Sample Section of
The Mathematica GuideBook for Numerics

This document was downloaded from http://www.mathematicaguidebooks.org. It contains extracted material from the notebooks accompanying *The Mathematica GuideBook for Numerics*. Copyright Springer 2005/2006. Redistributed with permission.

1.12 Two Applications

■ 1.12.1 Visualizing Electric and Magnetic Field Lines

In this subsection, we will calculate and visualize electric and magnetic field lines for some simple electrostatic and magnetostatic field configurations.

Field lines are curves that are parallel to the local field direction. Let $\vec{C}(\vec{x})$ be the vector field describing the electric field $\vec{E}(\vec{x})$ or magnetic field $\vec{B}(\vec{x})$ at the point $\vec{x}$, and let $\vec{c}(t)$ be a parametrized field line. The tangent of the field line points in the direction of the field at each point; this means $\vec{c}'(t) \sim \vec{C}(\vec{c}(t))$ [1686✱], [1020✱]. The set of coupled nonlinear ordinary differential equations is the defining set of equations for the field lines that we will solve numerically in this subsection.

In the electrostatic case, given the charge density $\rho(\vec{y})$, the electric field $\vec{E}(\vec{x})$ is given as the Poisson integral (ignoring units and powers of 4π; $d\vec{y}$ is the 3D volume element)

$$\vec{E}(\vec{x}) = \int_{\mathbb{R}^3} \frac{(\vec{x}-\vec{y})\,\rho(\vec{y})}{|\vec{x}-\vec{y}|^3}\, d\vec{y}.$$

For a point charge q at the point $\vec{r}$, this simplifies to $\vec{E}(\vec{x}) = q(\vec{x}-\vec{r})/|\vec{x}-\vec{r}|^3$.

Often in this subsection, we will have to normalize vectors. We will repeatedly make use of the two functions `norm` and n.

```
In[1]:= (* length of a vector *)
        norm[vector_] := Sqrt[vector.vector]
        (* normalizing vectors *)
        n[vector_] := vector/Sqrt[vector.vector]
```

Let us start the field line visualization with a simple example: point charges in two dimensions. (We imagine them embedded in three dimensions and use the r^{-1} potential instead of a pure 2D, this means $\varphi(r) = -\ln(r)$ potential.) Here, we will calculate the field lines by explicitly solving the corresponding differential equations; for 2D configurations, it is often possible to use complex variable techniques to obtain the equipotential and the field lines [1686✱], [311✱], [1389✱], [1390✱], [206✱], [1890✱].

Here is a square array of 5×5 alternatingly charged points.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[5]:= charges = Flatten[Table[{(-1)^(i + j), {i, j}},
                               {i, -2, 2}, {j, -2, 2}], 1];
```

φ`Coulomb` calculates the (normalized—no units) the Coulomb potential of a single charge q at position *pos*. We will use φ`Coulomb` for 2D as well as for 3D charge configurations.

```
In[6]:= φCoulomb[{q_, pos_}, xyz_] := q/norm[xyz - pos]
```

By the superposition principle, the electrostatic potential is simply the sum of the potentials of all charges.

```
In[7]:= φ[{x_, y_}] = Plus @@ (φCoulomb[#, {x, y}]& /@ charges);
```

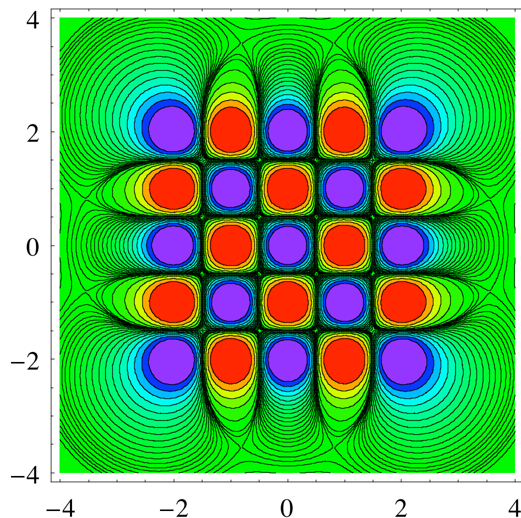
We generate a contour plot of the charge array.

```
In[8]:= cp = ContourPlot[Evaluate[φ[{x, y}]], {x, -4, 4}, {y, -4, 4},
                         PlotPoints -> 200, Contours -> 30,
                         ContourStyle -> {Thickness[0.002]},
                         DisplayFunction -> Identity];
```

For a nicer contour plot, we implement a function `makeContours` that will display a given contour graphic with n contour lines in such a way that the n contour lines are spaced as evenly as possible.

```
In[9]:= makeContours[ContourGraphics[data_, ___], n_] :=
        With[{λ = Sort[Flatten[data]]},
             (* generate about n equal length sets *)
             #[[Floor[Length[λ]/n/2]]]& /@
                       Partition[λ, Floor[Length[λ]/n]]]
```

```
In[10]:= equiPotentialPicture =
         ListContourPlot[cp[[1]], MeshRange -> {{-4, 4}, {-4, 4}},
             ColorFunction -> (Hue[# 0.75]&),  Contours -> makeContours[cp, 30],
              ContourStyle -> {{Thickness[0.002]}}]
```




```
Out[10]= - ContourGraphics -
```

This is the electric field corresponding to the charge array. We obtain it by using the formula $\vec{E}(\vec{x}) = -\vec{\nabla}\varphi(\vec{x})$.

```
In[11]:= EField[{x_, y_}] = -{D[φ[{x, y}], x], D[φ[{x, y}], y]};
```

For the calculation of the field lines $\vec{c}(t)$, we will not use $\vec{c}\,'(t) = \vec{E}(\vec{c}(t))$, but rather $\vec{c}\,'(t) = \vec{E}(\vec{c}(t)) \big/ \left|\vec{E}(\vec{c}(t))\right|$. This normalization $|\vec{c}\,'(t)| = 1$ of the field strength automatically gives the arc length of the field lines as the parameter t and gives a more

t

© 2004, 2005 Springer Science+Business Media, Inc.

$\vec{c}(t)$ $\vec{c}\,'(t) = \vec{E}(\vec{c}(t))$ $\vec{c}\,'(t) = \vec{E}(\vec{c}(t)) \big/ \left|\vec{E}(\vec{c}(t))\right|$
$|\vec{c}\,'(t)| = 1$ t
intuitive measure for the t-range required in solving the differential equations. The resulting field lines then have the natural parametrization.

Instead of defining *Ex*`[{`*x*`_,`*y*`_}] = `*n*`[`*EField*`[{`*x*`, `*y*`}]][[1]]`, *Ey*`[{`*x*`_,`*y*`_}] =`... for the components of the electric field, we define a more general function `makeFieldComponentDefinitions`. This function will create such definitions for a given list of field components *fieldComponents* and a given vector-valued field *field*. Because we will define a variety of different charge and current configurations in this subsection, the function `makeFieldComponentDefinitions` will in the long run make the inputs shorter and more readable. The optional last argument type *varType* has the following reason. When we do not supply it, the created definitions are of the form *Ex*`[{`*x*`_, `*y*`_}] := `*fieldComponent*. If we later would use *Ex*`[{`*x*`, `*y*`}]` on the right-hand side of a differential equation, *fieldComponent* will be substituted. `NDSolve` will compile *fieldComponent* and solve the differential equations. When calculating the x-, y-, and z-component of the fields, much of the computational work has to be repeated. To avoid redoing computational work, we will store the fields as vectors and access only their components. In such cases, we do not want *ex*`[{`*x*`, `*y*`}]` to evaluate for symbols (head `Symbol`) x and y. We avoid this by instead creating the definition *Ex*`[{`*x*`_Real, `*y*`_Real}] := `*fieldComponent*. Using `Real` for *varType* will generate such definitions.

```
In[12]:= makeFieldComponentDefinitions[
              fieldComponents_, field_, vars_, varType___] :=
         ((* remove existing definitions *)
          Clear /@ fieldComponents;
          (* create new definitions *)
          MapIndexed[(SetDelayed @@
                        (Hold[#1[Pattern[#, Blank[varType]]& /@ vars],
                              n[field[vars]][[C]]] /. C -> #2[[1]]))&,
                      fieldComponents])
```

Now, we invoke `makeFieldComponentDefinitions` to make the definitions for the field components.

```
In[13]:= makeFieldComponentDefinitions[{Ex, Ey}, EField, {x, y}];
```

These are the current definitions for the field components.

```
In[14]:= DownValues /@ {Ex, Ey}
Out[14]= {{HoldPattern[Ex[{x_, y_}]] :> n[EField[{x, y}]][[1]]},
          {HoldPattern[Ey[{x_, y_}]] :> n[EField[{x, y}]][[2]]}}
```

We separate the positive and the negative charge positions.

```
In[15]:= {positiveCharges, negativeCharges} =
           (Last /@ Cases[charges, {_?#, _}])& /@ {Positive, Negative};
```

The function `calculateFieldLine` calculates the field lines as `InterpolatingFunctions` by solving their differential equation numerically. To be flexible and to be able to use the function `calculateFieldLine` for the 2D and 3D case, and for the electric and for the magnetic field cases, we use the arguments *fieldLineVector*, *startPoint*, and *fields*. This allows us to specify the dimensions and the kind of field later.

```
In[16]:= calculateFieldLine[fieldLineVector_, startPoint_,
                            fields_, {t, t0_:0, t1_}, opts___] :=
         With[{flvt = #[t]& /@ fieldLineVector},
         NDSolve[Join[(* differential equation *)
                      Thread[D[flvt, t] == (#[flvt]& /@ fields)],
                      (* initial conditions *)
                      Thread[(flvt /. t -> t0) == startPoint]],
                 fieldLineVector, {t, t0, t1}, opts]]
```

Now, let us calculate the first set of field lines. We start near the positive charges and move toward the negative ones. To avoid integrating the differential equations near the singularity of the negative charges (which would need many steps without

© 2004, 2005 Springer Science+Business Media, Inc.

making progress in the field line), we stop the integration process (using the option `StoppingTest`) when we are too near to a negative charge.

```
In[17]:= startPoints[charges_, n_] :=
         Flatten[Table[charges[[j]] + ε {Cos[φ], Sin[φ]}, {j, Length[charges]},
                       {φ, 0, 2Pi(1 - 1/n), 2Pi/n}], 1]

In[18]:= sTest[cs_, pn_] := (Sqrt[Min[#.#& /@ ((cs - #)& /@ pn)]] < ε)

In[19]:= ε = 10^-2; n = 12;

         fieldLines["+" -> "-"] =
         calculateFieldLine[{cx, cy}, #, {Ex, Ey}, {t, 3},
            StoppingTest :> sTest[{cx[t], cy[t]}, negativeCharges]]& /@
         startPoints[positiveCharges, n];
```

Given the field lines in the form of `InterpolatingFunctions`, we use `ParametricPlot` to generate a line representing the field line.

```
In[21]:= fieldLineGraphic[cxcy_] :=
         ParametricPlot[Evaluate[{cx[t], cy[t]} /. cxcy],
                        Evaluate[Flatten[{t, cxcy[[1, 1, 2, 1, 1]]}]],
                        PlotPoints -> 300, DisplayFunction -> Identity]

In[22]:= fl["+" -> "-"] = fieldLineGraphic /@ fieldLines["+" -> "-"];
```

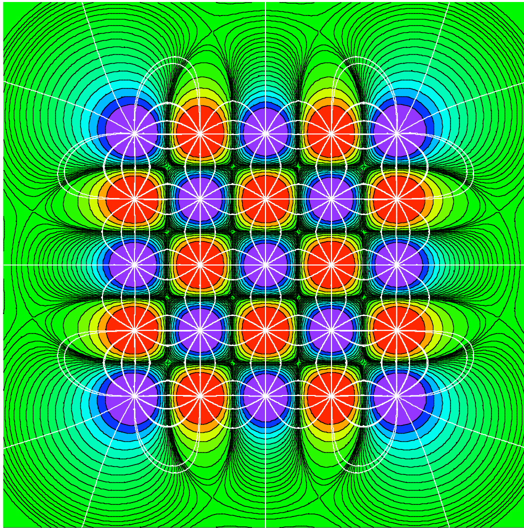
(As a side remark, we want to mention that the local density of the field lines does not reflect the local value of the field strength [1929✱], [641✱], [835✱].) Now, we do the same calculation, but we start at the negative charges. We could either integrate the equation $\vec{c}'(t) = \vec{E}(\vec{c}(t)) \big/ \left|\vec{E}(\vec{c}(t))\right|$ to move toward the positive charges or we move backwards with respect to t. We choose the latter approach; so, we can reuse the above definition of `calculateFieldLine`.

```
In[23]:= fieldLines["-" -> "+"] =
         calculateFieldLine[{cx, cy}, #, {Ex, Ey},
                                {t, (* go backwards *) -3},
             StoppingTest :> sTest[{cx[t], cy[t]}, positiveCharges]]& /@
                                             startPoints[negativeCharges, n];

In[24]:= fl["-" -> "+"] = fieldLineGraphic /@ fieldLines["-" -> "+"];
```

Now, we have all ingredients together to display the equicontour lines with the field lines (in white).

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[25]:= Show[{equiPotentialPicture,
         Graphics[{Thickness[0.002], GrayLevel[1],
                   Cases[{fl["+" -> "-"], fl["-" -> "+"]}, _Line, Infinity]}]},
              FrameTicks -> None, PlotRange -> {{-4, 4}, {-4, 4}}]
```



```
Out[25]= - Graphics -
```

In a similar way, we could treat nonsymmetric arrangements of charges. Here is an example made from 21 point charges. Each magnitude and position of the charges is randomly chosen. For the few charges under consideration here, we just sum their potentials (for more charges one could use faster methods [394✱], [1397✱], [587✱], [1437✱]; for visualizations of gradient fields in general, see [1573✱]).

```
In[26]:= SeedRandom[9999];
         n = 21;
         charges = Table[{Random[Real, {-1, 1}], {Random[], Random[]}}, {n}];

         (* split into positive and negative charges *)
         {positiveCharges, negativeCharges} =
           (Last /@ Cases[charges, {_?#, _}])& /@ {Positive, Negative};

         φ[{x_, y_}] = Plus @@ (φCoulomb[#, {x, y}]& /@ charges);

In[34]:= equiPotentialPicture = Function[cp, (* add features *)
                      ListContourPlot[cp[[1]],
                      MeshRange -> {{-1/4, 5/4}, {-1/4, 5/4}},
                      ColorFunction -> (Hue[# 0.75]&),
                      Contours -> makeContours[cp, 30],
                      ContourStyle -> {Thickness[0.002]},
                      DisplayFunction -> Identity]][
          (* the contour plot *)
          ContourPlot[Evaluate[φ[{x, y}]], {x, -1/4, 5/4}, {y, -1/4, 5/4},
                      PlotPoints -> 200, Contours -> 50,
                      DisplayFunction -> Identity]];
```

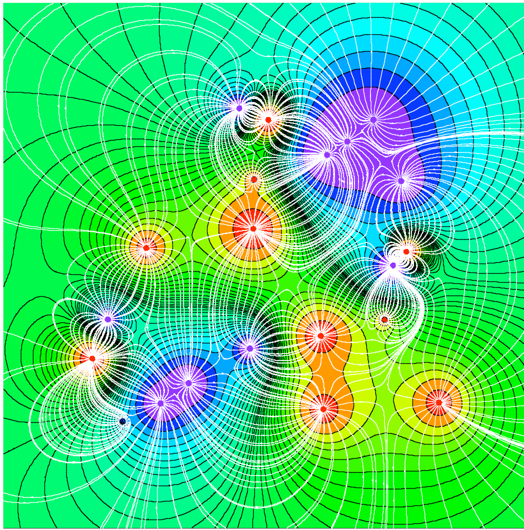
We just repeat the above calculations.

```
In[35]:= EField[{x_, y_}] = -{D[φ[{x, y}], x], D[φ[{x, y}], y]};
```

© 2004, 2005 Springer Science+Business Media, Inc.

In[36]:=
```
{fieldLines["+" -> "-"], fieldLines["-" -> "+"]} =
Apply[Function[{np, pn, T},
calculateFieldLine[{cx, cy}, #, {Ex, Ey}, {t, T},
       StoppingTest :> sTest[{cx[t], cy[t]}, np]]& /@
                                     startPoints[pn, n]],
{{negativeCharges, positiveCharges, +3},
 {positiveCharges, negativeCharges, -3}}, {1}];
```

In[37]:=
```
Show[{equiPotentialPicture,
Graphics[{Thickness[0.002], GrayLevel[1],
          Cases[{fieldLineGraphic /@ fieldLines["-" -> "+"],
                 fieldLineGraphic /@ fieldLines["+" -> "-"]},
                _Line, Infinity]}]},
     FrameTicks -> None, PlotRange -> {{-1/4, 5/4}, {-1/4, 5/4}},
     DisplayFunction -> $DisplayFunction]
```



Out[37]= - Graphics -

Now, let us deal with a slightly more complicated example: a set of charged straight wires in two dimensions. We obtain the potential by integration of the general formula along a parametrized line segment.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[38]:= Simplify[((# /. t -> 1) - (# /. t -> 0))&[
                   Integrate[1/Sqrt[(x - (p1x + t (p2x - p1x)))^2 +
                                    (y - (p1y + t (p2y - p1y)))^2], t]]]
```

Out[38]=
$$\left(-\text{Log}\left[\left(2\left(-\text{p1x}^2-\text{p1y}^2-\text{p2x}\,x+\text{p1x}\,(\text{p2x}+x)-\text{p2y}\,y+\text{p1y}\,(\text{p2y}+y)+\sqrt{\text{p1x}^2+\text{p1y}^2-2\,\text{p1x}\,\text{p2x}+\text{p2x}^2-2\,\text{p1y}\,\text{p2y}+\text{p2y}^2}\sqrt{\text{p1x}^2+\text{p1y}^2-2\,\text{p1x}\,x+x^2-2\,\text{p1y}\,y+y^2}\right)\right)\Big/\left(\sqrt{\text{p1x}^2+\text{p1y}^2-2\,\text{p1x}\,\text{p2x}+\text{p2x}^2-2\,\text{p1y}\,\text{p2y}+\text{p2y}^2}\right)\right]+\text{Log}\left[\left(2\left(\text{p2x}^2-\text{p1y}\,\text{p2y}+\text{p2y}^2-\text{p2x}\,x+\text{p1x}\,(-\text{p2x}+x)+\text{p1y}\,y-\text{p2y}\,y+\sqrt{\text{p1x}^2+\text{p1y}^2-2\,\text{p1x}\,\text{p2x}+\text{p2x}^2-2\,\text{p1y}\,\text{p2y}+\text{p2y}^2}\sqrt{\text{p2x}^2+\text{p2y}^2-2\,\text{p2x}\,x+x^2-2\,\text{p2y}\,y+y^2}\right)\right)\Big/\left(\sqrt{\text{p1x}^2+\text{p1y}^2-2\,\text{p1x}\,\text{p2x}+\text{p2x}^2-2\,\text{p1y}\,\text{p2y}+\text{p2y}^2}\right)\right]\right)\Big/\left(\sqrt{\text{p1x}^2+\text{p1y}^2-2\,\text{p1x}\,\text{p2x}+\text{p2x}^2-2\,\text{p1y}\,\text{p2y}+\text{p2y}^2}\right)$$

For a faster numerical evaluation, we implement a compiled form of the last result. The last result is simple enough that we can implement a procedural version of it “by hand”.

```
In[39]:= φFiniteStraightWire =
         Compile[{{xy, _Real, 1}, {p1, _Real, 1}, {p2, _Real, 1}},
         Module[{a, b, c, ax, ay, az, bx, by, bz, as, cs, abcs, α1, α2},
         {a, b, c} = {(xy - p1).(xy - p1), -2(p2 - p1).(xy - p1),
                      (p2 - p1).(p2 - p1)};
         {as, cs} = {Sqrt[a], Sqrt[c]}; abcs = Sqrt[a + b + c];
         {α1, α2} = {(2as + b/cs), (2abcs +(b + 2c)/cs)};
         (Log[α2] - Log[α1])/cs]];
```

The compilation was successful.

```
In[40]:= Union[Head /@ Flatten[φFiniteStraightWire[[4]]]]
Out[40]= {Integer}
```

We will take the following set of wires as our example configuration. It is a square surrounded by eight wires of opposite charge.

```
In[41]:= wires = N @
         {Partition[Table[{Cos[φ], Sin[φ]}/2, {φ, 0, 2Pi, 2Pi/4}], 2, 1],
          Partition[Table[{Cos[φ], Sin[φ]}  , {φ, 0, 2Pi, 2Pi/17}], 2]};
```

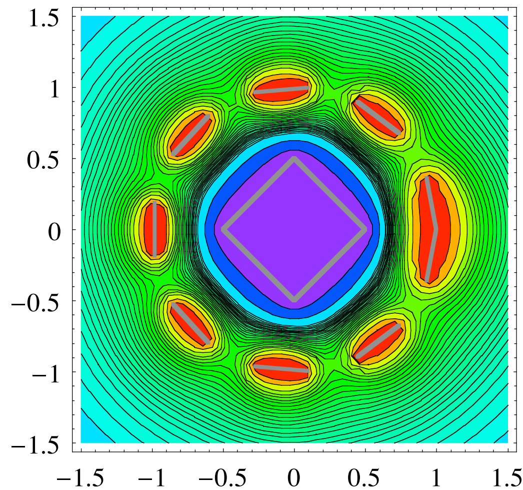
For the total potential φ, we obtain the following expression. We weight the contribution of all line segments according to their length.

```
In[42]:= {l1, l2} = (Plus @@ (norm /@ Apply[Subtract, #, {1}]))& /@ wires;

         φ[{x_, y_}] := Subtract @@ MapThread[
               (Plus @@ Apply[φFiniteStraightWire[{x, y}, ##]&,
                         wires[[#1]], {1}])/#2&, {{1, 2}, {l1, l2}}]
```

Here is a contour plot showing the distribution of the potential.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[45]:= equiPotentialPicture =
    Function[cp, ListContourPlot[cp[[1]],
              MeshRange -> {{-3/2, 3/2}, {-3/2, 3/2}},
              ColorFunction -> (Hue[# 0.75]&),
              Contours -> makeContours[cp, 30],
              ContourStyle -> {{Thickness[0.002]}},
              Epilog -> {GrayLevel[1/2], Thickness[0.01],
                         Map[Line, wires, {2}]}]][
     ContourPlot[φ[{x, y}], {x, -3/2, 3/2}, {y, -3/2, 3/2},
                 PlotPoints -> 50, Compiled -> False,
                 DisplayFunction -> Identity]]
```




```
Out[45]= - ContourGraphics -
```

For the electric field, we differentiate the above expression. For speed reasons, again, we use a compiled version.

```
In[46]:= EFieldFiniteStraightWire =
    Compile[{{xy, _Real, 1}, {p1, _Real, 1}, {p2, _Real, 1}},
    Module[{a, b, c, ax, ay, az, bx, by, bz, as, cs, abcs, α1, α2},
    {a, b, c} = {(xy - p1).(xy - p1), -2(p2 - p1).(xy - p1),
                 (p2 - p1).(p2 - p1)};
    {ax, ay, bx, by} = {{2, 0}.(xy - p1), {0, 2}.(xy - p1),
                    -(p2 - p1).{2, 0}, -(p2 - p1). {0, 2}};
    {as, cs, abcs} = {Sqrt[a], Sqrt[c], Sqrt[a + b + c]};
    {α1, α2} = {(2as + b/cs), (2abcs +(b + 2c)/cs)};
    {(-((ax/as + bx/cs)/α1) +
        ((ax + bx)/abcs + bx/cs)/α2)/cs,
     (-((ay/as + by/cs)/α1) + ((ay + by)/abcs + by/cs)/α2)/cs}]];
```

We get the electric field caused by all charged pieces by adding all of their contributions.

```
In[47]:= EField[{x_?NumberQ, y_?NumberQ}] := Subtract @@ MapThread[
    (Plus @@ Apply[EFieldFiniteStraightWire[{x, y}, ##]&,
         wires[[#1]], {1}])/#2&, {{1, 2}, {l1, l2}}]

In[48]:= makeFieldComponentDefinitions[{Ex, Ey}, EField, {x, y}, Real];
```

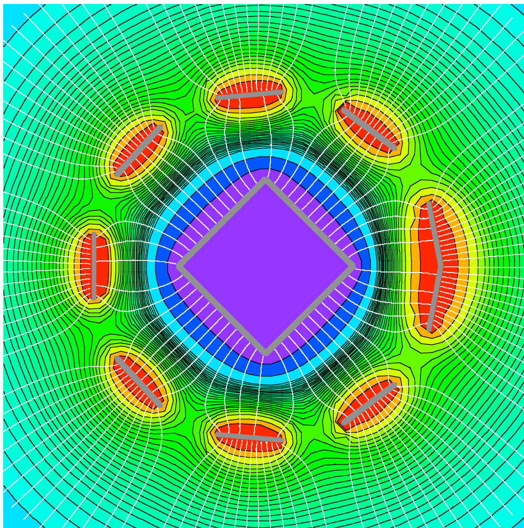
We integrate the field line differential equation starting from n points in distance ε to both sides of the lines.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[49]:= startPoints[{p1_, p2_}, n_] :=
         With[{normal = n[Reverse[p2 - p1]{1, -1}], ε = 0.02},
              Table[p1 + i/n(p2 - p1) + ε normal, {i, n}]]
```

Here is the resulting picture of the potential and of the field lines. We integrate the field lines until we reach the end of the integration domain or until the field strength becomes quite large (this means until we are very near a charge).

```
In[50]:= Show[{(* potential as background *)
               equiPotentialPicture,
               Graphics[{Thickness[0.002], GrayLevel[1],
                Cases[Function[ϵ, fieldLineGraphic /@
                (* calculate the field lines *)
         (calculateFieldLine[{cx, cy}, #, {Ex, Ey}, {t, (-1)^ϵ},
                             MaxSteps -> 1000,
                             PrecisionGoal -> 3, AccuracyGoal -> 3,
                 StoppingTest :> (* end integrating near to a charge *)
                 (#.#&[EField[{cx[t], cy[t]}]] > 10^6)]& /@
               Flatten[startPoints[#, 13]& /@ wires[[ϵ]], 1])] /@
                    {1, 2}, _Line, Infinity]}]},
                         PlotRange -> {{-3/2, 3/2}, {-3/2, 3/2}},
                         AspectRatio -> Automatic, FrameTicks -> None,
                         DisplayFunction -> $DisplayFunction]
```



```
Out[50]= - Graphics -
```

Now, let us calculate the electric field lines of a set of charged spheres in three dimensions. We start by generating a set of spheres of random diameter and at random positions. `makeCharges` yields a list of *pos* positive and *neg* negative spheres representing the charges. The `While` loops searches for a sphere configuration in which the individual spheres are nonoverlapping. We assume that the spheres are homogeneously charged (*charge* ~ *radius*^{3}).

© 2004, 2005 Springer Science+Business Media, Inc.

In[51]:=
```
makeCharges[{pos_, neg_}, {rMin_, rMax_, minDist_}] :=
Module[{p, n},
While[(* random values for charges and positions *)
      {p, n} = Table[
      (* charge *)  {Random[Real, {rMin, rMax}],
      (* position *)  Table[Random[Real, {-1, 1}], {3}]},
             {#}]& /@ {pos, neg};
      (* do the charged spheres intersect? *)
      Not[And @@ Flatten[
      Table[norm[#[[i, 2]] - #[[j, 2]]] > #[[i, 1]] + #[[j, 1]],
            {i, pos + neg}, {j, i - 1}]]]&[
      Join[p, n]], Null];
      (* charge ~ radius^3 *)
Join[{+#[[1]]^3, #[[2]]}& /@ p, {-#[[1]]^3, #[[2]]}& /@ n]]
```

In[52]:=
```
SeedRandom[1111111111111];
charges = makeCharges[{6, 5}, {0.04, 0.12, 0.3}];
```

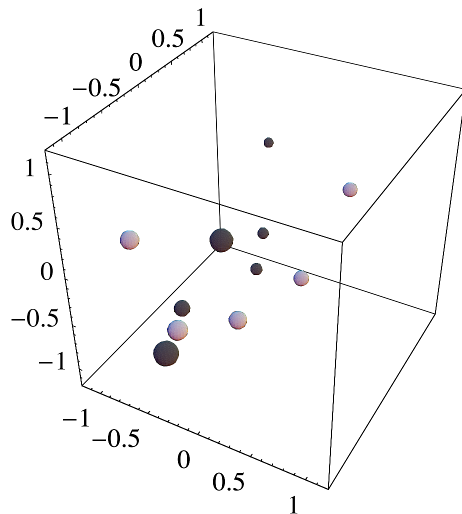
We color the positive and negative spheres differently.

In[54]:=
```
{color["⊕"], color["⊖"]} = SurfaceColor[GrayLevel[#]]& /@ {0.3, 0.9};
```

Here is the configuration of charges.

In[55]:=
```
chargePic :=
With[{(* a sphere *) sphere =
      Cases[ParametricPlot3D[
          {Cos[φ] Sin[ϑ], Sin[φ] Sin[ϑ], Cos[ϑ]},
          {φ, 0, 2Pi}, {ϑ, 0, Pi}, PlotPoints -> 20,
          DisplayFunction -> Identity], _Polygon, Infinity]},
(* color positive and negative charges *)
Graphics3D[{EdgeForm[],
 Function[c, {If[Sign[c[[1]]] < 0, color["⊖"], color["⊕"]],
 Map[(c[[2]] + Abs[c[[1]]]^(1/3) #)&, sphere, {-2}]}] /@ charges}]];
```

In[56]:=
```
pic = Show[chargePic, Axes -> True,
           PlotRange -> 1.2 {{-1, 1}, {-1, 1}, {-1, 1}}]
```



Out[56]= - Graphics3D -

© 2004, 2005 Springer Science+Business Media, Inc.

In the following example, we are only interested in the electric potential and the electric field outside of the spheres. In this case, the electric field coincides with the one from a point charge of the same charge at the position of the center of the sphere.

```
In[57]:= φ[{x_, y_, z_}] = Plus @@ (φCoulomb[#, {x, y, z}]& /@ charges);
```

```
In[58]:= EField[{x_, y_, z_}] = -{D[#, x], D[#, y], D[#, z]}&[φ[{x, y, z}]];
```

```
In[59]:= makeFieldComponentDefinitions[{Ex, Ey, Ez}, EField, {x, y, z}];
```

Similar to the 2D case, we calculate the field lines by solving their characteristic differential equation. We start the integration at a set of roughly uniformly distributed points just outside of the spheres.

```
In[60]:= {positiveCharges, negativeCharges} =
                Cases[charges, {_?#, _}]& /@ {Positive, Negative};
```

```
In[61]:= startPoints[charges_, {ppφ_, ppϑ_}, γ_] :=
         Flatten[Table[#[[2]] + γ Abs[#[[1]]]^(1/3)*
             {Cos[φ] Sin[ϑ], Sin[φ] Sin[ϑ], Cos[ϑ]},
                          {φ, 0, 2Pi(1 - 1/ppφ), 2Pi/ppφ},
            {ϑ, Pi/ppϑ, Pi(1 - 1/ppϑ), Pi/ppϑ}]& /@ charges, 2]
```

Now, we calculate the field lines. We end integrating the differential equations either when we are near the surface of another sphere or when the magnitude of the field strength becomes too large. The calculation of the field lines can be accomplished quickly

```
In[62]:= sTest[cs_, pn_] := (Min[(norm[cs -  #[[2]]] -
                      Abs[#[[1]]]^(1/3))& /@ pn] < 0 || (#.#&[cs] > 4))
```

```
In[63]:= (fieldLines["⊕" -> "⊖"] =
         calculateFieldLine[{cx, cy, cz}, #, {Ex, Ey, Ez}, {t, 2},
            StoppingTest :> sTest[{cx[t], cy[t], cz[t]}, negativeCharges]]& /@
         startPoints[positiveCharges, {5, 4}, 1.02];) // Timing

Out[63]= {4.386 Second, Null}
```

The function `fieldLineGraphic3D` uses `ParametricPlot3D` to generate a line approximation of the field lines.

```
In[64]:= fieldLineGraphic3D[cxcycz_] :=
         ParametricPlot3D[Evaluate[{cx[t], cy[t], cz[t]} /. cxcycz],
                          Evaluate[Flatten[{t, cxcycz[[1, 1, 2, 1, 1]]}]],
                          PlotPoints -> 300, DisplayFunction -> Identity]
```

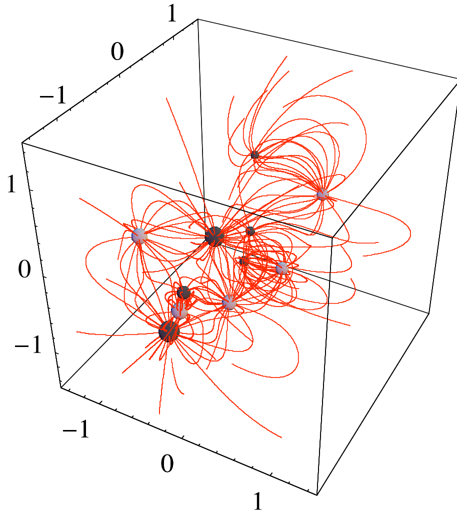
In an analogous manner, we calculate the field lines from the negative to the positive charges.

```
In[65]:= fieldLines["⊖" -> "⊕"] =
         calculateFieldLine[{cx, cy, cz}, #, {Ex, Ey, Ez}, {t, -2},
          StoppingTest :> sTest[{cx[t], cy[t], cz[t]}, positiveCharges]]& /@
         startPoints[negativeCharges, {5, 4}, 1.02];
```

Here is the picture of the charges and the field lines between them.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[66]:= Show[{pic,
            Graphics3D[{Hue[0], Thickness[0.002], Cases[
              fieldLineGraphic3D /@ #, _Line, Infinity]}]& /@
             {fieldLines["⊕" -> "⊖"], fieldLines["⊖" -> "⊕"]}},
          PlotRange -> 3/2 Table[{-1, 1}, {3}]]
```




```
Out[66]= - Graphics3D -
```

In the 2D case, the field lines gave a good visual representation of the field lines; this is not so much the case for the 3D picture. The main reason is that lines (head `Line`) are purely 1D objects and do not give any "depth feeling" [1984✱]. Thickening the lines to thin tubes gives a much better visualization. Let us do this. The function `makeTube` generates a tube (in the form of a list of polygons) along the line `Line[`*points*`]`, where *rFunction* determines its local radius. *startCrossSection* determines the initial cross section of the tube. Later, we will reuse these functions to graph wires.

```
In[67]:= makeTube[Line[points_], startCrossSection_, rFunction_] :=
         MapThread[Polygon[Join[#1, Reverse[#2]]]&, #1]& /@
         (Map[Partition[#, 2, 1]&, Partition[
         MapIndexed[(* change tube diameter *)
                     rFunction[#1, #2, Length[points]]&,
         FoldList[Function[{p, t},
         Module[{o = orthogonalDirections[t]},
                (* move along the line *)
                prolongate[#, t[[2]], t[[2]] - t[[1]], o]& /@ p]],
                startCrossSection,
                Rest[Partition[points, 3, 1]]]], 2, 1], {2}]);
```

The function `makeTube` works similarly to the one discussed in Subsection 2.3.1 of the Graphics volume [1807✱]. It prolongates a given cross section *startCrossSection* along a given set of points *points* (the set of points *points* can either form a smooth or a nonsmooth curve). The two functions `prolongate` and `orthogonalDirections` do most of the work involved. `prolongate` moves the cross section of the tube from one end of a line segment to the end of the next line segment, and `orthogonalDirections` creates a pair of orthogonal directions in the plane between two line segments.

```
In[68]:= prolongate[p_, q_, d_, {dirx_, diry_}] :=
         Module[{s, u, v}, First[p + s d /.
                Solve[Thread[(* line intersects plane *)
                             p + s d == q + u dirx + v diry], {s, u, v}]]];
```

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[69]:= orthogonalDirections[{p1_, p2_, p3_}] :=
    Module[{d},
           If[Abs[#1.#2] == 1, (* parallel case *)
              d = If[Abs[#1[[3]]] < 1, {-#1[[2]], #1[[1]], 0},
                     {0, #1[[3]], -#1[[2]]}],
              d = (#1 + #2)/2];
              n /@ {d, Cross[#1, d]}]&[n[p3 - p2], n[p1 - p2]];
```

We will change the diameter of the tubes in such a way that at the starting and ending points of the corresponding lines, they thin out. The function `ρFunc` determines the tube diameter along the line.

```
In[70]:= ρFunc[l_, {pos_}, n_] :=
    Function[mp, mp + Sin[(pos - 1)/(n - 1) Pi]^2 *
                (# - mp)& /@ l][(* cross-section center *)
                   (Plus @@ Rest[l])/(Length[l] - 1)]
```

The function `tubeGraphics3D` calculates the tubes.

```
In[71]:= tubeGraphics3D[line_Line, {r_:0.015, rFunction_:(#1&)},
                    color_, startCrossSection_:Automatic] :=
    Module[{dir, dir1, dir2, scs, ppSCS = 12},
    If[startCrossSection === Automatic,
       (* make start cross section *)
       dir = n[line[[1, 2]] - line[[1, 1]]];
       dir1 = n[# - # dir]&[Table[Random[], {3}]];
       dir2 = n[Cross[dir, dir1]];
       scs = Table[N[line[[1, 1]] + r (Cos[φ] dir1 + Sin[φ] dir2)],
                   {φ, 0, 2π, 2π/ppSCS}],
       scs = startCrossSection[[1]]];
    Graphics3D[{EdgeForm[], color, (* make tube *)
    makeTube[Line[Join[{line[[1, -2]]}, line[[1]],
                       {line[[1, 2]]}]], scs, rFunction]}]]
```

Using this function, we implement a function `fieldLine3D` that will convert an `InterpolatingFunction` representing the field line in a thin tube.

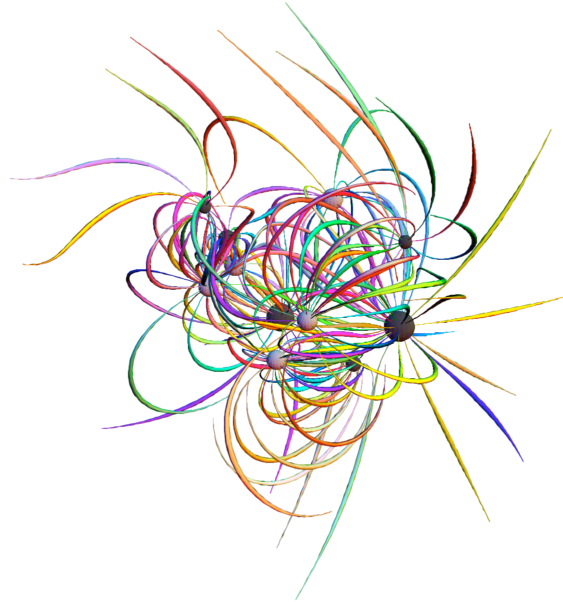
```
In[72]:= fieldLine3D[fieldLineIF_, color_] :=
    Module[{tMax, n = 60},
    tMax = DeleteCases[fieldLineIF[[1, 1, 2, 1, 1]], 0.][[1]];
    If[Abs[tMax] > 0,
       tubeGraphics3D[Line[Table[Evaluate[{cx[t], cy[t], cz[t]} /.
           fieldLineIF[[1]]], {t, 0, tMax, tMax/n}]],
                      {0.016, ρFunc}, color], {}]]

In[73]:= Length[Join[fieldLines["⊕" -> "⊖"], fieldLines["⊖" -> "⊕"]]]

Out[73]= 165
```

Now, we have everything together to generate a "real" 3D picture of the field lines. To enhance the appearance, we color each of the tubes differently. Now, we get a "3D" feeling for the field lines. We have, of course, to pay a price for this improved picture—it requires much more memory than does the line picture from above.

```
In[74]:= fieldLineColor :=
         SurfaceColor[Hue[Random[]], Hue[Random[]], 3 Random[]];
```

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[75]:= Show[{chargePic, fieldLine3D[#, fieldLineColor]& /@
              Join[fieldLines["⊕" -> "⊖"], fieldLines["⊖" -> "⊕"]]},
            PlotRange -> All, Boxed -> False, ViewPoint -> {1, 1, 1}]
```



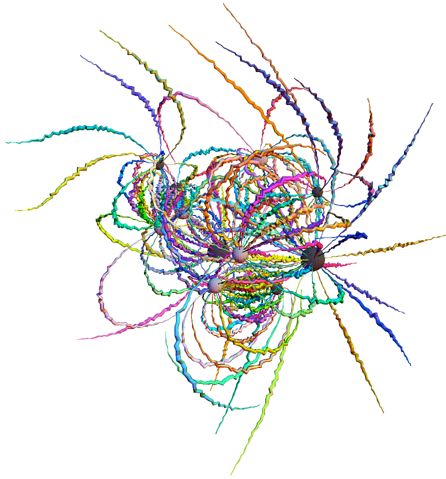
```
Out[75]= - Graphics3D -
```

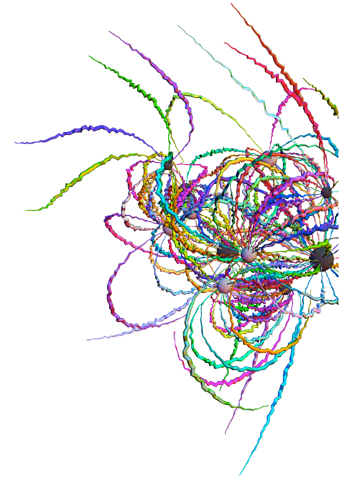
We could now make the last graphic more interesting graphically (but less physically realistic) by adding random translations along each of the field line tubes. To make such a graphic, we have to change the function ρFunc from above. In addition to forming a tube, we shift the centers of the tube cross sections randomly in the plane perpendicular to the field line.

```
In[76]:= (* make a zig-zag filed line tube *)
         ρFuncZigZag[l_, {pos_}, n_] := Function[{mp, shift},
         (mp + shift + Sin[(pos - 1)/(n - 1) Pi]^2 (# - mp))& /@ l][
                  (* center and random shift of tube cross section *)
                  Module[{R = 0.012, mp, dir1, dir2, φR, rR, shift},
                         mp = (Plus @@ Rest[l])/(Length[l] - 1);
                         dir1 = n[l[[1]] - mp]; dir2 = n[l[[2]] - mp];
                         dir2 = n[dir2 - dir2.dir1 dir1];
                         φR = 2Pi Random[]; rR = 2 R Random[];
                         shift = Sin[(pos - 1)/(n - 1) Pi] *
                                 rR (Cos[φR] dir1 + Sin[φR] dir2);
                         Sequence @@ {mp, shift}]]
```

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[78]:= SeedRandom[123];
         Block[{ρFunc = ρFuncZigZag},
               (* show two realizations of the randomized field lines *)
               Show[GraphicsArray[Table[
                Show[{chargePic, fieldLine3D[#, fieldLineColor]& /@
                       Take[Join[fieldLines["⊕" -> "⊖"], fieldLines["⊖" -> "⊕"]], All]
                     PlotRange -> All, Boxed -> False, ViewPoint -> {1, 1, 1},
                     DisplayFunction -> Identity], {2}]]]]
```





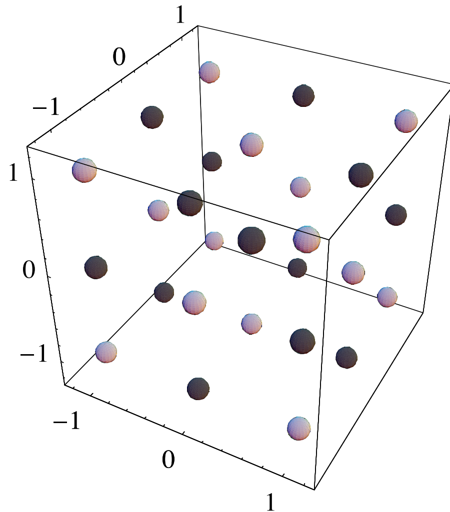
```
Out[79]= - GraphicsArray -
```

Now, let us calculate the electric field lines of a regular array of charged spheres. We will position the charges at the lattice points of a cubic lattice. To achieve charge neutrality of the whole cluster of $3 \times 3 \times 3$ charges, the central charge will be made twice as large as the other ones.

```
In[80]:= {color["⊕"], color["⊖"]} = SurfaceColor[GrayLevel[#]]& /@ {0.3, 0.9};

In[81]:= charges = Flatten[Table[{0.001 (-1)^(i + j + k), {i, j, k}},
                                 {i, -1, 1}, {j, -1, 1}, {k, -1, 1}], 2];

In[82]:= charges =  charges /. {f_, {0, 0, 0}} :> {2 f, {0, 0, 0}};

In[83]:= {positiveCharges, negativeCharges} =
                Cases[charges, {_?#, _}]& /@ {Positive, Negative};
```

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[84]:= pic = Show[chargePic, Axes -> True,
                    PlotRange -> 1.3 {{-1, 1}, {-1, 1}, {-1, 1}}]
```



```
Out[84]= - Graphics3D -
```

We just repeat the calculations from above for the current charge distribution.

```
In[85]:= φ[{x_, y_, z_}] = Plus @@ (φCoulomb[#, {x, y, z}]& /@ charges);

In[86]:= EField[{x_, y_, z_}] =  -{D[#, x], D[#, y], D[#, z]}&[φ[{x, y, z}]];

In[87]:= makeFieldComponentDefinitions[{Ex, Ey, Ez}, EField, {x, y, z}];
```

We will use another distribution for the starting points of the field lines around the charged spheres. The field lines will start at the vertices of a cube around the charges.

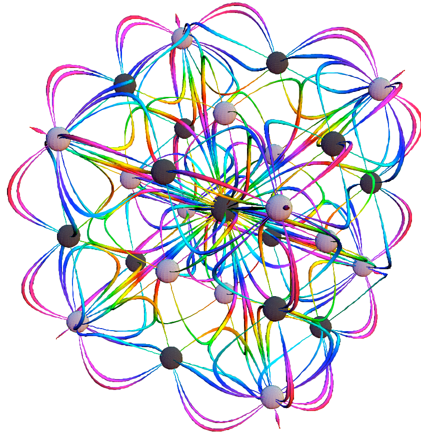
```
In[88]:= startPoints[charges_, γ_] :=
         (dirList = {{-1, -1, -1}, {-1, -1, 1}, {-1, 1, -1}, {-1, 1, 1},
                     {1, -1, -1}, {1, -1, 1}, {1, 1, -1}, {1, 1, 1}}/Sqrt[3.];
          Flatten[Table[charges[[j, 2]] +
                  γ Abs[charges[[j, 1]]]^(1/3) dirList[[k]],
                      {j, Length[charges]}, {k, Length[dirList]}], 1])

In[89]:= {fieldLines["⊕" -> "⊖"], fieldLines["⊖" -> "⊕"]} =
         Apply[Function[{np, pn, T},
          calculateFieldLine[{cx, cy, cz}, #, {Ex, Ey, Ez}, {t, T},
            StoppingTest :> sTest[{cx[t], cy[t], cz[t]}, np]]& /@
                                           startPoints[pn, 1.02]],
         {{negativeCharges, positiveCharges, +2},
          {positiveCharges, negativeCharges, -2}}, {1}];
```

We will also use another coloring for the field lines. Instead of coloring each field line with one color, we will color the field lines locally according to their distance from the origin. (Because this assigns a color to each single polygon, the following picture is more memory-consuming than the above one.)

```
In[90]:= fieldLineColor := {};

         colorFieldLine[gr3d_] := gr3d /. p_Polygon :>
          ({SurfaceColor[Hue[#], Hue[#], 2], p}&[norm[Plus @@ p[[1]]/4]])
```

© 2004, 2005 Springer Science+Business Media, Inc.

In[92]:=
```
Show[{pic, colorFieldLine[fieldLine3D[#, {}]]& /@
          Join[fieldLines["⊕" -> "⊖"], fieldLines["⊖" -> "⊕"]]},
        PlotRange -> All, Boxed -> False, Axes -> False]
```



Out[92]= - Graphics3D -

In[93]:=
```
(* to save memory *)
Clear[fieldLines];
```

Here ends the electric field line case—let us now deal with the more interesting case of magnetic field lines. Compared with the electric case, it is often much more difficult to intuitively "sketch" the behavior of magnetic field lines [1453✱]. Roughly speaking, the reason for the more predictable behavior of the electric field lines is the existence of the electrostatic potential φ. The set of differential equations for magnetic field lines can be rewritten as a time-dependent Hamiltonian system [1462✱], [902✱], [1591✱], [1793✱], [355✱], and as such, the field lines can exhibit chaotic behavior (we will see such examples later).

Given the (time-independent) current density $\vec{j}(\vec{y})$, the magnetic field $\vec{B}(\vec{x})$ is given by the Biot–Savart formula [344✱] (also applicable to electrostatics [1385✱]):

$$\vec{B}(\vec{x}) = \int_{\mathbb{R}^3} \frac{\vec{j}(\vec{y}) \times (\vec{x} - \vec{y})}{|\vec{x} - \vec{y}|^3} \, d\vec{y}.$$

For the simplest case of a 1D wire $\vec{\gamma}$ with parametric representation $\vec{\gamma}(t)$, the current is given by $\vec{j}(\vec{x}) = \int_{-\infty}^{\infty} \delta(\vec{x} - \vec{\gamma}(s))\, \vec{\gamma}'(s)\, ds$ (where s is the curve length parameter) and the above reduces to

$$\vec{B}(\vec{x}) = \int_{-\infty}^{\infty} \frac{\vec{\gamma}'(s) \times (\vec{x} - \vec{\gamma}(s))}{|\vec{x} - \vec{\gamma}(s)|^3} \, ds = \int_{-\infty}^{\infty} \frac{\vec{\gamma}'(t) \times (\vec{x} - \vec{\gamma}(t))}{|\vec{x} - \vec{\gamma}(t)|^3} \, dt.$$

We start with the simplest possible configuration—a straight, infinitely long, and infinitely thin current carrying wire. Here is a parametrization of the wire.

In[95]:=
```
infiniteWire[t_] = {0, 0, t};
```

Using the Biot–Savart law, we obtain the well-known magnetic field $\vec{B}(\vec{x}) \sim \{-y, x, 0\} / (x^2 + y^2)$.

© 2004, 2005 Springer Science+Business Media, Inc.

In[96]:=
```
Integrate[Cross[D[infiniteWire[t], t], {x, y, z} - infiniteWire[t]]/
                  norm[{x, y, z} - infiniteWire[t]]^3,
            {t, -Infinity, Infinity}, GenerateConditions -> False]
```

Out[96]= $\left\{-\frac{2\,y}{x^2+y^2},\ \frac{2\,x}{x^2+y^2},\ 0\right\}$

In[97]:=
```
BInfiniteWire[{x_, y_}] = 2 {-y, x}/(x^2 + y^2);
```

Using the last result, we define the components of the magnetic field.

In[98]:=
```
makeFieldComponentDefinitions[{Bx, By}, BInfiniteWire, {x, y}];
```

In the following example, we will display all current carrying wires as thin tubes with a yellow–orange color.

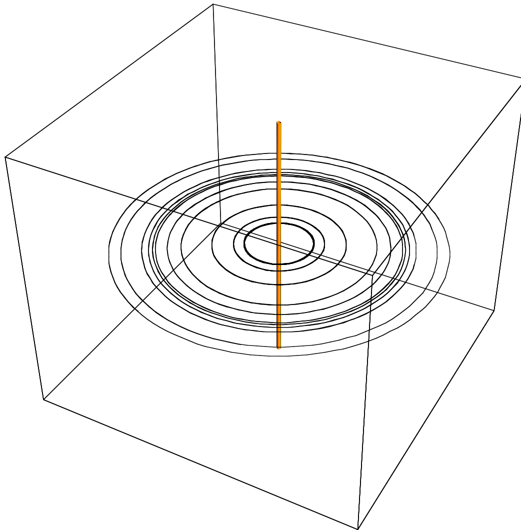
In[99]:=
```
currentColor = SurfaceColor[Hue[0.12], Hue[0.22], 2.12];
```

For 12 randomly selected initial points, we calculate the field lines.

In[100]:=
```
tMax = 12;
fieldLines =
calculateFieldLine[{cx, cy}, #, {Bx, By}, {t, tMax},
           MaxSteps -> 10^4]& /@ Table[Random[], {12}, {2}];
```

As expected, the field lines are concentric circles around the infinite wire. (For particle trajectories in the magnetic field of a long straight current-carrying wire, see [1349✱].)

In[102]:=
```
Show[{tubeGraphics3D[Line[{{0, 0, -1}, {0, 0, 0}, {0, 0, 1}}],
                     {}, currentColor],
ParametricPlot3D[Evaluate[{cx[t], cy[t], 0,
           {Thickness[0.002]}} /. #], {t, 0, tMax},
           PlotPoints -> 200, PlotRange -> All,
           DisplayFunction -> Identity]& /@ fieldLines},
           DisplayFunction -> $DisplayFunction]
```



Out[102]= - Graphics3D -

Instead of one infinite straight current, let us now deal with many of them. We choose 15 parallel currents at random positions and of random strengths. This system is translation-invariant along the wires. So, we restrict ourselves to the plane $z = 0$.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[103]:= SeedRandom[83569034823]
          BField[{x_, y_}] = Plus @@
          ((Random[Real, {-1, 1}] BInfiniteWire[{x, y} - #])& /@
            (mps = Table[{Random[Real], Random[Real]}, {15}]));
```

For a faster numerical calculation, we compile the last expression.

```
In[105]:= BFieldC = Compile[{x, y}, Evaluate[BField[{x, y}]]];
```

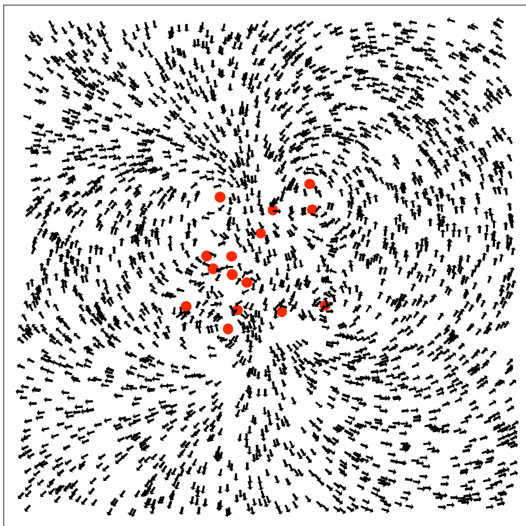
```
In[106]:= BField[{x_, y_}] := BFieldC[x, y]
```

To get an impression about the magnetic field of the 15 currents, we visualize it with little arrows pointing in the direction of the field.

```
In[107]:= (* a 2D arrow *)
          arrow2D[{p1_, d_}, l_] :=
          Module[{α = 0.14, β = 0.7, γ = 0.35, d1, d2},
                  d1 = n[d]; d2 = Reverse[d1]{-1, 1};
             Polygon[{p1 + α l d2, p1 + α l d2 + β l d1,
                     p1 + γ l d2 + β l d1, p1 + l d1,
                     p1 - γ l d2 + β l d1, p1 - α l d2 + β l d1, p1 - α l d2}]]
```

The red points are the positions of the currents (which flow perpendicular to the graphics plane).

```
In[109]:= fieldPic = Module[{point},
          Show[Graphics[{{PointSize[0.02], Hue[0], Point /@ mps},
              {Thickness[0.002],
          Table[arrow2D[{#, BFieldC @@ #}&[
                {Random[Real, {-1, 2}], Random[Real, {-1, 2}]}], 0.05],
                {2500}]}}],
          DisplayFunction -> $DisplayFunction, PlotRange -> All,
          AspectRatio -> Automatic, Frame -> True, FrameTicks -> None]]
```



```
Out[109]= - Graphics -
```

Now, we define the components of the magnetic field and calculate 100 field lines.

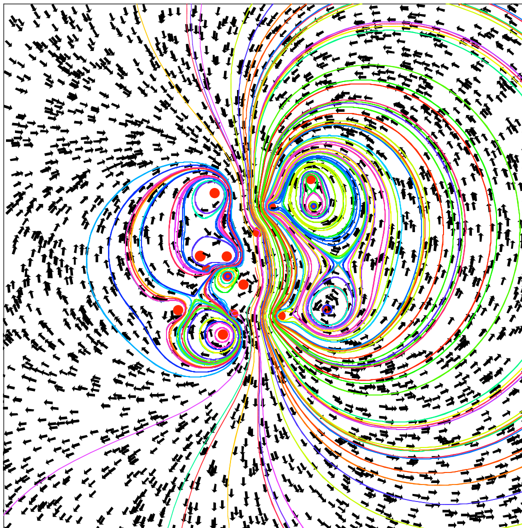
```
In[110]:= makeFieldComponentDefinitions[{Bx, By}, BField, {x, y}, Real];
```

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[111]:= tMax = 20;
          (fieldLines =
           Table[calculateFieldLine[{cx, cy},
                     {Random[], Random[]}, {Bx, By}, {t, tMax},
                      MaxSteps -> 10^4, Compiled -> False,
                      AccuracyGoal -> 6, PrecisionGoal -> 6], {100}];) // Timing
Out[112]= {43.954 Second, Null}
```

All field lines in the arrangement of parallel currents are closed curves.

```
In[113]:= Show[{fieldPic,
               ParametricPlot[Evaluate[{cx[t], cy[t]} /. fieldLines],
                              {t, 0, tMax}, PlotPoints -> 200,
                              PlotStyle ->
                               Table[{Thickness[0.002], Hue[Random[]]},
                                     {Length[fieldLines]}],
                              Compiled -> False, DisplayFunction -> Identity]},
                    PlotRange -> {{-1, 2}, {-1, 2}}]
```



```
Out[113]= - Graphics -
```

After an infinite straight line, the next simplest current is described by a circle. Although the integrations involved can be carried out in closed form using elliptic integrals, here we will use purely numerical techniques—namely, `NIntegrate` to calculate the magnetic field. `cfx`, `cfy`, and `cfz` are the compiled versions of the integrands for the field components.

```
In[114]:= {cfx, cfy, cfz} =
          Compile[{t, x, y, z},
                 Module[{c = Cos[t], s = Sin[t]},
                        #/(z^2 + (x - c)^2 + (y - s)^2)^(3/2)]]& /@
                  {z c, z s, (c (c - x) + s (s - y)^2)};
```

In the solution of the differential equations of the field lines, we need the components of the field at the same point. We calculate all components at once and store the already-calculated field values via a `SetDelayed[Set[…]]` construction.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[115]:= (* avoid automatic switching to high-precision numbers *)
          Developer`SetSystemOptions["CatchMachineUnderflow" -> False];

          (* avoid evaluation of arguments of NIntegrate *)
          Developer`SetSystemOptions["EvaluateNumericalFunctionArgument" -> False]

Out[118]= EvaluateNumericalFunctionArgument → False
```

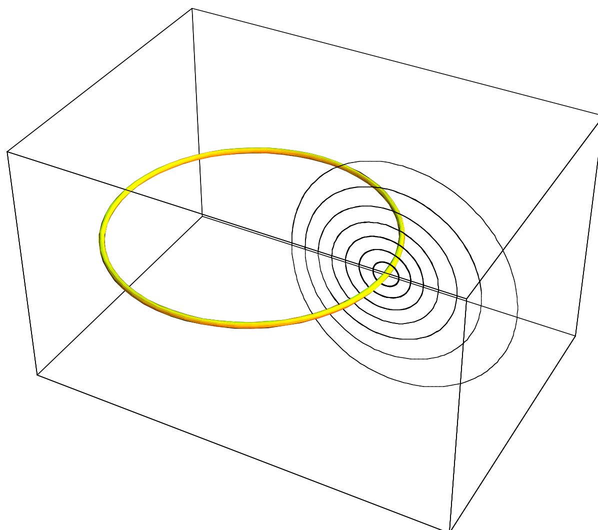
```
In[119]:=
          Clear[BField];

          BField[{x_, y_, z_}] := BField[{x, y, z}] =
          NIntegrate[{cfx[t, x, y, z], cfy[t, x, y, z], cfz[t, x, y, z]},
                     {t, 0, 2 Pi}, Compiled -> False,
                     AccuracyGoal -> 3, PrecisionGoal -> 6]
```

Proceeding as above, we make the field component assignments and calculate some field lines. Because of the obvious rotational invariance of the magnetic field, we calculate all field lines in the x,z-plane.

```
In[121]:= makeFieldComponentDefinitions[{Bx, By, Bz}, BField, {x, y, z}, Real];

In[122]:= tMax = 6;
          fieldLines = calculateFieldLine[{cx, cy, cz}, #, {Bx, By, Bz}, {t, tMax},
                                          MaxSteps -> 10^5]& /@
                                    Table[{x, 0., 0.}, {x, 0.3, 0.9, 0.1}];
```

As is well known, the field lines form concentric closed curves around the wire.

```
In[124]:= Show[{tubeGraphics3D[Line[
           Table[{Cos[φ], Sin[φ], 0}, {φ, 0, 2Pi, 2Pi/60}]], {0.02, #1&},
                                  currentColor],
          ParametricPlot3D[Evaluate[{cx[t], cy[t], cz[t],
                     {Thickness[0.002]}} /. #], {t, 0, tMax},
                     PlotPoints -> 200,
                     DisplayFunction -> Identity]& /@ fieldLines},
                     DisplayFunction -> $DisplayFunction, PlotRange -> All]
```



```
Out[124]= - Graphics3D -
```

The next example is the combination of an infinite straight current and a circular current in the plane perpendicular to the infinite wire. Instead, using an exact circle, we will approximate the circle with straight pieces. This has two advantages:

© 2004, 2005 Springer Science+Business Media, Inc.

First, it allows for a faster numerical calculation of the magnetic field; and second, using straight line segments allows us to model arbitrary current configurations. The qualitative behavior of the field is the same for the exact and the approximated circles.

```
In[125]:= lineSegment[t_, {p1_, p2_}] = p1 + t(p2 - p1);

In[126]:= BFiniteWire[{x_, y_, z_}, {{p1x_, p1y_, p1z_}, {p2x_, p2y_, p2z_}}] =
          With[{l = lineSegment[t, {{p1x, p1y, p1z}, {p2x, p2y, p2z}}]},
          Integrate[Cross[D[l, t], {x, y, z} - l]/
                    norm[{x, y, z} - l]^3, {t, 0, 1}, GenerateConditions -> False]];
```

The expression for `BFiniteWire` is relatively large.

```
In[127]:= BFiniteWire[{x, y, z}, {{p1x, p1y, p1z}, {p2x, p2y, p2z}}] // LeafCount

Out[127]= 1519
```

Using the closed-form formula in `BFiniteWire` directly is relatively slow. Calculating 10000 field values takes a few seconds on a 2 GHz computer.

```
In[128]:= Timing[Do[BFiniteWire[Table[Random[], {3}],
                                Table[Random[], {2}, {3}]], {10000}]]

Out[128]= {6.799 Second, Null}
```

To speed up the calculation carried out by `BFiniteWire`, we simplify the expression [743✱], use the package `NumericalMath`OptimizeExpression`` to rewrite the expression in a more effective way, and compile the resulting expression. Using this package avoids the time-consuming and cumbersome work of generating a procedural version of `BFiniteWire` “by hand”.

```
In[129]:= BFiniteWireC = (* use flat argument structure *)
               Compile[{x, y, z, p1x, p1y, p1z, p2x, p2y, p2z},
                 (* use function from Experimental` context *)
                 Evaluate[Experimental`OptimizeExpression[
                  Simplify[BFiniteWire[{x, y, z},
                           {{p1x, p1y, p1z}, {p2x, p2y, p2z}}]],
                  OptimizationLevel -> 1, ExcludedForms -> {}]]];
```

`BFiniteWireC` is about 20 times faster than `BFiniteWire`.

```
In[130]:= Timing[Do[BFiniteWireC @@ Table[Random[], {9}], {10000}]]

Out[130]= {0.471 Second, Null}
```

Both functions calculate the same values. We check it for a set of “random” values.

```
In[131]:= {BFiniteWire[{#1, #2, #3}, {{#4, #5, #6}, {#7, #8, #9}}],
           BFiniteWireC[##]}& @@ Table[N[Sin[k]], {k, 9}]

Out[131]= {{-1.82111, 1.9679, -1.82111}, {-1.82111, 1.9679, -1.82111}}
```

This is the wire approximating the circular current.

```
In[132]:= wire = N[Table[{Cos[φ], Sin[φ], 0}, {φ, 0, 2Pi, 2Pi/12}]];
```

To calculate the magnetic field of all wire pieces, we just sum the contributions from the 12 finite pieces and the infinite wire.

```
In[133]:= wireData = Flatten /@ Partition[wire, 2, 1];

In[134]:= BField[{x_, y_, z_}] := 2 {-y, x, 0.}/(x^2 + y^2) +
              (Plus @@ Apply[BFiniteWireC[x, y, z, ##]&, wireData, {1}]);
```

© 2004, 2005 Springer Science+Business Media, Inc.

These are the definitions for the three components of the magnetic field.

```
In[135]:= makeFieldComponentDefinitions[{Bx, By, Bz}, BField, {x, y, z}, Real];
```

```
In[136]:= SeedRandom[123454321];
          startPoints = Table[Random[Real, {-2, 2}], {9}, {3}];
```

```
In[138]:= tMax = 100;
          Timing[fieldLines = calculateFieldLine[{cx, cy, cz}, #, {Bx, By, Bz},
                                              {t, 0, tMax}, MaxSteps -> 10^5,
                                        PrecisionGoal -> 12]& /@  startPoints];
```

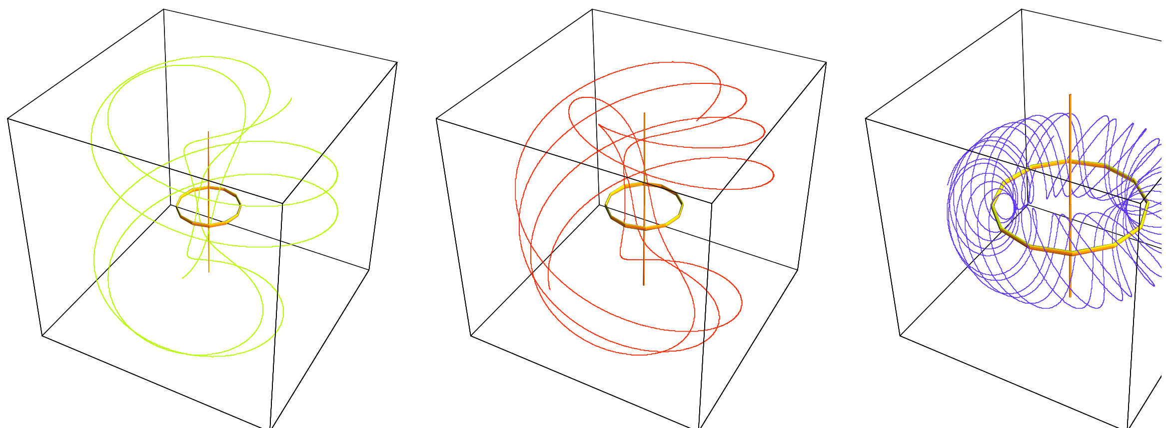
Because some of the following pictures of magnetic field lines might look unexpected, we should check for the quality of the result of `NDSolve`. We do this by running the just-calculated field lines backward. We could also change the `Method` option setting to `NDSolve` to make sure we carry out two independent calculations.

```
In[140]:= Norm /@ (startPoints - (({cx[0], cy[0], cz[0]} /.
           calculateFieldLine[{cx, cy, cz},
                 {cx[#], cy[#], cz[#]}&[tMax] /. #[[1]],
                 {Bx, By, Bz}, {t, tMax, 0},
                  MaxSteps -> 10^5, StartingStepSize -> 10^-2,
                  PrecisionGoal -> 12][[1]])& /@ fieldLines))
```

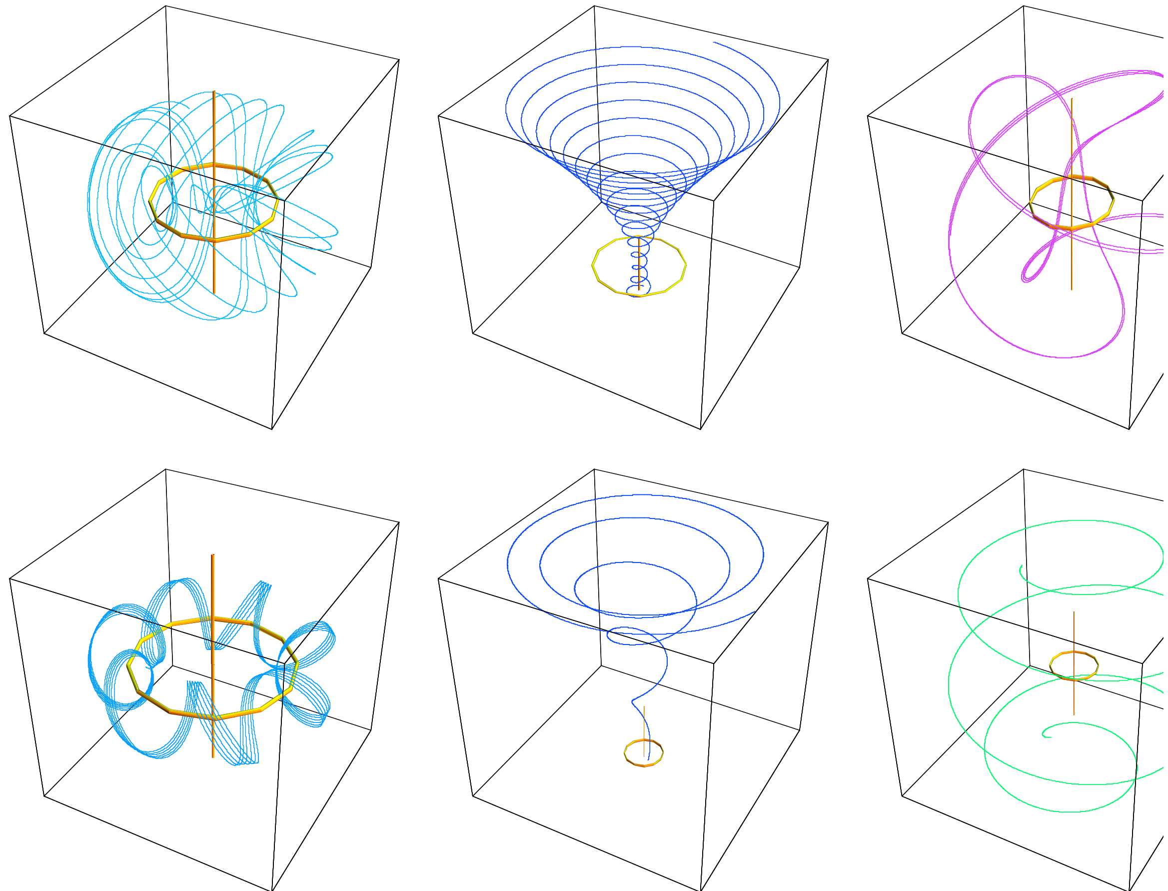
Out[140]= $\{6.2266\times10^{-6}$, 0.0000389089, 0.000109531, 0.000169618, 0.0000301555, 4.83168×10^{-6}, 0.0000402704, 1.0416×10^{-6}, 0.0000122784$\}$

As the last result shows, the differential equations were correctly solved within the resolution of the graphic shown.

```
In[141]:= With[{gr1 = tubeGraphics3D[Line[{{0, 0, -1}, {0, 0, 0}, {0, 0, 1}}],
                                      {}, currentColor],
                gr2 = tubeGraphics3D[Line[wire], {0.02, #1&}, currentColor]},
          Show[GraphicsArray[#]]& /@ Partition[
          Show[{gr1, gr2, ParametricPlot3D[
            Evaluate[{cx[t], cy[t], cz[t],
               {Thickness[0.002], Hue[Random[]]}} /. #], {t, 0, tMax},
               PlotPoints -> 10000, DisplayFunction -> Identity]},
                     DisplayFunction -> Identity, BoxRatios -> {1, 1, 1}]& /@
                                                                 fieldLines, 3]]
```



© 2004, 2005 Springer Science+Business Media, Inc.



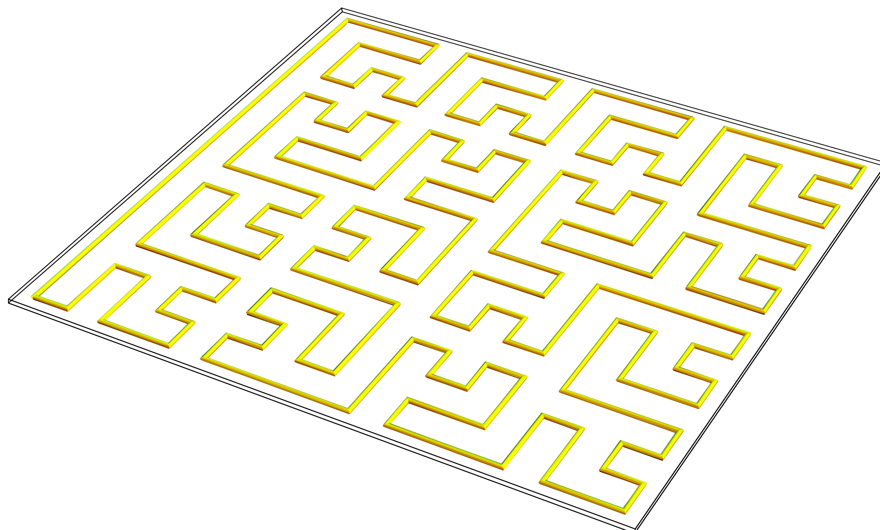
```
Out[141]= {- GraphicsArray -, - GraphicsArray -, - GraphicsArray -}
```

Now let us look at the field lines of a much more complicated wire—the wire will be shaped like a 2D Peano curve. Here are the points of this wire.

```
In[142]:= wire = Append[#, 0]& /@
          {{0, 0}, {1, 0}, {1, 2}, {2, 2}, {2, 0}, {4, 0}, {4, 1}, {3, 1},
           {3, 2}, {4, 2}, {4, 3}, {1, 3}, {1, 6}, {2, 6}, {2, 4}, {4, 4},
           {4, 5}, {3, 5}, {3, 6}, {4, 6}, {4, 7}, {1, 7}, {1, 11}, {2, 11},
           {2, 10}, {3, 10}, {3, 11}, {4, 11}, {4, 9}, {2, 9}, {2, 8},
           {5, 8}, {5, 11}, {6, 11}, {6, 10}, {7, 10}, {7, 11}, {8, 11},
           {8, 9}, {6, 9}, {6, 8}, {8, 8}, {8, 5}, {7, 5}, {7, 7}, {5, 7},
           {5, 6}, {6, 6}, {6, 5}, {5, 5}, {5, 4}, {8, 4}, {8, 1}, {7, 1},
           {7, 3}, {5, 3}, {5, 2}, {6, 2}, {6, 1}, {5, 1}, {5, 0}, {9, 0},
           {9, 3}, {10, 3}, {10, 2}, {11, 2}, {11, 3}, {12, 3}, {12, 1},
           {10, 1}, {10, 0}, {13, 0}, {13, 2}, {14, 2}, {14, 0}, {16, 0},
           {16, 1}, {15, 1}, {15, 2}, {16, 2}, {16, 3}, {13, 3}, {13, 6},
           {14, 6}, {14, 4}, {16, 4}, {16, 5}, {15, 5}, {15, 6}, {16, 6},
           {16, 7}, {12, 7}, {12, 4}, {11, 4}, {11, 5}, {10, 5}, {10, 4},
           {9, 4}, {9, 6}, {11, 6}, {11, 7}, {9, 7}, {9, 11}, {10, 11},
           {10, 10}, {11, 10}, {11, 11}, {12, 11}, {12, 9}, {10, 9},
           {10, 8}, {13, 8}, {13, 10}, {14, 10}, {14, 8}, {16, 8}, {16, 9},
           {15, 9}, {15, 10}, {16, 10}, {16, 11}, {13, 11}, {13, 14},
           {14, 14}, {14, 12}, {16, 12}, {16, 13}, {15, 13}, {15, 14},
           {16, 14}, {16, 15}, {12, 15}, {12, 12}, {11, 12}, {11, 13},
           {10, 13}, {10, 12}, {9, 12}, {9, 14}, {11, 14}, {11, 15},
           {8, 15}, {8, 12}, {7, 12}, {7, 13}, {6, 13}, {6, 12},
           {5, 12}, {5, 14}, {7, 14}, {7, 15}, {4, 15}, {4, 12},
           {3, 12}, {3, 13}, {2, 13}, {2, 12}, {1, 12}, {1, 14},
           {3, 14}, {3, 15}, {0, 15}, {0, 0}};
```

© 2004, 2005 Springer Science+Business Media, Inc.

In[143]:=
```
Show[tubeGraphics3D[Line[wire], {0.06, #1&}, currentColor]]
```



Out[143]= - Graphics3D -

In[144]:=
```
wireData =  Flatten /@ Partition[N[wire], 2, 1];
```

We define the magnetic field generated by this wire. We add the contributions from all wire segments using the function `BFiniteWireC`.

In[145]:=
```
BField[{x_, y_, z_}] := (Plus @@ Apply[BFiniteWireC[x, y, z, ##]&,
                                        wireData, {1}]);
```

In[146]:=
```
makeFieldComponentDefinitions[{Bx, By, Bz}, BField, {x, y, z}, Real];
```

We use four selected start points and calculate the field lines over a long "time".

In[147]:=
```
startPoints =
{{15.68110, 6.45551, 0.87960}, {11.85503, 0.72703, 0.87980},
 {10.72057, 12.10141, 0.70727}, {9.94457, 3.24733, 0.58636}};
```

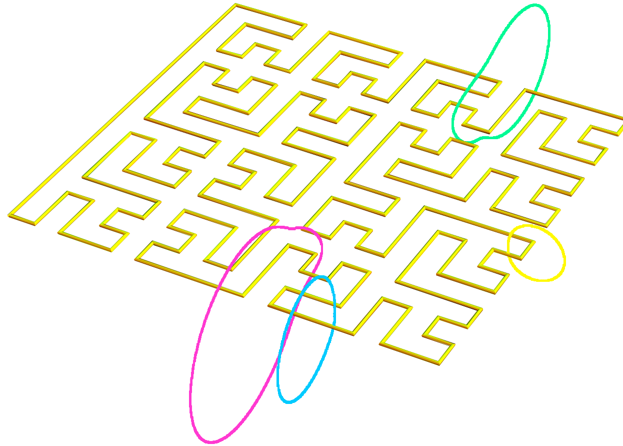
In[148]:=
```
tMax = 100;
Timing[fieldLines =
       calculateFieldLine[{cx, cy, cz}, #, {Bx, By, Bz},
        {t, 0, tMax}, MaxSteps -> 10^4,
             PrecisionGoal -> 6]& /@  startPoints;]
```

Out[149]= {153.21 Second, Null}

Despite the complicated shape of the current-carrying wire, the resulting field lines are closed curves, winding just once around.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[150]:= With[{gr =  tubeGraphics3D[Line[wire], {0.06, #1&}, currentColor]},
     Show[{gr, ParametricPlot3D[
       Evaluate[{cx[t], cy[t], cz[t],
          {Thickness[0.003], Hue[Random[]]}} /. #], {t, 0, tMax},
          PlotPoints -> 10000, DisplayFunction -> Identity]& /@ fieldLines},
                DisplayFunction -> $DisplayFunction,
                PlotRange -> All, Boxed -> False]]
```

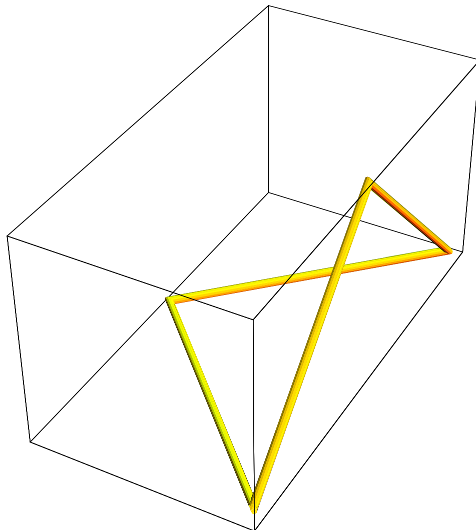


```
Out[150]= - Graphics3D -
```

The field lines for every planar set of currents have the topological shape, such as the three field lines in the last picture.

Now, let us calculate and visualize the magnetic field lines in the simplest nonplanar case—a wire made from four finite straight line segments.

```
In[151]:= wire = {{0, -1, 0}, {-1, 0, 0}, {0, 1, 0}, {0, 0, 1}, {0, -1, 0}};

In[152]:= Show[tubeGraphics3D[Line[wire], {0.02, #1&}, currentColor]]
```



```
Out[152]= - Graphics3D -
```

Proceeding as above, we define the field components.

© 2004, 2005 Springer Science+Business Media, Inc.

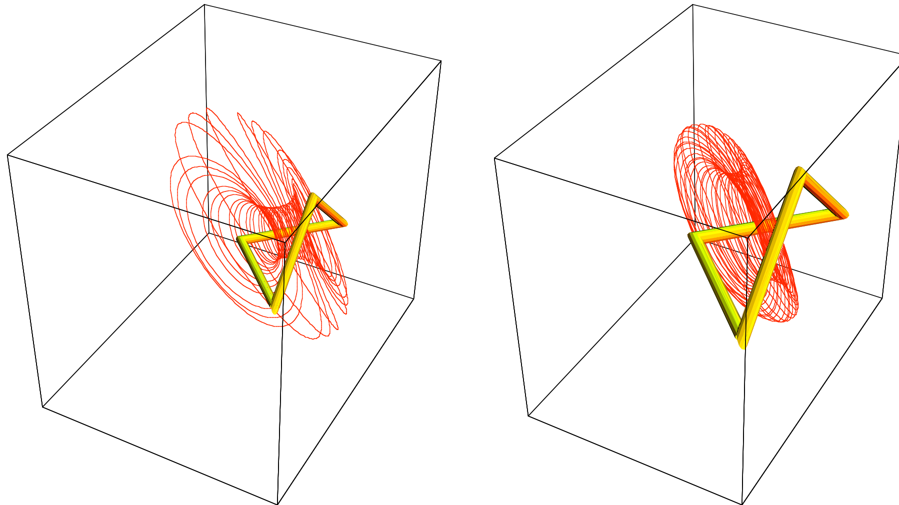
```
In[153]:= wireData = Flatten /@ Partition[N[wire], 2, 1];

          BField[{x_, y_, z_}] :=
              (Plus @@ Apply[BFiniteWireC[x, y, z, ##]&, wireData, {1}]);

In[155]:= makeFieldComponentDefinitions[
               {Bx, By, Bz}, BField, {x, y, z}, Real];
```

For two initial points, we solve the field line differential equation for an arc length of 200.

```
In[156]:= tMax = 200;
          fieldLines =
                 calculateFieldLine[{cx, cy, cz}, #, {Bx, By, Bz},
                  {t, 0, tMax}, MaxSteps -> 10^4,
                       PrecisionGoal -> 8, AccuracyGoal -> 8]& /@
                      {{-0.767890, 0.4911667, -1.415637},
                       {-1.402164, 1.9174317, -1.398096}};
```

The field lines show a rather complicated behavior. They are no longer closed lines. (Sometimes one reads the statement that magnetic field lines are always closed because of the Maxwell equation div $\vec{B}(\vec{x}) = 0$. This equation says nothing about a single field line; it only makes a statement about the *density* of field lines in some volume element [1680✱]. While there exists some closed field lines [1228✱], "most" field lines are not closed for a current arrangement.)

```
In[158]:=
          With[{gr = tubeGraphics3D[Line[wire], {0.06, #1&}, currentColor]},
          Show[GraphicsArray[
          Show[{gr, (* the field lines *)
          ParametricPlot3D[Evaluate[{cx[t], cy[t], cz[t],
                               {Thickness[0.002], Hue[0]}} /. #],
                              {t, 0, tMax}, PlotPoints -> 2500,
                              DisplayFunction -> Identity]},
                DisplayFunction -> Identity, PlotRange -> All,
                Boxed -> True, Axes -> False]& /@ fieldLines]]
```



```
Out[158]= - GraphicsArray -
```

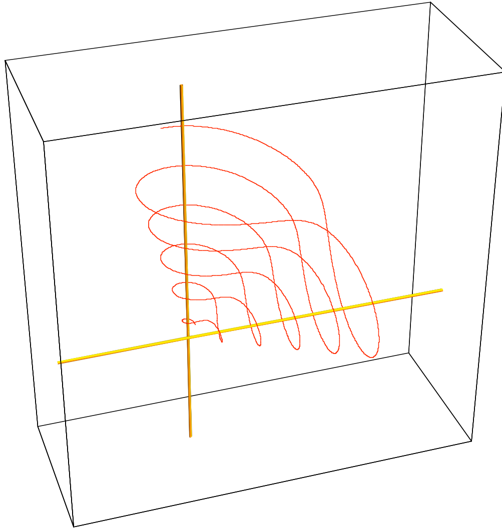
If we allow for infinite wires again, already two perpendicular wires generate interesting field lines. Here is the magnetic field generated by a current along the z-axis and a second current parallel to the y-axis through the point {1, 0, 0}.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[159]:= BField[{x_, y_, z_}] := {-y, x, 0}/(x^2 + y^2) -
                                  {-z, 0, (x - 1)}/((x - 1)^2 + z^2)

In[160]:= makeFieldComponentDefinitions[{Bx, By, Bz}, BField, {x, y, z}, Real];
```

The field line calculated and displayed winds around both wires in a "symmetric" way.

```
In[161]:= tMax = 200;
          fieldLine =  calculateFieldLine[{cx, cy, cz},
                    {0.027622, 0.551576, 0.378692}, {Bx, By, Bz},
                  {t, 0, tMax}, MaxSteps -> 10^4];

In[163]:= With[{grs = tubeGraphics3D[Line[#], {0.1}, currentColor]& /@
                {{{0, 0, -10}, {0, 0, 0}, {0, 0, 20}},
                 {{1, -10, 0}, {1, 0, 0}, {1, 20, 0}}}},
           Show[{grs, ParametricPlot3D[
              Evaluate[{cx[t], cy[t], cz[t],
               {Thickness[0.002], Hue[0]}} /. fieldLine], {t, 0, tMax},
               PlotPoints -> 2000, DisplayFunction -> Identity]},
                     DisplayFunction -> $DisplayFunction,
                     PlotRange -> All, Boxed -> True,
                     ViewPoint -> {3, -1, 2}, BoxRatios -> Automatic]]
```



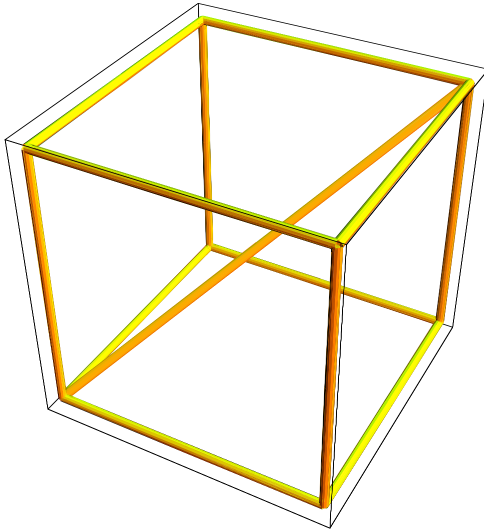
```
Out[163]= - Graphics3D -
```

From the last picture, one should not infer that any nonplanar or any nonparallel arrangement of currents causes a complicated field line pattern. Here, we consider a set of wires arranged in the form of the edges of a cube. The current flows from one vertex to the opposite one and back along the diagonal. These are the single straight wires forming the wire-cube.

```
In[164]:= wirePieces =
          {(* "diagonal wire" *)
           {{{1, 1, 1}, {0, 0, 0}}},
           (* from {0, 0, 0} and to {1, 1, 1} *)
           {{{0, 0, 0}, {1, 0, 0}}, {{0, 0, 0}, {0, 1, 0}},
            {{0, 0, 0}, {0, 0, 1}}, {{0, 1, 1}, {1, 1, 1}},
            {{1, 0, 1}, {1, 1, 1}}, {{1, 1, 0}, {1, 1, 1}}},
           (* the six "middle" pieces *)
           {{{1, 0, 0}, {1, 1, 0}}, {{1, 0, 0}, {1, 0, 1}},
            {{0, 1, 0}, {1, 1, 0}}, {{0, 1, 0}, {0, 1, 1}},
            {{0, 0, 1}, {1, 0, 1}}, {{0, 0, 1}, {0, 1, 1}}}} // N;
```

© 2004, 2005 Springer Science+Business Media, Inc.

In[165]:=
```
gr = Show[tubeGraphics3D[Line[{#[[1]], (#[[1]] + #[[2]])/2, #[[2]]}],
                          {}, currentColor]& /@ Flatten[wirePieces, 1]]
```



Out[165]=
```
- Graphics3D -
```

For the calculation of the magnetic field, we have to take into account that all of the current is flowing along the diagonal, but that the current along the cube edges is only a fraction of the first.

In[166]:=
```
BField[{x_, y_, z_}] :=
  1/1(Plus @@ Apply[BFiniteWireC[x, y, z, ##]&,
                    Flatten /@ wirePieces[[1]], {1}]) +
  1/3(Plus @@ Apply[BFiniteWireC[x, y, z, ##]&,
                    Flatten /@ wirePieces[[2]], {1}]) +
  1/6(Plus @@ Apply[BFiniteWireC[x, y, z, ##]&,
                    Flatten /@ wirePieces[[3]], {1}])
 (* or, shorter, but less readable:
  Sum[2/(k (k + 1)) (Plus @@ Apply[BFiniteWireC[x, y, z, ##]&,
                     Flatten /@ wirePieces[[k]], {1}]), {k, 3}] *)
```

We generate the definitions for the magnetic field and calculate two field lines for two selected initial conditions.

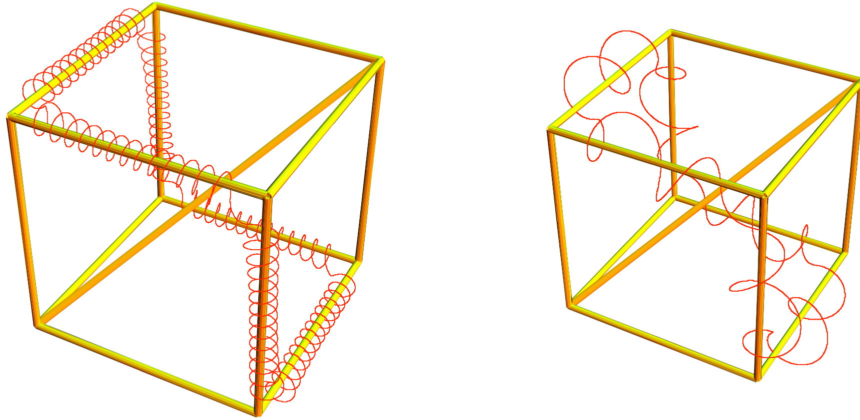
In[168]:=
```
makeFieldComponentDefinitions[
     {Bx, By, Bz}, BField, {x, y, z}, Real];
```

In[169]:=
```
tMax = 50;
fieldLines = calculateFieldLine[{cx, cy, cz}, #, {Bx, By, Bz},
        {t, 0, tMax}, MaxSteps -> 10^5, PrecisionGoal -> 6]& /@
        {{0.038, 1.029, 0.4565}, {0.74097, -0.0416, 1.10925}};
```

All field lines are now closed curves.

© 2004, 2005 Springer Science+Business Media, Inc.

In[171]:=
```
Show[GraphicsArray[
Show[{gr, ParametricPlot3D[
  Evaluate[{cx[t], cy[t], cz[t],
     {Thickness[0.002], Hue[0]}} /. #], {t, 0, tMax},
     PlotPoints -> 10000, DisplayFunction -> Identity]},
           DisplayFunction -> Identity,
           PlotRange -> All, Boxed -> False]& /@  fieldLines]]
```



Out[171]= - GraphicsArray -

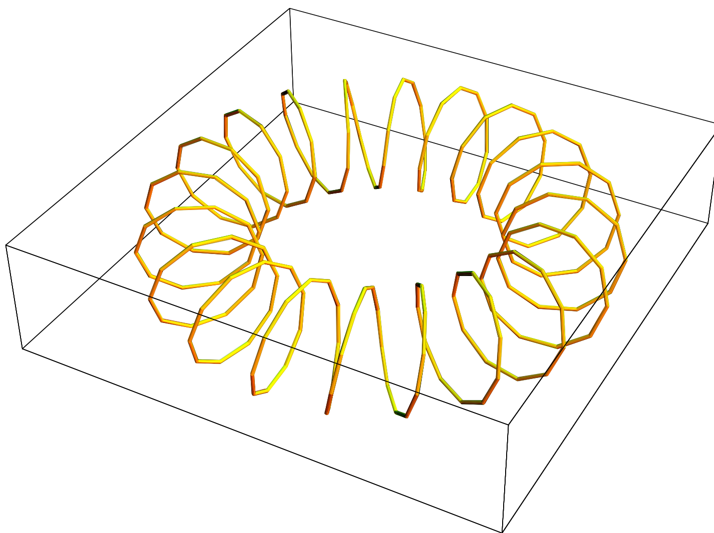
In the last example, we will calculate magnetic field lines of a ringcoil. We use the following parametrization for the coil.

In[172]:=
```
spaceCurve[ϑ_] = {3 Cos[ϑ] + Cos[ϑ] Cos[20 ϑ],
                  3 Sin[ϑ] + Sin[ϑ] Cos[20 ϑ], Sin[20 ϑ]};
```

We will start by calculating the field lines for a 200-piece approximation to `spaceCurve`.

In[173]:=
```
wire = Table[spaceCurve[ϑ], {ϑ, 0, 2. Pi, 2. Pi/200}];
```

In[174]:=
```
gr = Show[tubeGraphics3D[Line[wire], {0.03, #1&}, currentColor]]
```



Out[174]= - Graphics3D -

© 2004, 2005 Springer Science+Business Media, Inc.

Proceeding as above, we define the field components. This time, the calculation of `BField` is more expensive—because of the 200 finite straight wires involved. We use a `SetDelayed[Set[...]]` construction to avoid the recalculation of the field when calculating the field components.

```
In[175]:= wireData =
            Developer`ToPackedArray[Flatten /@ Partition[N[wire], 2, 1]];

In[176]:= Clear[BField];
          BField[{x_, y_, z_}] := BField[{x, y, z}] =
              (Plus @@ Apply[BFiniteWireC[x, y, z, ##]&, wireData, {1}]);

In[178]:= makeFieldComponentDefinitions[
                {Bx, By, Bz}, BField, {x, y, z}, Real];
```

For two initial conditions, we solve the field line differential equation for an arc length of 100. We choose three initial conditions representing different “types” of field lines.

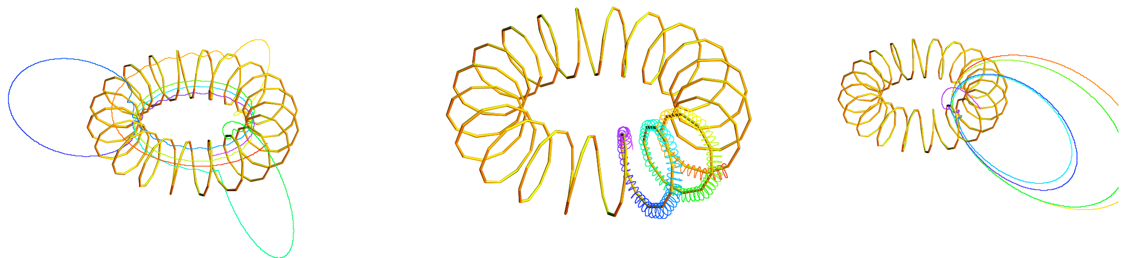
```
In[179]:= (tMax = 100;
          fieldLines = calculateFieldLine[{cx, cy, cz}, #, {Bx, By, Bz},
                           {t, 0, tMax}, MaxSteps -> 10^5,
                           PrecisionGoal -> 6, AccuracyGoal -> 6]& /@
               {{3.1, 0., 0.}, {3.7, 0., -0.667}, {1.55, 0., 0.}};) // Timing

Out[179]= {56.442 Second, Null}
```

To better see the behavior field lines inside the ringcoil, we color the field line from red to blue.

```
In[180]:= colorLines[l_List] :=
          With[{n = Length[l]}, MapIndexed[{Hue[#2[[1]]/n 0.8], #1}&, l]]
```

Here are the field lines. The first field line starts near the center of the cross section of the coil. After winding around, it leaves the coil and comes back. The second field line starts near the wire forming the coil and winds around the wire. The third line starts in the middle of the ring and surrounds the cross section (the whole cross section acts as a current-carrying wire for it). None of the field lines displayed is closed.

```
In[181]:= Show[GraphicsArray[
          Show[{gr, Graphics3D[{Thickness[0.002],
            colorLines[ParametricPlot3D[
              Evaluate[{cx[t], cy[t], cz[t]} /. #], {t, 0, tMax},
              PlotPoints -> 2500, DisplayFunction -> Identity][[1, 1]]]}]},
                     DisplayFunction -> Identity, PlotRange -> All,
                     Boxed -> False, Axes -> False]& /@ fieldLines],
                GraphicsSpacing -> -0.1]
```



```
Out[181]= - GraphicsArray -
```

The first of the last pictures might seem unexpected. To show that it is not an artifact based on the discretization of the wire, we will calculate the field for the nondiscretized version of `spaceCurve`. We will use `NIntegrate` to calculate the field values. To avoid compiling the integrand at each call to `NIntegrate`, we compile the integrand once in the beginning and then call `NIntegrate` with the option setting `Compiled -> False`. Of course, the calculation of the magnetic field will take longer than in the discretized case.

© 2004, 2005 Springer Science+Business Media, Inc.

The first of the last pictures might seem unexpected. To show that it is not an artifact based on the discretization of the wire, we will calculate the field for the nondiscretized version of `spaceCurve`. We will use `NIntegrate` to calculate the field values. To avoid compiling the integrand at each call to `NIntegrate`, we compile the integrand once in the beginning and then call `NIntegrate` with the option setting `Compiled -> False`. Of course, the calculation of the magnetic field will take longer than in the discretized case.

```
In[182]:= {cfx, cfy, cfz } =
     Compile[{t, x, y, z},
            Module[{c = Cos[t], s = Sin[t],
                    c20 = Cos[20t], s20 = Sin[20t]}, #]]& /@
            (Cross[spaceCurve'[ϑ], {x, y, z} - spaceCurve[ϑ]]/
              (#.#)^(3/2)&[{x, y, z} - spaceCurve[ϑ]] /.
      {Cos[ϑ] -> c, Sin[ϑ] -> s, Cos[20ϑ] -> c20, Sin[20ϑ] -> s20});

In[183]:= Clear[BField];

     BField[{x_, y_, z_}] := (BField[{x, y, z}] =
     NIntegrate[{cfx[t, x, y, z], cfy[t, x, y, z], cfz[t, x, y, z]},
                (* use intermediate points *)
                Evaluate[Flatten[{t, Table[ϑ, {ϑ, 0, 2Pi, 2Pi/20}]}]],
                Compiled -> False, Method -> GaussKronrod,
                AccuracyGoal -> 4, PrecisionGoal -> 6])

In[185]:= makeFieldComponentDefinitions[{Bx, By, Bz}, BField, {x, y, z}, Real];
```

We calculate a field line starting exactly at the center of the coil's cross section.

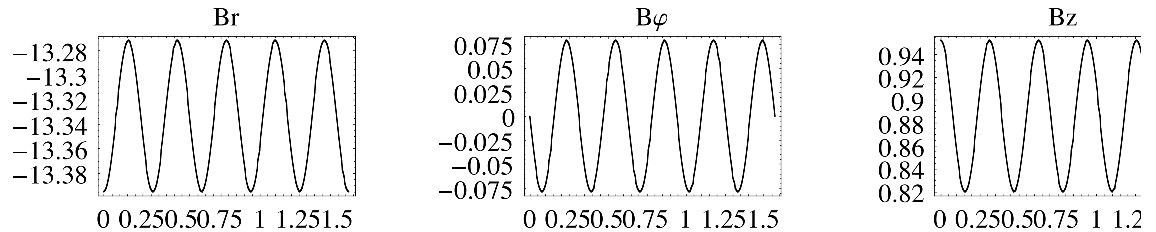
```
In[186]:= (tMax = 60;
     fieldLine = calculateFieldLine[{cx, cy, cz},
            {3., 0., 0.}, {Bx, By, Bz},
            {t, 0, tMax}, MaxSteps -> 10^5,
                 PrecisionGoal -> 6, AccuracyGoal -> 6];) // Timing

Out[186]= {136.947 Second, Null}
```

Because of the finite number of revolutions and the finite slope of the revolutions, the magnetic field is not rotational invariant around the z-axis. The next three graphics show the radial, the azimuthal, and the z-component of the field along the center of the cross section. The azimuthal periodicity is clearly visible.

```
In[187]:= BFieldData = Table[{φ, BField[{3 Cos[φ], 3 Sin[φ], 0}]} // N,
                            {φ, 0, Pi/2, Pi/2/200}];
```

© 2004, 2005 Springer Science+Business Media, Inc.

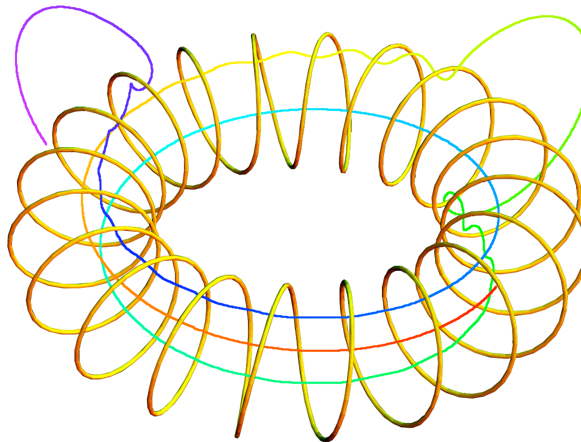
```
In[188]:= Show[GraphicsArray[
          Apply[Function[{f, l},
          ListPlot[{#[[1]], f}& /@ BFieldData,
                   PlotRange -> All, Frame -> True, Axes -> False,
                   PlotJoined -> True, PlotLabel -> l,
                   DisplayFunction -> Identity]],
                (* the components *)
                {{Unevaluated[#[[2, 1]] Sin[-#[[1]]] +
                              #[[2, 2]] Cos[ #[[1]]]], "Br"},
                 {Unevaluated[#[[2, 1]] Cos[ #[[1]]] +
                              #[[2, 2]] Sin[ #[[1]]]], "Bφ"},
                 {Unevaluated[#[[2, 3]]], "Bz"}}, {1}]]]
```




```
Out[188]= - GraphicsArray -
```

The field line that starts at the center of the cross section moves away from the center and actually leaves the coil after a while.

```
In[189]:= ringCoilGraphics = tubeGraphics3D[
           Line[Table[spaceCurve[ϑ], {ϑ, 0, 2. Pi, 2. Pi/600}]],
                      {0.03, #1&}, currentColor];

In[190]:= Show[{ringCoilGraphics, Graphics3D[{Thickness[0.002],
            colorLines[ParametricPlot3D[
              Evaluate[{cx[t], cy[t], cz[t]} /. fieldLine], {t, 0, tMax},
              PlotPoints -> 2500, DisplayFunction -> Identity][[1, 1]]]}]},
                     PlotRange -> All, Boxed -> False, Axes -> False]
```



```
Out[190]= - Graphics3D -
```

Here we end calculating and visualizing field lines and leave it to the reader to explore further configurations like molecules [788✱]. Current-carrying conductors of various shapes [1269✱], current-carrying 3D Peano curves, edges of polyhedra, and so

© 2004, 2005 Springer Science+Business Media, Inc.

on will yield interesting field lines. Knotted wires [1904✱], [1905✱], [1906✱] are another rich source of unexpected field line configurations. Are there closed field lines around them that are knotted too, [1828✱]? Then one could go on and investigate the field lines of time-dependent currents [650✱], [1516✱], [1851✱], [154✱], rotating magnets [1424✱], (super-)shielded magnetic fields [304✱] and so on.

```
In[191]:= Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

◦ **Session Summary:** (Evaluated with *Mathematica* 5.1)

Inputs evaluated:	190
CPU time used:	1828 s
Wall clock time elapsed:	1867 s
Max. memory used:	508 MB
Variables in use:	206

■ 1.12.2 Riemann Surfaces of Algebraic Functions

In Subsection 2.3.7 of the Graphics volume [1807✱], we discussed the visualization of Riemann surfaces of some simple, multivalued functions [1249✱]. Here, we will deal with a much more general class of functions: all algebraic functions. Given a (irreducible) polynomial p in two variables $p = \sum_{j,k=0}^{n} a_{ij}\, z^j\, w^k$, the equation $p = 0$ defines $w(z)$ implicitly as a (generically multivalued) function. This class of functions includes elliptic and hyperelliptic curves. Graphing the real or the imaginary part of $w(z)$ represents a faithful representation of the Riemann surface of $w(z)$ (for nonalgebraic functions, the faithfulness issue is more complicated). (For introductions on Riemann surfaces, see [153✱], [1245✱], [1296✱], [1402✱], and [1229✱]; for a more detailed mathematical discussion, see [28✱], [1532✱], [264✱], [589✱], [842✱], [1257✱], [1459✱], and [1910✱]. For some issues special to Riemann surfaces of algebraic functions, see [919✱], [297✱], [342✱], [930✱], [271✱], and [1519✱]. For the relevance of Riemann surfaces to physics, see, for instance, [734✱], [1238✱], [163✱], [265✱], and [1179✱].)

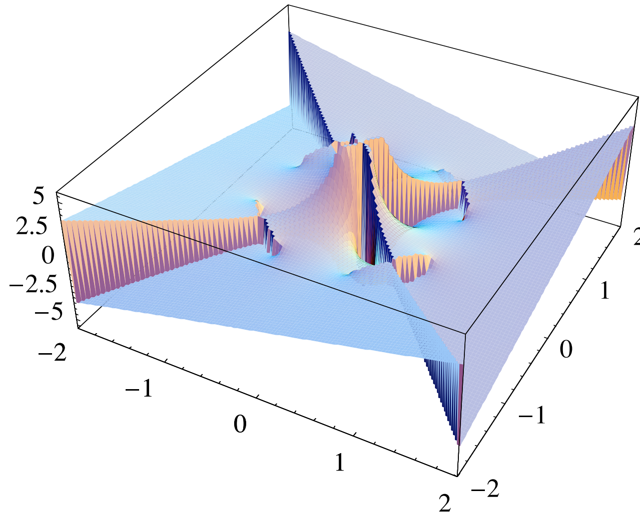
A numerical algorithm for the construction of the Riemann surface is given in the following and will be used to visualize the Riemann surfaces of some sample functions.

The key point of the functions used in Subsection 2.3.7 of the Graphics volume was the fact that their branch points and branch cuts were easy to determine and to exclude in the graphics to construct smooth surfaces.

For the function $w(z) = \sqrt[3]{z^4 + z^{-4}}$ (or $1 + w^3\, z^4 - z^8 = 0$), the determination of the form of the branch cuts becomes a bit more complicated; the discontinuities will no longer be located on a straight line. Here is the principal sheet.

© 2004, 2005 Springer Science+Business Media, Inc.

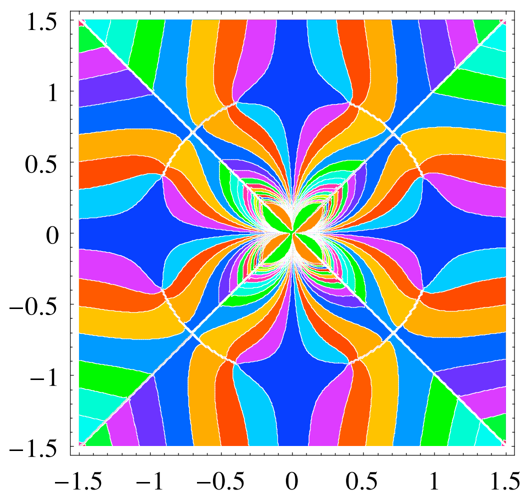
```
In[1]:= Plot3D[Evaluate[Im[(z^4 + z^-4)^(1/3) /. z -> x + I y]],
               {x, -2, 2}, {y, -2, 2}, PlotPoints -> 120, Mesh -> False]
```




```
Out[1]= - SurfaceGraphics -
```

Using a `ContourPlot` shows the branch cuts nicely; they form clusters of contour lines.

```
In[2]:= ContourPlot[Evaluate[Im[(z^4 + z^-4)^(1/3) /. z -> x + I y]],
                    {x, -3/2, 3/2}, {y, -3/2, 3/2},
                    PlotPoints -> 200, Contours -> 30,
                    ColorFunction -> (Hue[Random[]]&),
                    ContourStyle -> {{Thickness[0.002], GrayLevel[1]}}]
```




```
Out[2]= - ContourGraphics -
```

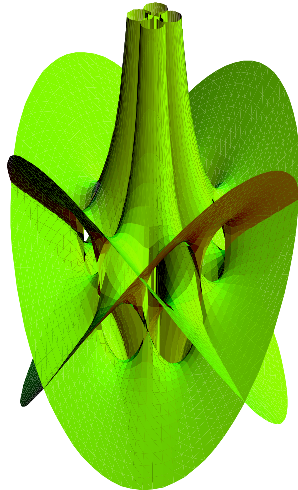
All of the branch cuts are caused by the branch cut of the `Power` function and occur when $z^4 + z^{-4}$ is equal to a negative real number v.

Analyzing the solutions of $z^4 + z^{-4} = v$ in detail shows that they form rays originating from $e^{i(\pi/4 + k\,\pi/2)}$, $k = 0,\ 1,\ 2,\ 3$ and going radially inward and outward, and arcs of the unit circle.

© 2004, 2005 Springer Science+Business Media, Inc.

Knowing the location of the branch cuts now allows us to split the complex z-plane into regions such that inside each region we can use `ParametricPlot3D` to get a smooth surface of the imaginary part. We generate all three possible values for the cube root and display all 24 (= 3 (different roots) × 4 (inside the unit circle) + 4 (outside the unit circle)) pieces.

```
In[3]:= With[{ε = 10^-6, ϵ = Exp[I 2Pi/3]},
    Show[Graphics3D[{EdgeForm[], Thickness[0.002],
         SurfaceColor[Hue[0.26], Hue[0.31], 2.15],
       Cases[Table[Apply[(* inside each sector *)
       ParametricPlot3D[{r Cos[φ], r Sin[φ],
         Im[ϵ^j ((r E^(I φ))^4 + (r E^(I φ))^-4)^(1/3)]},
              #1, #2, PlotPoints -> {12, 22},
                DisplayFunction -> Identity]&,
    Join @@ (* inside and outside the unit circle *)
       Table[{{r, ρ + ε, ρ + 1 - ε},
              {φ, -Pi/4 + ε + k Pi/2, +Pi/4 - ε + k Pi/2}},
              {ρ, 0, 1}, {k, 0, 3}], {1}], {j, 3}], _Polygon, Infinity]}],
     PlotRange -> {-4, 4}, BoxRatios -> {1, 1, 1.9}, Boxed -> False]]
```

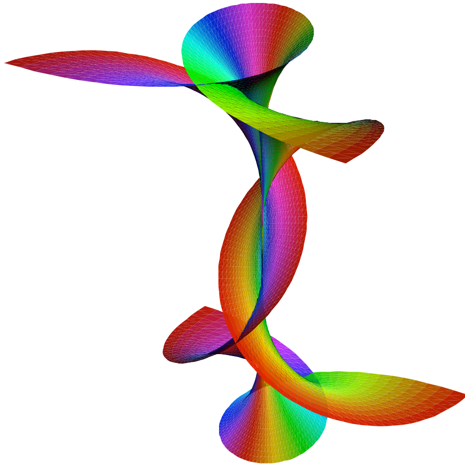


```
Out[3]= - Graphics3D -
```

When the polynomial $p = 0$ can be solved for z, it is possible to obtain a parametrization of the Riemann surface. Let us consider the polynomial $w^5 + w^3 = z$. The following parametric plot shows $\text{Im}(w)$ parametrized in the form $\text{Im}(w) = \text{Im}(w(z)) = \text{Im}(w(z(w))) = \text{Im}(w(z(\text{Re}(w) + i\ \text{Im}(w))))$. One clearly sees that the whole z-plane is covered multiple times.

© 2004, 2005 Springer Science+Business Media, Inc.

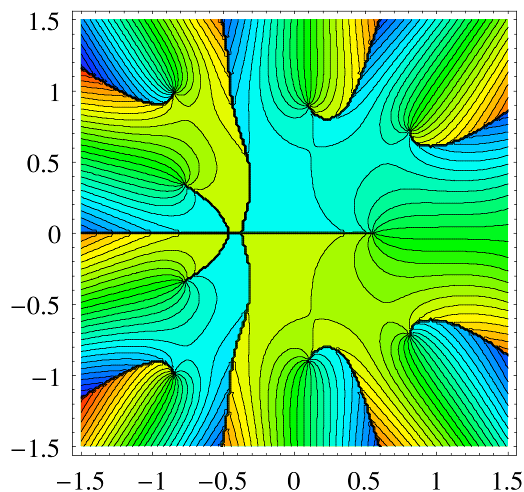
```
In[4]:= ParametricPlot3D[
        Evaluate[{Re[(wr + I wi)^5 + (wr + I wi)^3],
                  Im[(wr + I wi)^5 + (wr + I wi)^3], wi,
                  {EdgeForm[], SurfaceColor[Hue[wr/6]]}}],
           {wr, -6, 6}, {wi, -6, 6}, PlotPoints -> 100,
           BoxRatios -> {1, 1, 3/2}, PlotRange -> All,
           Axes -> False, Boxed -> False]
```



```
Out[4]= - Graphics3D -
```

For more complicated polynomials, which cannot be solved nicely for w or z, it turns out to be very difficult to determine the location of the branch cuts either numerically or in closed form. Here is an example of a more complicated polynomial.

```
In[5]:= ContourPlot[Evaluate[
             Im[(-5 + 4z + 6z^2 + z^3 + 5z^4 + 7z^5 -
                 7z^6 + 6z^7 + 4z^8 + 4z^9 - 4z^10)^(1/7) /. z -> x + I y]],
           {x, -3/2, 3/2}, {y, -3/2, 3/2}, PlotPoints -> 200,
           ColorFunction -> (Hue[0.7 #]&), Contours -> 30,
           ContourStyle -> {{Thickness[0.002], GrayLevel[0]}}]
```




```
Out[5]= - ContourGraphics -
```

When we cannot solve $p = \sum_{j,k=0}^{n} a_{ij}\, z^j\, w^k$ with respect to w [1546✱], we cannot describe the branch cuts at all. (Using the in Chapter 1 of the Symbolics volume [1808✱] discussed `Root`-objects, it is theoretically possible to describe the branch cuts, but for more complicated examples it will be prohibitively complicated.) (The function `Root` can always solve bivariate polynomials, but the resulting branch cuts—better lines of discontinuity—are quite complicated and do not solve the problem at hand.)

© 2004, 2005 Springer Science+Business Media, Inc.

When we cannot solve $p = \sum_{j,k=0}^{n} a_{ij}\, z^j\, w^k$ with respect to w [1546✱], we cannot describe the branch cuts at all. (Using the in Chapter 1 of the Symbolics volume [1808✱] discussed `Root`-objects, it is theoretically possible to describe the branch cuts, but for more complicated examples it will be prohibitively complicated.) (The function `Root` can always solve bivariate polynomials, but the resulting branch cuts—better lines of discontinuity—are quite complicated and do not solve the problem at hand.)

To avoid discontinuities in the graphics of the Riemann surface, we would have to restrict carefully the regions where to plot the function. On the other hand, branch cuts can be chosen to a large extent arbitrarily, and only the location of the branch points, where two sheets are glued together, represents a property inherent and well defined for an algebraic function. (Branch cuts have to connect branch points. But how exactly a branch cut is chosen in the principal sheet between the branch points does not matter). That is why we will avoid using closed-form radicals (or `Root`s) formulas and branch cuts at all, and instead use `NDSolve` to calculate smooth patches inside regions of the complex z-plane that are free of branch points [77✱]. This treatment completely *avoids* dealing with (on a Riemann surface anyway) nonexisting branch cuts. The patches will be chosen in such a way that the branch points (which must be treated carefully—here, two or more sheets are glued together) are always at their vertices. This also allows for a good spatial resolution near the branch points, which is important because typically $w(z)$ varies most near a branch point. This will result in a smooth in 3D self-intersecting surface that represents the Riemann surface of the polynomial under consideration.

To construct the Riemann surface, we will proceed with the following steps:
1) We use `NSolve` to calculate all potential branch points p.
2) We divide a circular part of the complex z-plane radially and azimuthally into sectors generated by the outer product of the radial and azimuthal coordinates of the branch points, so that every sector has no branch point inside.
3) We derive a differential equation for $w'(z)$ from p.
4) By starting (quite) near the branch points, we solve the differential equation radially outward for all sheets of every sector.
5) Starting from the radial solutions of the differential equations, we solve the differential equation azimuthally inside the sectors constructed in 3). This results in parametrized (by the radius r and the angle φ) parts of all sheets for each sector.
6) We generate polygons from the parametrizations for all sheets and all sectors and display all things together (and maybe cut holes into the polygons for better visibility of the inner parts).

Now, let us implement this construction.

1) At a branch point, the equation $p = 0$ has a multiple solution for $w(z)$. This means p and $\frac{\partial p}{\partial w}$ must both be zero [271✱]. We use `NSolve` to calculate a high-precision approximation of potential branch points.

```
In[6]:= branchPoints[poly_, {w_, z_}] :=
          Union[z /. NSolve[{poly == 0, D[poly, w] == 0}, {z, w},
                            WorkingPrecision -> 200],
                SameTest -> (Abs[#1 - #2] < 10^-6&)]
```

As an example for the polynomial p, let us take the following bivariate polynomial. (We will use this polynomial throughout implementing the approach sketched above for exemplifying the various functions defined below.)

```
In[7]:= pExample = 2z^5 - z^3 + z^4 - 2z^2 w - z w^2 + 2z w^4 + w^5;
```

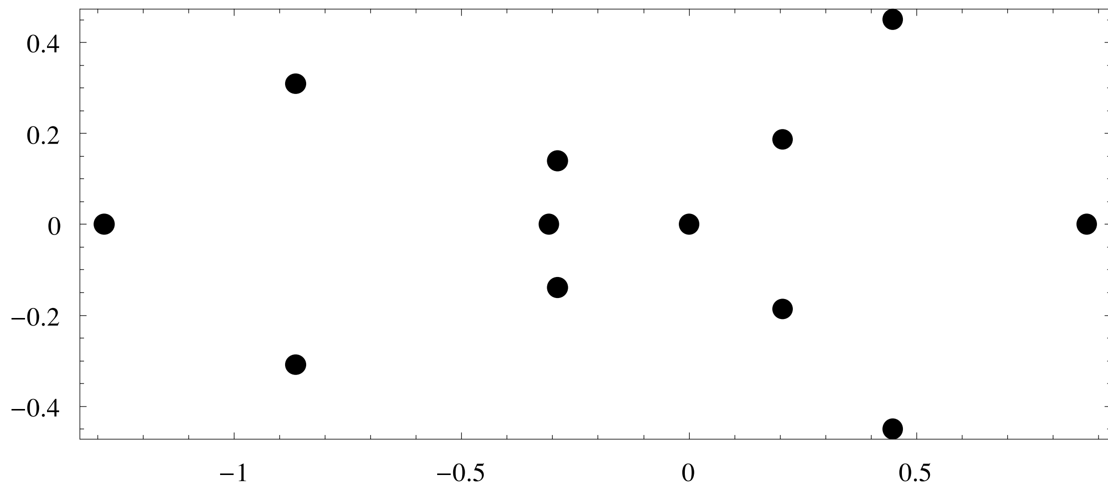
This function has 12 potential branch points. (For a detailed discussion of the properties of branch points of Riemann surfaces of algebraic functions, see [1662✱], [356✱], [612✱], [408✱], [1662✱], [1252✱], [842✱], [1253✱], [591✱], [9✱], [10✱], [532✱], [1883✱], [342✱], [614✱], [1288✱], and [1766✱].)

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[8]:= Off[NSolve::"zerosol"];
        (potentialBranchPointsExample = branchPoints[pExample, {w, z}]) // N
Out[9]= {-1.28509, -0.864198 - 0.308813 i, -0.864198 + 0.308813 i,
         -0.307643, -0.288979 - 0.139225 i, -0.288979 + 0.139225 i,
         -1.57035 × 10^-134, 0.206333 - 0.186587 i, 0.206333 + 0.186587 i,
         0.448111 - 0.450203 i, 0.448111 + 0.450203 i, 0.873818}
```

Here, their location in the complex *z*-plane is shown.

```
In[10]:= Show[Graphics[{PointSize[0.02], Point[{Re[#], Im[#]}]& /@
                         potentialBranchPointsExample}],
              PlotRange -> All, AspectRatio -> Automatic,
              Frame -> True, Axes -> False]
```




```
Out[10]= - Graphics -
```

2) Given the branch points, we calculate their distances from the origin and their phase and divide the r,φ-plane into radial and azimuthal sectors that are formed between radially and azimuthally adjacent branch points.

```
In[11]:= sectors[poly_, {w_, z_}] :=
         Module[{ε = 10^-5, (* the branch points *)
                 bps = branchPoints[poly, {w, z}], arg, rMax, rList, φList},
                 (* a relative of the Arg function;
                    defined at the origin *)
                 arg[ξ_] := If[ξ == 0., 0``25, Arg[ξ]];
                 rMax = (If[#1 == 0., 1, 12/10 #1] &)[
                 Max[Abs[rList = Abs /@ bps]]];
                 (* r- and φ-values *)
                 rList = Union[Prepend[Append[rList, rMax], 0],
                               SameTest -> (Abs[#1 - #2] < ε&)];
                 (* sort counter-clockwise *)
                 φList = Sort[Union[arg /@ bps, SameTest -> (Abs[#1 - #2] < ε&)],
                              #1 < #2&];
                 φList = Append[φList, First[φList] + 2Pi];
                 (* the sectors *)
                 N[#1, 25]&[Flatten[Table[
                  {rList[[i]] + ε, φList[[j]] + ε,
                   rList[[i + 1]] - ε, φList[[j + 1]] - ε},
                  {i, Length[rList] - 1}, {j, Length[φList] - 1}], 1]]]
```

Here, the sectors for our example polygon p`Example` are calculated.

© 2004, 2005 Springer Science+Business Media, Inc.

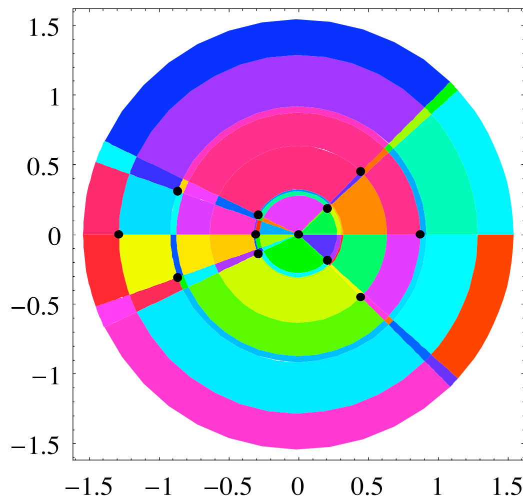
In[12]:=
```
Short[s1 = sectors[pExample, {w, z}], 8]
```

Out[12]//Short=
```
{{0.0000100000000000000000000000, -2.7983837826913865563l2530,
  0.278177147012134084851 5355, -2.6926368234612735428l0424},
 <<78>>, {1.28510130617028329693 5922, 3.14160265358979323846 2643,
  1.54209956740433995632 3107, 3.48478152448819992061 2757}}
```

Let us visualize the sectors together with the branch points by coloring each sector with a randomly chosen color.

In[13]:=
```
sectorPolygon[{r1_, φ1_, r2_, φ2_}] :=
  With[{pp = 12}, Polygon[
    Join[Table[r1 {Cos[φ], Sin[φ]}, {φ, φ1, φ2, (φ2 - φ1)/pp}],
         Table[r {Cos[φ2], Sin[φ2]}, {r, r1, r2, (r2 - r1)/pp}],
         Table[r2 {Cos[φ], Sin[φ]}, {φ, φ2, φ1, (φ1 - φ2)/pp}],
         Table[r {Cos[φ1], Sin[φ1]}, {r, r2, r1, (r1 - r2)/pp}]]]]
```

In[14]:=
```
Show[Graphics[{{Hue[Random[]], sectorPolygon[#1]}& /@ s1,
                PointSize[0.02], Point[{Re[#1], Im[#1]}]& /@
                                  potentialBranchPointsExample}],
   PlotRange -> All, Frame -> True, AspectRatio -> Automatic]
```



Out[14]= - Graphics -

3) Now, let us derive a differential equation that is equivalent to the polynomial under consideration.

The solution of the differential equation we are looking for should reproduce the root of the original polynomial. If we integrate the differential equation and avoid branch points along the integration path, this solution will be a smooth function representing the original root. We will derive a first-order nonlinear differential equation. (We could also generate higher order linear differential equations with polynomial coefficients [see [1412✱] for details], but for higher order equations, constructing the initial conditions is no longer so easy—for the first-order one just has to solve numerically the polynomial under consideration for a given z with respect to w.)

If we view w as a function of z, we can differentiate the polynomial with respect to z and solve the resulting linear equation in $w'(z)$.

In[15]:=
```
ode[poly_, {w_, z_}] :=
    Equal @@ Solve[D[poly /. w -> w[z], z] == 0, w'[z]][[1, 1]]
```

The nonlinear first-order differential equation we get this way is the following for our example polynomial.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[16]:= odepExample = ode[pExample, {w, z}]
```

$$\text{Out[16]=}\quad w'[z] == \frac{-3 z^2 + 4 z^3 + 10 z^4 - 4 z\, w[z] - w[z]^2 + 2 w[z]^4}{2 z^2 + 2 z\, w[z] - 8 z\, w[z]^3 - 5 w[z]^4}$$

Later, we need to solve this differential equation in azimuthal and radial directions in a polar coordinate system. Transforming the independent variable from z to φ, we get the differential equation in azimuthal direction.

```
In[17]:= odeφ[ode_, {w_, z_}, {ψ_, φ_}, r_] :=
          ode //. {(w'[z] == rhs_) -> (ψ'[φ] ==  rhs D[r Exp[I φ], φ]),
                   w[z] -> ψ[φ], z -> r Exp[I φ]}
```

For our example polynomial, we get the following differential equation.

```
In[18]:= odeφExample = odeφ[odepExample, {w, z}, {ψ, φ}, r]
```

$$\text{Out[18]=}\quad \psi'[\varphi] == \frac{i\, e^{i\varphi} r\, (-3 e^{2 i\varphi} r^2 + 4 e^{3 i\varphi} r^3 + 10 e^{4 i\varphi} r^4 - 4 e^{i\varphi} r\, \psi[\varphi] - \psi[\varphi]^2 + 2 \psi[\varphi]^4)}{2 e^{2 i\varphi} r^2 + 2 e^{i\varphi} r\, \psi[\varphi] - 8 e^{i\varphi} r\, \psi[\varphi]^3 - 5 \psi[\varphi]^4}$$

Transforming the independent variable from z to r, we get the differential equation in radial direction d.

```
In[19]:= oder[ode_, {w_, z_}, {ψ_, ρ_}, d_] :=
          ode //. {(w'[z] == rhs_) -> (ψ'[ρ] == rhs D[ρ d, ρ]),
                   w[z] -> ψ[ρ], z -> d ρ}
```

For our example polynomial, we get the following differential equation.

```
In[20]:= oderExample = oder[odepExample, {w, z}, {ψ, ρ}, d]
```

$$\text{Out[20]=}\quad \psi'[\rho] == \frac{d\, (-3 d^2 \rho^2 + 4 d^3 \rho^3 + 10 d^4 \rho^4 - 4 d\, \rho\, \psi[\rho] - \psi[\rho]^2 + 2 \psi[\rho]^4)}{2 d^2 \rho^2 + 2 d\, \rho\, \psi[\rho] - 8 d\, \rho\, \psi[\rho]^3 - 5 \psi[\rho]^4}$$

We will not carry out the points 4), 5), and 6) sequentially, but for each sector at once. The advantage of this approach is that we do not have to store a large number of `InterpolatingFunction`-objects (which can be quite memory intensive).

For the numerical solution of the obtained differential equations, we set some options in `NDSolve`. (The following option settings will be suitable for the examples treated in the following—more complicated polynomials might need larger values of the maximum number of steps as well of the precision/accuracy goals.)

```
In[21]:= SetOptions[NDSolve, PrecisionGoal -> 12, AccuracyGoal -> 12,
                             MaxSteps -> 10000];
```

The function `patch` calculates polygons of all sheets of the polynomial *poly* using *ppφ* azimuthal and *ppr* radial plot points in a given sector. (We give the radial as well as the azimuthal differential equation as arguments to `patch`. Although they could be calculated from *poly*, later we have to call `patch` on every one of the sectors, and we would like to avoid recalculating the differential equation for every sector.)

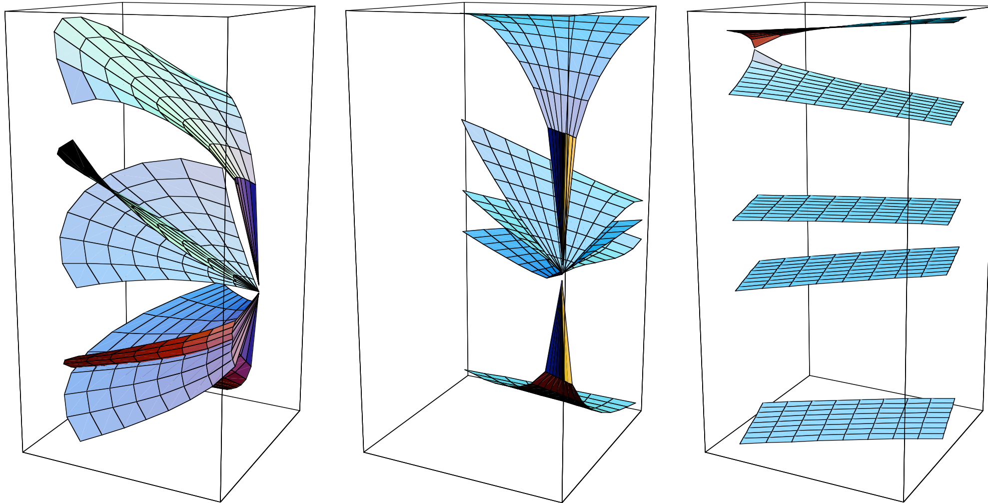
© 2004, 2005 Springer Science+Business Media, Inc.

```
In[22]:= patch[reIm_, {poly_, {w_, z_}},
          {odeφ_, {ψ_, φ_}, r_}, {oder_, {ψ_, ρ_}, dir_},
          {r1_, φ1_, r2_, φ2_}, {ppφ_, ppr_}] :=
    Module[{radialStartingValues, radialIFs, azimuthalIFs, points, ra, φa},
             (* solve poly == 0 to get initial values
                for the differential equation *)
            radialStartingValues = w /.
                  Solve[poly == 0 /. {z -> r1 Exp[I φ1]}, w];
             (* solve the differential equation radially *)
            radialIFs = NDSolve[{oder /. dir -> Exp[I φ1],
                        ψ[r1] == #}, ψ, {ρ, r1, r2}][[1, 1, 2]]& /@
                                 radialStartingValues;
             (* solve the differential equation azimuthally *)
            azimuthalIFs = Table[ra = r1 + i/ppr(r2 - r1);
             NDSolve[{odeφ /. r -> ra, ψ[φ1] == radialIFs[[j]][ra]},
                     ψ, {φ, φ1, φ2}][[1, 1, 2]],
                               {j, Length[radialIFs]}, {i, 0, ppr}];
            (* calculate points for all sheets *)
            points = Table[ra = r1 + i/ppr(r2 - r1);
                           φa = φ1 + k/ppφ(φ2 - φ1);
                           {ra Cos[φa], ra Sin[φa],
                           reIm[azimuthalIFs[[j, i + 1]][φa]]},
                           {j, Length[radialIFs]},
                           {i, 0, ppr}, {k, 0, ppφ}];
             (* make polygons for all sheets *)
            Function[s, Apply[Polygon[Join[#1, Reverse[#2]]]&,
              Transpose /@ Partition[Partition[#, 2, 1]& /@ s, 2, 1],
                                                        {2}]] /@ points]
```

Here, the six sheets inside three of the sectors of our example polynomial are calculated. Each of the three examples contains a clearly visible branch point.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[23]:= Show[GraphicsArray[
           (Graphics3D[{EdgeForm[Thickness[0.002]],
             patch[Im, {pExample, {w, z}},
               {odeφExample, {ψ, φ}, r}, {oderExample, {ψ, ρ}, d},
               s1[[#1]], {8, 8}]},
              BoxRatios -> {1, 1, 2}, ViewPoint -> {2, 1, 0.6}]&) /@
                        {2, 10, 21}]]
```



```
Out[23]= - GraphicsArray -
```

For the whole Riemann surface, we want the mesh in the patches as uniform as possible, and at the same time, we want the branch points always near mesh points. The function `patchPlotPoints` calculates the number of radial and azimuthal points inside a given sector, provided the whole number of radial and azimuthal points should be around *ppr* and *ppφ*. (The actual number of plot points may slightly deviate from *ppr* and *ppφ*.)

```
In[24]:= patchPlotPoints[{r1_, φ1_, r2_, φ2_}, rMax_, {ppφ_, ppr_}] :=
         Max[#, 1]& /@ Round[{ppφ, ppr} {φ2 - φ1, r2 - r1}/{2Pi, rMax}]
```

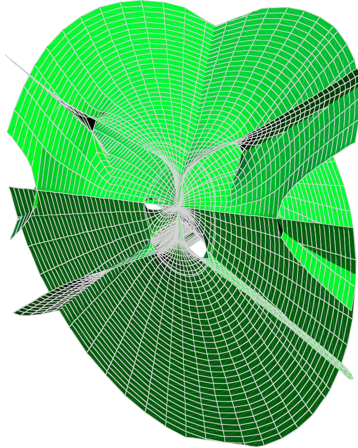
Now, we have all of the pieces together to implement the function `RiemannSurface`. Its first argument *poly* is a bivariate polynomial in *w* and *z*, the second argument is the function to be shown (in most cases `Re` or `Im`), and the last argument is the number of radial and azimuthal plot points. The function `RiemannSurface` returns a list of polygons of all sheets.

```
In[25]:= RiemannSurface[poly_, {w_, z_}, reIm_, {ppφ_, ppr_}] :=
         Module[{s, rMax, odePoly, odePolyφ, odePolyr, ψ, φ, r, dir},
                (* subdivide the complex plane *)
                s = sectors[poly, {w, z}];
                rMax = Max[#[[3]]& /@ s];
                (* calculate differential equations *)
                odePoly = ode[poly, {w, z}];
                odePolyφ = odeφ[odePoly, {w, z}, {ψ, φ}, r];
                odePolyr = oder[odePoly, {w, z}, {ψ, ρ}, dir];
                (* calculating the patches *)
                 Table[patch[reIm, {poly, {w, z}},
                       {odePolyφ, {ψ, φ}, r}, {odePolyr, {ψ, ρ}, dir}, s[[i]],
                       patchPlotPoints[s[[i]], rMax, {ppφ, ppr}]],
                     {i, Length[s]}]]
```

Here, one-half of the Riemann surface of `pExample` is shown. A cross-sectional view along the plane $y = 0$ nicely shows the branching at the four branch points along the real axis.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[26]:= Show[Graphics3D[{EdgeForm[{Thickness[0.002], GrayLevel[0.8]}],
              SurfaceColor[Hue[0.35], Hue[0.35], 2.2],
              RiemannSurface[pExample, {w, z}, Im, {50, 36}]}],
         PlotRange -> {All, {0, 3/2}, All},
         ViewPoint -> {0.9, -3.3, 1.6}, Boxed -> False]
```

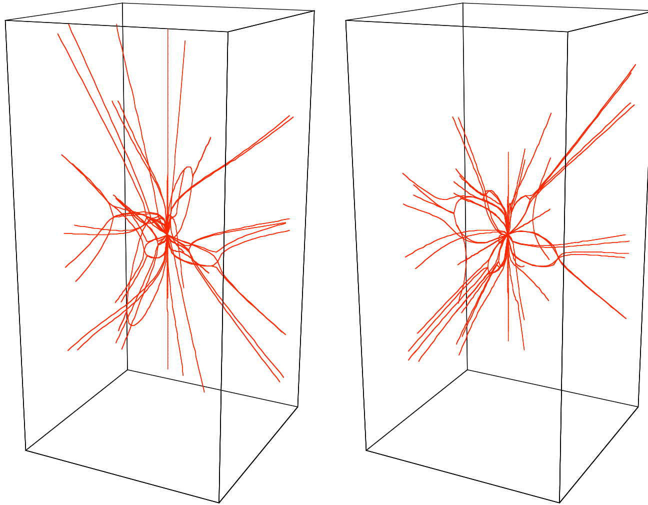


```
Out[26]= - Graphics3D -
```

By using only one plot point in the azimuthal direction and replacing polygons with lines, we can make a picture showing cross sections along all directions that contain branch points. One nicely sees the splitting of the branches at the branch points (generically, one expects most branch points to be of the square-root type [1986✱]). Both the real as well as the imaginary parts are shown.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[27]:= Show[GraphicsArray[(Function[l, Function[reIm,
           Graphics3D[{Thickness[0.002], Hue[0],
                             Map[If[Head[#1] === List,
                MapAt[reIm, #, 3], #]&, l, {-2}]},
          ViewPoint -> {-2, -1, 0.7}, BoxRatios -> {1, 1, 2},
          PlotRange -> {All, All, {-3, 3}}]] /@
           {Re, Im}])[RiemannSurface[𝓅Example, {w, z}, Identity,
             {1, 30}] /. Polygon[{p1_, p2_, p3_, p4_}] :>
             {Line[{p2, p3}], Line[{p4, p1}]}]]]
```



```
Out[27]= - GraphicsArray -
```

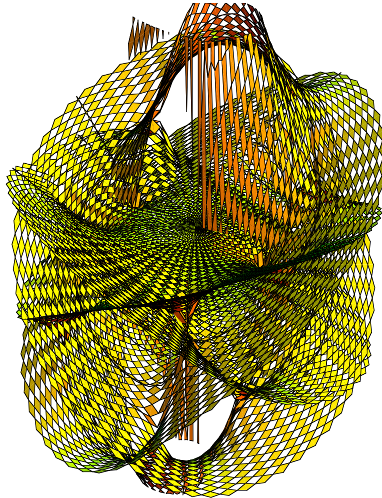
Now, let us use the function `RiemannSurface` to generate a few Riemann surfaces of algebraic functions. To better "see inside" the surface, we will replace all rectangles with "diamonds".

```
In[28]:= makeDiamond[Polygon[l_]] := Polygon[Apply[Plus,
                  Partition[Append[l, First[l]], 2, 1], {1}]/2]
```

The first example is a quartic polynomial in *w*. We see four sheets.

© 2004, 2005 Springer Science+Business Media, Inc.

In[29]:=
```
pExample = -1 + 2 w^2 + 5 w^3 + 3 w^4 z + 3 z^2 + 8 z^5;

Show[Graphics3D[{EdgeForm[Thickness[0.002]],
      SurfaceColor[Hue[0.12], Hue[0.22], 2.3],
      RiemannSurface[pExample, {w, z}, Im, {60, 30}]} /.
                              p_Polygon :> makeDiamond[p]],
    PlotRange -> {-3, 3}, BoxRatios -> {1, 1, 1.3},
    Boxed -> False]
```

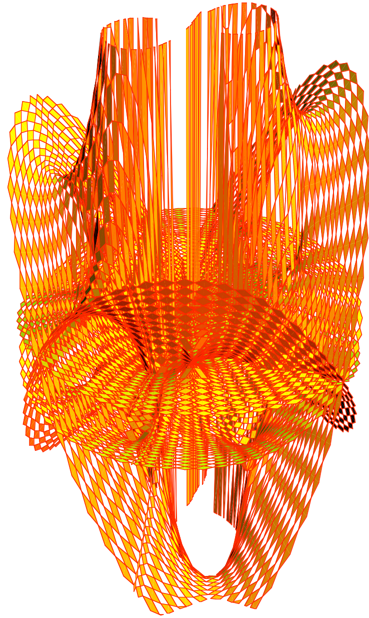


Out[30]= - Graphics3D -

The next example is also a quartic polynomial in *w*.

© 2004, 2005 Springer Science+Business Media, Inc.

In[31]:=
```
pExample = -1 - 4 w^3 - 3 z - w^4 z^2 - 4 z^3 + 3 w^2 z^5;

Show[Graphics3D[{EdgeForm[{Hue[0], Thickness[0.002]}],
      SurfaceColor[Hue[0.12], Hue[0.12], 2.2],
      RiemannSurface[pExample, {w, z}, Im, {60, 30}]} /.
                          p_Polygon :> makeDiamond[p]],
     PlotRange -> {-4, 6}, BoxRatios -> {1, 1, 1.6},
     Boxed -> False]
```

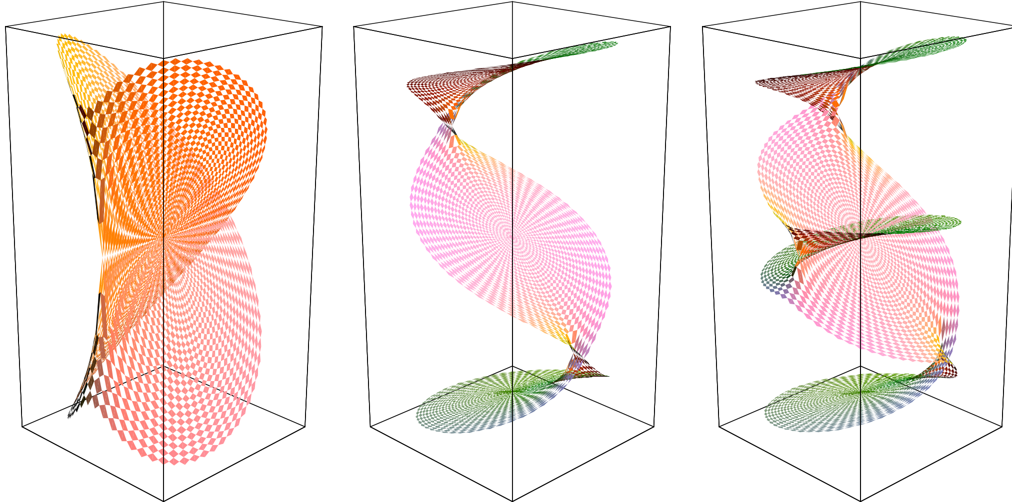


Out[32]= - Graphics3D -

Before the next degree seven polynomial, let us display the Riemann surfaces for the simple polynomials $w^2 - w - z = 0$, $w^3 - w - z = 0$, and $w^4 - w - z = 0$. We clearly see two, three, and four sheets in the pictures.

© 2004, 2005 Springer Science+Business Media, Inc.

In[33]:=
```
Show[GraphicsArray[Table[
Show[Graphics3D[{EdgeForm[],
     SurfaceColor[Hue[0.12], Hue[0.8], 2.6],
     RiemannSurface[w^k + w + z, {w, z}, Im, {60, 30}]}] /.
                              p_Polygon :> makeDiamond[p],
     PlotRange -> All, BoxRatios -> {1, 1, 2},
     DisplayFunction -> Identity, ViewPoint -> {2, 2, 1}],
               {k, 2, 4}]]]
```

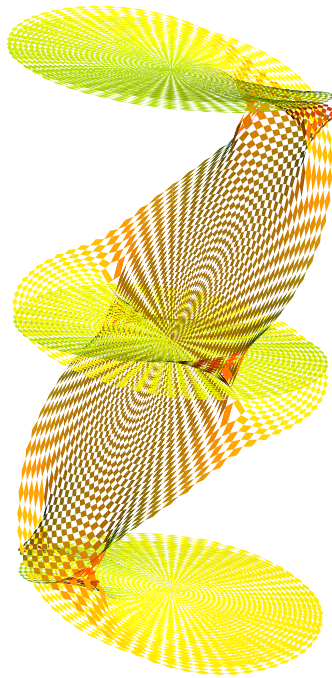


Out[33]= - GraphicsArray -

Here is the simplest form of a nontrivial septic: $w^7 - w - z = 0$. The seven sheets are nicely visible when viewed from the side.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[34]:= pExample = w^7 - w - z;

Show[Graphics3D[{EdgeForm[],
      SurfaceColor[Hue[0.1], Hue[0.1], 2.01],
      RiemannSurface[pExample, {w, z}, Im, {60, 30}]} /.
                           p_Polygon :> makeDiamond[p]],
    PlotRange -> Automatic, BoxRatios -> {1, 1, 2},
   ViewPoint -> {-4, 2, 2}, Boxed -> False]
```

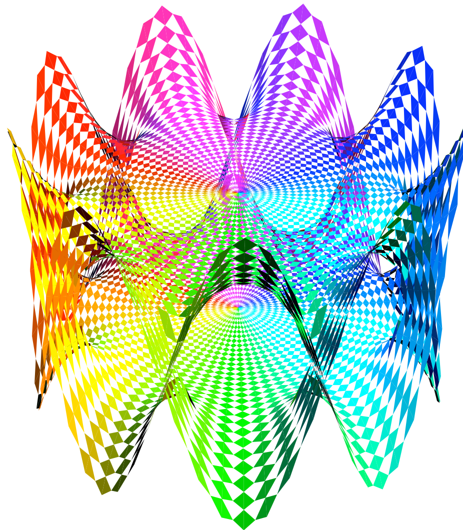


```
Out[35]= - Graphics3D -
```

Here is a quadratic with nine branch points (also called a hyperelliptic curve). We color the polygons according to their azimuthal position.

© 2004, 2005 Springer Science+Business Media, Inc.

In[36]:=
```
pExample = w^2 - z^9 + 1;

col[Polygon[l_]] :=
SurfaceColor[Hue[#], Hue[#], 2]&[
        (Pi + ArcTan @@ Take[Plus @@ l/4, 2])/(2 Pi)];

Show[Graphics3D[{EdgeForm[],
       SurfaceColor[Hue[0.3], Hue[0.25], 0.5],
       RiemannSurface[pExample, {w, z}, Im, {60, 30}]} /.
                         p_Polygon :> {col[p], makeDiamond[p]}],
     PlotRange -> All, BoxRatios -> {1, 1, 0.8}, Boxed -> False]
```

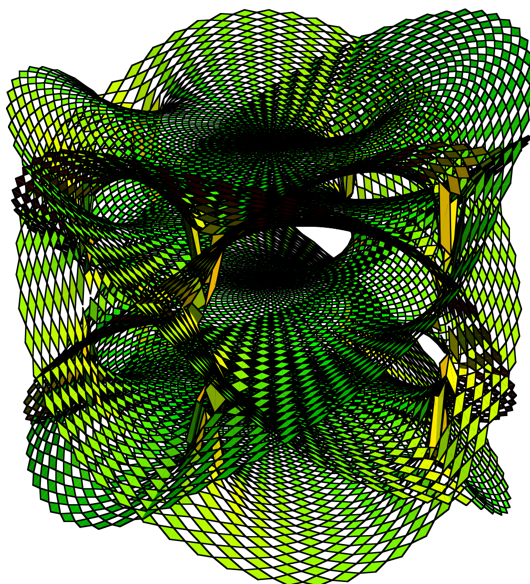


Out[38]= - Graphics3D -

The next picture shows a quintic.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[39]:= pExample = w^5 - 1 + z^2 - z^3 + z^4 - z^5;

         Show[Graphics3D[{EdgeForm[Thickness[0.002]],
               SurfaceColor[Hue[0.3], Hue[0.1], 2.4],
               RiemannSurface[pExample, {w, z}, Im, {60, 30}]} /.
                                    p_Polygon :> makeDiamond[p]],
             PlotRange -> All, BoxRatios -> {1, 1, 1},
             ViewPoint -> {2, 2, 1}, Boxed -> False]
```

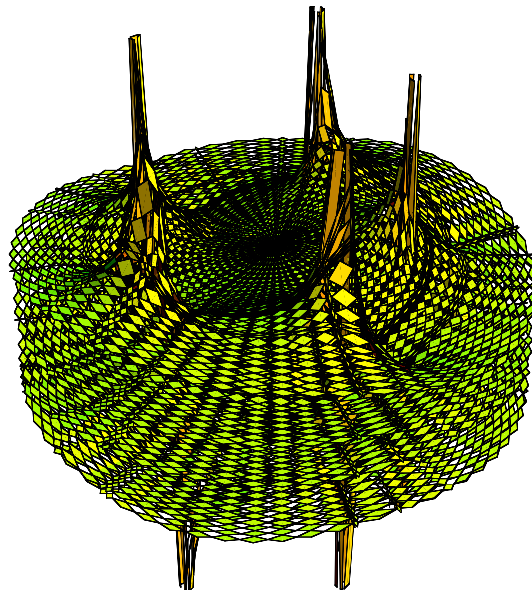


```
Out[40]= - Graphics3D -
```

The last picture also shows a quintic, but a much more complicated one. This time, we also have poles.

© 2004, 2005 Springer Science+Business Media, Inc.

```
In[41]:= pExample = 4 - 2 w^4 - 4 w^5 + 5 z + 2 w^4 z + 2 w^5 z - z^2 -
                    4 w z^2 - w^2 z^2 - z^3 + 4 w^2 z^3 +  4 w^3 z^3 +
                    2 z^4 - 4 w z^4 - 4 w^2 z^4 + 4 w^4 z^4 + 3 z^5 +
                    2 w^4 z^5 - 5 w^5 z^5;

         Show[Graphics3D[{EdgeForm[Thickness[0.002]],
                SurfaceColor[Hue[0.2], Hue[0.2], 2.3],
                 RiemannSurface[pExample, {w, z}, Im, {60, 30}]} /.
                                     p_Polygon :> makeDiamond[p]],
             PlotRange -> {-3, 3}, BoxRatios -> {1, 1, 1}, Boxed -> False]
```



```
Out[42]= - Graphics3D -
```

The interested reader can continue to visualize many more Riemann surfaces. For a larger selection of surfaces (including compactified versions over the Riemann sphere) produced along the procedure outlined above, see [1803✱]. For Riemann surfaces of nonalgebraic functions, see [1804✱], [1805✱], and [1810✱]. For some pictures of physical models of Riemann surfaces, see [534✱], [611✱], and [439✱].

```
In[43]:= Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

◦ **Session Summary:** (Evaluated with *Mathematica* 5.1)

Inputs evaluated:	42
CPU time used:	253 s
Wall clock time elapsed:	263 s
Max. memory used:	117 MB
Variables in use:	137

© 2004, 2005 Springer Science+Business Media, Inc.

References

✱9 S. S. Abhyankar, T. Moh. *J. reine angew. Math.* 260, 47 (1973).

✱10 S. S. Abhyankar, T. Moh. *J. reine angew. Math.* 261, 29 (1973).

✱28 L. V. Ahlfors, L. Sario. *Riemann Surfaces*, Princeton University Press, Princeton, 1960. *BookLink (3)*

✱77 D. A. Aruliah, R. M. Corless in J. Gutierrez (ed.). *ISSAC 04*, ACM Press, New York, 2004. *DOI-Link*

✱153 H. Behnke, F. Sommer. *Theorie der analytischen Funktionen einer komplexen Ver änderlichen*, Springer-Verlag, Berlin, 1962. *BookLink*

✱154 J. W. Belcher, S. Olbert. *Am. J. Phys.* 71, 220 (2003). *DOI-Link*

✱163 E. D. Belokolos, A. I. Bobenko, V. Z. Enol'skii, A. R. Its, V. B. Matveev. *Algebro-Geometric Approach to Nonlinear Integrable Equations*, Springer-Verlag, Berlin, 1994. *BookLink*

✱206 R. A. Beth. *J. Appl. Phys.* 37, 2568 (1966).

✱264 J. B. Bost in M. Waldschmidt, P. Moussa, J.-M. Luck, C. Itzykson (eds.). *From Number Theory to Physics*, Springer-Verlag, Berlin, 1992. *BookLink*

✱265 K. Boström. *arXiv:quant-ph*/0005024 (2000). *Get Preprint*

✱271 C. L. Bouton, M. Bocher. *Ann. Math.* 12, 1 (1898).

✱297 A. Brill. *Vorlesungen über ebene algebraische Kurven und algebraische Funktionen*, Vieweg, Braunschweig, 1925.

✱304 R. W. Brown, S. M. Shvartsman. *Phys. Rev. Lett.* 83, 1946 (1999). *DOI-Link*

✱311 H. Buchholz. *Elektrische und magnetische Potentialfelder*, Springer-Verlag, Berlin, 1957. *BookLink*

✱342 J. Cano in D. L. Tê, K. Saito, B. Teissier (eds.). *Singularity Theory*, World Scientific, Singapore, 1995. *BookLink*

✱344 J. Cantarella, D. DeTurck, H. Gluck. *Am. Math. Monthly* 109, 409 (2002).

✱355 J. R. Cary, R. G. Littlejohn. *Ann. Phys.* 151, 1 (1983). *DOI-Link*

✱356 E. Casas-Alvero. *Singularities of Plane Curves*, Cambridge University Press, Cambridge, 2000. *BookLink*

✱394 H. Cheng, L. Reemgard, V. Rokhlin. *J. Comput. Phys.* 155, 468 (1999). *DOI-Link*

✱408 D. V. Chudnovsky, G. V. Chudnovsky. *J. Complexity* 3, 1 (1987).

✱439 R. M. Corless, D. J. Jeffrey. *SIGSAM Bull.* 32, 11 (1998). *DOI-Link*

✱532 D. Duval. *Compositio Math.* 70, 119 (1989).

✱534 W. Dyck (ed.). *Katalog mathematischer und mathematisch physikalischer Modelle, Apparate und Instrumente*, Georg Olms Verlag, Heidelberg, 1994. *BookLink*

© 2004, 2005 Springer Science+Business Media, Inc.

★587 H. Fangohr, A. R. Price, S. J. Cox, P. A. J. de Groot, G. J. Daniell. *arXiv:physics*/0004013 (2000). *Get Preprint*

★589 H. M. Farkas, I. Kra. *Riemann Surface*, Springer-Verlag, New York, 1992. *BookLink (3)*

★591 J.-M. Farto. *J. Pure Appl. Alg.* 108, 203 (1996).

★611 G. Fischer (ed.). *Mathematical Models: Commentary*, Vieweg, Braunschweig, 1986. *BookLink*

★612 G. Fischer. *Ebene algebraische Kurven*, Vieweg, Wiesbaden, 1994. *BookLink*

★614 P. Flajolet, R. Sedgewick. *Preprint INRIA* n 4103 (2001). http://www.inria.fr/RRRT/RR-4103.html

★641 T. E. Freeman. *Am. J. Phys.* 63, 273 (1995). *DOI-Link*

★650 S. Fukao, T. Tsuda. *J. Plasma Phys.* 9, 409 (1973).

★734 C. Grama, N. Grama, I. Zamfirescu. *Phys. Rev.* A 61, 032716 (2000). *DOI-Link*

★743 M. I. Grivich, D. P. Jackson. *Am. J. Phys.* 68, 469 (2000). *DOI-Link*

★788 S. Handa, H. Kashiwagi, T. Takada. *J. Visual. Comput. Anim.* 12, 167 (2001). *DOI-Link*

★835 F. Herrmann, H. Hauptmann, M. Suleder. *Am. J. Phys.* 68, 171 (2000). *DOI-Link*

★842 E. Hille. *Analytic Function Theory* v.II, Chelsea, New York, 1973. *BookLink (2)*

★902 M. S. Janaki, G. Gash. *J. Phys.* A 20, 3679 (1987). *DOI-Link*

★919 G. A. Jones, D. Singerman. *Complex Functions—An Algebraic and Geometric ViewPoint*, Cambridge University Press, Cambridge, 1987. *BookLink (2)*

★930 H. W. Jung. *Einführung in die Theorie der algebraischen Funktionen einer Veränderlichen*, de Gruyter, Berlin, 1923. *BookLink*

★1020 H. E. Knoepfel. *Magnetic Fields*, Wiley, New York, 2000. *BookLink*

★1179 F. J. López, F. Martín. *Publ. Mat.* 43, 341 (1999).

★1228 L. Markus, K. R. Meyer. *Am. J. Math.* 102, 25 (1980).

★1229 A. I. Markushevich. *The Theory of Functions of a Complex Variable*, Prentice-Hall, Englewood Cliffs, 1965. *BookLink*

★1238 A. Marshakov. *Seiberg-Witten Theory and Integrable Systems*, World Scientific, Singapore, 1999. *BookLink (2)*

★1245 J. H. Mathews, R. W. Howell. *Complex Analysis for Mathematics and Engineering*, Jones and Bartlett, Boston, 1997. *BookLink (2)*

★1249 K. Maurin. *The Riemann Legacy* Kluwer, Dordrecht, 1997. *BookLink*

★1252 J. McDonald. *J. Pure Appl. Math.* 104, 213 (1995).

★1253 J. McDonald. *Disc. Comput. Geom.* 27, 501 (2002). *DOI-Link*

★1257 H. McKean, V. Moll. *Elliptic Curves*, Cambridge University Press, Cambridge, 1997. *BookLink (2)*

★1269 M. Menzel. *Ph.D. thesis*, Kaiserslautern, 1997.

© 2004, 2005 Springer Science+Business Media, Inc.

✱1288 A. M. Mitichkina, A. A. Ryabenko. *Program.* 27, 10 (2001).

✱1296 A. F. Monna. *Dirichlet's Principle*, Ososthoek, Scheltema & Holkema, Utrecht, 1975. *BookLink (3)*

✱1349 J. Neuberger, J. Gruenebaum. *Eur. J. Phys.* 3, 22 (1982). *DOI-Link*

✱1385 M. H. Oliveira, J. A. Miranda. *Eur. J. Phys.* 22, 31 (2001). *DOI-Link*

✱1389 F. Ollendorff. *Potentialfelder in der Elektrotechnik*, Springer-Verlag, Berlin, 1932.

✱1390 F. Ollendorff. *Berechnung magnetischer Felder*, Springer-Verlag, Wien, 1952. *BookLink*

✱1397 E. T. Ong, K. M. Lim, K. H. Lee, H. P. Lee. *J. Comput. Phys.* 192, 244 (2003). *DOI-Link*

✱1402 W. F. Osgood. *Lehrbuch der Funktionentheorie*, Teubner, Leipzig, 1923. *BookLink*

✱1412 C. Pabst, R. Naulin. *Rev. Columb. Mater.* 30, 25 (1996).

✱1424 R. L. Parker. *Proc. R. Soc. Lond.* 291, 60 (1966).

✱1437 J. M. Pérez–Jordá, W. Yang. *Chem. Phys. Lett.* 247, 484 (1995). *DOI-Link*

✱1453 E. Petrisor. *Physica* D 112, 319 (1998). *DOI-Link*

✱1459 A. Pfluger. *Theorie der Riemannschen Fläche*, Springer-Verlag, Berlin, 1957. *BookLink*

✱1462 E. Piña, T. Ortiz. *J. Phys.* A 21, 1293 (1988). *DOI-Link*

✱1516 A. F. Rañada, J. L. Trueba. *Phys. Lett.* A 232, 25 (1997). *DOI-Link*

✱1519 H. E. Rauch, A. Lebowitz. *Elliptic Functions, Theta Functions and Riemann Surfaces*, Williams & Wilkins, Baltimore, 1973. *BookLink*

✱1532 E. Reyssat. *Quelques Aspects des Surfaces de Riemann*, Birkhäuser, Basel, 1989. *BookLink*

✱1546 J. F. Ritt. *Trans. Am. Math. Soc.* 24, 21 (1922).

✱1573 K. Ruedenberg, J.-Q. Sun. *J. Chem. Phys.* 100, 5836 (1994). *DOI-Link*

✱1591 A. Salat. *Z. Naturf.* 40 a, 959 (1985).

✱1662 K. Shihara, T. Sasaki. *Jpn. J. Industr. Appl. Math.* 13, 107 (1996).

✱1680 J. Slepian. *Am. J. Phys.* 19, 87 (1951).

✱1686 W. R. Smythe. *Static and Dynamic Electricity*, McGraw-Hill, New York, 1968. *BookLink (3)*

✱1766 B. Teissier in D. L. Tê, K. Saito, B. Teissier (eds.). *Singularity Theory*, World Scientific, Singapore, 1995. *BookLink*

✱1793 G. F. Torres del Castillo. *J. Math. Phys.* 36, 3413 (1995). *DOI-Link*

✱1803 M. Trott. *Mathematica Edu. Res.* 6, n4, 15 (1997).

✱1804 M. Trott. *The Mathematica Journal* 4, 465 (2000).

✱1805 M. Trott. *The Mathematica Journal* 8, 409 (2002).

© 2004, 2005 Springer Science+Business Media, Inc.

✱1807 M. Trott. *The Mathematica GuideBook for Graphics*, Springer-Verlag, New York, 2004. *BookLink*

✱1808 M. Trott. *The Mathematica GuideBook for Symbolics*, Springer-Verlag, New York, 2005. *BookLink*

✱1810 M. Trott. *The Mathematica Journal* 8, 50 (2001).

✱1828 S. Ulam in D. Mauldin (ed.). *The Scottish Book*, Birkhäuser, Basel, 1981. *BookLink*

✱1851 V. M. Vasyliunas. *J. Geophys. Res.* 77, 6271 (1972).

✱1883 R. J. Walker. *Algebraic Curves*, Princeton University Press, Princeton, 1950. *BookLink*

✱1890 E. Weber. *Electromagnetic Fields* v.1, Wiley, London, 1950. *BookLink*

✱1904 D. H. Werner. *IEEE Trans. Antennas Prop.* 44, 157 (1996).

✱1905 D. H. Werner. *IEEE Trans. Antennas Prop.* 47, 1351 (1999).

✱1906 D. H. Werner in D. H. Werner, R. Mittra (eds.). *Frontiers in Electromagnetics*, IEEE Press, New York, 2000. *BookLink*

✱1910 H. Weyl. *Die Idee der Riemannschen Fläche*, Teubner, Stuttgart, 1955. *BookLink*

✱1929 A. Wolf, S. J. van Hook, E. R. Weeks. *Am. J. Phys.* 64, 714 (1996). *DOI-Link*

✱1984 M. Zöckler, D. Stalling, H.-C. Hege. *Proc. Visualization '96*, IEEE, New York, 1996. *BookLink*

✱1986 A. Zvonkin in D. Krob, A. A. Mikhalev, A. V. Mikhalev (eds.). *Formal Power Series and Algebraic Combinatorics*, Springer-Verlag, Berlin, 2000. *BookLink*

© 2004, 2005 Springer Science+Business Media, Inc.