Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Факультет: Институт информационных технологий, математики и механики

Кафедра: Прикладная математика

Отчёт по научно-исследовательской практике

Тема:

Осциллограмма движения для многопоршневого механизма

Выполнил студент:

Алексеев Данила Андреевич

Группа: 3821Б1ФИ1

Научный руководитель:

Доцент

Никифорова Ирина Владимировна

Нижний Новгород, 2024 г.

Содержание

1 Введение

В современных инженерных и технических решениях широко используются механизмы, подверженные воздействию виброударных нагрузок. Эти нагрузки могут возникнуть как в результате внешних факторов, так и в процессе работы механизмов. Виброударные явления могут оказывать значительное воздействие на долговечность, надежность и эффективность механизмов, а также являются важным аспектом в области безопасности.

Данное исследование направлено на анализ и понимание динамических характеристик виброударных механизмов с целью оптимизации их конструкции, повышения стойкости к нагрузкам и снижения рисков возникновения аварийных ситуаций.

2 Постановка задачи

В рамках данного исследования будет разработана программа, предназначенная для визуализации влияния различных параметров виброударного механизма на показатели системы при его работе на многопоршневом механизме. Основное внимание будет уделено количеству поршней в механизме, рассматриваемому как ключевой параметр, определяющий его характеристики.

Целью исследования является анализ воздействия количества поршней на динамические свойства механизмов, а также создание программных инструментов для визуализации этой зависимости в виде осциллограмы. Полученные результаты и визуализации смогут быть использованы для оптимизации конструкции виброударных механизмов и принятия соответствующих решений в процессе их проектирования.

3 Математическая модель

Рассматриваемый механизм (рис. 1) состоит из корпуса, внутри которого расположен эксцентриковый вал с маховиками на концах. На этом валу установлены эксцентриковые механизмы, каждый из которых состоит из двух эксцентриков, вставленных друг в друга с возможностью изменения положения шайб, что позволяет регулировать величины эксцентриситетов r_i и сдвигов по фазам φ_i между ними ($i = 1, 2, 3, ..., N$). На свободных концах шатунов установлены шарнирно поршни-ударники. Эксцентриковые механизмы вместе с шатунами и поршнями-ударниками преобразуют вращательное движение вала с постоянной циклической частотой вращения маховика ω в возвратно-поступательное движение корпуса относительно стоек. ПУ расположены один внутри другого и ударяются каждый по своей наковальне высоты h_i.

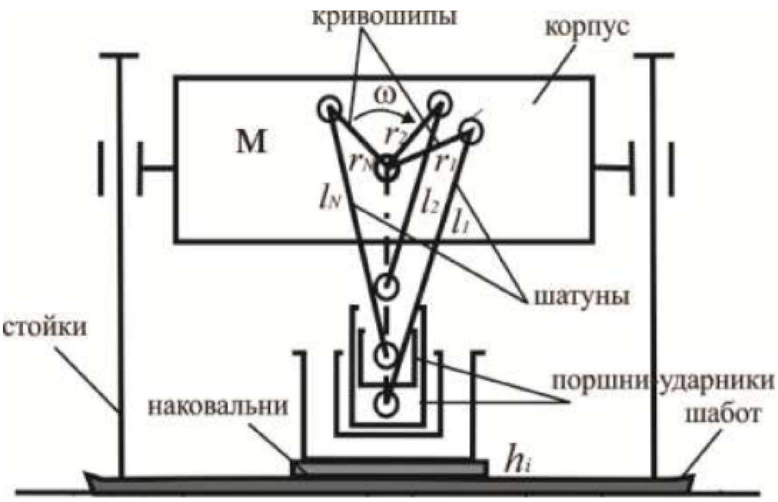

Рис. 1: Схема ударно-вибрационного механизма с кривошипношатунным возбудителем колебаний.

3.1 Размерный вид

Колебания корпуса совершаются относительно того механизма, сумма проекций геометрических размеров на вертикальную ось которого в данный момент наибольшая. Удар о наковальню одновременно одного или нескольких ПУ происходит либо при смене взаимодействующих с наковальней ПУ, либо после отрыва корпуса вместе с эксцентриковыми механизмами.

Пренебрегая массами ПУ, шатунов и кривошипов, уравнение свободного движения системы при $y_{pi} > 0$ может быть записано в виде

$$M\frac{d^2y}{dt^2} = -Mg$$

y - координата центра масс корпуса, отсчитываемая от наковальни, y_{pi} - величины, характеризующие отклонение оснований i - ого ПУ, g - ускорение силы тяжести.

При соприкосновении одного из ПУ с наковальней $y_{pi} = 0$ происходит мгновенное ударное взаимодействие, такое, что если $\dot{y}_{pi}^{-}$ и $\dot{y}_{pi}^{+}$ скорости i-го ПУ непосредственно до и после ударного взаимодействия соответственно, то

$$\dot{y}_{pi}^{+} = -R\dot{y}_{pi}^{-}$$

Положение эксцентриситетов длиной r_i будем измерять углами $\omega t - \varphi$ отсчитываемые от вертикальной оси (рис. 2). Тогда можно вывести соотношение:

$$y_{pi} = y - s_i + r_i \cos(\omega t - \varphi_i) - \sqrt{l_i^2 - r_i^2 \cos(\omega t - \varphi_i)}$$

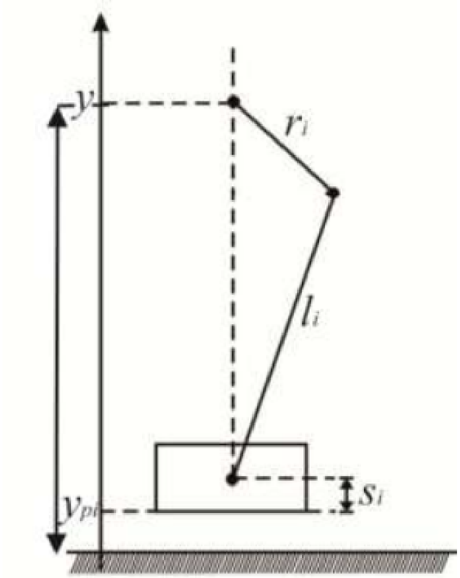

Рис. 2: Геометрическая схема.

Получаем уравнения движения механизма в размерном виде:

$$\begin{cases} \begin{cases} y_{pi} > 0 \\ M\frac{d^2y}{dt^2} = -Mg \end{cases} \\ \begin{cases} y_{pi} = 0 \\ \dot{y}_{pi}^{+} = -R\dot{y}_{pi}^{-} \end{cases} \\ y_{pi} = y - s_i + r_i \cos(\omega t - \varphi_i) - \sqrt{l_i^2 - r_i^2 \cos(\omega t - \varphi_i)} \end{cases} \tag{1}$$

С учётом условия $r_i << l, l_i \approx l$ последнее уравнение будет:

$$y_{pi} = y - s_i + r_i \cos(\omega t - \varphi_i) - l$$

3.2 Безразмерный вид

Перейдём к безразмерным: времени $\tau = \omega t$, координате $x = (y - s_i - l)/l$ центра вращения корпуса, параметрам $\mu = r_1/l, \quad \gamma_i = r_i/r_1, \quad \varepsilon_i = (s_i - s_2)/l$, $p = g/\omega^2 l$. А также введя в рассмотрение функции $f_i(\tau) = \varepsilon_i - \mu\gamma_i \cos(\tau - \varphi_i)$, приходим к уравнениям, описывающим ударно-колебательные движения механизма в виде

$$\begin{cases} \frac{d^2x}{d\tau^2} = -p, & x > f(\tau) \\ \frac{dx}{d\tau}|_+ = -R\frac{dx}{d\tau}|_- + (1+R)\frac{df(\tau)}{d\tau}, & x = f(\tau), \quad \dot{x} - \frac{df}{d\tau} < 0 \end{cases} \tag{2}$$

$$f(\tau) = \max\{f_1(\tau), f_2(\tau), ..., f_N(\tau)\}, \quad f_i(\tau) = \mathcal{E}_i - \mu\gamma_i \cos(\tau - \varphi_i) \tag{3}$$

4 Теоретическая часть программы

Программа написана на языке C++. Для графического интерфейса и отображения графика используется фреймворк qt.

Для построения осциллограммы движения на основе системы (2) используется метод Рунге-Кутты четвертого порядка.

Описание метода Рунге-Кутты:

Метод Рунге-Кутты четвертого порядка предполагает вычисление новых значений функции в следующей точке через линейную комбинацию оценок (коэффициентов), полученных в нескольких промежуточных точках. Процесс расчета включает следующие шаги:

1. Выбор начальных условий $\tau_0, x_0, \dot{x}_0$

2. Вычисление промежуточных коэффициентов

   Разложим ДУ второго порядка на систему уравнений первого порядка. Тогда $g(\tau)$ - функция производной $dx/d\tau$, $\quad f(\dot{x})$ - функция производной $d^2x/d\tau^2$

   Для каждого шага вычисляются четыре коэффициента k и l, которые учитывают значения функции и её производных в разных точках внутри текущего шага:

$$
\begin{aligned}
k_1 &= \Delta\tau \cdot f(\dot{x}_n) \\
l_1 &= \Delta\tau \cdot g(\tau_n) \\
k_2 &= \Delta\tau \cdot f(\dot{x}_n + \frac{l_1}{2}) \\
l_2 &= \Delta\tau \cdot g(\tau_n + \frac{\Delta\tau}{2}) \\
k_3 &= \Delta\tau \cdot f(\dot{x}_n + \frac{l_2}{2}) \\
l_3 &= \Delta\tau \cdot g(\tau_n + \frac{\Delta\tau}{2}) \\
k_4 &= \Delta\tau \cdot f(\dot{x}_n + l_3) \\
l_4 &= \Delta\tau \cdot g(\tau_n + \Delta\tau)
\end{aligned}
$$

   Здесь $\Delta\tau$ - шаг интегрирования.

3. Обновление значений функции

   Новые значения x и $\dot{x}$ вычисляются как средневзвешенные суммы промежуточных коэффициентов:

$$
\begin{aligned}
x_{n+1} &= x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\
\dot{x}_{n+1} &= \dot{x}_n + \frac{1}{6}(l_1 + 2l_2 + 2l_3 + l_4)
\end{aligned}
$$

   Здесь x_n и $\dot{x}_n$ - значения функции и её производной на текущем шаге, а x_{n+1} и $\dot{x}_{n+1}$ - значения на следующем шаге.

   Так же переходим к следующему шагу: $\tau_{n+1} = \tau_n + \Delta\tau$ и повторяем шаги 2 и 3.

В нашем случае нужно ещё учесть ударное взаимодействие с поверхностью $f(\tau)$ (3). Поэтому на каждой итерации добавляем проверку $x = f(\tau)$ и если условие выполняется, домножаем $\dot{x}$ на $-R$.

5 Описание программы

Программа состоит из 4 файлов:

- `main.cpp` – файл, содержащий точку входа программы. Он вызывает открытие окна, реализация которого находится в следующих файлах.
- `mainwindow.h` – файл с описанием класса `mainwindow` и определением методов.
- `mainwindow.ui` – содержит информацию о графическом представлении окна.
- `mainwindow.cpp` – файл, который содержит реализацию методов и вычислений. Разберём этот файл.

Описание файла mainwindow.cpp

1. Инициализация переменных

   В начале нужно проинициализировать все параметры, используемые в программе. Разберём каждую переменную за исключением вспомогательных. Переменные, помеченные как (*) в ходе программы могут меняться посредством графического интерфейса в виде ползунка.

   - `N` – кол-во поршней. (*)
   - `step` – итерационный шаг. Устанавливаем ему значение 0,01.
   - `P` – параметр p системы (2) (*)
   - `R` – параметр R системы (2) (*)
   - `M` – параметр μ системы (2)
   - `xEnd` – переменная, хранящая число, до которого будет вычисляться и выводиться график. Устанавливается значение `MAXX = 50` делённое на `P` и прибавляется `step` для корректного вывода конца графика.
   - `xBegin, yBegin, zBegin` – переменные, содержащие начальные значения (Метод Рунге-Кутты пункт 1). Переводя в математический формат: $\tau_0 = 0, x_0 = 1, \dot{x}_0 = 1,5$
   - `E1, E2, E3` – параметры $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$ уравнения (3)
   - `U1, U2, U3` – параметры $\varphi_1, \varphi_2, \varphi_3$ уравнения (3)
   - `Y1, Y2, Y3` – параметры $\gamma_1, \gamma_2, \gamma_3$ уравнения (3)

```
#define MAXX 50
static int N = 2, tmpN = N;
const double step = 0.01;
static double P = 1.5, tmpP = P, R = 0.95, tmpR = R, M = 0.15,
              xEnd = MAXX / P + step;
const double xBegin = 0, yBegin = 1, zBegin = 1.5;
static double E1 = 0, E2 = 0, E3 = 0;
static double U1 = 0, U2 = 2.5, U3 = 5;
static double Y1 = 1, Y2 = 1, Y3 = 3;
```

2. Определение функций

- double f(double z) – реализация функции $f(\dot{x})$ (Метод Рунге-Кутты пункт 2).
- double g(double x) – реализация функции $g(\tau)$ (Метод Рунге-Кутты пункт 2).
- double surfaceFoo(double x, double E, double Y, double U) – реализация функции поверхности $f_i(\tau)$ (3).

```
double f(double z) { return z; }

double g(double x) { return M * cos(x) - P; }

double surfaceFoo(double x, double E, double Y, double U) {
    return E - M * Y * cos(x - U);
}
```

- std::vector<std::pair<double, double>> buildFoo(funcSurface _f) – реализация функции построения поверхности $f(\tau)$ (3). В результате выполнения функции мы получаем вектор точек по которым далее строится график поверхности.

```
std::vector<std::pair<double, double>> buildFoo(funcSurface _f) {
    std::vector<std::pair<double, double>> points;
    if (N == 1) {
        for (double x = 0; x <= MAXX; x += step) {
            points.push_back(std::make_pair(x, _f(x, E1, Y1, U1)))
               ;
        }
    } else if (N == 2) {
        for (double x = 0; x <= MAXX; x += step) {
            points.push_back(std::make_pair(
                x, std::max(_f(x, E1, Y1, U1), _f(x, E2, Y2, U2)))
                   );
        }
    } else if (N == 3) {
        for (double x = 0; x <= MAXX; x += step) {
            points.push_back(std::make_pair(
                x, std::max(_f(x, E1, Y1, U1),
                            std::max(_f(x, E2, Y2, U2), _f(x, E3,
                               Y3, U3)))));
        }
    }

    return points;
}
```

- std::vector<std::pair<double, double>> rungeKutta(double x0, double y0, double z0, funcF _f, funcG _g, std::vector<std::pair<double, double>> _surface) – реализация метода Рунге-Кутты. В результате выполнения функции мы получаем вектор точек по которым далее строится график.

```
std::vector<std::pair<double, double>> rungeKutta(
    double x0, double y0, double z0, funcF _f, funcG _g,
    std::vector<std::pair<double, double>> _surface) {
    auto findYOnSurface = [&](double x) {
        double closestY = _surface[0].second;
        double minDist = std::fabs(_surface[0].first - x);
        for (const auto& point : _surface) {
            double dist = std::fabs(point.first - x);
            if (dist < minDist) {
                minDist = dist;
                closestY = point.second;
            }
        }
        return closestY;
    };
    std::vector<std::pair<double, double>> points;
    while (x0 <= xEnd) {
        points.push_back(std::make_pair(x0, y0));
        double k1 = step * _f(z0);
        double l1 = step * _g(x0);
        double k2 = step * _f(z0 + l1 / 2);
        double l2 = step * _g(x0 + step / 2);
        double k3 = step * _f(z0 + l2 / 2);
        double l3 = step * _g(x0 + step / 2);
        double k4 = step * _f(z0 + l3);
        double l4 = step * _g(x0 + step);

        double nextY = y0 + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
        double nextZ = z0 + (l1 + 2 * l2 + 2 * l3 + l4) / 6;

        double _surraceIt = findYOnSurface(x0 + step);
        double derivative = calculate_derivative(_surface, x0 +
           step);
        if (nextY <= _surraceIt && nextZ - derivative < 0)
            nextZ = -R * nextZ + (1 + R) * derivative;

        x0 += step;
        if (nextY < _surraceIt) {
            y0 = _surraceIt;
        } else {
            y0 = nextY;
        }
        z0 = nextZ;
    }

    return points;
}
```

- `void MainWindow::draw()` – реализация метода построения графиков по полученым точкам.

```cpp
void MainWindow::draw() {
    QVector<double> x, y, xSurf, ySurf;
    for (const auto& point : surface) {
        xSurf.append(point.first);
        ySurf.append(point.second);
    }
    double maxY = std::numeric_limits<double>::min();
    for (const auto& point : points) {
        x.append(point.first);
        y.append(point.second);
        if (maxY < point.second) maxY = point.second;
    }

    ui->widget->clearGraphs();

    ui->widget->xAxis->setRange(0, xEnd);
    ui->widget->yAxis->setRange(-maxY / 3, maxY + maxY / 3);

    ui->widget->addGraph();
    ui->widget->graph(0)->setData(x, y);
    ui->widget->graph(0)->setPen(QColor(Qt::blue));

    ui->widget->addGraph();
    ui->widget->graph(1)->setData(xSurf, ySurf);
    ui->widget->graph(1)->setPen(QColor(Qt::red));

    ui->widget->replot();
}
```