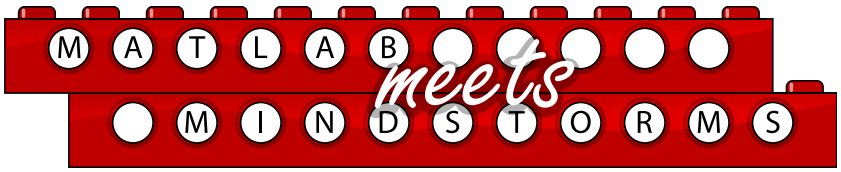
Projekt der Elektrotechnik und Informationstechnik



4. Projektversuch

- Motoren -

25. Februar 2020

Inhaltsverzeichnis

1 Einleitung

In diesem Versuch wird die Ansteuerung der LEGO Mindstorms EV3 Motoren unter MATLAB mit Hilfe der RWTH - Mindstorms EV3 Toolbox vorgestellt und die Motoreigenschaften anhand von verschiedenen Versuchsaufgaben und Anwendungsbeispielen analysiert. Außerdem werden unterschiedliche Übersetzungsverhältnisse von Getrieben berechnet und für Anwendungen ausgenutzt. Neben Motormessungen ist eine Maschine mit Ziffernblatt zur Zahlendarstellung zu bauen und zu programmieren. Zuletzt ist diese dann zur Darstellung von Phasen komplexer Zahlen zu verwenden.

2 Grundlagen

2.1 LEGO Mindstorms EV3 Motor

Die LEGO Mindstorms EV3 Motoren sind Servomotoren. Ein Servomotor ist ein Motor, der eine vorgegebene Position anfahren und halten kann. Da die EV3 Motoren rotatorische Motoren (siehe Abb. 1) sind, können als Zielgröße ausschließlich Winkelpositionen vorgegeben werden. Darüber hinaus verfügen die Motoren über ein Übersetzungsgetriebe, dessen Drehgeschwindigkeit und damit indirekt das Drehmoment variiert werden kann.

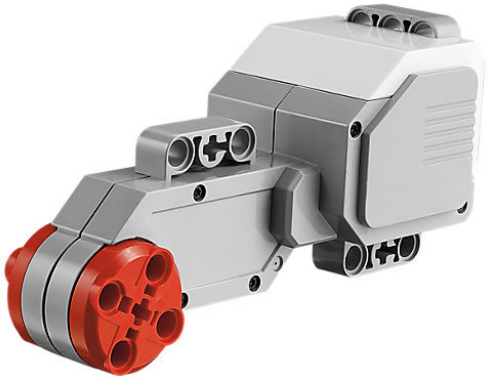


Abbildung 1: EV3 Servomotor (LEGO ©)

Für eine präzise Drehwinkelansteuerung verfügen die Motoren über einen Rotationssensor. Dieser ist durch eine optische Lichtschranke in einem sich drehenden Speichenrad realisiert. Der Rückgabewert des Rotationssensors ist der durchfahrende Winkel in Winkelgrad. Dieser kann ausgelesen und somit die aktuelle Winkelposition des Motors bestimmt werden. Eine ganze Motorumdrehung entspricht dabei 360° bei einer Messgenauigkeit von ±1 Grad. Des Weiteren besteht die Möglichkeit zwei Motoren miteinander zu synchronisieren. In diesem Falle werden die Motoren aktiv geregelt, so dass sich beide Motoren in einem definierten Drehwinkelverhältnis drehen. Diese Eigenschaft kann oft anwendungsspezifisch, z.B. für Fahrroboter die geradlinig fahren sollen, ausgenutzt werden.

2.2 Die Klasse Motor

In diesem Abschnitt wird die Klasse `Motor` der RWTH - Mindstorms EV3 Toolbox vorgestellt. Zusätzlich können weitere Erläuterungen und Hilfen zu den einzelnen Eigenschaften und Methoden der Klasse auch direkt aus der MATLAB Hilfe entnommen werden. Die Klasse bietet folgende Eigenschaften und Methoden, die im Folgenden genauer erläutert werden:

- Klassenname: `Motor`
  - Eigenschaften mit Lese- und Schreibzugriff

    - power
    - speedRegulation
    - limitValue
    - limitMode
    - brakeMode
    - smoothStart
    - smoothStop
    - debug
  - Eigenschaften mit ausschließlichem Lesezugriff
    - isRunning
    - tachCount
    - currentSpeed
    - type
  - Methoden
    - start
    - stop
    - syncedStart
    - syncedStop
    - waitFor
    - setBrake
    - resetTachoCount
    - internalReset

Motorobjekt

Die EV3 Motoren werden in MATLAB mit Objekten der `Motor` Klasse beschrieben. Bei der Erstellung eines `EV3`-Objekts werden automatisch Objekte für die vier verfügbaren Motorports A bis D erstellt. Auf die Motorobjekte kann über das erstellte `EV3`-Objekt zugegriffen werden. Durch Eingabe des Objektnamens in der MATLAB-Kommandozeile können die Eingenschaften des Motorobjekts angezeigt werden. Direkt nach Erzeugen des Objekts sehen diese wie folgt aus:

```
>> brick = EV3; % erzeugt das EV3-Objekt, die Motoren werden automatisch mit erzeugt
>> brick.motorA
   Motor with properties:

    Writable
```

```
              power:   0
    speedRegulation:   0
        smoothStart:   0
         smoothStop:   0
         limitValue:   0
          limitMode:   Tacho
          brakeMode:   Coast
              debug:   0

    Read-only
          isRunning:   0
         tachoCount:   0
       currentSpeed:   0.0
               type:   LargeMotor
>>
```

Eigenschaften

Die Eigenschaften beschreiben die individuelle Instanz einer Klasse (also ein Objekt). Sie enthalten Informationen über den aktuellen Zustand des Objekts. Bei der Erzeugung eines Motorobjekts werden diese mit Standardwerten vorinitialisiert. Es gibt durch den Benutzer veränderbare Eigenschaften (`Writable`) und solche, die nur Lesezugriff erlauben (`Read-only`). Über erstere kann das Verhalten des Motors verändert werden, wogegen die nur lesbaren Eigenschaften von internen Zuständen des Motorobjekts abhängen und Aussagen über den aktuellen Status des Objekts bzw. des damit assoziierten Motors erlauben. Lesezugriff auf Eigenschaften ist direkt über den Namen der Eigenschaft möglich. Beispiel:

```
>> brick.motorA.isRunning

ans =

     0

>> brick.motorA.type

ans =

    LargeMotor

>>
>>
```

Um schreibbare Eigenschaften zu verändern, gibt es zwei Möglichkeiten. Sie können einerseits durch direkten Aufruf und Benutzung des Punkt-Operators (.) mit einem Wert versehen werden:

```
>> brick.motorA.limitValue = 1500;
>> brick.motorA.brakeMode = 'Brake';
```

```
>> brick.motorA
   Motor with properties:

    Writable
              power:   0
    speedRegulation:   0
        smoothStart:   0
         smoothStop:   0
         limitValue:   1500
          limitMode:   Tacho
          brakeMode:   Brake
              debug:   0

    Read-only
           isRunning:   0
         tachoCount:   0
       currentSpeed:   0.0
               type:   LargeMotor

>>
```

Alternativ kann der `setProperties` - Befehl benutzt werden, der das Setzen mehrerer Eigenschaften auf einnmal erlaubt. Es ist zu beachten, dass hierbei die Bezeichner der Eigenschaften in Anführungszeichen gesetzt werden müssen:

```
>> brick.motorA.setProperties('limitValue', 1500, 'brakeMode', 'Brake');
>> brick.motorA
   Motor with properties:

    Writable
              power:   0
    speedRegulation:   0
        smoothStart:   0
         smoothStop:   0
         limitValue:   1500
          limitMode:   Tacho
          brakeMode:   Brake
              debug:   0

    Read-only
           isRunning:   0
         tachoCount:   0
       currentSpeed:   0.0
               type:   LargeMotor

>>
```

Achtung: Sollte ein übergebenes Stichwort keinem Attribut entsprechen, so wird ***keine*** Fehlermeldung ausgegeben.

Zulässig für die Eigenschaften sind folgende Parameterwerte:

```
power:              [-100...100]
speedRegulation:    0,1 oder false, true
smoothStart:        false, true
smoothStop:                                        false, true
limitValue:         [0...99999] (integer, 0 = läuft unendlich)
limitMode:          'Tacho', 'Time'
brakeMode:          'Coast', 'Brake'
%
```

Als Besonderheit kann ein `motor`–Objekt auch zwei Motoren gleichzeitig ansprechen. Dies ist besonders hilfreich, wenn zwei Motoren synchron gesteuert werden sollen. In diesem Modus können z.B. Fahrroboter auf einfach Weise geradlinig fahren. Hierzu werden die Funktionen syncedStart und syncedStop eingesetzt. Als Parameter erhält syncedStart den Motor, mit dem synchronisiert werden soll. Beispiel:

```
>> m = brick.motorA;
>> m.power = 50;
>> m.syncedStart(brick.motorB);
>> pause(3);
>> m.syncedStop();
```

Näheres zum synchronisierten Fahren erhalten Sie in der Hilfe zu `Motor`.

power

`Power` legt die gewünschte Drehgeschwindigkeit des Motors fest. Es kann ein ganzzahliger Parameterwert angegeben werden, der als ein prozentualer Wert der maximal möglichen Drehgeschwindigkeit zu verstehen ist. Der Wert 25 entspricht daher 25% der maximalen Drehgeschwindigkeit. Außerdem muss zusätzlich die Drehrichtung des Motors durch ein Vorzeichen angegeben werden. Damit ergibt sich der gesamte Wertebereich zu $[-100 \ldots 0 \ldots 100]$. Der Wert -32 entspricht daher 32% der maximalen Geschwindigkeit in entgegengesetzter Drehrichtung. Das Vorzeichen ist analog zu der Umpolung (von $+$ zu $-$) der an dem Motor anliegenden elektrischen Spannung zu verstehen. Der Gleichstrommotor ändert seine Drehrichtung, wenn die anliegende Spannung ihr Vorzeichen wechselt.
Zu `power` ist zu beachten, dass durch Verluste im Motor die praktisch erreichbare Geschwindigkeit bei etwa 85-90% der theoretisch möglichen liegt. Das bedeutet, dass der Motor auch bei `power = 100` und ohne zusätzliche Last maximal eine `currentSpeed` von 90 erreichen kann. Durch Lasten wie Räder, Zeiger oder angeschlossene Getriebe wird diese Zahl noch weiter gesenkt.

speedRegulation

Um den EV3 Motor auf einer möglichst konstanten Drehgeschwindigkeit zu halten, steht die Eigenschaft `speedRegulation` zur Verfügung. Mit den Funktionsargumenten `true, 1`

und `false, 0` kann der Modus an- bzw. ausgeschaltet werden. Die Geschwindigkeitsregulierung bewirkt, dass der Motor die durch `power` vorgegebene Drehgeschwindigkeit konstant hält. Diese Eigenschaft kann z.B. bei Roboterarmen, die unterschiedlich schwere Lasten heben müssen, verwendet werden. Das Drehmoment des Motors muss daher lastabhängig mal stärker und mal schwächer sein, um eine konstante Drehgeschwindigkeit zu halten. Dies wird durch die Geschwindigkeitsregulierung gesteuert. Für kleine Drehgeschwindigkeiten (z.B. `power(x)` mit $x < 10$) ermöglicht die `speedRegulation`, dass der Motor aus seiner Ruhelage noch anfahren kann. Denn die Geschwindigkeitsregulierung bewirkt ein geregeltes Erhöhen und Verringern der Motorkraft, um die angegebene Drehgeschwindigkeit zu erreichen. Diese Regelung kann allerdings bei kleinen `power`-Werten (je nach Belastung) ein Schwanken der Geschwindigkeit oder ruckartige Bewegungen hervorrufen. In solchen Fällen muss mit anderen Werten für `power` sowie der Einstellung für `speedRegulation` experimentiert werden.

Allgemein gilt: `speedRegulation = false` bewirkt ein konstantes Drehmoment des Motors, während `speedRegulation = true` durch Regelung des Drehmoments für eine möglichst konstante Geschwindigkeit sorgt. Welches Verhalten gewünscht ist, hängt von der Roboterkonstruktion ab.

Hinweis: Die Geschwindigkeitsregulierung kann nicht gleichzeitig mit dem Synchronisationsmodus s.o. aktiviert sein (also wenn zwei Motoren gleichzeitig über ein einzelnes Objekt gesteuert werden).

limitMode und limitValue

hier kann angegeben werden, ob sich der Motor nach einer bestimmten Zeit oder einem überstrichenen Winkel abschalten soll. Hierzu wird zunächst mit `limitMode` angegeben, ob der Motor sich nach einer konstanten Zeit (`limitMode = 'Time'`), angegeben in Milisekunden, oder einem bestimmten gedrehten Winkel (`limitMode = 'Tacho'`) abschalten soll. Eine volle Motorumdrehung entspricht dem Wert 360. Bei einem vorgegebenen ganzzahligen Wert > 0 versucht der Motor den gewünschten Drehwinkel anzufahren. Wird dagegen der Wert von `limitValue` auf 0 gesetzt, so dreht der Motor ohne Begrenzung immer weiter, bis er durch einen separaten Stoppbefehl (`stop` s.u.) angehalten wird. Negative und Fließkommawerte sind nicht zugelassen. Änderungen der Drehrichtung erfolgen über einen Vorzeichenwechsel von `power`.

brakeMode

`brakeMode` gibt an, wie sich der Motor verhalten soll, nachdem der durch `limitValue` festgelegte Zielwinkel bzw. die Zielzeit erreicht wurde. Es stehen die zwei Modi `'Coast'`und `'Brake'` zur Verfügung. Wird keiner der Parameter explizit vorgegeben, wird der Defaultwert `'Coast'` verwendet. Der Modus `'Coast'` bewirkt, dass der Motor nach Erreichen des vorgegebenen `tachoLimit` einfach ausgeschaltet wird. Abhängig von der Drehgeschwindigkeit und dem Trägheitsmoment des Motors rollt dieser langsam aus und dreht dabei

über das `limitValue` hinaus. Dies kann für Anwendungen ausgenutzt werden, bei denen ein sanftes Ausrollen des Motors erwünscht ist. `'Brake'` bewirkt, dass der rotierende Motor präzise (Genauigkeit von ca. $\pm$ 1°) bei dem vorgebenen `limitValue` zum Stehen kommt und auch dort gehalten wird. Dieses Verhalten wird durch eine aktive Motorregelung bewirkt, die versucht, ein Überschwingverhalten zu minimieren. Nachdem der Motor das `limitValue` erreicht hat und die Motorregelung beendet ist, wird die aktive Bremse weiter gehalten, um z.B. eine Drehung des Motors durch eine angehängte Last zu verhindern. Möchte man die Bremse wieder lösen, so muss der Befehl `setBrake` benutzt werden.

smoothStart und smoothStop

Soll der Motor nicht ruckartig, sondern sanft anfahren bzw. bremsen, so kann dies durch die Eigenschaften `smoothStart` und `smoothStop` bewirkt werden. Dann wird die Geschwindigkeit des Motors nicht sofort auf den über `power` definierten Wert gesetzt, sondern kontinuierlich erhöht. Es muss beachtet werden, dass die Summe aus `smoothStart`und `smoothStop` nicht größer als der Wert `limitValue` sein darf. Sollte dies doch der Fall sein, so bricht das Programm beim Übermitteln von `start` mit einer Fehlermeldung ab. Wird z.B. der Motor mit `limitValue = 360` und `smoothStart = 90` sowie `smoothStop = 90` aufgerufen, so werden die ersten 90 Grad zum langsamen Anfahren benutzt, dann dreht der Motor 180 Grad mit der angegebenen Geschwindigkeit, um dann in den letzten 90 Grad immer langsamer zu werden.

Methoden

Neben den genannten Eigenschaften der Klassen besitzt die `EV3Motor` Klasse auch Methoden, über die dem Motor Befehle gegeben werden können. Folgende Methoden stehen einer Instanz (Objekt) der EV3Motor Klasse zur Verfügung:

- `start`
- `stop`
- `syncedStart`
- `syncedStop`
- `waitFor`
- `resetTachoCount`
- `internalReset`

Zum Aufruf dieser Methoden ist auch hier der Punktoperator zu verwenden.

start

Beginnt, den Motor mit dem über `power` definierten Wert zu drehen. Der Motor dreht, bis der von `limitValue` erreicht Wert (Zeit oder überstrichener Winkel) erreicht worden ist, und rollt dann aus (wenn `brakeMode = 'Coast'` gesetzt ist) bzw. bremst aktiv ab (wenn `brakeMode = 'Brake'` gesetzt ist). Je nach gesetztem `smoothStart`- Wert wird sanft oder ruckartig angefahren bzw. gebremst.

Beispiel:

```
>> brick.motorA.power = 50;
>> brick.motorA.limitValue = 200;
>> brick.motorA.start;
>>
```

Der EV3-Motor an Port A dreht mit der Powereinstellung 50 um den Drehwinkel von 200 Grad. Da kein anderer Parameter geändert worden ist, wird ruckartig angefahren und bei Erreichen der Zielposition ausgerollt.

Hinweis: Wenn ein neuer Motorbefehl an den Brick gesendet werden soll, muss der entsprechende Motor bereit für einen Befehl sein (darf also nicht noch mit der Abarbeitung des letzten Befehls beschäftigt sein), ansonsten wird der Befehl ignoriert und eine Warnung wird auf dem Bildschirm ausgegeben. Dazu stehen zum Warten auf den Motor die Methode `waitFor` (s.u.) und zum Abbrechen der laufenden Bewegung die Methode `stop` (s.u.) zur Verfügung.

stop

Um einen Motor unmittelbar zu stoppen, d.h. auch während er noch einen Drehwinkel (`limitValue`) anzufahren versucht, dient die Methode `stop`. Sobald der Befehl vom EV3 verarbeitet ist, wird die Stromzufuhr zum Motor unterbrochen und der Motor kommt zum Stillstand. Je nach gesetztem `brakeMode` wird dann aktiv gebremst oder der Motor ausrollen gelassen.

Beispiel:

```
>> brick.motorA.setProperties('power', 45, 'limitValue', 0, 'brakeMode', 'Coast');
>> brick.motorA.start;
>> pause(4); % warte 4 Sekunden
>> brick.motorA.stop;
>>
```

waitFor

Eine wichtige Eigenschaft der bisher erläuterten Funktionen ist, dass die Befehle direkt an den EV3 gesendet und sofort ausgeführt werden. Dies kann zu zeitlichen Ablaufproblemen führen. Zur Erläuterung sei folgendes Beispiel gegeben:

```
>> brick.motorA.setProperties('power', 60, 'limitValue', 1000);
>> brick.motorA.start;
>> brick.motorA.stop;
>>
```

Zuerst wird Motor `'A'` mit der relativen Geschwindigkeit 60 und dem zu drehenden Rotationswinkel von 1000° gesetzt und die bewegung über `start` ausgeführt. Unmittelbar danach wird der Stoppbefehl `stop` gesendet. Sobald dieser beim EV3 angekommen ist, wird auch dieser ausgeführt und `'A'` stoppt. In diesem Fall wurde der eigentliche Befehl, dass der Motor auf 1000° drehen soll, unterbrochen. Da MATLAB mit der Befehlsabfolge nicht wartet, bis der Motor seinen Befehl beendet hat, wurde die Methode `waitFor` entwickelt. Diese unterbricht den weiteren MATLAB Befehlsablauf solange, bis der Motor seinen Drehwinkel erreicht hat.

Beispiel:

```
>> brick.motorA.setProperties('power', 60, 'limitValue', 1000);
>> brick.motorA.start;
>> brick.motorA.waitFor;
>> brick.beep;
```

setBrake

Mit dieser Methode kann die Bremse des Motors manuell getätigt und gelöst werden. So kann z.B. die Bremse, die nach einem gebremsten Stopp des Motors weiterhin gesetzt ist, wieder gelöst werden, um den Motor wieder manuell drehen zu können.

resetTachoCount

Der über die Eigenschaft `tacho` auslesbare Drehwinkelzähler kann mit der Funktion `resetTachoCount` auf null zurückgesetzt werden.
Hinweis: Der auslesbare Tacho wird mit resetTachoCount immer zurückgesetzt. Es gibt einen zweiten, versteckten Tachostand, der auch eine manuelle Bewegung des Motors registriert und speichert. Ändert man die Motor-Position per Hand, lässt sich dieser versteckte Zähler nur mit dem Befehl `internalReset` zurücksetzen.

2.3 MATLAB - Darstellung komplexer Zahlen

Neben reellen Zahlen, kann MATLAB auch mit komplexen Zahlen umgehen, die aus Real- und Imaginärteil bestehen. Der Datentyp der Werte kann ein beliebiger numerischer Datentyp sein, muss aber für beide Komponenten gleich sein. Sie können komplexe Zahlen entweder direkt erzeugen, indem Sie den Imaginärteil mit `i` oder `j` kennzeichnen.

```
>> c = 1 + 2i

c =    1.0000 + 2.0000i
>> c = 1 + 2j

c =    1.0000 + 2.0000i
>> c = 1 + 2*i

c =    1.0000 + 2.0000i
```

Die erste Schreibweise funktioniert nur für die direkte Eingabe von Zahlen, nicht aber für Variablen. Eine einfache Möglichkeit Real- und Imaginärteil mit Variablen zu setzen ist durch die Funktion `complex` gegeben. Dessen erstes Funktionsargument ist der Real- und das zweite der Imaginärteil. Dies funktioniert auf für Matrizen beliebiger Dimension:

```
>> a = 1;
>> b = 2;
>> c = complex(a,b)

c =    1.0000 + 2.0000i
```

Matrizen - also auch Vektoren und Skalare - mit komplexen Werten werden in MATLAB bezüglich der Standard-Operatoren behandelt wie reelle Matrizen. Die einzelnen Werte einer komplexen Matrix werden in Komponentenschreibweise angegeben, d.h. sie bestehen aus der Summe von Real- und Imaginärteil, wobei der Imaginärteil durch ein kleines `i` gekennzeichnet ist. Folgendes Beispiel soll dies veranschaulichen:

```
>> x = 1 + i

x =    1.0000 + 1.0000i

>> y = [1 , 2 + 2j ; -2 - 0.5*j , 3j]

y =     1.0000             2.0000 + 2.0000i
       -2.0000 - 0.5000i        0 + 3.0000i
```

Offensichtlich dürfen die Zeichen `i` und `j` benutzt werden, um damit die komplexe Zahl i zu erzeugen. Diese können wie im obigen Beispiel gezeigt direkt mit dem Imaginärteil verbunden werden `(3i, 3j)` oder durch explizite Multiplikation `(3*i, 3*j)`.

Vorsicht: Die Zeichen i und j sollten möglichst nicht als Variablen verwendet werden, da sonst eine verwirrende Doppeldeutigkeit entstehen kann:

```
>> i = 3;
>> x = 3*i

x =     9
```

Aber:

```
>> i = 3;
>> x = 3i

x =     0 + 3.0000i

>> y = 1i * x

y =     -3
```

MATLAB bietet verschiedene Funktionen und Operatoren zum Umgang mit komplexen Zahlen an. Die wichtigsten werden im Folgenden aufgelistet:

`real(c)` erzeugt den Realteil der komplexen Zahl c.
`imag(c)` erzeugt den Imaginärteil der komplexen Zahl c.
`abs(c)` erzeugt den Betrag der komplexen Zahl c.
`angle(c)` erzeugt die Phase im Bogenmaß der komplexen Zahl c im Bereich $-\pi$ bis π.
`conj(c)` erzeugt die konjugiert komplexe Zahl c^* der komplexen Zahl c.

Weiter Erläuterungen sind auch aus dem MATLAB Primer zur Vorlesung Mathematische Methoden der Elektrotechnik zu entnehmen.

3 Durchführung

Alle Teilaufgaben sind schriftlich zu protokollieren. Außerdem sind für die Aufgaben 3.1 und 3.3 die bereitgestellten MATLAB Templates zu benutzen.

3.1 Motormessungen

Dauer: ca. 90 Minuten

a) Verbinden Sie einen EV3 Motor mit Port A des Bricks durch ein schwarzes Verbindungskabel. Bauen Sie nun einen Zeiger aus LEGO Steinen an das sich drehende Motorrad. Befestigen Sie anschließend den Motor an dem Brick so, dass der Zeiger frei rotieren kann.

b) Führen Sie nun folgende Motormessungen durch. Schreiben Sie ein MATLAB Skript, welches den Motor nacheinander mit den `Power` Werten 30, 50 und 70 jeweils um 1000 Winkelgrad rotieren lässt. Benutzen Sie zunächst den Modus `'Coast'` der Motoreigenschaft `brakeMode`. Während der Motor rotiert, ist die aktuelle Winkelposition, der Zustand `IsRunning` des Motors und der aktuelle Messzeitpunkt in Sekunden zu ermitteln und in eine geeignete Matrix abzuspeichern (**Tipp:** Verwenden Sie hierzu eine `while`-Schleife in Verbindung mit einer zusätzlichen Indexvariablen, um die Anzahl der Messungen festzustellen.) Benutzen Sie für die Zeitmessung die MATLAB Funktionen `tic` und `toc` und begrenzen Sie die maximale Messdauer auf 6 Sekunden. Nachdem alle Messwerte aufgenommen sind, tragen Sie die gemessenen Winkelpositionen über der Zeit und eine waagerechte Linie bei 1000 Winkelgrad in einem Grafikplot auf. Fügen Sie außerdem die Graphen der `IsRunning` Zustände hinzu (**Hinweis:** Berücksichtigen Sie zur Darstellung eine geeignete Skalierung). Zuletzt beschriften Sie die gezeigten Graphen mit einer Legende und speichern den gesamten Grafikplot als MATLAB Figure ab.

   Beantworten Sie nun folgende Fragen:

   1) Wie groß sind die bleibenden Abweichungen zwischen Ist-Position und Soll-Position bei den drei Power Werten, wenn der Motor ausgerollt ist?
   2) Zu welchem Zeitpunkt hat der Motor die richtige Soll-Position erreicht? Benutzen Sie dafür den *Data Cursor* der MATLAB Figure. Untersuchen Sie auch den Werteverlauf des `IsRunning` Motorzustands zu dem ermittelten Zeitpunkt.
   3) Begründen Sie nun anhand Ihrer gewonnen Erkenntnisse die Abweichungen aus Teilaufgabe 1.

   Hinweis: Benutzen Sie für Ihr Skript das Template `motor_exercise_1bc.m`.

c) Verwenden Sie nun den Modus `Brake` der Motoreigenschaft `brakeMode` und führen den Versuch 4.1 b) noch einmal durch. Speichern Sie am Ende Ihrer Messungen wieder den Plot als eine neue MATLAB Figure ab.

Beantworten Sie erneut die Fragen 1-3 aus Versuch 3.1 b). Vergleichen Sie die Ergebnisse mit Teilaufgabe b).

d) Machen sie sich die Wirkung der Eigenschaft `speedRegulation` der Motorklasse klar. Erzeugen sie dazu zwei Objekte für einen Motorport ihrer Wahl. Nutzen sie die Parameterkombination `Power = 40, TachoLimit = 0, SmoothStart = false`. Die Eigenschaft `speedRegulation` soll mal auf `false`, mal auf `true` gesetzt werden. Verbinden sie eine Stange und Rad mit Reifen mit dem Motor. Jetzt starten sie den Motor wie bekannt und versuchen, ihn mit der Hand fest zu halten. Was beobachten sie bezüglich Drehmoment und der Kraft, die sie zum Blockieren benötigen? Welche Auswirkungen könnte die Einstellung von `speedRegulation` auf evtl. empfindliche Roboterarme oder Getriebe haben?

3.2 Getriebe

Dauer: ca. 30 Minuten

a) Gegeben sei das Getriebe in Abb. 2. Bestimmen Sie das Übersetzungsverhältnis zwischen Zahnrad A und D. Das Übersetzungsverhältnis ist definiert durch:

$$i = \frac{\text{Antriebsdrehzahl}}{\text{Abtriebsdrehzahl}} = \frac{\text{Abtriebszähnezahl}}{\text{Antriebszähnezahl}}$$

Hinweis: Insgesamt für das Getriebe in Abb. 2 wurden 3 Zahnräder mit 8 Zähnen, 1 Zahnrad mit 12 Zähnen, 4 Zahnräder mit 24 Zähnen und 3 Zahnräder mit 40 Zähnen verwendet.

b) Welchen Drehwinkel erfährt Zahnrad B, C und D, wenn Zahnrad A um eine ganze Umdrehung rotiert wird?

c) Bauen Sie aus LEGO Bausteinen ein neues Getriebe, welches das gleiche Übersetzungsverhältnis aus Teilaufgabe a) besitzt, aber möglichst wenige Zahnräder verwendet.

d) Welche Unterschiede weist das neue Getriebe aus Aufgabenteil c) zu dem aus Abbildung 2 auf?

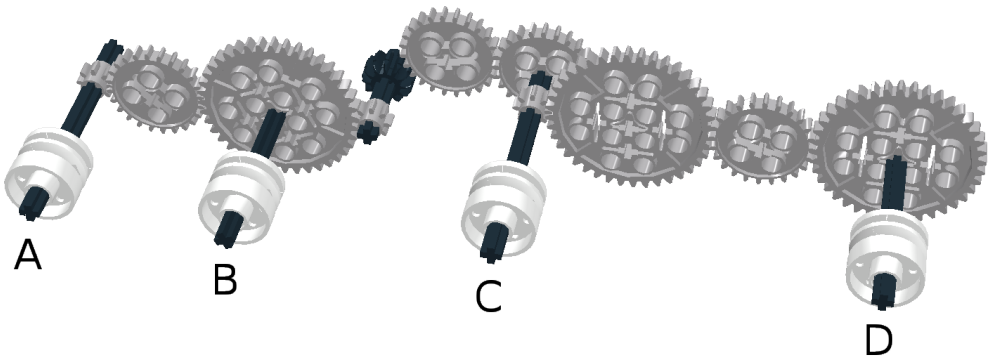


Abbildung 2: Getriebe

3.3 Zahlendarstellung

Dauer: ca. 120 Minuten

a) Bauen Sie die Mindstorms Phasor-Maschine nach der gegebenen Anleitung.

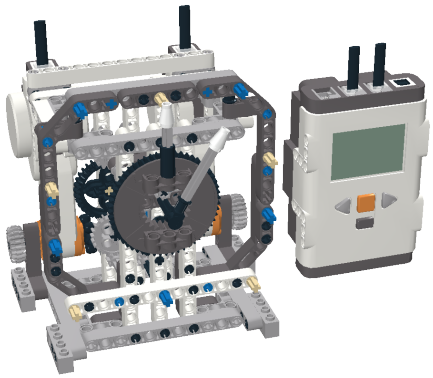


Abbildung 3: Phasor-Maschine

b) Bestimmen Sie alle drei Übersetzungsverhältnisse zwischen Motor und den Zeigern 1, 2 und 3.

c) Programmieren Sie ein MATLAB Skript, welches die Summe zweier Zahlen grafisch durch ein Ziffernblatt dargestellt. Die beiden Zahlen sind dabei durch ein Input-GUI-Dialog (`inputdlg`) einzugeben. Die Summe der Zahlen ist dann durch zwei Zeiger in einem Kreisdiagramm (Abb. 4) mit den Ziffern 0 bis 9 (vgl. Ziffernblatt einer Uhr) darzustellen. Der kürzere erste Zeiger repräsentiert die Zehnerstelle und der zweite die Einerstelle. Die Zeiger sollen ausschließlich direkt auf die Zahlen zeigen. Die Zwischenbereiche sind nicht anzufahren. Zum Plotten des Kreisdiagramms (Abb. 4) ist die bereitgestellte MATLAB Funktion `plot_number_face` zu verwenden.

   Nach der Darstellung des MATLAB Plot, sind auch die Zeiger der LEGO Phasor-Maschine gemäß der berechneten Winkel zu rotieren. Dabei sollen die Zeigerpositionen des MATLAB Plots und der Maschine übereinstimmen.

   Hinweis: Benutzen Sie für Ihr Skript das Template `motor_exercise_3c.m`.

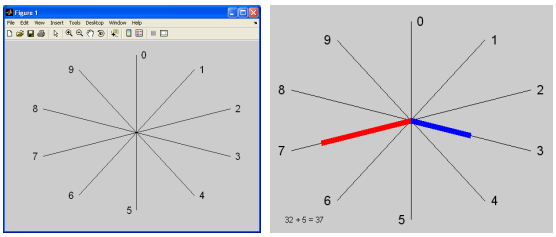


Abbildung 4: Leeres Zeigerdiagramm (links), Ergebnisbeispiel $32 + 5 = 37$ (rechts).

MATLAB Template für Aufgabe 3.3 c

```
%% Aufgabe Zahlendarstellung c)
% Template

%%  ----- MATLAB Calculation -----

%% Get two numbers from user dialog
% Tips:
% * use MATLAB command "inputdlg".
% * see MATLAB help for usage and more information.
% * convert the reponse cell array into numbers using "str2double"
%
% ... insert here your code


%% Calculate the summation of the two numbers
% ... insert here your code


%% Initialize figures
plot_number_face;   % plot calculator face figure
hold on             % hold on flag to plot more plots into the calculator face figure


%% Calculate pointers to plot
% Tips:
% * for line plotting only the start and end point of the line has to be given
% * the rotated pointers can be easily constructed by a complex number (value and phase)
% * the length of the complex vectors should be different for both pointers and less than one
% * note the number zero is located at the coordinates (x,y) = (0,1) or (0,i) respectively
% * take care to use degrees or radian
% * consider only angles which are related to the exact number position. Angles between two
% numbers should be neglected.
%
% ... insert here your code


%% Plot pointers into the figure
% Tips:
% * for line plotting only the start and end point of the line has to be given
% * use different colors for the pointers
%
% ... insert here your code


%% ----- Mindstorms NXT - Control -----

%% Program the Mindstorms machine
%
% ... insert here your code
```

3.4 Phasendarstellung komplexer Zahlen

Dauer: ca. 90 Minuten

a) Erstellen Sie ein MATLAB Skript, welches zwei komplexe Zahlen durch eine mathematische Operation verarbeitet. Die Eingabe der beiden Zahlen und die mathematische Operation soll dabei durch ein Input-GUI-Dialog (`inputdlg`) realisiert werden. In dem Dialogfenster sind drei Eingabefelder wie in Abb. 5 dargestellt, einzurichten.

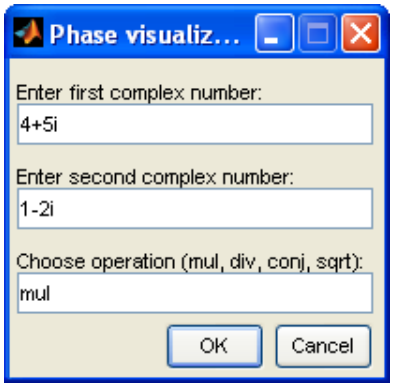


Abbildung 5: Input-GUI-Dialog

Die ersten zwei Eingabefelder sind für die beiden komplexen Zahlen und das dritte für die Angabe der mathematischen Operation vorgesehen. Die Operation soll durch die Worte `mul, div, conj` und `sqrt` spezifiziert werden, um eine Multiplikation, eine Division, die Berechnung der konjugiert komplexen Zahl und eine Wurzeloperation durchzuführen. Die Phasen der beiden komplexen Zahlen und der Ergebniswert (nach der mathematischen Operation) soll zum Einen mit Hilfe der MATLAB Plot-Funktion `compass` (Abb. 6) und mit Hilfe der Mindstorms Maschine aus Aufgabe 3.3 dargestellt werden.

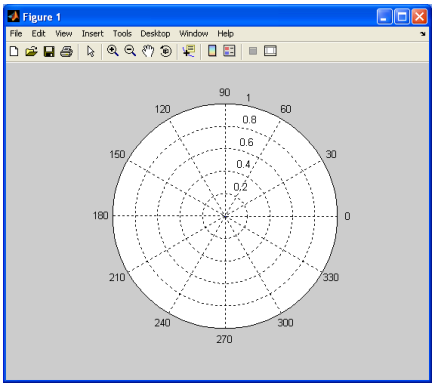


Abbildung 6: compass-Kreisdiagramm

Zur Bearbeitung der Aufgabe ist zunächst mit der MATLAB Plot-Funktion `compass` ein Kreisdiagramm zu plotten, in dem die Phasen der zwei komplexen Eingangszahlen und das Endergebnis dargestellt werden. Erst im zweiten Schritt sind dann die Zeigerpositionen mit der Mindstorms Maschine aus Aufgabe 3.3 anzuzeigen. Der Zeiger 1 soll dabei zuerst die Phase der ersten komplexen Zahl darstellen und Zeiger 2 die Phase der zweiten komplexen Zahl. Nach einer kurzen Pause soll dann Zeiger 1 die Phase des Ergebnisses darstellen. Am Schluss sind beide Zeiger wieder auf die Ausgangsposition (beide Zeiger zeigen senkrecht nach oben) automatisch zurückzudrehen.

Implementieren Sie nun zunächst ausschließlich die mathematischen Operationen: Multiplikation und Division.

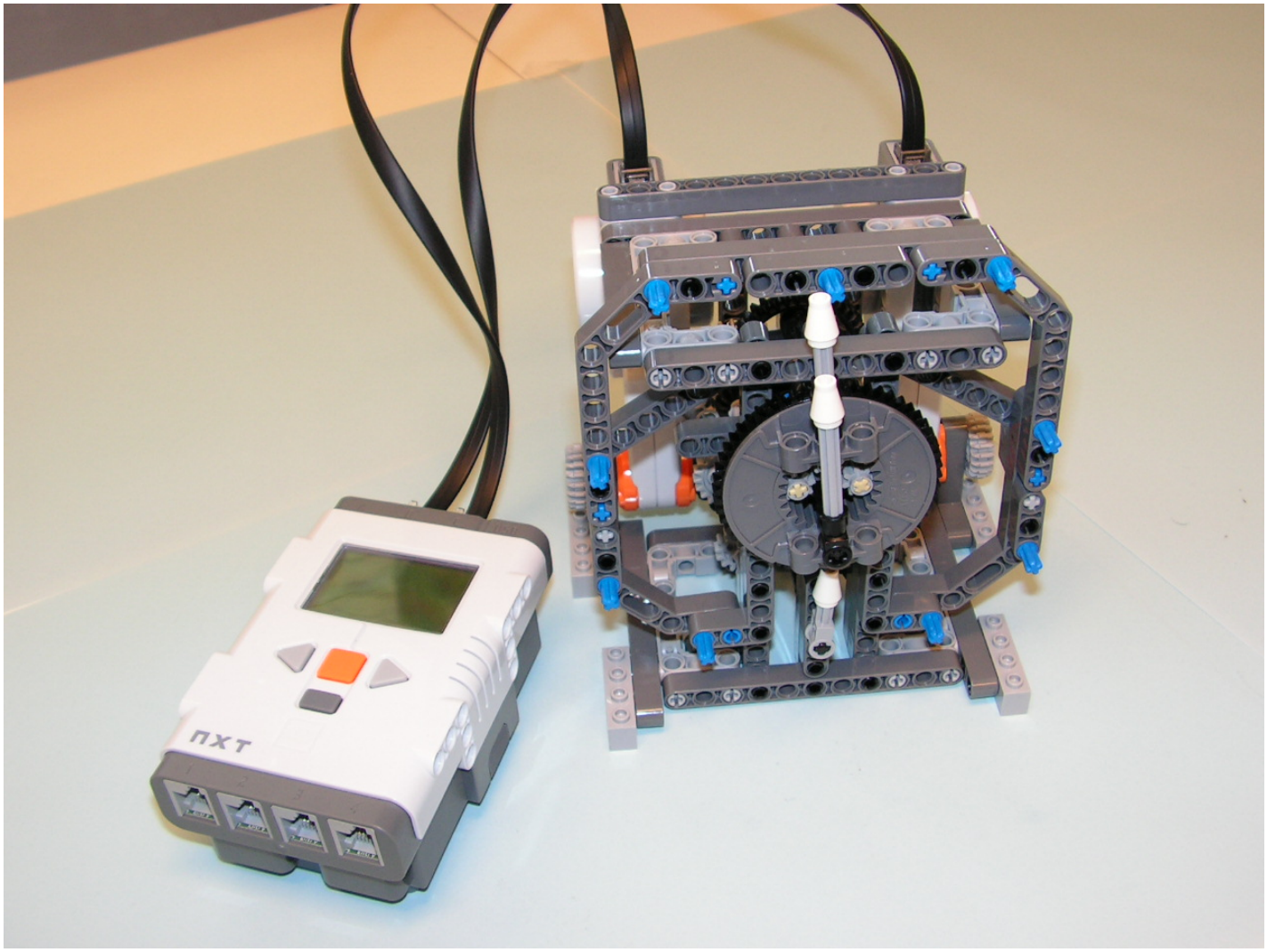
Hinweis: Um die Schlüsselworte `mul, div, conj` und `sqrt` im dritten Eingabefeld zu unterscheiden, können Sie die MATLAB Funktion `switch` verwenden. Genauere Informationen sind aus der MATLAB Hilfe zu entnehmen.

b) Welche Phasenänderungen erwarten Sie bei einer Multiplikation und Division zweier komplexer Zahlen? Überprüfen Sie Ihre Überlegungen mit der Ergebnisdarstellung aus Aufgabenteil a).

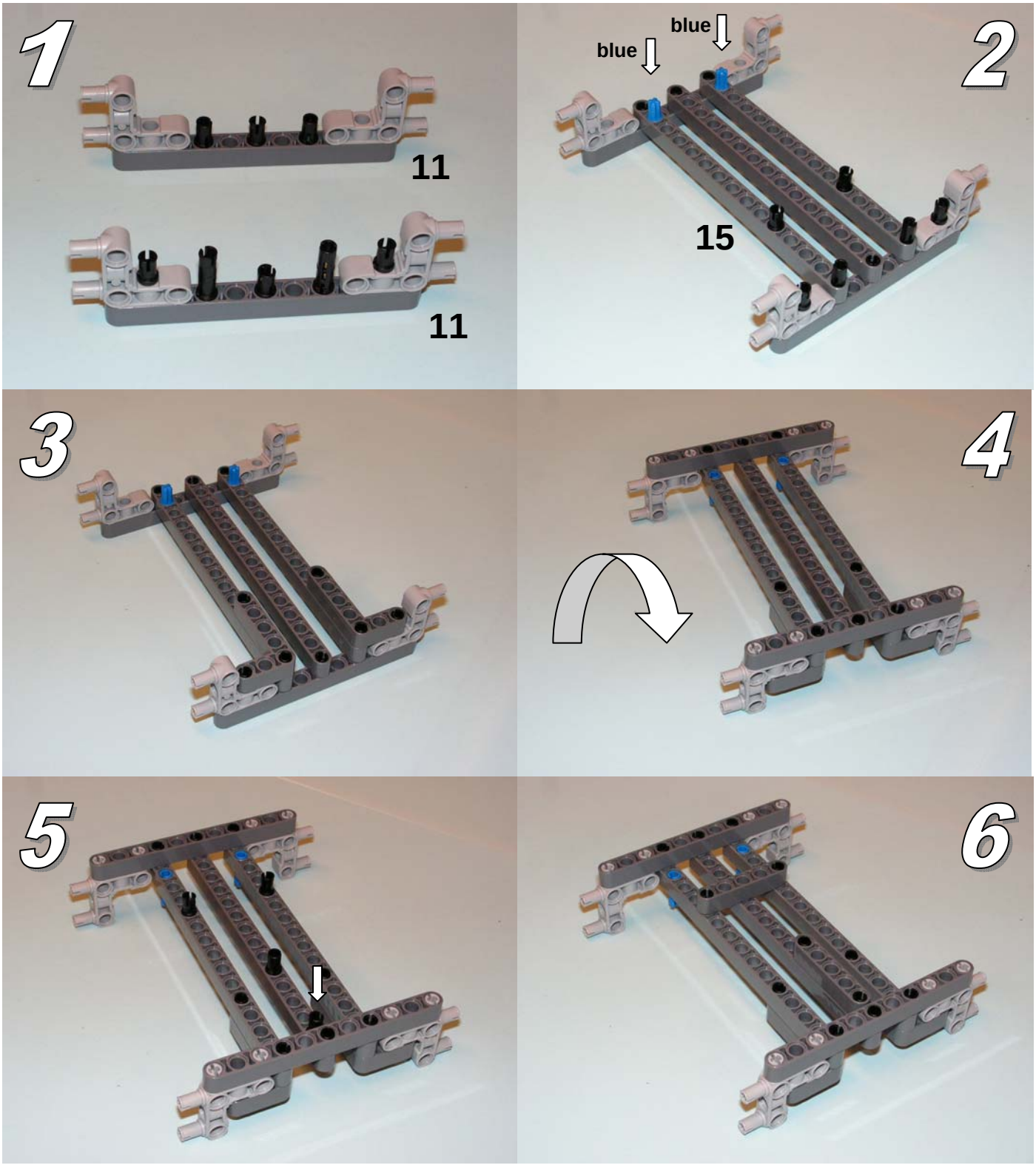
c) **Optional:** Erweitern Sie Ihr MATLAB Skript durch die mathematische Berechnung der konjugiert komplexen Zahl. Verwenden Sie dafür nur die erste komplexe Zahl in dem Input-GUI-Dialog. Stellen Sie die Phasen der Eingangszahl und dessen komplexe Konjugation dar. Welche allgemeine Phasenbeziehung besteht zwischen diesen?

d) **Optional:** Implementieren Sie die komplexe Wurzelberechnung für die erste Zahl in dem Input-GUI-Dialog. Berechnen Sie die ersten beiden Wurzeln. Als Beispiel überlegen Sie sich schriftlich die 2-ten Wurzeln der komplexen Zahl $z = 1 + i$ und stellen Sie die unterschiedlichen Phasen der Wurzeln sowohl im MATLAB Kreisdiagramm als auch mit der Mindstorms Maschine dar.

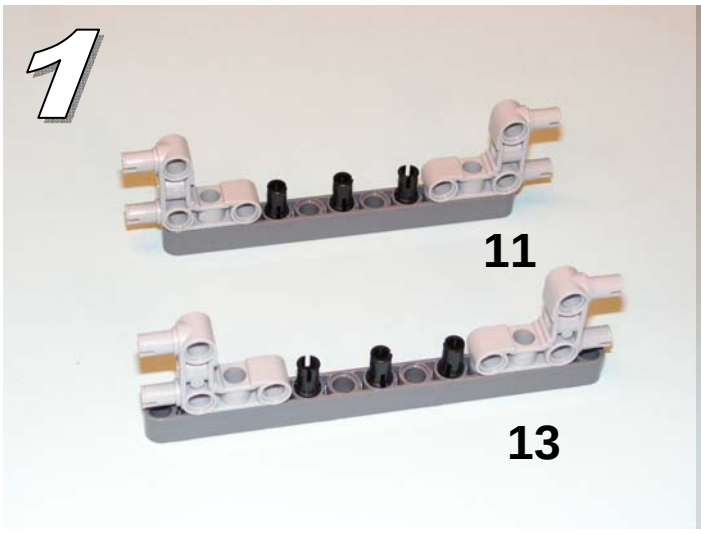
Building Instructions

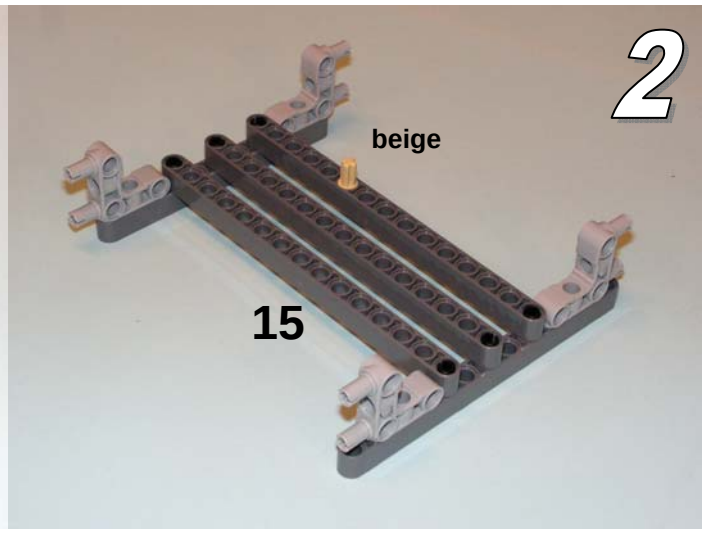


Part 1

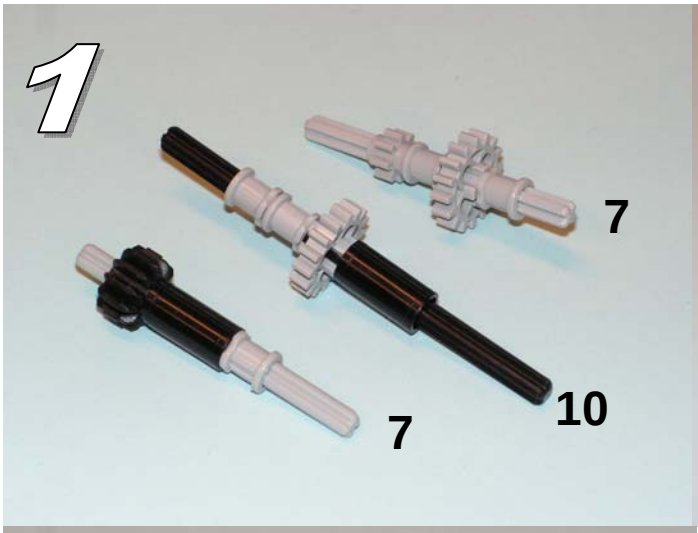


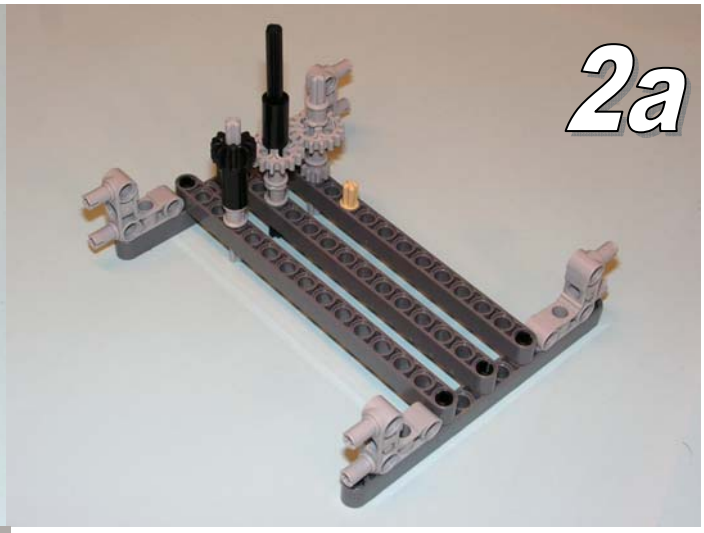
Part 2

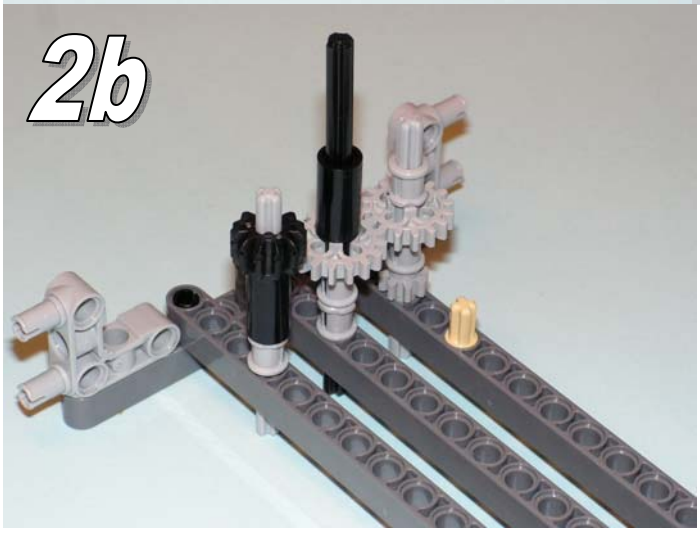




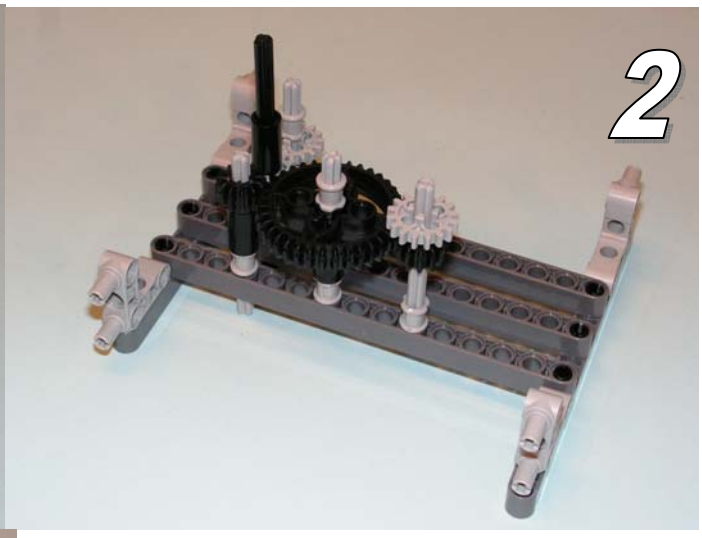
Part 3A

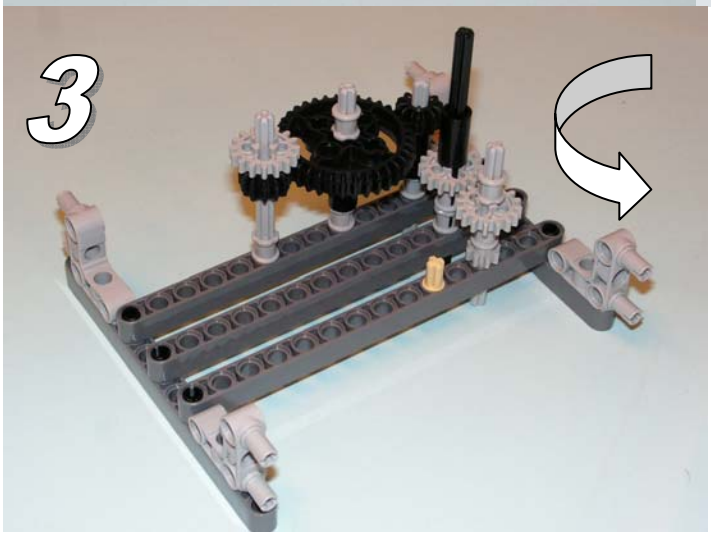




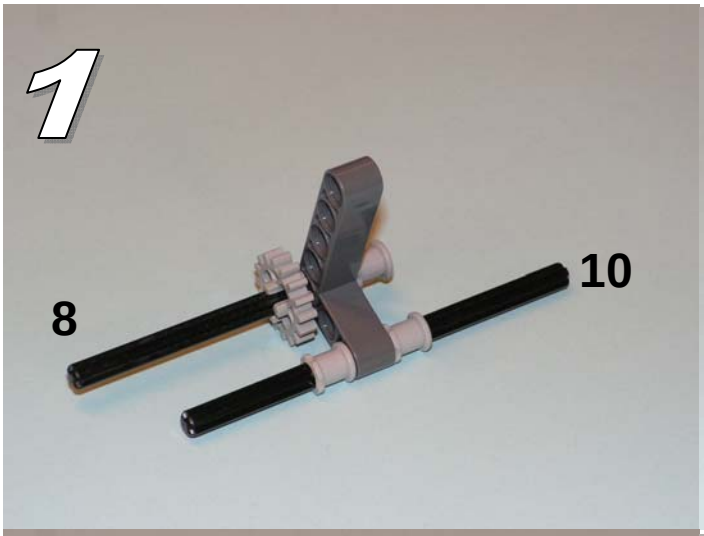


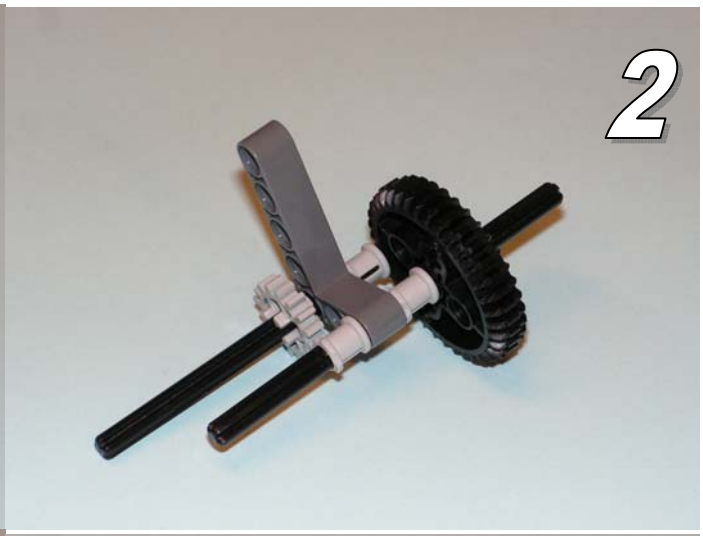
Part 3B

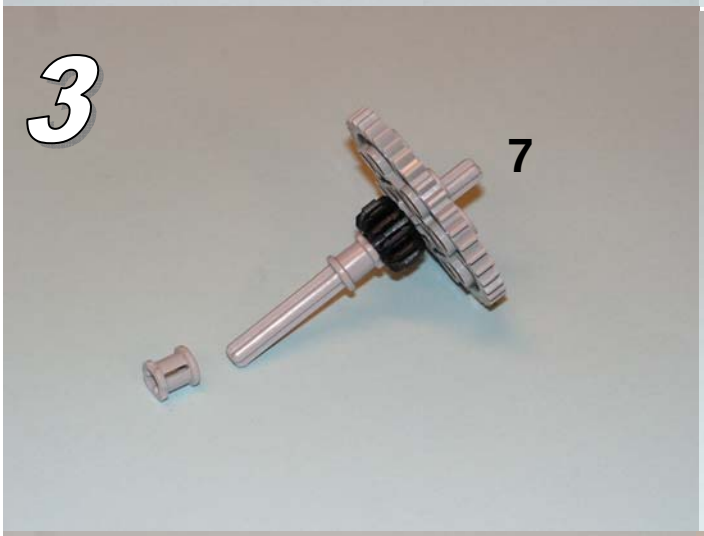


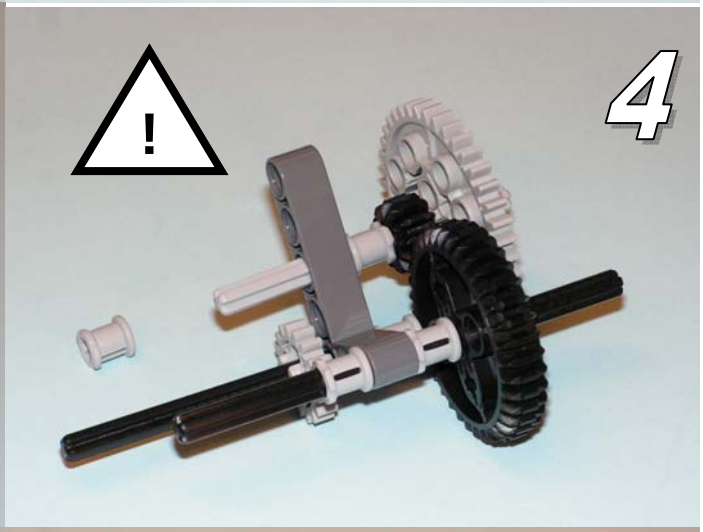


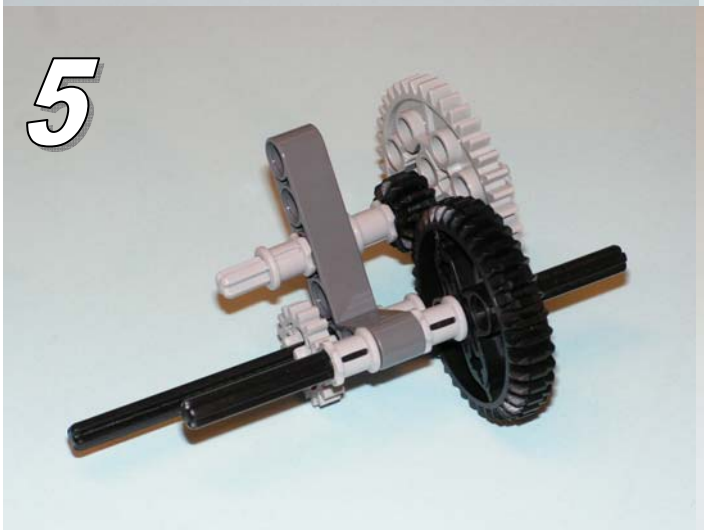
Part 3C

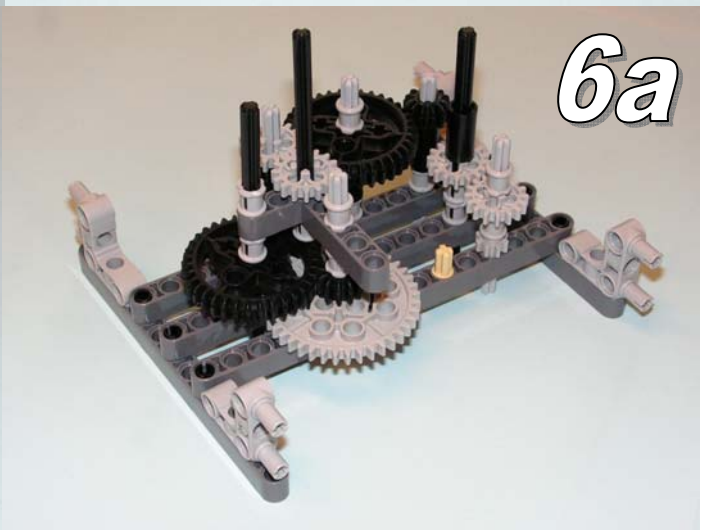


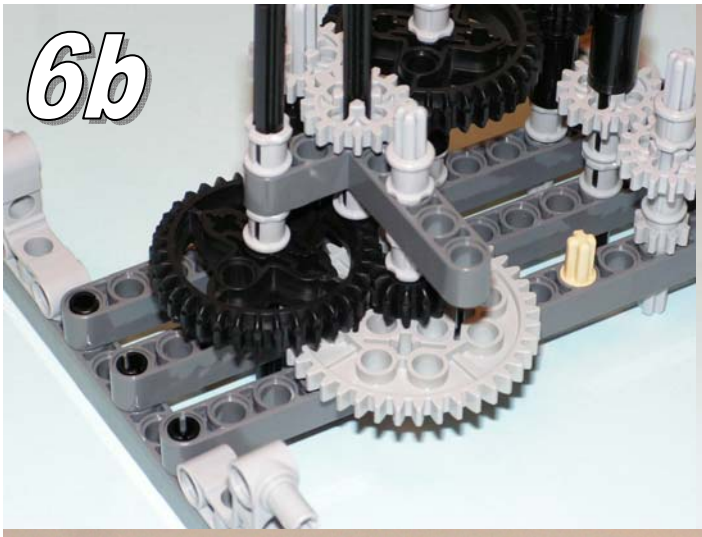


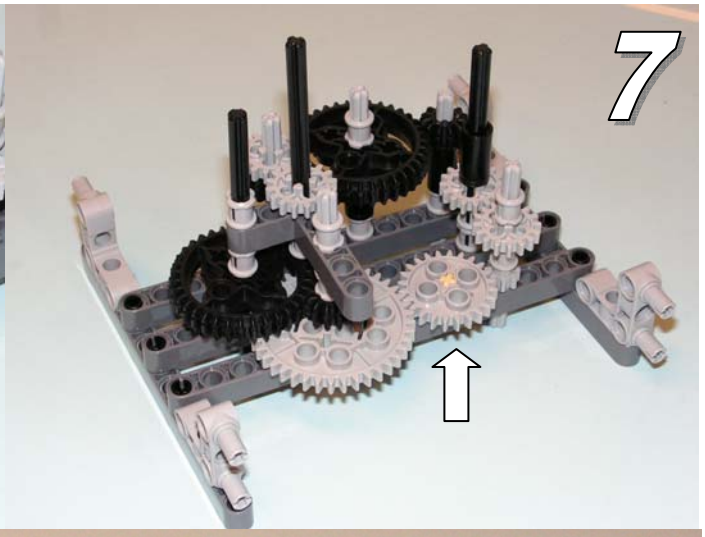


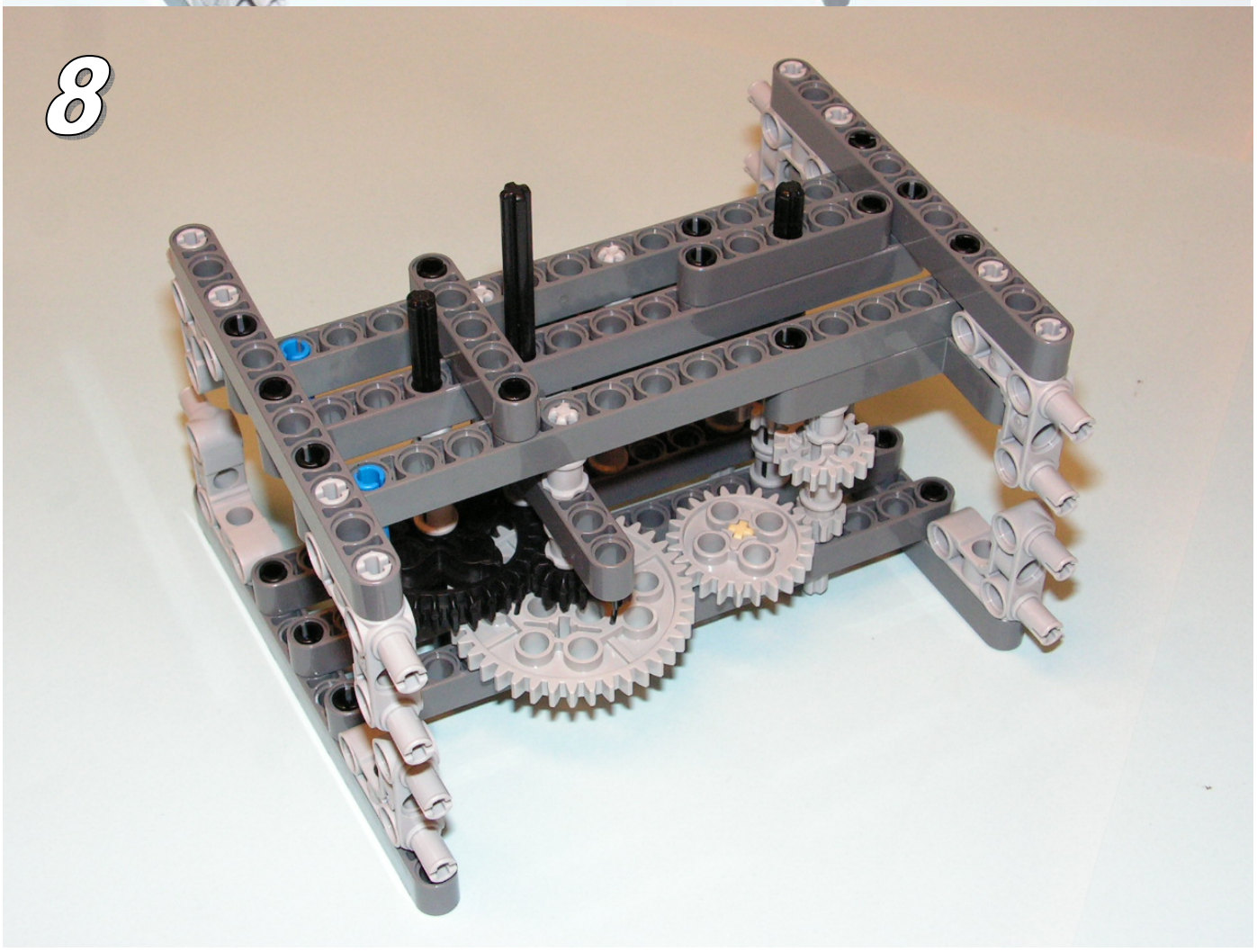




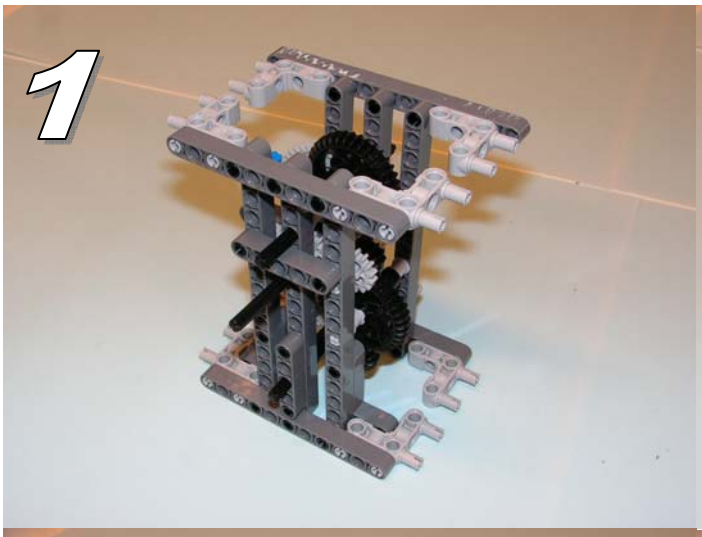


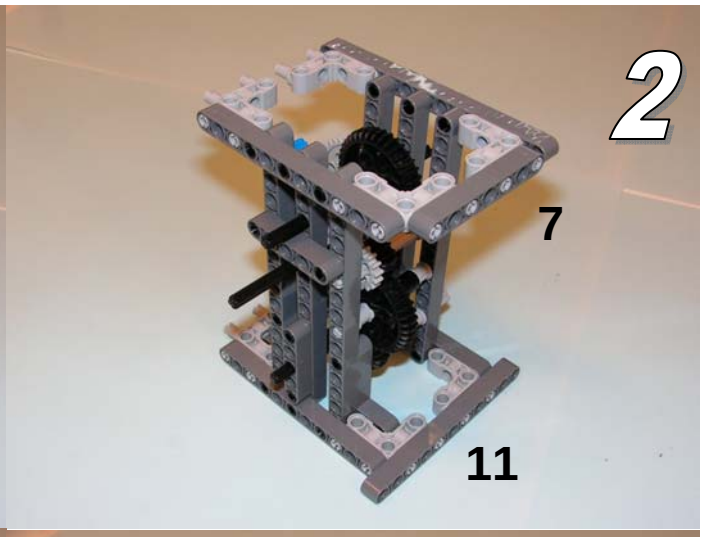

6b

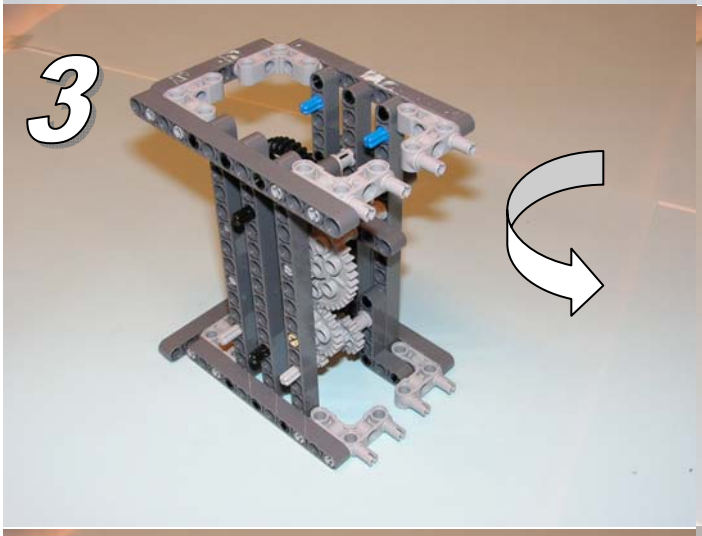

7

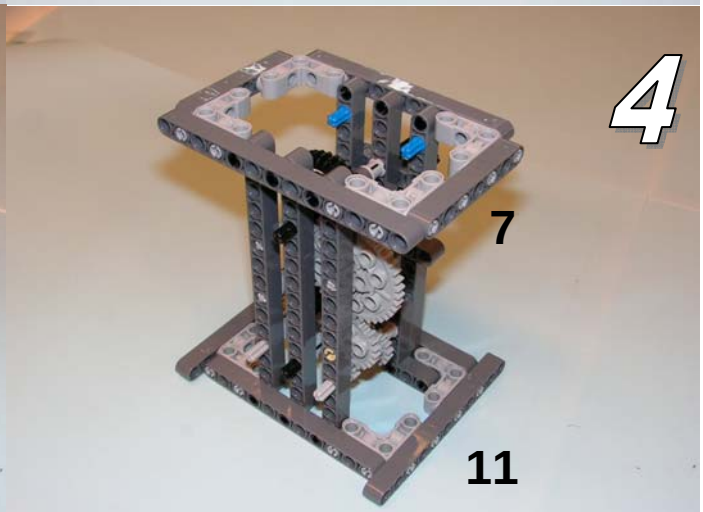

8

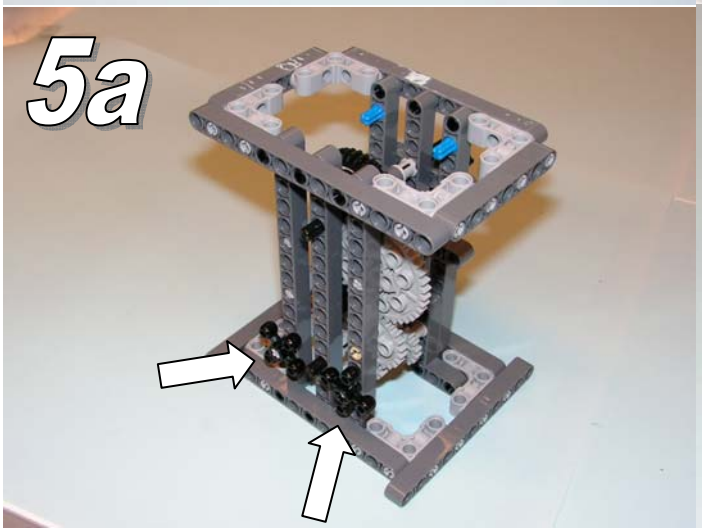
Part 4

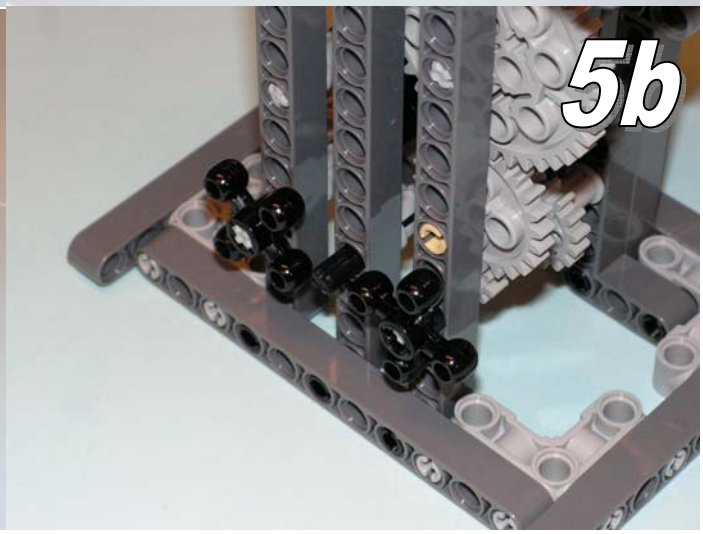




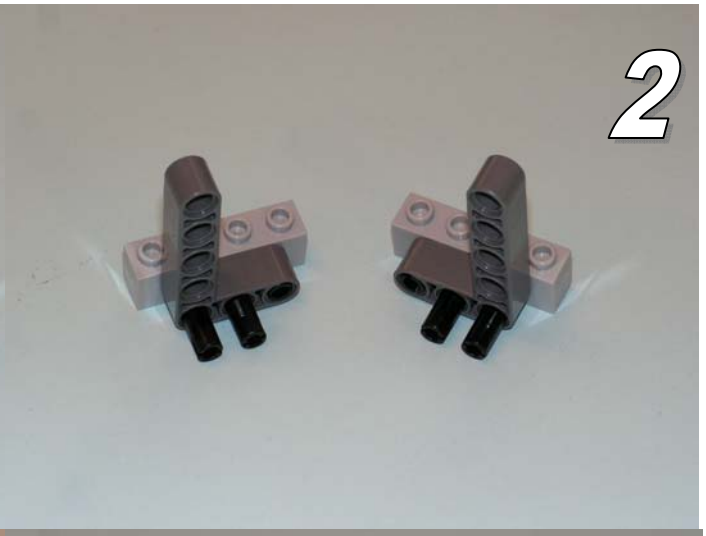


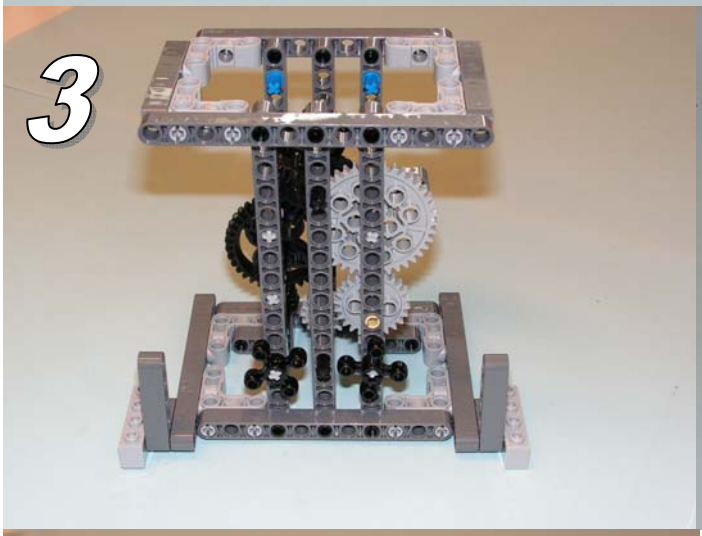


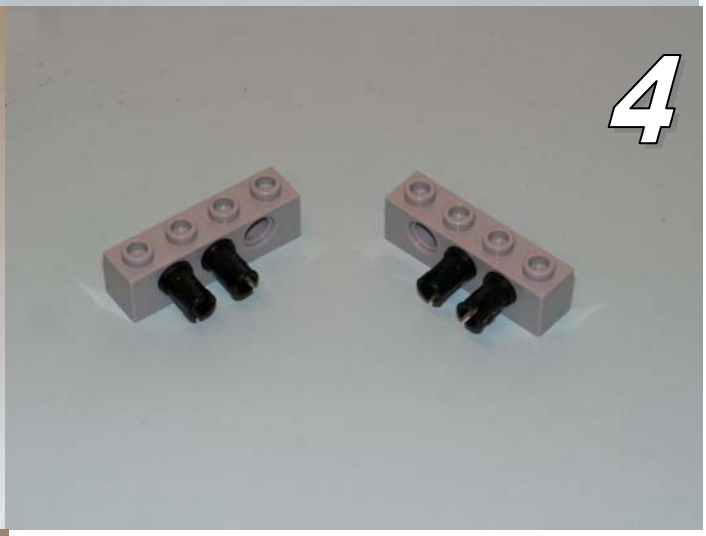


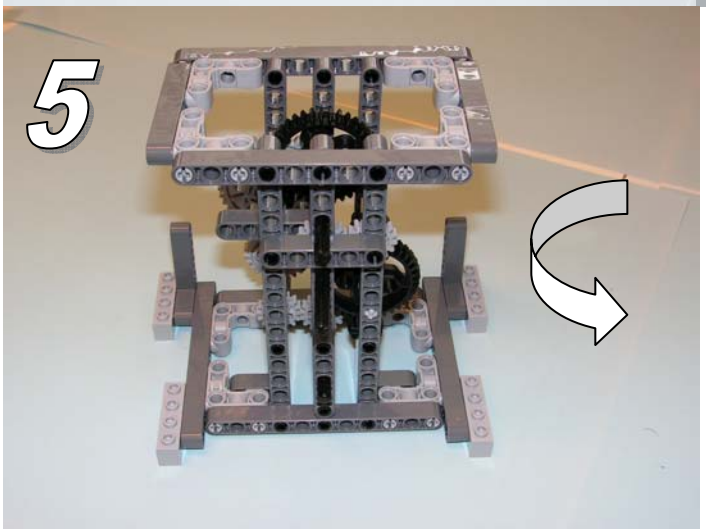


Part 5

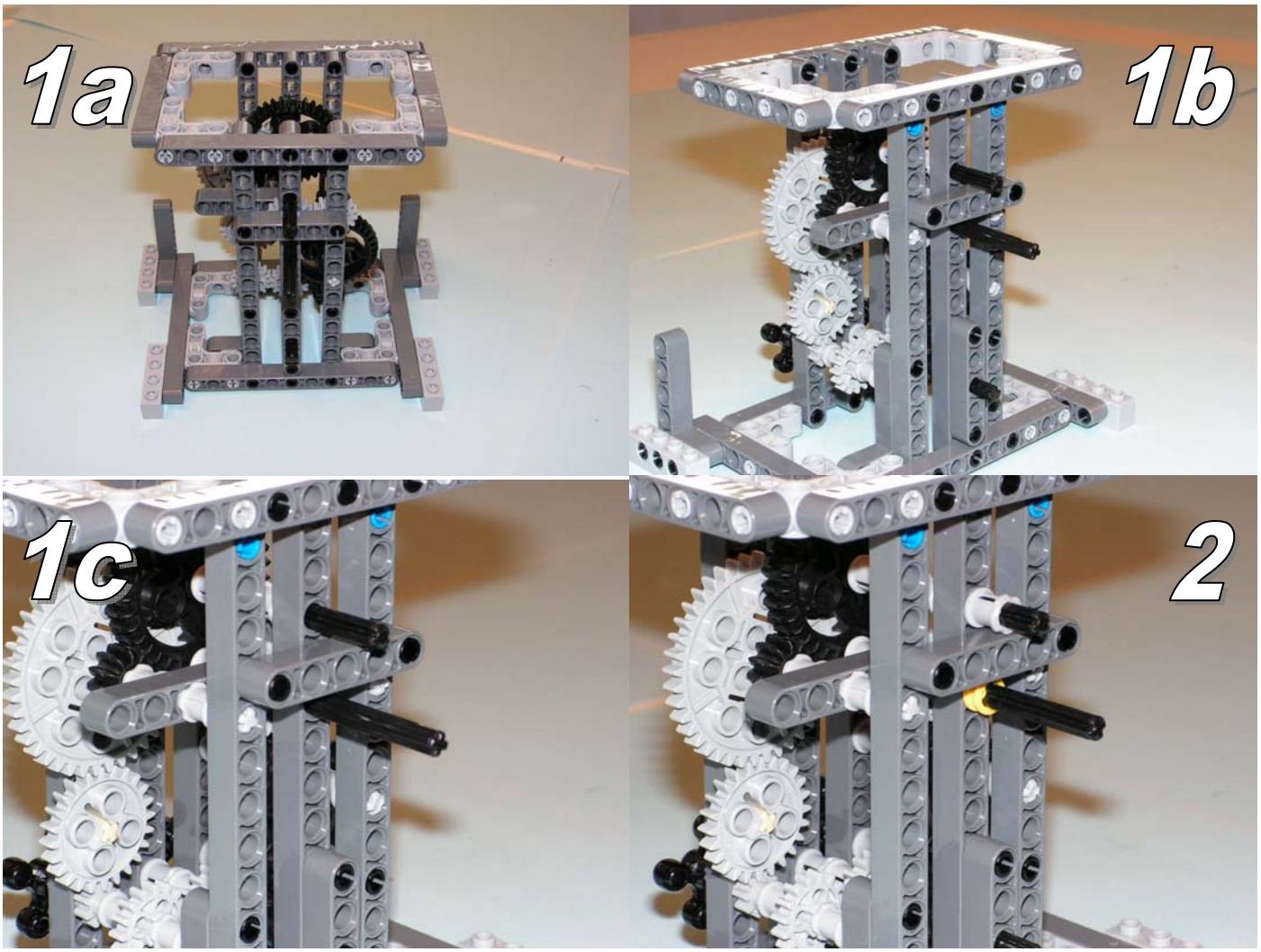




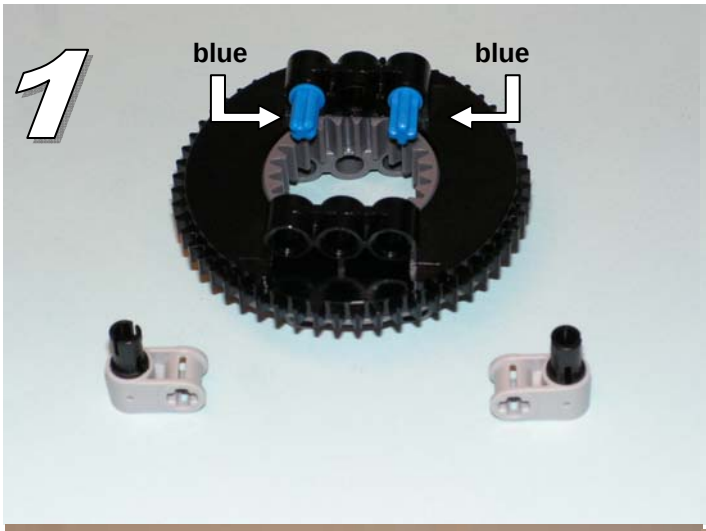


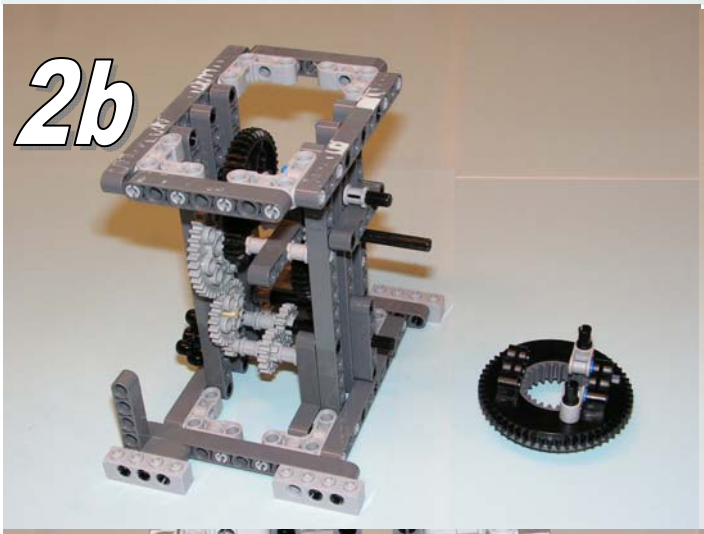


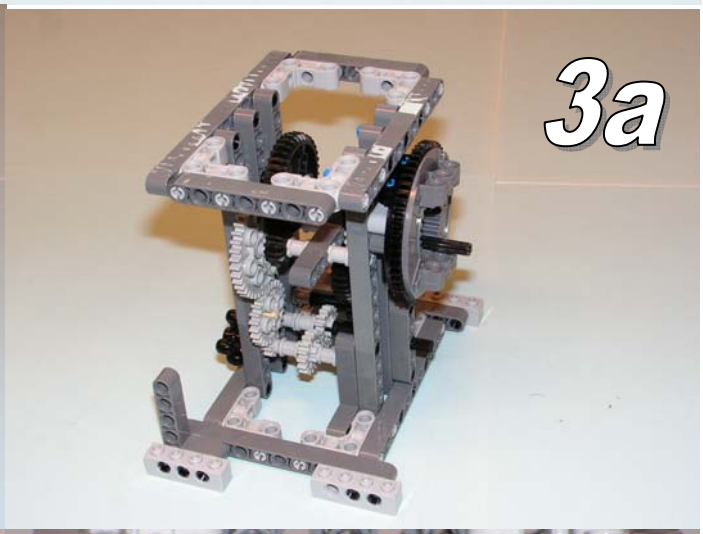
Part 6

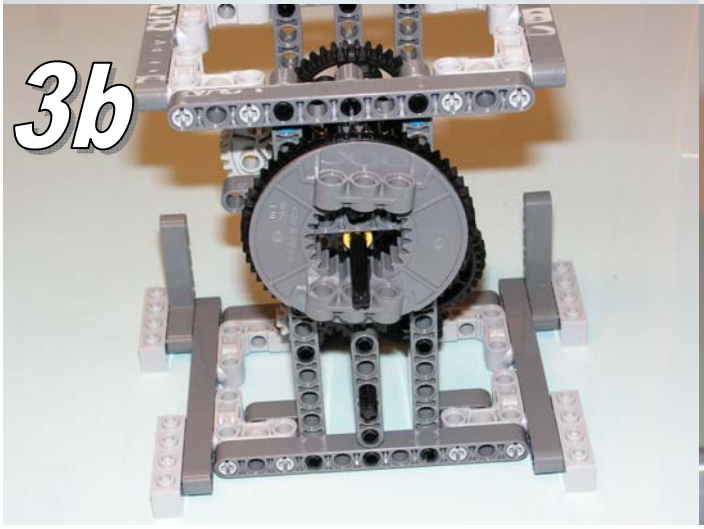


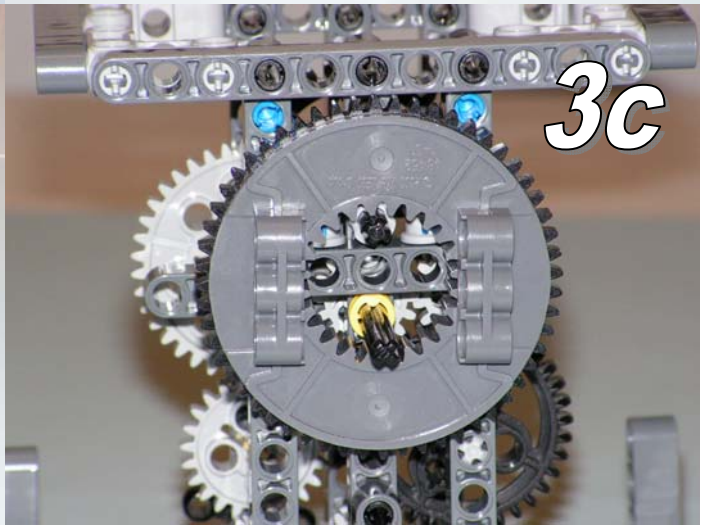
Part 7



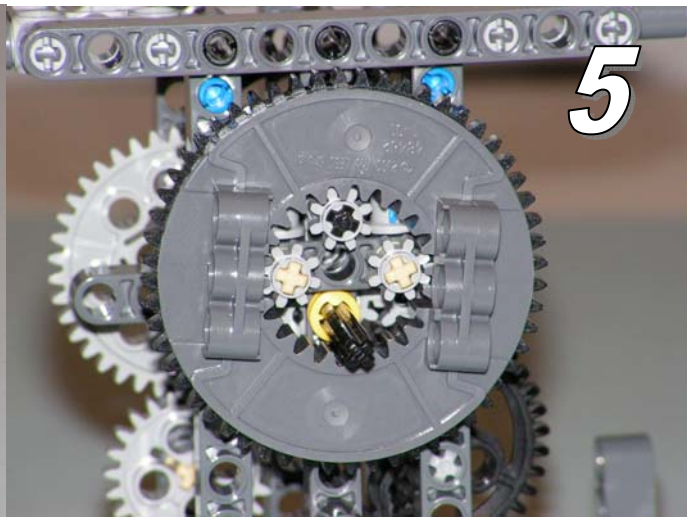




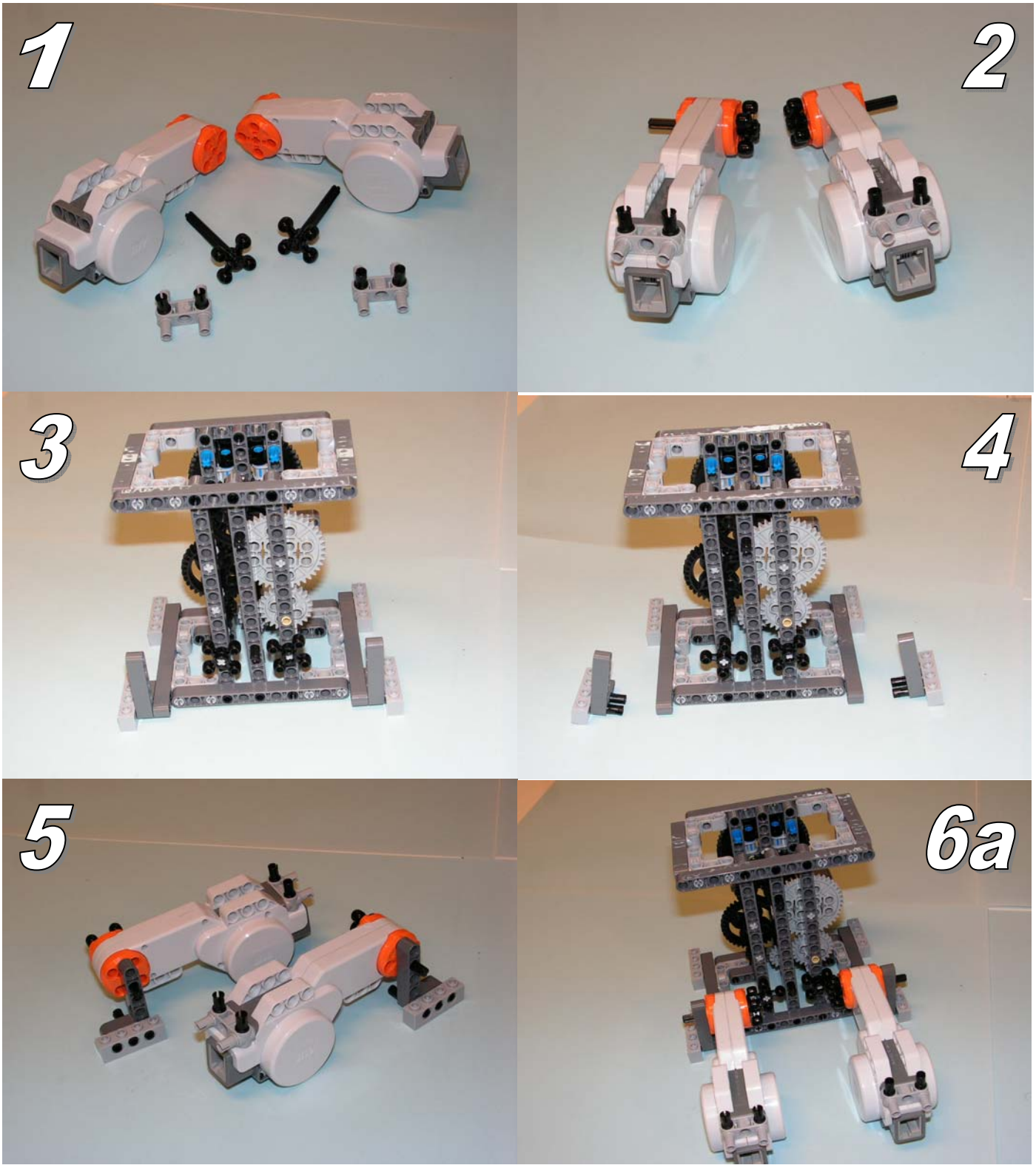


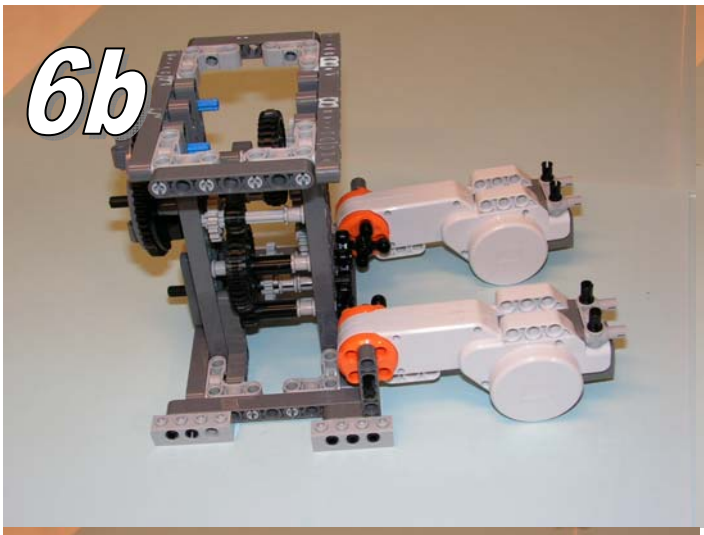


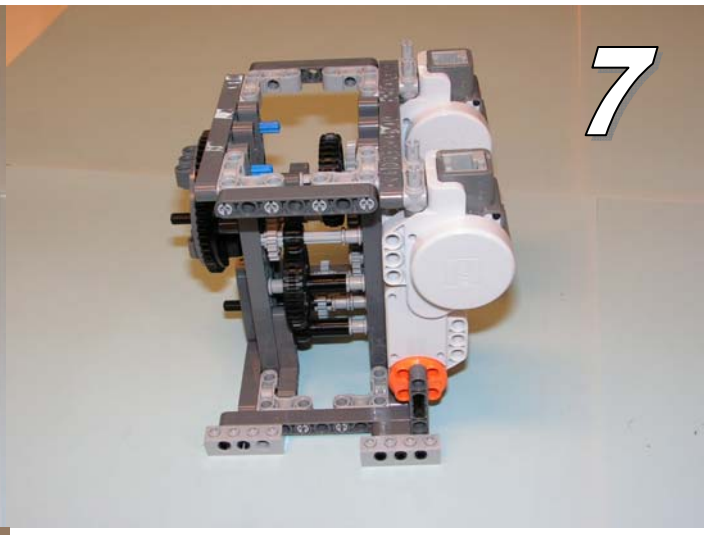

4
beige
beige

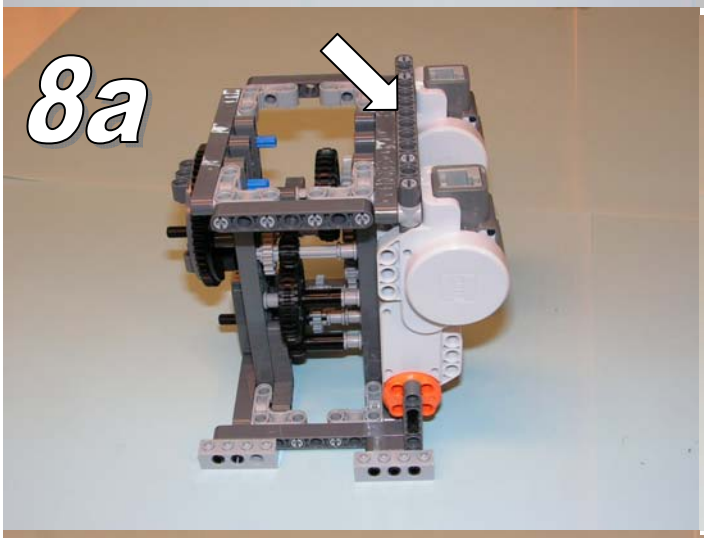

5

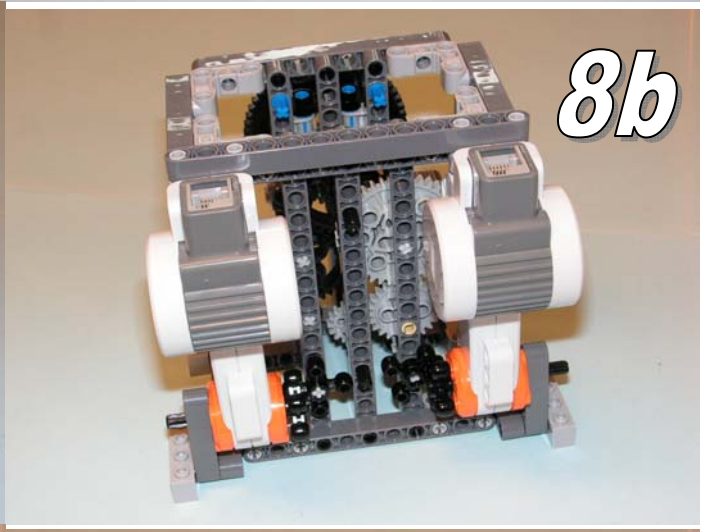
Part 8

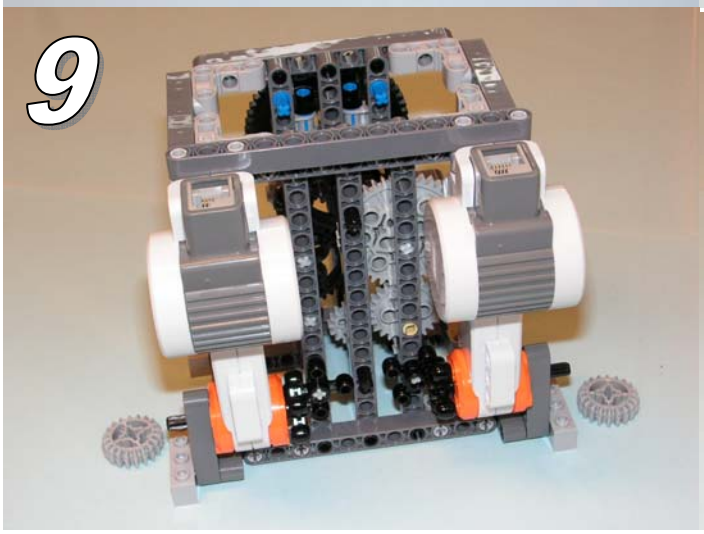


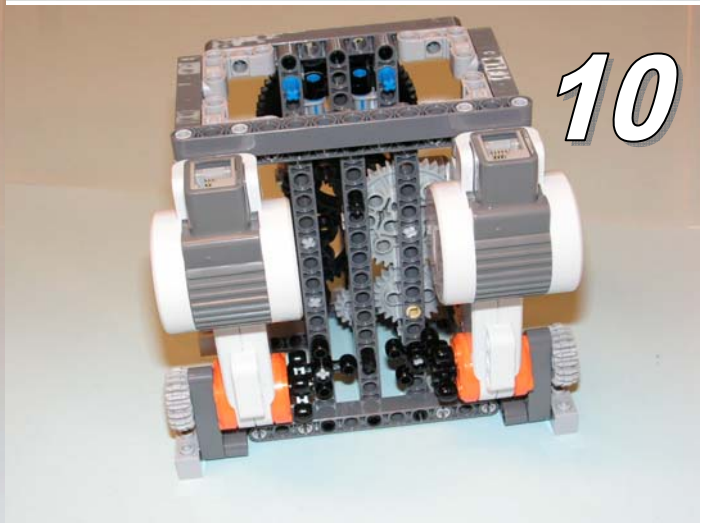

6b


7

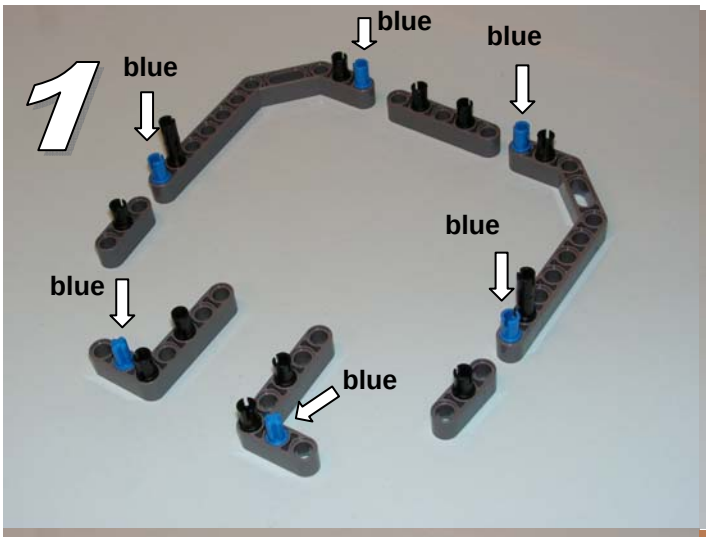

8a

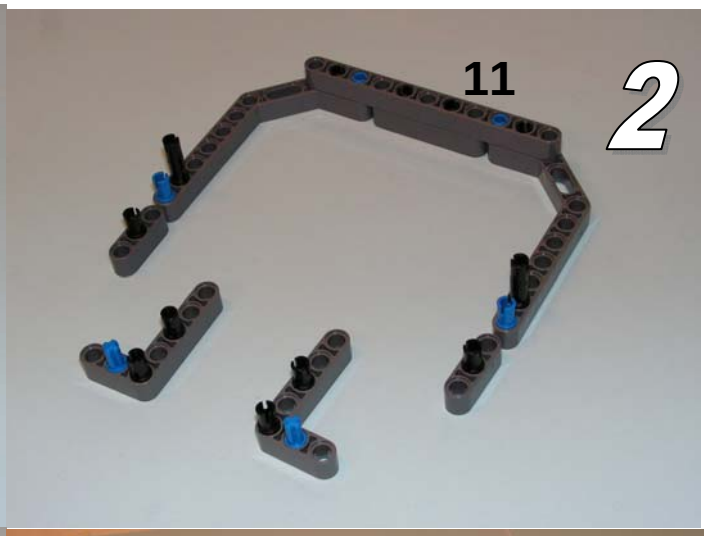

8b

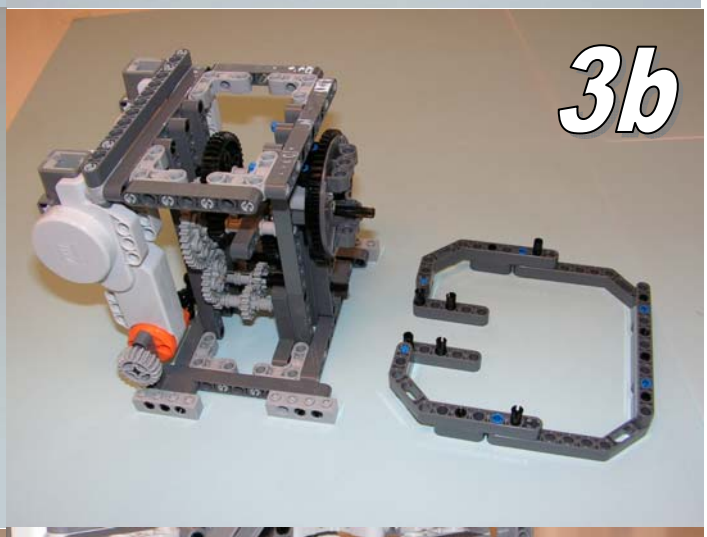

9

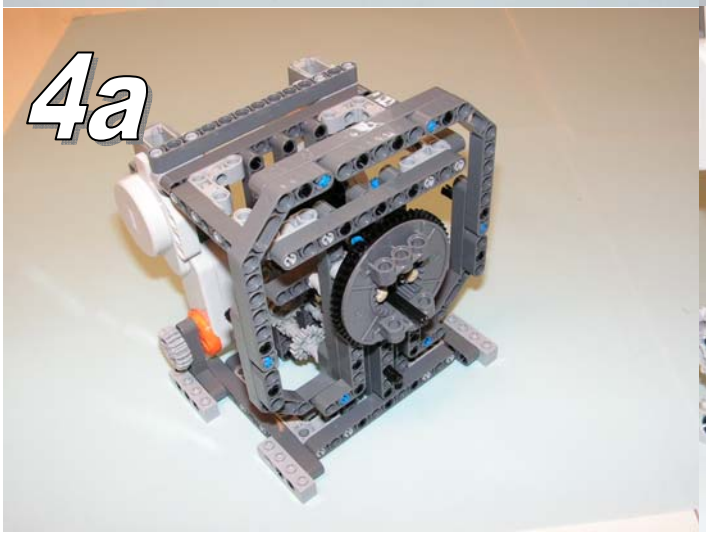

10

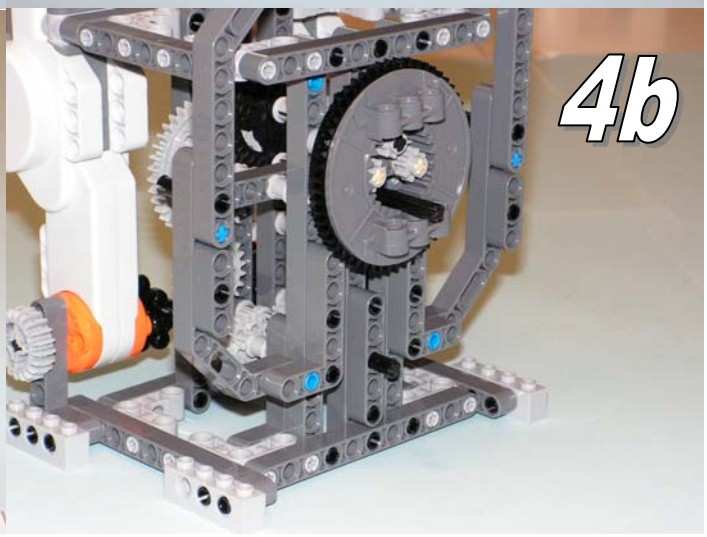
Part 9



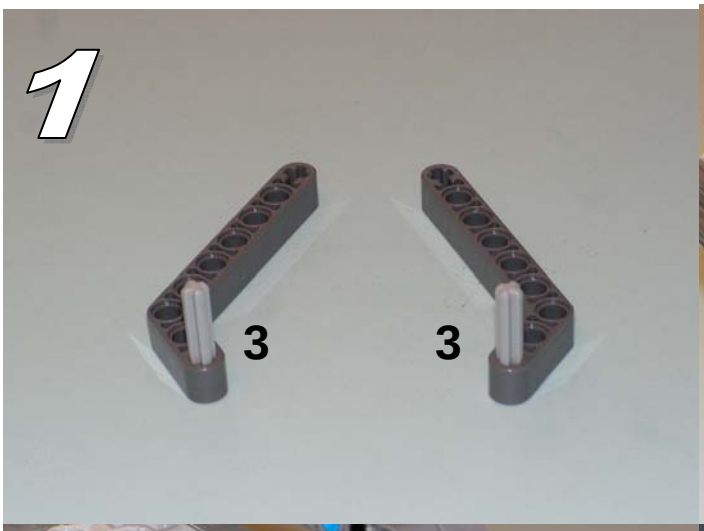


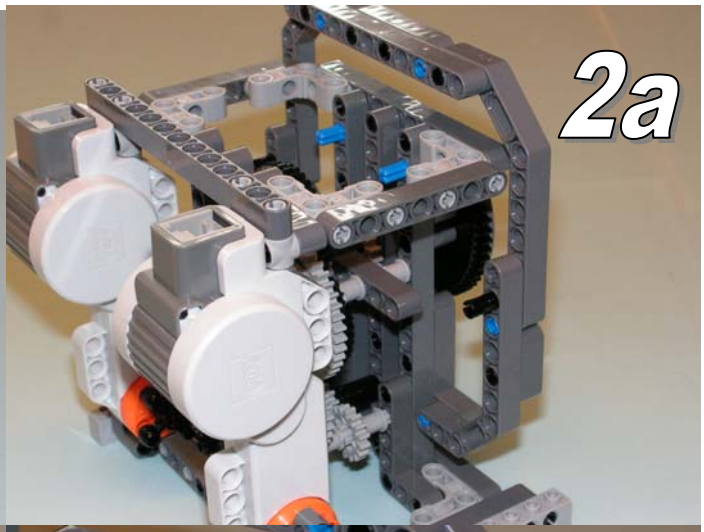


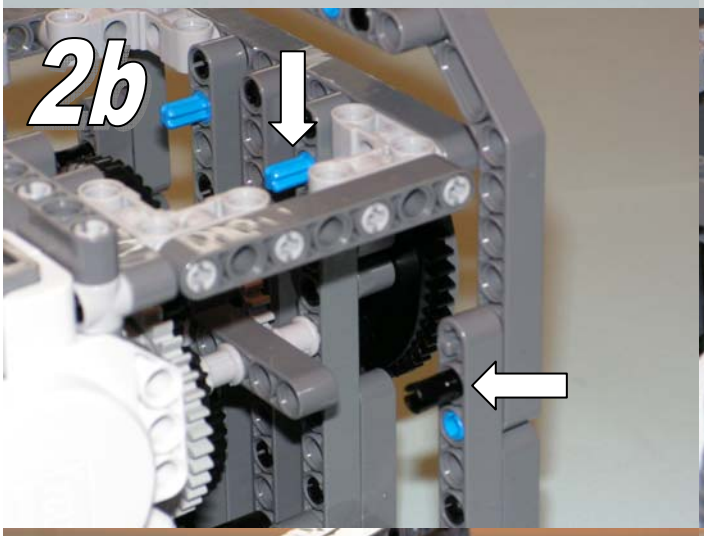


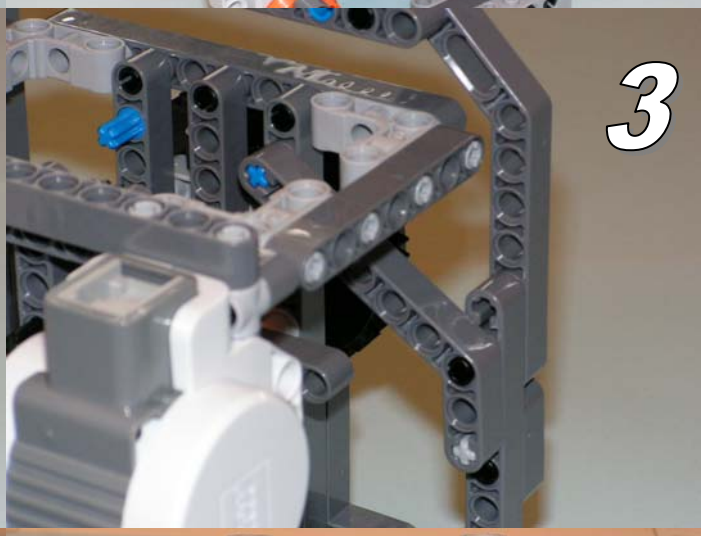


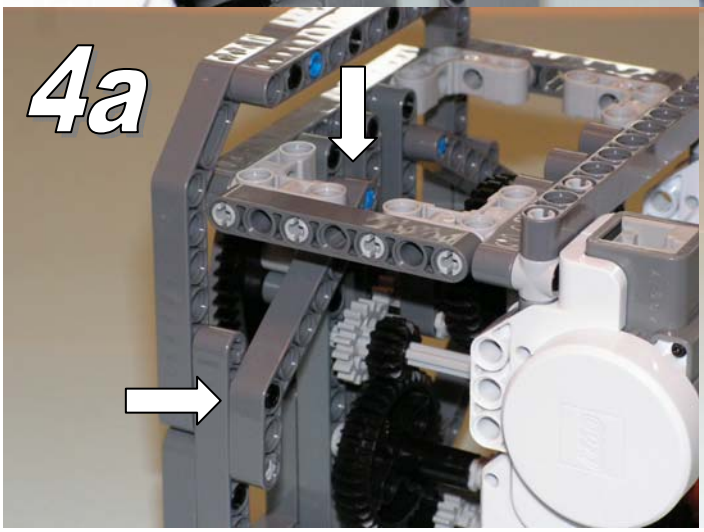
Part 10

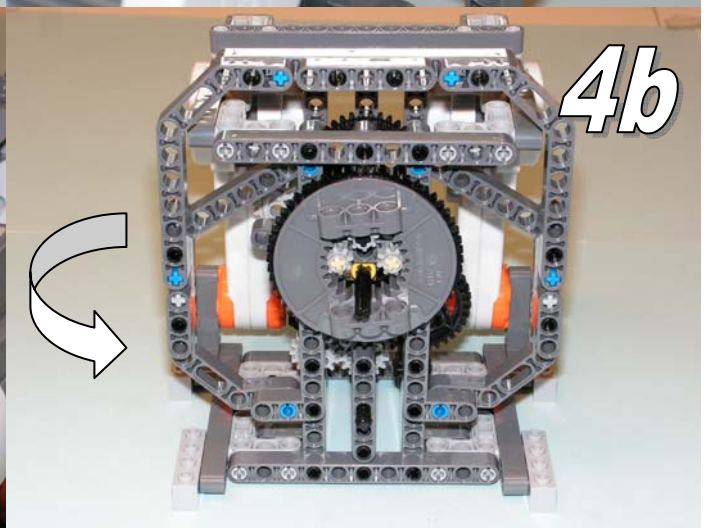




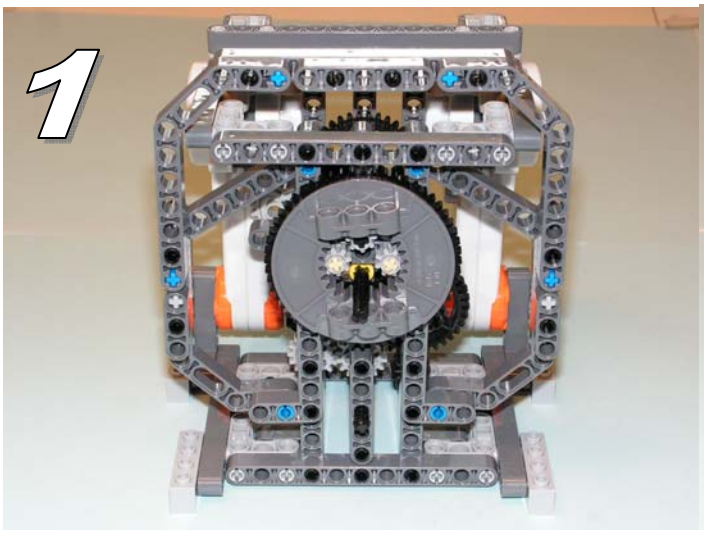


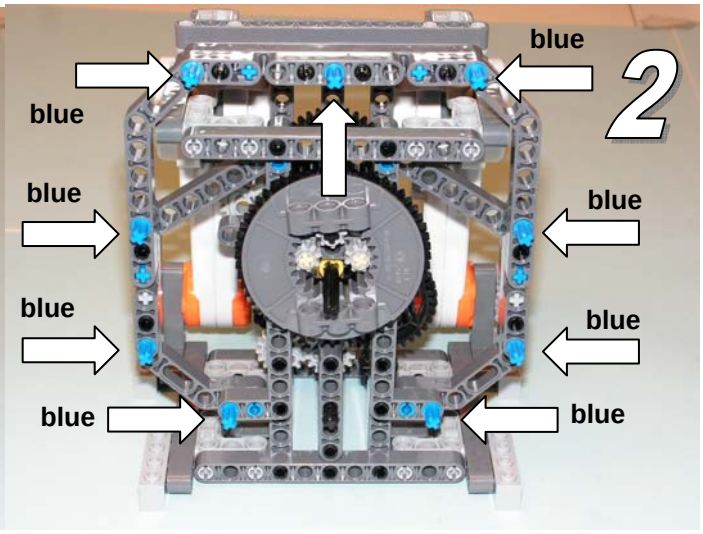




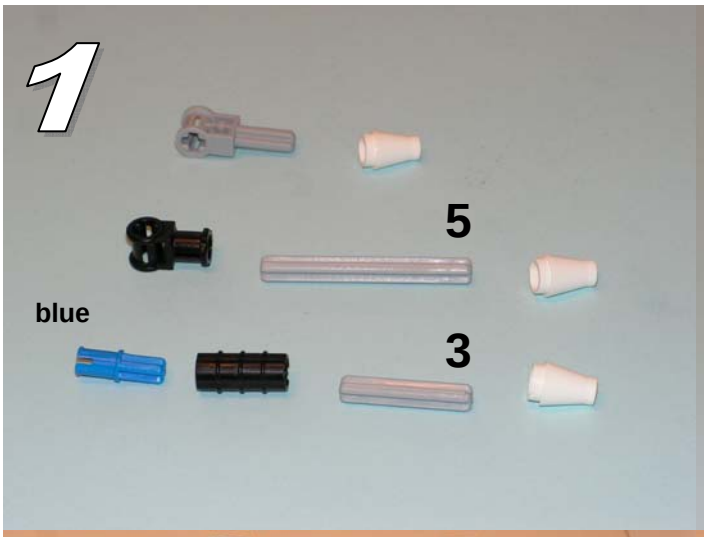


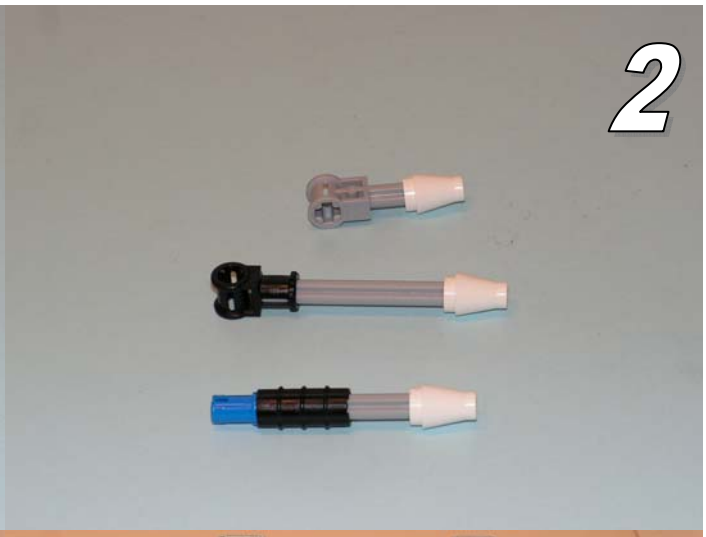
Part 11

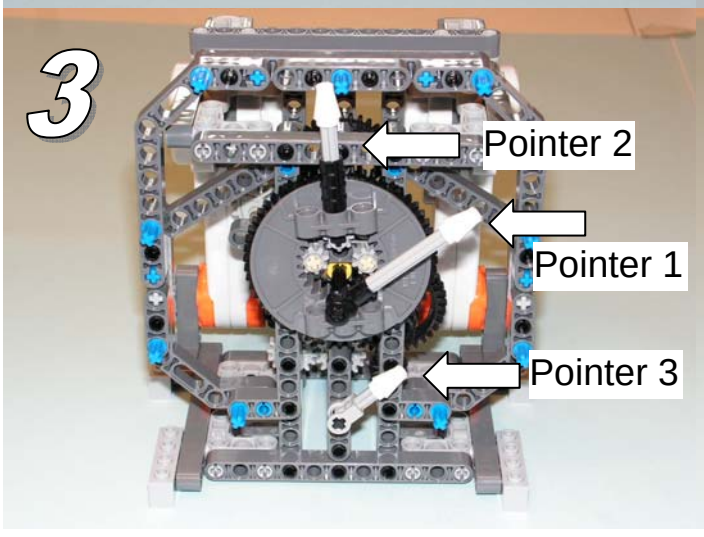


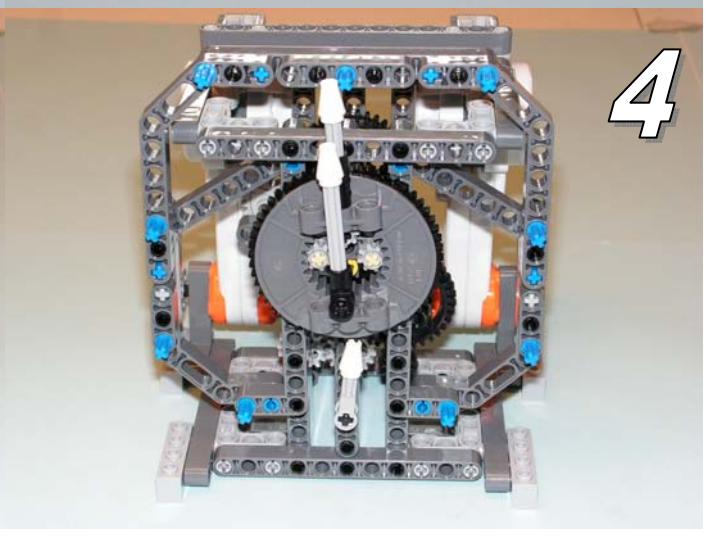


Part 12









Part 13