Mutual inductance between circular circuits

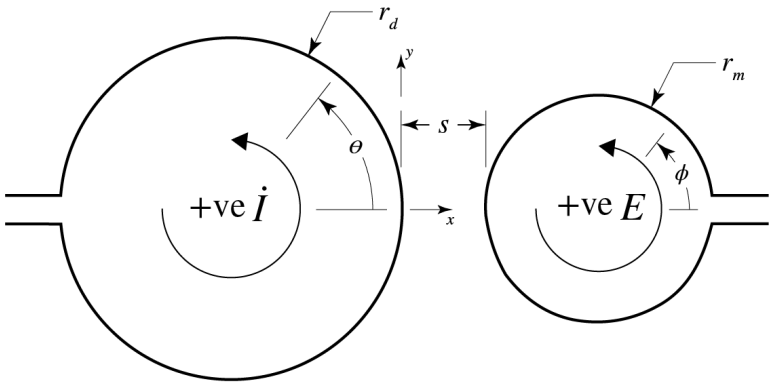
This notebook is part of the supplemental material for the paper *An inductive field component of the electric field that corresponds to Faraday's law.* The enclosed source code is used to find numerical solutions of electromagnetic force (EMF) due to mutual inductance between two circular wire circuits. This is done for the circuits in both a coplanar and coaxial arrangement, the calculations being performed for a set of horizontal and vertical separations, respectively. Results are found using Faraday's law, the proposed inductive field, and the conventional electric field acceleration term.

The calculations are done in a set of Mathematica modules in this notebook. The output of the calculations are two tables in the section *Comparison of generated EMFs*. If your the goal is simply to evaluate different circuit geometries and separations, this can be achieved by modifying parameters in that section and evaluating the notebook.

Note that the vertical separation symbol used here differs from that used in the paper. In the paper it is *v*, and here it is *h*.

Symbol definitions

The layout of the circuits and the symbol definitions are shown below. The driven circuit has a changing current applied to it while the induced voltage is calculated for the measured circuit. Note that in the paper the vertical separation symbol used is *v* while *h* is used here.



Equations for EMF using Faraday's law

The change of electromagnetic flux through a point in space due to a current flowing in a wire of length

dl is

$$\frac{dB}{dt} = \frac{\mu_0}{4\pi}\frac{\dot{\mathbf{I}} d\ell \times \mathbf{r}}{|\mathbf{r}|^3} \qquad (1)$$

For two circular wires the EMF generated will be

$$\varepsilon = \frac{\mu_0}{4\pi}\int_0^{2\pi}\int_0^{r_{m_o}}\int_0^{2\pi} r_m \frac{\left[\left(\vec{\mathbf{I}}\, r_{d_o} d\phi\right) \times \vec{\mathbf{r}}\right] \cdot \vec{\mathbf{k}}}{|\vec{\mathbf{r}}|^3}\, dr_m\, d\theta \qquad (2)$$

where the angles and dimensions are from the circuit diagram above. This is the integral over the area of the measured circuit for all points along the driven wire. The distance between a wire segment and a point on the wire is

$$\vec{\mathbf{r}} = (r_{d_0}(1 - cos\theta) + s + r_m - r_m\, cos\phi)\,\vec{\mathbf{i}} + (-r_{d_0}\, sin\theta + r_m\, sin\phi)\,\vec{\mathbf{j}} + h\,\vec{\mathbf{k}} \qquad (3)$$

The electric current vector at any point is parallel to the driven wire and is

$$\vec{\mathbf{I}} = \dot{I}\left[-sin\theta\,\vec{\mathbf{i}} + cos\theta\,\vec{\mathbf{j}}\right] \qquad (4)$$

Equations for EMF using the inductive field

For the proposed inductive field acceleration term the EMF is the sum of the forces from one segment of the driven wire circuit onto a segment of the measured wire circuit. The integral is

$$\varepsilon = -\frac{1}{4\pi c^2 \epsilon_0}\int_0^{2\pi}\int_0^{2\pi} \frac{\vec{\mathbf{I}}}{|\vec{\mathbf{r}}|} \cdot \left[-sin\varphi\,\vec{\mathbf{i}} + cos\varphi\,\vec{\mathbf{j}}\right] r_{d_o} r_{m_o}\, d\theta\, d\varphi \qquad (5)$$

For the conventional acceleration term the time derivative of the current is replaced by the double cross product term.

$$\varepsilon = \frac{1}{4\pi c^2 \epsilon_0}\int_0^{2\pi}\int_0^{2\pi} \frac{\vec{\mathbf{r}} \times (\vec{\mathbf{r}} \times \vec{\mathbf{I}})}{|\vec{\mathbf{r}}|^3} \cdot \left[-sin\varphi\,\vec{\mathbf{i}} + cos\varphi\,\vec{\mathbf{j}}\right] r_{d_o} r_{m_o}\, d\theta\, d\varphi \qquad (6)$$

The value of the vector ***r*** is found from

$$\vec{\mathbf{r}} = [r_{d_o}(1 - cos\theta) + s + r_{m_o}(1 + cos\phi)]\,\vec{\mathbf{i}} + [r_{m_o} sin\phi - r_{d_o} sin\theta]\,\vec{\mathbf{j}} + v\,\vec{\mathbf{k}} \qquad (7)$$

Modules overview

The numerical solutions were arrived at using a set of Mathematica modules, each one responsible for either part of the calculation or combining the results. They are arranged rather like classes in object oriented languages.

The most likely use of this notebook is to replicate the results of the paper for the same or different geometries. This is most easily done by copying the section *Comparison of generated EMFs* and modify-

ing geometries and separations.

Aside from inspecting the source code, a deeper understanding of the modules can come from browsing the unit tests, their input, and their output.

Comparison of generated EMFs

The table below shows the numerical results of the EMF generated using Faraday's law, the proposed acceleration term, and the conventional acceleration term. The EMF for the Faraday and proposed acceleration terms are equal or differ in the last few least significant digits. The conventional acceleration term is consistently close to zero.

If you wish to try other geometries, you can copy this section and change the arguments in the call to **ComparisonOfEmfForCircularCircuits**, or just do so below and then reevaluate the notebook.

```
ComparisonOfEmfForCircularCircuits[
    {0.005, 0.010, 0.025, 0.075},
    0.1,
    0.05]["writeResultsToTable"][]
```

		Coplanar (μV)				Coaxial (μV)		
separation (mm)		Faraday	Prop accel	Conv accel		Faraday	Prop accel	Conv accel
5.		20.6626	20.6626	6.73972×10^{-8}		−54.5454	−54.5454	8.51429×10^{-7}
10.		15.7659	15.7659	-6.96794×10^{-6}		−53.6194	−53.6195	8.33278×10^{-7}
25.		9.1054	9.10571	-8.03869×10^{-6}		−47.9665	−47.9665	6.53919×10^{-7}
75.		3.03574	3.03574	-1.16369×10^{-6}		−23.8586	−23.8586	6.50481×10^{-7}

Common functionality

This is the common base 'class' for the Faraday and inductive field modules. It mostly has a helper function for the constructing geometry of the circuits.

```
BaseObject[] := Module[
    {
        this = <||>, wiresFromCorners, log,
    makePublic, wireList, addCorner, iCorner, subs, argList
    },

   wiresFromCorners[
     wireCorners_: {_?NumericQ ..},
     offsetCornersBy_: {_?NumericQ ..},
     isClosed_?BooleanQ
    ] := (
     wireList = {};
     addCorner[start_, end_] := AppendTo[wireList,
        <| "start" → wireCorners[[start]] + offsetCornersBy,
         "end" →  wireCorners[[end]] + offsetCornersBy,
         "wireId" → start|>];
     For[iCorner = 1, iCorner < Length[wireCorners], ++iCorner,
      addCorner[iCorner, iCorner + 1];
     ];
     If[isClosed, addCorner[iCorner, 1]];
     wireList
    );

    (* Assign a value to 'this'. The argument is a list of the form
        { string, value, string value,...}
    *)
    makePublic[thisArg_, stringList_] := (
        subs = Partition[stringList, 2];
        argList = thisArg;
        Scan[(argList = AssociateTo[argList, #[[1]] -> #[[2]]]) &, subs];
        argList
    );

    log[name_, value_] := Print[name <> ": " <> TextString[value]];
    this = makePublic[this, {"wiresFromCorners",
      wiresFromCorners, "log", log, "makePublic", makePublic}];
    this
];
```

EMF from Faraday's law

This module calculates the induced EMF from Faraday's law.

```
FaradayEmfForCircularCircuits[] := Module[
  {
```

```
 (* public values *)
 this = <||>,

 (* constants *)
 e0, c, mu0,

 (* public functions *)
 faradayEmf, rVector,

 (* local values *)
 emf, rVec, dIdtVec, emfPerLen
},
this = BaseObject[];

(* Constants *)
e0 = 8.8541878128*^-12;
c = 299 792 458;
mu0 = 1 / (e0 c^2); (* 1.2566370621219*^-6 *)

rVector[
  rDrvOuter_, (* Driven circuit radius (m) *)
  rMsrOuter_, (* Measured circuit radius (m) *)
  separation_, (* Distance between nearest points of the wire (m) *)
  vertSeparation_ ,
  (* Vertical separation of circuits (m). 0 for coplanar circuits *)
  rMsr_, (* Measured circuit radius at evaluation point (m) *)
  drvCircuitAngle_, (* The angle around the driven circuit (radians) *)
  msrCircuitAngle_(* The angle around the measured circuit (radians) *)
  ] := (
  {
   rDrvOuter (1 - Cos[drvCircuitAngle]) +
    separation + rMsrOuter + rMsr Cos[msrCircuitAngle],
   rMsr Sin[msrCircuitAngle] - rDrvOuter Sin[drvCircuitAngle],
   vertSeparation
  }
 );

faradayEmf[
  dIdt_, (* Change of current (A/s) *)
  rDrvOuter_, (* Driven circuit radius (m) *)
  rMsrOuter_, (* Measured circuit radius (m) *)
  separation_, (* Distance between nearest points of the wire (m) *)
  vertSeparation_
```

```
    (* Vertical separation of circuits (m). 0 for coplanar circuits *)
   ] := (
    emf = mu0 / (4 Pi) NIntegrate[
       rVec = rVector[ rDrvOuter, rMsrOuter, separation ,
         vertSeparation, rMsr, drvCircuitAngle, msrCircuitAngle];
       dIdtVec = dIdt {-Sin[drvCircuitAngle], Cos[drvCircuitAngle], 0};
        rMsr rDrvOuter Dot[ Cross[dIdtVec, rVec], {0, 0, 1}] / Norm[rVec]^3,
        {rMsr, 0, rMsrOuter},
        {msrCircuitAngle, 0, 2 Pi},
        {drvCircuitAngle, 0, 2 Pi},
        Method -> "LocalAdaptive"
        (*,PrecisionGoal→10,
        IntegrationMonitor:→((errors=Through[#1@"Error"])&)*)
      ];
    (*Print["errs: " <> TextString[errors]];*)
    emf
   );

  this = this["makePublic"][this, {
     "faradayEmf", faradayEmf,
     "rVector", rVector
    }];
  this
 ]
```

EMF from inductive field

This module calculates the induced EMF from the proposed inductive field and the conventional acceleration term.

```
InductiveFieldEmfForCircularCircuits[] := Module[
  {
   (* public values *)
   this = <||>,

   (* constants *)
   e0, c,

   (* circular to circular functions *)
   setProposedAccelTermCoeff, setConventionalAccelTermCoeff, totalEmf,
   rVector, emfFromPosOnCircSrcAtPosOnCircMsrWire, emfBetweenPoints,
```

```
 (* local values *)
 proposedAccelTermCoeff,
 conventionalAccelTermCoeff, r, rUnit, rLength, rVec, accelVec
},
this = BaseObject[];

(* Constants *)
e0 = 8.8541878128*^-12;
c = 299 792 458;

proposedAccelTermCoeff = 1;
conventionalAccelTermCoeff = 1;

(* These two functions allow for switching between the proposed
 inductive field term or the conventional acceleration term. *)
setProposedAccelTermCoeff[coeff_] := proposedAccelTermCoeff = coeff;
setConventionalAccelTermCoeff[coeff_] := conventionalAccelTermCoeff = coeff;

(* Calculate the EMF separated by a vector r. Use of the proposed
 or convenctional term is based on the proposedAccelTermCoeff or
  conventionalAccelTermCoeff set by the two functions above. *)
emfBetweenPoints[dIdtVec_, r_] := (
  rUnit = Normalize[r];
  rLength = Norm[r];
  (
    conventionalAccelTermCoeff Cross[rUnit, Cross[rUnit, dIdtVec] ]
     - proposedAccelTermCoeff dIdtVec
   ) / (4 Pi e0 rLength c^2)
 );

rVector[
  rDrvOuter_, (* Driven circuit radius (m) *)
  rMsrOuter_, (* Measured circuit radius (m) *)
  separation_, (* Distance between nearest points of the wire (m) *)
  vertSeparation_ ,
  (* Vertical separation of circuits (m). 0 for coplanar circuits *)
  drvCircuitAngle_, (* The angle around the driven circuit (radians) *)
  msrCircuitAngle_(* The angle around the measured circuit (radians) *)
  ] := (
  {
   rDrvOuter (1 - Cos[drvCircuitAngle]) +
    separation + rMsrOuter (1 + Cos[msrCircuitAngle]),
   rMsrOuter Sin[msrCircuitAngle] - rDrvOuter Sin[drvCircuitAngle],
```

```
    vertSeparation
   }
  );

 (* This function calulates the
  total EMF in measured wire using the LW theory. *)
 totalEmf[
   dIdt_, (* Change of current (A/s) *)
   rDrvOuter_, (* Driven circuit radius (m) *)
   rMsrOuter_, (* Measured circuit radius (m) *)
   separation_, (* Distance between nearest points of the wire (m) *)
   vertSeparation_
   (* Vertical separation of circuits (m). 0 for coplanar circuits *)
  ] := (
   NIntegrate[

    rVec = rVector[rDrvOuter, rMsrOuter, separation,
      vertSeparation, drvCircuitAngle, msrCircuitAngle];

    accelVec = dIdt {-Sin[drvCircuitAngle], Cos[drvCircuitAngle] , 0};

    rDrvOuter rMsrOuter Dot[
      emfBetweenPoints[accelVec, rVec],
      {-Sin[msrCircuitAngle], Cos[msrCircuitAngle] , 0}
     ],

    {drvCircuitAngle, 0, 2 Pi},
    {msrCircuitAngle, 0, 2 Pi},
    Method -> "LocalAdaptive"
   ]
  );


 this = this["makePublic"][this, {
    "totalEmf", totalEmf,
    "setProposedAccelTermCoeff", setProposedAccelTermCoeff,
    "setConventionalAccelTermCoeff", setConventionalAccelTermCoeff,
    "emfBetweenPoints", emfBetweenPoints,
    "rVector", rVector
   }];
 this
]
```

Module for comparison of EMFs generated by Faradays law and electric field terms

The calculation and tabulation of all results is done with this class.

```
ComparisonOfEmfForCircularCircuits[
   separations_,
   drivenRadius_,
   measuredRadius_
  ] := Module[
   {
    (* public values *)
    this = <||>,

    (* functions *)
    writeResultsToTable,
    generateFaraday, generateProposedAccel, generateConvAccel,

    (* constants *)
    dIdt = 1000.0,

    (* local values *)
    faraday, eFieldProposed, eFieldConv, faradayEmfCoplanar,
    proposedAccelEmfCoplanar, convAccelEmfCoplanar, faradayEmfCoaxial,
    proposedAccelEmfCoaxial, convAccelEmfCoaxial, resultsArray,
    emf, separationForCoAxial, micro = 1.0*^6, milli = 1.0*^3
   },
   this = BaseObject[];
   faradayEmfCoplanar = {};
   proposedAccelEmfCoplanar = {};
   convAccelEmfCoplanar = {};
   faradayEmfCoaxial = {};
   proposedAccelEmfCoaxial = {};
   convAccelEmfCoaxial = {};
   separationForCoAxial = -(drivenRadius + measuredRadius);

   writeResultsToTable[] := (
     generateFaraday[];
     generateProposedAccel[];
     generateConvAccel[];
```

```
  resultsArray = {
    {"", "", "Coplanar (μV)", SpanFromLeft,
     SpanFromLeft, "", "Coaxial (μV)", SpanFromLeft, SpanFromLeft},
    {"separation (mm)", "", "Faraday", "Prop accel", "Conv accel",
     "   ", "Faraday", "Prop accel", "Conv accel"}
   };
  For[i = 1, i <= (separations // Length), i++,
   row = { milli separations[[i]] };
   row = Join[row, {"   "}];
   row = Join[row, {faradayEmfCoplanar[[i]],
      proposedAccelEmfCoplanar[[i]], convAccelEmfCoplanar[[i]]}];
   row = Join[row, {"   "}];
   row = Join[row, {faradayEmfCoaxial[[i]],
      proposedAccelEmfCoaxial[[i]], convAccelEmfCoaxial[[i]]}];
   AppendTo[resultsArray, row]
  ];
  TextGrid[resultsArray, Frame → All]
 );

generateFaraday[] := (
  If[Length[faradayEmfCoplanar] == 0,
   (
    faraday = FaradayEmfForCircularCircuits[];

    Scan[
     (
       emf =
        faraday["faradayEmf"][dIdt, drivenRadius, measuredRadius, #1, 0.0];
       AppendTo[faradayEmfCoplanar, -micro emf]) &,
     separations
    ];

    Scan[
     (
       emf = faraday["faradayEmf"][dIdt,
         drivenRadius, measuredRadius, separationForCoAxial, #1];
       AppendTo[faradayEmfCoaxial, -micro emf]) &,
     separations
    ];
   )
  ];
  {faradayEmfCoplanar, faradayEmfCoaxial}
 );
```

```
generateProposedAccel[] := (
  If[Length[proposedAccelEmfCoplanar] == 0,
   (
    eFieldProposed = InductiveFieldEmfForCircularCircuits[];
    eFieldProposed["setConventionalAccelTermCoeff"][0];
    eFieldProposed["setProposedAccelTermCoeff"][1];

    Scan[
     (
       emf = eFieldProposed["totalEmf"][
         dIdt, drivenRadius, measuredRadius, #1, 0.0];
       AppendTo[proposedAccelEmfCoplanar, micro emf]) &,
     separations
    ];

    Scan[
     (
       emf = eFieldProposed["totalEmf"][dIdt,
         drivenRadius, measuredRadius, separationForCoAxial, #1];
       AppendTo[proposedAccelEmfCoaxial, micro emf]) &,
     separations
    ];
   )
  ];
  {proposedAccelEmfCoplanar, proposedAccelEmfCoaxial}
 );

generateConvAccel[] := (
  If[Length[convAccelEmfCoplanar] == 0,
   (
    eFieldConv = InductiveFieldEmfForCircularCircuits[];
    eFieldConv["setConventionalAccelTermCoeff"][1];
    eFieldConv["setProposedAccelTermCoeff"][0];

    Scan[
     (
       emf =
        eFieldConv["totalEmf"][dIdt, drivenRadius, measuredRadius, #1, 0.0];
       AppendTo[convAccelEmfCoplanar, micro emf]) &,
     separations
    ];
```

```
      Scan[
       (
         emf = eFieldConv["totalEmf"][dIdt,
           drivenRadius, measuredRadius, separationForCoAxial, #1];
         AppendTo[convAccelEmfCoaxial, micro emf]) &,
       separations
      ];
     )
    ];
    {convAccelEmfCoplanar, convAccelEmfCoaxial}
   );

  this = this["makePublic"][this, {
     "writeResultsToTable", writeResultsToTable,
     "generateFaraday", generateFaraday,
     "generateProposedAccel", generateProposedAccel,
     "generateConvAccel", generateConvAccel
    }];
  this
 ];
```

Unit tests for EMF from Faraday's law

This section contains the module for unit tests for EMF generated using Faraday's law. Any errors will be noted in the output with red text highlighting the expected and actual output.

```
UnitTestsForFaradayEmfForCircularCircuits[
  verbose_: True, skipIntensiveTests_: False] := Module[
  {
   (* public values *)
   this = <||>,

   (* functions *)
   EXPECTNEAR, emfById, skipTests, log,

   (* local values *)
   micro = 1000000, milli = 1000, mu0, faraday, dIdt1, sep1,
   vertSep1, drvPntToCntrOfMsr1, rDrv1, rMsr1, r1, r2, r3, r4, r5, r6,
   vertSep2, emf7, dIdt8, radiusDrv8, radiusMsr8, sep8, emf8, dIdt9,
   rDrv9, rMsr9, oneMmSep9, emf9, sep10, emf10, vertSep11, emf11
  },
  this = UnitTestBaseObject[verbose, skipIntensiveTests];
  EXPECTNEAR = this["EXPECTNEAR"];
```

```
skipTests = this["skipTests"];
log = this["log"];

faraday = FaradayEmfForCircularCircuits[];

(* Test 1 *)
dIdt1 = 6000;
sep1 = 0.01;
vertSep1 = 0.0;
rDrv1 = 0.07;
rMsr1 = 0.05;
r1 = faraday["rVector"][rDrv1, rMsr1, sep1, vertSep1, rMsr1,  0, 180 Degree];
EXPECTNEAR[{sep1, 0, 0}, r1, 0.000001];

(* Test 2 *)
r2 = faraday["rVector"][rDrv1, rMsr1, sep1, vertSep1, 0.5 rMsr1,  0, 0];
EXPECTNEAR[{1.5 rMsr1 + sep1, 0, 0}, r2, 0.000001];

(* Test 3 *)
r3 = faraday["rVector"][rDrv1, rMsr1, sep1, vertSep1, rMsr1, 0, 90 Degree];
EXPECTNEAR[{ rMsr1 + sep1, rMsr1, 0}, r3, 0.000001];

(* Test 4 *)
r4 = faraday["rVector"][rDrv1,
  rMsr1, sep1, vertSep1, 0.25 rMsr1, 90 Degree, 90 Degree];
EXPECTNEAR[{ rMsr1 + sep1 + rDrv1, 0.25 rMsr1 - rDrv1, 0}, r4, 0.000001];

(* Test 5 *)
r5 = faraday["rVector"][rDrv1,
  rMsr1, sep1, vertSep1, 0.5 rMsr1, 180 Degree, 0 Degree];
EXPECTNEAR[{ 1.5 rMsr1 + sep1 + 2 rDrv1, 0, 0}, r5, 0.000001];

(* Test 6 *)
vertSep2 = 0.1;
r6 =
 faraday["rVector"][rDrv1, rMsr1, sep1, vertSep2, rMsr1,  180 Degree, 0 Degree];
EXPECTNEAR[{ 2 rMsr1 + sep1 + 2 rDrv1, 0, vertSep2}, r6, 0.000001];

(* Test 7 *)
emf7 = faraday["faradayEmf"][dIdt1, rDrv1, rMsr1, sep1, vertSep1];
EXPECTNEAR[76.4685, -micro emf7, 0.0001];

(* Test 8 : EMF for dimensions matching experiments *)
dIdt8 = 13000.0; (* Change of current (A/s) *)
```

```
  radiusDrv8 = 0.05; (* Radius of driven circuit (m) *)
  radiusMsr8 = 0.0375; (* Radius of measured circuit (m) *)
  sep8 = 0.0075; (* Separation (m) with experimental EMF approx 0.000125 V *)
  emf8 = faraday["faradayEmf"][dIdt8, radiusDrv8, radiusMsr8, sep8, 0];
  EXPECTNEAR[120.294, -micro emf8, 0.001];

  (* Test 9 : EMF for dimensions matching other calculation *)
  dIdt9 = 6110.3534985431;
  rDrv9 = 0.05;
  rMsr9 = 0.02;
  oneMmSep9 = 0.001;
  emf9 = faraday["faradayEmf"][dIdt9, rDrv9, rMsr9, oneMmSep9, 0];
  EXPECTNEAR[66.7189, -micro emf9, 0.0001];

  (* Test 10 : EMF for dimensions matching experiments *)
  sep10 = 0.00075;
  emf10 = faraday["faradayEmf"][dIdt8, radiusDrv8, radiusMsr8, sep10, 0];
  EXPECTNEAR[238.08, -micro emf10, 0.001];

  (* Test 11 : Inline  EMF for dimensions matching experiments *)
  vertSep11 = 0.00075;
  emf11 = faraday["faradayEmf"]⎡dIdt8,
    radiusDrv8, radiusMsr8, -(radiusDrv8 + radiusMsr8), vertSep11⎤;
  EXPECTNEAR[966.787, micro emf11, 0.001];

  this["printResults"][];

  this = this["makePublic"][this, {
    }];
  this
 ];
UnitTestsForFaradayEmfForCircularCircuits[True, False];
```

```
Test: 1  Expected: {0.01, 0, 0}  Actual: {0.01, 0.0, 0.0}
Test: 2  Expected: {0.085, 0, 0}  Actual: {0.085, 0.0, 0.0}
Test: 3  Expected: {0.06, 0.05, 0}  Actual: {0.06, 0.05, 0.0}
Test: 4  Expected: {0.13, -0.0575, 0}  Actual: {0.13, -0.0575, 0.0}
Test: 5  Expected: {0.225, 0, 0}  Actual: {0.225, 0.0, 0.0}
Test: 6  Expected: {0.25, 0, 0.1}  Actual: {0.25, 0.0, 0.1}
Test: 7  Expected: 76.4685  Actual: 76.4685
Test: 8  Expected: 120.294  Actual: 120.294
Test: 9  Expected: 66.7189  Actual: 66.7189
```

```
Test: 10  Expected: 238.08  Actual: 238.08
Test: 11  Expected: 966.787  Actual: 966.787
PASSED 11 tests
```

Unit tests for EMF from the inductive field

This section contains the module for unit tests for inductive field EMF. Any errors will be noted in the output with red text highlighting the expected and actual output.

```
UnitTestsForInductiveFieldEmfForCircularCircuits[
  verbose_: True, skipIntensiveTests_: False] := Module[
  {
   (* public values *)
   this = <||>,

   (* functions *)
   EXPECTNEAR, emfById, skipTests, log,

   (* local values *)
   micro = 1000000, milli = 1000, mu0, eField, dIdt1, sep1, vertSep1,
   drvPntToCntrOfMsr1, rDrv1, rMsr1, r1, r2, r3, r4, r5, r6, vertSep2, emf7,
   dIdt8, radiusDrv8, radiusMsr8, sep8, emf8, dIdt9, rDrv9, rMsr9, oneMmSep9,
   emf9, sep10, emf10, emf11, emf12, emf13, emf14, vertSep15, emf15, emf16
  },
  this = UnitTestBaseObject[verbose, skipIntensiveTests];
  EXPECTNEAR = this["EXPECTNEAR"];
  skipTests = this["skipTests"];
  log = this["log"];

  eField = InductiveFieldEmfForCircularCircuits[];

  (* Test 1 *)
  dIdt1 = 6000;
  sep1 = 0.01;
  vertSep1 = 0.0;
  rDrv1 = 0.07;
  rMsr1 = 0.05;
  r1 = eField["rVector"][rDrv1, rMsr1, sep1, vertSep1, 0, 180 Degree];
  EXPECTNEAR[{sep1, 0, 0}, r1, 0.000001];

  (* Test 2 *)
  r2 = eField["rVector"][rDrv1, rMsr1, sep1, vertSep1, 0, 0];
  EXPECTNEAR[{2 rMsr1 + sep1, 0, 0}, r2, 0.000001];
```

```
(* Test 3 *)
r3 = eField["rVector"][rDrv1, rMsr1, sep1, vertSep1, 0, 90 Degree];
EXPECTNEAR[{ rMsr1 + sep1, rMsr1, 0}, r3, 0.000001];

(* Test 4 *)
r4 = eField["rVector"][rDrv1, rMsr1, sep1, vertSep1, 90 Degree, 90 Degree];
EXPECTNEAR[{ rMsr1 + sep1 + rDrv1, rMsr1 - rDrv1, 0}, r4, 0.000001];

(* Test 5 *)
r5 = eField["rVector"][rDrv1, rMsr1, sep1, vertSep1, 180 Degree, 0 Degree];
EXPECTNEAR[{ 2 rMsr1 + sep1 + 2 rDrv1, 0, 0}, r5, 0.000001];

(* Test 6 *)
vertSep2 = 0.1;
r6 = eField["rVector"][rDrv1, rMsr1, sep1, vertSep2, 180 Degree, 0 Degree];
EXPECTNEAR[{ 2 rMsr1 + sep1 + 2 rDrv1, 0, vertSep2}, r6, 0.000001];

(* Test 7 *)
eField["setConventionalAccelTermCoeff"][0];
eField["setProposedAccelTermCoeff"][1];
emf7 = eField["totalEmf"][dIdt1, rDrv1, rMsr1, sep1, vertSep1];
EXPECTNEAR[76.4685, micro emf7, 0.0001];

(* Test 8 : EMF for dimensions matching experiments *)
dIdt8 = 13000.0; (* Change of current (A/s) *)
radiusDrv8 = 0.05; (* Radius of driven circuit (m) *)
radiusMsr8 = 0.0375; (* Radius of measured circuit (m) *)
sep8 = 0.0075; (* Separation (m) with experimental EMF approx 0.000125 V *)
emf8 = eField["totalEmf"][dIdt8, radiusDrv8, radiusMsr8, sep8, 0];
EXPECTNEAR[120.294, micro emf8, 0.001];

(* Test 9 : EMF for dimensions matching other calculation *)
dIdt9 = 6110.3534985431;
rDrv9 = 0.05;
rMsr9 = 0.02;
oneMmSep9 = 0.001;
emf9 = eField["totalEmf"][dIdt9, rDrv9, rMsr9, oneMmSep9, 0];
EXPECTNEAR[66.7189, micro emf9, 0.002];

(* Test 10 : EMF for dimensions matching experiments *)
sep10 = 0.00075;
emf10 = eField["totalEmf"][dIdt8, radiusDrv8, radiusMsr8, sep10, 0];
EXPECTNEAR[238.08, micro emf10, 0.001];
```

```
(* Test 11 *)
eField["setConventionalAccelTermCoeff"][1];
eField["setProposedAccelTermCoeff"][0];
emf11 = eField["totalEmf"][dIdt1, rDrv1, rMsr1, sep1, vertSep1];
EXPECTNEAR[0.0, micro emf11, 0.0001];

(* Test 12 : EMF for dimensions matching experiments *)
emf12 = eField["totalEmf"][dIdt8, radiusDrv8, radiusMsr8, sep8, 0];
EXPECTNEAR[0.0, micro emf12, 0.001];

(* Test 13 : EMF for dimensions matching other calculation *)
emf13 = eField["totalEmf"][dIdt9, rDrv9, rMsr9, oneMmSep9, 0];
EXPECTNEAR[0.0, micro emf13, 0.002];

(* Test 14 : EMF for dimensions matching experiments *)
sep14 = 0.00075;
emf14 = eField["totalEmf"][dIdt8, radiusDrv8, radiusMsr8, sep14, 0];
EXPECTNEAR[0.0, micro emf14, 0.001];

(* Test 15 : Inline  EMF for dimensions matching experiments *)
vertSep15 = 0.00075;
eField["setConventionalAccelTermCoeff"][0];
eField["setProposedAccelTermCoeff"][1];
emf15 = eField["totalEmf"][dIdt8, radiusDrv8,
  radiusMsr8, -(radiusDrv8 + radiusMsr8), vertSep15];
EXPECTNEAR[-966.781, micro emf15, 0.001];

(* Test 16 : EMF for dimensions matching experiments *)
eField["setConventionalAccelTermCoeff"][1];
eField["setProposedAccelTermCoeff"][0];
emf16 = eField["totalEmf"][dIdt8, radiusDrv8,
  radiusMsr8, -(radiusDrv8 + radiusMsr8), vertSep15];
EXPECTNEAR[-0.0, micro emf16, 0.001];

(*{0.00075,0.00189,0.00326,0.00567,0.0072,0.0091,0.01167,0.01624,0.02078,
 0.02744,0.03549,0.04454,0.05188,0.06231} (* Circuit separations (m) *),
{0.00023205,0.00019578,0.00016800,0.00013753,0.00012410,0.00010930,0.00009363,
 0.00007438,0.00006093,0.00004695,0.00003585,0.00002725,0.00002070,0.00001693}
*)

this["printResults"][];
```

```
  this = this["makePublic"][this, {
    }];
  this
 ];
UnitTestsForInductiveFieldEmfForCircularCircuits[True, False];
```

```
Test: 1  Expected: {0.01, 0, 0}  Actual: {0.01, 0.0, 0.0}
Test: 2  Expected: {0.11, 0, 0}  Actual: {0.11, 0.0, 0.0}
Test: 3  Expected: {0.06, 0.05, 0}  Actual: {0.06, 0.05, 0.0}
Test: 4  Expected: {0.13, -0.02, 0}  Actual: {0.13, -0.02, 0.0}
Test: 5  Expected: {0.25, 0, 0}  Actual: {0.25, 0.0, 0.0}
Test: 6  Expected: {0.25, 0, 0.1}  Actual: {0.25, 0.0, 0.1}
Test: 7  Expected: 76.4685  Actual: 76.4685
Test: 8  Expected: 120.294  Actual: 120.293
Test: 9  Expected: 66.7189  Actual: 66.7177
Test: 10  Expected: 238.08  Actual: 238.08
Test: 11  Expected: 0.0  Actual: -0.0000155353
Test: 12  Expected: 0.0  Actual: -0.00000394387
Test: 13  Expected: 0.0  Actual: -0.00000224778
Test: 14  Expected: 0.0  Actual: -0.00000927701
Test: 15  Expected: -966.781  Actual: -966.781
Test: 16  Expected: 0.0  Actual: -0.000025873
PASSED 16 tests
```

Unit tests for the base/common object

These are the unit tests for the BaseObject module. Any errors will be noted in the output with red text highlighting the expected and actual output.

```
UnitTestBaseObject[verbose_: True, skipIntensiveTests_: False] := Module[
   {
    (* public values *)
    this = <||>,

    (* functions *)
    EXPECTNEAR, emfById, skipTests, printResults,

    (* local values *)
    numTestsPassed, numTestsFailed, numTestsSkipped,
    testNumber, toUsed, actualTypeString, micro = 1000000, milli = 1000
   },
```

```
this = BaseObject[];
numTestsPassed = 0;
 numTestsFailed = 0;
numTestsSkipped = 0;
testNumber = 1;

skipTests[from_, to_: 0] := (
  toUsed = to;
  If[toUsed == 0, toUsed = from];
  If[skipIntensiveTests,
   Print[
    "Skipping tests " <> TextString[from] <> " to " <> TextString[toUsed]];
   numTestsSkipped += toUsed - from + 1;
   testNumber += toUsed - from + 1;
  ];
  skipIntensiveTests
 );

EXPECTNEAR[expected_, actual_, eps_] := (
  If[verbose,
   Print["Test: " <> TextString[testNumber] <> "  Expected: " <>
      TextString[expected] <> "  Actual: " <> TextString[actual]];
  ];
  testNumber++;

  actualTypeString = TextString[Head[actual]];

  If[! (actualTypeString == "List" || actualTypeString == "Real" ||
       actualTypeString == "Integer") || Max[Abs[expected - actual]] > eps,
   (
    Print[Style["Expected: " <> TextString[expected], Red]];
    Print[Style["Actual: " <> TextString[actual], Red]];
    Print[Style["Difference: " <> TextString[expected - actual], Red]];
    Print[Style["EPS: " <> TextString[eps], Red]];
    numTestsFailed++;
   ),
   (
    numTestsPassed++
   )
  ];
 );

printResults[] := (
```

```
   Print["PASSED " <> TextString[numTestsPassed] <> " tests"];
   If[numTestsFailed > 0,
    Print["FAILED: " <> TextString[numTestsFailed]];
   ];
  );

 this =
  this["makePublic"][this, {"skipTests", skipTests, "EXPECTNEAR", EXPECTNEAR,
    "numTestsPassed", numTestsPassed, "numTestsFailed", numTestsFailed,
    "numTestsSkipped", numTestsSkipped, "printResults", printResults}];
 this
];
```