1 Linear fitting as an example case

Suppose we have a data set containing N data points (t_i, x_i) which we are trying to fit to a linear function $f(t) = mt + b$. We can quantify how far each data point is from the fit by looking at the residuals

$$\Delta_i = x_i - f(t_i). \tag{1}$$

Each Δ_i quantifies how far the fit is off from a single data point, but we need some way to quantify how far the fit is from the entire data set, i.e. a way to account for all the Δ_i simultaneously. We cannot simply add them because some residuals will be positive and some will be negative and these will tend to cancel, whereas we would like all data points to contribute to the overall difference regardless of sign. We can avoid this cancellation issue by squaring and define

$$\chi^2 = \sum_{i=1}^{N} \Delta_i^2 = \sum_{i=1}^{N} \left(x_i - f(t_i)\right)^2. \tag{2}$$

This χ^2 gives us a measure of fitness or, in other words, a quantitative way to describe how well the model aligns with the data set. Note that the definition of χ^2 given by (2) does not depend on the form of $f(t)$; we happen to be using a linear $f(t)$ here (because this is a tractable problem), but this interpretation can be applied to any other model we are trying to fit. In the present case though,

$$\chi^2 = \sum_{i=1}^{N} \left(x_i - mt_i - b\right)^2. \tag{3}$$

Given χ^2, we now have a way to say a fit is 'best;' we want the one which has the smallest χ^2 value. For a linear fit, we are looking for m and b which minimize χ^2. Minimization is a standard problem in calculus. We can find such parameters by solving the equations[1]

$$0 = \frac{\partial \chi^2}{\partial m} \tag{4a}$$

$$0 = \frac{\partial \chi^2}{\partial b}. \tag{4b}$$

Slogging through some algebra, the slope and intercept can be determined by

$$m = \frac{\dfrac{1}{N}\left[\displaystyle\sum_{i=1}^{N} x_i\right]\left[\displaystyle\sum_{i=1}^{N} t_i\right] - \displaystyle\sum_{i=1}^{N} x_i t_i}{\dfrac{1}{N}\left[\displaystyle\sum_{i=1}^{N} t_i\right]^2 - \displaystyle\sum_{i=1}^{N} t_i^2} \tag{5a}$$

$$b = \frac{1}{N}\sum_{i=1}^{N} x_i - \frac{m}{N}\sum_{i=1}^{N} t_i. \tag{5b}$$

See the appendix in Section 3 for a more thorough derivation connecting (4) to (5).

[1]Note that we are treating χ^2 as our variable of interest and not worrying about χ itself.

2 Damped oscillation fitting code

Even though the code is trying to do a non-linear fit, it uses the same essential strategy laid out in Section 1. However, there is no way to write out f or do the algebra shown in Section 3 so it has to resort to less obvious ways to solve its version of (2).

The code's functions can be thought of as being split into two main groups: those which work together to determine the fit function (using Runge-Kutta), and those which work together to find the best fit parameters (using gradient descent). We'll go through the functions in approximately reverse order from how they appear in the code since it probably makes more logical sense to see how some of these functions are used.

2.1 Runge-Kutta part

The first bunch of functions allow the code to model the cart's motion given a set of fit parameters.

`fit_function` – This is our version of $f(t) = mt + b$. It takes in a set of fit parameters (analogous to m and b) and tells you what the fit function looks like. Because our function is complicated enough that it can't just be written down using basic functions, it doesn't return an equation but instead gives a list of times and the corresponding positions of the cart at those times.

This uses Runge-Kutta in order to solve the physics equation and find the cart's positions at all the times. Runge-Kutta is, in some sense, a fancier version of things like Euler's method or DVAT.[2] There are many different methods which could have been used to do this, but Runge-Kutta is one of the most widespread and useful.

The basic idea is to go step by step and, given the cart's current position and velocity and the equation modeling its motion, figure out where it is at the end of the step. As an example, imagine LoggerPro had taken data every $\Delta t = 0.1$ s. We could use DVATs to model where the cart is every Δt:

$$x(t_0 + \Delta t) = x(t_0) + \Delta t \cdot v(t_0) \tag{6a}$$
$$v(t_0 + \Delta t) = v(t_0) + \Delta t \cdot a(t_0) \,. \tag{6b}$$

By doing this over and over again, we can figure out position and velocity throughout the whole time interval. The `take_step` function is what takes each individual step, so `fit_function` is going to call it over and over.

`take_step` – This is essentially Runge-Kutta's version of (6); it's the same idea, but a more accurate version. You tell it the cart's position and velocity at the beginning of the step (`x0` is just a vector containing $x(t_0)$ and $v(t_0)$) and how big Δt is. The last thing it needs to know then is the acceleration a. Which is where `get_derivatives` comes in.

`get_derivatives` – This is basically where you tell the Runge-Kutta method what equation it is you want it to solve. The set of differential equations which (6) are meant to

[2]'DVAT' is one name given to the standard constant-acceleration equations which relate **D**isplacement, **V**elocity, **A**cceleration, and **T**ime, e.g. $\Delta x = v_0 \Delta t + \frac{1}{2} a \Delta t^2$.

approximate is

$$\frac{dx}{dt} = v \tag{7a}$$

$$\frac{dv}{dt} = a. \tag{7b}$$

So the code needs to know how to compute a in order to solve the equation.[3] This one piece contains essentially all the physics in the entire code.

Newton's law for the cart tells us

$$ma = \sum F = F_{\text{spring}} + F_{\text{friction}} = -k\left(x - x_{\text{eq}}\right) + F_{\text{friction}}, \tag{8}$$

so

$$a = \frac{-k\left(x - x_{\text{eq}}\right) + F_{\text{friction}}}{m}. \tag{9}$$

`get_sign` – This implements the sign function which is used to make sure the drag is always opposing the velocity. Mathematically, it is the piece-wise function

$$\text{sign}(x) = \begin{cases} 1 & x > 0 \\ -1 & x < 0 \\ 0 & x = 0 \end{cases}. \tag{10}$$

2.2 Gradient descent part

The rest of the functions help the code decide what the best fit parameters are. Before getting into the details of the algorithm, a few notes on notation are needed. We introduce a vector $\vec{A}$ which contains all the fit parameters

$$\vec{A} = \langle x_0, v_0, m, k, x_{\text{eq}}, \ldots\rangle \tag{11}$$

and a gradient with respect to each of the fit parameters

$$\vec{\nabla}_A = \left\langle \frac{\partial}{\partial x_0}, \frac{\partial}{\partial v_0}, \frac{\partial}{\partial m}, \frac{\partial}{\partial k}, \frac{\partial}{\partial x_{\text{eq}}}, \ldots \right\rangle. \tag{12}$$

Then χ^2 for these fit parameters would be calculated as

$$\chi^2(\vec{A}) = \sum_{i=1}^{N} \left(x_i - f(t_i; \vec{A})\right)^2, \tag{13}$$

where we now explicitly show f and χ's dependence on the fit variables. Up to this point, we've only really shown the t dependence of f. But it also depends on the fit parameters. We could write our linear function as something like $f(t; m, b)$ and it is maybe a bit clearer then what something like $\partial f(t; m, b)/\partial m$ means.

[3]For some differential equations, the dx/dt won't be so simple and it will be non-trivial to compute the right hand side of both (7). Because Newton's law cares about acceleration, in physics we often focus a lot more on (7b) and (7a) kind of just comes along for the ride.

`gradient_descent` – This function tries to find the best fit parameters by trying to solve our version of (4). Like the Runge-Kutta method, there are a number of other ways we could have chosen to solve the equations. The gradient descent method essentially tries to find the minimum by guess and check, though it is able to make somewhat informed guesses.

The `gd_iterations` parameter tells it how many guesses to make, the `alpha` parameter tells it how much to change its guess each time, the `data` (i.e., the stuff which came from LoggerPro) lets it figure out how to lower $\chi^2(\vec{A})$ so that it can make its next guess, and `initial_fit_parameters` gives it an initial guess to get started. For each guess, it will compute how much $\chi^2(\vec{A})$ is changing with each fit parameter. It can then get a new guess for the fit parameter using the equation

$$\vec{A}_{\text{new guess}} = \vec{A}_{\text{old guess}} - \alpha \cdot \vec{\nabla}_A \chi^2(\vec{A}). \tag{14}$$

Notice that α appears in (14) to control how much it's updating the guess each time. The strategy is pretty simple then: just apply (14) `gd_iterations` times. The difficult part is computing all the derivatives of $\chi^2(\vec{A})$, which is what `grad_chi2` does. As an example, applying (14) to the spring constant would look like

$$k_{\text{new guess}} = k_{\text{old guess}} - \alpha \cdot \frac{\partial \chi^2(\vec{A})}{\partial k} \tag{15}$$

and (14) contains a copy of (15) for each fit parameter.

`grad_chi2` – This is essentially just the chain rule. The derivative of $\chi^2(\vec{A})$ can be written out in terms of the derivative of $f(t;\vec{A})$

$$\vec{\nabla}_A \chi^2(\vec{A}) = \vec{\nabla}_A \sum_{i=1}^{N} \left(x_i - f(t_i;\vec{A})\right)^2 = -\sum_{i=1}^{N} 2\left(x_i - f(t_i;\vec{A})\right) \vec{\nabla}_A f(t_i;\vec{A}). \tag{16}$$

Again taking the spring constant as an example, one of the equations contained in (16) would be

$$\frac{\partial \chi^2(\vec{A})}{\partial k} = \frac{\partial}{\partial k} \sum_{i=1}^{N} \left(x_i - f(t_i;\vec{A})\right)^2 = -\sum_{i=1}^{N} 2\left(x_i - f(t_i;\vec{A})\right) \frac{\partial f(t_i;\vec{A})}{\partial k}, \tag{17}$$

with similar equations for m and all the other fit parameters. Now the challenge is to compute each of the derivatives of f.

`grad_fit_function` – As with the previous couple functions, we'll continue taking the k parameter as one example with the understanding that there are, again, many copies of the equation contained in the vector equation. Numerically, we can approximate the k derivative by

$$\begin{aligned}\frac{\partial f(t;\vec{A})}{\partial k} &= \frac{\partial f(t;x_0,v_0,m,k,x_{\text{eq}},\ldots)}{\partial k} \\ &\approx \frac{f(t;x_0,v_0,m,k+\Delta k,x_{\text{eq}},\ldots) - f(t;x_0,v_0,m,k,x_{\text{eq}},\ldots)}{\Delta k}.\end{aligned} \tag{18}$$

Note that (18) is exact in the limit that $\Delta k \to 0$; in fact, this is one way to define a derivative. In the code, we just pick some small Δk (this is determined by the parameter `fd_scale`).

`chi2` – This implements (13). In particular, the lower $\chi^2(\vec{A})$ is the better our fit is. The gradient descent algorithm's goal is to try to minimize $\chi^2(\vec{A})$. The gradient descent method doesn't strictly need to compute $\chi^2(\vec{A})$ in order to work, but by tracking $\chi^2(\vec{A})$ throughout the gradient descent process, we can see how the gradient descent method is performing. In AI terms, we can watch the model's learning (i.e., the smaller $\chi^2(\vec{A})$ is, the better the model is and the more it has 'learned').

2.3 A few notes on details and parameter choices

k – As discussed in our paper presenting this project, there is actually one less degree of freedom in the fit than might appear on first glance; the mass, spring constant, and friction coefficients are not independent. The code is capable of finding all these parameters separately, but there is an overall scaling which would remain undetermined and the computation becomes more costly. We therefore choose one of these parameters which which we fix the scaling of all the others and remove it from the search for best parameter. We choose to fix the spring constant, because our independent measurement of k is comparatively precise and we know that we are making an error in ignoring the mass of the springs.

`fd_scale` – This parameter controls the size of the difference used in computing the gradient of the fit function. The difference is a dimensionful quantity, so the current value of the parameter is used to determine the scale from the dimensionless `fd_scale`. We have found that a value of 10^{-9} works relatively well for `fd_scale`.

`rk_substeps` – This parameter controls the resolution of the Runge-Kutta method by fixing the number of steps the method takes per timestep of the data. We have found that $\Delta t_{\text{data}}/$`rk_substeps` $\approx 10^{-3}$ s works well. Note that constant friction provides a discontinuous force which ruins the convergence of the method. Thus, when including constant drag `rk_substeps` must be made much larger, possibly as large as an order of magnitude or more than when constant drag is not included.

3 Appendix: algebra for the linear fit

Carrying out the derivatives in (4) and carefully splitting apart the sums, we get

$$\begin{aligned}
0 &= \sum_{i=1}^{N} 2\left(x_i - mt_i - b\right)\left(-t_i\right) = 2\sum_{i=1}^{N}\left(mt_i^2 + bt_i - x_it_i\right) \\
&= 2\sum_{i=1}^{N} mt_i^2 + 2\sum_{i=1}^{N} bt_i - 2\sum_{i=1}^{N} x_it_i = 2m\sum_{i=1}^{N} t_i^2 + 2b\sum_{i=1}^{N} t_i - 2\sum_{i=1}^{N} x_it_i \qquad \text{(19a)} \\
0 &= \sum_{i=1}^{N} 2\left(x_i - mt_i - b\right)(-1) = 2\sum_{i=1}^{N}\left(mt_i + b - x_i\right) \\
&= 2\sum_{i=1}^{N} mt_i + 2\sum_{i=1}^{N} b - 2\sum_{i=1}^{N} x_i = 2m\sum_{i=1}^{N} t_i + 2b\sum_{i=1}^{N} 1 - 2\sum_{i=1}^{N} x_i \\
&= 2m\sum_{i=1}^{N} t_i + 2Nb - 2\sum_{i=1}^{N} x_i. \qquad \text{(19b)}
\end{aligned}$$

Stepping back for a minute, keep in mind that, despite how complicated they look, (19) are simply two linear equations in the two unknowns m and b – which just happen to have ugly coefficients. We proceed to solve (19b) for b, to get

$$b = \frac{1}{2N}\left[2\sum_{i=1}^{N} x_i - 2m\sum_{i=1}^{N} t_i\right] = \frac{1}{N}\sum_{i=1}^{N} x_i - \frac{m}{N}\sum_{i=1}^{N} t_i. \qquad (20)$$

Now plug (20) into (19a) in order to eliminate b

$$\begin{aligned}
0 &= 2m\sum_{i=1}^{N} t_i^2 + 2\left[\frac{1}{N}\sum_{i=1}^{N} x_i - m\frac{1}{N}\sum_{i=1}^{N} t_i\right]\sum_{i=1}^{N} t_i - 2\sum_{i=1}^{N} x_it_i \qquad (21) \\
&= 2m\sum_{i=1}^{N} t_i^2 + 2\left[\frac{1}{N}\sum_{i=1}^{N} x_i - m\frac{1}{N}\sum_{i=1}^{N} t_i\right]\sum_{i=1}^{N} t_i - 2\sum_{i=1}^{N} x_it_i \qquad (22) \\
&= 2m\sum_{i=1}^{N} t_i^2 + \frac{2}{N}\left[\sum_{i=1}^{N} x_i\right]\left[\sum_{i=1}^{N} t_i\right] - m\frac{2}{N}\left[\sum_{i=1}^{N} t_i\right]^2 - 2\sum_{i=1}^{N} x_it_i \qquad (23) \\
&= 2m\left(\sum_{i=1}^{N} t_i^2 - \frac{1}{N}\left[\sum_{i=1}^{N} t_i\right]^2\right) + \frac{2}{N}\left[\sum_{i=1}^{N} x_i\right]\left[\sum_{i=1}^{N} t_i\right] - 2\sum_{i=1}^{N} x_it_i. \qquad (24)
\end{aligned}$$

We are now in a position to solve for m

$$m = \frac{\frac{2}{N}\left[\sum_{i=1}^{N} x_i\right]\left[\sum_{i=1}^{N} t_i\right] - 2\sum_{i=1}^{N} x_it_i}{\frac{2}{N}\left[\sum_{i=1}^{N} t_i\right]^2 - 2\sum_{i=1}^{N} t_i^2} = \frac{\frac{1}{N}\left[\sum_{i=1}^{N} x_i\right]\left[\sum_{i=1}^{N} t_i\right] - \sum_{i=1}^{N} x_it_i}{\frac{1}{N}\left[\sum_{i=1}^{N} t_i\right]^2 - \sum_{i=1}^{N} t_i^2}, \qquad (25)$$

and – plugging (25) into (20) – also for b to obtain the final answers quoted in (5).

This answer looks very complicated, so working through an example could be helpful. We can do a linear fit of the data

$$(2\text{ s}, 6\text{ m}) \quad (3\text{ s}, 7\text{ m}) \quad (4\text{ s}, 9\text{ m}) \quad (5\text{ s}, 12\text{ m}) \quad (6\text{ s}, 13\text{ m})\,. \tag{26}$$

Note that $N = 5$. Start off by doing all the needed sums

$$\sum_{i=1}^{5} x_i = 6\text{ m} + 7\text{ m} + 9\text{ m} + 12\text{ m} + 13\text{ m} = 47\text{ m}, \tag{27a}$$

$$\sum_{i=1}^{5} t_i = 2\text{ s} + 3\text{ s} + 4\text{ s} + 5\text{ s} + 6\text{ s} = 20\text{ s}, \tag{27b}$$

$$\sum_{i=1}^{5} x_i t_i = 12\text{ m}\cdot\text{s} + 21\text{ m}\cdot\text{s} + 36\text{ m}\cdot\text{s} + 60\text{ m}\cdot\text{s} + 78\text{ m}\cdot\text{s} = 207\text{ m}\cdot\text{s}, \quad \text{and} \tag{27c}$$

$$\sum_{i=1}^{5} t_i^2 = 4\text{ s}^2 + 9\text{ s}^2 + 16\text{ s}^2 + 25\text{ s}^2 + 36\text{ s}^2 = 90\text{ s}^2. \tag{27d}$$

Plugging (27) into (5),

$$m = \frac{\frac{1}{5}\times(47\text{ m})\times(20\text{ s}) - (207\text{ m}\cdot\text{s})}{\frac{1}{5}\times(20\text{ s})^2 - (90\text{ s}^2)} = \frac{-19\text{ m}\cdot\text{s}}{-10\text{ s}^2} = 1.9\text{ m/s} \tag{28a}$$

$$b = \frac{1}{5}\times(47\text{ m}) - \frac{1.9\text{ m/s}}{5}\times(20\text{ s}) = 1.8\text{ m}. \tag{28b}$$

Note that the units work out as you would expect. Typing this data into any fitting program should confirm the numerical values.