Autonomous Flight in Unmanned Airplanes

B. Pence
Brigham Young University - Idaho

Rev. April 12, 2023

Table of Contents

Chapter 1

North-East-Down Airplane Equations

Contents

1.1 Equations for North-East-Down (NED) Coordinates

Because of the importance of the North-East-Down (NED) coordinate system in aerospace applications, this chapter provides a summary of the aircraft equations in the NED inertial reference frame. It also provides the parameters for the different airplane models. In the NED inertial frame, the inertial x-axis points towards the north, the y-axis points to the east, and the z-axis points down towards the center of the earth. The body-fixed coordinate system is also different. The body-fixed x-axis points out the nose of the aircraft, the y-axis points along the wing to the right, and the z-axis points down through the bottom of the aircraft as shown in Figure 1.1.

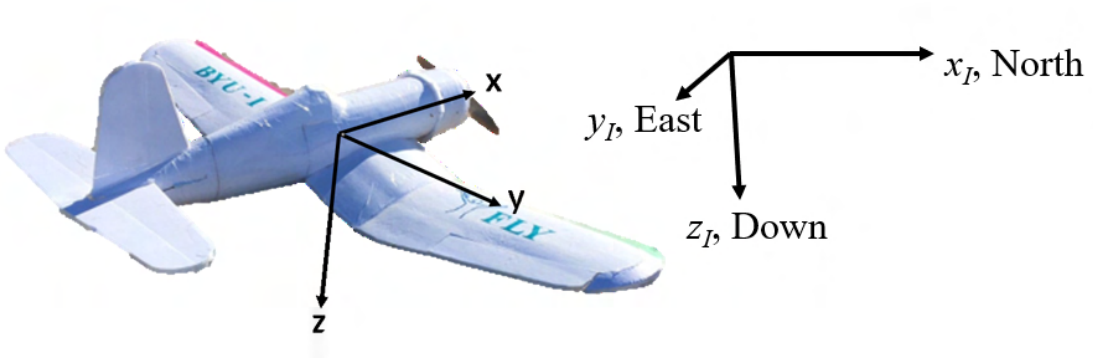


Figure 1.1 The traditional body-fixed coordinate system in the NED inertial frame.

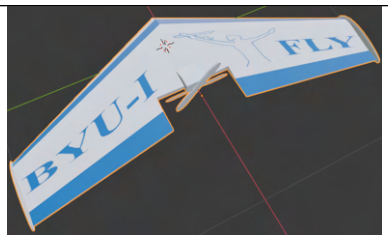
1.1.1 Airplane Parameters for NED

Table 1.1 lists general constant parameters, and Table 1.2 lists constant parameters for the BYU-I Wing and FT Corsair airplanes in the NED coordinate frame.

Table 1.1 General Parameters for North-East-Down (NED) Coordinates

Parameter	Value	Description
ρ	1.2682 kg/m^3	Air density
g	9.81 m/s^2	Gravitational Acceleration

Table 1.2 Constant Parameters for Airplane Simulations

		
	BYU-I Wing	**FT Corsair**
m	0.9 kg	1 kg
J_{xx}	0.115 kg m^2	0.12 kg m^2
J_{yy}	0.16 kg m^2	0.2 kg m^2
J_{zz}	0.17 kg m^2	0.18 kg m^2
J_{xy}	0 kg m^2	0 kg m^2
J_{xz}	0.0015 kg m^2	0.015 kg m^2
J_{yz}	0 kg m^2	0 kg m^2
S	1.42 m	1.5 m
c	0.33 m	0.3 m
A	0.47 m^2	0.45 m^2
$C_{L,0}$	-0.9	-0.9
$C_{L,1}$	2.5	2.5
$C_{L,2}$	1	1
$C_{L,3}$	11	11
$C_{L,4}$	0.004	0.004
C_{L,δ_e}	0.1	0.1
$C_{D,0}$	0.06	0.06
$C_{D,2}$	0.44	0.44
C_{D,δ_e}	0.001	0.001
$C_{y,\beta}$	0.01	0.015
C_{y,δ_r}	0	0.001
$C_{Ty,\alpha}$	0.15	0.1
C_{Ty,δ_e}	0.07	0.1
b_q	0.4 N m s	0.5 N m s
C_{δ_a}	0.02	0.03
b_p	1 N m s	1 N m s
$C_{Tz,\beta}$	0.005	0.0002
C_{Tz,δ_r}	0	0.04

Continued on next page

Table 1.2 – *Continued from previous page*

	BYU-I Wing	FT Corsair
b_r	0.5 N m s	0.5 N m s
$n_{\max}$	170 rev/s	170 rev/s
D	0.2286 m	0.254 m
α_b	5 inch	5 inch

1.1.2 Fixed-Wing Aircraft Equations for NED Coordinate System

This section lists the equations for the fixed-wing aircraft motion in the North-East-Down (NED) coordinate frame. The equations for the North-Up-East coordinate system were derived in earlier sections.

Airspeed, Angle of Attack, and Side-slip Angle

$$\begin{bmatrix} u_r \\ v_r \\ w_r \end{bmatrix} = \begin{bmatrix} u - u_w \\ v - v_w \\ w - w_w \end{bmatrix}$$

$$V_a = \sqrt{u_r^2 + v_r^2 + w_r^2}$$

$$\alpha = \tan^{-1}\left(\frac{w_r}{V_a}\right)$$

$$\beta = \sin^{-1}\left(\frac{v_r}{V_a}\right)$$

Wind Model

$$\dot{u}_{w_g} = -\frac{V_a}{L_u} u_{w_g} + \sigma_u \sqrt{\frac{2V_a}{\pi L_u}} \mathcal{N}_1$$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\left(\frac{V_a}{L_v}\right)^2 & -2\frac{V_a}{L_v} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \sigma_v \sqrt{\frac{3V_a}{\pi L_v}} \end{bmatrix} \mathcal{N}_2$$

$$v_{w_g} = \begin{bmatrix} \frac{V_a}{\sqrt{3}L_v} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\begin{bmatrix} \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\left(\frac{V_a}{L_w}\right)^2 & -2\frac{V_a}{L_w} \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ \sigma_w \sqrt{\frac{3V_a}{\pi L_w}} \end{bmatrix} \mathcal{N}_3$$

$$w_{w_g} = \begin{bmatrix} \frac{V_a}{\sqrt{3}L_w} & 1 \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}$$

Gust Parameters	$L_u = L_v$ (m)	L_w (m)	$\sigma_u = \sigma_v$ (m/s)	σ_w (m/s)
≈ 50 m altitude, light	200	50	1.06	0.7
≈ 50 m altitude, moderate	200	50	2.12	1.4
≈ 600 m altitude, light	533	533	1.5	1.5
≈ 600 m altitude, moderate	533	533	3.0	3.0

$$\begin{bmatrix} u_w \\ v_w \\ w_w \end{bmatrix} = R_I^b \begin{bmatrix} w_{xI} \\ w_{yI} \\ w_{zI} \end{bmatrix} + \begin{bmatrix} u_{w_g} \\ v_{w_g} \\ w_{w_g} \end{bmatrix}$$

where R_I^b is the rotation matrix from the inertial to the body frame:

$$R_I^b = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0 e_3 + e_1 e_2) & 2(e_1 e_3 - e_0 e_2) \\ 2(e_1 e_2 - e_0 e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0 e_1 + e_2 e_3) \\ 2(e_0 e_2 + e_1 e_3) & 2(e_2 e_3 - e_0 e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix}$$

Gravitational Forces

$$\begin{bmatrix} f_{x,g} \\ f_{y,g} \\ f_{z,g} \end{bmatrix} = mg \begin{bmatrix} 2(e_1 e_3 - e_0 e_2) \\ 2(e_0 e_1 + e_2 e_3) \\ e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \tag{1.1}$$

Propeller Torque and Thrust

$$n = n_{\max}\left(1 - (1 - \delta_t)^2\right)$$

Using the value of the relative speed u_r in the advance ratio $J = \frac{u_r}{nD}$, calculate the torque coefficient.

$$C_Q = z_Q C_{Q,L} + (1 - z_Q) C_{Q,u}$$

where

$$z_Q = \frac{1}{1 + e^{10(0.16 + 0.05\alpha_b - J)}}$$

$$C_{Q,L} = 0.0121 - 0.017\,(J - 0.1\,(\alpha_b - 5))$$

and

$$C_{Q,u} = 0.0057 + 0.00125\,(\alpha_b - 5) - 0.0005J$$

Calculate the propeller torque.

$$T_{x,Q} = \rho n^2 D^5 C_Q$$

Calculate the thrust coefficient C_T.

$$C_T = z C_{T,L} + (1 - z) C_{T,u}$$

where

$$z = \frac{1}{1 + e^{20(0.14+0.018\alpha_b - 0.6J)}}$$

$$C_{T,L} = 0.11 - 0.18\left(J - 0.105\left(\alpha_b - 5\right)\right)$$

and

$$C_{T,u} = 0.1 + 0.009\left(\alpha_b - 5\right) - 0.065J$$

and α_b has units of inches. Finally, calculate the propeller thrust.

$$f_{x,T} = \rho n^2 D^4 C_T$$

Forces and Torques from Aerodynamics

The aerodynamic force is

$$f_{\text{aero}} = \frac{1}{2}\rho V_a^2 A$$

where A is the wing area (m^2), and ρ (kg/m^3) is the air density.

The lift force in the body-fixed z-direction is

$$f_{z,L} = f_{\text{aero}}\left(-\left(C_{L,0} + \frac{C_{L,1}}{C_{L,2} + e^{-C_{L,3}(\alpha + C_{L,4})}}\right) - C_{L,\delta_e}\delta_e\right)$$

The drag force in the body-fixed x-direction is

$$f_{x,D} = f_{\text{aero}}\left(-\left(C_{D,0} + C_{D,2}\alpha^2\right) - C_{D,\delta_e}\delta_e\right)$$

The side-slip force in the y-direction is

$$f_{y,\beta} = f_{\text{aero}}\left(-C_{y,\beta}\beta - C_{y,\delta_r}\delta_r\right)$$

The pitching torque about the y-axis is

$$T_y = f_{\text{aero}}\, c\left(-C_{Ty,\alpha}\alpha - C_{Ty,\delta_e}\delta_e\right) - b_r r$$

where c is the wing chord. The roll torque about the x-axis is

$$T_{x,\text{aero}} = f_{\text{aero}} S C_{\delta_a}\delta_a - b_p p$$

where S is the wing span. The yaw torque about the z-axis is

$$T_z = f_{\text{aero}}\, c\left(C_{Tz,\beta}\beta + C_{Tz,\delta_r}\delta_r\right) - b_q q$$

Combined Forces and Torques in the Body-Fixed Directions

$$f_x = f_{x,T} + f_{x,D} + f_{x,g}$$
$$f_y = f_{y,\beta} + f_{y,g}$$
$$f_z = f_{z,L} + f_{z,g}$$
$$T_x = T_{x,\text{aero}} + T_{x,Q}$$

Equations of motion for an aircraft:

$$\begin{bmatrix}\dot{u}\\ \dot{v}\\ \dot{w}\end{bmatrix} = \begin{bmatrix}rv-qw\\ pw-ru\\ qu-pv\end{bmatrix} + \frac{1}{m}\begin{bmatrix}f_x\\ f_y\\ f_z\end{bmatrix}$$

$$\begin{bmatrix}\dot{p}\\ \dot{q}\\ \dot{r}\end{bmatrix} = J^{-1}\left(\begin{bmatrix}qr\left(J_{yy}-J_{zz}\right)+pqJ_{xz}\\ pr\left(J_{zz}-J_{xx}\right)+\left(r^2-p^2\right)J_{xz}\\ pq\left(J_{xx}-J_{yy}\right)-qrJ_{xz}\end{bmatrix} + \begin{bmatrix}T_x\\ T_y\\ T_z\end{bmatrix}\right)$$

$$\begin{bmatrix}\dot{x}_I\\ \dot{y}_I\\ \dot{z}_I\end{bmatrix} = \begin{bmatrix}e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_1e_2-e_0e_3) & 2(e_0e_2+e_1e_3)\\ 2(e_0e_3+e_1e_2) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_2e_3-e_0e_1)\\ 2(e_1e_3-e_0e_2) & 2(e_0e_1+e_2e_3) & e_0^2-e_1^2-e_2^2+e_3^2\end{bmatrix}\begin{bmatrix}u\\ v\\ w\end{bmatrix}$$

$$\begin{bmatrix}\dot{e}_0\\ \dot{e}_1\\ \dot{e}_2\\ \dot{e}_3\end{bmatrix} = \frac{1}{2}\begin{bmatrix}-e_1 & -e_2 & -e_3\\ e_0 & -e_3 & e_2\\ e_3 & e_0 & -e_1\\ -e_2 & e_1 & e_0\end{bmatrix}\begin{bmatrix}p\\ q\\ r\end{bmatrix}$$

where

$$J^{-1} = \frac{1}{J_{xx}J_{yy}J_{zz}-J_{yy}J_{xz}^2}\begin{bmatrix}J_{yy}J_{zz} & 0 & J_{yy}J_{xz}\\ 0 & J_{xx}J_{zz}-J_{xz}^2 & 0\\ J_{yy}J_{xz} & 0 & J_{xx}J_{yy}\end{bmatrix}$$

and each member of the quaternion e_i, $i = 0,1,2,3$ is normalized:

$$e_i = \frac{e_i}{\sqrt{e_0^2+e_1^2+e_2^2+e_3^2}}$$

1.2 Derivation of the Airplane Equations of Motion

The remainder of this chapter derives the equations presented in Section 1.1.2.

1.2.1 Quaternion Rotations

A 3D rotation about any unit vector $\hat{e} = \left\langle e_x\hat{i} \quad e_y\hat{j} \quad e_z\hat{k}\right\rangle$ pointing in any direction can be described by quaternions. When describing rotations, a quaternion consists of an angle ρ and a unit vector $\hat{e}$. A quaternion q can be described by four parts, e_0, e_1, e_2, and e_3 as follows:

$$q = \cos\frac{\rho}{2} + \sin\frac{\rho}{2}\left\langle e_x\hat{i} \quad e_y\hat{j} \quad e_z\hat{k}\right\rangle \tag{1.2}$$

$$= e_0 + \left\langle e_1\hat{i} \quad e_2\hat{j} \quad e_3\hat{k}\right\rangle \tag{1.3}$$

where $e_0 = \cos\frac{\rho}{2}$, $e_1 = \sin\frac{\rho}{2}e_x$, $e_2 = \sin\frac{\rho}{2}e_y$, and $e_3 = \sin\frac{\rho}{2}e_z$. A quaternion rotation is illustrated in Figure 1.2. The body-fixed point p is rotated ρ radians about the unit axis $\hat{e}$.

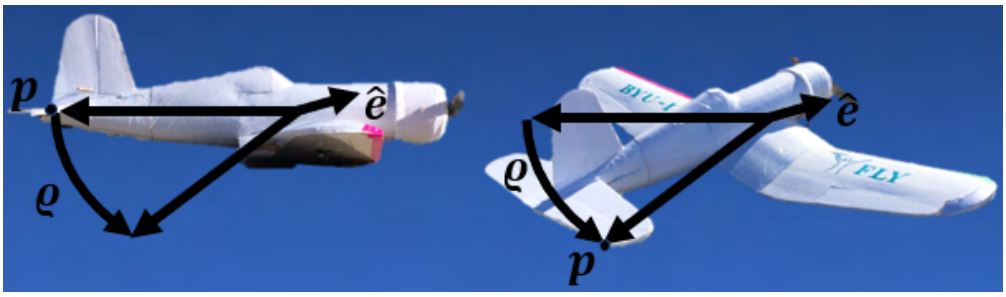


Figure 1.2 A rotation of ρ radians about the unit axis $\hat{e}$

Using quaternions, a rotation matrix can calculate the body-fixed x, y, and z coordinates in the inertial frame:

$$\begin{bmatrix} p_{x,I} \\ p_{y,I} \\ p_{z,I} \end{bmatrix} = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_1e_2-e_0e_3) & 2(e_0e_2+e_1e_3) \\ 2(e_0e_3+e_1e_2) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_2e_3-e_0e_1) \\ 2(e_1e_3-e_0e_2) & 2(e_0e_1+e_2e_3) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \tag{1.4}$$

Eq. (1.4) describes the locations of the body-fixed x, y, and z positions (p_x, p_y, p_z) with coordinates $(p_{x,I}, p_{y,I}, p_{z,I})$ in the inertial frame.

1.2.2 Properties of Orthonormal Rotation Matrices

The rotation matrix in Eq. (1.4) is orthonormal. Normalized quaternions must satisfy the condition

$$e_i = \frac{e_i}{\sqrt{e_0^2+e_1^2+e_2^2+e_3^2}} \tag{1.5}$$

for $i = 0, 1, 2, 3$.

Orthonormal rotation matrices have some useful properties. First, the inverse of an orthonormal rotation matrix $\mathscr{R}$ is equal to its transpose:

$$\mathscr{R}^{-1} = \mathscr{R}^T \tag{1.6}$$

where the superscript ^{T} indicates the matrix transpose. As a result, a point $(p_{x,I}, p_{y,I}, p_{z,I})$ in the inertial coordinate frame has the following body-fixed coordinates:

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_0e_3+e_1e_2) & 2(e_1e_3-e_0e_2) \\ 2(e_1e_2-e_0e_3) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_0e_1+e_2e_3) \\ 2(e_0e_2+e_1e_3) & 2(e_2e_3-e_0e_1) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \begin{bmatrix} p_{x,I} \\ p_{y,I} \\ p_{z,I} \end{bmatrix} \tag{1.7}$$

Another useful property is that the determinant of an orthonormal rotation matrix is one.

$$\det(\mathscr{R}) = 1$$

Orthonormal quaternion rotations are used extensively in the kinematic equations for an aircraft.

1.2.3 Angular Velocity and Quaternions

Angular velocity ω (rad/s) of a rigid body in three-dimensional space describes the rate of rotation of the rigid body. It is represented by an array of three body-fixed components:

$$\omega = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{1.8}$$

where p (rad/s) is the instantaneous roll rotation rate about the x-axis, q is the instantaneous pitch rotation rate about the y-axis, and r is the instantaneous yaw rotation about the z-axis.

Graf[1] does an excellent job explaining quaternion math. He shows the derivation to relate the time-derivative of a unit quaternion to the angular velocities.

$$\begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{1.9}$$

1.2.4 Coordinate Frame Translations

After applying rotations, we can translate the aircraft to a new position relative to the origin of the inertial frame. To do so, we add the inertial frame displacements to each point on the aircraft. For example, if the center of gravity of the aircraft is translated 5 m north of the origin of the inertial frame, we must also translate every point on the aircraft 5 m north after first applying any rotations.

1.3 Equations of Motion

This section derives the equations of motion for the aircraft. These equations are used to determine the position and orientation of the aircraft. First we will derive the equations of motion for any rigid body in 3D space. Then we will use the symmetry of aircraft to simplify the equations.

1.3.1 General 6 DOF Motion of a Rigid Body

Consider any rigid body having six degrees of freedom (DOF). The first three DOF are translations in the x, y, and z orthogonal directions. The velocity with respect to a body-fixed coordinate frame is $V = \begin{bmatrix} u & v & w \end{bmatrix}^T$, where u, v, and w are the linear velocities in the x, y, and z body-fixed directions respectively. The next three DOF are rotational: first rotation ϕ_x around the body-fixed x-axis, then rotation ψ_y around the body-fixed y-axis, and finally rotation θ_z around the body fixed z-axis. The subscripts on the angle symbols indicate the axes of rotation. The angular velocities in the body-fixed frame are $\omega = \begin{bmatrix} p & q & r \end{bmatrix}^T$, where p, q, and r are the angular velocities about the x, y, and z axes respectively.

[1] see Basile Graf, "Quaternions and Dynamics", 18 Nov. 2008, Cornell University, Online (accessed March 2021): https://arxiv.org/pdf/0811.2889.pdf

Newton's equations of motion describe the dynamic behavior of the airplane due to applied forces and torques. These equations of motion about the center of gravity for any rotating rigid body with these six DOF can be written as follows:

$$F = m\frac{dV_T}{dt} \tag{1.10}$$

$$T = \frac{dH_T}{dt} \tag{1.11}$$

where $F = \begin{bmatrix} f_x & f_y & f_z \end{bmatrix}^T$ are the external forces and $T = \begin{bmatrix} T_x & T_y & T_z \end{bmatrix}$ are the external torques acting on the rigid body. Angular momentum H in the body-fixed reference frame is calculated as follows

$$H = J\begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{1.12}$$

where J is the body-fixed moment of inertia matrix:

$$J = \begin{bmatrix} J_{xx} & -J_{xy} & -J_{xz} \\ -J_{xy} & J_{yy} & -J_{yz} \\ -J_{xz} & -J_{yz} & J_{zz} \end{bmatrix} \tag{1.13}$$

For a rotating reference frame, the total time-derivatives $\frac{da_T}{dt}$ of a vector a_T can be calculated as the summation of linear and rotational parts. The linear part is $\frac{da}{dt}$, and the rotational part is the cross-product between the angular velocity ω and the vector a:

$$\frac{da_T}{dt} = \frac{da}{dt} + \omega \times a \tag{1.14}$$

Therefore, the total derivative of the velocity $\frac{dV_T}{dt}$ in the body frame of the rigid body becomes

$$\begin{aligned} \frac{dV_T}{dt} &= \frac{dV}{dt} + \omega \times V \\ &= \begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} + \begin{bmatrix} qw - rv \\ ru - pw \\ pv - qu \end{bmatrix} \end{aligned} \tag{1.15}$$

Likewise, the total derivative $\frac{dH_T}{dt}$ of angular momentum relative to the body-fixed coordinate system in a

rotating rigid body is calculated by

$$\begin{aligned}\frac{dH_T}{dt} &= \frac{dH}{dt} + \omega \times H \\ &= \begin{bmatrix} J_{xx} & -J_{xy} & -J_{xz} \\ -J_{xy} & J_{yy} & -J_{yz} \\ -J_{xz} & -J_{yz} & J_{zz} \end{bmatrix} \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} + \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \begin{bmatrix} J_{xx} & -J_{xy} & -J_{xz} \\ -J_{xy} & J_{yy} & -J_{yz} \\ -J_{xz} & -J_{yz} & J_{zz} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \\ &= J \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} + \begin{bmatrix} \left(r^2 - q^2\right) J_{yz} + qr\left(J_{zz} - J_{yy}\right) - pqJ_{xz} + prJ_{xy} \\ \left(p^2 - r^2\right) J_{xz} + pr\left(J_{xx} - J_{zz}\right) + pqJ_{yz} - qrJ_{xy} \\ \left(q^2 - p^2\right) J_{xy} + pq\left(J_{yy} - J_{xx}\right) - prJ_{yz} + qrJ_{xz} \end{bmatrix} \end{aligned} \tag{1.16}$$

where J is the body-fixed inertia matrix defined in Eq. (1.13). We can substitute Eq. (1.16) back into Eq. (1.11) for the rotational dynamics equation. These dynamic equations, plus some kinematic equations, will describe the equations of motion in 3D for a rigid body.

1.3.2 State-Equations for 6 DOF Rigid Body Motion

State-Equations for 6 DOF Rigid Body Motion

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} rv - qw \\ pw - ru \\ qu - pv \end{bmatrix} + \frac{1}{m} \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix}$$

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = J^{-1} \left(\begin{bmatrix} \left(q^2 - r^2\right) J_{yz} + qr\left(J_{yy} - J_{zz}\right) + pqJ_{xz} - prJ_{xy} \\ \left(r^2 - p^2\right) J_{xz} + pr\left(J_{zz} - J_{xx}\right) - pqJ_{yz} + qrJ_{xy} \\ \left(p^2 - q^2\right) J_{xy} + pq\left(J_{xx} - J_{yy}\right) + prJ_{yz} - qrJ_{xz} \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \right)$$

$$\begin{bmatrix} \dot{x}_I \\ \dot{y}_I \\ \dot{z}_I \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_1e_2 - e_0e_3) & 2(e_0e_2 + e_1e_3) \\ 2(e_0e_3 + e_1e_2) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_2e_3 - e_0e_1) \\ 2(e_1e_3 - e_0e_2) & 2(e_0e_1 + e_2e_3) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix}$$

$$\begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

The objective in deriving Newton's equations is to find a set of state-equations that describes the motion of the 6 DOF rigid body. To do so, we substitute the derivatives defined in Eqs. (1.15) and (1.16) into Newton's equations, Eqs. (1.10) and (1.11). Rearranging them results in a set of state equations for the

body fixed velocities u, v, and w and the angular velocities p, q, and r.

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} rv - qw \\ pw - ru \\ qu - pv \end{bmatrix} + \frac{1}{m} \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} \tag{1.17}$$

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = J^{-1} \left(\begin{bmatrix} \left(q^2 - r^2\right) J_{yz} + qr\left(J_{yy} - J_{zz}\right) + pqJ_{xz} - prJ_{xy} \\ \left(r^2 - p^2\right) J_{xz} + pr\left(J_{zz} - J_{xx}\right) - pqJ_{yz} + qrJ_{xy} \\ \left(p^2 - q^2\right) J_{xy} + pq\left(J_{xx} - J_{yy}\right) + prJ_{yz} - qrJ_{xz} \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \right) \tag{1.18}$$

The variables in Eqs. (1.17) and (1.18) are as follows: $\dot{u}$, $\dot{v}$, and $\dot{w}$ are the body-fixed x-axis, y-axis, and z-axis accelerations respectively; p, q, and r are the angular velocities around the body-fixed x, y, and z axes respectively; f_x, f_y, and f_z are the external forces acting on the rigid body aligned with the respective body-fixed x, y, and z axes; T_x, T_y, and T_z are the external torques with respect to the body-fixed axes; m is the mass of the rigid body; J_{xx}, J_{yy}, and J_{zz} are the mass moments of inertia about the body-fixed x, y, and z axes respectively. J_{xy}, J_{xz}, and J_{yz} are the body-fixed products of inertia; and J^{-1} is calculated as follows:

$$J^{-1} = \begin{bmatrix} J_{xx} & -J_{xy} & -J_{xz} \\ -J_{xy} & J_{yy} & -J_{yz} \\ -J_{xz} & -J_{yz} & J_{zz} \end{bmatrix}^{-1} \tag{1.19}$$

$$= \frac{1}{|J|} \begin{bmatrix} J_{yy}J_{zz} - J_{yz}^2 & J_{xy}J_{zz} + J_{xz}J_{yz} & J_{xy}J_{yz} + J_{xz}J_{yy} \\ J_{xy}J_{zz} + J_{yz}J_{xz} & J_{xx}J_{zz} - J_{xz}^2 & J_{xx}J_{yz} + J_{xz}J_{xy} \\ J_{xy}J_{yz} + J_{yy}J_{xz} & J_{xx}J_{yz} + J_{xy}J_{xz} & J_{xx}J_{yy} - J_{xy}^2 \end{bmatrix} \tag{1.20}$$

where the determinant $|J|$ of the moment of inertia matrix is

$$|J| = J_{xx}J_{yy}J_{zz} - 2J_{xy}J_{yz}J_{xz} - J_{xx}J_{yz}^2 - J_{yy}J_{xz}^2 - J_{zz}J_{xy}^2 \tag{1.21}$$

In addition to calculating velocities, it may be desirable to know the displacement and rotational orientation of the rigid body with respect to the inertial frame. When the angles, ϕ_x, ψ_y, and θ_z are zero, we assume that the body-fixed frame has the same orientation as the inertial frame. The x, y, and z axes of both frames are in the same directions.

A 3D rotation about any unit vector $\hat{e} = \left\langle e_x\hat{i} \quad e_y\hat{j} \quad e_z\hat{k} \right\rangle$ pointing in any direction can be described by quaternions. When describing rotations, a quaternion consists of an angle ρ and a unit vector $\hat{e}$. A quaternion q can be described by four parts, e_0, e_1, e_2, and e_3 as follows:

$$q = \cos\frac{\rho}{2} + \sin\frac{\rho}{2} \left\langle e_x\hat{i} \quad e_y\hat{j} \quad e_z\hat{k} \right\rangle \tag{1.22}$$

$$= e_0 + \left\langle e_1\hat{i} \quad e_2\hat{j} \quad e_3\hat{k} \right\rangle \tag{1.23}$$

where $e_0 = \cos\frac{\rho}{2}$, $e_1 = \sin\frac{\rho}{2}e_x$, $e_2 = \sin\frac{\rho}{2}e_y$, and $e_3 = \sin\frac{\rho}{2}e_z$. Using quaternions, a rotation from the inertial frame orientation to a different orientation is calculated using a quaternion rotation matrix. The

rotation matrix helps determine how the body-fixed x, y, and z coordinates are mapped to the inertial frame:

$$\begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_1e_2-e_0e_3) & 2(e_0e_2+e_1e_3) \\ 2(e_0e_3+e_1e_2) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_2e_3-e_0e_1) \\ 2(e_1e_3-e_0e_2) & 2(e_0e_1+e_2e_3) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{1.24}$$

Eq. (1.24) describes the x, y, and z positions in the body-frame with respect to the inertial frame x_I, y_I, and z_I. The body-fixed velocities u, v, and w at the center of gravity can also be described in the inertial frame using the quaternion rotation matrix:

$$\begin{bmatrix} \dot{x}_I \\ \dot{y}_I \\ \dot{z}_I \end{bmatrix} = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_1e_2-e_0e_3) & 2(e_0e_2+e_1e_3) \\ 2(e_0e_3+e_1e_2) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_2e_3-e_0e_1) \\ 2(e_1e_3-e_0e_2) & 2(e_0e_1+e_2e_3) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{1.25}$$

This kinematic state-equation, Eq. (1.25), can be solved to find the position of the center of gravity of a rigid body in the inertial frame.

The time-derivative of a unit quaternion can be calculated as a function of the angular velocities p, q, and r as follows[2]:

$$\begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 & -p & -q & -r \\ p & 0 & r & -q \\ q & -r & 0 & p \\ r & q & -p & 0 \end{bmatrix} \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{bmatrix} \tag{1.26}$$

or

$$\begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{1.27}$$

The complete set of state-equations for 6 DOF rigid body motion combines Eqs. (1.17), (1.18), (1.25), and (1.27). These equations were summarized at the beginning of the section.

1.3.3 Equations of Motion for an Aircraft

Traditionally, when modeling aircraft, the body-fixed coordinate frame is oriented as follows (see Figure 1.3): the x-axis extends from the center of gravity out through the nose at the front of the plane; the y-axis extends from the center of gravity out through the right wing; the z-axis extends from the center of gravity down through the bottom of the fuselage.

In most airplanes, there is symmetry about the x-z plane. Therefore $J_{xy} = J_{yz} = 0$, and the equations of motion given in Section 1.3.2 can be simplified.

[2] see Basile Graf, "Quaternions and Dynamics", 18 Nov. 2008, Cornell University, Online (accessed March 2021): https://arxiv.org/pdf/0811.2889.pdf

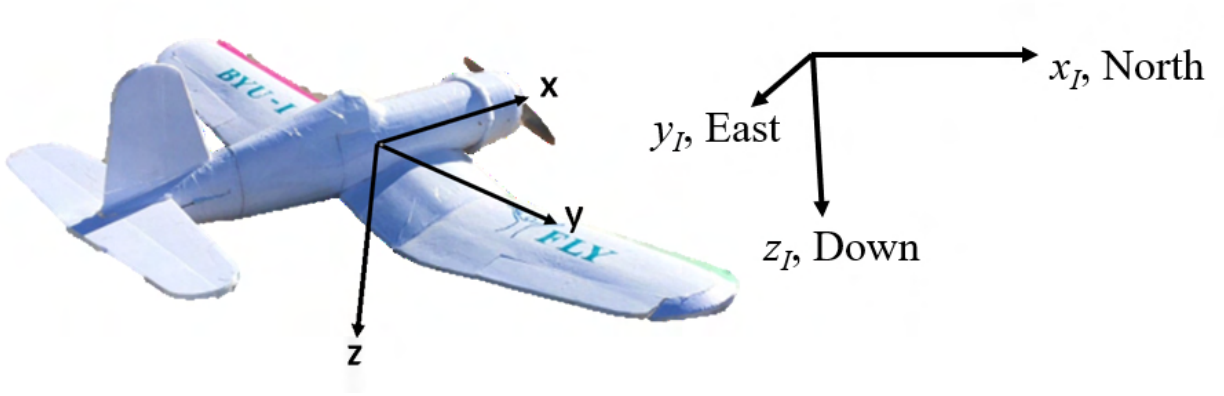


Figure 1.3 Left: Traditional body-fixed coordinate system.

Equations of motion for an aircraft:

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} rv - qw \\ pw - ru \\ qu - pv \end{bmatrix} + \frac{1}{m} \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} \tag{1.28}$$

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = J^{-1} \left(\begin{bmatrix} qr(J_{yy} - J_{zz}) + pqJ_{xz} \\ pr(J_{zz} - J_{xx}) + (r^2 - p^2)J_{xz} \\ pq(J_{xx} - J_{yy}) - qrJ_{xz} \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \right) \tag{1.29}$$

$$\begin{bmatrix} \dot{x}_I \\ \dot{y}_I \\ \dot{z}_I \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_1e_2 - e_0e_3) & 2(e_0e_2 + e_1e_3) \\ 2(e_0e_3 + e_1e_2) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_2e_3 - e_0e_1) \\ 2(e_1e_3 - e_0e_2) & 2(e_0e_1 + e_2e_3) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{1.30}$$

$$\begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{1.31}$$

where

$$J^{-1} = \frac{1}{J_{xx}J_{yy}J_{zz} - J_{yy}J_{xz}^2} \begin{bmatrix} J_{yy}J_{zz} & 0 & J_{yy}J_{xz} \\ 0 & J_{xx}J_{zz} - J_{xz}^2 & 0 \\ J_{yy}J_{xz} & 0 & J_{xx}J_{yy} \end{bmatrix} \tag{1.32}$$

⋆ These equations require that the quaternion is a unit quaternion. This can be accomplished by normalization at each iteration in the simulation after solving Eq. (1.31). Each member of the quaternion e_i, $i = 0,1,2,3$ is normalized as follows:

$$e_i = \frac{e_i}{\sqrt{e_0^2 + e_1^2 + e_2^2 + e_3^2}} \tag{1.33}$$

1.4 Airspeed, Angle of Attack, and Side-slip Angle

This section summarizes the equations that are used to calculate the forces and torques on the airplane. These forces and torques become the inputs to the equations of motion that describe the airplane dynamics and kinematics (see Eq. (1.28) - Eq. (1.33)). The force and torque equations depend on the airspeed V_a (m/s) of the airplane. The airspeed vector V_a^b in the body frame is affected by the body-frame wind speeds u_w , v_w, and w_w (m/s) and aircraft speeds u, v, and w (m/s) (see Section 1.5) as follows:

$$V_a^b = \begin{bmatrix} u_r \\ v_r \\ w_r \end{bmatrix} = \begin{bmatrix} u - u_w \\ v - v_w \\ w - w_w \end{bmatrix} \tag{1.34}$$

The airspeed V_a is then the magnitude of the airspeed vector V_a^b:

$$V_a = \sqrt{u_r^2 + v_r^2 + w_r^2} \tag{1.35}$$

The angle of attack α (rad) is another frequently used variable. It is a function of the relative air speeds w_r and V_a (m/s) and is calculated as follows:

$$\alpha = \tan^{-1}\left(\frac{w_r}{V_a}\right) \tag{1.36}$$

Finally, the side-slip angle β (rad) is another critical variable. It is dependent on the relative airspeed v_r (m/s) in the body y-axis and the airspeed V_a (m/s).

$$\beta = \sin^{-1}\left(\frac{v_r}{V_a}\right) \tag{1.37}$$

1.5 Wind Model

The wind model consists of two different parts: (1) constant wind speeds and (2) gusts. The constant wind speeds in the inertial coordinate frame are w_{xI}, w_{yI}, and w_{zI} (m/s) in the x, y, and z directions respectively.

The wind gusts are generated using random Gaussian noise processes $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_3$, each of which are zero mean with a standard deviation of one. These become inputs in the Dryden Gust Model to calculate the wind gusts. The wind gust speed in the body-fixed x-axis u_{w_g} (m/s) is calculated by solving the differential equation:

$$\dot{u}_{w_g} = -\frac{V_a}{L_u} u_{w_g} + \sigma_u \sqrt{\frac{2V_a}{\pi L_u}} \mathcal{N}_1 \tag{1.38}$$

The gust speed in the body-fixed y-axis v_{w_g} (m/s) is calculated by solving

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\left(\frac{V_a}{L_v}\right)^2 & -2\frac{V_a}{L_v} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \sigma_v \sqrt{\frac{3V_a}{\pi L_v}} \end{bmatrix} \mathcal{N}_2 \tag{1.39}$$

$$v_{w_g} = \begin{bmatrix} \frac{V_a}{\sqrt{3}L_v} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{1.40}$$

The gust speed in the body-fixed z-axis w_{w_g} (m/s) is calculated by solving

$$\begin{bmatrix} \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\left(\frac{V_a}{L_w}\right)^2 & -2\frac{V_a}{L_w} \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ \sigma_w \sqrt{\frac{3V_a}{\pi L_w}} \end{bmatrix} \mathcal{N}_3 \tag{1.41}$$

$$w_{w_g} = \begin{bmatrix} \frac{V_a}{\sqrt{3}L_w} & 1 \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} \tag{1.42}$$

The Dryden Gust Model parameters are given in Table 1.3. The initial conditions for the state variables x_1, x_2, x_3, and x_4 are all zero.

Table 1.3 Parameters for the Dryden Gust Model

Gust Parameters	$L_u = L_v$ (m)	L_w (m)	$\sigma_u = \sigma_v$ (m/s)	σ_w (m/s)
≈ 50 m altitude, light	200	50	1.06	0.7
≈ 50 m altitude, moderate	200	50	2.12	1.4
≈ 600 m altitude, light	533	533	1.5	1.5
≈ 600 m altitude, moderate	533	533	3.0	3.0

To get the overall wind speeds u_w, v_w, and w_w (m/s) in the body-fixed x, y, and z axes respectively, the constant wind speeds w_{xI}, w_{yI}, and w_{zI} (m/s) are converted to the body-frame and added to the gust speeds:

$$\begin{bmatrix} u_w \\ v_w \\ w_w \end{bmatrix} = R_I^b \begin{bmatrix} w_{xI} \\ w_{yI} \\ w_{zI} \end{bmatrix} + \begin{bmatrix} u_{w_g} \\ v_{w_g} \\ w_{w_g} \end{bmatrix} \tag{1.43}$$

where R_I^b is the rotation matrix from the inertial-frame to the body-frame:

$$R_I^b = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0 e_3 + e_1 e_2) & 2(e_1 e_3 - e_0 e_2) \\ 2(e_1 e_2 - e_0 e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0 e_1 + e_2 e_3) \\ 2(e_0 e_2 + e_1 e_3) & 2(e_2 e_3 - e_0 e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \tag{1.44}$$

The quaternion variables e_0, e_1, e_2, e_3 are calculated using the equations of motion derived in Section 1.3.3.

1.6 Input Commands

A four-channel remote controller with two joysticks can command four input signals: δ_e, δ_a, δ_t, and δ_r (see Figure 1.4). The command δ_e (-1 to +1) controls the pitching rate of the aircraft. A value of $\delta_e = +1$ causes the aircraft to nose down. A value of $\delta_e = -1$ causes the aircraft to pitch upwards. In traditional fixed-wing aircraft, δ_e controls the elevator. For a quadcopter, δ_e causes a difference between the speeds of the fore and aft propellers.

Figure 1.4 The input commands δ_e, δ_a, δ_t, and δ_r in their +1 positions

The command δ_a (-1 to +1) controls the rolling rate of the aircraft. A value of $\delta = +1$ causes the aircraft to roll to the right. A value of $\delta_a = -1$ causes the aircraft to roll to the the left. In traditional fixed-wing airplanes, the command δ_a controls the ailerons. In a quadcopter, δ_a causes a difference between the speeds of the left and right propellers.

The command δ_t (0 to +1) controls the propeller speed. A value of $\delta_t = +1$ commands full throttle. A value of $\delta_t = 0$ stops the propellers from spinning.

The command δ_r (-1 to +1) controls the yaw rate of an aircraft. A command of $\delta_r = +1$ causes the aircraft to yaw to the right. A command of $\delta_r = -1$ causes the aircraft to yaw to the left. In a traditional fixed-wing airplane, δ_r controls the rudder. In a quadcopter, it causes a difference in the speeds of diagonally opposite propellers, which results in a net yaw torque.

1.7 Modeling Propellers

Figure 1.5 A 0945 propeller

The goal of this section is to determine a mathematical relationship between the throttle command δ_t (0-1) from the remote controller and the propeller thrust f_T (N) and torque T_Q (N m). These relationships depend on a number of parameters: propeller diameter, blade pitch, number of blades, blade area, motor torque and speed ratios, motor size, battery voltage, battery current limits, air density, relative airspeed, and remote controller throttle curves. Though long, this list is still incomplete. However, we will use some approximations to simplify the analysis while attempting to retain the most relevant physical relationships.

1.7.1 Propeller Thrust

Aerodynamic force f_{aero} depends on the density of the fluid ρ, the fluid velocity V, the area A of the object in the fluid stream, and a lift or drag coefficient C.

$$f_{\text{aero}} = \frac{1}{2}\rho V^2 AC \tag{1.45}$$

The relative fluid velocity, area, and pitch vary along the length of the propeller, thereby causing the aerodynamic force to vary as well. We would need to integrate these forces along the length of the propeller to find the overall force. The complicated shape of the propeller makes this a difficult task.

Dimensional analysis and experimental studies[3] suggest that the overall propeller thrust force f_T (N) can be approximated by the dimensionless relationship

$$\frac{f_T}{\rho n^2 D^4} \approx C_T(J, \alpha_b) \tag{1.46}$$

where f_T (N) is the propeller thrust, D (m) is the propeller diameter, ρ (kg/m^3) is the air density, and n (rev/s) is the propeller angular speed. The thrust coefficient $C_T(J, \alpha_b)$ is dimensionless. It is usually

[3] see McCormick, "Aerodynamics, Aeronautics and Flight Mechanics", 2nd edition

determined experimentally. It is a function of the advance ratio $J = \frac{u_r}{nD}$ and the average pitch α_b (inch). The pitch α_b is defined as the ratio of the forward travel of the propeller per rotation assuming no slip. u_r (m/s) is the relative airspeed; for example, if there is no wind u_r is the x-axis forward velocity of a fixed-wing aircraft.

To understand the thrust coefficient $C_T(J, \alpha_b)$ better, consider a few different possibilities for a fixed-wing aircraft. Thrust is proportional to the change in the velocity of the air behind the propeller minus the velocity of the air in front of it: $f_T = \dot{m}(u_{\text{behind}} - u_{\text{ahead}})$. The greater the change in velocity, the greater the thrust. If the propeller speed is constant, nD is constant. As the plane flies faster, $u_r = u_{\text{ahead}}$ increases. If u_r increases but nD is constant, the thrust decreases because the change in velocity is less. Therefore as the advance ratio $J = \frac{u_r}{nD}$ increases, the thrust f_T (and C_T) decreases. If the advance ratio J decreases, the thrust f_T (and C_T) increases.

Next consider the effect of the propeller blade pitch α_b. If it is zero, the propeller produces no thrust. As α_b gets larger up to a certain point, it produces more thrust. Once the pitch is so great that the propeller is parallel to the airflow, it again produces no thrust. Above this pitch, it begins producing thrust in the opposite direction.

Barnes McCormick published a set of experimentally determined thrust coefficient C_p curves that depend on the advance ratio J and blade pitch α_b. A graph of these curves can be found on page 306 of his book, "Aerodynamics, Aeronautics and Flight Mechanics", 2nd edition.

McCormick's data was for a propeller with 3 blades. More recent curves have been produced for common two-blade model airplane propellers. Researchers at the University of Illinois Urbana-Champaign (UIUC) reproduced matching data to those collected at Ohio State University[4] (OSU). The data are for 1050 and 1070 propellers, *i.e.*, 10 inch diameter propellers with 5 inch and 7 inch pitches respectively.

In this document, we derived an equation to fit the thrust coefficient data collected by UIUC and OSU. Figure 1.6 shows the fit equations on the same graph as the data.

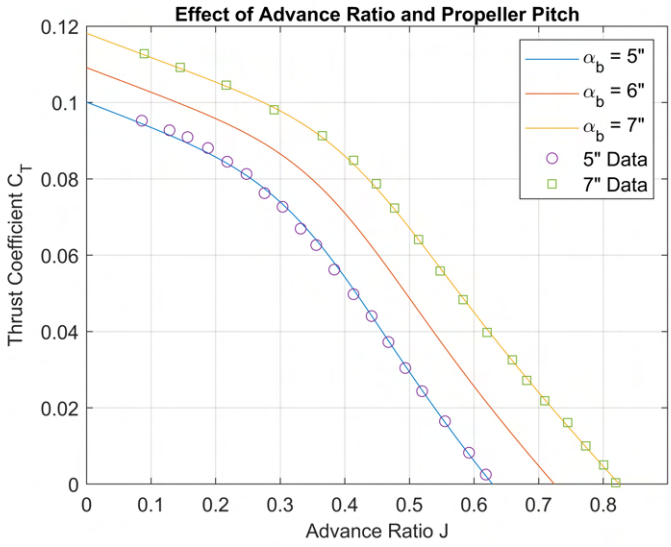


Figure 1.6 The propeller thrust coefficient C_T is affected by the pitch α_b and the advance ratio $J = \frac{u_r}{nD}$

[4] J.B. Brandt, R.W. Deters, G.K. Ananda, O.D. Dantsker, and M.S. Selig , UIUC Propeller Database, Vols 1-3, University of Illinois at Urbana-Champaign, retrieved 2 February 2021 from https://m-selig.ae.illinois.edu/props/propDB.html.

The empirical equation used to generate the curves in Figure 1.6 is

$$C_T = zC_{T,L} + (1-z)C_{T,u} \tag{1.47}$$

where

$$z = \frac{1}{1+e^{20(0.14+0.018\alpha_b-0.6J)}} \tag{1.48}$$

$$C_{T,L} = 0.11 - 0.18\,(J - 0.105\,(\alpha_b - 5)) \tag{1.49}$$

and

$$C_{T,u} = 0.1 + 0.009\,(\alpha_b - 5) - 0.065J \tag{1.50}$$

and α_b has units of inches.

Note that a single equation, Eq. (1.47), is used to fit the data for both the 1050 and 1070 propellers. The advantage of having a single equation for the thrust coefficient is that it gives us the ability to interpolate and extrapolate to fit slightly different propellers. For example, the thrust coefficient C_T for a 0945 propeller (9 inch diameter with a 4.5 inch pitch) can be approximated by using $\alpha_b = 4.5$ inches in Eq. (1.47) through Eq. (1.50).

Using Eqs. (1.46) and (1.47), the propeller thrust is

$$f_T = \rho n^2 D^4 C_T \tag{1.51}$$

1.7.2 Propeller Torque

The derivation for the propeller torque is nearly identical to the propeller thrust in the previous section. However, thrust is a force, but torque is a force multiplied by a moment arm. The dimensional analysis for torque treats the propeller diameter as the moment arm distance. Therefore, the equation for propeller torque T_Q (N m) is like the thrust equation Eq. (1.51) except it has an additional power on the diameter (D^5) and a dimensionless torque coefficient C_Q instead of a thrust coefficient C_T.

$$T_Q = \rho n^2 D^5 C_Q \tag{1.52}$$

In addition to thrust coefficient curves, the researchers at UIUC and OSU also published torque coefficient data for the 1050 and 1070 propellers. We fit empirical equations to this data. The data and fit equations are graphed in Figure 1.7.

The empirical equation used to generate the curves in Figure 1.6 is

$$C_Q = z_Q C_{Q,L} + (1-z_Q)C_{Q,u} \tag{1.53}$$

where

$$z_Q = \frac{1}{1+e^{10(0.16+0.05\alpha_b-J)}} \tag{1.54}$$

$$C_{Q,L} = 0.0121 - 0.017\,(J - 0.1\,(\alpha_b - 5)) \tag{1.55}$$

and

$$C_{Q,u} = 0.0057 + 0.00125\,(\alpha_b - 5) - 0.0005J \tag{1.56}$$

and α_b has units of inches.

Note that a single equation, Eq. (1.53), is used to fit the data for both the 1050 and 1070 propellers. The advantage of having a single equation for the torque coefficient is that it gives us the ability to interpolate

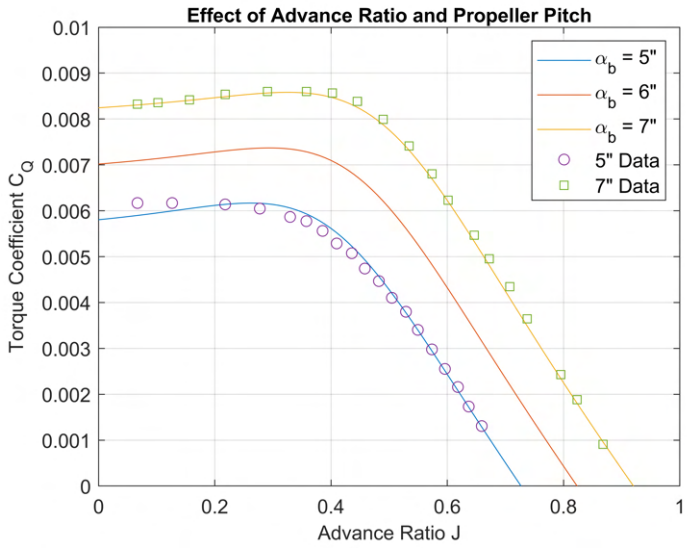


Figure 1.7 The propeller torque coefficient C_Q is affected by the pitch α_b and the advance ratio $J = \frac{u_r}{nD}$

and extrapolate to fit slightly different propellers. For example, the torque coefficient C_Q for a 0945 propeller (9 inch diameter with a 4.5 inch pitch) can be approximated by using $\alpha_b = 4.5$ in Eq. (1.53) through Eq. (1.56).

1.7.3 Motor Equations

Recall that our ultimate goal for modeling propellers is to find a relationship between the throttle command δ_t and the thrust and torque produced by the propeller. Section 1.7.1 modeled propeller thrust, and Section 1.7.2 modeled propeller torque; however, both models are incomplete without a model of the motor.

When spinning at a constant (steady) rate, brushed or brushless DC (Direct Current) motors can be modeled by the following relationship[5]:

$$T_Q = k_T \left(\frac{1}{R} (V_{\text{in}} - 2\pi k_V n) - I_0 \right) \tag{1.57}$$

where T_Q (N m) is the motor torque, k_T (Nm/A) is the motor torque constant, R (Ω) is the motor electrical resistance, V_{in} (V) is the supplied voltage, n (rev/s) is the rotational speed, and I_0 (A) is the no-load current of the motor. The values for k_T, R, and I_0 are often provided by the manufacturer. Note that when converted to the same units, the motor speed constant k_V (rad/s/V) and the motor torque constant k_T (Nm/A) have the same value.

Setting the torque from Eq. (1.57) equal to that of Eq. (1.52) results in the following equation:

$$\rho D^5 C_Q n^2 + \frac{2\pi k_T k_V}{R} n + \left(\frac{k_T}{R} V_{\text{in}} - k_T I_0 \right) = 0 \tag{1.58}$$

[5] see Chapter 4 - Forces and Moments of chap4.pdf PDF Slides by R. Beard and T. McLain https://uavbook.byu.edu/doku.php, downloaded 2 February 2021

which at first glance appears to be a quadratic function of the rotational propeller speed n. However, since C_Q is a function of the advance ratio $J = \frac{u}{nD}$, Eq. (1.58) is a complicated function of speed n. We would like to solve it for n as a function of the supplied voltage V_{in}. However, because of the complicated nature of C_Q, we cannot simply apply the quadratic formula to get an exact result in a single step.

Fortunately, however, Eq. (1.58) is numerically convergent, meaning that we can use an iterative guess-and-check method to calculate the value of n. First, we start by guessing a value of n and use it to find C_Q. Then we use the quadratic formula to solve Eq. (1.58) for n. If the calculated value matches our initial guess, we are done; otherwise, we use the new calculated value of n as our new guess to get C_Q and repeat the process. This iterative method is completed once the guessed value of n matches the value calculated by the quadratic equation to within an acceptable tolerance. The guess-and-check method is computationally expensive; therefore, Section 1.7.4 suggests alternative approaches.

The final relationship required for our motor model is to relate the supply voltage V_{in} and the throttle command δ_t. For this, we use a simple linear relationship:

$$V_{\text{in}} = V_{\max}\delta_t \tag{1.59}$$

Since δ_t varies from 0 to 1, we use the maximum supply voltage $V_{\max}$ as the proportionality constant.

1.7.4 Computational Simplifications for Propeller Speed

The complicated nature of Eq. (1.58) can cause the calculation of propeller speed n to be computationally expensive. To improve computational speed, this section proposes a numerical simplification: Eq. (1.58) is solved offline and only once to find a maximum speed $n_{\max}$ (or $n_{\max}$ is determined experimentally). The offline calculations use the maximum voltage $V_{\max}$ at zero relative velocity $u_r = 0$. The resulting propeller speed is called $n_{\max}$. Then the relationship between the throttle command δ_t (0-1) and the propeller speed n (rev/s) is approximated with a quadratic function.

$$n = n_{\max}\left(1 - (1-\delta_t)^2\right) \tag{1.60}$$

1.7.5 Summary of Propeller Equations

How to calculate f_T and T_Q:
The calculation of the propeller thrust force f_T and torque T_Q consists of two steps. The first step is performed offline and only once. The second step is performed at each iteration in the simulation.

Offline calculation solved only once:
(a) Determine $n_{\max}$ experimentally

OR

(b) Using the value $u_r = 0$ in the advance ratio $J = \frac{u_r}{nD}$ and $V_{in} = V_{\max}$, calculate the propeller speed n using the equations

$$C_Q = z_Q C_{Q,L} + (1 - z_Q) C_{Q,u}$$

where

$$z_Q = \frac{1}{1 + e^{10(0.16 + 0.05\alpha_b - J)}}$$

$$C_{Q,L} = 0.0121 - 0.017\,(J - 0.1\,(\alpha_b - 5))$$

and

$$C_{Q,u} = 0.0057 + 0.00125\,(\alpha_b - 5) - 0.0005J$$

Use a root-finding algorithm to determine the propeller speed n (rev/s).

$$\rho D^5 C_Q n^2 + \frac{2\pi k_T k_V}{R} n + \left(\frac{k_T}{R} V_{\text{in}} - k_T I_0\right) = 0$$

Call the resulting propeller speed $n_{\max}$.

Online calculations solved at each iteration of the simulation
Use the following relationship to find the propeller speed n:

$$n = n_{\max}\left(1 - (1 - \delta_t)^2\right)$$

Using the value of the relative speed u_r in the advance ratio $J = \frac{u_r}{nD}$, calculate the torque coefficient.

$$C_Q = z_Q C_{Q,L} + (1 - z_Q) C_{Q,u}$$

where

$$z_Q = \frac{1}{1 + e^{10(0.16 + 0.05\alpha_b - J)}}$$

$$C_{Q,L} = 0.0121 - 0.017\,(J - 0.1\,(\alpha_b - 5))$$

and

$$C_{Q,u} = 0.0057 + 0.00125\,(\alpha_b - 5) - 0.0005J$$

Calculate the propeller torque.

$$T_{x,Q} = \rho n^2 D^5 C_Q$$

Calculate the thrust coefficient C_T.

$$C_T = z C_{T,L} + (1 - z) C_{T,u}$$

where

$$z = \frac{1}{1 + e^{20(0.14 + 0.018\alpha_b - 0.6J)}}$$

$$C_{T,L} = 0.11 - 0.18\,(J - 0.105\,(\alpha_b - 5))$$

and

$$C_{T,u} = 0.1 + 0.009\,(\alpha_b - 5) - 0.065J$$

and α_b has units of inches. Finally, calculate the propeller thrust.

$$f_{x,T} = \rho n^2 D^4 C_T$$

1.8 Gravitational Forces

Gravitational forces act to pull the airplane down towards the center of the earth. Gravity exerts a force in the positive z-axis of the inertial frame. Because the plane may have a different orientation than the inertial frame, gravity results in components $f_{x,g}$, $f_{y,g}$, and $f_{z,g}$ (N) in potentially all three body-fixed directions. We use the quaternion rotation matrix R_I^b (see Eq. (1.44)) to convert from the inertial frame to the body-fixed frame:

$$\begin{bmatrix} f_{x,g} \\ f_{y,g} \\ f_{z,g} \end{bmatrix} = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_0e_3+e_1e_2) & 2(e_1e_3-e_0e_2) \\ 2(e_1e_2-e_0e_3) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_0e_1+e_2e_3) \\ 2(e_0e_2+e_1e_3) & 2(e_2e_3-e_0e_1) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} \tag{1.61}$$

These forces are functions of the airplane mass m (kg) and the gravitational acceleration $g = 9.81$ m/s^2. The equations depend on the quaternions e_0, e_1, e_2, and e_3. They are summarized below.

Gravitational Forces

$$\begin{bmatrix} f_{x,g} \\ f_{y,g} \\ f_{z,g} \end{bmatrix} = mg \begin{bmatrix} 2(e_1e_3-e_0e_2) \\ 2(e_0e_1+e_2e_3) \\ e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \tag{1.62}$$

1.9 Aerodynamics of Fixed-Wing Aircraft

Aerodynamic forces f_{aero} in general depend on the density of the fluid ρ, the fluid velocity V, the area A of the object in the fluid stream, and a lift or drag coefficient C.

$$f_{\text{aero}} = \frac{1}{2}\rho V^2 AC \tag{1.63}$$

Since torque result from forces multiplied by a moment arm, aerodynamic torques T_{aero} are calculated as

$$T_{\text{aero}} = \frac{1}{2}\rho V^2 ALC \tag{1.64}$$

where L is the length of the moment arm.

1.9.1 Lift, Drag, and Side-slip Forces

Aerodynamics cause lift f_z and drag f_x forces that act on the aircraft. The airspeed V_a (m/s) (see Eq. (1.35)) is the velocity V in the calculation of the aerodynamic force Eq. (1.63). The lift, drag, and side-slip coefficients in the calculations depend on a number of variables.

The independent variables in the drag and lift coefficients are the elevator command δ_e (-1 to 1) and the angle of attack α (rad) (see Eq. (1.36)). In this document, the lift C_L and drag coefficients are determined using equations that were loosely fit to data from an experimental airfoil (NACA 4412) in Figure 1.8. The fit equation for the lift does not completely account for the loss in lift at stall angles of

attack. Experimental data was not available for angles of attack exceeding 20 degrees. The offset between the CD NACA 4412 data and the CD Fit data is to account for the extra drag of the body of the airplane.

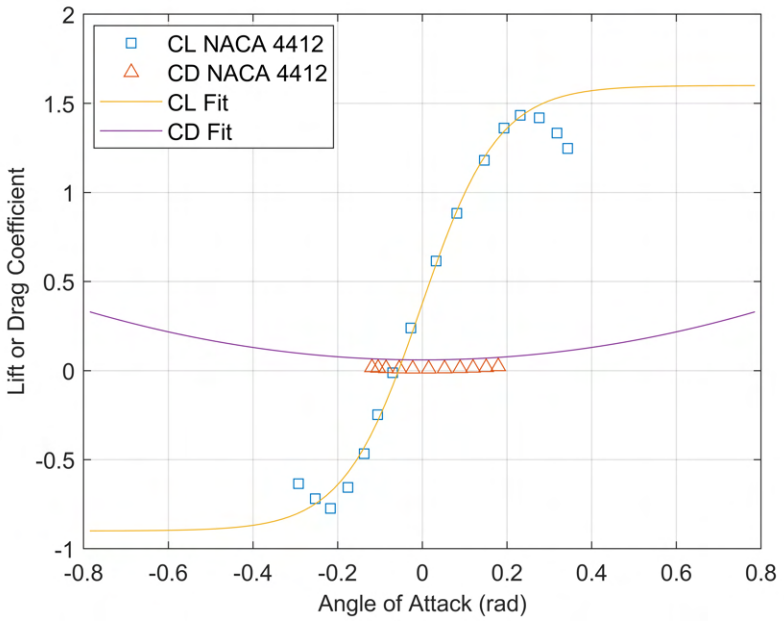


Figure 1.8 Lift (CL) and drag (CD) coefficients as a function of the angle of attack α. The NACA 4412 markers correspond to experimental airfoil data.

The lift coefficient is a sigmoidal function of the angle of attack α and a linear function of the elevator command δ_e.

$$C_L = -\left(C_{L,0} + \frac{C_{L,1}}{C_{L,2} + e^{-C_{L,3}(\alpha + C_{L,4})}}\right) - C_{L,\delta_e}\delta_e \tag{1.65}$$

The drag coefficient C_D is a quadratic function of the angle of attack α (seeFigure 1.8) and a linear function of the elevator command δ_e.

$$C_D = -\left(C_{D,0} + C_{D,2}\alpha^2\right) - C_{D,\delta_e}\delta_e \tag{1.66}$$

The side-slip coefficient C_β is a linear function of the side-slip angle β from Eq. (1.37) and the rudder command δ_r (-1 to 1).

$$C_\beta = -C_{z,\beta}\beta - C_{z,\delta_r}\delta_r \tag{1.67}$$

1.9.2 Aerodynamic Torques

Aerodynamic torques are caused by two fundamental phenomena. First, if the airplane is spinning about any axis, air resistance dampens the spinning motion. For example, if the airplane is rolling about the x-axis with a roll rate of p rad/s, a resistive torque $T_{b,p}$ dampens the motion. We model this resistive torque with a linear relationship with a sign opposite to the rotation. The resistive roll torque is

$$T_{b,p} = -b_p p \tag{1.68}$$

The resistive pitch torque is

$$T_{b,q} = -b_q q \tag{1.69}$$

The resistive yaw torque is

$$T_{b,r} = -b_r r \tag{1.70}$$

The second phenomenon that creates aerodynamic torques is caused by aerodynamic forces that act on the aircraft at a moment arm distance from the center of gravity. These force-and-moment-arm combinations cause torques that can be modeled by the aerodynamic force multiplied by a distance.

$$T_{\text{aero}} = \frac{1}{2}\rho V_a^2 ALC \tag{1.71}$$

The moment arm is L, and the coefficient is C. The torque coefficient C_{Ty} for the pitching torque is a linear function of the angle of attack α and the elevator command δ_e:

$$C_{Ty} = -C_{Ty,\alpha}\alpha - C_{Ty,\delta_e}\delta_e \tag{1.72}$$

The moment arm is the wing chord c (m).

The coefficient for the roll torque about the x-axis is a linear function of the aileron command δ_a (-1 to 1):

$$C_{Tx} = C_{\delta_a}\delta_a \tag{1.73}$$

The moment arm is the wing span S (m).

The coefficient for the yaw torque around the z-axis is a linear function of the side-slip angle β and the rudder command δ_r (-1 to 1):

$$C_{Tz} = C_{Tz,\beta}\beta + C_{Tz,\delta_r}\delta_r \tag{1.74}$$

The moment arm is the length L of the airplane from nose to tail.

Forces and Torques from Aerodynamics

The airspeed is a function of the body-fixed relative velocities:

$$V_a = \sqrt{u_r^2 + v_r^2 + w_r^2}$$

The angle of attack α (rad) is a function of the relative air speeds w_r and u_r (m/s) and is calculated as follows:

$$\alpha = \tan^{-1}\left(\frac{w_r}{V_a}\right)$$

The side-slip angle is

$$\beta = \sin^{-1}\left(\frac{v_r}{V_a}\right)$$

Aerodynamic forces are calculated by

$$f_{\text{aero}} = \frac{1}{2}\rho V_a^2 A$$

where A is the wing area (m^2), and ρ (kg/m^3) is the air density.
The lift force in the body-fixed z-direction is

$$f_{z,L} = f_{\text{aero}}\left(-\left(C_{L,0} + \frac{C_{L,1}}{C_{L,2} + e^{-C_{L,3}(\alpha + C_{L,4})}}\right) - C_{L,\delta_e}\delta_e\right)$$

The drag force in the body-fixed x-direction is

$$f_{x,D} = f_{\text{aero}}\left(-\left(C_{D,0} + C_{D,2}\alpha^2\right) - C_{D,\delta_e}\delta_e\right)$$

The side-slip force in the y-direction is

$$f_{y,\beta} = f_{\text{aero}}\left(-C_{y,\beta}\beta - C_{y,\delta_r}\delta_r\right)$$

The pitching torque about the y-axis is

$$T_y = f_{\text{aero}}\, c\left(-C_{Ty,\alpha}\alpha - C_{Ty,\delta_e}\delta_e\right) - b_r r$$

where c is the wing chord. The roll torque about the x-axis is

$$T_{x,\text{aero}} = f_{\text{aero}} S C_{\delta_a}\delta_a - b_p p$$

where S is the wing span. The yaw torque about the z-axis is

$$T_z = f_{\text{aero}}\, c\left(C_{Tz,\beta}\beta + C_{Tz,\delta_r}\delta_r\right) - b_q q$$

Combined Forces and Torques in the Body-Fixed Directions

$$\begin{aligned} f_x &= f_{x,T} + f_{x,D} + f_{x,g} \\ f_y &= f_{y,\beta} + f_{y,g} \\ f_z &= f_{z,L} + f_{z,g} \\ T_x &= T_{x,\text{aero}} + T_{x,Q} \end{aligned}$$

Chapter 2

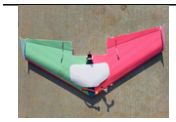
Flight Hardware and Arduino IDE

Contents

This chapter describes the hardware used for autonomous flight. It also explains some Arduino programs for interacting with the hardware. The hardware includes the airplane body, motor, speed controller, servos, battery, sensors, actuators, wires, microcontroller, RC transmitter, receiver, circuit boards, etc. Many different designs and configurations are possible. The objective of this chapter is to describe one example.

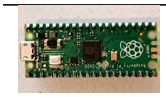
2.1 Bill of Materials: Flying Wing

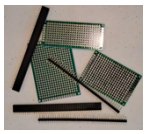
Table 2.1 Hardware Bill-of-Materials for the flying wing as of July 2022. Prices, URLs, and products are subject to change.

Image	Description	Price
	Body Frame and Remote Controller Electronics	
	Modified Version of the Flite Test Versa Wing (https://store.flitetest.com/ft-versa-wing-wr-965mm/) (https://www.youtube.com/watch?v=dveLR_WZDdw)	$29.99 +Shipping
	Colorful Duct Tape (Amazon Search for Colorful Duct Tape)	$14.99 +Shipping

	200 Hot Glue Sticks (Amazon search for Hot Glue Sticks)	$16.99 +Shipping
	Brushless DC motor (Part of FT Power Pack C Twin Radial) (https://store.flitetest.com/ft-power-pack-c-radial-v2/)	$129.99 +Shipping
	ESC Motor controller (Part of FT Power Pack C Twin Radial) (https://store.flitetest.com/ft-power-pack-c-radial-v2/)	See Brushless DC motor
	1045 Propeller (Part of FT Power Pack C Twin Radial) (https://store.flitetest.com/ft-power-pack-c-radial-v2/)	See Brushless DC motor
	Two Servo Motors (Part of FT Power Pack C Twin Radial) (https://store.flitetest.com/ft-power-pack-c-radial-v2/)	See Brushless DC motor
	Servo Rods (https://store.flitetest.com/push-rods-8-pack-16-5/)	$6.00 +Shipping
	Control Horns (https://store.flitetest.com/ft-control-horn-20-pieces/)	$5.00 +Shipping
	Motor Firewall (https://store.flitetest.com/swappable-firewalls-5-pack/)	$5.00 +Shipping
	Programmable Remote Control Transmitter (https://store.flitetest.com/dx6e-6-channel-transmitter-only/)	$249.99 +Shipping
	6 Channel Remote Control Receiver (flitetest.com search for 6-Channel Receiver)	$49.99 +Shipping

2.1 Bill of Materials: Flying Wing

	3 cell LiPo battery (flitetest.com search for 3S Battery)	$19.99 +Shipping
	3 cell LiPo battery charger (https://store.flitetest.com/omphobby-balance-charger/)	$14.99 +Shipping
Microcontroller, Sensors, and Data-Logger		
	Raspberry Pi Pico RP2040 Microcontroller (Google search for Raspberry Pi Pico)	$4.00 +Shipping
	Adafruit Precision NXP 9-DOF Breakout Board - FXOS8700 + FXAS21002 (https://www.adafruit.com/product/3463)	$14.95 +Shipping
	Adafruit Mini GPS PA1010D - UART and I2C - STEMMA QT (https://www.adafruit.com/product/4415)	$29.95 +Shipping
	Adafruit DPS310 Barometric Pressure / Altitude Sensor (https://www.adafruit.com/product/4494)	$6.95 +Shipping
	5V to 3.3V Logic Level Converter (Amazon Search for Level Converter 5 to 3.3v)	$6.99 +Shipping
	SD Card Reader / Data Logger (Amazon search for SD Card Reader Arduino)	$6.99 +Shipping
	Micro SD Card with Adapter (Amazon search for Micro SD Card With Adapter)	$9.32 +Shipping
Other Electronics, Wires, and Connectors		
	Spools of Wire (Amazon search for Spools of Wire)	$15.99 +Shipping

	Servo Wires and Connectors (Amazon search for Servo Wires with Connectors)	$14.99 +Shipping
	Heat Shrink Tubing (Amazon search for Heat Shrink Tubing)	$8.88 +Shipping
	Prototype PCB Boards and Headers (Amazon search for Prototype PCB Boards)	$14.99 +Shipping
	USB-A to Micro B Cable (Amazon search for Micro USB Cable)	$2.49 +Shipping
	2N2222A NPN Transistor (Amazon search for 2N2222A NPN Transistor)	$6.99 +Shipping
	Resistors (Amazon search for Arduino Resistors Kit)	$6.99 +Shipping
	USB-A to Micro B Cable (Amazon search for Arduino Switch)	$6.98 +Shipping
	LED (Amazon search for Arduino LED)	$5.99 +Shipping
	Diode (Amazon search for Low Forward Bias Diode)	$6.99 +Shipping
Tools and Accessories		
	Hot Glue Gun (https://store.flitetest.com/flite-test-80w-glue-gun/)	$14.99 +Shipping
	Digital Multimeter (Amazon search for Digital Multimeter)	$14.99 +Shipping

2.1 Bill of Materials: Flying Wing

	Soldering Kit (Amazon search for Soldering Kit)	$18.99 +Shipping
	Single Edge Razor Blades (Amazon search for Single Edge Razor Blade)	$2.75 +Shipping
	CR1220 Coin Cell Battery (Amazon search for CR 1220 Battery)	$5.69 +Shipping
	AA Batteries (Amazon search for AA Battery)	$6.48 +Shipping

According to the Bill-of-Materials in Table 2.1, remote-controlled flight costs around $550 - $600, and autonomous flight costs about $800. The flying wing used in this document is actually a little different from the Flite Test Versa Wing, and it is cut from a custom template that fits the required autopilot electronics better. Foam parts cut from the custom template are shown in Figure 2.1.

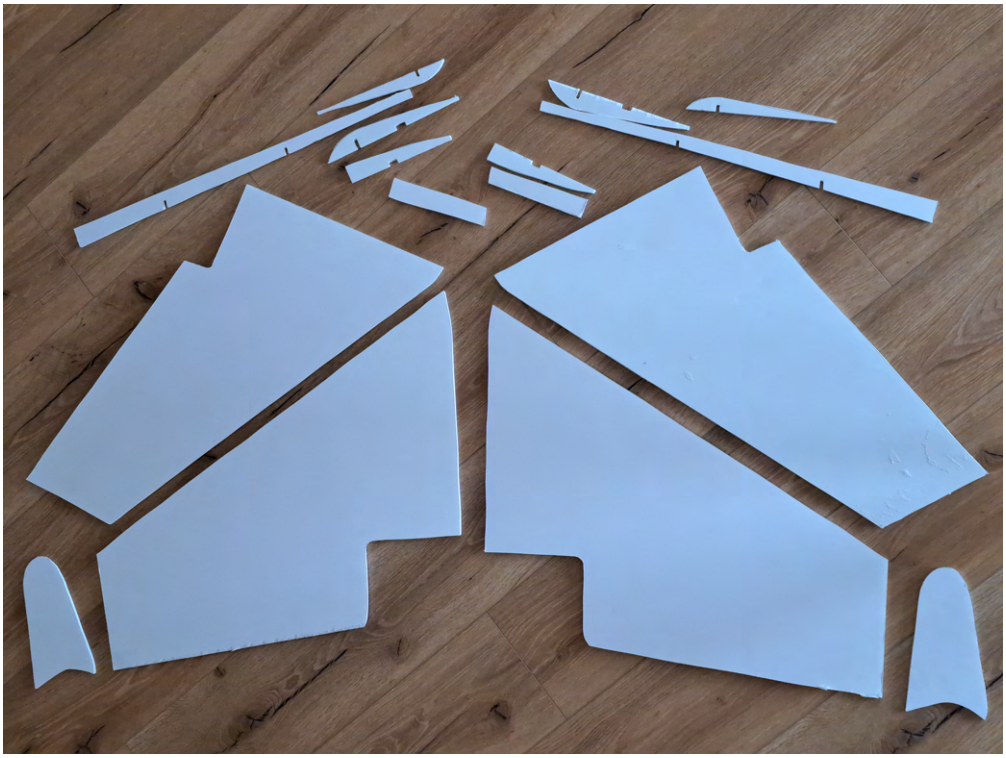


Figure 2.1 Foam pieces of the flying wing

2.2 Wiring Diagram: Flying Wing Autopilot

A custom Printed Circuit Board (PCB) is especially beneficial for decreasing the amount of wires required for the autopilot circuitry. Figure 2.2 shows a wiring sketch of the autopilot circuit. The portion of the circuit in the green dashed line is recommended to be part of a custom PCB.

Figure 2.3 shows a wiring diagram created in Easy-EDA for the sketch of Figure 2.2. Figure 2.4 shows the top and bottom layers of the flying wing Printed Circuit Board (PCB).

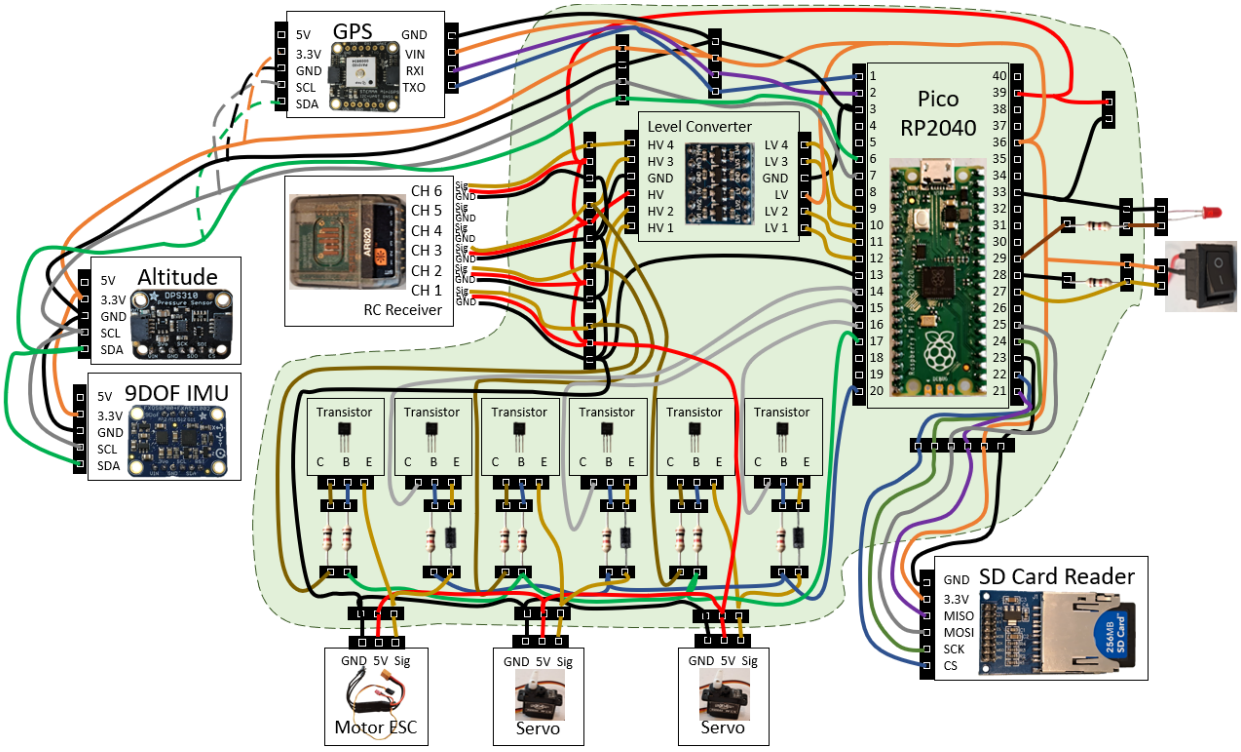


Figure 2.2 Wiring sketch for the autopilot circuit.

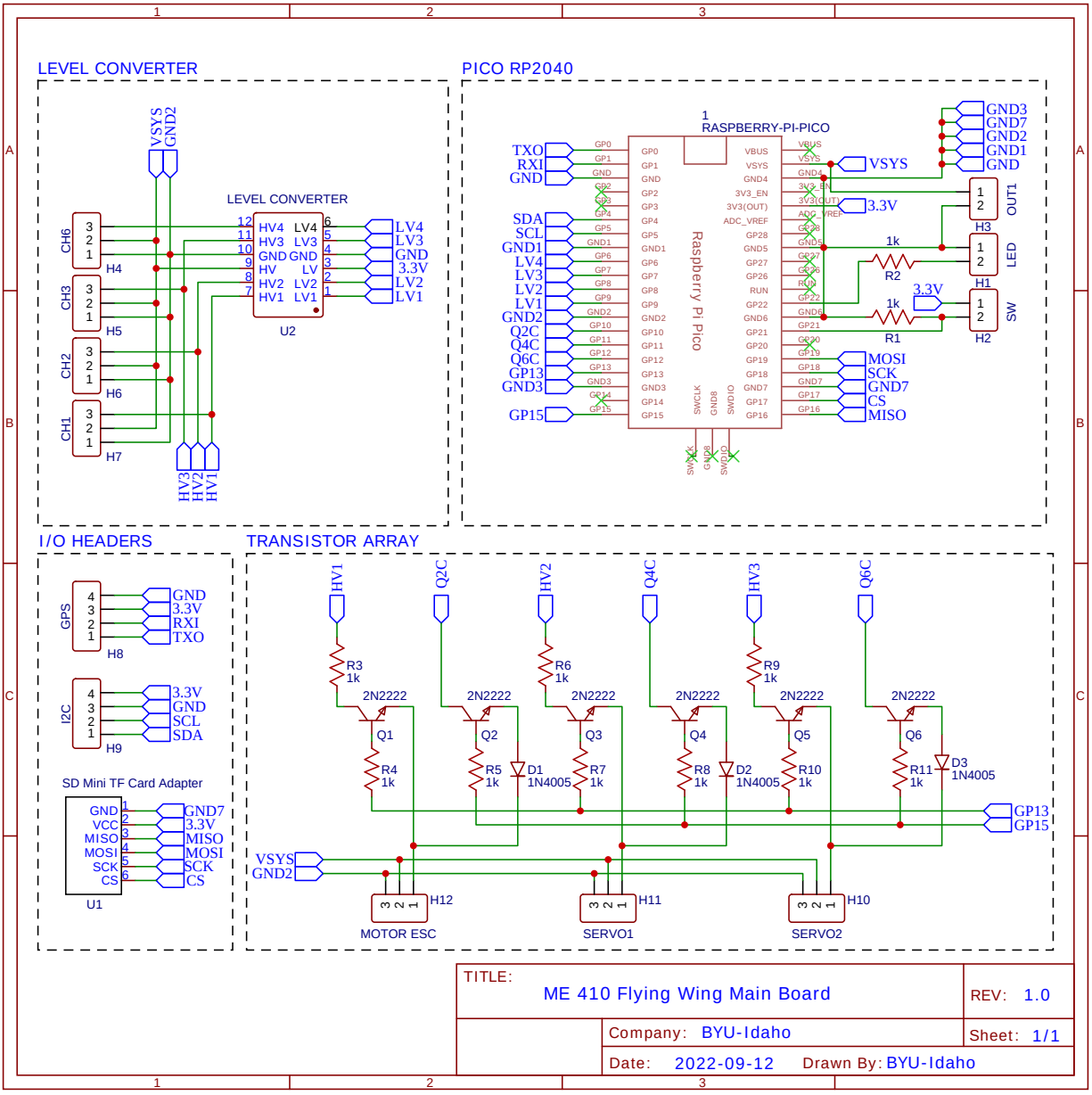
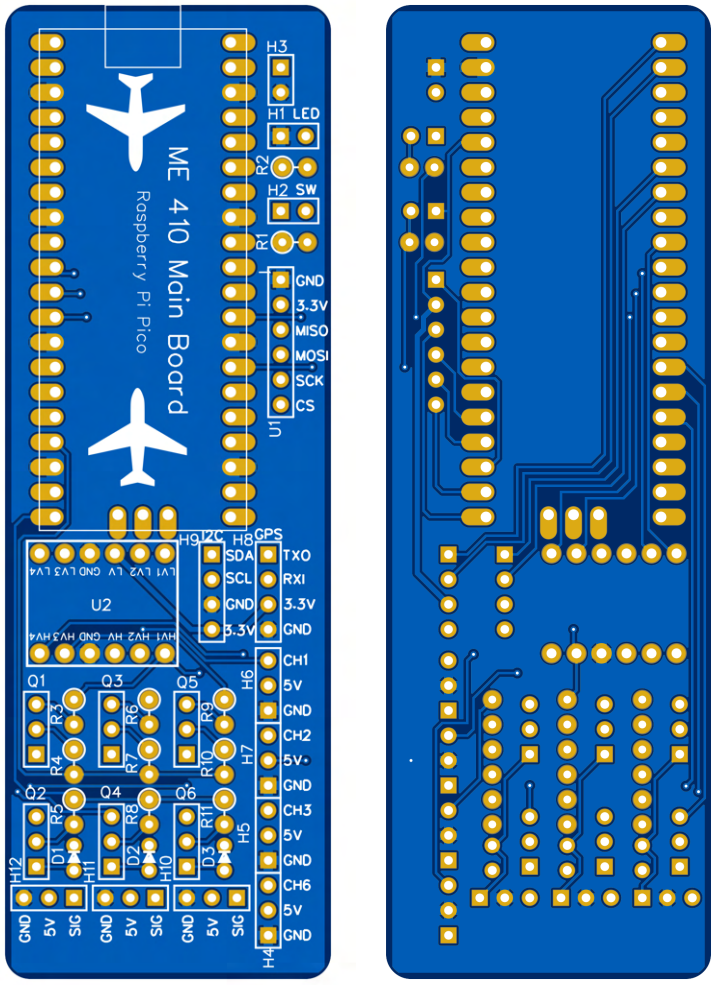


Figure 2.3 Wiring diagram for the flying wing autopilot. (Source: Paul Passe-Carlus, 2022)

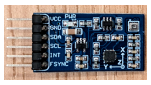
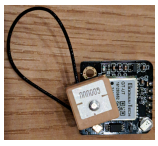


(a) **(b)**

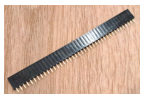
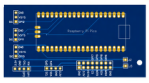
Figure 2.4 Top layer (a) and bottom layer (b) of flying wing PCB. (Source: Paul Passe-Carlus, 2022)

2.3 Bill of Materials: Glider

Table 2.2 Hardware Bill-of-Materials for the glider as of December 2022. Prices, URLs, and products are subject to change.

Image	Description	Price
	Body Frame and Components	
	Custom glider design from foam board (Foam board from dollar store)	$1.25
	200 Hot Glue Sticks (Amazon search for Hot Glue Sticks)	$16.99 +Shipping
	Three 9 Gram Servo Motors (Amazon search for 9 Gram Metal Gear Servo)	$13.79 +Shipping
	Servo Rods (https://store.flitetest.com/push-rods-8-pack-16-5/)	$6.00 +Shipping
	Control Horns (https://store.flitetest.com/ft-control-horn-20-pieces/)	$5.00 +Shipping
	Electronics	
	Raspberry Pi Pico RP2040 Microcontroller (Google search for Raspberry Pi Pico)	$4.00 +Shipping
	USB-A to Micro B Cable (Amazon search for Micro USB Cable)	$2.49 +Shipping
	10 DOF IMU (Amazon search for 10 DOF IMU)	$37.99 +Shipping
	GPS Module (Amazon search for Arduino GPS Module)	$12.59 +Shipping

2.3 Bill of Materials: Glider

	SD Card Reader / Data Logger (Amazon search for SD Card Reader Arduino)	$6.99 +Shipping
	Micro SD Card with Adapter (Amazon search for Micro SD Card With Adapter)	$9.32 +Shipping
	1 cell LiPo battery / charger with JST plug (Amazon search for 1s LiPo Battery with Charger)	$21.99 +Shipping
	JST Two-Pin Plug Connector (Amazon search for JST Connector)	$6.29 +Shipping
	Bent Male Pin Header (Amazon search for Bent Male Pin Header)	$4.99 +Shipping
	Female Pin Header (Amazon search for Female Pin Header)	$5.99 +Shipping
	Custom Printed Circuit Board (Custom order through EasyEDA / JLCPCB)	$21.05
Tools and Accessories		
	Hot Glue Gun (https://store.flitetest.com/flite-test-80w-glue-gun/)	$14.99 +Shipping
	Digital Multimeter (Amazon search for Digital Multimeter)	$14.99 +Shipping
	Soldering Kit (Amazon search for Soldering Kit)	$18.99 +Shipping

	Single Edge Razor Blades	$2.75
	(Amazon search for Single Edge Razor Blade)	+Shipping

2.4 Wiring Diagram: Glider Autopilot

Figure 2.5 shows the wiring diagram for the glider's autopilot. Figure 2.6 shows the top and bottom layers of the Printed Circuit Board (PCB) for the glider.

Figure 2.7 shows the PCB assembled with the sensors onboard the glider.

2.4 Wiring Diagram: Glider Autopilot

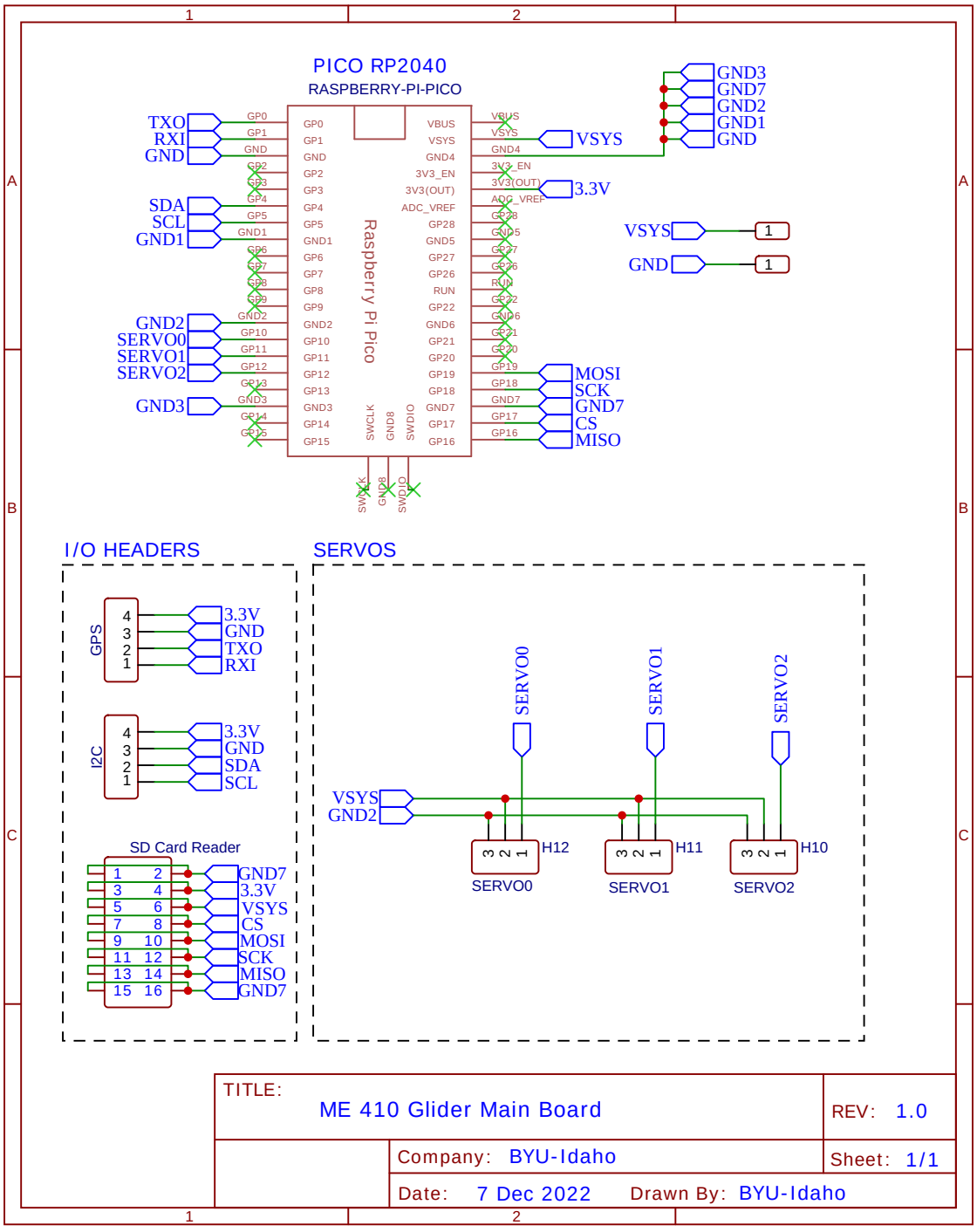
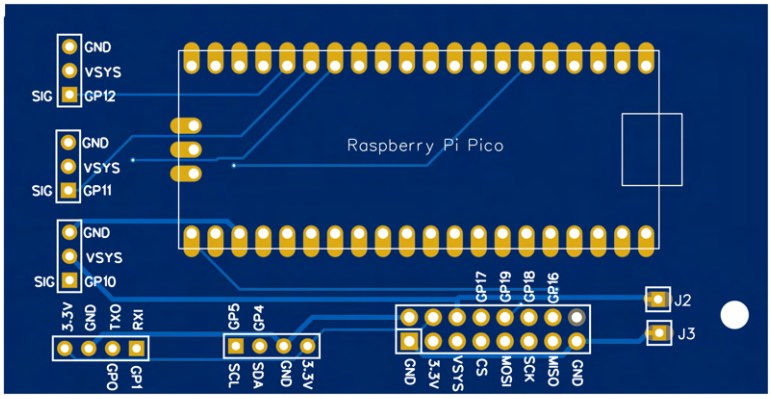
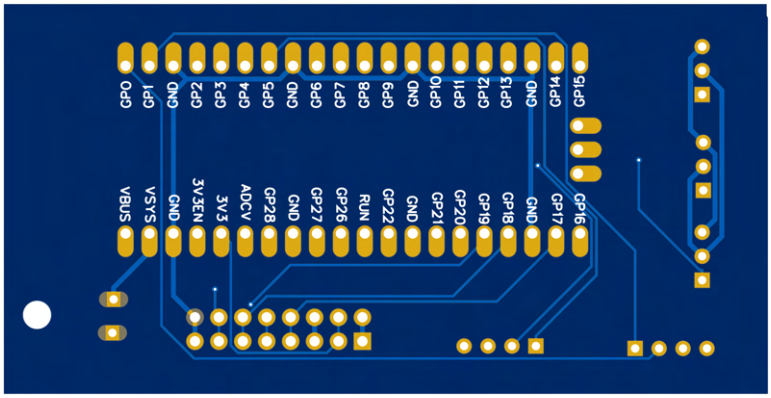


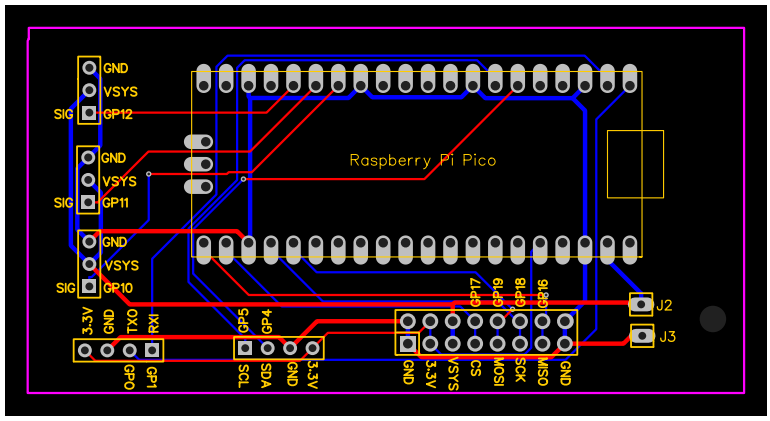
Figure 2.5 Wiring diagram for the glider's autopilot circuit.



(a)

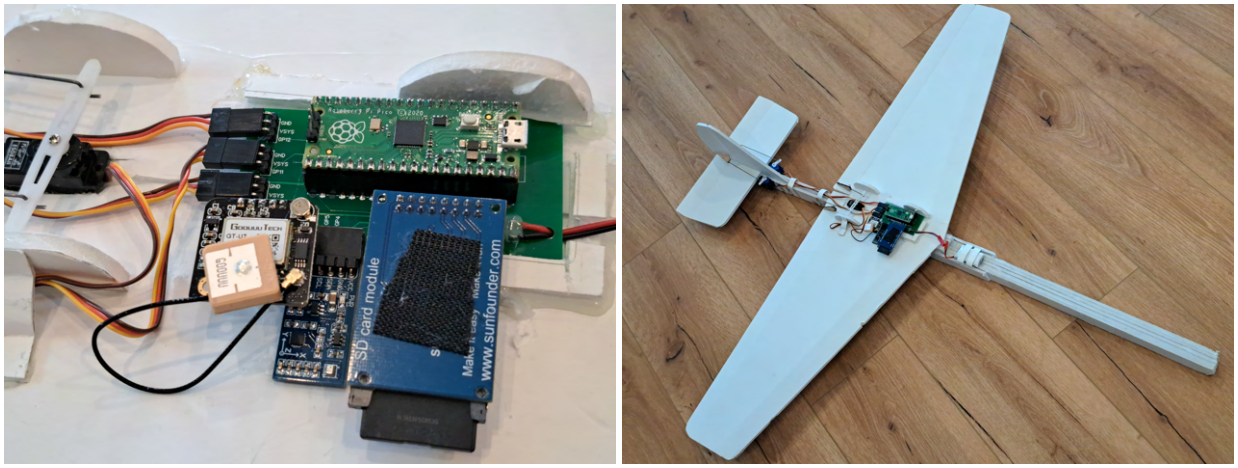


(b)



(c)

Figure 2.6 Top layer (a), bottom layer (b), and routing diagram (c) of the glider's PCB.



(a) **(b)**

Figure 2.7 PCB assembled with sensors, microcontroller, and actuators (a) onboard the glider (b).

Chapter 3

Sensor Setup and Calibration

Contents

This chapter describes sensor setup and calibration. It discusses initial experiments to verify that the sensors are producing expected signals. It discusses calibrating the sensors and filtering raw signals to produce cleaner data. Sensor setup and calibration is essential for successful flight monitoring and autonomous control.

3.1 Reading Measurements the IMU Sensor

Using a microcontroller to read data from sensors is often made easier by using libraries created by others that are available to the community. This section provides an example of how to use Arduino libraries to read gyrometer, accelerometer, magnetometer, temperature, pressure, and altitude data from a 10 DOF sensor (see Figure 3.1). The 10 DOF sensor uses an MPU9250 sensor which includes a 3-axis magnetometer, accelerometer, and gyrometer. The 10 DOF sensor also includes a BMP280 sensor, which measures temperature, pressure, and altitude.

The code below uses two Arduino libraries, “MPU9250_WE.h” (version 1.2.6) and “BMP280_DEV.h” (version 1.0.21) to read the measurements from the 10 DOF sensor. The code is modified from examples included with those libraries. Read the comments in the code to understand how the libraries are used to read measurements from the 10 DOF sensor. The code below specifically applies to the MPU9250 and BMP280 sensors; other sensors would possibly also have Arduino libraries with examples that could be modified to obtain the needed measurements.

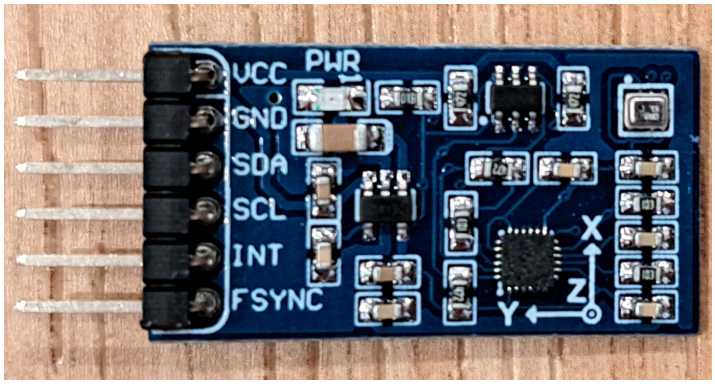


Figure 3.1 The 10 Degree-Of-Freedom (DOF) Inertial Measurement Unit (IMU) MPU9250 and Altitude Sensor BMP280.

```
//Include MPU 9250 version 1.2.6.  To get it, go to
//Sketch>>Include Library>>Manage Libraries...  Search for MPU6250_WE
#include <MPU9250_WE.h>
#include <BMP280_DEV.h> // Include the BMP280_DEV.h library

//Define MPU9250_ADDR to be the I2C address of the MPU 9250
#define MPU9250_ADDR 0x68
float temperature, pressure, altitude;  //declare temperature, pressure, and altitude

//Create the MPU9250 object and name it myMPU9250
MPU9250_WE myMPU9250 = MPU9250_WE(MPU9250_ADDR);
//Create the BMP280_DEV object and name it bmp280.  The I2C address is 0x77
BMP280_DEV bmp280;


void setup() {
  Serial.begin(9600); //Begin serial
  Wire.begin(); //Begin I2C
  delay(2000); //Wait 2 seconds

  if(!myMPU9250.init()){ //Start the MPU, if it fails, report an error
    Serial.println("MPU9250 does not respond");
  }
  else{ //If it succeeds, report success
    Serial.println("MPU9250 is connected");
  }
  if(!myMPU9250.initMagnetometer()){ //Start the magnetometer, if failure, report
    Serial.println("Magnetometer does not respond");
  }
  else{ //if success, report
    Serial.println("Magnetometer is connected");
  }

  /* The slope of the curve of acceleration vs measured values fits quite well to the
```

```
* theoretical values, e.g. 16384 units/g in the +/- 2g range. But the starting point,
* if you position the MPU9250 flat, is not necessarily 0g/0g/1g for x/y/z. The
* autoOffset function measures offset values. It assumes your MPU9250 is positioned
* flat in the x,y-plane. The more you deviate from this, the less accurate will be
* your results.
* The function also measures the offset of the gyroscope data. The gyroscope offset
* does not depend on the positioning.
* This function needs to be called at the beginning since it can overwrite your settings!
*/
Serial.println("Position your MPU9250 flat and don't move it - calibrating...");
delay(1000);
myMPU9250.autoOffsets(); //Callibrate the accelerometer and gyro offsets
Serial.println("Done!");

/*  This is a more accurate method for calibration. You have to determine the minimum
*  and maximum raw acceleration values of the axes determined in the range +/- 2 g.
*  You call the function as follows: setAccOffsets(xMin,xMax,yMin,yMax,zMin,zMax);
*  Use either autoOffset or setAccOffsets, not both.
*/
//myMPU9250.setAccOffsets(-14240.0, 18220.0, -17280.0, 15590.0, -20930.0, 12080.0);

/*  The gyroscope data is not zero, even if you don't move the MPU9250.
*  To start at zero, you can apply offset values. These are the gyroscope raw values you
* obtain using the +/- 250 degrees/s range.
*  Use either autoOffset or setGyrOffsets, not both.
*/
//myMPU9250.setGyrOffsets(45.0, 145.0, -105.0);

/*  You can enable or disable the digital low pass filter (DLPF). If you disable the DLPF,
*  you need to select the bandwdith, which can be either 8800 or 3600 Hz. 8800 Hz has a
*  shorter delay, but higher noise level. If DLPF is disabled, the output rate is 32 kHz.
*  MPU9250_BW_WO_DLPF_3600
*  MPU9250_BW_WO_DLPF_8800
*/
myMPU9250.enableGyrDLPF();
//myMPU9250.disableGyrDLPF(MPU9250_BW_WO_DLPF_8800); // bandwdith without DLPF

/*  Digital Low Pass Filter for the gyroscope must be enabled to choose the level.
*  MPU9250_DPLF_0, MPU9250_DPLF_2, ..... MPU9250_DPLF_7
*
*  DLPF    Bandwidth [Hz]   Delay [ms]   Output Rate [kHz]
*    0         250            0.97             8
*    1         184            2.9              1
*    2          92            3.9              1
*    3          41            5.9              1
*    4          20            9.9              1
*    5          10           17.85             1
*    6           5           33.48             1
*    7        3600            0.17             8
*
*    You achieve lowest noise using level 6, but it also results in a phase shift
*/
myMPU9250.setGyrDLPF(MPU9250_DLPF_1);
```

```
/*  Sample rate divider divides the output rate of the gyroscope and accelerometer.
*  Sample rate = Internal sample rate / (1 + divider)
*  It can only be applied if the corresponding DLPF is enabled and 0<DLPF<7!
*  Divider is a number 0...255
*/
myMPU9250.setSampleRateDivider(5);

/*  MPU9250_GYRO_RANGE_250       250 degrees per second (default)
*  MPU9250_GYRO_RANGE_500       500 degrees per second
*  MPU9250_GYRO_RANGE_1000     1000 degrees per second
*  MPU9250_GYRO_RANGE_2000     2000 degrees per second
*/
myMPU9250.setGyrRange(MPU9250_GYRO_RANGE_250);

/*  MPU9250_ACC_RANGE_2G      2 g   (default)
*  MPU9250_ACC_RANGE_4G      4 g
*  MPU9250_ACC_RANGE_8G      8 g
*  MPU9250_ACC_RANGE_16G    16 g
*/
myMPU9250.setAccRange(MPU9250_ACC_RANGE_2G);

/*  Enable/disable the digital low pass filter for the accelerometer
*  If disabled the bandwidth is 1.13 kHz, delay is 0.75 ms, output rate is 4 kHz
*/
myMPU9250.enableAccDLPF(true);

/*  Digital low pass filter (DLPF) for the accelerometer, if enabled
*  MPU9250_DPLF_0, MPU9250_DPLF_2, ..... MPU9250_DPLF_7
*   DLPF     Bandwidth [Hz]      Delay [ms]    Output rate [kHz]
*     0           460               1.94           1
*     1           184               5.80           1
*     2            92               7.80           1
*     3            41              11.80           1
*     4            20              19.80           1
*     5            10              35.70           1
*     6             5              66.96           1
*     7           460               1.94           1
*/
myMPU9250.setAccDLPF(MPU9250_DLPF_1);

/* You can enable or disable the axes for gyroscope and/or accelerometer measurements.
* By default all axes are enabled. Parameters are:
* MPU9250_ENABLE_XYZ  //all axes are enabled (default)
* MPU9250_ENABLE_XY0  // X, Y enabled, Z disabled
* MPU9250_ENABLE_X0Z
* MPU9250_ENABLE_X00
* MPU9250_ENABLE_0YZ
* MPU9250_ENABLE_0Y0
* MPU9250_ENABLE_00Z
* MPU9250_ENABLE_000  // all axes disabled
*/
//myMPU9250.enableAccAxes(MPU9250_ENABLE_XYZ);
```

```
  //myMPU9250.enableGyrAxes(MPU9250_ENABLE_XYZ);

  /*
  * AK8963_PWR_DOWN
  * AK8963_CONT_MODE_8HZ         default
  * AK8963_CONT_MODE_100HZ
  * AK8963_FUSE_ROM_ACC_MODE
  */
  myMPU9250.setMagOpMode(AK8963_CONT_MODE_100HZ);
  delay(200);

  bmp280.begin(); // Default initialisation, place the BMP280 into SLEEP_MODE
  //bmp280.setPresOversampling(OVERSAMPLING_X4); // Set the pressure oversampling to X4
  //bmp280.setTempOversampling(OVERSAMPLING_X1); // Set the temperature oversampling to X1
  //bmp280.setIIRFilter(IIR_FILTER_4); // Set the IIR filter to setting 4
  bmp280.setTimeStandby(TIME_STANDBY_2000MS); // Set the standby time to 2 seconds
  bmp280.startNormalConversion(); // Start BMP280 continuous conversion in NORMAL_MODE

  //Print a header
  //For example, only display X, Y, and Z magnetometer data
  Serial.println("X, Y, Z");
}

void loop() {
  //Get the values from the accelerometer
  xyzFloat accel = myMPU9250.getGValues();

  //Get the values from the Gyrometer
  xyzFloat gyro = myMPU9250.getGyrValues();

  //Get the values from the Magnetometer
  xyzFloat Mag = myMPU9250.getMagValues();

  //Get the bmp280 measurements
  //Temperature in Celsius, pressure in hectopascals, altitude in meters
  bmp280.getMeasurements(temperature, pressure, altitude);

  //Print the values to the Serial monitor or plotter
  Serial.print(Mag.x); Serial.print(",");
  Serial.print(Mag.y); Serial.print(",");
  Serial.println(Mag.z);

delay(50);

}
```

3.1.1 Magnetometer Calibration

Hard iron and soft iron disturbances cause errors in magnetometer measurements. **Hard iron** distortions are caused by local magnetic sources, such as permanent magnets or electromagnetic coils. Hard iron effects are common in electric aircraft because motors can have both permanent magnets and electro-

magnets. **Soft iron** disturbances are caused by ferromagnetic materials such as iron or nickel that can distort or deflect a magnetic field. Nuts and bolts, motor casing and shafts, or other local metal parts can cause soft iron distortions. Magnetometer calibration can help minimize the effect of these distortions.

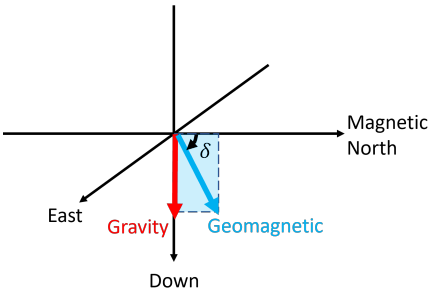


Figure 3.2 Gravitational and geomagnetic vectors in the NED reference frame

In the inertial frame, the geomagnetic field is a constant vector pointing in a single direction, see Figure 3.2. However, in the rotating body-frame, the magnitude is constant, but the direction changes. Therefore, in the body-frame, undistorted geomagnetic measurements would lie on the surface of a sphere centered around the origin (0,0,0) with a radius equal to the geomagnetic field strength B. Hard iron disturbances shift the entire geomagnetic field away from the (0,0,0) origin. Soft iron disturbances deform the sphere into an ellipsoid. These effects can be modeled by the following equation:

$$M_d = K^{-1}M + \beta \tag{3.1}$$

M_d is the raw magnetometer signal. It is the distorted magnetometer reading. M is the true, but unknown, undistorted geomagnetic vector from the perspective of the body-frame:

$$M = \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} \tag{3.2}$$

The hard iron effect β is the amount the measurement has been shifted from the (0,0,0) origin:

$$\beta = \begin{bmatrix} \beta_x \\ \beta_y \\ \beta_z \end{bmatrix} \tag{3.3}$$

The soft iron effect is due to the symmetric distortion matrix K:

$$K = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{12} & k_{22} & k_{23} \\ k_{13} & k_{23} & k_{33} \end{bmatrix} \tag{3.4}$$

Calibration of the magnetometer signal involves calculating both the offset β and distortion matrix K. If these are known, the undistorted geomagnetic vector is calculated by

$$M = K\left(M_d - \beta\right) \tag{3.5}$$

Determining β and K is done by first collecting many magnetometer measurements while the body is rotating. Finding the centroid of the measurements determines β. The distortion matrix is found by fitting the equation for an ellipsoid to the data.

There are many techniques to find the centroid β. One efficient and effective way is to iteratively move β closer to the farthest data point and farther from the nearest point. The MATLAB algorithm below uses this approach to find the centroid β. It finds the unit vectors between the present iteration's estimate of β and the nearest and farthest magnetometer data points. Then it takes small steps at each iteration towards the farthest data point and away from the nearest.

There are also many methods to find the distortion matrix K. The method in the MATLAB algorithm below works by sequentially performing the following steps:

1. Determine the unit vectors for the major (longest) and minor (shortest) axes of the ellipsoid
2. Rotate the major axis to align with a known x-axis and the minor axis to align with a known y-axis
3. Scale the major axis by $\frac{R_{avg}}{R_{\max}}$ where R_{avg} is the average distance between the centroid and the ellipsoid data and $R_{\max}$ is the radius of the major axis
4. Scale the minor axis by $\frac{R_{avg}}{R_{\min}}$ where R_{avg} is the average distance between the centroid and the ellipsoid data and $R_{\min}$ is the radius of the minor axis
5. Rotate the ellipsoid back to its original orientation

The process for finding β and K is shown schematically in Figure 3.3.

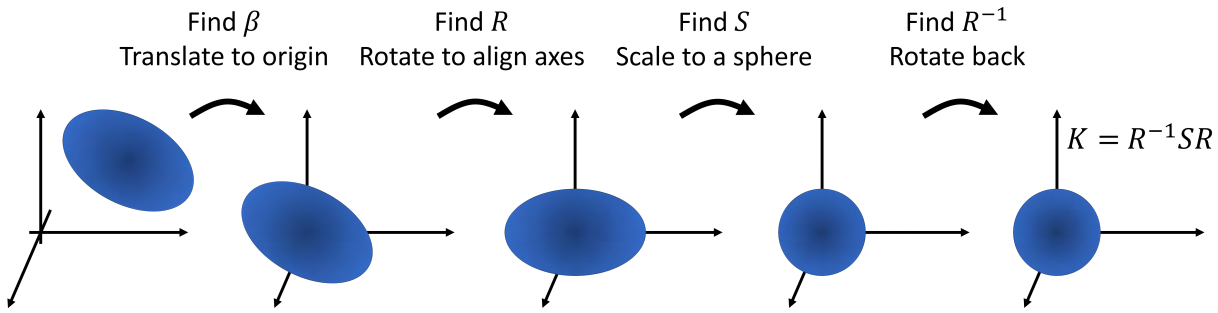


Figure 3.3 The process for finding β and K

To improve efficiency, the MATLAB algorithm below also down-samples the data to have a more uniform distribution around the magnetometer ellipsoid. The result of the algorithm on the data provided at the end of this section is shown in the graph of Figure 3.4. The calibrated data is labeled "Best Fit". The calibrated magnetometer data are more spherical (better fit to the surface of a sphere) than the original and are centered around the origin (0,0,0).

MATLAB algorithm to calibrate a magnetometer:

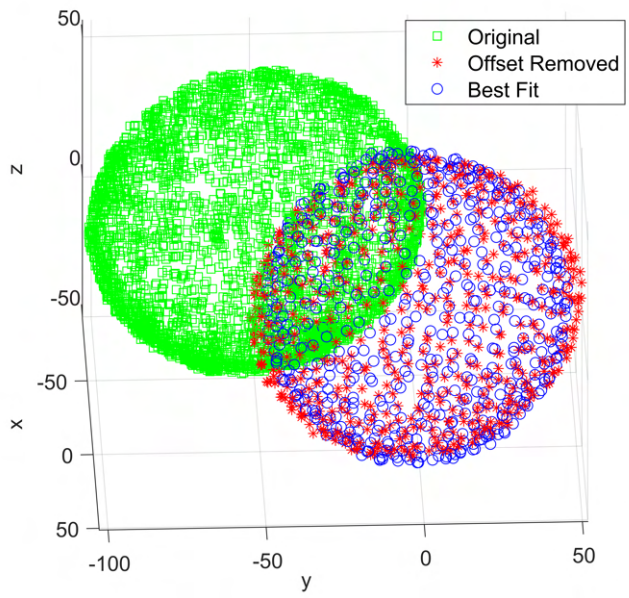


Figure 3.4 Performance of the MATLAB magnetometer calibration algorithm on actual magnetometer data

```
close all
clear all
clc

%% import the data from a .csv file
% The columns of the .csv file are as follows
% Column 1: The header is labeled mag0, the data is the x-axis
% magnetometer data in any units
% Column 2: The header is labeled mag1, the data is the y-axis
% magnetometer data in any units
% Column 3: The header is labeled mag2, the data is the z-axis
% magnetometer data in any units
[matlabFile,path] = uigetfile('*.csv', ...
  'Select The IMU data');
mData = readtable([path,matlabFile]);


%% extract the data
%magnetometer
xM = mData.mag0; %x-axis magnetometer data
yM = mData.mag1; %y-axis magnetometer data
zM = mData.mag2; %z-axis magnetometer data

%remove zeros from the data
Na = length(xM);
xMag = [ ]; %empty array to store nonzero x-axis magnetometer data
yMag = [ ]; %empty array to store nonzero y-axis magnetometer data
zMag = [ ]; %empty array to store nonzero z-axis magnetometer data
```

```
for ii = 1:Na
  if xM(ii) ~= 0 && yM(ii) ~= 0 && zM(ii) ~= 0
    %only populate with nonzero data
    xMag = [xMag;xM(ii)]; %nonzero x-axis magnetometer data
    yMag = [yMag;yM(ii)]; %nonzero y-axis magnetometer data
    zMag = [zMag;zM(ii)]; %nonzero z-axis magnetometer data
  end
end

% get the length of the data
N = length(xMag);

%select a threshold, data will be down-sampled so that remaining samples are
%separated by at least the threshold distance
threshold = 5;

%Get the first point that will be kept
XYZdata = [xMag(1),yMag(1),zMag(1)];
for ii = 1:N
  nXYZ = size(XYZdata,1); %get the length of the kept-data array
  flag = 0; %0 indicates that the current (iith) magnetometer data is not
  %within the threshold distance to any other kept data
  for jj = 1:nXYZ
    %check if the current (iith) magnetometer data is within the
    % threshold distance to any other kept data
    if sqrt((xMag(ii)-XYZdata(jj,1))^2+...
        (yMag(ii)-XYZdata(jj,2))^2+...
        (zMag(ii)-XYZdata(jj,3))^2) < threshold
      %the data is too close to existing kept data.  Remove it.
      flag = 1;
      break;
    end
  end
  if ~flag
    %The data was not too close to existing kept data.  Keep it.
    XYZdata = [XYZdata;[xMag(ii),yMag(ii),zMag(ii)]];
  end
end

% Find the offset (beta) of the data
[bx,by,bz] = function_FitSphereBeta(XYZdata);
beta = [bx,by,bz]

%Find the distortion matrix (K) of the data
K1 = function_FitSphereK(XYZdata, beta)
K2 = function_FitSphereK((K1*(XYZdata-beta)')', zeros(1,3))
K3 = function_FitSphereK((K2*K1*(XYZdata-beta)')', zeros(1,3))
K = K3 * K2 * K1

%remove the bias
x = XYZdata(:,1)-bx;
y = XYZdata(:,2)-by;
z = XYZdata(:,3)-bz;

%rotate and scale the data
```

```
xyz = K * [x';y';z'];

% plot the results
figure
scatter3(xMag,yMag,zMag,'gs')
hold on
scatter3(XYZdata(:,1)-bx,XYZdata(:,2)-by,XYZdata(:,3)-bz,'r*')
scatter3(xyz(1,:),xyz(2,:),xyz(3,:),'bo')
legend('Original','Offset Removed','Best Fit')
xlabel('x')
ylabel('y')
zlabel('z')
axis('equal')


function [bx,by,bz] = function_FitSphereBeta(XYZdata)
%Get the total number of magnetometer readings
N = length(XYZdata); %Number of magnetometer data points

%Get a rough prediction of the location of the sphere's centroid for
% an initial condition
beta = [mean(XYZdata(:,1)),mean(XYZdata(:,2)),mean(XYZdata(:,3))];

for jj = 1:1000
  % find the index of the largest distance from the center
  Rmax = 0; %reset the value of the largest distance
  Rmin = inf; %reset the value of the smallest distance
  Ravg = 0;
  for ii = 1:N
    %get the vector from the centroid (beta) to the data point
    vec = XYZdata(ii,:) - beta;
    %assign a distance from the centroid to each XYZdata point
    R = sqrt(vec*vec');
    %update the average radius
    Ravg = Ravg + R/N;
    if R > Rmax
      Rmax = R; %Set the value of the largest distance
      uMax = vec / norm(vec); %get the unit vector
    end
    if R < Rmin %If a new smallest distance is found
      Rmin = R; %set the value
      uMin = vec / norm(vec); %get the unit vector
    end
  end

  %move the center slightly towards the sphere surface particle that is
  % farthest away and slightly away from the point that is nearest
  beta = beta + ...
    0.5 * abs(Ravg-Rmax)/Ravg * uMax... %Move closer to farthest
    - 0.5 * abs(Ravg-Rmin)/Ravg * uMin; %Move away from nearest
end

%output the offset values
bx = beta(1); %magnetometer x-axis offset
by = beta(2); %magnetometer y-axis offset
```

```
bz = beta(3); %magnetometer z-axis offset

end


function K = function_FitSphereK(XYZdata, beta)
%Get the total number of magnetometer readings
N = length(XYZdata); %Number of magnetometer data points

%remove the bias
bx = beta(1);
by = beta(2);
bz = beta(3);
x = XYZdata(:,1)-bx;
y = XYZdata(:,2)-by;
z = XYZdata(:,3)-bz;
xyz = [x';y';z'];


% find the index of the largest distance from the center
Rmax = 0; %reset the value of the largest distance
uMax = zeros(1,3); %Reset the unit vector for the the largest distance
Rmin = inf; %reset the value of the smallest distance
uMin = zeros(1,3); %Reset the unit vector for the the smallest distance
Ravg = 0;
for ii = 1:N
  %get the vector from the centroid (beta) to the data point
  vec = xyz(:,ii)';
  %assign a distance from the centroid to each XYZdata point
  R = sqrt(vec*vec');
  %update the average radius
  Ravg = Ravg + R/N;
  if R > Rmax
    Rmax = R; %Set the value of the largest distance
    uMax = vec / norm(vec); %get the unit vector
  end
  if R < Rmin %If a new smallest distance is found
    Rmin = R; %set the value
    uMin = vec / norm(vec); %get the unit vector
  end
end

%set the body x-axis to the max unit vector
ux = uMax / norm(uMax);

% get the cross product between the min and max unit vectors
uz = cross(ux/norm(ux), uMin/norm(uMin));
uz = uz/norm(uz);

% the body y-axis is the cross product between the z and x axes
uy = cross(uz, ux);
uy = uy/norm(uy);

%get the rotation matrix
Rb2I = [ux',uy',uz'];
```

```
%get the scalar multipliers
multMax = Ravg / Rmax;
multMin = Ravg / Rmin;

%get the K matrix
K = Rb2I * diag([multMax,multMin,1]) * Rb2I';

end
```

Below are some actual magnetometer data to test the algorithm:

```
        mag0        mag1        mag2
        -35.9        -38.3        -32.2
        -33.1        -26.3        -27.4
        -47.3        -24.1        -29.7
        -22.1        -43.5        -30.4
        -9.5        -37.3        -24.8
        0.1        -37.1        -15.3
        4.3        -32.0        -2.3
        2.8        -26.4        11.1
        -1.4        -24.6        25.5
        -12.3        -20.9        35.0
        -24.7        -20.2        44.4
        -38.0        -21.1        49.0
        -53.1        -26.1        50.2
        -65.6        -27.6        45.8
        -78.2        -28.4        36.3
        -85.3        -27.8        24.5
        -85.9        -24.8        8.9
        -78.6        -21.1        -4.5
        -70.6        -21.8        -15.9
        -21.4        -21.6        -21.3
        -8.1        -24.6        -14.6
        -61.2        -31.7        -25.9
        -2.1        -34.5        35.0
        -13.4        -35.7        44.4
        -25.5        -37.4        51.1
        -39.9        -41.1        53.6
        -56.3        -43.7        53.1
        -71.1        -42.9        46.5
        -82.5        -43.4        36.8
        -89.0        -40.2        19.7
        -89.6        -38.1        6.4
        -81.8        -37.3        -11.0
        -73.3        -40.9        -21.3
        -58.8        -45.3        -30.4
        -44.0        -50.4        -34.4
        -29.2        -54.9        -33.5
        -9.0        -52.5        -26.6
        2.3        -52.3        -16.8
        9.2        -47.5        -2.7
        9.6        -42.6        11.5
```

```
-71.2        -53.8        -23.7
-57.0        -61.0        -29.7
-19.9        -64.9        -30.4
-3.7         -63.3        -20.5
7.5         -61.2        -8.1
10.7         -59.0         6.5
9.6         -56.0        20.7
2.7         -56.0        32.2
-6.8         -55.4         42.7
-18.6        -58.0         49.3
-37.1        -59.1         53.4
-53.1        -59.4         53.0
-71.1        -58.4         46.1
-88.3        -53.8         26.6
-90.5        -55.8         8.0
-83.8        -60.8         -6.7
-74.7        -66.0        -19.2
-47.1        -71.2        -30.5
-30.5        -72.5        -30.7
-26.6        -70.0         49.9
6.1         -38.6        24.3
-56.0        -14.2         41.5
-69.4        -17.5         37.4
-2.8         -20.6         -1.7
-77.9        -15.6         23.4
3.7         -72.4        24.6
-8.3         -83.4         29.8
-21.7        -88.0         34.8
-40.3        -92.9         34.3
-56.6        -92.5         28.5
-71.0        -88.0         19.4
-71.9        -87.4         4.5
-59.9        -93.1         -4.7
-47.8        -94.6         -9.9
-33.8        -92.5         -13.9
-20.1        -88.8         -17.1
-7.6         -87.2         -9.7
-1.2         -85.1         2.6
-61.5        -96.0         14.7
-21.9        -97.3         -5.9
-12.0        -95.0         2.6
-4.0         -87.7         18.2
-8.3         -78.2         -19.7
-75.7        -69.5         36.5
-81.8        -71.8         24.3
-84.5        -72.6         11.0
-79.1        -74.3         -4.1
7.6         -73.4        11.2
-38.8        -75.6         48.6
-53.3        -73.0         46.8
-64.2        -82.6         -14.4
-49.7        -85.9         -19.4
-32.5        -83.2         -22.9
-60.8        -81.0         39.7
-16.4        -14.6         -10.1
```

```
        -32.5        -85.1        41.6
        -39.2        -100.9        17.9
        -54.7        -16.9        -20.2
        -44.2        -4.0        -8.2
        -31.5        -0.5        3.0
        -38.5        2.3        13.6
        -38.6        -3.1        32.4
        -32.9        -101.3        2.7
        -60.0        -10.2        -10.3
        -16.3        -5.3        7.9
        -7.8        -14.0        20.9
        -3.3        -69.1        36.9
        -26.5        -98.3        22.5
        -51.2        0.8        6.2
        -40.7        -14.0        -20.4
        -15.9        -76.6        41.2
        -47.5        -100.4        7.3
        -20.6        -7.2        25.7
        -28.3        -11.3        -15.0
        -87.6        -47.6        -3.1
        3.0        -74.5        -6.7
        -71.3        -5.5        15.9
        -60.0        -1.6        22.1
        -76.6        -10.7        4.2
        -64.4        -4.4        1.0
        -28.3        -10.3        36.6
```

3.2 Reading Measurements from the GPS Sensor

Like the 10 DOF sensor, this section uses an Arduino library “TinyGPSPlus.h” (version 1.0.3) and modifies an example from it to read the latitude and longitude from a GPS module (see Figure 3.5). It also uses the Arduino library “SoftwareSerial.h” for serial communication with the GPS module.

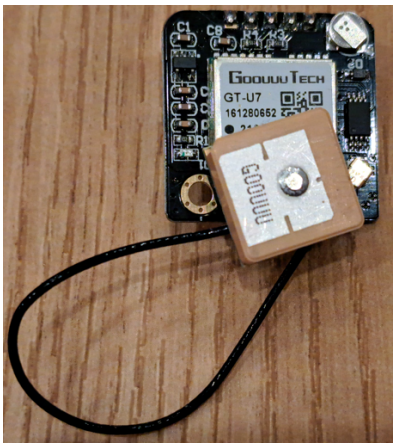


Figure 3.5 A NEO 6M Arduino GPS module by GOOUUU TECH.

```
#include <TinyGPSPlus.h>
#include <SoftwareSerial.h>
/*
This sample sketch demonstrates the normal use of a TinyGPSPlus (TinyGPSPlus) object.
It requires the use of SoftwareSerial, and assumes that you have a
4800-baud serial GPS device hooked up on pins 4(rx) and 3(tx).
*/

//Set the transmission and receive pins
static const int RXPin = 4, TXPin = 3;
static const uint32_t GPSBaud = 9600;

// The TinyGPSPlus object
TinyGPSPlus gps;

// The serial connection to the GPS device
SoftwareSerial ss(RXPin, TXPin);

void setup()
{
        //Start Serial
        Serial.begin(9600);
        //Start communication with the GPS module
        ss.begin(GPSBaud);
}

void loop()
{
        // This sketch displays information every time a new sentence is correctly encoded.
        while (ss.available() > 0)
        if (gps.encode(ss.read())) //if GPS data is read successfully
        displayInfo(); //Print the data

        if (millis() > 5000 && gps.charsProcessed() < 10) //If no GPS signal is available
        {
                //Indicate that the GPS signal is not available
                Serial.println(F("No GPS detected: check wiring."));
                while(true);
        }
}

void displayInfo()
{
        //Get and print the GPS location in latitude and longitude
        Serial.print(F("Location: "));
        if (gps.location.isValid())
        {
                Serial.print(gps.location.lat(), 6);
                Serial.print(F(","));
                Serial.print(gps.location.lng(), 6);
        }
        else
```

```
        {
                Serial.print(F("INVALID"));
        }
        //Print a new line
        Serial.println();
}
```

3.3 Interpreting GPS Data

GPS (Global Positioning System) provides an accurate method for determining position with respect to an inertial reference frame. A GPS sensor, (see Figure 3.6) receives time-stamp information from GPS satellites orbiting the earth (see Figure 3.7). From this information, the GPS sensor's algorithms can determine its geographical location, usually specified in coordinates of latitude, longitude (see Figure 3.8), and altitude above sea level.

Figure 3.6 A GPS sensor by Adafruit (Adafruit Mini GPS PA1010D Module)

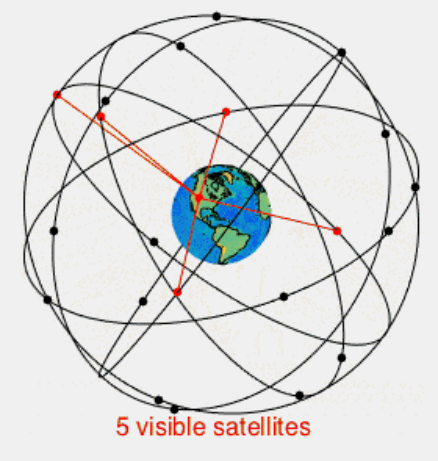


Figure 3.7 An example of a 24-satellite GPS constellation. Five satellites are visible to the GPS sensor. This image from https://en.wikipedia.org/wiki/Global_Positioning_System#/media/File:GPS24goldenSML.gif is licensed under the Creative Commons-Share Alike 4.0 International license.

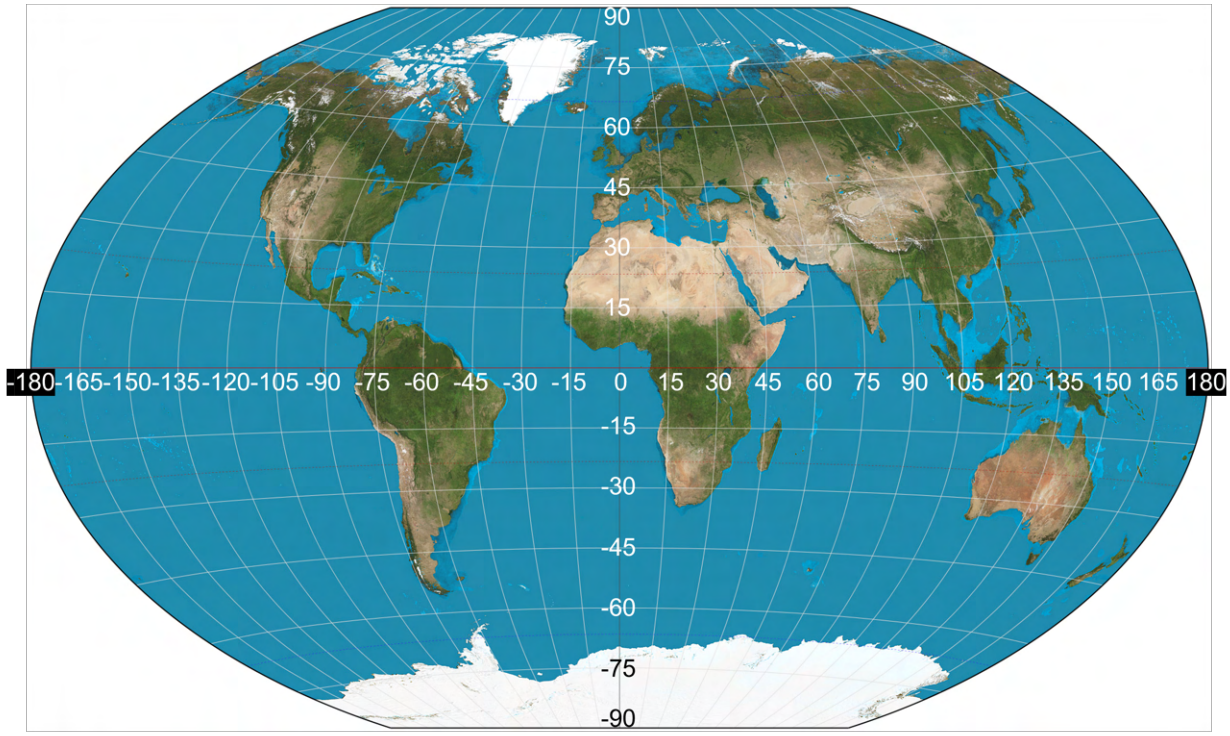


Figure 3.8 A world map with latitude and longitude coordinates. The original map by NASA is in the public domain (accessed June 2022 at https://en.wikipedia.org/wiki/File:Winkel_triple_projection_SW.jpg). The original image was modified to include values for latitude and longitude.

For example, the Adafruit_GPS.h library created for Arduino uses the command lastNMEA(). It returns a string containing an NMEA (National Marine Electronics Association) sentence. To understand the NMEA sentence, consider two NMEA example sentences. Each is for a GPS position located near the statue of a boy throwing an airplane (Boy in Flight) in front of the Mark Austin Building on the BYU-Idaho campus. The first is a GGA (Fix information) sentence for a GPS sensor:

```
$GNGGA,154010.715,4348.9690,N,11147.0400,W,1,3,6.65,1669.0,M,-16.9,M,,*48
```

```
Where:
$            Begin sentence indicator
GN           Multi GNSS solution
GGA          Global Positioning System Fix Data
154010.715   Fix taken at 15:40:10.715 UTC (3:40 p.m.)
4348.9690,N  Latitude 43 deg 48.9690' N
11147.0400,W Longitude 111 deg 47.0400' W (-111 deg 47.0400' E)
1            Fix quality: 0 = invalid
1 = GPS fix (SPS)
```

```
2 = DGPS fix
3 = PPS fix
4 = Real Time Kinematic
5 = Float RTK
6 = estimated (dead reckoning) (2.3 feature)
7 = Manual input mode
8 = Simulation mode
3            Number of satellites being tracked
6.65         Horizontal dilution of position
1669.0,M     Altitude, Meters, above mean sea level
-16.9,M      Height of geoid (mean sea level) above WGS84 ellipsoid
(empty)      Time in seconds since last DGPS update
(empty)      DGPS station ID number
*48          Checksum data, always begins with *
```

The second NMEA example sentence is a RMC (Recommended Minimum data) sentence. It contains position, velocity, and time (but not altitude) information:

```
$GNRMC,154005.715,A,4348.9690,N,11147.0400,W,0.11,21.74,030622,,,A*55
```

```
Where:
$            Begin sentence indicator
GN           Multi GNSS solution
RMC          Recommended Minimum Sentence C
154005.715   Fix taken at 15:40:07.715 UTC (3:40 p.m.)
4348.9690,N  Latitude 43 deg 48.9690' N
11147.0400,W Longitude 111 deg 47.0400' W (-111 deg 47.0400' E)
0.11         Speed over the ground in knots
21.74        Track angle in degrees
030622       Date (3 June 2022)
(empty,empty)Magnetic variation, Direction
*48          Checksum data, always begins with *
```

Before the GPS sensor has a position fix, it may output PMTK commands[1]. These commands communicate that the GPS sensor is powering-on or restarted, or they communicate other settings related to the status of the GPS sensor.

3.3.1 Interpreting Latitude and Longitude

As discussed above, the NMEA format communicates the GPS sensor's position in a specific format. Specifically, the NMEA latitude is given by four digits before the decimal point, and up to seven digits after

[1] see https://www.digikey.com/htmldatasheets/production/1801407/0/0/1/pmtk-command-packet.html, accessed June, 2022

the decimal point: $\underline{d}\,\underline{d}\,\underline{m}\,\underline{m}.\underline{m}\,\underline{m}\,\underline{m}\,\underline{m}\,\underline{m}\,\underline{m}\,\underline{m}$. A letter "N" (North) indicates that the latitude is positive, an "S" indicates a negative latitude. The first two digits ($\underline{d}\,\underline{d}$) indicate the latitude in degrees. The remaining digits ($\underline{m}\,\underline{m}.\underline{m}\,\underline{m}\,\underline{m}\,\underline{m}\,\underline{m}\,\underline{m}\,\underline{m}$) have units of minutes. It can be useful to convert the latitude to decimal degrees. This is done by converting minutes to degrees by dividing by 60 and adding the result to the first two digits. This is shown in the following example.

Example: Convert the NMEA latitude 4348.9690,N to decimal degrees.

Solution: The latitude will be positive, as indicated by "N" (North). The first two digits (43) indicate the latitude in degrees. The remaining digits (48.9690) have units of minutes. We must convert them to decimal degrees by dividing them by 60. Then we can add them to the first two digits (43) to get the latitude in decimal degrees. The conversion from NMEA latitude to decimal degrees is

$$4348.9690, N = 43 + \frac{48.9690}{60} = 43.8162$$

The first two digits were grayed out to show that they already have the correct units and only the remaining digits must be converted. Therefore, the latitude is 43.8162 decimal degrees.

A similar approach is used to convert the NMEA longitude to decimal degrees. The only difference is that there are five digits before the decimal point – the first three are already in decimal degrees. A letter "E" (East) indicates a positive longitude whereas "W" (West) is negative.

Example: Convert the NMEA longitude 11147.0400,W to decimal degrees.

Solution: The longitude will be negative, as indicated by "W" (West). The first three digits (111) indicate the longitude in degrees. The remaining digits (47.0400) have units of minutes. We must convert them to decimal degrees by dividing them by 60. Then, we can add them to the first three digits (111) to get the longitude in decimal degrees. The conversion from NMEA longitude to decimal degrees is

$$11147.0400, W = -\left(111 + \frac{47.0400}{60}\right) = -111.7840$$

The first three digits were grayed out to show that they already have the correct units and only the remaining digits must be converted. Therefore, the longitude is -111.7840 decimal degrees.

3.3.2 Latitude and Longitude to Distance in Meters

Assuming the earth to be spherical, the distance along its surface from one point to another is equal to the arc length. The arc length s of a circular arc of radius r and angle θ in radians is

$$s = r\theta \tag{3.6}$$

Eq. (3.6) requires knowledge of the angle between the two points as measured from the center of the earth. The cosine formula uses the dot-product to find the angle between two vectors $\vec{A}$ and $\vec{B}$:

$$\cos\theta = \frac{\vec{A}\cdot\vec{B}}{\|\vec{A}\|\,\|\vec{B}\|} \tag{3.7}$$

Consider a geocentric coordinate system with its origin at the center of the earth, see Figure 3.9. The x-axis points from the center of the earth through the equator (latitude of 0°) and longitude of 0°. The y-axis points from the center of the earth through the (latitude, longitude) point (0°,90°). The z-axis points from the center of the earth through the north pole at the (latitude, longitude) point (90°,0°).

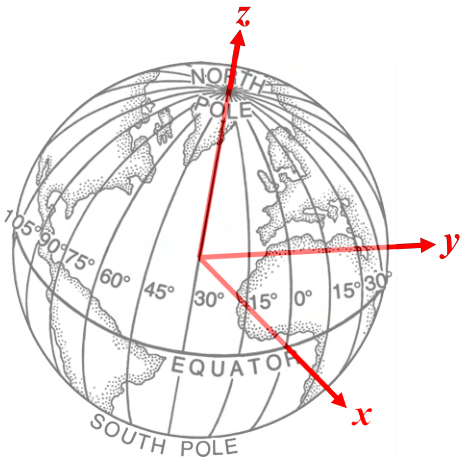


Figure 3.9 Geocentric coordinate system.

An arbitrary point A (see Figure 3.10) on the surface of the earth with latitude ϕ_A and longitude λ_A could be represented in the geocentric coordinate system by the vector

$$\vec{A} = r\left\langle \cos\lambda_A\cos\phi_A \quad \sin\lambda_A\cos\phi_A \quad \sin\phi_A \right\rangle \tag{3.8}$$

where r is the radius of the earth. The radius of the earth at sea level at the equator is 6,378,137 m. However, the earth's radius varies, and the local value should be used for highest accuracy.

Combining Eqs. (3.7) and (3.8), the angle θ between two vectors $\vec{A}$ and $\vec{B}$ pointing from the earth's center to points A and B on the earth's surface is

$$\cos\theta = \frac{r\left\langle \cos\lambda_A\cos\phi_A \quad \sin\lambda_A\cos\phi_A \quad \sin\phi_A \right\rangle \cdot r\left\langle \cos\lambda_B\cos\phi_B \quad \sin\lambda_B\cos\phi_B \quad \sin\phi_B \right\rangle}{r^2} \tag{3.9}$$

where ϕ_B and λ_B are the latitude and longitude respectively of point B. Eq. (3.9) can be simplified to

$$\begin{aligned}\cos\theta &= \left\langle \cos\phi_A \quad 0 \quad \sin\phi_A \right\rangle \cdot \left\langle \cos(\lambda_B-\lambda_A)\cos\phi_B \quad \sin(\lambda_B-\lambda_A)\cos\phi_B \quad \sin\phi_B \right\rangle \\ &= \cos\phi_A\cos\phi_B\cos(\lambda_B-\lambda_A) + \sin\phi_A\sin\phi_B\end{aligned} \tag{3.10}$$

The shortest distance d (see Figure 3.10) from A to B along the surface of the earth can be calculated by Eq. (3.6) as

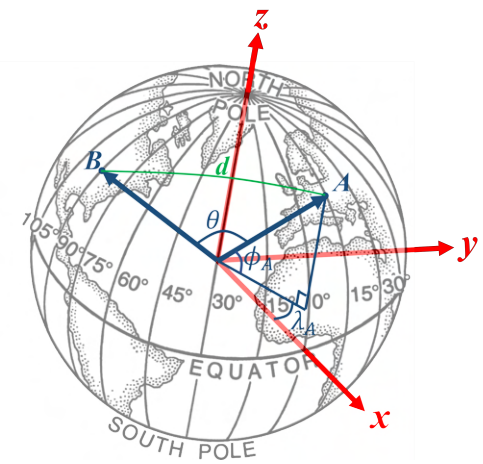


Figure 3.10 Arbitrary points A and B on the earth's surface can be represented by vectors.

$$d = r\cos^{-1}\left(\cos\phi_A\cos\phi_B\cos\left(\lambda_B-\lambda_A\right)+\sin\phi_A\sin\phi_B\right) \tag{3.11}$$

where r is the average earth radius, ϕ_A and λ_A are the latitude and longitude respectively of point A, and ϕ_B and λ_B are the latitude and longitude respectively of point B.

It is often useful to know the relative distance between two points A and B in terms of a two-dimensional grid with North and East as the axes. North is like the x-axis and East is like the y-axis. The distance d_N along the North axis is calculated by the arc-length formula:

$$d_N = r\left(\phi_B-\phi_A\right) \tag{3.12}$$

where the latitudes ϕ_A and ϕ_B must have units of radians[2].

The distance d_N is independent of the longitude. Conversely, the distance d_E along the East axis depends on the latitude; farther from the equator, two points with the same latitude $\phi_A = \phi_B$ but different longitudes $\lambda_A \neq \lambda_B$ are closer together. The distance is derived from Eq. (3.11):

$$d_E = \text{sign}\left(\lambda_B-\lambda_A\right) r\cos^{-1}\left(1+\cos^2\phi_A\left(\cos\left(\lambda_B-\lambda_A\right)-1\right)\right) \tag{3.13}$$

The "sign" operator returns the sign ($\pm$) of its input argument $(\lambda_B - \lambda_A)$.

When distances are small enough that the surface of the earth between the two points A and B can be approximated as flat, the distance d calculated in Eq. (3.11) can be approximated as

$$d \approx \sqrt{d_N^2 + d_E^2} \tag{3.14}$$

When calculating distances between two points on the earth, care must be taken when crossing the $\pm 180°$ longitude. For example, the distance along the equator from $\lambda_A = -170°$ to $\lambda_B = 170°$ longitude would be $-20\frac{\pi}{180}r$ and not $-340\frac{\pi}{180}r$.

To correct for longitudes whose difference results in an angle greater than 180°, the value 360° should first be added to the negative-valued longitude. For example, the distance along the equator from $\lambda_A = -170°$ to $\lambda_B = 170°$ longitude would be $(170-(-170+360))\frac{\pi}{180}r = -20\frac{\pi}{180}r$.

[2] To convert decimal degrees to radians, multiply by $\frac{\pi}{180}$.

3.3.3 Converting Distances to Latitude and Longitude

It may sometimes be beneficial to convert the distances d_N and d_E along north and east axes respectively to changes in latitude and longitude. From Eq. (3.12), the latitude change $\Delta\phi$ in radians is

$$\Delta\phi = \frac{d_N}{r} \tag{3.15}$$

From Eq. (3.13), the longitude change $\Delta\lambda$ in radians is

$$\Delta\lambda = \text{sign}\,(d_E)\cos^{-1}\left(1 + \frac{\cos\left(\frac{d_E}{r}\right) - 1}{\cos^2\phi_A}\right) \tag{3.16}$$

where ϕ_A is the latitude at which the change in eastern position d_E occurred.

Chapter 4

Background for Estimating Orientation

Contents

This chapter discusses how to estimate quaternion orientation using a 9 Degree Of Freedom (9dof) sensor ,*e.g.*, Figure 4.1. The 9dof sensor includes a 3-axis magnetometer, a 3-axis gyrometer (angular rate sensor), and a 3-axis accelerometer (for measuring gravity). Algorithms of this type are sometimes referred to as MARG (Magnetometer, Angular Rate, and Gravity) algorithms or AHRS (Attitude and Heading Reference Systems) filters.

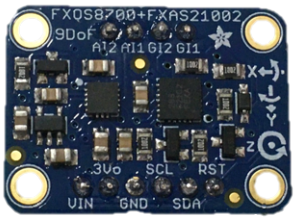


Figure 4.1 A 9dof sensor (FXOS8700 + FXAS21002) by Adafruit

Gyrometer data is typically less noisy than gravity data from an accelerometer. Integrating gyrometer data provides an estimate of orientation, but it is susceptible to long term integration drift. On the other hand, the combination of magnetometer and gravity data provides an independent estimate of orientation that is relatively noisy, but it does not suffer from integration drift. MARG algorithms fuse the information from these three sensors together to obtain a cleaner estimate of orientation that does not drift.

Various algorithms have been derived to estimate orientation. Generally, these algorithms can be categorized into one of three groups: (1) complementary filters, (2) extended or unscented Kalman filters, and (3) error state (indirect) Kalman filters.

Of these three groups, the complementary filters are the easiest to understand, easiest to program, easiest to tune, and require the least resources computationally. They can be robust and provide accurate estimates of orientation. A particularly simple and robust algorithm presented in this chapter is an adaptation of the algorithm developed by Kok and Schön[1].

The extended and unscented Kalman filters are easier to understand than the error state Kalman filters. They can also be easier to tune. The unscented Kalman filters provide an especially accurate, but computationally expensive solution. These algorithms have significantly more tuning parameters than the complementary filters. The unscented Kalman filters are more capable of handling the large nonlinearities of the quaternion state equations than the extended Kalman filters, and are therefore more accurate.

The error state Kalman filters are the most challenging conceptually because they operate on the errors of the dynamic equations rather than the dynamic equations themselves. The main benefit is that the error dynamics are more linear. Error state filters can be less computationally expensive than the unscented Kalman filters. They are the most difficult to tune, however, because they have the largest number of tuning parameters of all the algorithms. For MARG algorithms, the error state Kalman filter requires 12 by 12 process covariance matrices and 6 by 6 sensor covariance matrices that need to be parameterized.

This chapter provides a background for later chapters. The later chapters will explain how to estimate orientation from 9dof sensors. In each algorithm, the gyrometer data is integrated to predict the quaternion orientation. Then gravity and magnetometer data is used to correct the predictions.

4.1 Prediction Step: Quaternion Prediction with a Gyrometer

A gyrometer, *i.e.* gyroscope, is a sensor that measures angular velocities p, q, and r. The variables p, q, and r are the angular velocities about the body-fixed x, y, and z axes respectively. There are at least two ways that a gyrometer can predict quaternion orientation. The first is to numerically integrate the quaternion state-equation which includes the sensed angular velocities. The quaternion state-equation was derived in a previous chapter. It is repeated here for the North-East-Down (NED) coordinate system.

$$\begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{4.1}$$

The second way is similar in that it also numerically integrates the angular velocities p, q, and r over one time-step Δt from t_k to t_{k+1}. Numerical integration produces incremental angle changes:

[1] Manon Kok and Thomas B. Schön, "A Fast and Robust Algorithm for Orientation Estimation using Inertial Sensors", IEEE Signal Processing Letters, 2019. DOI: 10.1109/LSP.2019.2943995

$$\Delta\theta_x = \int_{t_k}^{t_{k+1}} p\,dt \tag{4.2}$$

$$\Delta\theta_y = \int_{t_k}^{t_{k+1}} q\,dt \tag{4.3}$$

$$\Delta\theta_z = \int_{t_k}^{t_{k+1}} r\,dt \tag{4.4}$$

If forward Euler numerical integration is used with a small enough time-step Δt, these can be simplified to the following approximations:

$$\Delta\theta_x \approx p\Delta t \tag{4.5}$$

$$\Delta\theta_y \approx q\Delta t \tag{4.6}$$

$$\Delta\theta_z \approx r\Delta t \tag{4.7}$$

$\Delta\theta_x$, $\Delta\theta_y$, and $\Delta\theta_z$ are the x, y, and z components of the incremental changes in the orientation. The magnitude $\|\Delta\theta\|$ of the incremental change is the square root of the sum of the squares of the components:

$$\|\Delta\theta\| = \sqrt{\Delta\theta_x^2 + \Delta\theta_y^2 + \Delta\theta_z^2} \tag{4.8}$$

A quaternion has both scalar and vector parts. The scalar part is the cosine of half the rotation angle. Therefore, the incremental change in the scalar part of the quaternion is

$$\Delta e_0 = \cos\left(\frac{\|\Delta\theta\|}{2}\right) \tag{4.9}$$

The vector parts are the sine of half the rotation angle multiplied by a unit vector pointing in the same direction as the vector formed by the components of the incremental change in orientation (which are in the same direction as the components of the angular velocities p, q, and r):

$$\Delta e_1 = \frac{\Delta\theta_x}{\|\Delta\theta\|}\sin\left(\frac{\|\Delta\theta\|}{2}\right) \tag{4.10}$$

$$\Delta e_2 = \frac{\Delta\theta_y}{\|\Delta\theta\|}\sin\left(\frac{\|\Delta\theta\|}{2}\right) \tag{4.11}$$

$$\Delta e_3 = \frac{\Delta\theta_z}{\|\Delta\theta\|}\sin\left(\frac{\|\Delta\theta\|}{2}\right) \tag{4.12}$$

The variables Δe_0, Δe_1, Δe_2, and Δe_3 are the four components of the incremental change in the quaternion during the time-step Δt from t_k to t_{k+1}. To predict the new quaternion at time t_{k+1}, we need to add the incremental change. This is done by the quaternion product:

$$\begin{bmatrix} e_{0,k+1} \\ e_{1,k+1} \\ e_{2,k+1} \\ e_{3,k+1} \end{bmatrix} = \begin{bmatrix} e_{0,k} & -e_{1,k} & -e_{2,k} & -e_{3,k} \\ e_{1,k} & e_{0,k} & -e_{3,k} & e_{2,k} \\ e_{2,k} & e_{3,k} & e_{0,k} & -e_{1,k} \\ e_{3,k} & -e_{2,k} & e_{1,k} & e_{0,k} \end{bmatrix} \begin{bmatrix} \Delta e_0 \\ \Delta e_1 \\ \Delta e_2 \\ \Delta e_3 \end{bmatrix} \tag{4.13}$$

The components $e_{0,k+1}$, $e_{1,k+1}$, $e_{2,k+1}$, and $e_{3,k+1}$ are the prediction of the quaternion from the gyrometer.

4.2 Prediction Step: Gyrometer Prediction of the Rotation Matrix

The previous section described how the gyrometer measurement predicts the quaternion. Using the quaternion, the rotation matrix from the body to inertial frame in any coordinate system is

$$R_b^I = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_1e_2-e_0e_3) & 2(e_0e_2+e_1e_3) \\ 2(e_0e_3+e_1e_2) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_2e_3-e_0e_1) \\ 2(e_1e_3-e_0e_2) & 2(e_0e_1+e_2e_3) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \tag{4.14}$$

The rotation matrix from the inertial frame to the body frame is the transpose of R_b^I:

$$R_I^b = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_0e_3+e_1e_2) & 2(e_1e_3-e_0e_2) \\ 2(e_1e_2-e_0e_3) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_0e_1+e_2e_3) \\ 2(e_0e_2+e_1e_3) & 2(e_2e_3-e_0e_1) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \tag{4.15}$$

4.3 Prediction Step: Gyrometer Prediction of Gravity

Using the rotation matrix R_I^b that was presented in the previous section, the gyrometer readings can be used to predict the direction of gravity in the body frame. If g is the acceleration of gravity, the gravitational vector in the inertial NED coordinate system only has a component in the positive z direction:

$$\begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \tag{4.16}$$

In the body frame, the gravity vector is the rotation matrix R_I^b multiplied by the NED gravity vector:

$$\begin{bmatrix} g_x \\ g_y \\ g_z \end{bmatrix} = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_0e_3+e_1e_2) & 2(e_1e_3-e_0e_2) \\ 2(e_1e_2-e_0e_3) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_0e_1+e_2e_3) \\ 2(e_0e_2+e_1e_3) & 2(e_2e_3-e_0e_1) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \tag{4.17}$$

$$= \begin{bmatrix} 2(e_1e_3-e_0e_2)g \\ 2(e_0e_1+e_2e_3)g \\ (e_0^2-e_1^2-e_2^2+e_3^2)g \end{bmatrix} \tag{4.18}$$

If acceleration has units of gravity (g), *i.e.*, g = 1g, the body-frame gravity vector is simply the last column of the inertial to body frame rotation matrix R_I^b.

4.4 Prediction Step: Gyrometer Prediction of Geomagnetic Vector

Unfortunately, even on earth, the geomagnetic vector does not only have a component in the magnetic North direction, it also has a component in the Down direction. The inclination angle δ, *i.e.*, the angle at which the geomagnetic field is tilted with respect the the earth's surface, varies from 90 degrees at the earth's poles to 0 degrees at its magnetic equator. For example, the geomagnetic inclination angle δ in Rexburg, Idaho is approximately 67 degrees. This means that the geomagnetic field in Rexburg points more towards the center of the earth than towards the north, see Figure 4.2. The strength B of the magnetic field in Rexburg is approximately 53 μT.

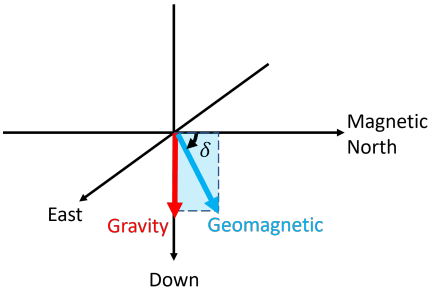


Figure 4.2 Gravitational and geomagnetic vectors in the NED reference frame

The geomagnetic vector is constant in the inertial frame. However, in the body frame, it is the rotation matrix R_I^b multiplied by the inertial geomagnetic vector:

$$\begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_0e_3+e_1e_2) & 2(e_1e_3-e_0e_2) \\ 2(e_1e_2-e_0e_3) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_0e_1+e_2e_3) \\ 2(e_0e_2+e_1e_3) & 2(e_2e_3-e_0e_1) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \begin{bmatrix} B\cos(\delta) \\ 0 \\ B\sin(\delta) \end{bmatrix} \tag{4.19}$$

In Eq. (4.19), M_x, M_y, and M_z are the gyrometer's predictions of the components of the geomagnetic field in the body frame. They have units of μT. A more useful prediction is the normalized unitless geomagnetic components m_x, m_y, and m_z which are predicted by the gyrometer as follows:

$$\begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix} = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_0e_3+e_1e_2) & 2(e_1e_3-e_0e_2) \\ 2(e_1e_2-e_0e_3) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_0e_1+e_2e_3) \\ 2(e_0e_2+e_1e_3) & 2(e_2e_3-e_0e_1) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \begin{bmatrix} \cos(\delta) \\ 0 \\ \sin(\delta) \end{bmatrix} \tag{4.20}$$

The only difference between Eq. (4.20) and Eq. (4.19) is the elimination of the field strength B from the prediction. The variables m_x, m_y, and m_z form the components of a unit vector pointing in the direction of the geomagnetic vector from the perspective of the body frame.

4.5 Measurement Step: Accelerometer Measurement of Gravity

Accelerometers typically measure the acceleration of gravity in addition to other linear accelerations[2]. Extracting the gravitational signals from accelerometer measurements is challenging but essential for orientation estimation. From the perspective of this paper, accelerometers measure gravity as positive and accelerations as negative:

$$\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} -(\dot{u} + qw - rv) + g_x \\ -(\dot{v} + ru - pw) + g_y \\ -(\dot{w} + pv - qu) + g_z \end{bmatrix} \tag{4.21}$$

For example, consider the x-component of the accelerometer signal:

$$a_x = -(\dot{u} + qw - rv) + g_x \tag{4.22}$$

The terms in the parentheses $(\dot{u} + qw - rv)$ form the total acceleration in the x-axis of the body frame. The total acceleration consists of the linear part $\dot{u}$ and the parts caused by the rotation of the body $qw - rv$. In addition to the total acceleration, the accelerometer also includes an x-axis component of gravity g_x. The goal is to extract the gravity component.

To extract the gravity component, the total acceleration $(\dot{u} + qw - rv)$ must be filtered out or removed. Fortunately, the gravity component can be extracted by using a low-pass filter with a sufficiently large time-constant. The low-pass filtered accelerometer signal provides an estimate of the gravity vector from the perspective of the body frame:

$$\begin{bmatrix} a_{x,LP} \\ a_{y,LP} \\ a_{z,LP} \end{bmatrix} \approx \begin{bmatrix} g_x \\ g_y \\ g_z \end{bmatrix} \tag{4.23}$$

In Eq. (4.23), $a_{x,LP}$, $a_{y,LP}$, and $a_{z,LP}$ are the low-pass filtered accelerometer signals in the body-fixed x, y, and z axes respectively. In sensor fusion orientation algorithms, Eq. (4.23) is compared with the gyrometer estimate of gravity Eq. (4.18) for updated estimates of orientation.

To understand why a low-pass filter can extract the gravity component from an accelerometer signal, consider what happens when the accelerometer signal is converted to the inertial frame via the rotation matrix R_b^I from the body to inertial frame:

$$R_b^I \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} -a_{x,I,total} \\ -a_{y,I,total} \\ -a_{z,I,total} + g_{z,I} \end{bmatrix} \tag{4.24}$$

[2] Piezoelectric accelerometers are an exception. They do not measure constant accelerations. However, these accelerometers are not commonly found on most inertial measurement units (IMUs).

Only the inertial z-axis (in a NED reference frame) has a nonzero component of gravity $g_{z,I}$. Gravity is constant, but all the other accelerations are transient. A low-pass filter with a sufficiently long time-constant removes the transient parts of the signals. Therefore only the constant gravitational part remains.

In practice, a low-pass filter with a very large time-constant introduces a delay that causes inaccuracies in orientation algorithms. Therefore, the low-pass filter time-constant becomes an important calibration parameter that requires careful tuning. The transient accelerations are rarely entirely removed from the accelerometer signals, but are only partially attenuated; this introduces error. Fortunately, the magnetometer sensor measurement and gyrometer predictions discussed in other sections partially compensate for the gravity error when used as part of a sensor fusion algorithm. If a more accurate estimate of gravity is needed, Kalman filtering approaches, see Chapter 5, can provide better estimates of gravity.

4.6 Measurement Step: Magnetometer Measurement of Geomagnetic Vector

A magnetometer sensor measures earth's geomagnetic vector. As discussed in Section 4.4, the geomagnetic vector does not only have a component in the magnetic North direction, it also has a component in the Down direction, see Figure 4.2. The inclination angle δ, *i.e.*, the angle at which the geomagnetic field is tilted with respect the the earth's surface, varies from 90 degrees at the earth's poles to 0 degrees at its magnetic equator.

The magnetometer sensor is fixed in the body-frame. It therefore provides a reading of the geomagnetic vector from the perspective of the body-frame:

$$\begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} \tag{4.25}$$

This signal can be normalized by dividing each component by the magnitude, *i.e.*, geomagnetic field strength $B = \sqrt{M_x^2 + M_y^2 + M_z^2}$:

$$u_m = \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix} = \begin{bmatrix} \frac{M_x}{B} \\ \frac{M_y}{B} \\ \frac{M_z}{B} \end{bmatrix} \tag{4.26}$$

The variables m_x, m_y, and m_z form the components of a unit vector that point in the direction of the geomagnetic vector. The magnetometer readings of Eq. (4.26) can be compared with the gyrometer estimate of the geomagnetic unit vector in Eq. (4.20) for an updated estimate of orientation.

Magnetometer signals can be distorted by local magnetic fields, such as a motor coil or inductor, or nearby ferromagnetic materials. These cause "hard iron" and "soft iron" disturbances that bias the magnetometer readings. Magnetometer calibration is required in most cases to correct or minimize the effect of these disturbances (see Section 3.1.1).

4.7 Update Step: Quaternion from Gravity and Geomagnetic Vector

Section 4.5 discussed how to extract gravity from an accelerometer measurement (see Eq. (4.23)), and Section 4.6 discussed accurately measuring the geomagnetic unit vector (see Eqs. (4.26) and (3.5)). A unit vector u_g in the body frame pointing towards the center of the earth (downward) can be found by normalizing the gravity vector Eq. (4.23):

$$u_g = \begin{bmatrix} u_{g,x} \\ u_{g,y} \\ u_{g,z} \end{bmatrix} = \begin{bmatrix} \frac{g_x}{g} \\ \frac{g_y}{g} \\ \frac{g_z}{g} \end{bmatrix} \tag{4.27}$$

where $g = \sqrt{g_x^2 + g_y^2 + g_z^2}$ is the norm of the gravity vector.

The unit vector u_g is in the body frame and points in the positive z_I direction of the inertial frame. Therefore, it forms the last column of the rotation matrix from the inertial to the body frame R_I^b.

The geomagnetic unit vector Eq. (4.26) and gravity unit vector Eq. (4.27) lie in the plane between the positive z_I (Down) and positive x_I (Magnetic North) axes of the inertial frame, see Figure 4.2. Their normalized cross-product ($u_g \otimes u_m$) results in a unit vector u_E in the positive y_I (Magnetic East) direction.

$$u_E = \frac{u_g \otimes u_m}{\left\| u_g \otimes u_m \right\|} \tag{4.28}$$

More explicitly,

$$\begin{bmatrix} u_{E,x} \\ u_{E,y} \\ u_{E,z} \end{bmatrix} = \frac{1}{\sqrt{(u_{g,y}m_z - u_{g,z}m_y)^2 + (u_{g,z}m_x - u_{g,x}m_z)^2 + (u_{g,x}m_y - u_{g,y}m_x)^2}} \begin{bmatrix} u_{g,y}m_z - u_{g,z}m_y \\ u_{g,z}m_x - u_{g,x}m_z \\ u_{g,x}m_y - u_{g,y}m_x \end{bmatrix} \tag{4.29}$$

This unit vector u_E forms the second column of the rotation matrix from the inertial to the body frame R_I^b.

The normalized cross product $u_N = u_E \otimes u_g$ results in a unit vector pointing in the positive x_I (Magnetic North) axis of the inertial frame.

$$u_N = \begin{bmatrix} u_{N,x} \\ u_{N,y} \\ u_{N,z} \end{bmatrix} = \begin{bmatrix} u_{E,y}u_{g,z} - u_{E,z}u_{g,y} \\ u_{E,z}u_{g,x} - u_{E,x}u_{g,z} \\ u_{E,x}u_{g,y} - u_{E,y}u_{g,x} \end{bmatrix} \tag{4.30}$$

This unit vector u_N forms the first column of the rotation matrix from the inertial to the body frame R_I^b. The complete rotation matrix is

$$R_I^b = \begin{bmatrix} u_{N,x} & u_{E,x} & u_{g,x} \\ u_{N,y} & u_{E,y} & u_{g,y} \\ u_{N,z} & u_{E,z} & u_{g,z} \end{bmatrix} \tag{4.31}$$

If we compare Eq. (4.31) with Eq. (4.15), we get the equation

$$\begin{bmatrix} u_{N,x} & u_{E,x} & u_{g,x} \\ u_{N,y} & u_{E,y} & u_{g,y} \\ u_{N,z} & u_{E,z} & u_{g,z} \end{bmatrix} = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_0e_3+e_1e_2) & 2(e_1e_3-e_0e_2) \\ 2(e_1e_2-e_0e_3) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_0e_1+e_2e_3) \\ 2(e_0e_2+e_1e_3) & 2(e_2e_3-e_0e_1) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \tag{4.32}$$

This relationship provides a way to calculate quaternion orientation from a rotation matrix.

4.7.1 Quaternion Orientation from a Rotation Matrix

The goal of this section is to calculate the quaternion orientation e_0, e_1, e_2, and e_3 from a known rotation matrix R. The variables e_0, e_1, e_2, and e_3 are the four parts of a unit quaternion. The first part, e_0 is the scalar part: $e_0 = \cos(\eta)$, where η is twice the rotation angle. The remaining parts e_1, e_2, and e_3 are the vector parts of the quaternion. Let the rotation matrix R from the body to inertial frame be defined as

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{4.33}$$

Then using Eq. (4.15), we get

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_0e_3+e_1e_2) & 2(e_1e_3-e_0e_2) \\ 2(e_1e_2-e_0e_3) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_0e_1+e_2e_3) \\ 2(e_0e_2+e_1e_3) & 2(e_2e_3-e_0e_1) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \tag{4.34}$$

The trace of each matrix produces the equation

$$r_{11}+r_{22}+r_{33} = (e_0^2+e_1^2-e_2^2-e_3^2)+(e_0^2-e_1^2+e_2^2-e_3^2)+(e_0^2-e_1^2-e_2^2+e_3^2) \tag{4.35}$$

$$= 3e_0^2-(e_1^2+e_2^2+e_3^2) \tag{4.36}$$

$$= 4e_0^2-(e_0^2+e_1^2+e_2^2+e_3^2) \tag{4.37}$$

$$= 4e_0^2-1 \tag{4.38}$$

The last equation, Eq. (4.38), results from the fact that the quaternion is a unit quaternion with magnitude equal to one. If $1+r_{11}+r_{22}+r_{33} > 0$, we can find the value of e_0 by solving Eq. (4.38):

$$e_0 = \frac{\sqrt{1+r_{11}+r_{22}+r_{33}}}{2} \tag{4.39}$$

If the sum $1+r_{11}+r_{22}+r_{33}$ is negative, Eq. (4.39) would result in a complex number with imaginary parts. An algorithm presented later will prevent this possibility.

Subtracting r_{32} from r_{23} and dividing by $4e_0$ (from Eq. (4.39)) yields an equation for e_1:

$$e_1 = \frac{r_{23}-r_{32}}{4e_0} \tag{4.40}$$

Similarly, e_2 and e_3 are

$$e_2 = \frac{r_{31} - r_{13}}{4e_0} \tag{4.41}$$

$$e_3 = \frac{r_{12} - r_{21}}{4e_0} \tag{4.42}$$

The problem with this approach is that Eq. (4.39) could result in imaginary numbers. However, a variation of the ideas presented in this section leads to an algorithm for finding the quaternion orientation e_0, e_1, e_2, and e_3 from a known rotation matrix R.

Algorithm: Rotation Matrix to Quaternion

The following MATLAB algorithm calculates the e_0, e_1, e_2, and e_3 from a known inertial to body frame rotation matrix R.

```
function q = function_RI2b_to_Quaternion(R)
% q = function_RI2b_to_Quaternion(R)
% This function calculates the quaternion orientation
% e0, e1, e2, and e3 from a known inertial to body
% rotation matrix R from the inertial to body frame

%calculate the trace of the rotation matrix
t = R(1,1) + R(2,2) + R(3,3);
if t > 0  %the trace is positive, no chance of imaginary numbers
  s = sqrt(t+1)*2; % calculate an intermediate parameter
  e0 = 0.25 * s; %get e0
  e1 = (R(2,3)-R(3,2))/s; %get e1
  e2 = (R(3,1)-R(1,3))/s; %get e2
  e3 = (R(1,2)-R(2,1))/s; %get e3
elseif R(1,1) > max(R(2,2), R(3,3))
  %The trace is negative...Avoid imaginary numbers
  s = sqrt(1+R(1,1)-R(2,2)-R(3,3))*2;
  e0 = (R(2,3)-R(3,2))/s; %get e0
  e1 = 0.25 * s; %get e1
  e2 = (R(1,2)+R(2,1))/s; %get e2
  e3 = (R(1,3)+R(3,1))/s; %get e3
elseif R(2,2) > max(R(1,1), R(3,3))
  %The trace is negative...Avoid imaginary numbers
  s = sqrt(1-R(1,1)+R(2,2)-R(3,3))*2;
  e0 = (R(3,1)-R(1,3))/s; %get e0
  e1 = (R(1,2)+R(2,1))/s; %get e1
  e2 = 0.25 * s; %get e2
  e3 = (R(2,3)+R(3,2))/s; %get e3
else
  %The trace is negative...Avoid imaginary numbers
  s = sqrt(1-R(1,1)-R(2,2)+R(3,3))*2;
  e0 = (R(1,2)-R(2,1))/s; %get e0
  e1 = (R(1,3)+R(3,1))/s; %get e1
  e2 = (R(2,3)+R(3,2))/s; %get e2
  e3 = 0.25 * s; %get e3
end
q = [e0;e1;e2;e3];
```

```
end
```

Chapter 5

Estimating Gravity

Contents

This is an optional chapter because orientation can be estimated with a simpler, less accurate estimate of gravity given in Eq. (4.23). However, this chapter presents an algorithm that can calculate better estimates of the gravity vector. A better gravity signal improves the orientation estimation algorithms presented in other chapters of this book.

5.1 3-Axis Accelerometer Measurements

Section 4.5 discussed the accelerations measured by a 3-axis accelerometer mounted at the center of gravity of a rigid body such as an airplane. The reference standard used in this discussion is that gravitational accelerations are positive and linear accelerations along the same axis are negative. For example, consider an accelerometer with a body-fixed x-axis pointing towards the center of the earth. If the accelerometer is not moving, the signal reads +1g of acceleration. If the accelerometer is dropped, and the x-axis continues to point towards the center of the earth, the accelerometer signal would be 0g. This is because gravitational accelerations are positive and linear accelerations are negative. The linear acceleration of 1g cancels the gravitational acceleration of 1g.

Consider another example. An accelerometer with a body-fixed x-axis pointing towards the center of the earth experiences a linear upward acceleration of 1g away from the earth. That is, the linear acceleration is -1g along its x-axis. Because the body-fixed x-axis points towards the center of the earth, the gravitational acceleration is +1g. Since gravitational accelerations are positive, and linear accelerations are negative, the accelerometer reads a value of 2g.

Eq. (1.15) derived the total linear acceleration of a rotating body, and Eq. (4.21) included gravity components to show what signals are measured by an accelerometer. For convenience, the accelerometer equation is included here:

$$\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} -(\dot{u} + qw - rv) + g_x \\ -(\dot{v} + ru - pw) + g_y \\ -(\dot{w} + pv - qu) + g_z \end{bmatrix} \tag{5.1}$$

Using a 9dof sensor, the following signals are measured:

- p: roll rate (rad/s) is measured by the gyrometer
- q: pitch rate (rad/s) is measured by the gyrometer
- r: yaw rate (rad/s) is measured by the gyrometer
- a_x: body-fixed x-axis accelerometer signal (m/s^2)
- a_y: body-fixed y-axis accelerometer signal (m/s^2)
- a_z: body-fixed z-axis accelerometer signal (m/s^2)

With these measurements, Eq. (5.1) can be written as follows:

$$\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} 0 & r & -q & 1 & 0 & 0 \\ -r & 0 & p & 0 & 1 & 0 \\ q & -p & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \\ g_x \\ g_y \\ g_z \end{bmatrix} - \begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} \tag{5.2}$$

5.1.1 Accelerations as Noise Sources

The linear, body-fixed accelerations, $\dot{u}$, $\dot{v}$, and $\dot{w}$ are not measured with physical sensors. They are dynamic, and outside of aggressive flight, these accelerations dampen out quickly. The following derivation approximates $\dot{u}$, $\dot{v}$, and $\dot{w}$ as random, zero-mean, Gaussian noise sources:

$$\begin{bmatrix} \dot{u} = \nu_x \\ \dot{v} = \nu_y \\ \dot{w} = \nu_z \end{bmatrix} \tag{5.3}$$

The unmeasured time-derivatives of the gravity components are also approximated as zero-mean Gaussian noise sources:

$$\begin{bmatrix} \dot{g}_x = \nu_{g_x} \\ \dot{g}_y = \nu_{g_y} \\ \dot{g}_z = \nu_{g_z} \end{bmatrix} \tag{5.4}$$

Combining Eqs. (5.3) and (5.4) into an array of state-equations and solving them using Euler numerical integration results in discrete-time state-equations. The state-transition equation is

$$\begin{bmatrix} u_{k+1} \\ v_{k+1} \\ w_{k+1} \\ g_{x,k+1} \\ g_{y,k+1} \\ g_{z,k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_k \\ v_k \\ w_k \\ g_{x,k} \\ g_{y,k} \\ g_{z,k} \end{bmatrix} + \Delta t \begin{bmatrix} \nu_{x,k} \\ \nu_{y,k} \\ \nu_{z,k} \\ \nu_{g_x,k} \\ \nu_{g_y,k} \\ \nu_{g_z,k} \end{bmatrix} \tag{5.5}$$

Eq. (5.5) can be written more compactly as

$$x_{k+1} = A_d x_k + \nu_k \tag{5.6}$$

where $x_k = \begin{bmatrix} u_k & v_k & w_k & g_{x,k} & g_{y,k} & g_{z,k} \end{bmatrix}^T$ is the state, the state-transition matrix A_d is the 6×6 identity matrix, and ν_k (which includes the time-step Δt) is a vector of zero-mean Gaussian noise signals with covariance Q.

The output equation is Eq. (5.2) written in discrete-time (hence the subscript k) format:

$$\begin{bmatrix} a_{x,k} \\ a_{y,k} \\ a_{z,k} \end{bmatrix} = \begin{bmatrix} 0 & r_k & -q_k & 1 & 0 & 0 \\ -r_k & 0 & p_k & 0 & 1 & 0 \\ q_k & -p_k & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_k \\ v_k \\ w_k \\ g_{x,k} \\ g_{y,k} \\ g_{z,k} \end{bmatrix} - \begin{bmatrix} \nu_{x,k} \\ \nu_{y,k} \\ \nu_{z,k} \end{bmatrix} \tag{5.7}$$

which can also be written as

$$a_k = C_k x_k + n_k \tag{5.8}$$

where $a_k = \begin{bmatrix} a_{x,k} & a_{y,k} & a_{z,k} \end{bmatrix}^T$ is the accelerometer measurement, x_k is the state, n_k is approximated as zero-mean, Gaussian accelerometer noise having a covariance of R, and the time-varying matrix C_k is

$$C_k = \begin{bmatrix} 0 & r_k & -q_k & 1 & 0 & 0 \\ -r_k & 0 & p_k & 0 & 1 & 0 \\ q_k & -p_k & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.9}$$

5.2 Kalman Filter Estimate of Gravity and Velocity

The discrete-state space form of the accelerometer equations Eqs. (5.6) and (5.8) provides an enabling framework for Kalman filtering. The Kalman filter of this section uses Eqs. (5.6) and (5.8) within a Kalman filter algorithm to estimate the state x_k. The Kalman filtering algorithm consists of the following sequential equations:

$$x_{p,k+1} = A_d x_{u,k} \tag{5.10}$$

$$P_{p,k+1} = A_d P_{u,k} A_d^T + Q \tag{5.11}$$

$$S_k = C_k P_{p,k+1} C_k^T + R \tag{5.12}$$

$$K_k = P_{p,k+1} C_k^T (S_k)^{-1} \tag{5.13}$$

$$x_{u,k+1} = x_{p,k+1} + K_k \left(a_k - C_k x_{p,k+1} \right) \tag{5.14}$$

$$P_{u,k+1} = (I - K_k C_k) P_{p,k+1} \tag{5.15}$$

The intermediate calculation $x_{p,k+1}$ is the model's prediction of the state x_{k+1}. The updated prediction $x_{u,k}$ is the Kalman filter estimate of the state x_k at the k^{th} time-iteration. $P_{p,k}$ is the model's prediction of the state noise covariance. $P_{u,k}$ is the Kalman filter's estimate of the state noise covariance. The matrix S_k is an intermediate step in calculating the Kalman gain K_k. The matrix I is the identity matrix. The other variables were defined in the previous sections of this chapter. It is important to note that the vector a_k is measured using the accelerometer, and the signals p_k, q_k, and r_k are measured using the gyrometer. The Kalman filter state $x_{u,k}$ includes estimates of the velocities u, v, w, and the body-fixed gravity signals g_x, g_y, and g_z.

The Kalman filter requires the process noise covariance matrix Q and accelerometer noise covariance R, both of which are positive definite matrices. These covariance matrices are treated as tuning parameters for the Kalman filter. Approximating the state variables as being independent results in Q being a diagonal matrix. If the accelerometer measurements are independent, R is a diagonal matrix.

5.2.1 Simulation Example

Data collected from a flight simulator was processed using the Kalman filtering algorithm. The resulting estimates of the velocities are shown in Figure 5.1. Figure 5.2 shows the Kalman filter's estimate of the gravity components.

The code that was used to implement the Kalman filter is provided below. The simulated data was imported using a user-defined function 'ImportSimulatorData.m'.

```
%% import the simulator data
ImportSimulatorData
%The data that is imported is
%t: Time (s)
%dt_avg: Average Time Step (s)
%xG:(rad/s) x-axis gyrometer angular velocity
%yG:(rad/s) y-axis gyrometer angular velocity
%zG:(rad/s) z-axis gyrometer angular velocity
%xA:(m/s2) x-axis accelerometer signal
%yA:(m/s2) y-axis accelerometer signal
```

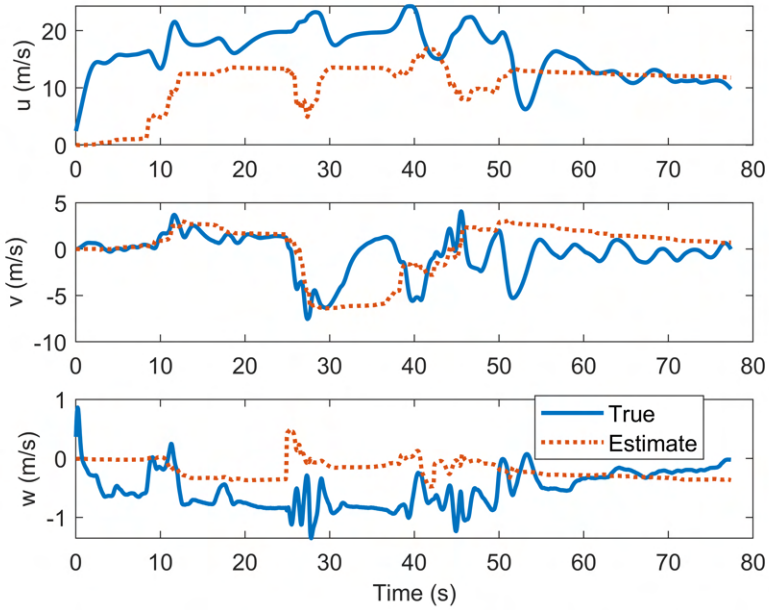


Figure 5.1 Results of the Kalman filter in estimating the body-fixed velocities from a flight simulator. The Kalman filter estimates are labeled 'Estimate'. The simulated velocities are labeled 'True'.

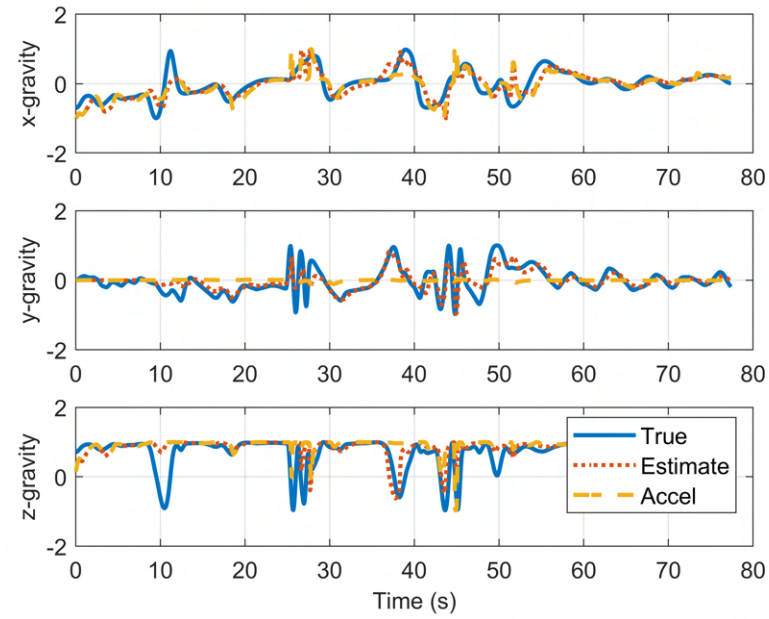


Figure 5.2 Results of the Kalman filter in estimating the body-fixed gravitational accelerations. The Kalman filter estimates are labeled 'Estimate'. The simulator results are labeled 'True'. The signals labeled 'Accel' are estimates of gravity based on Eq. (4.23).

```
%zA:(m/s2) z-axis accelerometer signal
p = xG; %(rad/s) x-axis gyrometer data
q = yG; %(rad/s) y-axis gyrometer data
r = zG; %(rad/s) z-axis gyrometer data
Accel = [xA,yA,zA]'; %(m/s2) 3-axis accelerometer data
u = sim.u_m_s; %(m/s) true forward velocity
v = sim.v_m_s; %(m/s) true side-slip velocity
w = sim.w_m_s; %(m/s) true lift velocity
gx = sim.gx_m_s2/9.8; %(g) true x-component of gravity
gy = sim.gy_m_s2/9.8; %(g) true y-component of gravity
gz = sim.gz_m_s2/9.8; %(g) true y-component of gravity

Ad = eye(6); %State-transition matrix
t = sim.Time_s; %(s) time vector
N = length(t); %Number of time iterations
xu = [0;0;0;0;0;1]; %[u,v,w,gx,gy,gz]
Pu = eye(6); %Initial condition for state noise covariance
Q = [1,0,  0  ,0,0,0;
0,0.5,0  ,0,0,0;
0,0,  0.1,0,0,0;
0,0,  0,  0.5, 0.01,0.01;
0,0,0,    0.01,0.8,0.01;
0,0,0,    0.01,0.01,0.6]; %Process noise covariance matrix
R = diag([0.001;0.001;0.001])*1e-6; %Accelerometer noise covariance matrix
%Allocate memory to store the Kalman filter estimates
ue = zeros(size(t)); %(m/s) estimate of forward velocity
ve = ue; %(m/s) estimate of sideslip velocity
we = ue; %(m/s) estimate of lift velocity
gxe = ue; %(g) estimate of x-component of gravity
gye = ue; %(g) estimate of y-component of gravity
gze = ue; %(g) estimate of z-component of gravity
for ii = 1:N
  %Kalman Filter Algorithm
  xp = Ad*xu;
  Pp = Ad*Pu*Ad' + Q;
  C = [0,r(ii),-q(ii),9.8,0,0;...
       -r(ii),0,p(ii),0,9.8,0;...
       q(ii),-p(ii),0,0,0,9.8];
  S = C*Pp*C' + R;
  Kk = Pp*C'/S;
  xu = xp + Kk*(Accel(:,ii)-C*xp);
  Pu = (eye(6)-Kk*C)*Pp;
  %extract the estimated variables from the state
  norm_g = norm(xu(4:6));
  ue(ii) = xu(1);
  ve(ii) = xu(2);
  we(ii) = xu(3);
  gxe(ii) = xu(4)/norm_g;
  gye(ii) = xu(5)/norm_g;
  gze(ii) = xu(6)/norm_g;
end

%Plot and compare true and estimated velocities
figure
subplot(311)
```

```
plot(t,u,t,ue,':', 'LineWidth', 2)
ylabel('u (m/s)')
subplot(312)
plot(t,v,t,ve,':', 'LineWidth', 2)
ylabel('v (m/s)')
subplot(313)
plot(t,w,t,we,':', 'LineWidth', 2)
ylabel('w (m/s)')
xlabel('Time (s)')
legend('True','Estimate')

%Plot and compare true and estimated gravity signals
norm_accel = zeros(size(Accel(1,:)));
for ii = 1:length(norm_accel)
  norm_accel(ii) = norm(Accel(:,ii));
end

figure
subplot(311)
plot(t,gx,t,gxe,':',t,Accel(1,:)'./norm_accel','--','LineWidth',2)
ylabel('x-gravity')
ylim([-2,2])
grid on
subplot(312)
plot(t,gy,t,gye,':',t,Accel(2,:)'./norm_accel','--','LineWidth',2)
ylabel('y-gravity')
grid on
ylim([-2,2])
subplot(313)
plot(t,gz,t,gze,':',t,Accel(3,:)'./norm_accel','--','LineWidth',2)
ylabel('z-gravity')
ylim([-2,2])
grid on
xlabel('Time (s)')
legend('True','Estimate','Accel')
```

5.2.2 Kalman Filtering Results: Actual Flight Data

The Kalman filtering approach to estimating the gravity signals can help reduce noise caused by propeller vibrations as well. Figure 5.3 compares the filtered gravity estimates from actual airplane data versus estimates calculated by Eq. (4.23). The effect of propeller vibrations is significantly reduced in the filtered estimates.

5.2.3 Discussion on the Kalman Filtering Results

The goal of the Kalman filtering algorithm of this chapter is to calculate an estimate of the gravity vector that is better than the estimate calculated by Eq. (4.23). In the simulation results (see Figure 5.2), although the Kalman filter's estimate of gravity is arguably not always very accurate, it is more accurate than the gravity estimate calculated by Eq. (4.23). This improved estimate of gravity could also improve the estimate of the airplane's orientation, which is critical to autonomous flight.

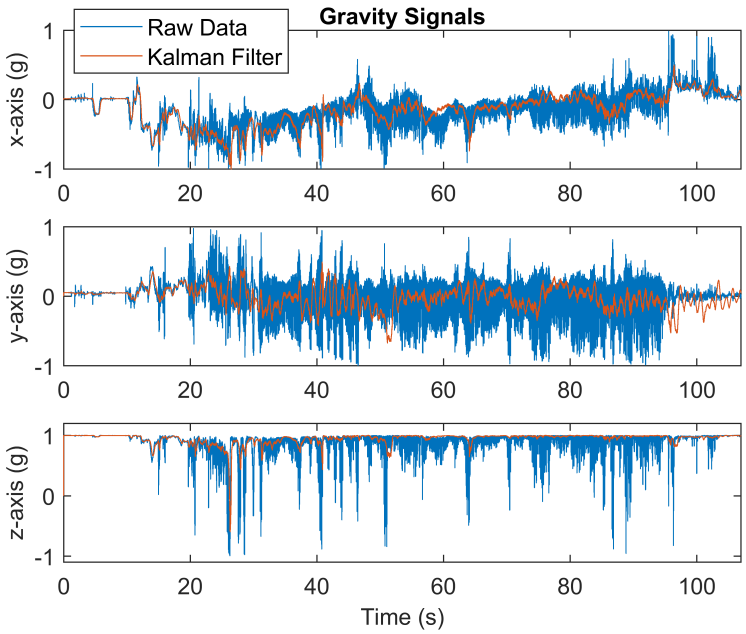


Figure 5.3 The Kalman filter decreases the effect of propeller vibrations in estimating the body-fixed gravitational accelerations. The Kalman filter estimates are labeled 'Kalman Filter'. The signals labeled 'Raw Data' are estimates of gravity based on Eq. (4.23).

In the actual flight, the Kalman filter estimate of gravity also helped remove the effect of propeller vibrations on the gravity estimates. Orientation estimation is the topic of other chapters in this book.

Chapter 6

Algorithm 1: Easy and Accurate Orientation Estimation

Contents

This chapter presents an algorithm for estimating quaternion orientation. Compared to the other algorithms in this book, it is the easiest to understand conceptually. It is also easy in the sense that it has very few (two) tuning parameters. It is accurate and robust because of its ability to filter out gyrometer bias and accelerometer / magnetometer sensor noise. It is more computationally simple than algorithms based on unscented and error state Kalman filters. Although it is the easiest conceptually, it is not the easiest to program on a computer, nor the least computationally complex. The Kok Schön algorithm, presented in Chapter 7, is simpler computationally and requires fewer lines of code with only one tuning parameter.

This chapter uses a complementary filter. As explained in Section 4.1, the gyrometer signal can predict the quaternion orientation. Generally, the gyrometer estimate is relatively noiseless, but it is susceptible to integration drift. Section 4.7 explained how the magnetometer plus accelerometer signals can predict the quaternion orientation. These signals are noisy, but do not suffer from drift. This chapter's algorithm exploits the complementary nature of these different signals to counter the effects of bias drift and sensor noise. As a result, the complementary filter can be relatively accurate and reliable.

6.1 Complementary Filter for Quaternion Estimation

The goal of the complementary filter is to calculate a quaternion estimate $\hat{q}_k$ at the present time t_k given the estimate $\hat{q}_{k-1}$ from the previous time-step t_{k-1} and the 9dof sensor measurements. The 9dof sensor includes a gyrometer, accelerometer, and magnetometer. The gyrometer measures 3-axis angular velocities with units of radians per second (rad/s). The accelerometer measures 3-axis acceleration in units of gravity (g). The magnetometer measures 3-axis magnetic field strength; its units are not important because it is always normalized by dividing each vector component by the magnitude as in Eq. (4.26). This

algorithm assumes that bias has been removed as much as possible from each of the sensors, especially the accelerometer and magnetometer; see Section 3.1.1 for magnetometer calibration.

Applying forward Euler integration to Eq. (4.1), the gyrometer's estimate $\hat{q}_{G,k}$ of the quaternion at the present time t_k is

$$\hat{q}_{G,k} = \hat{q}_{k-1} + \frac{\Delta t}{2} \hat{S}_{k-1} \omega_k \tag{6.1}$$

where

$$\hat{q}_{k-1} = \begin{bmatrix} \hat{e}_{0,k-1} \\ \hat{e}_{1,k-1} \\ \hat{e}_{2,k-1} \\ \hat{e}_{3,k-1} \end{bmatrix} \tag{6.2}$$

is the previous estimate of the quaternion from the complementary filter,

$$\hat{S}_{k-1} = \begin{bmatrix} -\hat{e}_{1,k-1} & -\hat{e}_{2,k-1} & -\hat{e}_{3,k-1} \\ \hat{e}_{0,k-1} & -\hat{e}_{3,k-1} & \hat{e}_{2,k-1} \\ \hat{e}_{3,k-1} & \hat{e}_{0,k-1} & -\hat{e}_{1,k-1} \\ -\hat{e}_{2,k-1} & \hat{e}_{1,k-1} & \hat{e}_{0,k-1} \end{bmatrix} \tag{6.3}$$

and

$$\omega_k = \begin{bmatrix} p_k \\ q_k \\ r_k \end{bmatrix} \tag{6.4}$$

is the 3-axis gyrometer measurement of the angular velocity (rad/s) at the present time t_k.

Quaternion orientations are not unique. However, their prediction of the gravity and geomagnetic unit vectors are unique. Using the gyrometer's estimate of the quaternion $\hat{q}_{G,k}$, the gyrometer's estimate of the gravity unit vector $\hat{g}_{G,k}$ is

$$\hat{g}_{G,k} = \begin{bmatrix} 2(\hat{e}_1\hat{e}_3 - \hat{e}_0\hat{e}_2) \\ 2(\hat{e}_0\hat{e}_1 + \hat{e}_2\hat{e}_3) \\ \hat{e}_0^2 - \hat{e}_1^2 - \hat{e}_2^2 + \hat{e}_3^2 \end{bmatrix} \tag{6.5}$$

which is just the last column of the inertial-to-body rotation matrix R_I^b using the gyrometer's estimate $\hat{q}_{G,k} = \begin{bmatrix} \hat{e}_0 & \hat{e}_1 & \hat{e}_2 & \hat{e}_3 \end{bmatrix}^T$.

The gyrometer's estimate $\hat{m}_{G,k}$ of the geomagnetic unit vector is

$$\hat{m}_{G,k} = \begin{bmatrix} \hat{e}_0^2 + \hat{e}_1^2 - \hat{e}_2^2 - \hat{e}_3^2 & 2(\hat{e}_0\hat{e}_3 + \hat{e}_1\hat{e}_2) & 2(\hat{e}_1\hat{e}_3 - \hat{e}_0\hat{e}_2) \\ 2(\hat{e}_1\hat{e}_2 - \hat{e}_0\hat{e}_3) & \hat{e}_0^2 - \hat{e}_1^2 + \hat{e}_2^2 - \hat{e}_3^2 & 2(\hat{e}_0\hat{e}_1 + \hat{e}_2\hat{e}_3) \\ 2(\hat{e}_0\hat{e}_2 + \hat{e}_1\hat{e}_3) & 2(\hat{e}_2\hat{e}_3 - \hat{e}_0\hat{e}_1) & \hat{e}_0^2 - \hat{e}_1^2 - \hat{e}_2^2 + \hat{e}_3^2 \end{bmatrix} \begin{bmatrix} \cos(\delta) \\ 0 \\ \sin(\delta) \end{bmatrix} \tag{6.6}$$

which is the inertial-to-body rotation matrix R_I^b using the gyrometer's estimate $\hat{g}_{G,k}$ multiplied by the inertial frame geomagnetic unit vector $\begin{bmatrix}\cos(\delta) & 0 & \sin(\delta)\end{bmatrix}^T$. The inclination angle δ is assumed to be known.

6.1.1 The Complementary Filter

The variable $\hat{g}_{G,k}$ is the gyrometer's estimate of the gravity unit vector at the present time t_k. The gyrometer's estimate of the geomagnetic unit vector is $\hat{m}_{G,k}$. These are stacked together into a single column vector to get the measurement prediction $\hat{y}_{G,k}$:

$$\hat{y}_{G,k} = \begin{bmatrix}\hat{g}_{G,k} \\ \hat{m}_{G,k}\end{bmatrix} \tag{6.7}$$

As discussed in Section 4.5, Eq. (4.23) is the accelerometer's estimate of the gravity vector. Normalizing it results in the accelerometer's estimate of the gravity unit vector:

$$\hat{g}_{a,k} = \frac{a_{LP}}{\|a_{LP}\|} \tag{6.8}$$

where $a_{LP} = \begin{bmatrix}a_{x,LP} & a_{y,LP} & a_{z,LP}\end{bmatrix}^T$ is the low-pass filtered accelerometer measurement at the present time step t_k. If a more accurate estimate of gravity is needed, Kalman filtering approaches, see Chapter 5, can provide better estimates of gravity.

The magnetometer's measurement u_m of the geomagnetic unit vector is given by Eq. (4.26), where the magnetometer has been calibrated as explained in Section 3.1.1.

The accelerometer's estimate of gravity $\hat{g}_{a,k}$ and the magnetometer's measurement of the geomagnetic unit vector u_m are stacked into a single column vector to get the measurement vector y_k:

$$y_k = \begin{bmatrix}\hat{g}_{a,k} \\ u_m\end{bmatrix} \tag{6.9}$$

The gyrometer's prediction $\hat{y}_{G,k}$ is generally less noisy than the actual measurement y_k, but it is more prone to drift. The measurement y_k is noisy, but is not susceptible to drift. The complementary filter exploits this nature to determine an improved estimate $\hat{y}_{c,k}$. The complementary filter is

$$\hat{y}_{c,k} = \left(I_6 - \begin{bmatrix}\beta_g I_3 & 0_3 \\ 0_3 & \beta_m I_3\end{bmatrix}\right)\hat{y}_{G,k} + \begin{bmatrix}\beta_g I_3 & 0_3 \\ 0_3 & \beta_m I_3\end{bmatrix} y_k \tag{6.10}$$

where I_3 is the 3×3 identity matrix, I_6 is the 6×6 identity matrix, and 0_3 is a 3×3 matrix of zeros. The scalar variables $\beta_g \in [0,1]$ and $\beta_m \in [0,1]$ are user-define filter coefficients that will be discussed later.

The complementary filter's estimate $\hat{y}_{c,k}$ can be written explicitly in terms of its estimates $\hat{y}_{g,k}$ and $\hat{y}_{m,k}$ of the gravity and geomagnetic unit vectors respectively.

$$\hat{y}_{c,k} = \begin{bmatrix} \hat{y}_{g,k} \\ \hat{y}_{m,k} \end{bmatrix} \tag{6.11}$$

With the estimates $\hat{y}_{g,k}$ and $\hat{y}_{m,k}$ of the gravity and geomagnetic unit vectors respectively, the updated quaternion estimate $\hat{q}_k$ is calculated as described in Section 4.7.

The estimate is forced to be a unit quaternion by normalizing at each iteration of the algorithm as follows:

$$\hat{q}_k = \frac{\hat{q}_k}{\|\hat{q}_k\|} \tag{6.12}$$

At the next iteration, this normalized quaternion is used as the previous estimate $\hat{q}_{k-1}$ in the gyrometer estimate of Eq. (6.1).

6.1.2 The Complementary Filter Coefficients β_g and β_m

The filter coefficient $\beta_j,\ j = g$ or $j = m$ decides whether to trust the gyrometer estimate $\hat{y}_{G,k}$ or accelerometer / magnetometer measurement y_k more. If $\beta_j = 1$, the filter trusts the accelerometer / magnetometer estimate y_k completely while ignoring the gyrometer estimate $\hat{y}_{G,k}$. If $\beta_j = 0$, the filter only trusts the gyrometer estimate $\hat{q}_{G,k}$. For $0 < \beta_j < 1$, the filter partially trusts both estimates.

The filter coefficient $\beta_j,\ j = g$ or $j = m$, is sensitive to the time-step Δt. In applications where various time-steps are possible, it is better to set constant parameters $\tau_g \in [\Delta t, \infty)$ and $\tau_m \in [\Delta t, \infty)$ and calculate the filter coefficients as follows

$$\beta_j = \frac{\Delta t}{\tau_j}, \qquad j = g, m \tag{6.13}$$

Linear accelerations that are not filtered out are simply ignored in the accelerometer's estimate of the gravity vector. This causes the gravity estimate to be noisy. Because the accelerometer noise is usually larger than the magnetometer noise, the value of τ_g is also typically larger than τ_m. For example, $\tau_g \approx 10$s may be appropriate where $\tau_m \approx 1$s is typical.

Chapter 7

Algorithm 2: Accurate and Computationally Simple Orientation Estimation

Contents

This chapter presents the Kok Schön algorithm[1]. It is the computationally simplest orientation algorithm, yet it maintains accuracy. It requires the fewest lines of code, has the least number of tuning parameters, and requires the least amount of computer memory.

Like the algorithm of Chapter 6, the Kok Schön algorithm is a type of complementary filter. It integrates the gyrometer measurements to predict the quaternion orientation. This integration is susceptible to drift, so the noisy but unbiased magnetometer and accelerometer measurements correct the effect of drift. The main difference is that the Kok Schön algorithm cleverly calculates from magnetometer and accelerometer measurements an angular velocity correction instead of an orientation correction. Therefore, the Kok Schön algorithm essentially applies the complementary filter concept to the angular velocity, whereas the previous chapter applied it to the measurements of gravity and geomagnetic vectors.

7.1 The Kok Schön Algorithm

The normalized estimate $\hat{g}_{k-1}$ of gravity in the body frame is

$$\hat{g}_{k-1} = \begin{bmatrix} 2(\hat{e}_1\hat{e}_3 - \hat{e}_0\hat{e}_2) \\ 2(\hat{e}_0\hat{e}_1 + \hat{e}_2\hat{e}_3) \\ \hat{e}_0^2 - \hat{e}_1^2 - \hat{e}_2^2 + \hat{e}_3^2 \end{bmatrix} \tag{7.1}$$

which is just the last column of the inertial-to-body rotation matrix R_I^b using the previous quaternion estimate $\hat{q}_{k-1} = \begin{bmatrix} \hat{e}_0 & \hat{e}_1 & \hat{e}_2 & \hat{e}_3 \end{bmatrix}^T$.

[1] Manon Kok and Thomas B. Schön, "A Fast and Robust Algorithm for Orientation Estimation using Inertial Sensors", IEEE Signal Processing Letters, 2019. DOI: 10.1109/LSP.2019.2943995

The normalized estimate $\hat{m}_{k-1}$ of the geomagnetic vector in the body frame is

$$\hat{m}_{k-1} = \begin{bmatrix} \hat{e}_0^2 + \hat{e}_1^2 - \hat{e}_2^2 - \hat{e}_3^2 & 2(\hat{e}_0\hat{e}_3 + \hat{e}_1\hat{e}_2) & 2(\hat{e}_1\hat{e}_3 - \hat{e}_0\hat{e}_2) \\ 2(\hat{e}_1\hat{e}_2 - \hat{e}_0\hat{e}_3) & \hat{e}_0^2 - \hat{e}_1^2 + \hat{e}_2^2 - \hat{e}_3^2 & 2(\hat{e}_0\hat{e}_1 + \hat{e}_2\hat{e}_3) \\ 2(\hat{e}_0\hat{e}_2 + \hat{e}_1\hat{e}_3) & 2(\hat{e}_2\hat{e}_3 - \hat{e}_0\hat{e}_1) & \hat{e}_0^2 - \hat{e}_1^2 - \hat{e}_2^2 + \hat{e}_3^2 \end{bmatrix} \begin{bmatrix} \cos(\delta) \\ 0 \\ \sin(\delta) \end{bmatrix} \tag{7.2}$$

which is the inertial-to-body rotation matrix R_I^b using the previous quaternion estimate $\hat{q}_{k-1}$ multiplied by the inertial frame geomagnetic unit vector $\begin{bmatrix} \cos(\delta) & 0 & \sin(\delta) \end{bmatrix}^T$. The inclination angle δ is assumed to be known.

As discussed in Section 4.5, Eq. (4.23) is the accelerometer's measurement of the gravity vector. Normalizing it results in the accelerometer's measurement of the gravity unit vector:

$$\hat{g}_{a,k} = \frac{a_{LP}}{\|a_{LP}\|} \tag{7.3}$$

where $a_{LP} = \begin{bmatrix} a_{x,LP} & a_{y,LP} & a_{z,LP} \end{bmatrix}^T$ is the low-pass filtered accelerometer measurement at the present time step t_k. If a more accurate estimate of gravity is needed, Kalman filtering approaches, see Chapter 5, can provide better estimates of gravity.

The magnetometer's measurement m_k of the geomagnetic unit vector is given by Eq. (4.26), where the magnetometer has been calibrated as explained in Section 3.1.1.

The first step in the Kok Schön algorithm is to calculate a steepest descent gradient ∇V in the direction that minimizes the difference between the estimated and measured magnetometer / accelerometer estimates of gravity and geomagnetic vectors. The steepest descent gradient ∇V is[2]

$$\nabla V = \hat{g}_{k-1} \times \left(\hat{g}_{a,k} - \hat{g}_{k-1}\right) + \alpha \hat{m}_{k-1} \times (m_k - \hat{m}_{k-1}) \tag{7.4}$$

where $\times$ is the cross product operator, and α is a positive, real-valued scalar, *e.g.*, $\alpha = 1$.

The second step is to calculate an angular velocity correction $\delta\omega_k$ based on the steepest descent gradient:

$$\delta\omega_k = \beta \frac{\nabla V}{\|\nabla V\|} \tag{7.5}$$

where β is a small, positive, real-valued scalar, *e.g.*, $\beta = 0.1$.

The final step is to apply the correction to the gyrometer's measurement ω_k and numerically integrate to get the next estimate of the quaternion:

$$\hat{q}_k = \hat{q}_{k-1} + \frac{\Delta t}{2} \hat{S}_{k-1} (\omega_k - \delta\omega_k) \tag{7.6}$$

[2]Including the coefficient α is a minor deviation from Kok and Schön's algorithm. A value of $\alpha > 1$ causes the algorithm to trust the magnetometer more than the accelerometer. A value of $0 \leq \alpha < 1$ causes the algorithm to trust the accelerometer more than the magnetometer. Setting the value to $\alpha = 1$ results in Kok and Schön's published algorithm.

where

$$\hat{q}_{k-1} = \begin{bmatrix} \hat{e}_0 \\ \hat{e}_1 \\ \hat{e}_2 \\ \hat{e}_3 \end{bmatrix} \tag{7.7}$$

is the previous estimate of the quaternion from the complementary filter,

$$\hat{S}_{k-1} = \begin{bmatrix} -\hat{e}_1 & -\hat{e}_2 & -\hat{e}_3 \\ \hat{e}_0 & -\hat{e}_3 & \hat{e}_2 \\ \hat{e}_3 & \hat{e}_0 & -\hat{e}_1 \\ -\hat{e}_2 & \hat{e}_1 & \hat{e}_0 \end{bmatrix} \tag{7.8}$$

and

$$\omega_k = \begin{bmatrix} p_k \\ q_k \\ r_k \end{bmatrix} \tag{7.9}$$

is the 3-axis gyrometer measurement of the angular velocity (rad/s) at the present time t_k.

7.2 Background and Derivation of the Kok Schön Algorithm

Eq. (7.6) is the Kok Schön algorithm. It estimates the quaternion orientation. Without the correction term, *i.e.*, if $\delta\omega_k = 0$, the quaternion estimate would entirely depend on integrating the gyrometer's angular velocity measurement ω_k. It would completely disregard the accelerometer and magnetometer signals $\hat{g}_{a,k}$ and m_k respectively. This would result in integration drift.

To address the drift problem, the Kok Schön algorithm uses a correction term $\delta\omega_k$. The correction term represents a small angular velocity step in the direction that minimizes the error between the estimated and measured gravity and geomagnetic vectors. To do so, it uses a steepest-descent optimization approach. The function to be minimized is

$$V = \frac{1}{2} \left\| \hat{g}_{a,k} - \hat{g}_k \right\|^2 + \alpha \frac{1}{2} \| m_k - \hat{m}_k \|^2 \tag{7.10}$$

which can be written as

$$V = \frac{1}{2} \left\| \hat{g}_{a,k} - \hat{R}^b_{I,k} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\|^2 + \alpha \frac{1}{2} \left\| m_k - \hat{R}^b_{I,k} \begin{bmatrix} \cos(\delta) \\ 0 \\ \sin(\delta) \end{bmatrix} \right\|^2 \tag{7.11}$$

where $\hat{R}^b_{I,k}$ is the inertial-to-body rotation matrix using the quaternion estimate $\hat{q}_k$.

The inertial-to-body rotation matrix $\hat{R}^b_{I,k}$ can be written as the product of two rotation matrices:

$$\hat{R}^b_{I,k} = \delta \mathrm{R} \hat{R}^b_{I,k-1} \tag{7.12}$$

where $\hat{R}^b_{I,k-1}$ was the rotation matrix at the previous time-step t_{k-1} and δR is the rotation from the orientation at time t_{k-1} to the present orientation at t_k. The Rodrigues form of any rotation matrix δR is

$$\delta \mathrm{R} = \begin{bmatrix} e_x^2 + (1-e_x^2)\cos(\eta) & e_x e_y(1-\cos(\eta)) + e_z\sin(\eta) & e_x e_z(1-\cos(\eta)) - e_y\sin(\eta) \\ e_x e_y(1-\cos(\eta)) - e_z\sin(\eta) & e_y^2 + (1-e_y^2)\cos(\eta) & e_y e_z(1-\cos(\eta)) + e_x\sin(\eta) \\ e_x e_z(1-\cos(\eta)) + e_y\sin(\eta) & e_y e_z(1-\cos(\eta)) - e_x\sin(\eta) & e_z^2 + (1-e_z^2)\cos(\eta) \end{bmatrix} \tag{7.13}$$

where $\begin{bmatrix} e_x & e_y & e_z \end{bmatrix}^T$ is the normalized axis of rotation, and η is the rotation angle.

Assuming a sufficiently high sampling rate such that the rotation from t_{k-1} to t_k is small, *i.e.*, η is small, we can use the small angle approximation. The small angle approximation of the sine function is $\sin(\eta) \approx \eta$ for η in radians. The small angle approximation of the cosine function is $\cos(\eta) \approx 1$. Therefore, the small angle approximation of the Rodrigues rotation matrix is

$$\delta \mathrm{R} \approx \begin{bmatrix} 1 & e_z\eta & -e_y\eta \\ -e_z\eta & 1 & e_x\eta \\ e_y\eta & -e_x\eta & 1 \end{bmatrix} \tag{7.14}$$

If the sampling rate is sufficiently high such that the angular velocity is constant from t_{k-1} to t_k, then

$$\begin{bmatrix} e_x \\ e_y \\ e_z \end{bmatrix} = \frac{\omega_k}{\|\omega_k\|} \tag{7.15}$$

where $\omega_k = \begin{bmatrix} p_k & q_k & r_k \end{bmatrix}^T$ is the angular velocity. Also, the rotation angle η is

$$\eta = \|\omega_k\| \Delta t \tag{7.16}$$

where $\Delta t = t_k - t_{k-1}$ is the time-step. Therefore, the Rodrigues rotation matrix becomes

$$\delta \mathrm{R} \approx \begin{bmatrix} 1 & r_k\Delta t & -q_k\Delta t \\ -r_k\Delta t & 1 & p_k\Delta t \\ q_k\Delta t & -p_k\Delta t & 1 \end{bmatrix} \tag{7.17}$$

and the function to be minimized is

$$\begin{aligned} V = &\frac{1}{2}\left\| \hat{g}_{a,k} - \begin{bmatrix} 1 & r_k\Delta t & -q_k\Delta t \\ -r_k\Delta t & 1 & p_k\Delta t \\ q_k\Delta t & -p_k\Delta t & 1 \end{bmatrix} \hat{g}_{k-1} \right\|^2 \\ &+ \alpha\frac{1}{2}\left\| m_k - \begin{bmatrix} 1 & r_k\Delta t & -q_k\Delta t \\ -r_k\Delta t & 1 & p_k\Delta t \\ q_k\Delta t & -p_k\Delta t & 1 \end{bmatrix} \hat{m}_{k-1} \right\|^2 \end{aligned} \tag{7.18}$$

where

$$\hat{g}_{k-1} = \hat{R}^b_{I,k-1} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{7.19}$$

is the previous estimate of the normalized gravity vector and

$$\hat{m}_{k-1} = \hat{R}^b_{I,k-1} \begin{bmatrix} \cos(\delta) \\ 0 \\ \sin(\delta) \end{bmatrix} \tag{7.20}$$

is the previous estimate of the normalized geomagnetic vector.

The angular velocity correction term $\delta\omega_k$ is a vector that points in the steepest descent direction of the function V of Eq. (7.18). The gradient ∇V also points in the direction of steepest descent. It is found by taking partial derivatives of V with respect to ω_k to be

$$\nabla V = \hat{g}_{k-1} \times \left(\hat{g}_{a,k} - \hat{g}_{k-1}\right) + \alpha \hat{m}_{k-1} \times (m_k - \hat{m}_{k-1}) \tag{7.21}$$

Therefore, the correction term $\delta\omega_k$ is the normalized gradient multiplied by a small positive step-size β:

$$\delta\omega_k = \beta \frac{\nabla V}{\|\nabla V\|} \tag{7.22}$$

The Kok Schön algorithm, Eq. (7.6), is the result of correcting the angular velocity ω_k by the correction term $\delta\omega_k$ and integrating[3] to get the quaternion orientation.

[3] see Basile Graf, "Quaternions and Dynamics", 18 Nov. 2008, Cornell University, Online (accessed March 2021): https://arxiv.org/pdf/0811.2889.pdf

Chapter 8

GPS Sensor Fusion with Orientation

Contents

This chapter describes a sensor fusion algorithm for position and orientation that combines GPS and altitude pressure signals with inertial measurements – acceleration, angular velocity, and geomagnetic strength. Although many different sensor fusion algorithms are possible, such as unscented Kalman filters, extended Kalman filters, error-state Kalman filters, and many others, this chapter uses the computationally simplest method: a complementary filter. It builds on the Kok Schön orientation estimator by adding inertial-frame position and linear velocity estimation.

8.1 "Holoptic" Sensor Fusion Algorithm

This section presents what it calls the Holoptic sensor fusion algorithm. The word "holoptic" refers to how the compound eyes of dragonflies and some other insects meet at the top of their heads. An insect with compound eyes fuses the information from multiple eyes together to make decisions. The Holoptic algorithm fuses information from multiple sensors together to get a better overall estimation of the airplane's orientation and inertial position.

In some flight monitoring systems, the GPS and altitude sensors are sampled at a rate different from the accelerometer, magnetometer, and gyrometer sensors. The following Holoptic sensor fusion algorithm applies to separate sampling rates. The time-step Δt (s) corresponds to the rate at which the accelerometer, magnetometer, and gyrometer sensors are sampled. It is also the update rate of the microcontroller. The time-step Δt_{GPS} (s) is the sampling interval of the GPS and altitude sensors. It is slower rate, *i.e.*, $\Delta t_{\text{GPS}} \geq \Delta t$.

Holoptic Sensor Fusion Algorithm for Separately Sampled Sensors
In this algorithm, the accelerometer, gyrometer, and magnetometer data are sampled at a time-step of Δt (s). The GPS and altitude pressure sensors are sampled separately at a time-step of Δt_{GPS}.

If new GPS information is available, latitude ϕ (rad), longitude λ (rad), and altitude h (m) are first

converted to inertial North-East-Down positions x_I, y_I, and z_I (m) respectively:

$$
\begin{aligned}
&\text{if (new GPS information)}\\
&\quad x_I = r\left(\phi - \phi_0\right)\\
&\quad \text{if (} |\lambda - \lambda_0| <= \pi \text{)}\\
&\qquad \Delta\lambda = \lambda - \lambda_0\\
&\quad \text{else if (} |\lambda - \lambda_0| > \pi\text{)}\\
&\qquad \text{if (} \lambda > \lambda_0 \text{)}\\
&\qquad\quad \Delta\lambda = \lambda - (\lambda_0 + 2\pi)\\
&\qquad \text{else}\\
&\qquad\quad \Delta\lambda = (2\pi + \lambda) - \lambda_0\\
&\qquad \text{end}\\
&\quad \text{end}\\
&\quad y_I = \text{sign}\left(\Delta\lambda\right) r \cos^{-1}\left(\left(\cos\left(\Delta\lambda\right) - 1\right)\cos^2\phi + 1\right)\\
&\quad z_I = -(h - h_0)\\
&\quad X_{I,k} = \begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix}\\
&\text{end}
\end{aligned}
\tag{8.1}
$$

The constants ϕ_0 (rad), λ_0 (rad), and h_0 (m) are the initial or origin GPS latitude, longitude, and altitude. The earth's radius r (m) is approximately 6,371,000 m, but a more accurate local value can be used if necessary. The subscript k indicates that $X_{I,k}$ (m) is an array of the present positions.

In addition, when new GPS information is available, the GPS linear velocity vector V_{GPS} (m/s) and it's low-pass filtered version V_I (m/s), both in the inertial frame, are calculated as

$$
\begin{aligned}
&\text{if (new GPS information)}\\
&\quad V_{GPS} = \frac{X_{I,k} - X_{I,k-1}}{\Delta t_{GPS}}\\
&\quad V_{I,k} = \left(1 - \gamma\right) V_{I,k-1} + \gamma V_{GPS}\\
&\text{else}\\
&\quad V_{I,k} = V_{I,k-1}\\
&\text{end}
\end{aligned}
\tag{8.2}
$$

where $X_{I,k-1}$ (m) and $V_{I,k-1}$ (m/s) are the vectors of the positions and velocities from the previous time at which GPS was available, *i.e.*, one time-step Δt_{GPS} ago. The constant $0 \le \gamma \le 1$ is a user-defined tuning parameter. Values close to $\gamma = 0.9$ are typical when the GPS signal is reliable.
Regardless of whether or not new GPS information is available, the inertial-to-body-frame rotation matrix $\hat{R}_I^b$ is estimated using the Holoptic algorithm's quaternion estimate $\hat{q}_k = \begin{bmatrix} e_0 & e_1 & e_2 & e_3 \end{bmatrix}^T$.

$$\hat{R}_I^b = \begin{bmatrix} e_0^2+e_1^2-e_2^2-e_3^2 & 2(e_0e_3+e_1e_2) & 2(e_1e_3-e_0e_2) \\ 2(e_1e_2-e_0e_3) & e_0^2-e_1^2+e_2^2-e_3^2 & 2(e_0e_1+e_2e_3) \\ 2(e_0e_2+e_1e_3) & 2(e_2e_3-e_0e_1) & e_0^2-e_1^2-e_2^2+e_3^2 \end{bmatrix} \tag{8.3}$$

Using the rotation matrix, the estimated linear velocity vector $\hat{V}_k$ (m/s) in the body-frame is

$$\hat{V}_k = \hat{R}_I^b V_{I,k} \tag{8.4}$$

Except in the case of severe wind or extreme acrobatic maneuvers, the forward velocity of the fixed-wing airplane should account for most of the airplane's speed, *i.e.* the first element of the vector $\hat{V}_k$ should be approximately equal to the total speed: $\hat{V}_k(1) \approx \left\|\hat{V}_k\right\|$. If not, it is an indication that the Holoptic algorithm's estimate of the quaternion $\hat{q}_k$ (and therefore the rotation matrix $\hat{R}_I^b$) is inaccurate. The Holoptic algorithm corrects for this case as follows:

$$\begin{gathered} \text{if } (\hat{V}_k(1) < f\left\|\hat{V}_k\right\|) \\ \hat{V}_k = \begin{bmatrix} \left\|\hat{V}_k\right\| \\ 0 \\ 0 \end{bmatrix} \\ \text{end} \end{gathered} \tag{8.5}$$

where the fraction $0 < f < 1$ is a user-defined constant. Typical values of f are in the range [0.6,0.8].

The accelerometer's prediction of the normalized gravity vector $g_{a,k}$ (unitless) in the body-frame is

$$g_{a,k} = \frac{a_k + \omega_k \times \hat{V}_k}{\left\|a_k + \omega_k \times \hat{V}_k\right\|} \tag{8.6}$$

where a_k (m/s^2) is the gravity-positive accelerometer measurement, ω_k (rad/s) is the calibrated gyrometer measurement, and $\hat{V}_k$ (m/s) is the Holoptic estimate of linear velocity from Eq. (8.4). If a more accurate estimate of gravity is needed, Kalman filtering approaches, see Chapter 5, can be combined with these concepts to provide better estimates of gravity.

The Holoptic algorithm also calculates an estimate $\hat{g}_k$ (unitless) of the gravity unit vector in the body-frame. It is the third column of the inertial-to-body rotation matrix $\hat{R}_I^b$:

$$\hat{g}_k = \hat{R}_I^b \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{8.7}$$

Similarly, the Holoptic algorithm calculates an estimate $\hat{m}_k$ (unitless) of the geomagnetic unit vector:

$$\hat{m}_k = \hat{R}_I^b \begin{bmatrix} \cos\delta \\ 0 \\ \sin\delta \end{bmatrix} \tag{8.8}$$

where δ (rad) is the local geomagnetic inclination angle. It is approximately 67° in Rexburg, Idaho, but it changes with location on the earth. The Kok Schön gradient ΔJ is the steepest-descent direction to minimize the difference between the Holoptic estimates $\hat{g}_k$ and $\hat{m}_k$ and the measurements $g_{a,k}$ and m_k.

$$\nabla J = \hat{g}_k \times \left(g_{a,k} - \hat{g}_k\right) + \alpha \hat{m}_k \times (m_k - \hat{m}_k) \tag{8.9}$$

where m_k is the calibrated and normalized magnetometer signal in the body-frame, and $\alpha \in [0, \infty)$ is a weighting constant to trust or distrust the magnetometer signal more. The original Kok Schön algorithm uses $\alpha = 1$. When the magnetometer signal is reliable, any value of $\alpha \in [1, 2]$ works well. The gradient ∇J is used in the angular velocity update vector $\delta\omega$ (rad/s):

$$\delta\omega = \beta \frac{\nabla J}{\|\nabla J\|} \tag{8.10}$$

where $\beta \in [0, 1]$ is a user-defined tuning parameter. Typical values are $\beta \in [0.05, 0.3]$. However, β can vary if desired. For example, the following condition has been used to help the algorithm converge quickly and then accurately track an airplane's orientation:

$$\begin{aligned} &\text{if } (\hat{V}_k(1) < f \left\| \hat{V}_k \right\|) \\ &\qquad \beta = 0.2 \\ &\text{else} \\ &\qquad \beta = 0.05 \\ &\text{end} \end{aligned} \tag{8.11}$$

The Holoptic update of the quaternion estimate is calculated as follows:

$$\hat{q}_{k+1} = \hat{q}_k + \frac{\Delta t}{2} \hat{S}_k \left(\omega_k - \delta\omega\right) \tag{8.12}$$

where ω_k is the gyrometer's measurement (rad/s) of the angular velocities. To be a unit quaternion, $\hat{q}_{k+1}$ must be normalized at each iteration.

$$\hat{q}_{k+1} = \frac{\hat{q}_{k+1}}{\left\| \hat{q}_{k+1} \right\|} \tag{8.13}$$

The matrix $\hat{S}_k$ is

$$\hat{S}_k = \begin{bmatrix} -\hat{e}_1 & -\hat{e}_2 & -\hat{e}_3 \\ \hat{e}_0 & -\hat{e}_3 & \hat{e}_2 \\ \hat{e}_3 & \hat{e}_0 & -\hat{e}_1 \\ -\hat{e}_2 & \hat{e}_1 & \hat{e}_0 \end{bmatrix} \tag{8.14}$$

The estimates $\hat{e}_0, \hat{e}_1, \hat{e}_2, \hat{e}_3$ are the components of the quaternion estimate $\hat{q}_k$.

The Holoptic sensor fusion algorithm calculates the quaternion orientation $\hat{q}_{k+1}$, the body-frame linear velocities $\hat{V}_k$, the inertial-frame linear velocities $V_{I,k}$, and the inertial-frame positions $X_{I,k}$. It

requires four tuning parameters: α, β, γ, and f. It requires initial conditions for the following arrays: $\hat{q}_k$, $X_{I,k-1}$, $V_{I,k-1}$, ϕ_0, λ_0, h_0. It fuses together sensor information from the following signals: ω_k, a_k, m_k, ϕ_k, λ_k, and h_k. It requires a flag to indicate when new GPS information is available. The local geomagnetic inclination angle δ,the algorithm time-step Δt, and the GPS time-step Δt_{GPS} must be known.

8.2 Possible Additional Improvements

If the GPS data is sufficiently fast and reliable, GPS data can help provide additional improved estimates of the airplane's orientation. It does so by providing a unit vector in the direction of the airplane's heading. The heading vector $h_{I,k}$ in the inertial frame is the normalized unit vector:

$$h_{I,k} = \frac{X_{I,k} - X_{I,k-1}}{\left\| X_{I,k} - X_{I,k-1} \right\|} \tag{8.15}$$

The gyrometer / Holoptic algorithm also predicts the heading unit vector. The prediction $\hat{h}_{I,k}$ is

$$\begin{aligned} \hat{h}_{I,k} &= \hat{R}_b^I \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 \\ 2(e_0 e_3 + e_1 e_2) \\ 2(e_1 e_3 - e_0 e_2) \end{bmatrix} \end{aligned} \tag{8.16}$$

This estimate of the heading unit vector ignores side-slip and angle-of-attack effects. Including the heading difference $\left(h_{I,k} - \hat{h}_{I,k}\right)$ in the steepest descent minimization changes the Kok Schön gradient as follows:

$$\nabla J = \hat{g}_k \times \left(g_{a,k} - \hat{g}_k\right) + \alpha_1 \hat{m}_k \times (m_k - \hat{m}_k) + \alpha_2 \hat{h}_{I,k} \times \left(h_{I,k} - \hat{h}_{I,k}\right) \tag{8.17}$$

All other steps in the Holoptic algorithm are the same as in Section 8.1. The scalar constants α_1 and α_2 are tuning parameters. This possible improvement ignores wind. When flying in windy conditions, it could degrade the estimate of orientation rather than improve it.

Chapter 9

Autonomous Flight

Contents

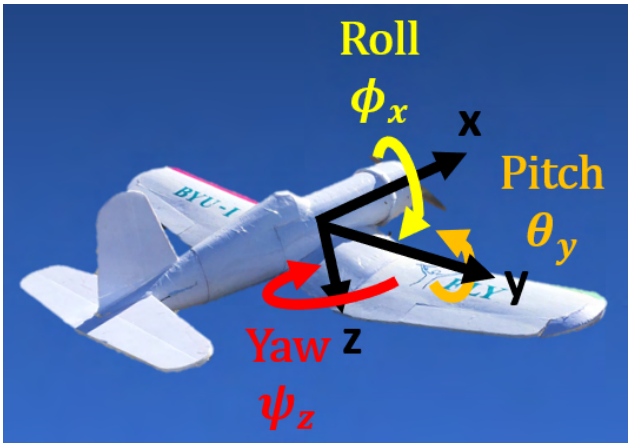


Figure 9.1 Roll, pitch, and yaw angles.

This chapter describes a strategy for autonomous control of roll, pitch, and yaw dynamics. It uses successive loop closure techniques with PID (Proportional-Integral-Derivative) error compensation. The derivative control is applied directly to gyro measurements of angular velocities. PID control is applied to roll, pitch, and yaw errors.

9.1 Roll, Pitch, and Yaw Euler Angles

Roll (ϕ_x), pitch (θ_y), and yaw (ψ_z) angles can describe the orientation of the aircraft, see Figure 9.1. One way to describe roll, pitch, and yaw is in terms of Euler angles. Another method based on a quaternion perspective is described in Section 9.4. Using Euler angles decouples roll, pitch, and yaw, making control simpler.

Euler angles depend on the order in which they are applied. The most intuitive sequence for aerospace applications is yaw – pitch – roll, see Figure 9.2. Yawing first means to rotate to the appropriate x-y heading angle, then pitch to the angle above or below the horizontal plane, then roll around the airplane's body-fixed x-axis. The derivations in this section use this sequence.

Figure 9.2 The yaw-pitch-roll Euler sequence. The airplane in the image demonstrates yaw −90°, then pitch 30°, and finally roll 45°.

9.1.1 Rotations Using Euler Angles

A yaw-rotation of ψ_z from one orientation in the inertial frame to a new orientation can be described mathematically using a rotation matrix. Conversion from the inertial frame (x_I, y_I, z_I) to the body frame (x, y, z) is calculated by the following equation:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos\psi_z & -\sin\psi_z & 0 \\ \sin\psi_z & \cos\psi_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} \tag{9.1}$$

If the plane undergoes a pitch rotation θ_y in addition to yaw ψ_z, the conversion from the inertial frame (x_I, y_I, z_I) to the body frame (x, y, z) is

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \begin{bmatrix} \cos\psi_z & -\sin\psi_z & 0 \\ \sin\psi_z & \cos\psi_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} \tag{9.2}$$

The order of the matrices from left to right is opposite of the sequence in which the rotations occur.

Finally, if a roll rotation of ϕ_x is applied after pitch and yaw, the conversion is

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi_x & -\sin\phi_x \\ 0 & \sin\phi_x & \cos\phi_x \end{bmatrix} \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \begin{bmatrix} \cos\psi_z & -\sin\psi_z & 0 \\ \sin\psi_z & \cos\psi_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} \tag{9.3}$$

Combining the rotation matrices, Eq. (9.3) becomes

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} c\theta_y c\psi_z & -c\theta_y s\psi_z & s\theta_y \\ c\phi_x s\psi_z + s\phi_x s\theta_y c\psi_z & c\phi_x c\psi_z - s\phi_x s\theta_y s\psi_z & -s\phi_x c\theta_y \\ s\phi_x s\psi_z - c\phi_x s\theta_y c\psi_z & s\phi_x c\psi_z + c\phi_x s\theta_y s\psi_z & c\phi_x c\theta_y \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} \tag{9.4}$$

where c indicates the cosine and s indicates the sine. The equivalent quaternion rotation is

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0 e_3 + e_1 e_2) & 2(e_1 e_3 - e_0 e_2) \\ 2(e_1 e_2 - e_0 e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0 e_1 + e_2 e_3) \\ 2(e_0 e_2 + e_1 e_3) & 2(e_2 e_3 - e_0 e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} \tag{9.5}$$

Comparing Eqs. (9.4) and (9.5), we can derive conversions from quaternions to Euler angles and vice-versa. The equations that convert from quaternion components to Euler angles are

$$\phi_x = \text{atan2}\left(2\left(e_0 e_1 + e_2 e_3\right),\, e_0^2 - e_1^2 - e_2^2 + e_3^2\right) \tag{9.6}$$

$$\theta_y = \sin^{-1}\left(2(e_0 e_2 - e_1 e_3)\right) \tag{9.7}$$

$$\psi_z = \text{atan2}\left(2(e_0 e_3 + e_1 e_2),\, e_0^2 + e_1^2 - e_2^2 - e_3^2\right) \tag{9.8}$$

The relationship for ϕ_x is derived by dividing the second element by the third element in the third column of each matrix. Similarly, the relationship for ψ_z is derived by dividing the second element by the first element in the first row of each matrix. The relationship for θ_y is derived by comparing the third element in the first row of each matrix. Converting back, we get

$$e_0 = \cos\phi_x \cos\theta_y \cos\psi_z + \sin\phi_x \sin\theta_y \sin\psi_z \tag{9.9}$$

$$e_1 = \sin\phi_x \cos\theta_y \cos\psi_z - \cos\phi_x \sin\theta_y \sin\psi_z \tag{9.10}$$

$$e_2 = \cos\phi_x \sin\theta_y \cos\psi_z + \sin\phi_x \cos\theta_y \sin\psi_z \tag{9.11}$$

$$e_3 = \cos\phi_x \cos\theta_y \sin\psi_z - \sin\phi_x \sin\theta_y \cos\psi_z \tag{9.12}$$

The conversion back to quaternions is not derived in this document.

9.2 Autonomous Control of Roll, Pitch, and Yaw

The previous section provided a way to calculate the roll ϕ_x, pitch θ_y, and yaw ψ_z Euler angles. These signals can be used for feedback control. The feedback controllers also use the gyrometer measurements of roll rate ω_x, pitch rate ω_y, and yaw rate ω_z as feedback signals. This section will design autonomous strategies to control the roll, pitch, and yaw angles of the airplane.

Calculating Differences in Angles

Calculating the difference or error between two angles α_1 and α_2 is not always as simple as subtracting one from the other: $\alpha_2 - \alpha_1$. For example, on the unit circle, the difference between the angles $\alpha_1 = 179°$ and $\alpha_2 = -179°$ is $\Delta\alpha = 2°$, but subtracting them results in a different value: $\alpha_2 - \alpha_1 = -179° - 179° = -358°$, see Figure 9.3.

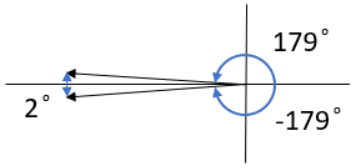


Figure 9.3 The difference is $\Delta\alpha = 2°$ but subtracting them produces $\alpha_2 - \alpha_1 = -179° - 179° = -358°$

The absolute shortest angle $|\delta\alpha|$ between α_1 and α_2 can be calculated by the cosine formula:

$$|\delta\alpha| = \cos^{-1}(\cos\alpha_1 \cos\alpha_2 + \sin\alpha_1 \sin\alpha_2) \tag{9.13}$$

The sign of the angle difference is also important. To get the signed difference $\Delta\alpha$ between α_1 and α_2, we use the following logic:

$$\begin{array}{l} |\delta\alpha| = \cos^{-1}(\cos\alpha_1 \cos\alpha_2 + \sin\alpha_1 \sin\alpha_2) \\ \text{if } |\alpha_2 - \alpha_1| - |\delta\alpha| > 0.1 \\ \qquad \Delta\alpha = -\text{sign}(\alpha_2 - \alpha_1)\,|\delta\alpha| \\ \text{else} \\ \qquad \Delta\alpha = \alpha_2 - \alpha_1 \\ \text{end} \end{array} \tag{9.14}$$

9.2.1 Autonomous Flight Control

The autopilot strategy for autonomous flight depends on whether the plane has a rudder or not. A rudder facilitates yaw control, but rudderless yaw control works whether the plane has a rudder or not, assuming it has ailerons and an elevator (or elevons, as with a flying wing).

Rudderless Control

Figure 9.4 shows a block diagram of a rudderless autonomous control strategy. The subtraction blocks containing stars indicate that the difference is calculated using the smallest angle formula of Eq. (9.14). The quaternion orientation $\hat{q}$ is calculated using the Holoptic sensor fusion algorithm of Chapter 8. The angular roll velocity ω_x is measured using a gyrometer. The desired angles $\psi_{z,d}$, $\theta_{y,d}$, and $\phi_{x,d}$ are found using the waypoint tracking algorithms of Section 9.3. The proportional and derivative gains $k_{p,\psi}$, $k_{p,\phi}$, $k_{p,\phi,\theta}$, $k_{p,\theta}$, and $k_{d,\phi}$ are user-selected. They can be chosen by trial and error; however, a later section will describe better methods such as root-locus design and successive loop closure.

The block diagram of Figure 9.4 shows that the reference roll angle $\phi_{x,r}$

$$\phi_{x,r} = k_{p,\psi}\Delta\psi \tag{9.15}$$

is saturated to stay within the limits $\phi_{x,r} \in [\phi_{\min}, \phi_{\max}]$. Saturation limits prevent the plane from rolling too much when trying reach a desired yaw angle $\psi_{z,d}$. For example, if the plane must make a 180° turn, setting the saturation limits to $\phi_{x,r} \in [-\pi/4, \pi/4]$ causes the airplane to bank to ±45° while making the

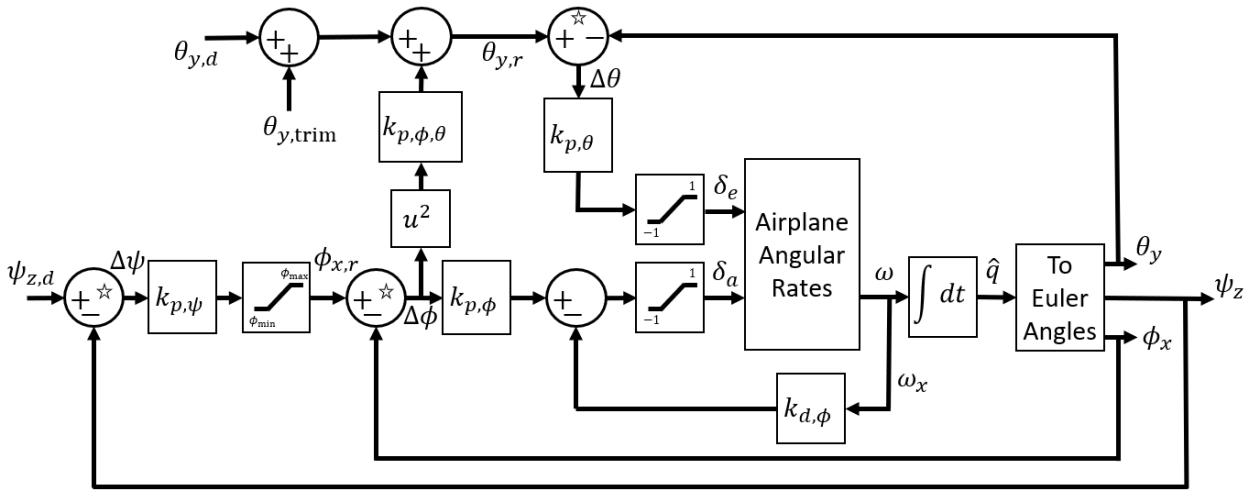


Figure 9.4 Block diagram of the rudderless control strategy.

turn. All the angle differences $\Delta\psi$, $\Delta\phi$, and $\Delta\theta$ are calculated using the smallest angle difference formula Eq. (9.14). The roll error $\Delta\phi$

$$\Delta\phi = \phi_{x,r} - \phi_x \tag{9.16}$$

is the smallest angle difference between the reference roll angle $\phi_{x,r}$ and the actual roll angle ϕ_x of the plane. The yaw error $\Delta\psi$

$$\Delta\psi = \psi_{z,d} - \psi_z \tag{9.17}$$

is the smallest angle difference between the desired yaw angle $\psi_{z,d}$ and the actual yaw angle ψ_z.

The aileron command δ_a is a proportional-derivative (PD) controller that depends on the roll error $\Delta\phi$ and the roll-rate ω_x.

$$\boxed{\delta_a = k_{p,\phi}\Delta\phi - k_{d,\phi}\omega_x} \tag{9.18}$$

It is constricted to within $\delta_a \in [-1,1]$. The proportional and derivative gains, $k_{p,\phi}$ and $k_{d,\phi}$ respectively, are user-selected tuning parameters.

The reference pitch angle $\theta_{y,r}$ depends on the desired pitch angle $\theta_{y,d}$, pitch trim $\theta_{y,\text{trim}}$, and the square of the roll error $\Delta\phi^2$.

$$\theta_{y,r} = \theta_{y,d} + \theta_{y,\text{trim}} + k_{p,\phi,\theta}\Delta\phi^2 \tag{9.19}$$

The gain $k_{p,\phi,\theta}$ is a user-selected tuning parameter. The pitch error $\Delta\theta$

$$\Delta\theta = \theta_{y,r} - \theta_y \tag{9.20}$$

is the smallest angle difference (see Eq. (9.14)) between the reference pitch angle $\theta_{y,r}$ and the actual pitch angle θ_y of the airplane. The elevator command δ_e is proportional to the pitch error.

$$\boxed{\delta_e = k_{p,\theta}\Delta\theta} \tag{9.21}$$

Saturation limits the elevator command to $\delta_e \in [-1,1]$.

In this control strategy, the throttle command δ_t is assumed to be set by the waypoint algorithm, see Section 9.3. The rudder command is $\delta_r = 0$.

Control with a Rudder

If the airplane has a rudder, the yaw control can be improved. The rudderless control algorithm of Figure 9.4 (see Eqs. (9.15) - (9.21)) is still applicable, and can be augmented with a proportional rudder command δ_r

$$\delta_r = k_\psi \Delta\psi \tag{9.22}$$

where $\Delta\psi$ is calculated using the smallest angle formula Eq. (9.14). The proportional gain k_ψ is a user-selected tuning parameter. A block diagram of the control strategy is shown in Figure 9.5.

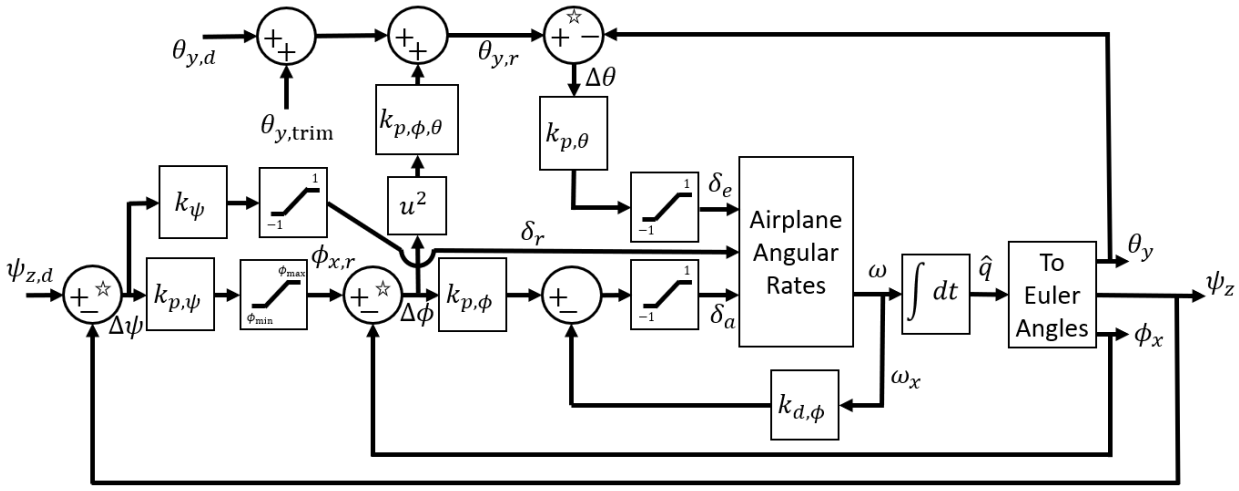


Figure 9.5 Block diagram of the autonomous flight control strategy with rudder control.

9.3 Waypoint Tracking and Path Planning

For waypoint tracking, a list of 3-dimensional coordinates are provided. The goal is to have the airplane visit each of the 3D coordinates (waypoints) in the listed sequence at a desired speed or throttle position. Waypoint tracking algorithms use the 3D coordinates to calculate desired roll, pitch, and yaw angles. They determine the distance from the plane to the present waypoint. The algorithms determine when the plane has visited one waypoint and can move on to the next.

In this document, a list of N waypoints is contained in an $N \times 4$ array. The first three elements in each row are the inertial $x_{I,d}$, $y_{I,d}$, and $z_{I,d}$ coordinates of the waypoint. If coordinates are specified as latitude (degrees), longitude (degrees), and altitude (m) instead of displacements x_I, y_I, and z_I, Section 3.3.2 explains how to convert the coordinates to displacements. The fourth element is the desired forward velocity u_d for approaching the waypoint or, alternatively, the throttle position δ_t. Including this fourth element is especially useful for landing the airplane, where the desired velocity for approaching the landing is the airplane's stall speed and the throttle command for unmanned airplanes is often $\delta_t = 0$ to protect the propeller.

A waypoint tracking algorithm calculates the x, y, and z distances Δx_I, Δy_I, and Δz_I between the airplane and the targeted waypoint:

$$\Delta x_I = x_{I,d} - x_I \tag{9.23}$$
$$\Delta y_I = y_{I,d} - y_I \tag{9.24}$$
$$\Delta z_I = z_{I,d} - z_I \tag{9.25}$$

where $x_{I,d}$ is the waypoint x-axis coordinate and x_I is the inertial x-position of the airplane, etc. The absolute distance between the airplane and the waypoint is

$$\sqrt{\Delta x_I^2 + \Delta y_I^2 + \Delta z_I^2} \tag{9.26}$$

The waypoint tracking algorithm can compare the absolute distance to a threshold to decide when a waypoint has been visited and to move onto the next.

Waypoint tracking algorithms use Equations (9.23) – (9.26) to calculate the desired roll $\phi_{x,d}$, pitch $\theta_{y,d}$, and yaw $\psi_{z,d}$ angles.

9.3.1 Desired Roll, Pitch, and Yaw Angles

In general, as the airplane approaches a waypoint, the desired roll angle is zero:

$$\phi_{x,d} = 0 \tag{9.27}$$

The desired pitch angle is calculated from the absolute distance $\sqrt{\Delta x_I^2 + \Delta y_I^2 + \Delta z_I^2}$ and the change in altitude Δz_I as follows (see Figure 9.6):

$$\theta_{y,d} = \sin^{-1}\left(\frac{-\Delta z_I}{\sqrt{\Delta x_I^2 + \Delta y_I^2 + \Delta z_I^2}}\right) \tag{9.28}$$

Eq. (9.28) assumes a positive downward z-direction.

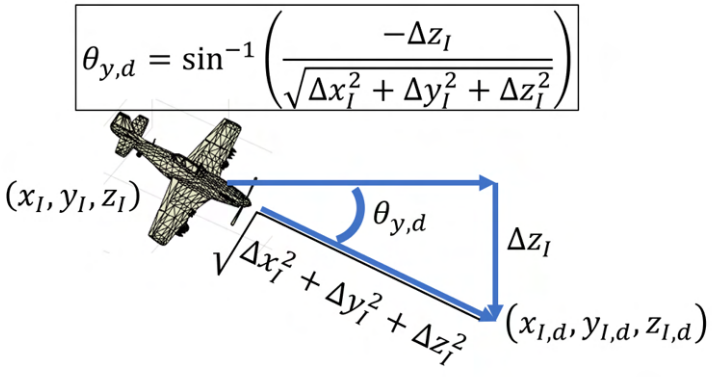


Figure 9.6 The desired pitch angle.

The desired yaw angle $\psi_{z,d}$ is calculated as

$$\psi_{z,d} = \text{atan2}\left(\Delta y_I, \Delta x_I\right) \tag{9.29}$$

where atan2 is the four-quadrant inverse tangent.

9.4 (*Optional*) Alternate Interpretation of Roll, Pitch, and Yaw Angles

Roll (ϕ_x), pitch (θ_y), and yaw (ψ_z) angles can describe the orientation of the aircraft, see Figure 9.1. Unlike Euler angles, defining roll, pitch, and yaw directly from quaternion components (the method of this section) is independent of the sequence in which the rotations are applied. This method is not susceptible to gimbal lock. However, control is more challenging because roll, pitch, and yaw are not decoupled as they are with Euler angles.

One mathematical perspective of the quaternion is that it is a rotation around a specific axis. The rotation angle is ρ, and the unit vector for the rotation axis is $\begin{pmatrix} e_x & e_y & e_z \end{pmatrix}$. The four components of the quaternion are

$$e_0 = \cos\left(\frac{\rho}{2}\right) \tag{9.30}$$

$$e_1 = \sin\left(\frac{\rho}{2}\right) e_x \tag{9.31}$$

$$e_2 = \sin\left(\frac{\rho}{2}\right) e_y \tag{9.32}$$

$$e_3 = \sin\left(\frac{\rho}{2}\right) e_z \tag{9.33}$$

With this perspective, because all rotation angles are between $\pm 180°$, *i.e.*, $-\pi \leq \rho \leq \pi$, the value of e_0 cannot be negative. If it is negative, the following correction is made to force it to be positive:

$$\begin{aligned} &\text{if } e_0 < 0 \\ &\quad e_0 = -e_0 \\ &\quad e_1 = -e_1 \\ &\quad e_2 = -e_2 \\ &\quad e_3 = -e_3 \\ &\text{end} \end{aligned} \tag{9.34}$$

Making this correction allows us to require the angle of rotation to also be positive: $\rho \geq 0$. We can calculate ρ from Eq. (9.30), the scalar part of the quaternion e_0.

$$\rho = 2\cos^{-1}\left(e_0\right) \tag{9.35}$$

If $\cos\left(\rho/2\right) = 1$, the angle of rotation is $\rho = 0$. Because the airplane is not rotated, the angles ϕ_x, θ_y, and ψ_z are all zero. When $\rho \neq 0$, the identity $\cos^2(\rho/2) + \sin^2(\rho/2) = 1$ provides an equation for $\sin\left(\rho/2\right)$:

$$\sin\frac{\rho}{2} = \sqrt{1 - e_0^2} \tag{9.36}$$

and the roll ϕ_x, pitch θ_y, and yaw ψ_z angles can be calculated by multiplying the rotation angle ρ by the respective rotation axis components e_x, e_y, and e_z (from Eq. (9.31) – Eq. (9.33)):

How to convert quaternions to roll ϕ_x, pitch θ_y, and yaw ψ_z

$$\rho = 2\cos^{-1}(e_0)$$

$$\phi_x = \rho \frac{e_1}{\sqrt{1-e_0^2}} \tag{9.37}$$

$$\theta_y = \rho \frac{e_2}{\sqrt{1-e_0^2}} \tag{9.38}$$

$$\psi_z = \rho \frac{e_3}{\sqrt{1-e_0^2}} \tag{9.39}$$

9.4.1 Setting Desired Quaternion Orientation

Eqs. (9.37) – (9.39) converted quaternions to roll, pitch, and yaw orientation. This section does the opposite. It converts roll, pitch, and yaw orientation to unit quaternions. It is conceptually easier to visualize orientation in terms of roll, pitch, and yaw than in terms of quaternions. For example, consider a desired yaw of 45° North-East, a pitch angle of −15° pitched downward below the horizon, and a desired roll of 30° banked to the right. What would its orientation be in terms of quaternions? The control strategies of this paper require reference orientations to be quaternions. This section explains how to convert roll, pitch, and yaw orientations to quaternions.

Eqs. (9.30) – (9.33) presented one perspective of a unit quaternion. The rotation angle ρ is the total magnitude of the roll ϕ_x, pitch θ_y, and yaw ψ_z angles:

$$\rho = \sqrt{\phi_x^2 + \theta_y^2 + \psi_z^2} \tag{9.40}$$

The rotation axis $\begin{pmatrix} e_x & e_y & e_z \end{pmatrix}$ is a unit vector. It can be found directly from the roll ϕ_x, pitch θ_a, and yaw ψ_z angles as follows:

$$e_x = \frac{\phi_x}{\rho} \tag{9.41}$$

$$e_y = \frac{\theta_y}{\rho} \tag{9.42}$$

$$e_z = \frac{\psi_z}{\rho} \tag{9.43}$$

Using Eqs. (9.30) – (9.33) we get the following:

How to convert roll ϕ_x, pitch θ_y, and yaw ψ_z to quaternions

$$\rho = \sqrt{\phi_x^2 + \theta_y^2 + \psi_z^2} \tag{9.44}$$

$$e_0 = \cos\left(\frac{\rho}{2}\right) \tag{9.45}$$

$$e_1 = \sin\left(\frac{\rho}{2}\right)\frac{\phi_x}{\rho} \tag{9.46}$$

$$e_2 = \sin\left(\frac{\rho}{2}\right)\frac{\theta_y}{\rho} \tag{9.47}$$

$$e_3 = \sin\left(\frac{\rho}{2}\right)\frac{\psi_z}{\rho} \tag{9.48}$$

The proof can be obtained by plugging Eqs. (9.37) – (9.39) into Eqs. (9.45) – (9.48).

Example:
Convert a desired yaw of 45° east of due north, a downward pitch angle of −15°, and a desired roll of 30° (see Figure 9.7) to a quaternion.

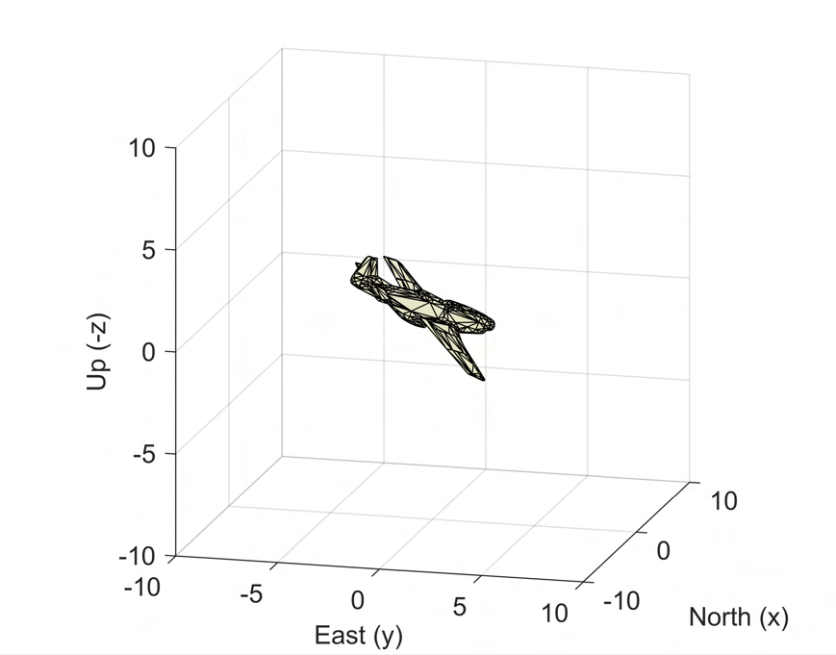


Figure 9.7 Airplane with an orientation of $\phi_x = 30°$, $\theta_y = -15°$, and $\psi_z = 45°$.

Solution: Using Eqs. (9.45) – (9.48), we get

$$\rho = \sqrt{\phi_x^2 + \theta_y^2 + \psi_z^2} = \sqrt{45^2 + (-15)^2 + 30^2} = 56.12^\circ$$
$$e_0 = \cos\left(\frac{\rho}{2}\right) = \cos\left(\frac{56.12^\circ}{2}\right) = 0.8824$$
$$e_1 = \sin\left(\frac{\rho}{2}\right)\frac{\phi_x}{\rho} = \sin(\frac{56.12^\circ}{2})\frac{30}{56.12} = 0.2515$$
$$e_2 = \sin\left(\frac{\rho}{2}\right)\frac{\theta_y}{\rho} = \sin\left(\frac{56.12^\circ}{2}\right)\frac{-15}{56.12} = -0.1257$$
$$e_3 = \sin\left(\frac{\rho}{2}\right)\frac{\psi_z}{\rho} = \sin\left(\frac{56.12^\circ}{2}\right)\frac{45}{56.12} = 0.3772$$

The quaternion is therefore

$$q = \begin{bmatrix} 0.8824 \\ 0.2515 \\ -0.1257 \\ 0.3772 \end{bmatrix}$$

9.4.2 Quaternion Errors and Roll, Pitch, and Yaw Errors

The difference or error Δq between the desired reference quaternion q_d and the actual quaternion orientation of the plane q is the quaternion product of q_d and the conjugate of q:

$$\begin{bmatrix} \Delta q_0 \\ \Delta q_1 \\ \Delta q_2 \\ \Delta q_3 \end{bmatrix} = \begin{bmatrix} q_{d,0} & -q_{d,1} & -q_{d,2} & -q_{d,3} \\ q_{d,1} & q_{d,0} & -q_{d,3} & q_{d,2} \\ q_{d,2} & q_{d,3} & q_{d,0} & -q_{d,1} \\ q_{d,3} & -q_{d,2} & q_{d,1} & q_{d,0} \end{bmatrix} \begin{bmatrix} q_0 \\ -q_1 \\ -q_2 \\ -q_3 \end{bmatrix} \tag{9.49}$$

The roll error $\Delta\phi_x$, pitch error $\Delta\theta_y$, and yaw error $\Delta\psi_z$ are calculated from Eqs. (9.37) – (9.39):

$$\Delta\rho = 2\cos^{-1}\left(\Delta q_0\right)$$

$$\Delta\phi_x = \Delta\rho\frac{\Delta q_1}{\sqrt{1-\Delta q_0^2}} \tag{9.50}$$
$$\Delta\theta_y = \Delta\rho\frac{\Delta q_2}{\sqrt{1-\Delta q_0^2}} \tag{9.51}$$
$$\Delta\psi_z = \Delta\rho\frac{\Delta q_3}{\sqrt{1-\Delta q_0^2}} \tag{9.52}$$

Chapter 10

System Identification

Contents

10.1 Fourier Transform

10.1.1 Discrete Fourier Transform (DFT)

The discrete Fourier transform calculates the frequency content of a signal. It works by comparing the measured time-domain signal against sinusoids of various frequencies. The comparison is performed one frequency at a time, over a range of frequencies. When the frequency content of the measured signal and comparison signal match, the summation of their element-wise product is large. When the frequency contents do not match, the summation of their product is small. Most textbooks define the Fourier

transform mathematically as follows:

$$\begin{aligned} x_n &= \sum_{k=0}^{N-1} x_k e^{-j\omega_n t_k} \\ &= \sum_{k=0}^{N-1} x_k e^{-j(2\pi f_n)(\Delta t k)} \\ &= \sum_{k=0}^{N-1} x_k e^{-j\left(2\pi \frac{n}{\Delta t N}\right)(\Delta t k)} \\ &= \sum_{k=0}^{N-1} x_k e^{-j2\pi \frac{n}{N} k} \qquad\qquad n = 0, \ldots, N-1 \end{aligned}$$

However, if you recall Euler's formula that $e^{-j\omega k} = \cos(\omega k) - j\sin(\omega k)$, you may find the following definition more useful, which separates the real and imaginary parts:

$$x_n = \left(\sum_{k=0}^{N-1} x_k \cos\left(2\pi \frac{n}{N} k\right)\right) - j\left(\sum_{k=0}^{N-1} x_k \sin\left(2\pi \frac{n}{N} k\right)\right) \qquad n = 0, \ldots, N-1$$

The input signal x_k sampled at discrete times $t_k = k\Delta t$ where Δt is the constant time interval between samples of data, $k = 0, \ldots, N-1$, and N is the total number of samples in the batch of samples being analyzed (we will assume N is an even number to simplify notation). The output (x_n) is the n^{th} term in the Fourier transform of the measured signal x_k. It contains frequency content information related to the frequency f_n where f_n is:

$$f_n = \begin{cases} \frac{n}{\Delta t N} & n = 0, \ldots, \frac{N}{2} - 1 \\ \frac{N-n-1}{\Delta t N} & n = \frac{N}{2}, \ldots, N-1 \end{cases}$$

The frequencies occur at discrete intervals and the step size between frequencies is

$$\Delta f = \frac{1}{\Delta t N}$$

If the data were collected at consistent time intervals of Δt then the corresponding sample rate would be:

$$f_s = \frac{1}{\Delta t}$$

and the smallest change in frequency that could be detected, or the resolution of the DFT, is

$$\Delta f = \frac{f_s}{N}$$

Only the first $\frac{N}{2}$ terms of the DFT (terms 0 through $\frac{N}{2} - 1$) provide new information. The last $\frac{N}{2}$ terms repeat the information from the first $\frac{N}{2}$ terms, but in reverse order, i.e., $x_o = x_{N-1}$, and $x_1 = x_{N-2}$, and $x_2 = x_{N-3}, \ldots, x_{N/2} = x_{N/2+1}$.

The maximum frequency that can be detected by the DFT is

$$f_{\frac{N}{2}-1} = \frac{\frac{N}{2} - 1}{\Delta t N} = \frac{N-2}{2\Delta t N}$$

which is about half of the sampling frequency

$$f_{\frac{N}{2}-1} \approx \frac{1}{2\Delta t} \approx \frac{1}{2} f_s$$

This frequency is called the Nyquist frequency, or

$$f_{Nyq} = \frac{1}{2} f_s$$

Thus we can only detect frequencies that are less than, but not equal to the Nyquist frequency.

The DFT x_n contains information about the signal at the frequency f_n. For example, the amplitude $|x_n|$ of the signal x_k at f_n can be calculated from x_n as

$$|x_n| = \frac{2}{N}\sqrt{\left(\sum_{k=0}^{N-1} x_k \cos\left(2\pi \frac{n}{N} k\right)\right)^2 + \left(\sum_{k=0}^{N-1} x_k \sin\left(2\pi \frac{n}{N} k\right)\right)^2} \tag{10.1}$$

or more compactly

$$|x_n| = \frac{2}{N}\sqrt{\left(\mathrm{Re}(x_n)\right)^2 + \mathrm{Im}(x_n)\right)^2}$$

where $\mathrm{Re}(x_n) = \sum_{k=0}^{N-1} x_k \cos\left(2\pi \frac{n}{N} k\right)$ is the real part of x_n, and $\mathrm{Im}(x_n) = \sum_{k=0}^{N-1} x_k \sin\left(2\pi \frac{n}{N} k\right)$ is the imaginary part of x_n. The phase shift $\angle x_n$ of the signal x_k at f_n can be calculated from x_n by the following equation:

$$\angle x_n = \mathrm{atan2}\left(\mathrm{Im}(x_n), \mathrm{Re}(x_n)\right) \tag{10.2}$$

Example 10.1 Using discrete Fourier transform to find frequency data from discrete time data.

Given 3000 discrete measured data from the signal

$$x(t) = 2\cos(2\pi 2t) + 4\cos\left(2\pi 4t + \frac{\pi}{2}\right)$$

collected at a sampling rate of $f_s = 100$ Hz, plot the DFT amplitude and phase shift as a function of frequency.

Solution: Collecting measurements at a sample rate of $f_s = 100$ Hz means we will take a measurement every $\Delta t = \frac{1}{f_s} = 0.01$s, or at the discrete times $t_k = 0, 0.01, 0.02, \ldots, 29.99$ s. The measured signal at each time interval is

$$x_k = 2\cos(2\pi 2 t_k) + 4\cos\left(2\pi 4 t_k + \frac{\pi}{2}\right) \qquad k = 0, 1, 2, 3, \ldots, 2999$$

and since $t_k = k\Delta t = 0.01k$

$$\begin{aligned} x_k &= 2\cos(2\pi 2(0.01k)) + 4\cos\left(2\pi 4(0.01k) + \frac{\pi}{2}\right) && k = 0, 1, 2, 3, \ldots, 2999 \\ &= 2\cos(0.04\pi k) + 4\cos\left(0.08\pi k + \frac{\pi}{2}\right) && k = 0, 1, 2, 3, \ldots, 2999 \end{aligned}$$

Notice that this signal has two frequency components. It has a 2 Hz signal with an amplitude of 2 and a phase shift of 0 rad, combined with a 4 Hz signal with an amplitude of 4 and a phase shift of $\frac{\pi}{2}$. A two-second sample

of the signal is shown in Figure 10.1. The dots on the graph mark the locations at which the continuous signal $x(t)$ is sampled to get the measured signal x_k.

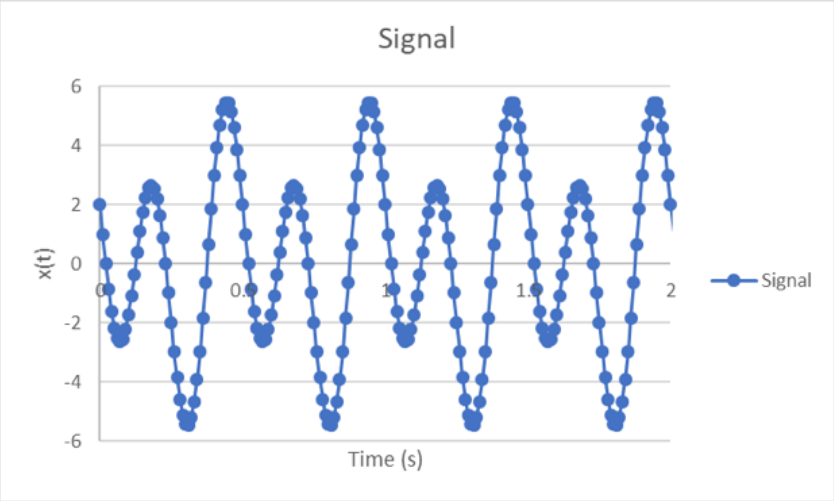


Figure 10.1 Two seconds of the signal $x(t) = 2\cos(2\pi 2t) + 4\cos\left(2\pi 4t + \frac{\pi}{2}\right)$ and sampled data points.

Normally we don't know the signal and would need to calculate all the frequencies from 0 to the Nyquist frequency. With $N = 3000$ measurements, a full DFT would need to calculate the amplitude and phase shift at $n = 0 \ldots \frac{N}{2} - 1 = 0 \ldots 1499$ different frequencies. In this case, lets just consider f_n at $n = 120$, which is $f_{120} = \frac{n}{\Delta t N} = \frac{120}{(0.01)(3000)} = 4$ Hz, which corresponds to the 4 Hz component of the signal. To find the amplitude we will use Eq. (10.1).

$$
\begin{aligned}
|x_{120}| &= \frac{2}{N}\sqrt{\left(\sum_{k=0}^{N-1} x_k \cos\left(2\pi \frac{n}{N} k\right)\right)^2 + \left(\sum_{k=0}^{N-1} x_k \sin\left(2\pi \frac{n}{N} k\right)\right)^2} \\
&= \frac{2}{3000}\sqrt{\left(\sum_{k=0}^{2999} x_k \cos\left(2\pi \frac{120}{3000} k\right)\right)^2 + \left(\sum_{k=0}^{2999} x_k \sin\left(2\pi \frac{120}{3000} k\right)\right)^2} \\
&= \frac{2}{3000}\sqrt{\left(\sum_{k=0}^{2999} x_k \cos\left(0.08\pi k\right)\right)^2 + \left(\sum_{k=0}^{2999} x_k \sin\left(0.08\pi k\right)\right)^2}
\end{aligned}
$$

and substituting the signal x_k in gives a real and imaginary parts underneath the square root

$$
\begin{aligned}
\mathrm{Re}(x_{120}) &= \sum_{k=0}^{2999} \left[2\cos(0.04\pi k) + 4\cos\left(0.08\pi k + \frac{\pi}{2}\right)\right] \cos(0.08\pi k) \\
\mathrm{Im}(x_{120}) &= \sum_{k=0}^{2999} \left[2\cos(0.04\pi k) + 4\cos\left(0.08\pi k + \frac{\pi}{2}\right)\right] \sin(0.08\pi k) \\
|x_{120}| &= \frac{2}{3000}\sqrt{(\mathrm{Re}(x_{120}))^2 + (\mathrm{Im}(x_{120}))^2}
\end{aligned}
$$

When all the measurements in the signal x_k are multiplied in the real part of the equation by $\cos(0.08\pi k)$ we get the plot shown on the left in Figure 10.2. When the signal x_k is multiplied in the imaginary part of the equation by $\sin(0.08\pi k)$ we get the plot shown on the right in Figure 10.2.

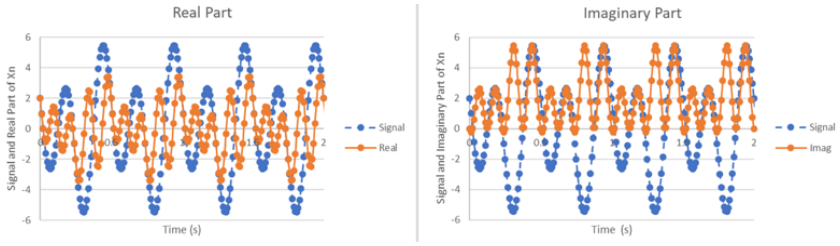


Figure 10.2 Plots of the both the real and imaginary parts of the underneath the square root. The measured signals are the blue dots whereas the multiplication is the orange dots

When we add up all the values of the real part, the sum is very close to zero

$$\text{Re}(x_{120}) = -4.29 \times 10^{-10}$$

This is because the negative values (shown on the left of Figure 10.2) cancel with the positive values.

When we add up all the values of the imaginary part we get a large positive number

$$\text{Im}(x_{120}) = 6000$$

This is because all the values are positive, as shown in the plot in Figure 10.2.

The amplitude of the 4 Hz component of the signal can now be calculated to be

$$\begin{aligned} |x_{120}| &= \frac{2}{3000}\sqrt{(\text{Re}(x_{120}))^2 + (\text{Im}(x_{120}))^2} \\ &= \frac{2}{3000}\sqrt{\left(-4.29 \times 10^{-10}\right)^2 + 6000^2} \\ &= 4 \end{aligned}$$

This exactly matches the 4 Hz amplitude of the signal x_k!

We can calculate the phase shift using Eq. (10.2).

$$\angle x_{120} = \text{atan2}\left(6000, -4.29 \times 10^{-10}\right) = \frac{\pi}{2}$$

This exactly matches the phase shift of the 4 Hz component of the signal x_k!

What if we apply the Fourier transform corresponding to $n = 60$ or $f_{60} = \frac{n}{\Delta t N} = \frac{60}{(0.01)(3000)} = 2$ Hz? The amplitude is found to be $|x_{60}| = 2$, which was exactly the amplitude of the 2 Hz component of the signal x_k! The phase shift is $\angle x_{60} = 0$, which exactly matches the phase shift of the 2 Hz component of the signal x_k!

What happens if we consider any other frequency? For example, consider $n = 90$ which corresponds to a frequency of $f_{90} = \frac{n}{\Delta t N} = \frac{90}{(0.01)(3000)} = 3$ Hz? The real part $\text{Re}(x_n)$=-9.93×10^{-13}, and the imaginary part $\text{Im}(x_n)$=1.20×10^{-14}. The magnitude is $|x_{90}| = 6.6 \times 10^{-6}$, which is practically zero; the phase shift $\angle x_{90} = \text{atan2}(0,0)$ is undefined. These results are exactly as expected because the signal x_k does not have a 3 Hz frequency component.

The discrete Fourier transform magnitude and phase of the signal x_k for the entire frequency domain can be seen in Figures 10.3 and 10.4. Only two frequencies have a non-zero amplitude: 2 Hz and 4 Hz. Only the 4 Hz frequency has a phase shift.

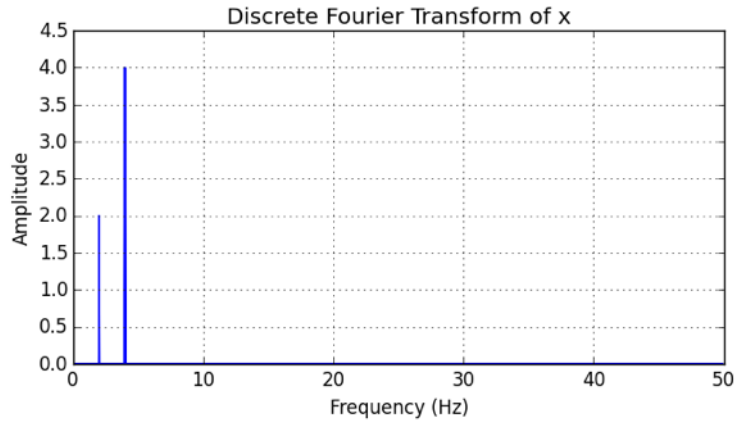


Figure 10.3 DFT amplitude

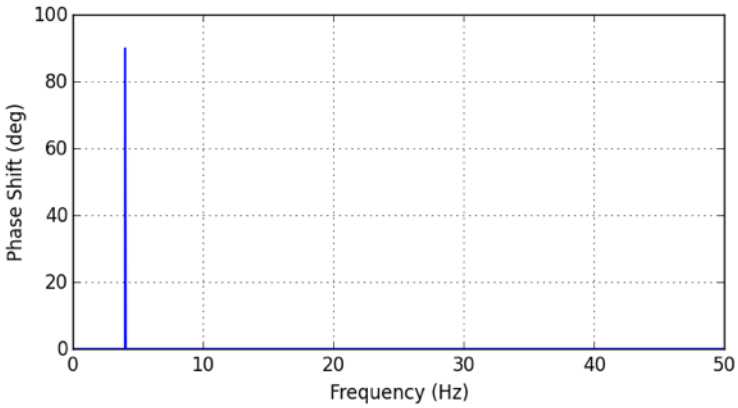


Figure 10.4 DFT phase shift

Note that the plots in Figures 10.3 and 10.4 only go to a maximum frequency of 50Hz. This is, if you recall, the Nyquist frequency because the sampling frequency was $f_s = 100$ Hz so the Nyquist frequency would be $f_{Nyq} = \frac{f_s}{2} = 50$ Hz. Thus, we can only detect frequencies below and not equal to the Nyquist frequency, or half the sampling frequency.

To be more exact, the maximum frequency that can be detected is:

$$\begin{aligned} f_{\frac{N}{2}-1} &= \frac{\frac{N}{2}-1}{\Delta t N} \\ &= \frac{\frac{3000}{2}-1}{(0.01)(3000)} \\ &= \frac{1499}{30} \\ &= 49.\overline{6} \end{aligned}$$

which is less than the Nyquist frequency of $f_{Nyq} = 50$ Hz.

10.2 Fast Fourier Transform (FFT)

Calculating the discrete Fourier transform can be computationally demanding. For fast sampling rates and limited computational processing speed, calculating the discrete Fourier transform directly may be too slow. A variety of algorithms have been developed to reduce computational requirements. These are called Fast Fourier Transform (FFT) algorithms. The most popular is known as the Cooley-Tukey algorithm. It reduces the computational requirements by dividing the N data points of the measured signal x_k into smaller samples. It calculates the discrete Fourier transform of these smaller samples and combines them to get the transform of the entire signal.

10.3 Aliasing

10.3.1 Aliasing Significance in Measured Signals

Now, suppose we are given the following input voltage:

$$V(t) = \sin(20\pi t)$$

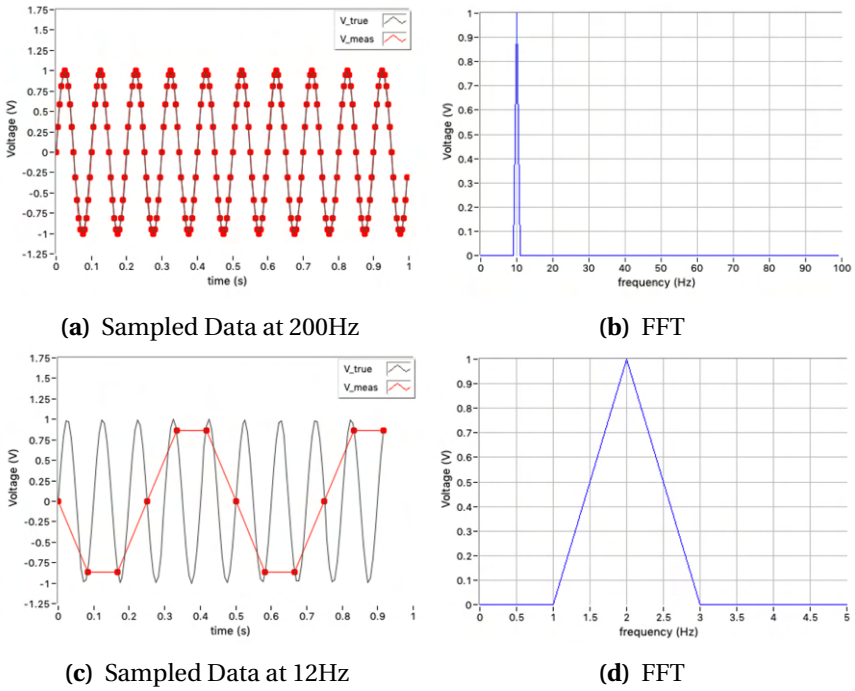
with an amplitude of 1 V and an input frequency of 10 Hz. If we measure the input voltage using a DAQ with a high number of bits so that the measured voltage is almost exactly equal to the input voltage (so quantization error is negligible), and we use a sample rate of 200 Hz to get 200 measurements for one second of data, we would get the resulting data plot in Figure 10.5a. From this plot, we can see that the waveform has a significant amount of measurements. If we use the data in an FFT we would get the plot in Figure 10.5b, indicating that the FFT detected the 10 Hz frequency with an amplitude of 1 V.

Now suppose we decrease the sample rate to 12 Hz and the number of samples to 12 so that we still get about one second of data. Our resulting data would look like the plot in Figure 10.5c. This shows that connecting the measured data together makes an oscillating waveform that doesn't follow the input voltage frequency. In fact, the FFT of the measured data can be seen in Figure 10.5d, indicating that the FFT thinks the input is oscillating at 2 Hz, when we really know that it is oscillating at 10 Hz.

The data in Figure 10.5 demonstrates aliasing, which means the data is suggesting one thing (an input voltage with a frequency of 2 Hz) when in fact it is not true (because the input voltage really has an input frequency of 10 Hz). Thus our FFT gives us a perceived or aliased frequency which is not equal to the correct frequency.

Notice in the FFT in Figure 10.5d using 12 samples of data collected at a sample rate of 12 Hz that the peak of the FFT is very wide in comparison to the FFT when we collected the 200 samples at a sample rate of 200 Hz. This is because the smallest change in frequency that can be detected in both cases are the same:

$$\Delta f = \frac{f_s}{N} = \frac{200}{200} = 1 \text{ Hz}$$
$$\Delta f = \frac{f_s}{N} = \frac{12}{12} = 1 \text{ Hz}$$



(a) Sampled Data at 200Hz

(b) FFT

(c) Sampled Data at 12Hz

(d) FFT

Figure 10.5 An input voltage signal of $V(t) = \sin(20\pi t)$ with an input frequency of 10 Hz is measured with a DAQ. When sampled at a rate of 200 Hz to get 200 measurements the top two plots are the results. When sampled at a lower sampling rate of 12 Hz to get 12 measurements, the bottom two plots are the results. The black lines are the true input voltage while the red dots are the measured data. The FFT plots on the left show the detected frequencies with the data given by the plots to their left.

but the maximum frequency that can be detected in each case is different

$$f_{\frac{N}{2}-1} = \frac{\frac{N}{2}-1}{\Delta t N} = \frac{\frac{200}{2}-1}{\frac{1}{200}(200)} = 99 \text{ Hz}$$
$$f_{\frac{N}{2}-1} = \frac{\frac{N}{2}-1}{\Delta t N} = \frac{\frac{12}{2}-1}{\frac{1}{12}(12)} = 5 \text{ Hz}$$

so the 1 Hz resolution looks to be a lot larger on the scale from 0 to 5 Hz in comparison with the 0 to 99 Hz scale.

10.3.2 Calculating the Aliasing Frequency

In the above example the FFT reported to have measured two different input frequencies, depending on the sampling frequency we used. In the end we knew the reported 2 Hz frequency was incorrect.

We can calculate the perceived or aliased frequency using the following equation

$$f_p = \left| f - f_s \cdot \text{NINT}\left(\frac{f}{f_s}\right) \right|$$

where f is the actual input frequency, f_s is the sample rate, and NINT is "nearest integer" or round the fraction to the nearest integer.

For example, using the 10 Hz input signal and 200 Hz sample rate from the example above, we get:

$$\begin{aligned} f_p &= \left| 10 - 200 \cdot \text{NINT}\left(\frac{10}{200}\right) \right| \\ &= |10 - 200 \cdot \text{NINT}(0.05)| \\ &= |10 - 200 \cdot 0| \\ &= 10 \text{ Hz} \end{aligned}$$

which is the correct frequency, thus the perceived frequency is equal to the input frequency and we aren't experiencing any aliasing.

Using the 12 Hz sample rate instead gives:

$$\begin{aligned} f_p &= \left| 10 - 12 \cdot \text{NINT}\left(\frac{10}{12}\right) \right| \\ &= |10 - 12 \cdot \text{NINT}(0.833)| \\ &= |10 - 12 \cdot 1| \\ &= 2 \text{ Hz} \end{aligned}$$

which is not the correct frequency, indicating that we are experiencing aliasing.

10.3.3 Calculating the Nyquist Frequency of an Input Signal

From the perceived frequency equation, we can see that if we would like the perceived frequency to equal the actual input frequency then we will need the NINT term to go to zero, which means the ratio of $\frac{f}{f_s}$ needs to be less than 0.5:

$$\frac{f}{f_s} < 0.5$$

and solving for f_s we get:

$$\begin{aligned} f_s &> \frac{f}{0.5} \\ &> 2f \end{aligned}$$

So, in order to get the perceived frequency to equal the true or input frequency then we need to make sure that the sample rate is greater than two times the input frequency.

If we recall that the Nyquist frequency is the maximum detectable frequency and is calculated as

$$f_{Nyq} = \frac{f_s}{2}$$

then we can see why the sample rate needs to be greater than twice the input frequency. We can combine the above two equations to see that if the sample rate is greater than $2f$ then the Nyquist frequency will be greater than the input frequency and thus detectable by the FFT:

$$\begin{aligned} f_{Nyq} &= \frac{f_s}{2} \\ &> \frac{2f}{2} \\ &> f \end{aligned}$$

Note that if we use a sample rate that is barely greater than the input frequency then the perceived frequency given by the FFT will equal the true input frequency, but it may not give the correct amplitude. In order to assure that you are getting the correct amplitude then we need to use a sampling rate that is much greater than the input frequency, usually about 10 times greater than the input frequency. This will assure that there will be 10 data measurements per period of the input waveform and give a better chance to capture the input amplitude.

For example, consider the FFT of an input signal with a frequency of 100 Hz and amplitude of 1 V:

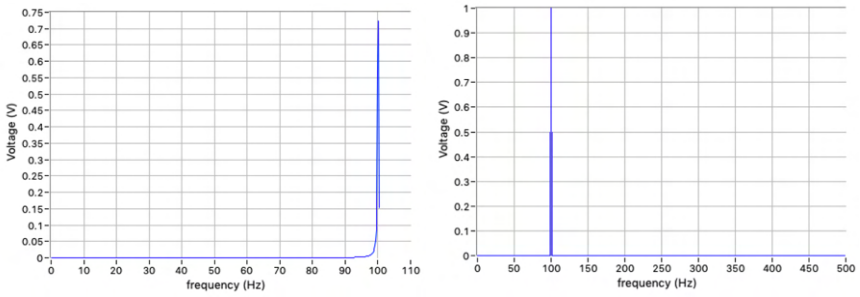
$$V(t) = \sin(2\pi(100)\,t)$$

and we use a sample rate of $f_s = 2f + 1 = 201$ Hz to collect 1000 samples. We would get the FFT in Figure 10.6a. Because the Nyquist frequency at this sample rate is $f_{Nyq} = \frac{201}{2} = 100.5$ Hz, then we are able to detect the true input frequency, but the detected amplitude is only 0.7 volts, not the true 1 volt amplitude. If we increase the sampling rate to 10 times the input frequency, or $f_s = 10f = 1000$ Hz, to collect the 1000 samples then we will get the FFT in Figure 10.6b. We can see that true input frequency of 100 Hz and the true input amplitude of 1 volt was captured in the FFT. Thus, using a sample rate that is much higher than twice the input frequency helped to capture the amplitude of the input signal.

Let's consider a different example. Assume we are given the following as an input signal:

$$V_{in}(t) = 5\sin\left(2\pi(375)\,t + \frac{\pi}{8}\right)$$

then the input signal has an amplitude of 5 volts, a frequency of 375 Hz, and a phase shift of $\frac{\pi}{8}$. Let's pretend that we don't know the true input frequency, but we decide to try to measure the signal with a sample rate of $f_s = 100$ Hz and we get the FFT plot in Figure 10.7a. It appears that the signal has a



(a) $f_s = 201$ Hz **(b)** $f_s = 1000$ Hz

Figure 10.6 Two FFT plots of an input signal with a frequency of 100 Hz and amplitude of 1 volt is measured with two different sampling rates. Since both were above the Nyquist frequency, they were both able to measure the frequency of the input signal. Only the FFT on the right was able to measure the true amplitude since the sample rate is 10 times greater than the frequency of the input signal.

frequency of 25 Hz and amplitude of 5 volts. To be sure, we increase the sample rate to 300 Hz and expect the FFT to show a peak again at 25 Hz, but instead we get the FFT plot in Figure 10.7b, which shows a frequency of 75 Hz. This should clue us in that we are experiencing aliasing because the perceived frequency has changed if we measure the data at a faster sampling rate. We can increase the sample rate again and again until we see that the perceived frequency has stopped changing. Once we increase the sampling rate such that the Nyquist frequency is greater than the input frequency then the perceived frequency will stop changing. Thus we will need to set the sampling rate to a frequency greater than twice the input frequency:

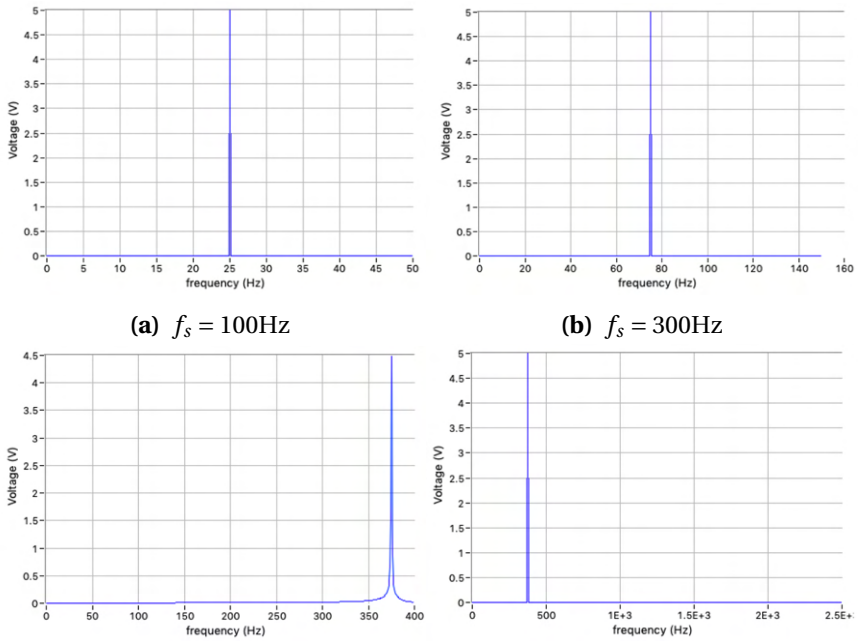
$$f_s > 2f = 750 \text{ Hz}$$

so suppose we choose to sample at $f_s = 800$ Hz, then the Nyquist frequency will be

$$f_{Nyq} = \frac{f_s}{2} = 400 \text{ Hz}$$

which is greater than the input frequency. Doing so we get the FFT in Figure 10.7c. which captured the correct frequency but at an amplitude of only 4.5 volts. If we continue to increase the sample rate we would see that the perceived frequency would stop changing but the perceived amplitude would get closer to the true amplitude of 5 volts. For example, using a sample rate of 5 kHz we would get the FFT in Figure 10.7d. Thus, in real life if we are not aware of the true input frequency, we may need to increase the sampling rate until the perceived frequency stops changing.

In real life, it is unlikely that our signal will be a pure sinusoidal signal with an FFT having only a single peak. For example, the FFT of the data collected from a microphone listening to an A note (440 Hz) can be seen in Figure 10.8. As we can see, real data will have a lot of scatter and consequently a lot of frequency content.



(a) $f_s = 100\text{Hz}$ **(b)** $f_s = 300\text{Hz}$

(c) $f_s = 800\text{Hz}$ **(d)** $f_s = 5\text{kHz}$

Figure 10.7 Various FFT plots measuring an unknown signal with increased sampling rates. At lower sampling rates aliasing is present in the data. Plot 10.7c catches the true frequency since increasing the sample rate does not change the measured frequency. Plot 10.7d is 10 times the measured sampling rate so it is also able to detect the amplitude of the signal.

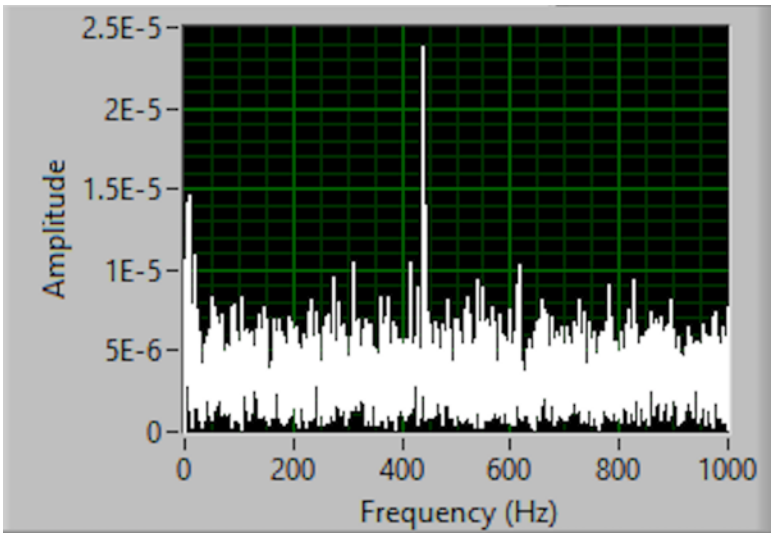


Figure 10.8 The FFT of data collected from a microphone listening to an A note (440 Hz).

10.4 Bode Plots from Experimental Data

A useful application of Fourier transforms is the ability to create Bode plots and frequency response plots from experimental data. Bode plots can be used to identify parameters of differential equations. The slope of the Bode plot is related to the order of the differential equation. The points on the Bode plot at which two asymptotes cross can be used to identify cutoff and natural frequencies, time constants, and damping ratios.

Fourier transforms can be used to create Bode plots without requiring a mathematical model or the underline governing differential equations. Recall that the transfer function of a system is the ratio of the output $Y(s)$ over the input $U(s)$

$$TF(s) = \frac{Y(s)}{U(s)} \tag{10.3}$$

A Bode plot consists of two graphs: (1) Magnitude $dB = 20\log_{10}(r_{TF})$, and (2) Phase angle θ_{TF}. In this chapter, we show that the Fourier transform can find them from experimental data. To find the magnitude r_{TF} of the transfer function at the frequencies $f_n,\ n = 0,\ldots,\frac{N}{2}-1$, we perform the following element-wise division

$$r_{TF,n} = \frac{|Y_n|}{|U_n|}, \quad n = 0,\ldots,\frac{N}{2}-1 \tag{10.4}$$

where $|Y_n|$ is the magnitude of the Fourier transform of the output $y(t)$ at the n^{th} frequency, and $|U_n|$ is the magnitude of the Fourier transform of the input $u(t)$ at the n^{th} frequency f_n.

To get the phase angle $\theta_{TF,n}$ of the transfer function of the n^{th} frequency f_n, we subtract the phase angle $\angle U_n$ of the input from the phase angle $\angle Y_n$ of the output:

$$\theta_{TF,n} = \angle Y_n - \angle U_n, \quad n = 0,\ldots,\frac{N}{2}-1 \tag{10.5}$$

The following example uses the Fourier transform to find the Bode plot of a series resistor-capacitor electrical circuit.

Example 10.2 Use the discrete Fourier transform to find the bode plot of an RC circuit

A series resistor capacitor circuit has a resistance of $R = 200\ \Omega$ and capacitance of $C = 1$ mF. The input voltage V_{in} to the circuit is a chirp waveform:

$$V_{in}(t) = \sin\left(2\pi f_0\left(\frac{r^{t-T}-1}{\ln(r)}\right)\right)$$

where the initial frequency is $f_0 = 1$ Hz, the exponentially increasing frequency rate is $r = 1.25$, and the sweep time interval is $T = 8$. The time-step is $\Delta t = 0.001$ s.

The output signal is the voltage drop across the capacitor V_C. Use the Fourier transform to determine the Bode plot of the system. Compare the Bode plot derived from the Fourier transform with the Bode plot calculated mathematically from the transfer function

$$TF(s) = \frac{V_C(s)}{V_{in}(s)}$$

Solution:

We can use Equation (10.4) to calculate the magnitude and Equation (10.5) to calculate the phase angle for the Bode plot. To use these equations, we must first calculate the Fourier transform of the input and output signals.

The input signal is V_{in}, and V_C is the output signal. The input signal is the chirp waveform. To get the input signal experimentally, we would measure the voltage input using a data acquisition device. We would store the result in an array V_{in}. Because the input is the chirp signal, the k^{th} element of the array would be

$$V_{in,k} = \sin\left(2\pi f_0\left(\frac{r^{k\Delta t - T} - 1}{\ln(r)}\right)\right), \quad k = 0,\ldots,N$$

To get the output signal V_C experimentally, we would measure the voltage across the capacitor using a data acquisition device. We would store the result in an array V_C. Because V_C is the voltage drop across the capacitor in a series resistor-capacitor circuit, the k^{th} element of the V_C array would be (assuming ideal components):

$$V_{C,k} = \exp\left(\frac{-\Delta t}{RC}\right)V_{C,k-1} + \left(1 - \exp\left(\frac{-\Delta t}{RC}\right)\right)V_{in,k-1}, \quad k = 1,\ldots,N$$

where the initial condition is $V_{C,0} = 0$, if the capacitor was initially discharged. This equation is the discrete-time solution to the ODE governing the behavior of the resistor-capacitor circuit. Experimentally, there would be no need to know this equation. We would only need to measure the voltage drop across the capacitor, and store the result in the array V_C.

We can calculate the magnitude (or amplitude) $|V_{in,n}|$ of the input by replacing the variable x_k in Equation (10.1) with the input $V_{in,k}$. We can also calculate the magnitude $|V_{C,n}|$ of the output by replacing the variable x_k in Equation (10.1) with the output $V_{C,k}$.

Similarly, we can calculate the phase angles $\angle V_{in}$ and $\angle V_C$ of V_{in} and V_C by replacing x_k in Equation (10.2) with V_{in} and V_C respectively. Finally, we can use Equations (10.4) and (10.5) to calculate the Bode plot magnitude $r_{TF,N}$ and phase angle $\theta_{TF,n}$ respectively. These results are compared with the theoretical Bode plot in Figure 10.9.

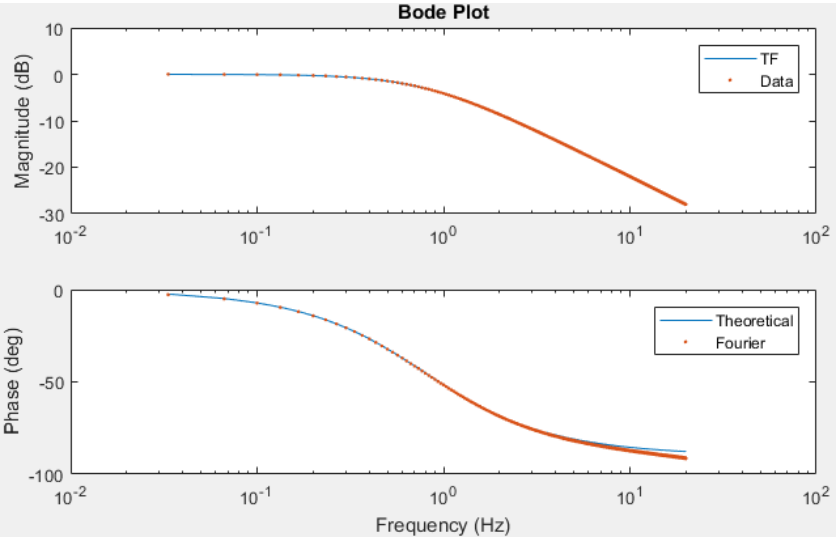


Figure 10.9 The Bode plot derived from using the Fourier transform is compared with the theoretical Bode plot

The code for this example is provided below:

```
close all
clear
clc
dt = 0.001; %(s) time step
t = 0:dt:(30.001-dt); %(s) time vector
N = length(t); % Number of samples
C = 0.001; %(F) capacitance
R = 200; % (Ohm) resistance
a = exp(-dt/C/R); % Parameter for numerical RC circuit
b = 1-a; % Parameter for numerical RC circuit
f0 = 1; %Initial Chirp Frequency
T = 8; % Chirp time interval
r = 1.25; % Chirp frequency sweep rate
V_in = sin(2*pi*f0*(r.^(t-T)-1)/log(r)); %Input Voltage
V_C = zeros(1,N); % Voltage drop across capacitor
V_C(1) = 0; %initial voltage across capacitor
for ii = 1:N-1
  V_C(ii+1) = a*V_C(ii)+b*V_in(ii);
end
figure %plot of voltage arrays versus sample number
plot(t/dt, V_in, t/dt, V_C,'--')
legend('V_i_n','V_C')
xlabel('Sample Number (k)')
ylabel('Voltage')

% Setup for fourier transforms
N_freq = 600; %number of frequencies to plot
MagY_U = zeros(1, N_freq); %Magnitude of V_C/V_in
phaseY_U = zeros(1, N_freq); %Phase angle of V_C/V_in
smallNum = 1e-10; %small number used for logic
for n = 0:N_freq-1
  V_C_Real = 0;
  V_C_Imag = 0;
  Vin_Real = 0;
  Vin_Imag = 0;
  %Get the sum of the real and imaginary parts
  for k = 0:N-1
    V_C_Real = V_C_Real + V_C(k+1)*cos(2*pi*n/N*k);
    V_C_Imag = V_C_Imag - V_C(k+1)*sin(2*pi*n/N*k);
    Vin_Real = Vin_Real + V_in(k+1)*cos(2*pi*n/N*k);
    Vin_Imag = Vin_Imag - V_in(k+1)*sin(2*pi*n/N*k);
  end
  %calculate the magnitudes of V_C and V_in and divide them element-wise
  MagY_U(n+1) = sqrt(V_C_Real^2+V_C_Imag^2)/sqrt(Vin_Real^2+Vin_Imag^2);
  %get the phase angle of V_C and V_in
  phaseV_C = atan2(V_C_Imag, V_C_Real);
  phaseVin = atan2(Vin_Imag, Vin_Real);
  %subtract the angle of V_in from V_C
  %Force it to be between 0 and 2*pi
  if phaseV_C-phaseVin > 0
    phaseY_U(n+1) = (mod((phaseV_C-phaseVin), 2*pi)-2*pi)*180/pi;
  else
```

```
    phaseY_U(n+1) = (phaseV_C-phaseVin)*180/pi;
  end
end
df = 1/(dt*N); % DFT frequency step size
fw = (0:N_freq-1)*df;

%theoretical Bode magnitude and phase angle
Y_U = 5./sqrt((2*pi*fw).^2+5^2);
phi_Y_U = -atan(2*pi*fw/5)*180/pi;

figure %plot the theoretical and fourier Bode plots
subplot(211)
semilogx(fw, 20*log10(Y_U), fw, 20*log10(MagY_U),'.')
ylabel('Magnitude (dB)')
title('Bode Plot')
legend('TF','Data')
subplot(212)
semilogx(fw, phi_Y_U, fw, phaseY_U,'.')
legend('Theoretical','Fourier')
ylabel('Phase (deg)')
xlabel('Frequency (Hz)')
```

Exercises

In this assignment set you will need to access electronic files provided in the online course documents or course assignment.

Exercises for 10 Fourier Transform

P10.1 Perform the following to calculate the discrete Fourier transform of the 8 Hz component of the signal data in the Excel file “DFT_Homework.xlsx" provided online in the course assignment.

(a) (1 pt.) To calculate the Fourier transform, you need to specify the value of N, which is the number of data points included in the discrete Fourier transform. In the signal data file, if you use all the data to calculate the discrete Fourier transform, what is the value of N?

(b) (1 pt.) What is the time step Δt of the data?

(c) (1 pt.) If the desired frequency to be analyzed is $f_n = 8$ Hz, what is the value of n?

(d) (0 pts.) Create a column for $k = 0, \ldots, N-1$.

(e) (1 pt.) Create a column and calculate the real part $\left(x_k \cos\left(2\pi \frac{n}{N} k\right)\right)$ of the discrete Fourier transform.

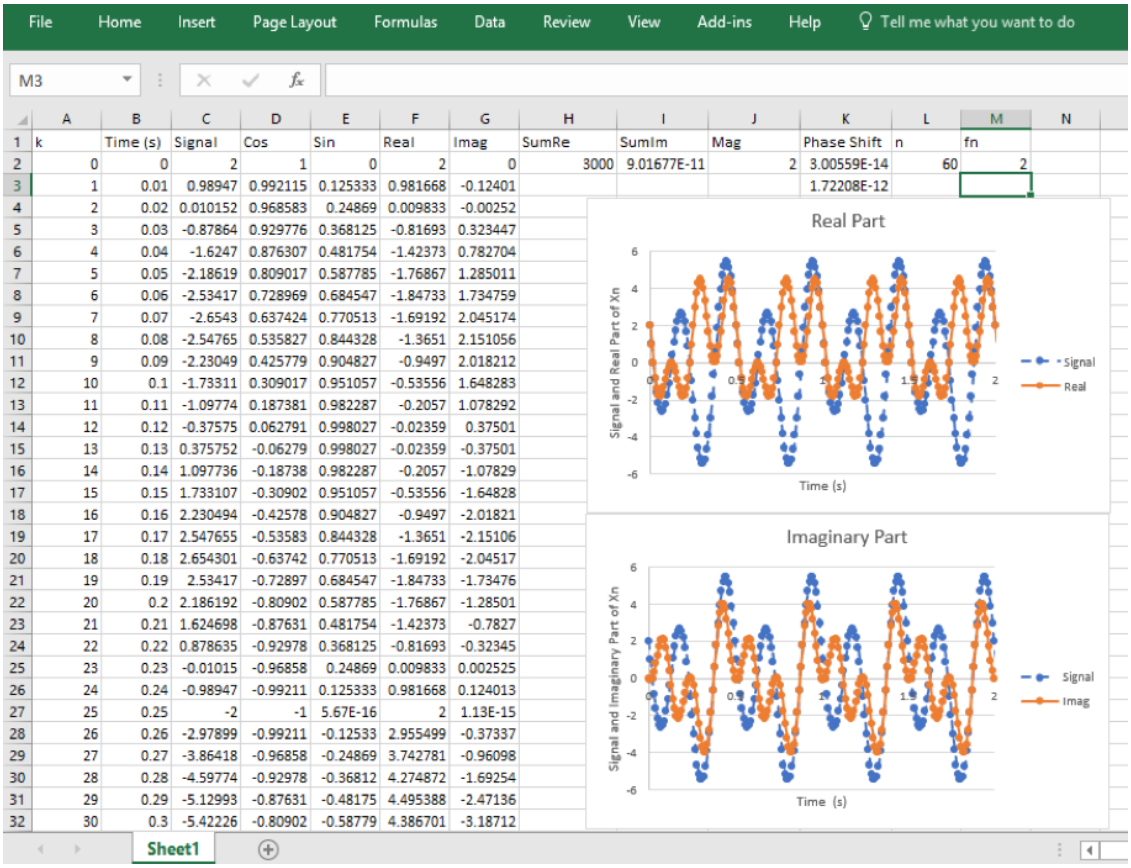
(f) (1 pt.) Create a column and calculate the imaginary part $\left(-x_k \sin\left(2\pi \frac{n}{N} k\right)\right)$ of the discrete Fourier transform.

(g) (1 pt.) Calculate the sum of the real part and the sum of the imaginary part.

(h) (2 pts.) What is the magnitude of the 8 Hz component of the signal?

(i) (2 pts.) What is the phase-shift (in degrees) of the 8 Hz component of the signal?

(j) (2 pts.) Submit a screen shot of your solution. The screenshot should look similar, but not exactly like the following screenshot.



k	Time (s)	Signal	Cos	Sin	Real	Imag	SumRe	SumIm	Mag	Phase Shift	n	fn
0	0	2	1	0	2	0	3000	9.01677E-11	2	3.00559E-14	60	2
1	0.01	0.98947	0.992115	0.125333	0.981668	-0.12401				1.72208E-12		
2	0.02	0.010152	0.968583	0.24869	0.009833	-0.00252						
3	0.03	-0.87864	0.929776	0.368125	-0.81693	0.323447						
4	0.04	-1.6247	0.876307	0.481754	-1.42373	0.782704						
5	0.05	-2.18619	0.809017	0.587785	-1.76867	1.285011						
6	0.06	-2.53417	0.728969	0.684547	-1.84733	1.734759						
7	0.07	-2.6543	0.637424	0.770513	-1.69192	2.045174						
8	0.08	-2.54765	0.535827	0.844328	-1.3651	2.151056						
9	0.09	-2.23049	0.425779	0.904827	-0.9497	2.018212						
10	0.1	-1.73311	0.309017	0.951057	-0.53556	1.648283						
11	0.11	-1.09774	0.187381	0.982287	-0.2057	1.078292						
12	0.12	-0.37575	0.062791	0.998027	-0.02359	0.37501						
13	0.13	0.375752	-0.06279	0.998027	-0.02359	-0.37501						
14	0.14	1.097736	-0.18738	0.982287	-0.2057	-1.07829						
15	0.15	1.733107	-0.30902	0.951057	-0.53556	-1.64828						
16	0.16	2.230494	-0.42578	0.904827	-0.9497	-2.01821						
17	0.17	2.547655	-0.53583	0.844328	-1.3651	-2.15106						
18	0.18	2.654301	-0.63742	0.770513	-1.69192	-2.04517						
19	0.19	2.53417	-0.72897	0.684547	-1.84733	-1.73476						
20	0.2	2.186192	-0.80902	0.587785	-1.76867	-1.28501						
21	0.21	1.624698	-0.87631	0.481754	-1.42373	-0.7827						
22	0.22	0.878635	-0.92978	0.368125	-0.81693	-0.32345						
23	0.23	-0.01015	-0.96858	0.24869	0.009833	0.002525						
24	0.24	-0.98947	-0.99211	0.125333	0.981668	0.124013						
25	0.25	-2	-1	5.67E-16	2	1.13E-15						
26	0.26	-2.97899	-0.99211	-0.12533	2.955499	-0.37337						
27	0.27	-3.86418	-0.96858	-0.24869	3.742781	-0.96098						
28	0.28	-4.59774	-0.92978	-0.36812	4.274872	-1.69254						
29	0.29	-5.12993	-0.87631	-0.48175	4.495388	-2.47136						
30	0.3	-5.42226	-0.80902	-0.58779	4.386701	-3.18712						

Chapter 11

C++ Crash Course 1

Contents

This crash courses for learning C++ teaches the following concepts:

1. How to create, compile, and run a C++ program in Visual Studio
2. How and why to include libraries in your C++ programs
3. How to declare variables and what data types are
4. How to write outputs to the console window
5. How to perform common engineering calculations
6. How to write functions that can return one variable
7. How to use pointers to enable functions to return multiple variables
8. How to create basic arrays and matrices
9. How to perform basic calculations with arrays and matrices

11.0.1 Installing Visual Studio

The C++ Crash Courses in this book use Visual Studio. If needed, download the Community version of Visual Studio for free at the website https://visualstudio.microsoft.com/downloads/ (accessed April 2023). Download the installer by clicking the **Free Download** button on the website. Once the download

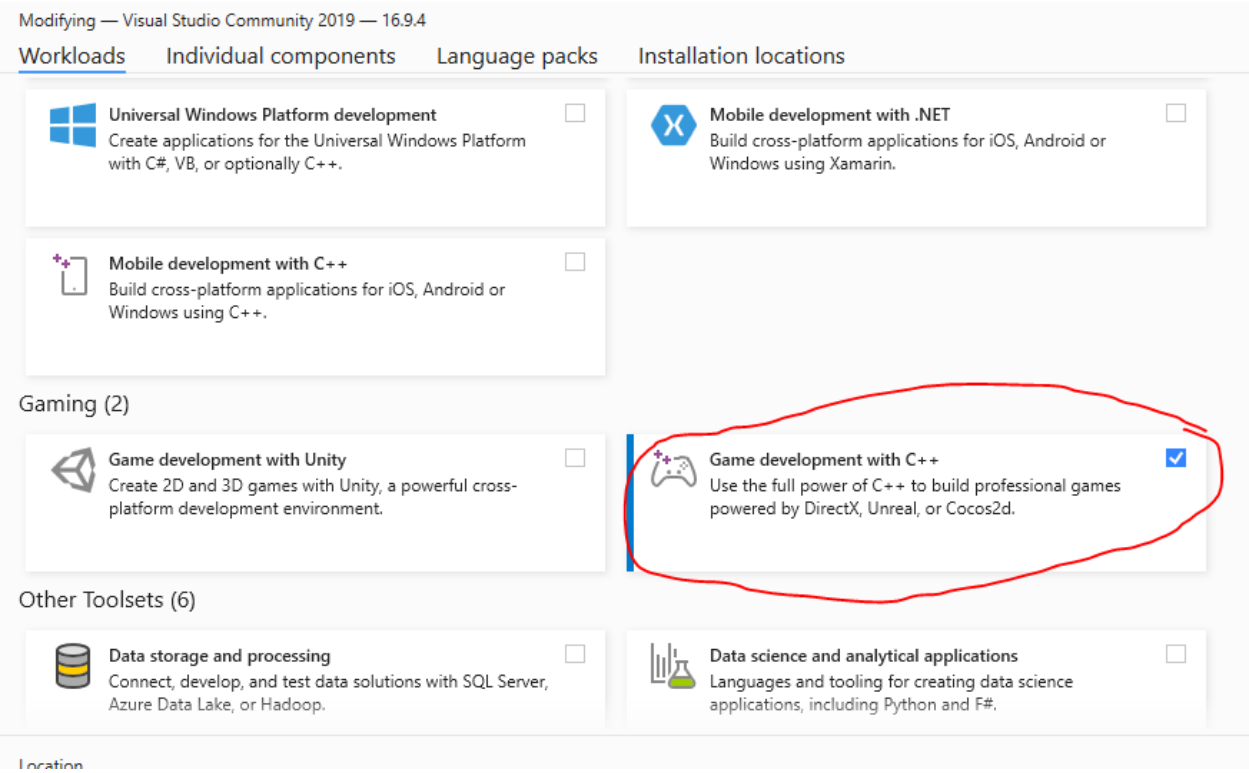


Figure 11.1 When installing Visual C++, select Game development with C++.

completes, launch the installer and follow the instructions until you see the following options shown in Figure 11.1 (it may be for a newer version of Visual Studio than what is shown):

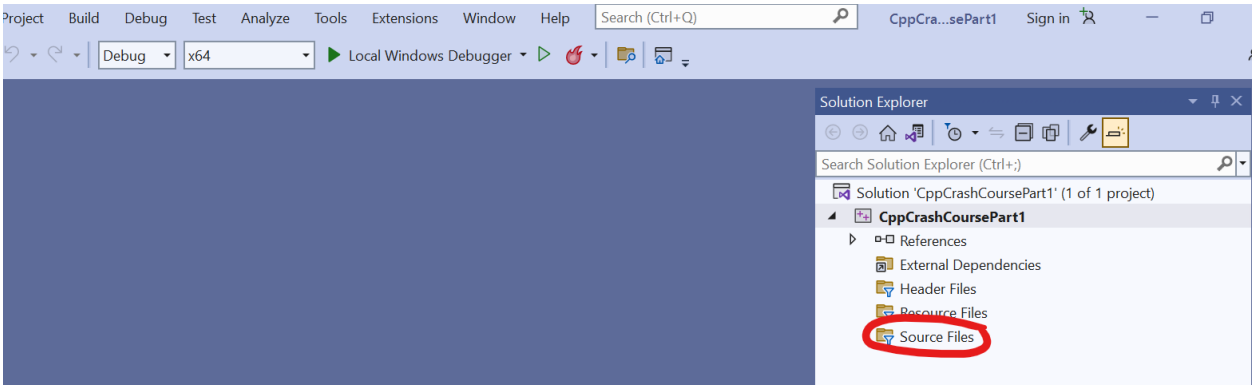
Select Game development with C++ by placing a checkmark in its box. Also make sure that checkmarks are in the same Optional boxes as shown in the figure. If you have already installed Visual Studio, you can modify it by running the Visual Studio Installer. Then click Modify. If you do not see the Modify option, select More, then Modify.

11.0.2 Create, Compile, and Run a C++ Program

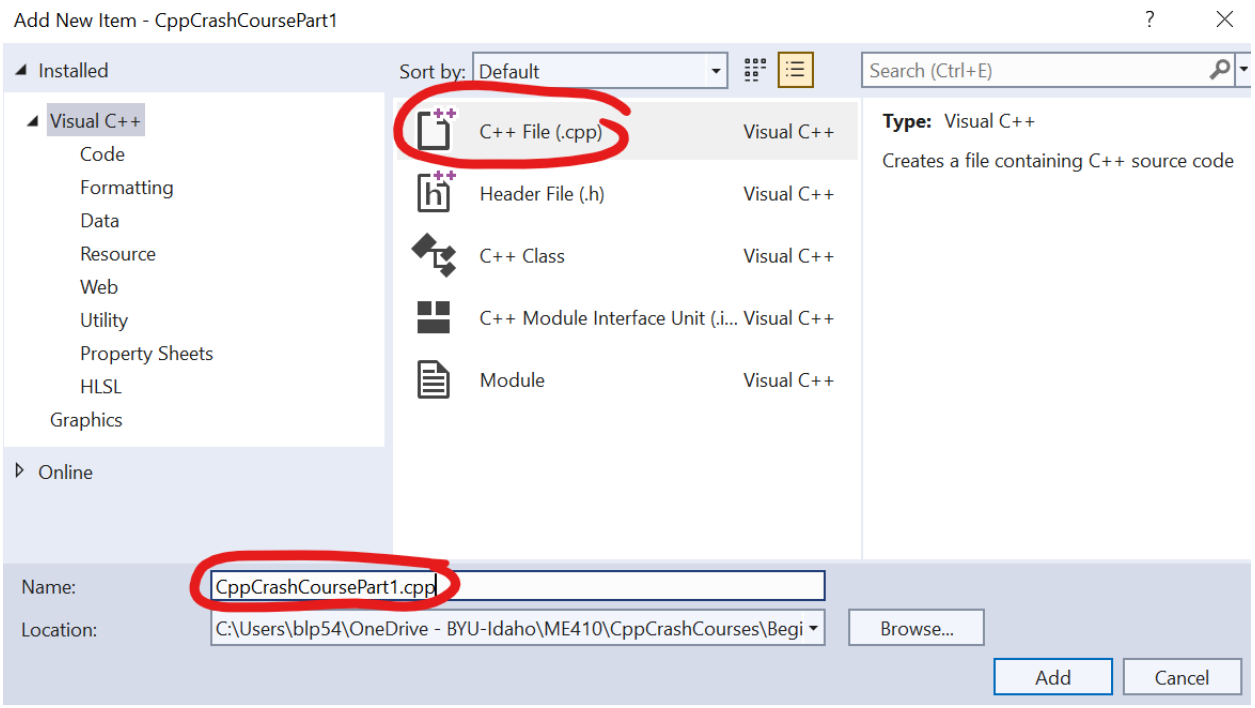
The following steps will walk you through creating, compiling, and running one of the most basic C++ programs: "Hello World". The examples of this chapter were generated using Microsoft Visual Studio Community 2022 (64-bit) Version 17.3.3. Different versions may have different steps.

- Open Visual Studio
- Select Create a new project
- Select Empty Project and click Next
- Click the three dots ... next to the folder location, and navigate to a folder location where you would like to save the project

- Name the project, for example, CppCrashCoursePart1
- Check the box next to Place solution and project in the same directory
- Select Create
- Right click on Source Files in the Solution Explorer window and select Add » New Item



- In the Add New Item window, select C++ File (.cpp), change the Name to (for example) CppCrashCoursePart1.cpp and click Add

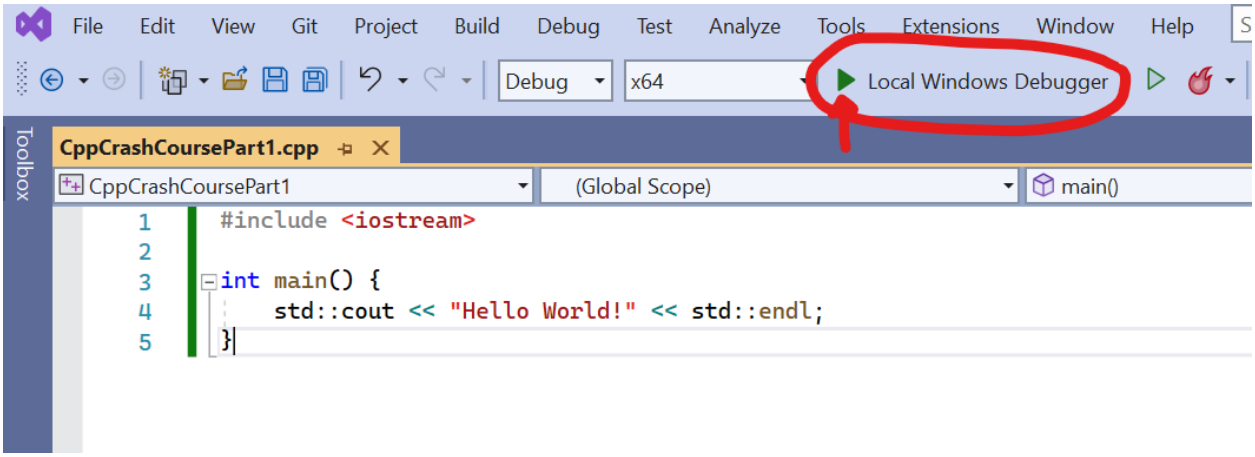


- A new window should open with the CppCrashCoursePart1.cpp file that you just added. (If not, you can open it by double-clicking CppCrashCoursePart1.cpp under Source Files in the Solution Explorer)

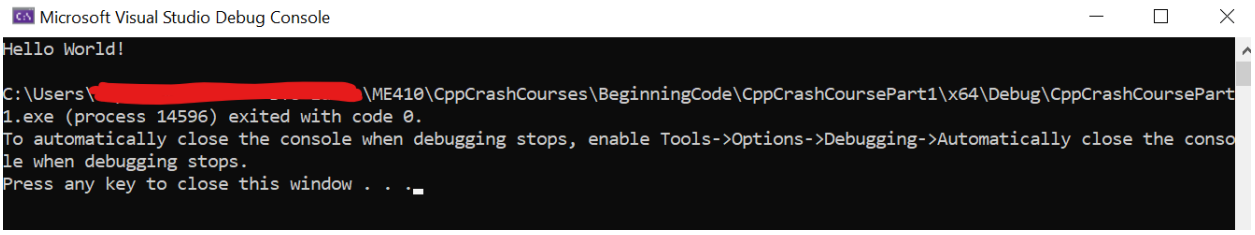
- Instead of copying the code in this assignment, it is recommended that you type it. Typing it helps you learn better, and may prevent compiler errors that are caused by incompatible font formats between this file and Visual Studio.
- Type the following code in the CppCrashCoursePart1.cpp file:

```
#include <iostream> //Add this command to use std::cout and std::endl

int main() { //The MAIN function is the required entry point for a C++ program
  std::cout << "Hello World!" << std::endl; //COUT prints "Hello World!" to the console.
                                            //ENDL moves the cursor to a new line.
  return 0; //The MAIN function should return an integer
}
```

- Then click the green play button next to Local Windows Debugger.



It should compile the code and produce a window that says Hello World! It should look something like



Congratulations! You have written and compiled a C++ program.

Explanation of the Hello World Code

The Hello World code began with a command to include the iostream library:

```
#include <iostream>
```

This library allows the C++ program to use the commands cout and endl. The cout command prints letters, numbers, and other characters to the console window. The endl command is a carriage return that causes the cursor in the console window to move to the next line. A comment in C++ is made using two forward slashes //. Whatever follows the two forward slashes is not part of the compiled code. For example, the phrase //Add this command to use std::cout and std::endl that follows the #include <iostream> command is ignored by the C++ compiler. It is not part of the compiled program. Comments are helpful to let programmers write explanations about the code. The next line is the main() function:

```
int main() {}
```

Each C++ program needs an entry point, or code that is called first. For console programs like the one we created, the main() function is a required entry point. It is the function that C++ calls first. (Later we will create other functions, and they will be called from within the main() function). The code that is within the curly braces is part of the main() function. It is said to be within the *scope* of the main() function. The main() function returns an int data type. The int data type is an integer which is a whole number. Our program could have returned any whole number. In this example, it returned zero using the command

```
return 0;
```

The part of the program that printed Hello World to the console window was the line

```
std::cout << "Hello World!" << std::endl;
```

The cout command prints letters, numbers, and other characters to the console window. The endl command is a carriage return that causes the cursor in the console window to move to the next line. The cout command is part of the std namespace, meaning that cout will not work without using the identifier std:: or by using the command using namespace std. Understanding namespace is a more advanced topic that can be explored later. For now, just know that to use cout and endl, they must be used with the std identifier.

11.0.3 Common Engineering Calculations

Engineers use math. To use many of the basic math operators, C++ code will need to include the library math.h. The math.h library includes at least the following math operations:

```
cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh, atanh, exp, frexp,
ldexp, log, log10, modf, exp2, expm1, ilogb, log1p, log2, logb, scalbn, scalbln, pow, sqrt,
cbrt, hypot, erf, erfc, tgamma, lgamma, ceil, floor, fmod, trunc, round, lround, llround,
rint, lrint, llrint, nearbyint, remainder, remquo, copysign, nan, nextafter, nexttoward,
fdim, fmax, fmin, fabs, abs, fma
```

To learn more about how to use these operators, do an Internet search using the keyword math.h. This section will practice using some of these math.h commands. Modify your CppCrashCoursePart1.cpp file as follows:

```
#include <iostream> //Add this command to use std::cout and std::endl
#include <math.h> //For math operations: sqrt, pow, abs, sin, cos, tan, asin, acos, atan, atan2

using namespace std; //allows us to remove std:: from std::cout and std::endl

int main() { //The MAIN function is the required entry point for a C++ program
  cout << "Hello World!" << endl; //COUT prints "Hello World!" to the console.
                                  //ENDL moves the cursor to a new line.

  //Perform very basic algebra such as add, subtract, multiply, and divide.
  //This does not require math.h
  double sum = 2.0 + 3.0;
  double difference = 2.0 - 3.0;
  double product = 2.0 * 3.0;
  double quotient = 2.0 / 3.0;
  cout << "sum = " << sum << endl; //Print the sum to the console window
  cout << "difference = " << difference << endl; //Print the difference to the console window
  cout << "product = " << product << endl; //Print the product to the console window
  cout << "quotient = " << quotient << endl; //Print the quotient to the console window

  //Perform math that does require the math.h library
  const double pi = 3.14159265358979323846264338327 9;
  double sin2pi = sin(2.0 * pi);
  double atan2_4_n2 = atan2(4.0, -2.0);
  double sixCubed = pow(6.0, 3.0);
  double sqrt_5 = sqrt(5.0);
  cout << "sin(2.0 * pi) = " << sin2pi << endl; //Print the sin of 2*pi to the console window
  cout << "atan2(4.0, -2.0) = " << atan2_4_n2 << endl; //Print the arctangent of 4/(-2)
  cout << "pow(6.0, 3.0) = " << sixCubed << endl; //Print 6^3 to the console window
  cout << "sqrt(5.0) = " << sqrt_5 << endl; //Print the square root of 5 to the console window

  return 0; //The MAIN function should return an integer
```

```
}
```

Click the green play button next to Local Windows Debugger. Did the math operators work correctly? You may want to compare the result with what you get in MATLAB or some other program.

More Data Types in C++

You may have noticed some new keywords in the C++ code above. May of these keywords tell the C++ compiler what type of data it needs to create memory for. The data type double in the phrase double sum = 2.0 + 3.0; indicates that sum is a floating point number with double precision. Floating point numbers have decimal places, unlike integers which are whole numbers. You may have also noticed the sum used the numbers 2.0 and 3.0 instead of 2 and 3. The C++ compiler automatically treats the numbers 2 and 3 like integers, but it treats 2.0 and 3.0 as double precision numbers. When writing code in C++, it can be extremely important to include the decimal on floating point numbers. Forgetting the decimal can lead to unexpected results than are difficult to debug.

Because of the keyword const, the phrase const double pi = 3.14159265358979323846264338327 9; defined pi as a constant that cannot be changed later in the C++ code. If an attempt was made to change pi to a different number later in the program, the compiler would throw an error.

The phrase using namespace std allows the C++ program to use the cout and endl commands directly without the std:: identifier.

11.0.4 Functions that Return One Variable

Functions in C++ require a function declaration, a function definition, and a call to the function. The declaration tells the compiler that the function exists. The definition contains the actual body of the function. The call to the function executes it. The function definition can double-count as the declaration if it is placed in the C++ code before it is called.

Modify your code to include the sind() function as shown below. The sind() function calculates the sine of an angle in degrees. The angle in degrees has a double precision data type, and the function returns a double precision number.

```
#include <iostream> //Add this command to use std::cout and std::endl
#include <math.h> //For math operations: sqrt, pow, abs, sin, cos, tan, asin, acos, atan, atan2

using namespace std; //allows us to remove std:: from std::cout and std::endl
const double pi = 3.14159265358979323846264338327 9;

//function declaration of sind()
double sind(const double angle_degrees);

int main() { //The MAIN function is the required entry point for a C++ program
  cout << "Hello World!" << endl; //COUT prints "Hello World!" to the console.

  //Perform very basic algebra such as add, subtract, multiply, and divide.
```

```
  //This does not require math.h
  double sum = 2.0 + 3.0;
  double difference = 2.0 - 3.0;
  double product = 2.0 * 3.0;
  double quotient = 2.0 / 3.0;
  cout << "sum = " << sum << endl; //Print the sum to the console window
  cout << "difference = " << difference << endl; //Print the difference to the console window
  cout << "product = " << product << endl; //Print the product to the console window
  cout << "quotient = " << quotient << endl; //Print the quotient to the console window

  //Perform math that does require the math.h library
  double sin2pi = sin(2.0 * pi);
  double atan2_4_n2 = atan2(4.0, -2.0);
  double sixCubed = pow(6.0, 3.0);
  double sqrt_5 = sqrt(5.0);
  cout << "sin(2.0 * pi) = " << sin2pi << endl; //Print the sin of 2*pi to the console window
  cout << "atan2(4.0, -2.0) = " << atan2_4_n2 << endl; //Print the arctangent of 4/(-2)
  cout << "pow(6.0, 3.0) = " << sixCubed << endl; //Print 6^3 to the console window
  cout << "sqrt(5.0) = " << sqrt_5 << endl; //Print the square root of 5 to the console window

  //Function call to sind
  double sin_45 = sind(45.0);
  cout << "sin(45 degrees) = " << sin_45 << endl; //Print the result to the console

  return 0; //The MAIN function should return an integer
}

//function definition of sind()
double sind(const double angle_degrees) {
  double angle_radians = angle_degrees * pi / 180.0;
  return (sin(angle_radians));
}
```

Pay attention to the following: First, const double pi = 3.14159265358979323846264338327 9; was moved outside the scope of the main() function. This was necessary so that it could be used in both the main() and sind() functions. Before, pi was only defined within the curly braces {} of the main() function after the line on which it was declared.

Second, notice that the function declaration came before the function call. That allowed the code to place the function definition after the function call. Finally, notice that the sind() function returned the double-precision value calculated by sin(angle_radians).

Do It Yourself

Create your own function named cosd(). It should take an input angle in degrees and output the cosine of the angle. Print the result to the C++ console window.

11.0.5 Functions that Return Multiple Variables

If a C++ function must return multiple variables, it must pass them by reference. C++ uses pointers, denoted with asterisks (*), and the the & operator to pass variables by reference. When passing a variable by reference, instead of passing the variable to the function, the function call passes the memory location of the variable to the function using the & operator. The function uses a pointer (*) to change the variable in the memory location that was passed to it.

Modify your code as follows to create a function that can return multiple variables.

```
#include <iostream> //Add this command to use std::cout and std::endl
#include <math.h> //For math operations: sqrt, pow, abs, sin, cos, tan, asin, acos, atan, atan2

using namespace std; //allows us to remove std:: from std::cout and std::endl
const double pi = 3.14159265358979323846264338327 9;

//function declaration of sind()
double sind(const double angle_degrees);

//This is both the function declaration and definition.  It must come before the function call
void manyOutputs(
double* output1, //first output
double* output2, //second output
const double input1, //first input
const double input2, //second input
const double input3 //third input
) {
  //Access the variable in the memory location of output1 to change it
  *output1 = input1 + input2;

  //Access the varaible in the memory location of output2 to change it
  *output2 = input2 * input3;
}

int main() { //The MAIN function is the required entry point for a C++ program
  cout << "Hello World!" << endl; //Print "Hello World!" to the console.

  //Perform very basic algebra such as add, subtract, multiply, and divide.
  //This does not require math.h
  double sum = 2.0 + 3.0;
  double difference = 2.0 - 3.0;
  double product = 2.0 * 3.0;
  double quotient = 2.0 / 3.0;
  cout << "sum = " << sum << endl; //Print the sum to the console window
  cout << "difference = " << difference << endl; //Print the difference to the console window
  cout << "product = " << product << endl; //Print the product to the console window
  cout << "quotient = " << quotient << endl; //Print the quotient to the console window

  //Perform math that does require the math.h library
  double sin2pi = sin(2.0 * pi);
  double atan2_4_n2 = atan2(4.0, -2.0);
```

```
  double sixCubed = pow(6.0, 3.0);
  double sqrt_5 = sqrt(5.0);
  cout << "sin(2.0 * pi) = " << sin2pi << endl; //Print the sin of 2*pi to the console window
  cout << "atan2(4.0, -2.0) = " << atan2_4_n2 << endl; //Print the arctangent of 4/(-2)
  cout << "pow(6.0, 3.0) = " << sixCubed << endl; //Print 6^3 to the console window
  cout << "sqrt(5.0) = " << sqrt_5 << endl; //Print the square root of 5 to the console window

  //Function call to sind
  double sin_45 = sind(45.0);
  cout << "sin(45 degrees) = " << sin_45 << endl;

  //Function call to manyOutputs()
  double out1, out2;
  manyOutputs(&out1, &out2, 1.0, 2.0, 3.0);
  cout << "out1 = " << out1 << " and out2 = " << out2 << endl;

  return 0; //The MAIN function should return an integer
}

//function definition of sind()
double sind(const double angle_degrees) {
  double angle_radians = angle_degrees * pi / 180.0;
  return (sin(angle_radians));
}
```

Pay attention to how the function call to manyOutputs() passed the memory locations (&out1 and &out2) of the function outputs to the function manyOutputs(). The function referenced the values in the memory locations using the pointer commands *output1 = input1 + input2; and *output2 = input2 * input3;. The function manyOutputs() never used the return() command. That is because it was of data type void. Void functions do not return anything directly, and can only return variables by passing them by reference.

Do It Yourself

Create a function named divideAndMultiply(). The function should have two inputs and two outputs. When calling the function, the two inputs should be the numbers 4 and 5. The two outputs should be the quotient (4/5) and product (4*5) of the inputs. Use the concept of passing by reference. Print the results to the console window.

11.0.6 Basic Arrays and Matrices

This section uses basic arrays and matrices. A later section will provide a more extensive framework for using matrices and arrays. C++ has many ways of working with arrays and matrices. This part of the tutorial teaches one of the most basic ways. Modify your code as follows:

```cpp
#include <iostream> //Add this command to use std::cout and std::endl
#include <math.h> //Math operations: sqrt, pow, abs, sin, cos, tan, asin, acos, atan, atan2

using namespace std; //allows us to remove std:: from std::cout and std::endl
const double pi = 3.14159265358979323846264338327 9;

//function declaration of sind()
double sind(const double angle_degrees);

//This is both the function declaration and definition.  It must come before the function call
void manyOutputs(
double* output1, //first output
double* output2, //second output
const double input1, //first input
const double input2, //second input
const double input3 //third input
) {
  //Access the variable in the memory location of output1 to change it
  *output1 = input1 + input2;

  //Access the varaible in the memory location of output2 to change it
  *output2 = input2 * input3;
}

//Declare the matrixMultiply function C = A * B
void matrixMultiply(double C[2][4], const double A[2][3], const double B[3][4]);

int main() { //The MAIN function is the required entry point for a C++ program
  cout << "Hello World!" << endl; //COUT prints "Hello World!" to the console.
                                  //ENDL moves the cursor to a new line.

  //Perform very basic algebra such as add, subtract, multiply, and divide.
  //This does not require math.h
  double sum = 2.0 + 3.0;
  double difference = 2.0 - 3.0;
  double product = 2.0 * 3.0;
  double quotient = 2.0 / 3.0;
  cout << "sum = " << sum << endl; //Print the sum to the console window
  cout << "difference = " << difference << endl; //Print the difference to the console window
  cout << "product = " << product << endl; //Print the product to the console window
  cout << "quotient = " << quotient << endl; //Print the quotient to the console window

  //Perform math that does require the math.h library
  double sin2pi = sin(2.0 * pi);
  double atan2_4_n2 = atan2(4.0, -2.0);
  double sixCubed = pow(6.0, 3.0);
  double sqrt_5 = sqrt(5.0);
  cout << "sin(2.0 * pi) = " << sin2pi << endl; //Print the sin of 2*pi to the console window
  cout << "atan2(4.0, -2.0) = " << atan2_4_n2 << endl; //Print the arctangent of 4/(-2)
  cout << "pow(6.0, 3.0) = " << sixCubed << endl; //Print 6^3 to the console window
  cout << "sqrt(5.0) = " << sqrt_5 << endl; //Print the square root of 5 to the console window
```

```
  //Function call to sind
  double sin_45 = sind(45.0);
  cout << "sin(45 degrees) = " << sin_45 << endl;

  //Function call to manyOutputs()
  double out1, out2;
  manyOutputs(&out1, &out2, 1.0, 2.0, 3.0);
  cout << "out1 = " << out1 << " and out2 = " << out2 << endl;

  //Define two arrays with three elements each and calculate their cross product
  float vec1[3] = { 1.0f, 2.0f, 3.0f };
  float vec2[3] = { 3.0f, 2.0f, 1.0f };
  float crossProduct[3];
  crossProduct[0] = vec1[1] * vec2[2] - vec2[1] * vec1[2];
  crossProduct[1] = vec1[2] * vec2[0] - vec2[2] * vec1[0];
  crossProduct[2] = vec1[0] * vec2[1] - vec2[0] * vec1[1];
  cout << "cross(vec1, vec2) = [  ";
  for (float& elem : crossProduct) {
    cout << elem << "  ";
  }
  cout << "]" << endl;

  //Multiply a 2x3 matrix by a 3x4 matrix C = A * B
  double A[2][3] = { {1.0,2.0,3.0},{3.0,2.0,1.0} };
  double B[3][4] = { {1.0,2.0,3.0,4.0},{5.0,6.0,7.0,8.0},{9.0,10.0,11.0,12.0} };
  double C[2][4];
  matrixMultiply(C, A, B);
  cout << "C = [ ";
  for (unsigned int ii = 0; ii < 2; ii++) {
    for (int jj = 0; jj < 4; jj++) {
      cout << C[ii][jj] << " ";
    }
    if (ii < 1)
      cout << endl;
    else
      cout << "]" << endl;
  }

  return 0; //The MAIN function should return an integer
}

//function definition of sind()
double sind(const double angle_degrees) {
  double angle_radians = angle_degrees * pi / 180.0;
  return (sin(angle_radians));
}

//Define the matrixMultiply function C = A * B
void matrixMultiply(double C[2][4], const double A[2][3], const double B[3][4]) {
  for (int ii = 0; ii < 2; ii++) {
    for (int jj = 0; jj < 4; jj++) {
      C[ii][jj] = 0.0;
      for (int kk = 0; kk < 3; kk++) {
```

```
        C[ii][jj] += A[ii][kk] * B[kk][jj];
      }
    }
  }
}
```

Pay special attention to how matrices are passed to the function matrixMultiply(). This approach requires the matrix sizes to be included in the function declaration and definition. Passing matrices without specifying their sizes is beyond the scope of this assignment but will be taught in later C++ Crash Courses. The function definition of matrixMultiply() includes three nested for loops. The line C[ii][jj] += A[ii][kk] * B[kk][jj]; is equivalent to C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];.

The for loop for printing the cross product includes the phrase for (float& elem : crossProduct). This phrase is especially useful when the size of the array (crossProduct in this example) is unknown. It iterates through each element of the crossProduct and assigns the element to the floating point value elem. The float data type is for single precision floating point (decimal) numbers.

Do It Yourself

Create a function that can add two 2x2 matrices: A = { {1.0, 2.0}, {3.0, 4.0} } and B = { {5.0, 6.0}, {7.0, 8.0} }. Print the result C = A + B; to the console window.

This concludes the first C++ crash course.

Chapter 12

C++ Crash Course 2

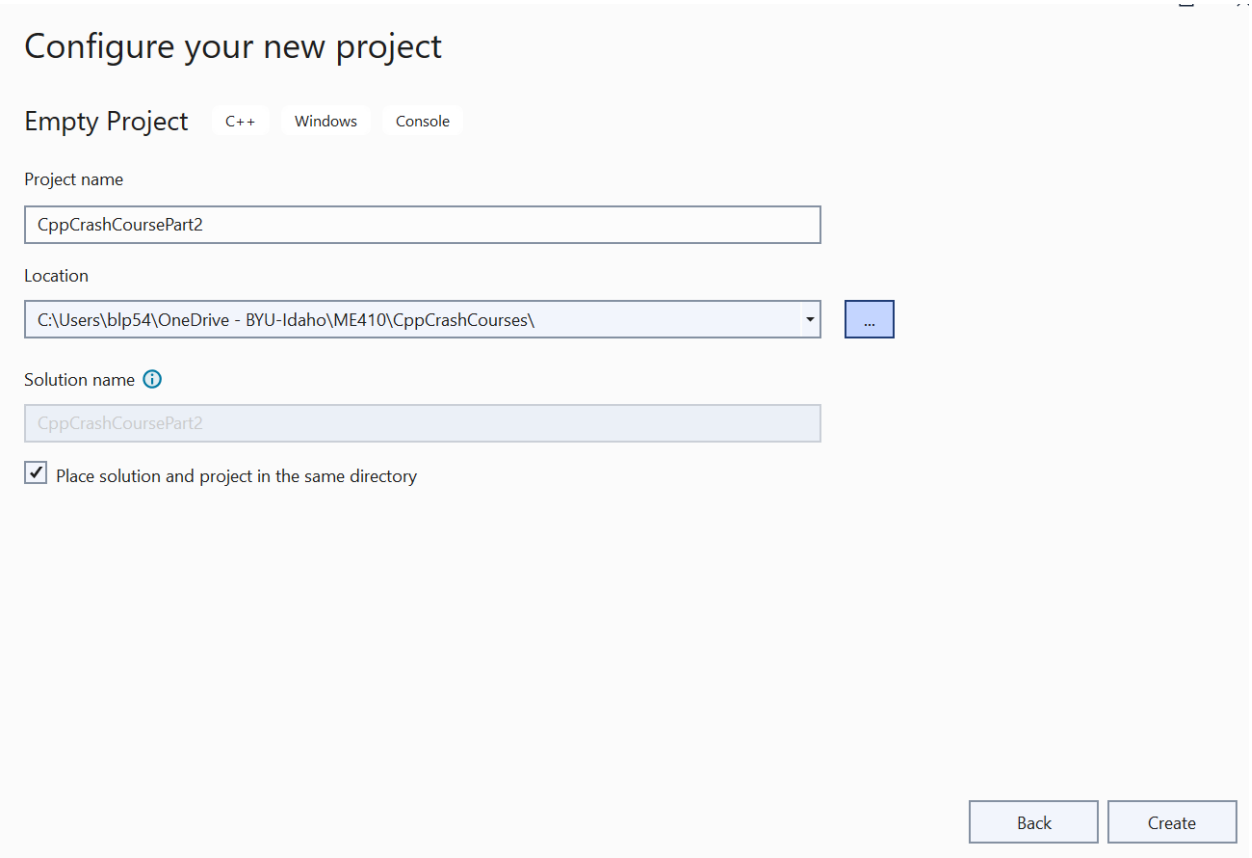
Contents

This is the second C++ crash course. It teaches the following concepts:

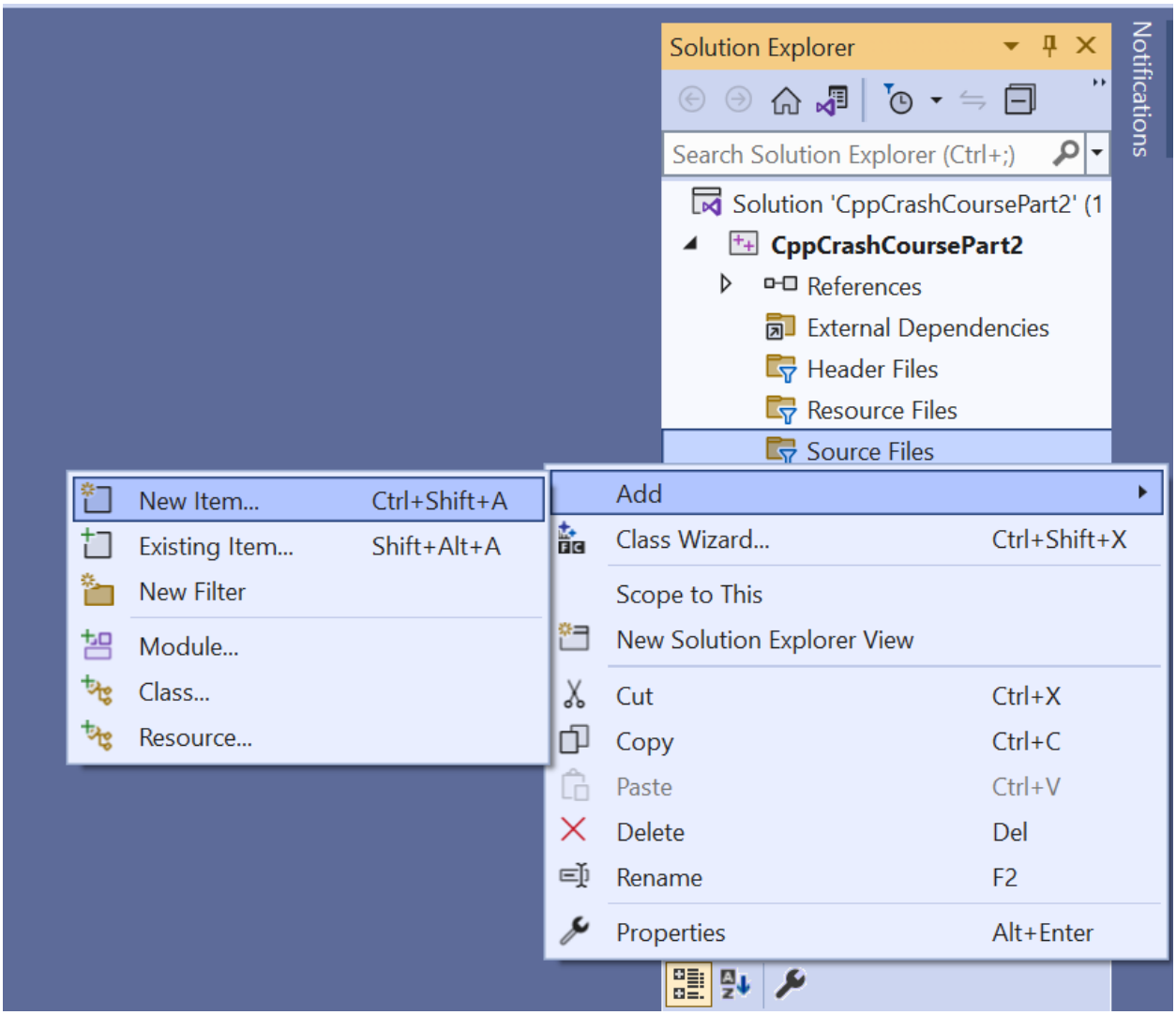
1. How to create, compile, and run a C++ program with multiple source files
2. How header files (.h) and source files (.cpp) differ
3. How to use the vector library when array or matrix sizes are unknown in advance
4. How to perform matrix operations like matrix inversions and matrix exponentials
5. How to solve the state-space equation $\dot{x} = Ax + Bu$

12.0.1 Creating a Program with Multiple Files

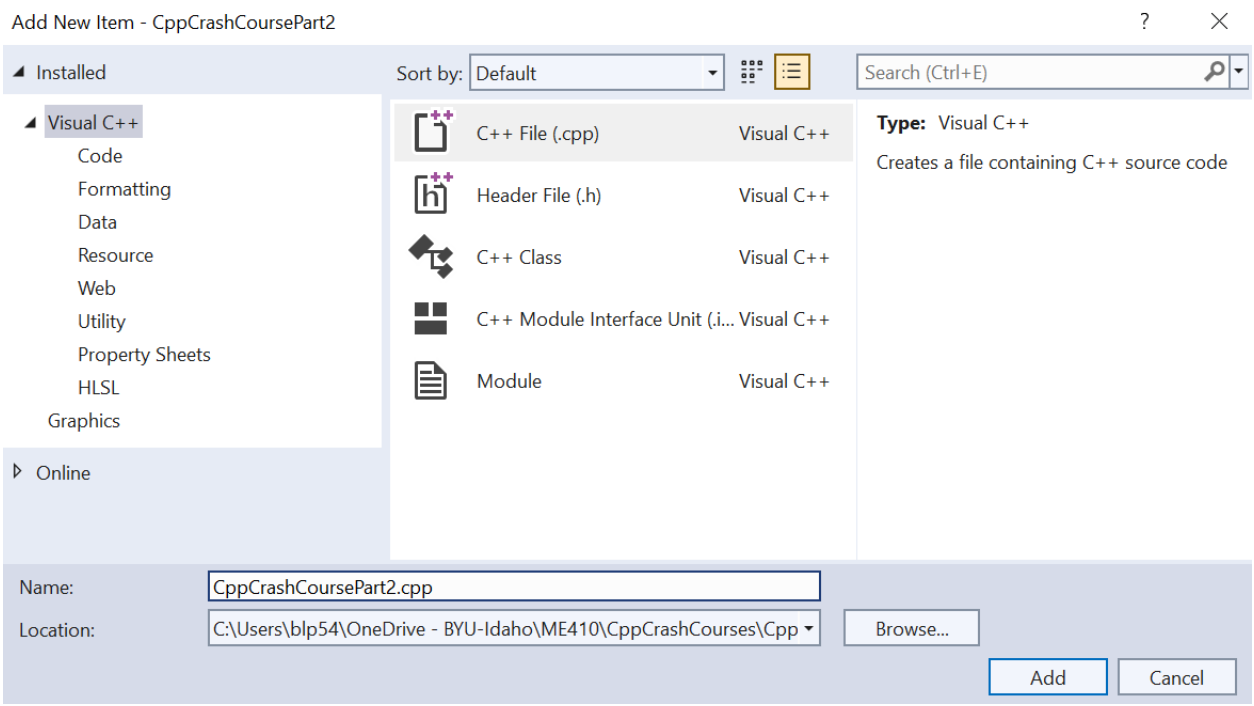
Open Visual Studio and select Create a New Project. Select the Empty Project option, and click Next. Change the Project name to CppCrashCoursePart2, choose the Location, checkmark the box Place solution and project in the same directory and click Create.



Right-click Source Files in the Solution Explorer, then select Add, then select New Item...



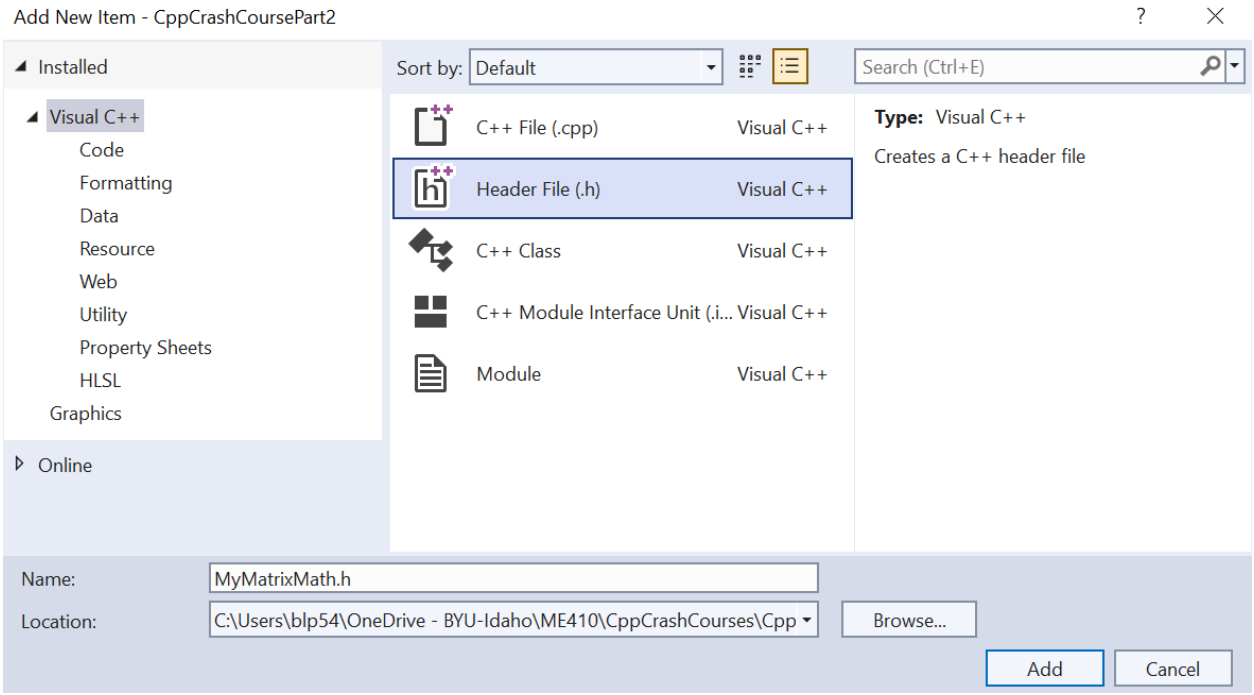
Select C++ File (.cpp), name the file CppCrashCoursePart2.cpp, and click Add.



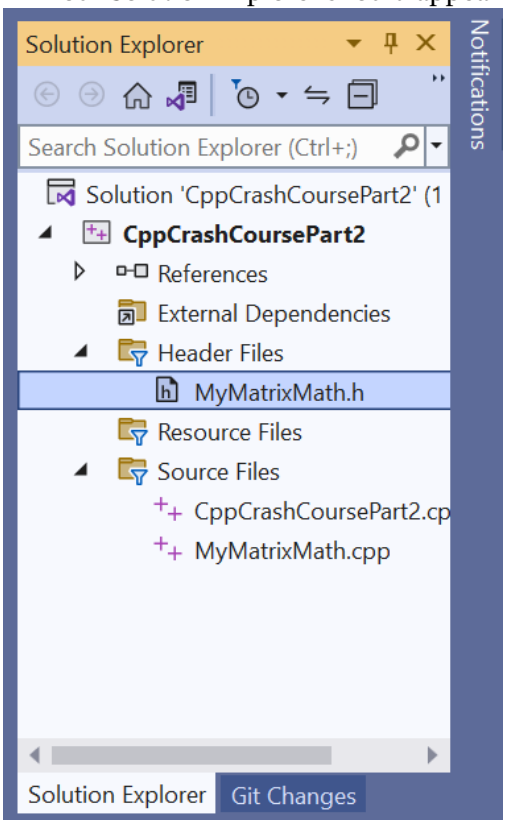
The CppCrashCoursePart2.cpp file will be the main file in our program.

Repeat the steps above to add another source file named MyMatrixMath.cpp.

After the MyMatrixMath.cpp file has been added, right-click Header Files in the Solution Explorer, then select Add, then select New Item... This time, however, select Header File (.h), name the file MyMatrixMath.h, and click Add.



Your Solution Explorer should appear as follows:



If not, please review the previous steps until your Solution Explorer is correct.

12.0.2 Header Files and Source Files

In the first C++ Crash Course, you should have performed basic matrix calculations (add, multiply). Matrix sizes must be known in advance to use the basic matrix calculations of the first C++ Crash Course. The vector library will allow us to generalize our C++ code to work for matrices of any size. Instead of copying the code in this assignment, it is recommended that you type it. Typing it helps you learn better and may prevent compiler errors that are caused by incompatible font formats with Visual Studio.

Open the MyMatrixMath.cpp file by clicking on it in the Solution Explorer. Modify MyMatrixMath.cpp as follows:

MyMatrixMath.cpp

```
#include <iostream> //cout, endl
#include <vector> //vector
#include "MyMatrixMath.h" //Header file we create for matrix math

using namespace std; //To use cout and endl without std::cout and std::endl
```

```
//Function to print the matrix to the console window
extern void printMatrix(vector<vector<double>> A) {
  for (int ii = 0; ii < (int)A.size(); ii++) {
    for (int jj = 0; jj < (int)A[0].size(); jj++) {
      cout << A[ii][jj] << " "; //Print the matrix to the console
    }
    cout << endl; //Go to the next row of the matrix
  }
}
```

Modify the MyMatrixMath.h file as follows:

MyMatrixMath.h

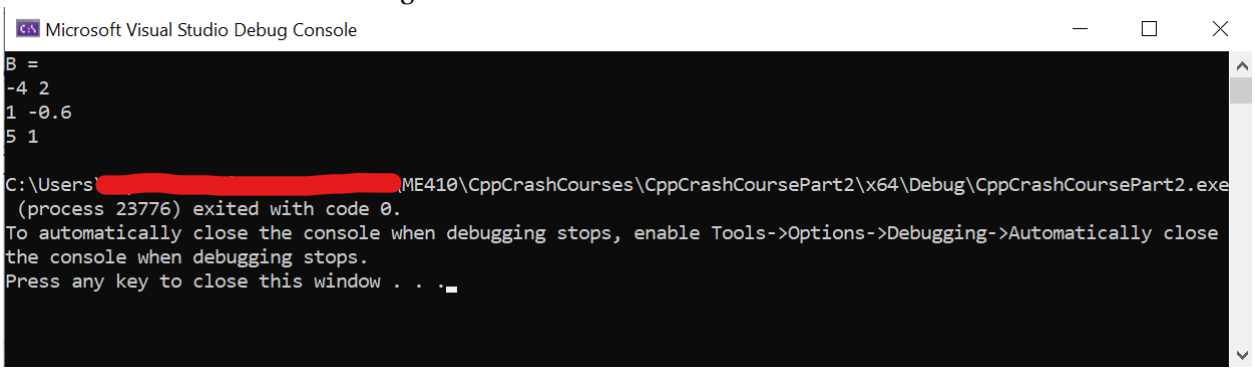
```
#pragma once //ensure MyMatrixMath.h is compiled only once
#include <vector> //vector type

using namespace std; //To use vector without std::vector

// Declare the printMatrix function
extern void printMatrix(vector<vector<double>> A);
```

Modify the CppCrashCoursePart2.cpp file as follows:

CppCrashCoursePart2.cpp

```
#include <iostream> //cout, endl
#include <vector> //vector type
#include "MyMatrixMath.h"  //Include our MyMatrixMath library

using namespace std; //To use cout, endl, and vector without std::

//The program's main function
int main() {
  //Declare and define the 3x2 matrix B
  vector<vector<double>> B{ {-4.0,2.0},{1.0,-0.6},{5.0,1.0} };

  //Print the matrix B using the MyMatrixMath library function
  cout << "B = " << endl;
  printMatrix(B);

  //Return 0 to indicate that the main function was successful
  return 0;
}
```

Click the green play arrow next to Local Windows Debugger to compile and run the code. If there are any compiler errors, check that you have followed all of the steps precisely, including making your Solution Explorer match the one presented above. If the code runs, you should get an output in the console window that is something like this:



This program has three files: two source files (CppCrashCoursePart2.cpp, MyMatrixMath.cpp), and one header file (MyMatrixMath.h). The source file CppCrashCoursePart2.cpp is the main file because it includes the entry function to the program. In this case, the entry function is the int main() function. A program can have multiple source files, but only one can include the program's entry function. The int main() function is one type of entry function. Other types of entry functions exist. For example, Arduino programs use the setup() and loop() functions as entry functions that are compiled with C++ libraries containing source (.cpp) and header (.h) files. Matlab uses specific mex functions as the program's entry functions.

The header file MyMatrixMath.h contains function declarations but no function definitions. As a reminder, function declarations tell the compiler that the function exists. The function definition is the code that is executed when the function is called. Although it is possible to include both the function declarations and definitions in the header file, that is not the common practice. Rather, it is common coding practice to declare functions in a header file but write function definitions in a separate source file. The file MyMatrixMath.h declares to the compiler that the function printMatrix() exists. It lets the compiler know that the output data type of printMatrix() is extern void. The keyword extern means that the function printMatrix() is available to be used by other files that include the MyMatrixMath.h library, and void means that it does not return any outputs.

The source file MyMatrixMath.cpp purposefully shares the same name as its header file. It includes the function definitions for all functions that are declared in the header MyMatrixMath.h file. Any file that will use the MyMatrixMath functions must use the code #include "MyMatrixMath.h" to tell the compiler that the MyMatrixMath functions exits.

The vector Library

The vector library is very helpful when array and matrix sizes could vary. The two-dimensional matrix A is the input argument to the printMatrix() function in MyMatrixMath.cpp. Its data type is vector<vector<double»>. More precisely, A is a vector of vectors, i.e., A is a vector in which each element is a vector of double precision numbers. The code (int)A.size() returns the number of rows in the A matrix as an integer. The code (int)A[0].size() returns the number of columns. With information about the number of rows and columns, the printMatrix() function can write two-dimensional matrices of any size to the

console. In addition to size(), the vector library has many other useful functions. You can learn more about them at this website: https://cplusplus.com/reference/vector/vector/ (accessed February 2023).

12.0.3 The MyMatrixMath Library

Modify your (.cpp) and (.h) files as shown in the code below. Try to understand how each function works.

MyMatrixMath.h

```
#pragma once
#include <vector> //vector type

using namespace std; //To use vector without std::vector

// Declare the printMatrix function
extern void printMatrix(vector<vector<double>> A);

// Function to multiply two matrices of any compatible sizes
extern vector<vector<double>> MatrixMultiply(const vector<vector<double>> A,
  const vector<vector<double>>);

//Function that returns the transpose of a matrix of any size C=transpose(A)
extern vector<vector<double>> transpose(const vector<vector<double>> A);

//Function that creates an nxn identity matrix I
extern vector<vector<double>> Identity(int n);

//This function splits the matrix "A" into two parts: rows above "index"
// are untouched, rows below "index" are searched for the row with the
// largest value in the "index" column.  The row having the largest value
// in the "index" column is moved up to the "index" row.  The other
// searched rows are moved below it.
extern vector<vector<double>> LeftInverseSolve(const vector<vector<double>> A,
  const vector<vector<double>> B);

//This function is part of the row-reduction matrix solver that
// can create an upper triangular matrix.
// It multiplies one row by a factor and adds it to another
// to eliminate the value in "startCol" from the "ReduceRow"
extern vector<vector<double>> ReduceOneRow(const vector<vector<double>> A,
  const int KeepRow, const int startCol, const int ReduceRow);

//This function uses Gaussian Elimination to solve a matrix
// inversion problem that is given in Reduced Row Echelon form
//The A matrix must be in row echelon form [u,B] where
// u is an nxn upper triangular matrix and B is an nxp solution matrix
extern vector<vector<double>> SolveRowEchelonMatrix(
  const vector<vector<double>> A);

//This function solves the matrix problem A*x = B for x.
```

```
// It uses Gaussian elimination to find x = A^-1*B and returns x
// as "SolvedMat"
extern vector<vector<double>> LeftInverseSolve(
  const vector<vector<double>> A, const vector<vector<double>> B);

//This function calculates the inverse of the matrix A by solving the
// equation A*x=I for x = A^-1*I where I is the identity matrix
extern vector<vector<double>> Inverse(const vector<vector<double>> A);

//This function uses the power law to calculate the matrix exponential of A
// expm(A) = I + A + A^2/2 + A^3/3! + A^4/4! + ...
extern vector<vector<double>> expm(const vector<vector<double>> A);

//This function multiplies a matrix by a scalar B = scalar*A
extern vector<vector<double>> MatrixScalarMult(const vector<vector<double>> A,
  const double scalar);

//This function adds two matrices of the same size C = A+B
extern vector<vector<double>> MatrixAdd(const vector<vector<double>> A,
  const vector<vector<double>> B);
```

MyMatrixMath.cpp

```
#include <iostream> //cout, endl
#include <vector> //vector
#include "MyMatrixMath.h" //Header file we create for matrix math

using namespace std; //To use cout and endl without std::cout and std::endl

//Function to print the matrix to the console window
extern void printMatrix(vector<vector<double>> A) {
  for (int ii = 0; ii < (int)A.size(); ii++) {
    for (int jj = 0; jj < (int)A[0].size(); jj++) {
      cout << A[ii][jj] << " "; //Print the matrix to the console
    }
    cout << endl; //Go to the next row of the matrix
  }
}

// Function to multiply two matrices of any compatible sizes
extern vector<vector<double>> MatrixMultiply(
  const vector<vector<double>> A, //INPUT: C = A*B
  const vector<vector<double>> B //INPUT: C= A*B
  ) {

  //Declare the output matrix C and give it the correct dimensions
  vector<vector<double>> C(A.size(), vector<double>(B[0].size()));
```

```
  //check that matrix inner dimensions are compatible
  if (A[0].size() != B.size()) {
    //The matrices are not compatible
    cout << "Error: Number of columns of A must equal number of rows of B for C = A*B" << endl;
    throw 1; //Throw an error
  }

  //Perform the matrix multiplication
  for (int ii = 0; ii < A.size(); ii++) {
    for (int jj = 0; jj < B[0].size(); jj++) {
      C[ii][jj] = 0.0;
      for (int kk = 0; kk < A[0].size(); kk++) {
        C[ii][jj] += A[ii][kk] * B[kk][jj];
      }
    }
  }

  //Return the matrix C=A*B
  return C;
}

//Function that returns the transpose of a matrix of any size C=transpose(A)
extern vector<vector<double>> transpose(
  const vector<vector<double>> A
  ) {

  //Declare the output matrix C and give it the correct dimensions
  vector<vector<double>> C(A[0].size(), vector<double> (A.size()));

  //Transpose the matrix by swapping rows with columns
  for (int ii = 0; ii < C.size(); ii++) {
    for (int jj = 0; jj < C[0].size(); jj++) {
      C[ii][jj] = A[jj][ii];
    }
  }

  //Return the transposed matrix C=transpose(A)
  return C;
}

//Function that creates an nxn identity matrix I
extern vector<vector<double>> Identity(int n) {

  //Declare the variable I and set its dimensions to be nxn
  vector<vector<double>> I(n, vector<double>(n, 0.0)); //initialize a nxn matrix to all 0.0
  for (int ii = 0; ii < n; ii++) {
    //Set the diagonal elements to be ones
    I[ii][ii] = 1.0;
  }

  //Return the nxn identity matrix
  return I;
}
```

```
//This function splits the matrix "A" into two parts: rows above "index"
// are untouched, rows below "index" are searched for the row with the
// largest value in the "index" column.  The row having the largest value
// in the "index" column is moved up to the "index" row.  The other
// searched rows are moved below it.
extern vector<vector<double>> SortLargestRowToTop(
  const vector<vector<double>> A,
  const int index
  ) {

  int n = (int)A.size(); //Number of rows in A
  int m = (int)A[0].size(); //Number of columns in A

  //Declare the output matrix C, and set its dimensions
  // to the dimensions of A
  vector<vector<double>> C(n, vector<double>(m));

  //Only search rows whose indices are larger than "index"
  int indexMax = index;
  //find the row in "A" with the largest value in the "index" column
  for (int ii = index+1; ii < n; ii++) {
    double absA = abs(A[ii][index]);
    double maxVal = abs(A[indexMax][index]);
    if (absA > maxVal) {
      //A larger value has been found, update the max row index
      indexMax = ii;
    }
  }

  //Move the largest to the "index" row
  for (int ii = 0; ii < n; ii++) {
    for (int jj = 0; jj < m; jj++) {
      if (ii < index)
        //Leave the row untouched if its index is smaller
        // than the "index" row
        C[ii][jj] = A[ii][jj];
      else if (ii < indexMax)
        //If a searched row had an "index" value smaller than the
        // largest value in the "index" column, move it below the
        // "index" row
        C[ii + 1][jj] = A[ii][jj];
      else if (ii == indexMax)
        //Make the row with the largest value in the "index" column
        // the "index" row
        C[index][jj] = A[ii][jj];
      else
        //These rows were already below the "index" row, and they
        // had smaller values than the largest value in the "index"
        // column.  They do not need to be moved.
        C[ii][jj] = A[ii][jj];
    }
  }
```

```
  //Return the sorted matrix
  return C;
}

//This function is part of the row-reduction matrix solver that
// can create an upper triangular matrix.
// It multiplies one row by a factor and adds it to another
// to eliminate the value in "startCol" from the "ReduceRow"
extern vector<vector<double>> ReduceOneRow(
  const vector<vector<double>> A,
  const int KeepRow,
  const int startCol,
  const int ReduceRow
  ) {

  //Find the factor to eliminate the value in "startCol" from "ReduceRow"
  double factor = -A[ReduceRow][startCol] / A[KeepRow][startCol];
  vector<vector<double>> C = A; //Set the reduced matrix to be the A matrix
  for (int jj = startCol; jj < A[KeepRow].size(); jj++) {
    //Remove the value in "startCol" from "ReduceRow"
    C[ReduceRow][jj] = factor * A[KeepRow][jj] + A[ReduceRow][jj];
  }

  //Return the reduced row
  return C;
}

//This function uses Gaussian Elimination to solve a matrix
// inversion problem that is given in Reduced Row Echelon form
//The A matrix must be in row echelon form [u,B] where
// u is an nxn upper triangular matrix and B is an nxp solution matrix
extern vector<vector<double>> SolveRowEchelonMatrix(
  const vector<vector<double>> A
  ) {

  int n = (int)A.size(); //Size of the square matrix
  int m = (int)A[0].size(); //# of columns of Row Ecehlon Matrix
  int p = m - n; //# of columns of the solution matrix

  //Check that solution matrix has at least one column
  if (p < 1) {
    cout << "Matrix Size Issue" << endl;
    throw 1;
  }

  //separate the augmented matrix into its upper triangular and
  //solution form
  vector<vector<double>> u(n, vector<double>(n)); //nxn upper triangular matrix
  vector<vector<double>> B(n, vector<double>(p)); //nxp solution matrix

  //Extract the upper triangular matrix u from the row echelon matrix A=[u,B]
  for (int ii = 0; ii < n; ii++)
```

```
    for (int jj = 0; jj < n; jj++)
      u[ii][jj] = A[ii][jj];

  //Extract the solution matrix B from the row echelon matrix A=[u,B]
  for (int ii = 0; ii < n; ii++)
    for (int jj = 0; jj < p; jj++)
      B[ii][jj] = A[ii][jj + n];

  //Declare the matrix C = u^(-1)*B, and set its dimensions
  vector<vector<double>> C(n, vector<double>(p));

  //Start at the bottom row, use back-substitution to solve for each value of C
  //This is the heart of the algorithm
  //(https://en.wikipedia.org/wiki/Gaussian_elimination)
  for (int ii = 0; ii < n; ii++) {
    int aa = n - ii - 1;
    for (int jj = 0; jj < p; jj++) {
      C[aa][jj] = B[aa][jj];
      for (int kk = aa+1; kk < n; kk++) {
        C[aa][jj] -= u[aa][kk] * C[kk][jj];
      }
      C[aa][jj] /= u[aa][aa];
    }
  }

  //Return the solution C=u(^-1)*B
  return C;
}

//This function solves the matrix problem A*x = B for x.
// It uses Gaussian elimination to find x = A^-1*B and returns x
// as "SolvedMat"
extern vector<vector<double>> LeftInverseSolve(
  const vector<vector<double>> A,
  const vector<vector<double>> B
  ) {

  //Make sure that the matrix sizes are compatible
  if (A.size() != A[0].size() ||
      A.size() != B.size()) {
    cout << "Error: Matrix size problems" << endl;
    throw 1;
  }

  //Get the matrix sizes.  A is nxn, B is nxm
  int n = (int)A.size();
  int m = n + (int)B[0].size();

  //Put together a big matrix BigMat = [A,B]
  // that will eventually be reduced to row echelon form [u,b]
  // where u is an upper triangular matrix and b is a solution matrix
  vector<vector<double>> BigMat(B.size(), vector<double>(m));
  for (int ii = 0; ii < n; ii++) {
```

```
    for (int jj = 0; jj < m; jj++) {
      if (jj < n)
        BigMat[ii][jj] = A[ii][jj];
      else
        BigMat[ii][jj] = B[ii][jj - n];
    }
  }

  //Pivot the row with the largest first element to the top of the big matrix
  vector<vector<double>> SortedMat = SortLargestRowToTop(BigMat, 0);

  //Perform Gaussian elimination with partial pivoting
  for (int ii = 0; ii < n-1; ii++)    {
    for (int jj = ii+1; jj < n; jj++) {
      //Row reduction
      SortedMat = ReduceOneRow(SortedMat, ii, ii, jj);
    }
    //Row pivoting
    SortedMat = SortLargestRowToTop(SortedMat, ii + 1);
  }

  //Use back substitution to solve the [u,b] matrix
  vector<vector<double>> SolvedMat = SolveRowEchelonMatrix(SortedMat);

  //Return the solution: SolvedMat = A^-1*B
  return SolvedMat;
}

//This function calculates the inverse of the matrix A by solving the
// equation A*x=I for x = A^-1*I
// where I is the identity matrix
extern vector<vector<double>> Inverse(const vector<vector<double>>A) {
  return LeftInverseSolve(A, Identity((int)A.size()));
}

//This function uses the power law to calculate the matrix exponential of A
// expm(A) = I + A + A^2/2 + A^3/3! + A^4/4! + ...
extern vector<vector<double>> expm(const vector<vector<double>>A) {
  //Get the size of the square A matrix
  int n = A.size();
  int m = A[0].size();
  if (m != n) {
    cout << "Matrix must be square " << endl;
    throw 1;
  }

  //Start with the identity matrix
  vector<vector<double>> eAin = Identity(n);

  //Set the maximum number of elements in the power law summation
  const int maxIter = 50;

  //Declare a temporary matrix
```

```
  vector<vector<double>> eAout(n, vector<double>(n));

  //Start with the identity matrix
  vector<vector<double>> expA = Identity(n);

  for (int kk = 1; kk < maxIter; kk++) {
    //eAout = eAin * A = A^kk / (kk-1)!
    eAout = MatrixMultiply(eAin, A);
    for (int ii = 0; ii < n; ii++) {
      for (int jj = 0; jj < n; jj++) {
        //eAin = eAout / kk = A^kk / kk!
        eAin[ii][jj] = eAout[ii][jj] / ((double)kk);
        //expA = expA + A^kk / kk!
        expA[ii][jj] += eAin[ii][jj]; //expm(A)
      }
    }
  }

  //Return the matrix exponential of A
  return expA;
}

//This function multiplies a matrix by a scalar B = scalar*A
extern vector<vector<double>> MatrixScalarMult(const vector<vector<double>> A,
  const double scalar) {

  int n = A.size();
  int m = A[0].size();
  vector<vector<double>> B(n, vector<double>(m));

  //Multiply each element of the matrix by the scalar
  for (int ii = 0; ii < n; ii++)
    for (int jj = 0; jj < m; jj++)
      B[ii][jj] = scalar * A[ii][jj];

  return B;
}

//This function adds two matrices of the same size C = A+B
extern vector<vector<double>> MatrixAdd(const vector<vector<double>> A,
  const vector<vector<double>> B) {

  //Declare the output matrix C and give it the correct dimensions
  vector<vector<double>> C(A.size(), vector<double>(A[0].size()));

  //check that matrix inner dimensions are compatible
  if (A.size() != B.size() || A[0].size() != B[0].size()) {
    //The matrices are not compatible
    cout << "Error: Matrix Sizes for C=A+B are Incompatible" << endl;
    throw 1; //Throw an error
  }

  //Add the matrices element-by-element
```

```
  for (int ii = 0; ii < C.size(); ii++)
    for (int jj = 0; jj < C[0].size(); jj++)
      C[ii][jj] = A[ii][jj] + B[ii][jj];

  //Return the result C=A+B
  return C;
}
```

Wow! Creating a C++ matrix math library is a lot of work!

12.0.4 Solving $\dot{x} = Ax + Bu$

The MyMatrixMath library can help solve dynamic equations like $\dot{x} = Ax + Bu$. To highlight this, consider the state-space system

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \tag{12.1}$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} x \tag{12.2}$$

with the initial condition

$$x(0) = \begin{bmatrix} -2 \\ 1 \end{bmatrix} \tag{12.3}$$

the time-step

$$\Delta t = 0.1 \tag{12.4}$$

and the input sequence

$$u = \begin{bmatrix} 1,2,3,4,5 \end{bmatrix} \tag{12.5}$$

Example code is provided in the CppCrashCoursePart2.cpp file below to solve it.

CppCrashCoursePart2.cpp

```
#include <iostream> //cout, endl
#include <vector> //vector type
#include "MyMatrixMath.h"  //Include our MyMatrixMath library

using namespace std; //To use cout, endl, and vector without std::

//The program's main function
int main() {

  //Declare and define the A matrix
  vector<vector<double>> A{ {0.0,1.0},{-1.0,-2.0} };

  //Declare and define the B matrix
```

```
vector<vector<double>> B{ {0.0},{1.0} };

//Declare and define the C matrix
vector<vector<double>> C{ {1.0,0.0} };

//Declare and define the u vector
vector<double> u{ 1.0,2.0,3.0,4.0,5.0 };

//Declare and define the initial condition
vector<vector<double>> x{ {-2.0},{1.0} };

//Declare and define the time-step
double dt = 0.1;

//Create a temporary vector Adt = A*dt
vector<vector<double>> Adt = MatrixScalarMult(A, dt);

//Create a temporary vector Bdt = B*dt
vector<vector<double>> Bdt = MatrixScalarMult(B, dt);

//Create a temporary vector F = [A*dt,B*dt;zeros(1,3)];
vector<vector<double>> F(3, vector<double>(3));
for (int ii = 0; ii < 3; ii++) {
  for (int jj = 0; jj < 3; jj++) {
    if (ii < 2)
      if (jj < 2)
        F[ii][jj] = Adt[ii][jj];
      else
        F[ii][jj] = Bdt[ii][0];
    else
      F[ii][jj] = 0.0;
  }
}

//Declare the Fd matrix and give it the correct dimensions 3x3
vector<vector<double>> Fd(3, vector<double>(3));

//Calculate the Fd matrix: Fd = expm([A*dt, B*dt; zeros(1,3)])
Fd = expm(F);

//Declare the Ad matrix and give it the correct dimensions 2x2
vector<vector<double>> Ad(2, vector<double>(2));

//Extract the Ad matrix from the Fd Matrix
for (int ii = 0; ii < 2; ii++)
  for (int jj = 0; jj < 2; jj++)
    Ad[ii][jj] = Fd[ii][jj];

//Declare the Bd matrix and give it the correct dimensions 2x1
vector<vector<double>> Bd(2, vector<double>(1));

//Extract the Bd matrix from the Fd Matrix
for (int ii = 0; ii < 2; ii++)
```

```
    for (int jj = 0; jj < 1; jj++)
      Bd[ii][jj] = Fd[ii][jj+2];

  //Allocate memory (5 elements) for the output
  vector<vector<double>> y(1, vector<double>(5));

  //Solve the state and output equations
  for (int ii = 0; ii < u.size(); ii++) {
    //Declare a temporary matrix so the output data type is correct
    vector<vector<double>> yii(1, vector<double>(1));

    //Solve the output equation
    yii = MatrixMultiply(C, x);
    y[0][ii] = yii[0][0];

    //Solve the state equation
    x = MatrixAdd(MatrixMultiply(Ad, x), MatrixScalarMult(Bd, u[ii]));
  }

  //Print the output y
  cout << "y = " << endl;
  printMatrix(y);

  //Return 0 to indicate that the main function was successful
  return 0;
}
```

If everything was coded and set up correctly, the output to the console window is

The code below shows how the same problem would be solved in MATLAB.

```
clear
A = [0,1;-1,-2];
B = [0;1];
C = [1,0];
u = 1:5;
dt = 0.1;
x = [-2;1];
Fd = expm([A*dt,B*dt;zeros(1,3)]);
Ad = Fd(1:2,1:2);
Bd = Fd(1:2,3);
for ii = 1:5
  y(ii) = C*x;
  x = Ad*x+Bd*u(ii);
end
y
```

The MATLAB program used far fewer lines of code to solve the same state-space system. Did it give the same answer?

Do It Yourself

Create a new function named getFd() in the MyMatrixMath.cpp file that takes the matrices A and B and the time-step dt and returns the Fd matrix. The function should work for any correctly sized A and B matrices. Then, in the CppCrashCoursePart2.cpp file, write code that uses your new function to solve the following state-space system:

$$\dot{x} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -6 & -7 & -8 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u \tag{12.6}$$

$$y = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} x + 0.8u \tag{12.7}$$

with the initial condition

$$x(0) = \begin{bmatrix} -2 \\ -1 \\ 3 \end{bmatrix} \tag{12.8}$$

the time-step

$$\Delta t = 0.1 \tag{12.9}$$

and the input sequence

$$u = \begin{bmatrix} 4, 3, 2, 1, 0, -1 \end{bmatrix} \tag{12.10}$$

If available, use MATLAB to check that your C++ program solved the state-space equations correctly. This is the end of C++ Crash Course 2.

Chapter 13

C++ Crash Course 3

Contents

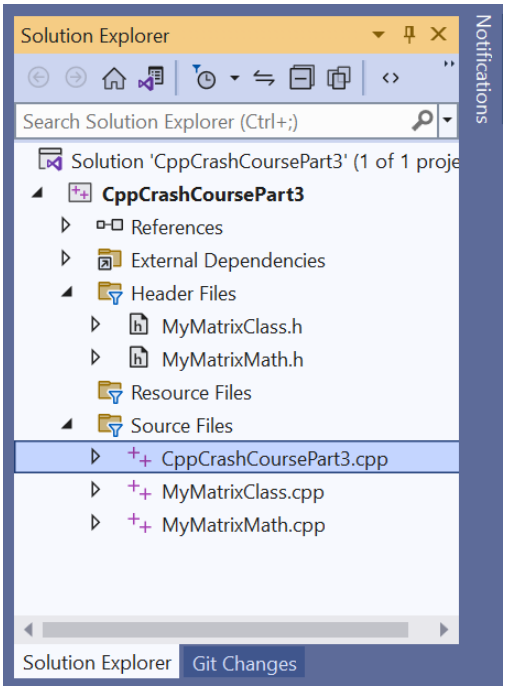
This third C++ Crash Course teaches the following concepts:

1. How to use classes in C++ for object-oriented programming
2. How to overload operators in C++ classes
3. How to create and use class member functions
4. How to solve the state-space system $\dot{x} = Ax + Bu$

This course builds on the previous two C++ crash courses. You will need to use some of the computer code that you generated in the past courses. As you have done in the past, create a new solution in Visual Studio. Add header and source files until your Solution Explorer includes the following files:

- MyMatrixClass.h
- MyMatrixClass.cpp
- MyMatrixMath.h (Same as C++ Crash Course 2)
- MyMatrixMath.cpp (Same as C++ Crash Course 2)
- CppCrashCoursePart3.cpp

Your solution explorer should appear as shown below:



You should have already created the files MyMatrixMath.h and MyMatrixMath.cpp in C++ Crash Course 2. If not, you will need to complete that crash course before doing this one.

13.0.1 Classes in C++

Classes in C++ enable object-oriented programming. Objects have attributes and methods. Methods are also called class member functions. For example, a circle is an object. Its attributes could include its radius, its origin, its fill-color, and its edge-color. A circle's methods could include the equation to calculate its area or the equation to calculate its circumference. Creating a circle class in C++ defines a new data type. As a reminder, double, int, float, char, and vector are examples of data types. Any variables declared by the circle data type would have all the attributes and methods of the circle object that are defined in the circle class.

13.0.2 The MyMatrixClass Class

In this C++ crash course, you will create a new matrix class named MyMatrixClass. MyMatrixClass defines a new data type for matrices. Matrices in MyMatrixClass have the following attributes:

- nRows
- nCols
- myMatrix

The attribute nRows is an int data type that defines the number of rows in the matrix. The int nCols defines the number of columns in the matrix. The attribute myMatrix is of data type vector<vector<double». It contains all the data in the rows and columns of the matrix.

MyMatrixClass objects will have the following methods:

- Constructor
- Destructor
- Matrix addition
- Matrix subtraction
- Matrix multiplication
- Matrix inversion solver
- Matrix-scalar division
- Matrix-scalar multiplication (you create this one)
- Matrix element accessing
- Matrix print
- Matrix transpose (you create this one)

Instead of copying the code in this assignment, it is recommended that you type it. Typing it helps you learn better and may prevent compiler errors that are caused by incompatible font formats with Visual Studio. MyMatrixClass is declared in the MyMatrixClass.h file:

MyMatrixClass.h

```
#pragma once //Tell the compiler to only compile this file once

#include <vector> //vector
using namespace std; //To use vector, cout, and endl without std::

//Declare the MyMatrixClass class
class MyMatrixClass
{
public:
  //Constructor for the MyMatrixClass object that initializes every element
  // of the matrix to init
  MyMatrixClass(int rows, int cols, double init);

  //Alternate constructor for the MyMatrixClass that allows it to
  // be initialized to a specific matrix A
  MyMatrixClass(vector<vector<double>> A);

  //MyMatrixClass destructor to delete the class when the program ends
  ~MyMatrixClass();

  //Overload operators to perform matrix operations more easily
  MyMatrixClass operator + (MyMatrixClass& B); //Matrix+Matrix addition
  MyMatrixClass operator - (MyMatrixClass& B); //Matrix-Matrix subtraction
```

```
  MyMatrixClass operator * (MyMatrixClass& B); //Matrix*Matrix multiplication
  MyMatrixClass operator | (MyMatrixClass& B); //A|B = A^(-1)*B Matrix Solve

  //Overload operators to make matrix / scalar operations easier
  MyMatrixClass operator + (double& b); //Matrix+scalar addition
  MyMatrixClass operator - (double& b); //Matrix-scalar subtraction
  MyMatrixClass operator / (double& b); //Matrix/scalar division

  //Overload operators to make accessing elements of a matrix easier
  double& operator () (int n, int m);

  //Create a function to print the matrix
  void print();

private:
  //Matrix attributes
  vector<vector<double>> myMatrix; //The matrix data
  int nRows; //Number of rows in the matrix
  int nCols; //Number of columns in the matrix
};
```

MyMatrixClass.cpp

```
#include <vector> //vector
#include "MyMatrixClass.h" //Where MyMatrixClass is declared
#include "MyMatrixMath.h" //MatrixAdd, MatrixMultiply, LeftInverseSolve
// MatrixScalarMult, printMatrix

using namespace std; //To use vector, cout, and endl without std::

//Constructor for the MyMatrixClass object that initializes every element
// of the matrix to init
MyMatrixClass::MyMatrixClass(int rows, int cols, double init)
{
  //Set the number of rows and columns in the matrix
  this->nRows = rows;
  this->nCols = cols;

  //Initialize all values of the matrix to init
  this->myMatrix.resize(rows); //Give the matrix the right # of rows
  for (int ii = 0; ii < rows; ii++) {
    this->myMatrix[ii].resize(cols); //Give the matrix the right # of columns
    for (int jj = 0; jj < cols; jj++) {
      this->myMatrix[ii][jj] = init;
    }
  }
}
```

```
//Alternate constructor for the MyMatrixClass that allows it to
// be initialized to a specific matrix A
MyMatrixClass::MyMatrixClass(vector<vector<double>> A)
{
  //Set the number of rows and columns in the matrix
  this->nRows = A.size();
  this->nCols = A[0].size();

  //Initialize all values of the matrix to init
  this->myMatrix.resize(this->nRows); //Give the matrix the right # of rows
  for (int ii = 0; ii < this->nRows; ii++) {
    this->myMatrix[ii].resize(this->nRows); //Give the matrix the right # of columns
    for (int jj = 0; jj < this->nCols; jj++) {
      //Copy A to myMatrix
      this->myMatrix[ii][jj] = A[ii][jj];
    }
  }
}

//Destructor...Deletes the MyMatrixClass object when the program terminates
MyMatrixClass::~MyMatrixClass()
{
}

//Matrix addition operator overload method
MyMatrixClass MyMatrixClass::operator+(MyMatrixClass& B)
{
  //Create the output matrix C=A+B and initialize it to zero
  MyMatrixClass C(nRows, nCols, 0.0);

  //Perform the matrix addition
  C.myMatrix = MatrixAdd(this->myMatrix, B.myMatrix);

  //Return the output
  return C;
}

//Matrix subtraction operator overload method
MyMatrixClass MyMatrixClass::operator-(MyMatrixClass& B)
{
  //Create the output matrix C=A-B and initialize it to zero
  MyMatrixClass C(nRows, nCols, 0.0);

  //Perform the matrix subtraction
  for (int ii = 0; ii < nRows; ii++)
    for (int jj = 0; jj < nCols; jj++)
      C.myMatrix[ii][jj] = this->myMatrix[ii][jj] - B.myMatrix[ii][jj];

  //Return the result C=A-B
  return C;
}

//Matrix multiplication operator overload method
```

```
MyMatrixClass MyMatrixClass::operator*(MyMatrixClass& B)
{
  //Create the output matrix C=A*B and initialize it to zero
  MyMatrixClass C(this->nRows, B.nCols, 0.0);

  //Perform the matrix multiplication
  C.myMatrix = MatrixMultiply(this->myMatrix, B.myMatrix);

  //Return the result C=A*B
  return C;
}

//Matrix inversion solver operator overload method
MyMatrixClass MyMatrixClass::operator|(MyMatrixClass& B)
{
  //Create the output matrix C=A\B (C=A^-1*B) and initialize it to zero
  MyMatrixClass C(this->nRows, B.nCols, 0.0);

  //Solve the matrix equation to get C = A^(-1)*B
  C.myMatrix = LeftInverseSolve(this->myMatrix, B.myMatrix);

  //Return the result C=A\B
  return C;
}

//Matrix+scalar addition operator overload method
MyMatrixClass MyMatrixClass::operator+(double& b)
{
  //Create the output matrix C and initialize it to zero
  MyMatrixClass C(nRows, nCols, 0.0);

  //Perform the matrix+scalar addition
  for (int ii = 0; ii < nRows; ii++)
    for (int jj = 0; jj < nCols; jj++)
      C.myMatrix[ii][jj] = this->myMatrix[ii][jj] + b;

  //Return the result C = A+b
  return C;
}

//Matrix-scalar subtraction operator overload method
MyMatrixClass MyMatrixClass::operator-(double& b)
{
  //Create the output matrix C and initialize it to zero
  MyMatrixClass C(nRows, nCols, 0.0);

  //Perform the matrix-scalar subtraction
  for (int ii = 0; ii < nRows; ii++)
    for (int jj = 0; jj < nCols; jj++)
      C.myMatrix[ii][jj] = this->myMatrix[ii][jj] - b;

  //Return the result C = A-b
  return C;
```

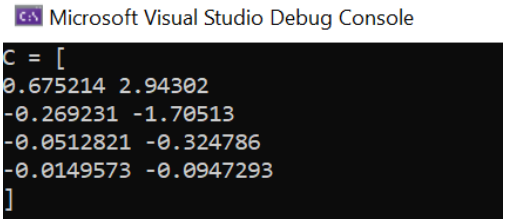
```
}

//Matrix/scalar division operator overload method
MyMatrixClass MyMatrixClass::operator/(double& b)
{
  //Create the output matrix C and initialize it to zero
  MyMatrixClass C(nRows, nCols, 0.0);

  //Perform the matrix/scalar division
  for (int ii = 0; ii < nRows; ii++)
    for (int jj = 0; jj < nCols; jj++)
      C.myMatrix[ii][jj] = this->myMatrix[ii][jj] / b;

  //Return the result C = A/b
  return C;
}

//Operator overload method that allows access to matrix elements A(n,m)
double& MyMatrixClass::operator()(int n, int m)
{
  //Return the value in row n column m of the matrix
  return this->myMatrix[n][m];
}

//Class member function to print the matrix to the console
void MyMatrixClass::print()
{
  //Use the printMatrix function to print the matrix to the console
  printMatrix(this->myMatrix);
}
```

CppCrashCoursePart3.cpp

```
#include <iostream> //cout, endl
#include "MyMatrixClass.h" //MyMatrixClass
using namespace std; //To use cout and endl without std::

int main()
{
  //Declare and define the A and B matrices
  MyMatrixClass A({
    {1.0,-2.0,3.0,4.0},
    {5.0,6.0,-7.0,8.0},
    {9.0,10.0,11.0,-12.0},
    {13.0,14.0,15.0,16.0} });
  MyMatrixClass B({
    {1.0,5.0},
    {2.0,6.0},
```

```
    {3.0,7.0},
    {4.0,8.0} });

  //Solve C = A^-1 * B
  MyMatrixClass C = A | B;

  //Print the output matrix C
  cout << "C = [" << endl;
  C.print();
  cout << "]" << endl;
}
```

Compile and run the code. The console window should appear as follows:



If not, fix any errors and make sure your Solution Explorer includes the files discussed above.

How it Works

We will discuss certain parts of the files listed above. The MyMatrixClass.h file declares the MyMatrixClass class. The class has public and private attributes and methods. Private attributes and methods can only be accessed by MyMatrixClass. For example, they can be accessed within the file MyMatrixClass.cpp, but they cannot be accessed from CppCrashCourse3.cpp or MyMatrixMath.cpp. Public attributes and methods can be accessed within other files that have #include "MyMatrixClass.h".

The functions MyMatrixClass::MyMatrixClass(int rows, int cols, double init) and MyMatrixClass:: MyMatrixClass(vector<vector<double» A) are called constructors. A class can have multiple constructors. The constructor builds the object using the arguments passed to it. For example, the constructor builds a matrix with rows rows and cols columns and initializes all values of the matrix to the same value: init. On the other hand, the constructor MyMatrixClass::MyMatrixClass(vector<vector<double» A) builds a matrix equal to the matrix A passed to it.

The function MyMatrixClass:: MyMatrixClass() is a called a destructor. It deletes the MyMatrixClass objects when the program terminates.

Because MyMatrixClass defines a new data type, if we want to use operators, we must define what they do. Defining what an operator does is called operator overloading. For example, matrix multiplication is different than scalar multiplication. Row elements of one matrix are multiplied by column elements of the other. The * operator is overloaded in the MyMatrixClass method MyMatrixClass MyMatrixClass::operator* (MyMatrixClass& B).

You may have noticed that operators can be overloaded multiple times, causing different results. For example, the + operator is overloaded twice: once in MyMatrixClass MyMatrixClass::operator+(MyMatrixClass& B) and again in MyMatrixClass MyMatrixClass::operator+(double& b). In the first method

MyMatrixClass MyMatrixClass::operator+(MyMatrixClass& B), it is overloaded for matrix + matrix addition. In the second method MyMatrixClass MyMatrixClass::operator+(double& b), it is overloaded for matrix + scalar addition. If, in CppCrashCoursePart3.cpp, a matrix is added to another matrix the compiler will automatically use the method MyMatrixClass MyMatrixClass::operator+(MyMatrixClass& B). If a matrix is added to a scalar, the compiler will automatically use the method MyMatrixClass MyMatrixClass::operator+(double& b) instead.

You may have noticed the keyword this->. It just means that the attribute belongs to MyMatrixClass. For example, this->myMatrix means that myMatrix belongs to MyMatrixClass. Using this->myMatrix is not strictly necessary within MyMatrixClass.cpp. We could simply write myMatrix instead.

Study the code carefully to understand how it works. Try out the different constructors and operators by calling them from CppCrashCoursePart3.cpp. See if you can understand how each works. If you need additional help, there are many good tutorials on C++ classes and object-oriented-programming (see for example https://www.w3schools.com/cpp/cpp_oop.asp, accessed Dec. 2022).

Do It Yourself

Modify the MyMatrixClass class in the following two ways:

1. Add an operator overload method to perform matrix-scalar multiplication of the matrix A and scalar b simply by using the * operator, i.e., C = A*b. The method must call the MatrixScalarMult function defined in MyMatrixMath.cpp. Use the matrix MyMatrixClass A(1.0,-2.0,3.0,4.0); and the scalar double b = 0.5;. Print the result to the console.

2. Create a new class member function named transpose(). It should cause a matrix A to be transposed by the code: A.transpose(). Use the matrix MyMatrixClass A(1.0,-2.0,3.0,4.0);. Print A.transpose() to the console.

13.0.3 Getting and Setting Private Members

It can sometimes be useful to grant access to the private members of a class. As a reminder, the private members of the MyMatrixClass class were nRows, nCols, and myMatrix. The following code shows changes to MyMatrixClass.h and MyMatrixClass.cpp to grant other files and functions access to either get or set the private members of the MyMatrixClass objects. Specifically, the class member function getRows() grants read-only (but not writing) privileges to the private member nRows. The function getCols() grants read-only access to nCols, and getMatrix() grants read-only access to myMatrix. The class member function setMatrix() grants write-only access to the myMatrix private member.

Two functions were added to the MyMatrixClass.cpp file to show how these get and set functions are used. The function matExp(A,B,Δt) calculates the state-space solution matrix exponential F_d:

$$F_d = \text{expm}\left(\begin{bmatrix} A\Delta t & B\Delta t \\ \mathbf{0} & \mathbf{0} \end{bmatrix}\right) = \begin{bmatrix} A_d & B_d \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$$

where **I** is the identity matrix, A_d is the discrete state-transition matrix, and B_d is the discrete input-transition matrix.

The function stateSpace(& A_d, & B_d, A, B, Δt) calls the matExp(A,B,Δt) function and returns the discrete state-transition matrices A_d and B_d.

The updated code for MyMatrixClass.h and MyMatrixClass.cpp is provided below:

MyMatrixClass.h

```
#pragma once //Tell the compiler to only compile this file once
#include <vector> //vector
#include "MyMatrixMath.h"
using namespace std; //To use vector, cout, and endl without std::
//Declare the MyMatrixClass class
class MyMatrixClass
{
public:
        //Constructor for the MyMatrixClass object that initializes every element
        // of the matrix to init
        MyMatrixClass(int rows, int cols, double init);
        //Alternate constructor for the MyMatrixClass that allows it to
        // be initialized to a specific matrix A
        MyMatrixClass(vector<vector<double>> A);
        //MyMatrixClass destructor to delete the class when the program ends
        ~MyMatrixClass();

        //Overload operators to perform matrix operations more easily
        MyMatrixClass operator + (MyMatrixClass& B); //Matrix+Matrix addition
        MyMatrixClass operator - (MyMatrixClass& B); //Matrix-Matrix subtraction
        MyMatrixClass operator * (MyMatrixClass& B); //Matrix*Matrix multiplication
        MyMatrixClass operator | (MyMatrixClass& B); //A|B = A^(-1)*B Matrix Solve

        //Overload operators to make matrix / scalar operations easier
        MyMatrixClass operator + (double& b); //Matrix+scalar addition
        MyMatrixClass operator - (double& b); //Matrix-scalar subtraction
        MyMatrixClass operator * (double& b); //Matrix-scalar multiplication
        MyMatrixClass operator / (double& b); //Matrix/scalar division

        //Overload operators to make accessing elements of a matrix easier
        double& operator () (int n, int m);

        //Create a function to print the matrix
        void print();
        MyMatrixClass transpose();

        //Get and Set functions
        int getRows();
        int getCols();
        vector<vector<double>> getMatrix();
        void setMatrix(vector<vector<double>> A);
```

```
private:
        //Matrix attributes
        vector<vector<double>> myMatrix; //The matrix data
        int nRows; //Number of rows in the matrix
        int nCols; //Number of columns in the matrix
};

//Function to calculate the Fd Matrix: Fd = expm(A*dt,B*dt;zeros(p,n+p));
MyMatrixClass matExp(MyMatrixClass A, MyMatrixClass B, const double dt);

//Function to calculate and return the state-transition matrices Ad and Bd
void stateSpace(MyMatrixClass& Ad, MyMatrixClass& Bd, MyMatrixClass A,
        MyMatrixClass B, const double dt);
```

MyMatrixClass.cpp

```
#include <vector> //vector
#include "MyMatrixMath.h" //MatrixAdd, MatrixMultiply, LeftInverseSolve
#include "MyMatrixClass.h" //Where MyMatrixClass is declared

// MatrixScalarMult, printMatrix
using namespace std; //To use vector, cout, and endl without std::
//Constructor for the MyMatrixClass object that initializes every element
// of the matrix to init
MyMatrixClass::MyMatrixClass(int rows, int cols, double init)
{
  //Set the number of rows and columns in the matrix
  this->nRows = rows;
  this->nCols = cols;

  //Resize the matrix to the correct size
  if ((int)this->myMatrix.size() != this->nRows)
  this->myMatrix.resize(this->nRows); //Give the matrix the right # of rows
  for (int ii = 0; ii < this->nRows; ii++) {
    if ((int)this->myMatrix[ii].size() != this->nCols)
    this->myMatrix[ii].resize(this->nCols); //Give the matrix the right # of columns
  }


  //Initialize all values of the matrix to init
  for (int ii = 0; ii < this->nRows; ii++) {
    for (int jj = 0; jj < this->nCols; jj++) {
      this->myMatrix[ii][jj] = init;
    }
  }
}
//Alternate constructor for the MyMatrixClass that allows it to
// be initialized to a specific matrix A
```

```
MyMatrixClass::MyMatrixClass(vector<vector<double>> A)
{
  //Set the number of rows and columns in the matrix
  this->nRows = (int)A.size();
  this->nCols = (int)A[0].size();

  //Resize myMatrix to the correct size
  if (this->myMatrix.size() != A.size())
  this->myMatrix.resize(A.size()); //Give the matrix the right # of rows
  for (int ii = 0; ii < A.size(); ii++) {
    if (this->myMatrix[ii].size() != A[0].size())
    this->myMatrix[ii].resize(A[0].size()); //Give the matrix the right # of columns
  }


  //Initialize all values of the matrix to init
  for (int ii = 0; ii < this->nRows; ii++) {
    for (int jj = 0; jj < this->nCols; jj++) {
      //Copy A to myMatrix
      this->myMatrix[ii][jj] = A[ii][jj];
    }
  }
}

//Destructor...Deletes the MyMatrixClass object when the program terminates
MyMatrixClass::~MyMatrixClass()
{
}
//Matrix addition operator overload method
MyMatrixClass MyMatrixClass::operator+(MyMatrixClass& B)
{
  //Create the output matrix C=A+B and initialize it to zero
  MyMatrixClass C(nRows, nCols, 0.0);
  //Perform the matrix addition
  C.myMatrix = MatrixAdd(this->myMatrix, B.myMatrix);
  //Return the output
  return C;
}
//Matrix subtraction operator overload method
MyMatrixClass MyMatrixClass::operator-(MyMatrixClass& B)
{
  //Create the output matrix C=A-B and initialize it to zero
  MyMatrixClass C(nRows, nCols, 0.0);
  //Perform the matrix subtraction
  for (int ii = 0; ii < nRows; ii++)
    for (int jj = 0; jj < nCols; jj++)
      C.myMatrix[ii][jj] = this->myMatrix[ii][jj] - B.myMatrix[ii][jj];
  //Return the result C=A-B
  return C;
}
//Matrix multiplication operator overload method
MyMatrixClass MyMatrixClass::operator*(MyMatrixClass& B)
{
```

```
  //Create the output matrix C=A*B and initialize it to zero
  MyMatrixClass C(this->nRows, B.nCols, 0.0);
  //Perform the matrix multiplication
  C.myMatrix = MatrixMultiply(this->myMatrix, B.myMatrix);
  //Return the result C=A*B
  return C;
}
//Matrix inversion solver operator overload method
MyMatrixClass MyMatrixClass::operator|(MyMatrixClass& B)
{
  //Create the output matrix C=A\B (C=A^-1*B) and initialize it to zero
  MyMatrixClass C(this->nRows, B.nCols, 0.0);
  //Solve the matrix equation to get C = A^(-1)*B
  C.myMatrix = LeftInverseSolve(this->myMatrix, B.myMatrix);
  //Return the result C=A\B
  return C;
}
//Matrix+scalar addition operator overload method
MyMatrixClass MyMatrixClass::operator+(double& b)
{
  //Create the output matrix C and initialize it to zero
  MyMatrixClass C(nRows, nCols, 0.0);
  //Perform the matrix+scalar addition
  for (int ii = 0; ii < nRows; ii++)
    for (int jj = 0; jj < nCols; jj++)
      C.myMatrix[ii][jj] = this->myMatrix[ii][jj] + b;
  //Return the result C = A+b
  return C;
}
//Matrix-scalar subtraction operator overload method
MyMatrixClass MyMatrixClass::operator-(double& b)
{
  //Create the output matrix C and initialize it to zero
  MyMatrixClass C(nRows, nCols, 0.0);
  //Perform the matrix-scalar subtraction
  for (int ii = 0; ii < nRows; ii++)
    for (int jj = 0; jj < nCols; jj++)
      C.myMatrix[ii][jj] = this->myMatrix[ii][jj] - b;
  //Return the result C = A-b
  return C;
}
MyMatrixClass MyMatrixClass::operator*(double& b)
{
  //Create the output matrix C and initialize it to zero
  MyMatrixClass C(this->nRows, this->nCols, 0.0);

  //Perform the matrix/scalar multiplication
  C.myMatrix = MatrixScalarMult(this->myMatrix, b);

  return C;
}
//Matrix/scalar division operator overload method
MyMatrixClass MyMatrixClass::operator/(double& b)
```

```
{
  //Create the output matrix C and initialize it to zero
  MyMatrixClass C(nRows, nCols, 0.0);
  //Perform the matrix/scalar division
  for (int ii = 0; ii < nRows; ii++)
    for (int jj = 0; jj < nCols; jj++)
      C.myMatrix[ii][jj] = this->myMatrix[ii][jj] / b;
  //Return the result C = A/b
  return C;
}
//Operator overload method that allows access to matrix elements A(n,m)
double& MyMatrixClass::operator()(int n, int m)
{
  //Return the value in row n column m of the matrix
  return this->myMatrix[n][m];
}
//Class member function to print the matrix to the console
void MyMatrixClass::print()
{
  //Use the printMatrix function to print the matrix to the console
  printMatrix(this->myMatrix);
}

MyMatrixClass MyMatrixClass::transpose()
{
  //Create the output matrix C and initialize it to zero
  MyMatrixClass C(nCols, nRows, 0.0);

  for (int ii = 0; ii < nCols; ii++)
    for (int jj = 0; jj < nRows; jj++)
      C.myMatrix[ii][jj] = this->myMatrix[jj][ii];

  return C;
}

int MyMatrixClass::getRows()
{
  return this->nRows;
}

int MyMatrixClass::getCols()
{
  return this->nCols;
}

vector<vector<double>> MyMatrixClass::getMatrix()
{
  return this->myMatrix;
}

void MyMatrixClass::setMatrix(vector<vector<double>> A)
{
  this->myMatrix = A;
```

```
}

MyMatrixClass matExp(MyMatrixClass A, MyMatrixClass B, const double dt)
{
  //ensure that the matrix sizes are compatible
  if (A.getRows() != A.getCols() || A.getRows() != B.getRows()) {
    throw 1;
  }

  MyMatrixClass Fd(A.getRows() + B.getCols(), A.getRows() + B.getCols(), 0.0);
  for (int ii = 0; ii < Fd.getRows(); ii++) {
    for (int jj = 0; jj < Fd.getCols(); jj++) {
      if (ii < A.getRows() && jj < A.getCols()) {
        Fd(ii, jj) = A(ii, jj) * dt;
      }
      else if (ii < A.getRows()) {
        Fd(ii, jj) = B(ii, jj-A.getCols()) * dt;
      }
      else {
        //do nothing
      }
    }
  }

  MyMatrixClass expFd(Fd.getRows(), Fd.getCols(), 0.0);
  expFd.setMatrix(expm(Fd.getMatrix()));

  return expFd;
}

void stateSpace(MyMatrixClass& Ad, MyMatrixClass& Bd, MyMatrixClass A,
  MyMatrixClass B, const double dt)
{
  //ensure that Ad and Bd are the correct sizes
  if (Ad.getRows() != A.getRows()
    || Ad.getCols() != A.getCols()
    || Bd.getRows() != B.getRows()
    || Bd.getCols() != B.getCols()) {
    throw 1;
  }
  MyMatrixClass Fd = matExp(A, B, dt);

  for (int ii = 0; ii < Ad.getRows(); ii++) {
    for (int jj = 0; jj < Fd.getCols(); jj++) {
      if (jj < Ad.getCols()) {
        Ad(ii, jj) = Fd(ii, jj);
      }
      else {
        Bd(ii, jj - Ad.getCols()) = Fd(ii, jj);
      }
    }
  }
}
```

13.0.4 Solving $\dot{x} = Ax + Bu$

The following CppCrashCoursePart3.cpp code uses the updated MyMatrixClass.h and MyMatrixClass.cpp libraries to solve the following state-space system:

$$\dot{x} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -6 & -7 & -8 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u \tag{13.1}$$

$$y = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} x + 0.8u \tag{13.2}$$

with the initial condition

$$x(0) = \begin{bmatrix} -2 \\ -1 \\ 3 \end{bmatrix} \tag{13.3}$$

the time-step

$$\Delta t = 0.1 \tag{13.4}$$

and the input sequence

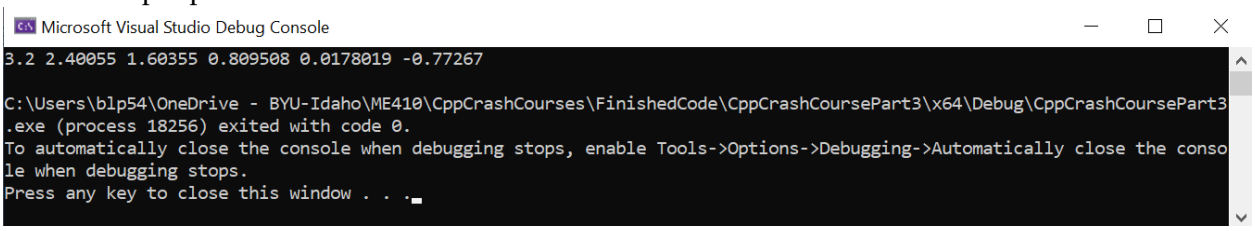
$$u = \begin{bmatrix} 4, 3, 2, 1, 0, -1 \end{bmatrix} \tag{13.5}$$

CppCrashCoursePart3.cpp

```
#include <iostream> //cout, endl
#include "MyMatrixClass.h" //MyMatrixClass
#include <vector>
using namespace std; //To use cout and endl without std::
int main()
{
  //Set the A and B matrices
  MyMatrixClass A({ {0.0,1.0,0.0},{0.0,0.0,1.0},{-6.0,-7.0,-8.0} });
  MyMatrixClass B(3, 1, 0.0);
  B(2, 0) = 1.0;

  //Set the initial condition for x
  MyMatrixClass x(3, 1, 0.0);
  x(0, 0) = -2.0;
  x(1, 0) = -1.0;
  x(2, 0) = 3.0;

  //Set the time-step
  double dt = 0.1;

  //Set the input sequence
  vector<double> u = { 4.0,3.0,2.0,1.0,0.0,-1.0 };

  //Get the distrete state-transition matrices Ad and Bd
  MyMatrixClass Ad(A.getRows(), A.getCols(), 0.0);
```

```
    MyMatrixClass Bd(B.getRows(), B.getCols(), 0.0);
    stateSpace(Ad, Bd, A, B, dt);

    //Allocate memory to store the output sequence
    int N = (int)u.size();
    MyMatrixClass y(1, N, 10.0);

    //run the FOR loop to simulate the state-space system
    for (int ii = 0; ii < N; ii++) {
      //Store the output
      y(0, ii) = x(0, 0) + 0.8 * u[ii];

      //Solve x = Ad*x+Bd*u
      MyMatrixClass Bdu = Bd * u[ii];
      x = Ad * x;
      x = x + Bdu;
    }

    //Print the result
    y.print();
  }
```

The output printed to the console window is shown below:



Compared to C++ Crash Course 2, did MyMatrixClass make solving this problem easier, or at least reduce the number of lines of code required?

Do It Yourself

Modify your MyMatrixClass.h and MyMatrixClass.cpp libraries as was done above. Strive to understand each line of code. Use the updated libraries to solve the following state-space system:

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ -6 & -7 \end{bmatrix} x + \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} u \tag{13.6}$$

$$y = \begin{bmatrix} 1 & 0 \\ 1 & -1 \end{bmatrix} x \tag{13.7}$$

with the initial condition

$$x(0) = \begin{bmatrix} -2 \\ -1 \end{bmatrix} \tag{13.8}$$

the time-step

$$\Delta t = 0.1 \tag{13.9}$$

and the input sequence

$$u = \begin{bmatrix} 4 & 3 & 2 & 1 & 0 & -1 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \tag{13.10}$$

Check your work by solving the same state-space system in MATLAB, python, or another programming language. This is the end of C++ Crash Course 3.

Chapter 14

Using C++ With MATLAB

Contents

After installing Visual C++ (see Section 11.0.1), you are ready to configure MATLAB to compile C++ code. Open MATLAB, and in the Command Window, type mex -setup C++. If the C++ compiler is installed correctly, the Command Window should say something like MEX configured to use 'Microsoft Visual C++ 2022' for C++ language compilation.

MATLAB can compile other coding languages into MATLAB Executable (mex) files. This can be a great benefit. For example, MATLAB can compile C++ code into mex functions. The compiled C++ code usually runs faster than MATLAB code, but C++ code does not have readily accessible and easy-to-use graphing, matrix math, symbolic math, etc. capabilities like MATLAB does. Also, code that is implemented in microcontrollers is often written in languages other than MATLAB. MATLAB can compile the code for these microcontrollers so it can be tested in the MATLAB and Simulink modeling environments before being flashed onto the microcontrollers.

14.1 Multiplying Matrices with mex Functions

This section will create MATLAB and C++ files. It is important that all the files for a mex program are saved in the same directory and that MATLAB's working directory is set to that file location. This section will use the folder location named "MultiplyMatricesWithMexCPP".

This section will create a mex function that uses C++ with MATLAB to multiply two matrices. Open Visual Studio, and select the option to Open a local folder. Navigate to the folder "MultiplyMatricesWith-MexCPP" and select it.

In the Solution Explorer in Visual Studio, right click the folder MultiplyMatricesWithMexCPP and click Add New Item... Click Visual C++, click C++ File (.cpp), and then click Open. Change the name of the file that was created to mexMatrixMultiply.cpp. Add the following C++ code to the file mexMatrixMultiply.cpp. Instead of copying and pasting the code, it is recommended that you type it. Typing it helps you learn better, and may prevent compiler errors that are caused by incompatible font

formats between this file and Visual Studio. Read the comments in the code and try to understand how it works.

mexMatrixMultiply.cpp

```
//Include the Matlab Executable (mex) header files to use the following:
//mex, MexFunction, matlab::mex::Function, matlab::data,
//matlab::mex::ArgumentList, getNumberOfElements(), MATLABEngine(),
//TypedArray, ArrayFactory
#include "mex.hpp"
#include "mexAdapter.hpp"

#include <vector> //vector

using namespace matlab::data; //To use TypedArray, ArrayFactory, ArrayType, Array
                              //ArrayDimensions,
                              //without matlab::data::
using matlab::mex::ArgumentList; //To use ArgumentList without matlab::mex::
using namespace std; //To use vector without std::

//Fill the matrix M with the array data in the vector M_array
void MatlabInputArrayToMatrix(vector<vector<double>>& M, size_t nRows,
    TypedArray<double> M_array)
{
  int ii = 0; //Row index
  int jj = 0; //Column index
  for (auto& elem : M_array) {
    M[ii][jj] = elem; //Populate the A matrix
    ii++;
    if (ii > nRows - 1) {
      //Go to the next column
      ii = 0;
      jj++;
    }
  }
}

//Convert the matrix OutMat into a Matlab output array Out_Array
matlab::data::TypedArray<double> MatrixToOutputMatlabArray(vector<vector<double>> OutMat)
{
  //Get the output matrix sizes for the Matlab output array
  size_t nRows = OutMat.size();
  size_t nCols = OutMat[0].size();
  vector<double> out(nRows * nCols);

  //Create the output Matlab array
  matlab::data::ArrayFactory factory;
  //Convert to a Matlab Array
  matlab::data::TypedArray<double> Out_Array =
    factory.createArray<double>({ nRows,nCols }, { 0.0 });
```

```
  //Put the elements from the OutMat matrix into the Out_Array
  int ii = 0;
  int jj = 0;
  for (auto& elem : Out_Array) {
    //Set the Out_Array elements to the OutMat elements
    elem = OutMat[ii][jj];
    ii++;
    if (ii > nRows - 1) {
      //Go to the next column
      ii = 0;
      jj++;
    }
  }

  //Return the Matlab output array
  return Out_Array;
}

//Set up the matrix M to have nRows and nCols
void setMatrixSize(vector<vector<double>>& M, size_t nRows,
  size_t nCols)
{
  //Resize the matrix M
  M.resize(nRows);
  for (int ii = 0; ii < nRows; ii++) {
    M[ii].resize(nCols);
  }
}

//Create a matrix multiplication function
vector<vector<double>> MultiplyMat(vector<vector<double>> A,
  vector<vector<double>> B)
{
  //Declare the output matrix C and give it the correct dimensions
  vector<vector<double>> C(A.size(), vector<double>(B[0].size()));

  //check that matrix inner dimensions are compatible
  if (A[0].size() != B.size()) {
    //The matrices are not compatible
    throw 1;
  }

  //Perform the matrix multiplication
  for (int ii = 0; ii < A.size(); ii++) {
    for (int jj = 0; jj < B[0].size(); jj++) {
      C[ii][jj] = 0.0;
      for (int kk = 0; kk < A[0].size(); kk++) {
        C[ii][jj] += A[ii][kk] * B[kk][jj];
      }
    }
  }
```

```
  //Return the matrix C=A*B
  return C;
}

//The class MexFunction which inherits from matlab::mex::Function
// is required for mex functions that use C++
class MexFunction : public matlab::mex::Function {
public:

  //Declare the inputs from Matlab and the Outputs from C++ as
  // ArgumentList variables
  void operator()(ArgumentList outputs, ArgumentList inputs) {
    vector<vector<double>> C; //The output matrix C = A*B

    //Get the input matrix dimensions
    ArrayDimensions sizeA = inputs[0].getDimensions(); //Dims of A
    ArrayDimensions sizeB = inputs[1].getDimensions(); //Dims of B
    //Set the row and column sizes for the A and B matrices
    const size_t rowsA = sizeA[0];
    const size_t colsA = sizeA[1];
    size_t rowsB = sizeB[0];
    size_t colsB = sizeB[1];

    vector<vector<double>> A; //Declare the input matrix A
    //Resize the matrix A
    setMatrixSize(A, rowsA, colsA);
    //Get the data for the A matrix from the input
    TypedArray<double> A_Array = std::move(inputs[0]);
    //Put the data from A_Array into the A matrix
    MatlabInputArrayToMatrix(A, rowsA, A_Array);

    vector<vector<double>> B;//Declare the input matrix B
    //Resize the matrix B
    setMatrixSize(B, rowsB, colsB);
    //Get the data for the B matrix from the input
    TypedArray<double> B_Array = std::move(inputs[1]);
    //Put the data from B_Array into the B matrix
    MatlabInputArrayToMatrix(B, rowsB, B_Array);

    //Do the matrix multiplication
    C = MultiplyMat(A, B);

    //Convert the output matrix C into a Matlab output array C_Array
    matlab::data::TypedArray<double> C_Array = MatrixToOutputMatlabArray(C);

    //Return the output array C_Array to Matlab
    outputs[0] = C_Array;
  }
};
```

Open a new tab in a Matlab editor, and save the file as runmeMultiplyMatricesWithMexCPP.m. Make sure that it is saved in the same folder location as mexMatrixMultiply.cpp. Type the following code:

runmeMultiplyMatricesWithMexCPP.m

```
close all
clear all
clc

%% compile the C++ mex function
mex -g mexMatrixMultiply.cpp

%% Run the function

A = randi(9,[2,4]);
B = randi(9,[4,3]);
C = mexMatrixMultiply(A,B)
A*B
```

Running the code in runmeMultiplyMatricesWithMexCPP.m compiles the mexMatrixMultiply.cpp file into a Matlab mex function. This was done by the line of code: mex -g mexMatrixMultiply.cpp. The -g option compiles it with debug symbols so that a debugger can step through the code if Visual Studio is attached to Matlab. Attaching Visual Studio to Matlab and debugging is a topic that will be discussed later in this chapter. Once it is compiled, the mex function can be called directly from Matlab, as long as the Matlab path includes the directory containing the mex function. The Matlab program runs the compiled mex program by calling the function using the line C = mexMatrixMultiply(A,B). Notice that the name of the function is the same name as the .cpp file: mexMatrixMultiply. If there are multiple files, which is the topic of the next section, the name of the Matlab function will match the name of the first .cpp file after the mex command: mex -g.

We will now discuss the code in the mexMatrixMultiply.cpp file. Many of the commands, data types, namespaces, and functions are original because they are defined in MATLAB's header files "mex.hpp" and "mexAdapter.hpp". We include these MATLAB executable libraries using the commands:

```
#include "mex.hpp"
#include "mexAdapter.hpp"
```

To learn more about these header files and their contents, visit the website https://github.com/alecjacobson/matlab/tree/master/extern/include. Only MATLAB can compile code with these .hpp files unless the compiler path in Visual Studio (or a different C++ development environment) is pointed to them. The namespace matlab::data is defined in the header files, and using namespace matlab::data; allows the program to use the mex defined functions TypedArray, ArrayFactory, ArrayType, Array, ArrayDimensions without first typing matlab::data:: in front of them. The command using matlab::mex::ArgumentList; is similar in that it allows the code to use the ArgumentList data type without first writing matlab::mex:: in front of it. Writing using namespace std; allows the code to use the vector data type without first writing std::.

The next part of the code is a function that we create to fill a matrix (actually a vector of vectors of double precision numbers) named M with the data in the Matlab array M_array:

```
void MatlabInputArrayToMatrix(vector<vector<double>>& M, size_t nRows,
TypedArray<double> M_array)
{
  int ii = 0; //Row index
  int jj = 0; //Column index
  for (auto& elem : M_array) {
    M[ii][jj] = elem; //Populate the A matrix
    ii++;
    if (ii > nRows - 1) {
      //Go to the next column
      ii = 0;
      jj++;
    }
  }
}
```

The matrix M is created elsewhere and is passed by reference to the function MatlabInputArrayToMatrix using the & operator. The Matlab array M_array is a one-dimensional array that contain all the data for the columns and rows of the M matrix. The command for (auto& elem : M_array) is a for loop that iterates through each element in M_array. The command M[ii][jj] = elem; copies the elements from the M_array vector into the rows and columns of the matrix M. The auto keyword automatically detects the data type of the elements in M_array. It is useful when the data type is complicated or unknown. The i++; line causes the matrix to move to the next row in the column. When the column is full, the code in the if (ii > nRows - 1) statement iterates to the next column of the M matrix.

The next function in the code is also one that we create, but it does exactly the opposite of the MatlabInputArrayToMatrix function. It converts a matrix OutMat of type vector<vector<double» to a one-dimensional Matlab array Out_Array of type TypedArray. The data type TypedArray is defined by the mex.hpp header file.

The function setMatrixSize is one that we create. It allocated computer memory to store the data for the matrix M, which is passed to it by reference. As was done in Section 12.0.3, the function MultiplyMat uses three nested for loops to perform the matrix multiplication $C = A * B$.

The interface between Matlab and C++ is created by the MexFunction class, which inherits from the matlab::mex::Function class. The parentheses operator is overloaded in the public space of the MexFunction class by the command void operator()(ArgumentList outputs, ArgumentList inputs). Overloading this operator allows Matlab to send variables to the C++ file as inputs. It also allows the C++ code to return variables to Matlab as outputs. Both inputs and outputs are of data type ArgumentList. Notice that in the runmeMultiplyMatricesWithMex.CPP.m file, that the function C = mexMatrixMultiply(A,B) has two inputs: A and B. It has one output C. The inputs variable is a pointer to A and B. Specifically, the command TypedArray<double> A_Array = std::move(inputs[0]); creates a one-dimensional Matlab array that contains the data in the A matrix. The command TypedArray<double> B_Array = std::move(inputs[1]); does the same for the B matrix. The C++ command MatlabInputArrayToMatrix(A, rowsA, A_Array); copies the data from A_Array into the rows and columns of vector<vector<double» A;, and the command MatlabInputArrayToMatrix(B, rowsB, B_Array); does the same for the B matrix. If there was a third input, it could be accessed by referencing inputs[2]. The command C = MultiplyMat(A, B); multiplies the

A and B matrices. The resulting matrix is C. The command matlab::data::TypedArray<double> C_Array = MatrixToOutputMatlabArray(C); copies the data from the C matrix into a Matlab array C_Array, and the code outputs[0] = C_Array; returns C_Array to Matlab. Because Out_Array was created to have nRows and nCols, C_Array is returned to Matlab having the correct dimensions. If there was a second output, it would be returned to Matlab using the code outputs[1].

14.2 Mex Functions With Multiple Files

This section accomplishes the same thing as the previous section. The difference is that it uses multiple C++ files instead of only one. The purpose is to demonstrate how to create and compile mex (Matlab Executable) functions from multiple files.

In this example, the code from the last section's mexMatrixMultiply.cpp file is divided into the following seven files:

1. MultiplyMat.h
2. MultiplyMat.cpp
3. MatlabInputArrayToMatrix.h
4. MatlabInputArrayToMatrix.cpp
5. MatrixToOutputMatlabArray.h
6. MatrixToOutputMatlabArray.cpp
7. mexMatlabInterface.cpp

There is also one Matlab file: runmeMexMultipleFiles.m. All of the files are contained in the same directory. The Matlab file runmeMexMultipleFiles.m is provided first:

```
%% compile the C++ mex function
mex -g MultiplyMat.cpp ...
mexMatlabInterface.cpp ...
MatlabInputArrayToMatrix.cpp ...
MatrixToOutputMatlabArray.cpp

%% Run the function
A = randi(9,[2,4]);
B = randi(9,[4,3]);
C = MultiplyMat(A,B)
A*B
```

The command to compile all of the C++ files is mex -g MultiplyMat.cpp mexMatlabInterface.cpp MatlabInputArrayToMatrix.cpp MatrixToOutputMatlabArray.cpp. Notice that when Matlab calls the compiled C++ function, the name of the function is MultiplyMat. That is because the first file after mex -g was MultiplyMat.cpp. Here are the seven C++ files:

MultiplyMat.h

```
#pragma once

#include <vector> //vector
using namespace std; //To use vector without std::

//Declare the MultiplyMat function
extern vector<vector<double>> MultiplyMat(vector<vector<double>> A,
  vector<vector<double>> B);
```

MultiplyMat.cpp

```
#include "MultiplyMat.h" //Include the MultiplyMat header file

//Create a matrix multiplication function
extern vector<vector<double>> MultiplyMat(vector<vector<double>> A,
  vector<vector<double>> B)
{
  //Declare the output matrix C and give it the correct dimensions
  vector<vector<double>> C(A.size(), vector<double>(B[0].size()));
  //check that matrix inner dimensions are compatible
  if (A[0].size() != B.size()) {
    //The matrices are not compatible
    throw 1;
  }
  //Perform the matrix multiplication
  for (int ii = 0; ii < A.size(); ii++) {
    for (int jj = 0; jj < B[0].size(); jj++) {
      C[ii][jj] = 0.0;
      for (int kk = 0; kk < A[0].size(); kk++) {
        C[ii][jj] += A[ii][kk] * B[kk][jj];
      }
    }
  }
  //Return the matrix C=A*B
  return C;
}
```

MatlabInputArrayToMatrix.h

```
#pragma once

#include "mex.hpp" //To use TypedArray
#include <vector> //vector

using namespace matlab::data; //To use TypedArray without matlab::data::
using namespace std; //To use vector without std::

//Declare the function
extern void MatlabInputArrayToMatrix(vector<vector<double>>& M, size_t nRows,
  TypedArray<double> M_array);
```

MatlabInputArrayToMatrix.cpp

```
#include "MatlabInputArrayToMatrix.h"

//Fill the matrix M with the array data in the vector M_array
extern void MatlabInputArrayToMatrix(vector<vector<double>>& M, size_t nRows,
  TypedArray<double> M_array)
{
  int ii = 0; //Row index
  int jj = 0; //Column index
  for (auto& elem : M_array) {
    M[ii][jj] = elem; //Populate the A matrix
    ii++;
    if (ii > nRows - 1) {
      //Go to the next column
      ii = 0;
      jj++;
    }
  }
}
```

MatrixToOutputMatlabArray.h

```
#pragma once

#include "mex.hpp" //To use TypedArray
#include <vector> //vector
```

```
using namespace matlab::data; //To use TypedArray without matlab::data::
using namespace std; //To use vector without std::

//Declare the function
extern matlab::data::TypedArray<double> MatrixToOutputMatlabArray(
  vector<vector<double>> OutMat);
```

MatrixToOutputMatlabArray.cpp

```
#include "MatrixToOutputMatlabArray.h"

//Convert the matrix OutMat into a Matlab output array Out_Array
extern matlab::data::TypedArray<double> MatrixToOutputMatlabArray(
  vector<vector<double>> OutMat)
{
  //Get the output matrix sizes for the Matlab output array
  size_t nRows = OutMat.size();
  size_t nCols = OutMat[0].size();
  vector<double> out(nRows * nCols);
  //Create the output Matlab array
  matlab::data::ArrayFactory factory;
  //Convert to a Matlab Array
  matlab::data::TypedArray<double> Out_Array =
    factory.createArray<double>({ nRows,nCols }, { 0.0 });
  //Put the elements from the OutMat matrix into the Out_Array
  int ii = 0;
  int jj = 0;
  for (auto& elem : Out_Array) {
    //Set the Out_Array elements to the OutMat elements
    elem = OutMat[ii][jj];
    ii++;
    if (ii > nRows - 1) {
      //Go to the next column
      ii = 0;
      jj++;
    }
  }
  //Return the Matlab output array
  return Out_Array;
}
```

mexMatlabInterface.cpp

```
//Include the Matlab Executable (mex) header files to use the following:
//mex, MexFunction, matlab::mex::Function, matlab::data,
//matlab::mex::ArgumentList, getNumberOfElements(), MATLABEngine(),
//TypedArray, ArrayFactory
#include "mex.hpp"
#include "mexAdapter.hpp"
#include "MatrixToOutputMatlabArray.h" //To use MatrixToOutputMatlabArray()
#include "MatlabInputArrayToMatrix.h" //To use MatlabInputArrayToMatrix()
#include "MultiplyMat.h" //Include the MultiplyMat header file
#include <vector> //vector
using namespace matlab::data; //To use TypedArray, ArrayFactory, ArrayType, Array
//ArrayDimensions,
//without matlab::data::
using matlab::mex::ArgumentList; //To use ArgumentList without matlab::mex::
using namespace std; //To use vector without std::

//Set up the matrix M to have nRows and nCols
void setMatrixSize(vector<vector<double>>& M, size_t nRows,
  size_t nCols)
{
  //Resize the matrix M
  M.resize(nRows);
  for (int ii = 0; ii < nRows; ii++) {
    M[ii].resize(nCols);
  }
}

//The class MexFunction which inherits from matlab::mex::Function
// is required for mex functions that use C++
class MexFunction : public matlab::mex::Function {
  public:
  //Declare the inputs from Matlab and the Outputs from C++ as
  // ArgumentList variables
  void operator()(ArgumentList outputs, ArgumentList inputs) {
    vector<vector<double>> C; //The output matrix C = A*B
    //Get the input matrix dimensions
    ArrayDimensions sizeA = inputs[0].getDimensions(); //Dims of A
    ArrayDimensions sizeB = inputs[1].getDimensions(); //Dims of B
    //Set the row and column sizes for the A and B matrices
    const size_t rowsA = sizeA[0];
    const size_t colsA = sizeA[1];
    size_t rowsB = sizeB[0];
    size_t colsB = sizeB[1];
    vector<vector<double>> A; //Declare the input matrix A
    //Resize the matrix A
    setMatrixSize(A, rowsA, colsA);
    //Get the data for the A matrix from the input
    TypedArray<double> A_Array = std::move(inputs[0]);
    //Put the data from A_Array into the A matrix
    MatlabInputArrayToMatrix(A, rowsA, A_Array);
```

```
    vector<vector<double>> B;//Declare the input matrix B
    //Resize the matrix B
    setMatrixSize(B, rowsB, colsB);
    //Get the data for the B matrix from the input
    TypedArray<double> B_Array = std::move(inputs[1]);
    //Put the data from B_Array into the B matrix
    MatlabInputArrayToMatrix(B, rowsB, B_Array);
    //Do the matrix multiplication
    C = MultiplyMat(A, B);
    //Convert the output matrix C into a Matlab output array C_Array
    matlab::data::TypedArray<double> C_Array = MatrixToOutputMatlabArray(C);
    //Return the output array C_Array to Matlab
    outputs[0] = C_Array;
  }
};
```

Take some time to compare the code in this section to the previous section. Both sections use the same code to accomplish the same purpose of matrix multiplication. However, this section divided the code into multiple files whereas the previous section used only one C++ file. The code was explained in the previous section, and it is not repeated here.

14.3 Using Debugging Tools

Visual Studio can attach to MATLAB processes during runtime. This can enable you to use Visual Studio's debugging capabilities to find errors as you walk line-by-line through your compiled mex function C++ code. At each step, you can watch how variables are assigned new values. This section introduces how to use these debugging capabilities.

To use the runtime debugger, your code must compile successfully. If your code fails to compile, fix the errors that are printed to MATLAB's Command Window after running the mex -g ... command. The **-g** option enables use of the debugging tools. If the code compiles, but crashes MATLAB or Visual Studio when you try to run it, it is usually because the program is attempting to access memory that has not been allocated. You can find the source of this error by using the debugging tools. If code runs without crashing the program but produces the wrong values, the debugging tools can help fix these problems as well.

To use the debugging tools, first clear all the mex functions by running the command clear functions or clear all in MATLAB's Command Window. Then compile the code using the mex -g ... command. Open the C++ file that you want to debug in Visual Studio. Click on the debug tab in Visual Studio and select Attach to Process (see Figure 14.1).

In the box that pops up, click the "Select..." button next to the "Attach to:" box. Then set the code type to "Native", see Figure 14.2 and click "OK".

In the "Available processes" option to select the **MATLAB.exe** process and click **Attach**, see Figure 14.3.

Now Visual Studio should be attached to the MATLAB process. You can now use the debugging tools in Visual Studio. *Make sure not to run the functions **clear all** or **clear functions** in MATLAB after attaching to Visual Studio. If you do, it's not a big problem, but you will need to reattach Visual Studio to MATLAB.*

A break-point in the code can freeze the process at a specified line in the code during runtime. To

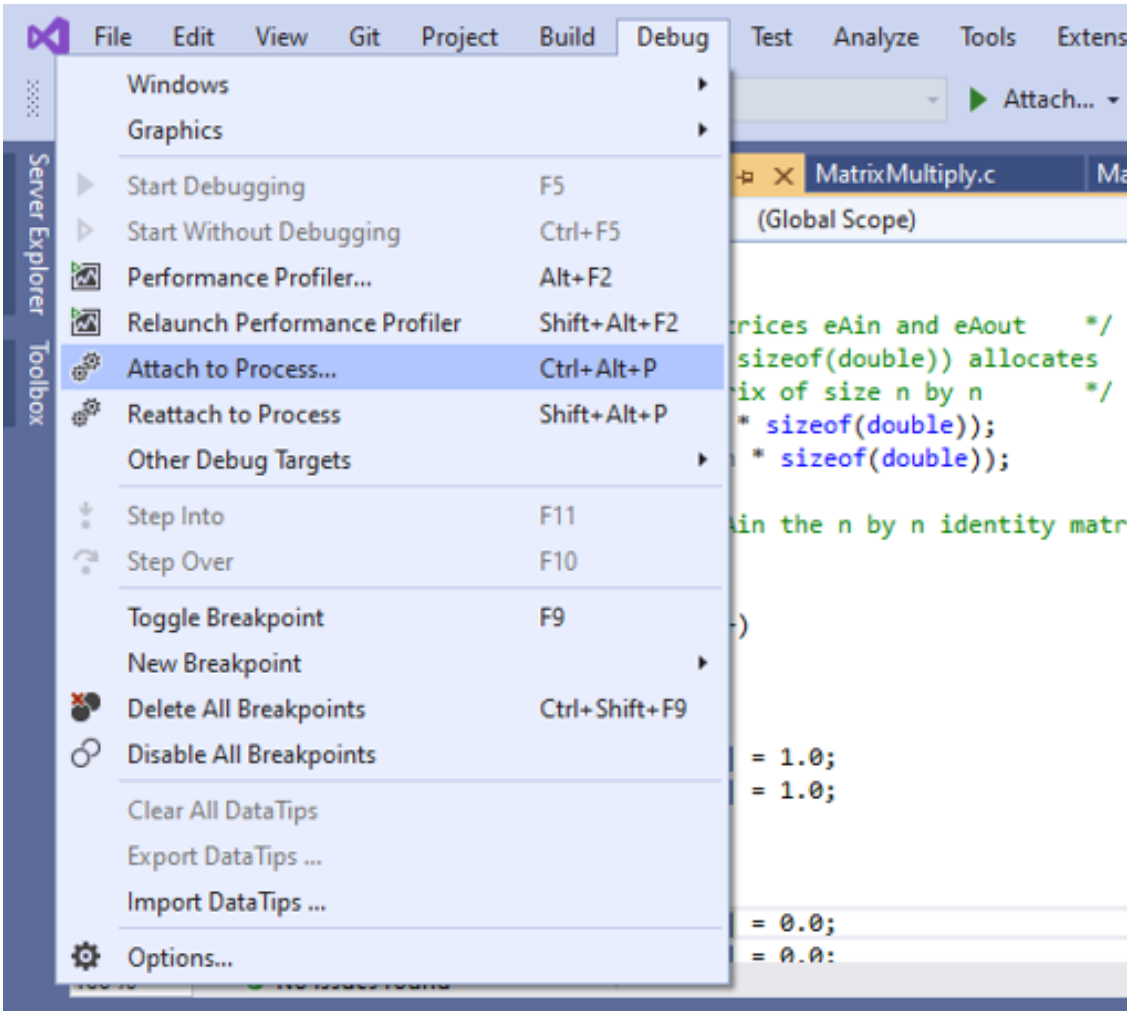


Figure 14.1 Select "Attach to Process..."

set a break-point in Visual Studio, click on the bar on the left of the line numbers (see Figure 14.4). The break-point may say that it will not be hit. That is likely because the MATLAB mex function is not currently being run.

In MATLAB, run the mex function, but be sure not to run the functions **clear all** or **clear functions**. The process should stop at the break-point that was set in Visual Studio. If the program crashes without stopping at the break-point, try setting the break-point at an earlier line in the C++ code, before any array operations are performed.

The debugging tools shown in Figure 14.5 include **Continue**, **Step Into**, **Step Over**, and **Step Out**. Use the debugging tools to step through the code. The autos box shows the values of the variables that are on the line of code that is being executed. As you step through the code, watch how these values change. Doing so can help you detect and correct mistakes.

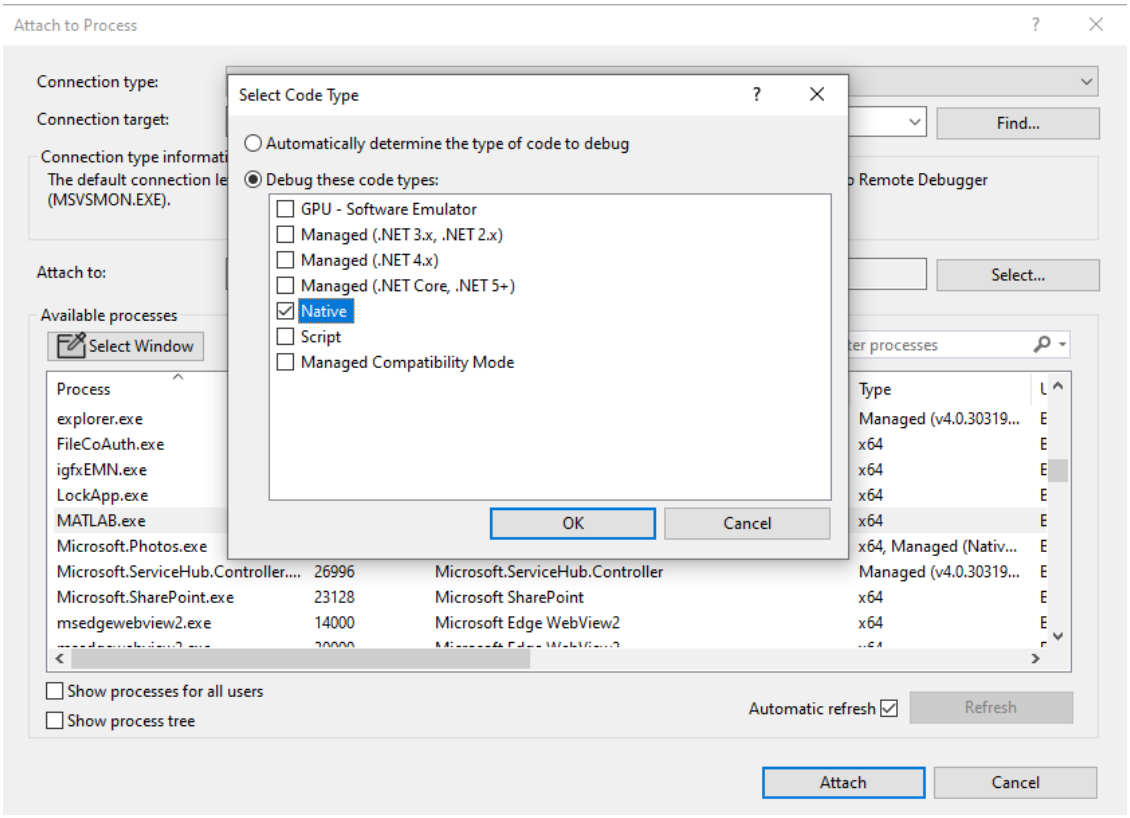


Figure 14.2 Select “Native” code type

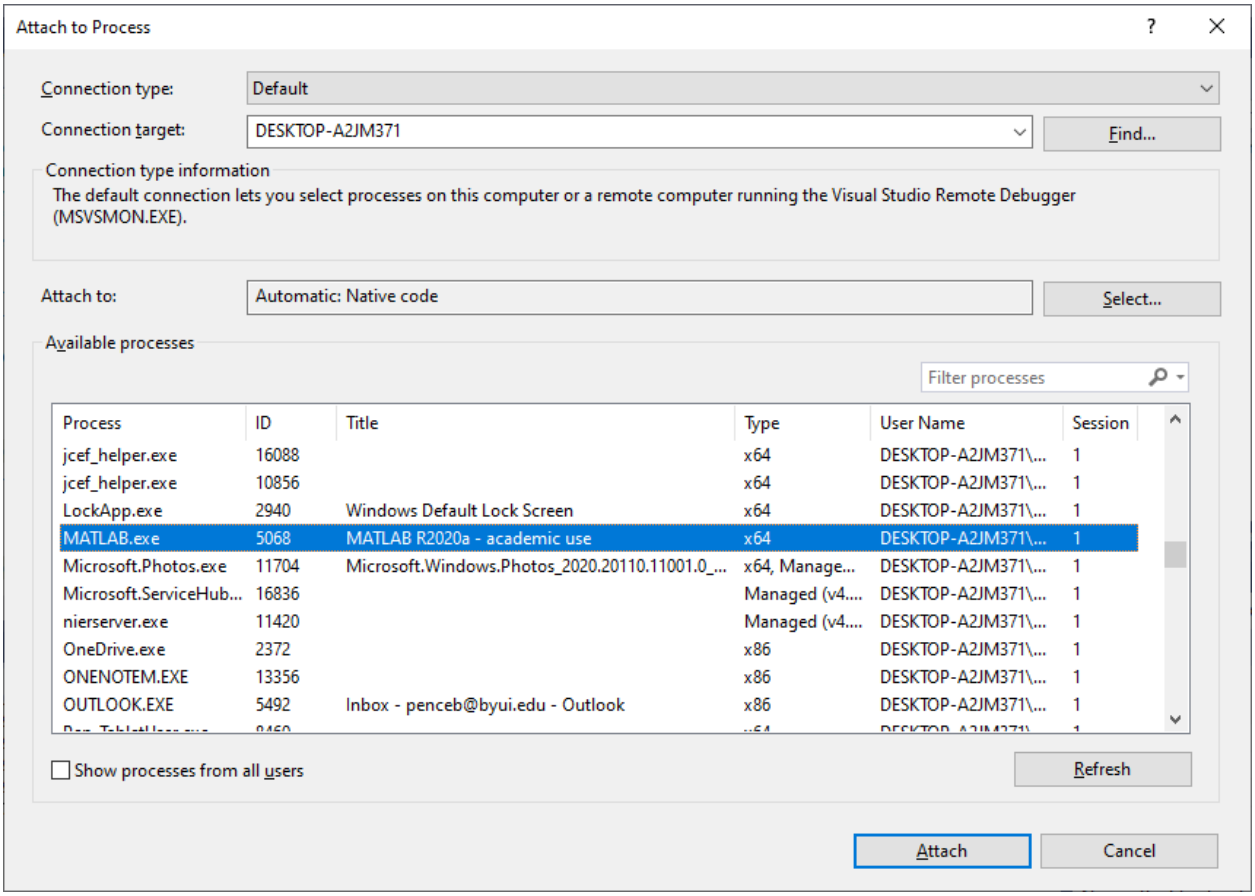


Figure 14.3 Select “MATLAB.exe” and click “Attach”

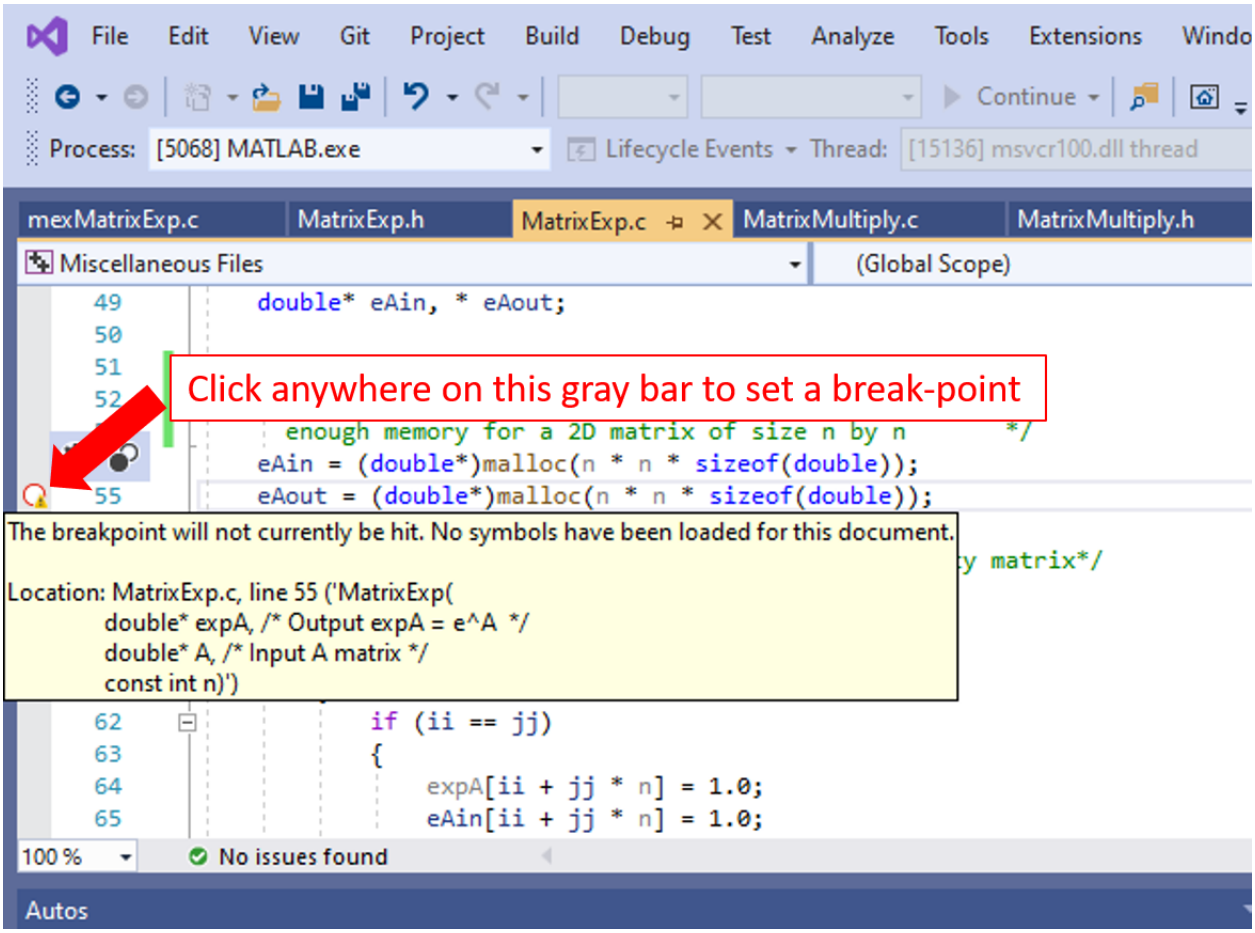


Figure 14.4 Click next to the line number to set a break-point

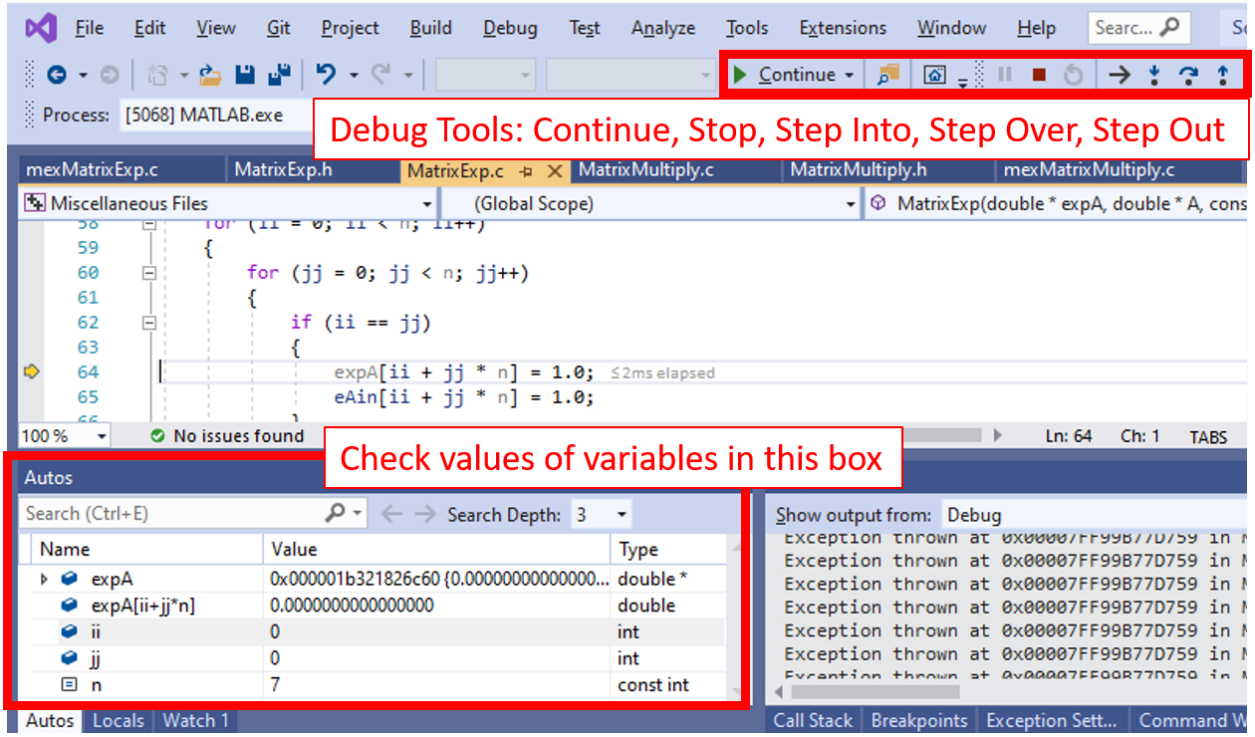


Figure 14.5 The debugging tools and autos box help detect errors

Chapter 15

Creating Custom Arduino Libraries

Contents

This chapter explains how to create Arduino libraries. The Raspberry Pi Pico microcontroller and many other microcontrollers can be programmed using the Arduino IDE (Integrated Development Environment), so learning how to write Arduino libraries can be invaluable. The Arduino website includes a guide to writing libraries which can be found here: https://docs.arduino.cc/learn/contributions/arduino-creating-library-guide (accessed April 2023). The sections below will also walk through basic steps to creating Arduino libraries.

15.1 Programming the Raspberry Pi Pico with the Arduino IDE

The Raspberry Pi Pico requires some minor setup before it can be programmed by the Arduino IDE. The steps are explained below:

- Open up the Arduino IDE and go to File->Preferences.
- In the dialog that pops up, copy and paste the following URL into the "Additional Boards Manager URLs" field: https://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json, see Figure 15.1
- Click OK to close the dialog.
- Go to Tools->Boards->Board Manager in the IDE
- Type "pico" in the search box and select "Add":

After it is done installing, to use the Pico, go to tools->Board->Raspberry Pi RP2040 Boards->Raspberry Pi Pico. Make sure to select the right port, and if necessary, press the BOOTSEL button on the Pico before and while connecting the USB cable. The raspberry Pi Pico should be ready to program with the Arduino IDE.

You cannot brick the Raspberry Pi Pico with software. If ever necessary, there is a file available from Raspberry Pi called “flash_nuke.uf2” available at https://datasheets.raspberrypi.com/soft/flash_nuke.uf2. This file can be used to reset the flash memory.

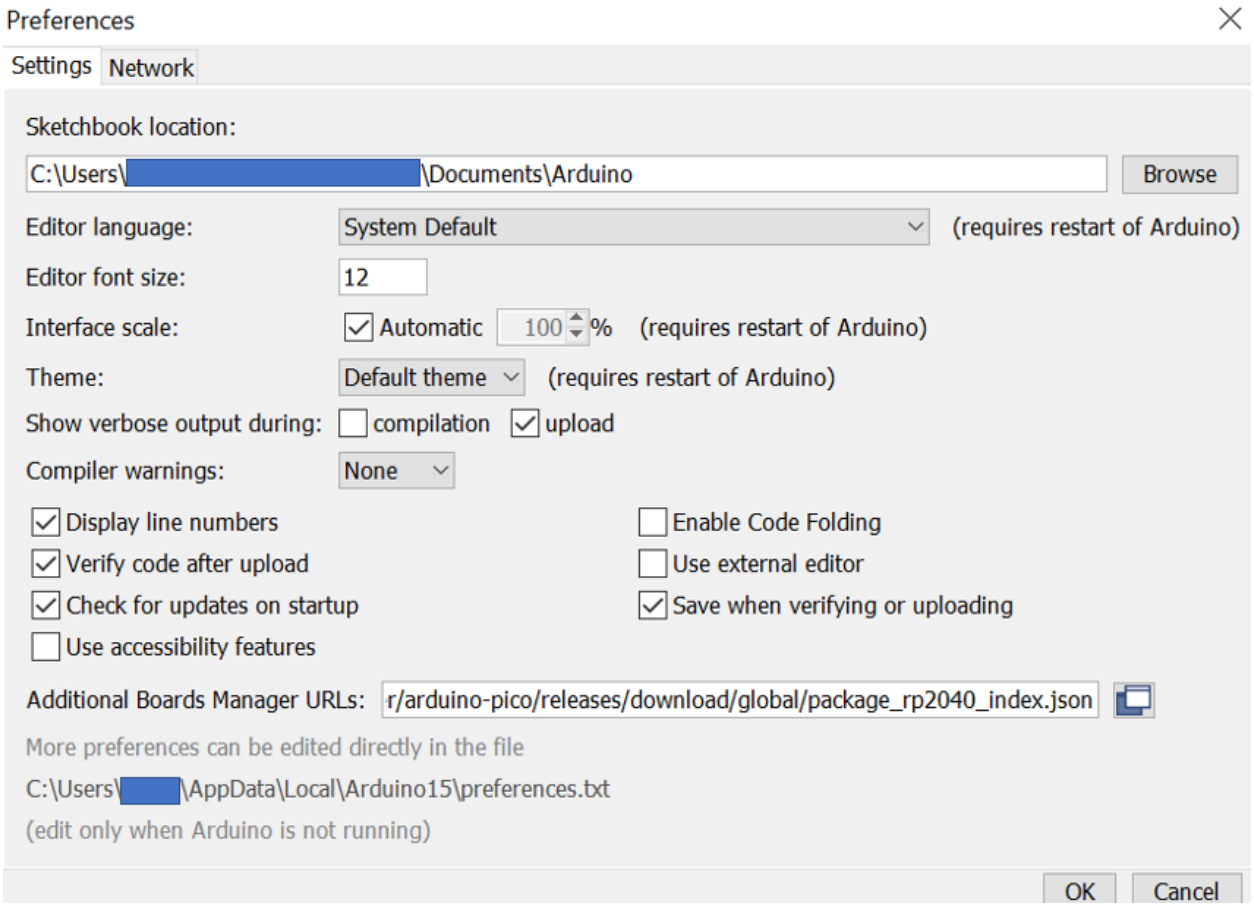


Figure 15.1 The Additional Boards Managers URLs box includes https://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json, and the Sketchbook location: shows the path to find the Arduino libraries folder.

15.2 Creating Arduino Libraries

An Arduino library requires at least two files: a C++ header (.h) file and a C++ source (.cpp) file. Both must have the same name, and they must be inside a folder with the same name. Additional source and header files can be included as well.

For example, we will create an Arduino library called Multiply3x3Matrix that will multiply two 3x3 matrices and return the resulting matrix. The library will include four files:

1. Multiply3x3Matrix.h
2. Multiply3x3Matrix.cpp

3. MatrixMultiply.h
4. MatrixMultiply.cpp

All four files will be located in the same folder named Multiply3x3Matrix. The Multiply3x3Matrix folder must be located in the location that Arduino looks for libraries. The default location is usually `C:/Users/.../Documents/Arduino/libraries`. To find the Arduino libraries location on your computer, open the Arduino IDE, click File»Preferences. Under the Settings tab, the Sketchbook location: entry box shows the path to the libraries folder, see Figure 15.1. Note that it does not include the final libraries directory.

The four Arduino library files are provided below:

Multiply3x3Matrix.h

```
#pragma once

//Multiply two 3x3 matrices: C=A*B
extern void Multiply3x3Matrix(double C[3][3],
  const double A[3][3], const double B[3][3]);
```

Multiply3x3Matrix.cpp

```
#include "Multiply3x3Matrix.h"
#include "MatrixMultiply.h"

//Multiply two 3x3 matrices: C=A*B
extern void Multiply3x3Matrix(double C[3][3],
  const double A[3][3], const double B[3][3])
{
  mat3_mult(C, A, B);
}
```

MatrixMultiply.h

```
#pragma once

//Multiply two 3x3 matrices
void mat3_mult(double C[3][3], const double A[3][3], const double B[3][3]);

//Multiply two 4x4 matrices
void mat4_mult(double C[4][4], const double A[4][4], const double B[4][4]);
```

MatrixMultiply.cpp

```
#include "MatrixMultiply.h"

//Multiply two 3x3 matrices
const int Msize = 3;
void mat3_mult(double C[3][3], const double A[3][3], const double B[3][3]) {
  for (int ii = 0; ii < Msize; ii++) {
    for (int jj = 0; jj < Msize; jj++) {
      C[ii][jj] = 0.0;
      for (int kk = 0; kk < Msize; kk++) {
        C[ii][jj] += A[ii][kk] * B[kk][jj];
      }
    }
  }
}

//Multiply two 4x4 matrices
const int MatSize = 4;
void mat4_mult(double C[4][4], const double A[4][4], const double B[4][4]) {
  for (int ii = 0; ii < MatSize; ii++) {
    for (int jj = 0; jj < MatSize; jj++) {
      C[ii][jj] = 0.0;
      for (int kk = 0; kk < MatSize; kk++) {
        C[ii][jj] += A[ii][kk] * B[kk][jj];
      }
    }
  }
}
```

Unfortunately, as of April 2023, Arduino does not support the use of the C++ Standard Template Libraries (STL). Therefore, the std::vector and std::string classes, etc. are not available for use in writing Arduino libraries.

If the Arduino IDE was open during the creation of the Arduino library files above, it will not recognize them until the Arduino IDE is closed and reopened. Once it is reopened, the new Arduino library can be found under Sketch » Include Library, see Figure 15.2.

The following Arduino code was used to test the Multiply3x3Matrix library:

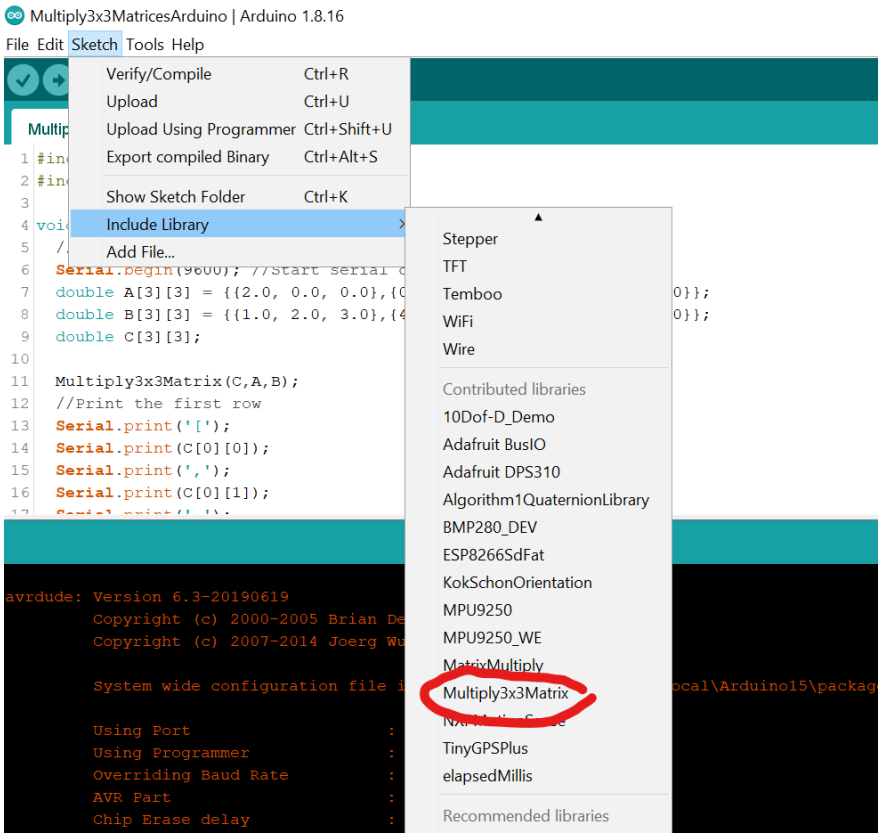


Figure 15.2 The Multiply3x3Matrix library is now in the Include Library directory.

Multiply3x3MatricesArduino.ino

```
#include <MatrixMultiply.h>
#include <Multiply3x3Matrix.h>

void setup() {
        // put your setup code here, to run once:
        Serial.begin(9600); //Start serial communication
        double A[3][3] = {{2.0, 0.0, 0.0},{0.0, 2.0, 0.0},{0.0, 0.0, 2.0}};
        double B[3][3] = {{1.0, 2.0, 3.0},{4.0, 5.0, 6.0},{7.0, 8.0, 9.0}};
        double C[3][3];

        Multiply3x3Matrix(C,A,B);
        //Print the first row
        Serial.print('[');
        Serial.print(C[0][0]);
        Serial.print(',');
        Serial.print(C[0][1]);
        Serial.print(',');
```

```
        Serial.print(C[0][2]);
        Serial.println(';');
        //Print the second row
        Serial.print(C[1][0]);
        Serial.print(',');
        Serial.print(C[1][1]);
        Serial.print(',');
        Serial.print(C[1][2]);
        Serial.println(';');
        //Print the third row
        Serial.print(C[2][0]);
        Serial.print(',');
        Serial.print(C[2][1]);
        Serial.print(',');
        Serial.print(C[2][2]);
        Serial.println(']');
}

void loop() {
        // put your main code here, to run repeatedly:
}
```

This code was uploaded to an Arduino Uno. The result was displayed on the Serial Monitor, see Figure 15.3.

Figure 15.3 Result of the Arduino code in Multiply3x3MatricesArduino.ino shown on the Arduino Serial Monitor.