SafeFlight

Martin Deegan

January 19, 2018

Contents

1 Introduction **1**

2 Extended Kalman Filter **2**

2.1 How does the Extended Kalman Filter work? . . . 2

2.1.1 Prediction . . . 2

2.1.2 Update . . . 4

2.2 Quaternion Error State . . . 5

2.3 Predict Step . . . 6

2.4 Update Accelerometer and Magnetometer . . . 6

2.5 Update GPS . . . 7

2.6 Update Barometer . . . 7

3 Quadcopter Control **7**

A Jacobians and Taylor Series **7**

A.1 Taylor Series . . . 7

A.2 Jacobians . . . 8

A.3 Multivariate Taylor Series . . . 8

B Covariances **9**

C Quaternions Briefly **10**

C.1 Rotations in the Complex Plane . . . 10

C.2 Quaternions . . . 11

C.2.1 Conjugate . . . 11

C.2.2 Quaternion Product . . . 11

C.2.3 Rotating a Vector . . . 12

C.3 Why Quaternions? . . . 12

1 Introduction

SafeFlight is a quadcopter flight controller written entirely in Rust. It runs on a Raspberry Pi 3 Model B and features everything from low level sensor readings and management up to high level state estimation and flight control.

SafeFlight started in the Summer of 2017 as a personal project and continued as a side project during school. This is my first dabble into robotics and increased my interest in robotics profoundly.

This document is a contains all of the mathematics and concepts that have gone into SafeFlight's subsystems and will be updated as more is added. Hopefully, this will act as a guide to some people looking to work on their own robotics projects and will simplify many of the confusing topics presented on the internet.

2 Extended Kalman Filter

SafeFlight uses an Extended Kalman Filter for attitude estimation and position estimation. There is a 20-dimensional state that measures position (North, East, Down), velocity (NED), orientation, gyroscope bias, accelerometer bias, magnetic field, and sea level pressure. We denote the state as $\mathbf{x}$ and the error state as $\delta\mathbf{x}$.

$$\mathbf{x} = \begin{bmatrix} p \\ v \\ q \\ b_g \\ b_a \\ m \\ b \end{bmatrix} \qquad \delta\mathbf{x} = \begin{bmatrix} \delta p \\ \delta v \\ \delta\theta \\ \delta b_g \\ \delta b_a \\ \delta m \\ \delta b \end{bmatrix}$$

If the absolute true position was $\mathbf{x}_t$, then $\mathbf{x}_t = \mathbf{x} \oplus \delta\mathbf{x}$ where $\oplus$ represents adding the error state. This addition is not always just vector addition. In the case of quaternions $q \oplus \delta\theta = \begin{bmatrix} \frac{1}{2}\delta\theta \\ 1 \end{bmatrix} \otimes q$. This is explained a bit more in section 2.2.

Much of the derivation of the prediction step in Kalman Filter comes from the work of Joan Sola in this paper.

2.1 How does the Extended Kalman Filter work?

The EKF is an amazing tool for combining any number of sensor readings into one big estimation system. It works by tracking both your predicted state, $\mathbf{x}$, and your uncertainty, $\mathbf{P}$, which is represented as a covariance matrix (how much each value varies with respect to each other). By tracking your uncertainty, you can accuratly update your state when you receive an absolute measurement (that is usually noisy). Here I will explain the two parts of a Kalman Filter.

2.1.1 Prediction

Often you will have high frequency measurements from your array of sensors that will be quite accurate. Examples are a gyroscope or an accelerometer. These measurements will be enough to predict your state for a few seconds, but will drift away from the true value over time.

Some notation quickly: $\mathbf{x}$ is the state (position, velocity, etc...) and $\mathbf{u}$ is a control variable (gyroscope readings, accelerometer readings, etc...). This control variable contains high frequency measurements. We can define a function (usually nonlinear) to represent how our state changes now that we have new readings. In the quadcopter case, this function is definitely nonlinear due to the rotation.

$$f(\mathbf{x}_{k-1}, \mathbf{u}) = \hat{\mathbf{x}}_k$$

Where $\hat{\mathbf{x}}_k$ is the predicted state. In a prediction step you usually integrate something. For example you can integrate angular velocity to get orientation or you can integrate acceleration to get velocity. Example:

$$\mathbf{v}_k = \mathbf{v}_{k-1} + \mathbf{a}_k \Delta t$$

How do we track uncertainty? Just as each part of the state has an transition function attached to it, the error state also follows the same transition function. Example:

$$\delta \mathbf{v}_k = \delta \mathbf{v}_{k-1} + \sigma_a \Delta t$$

Where $\sigma_{\mathbf{a}}$ is the variance (noise) of the accelerometer. There is one slight difference, we don't know the error state, we only know how uncertain we are. If we knew the error state, we could simply subtract it and we would know exactly where we are! So instead, we can transition our uncertainty.

This is where the EKF gets its "Extended" name. We need to transition our covariance matrix $\mathbf{P}$ somehow in the same way we transition $\mathbf{x}_{k-1}$ to $\hat{\mathbf{x}}_k$ with f. In a normal Kalman Filter f is represented as a linear transformation (a matrix), however here it is a nonlinear function. To do this we need to linearize f somehow. If you remember Taylor expansions form calculus, you have the single variable case (Appendix A):

$$f(x_k) = f(x_{k-1}) + f'(x_{k-1})(x-a) + \frac{f''(x_{k-1})}{2!}(x-a)^2 + \frac{f'''(x_{k-1})}{3!}(x-a)^3 + \dots$$

This is very similar to our transition function. In fact, we can represent the change in our state as a tiny little step in the direction of the derivative. This is simply a first order multivariable Taylor series where our change is just the change in time:

$$f(\mathbf{x}_{k-1}, \mathbf{u}) = \mathbf{x}_{k-1} \oplus \delta_{\mathbf{predicted}} \approx \mathbf{x}_{k-1} \oplus \frac{\partial f(\mathbf{x}_{k-1})}{\partial \mathbf{x}} \Delta t$$

Where $\oplus$ represents a small addition to the state and $\delta_{\mathbf{predicted}}$ is that small change amount. In our case, we have a multivariable function so the derivative is a jacobian in which case we can use the multivariable Taylor Series. Here is an example of a first order multivariate Taylor Series (Appendix A):

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

$$\begin{aligned} f(\mathbf{x}) = f(\mathbf{x}_0) &+ \frac{\partial f_1(\mathbf{x}_0)}{\partial x_1}(x_1 - a_1) + \dots + \frac{\partial f_1(\mathbf{x}_0)}{\partial x_n}(x_n - a_n) + \dots \\ &+ \frac{\partial f_m(\mathbf{x_0})}{\partial x_1}(x_1 - a_1) + \dots + \frac{\partial f_m(\mathbf{x}_0)}{\partial x_n}(x_n - a_n) \\ &= f(\mathbf{x}_0) + \frac{\partial f}{\partial \mathbf{x}}(\mathbf{x} - \mathbf{a}) = f(\mathbf{x}_0) + \mathbf{J}(f)(\mathbf{x} - \mathbf{a}) \end{aligned}$$

So we get $\mathbf{F} = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\mathbf{x}_{k-1}}$ is the jacobian of f evaluated at $\mathbf{x}_{k-1}$. $\mathbf{F}$ is the linearized version of f and we can now update the covariance matrix by using the following identity (Appendix B):

$$\mathbf{P}_k = \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^T$$

This idenetity basically says "if you act on these variables ($\mathbf{x}$) with this function $\mathbf{F}$, then their covariances will update in this way." This only transfers existing uncertainties. At each step we add more uncertainty with each reading. We can represent these uncertainties with a diagonal matrix $\mathbf{Q}$ where each component is the variance (noise) corresponding to each of the values in the control vector $\mathbf{u}$. We now need to know how to update x_p based on these new noises so we need to take another jacobian. $\mathbf{L} = \frac{\partial f}{\partial \mathbf{u}}\Big|_{\mathbf{x}_{k-1}}$. Then we can use similar reasoning as above to get:

$$\mathbf{P_k} = \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^T + \mathbf{L}\mathbf{Q}\mathbf{L}^T$$

This is our completed prediction step.

2.1.2 Update

The update step works a little differently, but has many of the same concepts as above. As I said above, the prediction measurements tend to drift over time so we need some way to keep them in check. Update measurements are usually slow and noisy, but give us an absolute reading with respect to the state. For example: our prediction could use the accelerometer to track position over time, but will eventually have too much error to be worth using. Adding a GPS would correct our position errors and reassure us of where we are.

Note that there is no way to reduce uncertainty with the prediction step so the update step is crucial to have any sense of where you are. If we have multiple "update" sensors, we will run an update step for each one individually.

Suppose we have a sensor that gives us some reading in the form of a vector

$$\mathbf{z}_m = \mathbf{z}_t + \mathbf{w}$$

Whatever this sensor is measuring, we know it is an absolute measurement meaning it is near the correct value of whatever it is measuring, $\mathbf{z}_t$, with some extra noise, $\mathbf{w}$.

We need a way to "expose" our error state. Remember, we don't know it, we just know how uncertain we are. So we create some function (usually nonlinear) that takes our state (predicted) and outputs a measurement in the exact same format as the sensor measurement.

$$h(\hat{\mathbf{x}}_k) = \mathbf{z}_p$$

Note that if we let $\mathbf{x}_t$ be the absolute true state, then

$$\mathbf{z}_m = h(\mathbf{x}_t) + \mathbf{w}$$

So now we have a way of exposing our error. First we take the residual $\mathbf{z} = \mathbf{z}_m - \mathbf{z}_p$. $\mathbf{z}$ represents the error in our state. We just need a way to convert it into the form $\delta\mathbf{x}$ so we can remove that error from our predicted state $\hat{\mathbf{x}}_k$ to get the optimal state $\mathbf{x}_k$. Just like in the prediction step we updated the uncertainty by using the transition function on it, now we do something very similar, we convert this residual into the error state using the same function we used to create it, h. First, we need to linearize it though.

$$\mathbf{H} = \frac{\partial h}{\partial \mathbf{x}}\bigg|_{\hat{\mathbf{x}}_k}$$

Also define $\mathbf{R}$ as the variance (noise) of the update sensor in a diagonal matrix. We can measure how much the noises affect the measurement too:

$$\mathbf{M} = \frac{\partial h}{\partial \mathbf{z}}\bigg|_{\hat{\mathbf{x}}_k}$$

So in total, the uncertainty of the whole state after the update can be put as

$$\mathbf{S} = \mathbf{H}\mathbf{P}_k\mathbf{H}^T + \mathbf{M}\mathbf{R}\mathbf{M}^T$$

So we have the uncertainty from the prediction step $\mathbf{P}$ and the uncertainty from the update step $\mathbf{S}$. We use the uncertainty to decide how much to trust the prediction or how much to trust the update measurement. We do this by computing the Kalman gain.

$$\mathbf{K} = \mathbf{P}_k\mathbf{H}^T\mathbf{S}^{-1}$$

Then we compute the error state.

$$\delta\mathbf{x}_k = \mathbf{K}\mathbf{z}$$

Now we have "exposed" the error in our system and we can remove that error.

$$\mathbf{x}_k = \hat{\mathbf{x}}_k \oplus \delta\mathbf{x}_k$$

And we reset the uncertainty after we updated (became more certain). Here the $\oplus$ represents some way of adding the error state into our state. Most of the time it will be addition, but for example, quaternions do not simply add together.

Finally, we reset the covariance matrix after decreasing our uncertainty.

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P}_k(\mathbf{I} - \mathbf{K}\mathbf{H})^T + \mathbf{K}\mathbf{R}\mathbf{K}^T$$

A bit more about the different matrices. $\mathbf{H}$ and $\mathbf{K}$ are very closely related. $\mathbf{H}$ serves as a linearized transition function betweem the error state to the measurement vector. $\mathbf{K}$ is the opposite, it takes an measurement error $\mathbf{z}$ and converts it into an error state vector. Since these functions are defined by jacobians, the error state propagates properly even though measurement error may have a vastly different dimension than the state error.

$$\mathbf{K} \in \mathbb{R}^{m\times n} \quad \mathbf{H} \in \mathbb{R}^{n\times m}$$

The errors:

$$\delta\mathbf{x} \in \mathbb{R}^{m} \quad \mathbf{z} \in \mathbb{R}^{n}$$

2.2 Quaternion Error State

SafeFlight represents the orientation as a quaternion. This means we have 4 variables to represent 3 degrees of freedom. As a result, the error state can reach singularities by using a 4 vector as a quaternion error.

Instead we use $\delta\theta$ defined in the following way.

$$\mathbf{q} = \begin{bmatrix} \sin\theta\ \mathbf{v} \\ \cos\theta \end{bmatrix} = \begin{bmatrix} \frac{1}{2}\delta\theta \\ 1 \end{bmatrix}$$

Since $\delta\mathbf{q}$ is small, $\cos\theta \approx 1$ and therefore, we can use this approximation. Because of this, our error state is smaller than our main state. This messes up our jacobians because now we have a covariance in $\mathbb{R}^{19\times 19}$, but we are taking jacobians with respect to our 20 dimensional state vector. To fix this we need to take our jacobians with respect to our error state. Example:

$$\mathbf{F} = \frac{\partial f}{\partial \delta\mathbf{x}} = \frac{\partial f}{\partial \mathbf{x}}\frac{\partial \mathbf{x}}{\partial \delta\mathbf{x}}$$

Using the chain rule, we can get jacobians with respect to our error state.

$$\mathbf{D} = \frac{\partial \mathbf{x}}{\partial \delta \mathbf{x}} = J(\mathbf{x} \oplus \delta \mathbf{x})$$

However you define your error state addition, you need its jacobian to update your error state properly. So for all jacobians defined in this document, you will need to multiply by $\frac{\partial \mathbf{x}}{\partial \delta \mathbf{x}}$ to get it in the correct form (only if using quaternions).

Using JPL styled quaternions, you will get this jacobian.

$$\frac{\partial \mathbf{x} \oplus \delta \mathbf{x}}{\partial \delta \mathbf{x}} = \begin{bmatrix} \frac{\partial \mathbf{p}+\delta \mathbf{p}}{\partial \delta \mathbf{p}} & & & & & & \\ & \frac{\partial \mathbf{v}+\delta \mathbf{p}}{\partial \delta \mathbf{v}} & & & & & \\ & & \frac{\partial \mathbf{q} \otimes \delta \mathbf{q}}{\partial \delta \theta} & & & & \\ & & & \frac{\partial \mathbf{b_g}+\delta \mathbf{b_g}}{\partial \delta \mathbf{b_g}} & & & \\ & & & & \frac{\partial \mathbf{b_a}+\delta \mathbf{b_a}}{\partial \delta \mathbf{b_a}} & & \\ & & & & & \frac{\partial \mathbf{m}+\delta \mathbf{m}}{\partial \delta \mathbf{m}} & \\ & & & & & & \frac{\partial \mathbf{b}+\delta \mathbf{b}}{\partial \delta \mathbf{b}} \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{I}_6 & 0 & 0 \\ 0 & \frac{\partial \mathbf{q} \otimes \delta \mathbf{q}}{\partial \delta \theta} & 0 \\ 0 & 0 & \mathbf{I}_{10} \end{bmatrix}$$

$$\frac{\partial \mathbf{q} \otimes \delta \mathbf{q}}{\partial \delta \theta} = \frac{1}{2} \begin{bmatrix} q_w & -q_z & q_y \\ q_z & q_w & -q_x \\ -q_y & q_x & q_w \\ -q_x & -q_y & -q_z \end{bmatrix}$$

Joan Sola derives the same matrix for Hamiltonian quaternions in section 6.1.

2.3 Predict Step

Once again, I'll refer you to this paper to learn about quaternions and see the derivation of the prediction step.

We define the state transition function as follows:

$$f(\mathbf{x}_{k-1}, \mathbf{u}) = \hat{\mathbf{x}}_k$$

2.4 Update Accelerometer and Magnetometer

Correcting orientation with the accelerometer and magnetometer are very similar. In both cases, we have some known field (we are also estimating the magnetic field, so we don't actually know it) we are tracking $\hat{\mathbf{F}} = \begin{bmatrix} f_x & f_y & f_z \end{bmatrix}$ In the case of the accelerometer, it is gravity and for the magnetometer we are tracking the Earth's magnetic field.

We get a reading from these sensors in a 3 axis configuration and can do some preprocessing on them such as removing the thrust vector from the acceleration or compensating for biases.

$$\mathbf{z}_m = \begin{bmatrix} z_x \\ z_y \\ z_z \end{bmatrix} - \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}$$

The magnetometer might have more advanced bias removal such as ellipsoid fitting. Next we define the transition function from our state to the sensor readings.

$$h : \mathbf{x} \to \mathbf{z}$$

$$h(\mathbf{x}_k) = \mathbf{z}_p = \mathbf{R}\{\mathbf{q}\}\hat{\mathbf{F}}$$

Here $\mathbf{R}$ is the rotation matrix formed by $\mathbf{q}$ and we are simply rotating the field to where we would expect it to be. So $\mathbf{z}_p$ is our expected reading from our sensor. Next we compute the residual:

$$\mathbf{z} = \mathbf{z}_m - \mathbf{z}_p$$

Now we need $\mathbf{H}$ and $\mathbf{M}$ to compute the Kalman gain.

$$\mathbf{H}_x = \frac{\partial h}{\partial \mathbf{x}}$$

$$= \begin{bmatrix} \Large 0_{3\times 6} & \begin{matrix} 2(f_y q_y + q_z f_z) & 2(f_y q_x - 2 f_x q_y + q_w f_z) & 2(q_x f_z - 2 f_x q_z - f_y q_w) & 2(q_y f_z - f_y q_z) \\ 2(f_x q_y - 2 f_y q_x - q_w f_z) & 2(f_x q_x + q_z f_z) & 2(f_x q_w - 2 f_y q_z + q_y f_z) & 2(f_x q_z - q_x f_z) \\ 2(f_y q_w + f_x q_z - 2 q_x f_z) & 2(f_y q_z - f_x q_w - 2 q_y f_z & 2(f_x q_x + f_y q_y & 2(f_y q_x - f_x q_y) \end{matrix} & \Large 0_{3\times 10} \end{bmatrix}$$

$$\mathbf{H} = \mathbf{H}_x \mathbf{D}$$

$$\mathbf{M}_x = \frac{\partial h}{\partial \mathbf{z}} = \begin{bmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_w q_z) & 2(q_w q_y + q_x q_z) \\ 2(q_w q_z - q_x q_y) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z + q_x q_w) \\ 2(q_x q_z - q_w q_y) & 2(q_w q_x + q_y q_z) & 1 - 2(q_x^2 + q_y^2) \end{bmatrix}$$

$$\mathbf{M} = \mathbf{M}_x \mathbf{D}$$

Now run the update as described above.

2.5 Update GPS

2.6 Update Barometer

3 Quadcopter Control

3.1 Torque Computation

3.2 Motor Power Computation

A Jacobians and Taylor Series

This section is a quick recap on calculus.

A.1 Taylor Series

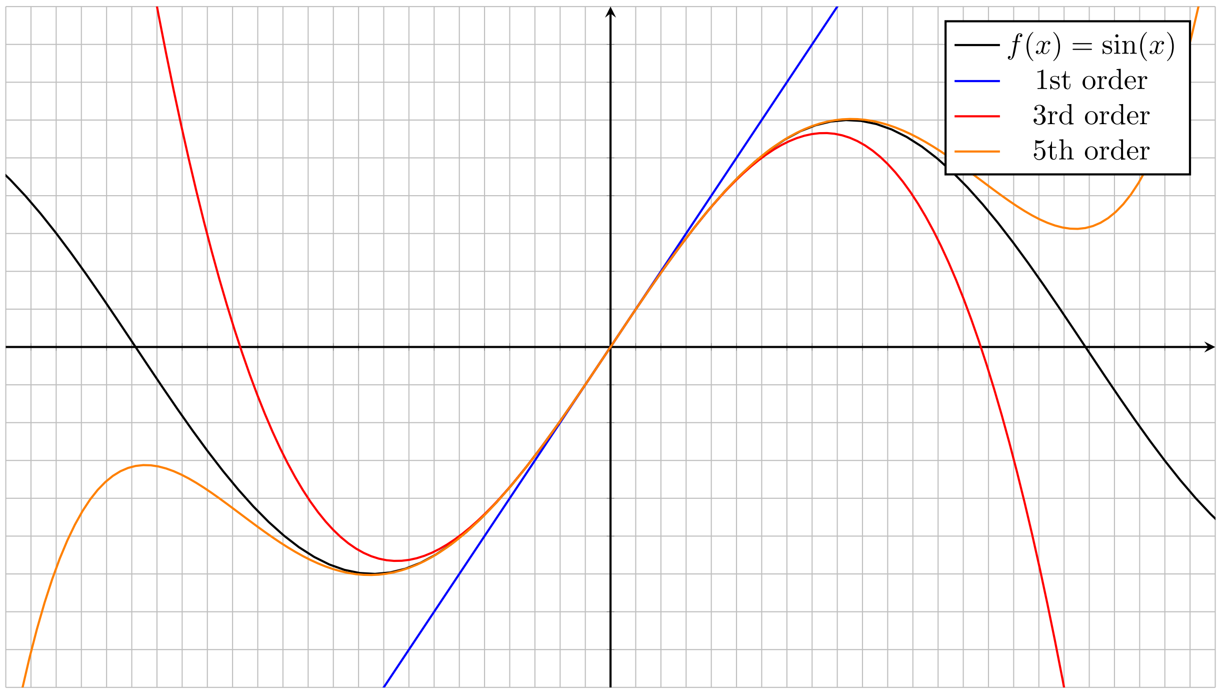
As you learned in Calc II, a Taylor Series is a way of representing a function f around a point, a, by representing it as a summation (with some other requirements).

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k$$

We can use it to approximate f if we don't know it explicitly. If we take an n^{th} order Taylor expansion of f, we get

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{f''(a)}{n!}(x-a)^2 + \cdots + \frac{f^{(n)}}{n!}(x-a)^n$$

Here is $\sin(x)$ approximated to several orders.



You can see that the higher order expansions approximate better, but still diverge fairly quickly.

A.2 Jacobians

A Jacobian is a matrix representing the partial derivatives of a function.

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

$$f : \mathbf{x} \mapsto \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_m(\mathbf{x}) \end{bmatrix}$$

So we define the Jacobian in the following way:

$$J(f) = \frac{\partial f}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

A.3 Multivariate Taylor Series

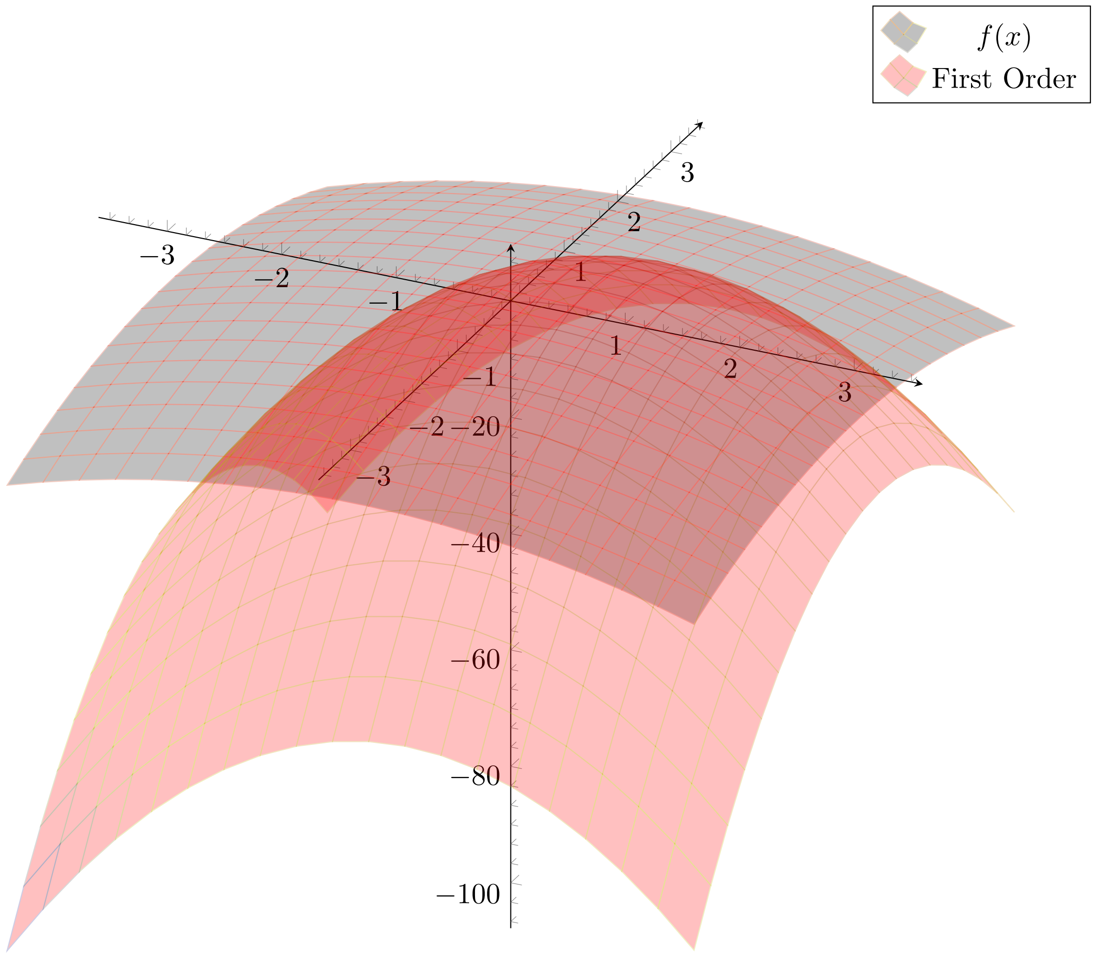
Now we can define the multivariate Taylor Series.

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

If we want to approximate f about $\mathbf{x}_0 = \begin{bmatrix} x_0 & x_1 & \dots & x_n \end{bmatrix}^T$ then we can do so by

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + J(f)\big|_{\mathbf{x}_0} (\mathbf{x} - \mathbf{x}_0)$$

Let $f(x, y) = x^2 y$ and we want to approximate this at $(1, 1)$.



B Covariances

For random variables X, Y their covariance, $\text{Cov}(X, Y)$, represents their joint variability. $\text{Cov}(X, X) = \text{Var}(X)$. So, if we have several random variables $X_1, \dots, X_n$, we can represent their covariances in a matrix.

$$\Sigma = \begin{bmatrix} \Sigma_{X_1 X_1} & \dots & \Sigma_{X_1 X_n} \\ \vdots & \ddots & \vdots \\ \Sigma_{X_n X_1} & \dots & \Sigma_{X_n X_n} \end{bmatrix}$$

Σ_{XY} =Cov(X, Y). This matrix is symmetric since Cov(X, Y) =Cov(Y, X).

Let

$$\mathbf{X} = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix}$$

and $\Sigma_{\mathbf{x}}$ be the covariance matrix of $\mathbf{X}$. Suppose we have some linear transformation $\mathbf{F} \in \mathbb{R}^{m \times n}$ and let $\mathbf{X'} = \mathbf{FX}$. Then we have

$$\Sigma_{\mathbf{x'}} = \mathbf{F}\Sigma_{\mathbf{x}}\mathbf{F}^T$$

Proof.

$$\mathbf{M} = E[\mathbf{X}]$$

$$\Sigma_{\mathbf{X'}} = \text{cov}(\mathbf{FX}) = E[(\mathbf{FX} - \mathbf{FM})(\mathbf{FX} - \mathbf{FM})^T]$$

$$= E[\mathbf{F}(\mathbf{X} - \mathbf{M})(\mathbf{X} - \mathbf{M})^T\mathbf{F}^T]$$

We can extract $\mathbf{F}$ and its transpose because they are not a function of $\mathbf{X}$x

$$= \mathbf{F}E[(\mathbf{X} - \mathbf{M})(\mathbf{X} - \mathbf{M})^T]\mathbf{F}^T$$

$$= \mathbf{F}\,\text{cov}(\mathbf{X})\mathbf{F}^T = \mathbf{F}\Sigma_{\mathbf{X}}\mathbf{F}^T$$

□

C Quaternions Briefly

Quaternions are 4-dimensional numbers which are an extension of the complex numbers. They are often used in robotics because they can represent a 3D rotation about an axis and have several nice properties which make them superior to other ways of representing rotations.

C.1 Rotations in the Complex Plane

Any number $z \in \mathbb{C}$ can be represented in the following way:

$$z = Re^{i\theta}$$

If we expand this out using a Taylor series for e^x:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

$$Re^{i\theta} = R(1 + i\theta - \theta^2 - i\theta^3 + \theta^4 + \dots)$$

$$= R((1 - \theta^2 + \theta^4 - \dots) + (i\theta - i\theta^3 + i\theta^5 - \dots))$$

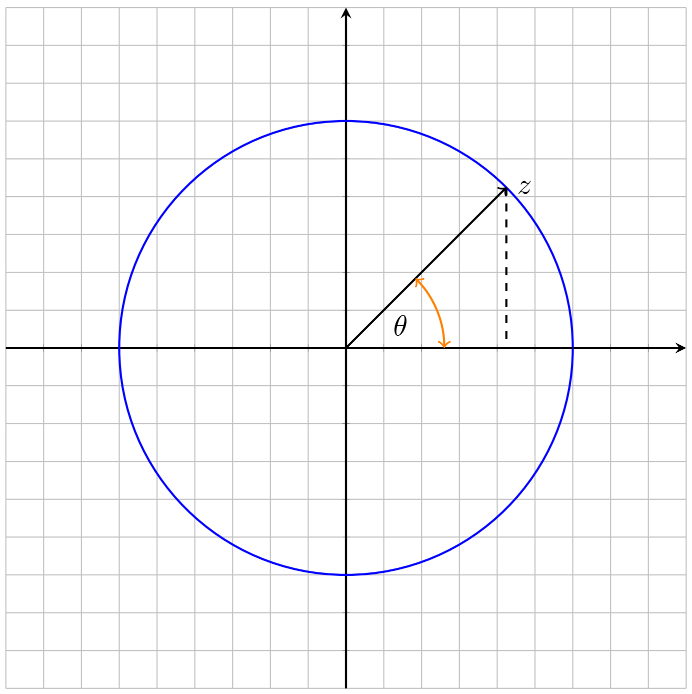
Here are the Taylor series for sin and cos:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots$$

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

Notice anything?

$$z = R((1 - \theta^2 + \theta^4 - \dots) + i(\theta - \theta^3 + \theta^5 - \dots)) = R\cos\theta + Ri\sin\theta$$



C.2 Quaternions

Quaternions are very similar to above. If we have some axis $\mathbf{u}$ we want to rotate about by angle θ, let $\mathbf{v} = \theta\mathbf{u}$ be the scaled axis representation of this rotation.

$$q = e^{\theta\mathbf{u}}$$

$$= 1 - \mathbf{u}\theta - \frac{\theta^2}{2!} + \frac{\mathbf{u}\theta^3}{3!} + \cdots = \cos\theta + \mathbf{u}\sin\theta = \begin{bmatrix}\mathbf{u}\sin\theta \\ \cos\theta\end{bmatrix} = \begin{bmatrix}q_x \\ q_y \\ q_z \\ q_w\end{bmatrix}$$

C.2.1 Conjugate

$$q^* = \begin{bmatrix}-\mathbf{u}\sin\theta \\ \cos\theta\end{bmatrix}$$

C.2.2 Quaternion Product

Now we can define the quaternion product:

$$\mathbf{q} \otimes \mathbf{p} = [\mathbf{q}]_l\mathbf{p} = [\mathbf{p}]_r\mathbf{q}$$

$$[\mathbf{q}]_l = \begin{bmatrix} q_w & q_z & -q_y & q_x \\ -q_z & q_w & q_x & q_y \\ q_y & -q_x & q_w & q_z \\ -q_x & -q_y & -q_z & q_z \end{bmatrix}$$

$$[\mathbf{q}]_r = \begin{bmatrix} q_w & -q_z & q_y & q_x \\ q_z & q_w & -q_x & q_y \\ -q_y & q_x & q_w & q_z \\ -q_x & -q_y & -q_z & q_z \end{bmatrix}$$

C.2.3 Rotating a Vector

We can easily rotate a vector $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ with quaternion $\mathbf{q}$. $\mathbf{q}$ must be a unit vector in order to rotations to be defined.

$$\mathbf{x}' = \mathbf{q} \otimes \mathbf{x} \otimes \mathbf{q}^* = \mathbf{q} \otimes \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 0 \end{bmatrix} \otimes \mathbf{q}^* = [\mathbf{q}]_l [\mathbf{q}^*]_r \mathbf{x} = \begin{bmatrix} \mathbf{R} & 0 \\ 0 & 1 \end{bmatrix} \mathbf{x}$$

where $\mathbf{R}$ is a rotation matrix.

It is easiest to construct the rotation matrix as follows:

$$\mathbf{R} = \begin{bmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_z q_w) & 2(q_x q_z + q_y q_w) \\ 2(q_x q_y + q_z q_w) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z - q_x q_w) \\ 2(q_x q_z - q_y q_w) & 2(q_y q_z + q_x q_w) & 1 - 2(q_x^2 - q_y^2) \end{bmatrix}$$

and then $\mathbf{x}' = \mathbf{R}\mathbf{x}$.

C.3 Why Quaternions?

You might ask yourself why we should use quaternions over the multitude of other rotation representations? We can list a few of the types here:

1. Euler angles (roll, pitch, yaw)
2. Scaled axis, like $\theta \mathbf{u}$ that we constructed our quaternion with
3. Rotation matrix

There are several reasons to use quaternions over these formats and I will list them here:

(a) They are free from gimbal lock which Euler angles are perceptable to. This is because quaternions represent a rotation about an axis.

(b) The axis angle representation allows us to easily interpolate between orientations or between vectors. Euler angles are noncommutative compositions of rotations about three axes which make it very difficult to interpolate.

(c) They only use four variables as opposed to the nine that would be required for a rotation matrix.

(d) Unlike the scaled axis representation, we aren't required to do expensive trigonometric computations to construct the rotation matrix.