Code

Standard Linear Regression

```
std::pair<double, double> LinReg::trendline(const std::vector<double>& x, const
std::vector<double>& y) {
  if (x.size() != y.size()) {
    throw std::runtime_error("Vectors not the same size");
  }
  const int n = x.size();
  const double sum_x = std::accumulate(x.begin(), x.end(), 0.0);
  const double sum_y = std::accumulate(y.begin(), y.end(), 0.0);
  const double sum_x_sq = std::inner_product(x.begin(), x.end(), x.begin(), 0.0);
  const double sum_xy_pair = std::inner_product(x.begin(), x.end(), y.begin(), 0.0);
  const double slope = ((n * sum_xy_pair) - (sum_x * sum_y)) / ((n * sum_x_sq) - sum_x *
sum_x);
  const double intercept = (sum_y - (slope * sum_x)) / n;

  return std::make_pair(slope, intercept);
}
```

Polynomial Regression (Using a Least Squares Approach)

```
std::vector<double> LinReg::polynomial_trendline(
    const std::vector<double>& x,
    const std::vector<double>& y,
    const int& terms) {
  std::vector<double> coefficients;

  if (x.size() != y.size()) {
    throw std::runtime_error("Vectors not the same size");
  }

  size_t N = x.size();
  int n = terms;
  int np1 = n + 1;
  int np2 = n + 2;
  int tnp1 = 2 * n + 1;
  double tmp;

  // X stores the values of sigma(xi^2n)
  std::vector<double> X(tnp1, 0);
  for (int i = 0; i < tnp1; ++i) {
    for (size_t j = 0u; j < N; ++j) {
      X[i] += std::pow(x[j], i);
    }
  }
```

```
// A stores the final coefficients
std::vector<double> A(np1);

// B is an augmented matrix
std::vector<std::vector<double>> B(np1, std::vector<double>(np2, 0));

for (int i = 0; i <= n; ++i) {
  for (int j = 0; j <= n; ++j) {
    B[i][j] = X[i + j];
  }
}

std::vector<double> Y(np1, 0);
for (int i = 0; i <= n; ++i) {
  for (int j = 0; j <= n; ++j) {
    Y[i] += std::pow(x[j], i) * y[j];
  }
}

// Put Y as last column of B
for (int i = 0; i <= n; ++i) {
  B[i][np1] = Y[i];
}

n += 1;
int nm1 = n - 1;

// Get pivot positions of B matrix
for (int i = 0; i < n; ++i) {
  for (int k = i+1; k < n; ++k) {
    if (B[i][i] < B[k][i]) {
      for (int j = 0; j <= n; ++j) {
        tmp = B[i][j];
        B[i][j] = B[k][j];
        B[k][j] = tmp;
      }
    }
  }
}

// Perform the gaussian elimination
for (int i = 0; i < n; ++i) {
  for (int k = i+1; k < n; ++k) {
    double t = B[k][i] / B[i][i];
    for (int j = 0; j <= n; ++j) {
      B[k][j] -= t * B[i][j];
    }
  }
}

// Back substitute up the call tree
for (int i = nm1; i >= 0; --i) {
  A[i] = B[i][n];
```

```
    for (int j = 0; j < n; ++j) {
      if (j != i) {
        A[i] -= B[i][j] * A[j];
      }
    }
    A[i] /= B[i][i];
  }

  coefficients.resize(A.size());
  for (size_t i = 0u; i < A.size(); ++i) {
    coefficients[i] = A[i];
  }

  return coefficients;
}
```