```java
package edu.wpi.first.math.controller;

import edu.wpi.first.math.MathSharedStore;
import edu.wpi.first.math.MathUsageId;
import edu.wpi.first.math.MathUtil;
import edu.wpi.first.util.sendable.Sendable;
import edu.wpi.first.util.sendable.SendableBuilder;
import edu.wpi.first.util.sendable.SendableRegistry;

/** Implements a PID control loop. */
public class PIDController implements Sendable, AutoCloseable {
  private static int instances;

  // Factor for "proportional" control
  private double m_kp;

  // Factor for "integral" control
  private double m_ki;

  // Factor for "derivative" control
  private double m_kd;

  // The error range where "integral" control applies
  private double m_iZone = Double.POSITIVE_INFINITY;

  // The period (in seconds) of the loop that calls the controller
  private final double m_period;

  private double m_maximumIntegral = 1.0;

  private double m_minimumIntegral = -1.0;

  private double m_maximumInput;

  private double m_minimumInput;

  // Do the endpoints wrap around? e.g. Absolute encoder
  private boolean m_continuous;

  // The error at the time of the most recent call to calculate()
  private double m_error;
  private double m_errorDerivative;

  // The error at the time of the second-most-recent call to calculate() (used to compute velocity)
  private double m_prevError;

  // The sum of the errors for use in the integral calc
  private double m_totalError;

  // The error that is considered at setpoint.
  private double m_errorTolerance = 0.05;
  private double m_errorDerivativeTolerance = Double.POSITIVE_INFINITY;
```

```
53
54      private double m_setpoint;
55      private double m_measurement;
56
57      private boolean m_haveMeasurement;
58      private boolean m_haveSetpoint;
59
60      /**
61       * Allocates a PIDController with the given constants for kp, ki, and kd and a default period of
62       * 0.02 seconds.
63       *
64       * @param kp The proportional coefficient.
65       * @param ki The integral coefficient.
66       * @param kd The derivative coefficient.
67       * @throws IllegalArgumentException if kp &lt; 0
68       * @throws IllegalArgumentException if ki &lt; 0
69       * @throws IllegalArgumentException if kd &lt; 0
70       */
71      public PIDController(double kp, double ki, double kd) {
72        this(kp, ki, kd, 0.02);
73      }
74
75      /**
76       * Allocates a PIDController with the given constants for kp, ki, and kd.
77       *
78       * @param kp The proportional coefficient.
79       * @param ki The integral coefficient.
80       * @param kd The derivative coefficient.
81       * @param period The period between controller updates in seconds.
82       * @throws IllegalArgumentException if kp &lt; 0
83       * @throws IllegalArgumentException if ki &lt; 0
84       * @throws IllegalArgumentException if kd &lt; 0
85       * @throws IllegalArgumentException if period &lt;= 0
86       */
87      @SuppressWarnings("this-escape")
88      public PIDController(double kp, double ki, double kd, double period) {
89        m_kp = kp;
90        m_ki = ki;
91        m_kd = kd;
92
93        if (kp < 0.0) {
94          throw new IllegalArgumentException("Kp must be a non-negative number!");
95        }
96        if (ki < 0.0) {
97          throw new IllegalArgumentException("Ki must be a non-negative number!");
98        }
99        if (kd < 0.0) {
100         throw new IllegalArgumentException("Kd must be a non-negative number!");
101       }
102       if (period <= 0.0) {
103         throw new IllegalArgumentException("Controller period must be a positive number!");
104       }
```

```
105       m_period = period;
106
107       instances++;
108       SendableRegistry.addLW(this, "PIDController", instances);
109
110       MathSharedStore.reportUsage(MathUsageId.kController_PIDController2, instances);
111     }
112
113     @Override
114     public void close() {
115       SendableRegistry.remove(this);
116     }
117
118     /**
119      * Sets the PID Controller gain parameters.
120      *
121      * <p>Set the proportional, integral, and differential coefficients.
122      *
123      * @param kp The proportional coefficient.
124      * @param ki The integral coefficient.
125      * @param kd The derivative coefficient.
126      */
127     public void setPID(double kp, double ki, double kd) {
128       m_kp = kp;
129       m_ki = ki;
130       m_kd = kd;
131     }
132
133     /**
134      * Sets the Proportional coefficient of the PID controller gain.
135      *
136      * @param kp The proportional coefficient. Must be &gt;= 0.
137      */
138     public void setP(double kp) {
139       m_kp = kp;
140     }
141
142     /**
143      * Sets the Integral coefficient of the PID controller gain.
144      *
145      * @param ki The integral coefficient. Must be &gt;= 0.
146      */
147     public void setI(double ki) {
148       m_ki = ki;
149     }
150
151     /**
152      * Sets the Differential coefficient of the PID controller gain.
153      *
154      * @param kd The differential coefficient. Must be &gt;= 0.
155      */
156     public void setD(double kd) {
```

```
157       m_kd = kd;
158     }
159
160     /**
161      * Sets the IZone range. When the absolute value of the position error is greater than IZone, the
162      * total accumulated error will reset to zero, disabling integral gain until the absolute value of
163      * the position error is less than IZone. This is used to prevent integral windup. Must be
164      * non-negative. Passing a value of zero will effectively disable integral gain. Passing a value
165      * of {@link Double#POSITIVE_INFINITY} disables IZone functionality.
166      *
167      * @param iZone Maximum magnitude of error to allow integral control.
168      * @throws IllegalArgumentException if iZone &lt; 0
169      */
170     public void setIZone(double iZone) {
171       if (iZone < 0) {
172         throw new IllegalArgumentException("IZone must be a non-negative number!");
173       }
174       m_iZone = iZone;
175     }
176
177     /**
178      * Get the Proportional coefficient.
179      *
180      * @return proportional coefficient
181      */
182     public double getP() {
183       return m_kp;
184     }
185
186     /**
187      * Get the Integral coefficient.
188      *
189      * @return integral coefficient
190      */
191     public double getI() {
192       return m_ki;
193     }
194
195     /**
196      * Get the Differential coefficient.
197      *
198      * @return differential coefficient
199      */
200     public double getD() {
201       return m_kd;
202     }
203
204     /**
205      * Get the IZone range.
206      *
207      * @return Maximum magnitude of error to allow integral control.
208      */
```

```
209     public double getIZone() {
210       return m_iZone;
211     }
212
213     /**
214      * Returns the period of this controller.
215      *
216      * @return the period of the controller.
217      */
218     public double getPeriod() {
219       return m_period;
220     }
221
222     /**
223      * Returns the position tolerance of this controller.
224      *
225      * @return the position tolerance of the controller.
226      * @deprecated Use getErrorTolerance() instead.
227      */
228     @Deprecated(forRemoval = true, since = "2025")
229     public double getPositionTolerance() {
230       return m_errorTolerance;
231     }
232
233     /**
234      * Returns the velocity tolerance of this controller.
235      *
236      * @return the velocity tolerance of the controller.
237      * @deprecated Use getErrorDerivativeTolerance() instead.
238      */
239     @Deprecated(forRemoval = true, since = "2025")
240     public double getVelocityTolerance() {
241       return m_errorDerivativeTolerance;
242     }
243
244     /**
245      * Returns the error tolerance of this controller. Defaults to 0.05.
246      *
247      * @return the error tolerance of the controller.
248      */
249     public double getErrorTolerance() {
250       return m_errorTolerance;
251     }
252
253     /**
254      * Returns the error derivative tolerance of this controller. Defaults to ∞.
255      *
256      * @return the error derivative tolerance of the controller.
257      */
258     public double getErrorDerivativeTolerance() {
259       return m_errorDerivativeTolerance;
260     }
```

```
261
262     /**
263      * Returns the accumulated error used in the integral calculation of this controller.
264      *
265      * @return The accumulated error of this controller.
266      */
267     public double getAccumulatedError() {
268       return m_totalError;
269     }
270
271     /**
272      * Sets the setpoint for the PIDController.
273      *
274      * @param setpoint The desired setpoint.
275      */
276     public void setSetpoint(double setpoint) {
277       m_setpoint = setpoint;
278       m_haveSetpoint = true;
279
280       if (m_continuous) {
281         double errorBound = (m_maximumInput - m_minimumInput) / 2.0;
282         m_error = MathUtil.inputModulus(m_setpoint - m_measurement, -errorBound, errorBound);
283       } else {
284         m_error = m_setpoint - m_measurement;
285       }
286
287       m_errorDerivative = (m_error - m_prevError) / m_period;
288     }
289
290     /**
291      * Returns the current setpoint of the PIDController.
292      *
293      * @return The current setpoint.
294      */
295     public double getSetpoint() {
296       return m_setpoint;
297     }
298
299     /**
300      * Returns true if the error is within the tolerance of the setpoint. The error tolerance defaults
301      * to 0.05, and the error derivative tolerance defaults to ∞.
302      *
303      * <p>This will return false until at least one input value has been computed.
304      *
305      * @return Whether the error is within the acceptable bounds.
306      */
307     public boolean atSetpoint() {
308       return m_haveMeasurement
309           && m_haveSetpoint
310           && Math.abs(m_error) < m_errorTolerance
311           && Math.abs(m_errorDerivative) < m_errorDerivativeTolerance;
312     }
```

```
313
314     /**
315      * Enables continuous input.
316      *
317      * <p>Rather then using the max and min input range as constraints, it considers them to be the
318      * same point and automatically calculates the shortest route to the setpoint.
319      *
320      * @param minimumInput The minimum value expected from the input.
321      * @param maximumInput The maximum value expected from the input.
322      */
323     public void enableContinuousInput(double minimumInput, double maximumInput) {
324       m_continuous = true;
325       m_minimumInput = minimumInput;
326       m_maximumInput = maximumInput;
327     }
328
329     /** Disables continuous input. */
330     public void disableContinuousInput() {
331       m_continuous = false;
332     }
333
334     /**
335      * Returns true if continuous input is enabled.
336      *
337      * @return True if continuous input is enabled.
338      */
339     public boolean isContinuousInputEnabled() {
340       return m_continuous;
341     }
342
343     /**
344      * Sets the minimum and maximum contributions of the integral term.
345      *
346      * <p>The internal integrator is clamped so that the integral term's contribution to the output
347      * stays between minimumIntegral and maximumIntegral. This prevents integral windup.
348      *
349      * @param minimumIntegral The minimum contribution of the integral term.
350      * @param maximumIntegral The maximum contribution of the integral term.
351      */
352     public void setIntegratorRange(double minimumIntegral, double maximumIntegral) {
353       m_minimumIntegral = minimumIntegral;
354       m_maximumIntegral = maximumIntegral;
355     }
356
357     /**
358      * Sets the error which is considered tolerable for use with atSetpoint().
359      *
360      * @param errorTolerance Error which is tolerable.
361      */
362     public void setTolerance(double errorTolerance) {
363       setTolerance(errorTolerance, Double.POSITIVE_INFINITY);
364     }
```

```
365
366     /**
367      * Sets the error which is considered tolerable for use with atSetpoint().
368      *
369      * @param errorTolerance Error which is tolerable.
370      * @param errorDerivativeTolerance Error derivative which is tolerable.
371      */
372     public void setTolerance(double errorTolerance, double errorDerivativeTolerance) {
373       m_errorTolerance = errorTolerance;
374       m_errorDerivativeTolerance = errorDerivativeTolerance;
375     }
376
377     /**
378      * Returns the difference between the setpoint and the measurement.
379      *
380      * @return The error.
381      * @deprecated Use getError() instead.
382      */
383     @Deprecated(forRemoval = true, since = "2025")
384     public double getPositionError() {
385       return m_error;
386     }
387
388     /**
389      * Returns the velocity error.
390      *
391      * @return The velocity error.
392      * @deprecated Use getErrorDerivative() instead.
393      */
394     @Deprecated(forRemoval = true, since = "2025")
395     public double getVelocityError() {
396       return m_errorDerivative;
397     }
398
399     /**
400      * Returns the difference between the setpoint and the measurement.
401      *
402      * @return The error.
403      */
404     public double getError() {
405       return m_error;
406     }
407
408     /**
409      * Returns the error derivative.
410      *
411      * @return The error derivative.
412      */
413     public double getErrorDerivative() {
414       return m_errorDerivative;
415     }
416
```

```
417     /**
418      * Returns the next output of the PID controller.
419      *
420      * @param measurement The current measurement of the process variable.
421      * @param setpoint The new setpoint of the controller.
422      * @return The next controller output.
423      */
424     public double calculate(double measurement, double setpoint) {
425       m_setpoint = setpoint;
426       m_haveSetpoint = true;
427       return calculate(measurement);
428     }
429
430     /**
431      * Returns the next output of the PID controller.
432      *
433      * @param measurement The current measurement of the process variable.
434      * @return The next controller output.
435      */
436     public double calculate(double measurement) {
437       m_measurement = measurement;
438       m_prevError = m_error;
439       m_haveMeasurement = true;
440
441       if (m_continuous) {
442         double errorBound = (m_maximumInput - m_minimumInput) / 2.0;
443         m_error = MathUtil.inputModulus(m_setpoint - m_measurement, -errorBound, errorBound);
444       } else {
445         m_error = m_setpoint - m_measurement;
446       }
447
448       m_errorDerivative = (m_error - m_prevError) / m_period;
449
450       // If the absolute value of the position error is greater than IZone, reset the total error
451       if (Math.abs(m_error) > m_iZone) {
452         m_totalError = 0;
453       } else if (m_ki != 0) {
454         m_totalError =
455             MathUtil.clamp(
456                 m_totalError + m_error * m_period,
457                 m_minimumIntegral / m_ki,
458                 m_maximumIntegral / m_ki);
459       }
460
461       return m_kp * m_error + m_ki * m_totalError + m_kd * m_errorDerivative;
462     }
463
464     /** Resets the previous error and the integral term. */
465     public void reset() {
466       m_error = 0;
467       m_prevError = 0;
468       m_totalError = 0;
```

```
469       m_errorDerivative = 0;
470       m_haveMeasurement = false;
471     }
472
473     @Override
474     public void initSendable(SendableBuilder builder) {
475       builder.setSmartDashboardType("PIDController");
476       builder.addDoubleProperty("p", this::getP, this::setP);
477       builder.addDoubleProperty("i", this::getI, this::setI);
478       builder.addDoubleProperty("d", this::getD, this::setD);
479       builder.addDoubleProperty(
480           "izone",
481           this::getIZone,
482           (double toSet) -> {
483             try {
484               setIZone(toSet);
485             } catch (IllegalArgumentException e) {
486               MathSharedStore.reportError("IZone must be a non-negative number!", e.getStackTrace());
487             }
488           });
489       builder.addDoubleProperty("setpoint", this::getSetpoint, this::setSetpoint);
490       builder.addDoubleProperty("measurement", () -> m_measurement, null);
491       builder.addDoubleProperty("error", this::getError, null);
492       builder.addDoubleProperty("error derivative", this::getErrorDerivative, null);
493       builder.addDoubleProperty("previous error", () -> this.m_prevError, null);
494       builder.addDoubleProperty("total error", this::getAccumulatedError, null);
495     }
496   }
497
```