8

Farben, Farbskalen und Heatmaps

Bei der Datenvisualisierung können Farben verschiedene Zwecke erfüllen: So können sie einfach dazu dienen, den Graphen optisch ansprechender zu gestalten, aber auch bestimmte Aspekte der Daten betonen oder sogar die Hauptträger der Informationen sein. In diesem Kapitel schauen wir uns an, wie Farben in D3 dargestellt werden, und besprechen dann verschiedene Farbschemas und ihre Verwendung zur Informationsvermittlung. Am Ende des Kapitels folgt eine Beschreibung von Falschfarbendiagrammen, in denen die Daten ausschließlich mithilfe von Farben dargestellt werden.

Farben und Farbräume

Eine einzelne Farbe lässt sich in D3 ganz einfach angeben. Dazu müssen Sie lediglich mit der CSS3-Syntax (siehe Anhang B) einen String mit einem vordefinierten Farbnamen, den RGB-Komponenten (Rot, Grün, Blau) oder den HSL-Komponenten (Hue, Saturation, Lightness, also Farbton, Sättigung, Helligkeit) übergeben. Wenn Sie aber Farben programmgesteuert bearbeiten wollen, ist das Stringformat dazu nicht sehr gut geeignet. Für diese Zwecke können Sie mit einer der Factory-

Funktionen aus Tabelle 8–1 ein `color`-Objekt erstellen und dann überall dort verwenden, wo eine Farbangabe erwartet wird. Seine Funktion `toString()` wird automatisch aufgerufen und gibt eine Darstellung der Farbe im CSS3-Format zurück.

Die API von `color`-Objekten ist allerdings stark eingeschränkt. Hauptsächlich dienen sie als Container für ihre Kanalinformationen. Jedes `color`-Objekt stellt seine drei Komponenten als entsprechend benannte Eigenschaften zur Verfügung, eine für jeden Kanal. Außerdem verfügt jedes Objekt über die Eigenschaft `opacity` für den Alphakanal.

Die Funktionen aus Tabelle 8–1 können die folgenden Argumente entgegennehmen und geben ein `color`-Objekt aus dem angeforderten Farbraum zurück:

- Ein CSS3-Farbstring
- Ein anderes `color`-Objekt (zur Umwandlung in einen anderen Farbraum)
- Eine Menge von drei Komponenten (bzw. vier bei Angabe eines Opazitätswerts)

Eine Ausnahme bildet die generische Factory `d3.color()`, die einen CSS3-String oder ein anderes `color`-Objekt entgegennimmt und je nach Eingabe ein RGB- oder HSL-`color`-Objekt zurückgibt.

Funktion	Komponenten[a]	Bemerkungen
`d3.rgb()`	`r`, `g`, `b`	$0 \le r, g, b \le 255$ (streng)
`d3.hsl()`	`h`, `s`, `l`	• h: Beliebiger (positiver oder negativer) Wert; der Farbton wird als modulo 360 aufgefasst. • $0 \le s, l \le 1$ (streng)
`d3.lab()`	`l`, `a`, `b`	• $0 \le l \le 100$ (streng) • $-100 \le a, b \le 100$ (genähert)
`d3.hcl()`	`h`, `c`, `l`	• h: Beliebiger (positiver oder negativer) Wert; der Farbton wird als modulo 360 aufgefasst. • $-125 \le c \le 125$ (genähert) • $0 \le l \le 100$ (genähert)
`d3.color()`	`r`, `g`, `b` oder `h`, `s`, `l`	Die Eingabe muss ein CSS3-String oder ein anderes Objekt sein. Die Ausgabe ist ein RGB-Objekt, es sei denn, die Eingabe liegt im HSL-Format vor.

[a] Alle `color`-Objekte verfügen auch über eine `opacity`-Komponente mit Werten von 0 (transparent) bis 1 (opak).

Tab. 8–1 *Factory-Funktionen zum Erstellen von* `color`*-Objekten in verschiedenen Farbräumen mit den Farbkomponenten des zurückgegebenen Objekts und den zulässigen Bereichen*

Die zulässigen Parameterbereiche unterscheiden sich im Allgemeinen je nach Farbraum. Bei offensichtlich unzulässigen Eingaben (etwa einer negativen RGB-Komponente) geben die Funktionen aus Tabelle 8–1 `null` zurück, aber auch wenn der Rückgabewert nicht `null` ist, ist nicht garantiert, dass die zurückgegebene Farbe auch *darstellbar* ist. Um das herauszufinden, können Sie die Funktion `displayable()` auf die zurückgegebene Farbinstanz anwenden. Kann eine Farbe nicht angezeigt werden, dann ersetzt ihre Methode `toString()` diese durch eine »passende« darstellbare Farbe (z. B. durch Weiß oder Schwarz, wenn die erforderliche Farbe zu hell oder zu dunkel ist).

Farbräume

Im Allgemeinen sind zur Angabe einer Farbe drei Koordinaten erforderlich (ohne Berücksichtigung der Transparenz). Im RGB-Modell wird der Farbraum als Würfel dargestellt, der von den drei Komponentenachsen im rechten Winkel aufgespannt wird. Der RGB-Farbraum kommt der technischen Umsetzung in der Hardware am nächsten, ist aber dafür berüchtigt, alles andere als intuitiv verständlich zu sein. (Oder können Sie auf Anhieb sagen, wie `#b8860b` aussieht?)

Im HSL-Modell wird der RGB-Würfel zu einem Doppelkegel verformt, dessen Achse von Schwarz nach Weiß entlang der Raumdiagonale des ursprünglichen Würfels verläuft. Die drei Komponenten werden jetzt als Zylinderkoordinaten aufgefasst. Der Farbton gibt den Winkel um die Zylinderachse an, die Helligkeit die Position entlang der Achse (von Schwarz am Boden bis zu Weiß an der Spitze) und die Sättigung den radialen Abstand von der Achse (wobei Grautöne die Achse bilden und völlig gesättigte Farben die Kegeloberfläche). Völlig gesättigte Farben mit maximaler Helligkeit sind auf halber Höhe am »Bauch« des Doppelkegels zu finden.

Im Vergleich zu RGB- lassen sich HSL-Koordinaten etwas intuitiver deuten, allerdings leiden beide Darstellungsarten unter demselben Mangel: Ihre Farbkomponenten geben die Intensitäten des Renderinggeräts wieder, ohne die ungleichmäßige menschliche *Farbwahrnehmung* zu berücksichtigen. Das führt zu einer Reihe von Unstimmigkeiten. Betrachten Sie beispielsweise die drei Farben mit den CSS3-Farbnamen DarkKhaki (`#bdb76b`), Gold (`#ffd700`) und Yellow (`#ffff00`). Von diesen dreien hat DarkKhaki im HSL-Raum die größte Helligkeit, obwohl sie dunkler erscheint als die anderen. Gelb und Gold haben gleiche Helligkeitswerte, wirken aber völlig unterschiedlich. Es lassen sich noch leicht weitere Beispiele finden, in denen die Helligkeitswerte nicht mit der Wahrnehmung übereinstimmen.

➥

Farbräume, die auf den *wahrgenommenen* Intensitäten basieren, sollen solche Unstimmigkeiten zu vermeiden helfen. Zwei davon sind CIELAB (oder LAB) und HCL (Hue, Chroma, Luminance). Im LAB-Farbraum spannen die Koordinatenachsen ein Parallelepiped auf, wobei `l` die Helligkeit misst und die Koordinaten `a` und `b` die Position entlang von zwei Achsen angeben, von denen die eine von Rot nach Grün und die andere von Blau nach Gelb verläuft. (Im Standardfarbkreis stehen diese beiden Achsen angenähert senkrecht aufeinander.) Der HCL-Farbraum wandelt dieselben Informationen in Zylinderkoordinaten ähnlich denen des HSL-Raums um, wobei der Chroma-Wert angibt, wie farbig eine Farbe erscheint, und die Luminanz ein Maß für die Leuchtkraft ist. Der HCL-Raum lässt sich ebenso intuitiv nutzen wie der vertraute HSL-Raum, allerdings funktionieren Farbvergleiche mit HSL-Komponenten besser.

Ein Problem bei der Verwendung von Farbkomponenten, die sich auf die menschliche Wahrnehmung statt auf die technischen Aspekte stützen, besteht darin, dass es damit ziemlich einfach ist, Farben zu konstruieren, die sich mit regulärer Hardware nicht anzeigen lassen. Anders als bei RGB und HSL sind die Bereiche der Komponenten nicht auf einen festen Satz von Werten beschränkt. Daher muss sorgfältig darauf geachtet werden, dass die resultierende Farbe auch tatsächlich darstellbar ist.

Farbschemas

Im Lieferumfang von D3 ist eine ziemlich große Menge von Farbschemas zur Verwendung mit Skalierungsobjekten enthalten. Da es schwierig sein kann, sich in diesen vielen Schemas zurechtzufinden, gebe ich Ihnen hier eine Übersicht über die verfügbaren Farbschemas.[1]

Kartografieschemas

Die folgenden Schemas basieren auf solchen, die ursprünglich für Landkarten entwickelt wurden, um politische und andere thematische Informationen darzustellen. Sie eignen sich am besten für *Flächen*, aber weniger gut für Linien oder für Formen mit winzigen Details. Da sie entworfen wurden, um ein angenehmes Erscheinungsbild hervorzurufen, sind sie auch oft eine gute Wahl, um die Wahrnehmung von Informationen, die bereits durch andere Mittel dargestellt werden, zu *verstärken*.

1 Farbige Darstellungen finden Sie auf *https://github.com/d3/d3-scale-chromatic*.

Kategorieschemas
: Bei diesen neun nicht semantischen Kategorieschemas mit jeweils acht bis zwölf einzelnen Farben weisen benachbarte Farben gewöhnlich einen starken Kontrast auf. Die einzige Ausnahme davon bildet das Schema `d3.schemePaired` aus sechs Paaren, die jeweils aus einer hellen und einer dunklen Nuance mit vergleichbarem Farbton bestehen (Beispiele siehe Abb. 5–7 und Abb. 9–3).

Divergierende Schemas
: Es gibt neun divergierende Schemas jeweils mit dunklen und gesättigten Farben unterschiedlichen Farbtons an den Enden und einem Verlauf über helle und blasse Versionen von Grau, Weiß oder Gelb in der Mitte. Mit diesen Farbschemas können Sie Abweichungen von einem Grundwert in beide Richtungen darstellen (siehe Abb. 8–1).

Sequenzielle Schemas mit einem Farbton
: Diese sechs Schemas verlaufen jeweils von einem hellen und blassen Weiß oder Grau zu einer dunklen und gesättigten Farbe, wobei nur ein Farbton verwendet wird. (Eine Beispielanwendung finden Sie in Abb. 9–2.)

Sequenzielle Schemas mit zwei oder mehr Farbtönen
: Diese zwölf Schemas verlaufen jeweils von einem hellen und blassen Weiß oder Grau über ein oder zwei einander »ähnliche« Zwischenfarben (wie Blau und Grün oder Gelb, Grün und Blau) zu einer dunklen und gesättigten Farbe.

Die sequenziellen Schemas gibt es jeweils in zwei Varianten:

- Eine kontinuierliche Variante (als Interpolierer) zur Verwendung mit `d3.scale.Sequential()`
- Eine diskrete Variante (als Array aus diskreten Farbwerten) zur Verwendung mit `d3.scaleOrdinal()` oder den Binning-Skalierungen (wie `d3.scaleThreshold()`)[2]

Von den meisten der diskreten Schemas gibt es mehrere Versionen mit jeweils einer anderen Anzahl (zwischen drei und zehn) gleichmäßig verteilter diskreter Elemente. Die genauen Zahlen für die einzelnen Elemente können Sie der D3-Dokumentation auf *https://github.com/d3/d3/blob/master/API.md* entnehmen.

2 Ein diskretes Farbschema zu verwenden, anstatt einfach den entsprechenden Interpolierer für diskrete Werte auszuwerten, ermöglicht es, auch *nicht numerische* Kategoriewerte auf Farben abzubilden.

Falschfarbenschemas

Die folgenden Farbschemas sind für Falschfarbendiagramme und Heatmaps gedacht. Sie sind nur als kontinuierliche Interpolierer verfügbar (also nicht in einer diskreten Version).

Sequenzielle Schemas mit mehreren Farbtönen
: Es gibt fünf Schemas mit mehreren Farbtönen, die alle von einer sehr dunklen (fast schwarzen) Farbe zu Gelb oder Weiß verlaufen. Bei diesen Schemas werden Sättigung und Helligkeit gleichzeitig geändert (was für die Praxis möglicherweise nicht so sinnvoll ist, wie es sich in der Theorie anhört).

Zyklische oder Regenbogenschemas
: Die beiden Reigenbogenschemas sind in der Wahrnehmung einheitlicher als der Standardfarbkreis im HSL-Raum. Bei einem von ihnen stehen die beiden Hälften auch getrennt zur Verfügung, wobei die eine über die warme Seite des Farbkreises verläuft (Rot) und die andere über die kalte (Blau, Grün).

Besonders bemerkenswert ist das »Sinusbogen-Schema« `d3.interpolateSinebow`. Im herkömmlichen Regenbogenschema variieren die RGB-Komponenten auf reguläre, stückweise lineare Weise im Farbtonbereich von 0 bis 360. Beim Sinusbogen wird dieses stückweise lineare Verhalten durch drei Sinusfunktionen ersetzt, deren Phasen jeweils um 120° gegeneinander verschoben sind. Das ergibt einen helleren, aber in der Wahrnehmung sehr viel einheitlicheren Regenbogen (siehe Abb. 8–1). Dieses Schema wird wegen seines Erscheinungsbilds und seiner konzeptionellen Einfachheit sehr empfohlen.[3]

Palettengestaltung

Die Gestaltung von Farbschemas oder Farbpaletten für die Datenvisualisierung ist ein schwieriges Thema. Beachten Sie dabei die folgenden Überlegungen und Empfehlungen:[4]

- Überlegen Sie, ob die Farbe nur zur Verstärkung der Wahrnehmung von Informationen dient, die auch auf andere Weise aus dem Graphen abgelesen werden können, oder ob sie der Hauptinformationsträger ist. Beispielsweise hebt die Tönung von topografischen Karten lediglich die Höheninformationen hervor, die auch schon durch die Höhenlinien zur Verfügung stehen, wohingegen alle wichtigen Informationen in einem echten Falschfarbendiagramm wie dem aus Abbildung 8–2 ausschließlich durch Farben angezeigt werden.

➞

3 Siehe auch *http://basecase.org/env/on-rainbows*.

- Überlegen Sie, ob die Daten eine inhärente Sortierung aufweisen. Wenn ja, müssen Sie dafür sorgen, dass diese Sortierung auch von dem Farbschema unterstützt wird. Beispielsweise verbinden wir vereinbarungsgemäß und intuitiv Blau mit niedrigeren und Rot mit höheren Werten, was in einem sorgfältig ausgearbeiteten Graphen berücksichtigt werden sollte. (In dieser Hinsicht sind die mitgelieferten Farbschemas inkonsistent!) Manchmal kann die wahrgenommene Sortierreihenfolge von Farben unerwünscht sein (etwa bei bestimmten Arten von Kategoriedaten).
- Beachten Sie die Bedeutungen, die wir gewöhnlich mit bestimmten Farben verknüpfen (z. B. bei den Ampelfarben Rot, Gelb und Grün).
- Vermeiden Sie sehr dunkle und sehr helle Farben, da sie Einzelheiten unterdrücken können (»Weiß auf Weiß«, »Schwarz auf Schwarz«). Vermeiden Sie auf der anderen Seite aber auch schreiende Farben. Finden Sie einen Ausgleich zwischen grell und nicht wahrnehmbar. (Farbräume, die in der Wahrnehmung einheitlich sind, wie HCL, können Ihnen dabei behilflich sein.)
- Verwenden Sie starke Farbverläufe, wo es sinnvoll ist. Oft sind relativ kleine Änderungen in einigen Teilen des Definitionsbereichs viel interessanter als anderswo. Wenn Sie starke optische Wechsel im Farbschema dort platzieren, wo die Änderungen in den Daten von größter Bedeutung sind, können Sie die Wahrnehmung der wichtigen Einzelheiten verstärken.
- Im gleichen Sinne können Sie auch nicht kontinuierliche Änderungen verwenden, um eine Schwelle in den Daten anzuzeigen. Beispielsweise werden in topografischen Karten kontinuierliche Farbverläufe eingesetzt, Küstenlinien aber durch einen scharfen Übergang gekennzeichnet.
- Immer wenn Farbe verwendet wird, um Informationen zu vermitteln (also nicht nur aus rein ästhetischen Gründen), sollte das Diagramm über eine Farblegende verfügen, die die Zuordnung der Farben zu Werten aufschlüsselt. Wie offensichtlich Ihr Farbschema auch erscheinen mag, so ist es doch unmöglich, die genaue Zuordnung allein aus dem Graphen zu entnehmen.
- Denken Sie daran, dass etwa 8 % der Männer und 0,5 % der Frauen zumindest teilweise von Farbenfehlsichtigkeit betroffen sind, wobei die Rot-Grün-Sehschwäche am weitesten verbreitet ist.

Andere Farbschemas

Die schiere Anzahl und Vielseitigkeit der im Lieferumfang enthaltenen Farbschemas scheint alle Bedürfnisse abdecken zu können, aber je nach Zusammenhang und Zweck kann es sein, dass andere Schemas besser geeignet sind (siehe »*Palet-*

4 In Anhang D meines Buches *Gnuplot in Action* (Manning Publications, 2. Auflage) erfahren Sie noch mehr zu diesem Thema.

tengestaltung« auf *S. 166*). Die vorhandenen Farbschemas können Sie auch als Anregungen nutzen.[5] Es gibt keinen Grund dafür, die mitgelieferten Schemas zu verwenden, wenn sie für Ihre Zwecke unpassend erscheinen oder sich nur schwer an eine bestimmte Anwendung anpassen lassen.

Farbskalierungen

Um Farben zur Darstellung von Informationen zu nutzen, ist es oft praktisch, sie mit einem Skalierungsobjekt zu kombinieren (siehe Kapitel 7), insbesondere, um den Wertebereich des Skalierungsobjekts mit Farben zu definieren. In diesem Abschnitt lernen Sie dazu einige einfache Anwendungen und Techniken kennen.

Diskrete Farben

Diskrete Farbskalierungen können mit Binning- und mit diskreten Skalierungen realisiert werden. Die Binning-Skalierungen, die von `d3.scaleQuantize()` und `d3.scaleThreshold()` erstellt werden, eignen sich für die Wiedergabe eines kontinuierlichen Zahlenbereichs durch eine diskrete Menge von Farben. Die erste dieser beiden Skalierungen teilt den Definitionsbereich in gleich große Bins auf, während die zweite erwartet, dass der Benutzer Schwellenwerte zur Definition der Bins angibt (siehe Beispiele 4–7 und 7–3). Denken Sie daran, dass die Funktion `domain()` für Binning-Skalierungen unterschiedliche Bedeutungen hat (Einzelheiten siehe Kapitel 7):

```
var sc1 = d3.scaleQuantize().domain( [0, 1] )
    .range( [ "black", "red", "yellow", "white"] );
var sc2 = d3.scaleThreshold().domain( [ -1, 1] )
    .range( [ "blue", "white", "red" ] );
```

Diskrete Skalierungen, wie sie von `d3.scaleOrdinal()` erstellt werden, eignen sich, wenn diskrete Eingabewerte durch verschiedene Farben dargestellt werden sollen. Eine diskrete Skalierung kann eine explizite Beziehung zwischen einzelnen Werten des Definitionsbereichs und den Farben zu ihrer Darstellung herstellen:

```
var sc = d3.scaleOrdinal( [ "green", "yellow", "red"] )
    .domain( [ "good", "medium", "bad" ] );
```

5 Eine sehr große und vielseitige Sammlung von Paletten, die allerdings nicht alle für die Visualisierung im wissenschaftlichen Bereich gedacht sind, finden Sie auf *http://soliton.vm.bytemark.co.uk/pub/cpt-city/*. Besonders interessant für unsere Zwecke sind die Arbeiten von Kenneth Moreland und des Projekts Generic Mapping Tools, insbesondere die Palette GMT Haxby.

Wurde kein Definitionsbereich angegeben, weist eine diskrete Skalierung jedem neuen Symbol bei dessen Auftreten die nächste noch nicht verwendete Farbe zu (siehe Beispiele 5–6 und 9–5).

Ganz unabhängig von ihrer Implementierung durch Skalierungsobjekte müssen die Farben für diskrete Farbschemas ihrem Zweck entsprechend ausgewählt werden. Manchmal besteht dieser Zweck einfach darin, verschiedene grafische Elemente besser unterscheiden zu können, ohne dass dadurch irgendeine Bedeutung oder Reihenfolge angedeutet werden soll. Die mitgelieferten Kategorieschemas sind von dieser Art (siehe Beispiele 5–7 und 9–3). In anderen Fällen sollen die Farben tatsächlich eine Bedeutung vermitteln, wenn auch nicht unbedingt eine monotone Sortierung oder strenge Anordnung. Ein Beispiel dafür ist das Ampelschema in Rot, Gelb und Grün aus Abbildung 7–5. Schließlich werden diskrete Farbschemas manchmal auch verwendet, um eine monoton geordnete Folge von Schritten darzustellen (siehe den rechten Teil von Abb. 9–2). In letzterem Fall können Binning-Skalierungen zusammen mit den mitgelieferten sequenziellen oder divergierenden Schemas eingesetzt werden, um einen kontinuierlichen Bereich von Werten auf eine feste Menge einzelner Farben mit einer deutlichen Sortierung abzubilden.

Farbverläufe

Die Möglichkeiten von D3 zur Interpolation von Farben machen es besonders einfach, Farbverläufe zu erstellen. Die Beispiele in diesem Abschnitt zeigen Ihnen die Syntax für einige einfache, aber typische Fälle (siehe Beispiel 8–1 und Abb. 8–1).

Beispiel 8–1: Farbverläufe erstellen (siehe Abb. 8–1)

```
sc1 = d3.scaleLinear().domain( [0, 3, 10] )   ❶
    .range( ["blue", "white", "red"] );

sc2 = d3.scaleLinear().domain( [0, 5, 5, 10] )   ❷
    .range( ["white", "blue", "red", "white"] );

sc3 = d3.scaleSequential( t => "" + d3.hsl( 360*t, 1, 0.5 ) )   ❸
    .domain( [0, 10] );

sc4 = d3.scaleSequential( t => d3.interpolateSinebow(2/3-3*t/4) )   ❹
    .domain( [0, 10] );

sc5 = d3.scaleDiverging( t => d3.interpolateRdYlGn(1-t) )   ❺
    .domain( [0, 2, 10] );

sc6 = d3.scaleSequential( d3.interpolateRgbBasis(   ❻
    ["#b2d899","#ffffbf","#bf9966","#ffffff"] ) ).domain( [0,10] );
```

❶ Ein einfacher Farbverlauf von Blau über Weiß nach Rot mithilfe der Standardinterpolation des Farbraums. Beachten Sie aber die asymmetrische Lage des weißen Bandes!

❷ Ein Verlauf mit einem scharfen Übergang in der Mitte.

❸ Der viel gescholtene »HSL-Standardregenbogen«. Das Beispiel ist hier vor allem angegeben, um die Syntax zu demonstrieren (und nicht etwa, weil dies ein besonders gut geeigneter Farbverlauf wäre).

❹ Ein weit besserer Regenbogen, erstellt mit dem mitgelieferten Sinusbogen-Interpolierer. Hier wird er rückwärts durchlaufen (also von Blau zu Rot, um die übliche Bedeutung von niedrigen zu hohen Werten zu vermitteln). Der Wertebereich ist beschränkt, um zu verhindern, dass der Verlauf umlaufend weitergeführt und Farben wiederverwendet werden.

❺ Ein Beispiel für einen divergierenden Farbverlauf. Hier wird der mitgelieferte Rot/Gelb/Grün-Interpolierer verwendet, aber die Richtung geändert, sodass höhere Werte rot dargestellt werden. Der Definitionsbereich eines divergierenden Verlaufs muss *drei* numerische Werte enthalten, wobei der mittlere Wert auf die Interpoliererposition t = `0,5` abgebildet wird. Hier wird die gelbe Farbe, die die Mitte des zurückgegebenen Farbbereichs darstellt, verschoben und mit dem Wert 2 des Definitionsbereichs verknüpft.

❻ Ein Beispiel für die Art von Farbschema, die gewöhnlich in topografischen Karten verwendet wird. Im Gegensatz zu den anderen Schemas wächst die Helligkeit hier nicht monoton, sondern unterliegt einer zusätzlichen, systematischen Schwankung (Grün und Braun sind relativ dunkel, Gelb und Weiß relativ hell), was als optische Orientierungshilfe dienen kann. Dieses Schema verwendet den Interpolierer `d3.interpolateRgbBasis`, der ein Spline durch alle bereitgestellten Farben konstruiert (die als gleichmäßig verteilt angenommen werden). Der Vorteil des Spline-Interpolierers liegt nicht so sehr im glatteren Farbverlauf, sondern darin, dass es nicht erforderlich ist, die Position der Zwischenfarben mit `domain()` anzugeben.

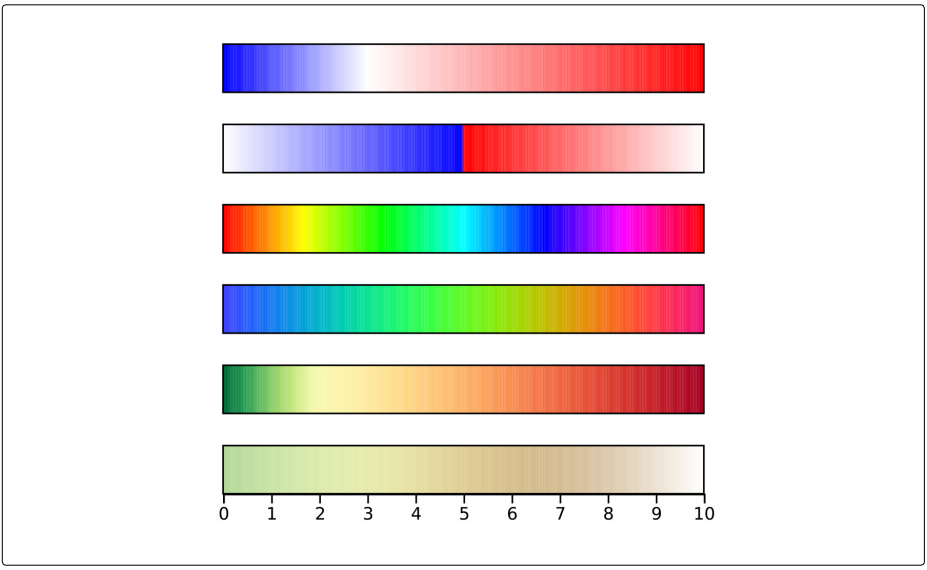

Abb. 8–1 *Farbverläufe (siehe Beispiel 8–1)*

Farblegenden

Ein Diagramm, das Farben als Informationsträger nutzt (also nicht nur, um das Erscheinungsbild und die Wahrnehmung zu verbessern), muss über eine *Legende* verfügen, die deutlich anzeigt, welche Werte welchen Farben entsprechen. Manchmal kann auch eine einfache Beschreibung in Textform ausreichen, aber eine Farblegende ist gewöhnlich viel deutlicher. D3 bringt keine Komponenten für diesen Zweck mit, aber es ist ziemlich einfach, sie selbst zu entwickeln. Die Legende aus Abbildung 8–1 wurde mit der Komponente aus Beispiel 8–2 gestaltet. Sie können sie als Vorlage auch für andere Anwendungen nutzen.

Beispiel 8–2: Befehle zum Erstellen einer Komponente für Farblegenden (siehe Abb. 8–1)

```
function colorbox( sel, size, colors, ticks ) {    ❶
    var [x0, x1] = d3.extent( colors.domain() );    ❷
    var bars = d3.range( x0, x1, (x1-x0)/size[0] );

    var sc = d3.scaleLinear()    ❸
        .domain( [x0, x1] ).range( [0, size[0] ] );
    sel.selectAll( "line" ).data( bars ).enter().append( "line" )    ❹
        .attr( "x1", sc ).attr( "x2", sc )
        .attr( "y1", 0 ).attr( "y2", size[1] )
        .attr( "stroke", colors );
```

```
    sel.append( "rect" )    ❺
        .attr( "width", size[0] ).attr( "height", size[1] )
        .attr( "fill", "none" ).attr( "stroke", "black" )

    if( ticks ) {    ❻
        sel.append( "g" ).call( d3.axisBottom( ticks ) )
            .attr( "transform", "translate( 0," + size[1] + ")" );
    }
}
```

❶ Neben der Ziel-Selection nimmt die Komponente noch die folgenden Argumente entgegen: die Größe der Farblegende (in Pixeln) als zweielementiges Array, die Farbskalierung und optional eine reguläre Skalierung für Teilstriche.

❷ Erstellt ein Array, das für jedes Pixel in der gewünschten Breite des Farbbalkens den zugehörigen Wert aus dem Definitionsbereich enthält.

❸ Diese Skalierung bildet die Werte des ursprünglichen Definitionsbereichs auf die Pixelkoordinaten der Farblegende ab.

❹ Erstellt eine ein Pixel breite, farbige Linie für jeden Eintrag in `bars`.

❺ Zeichnet einen Rahmen um die Farben ...

❻ ... und fügt Teilstriche hinzu, falls ein geeignetes Skalierungsobjekt übergeben wurde.

Falschfarbendiagramme und verwandte Techniken

Wenn sich Daten naturgemäß auf eine zweidimensionale Ebene abbilden lassen, bieten sich Falschfarbendiagramme oder Heatmaps als eine attraktive Darstellungsmöglichkeit an, möglicherweise kombiniert mit (oder als Alternative zu) Höhenlinien.

Heatmaps

Eine Form der Visualisierung, die sich sehr stark auf Farben als Informationsträger stützt, ist das *Falschfarbendiagramm* oder die *Heatmap*. Der Wert eines Datenpunkts in einem zweidimensionalen Raster wird dabei durch eine Farbe dargestellt. Beispielsweise werden in topografischen Karten Geländehöhen auf diese Weise angegeben.

Die Anzahl der einzelnen Graphenelemente (Datenpunkte oder Pixel) in einem Falschfarbendiagramm kann selbst für Raster bescheidener Größe rasch sehr groß werden. Aus Leistungsgründen kann es daher beim Erstellen solcher Diagramme in D3 ratsam (wenn nicht sogar notwendig) sein, das HTML5-Element `<canvas>` zu nutzen (siehe »*Das HTML5-Element <canvas>*« auf *S. 175*). Beispiel 8–3 führt

eine einfache Anwendung dieses Elements vor. Das Ergebnis sehen Sie in Abbildung 8–2.

Beispiel 8–3: Falschfarbendiagramm von Teilen der Mandelbrotmenge. Dieser Graph wurde mit dem `canvas`*-Element von HTML5 erstellt (siehe Abb. 8–2).*

```
function makeMandelbrot() {
    var cnv = d3.select( "#canvas" );    ❶
    var ctx = cnv.node().getContext( "2d" );

    var pxX = 465, pxY = 250, maxIter = 2000;    ❷
    var x0 = -1.31, x1 = -0.845, y0 = 0.2, y1 = 0.45;

    var scX = d3.scaleLinear().domain([0, pxX]).range([x0, x1]);    ❸
    var scY = d3.scaleLinear().domain([0, pxY]).range([y1, y0]);

    var scC = d3.scaleLinear().domain([0,10,23,35,55,1999,2000])    ❹
        .range( ["white","red","orange","yellow","lightyellow",
                 "white","darkgrey"] );

    function mandelbrot( x, y ) {    ❺
        var u=0.0, v=0.0, k=0;
        for( k=0; k<maxIter && (u*u + v*v)<4; k++ ) {
            var t = u*u - v*v + x;
            v = 2*u*v + y;
            u = t;
        }
        return k;
    }

    for( var j=0; j<pxY; j++ ) {    ❻
        for( var i=0; i<pxX; i++ ) {
            var d = mandelbrot( scX(i), scY(j) );
            ctx.fillStyle = scC( d );
            ctx.fillRect( i, j, 1, 1 );
        }
    }
}
```

❶ Wählt das `<canvas>`-Element auf der Seite aus und ruft damit einen Zeichenkontext für einfache, zweidimensionale Grafiken ab. Beachten Sie, dass `getContext()` nicht zu D3, sondern zur DOM-API gehört. Daher müssen Sie zunächst mit der Methode `node()` den zugrunde liegenden DOM-Knoten aus der D3-Selection gewinnen.

❷ Legt einige Konfigurationsparameter fest: die Größe der Leinwand in Pixeln, die maximale Anzahl der Schritte in der Mandelbrotiteration und den Ausschnitt der komplexen Ebene, an dem wir interessiert sind.

❸ Erstellt zwei Skalierungen, die Pixelkoordinaten auf Örter in der komplexen Ebene abbilden.

❹ Erstellt eine Skalierung, um die Anzahl der Iterationsschritte auf eine Farbe abzubilden. Das Aussehen des Graphen hängt sehr stark von der Platzierung der Zwischenwerte ab.

❺ Diese Funktion implementiert die eigentliche Mandelbrotiteration: Für einen gegebenen Punkt x + `i`y der komplexen Ebene führt sie die Iteration durch, bis entweder das Quadrat des Abstands vom Ursprung 2^2 überschreitet oder bis die Höchstzahl an Schritten überschritten ist. Der Rückgabewert ist die Anzahl der durchgeführten Schritte. (Mehr über Mandelbrotmengen erfahren Sie auf *https://de.wikipedia.org/wiki/Mandelbrot-Menge*.)

❻ Die doppelte Schleife durchläuft alle Pixel der Leinwand. Die Pixelkoordinaten werden in Örter in der komplexen Ebene transformiert, die dann an die Funktion `mandelbrot()` übergeben werden. Deren Rückgabewert wird in eine Farbe umgewandelt, mit der dann ein gefülltes Rechteck der Größe 1 × 1 (also ein Pixel) auf die Leinwand gezeichnet wird.

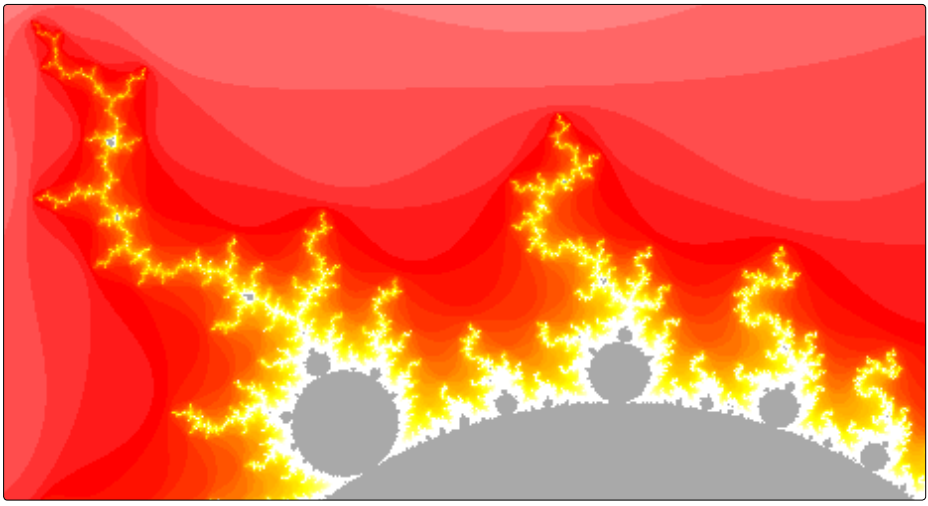

Abb. 8–2 *Verwendung des HTML5-Elements* `<canvas>` *(siehe Beispiel 8–3)*

Das HTML5-Element <canvas>

Das HTML5-Element `<canvas>` dient zum Erstellen von *Bitmapbildern*. Ebenso wie SVG kann es vom Browser aus mit Skripten gesteuert werden, allerdings bietet es im Gegensatz zu SVG nur elementare Bearbeitungsmöglichkeiten. Insbesondere stellt es keinen Zugriff auf einzelne Bestandteile des Diagramms bereit, um sie einzeln zu bearbeiten, wie es im DOM-Baum der Fall ist. Ein Bild auf dieser Leinwand ist eine »flache« Bitmap.

Sie müssen sich darüber im Klaren sein, dass es sich bei `<canvas>` in erster Linie um einen Platzhalter handelt. Um irgendetwas zeichnen zu können, müssen Sie zunächst mit dem API-Aufruf `getContext()` einen *Zeichenkontext* beschaffen. Dabei gibt es verschiedene Arten von Zeichenkontext mit unterschiedlichen APIs für unterschiedliche Arten der Programmierung (z. B. beschleunigte 3D-Grafik).

Wir beschäftigen uns hier ausschließlich mit dem einfachen Renderingkontext `CanvasRenderingContext2D`, der nur wenige grafische Grundelemente bietet. Darunter befinden sich Möglichkeiten, um gefüllte und nicht gefüllte Rechtecke und Text zu zeichnen, sowie eine Funktionalität zur Pfaderstellung, die ähnlich wie das SVG-Element `<path>` eine Art von Turtle-Grafik verwendet. Bevor Sie irgendetwas zeichnen können, müssen Sie die Strich- oder Füllfarbe als Kontexteigenschaft festlegen.

Der grundlegende Arbeitsablauf zum Zeichnen auf einer HTML5-Leinwand in D3 sieht wie folgt aus:

```
var cnv = d3.select( "#canvas" );
var ctx = cnv.node().getContext( '2d' );

ctx.fillStyle = color;
ctx.fillRect( x, y, w, h );
```

Der Wert von `color` muss eine CSS3-Farbspezifikation sein. Bei `x` und `y` handelt es sich um die Koordinaten der oberen linken Ecke des Rechtecks und bei `w` und `h` um seine Breite und Höhe.

Das Element `<canvas>` lässt sich im Großen und Ganzen recht einfach verwenden. Ein Tutorial finden Sie auf *https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial*.

Höhenlinien

Eine Alternative (oder Ergänzung) zu Falschfarbendiagrammen – insbesondere für Daten mit fließenden Übergängen – bilden *Höhenlinien*, also Kurven, die Punkte gleicher Höhe verbinden (wie in topografischen Karten). Zur Berechnung solcher Linien stellt D3 ein *Layout* bereit (siehe Tabelle 8–2).

Funktion	Beschreibung
`d3.contours()`	Gibt ein neues Höhenlinienlayout mit Standardeinstellungen zurück.
`conMkr( [data] )`	Berechnet die Höhenlinien für die übergebene Datenmenge. Die Daten müssen als eindimensionales Array so formatiert sein, dass das Arrayelement mit dem Index `[i + j*cols]` dem Element an der Position `[i, j]` im Raster entspricht. Gibt ein Array aus GeoJSON-Objekten zurück, die nach den Schwellenwerten sortiert sind, für die sie stehen.
`conMkr.size( [cols, rows] )`	Legt die Anzahl der Zeilen und Spalten als zweidimensionales Array fest.
`conMrk.thresholds( args )`	Das Argument ist gewöhnlich ein Array der Werte, für die Höhenlinien berechnet werden sollen. Ist es dagegen ein einzelner Integer *n*, dann werden etwa *n* Höhenlinien in geeignetem Abstand gezeichnet.
`conMkr.contour( [data], threshold )`	Erstellt am angegebenen Schwellenwert eine einzelne Höhenlinie für die übergebene Datenmenge und gibt ein einzelnes GeoJSON-Objekt zurück.

Tab. 8–2 *Funktionen zur Berechnung von Höhenlinien (wobei* `conMkr` *ein Höhenlinienlayout ist)*

Die erforderliche Datendarstellung ist etwas merkwürdig. Es wird vorausgesetzt, dass sich die Datenpunkte in einem gleichmäßigen, rechteckigen Raster aus Zeilen (`rows`) und Spalten (`cols`) befinden, das als *eindimensionales Array* aus Zahlen gespeichert ist, wobei die Zeilen eine kontinuierliche Folge bilden. Das Element am Arrayindex `i + j*cols` steht daher für den Punkt an der Position `[i, j]`. Es gibt keine Möglichkeit, um die einzelnen Datenpunkte mit den eigentlichen (Definitionsbereichs-)Koordinaten zu verknüpfen. Das hat unter anderem zur Folge, dass das resultierende Diagramm `cols` x `rows` Pixel groß ist. Wenn Sie eine andere Größe haben wollen, müssen Sie eine Skalierungstransformation anwenden.

Wenn das Höhenlinienlayout für die Daten aufgerufen wird, gibt es ein Array aus je einem GeoJSON-Objekt[6] für jede einzelne Höhenlinie zurück. Anschließend können Sie mit dem Generator `d3.geoPath()` einen geeigneten Befehlsstring für das Attribut `d` eines regulären `<path>`-Elements erstellen. (Dieser Vorgang ähnelt demjenigen aus Kapitel 5.) Jedes Höhenlinienobjekt stellt die Eigenschaft `value` mit dem Schwellenwert bereit, für den die Linie steht.

6 GeoJSON ist ein Standard zur Darstellung von geografischen Daten. Definiert ist er in RFC 7946.

Wenn Sie die Höhenlinien mit Farben füllen, ergibt sich wiederum ein Falschfarbendiagramm oder eine Heatmap. In Beispiel 8–4 werden beide Darstellungen gemeinsam verwendet: Erst wird eine große Menge von gefüllten Höhenlinien gezeichnet, um einen farbigen Hintergrund mit fließenden Verläufen zu bilden. Anschließend werden auf diesen Hintergrund einige Höhenlinien an ausgewählten Schwellenwerten gezeichnet (siehe Abb. 8–3).

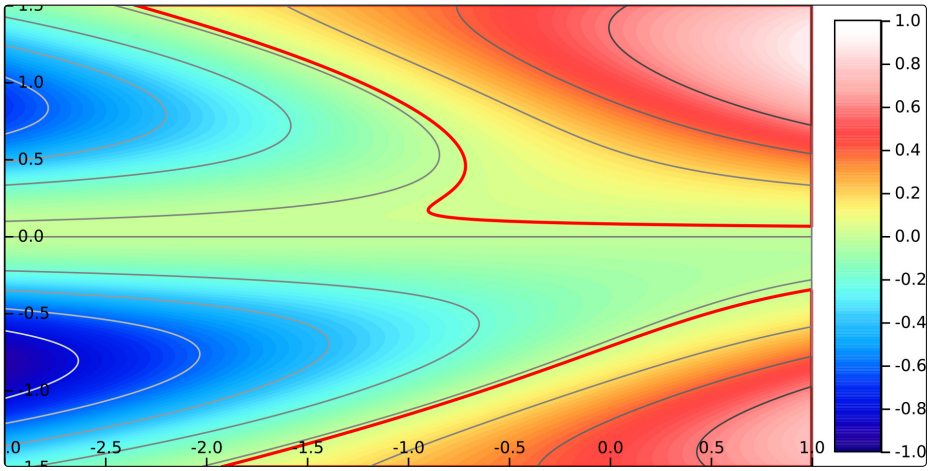

Abb. 8–3 *Ein Falschfarbendiagramm einer glatten Funktion. Dieser Graph wurde mit dem D3-Mechanismus der Höhenlinienlayouts erstellt (siehe Beispiel 8–4).*

Beispiel 8–4: Ein Falschfarbendiagramm mit Höhenlinienlayouts erstellen (siehe Abb. 8–3)

```
function makeContours() {
    d3.json( "haxby.json" ).then( drawContours );   ❶
}

function drawContours( scheme ) {
    // Richtet die Skalierungen einschließlich Farbskalierungen ein
    var pxX = 525, pxY = 300;   ❷
    var scX = d3.scaleLinear().domain([-3, 1]).range([0, pxX]);
    var scY = d3.scaleLinear().domain([-1.5, 1.5]).range([pxY, 0]);
    var scC = d3.scaleSequential(
        d3.interpolateRgbBasis(scheme["colors"]) ).domain([-1,1])   ❸
    var scZ = d3.scaleLinear().domain( [-1, -0.25, 0.25, 1] )
        .range( [ "white", "grey", "grey", "black" ] );

    // Generiert die Daten
    var data = [];   ❹
    var f = (x, y, b) => (y**4 + x*y**2 + b*y)*Math.exp(-(y**2))
```

```
    for( var j=0; j<pxY; j++ ) {
        for( var i=0; i<pxX; i++ ) {
            data.push( f( scX.invert(i), scY.invert(j), 0.3 ) );
        }
    }

    var svg = d3.select( "#contours" ), g = svg.append( "g" );   ❺
    var pathMkr = d3.geoPath();   ❻

    // Erstellt und zeichnet gefüllte Höhenlinien (Tönung)
    var conMkr = d3.contours().size([pxX, pxY]).thresholds(100);   ❼
    g.append("g").selectAll( "path" ).data( conMkr(data) ).enter()
        .append( "path" ).attr( "d", pathMkr )   ❽
        .attr( "fill", d=>scC(d.value) ).attr( "stroke", "none" )

    // Erstellt und zeichnet Höhenlinien
    conMkr = d3.contours().size( [pxX,pxY] ).thresholds( 10 );   ❾
    g.append("g").selectAll( "path" ).data( conMkr(data) ).enter()
        .append( "path" ).attr( "d", pathMkr )
        .attr( "fill", "none" ).attr( "stroke", d=>scZ(d.value) );

    // Erstellt eine einzelne Höhenlinie
    g.select( "g" ).append( "path" )   ❿
        .attr( "d", pathMkr( conMkr.contour( data, 0.025 ) ) )
        .attr( "fill", "none" ).attr( "stroke", "red" )
        .attr( "stroke-width", 2 );

    // Erstellt die Achsen
    svg.append( "g" ).call( d3.axisTop(scX).ticks(10) )   ⓫
        .attr( "transform", "translate(0," + pxY + ")" );
    svg.append( "g" ).call( d3.axisRight(scY).ticks(5) );

    // Erstellt die Farblegende
    svg.append( "g" ).call( colorbox, [280,30], scC )   ⓬
        .attr( "transform", "translate( 540,290 ) rotate(-90)" )
        .selectAll( "text" ).attr( "transform", "rotate(90)" );
    svg.append( "g" ).attr( "transform", "translate( 570,10 )" )
        .call( d3.axisRight( d3.scaleLinear()
                             .domain( [-1,1] ).range( [280,0] ) ) );
}
```

❶ Lädt die Farben des Farbschemas aus einer Datei und übergibt sie an die Funktion `drawContours()`, die die eigentliche Arbeit ausführt.[7]

❷ Legt die Größe des Diagramms in Pixeln fest. Beachten Sie, dass es einen Datenpunkt pro Pixel geben muss.

7 Dieses Farbschema basiert auf der Palette GMT Haxby des Projekts Generic Mapping Tools (siehe *http://soliton.vm.bytemark.co.uk/pub/cpt-city/gmt/index.html*).

❸ Die aus der Datei geladenen Farben werden zusammen mit dem Interpolierer `d3.interpolateRgbSpline` eingesetzt, um ein Farbskalierungsobjekt zu erstellen (siehe das letzte Element in Beispiel 8–1). Ein separates Skalierungsobjekt namens `scZ` bestimmt die Farbe der *Höhenlinien*, damit sie gegen den Hintergrund mit seinen wechselnden Farben gut sichtbar sind.

❹ Generiert die Daten durch Auswertung der Funktion `f(x, y)` für jede Pixelkoordinate. Mit der Funktion `invert()` des Skalierungsobjekts werden die Definitionsbereichskoordinaten für jede Pixelkoordinate abgerufen.

❺ Erstellt ein Handle für den DOM-Baum und hängt ein `<g>`-Element als Container für den Hauptteil des Diagramms an.

❻ Erstellt eine Instanz des Generators `d3.geoPath`. Hierbei handelt es sich um ein Funktionsobjekt. Wenn es für ein GeoJSON-Objekt aufgerufen wird, gibt es einen für das Attribut `d` eines `<path>`-Elements geeigneten String zurück.

❼ Erstellt ein Höhenlinienlayout und legt die Anzahl der Pixel im Graphen und die Daten fest. Die Anzahl der Höhenlinien ist sehr groß. Wenn jede davon mit Farbe gefüllt wird, ergeben sie insgesamt eine Heatmap mit einem fließenden Farbverlauf.

❽ Hängt für jede Höhenlinie ein `<path>`-Element an und füllt es entsprechend der Eigenschaft `value` des Höhenlinienobjekts mit einer Farbe. Für jede Höhenlinie wird der Generator `pathMkr` aufgerufen und gibt einen geeigneten String für das Attribut `d` eines `<path>`-Elements zurück.

❾ Konfiguriert das Höhenlinienlayout für etwa zehn Höhenlinien und ruft es dann auf, um ungefüllte Höhenlinien zu erstellen.

❿ Nur um Ihnen zu zeigen, wie es gemacht wird: Mit der Funktion `contour()` können Sie eine *einzelne* Höhenlinie für den angegebenen Schwellenwert erstellen.

⓫ Fügt mithilfe des ursprünglichen Skalierungsobjekts Koordinatenachsen zu dem Graphen hinzu.

⓬ Fügt eine Farblegende hinzu, die zeigt, welche numerischen Werte den Farben in der Heatmap entsprechen. Dieser Code nutzt die Komponente `colorbox` aus Beispiel 8–2 und wendet dann eine SVG-Transformation an, um sie vertikal auszurichten.