Basic Maths

primer-computational-mathematics.github.io/book/c_mathematics/

1 Logic and proof

see pdf

2 Sets

-set operations -naturtal numbers -integers -rationals -reals

2.1 Introduction

The notions of a *set* and an *element of a set* are some of the most primitive ones in mathematics so we normally do not even define them. However,

Logic and proof

Contents

(Logic and proof)=

Mathematics for Geoscientists

Logic symbols

The origins of logic symbols stem from first attempts at creating a clear language to express logic statements. Everyday languages are not suitable for this as the use of them can easily lead to ambiguities and paradoxes. This is the case in mathematics as well, so general mathematics was influenced by this language and borrowed some of it. The use of logic symbols is more widespread in areas of mathematics where there is a need for a clear language, for example set theory (which is full of paradoxes!).

Here we will list some more common logic symbols. For a full list visit Wikipedia.

Symbol	Name	Read as	Example	Meaning
$\forall$	universal quantification	for all, for any, for each	$\forall x \in \mathbb{R}, x^2 \geq 0$	For all real numbers x, x^2 is greater or equal to 0
$\exists$	existential quantification	there exists	$\forall n \in \mathbb{N}, \exists x \in \mathbb{R} : \lvert n - x \rvert < 1$	For all natural numbers n there exists a real number x such that $\lvert n - x \rvert < 1$
$\exists!$	uniqueness quantification	there exists exactly one	$A \subseteq \mathbb{N} \implies \exists! m \in A : m \geq n, \forall n \in A$	If A is a subset of $\mathbb{N}$ then there exists a unique $m \in A$ such that $m \geq n$, for all $n \in A$
$\neg$	negation	not	$\neg(x = y) \iff (x \neq y)$	$x = y$ is not true iff $x \neq y$
$\wedge$	conjunction	and	$((x \leq 1) \wedge (x \geq 1)) \iff x = 1$	x is less or equal than 1 and greater or equal to 1 iff $x = 1$
$\vee$	disjunction	or	$((x \leq 0) \vee (x \geq 0)) \iff x \in \mathbb{R}$	$x \leq 0$ or $x \geq 0$ iff x is a real number
$\implies$	implication	implies, if... then	$n \in \mathbb{N} \implies n \in \mathbb{Z}$	If n is element of $\mathbb{N}$ then it is element of $\mathbb{Z}$
$\iff$	equivalence	if and only if, iff	(x is prime) $\wedge (x \mod 2 = 0) \iff x = 2$	$x = 2$ if and only if x is a prime number and $x \mod 2 = 0$
:, $\lvert$, s.t.	condition	such that	$\forall x \in \mathbb{R} : x > 0, \sqrt{x} > 0$	For all x element of $\mathbb{R}$ such that x is greater than 0, $\sqrt{x}$ is greater than 0

Table 1 Common logic symbols

By Imperial College London
© Copyright 2020.

Sets

Contents

Mathematics for Geoscientists

The notions of a *set* and an *element of a set* are some of the most primitive ones in mathematics so we normally do not even define them. However, intuitively, a set represents a collection of objects, which we call its elements. These objects can be anything: numbers, vectors, other sets... The reader might, understandably, be dissatisfied with this circular interpretation of a set as a collection, but hopefully this notebook will provide some intuition about the topic.

Notation.

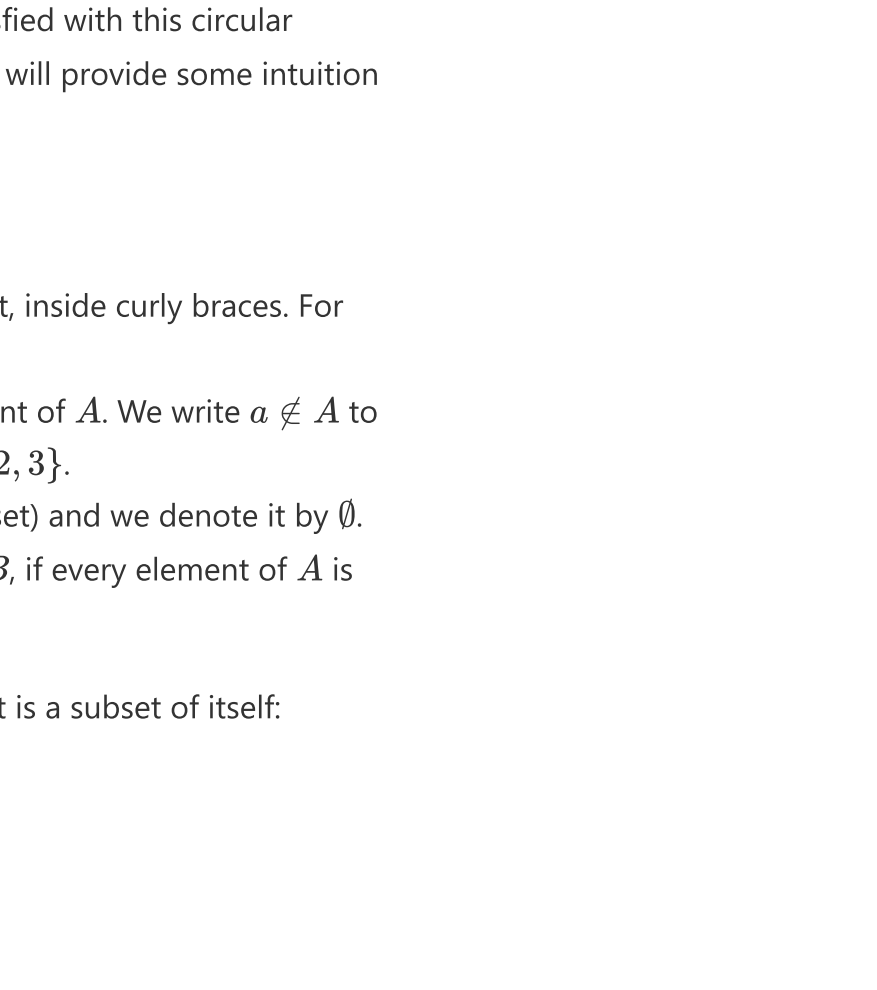
- We denote a set by listing its elements, in any order we want, inside curly braces. For example, $A = \{1, 2, 3\}$
- Let A be a set. We write $a \in A$ to denote that a is an element of A. We write $a \notin A$ to denote that a is not an element of A. For example, $3 \in \{1, 2, 3\}$.
- A set without any elements is called the **empty set** (or *null* set) and we denote it by $\emptyset$.
- Let A and B be sets. We say that A is a **subset** of B, $A \subseteq B$, if every element of A is also an element of B. For example, $\{1, 2\} \subseteq \{1, 2, 3\}$.

By convention, the empty set is a subset of every set and every set is a subset of itself:

- $\emptyset \subseteq A$
- $A \subseteq A$

Set operations

We can extend the logic symbols explained in the Logic notebook to define set operations.



Intersection $\cap$. The intersection of sets A and B is denoted as $A \cap B$ and it corresponds to the logical $\wedge$ ("and"). Its result is a set which contains elements that are common in *both* A and B. That is, $\forall x(x \in A \cap B) \iff ((x \in A) \wedge (x \in B))$. For example, $\{1, 2, 3\} \cap \{1, 3, 5\} = \{1, 3\}$.

Union $\cup$. The union of sets A and B is denoted as A $\cup$ B and it corresponds to the logical $\vee$ ("or"). Its result is a set which contains elements which are elements of A or elements of B, i.e. the two sets are "added" together. That is, $\forall x(x \in A \cup B) \iff ((x \in A) \vee (x \in B))$. For example, $\{1, 2, 3\} \cup \{1, 3, 5\} = \{1, 2, 3, 5\}$.

Complement $\setminus$ or $-$. Similarly to how we can "add" sets together, we can also "subtract" one set from another. We denote this as $A \setminus B$ and the result is a set which contains only elements that are in A but not in B (i.e. elements unique to A). We can write that as $\forall x(x \in A \setminus B) \iff ((x \in A) \wedge (x \notin B))$. For example, $\{1, 2, 3\} \setminus \{1, 3, 5\} = \{2\}$ and $\{1, 2\} \setminus \{1, 2\} = \emptyset$.

Set of natural numbers

In the examples above we considered finite sets, but more often we will deal with infinite sets. We cannot define such sets by listing all its elements, so our understanding of them must derive from their definition.

A set of **natural numbers** $\mathbb{N}$ is one such set whose elements are natural numbers $0, 1, 2, \ldots$, i.e.

$$\mathbb{N} = \{0, 1, 2, \ldots\}.$$

$\mathbb{N}$ is our natural choice for counting as we know the first (smallest) element and for every element we know what its successor is. Therefore, the set $\mathbb{N}$ has a natural ordering $<$ and we can easily define the binary operation of addition $(+)$ on $\mathbb{N}$, since $n + m$ is simply the sum of the n-th and m-th successor of 0.

(Extra) Peano axioms

We have just described properties which form the Peano axioms of $\mathbb{N}$.

Set of integer numbers

We very quickly find the limitations of the set $\mathbb{N}$. For example, the simple equation $n + x = 0$ has no solutions in $\mathbb{N}$ for $n \neq 0$. We call such numbers x an *additive inverse* of n.

To solve such problems we therefore expand the set $\mathbb{N}$ by adding to it additive inverses of all elements of $\mathbb{N}$, noting that 0 is its own additive inverse ($0 + 0 = 0$). By doing this we reached the set of **integer numbers** $\mathbb{Z}$:

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} = -\mathbb{N} \cup \mathbb{N}.$$

We will state without proof, but the reader can easily check this, that the set $\mathbb{Z}$ with the operation $+$ is an algebraic structure with properties:

1. **closure**: $\forall a, b \in \mathbb{Z}, a + b \in \mathbb{Z}$
2. **associativity**: $\forall a, b, c \in \mathbb{Z}, a + (b + c) = (a + b) + c$
3. **neutral element**: $\exists 0 \in \mathbb{Z} : \forall a \in \mathbb{Z}, 0 + a = a + 0 = a$
4. **inverse element**: $\forall a \in \mathbb{Z}, \exists -a \in \mathbb{Z} : a + -a = -a + a = 0$
5. **commutativity**: $\forall a, b \in \mathbb{Z}, a + b = b + a$

We call a set equipped with a binary operation which satisfies properties 1-4 a **group**. If the 5th property is satisfied as well we call it a **commutative (Abelian) group**. For addition, it is the basic algebraic structure in which the equation $n + x = m$ can always be solved.

Set of rational numbers

We can easily define another binary operation on the set $\mathbb{N}$: multiplication $\times$ or $\cdot$ by defining $n \times m = \underbrace{n + n + \cdots + n}_{m}$.

We can extend multiplication to the set $\mathbb{Z}$ as well. We will again state without proof, but the reader can easily check this, that multiplication on $\mathbb{N}$ and $\mathbb{Z}$ is associative, commutative, distributive over addition $(a(b + c) = ab + ac)$ and it has a neutral element 1 such that $n \times 1 = 1 \times n = n$.

Let us now consider the solution to the equation $n \times x = 1$ for $n \in \mathbb{Z}$. We find that the solution exists in $\mathbb{Z}$ only for $n = 1$. The number x for which $nx = 1$ is called the *multiplicative inverse* or the *reciprocal* of n and we denote it by $n^{-1} = \frac{1}{n}$.

To solve a general equation of the form $nx = m$ where $n, m \in \mathbb{Z}, n \neq 0$ we would therefore like to expand the set $\mathbb{Z}$ such that it includes all solutions of that equation. By doing this we have formed the set of **rational numbers** $\mathbb{Q}$:

$$\mathbb{Q} = \{\frac{m}{n}, \quad n, m \in \mathbb{Z}, n \neq 0\}.$$

What we have now achieved is very important. The set of rational numbers without zero equipped with multiplication $(\mathbb{Q}, \times)$ is also a commutative group, so we can always solve the equation of the form $nx = m$. Remember that zero does not have a reciprocal, so division by zero is impossible. Furthermore, multiplication distributes over addition and $(\mathbb{Q}, +)$ is also a commutative group. We call a set with these properties $(\mathbb{Q}, +, \times)$ a **field**.

Set of real numbers

If we were to represent certain numbers on a number line we could find that not all numbers are in $\mathbb{Q}$. The simplest example would be the length of a diagonal of a unit square, i.e. $\sqrt{2}$. There are no $m, n \in \mathbb{Z}$ such that $\frac{m}{n} = \sqrt{2}$, i.e. we cannot construct it as a ratio of integers. We call such numbers *irrational numbers*.

We therefore expand our set $\mathbb{Q}$ to include irrational numbers. By doing that we form a set of **real numbers** and we denote it by $\mathbb{R}$. Now all points on the number line are elements of $\mathbb{R}$.

By Imperial College London
© Copyright 2020.

Functions

Contents

Mathematics for Geoscientists

Basic definitions

Let us give a somewhat informal definition of a function.

Let $\mathcal{D}$ and $\mathcal{C}$ be non-empty sets and let f be a rule by which every element $x \in \mathcal{D}$ is assigned a **unique** element $y \in \mathcal{C}$. The ordered triple $(\mathcal{D}, \mathcal{C}, f)$ is called a **function** (also *map* or *mapping*). We write $f : \mathcal{D} \to \mathcal{C}$.

> **Note**
>
> What that means is that the following three functions are all different despite their assignment f being the same:
>
> $$\begin{aligned} f_1 : \mathbb{R} \to \mathbb{R}, &\quad f_1(x) = x^2 \\ f_2 : \mathbb{R} \to [0, \infty), &\quad f_2(x) = x^2 \\ f_3 : (-\infty, 0) \to (0, \infty), &\quad f_3(x) = x^2 \end{aligned}$$
>
> Functions $(\mathcal{D}_1, \mathcal{C}_1, f_1)$ and $(\mathcal{D}_2, \mathcal{C}_2, f_2)$ are equal iff $\mathcal{D}_1 = \mathcal{D}_2, \mathcal{C}_1 = \mathcal{C}_2, f_1 = f_2$.

- $\mathcal{D}(f)$ is called the **domain** of the function f
- $\mathcal{C}(f)$ is called the **codomain** of the function f
- $f(x)$ is the value (sometimes called output) of f at point x

Example:

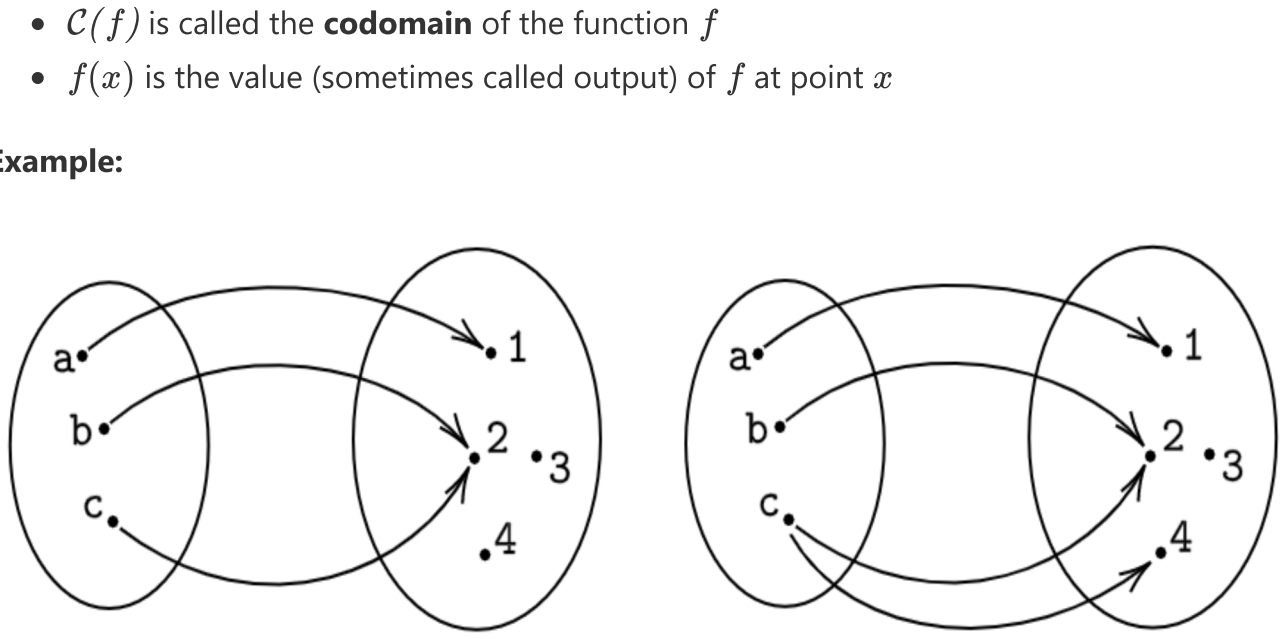


Fig. 1 Left is a function, right is not!

The figure on the left shows a function where $a \mapsto f(a) = 1, b \mapsto f(b) = 2, c \mapsto f(c) = 2$.
The figure on the right does not represent a function since c is assigned both 2 and 4.

Graph

The **graph** of the function is the set

Cartesian (Descartes) product

The Cartesian product $A \times B$ of two sets A and B is the set of all ordered pairs (a, b) where $a \in A, b \in B$. That is,

$$G(f) := \{(x, f(x)) : x \in \mathcal{D}\}.$$

Therefore, $G(f) \subseteq \mathcal{D} \times \mathcal{C} = \{(d, c) : d \in \mathcal{D}, c \in \mathcal{C}\}$.

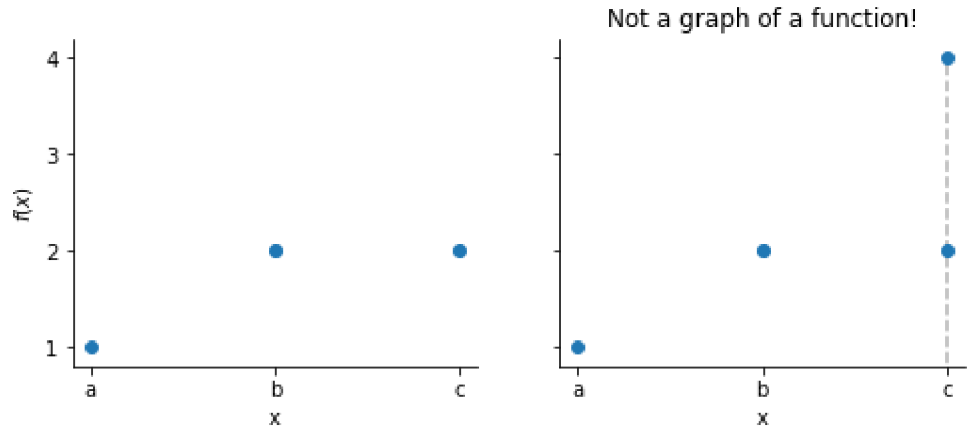
$A \times B = \{(a, b) : a \in A, b \in B\}.$

An example is the Cartesian plane,

$$\mathbb{R}^2 = \mathbb{R} \times \mathbb{R} = \{(x, y) : x, y \in \mathbb{R}\}.$$

From above examples we have the following graphs:

```
import matplotlib.pyplot as plt


fig, ax = plt.subplots(1, 2, figsize=(8, 3), sharey=True)

ax[0].scatter(['a', 'b', 'c'], y = [1, 2, 2])
ax[0].set_ylabel('$f(x)$')

ax[1].scatter(['a', 'b', 'c', 'c'], y = [1, 2, 2, 4], zorder=10)
ax[1].plot(['c', 'c'], [0, 4], '--k', alpha=0.3)
ax[1].set_title('Not a graph of a function!')

for i in [0, 1]:
    ax[i].set_yticks([1, 2, 3, 4])
    ax[i].spines['right'].set_visible(False)
    ax[i].spines['top'].set_visible(False)
    ax[i].set_ylim(0.8, 4.2)
    ax[i].set_xlabel('x')

plt.show()
```



The graph on the right is **not** a graph of a function because two points lie on the vertical line drawn through c, i.e. c is assigned two different values $f(c)$. This is a contradiction to our definition of a function. Sometimes this is called the *vertical line test*.

Restriction

Let $f : \mathcal{D} \to \mathcal{C}$ be a function and let $A \subseteq \mathcal{D}$. We can define a **restriction** of f to the domain of A as $f|_A : A \to \mathcal{C}$ with $f|_A(x) = f(x), \forall x \in A$.

Image

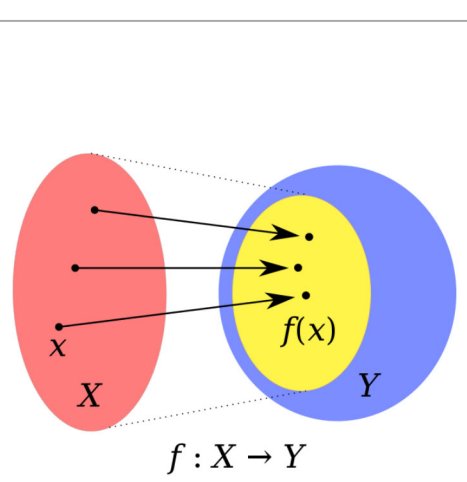
Let $f : \mathcal{D} \to \mathcal{C}$ be a function and let $A \subseteq \mathcal{D}$. The **image of a subset** A under f is the set $f(A) \subseteq \mathcal{C}$ such that:

$$f(A) := \{f(x) : x \in A\} \subseteq \mathcal{C}.$$

If $A = \mathcal{D}$, i.e. the entire domain, we call the set $f(D)$ the **image** or **range** of f. In other words, the image is a set of all values that $f(x)$ can take for all x in the domain.

From our example of a function above (left figure), let us take $A = \{a, c\}$. The image of a subset A is then:

$$f(A) = \{f(x) : x \in A\} = \{\underbrace{f(a)}_{=1}, \underbrace{f(b)}_{=2}\} = \{1, 2\}.$$



$f : X \to Y$

Source: Wikipedia

Similarly for the entire domain:

$$f(D) = \{f(x) : x \in \mathcal{D}\} = \{\underbrace{f(a)}_{=1}, \underbrace{f(b)}_{=2}, \underbrace{f(c)}_{=2}\} = \{1, 2\}.$$

Basic Maths

primer-computational-mathematics.github.io/book/c_mathematics/

1 Logic and proof

see pdf

2 Sets

-set operations -naturtal numbers -integers -rationals -reals

2.1 Introduction

The notions of a *set* and an *element of a set* are some of the most primitive ones in mathematics so we normally do not even define them. However, intuitively, a set represents a collection of objects, which we call its elements. These objects can be anything: numbers, vectors, other sets... The reader might, understandably, be dissatisfied with this circular interpretation of a set as a collection, but hopefully this notebook will provide some intuition about the topic.

Notation.

- We denote a set by listing its elements, in any order we want, inside curly braces. For example, $A = \{1, 2, 3\}$
- Let A be a set. We write $a \in A$ to denote that a is an element of A. We write $a \notin A$ to denote that a is not an element of A. For example, $3 \in \{1, 2, 3\}$.
- A set without any elements is called the **empty set** (or *null* set) and we denote it by $\emptyset$.
- Let A and B be sets. We say that A is a **subset** of B, $A \subseteq B$, if every element of A is also an element of B. For example, $\{1, 2\} \subseteq \{1, 2, 3\}$.

By convention, the empty set is a subset of every set and every set is a subset of itself:

- $\emptyset \subseteq A$
- $A \subseteq A$

2.1 Set operations

Intersection $\cap$

The intersection of sets A and B is denoted as $A \cap B$ and it corresponds to the logical $\wedge$ ("and"). Its result is a set which contains elements that are common in *both* A and B. That is, $\forall x(x \in A \cap B) \iff ((x \in A) \wedge (x \in B))$. For example, $\{1, 2, 3\} \cap \{1, 3, 5\} = \{1, 3\}$.

Union $\cup$

The union of sets A and B is denoted as A $\cup$ B and it corresponds to the logical $\vee$ ("or"). Its result is a set which contains elements which are elements of A or elements of B, i.e. the two sets are "added" together. That is, $\forall x(x \in A \cup B) \iff ((x \in A) \vee (x \in B))$. For example, $\{1, 2, 3\} \cup \{1, 3, 5\} = \{1, 2, 3, 5\}$.

Complement $\setminus$ or $-$

Similarly to how we can "add" sets together, we can also "subtract" one set from another. We denote this as $A \setminus B$ and the result is a set which contains only elements that are in A but not in B (i.e. elements unique to A). We can write that as $\forall x(x \in A \setminus B) \iff ((x \in A) \wedge (x \notin B))$. For example, $\{1, 2, 3\} \setminus \{1, 3, 5\} = \{2\}$ and $\{1, 2\} \setminus \{1, 2\} = \emptyset$.

2.2 Sets

Set of natural numbers

In the examples above we considered finite sets, but more often we will deal with infinite sets. We cannot define such sets by listing all its elements, so our understanding of them must derive from their definition. A set of **natural numbers** $\mathbb{N}$ is one such set whose elements are natural numbers $0, 1, 2, \ldots$, i.e.

$$\mathbb{N} = \{0, 1, 2, \ldots\}.$$

(Extra) Peano axioms

We have just described properties which form the Peano axioms of $\mathbb{N}$. $\mathbb{N}$ is our natural choice for counting as we know the first (smallest) element and for every element we know what its successor is. Therefore, the set $\mathbb{N}$ has a natural ordering $<$ and we can easily define the binary operation of addition ($+$) on $\mathbb{N}$, since $n + m$ is simply the sum of the n-th and m-th successor of 0.

Set of integer numbers

We very quickly find the limitations of the set $\mathbb{N}$. For example, the simple equation $n + x = 0$ has no solutions in $\mathbb{N}$ for $n \neq 0$. We call such numbers x an *additive inverse* of n. To solve such problems we therefore expand the set $\mathbb{N}$ by adding to it additive inverses of all elements of $\mathbb{N}$, noting that 0 is its own additive inverse ($0 + 0 = 0$). By doing this we reached the set of **integer numbers** $\mathbb{Z}$:

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} = -\mathbb{N} \cup \mathbb{N}.$$

We will state without proof, but the reader can easily check this, that the set $\mathbb{Z}$ with the operation $+$ is an algebraic structure with properties:

1. **closure**: $\forall a, b \in \mathbb{Z}, a + b \in \mathbb{Z}$
2. **associativity**: $\forall a, b, c \in \mathbb{Z}, a + (b + c) = (a + b) + c$
3. **neutral element**: $\exists 0 \in \mathbb{Z} : \forall a \in \mathbb{Z}, 0 + a = a + 0 = a$
4. **inverse element**: $\forall a \in \mathbb{Z}, \exists -a \in \mathbb{Z} : a + -a = -a + a = 0$
5. **commutativity**: $\forall a, b \in \mathbb{Z}, a + b = b + a$

We call a set equipped with a binary operation which satisfies properties 1-4 a **group**. If the 5th property is satisfied as well we call it a **commutative (Abelian) group**. For addition, it is the basic algebraic structure in which the equation $n + x = m$ can always be solved.

Set of rational numbers

We can easily define another binary operation on the set $\mathbb{N}$: multiplication $\times$ or $\cdot$ by defining $n \times m = \underbrace{n + n + \cdots + n}_{m}$.

We can extend multiplication to the set $\mathbb{Z}$ as well. We will again state without proof, but the reader can easily check this, that multiplication on $\mathbb{N}$ and $\mathbb{Z}$ is associative, commutative, distributive over addition $(a(b + c) = ab + ac)$ and it has a neutral element 1 such that $n \times 1 = 1 \times n = n$.

Let us now consider the solution to the equation $n \times x = 1$ for $n \in \mathbb{Z}$. We find that the solution exists in $\mathbb{Z}$ only for $n = 1$. The number x for which $nx = 1$ is called the *multiplicative inverse* or the *reciprocal* of n and we denote it by $n^{-1} = \frac{1}{n}$.

To solve a general equation of the form $nx = m$ where $n, m \in \mathbb{Z}, n \neq 0$ we would therefore like to expand the set $\mathbb{Z}$ such that it includes all solutions of that equation. By doing this we have formed the set of **rational numbers** $\mathbb{Q}$:

$$\mathbb{Q} = \{\frac{m}{n}, \quad n, m \in \mathbb{Z}, n \neq 0\}.$$

What we have now achieved is very important. The set of rational numbers without zero equipped with multiplication $(\mathbb{Q}, \times)$ is also a commutative group, so we can always solve the equation of the form $nx = m$. Remember that zero does not have a reciprocal, so division by zero is impossible. Furthermore, multiplication distributes over addition and $(\mathbb{Q}, +)$ is also a commutative group. We call a set with these properties $(\mathbb{Q}, +, \times)$ a **field**.

Set of real numbers

If we were to represent certain numbers on a number line we could find that not all numbers are in $\mathbb{Q}$. The simplest example would be the length of a diagonal of a unit square, i.e. $\sqrt{2}$. There are no $m, n \in \mathbb{Z}$ such that $\frac{m}{n} = \sqrt{2}$, i.e. we cannot construct it as a ratio of integers. We call such numbers *irrational numbers*.

We therefore expand our set $\mathbb{Q}$ to include irrational numbers. By doing that we form a set of **real numbers** and we denote it by $\mathbb{R}$. Now all points on the number line are elements of $\mathbb{R}$.

3 Functions

3.1 Basic definitions

Let us give a somewhat informal definition of a function.

Let $\mathcal{D}$ and $\mathcal{C}$ be non-empty sets and let f be a rule by which every element $x \in \mathcal{D}$ is assigned a **unique** element $y \in \mathcal{C}$. The ordered triple $(\mathcal{D}, \mathcal{C}, f)$ is called a **function** (also *map* or *mapping*). We write $f : \mathcal{D} \to \mathcal{C}$.

{note} What that means is that the following three functions are all different despite their assignment f being the same:

$$\begin{aligned} f_1 : \mathbb{R} \to \mathbb{R}, &\quad f_1(x) = x^2 \\ f_2 : \mathbb{R} \to [0, \infty), &\quad f_2(x) = x^2 \\ f_3 : (-\infty, 0) \to (0, \infty), &\quad f_3(x) = x^2 \end{aligned}$$

Functions $(\mathcal{D}_1, \mathcal{C}_1, f_1)$ and $(\mathcal{D}_2, \mathcal{C}_2, f_2)$ are equal iff $\mathcal{D}_1 = \mathcal{D}_2, \mathcal{C}_1 = \mathcal{C}_2, f_1 = f_2$.

- $\mathcal{D}(f)$ is called the **domain** of the function f
- $\mathcal{C}(f)$ is called the **codomain** of the function f
- $f(x)$ is the value (sometimes called output) of f at point x

Example:

Left is a function, right is not!

The figure on the left shows a function where $a \mapsto f(a) = 1, b \mapsto f(b) = 2, c \mapsto f(c) = 2$. The figure on the right does not represent a function since c is assigned both 2 and 4.

Cartesian (Descartes) product The Cartesian product $A \times B$ of two sets A and B is the set of all ordered pairs (a, b) where $a \in A, b \in B$. That is,

$$A \times B = \{(a, b) : a \in A, b \in B\}.$$

An example is the Cartesian plane,

$$\mathbb{R}^2 = \mathbb{R} \times \mathbb{R} = \{(x, y) : x, y \in \mathbb{R}\}.$$

Graph

The **graph** of the function is the set

$$G(f) := \{(x, f(x)) : x \in \mathcal{D}\}.$$

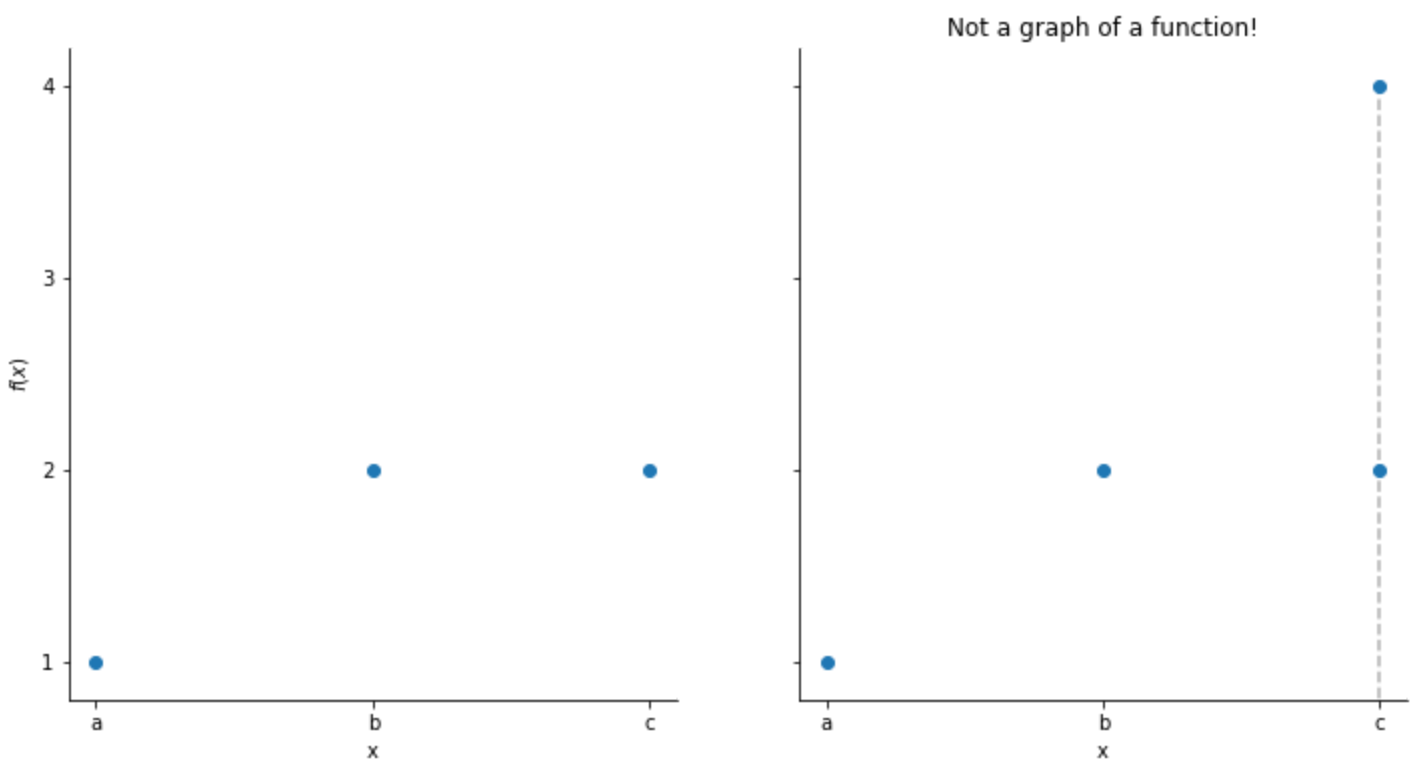
Therefore, $G(f) \subseteq \mathcal{D} \times \mathcal{C} = \{(d, c) : d \in \mathcal{D}, c \in \mathcal{C}\}$.

From above examples we have the following graphs:

In [1]:
```
#works in VSCode
import matplotlib.pyplot as plt


%matplotlib inline
%matplotlib --list
#%matplotlib notebook
```

```
Available matplotlib backends: ['tk', 'gtk', 'gtk3', 'wx', 'qt4', 'qt5', 'qt', 'osx', 'nbagg', 'notebook', 'agg', 'svg', 'pdf', 'ps', 'inline', 'ipympl', 'widget']
```

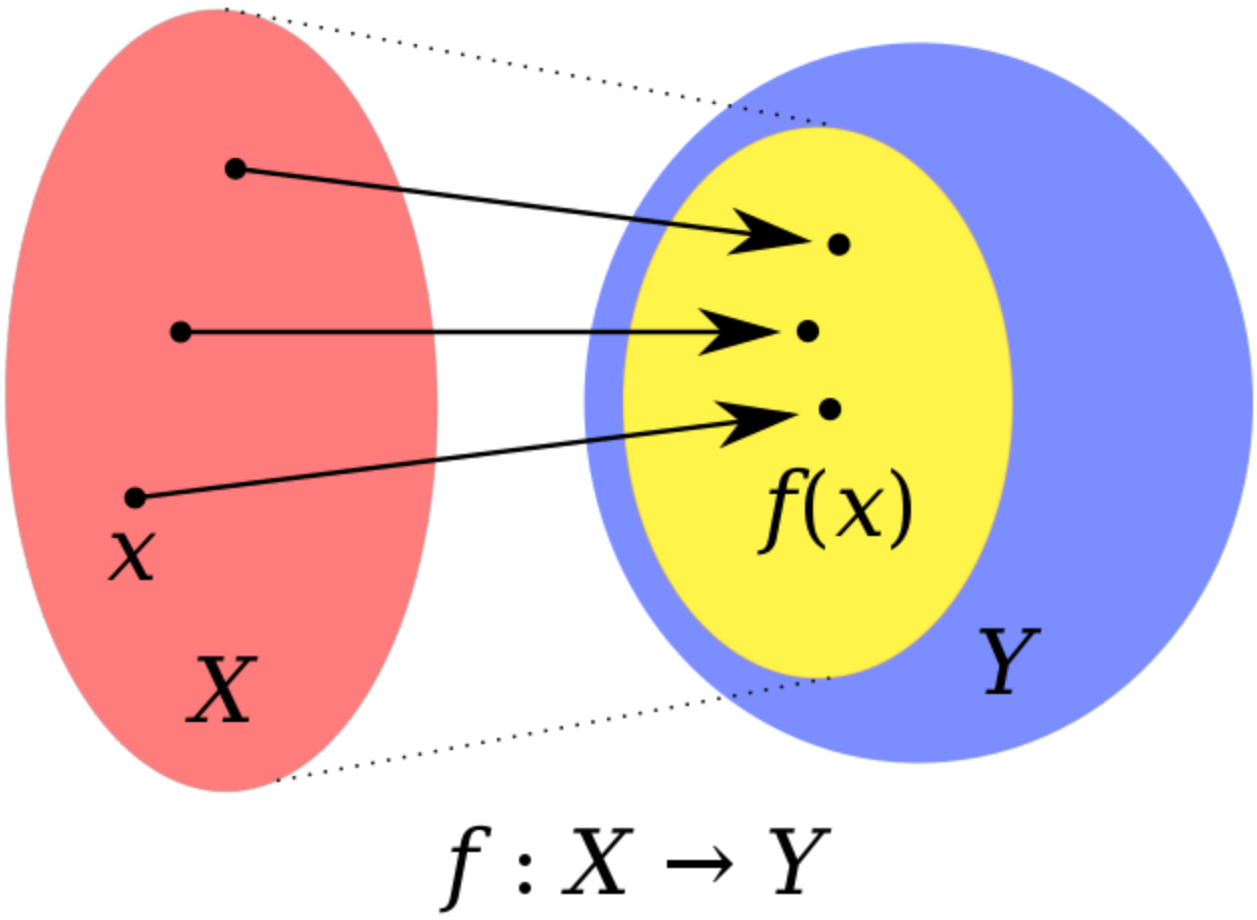
In [2]:
```
fig, ax = plt.subplots(1, 2, figsize=(12, 6), sharey=True)

ax[0].scatter(['a', 'b', 'c'], y = [1, 2, 2])
ax[0].set_ylabel('$f(x)$')

ax[1].scatter(['a', 'b', 'c', 'c'], y = [1, 2, 2, 4], zorder=10)
ax[1].plot(['c', 'c'], [0, 4], '--k', alpha=0.3)
ax[1].set_title('Not a graph of a function!')

for i in [0, 1]:
    ax[i].set_yticks([1, 2, 3, 4])
    ax[i].spines['right'].set_visible(False)
    ax[i].spines['top'].set_visible(False)
    ax[i].set_ylim(0.8, 4.2)
    ax[i].set_xlabel('x')

plt.show()
```



The graph on the right is **not** a graph of a function because two points lie on the vertical line drawn through c, i.e. c is assigned two different values $f(c)$. This is a contradiction to our definition of a function. Sometimes this is called the *vertical line test*.

3.2 Restriction

Let $f : \mathcal{D} \to \mathcal{C}$ be a function and let $A \subseteq \mathcal{D}$. We can define a **restriction** of f to the domain of A as $f|_A : A \to \mathcal{C}$ with $f|_A(x) = f(x), \forall x \in A$.



Source: Wikipedia)

Let $f : \mathcal{D} \to \mathcal{C}$ be a function and let $A \subseteq \mathcal{D}$. The **image of a subset** A under f is the set $f(A) \subseteq \mathcal{C}$ such that:

$$f(A) := \{f(x) : x \in A\} \subseteq \mathcal{C}.$$

If $A = \mathcal{D}$, i.e. the entire domain, we call the set $f(D)$ the **image** or **range** of f. In other words, the image is a set of all values that $f(x)$ can take for all x in the domain. From our example of a function above (left figure), let us take $A = \{a, c\}$. The image of a subset A is then:

$$f(A) = \{f(x) : x \in A\} = \{\underbrace{f(a)}_{=1}, \underbrace{f(b)}_{=2}\} = \{1, 2\}.$$

Similarly for the entire domain:

$$f(D) = \{f(x) : x \in \mathcal{D}\} = \{\underbrace{f(a)}_{=1}, \underbrace{f(b)}_{=2}, \underbrace{f(c)}_{=2}\} = \{1, 2\}.$$

3.3 Inverse image

Let $f : \mathcal{D} \to \mathcal{C}$ be a function and let $E \subseteq \mathcal{C}$. The **inverse image** (or *preimage*) of E under function f is the set

$$f^{-1}(E) := \{x \in \mathcal{D} : f(x) \in E\} \subseteq \mathcal{D}.$$

If we take $E = \mathcal{C}$, then $f^{-1}(\mathcal{C}) = \mathcal{D}$. In other words, we are looking for the values $x \in \mathcal{D}$ for which $f(x)$ takes specified values.

Again from the example above, let us consider $E_1 = \{1\}, E_2 = \{2\}, E_3 = \{1, 2\}$:

$$f^{-1}(E_1) = \{x \in \mathcal{D} : f(x) \in E_1\} = \{x \in \mathcal{D} : f(x) = 1\} = \{a\}$$
$$f^{-1}(E_2) = \{x \in \mathcal{D} : f(x) \in E_2\} = \{x \in \mathcal{D} : f(x) = 2\} = \{b, c\}$$

Or some may appreciate a more applied example: let $f : \mathbb{R} \to \mathbb{R}$ be a function with $f(x) = x^2$, then

$$f([0, \infty)) = [0, \infty), \qquad f((-\infty, 0]) = [0, \infty)$$
$$f^{-1}([0, \infty)) = \mathbb{R}, \qquad f^{-1}((-\infty, 0)) = \emptyset.$$

The last point means that there are no $x \in \mathbb{R}$ such that $f(x) = x^2$ is a negative number.

Or a more applied example: consider $f : \mathbb{R} \to \mathbb{R}$ where $f(x) = x - 5$ (we can simply write $x \mapsto x - 5$) and $A = (-3, 1]$. Let us find the preimage $f^{-1}(A)$ graphically and by calculating it.

$$\begin{aligned} f^{-1}(A) &= \{x \in \mathbb{R} : f(x) \in A\} \\ &= \{x \in \mathbb{R} : x - 5 \in (-3, 1]\} \\ &= \{x \in \mathbb{R} : -3 < x - 5 \leq -1\} \end{aligned}$$

$$\begin{aligned} x - 5 > -3 \quad &\text{and} \quad x - 5 \leq -1 \\ x > 2 \quad &\text{and} \quad x \leq 4 \end{aligned}$$

$$\begin{aligned} &\implies x \in (2, 4] \\ &\implies f^{-1}(A) = (2, 4] \end{aligned}$$

This agrees with our findings on the graph.

3.4 Real functions

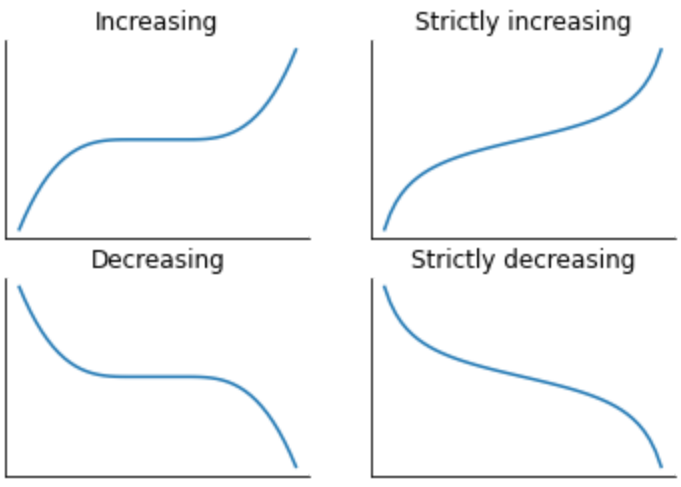
If $\mathcal{D}$ and $\mathcal{C}$ are subsets of $\mathbb{R}$ then we call a function $f : \mathcal{D} \to \mathcal{C}$ a **real-valued function of a real variable** or, simply, a real function. The largest subset of $\mathbb{R}$ for which the function f is defined is called the **natural domain**.

Monotonic functions

Many useful properties of real functions can easily be seen on the graph of the function. One of those is whether a function is **monotonic**, i.e. whether it is always increasing or decreasing on the whole domain. For a function $f : A \to \mathbb{R}, A \subseteq \mathbb{R}$, we say that it is:

- **increasing** on A if $(x_1 < x_2) \Rightarrow (f(x_1) \leq f(x_2))$ for all $x_1, x_2 \in A$
- **strictly increasing** on A if $(x_1 < x_2) \Rightarrow (f(x_1) < f(x_2))$ for all $x_1, x_2 \in A$
- **decreasing** on A if $(x_1 < x_2) \Rightarrow (f(x_1) \geq f(x_2))$ for all $x_1, x_2 \in A$
- **strictly decreasing** on A if $(x_1 < x_2) \Rightarrow (f(x_1) > f(x_2))$ for all $x_1, x_2 \in A$

In [3]:
```
import numpy as np

def f(x):
    if x <= 0:
        y = x**3 + 1 # +1 to offset and make a plateau
    elif x <= 1:     # y equals 1 for x between 0 and 1
        y = 1
    else:
        y = (x-1)**3 + 1 # -1 inside the cubed term shifts it to the right
    return y


titles = [['Increasing', 'Strictly increasing'],
          ['Decreasing', 'Strictly decreasing']]

x1 = np.linspace(-2, 3, 50) # 50 points between -2 and 3
x2 = np.linspace(-1.3, 1.3, 50) # 50 points between -1.3 and 1.3

fig, ax = plt.subplots(2, 2)

ax[0, 0].plot(x1, list(map(f, x1))) # map applies f to each element of x1
ax[0, 1].plot(x2, np.tan(x2)) # np.tan applies tan to each element of x2
ax[1, 0].plot(x1, -np.array(list(map(f, x1)))) # np.array converts the map object to an array
ax[1, 1].plot(x2, -np.tan(x2)) # applies negative tan to each element of the x2 array

for i in range(2):
    for j in range(2):
        ax[i, j].spines['right'].set_visible(False)
        ax[i, j].spines['top'].set_visible(False)
        ax[i, j].set_title(titles[i][j])
        ax[i, j].set_xticks([])
        ax[i, j].set_yticks([])

plt.show()
```



The map function is a useful tool for applying a function to each element of an iterable without having to write a loop. It can be used with any iterable, including lists, tuples, and strings. However, it's important to note that the map function returns a new iterable, so if you want to use the results more than once, you should convert the iterable to a list or tuple.

Overall, the map function is a powerful tool for working with iterables in Python, and can help simplify code that involves applying a function to each element of a list or other iterable.

Even and odd functions

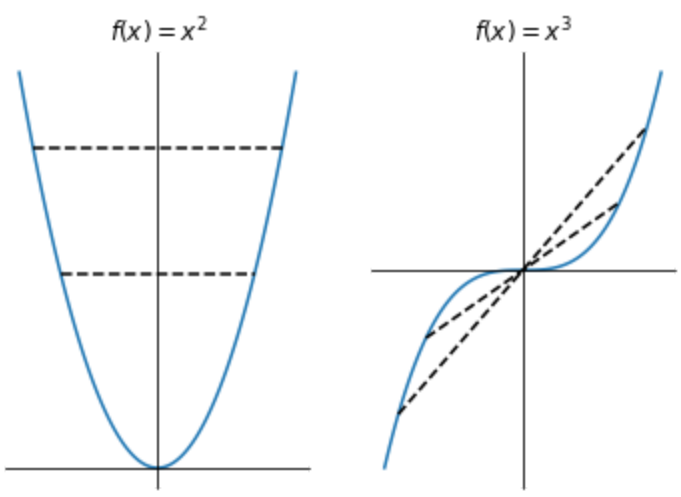
Even and odd functions satisfy particular symmetry relations. A function $f : A \to \mathbb{R}, A \subseteq \mathbb{R}$ is:

- **even** on A if $f(-x) = f(x)$ for all $x \in A$
- **odd** on A if $f(-x) = -f(x)$ for all $x \in A$

The names 'even' and 'odd' originate from the **power function** $f(x) = x^n$, which is even for even n and odd for odd n. For example, $f(x) = x^2$ and $f(x) = x^3$ below:

In [4]:
```
x = np.linspace(-2, 2, 100)
y1 = x**2
y2 = x**3 / 2

fig, ax = plt.subplots(1, 2)

ax[0].plot(x, y1)
ax[1].plot(x, y2)

ax[0].plot([-1.4, 1.4], [1.4**2, 1.4**2], '--k')
ax[0].plot([-1.8, 1.8], [1.8**2, 1.8**2], '--k')
ax[1].plot([-1.4, 1.4], [-1.4**3 / 2, 1.4**3 /2], '--k')
```

```
ax[1].plot([-1.8, 1.8], [-1.8**3 / 2, 1.8**3 /2], '--k')

ax[0].set_title(r'$f(x) = x^2$')
ax[1].set_title(r'$f(x) = x^3$')

for i in range(2):
    ax[i].spines['left'].set_position('zero')
    ax[i].spines['bottom'].set_position('zero')
    ax[i].spines['right'].set_visible(False)
    ax[i].spines['top'].set_visible(False)
    ax[i].set_yticks([])
    ax[i].set_xticks([])

plt.show()
```



4 Elementary functions

-polynomial -rational -exponential -hyperbolic -trigonometric -roots -logarithmic functions

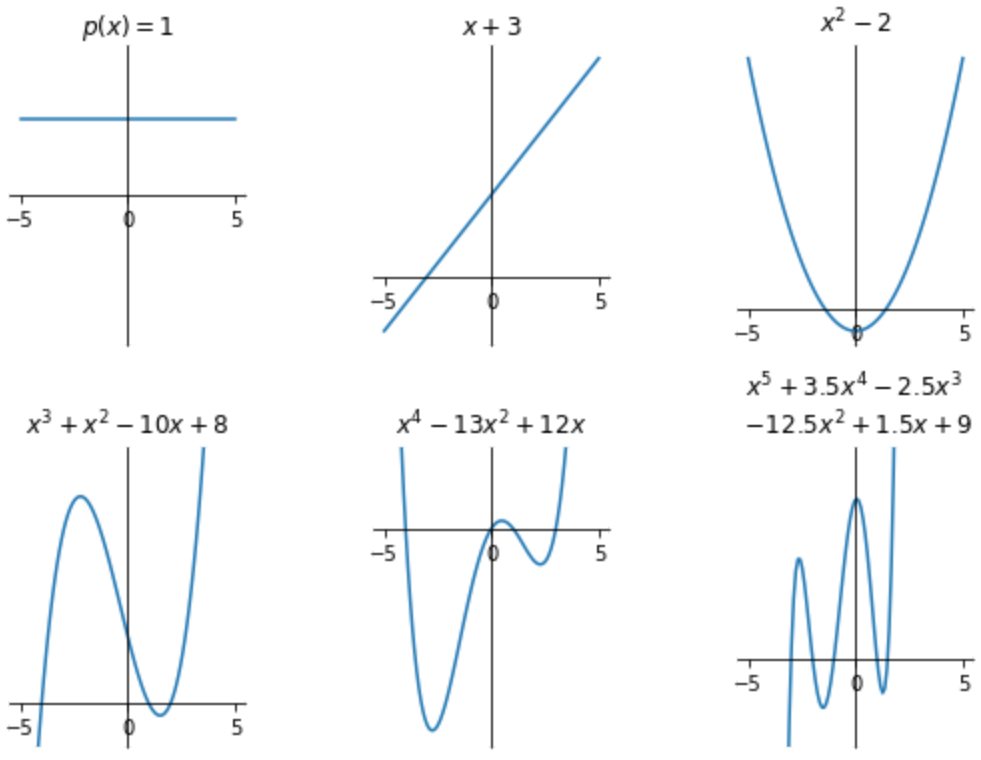
4.1 Polynomial

A polynomial of degree $n \in \mathbb{N}$ is a real function of a real variable given by the formula:

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

where $a_0, a_1, \ldots, a_n \in \mathbb{R}$ are the polynomial coefficients such that $a_n \neq 0$. The natural domain of polynomial functions is the entire $\mathbb{R}$.

In [5]:
```
# import numpy as np
# import matplotlib.pyplot as plt


x = np.linspace(-5, 5, 100) # 100 evenly spaced numbers over [-5, 5]
titles = [['$p(x)=1$', '$x + 3$' ,'$x^2 - 2$'],
          ['$x^3 + x^2 - 10x + 8$', '$x^4 - 13x^2 + 12x$' ,'$x^5 + 3.5x^4 - 2.5x^3$ \n $- 12.5x^2 + 1.5x + 9$']]

fig, ax = plt.subplots(2, 3, figsize=(8, 6)) # 2 rows, 3 columns
fig.tight_layout(pad=3.) # adjust spacing between subplots

ax[0, 0].plot(x, [1] * len(x)) # constant function
ax[0, 1].plot(x, x + 3) # linear function
ax[0, 2].plot(x, x**2 - 2)  # quadratic function
ax[1, 0].plot(x, x**3 + x**2 - 10*x + 8)    # cubic function
ax[1, 1].plot(x, x**4 - 13*x**2 + 12*x) # quartic function
ax[1, 2].plot(x, x**5 + 3.5*x**4 - 2.5*x**3 - 12.5*x**2 + 1.5*x + 9)    # quintic function


ax[0, 0].set_ylim(-2, 2)
ax[1, 0].set_ylim(-5, 30)
ax[1, 1].set_ylim(-80, 30)
ax[1, 2].set_ylim(-5, 12)

for i in range(2): # row
    for j in range(3): # column
        ax[i, j].spines['left'].set_position('zero') # set y spine to x=0
        ax[i, j].spines['bottom'].set_position('zero') # set x spine to y=0
        ax[i, j].spines['right'].set_visible(False) # turn off right spine
        ax[i, j].spines['top'].set_visible(False) # turn off top spine
        ax[i, j].set_title(titles[i][j]) # set title
        ax[i, j].set_yticks([]) # turn off y ticks

plt.show()
```



Linear functions

Quadratic functions

A function given by the formula $f(x) = ax^2 + bx + c, a \neq 0$ is called the **quadratic function**. We call its graph a **parabola**. The roots of a quadratic function are given by:

The quadratic function can be written in vertex form as $f(x) = a\left(x + \frac{b}{2a}\right) + c - \frac{b^2}{4a}$ from which we see that its vertex position is

$$\left(-\frac{b}{2a}, -\frac{D}{4a}\right),$$

where the number

$$D := b^2 - 4ac$$

is called the *discriminant* of a quadratic function. The discriminant is important because it tells us about the null-points (roots) of the function, which are given by

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-b \pm \sqrt{D}}{2a}.$$

Therefore, if:

- $D < 0$, no real roots, i.e. $x_1, x_2 \notin \mathbb{R}$
- $D = 0$, one real root i.e. $x_1, x_2 \in \mathbb{R} : x_1 = x_2$
- $D > 0$, two real roots, i.e. $x_1, x_2 \in \mathbb{R} : x_1 \neq x_2$

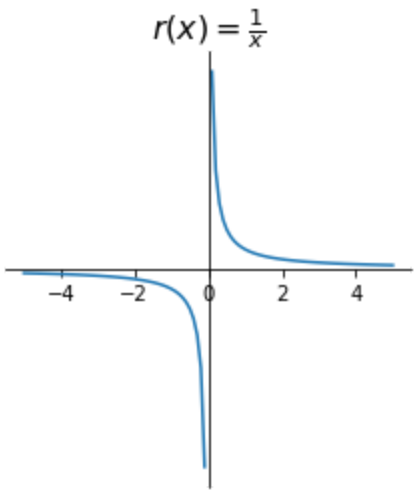
4.2 Rational

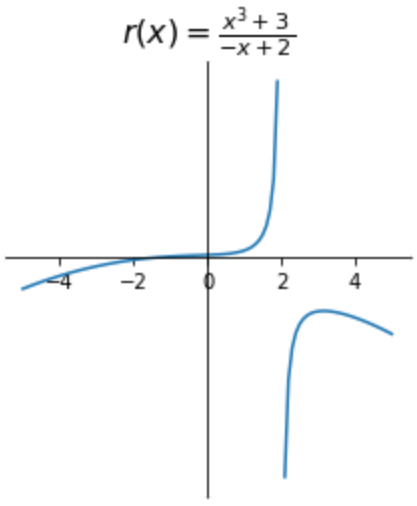
A **rational function** is a real function of a real variable given by the formula:

$$r(x) = \frac{p(x)}{q(x)},$$

where p and q are real polynomials. The natural domain of $r(x)$ is the entire $\mathbb{R}$ without the null-points of $q(x)$ (we cannot divide by zero). That is, $r(x)$ is defined on $\mathbb{R} \backslash \{x \in \mathbb{R} : q(x) = 0\}$.

In [6]:
```python
# for demonstration purposes will suppress warnings when div by 0
import warnings; warnings.simplefilter('ignore')


x = np.linspace(-5, 5, 101)

fig, ax = plt.subplots(1, 2, figsize=(8, 4))

ax[0].plot(x, 1/x)
ax[0].set_title(r'$r(x) = \frac{1}{x}$', fontsize=16)

ax[1].plot(x, (x**3 + 3)/(-x + 2))
ax[1].set_title(r'$r(x) = \frac{x^3 + 3}{-x + 2}$', fontsize=16)

for i in range(2):
    ax[i].spines['left'].set_position('zero')
    ax[i].spines['bottom'].set_position('zero')
    ax[i].spines['right'].set_visible(False)
    ax[i].spines['top'].set_visible(False)
    ax[i].set_yticks([])

plt.show()
```





4.3 Exponential

An **exponential function** is a function of the form

$$f(x) = a^x, \qquad a \in \mathbb{R}^+ \backslash \{1\},$$

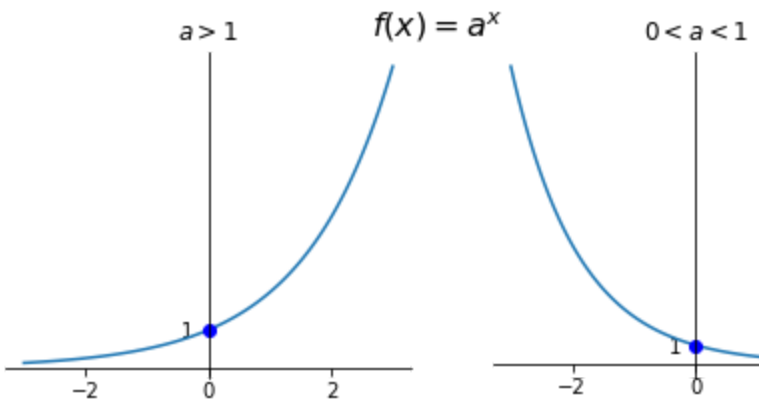
where a is called the *base* and is a positive real number different from 1. A special case of an exponential function is the **natural** exponential function $f(x) = e^x$ where the base is $e \approx 2.718282\ldots$ (Euler's number).

The natural domain of an exponential function is the entire set $\mathbb{R}$ and its image is $(0, +\infty)$, i.e. $a^x > 0, \forall x \in \mathbb{R}$. Exponential function $x \mapsto a^x$ always crosses the y-axis at point $(0, 1)$ since $a^0 = 1$.

Exponential function $f(x) = a^x$ is:

- strictly increasing if $a > 1$
- strictly decreasing if $0 < a < 1$

In [7]:
```
x = np.linspace(-3, 3, 100)

fig, ax = plt.subplots(1, 2, figsize=(8, 3))
fig.suptitle(r'$f(x) = a^x$', fontsize=16)

ax[0].plot(x, 2**x)
ax[0].set_title(r'$a > 1$')

ax[1].plot(x, 0.4**x)
ax[1].set_title(r'$0 < a < 1$')

for i in range(2):
    ax[i].plot(0, 1, 'bo')
    ax[i].spines['left'].set_position('zero')
    ax[i].spines['bottom'].set_position('zero')
    ax[i].spines['right'].set_visible(False)
    ax[i].spines['top'].set_visible(False)
    ax[i].set_yticks([1])

plt.show()
```



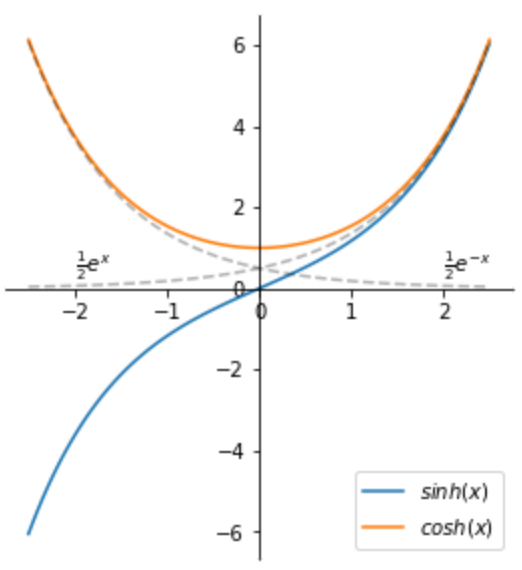
4.4 Hyperbolic functions

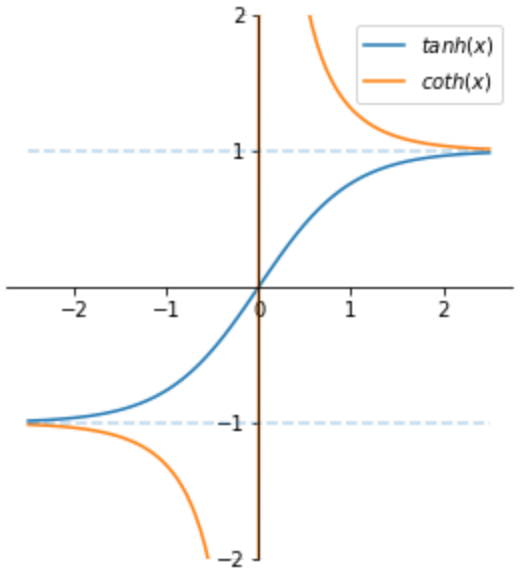
Using the exponential function we define **hyperbolic functions** as follows:

$$\sinh x := \frac{e^x - e^{-x}}{2}$$
$$\cosh x := \frac{e^x + e^{-x}}{2}$$
$$\tanh x := \frac{\sinh x}{\cosh x} = \frac{e^{2x} - 1}{e^{2x} + 1}$$
$$\coth x := \frac{\cosh x}{\sinh x} = \frac{e^{2x} + 1}{e^{2x} - 1}$$

In [8]:
```
x = np.linspace(-2.5, 2.5, 100)

fig, ax = plt.subplots(1, 2, figsize=(10, 5))

ax[0].plot(x, np.sinh(x), label=r'$sinh(x)$')
ax[0].plot(x, np.cosh(x), label=r'$cosh(x)$')

ax[0].plot(x, np.exp(-x) / 2, '--k', alpha=0.3)
```

```
ax[0].plot(x, np.exp(x)/2, '--k', alpha=0.3)
ax[0].text(-2, 0.4, r'$\frac{1}{2}e^x$')
ax[0].text(2, 0.4, r'$\frac{1}{2}e^{-x}$')

ax[1].plot(x, np.tanh(x), label=r'$tanh(x)$')
ax[1].plot(x, 1 / np.tanh(x), label=r'$coth(x)$')

ax[1].set_ylim(-2, 2)
ax[1].hlines(1, -2.5, 2.5, linestyles='--', alpha=0.3)
ax[1].hlines(-1, -2.5, 2.5, linestyles='--', alpha=0.3)
ax[1].set_yticks([-2, -1, 1, 2])

for i in range(2):
    ax[i].spines['left'].set_position('zero')
    ax[i].spines['bottom'].set_position('zero')
    ax[i].spines['right'].set_visible(False)
    ax[i].spines['top'].set_visible(False)
    ax[i].legend(loc='best')

plt.show()
```





Properties

	$\sinh x$	$\cosh x$	$\tanh x$	$\coth x$
Domain	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}\backslash\{0\}$
Image	$\mathbb{R}$	$(1, +\infty)$	$(-1, 1)$	$(-\infty, -1) \cup (1, +\infty)$
Monotonic?	strictly increasing	strictly decreasing on $(-\infty, 0]$, strictly increasing on $[0, +\infty)$	strictly increasing	strictly decreasing on $(-\infty, 0)$ and $(0, +\infty)$
Even/odd?	Odd	Even	Odd	Odd

Similar to the Pythagorean trigonometric identity, the principal formula for hyperbolic functions is:

$$\cosh^2 x - \sinh^2 x = 1.$$

Also similar are the argument addition and subtraction formulae:

$$\sinh(x \pm y) = \sinh x \cosh y \pm \cosh x \sinh y$$

$$\cosh(x \pm y) = \cosh x \cosh y \pm \sinh x \sinh y$$

4.5 Trigonometric functions

The **radian** is a measurement of the angle formed by two radii of a unit circle connecting the circle arc of unit length. $2\pi rad = 360^\circ$

A function $f : \mathcal{D} \to \mathbb{R}$ is **periodic** with a period $\tau > 0$ if for all $x \in \mathcal{D}$:

1. $x + \tau \in \mathcal{D}$
2. $f(x + \tau) = f(x)$.

Note that if τ is a period of a function, any multiple of it is also a period of the same function. We therefore define the **fundamental period** (also primitive, basic and prime period) as the smallest positive period. The six trigonometric functions are **sine, cosine, tangent** and their inverses **cosecant, secant, cotangent**, respectively. We can define them for acute angle θ in right-angled triangles as follows:

$$\sin\theta = \frac{\text{opposite}}{\text{hypotenuse}}, \quad \cos\theta = \frac{\text{adjacent}}{\text{hypotenuse}}, \quad \tan\theta = \frac{\text{opposite}}{\text{adjacent}}$$

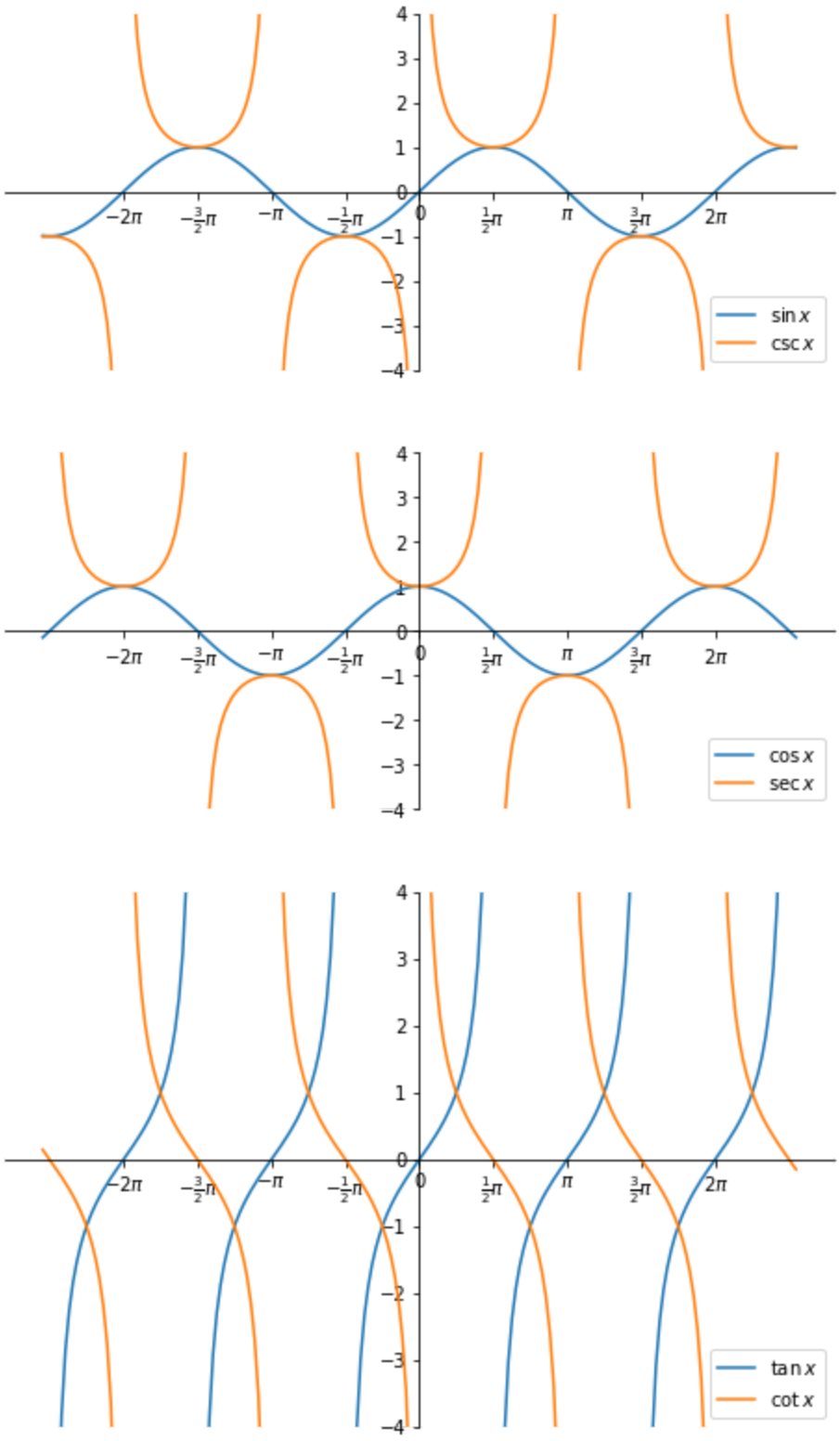
And their inverses:

$$\csc\theta = \frac{\text{hypotenuse}}{\text{opposite}}, \quad \sec\theta = \frac{\text{hypotenuse}}{\text{adjacent}}, \quad \cot\theta = \frac{\text{adjacent}}{\text{opposite}}$$

To extend the definitions of trigonometric functions to the entire set $\mathbb{R}$ we could consider a unit circle centered at the origin. The x and y coordinates of a point on a circle is then $\cos\theta$ and $\sin\theta$, respectively, where θ is the angle between the x-axis and the line joining the point and the origin. Using sine and cosine we can then define the other four functions.

In [9]:
```
x = np.linspace(-8, 8, 200)

fig, ax = plt.subplots(3, 1, figsize=(8, 14), gridspec_kw={'height_ratios': [2, 2, 3]})

_y = 1. / np.sin(x)
_y[:-1][np.abs(np.diff(_y)) > 10] = np.nan  # avoid joining lines
ax[0].plot(x, np.sin(x), label=r'$\sin x$')
ax[0].plot(x, _y, label=r'$\csc x$')
```

```
_y = 1. / np.cos(x)
_y[:-1][np.abs(np.diff(_y)) > 10] = np.nan
ax[1].plot(x, np.cos(x), label=r'$\cos x$')
ax[1].plot(x, _y, label=r'$\sec x$')

_y = np.tan(x)
_y[:-1][np.diff(_y) < 0] = np.nan
ax[2].plot(x, _y, label=r'$\tan x$')

_y = 1. / np.tan(x)
_y[:-1][np.diff(_y) > 0] = np.nan
ax[2].plot(x, _y, label=r'$\cot x$')

for i in range(3):
    ax[i].spines['left'].set_position('zero')
    ax[i].spines['bottom'].set_position('zero')
    ax[i].spines['right'].set_visible(False)
    ax[i].spines['top'].set_visible(False)
    ax[i].legend(loc='lower right')
    ax[i].set_xticks(np.pi * np.array([-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2]))
    ax[i].set_xticklabels(['$-2\pi $', r'$-\frac{3}{2} \pi $', r'$-\pi$',
                           r'$-\frac{1}{2}\pi$', '0', r'$\frac{1}{2}\pi$',
                           r'$\pi$', r'$\frac{3}{2}\pi$', r'$2\pi$'])
    ax[i].set_ylim(-4, 4)

plt.show()
```

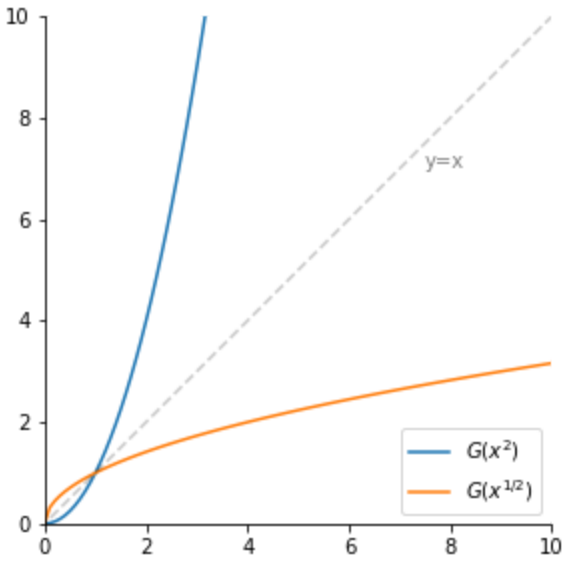


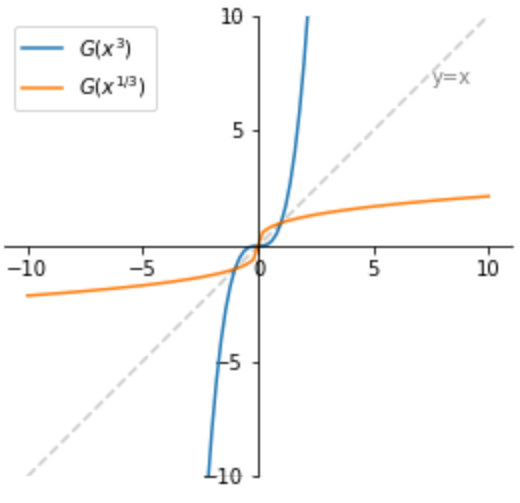
Now that we know what an inverse function is, let us think about the inverse functions of elementary functions we explored in a previous notebook.

4.6 Roots

In [10]:
```
# import numpy as np    #redundant import
# import matplotlib.pyplot as plt  #REDUNDANT INPUT


x1 = np.linspace(0, 10, 101)
x2 = np.linspace(-10, 10, 101)

fig, ax = plt.subplots(1, 2, figsize=(10, 5))

ax[0].plot(x1, x1**2, label=r'$G(x^2)$')
ax[0].plot(x1, np.sqrt(x1), label=r'$G(x^{1/2})$')
ax[0].set_xlim(0, 10)
ax[0].set_ylim(0, 10)

ax[1].plot(x2, x2**3, label=r'$G(x^3)$')
ax[1].plot(x2, np.cbrt(x2), label=r'$G(x^{1/3})$')
```

```
ax[1].set_ylim(-10, 10)
ax[1].set_yticks([-10, -5, 5, 10])

for i in range(2):
    ax[i].plot(x2, x2, '--k', alpha=0.2)
    ax[i].text(7.5, 7, 'y=x', alpha=0.5)
    ax[i].set_aspect('equal')
    ax[i].spines['left'].set_position('zero')
    ax[i].spines['bottom'].set_position('zero')
    ax[i].spines['right'].set_visible(False)
    ax[i].spines['top'].set_visible(False)
    ax[i].legend(loc='best')

plt.show()
```





4.7 Logarithmic functions

The exponential function $f : \mathbb{R} \to (0, +\infty), f(x) = a^x$ is an injection because it is strictly increasing and it is a surjection because its image is $f(\mathbb{R}) = (0, +\infty)$. Therefore, it is a bijection and the inverse function $f^{-1} : (0, +\infty) \to \mathbb{R}$ exists.

We denote this inverse function as $\log_a x := f^{-1}(x)$ and we call it the **logarithm with base a**.

We call a logarithm with base e the **natural logarithm** and define it as:

$$\ln x := \log_e x$$

{admonition} Properties

Domain: $(0, +\infty)$, range: $\mathbb{R}$

$$\log_a x = y \implies x = a^y$$

$$(f \circ f^{-1})(y) = a^{\log_a y} = y, \forall y > 0$$

$$(f^{-1} \circ f)(x) = \log_a a^x = x, \forall x \in \mathbb{R}$$

For all $x, y \in (0, +\infty), a, b, c \in (0, 1) \cup (1, +\infty)$:

$$\log_a 1 = 0$$

Since the exponential function maps a sum to a product, i.e. $a^{x+y} = a^x a^y$, the logarithmic function maps a product to a sum:

$$\log_a(xy) = \log_a x + \log_a y \tag{1}$$

$$\log_a \frac{x}{y} = \log_a x - \log_a y \tag{2}$$

$$\log_a x^\alpha = \alpha \log_a x \tag{3}$$

$$\log_{a^\beta} x = \frac{1}{\beta} \log_a x \tag{4}$$

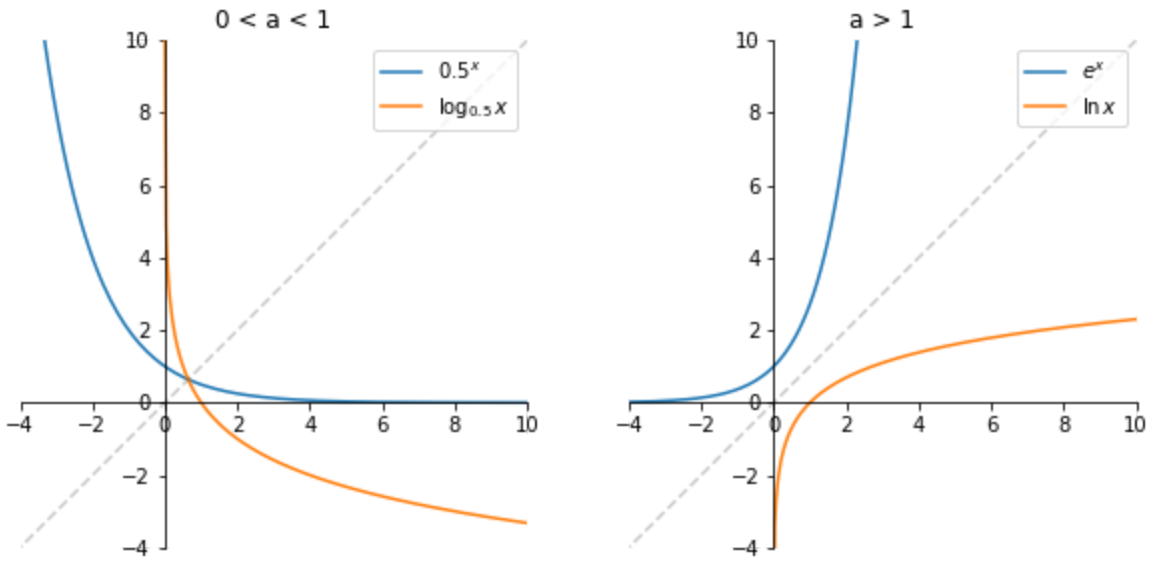
Change of base:

$$\log_b c = \frac{\log_a c}{\log_a b}$$

In [11]:

```
x1 = np.linspace(-10, 10, 201)
x2 = np.linspace(0.001, 10, 301)

fig, ax = plt.subplots(1, 2, figsize=(10, 5))

ax[0].plot(x1, 0.5**x1, label=r'$0.5^x$')
ax[0].plot(x2, np.log(x2)/np.log(0.5), label=r'$\log_{0.5}x$')
ax[0].set_title('0 < a < 1')

ax[1].plot(x1, np.exp(x1), label=r'$e^x$')
ax[1].plot(x2, np.log(x2), label=r'$\ln x$')
ax[1].set_title('a > 1')

for i in range(2):
    ax[i].plot(x1, x1, '--k', alpha=0.2)
    ax[i].set_aspect('equal')
    ax[i].spines['left'].set_position('zero')
    ax[i].spines['bottom'].set_position('zero')
    ax[i].spines['right'].set_visible(False)
```

```
    ax[i].spines['top'].set_visible(False)
    ax[i].legend(loc='upper right')
    ax[i].set_ylim(-4, 10)
    ax[i].set_xlim(-4, 10)
```



Inverse trigonometric functions

Inverse trigonometric functions are also called **arcus functions** so we denote them with the prefix arc-, e.g. $\arcsin x$. They are also often denoted as $\sin^{-1} x$. Periodic functions are not injections so they do not have inverses. Therefore, we need to restrict them to parts of their natural domain where they are strictly monotonic - here they are injections. Then trigonometric functions with the following domain restrictions are bijections:

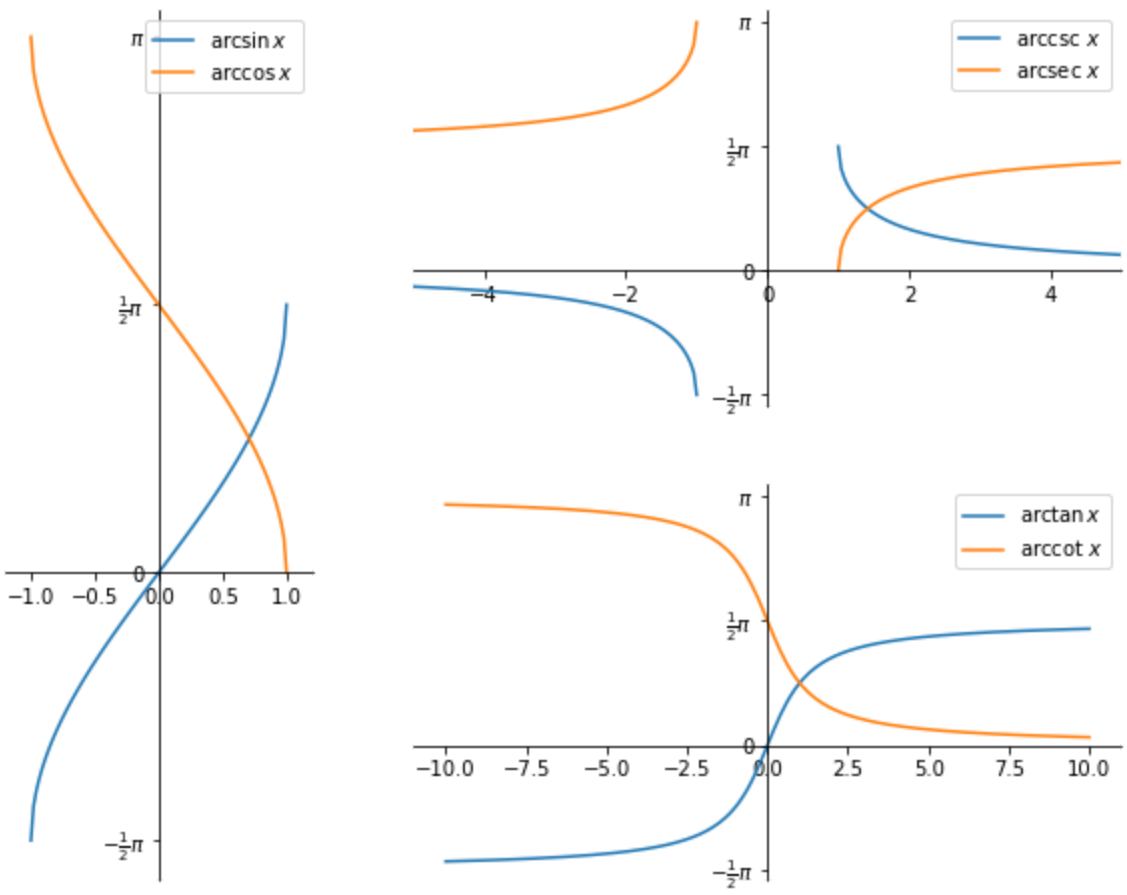
$$
\begin{aligned}
&\sin|_{[-\frac{\pi}{2},\frac{\pi}{2}]} : [-\frac{\pi}{2}, \frac{\pi}{2}] \to [-1, 1] \\
&\cos|_{[0,\pi]} : [0, \pi] \to [-1, 1] \\
&\tan|_{(-\frac{\pi}{2},\frac{\pi}{2})} : (-\frac{\pi}{2}, \frac{\pi}{2}) \to \mathbb{R} \\
&\csc|_{(-\frac{\pi}{2},0)\cup(0,\frac{\pi}{2})} : (-\frac{\pi}{2}, 0) \cup (0, \frac{\pi}{2}) \to (-\infty, -1) \cup (-1, +\infty) \\
&\sec|_{(0,\frac{\pi}{2})\cup(\frac{\pi}{2},\pi)} : (0, \frac{\pi}{2}) \cup (\frac{\pi}{2}, \pi) \to (-\infty, -1) \cup (-1, +\infty) \\
&\cot|_{(0,\pi)} : (0, \pi) \to \mathbb{R}
\end{aligned}
$$

so we can define their inverse functions:

$$
\begin{aligned}
&\arcsin : [-1, 1] \to [-\frac{\pi}{2}, \frac{\pi}{2}] \\
&\arccos : [-1, 1] \to [0, \pi] \\
&\arctan : \mathbb{R} \to (-\frac{\pi}{2}, \frac{\pi}{2}) \\
&\operatorname{arccsc} : (-\infty, -1) \cup (-1, +\infty) \to (-\frac{\pi}{2}, 0) \cup (0, \frac{\pi}{2}) \\
&\operatorname{arcsec} : (-\infty, -1) \cup (-1, +\infty) \to (0, \frac{\pi}{2}) \cup (\frac{\pi}{2}, \pi) \\
&\operatorname{arccot} : \mathbb{R} \to (0, \pi)
\end{aligned}
$$

In [12]:
```
# for demonstration purposes will suppress warnings when plotting
import warnings; warnings.simplefilter('ignore')

ax = [0, 0, 0]

fig = plt.figure(constrained_layout=True, figsize=(10, 8))
gs = plt.GridSpec(2, 2, width_ratios=[1.3, 3], height_ratios=[1, 1])

ax[0] = fig.add_subplot(gs[:, 0])
ax[1] = fig.add_subplot(gs[0, 1])
ax[2] = fig.add_subplot(gs[1, 1])

x = np.linspace(-1, 1, 101)
ax[0].plot(x, np.arcsin(x), label=r'$\arcsin x$')
ax[0].plot(x, np.arccos(x), label=r'$\arccos x$')

# will include 0 to separate lines joining
x = np.concatenate((np.linspace(-5, -1, 100), [0], np.linspace(1, 5, 100)))
ax[1].plot(x, np.arcsin(1/x), label=r'arccsc $x$')
ax[1].plot(x, np.arccos(1/x), label=r'arcsec $x$')

x = np.linspace(-10, 10, 201)
ax[2].plot(x, np.arctan(x), label=r'$\arctan x$')
ax[2].plot(x, np.pi/2 - np.arctan(x), label=r'arccot $x$')

for i in range(3):
    ax[i].spines['left'].set_position('zero')
    ax[i].spines['bottom'].set_position('zero')
    ax[i].spines['right'].set_visible(False)
    ax[i].spines['top'].set_visible(False)
    ax[i].legend(loc='best')
    ax[i].set_yticks(np.pi * np.array([-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2]))
    ax[i].set_yticklabels(['$-2\pi $', r'$-\frac{3}{2} \pi $', r'$-\pi$',
                           r'$-\frac{1}{2}\pi$', '0', r'$\frac{1}{2}\pi$',
                           r'$\pi$', r'$\frac{3}{2}\pi$', r'$2\pi$'])

ax[0].set_xlim(-1.2, 1.2)
```

```
ax[0].set_ylim(-1.8, 3.3)

ax[1].set_xlim(-5, 5)
ax[1].set_ylim(-1.7, 3.3)

ax[2].set_ylim(-1.7, 3.3)

plt.show()
```



For a more complete list of properties and their visualisation, visit Wikipedia. Here we will name a few.

Complementary angles:

$$\arccos x = \frac{\pi}{2} - \arcsin x, \quad \operatorname{arccot} x = \frac{\pi}{2} - \arctan x, \quad \operatorname{arccsc} x = \frac{\pi}{2} - \operatorname{arcsec} x$$

Reciprocal arguments:

$$\begin{aligned} \arccos\left(\frac{1}{x}\right) &= \operatorname{arcsec} x, & \operatorname{arcsec}\left(\frac{1}{x}\right) &= \arccos x \\ \arcsin\left(\frac{1}{x}\right) &= \operatorname{arccsc} x, & \operatorname{arccsc}\left(\frac{1}{x}\right) &= \arcsin x \\ \text{if } x > 0: \arctan\left(\frac{1}{x}\right) &= \operatorname{arccot} x, & \operatorname{arccot}\left(\frac{1}{x}\right) &= \arctan x \\ \text{if } x < 0: \arctan\left(\frac{1}{x}\right) &= \operatorname{arccot} x - \pi, & \operatorname{arccot}\left(\frac{1}{x}\right) &= \arctan x + \pi \end{aligned}$$

Inverse hyperbolic functions

Inverse hyperbolic functions are also called **area functions** so we denote them with the prefix ar-, e.g. $\operatorname{arsinh} x$. They are also often denoted as $\sinh^{-1} x$, for example.

Hyperbolic functions

$$\sinh : \mathbb{R} \to \mathbb{R} \tag{5}$$
$$\cosh|_{[0,+\infty)} : [0, +\infty) \to [1, +\infty) \tag{6}$$
$$\tanh : \mathbb{R} \to (-1, 1) \tag{7}$$
$$\coth : \mathbb{R} \backslash \{0\} \to (-\infty, -1) \cup (1, +\infty) \tag{8}$$

are bijections, so we can define their inverses:

$$\operatorname{arsinh} := \sinh^{-1} : \mathbb{R} \to \mathbb{R} \tag{9}$$
$$\operatorname{arcosh} := \left(\cosh|_{[0,+\infty)}\right)^{-1} : [1, +\infty) \to [0, +\infty) \tag{10}$$
$$\operatorname{artanh} := \tanh^{-1} : (-1, 1) \to \mathbb{R} \tag{11}$$
$$\operatorname{arcoth} := \coth^{-1} : (-\infty, -1) \cup (1, +\infty) \to \mathbb{R} \backslash \{0\} \tag{12}$$

where

$$\begin{aligned} \operatorname{arsinh} x &= \ln(x + \sqrt{x^2 + 1}), & \operatorname{arcosh} x &= \ln(x + \sqrt{x^2 - 1}), \\ \operatorname{artanh} x &= \frac{1}{2} \ln\left(\frac{1 + x}{1 - x}\right), & \operatorname{arcoth} x &= \frac{1}{2} \ln\left(\frac{x + 1}{x - 1}\right) \end{aligned}$$

{admonition} Derivation of arcosh x

For demonstration purposes let us derive the above equation for $\operatorname{arcosh} x$. Recall that $\cosh x = \frac{e^x + e^{-x}}{2}$. Let that equal some y, so that $\frac{e^x + e^{-x}}{2} = y$ or $e^x + e^{-x} - 2y = 0$. Now let $u = e^x$ and multiply both sides by u. We get:

$$u^2 - 2uy + 1 = 0.,$$

which is a simple quadratic equation with solutions:

$$u_{1,2} = y \pm \sqrt{y^2 - 1}.$$

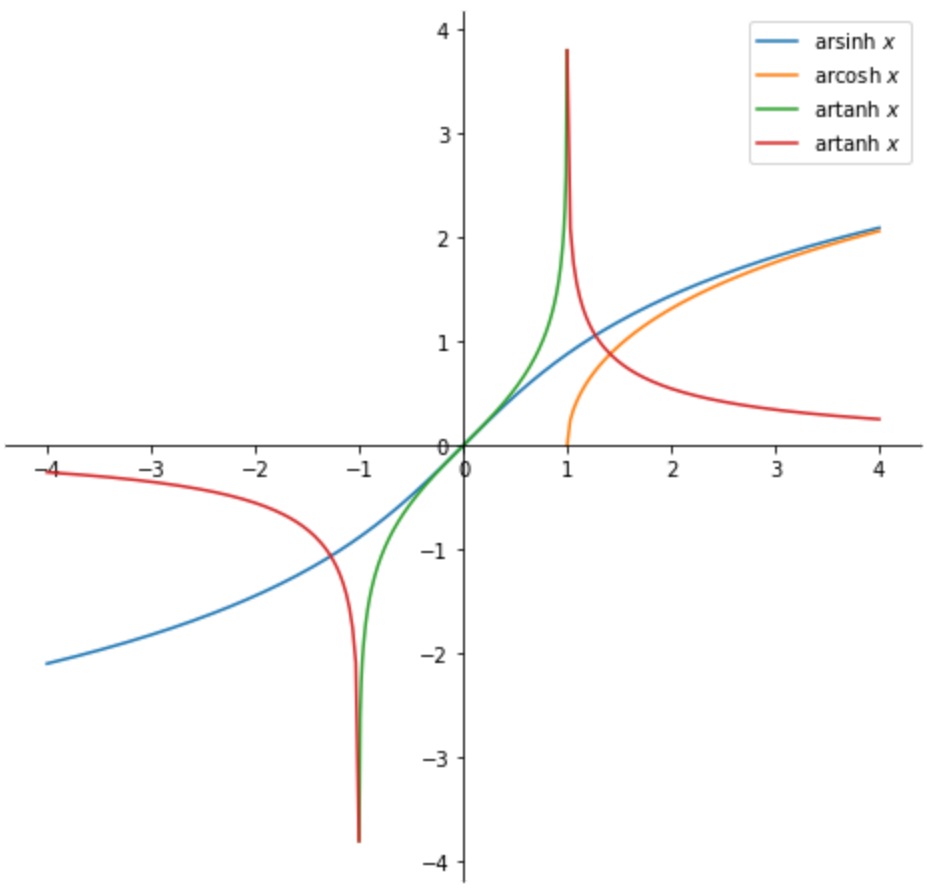
But since $u = e^x > 0$ the only possible solution is

$$u = e^x = y + \sqrt{y^2 - 1}.$$

We take the natural logarithm of both sides:

$$x = \ln\left(y + \sqrt{y^2 - 1}\right)$$

In [13]:
```
fig = plt.figure(figsize=(10, 8))
ax = plt.gca()

x = np.linspace(-4, 4, 201)
plt.plot(x, np.log(x + np.sqrt(x**2 + 1)), label=r'arsinh $x$')

x = np.linspace(1, 4, 101)
plt.plot(x, np.log(x + np.sqrt(x**2 - 1)), label=r'arcosh $x$')

x = np.linspace(-0.999, 0.999, 200)
plt.plot(x, np.log((1+x)/(1-x)) / 2, label=r'artanh $x$')

# will include 0 to avoid joining lines, this returns an error
x = np.concatenate((np.linspace(-4, -1.001, 100), [0], np.linspace(1.001, 4, 100)))
plt.plot(x, np.log((x+1)/(x-1)) / 2, label=r'artanh $x$')

ax.spines['left'].set_position('zero')
ax.spines['bottom'].set_position('zero')
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.set_aspect('equal')
ax.legend(loc='best')

plt.show()
```



5 Composition and Inverse Functions

5.1 Injection, Surjection, Bijection

Before we start talking about inverse functions, let us define some very important terms.

A function $f : A \to B$ is:

1. an **injection** (or *injective* or *one-to-one*) if different elements of the domain are assigned different elements of the codomain, i.e.:

$$\forall a, a' \in A, (a \neq a') \Rightarrow (f(a) \neq f(a')).$$

We can determine easily whether a function is an injection from its graph. For a function to be injective, any horizontal line must cross its graph in at most one point. This is why the introduced notion of strictly monotonic function is important: if a function is **strictly monotonic** (on some interval or on its entire domain), then it is an injection.

2. a **surjection** (or *surjective* or *onto*) if *every* element of its codomain B is a possible value of $f(a)$ for some $a \in A$. That is, the image of f must be equal to its codomain. That is,

$$\forall b \in B, \exists a \in A : f(a) = b.$$

3. a **bijection** (or *bijective* or one-to-one and onto or one-to-one correspondence) if it is both an injection and a surjection. That is,

$$\forall b \in B, \exists! a \in A : f(a) = b.$$

Let us summarise this in a table (source: Wikipedia):

5.2 Function composition

Let $f : A \to B$ and $g : C \to D$ be two functions. If $f[A] \subseteq C$, we can define a function $h : A \to D$ such that $h(x) = g[f(x)], \forall x \in A$. We denote the function h defined this way by $g \circ f$ and call it the **composition** of g and f.

This is generally *not* a commutative operation. For $f : \mathbb{R} \to \mathbb{R}, f(x) = x + 1$ and $g : \mathbb{R} \to \mathbb{R}, g(x) = x^2$:

$$(g \circ f)(x) = (x + 1)^2$$
$$(f \circ g)(x) = x^2 + 1$$

But it is associative. For $f : \mathbb{R} \to \mathbb{R}, f(x) = x + 1, g : \mathbb{R} \to \mathbb{R}, g(x) = \sin(x)$ and $h : \mathbb{R} \to \mathbb{R}, h(x) = \sqrt{x^2}$:

$$[h \circ (g \circ f)](x) = h[\sin(x + 1)] = \sqrt{\sin^2(x + 1)}$$
$$[(h \circ g) \circ f](x) = \sqrt{\sin^2[f(x)]} = \sqrt{\sin^2(x + 1)}$$

As an example of a function composition that is *not* defined, consider $f : \mathbb{R} \to \mathbb{R}$ where $f(x) = \cos x$ and $g : \mathbb{R} \to \mathbb{R}$ where $g(x) = \sqrt{x}$. The range of f is $[-1, 1]$ but the natural domain of g is $[0, +\infty)$.

5.2 Inverse function

Let $f : A \to B$ be a function and assume there exists another function $g : B \to A$ such that for all $x \in A$ and for all $y \in B$:

$$g[f(x)] = x \iff f[g(y)] = y.$$

We then call g the **inverse function** of f and we denote it $g = f^{-1}$.

- A function f is invertible iff it is a bijection
- The inverse function f^{-1} is unique.
- The graph of the inverse function $G(f^{-1})$ is symmetric to the graph $G(f)$ with respect to $y = x$.
- Monotonic properties of f are preserved on f^{-1}

6 Coordinate systems

In these notebooks, we will adopt the following prefix convention when naming variables:

```
's' (e.g. sDotProduct) means the variable is a scalar
'v' (e.g. vCrossProduct) means the variable is a vector
'm' (e.g. mA) means the variable is a matrix
```

This notebook will illustrate how to apply the maths discussed in the lecture using Python.

To help us apply coordinate transformations, we start by writing a number of functions for each of the different changes of coordinate that we may wish to apply. Note that we could write each of these functions much more efficiently, but as we are quite new to Python we will err on the side of clarity. First, let's consider the function for converting from 2D Cartesian coordinates to Polar coordinates.

In [14]:
```
# %pylab notebook
# this breaks colab

## This cell just imports necessary modules
from mpl_toolkits.mplot3d import Axes3D # For 3D plotting of Cylindrical and Spherical coordinates
```

Interactive plots

This notebook uses interactive plots that can't be used on the website. Please download this notebook into jupyter notebooks to gain access to the interactive plots.

6.1 Functions

`cartesian_to_polar()`

In [15]:
```
# import numpy as np      #REDUNDANT
import sympy
def cartesian_to_polar(x,y):
    '''Converts 2D Cartesian coordinates to Polar coordinates.'''
    vCoordinate = [x, y]
    # Interpret input
    sX = vCoordinate[0] # The first component of the input vector is the x-coordinate
    sY = vCoordinate[1] # The second component of the input vector is the y-coordinate

    # Coordinate transform
    sR = np.sqrt(sX**2 + sY**2)     # Pythagoras' theorem
    sTheta = np.arctan(sY/sX) # Simple trigonometry: 'TOA', Tangent = opposite/adjacent
```

```
    # Remember: we might have to modify sTheta
    # depending on which 'quadrant' we are in.
    sTheta = check_angle(sX, sY, sTheta)

    return (sR, sTheta)
    print(sR)
```

`check_angle()`

Notice that this function itself calls a function, because there will be circumstances when we need to resolve the ambiguity in the angle theta because the tangent fucntion repeats every 180 (or pi) degrees. Let's write a function `check_angle` to handle these situations.

In [34]:
```
def check_angle(sX, sY, sAngle):
    '''Adjust Polar coordinate angle based on quadrant, returning angle in
    radians between 0 and 2*pi.'''

    if(sX < 0 and sY >= 0):
        # We are in the upper left quadrant
        # so add 180 degrees (pi radians)
        # onto sAngle
        sAngle = sAngle + np.pi
    elif(sX < 0 and sY < 0):
        # We are in the lower left quadrant
        # so add 180 degrees (pi radians)
        # onto sAngle
        sAngle = sAngle + np.pi
    elif(sX >= 0 and sY < 0):
        # We are in the lower right quadrant
        # so add 360 degrees (2*pi radians)
        # onto sAngle
        sAngle = sAngle + 2*np.pi


    return sAngle
```

`polar_to_cartesian()`

Now consider the reverse function that converts from Polar coordinates to 2D Cartesian.

In [17]:
```
def polar_to_cartesian(vCoordinate):
    '''A function to convert 2D Polar coordinates to Cartesian coordinates.'''
    # Interpret input
    sR = vCoordinate[0]
    sTheta = vCoordinate[1]

    # Coordinate transform
    sX = sR*np.cos(sTheta)
    sY = sR*np.sin(sTheta)

    return (sX, sY)
```

`cartesian_to_culindrical()`

In [18]:
```
def cartesian_to_cylindrical(x,y,z):
    '''Converts 3D Cartesian coordinates to Cylindrical coordinates.'''
    vCoordinate = [x, y, z] # 3D Cartesian coordinate vector
    # Convert to cylindrical coordinates (r, phi, z)

    # Interpret input
    sX = vCoordinate[0]
    sY = vCoordinate[1]
    sZ = vCoordinate[2]

    # Coordinate transform
    sR = np.sqrt(sX**2 + sY**2)
    sPhi = np.arctan(sY/sX)

    # Again, check that we have the right value of sPhi
    # for the quadrant we are in.
    sPhi = check_angle(sX, sY, sPhi)

    return (sR, sPhi, sZ)
```

`cylindrical_to_cartesian()`

In [19]:
```
def cylindrical_to_cartesian(vCoordinate):
    '''Converts 3D Cylindrical coordinates to Cartesian coordinates.'''

    # Interpret input
    sR = vCoordinate[0]
    sPhi = vCoordinate[1]
    sZ = vCoordinate[2]

    # Coordinate transform
    sX = sR*np.cos(sPhi)
    sY = sR*np.sin(sPhi)
    sZ = sZ

    return (sX, sY, sZ)
```

`spherical_to_cartesian()`

In [20]:
```python
def spherical_to_cartesian(vCoordinate):
    '''Converts 3D Cartesian coordinates to Spherical coordinates.'''

    # Interpret inputs
    sR = vCoordinate[0]
    sPhi = vCoordinate[1]
    sTheta = vCoordinate[2]

    # Coordinate transform
    sX = sR*numpy.sin(sTheta)*cos(sPhi)
    sY = sR*numpy.sin(sTheta)*sin(sPhi)
    sZ = sR*numpy.cos(sTheta)

    return (sX, sY, sZ)
```

`cartesian_to_spherical()`

In [32]:
```python
def cartesian_to_spherical(x,y,z):
    '''Converts 3D Spherical coordinates to Cartesian coordinates.'''
    vCoordinate = [x, y, z] # 3D Cartesian coordinate vector
    # Convert to spherical coordinates (r, phi, theta)
    # Interpret inputs
    sX = vCoordinate[0]
    sY = vCoordinate[1]
    sZ = vCoordinate[2]

    # Coordinate transform
    sR = np.sqrt(sX**2 + sY**2 + sZ**2)
    sPhi = np.arctan(sY/sX)
    sTheta = np.arccos(sZ/sR)

    # Again, check that we have the right value of sPhi
    # for the quadrant we are in.
    sPhi = check_angle(sX, sY, sPhi)

    return (sR, sPhi, sTheta)
```

`plot_coordinates()`

Finally, we'll write a function to help us plot the coordinates for illustrative purposes. Don't worry about this function for now.

In [22]:
```python
def plot_coordinates(x,y,xSystem="Cart2D",projection=False):
    '''Plots 2D or 3D coordinates of a point on formatted axes with the
    option of showing the projection of the point onto the x-y-(z) axes.'''
    vCoordinate = [x, y]
    xlimits=[-5,5]
    ylimits=[-5,5]
    if (xSystem == 'Cart2D'):

        fig, ax = plt.subplots()

        # Set the spines to go through the origin
        ax.spines['left'].set_position(('data',0.))
        ax.spines['right'].set_color('none') # turn off the right spine/ticks
        ax.yaxis.tick_left()
        ax.spines['bottom'].set_position(('data',0.))
        ax.spines['top'].set_color('none') # turn off the top spine/ticks
        ax.xaxis.tick_bottom()
        ax.set_xlabel('x')
        ax.xaxis.set_label_coords(1.,0.475)
        ax.set_ylabel('y')
        ax.yaxis.set_label_coords(0.475,1.)
        ax.set_xlim(xlimits)
        ax.set_ylim(ylimits)

    elif (xSystem == 'Polar'):
        (sR, sTheta) = cartesian_to_polar(x,y)
        vCoordinate =[sTheta, sR]
        print("r = %.3f" % sR)
        print("theta = %.3f\n (radians)" % sTheta)
        fig = pylab.figure()
        ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True)


    # Plot the projections
    if (projection):
            ax.plot([0.,vCoordinate[0]],[0.,vCoordinate[1]],'b-',alpha=0.5)
            if (xSystem == 'Cart2D'):
                ax.plot([vCoordinate[0],vCoordinate[0]],[0.,vCoordinate[1]],'k--',alpha=0.5)
                ax.plot([0.,vCoordinate[0]],[vCoordinate[1],vCoordinate[1]],'k--',alpha=0.5)

    # Plot the point
    if (xSystem == 'Cylindrical' or xSystem == 'Spherical'):
        ax.plot([sX],[sY],[sZ],'ro')
    else:
        ax.plot(vCoordinate[0],vCoordinate[1],'ro')

    # Add a grid
    ax.grid(True)
```

`plot_vcoordinates()`

In [23]:
```
def plot_vcoordinates(vCoordinate, xSystem="Cart2D",
                      xlimits=[0.,1.], ylimits=[0.,1.], projection=False):
    '''Plots 2D or 3D coordinates of a point on formatted axes with the
    option of showing the projection of the point onto the x-y-(z) axes.'''

    if (xSystem == 'Cart2D'):

        fig, ax = plt.subplots()

        # Set the spines to go through the origin
        ax.spines['left'].set_position(('data',0.))
        ax.spines['right'].set_color('none') # turn off the right spine/ticks
        ax.yaxis.tick_left()
        ax.spines['bottom'].set_position(('data',0.))
        ax.spines['top'].set_color('none') # turn off the top spine/ticks
        ax.xaxis.tick_bottom()
        ax.set_xlabel('x')
        ax.xaxis.set_label_coords(1.,0.475)
        ax.set_ylabel('y')
        ax.yaxis.set_label_coords(0.475,1.)
        ax.set_xlim(xlimits)
        ax.set_ylim(ylimits)

    elif (xSystem == 'Polar'):

        fig = pylab.figure()
        ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True)

    elif (xSystem == 'Cylindrical' or xSystem == 'Spherical'):

        fig = pylab.figure()
        ax = Axes3D(fig)
        ax.set_xlabel("X")
        ax.set_ylabel("Y")
        ax.set_zlabel("Z")
        ax.plot([0.,1.2*vCoordinate[0]],[0.,0.],[0.,0.],'k-')
        ax.plot([0.,0.],[0.,1.2*vCoordinate[1]],[0.,0.],'k-')
        ax.plot([0.,0.],[0.,0.],[0.,1.2*vCoordinate[2]],'k-')
        [sX, sY, sZ] = vCoordinate

    # Plot the projections
    if (projection):
        if (xSystem == 'Cylindrical'):
            ax.plot([0.,sX],[0.,sY],[sZ,sZ],'b-',alpha=0.5)
            ax.plot([0.,sX],[0.,sY],[0.,0.],'k--',alpha=0.5)
            ax.plot([sX,sX],[sY,sY],[0.,sZ],'k:',alpha=0.5)
            ax.plot([sX],[sY],[0.],'ko',alpha=0.5)
        elif (xSystem == 'Spherical'):
            ax.plot([0.,sX],[0.,sY],[0.,sZ],'b-',alpha=0.5)
            ax.plot([0.,sX],[0.,sY],[0.,0.],'k--',alpha=0.5)
            ax.plot([sX,sX],[sY,sY],[0.,sZ],'k:',alpha=0.5)
            ax.plot([sX],[sY],[0.],'ko',alpha=0.5)
        else:
            ax.plot([0.,vCoordinate[0]],[0.,vCoordinate[1]],'b-',alpha=0.5)
            if (xSystem == 'Cart2D'):
                ax.plot([vCoordinate[0],vCoordinate[0]],[0.,vCoordinate[1]],'k--',alpha=0.5)
                ax.plot([0.,vCoordinate[0]],[vCoordinate[1],vCoordinate[1]],'k--',alpha=0.5)

    # Plot the point
    if (xSystem == 'Cylindrical' or xSystem == 'Spherical'):
        ax.plot([sX],[sY],[sZ],'ro')
    else:
        ax.plot(vCoordinate[0],vCoordinate[1],'ro')

    # Add a grid
    ax.grid(True)
```

```
plot_3Dcoordinates()
```

In [24]:
```
def plot_3Dcoordinates(x,y,z,xSystem="Cart2D",projection=False):
    '''Plots 3D coordinates of a point on formatted axes with the
    option of showing the projection of the point onto the x-y-(z) axes.'''
    vCoordinate = [x, y, z]
    xlimits=[-5,5]
    ylimits=[-5,5]
    if (xSystem == 'Cylindrical'):
        (sR, sPhi, sZ) = cartesian_to_cylindrical(x,y,z)
        print("Plotting cylindrical coordinates R, Phi(radians) and Z:", (sR, sPhi, sZ))
        fig = pylab.figure()
        ax = Axes3D(fig)
        ax.set_xlabel("R")
        ax.set_ylabel("Phi")
        ax.set_zlabel("Z")
        ax.plot([0.,1.2*vCoordinate[0]],[0.,0.],[0.,0.],'k-')
        ax.plot([0.,0.],[0.,1.2*vCoordinate[1]],[0.,0.],'k-')
        ax.plot([0.,0.],[0.,0.],[0.,1.2*vCoordinate[2]],'k-')
        [sX, sY, sZ] = vCoordinate
    elif (xSystem == 'Spherical'):
        (sR, sPhi, sTheta) = cartesian_to_spherical(x,y,z)
        print("Plotting spherical coordinates R, Phi(radians) and Theta(radians):", (sR, sPhi, sTheta))
        fig = pylab.figure()
        ax = Axes3D(fig)
        ax.set_xlabel("R")
        ax.set_ylabel("Phi")
```

```
    ax.set_zlabel("Theta")
    ax.plot([0.,1.2*vCoordinate[0]],[0.,0.],[0.,0.],'k-')
    ax.plot([0.,0.],[0.,1.2*vCoordinate[1]],[0.,0.],'k-')
    ax.plot([0.,0.],[0.,0.],[0.,1.2*vCoordinate[2]],'k-')
    [sX, sY, sZ] = vCoordinate
if (projection):
    if (xSystem == 'Cylindrical'):
        ax.plot([0.,sX],[0.,sY],[sZ,sZ],'b-',alpha=0.5)
        ax.plot([0.,sX],[0.,sY],[0.,0.],'k--',alpha=0.5)
        ax.plot([sX,sX],[sY,sY],[0.,sZ],'k:',alpha=0.5)
        ax.plot([sX],[sY],[0.],'ko',alpha=0.5)
    elif (xSystem == 'Spherical'):
        ax.plot([0.,sX],[0.,sY],[0.,sZ],'b-',alpha=0.5)
        ax.plot([0.,sX],[0.,sY],[0.,0.],'k--',alpha=0.5)
        ax.plot([sX,sX],[sY,sY],[0.,sZ],'k:',alpha=0.5)
        ax.plot([sX],[sY],[0.],'ko',alpha=0.5)
# Plot the point
if (xSystem == 'Cylindrical' or xSystem == 'Spherical'):
    ax.plot([sX],[sY],[sZ],'ro')
else:
    ax.plot(vCoordinate[0],vCoordinate[1],'ro')

# Add a grid
ax.grid(True)
```

6.2 Examples

Right, so let's now utilise our functions to convert between different coordinate systems. Let's start by defining a point in 2D Cartesian coordinates and plotting the point.

In [28]:
```
#VSCODE (calculations:1) (plots:1) (interactive:0.9)
        #DON'T RUN ALL - BECAUSE SOMETHING DOWNSTREAM WILL BREAK THIS
#COLAB (calculations:1) (plots:1) (interactive:0.5)
        # dropdown doesn't work

# A Cartesian coordinate vector in the form (x, y).
from ipywidgets import interactive
import pylab

#%matplotlib widget

# Let's plot coordinates on the X-Y plane using the sliders.
interactive_plot = interactive(plot_coordinates, xSystem=systems, x=(-5,5), y=(-5,5))
interactive_plot
```

Out[28]:
```
interactive(children=(IntSlider(value=0, description='x', max=5, min=-5), IntSlider(value=0, description='y', …
```

Cartesian to polar conversion

We can convert these coordinates to Polar coordinates using our function and plot on a Polar diagram.

In [29]:
```
# Cartesian to polar sliders
# Remember: theta will be in RADIANS
# Change the xSystems tab to 'Polar' to see the 2D cartesian coordinates converted
# to Polar coordinates on a Polar diagram

#WORKS CORRECTLY IN VSCODE

systems = ['Cart2D', 'Polar']
interactive_plot = interactive(plot_coordinates, xSystem=systems, x=(-5,5), y=(-5,5))
interactive_plot
```

Out[29]:
```
interactive(children=(IntSlider(value=0, description='x', max=5, min=-5), IntSlider(value=0, description='y', …
```

Polar to cartesian conversion

In [30]:
```
# Let's define some polar coordinates. . .
sR = 6400.0
# Convert angles in degrees to radians
sTheta = 112.9*(np.pi/180)

print("Converting r = %.2f, theta = %.2f back to Cartesian coordinates" % (sR, sTheta))
vCoordinate = [sR, sTheta]
(sX, sY) = polar_to_cartesian(vCoordinate)
print("x = %.2f" % sX)
print("y = %.2f" % sY)
```

```
Converting r = 6400.00, theta = 1.97 back to Cartesian coordinates
x = -2490.39
y = 5895.59
```

Cylindrical and spherical coordinates

In [41]:
```
#Change the xSystems tab to 'Cylindrical' and 'Spherical' to see the
# 3D cartesian coordinates converted to Cylindrical and Spherical coordinates respectively


#%matplotlib inline
```

```
# import pylab
systems = ['Cylindrical','Spherical']

interactive_plot = interactive(plot_3Dcoordinates, xSystem=systems, x=(-5,5,0.1), y=(-5,5,0.1), z=(-5,5,0.1))
interactive_plot
```

Out[41]:
```
interactive(children=(FloatSlider(value=0.0, description='x', max=5.0, min=-5.0), FloatSlider(value=0.0, descr…
```

Spherical to cartesian

In [40]:
```
#VSCODE no plot


# Let's define some spherical coordinates. . .
sR = 6400.0
# Convert angles in degrees to radians
sPhi = 316.8*(np.pi/180)
sTheta = 112.9*(np.pi/180)

print("Plotting spherical coordinates ", (sR, sPhi, sTheta), " in 3D Cartesian space")

# Convert back to Cartesian for 3D plotting purposes
vCoordinate = [sR, sPhi, sTheta]

def spherical_to_cartesian(vCoordinate):
    sX = vCoordinate[0] * np.sin(vCoordinate[1]) * np.cos(vCoordinate[2])
    sY = vCoordinate[0] * np.sin(vCoordinate[1]) * np.sin(vCoordinate[2])
    sZ = vCoordinate[0] * np.cos(vCoordinate[1])
    return (sX, sY, sZ)

(sX, sY, sZ) = spherical_to_cartesian(vCoordinate)

# Plot the 3D Cartesian coordinates
plot_vcoordinates([sX,sY,sZ], xSystem = 'Spherical', projection=True)
fig = pylab.figure()
axes = Axes3D(fig)
axes.plot([0.],[0.],[0.],'oy')
axes.plot([sX],[sY],[sZ],'or')
axes.plot([0.,sX],[0.,sY],[0.,sZ],'b-')
axes.plot([0.,sX],[0.,sY],[0.,0.],'k--')
axes.plot([sX,sX],[sY,sY],[0.,sZ],'k:')
axes.plot([sX],[sY],[0.],'ok')
axes.set_xlabel("X")
axes.set_ylabel("Y")
axes.set_zlabel("Z")
```

```
Plotting spherical coordinates  (6400.0, 5.529203070318037, 1.9704767255015982)  in 3D Cartesian space
```

Out[40]:
```
Text(0.5, 0, 'Z')
```

```
<Figure size 432x288 with 0 Axes>
<Figure size 432x288 with 0 Axes>
```

Final tip

So what did I mean when I said we could write these functions much more efficiently? Well, let's try writing the first function again in as few lines as possible:

In [ ]:
```
def cartesian_to_polar(vC):
    '''Converts 2D Cartesian coordinates to Polar coordinates.'''

    return sqrt(vC[0]**2 + vC[1]**2), check_angle(vC[0], vC[1], numpy.arctan(vC[1]/vC[0]))
    # i can't get this to work but copilot suggests the following code:
```

It turns out we can write this function in just one line of code, because there was no need to rename the input vector components as scalars and we can write the transformation formulae (and the `check_angle` function in the return statement.

To improve this notebook you could add interactive plots for the conversions from polar, cylindrical and spherical coordinates back to cartesian coordinates.

7 Vectors

In these notebooks, we will adopt the following prefix convention when naming variables:

```
    's' (e.g. sDotProduct) means the variable is a scalar
    'v' (e.g. vCrossProduct) means the variable is a vector
    'm' (e.g. mA) means the variable is a matrix
```

In [ ]:
```
# Install if necessary
# %pip install plotly
```

In [ ]:
```
## This cell just imports necessary modules
# import numpy as np     #redundant
import plotly.graph_objs as go

# This notebook looks awesome in VSCode and Colab, but the pdf lacks color
# The plots in this notebook are interactive - 3d style - but not in pdf
```

In [ ]:
```
# This cell shows a function that can plot vectors using the Scatter3d() in plotly.
#In the plots the vectors have a big point to mark the direction.

def vector_plot(tvects,is_vect=True,orig=[0,0,0]):
    """Plot vectors using plotly"""

    if is_vect:
        if not hasattr(orig[0],"__iter__"):
            coords = [[orig,np.sum([orig,v],axis=0)] for v in tvects]
        else:
            coords = [[o,np.sum([o,v],axis=0)] for o,v in zip(orig,tvects)]
    else:
        coords = tvects

    data = []
    for i,c in enumerate(coords):
        X1, Y1, Z1 = zip(c[0])
        X2, Y2, Z2 = zip(c[1])
        vector = go.Scatter3d(x = [X1[0],X2[0]],
                              y = [Y1[0],Y2[0]],
                              z = [Z1[0],Z2[0]],
                              marker = dict(size = [0,5],
                                            color = ['blue'],
                                            line=dict(width=5,
                                            color='DarkSlateGrey')),
                              name = 'Vector'+str(i+1))
        data.append(vector)

    layout = go.Layout(
             margin = dict(l = 4,
                           r = 4,
                           b = 4,
                           t = 4)
                  )
    fig = go.Figure(data=data,layout=layout)
    fig.show()
```

In [ ]:
```
# works interactively in Colab
# works interatively in VSCode
# Let's define two vectors, by listing their components
vA = [1, 2, 1]
vB = [-1, 1, 0]

# Convert vA and vB from a list to an array so we can use numpy
# to perform vector operations on them.
vA = np.array(vA)
vB = np.array(vB)

print("Plot of vA (blue) and vB (red)")
vector_plot([vA,vB])
```

Null vector

In [ ]:
```
# vNull is the null vector (a vector of all zeros,
# created using the zeros function)
vNull = np.zeros(3)
print("The null vector is: ", vNull)
```

Vector equality

In [ ]:
```
# Here we compare vectors vA and vB. This is done element-by-element.
# The .all() function allows us to check that the elements of vA.
# are ALL equal to those of vB. In other words, vA[i] == vB[i] for i = 1, 2, 3
if(np.equal(vA,vB).all()):
   print("Vectors vA and vB are equal")
else:
   print("Vectors vA and vB are NOT equal")
```

Multiplication of a vector by a scalar

In [ ]:
```
print("2*vA = ", 2*vA)
print("10*vB = ", 10*vB)

print("Plot of 2*vA (blue) and 10*vB (red)")
vector_plot([2*vA,10*vB])
```

Vector addition and subtraction

In [ ]:
```
print("vA + vB = ", vA + vB)
print("vA - vB = ", vA - vB)

print("Plot of vA + vB (blue) and vA - vB (red)")
vector_plot([vA + vB,vA - vB])
```

Vector magnitude

Compute the magnitude (i.e. Euclidean/L2 norm) of vectors `vA` and `vB` (to 3 significant figures). There are many ways to do this in Python. Try using the `sqrt` and `dot` functions (see below) to achieve the same result.

In [ ]:
```
sMagnitude_vA = np.linalg.norm(vA, ord=2)
sMagnitude_vB = np.linalg.norm(vB, ord=2)
print("Magnitude of vector vA is %.3f" % sMagnitude_vA)
print("Magnitude of vector vB is %.3f" % sMagnitude_vB)
```

Dot product

Remember: the dot product of two vectors always returns a scalar

In [ ]:
```
sDotProduct_vAvB = np.dot(vA,vB)
print("The dot product of vA and vB is %.3f" % sDotProduct_vAvB)
```

Angle between two vectors

In [ ]:
```
# NOTE: arccos is inverse cos
sTheta = np.arccos(sDotProduct_vAvB/(sMagnitude_vA*sMagnitude_vB))
print("The angle between vA and vB is %.3f radians (or %.3f degrees)"
      % (sTheta, sTheta*(180/np.pi)))
```

Cross product

Remember: the cross product of two vectors always returns another vector. We could use `np.cross` :

In [ ]:
```
print("The cross product of vA and vB is", np.cross(vA,vB))
```

Alternatively, we could compute determinants as in slide 22:

In [ ]:
```
# Remember: in Python, array indices always start from zero,
# so the x, y and z components have an index of 0, 1 and 2 respectively.
vCrossProduct = [ np.linalg.det([[vA[1], vA[2]],
                                 [vB[1], vB[2]]]),

                  np.linalg.det([[vA[0], vA[2]],
                                 [vB[0], vB[2]]]),

                  np.linalg.det([[vA[0], vA[1]],
                                 [vB[0], vB[1]]]) ]

print("The cross product of vA and vB is", vCrossProduct)

print("Plot of vA (blue),vB (red) and the cross product of vA and vB (green)")
vector_plot([vA,vB,np.cross(vA,vB)])
```

3.1 %matplotlib inline import matplotlib.pyplot as plt %matplotlib inline

3.4 import numpy as np

6.0 from mpl_toolkits.mplot3d import Axes3D

6.1 import sympy

6.2 from ipywidgets import interactive import pylab #disabling

7.0 import plotly.graph_objs as go