R5.A.06 - Prog multimédia - CM n°2

Pierre Nerzic

septembre 2023

d3.js = *Data Driven Documents JavaScript library*

bibliothèque JavaScript ES6 permettant de créer des diagrammes de haute qualité et réactifs.

Elle est destinée aux scientifiques voulant visualiser des informations numériques.

Mais pas seulement, car cette API est accessible à une autre profession, intermédiaire entre science, journalisme et informatique, le *data journalisme*. C'est un métier qui consiste à expliquer le plus simplement possible, avec des graphiques interactifs, des informations numériques complexes.

Voir par exemple, cet entretien.

Pierre Nerzic

Introduction

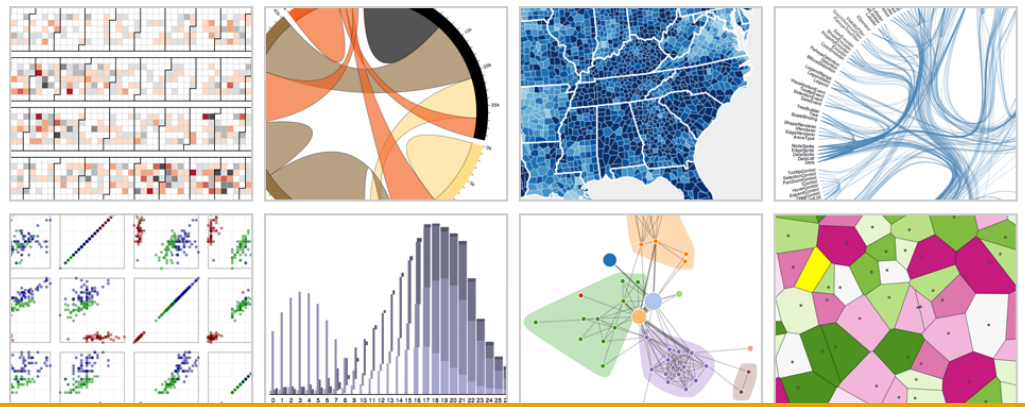
Pierre Nerzic

Concepts généraux

d3.js est une API de bas niveau, créée en 2011 par Mike Bostock dans le cadre de sa thèse sur la visualisation de données. Il a proposé de manipuler soi-même le DOM (image SVG) pour créer des graphiques, au lieu d'utiliser des fonctionnalités toutes faites.

- d3.js : permet de tout faire, mais avec beaucoup de travail
- autres : tout est déjà fait, sauf ce dont on a besoin.

d3.js est destiné à faire des visualisations innovantes, impossibles à faire avec des API ordinaires.



Exemples à aller voir

Quelques exemples dans le New York Times :

- corporate-taxes
- global business asia
- congrès républicain

Quelques exemples avec les sources : galerie observablehq

Voir aussi les remarquables pages perso de Mike Bostock
`https://bost.ocks.org/mike/` et `https://observablehq.com/@mbostock`.
Par exemple, l'illustration de cet algo de mélange.

Concepts de d3.js

On dispose de données : liste de n-uplets, tableau, arbre, etc.

Chaque information est dessinée avec des éléments SVG, ex: barres dans un histogramme, points ou chemins dans un diagramme, etc. On doit programmer la génération de ces éléments.

Il y a une association forte entre les données et leur représentation. L'API d3.js est centrée sur la manipulation du DOM en fonction des données à dessiner.

Il y a de nombreux modules pour ajouter des graduations, des légendes et aussi gérer les interactions avec l'utilisateur.

Points forts, points faibles

- d3.js optimise les changements provoqués par des données dynamiques. Tout n'est pas redessiné à chaque changement dans les données.
- d3.js demande du travail pour faire un graphique quel qu'il soit. Il faut bien connaître la norme SVG et bien comprendre JavaScript (lambdas et fermetures). Pour des graphiques standard, il existe d'autres bibliothèques, plus faciles. Certaines même sont basées sur d3.js. d3.js est appropriée quand rien n'existe pour le graphique voulu.
- d3.js évolue vite (v7) et de nombreux tutoriels sont devenus des antiquités (v5 et antérieures).

Mécanismes de d3.js

Pierre Nerzic

Installation

Avec un CDN (*content delivery network* ou réseau de diffusion de contenu) :

```
<!DOCTYPE html>
<html>
<head>
  <!-- d3.js -->
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
</head>
<body>
...
```

Les scripts de dessin seront plutôt placés dans le <body> ou alors déclenchés par un événement onload.

Pierre Nerzic

Sélection d'éléments

C'est la fonctionnalité de base de d3.js. Le but est d'attraper des éléments du DOM pour les modifier. Une *sélection* dans d3.js est une liste d'éléments (balises du DOM) attachée à un élément parent. Cette liste peut être vide.

Deux méthodes à connaître :

- `d3.select(`*sélecteur*`)` retourne une sélection contenant le premier élément qui correspond au sélecteur ; vide si aucun.
- `d3.selectAll(`*sélecteur*`)` retourne une sélection contenant tous les éléments qui correspondent au sélecteur, dans l'ordre du document.

```
const svg = d3.select("#dessin")
const textes = d3.selectAll("text")
```


Pierre Nerzic

Sélection d'éléments, suite

Les sélections peuvent être enchaînées, car toute sélection possède également ces deux méthodes, `select()` et `selectAll()`. Le résultat est l'ensemble de tous les descendants désignés.

```
d3.select("#dessin")
  .selectAll("g")
  .selectAll("rect")
```

sélectionne tous les éléments `<rect>`, enfants de tous les `<g>` qui sont dans l'élément identifié par `#dessin`.

Attention, on ne peut pas toujours la simplifier en :

```
d3.selectAll("#dessin > g > rect")
```

parce que le parent de cette sélection est le document HTML, alors que celui de la première est l'élément `<g>`.

Pierre Nerzic

Modifications des éléments sélectionnés

On peut modifier les attributs, la classe et le style par des méthodes spécifiques :

- *sélection*`.attr(`*nom*`,`*valeur*`)` crée ou modifie l'attribut
- *sélection*`.classed(`*nom(s)*`, true` ou `false)` ajoute ou retire la ou les classes indiquées
- *sélection*`.style(`*nom*`,`*valeur*`)` crée ou modifie le style

Si *valeur* est `null`, alors l'attribut ou style est enlevé.

```
const svg = d3.select("#dessin")
svg.selectAll("text")
    .attr("x", 10)
    .classed("titre important", true)   // ajoute ces classes
    .style("fill", "Gold")
    .style("stroke", null)    // supprime le style stroke
```


Pierre Nerzic

Modifications des éléments sélectionnés, suite

On peut aussi modifier le contenu des éléments d'une sélection :

- *sélection*`.text(`*valeur*`)` crée ou modifie le contenu texte de la sélection. Si *valeur* est `null`, alors le texte est enlevé.
- *sélection*`.append(`*nom*`)` ajoute un sous-élément `<nom>` et retourne cet élément en tant que nouvelle sélection
- *sélection*`.remove()` supprime les éléments de la sélection

```
svg.selectAll("text")
    .text("C'est pas faux.")
const g = svg
  .append("g")           // nouvelle sélection
    .attr("transform", "rotate(45)")
d3.select("#alerte")
  .remove()
```


Pierre Nerzic

Convention d'indentation

Il est recommandé d'indenter :

- 2 espaces pour une méthode qui retourne une nouvelle sélection
- 4 espaces pour une méthode qui retourne la sélection courante

```
d3.select("body")                  // sélection = l'élément <body>
  .append("svg")                   // <svg> ==> nouvelle sélection
    .attr("width", 960)
    .attr("height", 360)
  .append("g")                     // crée un <g> dans le <svg>
    .attr("transform", "translate(20,20)")
  .append("rect")
    .attr("width", 920)
    .attr("height", 360)
```

NB: il serait plus lisible de découper cette instruction en plusieurs parties.

Pierre Nerzic

Valeurs fournies aux méthodes

Les méthodes précédentes, `attr()`, `style()`, `text()` demandent une *valeur*. Elle peut être une constante, chaîne ou nombre et alors c'est cette valeur qui est affectée à tous les éléments de la sélection.

Ça peut aussi être une fonction. Elle est alors appelée pour chaque élément de la sélection, et peut donc retourner des valeurs distinctes. Elle reçoit deux paramètres : la donnée courante et l'indice de l'élément dans la sélection.

```
svg.selectAll("text")                    // sélection des <text>
  .data([2, 3, 5, 7, 11, 13, 17])        // voir plus loin
    .text((d, i) => `v${i} = ${d}`)      // contenu des <text>
```

Si c'est une fonction (pas une lambda), alors `this` est lié à l'élément courant du DOM.

Pierre Nerzic

Création/modification/suppression automatique d'éléments

Dans l'exemple précédent, il faut qu'il y ait préalablement exactement autant d'éléments `<text>` que de valeurs fournies à `data()`.

La méthode `join(`*élément*`)` permet d'adapter les éléments du DOM aux données. Elle crée des éléments s'il en manque, et supprime ceux qui sont en trop.

```
svg.selectAll("text")
  .data([2, 3, 5, 7, 11, 13, 17])
  .join("text")           // crée/supprime des <text>
    .text((data, i) => `valeur ${i} = ${data}`)
```

Les méthodes `selectAll()`, `data()` et `join()` travaillent ensemble.

Méthode data() approfondie

Reprenons le début de ce code dans un contexte complet :

```
<svg id="dessin" viewBox="..."></svg> // vide

<script>
const svg = d3.select("#dessin")       // élément <svg>
svg.selectAll("text")                  // sous-éléments <text>
  .data([2, 3, 5, 7, 11, 13, 17])      // données à associer
```

C'est assez bizarre au premier regard. On sélectionne tous les éléments <text> de l'élément <svg> mais il n'y en a aucun.

Oui, il n'y en a aucun, mais la sélection mémorise l'élément parent de ces <text>, c'est <svg>, donc la méthode data(*valeurs*) sait qu'il va falloir y créer 7 éléments enfants, un pour chaque donnée. NB: à ce stade, ces nouveaux éléments sont indéfinis.

Méthode `data()` approfondie, suite

sélection`.data(`*valeurs*`)` retourne une nouvelle sélection, celle des éléments existants de la sélection qui ont été liés avec les valeurs fournies. Cette sélection d'éléments existants est appelée *update*.

Elle retourne aussi deux autres sélections appelées *enter* et *exit*, mais elles ne sont pas directement visibles.

- *enter* est la sélection des nouveaux éléments (entrants), encore fictifs (*placeholders*) qui seront rajoutés aux existants pour coller aux données supplémentaires,
- *exit* est la sélection des éléments existants qui doivent être supprimés pour coller aux données.

En fait, les trois sélections voyagent ensemble, mais les méthodes `attr()`, `style()`, `text()`, etc. s'adressent seulement à *update*.

Pierre Nerzic

Sélections *enter* et *exit*

enter et *exit* sont des sortes de canaux cachés dans une sélection.
On peut y accéder avec les méthodes `enter()` et `exit()` :

```
svg.selectAll("text")
  .data([2, 3, 5, 7, 11, 13, 17, 19, 23, 29])
  .enter()                  // la sélection courante devient «enter»
    .append("text")         // ajoute un élément <text>
      .attr("fill", "green")
      .text((d, i) => `valeur ${d}`)
```

```
svg.selectAll("text")
  .data([11, 13, 17])
  .exit()                   // la sélection courante devient «exit»
    .remove()               // supprime ces éléments
```

Les anciennes versions de d3.js fonctionnaient comme ça.

Sélections *enter* et *exit*, suite

En fait, si on est sûr de uniquement créer des éléments, on peut n'utiliser que la sélection *enter*. Inversement, si on est sûr de ne faire qu'en supprimer, on peut utiliser seulement *exit* :

```
let points = [{"x": 1, "y": 4}, {"x": 7, "y": 3}, ...]
svg.selectAll("circle")
  .data(points)
  .enter().append("circle")
    .attr("cx", d => d.x)
    .attr("cy", d => d.y)
    .attr("r", 2.5)
```

`cx` et `cy` sont définis par des lambdas ; leur paramètre `d` est l'un des objets de `points` ; ces objets ont deux propriétés, x et y. C'est ainsi que l'attribut `cx` est calculé par la fonction qui à d retourne $d.x$.

Liaison des données aux éléments du DOM

En résumé, on sélectionne une partie du DOM par exemple des éléments `<text>` ou `<rect>` dans un SVG :

```
svg.selectAll("TYPE")
```

On les associe un par un à des données :

```
  .data(liste de valeurs)
```

On obtient une sorte de superposition de 3 listes : les éléments à modifier *update*, ceux à ajouter *enter* et ceux à supprimer *exit*.

```
  .join("TYPE")                  // le même TYPE qu'au selectAll
```

La méthode `join()` permet de spécifier quoi faire avec ces trois listes : créer, mettre à jour et supprimer.

Méthode `join()` approfondie

La méthode *sélection*`.join(`*élément*`)` gère les trois listes de la sélection comme on peut s'y attendre : elle supprime ce qu'il y a dans *exit*, crée des éléments pour *enter* et met à jour ceux de *update* avec ce qui suit l'appel à la méthode.

```
svg.selectAll("text")
  .data([2, 3, 5, 7, 11, 13, 17])
  .join("text")           // crée/édite/supprime des <text>
    .text((data, i) => `valeur ${i} = ${data}`)
```

Après exécution de `join()`, les listes *enter* et *update* sont fusionnées et *exit* est enlevée de la sélection. Il ne reste plus que les anciens éléments conservés avec les nouveaux. On peut donc chaîner d'autres méthodes pour les modifier tous de la même manière.

Méthode `join()` approfondie, suite

En fait, la méthode `join()` est plus riche. On peut lui fournir trois fonctions (ou lambdas) *enter*, *update* et *exit* dans cet ordre.

```
svg.selectAll("text")
  .data([2, 3, 5, 7, 11, 13, 17])
  .join(
      enter  => enter.append("text"),
      update => update.style("background-color", "Bisque"),
      exit   => exit.remove()
  )
    .text((data, i) => `valeur ${i} = ${data}`)
```

La première lambda gère les éléments à ajouter au DOM. La deuxième s'occupe des éléments existants, et ici on les colore en orange clair. La troisième lambda supprime les <text> devenus inutiles. Ensuite, on affecte le contenu texte des éléments restants.

Création d'un graphique avec d3.js

Pierre Nerzic

Présentation

On veut dessiner un graphique de type barres ultra simple.

Il représente ces valeurs : $4, 8, 15, 16, 42, 7, 23$.

NB: c'est l'adaptation d'une série de tutoriels de Mike Bostock, Let's Make a Bar Chart

Pierre Nerzic

Cadre général

On part de données disponibles dans une liste. Ces données doivent être liées à des rectangles SVG plus ou moins hauts.

```
<!DOCTYPE html>
<html>
<head>
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
</head>
<body>
  <svg id="dessin" viewBox="0 0 100 50" fill="purple"></svg>
  <script>
const valeurs = [4, 8, 15, 16, 42, 7, 23]

  </script>
</body>
</html>
```

Objectif

On doit générer quelque chose comme ça, automatiquement :

```
<svg id="dessin" viewBox="0 0 100 50" fill="purple">
  <rect x="0" y="46" width="9" height="4"></rect>
  <rect x="10" y="42" width="9" height="8"></rect>
  <rect x="20" y="35" width="9" height="15"></rect>
  <rect x="30" y="34" width="9" height="16"></rect>
  ...
</svg>
```

- Les rectangles ont tous la même largeur et leur position `x` va de 10 en 10.
- On reconnaît les valeurs dans les attributs `height` et aussi (50 − *valeur*) dans les `y`.

NB: les coordonnées et tailles sont relatives à la *viewBox*.

Réalisation

1. Obtenir une sélection des rectangles concernés.
   NB: initialement, cette sélection est vide.

```
const svg = d3.select("#dessin")
svg.selectAll("rect")
```

2. Lier les données à la sélection

```
  .data(valeurs)
  .join("rect")
```

3. Définir les coordonnées, tailles et couleur

```
    .attr("x",      (d, i) => i*10)   // d = valeurs[i]
    .attr("y",      (d, i) => 50-d)
    .attr("width",  9)
    .attr("height", (d, i) => d)
```

Compléments graphiques

Pierre Nerzic

Introduction

Le graphique précédent, simpliste, permet de comprendre les concepts de base : sélection et liaison avec des données.

- Il n'y a aucune adaptation des coordonnées à des données en plus grand nombre ou ayant une amplitude très variable. Il faudrait étirer horizontalement et verticalement en fonction des données.
- Le graphique est très pauvre en information. Il faudrait afficher une échelle sur les deux axes. On peut aussi ajouter un titre.

Ces améliorations se font à l'aide de modules de d3.js. Ils sont assez nombreux, certains généralistes, d'autres très spécifiques.

Mise à l'échelle

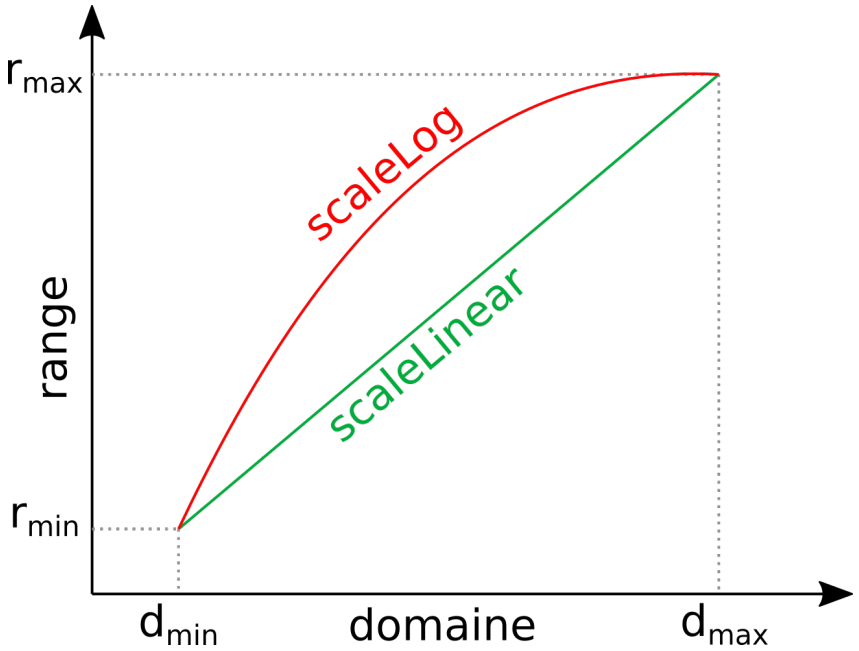
Dans la plupart des cas, il faut adapter les dimensions du graphique à la zone disponible sur l'écran. Cela passe par des transformations de coordonnées, par exemple une homothétie.

Il y a trois concepts à comprendre :

- l'amplitude d'entrée appelée *domain*, c'est la plage de variation des données d'entrée
- l'étendue de sortie appelée *range*, c'est ce qu'on veut comme amplitude à l'écran, par rapport à la *viewBox*
- la fonction de transfert, elle peut être linéaire (affine), logarithmique, temporelle... Cette fonction retourne une valeur du *range* à partir d'une donnée du *domain*.

Mise à l'échelle, suite

Une échelle met en correspondance un domaine et une étendue (*range*). Le domaine est celui des données d'entrée ; l'étendue est celle des coordonnées écran. Le but est que la plus petite donnée soit affichée à la plus petite coordonnée (à gauche ou en bas), et la plus grande donnée soit affichée à la plus grande coordonnée.



Pierre Nerzic

Mise à l'échelle, suite

Le module d3-scale.js offre de nombreux types d'échelles sous forme de méthodes de la classe d3, par exemple des échelles linéaires pour les axes x et y se définissent ainsi :

```
const fX = d3.scaleLinear()
    .domain([0, valeurs.length])  // domaine : 0..nb données
    .range([0, 100])              // 100 = viewBox.width
const fY = d3.scaleLinear()
    .domain(d3.extent(valeurs))   //min(valeurs)..max(valeurs)
    .range([0, 50])               // 50 = viewBox.height
```

`fX` et `fY` sont à la fois des fonctions et des objets possédant des méthodes. Par exemple, `fX.range()` retourne `[0,100]` et `fY.domain()` retourne `[0, 42]`. Voir la doc. Voir aussi d3-array pour `extent`, `min`, `max`...

Mise à l'échelle, suite

Comme on a des données un peu particulières, une liste de valeurs, on peut employer une échelle horizontale appropriée : 

```
const fX = d3.scaleBand()
    .domain(d3.range(valeurs.length))  // [0,1,2,...,5,6]
    .range([0, 100])                   // 100 = viewBox.width
    .padding(0.1)                      // espacement des bandes
```

scaleBand crée une échelle entre des valeurs discrètes (une liste) et un *range* continu (un axe). Ici, on veut lier les indices dans le tableau des valeurs à l'axe X.

La méthode `d3.range(`N`)` fonctionne exactement comme en Python. Elle retourne le tableau $[0..N-1]$ (doc), voir d3-range.

Mise à l'échelle, suite

Ces échelles `fX()` et `fY()` servent à affecter les coordonnées : 

```
    .attr("x",      (d, i) => fX(i))
    .attr("y",      (d, i) => 50-fY(d))
    .attr("width",  fX(1))
    .attr("height", (d, i) => fY(d))
```

Le graphique sera parfaitement cadré dans la *viewBox* quelles que soient les données.

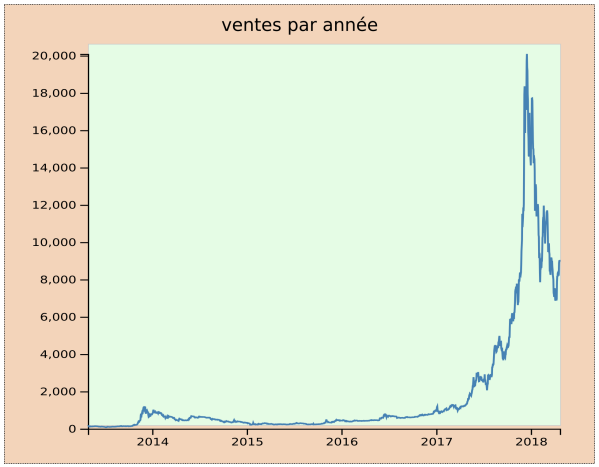
NB: il est important de transformer les dimensions de la viewBox en constantes et les employer partout.

Pierre Nerzic

Axes et graduations

Comment rajouter des axes et des graduations ?

d3.js propose une *convention pour les marges* margin convention. C'est une norme pour dessiner un graphique avec ses axes, sa légende et son titre.

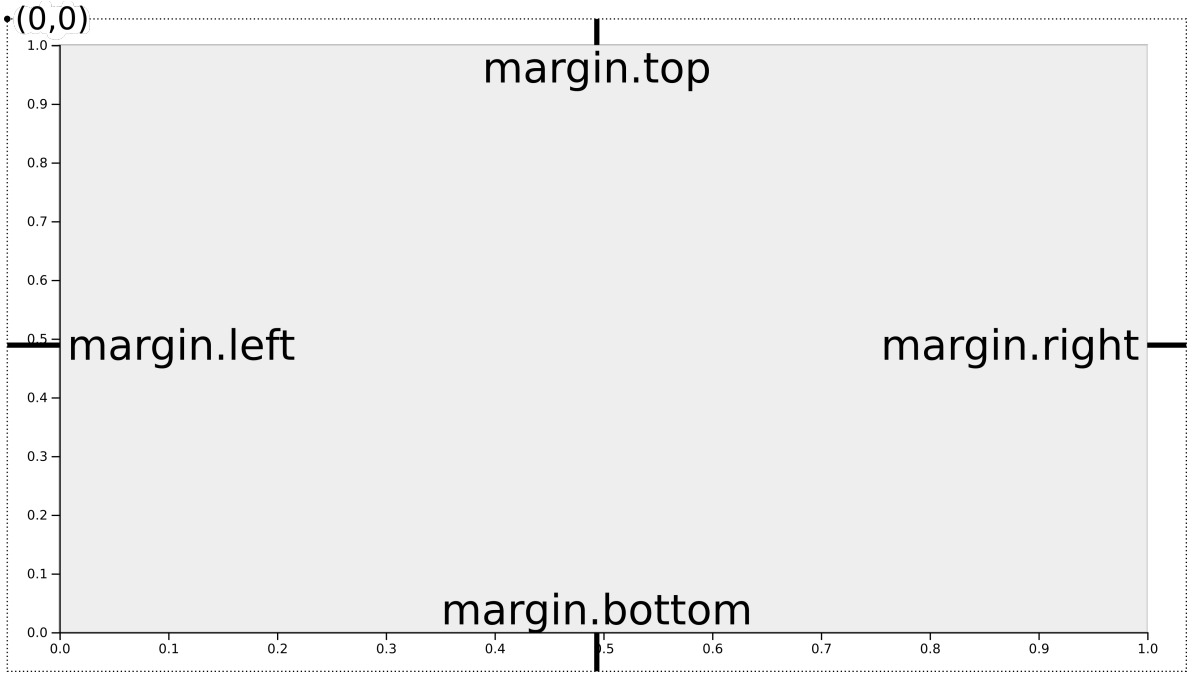


Le graphique est dessiné dans le rectangle vert intérieur par une modification astucieuse des fonctions d'échelle `fX` et `fY`.

Convention des marges

On commence par définir un objet qui contient les 4 marges :

```
const width = 200
const height = 100
const margin = {top: 20, right: 10, bottom: 30, left: 40}
```



Pierre Nerzic

Convention des marges, suite

Ensuite, on définit les fonctions de mise à l'échelle un peu différemment. Le *range* tient compte de la taille réduite de la zone disponible pour dessiner :

```
const fX = d3.scaleLinear() // ou d3.scaleBand()
    .domain(..)  // INCHANGÉ
    .range([margin.left, width - margin.right])

const fY = d3.scaleLinear()
    .domain(..)  // INCHANGÉ
    .range([height - margin.bottom, margin.top]) // ordre !!!
```

NB: pour `fY`, le *range* est inversé, car *y* va vers le bas dans un SVG.

Avec cette convention, on aura toujours ces *range*-là, quel que soit le graphique. Seuls les domaines et types d'échelles seront spécifiques.

Convention des marges, suite

On ne change presque rien à la manière de dessiner : 

```
svg.selectAll("rect")
  .data(valeurs)
  .join("rect")
    .attr("x",      (d, i) => fX(i))
    .attr("y",      (d, i) => fY(d))
    .attr("width",  fX(1) - fX(0))
    .attr("height", (d, i) => fY(0) - fY(d))
```

Remarquez les astuces de calcul pour calculer `width` et `height` avec les fonctions de mise à l'échelle. `fX(0)` et `fY(0)` donnent la référence du repère de dessin.

NB: les tutos de Mike Bostock regorgent d'astuces remarquables.

Ajout d'axes

d3.js offre des mécanismes pour ajouter des axes et des graduations à un graphique. Un axe est quelque chose d'assez complexe. Ça comprend une ligne horizontale ou verticale couvrant tout le domaine, des graduations (*ticks*) et des valeurs espacées régulièrement, le tout placé dans un groupe <g>.

Tout cela est créé par ces instructions :

```
svg.append("g")
    .attr("transform", `translate(0,${height - margin.bottom})`)
    .call(d3.axisBottom(fX))  // crée l'axe X en bas

svg.append("g")
    .attr("transform", `translate(${margin.left},0)`)
    .call(d3.axisLeft(fY))    // crée l'axe Y à gauche
```

Méthode `call`

La méthode `call` d'une sélection permet de simplifier l'écriture.

```
sélection.call(fonction, paramètres...)
```

est équivalente à

```
fonction(sélection, paramètres...)
```

On utilise `call` quand la sélection est complexe, et d'autre part, `call` retourne la sélection, ce qui permet de chaîner d'autres méthodes.

Méthode call, suite

Ainsi :

```
svg.append("g")
    .attr("transform", `translate(0,${height - margin.bottom})`)
    .call(d3.axisBottom(fX))
```

est équivalente à :

```
d3.axisBottom(fX)(
  svg.append("g")
      .attr("transform", `translate(0,${height - margin.bottom})`)
)
```

parce que `d3.axisBottom(fX)` est une fonction qui crée une multitude d'éléments SVG dans la sélection (le groupe <g>).

Ajout d'axes, suite

Mike Bostock ajoute les axes d'une autre manière :

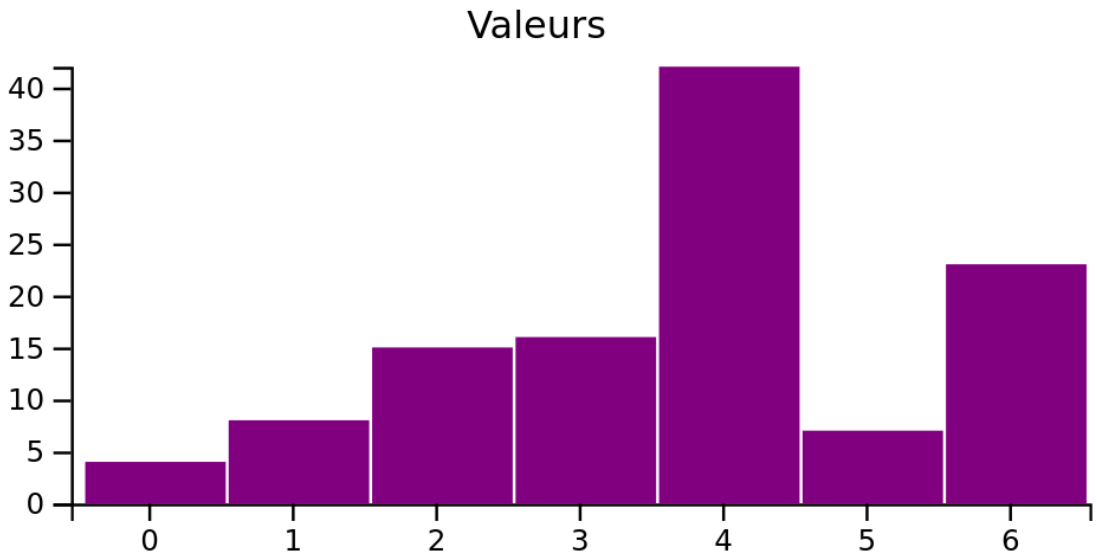
```
const axeX = g =>
  g.attr("transform", `translate(0,${height - margin.bottom})`)
      .call(d3.axisBottom(fX))

svg.append("g")
    .call(axeX)
```

L'axe X est défini dans une variable, `axeX`, sous forme d'une lambda et il est rajouté au graphique ultérieurement. Cela permet de rendre le dessin modulaire. On définit des lambdas pour créer les échelles, créer les axes, compléter le dessin.

Ajout d'un titre

Pour mettre un titre centré en haut :

```
svg.append("text")
    .attr("text-anchor", "middle")
    .attr("x", width/2)
    .attr("y", margin.top/2)
    .text("Valeurs")
```



Pierre Nerzic

Données dynamiques

Pierre Nerzic

Mise à jour des données

La modification des données ou des caractéristiques d'un graphique demande de tout redessiner ce qui est concerné, c'est à dire de relancer tout le script.

Ce n'est pas du tout catastrophique, car le DOM est déjà en place grâce à l'état précédent et d3.js minimise les modifications. La méthode `data()` calcule trois listes : les nouveaux éléments, les éléments existants réutilisables et les éléments superflus. Seuls les nouveaux éléments et ceux à modifier font l'objet de calculs. Il en est de même pour les axes.

Il y a une notion de clé (doc) pour accompagner les données de manière à garder un ordre entre les éléments existants et les nouveaux, mais c'est trop complexe pour ce cours, voir le TD3.

Pierre Nerzic

Obtention des données

Le module d3-fetch est très utile pour télécharger des données d'un serveur, ou lire des données dans un fichier local :

```
d3.json("https://serveur/data.json") // ou d3.json("data.json")
  .then(valeurs => dessiner(valeurs)) // dessin quand data reçu
```

Comme toutes les méthodes de ce module, `d3.json(`*URL* ou *fichier*`)` est *asynchrone*. Elle retourne une *promesse* qui permet d'attendre la réception des données pour dessiner.
On peut aussi placer le chargement dans une fonction asynchrone :

```
async function getData() {
  const valeurs = await d3.json("https://serveur/data.json")
  dessiner(valeurs)
}
getData()   // appel pour dessiner les données
```


Pierre Nerzic

Transitions

d3-transition permet de rendre progressifs des changements sur une sélection. C'est à dire qu'au lieu d'appliquer les changements instantanément, ils sont appliqués sur une certaine durée.

```
d3.selectAll("text")
    .transition()
    .duration(750)
    .ease(d3.easeCubicInOut)
        .style("fill", "black")
```

sélection`.transition()` retourne une nouvelle sélection dont les changements vont être progressifs. La méthode `duration(`*ms*`)` spécifie la durée en millisecondes.

La méthode ease permet d'indiquer l'accélération de variation, comme avec gsap. Voir la doc pour la liste détaillée.

Pierre Nerzic

Transitions, suite

On peut enchaîner des transitions successives dans la même instruction.

Un point subtil : seuls les changements situés *après* la méthode `transition` sont concernés.

Dans l'exemple suivant, on fait passer la couleur des textes de blanc à noir, puis à nouveau blanc.

```
d3.selectAll("text")
    .style("fill", "white")
    .transition().duration(500)
        .style("fill", "black")
      .transition().duration(500)
          .style("fill", "white")
```


Pierre Nerzic

Interactions avec l'utilisateur

Pierre Nerzic

Écouteurs sur les éléments

La méthode *sélection*`.on("`*événement*`",` *écouteur*`)` déclare un écouteur sur les éléments de la sélection pour l'événement indiqué. L'événement est n'importe lequel, comme `click`, `mouseover`, etc. L'écouteur est une fonction qui reçoit deux paramètres, l'événement déclencheur et la donnée associée à l'élément concerné (venant de `data()`). Dans cette fonction, `this` est lié à l'élément du DOM.

```
function onClick(event, data) {
  const that = d3.select(this)    // this en tant que sélection
  that.style("fill", "red")
}

d3.selectAll("rect").on("click", onClick)
```

Zoom et panoramique à la souris

Zoomer et cadrer à la souris est simple. Le module `d3-zoom` analyse les actions de l'utilisateur et retourne une transformation, càd une composition de translations et d'homothéties, qu'il faut appliquer aux échelles et aux axes du graphique.

On commence par configurer l'élément `<svg>` pour traiter les événements souris. Ils appelleront la fonction `onZoom` :

```
svg.call(d3.zoom().on("zoom", onZoom))
```

Cette fonction, `onZoom` reçoit la transformation comme ceci :

```
function onZoom(event) {
  // appliquer event.transform aux éléments soumis au zoom
  dessin.attr("transform", event.transform)
}
```

Zoom sur les axes

Quand il y a des axes, c'est un peu plus complexe. Il faut fournir la transformation des échelles aux axes. Mais elle doit être distincte en x et en y.

Cela se fait avec les deux méthodes `transform.rescaleX(fX)` pour l'axe X, et `transform.rescaleY(fY)` pour l'axe Y : 

```
xAxis.call(d3.axisBottom(transform.rescaleX(fX)))
yAxis.call(d3.axisLeft(transform.rescaleY(fY)))
```

Ces méthodes appliquent la transformation limitée à l'axe concerné sur l'échelle.

Techniques avancées

Une fonction est un objet

d3.js emploie énormément de « choses » qui sont des fonctions un peu particulières. En JavaScript une fonction est une sorte d'objet, et donc on peut lui affecter des variables membres ainsi que des méthodes.

```
function F1() {
  console.log(F1.message)
}

F1.message = "intéressant"
F1.affiche = () => console.log("passionnant")

F1()          // affiche "intéressant"
F1.affiche()  // affiche "passionnant"
```

Fonctions internes

```
function F2() {
  function action() {
    console.log(action.message)
  }
  return action
}

const f2 = F2()
f2.message = "étonnant !"
f2()    // appelle la fonction action() et affiche "étonnant !"
```

L'appel à la fonction `F2` retourne une fonction interne, `action`, qui est mise dans la variable `f2`. Ensuite, on réaffecte une variable membre de `f2` avant d'appeler `f2`. C'est étonnant car la fonction `action` trouve la variable `message` dans elle-même, alors que cette variable n'est affectée qu'après.

Fermetures (*closure*)

```
function F3() {
  var message = "défaut"
  function action() {
    console.log(message)
  }
  return action
}

const f3 = F3()
f3()
```

La fonction `F3` retourne une fonction interne, `action`. Or cette dernière emploie une variable qui lui est extérieure, `message`. Dans de tels cas, JavaScript mémorise cette variable dans un environnement qui accompagne la fonction interne, une *fermeture*. Ainsi, `action` a encore accès à la variable.

Accesseurs et modificateurs de fermetures

On peut même définir des *getters* et *setters* pour les variables.

```
function F3() {
  var message = "défaut"
  function action() {
    console.log(message)
  }
  action.getMessage =  () => message
  action.setMessage =  (msg) => message = msg
  return action
}

const f3 = F3()
f3.setMessage("parfait !")
console.log(f3.getMessage())
f3()
```


Pierre Nerzic

Fermetures dans d3.js

De nombreux dispositifs de d3.js sont des fermetures, comme par exemple les sélections, les échelles, les axes, etc.

Ce sont à la fois des objets et des fonctions. Et ces fonctions ont des paramètres qu'on précise avec des setters. Chacun d'eux retourne la même fermeture, de manière à pouvoir les chaîner.

```
const fX = d3.scaleLinear().domain(...).range(...)
```

parce que `d3.scaleLinear()` retourne une fermeture et elle a des setters programmés un peu comme ceci (voir le source) :

```
   scale.domain = (interval) => {
      domain = interval
      return scale
    }
```


Pierre Nerzic

Ajout de méthodes pour une classe

On peut ajouter une fonction à un objet. Elle devient une méthode, mais elle reste spécifique à cet objet.

Pour ajouter une méthode à une classe existante, il faut l'ajouter dans le *prototype*, c'est à dire à la classe elle-même. Le prototype est une sorte d'objet JSON contenant toutes les méthodes et variables de classes qui sont fournies à toutes les instances.

Modifier le prototype d'une classe permet de modifier les méthodes de cette classe, après sa définition.

Ajout si absent

Voici un exemple d'extension de l'API de sélection. La méthode append() ajoute forcément un élément. Cette nouvelle méthode ne l'ajoute que s'il est absent :

```
d3.selection.prototype.appendIfAbsent =
  function(selector, type) {
    const element = this.select(selector)
    return element.empty() ? this.append(type) : element
  }
```

Voici un exemple d'emploi :

```
svg.appendIfAbsent("#title", "text")
    .attr("id", "title")
    .text("Voici le titre")
```


Pierre Nerzic