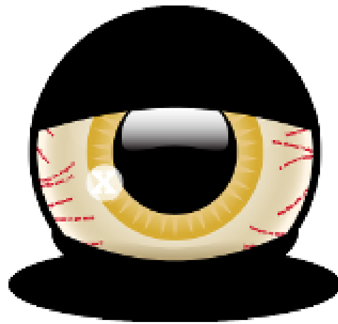


Challenge 24

10th Jubilee BME International 24-hour Programming Contest!

www.challenge24thelegend.com

Morgan Stanley

NetACADEMIA
A LEGJOBBAKAT TANÍTJUK

Contest

Welcome to the 10th BME International 24-hour Programming Contest!

Rules

During the EC many teams commented that they just skipped the rules, because the description was too long and they were only interested in the problems anyway.

So we changed our policy and removed these unnecessary babblings. You can go straight to the problems.

Achievements

Additionally to the problems defined below, there will be secret and publicly defined achievements. These achievements will be announced during the competition.

Have fun!

Virus (1000 points)

Do you remember the politicians having a LAN party during conference sessions?

Well, this time they managed to get a virus on all of their PCs. Fortunately the great antivirus software of ESET managed to clean up the mess, but now the politicians want to know who to blame for the virus. Hence the network administrator has to find the machine which started to spread the virus.

As you may remember the network topology is a tree, and PCs are directly connected to each other. It turns out the virus can only spread to neighboring machines and by examining the logs of a machine one can determine which directly connected PC it got the virus from (or if the machine was the source of the virus itself). So after checking a machine we either find the virus source or know which connected subtree contains it.

Unfortunately politicians are not going to let the admin look into their PCs easily, as they might keep some highly sensitive information there.

After a long (and tedious) debate the politicans arrived at the following consensus: A virus hunting plan must be made to find the source of the virus with the least number of checks possible in the worst case.

The format of the virus hunting plan is already determined: it simply assigns a non-negative integer to each machine. First the PC with the highest assigned number is checked. If it was the source of the virus then the hunting is finished, otherwise the algorithm continues in the appropriate subtree, where the checked PC got the virus from. In this subtree, again, the PC with the highest assigned number is checked. This checking procedure goes on until the source of the virus is found.

The politicians want you to give them a virus hunting plan that minimizes the required number of checks in the worst case, considering every possible PC as the source of the virus.

Input

The first line of the input contains N the number of machines. Machines are numbered from 0 to N-1.

The next line describes the network tree by N-1 numbers: the ith number gives a neighbor of the (i+1)th machine (i goes from 0 to N-2, and the ith number is at most i).

Output

The output is one line with N numbers. The ith number is the integer the virus hunting pla1n assigns to the ith machine (i goes from 0 to N-1).

Note that during the virus hunting process there always must be only one maximal number assigned to the remaining possible virus sources, so the decision is uniquely determined by the plan.

You should normalize the plan in the following way: the smallest number assigned in the plan should be 0, and the highest number should be the worst case number of checks needed. (So in the worst case the assigned numbers of the consecutively checked machines always decrease by one).

Only an optimal plan is accepted.

Example input

```
7
0 1 2 3 4 3
```

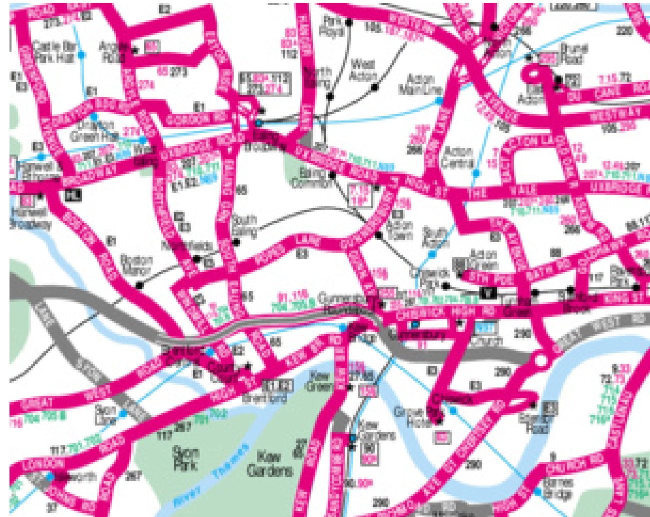
Example output

```
0 1 0 2 1 0 0
```

Scoring

Each solved input is worth 100 points.

Buses (1000 points)



There are *N* bus routes in our city, each going around a polygon shaped path with fixed schedule. We would like to know the shortest path between two given points in the city.

You can get off from a bus at any time and then wait there until some other bus gets close enough. You can take a bus if it is not more than *R* distance away.

Starting from a given *(X0,Y0)* point, the goal is to get within *R* distance to a given *(X1,Y1)* destination point.

Input

The first line of the input contains six integers: *X0 Y0 X1 Y1 R N*.

The following *N* lines describe the bus routes: the first integer is *P*, the number of vertices in the route, the second integer is the time it takes for the bus to finish one side of the polygon (a side is an edge of the polygon and each of them takes the same amount of time, no matter how long that edge is, but within the edge the bus travels at constant velocity), then *2P* integer follows, the *(X,Y)* coordinates of the vertices of the polygon in the order the bus visits them in one round. Each bus starts at the first given vertex at time 0.

Output

The first line of the output contains two numbers: the overall time to reach the destination and *K* the number of buses it takes to get there.

The following *K* lines contain three numbers: the index of the bus to take, the time when one should get on that bus and the time when one should get off.

Note that the numbers in the input are all integers, but time parameters in the output are reals. We will check the output with 0.0001 precision. The overall time should be optimal. Buses are indexed from 0 in the order they appear in the input.

Example input

```
0 0 3 0 1 3
3 1 0 2 1 1 0 1
3 3 1 2 2 0 1 0
4 1 2 2 2 1 4 1 4 2
```

Example output

```
5.5000 2
0 2.0000 4.0000
2 5.0000 5.5000
```

Scoring

Each solved input is worth 100 points.

War (1000 points)

You are the commander of an army and you have a map of the battle field. The map consists of the description of the terrain, a list of the enemy bases which have to be conquered and a list of units which are at your disposal.

The terrain is very diverse full of trenches. It is divided into NxM cells and for each cell it is known how long does it take for a unit to get through it. The terrain description consists of N lines containing M characters each. The jth character of the ith row is one of '1'...'9' and describes the time it takes for a unit to move from the [i][j] cell to one of its neighbors ([i+1][j], [i-1][j], [i][j+1] or [i][j-1]).

There are B enemy bases. The i,j coordinates of each base are given on a separate line (1 <= i <= N, 1 <= j <= M).

You have U units, each one is described on a separate line. A unit description starts with 3 integers X, Y and Q. (1 <= X <= N, 1 <= Y <= M, 1 <= Q <= B). X and Y are the coordinates of the unit. The description contains Q more integers V1, V2, ..., VQ showing which base the unit can conquer. The unit can only conquer one of these bases. (Enemy bases are numbered from 1 to B in the order they appear in the enemy base list).

Your job is to tell your boss the minimum amount of time required to conquer all the enemy bases. If it is impossible tell him so.

Input

The input contains multiple test cases. Each test case is as follows:

- The first line contains 4 integers separated by spaces N, M, B and U (B <= U).
- The next N lines contain M characters each, between '1' and '9'.
- The next B lines contain 2 integers i,j (1 <= i <= N, 1 <= j <= M).
- The next U lines start with 3 integers X, Y and Q (1 <= X <= N, 1 <= Y <= M, 1 <= Q <= B) and then contain another Q integers.

The last line of the input contains 0 0 0 0.

Output

The output should contain one line for each test case. Each line should contain one integer, the time that is required to conquer the enemy bases or -1 if it is impossible.

Notes:

- Each enemy base has to be conquered.
- A base is immediately conquered once the unit arrived there which was sent to conquer it.
- Once a unit conquered a base it must remain there, so a unit cannot conquer more than one base.
- Any number of units can be on one cell at a time.
- Units are moving independently at the same time.
- A unit can pass a cell containing an enemy base.
- The time in the output should be the time when the last base is conquered with the best strategy.

Example input

```
3 4 4 4
1231
9299
1211
1 1
1 2
1 3
1 4
3 1 2 2 3
3 2 2 1 3
3 3 3 1 2 4
3 4 4 1 2 3 4
2 2 2 2
11
11
1 1
1 2
2 1 1 2
2 2 1 2
0 0 0 0
```

Example output

```
10
-1
```

Scoring

Each solved input is worth 100 points.

Gears (1000 points)

We are in the process of building a mechanical system to rule the world. All that is left is to find a way to rotate a rod at a specific rate.

There is an engine in the system that rotates its own shaft at a fixed unit angular velocity. Help us place our gears properly to transmit the rotational motion to the required rod.

Rods can only be placed in our system in vertical position at the grid points of a horizontal NxM grid. Right now there are only two rods: the rotated shaft is at the (X0,Y0) position and the rod we want to rotate is at the (X1,Y1) position.

At most two gears can be placed on each rod: one at a lower level and another at a higher level. Gears rotate together with the rods they are attached to.

Transmission of rotation works the following way: Given two rods at distance D12 (one of which is already rotating), placing a gear with radius R1 on one and a gear with radius R2 on the other, the rotation is transmitted if $R1 + R2 = D12$ and the two gears are on the same level. The rotation speed V1 and V2 of the two rods (and of the two gears) will be such that $R1*V1 = -R2*V2$. (Note that the rotation speed has a sign and that two meshing gears rotate in different directions).

Gears on the same level must not intersect with each other, so $R1 + R2 <= D12$ must hold. Gears must not intersect with an already placed rod either, so $R1 < D12$ must hold as well (in case there is no gear on the other rod on the same level). A single rod cannot be rotated from more than one sources at the same time (so a gear must not mesh with two or more other already rotating gears).

Input

The first line of the input contains three integers: N, M and L. N, M is the size of the grid and L is the number of gear types we have.

The second line contains X0, Y0, X1, Y1 and V. (X0,Y0) are the coordinates of the rod rotating at unit speed, (X1,Y1) are the coordinates of the rod we want to rotate at speed V. V is a signed rational number given in the fractional form *A/B* where A and B are integers.

The following L lines list the gears we have: Each line contains two integers R and C, meaning that we have C number of gears with radius R.

Output

The first line of the output should contain K, the number of used gears.

The next K lines each describes a gear in the system with four integers: X, Y, R and H. (X,Y) are the coordinates of the rod the gear is attached to (must be in the (0,0), (N-1,M-1) rectangle), R is the radius and H is the level it is placed at, which is either 0 or 1.

Rods are assumed to be placed at positions where a gear is placed.

The required rotation speed must be exactly met.

You don't need to use all the gears we have.

Example input

```
5 7 4
0 0 6 1 -3/2
1 3
2 2
3 2
4 1
```

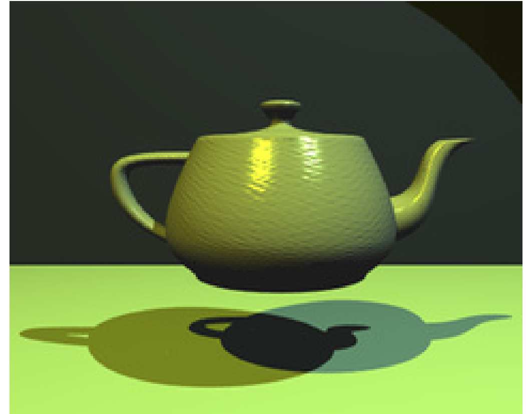
Example output

```
5
0 0 3 0
3 4 2 0
3 4 1 1
6 4 2 1
6 1 1 1
```

Scoring

Each solved input is worth 100 points.

Volume (1000 points)



You need to guess the volume of things for fun and profit!

Input

Each input consists of several images.

The images show the same 3D scene from various camera positions.

The scene contains an object with unkown volume and a cube with unit volume.

Output

The output should contain a single positive number, the volume you think the object has.

Example output

```
12.3
```

Scoring

For each submitted input you can get at most 100 points. The teams will be ranked based on how close their guess is to the real volume of the object. Then the score will be calculated the same way as the base score in the Polar tanks and Helicopter problems (See below).

You can only submit one solution in an hour for each input.

Ants (1000+1500 points)

An ant colony decided to clean up an area, help them organize their job.

The area consists of 32x32 cells. Each cell of the area has a height level between 0 and 9 units. The goal of the ants is to make the height level 0 at the entire area in the least number of iterations.

At each iteration, each ant can step one. A step can be staying at the current cell, moving to one of the four neighboring cells (in North, East, South and West directions), or moving to a neighboring cell with one unit soil, in which case at the height level at the current cell decreases and at the target increases by one.

Ants cannot step too far upwards, so a step is only possible if the target is at most one unit higher. Ants can fall down from any height just fine, so stepping downwards is allowed.

Stepping with soil is only possible if the height of the source and destination cells remain between 0 and 9. At each iteration there can be at most one ant at each cell and ants must not go outside the area.

There are special "depo" cells in the area which are connected to the storage system of the ant colony. Any number of soil units can be moved into the depos as well as any number of soil units can be taken out of it. The height of a depo cell always remains 0.

Input

The input consists of 32 lines, each containing 32 characters (and a newline). This is the map of the 32x32 cells. Each character describes one cell:

- ’0’..’9’ means a cell without ant and with height level 0..9
- ’a’..’j’ means a cell with ant and with height level 0..9
- ’-’ means a depo cell without ant
- ’+’ means a depo cell where an ant is standing right now

Output

The first line of the output contains a single integer N, the number of iterations it takes to clear the area. N does not need to be the optimal number of steps, but it must not be more than 10000.

The following N lines each describes one iteration with M characters where M is the number of ants at the area. Ants are numbered in the order they appear in the input. The ith character of an iteration line means the step the ith ant takes in that iteration:

- ’0’ means staying in place
- ’1’ means moving North
- ’2’ means moving East
- ’3’ means moving South
- ’4’ means moving West
- ’5’ means moving North with a unit soil
- ’6’ means moving East with a unit soil
- ’7’ means moving South with a unit soil
- ’8’ means moving West with a unit soil

After the N iteration the height of each cell should be 0.

Example input

```
00000000000000000000000000000000
0000a0-1000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
0000000000000c000000000000000000
0000000000000-000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

Example output

```
4
27
21
27
80
```

Sub problems

We divided the 10 inputs into 5 sub problems:

- input 1 and 2: The height of each cell is guaranteed to be either 0 or 1.
- input 3 and 4: Neighboring cells do not differ in more than 1 unit height.
- input 5 and 6: The heights are arranged so an ant can reach any cell from any other cell without moving soil.
- input 7 and 8: The heights are arranged in a way that placing an ant at any cell it can reach any other cell with height level 0 without moving soil.
- input 9 and 10: General inputs.

Ant bonus

This is a bonus part of the problem and it is worth +1500 points at maximum.

We would like to encourage you to design an input for the ants problem. Until midnight each team can submit one input design. The input is only accepted if the submitter can solve it in at most 5000 iterations.

A submission to the ant bonus problem should contain a correct input and a possible solution with at most 5000 iterations.

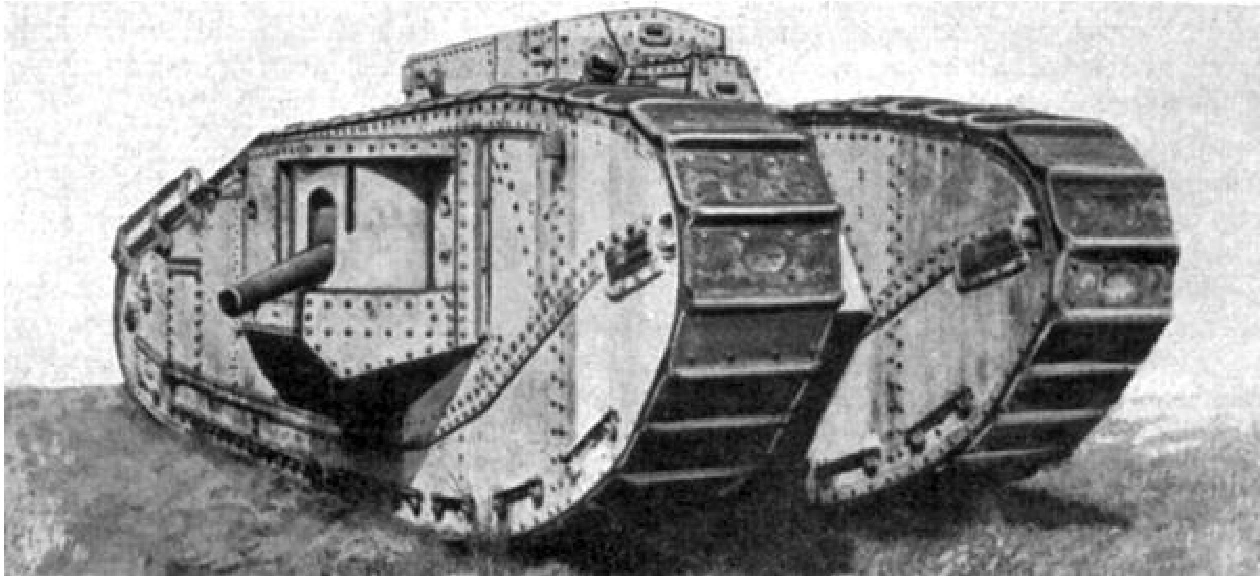
After midnight all the submitted inputs are published and teams can try to solve each other's input. (Submitting is optional, but in case you design an input you get points for solving your own submitted input as well).

Scoring

Ants is a scaled problem based on N, the iteration count of the solution. For each solved input one gets 50 + $50^{(5-N/min)/4}$ points, where *min* is the iteration count in the best correct submission.

For the ant bonus problems one can get 50 points each: 25 + $25^{(5-N/min)/4}$ (teams get points for the inputs they designed themselfs as well).

Polar tanks (2500 points)



Game flow

In this game each team controls a tank on the surface of a two dimensional planet to destroy other tanks. There are different weapons available for different price on a free market where players can sell and buy weapons for the next battle. Players earn money by causing damage to other players and by winning battles. Most weapons are ballistics with different effect on the area where they land. Weapons range from simple explosives through nukes to cluster bombs (with multiple explosives).

Throughout the 24 hours, the game is divided into six 4 hours long *campaigns*. Each *campaign* is a series of *trade-battle* pairs. During the trade phase, players can buy and sell weapons and tools that they can use in the next battles or sell in the next trade phases. Money and inventory are persistent along *battles* and *trade* phases, but are reset before a new *campaign*. Number of *trade-battle* pairs may vary from *campaign* to *campaign*, depending on how fast players finish with *trades* and *battles*.

When a phase is over, a new phase starts exactly when the server broadcasts a *phase* command.

Scoring

At the end of each campaign, teams are ordered by how much cash they have (having a lot of weapons does not matter!). Each team gets a final score depending on its place on this list (*base score*), multiplied by the *campaign factor*. This means at the end of the 24 hours, the final score of a team for this problem is calculated as: $5.555*(B1 * F1 + B2 * F2 + B3 * F3 + B4 * F4 + B5 * F5 + B6 * F6)$, where Bn is the base score the team made during campaign n and Fn is the factor of campaign n (see the tables below).

In case of a tie, the points for the first shared rank are used (e.g. if the 2nd, 3rd, 4th places are tied, all three teams get points for being the 2nd; the 5th placed team gets points for being the 5th).

If a team ends up with less money in a campaign than they initially received, they get 0 points for the campaign, regardless of their placement.

rank at the end of the campaign	base score
1st	100
2nd	80
3rd	66
4th..30th	52-(rank-4)*2

campaign number	starts at	campaign factor
1	9:00, Sat	0.2
2	13:00, Sat	0.4
3	17:00, Sat	0.6
4	21:00, Sat	0.8
5	01:00, Sun	1
6	05:00, Sun	1.5

Trade phase

Trade phases last for a campaign specific predefined time period. Teams can place offers to sell and buy certain amount of weapon for a certain price in a central database. If a sell and buy offer exactly meets, the transaction is made between the two players and both offers are removed from the database. Offers can be deleted any time from the database by the team that placed it and new offers can be placed any time.

A transaction is established if:

- the weapon name matches in both the sell and buy offer
- the quantity matches in both the sell and buy offer
- the price matches in both the sell and buy offer
- the seller does have the offered goods at the moment of the evaluation
- the buyer does have the offered amount of cash at the moment of the evaluation
- seller player ID differs from buyer player ID

Besides the teams, there may be merchants and manufacturers trading - although they won’t participate in battles. Team IDs for these players start with an underscore character.

Before a new trade phase, the offer database is purged and the phase starts with no offer. The number of offers a player may place is limited, but the player may cancel older offers to free up slots. The player may offer to sell something that is not in the player’s inventory or may place an offer for buy goods for cash the player doesn’t currently have - these offers can be placed and are broadcasted but transactions based on these offers are not established until the player happens to have the required goods or cash.

Battle

At the beginning of a new battle, a new planet is generated and the tank of each team is randomly placed on the surface in a random order specific to the given battle (*battle-specific order*). Each *battle* is divided into *turns*. In a *turn*, each player is queried for commands, one by one, in the *battle-specific* order. A player needs to react to the query and send a single command within a predefined *command timeout*, else the tank stays idle for the turn. The command is executed immediately. If the command was successful, the effect is calculated and broadcasted to all players, otherwise the active client gets an error message. In either case, the turn is over (in other words, an invalid command results in an idle turn for the player). Note: there may be a delay up to 1 second after processing the command before switching to the next player (during this delay, nothing will happen as the active player can not issue a second command).

Most often the command is to fire a ballistic missile in a given angle with a given initial velocity. Angle, velocity, the trace of the missile and the effect of the missile are all reported to all players. Different missiles act differently, but there is a common feature: after a predefined timeout, the missile initiates a self-destruct procedure (explodes wherever it is, even mid-air). NOTE: teams should carefully evaluate these information, since the value of gravity and wind speed is not reported to the teams.

Each tank has a given amount of health points (*HP*. A nearby explosion or falling down after the landscape changed below a tank decreases the *HP* of the tank. When *HP* is less then or equal to zero, the tank is considered dead - it is removed from the *battle* and is not queried anymore. Teams with dead tanks still can monitor the battle (they still get all information from the server). However, from the next *trade phase* the team can play again. The *HP* of tanks is set to 100 before each *battle*.

To avoid infinite long *battles* due to idling tanks, at the end of each turn, a battle specific predefined amount of points are subtracted from each player's *HP*, or in other words tanks are aging.

The world

The *world* is a 2 dimensional planet with a non-uniform surface that roughly ressambles a circle. All coordinates are given in a polar coordinate system with it's origin matching the center of the planet. Gravity affects any object with a constant acceleration towards the center of the planet. Because of the explosions and other effects caused by the weapons, the landscape of the planet may change during the battle: landmass may evaporate into the atmosphere or new hills can be built using dirt bombs.

There is a lower layer of the surface that is so hard that available weapons can not damage it. This layer is always at least R meters above the center.

Above the current surface of the planet there is non-stationer air. Wind blows in different directions with different speed on different altitudes. Wind may change between *turns*, but not during a *turn*. Wind affects different weapons in different ways and the effect can be modified by tools.

There is a single game server run by the organizers and client programs run by the teams.

Protocol

The protocol is text based, the game is played trough a TCP socket. The protocol consists of independent *messages*. A *message* is terminated by a newline character (\n, ASCII 0xA) and may contain letters, numbers, spaces, semicolons, colons, commas and underscore. The *message* is split into *words* by spaces. A continuous series of spaces counts as a single space. The first *word* is called the *command*. NOTE: messages sent by the server may be of any length.

To simplify protocol description, from now on, the following notations will be used:

- **I** means a signed integer number between -2^31 and +2^31-1 in decimal format;
- **S** an arbitrary string, at most 16 characters long, that may consist of upper and lowercase letters and numbers and underscores
- **T** text - an arbitrary string, at most 256 characters long, that may consist of upper and lowercase letters and numbers and underscores and spaces and tabs; always the last argument

A message from the server to the client can be of any length. A message from the client to the server shall never be longer than 512 bytes.

Setting up the connection

When a team joins the game, the client software needs to issue a *login* command first (typically after receiving an IIAM command from the server). There may be an amount of connections from a team, but there is only one connection that may send commands, and that client is called the controlling client. After connecting, the client must explicitly request control. Always the last client from a team that requested control will be in charge, all other clients become read-only. Commands from read-only clients are generally ignored but an error message is sent as a response.

Clients may join and leave (by closing the connection) any time. Any attempt for overloading the server by excess traffic may result in disqualification. If a team does not have a controlling client at the time the server sends the query for the team in a battle or during the trade phase, the tank/team keeps idling (in trade phase until the end of the phase, in battle until command timeout).

When a client connects, the following information are sent:

- campaign constants
- bank balance of each team
- inventory of the given team
- battle or trade setup (depending on which phase is going on)

Campaign constant (server -> client)

The server sends the following commands to inform the player about campaign constants.

message	description
cmdtimeout **I**	**I** is the *command timeout* in seconds
tradetime **I**	**I** describes how long a *trade phase* lasts in seconds
offers **I**	a player may have no more than **I** active offers simultaneously
slices **I**	**I** describes how many *slices* a planet has, angular coordinates will be given in terms of slices
flood **I**	if the client sends more than **I** messages in a second, the client is disconnected with EFLOOD.
phase **S**	if **S** is "B", a battle is going on, if **S** is "T", a trading phase is going on. Hint: phase command should reset all offers, landscape and player position information stored on client side. The server will update the client with the necessary information after a phase command.

Betweeen the flood and phase commands, the server also sends cash commands about players and for those items the player has in his/her inventory, a series of inv command as well.

NOTE: as a general rule, any information that is not sent by the server in messages should be measured or guessed from evaluating all available information by other messages.

cash (server -> client)

The server may send the following command multiple times to inform the player about cash states.

message	description
cash **S I**	**S** is the name of the team, **I** is the amount of cash the team has

inventory (server -> client)

The server may send the following command multiple times to inform the player about his/her own inventory.

message	description
inv **S I**	**S** is the name of the weapon or tool, **I** is the amount currently owned

battle setup (server -> client)

The server sends:

- a "planet" command
- height information for all slices

- position of each player
- HP of each player
- activete player

For more information on the format of these commands, refer to later sections. Planet command is:

message	description
planet	indicates a new planet is being described

trade setup (server -> client)

The server first sends the remaining amount of time for the trade phase:

message	description
trade **I**	**I** is the remaining time of trade phase in seconds

All currently active offers are sent to the client (format is described in a later section.

battle information (server -> client)

message	description
activate **S**	player of team **S** is the active player - when this command is received, the active player has *command timeout* seconds to issue a command (see later). There is no "deactivate" or "end of activity" command, a new "activate" command cancels the previous.
trajectory **S S I**;**I** ...	arguments are: player ID (who shot), weapon name and a series of slice;height information. Note: the server may send multiple trajectory messages for a single shot (for example in case of using cluster bombs).
height **I**;**I**	slice **I** (first number) currently has height **I** (second number)
plpos **S I**;**I**	player **S** is on slice **I** (first number) at height **I** (second argument)
HP **S I**	player **S** has HP **I**

When a new battle starts, after the phase command the server broadcasts a series of height and plpos and HP commands. Each turn starts with an activate command broadcasted and may result in multiple trajectory, height, plpos and HP messages received from the server. The turn ends with the next activate command or with a "phase T" message.

trade information (server -> client)

message	description
sell **I S S I I**	sell offer, unique offer ID **I** (first argument) by player **S** (second argument) for weapon **S** (third argument), **I** for amount and **I** (last word) for price
buy **I S S I I**	Buy offer, unique offer ID **I** (first argument) by player **S** (second argument) for weapon **S** (third argument), **I** for amount and **I** (last word) for price
trans **I I**	transaction between the two orders (sell first) is established; both orders are deleted. (NOTE: cash command is also sent about both players, inv commands are sent to the two affected players)
cancel **I**	cancel offer ID **I**

Trade phase ends when the server broadcasts a "phase B" message to indicate a new battle is starting.

trade commands (client -> server)

message	description
sell **S I I**	place a sell offer for weapon **S** (first argument), **I** for amount and **I** (last word) for price
buy **S I I**	place a buy offer for weapon **S** (first argument), **I** for amount and **I** (last word) for price
cancel **I**	cancel offer ID **I** (only if the offer is from the player)

NOTE: properly issuing any of the above commands will result in a broadcast message from the server. This means the client that initiated the action and all other clients as well will receive the command describing the effect.

battle commands (client -> server)

message	description
shoot **S I I**	shoot weapon **S** in direction **I** (in degrees) with velocity **I**.

misc commands (client -> server)

message	description
control	take over control of the player; if another client is already having control, control is taken from that client and ENOCONTROL is sent to that client immediately.
login **T**	log in with password **T**.

Error messages (server -> client)

These messages are sent only when a client sends unsuitable command to the server. A properly implemented client should never receive any of these messages from a properly implemented server. The only exception is ETOOMANY which may happen if the team runs too many properly implemented clients in parallel.

message	description
EPHASE	client tried to issue trade command in battle phase or battle command in trade phase
EFLOOD	excess flood (too many messages from the client in too short time) - disconnecting client
ENOCONTROL	client does not control the player (client is read-only)
ETOOMANYCONN	too many client connected for the team
ETOOMANYOFFER	too many offers are placed by the player
EINTERNAL **T**	internal server error (also reported on server console, the organizers should already know about it)
ECANTCANCEL **I**	the client can not cancel offer **I** (most probably the client does not own it)
ECANTSHOOT	the client could not shoot the weapon (most probably the weapon is not in inventory)
ENOTURN	last client command is ignored as this is not the team's turn
ESYNTAX **T**	syntax error - usually missing or bogus arguments for valid commands
ELOGIN	need to log in before issuing command or already logged in when a new login command is received

Announcement messages (server -> client)

These messages safely can be ignored and are implemented only to aid debugging client software.

message	description
ILEFT **I**	time left is **i** seconds (for trade in trade phase or for the active player to give a command in battle phase)
IYOUARE **S**	your current team ID is **s**
IIAM **T**	information about the game server (arguments include arbitrary number of words); first command sent by the server to the client
IEXPL **I**;**I I**	explosion at slice;height with radius **I**
IEND	campaign has ended
IGAMESTATE **S**	inform the player about new game state **S**; states are: • *lastturn*: campaign will end after end of current phase, thus this is the last phase • *paused*: game is paused; during pause sell, buy, cancel and shoot commands are ignored • *resumed*: game is resumed after a pause

Helicopters (7000 points)

It's 2010, it's **Challenge 24**, and it's time for a Networked Game! This time, it's about helicopters.

Our location is a small section of a little blue-green planet. This section is about 250 square kilometers, and happens to be conveniently bordered by an insanely high mountain chain (called the Edge Mountain). It has hills, valleys, rivers, cows, but above all, helipads.

The local people are an industrious sort, and have a lot of cargo to deliver to their fellow citizens. The preferred kind of transportation uses computer controlled helicopters - and it's your task to write the AI.

The Setup

Each team has three helicopters to control. Helicopters are controlled through TCP connections (a separate connection for each helicopter), using a text-based protocol. The helicopters will send sensor information and act on control input at a rate of **10 Hz**.

Each helicopter is equipped with a range of sensors:

- GPS (for helicopter position)
- Orientation and velocity sensors
- Fuel sensor
- Terrain radar (for surrounding terrain layout and static objects and obstacles)
- Long range radar (for detection of helicopters in a 2 km range)
- Battle radar (for tracking of helicopters within 100 meters)
- Video camera

Helicopters are controlled through four primary control inputs:

- Pitch
- Roll
- Collective
- Pedal

The Pitch and Roll inputs are together called the Cyclic. There is a hidden fifth input, the Throttle, but this is a modern type of helicopter, and the engine is controlled automatically.

For physical calculations, you can model the body of the helicopter as a brick (cuboid) that is **12 meters** long, **3.43 meters** wide and tall. Its **center of mass** is shifted from the middle, by 1.715 meters towards the nose. The **main rotor disc** has a radius of **5.14 meters**, and has its center point 1.767 meters above the center of mass.

Each team has a bank account. Fuel purchases and other costs will be deducted from this account, while prizes for successful cargo deliveries will be added to it. Teams have infinite credit, so you can go in the red as deeply as you want...

The video feed from the video camera of each helicopter may be watched on the stream server (URL specified outside this document). One team can watch only one of their three helicopter cameras at once.

Scoring

Throughout the 24 hours, the game is divided into six 4 hours long **campaigns**. In each campaign, teams begin with their bank accounts reset to 0 (however, no other stats - like map exploration percentage, number of deliveries etc. - are reset). At the end of each campaign, teams are ordered by how much money they have. Each team gets a final score depending on its place on this list (*base score*), multiplied by the *campaign factor*.

This means that at the end of 24 hours, a team's final score is calculated as: $11.111*(B1 * F1 + B2 * F2 + B3 * F3 + B4 * F4 + B5 * F5 + B6 * F6)$, where *Bn* is the base score the team achieved during campaign *n* and *Fn* is the factor of campaign (see the tables below).

In case of a tie, the points for the first shared rank are used (e.g. if the 2nd, 3rd, 4th places are tied, all three teams get points for being the 2nd; the 5th placed team gets points for being the 5th).

If in a campaign a team ends up with a non-positive amount of money, they get 0 points for the campaign, regardless of their placement.

rank at the end of the campaign	base score
1st	100
2nd	80
3rd	66
4th..30th	52-(rank-4)*2

campaign number	starts at	campaign factor
1	9:00, Sat	0.2
2	13:00, Sat	0.4
3	17:00, Sat	0.6
4	21:00, Sat	0.8
5	01:00, Sun	1
6	05:00, Sun	1.5

This means the **maximum score** a team can get from making a lot of money is (about) **5000 points**.

There are some other ways of getting more points:

- Completing **Intro Mission 1** is worth **200 points**.
- Completing **Intro Mission 2** is worth **350 points**.
- Completing **Intro Mission 3** is worth **450 points**.
- Delivering the **most packages** during the game is worth **300 points**.
- **Exploring** the most of the map is worth **300 points**.
- **Fragging** the most enemy helicopters is worth **400 points**.

This means, the **grand total** achievable score, adding up score for campaigns (5000), intro missions (1000) and achievements (1000) is **7000 points**.

Helipads

These are the places where the helicopters can land. (It may be possible to land elsewhere on the terrain, but such a landing will receive none of the benefits of helipads.)

Each helicopter begins stationary on a helipad. Should a helicopter crash, it will be moved to the nearest unoccupied helipad (after being restored to full operation - fortunately this is a pretty fast process, and takes the whole of 1 second).

Landed helicopters are refueled automatically (at a certain cost per fuel unit, which is removed from the team's bank account).

Cargo deliveries are available at certain helipads. Helicopters can take the cargo, then deliver it to the destination, which is always another helipad. Landing on the destination helipad will deliver the cargo, and automatically award the delivery prize.

Crashing

You will crash a lot, so it's useful to be aware how it exactly works.

There are three ways a helicopter can crash:

- Running out of fuel
- Touching water
- Intersecting the main rotor disc with something solid - such as the terrain, objects, or the bodies of other helicopters

Note that this list doesn't include such things as falling on the ground very fast, or maneuvering your helicopter's body into the main rotor disc of another helicopter. These things are absolutely harmless. (Destroying a helicopter of an enemy team in such a way is counted as a **frag**, and is even awarded with a little money.)

It is theoretically possible (if very difficult) to get stuck without crashing (e.g. by landing sideways on the edge of a cliff). In such cases, ask the organizers so they may reset your helicopter by a friendly smiting from the skies.

Crashing incurs a certain cost. More importantly, the cargo on the crashed helicopter is lost as well.

The Terrain

The world is described through a three-dimensional **coordinate system**. From smaller to larger values, the X coordinate goes from west to east; the Y coordinate goes from south to north; and the Z coordinate goes from the water level upwards. The measurement unit is the meter.

The world around the helicopters consists of the terrain and a few objects (such as helipads). The layout of the world is initially unknown; controlled helicopters will notify you of the things they see through their sensors.

The terrain can be modelled as a height field, where the water level is 0 (and everything at level 0 is water). Terrain is divided into **32x32 meter** sectors. For each sector, the helicopter will report **height lines** and any objects it sees.

Height lines are lines given using beginning and ending X, Y coordinate pairs, and a Z height. They give the contours of a certain height level of the terrain (that is, the terrain that lays under the given line segment has the constant height Z). They're not guaranteed to form closed polygons. However, height lines have a "direction": looking from the beginning towards the end, on the right side of the line the terrain will be higher, on the left side it will be lower.

Note that the height lines are just a representation of the terrain, not the terrain itself. It is not guaranteed that there are no higher points near a height line of a certain level, for example. It might be a good idea to keep a flight clearance of at least the difference between the levels of the two highest height line levels of the given sector.

Objects are static (unmoving) things in the world. They are modelled as cylinders, perpendicular to the X, Y plane. Objects are always given by their X, Y, Z coordinates (identifying the center point of the top disc of the cylinder), and the radius of the cylinder. The bottom of the cylinder touches the underlying terrain.

Deliveries

There is a constantly changing pool of available delivery jobs. All teams see the same pool, and helicopters from any team can take available jobs. A taken delivery job will disappear from the pool immediately, and won't reappear (no matter if it's delivered successfully or not).

Each delivery job has the following properties:

- ID (a unique positive integer, identifying the job)
- Deadline (until which the job has to be delivered, no matter when it's taken)
- Timeout (when the job disappears if not taken on by anyone)
- Cargo type
- Coordinates of the source helipad
- Coordinates of the destination helipad
- Cost of taking the job
- Prize for completing the job

To complete a delivery, a helicopter has to land on the source helipad, take the selected job, then fly to the destination helipad and land on it. One helicopter can take only one job at a time.

There are longer deliveries than the fuel tank on the helicopters can last. In this case, the helicopter will have to land on a helipad somewhere between for refueling.

The weight of cargo differs by cargo type (but is otherwise unknown). Some types of cargo may make your helicopter noticeably heavier.

Delivery prizes depend on multiple things: the kind of cargo, the deadline, but most importantly, the distance. It is much more difficult to take cargo on long distances, but the rewards are higher too.

Intro Missions

Controlling the helicopters through a complete delivery job is a challenging task. There are three easier practice tasks each team can do for extra points. Each mission can be tried any number of times, free of cost, but each team can complete one only once.

Each mission is done by a single helicopter, starting from a landed position (on any helipad). Missions are started by issuing an explicit command.

- Mission 1: get a helicopter to touch a given nearby point in space (given by world coordinates) - the center of mass has to be within 2 meters distance from the point. Deadline: 60 seconds.
- Mission 2: just like mission 1, but have to touch 5 consecutive points (each given after the previous is touched successfully), within 250 seconds.
- Mission 3: get a helicopter to a given nearby point in space, and keep the center of mass within 2 meters distance for 10 consecutive seconds. Deadline: 100 seconds.

Helicopter Control

Helicopters will use the last set of four controls they received through their corresponding TCP connection. Each control is given as a floating point number. A typical AI will wait for a 10 Hz update, calculate new controls, and reply with the new controls right away.

For the purpose of explaining the controls, we'll use a local coordinate system for a helicopter: the X axis points towards the nose, the Y axis to the left, and the Z axis up.

The first control is the **collective**. It results in an upward force (a push towards the direction of the main rotor). The control input has to be between **[0 .. 13]**.

The second control is the **pitch**. It produces a torque around the Y axis. A positive value will make the helicopter lean forward: push the nose down and the tail up. The control input has to be between **[-1 .. 1]**.

The third control is the **roll**. It produces a torque around the X axis. A positive value will make the helicopter bank right (turn around the forward axis clockwise). The control input has to be between **[-1 .. 1]**.

The fourth control is the **pedal**. It controls the force produced by the tail rotor. A positive value will push the tail to the right (and thus the nose will tend to look left). The control input has to be between **[-1 .. 1]**.

Control Protocol

In general, the control protocol is line based.

Unless otherwise indicated, all sent data fields are floating point numbers.

If a data field has a string type, the string never contains whitespace, and there are no quotes around it.

Each sent line (in either direction) consists of one or more words (strings or numbers), separated by single spaces, ending with a single unix newline.

Control Protocol: Authentication

After a TCP connection is made to the helicopter control server, the team has to log in by sending their **team password** on a single line. The server will reply with either of:

```
ERROR invalid password
```

(and will close the connection), or:

```
OK choose helicopter (1..3)
```

After the "OK" answer, the client has to send the number of the helicopter it wishes to control (1 or 2 or 3), on a single line. One helicopter can have only one open control connection at a time.

After a successful authentication, the server will begin to send updates.

Control Protocol Summary

Server to Client

POS *x y z* V *vx vy vz* QUAT *qx qy qz qw* FUEL *fuel*
CRASH
RADAR *type dx dy dz*
PROXI *type dx dy dz* V *vx vy vz* QUAT *qx qy qz qw*
TAKEOFF
LANDING
SECTOR *sx sy heightlines objects*
HL *z x1 y1 z2 y2*
OBJ *type x y z r*
JOB *id deadline cargo fx fy fz tx ty tz cost prize timeout*
TAKEN *id*
TIMEOUT *id*
FAIL
WIN
NOCARGO
CARGO *id deadline cargo fx fy fz tx ty tz cost prize*
FREESTYLE
MISSION 1 *tx ty tz*
MISSION 2 *tx ty tz*
MISSION 3 *tx ty tz countdown*
ACCOMPLISHED

Client to Server

collective pitch roll pedal
TAKE *id*
JETTISON
MISSION *number*
FREESTYLE

Control Protocol: Server to Client: Helicopter

The server sends updates at 10 Hz. Each such update consists of 0 or more "optional" lines, and a single `POS` line at the end, which is always sent.

`POS` `x y z V vx vy vz QUAT qx qy qz qw FUEL fuel`

- *x y z* is the position of the helicopter's center of mass, in world coordinates.
- *vx vy vz* is the velocity vector of the helicopter, in world coordinates.
- *qx qy qz qw* give the orientation of the helicopter, as a quaternion (see appendix)
- *fuel* is the amount of remaining fuel.

Again, the **POS** line is always sent as the final line of a 10 Hz update - so a client update of the controls should follow.

Lines below in this section are all "optional" lines.

`CRASH`

The helicopter is in a crashed state. It will remain so for a few cycles (the **CRASH** line will always be sent in these cycles). Control input is ignored for crashed helicopters.

`RADAR` `type dx dy dz`

Detection report from the long range radar. Such a report will be sent for each detected helicopter once a second.

- *type* is a string giving the type of detected object, and is always `HELI`.
- *dx dy dz* give the relative coordinates of the detected helicopter (in world space).

`PROXI` `type dx dy dz V vx vy vz QUAT qx qy qz qw`

Detection report from the battle (proximity) radar. Such a report will be sent for each detected helicopter in every cycle.

- *type* is a string giving the type of detected object, and is always `HELI`.
- *dx dy dz* give the relative coordinates (in world space) of the detected helicopter's center of mass.
- *vx vy vz* give the velocity vector (in world space).
- *qx qy qz qw* give the orientation, as a quaternion.

The long range and battle radars are independent. If a helicopter is in range for the battle radar, long range reports will still be sent once a second.

The proximity radar can track up to 8 helicopters at once.

`TAKEOFF`

The helicopter is now in flight.

`LANDING`

The helicopter has landed on a helipad successfully. At the beginning of a control connection, either **TAKEOFF** or **LANDING** will be sent, as part of the first report.

Control Protocol: Server to Client: Sector Description

`SECTOR` *`sx sy heightlines objects`*

This line begins a world sector description report. These reports are sent when the helicopter first sees a terrain sector since the beginning of the control connection - thus, for each connection and each sector, the report will be sent only once.

All data fields are integers.

- *sx sy* are the sector ID. Sector ID comes from world coordinates by dividing X, Y by 32 and rounding downwards.
- *heightlines* gives the number of following **HL** lines.
- *objects* gives the number of following **OBJ** lines.

A sector description report consists of:

- 1 **SECTOR** line
- Then, *heightlines* * **HL** lines
- Then, *objects* * **OBJ** lines

HL or **OBJ** lines are never sent outside a sector description report. Other report lines are never mixed in sector description reports.

Sector descriptions stay constant for the duration of the contest (so reports for the same sectors should be identical).

`HL` *`z x1 y1 z2 y2`*

This is a height line or contour line, as part of a sector description.

- *z* is an integer giving the height of the reported height line
- *x1 y1* give the beginning point
- *x2 y2* give the end point

`OBJ` *type x y z r*

This is a static, unmoving object, as part of a sector description.

- *type* is a string, giving the type of static object. Helipads have the type of `landingpad`. There are other types, which should be considered obstacles.
- *x y z* is the top center point of the object cylinder, in world coordinates
- *r* is the radius of the object cylinder

Control Protocol: Server to Client: Delivery

`JOB` *id deadline cargo fx fy fz tx ty tz cost prize timeout*

A new delivery job is available. On a new connection, such a line will be sent for all currently available jobs as part of the first report.

- *id* is an integer, and identifies the job.
- *deadline* is an integer, and gives the number of seconds until the job has to be delivered (so the deadline is at `now` + *deadline* seconds).
- *cargo* is a string, and identifies the cargo type.
- *fx fy fz* give the location of the helipad where the delivery job is available, in world coordinates.
- *tx ty tz* give the location of the destination helipad, in world coordinates.
- *cost* is the cost for taking on the job.
- *prize* is the prize for completing the job.
- *timeout* is an integer that gives the number of seconds until the job disappears if not taken by anyone.

`TAKEN` *id*

A delivery job has been taken, and is no longer available.

- *id* is an integer, and identifies the job.

`TIMEOUT` *id*

A delivery job has timed out, and is no longer available.

- *id* is an integer, and identifies the job.

`FAIL`

The controlled helicopter has failed the delivery job (because of a timeout, or maybe because it crashed).

`WIN`

The controlled helicopter has completed the delivery successfully.

`NOCARGO`

The helicopter is not carrying any cargo.

`CARGO` *`id deadline cargo fx fy fz tx ty tz cost prize`*

The helicopter is carrying cargo. *id* identifies the delivery job. All given data is just like with the **JOB** line.

At the beginning of the control connection, either **CARGO** or **NOCARGO** will be sent, as part of the first report.

Control Protocol: Server to Client: Intro Missions

`FREESTYLE`

The helicopter has no active mission.

`MISSION 1` *`tx ty tz`*

Intro mission 1 is active.

- *tx ty tz* give the position of the target point (in world coordinates).

`MISSION 2` *`tx ty tz`*

Intro mission 2 is active.

- *tx ty tz* give the position of the next target point (in world coordinates).

`MISSION 3` *`tx ty tz countdown`*

Intro mission 3 is active.

- *tx ty tz* give the position of the target point (in world coordinates).
- *countdown* is an integer, and indicates the number of remaining cycles for that the helicopter has to stay near the target (so it starts at 100 for 10 seconds, and goes to 0, when the mission is completed).

`ACCOMPLISHED`

The helicopter has accomplished the selected intro mission.

Control Protocol: Client to Server

Helicopter controls are updated on sent control lines. When the client sends no control line in a cycle, the controls stay in the previous position, except on a crash or disconnect (after which all controls are reset to zero).

`collective pitch roll pedal`

Sending four numbers on a line updates the helicopter controls.

`TAKE` *`id`*

Take on a delivery job.

- *id* is an integer, and identifies the job.

The controlled helicopter has to have no onboard cargo, and has to be landed on the proper helipad. If the job is taken successfully, the proper updates will be sent on the next cycle (i.e. **CARGO** and **TAKEN**).

`JETTISON`

Drop current cargo. The delivery job is failed and the cargo is lost.

`MISSION` *`number`*

Try intro mission.

- *number* is the intro mission number, either 1, 2, or 3.

The helicopter has to be landed on a helipad. If the mission is started successfully, a **MISSION** line will be sent on the next cycle.

`FREESTYLE`

Abort current intro mission.

Note that for each 0.1s cycle, only the last of the given **TAKE**, **JETTISON**, **MISSION**, **FREESTYLE** commands is processed (although this shouldn't cause problems in normal operation).

Appendix: Quaternions

In the coordinate system of the helicopter the *X* axis points forward, the *Y* axis to the left and the *Z* axis up.

The orientation of the helicopter with respect to the world coordinate system is represented by a unit quaternion *qx qy qz qw*. (The quaternion *(qx,qy,qz) = sin(alpha/2) u*, *qw = cos(alpha/2)* represents the rotation, where *u* is the unit length rotation axis and *alpha* is the rotation angle).

Formulas for calculating the *roll*, *pitch* and *yaw* angles of the helicopter from a quaternion:

```
roll = atan2(2*(qw*qx + qy*qz), 1-2*(qx*qx + qy*qy))
pitch = asin(2*(qw*qy - qx*qz))
yaw = atan2(2*(qw*qz + qx*qy), 1-2*(qy*qy + qz*qz))
```

Formulas for rotating a vector *v = (vx,vy,vz)* from the helicopter coordinate system to the world coordinate system:

```
t1 =  qw*qx
t2 =  qw*qy
t3 =  qw*qz
t4 = -qx*qx
t5 =  qx*qy
t6 =  qx*qz
t7 = -qy*qy
t8 =  qy*qz
t9 = -qz*qz
rotvx = 2*((t7 + t9)*vx + (t5 - t3)*vy + (t6 + t2)*vz) + vx
rotvy = 2*((t5 + t3)*vx + (t4 + t9)*vy + (t8 - t1)*vz) + vy
rotvz = 2*((t6 - t2)*vx + (t8 + t1)*vy + (t4 + t7)*vz) + vz
```

For example if *v = (1,0,0)* then *rotv* is the forward pointing unit vector in the world coordinate system.

The inverse rotation of a quaternion is *-qx -qy -qz qw*.

Appendix: Flight Dynamics

The helicopter is modelled through a rigid body. The forces applied to the body are gravity, profile drag, rotor blade drag and the control input forces.

In order to fly a helicopter one should master hovering first: keeping the helicopter at a given position in a stable way.

Starting from a helipad in a straight position, after increasing the lifting force above the helicopter weight using the collective, it will take off.

The rotor blades have some drag so the helicopter will start rotating slowly (yaw angle will increase), which can be compensated by the tail rotor using the pedal. (A small control input is enough, no need to apply full thrust). The pedal input can be used to keep the helicopter looking at a specific direction, or to turn to another direction.

The tail rotor applies a horizontal force at the tail so the helicopter will drift sideways slowly if you try to keep the direction in hovering.

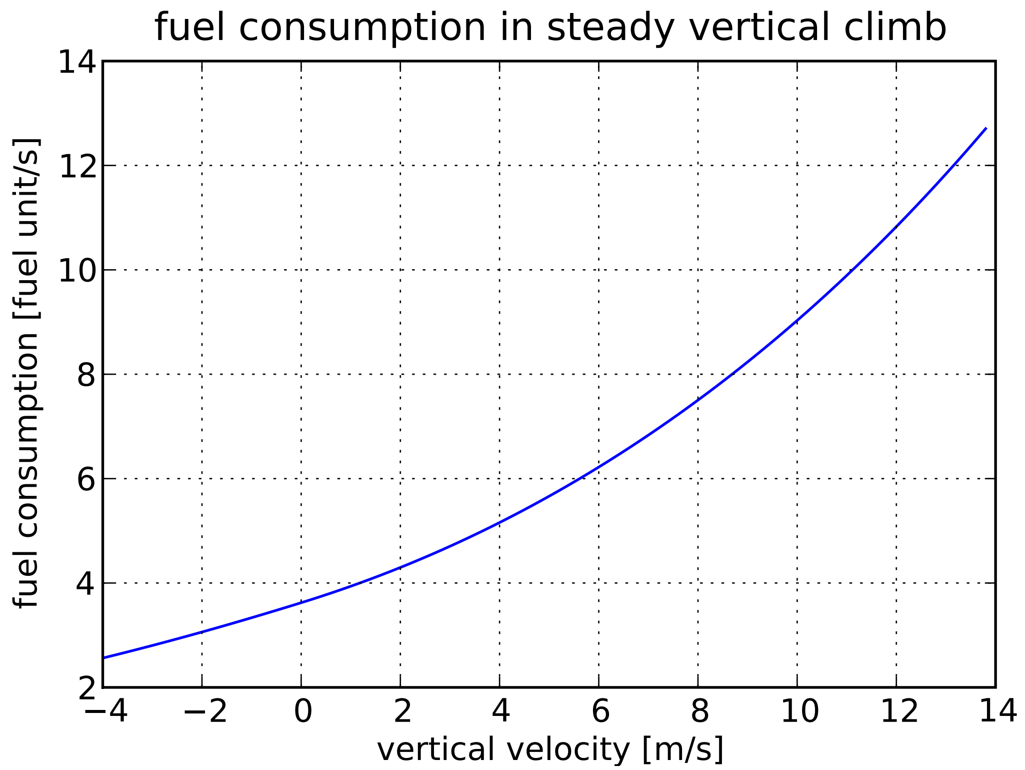
Sideways or forward drift can be compensated by increasing the roll or pitch angles a bit. Note that the control inputs are mapped to torques (or forces), so they control the acceleration not the exact angle (or position). Once the helicopter is leaning to some direction the main rotor no longer points exactly upwards so it starts pulling the helicopter sideways.

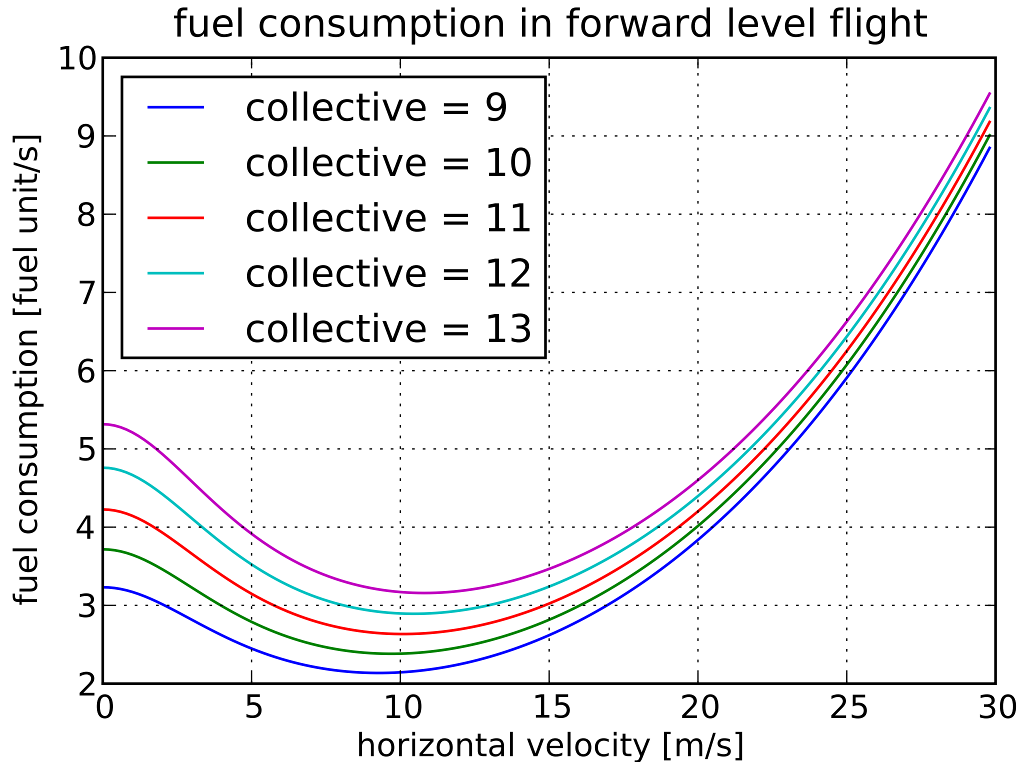
To maneuver the helicopter locally, only small control inputs are needed (except the collective which should approximately compensate the helicopter weight), and small angle changes are enough.

To move forward a longer distance, first direct the helicopter toward the destination, then increase the pitch angle and the lifting force together: the upward component of the lifting force should compensate the weight, the forward pointing component will accelerate the helicopter in forward flight.

The speed of the helicopter is limited by the profile drag force, which increases about the square of the velocity and points to the opposite direction of the velocity. The helicopter shape is streamlined in such a way that when moving forward the drag is less than when moving sideways.

Fuel consumption is a complicated function of the control inputs and the current state. See the following figures:


fuel consumption in steady vertical climb
fuel consumption [fuel unit/s]
14
12
10
8
6
4
2
−4
−2
0
2
4
6
8
10
12
14
vertical velocity [m/s]


fuel consumption in forward level flight
fuel consumption [fuel unit/s]
collective = 9
collective = 10
collective = 11
collective = 12
collective = 13
10
9
8
7
6
5
4
3
2
0
5
10
15
20
25
30
horizontal velocity [m/s]

Appendix: Flight Control

For those who have no idea how to approach the helicopter motion control problem we give a hint. This is not guaranteed to be the best solution though.

If the position *x* should be kept at some required value *r* and the control input is the acceleration (or force) *a*, then a simple PD controller is sufficient. At each iteration the control input is calculated as

```
e = r - x
delta_e = e - prev_e
prev_e = e

a = kp*e + kd*delta_e
```

for some non-negative *kp* and *kd* constants.

e is the error term, *delta_e* is the error change rate. Since *r* is fixed *delta_e* can be replaced by the change rate of -*x*, which is -velocity (in case the velocity is already known).

For example such a controller is enough to keep the helicopter at a given angle around the *z* axis using the pedal input. Of course the pedal control affects other parts of the dynamics as well, not just the yaw angle, but if the side effects are small, then this simple approach might work.