Introduction

Particle tracking is an important part of the processing and analysis of data received from particle detectors, such as the Compact Muon Solenoid (CMS). Tracking is the step that determines the momentum of charged particles escaping from the collision point. It identifies individual particles by reconstructing their trajectories from points where charged particle “hits” were measured by the detector and interpreting them.[1] Due to the Lorentz force, charged particles move in a helical motion when affected by the magnetic field (neglecting other effects due to material interactions, etc). This means we can figure out a specific particle trajectory through the detector by fitting a helix function to data points in such a way that the distance from the data points and the helix would be minimized. In mathematical terms, we need to find optimal helix parameters by minimizing a loss function composed of the sum of least squared distances, thus giving the best estimation of these parameters. For this purpose we can use Clad to efficiently minimize the loss function. This tutorial, explaining one way to do that, is written to accompany the code contained in the helix-example repository.

Levenberg-Marquardt algorithm

To solve this nonlinear least squares problem we will be using the Levenberg-Marquardt algorithm(1). The Levenberg-Marquardt algorithm combines two optimization methods: gradient descent and Gauss-Newton. Its behaviour changes based on how close the current coefficients are to the optimal value. When the coefficients are far from optimal, it uses gradient descent, which takes steps in the direction of steepest descent. When the coefficients are close to optimal, it uses Gauss-Newton, which assumes the problem is locally quadratic and finds the minimum of this quadratic. This combination allows the algorithm to provide a more reliable and efficient optimization than the other two methods mentioned.[1]

The implementation

First we define our helix. We chose to do so in the Cartesian coordinates. This is done here:

```
void HelixPoint(double a, double b, double c, double d, double alph,
                double bet, double t, double output[3])
{
   double x = a * (c + std::cos(t));
   double y = a * (d + std::sin(t));
   double z = a * b * t;
   output[0] = x;
   output[1] = y;
   output[2] = z;
   Rotate(x, y, z, alph, bet, output);
}
```

Where `Rotate` is described by

```
inline void RotateAlph(double x, double y, double z,
                       double alph, double output[3]) {
   output[0] = x;
   output[1] = y * cos(alph) - z * sin(alph);
   output[2] = y * sin(alph) + z * cos(alph);
}


inline void RotateBet(double x, double y, double z,
                      double bet, double output[3]) {
   output[0] = x * cos(bet) + z * sin(bet);
   output[1] = y;
   output[2] = -x * sin(bet) + z * cos(bet);
}


inline void Rotate(double x, double y, double z, double alph,
                   double bet, double output[3]) {
   double point[3];
   RotateAlph(x, y, z, alph, point);
   RotateBet(point[0], point[1], point[2], bet, output);
}
```

This code describes a circular helix of radius `a` and a slope `1/b` (in the net direction of travel, which is the z direction in the code above). The helix is shifted from the starting coordinates by distances `c` and `d` in the x and y directions, respectively. It can also be rotated around the x axis and then the y axis. The angle `alph` represents how much the helix is rotated counterclockwise around the x axis, when viewed from the positive direction of the axis. Likewise for `bet` and the y axis.

Next, for demonstration purposes, we generate a set of points using `GenerateFlawedPoints`. We pick some set of helix parameters and scan `t` in some range to generate an array containing `nr_of_points` data points. The data points are generated by calculating a point on the helix with given parameters at the time `t` using `HelixPoint` and also adding some randomness (representing measurement error), so that the helix points do not correspond to a perfect helix. We will use these points to then determine estimated helix parameters.

Let's now implement the Levenberg-Marquardt algorithm. The equation that dictates how to update the parameters in the Levenberg-Marquardt algorithm is this:

$$(J^T W J + \lambda I)\, h_{lm} = J^T W (y - \hat{y})$$

In this equation $\hat{y}$ represents a vector containing the differences between the data points and the closest point of each to the helix (given a set of helix parameters: a, b, c, d alph, bet). y is

a vector of expected values: in our case, ideally, we expect the difference between the data point and the helix to be zero.
The jacobian is a matrix of partial derivatives of ŷ with respect to the parameters we are searching for. So in our case it is a matrix of size 6 x `nr_of_points`, where `nr_of_points` is the number of data points and 6 is the number of parameters we have: `a`, `b`, `c`, `d`, `alph`, `bet`. λ is the damping coefficient, which determines the behaviour of the algorithm. It is not static: if the sum of the distances squared is smaller than in the previous iteration, we make it smaller, otherwise, we increase its size.

We start out by guessing the initial parameters, except `b`. Currently, the code doesn't support determination of `b` and one right now needs to set `b` to a number close to, but not equal to zero (like 0.1) to find the other parameters[2]. However, that is fine for our purposes, since knowing the magnetic field in an experiment constrains `b` for a given momentum (eg, the ratio of `a` and `b` is known). It is also expected for `alph` and `bet` to be between -π and π.

Next, we calculate the jacobian using `clad::gradient`[3]. We determine its transpose, find the closest distances of the points to the helix, do some matrix multiplication and we are left with an equation to solve for h. We solve it using the Gaussian elimination method. We update the parameters with h, recalculate the sum of all squared distances and change lambda accordingly.[4] The process is repeated for some number of iterations or until there is almost no change between the recalculated sum of all squared distances between several iterations.

To find the closest distance of a point to a helix, we do some scaling so that our helix is now defined by (cos$t$,sin$t$,ht). For a given point P(i,j,k), let Q be the closest point on the helix. The line segment connecting P and Q must be perpendicular to the helix's tangent line at Q, which is (−sin$t$,cos$t$,h). Knowing that the dot product of two perpendicular vectors is 0 leads to:

$$-(\cos t-\mathrm{i})\sin t+(\sin t-\mathrm{j})\cos t+(ht-\mathrm{k})h=0$$

This simplifies to $A\sin(t+B)+Ct+D=0$ for some constants A,B,C,D. [2]
To find the solution, we perform a binary search, as seen in [equations.h].

The function, taking all of this into account and returning the parameter `t`, during which the point is closest to the helix, is `HelixClosestTime`:

```
inline double HelixClosestTime(double a, double b, double c,
                               double d, double alph, double bet,
                               double x, double y, double z)
{
      auto MY_PI = std::numbers::pi_v<double>;
      double point[3];
      UnRotate(x, y, z, alph, bet, point);
      point[0] /= a;
      point[1] /= a;
      point[2] /= a;
      point[0] -= c;
      point[1] -= d;
      double A = std::sqrt(point[0] * point[0] + point[1] * point[1]);
      double B = std::atan2(-point[1], point[0]);
```

```
	double C = b * b;
	double D = -point[2] * b;

	double mi = point[2] / b - MY_PI;
	double ma = point[2] / b + MY_PI;
	double t1 = SolveSinPlusLin(A, B, C, D, mi, ma);

	double ans = t1;
	HelixPoint(a, b, c, d, alph, bet, ans, point);
	double dist = DistanceSquareA(point, x, y, z);
	for (double t = mi; t < ma; t = t)
	{
		double ttt = NextSinPlusInflection(A, B, C, t);

		if (ttt == t)
		{
			break;
		}

		double cur = SolveSinPlusLin(A, B, C, D, t, ttt);
		t = ttt;
		HelixPoint(a, b, c, d, alph, bet, cur, point);
		double dist2 = DistanceSquareA(point, x, y, z);

		if (dist2 < dist)
		{
			dist = dist2;
			ans = cur;
		}
	}

	return ans;
}
```

And where `SolveSinPlusLin` is the binary search implementation.

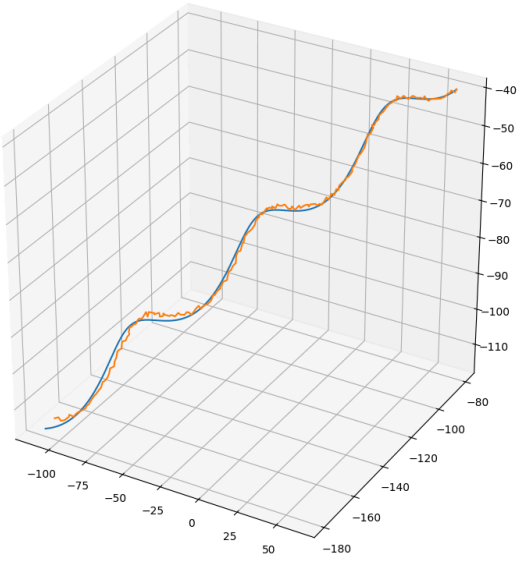
```
double SolveSinPlusLin(double A, double B, double C, double D, double mi, double ma)
{
	for (int i = 0; i < 100; i++)
	{
		double mid = (mi + ma) / 2;
		double vmi = EvaluateSinPlusLin(A, B, C, D, mi);
		double vmid = EvaluateSinPlusLin(A, B, C, D, mid);
		double vma = EvaluateSinPlusLin(A, B, C, D, ma);

		if (vmi < 0 and 0 < vmid)
		{
			ma = mid;
		}
		else if (vmid < 0 and 0 < vma)
		{
			mi = mid;
		}
		else if (vmid < 0 and 0 < vmi)
		{
			ma = mid;
		}
		else if (vma < 0 and 0 < vmid)
		{
			mi = mid;
		}
		else
		{
			break;
			mi = mid;
		}
	}

	double x = (mi + ma) / 2;
	return x;
}
```

The loop in `HelixClosestTime` checks if we really found the global minimum.

In the end, one can produce a graph by piping the output to `graph.py`. The way it is currently implemented, the graph shows the original generated points and the best helix approximation `LevenbergMarquardt` produced.



The graph above is an example of the approximation achieved by the Levenberg-Marquardt function. Note that parameter `b,` used for the fitted helix here, is given by the user, not the algorithm.

Overall, one can say that the parameters produced by the minimisation are quite close to expected results and the function seems to work well when using data with no added randomness. However, added noise sometimes skews the results and the fit ends up being visibly inaccurate.

Appendix

(1) That is because b is the parameter controlling the pitch of the helix and our distance function unfortunately has a local minimum at b ≈ 0. This occurs because for very small values of b, the helix practically turns into a cylinder. A cylinder will always pass through points that would otherwise be the minima on a helix with the correct b value. The way our function is written, it doesn't have the solution to overcome this local minimum. Making b a constant and leaving it out from the future calculations also seems to lead to highly inaccurate answers.

(2) Another improvement would be to use clad::jacobian, however, I ran into a problem where Clad's execute method doesn't support arguments that have length specified as a variable (a template argument may not reference a variable-length array type). Const variables do not solve the problem.

(3) Perhaps a better way to showcase Clad (but not necessarily a better way to approximate a helix) would be to use the gradient descent method, since it is more simplistic, however, the implementation found in `fitter.h` gets stuck in a local minimum that is very far off from the actual expected results.

(4) Unfortunately, due to the added randomness in `GenerateFlawedPoints`, sometimes the end result is not as expected.

References

[1] https://people.duke.edu/~hpgavin/lm.pdf
[2] https://math.stackexchange.com/questions/13341/shortest-distance-between-a-point-and-a-helix