимо, чтобы она была квадратной, надо это учитывать.

Всего может быть 2 кейса, когда матрица Якоби будет не квадратной и нужно будет учитывать маски:

1. $m \leq n$ (переменных больше, чем функций);
2. $m \geq n$ (функций больше, чем переменных).

В первом случае, маска будет применяться следующим образом:

1. Берётся квадратная матрица под матрицу Якоби
2. Бегит по функциям и переменным:
    1. Если в маске стоит `0`, то скипаем переменную для функции
    2. Если стоит `1`, то считаем производную по этой переменной для функции

Аналогично формируется вектор `-F`.

Во втором случае, аналогично:

1. Берётся квадратная матрица под матрицу Якоби
2. Бегит по функциям:
    1. Если в маске стоит `0`, то скипаем функцию
    2. Если стоит `1`, то считаем все производные для этой функции

Матрица Якоби должна быть в профильном формате, но пожалуй проще всего будет её генерировать в плотном формате и затем переводить в профильный формат.